

1 Help Contents



CoDeveloper™
from
Impulse Accelerated Technologies™
SGI RC100 Platform Support Package



CONTENTS

[Before You Begin \(Read This First\)](#)

[SGI Platform Support Package Overview](#)

[About the SGI RC100](#)

[Programming for the RC100](#)

[Using the SGI RC100 PSP](#)

1.1 SGI Platform Support Package Overview



Welcome to CoDeveloper™ for the SGI RC100



Welcome to CoDeveloper for the SGI RC100: advanced software technologies enabling FPGA-acceleration of high-performance computing applications. The CoDeveloper tools allow you to create FPGA hardware in the form of synthesizable hardware description language (HDL) files and related hardware and software interface logic. CoDeveloper is capable of generating hardware described by one or more hardware processes written in an ANSI C. These processes communicate with other hardware and software processes using methods that include streaming, shared memory, message passing, and simple registers.

The complete CoDeveloper environment consists of a set of libraries that allow Impulse C applications to be compiled by a standard C/C++ desktop compiler (for simulation and debugging purposes), as well as cross-compiler and translation tools that allow mixed hardware/software applications to be implemented on selected programmable hardware platforms.

The Platform Support Package described in this document extends the power of CoDeveloper by providing automated software/hardware generation for FPGA accelerators that execute on supported SGI Altix systems that contain an SGI Reconfigurable Application Specific Computing (RASC) module, or RASC RC100 blade. The combination of CoDeveloper and the RC100 Platform Support Package (PSP) enables a software application engineer to create FPGA accelerators from Impulse C source code alone. Typically, no user-supplied HDL is required. CoDeveloper for the RC100 is also integrated with Xilinx ISE and Synplicity Synplify Pro software, making CoDeveloper easier to use for first-time FPGA programmers.

Platform Support Package Highlights

- Supports the RC100 blade, coupled with an Altix server
- Supports Impulse C communication mechanisms, including streams, shared memory, signals, and registers, using SGI CoreServices version 2.20
- Supports IEEE 754-compliant single- and double-precision floating-point arithmetic through Xilinx IP libraries
- Provides integration with synthesis tools through generated Makefiles

Getting Started

To get started using CoDeveloper for the RC100 platform, you should load one of the sample projects included in the Examples\Scientific\SGI subdirectory of your CoDeveloper installation. These examples will help you understand how to create and manage an Impulse C project in conjunction with the software tools provided by Xilinx (or Synplicity) and SGI.

To load a sample project, open the Welcome page from the CoDeveloper Application Manager Eclipse

IDE's Help menu. Click the Impulse C Samples link and scroll to find the SGI project listing. Select one of the sample projects and click its name to import the project source code and settings into your workspace. Be sure to review any Readme files included with the sample project.

Before starting with a CoDeveloper example project, be sure you have reviewed the information in the [Read This First](#) section of this document. Also, please take time to familiarize yourself with the SGI RC100 documentation.

See Also

[Help Contents](#)
[Read This First](#)
[About the SGI RC100](#)

1.1.1 Read This First

Release Notes

This is release 2.20 of the **CoDeveloper for SGI RC100 Platform Support Package**, from Impulse Accelerated Technologies.

System Requirements

Two separate computers are required to use this Platform Support Package (PSP). One computer is used to create and compile the Impulse C application. This computer is called the *development* system. The compiled hardware application is downloaded to the second computer, called the *target*, where the software portion of the application is built and executed.

The development and target systems may be the same, but this arrangement has not been tested by Impulse.

The requirements for the development system computer are:

- x86 PC with 512 MB of RAM (2 GB recommended for place and route)
- Windows 2000, XP, or Vista; or Red Hat Enterprise Linux, or CentOS, version 5.0 or greater
- Impulse CoDeveloper 3.30 or later
- Xilinx ISE v9.1i, licensed and installed
- Optional: Third-party synthesis software supporting Xilinx FPGAs, such as Synplicity Synplify Pro
- Optional: Third-party HDL simulation software

The requirements for the target computer are:

- SGI Altix system with RASC RC100 blade, connected with NUMalink cables
- SGI CoreServices 2.20 (other versions not supported)
- SUSE Linux Enterprise Server (SLES) version 10 SP1 installed on the SGI Altix system
- SGI patch #10527 applied to the SGI Altix Linux system; the patch may be downloaded from <http://support.sgi.com/>

To target an SGI RASC system with Impulse C, simply select the platform from the drop-down list in the project's Properties dialog in CoDeveloper, as described in [Using the SGI RC100 PSP](#).

