INTEGRATION OF FPGAS INTO SDR VIA MEMORY-MAPPED I/O

Matt Carrick (W@VT, Blacksburg, VA, USA, mcarrick@vt.edu); Shereef Sayed (W@VT, Blacksburg, VA, USA; ssayed@vt.edu); Dr. Carl Dietrich (W@VT, Blacksburg, VA, USA, cdietric@vt.edu); Dr. Jeff Reed (W@VT, Blacksburg, VA, USA, reedjh@vt.edu)

ABSTRACT

A primary appeal of Field Programmable Gate Arrays (FPGAs) is their high computational performance, but there has been a challenge to integrate FPGAs into the SDR hardware and software co-design process. The addition of FPGAs into the design process requires a logical representation of the FPGA as well as an interface to the General Purpose Processor (GPP). Current approaches for interfacing with FPGAs include the use of kernel-level drivers or standardized interfaces. This paper presents an example interface to FPGAs through the use of memory-mapped I/O. This interface is encapsulated as a logical software representation of the FPGA in order to enhance portability across platforms. We will target a Xilinx ML403, where the FPGA is collocated with a PowerPC core and leverage the use of memory-mapped I/O to interface with the FPGA and demonstrate the portable nature of our implementation.

1. INTRODUCTION

Using a General Purpose Processor (GPP) allows the developer to easily reconfigure the radio; however, the designer must trade flexibility versus performance. The limited processing power of GPPs leads to an inability to implement modern standards such as WiFi or WiMax. Additional processing power is needed to implement these standards; however, the solution must still fit into the software radio paradigm of being flexible and reprogrammable. Field Programmable Gate Arrays (FPGA) provide developers the bandwidth necessary to implement modern standards, while still being sufficiently flexible and reprogrammable to be incorporated into software radios.

Beyond simply integrating all of the hardware on the radio platform with simple logic, a software radio must have a software architecture used to provide command and control for all aspects of the system. Different software radio architectures exist and the targeted architecture within this document is the Software Communications Architecture (SCA). Adapters are used to interface to the Core Framework with devices that are not capable of running CORBA. This interface also acts as an abstractor, allowing processing to be completed on a hardware device. This Adapter is simply a software component which includes within it the interface to the hardware device. This allows a minimal amount of overhead to be added to the interface and the Core Framework can operate on the component as if the processing is being done on the GPP.

The goal of this document is to show that traditional methods of integrating hardware into a radio are still viable and provide an example implementation of how this is done with minimal overhead. The example implementation will also demonstrate that the FPGA interface is still portable without using CORBA and that by offloading processing to the FPGA, the data rate of the radio is limited by the bus speed and the overhead produced by the operating environment. A simple block diagram to demonstrate how the processing will be partitioned is given in Figure 1.

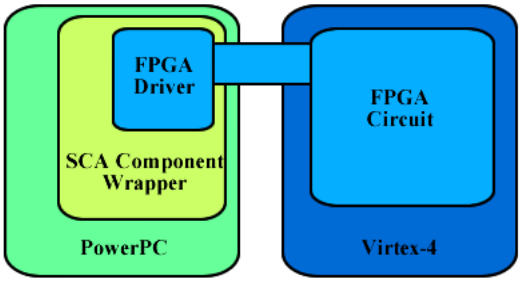

Figure 1: HW/SW Partitioning

2. FPGA INTERFACE

The targeted platform is a Xilinx Virtex-4 ML403 Embedded Platform [1]. The platform has a Virtex-4 FX12 FPGA which is clocked at 100 MHz. Within the die of the FPGA is a PowerPC405D5 core which is clocked at 300 MHz. The PowerPC is running a Linux kernel and is connected to the FPGA through the CoreConnect Bus using a Processor Local Bus (PLB) IPIF (Intellectual Property Interface) Xilinx Core at 100 MHz. The ML403 platform was selected due to the ease of integration of the FPGA and

the PowerPC, as well as the availability of driver support for peripherals. Peripherals on the platform use memory mapped I/O, which allows the FPGA to be easily accessed by simply writing to and reading from memory addresses. The hardware support for the Ethernet and Serial Port peripherals are provided in XPS, while software support is provided in the Linux kernel distributed by Xilinx.

