XAPP901 (v1.0) December 16, 2005

Accelerating Software Applications Using the APU Controller and C-to-HDL Tools

Author: Kunal Shenoy

Summary

Platform-FPGA software applications are significantly faster when critical functions are moved to the hardware domain and a high bandwidth data transfer mechanism is used to exchange data between the hardware and software. This application note describes how C-to-HDL tools can easily create a hardware coprocessor from a critical function in the software system. These tools enable software engineers with cursory hardware knowledge to leverage the advantages of hardware-software co-design. The Auxiliary Processor Unit (APU) controller closely couples the embedded PowerPC™ processor and the Fabric Coprocessor Module (FCM), and provides a low-latency, high-bandwidth communication path. This application note demonstrates an accelerated Mandelbrot image generation application by moving computation-intensive functions to the hardware domain and attaching it to the PowerPC processor using the Virtex™-4 FX APU controller.

Introduction

The Mandelbrot image, a classic example of fractal geometry, is widely used in the scientific and engineering community to simulate chaotic events such as weather. Fractals are also used to generate textures and imaging in video rendering applications. Mandelbrot images are described as self-similar; on magnifying a portion of the image, another image similar to the whole is obtained.

The Mandelbrot image is an ideal candidate for hardware-software co-design because it has a single computation-intensive function. Making this critical function faster by moving it to the hardware domain, significantly increases the speed of the whole system. The Mandelbrot application also lends itself nicely to clear divisions between hardware and software processes, making it easy to implement using C-to-HDL tools. In this application note the CoDeveloper™ toolset, provided by Impulse Accelerated Technologies, is used as the C-to-HDL toolset.

A software-only Mandelbrot C program is modified to make it compatible with the C-to-HDL tools. These changes include: division of the software project into distinct processes (independent units of sequential execution), conversion of function interfaces (hardware to software) into streams, and adding of compiler directives to optimize the generated hardware. The CoDeveloper toolset subsequently creates Pcores that are easy to import into the Xilinx Platform Studio™ (XPS) tool.

The generated Pcore is attached to the APU controller interface and tested. The PowerPC APU controller interface is an excellent solution for the required hardware-software data exchange interface due to its low overhead and high bandwidth capability.

Overview of Key Concepts

CoDeveloper Software

The steep learning curve associated with hardware design prevents many software engineers from taking advantage of hardware-software co-design in embedded systems. The CoDeveloper toolset, developed by Impulse Accelerated Technologies, makes the process easier by converting untimed C functions into HDL code. CoDeveloper software includes the Impulse C™ function library (for individual hardware platforms and for desktop simulations) and compiler (software to hardware conversion). The CoDeveloper toolset forms a development environment around the Impulse C library and compiler that includes debugging tools. DSP, image processing, encryption/decryption, bioinformatics, and high-performance embedded systems fall within the CoDeveloper toolset application domain.

Once a critical software function is identified, the CoDeveloper toolset converts it to hardware and stores it in a format easily imported into XPS. The CoDeveloper toolset provides additional capabilities including desktop simulations for logical verification and the Stage Master Explorer™ tool to optimize C code and generated hardware. More information can be obtained at http://www.impulsec.com.

Impulse C software is based on the Communicating Sequential Processes (CSP) programming model. CSP consists of well-defined, individual, and distinct units of sequential execution called processes. These processes communicate with each other through synchronization buffers called streams. In CoDeveloper systems, each process is classified as a hardware or software process, communicating through these streams. Function calls to the Impulse C library are used to create the processes and to perform reads and writes to streams. These abstract streams are mapped to the target platform technologies via additional library calls during the implementation phase. The platform support package assigned to the project determines the technology on the target platform. The APU controller interface on a Virtex-4 device is the target platform in this application note.

Impulse C software is not expected to outperform hand-optimized HDL in terms of performance or size. The CoDeveloper toolset is positioned for engineers lacking the time or educational resources for conversion, and are not concerned about a slight performance or size penalty associated with an automated process performing the conversion. Figure 1 is a flow diagram of the CoDeveloper toolset.

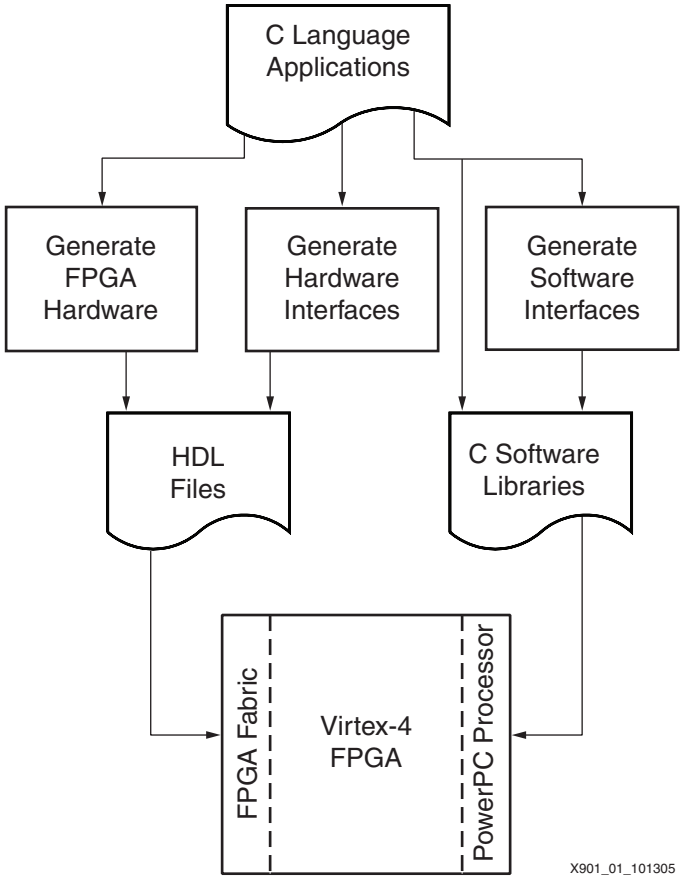


Figure 1: **CoDeveloper Toolset Developed by Impulse Accelerated Technologies**

CoDeveloper Toolset to XPS Tool Flow

Figure 2 shows the CoDeveloper toolset integrated with the XPS tool flow. The tool flow primarily involves finding critical functions in a legacy project, converting them to a Pcore using the CoDeveloper toolset, and using that Pcore in the FPGA system to speed it up.

