DISSERTATION

AN ABSTRACT TARGET ARCHITECTURE FOR FPGA COMPILATION

Submitted by

Charles A. Ross

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

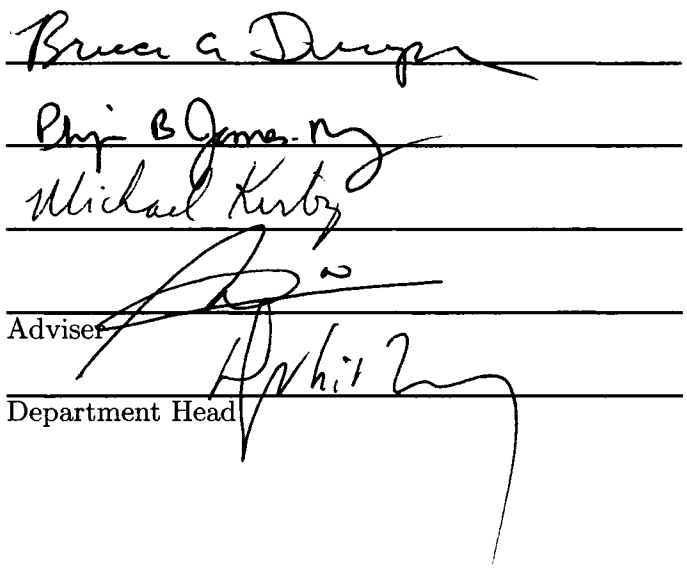
Fall 2006

COLORADO STATE UNIVERSITY

November 2, 2006

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY CHARLES A. ROSS ENTITLED AN ABSTRACT TARGET ARCHITECTURE FOR FPGA COMPILATION BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work



Adviser

Department Head

ABSTRACT OF DISSERTATION

AN ABSTRACT TARGET ARCHITECTURE FOR FPGA COMPILATION

Modern field programmable gate arrays (FPGAs) can outperform general purpose processors on a wide variety of tasks. They are particularly well suited for regular computations in which fine-grained parallelism can be exploited. Despite their tremendous potential for speedup, most high-performance application programmers do not use FPGAs because there is currently no widely accepted programming paradigm. Typical FPGA application design is done at a very low level and requires detailed knowledge of the target FPGA resources and hardware design practices in general.

The primary goal of this dissertation is to provide a flexible, high-performance framework for programming FPGAs, without requiring the user to be concerned with hardware design details. The Aggregated Hierarchical Abstract Hardware Architecture (AHAHA) presented here is the target architecture used in the SA-C* compiler; however, its concepts can easily be adapted to other compilers which use dataflow graphs as internal structures. It allows developers of high-level language compilers to concentrate on language features, program optimizations, and exploiting parallelism, without being concerned with the specifics of hardware design. By using the AHAHA framework as an intermediate form when targeting FPGAs, more advanced compilers can be developed to unleash the computational potential of FPGAs. The AHAHA embodies several novel ideas which contribute to its success, however, the principal contribution is its handshaking model which is based on "sections".

Using the SA-C compiler as an example, the research presented in this dissertation will answer questions regarding the ramifications of using an abstraction of the FPGA hardware. The AHAHA imposes a well-defined structure on the otherwise unconstrained hardware. This formal structure simplifies compiler development, and its impact upon clock frequency and area are negligible. The benefits of using an abstraction of the FPGA far outweigh the penalties. The AHAHA framework imposes no significant restrictions on the types of programs which may be implemented on the FPGA and maximizes efficiency via effective use of the hardware resources.

Charles A. Ross
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Fall 2006

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

When programming a standard CPU, a compiler is used to translate a high-level language into assembly language. The assembly is then converted into machine code which is executed by the processor. Each step in the process has a clearly defined target. When a high-level language compiler converts an input program into assembly language, the set of available assembly instructions is rigidly defined, and the behavior of each instruction is fixed and known. The semantics and complexity model of standard sequential execution on Von Neumann processors is very widely understood. Initially, processors were programmed directly in machine code. Later, assembly language was created to simplify programming and to make the processor more accessible to programmers. Finally, high-level languages, like C and Fortran, were developed to allow the programmer to think at even higher levels of abstraction. Each layer abstracts the underlying complexity without reducing expressibility. This multilayered approach simplifies software design, eases compiler development, and allows porting to new types of sequential hardware with minimal effort.

By contrast, an FPGA does not present a well-defined target architecture. Nearly any computational architecture can be implemented on an FPGA, including ASIC inspired designs, SIMD or MIMD parallel architectures, and Systolic arrays. Several Von Neumann processors have even been implemented in FPGAs as well, including the Xilinx MicroBlaze [1, 2] and PicoBlaze [3, 4], as well as the Altera Nios [5, 6] and Nios-II [7, 8] Processors. While processors embedded in FPGA fabric are not particularly efficient

compared with standard processors, they have been shown to be effective in a wide variety of FPGA-based tasks. The flexibility of FPGAs (while still enjoying the benefits of fine grained parallelism normally associated with rigidly defined ASICs) is the source of their incredible potential, but it is also the source of many programming obstacles. When programming an FPGA, the programmer must first settle on an appropriate architectural paradigm, then design the architecture itself, and in some cases, provide a software program to be executed by the final architecture. There are not many who can claim expertise in all of these areas at once, so expert FPGA programmers are rare.

Many users prefer to use high-level languages to describe their FPGA programs, and then rely on a compiler to do the low-level architectural design for them. Utilizing a compiler removes much of the user's detailed control over the implementation of the program, but also frees him/her from the low-level design difficulties, and speeds development time. However, the design difficulties have not been completely resolved; they have merely been shifted from being the responsibility of the programmer to being the responsibility of the compiler, and indirectly, the compiler developer. In many ways, these issues are compounded when moved to a compiler. The compiler must provide a general and robust solution, while a hand-implementation can make many simplifying assumptions which may only be valid for a particular program. This makes developing compilers which target FPGAs quite difficult, and sometimes leads to compilers which produce executables which work in only very particular circumstances, with very particular sets of inputs.

The goal of this work is to derive an abstract model of the FPGA hardware which provides functionality to the compiler developer without the typical complexity of general FPGA hardware design. The abstract model must provide concrete semantics, an overall simplifying structure, and a well-understood model of execution. In order to be fully effective, it should be a simple, high-performance, scalable solution which permits the compiler to represent varying levels of parallelism. It must also be expressive enough to provide access to all of the features and behaviors available in the hardware. To generate

efficient code, and facilitate design space exploration, it must provide the ability to make optimizations, and be highly modular so that adjustments made in one region of the program do not have far-reaching consequences in other areas. Finally, the model must be easily translatable into FPGA hardware, via a VHDL, Verilog, or netlist description.

The Aggregated Hierarchical Abstract Hardware Architecture (AHAHA) is an attempt to fulfill all of these criteria to free the compiler developer from hardware implementation details, static timing analysis, and complex monolithic VHDL descriptions. The AHAHA uses a novel approach to handshaking, based on graph "sections". This handshaking model provides the dynamic behavior of the AHAHA, while minimizing overhead. The section-based handshaking model is the primary contribution of this work. The handshaking approach used in the AHAHA allows data-driven dataflow computations (including a robust streaming model) to be implemented on FPGAs efficiently and easily.

There are four primary theses presented and defended in this work.

1. A flexible, dynamically scheduled system of small atomic functional units (such as the AHAHA) can be used in a data-driven model in order to implement complex high-performance computing applications on FPGAs.

2. Such a dynamic system can be used for FPGA design without accruing heavy penalties in FPGA area, execution time, expressibility, or complexity, over traditional methods with less dynamic behaviors.

3. A system such as AHAHA is an appropriate intermediate stage for high-level language compilation to FPGAs.

4. Using this design methodology, compilers are able to take advantage of feedback and additional information from the lower level to improve and report statistics about the final design.

In order to thoroughly investigate these points, the compiler for the second version of the SA-C language (called SA-C*) was designed to utilize the proposed AHAHA framework.

Although the SA-C* compiler is used, other compilers could be adapted to use the AHAHA for their target architecture, as long as they utilize an internal representation similar to the DFG, described in section 2.2.2.2. The performance and behavior of the AHAHA system is analyzed by using the SA-C* compiler to generate AHAHA graphs, and studying their runtime behavior in a high-level behavioral simulation environment, a low-level hardware simulation, and when executing on physical FPGA hardware.

The remaining chapters of this dissertation are organized as follows: Chapter 2 gives background information about FPGAs, the Cameron project, the (old) SA-C and (new) SA-C* languages, their compilers, and the preexisting intermediate structures used during compilation. Chapter 3 describes the AHAHA framework, the structure of AHAHA graphs, and the behavior of the individual AHAHA nodes, and the section-based handshaking model. Chapter 4 shows several examples of AHAHA graphs and gives examples of how AHAHA nodes can be composed to create common behaviors. Chapter 5 contains analysis of the behavior of AHAHA graphs generated from a variety of SA-C* programs, and of the AHAHA system in the large. Chapter 6 discusses related work done elsewhere, in relation to the work done here, and describes the relevance of this work to the FPGA community. And lastly, Chapter 7 draws conclusions about the AHAHA system, and outlines possible future work. Detailed descriptions of the individual AHAHA nodes can be found in Appendix A. Appendix B describes the hardware co-simulation environment developed to analyze and debug the behavior of the AHAHA generated hardware.

Chapter 2

Background

2.1 FPGAs and Reconfigurable Computing

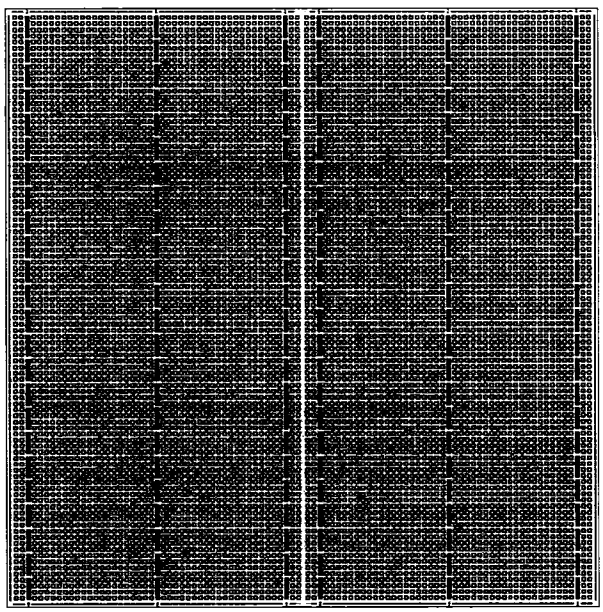


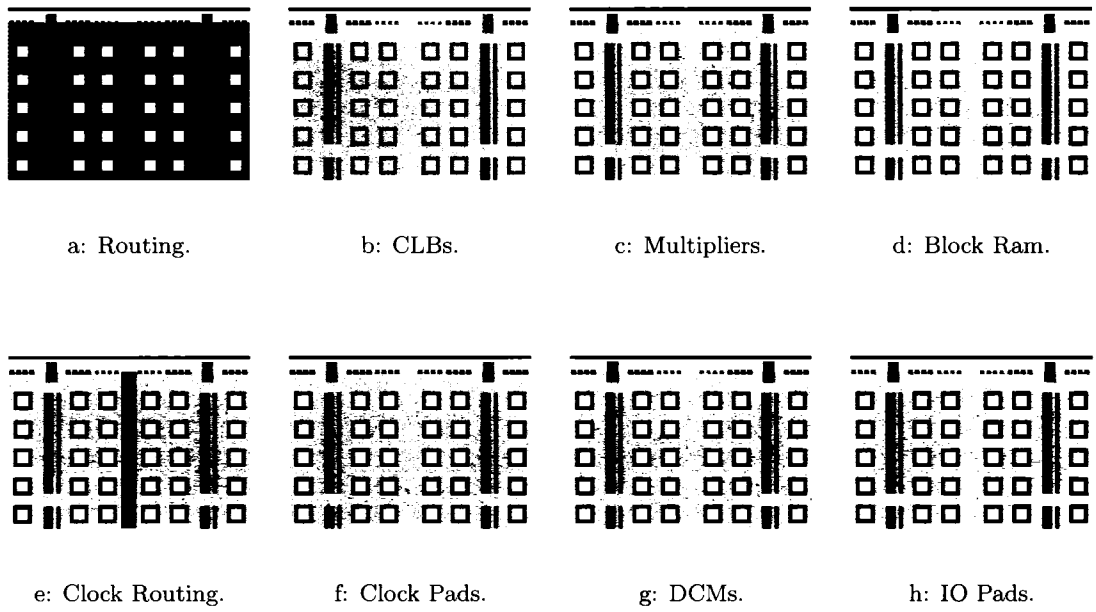
Figure 2.1: Virtex-II 6000 FPGA.

A field programmable gate array (FPGA) is a semiconductor device containing configurable logic blocks (CLBs) connected with programmable routing resources [9, 10]. The CLBs can be configured (using lookup tables, or LUTs) to implement any logical function, from simple boolean logic (ie, AND, OR, and NOT gates), to primitive circuits (ie, MUXes, decoders, and finite state machines), to complex mathematic functions. The CLBs also contain small memory elements, in the form of flip-flops, which can be used

to store state in sequential circuits. The programmable interconnect allows the CLBs to be connected to each other in order to create more complex behaviors. It is often convenient to envision an FPGA as a programmable breadboard. The CLBs are laid out in a 2-dimensional grid, with the routing resources between the CLBs running vertically or horizontally throughout the chip. The perimeter of the chip is connected to external pins via I/O Blocks (or IOBs). Each IOB can be used for input, output, or both (using tristate logic), and can optionally be registered within the IOB to provide sharp, glitch-free transitions at the cost of 1 clock cycle of latency. In addition to CLBs and IOBs, many modern FPGAs also contain a variety of coarse-grained functional units incorporated directly into the FPGA fabric. The Xilinx [11] FPGAs used in this research, for example, contain several small memories (called BlockRAMs), and embedded 18-bit signed integer multipliers. The multipliers and BlockRAMs are arranged in columns throughout the FPGA. Interfacing with external devices and signals is done via "pads" which connect to the conductive pins on the exterior of the chip encasement. The FPGA contains resources specifically for clock signals. These include pads specially designed to carry clock signals (located at the top and bottom of the chip), digital clock managers (DCMs) and dedicated routing resources to carry clock signals within the FPGA fabric. These clock specific resources minimize clock skew and timing glitches. Figure 2.1 shows the layout of one of our Virtex-II 6000 FPGAs. Figure 2.2 shows a closeup of the top center of the FPGA and highlights the various structures present within it.

In most cases, FPGAs are slower than ASICs[1] and usually consume more power. However, the time required to fabricate an ASIC is measured in days, weeks, and months, where an FPGA can be reconfigured in a fraction of a second. This high-speed runtime reconfiguration has precipitated a new computing model, termed "Reconfigurable Computing." In reconfigurable computing, FPGAs (or similar Reconfigurable Devices) are

[1] An exception can be made when the reconfigurability of the FPGAs can be exploited by folding runtime information directly into the hardware. This technique was used in a paper presented at FCCM 2000, which used FPGA technology to create the fastest hardware-based DES encryption routine available at the time.[12]



a: Routing. b: CLBs. c: Multipliers. d: Block Ram.

e: Clock Routing. f: Clock Pads. g: DCMs. h: IO Pads.

Figure 2.2: Closeups of top center of a Virtex-II 6000 FPGA.

used to build "just-in-time" hardware to perform computations when they are needed. After the computation is complete, the FPGA may be reconfigured to perform the next required computation using the same physical chip.

2.2 Cameron Project

The goal of the Cameron Project is to make FPGAs and other adaptive computer systems available to more applications programmers, by raising the abstraction level from hardware circuits to software algorithms. To this end, we have developed a variant of the C programming language and an optimizing compiler that maps high-level programs directly onto FPGAs, and have tested the language and compiler on a variety of image processing (and other) applications.[13]

The Cameron Project [14, 15] first developed SA-C, and later SA-C* to achieve the project goals. The languages are similar, and have the same basic goals, namely to provide a method of programming FPGAs from a high-level language. SA-C* is an extension of the older SA-C, to make it more expressive and include more language-level features.

2.2.1 SA-C

SA-C is often described as an "Extended Subset of C." While this description technically imparts no constraints whatsoever, it does embody the intention of SA-C to be as similar to C as possible, without slavishly implementing the entire language with no additions or omissions. For example, SA-C expressions are remarkably similar to C expressions, but the type system in SA-C is more flexible than, but still based on, C. The syntax and semantics of the C language are preserved whenever possible. However, because C is fundamentally imperative and sequential, many features of C are contrary to hardware execution (ie, pointers, dynamic memory, goto statements, etc.). These problematic portions of C have not been retained. On the other hand, there are many features in SA-C which have been added to simplify hardware generation, express parallelism, and make the resulting programs more efficient on the FPGA hardware. Because the SA-C language [16] was designed to be suitable for FPGA compilation, it contains many features, restrictions, and design decisions which are inspired by, and taken from, hardware design concepts.

- **Functional**: Because hardware circuitry has many similarities to dataflow graphs, an early decision was made that SA-C would use dataflow graphs internally to represent the hardware computations. Functional languages are particularly suited to generating dataflow, so SA-C is a functional language. For these same reasons, function calls must be inlined before conversion to hardware.

- **Single Assignment**: Variables in SA-C correspond to wires in hardware. Since wires can only have a single driver,[2] SA-C variables use single-assignment semantics. Once a variable as been assigned a value, it cannot be reassigned.[3] SA-C does

[2]SA-C does not attempt to generate tristate logic.

[3]In retrospect, this restriction is not strictly necessary because the other features and limitations of the SA-C language make it possible to convert SA-C programs into a static single-assignment representation automatically.

allow the value of a variable to change with each iteration of a loop, using "Nextification," and variable names may be reused in a later declaration[4]. The SA-C language documentation [16] contains detailed information on single-assignment restrictions.

- **Arbitrary Bit Precision**: In the FPGA, signals should be made as narrow as possible so as to not waste the chip area. C has a small number of integer types: `char`, `short`, `int` (or `long`), and (in some implementations) `long long` which are 8, 16, 32, and 64 bits respectively. In SA-C, the user can specify the bit-width of each variable arbitrarily. For example: a `uint4` is a 4-bit unsigned integer, capable of holding values from 0 to 15, while `int40` is a 40-bit signed integer capable of representing numbers from -549755813888 to 549755813887. In addition, the SA-C compiler does some analysis of programs to narrow the bit-widths of variables as much as possible without affecting program results.

- **Fixed-point Data Types**: On FPGAs, floating-point arithmetic is inefficient, and occupies a lot of chip area.[5] While SA-C does support floating-point data types, they are not supported by the hardware implementation. In order to execute programs which are based on real numbers, SA-C implements fixed-point data types. For example, in SA-C, `ufix16.4` is a variable with 16 bits of data, 4 of which are fractional bits, (the remaining 12 bits are the integer portion). On FPGAs, fixed-point arithmetic is not any more complex than standard integer arithmetic. Fixed-point data needs to be right/left shifted in order to align the radix point, but the shifts are all by constants. Shifts of this type are essentially free in the FPGA, because they do not require any logic; only the routing is affected.

---

[4]Because variables are lexically scoped, the latter declaration will shadow the former, effectively making the first declaration inaccessible for the remainder of the code.

[5]While this was definitely true when the Cameron Project was founded, due to advances in FPGAs, floating-point is becoming more attainable.

Since image processing applications were the primary focus of SA-C, it contains support for many common image processing data structures and access methods. For example, SA-C supports multidimensional rectangular arrays of scalars because images are essentially just 2-dimensional arrays of pixels (or 3-dimensional for multispectral imagery). Multiple images (frames) can be collected into higher dimensional arrays; however, the size of the arrays must be fixed. This means that dimensionality of the individual images must be the same, and the number of images must be static. SA-C also has extensive support for windowing access methods: ie, "for substructure in structure." For example, in SA-C it is trivial to construct a new image by applying a function to every 3x3 window in a source image. For the sake of simplicity, a trivial 1-D signal filter (shown in Figure 2.3) will be used in examples throughout this, and subsequent chapters. The result R is a vector produced by the convolution of a mask vector M over an input

$$R_i = \frac{\sum_{j=1}^{|M|} S_{i-j} \times M_j}{\sum_{j=1}^{|M|} M_i} \qquad M = \{1, 3, 3, 1\}$$

Figure 2.3: Simple 1-D signal filter.

vector S. The mask M forms a low-pass filter. For our purposes, we will assume that the input vector S contains non-negative integers less than or equal to 255. The SA-C program which implements this computation can be seen in Figure 2.4.

```
1: uint8[:]  main (uint8 S[:])  {
2:   uint8 M[4] = {1,3,3,1};
3:   uint8 R[:]  = for window W[4] in S {
4:     uint16 r = for w in W dot m in M
5:       return (sum((uint16) w*m));
6:   } return(array((uint8)(r/array_sum(M))));
7: } return (R);
```

Figure 2.4: Listing of filter_sac.sc.

2.2.2 Early SA-C Compiler

The SA-C compiler is implemented as several translation stages. Each stage brings the program closer to the eventual goal of executing on FPGA hardware. Several aggressive optimizations are applied at each of the translation phases. More information about the compiler optimizations can be found in [17, 18].

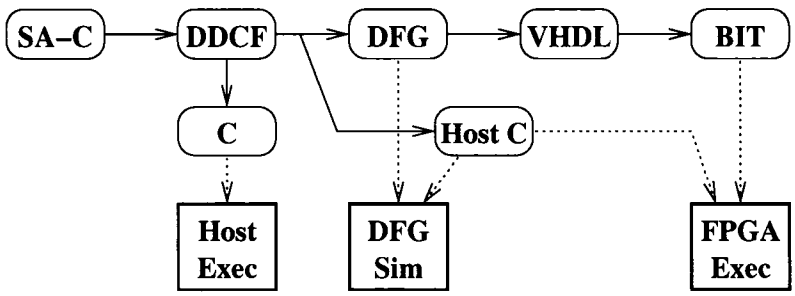


Figure 2.5: Early SA-C Compiler Pipeline.

Figure 2.5 shows the early compiler pipeline of transformation stages, and where the code can be executed or simulated. The compiler uses two internal forms (DDCF, and DFG). The purpose and structure of these two forms are discussed in the following sections.

2.2.2.1 DDCF

Initially, the SA-C code is translated into a Data Dependence and Control Flow (DDCF) graph, which is an abstract syntax tree of the program structure with additional dependence information; a DDCF graph is a direct translation of the program source code into a hierarchical graph-based representation. DDCF graphs do not have executable semantics; they merely describe the program's structure, and so cannot be directly executed. However, they can be translated to C which can then be executed on a host. This is indicated by the "Host Exec" in Figure 2.5. This allows the user to verify the correctness of the input SA-C code and the SA-C to DDCF translation phase. Since host execution requires no RCS hardware, and has the shortest compilation time, the user typically debugs their SA-C code in this way.

The DDCF is analyzed to identify innermost loops which are suitable for FPGA execution. These loops are converted to Dataflow Graphs or DFG [19]. The remaining portion of the DDCF is translated into host C annotated with calls to the appropriate runtime system, or RTS. The RTS allows the host to communicate with the hardware (or the DFG simulator) during program execution. Two different RTS libraries (with identical APIs) are used to communicate either with the simulator, or with the FPGA board. Figure 2.6 shows the runtime systems and communication paths to the execution models; areas in gray are the program which is being executed, areas in white are static. Only one of the RTS libraries can be used at any one time. The RTS libraries provide an API between the host and the underlying execution model. The host code is the same for all simulations, as well as for the final execution on the FPGA.

In the example program in Figure 2.4, the innermost loop which computes r is of known size (4 iterations), so the compiler will unroll it so that all iterations are done in parallel. The innermost loop that goes to the hardware will be the loop which computes the array $R[:]$. The DDCF is not shown, because it is difficult to depict, and does not provide any information which is not available in the SA-C program.

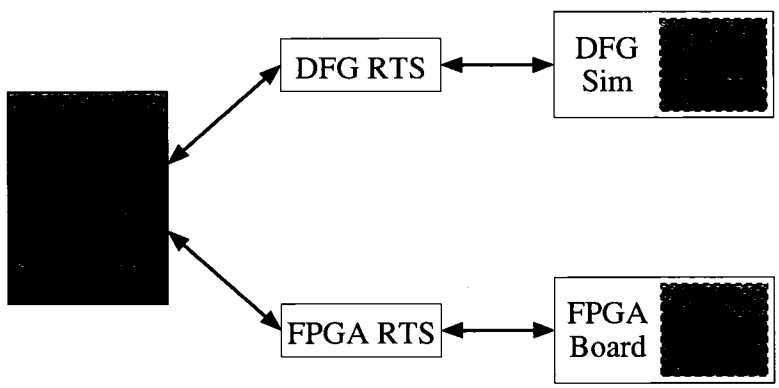


Figure 2.6: DFG and FPGA Runtime Systems.

2.2.2.2 DFG

DFGs are flat graphs of nodes with executable semantics which expose fine-grain, spatial, and pipeline parallelism. Each edge between DFG nodes contains an implicit unbounded

buffer to allow asynchronous execution of the nodes. The DFG nodes may encompass complex and general behaviors, such as loop generators, or loop collectors which would be difficult to implement in hardware. DFGs do not include a timing model; each node in the graph is free to execute when all of its input values are available.

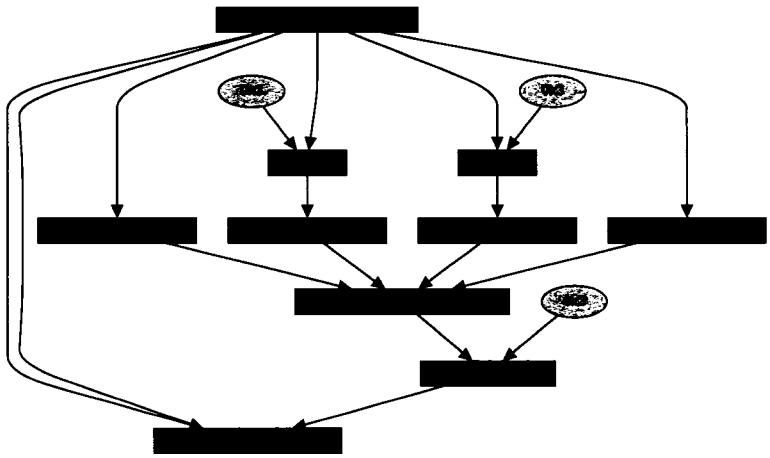


Figure 2.7: DFG of code in Figure 2.4.

Figure 2.7 shows the DFG graph generated from the SA-C code in Figure 2.4. The `RC_WINDOW_GEN_1D` node at the top produces every window of width 4 which is contained within the input array S, as well as control signals which go to the `WRITE_TILE_1D_1D` node at the bottom. The control signals are used to identify dummy iterations and the final iteration. Notice that the inner loop on line 5 has been fully unrolled, the vector M has been folded into the graph, and the multiplications by 1 have been optimized away. Also note that the division (by 8, the `array_sum(M)` has been evaluated at compile time) has been replaced with a `RIGHT_SHIFT` (by 3). Although it is not visible in the graph, the data values from the `RC_WINDOW_GEN_1D` are all 8 bits, the values from the `UMUL`s are all 10 bits, the outputs of the `CHANGE_WIDTH`s and the `REDUCE_SUM_MACRO` are all 11 bits, and the `RIGHT_SHIFT` produces an 8-bit value. The code in Figure 2.4, however, only has 8-bit values and 16-bit values. The reduction of the bit-widths is done automatically by the compiler via analysis of the value ranges of the input and output of each node, combined with the semantics of the node. For example, the `UMUL` node, multiplying an

arbitrary unsigned 8-bit value times the constant 3, can produce a maximum result of 765, which can be expressed using 10 bits.

2.2.2.3 DFG Simulator

DFGs can be executed using a simulator, as is indicated by "DFG Sim" in Figure 2.5. The DFG simulator can create a trace file as it executes the graph. The trace file contains a chronological recording of the graph execution. It describes every datum that flows over every edge in the graph. When a value is created, the trace file records it, along with which node number created it, and port of the node it was created on. DFG trace files can be used to compare the DFG execution to the execution of the host-only version. This is done by inserting debugging print statements into the SA-C code in critical areas, and comparing them with the values that were recorded in the DFG trace file.

2.2.2.4 TwoPE, OnePE

The early SA-C compiler attempted to map dataflow graphs directly to VHDL, using the TwoPE system [20]. In the TwoPE system, the loop being converted was divided into 3 segments. The first segment performed the loop generator(s), (`RC_WINDOW_GEN_1D` in this example). The second segment contained only the computational loop body of the loop (called the "Inner Loop Body," or ILB). The final segment was responsible for implementing the collector(s) (`RC_WINDOW_GEN_1D`). The loop generators were placed on the first (master) FPGA, the collectors on the second (slave) FPGA, and the ILB on either one. The communication path between the FPGAs was unidirectional and narrow (only 36 bits), which presented a major bottleneck either between the generators and the ILB, or between the ILB and the collectors. While not *entirely* unsuccessful, this approach presented several significant difficulties.

Every node in the dataflow graph needed to have a VHDL implementation, many of which had very complex behaviors. In the example shown in Figure 2.7 , the `RC_WINDOW_GEN_1D` must take the address in memory where S is stored, the length of S, as well as the width of the window (4), the stride of the window (1), the data width

of the elements (8 bits). It must then stream all of the data in from the memory, unpack it from 32-bit words into individual 8-bit values, and produce windows of 4 consecutive elements, stepping by value per window. The node must be flexible enough to work for any window width, window stride, data width, and image size. The VHDL for this is quite complex and fragile.

The graphs also needed to be analyzed and statically timed so that each node would execute at the correct time when its data was arriving, and the FPGAs would synchronize correctly. Obviously, this timing is dependent upon how each node behaves. To simplify the analysis, none of the nodes in the ILB could store any state, and consisted of combinational logic only. All of the state information was stored in either generators, or the collectors. If the VHDL of a stateful node is changed, the analysis must be modified to take the new behavior into account. The dataflow to VHDL translator contained hard-coded information about each node, such as their timing behavior, and port ordering. Since the loop body and collector(s) could not communicate information up to the generator(s), the timing behavior of both chips had to be taken into account when scheduling the computation.

As the source programs became more complex, the dataflow to VHDL translation became more fragile and harder and harder to maintain, until eventually it became the largest roadblock to a successful project.

Later, the TwoPE system was rewritten as the OnePE system [21], which worked in a similar manner, except that the generator, loop body, and collectors all resided on the same FPGA, and so could communicate their status to each other with no bottleneck. Memory arbitrators were added to help reduce the complexity of the timing analysis. This allowed for some simplistic dynamic scheduling, and alleviated a few of the problems associated with the static timing analysis, but the fundamental problems were still present.

In large programs, the ILB became very long and gave rise to incredibly long critical path delays. To overcome this, the ILB needed to be pipelined. But since there was

no state allowed in the ILB, a value to be pipelined had to be sent out of the ILB, into a stateful node outside the ILB, and then passed back into the ILB. In many cases, a value needed to be registered several times before its eventual use. This caused signals to "spiral" in and out of the ILB many times, adding new ports to the ILB entity on each cycle. Eventually ILBs became so complex that the synthesis tools began to malfunction during VHDL compilation. The OnePE and TwoPE systems simply did not scale effectively.

Without the benefit of a target architecture that abstracted from the underlying hardware implementation, the translation of dataflow graphs to hardware was just too wide of a gap to bridge. An intermediate, finer-grained structure was needed so the complex node behaviors could be implemented using smaller components, instead of creating large blocks of VHDL code for each type of node. Additionally, the restriction of having no stateful nodes in the ILB was causing many difficulties. However, adding state to the ILB would complicate (or preclude) static timing analysis of the loop. All of these issues needed to be addressed.

2.2.2.5 AHA

An intermediate, finer-grained structure was needed so the complex node behaviors could be implemented using smaller components, instead of creating monolithic VHDL code for each type of node. To this end, the Abstract Hardware Architecture (AHA) was created. AHA graphs were inserted into the compiler pipeline between DFG graphs and VHDL, as shown in Figure 2.8.

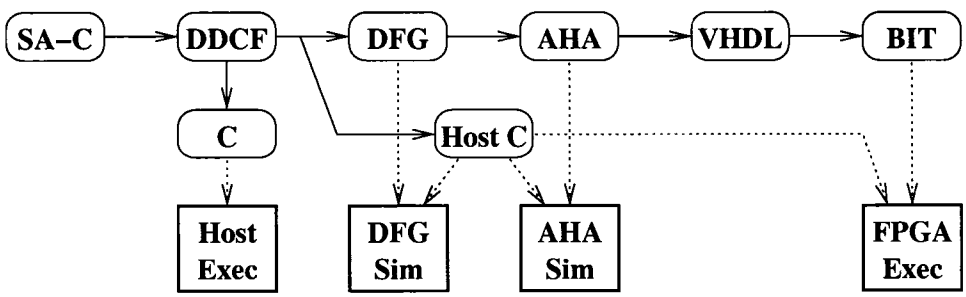


Figure 2.8: Newer SA-C Compiler Pipeline.

A copy of the DFG simulator was also augmented to be able to execute AHA graphs, producing "AHA Sim" in Figure 2.8. Additionally, an AHA RTS was created (see Figure 2.9) so that the host executable could communicate with the AHA simulator to provide a high-level behavioral simulation of the AHA layer, which could be analyzed to debug the behavior and structure of the AHA graphs. The simulation behavior of the AHA nodes was designed to mimic their VHDL description as closely as possible, but slight deviations were necessary in a few cases.

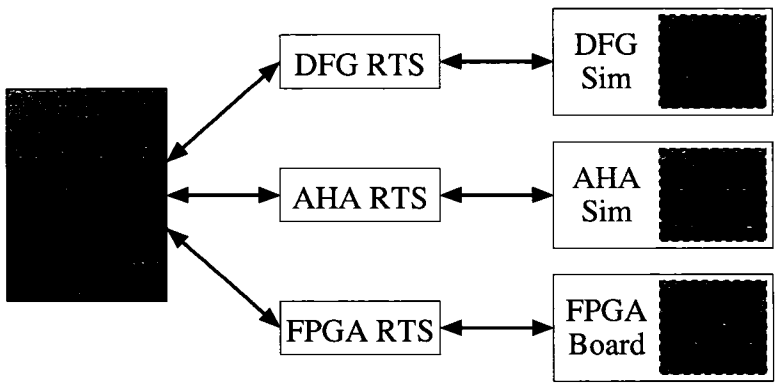


Figure 2.9: DFG, AHAHA and FPGA Runtime Systems.

The various AHA nodes were invented as needed, and no attempt was made to formalize the semantics or execution model of the graphs. Each node was implemented independently of the rest, and were designed only to fit together in very particular ways. As a result, the AHA was limited to expressing the constructs represented in the existing DFGs.

The most significant change from the OnePE/TwoPE systems to AHA was that the AHA nodes in the ILB were allowed to contain state, and the graph did not require any static timing analysis. Instead, each node communicated locally with its immediate neighbors to synchronize their behavior, and identify when to transfer data in a data-driven fashion. Like much of the AHA, the synchronization method was not formalized, and was often difficult to understand. Despite its obvious limitations, and lack of formalization, the AHA was a major breakthrough for the Cameron Project. It allowed

very complex programs to be executed on the hardware, and was much more scalable than the older systems.

2.3 SA-C*

For her PhD dissertation research, Monica Chawathe expanded the SA-C [22] language and compiler to include support for several new features:

- **Nested loops**: Loops with loops inside them are now permitted at the hardware level. In old SA-C, only a single, inner most loop could be targeted to compile for the hardware.

- **While-loops**: SA-C included while-loops, but they were never implemented on the hardware, because only loops whose iteration counts are statically known before execution were supported. Support for hardware while-loops was included in the expansion.

- **Multiple loops**: In addition to loop nesting, hardware loops can now be placed one after another sequentially, as in a producer/consumer relationship, or in parallel, as in a coarse-grained dataflow parallel network.

- **Processes connected by streams**: The largest change is the addition of a stream model. It is similar to the Kahn process model[23, 24], except that the buffers within the streams are finite. It allows more coarse-grained parallelism in the user codes. SA-C only had strict data structures, with known dimensions. Streams allow controlled non-strictness, while retaining many of the benefits of strict data structures. The elements within the stream remain strict.

