F

Application Note

Using Hardware Libraries with Impulse C

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

Overview

The Impulse C Compiler allows customized platform- or application-specific hardware components to be called from C code, either through the use of libraries in Platform Support Packages or through individual hardware primitive functions.

Custom hardware components may be required for a number of reasons. For example, a complex calculation may benefit from the use of a hand-crafted, low-level hardware primitive, or a custom hardware component related to system I/O may be required. Such components can be integrated into Impulse C at the language level using an open interface to the Impulse C Compiler, or via a Platform Support Package library reference. Whether your goal is to redefine the division operator, add a new math function, or provide an entire floating-point library for a new Platform Support Package, it's possible to customize the C-to-HDL translation and optimization capabilities of Impulse C for your target hardware platform.

This document describes how custom hardware components can be specified and used via the Platform Support Package interface.

Note: The Platform Support Package files and related syntax described in this document have been developed jointly by Impulse Accelerated Technologies and Green Mountain Computing Systems. Permission to use and modify these files is granted to users of the Impulse C Compiler, with the restriction that any copyright messages, where existing, are to be maintained in the modified/derivative source files.

Revision History

Date	**Changes**
2/13/2007	Initial revision
4/27/2007	Fixed description of "cycles" attribute for async HDL implementations Added "Appendix: Operator Names" Expanded section "Simulating with HDL Implementations"
6/27/2008	Added example of pipelined primitive function, clarified how component signals are declared for primitives vs. operators, fixed website link, cleaned up text

Using Custom Hardware in Applications

Impulse C applications can use custom hardware by calling C functions that the Impulse C Compiler will recognize and translate into references to external HDL components. These HDL components can be defined by the application, in an Impulse C project's source code—or by the platform, in a custom Platform Support Package.

This document describes how to define custom hardware libraries associated with a particular hardware platform, using Platform Support Packages. Please see the section of the *Impulse C User Guide* entitled "Using External HDL Hardware Functions" for information on how to define custom hardware functions at the application level. (The *Impulse C User Guide* is accessible from the Help menu in CoDeveloper.)

Units of Translation: Operators and Functions

The Impulse C Compiler supports user-defined translation of both C operators and C functions. Through the use of external interface definitions, it is possible to associate a piece of HDL with any supported C operator or user-defined C function. When such operators and functions are referenced in an application, the Impulse C Compiler schedules and instantiates the lower-level components wherever needed, as part of the optimization and HDL generation process.

To make this possible, every arithmetic operation or function available from C has an internal name associated with it. The Impulse C Compiler refers to these internal names when deciding how to schedule and generate code for an operation.

The Impulse C Compiler translates the C-language operators, such as '*' (multiplication) and '!' (logical negation), to HDL constructs. Since the C operators are overloaded for different data types, the same line of Impulse C code may infer slightly different logic, depending on the types of the operands. The compiler defines many internal operations that map to the C operators. For example, the C '/' operator translates to four different internal operations, depending on the operand context:

Arithmetic operation	C operator	Impulse C operation name
Signed integer division	/	div_s
Unsigned integer division	/	div_u
Single-precision floating-point division	/	fdiv
Double-precision floating-point division	/	fdivd

Figure 1, Division operations for the '/' operator

The Impulse C Compiler is also capable of translating C function calls into predefined blocks of hardware. For any function call in an Impulse C hardware process, the compiler will look for a *primitive* associated with that function's signature and generate references to the HDL block that implements that primitive.

Defining Libraries and Core Operations

Platform designers can specify how Impulse C is translated to custom hardware by editing the collection of human-readable files that make up an Impulse C Platform Support Package (PSP). It is possible to edit an existing PSP, but we highly recommend copying and renaming a PSP that most closely resembles your custom hardware platform, before you start making changes. All of the files associated with a PSP are organized under the “Architectures” subdirectory of the CoDeveloper installation. See Impulse Application Note IATAPP109, “Creating Platform Support Packages” (available from the Impulse website) for details on PSPs.