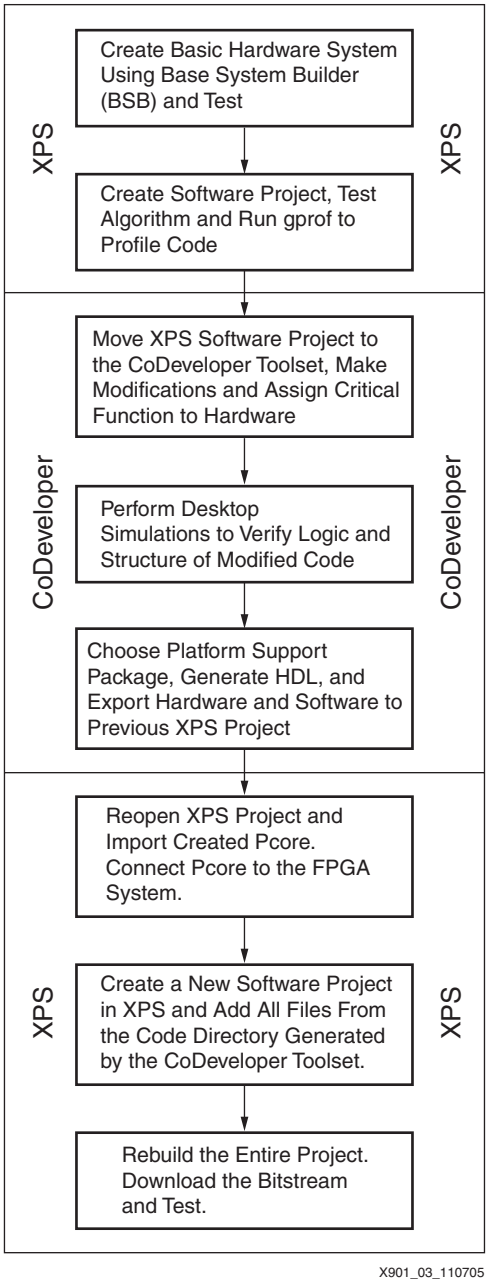


Figure 2: **CoDeveloper to XPS Tool Flow**

The steps in the initial Xilinx Platform Studio (XPS) flow are:

1. A basic hardware system in Base System Builder (BSB) is created and tested.
2. A software project within the XPS project is created, and the user algorithm is coded and tested. If the project does not meet performance specifications, the flow continues to Step 3.
3. The gprof application or any other code-profiling application is run to find the critical function in the XPS software project.

The steps in the CoDeveloper flow are:

4. The software project is copied from XPS, and the code is modified to make it compatible with the Impulse C software. Critical functions found from profiling the code are assigned to hardware.
5. Desktop simulations are performed to verify the program structure is properly implemented.
6. A platform support package is chosen, HDL is generated, and is exported to the XPS project together with the software.

The steps in the final XPS flow are:

7. The created hardware is imported as a Pcore and connected to the FPGA system.
8. A new software project is created in XPS and the software application files generated by the CoDeveloper toolset are included.
9. The entire project is rebuilt and the bitstream is downloaded to the target to test.

APU Controller

The APU controller provides a flexible and high-bandwidth data transfer mechanism between FCMs and the embedded PowerPC processor on Virtex-4 FX FPGAs. The APU interface is connected directly to the instruction pipeline and to one or more FCMs. The typical latency associated with arbitration on a peripheral bus (PLB or OPB) is absent. This interface is ideal for FCMs in the critical path where the transfer of data can create a system performance bottleneck. Figure 3 shows a block diagram of the APU controller.

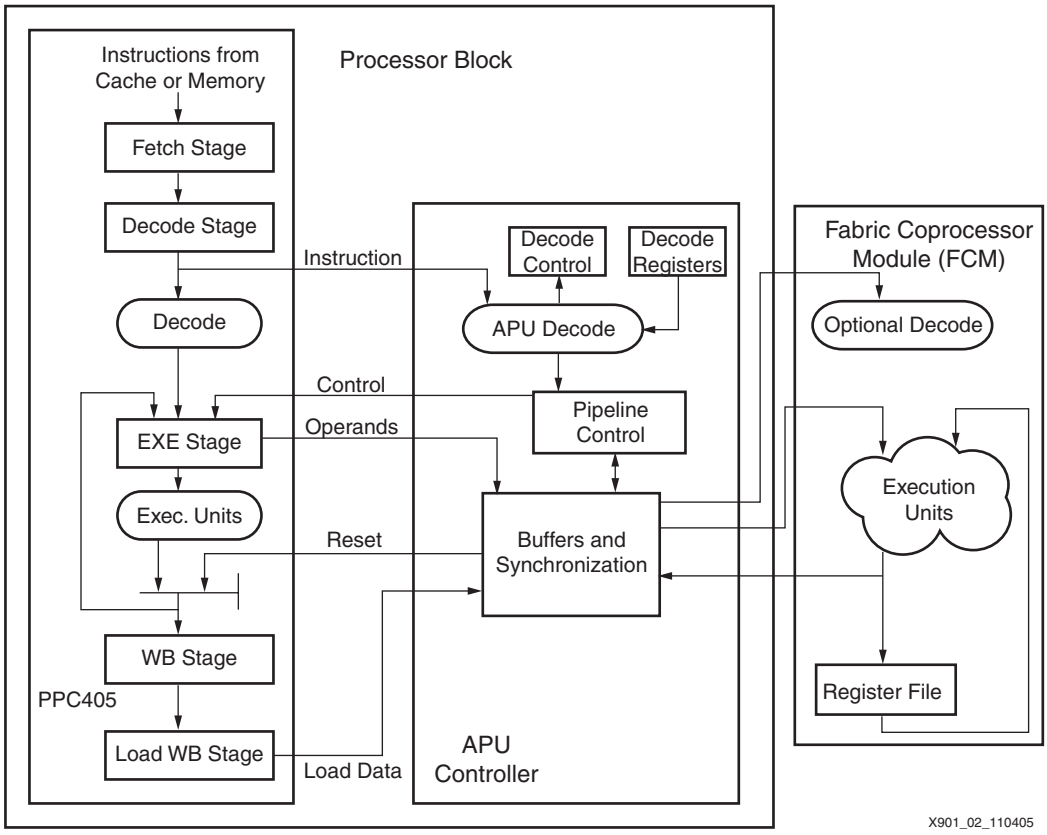


Figure 3: **APU Controller Processing Operative Block Diagram**

The APU controller performs two main functions:

- It provides a synchronization mechanism between the PowerPC processor and the FCM running at a lower clock rate.
- It decodes instructions or allows the FCM to decode instructions. Execution, however, is always carried out by the FCM.

When the instruction is due for decoding, it is presented to both the PowerPC processor and APU controller. If the instruction is not recognized as a CPU instruction, the PowerPC processor looks for a response from the APU controller to signal a valid instruction. If valid, the required operands are fetched and passed to the APU for processing. Instructions directed towards the FCM can be either predefined in the Instruction Set Architecture (ISA), such as floating-point instructions, or can be user-defined instructions.

