

1 Impulse Tutorial: Using C-Language Simulation for Algorithm Verification



Overview

This Getting Started tutorial describes the process of C-language simulation, by showing how you can compile your C-language hardware module along with test **producer** and **consumer** processes to verify correct behavior. This tutorial builds on what you learned in the first tutorial ([Creating VHDL and Verilog from C-Language](#)). This tutorial concludes with additional discussions of multiple-process parallelism, hardware generation and pipeline optimization.

This tutorial will require approximately 20 minutes to complete, including software run times.

Steps

[Loading the 5 x 5 Image Filter Application](#)
[Understanding the 5 x 5 Image Filter Application](#)
[Compiling and Running the C Code for Simulation](#)
[Notes on Hardware Generation](#)

For additional information about Impulse CoDeveloper, including detailed tutorials describing more advanced design techniques, please visit the Tutorials page at the following location:

www.ImpulseAccelerated.com/Tutorials

1.1 Loading the 5 x 5 Image Filter Application

Image Filter Tutorial, Step 1

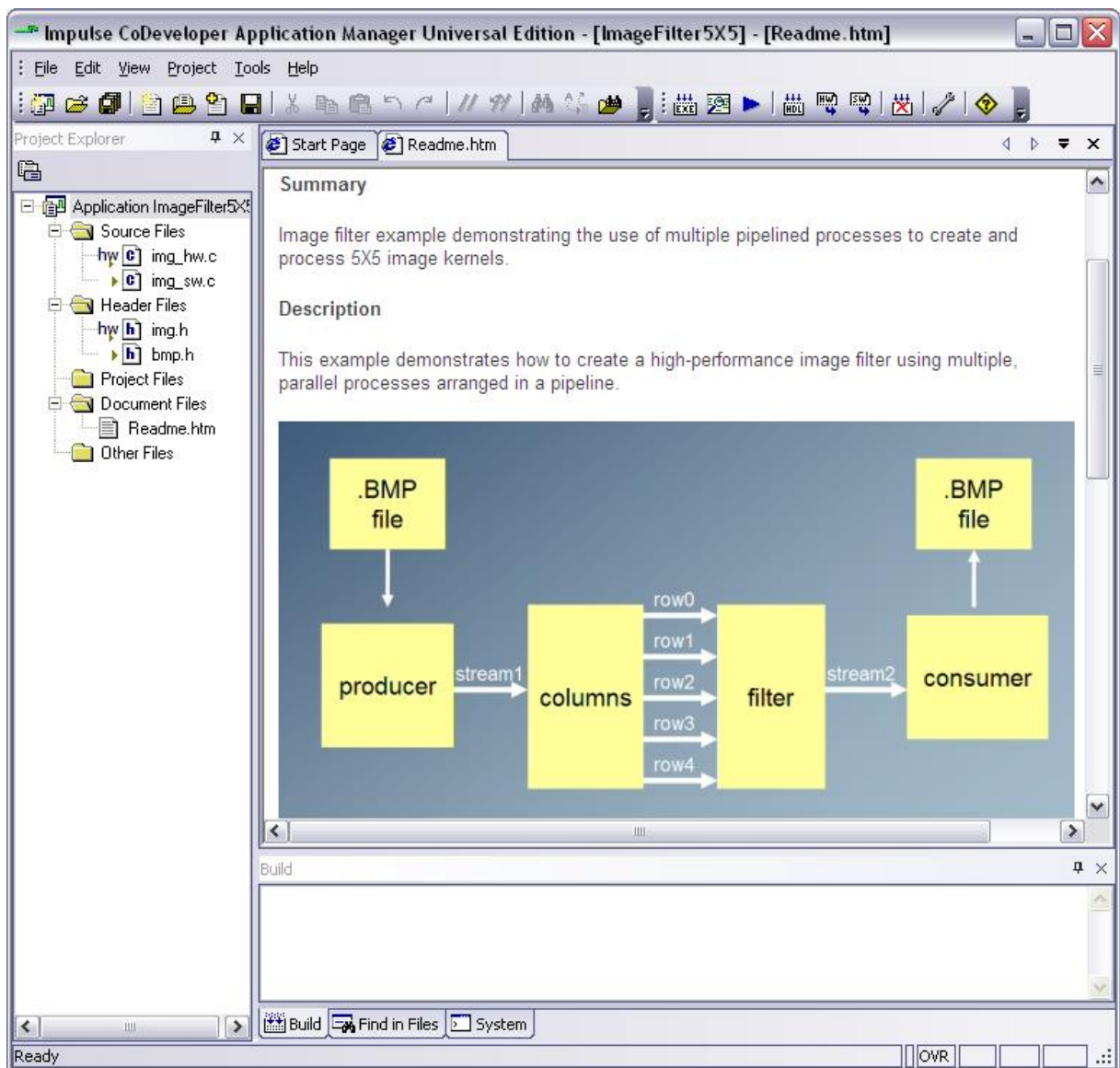
To begin, start the **CoDeveloper** Application Manager:

Start -> Programs -> Impulse Accelerated Technologies -> CoDeveloper -> CoDeveloper Application Manager

Open the **ImageFilterKernel5X5** sample project by selecting **Open Project** from the **File** menu, or by clicking the **Open Project** toolbar button. Navigate to the **.\Examples\Image\ImageFilterKernel5X5** directory within your CoDeveloper installation. (You may wish to copy this example to an alternate directory before beginning.)

The project file is also available from the CoDeveloper Start Page, in the **Help and Support** tab.

After loading the project, you will see a **Readme** file with a block diagram, and a **Project Explorer** window as shown below:



You can scroll down in the **Readme** file as shown to learn more about this application. To summarize, this is a 5-pixel by 5-pixel, 2-dimensional image convolution filter that operates on 16-bit grayscale data. This filter could represent one section of a larger video filtering application, for example one of the first steps in a more complex object recognition algorithm.

The method used for creating this filter involves the creation of two parallel C-language processes named **columns** and **filter**, respectively.

The **columns** process accepts incoming pixels, for example from a video stream, and stores those pixels in an internal buffer large enough to store a little more than four scan lines. When its internal buffers are filled, the process begins to emit five parallel streams of pixels representing five adjacent scan line rows. This is what is referred to as a *marching columns* method of buffering.

The **filter** process executes in parallel with the **columns** process, accepting the five incoming streams and performing a 5-pixel by 5-pixel convolution to generate a stream of filtered outputs.

The **producer** and **consumer** processes are used during software testing to read and write sample

image files as described later in this tutorial.

Source files included in this project include:

- **img_hw.c** - This source file includes the C-language description of the image filter, including its I/O. This description includes the two hardware processes, **columns** and **filter**, as well as a configuration subroutine.
- **img_sw.c** - This source file includes a set of software testing routines including a **main()** function, and **consumer** and **producer** software processes as illustrated in the block diagram.
- **img.h** - This source file includes common declarations used in both the filter description (in **img_hw.c**), and in the test routines (**img_sw.c**).
- **bmp.h** - This source file includes declarations used only in the test routines. These declarations are related to the processing of BMP format files.

