Advances in Architectures and Tools for FPGAs and their Impact on the Design of Complex Systems for Particle Physics

Anthony Gregerson[a], Amin Farmahini-Farahani[a], William Plishker[b], Zaipeng Xie[a],

Katherine Compton[a], Shuvra Bhattacharyya[b], Michael Schulte[a]

[a] *University of Wisconsin - Madison*

[b] *University of Maryland - College Park*

{agregerson, farmahinifar, zxie2}@wisc.edu
{plishker, ssb}@umd.edu {compton, schulte}@engr.wisc.edu

Abstract

The continual improvement of semiconductor technology has provided rapid advancements in device frequency and density. Designers of electronics systems for high-energy physics (HEP) have benefited from these advancements, transitioning many designs from fixed-function ASICs to more flexible FPGA-based platforms. Today's FPGA devices provide a significantly higher amount of resources than those available during the initial Large Hadron Collider design phase. To take advantage of the capabilities of future FPGAs in the next generation of HEP experiments, designers must not only anticipate further improvements in FPGA hardware, but must also adopt design tools and methodologies that can scale along with that hardware. In this paper, we outline the major trends in FPGA hardware, describe the design challenges these trends will present to developers of HEP electronics, and discuss a range of techniques that can be adopted to overcome these challenges.

I. INTRODUCTION

High-energy physics systems have a history of pushing the boundaries of technology. The electronics in HEP systems often require extremely high bandwidth and computational throughput, precise timing, and tight real-time processing constraints. These stringent performance specifications historically demanded the use of custom ASIC solutions [2], because in the past, programmable hardware such as FPGAs were inadequate to the task. Although ASICs are capable of achieving the highest possible performance, they suffer from two major shortcomings for HEP applications. First, they are very expensive to produce in low volumes because the costs of fabrication are not well-amortized. Second, they are rigid, fixed-function devices that offer very limited flexibility for adjustment to new experimental parameters or algorithms. Early designers were forced to cope with these shortcomings, as ASICs were the only technology capable of meeting key performance requirements of HEP systems. However, as time has passed, continual advancements in the semiconductor industry have produced major improvements in the density and speed of electronics. Consequently, FPGAs have also improved in capacity and performance. Modern FPGAs are able to achieve performance levels suitable for many HEP applications and provide attractive properties such as re-programmability and smaller low-volume costs. The result of

these trends has been a rapid adoption of FPGAs in HEP electronics. A large proportion of the electronics in the Compact Muon Solenoid Level-1 Trigger, for example, are based on FPGAs, and many of the remaining ASICs are scheduled to be replaced with FPGAs in proposed upgrades [1].

Improvements in FPGA technology are not likely to end soon. Today's high-density FPGAs are based on a 40-nm silicon process and already contain an order of magnitude more logic than the FPGAs available at planning stage of the Large Hadron Collider's electronics. 32 and 22 nm silicon process technologies have already been demonstrated to be feasible; as FPGAs migrate to these improved technologies their logic density and performance will continue to increase. With the next generation of HEP designs, the question has changed from *'When will programmable hardware be good enough to meet our needs?'* to *'How can we take maximum advantage of the advancing density and performance of programmable hardware in our designs?'* The answer to this question is not as simple as it may seem. Faster, higher-density devices may enable more complex algorithms, greater functionality, and higher-resolution processing—but only if the methods of designing, testing, implementing, and verifying these systems adapt to meet the needs of these new levels of complexity. As devices continue to improve, the importance of using the right combination of tools and methodologies to enhance developer productivity and create maintainable designs will become increasingly critical. Relying solely on established hardware design languages (HDLs) may not be sufficient to meet the challenges of future system design. In this paper, we examine recent trends in FPGAs and the implication these trends have on the adoption of new software tools, techniques, and methods for the design of HEP systems based on future generations of FPGAs.

The rest of this paper is organized as follows. In Section II, we cover the major trends in FPGA hardware. In Section III, we describe the problem of managing increased design complexity and describe a series of tools and techniques that can be used to create scalable design processes. In Section IV, we describe the effects of FPGA trends on the problem of hardware verification and debugging and present tools and techniques for managing this problem. Finally, in Section V, we provide our conclusions about the impacts of FPGA trends on the future of electronics design for high-energy physics applications.

II. FPGA HARDWARE TRENDS

We divide FPGA hardware trends into two different categories: trends in performance and trends in resource capacity. In this section we examine performance and resource capacity trends for high-end FPGAs from Xilinx over the past ten years [34, 35, 36, 37, 38, 39, 40]. HEP applications often rely on cutting-edge technology to meet their stringent requirements. Therefore, we will present composite data based on the largest and highest-performance device available from either vendor at a given point in time.

A. Performance

There are two aspects of FPGA performance that have a strong impact on HEP designs, maximum operating frequency and I/O bandwidth.

Operating frequency is directly related to the computational capabilities of a device. Higher frequencies allow calculations to be completed faster. This is important because computational latency is one of the key constraints of many HEP designs. A graph of the maximum frequency and silicon process technology of high-end commercial FPGAs is shown in Fig. 1. Frequency has scaled linearly with time. It is also notable that at 600 MHz, modern FPGAs are still operating at relatively low frequencies compared to high-end ASIC-based chips, such as microprocessors. Whereas ASICs have experienced many challenges in continuing to scale their frequency up, such as power density concerns and pipeline scaling, FPGAs still have some headroom before they encounter these problems. As indicated in the graph, frequency is closely related to the silicon process size used to manufacture devices. Smaller processes can produce transistors with lower latencies (and correspondingly higher frequencies). As of 2009, high-end FPGAs are being manufactured on a 40-nm silicon process. Intel has already demonstrated viable 32-nm and 22-nm processes [3]. Therefore, we expect that FPGA frequencies will continue to follow increasing trends through the near future.
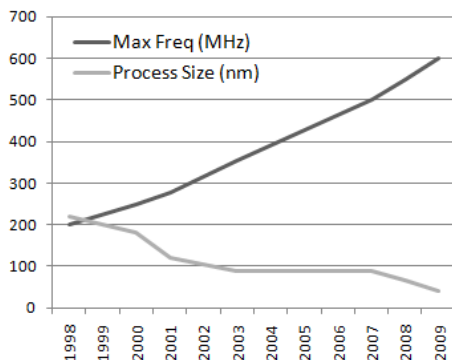

Figure 1: Frequency and CMOS process size trends for high-end commercial FPGAs.