The CoDeveloper toolset creates hardware cores designed to interface with the APU interface for easy integration into FPGA systems using XPS. Impulse C software currently uses the load/store instructions, predefined by the ISA, to transfer data between the PowerPC data memory system and the FCM.

This application note uses the APU interface available on the Virtex-4 FX FPGA. With a few changes, as described in the CoDeveloper documentation, the application can easily be ported to other hardware technologies, such as the MicroBlaze™ processor with its Fast Simplex Link (FSL) or the PowerPC processor with its Processor Local Bus (PLB). Though the implementation details are transparent to the user, the user needs to be aware of options available for hardware-software data interfaces.

Mandelbrot Image Calculation

Mandelbrot images are created by calculating the color of each pixel of the image using the iterative computation of a formula with the previous results used in the present calculation. The Mandelbrot image is formed by repeatedly squaring a complex number and adding another complex number. The complex number added to the equation is constant for a particular pixel and is changed by a fixed amount for each pixel.

To generate a Mandelbrot image, the image corners are first given complex number constants. The difference between the same axis corner values is divided by the number of pixels needed. This number is then added to the complex number constant (c) whenever traversing pixels. Equation 1 provides the initial value of Z. In Equation 2, c is the constant, complex number.

$$Z_0 = 0 + j0 \quad \text{Equation 1}$$

$$Z_{N+1} = Z_N^2 + c \quad \text{Equation 2}$$

Equation 2 is computed repeatedly for each pixel until a maximum iteration count is reached or the value of Z_N diverges towards infinity. A pixel's color in the image depends on whether that pixel is in the Mandelbrot set and the number of iterations it takes to determine that it is in the set. Specifying a larger maximum iteration count provides a better quality image but also increases the associated computation time.

Software-Only Implementation of Mandelbrot Function

Software-Only Algorithm

Figure 4 lists the software-only algorithm of the Mandelbrot generation function. The code implementation of Algorithm 1 is available in the reference design in the `mand_sw_only.c` file.

```
For each horizontal line
  For each pixel of the horizontal line
    Z = 0
    Number of iterations = 0
    Do
      Square the complex number Z from the previous iteration
      Add the complex constant number to Z
      Increment number of iterations by one
    While number of iterations has not exceeded max count or convergence condition
    of Z fails
    If number of iterations is equal to max count then
      Pixel color = black
    Else
      Pixel color is proportional to number of iterations taken
    Plot the pixel on the screen through the VGA port
```

Figure 4: **Algorithm 1: Software-only Algorithm of the Mandelbrot Generation Function**

Implementation Details

The implementation is a fixed-point version of the Mandelbrot set generator. Hardware is inefficient at implementing floating-point calculations, and the CoDeveloper toolset currently does not support it. Thus to maintain fairness between the comparison of the software-only and hardware projects, the software version uses a fixed-point version, too.

At several places, the preprocessor directive of `#ifndef IMPULSE_C_TARGET` is used to perform certain functions if the program is run in desktop simulation mode in the CoDeveloper system. This directive allows users to perform additional analyses of the application and functionally verify the system while desktop simulations are taking place.

Parameters are passed to the `mand_sw_only()` function as arguments from the calling function. These parameters greatly affect the image generated on the screen.

The calculation of the Mandelbrot image is done using three nested loops. The outermost loop traverses through horizontal lines. The next inner loop traverses all pixels across a horizontal line of the image before moving to the next line. In the innermost loop, the test number is repeatedly squared and added to the complex constant. This loop is terminated when the number of iterations reaches a maximum count or the number starts diverging towards infinity.

If the point diverges towards infinity, it is assigned a color proportional to the number of iterations taken to reach the conclusion. If the point does not diverge by the time the maximum number of iterations is reached, then it is assigned black. The pixels are then displayed on the LCD display via an external function.

Because pixel calculation is the major component of the critical path of the system, this function is moved to hardware using the CoDeveloper toolset and connected to the APU controller. For applications where the critical function is not clear, a simple analysis using the gprof tool is sufficient.

Making Legacy Mandelbrot Code Impulse C Compatible

Defining Processes and Streams in the System

Processes, which are fundamental units of computation in the Impulse C software, execute sequentially. This model of parallel programming is different from the usual software model of a monolithic main function which calls lower hierarchy subroutines as required. In Impulse C software, processes are clearly demarcated and communicate between themselves only through communication channels called streams. The processes only depend on one another via calls to blocking stream read or stream write functions.

For the Mandelbrot application, the processes are divided as shown in Figure 5. The pixel calculation function is assigned to hardware, which reduces its computation time. With the assistance of the high-bandwidth APU controller, the speed of the critical path of the system is increased.

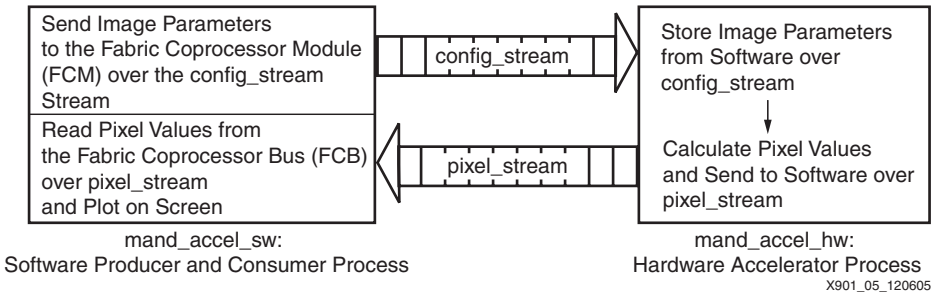


Figure 5: **Basic Structure of the Mandelbrot Application with Processes and Streams**

Once implemented in hardware, software functions can communicate with the hardware via function calls to streams, including the `co_stream_read` and `co_stream_write` functions. Because streams are unidirectional, two streams are used for this application: one to send the configuration data to the hardware and the second to read the pixel values back into the PowerPC processor. These streams also provide the buffering and synchronization required between the hardware and software domains.

Important Impulse C Functions

There are four key Impulse C Stream functions:

co_stream_open

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

This function, which is called from within a process, initializes a stream to a known state.

co_stream_write

```
co_error co_stream_write(co_stream <stream_name>, const void *<buffer>, size_t
<buffer_size>);
```

Processes use this function to write to an opened stream from `*buffer`. Streams are the mechanism through which processes communicate.

co_stream_read

```
co_error co_stream_read(co_stream <stream_name>, void *<buffer>, size_t
<buffer_size>);
```

Processes use this function to read in data into `*buffer` from open streams.

co_stream_close

```
co_error co_stream_close(co_stream <stream_name>);
```

This function closes the calling process's side of the stream by sending a end-of-stream signal to the other process.

Adding a Configuration Function

Every Impulse C project requires a configuration function that specifies the processes in the system and whether the process is implemented in hardware or software. The configuration function also declares the streams created and how they are used to interconnect the processes. Streams and processes are created via impulse C function calls in the configuration function.