Installation Notes

The RC100 PSP assumes that the Xilinx ISE or Synplicity Synplify Pro software from is properly

installed on the development system. Specifically, the "synplify_pro" or "xst" programs must be found using the PATH environment variable.

In order for the exported hardware Makefile to function properly, the following environment variables must be set as indicated on *both* the development system and the target system:

RASC = <Path to RASC software environment, using UNIX-style '/' slashes>

If the RASC path is not set on a Windows development system using UNIX-style slashes, running "make" to generate a hardware bitstream may result in an error similar to the following:

```
C:\Impulse\CoDeveloper3\workspace\alg12_istrm\Hardware\hw>make
SHELL=C:/Impulse/CoDeveloper3/bin/sh.exe use_ise
make SYNTHESIS_RESULT_EXT=ngc ise_xst
make[1]: Entering directory
~/cygdrive/c/Impulse/CoDeveloper3/workspace/alg12_istrm/Hardware/hw'

Implementation folder 'rev_1' created!

rm -rf xstlib pd top shrd wrapper pio emp mem_port qdr_sram_intf srm
mkdir xstlib pd top shrd wrapper pio emp mem_port qdr_sram_intf srm
cp C:\SGI\RASC/design/shrd/rst_rep_delay.v xstlib
cp: cannot stat `C:\SGIRASC/design/shrd/rst_rep_delay.v': No such file or directory
make[1]: *** [ise_xst] Error 1
make[1]: Leaving directory
~/cygdrive/c/Impulse/CoDeveloper3/workspace/alg12_istrm/Hardware/hw'
make: *** [use_ise] Error 2
```

Limitations

The SGI RC100 Platform Support Package provides the interface between Impulse C-generated logic and SGI CoreServices. This release of the PSP supports most Impulse C features, but some attention must be paid to RASC-specific requirements. All interfaces between the Impulse C-generated hardware and the host software use the SGI CoreServices facilities; this imposes some limitations on how algorithms in the FPGA communicate with the host CPU. In summary:

- Stream, shared memory, register, and signal communication channels are supported
- Multiple software processes on the host CPU are supported, but communication using `co_*` objects between software processes is not thread-safe
- Software stream transfers are scheduled immediately, but executed when `co_iterate_hardware()` is called
- The hardware "done" flag must be set when an algorithm finishes
- At most four of each input and output streams are supported
- There is a limited number of hardware registers available in CoreServices
- Names reserved by CoreServices may conflict with Impulse C identifiers
- EOS only works if full 128-byte cache lines are transferred
- 128-bit stream sizes are recommended for performance and deadlock safety
- Dual clocks are not supported
- Active-low reset is not supported

For details on these limitations and how they affect the design of an application, see the section [Programming for the RC100](#).

Check the [Impulse Support Forum](#) periodically for news about RC100 PSP updates.

Prerequisites

This Platform Support Package documentation assumes that you have read and understand the

introductory sections of the CoDeveloper User's Guide, which is installed with your CoDeveloper product and can be accessed from the CoDeveloper Help menu. In particular, you should take the time to go through the tutorials provided with CoDeveloper so you have a good understanding of the front-end design flow including both desktop simulation and hardware compilation.

Note: The tutorials in this document are intended for illustrative purposes only. The steps required to use CoDeveloper with the SGI RC100 FPGA platform may differ somewhat from the steps described here. Please take the time to familiarize yourself with the SGI documentation prior to beginning.

Getting Started

A sample project targeting the RC100 platform is included with the Platform Support Package. To work with this example:

1. Start the Impulse Application Manager by selecting Start > Impulse Accelerated Technologies > CoDeveloper > CoDeveloper Application Manager (Eclipse IDE).
2. Select the Help > Welcome menu item in CoDeveloper. Click the "Browse Impulse C Samples" link to view the samples page.
3. Navigate to the Mandelbrot sample project in the SGI section of the Impulse C Sample Projects list.

Once you have opened the sample project, you can review the Impulse C source code, launch a desktop simulation or invoke the Impulse C hardware generator to process the sample application and generate output files compatible with the RC100. These steps are described in the [Using the SGI RC100](#) section of this document.

1.1.2 Contents



CoDeveloper™
from
Impulse Accelerated Technologies™
SGI RC100 Platform Support Package



CONTENTS

[Before You Begin \(Read This First\)](#)

[SGI Platform Support Package Overview](#)

[About the SGI RC100](#)

[Programming for the RC100](#)

[Using the SGI RC100 PSP](#)

1.2 About the SGI RC100

RASC Overview

The RASC program leverages more than 20 years of SGI experience accelerating algorithms in hardware. Rather than using relatively fixed implementations, such as graphics processing units (GPUs), RASC uses FPGA technology to develop a full-featured reconfigurable computer. The RASC program also addresses the ease of use and performance issues present in typical RC environments.

