Application Note

Creating Platform Support Packages

Ralph Bodenner, Director of Product Development
Impulse Accelerated Technologies, Inc.

Overview

The CoDeveloper environment for Impulse C supports a wide range of FPGA-based hardware/software platforms for embedded systems and high-performance computing. Each of these platforms is described by a Platform Support Package (PSP), which serves two purposes:

1. To specify platform capabilities so the Impulse tools can generate appropriate hardware descriptions and hardware/software interfaces
2. To extend the CoDeveloper environment to interact with third-party development tools

This document describes how to add support for a new platform to Impulse C using the PSP infrastructure.

Revision History

Date	Revision
August, 2011	Added co_memory level 2 update
July 6, 2007	Fixed broken document reference
May 1, 2007	Initial version

FPGA Platforms

Through Platform Support Packages, Impulse C allows the programmer to target a wide range of FPGA-based platforms. These platforms may include an embedded processor, an operating system, hardware and software libraries, and many different input/output interfaces, as well as the FPGA device itself. A platform may be associated with one particular hardware system, such as the XtremeData XD1000 high-performance computing platform, or may cover a broad family of development boards and devices, as with the “Xilinx Virtex-4 APU (VHDL)” Platform Support Package. System-level design tools, such as Nallatech’s DIMEtalk software, can also be supported from Impulse C using Platform Support Packages, allowing Impulse C-generated hardware modules to be plugged in to a multi-component system.

The Impulse C compiler uses a PSP to generate files that map Impulse C code to the chosen platform. A PSP specifies how co_stream, co_memory, and other I/O channels are connected to the platform’s hardware I/O devices. A co_memory, for example, might be connected to an SDRAM controller core, or a co_stream connected to a system bus.

Similarly, a PSP can specify how software drivers for those same interfaces are created by the Impulse C compiler. Other platform implementation details can be specified using a PSP, as well—from how C arrays are turned into HDL to what logic is used to implement floating-point operations.

Platform Support Package Files

A Platform Support Package is a heterogeneous, hierarchical collection of files that describe a hardware/software platform to the Impulse C compiler tools. The structure of a PSP is defined using Extensible Markup Language (XML) files, according to formats and conventions specified by Impulse. Many other types of files can be contained in a PSP, including (but not limited to):

- Tcl scripts that generate HDL or C code
- HDL libraries
- Encrypted netlists describing IP cores
- C source files implementing software drivers
- Makefiles

XML File Structure

The components of a platform and their relationships to one another are defined in XML files. The platform definition (“top-level”) file refers to component definition files. For example, this XML element in a Xilinx MicroBlaze platform’s definition file refers to an OPB bus component and names the location of the XML file defining that bus:

```
<bus name="opb0" file="VHDL/Xilinx/OPB/bus.xml"/>
```

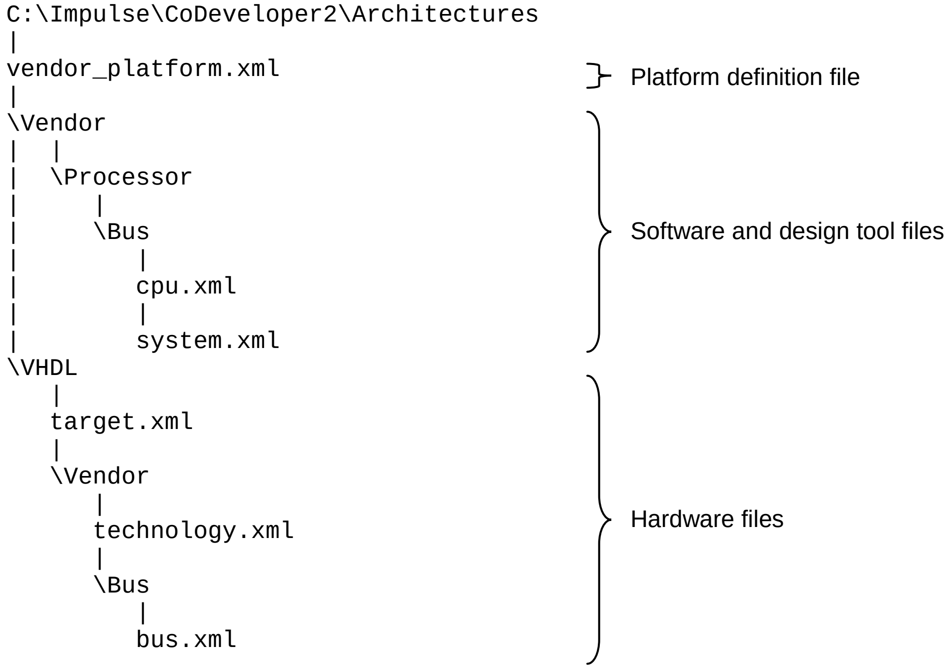
Each major component of a platform has its own XML definition file, which details the features of that component.

Filename	Component	Description
bus.xml	Bus (or other I/O link)	Connects FPGA logic to external peripherals, off-chip I/O, shared memories, and embedded or host CPUs
cpu.xml	CPU	External or embedded (hard- or soft-core) CPU
target.xml	HDL	Hardware Description Language (Verilog or VHDL) targeted by Impulse C-to-HDL compiler
technology.xml	FPGA	FPGA device family or vendor
system.xml	System	Characteristics of the entire system; a catch-all category

Figure 1: PSP component files

Directory Structure

Files that describe the PSPs in the CoDeveloper environment are arranged in a hierarchical directory structure. The root of this directory tree is the “Architectures” subdirectory of the CoDeveloper installation (for example, “C:\Impulse\CoDeveloper2”). All platform definition files must be located directly under this root directory so CoDeveloper will find them.



```
C:\Impulse\CoDeveloper2\Architectures
|
vendor_platform.xml
|
\Vendor
|  |
|  \Processor
|     |
|     \Bus
|        |
|        cpu.xml
|        |
|        system.xml
\VHDL
   |
   target.xml
   |
   \Vendor
      |
      technology.xml
      |
      \Bus
         |
         bus.xml
```

Figure 2: PSP XML file organization

The branches of the tree are organized by convention, as follows:

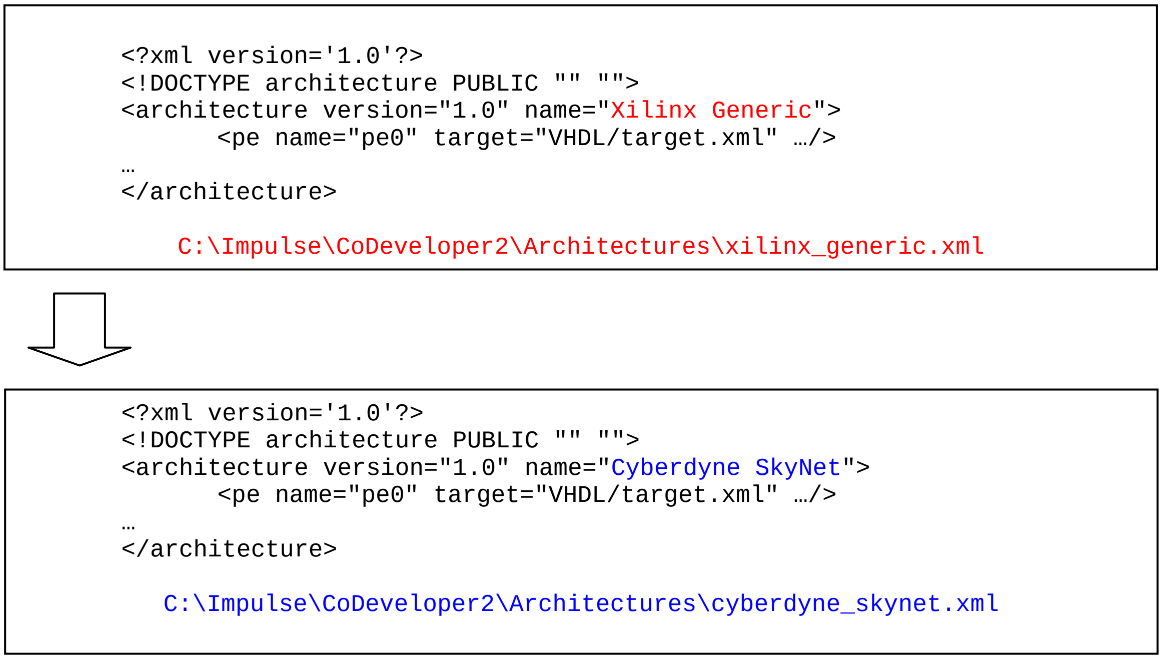
- Hardware-related files, such as "bus.xml", "target.xml", or HDL libraries, are found under either the "VHDL" or "Verilog" subdirectories, according to the HDL they are written for.
- Software- or third-party-tool-related files, such as "cpu.xml" or "export.tcl", are organized directly under "Architectures".
- Files are stored in subdirectories by vendor name, then by more specific component names. For example, software driver files for the Altera Nios II embedded CPU are located in "Architectures/Altera/NiosII".

