FPGA-Based Accelerator Development for Non-Engineers

David C. Uliana

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter M. Athanas, Chair
Wu-Chun Feng
Krzysztof K. Kępa
Thomas L. Martin
Liqing Zhang

April 15, 2014
Blacksburg, Virginia

Keywords: FPGA, Heterogeneous Computing, HPC, Big-data, Life Sciences

FPGA-Based Accelerator Development for Non-Engineers

David C. Uliana

(ABSTRACT)

In today's world of big-data computing, access to massive, complex data sets has reached an unprecedented level, and the task of intelligently processing such data into useful information has become a growing concern to the high-performance computing community. However, domain experts, who are the brains behind this processing, typically lack the skills required to build FPGA-based hardware accelerators ideal for their applications, as traditional development flows targeting such hardware require digital design expertise. This work proposes a usable, end-to-end accelerator development methodology that attempts to bridge this gap between domain-experts and the vast computational capacity of FPGA-based heterogeneous platforms. To accomplish this, two development flows were assembled, both targeting the Convey Hybrid-Core HC-1 heterogeneous platform and utilizing existing graphical design environments for design entry. Furthermore, incremental implementation techniques were applied to one of the flows to accelerate bitstream compilation, improving design productivity. The efficacy of these flows in extending FPGA-based acceleration to non-engineers in the life sciences was informally tested at two separate instances of an NSF-funded summer workshop, organized and hosted by the Virginia Bioinformatics Institute at Virginia Tech. In both workshops, groups of four or five non-engineer participants made significant modifications to a bare-bones Smith-Waterman accelerator, extending functionality and improving performance.

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022, and by NSF Award No. OCI-1124123, High Performance Computing in the Life/Medical Sciences.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

From measurements of physical phenomena at CERN's Large Hadron Collider (LHC) to the sequences of billions of nucleotides in the human body's DNA, we have access to an unprecedented amount of stored data. The world we live in is saturated with it, and while, at one point in time the production of sufficient data was a primary challenge, the problem has now shifted to its consumption and intelligent processing at reasonable rates. Petabytes of sensor data from the LHC and the three billion base-pair genomes of every human on the planet are just two of many examples of fast-growing, complex data sets that have been generating concern for the high-performance computing (HPC) community.

The task of compiling such massive data sets into useful information falls to *domain-experts*, scientists with domain-specific knowledge of algorithms needed for such analysis. Due to the size of the data sets, these algorithms often have extensive computational requirements, and such experts resort to the use of large, data-center class computing systems for algorithm acceleration. The rigid, coarse-grain architecture of such systems, however, is intended for high-performance over a *general mix* of problems, and cannot take advantage of many application-specific optimizations. FPGA-based, heterogeneous platforms, such as the Convey Hybrid-Core series, provide the fine-grain flexibility needed to provide optimizations specific to an application's architecture.

1.1 Motivation

Despite the benefits that heterogeneous computing has to offer domain experts, the existing tools that enable development for FPGA-platforms are highly dependent on digital design expertise, i.e. an excellent understanding of a hardware description language (HDL) and fine-grain digital hardware architecture. This impedes the ability of non-engineers to explore acceleration on such platforms, and creates a wide gap between the domain expert's knowledge of the algorithm and the vast computational capacity of FPGAs.

In addition to poor usability, most traditional flows are characterized by a lack of high productivity. Reasons for this include primitive support for IP reuse, which often results in redundant effort in the design phase, as well as limited interactivity throughout the development process which lengthens the time to solution by constraining the number of meaningful design iterations explored and verified by the user each day. These problems raise the already high barrier to entry for application development targeting FPGA-based platforms for digital designers and non-engineers alike.

While many non-traditional, text-based syntaxes have emerged in an attempt to bridge the gap between domain experts and FPGA-based, heterogeneous acceleration, they have yet to consistently produce efficient implementations without designer input of a nature requiring hardware design expertise. Some existing graphical tools present a more intuitive approach to hardware design, constraining the user to describe functionality on an inherently concurrent graphical canvas. This work takes advantage of such tools.

1.2 Contributions

This thesis attempts to address the problems stated above through the assembly of end-to-end development environments targeting FPGA-based, heterogeneous platforms, and utilizing existing, highly abstracted graphical syntaxes usable by non-engineer domain experts

without digital hardware design experience. Specifically, two development flows were created, one using the Data I/O Azido graphical environment as the flow's design front-end, and the second using the National Instruments LabVIEW FPGA platform as the front-end. Both flows are seamlessly integrated into Convey's Personality Development Kit for creating accelerators on the Convey HC-1 platform, including integration of the system simulation framework into the front-ends. To aid in the hardware/software co-design portions of development, abstractions of the Convey HC memory and custom instruction interfaces were provided in the front-end environments. Furthermore, the productivity of the Azido-based flow was improved through the use of two incremental compilation techniques for the acceleration of the flow's bitstream compilation phase.

To demonstrate the efficacy of the two flows, an informal evaluation of their usability was conducted during two summer workshops, one in 2012 and one in 2013. Each workshop, a small group of students and professionals with were given a bare-bones, systolic array-based implementation of the Smith-Waterman sequence alignment algorithm, which made heavy use of the interface abstractions provided with the flows. In these experiments, the group members were tasked with extending the functionally and performance of the implementation using one of the two flows, with limited involvement of technical experts.

1.3 Organization

This thesis is organized as follows. Chapter 2 discusses the growing concern of big-data science to the field of HPC, and how heterogeneous computing with FPGAs can address these concerns. Also discussed is the current state of development tools targeting these platforms, and the lack of such tools intended for use by non-engineer domain experts. Chapter 3 presents *bFlow* and *ConVI*, seamless, end-to-end development flows targeting the Convey HC-1 heterogeneous platform, and using Azido and LabVIEW FPGA as design front-ends. The chapter describes two informal usability experiments, in which these tools are given

to non-engineer domain experts with the task of improving a skeleton implementation of a widely-used bioinformatics algorithm. Chapter 4 contains the qualitative and quantitate results of this experiment—a discussion of the usability of the two flows, as well as the runtime results for compilation acceleration. Finally, Chapter 5 provides a summary of the motivation of this thesis, its contributions, and its various results.

Chapter 2

Background

This chapter is structured as follows. It begins with an overview of the challenge big-data science poses to the high performance computing (HPC) community, and discusses how heterogeneous computing, especially the use of FPGA-based systems, can help address this problem. Consideration is given to the poor usability and productivity of development flows targeting such systems, and the current state of non-traditional development flows, including those used heavily in this work, is described.

2.1 Big Data

The current era is one characterized by an abundance of raw facts, as the size and availability of valuable data sets is rapidly increasing. This trend is reinforced by the existence of projects like the Large Hadron Collider (LHC), which contains 150 million sensors and will produce about 25 Petabytes of data in 2013, even after discarding over 99.99% of the sensor output [4, 20, 46]. Another example—more pertinent to this work—is the construction of the genome of a single human individual, which is comprised of over three billion DNA base pairs. Producing such a genome using output from top-of-the-line next-generation DNA

sequencers involves the complex process of assembling billions of 100-200 base-pair reads into a single sequence [36]. This is especially concerning given the expectation that, in the near future, next-generation sequencing machines will produce these reads in a matter of hours [24]. This flood of information necessitates exceptional computing performance in order to process such data sets within tolerable times.

At the forefront of big-data analysis efforts are domain experts—the brains behind extracting useful information from vast data sets. These include scientists who interpret the LHC sensor output and bioinformaticians who explore genome hypotheses by analyzing large quantities of genome data [26]. Such domain experts may have significant programming skill; however, their productivity——this will be defined here as their hypothesis discovery and verification rate——is limited by the processing capabilities of available computing resources. For example, the architecture of data-center class computing platforms is designed for speed over a general mix of problems, and efficiently applying such platforms to domain-specific data structures and algorithms such as DNA/protein sequence alignment can be very difficult [37]. Heterogeneous computing machines, such as the Convey Hybrid-Core (HC) servers have potential to address this problem, and are discussed in the next section.

2.2 Heterogeneous Computing

Heterogeneous computing (HC) can be defined as the application of diverse computing resources to the acceleration of computationally intense tasks with diverse requirements [19]. Moving a computing task from general purpose processing resource to a resource specifically designed for that task or class of tasks has potential to increase execution efficiency, specifically in terms of time and space/energy savings [23]. These benefits have resulted in the widespread use of domain- and application-specific computing resources, including graphics processing units (GPUs) and application-specific integrated circuits (ASICs), for HPC problems.

Each of these resource types demonstrates a tradeoff between application performance and development time. Multicore processors are easiest to program and debug, resulting in short development times; however, such hardware executes code sequentially with very coarse-grained parallelism, failing to take advantage of significant fine-grained parallelism found in many HPC applications. GPUs, which typically perform computer graphics tasks, are much more specialized, providing great acceleration to the domain of applications that map to the same instruction/multiple data (SIMD) model, and operate on floating point data. The parallel acceleration offered by GPUs is much finer than that of multicore processors (e.g. hundreds of small processing elements vs. 4-16 large elements per unit). Finally, ASIC devices are completely specialized, and enable the highest level of application performance per space/energy due to the ultra-fine grained level of control over the physical layout of the chip. These are custom solutions, and come at the cost of very long development times, as well as the inability to make changes to the implementation after fabrication, a severe limitation when compared to the ease of recompiling software for CPUs and GPUs.

One alternative to the devices discussed above are field-programmable gate arrays (FPGAs), which provide a compromise between the programmability of software and the performance of hardware. FPGAs can be configured at a fine level of granularity to execute virtually any hardware description. This flexibility means that, when properly programmed, FPGAs outperform multicore CPUs and GPUs for many applications (see Table 2.1 for a high-level comparison of these devices). This is especially true for arithmetic involving non-standard data types, which dominate many bioinformatics applications. For this reason, and others discussed in the next section, FPGA targets are the focus of this work.

2.2.1 FPGAs

FPGAs consist of large, two-dimensional arrays of logic blocks, arithmetic blocks, and memory elements connected to each other through a highly configurable interconnect fabric. These devices can be programmed with virtually any hardware description, provided it does

Table 2.1: Comparison of multicore CPUs, GPUs, and FPGAs. MIPS/W refers to the typical power efficiency—energy consumed per meaningful computation.

Device	Granularity	Programmability	MIPS/W
Multi-core	Coarse	Software	Low
GPUs	Medium	Software	Medium
ASICs	Ultra-fine	Hardware (once)	Very High
FPGAs	Fine	Hardware	High

not require more resources than those available in the device. The performance of FPGAs, when compared to other resources, depends heavily on the application in question. For example, FPGAs will generally underperform GPUs for applications requiring SIMD operations on large, floating-point data sets, as this problem domain is the focus of GPU architectures, and FPGA implementations suffer from significant overhead due to the fabric flexibility [34]. Also, CPUs maintain an advantage over FPGAs for applications with extensive control requirements and frequent context changes, due to their architecture and much higher clock speeds. However, the abundant parallelism and fine-grain control accessible to FPGA designers permit application-specific optimizations unavailable on the rigid architectures of CPUs and GPUs, which require the use of standard, 32- or 64-bit data types for full utilization—FPGAs enable non-standard resources, such as 2-bit arithmetic units. Due to the problem-specific nature of data types found in many HPC computations in the sciences (e.g. 2-bit encoding of DNA base pairs), this work focuses on heterogeneous platforms that provide acceleration with FPGA devices.

2.2.2 Convey Hybrid-Core

Convey Computer's Hybrid-Core (HC) machines are excellent examples of FPGA-based, heterogeneous computing platforms [10]. These systems provide application acceleration

through the tight coupling of a general purpose processor and an FPGA-based application accelerator. The work presented in this paper makes heavy use of the Convey HC-1 Hybrid-Core Computer, a 2U chassis server partitioned into an Intel Xeon-based host board and a co-processor accelerator board, the focus of which is four user-programmable Xilinx Virtex 5 FPGAs. A unique configuration of these FPGAs is referred to as a "personality," and provides a set of x86 custom assembly instructions to application running on the host server.

By using a personality tailored to a particular application, significant speedup can be realized. Convey Computer has created personalities that target particular application domains, including graph problems and algorithms common to the financial sector [12, 11]. However, it is probable that existing personalities will not meet the requirements of a user's particular application; as a result, Convey has provided a development environment, the Personality Development Kit (PDK), for designing, building, and simulating custom personalities. Following is an overview of the HC-1's architecture and Convey's supported development process using the PDK.

HC-1 Architecture

The high-level architecture of the HC-1 system is shown in Figure 2.1. The bottom half of the server chassis is dedicated to a dual-socket motherboard containing a dual-core, 2.13 GHz Intel Xeon 5138 processor with an Intel 5400 memory controller hub attached to up to 128 GB of memory in 16 DIMMs. The other socket houses a mezzanine connector to join the co-processor board to the host's 1,066 MHz front-side bus (FSB). The co-processor board contains eight memory controllers connected to two DIMMs each [3, 35, 13].

The co-processor contains four user-programmable Xilinx Virtex-5 XC5VLX330 FPGAs, referred to by Convey as Application Engines (AE). Each AE FPGA connects to the rest of the system through four primary interfaces (see Figure 2.2):

- The dispatch interface handles custom instruction calls from the the scalar processor, as

well as 64-bit register transfers. The registers, known as the application engine general (AEG) registers, are typically utilized for runtime configuration of the personality.

- Each AE contains 16 ports on eight memory controller interfaces, each connected to one of the coprocessor's eight memory controllers. Assuming an ideal memory access pattern, each AE has up to a 20 GB/s link to the 16 DDR2 DIMMs.
- The Control and Status Register (CSR) interface, which connects all four AEs and the scalar processor in a ring topology, runs off its own clock, and enables access to registers in each AE. Registers on this ring can be accessed through read and write commands sent to a local telnet server provided by the MPIP program. This is also referred to as the management interface.
- An AE-to-AE interface is provided to directly support communication between neighboring FPGAs. With this interface, all four AEs are interconnected in a ring topology using 668 MB/s full-duplex links.

The co-processor board also contains two non-user programmable FPGAs, which constitute the Application Engine Hub (AEH). One FPGA handles communication with the host over the FSB, and enforces memory coherency across the host and coprocessor memory systems. The second contains the scalar processor, which implements Convey's base instruction-set and invokes the personality custom instructions through the dispatch interface.

Personality Development Kit (PDK)

To enable the creation of custom personalities, Convey provides a Personality Development Kit (PDK) flow, which is illustrated in Figure 2.3. The PDK encourages the following development process, when constructing an accelerator from an existing software-only implementation:

1. Analyze the application's performance bottlenecks and heavily-used data structures,

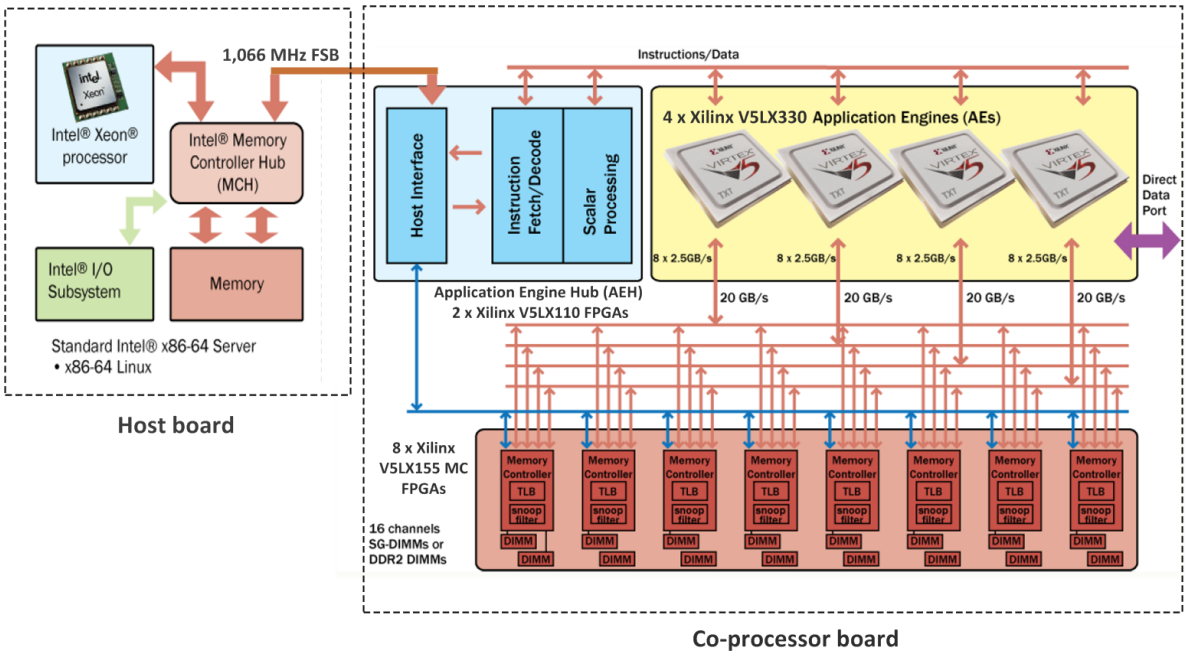


Figure 2.1: Convey HC-1 system architecture. Convey Computer Corporation. Convey HC-1 Personality Development Kit Reference Manual, v.4.1, 2011. Used with permission of Convey Computer Corporation, 2014.

and identify core functionality that can be moved to the FPGA-based co-processor for acceleration. Extract this functionality into a set of software kernels, which will serve to model the desired co-processor behavior.

2. Map the software kernels to the subset of the Convey Instruction Set Architecture reserved for custom personalities. This replaces the original calls to the kernels with calls to the coprocessor software model, allowing the designer to verify the partitioning using a functional, system-wide simulation framework provided by Convey.

3. Provide implementations of the kernels using a hardware description language (HDL), such as Verilog and VHDL, or as synthesized netlists (e.g. EDIF), and verify their functionality using the aforementioned simulation framework, which contains bus-functional models of all AE interfaces, and co-simulates the host software with the co-processor HDL through the Verilog Procedural Interface.

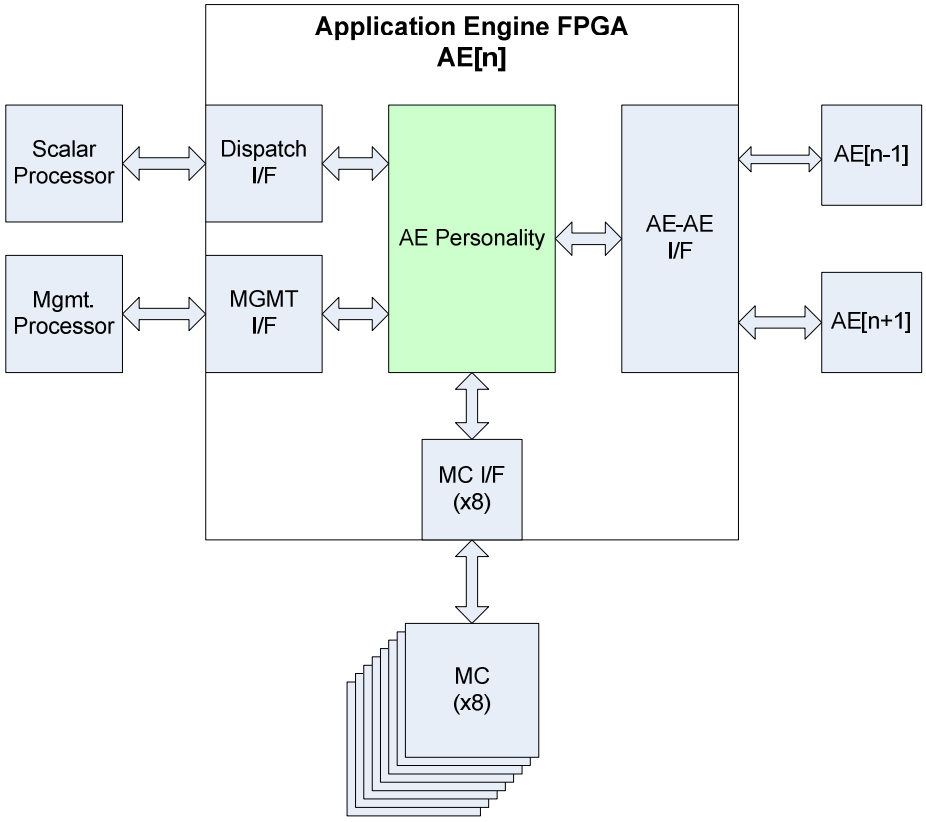


Figure 2.2: HC-1 Application Engine (AE) architecture. Convey Computer Corporation. Convey HC-1 Personality Development Kit Reference Manual, v.4.1, 2011. Used with permission of Convey Computer Corporation, 2014.

4. Lastly, the personality is implemented to a bitstream package using the Xilinx FPGA design flow. Each personality package consists of the kernel software models, FPGA configuration files for programming the AEs, and some metadata and documentation. On-demand, the package is loaded to the co-processor and its instructions made available to the host application.

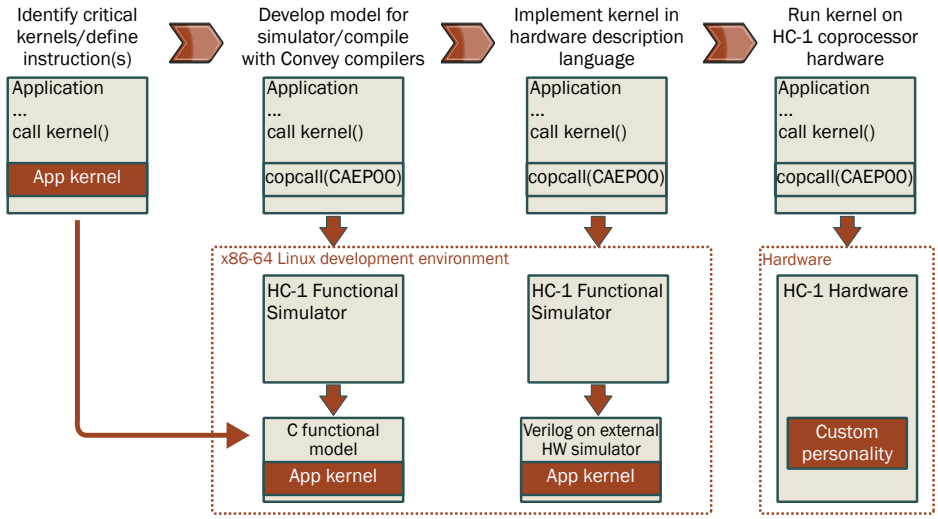


Figure 2.3: Convey's Personality Development Kit (PDK) process for developing custom personalities. Convey Computer Corporation. The Convey HC-2 Computer Architectural Overview. http://www.conveycomputer.com/files/4113/5394/7097/Convey_HC-2_Architectual_Overview.pdf. [Online; accessed 3 Dec 2013] Used with permission of Convey Computer Corporation, 2014.

2.2.3 Usability Challenges

Computing on platforms with heterogeneous resources poses several challenges. First, there is the problem of effectively partitioning the task into subtasks to run on the diverse set of resources. This is made more difficult by the vast differences between development flows for each resource type, and the inability to use a single specification language to target the entire system. Furthermore, when such platforms are used for HPC in the sciences, however, the problem of usability by non-engineers is brought to center-stage. Development flows for heterogeneous platforms like the Convey HC series require extensive computer engineering and digital design expertise, rendering the use of these platforms impractical for most experts in the science communities, who typically lack such backgrounds [21].