The set of C operators supported by Impulse C is fixed and not currently extensible through Platform Support Packages. The implementation of these operators, however, can be defined in PSPs to meet the requirements of specific platform targets.

In addition to defining basic C operators, any C function that conforms to certain requirements can be associated with a lower-level HDL implementation in the PSP. Platform Support Package developers can associate libraries of such functions with a new PSP by creating XML and HDL source files, or netlists as appropriate, that describe the functions at the hardware interface level so the compiler can schedule references to the hardware and generate appropriate component instantiations.

Note that Impulse C application developers can also define and reference new hardware functions directly in their C code, independent of a PSP, and call these functions from Impulse C hardware processes. See the topics “Using External HDL Hardware Functions” and “Hardware Primitive Functions” in the *Impulse C User Guide* for additional information.

Declaration (XML) and Definition (HDL)

Translations between C-language function/operator references and corresponding, lower-level hardware primitives are declared in XML files and defined in HDL files, in much the same way that header files and source files are used in the C language. The following sections describe the relevant XML files and required interfaces.

Core Operations: target.xml

The core C-language operators and functions supported by Impulse C include all of the operations needed to support arithmetic, logical, relational, and bitwise operators in C for signed and unsigned integer types.

Impulse strongly recommends that users wanting to extend existing Platform Support Package core operations do so using a library, rather than by directly modifying the Impulse standard PSP files. PSPs may be used to modify the core C operations, but this is rarely necessary and may result in incompatibility with newer versions of CoDeveloper, or in the overwriting of changes when new CoDeveloper versions are installed. CoDeveloper includes implementations of the core operations for both Verilog and VHDL, and these implementations are generally, but not always, shared among multiple platform targets.

The core operations are declared in the “target” XML element. This XML element is described in a file called “target.xml”, located in a subdirectory of “Architectures” named after the target HDL. For example, the file “C:\Impulse\CoDeveloper3\Architectures\VHDL\target.xml” declares translations of the core Impulse C operations to generic (non-platform-specific) VHDL.

To change how any of the core operations are implemented, first copy “target.xml” for your HDL of choice to a new file, then edit your PSP’s definition file and change all the “target” attributes of the “pe” elements to point to the new target definition file.

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

                      xilinx_generic.xml
```

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

                     cyberdyne_skynet.xml
```

Figure 2, Declaring a new target definition file

The new target definition file can now be edited to change the core operations supported by the new PSP.

Defining Extended Operations with Libraries

Extensions to the core operations are defined in libraries. A library is defined in an XML file containing a "library" element. A library is associated with a PSP by declaring the library in the PSP definition file.

Floating-point operations on Xilinx FPGAs, for example, are defined in two different libraries, found in the files "VHDL/Xilinx/float.xml" and "VHDL/Xilinx/float_fast.xml".

```
<?xml version='1.0'?>
<!DOCTYPE architecture PUBLIC "" "">
<architecture version="1.0" name="Xilinx Generic">
      …
      <library name="float" file="VHDL/Xilinx/float.xml"/>
      <library name="float_fast" file="VHDL/Xilinx/float_fast.xml"/>
</architecture>

                      xilinx generic.xml
```

Figure 3, Library declaration examples

When using a library's operations in an Impulse C application, the "name" attribute of the library must be passed in an option to the Impulse C Compiler when generating hardware. See the "Generate Options" section of the *CoDeveloper User Guide* for details on how to specify libraries used by an application.

Structure of Library and Target Files

A library or target definition file is an XML file containing one top-level parent element of type “library” or “target”, respectively. Child elements then define how each operation is to be implemented in HDL.

Two special child elements can be defined to cause blocks of raw HDL to be output in the top-level HDL file generated by CoDeveloper. Only one instance of each of these elements should appear. The text content of these elements will be output as-is in the “*_top” HDL file:

- header
- include

```
<target version="1.0">
  <header>
-- TARGET: VHDL
  </header>
  <include>
library impulse;
use impulse.components.all;
  </include>
…
                          VHDL/target.xml
```

Figure 4, “header” and “include” elements

The parent element may also contain any number of the following child elements:

- operator
- primitive
- io
- require

The “operator” and “primitive” elements describe the implementation of C operators and functions, respectively. They are described later in this document in detail and make up the bulk of a library or target definition file. One instance of either of these elements must be present for each Impulse C operation supported by the Platform Support Package.

