1 Impulse C User Guide



Impulse C User Guide Contents

Impulse C Overview
The Programming Model
Processes and How They Communicate
Impulse C Datatypes
Impulse C Processes
Creating Processes
Creating Streams
Reading and Writing Streams
Creating and Using Signals
Creating and Using Semaphores
Creating and Using Registers
Fixed-Point Arithmetic
Programming for Hardware
Instrumentation and Debugging
Function Reference

1.1 Impulse C Overview



Description

Impulse C will change the way you describe highly parallel applications and algorithms, but Impulse C is not a new programming language. Instead, Impulse C extends standard ANSI C, using C-compliant predefined library functions in support of a modified form of the communicating sequential processes (or CSP) programming model. CSP is conceptually similar to the dataflow programming model, in that it simplifies the expression of highly parallel algorithms through the use of well-defined data communication, message passing and synchronization mechanisms.

In Impulse C, the CSP programming model is modified somewhat by allowing the buffering of data being transmitted between processes. This buffering of data (which is implementing using FIFOs that are specified and configured by the Impulse C programmer) makes it possible to write parallel applications at a higher level of abstraction, without the clock-cycle by clock-cycle synchronization that would otherwise be required.

Impulse C is designed for dataflow-oriented applications, but is also flexible enough to support alternate programming models including the use of shared memory as a communication mechanism. The programming model that you, as an Impulse C programmer, will select will depend on the requirements of your application, but also on the architectural constraints of the selected programmable platform target.

© 2003-2008 Impulse Accelerated Technologies

The Impulse C library consists of minimal extensions to the C language (in the form of new data types and predefined function calls) that allow multiple, parallel program segments to be described, interconnected and synchronized. The Impulse C compiler translates and optimizes Impulse C programs into appropriate lower-level representations, including Register-Transfer-Logic (RTL) VHDL descriptions that can be synthesized to FPGAs, and standard C (with associated library calls) that can be compiled onto supported microprocessors through the use of widely available C cross-compilers compilers.

The complete CoDeveloper environment consists of a set of libraries allowing Impulse C applications to be executed in a standard desktop compiler (for simulation and debugging purposes) as well as cross-compiler and translation tools allowing Impulse C applications to be implemented on selected programmable hardware platforms. Additional tools for application profiling and co-simulation with other environments (including links to EDA tools for hardware simulation) are provided.

See Also

Processes and How They Communicate
Impulse C Processes
Creating Processes
Creating Streams
Reading and Writing Streams
Instrumentation and Debugging

1.2 The Programming Model



Introduction

CoDeveloper is a parallel programming environment based on standard ANSI C, extended using the Impulse C library. Impulse C extends standard C to support a modified form of the communicating sequential processes, or CSP, programming model. CSP is conceptually similar to the dataflow programming model, in that it simplifies the expression of highly parallel algorithms through the use of well-defined data communication, message passing and synchronization mechanisms.

In Impulse C applications, hardware and software elements (processes) communicate primarily through buffered data streams that are implemented directly in hardware. This buffering of data, which is implemented using FIFOs that are specified and configured by the Impulse C programmer, makes it possible to write parallel applications at a relatively high level of abstraction, without the clock-cycle-by-clock-cycle synchronization that would otherwise be required.

Impulse C is designed for dataflow-oriented applications, but is also flexible enough to support alternate programming models including the use of shared memory as a communication mechanism. The programming model that you, as an Impulse C programmer, will select will depend on the requirements of your application, but also on the architectural constraints of the selected programmable platform target.

The Impulse C library consists of minimal extensions to the C language (in the form of new data types and intrinsic function calls) that allow multiple, parallel program segments to be described, interconnected and synchronized. The Impulse C compiler translates and optimizes Impulse C programs into appropriate lower-level representations, including Register-Transfer-Logic (RTL) VHDL or Verilog that can be synthesized to FPGAs and standard C (with associated library calls) that can be

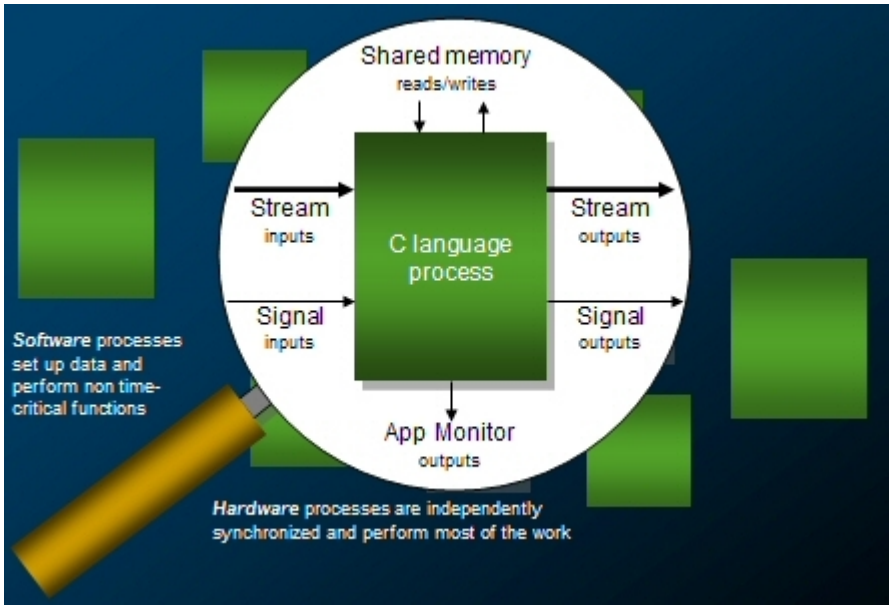
© 2003-2008 Impulse Accelerated Technologies

compiled onto supported microprocessors through the use of widely available C cross-compilers compilers.

The Programming Model

The Impulse C programming model, which is loosely based on the communicating sequential process model originally described by C. A. R Hoare, is targeted at stream-oriented, mixed hardware/software applications.

At the heart of the Impulse C programming model are processes and streams. Processes are independently synchronized, concurrently operating segments of your application. Processes are written using standard C (subject to the limitations of the target platform) and perform the work of your application by accepting data, performing computations and generating outputs.



The data that are processed by your application will flow from process to process by means of streams, or in some cases by means of messages and/or shared memories. Streams represent one-way channels of communication between concurrent processes, and are self-synchronizing with respect to the processes by virtue of buffering, the characteristics of which you specify at the time a stream is created in your application.

Impulse C and Stream-oriented Programming

Impulse C is designed to simplify the development of highly parallel applications using a stream-oriented approach to data movement, processing and synchronization. Stream-oriented programming is conceptually similar to dataflow programming, but is less restrictive in that it more easily supports process synchronization (through data buffering and message passing) and non-dataflow concepts such as shared memories.

There are an unlimited number of applications that may be expressed using Impulse C, but the most efficient applications will be those that have some or all of the following characteristics:

- The application features high data rates to and from data sources and between processing elements.
- Data sizes are fixed (typically one byte to one word), with a relatively small stream payload to prevent processes from becoming blocked.

© 2003-2008 Impulse Accelerated Technologies

- Multiple related but independent computations are required to be performed on the same data stream.
- The data consists of low or fixed precision data values, typically fixed width integers or fixed point fractional values.
- There are references to local or shared memories, which may be used for storage of data arrays, coefficients and other constants, and to deposit results of computations.
- There are multiple independent processes communicating primarily through the data being passed, with occasional synchronization being requested via messages.

Impulse C is specifically designed to address such applications. Impulse C is represented by a small set of data types and intrinsic library functions callable from a conventional ANSI C program. The C-compatible language extensions are used to specify an application as a collection of processes communicating via streams, signals and shared memories. Intrinsic functions defined by Impulse C are used to communicate stream and signal data between the processes and to move data in and out of shared or local memory elements. The processes are to assigned to actual resources on the target programmable platform through the use of additional Impulse C intrinsic functions.

See Also

Impulse C User Guide
Programming for Hardware
Processes and How They Communicate
Hardware/Software Interfaces

1.3 Processes and How They Communicate

In the Impulse C programming model, there are two important types of elements: processes and communication objects. A set of processes, connected to each other using communication objects, comprises the core of an Impulse C application.

Processes

A process is an independently executing section of code that is defined by a C subroutine called the process run function. A software process is designated to run on a conventional processor, while a hardware process is designated to run on an FPGA or other programmable hardware element.

A software process is constrained only by the limitations of the target processor (whether a common RISC, a custom DSP processor, or a soft processor core), while a hardware process is typically more constrained. A process written for an FPGA, for example, must be written using a somewhat narrowly-defined subset of C to meet the constraints of the Impulse CoBuilder FPGA compiler. The language constraints for each type of platform processor are defined in the Platform Support Package documentation for each supported platform.

In addition to standard C expressions, predefined Impulse C functions that perform interprocess communication may be referenced in a software or hardware process. These functions operate on communication objects to share data among processes. For example, read/write operations on a data stream connecting two processes can be performed by calling functions on **co_stream** objects within the processes.

The Impulse CoBuilder compiler generates synthesis-compatible hardware descriptions (in the form of HDL code compatible with common FPGA synthesis tools) for one or more FPGAs, as well as a set of communicating processes (in the form of C code compatible with the target cross-compiler) to be

© 2003-2008 Impulse Accelerated Technologies

implemented on conventional processors. These two portions of an Impulse C application correspond to the set of hardware processes and the set of software processes, respectively.

Communication Objects

Impulse C defines several distinct mechanisms for communicating among processes:

Streams	Buffered, fixed-width data streams
Signals	One-to-one synchronization with optional data
Semaphores	One-to-many synchronization
Registers	Unbuffered data or control lines
Shared memories	Memory shared between hardware and software

To use these mechanisms, the programmer creates objects in the configuration function and passes them to processes, which then call functions on the objects.

Whether the communicating processes are designated as software or hardware, the Impulse C compiler automatically generates the C and HDL interfaces needed to make the connection.

See Also

Programming for Hardware
Creating Processes
Creating Streams
Reading and Writing Streams
Creating and Using Signals
Creating and Using Semaphores
Creating and Using Registers
Impulse C Datatypes

1.4 Impulse C Datatypes

Signed and Unsigned Types

Impulse C provides predefined unsigned and signed integer data types for selected bit widths ranging from 1 to 64, as shown below. All bit widths from 1 to 64 bits, inclusive, are supported. A simple convention is used to name these predefined types. Signed types have the name **co_int** followed by the bit width, while unsigned types are prefixed with **co_uint**.

Variables of these types may be used in an Impulse C program for either software or hardware processes. A stream, for example, may have one of these C integer types as its data element type.

During desktop simulation, types that do not have the same bit width as standard C types are modeled using the next largest type. This can result in differences in bit-accurate behavior between the desktop simulation environment and a hardware implementation. To prevent such differences and ensure bit-accurate modeling, you may choose to use the bit-accurate arithmetic macro operations defined in the following chart.

© 2003-2008 Impulse Accelerated Technologies

Impulse C type	Modeled as	Bit-exact macro operations
co_int2 – co_int8	int8	IADDn(a,b), ISUBn(a,b), IMULn(a,b), IDIVn(a,b)
co_int9 – co_int16	int16	IADDn(a,b), ISUBn(a,b), IMULn(a,b), IDIVn(a,b)
co_int17 – co_int32	int32	IADDn(a,b), ISUBn(a,b), IMULn(a,b), IDIVn(a,b)
co_int33 – co_int64	int64	IADDn(a,b), ISUBn(a,b), IMULn(a,b), IDIVn(a,b)
co_uint1	uint8	UADDn(a,b), USUBn(a,b), UMULn(a,b), UDIVn(a,b)
co_uint2 – co_uint8	uint8	UADDn(a,b), USUBn(a,b), UMULn(a,b), UDIVn(a,b)
co_uint9 – co_uint16	uint16	UADDn(a,b), USUBn(a,b), UMULn(a,b), UDIVn(a,b)
co_uint17 – co_uint32	uint32	UADDn(a,b), USUBn(a,b), UMULn(a,b), UDIVn(a,b)
co_uint33 – co_uint64	uint64	UADDn(a,b), USUBn(a,b), UMULn(a,b), UDIVn(a,b)

Note that the above macro operations are specified in terms of the bit width of their results. There is no enforcement or checking of the operands' bit widths.

Enumeration Types

The C keyword **enum** is supported by Impulse C, in both hardware and software processes, for defining enumerated integer types.

See Also

Processes and How They Communicate
Creating Streams
Simulation Features

1.5 Impulse C Processes

Description

Processes are the fundamental units of computation in an Impulse C application. Once they have been created, assigned and started, the processes in an application execute as independently synchronized units of code on the target hardware.

Processes vs. Threads

Programming with Impulse C processes is conceptually similar to programming with threads. As with thread programming, each Impulse C process has its own control flow, is independently synchronized and has access to its own local memory resources (which will vary depending on the target platform). For this reason it is relatively easy to convert applications written in threaded C (for example, using the Posix thread library) to Impulse C. There are some key differences, however:

© 2003-2008 Impulse Accelerated Technologies

- In thread programming, it is assumed that globals and heap memory are shared among threads. In Impulse C, memory may be created local to the process, or may be explicitly shared depending on the capabilities of the target platform.
- In thread programming, the threads are assumed to execute on the same processor. In Impulse C, the assumption is that each process is assigned to an independently synchronized processor.
- The Impulse C programming model is specifically designed to support mixed hardware and software targets, with process synchronization occurring primarily in hardware buffers (FIFOs) rather than through semaphores or messages.
- The Impulse C programming model assumes that processes are defined and created at the time the application is initialized and loaded, rather than dynamically created, invoked and torn down as in a threaded application.

Note that the Impulse C desktop simulation library is based on a threading model, so global variables and heap memory will be shared. Also, during simulation all Impulse C processes will be executing on the one processor (your desktop computer) although this may not be the case on the target platform.

Note

In simulation (when all processes are translated to software running on a single processor), the scheduling of instructions being executed within each process will predictable, but the scheduling of instructions across processes will be dependent on the threading model used in the host compiler and/or the host operating system.

See Also

Processes and How They Communicate

1.6 Creating Processes

The Configuration Function

The configuration function is a C function that must be present in every Impulse C application. It describes the application as a set of interconnected processes running in hardware and/or software. The configuration function is not called in the usual fashion, but is passed in a call to **co_architecture_create** as a function pointer, where it will be processed internally by Impulse C. For example:

```
// The configuration function.  Defined here...
void configure(void * arg)
{
     //...
}

co_architecture co_initialize(void * arg)
{
     // ... and "called" here.
     return co_architecture_create("filter_arch", "generic_vhdl", configure, arg);
}
```

The configuration function must return void and may take a single void pointer as its argument.

Creating Processes

Processes are created and named in an application's configuration function. In the following example, the **my_app_configuration** routine declares three processes (**procHost1**, **procPe1** and **procPe2**)

© 2003-2008 Impulse Accelerated Technologies

and associates these processes with three corresponding process run functions named **Host1**, **Pe1** and **Pe2**:

```
#define BUFSIZE 4
void my_app_configuration()
{
  co_process procHost1, procPe1, procPe2;
  co_stream s1, s2;

  s1 = co_stream_create("s1",INT_TYPE(16),BUFSIZE);
  s2 = co_stream_create("s2",INT_TYPE(16),BUFSIZE);

  procHost1 = co_process_create("Host1", (co_function)Host1, 1, s1);
  procPe1 = co_process_create("Pe1", (co_function)Pe1, 2, s1, s2);
  procPe2 = co_process_create("Pe2", (co_function)Pe2, 1, s2);
}
```

The **co_process_create** function accepts three or more arguments. The first argument must be a pointer to a character string (NULL-terminated) containing a stream name. (This name does not have to match the variable name used within the process itself; it is only used as a label when monitoring the process externally, for example using the CoMonitor Application Monitor.)

The second argument to **co_process_create** is a function pointer (of type **co_function**). This function pointer identifies the specific run function that is to be associated with (or instantiated from) the call to **co_process_create**.

The third argument to **co_process_create** indicates the number of ports (inputs and outputs) that are defined by the process. This number must match the number of actual port arguments that follow. (For example, if the number of ports is 2, then there must be two ports declared as arguments four and five.)

Port arguments specified for an Impulse C run process may be one of these distinct types:

co_stream
co_signal
co_semaphore
co_register
co_memory
co_parameter

Variables of these types must be declared in the configuration function and (with the exception of **co_parameter**) the appropriate **co_xxxxx_create** function must be used to create an instance of the desired communication channel.

When your application creates a process (using the **co_process_create** function), that process is made available for immediate loading and execution on one processing element of the target platform, with little or no overhead required to set up the process. This is a fundamental difference between thread programming and programming in Impulse C: in Impulse C, the entire application and all of its parallel processing components are set up in advance in order to minimize run-time overhead.

Unless otherwise assigned, all processes created using the **co_create_process** routine are assumed to be software processes. Call the co_process_config function to assign a process to a hardware element.

See Also

Processes and How They Communicate
Creating Streams

© 2003-2008 Impulse Accelerated Technologies

Creating and Using Signals
Creating and Using Semaphores
Creating and Using Registers

1.7 Creating Streams

Creating Streams

Streams are used to connect processes, whether hardware or software. The **co_stream_create** function creates a stream, defines its data type and its buffer size, and makes it available for use in subsequent **co_process_create** calls. The following is an example of a stream being created with **co_stream_create**:

```
strm_image_value = co_stream_create("IMG_VAL",INT_TYPE(16),BUFSIZE);
```

There are three arguments to the **co_stream_create** function. The first argument is an optional name that may be assigned to the stream for debugging, external monitoring and post-processing purposes. This name has no semantic meaning in the application, but may be useful for certain downstream synthesis or simulation tools. See your Platform Support Package documentation for additional details. If application monitoring will be used (as indicated by any use of the **cosim_** monitoring functions), stream names are required, and the chosen stream names must be unique across the entire application.

The second argument specifies the type and size of the stream's data element. Macros are provided for defining specific stream types, including INT_TYPE(n), UINT_TYPE(n), CHAR_TYPE, FLOAT_TYPE, and DOUBLE_TYPE.

The third and final argument to **co_stream_create** is the buffer size. This buffer size directly relates to the size of the FIFO buffer that will be created between to processes that are connected with a stream. A buffer size of 1 indicates that the stream is essentially unbuffered; the receiving process will block until the sending process has completed and moved data onto the stream. In contrast, a larger buffer size will result in additional hardware resources (registers and corresponding interconnect resources) being generated, but may result in more efficient process synchronization. As an application designer, you will choose buffer sizes that best meet the requirements of your particular application.

Notes

Stream connections between processes must be one-to-one; broadcast patterns are not supported, and many-to-one connections are not supported. It is possible to create such data distribution patterns in an application, however, by creating intermediate stream collector and distributor processes.

Also note that the size of a stream's buffer (as specified in the call to **co_stream_create**) will have a significant impact on the amount of hardware required to implement the process. You should therefore choose a buffer size that is as small as practical for the given process.

Example

```
#define BUFSIZE 4
void my_config_function(void *arg)
{
  int iterations=(int)arg;
  co_stream host2controller,controller2pe,pe2host;
  co_process host1, host2;
  co_process controller;
  co_process pe;
```

© 2003-2008 Impulse Accelerated Technologies

```
  host2controller=co_stream_create("host2controller",INT_TYPE(32),BUFSIZE);
  controller2pe=co_stream_create("controller2pe",INT_TYPE(32),BUFSIZE);
  pe2host=co_stream_create("pe2host",INT_TYPE(32),BUFSIZE);

  host2=co_process_create("host2",(co_function)host2_run,1,pe2host);
  pe=co_process_create("pe",(co_function)pe1_proc_run,2,controller2pe,pe2host);
  controller=co_process_create("controller",(co_function)controller_run,2,host2con
troller,controller2pe);
  host1=co_process_create("host1",(co_function)host1_run,2,host2controller,iterati
ons);

  co_process_config(controller,co_loc,"PE0");
  co_process_config(pe,co_loc,"PE0");
}
```

See Also

co_stream_create

1.8 Reading and Writing Streams

Stream Inputs and Outputs

Reading and writing of streams is performed within the process run functions that form an application. Each process run function in a dataflow-oriented Impulse C application iteratively reads (polls) one or more input streams and performs the necessary processing when data is available. If there are no data available on the stream being read, the process blocks until such time as data are made available by the upstream process.