2.3 FPGA Design Productivity

Design productivity for FPGA-based computing has suffered from the contemporary ASIC "design productivity gap," in which the rapid rise in transistor density overwhelms the abilities of design tools and methodologies [1]. This gap especially hurts FPGA design productivity as these devices present unique needs and opportunities, such as reprogrammability (not available on ASICs). Contributors to high FPGA design productivity and challenges unique to non-engineer domain experts are discussed in this section.

2.3.1 Contributors to Productivity

In [32] Nelson et al. proposed a productivity model that exposes three key contributors to high design productivity for FPGAs. This model draws from the rich history of software development by comparison with hardware development, and defines productivity as dependent primarily on the extent of object reuse during development, the level of abstraction of the design environment, and the level of interactivity of the design tools. Following are descriptions of each of these contributors:

- *Object reuse.* The reuse of components in software has proven hugely beneficial to productivity; in fact, even the simplest "Hello World" programs depend on a huge level of reuse [32]. Applying these concepts to FPGA design leads to libraries of reusable hardware blocks and the standardization of interfaces between such blocks. When such components can be integrated into the rest of the system with significantly less effort than designing them from scratch, productivity is increased.

- *High-levels of abstraction.* The move from machine code and assembly to high-level languages such as C++ and Python has effected massive reductions in software design and verification time. Similar benefits can be had in FPGA development by elevating design granularity to a coarse level using hierarchy and object reuse, e.g. designing with

large blocks, such as FFTs, rather than individual logic and arithmetic operators. This has the positive side effect of improving the portability of designs. Furthermore, beneficial abstraction can be achieved by providing mappings from behavioral descriptions to structural descriptions—this is the focus of HLS tools.

- *Considerable interactivity in development flows.* By decreasing the time required to evaluate meaningful design iterations, or *turns*, the turns-per-day can be increased, and the total time-to-solution reduced. In FPGA development, this evaluation consists of verifying and debugging the functionality of the design, and analyzing its performance, providing valuable insight to the designer. Hence, when the turns-per-day rate is increased, the volume of beneficial feedback to the designer is increased. This is the desired *interactivity* that benefits design productivity.

2.3.2 Non-Engineer Usability

Unique to FPGA-based HPC in the sciences is the involvement of non-engineer domain experts. In addition to productivity challenges faced by digital designers, non-engineer designers face a steep learning curve presented by traditional, HDL-based tools. This barrier to entry exists for two reasons: *1*) learning an unfamiliar text-based language syntax and *2*) thinking in terms of concurrent processes, all executing simultaneously. Unfortunately, students beginning a computer science or engineering education are taught to think of an application as a sequence of steps, rather than a set of concurrent tasks [38]. This is true for engineers, and even more so for non-engineer domain experts.

2.4 Non-Traditional Development Flows

Due to the productivity and usability limitations of traditional design entry frameworks (i.e. HDL-based flows) discussed in the previous section, many non-traditional languages and

processes have emerged in an attempt to ease the use of FPGAs in heterogeneous systems by *1*) decreasing the time-to-solution for FPGA implementations and *2*) broadening the FPGA user base through the use of highly abstracted syntaxes. This section discusses the role or purpose of these tools in the development cycle, and describes the current state of such tools.

2.4.1 High Level Synthesis

High-level synthesis (HLS) is defined here as the compilation of a high-level, behavioral description language to a low, register-transfer level (RTL) hardware specification (e.g. Verilog and VHDL). These tools consume textual or graphical behavioral descriptions, which are analyzed to extract control dependencies and parallelism. By using a very high-level specification, HLS can drastically reduce the time required to arrive at a functionally correct solution when compared to RTL-based development. However, as the input description is abstracted, optimization of the hardware is shifted from the designer to the HLS compiler, often resulting in inefficient implementations. This effect can usually be mitigated by providing 'hints' (e.g. pragma statements in C-based HLS), to aid the compiler, though these hints are usually of a nature requiring digital design expertise.

Many early HLS tools made use of unconventional or tool-specific languages. In general, however, the use of a non-standard language as the entry format of an HLS tool creates a barrier to widespread adoption [25]. To avoid this, numerous attempts have been made to use existing, widely adopted languages, such as C/C++/SystemC, Java, Python, OpenCL, etc., as the design entry point. Current HLS tools that accept C/C++/SystemC as input include Catapult (formerly Catapult C from Mentor Graphics) [7], Bluespec [5], ImpulseC [18], Xilinx HLS (formerly AutoPilot from AutoESL) [51], LegUp [8], and Synphony C [40]. Examples of OpenCL HLS tools include Altera's SDK for OpenCL [15, 2]. An excellent, albeit slightly outdated summary of HLS history is presented in [25].

2.4.2 Graphical Environments

Among the vast spectrum of HLS tools one can find the subset devoted to graphical development environments. Such environments are distinguished from text-based HLS tools by the use of a graphical syntax as the entry format, rather than text. It can be argued that constraining the designer to indicating application behavior and structure graphically, rather than textually, forces him/her to think in terms of concurrent—rather than sequential—processes, which map well to the parallelism provided by FPGAs. These tools provide function ranging from simply visually capturing the hardware structure of an implementation, to capturing a highly abstracted, behavioral model of the computation to be implemented. In the following text both ends of this spectrum are discussed.

Schematic Capture

Designing a digital hardware system by placing and connecting purely structural logic blocks on a graphical, block diagram canvas is referred to as schematic capture. It is a graphical analog to structural RTL design, and shares its low level of abstraction and limitations. For example, a general disadvantage of schematic capture tools is the lack of state machine constructs to compare with the intuitive, almost C-like constructs of the Verilog and VHDL state machine syntaxes.

Beyond Schematics

Pure schematic capture tools like those discussed above do little to abstract the low level implementation details from the designer, and do nothing to lower the barrier to entry of prerequisite digital design knowledge. Furthermore, the lack of high-level abstractions results in development times as long if not longer than with HDLs. In an effort to address this, graphical tools emerged with a focus on providing an abstracted front-end to engineers and non-engineers alike in an effort to accelerate development and broaden the user base. These

tools include MathWorks Simulink [43], Xilinx System Generator [50], Data I/O's Azido [6], and National Instrument's LabVIEW platform with the LabVIEW FPGA Module [29, 30]. This list is not exhaustive. Screenshots of these tools are shown in Figure 2.4.

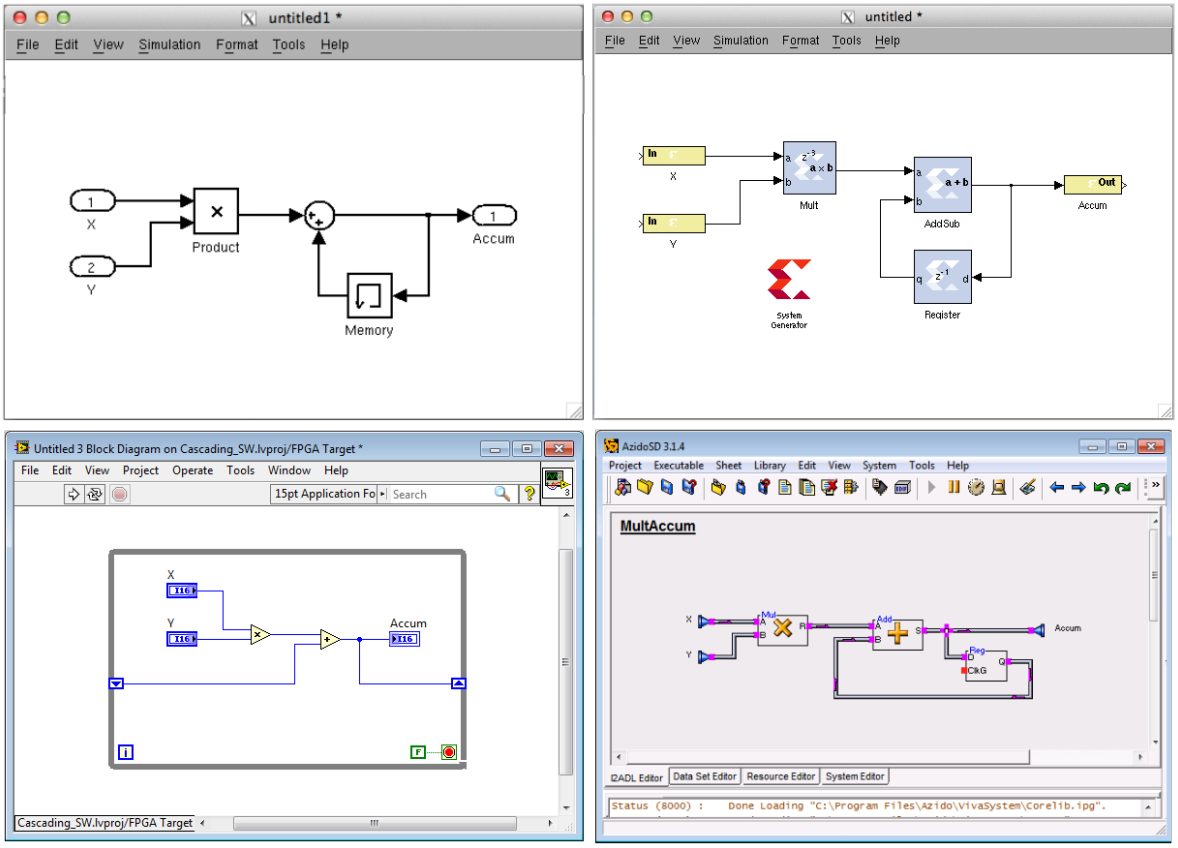


Figure 2.4: Screenshots of four graphical hardware development environments used to describe a multiply-accumulate operation. Clockwise from top-left: Simulink, System Generator, Azido, and LabVIEW.

Simulink, developed by The MathWorks, is a graphical tool tightly integrated with MATLAB, and intended for modeling and simulating dynamic systems [43, 42]. Design entry is accomplished by placing, connecting, and configuring hierarchical "sub-systems" on a graphical canvas. Using The MathWork's HDL Coder, HDL code and testbenches can be generated from a Simulink model and used to configure Xilinx and Altera FPGA devices [41]. Xilinx System Generator is used in conjunction with Simulink and HDL Coder as

a unified environment. System Generator provides a block-diagram front-end which takes advantage of the Xilinx CoreGen tool for generating high-performance computation cores [48, 50]. The abstractions shown by both Simulink and System Generator indicate a heavy focus on dynamic systems and signal processing applications.

Data I/O's Azido, which is used in this work and discussed in depth in the next section, is a graphical hardware development environment [6]. One of the features that distinguishes it from schematic capture and other graphical tools is the automatic compile-time resolution of data types. This enables graphical polymorphism, and allows recursive instantiations of an object within itself. Azido provides a large library of logic, arithmetic, and control objects called the CoreLib, and defines a framework for providing support for arbitrary target platforms. In this work Azido is used to target the Convey HC-1 coprocessor.

The LabVIEW platform, created by National Instruments, is a mature, widely adopted environment for system design, with a focus on instrument control/monitoring and data acquisition [27]. Design with the tool is very behavioral, making heavy use of synchronization and control constructs, such as for loops, timeline structures, and case statements. A behavioral block designed in LabVIEW is called a Virtual Instrument (VI), and can be run either within the environment or as an independent application. With the LabVIEW FPGA Module, these applications can be deployed to supported FPGA-based systems for accelerated computation. While some of LabVIEW's control abstractions are unavailable when targeting FPGA systems, the environment still presents a highly abstracted view to the designer, making it appropriate for use by non-engineers. As with Azido, LabVIEW is used in this work to target the HC-1 coprocessor, despite no official support for the platform.

2.4.3 Role

It is crucial to understand the role of such tools in the development process. The target users for early HLS tools were digital designers, and the purpose of the tools was to decrease time-to-solution while maintaining application performance. Few HLS tools have produced

implementations with a quality level high enough to draw such users from familiar, traditional flows [25]. Current tools also target non-engineers, in the hope of broadening the FPGA user base by abstracting away the nasty low-level details of hardware design.

The typical approach taken by a non-engineer domain expert to accelerate a big-data computation in the sciences is to present the problem to a hardware engineer along with a specification and set of constraints. The engineer implements the accelerator as per the specification, and it is inserted into a scientific workflow used by the domain expert [14]. When providing the domain expert with direct access to hardware and an abstracted, high-level development environment, two scenarios can occur.

1. The domain expert independently describes the computation using the tool, and correct implementation is produced. This requires the tool to feature a syntax usable to the user without significant training. Furthermore, the tool must be capable of producing an efficient, high-performance implementation from the abstracted syntax.

2. The domain expert fails to independently describe the computation in the tool in a manner that produces a functionally correct or efficient, high-performance implementation. However, through collaboration with a hardware engineer, the expert can use the tool to produce such an accelerator.

In this work, both situations are experienced. The former is ideal, but unlikely. In the second scenario, the tool acts as a communication device, facilitating a mutual understanding among the domain expert and engineer of the computation and desired acceleration. This supports the interdisciplinary effort to produce a correct, high-performance implementation.

2.5 Azido

Data I/O's Azido, shown in Figure 2.5, is a graphical, object-oriented design environment based on the Implementation Independent Algorithm Description Language (I2ADL) [6].

Like most graphical front-ends, Azido attempts to abstract the low-level complexities of digital hardware design to higher-level algorithmic objects more intuitive to designers without traditional digital design expertise (e.g. knowledge of HDL). Also, it simplifies and accelerates design by heavily encouraging object reuse and providing an extensible library of I2ADL primitives, known as the CoreLib. Azido can be extended to support virtually any hardware target through the use of system descriptions (SDs).

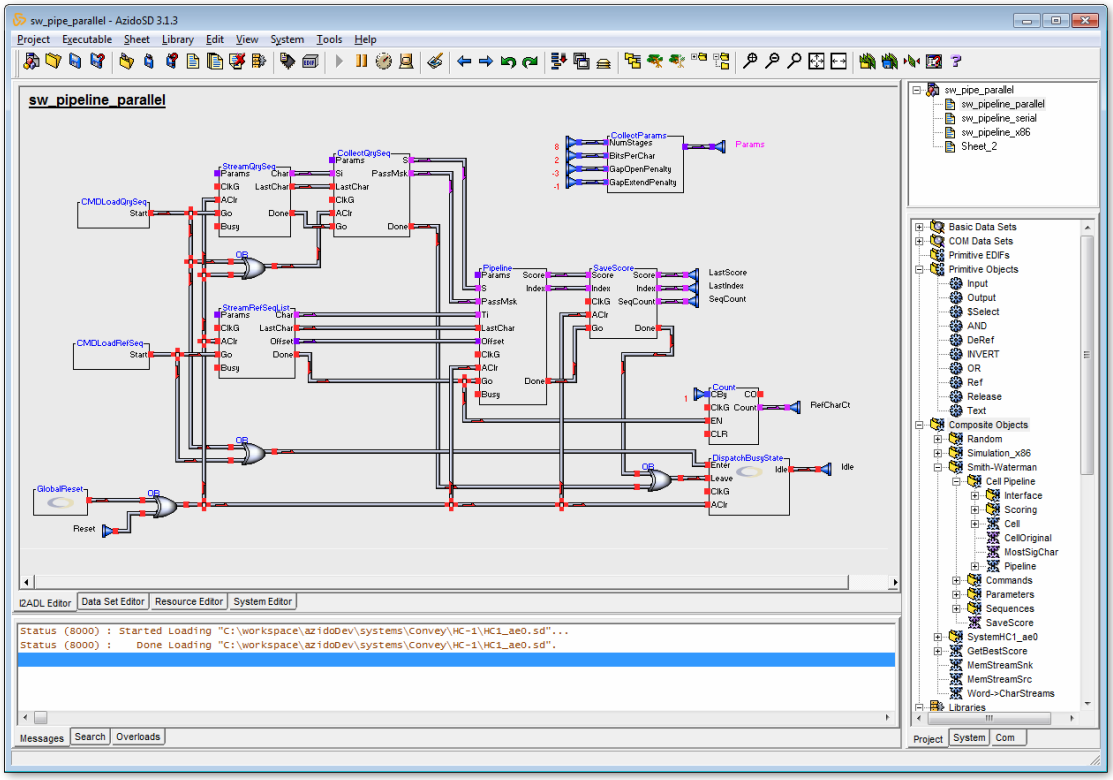


Figure 2.5: Screenshot of the Azido desktop environment.

Development in the Azido environment for a heterogeneous target platform is structured as follows. First, the user indicates the desired functionality on the top-level canvas using the graphical, I2ADL syntax. For this step the reuse of existing objects in the CoreLib is highly encouraged. By selecting the x86 system description and running the design, the user can perform a rough verification of the implementation. Finally, the target system description can be selected, the design synthesized for that target, and the implementation run with

real data. All top-level inputs and outputs not tied to real system I/O will be displayed and controlled in the widget window, shown in Figure 2.7.

Among several existing graphical design environments, Azido was selected for this work primarily for three reasons: *1*) it provides a flexible design environment and core library capable of servicing many application domains at multiple levels of abstraction, *2*) the system description-based implementation framework facilitates extension of the tool to many target platforms, and *3*) beyond standard schematic-capture abilities, Azido provides some dynamic features, such as automatic data typing and graphical polymorphism. The graphical polymorphism is enabled by unspecified (in terms of size) data types, and allows the recursive instantiation of an object within itself, as shown in Figure 2.6.

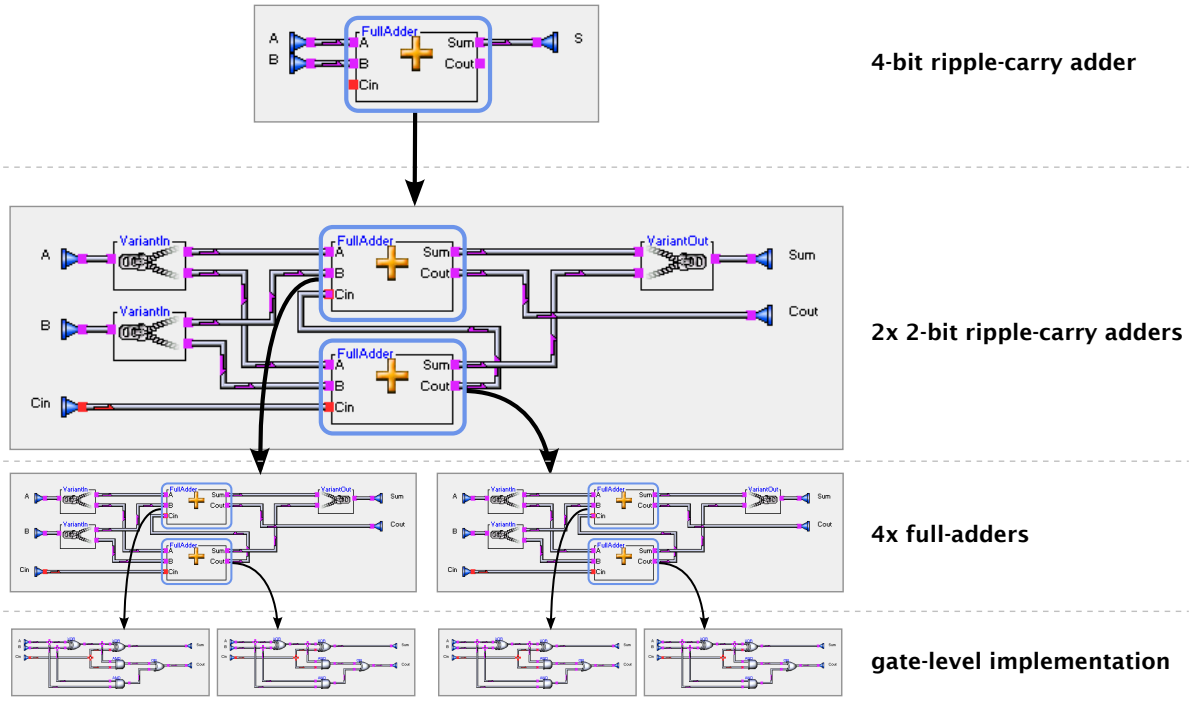


Figure 2.6: Recursive self-instantiation of an object using Azido's I2ADL syntax. This example shows a 4-bit ripple-carry adder.

2.5.1 System Descriptions

At the core of the Azido implementation engine are system descriptions. These are script-based "plugins" to Azido that encapsulate details of the target's architecture and implementation process, which is shown in Figure 2.8. SDs provide the following primary functions:

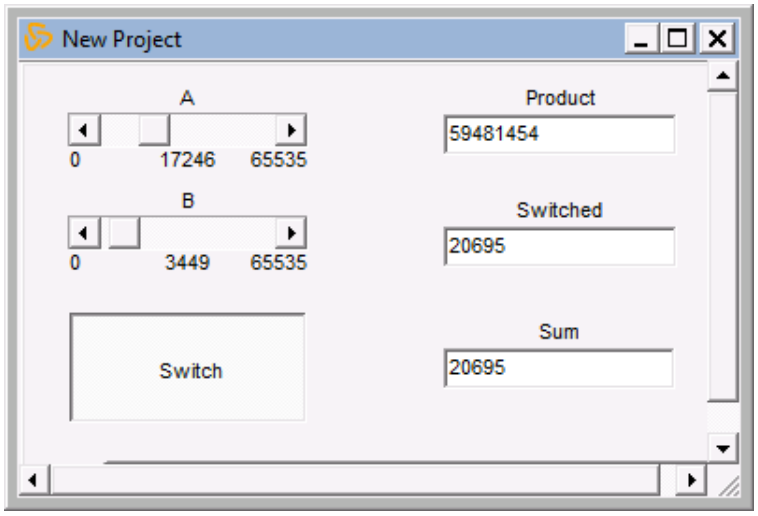


Figure 2.7: Azido widget window displayed at runtime for stimulation and diagnostic purposes.

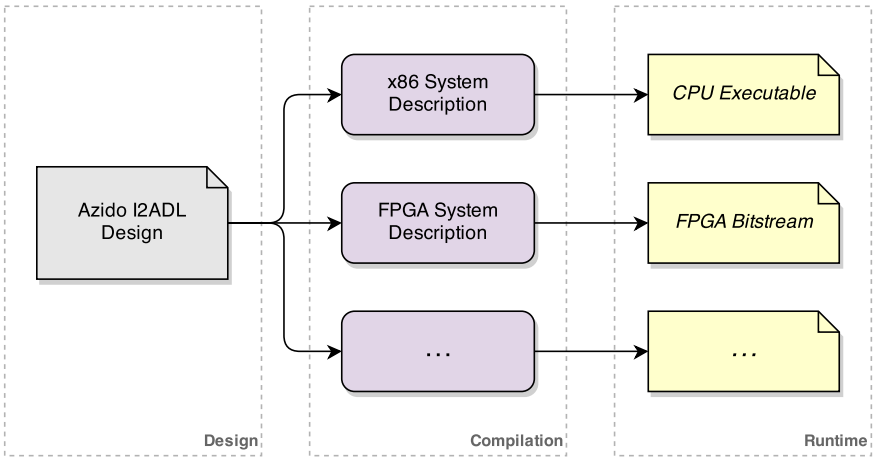


Figure 2.8: Azido system description-based implementation flow.

- Implementation flow. At compile time, Azido synthesizes the user's I2ADL design into

a structural EDIF (Electronic Design Interchange Format) netlist, which is passed to a implementation script defined by the SD. This script produces a runtime executable or configuration—a software binary for a CPU target, a configuration bitstream for an FPGA target, etc.

