A Research Agenda for Improving Configurable Computing Design Productivity

Executive Summary

2008 FPGA Tool-Flow Workshop, Salt Lake City, UT

Mike Wirthlin, BYU
Brent Nelson, BYU
Brad Hutchings, BYU
Peter Athanas, VT
Shawn Bohner, VT

This report summarizes the results of a research study group conducted by Brigham Young University and Virginia Tech to investigate Field-Programmable Gate Array (FPGA) design productivity. The objective of this study is to investigate the full FPGA tool flow and identify potential solutions in all stages of the tool flow to provide *revolutionary* improvements in design productivity. The investigators in this study have met with many experts in the field, read many reports and articles, participated in a number of conferences, and experimented with a number of tools to help us better understand the technical challenges limiting FPGA design productivity.

In the course of this study we have identified several key challenges limiting design productivity and identified several critical technical research focus areas aimed at resolving the FPGA design productivity problem. This report summarizes our recommendations and proposes a research plan for solving the most important design productivity challenges. We believe that significant advances can be made in FPGA design productivity with adequate investment in the proposed research areas.

This report has been prepared for participants of the 2008 FPGA Tool-Flow Studies Workshop held in Salt Lake City on June 5, 2008. Details from this report will be presented at the workshop and attendees will have the opportunity to comment on the report findings and provide additional suggestions and recommendations.

This study is sponsored by DARPA/IPTO under contract FA8650-07-C7745 and administered by AFRL/RYDI.

1 Introduction

The importance of Field Programmable Gate Arrays (FPGAs) for Department of Defense systems is well understood. The Special Technology Area Review (STAR) on FPGAs, for example, clearly indicates that FPGAs are a crucial electronic component in many DoD electronic systems[1]. The report indicates that FPGAs will be used within many DoD systems for some time and will likely grow in importance as the performance and architectures of FPGAs improve. FPGAs are used within DoD for the same reasons they are used in commercial systems: reduced time to market, lower NRE costs, infield programmability, lower design and validation costs, and rapid prototyping.

Several FPGA architecture trends suggest that FPGAs will become more important in the future. First, FPGAs are closely following Moore's law and are benefiting from the increased logic density available with new process technologies. Second, FPGAs are continually adding more system level functionality such as advanced I/O standards, bus interfaces, and memories. Third, FPGAs are integrating a variety of heterogeneous processing elements such as DSP processors, programmable processors, and computing elements. Fourth, FPGAs are providing multiple processors (both hard and soft) that can be organized into chip-level multiprocessing. This growing density, raw computational throughput, and system functionality suggests that FPGAs will play an increasingly important role in future DoD systems.

While FPGAs provide many benefits, the effort and skill required to create working FPGA designs is growing and consumes significant design resources during system development. The inability to create FPGA designs more productively limits the ability to exploit the growing density, capability, and performance potential of modern FPGA architectures. In fact, one of the key recommendations of the STAR report is the need to address the science and technology gap that includes the advancement of electronic design automation (EDA) for FPGAs. Unless significant advances in FPGA design productivity are made, the full benefits of FPGAs cannot be realized.

1.1 Design Productivity Gap

Perhaps the biggest challenge limiting FPGA design productivity is the so called design productivity gap[2]. Silicon density continues to double every 1.5 to 2 years while design capabilities are growing at a much slower rate. Design productivity must improve at a rate similar to Moore's Law just to keep from falling behind. While incremental improvements in design productivity are being made, the rate of growth in design productivity is much lower than Moore's law resulting in increasing design times for each new FPGA generation. Significant effort and investment in design techniques and methods are necessary for closing this design productivity gap.

Most of the largest FPGA devices available today are built using 65 nm technology[*]. These modern FPGAs contain a tremendous amount of logic, computation, and memory resources and can be used for a variety of high-speed digital systems and high-performance computing applications. The growth in density and capability of FPGAs will undoubtedly continue in the future. **Error! Reference source not found.** suggests the resources that may become available on future FPGA devices using newer fabrication technologies. If FPGA density keeps pace with Moore's law, we expect the largest FPGAs in a 22 nm technology to

[*] Altera announced the introduction of the first 40-nm FPGA (Stratix IV) on May 19, 2008.

contain almost 3 million look-up tables, several thousand dedicated multiplier/DSP blocks, and up to 100Mb of internal memory.

Table 1 Density and capability of future FPGA technologies

Technology	Year	LUTs	DSPs	Memory
65 nm	2007	340 k	500	10 Mbit
45 nm	2010	700 k	1000	21 MBit
32 nm	2013	1,400 k	2000	42 MBit
22 nm	2016	2,900 k	4300	89 MBit

While the density of future FPGAs will certainly increase, it is likely that the architecture of future FPGAs will continue to evolve. As more transistors become available, it is likely that the logic and computing resources will become coarser grain and more "hard-core" resources (such as PCI express) will be added to keep up with the latest and highest speed I/O interfaces. We also expect that a variety of new FPGA device families will be introduced to address the needs of specific markets. As such, FPGAs will present a moving target to Computer Aided Design (CAD) tools and we believe it will become increasingly difficult to address the gap between FPGA design productivity and FPGA circuit density as a result.