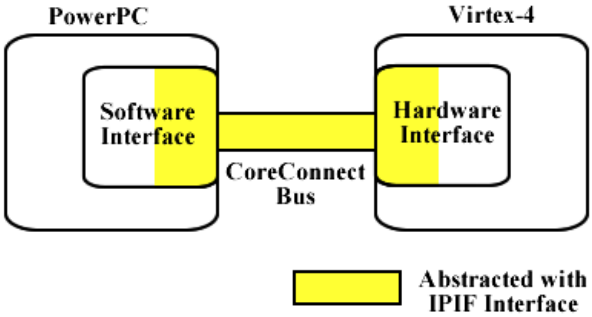

Figure 2: HW/SW Interface [2]

The SCA is dependent on devices that can run CORBA for Inter-process Communication (IPC). Multiple software implementations of CORBA exist, and requiring CORBA typically translates to running the software on a GPP. With the introduction of the SCA, CORBA has been extended to both DSPs and FPGAs through commercial vendors such as Objective Interface Systems (OIS) [3]. Although this method of communicating with a device through CORBA exists, as both a design choice and an interpretation of the specification, this method will not be used. If a device is not capable of running CORBA, the SCA allows the use of an "adapter." The specification states that "Adapters are resources or devices used to support the use of non-CORBA capable elements within the domain" [4]. The interface to the FPGA will be abstracted into a component, which will be running on the PowerPC and therefore have access to CORBA for communicating with other processes. The Core Framework is agnostic as to how a component is implemented, therefore mixing components running on the GPP and the FPGA with an adapter is allowed.

The PowerPC and the FPGA are connected over a CoreConnect bus, and interfaces to the bus from each device must be implemented. An interface from the PowerPC to the PLB may be written by the developer or the Xilinx IPIF interface may be used. The IPIF interface is a Xilinx core which "provides a bi-directional interface between a User IP core and the PLB 64-bit bus standard" [5]. Interactions with the PLB including toggling the correct bus lines during read and write cycles are abstracted with the use of this core. Interaction with the PLB is further abstracted through the use of Memory Mapped I/O. Memory Mapped I/O allows peripherals to be accessed very easily using just a simple read or write to memory command. During a read cycle the operating system accepts the command to read from memory, which must then be translated into a physical memory address by the Memory Management Unit (MMU). Once this address has been determined, it is sent to the IPIF interface which interacts with the PLB. The various software interfaces written to interface with the FPGA were derived from the driver from [6]. The interface first opens the `/dev/mem` device, and maps the requested memory into user space. Then memory can then be read from using a single line in C:

```
unsigned long value = *((unsigned long *) (BASE_ADDR + REG0_OFFSET));
```

This instruction has several operations embedded within it. The address to read from is calculated by adding the register offset to the base memory address. This address is then type cast as a pointer, so the memory can be accessed. Finally, the pointer is then de-referenced which starts the read cycle. The operating system determines that the software is requesting a memory read, the hardware decodes the address, retrieves the value from the FPGA and ultimately is returned to the variable.

Writing to the address is accomplished in a similar fashion:

```
*((unsigned long *) (BASE_ADDR + REG0_OFFSET)) = someValue;
```

The hardware interface to the CoreConnect bus can be fully implemented by a developer however in this work it was built using the Create or Import Peripheral Wizard in XPS. Settings such as the number of registers and which bus lines are available can be selected in the wizard. Once the settings are selected, the skeleton VHDL or Verilog code is generated.

The FPGA does not posses any ability to abstract the operation of accessing memory as the PowerPC does with the operating system. The developer is left to interface directly with the CoreConnect bus and toggle the necessary bus lines when appropriate. While this may add additional lines of code, it provides the developer more flexibility regarding how and when information is transferred over the bus.