Other, non-dataflow processing models are supported, including message-based process synchronization. These alternate models will be described in subsequent sections.

At the start of a process run function, the **co_stream_open** function resets a stream's internal state. The **co_stream_open** function must be used to open all streams (input and output) prior to their being read from or written to. An example of a stream being opened is shown below:

```
err = co_stream_open(input_stream, O_RDONLY, INT_TYPE(32));
```

The **co_stream_open** functions accepts three arguments: the stream (which has previously been declared as a process argument of type **co_stream**), the read/write mode (which may be O_RDONLY or O_WRONLY) and the data type, as expressed using the macros INT_TYPE(n), UINT_TYPE(n), CHAR_TYPE, FLOAT_TYPE, or DOUBLE_TYPE.

If a stream being opened with **co_stream_open** has already been opened, the **co_stream_open** function will return the error code **co_err_already_open**.

When the stream is no longer needed, it may be closed using the **co_stream_close** function:

```
err = co_stream_close(input_stream);
```

The **co_stream_close** function writes an "end-of-stream" (EOS) token to the output stream, which can then be detected by the downstream process when the stream is read using **co_stream_read**.

If a stream being closed is not open (or has already been closed), the **co_stream_close** function will return the error code **co_err_not_open**.

Input Streams

© 2003-2008 Impulse Accelerated Technologies

On an input stream, two operations may be performed: an end-of-stream test and a stream read. The end-of-stream test checks to see whether a "close" operation was performed on the stream by the stream writer. It does this by checking the current element at the head of the stream. If this element is determined to be an "end-of-stream" token, a true value is returned; otherwise a false value is returned indicating that the stream is open.

The **co_stream_read** function attempts to read the next stream element and blocks if the stream is empty. A read operation on a closed stream returns an error condition (**co_err_eos**). Thus the preferred sequence of operations is to describe a stream reading loop as shown below:

```
err = co_stream_open(input_stream, O_RDONLY, INT_TYPE(32));

while(co_stream_read(input_stream, &data, sizeof(co_int32)) == co_err_none) {
  ...    // Process the data here
}
co_stream_close(input_stream);
```

If the stream is closed by the writer (the "upstream" process), **co_err_eos** is returned from **co_stream_read** and any subsequent call to **co_stream_eos** also returns the value **co_err_eos**.

Note: when closing an input stream, all of the unread data in the stream will be flushed out and the EOS token will be consumed. If there is no EOS in the stream (i.e., the writer hasn't closed it yet), ***co_stream_close*** *will block until an EOS is detected. Note that* ***co_stream_close*** *only writes the EOS token when called from the writer process, so it is important not to close a stream from the read side unless the EOS token has been detected.*

Checking for End of Stream

The **co_stream_eos** function returns 0 (false) to indicate that the stream has been closed by the writer and must be closed and reopened before it can be read from again. Once **co_stream_eos** returns false, all subsequent calls to **co_stream_eos** will return false until the reader closes the stream. Similarly, all subsequent calls to **co_stream_read** will fail with **co_err_eos** until the stream is closed and reopened for read.

Non-blocking Stream Reads

At times it may be necessary to check if there is a value available on a stream without blocking (waiting) until a value is available. The **co_stream_read_nb** function is provided for this purpose. The following example, a simple counter with load, uses the **co_stream_read_nb** function for this purpose:

```
while ( 1 ) {
  if (co_stream_read_nb(load,&loadvalue,sizeof(co_int32)))
    nValue=loadvalue;
  else
    nValue += 2;
  co_register_write(count,&nValue,sizeof(co_int32));
}
```

Output Streams

Output streams may be written using the **co_stream_write** function as follows:

```
co_stream_open(output_stream, O_WRONLY, INT_TYPE(32));

for (i=0; i < ARRAYSIZE; i++) {
  co_stream_write(output_stream, &data[i], sizeof(co_int32));
}

co_stream_close(output_stream);
```

© 2003-2008 Impulse Accelerated Technologies

The stream must be a writable stream and the value must be coercible to the stream data type.

Burst Operations

Multiple packets may be read or written across a stream in a single *burst operation*.

Stream burst operations are only supported in software processes. If you use a burst operation in a hardware process, you will see an error message when generating HDL similar to the following:

Error (mand_hw.c:47): Expecting a size value of 4.

To do a burst operation, call **co_stream_read** or **co_stream_write** with a **buffersize** argument equal to a multiple of the stream's packet size, making sure there is enough space in the local **buffer** variable. For example, to write three 16-bit packets:

```
co_int16 buffer[3];

co_stream_open(stream, O_WRONLY, INT_TYPE(16));
co_stream_write(stream, &buffer[0], sizeof(co_int16)*3));
```

If using a stream whose datatype's size is not 8, 16, 32, or 64, please note that the actual packet size used by the stream will be equal to the number of bits in the datatype, rounded up to the size of the nearest standard C type. For datatype sizes larger than 64 bits, the packet size will be rounded up to the nearest byte. Therefore, be sure to use a multiple of the stream packet size when specifying the **buffersize** argument, rather than a multiple of the datatype's size.

For example, the following code will result in only three packets being written, not four as intended:

```
co_int24 buffer[4];  // Implemented as an array of four 32-bit integers

co_stream_open(stream, O_WRONLY, INT_TYPE(24));  // 32-bit packets
co_stream_write(stream, &buffer[0], 3*4);  // 3 bytes (24 bits) * 4 packets == 12
bytes
// Oops!  12 bytes is only enough to send three 32-bit packets.
```

A good rule of thumb is to use the **sizeof** operator. To write four **co_int24** values:

```
co_int24 buffer[4];

co_stream_open(stream, O_WRONLY, INT_TYPE(24));
co_stream_write(stream, &buffer[0], sizeof(co_int24)*4);
```

See Also

co_stream_write
co_stream_write_nb
co_stream_read
co_stream_read_nb
Constraints for Hardware Processes
Efficient Use of Stream Reads in Loops
Stream Macro Interfaces

1.9 Creating and Using Signals

Creating Signals

Signals provide one-to-one process synchronization in Impulse C. To use a signal in an Impulse C application, create a **co_signal** object by calling the **co_signal_create** or **co_signal_create_ex**

© 2003-2008 Impulse Accelerated Technologies

function from your application's configuration function:

```
       void config(void *arg)
       {
              co_signal signal32 = co_signal_create("signal w/32-bit data");
              co_signal signal0 = co_signal_create_ex("signal w/o data",
INT_TYPE(0));
              //...
       }
```

Both functions create a **co_signal** object, which is then passed as an argument to **co_process_create** to establish a connection between processes using the signal. The difference between the two functions in the type of the data associated with the signal:

Function	Data type
co_signal_create	32-bit integer
co_signal_create_ex	Any type, including zero bits (no data)

Using Signals for Process Synchronization

The Impulse C programming model primarily depends upon the buffering of data between processes and on the management of streams. The more efficiently your application manages stream data and the more balanced your application is (in terms of buffer loads and process utilization) the faster it will operate.

There are times, however, when you need more direct control over the starting and stopping of processes and the synchronization of processes to external events. For these times, Impulse C provides alternate methods of synchronization, such as signals and semaphores.

Using signals, you can communicate status information from one process to another. The mechanism is simple, and makes use of a co_signal_post function call in the sending process, and a co_signal_wait function call in the receiving process as shown below:

Process 1:

```
    co_signal_post(start, value);  // Post a signal to process 2
    co_signal_wait(end, &data);    // Wait for process 2 to return a signal
```

Process 2:

```
    co_signal_wait(start, &data);   // Wait for the signal from process 1
    // Do something here, such as accessing shared memory
    co_signal_post(end, 1);         // Post a signal to process 1
```

As shown above, signals are often used to coordinate the operation of multiple processes that must access the same shared resource (such as a shared memory). Signals may also be used to pass values, using the second arguments to the **co_signal_post** and **co_signal_wait** functions.

The **co_signal_wait** function is blocking; the calling process will not continue until a message is received on the specified signal. Note also that repeated calls to **co_signal_post** will overwrite existing message values on the output signal; **co_signal_post** does not wait until an existing (previously-posted) message has been received by the downstream process.

See Also

Processes and How They Communicate

© 2003-2008 Impulse Accelerated Technologies

co_signal_create
co_signal_create_ex
co_signal_post
co_signal_wait

1.10 Creating and Using Semaphores

Creating Semaphores

Semaphores provide one-to-many process synchronization in Impulse C. To use a semaphore in an Impulse C application, create a **co_semaphore** object by calling the **co_semaphore_create** function from your application's configuration function:

```
void config(void *arg)
{
      co_semaphore sema = co_semaphore_create("semaphore0", 1, 2);
      //...
}
```

This function creates a **co_semaphore** object, which is passed as an argument to **co_process_create** to establish a connection between processes sharing the semaphore. More than two processes can use the same semaphore.

Using Semaphores for Process Synchronization

Semaphores are a useful method of multi-process synchronization. Using semaphores, you can manage the operation of multiple processes by setting up a reference count. The mechanism is simple, and makes use of co_semaphore_wait and co_semaphore_release function calls. The state of a semaphore is signaled (meaning a semaphore-controlled code section is allowed to execute) when its count is greater than zero, and nonsignaled when it is zero. The count is decreased by 1 whenever a co_semaphore_wait function is called. The count is increased by 1 when the co_semaphore_release function is called.

```
    co_semaphore_wait(sema);    // Wait for the semaphore to be signaled, then
decrement the count

    . . . // Do something

    co_semaphore_release(sema);  // Release the semaphore, incrementing the
reference count
```

The **co_semaphore_wait** function is blocking; the calling process will not continue until the semaphore reference count is non-zero, indicating that some other process has released the semaphore.

See Also

Processes and How They Communicate
co_semaphore_create
co_semaphore_release
co_semaphore_wait

1.11 Creating and Using Registers

Overview

© 2003-2008 Impulse Accelerated Technologies

The most common and convenient method of communicating between Impulse C processes is to use streams, as described in the sections Creating Streams and Reading and Writing Streams. Since streams are synchronized and support buffering of their data, it is possible to create highly parallel systems using streams, without the need to handle low-level process synchronization, assuming the application being described lends itself to data-centric methods of process synchronization.

There are many applications, however, that require more direct, unsynchronized input and output of data. Some applications may interface directly to hardware devices and their corresponding control signals, while other applications may require that an unsynchronized direct connection be set up between two independent hardware processes.

For this purpose, Impulse C includes the **co_register** data object, which corresponds to a wired connection (input or output) in hardware.

Creating Registers

Like streams and signals, registers are declared and created in the configuration function of your application (using co_register_create) and are passed as register pointers to the application's processes, as in the following example:

```
void config_counter(void *arg)
{
    co_register counter_direction;
    co_register counter_value;
    co_process main_process;
    co_process counter_process;

    counter_direction = co_register_create("counter_direction", UINT_TYPE(1));
    counter_value = co_register_create("counter_value", INT_TYPE(32));

    main_process = co_process_create("main_process", (co_function)counter_main,
                                     2, counter_direction, counter_value);

    counter_process = co_process_create("counter_process",
(co_function)counter_hw,
                                        2, counter_direction, counter_value);
}
```

In this example, two processes are declared and connected via two registers, **counter_direction** and **counter_value**. Register **counter_direction** represents an unbuffered control input to the counter process, while **counter_value** represents the output of the counter process, which is also unbuffered.

Using Registers

Within a process, the functions co_register_get, co_register_put, co_register_read, and co_register_write are used to access the value appearing on a register or to write a value to that register. The following process uses co_register_get and co_register_put to describe a simple 32-bit up/down counter:

```
void counter_hw(co_register direction, co_register count)
{
     int32 nValue = 0;

     /* Main processing loop... */
     while ( 1 ) {
            if (co_register_get(direction) == 1)
                   nValue++;
            else
                   nValue--;
            co_register_put(count, nValue);
```

© 2003-2008 Impulse Accelerated Technologies

```
        }
    }
```

In this process, the variable **nValue** represents a local storage element (a set of clocked hardware registers), while the direction and count register parameters represent inputs and outputs that may be tied directly to device I/O pins or to other hardware elements in the system as a whole.

Controlling Registers From Software Processes

In most cases, the programmer will use registers to communicate only between an application's hardware elements (hardware processes and other, external hardware interfaces). It is possible to interface between hardware and software processes using registers, but this practice is not recommended because the registers are not synchronized in any way—they represent direct wired connections.

You may wish to create software test benches in order to test your application (including the register connections) in desktop simulation, such as within Visual Studio. You can do this with Impulse C, but keep in mind that the order in which processes (and the statements within processes) will run in a desktop operating system environment or debugger can not be guaranteed. The result is that values placed on a register in one process cannot be guaranteed to be available to the destination process unless some other synchronization method (a signal, for example) is used to pause or yield the originating process. You may therefore wish to restrict your use of registers only to hardware processes and, if necessary, create intermediate hardware/software interface processes for the purpose of hardware/software testing.

Sharing Registers

The hardware implementation of registers requires that there be only one process that writes to a given register. Bidirectional registers are not supported in Impulse C, so the hardware compiler will report an error if more than one process writes to the same register, as indicated by calls to either co_register_put or co_register_write. This condition may or may not be detected during software (desktop) simulation.

The same register may be read by multiple processes ("register fan-out").

See Also

Understanding Register Interfaces
Processes and How They Communicate
co_register_create
co_register_read
co_register_write
co_register_get
co_register_put

1.12 Fixed-Point Arithmetic

Overview

Impulse C currently supports fixed-point arithmetic with a combination of datatypes and macros for arithmetic operations, fixed-point formatting, and fixed- to floating-point conversion. In Impulse C, programmers are responsible for keeping track of the fixed-point format in each variable in order to make sense of the variable's value.

Fixed-point applications are most often created by converting a well-tested floating-point

© 2003-2008 Impulse Accelerated Technologies

implementation. This process is time-consuming and involves managing tradeoffs in precision, range, performance, and, in hardware processes, the size of the generated hardware. See the Impulse website (http://www.ImpulseC.com) for an Application Note describing fixed-point arithmetic in more detail and offering a brief overview of some techniques for creating fixed-point programs.

Impulse C Fixed-Point Macros

Impulse C provides support for fixed-point arithmetic in the form of macros and datatypes that allow you to express fixed-point operations in ANSI C and perform computations either as software on an embedded CPU or as hardware modules running in an FPGA's logic.

Impulse C currently supports three fixed-point bit widths (8, 16, and 32 bits) through a combination of datatypes and arithmetic macros. The **co_int8**, **co_int16**, and **co_int32** datatypes are provided for signed fixed-point numbers, while **co_uint8**, **co_uint16**, and **co_uint32** are for unsigned fixed-point numbers. When used with the appropriate class of macros, operands of a given type will be translated by the CoBuilder hardware compiler into a datapath of the same bit width, with two exceptions: for division and multiplication operations, CoBuilder will generate intermediate datapaths twice the size of their operands.

The Impulse C fixed-point macros are defined in the C header file **co_math.h**. Each macro takes two or three arguments: one or two operands of the same fixed-point format (**a** and **b**) and one constant integer (**DW**) whose value is the fractional bit width F_{abc} of the operands and the result. The programmer is responsible for pre-scaling the operands appropriately to prevent overflow or underflow. Macros for formatting fixed-point numbers, converting fixed-point values to floating-point, and performing fixed-point arithmetic are described in the following sections.

Formatting

To convert an integer value to a fixed-point format, use the **FXCONST8**, **FXCONST16**, or **FXCONST32** macros. The result is an unsigned integer (**co_uint8**, **co_uint16**, or **co_uint32**, respectively) with the given fractional bit width; you may cast the result to obtain the desired bit width and sign.

For example:

```
co_int16 a = (co_int16) FXCONST16(96, 7);
// 96 in 1s8.7 format == 0x3000
```

Converting Fixed-point to Floating-point

To convert a fixed-point number to floating-point, use the **FX2REAL32** macro. The result is a single-precision floating-point number (**float**) equal in value to the fixed-point number.

For example:

```
IF_SIM(
co_int32 a = 0x000000F4;  // 15.25 in 1s11.4
float f = FX2REAL32(a, 4);
printf("int: 0x%x, float: %f\n");
// prints "int: 0xF4, float: 15.250000"
)
```

The **FX2REAL32** macro is useful for debugging fixed-point values as they pass through a computation. Note that since floating-point numbers are not supported in hardware processes, this macro will only compile in simulation and in software processes targeting processors with floating-point hardware.

Addition/Subtraction

© 2003-2008 Impulse Accelerated Technologies

The Impulse C macros **FXADD8**, **FXADD16**, and **FXADD32** implement fixed-point addition.

For example:

```
co_int16 a, b, c;
a = 0xFF00;            // -1.0 in 1s7.8
b = 0x0180;            //  1.5 in 1s7.8
c = FXADD16(a, b, 8);  // 0x0080 == 0.5 in 1s7.8

co_uint8 x, y, z;
x = 0xFD;              // 63.25 in 0s6.2
y = 0x02;              //  0.5  in 0s6.2
z = FXADD8(x, y, 2);   // 0xFF == 63.75 in 0s6.2
```

Multiplication

The Impulse C macros **FXMUL8**, **FXMUL16**, and **FXMUL32** implement fixed-point multiplication, rounding the result to the nearest representable number. The macros use a double-precision intermediate datapath and return the low-order half of the result; if 64-bit integers are not supported on your target software platform, then multiplication of 32-bit fixed-width numbers will also not be supported in software.

For example:

```
co_int32 a, b, c;
a = 0x00002000;        // 32.0 in 1s23.8
b = 0x80000080;        // -0.5 in 1s23.8
c = FXMUL32(a, b, 8);  // 0x80001000 == -16.0 in 1s23.8
```

Division

The Impulse C macros **FXDIV8**, **FXDIV16**, and **FXDIV32** implement fixed-point division, rounding the result to the nearest representable number. The macros use a double-precision intermediate datapath and return the low-order half of the result; if 64-bit integers are not supported on your target software platform, then division of 32-bit fixed-width numbers will also not be supported in software.

For example:

```
co_int16 a, b, c;
a = 0x1000;             // 4.0  in 1s5.10
b = 0x0100;             // 0.25 in 1s5.10
c = FXDIV16(a, b, 10);  // 0x4000 == 16.0 in 1s5.10
```

Using Non-Standard Bit Widths

It is possible to use non-standard bit widths for integer and fixed-point operations, but you should keep in mind that desktop simulation will not necessarily match the hardware implementation. This is because all non-standard data types are promoted to the next largest standard type (for example an 8-bit character type, or a 16-bit, 32-bit or 64-bit integer), resulting in different overflow behaviors.

Using 18-bit Multipliers

Because currently-popular FPGAs include 18x18 multipliers, the use of 18-bit data types can be very useful. The following technique can be used to make full use of these multipliers, with no waste of logic to generate a 36-bit result:

```
co_int18 a,b;
co_int36 c;
c = (int64)a * (int64)b;
```

Notice that the two operands **a** and **b** have been casted to larger 64-bit types prior to the multiply

© 2003-2008 Impulse Accelerated Technologies

operation. The subsequent assignment to a 36-bit integer results in the proper bit trimming and the generation of a single 18X18 multiplier element in the FPGA logic.

See Also

Impulse C Datatypes

1.13 Programming for Hardware

Overview

When writing Impulse C processes that will be compiled to hardware using CoBuilder, there are various programming constraints that you must observe, and methods of description that will result in better (or worse) results being generated. This section describes these constraints and programming methods in a general way; additional platform-specific information may be found in your Platform Support Package documentation.

Topics in this section include:

Constraints for Hardware Processes
Efficient Use of Stream Reads in Loops
Understanding Loop Pipelining
Understanding Loop Unrolling
Pipeline Delay and Rate Reports
Avoiding Pipeline Stalls
Efficient Use of the Optimizers
Stream Macro Interfaces
CoBuilder C Pragmas
Hardware Primitive Functions
Using External HDL Hardware Functions
Pointer Support
Hardware Generation Notes

See Also

Tutorial 2: Generating Hardware
Tutorial 5: The CoDeveloper Pro Tools

1.13.1 Constraints for Hardware Processes

Overview

When writing C for hardware compilation, you will need to consider the limitations of the target hardware and the target hardware synthesis tools. CoBuilder will attempt to generate synthesizable (HDL format) outputs based on your C source files, and will attempt to optimize that generated hardware for the specific FPGA platform you are targeting. Due to the limitations inherent in FPGA hardware synthesis, however, there are certain operations that are commonly expressed in C that are difficult or impractical to implement directly in FPGA hardware. An example of such an operation is integer division, which usually expands into an impractical amount of FPGA logic.