You can open any of these three files by simply double-clicking on the file name in the Project Explorer window. In the next step, we will describe in detail how this example works.

Next Step

[Understanding the 5 x 5 Image Filter Application](#)

1.2 Understanding the 5 x 5 Image Filter Application

Image Filter Tutorial, Step 2

Before compiling the image filter application to perform a simulation, let's first take a moment to understand its basic operation.

The Image Filter C-Language Processes

The specific hardware processes that we will be testing are represented by the following two functions, which are located in **img_hw.c**:

```
void columns (co_stream input_stream, co_stream r0, co_stream r1, co_stream r2,
co_stream r3, co_stream r4)

void filter (co_stream r0, co_stream r1, co_stream r2, co_stream r3, co_stream r4,
co_stream output_stream)
```

These two C-language subroutines each represent an *Impulse C process*. As described in the first tutorial, a *process* in Impulse C is a module of code, expressed as a **void** subroutine, that describes a hardware or software component.

If you are an experienced hardware designer, you can simply think of a process as being analogous to a VHDL *entity*, or to a Verilog *module*.

If you are a software programmer, you can think of a process as being a subroutine that will loop forever, in a separate *thread of execution* from other processes.

When implemented as hardware in the FPGA, these two processes will run concurrently, processing data on their respective inputs and outputs. Because the two processes will be arranged such that process **filter** accepts inputs generated as outputs by process **columns**, we can think of these two processes as a *system-level pipeline*.

System-level pipelining is an important concept to understand. When combined with *statement-level parallelism* and *loop-level pipelining*, system-level pipelining can create remarkable levels of software acceleration when compared to traditional, instruction-based processors. In fact, for video processing the combination of loop-level and system-level pipelining allows video signals to be processed and filtered in real-time, with no degradation of the signal or reduction in data throughput.

The Columns Process

Scroll down to find the definition of the **columns** process. The process has no return value, and has a total of six streaming interfaces that have been defined using Impulse C **co_stream** data types. These streams are used to:

- Read in raw, unfiltered pixels, for example from a video source. If this process is to operate on a live video stream, then the process will need to accept a new pixel value every clock cycle.
- Write out five pixel values on the **r0**, **r1**, **r2**, **r3** and **r4** streams. These pixel values will then be read into the **filter** process that follows.

Scroll down in the source code to view the inner loops for process **columns**:

```
do {
    for ( i = 2; i < HEIGHT; i++ ) {
        // Note: the following loop will pipeline with a rate of
        // one cycle if the target platform supports dual-port RAM.
        for (j=0; j < WIDTH; j++) {
            #pragma CO PIPELINE
            p04 = B[j];
            p14 = C[j];
            p24 = D[j];
            p34 = E[j];
            co_stream_read(input_stream, &p44, sizeof(co_uint16));
            co_stream_write(r0, &p04, sizeof(co_uint16));
            co_stream_write(r1, &p14, sizeof(co_uint16));
            co_stream_write(r2, &p24, sizeof(co_uint16));
            co_stream_write(r3, &p34, sizeof(co_uint16));
            co_stream_write(r4, &p44, sizeof(co_uint16));
            B[j] = p14;
            C[j] = p24;
            D[j] = p34;
            E[j] = p44;
        }
    }
    IF_SIM(break;) // For simulation we break after one frame
} while (1);
```

When compiled as hardware, the outer **do-while** loop runs forever, accepting single-pixel input values using **co_stream_read**, and writing out five parallel pixel values using **co_stream_write**. When examining this code, note that:

- A **PIPELINE** pragma has been placed at the top of the loop, indicating to the compiler that this is a critical loop that requires high throughput. As a result of this pragma, the compiler will generate hardware with pipeline control logic and parallel pipeline stages. As the code comment indicates, the pipeline rate that will be achieved by the compiler will depend in part on the type of FPGA memory available for the **B**, **C**, **D** and **E** arrays. This pragma only has meaning during hardware generation; it is ignored during software simulation.
- An **IF_SIM** macro has been used along with a **break** statement to exit the outer **do-while** loop during simulation. This is a useful technique to allow the simulation to end cleanly, with output files properly closed. (If the loop was not exited in this way, the application would not stop running during simulation and would need to be forcibly halted.)

The Filter Process

Scroll down more to find the definition of the **filter** process. The process also has a total of six streaming interfaces. These streams are used to:

- Read in the five streams of pixels that were generated by the columns process, representing five adjacent scan lines.
- Write out a single filtered pixel value on the **output_stream** streams.

Scroll down in the source code to view the inner loop for process **filter**:

```
do {
#pragma CO PIPELINE
#pragma CO set stageDelay 100
    err = co_stream_read(r0, &data0, sizeof(co_uint16));
    err &= co_stream_read(r1, &data1, sizeof(co_uint16));
    err &= co_stream_read(r2, &data2, sizeof(co_uint16));
    err &= co_stream_read(r3, &data3, sizeof(co_uint16));
    err &= co_stream_read(r4, &data4, sizeof(co_uint16));
    if (err != co_err_none) break;

    p00 = p01; p01 = p02; p02 = p03; p03 = p04;
    p10 = p11; p11 = p12; p12 = p13; p13 = p14;
    p20 = p21; p21 = p22; p22 = p23; p23 = p24;
    p30 = p31; p31 = p32; p32 = p33; p33 = p34;
    p40 = p41; p41 = p42; p42 = p43; p43 = p44;

    p04 = data0;
    p14 = data1;
    p24 = data2;
    p34 = data3;
    p44 = data4;

    sop = p00*F00 + p01*F01 + p02*F02 + p03*F03 + p04*F04
          + p10*F10 + p11*F11 + p12*F12 + p13*F13 + p14*F14
          + p20*F20 + p21*F21 + p22*F22 + p23*F23 + p24*F24
          + p30*F30 + p31*F31 + p32*F32 + p33*F33 + p34*F34
          + p40*F40 + p41*F41 + p42*F42 + p43*F43 + p44*F44;
    if (sop > 255*FDIV)
        result = 255;
    else
        result = (co_uint16) (sop >> 7); // Divide by 128
    co_stream_write(output_stream, &result, sizeof(co_uint16));
} while (1);
```

As in the columns process, the outer **do-while** loop runs forever, accepting input values using **co_stream_read**, and writing out five parallel pixel values using **co_stream_write**. When examining this code, note that:

- A **PIPELINE** pragma has been placed at the top of the loop, indicating to the compiler that this is a critical loop that requires high throughput. An additional pragma, **SET StageDelay**, provides additional information about the maximum pipeline stage delay for this loop, for the purpose of making size/speed tradeoffs. (The **SET StateDelay** pragma is described in more detail in the Impulse C User's Guide.) These pragmas only have meaning during hardware generation; they are ignored during software simulation.
- A large, sum-of-products statement is used to perform the convolution on a sliding sub-window of pixels being read from the **r0** through **r4** inputs. This statement represents the most computationally intensive portion of this algorithm, and the part of the code therefore needing the greatest level of parallel optimization.

