In Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, San Jose, California, July 2002.

A Component Architecture for FPGA-based, DSP System Design

Gary Spivey
ECE Department
University of Maryland, USA and
Rincon Research Corp.
Tucson, AZ, USA
spivey@rincon.com

Shuvra S. Bhattacharyya*
ECE Department and
UMIACS
University of Maryland, USA
ssb@eng.umd.edu

Kazuo Nakajima†
ECE Department
University of Maryland, USA, and
Graduate School of Information Science,
Nara Institute of Science and Technology,
Ikoma, Nara, Japan
kazuo@is.aist-nara.ac.jp

Abstract

Introducing FPGA components into DSP system implementations creates an assortment of challenges across system architecture and logic design. Recognizing that some of the greatest challenges occur in the integration of the various components, we have developed a component architecture and an associated set of software tools, collectively called the Logic Foundry. Using the Logic Foundry, an FPGA-based DSP system can be easily constructed from pre-built components and implemented on a variety of back-end FPGA platforms. The resulting implementation can then be encapsulated and integrated into a variety of front-end software application environments. This paper develops the component architecture and integration capabilities of the Logic Foundry, and examines a number of application case studies that we have experimented with using the Logic Foundry.

1. Introduction

Introducing FPGA components into DSP system implementations creates an assortment of challenges across system architecture and logic design. Where system architects may be available, skilled logic designers are a scarce resource. There is a growing need for tools to allow system architects to be able to implement FPGA-based platforms with limited input from logic designers. Unfortunately, getting designs translated from software algorithms to hardware implementations has proven to be difficult.

Earlier efforts such as the GRAPE-II [1] system tended to focus on creating a heterogeneous multiprocessor rather than an FPGA-based subsystem — typically enforcing a static dataflow model. Current efforts like MATCH [2] have attempted to compile high-level languages such as MatLab directly into FPGA implementations. Certain tools such as C-Level Design [3] have attempted to convert “C” software into a hardware description language (HDL) format such as Verilog or VHDL that can be processed by traditional FPGA design flows. Other tools use derived languages based on C such as Handel-C [4], C++ extensions such as

* S. S. Bhattacharyya was supported in part by the National Science Foundation (Grant #9734275)
† Previously affiliated with NTT Communication Science Laboratories, Kyoto, Japan

SystemC [5], or Java classes such as JHDL [6]. These tools give designers the ability to more accurately model the parallelism offered of the underlying hardware elements. While these approaches attempt to raise the abstraction level for design entry, many experienced logic designers argue that these higher levels of abstraction do not address the underlying complexities required for efficient hardware implementations.

Another approach has been to use “block-based design” [7] where system designers can behaviorally model at the system level, and then partition and map design components onto specific hardware blocks which are then designed to meet timing, power, and area constraints. An example of this technique is the Xilinx System Generator for the MathWorks Simulink Interface [8]. Using this tool, a system designer can “develop high-performance DSP systems for Xilinx FPGA’s. Designers can design and simulate a system using MatLab, Simulink, and a Xilinx library of bit/cycle-true models. The tool will then automatically generate synthesizable Hardware Description Language (HDL) code mapped to Xilinx pre-optimized algorithms” [8]. However, this block-based approach still requires that the designer be intimately involved with the timing, and control aspects of cores in addition to being able to execute the back-end processes of the FPGA design flow. Furthermore, the only blocks available to the designer are the standard library of Xilinx IP Cores. Other “black-box” cores can be developed by a logic designer using standard HDL techniques, but these cannot currently be modeled in the same environment. Annapolis MicroSystems has developed a tool entitled “CoreFire” that uses pre-built blocks to obviate the need for the back-end processes of the FPGA design flow, but is limited in application to Annapolis MicroSystems hardware [9]. In both of the above cases, the system designer must still be intimate with the underlying hardware in order to effectively integrate the hardware into a given software environment.

Another related approach is the use of high-level, embedded system design tools, such as Ptolemy [10], and Polis [11]. These tools emphasize overall system simulation and software synthesis rather than the details required in creating and integrating FPGA-based hardware into an existing system. An effort funded by the DARPA Adaptive Computing Systems (ACS) was performed by Sanders (now BAE Systems) [12] that was successful in transforming an SDF graph into a reasonable FPGA implementation. However, this effort was strictly limited to the implementation of a signal processing datapath with no provisions for runtime control of processing elements. Another ACS effort, Champion [13], was implemented using Khoros's Cantata [14] as a development and simulation environment. This effort was also limited to datapaths without runtime control considerations. While datapath generation is easily scalable, control synthesis is not. Increased amounts of control will rapidly degrade system timing, often to the point where the design becomes unusable.