Constraints

The following types of C statements and constructs are either not supported, or are minimally supported in FPGA hardware processes:

© 2003-2008 Impulse Accelerated Technologies

No recursion
A hardware process or function may not call itself, either directly or indirectly.

Limited use of function calls
CoBuilder supports limited use of function calls in hardware processes. A hardware process may call only the following types of C functions:

- Impulse C API functions (named co_*)
- Hardware primitive functions, defined using #pragma CO PRIMITIVE
- External HDL functions, defined using #pragma CO IMPLEMENTATION

Pointers must be resolvable at compile time
See the section Pointer Support in Hardware Processes.

Limitations on conditional statements and break
'break' statements must occur only at the end of each case in a 'switch' statement

Limited support for C structs
Structs are supported in C hardware processes, with the following restrictions:

- Structs may not be assigned to other structs
- Structs (or arrays of structs) cannot be used as parameters to co_* functions
- Structs may not be used in global variables/arrays
- Hardware primitive functions cannot have struct parameters or return values
- Structs cannot be autoinitialized
- Structs cannot contain fields that are themselves structures, pointers, or arrays
- Bit fields are not supported
- Pointers to structures are not supported

Unions not supported
The C keyword *union* is not supported in hardware processes.

Integer Math Operations

For integer data types, the standard C arithmetic operators **+**, **-**, *****, **++**, and **--** are supported as single-cycle operations. Division using the **/** operator results in a multi-cycle implementation requiring one cycle for each bit in the result. The algorithm is designed to be low in area and delay, with the goal of increasing the clock speed and system throughput.

The CoBuilder compiler automatically applies strength-reduction optimizations to some integer operations:

- Multiplication (*****) and division (**/**) by two are replaced with bit shift operations
- Modulo (**%**) by powers of two is replaced with logical AND operations

Floating-Point Math Operations

Floating-point arithmetic is supported using standard C types and operators, but only if the selected Platform Support Package provides implementations of the floating-point operations.

See Also

Efficient Use of Stream Reads in Loops
Efficient Use of the Optimizers
CoBuilder C Pragmas
Pointer Support in Hardware Processes
Hardware Generation Notes

© 2003-2008 Impulse Accelerated Technologies

1.13.2 Efficient Use of Stream Reads in Loops

Overview

The efficient processing of stream-related data is a key part of programming using Impulse C. There are several possible methods of reading data from streams, as shown below:

Method 1: while *(preferred, smallest):*

```
while ( co_stream_read(input_stream, &i, sizeof(co_int32) ) == co_err_none ) {
  ...    // Process the data here
}
```

or the following functionally equivalent methods:

```
do {
  if ( co_stream_read(input_stream, &i, sizeof(co_int32)) != co_err_none ) {
     break;
  }
  ...    // Process the data here
} while ( 1 );

// or

while ( 1 ) {
  if ( co_stream_read(input_stream, &i, sizeof(co_int32)) != co_err_none ) {
     break;
  }
  ...    // Process the data here
}
```

Method 2: do-within-if *(preferred, fastest):*

```
if ( co_stream_read(input_stream, &i, sizeof(co_int32)) == co_err_none ) {
  do {
    ...    // Process the data here
  } while( co_stream_read(input_stream, &i, sizeof(co_int32)) == co_err_none );
}
```

Method 3: check EOS and read *(acceptable)*

```
while ( ! co_stream_eos(input_stream) ) {
  if ( co_stream_read(input_stream, &i, sizeof(i)) == co_err_none ) {
    ...
  }
}
```

or the following derivative:

```
do {
  if ( co_stream_read(input_stream, &i, sizeof(co_int32)) == co_err_none ) {
    ...    // Process the data here
  }
} while ( ! co_stream_eos(input_stream) );
```

Method 4: check EOS, not read *(not recommended):*

```
while ( ! co_stream_eos(input_stream) ) {
  co_stream_read(input_stream, &i, sizeof(co_int32));
  ...    // Process the data here
}
```

or the following derivative (which is also *not* recommended):

© 2003-2008 Impulse Accelerated Technologies

```
do {
  co_stream_read(input_stream, &i, sizeof(co_int32));
    ...    // Process the data here
} while ( ! co_stream_eos(input_stream) );
```

As indicated, the first two methods are acceptable, but the fourth may result in problems during simulation and/or in the generated VHDL and should not be used. (The reason? It's possible that between the call to **co_stream_eos** and **co_stream_read**, the stream may be closed by the upstream process. Since Method 4 does not check the return value of **co_stream_read**, the read buffer could contain invalid data.)

Space/Time Optimization

Which method you will use depends on the nature of your application, but there are significant tradeoffs for processes that will be compiled to hardware.

Method	Min. cycles/iteration	Min. cycles to set up	Space required in device
1: while	2	0	least
2: do-within-if	1	1	most, adds one state to the state mach
3: check EOS and read	3	0	more, redundant operation
4: check EOS, not read	n/a	n/a	n/a

The best strategy is almost always the first or second of these four methods. In Method 3, the call to **co_stream_eos** is redundant and will result in an additional cycle, but is shown here for comparison.

© 2003-2008 Impulse Accelerated Technologies

In general, the tradeoffs inherent in the looping methods shown here apply whether stream read return values are used in the loop condition or some other comparison is done. For example, the space required in the device for this loop would be relatively large, since two multipliers are used--one for the if-statement and another for the loop condition:

```
if (i*j<1024) {
      do {
      } while (i*j<1024);
}
```

See Also

co_stream_read
Reading and Writing Streams

1.13.3 Understanding Loop Pipelining

Overview

Pipelining and loop unrolling are two techniques that you can use to increase the parallelism (and therefore the performance) of Impulse C processes that will be compiled to hardware. Both of these techniques (which are controlled by Impulse C pragmas) allow the CoBuilder compiler to take advantage of hardware resources to create parallel structures within the innermost sections of your computationally-intensive C processes.

Pipelining

When pipelining is enabled for inner code loops of your application through the use of the Pipeline pragma, CoBuilder will attempt to parallelize statements appearing within that loop with the goal of reducing the number of instruction cycles required to process the entire pipeline.

Pipelining is an optimization that reduces the number of cycles required to execute a loop by allowing the operations of one iteration to execute in parallel with operations of one or more subsequent iterations. Pipelining is conceptually similar to a manufacturing assembly line where the person at the first station of the assembly line can send a product on to the next station for more assembly while they start working on a second product. In this example, each station performs a portion of the overall assembly work. In a hardware pipeline, the body of a loop containing a sequence of operations is divided up into *stages* that are analogous to the stations of an assembly line. Each stage performs a portion of the loop body. After the first stage has completed its portion of loop iteration **i**, the second stage will begin its portion of iteration **i** while the first stage starts processing iteration **i+1** in parallel.

For example, consider the following loop:

```
while (1) {
  if (co_stream_read(istream,&data,sizeof(data))!=co_err_none) break;
  sum += data;
  co_stream_write(ostream,&sum,sizeof(sum));
}
```

Without pipelining, the body of this loop requires two cycles: the first cycle reads data from the input stream, and the second performs the addition and writes the result to the output stream. This example requires 200 cycles to process 100 inputs. Now, suppose that pipelining was used:

```
 while (1) {
 #pragma CO PIPELINE
  if (co_stream_read(istream,&data,sizeof(data))!=co_err_none) break;
  sum += data;
  co_stream_write(ostream,&sum,sizeof(sum));
```

© 2003-2008 Impulse Accelerated Technologies

```
}
```

This example will result in a pipeline with two stages. The first stage will read data from the input stream, and the second stage will perform the addition and write the result to the output stream, similar to before. In the pipelined version, however, after the first data values is read, stage one will immediately start reading the next input in parallel with the computation and output of the sum using the first input. Over time the execution looks like this:

	Cycle 1	**Cycle 2**	**Cycle 3**	**Cycle 4**	**Cycle 5**
Stage 1	Read data 1	Read data 2	Read data 3	Read data 4	...
Stage 2	...	Add/Output (sum+data 1)	Add/Output (sum+data 2)	Add/Output (sum+data 3)	Add/Output (sum+data 4)

This example now requires only 101 cycles to process 100 inputs, or about one cycle per iteration.

The number of cycles required to complete one iteration of the loop is equal to the number of stages in the pipeline, and is usually called the *latency* of the pipeline. In this example the latency is two.

Pipeline Rate

In some cases it is not possible to perform all stages of a pipeline in parallel. This can occur, for example, if two stages read from the same memory (local variable). As a result, the pipeline will not be able to complete an iteration of the loop every cycle. The pipeline rate is equal to the number of cycles required to complete a single iteration of the loop. For example, if the rate was two then the pipeline would require about two times the number of iterations to complete the loop.

The rate of a pipeline that contains multiple reads of a single array can sometimes be reduced by using multiple arrays. For example, an image stored as rgb values might be implemented in C as:

```
int8 img[16][3];
```

This array has three columns for the red, blue, and green components of each pixel. If **img[i][0]**, **img[i][1]**, and **img[i][2]** all appeared in the loop then the pipeline rate would be increased to permit three reads from the **img** array.

To improve performance, the image array might be re-partitioned as follows:

```
int8 red[16],green[16],blue[16];
```

The loop body will now reference **red[i]**, **green[i]**, and **blue[i]**. Because these are separate arrays they can be read simultaneously and the pipeline rate may be improved.

See Also

Avoiding Pipeline Stalls
Understanding Loop Unrolling
CoBuilder C Pragmas
Efficient Use of the Optimizers

1.13.4 Understanding Loop Unrolling

Overview

If the number of iterations of a loop is known at compile time, then it is possible to unroll the loop to create dedicated logic for each iteration of the loop. Unrolling simply duplicates the body of the loop as

© 2003-2008 Impulse Accelerated Technologies

many times as there are iterations in the loop. When used in conjunction with the "Scalarize array variables" option, unrolling can greatly increase the number of parallel computations performed in one clock cycle.

Unrolling Example

For example, consider the following loop:

```
for (i=0; i<10; i++) {
  sum += A[i];
}
```

Without unrolling, this loop will generate logic to perform each iteration in two cycles. The first cycle will read from memory **A**, and the second cycle will calculate the addition. One adder is generated that will be used ten times during execution of the loop. Now consider the same loop with unrolling.

```
int i;    // Loop index must be type 'int'
...
for (i=0; i<10; i++) { #pragma CO UNROLL
  sum += A[i];
}
```

Unrolling simply duplicates the body of the loop for all values of **i**. In this example, the result is equivalent to the following:

```
sum += A[0];
sum += A[1];
sum += A[2];
sum += A[3];
sum += A[4];
sum += A[5];
sum += A[6];
sum += A[7];
sum += A[8];
sum += A[9];
```

In this case, ten adders will be generated and each one will be used only once during the execution of the loop. Most of the time, as in this case, loop unrolling alone has no specific benefit. Only one value of **A** can be read in a given cycle, so this example loop still requires ten cycles to execute, and ten adders have been generated, which requires a lot of logic. However, if the "Scalarize array variables" option (see Generate Options) is used together with loop unrolling, then the elements of the array are replaced with scalar variables. The results would be then equivalent to the following:

```
sum += A_0;
sum += A_1;
sum += A_2;
sum += A_3;
sum += A_4;
sum += A_5;
sum += A_6;
sum += A_7;
sum += A_8;
sum += A_9;
```

Instead of generating a memory for the array **A**, registers will be generated for each of the ten elements of the array. All ten registers can be read simultaneously, and thus, with both unrolling and array scalarization, this entire loop can be executed in a single cycle.

Scalarizing Arrays

The CoBuilder compiler will attempt to scalarize arrays if the "Scalarize array variables" option is

© 2003-2008 Impulse Accelerated Technologies

enabled in the Impulse C project. Not every array may be scalarizable, however. For example, if any of the following conditions are met, the array cannot be scalarized:

The array is not of type "int"

Only arrays of the C type **int** may be scalarized.

The array is accessed with a non-constant index (after loop unrolling)

For example, array **A** cannot be scalarized in this code:

```
co_int32 A[256];
co_int32 sum;
int i;
for ( i = 0; i < 256; i++ ) {
#pragma CO UNROLL
  sum += A[i];
}
co_stream_write(stream, &A[sum & 0xff], sizeof(co_int32));  // This statement
breaks scalarization
```

Eliminating the statement containing a variable-index array access, however, will allow the compiler to scalarize **A** and thus parallelize the entire loop.

The array is read and written in a single C statement

In this code, **A** cannot be scalarized due to a compiler limitation:

```
A[0]=A[0]+A[1];
```

However, manually inserting a temporary variable will allow **A** to be scalarized:

```
tmp=A[0]+A[1];
A[0]=tmp;
```

#pragma CO NONRECURSIVE is applied to the array

Applying the CO NONRECURSIVE pragma to an array prevents it from being scalarized.

The array is initialized when it is declared

Initializing the values in an array when declaring it will prevent the compiler from scalarizing the array elements.

For example, the following initializer code prevents **A** from being scalarized:

```
int A[] = {0, 1, 2, 4};
```

Assigning the value of each element after declaration will allow **A** to be scalarized:

```
int A[4];
A[0] = 0;
A[1] = 1;
A[2] = 2;
A[3] = 4;
```

Unrolling Tips

Beware of increased logic requirements

© 2003-2008 Impulse Accelerated Technologies

The use of unrolling requires some care because a large amount of logic can be easily generated and the cycle delay can be greatly increased. In the unrolled example above, ten adders will be generated and because the result of each adder is used by the next, this cycle's delay will be the ten times the delay of a single adder. The large delay may severely reduce the design's overall maximum operating frequency.

Loop index must be type int

The loop index must be of type **int**. Use of other types (such as **co_uint8**) as a loop index will result in a "malformed loop" error message being generated by the impulse_prep program.

See Also

Understanding Loop Pipelining
CoBuilder C Pragmas
Generate Options

1.13.5 Pipeline Delay and Rate Reports

Pipeline Results and Optimizer Reports

After it has completed processing your application, the Stage Master optimizer reports information about the generated pipelines, including the number of instruction stages identified, the latency (in terms of clock cycles) required to process the pipeline and the rate at which the pipeline operates (also expressed as clock cycles).

A *stage* is defined as a unit of logic that will operate in one cycle. For example, in the following sequence of instructions:

```
i = 0;
do {
    i++;
} while ( i < 10 );
```

there are two stages that will be generated by the compiler and optimizer. One stage will execute the statement **i = 0**, and the other stage will compute **i++** and **i < 10**. The total execution time for this loop is eleven cycles: one cycle for **i = 0**, and ten cycles to complete the loop, during which **i** is incremented (**i++**) and is checked against the termination value (**i < 10**).

Latency and Rate

The latency and rate numbers reported by Stage Master apply to pipelines.

Latency is the number of cycles required for an input to reach the output of a pipeline, i.e. the length of the pipeline.

The *rate* is the number of cycles required for each input to the pipeline. A rate of 1 means that the pipeline accepts inputs every cycle. A rate of 2 means that the pipeline accepts an input every other cycle.

Stage Delays

All statements within an instruction stage are implemented in a single clock cycle. One implication of this is that the number of individual statements (operations and assignments) may have a direct impact on the maximum clock rate of your application when synthesized to actual hardware. The *stage delay* refers to the maximum number of combinational delays (levels of logic) that are allowed within a given

© 2003-2008 Impulse Accelerated Technologies

pipelined stage. Note that optimizations performed by FPGA synthesis tools may further reduce (or in some cases expand) the number of combinational delays in the final implementation.

Maximum stage delay for a pipeline may be specified using the generic CoBuilder SET pragma:

```
#pragma CO SET stageDelay 32
```

The stage delay for a given stage is calculated as a sum of the unit delays for all the operations in the stage. Unit delays for the arithmetic, logical, relational, and bitwise operators are as follows:

Operator Type	Operators	Unit delay
Bitwise	`& | ^ >> << ~`	1
Bitwise assignment	`&= |= ^=`	1
Arithmetic	`+ - * / %`	Bitwidth of the widest operand
Arithmetic assignment	`+= -= *= /= %= ++ --`	Bitwidth of the widest operand
Arithmetic (unary)	`- +`	Bitwidth of the widest operand
Logical	`&& || !`	Bitwidth of the widest operand
Relational	`< > <= >= == !=`	Bitwidth of the widest operand

For example, the unit delay for this operation is 32:

```
co_int32 result, a;
co_int8 b;
result = a + b;
```

See Also

Understanding Loop Pipelining
Understanding Loop Unrolling

1.13.6 Avoiding Pipeline Stalls

When one process both sends and receives data to and from a pipelined process, the process may stall, waiting for the pipelined process to send its first result. Consider the common case where a software process is both the producer and consumer of a filter process running in hardware. This code snippet shows the inner loop of such a hardware process:

```
while ( co_stream_read(inputStream, &val, sizeof(val)) == co_err_none ) {
#pragma CO PIPELINE
  // Do some work on val
  co_stream_write(outputStream, &val, sizeof(val));
}
```

Both streams are connected to a single software process, which looks like this:

```
while ( moreDataToOutput ) {
  // Produce input to hardware
  co_stream_write(outputStream, &outData, sizeof(outData));
  // Consume result from hardware
  co_stream_read(inputStream, &inData, sizeof(inData));
```

© 2003-2008 Impulse Accelerated Technologies

```
  // do something with inData
}
```

When the hardware process is pipelined, instructions may be scheduled in the following order:

	Cycle 1	**Cycle 2**	**Cycle 3**
Stage 1	Read input 1	Read input 2	Read input 3
Stage 2	n/a	Calculate/Output result 1	Calculate/Output result 2

The software producer/consumer waits to read the first result before it can send the second output, since **co_stream_read** is blocking. Thus, the hardware pipeline will stall in Cycle 2 waiting for more data ("Read input 2"). From the software perspective, the application will hang.

That raises the question: what do we do in this two-process system where the other process is a single loop that both produces data for and consumes the data from the pipeline? How do we avoid the pipeline stall? The answer is the **co_stream_read_nb** primitive.

The **co_stream_read_nb** function is a non-blocking read that will allow the producer to continue sending data while we fill the pipeline, and receive the data as soon has the pipeline starts producing results. Here is the pattern we want to use for the software process in this kind of two-process system:

```
while (moreDataToOutput) {
  // get outData
  co_stream_write(outputStream, &outData, sizeof(outData));
  if (co_stream_read_nb(inputStream, &inData, sizeof(inData))) {
    // do something with inData
  }
}

// read any remaining data from pipeline
while (co_stream_read(inputStream, &inData, sizeof(inData)) == co_err_none) {
  // do something with inData
}
```

The **co_stream_read_nb** function returns true if there was data available. Because the pipeline introduces latency, we will run out of data to send before the pipeline is finished sending the results, so the second loop is needed to read the remaining results as the pipeline is flushed.

This is probably the simplest implementation, but it is not the most efficient. The conditional in the main loop adds overhead to the execution. However, we know that the stalling problem is only an issue as we fill the pipeline. Here's a second pattern that is a little longer but takes the conditional out of the main loop:

```
// startup loop fills the pipeline, exiting once full
do {
  // get outData
  co_stream_write(outputStream, &outData, sizeof(outData));
} while (!co_stream_read_nb(inputStream, &inData, sizeof(inData)));

// this is the main loop operating at full speed
do {
  // do something with inData
  // get outData
  co_stream_write(outputStream, &outData, sizeof(outData));
  co_stream_read(inputStream, &inData, sizeof(inData));
} while (moreDataToOutput);

// read any remaining data from pipeline
do {
```

© 2003-2008 Impulse Accelerated Technologies

```
    // do something with inData
  } while (co_stream_read(inputStream, &inData, sizeof(inData)) == co_err_none);
```

This results in a tight inner loop for the bulk of the execution.

See Also

Understanding Loop Pipelining
co_stream_read_nb

1.13.7 Efficient Use of the Optimizers

Overview

CoDeveloper includes C and RTL optimizers including impulse_porky (used for general C-language optimizations and certain parallel optimizations) and impulse_sm (Stage Master). When writing Impulse C processes, it is important to have a basic understanding of how C code is parallelized by Stage Master so you can achieve the best possible results in terms of execution speed and size of the resulting logic.