The “io” element is not described here, as it is currently for Impulse internal use only.

Including Source Files

The “require” element describes the location of a file containing HDL source code (or other hardware descriptions, such as netlists) that implements operations. In this way, the “require” element acts somewhat like the C preprocessor directive `#include`, but in reverse: you include the “source code” in the “header file”.

```
        <library version="1.0">
          <!-- Operator and function declarations go here -->
          <require file="VHDL/Xilinx/lib/float_ll.vhd" dst="lib" type="hdl"/>
        </library>
                             VHDL/Xilinx/float.xml
```

Figure 5, Including a source file with the "require" element

The “require” element takes three attributes:

- file: The location of the file, relative to $IMPULSEC_HOME/Architectures
- dst: The subdirectory of the project’s hardware build directory (usually “hw”) where the file will be copied
- type: Must take the value “hdl”

Source files referred to with “require” must be brought into the post-CoDeveloper design flow (including FPGA synthesis), so CoDeveloper will copy them into an Impulse C project’s hardware build directory when it generates HDL for the project.

Declaring Operator and Function Implementations

Several different types of HDL implementation can be specified for operators and functions:

- Built-ins
- Macros
- VHDL functions
- Components

Each “operator” or “primitive” element in the XML library definition file describes how one operation is to be implemented. XML attributes and child elements describe the details of the implementation.

```
<?xml version='1.0'?>
<!DOCTYPE target PUBLIC "" "">
<target version="1.0">
   <operator name="dtoi" component="dtoi_ll" cycles="2" rate="1">
     <generic name="iwidth" type="out1_width"/>
     <signal name="clk" type="clock"/>
     <signal name="a" type="in1"/>
     <signal name="go" type="request" timing="late"/>
     <signal name="result" type="out1"/>
     <signal name="pipeEn" type="pipeEn"/>
   </operator>
   <primitive name="fsqrt_ll" cycles="25*" proc="sqrtf" type="component">
     <signal name="clk" type="clock"/>
     <signal name="a" type="input" carg="0"/>
     <signal name="go" type="request" timing="late"/>
     <signal name="result" type="return"/>
     <signal name="done" type="acknowledge"/>
   </primitive>
 …
</target>
                              VHDL/target.xml
```

Figure 6, Operator and function declarations

Operators

An Impulse C operator's implementation is declared by the "operator" XML element. This element requires one attribute:

- name: The predefined internal Impulse C name of the operation. Valid values of this attribute are listed in the Appendix: Operation Names.

The type of an operator's implementation is determined by which attributes are defined (see below).

If an "operator" element's optional "primary" attribute is set to "true", the operator will not appear in a subexpression in generated HDL. By default, this attribute's value is "false".

Operators are built in to the C language and will be available to user code without any need for extra header files or prototypes.

Functions

A C function's implementation is declared using the "primitive" XML element. This element requires three attributes:

- name: The internal Impulse C name of the operation
- proc: The name of the C function being implemented
- type: The type of implementation ("component" or VHDL "function")

To be called from user code, a C function with a hardware implementation must make a prototype available, for example through a header file. Functions in existing libraries, such as the standard C math library declared in math.h, can thus be implemented for hardware using the Impulse C library system and their prototypes made visible to calling code simply by #include'ing the header file as usual.

Implementation Types: Built-ins, Macros, and VHDL Functions

Built-in HDL Operators

To cause an operator to be implemented using the corresponding operator native to the chosen HDL, you must define the "builtin" attribute and set its value to "true".

The choice of HDL operator is determined by the language and is not configurable.