Process Functions

- co_process_create

```
co_process co_process_create(const char *<name>, co_function <function_name>, int
argc,….);
```

This function, called from within the system's configuration function, creates a process out of <function_name>.

- co_process_config

```
co_error co_process_config(co_process <process_name>, co_attribute <attribute>,
const char *<value>);
```

This function, called from within the system's configuration function, changes attributes of a created process. As of this printing, only the co_loc attribute is supported, which specifies the FPGA on which the process is to be implemented in a multi-FPGA system.

Stream Functions

- co_stream_create

```
co_stream co_stream_create(const char *<stream_name>);
```

Called from within the system's configuration function, this function creates a stream called <stream_name>.

System Functions

- co_architecture_create

```
(co_architecture) <architecture_name> = co_architecture_create(const char*
<name_for_simulations>, const char* <architecture>, co_function
<configuration_function>, void* <arguments>);
```

This function associates the user's application with a specific architecture definition.

- co_execute

```
co_execute((co_architecture) <architecture_to_execute>);
```

This function starts execution of an Impulse C system.

- co_initialize

```
(co_architecture) <architecture_name> = co_initialize(void* <arguments>);
```

This function defines the relation between the application's main function and its configuration function.

Adding #pragmas

Pragmas are compiler-specific directives, which guide the software-to-hardware compiler in Impulse C software. Currently, Impulse C software allows pipelining and unrolling of loops to optimize the logic created either for performance or space. Maximum logic delays of pipeline stages can also be specified, determining the maximum frequency at which the created peripheral can be clocked when implemented in the FPGA fabric. Clock boundaries can be introduced into the untimed C application.

Fixed-Point Conversion

Real numbers are frequently converted to fixed-point representations in hardware because they are quicker to process and use fewer resources. Fixed-point numbers have fixed bit lengths used to represent the sign, integer, and decimal portions of the value. The exact division of space is up to the user and is a trade-off between the size of the variable, its range, and its precision. Any operation performed on fixed-point variables can potentially cause errors if care is not taken to maintain the correct format.

If a hardware function utilizes floating-point variables, they have to be modified to fixed-point versions. The Impulse C software provides a number of fixed-point data types and macros to perform calculations with fixed-point numbers. [Ref 7] is an application note on floating-point to fixed-point conversion.

Desktop Simulations

Desktop simulations allow users to verify the functionality of their mixed mode systems before implementation in the FPGA. Using simulations, logical errors can be caught early in the design cycle and corrected. Impulse C libraries for desktop simulations can be used in any popular development environment, such as Visual C or GCC. Users can utilize familiar tools, such as GDB, to debug their designs.

To simulate interfaces, the CoDeveloper toolset includes the CoMonitor Application Monitor. The CoMonitor application is started prior to running the simulation. It allows users to capture messaging, the streamed data values and other instrumenting information. Each process in the user project can open a log window using the `cosim_logwindow_create()` function to monitor the execution of that process.

Hardware-Software Implementation of the Mandelbrot Function

Figure 6 shows how the Mandelbrot image generator function, which is Impulse C compatible, is implemented in hardware and software.

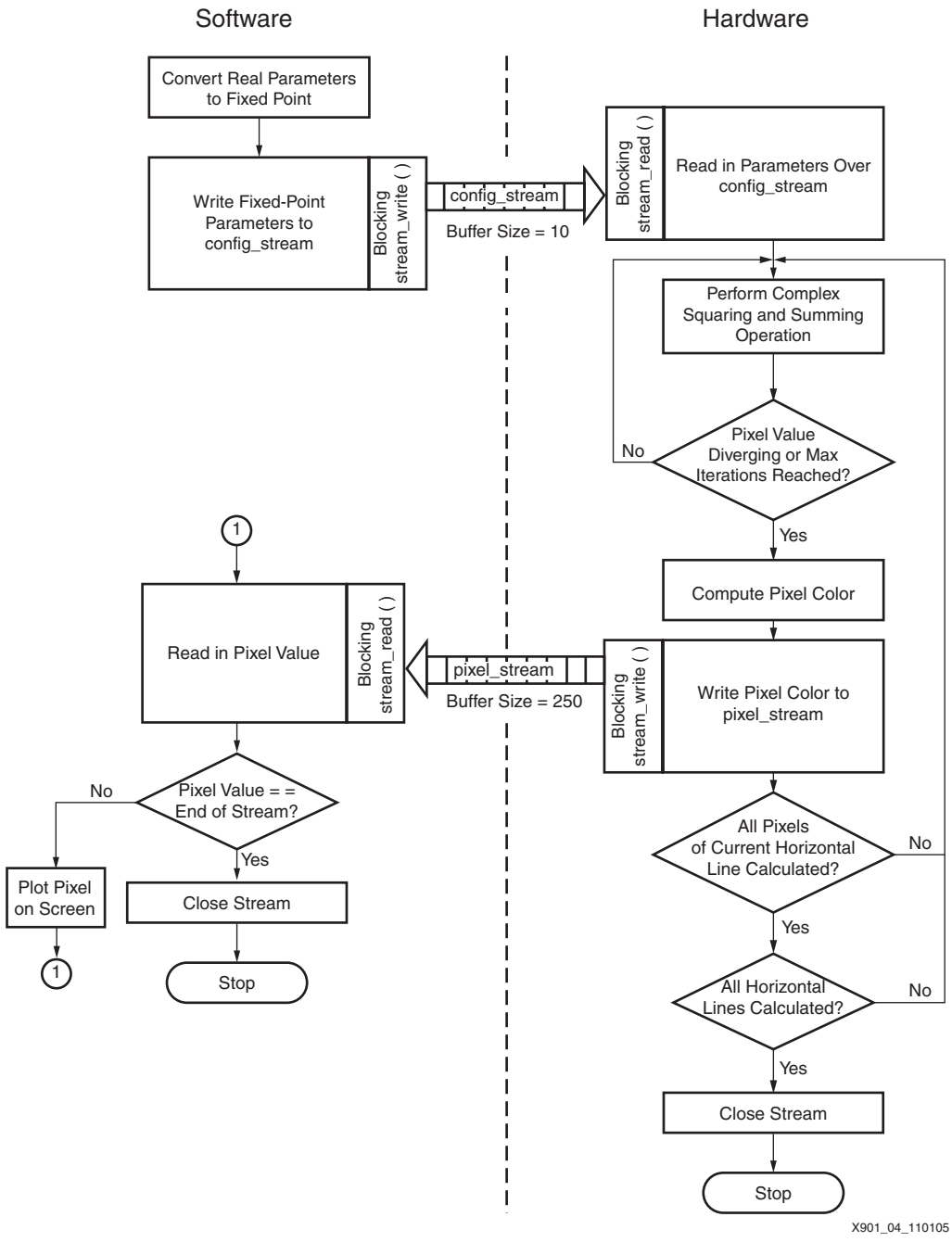


Figure 6: **Flowchart of Impulse C Compatible Mandelbrot Image Generation Application**