The new language is called SA-C*. The * represents the Kleene star, referring to the addition of streams. In the old SA-C language, the compiler could easily determine which portions of the code were intended for the hardware (only innermost loops which did not contain any features which were not implemented in hardware), so the code partitioning could be done automatically. In SA-C*, however, there are many more

valid ways to partition the codes, and the compiler could not make the determination automatically. Instead, a *hardware* statement is used to declare the code partitioning scheme. Figure 2.10 shows the SA-C* version of the program from Figure 2.4.

```
1: uint8[:]  main (uint8 S[:])  {
2:   uint8 M[4] = {1,3,3,1};
3:   hardware () {
4:     uint8 R[:]  = for window W[4] in S {
5:       uint16 r = for w in W dot m in M
6:         return (sum((uint16) w*m));
7:     } return(array((uint8)(r/array_sum(M))));
8:   };
9: } return (R);
```

Figure 2.10: Listing of filter_sac_star.sc.

Unfortunately, the AHA did not support any of the new SA-C* features, so it needed to be expanded. Instead of merely augmenting the existing ad hoc AHA system to include these new features, the concepts in the AHA were formalized, a concrete execution model was created, and support for aggregation (stream connected processes) and hierarchy (nested loops) was included. The new model is the Aggregated Hierarchical Abstract Hardware Architecture, or AHAHA, which is discussed in full in the following chapter.

2.4 Alpha Data ADM-XRC-II

The SA-C compiler is able to generate co-designs for multiple types of FPGA boards containing Xilinx Virtex [25] chips. The compiler is able to target Annapolis Micro Systems [26] StarFire [27] and WildStar [28] boards, as well as Alpha Data [29] ADM-XRC-II [30] boards. The Alpha Data ADM-XRC-II board used in this research contains a single Xilinx Virtex-II 6000 chip, which is substantially larger than the chips present on the other target platforms currently available. The size of the chip on the board makes the Alpha Data ADM-XRC-II preferable over the other platforms available, because it enables larger designs, and experiments.

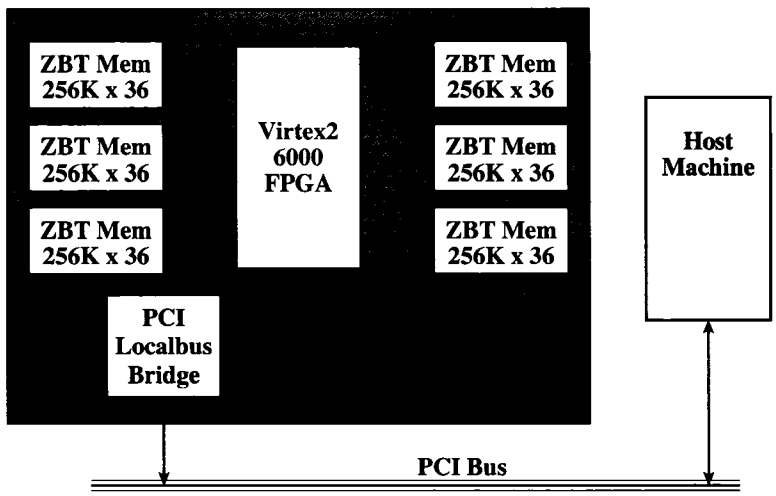


Figure 2.11: Alpha Data ADM-XRC-II board.

The ADM-XRC-II board contains 6 independent Zero Bus Turnaround (ZBT) Synchronous Static Random Access Memory (SSRAM) chips, a PCI localbus bridge which connects the FPGA to the host machine, DMA controllers for performing DMA transfers, and programmable clocks which allow the clock frequencies used for the on-board devices (including the memories, FPGA, and localbus) to be changed. Figure 2.11 shows a logical block diagram of the portions of the board which we utilize in our co-designs.

The memories on the ADM-XRC-II are 8 megabyte pipelined ZBT SSRAM memories from Micron [31]. Each is capable of storing 262144 (256K) words, with each word consisting of 4 bytes (32 bits in all). The memories support access to the individual bytes in each word via 4 byte-enable control pins. In addition, each byte is accompanied by a parity bit, which can be used to provide error detection, or to increase the total capacity of each word to 36 bits. The byte level access and parity bits are not used in this research; each memory cell is treated as a single 32-bit word. Traditional SSRAMs are capable of back-to-back reads, or back-to-back writes, but require a few clock cycles to switch from reading to writing, or vice versa. This delay is referred to as "bus turnaround." ZBT memories are a type of SSRAM which require no bus turnaround latency. That is, no clock cycles are wasted when switching between reads and writes. This allows the FPGA design more flexibility (and less pre-planning) when performing memory accesses. The

memories are able to perform accesses with 2 clock cycles of latency. After requesting a read from the memory, the value is available 2 clock cycles later. However, in order to minimize the risk of short circuits while using the tristate logic interface to the memory, the inputs and outputs connected to the memories are registered in the I/O Blocks of the FPGA, so the effective latency for any memory access is 4 clock cycles. Because the memories are pipelined, a new access may be requested before the previous one has completed. Therefore, at any time there may be up to 4 memory requests (at various stages of completion) in flight, and data may be streamed to/from the memories at the sustained rate of 1 value per clock cycle.

The ADM-XRC-II localbus is a simple 32-bit bus which provides 8 megabytes of address space, which can be allocated arbitrarily, to allow access to any of the ZBT memories or other devices inside the FPGA. Like the memories, the localbus provides 4 byte-enable control pins which allow selective access to the bytes within each 32-bit word. This feature is not used in this research; all transfers over the localbus are complete words which are aligned to a 32-bit boundary. 6 megabytes of the 8 available addresses are used to communicate with the ZBT memories. This leaves two additional megabytes of addresses which can be used to communicate with components inside the FPGA. The local bus supports pipelined read access, which allows burst transfers of large consecutive blocks of values. While this increases the throughput of the bus, it also creates some potential issues involving overrun conditions where the FPGA may anticipate reads which will not be requested by the host. This must be taken into account in the FPGA design. Write transfers require no pipelining.

The handling of the localbus is done by the PCI localbus bridge. The bridge is a PLX PCI 9656 chip [32]. It allows the localbus to operate independently from the PCI bus at any clock frequency from 25 to 66 MHz. Although the bridge is theoretically able to operate below 25 MHz, the DCM used to synchronize the external clock with the clock inside the FPGA is unable to operate reliably below 25 MHz. The PCI bus operates

at 66 MHz, and because the localbus is tightly coupled to the PCI bus, the maximum operational speed of the localbus is also 66 MHz.

The localbus bridge has the added advantage of making host-initiated DMA transfers indistinguishable from a burst of PCI traffic. This means that no additional logic is required inside the FPGA to support DMA transfers. In order to use full DMA speeds, the host simply initiates the transfer, and the localbus bridge handles the specifics of the DMA control, and accesses the FPGA in the same manner as it would for standard PCI I/O. In the systems described in this research, any transfer of more than a few words is performed using DMA. This includes the transfer of the configuration bitstream into the FPGA.

Chapter 3

AHAHA

The Aggregated Hierarchical Abstract Hardware Architecture (AHAHA) is a small collection of simple nodes, which are combined in a graph structure to describe the behavior of SA-C* programs. The execution of the graph is governed by a powerful, but simple, section-based handshaking protocol, which provides a dynamic execution model, with very little overhead. This handshaking model frees the compiler developer from the need to schedule the detailed behavior of the graph, while allowing each node to execute in a data driven fashion. The VHDL implementation of each of these nodes is small and maintainable, with each node averaging about 100 lines of VHDL. While the behavior of each node in itself is quite simple, their combined behavior can be arbitrarily complex. Using the AHAHA frees the compiler developer from hardware implementation details, static timing analysis, and large monolithic VHDL descriptions.

The process of converting from DFG to AHAHA is similar to macro expansion; each node in the DFG is converted into a functionally equivalent set of simpler AHAHA nodes. For example, a DFG loop generator which reads tiles of an image from memory will be converted into several groupings of nodes: One group generates loop indexes, another group provides addresses in the correct order to the memory, yet another group repackages the words arriving from the memory into a form that the loop body can use. A specific example is presented in Section 3.4.2. Further examples of how AHAHA nodes can be combined to generate more complex behaviors are given in Chapter 4.

The purpose of the AHAHA layer is to provide an executable abstract machine, which is directly implementable in hardware, to the higher levels of a compiler. The form and function of the final VHDL is dictated directly by the AHAHA graph and its execution model. Unlike DFGs, AHAHA graphs have an implicit timing model similar to a hardware clock. Each AHAHA node describes a small atomic operation and has a corresponding VHDL entity/architecture implementation.

Fundamentally, the AHAHA is a refinement of the earlier AHA, with clearly defined semantics and behavior. Like the AHA, it does not make a distinction between nodes in the ILB and nodes elsewhere; all nodes may optionally contain state regardless of their location. In the spirit of this homogenization, the boundary which separated the ILB from the surrounding loop control nodes has been completely removed to create a single simpler graph. The AHAHA includes many nodes which were not present in the AHA. These additional nodes increase expressiveness of the graphs, and facilitate access to the special purpose resources of the chip, such as BlockRAMs, SRL16s, and embedded multipliers. The handshaking protocol, which allowed the AHA nodes to determine when to fire dynamically, has been formalized and expanded in the AHAHA to create more complex graph structures and behaviors while maintaining simplicity. The communication pathways between nodes in the graph and off-chip resources (such as memories on the RCS board, and the host) have been generalized and given clearly defined behavior.

3.1 AHAHA Motivation and Design

The AHAHA was designed to address the shortcomings of the OnePE, TwoPE, and AHA systems. Its structure was devised as a solution for the following requirements and considerations:

- **State Throughout Graph**: In the OnePE and TwoPE systems, no state was permitted within the ILB, so that it could be made entirely combinational. This restriction on the placement of state caused several significant difficulties. These

early difficulties motivated a requirement on the AHAHA that state must be permitted to occur anywhere in the graph. In hardware circuit design, state equates to registers, which in turn, requires a clock signal.

- **Combinational Arithmetic**: Often, applications suitable for FPGAs perform a large number of simple arithmetic operations and boolean operators (AND, OR, NOT, etc.). When the output of one of these operators is directly utilized by another as input (without a register separating them), they are often merged by the synthesis tools to create a larger boolean operator. Allowing combinational operators to be adjacent to one another causes them to execute in the same clock cycle, and often reduces the area utilization of the chip. This requirement makes the AHAHA deviate from traditional dataflow (and the SA-C* generated DFG graphs) in that multiple nodes may execute within a single time step. The AHAHA needs to support stateless combinational operators, and allow them to be used in this way.

- **Dynamic Data-Driven Scheduling**: Statically timing graphs (as in TwoPE), has several complicating factors. Most notable are the limitations presented by the hardware resources. For example, memories only permit a small number of accesses to begin in any clock cycle[1]. The possibility that there may be more simultaneous accesses requested by the graph than can be serviced at once must be taken into account when scheduling the graph. Additionally, static analysis can become complicated when the timing of the program is data dependent. In such cases, static timing can result in inefficient worst case scenario timing, in which many clock cycles are wasted. Dataflow graphs (from which the AHAHA graphs are generated) are data-driven. Data-driven execution is the ultimate in dynamic scheduling, where each node is responsible for identifying when it is able to execute based on the presence or absence of its inputs; nodes simply fire when their

[1]Most memories allow only a single memory access per clock cycle.

inputs are available. Ideally, the AHAHA should provide a similar functionality to simplify translation from dataflow to AHAHA graphs. However, presence bits on every signal may be overly wasteful.

- **Robust Execution Semantics**: Typical data-driven operators stall until all of their inputs are available. When a node fires, it consumes data on all of its inputs and produces data on all of its outputs. However, there may be cases in which it may be preferable to relax these firing rules, such as:
  - allowing a node to accept some of its inputs before the rest are available
  - allowing a node to accept values without producing any
  - allowing a node to produce values without accepting any

  One situation in which this may be required is when constructing reduction operators. A reduction node may first receive an initial value on one of its inputs, followed by a sequence of operands on another input, and finally produce its result when the sequence has terminated. There is no need for the node to wait for the first value in the sequence of operands before consuming the initial value, nor is there a reason to require that the node produce a value after consuming every element of the sequence. The AHAHA must support such nodes.

- **Memory Arbitration**: Given that the graphs will be dynamically timed in a data-driven way, there is no way to predetermine which memories will be accessed at what time. This introduces the possibility of collisions where multiple accesses occur simultaneously. One possible method of addressing this is to time multiplex the memories, so that each of the n potential memory clients receive $\frac{1}{n}$th of the clock cycles in which to initiate a transaction. Time is allocated in a round robin way. However, this method is potentially inefficient. For example, imagine a situation in which there are two memory clients in the graph accessing the same physical memory. The first client (A) is attempting to access the memory in every clock cycle, while the second (B) only needs access in $\frac{1}{10}$th of the cycles. Using

time multiplexing, both A and B will be allocated $\frac{1}{2}$ of the available cycles. In this simple model, client B is being allocated more memory access than it requires; $\frac{4}{5}$ths of the time slices allocated to client B are wasted, even though client A could have made use of them. The issues become even more complex when the number, and pattern, of memory requests is data dependent, or when the access rates of the various memory clients are related to one another. In such situations such as these, the memory will be underutilized and the program execution speed will be degraded. The AHAHA runtime system must perform dynamic arbitration of memory accesses so that the memory is only idle when no accesses are requested. Using this model, client A will be dynamically allocated $\frac{9}{10}$ths of the clock cycles, while client B will be given $\frac{1}{10}$th; the memory bandwidth will be fully utilized.

- **Unbuffered Edges**: DFG graphs rely on edges which contain an unbounded buffer. An infinite buffer is unimplementable in hardware. One can envision a system where each edge contained a large, but finite, buffer to prevent deadlock in the graph. Unfortunately, determining the required buffer sizes in the general case is undecidable. Moreover, graphs often contain many edges, and it is not desirable to waste a large amount of resources on each one. Instead, AHAHA should use edges which are more similar to signals in hardware circuits, where each edge may contain only one unbuffered value at a time. If buffers are needed for efficiency or to prevent deadlock, they should be included explicitly.

- **No Global Data**: The AHA graphs contained 2 different types of signals: global and non-global. Global signals consisted primarily of graph inputs and other runtime static values. It was thought that runtime static values should be treated differently than the rest of the data in the graph since they did not require registers within the graph. However, the requirement that the compiler must treat these two types of signals differently complicated the compiler construction. Furthermore, it was determined that the area saved by not placing registers on runtime

static values was negligible. The AHAHA removes this distinction and treats all signals equivalently.

- **No Hard-coded Information or Redundancy**: In TwoPE, OnePE, and AHA, the compiler needed to contain hard-coded information about each node, and their implementation. Whenever the VHDL implementation of a node was modified in such a way that would affect its timing, the source code of the compiler needed to be modified to take the new behavior into account. Because the information was redundantly stored in more than one location, the compiler development was made more difficult. The AHAHA must provide a single location where all information about the nodes, including their timing behavior, latency, and port definitions are stored.

These requirements are the basis of the AHAHA, and directly influence its form and function. Because the AHAHA needed to incorporate both stateless combinational operators and stateful clocked behavior, there needed to be 2 classifications of nodes. These classifications were termed "unclocked" and "clocked" nodes, respectively. Unclocked nodes represented combinational circuits in the hardware, while clocked nodes represented sequential operations. This design decision solved the requirements of "state throughout graph," and "combinational arithmetic." In fact, because these requirements are essentially mutually exclusive, the only possible way to permit both was to treat them independently.

The next design consideration for the AHAHA was to devise a scheduling method which would take into account the "dynamic data-driven scheduling," "robust execution semantics," "memory arbitration," and "unbuffered edges" requirements. Combinational (unclocked) nodes already have execution semantics inherited from hardware circuits. Every time any of the inputs of a combinational circuit change, outputs of the circuit are recomputed. Because combinational circuits are unable to determine whether their inputs are valid, they can produce meaningless results when given meaningless inputs. The AHAHA scheduling paradigm must involve only the stateful nodes, and take into

account the possibility of edges of the graph containing undefined (or invalid) data. The AHAHA addresses these requirements using clusters, sections, and handshaking, all of which focus on clocked nodes.

The term "cluster" refers to a group of input ports or output ports of a node which will always consume or produce values in the same clock cycle. Each port of a node is contained in exactly one cluster, and no cluster may consist of both input and output ports. In the AHAHA model, unlike typical dataflow systems, nodes do not fire as a complete unit; rather, the individual clusters of a node may fire exclusively. An example of a node, and its clusters is shown in Figure 3.1. In this example, the input ports are divided into 3 clusters, and the output ports are divided into 2 clusters, (separated by the red dotted lines).

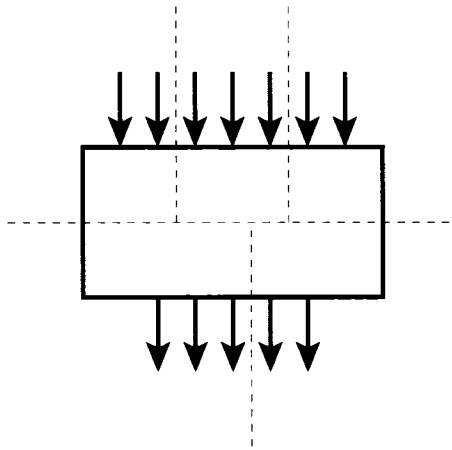


Figure 3.1: Node Clusters.

A "section" is a connected region of an AHAHA graph which includes a set of producing clusters, a set of consuming clusters and all of the connected unclocked nodes which lie between the producers and consumers. No edges in the graph may cross a boundary between sections. This constraint implies that all values consumed within a section must also originate in that section, and vice versa. A cluster may only participate in a single section. Sections are constructed using a connected components algorithm, where the "connected" relationship is symmetric. Ports of a clocked node are connected to each other if they lie within the same cluster, ports of unclocked nodes are connected, and

an edge in the graph connects its source and target ports. An example of a sectioned AHAHA graph is shown in Figure 3.2. The red dotted lines indicate section boundaries (and, consequently, cluster boundaries). Clocked node are represented as black rectangles, and unclocked nodes as smaller gray rounded rectangles. Sections are created

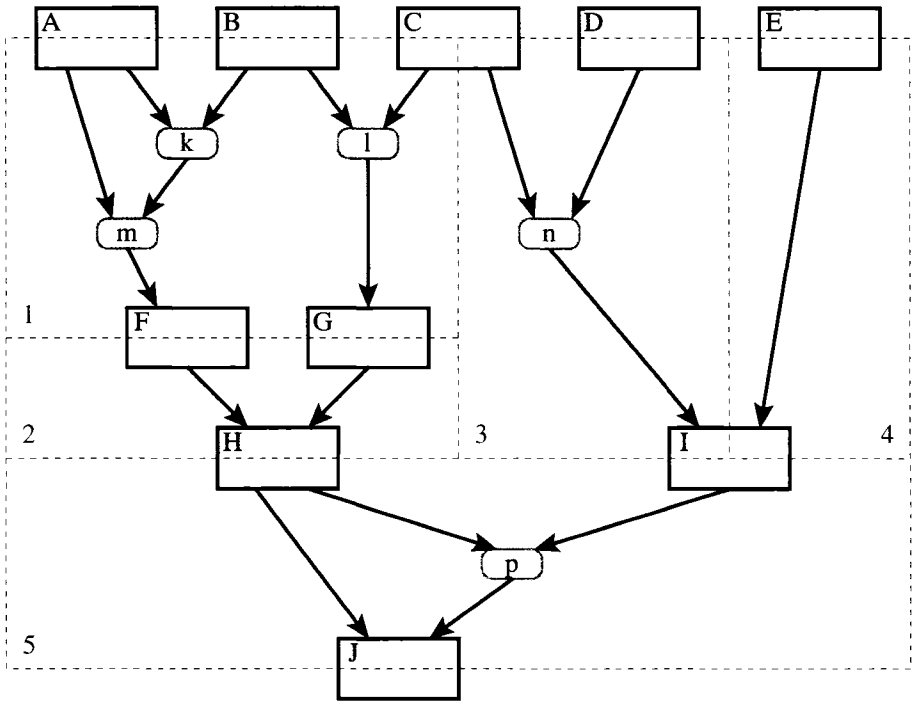


Figure 3.2: Sections.

beginning at any cluster in the graph which is not currently part of a section. A new section is created to contain that cluster, and the connected components algorithm is used to add unclocked nodes and other clusters to the section. This process is repeated until all clusters in the graph belong to a section. For example, section 1 of Figure 3.2 could have been created by making a new section initially containing the first (and only) output cluster for node A. The outputs of this cluster are connected to unclocked nodes k and m, which are added to the section. Node m is connected to the only input cluster of node F. Likewise node k is connected to the only output cluster of node B, which is in turn connected to node l. Node l is connected to the first output section of node C, and the input cluster of node G. Conceptually, a section contains clusters (and unclocked

nodes) which must fire concurrently in order to correctly execute the graph. Each section must be able to determine whether or not to fire on its own, without information from any other sections. No 2 input clusters of a node may be included in the same section, and the same is true for output clusters. However, a section may (in some restricted cases) include an input cluster and an output cluster from the same node. Complex sectioned AHAHA graphs are not necessarily planar.

"Handshaking" is used to synchronize the nodes, and implement the firing rules of the AHAHA. Each producing cluster of a node provides a signal to indicate that the outputs in that cluster are valid and usable. Likewise, each consuming cluster of a node provides a value which indicates that its inputs are ready to accept a value. When all of the producing clusters of a section are valid, then it is guaranteed that all unclocked nodes within that section are operating on valid data, and all consuming clusters are receiving valid data. When all of the consuming clusters in a section are being provided valid data, and they are ready to receive it, then all of the clusters contained within that section may fire simultaneously. In order for handshaking to work, each cluster must be given a signal which indicates that all of the other participating clusters in its section are ready.

The restriction of "no hard-coded information or redundancy" requires that there be a single location where all of the information about each node resides. Since the VHDL implementation of a node must reside in a `.vhd` file, it follows that the other information about the node should reside there as well. However, in some cases VHDL nodes are parameterized using generics, so the supplemental information in the `.vhd` file must be able to be parameterized as well. To this end, a preamble is used in the beginning of each `.vhd` file, written in a simple interpreted language. This region of the `.vhd` file is used to manipulate a symbol table containing variables which define information about the node being implemented. These symbol tables are implemented using hash tables, and are accessible to the compiler so that it may query properties associated with nodes. The compiler evaluates the preamble, and parses the VHDL entity declaration as it is

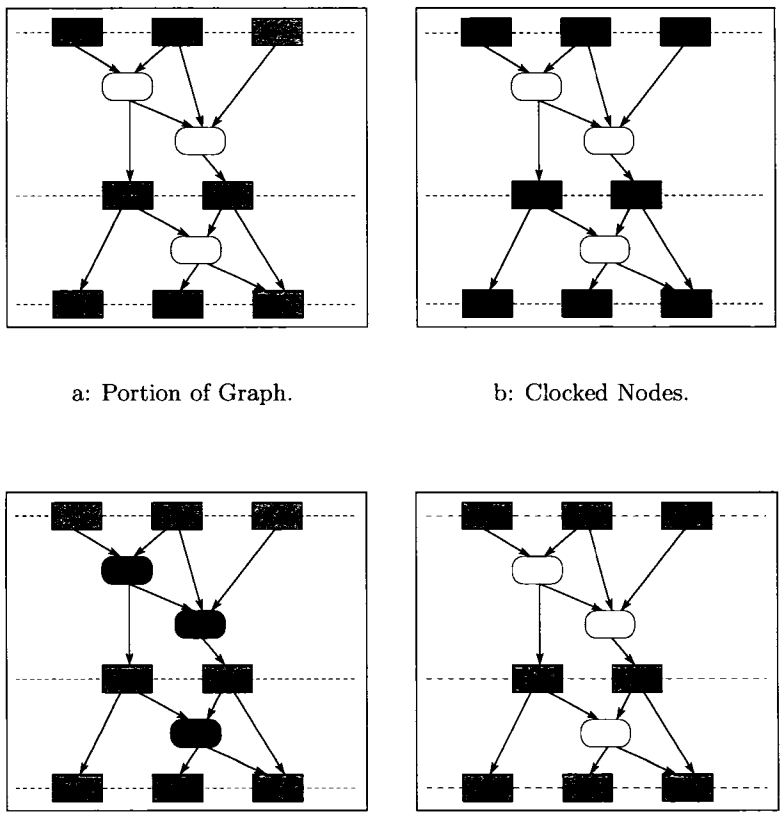
needed. Together, the preamble and entity provide all information specific to the node(s) implemented in the `.vhd` file. When an existing node is modified, or a new node type is added to the AHAHA, only the file defining it needs to be modified or created; no compiler source code needs to be touched.

3.2 AHAHA Graph Structure

The motivation and design decisions above, form the basis of the AHAHA design, described in this section. The relative placement of the 2 main types of nodes in an AHAHA graph defines its behavior. Figure 3.3 shows a portion of an example AHAHA graph, and identifies its parts. Nodes with internal state are called "clocked" nodes (Figure 3.3 b) because they require a clock signal for the registers they contain. Some examples of nodes of this type include the `READ_WORD_MEMORY` and `WRITE_WORD_MEMORY` nodes which provide access to external memories on the RCS board. The remaining nodes in the graph which have no internal state (and therefore, no registers, or clock signal) are called "unclocked" nodes (Figure 3.3 c). Examples of this type of node include the simple arithmetic nodes, such as add/subtract, and left/right shift. There is a third type of node called "semi-clocked," which is much less common. They are nodes which do require a clock, but do not contribute to the handshaking.

The AHAHA graph is divided into discrete sections. Each clocked node lies on boundaries between sections, they consume data from the section(s) above, and produce data in the section(s) below. Unclocked (and semiclocked) nodes lie entirely within a section. Edges do not cross section boundaries. Figure 3.3 d shows this structure. The data flows downward through the graph from section to section.

Cyclic graphs are handled as a special case. Because the SA-C* language contains the idea of a "nextified variable" (which is a variable whose value is initially defined before a loop, and given a new value at every loop iteration, which may be based on the previous), cyclic graphs do occur. In these cases, a special node called a `CIRCULATE` node, is created. Each `CIRCULATE` node contains 4 clusters (2 input, and 2 output), and



a: Portion of Graph.

b: Clocked Nodes.

c: Unclocked Nodes.

d: Section Boundaries.

Figure 3.3: AHAHA Graph Structure.

therefore participates in 4 sections. It receives its initial value in one section, receives the updated value for the next iteration in another, produces its output for each iteration in a third, and produces the final value in a fourth. These nodes are different from others in that all their outputs are not lower in the graph than all of their inputs. In compiler theory, these edges which target a dominator (or predecessor) are called back-edges. The logical layout of a `CIRCULATE` node is illustrated in Figure 3.4, but its physical layout is no different than any other node (inputs on top, outputs on bottom).

Clocked nodes that lie on the top boundary of each section are the source for all the data in that section, while the ones on the bottom boundary are the sinks. Each clocked node is a source in (at least) 1 section and a sink in the preceding section(s).

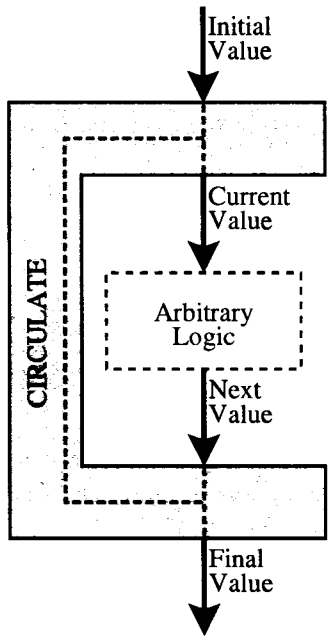


Figure 3.4: Logical layout of a CIRCULATE node.

The inputs of each clocked node are grouped into clusters. The ports in each input cluster receive data from the same section. Likewise, output ports are grouped into output clusters. All ports of an output cluster produce data in the same section. The majority of the clocked nodes have a single input cluster, and a single output cluster, but there are several nodes with more. For example, the `CIRCULATE` nodes have 2 input clusters and 2 output clusters, which allows them to participate in up to 4 sections.

Each clocked node has a certain latency, which is the minimum time required between when tokens arrive at one of its input clusters, and the time when any tokens generated by it are available in the output clusters. When multiple paths through the graph converge at a single node, the paths must be "buffer balanced." This makes the resulting graph more efficient by reducing dead cycles, and permitting pipelining. For example, consider the graph portion shown in the left side of Figure 3.5. The numeric label on each node describes its latency. For this example, assume that all nodes are capable of pipelined operation; they behave like FIFOs which can hold a number of elements equivalent to their latency in steady state, and one more than their latency when stalling. The left

Section 1		•• •• •• •• •• ••
A	Out	••
B	In	••
E	In	•• •• •• •• •• ••

Section 2		•• •• •• •• •• ••
B	Out	•• •• •• •• •• ••
C	In	••

Section 3		•• •• •• •• •• ••
C	Out	•• •• •• •• •• ••
D	In	••

Section 4		•• •• •• •• •• •
D	Out	•• •• •• •• •• •
E	Out	•••
F	In	••

Table 3.1: Firing pattern of the left graph in Figure 3.5.

Section 1		••
A	Out	••
B	In	••
E	In	••

Section 2		•••
B	Out	•••
C	In	••

Section 3		•••
C	Out	•••
D	In	••

Section 4		•••
D	Out	•••
G	Out	••
F	In	••

Section 5		•••
E	Out	•••
G	In	••

Table 3.2: Firing pattern of the right graph in Figure 3.5.

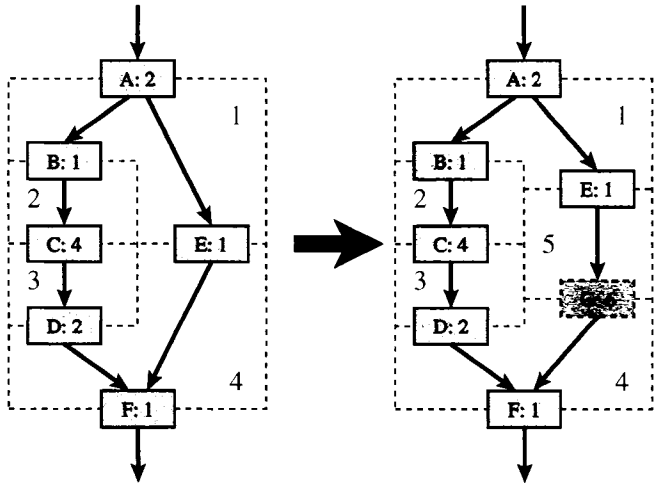


Figure 3.5: Example of buffer balancing.

path from A to F has a total combined latency of 9 (2+1+4+2), but the right path has only 3. Because these values are different, the flow through this segment of the graph will be restricted. The firing pattern of the graph is shown in Table 3.1. A black dot indicates that a cluster is willing to fire, while a red dot indicates that the section (including all of its clusters) is firing. Notice that the pattern becomes periodic after the first few clock cycles. The period is 8 clock cycles and any given section is only firing for 2 of the 8. The 6 wasted clock cycles are a result of the imbalance of the accumulated latencies at node F. Because section 4 cannot fire until the output cluster of node D is ready (in clock cycle 8), and node E can only hold 2 tokens, the subgraph is inefficient, running at only $\frac{1}{4}$ of its maximum rate.

In order to achieve maximal throughput, adjustments must be made so that node E can continue to accept tokens while nodes C and D are executing their pipelines. The solution is to add an extra node, G, which is able to buffer at least 6 tokens in steady state, and has a latency no greater than 6. Node G is a `BUFFERX` node, discussed in Section A.2.2. This particular instance of a `BUFFERX` node has a latency of 1, and can hold up to 15 tokens in the steady state, and 16 while stalling, which is adequate for this task. Node E may now deliver its tokens to node G, allowing it to continue to accept

tokens from node A uninterrupted. The addition of node G achieves a higher overall throughput, as shown in Table 3.2.

In this example, the throughput increased from $\frac{1}{4}$ to 1. In most cases where pipelining is exploitable, it is possible to achieve a continuous firing pattern using buffer balancing. This is especially important in the innermost loops of the graph, where any inefficiency is compounded by the number of loop iterations.

3.3 Implicit Structures

In addition to the nodes, AHAHA graphs include several implicit structures. Handshaking logic, memory arbitrators, BlockRAMs and the host interface, are all implicit in AHAHA graphs, and so are not depicted when they are displayed.

3.3.1 Handshaking

AHA nodes communicate with a simple handshaking protocol. The purpose of handshaking is to identify when all of the clocked nodes bordering a section are ready to execute. This implements the firing rules of the AHA graph and ensures that no information is lost, or gets out of sync. Strictly speaking, handshaking is not *required* for all designs generated by the SA-C* compiler; many are simple enough that they could hypothetically be statically timed with a finite state machine to control the firing pattern of each section. In a statically timed model, each node begins firing at a fixed point in the execution and continues at a fixed frequency. Static timing is based on worst case analysis of the program, and often wastes clock cycles as compared with a dynamic handshaking solution. Additionally, there are situations where the timing is data dependent and, therefore, precludes an optimally efficient static solution. For example, the amount of data produced by compression algorithms (such as run-length encoding) is data dependent, and the `WRITE_WORD_MEMORY` nodes do not have a compile-time predictable firing pattern. Likewise decompression algorithms (such as run-length decoding) make the firing pattern of the `READ_WORD_MEMORY` nodes data dependent. Conditional reads and writes (based on runtime data) in the AHAHA graph may create situations where many

requests are made to the same memory simultaneously, and the memory arbitrators will serialize them in an unpredictable manner. Handshaking allows the graphs to time themselves optimally, and eliminates the need for complex analysis which may produce suboptimal static solutions, increasing runtime. The handshaking protocol is actually quite simple. So simple, in fact, that it is doubtful that switching to a static timing model would have any positive effect on FPGA utilization, or clock frequency. Analysis of the overhead required for the handshaking is discussed in Section 5.1, page 112. Due to its relatively small cost, handshaking is used exclusively.

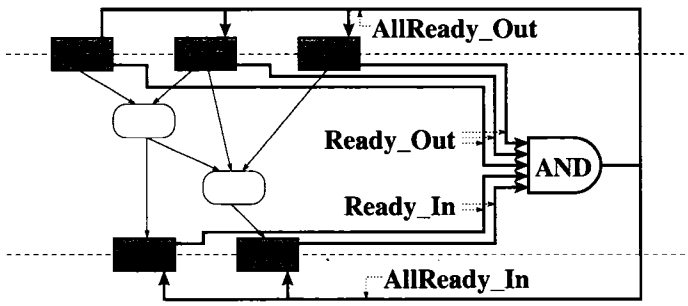


Figure 3.6: Example Handshaking Logic.

An example of the handshaking logic present in each section is shown in Figure 3.6. Clocked nodes (on section boundaries) are responsible for the synchronization of the sections. The unclocked nodes represent only combinational logic. To control when the data is consumed and produced, all clocked nodes have 4 handshaking signals: **Ready_In**, **Ready_Out**, **AllReady_In**, and **AllReady_Out**. The **_In** signals are used to synchronize inputs with the section(s) above (from the perspective of the node), while the **_Out** signals are used in the section(s) below. The **Ready_** signals are outputs which indicate that the node is ready to produce or consume. The **AllReady_** signals are inputs to the node indicating that all nodes in the section are ready, and the data is currently valid. If a **Ready_** signal of a node is low, then the node is guaranteed that the corresponding **AllReady_** signal is also low. These signals have a similar purpose as FIFO control signals. In fact, it may be convenient to think of each edge entering or exiting a clocked node as containing a small FIFO.

Semiclocked nodes have a single input called **AllReady** which is used to identify when the section containing the node fires. Other than this signal, semiclocked nodes do not contribute to the handshaking signals; in this way they are like unclocked nodes.

All of the handshaking signals described here are implemented using the VHDL type `std_logic_vector` (which is simply a vector of bits), so that they can participate in multiple clusters. Bit n of each vector corresponds to the handshaking signal in cluster n.

Handshaking logic is created for each section independently, and consists of nothing more than 1 AND gate. All of the **Ready_Out** signals of the clocked nodes which produce data and all of the **Ready_In** signals of the clocked nodes which consume data are inputs for the AND gate. The output of the AND gate is attached to the **AllReady_Out** ports of the producing nodes, all of the **AllReady_In** ports of the consuming nodes, and to the **AllReady** ports of any semiclocked nodes.