A second key performance parameter of FPGA devices is their total I/O bandwidth. One of the key distinguishing characteristics of particle physics applications are the tremendous data rates produced by HEP experiments. The electronics involved in triggering, data acquisition, compression, and other real-time data processing need very high bandwidth to handle copious amounts of experimental data. Often, the amount of data that can be processed by each device in these systems is limited by device bandwidth rather than by logic resources or computational speed. Such systems require many duplicate devices to handle all the data.
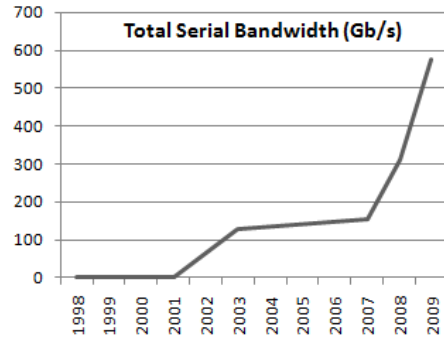

Figure 2: Total serial I/O bandwidth trends for high-end commercial FPGAs.

As discussed later in Section II.B, although the total number of I/O pins on FPGAs has not experienced significant growth in recent years, total device bandwidth has rapidly improved due to the introduction of high-speed serial transceivers. Fig. 2 shows the trend in total serial I/O bandwidth over the past decade. I/O bandwidth has managed to maintain an exponential growth rate in recent years, allowing it to keep pace with the growth of logic resources (see Section II.B). The matching growth of bandwidth and logic is a key trend for HEP system designers. If FPGAs continue this trend in the future, devices will maintain consistent resource ratios, making it easier to consolidate distributed, multi-device systems into a smaller number of devices. If, on the other hand, bandwidth growth falls behind logic growth, designers will need to consider ways they can use extra logic to improve the quality of their systems. This might mean increasing the precision of computations or adding redundant error correction. Both Xilinx and Altera have recently introduced 11-Gb/s transceivers, but have not yet integrated these transceivers on all serial I/O pins. Moreover, the number of pins dedicated to high-speed serial I/O could be increased; serial I/O is currently only available on a small fraction of available pins. Therefore, it is feasible for total I/O bandwidth to continue to grow.

B. Resource Capacity

In addition to FPGA performance, resource capacity is also a major concern. The quantity of logic resources available to developers may determine the amount of functionality and precision of computation that can be incorporated into each device. Unlike the monolithic silicon wafers used to implement ASIC designs, modern FPGAs have a heterogeneous design substrate. They include look-up-tables (LUTs), flip-flops, high-density block RAM (BRAM), optimized multiply and accumulate chains (DSP blocks). Since FPGAs are packaged chips,

it is also worthwhile to consider the total number of I/O pins available. Graphs of the growth of each of these resource types, normalized to the resource capacity of one of the highest capacity FPGAs from 1998, are shown in Fig. 3. Note that the DSP blocks were not introduced into Xilinx FPGAs until 2001, so multiplier growth is normalized to this later device.
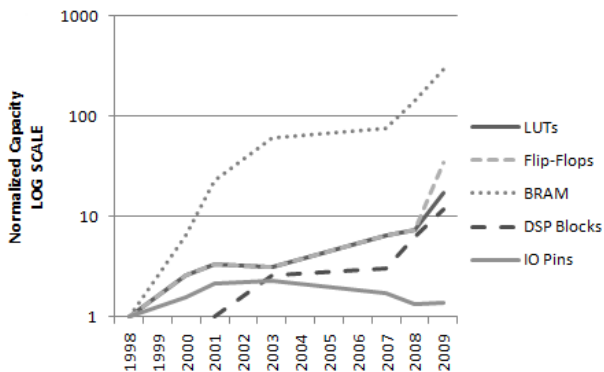

Figure 3: Resource capacity trends for high-end commercial FPGAs, shown in on a logarithmic scale.

There are several key relationships to be observed from these trends. All logic resources (LUTs, flip-flops, BRAM, and DSP blocks) have exhibited exponential growth. This can be attributed to advancements in silicon process technology, and thus is likely to continue in the near future. In particular, sequential state (BRAM and flip-flops) makes up a larger percentage of the total logic resources. The total number of I/O pins, however, has not shown sustained growth due to physical limitations. The package size of the device, the number of pins that can fit in that space, and the feasibility of board-level routing for those pins are significant limiting factors. In fact, pin count has shown a downward trend in recent years; I/O bandwidth has only managed to increase due to the introduction of high-speed serial transceivers on a subset of the remaining pins.

Of most importance are the ways in which these trends interact to impact the design process. As the amount of logic and bandwidth to each device increases exponentially, the size and complexity of designs possible on a single device increases dramatically. We discuss design methods and software advances that can be used to manage this challenge in Section III. Logic is growing at a much faster rate than the number of I/O pins. The rapidly increasing ratio of device state to I/O pins will make it more difficult to rely on the use of external logic analyzers to perform hardware verification and debugging for complex sequential circuits. We discuss FPGA-centric verification and debugging tools in Section IV.

III. Design Complexity

New FPGA generations will continue to bring increases in device resources and performance. Future architectures will have significantly more logic and bandwidth available on each chip than what is available today. Designers can leverage these improvements to enable higher levels of system integration,

more nuanced algorithms, robust error correction and reliability, and higher-resolution processing. However, these design enhancements come at a price; as designs become larger and more capable, they also become more complicated. If it already takes several person-months to properly design, simulate, debug, and verify the firmware of an FPGA with tens to hundreds of thousands of logic cells, how long will it take to do the same for FPGAs with tens of millions of cells?