- The **IF_SIM** macro is not being used in this loop, but a **break** statement is being used to exit the outer **do-while** loop if there is any error found when reading data on the inputs streams. The effect of this is that if the producing process (in this case **columns**) closes its output stream, then this loop will exit, causing the simulation to stop as needed after a single test frame. (Checking for stream read errors in this manner may or may not have any benefit in the generated hardware, depending on the nature of the application and the capabilities of the target platform.)

The Configuration Subroutine

The **columns** and **filter** subroutines together represent the algorithm to be implemented as hardware in the FPGA. To complete the application, however, we need to include one additional routine that describes the I/O connections and other compile-time characteristics for this application. This *configuration routine* serves three important purposes, allowing us to:

1. define I/O characteristics such as FIFO depths and the sizes of shared memories.
2. instantiate and interconnect one or more copies of our Impulse C processes.
3. optionally assign physical, chip-level names and/or locations to specific I/O ports.

This example includes two hardware processes (**columns** and **filter**) and also includes the two testing routines, **producer** and **consumer**. Our configuration routine therefore includes statements that describe how the **producer**, **columns**, **filter** and **consumer** processes are connected together. The complete configuration routine is shown below:

```
void config_img(void *arg)
{
    int error;
    co_stream stream1, r0, r1, r2, r3, r4, stream2;
    co_process columns_process, filter_process;
    co_process producer_process, consumer_process;
    co_signal header_ready;

    stream1 = co_stream_create("stream1", UINT_TYPE(16), 2);
    r0 = co_stream_create("r0", UINT_TYPE(16), 5);
    r1 = co_stream_create("r1", UINT_TYPE(16), 5);
    r2 = co_stream_create("r2", UINT_TYPE(16), 5);
    r3 = co_stream_create("r3", UINT_TYPE(16), 5);
    r4 = co_stream_create("r4", UINT_TYPE(16), 5);
    stream2 = co_stream_create("stream2", UINT_TYPE(16), 2);
    header_ready = co_signal_create("header_ready");

    columns_process = co_process_create("columns",
                                       (co_function)columns,
                                       6, stream1, r0, r1, r2, r3, r4);
    filter_process = co_process_create("filter",
                                       (co_function)filter,
                                       6, r0, r1, r2, r3, r4, stream2);
    producer_process = co_process_create("producer",
                                       (co_function)producer,
                                       2, stream1, header_ready);
    consumer_process = co_process_create("consumer",
                                       (co_function)consumer,
                                       2, stream2, header_ready);

    co_process_config(columns_process, co_loc, "PE0");
    co_process_config(filter_process, co_loc, "PE0");

    IF_SIM(error = cosim_logwindow_init();)
}
```

To summarize, the **columns** and **filter** subroutines describe the algorithm to be generated as FPGA

hardware, while the **producer** and **consumer** subroutines (described elsewhere, in **img_sw.c**) are used for testing purposes. The configuration routine is used to describe how these three processes communicate, and to describe other characteristics of the process I/O. In this configuration subroutine, note the following:

- There are a total of seven streams being declared and created. They include the system-level inputs and outputs (here labeled **stream1** and **stream2**), and the five intermediate streams (**r0**, **r1**, **r2**, **r3** and **r4**). Notice that each stream is created with a width (in this case 16 bits) and a depth. The stream depth is an important decision when creating pipelined systems. (Pipeline and stream optimization techniques, and other methods of improving FPGA resource efficiency, are available as application notes from Impulse.)
- Four processes are declared and created, represent unique *instances* of the **producer**, **consumer**, **columns** and **filter** subroutines. Two of these processes are hardware, as indicated by the **co_process_config** function calls, while the other two are software processes used only for testing. During simulation, each of these four processes will be run in a separate thread, allowing us to model their transaction-level parallel behaviors.
- A call to **cosim_logwindow_init** appears at the end of the configuration subroutine. This function call is used to enable the Application Monitor that we will be using in the next tutorial step.

The Producer and Consumer Processes

The **producer** and **consumer** processes, along with a **main** function, together represent a *software test bench*. The term *test bench* comes from the world of hardware design, and it refers to a device or fixture that allows a hardware module - or in this case a software description of hardware - to be tested by applying sample inputs to it, and by observing the resulting outputs.

The declarations for **producer** and **consumer** appear in **img_sw.c** as follows:

```
void producer(co_stream pixels_raw, co_signal header_ready)

void consumer(co_stream pixels_filtered, co_signal header_ready)
```

To test this image filter application, we have decided to use a BMP format file as the input, representing one frame of a video signal. The producer process (found in the file **img_sw.c**) describes how the test input file is read and how its pixel values are streamed into the **columns** process. The section of C code that actually performs this streaming of data into the **columns** process looks like this:

```
// Now send in all the pixels for the image
printf("Sending BMP pixels...\n");
for (i=0; i < ByteCount; i++) {
    pixelValue = (pBitMap[i++]);
    co_stream_write(pixels_raw, &pixelValue, sizeof(pixelValue));
}
```

As you can see, the **co_stream_write** function is being used in the **producer** process to move data pixel-by-pixel into the filter.

On the other end of the filter, the **consumer** process reads the data coming out of the **filter** process, and writes it out to a new BMP format file representing the filtered output:

```
printf("Consumer reading data...\n");
while ( co_stream_read(pixels_filtered, &pixelValue, sizeof(co_uint16)) ==
co_err_none ) {
    putc(pixelValue, outfile);
    pixelCount++;
}
```


When examining the **producer** and **consumer** processes, note that:

- A signal called **header_ready** is used to coordinate the reading and writing of the BMP files. This is done because the **producer** process must read the BMP file header first, before the **consumer** process can write out an equivalent header. Without this signaling mechanism, the **producer** and **consumer** processes would run concurrently (in separate threads) possibly resulting in incomplete or wrong data being written to the output BMP file header.
- **cosim_logwindow_create** and **cosim_logwindow_fwrite** functions are used to instrument the code for debugging purposes, using the Application Monitor.
- A **main** function (located at the end of **img_sw.c**) is used to start the test running. This main function makes use of **co_initialize** and **co_execute** functions to initialize and bring up the four processes (**producer**, **consumer**, **columns** and **filter**) as separate threads for simulation.

```
int main(int argc, char *argv[])
{
    co_architecture my_arch;
    void *param = NULL;
    int c;

    printf("Impulse C is Copyright 2003-2009 Impulse Accelerated Technologies,
Inc.\n");

    my_arch = co_initialize(param);
    co_execute(my_arch);
    printf("\n\nApplication complete. Press the Enter key to continue.\n");
    c = getc(stdin);

    return(0);
}
```