While specific implementations of reading and writing to the bus may differ, the skeleton code generated by XPS provides the developer the ability to read and write from registers on the FPGA. The instructions from the bus are then decoded on the FPGA and the appropriate actions are taken. For example, if the line requesting a read cycle is toggled high, the bus must also specify which register is being read from. The developer must provide the necessary logic to check for the read cycle, decode the register address, and then transfer the information back over the bus.

3. FPGA INTEGRATION

Two waveforms were developed to demonstrate the integration of an FPGA into the OSSIE software. These waveforms are `ml403_ossie_demo` and `FIRFilterDemo`. These waveforms are dependent on the software devices `GPP` and `XilinxFPGA`, as well as the `default_ml403_node` node. These waveforms are dependent on the logical representation of the PowerPC processor and the Virtex-4 FPGA. Both of these devices must be started through a `Device Manager`, and this information will be stored in the node `default_ml403_node`.

The integration of an FPGA circuit into the OSSIE software requires two interfaces. The first interface is the FPGA controller component. This interface will contain the protocol for reading and writing to the FPGA as well as converting the data to the appropriate format. The second interface that is needed is the interface to the OSSIE software. This second interface is simply a software wrapper generated by the OSSIE tool suite which calls the FPGA controller component within it. The OSSIE software will be unaware and agnostic to the fact that the processing is done on the FPGA due to the software wrapper. The OSSIE component wrapper will interface with the OSSIE Core Framework, register with the naming service, and interact with other OSSIE software components in the domain. Generally this interaction will be limited to receiving and transmitting data between components.

The specific operation of both FPGA controllers will be different, although they follow a similar form. The controllers have three major functions; opening the interface to the FPGA, reading and writing to the FPGA, and closing the interface. Opening and closing the interface is standard among the three controllers as it requires mounting and un-mounting `/dev/mem`, respectively. The reading and writing operations are specific to the FPGA circuit that the controller is interacting with.

4. EXAMPLE APPLICATIONS

A very basic demonstration of the integration of an FPGA with the OSSIE software is the `ml403_ossie_demo` waveform. This waveform is very similar to the `ossie_demo` waveform, however it is targeted for the PowerPC processor and Virtex-4 FPGA. This waveform includes three components, `TxDemo`, `ChannelDemo`, and `RxDemo`. The first component, `TxDemo`, generates Quadrature Phase Shift Keying (QPSK) symbols, while the `ChannelDemo` component adds Gaussian noise, and finally `RxDemo` decodes the symbols and determines the Bit Error Rate (BER).

The basic nature of the `ml403_ossie_demo` waveform does not lend itself to a meaningful signal processing application, instead it provides a basic application to demonstrate the capabilities of the OSSIE software and in this case the FPGA integration.

The `TxDemo` component is a data source which generates 512 QPSK in-phase and quadrature symbols and transmits them using the `complexShort` port type. The data generation is done by reading out two arrays; one array for symbols to be modulated against the in-phase carrier and another for the quadrature carrier. The order of the symbols must be maintained as the `RxDemo` component is using the same order to compare the incoming symbols against and determine the BER. The FPGA implementation of the `TxDemo` component operates in the same manner as the software implementation, however the data generation is done on the FPGA. The OSSIE software wrapper passes two empty vectors to the FPGA driver which will ultimately be returned with the QPSK symbols. The FPGA driver then iterates through 64 read cycles to get the symbols from the FPGA.

The IPIF interface supports 32 bit data transfers, however 512 in-phase and 512 quadrature symbols need to be transferred. This multiple stage transfer is done by transmitting 8 in-phase symbols, 8 quadrature symbols and additional protocol information over the bus 64 times. When the FPGA controller component requests a read, a 6 bit counter on the FPGA is incremented by 1. The value of the counter is then used to access two Look Up Tables (LUT) containing the in-phase and quadrature symbols. These symbols from the LUTs and the counter value are then transmitted over the bus back to the FPGA controller. The FPGA controller then uses the counter value to load the symbols in the correct position in the two arrays, which are finally sent back to the OSSIE software component after all 64 transfers are completed. The OSSIE software component then converts the data to be used by the complexShort port type and sends the information to the next component.