In the brief survey above of relevant work, we have observed that while some of these efforts have focused on the design of FPGA-based DSP processing systems, there has been less work in the area of implementing and integrating these designs into existing software application environments. Typically a specific hardware platform has been targeted and integration into this platform is left as a task for the user. Software front-ends are generally designed on an application-by-application basis and for specific software environments. Because requirements are often rapidly changing and increasing in complexity, it is necessary for any solution to be rapidly designed and modified, portable to the latest, most powerful processing platform, and easily integrated into a variety of front-end software application environments. In other words, in addition to the challenge of creating an FPGA-based DSP design, there is another great challenge in implementing that design and integrating it into a working software application environment.

To help address this challenge we have created the “Logic Foundry”. The Logic Foundry uses a “platform-based” design approach. Platform-based design starts at the system level and “achieves its high productivity through extensive, planned design reuse … productivity is

increased by using predictable, pre-verified blocks that have standardized interfaces" [7]. To facilitate the rapid implementation and deployment of these platform-based designs, we have created a component-based architecture that allows for run-time control of processing elements. Using this architecture, an FPGA-based DSP system can be easily constructed from pre-built components and implemented on a variety of back-end FPGA platforms. The resulting implementation can then be automatically encapsulated and integrated into a variety of front-end software application environments.

2. Component Architecture

A Logic Foundry component specifies attributes and portals. Essentially, an attribute is any publicly accessible part of the component, providing state inspectors and behavioral controls. Portals are the elements on a component that provide interconnection to the outside and are made up of user-defined pins.

2.1 The Attribute Interface

Other attempts at component-based FPGA-based development systems have assumed that the FPGA implementation is simply a static data modifying piece in a processing chain [12],[13]. Logic Foundry components are designed assuming that they will require run-time control and thus are specified as having a single attribute interface through which all data-asynchronous control information flows. Each FPGA in a system has exactly one controlling attribute interface and every component has exactly one attribute interface. All data-asynchronous communications to the components are done through this interface.

An attribute interface consists of: an attribute bus, a strobe signal from the controlling attribute interface, and an event signal from each component. We have implemented the attribute bus with a tri-state bus that traverses the entire chip and connects each component's attribute interface to the controlling attribute interface. Because attribute accesses are relatively infrequent and asynchronous, the attribute bus uses a multi-cycle path to eliminate timing concerns and minimize routing resources.

Each component in a system has a unique address in the system. The controlling attribute interface decodes this address and enables the component via a unique strobe line from the controlling attribute interface to the addressed component. These strobe lines are distributed via delay chains and are also used by the components for attribute bus synchronization. Using delay chains costs very little in an FPGA as there are typically a large number of unused registers throughout a design. Data and control are multiplexed on the bus and handled by state machines in each component which provide address, control, and data buses inside each component.

Each component also has an individual event signal that is passed back to the controlling attribute interface. With the strobe and the event lines, communication can be initiated by each end of the system. This architecture elegantly handles data-asynchronous communication requirements for our FPGA-based processing systems.

2.2 Data Portals

Components may have any number of input/output portals, and in a DSP system, these are generally characterized by a streaming data portal. Each streaming portal is implemented using a FIFO with ready and valid signals. Using FIFO's on the inputs and outputs of a component isolates both the input and the output of each cell from timing concerns as all signals going to and

coming from an interface are registered. This allows components to be assembled in a larger system without fear of timing restrictions arising from component loading.

By using FIFO's to monitor data flow, flow control is automatically propagated throughout the system. It is the responsibility of every component to ensure that this behavior is followed inside the component. When an interface cannot accept data, the component is responsible for stopping. If the component cannot stop, then it is up to the component to handle any dropped data. In our DSP environment, each data transfer represents a sample. By using flow control on each stream, there is no need to insert delay elements for balancing stream paths — synchronization is self-timed [15].

FIFO's are extremely easy to implement in modern FPGA's by using the Lookup Table (LUT) as a small RAM component. So, rather than providing a flip-flop for each bit as a registration between components, a single LUT can be used and (in the case of the Xilinx Virtex part), a 16 deep FIFO is created. In the Virtex parts, each FIFO controller requires but 4 configurable logic blocks (CLB’s). In the larger FPGA's that we are targeting, this usage of resources is barely noticeable.

