1 Impulse Tutorial: Generating a Xilinx FPGA Netlist from C-Language

Overview

This Getting Started tutorial demonstrates how to compile C code into HDL, and then synthesize the HDL using Xilinx ISE. This tutorial assumes that you already know the basics of C-to-HDL compilation. If you have not already done so, you are encouraged to read through one of the basic tutorials provided on the following site:

www.ImpulseAccelerated.com/Tutorials

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

Steps

Loading the Image Filter Application
Understanding the Image Filter Application
Compiling the C Code to Create HDL
Creating and Using an ISE Project

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 Image Filter Application

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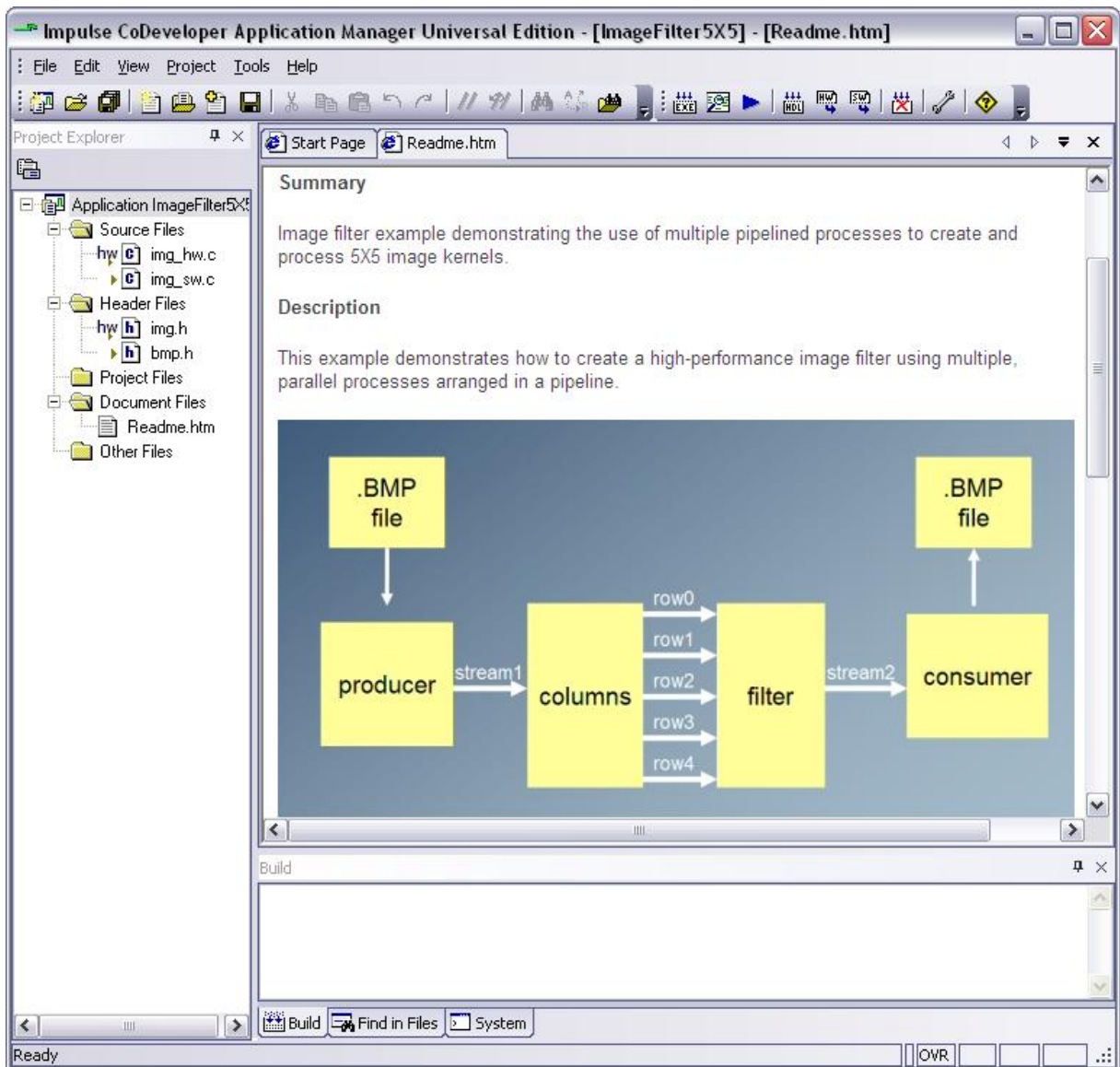
You can scroll down in the **Readme** file as shown to learn more about this application. To summarize, this is a 5-pixel by 5-pixel, 2-dimensional image convolution filter that operates on 16-bit grayscale data. This filter could represent one section of a larger video filtering application, for example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

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

1.2   Understanding the Image Filter Application

Xilinx ISE 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 seperate *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 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 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.

Next Step

Compiling the C Code to Create HDL

1.3   Compiling the C Code to Create HDL

Xilinx ISE Tutorial, Step 3

Now that you have examined the sample C code, the next step is to create FPGA hardware and related files from the C code found in the **img_hw.c** source file. This requires that you select a platform target, specify any needed options, and initiate the hardware compilation process.