The next waveform developed is an FIR filter. Embedded multipliers are common on FPGAs in the Xilinx Virtex-4, Virtex-5 and Virtex-6 series and can operate much faster than the multiplication implemented on a GPP. Although by themselves the multipliers will operate faster on the FPGA than the GPP, the overhead for interfacing with the FPGA must also be taken into account.

The FIR filter uses a very basic waveform, `FIRFilterDemo`, consisting of the `CarrierDataSource` component and the `FIR_filter` component. The `CarrierDataSource` component provides a carrier which the `FIR_filter` processes with a FIR filter on the FPGA.

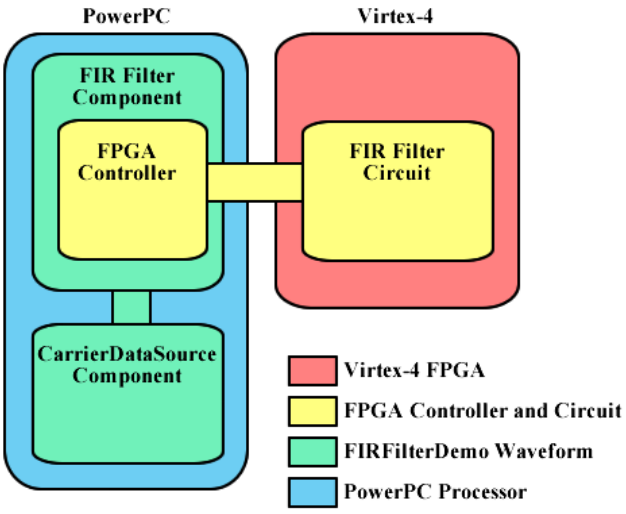

Figure 3: FIR Filter on the Xilinx ML403

Since the data being processed by the FPGA is stream-based, the protocol for interfacing with the FPGA is very simple. The OSSIE software wrapper receives data from a source component and the data is then iterated through. In each iteration, a sample of data from the OSSIE software wrapper is passed to the FPGA controller which in turn writes this value to the FPGA. When the FPGA receives the write request, the sample is then registered and filtered. After the sample is filtered, it is then stored in a First In First Out (FIFO) buffer of depth 512. The FPGA controller then polls the FPGA to determine if the filtered value can be read. The value is presented on the bus when both a read request is being made and the FIFO is not empty. One of these conditions is not true a code word is presented on the bus stating that the FPGA is not ready for a read cycle. However since both of these conditions are true, the sample is presented on the bus and received by the FPGA controller. Finally, the FPGA controller sends the filtered sample back to the OSSIE software wrapper which continues to iterate through all of the samples.

5. PROFILING AND RESULTS

The motivation for the integration of FPGAs into software radio is the ability to offload computationally complex signal processing algorithms from the GPP to the FPGA to improve the maximum data rates. Performance metrics were taken to determine the maximum data rate when interfacing with the FPGA, doing type conversions to a CORBA `ShortSequence`, and for the full waveform. These results were profiled for the `FIRFilterDemo` waveform.

Timing measurements were taken using the `gettimeofday()` and `getrusage()` [7] functions. The `gettimeofday()` function is used to determine the overall time difference for a process to run and may be referred to as the "wall clock" time. This measurement will be used to determine the maximum data rate possible for a waveform. The `getrusage()` function is used to determine the amount of time the processor has dedicated to running a specific user process and the amount of time used for running other system level processes. This measurement will be used to show the amount of processor usage of a waveform. Additionally, this metric shows how much impact the rest of the operating environment has on the rates the process is running at. As the usage percentages for the process increases towards 100%, the less impact the operating system or Core Framework has on the rate of the process.

To obtain the data rates, both timing functions are called to get a start time and the process is then run in a `for` loop of limited length. After the `for` loop completes, the two timing functions are then called again to obtain the end times. The wall clock time is calculated to determine the maximum date rate, and then the user time and system time are calculated to determine the processor utilization.