In general, we recommend that you follow the example of the existing PSPs in organizing files.

Copy an Existing Platform

The best way to start creating a new Platform Support Package is to copy an existing one. Choose a PSP that is similar to yours and copy its platform definition file. Give the new file a unique name that includes the vendor and platform name, lowercase and separated by underscores by convention. For example, Cyberdyne Corporation's new SkyNet platform would be defined in the file "cyberdyne_skynet.xml".

Once the new file is created, a new entry should appear in the drop-down list of PSPs in an Impulse C project's options dialog in CoDeveloper



```
<?xml version='1.0'?>
<!DOCTYPE architecture PUBLIC "" "">
<architecture version="1.0" name="Xilinx Generic">
      <pe name="pe0" target="VHDL/target.xml" …/>
…
</architecture>

    C:\Impulse\CoDeveloper2\Architectures\xilinx_generic.xml
```

```
<?xml version='1.0'?>
<!DOCTYPE architecture PUBLIC "" "">
<architecture version="1.0" name="Cyberdyne SkyNet">
      <pe name="pe0" target="VHDL/target.xml" …/>
…
</architecture>

   C:\Impulse\CoDeveloper2\Architectures\cyberdyne_skynet.xml
```

Figure 3: Creating a new platform definition file

Edit the new top-level file and change the XML elements for components that are different in your platform. Depending on the platform, many elements will remain as-is, pointing to existing files. New components require specifying and creating new files.

When creating new component definition files, it may help to copy and rename a similar existing file. Once new component files are created, edit those files to define the characteristics of each component. Creating a new PSP is, in this way, a recursive process.

Where a platform does not include a component, such as a CPU, change the top-level file to refer to a “Generic” component definition file. For example, the “Lattice Generic (Verilog)” PSP does not have a CPU and is designed simply to generate standalone HDL modules appropriate to the Lattice FPGA technology, so the “system”, “exporter”, “bus”, and “proc” XML elements and attributes all point to files in the “Generic” directories. These files, which ship with CoDeveloper, act as placeholders or provide minimal functionality.

```
<?xml version='1.0'?>
<!DOCTYPE architecture PUBLIC "" "">
<architecture version="1.0" name="Lattice Generic" hdl="Verilog">
      <bus name="generic0" file="Verilog/Generic/Generic/bus.xml"/>
      <proc name="cpu0" file="Generic/cpu.xml" bus="generic0"/>
      <pe name="pe0" target="Verilog/target.xml"
technology="Verilog/Lattice/technology.xml"
system="Verilog/Generic/Generic/system.xml" cpu="cpu0" bus="generic0"/>
      <mem name="mem0" bus="generic0" alloc="malloc"/>
      <exporter file="Generic/export.tcl"/>
</architecture>
                           lattice_generic_vlog.xml
```

Figure 4: Using "Generic" files as placeholders

The following sections of this document describe the platform definition file and the component definition files in more detail.

Edit the Platform Definition File

The platform definition file (e.g., “xilinx_v4_apu.xml”) defines a single “architecture” element containing several child elements that define components of the platform.

```
<?xml version='1.0'?>
<!DOCTYPE architecture PUBLIC "" "">
<architecture version="1.0" name="Xilinx Virtex-4 APU">
       <bus name="apu0" file="VHDL/Xilinx/APU/bus.xml"/>
       <proc name="cpu0" file="Xilinx/PPC/APU/cpu.xml" bus="apu0"/>
       <pe name="PE0" target="VHDL/target.xml"
              technology="VHDL/Xilinx/v4tech.xml"
              system="VHDL/Generic/Generic/system.xml" bus="apu0"
              cpu="cpu0"/>
       <bus name="opb0" file="VHDL/Xilinx/OPB/bus.xml"/>
       <mem name="ext0" bus="opb0" alloc="ext0_alloc"/>
       <exporter file="Xilinx/EDK/export.tcl"/>
       <library name="float" file="VHDL/Xilinx/float.xml"/>
       <library name="float_fast" file="VHDL/Xilinx/float_fast.xml"/>
</architecture>
                              xilinx_v4_apu.xml
```

Figure 5: Example platform definition file

The XML elements that define the components are described below. Each element requires a “name” attribute (with the exception of “exporter”). Elements refer to one another by name to conceptually connect to one another. Each name must be unique to the PSP.

FPGA Elements

The “pe” (“*p*rocessing *e*lement”) XML element describes a platform’s FPGA.

Three attributes of the FPGA must be defined: “target”, “technology”, and “system”. Two optional elements, “cpu” and “bus”, may be defined to connect the FPGA to other components of the platform.

Attribute	Example File Path	Description
target	VHDL/target.xml	HDL targeted by the Impulse C compiler
technology	VHDL/Xilinx/technology.xml	FPGA device family or vendor
system	VHDL/Xtreme/xd1000_system.xml	Other system components

Figure 6: "pe" element required attributes

The “target” attribute should always point to “VHDL/target.xml” or “Verilog/target.xml”, depending on which HDL the Impulse C compiler should target when generating hardware for a user’s application.

The “technology” attribute refers to the FPGA device family or vendor, and how certain aspects of Impulse C code are mapped to that specific FPGA technology. Few platforms—those based on an FPGA family that Impulse C does not already support—will require a new “technology”. Point to a new “technology.xml” file if your platform will change:

- How C arrays are mapped to FPGA-local memories
- HDL library files that help implement Impulse C I/O channels (streams, registers, etc.) and some basic C operations (multipliers, etc.)

The details of the “target.xml” and “technology.xml” files are advanced topics beyond the scope of this document.

The “system” attribute refers to an XML file listing source or library files that should be included in every application targeting the platform. The “system.xml” file is a catch-all for files that aren’t associated with another part of the system, such as a bus or CPU.

Associate the FPGA with a CPU or communication layer by setting the “cpu” and “bus” attributes to the names of “proc” and “bus” elements, respectively, in the PSP’s top-level XML file.

Bus Elements

The “bus” XML element refers to a bus or other communication logic used to move data between the FPGA and the rest of the system. The file (“bus.xml”) this element refers to will define hardware libraries and a Tcl script that form the interface between Impulse C-generated I/O ports and the platform’s communication logic.

Only the “name” and “file” attributes need be defined here. Change the “file” attribute to point to a new “bus.xml” file if the platform uses a new communication mechanism to or from the FPGA.

Example file path: “Verilog/Altera/Avalon/bus.xml”

CPU Elements

Define a “proc” element if the platform supports an embedded or off-chip CPU, such as a PowerPC, Nios II, or Opteron, that will communicate with the FPGA. The “proc” element’s file will describe the C source code and scripts that constitute the software driver for the Impulse C hardware.

The “file” attribute must point to the “cpu.xml” file that describes the CPU in detail. The “bus” attribute must refer to a “bus” element by name, and conceptually connects the CPU to that communication layer.

Since a software driver often depends on the choice of communication path to the FPGA (“bus”), the “cpu.xml” file is often organized under subdirectories first by the name of the CPU and then by the name of the bus.

Example file path: “Cray/Opteron/RT/cpu.xml”

Memory Elements

Define a “mem” element if the platform will support a memory device that is shared between the FPGA and the rest of the system, such as an SRAM or SDRAM.

The “mem” element differs from most of the other elements in the platform definition file in that it does not refer to an XML file. The “name” attribute is required, as is the “bus” attribute, which should be set to the name of a “bus” element defined elsewhere in the file.