3.3.2 Memory Arbitrators

Every memory on the board which is utilized by the application is attached to a dedicated memory arbitrator which is in turn attached to memory access nodes in the design (ie, `READ_WORD_MEMORY` or `WRITE_WORD_MEMORY`), and the host interface. The memory arbitrator allows multiple nodes in the design (and the host) to be connected to the same memory. Only 1 access per clock cycle is actually performed by each memory, so the arbitrator uses a simple priority scheme to decide which memory access is serviced in each clock cycle. Each node which needs to access the memory makes a request to the arbitrator. The request with the highest priority is acknowledged, and the rest are denied. All of the denied clients must wait for a clock cycle and try to access the memory again in the next cycle. The acknowledged request is sent to the memory to be serviced. The host is given the highest priority, because after a host transfer has begun it is not possible to stall it. By giving the host the highest priority, communication between the memories and the host is simplified, since the transfer will never be denied or interrupted. Nodes which read are next in priority, followed by nodes which write. In situations where

the application is memory-bandwidth limited, this scheme allows data to execute in a more eager fashion.

3.3.3 BlockRAMs

In some cases, array values are stored in BlockRAMs instead of on board memories. BlockRAMs are used when the user requests them in the SA-C* code. To construct BlockRAMs in the AHAHA layer, pragmas are attached to the top-level AHAHA wrapper node which indicate their requested geometry (width, depth etc.). When required, several physical BlockRAMs can be composed to create a larger logical BlockRAM. One can create logical BlockRams wider than the widest available physical BlockRAM by dividing each value to be stored into segments which are small enough to store. For example, 32-bit values can be stored in four 8-bit wide BlockRAMs without any need for additional supporting logic. Likewise, one can create logical BlockRAMs deeper than the deepest available physical BlockRam by separating the memory into smaller address ranges. The high address bits are used to select which BlockRAM will be accessed. Concatenating BlockRAMs in this manner requires at least 1 decoder on the address pins, at least 1 register to delay the selection information, and a MUX (or combination of OR & AND gates) to select the correct value being returned. Because this method requires some additional logic (even if it is trivially small), the previous method of combining BlockRAMs in parallel is always preferred, and used first by the AHAHA to VHDL translation. The constituent BlockRAM geometry, is always selected to be as deep as is needed before considering the width. Partitioning the address range is only needed when the requested BlockRAM is deeper than the deepest available configuration on the chip. However, these methods are orthogonal, and can be employed independently in order to create BlockRAMs that are deeper and wider than the physical limits of a single BlockRAM. In addition to storing arrays, BlockRAMs are used to store queued elements in the implementation of large streams (discussed in Section A.1.3).

3.3.4 Host Interface

The host interface is a conduit between the host and the RCS which allows them to communicate in several ways.

- The host can read or write arrays in the board memories via the appropriate memory arbitrator.
- The host may write to input streams (streams which are produced by the host, and utilized in the hardware).
- The host may read from output streams (streams which are produced in the hardware, and utilized by the host).
- The host can store small runtime constants (such as memory addresses or constants used in computations) directly into a register file which supplies the initial data into the top of the AHAHA graph.
- The host can instruct the FPGA to begin computing, or halt the FPGA execution.
- The FPGA can use a hardware interrupt to alert the host to a variety of conditions, such as when it has finished executing, when the input streams need filling, or when the output streams need emptying.

The host interface is not a monolithic VHDL component. Rather, it is created out of several components which are selected and tailored to the needs of each program. The host interface is board-specific and must be synchronized with the RTS for that board. The following describes the host interface for the Alpha Data ADM-XRC-II board, but the host interfaces for other boards are quite similar. Available components include:

- **Local Bus Controller**: Allows the host to access on-chip components by providing a bridge between the off-chip bus (such as the PCI bus) and an on-chip bus of host-accessible components. All components are memory mapped on the on-chip bus, and are placed in predefined memory locations so that the host (or more

specifically, the RTS) knows where to locate them. Figure 3.7 shows the typical memory map used in the Alpha Data ADM-XRC-II RTS.

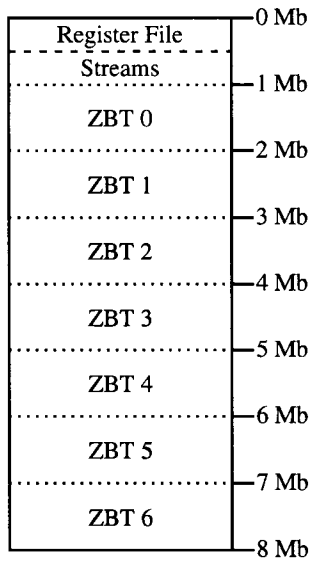


Figure 3.7: Memory Map for the Alpha Data ADM-XRC-II RTS.

- **Register File**: Attaches to the on-chip bus and provides a register file to the AHAHA graph. The primary use for the register file is to provide values used in INPUT nodes, with a few notable exceptions:
  - **Input element 0** is used as a 1-bit signal, and is provided to the foreman as "Status_In" (discussed below) to control the graph execution.
  - **Output element 0** is a 32-bit word which reports the hardware status, divided into several smaller parts:
    * **Bits 31 down to 29: DCM Status**: Digital Clock Managers (DCMs) are used to remove clock skew and jitter from a clock domain, and require a short time after the chip is configured to "Lock" onto the source clock signal. The first DCM (Bit 31) synchronizes the FPGA with the clock supplied by the off-chip bus, so that communication between the host and the FPGA is reliable. The other 2 DCMs synchronize the clocks of

the off-chip memories with the clock signal used in the FPGA; the first memory DCM (Bit 30) controls the clock signal of ZBT banks 0, 1, and 2; the other DCM (Bit 29) controls the clock signal of ZBT banks 3, 4, and 5. These bits indicate whether or not the corresponding DCM is locked (1 indicates it is locked, 0 indicates it is not).

* **Bit 28: Reset**: This bit contains the reset signal from the foreman. A value of 1 indicates that the foreman is holding the AHAHA graph in a reset state, 0 indicates that the graph is free to execute.
* **Bit 27: Interrupt Request Status**: The status of the interrupt line. A value of 0 indicates that the graph has sent an interrupt to the hardware, while 1 indicates that no interrupt has been requested. This is because the interrupt line on the Alpha Data ADM-XRC-II is triggered by a falling edge. This value can be used by the RTS to squelch false interrupts which may be a result of the chip being reconfigured.
* **Bits 23 down to 8**: These bits contain the high 16 bits of the 48-bit clock-cycle count, (see Output element 1 below).
* **Bits 1 and 0: Foreman Status**: These bits contain the status bits of the foreman, which indicate the current status of the graph.

The remaining bits are all undefined, and reserved for future use, or for use on other RTS boards.

- **Output element 1** provides the low 32 bits of the clock-cycle count to the host from the foreman. Since this value is often larger than 32 bits,[2] 48 bits are used.[3] The high 16 bits are stored in a portion of the previous output element.

---

[2]When operating at 66 MHz, 2^{32} clock cycles correspond to only 65 seconds of execution time.

[3]When operating at 66 MHz, 2^{48} clock cycles correspond to nearly 50 days of execution time.

The Register file is always a power of 2 in size. The smallest power of 2 which provides the necessary registers is automatically selected. Registers in the file which are not used are dissolved by the synthesis tools, so do not waste any chip area.

- **Foreman**: The job of the foreman is to coordinate the graph execution. It provides a signal "AHAHAReset" to the rest of the graph, which is used as a reset signal for all the clocked nodes in the graph. It begins in a stalling state with AHAHAReset asserted, and remains there as long as its "Status_In" signal is 0. When Status_In becomes 1, it lowers the AHAHAReset line and the graph begins to execute. As the graph executes, the foreman counts the number of clock cycles which pass. When the graph signals that it is finished (via the OUTPUT node), the foreman reasserts the AHAHAReset line, stops incrementing the clock cycle counter, and initiates an interrupt to the host to signal the completion of the graph execution. The foreman provides a "Status_Out" signal to the host via the register file, which indicates the current status of the graph. "00" indicates that the graph has not begun execution yet, "01" indicates that the graph is currently running, and "10" indicates that the graph has finished executing.

  In addition to the foreman, a "trigger" node is created for every section which does not have any producers. The job of the trigger node is to participate in the handshaking as if it were a producer, but does not create any values. After the AHAHAReset signal is deasserted, it will enter a state in which it is willing to fire. After firing, it enters a trap state in which it is unwilling to fire. The end result is that sections which have no producers will fire only once as soon as the consumers of the section are ready.

- **ZBT Memory Bridges**: Only ZBT memories which are used in the computation have a corresponding bridge. The bridge allows burst transfers of data between the host and the ZBT memories via the corresponding memory arbitrator. The bridge is always given the highest priority on the arbitrator (priority 0).

The ZBT bridge supports burst reads and writes. Burst reads from the hardware to the host are pipelined, meaning that values are being requested ahead of time in order to maintain a full speed burst transfer. Burst writes are not pipelined; the data arrives in lock step with the address being written. The ZBT memories on the Alpha Data ADM-XRC-II board have a latency of 4 clock cycles, so when reading, the bridge makes requests 4 clock cycles early. This leads to an overrun at the end of the read. In the case of the ZBT bridges, the overrun is not an issue because reading from a memory does not cause any side effecting; the 4 extra values read from the memory at the end of the transfer are simply ignored by the bridge, and not sent to the host.

- **BlockRAM Memory Bridges**: For BlockRams which require host access, a BlockRAM Bridge is created which attaches one port of the BlockRAM to the on-chip bus. Because BlockRams are dual ported, and only 2 clients may attach to any one BlockRAM (including the host), no arbitrator is needed. BlockRAM bridges also support burst transfers. The BlockRAMs have a single clock cycle of latency, so take advantage of the read pipelining to create sustainable block transfers. Like ZBT memories, the read pipelining is not an issue because reading from a BlockRAM does not cause any side effecting.

- **Stream Bridges**: A stream bridge is created for each stream in the design which requires host participation. (See Sections A.1.3 and A.1.5). The bridge allows the host to query the stream status (ie, how much space is available, how many elements are present, whether the stream is still open or not), read or write blocks of data from/to the stream, and close the stream. The stream bridge is also responsible for sending a hardware interrupt to the host when the stream needs to be refilled, or emptied. By default, the stream signals the host when an input stream has

space remaining and is not closed, or when an output stream is not empty, or has become closed.[4]

Stream bridges support burst transfers, but since it is impossible to use a read pipeline of 0 clock cycles on the Alpha Data ADM-XRC-II board, the pipeline causes requests to occur 1 clock cycle ahead; there is a fetch occurring for the next value as the current value is written to the local bus. This causes one more value to be read from the stream than the host actually wants to read. This causes 2 problems when the host reads the stream, since reading from a stream side effects it (as opposed to reading from a memory). First, there is a possibility that the host will attempt to read an additional value after the stream is empty. This is not supported by the stream, and will put it into an invalid state. Reads on an empty stream are therefore suppressed. Second, there is a possibility of a transfer over the local bus being terminated in such a way that the last value of the burst transfer is valid, but not received by the host. To solve these problems, the stream bridge also provides 1 value worth of buffer (with a single presence bit) to hold the extra value. This mechanism is called a "runaway truck ramp." It allows the data to be transferred at full speed (1 value per clock cycle), but when the transfer is suddenly terminated, the value which was in flight from the stream to the local bus is redirected into a holding cell, where it will be used to begin the next read. This ensures that no values are lost when performing block reads from the stream. Values which result from a suppressed read (when the FIFO was empty) are not stored. The presence bit is taken into account when reporting the stream state.

Each stream bridge requires 4 consecutive addresses in the memory map. During read or write operations to/from stream bridges, the addresses do not increment

---

[4]This behavior is controlled with a `#define` called "GREEDY_INTERRUPTS" at the top of ahaha_to_vhd.c. The default is 1 which causes the described behavior. If changed to 0, the interrupt will not be sent until the stream is causing the graph to stall (an input stream is empty, or an output stream is full).

as they do with other components. The type of transfer is indicated by the low 2 bits of the read/write addresses, and are mapped as follows:

- **"00" Values**: When writing to this address, values are enqueued in the stream. When reading, values are dequeued from the stream.
- **"01" Count**: Reading from this address returns the number of elements of the stream that are present in the buffer, and available for reading. Writing to this address is unsupported.
- **"10" Space**: Reading from this address returns the number of empty cells available in the buffer for writing. Writing to this address is unsupported.
- **"11" Close**: Reading from this address returns 1 if the stream is closed, and 0 if it is open. Writing a value with a low bit of 1 to this address closes the stream, if the low bit is 0 the write is ignored.

Stream bridges are parameterized so that they only support the required operations for that stream. Input streams (Section A.1.3) only support writing to "values," reading from "space," and writing to "close." Output streams (Section A.1.5) only support reading from "values," reading from "count," and reading from "close."

3.4 AHAHA Nodes

There are 71 nodes available in the AHAHA model, divided into 4 categories: special (5 nodes), clocked (31 nodes), unclocked (34 nodes) and semiclocked (1 node). Each node is described in detail in Appendix A. The nodes are implemented in a collection of VHDL files. Each node may have more than one *style* (listed in Table 3.3). Each style of a node is a different method of implementing the behavior of that node. In the case of clocked nodes, these styles tend to differ in their timing behavior, and steady-state throughput. The multiple styles of unclocked nodes simply provide alternate implementations which may vary in chip utilization area. Each VHDL description is stored in a separate file. (43 "`.vhd`" files contain 127 node descriptions in all). The mapping from node to VHDL

description is many-to-many; the .vhd files identify which node(s) they implement, and under which circumstances they are usable.

Style	Wgt	Description
`orig`	1	Implementation carried over from AHA to AHAHA. None of these styles are actually used due to the very low weight. They are only included for historic interest.
`unclocked`	10	Used for unclocked nodes with only one available style. Such nodes are usually extremely simple.
`unclocked1`	10	Used when multiple unclocked styles are needed. Needed when there are multiple ways to implement a node and it is not clear which implementation will use the least chip area.
`unclocked2`		
`unclocked3`		
`unclocked4`		
`clocked`	10	Used for clocked nodes with only one available style. No internal buffering is used; these implementations may occasionally waste a clock cycle.
`clocked1`	10	Used when multiple clocked styles are needed.
`clocked2`		
`clocked3`		
`clocked4`		
`buffered`	20	Used for clocked nodes with only one available style. Internal buffering is used to increase performance, at a small cost of chip area. Buffered styles are always preferred over clocked styles.
`buffered1`	20	Used when multiple buffered styles are needed.
`buffered2`		
`buffered3`		
`buffered4`		
`semiclocked`	10	The only style available for semiclocked nodes.

Table 3.3: Description of available node styles.

Some nodes contain styles which are capable of implementing the node only under specific circumstances. For example, the `BUFFERX` node has a style which is implemented using a double buffer, which is only valid when the depth of the `BUFFERX` node being requested is 1. When a larger buffer is required, this style cannot be used, and one of the more general case styles is used instead. Each time an implementation of an AHAHA node is requested by the compiler, all VHDL files corresponding to that node are examined to determine whether or not the VHDL file is capable of implementing it, and if so, what its approximate utilization is. The AHAHA node being implemented is

used to provide context during parsing. The translator decides which of the available node styles will be used in the final design based on 2 factors. First, every style has an associated weight.[5] Nodes with a higher weight are preferred. These weights are primarily used to test new node implementations, and to allow certain implementations to be preferred over others. Next each implementation is synthesized (using the Xilinx synthesis tool XST[33]) to determine its area utilization in terms of flip-flops, lookup tables, BlockRAMs, and embedded multipliers (see the `usage_*` variables in Table 3.6). Each of these are multiplied by a weighting factor (supplied by the board description file, which is described in Section 3.5, and the `weight_*` variables in Table 3.9), and summed to produce an overall utilization (see `usage_overall` in Table 3.6). Smaller overall utilization is preferred. The available styles and their associated weights are listed in Table 3.3

3.4.1 VHDL Preamble

To simplify AHAHA to VHDL translation, and to make adding new nodes simpler, each .vhd file contains a preamble section which is embedded in VHDL comments, so that its contents will be ignored by VHDL synthesis and simulation tools. Figure 3.8 shows the preamble for the buffered2 style of the BUFFERX node. The preamble allows a VHDL description to define its context in terms of the AHAHA node being implemented, make assertions, do simple math, and pass information back to the AHAHA to VHDL translator, or to the generics and ports in the VHDL entity. The preamble has been implemented using a small special-purpose language, but an existing language, such as perl, could have been used in its place. Using a general language for such a simple and specific task may be overkill, however, inventing a custom syntax for this purpose also has some negligible learning overhead. All information about the structure and behavior of a node must be described in its preamble. For example, the preamble contains physical

[5]The style weights themselves are stored in a file called **node_weights** in the **data** directory of the AHAHA to VHDL translator.

information about a node, such as the number of clusters and ports, so that it can be correctly integrated into the AHAHA section model, and participate in the handshaking correctly. Behavioral information about the node is also defined in the preamble. For instance, the preamble defines the steady-state latency of the node, which is used in buffer balancing to ensure for maximal performance. A list of all the defined preamble variables can be found in Figure 3.6, where mandatory variables which must be supplied by the user are indicated with a † symbol. By localizing all of the information regarding a particular node into a single file, the process of adding new nodes is simplified, and the issues involved with keeping multiple files in agreement is avoided. The preamble also allows multiple implementations (or "styles") of a node to be created without modifying the structure of the compiler. When a newer implementation of a node is derived, it can be simply added to the library of nodes, and it will be automatically utilized by the compiler. The remainder of section 3.4.1 contains low level implementation details of the VHDL preamble.

The preamble is divided into 2 pieces. The first (header) portion simply defines which nodes and styles are implemented by the VHDL description. This section contains one or more lines of the form:

```
--NODE: NODENAME style1 style2 style3
```

The first style listed (`style1` in the above example) is the actual name of the style, and the rest are aliases for the same style. Aliases are only to allow multiple names for the same style so that certain sets of styles can be referenced in the `node_weights` file.

The second (body) portion of the preamble is much more substantial and contains commands which are executed when the translator parses a VHDL file. Each line in this section begins with a tag: "`-->`" which is interpreted as a comment by the VHDL synthesis tools. There are 3 types of statements in this region: assignments, assertions and if statements.

Following the preamble, is either a VHDL description (consisting of `library`, `use`, `entity`, and `architecture` statements, in that order), or a special preamble command:

```
--NODE: BUFFERX buffered2

--> $clktype = "Clocked";
--> $latency = $DEPTH;

--> $NUMSRL = ($DEPTH + 15)/16;
--> $REALDEPTH = $NUMSRL * 16;
--> $ABITS = bits($REALDEPTH - 1);
--> $CBITS = bits($REALDEPTH);

--> $min_latency = 1;
--> $max_latency = $REALDEPTH;
--> $num_inputs == 1;
--> $num_outputs == 1;
--> $num_in_clusters == 1;
--> $num_out_clusters == 1;

--> $pt_data_in = 0;
--> $pt_data_out = 0;

--> $WIDTH = sizeof(in,$pt_data_in);
--> sizeof(out,$pt_data_out) == $WIDTH;

--> $Data_In = [$pt_data_in];
--> $Data_Out = [$pt_data_out];

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.ahaha_lib_pkg.all;

entity BUFFERX_buffered2 is

  generic (
    ABITS     : natural := 6;
    CBITS     : natural := 7;
    NUMSRL    : natural := 3;
    REALDEPTH : natural := 48;
    WIDTH     : natural := 8);

  port (
    Clock        : in  std_logic;
    Reset        : in  std_logic;
    Ready_In     : out std_logic_vector(0 downto 0);
    AllReady_In  : in  std_logic_vector(0 downto 0);
    Data_In      : in  std_logic_vector(WIDTH-1 downto 0);
    Ready_Out    : out std_logic_vector(0 downto 0);
    AllReady_Out : in  std_logic_vector(0 downto 0);
    Data_Out     : out std_logic_vector(WIDTH-1 downto 0));

end BUFFERX_buffered2;
```

Figure 3.8: Preamble and Entity declaration for buffered2 style of BUFFERX node.

SPECIAL_CASE_IMPLEMENTATION followed by a string. The latter is used to indicate that the node is not implemented using a fixed VHDL file, but rather is implemented using special case C code, which generates the VHDL implementation for the node dynamically. The C code uses information about the node which is gathered by the preamble and stored into a series of variables in the symbol table of the node.

The generated VHDL code is inserted directly into the top-level architecture. The string indicates which special case to use for this node.

For example, the line:

```
--> SPECIAL_CASE_IMPLEMENTATION "SELECTOR";
```

declares that this node is implemented using the special case code reserved for selector nodes. In this example, the special case code can be found in the files `ahaha_special_selector.c` and `ahaha_special_selector.h`. This method of implementing a node is reserved for nodes whose VHDL is context dependent and, therefore, needs to be dynamically generated, as described above. Currently, only 2 nodes are implemented using special cases: `SELECTOR` nodes, and `ROMREF` nodes. `SELECTOR` nodes are basically a priority MUX, and can be implemented using a single VHDL `when` statement. However, the number of clauses present in the `when` statement depends upon the specific `SELECTOR` being generated. This is difficult to implement using a VHDL entity, but very simple to generate appropriate VHDL from C. `ROMREF` nodes require the declaration of a constant ROM table before the begin statement of the VHDL architecture. The contents of the table depend on the `ROMREF` being created, and so cannot be implemented statically in a `.vhd` file. The table is easy to generate from C, however, and so is handled in a special case. Once the table is generated, the node itself becomes a single-assignment statement which indexes the appropriate row of the table.

In order for the entity declaration in the VHDL file to be compatible with the AHAHA framework, it must use only `Natural`, `Integer`, `String`, `std_logic`, and `std_logic_vector` types for generics, and only `std_logic`, and `std_logic_vector` types for ports in the entity. Ports which correspond to ports of the AHAHA node, must

use `std_logic_vector`, even if they are only a single bit, in which case they should be declared as "`std_logic_vector(0 downto 0)`."

All variables in the preamble are lexically scoped, and are prefixed with a `$` to identify them as variables, and may contain only alphanumeric characters and underscores (but may not begin with an underscore). Statements may include any of the standard infix operators described in Table 3.4.

Op	Assoc.	Types		Description
		Result	Args	
|	Left	Integer	Integer	**Logical OR**: Returns 1 if either operand is nonzero, and 0 otherwise.
&	Left	Integer	Integer	**Logical AND**: Returns 1 if both operands are nonzero, and 0 otherwise.
<	Left	Integer	Numeric	**Arithmetic Less Than**: Returns 1 if the first operand is less than the second.
>	Left	Integer	Numeric	**Arithmetic Greater Than**: Returns 1 if the first operand is greater than the second.
==	Left	Integer	Numeric	**Arithmetic Equal to**: Returns 1 if the two operands are equal, 0 otherwise.*
			String	**String Equal to**: Returns 1 if both strings contain identical text, and 0 otherwise. The comparison is case-sensitive.
<=	Left	Integer	Numeric	**Arithmetic Less or Equal**: Returns 1 if the first operand is less than or equal to the second, 0 otherwise.*
>=	Left	Integer	Numeric	**Arithmetic Greater or Equal**: Returns 1 if the first operand is greater than or equal to the second, 0 otherwise.*
!=	Left	Integer	Numeric	**Arithmetic Not Equal to**: Returns 0 if the two operands are equal, 1 otherwise.*
			String	**String Not Equal to**: Returns 0 if both strings contain identical text, and 1 otherwise. The comparison is case-sensitive.
+	Left	Numeric	Numeric	**Arithmetic Addition**:Adds the operands and returns the result.
-	Left	Numeric	Numeric	**Arithmetic Subtraction**: Subtracts the second operand from the first and returns the result.
.	Left	String	String	**String Concatenation**: Constructs a new string which contains the text of the left operand immediately followed by the text of the second.

Continued on next page

Continued from previous page				
Op	Assoc.	Types		Description
		Result	Args	
*	Left	Numeric	Numeric	**Arithmetic Multiplication**: Multiplies the operands and returns the result.
/	Left	Numeric	Numeric	**Arithmetic Division**: Divides the first operand by the second and returns the result.
^	Right	Numeric	Numeric	**Arithmetic Exponentiation**: Returns the first operand raised to the second operand.
**				
-	Right	Numeric	Numeric	**Arithmetic Negation**: Returns the negated value of the operand.
!	Right	Integer	Integer	**Logical Not**: Returns 1 if the operand is 0, and 1 otherwise.
float	Right	Float	Numeric	**Cast to Float**: Typecasts the operand into a float.
int	Right	Integer	Numeric	**Cast to Int**: Typecasts the operand into an integer.

* It is probably a bad idea to trust an "equal to" involving one or more floating-point arguments. Floating-point numbers are very seldom "equal." but, it is supported nevertheless.

Table 3.4: Operators available in VHDL Preambles and Board Descriptions.

In addition to the infix operators, several functions are also available. All of the defined functions can be found in Table 3.5. Some of them (`min`, `max`, and `bits`) are simple arithmetic functions. Others allow the preamble to reference information about the node being implemented. These later functions allow the VHDL to contain optimizations based on the context of the node in the AHAHA graph. For example, many nodes obtain a value from one of their inputs which is retained in a register and used for several following clock cycles. If such an input is actually a compile time constant, and not a signal, then the register can be omitted to save chip resources without changing the behavior of the node.

There are also 2 types of "special variables" in the preamble. The first type is "context variables" which are defined, assigned, and inserted into the symbol table before the preamble is parsed. Context variables (like many of the functions from Table 3.5) provide context information in the preamble about the node being implemented. Context

Function	Description
`bits(x)`	**Bit-width**: Returns an integer of the number of bits required to represent the integer value `x`.
`min(x,y)`	**Minimum**: Returns the smaller of the two numeric arguments `x` and `y`.
`max(x,y)`	**Maximum**: Returns the larger of the two numeric arguments `x` and `y`.
`isconst(x)`*	**Is Input Port Constant**: Returns 1 if input `x` of the node being implemented is a constant.
`const(x)`*	**Const Value of Input Port**: Returns the constant value on input `x`. It is an error if `x` is not a constant.
`clusterof(x,y)`*	**Cluster Number of Port**: Returns the number of the cluster which contains a given port. The first argument, `x`, must be one of the keywords `in` or `out`; `y` is the port number. For example "`clusterof(in,5)`" returns the input cluster number which contains port 5.
`sizeof(x,y)`*	**Size of Port**: Returns the bit-width of a port. Like `clusterof`, `x`, is `in` or `out`, and `y` is an integer. For example "`sizeof(out,3)`" returns the bit-width of output port 3.
`defined(x)`*	**Variable Existence**: Returns 1 if the variable `x` is defined, and 0 otherwise. For example "`defined($INIT)`" could be used to detect whether or not a node has an optional INIT pragma.

* This function is not available in Board Description Files.

Table 3.5: Functions available in VHDL Preambles and Board Descriptions.

information which requires an argument (such as information about a particular port) is handled by functions, while context information about the node itself (which require no arguments) is accessible using variables. The other type of special variables are ones that are declared in the preamble, and are used to provide special hints or instructions to the AHAHA to VHDL translation system. Some of these user-defined variables are mandatory, and some of them are optional. A list of all the special variables that are defined in the preamble, along with their description, is available in Table 3.6.

Variable	Type	Description
NODETYPE*§	String	The name of the node being parsed. For example BUFFERX. This is one of the node names that appear in the header section of the preamble. Because a single VHDL description can implement several nodes, this string is provided so that the VHDL can be customized based on which node it is implementing.
STYLE*§	String	The name of the node style being parsed. For example: buffered2. This is one of the styles which appear in the header section of the preamble (the actual style, not an alias). Because a single VHDL description can implement several styles, this string is provided so that the VHDL can be customized based on which style it is implementing.
style¶	String	Identical to STYLE. Only defined if the implementation is selected as the "best" style to implement the node.
library_name*	String	The name of the library to which this node implementation belongs. Currently all nodes are part of the "shared_nodes" library. This is used so that the AHAHA to VHDL translator can correctly include the package which includes the entity declaration for this node.
filename*	String	The name of the VHDL file being parsed.
num_inputs*	Integer	The number of input ports present in the node being implemented. In some nodes, this is a constant, but many of them have a variable number of inputs depending on how they are used.
num_outputs*	Integer	The number of output ports present in the node being implemented. As with input ports, some nodes are constant and some are variable.
filename*	String	The name of the VHDL file being parsed.
node_number*	Integer	The unique ID number of the node being implemented. This value is intended to help in creating useful error messages in VHDL assert statements.
num_in_clusters*	Integer	The number of input clusters which are present in the node being implemented.

Continued on next page

Continued from previous page

Variable	Type	Description
num_out_clusters*	Integer	The number of output clusters which are present in the node being implemented.
memory_latency*	Integer	The number of clock cycles which must pass between the time a request is made to memory and the value arrives. This value is only defined for nodes which access memory (currently only ND_READ_WORD_MEMORY and ND_WRITE_WORD_MEMORY), and is defined based on the memory to which they are connected.
memory_data_width*	Integer	The bit-width of the data path to memory. (It is only defined for nodes which access memory, and is defined by the memory to which it is attached).
memory_addr_width*	Integer	The bit-width of the address bus for the memory. (It is only defined for nodes which access memory, and is defined by the memory to which it is attached).
memory_words*	Integer	The number of words the memory can store. (It is only defined for nodes which access memory, and is defined by the memory to which it is attached).
DEPTH*§	Integer	The value taken from the PRAG_DEPTH pragma on the node. If no such pragma exists, then it is undefined. Used to parameterize nodes which have a variable depth, such as SHIFT_REGISTER and BUFFERX.
STEP*§	Integer	The value taken from the PRAG_STEP pragma on the node. If no such pragma exists, then it is undefined. Used to parameterize nodes which have a variable step, such as SHIFT_REGISTER.
SHIFT*§	Integer	The value taken from the PRAG_SHIFT pragma on the node. If no such pragma exists, then it is undefined. Used to parameterize the size of the shift on LEFT_SHIFT_CONST and RIGHT_SHIFT_CONST nodes.

Continued on next page

Continued from previous page

Variable	Type	Description
INIT*§	String	The value taken from the PRAG_INIT pragma on the node represented as a string of 1s and 0s. If no such pragma exists, then it is undefined. Used to parameterize nodes which need a constant initial value, such as COUNTER_CONST, or CIRCULATE_CONST.
INCR*§	String	The value taken from the PRAG_INCR pragma on the node represented as a string of 1s and 0s. If no such pragma exists, then it is undefined. Used to parameterize nodes which need a constant increment value, such as COUNTER_CONST.
entity_name¶	String	The name of the entity being declared in the VHDL file. This variable is not defined when using a special case node.
architecture_name¶	String	The name of the architecture being declared in the VHDL file. This variable is not defined when using a special case node.
SPECIAL_CASE¶	String	The name of the special case which is used to implement this node. This variable is only defined when using a special case node.
usage_ff¶ usage_lut usage_ramb usage_mult18x18	Integer	All of these variables are reserved, and contain the usage information for the node. For example usage_ff contains the number of flip-flops which are required to implement the node on the hardware.
usage_overall¶	Integer	The combined (weighted) usage information for the node. Used to select which style will be used to implement a node. Computed as described in Section 3.4
dly_in¶	Integer	The longest propagation delay from any input port to any register in the node. Together with the dly_out and dly_thru values of other nodes, this value can be used to estimate the clock frequency limitations caused by interactions between nodes. This value is only defined in clocked nodes.

Continued on next page

Continued from previous page

Variable	Type	Description
dly_out¶	**Integer**	The longest propagation delay from any register to an output of the node. This value is only defined in clocked nodes.
dly_thru¶	**Integer**	The longest propagation delay from any input port to any output port. This value is only defined in unclocked nodes, since clocked nodes do not have combinational paths though them.
dly_int¶	**Integer**	The longest propagation delay from any register inside the node to any register inside the node. This value can be used to estimate the maximum clock frequency of the node, disregarding any other nodes in the design. This value is only defined in clocked nodes.
clktype†	**String**	Must be one of "clocked," "unclocked," or "semiclocked," and is case-insensitive. Indicates whether the node is clocked, unclocked, or semiclocked.
latency‡	**Integer**	Declares the number of clock cycles between the arrival of input tokens, and the production of any corresponding output tokens. Either this value must be defined, or min_latency and max_latency must be defined. These 3 values are present so that the buffer balancing routine can function correctly. The latency value is preferred over the others if it is defined.
min_latency‡	**Integer**	Declares the minimum number of clock cycles between the arrival of input tokens, and the production of any corresponding output tokens.
max_latency‡	**Integer**	Declares the maximum number of clock cycles between the arrival of input tokens, and the production of any corresponding output tokens without any stalling required.

Continued on next page

Continued from previous page		
Variable	Type	Description
skip_compile‡	Integer	If this variable is defined, and contains a nonzero value, then the test-compilation of this node will be skipped. This is intended for trivial nodes which are guaranteed not to generate any logic whatsoever. CHANGE_WIDTH, and LEFT_SHIFT_CONST nodes are an example of nodes where this is useful. It is never an error to omit this variable (it will merely waste time). Including it on a non-trivial node will lead to incorrect behavior.
ignore_consts‡	Integer List	Because the AHAHA to VHDL translator caches information generated by test compilations, it normally needs to ensure that a change in a constant input will lead to a different test compilation. For example, multiplying by a constant 2, and multiplying by a constant 13 will generate wildly different hardware. This variable contains the indexes of input ports which will not have an impact on the generated hardware, such as the stream number on STREAM_PUT nodes.
stream_id‡	Integer List	Must be defined for any node which operates on a stream. Defines which stream it operates upon. Copied from the input port which contains this information, which varies depending on which stream operation node is being implemented.

∗ This variable is automatically defined by the parser. It cannot be overridden by the user, but may be referred to in the preamble.
† This variable is mandatory, and must be defined by the user.
‡ This variable is optionally defined by the user, and has an effect in the handling of the node.
§ This variable is intended to be used to override the generic by the same name in the entity, if such a generic exists.
¶ This variable is created by the parser *after* the preamble is parsed. It is an error to define or reference this variable in the preamble. It is used internally by the AHAHA to VHDL translator.

Table 3.6: Special variables available in the VHDL Preamble.

Variables which share the same name as a generic in the generic map of the VHDL entity will be used to set the value of that generic when instantiating the VHDL component. By convention, VHDL generics are in all capitals, although this convention is not

enforced, it is used in the preamble as well; variables in all capitals are intended to be bound to the generics of the same name. This is true for both context variables, as well as user-defined variables.

Variables which have the same name as a VHDL port also have a special meaning. Such variables are expected to be integer lists, and represent a binding from one or more ports on the node to the VHDL port. If the VHDL port is an "`in`" port, then the binding is made to input ports of the node; if the VHDL port is an "`out`" port, then the binding is made to output ports. When the list of integers is longer than 1 element, the values are concatenated for input ports and separated for output ports. For example, if the VHDL entity has an `in` port named `Data`, and the preamble defines a variable `Data` to be `[2,5,1]`, then inputs 2, 5, and 1 of the node will be concatenated together into one large std_logic_vector, and supplied to the `Data` port of the entity when it is instantiated. The most significant bit of input 2 will be the most significant bit of the concatenated value. The least significant bit of input 1 will be the least significant bit of the concatenated value. If the VHDL entity had an `out` port named `Values`, and the preamble defined a variable `Values` to be `[3,4]`, then the value produced by the `Values` port of the VHDL entity will be separated and assigned to outputs 3 and 4 of the node. Output 3 will contain the high bits, and 4 will contain the low bits. All ports which are bound in this fashion must be of type `std_logic_vector`.

Many nodes require a compile time constant value to parameterize their behavior. In cases where a value is provided to the node which must be a constant, it is given by a pragma on the node. A port on a node is intended for values which may or may not be constant, and are bound to the ports of the entity. Pragmas provide values which must be constants, and are bound to the generics of the entity.

Any ports in the entity, which are not bound to integer lists in the preamble, must be one of the special port names listed in Table 3.7. When one of the named ports appears in an entity declaration in the VHDL file, it is automatically connected to the correct signal in the design, as described in the table. Handshaking signals were explained in

Section 3.3.1. All of the `port_` signals are used to communicate to memories via the appropriate arbitrator, and are only available for nodes which are attached to memory. Similarly, all of the `bram_` signals are used to communicate with BlockRAMs, and are only available for nodes which are attached to a BlockRAM. There is no arbitrator involved for BlockRAMs, so these signals are attached directly to the appropriate BlockRAM. The `stream_` signals are used to communicate with streams, and are only valid for stream operation nodes. All of the `stream_` signals must be 0 when not in use, because all of the signals of the same type from the stream clients are OR-ed together. The graph contains explicit signals to ensure that no stream nodes will collide.