- Low-level definitions of any CoreLib objects. Azido synthesizes the design by traversing the I2ADL graph recursively and flattening each object into its sub-objects. When the process encounters an object without a definition (i.e. a black box), it is placed into the synthesized netlist and connected accordingly. For example, the CoreLib adder object is, by default, defined as the combination of two half-size adders connected in ripple-carry fashion. During synthesis, the adder is elaborated recursively until the "leaf" objects (i.e. bit-wide full-adders) are reached. Hence, after elaboration, a 32-bit, integer adder will synthesize to 32 full-adders connected in series. The strategy Azido uses when synthesizing an object can be directed using a graphical equivalent of compiler hints or pragmas.

- Target-specific objects that supplement the CoreLib. These objects are used to represent the top-level inputs and outputs of the target, such as physical switches and indicator LEDs, or interfaces to memory or other system components.

- Inter-platform communication implementers. To enable the use of multi-system targets, Azido requires the definition of communication handlers for data transfers between platforms. At a minimum, the SD must define an x86 $\leftrightarrow$ target communication implementer for use by the runtime widget window (see Figure 2.7).

2.6 LabVIEW

The Laboratory Virtual Instrument Engineering Workbench (LabVIEW) is a mature graphical programming platform developed by National Instruments [29]. The platform is intended primarily for data acquisition and instrument control and monitoring, and provides support

for thousands of interfaces and hardware devices. LabVIEW code can also be deployed to other platforms, including embedded platforms and FPGA-based machines. In this work, LabVIEW is used to target the Convey HC-1 accelerator, despite no official support for the platform.

LabVIEW uses a graphical syntax, called the G programming language, which enables design through the drag-and-drop placement and interconnection of behavioral blocks in a flowchart-like manner [28]. The language is used to design Virtual Instruments (VIs), functional blocks that can be run independently or instantiated in a higher-level VI. Each VI must have a block diagram (see Figure 2.9) and a front panel (see Figure 2.10). The former specifies the behavior of the VI, while the latter acts as an interface to the VI at runtime. While programming a VI, designers have access to an extensive library of reusable arithmetic, logic, and higher-level behavioral blocks.

Alongside the standard development environment is the LabVIEW FPGA Module, a plugin that enables LabVIEW to target *supported* FPGA-based platforms [30]. The G graphical language is used when programming accelerators for FPGA targets; however, because some of LabVIEW's convenient control constructs and flexible data types cannot be mapped efficiently to FPGA fabric, they are not enabled when developing in the FPGA module. These features include arrays of variable-size at runtime, multidimensional arrays, and shared variables [31]. Nonetheless, the tool maintains a high level of abstraction appropriate for non-engineers.

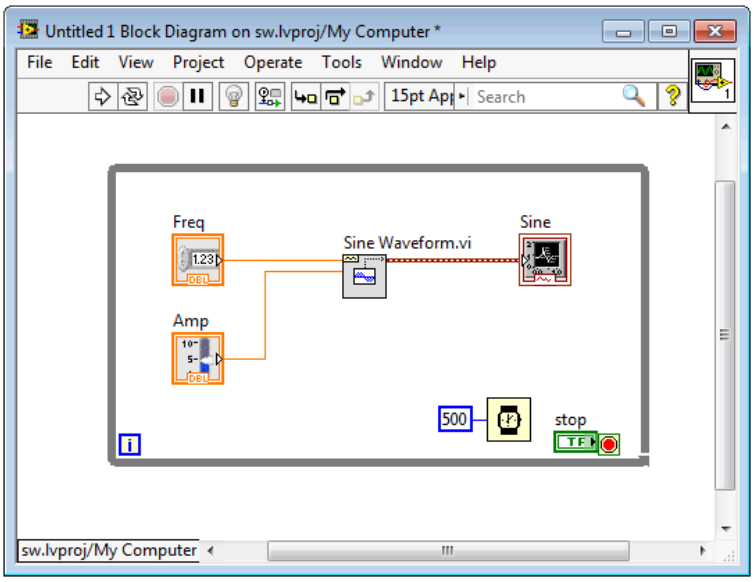


Figure 2.9: LabVIEW block diagram. This VI generates a sine wave using parameters from the front panel (see Figure 2.10).

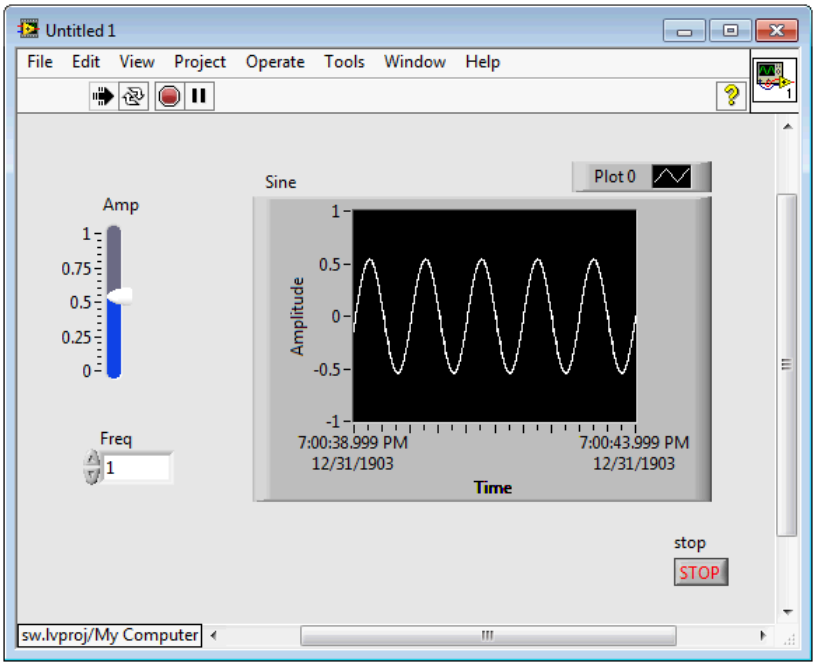


Figure 2.10: LabVIEW front panel. This VI uses controls and indicators to generate and display a sine wave (block diagram available in Figure 2.9).

2.7 Summary

This chapter introduced the increasing reality of big-data in the sciences, and discussed the challenges it poses to the field of HPC. These concerns can be addressed through the use of FPGA-based heterogeneous systems, and one such system, the Convey HC platform, is used in this work. However, these platforms provide acceleration at the cost of productivity and usability, especially for non-engineer domain experts; hence, non-traditional development flows, especially graphical programming front-ends, have emerged to tackle these problems through the use of interactive design environments and high-level abstractions. Two such graphical front-ends, Azido and LabVIEW FPGA, provide features essential to non-engineers targeting systems like the HC-1; hence, this work involves the integration of these two environments into development flows targeting the Convey platform.

Chapter 3

Approach

The approach taken in this work is divided into two distinct efforts to assemble a seamless, end-to-end development flow for the Convey HC-1 platform using an existing FPGA design front-end environment. The first effort involves integrating the Azido desktop tool into such a flow, and accelerating the bitstream compilation process to improve interactivity, as defined in Section 2.3.1. The second effort utilizes the LabVIEW platform as the front-end but, due to time constraints, acceleration of the bitstream implementation was not pursued. Both flows have the high-level architecture presented in Figure 3.1. Development and rough functional verification of the accelerator is performed in a desktop environment (i.e. Azido and LabVIEW). The host application, which runs on the HC-1's x86 server, is developed separately in a Linux environment. System-wide simulation of the host application with the accelerator is performed on the Convey platform itself. Lastly, the accelerator can be deployed as a personality, and utilized by the host application in a production context. During simulation and production runtime, the front-end is utilized for monitoring and providing diagnostic stimuli to the accelerator.

Each flow was used to implement the Smith-Waterman algorithm, a common sequence alignment computation in the life sciences. They were then informally tested in the context of a two-week, NSF-sponsored Summer Institute workshop organized by the Virginia Bioin-

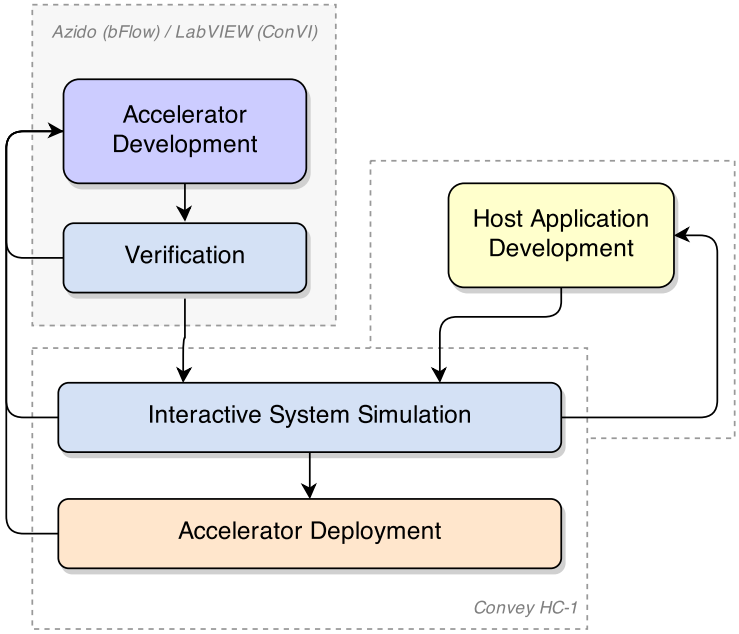


Figure 3.1: High-level development process of the *bFlow* and *ConVI* flows.

formatics Institute (VBI) at Virginia Tech [44]. This section contains the details of each effort and the resulting Azido- and LabVIEW-based flows, referred to as bFlow and ConVI, respectively, as well as a short overview of the Smith-Waterman implementations used in the workshops.

3.1 Azido Flow (bFlow)

The first attempt to assemble such a flow is based on the Azido algorithm design environment, which is discussed in Section 2.5. In *bFlow*, Azido I2ADL designs are synthesized for system simulation or runtime, and are integrated into the HC-1 personality architecture. The assembly of *bFlow*, which is visualized in Figure 3.2, is divided into three sub-efforts: *1*) development of an Azido system description (SD) to enable targeting the Convey HC-1 coprocessor, *2*) construction of a small software library, and *3*) acceleration of the back-end bitstream compilation process using incremental compilation frameworks. Each are described

in this section.

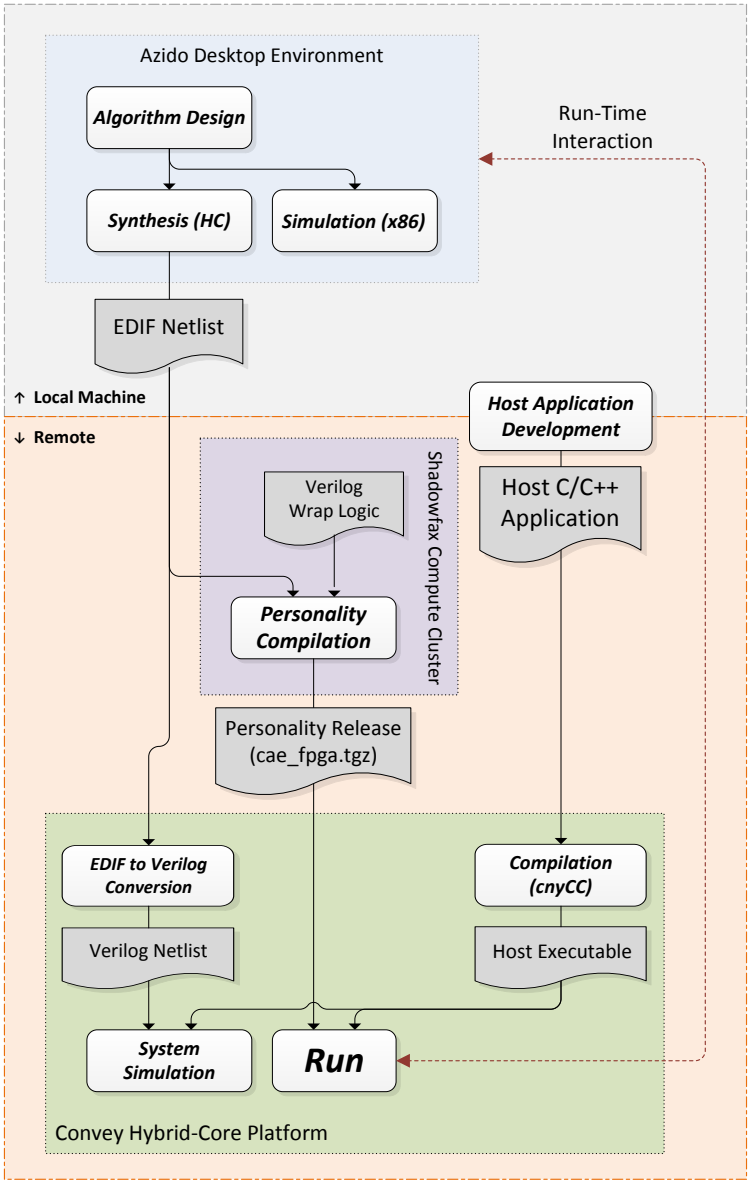


Figure 3.2: *bFlow* development process.

3.1.1 System Description

The Azido SD for the Convey HC-1 consists of a communication implementer responsible for transferring probe data and stimuli between the Azido front-end and the accelerator

during runtime, and two sets of abstractions, one for the HC-1 dispatch interface and one for the memory controller interface. Furthermore, the SD is responsible for implementing a synthesized netlist provided by Azido into an executable (or simulatable) format. In *bFlow*, the top-level netlist generated by Azido is integrated into AE architecture as shown in Figure 3.3.

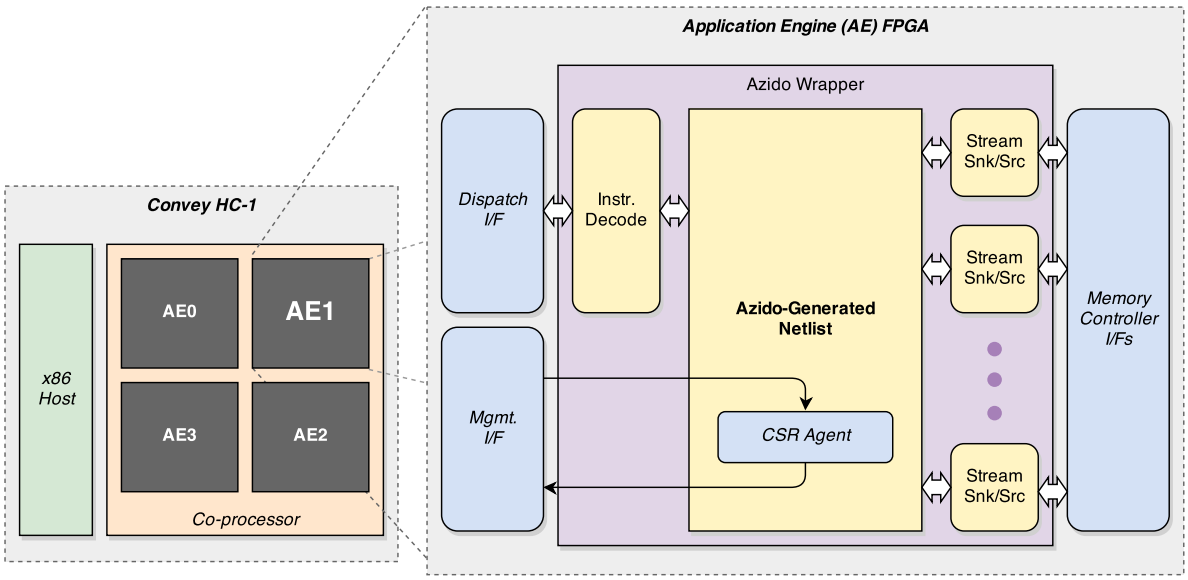


Figure 3.3: Integration of the top-level Azido I2ADL design into the existing HC-1 HDL framework.

Communication Implementer

The first component is invisible to the designer, and handles all communication between the Azido front-end and the HC-1 platform at runtime, to provide real-time diagnostic probing data and stimulation of the running accelerator. These probe and control points are indicated in the design using top-level input and output horns. At runtime, these top-level ports are shown in a pop-up window, as shown in Figure 3.4.

To enable this communication, the SD must provide a definition of the x86CPU $\leftrightarrow$ PEn interface object, by which data is passed back and forth between the user's desktop (x86)

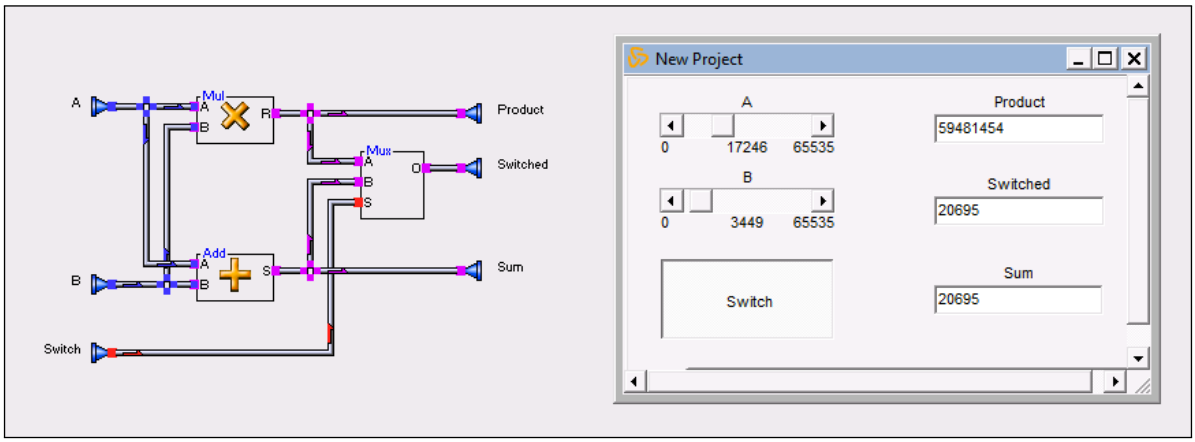


Figure 3.4: Azido top-level control and indicator widgets. Top-level ports in the Azido design on the left are mapped to widgets in the pop-up window on the right. This example makes use of the slider, button, and text box widgets.

and the Convey platform (PEn) at runtime. For the HC-1 SD, this definition contains a Windows COM object interface. The COM object passes data through an SSH tunnel to a relay server running on the HC-1 host server, which in turn uses the management interface (MPIP telnet server) of the coprocessor to read and write AE CSR registers, which capture and excite the aforementioned top-level signals. This architecture is shown in Figure 3.5.

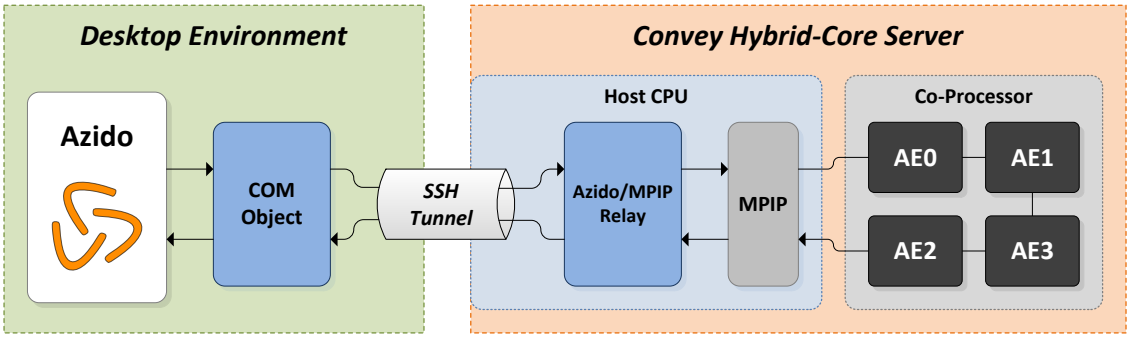


Figure 3.5: Overview of the Azido ↔ HC-1 runtime communication implementer.

Dispatch Interface

The first set of abstractions included in the HC-1 SD are of the AE dispatch interface, which is described in Section 2.2.2. This interface handles AEG register transactions, personality instruction calls, and informs the host of the AE execution status. Hence, abstractions for all of these tasks are provided to the user as library objects (see Figure 3.6), and can be placed anywhere through the I2ADL design. Three block types are provided:

- 64-bit register blocks provide read/write access to the AEG registers. Their behavior depends on how they are connected in the Azido block-diagram—if the read port is connected, they can be written by the host and read by the AE, while if the write port is connected, they are read by the host and written by the AE.
- Command trigger blocks have boolean outputs that are pulsed high when the host invokes the corresponding personality custom instruction.
- Finally, a status block enables the AE to inform the host that execution of a custom instruction has completed.

Considering the AE as a state machine with two states, *busy* (i.e. executing) and *idle*, the command and status objects precipitate transitions between the two states. For example, the host invokes custom instruction 3, which causes the AE to enter the *busy* state, pulsing the *Start* output of the *Cmd03* object. When the AE completes the computation associated with that instruction, it indicates completion by pulsing the *Cmplt* input of the *ExecDone* object, causing the AE to transition back to the *idle* state.

Management of theses states, as well as the AEG register contents, is performed by glue logic external to the hardware module generated from the Azido I2ADL design (see the instruction decode block in Figure 3.3). The I2ADL design connects to this logic through top-level ports designated as non-widget I/O. As a result, these blocks are simply wrappers providing connections to such top-level ports.

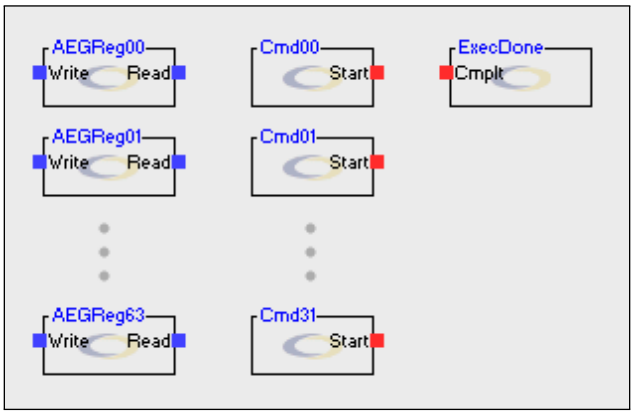


Figure 3.6: HC-1 AE dispatch interface functionality as Azido library objects.

Memory Streams

Lastly, to conceal the complexities of random memory access and address arithmetic, *bFlow* contains a simplified, streaming abstraction to the memory controller interfaces available to each AE. Two blocks are provided to the designer: a source streamer and a sink streamer. Both operate on a sequence of data memory words, starting at a specified address—the former reads the stream from memory while the latter writes the stream. A usage example, shown in Figure 3.7, demonstrates the use of these objects to invert a sequential block of memory, and complies with Azido's Go-Done-Busy-Wait (GDBW) flow control methodology. Note that in early iterations of *bFlow*, both streaming and standard random-access memory objects were provided to the designer; however, because random-access memory was not required during the VBI workshops and other demonstrations, those objects were removed. Memory streamers are provided for each of the eight AE controller interfaces, and the control logic mapping the streaming interface to the standard, random-access memory controller interfaces is contained in the Azido wrapper (see Figure 3.3). Thus, as with the dispatch interface objects, the memory streamers are essentially wrappers containing connections to top-level ports.

The decision to provide a streaming abstraction of the memory interface is based on its simplicity and intuitive behavior, and justified by the prevalence of applications in big-data

HPC which can be mapped to a stream-based processing model (e.g. pipelined, SIMD processing). It is especially appropriate for the Smith-Waterman application used by the students during the VBI Summer Institute workshop, as the algorithm involves reading large, sequential blocks of memory.