To address performance issues, RASC connects FPGAs into the NUMalink fabric making them a peer to the microprocessor and providing both high bandwidth and low latency. By attaching the FPGA devices to the NUMalink interconnect, RASC places the FPGA resources inside the coherency domain of the computer system. This placement allows the FPGAs extremely high bandwidth (up to 6.4GB/s/FPGA), low latency, and hardware barriers. These features enable both extreme performance and scalability. The RASC product also provides system infrastructure to manage and reprogram the contents of the FPGA quickly for reuse of resources.

For more information about the SGI RC100 system and RASC technology, see <http://www.sgi.com/products/rasc/>.

See Also

[Programming for the RC100 Coprocessor](#)

[Using the SGI RC100 PSP](#)

Quick Start Tutorials

1.3 Programming for the RC100

Please follow these guidelines when writing an Impulse C application for the RC100. These tips will help you to take full advantage of SGI CoreServices for FPGA-to-host communication, and to use the included Makefile to build the FPGA bitfile automatically.

Configuration Function

Make sure the 'name' (first) argument to `co_architecture_create` is the same as the name of the CoDeveloper project. Otherwise, you may see errors from ISE about missing files when running 'make use_ise' to build the FPGA bitfile, for example:

```
=====
*                               HDL Compilation                               *
=====
ERROR:HDLCompilers:175 - Source file userip/des3_comp.v does not exist
ERROR:HDLCompilers:175 - Source file userip/des3_top.v does not exist
Analysis of file <"des3_xst.prj"> failed.
```

Execution of Software I/O Commands

Software: Control Hardware Execution

Impulse C software I/O commands may not be completed immediately when issued. The current PSP implementation will perform `co_register` reads and writes immediately, but will only start up the data transfers for `co_stream` reads and writes. The algorithm will not finalize any operations until a call to `co_iterate_hardware` is executed by the software process. After return from the call, the FPGA algorithm will have been started and all queued data transfers will be complete. This may require that programmers think a little differently about their problems.

Several RC100-specific Impulse C functions control execution of FPGA hardware. These functions, called in code running on the host CPU, are described in detail here:

- [co_iterate_hardware](#)
- [co_release_hardware](#)
- [co_start_hardware](#)
- [co_wait_hardware](#)

Hardware: Set the Done Flag

The hardware portion of an Impulse C application must use a special `co_signal`, the "done" flag, to indicate to Core Services that work is complete on the FPGA. This "done" signal must be created and attached to a `co_port` as follows:

```
// Configuration function
void config_app(void *arg)
{
    co_signal done_sig;
    IF_NSIM(co_port done_port;)
    co_process hardware_process;

    // Create "input" and "output" streams, etc., here

    done_sig = co_signal_create_ex("done_sig", UINT_TYPE(0));
    IF_NSIM(done_port = co_port_create("done", co_output, done_sig);)

    hardware_process = co_process_create("hardware", (co_function)hardware,
        3,
        input, output, done_sig);
```

```

    //...
}

```

Important: For the "done" flag to be properly connected to CoreServices, the "Do not include co_ports in bus interface" compiler option must *not* be set before using CoDeveloper to generate logic.

In the above example code, note the use of the IF_NSIM macro to hide the use of the co_port when compiling the application for desktop simulation. To simulate the hardware/software system under Windows or Linux, the "done" flag must be connected to a software process--not a co_port.

See the example projects in the "Examples\Scientific\SGI" subdirectory of the CoDeveloper installation for more details.

Mapping I/O to Hardware Resources

Impulse C co_stream, co_register, and co_signal all require allocation of the fixed register resources provided in CoreServices. Although this limit is large, it is possible that an algorithm may run out of resources. This condition is detected in the PSP and the program will fail to generate code if the condition is detected.

All of the SGI CoreServices names are considered reserved and can not be used for variable names in the argument list of the hardware or software processes or in the name field of a co_port_create command. See the RC100 User's Guide for lists of reserved names.

Assignment of co_stream, co_register, and co_signal objects to available CoreServices hardware resources is automatic. The "Default resource" column in the table below lists which resources are used, by default, to implement each co_* type. Specific resources can be assigned if a particular application requires it, however. This is done by appending certain strings to prototype identifiers in the Impulse C hardware code. To allocate a co_register to the debug registers, for example, append "debug_port_N" to the corresponding parameter name in the host software process' prototype declaration. The following code, from a hardware source file, illustrates this example:

```

// Software process defined in app_sw.c, but this prototype is in app_hw.c
void app_software(co_stream host2fpga,
                 co_stream fpga2host,
                 co_register inc_val,
                 co_register debug_port_0,
                 co_register debug_port_1,
                 co_register debug_port_2);

```

The three co_registers "debug_port_0", "debug_port_1", "debug_port_2" will be assigned to the debug registers in CoreServices and can then be observed from SGI's RASC-specific gdb when debugging host software (see the [Troubleshooting](#) section).