Specifying the Xilinx Generic VHDL Platform Support Package

To specify a platform target, select **Project** -> **Options**, the select the **Generate** tab to open the **Generate Options** dialog as shown below:
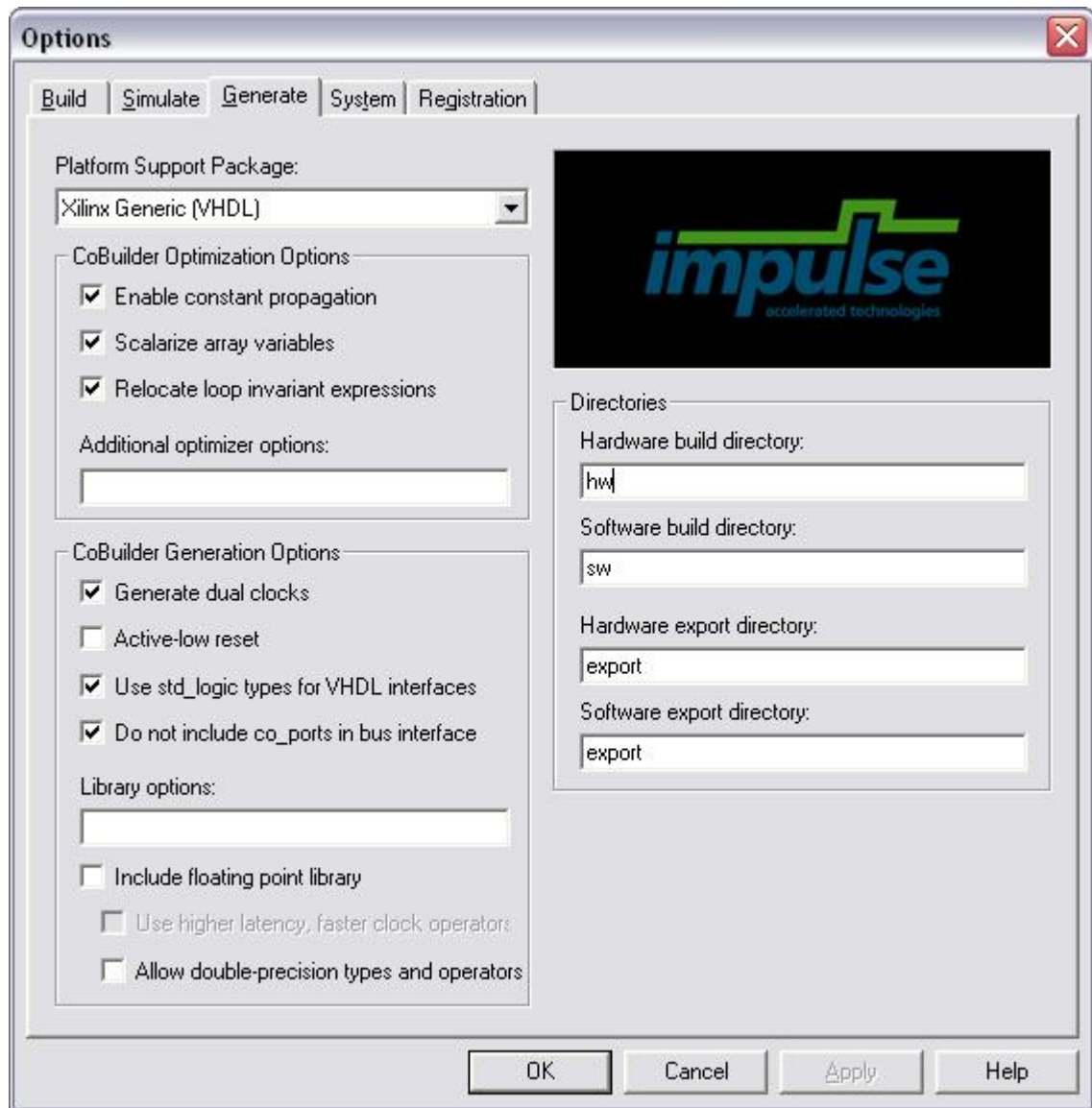
This dialog allows you to set various options for hardware generation, and to select a target platform. Notice that the Platform Support Package setting indicates "Xilinx Generic (VHDL)" for the output. You can click on the drop-down Platform Support Package selection list to see what kind of *platform support packages* are installed on your system. Please select "Xilinx Generic (VHDL)" as shown..

Note: if you would prefer to generate Verilog, you can change the setting to "Xilinx Generic (Verilog)" before continuing.

Other options on this dialog allow you to set the target directory for generating and exporting your HDL, and set options related to the clock and reset hardware, and include optional hardware libraries.

Click **OK** to save the options and exit the dialog.

Generating HDL for the Hardware Process

To generate hardware in the form of HDL files, select **Project** -> **Generate HDL**. A series of processing steps will run in the **Build** console window as shown below (you can use your mouse to