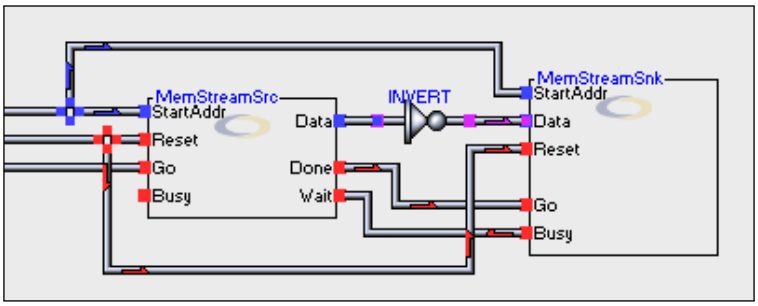


Figure 3.7: HC-1 AE memory controller interface functionality provided as Azido library objects. This example demonstrates the streaming access abstraction, as well as Azido's Go-Done-Busy-Wait flow control scheme.

3.1.2 Software Routines

In *bFlow*, the Azido front-end is used generate only the coprocessor configuration, or personality. The host app must be developed separately in the Convey host Linux environment, and verified with the Azido-generated accelerator implementation in system simulation.

Convey provides a set of low-level software and assembly routines to the host application for loading, initializing, and running the personality. Core functionality provided by these routines includes reading and writing the AEG registers, transferring data between the host and coprocessor memory subsystems (while these are coherent, performance can be preserved by moving large blocks of memory to the coprocessor before streaming it through the AEs), and, of course, invoking the custom instructions handled by the personality. To simplify the configuration and execution of the personality, several convenient wrappers are provided to the host app programmer in the form of a C++ class called *CPLib*, which is detailed in Table 3.1, and partially listed in Appendix A. This library is used in both the *bFlow* and

Table 3.1: List of software routines provided to the host app developer.

Method	Description
readAEGReg()	Read an AEG register (64-bit) on one AEs.
writeAEGReg()	Write an AEG register (64-bit) on one or all AEs.
mallocCP()	Allocate a chunk of coprocessor memory and [optional] fill it with the contents of the passed buffer.
execCPInstr()	Execute the custom personality instruction with the given index.
writeReg()	These routines are used only in the *ConVI* flow, and provide the same functionality as those above, but in a simpler format that maps well to the LabVIEW dispatch interface abstractions.
readReg()	
execCmd()	

ConVI flows.

3.1.3 Incremental Compilation

Acceleration of the bitstream compilation process is achieved by two methods, or tools: Xilinx Hierarchical Design Partitions flow [49] and *qFlow* [16]. Both are incremental, partial implementation frameworks that reduce build times through high-level management of the Xilinx ISE implementation process. The key to both approaches is the exploitation of the high-level architecture of all Convey personalities—that is, the inclusion of interface logic that remains nearly static throughout the development process. This logic consists of interfaces to the eight memory controllers, a memory crossbar, and *bFlow*-related logic that communicates with the dispatch and management processors. All of this consumes roughly 25% of each of the HC-1's AEs (Xilinx part XC5VLX330) and, when using Convey's traditional compilation process, is re-implemented each build, costing precious minutes of development time. Given the rarely-changing nature of this logic, both of the following strategies accelerate compilation by implementing it once, constraining its placement to the edges of the FPGA near the I/O,

and then preserving it's placement and routing during consecutive builds of the personality. For these consecutive builds, the dynamic, Azido-generated netlist is placed in a "dynamic region" at the center of the device.

Xilinx Partitions

The first approach taken makes use of the Xilinx Hierarchical Design Partitions flow [49]. Two partitions were selected for this flow: *1*) a top-level partition containing the entire FPGA design and *2*) the Azido-generated logic. After an initial compilation, the top-level is preserved, and remains mostly static as the Azido accelerator logic evolves. This top-level is re-implemented only when major changes to memory access patterns are needed (i.e. changing the memory controller interface configuration). The application of this flow is simple, requiring only some extensions to the Convey PDK makefiles and some changes to a few .ucf constraint files.

qFlow

This second utilizes a subset of the *qFlow* framework [16], a tool for accelerating back-end compilation of designs with hierarchical structure similar to that enforced by the Convey PDK. Though *qFlow* offers a superset of the functionality provided by Partitions, the application discussed here is similar. When compared to the partitions-based approach discussed above, *qFlow* provided generally much faster compilation times (see Section 4.3.1); however, as *qFlow* is a research product, it was much more difficult to work with.

3.2 LabVIEW Flow (ConVI)

The second effort toward the assembly of a seamless flow targeting the HC-1 began with the LabVIEW platform, which is discussed in Section 2.6. The complete *ConVI* flow is similar to

bFlow, providing similar low-level hardware abstractions of software wrapper library to the programmer. There are two key differences, however, which are *ConVI*'s support for front-panel control and monitoring during system simulation on the HC-1 host (not restricted to only runtime, as is the case with *bFlow*), and its lack of compilation acceleration via the incremental techniques discussed in Section 3.1.3. The primary steps taken to implement the flow include *1*) constructing abstractions for low-level hardware similar to those provided in *bFlow*, *2*) incorporating the Convey HC compilation framework into LabVIEW's compilation process, and *3*) enabling front-panel control during runtime and simulation through a register access framework. The latter two efforts are detailed in this section.

3.2.1 VI Compilation

The LabVIEW FPGA does not provide official target support for any of Convey's HC systems. Support for the HC-1 was added by modifying an existing target package, specifically that for the Spartan 3E Starter Kit. Since the VI generated by LabVIEW is instantiated as a VHDL module within the HC-1 HDL hierarchy, the XML files containing the Spartan 3E FPGA device I/O were modified to reflect the ports available to the module. The integration of the VI in the AE architecture is shown in Figure 3.8.

LabVIEW FPGA includes a instance of Xilinx ISE for compiling VIs into FPGA configuration bitstreams. During assembly of *ConVI*, this implementation process was replaced by a set of scripts that extract the VI functionality and insert it into the AE's HDL hierarchy at compile time. This is seamlessly integrated into the LabVIEW build process, such that the designer can remain in the friendly LabVIEW environment despite the outsourcing of the compilation process (see Figure 3.9).

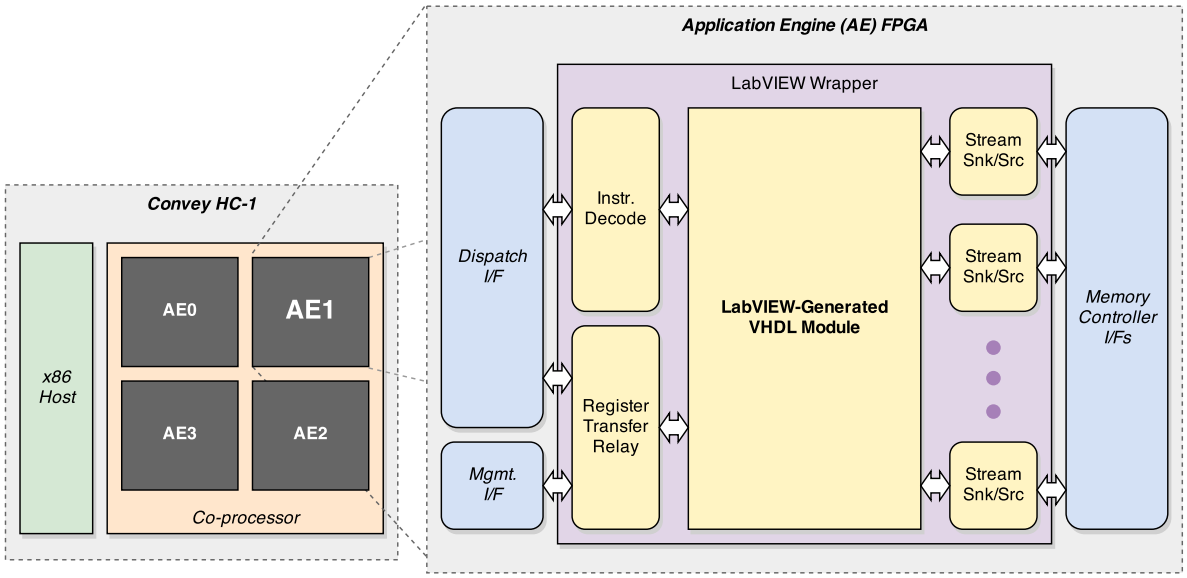


Figure 3.8: Integration of the top-level LabVIEW VI into the existing HC-1 HDL framework.

3.2.2 Front-Panel Control

The manner in which LabVIEW transfers control and monitor data between the front-panel and the running FPGA-based target varies by system. For the Spartan 3E target, such communication is accomplished by a local instance of the Xilinx Cable Server, which connects to the device through its JTAG interface. Using this connection, LabVIEW reads and writes hardware registers tied to the control and indicator ports in the VI, providing runtime control to the user.

To achieve the same level of interactivity and seamless behavior for the remote HC-1 in *ConVI*, the LabVIEW front-panel was integrated into the simulation and runtime contexts of the Convey platform using a client-side server. During simulation and runtime, this server ships these register transactions to a relay script on the remote HC-1 host, which uses the MPIP server to send the transactions through the AE's CSR interface. However, since the CSR interface is *not* included in the system simulation framework, an emulator of the MPIP server was added to the software framework for host application development. Thus, during simulation, this emulator is run alongside the host application, makes a connection to the

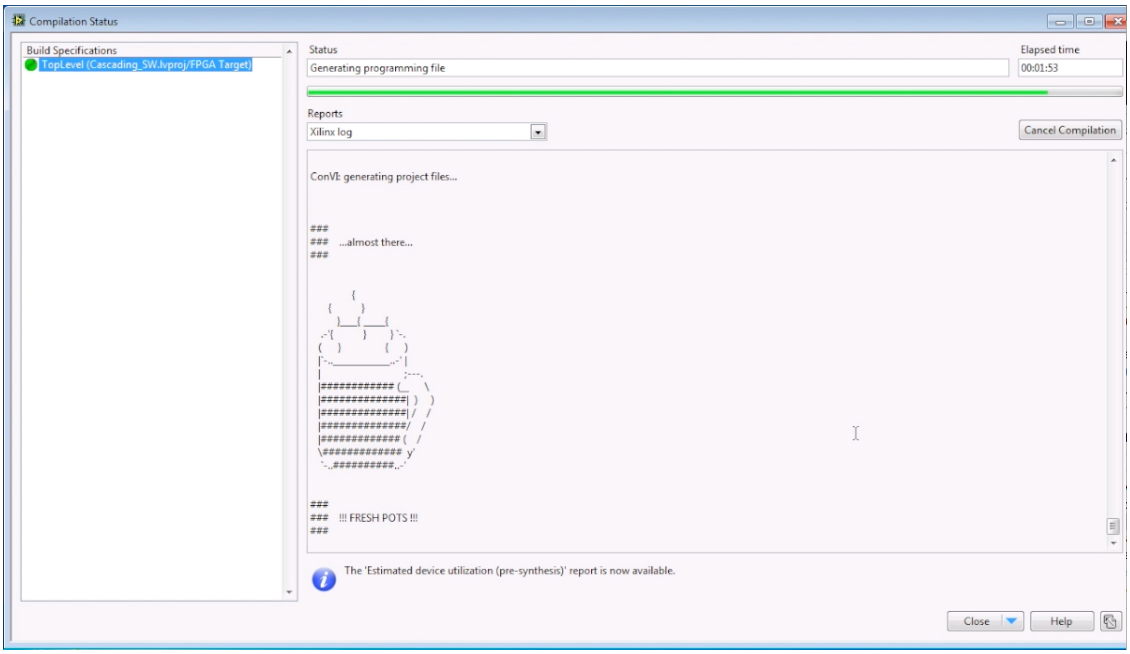


Figure 3.9: LabVIEW compilation window during a *ConVI* build. Progress is indicated by the level of coffee in the mug.

relay script, and tunnels the relayed MPIP commands through the dispatch interface to the AE. Finally, using some simple VHDL glue, the transactions are executed through a register access interface provided by LabVIEW's generated VHDL module. This architecture is better explained visually, and is shown in Figure 3.10. Note that most of this work was performed by Ramakrishna Chakri as part of his Master's Thesis [9].

3.3 Smith-Waterman

The Smith-Waterman (SW) algorithm is a technique for comparing and aligning two sequences with maximum sensitivity. Proposed in 1981 to detect similar regions and compute optimal alignments of two sequences of DNA nucleotide or protein data, it is still widely used today [39]. Figure 3.11 shows the SW alignment of two short DNA sequences.

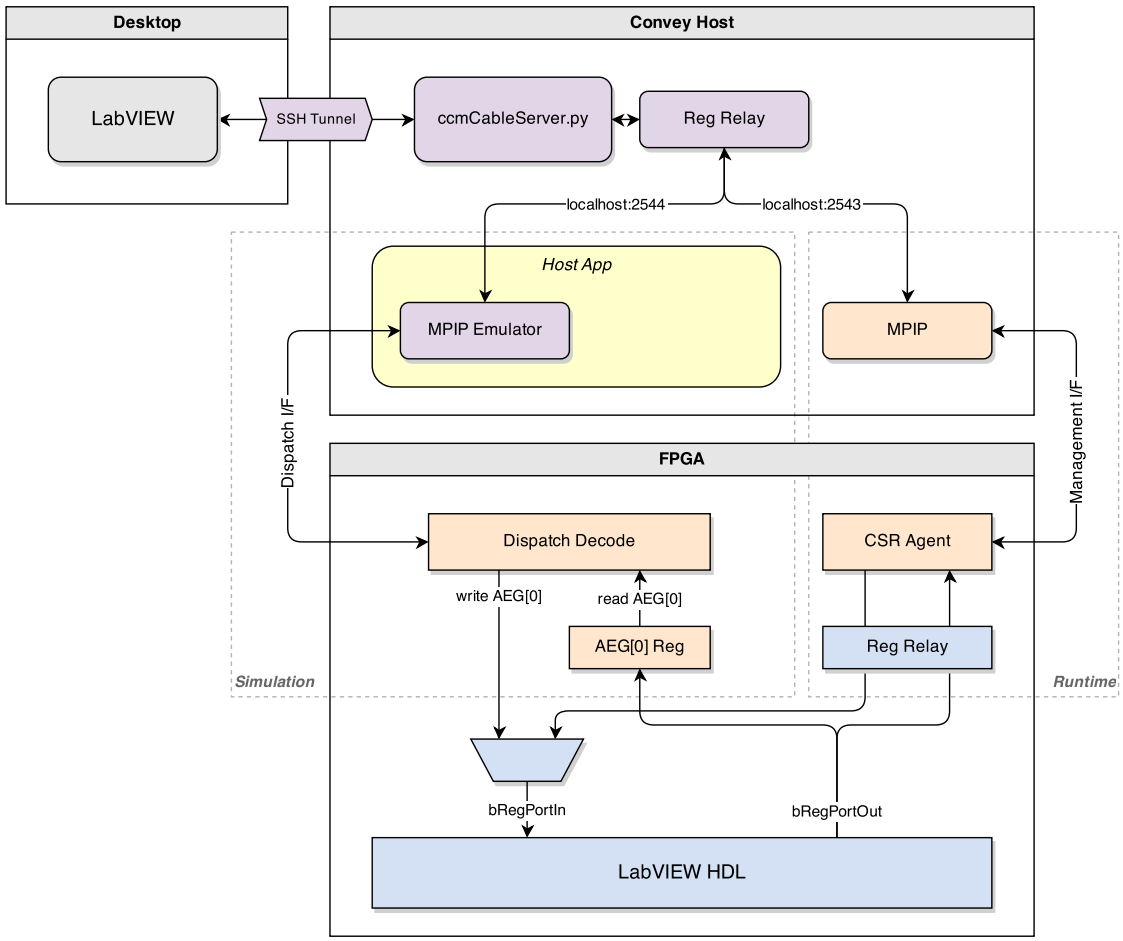


Figure 3.10: Architecture of the register access framework that enables front-panel control during runtime or simulation.

3.3.1 Overview

The proposed method has two steps: a matrix fill operation and a traceback operation. Given two sequences S and T with respective lengths m and n, the proposed method executes two steps: a matrix fill operation and a traceback operation. The former involves filling a matrix $V(i, j)$ of size $m \times n$. This is the *scoring matrix*, in which each cell contains the score of the 'best' alignment of the two sequences that ends at the point given by the cell coordinates, i.e. the value of cell (x, y) indicates the score of the best alignment of $S[0..x]$ and $T[0..y]$. This fill operation, augmented by Gotoh in 1982 [17], considers three sources of difference between

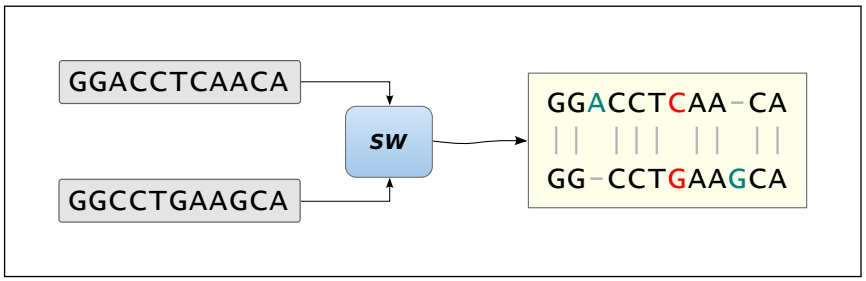


Figure 3.11: Example of Smith-Waterman (SW) algorithm used to compare two DNA sequences. Note the mutations and gaps present in the resulting alignment.

the two sequences: the *insertion* of a base into one sequence, the *deletion* of a base from one sequence, and the *mutation* of a base in only one sequence. Hence, in the alignment of two sequences, a point of mismatch results in either *1*) the creation or extension of a gap in one sequence or *2*) the labeling of the mismatch as a mutation. The example in Figure 3.11 demonstrates gap creation, gap extension, and mutation.

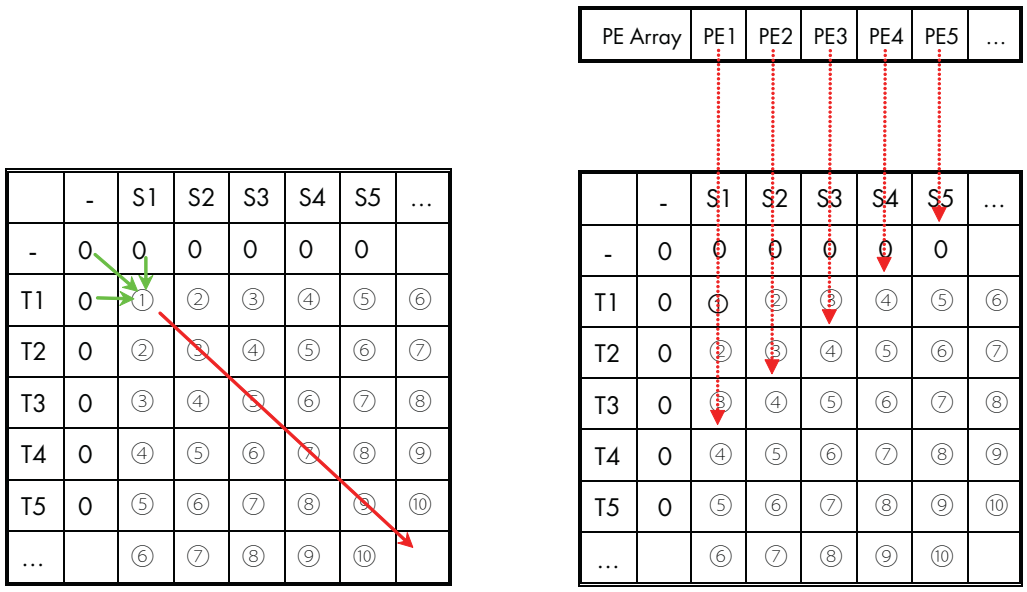
The scoring matrix is filled per the recursive definition given in Equation 3.1, and the resulting data dependencies are shown in Figure 3.12a. In this definition, $V(i, j)$ is a function of $E(i, j)$ and $F(i, j)$, which incorporate the cost of opening and extending gaps in S and T, respectively. α is the cost of opening a gap, β is the cost of extending a gap, and $\sigma(s, t)$ is the substitution score for s and t. For DNA nucleotides, this is often defined as constant

match/mismatch penalties (e.g. $\sigma(s,t) = 5$ if $s == t$, otherwise $\sigma(s,t) = -4$).

$$V(i,j) = max \begin{cases} 0 \\ E(i,j) \\ F(i,j) \\ V(i-1,j-1) + \sigma(S[i],T[j]) \end{cases} \quad ,1 \leq i \leq n, 1 \leq j \leq m \tag{3.1a}$$

$$E(i,j) = max \begin{cases} V(i,j-1) - \alpha \\ E(i,j-1) - \beta \end{cases} \quad ,1 \leq i \leq n, 1 \leq j \leq m \tag{3.1b}$$

$$F(i,j) = max \begin{cases} V(i-1,j) - \alpha \\ F(i-1,j) - \beta \end{cases} \quad ,1 \leq i \leq n, 1 \leq j \leq m \tag{3.1c}$$



(a) (b)

Figure 3.12: Smith-Waterman scoring matrix data dependencies (a) and systolic array mapping (b). The earliest possible time step at which a cell's value can be computed is indicated by the number it contains. Source: [52].

When the end location of a high-scoring alignment is identified, the algorithm can perform *traceback*, in which the whole alignment is traced in reversed, moving from the end point

to the alignment start by following a set of rules. This step is control-intensive, and not ideal for FPGA acceleration; furthermore, in practice, the matrix fill accounts for the vast majority of the execution time. Because of this, only the first step was offloaded to the HC-1 coprocessor.

3.3.2 Implementation

Because of the data dependencies in the scoring matrix (see Figure 3.12a), up to m cells on a single anti-diagonal can be computed at one time step. Taking advantage of this, acceleration of the matrix fill step is performed with a systolic array that maps to the anti-diagonals as shown in Figure 3.12b. This results in "wave-front" computation of the matrix from coordinate $(0, 0)$ to (m, n). In hardware, the systolic array is fixed with the contents of sequence S, and sequence T is streamed through the array.

The processing element (PE) implemented in this work is based on [52], and is shown in Figure 3.13. Each PE is configured with one base from sequence S. Every time step, each PE consumes a base of sequence T from its predecessor and computes the value of one cell of the scoring matrix. Because sequence T may be very large, it is infeasible to store the entire matrix contents in memory; rather, each PE compares its computed score with a "max" score (higher values indicates better alignments) provided by the previous PE. Logic at the end of the array maintains one or more best alignment locations.

Both *bFlow* and *ConVI* were used to implement such an accelerator for the HC-1 platform. While both have significant differences, the top-level architecture is identical, and is displayed in Figure 3.14. The accelerator is initialized by loading a query sequence S—the length of S is limited by the number of PEs instanced in the accelerator—into the systolic array. Then, a large reference T is streamed through the array, and the best alignment locations are computed. Both sequences are loaded into coprocessor memory prior to execution, and AEG registers are used to provide the AEs with the length and location of each sequence, as well as send the alignment locations back to the host.