Example:

```
        <target version="1.0">
            <operator name="not" builtin="true"/>
        …
                                VHDL/target.xml
```

Figure 7, Declaring a built-in implementation

Macros

In a fashion similar to a C preprocessor macro, a snippet of in-line HDL code can be associated with an operator. To define such a macro, define the "macro" attribute and set its value to the string that should be printed in the generated HDL wherever the operator is used.

As with C preprocessor macros, arguments can be replaced in the macro body. The available arguments, however, are fixed by the Impulse C Compiler for each operator. Arguments are referred to in the macro body by the substrings “%0”, “%1”, and so on, in the order in which the arguments are declared in “arg” child elements. Each of these argument tokens will be replaced with the appropriate operand’s HDL identifier in the generated HDL code.

The “operator” element must contain one child element of type “arg” for each operand. Each “arg” element must define the “type” attribute, which can take one of the following values:

- “in1”, “in2”, etc.: An integer variable
- “param”: A numerical constant

If the “type” is of the “in*N*” family, the following additional attributes can also be defined and set to “true”:

- primary: If this attribute is “true”, the operand will be forced into a signal, not passed as an expression. The default value is “false”.
- signed: The variable contains a signed value (otherwise, it is considered unsigned)

If the “type” is “param”, an additional “name” attribute must be declared whose value is the internal compiler name of the argument (see existing implementations for examples).

Example:
The following XML snippet declares that the Impulse C arithmetic right-shift (“asr”) operator will be implemented as a macro using the Verilog ‘>>>’ operator. This operator takes two operands, the input (“%0”) and the shift-by value (“%1”). The input is a signed integer variable and the shift amount, known as “param” to the compiler, is an integer constant.

```
<target version="1.0">
      <operator name="asr" macro="(%0 >>> %1)">
            <arg type="in1" primary="true" signed="true"/>
            <arg type="param" name="param"/>
      </operator>
…
                        Verilog/target.xml
```

Figure 8, Declaring a macro implementation

VHDL Functions

Operators may be implemented using the VHDL `function` construct. To specify this type of implementation, define the “function” attribute and set its value to the name of the VHDL function.

The VHDL function that implements the operator must take a `std_ulogic_vector` type for each variable argument and a `natural` type for each integer constant argument, and must return a `std_ulogic_vector` type in the result. The arguments used by each operator are fixed by the compiler and not defined explicitly in the XML declaration; see an existing implementation for details.

Example Operator:
The following XML markup associates a VHDL function named "sign_ext" with the Impulse C operator "sign_extend". This operator takes two operands: the input (a variable) and the size (an integer constant) and returns a result.

```
<target version=”1.0”>
      <operator name="sign_extend" function="sign_ext"/>
…
                        VHDL/target.xml
```

Figure 9, Declaring a VHDL function implementation for an operator

```
function sign_ext(v : std_ulogic_vector; size : natural) return
std_ulogic_vector;

  function sign_ext(v : std_ulogic_vector; size : natural) return
std_ulogic_vector is
    variable res : std_ulogic_vector (size-1 downto 0);
  begin
    res(size-1 downto v'length) := (others => v(v'left));
    res(v'length-1 downto 0) := v;
    return res;
  end function;

                  VHDL/Generic/lib/impack.vhd
```

Figure 10, VHDL function implementation

Example Function:
The following XML markup associates a VHDL function named "satredu" with the C function "satredu32". The VHDL function takes two inputs, a variable and an integer constant, and returns a result.

```
<primitive name="satredu" cycles="0" proc="satredu32" type="function">
    <signal name="i1" type="input" carg="0" width="*"/>
    <signal name="i2" type="param" carg="1"/>
</primitive>

                         VHDL/target.xml
```

Figure 11, Declaring a VHDL function implementation for a C function

Note that the "cycles" attribute must take the value 0 when using a VHDL function to implement a C function.

Each "signal" child element corresponds to an input to the function, in VHDL and in C. Signals of type "input" are variable types and those of type "param" are constants. The bitwidth of each input signal (in the HDL) is given in the "width" attribute, whose value can be:

- An integer literal
- "*", indicating an arbitrary width
- Hash-notation, indicating the width value will be replaced with that of another input signal. For example: "#i1" means that the width is the same as that of signal "i1".