Next Step

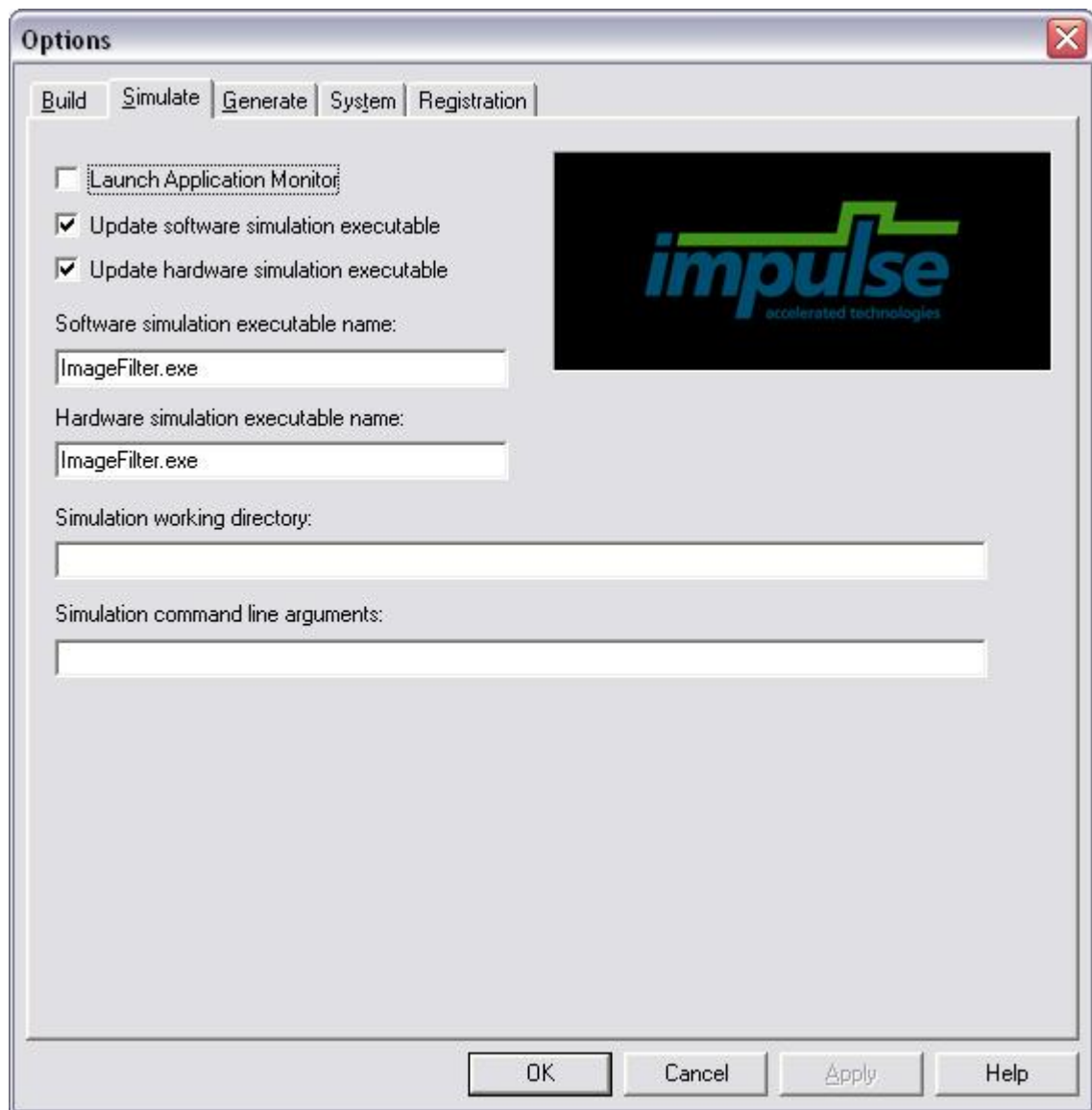
[Compiling and Running the C Code for Simulation](#)

1.3 Compiling and Running the C Code for Simulation

Image Filter Tutorial, Step 3

Now that you have examined the application source code, the next step is to compile the complete application using a standard C compiler, for the purpose of simulation.

Before compiling, let's take a moment to examine some of the options available for simulation. Open the Simulation Options dialog by selecting **Project -> Options**, then clicking on the **Simulate** tab as shown below:



Notice in the Simulation Options dialog that you can specify the target executable name (in this case **ImageFilter.exe**), as well as its location. In the case the location is left blank, so the executable will be generated in the project directory. Also notice that there is a field allowing you to enter any command line arguments that might be needed in your software test. In this example there are no arguments passed, but you may use this field to pass in file names or other information for your test, using the **argc**, **argv** method of handling arguments in your **main** function.

Click OK to exit the dialog.

To compile the application, select **Project -> Build Software Simulation Executable** resulting the messages shown below:

```
===== Building target 'build_exe' in file _Makefile =====
"C:/Impulse/CoDeveloper3/MinGW/bin/gcc" -g "-IC:/Impulse/CoDeveloper3/Include" "-IC:/Impulse/CoDeveloper3/StageMaster/include" -DWIN32 "-IC:/Impulse/CoDeveloper3/MinGW/include" -o img_hw.o -c img_hw.c
"C:/Impulse/CoDeveloper3/MinGW/bin/gcc" -g "-IC:/Impulse/CoDeveloper3/Include" "-IC:/Impulse/CoDeveloper3/StageMaster/include" -DWIN32 "-
```

```
IC:/Impulse/CoDeveloper3/MinGW/include" -o img_sw.o -c img_sw.c
"C:/Impulse/CoDeveloper3/MinGW/bin/gcc" -g img_hw.o img_sw.o
"C:\Impulse\CoDeveloper3\Libraries\ImpulseC.lib" -o ImageFilter.exe
```

```
===== Build of target 'build_exe' complete =====
```

As you can see, CoDeveloper has invoked the **gcc** compiler three times to build and link this application, resulting in an output file called **ImageFilter.exe**. This executable is what is called a *software simulation executable*. You can execute this file either using a command line, or by executing from within CoDeveloper.