expand the **Build** window as shown):

```
Build                                                                    ⊓  ×
|----------------------------------------                                    ^
| filter
|----------------------------------------
| Block #0:
|    Stages: 1
|    Max. Unit Delay: 0
| Block #1 pipeline:
|    Latency: 9
|    Rate:   1
|    Max. Unit Delay: 96
|    Effective Rate: 96|  Block #2:
|    Stages: 1
|    Max. Unit Delay: 0
|----------------------------------------
| Operators:
| 23 Adder(s)/Subtractor(s) (32 bit)
| 3 Multiplier(s) (17 bit)
| 1 Comparator(s) (9 bit)
| 1 Comparator(s) (32 bit)
|----------------------------------------
| Total Stages: 11
| Max. Unit Delay: 96
| Estimated DSPs: 12
|----------------------------------------
Writing output ... done                                                      v
[⊞ Build][🔍 Find in Files][▣ System]
```
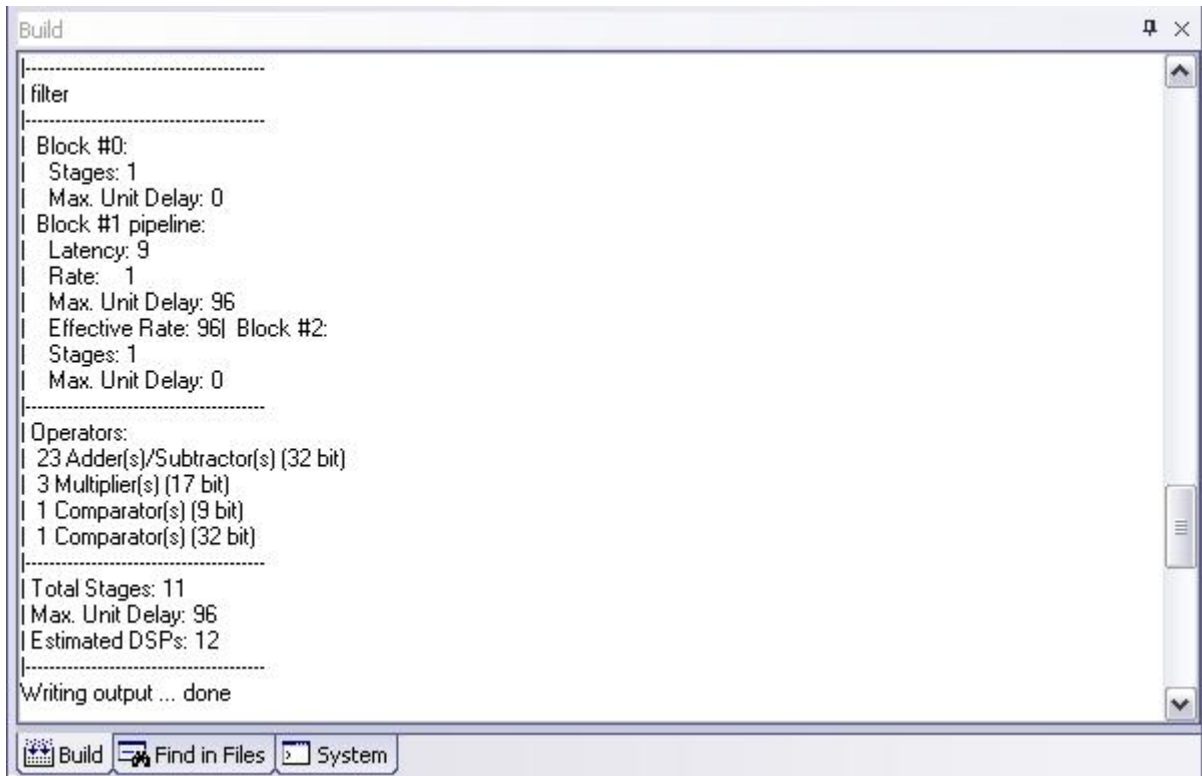
The messages generated by the hardware compiler include estimates of loop latencies and pipeline rates as well as estimates of the number of required hardware operations, as shown above. These messages can help you to quickly evaluate the effectiveness of your C-language coding methods, allowing you to iteratively refactor and improve your algorithms *before* going through a potentially long process of FPGA synthesis.

When the optimization and C-to-HDL processing has completed you will have two resulting HDL files (either VHDL or Verilog) created in the **hw** subdirectory of your project directory, including a **lib** subdirectory, as shown below:
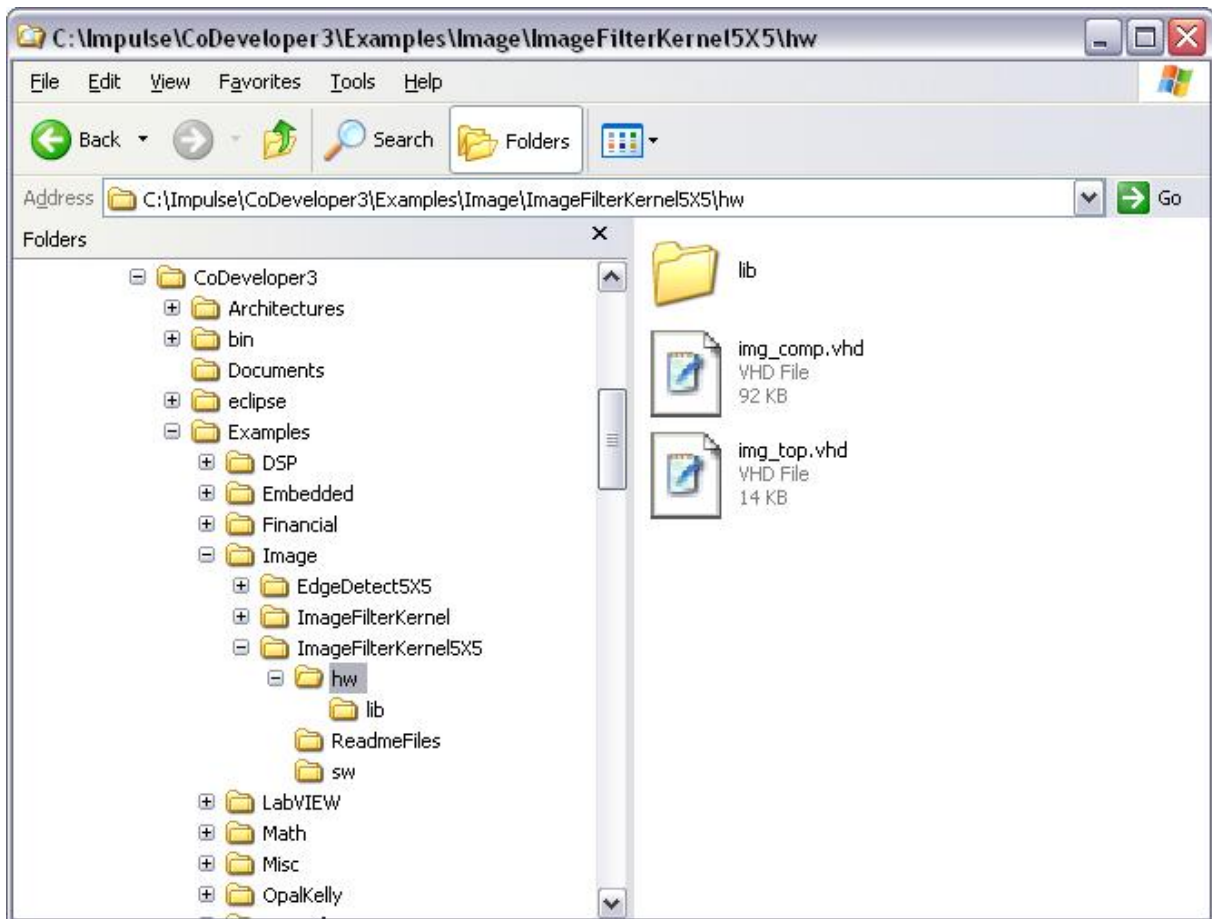
Next Step