1.2 FPGA Use Models

There has been considerable interest by non-traditional circuit designers to use and "program" FPGAs. These application experts and programmers recognize the benefits of FPGAs and seek ways to exploit the efficiency, reprogrammability, and computational density of FPGAs for their application-specific problems. These non-traditional FPGA programmers come from a variety of backgrounds including signal processing, embedded systems, communications, and high-performance computing. These experts, however, do not have the traditional digital design skills to effectively "program" the FPGA using existing FPGA design tools. While there are some FPGA design tools created for such domain experts, FPGA design for non-traditional circuit designers remains very difficult. Creating FPGA solutions to their application specific problems is far less productive than the programming environments they are used to. Future advances in design productivity for FPGAs must significantly simplify the design/programming process of FPGAs for non-traditional FPGA users.

The wide variety of users interested in using FPGAs suggests that new design methods and techniques are needed for FPGA design. We introduce the concept of an FPGA "use model" and define a number of "use models" to clarify the design issues that face FPGA designers and non-traditional FPGA programmers. Each model has a different set of design challenges, design constraints, and programming model. While we have identified a variety of unique FPGA use models, we will focus on two FPGA use models for this report: *ASIC replacement* and *Configurable Computing*.

ASIC Replacement is the most common FPGA use model. In this use model, FPGA devices are used to perform general purpose digital functions that might otherwise be performed in a custom integrated circuit. In this use model, the behavior and timing of the FPGA are specified in great detail including clock-cycle accuracy of the interfaces and internal logic. The design goal is to minimize cost (i.e. optimize hardware) and validate

circuit functionality (including meeting timing constraints). The design is optimized in a way that allows the least expensive FPGA device to be used in the system. ASIC replacement applications typically involve the design of custom PC boards onto which the FPGA is placed, custom I/O interfaces, custom clocking requirements, etc. Much of the design activity involves creating the register transfer level (RTL) implementation from some detailed system specification.

Configurable computing is an FPGA use model in which FPGA devices are used to perform application specific *computation*. The large amount of logic resources available in modern FPGAs allows complex calculations and application-specific computations to be performed more efficiently and often with higher performance than more traditional CPU-based architectures[3]. Standard platforms and boards are most often used for configurable computing to simplify the design process and facilitate reuse. When mapping a computation onto a CCM the goal is often to simply get the design to fit into the available FPGA(s) rather than to optimize the design down to the last gate.

We have focused our productivity study on technologies and design methods that improve design productivity for *configurable computing* rather than for *ASIC replacement*. We believe that there is great potential for improving the design productivity for configurable computing and that with sufficient investment in a number of important technical areas, revolutionary improvements in design productivity for configurable computing are possible. While the techniques and ideas we present in this report are targeted towards configurable computing, we believe that many of these ideas can be successfully applied to ASIC replacement and that some improvements in ASIC replacement design productivity are also possible.

Perhaps the biggest challenge limiting design productivity for configurable computing is that the design methods used in configurable computing are primarily the low-level design methods developed for the ASIC replacement use model. The design of configurable computing "programs" is essentially circuit design – low-level digital design methods such as RTL design are used to define complex computation and behavior. In fact, most of the design processes in contemporary configurable computing have direct counterparts in ASIC design[4]. ASIC replacement design methods are insufficient for configurable computing and new methodologies are needed to improve design productivity.

2 Lessons from Software Productivity

While current design methods for configurable computing closely resemble the design methods for ASIC replacement, the design goals and constraints of configurable computing are more closely related to traditional software development. In traditional software design, the programmer specifies high-level behavior and relies on optimizing compilers, profilers, debuggers, and other tools to translate the behavioral description into an efficient implementation. Ideally, FPGA design for the configurable computing use model should look the same – programmers specify application-specific behavior in some high-level specification and use a variety of tools to translate this behavior into an efficient implementation onto the FPGA or configurable computing machine (CCM).

Because of the close relationship between configurable computing design and software programming, it is instructive to look at the major innovations in software productivity over the last fifty years. *We believe that the current design tools and methods for configurable computing are still primitive and resemble the software practices of the 1960's.* Software

productivity has progressed dramatically in the past half century and these improvements hold important insights for the configurable computing community. Many of the improvements in software productivity can be applied to configurable computing. The major advances in software productivity can be categorized into one of four different groups:

1. **Increased Abstraction**. Major improvements in programmer productivity have been realized by introducing new languages and design methods that reduce the amount of detail required by the programmer. The transition from machine code to assembly language and from assembly language to 3rd generation languages[5] allowed programmers to create complex programs without understanding low-level details of the microprocessor architecture.
2. **Reusable Artifacts**. An important way of improving software productivity is reusing previously created software artifacts[6]. There are many levels of software reuse including reuse of applications, concepts, libraries, design patterns, and portable programs. The recent growth in reusable software components for web-based applications such as web services demonstrates the potential improvements in productivity through reuse.
3. **Software Process**. Recognizing that most early software development was done in an ad-hoc manner, new software processes were developed to improve productivity. Productivity improvements of 20% - 40% have been demonstrated for small software projects and up to 500% for large software projects[7,8].
4. **Automation**. Automating tedious tasks played an important role in improving productivity[9]. Tools to automate and integrate a variety of tasks have reduced errors and sped software development by over 30%.

