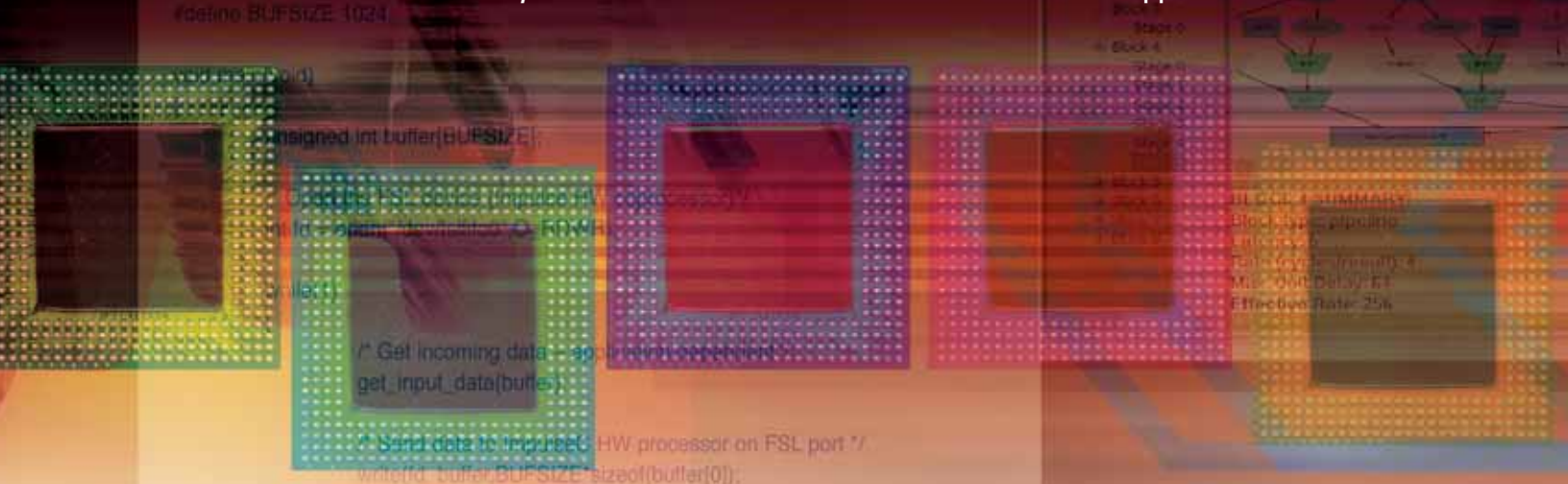


Accelerating FFTs in Hardware Using a MicroBlaze Processor

A simple FFT, generated as hardware from C language, illustrates how quickly a software concept can be taken to hardware and how little you need to know about FPGAs to use them for application acceleration.



by John Williams, Ph.D.
CEO
PetaLogix
john.williams@petalogix.com

Scott Thibault, Ph.D.
President
Green Mountain Computing Systems, Inc.
thibault@gmvhdl.com

David Pellerin
CTO
Impulse Accelerated Technologies, Inc.
david.pellerin@impulsec.com

FPGAs are compelling platforms for hardware acceleration of embedded systems. These devices, by virtue of their massively parallel structures, provide embedded systems designers with new alternatives for creating high-performance applications.

There are challenges to using FPGAs as software platforms, however. Historically, low-level hardware descriptions must be

written in VHDL or Verilog, languages that are not generally part of a software programmer's expertise. Other challenges have included deciding how and when to partition complex applications between hardware and software and how to structure an application to take maximum advantage of hardware parallelism.

Tools providing C compilation and optimization for FPGAs can help solve these problems by providing a new level of programming abstraction. When FPGAs first appeared two decades ago, the primary method of design for these devices was the venerable schematic. FPGA application developers used schematics to assemble low-level components (registers, logic gates, and larger blocks such as counters and adders/subtractors) to create FPGA-based systems. As FPGA devices became more complex and applications targeting them grew larger, schematics were gradually replaced by higher level

methods involving hardware description languages like VHDL and Verilog. Now, with ever-higher FPGA gate densities and the proliferation of FPGA embedded processors, there is strong demand for even higher levels of abstraction. C represents that next generation of abstraction, allowing you to access the resources of FPGAs for application acceleration.