The “alloc” attribute names the software memory allocation function for this memory. Software driver code generated by the Impulse C compiler will expect this function to be defined in source code provided by the PSP. See the section “Shared Memory Allocation” for details.

Export Elements

After generating HDL and driver code for an Impulse C application, CoDeveloper can export its output files for the next stages of development, which involve using hardware synthesis and software cross-compiler tools. This export process is controlled by a Tcl script that can be extended to do a variety of tasks, from simply copying the output files to another directory, to integrating with third-party tools such as Altera’s SOPC Builder and Quartus II to assemble a complete system-on-a-chip and generate a bitfile for programming the FPGA.

If the new platform will change the export process, edit the “exporter” element to point to a new Tcl script.

Example file path: “Xilinx/EDK/export.tcl”

Library Elements

Platforms may specify any number of optional hardware libraries. These libraries define how C functions and operators are implemented in hardware by the Impulse C compiler. Floating-point operations or complex math functions, for example, can be specified in Impulse C hardware libraries.

For more information, see Impulse Application Note IATAPP108, “Using Hardware Libraries with Impulse C”, available here:

http://impulsec.com/support_appnotes.htm

Hardware/Software Interface

The most important purpose of a Platform Support Package is to define how the hardware/software interface is created. The hardware/software interface is the link between Impulse C-generated hardware in the FPGA and the rest of the system. PSPs define that link using Tcl scripts to create HDL (on the hardware side) and C source code (on the software side).

On the hardware side, compiler-generated HDL must be connected to physical hardware. Each co_stream, co_signal, co_register, etc., that is connected between an Impulse C hardware process and a software process (or left open, using co_port) will cause a set of ports to be generated by the compiler on the top-level HDL module. These ports must be connected to the actual hardware that will carry data between the FPGA and the rest of the system, whether that hardware is a system bus, a serial port, or any other I/O device.

On the software side, there are many ways to talk to the FPGA. Most commonly, the Impulse C hardware module's I/O interface registers are mapped into the memory address space of the platform's CPU. A software driver library is created that implements the Impulse C API functions (co_stream_read, co_signal_wait, etc.), reading and writing registers to exchange data and status information with the FPGA. Other implementations are possible; a co_signal could be implemented using a CPU interrupt, for example.

The next sections describe how XML, Tcl, C, and HDL files used to implement both sides of the hardware/software interface

Hardware Bus Wrappers: bus.xml and genbus.tcl

The "bus.xml" file controls how hardware is generated to connect Impulse C I/O channels (streams, etc.) to a particular hardware communication layer, such as a system bus. This file influences hardware generation in three ways:

- Defining bus characteristics used by the Impulse C compiler
- Listing HDL library files, netlists, or other files required to build bus-related hardware
- Naming a Tcl script called by the compiler to generate an HDL "bus wrapper" module

```
<?xml version='1.0'?>
<!DOCTYPE bus PUBLIC "" "">
<bus name="avalon" version="1.0">
  <param name="steering" value="true"/>
  <param name="endianess" value="little"/>
  <busgen file="VHDL/Altera/Avalon/genbus.tcl"/>
  <require file="VHDL/Altera/Avalon/avalon_if.vhd" dst="lib"
type="hdl"/>
</bus>

                   VHDL/Altera/Avalon/bus.xml
```

Figure 7: Example "bus" component definition file ("bus.xml")

The "bus.xml" file defines a single "bus" XML parent element with two attributes, "name" and "version". The "name" attribute will be used by the compiler to refer to buses of this type. (The "version" attribute is currently unused, but should be supplied.)

Bus Parameters

The "param" XML child element is used to define characteristics of the bus that influence how the Impulse C compiler generates HDL for an application's I/O interfaces. Define each parameter by creating a "param" element with "name" and "value" attributes. Three parameters can be defined:

Name	Valid Values	Default	Description
steering	true, false	None	

endianess *[sic]*	little, big	None	Endianness expected of the data
size	Any integer >0	32	Number of data bits

Figure 8: Bus parameters

The Impulse C compiler generates HDL for I/O interfaces according to these parameters. For example, if a bus with a “size” of 32 is used by an application with 64-bit streams, the compiler will instantiate a resizing adapter component that will send each 64-bit stream packets using two 32-bit bus operations. A “size” parameter with the value “*” will cause all Impulse C I/O interfaces to be generated with their requested data widths, as if the data bus were infinitely wide.

Library Files

Files containing HDL or netlist components needed to build the bus interface hardware are listed using the “require” XML child element. Each “require” element must define the following attributes:

Name	Valid Values	Description
file	UNIX-style path	File path, relative to CoDeveloper’s “Architectures” subdirectory
dst	UNIX-style path	Directory path, relative to the “Hardware build directory”, where the file should be copied during hardware generation
type	“hdl"	Type of file; must take value “hdl” (even for netlist files, etc.)

Figure 9: "require" element attributes (“bus.xml”)

When CoDeveloper generates hardware, each “required” file is copied from the “Architectures” tree to the Impulse C project’s “Hardware build directory”. Each listed file must therefore be distributed to end users with the PSP.

Bus Wrapper Generator Script: genbus.tcl

The bus wrapper is an HDL file that instantiates the top-level Impulse C module (in “*_top.vhd” or “*_top.v”) and connects each Impulse C I/O interface to the system hardware that will carry the data. This wrapper component is generated by a Tcl script.

The Impulse C compiler will look for a “busgen” XML element in the “bus.xml” file. The “busgen” element must have one attribute, “file”, whose value is the path to a Tcl script, the bus wrapper generator (named “genbus.tcl” by convention). The impulse_arch compiler tool will invoke a callback function defined in the script to generate a bus wrapper for the top-level Impulse C HDL module.

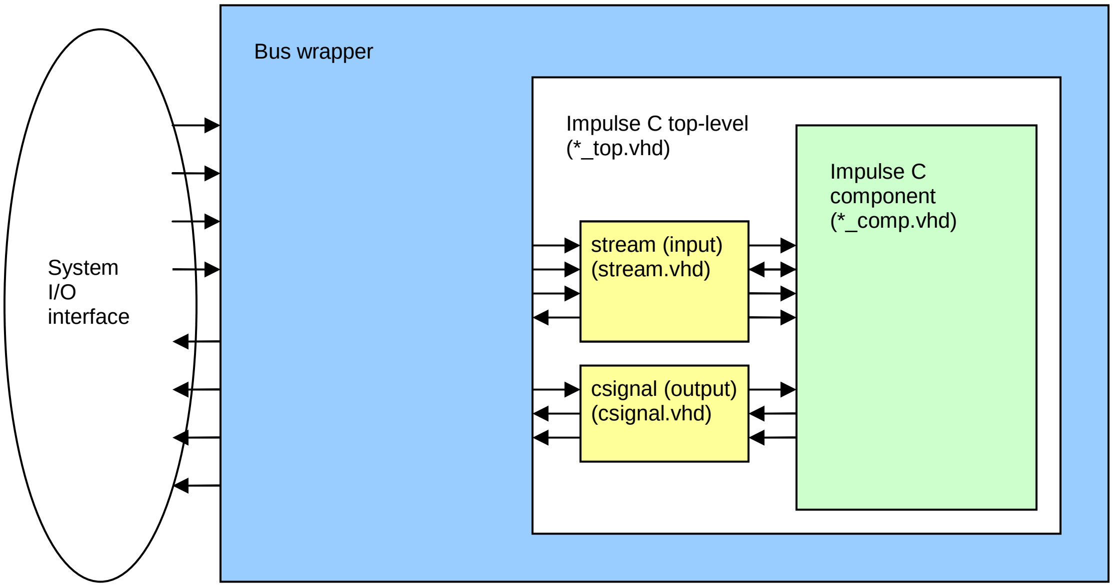


Figure 10: Bus wrapper HDL hierarchy (example)

The contents of the generated bus wrapper file are entirely up to the platform developer, who writes the "genbus.tcl" script. This script must define one Tcl callback procedure, "GenerateBUS", to which the compiler will pass data structures for all the Impulse C I/O interfaces between the application's hardware processes and the system. For complete details on the callback procedure's parameters, and Tcl procedures available to the programmer in the context of the script, see the "Impulse C HDL Interface Generation API" documentation, available for download on the Impulse Support Forum here:

http://www.impulse-support.com/forums/index.php?showtopic=342

The task of connecting Impulse C I/O interfaces to a system may be as simple as wiring ports to one another, or it may involve complex logic to arbitrate and multiplex data over a shared system bus. As before, we recommend copying an existing PSP's "genbus.tcl" script and using its code as a guide for developing a new bus interface generator.