Before designers can take advantage of larger devices, they must ensure that they can meet three main objectives. First, we must make sure we can control the design costs. The logic density of FPGAs may double every few years, but the budgets of scientific research organizations do not. Second, to maintain a reasonable pace of advancement of HEP systems, the design time for these circuits cannot simply increase proportionally with the growth of logic capacity. Third, ensure the collection of valid scientific results and protect the operation of critical experimental systems, the number of bugs and defects in these circuits must be held to a very low level. To achieve these three objectives, we must increase the productivity and effectiveness of the design and testing of systems. In some cases this can be achieved by adopting new software tools and technologies. In other cases it may mean that developers must transition from ad hoc design practices to more formal and rigorous methodologies.

In this section we cover three concepts for increasing productivity for complex designs: collaborative techniques, scalable design methodology, and high-level-language tools.

A. Collaborative Design

One approach to managing larger, more complex designs is to tap into a larger pool of design talent and expertise. On a global system-wide scale, HEP projects already rely on large-scale collaborative efforts from many research and design groups. However, collaboration can also employed at the level of individual designs. This sort of collaboration can be implemented on varying scales.

On a small scale, each design group could institute a policy of seeking peer review of their work to ensure that it is of the highest quality. Presenting design decisions for external review not only provides the benefit of outside expert experience and insight, but also helps the group to systematically explore, justify, and document their design choices. Such a review could be regularly applied at multiple levels, including specifications, major design decisions, and actual firmware code.

On a larger scale, related groups within HEP projects could implement infrastructure for sharing firmware code with each other. Although the LHC collaboration has access to a spectacular range of expertise from numerous universities and labs, teams often work in isolation until it is time to begin integrating their systems. Although each group has its own set of goals, constraints, and platforms, it is reasonable to expect that some design work could be shared between teams. For example, many systems may need to decompress zero-suppressed data transmissions, calculate parity and apply error correction, or sort sets of data. If many groups replicate the same design process needed to implement these functions, time and money are being

wasted.

On the largest scale, firmware source code published and made open to public scrutiny after the initial internal design. Studies suggest that the average defect rate for open source software is significantly lower than that of proprietary software [5]. It may be possible to achieve similar improvements with open source firmware. Moreover, opening the code up to the public for comment could allow review by hundreds of external designers at very low cost.

B. Methodology

One of the most crucial components to managing complex projects is adherence to a structured design methodology. The topic of design methodology is too broad to be thoroughly covered in a single paper. Therefore, rather than attempting to provide an exhaustive summary, we focus on a few concepts that are particularly useful to the design of complex digital systems for HEP applications.

1) Specification

The first step in the design of any reasonably large system is the development of the design specifications. The specifications include the performance requirements of the design — which may include aspects such as latency, throughput, I/O bandwidth, error correction capabilities, and other factors — a description of the algorithms to be implemented, and input and output data formats. At a higher level, the specifications may also include factors such as the monetary and time budgets. Development of a robust and well-documented set of specifications for each major portion of a design should be performed early in the design process, possibly before the first line of firmware code is written. Clear and early communication of these requirements helps to avoid the errors and incompatibilities that arise when teams work from a set of incomplete or unclear specifications. Moreover, the development of the specifications may itself yield insight into the strategies to take during the firmware development process, guide resources to the most challenging aspects of the design, and uncover potential problems before a major engineering investment has been made.

For systems that implement physics algorithms, such as trigger systems, the specifications for the physics and electronics components are typically kept separate. For example, the algorithms are developed in order to meet the physics requirements and verified via simulation and mathematical models. Then, these algorithms are used to develop specifications for the electronics. This methodology makes sense when considering that the physics performance is the first concern of an experiment. However, the problem with this method of developing electronics specifications is that it may constrain the ability of engineers to evaluate alternate designs. For example, making slight alterations to a triggering algorithm might have minimal impact on the triggering efficiency (the physics) but yield major savings in the complexity of the hardware (the electronics). With limited budgets and more stringent hardware requirements, it may become prudent to view the development of the hardware systems that support the experiments as a first-class concern. Efforts should be made to integrate the physics and electronics specifi-

cations and form multi-disciplinary teams to evaluate the impact of algorithmic modifications in both the physics and electronics domains.

2) Design Practices

When designing firmware code for complex systems, there are a variety of techniques that can be used to help manage large projects. One of the most basic of these is the concept of modular design. Modular design uses a 'divide and conquer' approach to break up big projects into smaller parts that are easier to design, test, and verify. Systems are partitioned into a group of interconnected modules that each implement a basic function, and these can be combined (perhaps hierarchically) into the required larger structure. Ideally the system should be partitioned in such a way that modules have few interdependencies and each module's function can be analyzed and understood in isolation. Furthermore, modularity provides the benefit of module reuse. For example, a 32-bit-wide adder module could be hierarchically designed, composed of several 8-bit-wide adder modules.

Modular design offers several important advantages over monolithic design. Building a system up by starting with smaller modules allows the developer to test and debug the firmware code in small pieces. This makes it easier to identify and isolate bugs in the code. Modular design also allows developers to perform synthesis on basic computational modules and obtain early performance estimates to guide later development. Building up a library of modules that implement basic functions also allows code re-use, avoiding duplicate coding work and reducing the testing burden. Such modules could also be shared across different projects using a collaborative firmware repository as described in Section III.A.

For the development of HEP systems, it may be beneficial to use parameterization to further increase design re-use beyond what would be possible with modularity alone. Parameterization is a powerful construct available in all major HDLs. It allows a designer to use computations on constants to determine design features such as the size of registers and width of buses. Modifying paramterized features requires simply changing the parameter value in the code, then re-compiling the HDL code into a new (modified) hardware structure. By parameterizing modules, one could, for example, use the same parameterized adder code to create multiple adders of different bit-widths without having to alter the firmware, potentially introducing new bugs. Parameters can also be used to rapidly explore the impact of different design decisions. For example a developer could study the effect that varying the precision of a multiplication unit has on its maximum frequency.