The Stage Master optimizer works at the level of individual blocks of C code, such as are found within a process, within a loop body, or within a chain of control statements, such as if-then-else, switch or conditional statements. (Note that control statements such as if-then-else may be parallelized to form a single block automatically by the compiler, or through the use of the FLATTEN pragma.)

For each block of C code, the optimizer will attempt by default to create the minimum number of instruction stages, using instruction pipelining where possible and by scheduling instructions that do not have conflicting data dependencies.

If pipelining is enabled (via the PIPELINE pragma, which is added within the body of a loop and applies only to that loop), then such identified stages will occur in parallel. If there is no pipelining possible, the stages are generated sequentially. Note that all statements within a stage are implemented in combinational logic operating within a single clock cycle.

If you want to exert control over the parallelizing process, you can make use of the co_par_break function to suppress the generation of parallel logic at the level of individual C statements, or to insert additional register stages for timing purposes.

To get maximum benefit from the optimizers, you should keep in mind those types of statements that will result in a new instruction stage being created. These include:

- A control statement such as an **if** test, **switch-case** statement, or conditional (e.g. `a ? b : c`) statement
- A loop
- Any access (read or write) to a memory or array that is already being addressed in the current stage

In the case of a loop, pipelining will attempt to execute multiple iterations of the loop in parallel without duplicating any code. For example, the loop:

```
for (i=0; i<10; i++)
    sum += i << 1;
```

Will result in one stage (with one adder and one shifter) being executed ten times. When not pipelined, the computation time will be at least ten times the sum of the shifter and adder delays, or:

© 2003-2008 Impulse Accelerated Technologies

10 X (delay(shifter) + delay(adder))

If we enable pipelining, however:

```
for (i=0; i<10; i++) {
#pragma CO PIPELINE
    sum += i << 1;
}
```

the result for the same loop is that two stages representing the shifter and adder are executed concurrently, with the computation time being approximately:

10 X max((delay(shifter), delay(adder))

Instruction Scheduling and Assignments

The instruction scheduler that organizes statements within a block or stage will always produce correct results (it will not "break" the logic of your C source code), if necessary by introducing stalls in the pipeline. To minimize such stalling, you should consider where you are making assignments and reduce the number of dependent assignments within a section of code. In some instances you may find that adding one or more intermediate variables to read and store smaller elements of data (such as an array) at the start of a process may result in less stage delay. You should also make assignments to any recursive (self-referencing) variable as early as possible in the body of a loop, and make other references to such variables as late as possible in the loop body so that introduced delays in one stage of the loop do not overly impact later stages.

Note: The CoDeveloper Pro Tools, which include a graphical optimizer and cycle-accurate C debugger, can be useful for analyzing C statements and their parallel optimization to identify opportunities for improved performance.

Impacts of Memory Access

An important consideration when writing your inner code loops for maximum parallelism is to consider data dependencies. In particular, the optimizer will not be capable of parallelizing stages that access the same "bank" of memory (whether expressed as a C array or using the **co_memory_readblock** and **co_memory_writeblock** functions). For this reason you may want to move subregions of a large arrays into local storage (local variables or smaller arrays) before performing multiple, otherwise parallel computations on the local data. Doing so will allow the optimizer to parallelize stages more efficiently, with a small tradeoff of extra assignments that may be required.

Enabling Pipelining

Note that pipelining is not automatic and requires an explicit declaration in your C source code:

```
#pragma CO PIPELINE
```

This declaration must be included within the body of the loop and prior to any statements that are to be pipelined. For example:

```
for (i=0; i<10; i++) {
#pragma CO PIPELINE
    co_stream_read(stream_in1, &nSample1, sizeof(int32));
    co_stream_read(stream_in2, &nSample2, sizeof(int32));
    sum += (nSample2 + nSample2) >> 1;
}
```

Pipeline generation is also controllable via the **StageDelay** parameter. This parameter specifies the maximum delay for the stages of the generated pipeline. **StageDelay** parameters are specified using

© 2003-2008 Impulse Accelerated Technologies

the generic CoBuilder SET pragma, for example:

```
for (i=0; i<10; i++) {
#pragma CO PIPELINE
#pragma CO SET StageDelay 32
    co_stream_read(stream_in1, &nSample1, sizeof(int32));
    co_stream_read(stream_in2, &nSample2, sizeof(int32));
    sum += (nSample2 + nSample2) >> 1;
}
```

Note: the PIPELINE pragma must appear at the top of the loop to be pipelined, before any other statements within the loop, and the loop may not contain any nested loops.

Controlling Stage Delays

It was stated earlier in this section that all statements within a stage are implemented in a single clock cycle. One implication of this is that the number of individual statements (operations and assignments) may have a direct impact on the maximum clock rate of your application when synthesized to actual hardware. Pipeline generation is therefore controllable further via the **StageDelay** parameter. This parameter specifies the maximum delay for the stages of the generated pipeline. **StageDelay** parameters are specified using the generic CoBuilder SET pragma, for example:

```
#pragma CO SET stageDelay 32
```

The stage delay specified using this pragma refers to the maximum number of combinational delays (levels of logic) that are allowed within a given pipelined stage. Note that optimizations performed by FPGA synthesis tools may further reduce (or in some cases expand) the number of combinational delays in the final implementation. Stage delay is calculated as a sum of the unit delays of all the operations in a stage; see Pipeline Delay and Rate Reports.

Using co_par_break to Insert Clock Cycle Boundaries

A second method of controlling the generation of stages and corresponding delays is to use the co_par_break statement. This statement allows you to specify clock cycle boundaries as required to fine-tune the generated logic. The co_par_break function can be used to more precisely control the generation of parallelism. For example, if you want the optimizer to generate four cycles, simply add three co_par_break statements, as in the following example:

```
do {
    // Cycle 1 operations
    co_par_break();
    // Cycle 2 operations
    co_par_break();
    // Cycle 3 operations
    co_par_break();
    // Cycle 4 operations
} while (...);
```

If the optimizer generates more than four cycles for this block of code, that means that some operations in one of the cycles cannot be performed in parallel.

See Also

CoBuilder C Pragmas
co_par_break
Pipeline Delay and Rate Reports

© 2003-2008 Impulse Accelerated Technologies

1.13.8 Stream Macro Interfaces

Overview

The standard method provided for opening, reading, writing, and closing streams is represented by the **co_stream_open**, **co_stream_read**, **co_stream_read_nb**, **co_stream_write**, and **co_stream_close** functions. These functions may be used in hardware or software processes to manage the communication of data across stream interfaces. These stream interfaces are subsequently implemented as buffered channels (FIFOs) on the hardware side of the interface and as low-level (memory-mapped) bus interfaces on the software side.

For hardware processes you should always use these standard functions and observe the guidelines described in the section Efficient Use of Streams.

In software processes, however, you may increase the performance of your stream I/O by replacing the **co_stream**_* function calls, which embody the *procedural interface* to streams, with the following Impulse C macros:

Macro	Corresponding Function Call
`HW_STREAM_OPEN(proc,stream,mode,type)`	`co_stream_open(stream,mode,type)`
`HW_STREAM_READ(proc,stream,var,evar)`	`evar = (co_stream_read(stream,&var,`*`BUS_SIZE`*`) != co_err_none)`
`HW_STREAM_READ_NB(proc,stream,var,evar)`	`evar = (co_stream_read_nb(stream,&var,`*`BUS_SIZE`*`) != 1)`
`HW_STREAM_WRITE(proc,stream,var)`	`co_stream_write(stream,&var,`*`BUS_SIZE`*`)`
`HW_STREAM_CLOSE(proc,stream)`	`co_stream_close(stream)`

The implementation of these macros varies widely from platform to platform, but as a rule, the macro interface:

- Is faster
- Does little or no error-checking
- Performs a single-packet transaction of N bits, where N is the size of the system bus (*BUS_SIZE*)

In general, you should defer the use of these macros until late in the development process in order to preserve hardware/software compatibility for your Impulse C processes. It is quite acceptable to use a mixture of the procedural and macro stream interfaces. *(It is not possible or desirable to use the stream macros in hardware processes.)*

Macro Parameters

The **proc** parameter, used in each of the macros, must take as its argument the name of the software process run function in which the macro is used. For example:

```
void mux_proc(co_stream input1, co_stream input2, co_stream output)
{
      int streamError;
      co_int32 packet;
      ...
      HW_STREAM_READ_NB(mux_proc, input1, packet, streamError);
```

© 2003-2008 Impulse Accelerated Technologies

```
        if ( ! streamError ) {
              // read a packet successfully
        ...
}
```

The **var** parameter's argument should be a variable, not an expression. On some platforms, the implementation of stream macros does not support expressions in the **var** parameter.

The **evar** parameter, used in the two stream read macros, will be assigned the integer value 1 if there was an error or 0 if the macro was successful. Check **evar** instead of evaluating a macro's "return value" to determine whether it was successful. On any given software target, a macro may expand to multiple statements and thus may not compile if used in a conditional (if, while, for, etc.).

Burst Operations

The stream macros are not suited to performing burst operations, in which a single operation sends more than one packet across the stream, depending on the size of an argument. Use the stream procedural interface to do burst reads or writes, or use the stream macros for wide transactions if your chosen platform supports them.

Wide Transactions

Some platforms support wide transactions, in which a piece of data larger than a single bus channel is sent in a single operation. Wide transactions differ from burst operations in that burst operations may send a variable number of packets, while wide transactions send a single fixed-width packet. Macros are provided to support wide transactions, in the following form, where *N* is the number of bytes in the transaction:

Macro	Corresponding Function Call
`HW_STREAM_READ_N(proc,stream,varp,evar)`	`evar = (co_stream_read(stream,varp,N) != co_err_none)`
`HW_STREAM_WRITE_N(proc,stream,varp)`	`co_stream_write(stream,varp,N)`

For cross-platform code compatibility, macros for transactions of a range of fixed sizes may be included with a Platform Support Package even if that platform does not support wide transactions. If wide transactions are not supported natively, then the macros will emulate them by sending multiple requests without checking the stream's status between requests. Use such macros with great care.

See Also

Examples of the use of stream macros can found in your Platform Support Package documentation and in platform-specific sample projects.

Efficient Use of Stream Reads in Loops
Platform Support Package Overview

1.13.9 CoBuilder C Pragmas

Overview

CoBuilder's pipelining and scheduling features may be controlled at the level of your C source code through the use of certain predefined pragmas. All Impulse C and CoBuilder pragmas are identified by the pragma name "CO".

© 2003-2008 Impulse Accelerated Technologies

Scope

Unless otherwise indicated, CoBuilder pragmas have block-level scope. That is, their effects apply to all statements between the pragma declaration and the curly brace ("}") that ends a block of C code.

CO IMPLEMENTATION Pragma

This pragma is used to identify a specific C subroutine as an externally implemented block of hardware. When a function or procedure is tagged with this pragma, the compiler will not attempt to generate hardware for the function or procedure, and will instead generate netlist-level reference to the external hardware, as identified in the second argument to the pragma.

There are three ways in which this pragma can be used. For combinational logic,

```
#pragma CO implementation myfunction logic
```

indicates a combinational hardware implementation with the name "myfunction".

For registered-asynchronous logic (without a deterministic latency),

```
#pragma CO implementation myfunction async
```

indicates an asynchronous hardware implementation with the name "myfunction".

For pipelined logic with a deterministic latency and a pipeline rate of one cycle,

```
#pragma CO implementation myfunction pipeline latency=2
```

indicates a pipelined hardware implementation with a specified latency, in this case two cycles.

See the section Using External HDL Hardware Functions for more details on use of this pragma. For examples of the CO IMPLEMENTATION pragma, see the Impulse C example ExtFunction.

CO FLATTEN Pragma

This pragma instructs the compiler to explicitly flatten the logic for a block. This pragma is particularly useful for flattening control logic generated as a result of case statements or complex if-then-else chains. The following switch statement, for example, requires multiple stages if unflattened, but only one stage if flattened:

```
#pragma CO FLATTEN
    switch (nSel) {
        case 0:
            nResult = (co_uint8) (nSample32 >> 24);
            break;
        case 1:
            nResult = (co_uint8) (nSample32 >> 16);
            break;
        case 2:
            nResult = (co_uint8) (nSample32 >> 8);
            break;
        case 3:
            nResult = (co_uint8) nSample32;
            break;
    }
```

The CO FLATTEN pragma is also useful when creating hardware primitive functions.

Note: The compiler will implicitly flatten control logic when the CO PIPELINE pragma is used.

© 2003-2008 Impulse Accelerated Technologies

CO NONRECURSIVE Pragma

The CO NONRECURSIVE pragma is used to instruct the optimizer that a given array variable is not a recursive variable. When an array is accessed in a pipeline (see the CO PIPELINE pragma), the compiler must schedule access to the array in such a way that the values written in previous pipeline iterations are correctly read in future iterations. There are many situations in which array variables are read and written in a pipeline in such a way that the compiler conservatively considers them to be recursive, even though the written values are never actually read in a recursive manner. In such cases, when you know that array access will be non-recursive, you can increase the performance of your logic by declaring the array variable non-recursive, as follows:

```
int32 A[255];
int32 B[255];
#pragma CO NONRECURSIVE A
#pragma CO NONRECURSIVE B
```

An array is non-recursive in a loop if no array access is made to values computed in a previous iteration. For example:

```
for (i=8; i>=0; i--) {
#pragma CO PIPELINE
    A[i+1] = A[i];
}
```

This example is non-recursive because it does not reference any values assigned in a previous iteration. It is safe, and will improve the pipline's performance, if the CO NONRECURSIVE pragma is applied to the array variable `A`. The compiler will then be able to schedule more array reads and writes in parallel and possibly reduce the number of cycles per output from the pipeline (lower pipeline "rate").

Consider the following recursive example:

```
for (i=1; i<9; i++) {
#pragma CO PIPELINE
    A[i] = A[i-1] + A[i+1]
}
```

In this example, iteration 1 assigns `A[1] = A[0] + A[2]`, and iteration 2 uses that new value of `A[1]` when calculating `A[2] = A[1] + A[3]`. The expression is therefore recursive in fact, and the CO NONRECURSIVE pragma should *not* be used, or else data will be corrupted in the pipeline.

CO PIPELINE Pragma

Pipelining of instructions is not automatic, and requires an explicit declaration in your C source code as follows:

```
#pragma CO PIPELINE
```

This declaration must be included within the body of the loop and prior to any statements that are to be pipelined. For example:

```
for (i=0; i<10; i++) {
#pragma CO PIPELINE
    co_stream_read(stream_in1, &nSample1, sizeof(int32));
    co_stream_read(stream_in2, &nSample2, sizeof(int32));
    sum += (nSample2 + nSample2) >> 1;
}
```

Pipeline generation is also controllable via the **stageDelay** parameter. This parameter specifies the maximum delay for the stages of the generated pipeline. **stageDelay** parameters are specified using

© 2003-2008 Impulse Accelerated Technologies

the generic CoBuilder SET pragma. For example:

```
for (i=0; i<10; i++) {
#pragma CO PIPELINE
#pragma CO SET stageDelay 32
    co_stream_read(stream_in1, &nSample1, sizeof(int32));
    co_stream_read(stream_in2, &nSample2, sizeof(int32));
    sum += (nSample2 + nSample2) >> 1;
}
```

Note: The PIPELINE pragma must appear at the top of the loop to be pipelined, before any other statements within the loop, and the loop may not contain any nested loops. Also note that the PIPELINE pragma will result in additional flattening of logic within the loop, which may dramatically increase the depth of the generated logic. For this reason, use of the ***stageDelay*** *parameter is generally recommended for pipelined loops.*

CO PRIMITIVE Pragma

This pragma is used to define hardware primitive functions that are callable from hardware processes. When applied at the top of a function, the primitive pragma results in an HDL block for that function, as shown below:

```
int32 isqrt(int32 val) {
#pragma CO PRIMITIVE
. . .
}
```

The generated HDL block for the promitive function is instantiated once for every call to that function.

Note that the generated HDL block may be single cycle, or it may represent many cycles. If the function is multi-cycle, the calling hardware can operate in parallel to the called function, but usually waits (stalls) until a result is obtained from the function.

See the section Hardware Primitive Functions for more details.

CO UNROLL Pragma

Loop unrolling may be enabled with the use of the UNROLL pragma, which appears as follows:

```
for (tap = 0; tap < TAPS; tap++) {
#pragma CO UNROLL
    accum += firbuffer[tap] * coef[tap];
}
```

Unrolling a loop will result in that code within the loop being duplicated in hardware as many times as is needed to implement the operation being described. It is therefore important to consider the size of the resulting hardware and unroll loops that have a relatively small number of iterations. The iterations of the loop must also not include interdependent calculations and/or assignments that would prevent the loop from being implemented as a parallel (unrolled) structure in hardware.

Note: The UNROLL pragma must appear at the top of the loop, before any other statements in the loop, and the loop must be a ***for*** *loop with a loop variable of type* ***int*** *and constant bounds.*

See the section Understanding Loop Unrolling for more details.

CO SET Pragma

The general-purpose pragma "CO SET" is used to pass optimization information to the CoBuilder optimizers. The arguments to the pragma are a parameter name and a value.

© 2003-2008 Impulse Accelerated Technologies

Gate Delay Parameters

Two parameters are defined that allow the programmer to influence the estimated gate delay through a single instruction stage:

```
#pragma CO SET defaultDelay 48
#pragma CO SET stageDelay 32
```

where the numeric argument refers to the maximum number of combinational gate delays permissible for an instruction stage. The value specified for **defaultDelay** or **stageDelay** will have a direct impact on the maximum rate at which the resulting logic can be clocked: the lower the value is, the faster the resulting maximum clock rate will be.

The **SET stageDelay** pragma applies only to the current block of C code. The **SET defaultDelay** pragma applies to all blocks of code following the pragma, unless the value is over-ridden by a **SET stageDelay** pragma

*Note: A combinational gate delay in the description above is roughly equivalent to the gate delay in the target hardware. Depending on the capabilities of the synthesis and routing tools being used, a logic operator such as an AND, OR, or SHIFT will require one delay unit, while an arithmetic operation or relational operation may require **n** or more delays, where **n** is the bit width*.

See the section Pipeline Delay and Rate Reports for more details.

Integer Multiplier Generation Parameters

A set of parameters is defined that allows the programmer to choose how integer multipliers are implemented by the Impulse C compiler. These parameters may only be supported by certain Platform Support Packages.

Three parameters ("lut", "dsp", and "pipeline") are defined for each of the two integer multiplication operators, "mul2_s" (signed multiply) and "mul2_u" (unsigned multiply). Set the parameter for a particular operator by appending the parameter to the name of the operator, using the period ('.') as a separator, and enclosing the entire parameter string in double-quotes. For example:

```
#pragma CO SET "mul2_s.pipeline" 1
```

All such parameters can have the value 0 (false) or 1 (true). The default value is 0 (false). Any CO SET pragmas specifying these parameters can be placed in a procedure, loop body, then-branch, or else-branch and remain in effect until the end of that block's scope.

Operator Parameter Values			Multipliers Generated As...
lut	dsp	pipeline	
0	0	0	Default synthesis behavior (logic or DSP blocks)
1	0	0	Logic only
0	1	0	DSP blocks
0	0	1	Pipelined, with DSP blocks

Other combinations of parameter values are not supported.

See Also

© 2003-2008 Impulse Accelerated Technologies

Efficient Use of the Optimizers
Pipeline Delay and Rate Reports

1.13.10 Hardware Primitive Functions

Overview

The CoBuilder compiler supports limited use of function calls in hardware processes. A *hardware primitive function* is a user-defined C function that may be called from a hardware process. When hardware is generated, primitive functions are generated as distinct HDL entities (components) that are invoked from the calling process.