To allocate...	append string:	N in range	Connects to	Default?
co_stream	"strm_N"	0 .. 3	DMA streaming engines	Yes
co_stream	"_mmr_N"	0 .. 15	Algorithm Defined Registers (ADRs)	No
co_register	"alg_def_reg_N"	0 .. 63	ADRs	Yes
co_register	"debug_port_N"	0 .. 63	Debug registers	No
co_signal	"alg_def_reg_N"	0 .. 63	ADRs	Yes

There is the limit of four input and four output dedicated streams. The RC100 environment is biased

towards providing the best performance that can be obtained and provides only four input and four output ports for streaming. Mapping multiple algorithm streams onto a single hardware stream would give up substantial performance gains possible when using dedicated streaming ports, so streams are mapped directly onto the SGI CoreServices facilities.

If more streams are needed, the non-performance-critical streams should be assigned to ADRs by naming them appropriately (e.g., "myStream_mmr_0"). Streams implemented with memory-mapped ADRs require polling across the RC100's buses, so offer much lower performance than dedicated streaming ports. See also: [co_stream_mmr](#).

Shared Memory Resources

Assigning a `co_memory` to a particular hardware resource is done with the `co_memory_create` function. For example:

```
sram0_c=co_memory_create("sram0_c","sram128_0",2048);
sram1_d=co_memory_create("sram1_d","sram128_1",2048);
```

Memory Device	'loc' parameter (co_memory_create)	N in range
8MB 64-bit-wide SRAM	"sram_N"	0 .. 4
16MB 128-bit-wide SRAM	"sram128_N"	0 .. 1

See the RC100 User's Guide for more on available resource limits.

Optimal Stream Transfer Sizes

For best performance and deadlock safety, we recommend that only 128-bit streams be used between hardware and software processes.

Both the total amount of data transferred and the size of the stream packets involved should be carefully managed when using the SGI RC100 with Impulse C. The SGI RC100's memory network is biased towards moving full cache lines (128 bytes) of data. No mechanism is provided by CoreServices to recognize how much of the data within a cache line is valid. As a result, end-of-stream (EOS) indicators, returned by `co_stream_read` and `co_stream_eos`, will only function correctly if the user transfers an amount of data that completely fills the final cache line. This restriction exists on both input streams and output streams.

It is possible to process less data from the input streams if the algorithm is given some other indication of how to detect the last data items (such as through a register or signal). In that case, any unused data in the stream buffer will be discarded when the stream is closed. For output streams, transfer sizes must be a multiple of cache line size, as the final cache line will not be flushed from CoreServices to main memory, and available to the host software, unless the cache line has been completely filled. (Note: Plans are to lift this restriction on output streams in CoreServices version 2.2.)

Streams have 128-bit interfaces to the memory bus. Writing to a stream from software will cause smaller packets to be queued together and sent in 128-bit chunks. Impulse C will provide a narrower interface on stream reads by the hardware, if needed, but each hardware read will consume 128 bits from the input buffer and only pass the (smaller) requested number of bits into the hardware process. Deadlock can therefore result if the hardware expects to read the same number of small packets as are written by software. For example, if software writes 1024 32-bit packets to a stream, CoreServices will actually send 256 128-bit packets. One of those 128-bit packets will be consumed by each read done by the hardware, regardless of the specified read size (32 bits), so the hardware process will block when it starts to read for the 257th time. This situation also results in data loss for the hardware process.

Furthermore, the PSP does not provide a "pack" interface on hardware output, so each hardware write will send 128 bits to CoreServices even when fewer bits are specified on the write command. The upper bits of such small writes will be filled with zeroes.

Stream Buffer Sizes

CoreServices allocates sufficient buffer space for managing streams, so it is not necessary to create deep FIFOs for streams between host and FPGA. A value of 1 in the third ('numelements') argument to `co_stream_create` is usually sufficient, even when sending large numbers of packets:

```
co_stream host2fpga = co_stream_create("input", UINT_TYPE(128), 1);
```

Creating a large buffer only increases the place-and-route time and increases the timing error of the final bitstream. There is a need for some space, to manage potential timing issues, but the PSP's software library already insures that the necessary minimum is reserved. The only reason to specify a larger depth would be for streams used to communicate between hardware processes that do not connect to CoreServices.