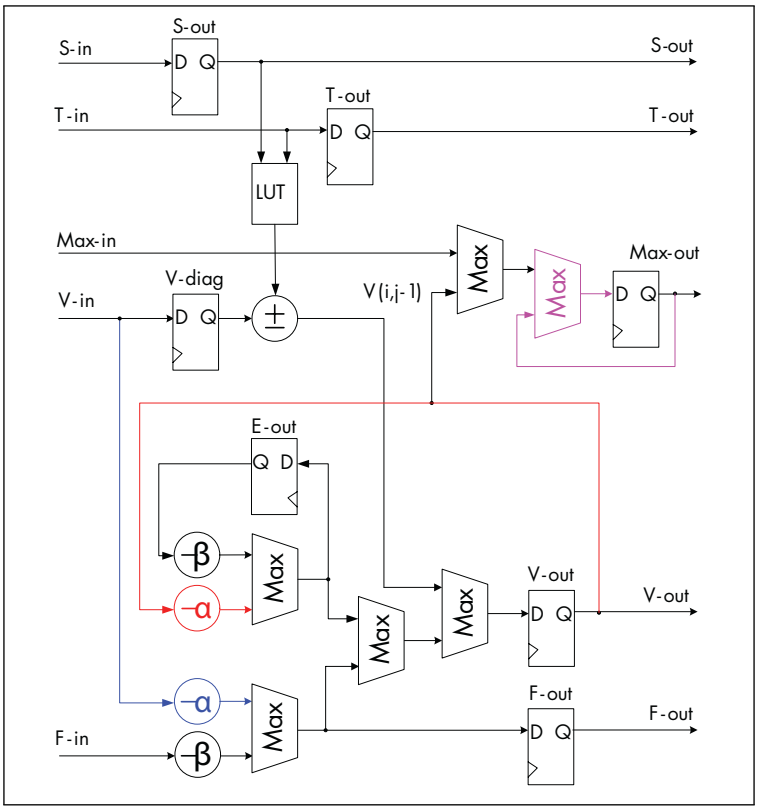


Figure 3.13: Diagram of the Smith-Waterman systolic array processing element used in this work. Source: [52].

A comprehensive set of screenshots of the *bFlow* and *ConVI* SW accelerator implementations can be found in Appendices C and D, respectively. Screenshots of the SW PE implementations in can be found in Figures 3.16 and 3.17, respectively. Each of the front-ends brought unique challenges to the development of the SW accelerator, especially the PE, in part due to the prioritization of high-throughput in the result. Some of those challenges are discussed in the following text.

Synchronous Design

While Azido provides flow-control abstractions to aid in synchronous design, they rely heavily on FIFOs and, because of the existence of feedback in the SW PE, the design could not

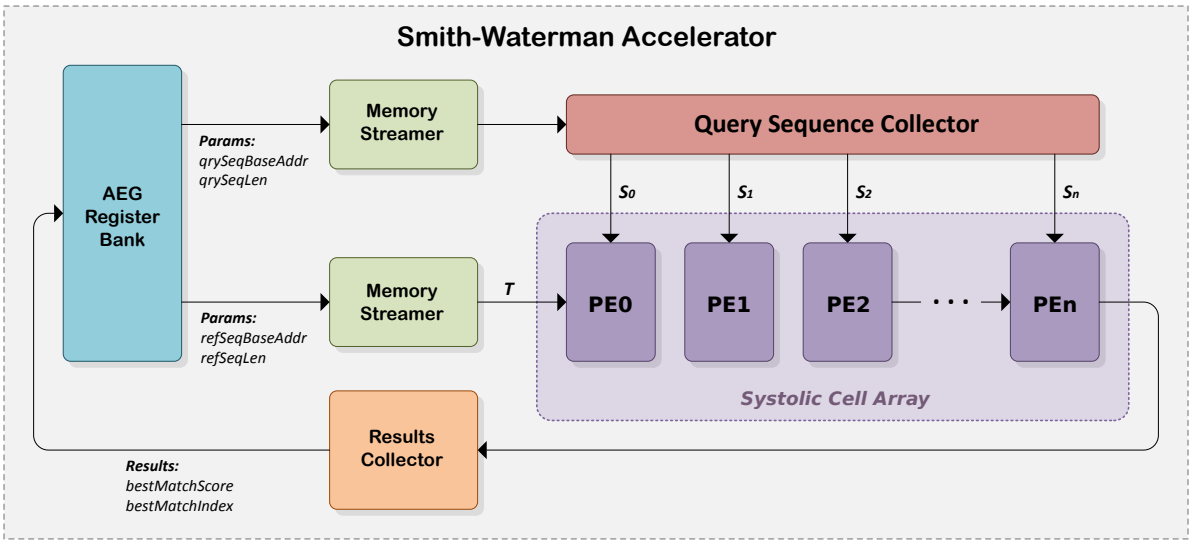


Figure 3.14: Top-level architecture of the HC-1 accelerator for the Smith-Waterman matrix fill operation.

synthesize to a netlist capable of meeting the desired single-cycle latency. The alternative method was to implement the PE using clocked registers for synchronization, as shown in Figure 3.16. This approach proved straightforward for this thesis' author, who has digital design experience; however, such limited abstraction would not be appropriate for a non-engineer attempting to do the same.

LabVIEW FPGA's approach to synchronous design is much higher level, including abstractions such as for and while loops, which indicate periodic execution of all functions within the structure. Using these constructs, the SW systolic array could be implemented with great abstraction; however, as with Azido, limitations in the throughput of the resulting netlist resulted in the use of more explicit constructs. Specifically, the implementation was created using a top-level Single-Cycle Timed Loop (SCTL), within which was nested all the design's functionality, as well as synchronization constructs called "feedback nodes." By this approach, the feedback nodes operated as simple registers, updating once every clock cycle (see Figure 3.17).

Replication of the Processing Element

The approach taken for replicating and connecting the PE to form a systolic array is different across the two flows. The ability define objects recursively in Azido was utilized for the *bFlow* implementation in the following manner. Two definitions were provided for the PE, *Cell* (Figure 3.15) and *Cell (leaf)* (Figure 3.16). The latter is the definition of a single PE, while the former is defined recursively, such that each instantiation prunes one base from the input S and passes it to *Cell (leaf)*, passing the rest of S to a self-instance. Thus, after elaboration, passing a query sequence S with 16 bases to a single instance of *Cell* results in 16 instances of *Cell (leaf)*.

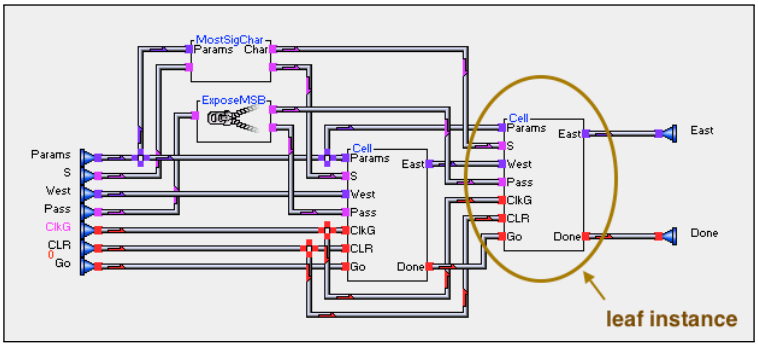


Figure 3.15: Recursive definition of the Smith-Waterman processing element in Azido. One base is pruned from input S by the *MostSigChar* object and passed to the leaf instance on the right, while the remainder of S is fed to a self-instance.

PE replication in *ConVI* required a different approach. While the LabVIEW platform permits calling a VI within itself in a recursive manner, this feature does not extend to the the LabVIEW FPGA Module. Hence, a LabVIEW script, shown in Figure 3.18, was created[1] to automate systolic array generation through VI replication and interconnection. Using this script, systolic pipelines with an arbitrary number of PEs can constructed automatically.

[1]The systolic array generation script was provided by lab mate Kevin Lee.

3.4 VBI Workshops

To verify the usability and productivity that *bFlow* and *ConVI* aim to provide, they were used in an NSF-funded summer program at the Virginia Bioinformatics Institute at Virginia Tech [44]. This program was hosted during the summers of 2012, 2013, and will be hosted again in 2014. The *bFlow* framework was tested at the 2012 institute, while *ConVI* was used in 2013.

Each occurrence of this two-week institute brought in 16 students from a diverse set of fields and levels of expertise. For the first week, all students attended workshops covering a wide range of topics related to HPC in the field of bioinformatics, and completed short assignments related to the topic. The students then split into four- or five-person groups, each spending the second week attempting to address a specific problem in the field. In each workshop, one of the groups was instructed to provide acceleration of the Smith-Waterman algorithm on the Convey HC-1 platform using one of the graphical tools developed in this thesis (*bFlow* in 2012 and *ConVI* in 2013). Prior to the event, the students in this group were asked to review [52] to gain a basic understanding of the systolic array approach to implementing the accelerator. At the end of the first week, this group was given the skeleton SW implementation described in Section 3.3.2, and tasked with extending its functionality and improving its performance. The results from these workshops are given in Section 4.1.

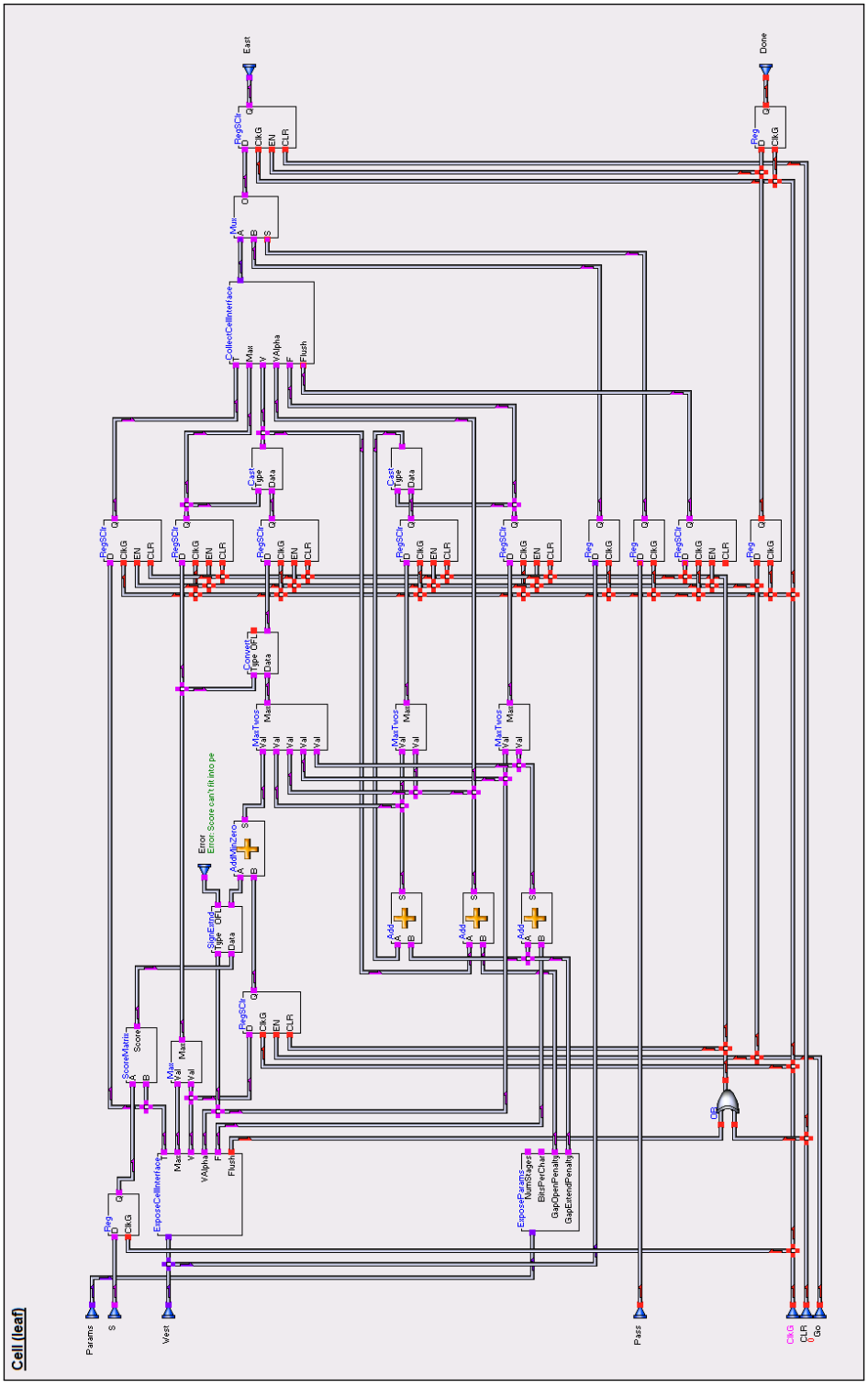


Figure 3.16: Implementation of the Smith-Waterman processing element in Azido.

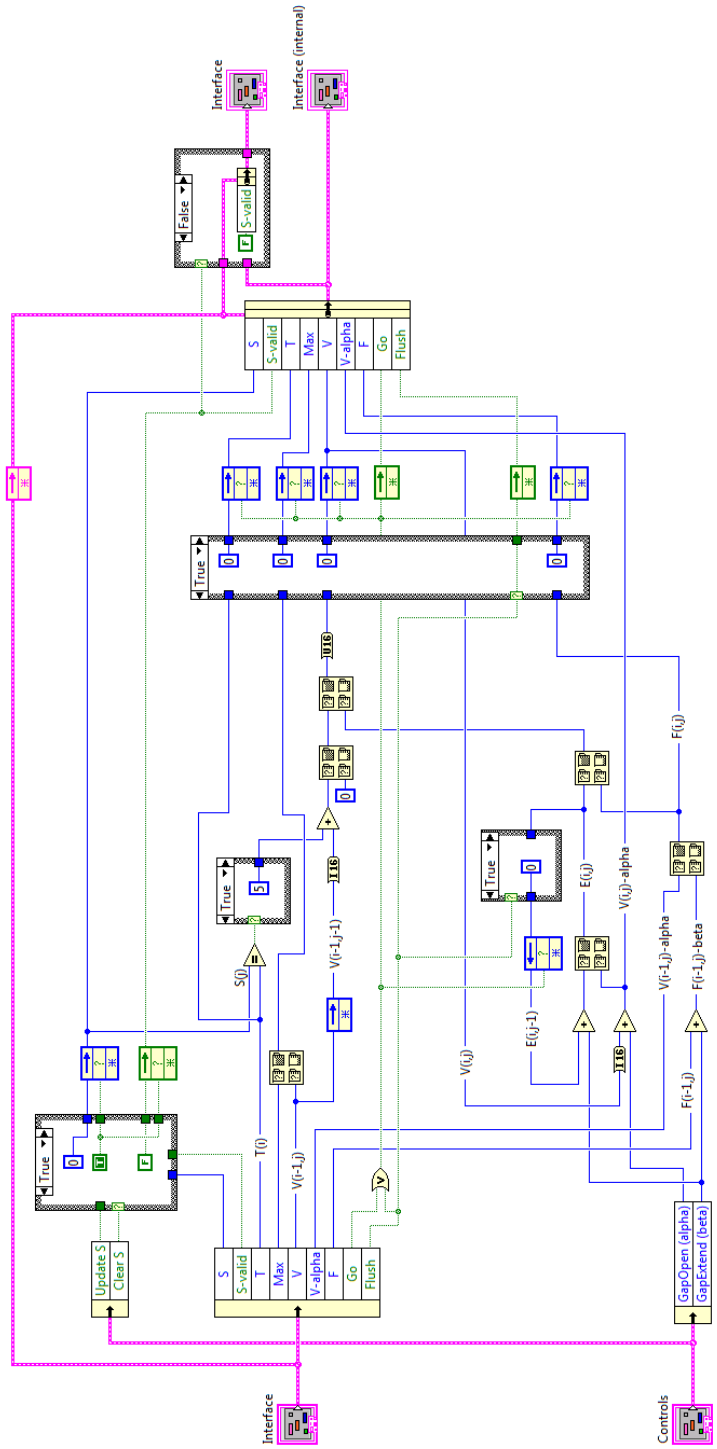


Figure 3.17: Implementation of the Smith-Waterman processing element in LabVIEW.

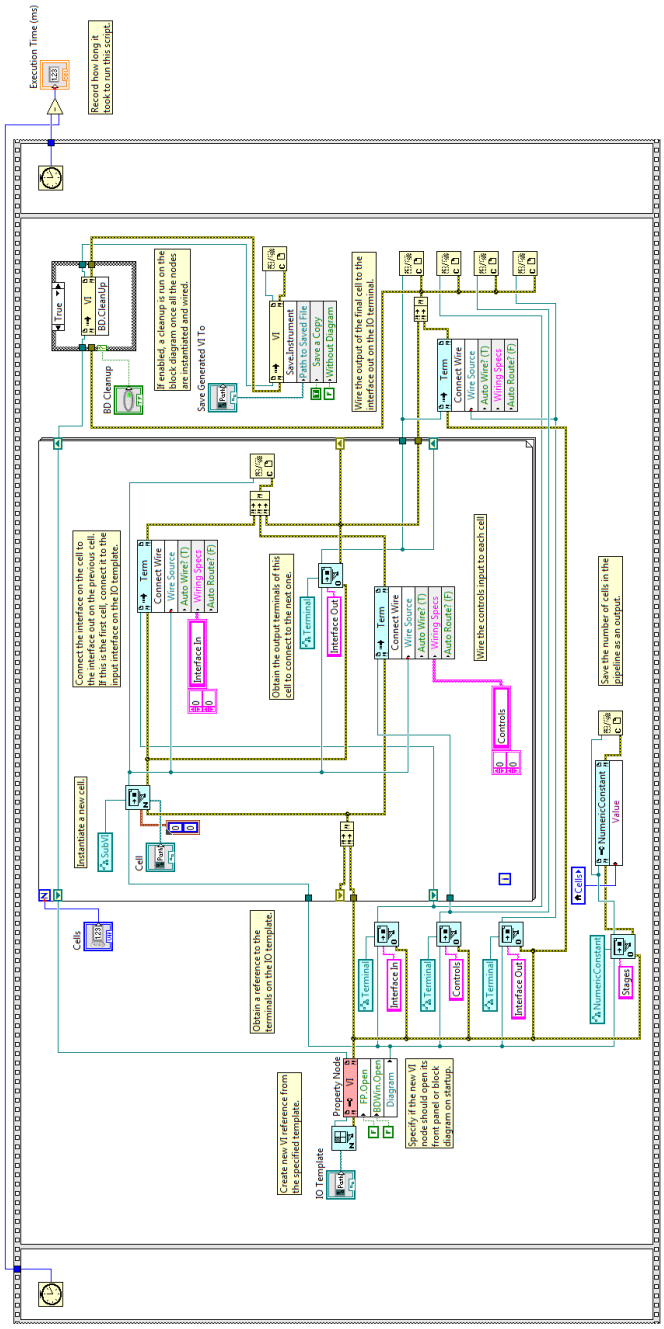


Figure 3.18: LabVIEW script that automates VI replication and interconnection to generate systolic arrays. Inputs to this script include the PE VI, a specification of the systolic interface, and the number of desired PEs in the generated array.

Chapter 4

Results & Analysis

The two flows constructed in this work, *bFlow* and *ConVI*, were given to participants of the NSF-sponsored workshop at the Virginia Bioinformatics Institute at Virginia Tech [44]. In 2012 and 2013, two groups of 4-5 students each were tasked with using the flows to improve the performance and function of a skeleton Smith-Waterman accelerator. This chapter contains the outcomes of these workshops, a qualitative discussion of the usability challenges countered by those using the flows, and the results of reducing accelerator compilation time in *bFlow* using incremental implementation techniques.

4.1 Workshop Results

Over two years, two groups of 4-5 students were tasked with exploring the acceleration of DNA nucleotide sequence alignment using *bFlow* in 2012 and *ConVI* in 2013, and starting with the bare-bones Smith-Waterman accelerator discussed in Section 3.3.2. The results from these workshops are presented in the following text.

4.1.1 2012 Workshop (bFlow)

Given the task of improving performance and functionality, the first group, which was comprised of undergraduate and graduate-level students in the sciences, planned to make two changes to the provided reference implementation. The first modification was functional, and involved the addition of logic (see Figure 4.1) to the end of pipeline with the function of maintaining the index of the single highest scoring alignment. This logic consisted of Azido's counter, maximum, multiplexer, and register objects, and was implemented in less than one day.

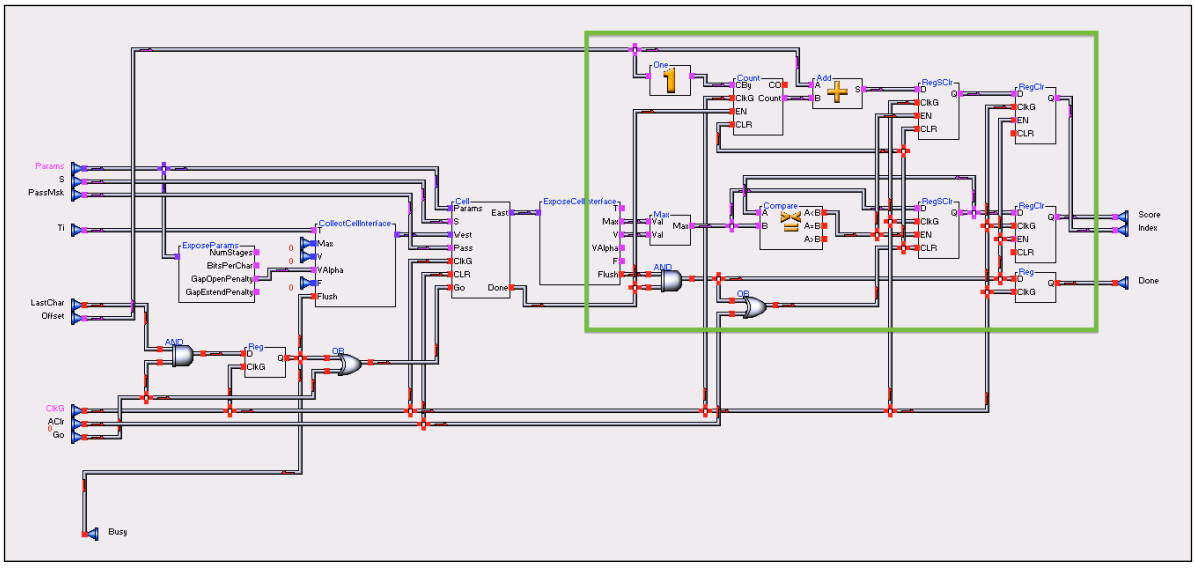


Figure 4.1: Smith-Waterman pipeline logic in Azido. The logic that maintains the index of the best alignment is surrounded by the green box.

The second modification stems from the poor AE utilization when only one query sequence is loaded into the PE pipeline (see Figure 3.14). By replicating the pipeline such that n_p pipelines are in the AE, this utilization can be improved by streaming the reference through n_p query sequences in parallel. If the *same* query sequence is loaded into all pipelines, the reference sequence can be split into n_p partitions and streamed through the AE with an almost-linear throughput speedup proportional to n_p (increasingly linear as length of

$T \to \infty$, due to partition overlap requirements). The query sequence is aligned to each partition in parallel, and the results merged at the end of the pipelines. The implementation of this parallelization involved changes to the high-level accelerator architecture—specifically, replication of the systolic cell array and addition of logic at the front and back of the arrays to split the incoming stream and collect results from each array.

By splitting the reference into multiple chunks (up to 32) and streaming each chunk into separate, parallel pipelines, the accelerator can consume the reference up to 32 times faster than the single-pipeline approach. Using this technique, the group realized a 4× bandwidth increase from 150 million to 600 million bases per second (bps), and a theoretical speedup of 32× to 4.8 billion bps, given enough parallel cell arrays. This is not entirely without cost, however, as aligning to partitions of the reference sequence does require preprocessing, and as the number of parallel pipelines increase, the maximum length of the query sequence decreases due to FPGA resource limitations. Note that, due to *bFlow*'s lack of an intuitive abstraction for multi-AE development, the accelerator was run on only one of the four AE FPGAs in the HC-1.

