

Logic Foundry: A Rapid Prototyping Tool for FPGA-based DSP Systems

Gary Spivey

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

Shuvra S. Bhattacharyya

ECE Department and UMIACS
University of Maryland, USA
ssb@eng.umd.edu

Kazuo Nakajima

ECE Dept, UMD, USA and
New Architecture Open Lab
NTT Communication Science
Labs, Kyoto, Japan
kazuo@cslab.kecl.ntt.co.jp

Abstract

We introduce the Logic Foundry, a system for the rapid creation and integration of FPGA-based digital signal processing systems. Recognizing that some of the greatest challenges in creating FPGA-based systems occur in the integration of the various components, we have proposed a system that targets the following four areas of integration: design flow integration, component integration, platform integration, and software integration. Using the Logic Foundry, a system can be easily specified, and then automatically constructed and integrated with system level software.

1 Introduction

A large number of system development and integration companies, labs, and government agencies (hereafter referred to as “the community”) exist that have traditionally produced digital signal processing 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. The 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. Recent improvements in Field Programmable Gate Array technology have made FPGA’s a viable platform for the development of special-purpose digital signal processing hardware [1], while still allowing design flexibility and the promise of design migration to future technologies [2]. Many entities within the community are eyeing FPGA-based platforms as a way to provide rapidly 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.

Current efforts like MATCH [3] have attempted to compile high-level languages such as MatLab directly into FPGA implementations. Certain tools such as C-Level Design [4] have attempted to convert “C” software into a hardware description language (HDL) format such as Verilog HDL or VHDL that can be processed by traditional FPGA design flows. Other tools use derived languages based on C such as Handel-C [5], C++ extensions such as SystemC [6], or Java classes such as JHDL [7]. 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” [8] 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 [9]. 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” [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 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 [10]. 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.

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) was performed by Sanders (now BAE Systems) [13] 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 [14], was implemented using Khoros's Cantata [15] 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 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.

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" [8]. To facilitate the rapid implementation and deployment of these platform-based designs, we have identified four areas of integration as targets for improvement in a rapid prototyping environment for digital signal processing systems. These four areas are *design flow integration*, *component integration*, *platform integration*, and *software integration*.

Design Flow Integration — In addition to standardized component development methodologies [16][17], we have also proposed that these pre-verified blocks, be assembled with all the information required for back-end FPGA design automation. This will allow logic designers to "integrate" the FPGA design flow into their components. With tools we have developed as part of the Logic Foundry, a system designer can perform back-end FPGA processing automatically without any involvement with the technical details of timing and layout.

Component Integration — We have proposed that any of the aforementioned pre-verified blocks, or *components*, that are presented to the high level system designer should consist of standardized interfaces that

we call *portals*. Portals are made up of a collection of data and control pins that can be automatically connected by the Logic Foundry while protecting all timing concerns. The Logic Foundry was built with the requirement that it had to handle run-time control of its components; therefore we have designed a control portal that can scale easily with the number of components in the system without adversely affecting overall system timing.

Platform Integration — With the continuing gains in hardware performance, faster FPGA platforms are continually being developed. These platforms are often quite different than the current generation platforms. This can cause portability problems if the unique platform interface details have been tied deeply into the FPGA design (e.g., memory latency). Additionally, underlying FPGA technology changes (for example, from Altera to Xilinx) can easily break former FPGA designs. Because of the community's need to frequently upgrade to the latest, most powerful hardware platforms, Logic Foundry components are developed in a platform independent manner. By providing abstract interface portals for system input/output, and memory accesses, designs can be easily mapped into most platform architectures.

Software Integration — In addition to the hardware portability challenges, software faces the same issues 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. It is also 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 [18] (a software system developed by Rincon Research for digital signal processing). 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 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 an 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 as 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.

Each area of integration in the Logic Foundry can be used 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 community. For clarity, we will begin the Logic Foundry discussion

with a design example, explaining how the design would be implemented in an FPGA, and then how a software system might make use of the hardware implementation. Section 2 introduces this design that will serve as an example throughout the report and Section 3 details prior work in the field. Sections 4 through 7 describe the four areas of integration, and how they are addressed by the Logic Foundry design environment. In Section 8 we describe three different designs that have been built using the Logic Foundry and detail performance results of these designs.

2 Design Example

For an FPGA-based system example, we examine a signal processing system that contains a *Tune/Filter/Decimate (TFD)* process being performed in a general-purpose computer (see Figure 1). The *TFD* is a standard digital signal processing technique used to downconvert a tuned signal to baseband and often implemented in FPGA's [19].

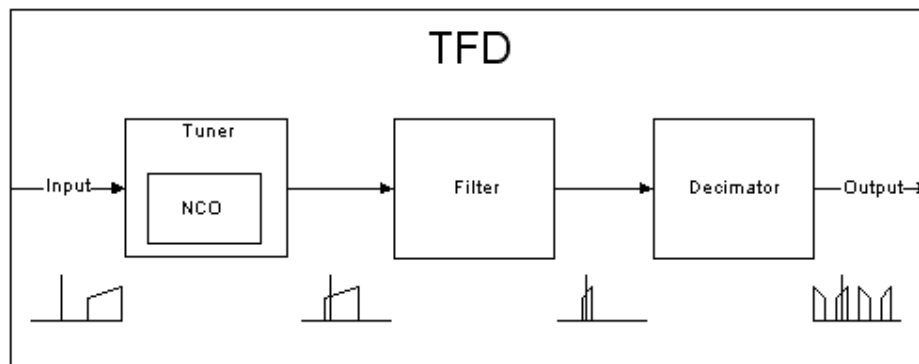


Figure 1: Tune/Filter/Decimate

We would like to move the *TFD* functionality to an FPGA platform for processing acceleration. Inside the FPGA, the *TFD* block will be made up of three cores, a tuner with a modifiable *frequency* parameter, a FIR filter with reloadable *taps*, and a decimator with a modifiable decimation *amount*. The *tuner* core will contain a numerically controlled oscillator (NCO) core as well. The *TFD* will be required to interface to streaming inputs and streaming outputs that themselves interface via the pins of the FPGA to the general-purpose host computer.

The system will stream data through the *TFD* in blocks of potentially varying size. While this is occurring, the system may dynamically change the tune frequency, filter taps, or decimation value in both a *data-asynchronous* and *data-synchronous* manner. We define a data-asynchronous parameter access as a parameter access that occurs at an indeterminate point in the data stream. A data-synchronous parameter access occurs at a determinate point in a data stream.

The output of the *TFD* will be read into a general-purpose computer where software will pass the result on to other processes such as a demodulator. We would like to input the data from either the general-purpose computer or from an external I/O port on the FPGA platform. Rather than have a run-time configurable option, we would like to be able to quickly make two different FPGA images for each case.

In our example, we assume that we will be using an Annapolis MicroSystems Starfire [20] card as an FPGA platform. This card has one Xilinx FPGA, plugs into a PCI bus, and is delivered with software drivers. Our systems application software will be Midas 2k [18].

3 Prior Work

Much of the prior work cited was done in support of the DARPA Adaptive Computing Systems (ACS) Program which was run by the Information Technology Office (ITO) of DARPA. This was a large-scale project with a vision to develop and exploit reconfigurable computing for evolving Defense systems. Systems like MATCH from Northwestern University and DEFACTO from USC/ISI attempted to create FPGA implementations from a higher-level systems language whereas efforts like Champion from the University of Tennessee and the effort by Sanders Corporation attempted to automate FPGA construction using more traditional HDL approaches. The SLAAC effort attempted to define a complete FPGA processing environment for use in the DARPA ACS program.