Converting C Code to a Pcore

Figure 7 lists the algorithm of the Impulse C compatible Mandelbrot function. The algorithm shown in Figure 4 has been changed in Figure 7, and the changes are in bold type.The implementation of this algorithm is available in the reference design in the `mand_accel_hw.c` file.

```
Open the parameter input stream to read in image parameters
Read in the image parameters from the stream
Open the pixel output stream to write out pixel colors
For each horizontal line
  For each pixel of that horizontal line
    Z = 0
    Number of iterations = 0
    Do
      Square the complex number Z from the previous iteration
      Add the complex constant number to Z
      Increment number of iterations by one
    While number of iterations has not exceeded max count or convergence condition
    of Z fails
    If number of iterations is equal to max count then
      Pixel color = black
    Else
      Pixel color is proportional to number of iterations taken
    Write the pixel value to the pixel output stream
Close pixel output stream
Close parameter input stream
```

Figure 7: **Algorithm 2: Impulse C Compatible Mandelbrot Function**

Implementation Details

The following code shows an implementation of the Impulse C compatible Mandelbrot function.

```
//mand_accel_hw() is the mandelbrot generation function to be converted to a HW process.
//config_stream = stream over which mand_accel_hw() reads in parameters from mand_accel_sw().
//pixel_stream = stream over which mand_accel_hw() sends pixel values to mand_accel_sw().
//mand_accel_hw takes in image parameters from mand_accel_sw, calculates pixel values
//and sends it back to mand_accel_sw.
void mand_accel_hw(co_stream config_stream, co_stream pixel_stream)
{
    co_int32 xmax, xmin, ymax, ymin, dx, dy;//image parameters
    co_uint24 B, G, R, BGR;                 //pixel color variables
    co_int32 i, j, k;                       //loop variables
    co_int32 c_imag, c_real;                //complex constant number added to product of squaring
    co_int32 two, four;                     //used to hold integer constants in fixed-point format
    co_int32 result, tmp;                   //calculation variables
    co_int32 z_real, z_imag;                //complex number used in calculation

    //constant integers assigned in fixed point format
    //#define FXCONST(a) FXCONST32(a,FRACBITS)\
    //#define FXCONST32(a,DW) ((uint32)((a)<<DW))
    two = FXCONST(2);
    four = FXCONST(4);

    //co_stream_open() resets the internal state of the stream
    //Usage: co_stream_open(stream_name, O_RDONLY/O_WRONLY, size_of_stream_element);
    //Streams are unidirectional and hence they have to be opened to be read
    //or written to exclusively
    co_stream_open(config_stream, O_RDONLY, INT_TYPE(32));

    //Read in parameters of Mandelbrot image from sw function over the config_stream stream.
    //Function will return error code and loop will exit, if SW process on other end writes
    //an end-of-stream token.
```

```
    //co_stream_read() reads data from a non-empty stream or else it blocks.
    //Usage: co_stream_read(stream_name, variable_address, data_read_size);
    //xmax and ymin are not used in this function. The parameters are included for completeness.
    while (co_stream_read(config_stream, &xmax, sizeof(co_int32)) == co_err_none)
    {
         co_stream_read(config_stream, &xmin, sizeof(co_int32));
         co_stream_read(config_stream, &ymax, sizeof(co_int32));
         co_stream_read(config_stream, &ymin, sizeof(co_int32));
         co_stream_read(config_stream, &dx, sizeof(co_int32));
         co_stream_read(config_stream, &dy, sizeof(co_int32));

       //co_stream_open() resets the internal state of the stream
       //Usage: co_stream_open(stream_name, O_RDONLY/O_WRONLY, size_of_stream_element);
       co_stream_open(pixel_stream, O_WRONLY, UINT_TYPE(24));
       c_imag = ymax;
       for (j = 0; j < YSIZE; j++)              //For each horizontal line
       {
         c_real=xmin;
         for (i=0; i<XSIZE; i++)                //For each pixel of each horizontal line
         {
             z_real = z_imag = 0;
             k = 0;
             //Calculate pixel color
             do
             {
//Hardware optimization pragmas
//See Impulse C documentation for more information
#pragma CO pipeline //pipeline the following code to increase throughput
#pragma CO set stageDelay 200//set max-stage-delay to 200 units.
                            //Units roughly correspond to gate delays of target technology.
                            //Max stage delay determines the maximum
                            //frequency the logic can be clocked.
               //Square complex number, add complex constant and check magnitude
               tmp = z_real;
               z_real = FXMUL(z_real, z_real);
               z_real = FXSUB(z_real, FXMUL(z_imag, z_imag));
               z_real = FXADD(z_real, c_real);
               z_imag = FXMUL(tmp, z_imag);
               //change *2 to <<1, which is more efficiently implemented in HW
               //z_imag = FXMUL(two,z_imag);
               z_imag = z_imag << 1;
               z_imag = FXADD(z_imag, c_imag);
               tmp=FXMUL(z_real, z_real);
               result = FXADD(tmp, FXMUL(z_imag, z_imag));
               co_par_break();//cycle boundary added to make logic fit within 10ns clock cycle.
                            //If removed timing will fail during implementation.
               k++;
             } while ((result < four) && (k < MAX_ITERATIONS));
             //loop will exit if complex number diverges or max iterations is reached
             //Calculate pixel color
             //If number diverged then color assigned depends on number of iterations
             if (k != MAX_ITERATIONS)
             {
                G = (k > 255) ? 255 : k;
                B = R = 0;
             }
             else //Else pixel is colored black
             {
                B = G = R = 0;
             }
```

```
            //Assign color in Blue(8 bit)-Green(8 bit)-Red(8 bit) format for storage in video memory.
            BGR = ((B << 16) & BLUEMASK) | ((G << 8) & GREENMASK) | (R & REDMASK);

            //Write pixel value to stream, which will be read by the SW process on other end of stream.
            //co_stream_write() writes a data value to a stream.
            //Usage: co_stream_write(stream_name, variable_address, size_of_stream_element);
            co_stream_write(pixel_stream, &BGR, sizeof(co_uint24));
            c_real = FXADD(c_real, dx);
          }
          c_imag = FXSUB(c_imag, dy);
        }
        //co_stream_close() writes a end-of-stream(EOS) token to the selected stream
        //When the process on the other end of the stream reads the EOS, it can close its side of the
        //stream
        //Usage: co_stream_close(stream_name);
        //Close pixel_stream after writing all pixel values to it.
        co_stream_close(pixel_stream);
    }
    //End-of-stream token has been read on config_stream, hence close HW side of config_stream
    co_stream_close(config_stream);
}
```