Running the Application Monitor

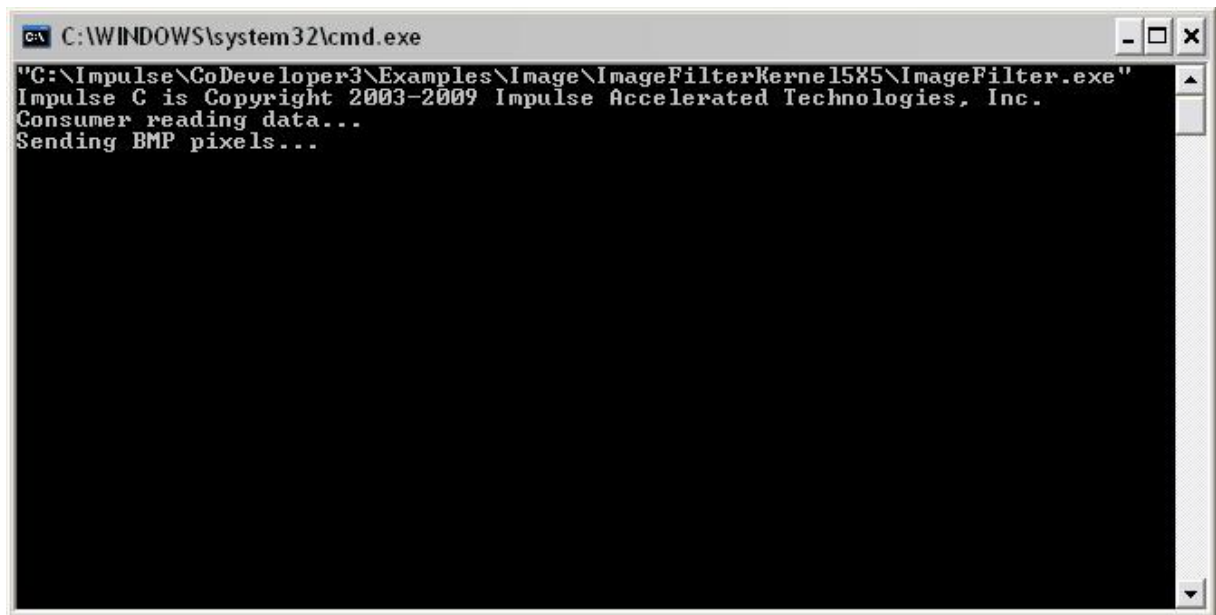
When running the executable, you have the choice of also running the CoDeveloper Application Monitor, which allows you to observe the movement of data through the data streams and verify correct process-to-process connectivity.

Start the Application Monitor by selecting **Tools -> Application Monitor**. A new window will appear as shown below:



To start your simulation executable running, leave the Application Monitor open on your desktop, return to the CoDeveloper application and select **Project -> Launch Software Simulation Executable**.

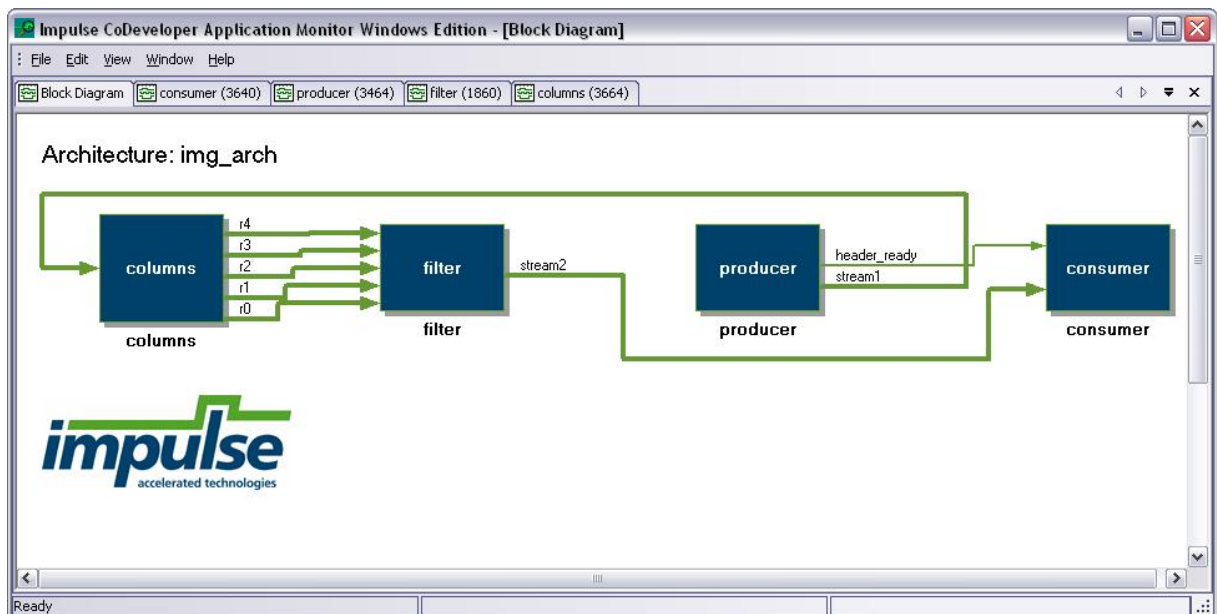
You will see a console window appear as the application begins running:



```
C:\WINDOWS\system32\cmd.exe
"C:\Impulse\CoDeveloper3\Examples\Image\ImageFilterKernel15X5\ImageFilter.exe"
Impulse C is Copyright 2003-2009 Impulse Accelerated Technologies, Inc.
Consumer reading data...
Sending BMP pixels...
```

This console window displays any messages that result from **printf** statements in the C application, as well as accepting any required keyboard input.