To use a function in a hardware process, the function must be defined in the hardware source file and include the pragma CO PRIMITIVE at the top of the function body. The function may then be called normally within any hardware process. The following constraints apply to hardware primitive functions:

- The return type may be any integer or floating-point type, or void.
- Each parameter may be any integer or floating-point type, a pointer to an integer or floating-point type, or an array of integer or floating-point types.
- Any pointer parameter is treated as an input/output parameter even if the function only uses the pointer in one mode. Care should therefore be taken to avoid the accidental creation of recusive variables in a pipeline. For example, in the following code:

  ```
  do {
      sum2(a++,b++,&res);
  } while (a<10);
  ```

  the **res** variable is considered recursive because its value is read when **sum2** is invoked and it is not defined, even though **sum2** does not read the value of **res**. This can be avoided with the following:

  ```
  do {
     res=0;
     sum2(a++,b++,&res);
  } while (a<10);
  ```

  or the alternative:

  ```
  do {
     res=sum2(a++,b++);
  } while (a<10);
  ```

- Access to arrays using global variables is not permitted.

Array Arguments

Only names of array variables may be passed as arguments to hardware primitives. Pointers or individual array elements may not be passed where arrays are expected.

```
co_uint64 primitive_function(co_uint32 array_param[]);

void hardware_process(co_stream s)
{
     co_uint32 array[256];
     co_uint64 scalar;

     // Supported
     scalar = primitive_function(array);
```

© 2003-2008 Impulse Accelerated Technologies

```
    // NOT SUPPORTED
    scalar = primitive_function(array + 8);
    scalar = primitive_function(&array[0]);
    scalar = primitive_function(&scalar);

    //...
}
```

Array elements may be accessed inside a primitive function, with the usual constraints on use of pointers.

Optimization Hints

The pragma CO FLATTEN may be used at the top of a primitive function to indicate that control logic in the form of an if statement or conditional expression should be "flattened" into combinational logic instead of using a state machine.

For example:

```
int abs(int a, int b) {
#pragma CO PRIMITIVE
    return (a>=b) ? a : b;
}
```

The above function is implemented with a state machine controlled by the condition `a>=b` and requires 2 cycles to complete.

```
int abs(int a, int b) {
#pragma CO PRIMITIVE
#pragma CO FLATTEN
    return (a>=b) ? a : b;
}
```

This function is implemented in a single state and requires only a single cycle due to the use of the CO FLATTEN pragma.

Example Projects

The following projects, found in the Examples directory of the CoDeveloper installation, make use of hardware primitive functions:

- Generic\CordicMath
- Generic\SquareRoot

See Also

Constraints for Hardware Processes
CoBuilder C Pragmas

1.13.11 Using External HDL Hardware Functions

Overview

Custom hardware functions written in VHDL or Verilog can be combined with C by using the CO IMPLEMENTATION pragma. This pragma declares that a specific C subroutine (a function or procedure) has an equivalent HDL implementation, and that the CoBuilder compiler should not attempt to generate hardware from the C code describing the subroutine. This capability allows you to use custom, hand-optimized hardware modules, or existing hardware IP blocks, in your C applications.

© 2003-2008 Impulse Accelerated Technologies

When a C subroutine is decorated with the CO IMPLEMENTATION pragma, the compiler will not attempt to generate hardware for the function or procedure, and will instead generate a netlist-level reference to the external hardware.

If a directory named "userip" exists in an Impulse C project's directory, CoDeveloper's "Export Generated Hardware (HDL)" command will export all HDL files it finds there. This makes "userip" a good place to store the implementation of any external HDL functions used in an Impulse C application.

The CoBuilder compiler supports three types of external HDL functions:

- Combinational logic
- Registered-asynchronous logic (with non-deterministic latency)
- Pipelined logic (with deterministic latency)

Combinational Functions and Procedures

For combinational logic, the pragma

```
#pragma CO implementation myfunction logic
```

appearing at the start of a C function or procedure indicates that there is a corresponding combinational hardware implementation with the name "myfunction". The special identifier "logic" indicates that this is a combinational logic function.

For example, the following C function:

```
co_int8 combFun(co_int4 a, co_int4 b)
{
#pragma CO implementation combFun logic
  co_int8 r=a;
  r=(r<<4)|b;
  return r;
}
```

may have a corresponding VHDL implementation written as:

```
entity combFun is
  port (
    signal a : in std_ulogic_vector(3 downto 0);
    signal b : in std_ulogic_vector(3 downto 0);
    signal r_e_t_u_r_n : out std_ulogic_vector(7 downto 0));
end;

architecture test of combFun is
begin
  r_e_t_u_r_n <= a & b;
end test;
```

In the above example, notice that the return value of the function is given the special name of **r_e_t_u_r_n**, and that the input arguments **a** and **b** are given the same names and bit widths as the C-language equivalent. Notice also that the entity name (in this case **combFun**) matches the name specified in the CO IMPLEMENTATION pragma. This HDL code must be included with your project when running FPGA synthesis, or when performing HDL simulations.

A C procedure that does not include a return value is shown below:

```
void combProc(co_int4 a, co_int4 b, co_int8 *outp)
{
#pragma CO implementation combProc logic
```

© 2003-2008 Impulse Accelerated Technologies

```
  co_int8 r=a;
  r=(r<<4)|b;
  *outp=r;
}
```

The corresponding HDL implementation for this procedure may be written as:

```
entity combProc is
  port (
    signal a : in std_ulogic_vector(3 downto 0);
    signal b : in std_ulogic_vector(3 downto 0);
    signal outp : out std_ulogic_vector(7 downto 0));
end;

architecture test of combProc is
begin
  outp <= a & b;
end test;
```

In this example, notice that the pointer output value of the procedure (***outp**) corresponds to a mode **out** signal of the same name in the corresponding VHDL. Pointer parameters such as this are treated by the compiler as output signals. The types of the signals are always a **std_ulogic_vector** type of the same width as the parameter, or the width of the element type in the case of pointer parameters.

Asynchronous Registered Functions and Procedures

Registered asynchronous hardware components are clocked components that take an indeterminate number of cycles to complete. For such registered asynchronous logic without a deterministic latency, the pragma line

```
#pragma CO implementation myfunction async
```

appearing at the start of a function or procedure indicates an asynchronous hardware implementation with the name "myfunction". The special identifier "async" indicates that this is a registered asynchronous component.

Writing HDL components that include the necessary handshaking is more complex than writing simple combinational components. For example, the C function:

```
co_int32 asyncFun(co_int32 i1)
{
#pragma CO implementation asyncFun async
  return i1;
}
```

may be implemented by the following VHDL entity:

```
entity asyncFun is
  port (
    signal reset : in std_logic;
    signal clk : in std_logic;
    signal request : in std_logic;
    signal i1 : in std_ulogic_vector(31 downto 0);
    signal r_e_t_u_r_n : out std_ulogic_vector(31 downto 0);
    signal acknowledge : out std_logic);
end;

architecture test of asyncFun is
  signal val : std_ulogic_vector(31 downto 0);
  signal count : unsigned(31 downto 0);
  signal done : std_ulogic;
begin
```

© 2003-2008 Impulse Accelerated Technologies

```
  process (clk)
  begin
    if clk'event and clk = '1' then  -- rising clock edge
      if request = '1' then
        val <= i1;
      else
        val <= '0' & val(31 downto 1);
      end if;
    end if;
  end process;

  done <= '1' when val = X"00000000" else '0';

  process (clk)
  begin
    if clk'event and clk = '1' then  -- rising clock edge
      if request = '1' then
        count <= X"00000000";
      elsif done = '0' then
        count <= count + 1;
      end if;
    end if;
  end process;

  r_e_t_u_r_n <= std_ulogic_vector(count);
  acknowledge <= not request and done;
end test;
```

In the above VHDL code, notice that, as in the case of the combinational function, a special identifier **r_e_t_u_r_n** is used to identify the return value of the function. Notice also that the function includes a reset, a clock signal and handshake logic represented by the **request** and **acknowledge** ports. When writing asynchronous registered functions, it is important to follow the handshake prototocol illustrated in this example and to provide corresponding **reset**, **clk**, **request** and **acknowledge** ports.

During operation, the **request** signal will be active high for exactly one cycle to initiate operation. The **acknowledge** signal should become active high when the operation is complete and the output signals are valid. The **acknowledge** signal and any output signals MUST hold their value until the next request. Additionally, the **acknowledge** signal MUST become inactive low during the cycle in which the **request** signal becomes high. In other words, **acknowledge** should always be:

`acknowledge <= NOT request and` [some condition indicating the calculation has completed]`;`

Asynchronous procedures with no return values can also be described, using the same pointer method described above for combinational procedures.

Pipelined Functions and Procedures

For pipelined logic with a deterministic latency and a pipeline rate of one cycle, the line

```
#pragma CO implementation myfunction pipeline latency=2
```

appearing at the start of a function or procedure indicates a pipelined hardware implementation with a specified latency, in this case two cycles. The special identifier "pipeline" indicates that this is a pipelined component with a specific latency.

For example, the C function:

```
co_int32 pipeFun(co_int32 i1)
{
#pragma CO implementation pipeFun pipeline latency=2
  return i1;
```

© 2003-2008 Impulse Accelerated Technologies

```
}
```

Might be described by the following VHDL code:

```
entity pipeFun is
  port (
    signal reset : in std_ulogic;
    signal clk : in std_ulogic;
    signal ce : in std_ulogic;
    signal i1 : in std_ulogic_vector(31 downto 0);
    signal r_e_t_u_r_n : out std_ulogic_vector(31 downto 0));
end;

architecture test of pipeFun is
  signal s1out, s2out : std_ulogic_vector(31 downto 0);
begin

  process (clk)
  begin
    if clk'event and clk = '1' then  -- rising clock edge
      if ce = '1' then
        s1out <= i1(15 downto 0) & i1(31 downto 16);
        s2out <= s1out xor X"ff00ff00";
      end if;
    end if;
  end process;

  r_e_t_u_r_n <= s2out;
end test;
```

As with asynchronous registered components, pipelined components must be provided with a **reset** and **clk** signal, as well as signals for each parameter (in this case the input **i1**). In addition, a **ce** (clock enable) control signal must be provided. The pipeline MUST accept new inputs every active cycle (meaning, every cycle in which **ce** is active high) and corresponding outputs must be provided exactly *N* active cycles later, where *N* is the latency specified in the pragma.

Example Project

The ExtFunction project, in the Examples\Generic\ExtFunction directory of your CoDeveloper installation, illustrates how to use external HDL hardware functions.

See Also

Constraints for Hardware Processes
CoBuilder C Pragmas

1.13.12 Pointer Support in Hardware Processes

Overview

The use of pointers to reference array data in your Impulse C hardware processes is supported, with some limitations. Fundamentally, all uses of pointers must be resolvable at compile time to static references to specific memory locations. Therefore, pointer support in Impulse C hardware processes is limited to the following:

1. <pointer> = & <array element>

   For example,

   ```
   p = &(a[4]);   // Assign address of fifth element
   ```

© 2003-2008 Impulse Accelerated Technologies

2. <pointer> ++

   For example,

```
p++;   // Increment pointer
```

3. <pointer> = <pointer> + <integer expression>

   For example,

```
p = p + 2;    // Pointer addition
```

Also, pointers may not point to more than one array. The following, however, is allowed:

```
p = &(a[2]);
...
p = &(a[3]);
```

The following is not ok:

```
p = &(a[2]);   // This is OK
...
p = &(b[3]);   // This is not OK... can't re-use the pointer for a different
array
```

Using Pointers with Multi-dimensional Arrays

To go through a two dimensional array, rather than using nested loops it is more efficient to write the following:

```
p = &(a[0][0]);
for (i=0; i<4*5; i++) {
  ... // Access the array elements using p
  p++;
}
```

1.13.13 Hardware Generation Notes

Description

When processes written using the Impulse C libraries are compiled to hardware using CoBuilder, HDL design files are generated that may be interfaced to other hardware or software elements through use of a defined stream (or signal, etc.) interface. If you are using a Platform Support Package other than "Generic", these hardware/software interfaces may be generated automatically. If, however, you are interfacing Impulse C hardware processes to other components of your system (such as external HDL hardware elements), you will need to study and understand the basic communication mechanisms used to implement streams and the other Impulse C I/O objects.

CoBuilder uses the following conventions when generating hardware interfaces for streams and signals:

Streams (WRITE mode)

<*stream_name*>_rdy : OUT -- Ready to accept data.
<*stream_name*>_en : IN -- Enable write.
<*stream_name*>_eos: IN -- Write is EOS.
<*stream_name*>_data: IN -- Write data.

© 2003-2008 Impulse Accelerated Technologies

Streams (READ mode)

<*stream_name*>_rdy : OUT -- Data is available.
<*stream_name*>_en : IN -- Enable read.
<*stream_name*>_eos : OUT -- Read is EOS
<*stream_name*>_data : OUT -- Read data.

Signals (WRITE mode)

<*signal_name*>_en : IN -- Enable write.
<*signal_name*>_data: IN -- Write data.

Signals (READ mode)

<*signal_name*>_rdy : OUT -- Signal is posted.
<*signal_name*>_en : IN -- Enable read.
<*signal_name*>_data : OUT -- Read data.

See Also

Platform Support Package Overview
Impulse C Input/Output Interfaces

1.14 Instrumentation and Debugging

Adding Instrumentation to Impulse C Applications

In addition to the intrinsic functions for application programming described in previous chapters, Impulse C also includes a collection of instrumentation functions, prefixed with **cosim**_, that can be used to instrument an Impulse C application for debugging and profiling purposes. Unless otherwise indicated, these instrumentation functions are ignored by the CoBuilder HDL Generator (they have no semantic meaning) and apply only to the desktop simulation environment.

The **cosim**_ instrumentation functions are compatible with all supported desktop compiler environments. When included in your running Impulse C application, these functions communicate with the Impulse C application monitoring application to provide you with additional debugging capabilities that may be used in conjunction with standard desktop debugging environments.

Why use these functions? Standard C debuggers (such as the debugger provided with Visual Studio) are useful for examining control flow in an application, and for examining values of variables within specific processes. For highly parallel applications, however, (applications consisting of many, and perhaps hundreds of inter-related processes) it is useful as well to add debug-related instrumentation directly within the application source code. Software instrumentation can be used to analyze data dependencies, to dynamically view the contents of streams, and to better understand how an application and its component algorithms might be optimized by the application programmer.

Monitoring with Log Windows

Log windows (which appear as child windows of the CoMonitor Application Monitor) are used to organize, format and display messages and other information from within an executing Impulse C application. Any number of log windows may be created. To create and use a log window, you must

1. Call the cosim_logwindow_init function from within the configuration function of your application. (It is only necessary to call this function once, no matter how many log windows you will be creating.)

© 2003-2008 Impulse Accelerated Technologies

2. From within a process run function, declare a variable of type **cosim_logwindow**.
3. Create and name the log window using cosim_logwindow_create. Assign the return value of **cosim_logwindow_create** to the variable declared in step 2.
4. Write to the window any time after creating it by using cosim_logwindow_write or cosim_logwindow_fwrite.

Example

```
////////////////////////////////////////////////////////////////
// Copyright (c) 2003, Impulse Accelerated Technologies, Inc.
// All Rights Reserved.
//
// ex2.c: Example program demonstrating stream usage.
//
// Includes cosim (log window) instrumentation.
//

#include <stdio.h>
#include "co.h"
#include "cosim_log.h"

#define MAX 10

void host1_run(co_stream output_stream, co_parameter iparam)
{
  int iterations=(int) iparam;
  int32 i;

  cosim_logwindow log;
  log = cosim_logwindow_create("host1_run");
  cosim_logwindow_write(log, "Process host1 entered\n");

  cosim_logwindow_write(log, "Process host1 opening stream: output_stream\n");
  co_stream_open(output_stream,O_WRONLY,INT_TYPE(32));

  for(i=0; i<iterations; i++) {
    cosim_logwindow_fwrite(log, "Process host1 writing stream: output_stream with:
%x\n", i);
    co_stream_write(output_stream,&i,sizeof(i));
  }
  cosim_logwindow_write(log, "Prcess host1 closing stream output_stream\n");
  co_stream_close(output_stream);

  cosim_logwindow_write(log, "Process host1 exiting\n");
}

void host2_run(co_stream input_stream)
{
  int32 j;
  FILE *fp;
  char *output_file = "output.dat";

  fp = fopen(output_file,"w");

  cosim_logwindow log;
  log = cosim_logwindow_create("host2_run");
  cosim_logwindow_write(log, "Process host2 entered\n");

  if (fp == NULL) {
    cosim_logwindow_fwrite(log, "Process host2 unable to open output file %s\n",
output_file);
  }
  else {
```

© 2003-2008 Impulse Accelerated Technologies

```
    cosim_logwindow_write(log, "Process host2 opening stream: input_stream\n");
    co_stream_open(input_stream,O_RDONLY, INT_TYPE(32));

    cosim_logwindow_write(log, "Process host2 reading stream: input_stream\n");
    while (!co_stream_eos(input_stream)) {
      co_stream_read(input_stream,&j,sizeof(j));
      cosim_logwindow_fwrite(log, "Process host2 read %08x from stream:
input_stream\n", j);
      fprintf(fp, "%08x\n", (int) j );
    }
    cosim_logwindow_fwrite(log, "Process host2 closing stream input_stream\n");
    co_stream_close(input_stream);
    fclose(fp);
  }
  cosim_logwindow_fwrite(log, "Process host2 exiting\n");
}

void pe1_proc_run(co_stream input_stream, co_stream output_stream)
{
  int32 i;
  uint32 A[4];

  cosim_logwindow log;
  log = cosim_logwindow_create("pe1_proc_run");

  cosim_logwindow_write(log, "Process pe1 opening stream: input_stream\n");
  co_stream_open(input_stream,O_RDONLY, INT_TYPE(32));
  cosim_logwindow_write(log, "Process pe1 opening stream: output_stream\n");
  co_stream_open(output_stream,O_WRONLY, INT_TYPE(32));

  do {
    for (i=0; i<4; i++) {
      if (co_stream_read(input_stream,&A[i],sizeof(uint32))) break;
    }
    cosim_logwindow_fwrite(log,"Process pe1 read
[%08x,%08x,%08x,%08x]\n",A[0],A[1],A[2],A[3]);
    if (i<4) break;
    for (i=0; i<4; i++) {
      co_stream_write(output_stream,&A[i],sizeof(uint32));
    }
    printf(".");    // Action indicator (stdout);
  } while (1);

  co_stream_close(input_stream);
  co_stream_close(output_stream);
}

#define BUFSIZE 4
void config_generic(void *arg)
{
  int iterations=(int)arg;
  co_stream host2pe,pe2host;
  co_process host1, host2;
  co_process controller;
  co_process pe;

  cosim_logwindow_init();

  host2pe=co_stream_create("host2pe",INT_TYPE(32),BUFSIZE);
  pe2host=co_stream_create("pe2host",INT_TYPE(32),BUFSIZE);

  host2=co_process_create("host2",(co_function)host2_run,1,pe2host);
  pe=co_process_create("pe",(co_function)pe1_proc_run,2,host2pe,pe2host);
  host1=co_process_create("host1",(co_function)host1_run,2,host2pe,iterations);

  co_process_config(pe,co_loc,"PE0");
```

© 2003-2008 Impulse Accelerated Technologies

```
}

co_architecture co_initialize(int iterations)
{
  return(co_architecture_create(NULL,"generic hardware",config_generic,(void
*)iterations));
}

int main(int argc, char *argv[]) {   // Arguments are ignored
  int c;
  int iterations = MAX;
  co_architecture firebird;

  printf("Copyright 2003 Impulse Accelerated Technology, Inc.\n");
  printf("See Application Monitor for transcript messages.\n");

  firebird=co_initialize(iterations);
  co_execute(firebird);

  printf("Application ex2 complete. Press Enter to continue...\n");
  c = getc(stdin);

  return(0);
}
```

1.15 Function Reference

Impulse C Application Programming Interface (API)

Impulse C includes a library of predefined functions. These functions, along with two programmer-defined functions, make up the Impulse C API. A programmer uses the Impulse C API to create an application that can be simulated on the desktop and compiled into hardware, or a combination of hardware and embedded software.

The Impulse C API lets a programmer define an application as a set of processes and create communication channels between those processes. Functions related to processes, streams, signals, registers, and shared memories make up the core of the API. Other functions let the programmer influence the hardware generation process and monitor the application during simulation. The C language's computational operators are supplemented in the Impulse C API by four bit-manipulation functions, as well as a variety of macros for common mathematical and fixed-point operations.