Specifying Hardware Clock Rate

The hardware clock speed will default to 100MHz unless the user selects one of the other supported speeds. This is accomplished by including a `#define SGI_CLOCK_SPEED N` macro in a hardware source file. Values of N are interpreted as megahertz. The supported values for N are 200, 100, 66 and 50, which result in generation of a clock interval of 5, 10, 15 or 20 nanoseconds. The target clock rate will be reported by the Impulse C compiler when generating HDL.

It is important to examine the post-place-and-route timing report, produced by the ISE or Synplify tools, to verify that the hardware design was able to meet the target clock rate. Timing estimates produced in synthesis may be inaccurate. When using ISE, examine the `timing.twr` file. No timing errors should be reported; otherwise, the complete application may not produce the expected results. In such cases, it may help to aim for a lower clock rate using the `SGI_CLOCK_SPEED` macro, regenerate HDL in CoDeveloper, export, and rebuild the bitfile.

See Also

[Using the SGI RC100 PSP](#)

1.3.1 `co_iterate_hardware`

```
void co_iterate_hardware();
```

Header File

```
co.h
```

Callable Within

Host CPU software processes

Description

Initiate execution of FPGA hardware processes and wait for completion of I/O tasks.

Parameters

None

Return Value

None

Notes

Must be called after `co_initialize`, or exits the program. Equivalent to calling [co_start_hardware](#), then [co_wait_hardware](#).

Returns only after the "done" signal has been received from the FPGA.

1.3.2 co_release_hardware

```
void co_release_hardware();
```

Header File

`co.h`

Callable Within

Host CPU software processes

Description

Release all CoreServices resources.

Parameters

None

Return Value

None

Notes

Called after the host application is finished using the FPGA to free operating system resources.

1.3.3 co_start_hardware

```
void co_start_hardware();
```

Header File

`co.h`

Callable Within

Host CPU software processes

Description

Check that FPGA hardware has been initialized.

Parameters

None

Return Value

None

Notes

Must be called after `co_initialize`, or exits the program.

1.3.4 `co_stream_mmr`

```
void co_stream_mmr(co_stream s);
```

Header File

`co.h`

Callable Within

`co_init.c` (compiler-generated)

Description

Assign a stream to memory-mapped registers, rather than dedicated streaming hardware resources in CoreServices.

Parameters

<code>co_stream s</code>	The stream to implement using memory-mapped registers
--------------------------	-------------------------------------------------------

Return Value

None

Notes

By default, streams are implemented using high-speed, dedicated hardware managed by CoreServices. This function will cause the Impulse C compiler to implement a stream using memory-mapped registers. The register-based implementation has much lower throughput, but since dedicated streams are a limited resource, it may be necessary to use a register-based stream for a non-critical application function.

Calls to this function are automatically generated in `co_init.c` if the stream's name contains the string `"_mmr_"`. See the heading "Mapping I/O to Hardware Resources" in the section

[Programming for the RC100](#) for details on this naming convention.

1.3.5 co_wait_hardware

```
void co_wait_hardware();
```

Header File

```
co.h
```

Callable Within

Host CPU software processes

Description

Initiate and wait for completion of I/O tasks.

Parameters

None

Return Value

None

Notes

Must be called after co_initialize, or exits the program.

Returns only after the "done" signal has been received from the FPGA.