In addition to these we have cited prior efforts at creating FPGA-based systems as well as other tools that were used to support the efforts mentioned. It is important to note that technology changes within the past few years have resulted in FPGA's much larger than those previously seen. Where some former tools looked at FPGA's as coprocessors, and thus focused on issues of hardware/software co-synthesis (e.g. see [21],[22]) today's FPGA have the possibility of being used as complete processing subsystems. This larger scale of device presents problems such as control of the FPGA and design methodology management that were much simpler in smaller FPGA designs. The Logic Foundry addresses this latter problem, attempting to allow FPGA designers to create component implementations and then automatically integrate these components into a processing subsystem that can be easily integrated into a software application environment such as Python or MATLAB.

Some of the following tools address data synchronization and static-rate, dataflow modeling (e.g. using synchronous or cyclo-static dataflow [23],[24]) in order to meet predictability requirements for automated assembly or hardware/software cosynthesis algorithms. The Logic Foundry does not seek to integrate FPGA components with microprocessor programs automatically, but rather to facilitate the development of FPGA

subsystems that can be easily controlled and interacted with (with software, other FPGA subsystems, ASICs, or other types of processing platforms). For this reason, the Logic Foundry imposes minimal dataflow restrictions on its component designers, using a general, self-timed connection model for dataflow, which, while less predictable, is more flexible than static-rate dataflow.

Future Logic Foundry work could look at the adaptation of its FIFO management mechanisms to specific dataflow models (like synchronous or cyclostatic dataflow). With this effort, the Logic Foundry could be used to create components that could be described in these other environments, and then used as the lower level building blocks in a larger system analysis.

We begin with a look at some high-level system modeling environments, move into a group of current design entry methodologies, and then examine some of the more recent DARPA efforts.

3.1 Polis

The POLIS project is being conducted in the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley. The project is directed by managed by the Hardware/Software Co-design Group. POLIS is an embedded system design framework.

The authors of POLIS state: “There are many different academic approaches to try to solve the problem of embedded system design. In our opinion, none of them address satisfactorily the issues of unbiased specification and efficient automated synthesis for control-intensive reactive real-time systems. Therefore, we are developing a methodology for specification, automatic synthesis, and validation of this sub-class of embedded systems (that includes the examples described above). Design is done in a unified framework, POLIS, with a unified hardware-software representation, so as to prejudice neither hardware nor software implementation. This model is maintained throughout the design process, in order to preserve the formal properties of the design” [12].

POLIS is a tool that attempts to solve high level abstraction issue — mainly embedded system hardware-software co-design. POLIS attempts to understand the properties of the lower level building blocks (either hardware or software implementation). Using the POLIS system, a user can rapidly explore design implementation trade-offs between hardware and software partitions and have POLIS attempt to interface the created subsystems.

Our system would properly sit underneath POLIS as a way to both define hardware interfaces and hardware components that could be used in POLIS. There is no notion in POLIS of automating the hardware implementation process as we have defined.

Integrating POLIS with our system would be a useful and challenging exercise when implementing a platform that contains FPGA's and independent processing elements (DSP's or microprocessors). Another interesting application would be the use of FPGA based microprocessor cores inside of the FPGA itself.

3.2 Ptolemy

The Ptolemy project is being conducted in the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley.

“The Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent systems. The focus is on embedded systems, particularly those that mix technologies, including for example analog and digital electronics, hardware and software, and electronics and mechanical devices. The focus is also on systems that are complex in the sense that they mix widely different operations, such as signal processing, feedback control, sequential decision making, and user interfaces” [11]. Ptolemy focuses more on higher-level system abstractions, with the potential for low-level code generation being pushed to the user via back end compilers.

“Ptolemy II has an evolving code generation mechanism that is very different from that in Ptolemy Classic. In Ptolemy Classic, each component has to have a definition in the target language, and the code generator merely stitches together these components. In Ptolemy II, components are defined in Java, and the Java definition is parsed. An API for performing optimization transformations on the abstract syntax tree is defined, and then compiler back ends can be used to generate target code” [25].

It is the goal of our research to produce the low-level component definitions for a specific hardware implementation and link them into the application software. Ptolemy could be used at a higher level to link the defined special purpose system into a larger level system. The Ptolemy II back end compiler would need to be developed that could possibly generate all or some of the component/design/implementation specification files depending on the granularity desired and success in efficiently implementing high to low-level translations. While we believe the component compiler would be quite difficult to effectively implement generally, a smart component parameterizer could use MEADE to take existing generic components and automatically create the requested specific component. Ptolemy could then very well be a successful method for creating the design or implementation specification files and integrating the resultant implementation into a wider system.

3.3 GRAPE-II

“Grape-II (Graphical Rapid Prototyping Environment) is an advanced system-level development environment for specifying, compiling, debugging, simulating, and emulating digital-signal-processing

applications. Its structured prototyping methodology reduces programming effort, and its use of general-purpose reusable hardware minimizes development cost. The general-purpose hardware consists of commercial DSP processors, bond-out versions of core processors, and FPGA's linked to form a powerful, heterogeneous multiprocessor. GRAPE II automates the prototyping methodology for these systems by offering tools for resource estimation, partitioning, assignment, routing, scheduling, code generation, and parameter modification" [26].

The input to GRAPE-II was a cyclostatic dataflow model of the system. The output of the system was available on a range of platforms, including FPGA, DSP, and heterogeneous systems. GRAPE-II differs from the Logic Foundry in that it focused on creating a heterogeneous multiprocessor rather than an FPGA-based subsystem. Where the GRAPE-II system used a cyclostatic dataflow model of a system, the Logic Foundry allows a more flexible, though less predictable, dataflow model, using FIFO's to self-time interconnects between components. When a design failed to meet timing constraints in the GRAPE-II system, the designer attempted to manually optimize the design. The Logic Foundry attempts to use pre-verified components and interconnect them in a manner that provides timing isolation to each component. In this manner, FPGA-based subsystems can be constructed with minimally-constrained, inter-component timing.

3.4 JHDL

"JHDL is a set of FPGA CAD tools developed at BYU that allows the user to design the structure and layout of a circuit, debug the circuit in simulation, netlist and interface with back-end tools for synthesis, and so forth. It is an exploratory attempt to identify the key features and functionality of good FPGA tools. The design of an FPGA system has three major arenas:

The structure or organization of the circuits.

The layout of the circuits.

The interface of the FPGA circuits with the host application software.

All traditional FPGA tools present some method for designing the circuit structure. A few of these also permit the user to perform the layout of the circuit; more often, circuit layout must be performed with a separate, non-integrated tool. But almost no tools provide a way to naturally interface the running hardware platform with the software running on the host machine. This last issue is important: FPGA-based systems typically operate in tandem with a general-purpose host microprocessor and it is important to simulate the entire system, including the host computer system and its application software in conjunction with the FPGA design to ensure that the entire application works as desired" [7].

JHDL uses a high level Java description rather than a traditional HDL. This means that for any design created in JHDL, JHDL tools must be used to implement the design. Currently, JHDL only supports Xilinx 4K, Virtex, and Virtex II series devices. The Logic Foundry uses standard industry tools and is capable of implementing FPGA's on any architecture supported by standard industry tools.

JHDL does not attempt to define component interactions or structures for control of components like the Logic Foundry does. However, JHDL offers some similarity for platform abstraction and software integration with its HWSysClass. This class abstracts 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 as they are in the Logic Foundry. Furthermore, the HWSysClass abstracts only the interface, and not the components residing on the other side of the interface. This means that the application programmer interfacing to an FPGA subsystem using the HWSysClass would still have to be acquainted with details of the internal architecture. The Logic Foundry encapsulates all details of the FPGA implementation in component objects so that an application programmer can interact with objects and methods rather than addresses and control protocols.