Port	Dir	Type	Description
Clock	in	std_logic	The clock signal used in the design. Generated by the clock generator on the RCS board.
Reset	in	std_logic	Reset signal which puts the node into a valid known initial state. This signal is the AHAHAReset signal from the foreman.
Ready_In	out	std_logic_vector	Described in Section 3.3.1
Ready_Out	out	std_logic_vector	Described in Section 3.3.1
AllReady	in	std_logic_vector	Described in Section 3.3.1
AllReady_In	in	std_logic_vector	Described in Section 3.3.1
AllReady_Out	in	std_logic_vector	Described in Section 3.3.1
port_req	out	std_logic	Signals a request is being made to access memory.
port_ack	in	std_logic	Signals whether the memory request is being serviced or not. Only valid when port_req is asserted.
port_write	out	std_logic	Indicates that the memory request is a write request or a read request. Only valid when port_req is asserted.
port_addr	out	std_logic_vector	The address where the request is being made. Only valid when port_req is asserted.
port_data_out	out	std_logic_vector	The data being written to the memory on a write request. Only valid when port_req and port_write are asserted.
port_data_in	in	std_logic_vector	The data being returned from the memory after a read request. Only valid when port_data_valid is asserted.
port_data_valid	in	std_logic	This signal is simply the port_req signal, delayed by the memory latency. It signals when a memory transaction has completed. In the case of reads, it signals that the data is available on the port_data_in signal.
bram_req	out	std_logic	Identical to port_req, except that it is used for BlockRAMs instead of off-chip memories.
bram_write	out	std_logic	Same as port_write for BlockRAMs.

Continued on next page

Continued from previous page			
Port	Dir	Type	Description
bram_address	out	std_logic_vector	Same as port_addr for Block-RAMs.
bram_dataout	out	std_logic_vector	Same as port_data_out for Block-RAMs.
bram_datain	in	std_logic_vector	Same as port_data_in for Block-RAMs. Note that since Block-RAMs always have a latency of 1 clock cycle, no valid signal is needed.
stream_close	out	std_logic	Closes the associated stream, indicating that no more data will be enqueued.
stream_enqueue	out	std_logic	Causes a value currently on the stream_data_out signal to be enqueued into the stream.
stream_dequeue	out	std_logic	Causes the value currently present on the stream_data_in to be dequeued from the stream.
stream_closed	in	std_logic	Indicates that the stream has been closed and is empty, indicating that no further dequeuing is possible.
stream_inclosed	in	std_logic	Indicates that the stream has been closed, indicating that no further enqueuing is possible.
stream_exists	in	std_logic	Indicates that the stream_data_in signal contains a valid stream value. Or alternatively, indicates that a dequeue operation is permitted.
stream_notfull	in	std_logic	Indicates that the stream has space remaining. Or alternatively, indicates that an enqueue operation is permitted.
stream_data_in	in	std_logic_vector	The next value in the stream.
stream_data_out	out	std_logic_vector	The value to be enqueued into the stream.

Table 3.7: Special port names in VHDL port map.

3.4.1.1 Assignment Statements

Assignment statements tend to be the most prevalent statements in the preamble. They are all of the form:

```
--> $variable = expression;
```

are used to simultaneously create, and assign a new variable. The value of variables cannot be changed. Variables inherit their type from the type of the expression used on the right-hand side of the assignment. There are 6 types: integer, float, string, list of integers, list of floats, or list of strings. Lists are defined by surrounding the elements with square brackets, and separating the elements with commas. Portions of lists containing sequential values may be shortened to only include the list extrema separated by a tilde ($\sim$). For example the following 2 lines are equivalent.

```
--> $my_list = [11, 7 ~ 10, 1, 6 ~ 2, 0];
--> $my_list = [11, 7, 8, 9, 10, 1, 6, 5, 4, 3, 2, 0];
```

Mixed type lists are not allowed. For example the right-hand side of:

```
--> $my_list = [6, 7.4, 8.3];
```

is not a valid list since it contains both integers and floats. If the "`6`" were replaced with "`6.0`" or "`float 6`" it would be valid. Strings are enclosed in double quotes. Expressions may contain any of the operators described in Table 3.4, as well as any of the functions described in Table 3.5.

3.4.1.2 Assertion Statements

Assertion statements are statements which include only a single integer expression. If the expression is nonzero, then the statement is ignored. However, if any of the assert statements in the preamble evaluate to zero, then the node style being parsed is marked as not being a valid implementation of the node, and is discarded. This allows special case implementations of certain nodes to be built. For example, there may be an especially efficient method of implementing a particular node, which is only valid under certain contexts. In the preamble, the expected context can be asserted. If any of the assertions fail, then the translator will fall back to a more general (and presumably more costly) implementation of the node.

Assertion statements are also used for basic sanity checking. Many nodes have a constant number of ports, or clusters. It is wise to assert these constants are true at the

top of the preamble. Likewise, many of the ports of a node may need to have identical (or related) bit-widths. These sorts of restrictions should be asserted. Some nodes expect that some of their ports are always (or never) constants. These situations can be asserted by using the `isconst` function described in Table 3.5.

In the case where all possible implementations of a node failed at least one of their assertions, the translator will produce an error message indicating that no suitable node implementations were found. In most cases, this is the result of a malformed AHAHA graph being generated by the SA-C* compiler.

3.4.1.3 If Statements

If statements work exactly as one might expect, and come in 2 forms:

```
--> if(expression) {
-->   statements
--> } else {
-->   statements
--> }
```

```
--> if(expression) {
-->   statements
--> }
```

If the expression in parenthesis is true, then the second block of statements (if one exists) will be ignored. If the expression is false, then the first block will be ignored. Nested if statements are permitted, and behave as expected. It is worth noting that the ignored statements will still be parsed and evaluated, even though they will not have any discernible effect. Any syntax errors in the ignored section will still be syntax errors. Effectively, when in an ignored section, assignment statements merely discard their right-hand side, and assert statements always succeed regardless of their value. Therefore, in an ignored section, it is acceptable to have statements which redefine existing variables, but it is not acceptable to have statements which reference nonexistent variables in their expressions.

3.4.2 Example AHAHA Graph

Figure 3.9 shows the AHAHA graph generated from the DFG in Figure 2.7. The logic implementing the control signals used to identify dummy-iterations and the final iteration

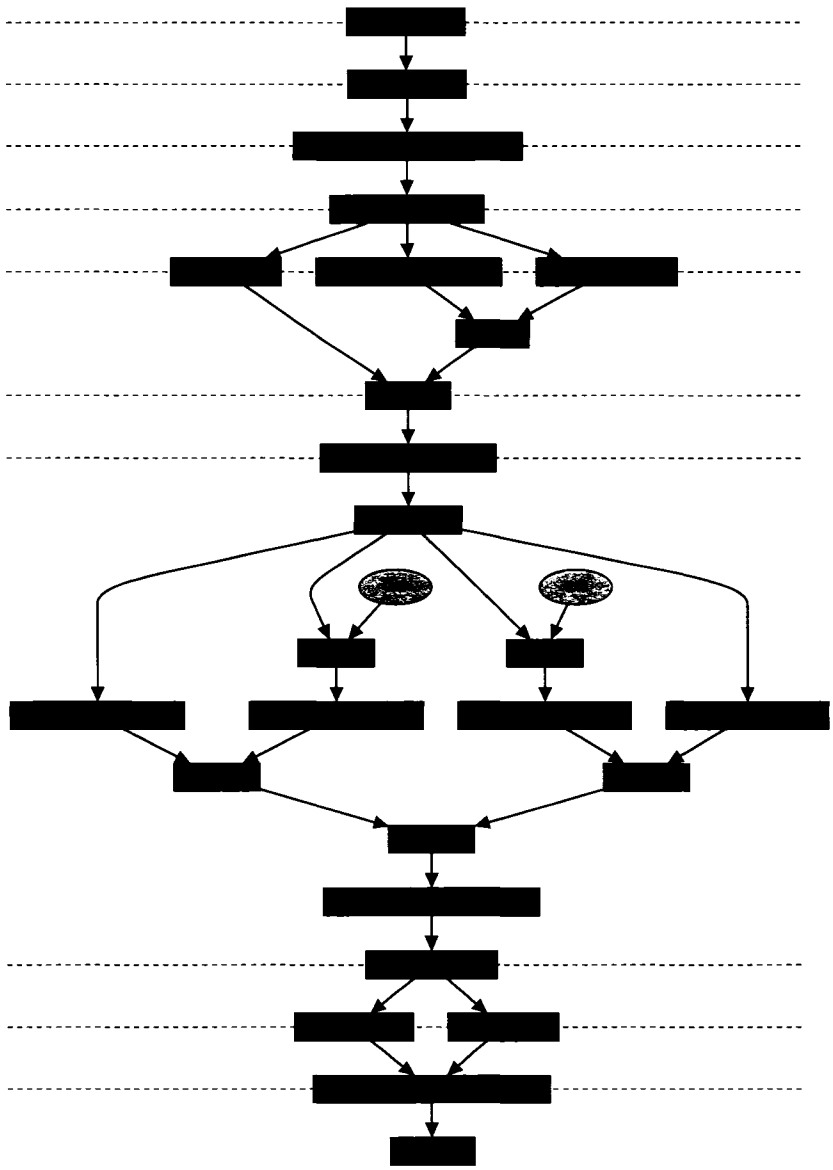


Figure 3.9: AHAHA graph of code in Figure 2.10.

has been removed for visual simplicity. The `RC_WINDOW_GEN_1D` from the DFG has been expanded into several nodes at the top of the graph:

- **`TOK_GEN`** reads a runtime constant n, from the register file (provided by the host), which represents the number of 32-bit words used to store S in the off-chip memory. The `TOK_GEN` node then produces n tokens which drive the loop. In this case $n = \lceil |S|/4 \rceil$, since 4 of the 8-bit values of S are packed into each 32-bit word.

- **COUNTER** begins counting at the memory address of S (obtained from the register file), and increments each time it receives a token from TOK_GEN. In this case, S is stored at address 0, so the COUNTER will produce the values 0,1,2,3...$(n-1)$.

- **READ_WORD_MEMORY** takes the address from the COUNTER, and produces the data values stored in those addresses. It is attached to the memory arbitrator which is not shown.

- **FIFO_UNPACK** takes each 32-bit word, and unpacks it serially into four 8-bit values; therefore, it will produce $n \times 4$ values, which might be (up to 3) more values that are actually in S.

- **REPLICATOR** is used to replicate a value several times. In this example, it is replicating a value, l, the length of S, which was obtained from the register file. Each time it receives a token from the FIFO_UNPACK, it produces a copy of l.

- **COUNTER_CONST** is similar to a COUNTER, except that the value where it begins counting is a compile time constant. In this example, the constant is 0. It counts the number of values of S which have been produced by the FIFO_UNPACK.

- **BUFFERX** is a variable depth buffer (in this case only depth 1), which is present for buffer balancing. It has been inserted by the compiler to prevent the edge between the FIFO_UNPACK and the MASK from crossing a section boundary.

- **MASK** is responsible for removing the excess values from the end of the stream, which are not part of S. Such values are an artifact of packing. It uses the value from UGT (unsigned greater than) to determine when to begin removing values from the stream.

- **SHIFT_REGISTER** produces the sliding window of 4 values packed together into a 32-bit word. It will produce $(n \times 4) - 3$ values.

- **UNPACK** is an unclocked node, which simply unpacks the 4 8-bit values from the 32-bit value from the SHIFT_REGISTER in parallel. This node is essentially just rewiring and is free in the hardware implementation.

The computational body of the loop, is a direct translation of the loop body from the DFG. The WRITE_TILE_1D_1D from the DFG in Figure 2.7 has been expanded into several nodes:

- **FIFO_PACK** takes four 8-bit values serially, and packs them into a 32-bit word.
- **COUNTER** behaves in a similar way as described above.
- **BUFFERX** serves the same purpose as in the window generator portion.
- **WRITE_WORD_MEMORY** takes the address from the COUNTER, and the data from the FIFO_PACK, and writes each word into the specified address. When the loop has completed, it signals the DONE node.
- **DONE** signals the host that the FPGA has finished the computation.

3.5 Board Description Files

To allow AHAHA to target multiple RCS boards, the details of each board are not hard-coded into the translator, but are defined within a board description file. An example board description file is shown in Figure 3.10. The strings "//," "--," and "#," are all used to begin single-line comments. C-style block comments (`/* Like this */`) are also supported. Because the board information may be requested multiple times, and parsing it requires a non-trivial amount of work, once a particular board description file is parsed the first time, it is cached internally, so that it does not need to be parsed again. Because there is no external context information associated with a board description file, this is safe and efficient.

The operators from Table 3.4, and some of the functions from Table 3.5 are available for use in expressions within the board description files. The board description file for

```
// The Alpha Data ADM-XRC-II boards in lennon and harrison

1 chip 6 mem;

// Vitals about the board
board {
  $config_mem_start  = 2;
  $default_chip      = 1;
  $words_per_mem     = 1024 * 256;
  $library           = ["alphadata"];
  $syn               = "alphadata/syn";
  $sim               = "alphadata/sim";
}

// all 6 of the memories are like this:
default_mem {
  $type         = "ZBT";
  $chip         = 1;
  $latency      = 4;
  $data_width   = 32;
  $addr_width   = 18;
  $words        = $words_per_mem;
}

// Only one chip
chip 1 {
  $vhdl_entity       = "alphadata.vhd";
  $library           = ["virtex2"];
  $default_read_mem  = 1;
  $default_write_mem = 2;
  $technology        = "VIRTEX2";
  $device            = "2V6000";
  $package           = "FF1152";
  $speed_grade       = "5";

  $ramb_widths       = [    1,    2,    4,    9,   18,  36];
  $ramb_depths       = [16384, 8192, 4096, 2048, 1024, 512];
  $rsrc_row          = 96;
  $rsrc_col          = 88;
  $rsrc_clb          = $rsrc_col * $rsrc_row;
  $rsrc_slice        = $rsrc_clb * 4;
  $rsrc_ff           = $rsrc_slice * 2;
  $rsrc_lut          = $rsrc_slice * 2;
  $rsrc_ram_mul_row  = 24;
  $rsrc_ram_mul_col  = 6;
  $rsrc_ramb         = $rsrc_ram_mul_col * $rsrc_ram_mul_row;
  $rsrc_mult18x18    = $rsrc_ram_mul_col * $rsrc_ram_mul_row;
  $weight_ff         = 1.0;
  $weight_lut        = 1.0;
  $weight_ramb       = float $rsrc_row * $rsrc_ff / $rsrc_clb / $rsrc_ram_mul_row;
  $weight_mult18x18  = float $rsrc_row * $rsrc_ff / $rsrc_clb / $rsrc_ram_mul_row;
}
```

Figure 3.10: Board Description File of the Alpha Data ADM-XRC-II board.

a given board contains information about its structure including the FPGA chips and memories on the board. The file begins with a "size line" which declares how many chips and memories are available on the board. For the ADM-XRC-II board, there is 1 chip and 6 on-board memories, hence the "`1 chip 6 mem`" size line in the file. The remainder of the file defines several symbol tables. A symbol table is a series of variable assignments enclosed with braces, which is prefixed by a declaration of what type of symbol table it is. The available symbol table declarations are: "`board`," "`default_chip`," "`default_mem`," "`chip x`" (where x is an integer defining which chip), and "`mem x`" (where x is an integer defining which memory).[6]

The first symbol table contains variables which define the structure, behavior and parameters of the board itself. There are several special variable names which are required, or otherwise have special meaning. They are described in Table 3.8.

Following the board symbol table definition are the default chip and default memory symbol table definitions. These are optional tables which define symbols globally for all of the chips and memories on the board, respectively. If a variable is referenced within one of these tables, which is not previously defined in it, then it will refer to the symbol (if any) in the board symbol table. In effect, these symbol tables implicitly inherit all of the values in the board table. However, if a symbol is defined which is also present in the board table, it does not affect the value of the symbol in the board table, it merely shadows it for the remainder of the current table.

[6]Chips and memories are both 1-based; there is no chip 0, and no memory 0.

Variable	Type	Description
name*	String	The name of the board being described. This is also the name of the board description file. (For example "alphadata_0" is the default in the SA-C* compiler, if the program does not specify otherwise.)
filename*	String	The full absolute path and file name of the board description file.
num_chips*	Integer	The number of chips on the board. Determined by the size line of the board description file.
num_mems*	Integer	The number of memories on the board. Determined by the size line of the board description file.
default_chip†	Integer	The chip number to be used by default if the SA-C* code does not specify which chip to use.
config_mem_start†	Integer	Address in the configuration memory where parameters begin. (This corresponds to the first address in the register file which can be used by the SA-C* program, and is not allocated for use by the RTS API for the target board).
library†	String List	A list of libraries which need to be included in the VHDL libraries list when compiling designs for this board.
syn†	String	The name of the directory (under the "data" directory in the top-level directory of the AHAHA to VHDL translator) which contains the VHDL files needed when synthesizing designs for this board.
sim†	String	The name of the directory (under the "data" directory) which contains the VHDL files needed when simulating designs for this board.

* This variable is mandatory, and is automatically defined by the parser. It cannot be overridden by the user, but may be referred to.
† This variable is mandatory, but must be defined by the user.

Table 3.8: Special variables in the "board" symbol table.

Next, is a series of optional symbol tables for each chip. Each of the chip symbol tables inherit values from the default chip symbol table in the same way that the default chip table inherits its values from the board table. Because the default chip table inherits the board symbol table, so do the individual chip symbol tables. The special variable names which are required, or otherwise have special meaning in the chip tables are listed in Table 3.9. Mandatory variables can either be present in the per-chip table(s), the default chip table, or the board table.

Variable	Type	Description
`chip_id`*	**Integer**	Which chip number to which this symbol table belongs.
`default_read_mem`†	**Integer**	The memory number which should be used by default to store arrays which are read by the graph. This can be overridden by the SA-C* code, and is only used if it is not specified there. The memory must be attached to this chip (see "`chip`" in Table 3.10)
`default_write_mem`†	**Integer**	The memory number which should be used by default to store arrays which are produced by the graph. This can be overridden by the user's SA-C* code, and is only used if it is not specified there. The memory must be attached to this chip (see "`chip`" in Table 3.10)
`default_read_mem`†	**Integer**	The memory number which should be used by default to store arrays which are read by the graph. This can be overridden by the SA-C* code, and is only used if it is not specified there.
`library`†	**String List**	A list of libraries which need to be included in the VHDL libraries list when compiling nodes for this chip. Typically, this list will include libraries which contain VHDL implementations which are specific to this chip family.

Continued on next page

Continued from previous page		
Variable	Type	Description
vhdl_entity†	String	The name of the VHDL file which contains the top-level entity declaration, static libraries, and the static portion of the architecture for this chip. The file should be in the "data" directory.
ramb_widths	Integer List	A list of the available bit widths of the BlockRAMs available on the chip. When combined with ramb_depths, they enumerate all of the possible physical BlockRAM geometries available. These variables are required if BlockRAMs will be used.
ramb_depths	Integer List	A list of the available depths of the BlockRAMs available on the chip. They must be arranged in a one-to-one correspondence with the values in the ramb_widths variable above.
weight_ff† weight_lut† weight_ramb† weight_mult18x18†	Float	This set of variables which are used to weight the cost of using the various on chip resources. Currently, the supported resources are ff, lut, ramb, and mult18x18, which correspond to Flip-Flops, Lookup Tables, BlockRAMs, and Embedded Multipliers, respectively. For example weight_ff stores the relative cost of using flip-flops.
technology†	String	The technology of the chip, ie, "VIRTEX2."
part†	String	The part number of the chip, ie, "2V6000."
package†	String	The package of the chip, ie, "FF1152."
speed_grade†	String	The speed grade of the chip, not including the dash, i.e "5."

* This variable is mandatory, and is automatically defined by the parser. It cannot be overridden by the user, but may be referred to.
† This variable is mandatory, but must be defined by the user.

Table 3.9: Special variables available in the "chip" symbol tables.

Lastly, the file contains a series of optional tables which are specific to each memory. As with the chip tables, the memory tables inherit their values from the default memory table (and indirectly, the board table). The special variable names which are required, or otherwise have special meaning in the memory tables, are listed in Table 3.10. Mandatory variables can either be present in the per-memory table(s), the default memory table, or the board table.

Although both the default chip/memory tables and the individual tables for each chip or memory are optional, one or the other should be defined.

Variable	Type	Description
`mem_id`*	**Integer**	Memory number to which this symbol table is associated.
`words`†	**Integer**	The number of words that the memory can address.
`addr_width` ‡	**Integer**	The number of bits that are required to address the memory.
`data_width`†	**Integer**	The number of bits in each word in the memory. Currently the upper levels of the compiler only support 32-bit memories, although the AHAHA is more flexible.
`chip`†	**Integer**	The chip number to which the memory is attached. Currently, only a single chip may be connected to a memory.
`latency`†	**Integer**	The number of clock cycles which must pass between a memory read request, and the value arriving from the memory.

* This variable is mandatory, and is automatically defined by the parser. It cannot be overridden by the user, but may be referred to.
† This variable is mandatory, but must be defined by the user.
‡ This variable is mandatory, but if the user does not set it, it will be inferred from other variables.

Table 3.10: Special variables in the board description file's "chip" symbol tables.

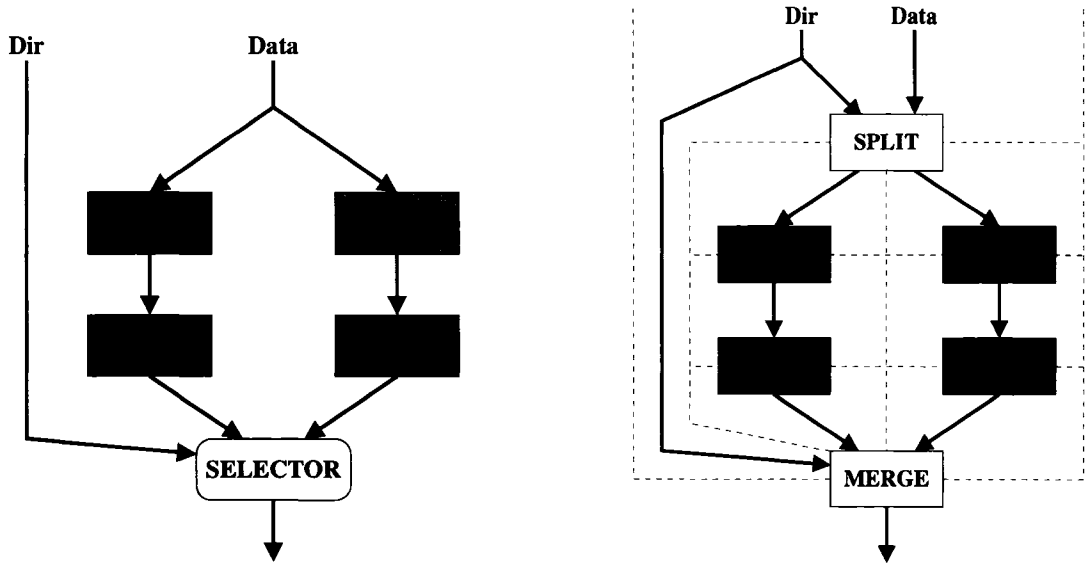
Chapter 4

AHAHA Usage

There are a vast number of ways in which the AHAHA nodes may be combined to create hardware behaviors. Only a small portion of these are exploited by the SA-C* compiler. This chapter presents possible configurations of AHAHA nodes to illustrate its flexibility, and potential complexity. In many cases, there are multiple possible mappings from source code to AHAHA. The graphs shown in this chapter are just one illustrative possibility intended to provide the reader with an intuition for the construction of AHAHA graphs, and are not meant to be a definitive mapping.

4.1 Conditionals

The AHAHA supports 2 main methods for implementing conditionals. They are shown in Figure 4.1. In both examples, more than 2 conditional branches are supported, and a signal (`Dir`) is used to select which of the conditional branches is active. The `Data` is simply any data needed to execute the body of the branches. When none of the code blocks in the branches of the conditional are side effecting, a `SELECTOR` (Figure 4.1 a) may be used. Using this method, all branches of the conditional are executed in parallel, and multiple results are computed. The `SELECTOR` node is then used to select which of the values will be used. Because the `SELECTOR` node itself is unclocked, there are no wasted clock cycles when using this method. The time taken to execute the conditional is the maximum of the times to execute the branches. Conditionals using `SELECTOR`s are easily pipelined.



a: Using a SELECTOR.

b: Using SPLIT and MERGE.

Figure 4.1: Methods of implementing conditionals.

If one or more of the conditional branches are side effecting then it is not acceptable to execute all of the branches; only the active branch should be executed, the others should be dormant. For example, imagine a conditional which contains a `STREAM_GET` node in each of its branches. If both of these nodes are allowed to execute, only one of the values will be used, and the other will be lost. The `SA-C*` compiler considers all stream and memory nodes to be side effecting. Technically, some of them may not actually be side effecting, such as `READ_WORD_MEMORY`, or `STREAM_IS_OPEN` nodes. A false positive (identifying a node as causing a side effect, when in fact they do not) is not a problem.[1] False negatives, however, will cause execution errors. In cases where one or more of the conditionals are considered to be side effecting, a pair of `SPLIT` and `MERGE` nodes is used. The `SPLIT` lies above the conditional branches, and `MERGE` lies below. Both nodes are given the `Dir` signal. The `SPLIT` node uses the `Dir` signal to decide which branch will receive the `Data`. The `MERGE` node uses the `Dir` signal to know which

[1] In fact, as we will see in Section 5.2, there are occasions where it may be more efficient.

branch it should expect data from. Because each conditional branch lies within its own section, the nodes in the non-active branches do not execute. The SPLIT and MERGE nodes both are clocked nodes, so a minimum of 2 clock cycles of latency are added to the computation (one for each node). In typical usage, the conditional will be pipelined, making the extra latency unimportant. However, when the conditional lies within a loop which precludes pipelining, the latency may be costly.

4.1.1 Predicated Instruction Optimization

There are several circumstances where the SA-C* compiler chooses to use a costly SPLIT-MERGE conditional, while a more optimized implementation is possible using predicate-flags. One of these circumstances occurs when conditional branches contain exclusively stream operations. The stream-flag which is used to serialize the stream operations may also be used to suppress the stream operations in all but one of the branches of the conditional.

This technique is similar to, for example, IA-64 [34] predicated instructions present in the Itanium [35] architecture. Predicated instructions in Von Neumann processors incur a time penalty; time is taken to process the instruction regardless of the predicate.

In hardware, no time is wasted, and no additional area is required, compared with a SPLIT/MERGE pair. Predicated instructions remove the latency added by the SPLIT and MERGE nodes, improving the performance, and area utilization of the codes. Some AHAHA nodes have built-in support for predicated instructions. The CIRCULATE nodes take a boolean flag which is used to suppress the update of the stored value internally. This allows them to be used to perform masked reduction operations. The STREAM_PUT nodes accept a boolean flag (which is distinct from the stream-serialization-flag) which causes them to ignore the associated data element.

A similar behavior is achieved with the Conditional ALU (CALU) mode supported by the computational elements of the Instruction Set Extension Fabric (ISEF) of the S5 processor engine[36] developed at Stretch[37]. When an Arithmetic Logic Unit (ALU), is used in CALU mode, an additional user supplied bit is used to select the ALU behavior

between two operations. For example, the CALU mode can be used to select between a mathematical function (such as a multiply, or an add) and a no-op instruction which simply passes one of its operands through unaltered. This technique can also be used to create multiplexers and adder-subtracters efficiently.

4.1.1.1 Pipelined Mergesort

The pipelined mergesort (shown in Table 4.1) is one such code where the predicated instruction optimization is possible. The details of the algorithm are discussed by Böhm [38]. The SA-C* code shown here uses a sorting network with 16 sorting processes, which, when combined, is capable of sorting sequences of 2^{16} (65536) elements; each process in the network doubles the size of the resulting sorted sequences. In addition to the 16-process version shown here, variants using 1 to 15 processes were used in the testing, and follow trivially from this code.

```
1: export main;
2:
3: process leftimg (out stream uint8 outS1, stream uint8 outS2, param uint8 A[:])  {
4:   for window a[2] in A step (2) {
5:     put(a[0],outS1);
6:     put(a[1],outS2);
7:   };
8: };
9:
10: process middle (in stream uint8 inS1, stream uint8 inS2,
11:                 out stream uint8 outS1, stream uint8 outS2,
12:                 param uint18 chunksize) {
13:   if(chunksize == 1) {
14:     while (inS1) {
15:       uint8 t_l, uint8 t_r = get(inS1), get(inS2);
16:       uint8 t_min, uint8 t_max = (t_l < t_r ?  t_l , t_r :  t_r , t_l);
17:       put(t_min,outS1);
18:       put(t_max,outS1);
19:
20:       uint8 b_l, uint8 b_r = get(inS1), get(inS2);
21:       uint8 b_min, uint8 b_max = (b_l < b_r ?  b_l , b_r :  b_r , b_l);
22:       put(b_min,outS2);
23:       put(b_max,outS2);
24:     };
25:   } else {
26:     while (inS1) {
27:       uint18 t1_incnt1, uint18 t1_incnt2 = 0, 0;
28:
29:       uint18 t2_incnt1, uint18 t2_incnt2 =
30:         while ((t1_incnt1 < chunksize) && (t1_incnt2 < chunksize)) {
31:           uint8 p1, uint8 p2 = peek(inS1), peek(inS2);
32:           bool which = p1 < p2;
33:           put(which ?  p1 :  p2, outS1);
34:           if(which) {
35:             get(inS1);
36:           } else {
37:             get(inS2);
38:           };
39:
40:           next t1_incnt1 = t1_incnt1 + (which?1:0);
41:           next t1_incnt2 = t1_incnt2 + (which?0:1);
42:         } return(final(t1_incnt1),final(t1_incnt2));
43:
44:       while(t2_incnt1 < chunksize) {
45:         put(get(inS1), outS1);
46:         next t2_incnt1 = t2_incnt1 + 1;
47:       };
48:
49:       while(t2_incnt2 < chunksize) {
50:         put(get(inS2), outS1);
51:         next t2_incnt2 = t2_incnt2 + 1;
52:       };
53:
54:       uint18 b1_incnt1, uint18 b1_incnt2 = 0, 0;
55:
```

Continued on next page

Continued from previous page

```
56:       uint18 b2_incnt1, uint18 b2_incnt2 =
57:         while ((b1_incnt1 < chunksize) && (b1_incnt2 < chunksize)) {
58:           uint8 p1, uint8 p2 = peek(inS1), peek(inS2);
59:           bool which = p1 < p2;
60:           put(which ?  p1 :  p2, outS2);
61:           if(which) {
62:             get(inS1);
63:           } else {
64:             get(inS2);
65:           };
66:
67:           next b1_incnt1 = b1_incnt1 + (which?1:0);
68:           next b1_incnt2 = b1_incnt2 + (which?0:1);
69:         } return(final(b1_incnt1),final(b1_incnt2));
70:
71:       while(b2_incnt1 < chunksize) {
72:         put(get(inS1), outS2);
73:         next b2_incnt1 = b2_incnt1 + 1;
74:       };
75:
76:       while(b2_incnt2 < chunksize) {
77:         put(get(inS2), outS2);
78:         next b2_incnt2 = b2_incnt2 + 1;
79:       };
80:     };
81:   };
82: };
83:
84: process right (in stream uint8 inS1, stream uint8 inS2,
85:                out stream uint8 outS,
86:                param uint18 chunksize) {
87:   if(chunksize == 1) {
88:     while (inS1) {
89:       uint8 t_l, uint8 t_r = get(inS1), get(inS2);
90:       uint8 t_min, uint8 t_max = (t_l < t_r ?  t_l , t_r :  t_r , t_l);
91:       put(t_min,outS);
92:       put(t_max,outS);
93:     };
94:   } else {
95:     while (inS1) {
96:       uint18 t1_incnt1, uint18 t1_incnt2 = 0, 0;
97:       uint18 t2_incnt1, uint18 t2_incnt2 =
98:         while ((t1_incnt1 < chunksize) && (t1_incnt2 < chunksize)) {
99:           uint8 p1, uint8 p2 = peek(inS1), peek(inS2);
100:          bool which = p1 < p2;
101:          put(which ?  p1 :  p2, outS);
102:          if(which) {
103:            get(inS1);
104:          } else {
105:            get(inS2);
106:          };
107:
108:          next t1_incnt1 = t1_incnt1 + (which?1:0);
109:          next t1_incnt2 = t1_incnt2 + (which?0:1);
110:        } return(final(t1_incnt1),final(t1_incnt2));
111:
```

Continued on next page

Continued from previous page

```
112:       while(t2_incnt1 < chunksize) {
113:         put(get(inS1), outS);
114:         next t2_incnt1 = t2_incnt1 + 1;
115:       };
116:
117:       while(t2_incnt2 < chunksize) {
118:         put(get(inS2), outS);
119:         next t2_incnt2 = t2_incnt2 + 1;
120:       };
121:     };
122:   };
123: };
124:
125: process network main(out stream uint8 sorted, param uint8 A[:])  {
126:   stream uint8 t0; stream uint8 b0;
127:   stream uint8 t1; stream uint8 b1;
128:   stream uint8 t2; stream uint8 b2;
129:   stream uint8 t3; stream uint8 b3;
130:   stream uint8 t4; stream uint8 b4;
131:   stream uint8 t5; stream uint8 b5;
132:   stream uint8 t6; stream uint8 b6;
133:   stream uint8 t7; stream uint8 b7;
134:   stream uint8 t8; stream uint8 b8;
135:   stream uint8 t9; stream uint8 b9;
136:   stream uint8 t10; stream uint8 b10;
137:   stream uint8 t11; stream uint8 b11;
138:   stream uint8 t12; stream uint8 b12;
139:   stream uint8 t13; stream uint8 b13;
140:   stream uint8 t14; stream uint8 b14;
141:   stream uint8 t15; stream uint8 b15;
142:
143:   hardware() {
144:     instantiate leftimg (t0, b0, A);
145:     instantiate middle (t0 @ regfile 16, b0 @ regfile 16, t1, b1, 1);
146:     instantiate middle (t1 @ regfile 16, b1 @ regfile 16, t2, b2, 2);
147:     instantiate middle (t2 @ regfile 16, b2 @ regfile 16, t3, b3, 4);
148:     instantiate middle (t3 @ regfile 16, b3 @ regfile 16, t4, b4, 8);
149:     instantiate middle (t4 @ regfile 17, b4 @ regfile 16, t5, b5, 16);
150:     instantiate middle (t5 @ regfile 33, b5 @ regfile 32, t6, b6, 32);
151:     instantiate middle (t6 @ regfile 65, b6 @ regfile 64, t7, b7, 64);
152:     instantiate middle (t7 @ regfile 129, b7 @ regfile 128, t8, b8, 128);
153:     instantiate middle (t8 @ regfile 257, b8 @ regfile 256, t9, b9, 256);
154:     instantiate middle (t9 @ regfile 513, b9 @ regfile 512, t10, b10, 512);
155:     instantiate middle (t10 @ regfile 1025, b10 @ regfile 1024, t11, b11, 1024);
156:     instantiate middle (t11 @ regfile 2049, b11 @ regfile 2048, t12, b12, 2048);
157:     instantiate middle (t12 @ regfile 4097, b12 @ regfile 4096, t13, b13, 4096);
158:     instantiate middle (t13 @ regfile 8193, b13 @ regfile 8192, t14, b14, 8192);
159:     instantiate middle (t14 @ regfile 16385, b14 @ regfile 16384, t15, b15,
    16384);
160:     instantiate right (t15 @ regfile 32769, b15 @ regfile 32768, sorted @ regfile
    65536, 32768);
161:   };
162: };
```

Table 4.1: Listing of pipelineSort_16_65536.sc.

The code has been tailored to be as efficient as possible in terms of performance, at the cost of clarity, code size, and chip area. The code makes use of 3 processes, `leftimg`, `middle`, and `right`, arranged in a linear sequence with streams of varying sizes between them. Between each pair of adjacent processes, there are 2 streams, "top" and "bottom." Figure 4.2 shows the sorting network for the 6-process version of this code.

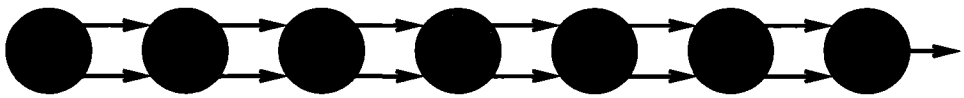


Figure 4.2: Sorting network for the pipelined mergesort.

The `leftimg` process simply reads pairs of elements from an array, and writes them to its output streams. The top output stream will receive all of the even numbered elements, while the bottom receives all of the odd numbered elements.