When parameterized modules are combined with code generation constructs available in HDLs, they give designers a powerful tool for exploring large-scope considerations, such as the number of design units that can fit on a given FPGA model. This analysis can be especially useful in large-scale HEP systems where a design may need to be partitioned across multiple devices. The use of fully-parameterized designs enables a rapid evaluation of various partitioning schemes and the ability to gauge the tradeoffs of using different models. It also allows the HDL code to be quickly adapted to different FPGAs; this can be a very valuable trait in HEP designs where the long develop-

ment process may mean that the target device is not finalized until well into the development cycle. Moreover, it allows the design to be gracefully adapted to larger FPGAs if the hardware is upgraded in the future.

3) Firmware/Emulator Co-design

Designing firmware is generally a more time-consuming process than writing software using a high-level language. As such, it is common practice to first create an emulator for a HEP hardware system in software, use it to explore and test new algorithms, then design the hardware to match the function of the emulator. This approach is effective for rapidly testing algorithmic changes, but often leaves a large implementation gap between the emulator and the hardware. Algorithms that are easy to implement and achieve high performance in software do not necessarily share those properties in hardware. This may lead the software designers to describe algorithms that are very difficult for the hardware designers to implement efficiently. Also, the high-level code that implements the emulator may have a much different structure and interface than the HDL code that implements the firmware, making it difficult to share the same testing infrastructure between the two.

In the future, it may be advantageous to move to a methodology that focuses on firmware/emulator co-design rather than a sequential process of creating the emulator and then creating the firmware or vice versa. The concept of co-design is to allow systems to be developed in tandem, allowing rapid transmission of feedback and implementation of changes. Better communication between firmware designers and algorithm developers should lead to the adoption of algorithms that both meet the needs of the experiment and are well-suited to hardware. Moreover, a co-design process would encourage the use of similar structural hierarchies in the emulator and firmware. This would allow the use of a unified test framework, making it much easier to pinpoint bugs in either implementation.

One of the most important aspects of a co-design methodology is to ensure that the speed of firmware design does not impede the software design. Therefore, rather than directly going between a high-level language, such as C/C++, and an HDL, it may be beneficial to use a hardware verification language (HVL) such as SystemVerilog or SystemC [7, 8] to help bridge the gap.

4) Testing Practices

As projects migrate functionality from high-level languages to firmware implementations or collaboration begins via a co-design methodology, a wide gap separates hardware system and software emulator design approaches. Each has their own programming models and development environments. The original software application description can range from general imperative languages like C, to object-oriented languages like C++ or Java, to domain-specific approaches like MATLAB. Firmware may be developed in SystemVerilog or SystemC during the early design exploration phase and in VHDL or Verilog in the implementation phase. This multitude of languages and programming environments makes the design time lengthy and error prone, as developers must often manually transcode between different languages and environments. Many 'best practices' are utilized in industrial and academic environments to help this process, such as automatically generating documentation (e.g. Javadoc), auto-configuration, adherence to interface specifications, and unit testing.

In particular, unit testing facilitates productive design by integrating testing early into the design flow to catch erroneous or unexpected module behavior earlier in the design cycle, when it is cheaper and easier to alter the design or specifications. Such techniques have proven effective for many languages and platforms, but for design projects that involve transcoding and retooling for the final implementation, existing tools still leave many manual, error-prone steps in the process. This leads to longer design times with lower-quality implementations.

Typically when software designers employ unit testing, they use frameworks that are language-specific (e.g. see [9]). More than just a syntactic customization, such frameworks are often tied to fundamental constructs of the language, such as checking that methods exhibits the proper form of polymorphism in an object-oriented language. Furthermore, these language-specific approaches work well when designers are using only a single language or a single platform for both development and final implementation. But when designers must move between languages with different constructs (such as when moving between an emulator coded in C++ and firmware written in VHDL), the existing tests must be rewritten. This consumes extra design time and creates a new verification challenge to ensure that the corresponding unit tests between these two languages are, in fact, performing the same test.

A new testing approach is needed that is language and platform agnostic. Such an approach is possible by leveraging model-based design for projects that integrate heterogeneous programming languages and by applying and integrating different kinds of design and testing methodologies. With model-based development, automatic testbench creation is possible, improving the ease with which designers can create cross-platform tests.

One tool that has been developed to manage this aspect of the design process is the *DSPCAD Integrative Command Line Environment* (DICE) [10]. It provides a framework for facilitating efficient management of the test and development of cross-platform design projects. In order to accommodate cross-platform operation, the DICE engine provides a collection of utilities implemented as bash scripts, C programs, and python scripts. By using free and open source command-line interfaces and languages, DICE is able to operate on different platforms, such as Windows (equipped with Cygwin), Solaris, and Linux.

5) Design Verification

To improve the quality and performance of hardware designs while reducing their development time, a cross-platform design environment is needed that accommodates both early design exploration and final implementation tuning. One could make effective use of the initial higher-level application specification to create a functionally-accurate, language-independent design model. This model could be used in the development and validation of both the emulator and hardware.

The Dataflow Interchange Format (DIF) is tool for model-based design and implementation of signal processing systems using dataflow graphs [13, 14]. A designer starts by translating the high-level design specification into a platform-independent description of the application in the DIF format. This structured, formal application description is an ideal starting point for capturing concurrency and optimizing and analyzing the application. Because the application description in DIF exposes communication as a first-class citizen, DIF descriptions are suitable for targeting hardware design, where modules must be interconnected by wires. After creating the initial DIF description, a designer can use it to perform side-by-side development and validation of optimized hardware or software implementations. One of the main advantages of using the DIF format is that it is dataflow-based description that allows the use of sophisticated analysis techniques that have been developed for dataflow languages.

A formal model such as dataflow can improve the test quality and provide information and tools that can be used to optimize a design. Dataflow models have proven invaluable for application areas such as digital signal processing. Their graph-based formalisms allow designers to describe applications in a natural yet semantically-rigorous way. Such a semantic foundation has permitted the development of a variety of analysis tools, including tools for balancing input and output buffers and for efficiently scheduling multiplexed operations [11]. As a result, dataflow languages are increasingly popular. Their diversity, portability, and intuitive appeal have extended them into many application areas and target platforms.

A typical approach involves specifying the application in DIF. Such an application specification typically defines the underlying modules and subsystems, along with their interfaces and connections. This specification is complete in terms of ensuring a correct functional behavior and module interfaces. The DICE framework can be applied to test each of the individual modules for its correctness, or extended to a larger subsystem or the entire application.