4.1.2 2013 Workshop (ConVI)

The second group contained both undergraduate and graduate biology students, as well as two computer-savvy participants—one a computer programmer and the other an engineering undergraduate. Rather than focus on improving alignment throughput, this group focused solely on developing extensions to functionality Specifically, the participants created logic to track multiple alignment locations in a approach similar to that used by the previous year's group, but with support for saving more than one alignment location.

Several approaches were discussed, and several of the group members independently developed LabVIEW VIs to perform this function. The simple, naive technique of saving the indices of the k best alignments was discussed but discarded due to the problem long, high-scoring alignments contributing many indices to the results table (i.e. a subset of a

high-scoring alignment may still be considered a "best" alignment). The final approach attempted to find *local maxima* alignments, by monitoring the scores produced by the last PE. The logic considers adding an index to the table only when the produced alignment score is less than its predecessor, which must be greater than *its* predecessor, indicating a local maxima.

4.2 Usability Challenges

The participants' use of *bFlow* and *ConVI* provided valuable insight into how non-engineers might use the flows. This section contains an informal discussion of the successes and failures of the two tools as used by the workshop participants.

4.2.1 bFlow

The students in the first workshop found synchronous design in the Azido I2ADL environment to be quite challenging. Azido encourages the use of synchronous objects with built-in Go-Done-Busy-Wait (GDBW) interface flow control, and synchronous, GDBW-based definitions are provided for most CoreLib objects. However, this flow control is implemented using FIFOs not optimized away during synthesis, resulting in significant, and often unnecessary resource overhead. Because of this, the students were encouraged to stay away from GDBW-based objects, requiring explicit specification of synchronous behavior using registers, etc. Furthermore, the abstraction provided by Azido to implement state machine control is poor, complicating the design process.

The heavy use of asynchronous objects increased challenge of meeting timing closure. Under the standard parameterization, the Convey PDK enforces a clock rate of 150 MHz, which is easily broken by moderately long chains of asynchronous operations. However, Azido neither analyses the design nor enforces any timing restrictions at compile time; hence,

whether or not a design meets timing is determined only during the "hidden," behind-the-scenes implementation processes, resulting in the loss of the abstraction that Azido provides when a constraint is not met and the design proves dysfunctional.

In *bFlow* the Azido front-end connects to the accelerator only when running on the co-processor, and not during system simulation. Thus, while the generated accelerator can be verified in Convey's simulation framework, it must be initiated in a command-line environment, and the results examined using a waveform viewer. This limited the independence of the users, as system-wide verification required the involvement of an instructor.

4.2.2 ConVI

In general, the *ConVI* group's experience was much more positive than the previous year's group for several reasons. Fundamentally, the usability of the LabVIEW front-end exceeded that of the Azido environment. This can be attributed to the stability and maturity of the LabVIEW platform, and the intuitiveness of its provided control abstractions, such as state and loop control containers. The behavior of these constructs was easily understood by the students, and they were quickly able to independently design functional VIs.

Also, the seamless integration of the LabVIEW front-end into Convey's system simulation framework allowed for relatively fast verification without significant technical expertise, as required in *bFlow*. Ideally, this would have allowed independent verification of the group's design; however, in practice, bugs in the integration processes limited this independence.

One of the more substantial drawbacks discovered was that the students designed primarily in LabVIEW under the *My Computer* or *Desktop* target, rather than the *FPGA* target, since only VIs under the former can be verified locally (i.e. without building the VI for verification through external tools). Because desktop-bound VIs have access to control and data constructs incompatible with FPGA targets, the students initially designed using FPGA-incompatible objects, and often had to be reminded to use only constructs and

sub-VIs available to VIs under the FPGA target.

4.3 Compilation Performance

This section covers the results of reducing the compilation times in the *bFlow* framework using the Xilinx Partitions flow [49] and the *qFlow* framework [16]. Due to time constraints, these techniques were not applied to the bitstream compilation phase in *ConVI*.

4.3.1 bFlow

In the *bFlow* compilation phase, the synthesized netlist generated from the Azido accelerator design is instantiated within a hierarchy of Verilog modules (see Figure 3.3), which is compiled to a bitstream for configuration of the HC-1 AE FPGAs. In addition to the standard, Convey-provided make process, two incremental implementation frameworks were considered (discussion in Section 3.1.3). The performance of each implementation framework was measured for the compilation of the bare-bones Smith-Waterman implementation (Section 3.3.2) with the top-level pipeline modifications from the first VBI workshop (see Section 4.1.1). The compilations were executed on the VBI ShadowFax compute cluster [45], and their run-time durations recorded in Table 4.1 and visualized in Figure 4.2. Each SW configuration is named with convention sw_MxN, where M is the number of parallel systolic arrays and N is the length of each array. The median speedup over the standard, Convey-provided flow for the partitions-based approach 1.51, while that of *qFlow* was 2.76. The last two configurations tested, 4×48 and 4×64, could not be placed into the dynamic region, and those compilations exited with errors. Note that the same dynamic region area constraints were used for both flows. Also note the jump in build time from configuration 4×16 to 4×32, which is due to the the significantly increased utilization of the dynamic region. In fact, due to the resource overhead introduced by the flows, the area devoted to user logic was reduced, and the largest two configurations could not be placed.

Table 4.1: Build times (mean of three runs) for Convey's standard flow and the Partitions- and *qFlow*-based flows for the Smith-Waterman accelerator. The speedup over the standard flow is given in parentheses. Note: Device utilization listed is the utilization due to only the user logic.

Design	**Cell Ct**	**Device Util. (%)**		**Mean Build Time (min)**		
		LUTs	FFs	Standard	Partitions	qFlow
sw_1x8	8	1.83	2.16	89.10	65.60 (1.36)	26.58 (3.35)
sw_1x12	12	2.43	2.44	89.90	54.16 (1.66)	26.12 (3.44)
sw_1x16	16	3.02	2.73	104.09	57.92 (1.80)	32.94 (3.16)
sw_1x24	24	4.22	3.30	98.80	76.85 (1.29)	34.75 (2.84)
sw_1x32	32	5.41	3.87	102.20	72.89 (1.40)	38.27 (2.67)
sw_4x8	32	5.62	4.10	96.90	61.58 (1.57)	39.20 (2.47)
sw_1x48	48	7.79	5.01	128.50	82.95 (1.55)	43.02 (2.99)
sw_1x64	64	10.18	6.15	129.73	87.92 (1.48)	53.75 (2.41)
sw_4x16	64	10.36	6.34	130.08	84.74 (1.54)	53.94 (2.41)
sw_4x32	128	19.85	10.81	165.99	173.62 (0.96)	97.33 (1.71)
sw_4x48	192	29.34	15.29	173.22		
sw_4x64	256	38.83	19.76	208.89		

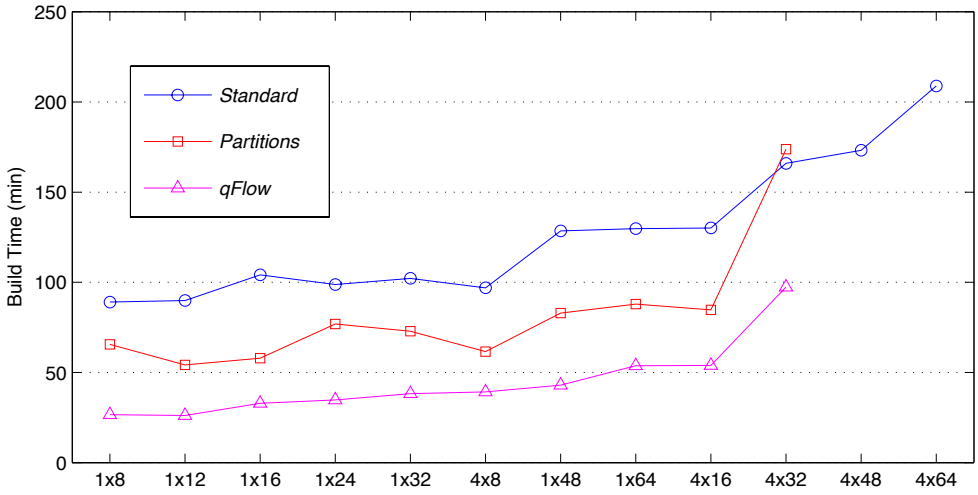


Figure 4.2: Build times (mean of three runs) for Convey's standard flow and the Partitions- and *qFlow*-based flows for the Smith-Waterman accelerator.

4.4 Resource Utilization

Also important is the efficiency of the netlists produced by the two flows—here this is defined in terms of resource utilization per functional unit. For *bFlow* and *ConVI*, this is almost exclusively dependent on the netlist generation processes used by Azido and LabVIEW FPGA, respectively. Table 4.2 compares the resource utilization of a Smith-Waterman processing element with a 16-bit data path (i.e. support for maximum sequence length of 10,000), as described in Azido, LabVIEW, and hand-written Verilog, and mapped by Xilinx ISE 13.1 [47] for the Virtex-5 XC5VLX330 part. This test was performed by building using each front-end to implement a 32-cell systolic array, while omitting of all memory access control and arithmetic hardware. The cell designs for *bFlow* and *ConVI* are in Appendices C and D, respectively. The per-cell utilization is $1/32$ of that due to the 32-cell array. Note that while Azido generated netlist and the Verilog design contain no communication overhead, LabVIEW includes register access hardware in every compilation; thus, a baseline utilization was computed and subtracted from the pipeline utilization.

After mapping, the resource overhead compared to the hand-written Verilog design is non-

Table 4.2: Resource utilization of a single Smith-Waterman processing element described using handwritten Verilog, LabVIEW FPGA, and Verilog. The resource overhead due to using the abstracted design environments is given in parentheses.

Front-end	Slice LUTs	Flip-Flops
Verilog (HDL)	300	144
LabVIEW FPGA (*ConVI*)	383 (1.27)	159 (1.10)
Azido (*bFlow*)	492 (1.64)	242 (1.68)

trivial. The LabVIEW-generated netlist consumes 27% more lookup tables (LUTs) and 10% more flip-flops, while the Azido netlist consumes 64% more LUTs and 68% more flip-flops. Comparing Azido and LabVIEW, Azido's higher LUT usage is probably due to its generation of an EDIF netlist that is processed by the Xilinx ISE implementation after synthesis, a step responsible for significant optimization. Similarly, the lower flip-flop usage of the LabVIEW generated HDL may be due to optimization during synthesis.

Chapter 5

Conclusions

In summary, big-data in the sciences is a growing concern, especially in the field of bioinformatics, as the explosion of available genomic data is quickly outpacing the growth in usable HPC technology. FPGA-based heterogeneous systems, such as the Convey HC-1 platform, have the potential to address such growth through the creation of custom, high-performance accelerators; however, the use of existing development flows for such platforms (i.e. Convey Personality Development Kit) necessitates a heavy background in digital design concepts, forcing bioinformatics domain experts to influence accelerator development through a hardware engineer. This thesis presents two end-to-end development flows, *bFlow* and *ConVI*, which are based on graphical, design front-end tools intended for hardware design by non-engineers. These flows enable domain experts without significant digital design experience to design, test, and deploy accelerators on the Convey HC-1 platform with minor to no intervention by a hardware engineer. The specific contributions of this work are:

- The assembly of two end-to-end accelerator development flows targeting the Convey HC-1 coprocessor, *bFlow* and *ConVI*, which are based on the Azido and LabVIEW graphical programming environments, respectively. This assembly included the integration of the run-time feedback mechanisms of both tools into the Convey HC run-time

and system simulation (*ConVI* only) frameworks. Also, for *bFlow*, the bitstream compilation process was accelerated using two incremental compilation techniques, in the interest of improving the productivity of the flow.

- A bare-bones implementation of a HC-1-based accelerator for the matrix-fill step of the Smith-Waterman sequence alignment algorithm, developed in both *bFlow* and *ConVI*.
- An informal evaluation of the usability of each flow across two years of an annual, two-week summer program on HPC in bioinformatics, including a short quantitative evaluation of the performance of the two incremental compilation techniques integrated into *bFlow*.

A summary of the results and analysis presented in Chapter 4 and a list of potential future work items are given in the following two sections.

5.1 Summary of Results

In the summers of 2012 and 2013, the *bFlow* and *ConVI* flows, respectively, were given to participants of a two-week program at the Virginia Bioinformatics Institute at Virginia Tech. The participants were tasked with extending a bare-bones Smith-Waterman (SW) implementation with regard to functionality and performance. In the first workshop, the group made two modifications. The first was functional, involving the addition of logic at the end of the SW systolic array to track the index of the best alignment. The second effort, which was the primary focus of the group, involved replication of the systolic array in order to improve accelerator throughput. This resulted in a theoretical AE throughput increase of 32×, and a 4× increased realized in hardware. In the second year, the group members focused exclusively on functional improvements, adding hardware to the end of the pipeline to track *multiple* best alignments. Several implementations of this logic were described in LabVIEW without instructor intervention; however, most of the objects used were supported

by LabVIEW but not the FPGA module, and only one of the designs was suitable for the HC-1-based accelerator.

Both workshops provided valuable insight into the usability of the development flows, and the front-end tools they utilize. In the first workshop, the students gained comprehension of the algorithm as described in Azido, and were able to make simple data path-centric changes independently; however, poor abstractions for synchronous design and the inability to perform system simulation from the Azido front-end resulted in the participants depending heavily on the presence of an instructor during development. In the second workshop, which utilized the *ConVI* flow, the students were far more independent, due to the usability of the abstracted syntax provided by the LabVIEW front-end, as well as the ability to perform system simulation from the LabVIEW front panel. They group was able to describe and test functional blocks independently; however, as many of the LabVIEW abstractions are unsupported by the FPGA Module, the students did require help when preparing their VIs for execution on the FPGA target.

The techniques used to reduce bitstream implementation runtime in *bFlow* were successful, achieving a mean speedup over Convey's standard framework of 1.51 and 2.76 for the Xilinx Partitions and *qFlow* tools, respectively, for the compilation of several configurations of the multi-pipeline SW implementation. The primary downside of using such tools proved to be the area overhead introduced by the modular, incremental approach that they take; hence, some especially large accelerators could not be fit into the dynamic sandbox region, and failed placement.

5.2 Future Work

A list of future work items that would effectively reinforce or extend this thesis is provided here:

- A premise of this work is that traditional, HDL-based flows are practically unusable by non-engineers. However, a controlled study evaluating the usability and performance of *bFlow* and *ConVI* in comparison to such traditional flows would be beneficial. For example, provide one group with HDL educational materials and a traditional flow, give the other group *bFlow* or *ConVI*, and assign the same objective to both groups, measuring the time-to-solution and quality of implementation of both groups.

- While Smith-Waterman is still in widespread use, it is often relegated to aligning small sequences as part of a larger algorithm, and its level of complexity may not represent that of most bioinformatics algorithms [22, 33]. For this reason, more complicated, more "real-world" algorithms should be considered.

- Exploring memory-access abstractions other than the streaming model, which is heavily relied upon in this thesis, is prerequisite to the application of these flows to more complex algorithms employing non-streaming memory access, unlike the Smith-Waterman accelerator.

- This work focuses on enabling non-engineers to design HC-1 coprocessor personalities, leaving the host software to be developed separately in the host Linux environment. Unifying the development process by targeting the HC-1 x86 host in addition in to the coprocessor AEs within the LabVIEW environment would prove beneficial to the usability of *ConVI*.

Bibliography

[1] A. Allan, D. Edenfeld, W.H. Joyner, A.B. Kahng, M. Rodgers, and Y. Zorian. 2001 technology roadmap for semiconductors. *Computer*, 35(1):42–53, January 2002.

[2] Altera Corporation. Altera SDK for OpenCL. http://www.altera.com/products/software/opencl/opencl-index.html. [Online; accessed 1 Dec 2013].

[3] Jason D. Bakos. High-Performance Heterogeneous Computing with the Convey HC-1. *Computing in Science & Engineering*, 12(6):80–87, November 2010.

[4] Ian Bird. Computing for the Large Hadron Collider. *Annual Review of Nuclear and Particle Science*, 61(1):99–118, October 2011.

[5] Bluespec, Inc. Bluespec compiler. http://www.bluespec.com/high-level-synthesis-tools.html. [Online; accessed 1 Dec 2013].

[6] Andy Caley and Kent Gilson. Isolation of behavior design from system implementation. In *2012 International Conference on Reconfigurable Computing and FPGAs*, pages 1–6. IEEE, December 2012.

[7] Calypto Design Systems, Inc. Catapult: Product Family Overview. http://calypto.com/en/products/catapult/overview/. [Online; accessed 1 Dec 2013].

[8] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp. In *Proceedings of*

the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11, page 33, New York, New York, USA, February 2011. ACM Press.

[9] Ramakrishna Bijanapalli Chakri. *Enabling the Use of Heterogeneous Computing for Bioinformatics*. PhD thesis, Virginia Polytechnic Institute and State University, 2013.

[10] Convey Computer Corporation. Convey: Better Computing for Better Analytics. http://www.conveycomputer.com/products/hcseries/. [Online; accessed 19 Nov 2013].

[11] Convey Computer Corporation. Financial Analytics Personality. http://www.conveycomputer.com/files/8213/5085/5812/FinancialAnalyticsPersonalityDatasheet.pdf. [Online; accessed 2 Dec 2013].

[12] Convey Computer Corporation. GraphConstructor Personality. http://www.conveycomputer.com/files/1513/5085/5638/ConveyGraphConstructor_datasheet_V_11_019.1CGCe.pdf. [Online; accessed 2 Dec 2013].

[13] Convey Computer Corporation. HC-1 Data Sheet. http://bgcomm.com/Resources/HC-1DataSheet.pdf.

[14] V. Curcin and M. Ghanem. Scientific workflow systems - can one size fit all? In *2008 Cairo International Biomedical Engineering Conference*, pages 1–9. IEEE, December 2008.

[15] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. From opencl to high-performance hardware on FPGAS. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534. IEEE, August 2012.

[16] Tannous Frangieh and Peter Athanas. A design assembly framework for FPGA backend acceleration. In *2012 International Conference on Reconfigurable Computing and FPGAs*, pages 1–6. IEEE, December 2012.

[17] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[18] Impulse Accelerated Technologies, Inc. Impulse C. http://www.impulseaccelerated.com/products_universal.htm. [Online; accessed 18 Nov 2013].

[19] A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, and C.-L. Wang. Heterogeneous computing: challenges and opportunities. *Computer*, 26(6):18–27, June 1993.

[20] Christiane Lefevre. LHC Guide, English version. A collection of facts and figures about the Large Hadron Collider (LHC) in the form of questions and answers. [Online; accessed 18 Nov 2013], January 2008.

[21] Fran Lewitter and Michael Rebhan. Establishing a successful bioinformatics core facility team. *PLoS computational biology*, 5(6):e1000368, June 2009.

[22] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics (Oxford, England)*, 25(14):1754–60, July 2009.

[23] Kenli Li, Xiaoyong Tang, and Keqin Li. Energy-Efficient Stochastic Task Scheduling on Heterogeneous Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2013.

[24] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. Comparison of next-generation sequencing systems. *Journal of Biomedicine and Biotechnology*, 2012, 2012.

[25] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers*, 26(4):18–25, July 2009.

[26] L J McIver, J W Fondon III, M A Skinner, and H R Garner. Evaluation of microsatellite variation in the 1000 Genomes Project pilot studies is indicative of the quality and utility of the raw data and alignments. *Genomics*, 97(4):193–199, 2011.

[27] National Instruments Corporation. How Can I Use NI LabVIEW? - Application Areas. http://www.ni.com/labview/applications/. [Online; accessed 6 Dec 2013].

[28] National Instruments Corporation. Introduction to G Programming. http://www.ni.com/white-paper/7668/en/. [Online; accessed 7 Dec 2013].

[29] National Instruments Corporation. NI LabVIEW. http://www.ni.com/labview/. [Online; accessed 1 Dec 2013].

[30] National Instruments Corporation. NI LabVIEW FPGA Module. http://www.ni.com/labview/fpga/. [Online; accessed 1 Dec 2013].

[31] National Instruments Corporation. Unsupported LabVIEW Features (FPGA Module). http://zone.ni.com/reference/en-XX/help/371599J-01/lvfpgaconcepts/fpgamisc/. [Online; accessed 1 Dec 2013].

[32] Brent E Nelson, Michael J Wirthlin, Brad L Hutchings, Peter M Athanas, and Shawn Bohner. Design Productivity for Configurable Computing. In *ERSA*, volume 8, pages 57–66, 2008.

[33] Corey B. Olson, Maria Kim, Cooper Clauson, Boris Kogon, Carl Ebeling, Scott Hauck, and Walter L. Ruzzo. Hardware Acceleration of Short Read Mapping. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 161–168. IEEE, April 2012.

[34] Karl Pereira, Peter Athanas, Heshan Lin, and Wu Feng. Spectral Method Characterization on FPGA and GPU Accelerators. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 487–492. IEEE, November 2011.

[35] Karl Savio Pimenta Pereira. *Characterization of FPGA-based high performance computers*. PhD thesis, Virginia Polytechnic Institute and State University, 2011.

[36] Michael A Quail, Miriam Smith, Paul Coupland, Thomas D Otto, Simon R Harris, Thomas R Connor, Anna Bertoni, Harold P Swerdlow, and Yong Gu. A tale of three

next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC genomics*, 13(1):341, January 2012.

[37] Michael S Rosenberg. *Sequence alignment: methods, models, concepts, and strategies*. University of California Pr, 2009.

[38] Caitlin Sadowski, Thomas Ball, Judith Bishop, Sebastian Burckhardt, Ganesh Gopalakrishnan, Joseph Mayo, Madanlal Musuvathi, Shaz Qadeer, and Stephen Toub. Practical parallel and concurrent programming. In *Proceedings of the 42nd ACM technical symposium on Computer science education - SIGCSE '11*, page 189, New York, New York, USA, March 2011. ACM Press.

[39] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[40] Synopsys, Inc. Synphony C Compiler Tool for High-Level C Synthesis. http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx. [Online; accessed 18 Nov 2013].

[41] The MathWorks, Inc. HDL Coder - MATLAB & Simulink. http://www.mathworks.com/products/hdl-coder/. [Online; accessed 6 Dec 2013].

[42] The MathWorks, Inc. MATLAB - The Language of Technical Computing. http://www.mathworks.com/products/matlab/. [Online; accessed 6 Dec 2013].

[43] The MathWorks, Inc. Simulink: Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/. [Online; accessed 2 Dec 2013].

[44] Virginia Bioinformatics Institute. High Performance Computing in the Life/Medical Sciences: Summer Institute. http://nsfsi.vbi.vt.edu/.

[45] Virginia Bioinformatics Institute. Partnership Supercomputing Program. http://www.vbi.vt.edu/high_performance_computing/. [Online; accessed 6 Dec 2013].

[46] Worldwide LHC Computing Grid. Worldwide LHC Computing Grid | WLCG. http://wlcg.web.cern.ch/. [Online; accessed 18 Nov 2013].

[47] Xilinx, Inc. ISE Design Suite. http://www.xilinx.com/products/design-tools/ise-design-suite/. [Online; accessed 7 Dec 2013].