The results of the FIR filter interfacing with the FPGA show that simply reading and writing to the FPGA can be done at a rate of 10.134 MHz. The drawback to this value is the amount of processor utilization. The process uses 98.91% of the processor when executing these cycles, and only allows the processor 0.30% of the time to run system processes. Since the processor usage is very large, the effect of the operating system and Core Framework on the process is negligible and the values are near the maximum values possible.

When performing the same benchmark, but adding the process of converting the data of the FIR filter to a sequence of CORBA short type, the results show that the operating frequency has been reduced from 10.134 MHz to 8.456 MHz due to the data conversion. The processor utilization is still very high at 99.16% and the system utilization is low at 0.19%. Similar to the previous result, the processor utilization is very large, therefore the results are near the maximum values and the operating system and Core Framework are providing very little overhead.

The third measurement is to determine the maximum rate at which the entire FIR filter waveform can be run. The measurement is taken within the `ProcessData()` function of the `FIR_filter` component. The measurement includes all of the processing overhead of the component interacting with the OSSIE Core Framework as well as the `CarrierDataSource` component generating and transmitting its data. The results from the `FIRFilterDemo` waveform show that the maximum rate possible is 116.849 kHz. This value is substantially less than

the previous operating frequency of 8.456 MHz. This is due to the processor allocating clock cycles to system processes, components within the OSSIE Core Framework and its dependencies. This can be seen in the utilization, which has also dropped from 99.16% to 25.26% while the system utilization has increased from 0.19% to 30.59%.

FIR_filter interfacing with FPGA directly		
Measurement	***Time (sec)***	***Frequency (MHz)***
Wall Clock	0.987	10.134
User Time	0.976	***Processor***
System Time	0.003	98.91%
Iterations	1.00E+07	***System Utilization***
Samples	1	0.30%
FIR_filter component interfacing with FPGA and conversion to CORBA		
Measurement	***Time (sec)***	***Frequency (MHz)***
Wall Clock	36.3287	8.456
User Time	36.024	***Processor***
System Time	0.068	99.16%
Iterations	6.00E+05	***System Utilization***
Samples	512	0.19%
FIRFilterDemo Waveform		
Measurement	***Time (sec)***	***Frequency (kHz)***
Wall Clock	13.1452	116.849
User Time	3.321	***Processor***
System Time	4.021	25.26%
Iterations	3.00E+03	***System Utilization***
Samples	512	30.59%

Table 1: Profiling Results of FIR filter

The introduction of the Core Framework into the profiling results decreases the operating rates for the `FIRFilterDemo` waveform by a factor of 87. To determine what kind of overhead the Core Framework was providing, it was removed and additional measurements were taken. The results for the `FIRFilterDemo` waveform after removing the Core Framework show similar performance as the initial component benchmarks with 10.407 MHz. The processor utilization is similar, as well, with 99.83%.

These profiling results are very close to the original `FIRFilterDemo` waveform. The significant drop in operating frequency in the `FIRFilterDemo` waveform from 10.143 MHz to 116.849 kHz comes from the transmission of data through CORBA instead of simply reading the values out of the memory locally. Once the Core Framework is removed, the value is very close to the original profiling result for both waveforms where the Core Framework is not interacted with. Referring back to the initial benchmarks for the FIR filter on the PowerPC, the maximum data rate was 2.43 MHz. This value includes no overhead from an operating system, a software radio architecture or additional signal processing components. This benchmark is best compared against the frequency measurements of interfacing with the FPGA in the `FIR_filter` component which operates at 10.134 MHz. The FPGA interface operates more than 4 times faster than performing the filtering on the PowerPC, including the overhead from the operating system and the OSSIE Core Framework.

6. CONCLUSIONS

The integration of FPGAs into the SCA has been done with the goal of minimizing overhead. Concepts that work well for software or processing at the enterprise level do not always translate well to embedded systems and hardware; therefore they have been left out of the design. Two examples of this are CORBA-on-a-chip and standard interfaces for FPGA circuits.

Explicit support for FPGAs is also not required for FPGAs within the SCA as it leads to the development of standard interfaces and HALs which will introduce unnecessary overhead into the radio platform. The 2.2.2 revision of the SCA provides support for any hardware device that a designer might choose through the use of an Adapter. By using an Adapter, the developer is able to integrate hardware into the radio platform as they see fit, which will allow overhead to be minimized and performance maximized.