To use a C function with an HDL implementation in an Impulse C hardware process, declare the C function in the application's source code with a prototype.

```
        co_uint32 satredu32(co_uint32 i1, const co_uint32 i2);
```

Figure 12, Prototype of C function with VHDL function implementation

Implementation Types: Components

Operators and functions may be implemented using HDL components (VHDL "entity" or Verilog "module"). These components can be of three types:

- Combinational logic
- Registered-asynchronous logic
- Pipelined logic

These types correspond to the types of external hardware functions supported in Impulse C applications by the CO IMPLEMENTATION pragma. (See the *Impulse C User Guide* for details on use of this compiler pragma.)

The types of components are distinguished from one another in their XML definitions by the values of their "cycles" and "rate" attributes, and by which "signal" elements they contain.

The signals making up the interface to an HDL component are listed using "signal" child elements. Each signal has a "name" and a "type"; some signal types may require additional attributes. The input and output signals, corresponding to C parameters and the return value, are specified differently for operators and for primitives.

Parameter	**Operation Type**	**Signal Attributes**
0	operator	`type="in1"`
	primitive	`type="input" carg="0"`
1	operator	`type="in2"`
	primitive	`type="input" carg="1"`
Return Value	operator	`type="out1"`
	primitive	`type="return"`

Figure 13, XML attributes of input/output signal declarations

Parameterized properties, such as sizes, may be passed to components as VHDL "generic" or Verilog "parameter" types, using the "generic" element. Any parameters are specific to the Impulse C operator; see an existing implementation for examples.

```
          <operator name="dtoi" component="dtoi_ll" cycles="2" rate="1">
              <generic name="iwidth" type="out1_width"/>
              <signal name="clk" type="clock"/>
              <signal name="a" type="in1"/>
              <signal name="go" type="request" timing="late"/>
              <signal name="result" type="out1"/>
              <signal name="pipeEn" type="pipeEn"/>
          </operator>
                                VHDL/Xilinx/float.xml
```

Figure 14, Example of signal and parameter/generic declarations

Combinational Logic

Components using combinational logic are distinguished by a “cycles” attribute of 0. One signal for each input and output are the only signals declared; no clock signal is present. For example, the floating-point negation operator is implemented in a library as combinational logic:

```
          <operator name="fneg" component="fneg_ll" cycles="0">
              <signal name="a" type="in1"/>
              <signal name="result" type="out1"/>
          </operator>
          …
          <require file="VHDL/Xilinx/lib/float_ll.vhd" dst="lib" type="hdl"/>

                                VHDL/Xilinx/float.xml
```

Figure 15, Operator declaration using combinational logic

The implementation of such an operation is defined in a VHDL file that the library definition file refers to using the “require” element:

```
        entity fneg_ll is
          port (
            a: in std_ulogic_vector(31 downto 0);
            result: out std_ulogic_vector(31 downto 0));
        end fneg_ll;

        architecture fneg_ll_a of fneg_ll is
        begin
          result(31) <= a(31) xor '1';
          result(30 downto 0) <= a(30 downto 0);
        end fneg_ll_a;

                         VHDL/Xilinx/lib/float_ll.vhd
```

Figure 14, Operator implementation using combinational logic

Functions implemented using combinational logic are declared like operators, but using the “primitive” element instead of “operator”:

```
        <primitive name="fabs_ll" cycles="0" proc="fabsf" type="component">
            <signal name="a" type="input" carg="0" width="*"/>
            <signal name="result" type="return"/>
        </primitive>

                              VHDL/Xilinx/float.xml
```