[48] Xilinx, Inc. Xilinx CORE Generator System. http://www.xilinx.com/tools/coregen.htm. [Online; accessed 6 Dec 2013].

[49] Xilinx, Inc. Hierarchical Design Methodology Guide, UG748 (v13.1). http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/Hierarchical_Design_Methodology_Guide.pdf, 2011.

[50] Xilinx, Inc. System Generator for DSP, UG640 (v14.3). http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/sysgen_user.pdf, 2012.

[51] Xilinx, Inc. Introduction to FPGA Design with Vivado High-Level Synthesis, UG998 (v1.0). http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf, 2013.

[52] Peiheng Zhang, Guangming Tan, and Guang R. Gao. Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. In *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications held in conjunction with SC07 - HPRCTA '07*, page 39, New York, New York, USA, November 2007. ACM Press.

Appendix A

CPLib Software Library

Listing A.1: cplib.h

```
1   #ifndef _CPLIB_H
2   #define _CPLIB_H
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <boost/thread.hpp>
7
8   #ifdef CONVEY
9   #include <convey/usr/cny_comp.h>
10  #endif
11
12  #include "mpipserver.h"
13
14  // Typedefs
15  typedef unsigned long long int uint64;
16
17  class CPLib {
18  public:
19      // Constructor
20      CPLib(int debugLvl = 0);
21
22      // Destructor--ensures that the mpip server is stopped
23      ~CPLib();
24
25      // Returns true if the signature was acquired successfully
26      bool hasSig();
27
28      // Stats accessors
29      int getCPCallCt();
30      int getAEGWrCt();
31      int getAEGRdCt();
32
33      // Start and stop the mpip server
34      void runMPIPServer();
35      void stopMPIPServer();
36
```

```
37      // Allocate memory on the coprocessor
38      void * mallocCP(size_t size);
39      void * mallocCP(const void *buf, size_t size);
40
41      // Write AEG registers
42      void writeAEGReg(uint64 data, int aegIdx);
43      void writeAEGReg(uint64 data, int aegIdx, int aeIdx);
44
45      // Read AEG registers
46      uint64 readAEGReg(int aegIdx);
47      uint64 readAEGReg(int aegIdx, int aeIdx);
48
49      // Execute custom coprocessor instructions
50      void execCPInstr(int instIdx);
51      void execCPInstr(int instIdx, int aeIdx);
52
53      // Simplified register access and custom instruction methods for LabVIEW designs
54      void writeReg(uint64 data, int index);
55      uint64 readReg(int index);
56      void execCmd(int index);
57
58  protected:
59      // Personality signatures
60      #ifdef CONVEY
61      cny_image_t mSig, mSig2;
62      #endif
63
64      // Flags
65      bool mHasSig;
66      int mSigStat;
67
68      // Stats
69      int mCPCallCt;
70      int mAEGWrCt;
71      int mAEGRdCt;
72
73      // Config
74      int mDebugLvl;
75
76      // Mutex for dispatch interface ops
77      boost::mutex mDIMutex;
78
79      // MPIP Server
80      MPIPServer mMPIPServer;
81  };
82
83  #endif // _CPLIB_H
```

Listing A.2: cplib.cpp

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <boost/thread.hpp>
4
5   #ifdef CONVEY
6   #include <convey/usr/cny_comp.h>
7   //#include "cplib_cny.s"
8   #endif
9
10  #include "cplib.h"
11  #include "timer.h"
12
13  //
14  // Coprocessor assembly routines
15  //
16
17  // AEG Write routines
18  extern "C" void cpWrAEG();
19  extern "C" void cpAE0WrAEG();
20  extern "C" void cpAE1WrAEG();
21  extern "C" void cpAE2WrAEG();
22  extern "C" void cpAE3WrAEG();
23
24  // AEG Read Routines
25  extern "C" long cpAE0RdAEG();
26  extern "C" long cpAE1RdAEG();
27  extern "C" long cpAE2RdAEG();
28  extern "C" long cpAE3RdAEG();
29
30  // Co-processor Call Routines
31  extern "C" void cpCaep00();
32  extern "C" void cpCaep01();
33  extern "C" void cpCaep02();
34  extern "C" void cpCaep03();
35  extern "C" void cpCaep04();
36  extern "C" void cpCaep05();
37  extern "C" void cpCaep06();
38  extern "C" void cpCaep07();
39  extern "C" void cpAE0Caep00();
40  extern "C" void cpAE0Caep01();
41  extern "C" void cpAE0Caep02();
42  extern "C" void cpAE0Caep03();
43  extern "C" void cpAE0Caep04();
44  extern "C" void cpAE0Caep05();
45  extern "C" void cpAE0Caep06();
46  extern "C" void cpAE0Caep07();
47  extern "C" void cpAE1Caep00();
48  extern "C" void cpAE1Caep01();
49  extern "C" void cpAE1Caep02();
50  extern "C" void cpAE1Caep03();
51  extern "C" void cpAE1Caep04();
52  extern "C" void cpAE1Caep05();
53  extern "C" void cpAE1Caep06();
54  extern "C" void cpAE1Caep07();
55  extern "C" void cpAE2Caep00();
56  extern "C" void cpAE2Caep01();
57  extern "C" void cpAE2Caep02();
```

```
58  extern "C" void cpAE2Caep03();
59  extern "C" void cpAE2Caep04();
60  extern "C" void cpAE2Caep05();
61  extern "C" void cpAE2Caep06();
62  extern "C" void cpAE2Caep07();
63  extern "C" void cpAE3Caep00();
64  extern "C" void cpAE3Caep01();
65  extern "C" void cpAE3Caep02();
66  extern "C" void cpAE3Caep03();
67  extern "C" void cpAE3Caep04();
68  extern "C" void cpAE3Caep05();
69  extern "C" void cpAE3Caep06();
70  extern "C" void cpAE3Caep07();
71
72  // Constructor
73  CPLib::CPLib(int debugLvl)
74          : mHasSig(false),
75            mCPCallCt(0),
76            mAEGWrCt(0),
77            mAEGRdCt(0),
78            mDebugLvl(debugLvl),
79            mMPIPServer(this, debugLvl) {
80      #ifdef CONVEY
81      cny_get_signature((char *) "pdk", &mSig, &mSig2, &mSigStat);
82      if (mSigStat) {
83           fprintf(stderr, "ERROR: CPLib::CPLib(): cny_get_signature() failed (%d)!\n",
               mSigStat);
84      } else mHasSig = true;
85      #else
86      mHasSig = true;
87      #endif
88
89      // Run this regardless--not like it consumes any resources...
90      //runMPIPServer();
91  }
92
93  // Destructor
94  CPLib::~CPLib() {
95      stopMPIPServer();
96  }
97
98  // Accessors
99  bool CPLib::hasSig() {
100     return mHasSig;
101 }
102 int CPLib::getCPCallCt() {
103     return mCPCallCt;
104 }
105 int CPLib::getAEGWrCt() {
106     return mAEGWrCt;
107 }
108 int CPLib::getAEGRdCt() {
109     return mAEGRdCt;
110 }
111
112 // Run MPIP server
113 void CPLib::runMPIPServer() {
114     mMPIPServer.start();
```

```
115   }
116
117   // Stop MPIP server
118   void CPLib::stopMPIPServer() {
119       mMPIPServer.stop();
120   }
121
122   // Allocate co-processor memory
123   void * CPLib::mallocCP(size_t size) {
124       #ifdef CONVEY
125       return cny_cp_malloc(size);
126       #else
127       return (void *) 0;
128       #endif
129   }
130
131   // Allocate co-processor memory and init with buffer contents
132   void * CPLib::mallocCP(const void *buf, size_t size) {
133       #ifdef CONVEY
134       void *ret = cny_cp_malloc(size);
135       memcpy(ret, buf, size);
136       return ret;
137       #else
138       return (void *) 0;
139       #endif
140   }
141
142   // Write the AEG registers
143   void CPLib::writeAEGReg(uint64 data, int aegIdx) {
144       writeAEGReg(data, aegIdx, -1);
145   }
146   void CPLib::writeAEGReg(uint64 data, int aegIdx, int aeIdx) {
147       mDIMutex.lock();
148
149       if (mDebugLvl >= 1)
150           printf("INFO: writeAEGReg(data=0x%016llx, aegIdx=%d, aeIdx=%d)\n", data,
                aegIdx, aeIdx);
151
152       // Invalid index flags
153       bool badAEIdx = false;
154
155       #ifdef CONVEY
156       // Switch on AE index
157       switch (aeIdx) {
158            case -1: copcall_fmt(mSig, &cpWrAEG,    "AL", (uint64) aegIdx, data); break;
159            case 0:  copcall_fmt(mSig, &cpAE0WrAEG, "AL", (uint64) aegIdx, data); break;
160            case 1:  copcall_fmt(mSig, &cpAE1WrAEG, "AL", (uint64) aegIdx, data); break;
161            case 2:  copcall_fmt(mSig, &cpAE2WrAEG, "AL", (uint64) aegIdx, data); break;
162            case 3:  copcall_fmt(mSig, &cpAE3WrAEG, "AL", (uint64) aegIdx, data); break;
163            default: badAEIdx = true;
164       }
165       #endif
166
167       // Check for invalid AE index
168       if (badAEIdx)
169           fprintf(stderr, "ERROR: CPLib::writeAEGReg(): Bad AE index (%d)!\n", aeIdx);
170
171       // Stats
```

```
172         mAEGWrCt++;
173
174         mDIMutex.unlock();
175     }
176
177     // Read the AEG registers
178     uint64 CPLib::readAEGReg(int aegIdx) {
179         return readAEGReg(aegIdx, 0);
180     }
181     uint64 CPLib::readAEGReg(int aegIdx, int aeIdx) {
182         mDIMutex.lock();
183
184         if (mDebugLvl >= 1)
185             printf("INFO: readAEGReg(aegIdx=%d, aeIdx=%d)\n", aegIdx, aeIdx);
186
187         // Invalid index flags
188         bool badAEIdx = false;
189
190         uint64 ret = 0;
191         #ifdef CONVEY
192         // Switch on AE index
193         switch (aeIdx) {
194              case 0: ret = l_copcall_fmt(mSig, &cpAE0RdAEG, "A", (uint64) aegIdx); break;
195              case 1: ret = l_copcall_fmt(mSig, &cpAE1RdAEG, "A", (uint64) aegIdx); break;
196              case 2: ret = l_copcall_fmt(mSig, &cpAE2RdAEG, "A", (uint64) aegIdx); break;
197              case 3: ret = l_copcall_fmt(mSig, &cpAE3RdAEG, "A", (uint64) aegIdx); break;
198              default: badAEIdx = true;
199         }
200         #endif
201
202         if (mDebugLvl >= 1)
203             printf("INFO: readAEGReg(aegIdx=%d, aeIdx=%d) => 0x%016llx\n", aegIdx, aeIdx,
                  ret);
204
205         // Check for invalid AE index
206         if (badAEIdx)
207             fprintf(stderr, "ERROR: CPLib::readAEGReg(): Bad AE index (%d)!\n", aeIdx);
208
209         // Stats
210         mAEGRdCt++;
211
212         mDIMutex.unlock();
213         return ret;
214     }
215
216     // Call co-processor instructions
217     void CPLib::execCPInstr(int instIdx) {
218         execCPInstr(instIdx, -1);
219     }
220     void CPLib::execCPInstr(int instIdx, int aeIdx) {
221
222         if (mDebugLvl >= 1)
223             printf("INFO: execCPInstr(instIdx=%d, aeIdx=%d)\n", instIdx, aeIdx);
224
225         // Invalid index flags
226         bool badAEIdx = false;
227         bool badInstIdx = false;
228
```

```
229         // Time execution
230         Timer t0;
231         t0.start();
232 
233         // Write custom instruction index [7:0] to AEG[1]
234         if (aeIdx < 0 || aeIdx > 3) {
235             fprintf(stderr, "ERROR: CPLib::readAEGReg(): Bad AE index (%d)!\n", aeIdx);
236             return;
237         }
238         writeAEGReg(1, aeIdx, instIdx & 0xff);
239 
240         // Poll AEG[1] until idle status
241         while (readAEGReg(1, aeIdx) >> 63)
242             usleep(500);
243 
244         #ifdef CONVEY
245         // Switch on AE index (-1 == all AEs)
246         switch (aeIdx) {
247             // AE0, AE1, AE2, AE3
248             case -1:
249                 switch (instIdx) {
250                     case 0: copcall_fmt(mSig, &cpCaep00, ""); break;
251                     case 1: copcall_fmt(mSig, &cpCaep01, ""); break;
252                     case 2: copcall_fmt(mSig, &cpCaep02, ""); break;
253                     case 3: copcall_fmt(mSig, &cpCaep03, ""); break;
254                     case 4: copcall_fmt(mSig, &cpCaep04, ""); break;
255                     case 5: copcall_fmt(mSig, &cpCaep05, ""); break;
256                     case 6: copcall_fmt(mSig, &cpCaep06, ""); break;
257                     case 7: copcall_fmt(mSig, &cpCaep07, ""); break;
258                     default: badInstIdx = true;
259                 }
260                 break;
261 
262             // AE0
263             case 0:
264                 switch (instIdx) {
265                     case 0: copcall_fmt(mSig, &cpAE0Caep00, ""); break;
266                     case 1: copcall_fmt(mSig, &cpAE0Caep01, ""); break;
267                     case 2: copcall_fmt(mSig, &cpAE0Caep02, ""); break;
268                     case 3: copcall_fmt(mSig, &cpAE0Caep03, ""); break;
269                     case 4: copcall_fmt(mSig, &cpAE0Caep04, ""); break;
270                     case 5: copcall_fmt(mSig, &cpAE0Caep05, ""); break;
271                     case 6: copcall_fmt(mSig, &cpAE0Caep06, ""); break;
272                     case 7: copcall_fmt(mSig, &cpAE0Caep07, ""); break;
273                     default: badInstIdx = true;
274                 }
275                 break;
276 
277             // AE1
278             case 1:
279                 switch (instIdx) {
280                     case 0: copcall_fmt(mSig, &cpAE1Caep00, ""); break;
281                     case 1: copcall_fmt(mSig, &cpAE1Caep01, ""); break;
282                     case 2: copcall_fmt(mSig, &cpAE1Caep02, ""); break;
283                     case 3: copcall_fmt(mSig, &cpAE1Caep03, ""); break;
284                     case 4: copcall_fmt(mSig, &cpAE1Caep04, ""); break;
285                     case 5: copcall_fmt(mSig, &cpAE1Caep05, ""); break;
286                     case 6: copcall_fmt(mSig, &cpAE1Caep06, ""); break;
```

```
287                 case 7: copcall_fmt(mSig, &cpAE1Caep07, ""); break;
288                 default: badInstIdx = true;
289             }
290             break;
291
292         // AE2
293         case 2:
294             switch (instIdx) {
295                 case 0: copcall_fmt(mSig, &cpAE2Caep00, ""); break;
296                 case 1: copcall_fmt(mSig, &cpAE2Caep01, ""); break;
297                 case 2: copcall_fmt(mSig, &cpAE2Caep02, ""); break;
298                 case 3: copcall_fmt(mSig, &cpAE2Caep03, ""); break;
299                 case 4: copcall_fmt(mSig, &cpAE2Caep04, ""); break;
300                 case 5: copcall_fmt(mSig, &cpAE2Caep05, ""); break;
301                 case 6: copcall_fmt(mSig, &cpAE2Caep06, ""); break;
302                 case 7: copcall_fmt(mSig, &cpAE2Caep07, ""); break;
303                 default: badInstIdx = true;
304             }
305             break;
306
307         // AE3
308         case 3:
309             switch (instIdx) {
310                 case 0: copcall_fmt(mSig, &cpAE3Caep00, ""); break;
311                 case 1: copcall_fmt(mSig, &cpAE3Caep01, ""); break;
312                 case 2: copcall_fmt(mSig, &cpAE3Caep02, ""); break;
313                 case 3: copcall_fmt(mSig, &cpAE3Caep03, ""); break;
314                 case 4: copcall_fmt(mSig, &cpAE3Caep04, ""); break;
315                 case 5: copcall_fmt(mSig, &cpAE3Caep05, ""); break;
316                 case 6: copcall_fmt(mSig, &cpAE3Caep06, ""); break;
317                 case 7: copcall_fmt(mSig, &cpAE3Caep07, ""); break;
318                 default: badInstIdx = true;
319             }
320             break;
321
322         default:
323             badAEIdx = true;
324     }
325     #endif
326
327     // Check for invalid AE index
328     if (badAEIdx)
329         fprintf(stderr, "ERROR: CPLib::execCPInstr(): Bad AE index (%d)!\n", aeIdx);
330     // Check for invalid instruction index
331     if (badInstIdx)
332         fprintf(stderr, "ERROR: CPLib::execCPInstr(): Bad instruction index (%d)!\n",
              instIdx);
333     if (badAEIdx || badInstIdx) return;
334
335     // Time stats
336     t0.stop();
337     if (mDebugLvl >= 1) {
338         if (aeIdx < 0)
339             printf("INFO: CPLib::execCPInstr(): caep%02d completed in %fms\n",
                  instIdx,
340                       t0.elapsed());
341         else
```

```
342                 printf("INFO: CPLib::execCPInstr(): caep%02d.ae%d completed in %fms\n",
                      instIdx,
343                         aeIdx, t0.elapsed());
344         }
345
346         // Other stats
347         mCPCallCt++;
348     }
349
350     // Simplified register write for LabVIEW designs
351     void CPLib::writeReg(uint64 data, int index) {
352         if (index > 3) {
353             fprintf(stderr, "ERROR: CPLib::writeReg(): Bad register index (%d)!\n",
                      index);
354             return;
355         }
356         writeAEGReg(data, index + 20);
357     }
358
359     // Simplified register read for LabVIEW designs
360     uint64 CPLib::readReg(int index) {
361         if (index > 3) {
362             fprintf(stderr, "ERROR: CPLib::readReg(): Bad register index (%d)!\n", index);
363             return 0;
364         }
365         return readAEGReg(index + 10);
366     }
367
368     // Simplified command execution method for LabVIEW designs
369     void CPLib::execCmd(int index) {
370         if (index > 3) {
371             fprintf(stderr, "ERROR: CPLib::execCmd(): Bad instruction index (%d)!\n",
                      index);
372             return;
373         }
374         // Assert bit <index> of aeg[0] and then clear it
375         writeAEGReg(0x1 << index, 1);
376         writeAEGReg(0, 1);
377         // Wait util busy flag goes low
378         while (readAEGReg(2) & 0x1);
379     }
```

Appendix B

MPIP Server Emulator

Listing B.1: mpipserver.h

```
1   #ifndef _MPIPSERVER_H
2   #define _MPIPSERVER_H
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <boost/shared_ptr.hpp>
7   #include <boost/thread.hpp>
8   #include <boost/asio.hpp>
9
10  using boost::asio::ip::tcp;
11
12  typedef unsigned long long int uint64;
13
14  class CPLib;
15
16  class MPIPServer {
17  public:
18      // Constructor--does not start the server thread
19      MPIPServer(CPLib *cpl, int debugLvl = 0, int listenPort = 2544);
20
21      // Destructor--stops the server thread if still running
22      ~MPIPServer();
23
24      // Returns true if thread is running
25      bool isRunning();
26
27      // Runs the server thread (run())
28      void start();
29
30      // Sets "stop requested" flag and waits for thread to terminate
31      void stop();
32
33  private:
34      // Pointer to CPLib instance
35      CPLib *mCpl;
36
```

```
37        // Thread object
38        boost::thread mThread;
39
40        // Boost io_service pointer
41        boost::shared_ptr<boost::asio::io_service> mIOService;
42
43        // Flags
44        bool mRunning;
45
46        // Config
47        int mDebugLvl;
48        int mListenPort;
49
50        // Thread function
51        void run();
52
53        // Asynchronous accept routines
54        void startAccept(tcp::acceptor& acceptor);
55        void handleAccept(boost::shared_ptr<tcp::socket> socket,
56                          const boost::system::error_code& error,
57                          tcp::acceptor& acceptor);
58
59        // Asynchronous socket read routines
60        void startRead(boost::shared_ptr<tcp::socket> socket);
61        void handleRead(boost::shared_ptr<tcp::socket> socket,
62                        const boost::system::error_code& error,
63                        char *data,
64                        size_t len);
65
66        // Parses, runs, and returns the results of incoming commands
67        std::string processCmd(std::string cmd);
68
69        // Cleans up the received command string
70        std::string cleanupCmd(std::string s);
71
72        // Converts hex string ('0x1234') to uint64
73        uint64 hex2uint64(std::string s);
74    };
75
76    #endif // _MPIPSERVER_H
```

Listing B.2: mpipserver.cpp

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string>
4   #include <boost/thread.hpp>
5   #include <boost/asio.hpp>
6   #include <boost/regex.hpp>
7   #include <boost/lexical_cast.hpp>
8   #include <boost/algorithm/string.hpp>
9
10  #include "mpipserver.h"
11  #include "cplib.h"
12
13  #define CSR_REQ_AEG_ADDR 0
14  #define CSR_REQ_CSR_ADDR 0x8007
15
16  using boost::asio::ip::tcp;
17
18  MPIPServer::MPIPServer(CPLib *cpl, int debugLvl, int listenPort)
19          : mCpl(cpl),
20            mDebugLvl(debugLvl),
21            mListenPort(listenPort) {
22      // ...
23  }
24
25  MPIPServer::~MPIPServer() {
26      if (mRunning)
27          stop();
28  }
29
30  bool MPIPServer::isRunning() {
31      return mRunning;
32  }
33
34  void MPIPServer::start() {
35      if (mDebugLvl >= 1)
36          printf("INFO:MPIPServer: Starting server\n");
37      mThread = boost::thread(&MPIPServer::run, this);
38      mRunning = true;
39  }
40
41  void MPIPServer::stop() {
42      // Check for stupid usage
43      if (!mRunning) {
44          fprintf(stderr, "ERROR:MPIPServer: Attempting to stop non-running MPIP server
              thread!\n");
45          return;
46      }
47
48      // Stop io_service
49      if (mDebugLvl >= 1)
50          printf("INFO:MPIPServer: Stopping server\n");
51      mIOService->stop();
52
53      // Wait for server thread to terminate
54      mThread.join();
55      mRunning = false;
56  }
```