The `middle` process performs most of the work in the graph. Its purpose is to read pairs of sorted sequences of size 2^n from its input streams, and merge them into sequences of length 2^{n+1} on its output streams. The merged sequences are then written on output streams in an alternating fashion. For example, The first, third, and fifth sequences are written to the top stream, and the second, fourth, sixth sequences are written to the bottom stream. The parameter `chunksize` contains the value 2^n. In the case where `chunksize` is 1, the sorted input sequences are only 1 element long, and the code may be optimized to take this into account. 1 element is read from each of the input streams. They are arranged in the proper order, and written to the appropriate output stream. The body of the loop sorts 1 pair of elements into the top stream, and then sorts a second pair of elements into the bottom stream. This behavior is implemented by the code in lines 14~24. The loop could have been written to merge a single sequence, and select the output stream based on an alternating boolean flag, however by unrolling the loop to contain 2 loop bodies, the conditional can be avoided.

When chunksize is larger than 1, the process must be more complex. As before, the loop is unrolled to contain 2 loop bodies to prevent the need for conditionals to decide which output stream to write to. Lines 27~52 merge the first pair of input sequences and store the resulting sequence into the top output stream. Lines 54~79 are functionally equivalent, except that they store the merge sequence in the bottom output stream. The following discussion refers to the first of these code segments, although the same can be said for the second. The work is done by 3 consecutive while-loops. The first loop merges elements from the input streams as long as they both contain more elements. The second 2 loops append the remaining elements from the input streams onto the output stream. Since one of the 2 input streams will be empty, one of the loops will be a zero-trip loop (executing 0 times). The other loop will copy the remaining values from the nonempty input sequence to the output sequence. Again, the same behavior could have been obtained with a single loop; however, the last 2 loops can execute more efficiently than the first, so the overall performance is increased by using 3 loops. A count of the number of elements which have been dequeued from each of the input streams must be maintained, so that the process can determine when the input sequences are empty. These counts are initially stored in the variables t1_incnt1 and t1_incnt2. After the first loop finishes, the numbers of elements dequeued from the 2 sequences are stored in t2_incnt1 and t2_incnt2.

The right process works in a similar fashion as the middle process, except that it writes its output sequences to a single stream. Like middle, right has a special case which is used when chunksize is 1, and a general case when it is not.

In order to prevent deadlock, the buffer of the top input stream of a middle or right process with chunksize $= n$, must be able to hold n values. This is because the process will not begin operating until both the top and bottom stream contain at least 1 element. The preceding process will not write any values into the bottom stream until the entire top sequence has been written. The bottom stream must be able to hold at least 1 value to prevent deadlock. In order to allow the sorting network to operate at full efficiency,

the top stream must be able to hold $n + 1$ elements, and the bottom stream n. With these stream sizes, a process will never need to stall once it begins its execution, so this is the configuration which is used in these tests. These sizes were determined empirically using a simulation in C. The default size for AHAHA buffers is 16, because they can be built very efficiently. So, any buffers in the design which require less than 16 elements, are rounded up to 16 to take advantage of this feature.

4.1.1.2 Analysis

Lines 34~38 of the code (as well as 61~65 and 102~106) contain conditionals with side effecting stream operations in the branches. Such situations result in a `SPLIT- /MERGE-` based conditional being generated by the SA-C* compiler (shown in Figure 4.3).

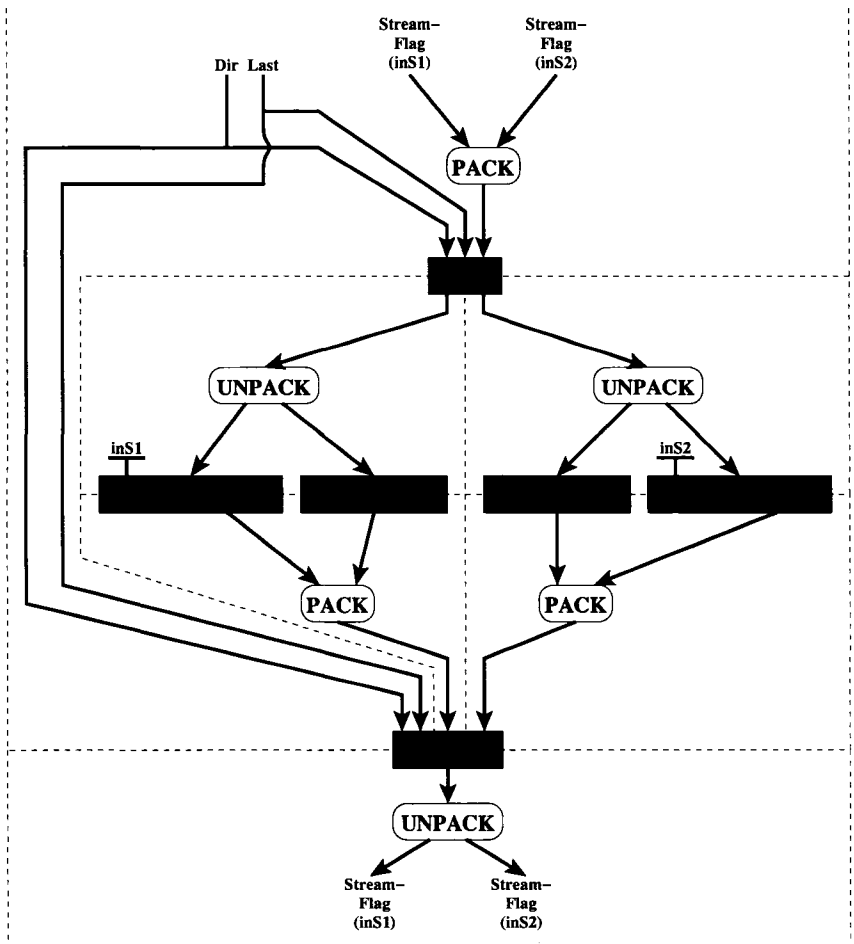


Figure 4.3: SPLIT/MERGE conditional in the pipelined mergesort.

One branch of the conditional contains a STREAM_GET node for the stream inS1, the other contains a STREAM_GET node for inS2. The input to the conditional consists of the 2 stream-serialization-flags, a "Dir" signal indicating which branch of the conditional will be executed (either 0 for the left branch, or 1 for the right), and the last-flag. Because the SPLIT and MERGE nodes can only direct a single value through the conditional branches, PACK and UNPACK nodes are used to combine the 2 serialization-flags into 1 signal. The BUFFERX nodes present inside the conditional branches were added by buffer balancing to prevent the unused serialization-flags from crossing a section boundary. Because the actual value of the get is ignored (the intended effect is to remove the smaller of the 2 values from its stream, and discard it), only the serialization-flags are produced by the conditional.

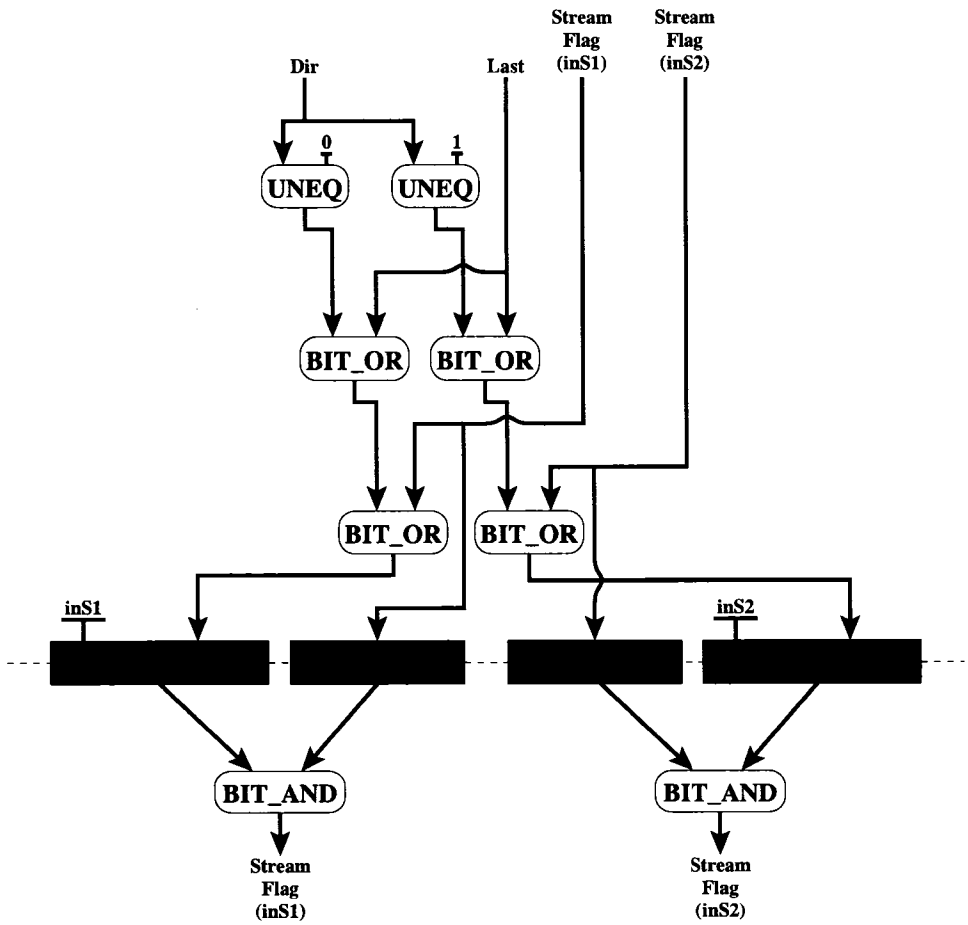


Figure 4.4: Logic-based conditional in the pipelined mergesort.

Using some simple boolean logic instead of the SPLIT and MERGE nodes (Figure 4.4) can ensure that only one of the STREAM_GET nodes will actually be executed in any iteration. When it is 1, the last-flag indicates that neither branch should execute. Likewise, when one of the stream-serialization-flags is 1, it indicates that the corresponding get operation should not execute. The left branch should not execute when the Dir signal is not 0, and the right should not execute when Dir is not 1. This logic is implemented using a UNEQ and 2 BIT_OR nodes per branch in place of the SPLIT. At the bottom of the conditional, the flag signals need to be restored to their original values since they will be used in the next iteration of the loop. The flags provided to STREAM_GET nodes have been altered, so the flags which they produce cannot be used directly. However, to preserve the serialization effect, the output flag of the STREAM_GET must be used in some fashion, otherwise the subsequent stream operations may begin before the STREAM_GET has completed. This problem is addressed with a BIT_AND node at the bottom, in place of the MERGE. The BUFFERX is added to prevent a signal from crossing a section boundary.

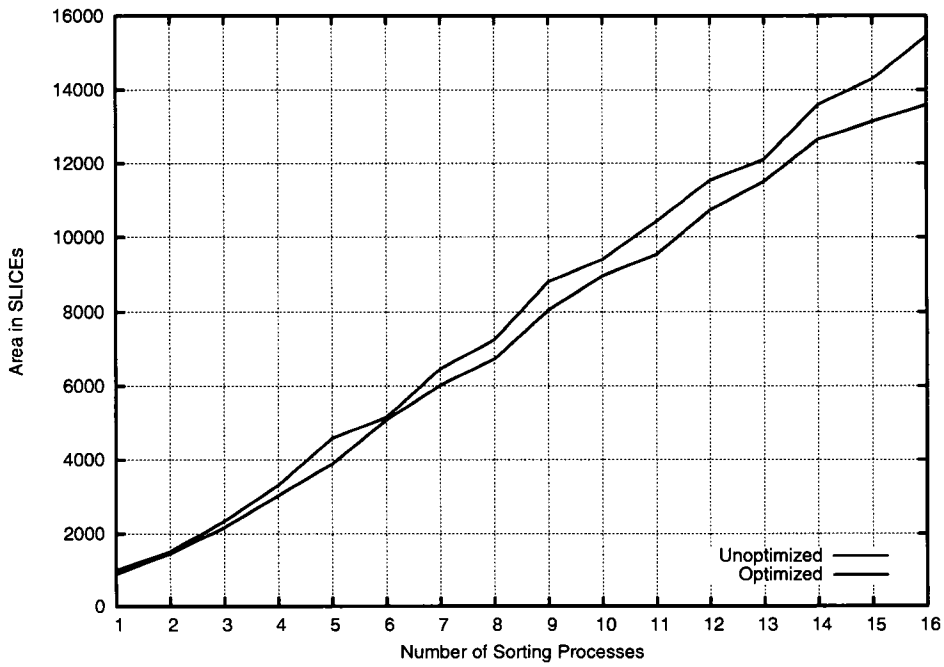


Figure 4.5: Area utilization of optimized and unoptimized pipelined mergesort.

The graph in Figure 4.5 shows area utilization for 16 different instances of the pipelined mergesort, both unoptimized and optimized. The SA-C* compiler was used to generate the unoptimized graphs, which were modified using a special case post-processor. The horizontal axis represents the number of sorting processes. Each program is given the same unsorted input file containing $2^{16} = 65536$ elements, and returns all of the elements after they have been reordered. A sorting network with n sorting processes produces sorted subsequences of length 2^n. The graph shows that the optimized version of the code uses slightly less (between 1% and 18%) area consistently. This is because the logic required for the `SPLIT` and `MERGE` nodes has been removed and replaced with a trivial amount of combinational logic, which can be mapped into a single LUT for each conditional branch. What is significantly more striking than the area savings is the impact of the optimization on runtime, shown in Figure 4.6.

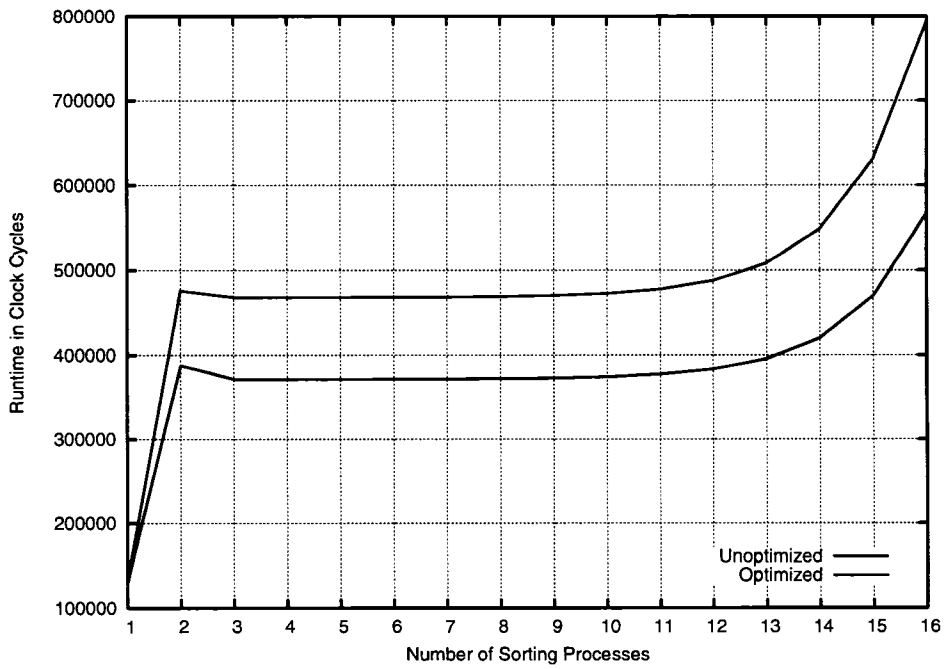


Figure 4.6: Runtime of optimized and unoptimized pipelined mergesort.

The runtime for the 1-process implementation is unaffected, since it consists of a `leftimg` process, and a single `right` process with `chunksize` = 1. Neither of these processes contain any `SPLIT/MERGE` conditionals. However, in sorting networks with

more processes, the conditionals are present, and the runtime is dramatically reduced. Because the conditional is located inside the innermost loop, its effect is significant. The time to compute the conditional has been reduced from 4 clock cycles to 2, reducing the execution time by 18% to 28%. The "Ratio" columns in Table 4.2 show the number of clock cycles divided by the number of elements in the input sequence (65536). The result shows the average number of clock cycles spent on each value. The "Improvement" column shows the percent improvement of the optimized version over the unoptimized version. The runtime of both versions increases exponentially, but the exponent has been reduced in the optimized implementation; as the program becomes larger, the benefits also increase.

	Unoptimized		Optimized		
Processes	Clocks	Ratio	Clocks	Ratio	Improvement
1	131096	2.0004	131096	2.0004	0.0000
2	475439	7.2546	387879	5.9186	18.4167
3	467265	7.1299	371509	5.6688	20.4929
4	467303	7.1305	371531	5.6691	20.4946
5	467387	7.1318	371581	5.6699	20.4982
6	467548	7.1342	371678	5.6714	20.5048
7	467869	7.1391	371871	5.6743	20.5181
8	468507	7.1488	372255	5.6802	20.5444
9	469794	7.1685	373026	5.6919	20.5980
10	472343	7.2074	374559	5.7153	20.7019
11	477467	7.2856	377633	5.7622	20.9091
12	487699	7.4417	383775	5.8559	21.3090
13	508138	7.7536	396050	6.0432	22.0586
14	549075	8.3782	420619	6.4181	23.3950
15	630891	9.6266	469737	7.1676	25.5439
16	794549	12.1239	567981	8.6667	28.5153

Table 4.2: Runtime of optimized and unoptimized pipelined mergesort.

The idea of using predicated instructions is not exclusive to stream operations, but can also be used on any side effecting clocked nodes which accept a boolean flag to suppress their operation, such as `CIRCULATE` nodes. In some cases where the nodes do not have an explicit flag for this purpose, the last-flag can be utilized in the same way as the stream-flag in the previous example. Whenever the `SPLIT` and `MERGE` nodes can

be removed and replaced with predicated logic, the optimization is always beneficial; it always reduces area and execution time. The possibility of modifying the AHAHA structure to include a "predicate-flag" on every clocked node is interesting. It would make the generation of predicated logic simpler because there would be no need to overload the functionality of stream-flags and last-flags. In the event that the node is actually unconditional, a constant value of '1' could be applied to the predicate input, and the logic supporting the predicate-flag would be dissolved by the VHDL synthesis tools, making the nodes equivalent to their current, non-predicated implementation.

4.2 Loops

One of the primary goals of the AHAHA is to implement (nested) loop-based programs on hardware. Programs without loops are possible as well, but are far less likely to achieve any appreciable gain when ported to an FPGA. Because the AHAHA focuses on loops, there are often several methods available to implement them.

4.2.1 For-Loops

In the context of the AHAHA, a for-loop is a loop which will execute a predetermined number of times. The loop is provided with the number of times to execute before any iterations are started. These loops can iterate over a simple range of integers, or over a substructure in a larger structure, such as elements from an array, or windows from an image.

4.2.1.1 Simple For-Loops

For-loops are most effectively implemented with a `TOK_GEN` node to drive the loop, and a `DONE` node to finish it. The body of the loop lies entirely between these nodes. For example, the program shown in Figure 4.7 generates the AHAHA graph in Figure 4.8.

In this example, the graph receives a runtime parameter supplied by the user (`C` from the SA-C* code), and produces all the integers from `0` to `C-1` in the array `R`. The `TOK_GEN` node in the graph gets the value of `C` (via `INPUT #1`, and causes the loop to

```
1: int32[:]  main (int32 C) {
2:   hardware () {
3:     int32 R[:]  = for int32 s in [C]
4:       return (array(s));
5:   };
6: } return (R);
```

Figure 4.7: Listing of ints.sc.

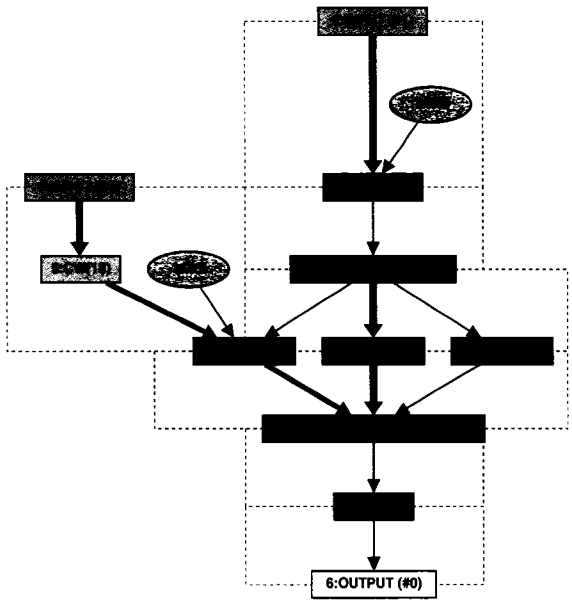


Figure 4.8: AHAHA graph of the program in Figure 4.7.

execute C times. The DONE node at the bottom of the graph identifies when the loop has finished writing to the memory (via the last-flag coming from the WRITE_WORD_MEMORY node, and signals the OUTPUT node when it has completed. The rest of the nodes in the graph implement the body of the loop. In this case, the COUNTER_CONST provides the integers being returned, the COUNTER provides the addresses in memory where to store them, the WRITE_WORD_MEMORY writes them into the memory, and the BUFFERX nodes are present for buffer balancing purposes. INPUT #3 provides the location in memory where R should be stored.

Basically, a for-loop is triggered to execute by a single value arriving at the TOK_GEN, and when it has completed, drops a single value from the DONE node. This behavior makes

it possible to create nested loops quite easily. An outer loop would simply supply the values which are currently supplied by the INPUT nodes, and expect the final token from the DONE, and then the inner loop will restart for every iteration of the outer loop. The data-driven model of AHAHA makes the outer loop stall as it waits for the DONE node to produce its value. More information about nested loops is available in Section 4.2.3.

4.2.1.2 For-Loops Using Arrays

For-loops in SA-C* most commonly iterate over an array structure. A simple form of this is illustrated by the program shown in Figure 4.9 and its corresponding AHAHA graph in Figure 4.10.

```
1: uint32[:]  main (uint32 S[:])  {
2:   hardware () {
3:     uint32 R[:]  = for s in S
4:       return (array(4294967295-s));
5:   };
6: } return (R);
```

Figure 4.9: Listing of invert_32bit.sc.

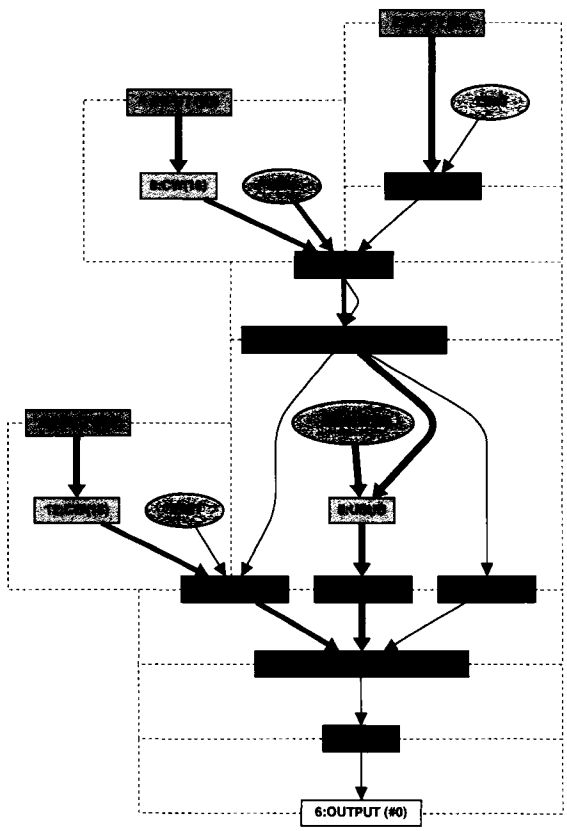


Figure 4.10: AHAHA graph of the program in Figure 4.9.

In this example, `INPUT #1` provides the number of values in the `S` array, which is the number of iterations the loop will execute. `INPUT #0` provides the address in memory

where S is stored, and INPUT #3 provides the address in memory where R should be written. The USUB node performs the computation in the loop body. As before, the TOK_GEN node drives the loop, and the DONE node identifies when the last value in R has been written.

As long as the READ_WORD_MEMORY and WRITE_WORD_MEMORY nodes are not communicating with the same memory, the graph is able to process 1 element from S (and produce 1 element of R) in every clock cycle once pipeline reaches steady state. If they are connected to the same memory, the throughput will be halved. In either scenario, the memories are busy continuously.

Note that the bit-width of the memory, and the bit-width of the elements in the arrays match, so each memory cell corresponds to an array value. This is not always the case, however. Arrays with smaller width elements can often fit more than 1 array element into each memory word. Alternatively, larger bit-width arrays may require multiple words for each element.

```
1: uint8[:]  main (uint8 S[:])  {
2:   hardware () {
3:     uint8 R[:]  = for s in S
4:       return (array(255-s));
5:   };
6: } return (R);
```

Figure 4.11: Listing of invert_8bit.sc.

When array elements are smaller than the memory width, SA-C* packs them into words to reduce bandwidth in and out of the RCS board, and to reduce memory use at runtime. The graph must unpack the values from the memory words as they arrive, before they can be passed into the loop body. Likewise, values may need to be packed into words before they are written to the memory. By modifying the program to manipulate 8-bit values instead of 32-bit values (as in Figure 4.11), the SA-C* compiler will generate the necessary modifications, as can be seen in Figure 4.12. In this example, there have

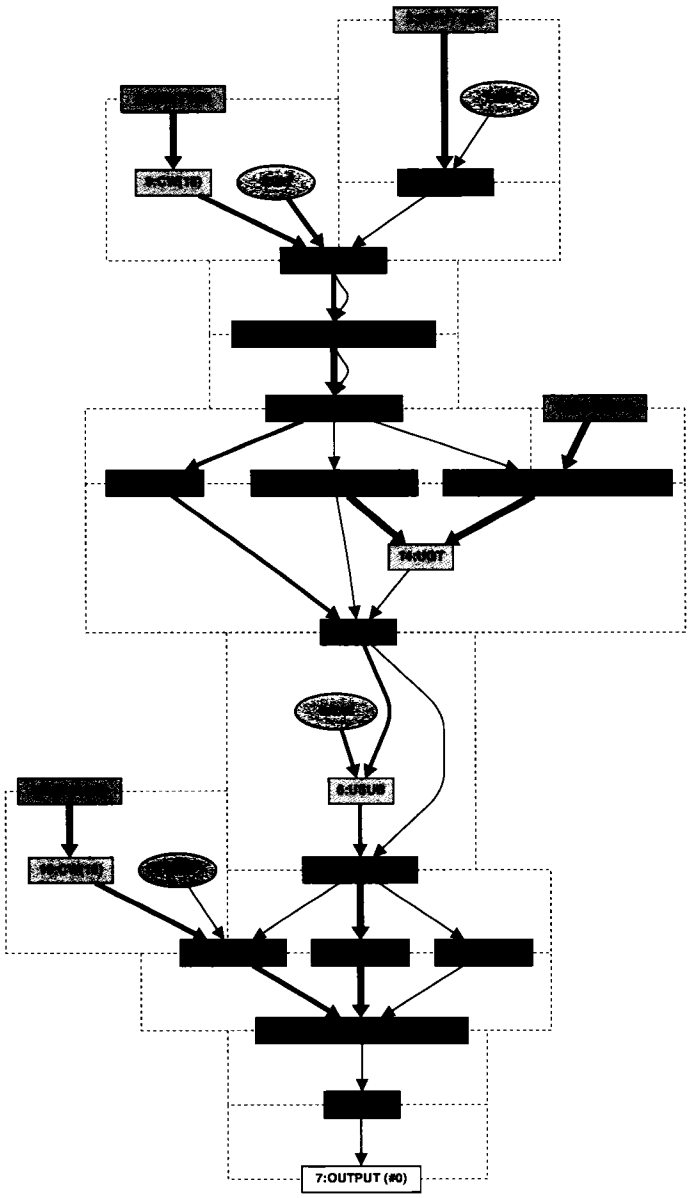


Figure 4.12: AHAHA graph of the program in Figure 4.11.

been several notable changes which are required when multiple elements are packed into a single memory word:

- A FIFO_UNPACK node has been added after the READ_WORD_MEMORY. As each word is returned from memory, it is broken into 4 separate elements and produced serially.
- Likewise, a FIFO_PACK has been added at the bottom of the loop body, to collect the 8-bit values being generated into 32-bit words before they are written to the memory by the WRITE_WORD_MEMORY.
- Because each read from memory will retrieve 4 values at once, the TOK_GEN node no longer produces a token for every element in S, but instead produces a token for every word of memory occupied by S. INPUT #2 supplies this value. INPUT #1 has not changed; it still supplies the number of elements in S.
- Even though the 8-bit values are stored in 32-bit words, there is no guarantee that the S vector will be a multiple of 4 in length. If it is not an even multiple, the last word of the memory which is read will not contain 4 valid values which need to be removed from the sequence. This is done by the MASK node. The COUNTER_CONST node in this example begins counting at 1 and increments by 1. Its function is to count the elements in S as they are produced by the FIFO_PACK. The REPLICATOR_FLAG_EXTRA simply repeats the length of S each time the loop body executes. The UGT node identifies when the value being produced by the FIFO_PACK is beyond the extents of S, so that it can be removed by the MASK node.

Like the previous, 32-bit example, this version is able to process 1 value of S per clock cycle, but the READ_WORD_MEMORY and WRITE_WORD_MEMORY nodes only access memory once every 4 clock cycles. In this program, the execution will not be slowed down if the READ_WORD_MEMORY and WRITE_WORD_MEMORY nodes are accessing the same memory. The memory bandwidth is not saturated.

In this particular program, the input size and the output size are identical. If some of the elements of the last word of memory read are invalid, then the corresponding values in

the last word of output memory will also be invalid, so there is no actual need to handle them in a special case way. The MASK, COUNTER_CONST and REPLICATOR_FLAG_EXTRA nodes are not actually required in this example. The SA-C* compiler generates them because it always generates general case code, and does not always optimize the extra nodes away. Removing these extra nodes makes it possible to optimize the behavior of the graph by partially unrolling the loop 4 times. This will replicate the loop body 4 times, and perform them in parallel. The FIFO_UNPACK and FIFO_UNPACK nodes can be replaced with much simpler UNPACK and PACK nodes. An optimized version of this program can be seen in Figure 4.13. Because the FIFO_UNPACK and FIFO_UNPACK nodes were the bottleneck of the graph, these optimizations increase the throughput of the

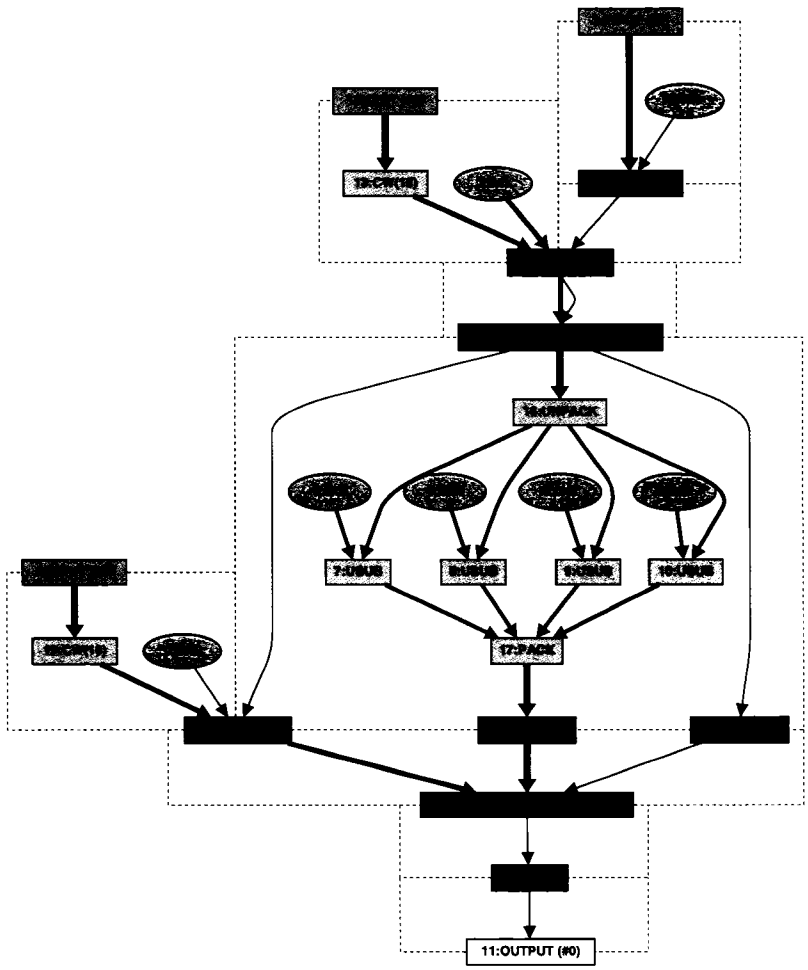


Figure 4.13: Optimized AHAHA graph of the program in Figure 4.11.

graph 4 times.[2] The SA-C* compiler does not currently provide this optimization, but the AHAHA easily supports it.

4.2.1.3 For-Loops With Simple Loop-Carried Dependencies

All of the examples shown so far have no loop-carried dependencies, which means there are no values being used in the loop body which were computed in a previous body of the same loop. Since each iteration of the loop is independent, multiple loop bodies may be executed concurrently. This allows the loops to be completely pipelined, so that they can complete 1 loop body per clock cycle. Loop-carried dependencies are supported using a `CIRCULATE` node (or one of its variants). In some cases, a loop-carried dependence can cause a slowdown of the loop. The number of clock cycles required to compute the value which is carried into the next iteration defines the amount of slowdown. In the event that it can be computed with a combinational circuit, no slowdown occurs. This is the case for the simple CBC encryption and decryption codes shown in Figures 4.14 and 4.15, respectively. For more information on CBC encryption, please see [39]. The AHAHA graphs for these codes can be found in Figures 4.16 and 4.17, respectively.

```
1: uint32[:]  main (uint32 A[:], uint32 key) {
2:   hardware () {
3:     uint32 init = 0xdeadbeef;
4:     uint32 R[:]  = for plaintext in A {
5:       uint32 xored = (bits32)plaintext ^ (bits32)init;
6:       uint32 ciphertext = xored + key;
7:       next init = ciphertext;
8:     } return (array(ciphertext));
9:   };
10: } return (R);
```

Figure 4.14: Listing of encrypt.sc.

In the SA-C* code for encryption, the value to be used for `init` in the next iteration depends upon the `init` from the current iteration (as well as the `plaintext`). Because

[2]This analysis assumes that the input and output are stored in different memories. If they are not, the speedup will only be 2×.

```
1: uint32[:]  main (uint32 A[:], uint32 key) {
2:   hardware () {
3:     uint32 init = 0xdeadbeef;
4:     uint32 R[:]  = for cipertext in A {
5:       uint32 xored = cipertext - key;
6:       uint32 plaintext = (bits32)xored ^ (bits32)init;
7:       next init = cipertext;
8:     } return (array(plaintext));
9:   };
10: } return (R);
```

Figure 4.15: Listing of decrypt.sc.

`init` depends on itself, a circularity in the graph is created, involving the `BIT_EOR` and `UADD` nodes. The `CIRCULATE` node used to implement this behavior is initialized with an arbitrary 32-bit constant (`0xdeadbeef`). Because it is a constant, the `_CONST` variant of the `CIRCULATE` is used. Because the source and target sections of the cycle are the same, the `_LOCAL` variant is used. The resulting node is the `CIRCULATE_LOCAL_CONST`. Whenever a graph circularity can be implemented using a `_LOCAL` variant of a `CIRCULATE` node, no slowdown will result from it.

At first glance, the SA-C* code for the decryption algorithm seems to have circularity in it as well, since the **next** keyword is used to carry a value from the current iteration into the next. However, the value being assigned to the nextified variable (`init`) is not dependent upon its own previous value. For this reason, there is no circularity in the resulting graph. Nevertheless, a `CIRCULATE_CONST` node is used in the implementation. Because there is no real circularity, no slowdown will occur, regardless of the time required to compute the "nextified" value. The computation will be fully pipelined.