While JHDL is distinctly different than what we are proposing, it is interesting to note what the JHDL creators considered important — a natural “interface to the running hardware platform with the software running on the host machine”. We believe that, for the DSP domain, the Logic Foundry is such a system. While slightly different than the JHDL single language approach, the ability to either easily integrate the HDL simulator into the application software in place of the hardware or, using software implementation models stored in the MEADE component, our system effectively integrates either a simulation of the entire system or the actual system itself.

3.5 C-Level Design

C-Level Design was a tool that took ANSI C descriptions and converted them into HDL descriptions. The language had many pragmas that could be inserted into the standard ANSI C description to help tailor the HDL conversion.

In 2001, C-Level Design ceased operations and their assets were absorbed by Synopsys Corporation. Synopsys plans to integrate C Level Design's CycleC simulation technology into Synopsys' VCS simulator to accelerate HDL simulation. Synopsys has also agreed to acquire the technology behind C Level's System

Compiler software for RTL-C synthesis, as well as the Panchul patent for synthesizing high level languages into HDL.

3.6 Handel-C

Handel-C, from Celoxica is a language for implementing algorithms in hardware, architectural design space exploration, and hardware/software co-design. Based on ISO/ ANSI-C, it has extensions required for hardware development. These include flexible data widths, parallel processing and communications between parallel elements. The language is designed around a simple timing model that makes it very accessible to system architects and software engineers.

Celoxica does not take the approach of converting the Handel-C specification to HDL, but provides a tool that synthesizes the Handel-C directly to the back-end technology. The Logic Foundry has been built using VHDL, but could be adapted to integrate a Handel-C based design entry approach.

Celoxica has also undertaken work in the arena of system software integration as a joint venture between Wind River with their Board Support Package (BSP) and Celoxica's Platform Abstraction Layer (PAL) [27]. This effort attempts to abstract the I/O interfaces of a processing platform, allowing an application that is developed on one platform to be migrated to another platform. However, the front-end application must still be intimate with the internals of the back-end hardware. The Logic Foundry differs from this approach in that it completely abstracts all of the hardware implementation details from the software application.

3.7 SystemC

SystemC is a modeling platform consisting of C++ class libraries and a simulation kernel for design at the system-behavioral and register-transfer-levels. Designers create models using SystemC and standard ANSI C++. EDA vendors create tools that are automatically interoperable.

The Logic Foundry was developed to support a VHDL design flow, but architected to be able to integrate other design entry specifications with SystemC being the primary consideration. It is also a future plan to create a SystemC DynamO back-end that allows designers to model FPGA designs and integrate the model with front-end applications.

3.8 Xilinx System Generator

“The Xilinx System Generator bridges the gap between the high-level abstract version of a design and its actual implementation in a Xilinx FPGA. The System Generator for DSP, developed in partnership with The

MathWorks, Inc. enables designers to develop high-performance DSP systems for Xilinx FPGA's using the popular MATLAB/Simulink products from The MathWorks, Inc.

The Xilinx System Generator runs in the framework supplied by the Simulink modeling software. System Generator provides a bit and cycle-accurate model of FPGA circuits, and automatically generates a synthesizable Hardware Description Language (HDL) code and a testbench. The HDL code can be synthesized for implementation in Xilinx Virtex-II, Virtex-E, Virtex, and Spartan-II FPGA's. With the push of a button, the abstract representation of a system-level design is transformed into a gate-level representation. Automatic generation of a testbench enables design verification upon implementation. To maximize predictability, density, and performance, the System Generator automatically maps the system design to Xilinx optimized LogiCORE modules."

The Xilinx System Generator differs from the Logic Foundry in that it uses strictly Xilinx LogiCORE components. Other components can be built, but are included in the MATLAB specification as black-boxes and cannot then be simulated with the system.

The Xilinx System Generator is promising as it grows to include more than Xilinx LOGICore components. However, its proprietary nature is limiting for applications on other platforms. The Logic Foundry can make use of the System Generator as a graphical front-end and potential constructor of top-level implementations. The Logic Foundry would complement the System Generator by providing an architecture interface as well as automated software integration and design flow management.

3.9 MATCH

The MATCH effort was part of the DARPA funded ACS program. The objective of the MATCH (MATLAB Compiler for Heterogeneous Computing Systems) compiler project is to "make it easier for users to develop efficient codes for configurable computing systems" [3]. Towards this end, the MATCH project takes MATLAB descriptions of various embedded system applications, and automatically maps them onto a configurable computing environment consisting of field-programmable-gate-arrays, embedded processors, and digital signal processors built from commercial, off-the-shelf components. The MATCH authors state that "the goal of our compiler is to generate efficient code automatically for such a heterogeneous target that will be within a factor of 2-4 of the best manual approach with regard to 2 optimization objectives: (1) Optimizing resources (such as type and number of processors, FPGA's, etc.) under performance constraints (such as delays and throughputs) (2) Optimizing performance under resource constraints."

MATCH differs from the Logic Foundry in that it is trying to develop an integrated compilation environment for generating code for DSP and embedded processors, as well as FPGA's and attempts to fall within a factor of 2-4 of the best manual approach for the system. The Logic Foundry focuses on FPGA-based implementations only, and attempts to approach the best manual approach by using pre-verified, performance-tuned blocks.

3.10 DEFACTO

DEFACTO is an 'end-to-end' design environment for developing applications mapped to adaptive computing architectures. A user of DEFACTO develops an application in a high-level programming language such as C, possibly augmented with application-specific information. The system maps this application to an adaptive computing architecture that consists of multiple FPGA's as coprocessors to a conventional general-purpose processor" [28].

The Logic Foundry differs substantially from DEFACTO in that the Logic Foundry is focused on optimizing the implementation of an FPGA-based system and automating routine stages of design methodology. DEFACTO focused on a DARPA computing platform and performed high-level synthesis of HDL code from constrained C or MATLAB code, integrating the FPGA's as coprocessors to a general-purpose processor. Where DEFACTO attempts to synthesize an entire design, the Logic Foundry attempts to construct a design from pre-verified, performance-tuned blocks.

3.11 Sanders

Sanders — A Lockheed Martin Company has worked in the ACS program to provide "Algorithm Analysis and Mapping Environment for Adaptive Computing Systems" [13]. Their approach has been to "focus on three areas of capability critical to the success of adaptive computing: algorithm analysis, algorithm mapping, and smart generators. These capabilities are taking advantage of the special characteristics of signal processing algorithms to reduce the time to field the ACS, and are being implemented as extensions to the Ptolemy design environment developed at the University of California, Berkeley."

This approach is probably the most closely related of all related work. The Sanders ACS effort used Ptolemy Classic and created a specific an ACS domain. Methods for logic generation were built around libraries of functions or general-purpose logic synthesis. To perform this task the Sanders ACS team utilized "smart generators" that were capable of choosing or parameterizing parts based on specific implementation techniques/trade-offs.

One difference between the Sanders ACS effort and the Logic Foundry is in design automation. The Sanders ACS effort leaves the tasks of synthesis and FPGA design implementation to an FPGA designer, whereas the Logic Foundry has provided MEADE to help automate this task.

The key difference between the two efforts is in the Logic Foundry's ability to supply run-time control to the components of the system. The Sanders ACS effort was focused on datapath construction and provided no mechanism for run-time control of components. Furthermore, the Sanders ACS effort has no provision for data-synchronous control, and no general software API to interface to Midas2k or MATLAB or other software application environments.

3.12 Champion

Champion is a software design environment for adaptive computing systems and ASIC's being developed at the University of Tennessee Microelectronic Systems Research Laboratory [14]. Champion uses the Cantata graphical programming environment from Khoral Research, Inc. to graphically represent a set of functional blocks connected by signal paths. Champion distinguishes themselves from other methods in that they create hardware models and synthesize and layout these models into library cells before being used in creating a system. Because a large part of their system is the automatic partitioning of these blocks onto the FPGA the pre-compiled blocks give accurate timing and resource value and provide a larger granularity for automated partitioning.