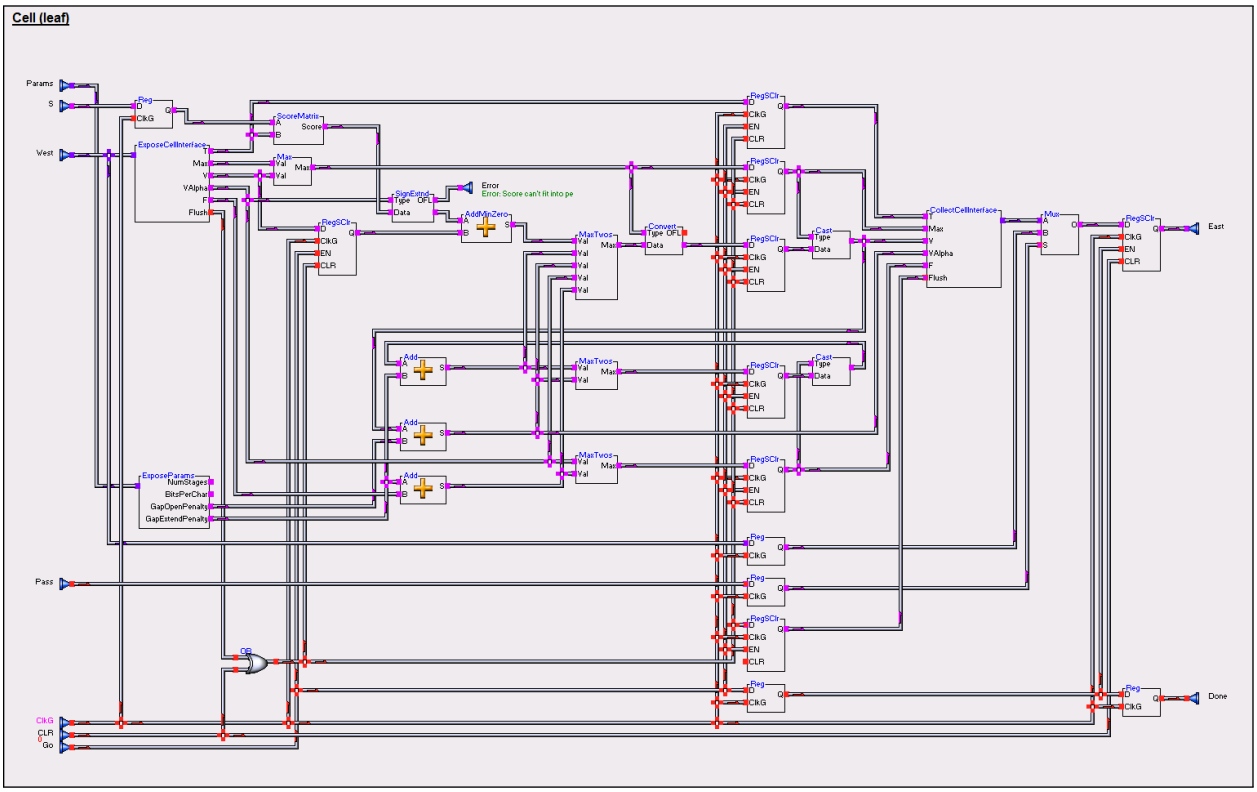
```
57
58  void MPIPServer::run() {
59      // Create new io_service
60      mIOService = boost::shared_ptr<boost::asio::io_service>(new
          boost::asio::io_service());
61
62      // Construct acceptor object with io_service and start accepting
63      tcp::acceptor acceptor(*mIOService, tcp::endpoint(tcp::v4(), mListenPort));
64      startAccept(acceptor);
65
66      // Run io_servce--this can be interrupted by io_service::stop()
67      mIOService->run();
68  }
69
70  void MPIPServer::startAccept(tcp::acceptor& acceptor) {
71      // Setup socket with io_service
72      boost::shared_ptr<tcp::socket> socket(new tcp::socket(acceptor.get_io_service()));
73
74      // Wait for connection asynchronously
75      if (mDebugLvl >= 2)
76          printf("INFO:MPIPServer: Waiting for TCP connection on port %d\n",
              acceptor.local_endpoint().port());
77      acceptor.async_accept(*socket, boost::bind(&MPIPServer::handleAccept, this,
          socket,
78          boost::asio::placeholders::error, boost::ref(acceptor)));
79  }
80
81  void MPIPServer::handleAccept(boost::shared_ptr<tcp::socket> socket,
82          const boost::system::error_code& error, tcp::acceptor& acceptor) {
83      // Check for fail
84      if (error) {
85          fprintf(stderr, "ERROR:MPIPServer: Error accepting connection: %s\n",
              error.message().c_str());
86          return;
87      }
88
89      // We have a client!
90      if (mDebugLvl >= 2)
91          printf("INFO:MPIPServer: Connection established with %s\n",
92              boost::lexical_cast<std::string>(socket->remote_endpoint()).c_str());
93
94      // Setup asynchronous callbacks on the socket
95      startRead(socket);
96
97      // Done with socket, start
98      startAccept(acceptor);
99  }
100
101 void MPIPServer::startRead(boost::shared_ptr<tcp::socket> socket) {
102     // Allocate buffer
103     char *data = new char[256];
104
105     // Asynchronous read with callback
106     socket->async_read_some(boost::asio::buffer(data, 256),
          boost::bind(&MPIPServer::handleRead, this,
107         socket, boost::asio::placeholders::error, data,
              boost::asio::placeholders::bytes_transferred));
108 }
```

```
109
110 void MPIPServer::handleRead(boost::shared_ptr<tcp::socket> socket,
111         const boost::system::error_code& error, char *data, size_t len) {
112     // Check for fail
113     if (error) {
114         if (error == boost::asio::error::eof) {
115             if (mDebugLvl >= 2)
116                 printf("INFO:MPIPServer: Connection closed by client\n");
117         } else
118             fprintf(stderr, "ERROR:MPIPServer: Error reading from socket: %s\n",
                error.message().c_str());
119         return;
120     }
121
122     // Check if socket was closed
123     if (!socket->is_open())
124         return;
125
126     // Parse command
127     std::string s = cleanupCmd(std::string(data));
128     if (!s.empty()) {
129         std::string ret = processCmd(s);
130         boost::asio::write(*socket, boost::asio::buffer(ret));
131     }
132
133     // Deallocate buffer
134     delete[] data;
135
136     // Start next asynchronous read
137     startRead(socket);
138 }
139
140 std::string MPIPServer::cleanupCmd(std::string s) {
141     using namespace boost::algorithm;
142
143     // Return trimmed first line of string
144     std::vector<std::string> lines;
145     split(lines, s, is_any_of("\r\n"));
146     trim(lines[0]);
147     return lines[0];
148 }
149
150 std::string MPIPServer::processCmd(std::string s) {
151     using namespace boost::algorithm;
152
153     if (mDebugLvl >= 3)
154         printf("INFO:MPIPServer: Parsing '%s'\n", s.c_str());
155
156     // Default response is syntax error
157     char ret[128];
158     sprintf(ret, "Syntax error!\n");
159
160     // Match command
161     boost::regex pat("(?<cmd>ae_csr_write|ae_csr_read)\\s+ae\\s+"
162                      "(?<aeidx>[0-3])\\s+(?<addr>0x[a-f0-9]{4})"
163                      "(?:\\s+(?<data>0x[a-f0-9]{16})(?:\\s+(?<mask>"
164                      "0x[a-f0-9]{16}))?)?",
165                      boost::regex::perl | boost::regex::icase);
```

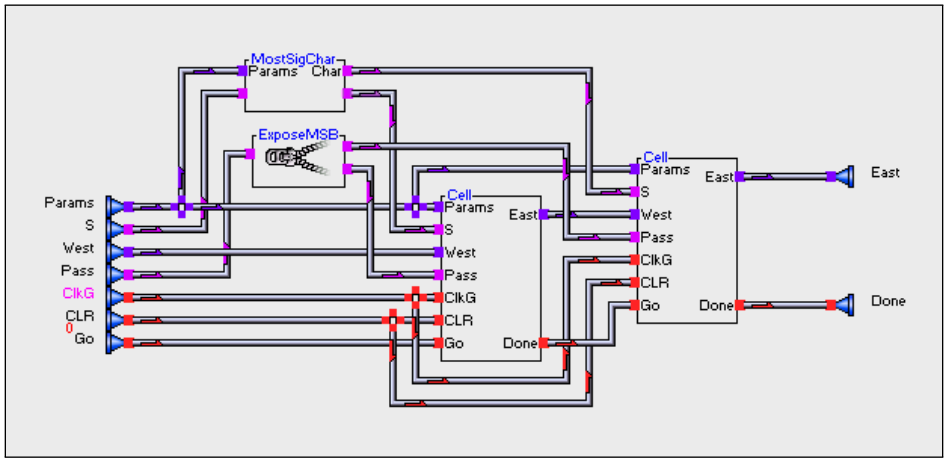
```
166       boost::smatch what;
167       if (boost::regex_search(s, what, pat)) {
168           // Extract parameters common to both read and write commands
169           int aeIdx = boost::lexical_cast<int>(what["aeidx"].str());
170           int addr = hex2uint64(what["addr"].str());
171
172           // Only do stuff on reg 0x8007
173           if (addr != CSR_REQ_CSR_ADDR) {
174               sprintf(ret, "This emulator handles transactions to only CSR register
                      0x8007\n");
175               return ret;
176           }
177
178           // Handle reads and writes
179           if (what["cmd"] == std::string("ae_csr_read")) {
180               // Retrieve read data and send to client
181               uint64 data = mCpl->readAEGReg(CSR_REQ_AEG_ADDR, aeIdx);
182               sprintf(ret, "AE %d 0x%04x: 0x%016llx\n", aeIdx, addr, data);
183           } else {
184               // Extract write-specific args
185               if (!what["data"].matched)
186                   return ret;
187               uint64 data = hex2uint64(what["data"].str());
188               uint64 mask = what["mask"].matched ? hex2uint64(what["mask"].str()) :
                      0xffffffffffffffffLL;
189               data &= mask;
190
191               // Send write request and send confirmation to client
192               mCpl->writeAEGReg(data, CSR_REQ_AEG_ADDR, aeIdx);
193               sprintf(ret, "Wrote AE %d 0x%04x to 0x%016llx with mask 0x%016llx\n",
                      aeIdx, addr, data, mask);
194           }
195       }
196
197       return ret;
198   }
199
200   uint64 MPIPServer::hex2uint64(std::string s) {
201       uint64 ret;
202       std::stringstream ss;
203       ss << std::hex << s;
204       ss >> ret;
205       return ret;
206   }
```

Appendix C

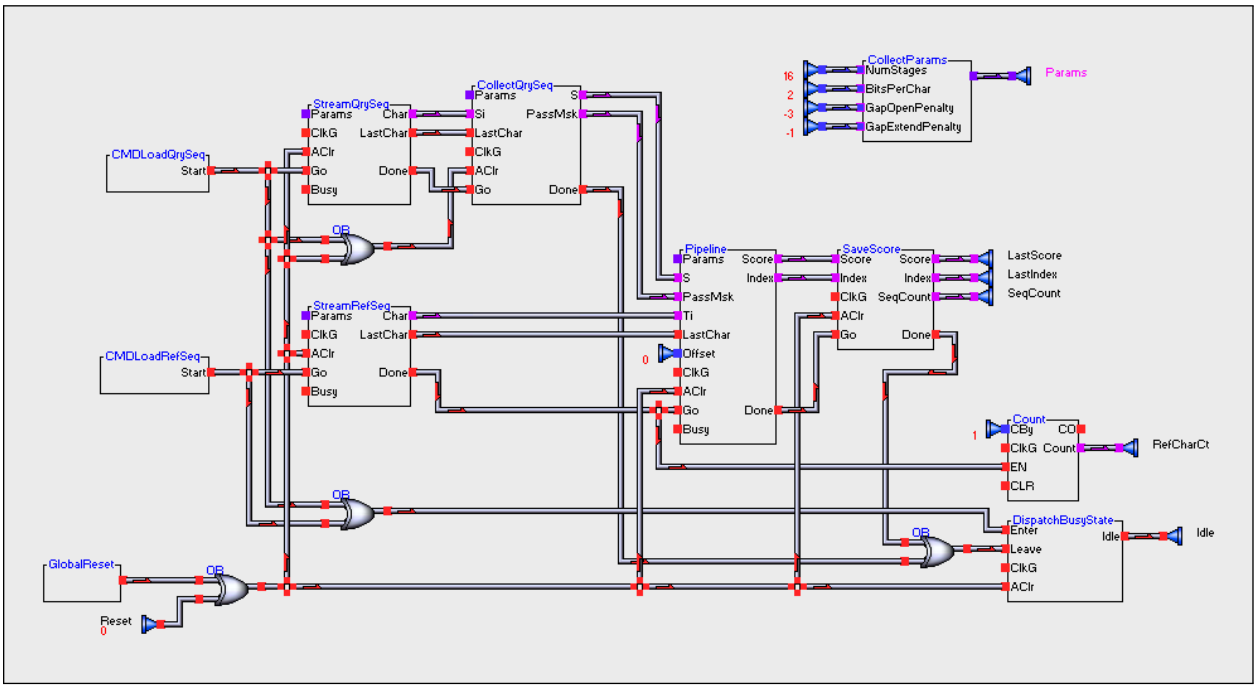
Smith-Waterman in bFlow



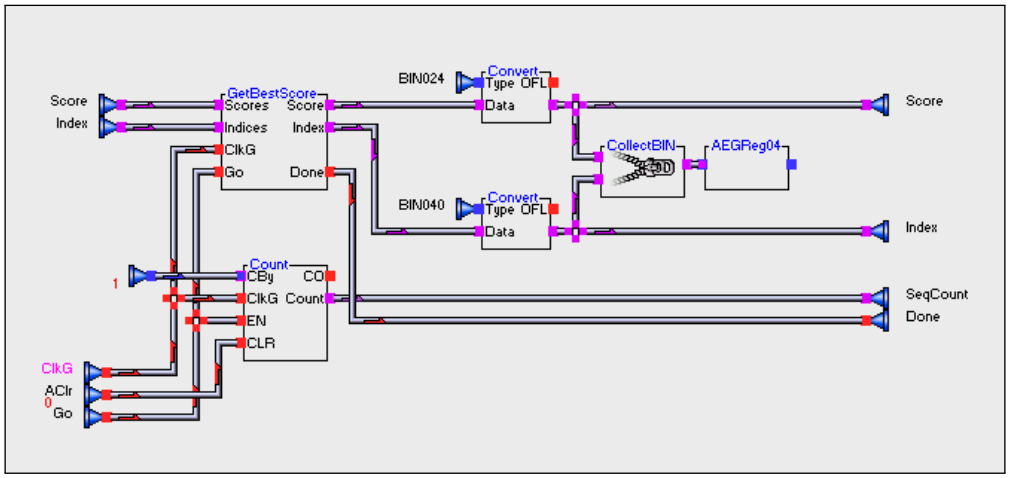
Smith-Waterman processing element object.



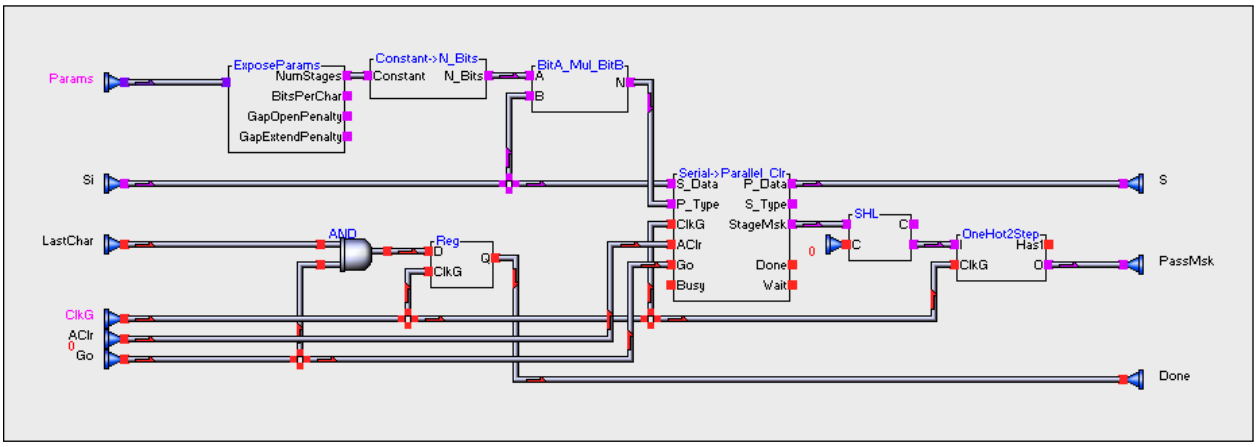
Recursive definition of processing element object. This enables replication of the PE as directed by the width of the S input port.



Top-level Azido canvas for the Smith-Waterman implementation.



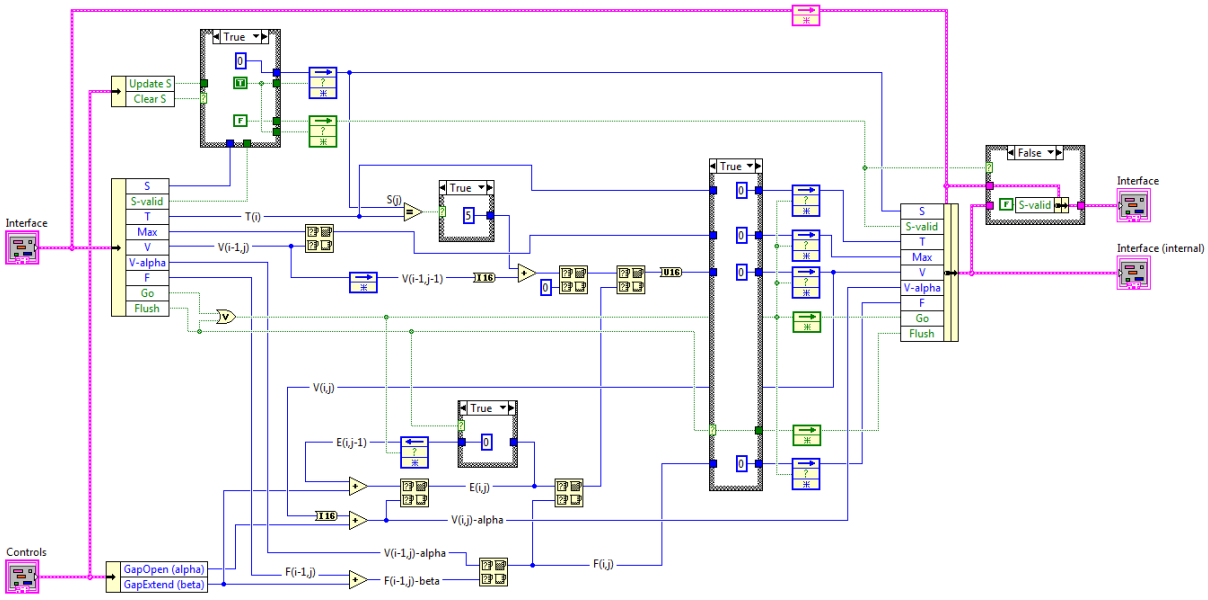
Object responsible for maintaining the best alignment's score and index.



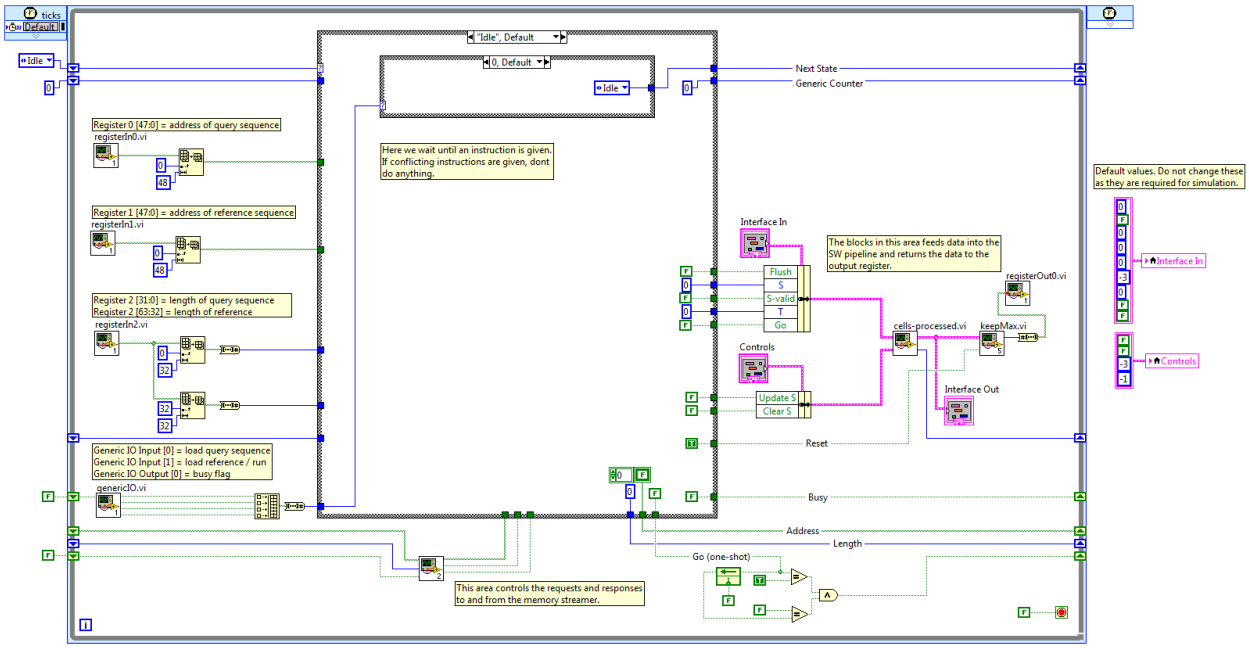
Query collector object, which feeds the query sequence to the pipline (see Figure 3.14).

Appendix D

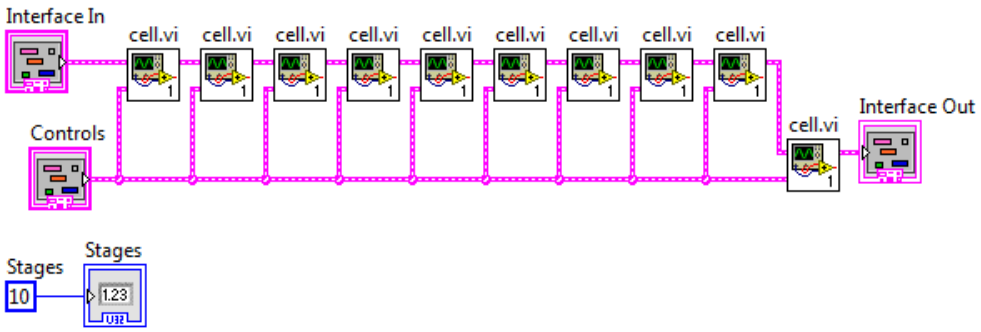
Smith-Waterman in ConVI



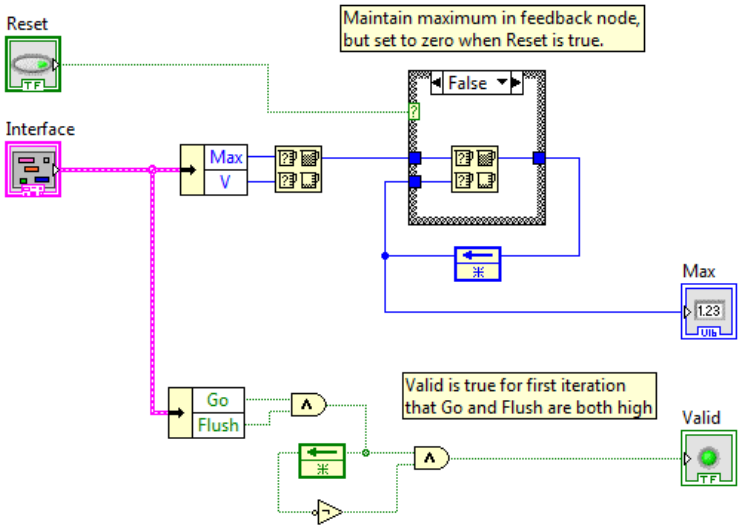
Smith-Waterman processing element sub-VI.



The top-level VI of the ConVI SW implementation. Centric to this VI is a large case container, which controls requests to and responses from memory, as well as inputs to the pipeline.



The systolic processing pipeline sub-VI generated by the LabVIEW script in Figure 3.18.



The *keepMax* sub-VI, responsible for maintaining the maximum alignment score produced by the pipeline. In the second workshop, as discussed in Section 4.1.2, the students extended this to track multiple best alignment locations.