Impulse C Hardware I/O Interfaces

Each type of Impulse C I/O interface causes a different set of ports to be generated on the top-level module. These ports, and the behavior of the interfaces for input and output, are described here.

Input Streams

Streams (co_stream) serving as inputs to the Impulse C hardware have the following ports exposed on the top-level module:

Name	Direction	Width (bits)	Description
<*stream_name*>_rdy	OUT	1	Ready to accept data
<*stream_name*>_en	IN	1	Enable write
<*stream_name*>_eos	IN	1	Write operation indicates end-of-stream (EOS)

<*stream_name*>_data	IN	Bus size	Data to be pushed onto the input buffer

Figure 11: Input stream ports

The width of the "_data" signal will be determined by the "size" parameter in the "bus" element defined in "bus.xml". If an Impulse C application uses streams that are wider than the bus size, the Impulse C compiler will automatically instantiate "stream_narrow" and "stream_widen" components between the top-level ports and the hardware processes that allow large packets to be sent as a series of bus-size transactions. Smaller-than-bus-size packets, however, are not automatically packed into fewer, bus-size transactions.

To write data to a stream, set the "_en" signal high and present the input data on the "_data" signal. When the "_rdy" signal is high on a rising clock edge, the input buffer will register the data and the "_en" input should be set low. The "_eos" signal must be low while writing data.

To close an input stream, follow the same procedure for writing data, but set the "_eos" signal high. Any data value written when the "_eos" signal is raised will cause the stream to be closed.

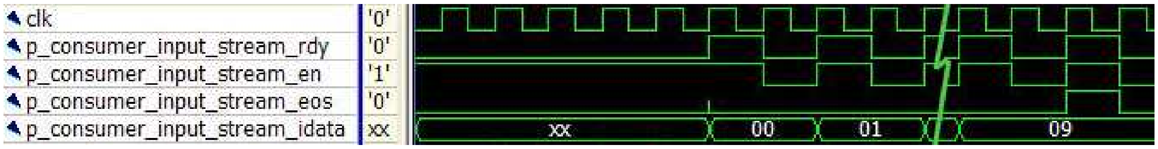

clk
p_consumer_input_stream_rdy
p_consumer_input_stream_en
p_consumer_input_stream_eos
p_consumer_input_stream_idata
'0'
'0'
'1'
'0'
xx
xx
00
01
09

Figure 12: Writing and closing an input stream

Output Streams

Streams (co_stream) serving as outputs from the Impulse C hardware have the following ports exposed on the top-level module:

Name	Direction	Width (bits)	Description
<*stream_name*>_rdy	OUT	1	Data is ready to be read
<*stream_name*>_en	IN	1	Enable read
<*stream_name*>_eos	OUT	1	Stream is closed; indicates end-of-stream (EOS)
<*stream_name*>_data	OUT	Bus size (see "Input Streams"	Data on the output buffer

Figure 13: Output stream ports

To read data from a stream, wait for the "_rdy" signal to go high, indicating data is ready on the "_data" signal. If the "_eos" signal is high, the stream has been closed and the data is not valid. Set "_en" high to remove the data from the output buffer, wait for the next rising clock edge, and set "_en" low again.

To close an output stream, flush the output buffer by performing read operations until the "_eos" signal goes high. This assumes the writer eventually closes the stream—the flush operation will otherwise run forever.

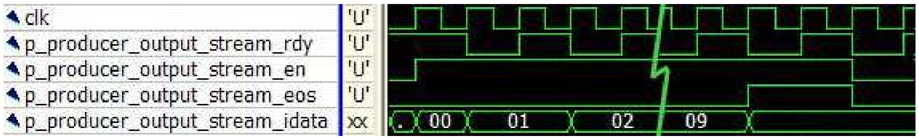

clk
p_producer_output_stream_rdy
p_producer_output_stream_en
p_producer_output_stream_eos
p_producer_output_stream_idata
'U'
'U'
'U'
'U'
xx
00
01
02
09

Figure 14: Reading an output stream until close (EOS)

Input Signals

An input signal interface is generated when co_signal_wait is called on a co_signal object in an Impulse C hardware process. The following set of ports is generated on the top-level HDL module:

Name	Direction	Width (bits)	Description
<*signal_name*>_en	IN	1	Signal is valid
<*signal_name*>_data	IN	Application-defined	Incoming data

Figure 15: Input signal ports

The width of a signal's "_data" port is determined by the user application, in the call to co_signal_create or co_signal_create_ex. A signal with a zero-bit datatype will be generated entirely without a "_data" port.

The input signal protocol is simple: on a rising clock edge when "_en" is high, the Impulse C hardware will register the data on the "_data" port (and return from the co_signal_wait function).

Output Signals

An output signal interface is generated when co_signal_post is called on a co_signal object in an Impulse C hardware process. The following set of ports is generated on the top-level HDL module:

Name	Direction	Width (bits)	Description
<*signal_name*>_en	IN	1	System is waiting on signal
<*signal_name*>_rdy	OUT	1	Output is valid
<*signal_name*>_data	OUT	Application-defined (see "Input Signals")	Output data

Figure 16: Output signal ports

The system indicates it is waiting to receive a signal from the Impulse C hardware by setting the "_en" port high. The hardware will respond by raising the "_rdy" signal and presenting the data on the "_data" port.

Input Registers

Registers (co_register) serving as inputs to the Impulse C hardware have the following ports exposed in the top-level HDL module:

Name	Direction	Width (bits)	Description
<*register_name*>_en	IN	1	Input is valid
<*register_name*>_data	IN	Application-defined	Input data

Figure 17: Input register ports

These ports make up a simple write-only register interface with an enable ("_en") signal. To write the register, raise the "_en" signal and present the data on "_data"; data will be registered on the rising clock edge.

Output Registers

Registers (co_register) serving as outputs from the Impulse C hardware have the following ports exposed in the top-level HDL module:

Name	Direction	Width (bits)	Description
<*register_name*>_value	IN	Application-defined	Output data

Figure 18: Output register ports

To read an output register, sample the "_value" port on a rising clock edge.

Impulse C Shared Memory Hardware Interface

The Impulse C co_memory shared memory interface is designed to allow Direct Memory Access (DMA) from FPGA hardware to a specific memory device over a platform-specific system bus. A PSP must implement the bus controller logic necessary to translate the co_memory requests coming from an Impulse C generated block into bus transactions for the system bus to be supported. If the platform uses a standard bus already implemented in for another platform then the bus controller can be reused. Implementations already exist for OPB, PLB, and Avalon.

As of CoDeveloper version 3, there is a new hardware interface with new features and new signals. The previous hardware interface is still supported via an adaptor component that is automatically instantiated when a shared memory based on the legacy interface is used. The legacy interface support is only available to support previous older shared memory implementations and **new shared memory implementations should not use the legacy interface**. The next section describes the new (level 2) hardware interface and the subsequent section describes the legacy (level 1) interface.