Figure 16, Function declaration using combinational logic

Registered-asynchronous Logic

Operators or functions can be implemented using logic whose latency cannot be determined at compile time, for example if that latency depends on the values of the input data. The “cycles” attribute should be given using “*” notation. A “cycles” value of “25*” indicates an indeterminate latency with a minimum of 25 cycles.

```
        <operator name="fdivd" component="fdivd_ll" cycles="1*">
            <signal name="clk" type="clock"/>
            <signal name="a" type="in1"/>
            <signal name="b" type="in2"/>
            <signal name="go" type="request" timing="early"/>
            <signal name="result" type="out1"/>
            <signal name="done" type="acknowledge"/>
        </operator>

                      Architectures/VHDL/Xilinx/float.xml
```

Figure 17, Operator declaration using registered-asynchronous logic

In the implementation of a registered-asynchronous operation, certain signals are required. A “request” signal is input to start processing and an “acknowledge” signal is output to indicate completion. These signals, as well as each data input/output and a clock signal, are declared using the “signal” element.

The “timing” attribute of the “request” signal must be “early” for asynchronous components.

Pipelined Logic

A pipelined implementation is distinguished by constant values of the attributes “cycles” and “rate”, which correspond to the latency and pipeline rate of the logic implementing the operation. After receiving a signal of type “request”, output appears “cycles” clock cycles later, and subsequent outputs appear “rate” cycles after the first. The pipeline runs when the “pipeEn” signal is high, otherwise it must stall.

```
        <operator name="fmuld" component="fmuld_ll" cycles="4" rate="1">
            <signal name="clk" type="clock"/>
            <signal name="a" type="in1"/>
            <signal name="b" type="in2"/>
            <signal name="go" type="request" timing="late"/>
            <signal name="result" type="out1"/>
            <signal name="pipeEn" type="pipeEn"/>
        </operator>

                             VHDL/Xilinx/float.xml
```

Figure 18, Operator declaration using pipelined logic

```
        <primitive name="fsqrtd" cycles="30" rate="1" proc="sqrt"
        type="component">
            <signal name="clk" type="clock"/>
            <signal name="a" type="input" carg="0"/>
            <signal name="go" type="request" timing="late"/>
            <signal name="result" type="return"/>
            <signal name="pipeEn" type="pipeEn"/>
        </operator>

                              VHDL/Altera/float.xml
```

