In Proceedings of the Asia South Pacific Design Automation Conference,
Kitakyushu, Japan, January 2003.

Logic Foundry: Rapid Prototyping of FPGA-based DSP Systems

Gary Spivey
Rincon Research Corporation
Tucson, AZ, USA
spivey@rincon.com

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

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

Abstract – The Logic Foundry is a system for the creation and integration of FPGA-based DSP systems. Recognizing that some of the greatest challenges in creating FPGA-based systems occur in the integration of the various components, we have developed a system that addresses the following four areas of integration: design flow integration, component integration, platform integration, and software integration. Using the Logic Foundry, a system can easily be specified, and then automatically constructed and integrated with system level software.

I. Introduction

A large number of system development and integration companies, laboratories, and government organizations exist that have traditionally produced applications requiring rapid development and deployment as well as ongoing design flexibility. These applications are generally low-volume and frequently specific to defense and government requirements. This task has generally been performed by software applications on general-purpose computers. Often these general-purpose solutions are not adequate for the processing requirements of the applications and the designers have been forced to employ solutions involving special purpose hardware acceleration capabilities.

These special purpose hardware accelerators come at a significant cost. This community does not possess the large infrastructure or volume requirements necessary to produce or maintain special-purpose hardware. Additionally, the investment made in integrating special purpose hardware makes technology migration difficult in an environment where utilization of leading-edge technology is critical and often pioneered. Many of these entities are eyeing FPGA-based platforms as a way to rapidly provide deployable, flexible, and portable hardware solutions.

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 the Verilog HDL (hereafter referred to 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 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 architects 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 architect 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

* S.S. Bhattacharyya was supported in part by the National Science Foundataion (Grant #9734275), and the Advanced Sensors Collaborative Technology Alliance.

algorithms" [9]. 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 Micro Systems 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 Micro Systems hardware [10]. In both of the above cases, the system architect must still be intimate with the underlying hardware in order to effectively integrate the hardware into a given software environment.

Some have proposed using high-level, embedded system design tools, such as Ptolemy [11] and Polis [12]. 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) program was performed by Sanders (now BAE Systems) [13] that was successful in transforming a synchronous dataflow graph into a reasonable FPGA implementation. However, this effort was strictly limited to the implementation of a signal processing datapath with no provisions for run-time control of processing elements. Another ACS effort, Champion [14] was implemented using Khoros's Cantata [15] as a development and simulation environment. This effort was also limited to datapaths without run-time 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-end*s are generally designed on an application-by-application basis and for specific software environments. Because the community 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.

It is our experience that one of the greatest challenges in designing these solutions is the integration of the hardware into a pre-existing system. 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.

The Logic Foundry was created for four areas of integration that present challenges in rapid prototyping of FPGA-based DSP systems: design flow integration, component integration, platform integration, and software integration. Each area of integration in the Logic Foundry operates independently. While the Logic Foundry provides easy linkages between all areas, a user might make use of but one area, allowing the Logic Foundry to be adopted incrementally throughout the design community.

This paper gives an overview of how the Logic Foundry is used in the rapid prototyping, development, and deployment of FPGA-based DSP systems. Sections II through V detail the four areas of integration and how they are addressed by the Logic Foundry design environment.

II. Design Flow Integration

An FPGA design flow is the process of turning an FPGA design into a correctly timed image file used to program the FPGA. Due to the difference in resources between FPGA's and general-purpose processors, the realized algorithm on an FPGA may be quite different than an algorithm originally specified by a system designer. While many languages are being proposed as system design languages (among them C++, Java, and MATLAB), none of these languages perform this algorithmic translation step. Therefore, a uniquely skilled logic designer is generally required to construct an FPGA design in a Hardware Description Language (HDL). While this expert may be required for optimal design entry, many

mundane tasks are performed in the process of converting the design into an FPGA image file using Electronic Design Automation (EDA) tools. We desire to automate many of these steps without inhibiting the abilities of the skilled logic designer.

A. MEADE

To efficiently integrate designs into a user-defined EDA tool flow, we have developed MEADE – the Modular, Extensible, Adaptable Design Environment [16]. MEADE allows users to specify a *node* to represent a design building block. A node can be a small function such as an adder, or a large design like a Turbo-Decoder. Furthermore, nodes can be connected to other nodes or contain other nodes, allowing for design reuse and large system definitions. A node not only contains the elements that are required for a design (e.g., HDL files, synthesis files), but also the information required by the design flow to build the node (e.g., HDL libraries and packages required, sub-nodes included, element dependencies).

MEADE provides an extensible set of *procedures*, *actions*, and *agents*. MEADE procedures are sequences of MEADE actions. A MEADE action can be performed by one or more MEADE agents. These agents are used to either perform specific design flow tasks or encapsulate EDA tools. For example, a *simulation* procedure can be defined that has a sequence of actions – *make*, *analysis setup*, *simulate*, *output comparison*, and *analysis*. If a design house has multiple different simulators, such as ModelSim and NC-Sim, an agent for each simulator exists and is selectable by the user at run-time. The same holds true for any other tools (analysis, synthesis, etc.).