Shared Memory Hardware Interface (Level 2)

Shared memory implementations must include the following element in the corresponding bus.xml file to indicate that it uses the level 2 interface:

```
      <param name="level" value="2"/>
```

The level 2 shared memory implementations must support three types of requests: single-transaction load/store, contiguous block read/write, and multiple contiguous block read/write with stride (*slice*). Requests are initiated by Impulse C hardware in the top-level HDL file; the PSP is responsible for servicing these DMA requests with logic that connects to a system bus. Requests are transmitted to the PSP's bus wrapper via the following top-level HDL signals:

Name	Direction	Width (bits)	Description
<*mem_loc*>_idata	IN	Bus width	Data read from memory
<*mem_loc*>_addr	IN	32	Byte address of current word (FPGA memory)

<*mem_loc*>_nextaddr	IN	32	Byte address of next word (FPGA memory)
<*mem_loc*>_wri	IN	1	Enable write to FPGA memory
<*mem_loc*>_re	IN	1	Enable read from FPGA memory
<*mem_loc*>_odata	OUT	Bus width	Data to write to memory
<*mem_loc*>_ack	IN	1	Indicates last word of request
<*mem_loc*>_req	OUT	1	Indicates an active request
<*mem_loc*>_block	OUT	1	When high, indicates block operation; low indicates load/store
<*mem_loc*>_slice	OUT	1	When high, indicates a slice operation.
<*mem_loc*>_mode	OUT	1	When high, indicates write operation; low indicates read
<*mem_loc*>_base	OUT	32	Base byte address into memory
<*mem_loc*>_size	OUT	ciel(log$_2$(bus width/8))	Number of bytes per element
<*mem_loc*>_chunk	OUT	32	Number of *elements* per block (slice operations only)
<*mem_loc*>_stride	OUT	32	Number of *bytes* from the start of a block to the start of the next block in system memory. (slice operations only)
<*mem_loc*>_start	OUT	32	Base byte address in FPGA memory
<*mem_loc*>_count	OUT	32	Number of *elements* to transfer

Single transaction load/store

When <*mem_loc*>_block is low, a single element of <*mem_loc*>_size bytes is to be transferred between the system memory at address <*mem_loc*>_base and the FPGA. The FPGA address signals (<*mem_loc*>_start, <*mem_loc*>_addr, and <*mem_loc*>_nextaddr) are unused. The <*mem_loc*>_req signal is only active for one cycle to initiate the request. The <*mem_loc*>_ack signal should be asserted high when the transaction is complete and the controller will be ready to accept another request in the subsequent cycle.

For writes, the write data <*mem_loc*>_odata is valid in the cycle that <*mem_loc*>_req is asserted and remains valid until the <*mem_loc*>_ack signals is received. For reads, the <*mem_loc*>_wri signal should be asserted high during the cycle that <*mem_loc*>_idata becomes valid.

Contiguous block read/write

When <*mem_loc*>_block is high and <*mem_loc*>_slice is low, a single contiguous block of <*mem_loc*>_count elements of <*mem_loc*>_size bytes are to be transferred between the system memory starting at address <*mem_loc*>_base and the FPGA memory starting at address <*mem_loc*>_start.

Data must be transferred to/from the FPGA memory in order. The signals <*mem_loc*>_wri and <*mem_loc*>_re must be asserted for exactly one cycle per element.

Multiple block read/write (slice)

When <*mem_loc*>_block is high and <*mem_loc*>_slice is slice, <*mem_loc*>_count/<*mem_loc*>_chunk contiguous blocks of <*mem_loc*>_chunk elements of <*mem_loc*>_size bytes are to be transferred between the system memory starting at address <*mem_loc*>_base and the FPGA memory starting at address <*mem_loc*>_start. The starting address of block N+1 in system memory is equal to the starting address of block N + <*mem_loc*>_stride. The blocks in FPGA memory are contiguous. Thus, a read request gathers data blocks from system memory into a contiguous block in the FPGA memory, and a write request scatters data from a contiguous block of memory into blocks of system memory.

Data must be transferred to/from the FPGA memory in order. The signals <*mem_loc*>_wri and <*mem_loc*>_re must be asserted for exactly one cycle per element.

Upgrading from legacy interface

The level 2 interface introduces the new <*mem_loc*>_re, <*mem_loc*>_slice, <*mem_loc*>_chunk, <*mem_loc*>_stride, and <*mem_loc*>_start signals. Additionally, the existing signals <*mem_loc*>_addr and <*mem_loc*>_nextaddr have been redefined to be the *byte* address of the current element being transferred to/from the FPGA memory.

Legacy Shared Memory Hardware Interface

Read and write access is supported using contiguous block transfers or load/store (single-word) operations. Requests are initiated by Impulse C hardware in the top-level HDL file; the bus wrapper is responsible for servicing DMA requests with logic that connects to a system bus master or memory controller.

Name	Direction	Width (bits)	Description
config_en	IN	1	Base address input stream, enable write
config_eos	IN	1	Base address input stream, end-of-stream
config_data	IN	32	Base address input stream, base address value
config_rdy	OUT	1	Base address input stream, ready for write
<*mem_loc*>_idata	IN	Bus width	Data read from memory
<*mem_loc*>_addr	IN	32	Address of current word in on-chip buffer
<*mem_loc*>_nextaddr	IN	32	Address of next word in on-chip buffer
<*mem_loc*>_wri	IN	1	Enable write to local input buffer
<*mem_loc*>_odata	OUT	Bus width	Data to write to memory
<*mem_loc*>_ack	IN	1	Indicates last word of request
<*mem_loc*>_req	OUT	1	Indicates an active request
<*mem_loc*>_block	OUT	1	When high, indicates block operation; low indicates load/store
<*mem_loc*>_mode	OUT	1	When high, indicates write operation; low indicates read
<*mem_loc*>_base	OUT	32	Base byte address into memory
<*mem_loc*>_size	OUT	$\log_2$(bus width in	Number of bytes per request word

		bytes), round up to nearest integer	
<*mem_loc*>_count	OUT	32	Number of words in the request

Figure 19: Shared memory interface ports

One set of interface ports will be generated on the top-level HDL module for each memory location used by an application's hardware processes. Memory locations are specified by the PSP's platform definition file; the "name" attribute of each "mem" element is a memory location that may be passed to the co_memory_create function. Many co_memory objects can use to the same memory location by passing the same string in co_memory_create's "loc" parameter.

```
<?xml version='1.0'?>
<!DOCTYPE architecture PUBLIC "" "">
<architecture version="1.0" name="Altera Nios II">
      <bus name="avalon0" file="VHDL/Altera/Avalon/bus.xml"/>
      <mem name="heap0" bus="avalon0" alloc="malloc"/>
      <mem name="ext0" bus="avalon0" alloc="co_memory_ext0_alloc"/>
      <!-- ... -->
</architecture>
                        Architectures/altera_nios2.xml
```

```
void config(void * arg) {
      co_memory memA, memB, memCoeff;
      co_process sw0, hw0, hw1;

      memA = co_memory_create(“A”, “heap0”, 4096);
      memB = co_memory_create(“B”, “heap0”, 4096);
      memCoeff = co_memory_create(“coeff”, “ext0”, 256);

      sw0 = co_process_create(“sw”, sw_function, 3, memA, memB, memCoeff);
      hw0 = co_process_create(“hw0”, hw_function, 2, memA, memCoeff);
      hw0 = co_process_create(“hw1”, hw_function, 2, memB, memCoeff);
      //...
}
                                  UserApp_hw.c
```

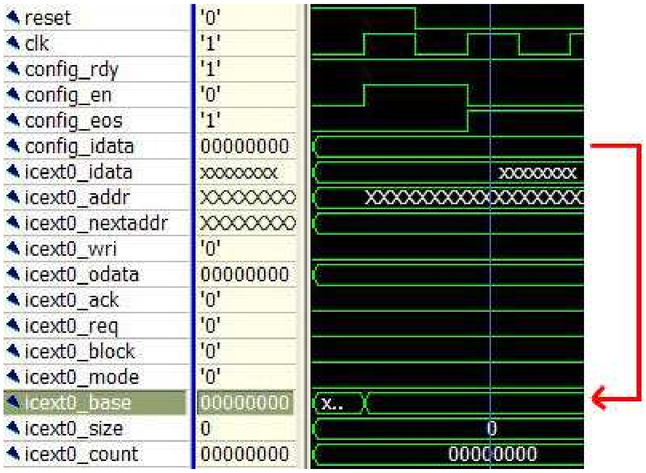
Figure 20: Memory locations

To share access to the single interface among multiple hardware processes, the Impulse C compiler will generate logic in the top-level HDL file that arbitrates between the processes using a compile-time priority mechanism. The hardware process created first in the Impulse C configuration function (by calling co_process_create) will receive first priority, followed by the other hardware processes in the order they were created. The arbitration logic ensures that DMA operations are atomic, but the user's application is responsible for synchronizing read/write access to the shared memory by the various Impulse C processes. (This synchronization can be accomplished using co_signal, for example.)