Figure 19, Function declaration using pipelined logic

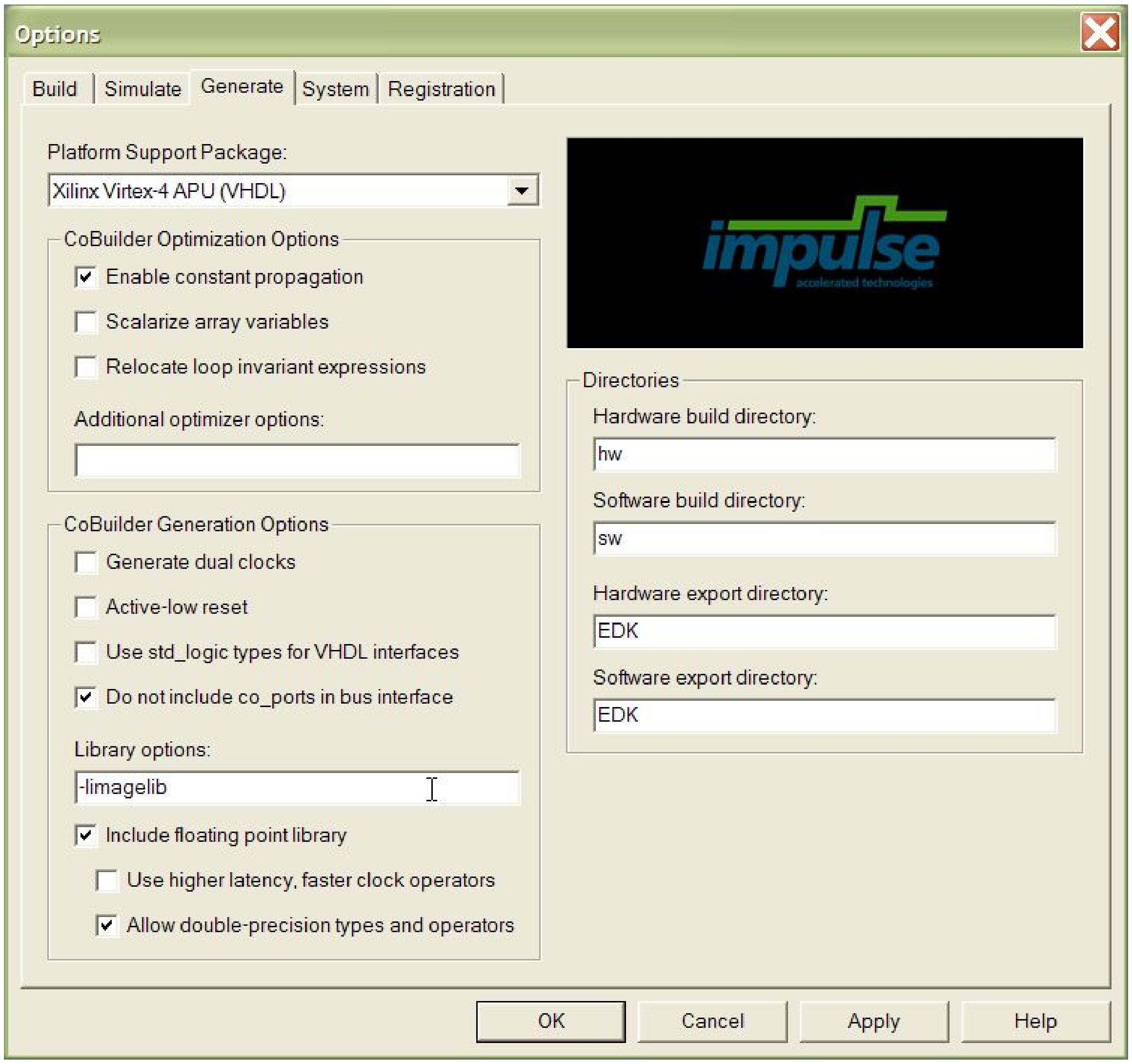
The “timing” attribute of the “request” signal can take one of two values, with the following implications for the design of the HDL component:

- early: All inputs will be registered
- late: Inputs may be combinational

Using Libraries in CoDeveloper HDL Generation

To use an HDL implementation library when generating HDL for an application in Impulse CoDeveloper, the name of the library, prefixed with “-l” (as in “library”), must be passed as an option to the Impulse C Compiler.

In CoDeveloper version 2, library options are specified in a text field on the Generate tab of the Project Options dialog (Project > Options menu). Floating-point libraries are an exception—two built-in library names, “float” and “float_fast”, are passed to the Impulse C Compiler by selecting the appropriate checkboxes for floating-point support in the project options.



Platform Support Package: Xilinx Virtex-4 APU (VHDL)

CoBuilder Optimization Options
- [✓] Enable constant propagation
- [ ] Scalarize array variables
- [ ] Relocate loop invariant expressions

Additional optimizer options:

CoBuilder Generation Options
- [ ] Generate dual clocks
- [ ] Active-low reset
- [ ] Use std_logic types for VHDL interfaces
- [✓] Do not include co_ports in bus interface

Library options: -limagelib

- [✓] Include floating point library
  - [ ] Use higher latency, faster clock operators
  - [✓] Allow double-precision types and operators

Directories

Hardware build directory: hw

Software build directory: sw

Hardware export directory: EDK

Software export directory: EDK

Figure 20, Specifying libraries in CoDeveloper version 2

In the figure above, a library named “imagelib” is being used by the application, as well as the “float” floating-point library (selected by the checkbox “Include floating-point library”).

See the section “CoBuilder Command Line Tools” in the *Impulse C User Guide,* accessible from the CoDeveloper Help menu, for details on how library options are passed to the individual Impulse C Compiler tools.

Simulating with HDL Implementations

Impulse C applications can be simulated on the Windows or Linux desktop as C programs. In such a simulation, both hardware and software processes are compiled together into an executable and run in separate threads. Any HDL implementations used in an application code do not come into play in desktop simulation—only C implementations.