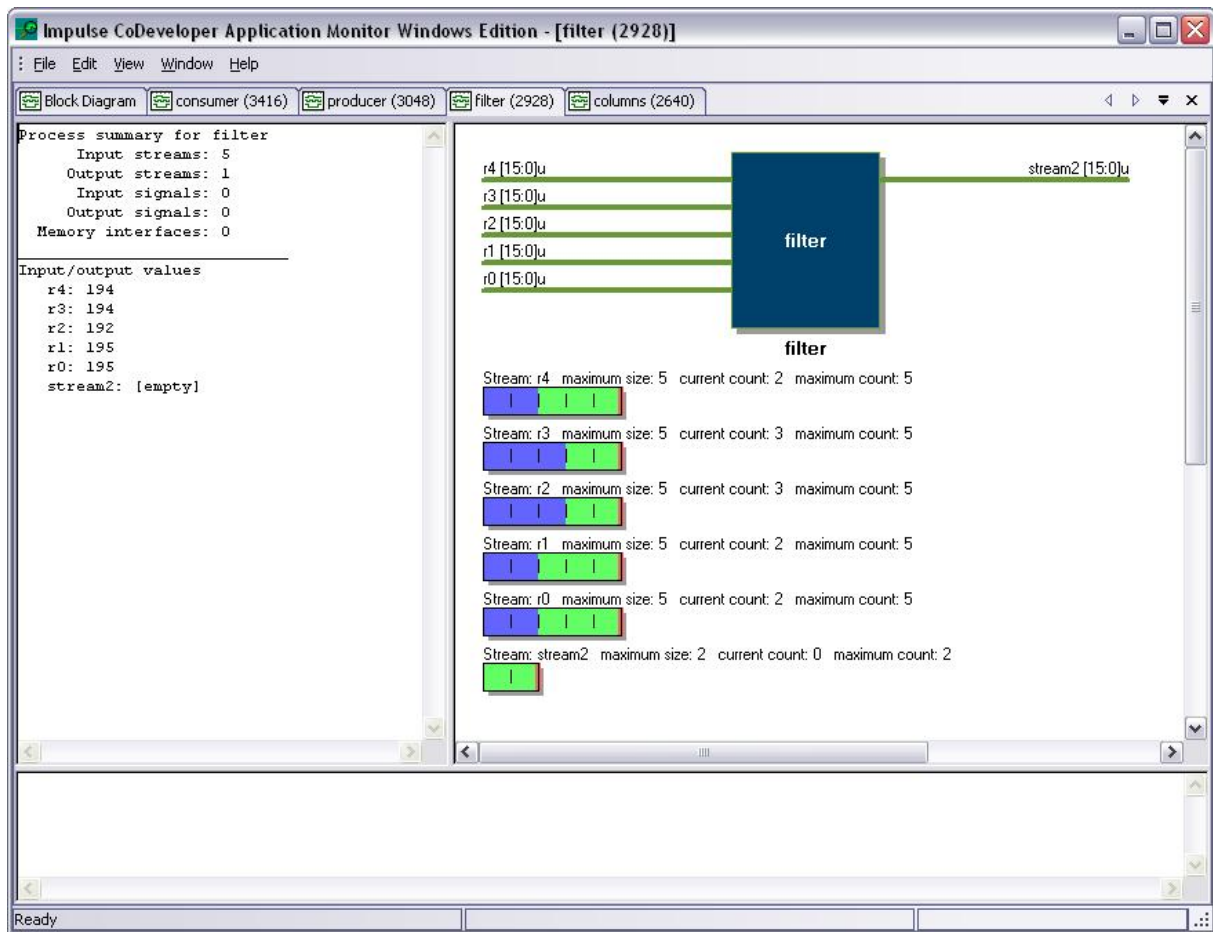
Now make the Application Monitor Window window active, and select the **Block Diagram** tab. You should see an automatically generated block diagram similar to the following:



(The order of the blocks displayed in the diagram may be different, depending on the order in which the threads initialize on your system.)

This block diagram provides you with a quick way to verify correct stream connections.

Now, click on the **filter** block to push into the **filter** process. You will see a display similar to the following:



This view allows you to monitor the transaction-by-transaction movement of data between the processes in your application. If you have used `cosim_logwindow_fwrit` functions to instrument your processes, you would also see messages appearing in the transcript section of this window. These `cosim_logwindow` related functions can be particularly useful to augment source level debugging for larger, multiple process applications.

Next Step

[Notes on Hardware Generation](#)

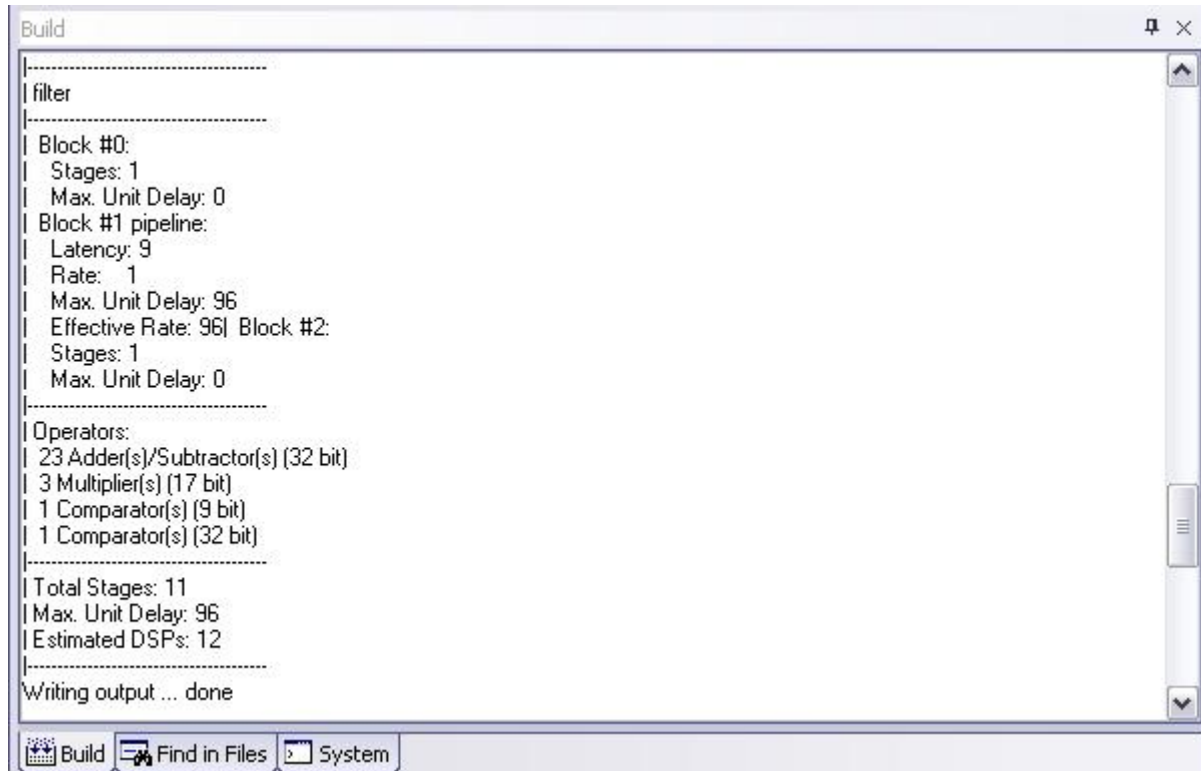
1.4 Notes on Hardware Generation

Image Filter Tutorial, Step 4

At this point you have successfully simulated a multiple-process C-language application. If you also went through the the tutorial titled [Creating VHDL and Verilog from C-Language](#), then you also have a basic knowledge of the software-to-hardware generation process.