Any transcoding or platform-specific enhancements are accommodated by DICE via its standardized build and test framework. This allows designers to utilize the same testing framework at inception as they do at final implementation. Software developed jointly with DIF and DICE uses a single, cross-platform framework to handle design validation throughout each phase of development. The amount of time required to perform validation can be reduced through the direct reuse of unit tests in DICE. Model-based development can allow automatic testbench creation, improving the ease with which designers can create cross-platform tests.

C. High-Level-Language Tools

Several tools have been developed to enable designers to specify algorithms using high-level languages and/or graphical user interfaces and automatically map those algorithms into an HDL. The resulting HDL code can be simulated to ensure correct performance and synthesized, placed, and routed to produce an ASIC or FPGA implementation. These tools facilitate rapid design-space exploration and for certain classes of algorithms lead to efficient implementations. In addition to generating HDL, several of these tools also generate testbenches, hardware interfaces, and synthesis scripts. However, the HDL produced by these tools is often difficult to read and debug. Furthermore, for certain tools and algorithms, the original high-level language code may require significant modifications to yield acceptable results and various high-level language constructs cannot be converted to synthesizable HDL. With some tools, the generated HDL instantiates components that are specific to a particular FPGA family, which can make it difficult to port to other platforms.

1) C-to-HDL Tools

Numerous companies and universities have developed tools that convert C code to Verilog or VHDL. These tools typically take a program written in C, along with a set of design constraints or guidelines, and produce functionally-equivalent Verilog or VHDL. They may also produce accompanying C code (if not all of the original C code is meant to be synthesized), testbenches, synthesis and place-and-route scripts, and interfaces to the resulting hardware designs. With many of these tools, only a subset of the C language is supported, since constructs such as library calls, dynamic memory allocation, function pointers, complex data structures, and recursive functions cannot be easily implemented using synthesizable HDL code. Some of these tools provide extensions to the C language to allow the designer to specify operand lengths, hardware interfaces, timing-related information, and the desired level of parallelism in the resulting HDL. In the remainder of this section, we provide several examples of C-to-HDL conversion tools and then discuss their strengths and weaknesses.

The Impulse CoDeveloper Toolset from Impulse Accelerated Technologies provides a C-based development framework for FPGA-based systems. It includes the CoDeveloper C-to-FPGA Tools, the CoValidator Test Bench Generator, and the CoDeveloper Platform Support Packages [15, 16] . Collectively, these tools allow designers to (1) specify their hardware designs with Impulse-C, which supports a subset of C plus some extensions, (2) profile their Impulse-C code to determine potential performance bottlenecks, (3) if desired, partition the code such that certain code sections are run on an FPGA and other portions are run on a programmable processor, (4) use interactive, graphical tools to specify design constraints and perform optimizations, (5) map selected Impulse-C code into either VHDL or Verilog, (6) generate hardware interfaces for specific FPGA platforms, and (7) create HDL testbenches and simulation scripts to test the resulting designs. The Impulse CoDeveloper Toolset can be used to generate either standalone hardware designs or hardware design that interface with an embedded or external processor. They also provide several optimizations to improve hardware efficiency and parallelism including common sub-expression elimination, constant folding, loop pipelining, and loop unrolling. The Impulse CoDeveloper Toolset has been used to develop FPGA-based solutions for a wide range of applications including image and video processing, security, digital signal processing, and scientific and financial computing.

Pico Express FPGA from Synfora takes an algorithm written using a subset of the C programming language and a set of de-

sign requirements, such as clock frequency and target throughput, and creates register transfer level (RTL) and SystemC implementation models [17]. It also generates testbenches and an application driver program. PICO Express FPGA includes design space exploration capabilities that, based on user-specified design parameters, create multiple implementations and provide FPGA resource and performance estimates for these implementations to allow design tradeoffs to be evaluated. To achieve efficient designs and provide accurate performance and resource estimates, PICO Express FPGA utilizes several device-independent optimizations and also optimizes the resulting RTL for a particular Xilinx FPGA family. PICO Express FPGA has been used to design FGPA-based hardware for a wide range of systems including video, audio, and image processing, wireless communication, and security.

The C2R Compiler from Cebatech provides an automated mechanism for converting structured C source code, along with a small set of compiler directives, to Verilog and SystemC [19, 20]. Internally, the C2R Compiler creates a control dataflow graph and then uses allocation and scheduling algorithms to produce Verilog that is functionally equivalent to the C source code. Consequently, the original C code can be used to perform functional verification of the resulting Verilog. The C2R design flow allows designers to instrument the C source code with various compiler directives and explore the design space of the resulting architectures. The compiler directives can be used to specify state machines for control, create interfaces to the resulting Verilog code, bind arrays to specific FPGA resources, specify the degree of pipelining to be used to implement loops, control variable bit widths, and enable clock gating of registers in the resulting design . C2R has been used to implement hardware designs for security, data compression, and floating-point arithmetic.

The Catapult C Synthesis Tools from Mentor Graphics synthesizes C++ source code without extensions to SystemC, Verilog, or VHDL [18]. Catapult C provides a graphical user interface that lets the designer specify area, performance, and power constraints, apply a variety of optimizations including loop merging, loop unrolling, and loop pipeling, specify operand bit widths, generate hardware interfaces, evaluate design tradeoffs, and identify bottlenecks and inefficiencies in the generated design. Catapult C also provides options for clock-gating to reduce power consumption, takes advantage of optimized FPGA resources such as block RAMs and DSP blocks, and provides automated equivalence checking to formally prove that the original C++ code and the generated HDL are functionally equivalent. Catapult C has been successfully used to generate complex hardware designs for wireless communication and image and video processing. By the end of 2008, over 100 million ASICs had shipped with hardware designed using Catapult C [18]..

Several other tools for C-to-HDL conversion have been developed. These include (but are not limited to):

1. The Nios II C-to-Hardware Acceleration Compiler from Altera [22, 23]

2. The C-to-Verilog Automated Circuit Design Tool from C-to-Verilog.com [24]

3. The Trident Compiler from Los Alamos National Labora-

tory [25, 26]

4. The No Instruction Set Computer (NISC) Technology and Toolset from the Center for Embedded Systems at the University of California at Irvine [21].