Every function called in an Impulse C application must have a C implementation in order to be simulated. Together with the GCC compiler tools, the Impulse C simulation library implements all ANSI C operators and the co_* functions for the desktop simulation environment. To simulate other functions, including those defined in libraries using "primitive" XML elements, the application must provide a C-language implementation.

Impulse C code used only in desktop simulation may need to be hidden from the HDL generation process; for example, file I/O done for debugging purposes in simulation will not compile to HDL. This can be accomplished through the use of Impulse-supplied compile-time macros, such as IF_SIM, and by tagging source files containing simulation-only code in CoDeveloper. Please refer to the *Impulse C User Guide* for additional information.

For example, consider the "satredu32" C function, which has an HDL implementation defined in "VHDL/target.xml".

```
<primitive name="satredu" cycles="0" proc="satredu32" type="function">
    <signal name="i1" type="input" carg="0" width="*"/>
    <signal name="i2" type="param" carg="1"/>
</primitive>

                         VHDL/target.xml
```

Figure 21, "satredu32" library function definition

To simulate a hardware process that calls this function, write a C function that implements the operation (32-bit unsigned saturation reduction) and include the code in a source file marked as a "Desktop simulation source file" in CoDeveloper.

```
co_uint32 satredu32(co_uint32 i1, const co_uint32 i2)
{
      return (i1 >= (1<<i2)) ? ((1<<i2)-1) : i1;
}
                       UserApp/UserApp.c
```

Figure 22, C implementation of "satredu32" for desktop simulation

The above C code will not be processed by the Impulse C Compiler for HDL generation. Instead, the HDL implementation indicated by the "primitive" XML element will be used in place of calls to "satredu32".

Appendix: Operation Names

The following internal names are used by the Impulse C Compiler to refer to operations. Most compiler operations correspond to an instance of a C operator (e.g., '*') with operands of a particular datatype (e.g., unsigned integers or floating-point types). Some operations are generated for internal compiler use and do not relate directly to the user's C code; these are indicated as "internal". Finally, operators are generated for C typecasts.

Datatype Key:

- uint: Unsigned integers, any bitwidth
- int: Signed integers, any bitwidth
- float: Single-precision IEEE 754 floating-point type
- double: Double-precision IEEE 754 floating-point type

Name	**C operator**	**Operand datatype**
slice	Internal	uint, int
sign_extend	Internal	uint
lnot	!	uint, int
not	~	uint, int
and	&, &&	uint, int
or	|, ||	uint, int
xor	^	uint, int
xnor	!(a^b)	uint, int
lsl	<<	uint, int
lsr	>>	uint
asr	>>	int
add	+	uint, int
sub	-	uint, int
mul2_u	*	uint
mul2_s	*	int
div_u	/	uint
div_s	/	int
mod_u	%	uint
mod_s	%	int
neg	- (unary)	int
cmp_eq	==	uint, int
cmp_neq	!=	uint, int
cmp_lt_s	<	int
cmp_lteq_s	<=	int
cmp_lt_u	<	uint
cmp_lteq_u	<=	uint
fadd	+	float
fsub	-	float
fmul	*	float
fdiv	/	float
fneg	- (unary)	float
fcmp_eq	=	float
fcmp_neq	!=	float

fcmp_lt	<	float
fcmp_lteq	<=	float
faddd	+	double
fsubd	-	double
fmuld	*	double
fdivd	/	double
fnegd	- (unary)	double
fcmpd_eq	=	double
fcmpd_neq	!=	double
fcmpd_lt	<	double
fcmpd_lteq	<=	double

Figure 23: Impulse C Compiler operators

Name	**Source type**	**Destination type**
ftod	float	double
dtof	double	float
itof	int	float
itod	int	double
utof	uint	float
utod	uint	double
ftoi	float	int
dtoi	double	int

Figure 24: Impulse C Compiler operators, typecast operations