For applications that involve embedded processors, a C-to-hardware tool such as Impulse C (Figure 1) can abstract away many of the details of hardware-to-software communication, allowing you to focus on application partitioning without having to worry about the low-level details of the hardware. This also allows you to experiment with alternative software/hardware implementations.

Although such tools can dramatically improve your ability to create FPGA-based applications, for the highest performance you still need to understand

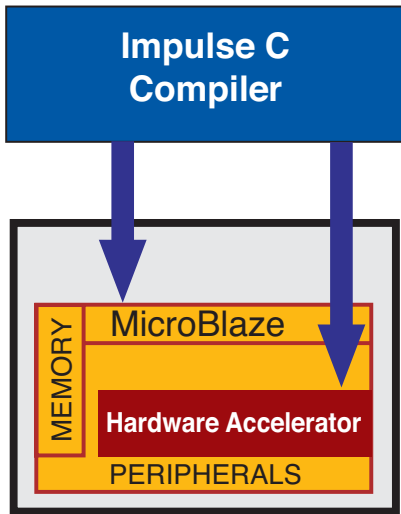


Figure 1 – Impulse C custom hardware accelerators run in the FPGA fabric to accelerate μ Clinix processor-based applications.

certain aspects of the underlying hardware. In particular, you must understand how partitioning decisions and C coding styles will impact performance, size, and power usage. For example, the acceleration of critical computations and inner-code loops must be balanced against the expense of moving data between hardware and software. Fortunately, modern tools for FPGA compilation provide various types of analysis tools that can help you more clearly understand and respond to these issues.

Practically speaking, the initial results of software-to-hardware compilation from C-language descriptions will not equal the performance of hand-coded VHDL, but the turnaround time to get those first results working may be an order of magnitude better. Performance improvements occur iteratively,

through an analysis of how the application is being compiled to the hardware and through the experimentation that C-language programming allows.

Graphical tools (see Figure 2) can help to provide initial estimates of algorithm throughput such as loop latencies and pipeline effective rates. Using such tools, you can interactively change optimization options or iteratively modify and recompile C code to obtain higher performance. Such design iterations may take only a matter of minutes when using C, whereas the same iterations may require hours of even days when using VHDL or Verilog.

Case Study: Accelerating an FFT

The Fast Fourier Transform (FFT) is an example of a DSP function that must accept sample data on its inputs and generate the resulting filtered values on its outputs. Using C-to-hardware tools, you can combine traditional C programming methods with hardware/software partitioning to create an accelerated DSP application. The FFT developer for this example is compatible with any Xilinx® FPGA target, and demonstrates that you can achieve results similar to hand-coded HDL without resorting to low-level programming methods.

Our FFT, illustrated in Figure 3, utilizes a 32-bit stream input, a 32-bit stream output, and two clocks, allowing the FFT to be clocked at a different rate than the embedded processor with which it communicates. The algorithm itself is described using relatively straightforward, hardware-independent C code, with some minor C-level optimizations for increased parallelism and performance.

The FFT is a divide and conquer algorithm that is most easily expressed recursively. Of course, recursion is not possible on the FPGA, so the algorithm must be implemented using iteration instead. In fact, almost all software implementations are written iteratively (using a loop) for efficiency. Once the algorithm has been implemented as a loop, we are able to enable the automatic pipelining capabilities of the Impulse compiler.