The `co_stream_read` function reads in data placed by another process on the opposite end of the stream. It is used in the hardware function to read in the parameters of the Mandelbrot image. The function then iterates over all the pixels of the image and calculates each color to be plotted on the screen.

```
co_stream_read(config_stream, &xmin, sizeof(co_int32));
```

The *CO pipeline* pragma instructs the compiler to pipeline the generated hardware to speed up execution. Because the amount of logic in each stage of the pipeline directly affects the maximum clock frequency of the fabric, it can be controlled using the stageDelay pragma. The specific stageDelay was chosen after analysis done using the Stage Master Explorer for maximum performance. A unit in the stageDelay parameter represents a unit logic delay in the target FPGA.

```
#pragma CO pipeline
#pragma CO set stageDelay 200
```

These lines of code in the hardware function do the actual calculation and are looped over all the pixels to be generated. Here the complex number is first squared up and then a constant is added to the result.

```
tmp = z_real;
z_real = FXMUL(z_real,z_real);
z_real = FXSUB(z_real,FXMUL(z_imag,z_imag));
z_real = FXADD(z_real,c_real);
z_imag = FXMUL(tmp,z_imag);
//z_imag = FXMUL(two,z_imag);
z_imag = z_imag << 1;
z_imag = FXADD(z_imag,c_imag);
tmp=FXMUL(z_real,z_real);
result = FXADD(tmp,FXMUL(z_imag,z_imag));
k++;
```

The color of the pixel is then calculated and written out to the pixel_stream. The consumer process at the other end of the pixel_stream reads in and plots the data on the screen.

```
B = G = R = 0;
if (k != MAX_ITERATIONS)
{
    G = k > 255 ? 255 : k;
}
BGR=((B<<16)&BLUEMASK)|((G<<8)&GREENMASK)|(R&REDMASK);
```

```
co_stream_write(pixel_stream,&BGR,sizeof(co_uint24));
```

The end of the function closes the stream. Closing the stream, by the process that writes to it, causes an end–of-stream token to be transmitted down the stream. When the consuming process reads an end-of-stream token, it interprets it as an error and the reading function returns a non-zero value. Closing the stream is important to ensure all data in the stream has been read by the consumer function and that the consumer function does not close the stream before all present data is read.

```
co_stream_close(config_stream);
```

The configuration function defines the system in terms of processes and their interconnection via streams. This system has two processes and two created streams. The stream names are passed as arguments to the `co_process_create()` calls to form the connection. Lastly, the `co_process_config()` function specifies which of the processes previously created are to be implemented as hardware processes.

```
//Configuration function of CoDeveloper project
//Every CoDeveloper project must contain a single configuration function, which
//declares processes, streams and connectivity in the system
void config_mand(void *arg)
{
  co_process mand_hw_proc, mand_sw_proc;//system processes
  co_stream pixel_stream, config_stream;//system streams

  //Declaration of streams
  /*Usage : (co_stream)stream_name = co_stream_create("stream_name_for_sims", size_of_stream_element,
    no_of_elements_in_stream);*/
  //STREAM_BUFSIZE = 250. Large enough to hold data for one horizontal line.
  pixel_stream = co_stream_create("pixel_stream", UINT_TYPE(24), STREAM_BUFSIZE);
  //CONFIG_BUFSIZE = 10. Large enough to hold all the image parameters
  config_stream = co_stream_create("config_stream", INT_TYPE(32), CONFIG_BUFSIZE);

  //Declaration of processes
  /*Usage : (co_process)process_names = co_process_create("process_name_for_sims", (co_function)function_name,
    number_of_arguments, arg1, arg2,....); */
  mand_hw_proc = co_process_create("mand_proc", (co_function)mand_accel_hw, 2, config_stream, pixel_stream);
  mand_sw_proc = co_process_create("test_proc", (co_function)mand_accel_sw, 2, config_stream, pixel_stream);

  //mand_hw_proc to be moved to hardware
  //Usage: co_process_config(process, attribute_to_change, attribute_value);
  //currently only attribute_to_change = co_loc and attribute_value = PE0 are
  //supported
  co_process_config(mand_hw_proc, co_loc, "PE0");
}
```

Generated Pcore Characteristics

The Pcore generated by the CoDeveloper toolset has the necessary interface to connect it to the Fabric Coprocessor Bus (FCB). The Pcore is placed in the `pcores/` directory, while the associated drivers are placed in the `drivers/` directory. The Pcore, written in VHDL, maintains the variable names used in the original C function as far as possible. Engineers experienced with HDLs can, if desired, hand-tune the generated HDL to improve performance.

Table 1 shows that significant system improvement is obtained utilizing just 606 slices or 11% of a Virtex-4 FX12 device.

Table 1: **Logic Utilization of Pcore Accelerator**

Resource Type	Used	Available	Percent Utilization
Slices	606	5472	11
Slice Flip-Flops	670	10944	6

Table 1: **Logic Utilization of Pcore Accelerator**

Resource Type	Used	Available	Percent Utilization
4-input LUTs	1044	10944	9
Bonded IOBs	217	320	67
FIFO16s/RAMB16s	2	36	5
DSP48s	20	32	62

The DSP slices perform the numerous multiplications and additions required in the algorithm. Xilinx supplies larger devices such as the XC4VFX140, which contains 192 DSP48 slices, if multiple accelerators are to be implemented.

The block RAMs are used as FIFOs to implement the streams interface to the hardware and software processes. The generated Pcore uses one FIFO16 for the configuration stream and another for the pixel stream.

Connecting the Pcore to the APU Bus

Figure 8 shows the system implementation that connects the Pcore to the APU bus.

The CoDeveloper toolset automatically generates all signals that connect the Pcore to the APU controller or the FCB. These connections are transparent to the user, requiring no manual intervention. Decisions regarding the clock signals have to be made. The generated Pcore has dual-clock streams, which allows the two ends of the stream to be clocked at different rates. Two clocks are useful in this application because the APU controller side (100 MHz) runs faster than the FCM side (50 MHz).

The maximum frequency at which the fabric can be clocked is determined by the stage delay parameter used in the CoDeveloper project. A smaller stage delay allows a higher frequency but increases the number of pipeline stages. The Stage Master Explorer tool within the CoDeveloper toolset performs a parametric analysis of the various pipeline rates that result for various values of stage delay. It also provides a graphical display of the basic blocks identified in the code.