1.4   Creating and Using an ISE Project

Xilinx ISE Tutorial, Step 4

You have successfully generated HDL from a C-language description. The next step will be to create a Xilinx ISE project file, and import the generated HDL files into the new project. We will also select a specific FPGA target device at this point, and start the synthesis process..

Creating the Xilinx ISE Project

Run the Xilinx ISE software from the Windows Start menu:

Start -> Programs -> Xilinx ISE Design Suite -> ISE -> Project Navigator
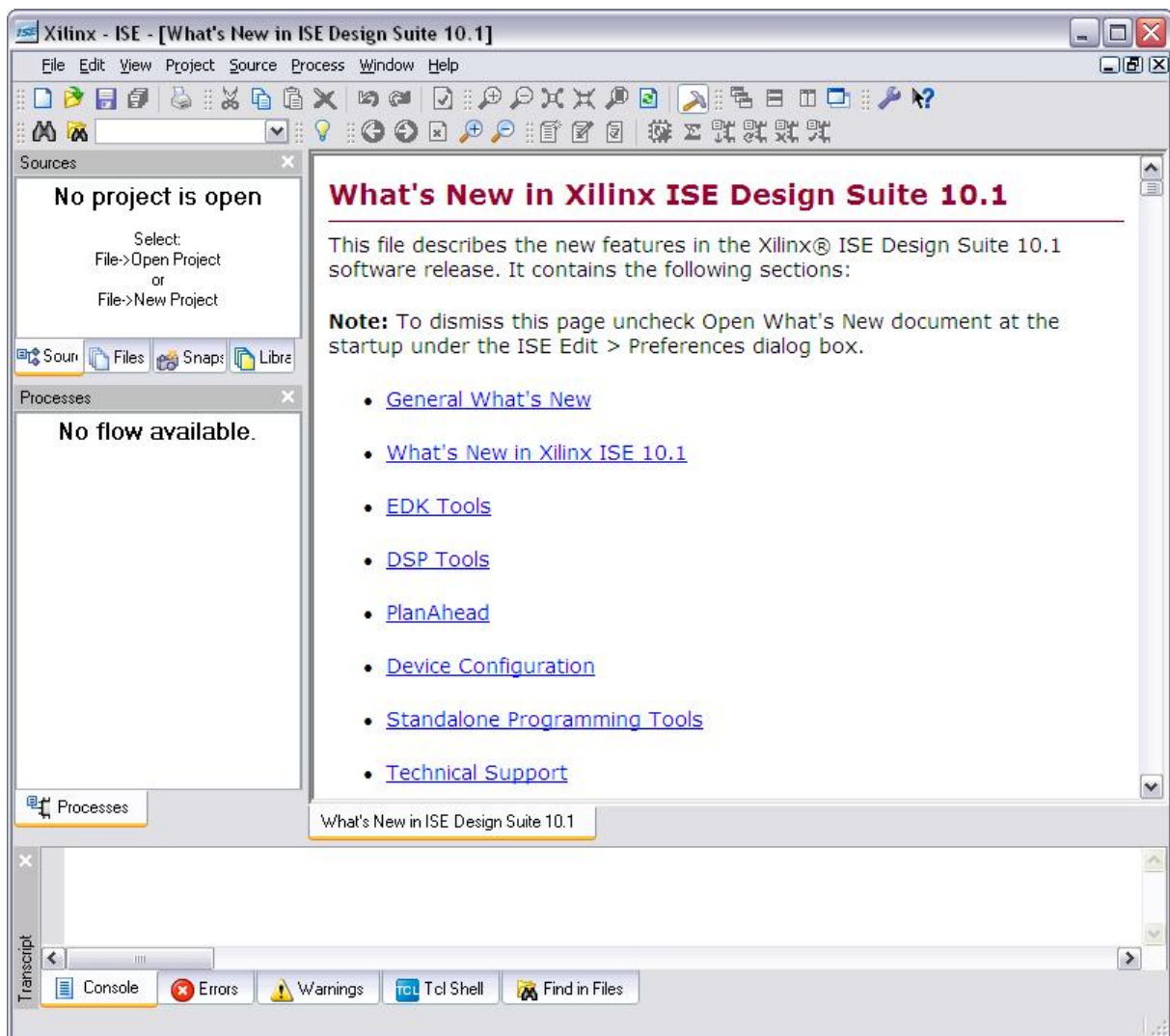
Create a new ISE project by selecting **File -> New Project**. When prompted for a project name and location, use the **Browse** button to select and/or create a directory in which the new ISE project should be generated, and provide a name for the new project as shown below:
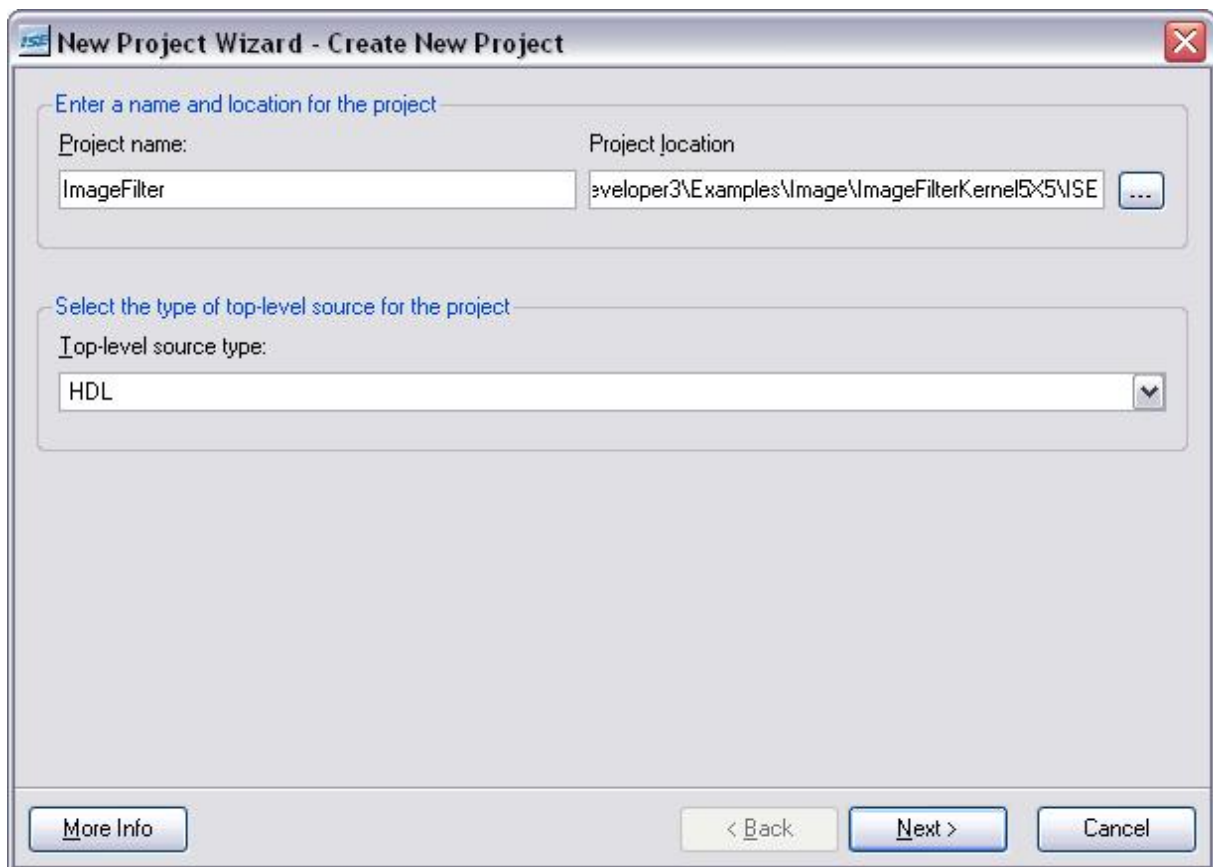
In this case, we have chosen to create a new project called **ImageFilter** within our
ImpulseFilterKernel5X5 project directory.