5. The Riverside Optimizing Compiler for Configurable Computing (ROCCC) Toolset from the University of California at Riverside [27, 28]

6. The SPARK Toolset from the Microelectronic Embedded Systems Laboratory at the University of California at San Diego [29, 30]

7. The GAUT High-level Synthesis Tool from the Laboratory of Science and Technology Information, Communication and Knowledge [31]

In general, the C-to-HDL tools discussed in this paper help simplify the design process, especially for people not familiar with HDLs. They allow the designs to be specified using a subset of C, sometimes with extensions. These tools also facilitate design-space exploration by allowing the designer to specify design constraints, bitwidths, and desired levels of parallelism and then evaluate design tradeoffs based on these specifications. Several of the tools generate additional resources including C support code, test benches, hardware interfaces, and synthesis and place-and-route scripts.

The C-to-HDL tools, however, also have several limitations. Only a subset of the C language is generally supported, and for several tools, extensions to the C language are needed to enable correct synthesis. In order to generate efficient code, it may be necessary to rewrite the original C code to adhere to tool-specific guidelines. Furthermore, the generated code is usually difficult to read and debug. Code that is not well written or too complex can result in designs that are much less efficient than hand-coded HDL designs. On the other hand, it is expected that the tools will continue to improve so that in the future several of these limitations may not be as severe.

2) AccelDSP and System Generator

Xilinx's AccelDSP Synthesis Tool is a high-level MATLAB-based development tool for designing and analyzing algorithmic blocks for Xilinx FPGAs [32]. Although MATLAB is a powerful algorithm development tool, many of its benefits are reduced when converting a floating-point algorithm into fixed-point hardware. For example, quantization errors and the potential for overflow and underflow are introduced into the algorithm due to floating-point to fixed-point conversion. Consequently designers may need to rewrite the code to reduce the impact of these errors and analyze the results produced by the fixed-point code to ensure they are acceptable. To facilitate this, AccelDSP provides the capability to replace high-level MATLAB functions with fixed-point C++ or Matlab models and automatically generates testbenches to facilitate fixed-point simulations. The tool automatically converts a floating-point algorithm to a fixed-point C++ or MATLAB model. It then generates synthesizable VHDL or Verilog code from the fixed-point model, and creates a testbench for verification. During the HDL generation process, it performs several optimizations including loop unrolling, pipelining, and device-specific memory mapping. A

graphical user interface allows the user to specify the bitwidths used in the generated code and to guide the synthesis process.

The AccelDSP Synthesis tool provides several advantages. It is a tightly integrated component of the Xilinx XtremeDSP Solution and the MATLAB toolset, which allows it to utilize MATLAB's mathematical modeling and data visualization features. To improve the design's efficiency, it automatically utilizes Xilinx IP cores and generates code blocks for use in Xilinx System Generator, which is described below. AccelDSP also provides capabilities to replace high-level MATLAB functions with fixed-point C++, MATLAB, or HDL code by specifying the target Xilinx FPGA model, intermediate data precision, and desired resource distribution. HDL test benches are generated automatically from the corresponding fixed-point C++ or MATLAB model and these testbenches can be used to verify functional equivalence between the higher-level model and the resulting HDL. Furthermore, overflow and underflow that occur in the fixed-point code are reported by the AccelDSP simulation tool to help designers find potential errors that occur due to the floating-point to fixed-point conversion process. AccelDSP also provides a set of graphical tools, including probe functions, design reports, and plots to visualize and analyze the system. AccelDSP allows designers to define constraints and control resource usage and timing. For example, the user may choose to expand a "for loop" into multiple parallel hardware blocks or a single hardware block that is reused for several iterations. The user may also provide timing constraints that result in a pipelined design.

AccelDSP also has several limitations. For example, it cannot convert all MATLAB files. Rather, the MATLAB file has to be written in a specific way, and only a limited subset of MATLAB can be used. AccelDSP only works with Xilinx FPGA chips so designs cannot easily be ported to FPGAs from other vendors. Furthermore, the generated HDL can be difficult to read and debug. For many algorithms, the amount of resources required by designs generated using AccelDSP is greater than the amount of resources required by designs generated using hand-coded HDLs.

Xilinx's System Generator is a high level design tool that utilizes MATLAB Simulink and enables designers to develop DSP hardware designs for Xilinx FPGAs [33]. It provides over 90 parameterized DSP building blocks that can be used in the Matlab Simulink graphical environment. The design process with Simulink and System Generator is simply selecting DSP blocks, dragging the blocks to their desired location, and connecting the blocks via wires. These blocks and their communication links can be converted from Simulink to Verilog, VHDL, or FPGA bit files. System Generator can also utilize blocks generated by AccelDSP.

System Generator has several strengths. In particular, it is a useful tool for designers with no previous experience with FPGAs or HDL design. In addition to directly generating VHDL and Verilog code, it also provides a resource estimator that quickly estimates the FPGA resources required by the design prior to placement and routing. System Generator can create a hardware simulation model, and integrate with a Simulink software model to evaluate complete applications including analog signals. For example, Simulink can be used to create a sine wave with pseudo-random noise that serves as an input to a System Generator hardware model, which writes it outputs to a file. The complete Simulink model, which includes the System Generator model can then be used to simulate the entire system and generate a testbench for the hardware module. System Generator also has several limitations. It requires experience with Simulink to create efficient designs The Simulink tool uses an interactive graphical environment and a parameterized set of block libraries, which may not be convenient for programmers who are more familiar with high-level program languages, such as C++ or Java. Furthermore, although the blocks provided by System Generator are very useful for certain types of signal processing applications, these blocks may not meet the needs of other types of applications. Similar to AccelDSP, the HDL code produced by System Generator only works with Xilinx FPGA chips and can be difficult to read and debug.