Calling Impulse C Functions

The CoDeveloper desktop simulation environment and the CoBuilder hardware compiler enforce certain restrictions in the Impulse C API when compiling or running a simulation, or generating HDL. These restrictions ensure that all parts of an Impulse C application will run properly, whether running as embedded software or as hardware synthesized from generated HDL. The Function Reference page for each Impulse C API function indicates where that function may be called, under the heading "Callable Within".

Function Classes

Process-related functions (prefixed with **co_process**_) create the major processing elements of your application when called within the user-defined configuration function via **co_initialize**. **co_execute** starts the entire Impulse C application.

Stream-related functions (prefixed with **co_stream**_) allow the programmer to exchange buffered, asynchronous data between processes.

Memory-related functions (prefixed by **co_memory**_) allow the programmer to store and access

© 2003-2008 Impulse Accelerated Technologies

shared memory data.

The signal-related functions (**co_signal_post** and **co_signal_wait**) allow the programmer to synchronize processes through the use of messages; one process posts an event, while another process waits for that event to be posted.

The semaphore-related functions **(co_semaphore_wait** and **co_semaphore_release**) provide an alternative method of synchronization useful for multiple processes.

Bit manipulation functions (such as **co_bit_extract**) provide a concise way to insert and extract bits from integers.

Simulation-related functions (prefixed with **cosim**_) allow the programmer to instrument an application and gain quick access to internal application data during desktop simulation.

Predefined Functions

co_architecture_create
co_array_config
co_bit_extract
co_bit_extract_u
co_bit_insert
co_bit_insert_u
co_execute
co_memory_create
co_memory_ptr
co_memory_readblock
co_memory_writeblock
co_par_break
co_port_create
co_process_config
co_process_create
co_register_attach
co_register_create
co_register_get
co_register_put
co_register_read
co_register_write
co_semaphore_create
co_semaphore_release
co_semaphore_wait
co_signal_attach
co_signal_create
co_signal_post
co_signal_wait
co_stream_attach
co_stream_close
co_stream_config
co_stream_create
co_stream_eos
co_stream_open
co_stream_read
co_stream_read_nb
co_stream_write
co_stream_write_nb
cosim_logwindow_create
cosim_logwindow_fwrite

© 2003-2008 Impulse Accelerated Technologies

cosim_logwindow_init
cosim_logwindow_write

User-Defined Functions

co_initialize
Configuration function
Hardware processes
Software processes

See Also

Instrumentation and Debugging
Processes and How They Communicate

1.15.1 co_architecture_create

```
co_architecture co_architecture_create(const char *name, const char *arch,
co_function configure, void *arg);
```

Header File

```
co.h
```

Callable Within

co_initialize

Description

Associates your application with a specific architecture definition and executes the configuration function. Architecture definitions are supplied with CoDeveloper and/or with optional Platform Support Packages.

Parameters

const char *name	A programmer-defined name. This name is used to identify this particular application's use (or instance) of the selected architecture.
const char *arch	This text string specifies which architecture (as supplied with CoDeveloper or with an optional Platform Support Package) is to be used when compiling this application. *This parameter is overridden by the architecture chosen in CoDeveloper's Generate Options dialog.*
co_function configure	The configuration function associated with this architecture. **configure** will be executed by **co_architecture_create**.
void *arg	This parameter will be passed as the argument to the configuration function specified in **configure**.

© 2003-2008 Impulse Accelerated Technologies

Return Value

A pointer to an architecture. This return value may be subsequently passed (in the **main** function of your application) into the **co_execute** function to begin the simulation or execution of your application.

Note

The **name** argument is used as the basis for the name of the top-level HDL module generated by CoBuilder for the application. The name specified must therefore be compatible with any downstream HDL synthesis and simulation tools.

The function will generate an error and terminate the application if the name argument is not provided or the function is called from outside the **co_initialize** function.

1.15.2 co_array_config

```
co_error co_array_config(void *buffer, co_attribute attribute, const char *value);
```

Header File

```
co.h
```

Callable Within

Hardware processes

Description

Configures the hardware implementation of a local array variable.

Parameters

void *buffer	Pointer to the local array variable to be configured.
co_attribute attribute	The attribute to configure; must be **co_kind**.
const char *value	The value to assign to the given attribute; in this case, the kind of memory used to implement the array.

The only supported attribute is **co_kind**, which configures the type of memory used to implement the array. The acceptable values are dependent on the type of Platform Support Package selected:

© 2003-2008 Impulse Accelerated Technologies

Platform Support Package	Value	Type of memory
Generic	"generic" (default)	Asynchronous
Altera	"sync" (default)	ALTSYNCRAM MegaFunction
Xilinx	"dualsync" (default)	Dual-port synchronous
	"sync"	Single-port synchronous block
	"async"	Asynchronous distributed
	"asyncrom"	Asynchronous distributed ROM

Return Value

An error code; always returns **co_err_none** in desktop simulation.

Example

```
co_int32 A[64];
co_array_config(A, co_kind, "sync");
```

1.15.3 co_bit_extract

```
int32 co_bit_extract(int32 value, uint8 start_bit, uint8 num_bits);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

This function provides a concise and efficient method of extracting bits from a 32-bit signed integer.

Parameters

int32 value	A 32-bit signed number to extract bits from.
uint8 start_bit	The starting bit (LSB) of the extracted bit sequence.
uint8 num_bits	The number of bits to extract.

Return Value

Returns a 32-bit signed integer representation of the extracted bits. The least-significant extracted bit is the least-significant bit of the result. The extracted bits are not sign-extended.

© 2003-2008 Impulse Accelerated Technologies

Notes

The **co_bit_insert**, **co_bit_insert_u**, **co_bit_extract** and **co_bit_extract_u** functions operation on 32-bit data. If required, you can truncate the input and results to suit your needs. (Values greater than 32 bits are currently not supported.) For example, if you are operating on 16-bit numbers you can use the functions as follows:

```
co_int16 src = 0xffff;
co_int16 dest;
dest = co_bit_extract(src, 8, 16);
// dest == 0x00ff

dest = co_bit_insert(src, 8, 8, dest);
// dest == 0xffff
```

In this example `src` and `dest` automatically get promoted (and sign-extended, since they are signed values) to 32-bit integers when used as arguments, and the return value gets cast invisibly to a **co_int16**, which just truncates 32 to 16 bits.

The difference between signed and unsigned bit extractions (**co_bit_extract** and **co_bit_extract_u**, respectively) has to do with what's beyond the most-significant bit. If you try to extract 16 bits from signed value 0xffff, starting (halfway) at bit 8, you will get 0xffff. If you extract the same bits from unsigned 0xffffU, you get 0x00ffU, since the sign is not extended beyond the existing 16 bits. Otherwise, the extract functions do the same thing.

See Also

co_bit_insert
co_bit_insert_u
co_bit_extract_u

1.15.4 co_bit_extract_u

```
uint32 co_bit_extract_u(uint32 value, uint8 start_bit, uint8 num_bits);
```

Header File

`co.h`

Callable Within

Hardware and software processes

Description

This function provides a concise and efficient method of extracting bits from a 32-bit unsigned integer.

Parameters

uint32 value	A 32-bit unsigned number to extract bits from.
uint8 start_bit	The starting bit (LSB) of the extracted bit sequence.
uint8 num_bits	The number of bits to extract.

© 2003-2008 Impulse Accelerated Technologies

Return Value

Returns a 32-bit unsigned integer representation of the extracted bits. The least-significant extracted bit is the least-significant bit of the result. The extracted bits are not sign-extended.

Notes

The **co_bit_insert**, **co_bit_insert_u**, **co_bit_extract** and **co_bit_extract_u** functions operation on 32-bit data. If required, you can truncate the input and results to suit your needs. (Values greater than 32 bits are currently not supported.) For example, if you are operating on 16-bit numbers you can use the functions as follows:

```
co_int16 src = 0xffff;
co_int16 dest;
dest = co_bit_extract(src, 8, 16);
// dest == 0x00ff

dest = co_bit_insert(src, 8, 8, dest);
// dest == 0xffff
```

In this example `src` and `dest` automatically get promoted (and sign-extended, since they are signed values) to 32-bit integers when used as arguments, and the return value gets cast invisibly to a **co_int16**, which just truncates 32 to 16 bits.

The difference between signed and unsigned bit extractions (**co_bit_extract** and **co_bit_extract_u**, respectively) has to do with what's beyond the most-significant bit. If you try to extract 16 bits from signed value 0xffff, starting (halfway) at bit 8, you will get 0xffff. If you extract the same bits from unsigned 0xffffU, you get 0x00ffU, since the sign is not extended beyond the existing 16 bits. Otherwise, the extract functions do the same thing.

See Also

co_bit_insert
co_bit_insert_u
co_bit_extract

1.15.5 co_bit_insert

```
int32 co_bit_insert(int32 destination, uint8 dest_start_bit, uint8 num_bits, int32
source);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

This function provides a concise and efficient method of inserting bits into a 32-bit signed integer.

Parameters

© 2003-2008 Impulse Accelerated Technologies

int32 destination	A 32-bit signed number to insert bits into.
uint8 dest_start_bit	The position where the bits will be inserted into **destination**, toward the most-significant bit.
uint8 num_bits	The number of bits to extract from the source, starting at the least-significant bit.
int32 source	The source of the bits to be inserted.

Return Value

Returns a 32-bit signed integer, the result of inserting **num_bits** bits from **source** into **destination**, overwriting existing bit values.

Notes

The **co_bit_insert**, **co_bit_insert_u**, **co_bit_extract** and **co_bit_extract_u** functions operation on 32-bit data. If required, you can truncate the input and results to suit your needs. (Values greater than 32 bits are currently not supported.) For example, if you are operating on 16-bit numbers you can use the functions as follows:

```
co_int16 src = 0xffff;
co_int16 dest;
dest = co_bit_extract(src, 8, 16);
// dest == 0x00ff

dest = co_bit_insert(src, 8, 8, dest);
// dest == 0xffff
```

In this example `src` and `dest` automatically get promoted (and sign-extended, since they are signed values) to 32-bit integers when used as arguments, and the return value gets cast invisibly to a **co_int16**, which just truncates 32 to 16 bits.

The difference between signed and unsigned bit extractions (**co_bit_extract** and **co_bit_extract_u**, respectively) has to do with what's beyond the most-significant bit. If you try to extract 16 bits from signed value 0xffff, starting (halfway) at bit 8, you will get 0xffff. If you extract the same bits from unsigned 0xffffU, you get 0x00ffU, since the sign is not extended beyond the existing 16 bits. Otherwise, the extract functions do the same thing.

See Also

co_bit_insert_u
co_bit_extract
co_bit_extract_u

1.15.6 co_bit_insert_u

```
uint32 co_bit_insert_u(uint32 destination, uint8 dest_start_bit, uint8 num_bits,
uint32 source);
```

Header File

```
co.h
```

© 2003-2008 Impulse Accelerated Technologies

Callable Within

Hardware and software processes

Description

This function provides a concise and efficient method of inserting bits into a 32-bit unsigned integer.

Parameters

uint32 destination	A 32-bit unsigned number to insert bits into.
uint8 dest_start_bit	The position where the bits will be inserted into **destination**, toward the most-significant bit.
uint8 num_bits	The number of bits to extract from the source, starting at the least-significant bit.
uint32 source	The source of the bits to be inserted.

Return Value

Returns a 32-bit unsigned integer, the result of inserting **num_bits** bits from **source** into **destination**, overwriting existing bit values.

Notes

The **co_bit_insert**, **co_bit_insert_u**, **co_bit_extract** and **co_bit_extract_u** functions operation on 32-bit data. If required, you can truncate the input and results to suit your needs. (Values greater than 32 bits are currently not supported.) For example, if you are operating on 16-bit numbers you can use the functions as follows:

```
co_int16 src = 0xffff;
co_int16 dest;
dest = co_bit_extract(src, 8, 16);
// dest == 0x00ff

dest = co_bit_insert(src, 8, 8, dest);
// dest == 0xffff
```

In this example `src` and `dest` automatically get promoted (and sign-extended, since they are signed values) to 32-bit integers when used as arguments, and the return value gets cast invisibly to a **co_int16**, which just truncates 32 to 16 bits.

The difference between signed and unsigned bit extractions (**co_bit_extract** and **co_bit_extract_u**, respectively) has to do with what's beyond the most-significant bit. If you try to extract 16 bits from signed value 0xffff, starting (halfway) at bit 8, you will get 0xffff. If you extract the same bits from unsigned 0xffffU, you get 0x00ffU, since the sign is not extended beyond the existing 16 bits. Otherwise, the extract functions do the same thing.

See Also

co_bit_insert
co_bit_extract
co_bit_extract_u

© 2003-2008 Impulse Accelerated Technologies

1.15.7 co_execute

```
void co_execute(co_architecture architecture);
```

Header File

```
co.h
```

Callable Within

Application code (usually **main**)

Description

This function starts execution of an Impulse C application.

In desktop simulation, all Impulse C processes will begin executing in parallel as threads. **co_execute** will return when all processes have completed (i.e., their threads have exited).

When running on the target platform, **co_execute** will execute the software processes serially, where the order of execution is defined by the order in which **co_process_create** statements appear in the configuration function. When the first software process exits, the next will start, and so on, until all software processes have completed, at which time **co_execute** returns.

On target platforms with an OS that supports threading, the software processes will execute in parallel, like in desktop simulation. See the Platform Support Package documentation for the target platform for information on whether the platform supports threads.

Parameters

co_architecture architecture	A pointer to an architecture previously created using function **co_architecture_create**.

Return Value

None.

Notes

This function must be called from an Impulse C application's **main** function.

See Also

co_process_create

1.15.8 co_initialize

```
co_architecture co_initialize(void *arg);
```

Header File

© 2003-2008 Impulse Accelerated Technologies

None.

Callable Within

Application code (usually **main**)

Description

Entry point for defining and configuring an Impulse C application.

Parameters

Use of the single parameter is defined by the programmer.

Return Value

Should be written to return a pointer to the architecture object returned from a call to **co_architecture_create**.

Notes

This function is not predefined; it must be written by the programmer as part of every Impulse C application. It usually consists of a single function call to **co_architecture_create**, returning the result.

This function is the common entry point for desktop simulation, embedded software execution, and hardware compilation. For simulation and for execution on the target embedded processor, **co_initialize** returns the architecture description so **co_execute** can use it. For this reason, **co_initialize** is called from **main**. The CoBuilder compiler uses the architecture description returned by **co_initialize** as the starting point for the process of generating HDL for an Impulse C application.

1.15.9 co_memory_create

```
co_memory co_memory_create(const char *name, const char *loc, size_t size);
```

Header File

co.h

Callable Within

Configuration function

Description

Creates a shared memory for use in hardware and software processes.

Parameters

© 2003-2008 Impulse Accelerated Technologies

const char *name	A programmer-defined name. This name may be any text string, and is used for identifying the memory externally, for example when using the Application Monitor.
const char *loc	Architecture-specific physical hardware location associated with this memory. Valid values of this argument are platform-dependent. (See your Platform Support Package documentation.)
size_t size	Size of the memory in bytes. This memory will be allocated on the heap during desktop simulation.

Return Value

A pointer to the created memory. This return value may subsequently be used as an argument to function **co_process_create**.

Notes

Memory names, as specified in the **name** argument to **co_memory_create**, must be unique across the application when using the Application Monitor.

1.15.10 co_memory_ptr

```
void *co_memory_ptr(co_memory mem);
```

Header File

```
co.h
```

Callable Within

Software processes

Description

Returns a pointer to the data buffer of a shared memory.

Parameters

co_memory mem	A memory object, as passed to the process on the run function's parameter list.

Return Value

A pointer to the memory buffer. Returns NULL if the **mem** argument is NULL, in which case **co_errno** will be set to CO_ENULL_PTR.

Notes

This function is normally used in software processes to provide a direct means of accessing memory contents for reading and writing. In hardware process, use the functions **co_memory_readblock** and

© 2003-2008 Impulse Accelerated Technologies

co_memory_writeblock.

See Also

co_memory_readblock
co_memory_writeblock

1.15.11 co_memory_readblock

```
void co_memory_readblock(co_memory mem, unsigned int offset, void *buf, size_t
buffersize);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Reads a block of data from a shared memory.

Parameters

co_memory mem	A memory object, as passed on the run function's parameter list.
int offset	The address, in bytes, where the data will be read from the shared memory.
void *buf	A pointer to the local buffer to which data will be transferred.
size_t buffersize	Number of bytes to transfer to the destination buffer in the block read operation.

Return Value

None.

The **co_errno** variable is set to CO_EMEM_OUT_OF_BOUNDS if the **offset** is outside the bounds of the memory. **co_errno** is set to CO_ENULL_PTR if the argument **mem** is NULL.

Examples

```
co_memory mem;
co_int8 data[32];  // May be a local or global variable

// RECOMMENDED: Read all 32 bytes at once into 'data'
co_memory_readblock(mem, 0, data, 32);

// RECOMMENDED: Store bytes starting at an offset in 'data'
co_memory_readblock(mem, 16, &data[16], 16);
```

© 2003-2008 Impulse Accelerated Technologies

```
// ACCEPTABLE: Read each byte one at a time; less efficient
co_int8 datum[1];
int i;
for ( i = 0; i < 32; i++ ) {
    co_memory_readblock(mem, 0, datum, 1);
    data[i] = datum[0];
}
```

Notes

co_memory_readblock performs a block DMA transfer between shared memories and local memories. The third argument, **buf**, is a pointer to an array that represents a block of local RAM. Note that **co_memory_readblock** is not designed for efficient random access of individual memory locations. Note also that stream interfaces may actually provide better hardware performance than memory block reads and writes if the system contains a CPU.

See Also

co_memory_writeblock
co_memory_ptr

1.15.12 co_memory_writeblock

```
void co_memory_writeblock(co_memory mem, unsigned int offset, void *buf, size_t
buffersize);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Writes a block of data to a shared memory.

Parameters

co_memory mem	A memory object, as passed on the run function's parameter list.
unsigned int offset	The address, in bytes, where the data will be written to the shared memory.
void *buf	A pointer to the data source.
size_t buffersize	Number of bytes to transfer from the data source in the block write operation.

Return Value

None.

© 2003-2008 Impulse Accelerated Technologies

The **co_errno** variable is set to CO_EMEM_OUT_OF_BOUNDS if the **offset** is outside the bounds of the memory. **co_errno** is set to CO_ENULL_PTR if the argument **mem** is NULL.

Examples

```
co_memory mem;
co_int8 data[32];

// RECOMMENDED: Write all 32 bytes at once from 'data'
co_memory_writeblock(mem, 0, data, 32);

// RECOMMENDED: Write a subset of the bytes from 'data'
co_memory_writeblock(mem, 16, &data[16], 16);

// ACCEPTABLE: Write each byte one at a time; less efficient
co_int8 datum[1];
int i;
for ( i = 0; i < 32; i++ ) {
    datum[0] = data[i];
    co_memory_writeblock(mem, 0, datum, 1);
}
```

Notes

co_memory_writeblock performs a block DMA transfer between shared memories and local memories. The third argument, **buf**, is a pointer to an array that represents a block of shared memory. The type of memory being accessed will depend on the targeted platform (see your Platform Support Package documentation). Note that **co_memory_writeblock** is not designed for efficient random access of individual memory locations. Note also that stream interfaces may actually provide better performance in hardware than memory block reads and writes if the system contains a CPU.

See Also

co_memory_readblock
co_memory_ptr

1.15.13 co_par_break

```
void co_par_break();
```

Header File

```
co.h
```

Callable Within

Hardware processes

Description

Explicitly specifies a cycle boundary in a block of C code.

Parameters

None.

© 2003-2008 Impulse Accelerated Technologies

Return Value

None.

Notes

When invoked in your C program, this function instructs the optimizer to insert a cycle boundary and suppress the generation of parallel logic. This function can be used to more precisely control the generation of parallelism in your C code. For example, if you want the optimizer to generate four cycles, simply add three **co_par_break** statements, as in the following example:

```
do {
    // Cycle 1 operations
    co_par_break();
    // Cycle 2 operations
    co_par_break();
    // Cycle 3 operations
    co_par_break();
    // Cycle 4 operations
} while (...);
```

If the optimizer generates more than four cycles for this block of code, that means that some operations in one of the cycles cannot be performed in parallel.