Click **Next** to advance to the device selection dialog. For demonstration purposes we will select a Xilinx
Virtex-5 LX50 device, as shown below:
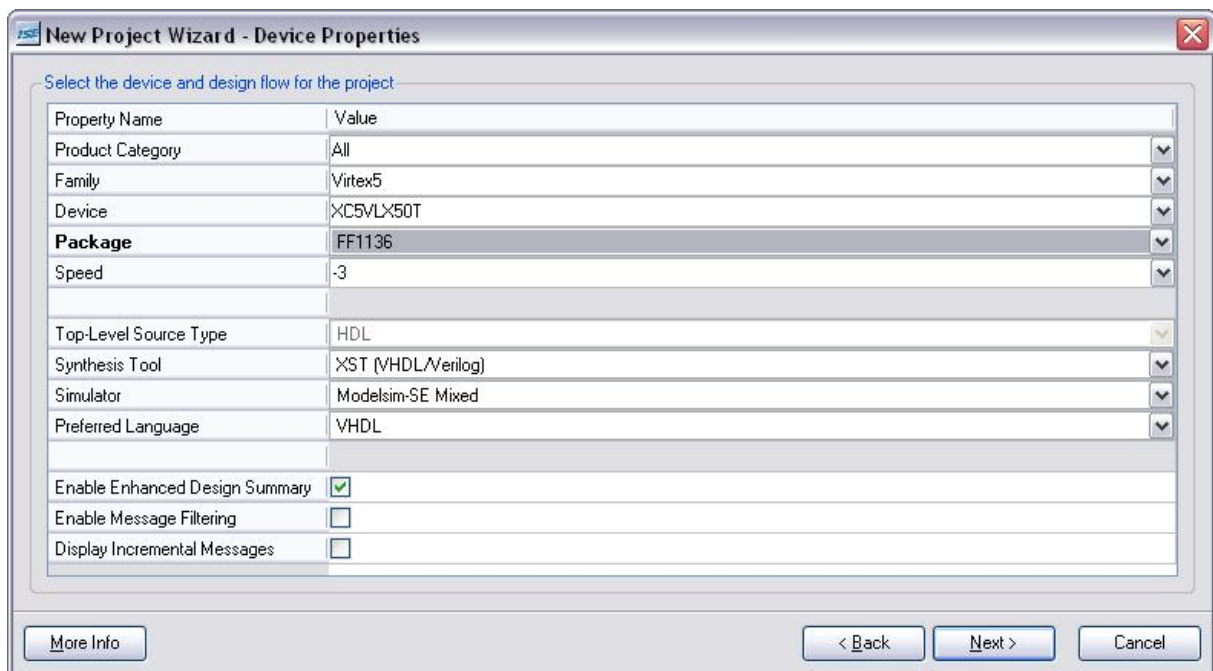
Note that we have selected **Preferred Language** to **VHDL**.