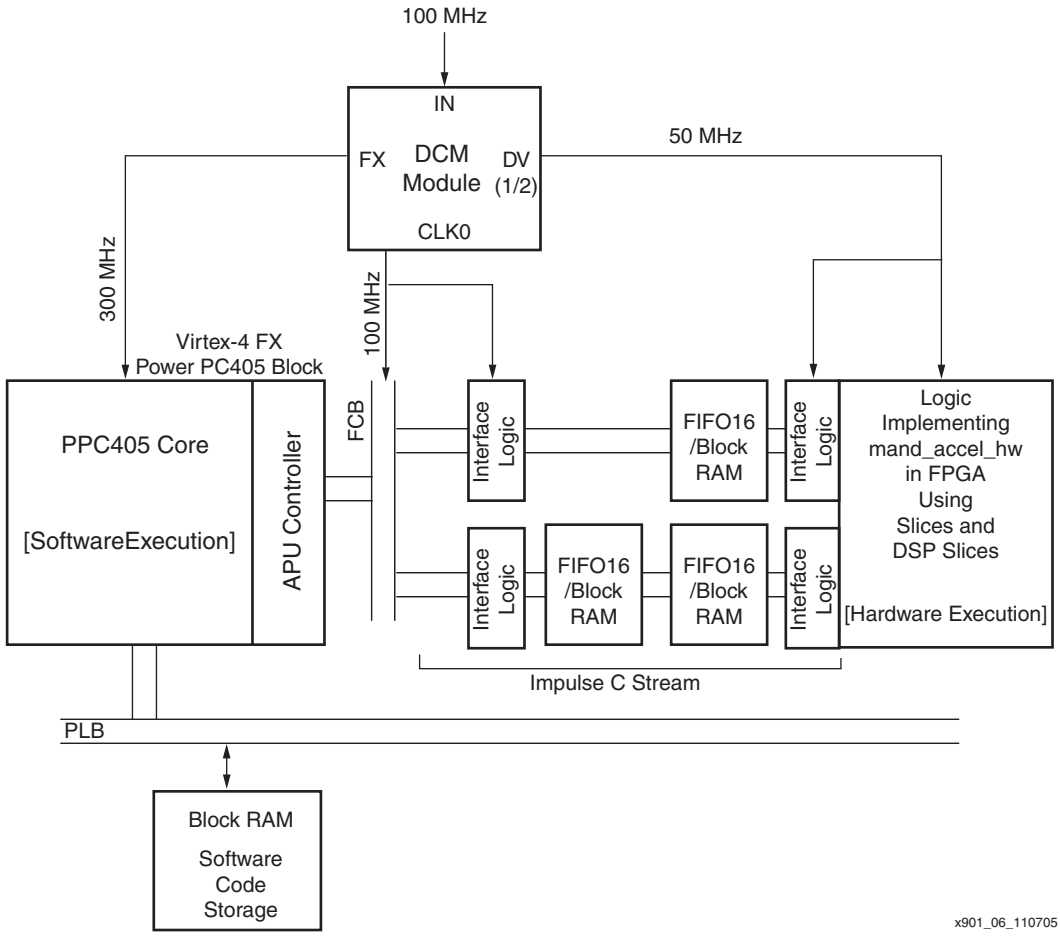


Figure 8: **System Implementation on the Virtex-4 FPGA**

Reference Design

Required Hardware and Software Tools

The required tools for this reference design are:

- Xilinx ML403 Virtex-4 Evaluation Platform or Avnet Virtex-4 FX12 Evaluation Platform with Audio/Visual Daughter card
- Programming cable and board power supply
- VGA Monitor
- ISE v7.1.04i (Service Pack 4) or later
- Xilinx Platform Studio 7.1.02i (Service Pack 2) tool or later
- CoDeveloper v2.01.b.20 toolset (optional)

  The reference design carries a Pcore previously generated by the CoDeveloper toolset and can be used in cases where access to the CoDeveloper toolset is unavailable. Evaluation versions of the CoDeveloper toolset are available at http://www.impulsec.com.

Full-Design Device Utilization Summary

Table 2 summarizes the utilization of devices in the Mandelbrot reference design.

Table 2: **Full-Design Device Utilization Summary**

Resource Type	Used	Available	Percent Utilization
BUFGs	7	32	21
DCM_ADVs	2	4	50
DSP48	20	32	62
JTAGPPCs	1	1	100
PPC405_ADVs	1	1	100
RAMB16/FIFO16	35	36	97
Slices	2829	5472	51
4-input LUTs	3017	10944	24

The single PowerPC processor on the Virtex-4 FX12 device runs the software side of the Impulse C application. Wide multiplications and additions are implemented in the DSP48 blocks in the created Pcore. Program and data are stored in 32 of the 35 used block RAMs, and two block RAMs are used in the Pcore as streams. The final stream is used in the VGA controller Pcore to buffer display line information.

Project Files

The reference design contains multiple project directories for both the Xilinx ML403 and Avnet boards (Table 3). These directories are implementations of the Mandelbrot generation project in incremental states of completion. This structure allows users to incrementally test designs and use fully configured projects as references. The `readme.txt` file in the reference design ZIP file contains more information on running the reference design. As well, UG096: *Implementing a Virtex-4 FX PowerPC System with a C-to-HDL Hardware Coprocessor Accelerator* is a step-by-step tutorial.