The MEADE agents extract design information from the nodes when operating on them. Design flow details are localized in the node by the designer building the node. When the node is used in a larger system, the system designer does not need to know the information required to build a sub-node as that information is automatically acquired from the sub-node by MEADE. This feature enables efficient design reuse and provides a mechanism for IP transfer between different design groups.

MEADE also provides the ability to specify unique 'builds' within a given node. For example, a node can be delivered with Verilog HDL, VHDL, or SystemC implementations, or with generic, Xilinx, or Altera implementations. These builds can easily be specified by a top-level so that if an Altera build is desired, the top node specifies the Altera build, and then any build that has an Altera option uses its custom Altera elements. Those elements that are generic continue to be used.

B. EP3

While most of the flow management in MEADE can be done by tracking files and data through the MEADE agents, some processes require that files be generated or modified in unique and complex manners. For these instances, a preprocessor step has proved effective for many of the detailed MEADE files.

The advantage of using a preprocessor rather than a code generation program is that it gives the HDL designer the ability to use automation where wanted, but the freedom to enter absolute specifications at will. This is an important feature when developing sophisticated systems as the designer typically ventures into areas that the tool programmer had not thought of.

Traditional preprocessors come with a limited set of directives, making some file manipulations hard or impossible. To rectify this we developed the extensible Perl pre-processor (EP3) [17]. EP3 enables a designer to create their own directives and embed the power of the Perl language into all of their files – linking them with the node and enabling MEADE to dynamically create files for its processes. Because it is a preprocessor rather than an explicit file manipulator, the designer can easily and selectively enact or eliminate special preprocessing directives in choice files for specific agents.

EP3 has been extended not only to parse files, but to read in specification files, build large tables of information, and subsequently do dynamic code construction based on the information. This allows for a simple template file to create a very complex HDL description with component instantiations and interconnections done automatically and with error checking.

III. Component Integration

One of the challenges in rapidly creating FPGA-based systems is effective design reuse. Many designers find it preferable to redesign a component rather than invest the time required to effectively integrate a previously designed component. As integration is typically done in the realm of the logic designer, a system designer cannot prototype a system without requiring the detailed skills of the logic designer. The Logic Foundry provides a component abstraction that makes component integration efficient and provides MEADE constructs that allow a system designer to create prototype systems from existing components.

A Logic Foundry component specifies *attributes* and *portals*. If you think of a component as a black box containing some kind of functionality, then *attributes* are the lights, knobs, and switches on that box. 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.

A. The Attribute Interface

Other attempts at FPGA-based development systems have assumed that the FPGA implementation is simply a static data modifying piece in a processing chain [13,14]. 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. The specification of this interface is left as an implementation specific detail for each platform (interface mapping to platforms is described in Section 0). 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 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 main 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.

The strobe line from the attribute interface is sent to every component via distributed delay chains and is used by the components for bus synchronization on the attribute bus. Using delay chains costs very little in an FPGA as there are typically a large number of unused registers throughout a given design. Data and control are multiplexed on the bus and handled by small state machines in each component which provide simple address, control, and data buses inside each component.

Each component also has an individual event signal that is passed back to the main 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.

B. 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.

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.

While the attribute interface handles data-asynchronous control, we have defined a packet specification that allows for data-synchronous control by allowing control and data to flow serially on the same paths between components. This specification is beyond the scope of this paper.

C. The Component Specification

A component is implemented as a MEADE node that contains a component specification file. This file describes all attributes for a component, as well as its portals. Attributes can be declared with varying widths, lengths, and initial values. Because attribute portals and streaming portals are typical for components, EP3 directives exist for simple construction of these ports. If desired, unique portal types can easily be declared and constructed. The component specification file is included via EP3 in the component HDL specification. EP3 automatically generates all of the attribute assignment and read statements and connects up the attribute interface.

IV. 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 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. However, as considerably

better platforms are always being developed, it is necessary to be able to rapidly port to these platforms.

To combat this problem, we are using an *abstract portal* for design level interfaces. A design can be specified in a *design* node (as opposed to a *component* node) with *abstract portals*. Design nodes are completely platform independent and use generic portals. *Abstract portals* are connected to component portals when building a design. These *abstract portals* can then be mapped to a specific platform portal in what we call an *implementation* node.

A. Abstract Portal Types

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.

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.

Memory portals implement a standard memory interface with arbitration via request and acknowledge lines. 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.

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.

B. The Design Specification

The design is constructed as a MEADE node that contains a design specification file. This file describes the components included in a design as well as the design portals. Components are connected to other components or portals via their ports. Designs are platform independent.

The design specification file is included via EP3 in the design HDL specification. The design HDL specification is a shell HDL template that is completely filled in as EP3 instantiates and interconnects all of the design components. The portals become nothing more than HDL ports in the top-level HDL design file. EP3 checks to ensure that all port connections are correct in type, direction, and size. It also assigns addresses to each component. In the HDL testbench, all of the portals supply test models so that the design can be fully simulated as a platform independent design.