As suggested above, configurable computing systems have yet to enjoy even the most basic productivity benefits demonstrated by software. While there are some encouraging signs of progress with new languages and compilation tools, contemporary FPGA design more closely resembles the lowest-level machine code programming of the very earliest computer systems. Significant advances in each of the four areas above are necessary for FPGA design in configurable computing systems to enjoy the benefits in productivity that were demonstrated by traditional software systems.

Using advances in software productivity as a guide, we have identified three broad technical areas that are most promising for configurable computing design productivity: *reusing artifacts*, *raising design abstractions*, and *increasing the interactivity and debug infrastructure* (i.e. "turns per day"). Software productivity has made *significant* advances in the last fifty years by making many advances in each of these areas. These areas of productivity are interrelated and design productivity will significantly increase if advances are made in each of these areas and applied at all levels of the design methodology.

3 Measuring Productivity

Before suggesting approaches and techniques for improving design productivity, we must have a clear definition and measure of design productivity. During the course of this study we developed a productivity model to guide our investigation[10]. Models have limitations and the model we propose is no exception. It is not meant to predict the precise design time required for a given application or design. Rather, it is more qualitative in nature and points out what

we believe to be the first-order contributors to design productivity and their inter-relationships.

Our first measure of design productivity is simply the rate at which hardware is developed:

$$DesignProductivity = \frac{CC}{DesignTime} \quad \textbf{(1)}$$

Here, *CC* represents the *circuit complexity* of the final design, as measured in gates, LUTs, transistors, etc. The output of hardware design is hardware, a physical artifact that can be measured and that has quantifiable costs in several dimensions (silicon area, power, etc.). Unlike software, our model does not measure the *input* of the design process (i.e. lines of code/day) but rather the physical *output* of the design process (the amount of circuitry produced).

3.1 Design Time

The majority of the effort required to complete a hardware design is spent in debug and verification, with values in the 70% range being common. Thus, design time for configurable computing applications strongly depends on the number of design turns required to complete the verification of the design, and the ease with which those design turns can be completed. The design time is proportional to the number of design "turns" and can be approximated as:

$$Days = \frac{Turns}{TPD} \quad \textbf{(2)}$$

where, *Turns* is the total number of design iterations required and *TPD* is "turns per day" (debug iterations per day).

3.2 Number of Turns Required to Complete a Design

The number of design turns required to generate a bug-free design (*Turns*) is dependent on the size of the input description as well as the frequency of occurrence of bugs embedded in that input description. We represent *Turns* as:

$$Turns = ILOC \times \frac{Turns}{ILOC} \quad \textbf{(3)}$$

In this equation, *ILOC* stands for "Input Lines of Code" and should be considered as a proxy for the quantity "complexity of the design source", and could be measured in lines of input code, number of nodes in a graphical description of the circuit, etc.

The term *Turns/ILOC* in Equation (3) is a measure of how many debug iterations are required per *ILOC* and is based on a simple assumption — that design errors are distributed uniformly through the design at a certain rate, and that it will require one design turn to uncover and fix each such error.

Combining Equations (1), (2), and (3) leads to the following design productivity equation:

$$DesignProductivity = \frac{CC \times TPD}{ILOC \times \frac{Turns}{ILOC}} \quad (4)$$

3.3 Effect of Reuse on Design Time

Equation (4) fails to capture the effect of reuse on design productivity. That is, design productivity improves when the designer is able to reuse pre-existing design pieces, requiring less original design. Reuse can be modeled as reducing the number of lines of code that the designer must write from scratch. *ILOC* (the code the user must create) can thus be modeled as consisting of two parts, the new part of the design that must be created from scratch and the interface code that must be created for the integrating the reused portions. It is useful to express this in a form where the amount of reuse is explicitly represented, along with the overhead associated with that reuse:

$$ILOC = ILOC_0 \times [(1-R) + (O \times R)] \quad (5)$$

In this equation, $ILOC_0$ is the amount of code originally required to describe the circuit without the benefit of any reuse (the amount of code required to create it entirely from scratch). *R* is the *fraction* of the design satisfied by reusing previously-design circuitry – the user must only create $ILOC_0 \times (1-R)$ lines of new design code.

Reuse is not free, however, and *O* represents the *overhead* of that reuse. It is expressed as a percentage of *R* and represents lines of new code that the designer must create to interface the reused circuitry to the rest of the design. As a concrete example, consider a design where $ILOC_0$=100, *R*=80%, and *O*=10%. Without the benefit of reuse, this would require the designer to write 100 lines of code. With reuse, the user would have to create: $100 \times [0.2 + 0.1 \times 0.8] = 28$ lines of code. The reuse overhead (*O*) reduces the benefit of reuse and if too high will eliminate any of the net advantages of reuse.