Base Address Configuration

The base address of each memory location must be configured when the Impulse C hardware is initialized, before any of the application logic will begin executing. An input stream interface will be automatically generated in the top-level HDL file for any application that uses co_memory in its hardware processes. This stream, named “config”, is used to write the base address for each memory location to an internal register.

To configure the base addresses, perform stream write operations as described in the section “Input Streams”. One data packet must be written for each memory location, in the order in which co_memory objects using the locations were created in the Impulse C application’s configuration function. The “config” stream should be closed after the last base address is written.



Base address (value: 0x0) written to “config” stream appears on the “_base” port

Figure 21: Shared memory base address configuration

In a platform with a CPU, generated C code will be responsible for stimulating the “config” stream to write the base addresses. For more details on configuring shared memory base addresses from software, see the section “Shared Memory Initialization”.

If a platform does not have a CPU, logic must be generated in the bus wrapper file that stimulates the “config” stream to write base addresses.

On-chip Buffers

All Impulse C shared memory operations use on-chip memory resources to buffer input/output data. Each on-chip memory corresponds to a C array in application code and is implemented in hardware according to the PSP. Tcl scripts referred to by “ramgen” elements in the “technology.xml” file are responsible for generating HDL code to instantiate on-chip memory resources.

Block Read Requests

A block read will be requested by the Impulse C hardware for each co_memory_readblock function call in a hardware process. The block read operation transfers a number of words from the shared memory to an on-chip buffer. After a request is initiated, one word can be transferred from the bus interface per clock cycle.

At the start of a block read request, the Impulse C hardware will raise the “_req” and “_block” signals. The “_mode” signal will be low, to indicate a read operation. The “_size” signal will contain the word size of the request (in bytes) and the “_count” signal will indicate the number of these words in the request. The “_base” signal will contain the byte offset from the shared memory’s base address; the data sent in response should be read starting from this address.

Once the system is ready to respond to the request, it should place the input data on the "_idata" port and raise the "_wri" signal to indicate that the data should be written to the on-chip buffer on the rising clock edge. The word address in the buffer where the data will be written must be placed on the "_addr" signal; for block read requests, this address always starts at zero and increments by one per clock cycle.

When "_count" number of words have been read, raise the "_ack" signal until the next rising clock edge (keeping the "_wri" signal high) to indicate the request has been fulfilled. The Impulse C hardware will then set the "_req" signal low and, "_wri" and "_ack" should be set low; the request is complete.

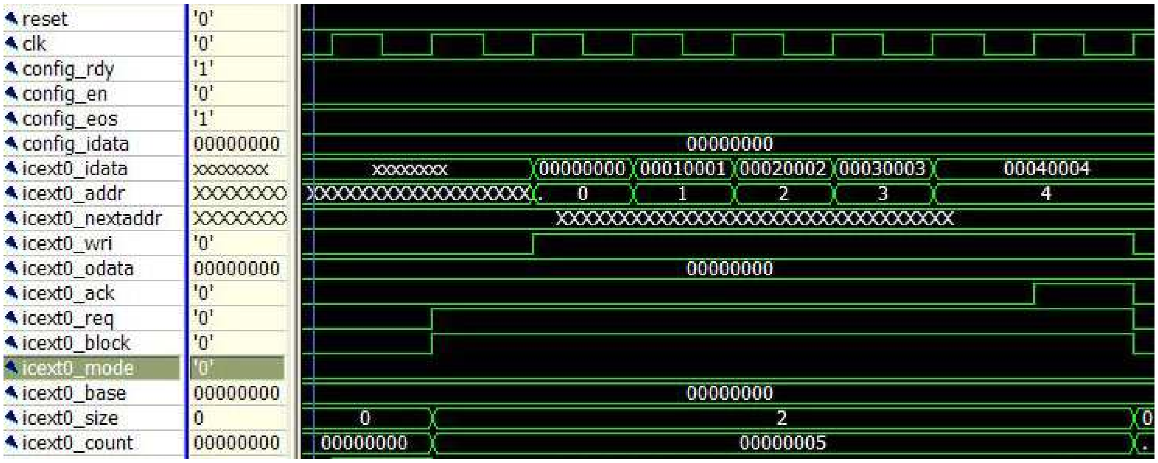


Figure 22: Shared memory block read request (16-bit words on 32-bit bus)

If the request word size is smaller than the bus data width, then the response data must be mirrored across the "_idata" signal (as in the above figure). The word size is determined by the size of the elements in the destination C array. For example, the following Impulse C code will result in a value of "2" appearing on the "_size" signal, since the destination array contains 16-bit words:

```
co_int16 A[256];
co_memory_readblock(mem, 0, A, 256*sizeof(co_int16));
```

Block Write Requests

A block write will be requested by the Impulse C hardware for each co_memory_writeblock function call in a hardware process. The block write operation transfers a number of words from an on-chip buffer to a shared memory. After a request is initiated, one word can be transferred to the bus interface per clock cycle.

At the start of a block write request, the Impulse C hardware will raise the "_req" and "_block" signals. The "_mode" signal will be high, to indicate a write operation. The "_size" signal will contain the word size of the request (in bytes) and the "_count" signal will indicate the number of these words in the request. The "_base" signal will contain the byte offset from the shared memory's base address; the data sent by the request should be written to the shared memory starting at this address.

Once the system is ready to respond to the request, it should place the word address in on the "_addr" signal and the next word address on the "_nextaddr" signal; for block write requests, the address always starts at zero and increments by one per clock cycle. The data coming from the on-chip buffer will appear on the "_odata" signal. Note that the "_wri" signal must be held low throughout a block write.

When "_count" words have been written, raise the "_ack" signal until the next rising clock edge to indicate the request has been fulfilled. The Impulse C hardware will then set the "_req" signal low, and "_ack" should be set low; the request is complete.

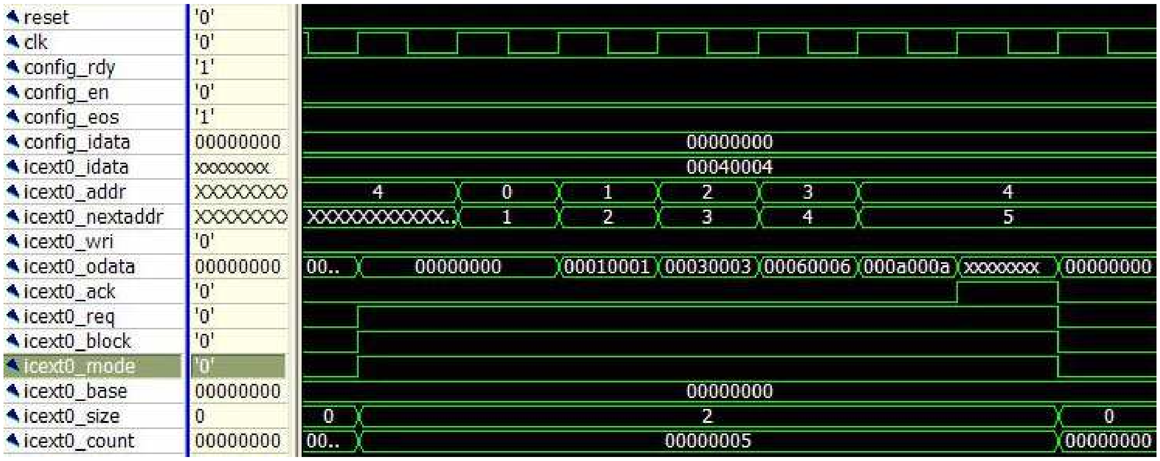


Figure 23: Shared memory block write request (16-bit words on 32-bit bus)

If the request word size is smaller than the bus data width, then the request data will be mirrored across the "_odata" signal (as in the above figure). The word size is determined by the size of the elements in the source C array. For example, the following Impulse C code will result in a value of "2" appearing on the "_size" signal, since the source array contains 16-bit words:

```
co_int16 A[256];
co_memory_writeblock(mem, 0, A, 256*sizeof(co_int16));
```

Load Word Requests

To be documented.

Store Word Requests

To be documented.