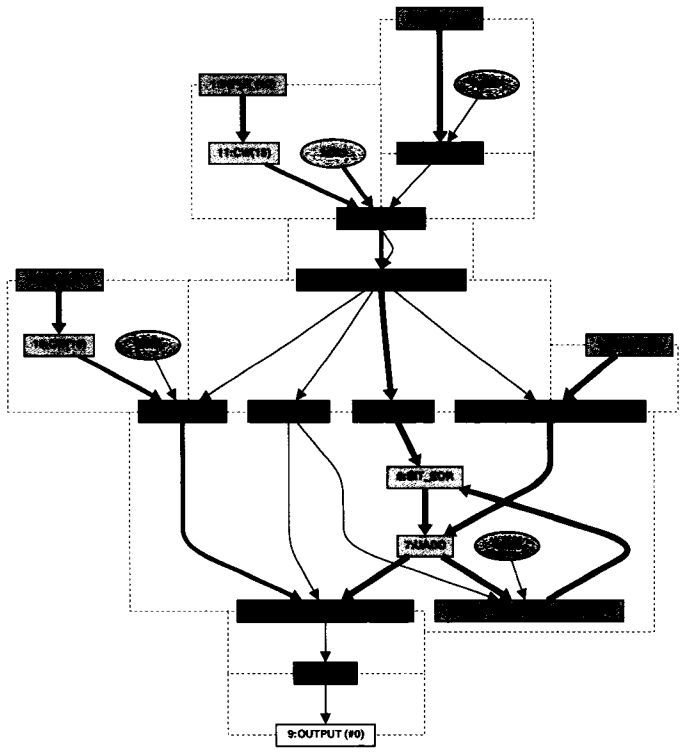


Figure 4.16: AHAHA graph of the encryption program in Figure 4.14.

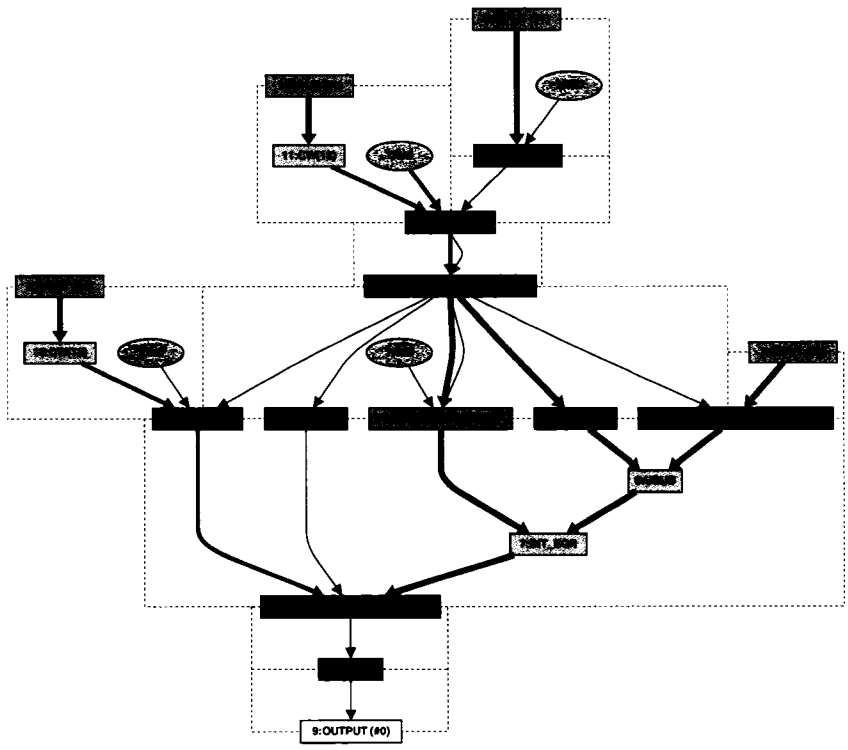


Figure 4.17: AHAHA graph of the decryption program in Figure 4.15.

4.2.1.4 Loop-Carried Dependencies With Large Latency

When a circularity in the AHAHA graph contains a large computation which requires many clock cycles of latency to compute, it becomes very inefficient.

```
1: uint32[:]  main (uint32 A[:], uint32 key, uint32 D[256]) {
2:   hardware () {
3:     uint32 init = 0xdeadbeef;
4:     uint32 R[:]  =
5:     // PRAGMA(input(A memory 1,D memory 2),output(memory 3))
6:     for plaintext in A {
7:       uint32 thiskey = (bits32)D[(uint8)init] ^ (bits32)key;
8:       uint32 xored = (bits32)plaintext ^ (bits32)init;
9:       uint32 ciphertext = xored + thiskey;
10:      next init = ciphertext;
11:    } return (array(ciphertext));
12:  };
13: } return (R);
```

Figure 4.18: Listing of encrypt2.sc.

By making the circularity in the encryption program require multiple clock cycles, (Figure 4.18), the loop is slowed dramatically. The corresponding decryption program is shown in Figure 4.19. The AHAHA graphs for these 2 codes are too large to clearly depict here, and so have been omitted.

```
1: uint32[:]  main (uint32 A[:], uint32 key, uint32 D[256]) {
2:   hardware () {
3:     uint32 init = 0xdeadbeef;
4:     uint32 R[:]  =
5:     // PRAGMA(input(A memory 1,D memory 2),output(memory 3))
6:     for cipertext in A {
7:       uint32 thiskey = (bits32)D[(uint8)init] ^ (bits32)key;
8:       uint32 xored = cipertext - thiskey;
9:       uint32 plaintext = (bits32)xored ^ (bits32)init;
10:      next init = cipertext;
11:    } return (array(plaintext));
12:  };
13: } return (R);
```

Figure 4.19: Listing of decrypt2.sc.

Because the modified decryption algorithm still contains no true circularity, there is no loop slowdown. The graph will be pipelined, and the hardware will still be able to produce 1 decrypted value per clock cycle.[3] In the new encryption routine, however, the first value read from D is used to compute which value from D will be read in the next iteration, so pipelining is not possible. The loop is only able to produce a value every 6 clock cycles. The CIRCULATE node produces its first (initial) value in clock cycle x, the request to the memory occurs in clock cycle $x + 1$, and the READ_WORD_MEMORY node produces its value in clock cycle $n + 5$. In clock cycle $n + 6$, the CIRCULATE node produces its next value, and the pattern repeats.[4] Further speedup could be obtained by implementing the D array as an on-chip lookup table using a ROMREF node.[5] This would remove the latency all together, and allow the modified algorithm to execute at the same rate as the original encryption algorithm.

4.2.2 While-Loops

Essentially, a while-loop is a for-loop in which the TOK_GEN node has been replaced with a CIRCULATE node plus some combinational logic. They are simply loops with a loop-carried dependence which is used to determine when the loop should terminate. Just as in for-loops, the loop-carried dependence can cause the loop to slow down. In each iteration, the loop test needs to be evaluated before the loop body is executed. In most circumstances, the test of the loop uses a nextified value which was defined by the loop body of the previous iteration. While-loops are, in general, less efficient than for-loops, since their loop test usually contains a loop-carried dependence, but are fully supported by AHAHA, and SA-C*. A simple while-loop-based program is shown in Figure 4.20.

---

[3]This assumes that the A array and the D arrays are stored in separate memories. If they are stored in the same memory, the loop would produce 1 value every 2 clock cycles due to memory contention.

[4]The loop can be made to run faster by storing D in a BlockRAM instead of an off-chip memory. This would remove 3 clock cycles of memory latency from the loop, causing it to produce an encrypted value every 3 cycles instead of 6.

[5]This speedup is only possible if the values in D are a constant at compile time.

```
1: uint32 main (uint16 start, uint16 perm[65536]) {
2:   hardware () {
3:     uint32 count = 0;
4:     uint32 hops = while(start != 0 &&
5:                         count <= 65536) {
6:       next count = count + 1;
7:       next start = perm[start];
8:     } return(final(count));
9:   };
10: } return (hops);
```

Figure 4.20: Listing of while.sc.

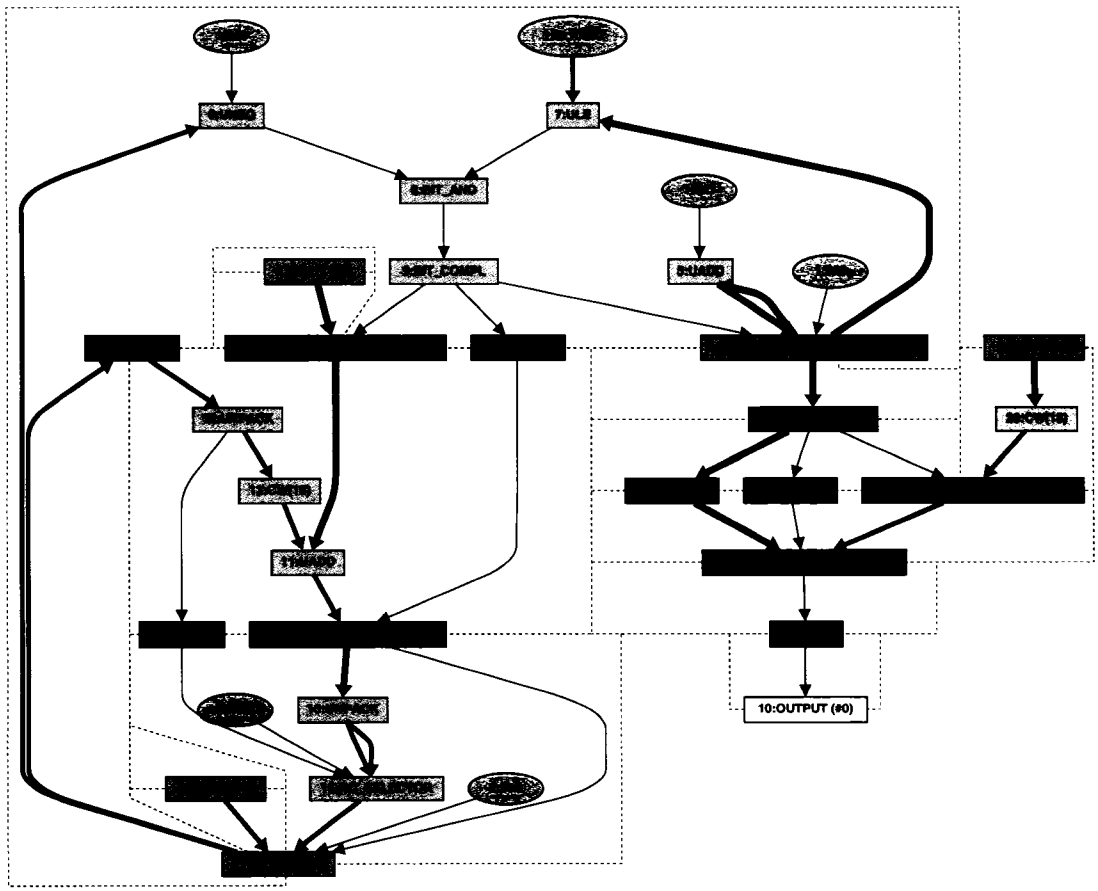


Figure 4.21: AHAHA graph of the while-loop program in Figure 4.20.

The AHAHA graph which implements it is shown in Figure 4.21.

The BIT_COMPL produces the flag which would have been produced by a TOK_GEN in a for-loop. The large section at the top of the graph has the 2 CIRCULATE nodes as its producers, which begin by dropping their initial values. The test of the while-loop is evaluated with combinational logic, and provides a "last-flag" to the sections on the left and right sides. The sections on the left will perform a memory access. The READ_WORD_MEMORY, UNPACK, and RC_SELECTOR implement the 16-bit array access. The sections on the right are triggered when the loop is finished. They cause the final value to be written to the memory, and then signal the host that the computation is complete via the OUTPUT node.

4.2.3 Loop Nests

Loop nests are completely supported by the AHAHA model. The outer loop supplies values to trigger the inner loop, and waits for any values it produces. For example, consider the program in Figure 4.22.

```
1: uint32[:] main (uint32 A[:], uint32 B[:]) {
2:   hardware () {
3:     uint32 R[:] = for a in A dot b in B {
4:       uint32 t = 1;
5:       uint32 s = while(b!=0) {
6:         next t = t * a;
7:         next b = b - 1;
8:       } return(final(t));
9:     } return (array(s));
10:  };
11: } return (R);
```

Figure 4.22: Listing of exp.sc.

The example program computes and returns the sequence

$$A[0]^{B[0]}, A[1]^{B[1]}, A[2]^{B[2]}, \cdots, A[n-2]^{B[n-2]}, A[n-1]^{B[n-1]}$$

Because exponentiation is not natively supported by the hardware, it is implemented using multiplications in a while-loop.[6] The AHAHA graph of this example can be found in Figure 4.23.

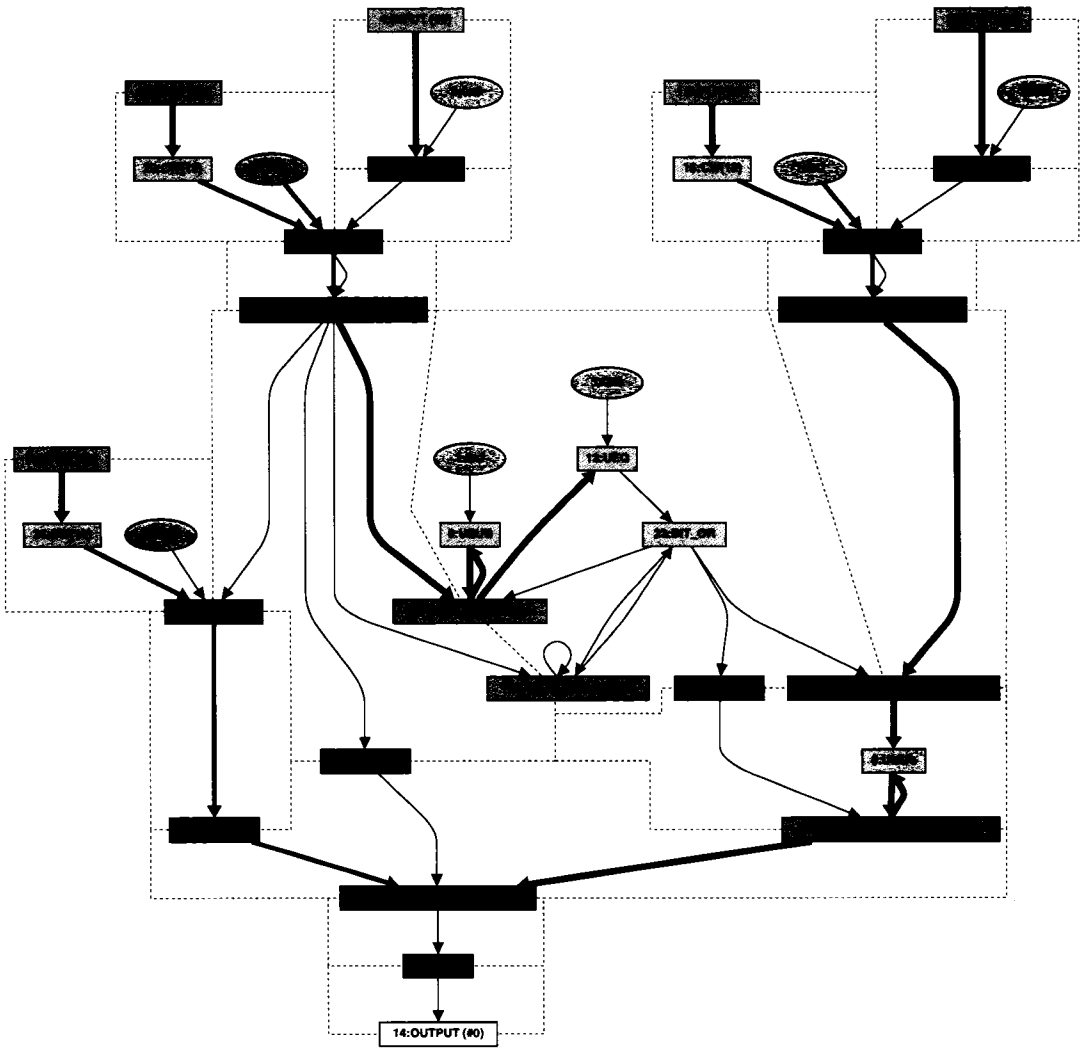


Figure 4.23: AHAHA graph of the nested loop program in Figure 4.22.

The graph contains 2 groups of sections at the top left and top right of the figure which implement the generators for the outer loop. The left portion generates the sequence of B values, while the right loop generates the sequence of A values. The oddly shaped

[6]Note that this is not a very efficient exponentiation algorithm. The method used here executes in linear time, where alternative algorithms exist which execute in logarithmic time.

section in the center, the section immediately below it, and the one to its right implement the inner while-loop. The remaining sections collect the values returned by the inner loop and store them in the memory.

```
1: uint32[:]  main (uint32 A[:], uint32 B[:])  {
2:   hardware () {
3:     uint32 R[:]  = for a in A dot b in B {
4:       uint32 t = 1;
5:       // Scalar gen in SA-C* can't handle uint32
6:       uint32 s = for _ in [(uint31)b] {
7:         next t = t * a;
8:       } return(final(t));
9:     } return (array(s));
10:  };
11: } return (R);
```

Figure 4.24: Listing of exp_for.sc.

There is no algorithmic reason that the inner while-loop could not have been implemented using a for-loop with a SA-C* scalar generator as well (as shown in Figure 4.24). The while loop version was chosen for 2 reasons. First, the while-loop better illustrates the flexibility of nested loops. Second, the scalar generators in SA-C* have a limitation which prevents them from iterating over `uint32` data types. They are limited to `uint31` or `int32` types. This limitation is imposed by the SA-C* compiler itself due to its internal implementation. No such restriction exists in the AHAHA framework. In fact, the AHAHA model imposes no bit-width limitations of any kind, and is restricted only by what can fit on the target hardware. In cases where a while-loop can be trivially transformed into a for-loop, the SA-C* compiler often generates AHAHA graphs which are identical to their for-loop counterparts.

4.3 Streams

One of the new features introduced in SA-C* is the concept of process networks connected via streams. The model used in `SA-C*` is similar to the Kahn process model[23, 24], except

that the buffers within the streams are finite. The AHAHA supports stream programs with a collection of stream processing nodes.

The streams themselves are implemented using SRL16E-based FIFOs, or BlockRAM-based FIFOs depending on the requested depth of the stream. Small streams (less than 512 elements) are implemented using SRL16Es, larger streams use BlockRAM. SRL16E streams have a latency of 1 clock cycle, and continually present the "next" element in the stream on its outputs, and fetch the next value once it has been dequeued. BlockRAMs, on the other hand, only retrieve a value from the memory when it has been requested. Special care has been taken to make the BlockRAM streams behave more like the SRL16E streams. Values are pre-fetched from the BlockRAM and stored in a small buffer so that they can be presented on the outputs continually. When the BlockRAM is empty, values are enqueued directly into the buffer to provide 1 clock cycle of latency. There is only 1 specific circumstance where the BlockRAM stream behavior differs from that of the SRL16E stream. When the stream is being continually dequeued, and a value is enqueued at the same time that the last value from BlockRAM is pre-fetched, there will be a single wasted clock cycle. This circumstance is incredibly rare, however, and there have been no instances of its occurring in any tested applications.

Most stream processes are created using a while-loop which continues to operate until its input streams have been closed. Figure 4.25 shows a simple stream-based program.

Unfortunately, stream codes give rise to a large number of nodes; this example alone contains 56 nodes. The AHAHA graph for this code is shown in Figure 4.26.

The size of the graph makes it difficult to depict it clearly here. However, it is possible to identify some of the more significant structures. The top of the graph contains the nodes which declare the streams. There are 4 streams in the graph which correspond to `ivals`, `ovals`, `temp1`, and `temp2` from the source code. The `ivals` stream is declared as an input stream (`RC_INPUT_SCALAR_STREAM`) since it originates on the host, and is transferred into the graph. The `ovals` stream is declared as an output stream (`RC_OUTPUT_SCALAR_STREAM`) since it originates in the hardware and is trans-

```
1: export main;
2:
3: process proc_a(in stream uint8 i, out stream uint8 o) {
4:   while(i) {
5:     put(get(i) * 3, o);
6:   };
7: };
8:
9: process proc_b(in stream uint8 i, out stream uint8 o) {
10:  while(i) {
11:    uint8 v = get(i);
12:    bool iseven = (((bits8)v & 0x01)==0x00);
13:    put(v/2,iseven,o);
14:  };
15: };
16:
17: process proc_c(in stream uint8 i, out stream uint8 o) {
18:  while(i) {
19:    uint8 v = get(i);
20:    bool isodd = (((bits8)v & 0x01)==0x01);
21:    put(v,o);
22:    put(v,isodd,o);
23:  };
24: };
25:
26: process network main(in stream uint8 ivals,
27:                      out stream uint8 ovals) {
28:  stream uint8 temp1;
29:  stream uint8 temp2;
30:  hardware() {
31:    instantiate proc_a (ivals, temp1);
32:    instantiate proc_b (temp1, temp2);
33:    instantiate proc_c (temp2, ovals);
34:  };
35: };
```

Figure 4.25: Listing of stream.sc.

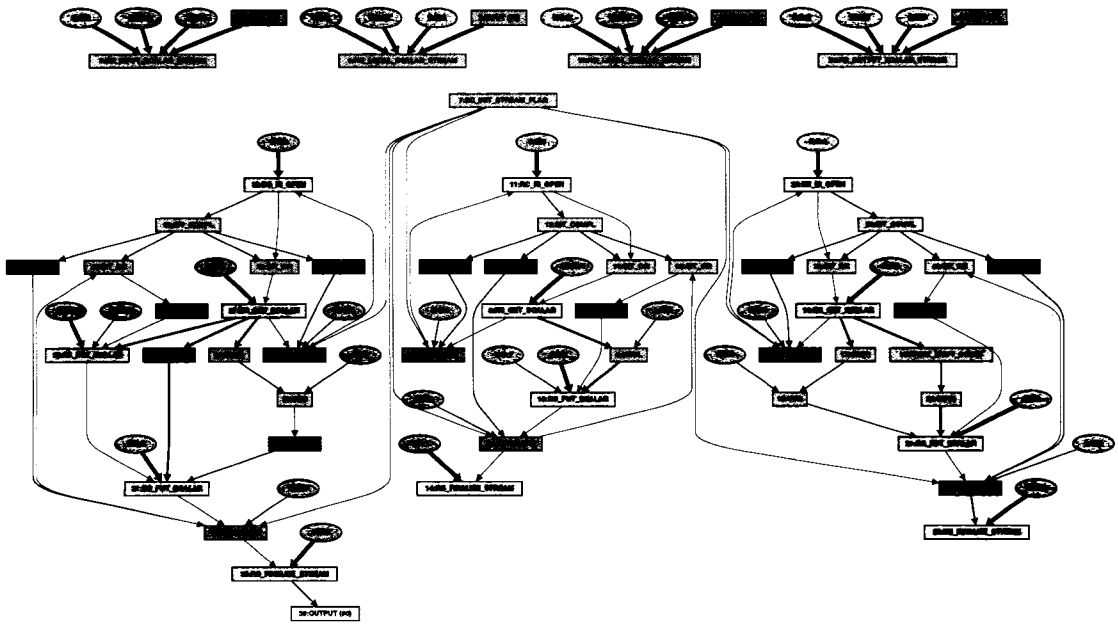


Figure 4.26: AHAHA graph of the stream program in Figure 4.25.

ferred to the host. The `temp1`, and `temp2` streams are both declared as local streams (`RC_LOCAL_SCALAR_STREAM`) since they are produced and consumed within the hardware.

Immediately below the stream declaration nodes is a (`RC_INIT_STREAM_FLAG`) node which supplies the stream-serialization-flags to all of the serialization loops in the graph. Each stream has 2 serialization loops, 1 to serialize the nodes which operate on the input side of the FIFO, and 1 to serialize the nodes operating on the output side. The edges in the graph, which carry the stream-serialization-flags, have been colored red. The stream operations which utilize the flags have been colored orange. In this example, there are 6 serialization loops: 1 each for the output sides of `ivals`, `temp1`, and `temp2`, and 1 each for the input sides of `ovals`, `temp1`, and `temp2`.

Visually, the rest of the graph is clustered into 3 regions. These regions correspond to the 3 processes in the code. The center region implements `proc_a`, the right implements `proc_b`, and the left implements `proc_c`. Each process has an `RC_IS_OPEN` node at the top of its region which drives the while-loop for that process. Each process continues while the stream it is consuming is still open. The 2 `CIRCULATE` nodes in each process provide the circularity for the stream-serialization-flags. Each process has at the bottom of its

region an RC_FINALIZE_STREAM node which closes the output stream when the process terminates. The RC_FINALIZE_STREAM for the proc_c process is attached to the OUTPUT node to signal the host when the last (only) output stream is closed.

Chapter 5

AHAHA Analysis

The major contribution of the AHAHA is its handshaking model. This chapter provides analysis of the overhead required for the handshaking logic, and relates it to other methods of scheduling. The overhead is negligible when weighed against its benefits (discussed later in section 5.2). Analysis of the process and stream model supported by the AHAHA, and the SA-C* compiler is discussed at the end of the chapter.

5.1 Handshaking Overhead

Intuitively, the handshaking in the AHAHA implementation is quite simple, consisting of a single multi-input AND-gate per section, and so uses very little chip area, and is not likely to reduce clock frequency. This section provides analysis of the overhead associated with the AHAHA handshaking protocol.

5.1.1 Test Program Selection Criteria

The amount of AHAHA handshaking logic present in the implementation of any graph is proportional to the number of sections in it. The handshaking is entirely independent of the computation being performed by the unclocked nodes inside each section, so the actual behavior of the program is irrelevant. In order to investigate the handshaking overhead, a program whose size (in terms of the number of sections) is parameterizable is preferable. The more significant the change in the section count is, the more exaggerated the handshaking overhead will be. The pipelined mergesort (shown in Table 4.1),

generates AHAHA graphs with a large number of sections (as seen in Table 5.1), and is

Processes	Sections	Increase	Nodes	Increase
1	31	-N/A-	56	-N/A-
2	81	50	170	114
3	166	85	361	191
4	251	85	552	191
5	336	85	743	191
6	421	85	934	191
7	506	85	1125	191
8	591	85	1316	191
9	676	85	1507	191
10	761	85	1698	191
11	846	85	1889	191
12	931	85	2080	191
13	1016	85	2271	191
14	1101	85	2462	191
15	1186	85	2653	191
16	1271	85	2844	191

Table 5.1: Sections in pipelined mergesort AHAHA graphs.

parameterizable. Each new process in the sorting network contributes 85 sections, each consisting of 191 nodes. No other program which was used in testing has as sharp of an increase in sections as the program size increases. Additionally, the computational logic within the sorting processes is limited exclusively to 8-bit comparisons, which are relatively small. This small computational body will help to exaggerate the handshaking effects by reducing the rate of program size increase due to computational logic.

These features of the pipelined mergesort come from its SA-C* implementation. It was written to be as efficient as possible in terms of runtime, and in doing so, the code size and area have been sacrificed. For example, the main loop contains 6 loops within it (lines 26, 44, 49, 57, 71, and 76). At any time, only 1 of these loops is active, and the other 5 are idle. The program could have been written with a single inner while-loop instead of 6, but the runtime would have been degraded due to the need for additional conditionals. Each of the 6 loops generate independent hardware, with their own sections and handshaking logic. To illustrate the complexity of this program, the AHAHA graph for the 16-process version of the code is shown in Figure 5.1. It contains 2844 nodes,

Figure 5.1: AHAHA graph of the mergesort program in Figure 4.1.

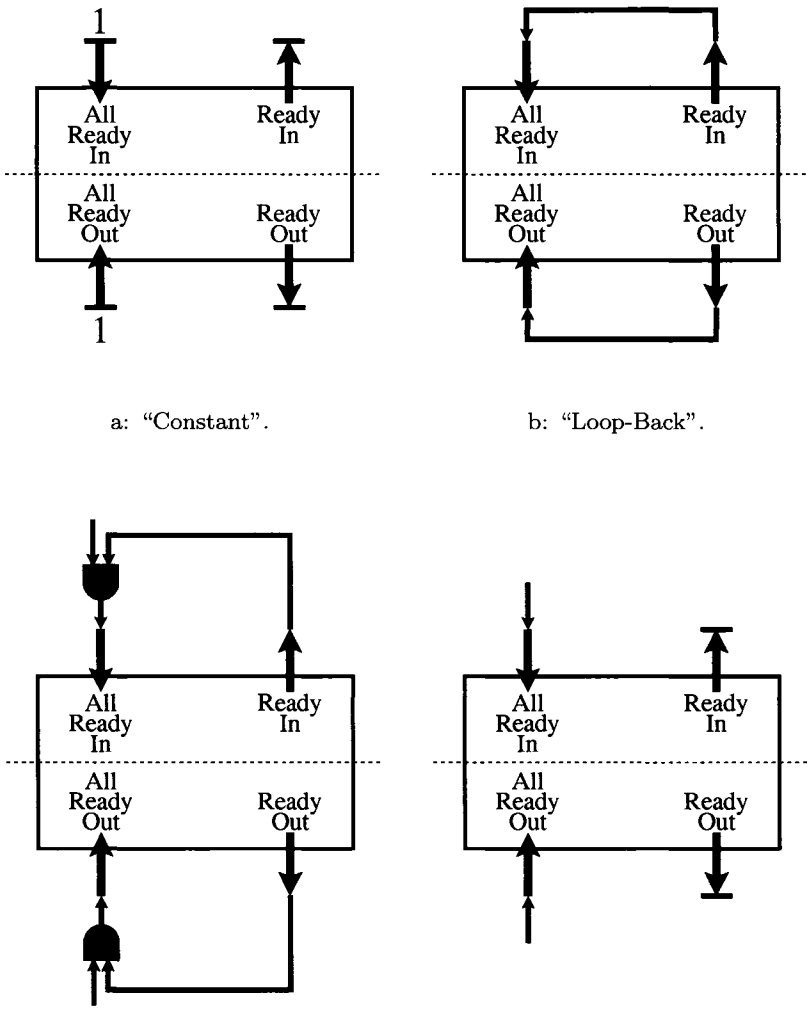
divided into 1271 sections. Although the graph is far too small to see clearly, one can still make out the row of stream declaration nodes on the top left. The rest of the graph is visually divided into the 16 processes, and the last process is approximately half the size of the rest. The last process is the `right` process, which contains approximately half as much logic as the `middle` process. The logic on the top left corner below the stream declarations corresponds to the `leftimg` process.

5.1.2 Handshaking Area

To quantify the amount of area used by the handshaking (as opposed to the computation itself), programs can be generated without the handshaking logic included. The resulting programs will not be capable of executing, since the nodes will not fire in the correct sequence, but are useful to determine handshaking area requirements. There are several ways of removing handshaking logic, shown in Figures 5.2 a, 5.2 b, 5.2 c, and 5.2 d. The first, (the "Constant" method) completely ignores **Ready_** signal and supplies a constant 1 to **AllReady_**. The second method ("Loop-Back") uses the **Ready_** signal and supplies it as input to **AllReady_**. The third ("And") includes an additional external signal, and the fourth ("External") uses the external signal exclusively, ignoring the **Ready_** signal.

The first method ("Constant") is the simplest: all clusters of all nodes always fire. This method might be the first that comes to mind, but has some serious deficiencies which make it inappropriate for handshaking analysis:

1. Clocked nodes are built with the assumption that their clusters will not fire when they are not ready. This means that they require the **AllReady_** signals to be low when their corresponding **Ready_** is low. Violating this assumption will cause the behavior of the node to differ from what was intended. Since the graphs without



a: "Constant". b: "Loop-Back".

c: "And". d: "External".

Figure 5.2: AHAHA Handshaking removal methods.

handshaking will not be executed anyway, this does not seem like an important consideration. However, this issue compounds the other issues considerably.

2. Because the `Ready_` signal is ignored, any logic within the node which exists to generate that signal will be dissolved. If a cluster of a node fires when it is not ready, the node will malfunction. Hypothetical AHAHA nodes which do not rely on handshaking would likely still need to contain this logic, to prevent errors. The dissolving of this logic will make the node artificially small.

3. The synthesis tool (`XST`) is quite aggressive in its optimizations. When a constant value is supplied to an input of a VHDL entity, `XST` will take it into account when synthesizing the node. This can lead to an artificial reduction in the area of the node. For example, many nodes are implemented using synthesizable behavioral VHDL processes. Such processes may contain if-then-else statements. When a conditional is based upon a constant `true`, `XST` will dissolve the logic present in the "else" portion of the conditional, leaving only the "then" portion. Dissolving of the "then" clause will occur when a constant `false` is supplied. When the expression is not a constant, both branches must be taken into account.

Despite its shortcomings, the "Constant" method of handshaking removal was implemented, and its results are shown in Figure 5.3 for completeness. As expected, the

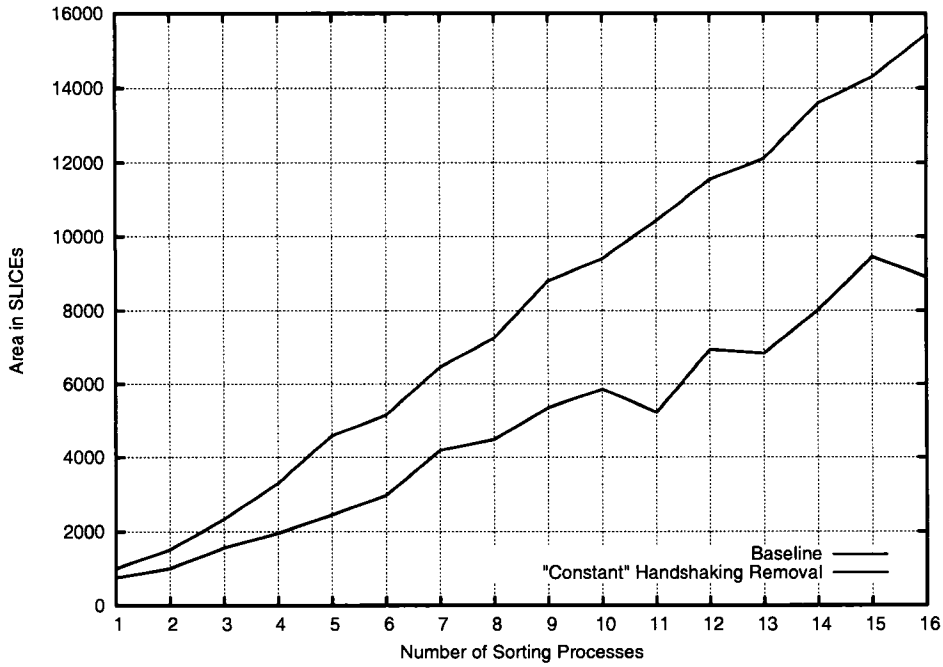


Figure 5.3: Area of pipelined mergesort using "Constant" handshaking removal.

area is sharply reduced, but much of the logic in the nodes has been over-optimized by `XST`. This method, while simple, does not give an accurate depiction of the handshaking overhead.

The second method ("Loop-Back") causes each cluster of the node to fire when it has determined on its own when it is ready. This ensures that the logical behavior of the node will be unaffected. The results of the Loop-Back handshaking removal method is shown in Figure 5.4. The results show a modest reduction in area, which is consistent with the

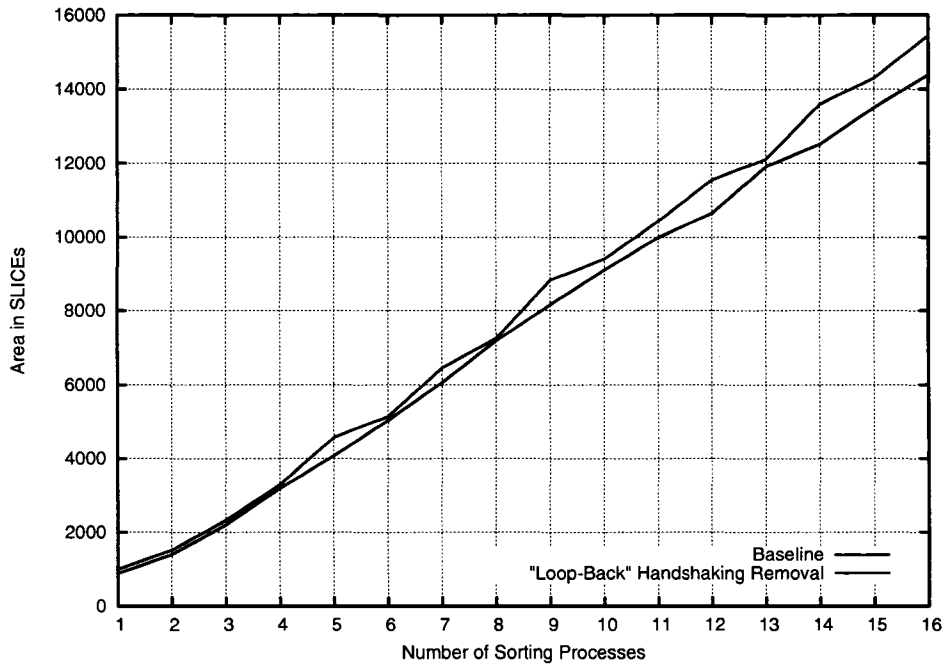


Figure 5.4: Area of pipelined mergesort using "Loop-Back" handshaking removal.

claim that the handshaking logic within the sections is inconsequential compared with the computational logic. In this method, XST may still be able to optimize the node logic somewhat since it can ignore the possibility that a cluster may stall when it is ready to fire. However, this method is likely the most accurate for ascertaining the typical overhead for handshaking in the AHAHA graphs. Using this method, we can estimate that the handshaking accounts for about $\frac{1}{20}th$ of the design area.