3.4 A Final Model

Substituting Equation (5) into Equation (4) gives the following final equation for design productivity:

$$DesignProductivity = \frac{CC \times TPD}{ILOC_0 \times [(1-R) + (O \times R)] \times \frac{Turns}{ILOC}} \quad (6)$$

This productivity model brings together design abstraction, turns per day, and reuse, and describes how each of these factors individually contributes to programmer productivity. We believe that orders of magnitude improvements in design productivity are possible if revolutionary advances are made in each of these three areas. For example, reuse alone may provide a 4× improvement in productivity as shown above. By developing and embracing higher levels of abstractions, the design detail required for a system may be reduced by a factor of 2× (i.e. increase the ratio of CC/ILOC by 2). Raising the abstraction and reusing FPGA artifacts may ultimately reduce the number of “turns” required to verify the design by 50% (Turns/ILOC). Finally, creating infrastructure, tools, and new processes to significantly improve interactivity may increase the “Turns per day” by a factor of 2× or more. Taken together, these advances in all three areas would provide more than an order of magnitude improvement in design productivity.

4 Research Approaches

The productivity model defined in the previous section identifies three research focus areas that we feel are most important to address (reuse, raising design abstractions, and increasing the number of "turns per day"). We do not believe that design productivity for configurable computing will increase appreciably unless advances are made in each of these areas – there is no silver bullet to solve this problem. Each of these areas is interconnected and design productivity will significantly increase only if advances are made in each of these areas and applied at all levels of the design methodology. This section will summarize each these three areas and suggest specific approaches that could be applied to improve productivity.

4.1 Reuse

It is well known that reuse of software has been a significant factor in improving software design productivity[11,12]. Today's software systems are typically created by reusing software libraries, integrating reusable components, and dynamically integrating autonomous executables (COM, CORBA, etc.). Very large and complex software services can be created by exploiting the many available reusable software components and service oriented architectures. The successful exploitation of software reuse has led to significant improvements in productivity, higher quality code, fewer bugs, and lower software maintenance costs [13].

While these relatively new forms of reuse have provided remarkable improvements in productivity, software systems have exploited reuse of system infrastructure for many years. For example, even the simplest "Hello World" program involves a tremendous amount of code reuse. Reusable firmware, operating system calls, and run-time libraries (i.e. libc) are necessary to run this simple program.

Reuse within hardware systems, however, has significantly lagged behind that of software. While there is great interest in exploiting reuse for hardware design, the risk associated with reusing 3rd party circuits and the technical challenges of integrating "reusable" hardware circuits has inhibited the widespread adoption of reuse methods. One study suggested that if the time required to reuse a component was greater than 30% of the time required to design the component from scratch, design reuse would fail (designers would choose not to reuse)[14]. The risk and cost of hardware reuse must be reduced before hardware reuse is widely used.

While hardware reuse is difficult, the potential improvements in productivity are significant[15]. For example, if 80% of a hardware design is created by reusing existing hardware (i.e. R=0.8) and the effort to integrate reusable hardware is 10% (i.e. O=0.1) then hardware design productivity will increase by a factor of 4 (see Equation 6). Achieving this level of reuse today and at such a low cost is difficult. However, the improvements in software reuse over the last four decades suggests that significant improvements in hardware reuse can be made with appropriate technology advancement and community cooperation.

Like software, there are many different ways to exploit reuse during the design and deployment of a hardware system. These include the following:

- Library cell reuse - this is what most think of when reuse is proposed and is the use of cells from a standard library which perform a specific function (FFT, for example).

- Retargeting reuse - the porting of designs between devices from different manufacturers or even between devices from a single manufacturer.
- Design pattern reuse - the reuse of structures such as pipelining or bit-serial arithmetic in the creation of a design[16].
- Architecture reuse - meta-architectures are architectures layered on top of traditional reconfigurable fabrics to facilitate reuse.
- Platform reuse - the use of standard CCM-like platforms with FPGAs, memories, and I/O capabilities.
- Interface reuse - the use of standard I/O connections to alleviate the designer creating custom interconnect for each application.

We propose four specific research topics related to reuse that we believe can significantly improve the benefits of reuse within the FPGA design flow.

4.1.1 Library Reuse Infrastructure

The most common and direct form of hardware reuse is the reuse of hardware components. Predefined hardware circuits (otherwise known as ``intellectual property'' or IP cores) are created and verified and then later inserted in a larger hardware circuit. While such reuse occurs frequently within an organization, reuse between organizations and third-party developers is limited. In addition, it is difficult to reuse hardware components over time – they become obsolete and reusing today's modules on tomorrow's devices is problematic. One problem is the lack of standards – hardware circuits are developed in a variety of tools and incompatible languages that inhibit the reuse of the circuit in new environments and design flows. Developing standards for describing and representing reusable hardware will enable a variety of high-level tools to take advantage of a variety of cell libraries developed within different tools[17].

The concept of library reuse could go one step further and adopt the library and sharing models that have demonstrated promise in the software engineering realm. One example from the software realm is the Common Object Request Broker Architecture (CORBA), which enables software components written in multiple computer languages and running on multiple computers to work together. This objective is similar to the needs of reconfigurable computing, but goes one step further. In reconfigurable computing, a repository architecture is desired that not only enables hardware components written using different specification languages to be maintained in a common repository, but also provides the capability of interface synthesis (see below) that promotes IP portability. Library extensions such as this would have an obvious impact in enhancing reuse in a typical design environment.