3. Platform Integration

When designing on a particular platform, certain aspects of the component such as memory and control interfaces are often built into the design. This poses a difficulty in altering the design, even on the same platform. Changing a data source from an external source to Direct Memory Access (DMA) from the PCI bus could amount to a considerable design change as memory resources and data availability are considerably altered. This problem is exacerbated when completely changing platforms. As considerably better platforms are always being developed, it is necessary to be able to rapidly port to these platforms.

Some work has recently been undertaken in this arena as a joint venture between Wind River with their Board Support Package (BSP) and Celoxica’s Platform Abstraction Layer (PAL) [16]. A similar methodology was undertaken by JHDL [6] with its HWSystem Class. These efforts attempt to abstract the I/O interfaces between a processing platform and its host software environment, allowing an application that is developed on one platform to be migrated to another platform. However, the issues of platform-specific I/O to destinations other than the host software environment and on-board memory interfaces are not specifically addressed.

To combat this problem, the Logic Foundry employs an abstract portal for all design level interfaces. A Logic Foundry design consists of Logic Foundry components with abstract portals and is thus platform independent. Abstract portals are connected to the component portals when building a design. These abstract portals can then be mapped to a specific platform portal in a platform implementation. This form of interface abstraction is common in the design of reusable software; our contribution here is to develop its capabilities in the context of FPGA implementation and DSP hardware/software integration.

There are various portal types for differing needs. While new portal types can easily be developed to suit any given need, each abstract portal type requires a corresponding implementation portal for every platform. For this reason, we attempt to reuse existing portals whenever possible. We currently support three portal types: the Streaming Portal, the Memory Portal, and the Block Portal.

3.1 The Streaming Portal

A streaming portal is used whenever an application expects to stream data continuously. Depending on the implementation, this may or may not be the case (compare an A/D converter direct input to a PCI bus input that is buffered in memory via a DMA), but the design will be able to handle a streaming input with flow control. A streaming input portal consists of a data output, a data valid output, and a data ready input. Streaming portals connect directly to the streaming portals of a component.

Streaming portals may be implemented in many different ways — among these, a direct DMA input to the design, a direct hardware input, a gigabit Ethernet input, or a PMC bus interface. At the design level, all of these interface types can be abstracted as a streaming portal.

3.2 The Memory Portal

In the case of the off-chip dedicated memory, it may be desirable to pipeline memory accesses so that data can be rapidly streamed with a little latency. In the case of an off-chip arbitered memory, the memory portal must follow a transaction model, holding its memory access request until an acknowledgement is given. These two conflicting models must be merged into a single abstract memory portal. We do this by changing the read enable and write enable lines to read request and write request lines, respectively, and adding control pins for an access acknowledgement. By using these control signals for every external memory portal, the implementation will be able to map the abstract memory portals to available memory resources, using arbitered or dedicated memories wherever appropriate.

For many FPGA applications, we allow the assumption that the design has access to some amount of dedicated local memory (e.g. Block RAMS in a Xilinx Virtex Part). The Logic Foundry integrates such local memories as sub-nodes of a design rather than memory portals as the performance and control gains are too significant to be ignored. This does not greatly affect portability as successive generations of FPGA's tend to have more local memory rather than less. Additionally, drastically limiting the amount of memory available to a design would likely require algorithmic changes that would render the design unportable anyway.

3.3 The Block Portal

A block portal is similar to the memory portal and provides the same memory interface to access a block of data. It differs from the memory portal in that the block portal also provides transfer initiation control signals that allow an entity on the other side of the portal to transfer in/out the block. The block portal differs from the streaming portal in the location of the transfer initiation control. In the streaming portal, all transfers are initiated outside of the design block and the design block responds in a continuous manner. In the block portal, transfer initiation and block size are dictated by the block portal.