IV. HARDWARE VERIFICATION AND DEBUGGING

Differences between the simulation results and the performance of the real hardware may result from hardware defects that went undetected by the manufacturer, inaccuracies in the models used for hardware simulation, variation from nominal environmental parameters, or unexpected operating conditions such as mutual inductance or capacitive coupling from other systems, clock jitter, power supply noise, etc. Such issues become more important for high-performance systems with tight tolerances, since they are more susceptible to problems arising from variations in the timing of internal signals. Additionally, for large, interconnected system, such as those used in HEP, full system simulation may be very costly or simply infeasible. This further motivates the importance of thoroughly testing the hardware. As we have discussed, FPGA hardware trends show rapid increases in the number of logic resources on each device. In particular, the number of registers on each devices has increased at an especially fast pace recently. The growth trends in registers and in on-chip RAM contribute to an overall trend of increasing state in FPGAs. Increasing the amount of state in a device can prove particularly troublesome during hardware verification-the process of confirming that a circuit built in hardware is consistent in behavior and performance with the circuit as it performed in simulation. Differences between the simulation results and the performance of the real hardware may result from hardware defects that went undetected by the manufacturer, inaccuracies in the models used for hardware simulation, variation from nominal environmental parameters, or unexpected operating conditions such as mutual inductance or capacitive coupling from other systems, clock jitter, power supply noise, etc. Such issues become more important for high-performance systems with tight tolerances, since they are more susceptible to problems arising from variations in the timing of internal signals. Additionally, for large, interconnected system, such as those used in HEP, full system simulation may be very costly or simply infeasible. This further motivates the importance of thoroughly testing the hardware.

Hardware verification is performed by subjecting hardware to a series of test patterns and comparing the performance to the expected results. When an error occurs, it is important to find the source of the error to determine an appropriate way of cor-

recting it. The process of locating the source of errors becomes much more difficult as the quantity of state in a device increases. This is because faulty values may contaminate the state and may propagate to different parts of the state and may take many cycles before they generate an observable error. At the same time, the number of pins on FPGAs is growing at a much slower rate than the internal state. It will become more difficult to observe internal state using external logic analyzer as the ratio of state to pins increases. This is particularly concerning because it means that designers must use much longer and more elaborate tests to verify their hardware. However, in physics applications, it is crucial to identify and eliminate any such bugs before the start of experimentation to ensure confidence in experimental results.

The problem of verifying and debugging circuits with large amounts of state is not unique to FPGAs, and has been extensively studied in the integrated circuit domain [4]. Today, engineers use a set of design techniques known as design for testability (DFT) and built-in self-test (BIST) to automatically apply tests internally and more easily probe the contents of state registers [41]. While these techniques are useful, they come with a cost; adding DFT and BIST consumes chip resources, may require extended design time, and often results in reduced operating frequency. However, because FPGAs are reprogrammable, they have the unique ability to be able to potentially use these techniques without reducing the performance of the final design. In the remainder of this section, we will describe the software tools available for performing BIST on FPGAs in a fast and efficient manner.

A. Integrated Logic Analyzers

Major FPGA vendors have provided tools to alleviate the problem of hardware verification. Xilinx's ChipScope Pro [6] and Altera's SignalTap II Embedded Logic Analyzer [12] enable designers to probe and monitor an FPGA's internal signals in real-time. These tools considerably cut verification time and effort in order to eliminate hard-to-detect bugs. The tools help equip a design with embedded hardware logic analyzers that sample data and transactions on selected signals and nodes. ChipScope Pro further provides the ability of forcing internal signals to specified values. Any internal signals in the design can be selected for monitoring. The sampled data are stored in the FPGA's embedded Block RAMs. Data are sent to a personal computer using the JTAG interface, the same interface used in FPGA programming, to give a visualized demonstration of the internal signals. Designers can easily observe and analyze transactions on internal signals of the design in real-time by means of a Software Logic Analyzer installed on a PC. Data sampling is triggered at runtime by a set of predefined conditions that can be set using a graphical user interface. The data sampling lasts for the number of clock cycles specified by the designer.

This approach of utilizing integrated logic analyzers removes or reduces the need for specific external hardware. These tools provide relatively complete observability to designers. They are especially useful for large designs, which often have a myriad of signal and data variations to verify. Designers are able to control the value of internal signals with ChipScope Pro. This is especially valuable in complex sequential circuits where it may take a long sequence to external inputs to change certain internal signals. In addition, signal monitoring is done by on-chip configurable logic analyzers, while the FPGA is working under standard operating conditions. This eliminates the need to purchase expensive external logic analyzers and chip testers. Hence, these tools provide an easy, yet powerful approach for FPGA design verification that lowers project costs, saves design time, and helps find bugs early in the implementation process.

Although this approach supplies the designer with new verification capabilities, it has some drawbacks and limitations. The number of observed signals and sampled time depend upon the free Block RAMs available on an FPGA, and since this approach uses FPGA resources, it might have negative timing impact on a design. Furthermore, defining a proper trigger condition that leads to bug detection might be challenging. Finally, embedded logic analyzers are not able to capture signal glitches and to test clock signals, because data is sampled at the hardware's clock frequency and thus cannot perform clock super-sampling.

1) Chipscope Cores

ChipScope is composed of a set of cores to promote the design verification process. The Integrated Logic Analyzer (ILA) core, as the most common core, is used for signal monitoring. The Integrated Bus Analyzer (IBA) core simplifies system bus monitoring. The Integrated Controller (ICON) core is used to set trigger conditions and send data from Block RAMs to the PC via the JTAG interface. The Agilent Trace Core 2 (ATC2) core provides an interface between the embedded logic analyzer and the Agilent FPGA trace port analyzer. The virtual input/output (VIO) core provides the designer with signal controllability along with signal monitoring. The internal bit error ratio tester (IBERT) core allows the designer to detect bugs hidden in RocketIO serial I/O designs.

V. Conclusion

The trends in FPGA hardware show exponential increases in device logic, on-chip memory, and I/O bandwidth over recent years. Process technology is in place to allow FPGA manufacturers to maintain this growth in the near future. This improvement in FPGAs could allow future HEP systems to incorporate more intricate and flexible algorithms, implement higher-resolution processing, and perform system integration. Achieving these goals will require larger, more complex designs on each FPGA. To manage increasingly complex designs while still working within constrained cost and time budgets, system developers must adopt a more scalable design methodology. This methodology must extend across the entire design process, from specification and design exploration to testing and hardware verification. In this paper, we have presented core design concepts and emerging software tools that can serve as the foundation for a design methodology that can scale to meet the next generation of FPGA-based HEP systems.