4.1.2 Architecture Shaping Through Library Standards

Standardized well-characterized libraries, common among all qualified DoD FPGA vendors would greatly enhance code reuse and code portability, and mitigate early obsolescence of code bases. In the software world, standardized libraries such as VSIPL[18] and LinPack have directly affected how compilers are built and even how machines are made. If such a configurable computing library had a (forcibly) high adoption rate, it is likely that device vendors would be motivated to optimize their mappings to elements in the library, or even make architectural enhancements to give them a competitive advantage over their peers. This seems to be an obvious tactic for the industry to deploy; however, there is currently little

incentive for FPGA vendors to do this. Furthermore, contemporary FPGA architectures are crafted to suit the needs of their primary customers who value logic density above all else. It is conceivable that a critical mass of users with a common use-model (via mandatory library interfaces) could ultimately inspire competitive forces among device manufactures to optimize their architectures. This process is referred to here as architecture shaping.

There is historical precedent that suggests that FPGA architecture shaping can achieve success. Consider the RISC "revolution" of the 1980's. Here, the concept of highly dense and complex ISAs (analogous to contemporary hardware-centric FPGA architectures) were abandoned in favor of giving the compiler more control in the process. If there were an entity that could create a broadly acceptable library, possibly through a standards process, it is possible that a “critical mass” could be attained. Compliance to this standard could be mandated by the DoD as a condition of these requirements and mandates could be phased in over time. Ultimately, vendors could be required to comply as a condition for DoD participation.

There are potentially secondary rewards from architecture shaping. Standards will also create the opportunity for 3rd-party tool vendors to compete in the CAD space that is currently mostly exclusive to the device vendors. This could potentially impact the TPD factor in the productivity equation.

4.1.3 Dual Layer Compilation

Synthesizing computing circuits onto arbitrary hardware is much more difficult than compiling a program onto a sequential processor. The tasks must be assigned to resources and scheduled in time. A two-level compilation strategy may assist the compiler and synthesis tools during this process. Standard “meta-architectures” are defined that represent more coarse grain architectures than FPGAs and provide a higher level abstraction than low-level LUTs and wires[19]. The compilation and synthesis process can be simplified by compiling to this meta architecture level using higher level abstraction tools and then using low-level device specific tools to generate actual computing circuits. Further, a two-level compilation strategy will lead to greater portability and reusability by more easily allowing computations compiled to a meta-architecture to be retargeted to other low-level device architectures.

4.1.4 Interface Synthesis

FPGA circuits are difficult to reuse for several reasons. First, the designer must choose a circuit to reuse. Second, the designer must understand the low-level details of the reusable circuit interface. Third, the designer must create custom circuits to talk to the interface, and fourth, the designer must then verify the system with the reusable core. Much of the time involved in reusing FPGA circuits is the extra design time required to interface a reusable circuit to a new system.

The objective of interface synthesis is to reduce the effort required to reuse a circuit. This is done by automatically synthesizing the interface between a reusable circuit and the new circuit. This is done by encapsulating the circuit interface of reusable circuits in meta-data descriptions and automatically synthesizing the interface between the circuit and the system. If done properly, modules can ”seamlessly“ transition from one design with one set of interface requirements and standards to another design. The resulting impact is that more reuse will occur, resulting in a productivity improvement.

4.2 Abstraction

Since its inception, computer science has claimed (and proven) that raising the level of abstraction leads to significant productivity gains. Programming for software systems has undergone a transition between many different levels of abstraction including machine code, assembly language, procedural programming languages, etc. Indeed, early gains of 5X were reported as programmers moved away from assembly language toward PL/I and other higher-level languages. These productivity improvements came about for two reasons[20]. First, the statements in higher-level languages are more powerful thereby allowing programmers to describe their application with fewer lines of code. Second, higher-level languages eliminate whole classes of bugs by automatically taking care of many low-level details. The bugs that remain are fewer in number and easier to find because they tend to be less obscure.

Many new hardware design tools and languages are being introduced for FPGAs with higher level design abstractions. Many of these new abstractions are based on existing programming languages such as C. While these tools certainly provide improvements in productivity for those that learn and use them, they have not significantly improved design productivity for the general design community. One reason for this is that such languages are essentially HDLs – the skill set for the designer with such languages still requires hardware capabilities. They may provide higher levels of abstraction than VHDL or Verilog and thus remove details from the designer's view, but they still require an understanding of clocking, scheduling, pipelining, and other digital systems design concepts. Another reason is that these languages, while based on familiar programming languages such as C, have new concurrent semantics. A familiarity with the base language such as C may actually be a handicap when trying to learn these new semantics. Third, many of these abstractions are based on inherently sequential languages. The sequential nature of these languages limits the ability to specify and to exploit the massive parallelism available in hardware circuits[21].