Click **Next** to continue.

On the screen that appears, we are asked if we wish to create a new source file. We will need to create one source file at this point. Click the **New Source** button to invoke the New Source dialog.

For the **Source File Type**, choose **VHDL Library**. For the **File Name**, enter "impulse" as shown:
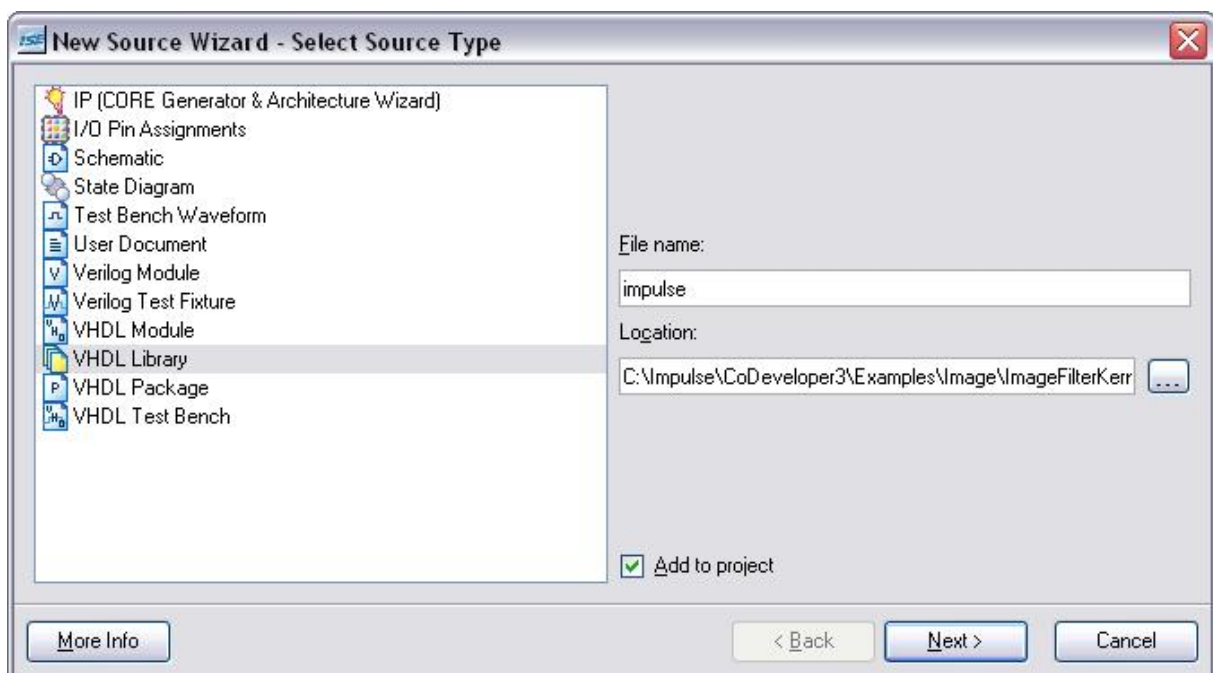
Click **Next** to continue, the click **Finish** to exit the New Source Wizard.

Click **Next** to continue to the Add Existing Sources dialog. We will not add sources at this point.

Click **Next** to continue.

Click **Finish** to create the new project.