The Logic Foundry differs in many respects with the implementation of Champion. There is no flow control described in Champion. Champion renders the entire design flow as a graph and uses a group of mathematical algorithms to discover timing mismatches that cause synchronization errors. When this is completed, Champion inserts the requisite delay elements on the entire datapath. Therefore, Champion's implemented design with its synchronized data streams is static and cannot be used to handle control packets embedded in the data streams for data-synchronous control. The Logic Foundry uses a FIFO interconnect approach. The self-timed FIFO's can be used to perform data-synchronous control using control packets embedded in the data streams.

The different methods for component interconnect can have performance tradeoffs depending on the number of delay elements required by Champion's implementation. For large delay differences, the FIFO approach reduces resource count, whereas, if no delay elements are required, the FIFO approach adds resource. However, the Logic Foundry separates the interfacing of components from the component behavior, and in this latter case, FIFO's can easily be swapped with registration or pass-through stages.

Champion also defines no board interface abstraction for portability, no control interface for component attributes, and no clearly defined software interface to allow other software application environments other than the Cantata environment. The Logic Foundry was designed to interact with a variety of front-end software application environments.

Champion was effective in the partitioning and mapping arena because of its coarse granularity and precise resource and timing values. Further research in automated mapping of our systems should look to Champion as a model for a pre-compiled, component based, bottom-up development system.

3.13 SLAAC

The Systems Level Applications of Adaptive Computing (SLAAC) project was conducted by the University of Southern California-Information Sciences Institute as part of DARPA's ACS program. SLAAC project attempted to define an open, distributed, scalable, adaptive computing systems architecture based on a high-speed network cluster of heterogeneous, FPGA-accelerated nodes.

“The system-level focus of the SLAAC project came about because of the realization that scalability and portability are the two primary obstructions preventing innovative ACS research from being directly useful in deployed systems” [29].

To solve the problem of scalability the SLAAC project attempted to define and develop a scalable API and runtime software to support application control of network-distributed multiple-host multiple-board ACS systems. To this end, the SLAAC project was successful in providing a concrete integration path for a demonstration system with other ACS program participants. This differs from the Logic Foundry in that we abstract hardware completely from the software and allows the upper level software system to independently manage local resources. Thus, the problem of network-distributed hosts and large application systems is left to the software implementation domain.

Where the SLAAC project attempted to define a hardware architecture for other boards to follow, the Logic Foundry created platform interface abstractions and attempts to map them to other platform implementations, thus allowing designs to be portable among different FPGA platform architectures.

3.14 CoWare N2C Design System

CoWare N2C is an environment and methodology for the development and proliferation of embedded processor based designs. Embedded systems contain both hardware and software components. The connection between these components consists of address decoders, bridges, interrupt priority encoders and software

drivers. The actual implementation of the components is dependent on the protocols associated between the different elements. CoWare N2C provides the ability to automate the integration of components of differing types through a technology called Interface Synthesis™.

CoWare provides no mechanism for FPGA creation, but does provide a mechanism for interfacing the FPGA to the world around it. The interface synthesis technology is a more robust version of our interface abstractions. However, the interface synthesis model is focused on outside protocols whereas our interface abstractions are focused on chip internals. The CoWare system could be quite effective in creating the platform dependent hardware interfaces and the software the goes with them.

4 Design Flow Integration

An FPGA design flow is the process of turning an FPGA design description into a correctly timed image file with which to program the FPGA. Implementing a design on an FPGA requires that (typically) a design be constructed in a hardware description language (HDL) such as Verilog HDL or VHDL. This must be done by a uniquely skilled logic designer who is generally not as involved in the system design process. It is important to note that often, due to the difference in resources between FPGA's and general-purpose processors, the realized algorithm on an FPGA may be quite different than the algorithm originally specified by the 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. A common belief in the industry is that there will always be a place for the expert in the construction of FPGA's [30]. While an expert may be required for optimal design entry, many mundane tasks are performed in the design process using a unique set of Electronic Design Automation (EDA) tools. It is desirable to automate many of these steps without inhibiting the abilities of the skilled logic designer.

4.1 MEADE

To more efficiently integrate FPGA designs into a user defined EDA tool flow, we have developed MEADE – the Modular, Extensible, Adaptable Design Environment [31,32]. MEADE has been implemented in Perl because of its widespread use in the community and dominant success as a glue language and text parser, two requirements for an integration framework for FPGA design flows.

MEADE requires 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. In the *TFD* example, nodes exist for the *TFD*, the *tuner*, the *filter*, the *decimator*, and the *NCO* within the tuner (see Figure 2).

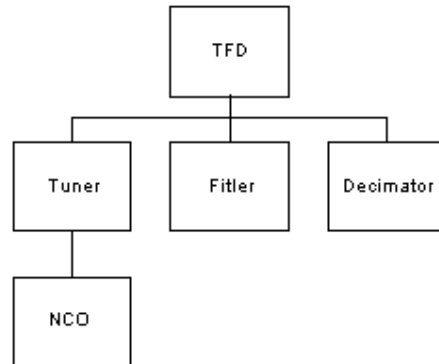


Figure 2: MEADE Node Structure for the Tune/Filter/Decimate

MEADE nodes are directory structures with an accompanying database that fully describes the aspects of the node. The database is contained in a `.meade` subdirectory via persistent Perl objects [33]. The database includes information about node elements such as HDL models and testbenches, target and included libraries, and included packages. This information includes file location, any node children, and special “blackboards” that can be written and read by MEADE components for extensible requirements.

MEADE nodes also provide the ability to specify unique ‘builds’ within a given node. Using the build mechanism, a node can be delivered with Verilog HDL, VHDL, and 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.

To manipulate the nodes and node information, MEADE contains an extensible set of *MEADE 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”, “Debug Setup”, “Simulate”, “Debug” and “Output Comparison” (see Figure 3). If a design house has multiple different simulators, such as Mentor Graphics ModelSim or Cadence NC-Sim, or third party debuggers such as Novas Debussy, 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.). We have currently implemented simulation agents for Mentor Graphics ModelSim simulator, analysis agents for ModelSim and Novas’ Debussy debugger, and synthesis agents for Synplify’s Synplicity synthesis tool.

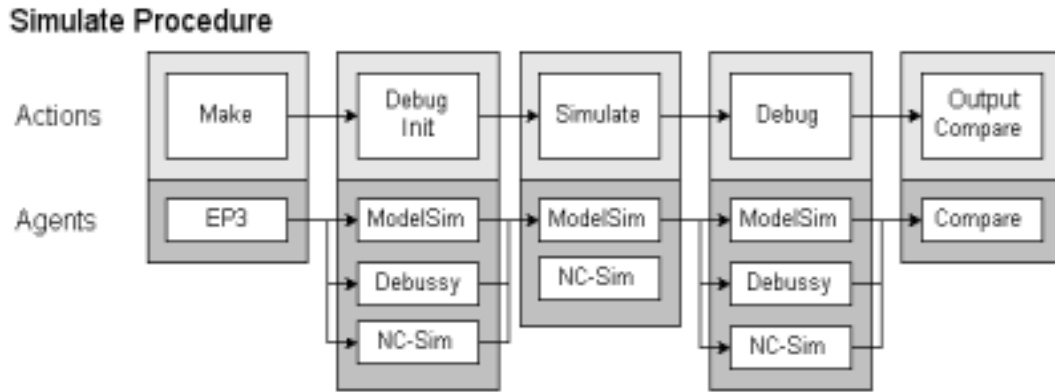


Figure 3: The MEADE Simulate Procedure

MEADE provides node generation procedures that construct standard nodes with HDL templates for the design and testbenches. To accommodate rapid testbench construction, MEADE employs a client/server testbench model [34] and supplies a group of test modules for interfacing to HDL debuggers. Design flow scripting is typically automated by MEADE, but custom tool scripts can be designed by the node designer. This information is localized to the node being designed, by the designer building the node. When 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 makes MEADE nodes very usable as methods of IP transfer between different design groups using MEADE.