Software Drivers: cpu.xml and genlib.tcl

A PSP's software driver gives an embedded or off-chip CPU access to FPGA accelerator hardware using the Impulse C I/O interfaces (co_stream, co_signal, etc.). The driver uses low-level hardware access mechanisms, such as memory-mapped registers, interrupts, or firmware libraries, to implement the Impulse C API functions (co_stream_read, etc.) called by an application's software processes. To initialize the I/O interface objects in the application code with the addresses of the underlying hardware, the compiler generates a C source file called "co_init.c".

The source code (or precompiled library) that implements the driver is distributed with the PSP and each file is listed in an XML component definition file ("cpu.xml"). These driver files will then be compiled by the application developer, along with their application source code and the generated "co_init.c", into a software executable targeted to the platform's CPU that uses the FPGA hardware for acceleration.

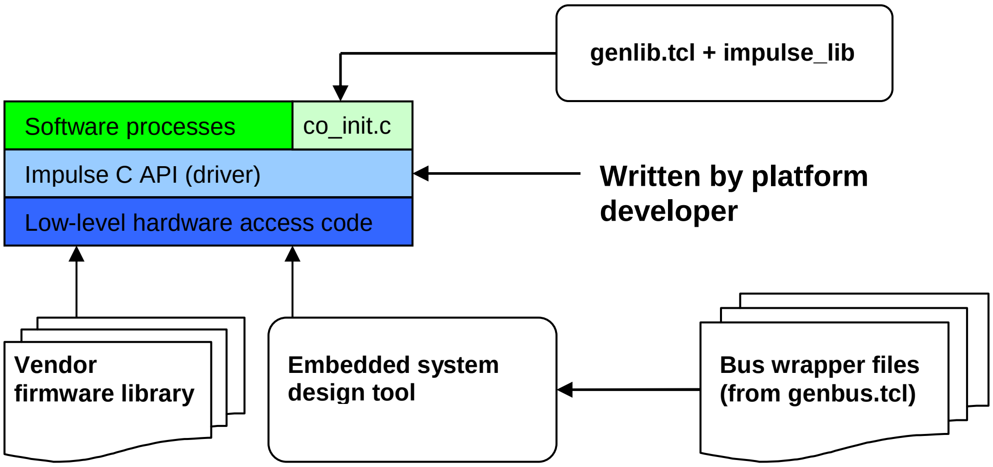

genlib.tcl + impulse_lib
Software processes
co_init.c
Impulse C API (driver)
Low-level hardware access code
Written by platform developer
Vendor firmware library
Embedded system design tool
Bus wrapper files (from genbus.tcl)

Figure 24: Implementing Impulse C software drivers

The "cpu.xml" file controls how the Impulse C compiler generates software driver code. This file lists driver source code and library files, and specifies a Tcl script ("genlib.tcl") called by the compiler to generate the "co_init.c" application initialization code.

The "cpu.xml" file defines a single "cpu" XML parent element with a "version" attribute. (The "version" attribute is currently unused, but should be supplied.) Child elements of "cpu" specify driver files and the driver generator script.

```
<?xml version='1.0'?>
<!DOCTYPE cpu PUBLIC "" "">
<cpu version="1.0">
  <libgen file="Xtreme/Opteron/genlib.tcl"/>
  <require file="Xtreme/Opteron/inc/co.h" dst="inc" type="code"/>
  <require file="Xtreme/Opteron/inc/co_if_sim.h" dst="inc" type="code"/>
  <require file="Xtreme/Opteron/inc/co_math.h" dst="inc" type="code"/>
  <require file="Xtreme/Opteron/inc/co_types.h" dst="inc" type="code"/>
  <require file="Xtreme/Opteron/inc/cosim_log.h" dst="inc" type="code"/>
  <require file="Xtreme/Opteron/inc/xd_fpga_sram_alloc.h" dst="inc"
type="code"/>
  <require file="Xtreme/Opteron/src/co_memory.c" dst="src" type="code"/>
  <require file="Xtreme/Opteron/src/co_process.c" dst="src" type="code"/>
  <require file="Xtreme/Opteron/src/co_register.c" dst="src" type="code"/>
  <require file="Xtreme/Opteron/src/co_signal.c" dst="src" type="code"/>
  <require file="Xtreme/Opteron/src/co_stream.c" dst="src" type="code"/>
  <require file="Xtreme/Opteron/src/co_type.c" dst="src" type="code"/>
  <require file="Xtreme/Opteron/src/xd_impc_rw.c" dst="src" type="code"/>
  <require file="Xtreme/Opteron/src/xd_fpga_sram_alloc.c" dst="src"
type="code"/>
  <require file="Xtreme/Opteron/linux_driver/xd_fpga.c" dst="linux_driver"
type="code"/>
  <require file="Xtreme/Opteron/linux_driver/xd_drv_iface.h"
dst="linux_driver" type="code"/>
  <require file="Xtreme/Opteron/xd-fpga/xd-fpga.cpp" dst="src" type="code"/>
  <require file="Xtreme/Opteron/xd-fpga/xd-fpga.h" dst="inc" type="code"/>
</cpu>
                    Architectures/Xtreme/Opteron/cpu.xml
```

Figure 25: Example "proc" component definition file ("cpu.xml")

Library Files

Files containing source code, precompiled libraries, Makefiles, or any other components that constitute an Impulse C I/O driver for are listed using the "require" XML child element. Each "require" element must define the following attributes:

Name	Valid Values	Description
file	UNIX-style path	File path, relative to CoDeveloper's "Architectures" subdirectory
dst	UNIX-style path	Directory path, relative to the "Software build directory", where the file should be copied during hardware generation
type	"code"	Type of file; must take value "code"

Figure 26: "require" element attributes ("cpu.xml")

When CoDeveloper generates hardware (through the "Generate HDL" menu item), each "required" file is copied from the "Architectures" tree to the Impulse C project's "Software build directory". Each listed file must therefore be distributed to end users with the PSP.

Driver Generator Script: genlib.tcl

A C source file named "co_init.c" will be created in the "Software build directory" by the Impulse C compiler when generating hardware for an application. The code in this file is responsible for configuring a driver environment for Impulse C software processes. The "co_init.c" file is generated partially by the Impulse C compiler, and partially by a PSP-specific Tcl script, "genlib.tcl".

The Impulse C compiler will look for a “libgen” XML element in the “cpu.xml” file. The “libgen” element must have one attribute, “file”, whose value is the path to a driver generator Tcl script, named “genlib.tcl” by convention. The impulse_lib compiler tool will invoke callback functions defined in the script to generate code within the “co_init.c” file.

The bulk of the “co_init.c” file consists of a co_initialize function that mirrors the portions of the Impulse C application’s configuration function relating to software processes. In several places in “co_init.c”, code can be inserted by the PSP via “genlib.tcl”. Each Tcl callback procedure (e.g., GenerateInit), is responsible for generating a separate section of the code in “co_init.c”.

The following figure illustrates the structure of “co_init.c”. Portions in black are generated automatically by the Impulse C compiler, while portions in red are generated by callbacks in “genlib.tcl”.

```
#include "co.h"

/* Architecture Includes */
#include <stdlib.h>
#include "xio.h"
#include "xparameters.h"
#include "xpseudo_asm.h"

extern void *ext0_alloc(size_t);

/* Run Procedures */
extern void Producer(co_stream);
extern void Sum(co_stream, co_stream);
extern void Consumer(co_stream);

co_architecture co_initialize(void *arg)
{
  co_stream istream;
  co_stream ostream;

  unsigned int membase;

  istream = co_stream_create("istream",
INT_TYPE(32), 4);
  ostream = co_stream_create("ostream",
INT_TYPE(32), 4);

  co_process_create("producer_process",
                    (co_function)Producer,
                    1,
                    istream);
  co_process_create("consumer_proc",
                    (co_function)Consumer,
                    1,
                    ostream);

  /* Architecture Initialization */
  mtmsr(XREG_MSR_APU_AVAILABLE);
  co_stream_attach(istream,0,HW_INPUT);
  co_stream_attach(ostream,4,HW_OUTPUT);

  return(NULL);
}
```

PSP-specific #includes
Callback: GenerateIncludes

Allocation function prototypes

Process run procedure prototypes and I/O object declarations

Variable declarations, etc.
Callback: GenerateBegin

I/O and software process object creation

Arbitrary initialization code, e.g., to set base addresses
Callback: GenerateInit

Figure 27: “co_init.c” example

For detailed documentation of the callback procedures that may be defined in the “genlib.tcl” script, and of the Tcl utility procedures available to the script programmer, see the “Impulse C Software Interface Generation API” documentation, available for download on the Impulse Support Forum here:

http://www.impulse-support.com/forums/index.php?showtopic=342

Portability and the Impulse C API

The driver software specified through “cpu.xml” is free to implement any Application Programming Interface (API) that allows software to communicate with the hardware generated for an Impulse C application. To enable Impulse C processes to be ported more easily between desktop simulation, hardware, and target platform software, however, the Impulse C API should be implemented by the PSP’s software driver. This means writing versions of the co_* functions that work in the target platform's software environment.