Adding Source Files to the Impulse Library

When the project appears, click on the **Libraries** tab of the **Sources** window to view the project libraries. You will see three libraries listed as follows:
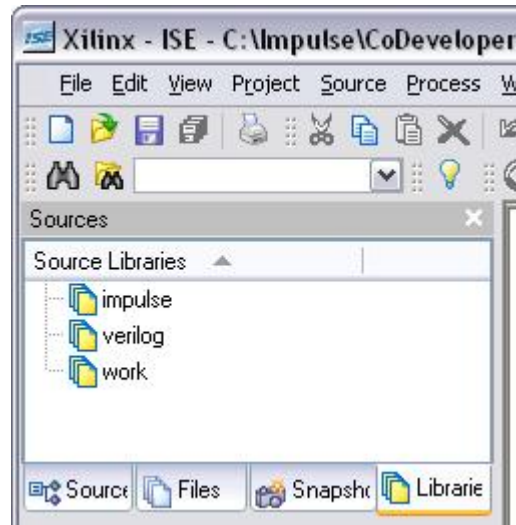
To add the Impulse C library HDL files, right-click over the "impulse" entry in the **Source Libraries** area, then select **Add Source** from the popup menu. A file selection dialog will appear. Navigate into the **hw/lib** subdirectory of your project, and highlight all of the VHDL source files as shown below:
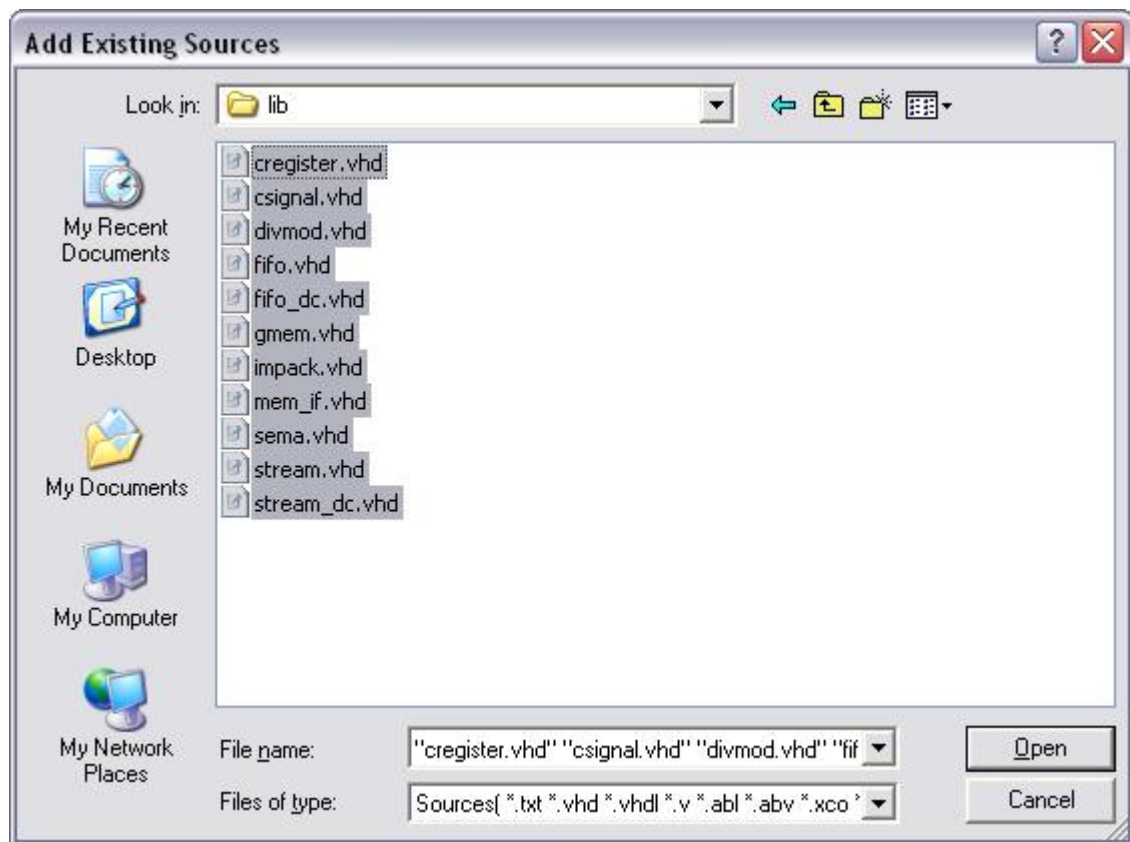
Click **Open** and **OK** to add the selected files to the project.

Adding Source Files to the Work Library

Next, repeat this method to add the two VHDL source files **img_comp.vhd** and **img_top.vhd** from the **hw** directory, to the **work** library as shown below:
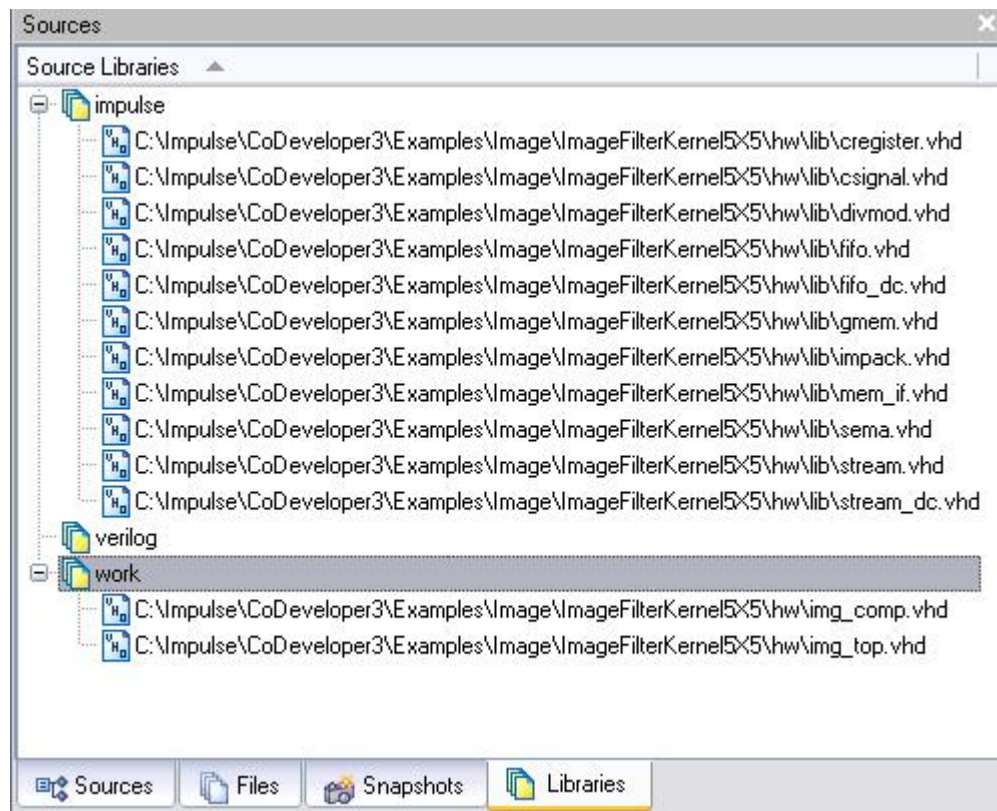
Setting the Top Level Module

Next click on the **Sources** tab to see the source files you have added and their hierarchy.