The third method of handshaking removal ("And") causes each cluster to fire when it is ready, and an external signal indicates that it should fire. For the purposes of testing, the external signal used here is supplied by an otherwise unused bit of the register file data from the host. This method would be used when some external device (such as a finite state machine) is able to determine when each section should fire, but each node

is able to refuse to fire if it is not ready. Presumably, the external device would have some form of feedback to recognize when a refusal takes place. The area which would be used to implement the external device is not included, but it should be noted that such a device would be quite substantial. It would require at least 1 output signal for each section in the graph, and an input for each cluster. This method of handshaking removal has the advantage that no over-optimization of the node is possible because the node must be prepared to fire at any time. However this method also has the disadvantage that a single 2-input AND gate is created for each cluster of each node, which results in additional overhead. Since the overhead is proportional to the number of *clusters* in the graph, and the handshaking is proportional to the number of *sections*, and since each section contains at least 2 clusters (possibly more), the overhead of this method is actually greater than the handshaking overhead. This method provides an absolute upper-bound on the size of the logic required when being controlled by an external timing device. The results of this method are shown in Figure 5.5. As the graph shows, this

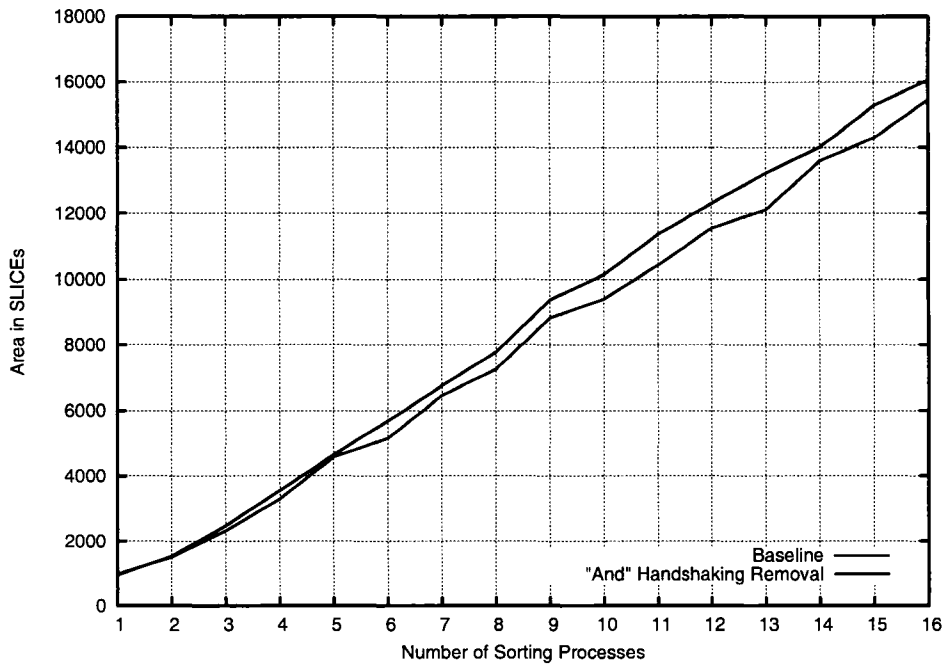


Figure 5.5: Area of pipelined mergesort using "And" handshaking removal.

method results in an increase in area. From this we can conclude that the area being

used in the AHAHA handshaking model is less than the area required to implement a 2-input AND gate for each cluster.

The final method ("External") uses an external signal to determine when each cluster should fire, but does not take into account whether or not the cluster is ready. This method would be used in a static timing solution. An external device which knows the scheduling and behavior of the nodes, instructs the clusters to fire in some preset pattern, with no feedback from the nodes. As before, the external signal is supplied by the register file for testing, and the area which would be used to implement the timing device has not been taken into account. The control device would be nearly as large as the one mentioned earlier, with a slight reduction in area due to the lack of feedback from the cluster. This method provides the absolute lower-bound on the area required to generate the logic of the graph. Remember, the "Constant" method actually causes much of the logic in the graph to be dissolved along with the handshaking, so its curve actually falls below this lower-bound. In the "External" method, all logic generating the `Ready_` signals is dissolved, leaving only the computational body of the nodes with no feedback logic. The graph shows a large reduction in area due primarily to the removal of all the feedback logic from each node.

A composite graph (Figure 5.7) shows how all of these methods compare. The "Constant" method has been ignored, due to its flaws. From the graph, it is clear that the dominating factor in every case (discounting the flawed "Constant" method) is the size of the program being implemented. The area used for handshaking is linear with the number of sections in the graph. The type of handshaking removal has only a small effect on the slope of the line. Additionally, the baseline lies between the "External" method and the "And" method, both of which would require additional logic to function properly. As stated earlier, the program used in this analysis was selected deliberately to exaggerate the handshaking overhead, and the overhead was found to be quite small. It can be concluded from this that the handshaking logic used in the AHAHA graphs

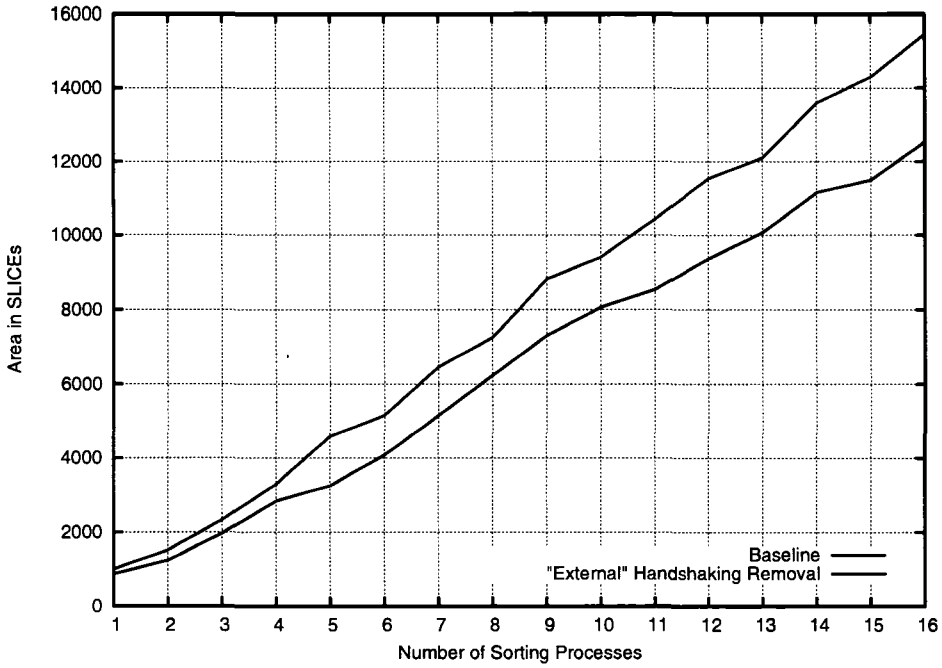


Figure 5.6: Area of pipelined mergesort using "External" handshaking removal.

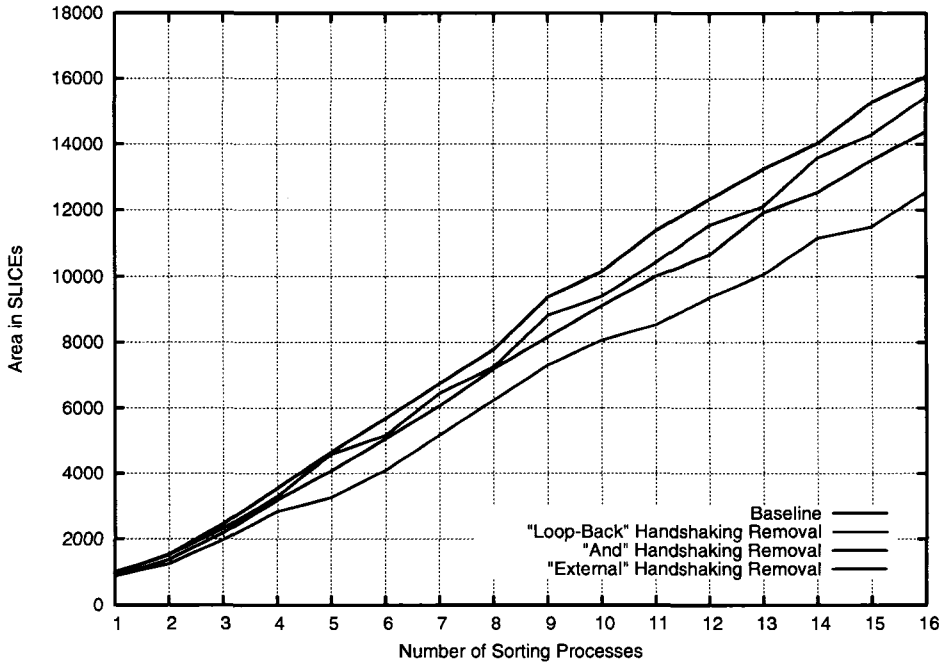


Figure 5.7: Comparison of handshaking removal methods.

does not have a severe impact on chip area. Its benefits, (no timing analysis, optimal firing pattern, simple graph construction) far outweigh the minor area requirements.

5.1.3 Handshaking Clock Frequency

It is difficult to quantify how handshaking logic affects the clock frequency of the designs. In every SA-C* program tested, the critical path does not lie within the handshaking logic, but rather within the computation itself, or in the implicit structures. The Alpha Data board used in testing is only capable of operating at a maximum frequency of 66 MHz. All tested designs either achieve this frequency, or fall short due to long combinational paths within the computational logic of the graph. Such paths can be overcome using pipelining. We can be assured that the handshaking is capable of operating well in excess of this frequency; however, its theoretical limit is not known.

5.2 Handshaking Benefits

It is clear that there is a small overhead associated with handshaking, so the obvious question is, "What benefits does it provide to offset the cost?" The primary benefits of handshaking are its flexibility, expressiveness, scalability, and performance.

The flexibility of the handshaking model can be illustrated by describing the process of adding a new node to the AHAHA model. This process is remarkably simple. All that is required to add a new node is a `.vhd` file containing the VHDL description, with a properly formatted preamble. If the node is a combinational node, then no extra considerations must be taken into account. If the node is clocked, then all that is required is to obey the handshaking protocol. If this is done, then the node will function correctly without any need to take into account the context in which the node is used. If the node requires access to an off-chip memory, it simply needs to supply the appropriate interface signals, and the timing of the other nodes accessing the same memory will automatically adjust. Additionally, porting the AHAHA to target a new hardware architecture is simple. Some of the implicit structures would need to be adapted to the new architecture, such as the interfaces to the host and memories. The graph itself, however, will automatically adapt to different memory latencies, memories with multiple ports, varying degrees of bandwidth to the memories and host. For example, consider an

environment where multiple FPGAs (each with their own independent AHAHA graphs) coexist on the same board and share a single port to memory. The memory arbitrators would need to communicate to serialize the memory accesses, but the rest of the graph would function correctly. The graph does not need to be tailored to the target hardware in any way.

The AHAHA handshaking model is expressive. It implicitly expresses parallelism, and exploits it fully at runtime. Inner loops and outer loops can execute concurrently. For example, if an outer loop contains some computations whose results are passed to the inner loop, then the outer loop will compute the values for iteration $n+1$ while the inner loop of iteration n is still executing. There is no need for a loop iteration to complete before the next one can begin, regardless of the nesting structure. Loop slowdown only occurs when there is a specific data-dependence loop which requires it, and then it is handled automatically.

The scalability of the AHAHA handshaking has already been illustrated using the pipeline sort algorithm. The handshaking cost grows linearly with the number of sections in the graph, and its performance does not degrade. The graphs can be arbitrarily complex, and as long as the buffer balancing requirement has been honored, the graph will execute correctly. In a statically timed model, a more complex graph results in a more complex schedule. In a handshaking model, the timing is automatic, and requires no extra work.

The performance of graphs using handshaking is at least as high as statically timed models. Nodes fire as soon as they are able, and because the handshaking is purely combinational, no extra time is wasted implementing it. In some cases though, the handshaking model is capable of out-performing a statically timed solution. For example, consider the program shown in Figure 5.8. Assume that the values of `S` are uniformly distributed in the range $[0 \ldots 65535]$.

In this example, there is a loop-carried dependence which encompasses the entire computation being done in the loop body. The iterations of the loop must be completely

```
1: uint16[:]  main (uint16 S[:], uint16 A[65536]) {
2:   hardware () {
3:     uint16 n = 0;
4:     uint16 R[:]  = for s in S {
5:       uint16 v =
6:         if((bits8)s == 0b00000000)
7:           return(A[A[A[A[A[A[s+n]]]]]])
8:         else
9:           return(s+n);
10:      next n = n + v;
11:    } return(array(v+n+s));
12:  };
13: } return (R);
```

Figure 5.8: Listing of unbalanced_conditional.sc.

serialized, and no pipeline parallelism is possible. However, the loop body contains a conditional, which contains 2 very different computations. In the first, 6 memory accesses are performed, requiring a minimum of 30 clock cycles. The second branch contains only combinational logic and so requires no additional clock cycles to execute. The more time-consuming branch will only be executed $\frac{1}{256}$th as often as the simpler branch. In this example, a statically timed implementation would need to assume the worst case. It would need to add artificial delay to the shorter branch to make both branches have the same delay (30 clock cycles). Using the AHAHA handshaking model, each loop iteration determines its rate independently of the others.

When the longer branch of the conditional is executed, the loop body requires 33 clock cycles, while the shorter only requires 3. 2 of these clock cycles are overhead from the `SPLIT` and `MERGE` nodes, and 1 is overhead from the `CIRCULATE`; the rest are contained within the branches of the conditional. Given n uniformly distributed values in the `S` array, a statically timed implementation will require $33 \cdot n$ clock cycles.[1] A handshaking model on the other hand will require only $(\frac{255}{256} \cdot 3 + \frac{1}{256} \cdot 33) \cdot n$, or $\sim 3.12 \cdot n$ clock cycles

[1]This analysis ignores any constant overhead required to begin and finish the loop, and focuses only on the loop body execution. The overhead in this example has been empirically found to be between 28 and 48 clock cycles.

on average. A statically timed implementation will take approximately ten and a half times longer to execute than a version using handshaking.

5.3 Stream Performance

Streams are a new feature of the SA-C* compiler, so their performance has not been evaluated previously. There are several interesting aspects to their performance which are worth discussing. Unless otherwise noted, the buffers which implement the streams have a depth of 16 elements.

There are 3 types of streams available: input streams, output streams, and local streams.[2] Input streams originate on the host, and are transferred over the PCI bus, to be consumed in the hardware. Output streams are exactly the opposite; they are produced on hardware, and consumed in the host. Local streams are produced and consumed on the hardware.

5.3.1 Test Program Selection Criteria

When testing streams, we would like to test the 3 varieties independently, so a code which can stress them separately is desirable. Simply put, a program which can consume a short stream and produce a long one, or consume a long one and produce a short one is needed.

Run Length Encoding (and Decoding) is a form of compression which removes long sequences of repeated values in a sequence of numbers, and replaces them with tuples of values, and their repeat count. It is well suited to these criteria.

For example, the sequence:

```
3, 3, 3, 3, 3, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 9, 9, 9, 9
```

would become:

---

[2]There is actually a fourth type of stream as well, which is both produced and consumed within the host processor. Since this stream type is implemented using multi-threading, and has absolutely nothing to do with FPGA hardware, or AHAHA, it has been intentionally omitted.

```
3, 5, 7, 11, 9, 4
```

This is because there are 5 3s, followed by 11 7s and 4 9s. In practical implementations, however, the repeat count is really the repeat count $-$ 1, since a repeat count of 0 would be meaningless. The example above would really be encoded as:

```
3, 4, 7, 10, 9, 3
```

This can be thought of as a 3, followed by 4 more 3s, a 7 followed by 10 more 7s, and a 9 followed by 3 more 9s.

```
1: export main;
2:
3: process array_to_stream (out stream uint32 S, param uint32 A[:])  {
4:   for a in A {
5:     put(a,S);
6:   };
7: };
8:
9: process rle_encode (in stream uint32 Sin,
10:       out stream uint32 Dout, stream uint32 Cout) {
11:   uint32 value = get(Sin);
12:   uint32 count = 0;
13:
14:   uint32 last_value, uint32 last_count = while(Sin) {
15:     uint32 new_value = get(Sin);
16:     bool different = (new_value != value ||
17:                       count == (uint32)0xffffffff);
18:     put(value,different,Dout);
19:     put(count,different,Cout);
20:     next value = new_value;
21:     next count = different?0:count+1;
22:   } return(final(value),final(count));
23:   put(last_value,Dout);
24:   put(last_count,Cout);
25: };
26:
27: process rle_decode (in stream uint32 Din, stream uint32 Cin,
28:       out stream uint32 Sout) {
29:   while(Din) {
30:     uint32 value = get(Din);
31:     uint32 count = get(Cin);
32:     put(value,Sout);
33:     while(count > 0) {
34:       put(value,Sout);
35:       next count = count-1;
36:     };
37:   };
38: };
39:
40: process interleave (in stream uint32 I1, stream uint32 I2,
41:       out stream uint32 O) {
42:   while(I1) {
43:     put(get(I1),O);
44:     put(get(I2),O);
45:   };
46: };
47:
48: process deinterleave (in stream uint32 I,
49:         out stream uint32 O1, stream uint32 O2) {
50:   while(I) {
51:     put(get(I),O1);
52:     put(get(I),O2);
53:   };
54: };
55:
```

Continued on next page

Continued from previous page

```
process network main(out stream uint32 S, param uint32 A[:])  {
  stream uint32 datain;
  stream uint32 dataout1;
  stream uint32 countout1;
  stream uint32 complete;
  stream uint32 dataout2;
  stream uint32 countout2;

  hardware() {
    instantiate array_to_stream(datain,A);
    instantiate rle_encode(datain, dataout1 ,countout1);
    instantiate interleave(dataout1,countout1,complete);
    instantiate deinterleave(complete,dataout2,countout2);
    instantiate rle_decode(dataout2 ,countout2,S);
  };
};
```

Table 5.2: Listing of rle_full_loop.sc.

In the codes used in this section (such as the one in Figure 5.2), the process `array_to_stream` is used to take all of the elements from an array and create a stream of its values. This allows stream algorithms to be executed on arrays. The processes `array_to_streama` and `array_to_streamb` are identical in operation to `array_to_stream`, except that they have pragmas applied to them so that the input array will be stored in different memories. They are used when a test is needed which uses 2 input arrays. This prevents slowdown due to memory contention.

The `rle_encode` process takes a stream of input values and produces 2 streams. 1 contains the value, and the other contains the repeat count for that value. The `rle_decode` process is the inverse operation to `rle_encode`. The `interleave` process combines the 2 streams from `rle_encode` into a single stream by interleaving the values, and `deinterleave` process the inverse operation. Table 5.2 shows a complete loop of encoding, interleaving, deinterleaving, and decoding an array. There is no way in SA-C* to convert an output stream back into an array, so the resulting stream is simply streamed back to the host. Because the measured runtimes do not include time required to transfer the stream back to the host (unless the stream fills and causes stalling within

the processes), this does not cause a problem when measuring the performance of the programs. Variations on this program can be used to create other programs which have varying degrees of input and output. For example, a program which only decodes can be given the sequence:

```
1, 999999
```

which consists of only 2 values, but will produce 1 million elements of output.

5.3.2 Program Performance

In order to analyze the behavior of the streams, it is useful to first understand the runtimes of the various processes used in the programs on their own. Runtimes are measured beginning after any input arrays have been uploaded to the hardware, and ending when all of the output streams have been closed. Output streams may still contain values which have not been transferred to the host, but the time required to transfer them does not affect the runtime. All runtimes are measured in clock cycles.

```
1: export main;
2:
3: process array_to_stream (out stream uint32 S, param uint32 A[:])  {
4:   for a in A {
5:     put(a,S);
6:   };
7: };
8:
9: process network main(out stream uint32 S, param uint32 A[:])  {
10:  hardware() {
11:    instantiate array_to_stream(S @ regfile 32768,A);
12:  };
13: };
```

Figure 5.9: Listing of a2s.sc.

First, the array_to_stream[3] can be analyzed using the program shown in Figure 5.9. The performance of the output stream can be ignored, as long as the input array is less than the depth of the output stream (32768). As long as the output stream buffer never completely fills, it will never stall the program, so the execution time will be deterministic, and will not involve the PCI bus or host. Table 5.3 shows the runtime of the array_to_stream on its own. From the table, it is clear that the process requires 2

Input Size	Clock Cycles
100	213
200	413
500	1013
1000	2013
2000	4013
5000	10013

Table 5.3: Runtime of the array_to_stream process.

clock cycles per data element, and that there is a constant overhead of 13 clock cycles. The execution time of the array_to_stream process is not data-dependent because it does not contain any conditionals.

The rle_encode process can be evaluated using the program in Figure 5.10, which uses the array_to_stream process, which is known to use 2 clock cycles for each data element. As long as the rle_encode process running concurrently with the array_to_stream process takes more than 2 clock cycles per data element, then we can be sure that we are measuring the rle_encode process, and not array_to_stream, since the 2 processes will execute at the rate of the slower one.

The timing of the rle_encode process might be data-dependent, since the length of its output stream is data-dependent. In order to analyze the effect of the output data size, different types of data are used. From Table 5.4, we can see that the process requires 3 clock cycles per data element, and that the overhead has increased to 15 clock

[3]The array_to_streama and array_to_streamb processes behave identically to the array_to_stream process.

```
1: export main;
2:
3: process array_to_stream (out stream uint32 S, param uint32 A[:])  {
4:   for a in A {
5:     put(a,S);
6:   };
7: };
8:
9: process rle_encode (in stream uint32 Sin,
10:       out stream uint32 Dout, stream uint32 Cout) {
11:   uint32 value = get(Sin);
12:   uint32 count = 0;
13:
14:   uint32 last_value, uint32 last_count = while(Sin) {
15:     uint32 new_value = get(Sin);
16:     bool different = (new_value != value ||
17:                       count == (uint32)0xffffffff);
18:     put(value,different,Dout);
19:     put(count,different,Cout);
20:     next value = new_value;
21:     next count = different?0:count+1;
22:   } return(final(value),final(count));
23:   put(last_value,Dout);
24:   put(last_count,Cout);
25: };
26:
27: process network main(out stream uint32 Vals,
28:                           stream uint32 Counts,
29:                      param uint32 A[:])  {
30:   stream uint32 tmp;
31:   hardware() {
32:     instantiate array_to_stream(tmp,A);
33:     instantiate rle_encode(tmp,
34:                            Vals @ regfile 32768,
35:                            Counts @ regfile 32768);
36:   };
37: };
```

Figure 5.10: Listing of a2s_enc.sc.

Input Size	Input	Clock Cycles
100	100 1s	315
200	200 1s	615
500	500 1s	1515
1000	1000 1s	3015
2000	2000 1s	6015
5000	5000 1s	15015
100	1, 2, 3, ..., 100	315
200	1, 2, 3, ..., 200	615
500	1, 2, 3, ..., 500	1515
1000	1, 2, 3, ..., 1000	3015
2000	1, 2, 3, ..., 2000	6015
5000	1, 2, 3, ..., 5000	15015

Table 5.4: Runtime of the rle_encode process.

cycles. Additionally, the nature of the data, and the size of the output do not affect the runtime of the process. This is not surprising, since the code does not actually contain any conditionals, just predicated instructions (the boolean flag being used in the `put`, is implemented using predicated instructions).

The `interleave` process can be tested using the program in Figure 5.11. Note that in this program, the `array_to_streama` and `array_to_streamb` processes are used in place of the `array_to_stream` process to prevent memory contention.

Because the timing of the `interleave` process is not data-dependent, the runtime will only vary with the input size. The second input array, `B`, is the same size as the first. Table 5.5 shows that the process requires 3 clock cycles per data element, and

Input Size	Clock Cycles
100	316
200	616
500	1516
1000	3016
2000	6016
5000	15016

Table 5.5: Runtime of the interleave process.

that the overhead is now 16 clock cycles. 3 clock cycles per iteration may seem high, since the interleave program is so simple, but it can be easily explained. The loop will

```
1: export main;
2:
3: process array_to_streama (out stream uint32 S, param uint32 A[:])  {
4:   // PRAGMA(input(A memory 1))
5:   for a in A {
6:     put(a,S);
7:   };
8: };
9:
10: process array_to_streamb (out stream uint32 S, param uint32 B[:])  {
11:   // PRAGMA(input(B memory 2))
12:   for b in B {
13:     put(b,S);
14:   };
15: };
16:
17: process interleave (in stream uint32 I1, stream uint32 I2,
18:       out stream uint32 O) {
19:   while(I1) {
20:     put(get(I1),O);
21:     put(get(I2),O);
22:   };
23: };
24:
25: process network main(out stream uint32 S,
26:                      param uint32 A[:], uint32 B[:])  {
27:   stream uint32 tmp1;
28:   stream uint32 tmp2;
29:   hardware() {
30:     instantiate array_to_streama(tmp1,A);
31:     instantiate array_to_streamb(tmp2,B);
32:     instantiate interleave(tmp1,tmp2,S @ regfile 32768);
33:   };
34: };
```

Figure 5.11: Listing of a2s_int.sc.

execute at the speed of its largest dependence cycle. There are 3 cycles in the loop, corresponding to the stream-serialization chains of I1, I2, and O. The I1 chain includes a STREAM_IS_OPEN node which makes up the test of the while-loop, a STREAM_GET node to read from the stream, and a CIRCULATE node to cycle the flag into the next iteration. Each of these 3 nodes require 1 clock cycle to operate. The I2 chain includes only a STREAM_GET node and a CIRCULATE node, so it is able to cycle once every 2 clock cycles. The O chain includes 2 STREAM_PUT nodes, and a CIRCULATE node, so it is able to cycle once every 3 clock cycles. Notice that the gets and puts use separate serialization-flag chains, but the gets and puts of the same iteration cannot be executed in parallel, since there is a direct data-dependence between them.

This dependence will define the relative timing behavior of the 3 serialization chains. For example, in clock cycle n, the STREAM_IS_OPEN node produces its value. In $n+1$, both of the STREAM_GET nodes are able to execute, but both of the STREAM_PUT nodes must stall. The first is waiting for the data from the first STREAM_GET node, and the second is waiting for the data from the second STREAM_GET node as well as the serialization-flag from the first STREAM_PUT. In $n+2$, the first STREAM_PUT may execute, as well as the CIRCULATE nodes for the I1 and I2 chains. In $n+3$, the STREAM_IS_OPEN is ready to execute for the second iteration of the loop, while the second STREAM_PUT node executes for the first iteration. In $n+4$, both of the STREAM_GET nodes execute for the second loop iteration, while the CIRCULATE node for the O chain executes for its first iteration. From there, $n+5$ operates exactly as $n+2$, and the process will continue until the I1 stream is closed. In the initial loop iteration, the bottom half of the loop (the puts) stall waiting for the top half (the while-loop test and the gets) to provide data. However, the top half of the second iteration is able to begin before the first iteration has fully completed, allowing a small amount of pipeline parallelism. Without this parallelism, the loop would require 4 clock cycles per iteration. This example shows some of the power of the dynamic scheduling model of the AHAHA.

```
1: export main;
2:
3: process array_to_stream (out stream uint32 S, param uint32 A[:])  {
4:   for a in A {
5:     put(a,S);
6:   };
7: };
8:
9: process deinterleave (in stream uint32 I,
10:         out stream uint32 O1, stream uint32 O2) {
11:   while(I) {
12:     put(get(I),O1);
13:     put(get(I),O2);
14:   };
15: };
16:
17: process network main(out stream uint32 S1,
18:                          stream uint32 S2,
19:                      param uint32 A[:])  {
20:   stream uint32 tmp;
21:   hardware() {
22:     instantiate array_to_stream(tmp,A);
23:     instantiate deinterleave(tmp,
24:                              S1 @ regfile 32768,
25:                              S2 @ regfile 32768);
26:   };
27: };
```

Figure 5.12: Listing of a2s_deint.sc.

The `deinterleave` process can be tested using the program in Figure 5.12. Since its behavior is not data-dependent, the runtime only varies with input size. Table 5.6 shows that it requires 2 clock cycles element, with 18 clock cycles of overhead.

Input Size	Clock Cycles
100	218
200	418
500	1018
1000	2018
2000	4018
5000	10018

Table 5.6: Runtime of the deinterleave process.

Lastly, the `rle_decode` process can be tested using the program in Figure 5.13. Because the timing of the `rle_decode` process is data dependent, the runtime may vary with the input size, its content, and the output size. Different types of data are provided to it in order to determine its behavior.

From Table 5.7, we can see that the process requires $20 + 6i + 2o$ where i is the number of elements in the input arrays, and o is the number of elements in the output stream. Further testing was done to verify this behavior, but the results are not shown here. The formula was tested by executing the program with over 50,000 random sequences and comparing the measured number of clock cycles against the expected value from the equation. Only tests which produced output sequences with less than 32768 values were used. All tests confirmed the equation.

5.3.3 Local Streams

Because local streams do not involve the PCI bus or host, their behavior is quite predictable. They allow multiple processes to run at their own rate, with some slack between them. As long as their size is large enough to prevent deadlock, and to allow some flexibility in the execution of the individual processes, they do not directly affect the performance of the programs. When one or both of the processes they join have data-dependent rates, a larger buffer may be used to prevent stalling. But in the absence of

```
1: export main;
2:
3: process array_to_streama (out stream uint32 S, param uint32 A[:])  {
4:   // PRAGMA(input(A memory 1))
5:   for a in A {
6:     put(a,S);
7:   };
8: };
9:
10: process array_to_streamb (out stream uint32 S, param uint32 B[:])  {
11:   // PRAGMA(input(B memory 2))
12:   for b in B {
13:     put(b,S);
14:   };
15: };
16:
17: process rle_decode (in stream uint32 Din, stream uint32 Cin,
18:       out stream uint32 Sout) {
19:   while(Din) {
20:     uint32 value = get(Din);
21:     uint32 count = get(Cin);
22:     put(value,Sout);
23:     while(count > 0) {
24:       put(value,Sout);
25:       next count = count-1;
26:     };
27:   };
28: };
29:
30: process network main(out stream uint32 S,
31:                       param uint32 A[:], uint32 B[:])  {
32:   stream uint32 tmp1;
33:   stream uint32 tmp2;
34:   hardware() {
35:     instantiate array_to_streama(tmp1,A);
36:     instantiate array_to_streamb(tmp2,B);
37:     instantiate rle_decode(tmp1,tmp2,S @ regfile 32768);
38:   };
39: };
```

Figure 5.13: Listing of a2s_dec.sc.

Input Size	Output Size	Input A	Input B	Clock Cycles
1	100	1	99	226
1	200	1	199	426
1	500	1	499	1026
1	1000	1	999	2026
1	2000	1	1999	4026
1	5000	1	4999	10026
2	100	1, 2	49, 49	232
2	200	1, 2	99, 99	432
2	500	1, 2	249, 249	1032
2	1000	1, 2	499, 499	2032
2	2000	1, 2	999, 999	4032
2	5000	1, 2	2499, 2499	10032
5	100	1, 2, ..., 5	19, 19, ...	250
5	200	1, 2, ..., 5	39, 39, ...	450
5	500	1, 2, ..., 5	99, 99, ...	1050
5	1000	1, 2, ..., 5	199, 199, ...	2050
5	2000	1, 2, ..., 5	399, 399, ...	4050
5	5000	1, 2, ..., 5	999, 999, ...	10050
10	100	1, 2, ..., 10	9, 9, ...	280
10	200	1, 2, ..., 10	19, 19, ...	480
10	500	1, 2, ..., 10	49, 49, ...	1080
10	1000	1, 2, ..., 10	99, 99, ...	2080
10	2000	1, 2, ..., 10	199, 199, ...	4080
10	5000	1, 2, ..., 10	499, 499, ...	10080

Table 5.7: Runtime of the rle_decode process.

a deadlock-prone program, 16 is usually sufficient. It is impossible to create a SA-C* program which uses local streams, which does not produce an output stream, so local streams are difficult to analyze empirically. However, using the co-simulation environment discussed in Appendix B, one can easily verify that they behave as expected in the RLE-based programs.

The performance of local streams depends entirely upon the loop in which they are used. Stream operations execute in the same manner as any other clocked node. The streams are able to perform operations in every clock cycle. The critical path is often the stream-serialization-flag. For example, the `array_to_stream` process produces a new data element once every 2 clock cycles, but from the SA-C* code, one might expect

it to produce in every clock cycle. The SA-C* compiler is not able to generate the required optimizations to make this possible. It uses a CIRCULATE node to provide the serialization-flag, which causes the loop to slow down.

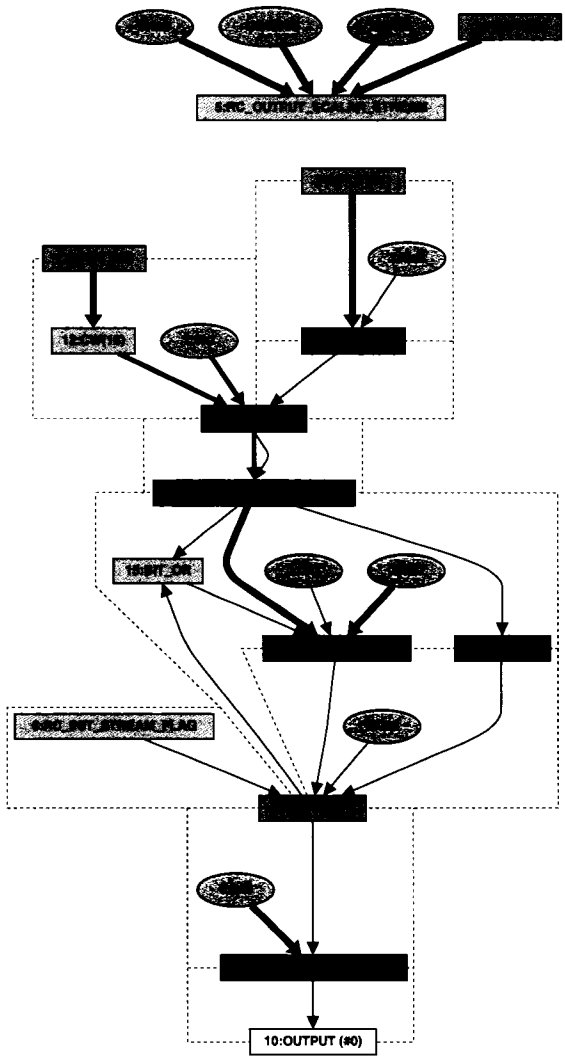


Figure 5.14: SA-C* generated AHAHA graph of the program in Figure 5.9.

Figure 5.14 shows the AHAHA graph generated by the SA-C* compiler by the array_to_stream test code (Figure 5.9). In this case, however, it is possible to construct an AHAHA graph which can produce a value in every cycle. It requires fusing the last-flag, and the stream-serialization-flag. Effectively, the last-flag would be used directly as the stream-serialization-flag for the STREAM_PUT node, and a DONE node would be

used to generate the serialization-flag for the STREAM_FINALIZE. This is possible because there is only 1 stream operation int he loop, so no serialization is required. Figure 5.15 shows the optimized graph.