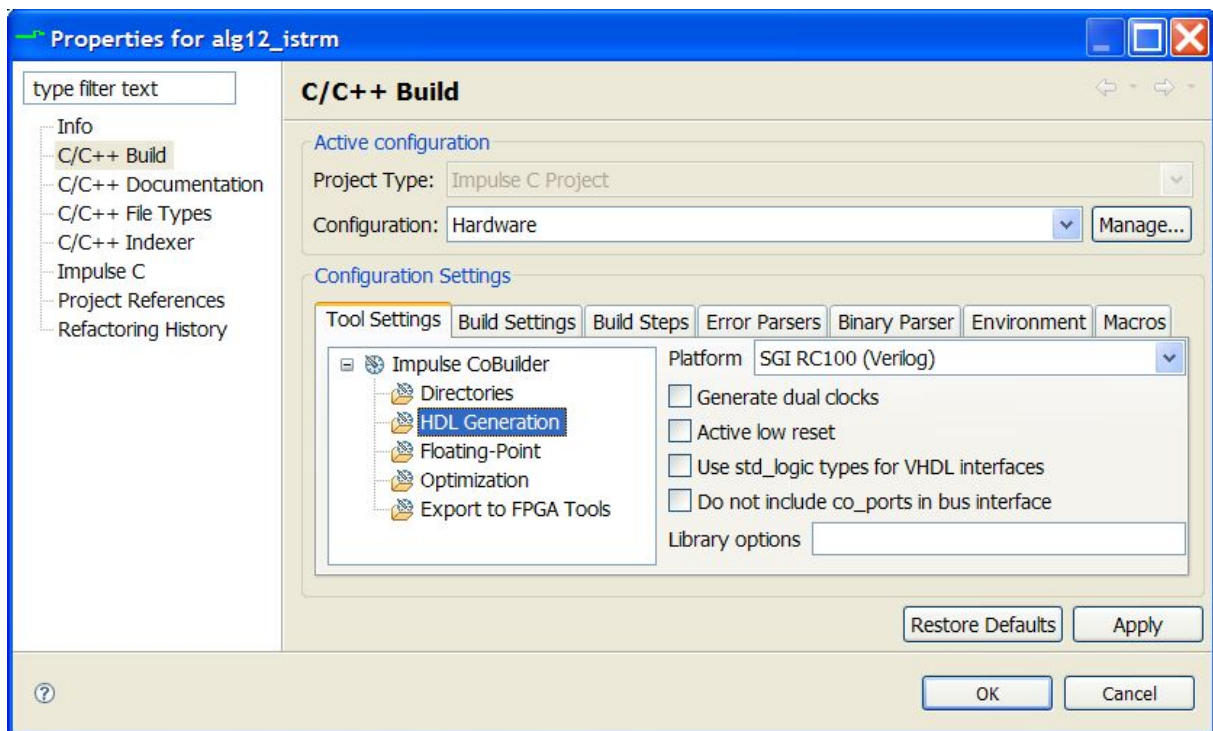
1.4 Using the SGI RC100 PSP

Generating Hardware for the RC100

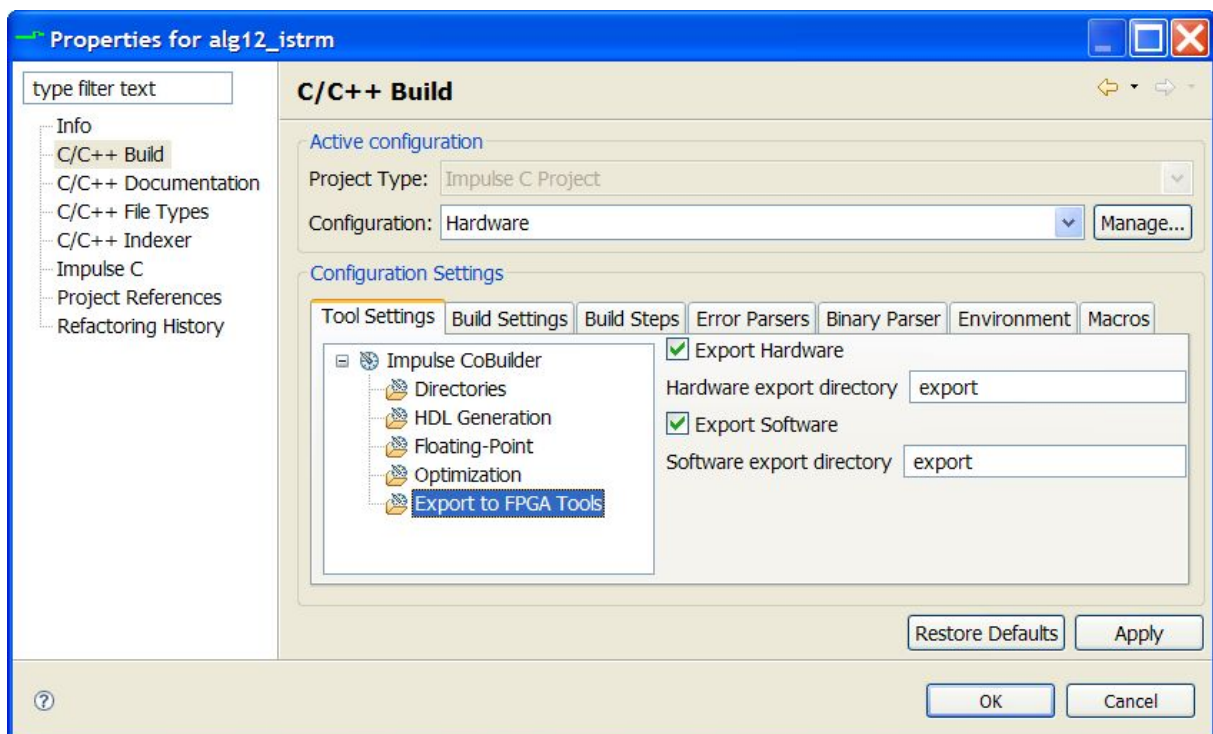
Set Hardware Configuration Properties

To target the RC100 when generating hardware, select the "SGI RC100 (Verilog)" Platform Support Package from the "Platform" drop-down list in the Impulse C project's Hardware Configuration Settings. Right-click the project and select Properties. In the dialog that appears, make sure "Hardware" is selected as the Active Configuration. Under "Impulse CoBuilder" in the "Tool Settings" tab, select "HDL Generation" to see the "Platform" drop-down list.