1.15.14 co_port_create

```
co_port co_port_create(const char *name, co_port_mode mode, void *io_object);
```

Header File

```
co.h
```

Callable Within

Configuration function

Description

Specify that an Impulse C I/O object, such as a **co_stream**, will be connected to an external data source/sink in hardware.

Parameters

const char *name	The name of the port. This name will be prepended to the individual port names ("*_en", "*_rdy", etc.) in the generated HDL.
co_port_mode mode	The direction ("mode" in stream parlance) of the port. This parameter must be either **co_input** or **co_output**.
void *io_object	The Impulse C I/O object (e.g., **co_stream**, **co_signal**) to create the port for. This object must be connected to a hardware process on the other side of the port.

Return Value

© 2003-2008 Impulse Accelerated Technologies

A port object (unused at this time).

Notes

This function is intended to allow a user to connect one end of an Impulse C stream (or signal, register, etc.) to an HDL component not generated by CoDeveloper. The other end of the I/O object must be connected to an Impulse C hardware process, or the CoBuilder hardware compiler will throw an error.

Example

To specify that a particular I/O object is connected externally (not through an automatically-generated bus interface), you must first create the I/O object as usual, then create *ports* for each object that you wish to connect to an external HDL component. In the following example, the input end of the `datain` stream and the output end of the `dataout` stream will be connected externally, via named ports:

```
void configure(void * arg)
{
       co_stream datain, dataout;
       co_process fpga;
       co_port portin, portout;
       //...

       datain = co_stream_create(...);
       dataout = co_stream_create(...);

       // The input side of 'datain' will be served by external hardware
       portin = co_port_create("input_stream", co_input, datain);

       // The output side of 'dataout' will be served by external hardware
       portout = co_port_create("output_stream", co_output, dataout);

       // This hardware process must read the output of 'datain' and
       // write the input side of 'dataout'
       fpga = co_process_create("fpga", run_fpga, 2, datain, dataout);
       co_process_config(fpga, co_loc, "pe0");

       //
       //    [ External HDL component ]
       //              v
       //              v : 'datain' (co_stream)
       //              v
       //    [ 'fpga' (co_process) ]
       //              v
       //              v : 'dataout' (co_stream)
       //              v
       //    [ External HDL component ]
       //

       //...
}
```

Bus Interfaces and co_ports

By default, any input or output that is declared as a **co_port** type will not be included in generated bus interface wrappers. This is the preferred behavior if you are creating a bus peripheral that requires additional direct (not bus-connected) hardware interfaces.

If you want your **co_port** interfaces to be connected to the bus specified by the Platform Support Package, un-select the "Do not include co_ports in bus interfaces" option in the Generate Options dialog.

© 2003-2008 Impulse Accelerated Technologies

1.15.15 co_process_config

```
co_error co_process_config(co_process process, co_attribute attribute, const char
*val);
```

Header File

```
co.h
```

Callable Within

Configuration function

Description

Configures an attribute of an Impulse C process.

Parameters

co_process process	A pointer to a process created using **co_process_create**.
co_attribute attribute	The attribute to assign a value to. Valid attributes are: • **co_loc**: Hardware location where a process is located.
const char *val	Value to assign to **attribute**.

Return Value

This function always returns **co_err_none**.

Notes

The **co_process_config** function is used to assign certain attributes to Impulse C processes. At the current time, there is only one attribute defined for this function, **co_loc**. The **co_loc** attribute defines the physical hardware location of the specified process. By assigning a value such as "PE0" (processing element 0) to this attribute, you are specifying that the process is to be compiled to hardware in the form of HDL output files. Depending on the capabilities of the target platform, locations other than "PE0" may be supported.

If no **co_process_config** function is specified for a given process, that process is assumed to be a software process and will not be analyzed by the hardware compiler.

1.15.16 co_process_create

```
co_process co_process_create(const char *name, co_function run, int argc, ...);
```

Header File

```
co.h
```

Callable Within

© 2003-2008 Impulse Accelerated Technologies

Configuration function

Description

Creates a process.

Parameters

const char *name	A programmer-defined name. This name may be any text string, and is used for identifying the process externally, for example when using the Application Monitor.
co_function run	A pointer to the run function associated with this process.
int argc	Number of input/output ports (streams, signals, memories, registers, and parameters) associated with the run function. The **argc** argument is followed by a list of port arguments as specified in the run function declaration.

Return Value

A pointer to the created process. This return value may subsequently be used as an argument to function **co_process_config**.

Notes

Processes created using **co_process_create** represent specific *instances* of the specified run function. It is possible (and common) for **co_process_create** to be called repeatedly with the same **run** argument in order to create multiple processes that execute the same run function.

co_process_create prints an error message and terminates the application if it is called from outside the configuration function, or if the Application Monitor is being used and a **name** argument is not supplied (== NULL).

1.15.17 co_register_attach

```
void co_register_attach(co_register reg, void * io, co_stream_direction dir);
```

Header File

```
co.h
```

Callable Within

Configuration function (auto-generated)

Description

Associates a register object with a specific hardware location.

Parameters

© 2003-2008 Impulse Accelerated Technologies

co_register reg	A register object, as passed on the run function's parameter list.
void * io	The address of the register in the FPGA hardware.
co_stream_direction dir	The direction in which data flows through the register, from the hardware process' perspective. Valid values are HW_INPUT and HW_OUTPUT.

Return Value

None.

Notes

Calls to this function are automatically generated by the CoDeveloper compiler in the file **co_init.c**. There is no need to call this function manually; it has no meaning in desktop simulation and will not compile in that context.

The value the **io** parameter will depend on the target platform, as well as the type and number of other Impulse C communication primitives used by an application's hardware processes.

See Also

Creating and Using Registers

1.15.18 co_register_create

```
co_register co_register_create(const char *name, co_type type);
```

Header File

```
co.h
```

Callable Within

Configuration function

Description

Creates a register object.

Parameters

const char *name	A programmer-defined name. This name may be any text string, and is used for identifying the register externally, for example when using the Application Monitor.
co_type type	The data type of the register, normally expressed as a signed or unsigned integer of a specific width using the INT_TYPE(n), UINT_TYPE(n), or CHAR_TYPE macros.

© 2003-2008 Impulse Accelerated Technologies

Return Value

A pointer to the created register. This return value may subsequently be used as an argument to the function **co_process_create**.

Returns NULL if the **type** argument is NULL.

Notes

Register names, as specified in the **name** argument to **co_register_create**, must be unique across the application when using the Application Monitor.

See Also

Creating and Using Registers
Understanding Register Interfaces

1.15.19 co_register_get

```
int32 co_register_get(co_register reg);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Reads the contents of a register as a 32-bit value.

Parameters

co_register reg	A register object, as passed on the run function's parameter list.

Return Value

The value of the register is returned as a 32-bit integer. If the register is wider than 32 bits, the least-significant 32 bits of data are returned as an integer and **co_errno** is set to CO_EINVALID_REGISTER_WIDTH. If the register is smaller than 32 bits, the integer value of the data in the register is returned and **co_errno** is set to CO_ENOERROR.

If the **reg** argument is NULL, the return value is 0 and **co_errno** is set to CO_ENULL_PTR.

See Also

co_register_create
co_register_put
co_register_read
co_register_write

© 2003-2008 Impulse Accelerated Technologies

Creating and Using Registers

1.15.20 co_register_put

```
void co_register_put(co_register reg, int32 value);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Writes a 32-bit data value to a register object.

Parameters

co_register reg	A register object, as passed on the run function's argument list.
int32 value	The value to be assigned to the register.

Return Value

None.

co_errno is set to one the following values if an error occurs in this function:

- CO_ENULL_PTR: The **reg** argument is NULL
- CO_EINVALID_REGISTER_WIDTH: The size of the register is less than 32 bits

Notes

If the register is wider than 32 bits, the data value is written into the least-significant bits of the register and higher-order bits are unchanged. If the register is smaller than 32 bits, the least-significant bits of the value will be written to the register until it is full.

See Also

co_register_create
co_register_put
co_register_read
co_register_write
Creating and Using Registers

1.15.21 co_register_read

```
co_error co_register_read(co_register reg, void *buffer, size_t buffersize);
```

© 2003-2008 Impulse Accelerated Technologies

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Reads a data value from a register object and copies the value to a local memory.

Parameters

co_register reg	A register object, as passed on the run function's argument list.
void *buffer	A pointer to buffer that will receive data from the register. This pointer is typically the address of a local variable.
size_t buffersize	Number of bytes to read from the register into **buffer**. Must be less than or equal to the size of the register (as specified with **co_register_create**).

Return Value

Returns **co_err_none** on success.

Returns **co_invalid_arg** if the **reg** argument is NULL (**co_errno**: CO_ENULL_PTR) or **buffersize** is less than the size of the register, rounded up to the nearest byte (**co_errno**: CO_EINVALID_REGISTER_WIDTH).

Notes

In desktop simulation, the maximum number of bytes stored in **buffer** is equal to the size of the register, rounded up to the nearest byte. For example, reading `sizeof(co_int18)` bytes from a co_register created with `INT_TYPE(18)` will cause three bytes to be written to the buffer, not four. In this case, the error code **co_invalid_arg** is returned, but the data is still stored in **buffer**.

In hardware, exactly the number of bits equal to the size of the register will be stored in the buffer.

See Also

co_register_create
co_register_get
co_register_put
co_register_write
Creating and Using Registers

1.15.22 co_register_write

```
co_error co_register_write(co_register reg, const void *buffer, size_t
buffersize);
```

© 2003-2008 Impulse Accelerated Technologies

Header File

`co.h`

Callable Within

Hardware and software processes

Description

Writes a data value from a local variable to a register object.

Parameters

co_register reg	A register object, as passed on the run function's argument list.
const void *buffer	A pointer to the data source. This source pointer is typically the address of a local variable.
size_t buffersize	Number of bytes to write to the register from **buffer**. Must be less than or equal to the size of the register (as specified with **co_register_create**).

Return Value

Returns **co_err_none** on success.

Returns **co_invalid_arg** if the **reg** argument is NULL (**co_errno**: CO_ENULL_PTR) or **buffersize** is greater than the size of the register, rounded up to the nearest byte (**co_errno**: CO_EINVALID_REGISTER_WIDTH).

Notes

In desktop simulation, the maximum number of bytes stored in the co_register object is equal to the size of the register, rounded up to the nearest byte. For example, writing `sizeof(co_int18)` bytes to a co_register created with `INT_TYPE(18)` will cause three bytes to be written, not four. In this case, the error code **co_invalid_arg** is returned, but the data is still stored in the register.

In hardware, exactly the number of bits equal to the size of the register will be stored.

See Also

co_register_create
co_register_get
co_register_put
co_register_read
Creating and Using Registers

1.15.23 co_semaphore_create

```
co_semaphore co_semaphore_create(const char *name,  int init, int max);
```

© 2003-2008 Impulse Accelerated Technologies

Header File

```
co.h
```

Callable Within

Configuration function

Description

Creates a semaphore, which can be used to synchronize the operation of multiple processes.

Parameters

const char *name	A programmer-defined name. This name may be any text string, and is used for identifying the semaphore externally, for example when using the Application Monitor.
int init	Specifies an initial count for the semaphore. This value must be greater than or equal to zero and less than or equal to the value of max.
int max	Specifies a maximum count for the semaphore.

Return Value

A pointer to the created semaphore. This return value may subsequently be used as an argument to the **co_process_create** function.

Notes

Semaphors are used to synchronize the operation of multiple parallel processes, such as when multiple processes must access the same memory location or I/O device. The state of a semaphore is signaled when its count is greater than zero, and nonsignaled when it is zero. The count is decreased by 1 whenever a **co_semaphore_wait** function is called. The count is increased by 1 by calling the **co_semaphore_release** function.

Semaphore names, as specified in the **name** parameter to **co_semaphore_create**, must be unique across the application when using the Application Monitor. If `NULL` or the empty string ("") is given for this parameter when using the Application Monitor, an error message will be printed and the application will terminate.

The function will print an error message and terminate the application if it is called from outside the configuration function.

See Also

co_semaphore_wait
co_semaphore_release

1.15.24 co_semaphore_wait

```
co_error co_semaphore_wait(co_semaphore sema);
```

© 2003-2008 Impulse Accelerated Technologies

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Waits for a semaphore.

Parameters

co_semaphore sema	A semaphore object, as passed on the run function's argument list.

Return Value

Possible return values are listed below, with the resulting value of **co_errno** shown in parentheses:

co_err_none: Success (CO_ENOERROR).

co_err_invalid_arg: The **sema** argument is NULL (CO_ENULL_PTR).

co_err_unknown: There was an error communicating with the Application Monitor.

Notes

The **co_semaphore_wait** function will block (wait) until the semaphore is available (has a non-zero count).

Semaphors are used to synchronize the operation of multiple parallel processes, such as when multiple processes must access the same memory location or I/O device. The state of a semaphore is signaled when its count is greater than zero, and nonsignaled when it is zero. The count is decreased by 1 whenever a **co_semaphore_wait** function is called. The count is increased by 1 by calling the **co_semaphore_release** function.

See Also

co_semaphore_release

1.15.25 co_semaphore_release

```
co_error co_semaphore_release(co_semaphore sema);
```

Header File

```
co.h
```

Callable Within

© 2003-2008 Impulse Accelerated Technologies

Hardware and software processes

Description

Releases a semaphore.

Parameters

co_semaphore sema	A semaphore object, as passed on the run function's argument list.

Return Value

Possible return values are listed below, with the resulting value of **co_errno** shown in parentheses:

co_err_none: Success (CO_ENOERROR).

co_err_invalid_arg: The **sema** argument is NULL (CO_ENULL_PTR).

co_err_unknown: There was an error communicating with the Application Monitor.

Notes

The **co_semaphore_release** function is non-blocking, and results in the value of the semaphore being incremented by 1.

See Also

co_semaphore_wait

1.15.26 co_signal_attach

```
void co_signal_attach(co_signal signal, void * io, co_stream_direction dir);
```

Header File

```
co.h
```

Callable Within

Configuration function (auto-generated)

Description

Associates a signal object with a specific hardware location.

Parameters

© 2003-2008 Impulse Accelerated Technologies

co_signal signal	A signal object, as passed on the run function's parameter list.
void * io	The base address of the signal in the FPGA hardware.
co_stream_direction dir	The direction in which data flows through the signal, from the hardware process' perspective. Valid values are HW_INPUT and HW_OUTPUT.

Return Value

None.

Notes

Calls to this function are automatically generated by the CoDeveloper compiler in the file **co_init.c**. There is no need to call this function manually; it has no meaning in desktop simulation and will not compile in that context.

The value the **io** parameter will depend on the target platform, as well as the type and number of other Impulse C communication primitives used by an application's hardware processes.

See Also

Creating and Using Signals

1.15.27 co_signal_create

```
co_signal co_signal_create(const char *name);
```

Header File

```
co.h
```

Callable Within

Configuration function

Description

Creates a signal, which can be used to send and receive asynchronous data between processes.

Parameters

const char *name	A programmer-defined name. This name may be any text string, and is used for identifying the signal externally, for example when using the Application Monitor.

Return Value

A pointer to the created signal. This return value may subsequently be used as an argument to the

© 2003-2008 Impulse Accelerated Technologies

co_process_create function.

Notes

Signal names, as specified in the **name** parameter to **co_signal_create**, must be unique across the application when using the Application Monitor. If `NULL` or the empty string ("") is given for this parameter when using the Application Monitor, an error message will be printed and the application will terminate.

The size of the data packet is 32 bits. For alternate (*n*-bit) data types, use the co_signal_create_ex function.

The function will print an error message and terminate the application if it is called from outside the configuration function.

See Also

co_signal_create_ex

1.15.28 co_signal_create_ex

```
co_signal co_signal_create_ex(const char *name, co_type type);
```

Header File

`co.h`

Callable Within

Configuration function

Description

Creates an n-bit signal, which can be used to send and receive asynchronous data between processes.

Parameters

const char *name	A programmer-defined name. This name may be any text string, and is used for identifying the signal externally, for example when using the Application Monitor.
co_type type	The type of the signal's data packet, including its width. Macros are provided for commonly-used types: INT_TYPE(n), UINT_TYPE(n) and CHAR_TYPE.

Return Value

A pointer to the created signal. This return value may subsequently be used as an argument to the **co_process_create** function.

Notes

© 2003-2008 Impulse Accelerated Technologies

Signal names, as specified in the **name** parameter to **co_signal_create**, must be unique across the application when using the Application Monitor. If `NULL` or the empty string ("") is given for this parameter when using the Application Monitor, an error message will be printed and the application will terminate.

The function will print an error message and terminate the application if it is called from outside the configuration function.

See Also

co_signal_create

1.15.29 co_signal_post

```
co_error co_signal_post(co_signal signal, int32 value);
```

Header File

`co.h`

Callable Within

Hardware and software processes

Description

Posts a message (a 32-bit value) on a signal.

Parameters

co_signal signal	A signal object, as passed on the run function's argument list.
int32 value	The integer value to be posted.

Return Value

Possible return values are listed below, with the resulting value of **co_errno** shown in parentheses:

co_err_none: Success (CO_ENOERROR).

co_err_invalid_arg: The **signal** argument is NULL (CO_ENULL_PTR).

co_err_unknown: There was an error communicating with the Application Monitor.

Notes

The **co_signal_post** function is non-blocking and will overwrite any message that is already pending on the signal.

Signals are unidirectional. A process may either post to a signal or wait on a signal, but not both. If processes must exchange signals in both directions, simply use a pair of signals in each process, one for each direction. If a signal is posted to and waited on by the same process, an error message will be printed and the application will terminate.

© 2003-2008 Impulse Accelerated Technologies

See Also

co_signal_wait

1.15.30 co_signal_wait

```
co_error co_signal_wait(co_signal signal, int32 *ip);
```

Header File

co.h

Callable Within

Hardware and software processes

Description

Waits for a message on a signal.

Parameters

co_signal signal	A signal object, as passed on the run function's argument list.
int32 *ip	A pointer to a local variable that will receive the message value.

Return Value

Possible return values are listed below, with the resulting value of **co_errno** shown in parentheses:

co_err_none: Success (CO_ENOERROR).

co_err_invalid_arg: The **signal** argument is NULL (CO_ENULL_PTR).

co_err_unknown: There was an error communicating with the Application Monitor.

Notes

The **co_signal_wait** function will block (wait) until a message has been posted to the signal by another process using **co_signal_post**.

Any number of processes can wait on a given signal, but only one process will receive and consume the signal and its value. The first process to receive the signal will consume it, and any other waiting processes will continue to wait until another signal is posted.

Signals are unidirectional. A process may either post to a signal or wait on a signal, but not both. If processes must exchange signal data in both directions, simply use a pair of signals in each process, one for each direction. If a signal is posted to and waited on by the same process, an error message will be printed and the application will terminate.

See Also

© 2003-2008 Impulse Accelerated Technologies

co_signal_post

1.15.31 co_stream_attach

```
void co_stream_attach(co_stream stream, void * io, co_stream_direction dir);
```

Header File

```
co.h
```

Callable Within

Configuration function (auto-generated)

Description

Associates a stream object with a specific hardware location.

Parameters

co_stream stream	A stream object, as passed on the run function's parameter list.
void * io	The base address of the stream in the FPGA hardware.
co_stream_direction dir	The direction in which data flows through the stream, from the hardware process' perspective. Valid values are HW_INPUT and HW_OUTPUT.

Return Value

None.

Notes

Calls to this function are automatically generated by the CoDeveloper compiler in the file **co_init.c**. There is no need to call this function manually; it has no meaning in desktop simulation and will not compile in that context.

The value the **io** parameter will depend on the target platform, as well as the type and number of other Impulse C communication primitives used by an application's hardware processes.

See Also

Creating Streams

1.15.32 co_stream_close

```
co_error co_stream_close(co_stream stream);
```

Header File

© 2003-2008 Impulse Accelerated Technologies

```
co.h
```

Callable Within

Hardware and software processes

Description

Closes a previously opened stream.

Parameters

co_stream stream	A stream object, as passed in the run function's argument list.