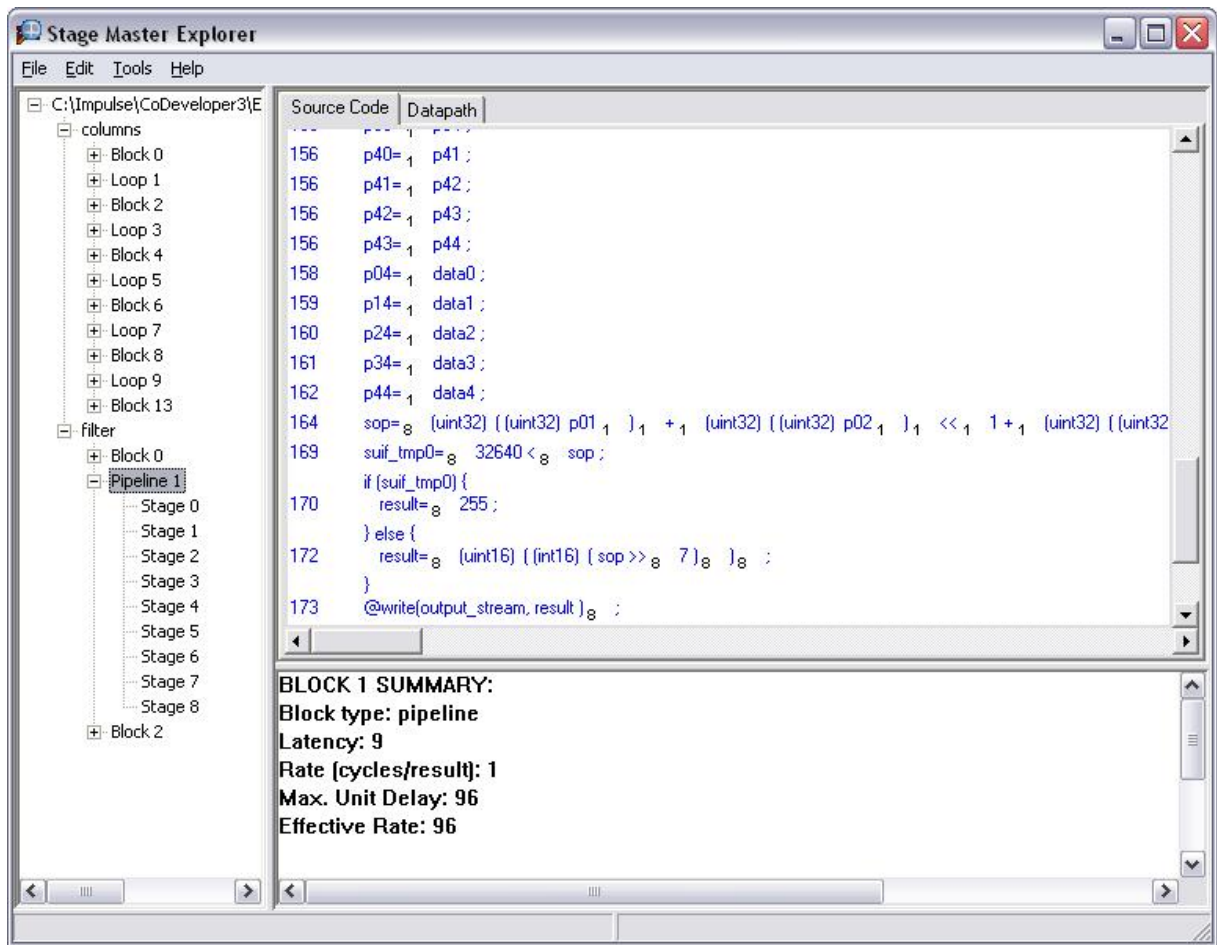
Let's take a moment to generate hardware for this example, and examine the generated HDL. We will view the generated hardware in the form of VHDL; if you generated Verilog, the syntax will be different but the generated hardware will be similar.

Select **Project -> Generate HDL** to start the hardware generation. When processing is complete, you should see messages that include the following:



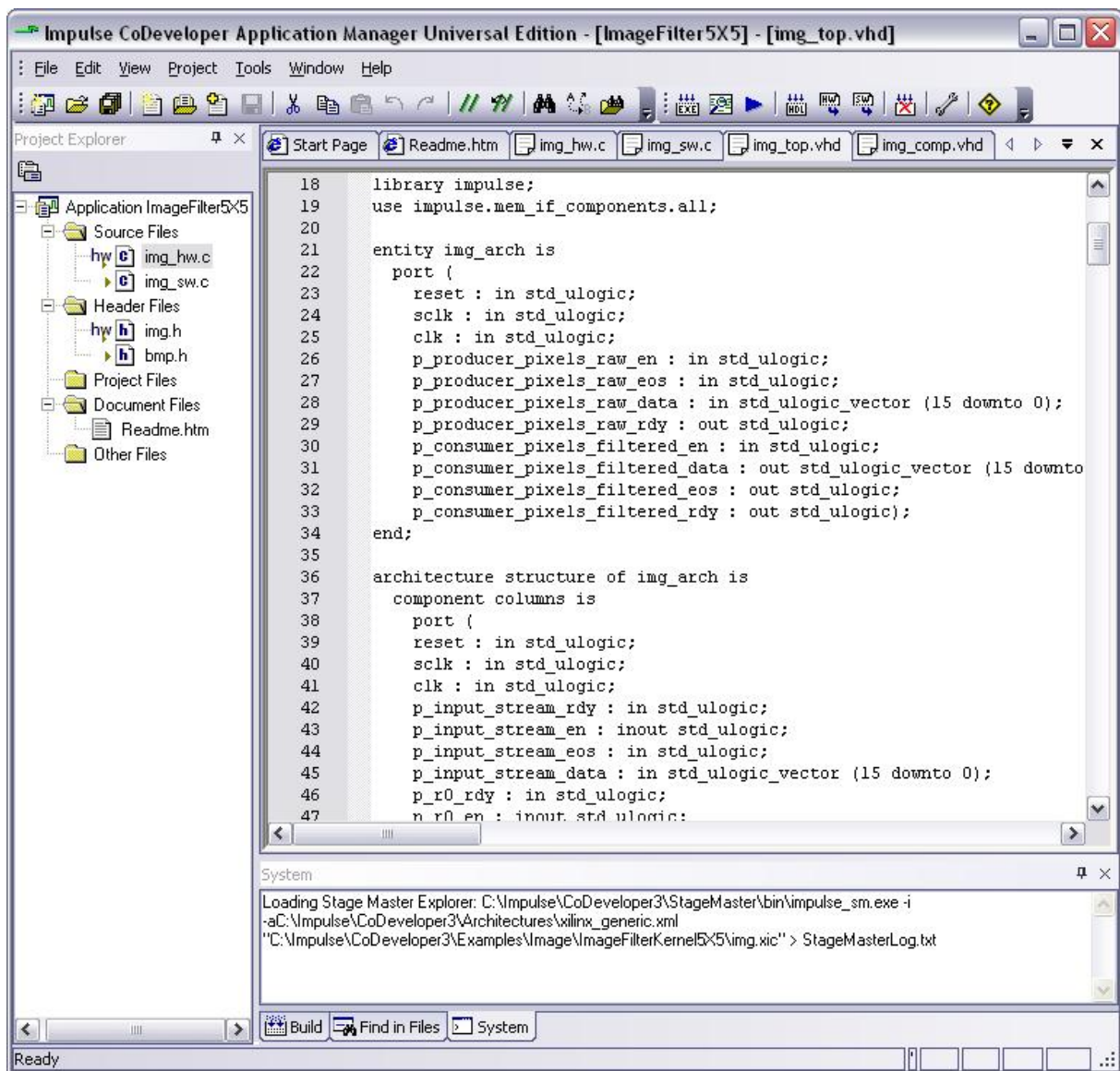
As in the simpler FIR filter example, these messages provide you with a summary of estimated pipeline rates and operator usage for each Impulse C process.