4. Software Integration

Like the hardware portability challenges, software portability can be challenging as unique driver calls and system access methodologies become embedded deeply in the software application program. This can require an application program to be substantially rewritten for a new FPGA platform. Furthermore, it is desirable to be able to make use of the same FPGA acceleration platform from different software environments such as Python, straight C code, MATLAB, or Midas 2k [17]. For example, the same application could be used in a fielded Midas

2k application as a researcher would access in a MATLAB simulation. Porting the application amongst the various software environments can be a difficult endeavor. In order to accommodate a wide variety of software front-ends, the Logic Foundry isolates front-end software applications environments and back-end processing environments through a standardized API. While other tools such as Handel-C and JHDL provide and API that allows software to abstractly interact with the I/O interfaces, the application must still be aware of internal hardware details. Our API, known as the DynamO API, provides dynamic object creation for the software front-end that completely encapsulates both I/O details and component control parameters such register addresses and control protocols. Using the DynamO object and API, an application programmer interacts solely with the conceptual objects provided by the logic designer.

4.1 The Dynamic Object (DynamO)

The DynamO object consists of a top level system component. This is a container for the entire back-end system. DynamO components can contain portals, attributes, and other components. In addition to these objects, methods and parameters are provided that allow the DynamO API to uniquely interact with the given object. Consider a digital downconverter [18] with a tuner, a filter, and a decimator (TFD). This component would contain an input portal, an output portal, and three components, tuner, filter, and decimator. Each of these components would themselves contain an attribute, *frequency*, *taps*, and *amount*, respectively (see Figure 1).

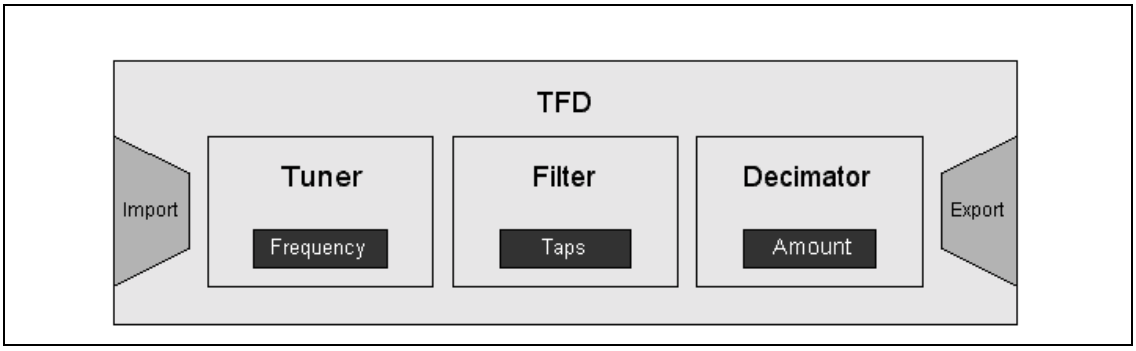

Figure 1: The DynamO Object

Along with the objects, the DyanmO Starfire back-end would attach methods for attribute sets and gets, and portal reads and writes. Embedded within each object is the information required by the back-end to uniquely identify itself. For example, while the frequency attribute of the tuner component, the taps attribute of the filter component, and the amount attribute of the decimator component would all use the same set/get methods for attributes, the component and attribute addresses embedded within them would be different.

Using the DynamO methodology, any back-end reconfigurable system can dynamically be built by a back-end based on the current configuration of the hardware. While the Logic Foundry uses a consistent attribute interface for all components and thus has but one interface method, a DynamO back-end could be constructed with different types of attribute access and multiple methods. By attaching these different methods to the required attributes on object build, the same level of software application independence can be achieved.

4.2 The DynamO API

The DynamO API represents the contract that DynamO back-ends and front-ends need to follow. The DynamO API consists of calls to allocate a system, set and get attributes, and write and read portals. These calls are implemented by the back-end library as the functionality is unique to each back-end platform (see Figure 2).

The API 'System' call requires a system specification file as an argument. The very beginning of this file points to a back-end implementation and a library to parse the rest of the specification file. In this manner, different back-ends can, if desired, have their own specifications unique to a given platform. By making the parsing of a specification file the responsibility of the back-end, there is no limitation on future back-end implementations. The result of the system call is an object representing the system being allocated (typically an FPGA board).

Each attribute in the system is writable by the back-end, front-end, or even both. This can be specified in the component specification file. The back-end is responsible for providing a method for attribute sets/gets. If a user is using the complete Logic Foundry implementation, then software wrappers around the board drivers exist that use the FPGA attribute portal to write the component attributes.

Portals are designed to have simple read/write interfaces. The DynamO API uses a packet structure to communicate with portals. This allows portals to differentiate between control and data and allows data-synchronous control to be passed into the portal rather than asynchronously through the attribute interface. The underlying FPGA hardware must be configured to handle these packets as well.