Return Value

Possible return values are listed below, with values of **co_errno** shown in parentheses:

co_err_none: Success (CO_ENOERROR)

co_err_invalid_arg: Stream argument is NULL (CO_ENULL_PTR)

co_err_not_open: Stream is not open (CO_ENOT_OPEN)

Note

Closing an input stream flushes any unread data from the stream and consumes one end-of-stream token. An attempt to close an input stream that has not yet received an end-of-stream token will block until the writer closes the stream.

1.15.33 co_stream_config

```
co_error co_stream_config(co_stream stream, co_attribute attribute, const char
*val);
```

Header File

```
co.h
```

Callable Within

Configuration function

Description

Configures a stream for specific hardware characteristics and/or location.

Parameters

© 2003-2008 Impulse Accelerated Technologies

co_stream stream	A stream object, as returned by co_stream_create().
co_attribute attribute	The attribute to assign a value to. Valid attributes are: • **co_kind**: Type of hardware stream.
const char *val	Value to assign to **attribute**.

Return Value

This function always returns **co_err_none**.

Notes

The **co_stream_config** function is used to assign certain attributes to Impulse C streams. At the current time, there is only one attribute defined for this function, **co_kind**. The **co_kind** attribute defines the physical attributes of the specified stream. By assigning a value such as "bram" (block RAM) to this attribute, you are specifying that the stream is to be implementing using hardware FIFOs in block RAM. (The default stream hardware implementation is distributed RAM.) Depending on the capabilities of the target platform, locations other than "bram" may be supported.

1.15.34 co_stream_create

```
co_stream co_stream_create(const char *name, co_type type, int numelements);
```

Header File

```
co.h
```

Callable Within

Configuration function

Description

Creates a stream, which can be used to send and receive buffered, asynchronous data between processes.

Parameters

const char *name	A programmer-defined name. This name may be any text string, and is used for identifying the stream externally, for example when using the Application Monitor.
co_type type	The type of the stream's data packets, including their width. Macros are provided for commonly-used types: INT_TYPE(n), UINT_TYPE(n), CHAR_TYPE, FLOAT_TYPE, and DOUBLE_TYPE.
int numelements	The size of the stream buffer in packets.

Return Value

© 2003-2008 Impulse Accelerated Technologies

A pointer to the created stream. This return value may subsequently be used as an argument to function **co_process_create**.

Notes

Streams are implemented using first-in-first-out (FIFO) buffers. The size of this buffer in bytes will be the value of the **numelements** argument multiplied by the size (in bytes) of the data type specified in the **type** argument. *Note that the size of a stream may have a dramatic impact on the size of the resulting logic after compilation to hardware.*

Stream names, as specified in the **name** argument to **co_stream_create**, must be unique across the application when using the Application Monitor.

co_stream_create prints an error message and terminates the application if:

- It is called from outside the configuration function, or
- The Application Monitor is being used and a **name** argument is not supplied (== NULL), or
- A **type** wider than 128 bits is specified

1.15.35 co_stream_eos

```
int co_stream_eos(co_stream stream);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Checks whether a stream is at the end-of-stream condition. End-of-stream occurs when a process that has finished writing data to a stream closes it with **co_stream_close**.

Parameters

co_stream stream	A stream object, as passed in the run function's argument list.

Return Value

Returns 0 (false) if there is no end-of-stream detected or 1 (true) if there is an end-of-stream.

Notes

End-of-stream is detected only when *all* of the following conditions are met:

1. The stream is open for read (O_RDONLY).
2. Data have been written to the stream by another process.
3. All data have been read from the stream.
4. The stream is closed by the writer process with **co_stream_close**.

© 2003-2008 Impulse Accelerated Technologies

When **co_stream_eos** returns 1 (true), subsequent calls to **co_stream_eos** will continue to return 1 until the stream is closed by the reader.

1.15.36 co_stream_open

```
co_error co_stream_open(co_stream stream, mode_t mode, co_type type);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Opens a stream for reading or writing.

Parameters

co_stream stream	A stream object, as passed in the run function's argument list.
mode_t mode	The read/write mode of the stream. Valid modes are O_RDONLY and O_WRONLY. (There are no bidirectional streams.)
co_type type	The data type of the stream. The following macros are provided to specify the type: INT_TYPE(n), UINT_TYPE(n), CHAR_TYPE, FLOAT_TYPE, and DOUBLE_TYPE. This argument must be the same as the type used when creating the stream with **co_stream_create**.

Return Value

Possible return values listed below, with the resulting value of **co_errno** shown in parentheses:

co_err_none: Success (CO_ENOERROR).

co_err_invalid_arg: The **stream** argument is NULL (CO_ENULL_PTR) or the **mode** argument is neither of O_RDONLY or O_WRONLY (CO_EINVALID_MODE).

co_err_already_open: The stream is already open in this mode (CO_EALREADY_OPEN).

Notes

This function prints an error message and terminates the application if the **type** argument is different than the type specified when the stream was created with **co_stream_create**.

1.15.37 co_stream_read

```
co_error co_stream_read(co_stream stream, void *buffer, size_t buffersize);
```

© 2003-2008 Impulse Accelerated Technologies

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Reads data packets from a previously opened stream.

Parameters

co_stream stream	A stream object, as passed in the run function's argument list.
void *buffer	A pointer to the buffer that will receive stream data. This pointer is typically the address of a local variable or the address of the first element of an array. A number of bytes, equal to **buffersize**, will be read into this buffer from the stream.
size_t buffersize	Size of the buffer receiving data, in bytes. Must be an integer multiple of the packet width of the stream (and greater than zero).

Return Value

Possible return values are listed below, with the corresponding value of **co_errno** shown in parentheses:

co_err_none: Success (CO_ENOERROR).

co_err_eos: Encountered an end-of-stream condition (CO_EEOS)

co_err_invalid_arg:
stream or **buffer** argument is NULL (CO_ENULL_PTR), or
buffersize argument is smaller than the packet width of the stream, in bytes (CO_EINVALID_STREAM_WIDTH), or
buffersize argument is not an integer multiple of the packet width of the stream, in bytes (CO_EUNALIGNED_STREAM_REQUEST)

co_err_not_open: Stream is not open for read (CO_ENOT_OPEN).

Burst Operations

Burst operations are only supported in *software processes*. Hardware generation will fail if hardware processes attempt to use burst operations.

If the **buffersize** argument is larger than the size of one packet, a *burst operation* will be performed. In desktop simulation, multiple operations will be performed to move all the data across the stream, one packet at a time.

If all of the data requested is not available, **co_stream_read** will block until all data has been read.

In a software process running on an embedded processor, burst operations may be able to be

© 2003-2008 Impulse Accelerated Technologies

performed in a single transaction, depending on the characteristics of the system bus and the size of the stream. In general, the Impulse C stream driver will attempt to send as much data as possible per transaction. See the documentation for your chosen Platform Support Package for details and performance information specific to that platform.

Notes

The **co_stream_read** function must be used only on streams that have been opened with mode O_RDONLY.

The **co_stream_read** function will block (wait) if the input stream is empty.

1.15.38 co_stream_read_nb

```
int co_stream_read_nb(co_stream stream, void *buffer, size_t buffersize);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Reads data packets from a previously opened stream, but does not block (wait) for a value to be available on the stream.

Parameters

co_stream stream	A pointer to a stream as passed in the run function's argument list.
void *buffer	A pointer to the buffer that will receive stream data. This pointer is typically the address of a local variable or the address of the first element of an array. A number of bytes, equal to **buffersize**, will be read into this buffer from the stream. If no data is available on the stream, the buffer pointed to by this argument is unchanged.
size_t buffersize	Size of the buffer receiving data, in bytes. Must be an integer multiple of the packet width of the stream (and greater than zero).

Return Value

Returns 1 if data was available on the stream, or 0 if no data was available. In these cases, **co_errno** is set according to the following conditions:

CO_ENOERROR: No error

CO_EEOS: Encountered an end-of-stream condition

© 2003-2008 Impulse Accelerated Technologies

Returns 0 if an error occurred. Error conditions are listed below, with the corresponding value of **co_errno**:

CO_ENULL_PTR: stream or **buffer** argument is NULL

CO_EINVALID_STREAM_WIDTH: buffersize argument is smaller than the packet width of the stream, in bytes

CO_EUNALIGNED_STREAM_REQUEST: buffersize argument is not an integer multiple of the packet width of the stream, in bytes

CO_ENOT_OPEN: Stream is not open for read (O_RDONLY)

Burst Operations

Burst operations are only supported in *software processes*. Hardware generation will fail if hardware processes attempt to use burst operations.

If the **buffersize** argument is larger than the size of one packet, a *burst operation* will be performed. In desktop simulation, multiple operations will be performed to move all the data across the stream, one packet at a time.

If all of the data requested is not available, **co_stream_read_nb** will return 0. Only those packets that were available will be stored in **buffer**. It is therefore recommended that **co_stream_read_nb** be used for burst read operations only if you can guarantee that all the data will be available for reading.

In a software process running on an embedded processor, burst operations may be able to be performed in a single transaction, depending on the characteristics of the system bus and the size of the stream. In general, the Impulse C stream driver will attempt to send as much data as possible per transaction. See the documentation for your chosen Platform Support Package for details and performance information specific to that platform.

Notes

The **co_stream_read_nb** function must be used only on streams that have been opened with mode O_RDONLY.

The **co_stream_read_nb** function will *not* block (wait) if the input stream is empty. If no data is available on the stream, the function returns immediately.

1.15.39 co_stream_write

```
co_error co_stream_write(co_stream stream, const void *buffer, size_t buffersize);
```

Header File

```
co.h
```

Callable Within

Hardware and software processes

Description

Writes data packets to a previously opened stream.

© 2003-2008 Impulse Accelerated Technologies

Parameters

co_stream stream	A stream object, as passed in the run function's argument list.
void *buffer	A pointer to the data source. This source pointer is typically the address of a local variable or the address of the first element of an array. A number of bytes, equal to **buffersize**, will be written from this buffer to the stream.
size_t buffersize	Size of the data source, in bytes. Must be an integer multiple of the packet width of the stream (and greater than zero).

Return Value

Return values are listed below. Corresponding values of **co_errno** in parentheses.

co_err_none: success (CO_ENOERROR).

co_err_eos: encountered an end-of-stream condition (CO_EEOS)

co_err_invalid_arg:
stream or **buffer** argument is NULL (CO_ENULL_PTR), or
buffersize argument is smaller than the packet width of the stream in bytes (CO_EINVALID_STREAM_WIDTH), or
buffersize argument is not an integer multiple of the packet width of the stream, in bytes (CO_EUNALIGNED_STREAM_REQUEST)

co_err_not_open: stream is not open for write (CO_ENOT_OPEN).

Burst Operations

Burst operations are only supported in *software processes*. Hardware generation will fail if hardware processes attempt to use burst operations.

If the **buffersize** argument is larger than the size of one packet, a *burst operation* will be performed. In desktop simulation, multiple operations will be performed to move all the data across the stream, one packet at a time.

If the stream FIFO fills up before all the data is written, **co_stream_write** will block until space in the FIFO becomes available.

In a software process running on an embedded processor, burst operations may be able to be performed in a single transaction, depending on the characteristics of the system bus and the size of the stream. In general, the Impulse C stream driver will attempt to send as much data as possible per transaction. See the documentation for your chosen Platform Support Package for details and performance information specific to that platform.

Notes

The **co_stream_write** function must be used only on streams that have been opened with mode O_WRONLY.

The **co_stream_write** function will block (wait) if the output stream is already full.

© 2003-2008 Impulse Accelerated Technologies

1.15.40 co_stream_write_nb

```
int co_stream_write_nb(co_stream stream, void *buffer, size_t buffersize);
```

Header File

```
co.h
```

Callable Within

Software processes

Description

Writes a data value (packet) to a previously opened stream, but does not block (wait) for the stream to accept the data.

This function is only available in software processes. It may not be supported on all target software platforms.

Parameters

co_stream stream	A stream object, as passed in the run function's argument list.
void *buffer	A pointer to the data source. This source pointer is typically the address of a local variable or the address of the first element of an array. A number of bytes, equal to **buffersize**, will be written from this buffer to the stream.
size_t buffersize	Size of the data source, in bytes. Must be an integer multiple of the packet width of the stream (and greater than zero).

Return Value

Returns 1 if the stream accepted the packet, or 0 if the stream was unavailable. In these cases, **co_errno** is set according to the following conditions:

CO_ENOERROR: no error

Return 0 in case of error. Error conditions are listed below, with the corresponding value of **co_errno**:

CO_ENULL_PTR: stream or **buffer** argument is NULL

CO_EINVALID_STREAM_WIDTH: buffersize argument is smaller than the packet width of the stream in bytes

CO_EUNALIGNED_STREAM_REQUEST: buffersize argument is not an integer multiple of the packet width of the stream, in bytes

CO_ENOT_OPEN: Stream is not open for write (O_WRONLY)

Burst Operations

© 2003-2008 Impulse Accelerated Technologies

Burst operations are only supported in *software processes*. Hardware generation will fail if hardware processes attempt to use burst operations.

If the **buffersize** argument is larger than the size of one packet, a *burst operation* will be performed. In desktop simulation, multiple operations will be performed to move all the data across the stream, one packet at a time.

If all of the data cannot be written immediately, **co_stream_write_nb** will return 0. Only those packets that could be written will be. It is therefore recommended that **co_stream_write_nb** be used for burst write operations only if you can guarantee that all the data can be sent before the stream FIFO fills up.

In a software process running on an embedded processor, burst operations may be able to be performed in a single transaction, depending on the characteristics of the system bus and the size of the stream. In general, the Impulse C stream driver will attempt to send as much data as possible per transaction. See the documentation for your chosen Platform Support Package for details and performance information specific to that platform.

Notes

The **co_stream_write_nb** function must be used only on streams that have been opened with mode O_WRONLY.

The **co_stream_write_nb** function will *not* block (wait) if the output stream is full. If the stream is unavailable (i.e., because the FIFO is full), the function returns immediately and the data is discarded.

Including this function in a hardware process will cause CoBuilder to terminate and HDL generation will fail.

1.15.41 cosim_logwindow_create

```
cosim_logwindow cosim_logwindow_create(const char * name);
```

Header File

```
cosim_log.h
```

Callable Within

Any application code

Description

This function creates an Application Monitor log window.

Parameters

const char *name	A programmer-defined name. This name may be any text string, and is used for visually identifying a specific log window when using the Application Monitor.

Return Value

A pointer to the created log window. This return value may subsequently be used as an argument to

© 2003-2008 Impulse Accelerated Technologies

the functions **cosim_logwindow_write** and **cosim_logwindow_fwrite**.

Returns NULL (**co_errno**: CO_ELOGWINDOW_NOT_INITIALIZED) if a previous call to **cosim_logwindow_init** failed or **cosim_logwindow_init** was not called in the configuration function.

Also returns NULL (**co_errno**: CO_ENULL_PTR) if a NULL name pointer ("") was given as an argument.

Notes

The Application Monitor must be running and must have been initialized by calling **cosim_logwindow_init** before any log windows can be created.

See Also

cosim_logwindow_write
cosim_logwindow_fwrite
cosim_logwindow_init
Application Monitor

1.15.42 cosim_logwindow_fwrite

```
int cosim_logwindow_fwrite(cosim_logwindow log, const char * format, ...);
```

Header File

```
cosim_log.h
```

Callable Within

Any application code

Description

Writes a formatted message to an Application Monitor log window.

Parameters

cosim_logwindow log	A log window pointer as returned by a preceding call to function **cosim_logwindow_create**.
const char *format	A NULL-terminated character string representing the format string for the message to be written, followed by zero or more arguments.

Return Value

Returns 1 if successful. Returns 0 (**co_errno**: CO_ENULL_PTR) if the **log** argument is NULL.

Notes

Format strings and formatting characters follow the same rules as the C **printf** function.

© 2003-2008 Impulse Accelerated Technologies

See Also

cosim_logwindow_create
cosim_logwindow_write
cosim_logwindow_init

1.15.43 cosim_logwindow_init

```
int cosim_logwindow_init();
```

Header File

```
cosim_log.h
```

Callable Within

Configuration function

Description

Initializes the Application Monitor for interaction with the Impulse C application. The Application Monitor must be running prior to calling this function.

Parameters

None.

Return Value

Returns 1 if successful. Returns 0 if called outside of the configuration function (**co_errno**: CO_ECONFIG_ONLY) or if the Application Monitor is not running.

See Also

cosim_logwindow_create
cosim_logwindow_write
cosim_logwindow_fwrite

1.15.44 cosim_logwindow_write

```
int cosim_logwindow_write(cosim_logwindow log, const char * msg);
```

Header File

```
cosim_log.h
```

Callable Within

Any application code

Description

© 2003-2008 Impulse Accelerated Technologies

Writes a message to an Application Monitor log window.

Parameters

cosim_logwindow log	A log window pointer as returned by a preceding call to function **cosim_logwindow_create**.
const char *msg	A NULL-terminated character string representing the message to be written.

Return Value

Returns 1 if successful. Returns 0 (**co_errno**: CO_ENULL_PTR) if the log argument is NULL.

See Also

cosim_logwindow_create
cosim_logwindow_fwrite
cosim_logwindow_init

© 2003-2008 Impulse Accelerated Technologies

Index

- # -

#pragma 30

- A -

ANSI C 1

- B -

buffered data 2

- C -

C. A. R Hoare 2
CO pipeline Pragma 34
CO set Pragma 34
co_architecture_create 51
co_bit_extract 53
co_bit_extract_u 54
co_bit_insert 55
co_bit_insert_u 56
co_err_eos 10
co_execute 58
co_initialize 58
co_int 5
co_memory 7
co_memory_create 59
co_memory_readblock 61
co_memory_writeblock 62
co_parameter 7
co_process_configure 66
co_process_create 7, 66
co_signal 7
co_signal_create 76
co_signal_post 12, 78
co_signal_wait 12, 79
co_stream 7
co_stream_close 80
co_stream_create 9, 82
co_stream_eos 21, 83
co_stream_open 10, 84
co_stream_read 10, 21, 84
co_stream_write 10, 87
co_uint 5
CoBuilder 19, 34
CoDeveloper 2
communicating sequential processes 1, 2
CoMonitor Application Monitor 46
configuration subroutine 7
Constraints for Hardware 19
cosim_logwindow 46
cosim_logwindow_create 46, 90
cosim_logwindow_fwrite 46, 91
cosim_logwindow_init 46, 92
cosim_logwindow_write 46, 92
Creating Processes 7
Creating Streams 9
CSP 2

- D -

dataflow 1, 2
Datatypes 5
direct connection 14

- E -

EDA 1
efficient applications 2

- F -

FIFOs 1, 2, 6
FLATTEN 34
function calls 19
Function Reference 49

- H -

Hardware 19
Hardware Generation 45
HW_STREAM_CLOSE 33
HW_STREAM_OPEN 33
HW_STREAM_READ 33
HW_STREAM_WRITE 33

© 2003-2008 Impulse Accelerated Technologies

- I -

Impulse C 1
impulse_porky 30
impulse_sm 30
Instruction Scheduling 30
Instrumentation 46

- L -

latency 23
Log Windows 46

- M -

Macro Interfaces 33
Memory 4
Memory Access 30
message passing 1
Monitoring 46

- N -

NONRECURSIVE 34

- O -

Optimizers 30

- P -

pipeline 30, 34
pipelining 23, 30
Platform Support Package 19
Pointer Support 44
Pointers 19, 44
pragma CO pipeline 30
Pragmas 34
PRIMITIVE 34
process run functions 7
Process Synchronization 12
processes 2, 4, 6, 7
Programming for Hardware 19
programming model 1, 2

- R -

rate 23
Reading and Writing Streams 10
records 19
recursion 19
register 14
RTL 1

- S -

SET 34
shared memory 1, 2
Signals 4, 12
Signed 5
simulation library 6
software process 4
stage 23
Stage Delays 30
Stage Master 23
stageDelay 30, 34
Stream Inputs 10
Stream Macro Interfaces 33
Stream Reads 21
Stream-oriented Programming 2
streams 2, 4, 9
structs 19
synchronization 1, 2
synthesize 1

- T -

thread programming 6
Threads 6
Types 5

- U -

UNROLL 34
Unsigned 5
unsynchronized input 14
up/down counter 14

© 2003-2008 Impulse Accelerated Technologies

- V -

VHDL 1

- W -

wired connection 14

© 2003-2008 Impulse Accelerated Technologies