While these recent tools and languages are a step in the right direction, we believe that they are insufficient for moving hardware design to a significantly new level of design productivity. Additional advances in abstractions, languages, and compiler/synthesis tools are needed to increase productivity of FPGA based configurable systems. We propose several approaches that believe may extend the advantages of abstractions.

4.2.1 Parallel Languages

It is well known that the incremental performance gains through architectural improvements of uni-processors is slowing and that microprocessors will not improve performance at the rate seen in the previous three decades. To address this trend, microprocessor manufacturers are using multiple processor cores within a single device to improve performance. Multi-core processors have the potential of achieving higher levels of performance with less power and cost. Multi-core processors, however, are more difficult to program than traditional uni-processors. Most programmers are taught to program using sequential languages and compilers struggle to exploit sufficient parallelism from such sequential descriptions. To address this issue, there is much interest in parallel programming languages and compiler tools for targeting multi-core architectures.

We believe that we have a unique opportunity to exploit this growing trend. We advocate the adoption of standard, concurrent programming languages for hardware design rather than adopting sequential programming languages and adding non-standard semantic extensions. The use of concurrent programming languages will facilitate the extraction of

concurrency for hardware. Further, standard concurrent languages will lead to more platform independent descriptions of algorithms that can be targeted to either hardware or software.

4.2.2 Multi-FPGA Synthesis and Compilation

Many configurable computing systems are designed with multiple-FPGAs to provide a large amount of computing performance. These systems integrate multiple FPGAs in a mesh, ring, systolic array or other topology to provide high levels of performance for computing problems that have a large amount of parallelism. While multi-FPGA systems provide a large amount of potential computing performance, they are more difficult to program than single FPGA systems. In addition to logic design, programmers of these multi-FPGA systems must manually partition the behavior between the various FPGAs in the system.

New high-level synthesis and compilation methods are needed to automatically target multi-FPGA systems. Most synthesis and compilation techniques assume a uniform array of logic and do not consider the impact of partitioning logic and computation between disparate FPGAs with limited connectivity. Future high-level synthesis approaches must consider the impact of inter-FPGA communication and perform coarse level partitioning and resource allocation based on the topology of the multi-FPGA system. Ideally, compilers for multi-FPGA systems would be able to target *any* multi-FPGA platform to facilitate the portability of configurable computing applications across different vendors and system topologies.

4.2.3 High-level Abstraction Debugging Environments

With very few exceptions, the higher-level languages developed for configurable computing and hardware design have not provided integrated debugging tools. While these tools provide simulators, there are no tools to debug the application behavior on actual hardware systems. As such, programmers are forced to debug their hardware with low-level logic analyzers even if they adopt these higher-level programming tools. Worse yet, these higher-level tools often exacerbate the debugging task. Just as it is extremely difficult to debug the assembly code that was generated by a high-level compiler, it can be extraordinarily difficult to debug net-lists that were generated by a hardware synthesis tool.

It is essential that new tool environments are created that allow configurable-computing programmers to debug their applications in native configurable hardware using the same abstractions as their code. To date, little effort has been spent in this area, relative to the study of high-level languages. It is important to provide the programmer with the tools necessary to debug their applications on their configurable-computing machines, just as was done so successfully in the software community.

4.3 Turns Per Day

There is a big difference between debug productivity for software and debug productivity for hardware. In a typical FPGA hardware design flow, we typically achieve one to two debug iterations in a given day. With a software development tool such as *gcc,* it is possible to achieve more than 20 debug iterations per day. In fact the number 20 was chosen somewhat arbitrarily and likely is much higher, especially if one counts the use of *printf()*-based runs as debug iterations.

One of the key issues with regards to hardware debug is that there are actually two development cycles that the designer must navigate (see Figure 1). On the left is a debug cycle that approximates software development, consisting of compile, simulate, modify

design, and repeat. Once this has been done to the designer's satisfaction he/she moves to the cycle on the right which consists of synthesis/place-and-route/timing-closure/download followed by hardware execution and often confusion. These are two very different types of debug cycles. The simulation cycle on the left is very slow to simulate but provides excellent visibility into the operation of the circuit. The cycle on the right runs thousands of times faster but provides very little visibility into the operation of the circuit.

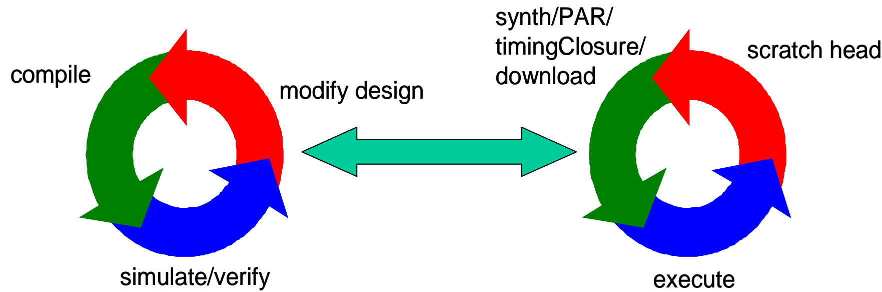

Figure 1 - Configurable Computing Development Cycles

One of the chief difficulties with this hardware design cycle is the difficulty of conducting *what-if* experiments. Such experiments are an important part of many design processes, and are exceedingly difficult in hardware design. To perform such an experiment, the user modifies his design code, and then may spend significant amounts of time stimulating to the point that he can determine whether the experiment will be successful. Often however, he must do the experiment in hardware which requires even more additional time to synthesize and implement the circuit before the experiment can even be run. In either case running such an experiment may take multiple hours. In short, most hardware design environments do not encourage interactive development.