Now, invoke the State Master Explorer tool to investigate the generated hardware. Select **Tools -> Stage Master Explorer**, then select the img.xic file to start the Explorer tool.



Use the navigation tree control and the **Source Code** and **Datapath** windows to explore the generated hardware structures and see how the C code for **columns** and **filter** have been parallelized.

If you wish, you can also open the generated VHDL source code and explore the generated hardware code. The generated hardware is located in the **hw** subdirectory of the project, in the files **img_top.vhd** and **img_comp.vhd**.



Top-Level HDL Entity (Module)

Recall that in our original C code, the I/O interfaces were described using `co_stream` data, and using stream-related functions such as `co_stream_read` and `co_stream_write`.

In the generated hardware, the HDL file with the `_top` file name suffix (in this case `img_top.vhd`) represents the top-level I/O implementing these streaming interfaces, as shown below:

```

entity img_arch is
  port (
    reset : in std_ulogic;
    sclk : in std_ulogic;
    clk : in std_ulogic;
    p_producer_pixels_raw_en : in std_ulogic;
    p_producer_pixels_raw_eos : in std_ulogic;
    p_producer_pixels_raw_data : in std_ulogic_vector (15 downto 0);
    p_producer_pixels_raw_rdy : out std_ulogic;
    p_consumer_pixels_filtered_en : in std_ulogic;

```



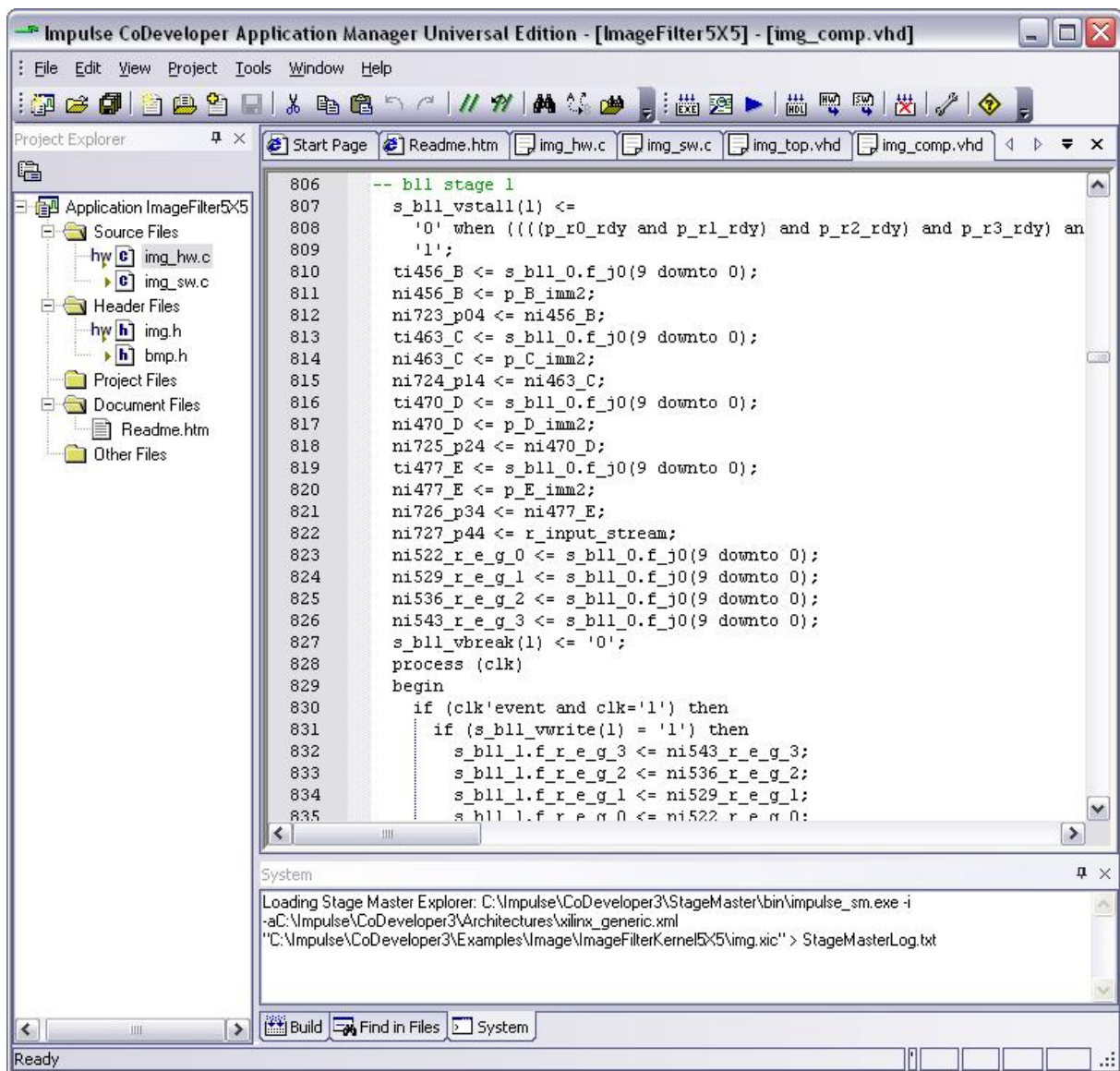
```
p_consumer_pixels_filtered_data : out std_ulogic_vector (15 downto 0);  
p_consumer_pixels_filtered_eos : out std_ulogic;  
p_consumer_pixels_filtered_rdy : out std_ulogic);  
end;
```

For each of the two streams, notice that there are data and flow control signals with the suffix **_data**, **_en**, **_rdy** and **_eos**. These flow control hardware signals are documented in the Impulse User's Guide and can be used to connect other streaming hardware (as as analog-to-digital inputs, video inputs and other streaming hardware) directly to an Impulse-generated streaming hardware process.

Also notice the names used when generating the I/O signals. Because we did not specify actual port names for our input and output streams, the compiler has assigned names to the hardware streams based on their source and destination, in this case the **producer** and **consumer** processes. In a real-world application we might choose to assign specific names to these streams, using a **co_port_create** function, or choose a platform support package that automatically generates appropriately named I/O wrappers for our target platform.

Component-Level HDL Entities for Columns and Filter

To view the lower-level HDL code for the columns and filter subroutines, open the **img_comp.vhd**. This HDL file includes the state machines and other logic that implements the parellized and pipelined operations described in C.



Next Steps

You have now completed this tutorial. At this point you may want to explore other examples provided with CoDeveloper, or explore some of the more advanced, platform-specific tutorials to learn more about how to use the generated HDL in actual hardware.

For additional information other detailed tutorials, please visit the Tutorials page at the following location:

www.ImpulseAccelerated.com/Tutorials