4.2 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 manipulated in unique and complex manners. Additionally, this manipulation is not always desirable to be done in the background in the event that the core designer may have expert custom tailoring that the agent designer cannot anticipate. In these instances, we have found that a preprocessor step is an excellent option for many of the detailed MEADE files.

The advantage to 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 this end we developed the extensible Perl pre-processor (EP3) [35]. 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.

Originally, EP3 was designed as a Verilog HDL preprocessor, but as it was developed, we decided that it should be simply an extensible standard preprocessor with the ability to dynamically include directive modules (for Verilog HDL, VHDL, etc.) at compile time or in the middle of a run. EP3 scans a file, looks for directives, strips off the delimiter, and then calls a function of the same name. The standard directives are defined within the EP3 program. Library-directives or user-defined directives may be loaded as Perl modules via a command line switch for inclusion at the beginning of the EP3 run. Perl subroutines (and hence EP3 directives) may be dynamically included during the EP3 run by simply including the subroutine in the text of the file to be preprocessed.

EP3 has been extended to not only 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.

4.3 Design Flow Integration Example

Consider the construction of the *NCO* node in the *TFD* example. We begin by first creating a MEADE node with the command: `meade node NCO`. This creates a directory entitled *NCO*. Inside of this directory, `src` and `sim` subdirectories are created. Template source files (`NCO.ep3`, `NCO_pkg.ep3`, and `NCO_tb.ep3`) are copied from the global MEADE configuration space and modified with the new node name *NCO*. Element objects for each of these files are automatically created in the node's database. The database would also be populated with a target compilation library for the node and a standard build. The package file includes the VHDL component specification for this entity – this definition is automatically included in the design file and the testbench automatically by EP3 so that component specifications can be entered once rather than the several times standard HDL entry requires. The testbench file includes modules that provide system and data clocks, resets, and interfaces to debuggers in a format for run-time configuration by the MEADE simulation agents.

After editing the files to create the desired VHDL component, the command `meade make` will invoke the EP3 agent to run EP3 on the files and produce the output files `NCO.vhd`, `NCO_pkg.vhd`, and `NCO_tb.vhd`. The make procedure is often a subset of other procedures and does not necessarily have to be run independently. Entering the command `meade sim` will execute the default simulator, MentorGraphics'

ModelSim. This involves the creation of a `modelsim.ini` file that provides linkages to all required simulation libraries. In a low-level node such as this one, there are few libraries – however, all of the MEADE support modules that are included in the testbench have their libraries automatically included in the `modelsim.ini` file at this time. The command line (which can be quite extensive) is formed for the appropriate options and the simulation is run. There are many options that can be handled by the simulation agent, such as whether or not the simulation is to be interactive or batch mode, which debugger format is to be used for data dumps, and simulation frequency to name a few. Simulation output is directed to an appropriate text output files or simulation dump files and managed for the user as are any simulation make files that are created to avoid excessive recompiles. Using similar procedures in MEADE, the node can be run through a debugger (`meade analyze`), or synthesized to a structural netlist (`meade synthesize`).

Using MEADE, designers who may be either learning a hardware description language or unfamiliar with the nuances of many of the tools are able to effectively construct and debug designs. MEADE has been used successfully to automate mundane aspects of the design flow in many applications, including HDL file generation and manipulation, generation of simulation, analysis, and synthesis configuration files, tool invocation, and design file management. Admittedly, some designers find tool encapsulation intrusive and would rather work outside of MEADE when developing cores. In these cases, a finished design can be encapsulated by MEADE in a relatively simple manner.

Upon node completion, everything about the node is encapsulated in the MEADE database. This includes such features as which files required for simulation, which files are required for synthesis, required simulation libraries and simulation target libraries, and any sub-nodes that may be required by the node. When the *tuner* component is constructed, a child reference to the *NCO* node is simply included in *tuner*'s required element files. When any MEADE operations are performed on the *tuner* node, all tool files and command lines are automatically constructed to include the directions specified in the *NCO* node.

5 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.

5.1 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 [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 6). 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 (see Figure 4). Because attribute accesses are relatively infrequent and asynchronous, the attribute bus uses a multi-cycle path to eliminate timing concerns and minimize routing resources. Using a simple incrementer component that has an input, an output, and a single *amount* attribute, we have effectively implemented a design for 1 incrementer, 10 serial incrementers, and 50 serial incrementers with no degradation in performance.

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.

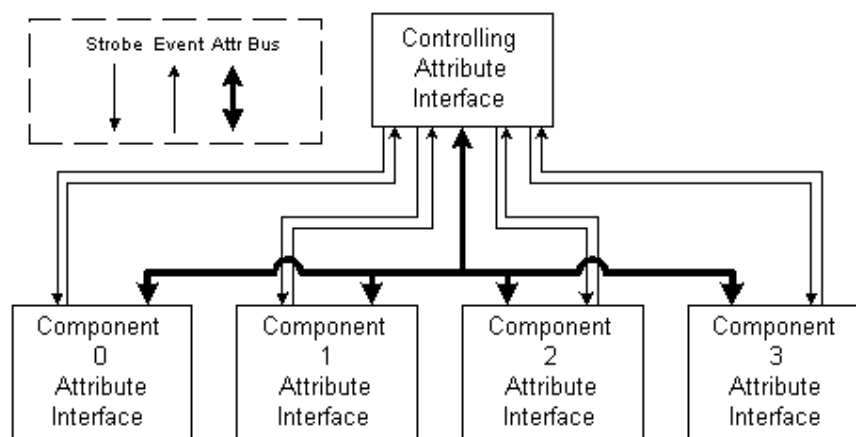


Figure 4: The Attribute Interface

Consider the case in the *TFD* example where a user wishes to dynamically alter the decimation amount. With the implementation that we have developed for the Annapolis MicroSystems Starfire board, the application would first write the controlling attribute interface with: the component address of the *decimator*; the address of the *amount* register within the *decimator* component; the number of words in the transfer; the data to be written; and a control word to initiate the transfer. The controlling attribute interface then begins the process of transferring the data across the attribute bus using the distributed delay chain to strobe the component enable. When the transfer is completed, the controlling attribute interface sets a done flag in its control register and awaits the next transfer.

5.2 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 (see Figure 5). 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 dataflow, 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 [36].

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. Control of the FIFO is performed with simple valid and ready signals. Whenever both valid and ready are active, data transitions.

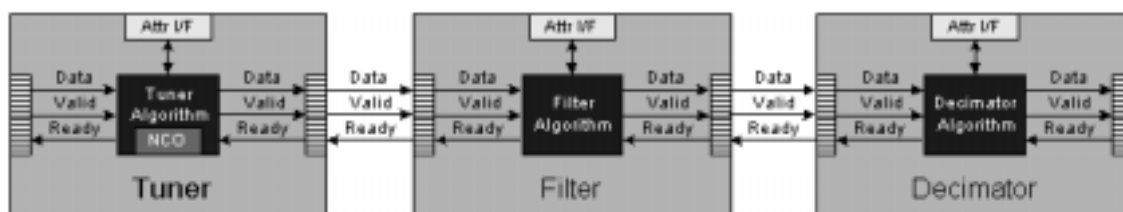


Figure 5: Component FIFO interface

In the *TFD* example, each component receives input and output FIFO's. Note that the *NCO* inside of the *tuner* component is simply a MEADE node and not a component, and thus receives no FIFO's. This allows logic designers to build components out of many sub-nodes, but expose only the top level component to the system designer.