A second difficulty with hardware development environments is a lack of infrastructure. As shown on the right side of Figure 2, typical software development environments have mature tools available for use, with many choices available. In contrast, hardware development environments are missing whole types of tools. In addition, the tool choice on the hardware side is often very limited and the tools themselves not of high quality.

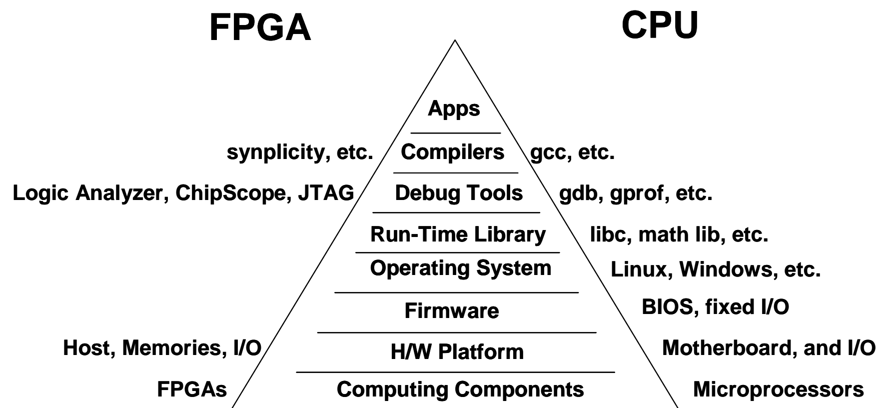

Figure 2 - Sparse Infrastructure for Configurable Computing Systems

It is our belief that the impact of improved debug infrastructure for increasing the number of debug turns per day cannot be overstated. If we could increase the number of turns per day by 10 times to achieve parity with software development environments, one could say that we would experience a 10 times increase in design productivity. However, the effect may be much greater. Increasing the number of turns per day in the debug environment has a systemic effect on the entire design process. Users no longer are forced to multitask while waiting for long implementation runs to complete. Rather, they can focus on the debug task, rapidly iterating with *what-if* scenarios and experiments and greatly multiplying their current capabilities. Thus, we believe that improving debug infrastructure may provide a nonlinear impact and give a much greater than 10 times productivity improvement, and mitigates the unproductive "busy-wait" mode of development characteristic of contemporary practices. Below we provide a number of approaches which we believe should be investigated to increase the number of turns per day a hardware designer can achieve.

4.3.1 Debug and Runtime Aids

We propose that debug and run time support can have a great impact on design productivity. Examples of the kind of tools required include:

1. An integrated design and debug environment, with support for all phases of the design, debug, and deployment process. This support should be built in from the start. The JHDL system built at Brigham Young University is an example of such an integrated environment.
2. Specialized hardware support to help during debug. Additional circuitry can be automatically synthesized into a design to help monitor the circuit's operation and thereby provide valuable debug information to the user.
3. Techniques such as checkpointing, commonly found in software systems, are feasible for hardware design and debug and can greatly streamline the debug process. Additionally, domain-specific design tools can help users develop and debug their designs at much higher levels of abstraction than gates and flip-flops.

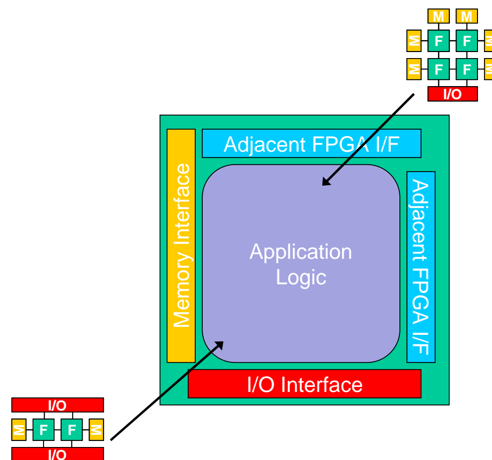

Figure 3 RC Firmware

In summary, debug and runtime aids can only be successful if they are built into the design process and CAD tools from the outset. A major problem with today's CAD tools is that they make little provision for debug, typically obfuscating their operation and intermediate file formats, and thereby preventing users from adding such debugging aids on after the fact. Importantly, we believe support for debugging runtime such as we have outlined above will not come for free — a few percentage increase in circuit area should be a good trade off for large gains in design productivity, something the software world accepted years ago.