Note that the "Do not include co_ports in bus interface" option must *not* be selected, as shown.



Specify export directories for hardware and software files generated by the Impulse C compiler. From the Hardware Configuration Properties dialog's Tool Settings tab, select "Export to FPGA Tools" under "Impulse CoBuilder". Enter a subdirectory path (e.g., "export") in the "Hardware export directory" and "Software export directory" fields, and make sure the "Export Hardware" and "Export Software" options are selected, as shown:



Click OK in the Properties dialog to save settings and return to the CoDeveloper workbench.

Generate Hardware

Right-click your project in the Impulse C Projects view and set the Active Build Configuration to "Hardware". Right-click the project again and select Build Hardware. (If the Build Hardware option is not available, first disable the Project > Build Automatically menu item.) The Impulse C compiler and SGI RC100 PSP will generate Verilog and hardware/software interface files for your application. These files will be created in the "Hardware" subdirectory of your project's workspace.

Generating a Bitfile for the RC100

Please note the [System Requirements](#) before proceeding. Errors may occur when generating a bitfile or running the application if the development or target systems are not properly configured.

The CoDeveloper SGI RC100 PSP is designed to automate the task of producing an FPGA accelerator for the RC100, reducing the development time otherwise required. The following assumes the project has been created, the Impulse C application has been written, and hardware has been both generated and exported:

1. Open a Command Prompt (Start > Run > "cmd").
2. Navigate ("cd") to the Impulse C project's subdirectory of your CoDeveloper workspace (usually under "C:\Impulse\CoDeveloper3\workspace").
3. Navigate to the subdirectory where the generated hardware files were exported ("Hardware\export\hw").
4. To build using Xilinx ISE, enter the command "make use_ise". The synthesis, place-and-route, and bitfile generation processes will run. This will take some time, from 15-90 minutes depending on the application and your workstation's capabilities.
5. (Optional) To build using Synplicity Synplify Pro, enter the command "make use_synplify".

Note: On a Windows development system, you may have to pass the path to the Impulse "sh.exe" program to "make" in the "SHELL" macro, using UNIX-style slashes, as follows:

```
make SHELL=C:/Impulse/CoDeveloper3/bin/sh.exe use_ise
```

Running an Application on the RC100

Once the bitfile has been generated, copy generated hardware and software files to the RC100 and run the application.

1. On the RC100 machine, create a working directory for the application.
2. Copy the project's "Hardware\export\hw\bit" subdirectory from the development system to the RC100 machine.
3. Copy the project's "Hardware\export\sw" subdirectory from the development system to the RC100 machine.
4. Now on the RC100, register the bitfile with the system. Navigate to the "bit" directory you copied over from the development system.
5. (Windows only) If the CoreServices configuration file "core_services.cfg" does not exist, enter the command "sh extractor" to create it. *This command should only be run once, when the bitfile has changed, to create "core_services.cfg".*
6. Enter the command "sh install" to register the bitfile.
7. Build the software executable. Navigate to the "sw" directory you copied over from the development system and enter the command "make".
8. Run the software executable with the command "./a.out".

1.5 Troubleshooting

Problems Meeting Timing

The hardware design exported from CoDeveloper may fail to meet timing requirements during the process of building a bitfile. If this occurs, first try lowering the target clock rate specified in your Impulse C code. See the heading "Specifying Hardware Clock Rate" in the section [Programming for the RC100](#).