Pipelining introduces a potentially high degree of parallelism in the generated

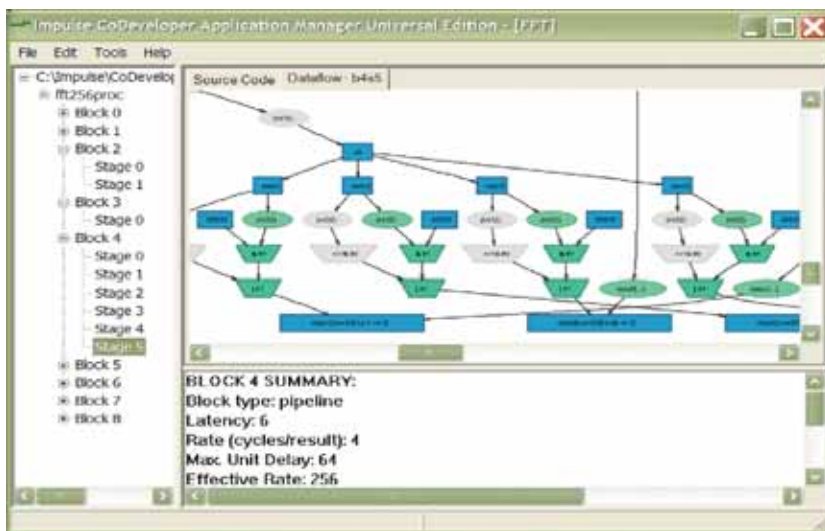


Figure 2 – A dataflow graph allows C programmers to analyze the generated hardware and perform explorative optimizations to balance tradeoffs between size and speed. Illustrated in this graph is the final stage of a six-stage pipelined loop. This graph also helps C programmers understand how sequential C statements are parallelized and optimized.

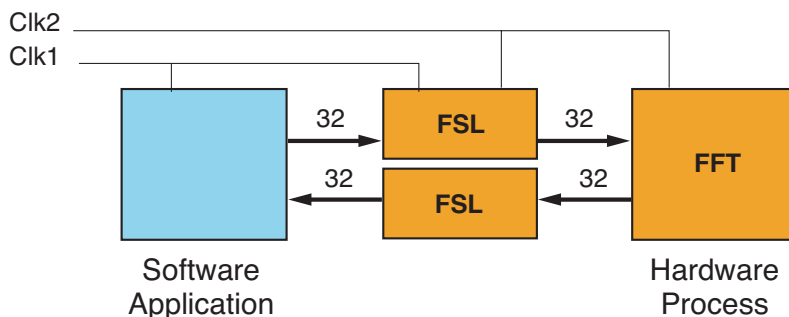


Figure 3 – The FFT includes a 32-bit stream input, a 32-bit stream output, and two clocks, allowing the FFT to be clocked at a different rate than the embedded processor.

The Impulse compiler generates appropriate FIFO buffers and Fast Simplex Link (FSL) interconnections for the target platform, thereby saving you from the low-level hardware design that would otherwise be needed.

logic, allowing us to achieve the best possible throughput. Our radix-4 FFT algorithm on 256 samples requires approximately 3,000 multiplications and 6,000 additions. Nonetheless, using the pipelining feature of Impulse C, we were able to generate hardware to compute the FFT in just 263 clock cycles.

We then integrated the resulting FFT hardware processing core into an embedded Linux (μ Clinux) application running on the Xilinx MicroBlaze™ soft-processor core. MicroBlaze μ Clinux is a free Linux-variant operating system ported at the University of Queensland and commercially supported by PetaLogix.

The software side of the application running under the control of the operating system interacts with the FFT through data streams to send and receive data, and to initialize the hardware process. The streams themselves are defined using abstract communication methods provided in the Impulse C libraries. These stream communication functions include functions for opening and closing data streams and reading and writing those streams. Other functions allow the size (width and depth) of the streams to be defined.

By using these functions on both the software and hardware sides of the application, it is easy to create applications in which hardware/software communication is abstracted through a software API. The Impulse compiler generates appropriate FIFO buffers and Fast Simplex Link (FSL) interconnections for the target platform, thereby saving you from the low-level hardware design that would otherwise be needed.

Embedded Linux Integration

The default Impulse C tool flow targets a standalone MicroBlaze software system. In some applications, however, a fully featured operating system like μ Clinux is required. Advantages of embedded Linux include a familiar development environment (appli-

cations may be prototyped on desktop Linux machines), a feature-rich set of networking and file storage capabilities, a tremendous array of existing software, and no per-unit distribution royalties.

The μ Clinux (pronounced “you-see-Linux”) operating system is a port of the open-source Linux version 2.4. The μ Clinux kernel is a compact operating system appropriate for a wide variety of 32-bit, non-memory management unit (MMU) processor cores. μ Clinux supports a huge range of microprocessor architectures, including the

Xilinx MicroBlaze processor, and is deployed in millions of consumer and industrial embedded systems worldwide.