Two separate examples have been provided showing how FPGAs can be integrated into the SCA. The first example is the waveform `ml403_ossie_demo`, which is very similar to the `ossie_demo` waveform with the exception that the symbols are generated on the FPGA. The second example, `FIRFilterDemo` is an example which demonstrates how signal processing can be offloaded to the FPGA and how to interface with the FPGA using memory mapped I/O.

A kernel level driver would be a more efficient method of transferring information between the FPGA and the processor. When the FPGA has calculated a sample and is ready to transmit over the bus, an interrupt is set and the processor will then read from the bus. This allows the processor to allocate clock cycles in the interim between samples for other processing tasks or simply to idle and reduce power consumption.

If the data rate is large enough, a kernel driver may be insufficient and Direct Memory Access (DMA) should be used. Direct Memory Access would allow the FPGA to write directly to main memory, bypassing the processor for this task. This adds to the system's complexity but ultimately will make the system more efficient. Ideally the processor should interact with the devices over the bus as little as possible as it uses cycles that could be better used in another process or by idling.

One benefit of using a simple memory mapped I/O driver is the ability to port it to other platforms. Given shared memory between the processor and the FPGA and memory mapped I/O, controller components for accessing devices can be written which are agnostic to the target platform. Having memory mapped I/O allows driver components to simply make read and write calls to memory, which are then translated to accessing a peripheral. By simply passing in a base address and the offsets for registers, the same driver can be cross-compiled for another processor very easily.

The problem with using a simple memory mapped I/O driver which accesses a peripheral by polling is the large utilization. Ideally the processor would access the peripheral at the minimum rate required, allowing the other clock cycles to be devoted to other processes. This utilization could be mitigated by the use of a Real Time Operating System (RTOS). Instead of polling to determine if a device is ready, the RTOS would have the write and read cycles internally scheduled. This would reduce the processor utilization while still retaining the portability of the driver.

All source code for the OSSIE Core Framework, FPGA interfaces and example waveforms can be found at [8], and further discussion on the integration of FPGAs into the Core Framework can be found at [2].

7. REFERENCES

[1] Xilinx, "Virtex-4 ML403 Embedded Platform," [Online document] 1994-2008, [2009 March 25], Available at HTTP: http://www.xilinx.com/products/devkits/HW-V4-ML403-UNI-G.htm

[2] M. Carrick, "Logical Representation of FPGAs and FPGA Circuits within the SCA," M.S. Thesis, Virginia Polytechnic Institute and State University, 2009. http://scholar.lib.vt.edu/theses/available/etd-07012009-203400/

[3] Objective Interface Systems, "ORBexpress: An Overview," [Online document] 1996-2009 [2009 May 06], Available at HTTP: http://www.ois.com/Products/Communications-Middleware.html

[4] JTRS Standards, "Software Communications Architecture Specification," [Online document] 2006 May, [2009 March 20], Available at HTTP: http://sca.jpeojtrs.mil/_downloads.asp?folder=SCAv2_2_2&file=SCA_version_2_2_2.pdf

[5] Xilinx, "PLB IPIF (DO-EDK)," [Online document] 1994-2008 [2009 April 6], Available at HTTP: http://www.xilinx.com/products/ipcenter/plb_ipif.htm

[6] Xilinx Open Source Linux Wiki, "OSL user mode pseudo driver," [Online document] Oct. 23, 2008 [2009 March 25], Available at HTTP: http://xilinx.wikidot.com/osl-user-mode-pseudo-driver

[7] The Open Group Base Specifications Issue 6, "getrusage - get information about resource utilization," [Online document] [2009 May 06], Available at HTTP: http://www.opengroup.org/onlinepubs/000095399/functions/getrusage.html

[8] OSSIE, "SCA-Based Open Source Software Defined Radio," [Online Document], [2009 October 18], Available at HTTP: http://ossie.wireless.vt.edu/