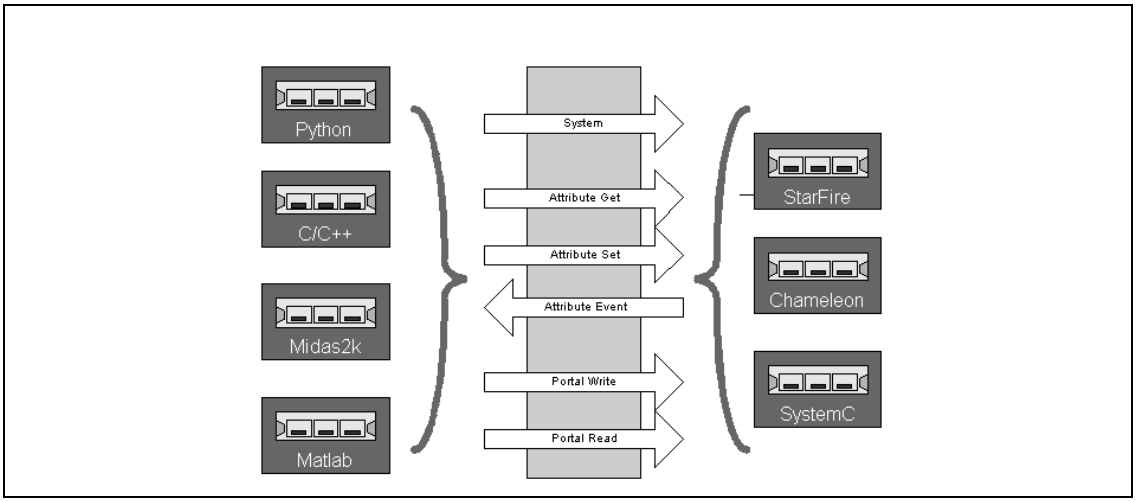

Figure 2: The DynamO API

4.3 DynamO back-ends

The DynamO back-end connects a platform to the DynamO API. When the DynamO is allocated, the back-end provides a library method to parse the specification file, and returns a hierarchical DynamO object that contains all of the information for the requested system. In this manner, the application environment is given an object with methods that represent the architecture of the system that is to be interacted with. No understanding of the implementation details of the underlying hardware is required.

While we hope that others find the Logic Foundry easy to use, it is important to note that the DynamO specification file does not require anything from the Logic Foundry. A designer could build a completely unique implementation, and then specify the underlying objects and methods for accessing them in a specification file.

Additionally, a software emulator could be constructed as a back-end. Future plans call for the inclusion of a DynamO back-end wrapper for a software emulator written in SystemC. System designers can do the first stages of algorithm definition in a C-based environment that can be more readily ported to an FPGA by a logic designer. By using a software emulator as a DynamO back-end, the entire front-end application can be developed and run before the FPGA-based application is completed. When the FPGA is complete, a new specification file for that back-end is used and the application requires no change.

4.4 DynamO Front-Ends

The DynamO front-end is responsible for taking the DynamO object returned by the system method and transforming it into an object that the software environment can understand and access. For instance, using a Python front-end, the DynamO object is recreated in Python objects, with its methods mapped to the supplied DynamO object methods. Figure 3 demonstrates how a Python application script would interact with the DynamO API and the DynamO object in the TFD mentioned in Section 4.1. Note that there is absolutely no evidence of implementation-specific details such as register addresses or communication protocols.

```
# load the library in Python
import dynamo

# open up a dynamo object
tfd = dynamo.system("tfd.spec")

# get an attribute
tune_freq = tfd.tuner.frequency

# set an attribute
tfd.decimator.amount = 10

# set an attribute with an array
taps = dynamo.array('d', 2)
taps[0] = 123
taps[1] = 456
tfd.filter.taps = taps
```

```
# Create a dynamo packet
p = dynamo.DataPacket('d' 1000)

# initialize p
for i in xrange(1000):
  p.data[i] = i*2

# write data to the import portal
tfd.import.write(p)

# read data from the export portal
p = tfd.export.read()
```

Figure 3: Python DynamO Example

5. Design Case Studies

We have developed the Logic Foundry including all of the major building blocks described — attribute interfaces, component abstractions and interface portals, the get/set and data write/read portions of the DynamO API, DynamO back-ends for an Annapolis MicroSystems Starfire board [19], and Dynamo front-ends for C++, Python, and Midas 2k. To test the effectiveness of the Logic Foundry, three systems have been developed, a series incrementer, the TFD, and a Turbo Decoder.