5.3 The Component Specification File

A component is implemented as a MEADE node that contains a component specification file (see **Error! Reference source not found.**). The component specification file describes any attributes for a component, as well as a component's ports and the pins that make up those ports. In the *TFD* example, attributes can be declared of varying widths, lengths, and initial values. The attribute can be written by the system, the hardware, or both. Attribute addresses may be auto-generated. Because attribute ports, and streaming data in and out ports are standard for components, EP3 directives exist to construct these ports. However, any port type can be declared.

```

@attribute_portal
@data_portal in import
@data_portal out export

@attribute {
    name      =>  amount,
    width     =>  IMPORT_DATA_WIDTH
    length    =>  1,
    source    =>  BOTH,
}

```

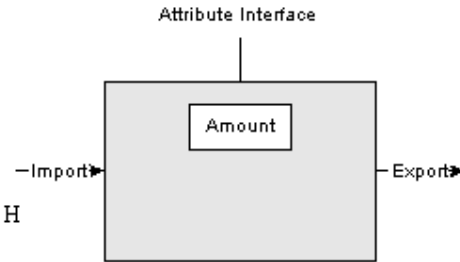


Figure 6: The Component Specification File

A component's attributes can have an open-ended number of parameters, including *address*, *size*, *depth*, *initial values*, and *writing source* (either hardware, software, or both).

The Component specification file is included via EP3 in the component HDL specification. EP3 automatically generates all of the attribute assignments and read statements and connects up the attribute interface. This has to be done in the actual HDL specification because synthesis tools require that all assignments to a given register occur in the same process block. Because the component author likely wants internal access to most of the created attributes, EP3 has to insert the system portion of the attributes in the same process block. This same Component specification file is ultimately parsed by the top level software to describe to the system the view of the component.

It should also be noted that all attribute addresses are relative to the component. Components are individually addressed by the attribute interface. In this manner, multiple instances of the same component can easily coexist with identical attribute addresses, but different component addresses.

5.4 Packet Definitions

While attributes may be changed asynchronously via the attribute interface, a mechanism must exist to provide synchronous changes as well. For example, one might want a tuner component to change frequency on a given sample. Additionally, control packets might need to proceed through the system to identify elements such as time code or restart commands, etc. Furthermore, as systems may require other elements than the FPGA (such as a DSP chip), the packet abstraction allows control to proceed among different device types without a significant control interface. Control packets are to be reserved for strictly synchronous activity (synchronous with the surrounding samples). Because of this synchrony requirement, it is imperative that there is never any reordering done between control packets and the surrounding data (a packet may, however, be consumed).

Therefore, as packets travel through components in route to a destination, it is each component's responsibility to ensure that the packet is reinserted into the correct output stream and at the correct sample point.

5.5 Component Integration Example

The component construction process is very similar to the node construction process described in Section 4.3 as a component is simply a special type of MEADE node. Consider construction of the *decimator* component from the *TFD* example. Entering the command: `meade component decimator` creates a MEADE node entitled *decimator*. In addition to the node's design and template files (which represent an incrementer by default), a standard component definition file is also copied into the node. This file can be edited to add or subtract any component attributes or portals.

In the case of the *decimator* component, the definition file would not have to be altered as the stock definition file has an input portal, an output portal, and a single attribute entitled *amount*. The `decimator.vhd` file would be edited to change the templates increment function to a decimate function. The portions of the template file that manage the attribute interface and portal FIFO instantiations would normally remain unaltered as they are auto-generated via EP3 directives..

The testbench template contains servers for the data portals as well as the attribute portal so that system level commands (portal writes/reads and attribute sets/gets) can be simulated easily in the testbench. While most of the testbench would be unaltered, the stimulus section of the testbench would be modified to make the appropriate attribute set/get calls and portal writes and reads.

Performing simulation or synthesis procedures on the component node is identical to the standard MEADE node. This process is simplified greatly by MEADE as the FIFO interconnects, attribute interfaces, and testbench modules are all automatically included as child nodes by MEADE without any intervention from the component node designer.

6 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. However, 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) [27]. A similar methodology was undertaken by JHDL [7] with its HWSysm 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* is specified in a *design* node (as opposed to a *component* node) with *abstract portals*. *Design* nodes represent complete designs that are 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. 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.

6.1 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.

6.1.1 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. The design deasserts the data ready flag when it cannot accept data. Whenever the valid and ready signals are asserted, data transitions occur across the portal. A streaming output portal is identical to a streaming input portal with the directions changed. Streaming portals connect directly to the streaming portals of a component (see Figure 7).

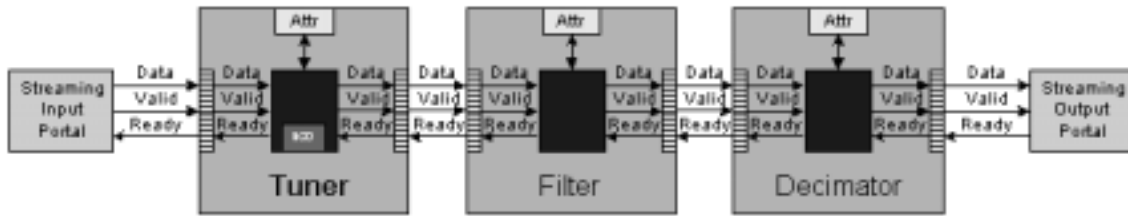


Figure 7: Streaming Portals

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.

6.1.2 The Memory Portal

There are different types of memory accesses that need to be accounted for: local memory, external memory, dedicated memory and an arbitrated memory, dual-port varieties, etc. All memory portals consist of data in, data out, address, read enable, write enable, and clock pins. We provide a group of portals that build on these common characteristics.

6.1.2.1 Local (on-chip) Memory

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 FPGAs 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.

6.1.2.2 Design External Memory

In the case of the 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 arbitrated 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 arbitrated or dedicated memories wherever appropriate.

One issue in the memory portal is the variable width of the memory port. By specifying a width on the portal, we will currently allow mapping to a memory implementation that is as wide as or wider than specified, padding the unused bits. This can result in an inefficient use of memory when the abstract width is specified as 8 bits and the actual memory is 32 bits wide. In this situation, it might be desirable to pack memory words into the larger memory, however, each memory write would have to be replaced by a read-modify-write, thus slowing memory access times. When the situation is reversed and the implementation memory is smaller than the abstract memory portal, the implementation will be forced to do address modifications and multiple read/write accesses for each memory access request.

This situation can be addressed intelligently in certain cases. Consider the case where four memories hold four separate arrays to be processed in a vector fashion. If the data is 8 bits wide, all of the memories can be implemented by one 32 bit wide memory that shares address control.

Future work will involve a more efficient method for handling dual port memories. Because a design cannot assume that a dual port memory will exist, an effective abstraction needs to be developed that can be mapped to a dual port memory if available, or to an arbitered model of the same memory if it is not dual ported. This is important mainly if an application expects to use the dual port memory as a single memory portal, with one port writing data and another port reading data. While this abstraction should be relatively simple to construct in a functional manner, the availability of a dual port memory may well affect the structure of the underlying algorithm. It is situations like this one that present the greatest challenges to our generic abstraction model. However, it is important to point out that the overall system was developed to be able to adapt to isolated instances such as this one. In a specific case, inserting a platform-specific module, while destroying portability, is straightforward. Furthermore, the detrimental effect on portability is localized to the specific module in question.

6.1.3 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.

6.2 The Design Specification File

Logic Foundry designs are constructed as MEADE nodes that contain a design specification file. The design specification 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.

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.