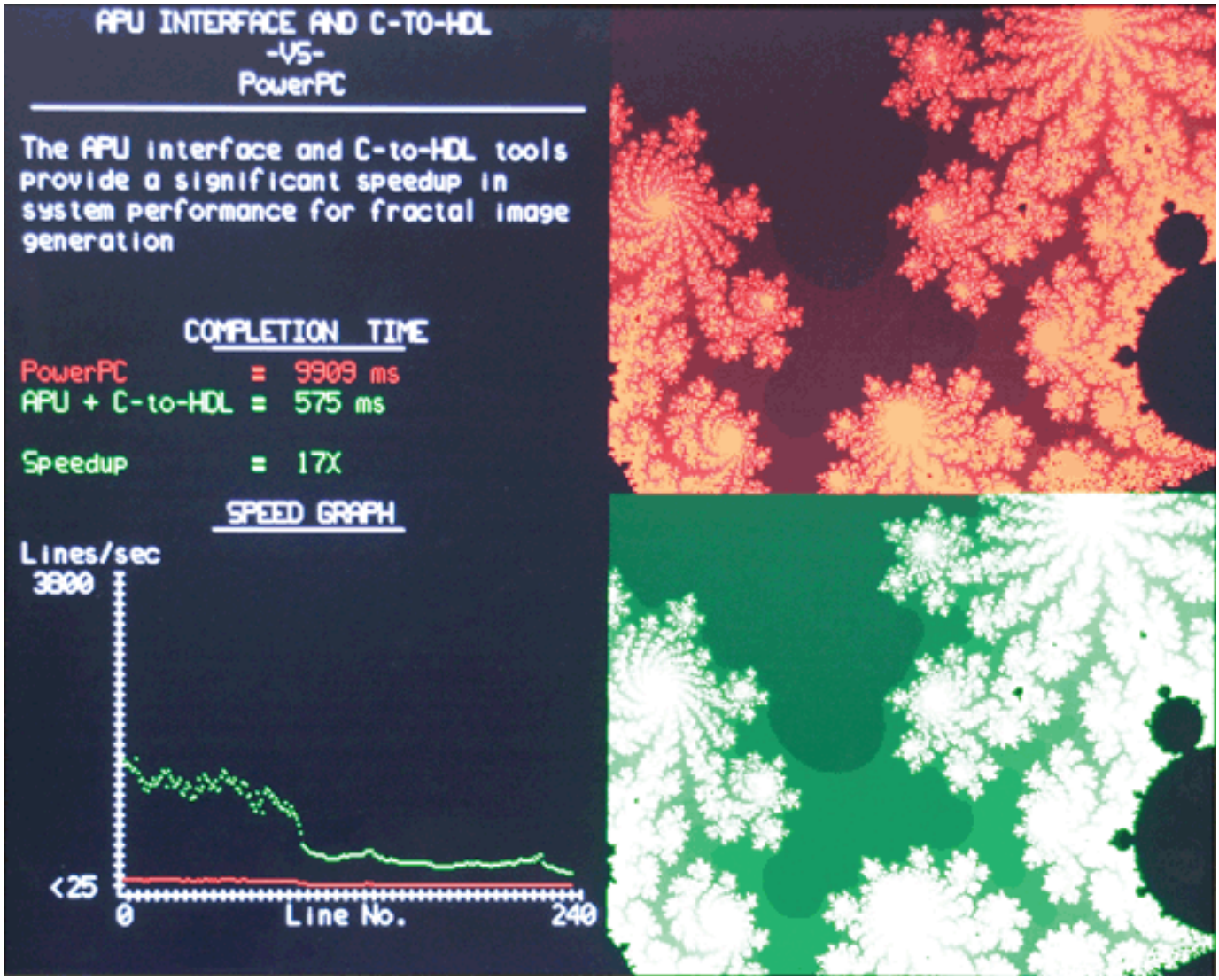
Table 3: **Important Files and Directories in the Reference Design ZIP File**

File Name	Description
`readme.txt`	Text file that explains the reference design and how to run it
`Xilinx_Design\V4FX_Labs\C2HDL_BitFiles\Mandelbrot_Xilinx.bit`	Download-ready Mandelbrot bit file (Xilinx Version)
`Xilinx_Design\V4FX_Labs\C2HDL_BitFiles\Mandelbrot_Avnet.bit`	Download-ready Mandelbrot bit file (Avnet Version)
`Xilinx_Design\V4FXLabs\C2HDL_Lab_Xilinx\`	Contains TFT and Mandelbrot pcore files. Contains application code (Xilinx Version)
`Xilinx_Design\V4FXLabs\C2HDL_Lab_Xilinx_Part2\`	Contains prebuilt basic system (Xilinx Version)
`Xilinx_Design\V4FXLabs\C2HDL_Lab_Xilinx_Part3\`	Contains prebuilt nearly complete system (Xilinx Version)
`V4FX12_xbd.zip`	Contains Xilinx and Avnet board configuration (XBD) files. These are required to insert the TFT controller in the system through base system builder. The file should be unzipped to your EDK installation directory to be visible to XPS.
`Xilinx_Design\V4FX_Labs\C2HDL_CoDeveloper_Project\Mandelbrot.icProj`	CoDeveloper toolset project file. Double-click this file to open it in the CoDeveloper toolset (if installed)
`Xilinx_Design\V4FX_Labs\C2HDL_CoDeveloper_Project\mand_accel_hw.c`	C function that was converted to a pcore using the CoDeveloper C-to-HDL toolset.

Table 3: **Important Files and Directories in the Reference Design ZIP File *(Continued)***

File Name	Description
Xilinx_Design\V4FXLabs\C2HDL_Lab_Avnet\	Contains TFT and Mandelbrot pcore files. Contains application code (Avnet Version)
Xilinx_Design\V4FXLabs\C2HDL_Lab_Avnet_Part2\	Contains prebuilt basic system (Avnet Version)
Xilinx_Design\V4FXLabs\C2HDL_Lab_Avnet_Part3\	Contains prebuilt nearly complete system (Avnet Version)

Figure 9 shows that the speed of the Mandelbrot application is increased by 10x to 17x. This acceleration varies based on the nature of the application, if the application is computation-intensive or data-intensive.



x901_09_110105

Figure 9: **Screen Shot of Reference Design Output**

Conclusion

Software applications can be significantly accelerated by implementing time-critical computation loops in hardware and using the APU controller interface in a Virtex-4 FPGA to exchange data between hardware and software. The CoDeveloper C-to-HDL toolset allows critical functions in legacy software applications to be easily converted to FCMs, resulting in faster computations and smaller logic sizes, because most of the other non-critical computations are implemented on the embedded processor. An FCM attached to the PLB provides significant acceleration. However, the PLB requires arbitration that can hinder the potential increase in performance. In the Virtex-4 FPGA, the APU interface allows a coprocessor to be tightly coupled to the processor pipeline, reducing overhead and further improving mixed-mode system performance.

This application note demonstrates a Mandelbrot image generation application on the Xilinx ML403 development board. Performance is compared between a software implementation and a mixed-mode implementation using the APU interface, giving an acceleration of approximately 17x. Results are shown graphically on a VGA monitor connected to the VGA port of the ML403 board.

References

These documents provide supplemental material useful to this application note:

1. Pellerin, David and Scott Thibault. 2005. *Practical FPGA Programming in C*. Prentice Hall. http://www.impulsec.com/practical/index.html
2. UG070, *Virtex-4 User Guide*.
3. Ansari, Ahmad, Peter Ryser, and Dan Isaacs. *Accelerated System Performance with APU-Enhanced Processing.* http://www.xilinx.com/publications/xcellonline/xcell_52/xc_v4acu52.htm
4. UG018, *PowerPC 405 Processor Block Reference Guide*.
5. XAPP717, *Accelerated System Performance with the APU Controller and XtremeDSP Slices*.
6. UG080, *ML40x Evaluation Platform User Guide*.
7. Bodenner, Ralph. *Fixed-Point Arithmetic in Impulse C*. http://www.impulsec.com/IATAPP106_FIXEDPT.pdf
8. DeAlmo, Joe. *Applications of Fractal Geometry*. http://hypatia.math.uri.edu/~kulenm/honprsp02/
9. Virtex-4 ML403 Embedded Platform. http://www.xilinx.com/ml403
10. UG096, *Implementing a Virtex-4 FX PowerPC System with a C-to-HDL Hardware Coprocessor Accelerator*.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/16/05	1.0	Initial Xilinx release to www.xilinx.com.