5.1 Incrementer Design

The incrementer component consists of a streaming input portal, a streaming output portal, and an amount attribute that is added to the input before being passed to the output. We experimented with the incrementer component using an Annapolis MicroSystems Starfire board. This platform consists of four memory ports attached to an FPGA. Annapolis MicroSystems provides a shell for the FPGA, DMA bridges to transfer data from the PCI bus to the memory, and software driver calls to perform the DMA's. To create the streaming input and output portals, we modified the DMA bridges to add control for streaming data into and out of memory. Additionally, a DynamO library was created that provided portal write and read methods using the Annapolis DMA Driver calls wrapped with the extra control to manage the modifications to the DMA bridges.

To control the Starfire card, Annapolis MicroSystems supplies driver calls to do addressable I/O via the PCI bus. However, the control is tightly timed and the Annapolis MicroSystems architecture implementing our portal functionality requires 7 control elements at the top level. When the number of elements attached to the control bus begins to exceed 10 or so elements, achieving the required timing of 66 MHz on a Xilinx XCV1000-4 can be difficult. For the Logic Foundry, we have built an attribute interface for all component control in the Starfire system and created DynamO interfaces to set/get attributes via this interface. The Starfire control bus is thus required to connect only to the DMA bridges, the attribute interface, and any top level control registers required for operation. These connections remain constant with the addition of new components.

To test the scalability of the Logic Foundry architecture, we created incrementer designs consisting of 1, 10, and 50 incrementer components connected together in series. In each case, system timing remained the same as the synthesis and layout tools were able to achieve the required 66 MHz control timing for the Starfire control bus, while the attribute interface scaled using the multi-cycle attribute bus (see Table 1). It was initially our intention to do a design consisting of 100 serial incrementers, however, we reached a limit for the XCV1000 parts that only allows a tri-state net to drive 98 locations. This limits an XCV1000 part to 98 components which is acceptable for our typical designs.

5.2 TFD Design

The TFD design was created to test the component reuse aspects of the Logic Foundry architecture along with the Logic Foundry's automated design flow. By creating a tuner, filter, and decimator component in the Logic Foundry, we were able to use the Logic Foundry software to automatically implement the TFD design and corresponding DynamO object. In order to test the ease of component reuse in the Logic Foundry, we opted to create a filter/tune/decimate (FTD) system out of the TFD system components by rearranging the top level connection specifications. In both cases, control timing was achieved and system timing limited by the speed of the tuner component (see Table 1).

5.3 The Turbo Decoder Design

The Turbo Decoder was a large design (several thousand lines of VHDL code) constructed with a view to fitting into the Logic Foundry attribute/portal design structure. This design created unique challenges — firstly, the streaming portal design would not work as the Turbo Decoder worked on blocks of data and had to individually address these blocks of data. For this reason we created the block portal interface described in Section 3.3.

The Turbo Decoder design required seven attributes and these were easily included via the attribute interface model. Implementing the block portals was more difficult as the completed Turbo Decoder design required eight unique block portals, five of which requiring simultaneous access. As the Starfire board had but four memories, this was a problem. However, as some of the portals did not require independent addressing, we were able to merge them into a single memory and achieve an implementation that required four independently addressable memories.

5.4 Summary of Designs

Table 1 shows results for each of the test designs implemented for the XCV1000-4 FPGA on the Annapolis MicroSystems Starfire board. Because control on this system is achieved via a 66 MHz PCI bus, the control clocks were all constrained to achieve this timing. In the case of the incrementer designs, the system clock performance was limited by the portal implementations. The other designs (TFD, FTD, TurboDecoder) were limited by issues internal to their design components. Further development will be done to optimize the portal implementations for this architecture. The differences within design groups (incrementers and downconverters) are attributable to variances in the Xilinx software. The pseudo-random nature of the algorithms often results in variances. By doing a more extensive place-and-route operation, we would likely see these numbers converge.