If the minimum clock rate is already specified, try adding constraints to the hardware build process. Timing issues can be mitigated with the use of a .ucf ("user constraints") file added to the "hw" subdirectory of your project's "Hardware export directory". You can modify the "Makefile.local" found there to use a user-generated .ucf file, named "alg.ucf", by uncommenting the following assignment:

```
# Algorithm dependent UCF constraints file
#ALG_UCF=alg.ucf
```

User constraints that assign optimum locations to critical hardware components can help your design meet timing. For example, the suggested locations for the CoreServices 2.20 stream buffers are as follows; add these lines to "alg.ucf" if necessary:

```
INST "sbi/sbi_0/buff[0]" LOC=RAMB16_X3Y13;
INST "sbi/sbi_0/buff[1]" LOC=RAMB16_X3Y14;
INST "sbi/sbi_0/buff[2]" LOC=RAMB16_X3Y15;
INST "sbi/sbi_0/buff[3]" LOC=RAMB16_X3Y16;
INST "sbi/sbi_1/buff[0]" LOC=RAMB16_X3Y17;
INST "sbi/sbi_1/buff[1]" LOC=RAMB16_X3Y18;
INST "sbi/sbi_1/buff[2]" LOC=RAMB16_X3Y19;
INST "sbi/sbi_1/buff[3]" LOC=RAMB16_X3Y20;
INST "sbi/sbi_2/buff[0]" LOC=RAMB16_X3Y21;
INST "sbi/sbi_2/buff[1]" LOC=RAMB16_X3Y22;
INST "sbi/sbi_2/buff[2]" LOC=RAMB16_X3Y23;
INST "sbi/sbi_2/buff[3]" LOC=RAMB16_X3Y24;
INST "sbi/sbi_3/buff[0]" LOC=RAMB16_X3Y25;
INST "sbi/sbi_3/buff[1]" LOC=RAMB16_X3Y26;
INST "sbi/sbi_3/buff[2]" LOC=RAMB16_X3Y27;
INST "sbi/sbi_3/buff[3]" LOC=RAMB16_X3Y28;

INST "sbo/sbo_0/fifo[0]" LOC=FIFO16_X2Y13;
INST "sbo/sbo_0/fifo[1]" LOC=FIFO16_X2Y14;
INST "sbo/sbo_0/fifo[2]" LOC=FIFO16_X2Y15;
INST "sbo/sbo_0/fifo[3]" LOC=FIFO16_X2Y16;
INST "sbo/sbo_1/fifo[0]" LOC=FIFO16_X2Y17;
INST "sbo/sbo_1/fifo[1]" LOC=FIFO16_X2Y18;
INST "sbo/sbo_1/fifo[2]" LOC=FIFO16_X2Y19;
INST "sbo/sbo_1/fifo[3]" LOC=FIFO16_X2Y20;
INST "sbo/sbo_2/fifo[0]" LOC=FIFO16_X2Y21;
INST "sbo/sbo_2/fifo[1]" LOC=FIFO16_X2Y22;
INST "sbo/sbo_2/fifo[2]" LOC=FIFO16_X2Y23;
INST "sbo/sbo_2/fifo[3]" LOC=FIFO16_X2Y24;
INST "sbo/sbo_3/fifo[0]" LOC=FIFO16_X2Y25;
INST "sbo/sbo_3/fifo[1]" LOC=FIFO16_X2Y26;
INST "sbo/sbo_3/fifo[2]" LOC=FIFO16_X2Y27;
INST "sbo/sbo_3/fifo[3]" LOC=FIFO16_X2Y28;
```

Using the gdbfpga Debugger

The RASC software ships with gdbfpga, a version of the GNU Project's gdb debugger, modified to help debug programs that use RASC FPGAs. For documentation on gdbfpga, see SGI's RASC User's Guide.

Adding Debug Registers

Registers in an Impulse C design (`co_register`) can be mapped through SGI's CoreServices so they are visible to the RASC gdb debugger. To add a debug register, add a `co_register` object to any hardware process and write to it the values you wish to observe while debugging. Debug registers must be created with special names in the software process' prototype declaration in order to be mapped from hardware to software.

See the topic "Mapping I/O to Hardware Resources" under [Programming for the RC100](#) for details on the naming convention for debug registers.

Debug Output from Device Manager (devmgr)

Errors from CoreServices can be debugged using output from the "devmgr" server. To turn server debugging output on/off, use these commands:

```
devmgr -x on
devmgr -x off
```

Debug output is directed to the system console by default. If output does not appear, even after running a failing program, restart the devmgr server so its output is directed to your current console:

```
/etc/init.d/rasc restart
```

Error code -41

The RASC system may run out of the contiguous memory it needs to store FPGA bitmaps and handle streaming data. In this case, attempting to run an FPGA-accelerated application may result in a "Configure failed" error:

```
sgi-impulse:/usr/share/rasc/examples # ./alg12_strm
configure failed at 55: -41
configure: Configure failed. Either no such bitstream or reservation pool
exhausted. (error code: -41)
```

To fix this problem, try one of the following:

- Unload unused algorithms. `devmgr -q` shows which algorithms are loaded, and `devmgr -d -n <algorithm name>` removes an algorithm.
- Restart devmgr: `/etc/init.d/rasc restart`
- Reboot the system