However, in the HDL testbench, all of the portals supply test models so that the design can be fully simulated as a platform independent design. Figure 8 shows a sample design specification file for the *TFD* design. In this design, data portals are created (named import and export). The components required are declared and then the components and the portals are connected. The attribute portals of the design and the components are automatically connected.

```
@attribute_portal
@data_portal in import
@data_portal out export

@component tuner t
@component filter f
@component decimator d

@connect import to t.import
@connect t.export to f.import
@connect f.export to d.import
@connect d.export to export
```

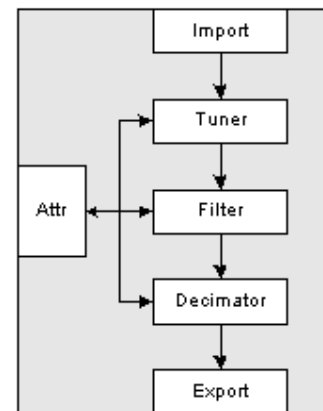


Figure 8: Design Specification File

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. If a Filter/Tune/Decimate (FTD) is desired rather than the *TFD*, the connection order is changed and the MEADE procedures can be rerun.

6.3 The Implementation specification file

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. Additionally, individual components of the design may be mapped to different FPGAs on a platform with multiple different FPGAs. For the purposes 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 [22].

The implementation specification file (see Figure 9) is included via EP3 in the implementation HDL specification. Essentially, the implementation HDL specification is a shell HDL template that is completely filled in as EP3 instantiates and interconnects all of the interfaces objects and the design core.

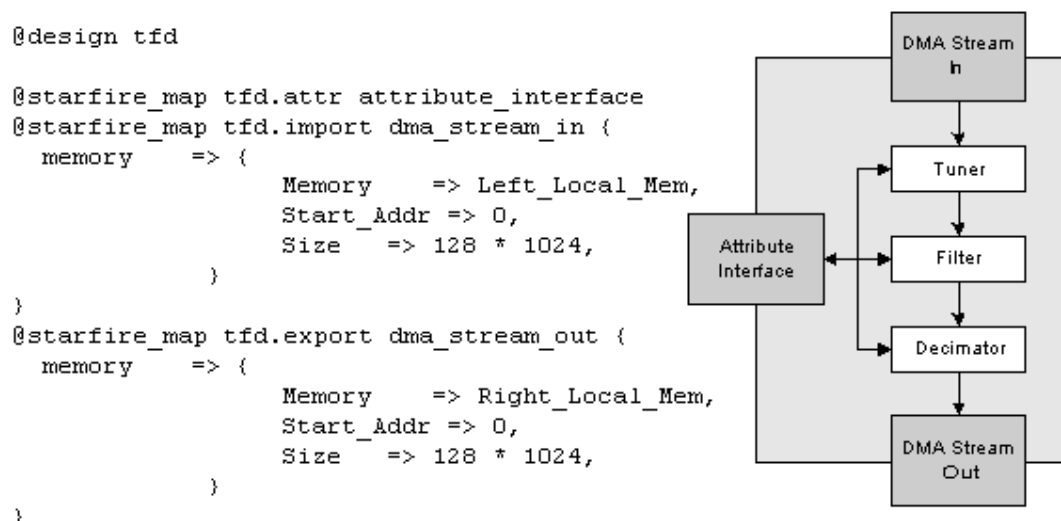


Figure 9: Implementation Specification File

In the implementation file, platform-dependent mappings (starfire_map represents a mapping call for the Annapolis MicroSystems Starfire board) map implementation-specific nodes to the design portals desired. In this example, a dma_stream_in node exists that performs the function of a stream_in portal on a Starfire board. This node has parameters that indicate which on-board memory to map to, the start address, and the size of the memory being used.

7 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. The DynamO model consists of a DynamO object, a DynamO API, DynamO front-ends, and DynamO back-ends. The DynamO object represents the entire back-end system to the front-end application with a hierarchical object that corresponds to the hierarchy of components and portals that were assembled in the Logic Foundry design. These elements are accessed via the DynamO API. The DynamO API represents the contract that DynamO back-ends and front-ends need to follow. The front-ends are plug-ins to higher level development environments like MATLAB, Python, Perl, and Midas 2k [18]. DynamO back-ends are wrappers around the board-specific drivers or emulators such as the Annapolis Starfire board [20].

7.1 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. In the case of the *TFD* example on the Annapolis *Starfire* board, a *DynamO Starfire* back-end creates a DynamO object with a top level system component *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 10).

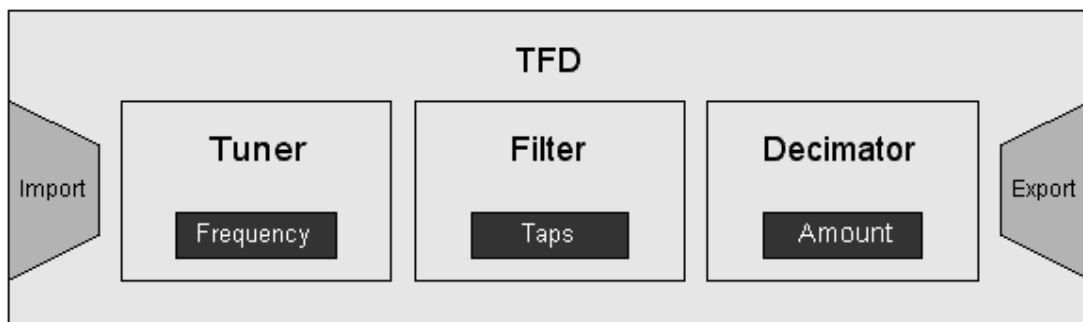


Figure 10: The DynamO Object

Along with the objects, the DynamO 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.

7.2 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 11).

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.

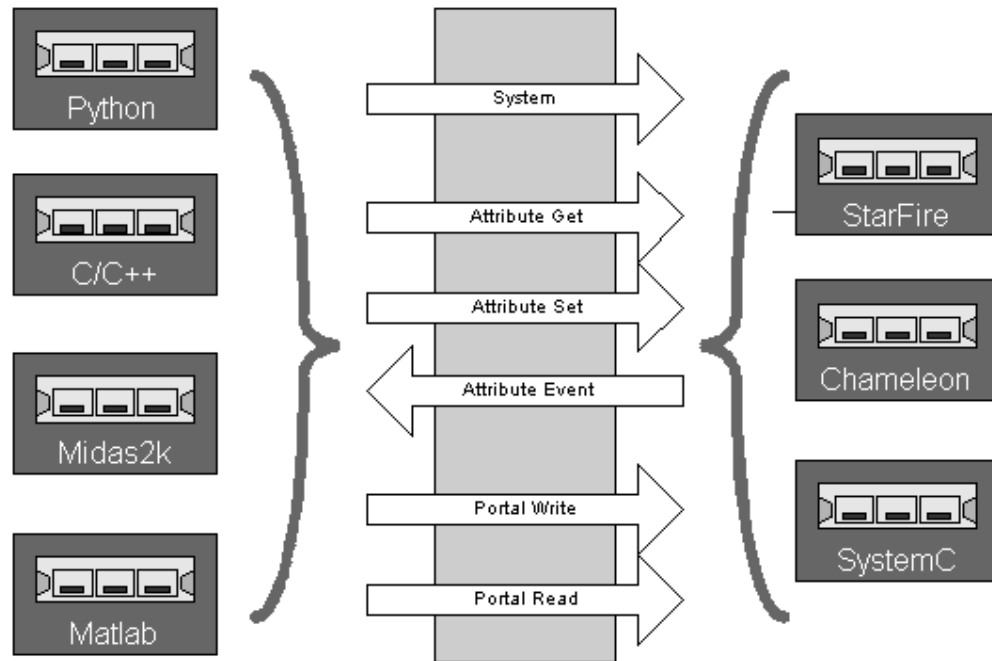


Figure 11: The DynamO API

7.3 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 [6]. 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.

7.4 DynamO Front-Ends

A DynamO front-end is possible for many software application 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 already implemented a DynamO front-end for C++, Python, Rincon Research's Midas2k DSP environment and are planning on a MATLAB front-end.