VI. Acknowledgement

References

[1] CMS Collaboration. CMS TriDaS Project: Technical Design Report; 1, The Trigger Systems, CERN (2000).

[2] W.H. Smith, P. Chumney, S. Dasu, M. Jaworski, and J. Lackey. CMS Regional Calorimeter Trigger High-Speed ASICs, *6th Workshop on Electronics for LHC Experiments*, 2000-2010 (2000).

[3] Intel Corp. Press Release: Intel Developer Forum 22nm News Facts (2009).

[4] A. Ghosh, S. Devadas, and A.R. Newton. Test Generation and Verification for Highly-Sequential Circuits, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 652-667 (1991).

[5] Coverity Inc. Scan Open Source Report (2009).

[6] Xilinx Inc. ChipScope Pro 11.3 Software and Cores (2009).

[7] IEEE Std. 1666-2005. Open SystemC Reference Manual (2005).

[8] IEEE Std. 1800-2005. System Verilog: Unified Hardware Design, Specification, and Verification Language (2005).

[9] P. Hamill. Unit Test Frameworks (2004).

[10] S.S. Bhattacharya, S. Kedilaya, W. Plishker, N. Sane, C. Shen, and G. Zaki. The DSPCAD Integrative Command Line Environment: Introduction to DICE Version 1, UMIACS-TR-2009-13 (2009).

[11] E.A. Lee and D.G. Messerschmitt. Synchronous Dataflow, *Proceedings of the IEEE*, 1235-1245 (1987).

[12] Altera Corp. Design Debugging Using the SignalTap II Embedded Logic Analyzer (2009).

[13] C. Hsu, M. Ko, and S.S. Bhattacharyya. Software Synthesis from the Dataflow Interchange Formal, *Int. Workshop on Software and Compilers for Embedded Systems*, 37-49 (2005).

[14] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S.S. Bhattacharyya. Functional DIF for Rapid Prototyping, *Rapid System Prototyping*, 17-23 (2008).

[15] Impulse Accelerated Technologies. Impulse CoDeveloper (2009). Available from `http://www.impulseaccelerated.com/products.htm`.

[16] D. Pellerin. Impulse C-to-FPGA Workshop, *Fourth Annual Reconfigurable Systems Summer Institute* (2008). Available from `http://www.rssi2008.org/proceedings/tutorial/Impulse.pdf`.

[17] Synforma. PICO Express FPGA (2009). Available from `http://www.synfora.com/products/picoExpressFPGA.html`.

[18] Mentor Graphics. Catapult C Synthesis (2009). Available from `http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/`.

[19] Cebatech. Cebatech Technology (2009). Available from `http://www.cebatech.com/technology`.

[20] S. Ahuja, S.T. Gurumani, C. Spackman, and S.K. Shukla. Hardware Coprocessor Synthesis from an ANSI C Specification, *IEEE Design & Test of Computers*, 58-67 (2009).

[21] Center for Embedded Systems. NISC Toolset User Guide (2007). Available from `http://www.ics.uci.edu/~nisc/toolset/Quick-Guide.pdf`.

[22] Altera Corporation. Nios II C-to-Hardware Acceleration Compiler (2009). Available from `http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html`.

[23] Altera Corporation. Nios II C2H Compiler User Guide (2009). Available from `http://www.altera.com/literature/ug/ug\_nios2\_c2h\_compiler.pdf`.

[24] The C-to-Verilog Automated Circuit Design Tool (2009). Available from `http://www.c-to-verilog.com/`.

[25] Trident Compiler (2009). Available from `http://sourceforge.net/projects/trident/`.

[26] J.L. Tripp, M.B. Gokhale, and K.D. Peterson. Trident: From High-Level Language to Hardware Circuitry, *IEEE Computer*, 28-37 (2007).

[27] The University of California at Riverside. The Riverside Optimizing Compiler for Configurable Computing (ROCCC) Toolset (2009). Available from `http://www.cs.ucr.edu/~roccc/`.

[28] Z. Guo, W. Najjar, and A.B. Buyukkurt. Efficient Hardware Code Generation for FPGAs, *ACM Transactions on Architecture and Compiler Optimizations*, 1-26 (2008).

[29] Microelectronic Embedded Systems Laboratory. SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits (2004). Available from `http://mesl.ucsd.edu/spark/`.

[30] S. Gupta, R.K. Gupta, N.D. Dutt, and A. Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, Kluwer Academic Publishers, (2004).

[31] LABSTICC. The GAUT High-level Synthesis Tool (2009). Available from `http://www-labsticc.univ-ubs.fr/www-gaut/`.

[32] Xilinx, Inc. AccelDSP Synthesis Tool (2009). Available from `http://www.xilinx.com/tools/acceldsp.htm`.

[33] Xilinx, Inc. System Generator for DSP (2009). Available from `http://www.xilinx.com/tools/sysgen.htm`.

[34] Xilinx, Inc. Virtex 2.5 V Field Programmable Gate Arrays, DS003-1 v2.5 (2001).

[35] Xilinx, Inc. Virtex-E 1.8V Field Programmable Gate Arrays, DS022-1 v.2.3 (2002).

[36] Xilinx, Inc. Virtex-II Platform FPGAs: Complete Data Sheet, DS031 v3.5 (2007).

[37] Xilinx, Inc. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, DS083 v4.7 (2007).

[38] Xilinx, Inc. Virtex-4 Family Overview, DS112 v3.0 (2007).

[39] Xilinx, Inc. Virtex-5 Family Overview, DS100 v5.0 (2009).

[40] Xilinx, Inc. Virtex-6 Family Overview, DS150 v2.0 (2009).

[41] L.T. Wang, C.W. Wu, and X. Wen. VLSI Test Principles and Architectures: Design For Testability (2006).