Find the entry named **img_arch_sl** in file **img_top.vhd**. Use your right mouse button and set the **img_arch_sl** entry to the Top-Level Module as shown below:
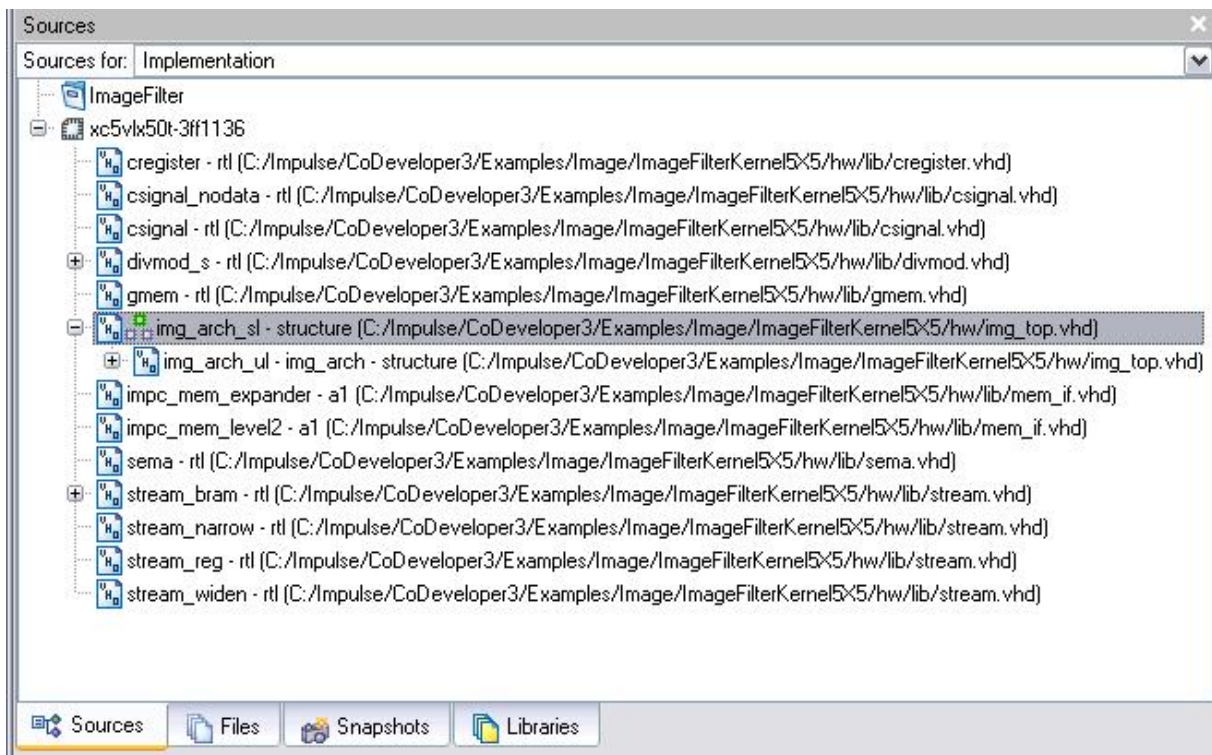
Synthesizing the Hardware

You are now ready to start the synthesis process. In the **Processes** window, double-click on **Synthesize - XST** to start FPGA synthesis:
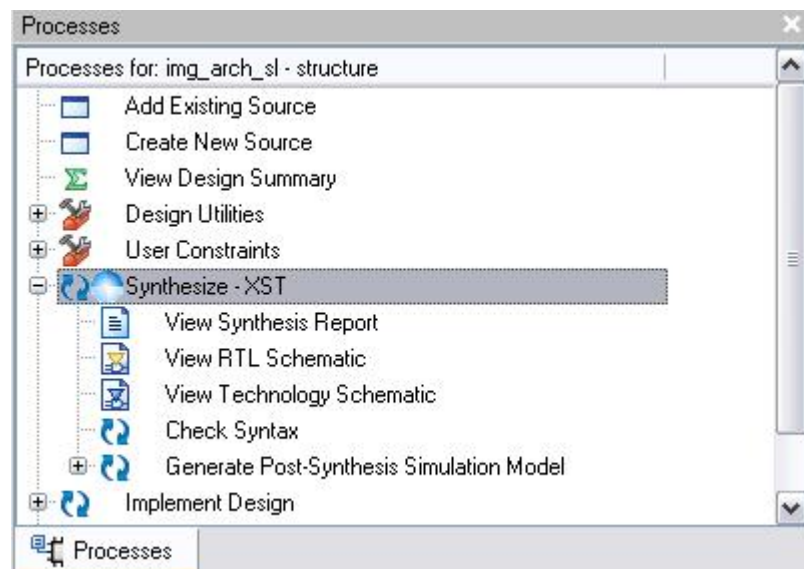
After synthesis has completed you will be able to view a report of the results:

ImageFilter Project Status (03/30/2009 - 16:00:25)			
Project File:	ImageFilter.ise	Current State:	Synthesized
Module Name:	img_arch_sl	• Errors:	No Errors
Target Device:	xc5vlx50t-3ff1136	• Warnings:	497 Warnings
Product Version:	ISE 10.1.01 - WebPACK	• Routing Results:	
Design Goal:	Balanced	• Timing Constraints:	
Design Strategy:	Xilinx Default (unlocked)	• Final Timing Score:	

ImageFilter Partition Summary			[-]
No partition information was found.			

Device Utilization Summary (estimated values)			[-]
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	2144	28800	7%
Number of Slice LUTs	1727	28800	5%
Number of fully used LUT-FF pairs	1020	2851	35%
Number of bonded IOBs	41	480	8%
Number of Block RAM/FIFO	2	60	3%
Number of BUFG/BUFGCTRLs	2	32	6%
Number of DSP48Es	3	48	6%

E Design Suite 10.1   Σ Design Summary

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 FPGA hardware.

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

www.ImpulseAccelerated.com/Tutorials