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 12 demonstrates how a Python application script would interact with the DynamO API and the DynamO object in the *TFD* design example. Note that there is absolutely no evidence of implementation-specific details such as register addresses or communication protocols.

<pre># 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</pre>	<pre># 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()</pre>
---	---

Figure 12: Python DynamO Example

8 Experimental Results

We have developed the Logic Foundry including all of the major building blocks described – VHDL implementations for MEADE, EP3, attribute interfaces, component abstractions and interface portals, the get/set and data write/read portions of the DynamO API, DynamO back-ends for the 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 Turbo Decoder.

8.1 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.

To create the different designs, we used a simple Perl script to generate the design specification files that instantiate and connect the components. All of the simulation, synthesis, and place and route steps were performed using simple MEADE commands.

8.2 TFD Design

In order to create the *TFD* design, we began with an *NCO* element created by using the Xilinx Coregen tool. This resulted in a Xilinx-specific structural VHDL file. To contain this file, an *NCO* node was created in MEADE. An element object was created for the Xilinx VHDL file and the node object was edited to contain this element. This node was then wrapped by a tuner. To accomplish this, a new component node was created and its element object was edited to point to the *NCO* node as child node. The component definition file was edited to contain a phase increment attribute for the *NCO* (rather than a frequency attribute), and then the component VHDL files were edited to instantiate the *NCO*, provide portal connectivity and control, and perform the tuner function by multiplying the *NCO* outputs with the tuner's inputs. These files, in their preprocessed state, contain approximately 50 lines of user-edited VHDL code for component functionality and 100 lines of total code. The attribute interfaces and portal FIFO instantiations are all automatically included via the preprocessor. The processed VHDL files contain approximately 240 lines of VHDL code. The number of automated lines of code will vary depending on the number of attributes in the component. This same process was performed to create the *filter* component (using an fir VHDL file created by the Xilinx Coregen tool). The *decimator* component was created with a few simple additions to the default MEADE template.

When all three components were completed, a design specification file was created (see Figure 8) as well as an implementation specification file (see Figure 9). These files were used by MEADE to create a complete implementation for the FPGA, and system controlling software via the DyanmO interfaces.

In order to test the ease of component reuse in the Logic Foundry, we opted to create a FTD system out of the *TFD* system components. To accomplish this task, the design specification file was altered to change the connection order of the three components. MEADE was used to recreate the design using the same implementation specification file. DyanmO front-end objects were automatically changed due to the change in the design specification file. In both cases, control timing was achieved and system timing was limited by the speed of the tuner component (see Table 1).

8.3 The Turbo Decoder Design

Our final design was to take a large design (several thousand lines of VHDL code) written outside of MEADE, but with a view to fitting into the Logic Foundry attribute/portal design structure. When the design was completed, it was wrapped with a MEADE component in the same manner as the *tuner* described above. The Turbo Decoder 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 6.1.3.

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. To accomplish this, we hand edited the implementation VHDL file and did not auto generate it from the implementation specification file.

This example showed that we could take a design and use some of the Logic Foundry without requiring tight integration into the whole of the Logic Foundry. Software integration was done automatically, and the component architecture of the Logic Foundry was used, but the top level FPGA implementation specification and Turbo Decoder design and test was performed outside of the Foundry.

8.4 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	FlipFlops	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
TurboDecoder	67.290	39.787	17031	5600	27

Table 1: Summary of Designs

9 Conclusion

We have introduced the Logic Foundry and shown how this approach allows for the rapid prototyping and deployment of FPGA-based systems. By using MEADE and EP3, a design engineer can rapidly create and operate on design components. These components can be entered in a library for efficient reuse by system designers. By using MEADE and EP3, system engineers can create an FPGA image file without requiring the specialized design tool knowledge of a logic designer. 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. Finally, 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.

While the entire Logic Foundry fits together as a whole, it is important to recognize that each piece can be used independently. One design house may simply want to use MEADE and EP3 as automation tools for FPGA construction, where another design house may want to use the DynamO aspects without requiring any of the FPGA construction mechanisms.

Our future work will focus on the complete implementation of the 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.

10 References

- [1] R.T. Olay, "Xilinx XtremeDSP Initiative Meets the Demand for Extreme Performance and Flexibility," *Xcell Journal*, Issue 40, Summer 2001, Xilinx, pp. 30-33.
- [2] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, pp. 9-25, Springer-Verlag, 2001
- [3] 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.

- [4] <http://www.synopsys.com/C-level.html>
- [5] OXFORD Hardware Compilation Group, The Handel language, Technical Report, Oxford University 1997.
- [6] J. Gerlach and W. Rosenstiel, "System Level Design Using the SystemC Modeling Platform",
<http://www.systemc.org/papers/sda-2000.pdf>.
- [7] 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.
- [8] 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.
- [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] R. Rajsuman, *System-on-a-Chip: Design and Test*, pp.33-55, Artech House, 2000.
- [17] M. Keating and P. Bricaud, *Reuse Methodology Manual: For System-On-A-Chip Designs*, 2nd Edition, Kluwer Academic Publishers, 1999.

- [18] Rincon Research Corporation, Introduction to Midas 2k, Tucson, AZ, Jan 2000.
- [19] R. Andraka, "High Performance Digital Down-Converters for FPGAs", Xcell Journal, Issue 38, Winter 2000, Xilinx, pp. 48-51.
- [20] Annapolis Micro Systems Incorporated, , WILDSTAR Reference Manual, rev 3.1, Annapolis, MD, 2000.
- [21] A. Kalavade and E. Lee, "A global critically/local phase driven algorithm for the constrained hardware/software partitioning problem", Proceedings of the International Workshop on Hardware/Software Co-Design, pages 42-48, 1994.
- [22] S. S. Bhattacharyya, "Hardware/software co-synthesis of DSP systems", In Y. H. Hu, editor, *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, pages 333-378. Marcel Dekker, 2002.
- [23] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow", *Proceedings of the IEEE*, 75(9), pp. 1235-1245, Sep. 1987.
- [24] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow", *IEEE Transactions on Signal Processing*, 44(2), pp. 397-408, Feb. 1996.
- [25] J. Tsay, C. Hylands, and E.A. Lee, "A code-generation framework for Java component-based designs", *Proceedings of the International Conference on Compilers, Architecture, and Synthesis of Embedded Systems*, Nov. 2000.
- [26] R. Lauwereins, M. Engels, M. Adé, J. Peperstraete, "Grape-II: A System-Level Prototyping Environment for DSP Applications", *IEEE Computer*, Vol. 28, No. 2, pp. 35-43, Feb. 1995.
- [27] 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
- [28] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler, "DEFACTO: A Design Environment for Adaptive Computing Technology", IPPS/SPDP Workshops, 1999.
- [29] B. Schott, S. Crago, R. Parker, L. Carter, C. Chen, J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, "Reconfigurable Architectures for System-Level Applications of Adaptive Computing," submitted to VLSI Design special issue on reconfigurable computing.
- [30] C. Matsumoto, EE Times.com, February 14, 2000. Monterey, CA. Coverage of the FPGA2000 Panel discussion evaluating the ability of CAE tools for FPGAs.

- [31] 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.
- [32] G. Spivey, "MEADE, A Modular Extensible Adaptable Design Environment for ASIC and FPGA Development", *Perl Conference 3.0*, San Jose, CA, July 1999.
- [33] D. Conway, *Object Oriented Perl*, pp. 387-428, Manning, 1999.
- [34] B. Cohen, *Component Design By Example*, pp. 129-228, VhdlCohen Publishing, 2001.
- [35] G. Spivey, "EP3: An Extensible Perl PreProcessor," *Proc. of the International Verilog HDL Conference and VHDL International Users Forum*, Santa Clara, CA, pp. 106-113, March 1998.
- [36] S. Sriram and S.S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, 2000.