4.3.2 Firmware

To address the lacking infrastructure shown in **Error! Reference source not found.**, we propose the use of RC firmware to significantly simplify the design and debug process. This is illustrated in **Error! Reference source not found.**, where the I/O interfaces around the periphery of a chip are standardized. These can even be precompiled onto the chip itself and may be application-independent. User designs are then compiled and, using partial configuration, they are configured onto the chip and wired up to the standardized interfaces. The benefits of such an approach would be much faster place-and-route, the possibility of the creation of a platform-independent design flow, enhanced portability, and increased reuse. We understand that such approaches have been tried by vendors in the past, and it is our belief that these have failed because they may have included too much circuitry and thus impacted the ability of a designer to place a significant design in the remaining circuit area. The approach we propose would rely heavily on synthesis and CAD tools to only insert the standardized I/O interfaces which were required for a given design, leading the maximum circuit area available for user designs.

5 Research Agenda for FPGA Productivity

As suggested earlier, design productivity for FPGA design in configurable computing systems is inadequate and many years behind the productivity available for traditional computing architectures. The primitive ASIC design tools and methods for configurable computing are inadequate for the needs and skills of the application-specific programmers of modern FPGA-based systems. Revolutionary improvements in the design tools, compilers, and design methods are necessary before FPGA-based computing systems are adopted more widely.

The progress in software productivity over the last fifty years suggests that significant improvements in productivity are possible. The major advances in software productivity that show the greatest promise for configurable computing include (1) reuse at all steps of the design flow, (2) exploiting higher levels of design abstraction, and (3) increasing the interactivity of debug and verification (turns per day). We believe that revolutionary advances in design productivity are possible only by making major improvements in each of these three areas.

6 References

[1] "Special Technology Area Review on Field Programmable Gate Arrays FPGAs", DOD Advisory Group on Electronic Devices, July 2005, ARL-SR-147.
[2] Alan Allan, Don Edenfeld, William H. Joyner, Andrew B. Kahng, Mike Rodgers, and Yervant Zorian, "2001 Technology Roadmap for Semiconductors," *IEEE Computer*, vol. 35, no. 1, pp. 42-53, 2002.
[3] Scott Hauck and André DeHon ed., *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing*, Morgan Kauffman, 2007, Chapter 21.
[4] B. Hutchings and B. Nelson, "Using General-Purpose Programming Languages for FPGA design," in *Proceedings of the 37th Design Automation Conference (DAC)*, June 2000, pp. 561-566.
[5] S. Moser and O. Nierstrasz, "The Effect of Object-Oriented Frameworks on Developer Productivity," *IEEE Computer*, vol. 29, no. 9, pp. 45-51, Sept. 1996.
[6] B.W. Boehm, "Managing Software Productivity and Reuse," *IEEE Computer*, vol. 32, no. 9, pp. 111-113, Sept., 1999
[7] J. E. Gaffney, and R. D. Cruickshank, 1992. "A general economics model of software reuse." In *Proceedings of the 14th international Conference on Software Engineering* (Melbourne, Australia, May 11 - 15, 1992). ICSE '92. ACM, New York, NY, 327-337.
[8] R.W. Selby, "Enabling reuse-based software development of large-scale systems," *IEEE Transactions on Software Engineering*, vol.31, no.6, pp. 495-510, June 2005.
[9] T. Bruckhaus, N.H. Madhavii, I. Janssen, and J. Henshaw, "The impact of tools on software productivity," *IEEE Software*, vol.13, no.5, pp.29-38, Sep 1996.
[10] B. Nelson, M. Wirthlin, B. Hutchings, P. Athanas, and S. Bohner, "Design Productivity for Configurable Computing", *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, to be published, July 2008.
[11] Jeffrey S. Poulin, *Measuring Software Reuse: Principles, Practices, and Economic Models*, Addison Wesley, 1997.
[12] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, ACM Press, 1997.
[13] Will Tracz, *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*, Addison Wesley, 1995.
[14] Emil Girczyc and Steve Carlson, "Increasing design quality and engineering productivity through design reuse," in *Proceedings of the ACM IEEE Design Automation Conference (DAC), 1993.*
[15] Jeremy Kepner, "HPC Productivity: An Overarching View," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 393-397, 2004.
[16] André DeHon, Joshua Adams, Michael DeLorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton, "Design patterns for reconfigurable computing," *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2004, pp. 13–23, IEEE Computer Society.
[17] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov, "OpenFPGA corelib core library interoperability effort," in *Proceedings of the 2007 Reconfigurable Systems Summer Institute*, 2007.
[18] Janka, R., Judd, R., Lebak, J., Richards, M., Campbell, D., "VSIPL: An Object-Based Open Standard API for Vector, Signal, and Image Processing", Proceedings of the 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vol. 2, pages 949-952, 2001.
[19] D. Campbell, D. Cottel, R. Judd, K. MacKenzie, and M. Richards, "The Morphware Stable Interface: A Software Framework for Polymorphous Computing Architectures", in *Seventh Annual Workshop on High Performance Embedded Computing*, 2003.
[20] Fredrick P. Brooks, The Mythical Man-Month: Essays on Software Engineering, Addison Wesley, 1995.
[21] Stephen A. Edwards, "The Challenges of synthesizing hardware from C-like languages," in *IEEE Design & Test of Computers*, 2006, pp. 375-386, IEEE.