Integrating an Impulse C hardware core into μ Clinux is straightforward; the Impulse tools include support for μ Clinux and can generate the required hardware/software interfaces automatically, as well as generate a makefile and associated software libraries to implement the streaming and other functions mentioned previously. Using the Xilinx FSL hardware interface, combined with a freely available generic FSL device

```
/* example 1 – simple use of ImpulseC-generated HW coprocessor and
 * Linux FSL driver
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define BUFSIZE 1024

void main(void)
{
    unsigned int buffer[BUFSIZE];

    /* Open the FSL device (Impulse HW coprocessor)*/
    int fd = open("/dev/fslfifo",O_RDWR);

    while(1)
    {
        /* Get incoming data – application dependent*/
        get_input_data(buffer);

        /* Send data to ImpulseC HW processor on FSL port */
        write(fd, buffer,BUFSIZE*sizeof(buffer[0]));

        /* Read the processed data back from the HW coprocessor */
        read(fd, buffer,BUFSIZE*sizeof(buffer[0]));

        /* Do something with the data – application dependent */
        send_output_data(buffer);
    }
}
```

Figure 4 – Simple communication between μ Clinux applications and ImpulseC hardware using the generic FSL FIFO device driver

driver in the MicroBlaze μ Clinux kernel, makes the process of connecting the software application to the Impulse C hardware accelerator relatively easy.

The generic FSL device driver maps the FSL ports onto regular Linux device nodes, named `/dev/fslfifo0` through to `fslfifo7`, with the numbers corresponding to the physical FSL channel ID.

The FIFO semantics of the FSL channels map naturally onto the standard Linux software FIFO model, and to the streaming programming model of Impulse C. An FSL port may be opened, read, or written to, just like a normal file. Here is a simple example that shows how easily a software application can interface to a hardware co-processing core through the FSL interconnect (Figure 4).

You can easily modify this basic structure to further exploit the parallelism available. One easy performance improvement is to overlap I/O and computation, using a double-buffering approach (Figure 5).

From these basic building blocks, you are ready to tune and optimize your application. For example, it becomes a simple matter to instantiate a second FFT core in the system, connect it to the MicroBlaze processor, and integrate it into an embedded Linux application.

An interesting benefit of the embedded Linux integration approach is that it allows developers to take advantage of all that Linux has to offer. For example, with the FFT core mapped onto FSL channel 0, we can use MicroBlaze Linux shell commands to drive and test the core:

```
$ cat input.dat > /dev/fslfifo0 &; cat /dev/fslfifo0 > output.dat;
```

Linux symbolic links permit us to alias the device names onto something more user-friendly:

```
$ ln -s /dev/fslfifo0 fft_core
```

```
$ cat input.dat > fft_core &; cat fft_core > output.dat;
```

Conclusion

Although our example demonstrates how you can accelerate a single embedded application using one FSL-attached accelerator, Xilinx Platform Studio tools also permit multiple MicroBlaze CPUs to be instantiated in the same system, on the same FPGA. By connecting these CPUs with FSL channels and employing the generic FSL device driver architecture, it becomes possible to create a small-scale, single-chip multiprocessor system with fast inter-processor communication. In such a system, each CPU may have one or more hardware acceleration modules (generated using Impulse C), providing a balanced and scalable multi-processor hybrid architecture. The result is, in essence, a single-chip, hardware-accelerated cluster computer.

To discover what reconfigurable cluster-on-chip technology combined with C-to-hardware compilation can do for your application, visit www.petalogix.com and www.impulsec.com.

```
/* example 2 – Overlapping communication and computation to exploit
 * parallelism
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define BUFSIZE 1024

void main(void)
{
    unsigned int buffer1[BUFSIZE],buffer2[BUFSIZE];
    unsigned int *buf1=buffer1;
    unsigned int *buf2=buffer2;
    unsigned int *tmp;

    /* Open the FSL device (Impulse HW coprocessor)*/
    int fd = open("/dev/fslfifo0",O_RDWR);

    /* Get incoming data – application dependent*/
    get_input_data(buf1);

    while(1)
    {
        /* Send data to ImpulseC HW processor on FSL port */
        write(fd, buf1,BUFSIZE*sizeof(buffer[0]));

        /* Read more data while HW coprocessor is working */
        get_input_data(buf2);

        /* Read the processed data back from the HW processor */
        read(fd, buf1,BUFSIZE*sizeof(buffer[0]));

        /* Do something with the data – application dependent */
        send_output_data(buf1);

        /* Swap buffers */
        tmp=buf1;
        buf1=buf2;
        buf2=tmp;
    }
}
```

Figure 5 – Overlapping communication and computation for greater system throughput