The Role of Embedded System Design Tools

Many embedded systems platforms depend on tools such as Xilinx Platform Studio or Altera SOPC Builder to make each I/O interface accessible to software code. These tools are used to map I/O ports to a bus slave or master interface, and to generate C code to help refer to the interfaces in software, for example as memory-mapped registers.

For example, the "Xilinx MicroBlaze OPB" PSP uses the OPB system bus to implement streams between the FPGA logic and an embedded MicroBlaze processor. The PSP's "genbus.tcl" script generates HDL and other files that connect each stream interface to an OPB bus slave. Each slave is connected to registers when a "pcore" module, comprising the generated hardware files, is connected in a Xilinx Platform Studio (XPS) project. A command within XPS then maps these registers into the embedded processor's address space. XPS creates a C header file that defines a macro for each bus slave, whose value is the base address of the slave's memory-mapped registers. The "genlib.tcl" script generates a #include directive for this header file, as well as code that associates each stream's base address (using the macro) with the corresponding co_stream object, in "co_init.c". A driver library implements the Impulse C API by reading and writing the registers using processor-specific assembly code. Finally, the Impulse C user application's software process code is compiled with the driver and "co_init.c" into an executable that is uploaded to the FPGA from within XPS.

Impulse C Software I/O Interfaces

The Impulse C API can be implemented in numerous ways, depending on the platform and what low-level hardware access functions are available on the platform CPU. This section offers suggestions for implementations using memory-mapped registers, based on a hardware model described below.

CoDeveloper ships with several PSPs whose driver source code can be used as a basis for developing a new PSP. These PSPs implement the complete Impulse C API to greater or lesser degrees; choose one that most closely matches your platform's hardware model.

The best reference for the expected behavior of each Impulse C API function is the "Function Reference" section of the "Impulse C User Guide", accessible from the Help menu in CoDeveloper.

Assigning Base Addresses

In all of the following examples, the base address of each interface is stored in a data structure (e.g., a co_stream) passed to each software process that uses that interface. This initialization is done in "co_init.c" by a call to a co_*attach function (e.g., co_stream_attach) defined in the driver. The actual addresses passed to co_*attach are determined by the "genlib.tcl" script based on what the platform developer knows about the hardware address space. On a Xilinx embedded platform, for example, "genlib.tcl" can pass as base addresses macros named after the Impulse C "pcore" and defined in a C header file generated by Xilinx Platform Studio.

Reading and Writing Streams

Hardware model: Each stream is connected to a system bus slave, which is assigned a base address. HDL components, one each for input and output streams, bridge the bus-side ports and the stream-side ports, translating bus addresses to one of up to three available registers within the HDL component. A read-only status register encodes “ready”, “error”, and “end-of-stream” signals in single bits. A write-only register on input streams can be written with any value to close the stream.

To read or write a stream, the driver code polls the status register until only the “ready” bit is high, then reads (or writes) the data register.

Register	Read/write from software?	Address Offset (bytes)
Transmit data	Write	0
Status	Read	4
End-of-stream (close)	Write	8

Figure 28: Bus-to-FPGA stream registers (example)

Register	Read/write from software?	Address Offset (bytes)
Receive data	Read	0
Status	Read	4

Figure 29: FPGA-to-bus stream registers (example)

Posting and Waiting on Signals

Hardware model: Signals are mapped through bus slaves into the CPU address space, as described above with streams. Each signal has a data and status register within the bus/signal HDL bridge component. The status register has a “ready” bit, and an “error” bit for debugging.

To post a signal from software, simply write the data register. To wait on a signal, poll the status register until only the “ready” bit is high, then read the data register.

Reading and Writing Registers

Hardware model: Each co_register is mapped to a single register in an HDL component that bridges between the bus and the register.

Reading and writing a co_register is as simple as reading/writing the memory location mapped to the register.

Shared Memory Allocation

Each memory location (see “Figure 20: Memory locations”) has its own memory allocation function, which must be defined as part of the software driver library. All memory allocation functions are modeled after the C library’s malloc function:

```
      void * malloc(size_t size);
```

That is, they take a number of bytes as an argument and return a pointer to a free memory region of the requested size. It is entirely up to the platform developer to decide how the memory allocation function manages memory.

Shared Memory Initialization

Calls to the co_memory_create function are generated by the Impulse C compiler in “co_init.c”. A pointer to the associated memory location’s allocation function is passed as an argument, so that co_memory_create can call it to allocate the internal storage used by the co_memory object. Each co_memory* function should operate on the pointer returned by the allocation function.

The Impulse C DMA interface hardware must be initialized with a base address for each memory location. The base address values (which may be the same as the pointer values stored in the co_memory structures) must be written sequentially to the automatically generated “config” stream using code generated in “co_init.c”. The “GenerateInit” Tcl callback is the usual place for generating shared memory base address initialization code.

Shared Memory Operations

Shared memory operations are easy to implement, as the platform CPU should support some way to read and write the off-chip memory. For example, the memcpy function may be supported.

Integration with Third-party Tools: export.tcl

The Platform Support Package infrastructure allows external design tools to be invoked from Tcl scripts in order to integrate CoDeveloper with the entire hardware/software design flow. After CoDeveloper has generated HDL, bus wrappers, and software drivers for an Impulse C application, the hardware and software portions can be exported for compilation down to binary files suitable for programming the FPGAs and CPUs in the system. These steps in the development process are accomplished using third-party compilers and synthesis tools; a PSP can control such tools through the “export.tcl” script.

The simplest implementation of an export script simply copies the contents of the “Hardware build directory” to the “Hardware export directory” (and similarly for software). However, arbitrary code can be written in the export script, so integration with the post-CoDeveloper design flow is limited only by the platform developer’s imagination. In fact, any of the Tcl scripts in a PSP (“genbus.tcl”, “genlib.tcl”) can be used to integrate with third-party design tools to automate the development process.

The export script (“export.tcl” by convention) is defined by the “exporter” element in the PSP’s top-level XML file, the platform definition file.

```
<?xml version='1.0'?>
<!DOCTYPE architecture PUBLIC "" "">
<architecture version="1.0" name="Xilinx Virtex-4 APU">
	<!-- ... -->
	<exporter file="Xilinx/EDK/export.tcl"/>
</architecture>
                              xilinx_v4_apu.xml
```

Figure 30: Defining an export script ("exporter")

The export script can define two Tcl callbacks, each invoked by the impulse_export compiler tool when exporting the hardware and software portions of an Impulse C application, respectively.

Export Hardware

The Tcl callback procedure CopyHardwareFiles is invoked when the "Export generated hardware" action is selected. This procedure takes three parameters:

Parameter name	Description
name	Name of the Impulse C hardware module. Corresponds to the third argument to co_architecture_create ("arch"), as called in the Impulse C application.
srcdir	UNIX-style path to the "Hardware build directory"
dstdir	UNIX-style path to the "Hardware export directory"

Figure 31: Parameters to CopyHardwareFiles

Export Software

The Tcl callback procedure CopySoftwareFiles is invoked when the "Export generated software" action is selected. This procedure takes three parameters:

Parameter name	Description
name	Name of the Impulse C hardware module. Corresponds to the third argument to co_architecture_create ("arch"), as called in the Impulse C application.
srcdir	UNIX-style path to the "Software build directory"
dstdir	UNIX-style path to the "Software export directory"

Figure 32: Parameters to CopySoftwareFiles