	Control Clk	System Clk	LUT's	Flip-flops	BlockRams
1 Incrementer	68.648	62.278	1328	1809	5
10 Incrementers	68.078	65.557	2007	2244	5
50 Incrementers	66.885	70.299	4959	4076	5
TFD	68.018	35.661	2873	2238	6
FTD	67.604	35.177	2873	2222	6
Turbo Decoder	67.290	39.787	17031	5600	27

Table 1: Summary of Designs

6. Conclusion

We have shown how the Logic Foundry approach allows for the rapid prototyping and deployment of FPGA-based systems. Using design portals for interface abstractions, designs can be created in a platform independent manner and easily ported from one FPGA platform to another where implementation portals exist. By using the DynamO software construction, applications can be built that have no dependence on the underlying FPGA platform and can easily be ported from platform to platform. Inserting a platform into a different software environment can also be done with relative ease.

Our future work will focus on the complete implementation of data-synchronous control packets, component event control, and the control write/read portions of the DynamO API. We will also be integrating the Chameleon board from Catalina Research Incorporated as a demonstration of platform migration. We have implemented the Logic Foundry at Rincon Research and the tool is being used extensively in the development of high performance FPGA

implementations of DSP applications, including turbo coding, digital downconversion, and despreading applications.

7. References

[1] R. Lauwereins, M. Engels, M. Adé and J. Peperstraete, “Grape-II: A system-level prototyping environment for DSP applications”, *IEEE Computer*, vol. 28, no. 2, pp. 35-43, February, 1995.

[2] P. Banerjee et al, “MATCH: A MATLAB Compiler for Configurable Computing Systems,” Technical Report, Center for Parallel and Distributed Computing, Northwestern University, Aug. 1999, CPDC-TR-9908-013.

[3] http://www.synopsys.com/C-level.html

[4] OXFORD Hardware Compilation Group, The Handel language, Technical Report, Oxford University 1997.

[5] J. Gerlach and W. Rosenstiel, “System Level Design Using the SystemC Modeling Platform,” http://www.systemc.org/papers/sda-2000.pdf.

[6] P. Bellows and B. Hutchings. “JHDL — an HDL for Reconfigurable Systems,” *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, pp. 175-184, April 1998.

[7] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly and L. Todd, *Surviving the SOC Revolution: A Guide to Platform-Based Design*, Kluwer Academic Publishers, 1999.

[8] *Xilinx System Generator v2.1 for Simulink Reference Guide*, Xilinx, 2000.

[9] J. Donaldson, “From Algorithm to Hardware — The Great Tools Disconnect”, *COTS Journal*, pp. 48-54, October 2001.

[10] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. “Ptolemy: A framework for simulating and prototyping heterogeneous systems,” *International Journal of Computer Simulation*, Vol. 4, pp. 155-182, April 1994.

[11] F. Balarin, et al., *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, pp. 10-33, Kluwer Academic Publishers, 1997.

[12] E. Pauer, C. Myers, P. D. Fiore, C. M. Crawford, E. A. Lee, J. A. Lundblad, and C. X. Hylands. “Algorithm analysis and mapping environment for adaptive computing system,” *Proc. Second Annual Workshop on High Performance Embedded Computing*. Boston, MA, pp. 264-265, Sept. 1998.

[13] S. Natarajan, B. Levine, C. Tan, D. Newport, and D. Bouldin, “Automatic Mapping of Khoros-based Applications to Adaptive Computing Systems”, *Proc. of 1999 Military and Aerospace Applications of Programmable Devices and Technologies International Conference (MAPLD)* , pp. 101-107, Laurel, MD, Sept. 1999.

[14] D. Argiro, S. Kubica, “Cantata: The Visual Programming Environment for the Khoros System”, *Visualization, Imaging and Image Processing (VIIP) Conference Proceedings*, Sep. 2001.

[15] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Deckker, 2000.

[16] M. Newman and S. Newman, “New Solutions for Reconfigurable Electronics: Developing Prototypes and Reconfigurable Equipment with Celoxica and Wind River”, Celoxica White Paper, http://www.embedded-solutions.ltd.uk/products/technical_papers/white_papers

[17] Rincon Research Corporation, *Introduction to Midas 2k*, Tucson, AZ, Jan 2000.

[18] R. Andraka, “High Performance Digital Down-Converters for FPGAs”, *Xcell Journal*, Issue 38, Winter 2000, Xilinx, pp. 48-51.

[19] Annapolis Micro Systems Incorporated, *WILDSTAR Reference Manual, rev 3.1*, Annapolis, MD, 2000.