In the MEADE design node, the top-level HDL specification is generated via EP3, and the entire design can be simulated and synthesized with MEADE.

C. The Implementation Specification

The final platform implementation is implemented as a MEADE node that contains an implementation specification file. The implementation specification file includes the design to be implemented as well as a map for each portal to an implementation specific interface. Individual components of the design may be mapped to different FPGA's on a platform with multiple different FPGA's. The implementation specification file is included via EP3 in the implementation HDL specification.

For the purpose of this work, we will focus on a single FPGA implementation and do the implementation by hand. However, it is at this point that other research efforts could be facilitated, performing partitioning and mapping of the design components. This problem becomes more interesting when each component has both FPGA and DSP chip implementation described within the node. If a platform consists of both an FPGA and a DSP chip, the system we are describing would provide an excellent foundation for research work in automated partitioning and mapping for hardware software co-synthesis [18].

Each platform to be used in the Logic Foundry requires that implementation specific portals are written for that platform. Once this has been completed, any prior Logic Foundry design can be mapped to that platform - assuming the platform can support all specified design portals. For our DSP applications on standard FPGA platforms, this is a reasonable assumption.

V. Software Integration

Another challenge encountered when creating a special purpose hardware solution is the custom software that must be developed to access the hardware. Often, a completely new software interface is developed for each application to be placed on a platform. When changing platforms, the entire software development process may be redone for the new application. It is also desirable to embed the performance of FPGA-based processors into different application environments. This requires understanding of both the application environment and the underlying FPGA-based system – knowledge that is difficult to find.

To resolve this problem we have developed the Dynamic Object (DynamO) model. This model provides a very thin API designed to provide a clean abstraction between FPGA boards or emulators and application environments.

The DynamO API represents the contract that DynamO back ends and front ends need to follow. Dynamo back ends are wrappers around the board-specific drivers or emulators. The front ends are plug-ins to higher level development environments like MATLAB, Python, Perl, and Midas 2k (a commercial DSP software product). The DynamO API consists of a few calls to allocate a system, to get and set attributes, and to write and read portals. These calls are implemented by the back-end library as the functionality is unique to each back-end platform.

A. DynamO API

The API *system* call uses 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). This object is dynamically built at allocation time and contains objects representing every component and portal in the system. Each component can contain attributes as well as other components. Each object also has methods that allow access to the attributes and portals. 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.

This methodology allows an application to be developed focusing on the elements to be interacted with and not the interfaces. By using a software back end (such as C code written in SystemC), an FPGA system can be modeled in software. Then the entire 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 front-end application requires no change.

B. 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.

With the Logic Foundry, each portal that has been mapped in an implementation requires a software library method to access it. Consider the Annapolis MicroSystems Starfire board. For the attribute portal, set and get methods are provided for the Starfire board that wrap the driver calls to communicate with the attribute portal. Read, write, and query methods are provided for the DMA driver calls that communicate with dma_stream_in and dma_stream_out implementation portals.

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 any of the former Logic Foundry components. A designer could build a completely unique implementation, and then specify the underlying objects and methods for accessing them in a specification file.

C. DynamO Front Ends

The DynamO front end is responsible for taking the DynamO object returned by the allocate 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. Additionally, the front end is responsible for any type conversion that may be required.

A DynamO front end is possible for many software environments. It is easier to implement applications in a multi-threaded environment as the application does not have to be concerned with the possibility of blocking on portal reads and writes. We have implemented a DynamO front end in C++, Python, and Midas2k.

VI. 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, 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 TurboDecoder.

A. 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. 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.

B. 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).

C. 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 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.

D. 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.

Table 1: Summary of Designs

	Ctrl Clk	Sys Clk	LUT's	FF's	RAM's
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

VII. Conclusion

We have shown how the Logic Foundry approach allows for the rapid prototyping and deployment of FPGA-based systems. Using MEADE and EP3, FPGA implementations can be rapidly developed from specifications. 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 have implemented the Logic Foundry 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.

VIII. 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] Xilinx System Generator v2.1 for Simulink Reference Guide, Xilinx, 2000.

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

[11] 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.

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

[13] 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.

[14] 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.

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

[16] G. Spivey and K. Nakajima, “The Philosophy of MEADE: A Modular, Extensible, Adaptable Design Environment,” Proc. of the International HDL Conference and Exhibition (HDLCON), San Jose, CA, pp. 159-165, March 1999.

[17] G. Spivey, “EP3: An Extensible Perl PreProcessor,” Proc. of the International Verilog HDL Conference and VHDL International Users Forum (IVC/VIUF), Santa Clara, CA, pp. 106-113, March 1998.

[18] S. S. Bhattacharyya. “Hardware/software co-synthesis of DSP systems,” in Y. H. Hu, editor, Programmable Digital Signal Processors: Architecture, Programming, and Applications, pp. 333-378, Marcel Dekker, Inc., 2002.