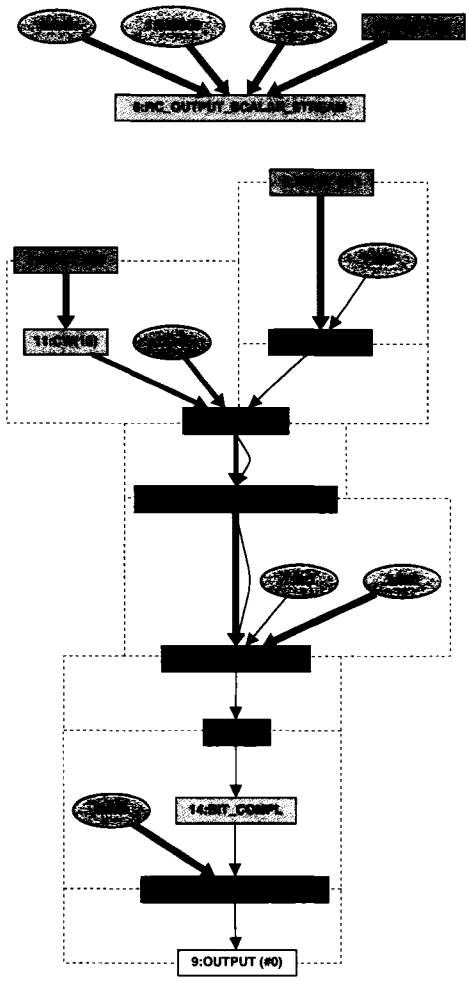


Figure 5.15: Optimized AHAHA graph of the program in Figure 5.9.

The optimized version is able to produce 1 value per clock cycle. The same technique could be used for processes which read from a stream and contain only 1 STREAM_GET node. However, this situation is far less common, since most processes which read from streams often require a STREAM_IS_OPEN node to drive the loop. When combined with the STREAM_GET nodes in the loop body, there will be at least 2 stream operations which must be serialized. With the addition of the CIRCULATE node, processes which read from streams are often limited to using an inner loop which executes in at least 3 clock cycles.

The effects can be minimized by using multiple `STREAM_GET` nodes within the body. In some situations where the number of elements in the stream is known (such as a stream that is generated from an array or image), the stream-serialization can be removed, as with the `STREAM_PUT` example above. This is done by using a for-loop instead of a while-loop, and not testing whether the stream is open or not.

5.3.4 Input Streams

In order to test input streams, a program which accepts a variable amount of data, and produces a small amount of output, is required. For this reason, the RLE encoding algorithm is used (Figure 5.16). The input is a sequence of repeating 0s. The number of 0s in the sequence can be changed to control the size of the input stream. The input buffer depth is varied from its default depth of 16 elements, up to 64k (65536). A 64k-deep buffer of 32-bit words is the largest buffer that fits within the FPGA; a 128k buffer would use 256 BlockRAMs, but the chip only contains 144. The output buffer is left at its default size of 16, since the output will contain only 2 values.

There are 2 models that the hardware can use to notify the host. In the first model, the host is interrupted when an input stream is empty, or an output stream is full, causing the hardware to stall. This method causes an interrupt to be sent only when the hardware requires host intervention in order to proceed. In the second model (called "greedy interrupts"), the hardware sends an interrupt as soon as an input buffer is not full, or an output buffer is not empty. This causes the hardware to send an interrupt when host intervention might be helpful, but before a stall occurs. This model is more active, and puts more pressure on the host, but makes the overall process more efficient since the buffers are filled and emptied more often. Additionally, the host takes a long time to respond to an interrupt, so sending the interrupt early reduces time wasted while waiting for the host to respond. Greedy interrupts are used for all of the results shown here, since they only improve performance. Additionally, the Alpha Data board supports DMA transfers as well as regular (Programmed I/O- or PIO-based) transfers. The results in this section were obtained using PIO.

```
1: export main;
2:
3: process rle_encode (in stream uint32 Sin,
4:       out stream uint32 Dout, stream uint32 Cout) {
5:   uint32 value = get(Sin);
6:   uint32 count = 0;
7:
8:   uint32 last_value, uint32 last_count = while(Sin) {
9:     uint32 new_value = get(Sin);
10:    bool different = (new_value != value ||
11:                      count == (uint32)0xffffffff);
12:    put(value,different,Dout);
13:    put(count,different,Cout);
14:    next value = new_value;
15:    next count = different?0:count+1;
16:  } return(final(value),final(count));
17:  put(last_value,Dout);
18:  put(last_count,Cout);
19: };
20:
21: process interleave (in stream uint32 I1, stream uint32 I2,
22:       out stream uint32 O) {
23:   while(I1) {
24:     put(get(I1),O);
25:     put(get(I2),O);
26:   };
27: };
28:
29: process network main(in stream uint32 inS, out stream uint32 outS) {
30:   stream uint32 data;
31:   stream uint32 count;
32:
33:   hardware() {
34:     instantiate rle_encode(inS @ regfile 16, data ,count);
35:     instantiate interleave(data,count,outS);
36:   };
37: };
```

Figure 5.16: Listing of rle_encode16.sc.

Since the host is a multi-tasking, multi-user machine, it is not exclusively dedicated to managing the FPGA application. While the tests are running, it may be processing data from the network, running system maintenance jobs, or other tasks which may utilize some of its resources. Because the host machines containing the Alpha Data boards are only accessible via a remote login, it is not an option to disable most of these services. For all intents and purposes, the runtime behavior of the host processor is nondeterministic. Therefore, in cases where the runtime of the hardware is influenced by the behavior of the host processor (such as when streams which span the PCI bus are used), the hardware execution time is also nondeterministic. The host may cause the runtimes to be longer than expected, but cannot cause them to be shorter. These fluctuations have nothing to do with the hardware, the AHAHA model, or the runtime system and, as such, should not be used to characterize their behavior. Ideally, the minimum runtime of a vast number of executions would be used to characterize the performance of the AHAHA model, because it represents the behavior of the hardware when the host is performing optimally. However, the measured runtimes have significant noise in them, and the number of runs which would be required to separate the underlying trends from the noise would be tremendous. Data was gathered continuously for over a week to obtain 500 runtimes for each program. Many of the observed distributions conform roughly to a Poisson, but many do not. Furthermore, when DMA is used, the runtimes become even more unpredictable; some even appear to be bimodal. Because of the non-uniform behavior of the hardware, it is difficult to characterize the runtimes using a well-known statistical method. Even using the minimum runtime for each program has significant problems. Because of the limited number of samples, it is still heavily influenced by the noise. A more robust metric is required to identify overall trends and separate them from the noise. To accomplish this, the average of the fastest 5% (25) runtimes is used. The same metric is used in Section 5.3.6 when DMA transfers are used.

Table 5.8 shows the results of testing the input streams. Figure 5.17 shows a plot of the same values. All 3 axes of the plot use a log scale. The graph is 4-dimensional. Buffer

Input Size	Input Buffer Depth												
	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k
10	45	45	45	45	45	45	45	45	45	45	45	45	45
20	3490	75	75	75	75	75	75	75	75	75	75	75	75
50	26988	3637	165	165	165	165	165	165	165	165	165	165	165
1e2	45615	32672	3756	315	315	315	315	315	315	315	315	315	315
2e2	80683	57445	37610	4108	615	615	615	615	615	615	615	615	615
5e2	1.86e5	1.34e5	99802	63935	6634	1515	1515	1515	1515	1515	1515	1515	1515
1e3	3.62e5	2.64e5	2.06e5	1.61e5	1.12e5	10191	3015	3015	3015	3015	3015	3015	3015
2e3	7.12e5	5.18e5	4.20e5	3.57e5	2.95e5	2.01e5	18605	6015	6015	6015	6015	6015	6015
5e3	1.77e6	1.29e6	1.06e6	9.36e5	8.39e5	7.34e5	5.51e5	2.12e5	16637	15015	15015	15015	15015
1e4	3.52e6	2.58e6	2.11e6	1.90e6	1.75e6	1.62e6	1.42e6	1.08e6	3.97e5	30916	30015	30015	30015
2e4	7.01e6	5.14e6	4.22e6	3.80e6	3.55e6	3.39e6	3.15e6	2.81e6	2.13e6	7.77e5	60015	60015	60015
5e4	1.75e7	1.28e7	1.05e7	9.49e6	8.96e6	8.66e6	8.33e6	7.97e6	7.27e6	5.92e6	3.23e6	2.61e5	1.50e5
1e5	3.51e7	2.57e7	2.11e7	1.90e7	1.80e7	1.75e7	1.70e7	1.66e7	1.58e7	1.45e7	1.18e7	6.43e6	1.10e7
2e5	7.03e7	5.14e7	4.22e7	3.81e7	3.60e7	3.51e7	3.43e7	3.38e7	3.30e7	3.16e7	2.89e7	2.36e7	4.88e7
5e5	1.76e8	1.29e8	1.06e8	9.55e7	9.03e7	8.81e7	8.64e7	8.57e7	8.46e7	8.31e7	8.05e7	7.51e7	1.67e8

Table 5.8: Runtimes of RLE encoding with varying buffer and input sizes.

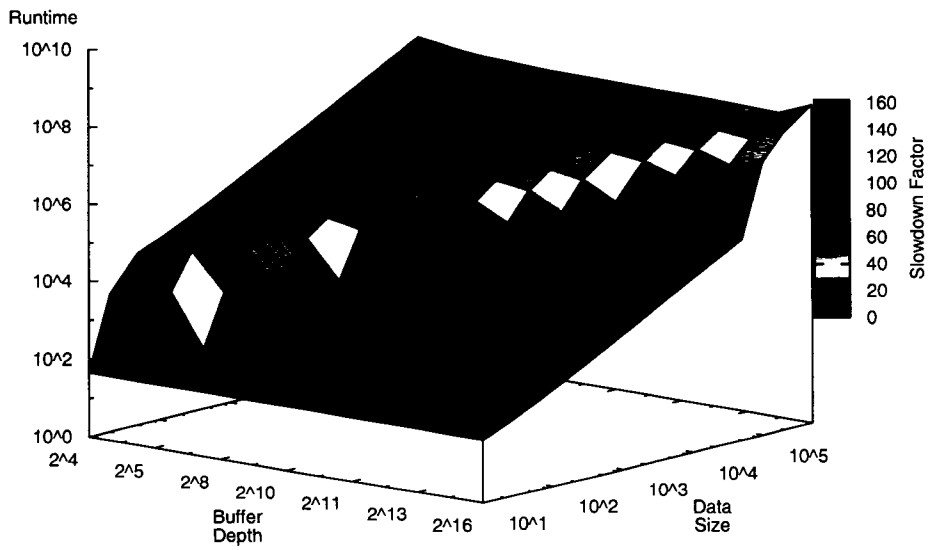


Figure 5.17: Plot of values in Table 5.8.

Depth and Data Size are the independent variables. Runtime and Slowdown Factor are dependent variables. Runtime is the absolute measured runtime of the execution, and is represented by the height of the surface. The Slowdown Factor represents the relative slowdown versus the optimal (non-stalling) rate, and is represented by the color of the surface. For example, a value of 1 indicates that it executed at the optimal rate, and a value of 10 indicates that it ran 10 times slower than optimal.

In the table, numbers in blue indicate cases where the entire stream fits within the available buffer space. In this situation, the graph never stalls waiting for data, and the hardware finishes its execution without the host playing a role in the runtime. In these situations, it is able to process a value every 3 clock cycles, since the loop bodies of both the `rle_encode` and `interleave` processes are able to execute at this rate. 15 clock cycles of overhead are present in the computation.

Red numbers in the table indicate that the hardware was able to run to completion without stalling, even though the entire stream did not fit in the buffer at one time. In these cases, the host was able to respond to the interrupt and begin transferring data into the input buffer before it emptied. This does not occur often because the PCI bus is such a significant bottleneck in the system.

In most cases, the runtimes increase dramatically when the buffer cannot contain the entire stream. In these situations, the minimum number on each row of the table is indicated in green to identify the "best" buffer size for each input size. In general, a larger buffer operates more efficiently, and shows less slowdown than a smaller one. Smaller buffers will generate many more interrupts, and transfer small amounts of data at a time. Larger buffers use larger block transfers, with fewer interrupts, which is usually more efficient. However, it is interesting to note that this trend stops at the 32k buffer size, which outperforms the larger 64k buffer. To explain this phenomenon, some details about the runtime system are needed. First, the reaction time of the host (that is the time before the host acknowledges an interrupt) is large. Second, the host requires some additional time to prepare the data to be sent to the hardware which is proportional to

the amount of data being transferred. And lastly, as soon as the first value from the host is enqueued into the input stream on the hardware, the computation can continue; the hardware does not wait for the transfer to complete before it resumes execution.

When the buffer size is very small, the efficiency is very low because the hardware runs out of data very quickly. In most cases, it runs out of data before the host has acknowledged the interrupt. Most of the runtime is consumed waiting for the host to respond. Once it does respond, the host prepares the next batch of data, which is no larger than the available space in the hardware buffer. As the buffer size increases, the efficiency also increases since the hardware is able to work for longer before running out of data. However, there is a point where the host takes so long to prepare the data before sending it that it begins to outweigh the constant response time.

5.3.5 Output Streams

To test output streams, the RLE decoding algorithm shown in Figure 5.18 is used. It is able to produce large amounts of output from small input sequences. The input supplied is simply 2 values long. The first value is a constant 0 (corresponding to the value which will be repeated in the output stream), and the second value (which corresponds to the repeat count) is varied to change the output size. For example, the sequence: `(0, 999)` is used to create an output stream of 1000 elements. The output buffer depth is varied from 16, to 64k (65536) while the input buffer is left at a constant 16, since it will contain only 2 values. Each program is executed 500 times, and the average of the fastest 5% runtimes is used.

Just as with the input streams, Table 5.9 and Figure 5.19 show that when the entire output sequence fits within the available buffer, the program exhibits no slowdown. In these situations, the hardware is able to process a value every 2 clock cycles, since the loop bodies of both the `rle_decode` and `deinterleave` processes are able to execute at this rate. 22 clock cycles of overhead are present in the computation.

Unlike with the input streams, a larger buffer is always beneficial. This is due to the behavior of the host when dealing with output streams. The host delay which is

```
1: export main;
2:
3: process rle_decode (in stream uint32 Din, stream uint32 Cin,
4:       out stream uint32 Sout) {
5:   while(Din) {
6:     uint32 value = get(Din);
7:     uint32 count = get(Cin);
8:     put(value,Sout);
9:     while(count > 0) {
10:      put(value,Sout);
11:      next count = count-1;
12:    };
13:  };
14: };
15:
16: process deinterleave (in stream uint32 I,
17:         out stream uint32 O1, stream uint32 O2) {
18:   while(I) {
19:     put(get(I),O1);
20:     put(get(I),O2);
21:   };
22: };
23:
24: process network main(in stream uint32 Sin, out stream uint32 Sout) {
25:   stream uint32 datain;
26:   stream uint32 dataout;
27:   stream uint32 countout;
28:
29:   hardware() {
30:     instantiate deinterleave(Sin,dataout,countout);
31:     instantiate rle_decode(dataout ,countout,Sout @ regfile 16);
32:   };
33: };
```

Figure 5.18: Listing of rle_decode16.sc.

Output Size	Output Buffer Depth												
	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k
10	42	42	42	42	42	42	42	42	42	42	42	42	42
20	2609	62	62	62	62	62	62	62	62	62	62	62	62
50	51856	3531	122	122	122	122	122	122	122	122	122	122	122
1e2	1.17e5	89329	4722	222	222	222	222	222	222	222	222	222	222
2e2	2.48e5	2.12e5	1.64e5	7166	422	422	422	422	422	422	422	422	422
5e2	6.61e5	5.77e5	4.79e5	3.22e5	18396	1022	1022	1022	1022	1022	1022	1022	1022
1e3	1.34e6	1.23e6	1.11e6	9.36e5	6.32e5	34236	2022	2022	2022	2022	2022	2022	2022
2e3	2.71e6	2.50e6	2.38e6	2.18e6	1.86e6	1.25e6	66341	4022	4022	4022	4022	4022	4022
5e3	6.80e6	6.33e6	6.05e6	5.86e6	5.52e6	4.88e6	3.60e6	1.23e6	61469	10022	10022	10022	10022
1e4	1.36e7	1.27e7	1.22e7	1.19e7	1.16e7	1.09e7	9.64e6	7.61e6	4.39e6	1.69e5	20022	20022	20022
2e4	2.73e7	2.54e7	2.44e7	2.39e7	2.36e7	2.31e7	2.17e7	2.02e7	1.58e7	1.10e7	1.27e6	40022	40022
5e4	6.85e7	6.37e7	6.12e7	5.99e7	5.93e7	5.83e7	5.69e7	5.45e7	4.97e7	4.05e7	2.17e7	2.17e6	1.00e5
1e5	1.37e8	1.27e8	1.23e8	1.20e8	1.19e8	1.18e8	1.16e8	1.15e8	1.12e8	1.04e8	9.37e7	6.38e7	3.31e6
2e5	2.75e8	2.56e8	2.46e8	2.41e8	2.39e8	2.37e8	2.35e8	2.34e8	2.30e8	2.25e8	2.15e8	1.93e8	1.54e8
5e5	6.94e8	6.47e8	6.21e8	6.09e8	6.15e8	6.07e8	6.15e8	6.35e8	6.29e8	6.19e8	6.03e8	5.77e8	4.82e8

Table 5.9: Runtimes of RLE decoding with varying buffer and output sizes.

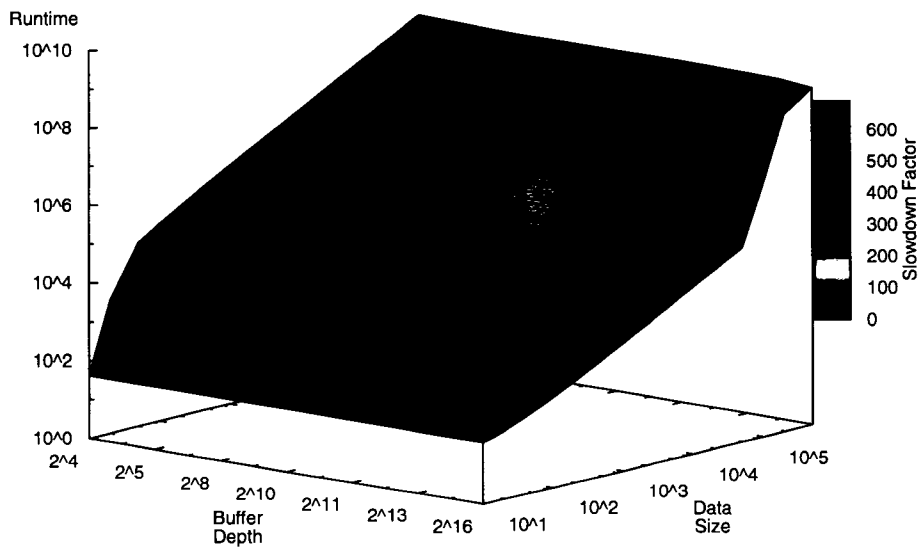


Figure 5.19: Plot of values in Table 5.9.

proportional to the size of the transfer does not take place before the transfer commences, but rather after it completes. Because of this difference, the host begins emptying the output buffers after a constant delay, and then processes the values while the hardware executes. With input streams, the host processes the values while the hardware is stalled. For this reason, the larger the output buffer is, the more efficient the hardware execution will be. The delay caused by the host is simply the time required to respond to an interrupt, times the number of interrupts. The larger the buffer, the fewer interrupts are generated, and the better the performance.

However, the slowdown factors are much higher in the case of output streams. This is because the rate at which the RLE decode program executes (2 data elements per clock cycle) is faster than the RLE encoding program (3 data elements per clock cycle). The "optimal rate" is much faster for decoding than encoding.

To understand why this has such an impact on the runtime, first consider a hypothetical program which is able to process a value once every clock cycle, and assume that the host is able to transfer a value into the stream at every clock cycle.[4] In the case of an input stream, this would mean that the process is able to remove elements from the stream at the same rate they are added. For an output stream, it means that the process can begin refilling the stream as it is being emptied. In both of these conditions, the host is never able to get ahead of the hardware. If the hardware was stalling before the transfer begins, it will stall again immediately after the transfer completes. Since input streams begin full, and output streams begin empty, the host has a certain amount of "slack" at the beginning of the execution. Any delays caused by the host consume this slack. After the slack is exhausted, any delays caused by the host will cause the hardware to stall. In contrast, consider a program (like RLE decode) which processes a value once every 2 clock cycles. As the host transfers values, the process is not able to

[4]Realistically, the host is not able to provide a value at every clock cycle when transferring, and so runs more slowly. This further exacerbates the problem, which helps to explain why the difference is so significant between the RLE encoding and decoding.

keep up, and the host is able to regain some slack. When the hardware processes a value every 3 clock cycles, the behavior is even more pronounced.

For example, imagine that the host transfers 1000 values, and that the hardware is stalling before the transfer begins. If the rate is 1 clock cycle per element, the hardware will process the 1000 values as they arrive, and then immediately stall again since the host has 0 clock cycles of slack to begin the next transfer. If the rate is 2 clock per element, the hardware will only process 500 values while the transfer occurs, and 500 more in the following 1000 clock cycles. The host has 1000 clock cycles of slack in which to begin the next transfer to prevent stalling. If the rate is 3 clock cycles per element, the hardware will process $\sim$333 values as the transfer occurs, and then the remaining $\sim$667 values will be processed in the following 2000 clock cycles. The faster the hardware operates, the faster the host needs to respond to prevent slowdown.

5.3.6 Streams Using DMA

The RTS for the Alpha Data board supports DMA-based transfers, as well as regular PIO-based transfers. Unfortunately, the behavior of the host is even more unpredictable when DMA-based I/O is used. The transfers themselves are able to go much faster, but there is a significant overhead before and after the transfers occur. In the case of stream data, the data transfer size is at most 64k words, which is small for a DMA transfer. The overhead to initiate DMA is large, so it cannot be effectively amortized using such small transfer sizes. When transferring large amounts of data, DMA can be quite efficient, such as when the FPGA bitstream is being downloaded, or when a large array is being stored in memory before the hardware begins execution. But for stream operations, DMA performs poorly and erratically. Nevertheless, the same experiments were carried out using DMA-based transfers, and the results are shown in the following two sections. Entries in the tables are underlined whenever the DMA transfers outperformed standard PIO transfers. As before, the best 5% of 500 runs are averaged, but there is no suitable statistical justification for this, except that it discards the large number of outliers which run slowly.

5.3.7 Input Streams Using DMA

Table 5.10 and Figure 5.20 show the execution time of the RLE encode when DMA is used. When the input fits entirely within the stream buffer, DMA does not affect the runtime since the time for the initial filling of the buffer occurs prior to timing the execution. When the buffer sizes are in the midrange (256 to 2048 elements) the DMA outperforms PIO. The runtime degradation for larger buffers is perplexing, and is likely due to the complex and nondeterministic behavior of the host. It is interesting to note that the red entries in the table (which indicate that the hardware did not stall when the stream was larger than the buffer) are more prevalent here than they were in the regular transfers. This is due to the fact that once the transfer begins, it runs at a faster rate than a PIO transfer. DMA transfers are able to provide a value in every clock cycle. If the host responds to the interrupt before the buffer completely empties and begins transferring data, then the hardware will not stall.

Also, notice that when more than 2 transfers are required, the hardware always stalls. When only 1 transfer occurs, the setup and shutdown time for the DMA transfer are not timed. When an additional transfer is required, its setup time occurs during execution, but the shutdown time may occur after the hardware execution has completed. When more than 2 transfers are required, both the setup and shutdown times have an impact, and their combined impact causes the hardware to stall.

Input Size	Input Buffer Depth												
	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k
10	45	45	45	45	45	45	45	45	45	45	45	45	45
20	5240	75	75	75	75	75	75	75	75	75	75	75	75
50	34112	5280	165	165	165	165	165	165	165	165	165	165	165
1e2	55414	36264	5387	315	315	315	315	315	315	315	315	315	315
2e2	99093	66724	42090	5525	615	615	615	615	615	615	615	615	615
5e2	2.29e5	1.53e5	1.11e5	72023	6435	1515	1515	1515	1515	1515	1515	1515	1515
1e3	4.39e5	3.06e5	2.29e5	1.67e5	1.13e5	7684	3015	3015	3015	3015	3015	3015	3015
2e3	8.45e5	5.97e5	4.64e5	3.70e5	2.97e5	1.96e5	13095	6015	6015	6015	6015	6015	6015
5e3	2.07e6	1.46e6	1.15e6	9.76e5	8.51e5	7.25e5	5.44e5	1.97e5	15015	15015	15015	15015	15015
1e4	4.11e6	2.89e6	2.26e6	1.94e6	1.76e6	1.60e6	1.39e6	1.04e6	7.48e5	30015	30015	30015	30015
2e4	8.20e6	5.73e6	4.49e6	3.87e6	3.54e6	3.33e6	3.10e6	2.70e6	2.01e6	1.57e6	60015	60015	60015
5e4	2.08e7	1.45e7	1.13e7	9.76e6	8.91e6	8.50e6	8.12e6	7.84e6	7.43e6	6.44e6	5.14e6	1.50e5	1.50e5
1e5	4.19e7	2.91e7	2.29e7	1.97e7	1.80e7	1.72e7	1.69e7	1.74e7	2.00e7	1.77e7	1.57e7	9.82e6	3.00e5
2e5	8.39e7	5.83e7	4.63e7	3.96e7	3.64e7	3.44e7	3.34e7	3.34e7	3.92e7	3.79e7	4.14e7	4.81e7	5.60e7
5e5	2.10e8	1.46e8	1.15e8	9.92e7	9.19e7	8.76e7	9.79e7	1.12e8	1.21e8	1.17e8	1.35e8	1.51e8	1.53e8

Table 5.10: Runtimes of RLE encoding with DMA.

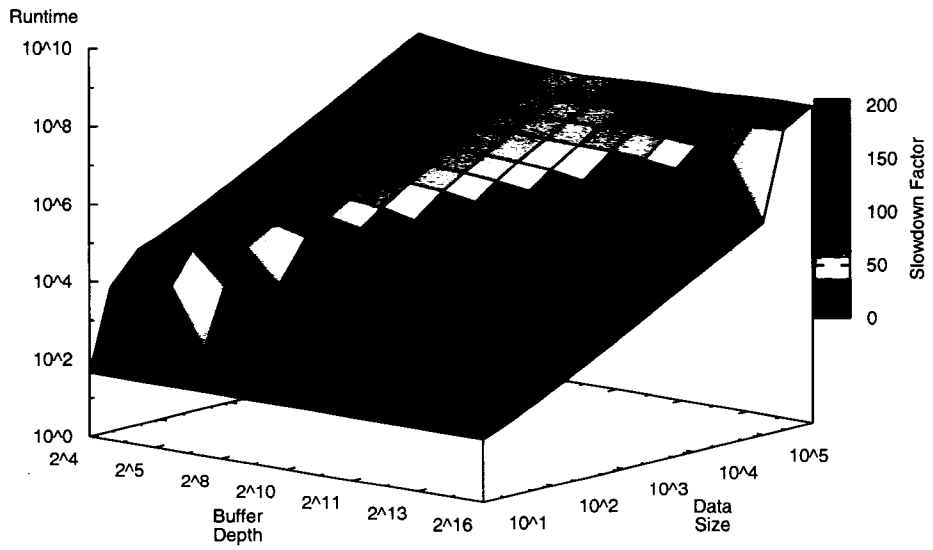


Figure 5.20: Plot of values in Table 5.10.

5.3.8 Output Streams Using DMA

Table 5.11 and Figure 5.21 show the transfer times for the RLE decoding when DMA is used. As with the input streams, the optimal non-stalling performance occurs more often with DMA than with PIO because the transfers are faster. In general, the behavior of the host in this case is even less predictable than with input streams. For example, it is not known what causes the dark red plateau in the figure. Although, in that region, the distribution of the runtimes is bimodal, and the effects of the primary Gaussian are quite small, the secondary Gaussian dominates. The change in the distribution may account for the plateau, but as to what causes the bimodal distribution in the first place, one can only speculate.

Output Size	Output Buffer Depth												
	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k
10	42	42	42	42	42	42	42	42	42	42	42	42	42
20	2352	62	62	62	62	62	62	62	62	62	62	62	62
50	50925	2366	122	122	122	122	122	122	122	122	122	122	122
1e2	1.12e5	84866	2423	222	222	222	222	222	222	222	222	222	222
2e2	2.45e5	1.91e5	1.11e5	2556	422	422	422	422	422	422	422	422	422
5e2	6.53e5	5.60e5	4.55e5	3.00e5	2890	1022	1022	1022	1022	1022	1022	1022	1022
1e3	1.33e6	1.18e6	1.05e6	8.79e5	5.84e5	3370	2022	2022	2022	2022	2022	2022	2022
2e3	2.70e6	2.42e6	2.23e6	2.03e6	1.72e6	1.14e6	7608	4022	4022	4022	4022	4022	4022
5e3	6.84e6	6.17e6	5.78e6	5.48e6	5.16e6	4.53e6	3.39e6	1.06e6	10022	10022	10022	10022	10022
1e4	1.38e7	1.24e7	1.17e7	1.12e7	1.09e7	1.02e7	8.99e6	6.89e6	2.22e6	20022	20022	20022	20022
2e4	2.79e7	2.54e7	2.40e7	2.33e7	2.30e7	2.23e7	2.09e7	1.97e7	1.44e7	4.70e6	40022	40022	40022
5e4	7.05e7	6.36e7	6.05e7	5.97e7	5.89e7	6.00e7	5.59e7	5.53e7	4.85e7	3.86e7	1.97e7	1.00e5	1.00e5
1e5	1.42e8	1.28e8	1.53e8	3.14e8	2.94e8	3.24e8	3.18e8	3.18e8	2.84e8	2.37e8	1.64e8	4.45e7	2.00e5
2e5	2.83e8	2.63e8	7.07e8	7.64e8	8.07e8	7.60e8	7.80e8	7.68e8	7.45e8	7.03e8	6.39e8	4.88e8	1.70e8
5e5	7.23e8	6.65e8	1.87e9	2.05e9	2.08e9	2.05e9	2.04e9	2.03e9	2.01e9	1.99e9	1.90e9	1.76e9	1.49e9

Table 5.11: Runtimes of RLE decoding with DMA.

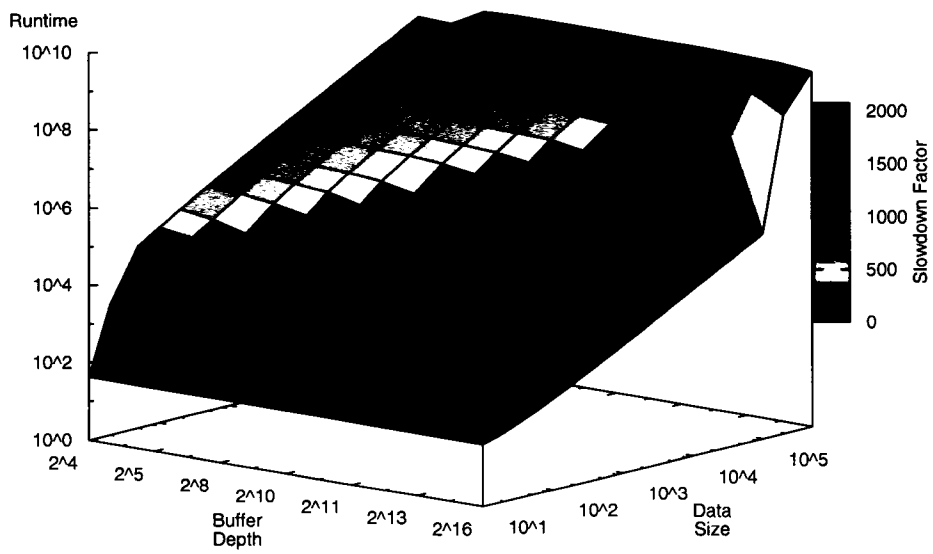


Figure 5.21: Plot of values in Table 5.11.

Chapter 6

Related Research

This chapter will discuss past and current related research topics. The AHAHA is a dataflow execution model designed for reconfigurable computing systems (RCS), and FPGAs in particular, which utilizes handshaking. In a broad sense, any RCS execution model could be deemed to be related, as could any dataflow system, or any system which utilizes handshaking for runtime synchronization. It is impossible to enumerate all such works here, so the discussion is limited to discussion of work which highlights similarities or differences with the AHAHA model.

6.1 Dynamic Solutions

There are several RCS projects which use some form of dynamic scheduling. None are identical to the model described here, but many share common features and motivations. Because these models share common ground with the ideas which motivated the AHAHA, they are discussed in detail here. Among these systems are the Compaan VHDL Visitor, the Elemental Computing Array, and the RaPiD configurable architecture.

6.1.1 Compaan

The Compaan [40, 41] project is an effort to automatically transform Matlab programs (which are imperative and sequential) into a network of concurrent process, similar to a Kahn process network [23, 24], called a "Stream-Based Functional Model," or SBF. Compaan limits its source programs to a subset of Matlab programs, which the authors

term "affine nested loop programs." This subset of Matlab is analogous to the SA-C* language, and the SBF networks produced by Compaan are analogous to the DFGs generated by the SA-C* compiler. The edges connecting the SBF processes (which are called streams), contain an implicit infinite FIFO, just like SA-C* DFGs. The processes themselves can unconditionally write to their output streams, since they are never full. However, the process must handle the possibility that an input stream is empty, and stall until data is available. The Compaan project stops at this level and does not attempt to target a physical hardware model.

T. Harriss et al. [42] use Compaan to generate process networks from Matlab code, and then implement those process networks on an FPGA using a tool called the "VHDL Visitor." In order to implement the SBF networks on the FPGA, processes are translated into VHDL components, and the streams into FIFOs with bounded buffers. It is not obvious from their papers exactly how the buffer sizes are determined. Compaan does not supply such information with the SBF network, so the information cannot be contained in the original Matlab code. Presumably, the buffer sizes are tuned by hand, or use some predefined default. They suggest that simulation can be used to determine the minimum buffer sizes required to prevent deadlock although, in general, this is not always possible. Graphs which contain cycles, or whose operation is data-dependent further complicate this type of analysis, and make it generally undecidable. It is not clear if the buffer sizes are assigned on a per-stream basis, or if all streams are set to be the same size. Regardless, once the stream sizes have been bounded, they use a method which they call "request acknowledge" or "available/request" handshaking protocol, to prevent a process from writing to a stream whose buffer is full. This protocol is nearly identical to the handshaking model used in the AHAHA, except that the VHDL Visitor generates handshaking for each process on a per-edge basis. In their model, a process may be connected to multiple streams, and the process itself is responsible for determining whether or not it should stall due to a full or empty stream. By contrast, the edges of the AHAHA graph are simply wires with no buffering whatsoever, and handshaking is

done on a per-section basis. This allows a single edge to have multiple targets, and uses far less handshaking since each section can contain multiple edges. The model used by the VHDL Visitor requires each process to consider handshaking signals for each port connected to the process, where the AHAHA model reduces this complexity by using port clusters.

6.1.2 Elemental Computing Array

The Elemental Computing Array [43] (ECA) architecture developed by ElementCXI [44] uses a model very similar to the AHAHA handshaking model to apply what they call "back pressure" into the executing dataflow graph. The back pressure can cause a portion of the graph to stall when the target(s) of its output are not able to receive data. This behavior is identical to the behavior achieved by the handshaking in AHAHA.

6.1.3 RaPiD

The RaPiD Configurable Architecture [45, 46, 47] is a novel reconfigurable platform developed at the University of Washington. RaPiD is more coarse-grained than an FPGA. It is basically a linear array of ALUs, multipliers, registers, and memories. The array is tailored to implement systolic array applications, but is not exclusively limited to this type of computation. Each processing element may implement a function which is independent of the others. The data path of the RaPiD array is controlled with a combination of static and dynamic control. The portions of the computation which remain static throughout the computation are controlled by static RAM, while the dynamic portion can be controlled by the application itself. This is similar in concept to an FPGA which has some limited control over its own routing resources at runtime. Data flows into, and out of, the array via FIFOs which serve as connections to external memories. The memory accesses required to fill or empty the FIFOs are handled externally, and are not controlled by the array fabric. In the event that the input FIFOs are empty, or the output FIFOs are full, and an access to them is requested by the array, the entire computation is stalled until the stalling FIFOs become ready. This process is similar

in design to the AHAHA handshaking model, except that when a stall occurs in the RaPiD, the entire computation stalls, and in the AHAHA, the computation dynamically identifies the portions of the computation which must be stalled. In the AHAHA, this dynamic scheduling leads to self-synchronizing behavior.

The RaPiD is an alternative to a standard FPGA, and is not a model of computation, like the AHAHA, so a direct comparison is not possible. However, it is clear that the designers of the RaPiD array have recognized the need for dynamic control and timing in reconfigurable applications. Reconfigurable architectures have a completely different niche than ASICs, and so require a more dynamic approach. The static design practices used in ASICs are not appropriate for reconfigurable systems.

6.2 Static Timing Solutions

Many RCS execution models are done using a static timing solution. When using static timing, the schedule of the computation is derived off-line, during compilation. Because the compilation takes place without prior knowledge of the runtime data, no runtime-dependent scheduling is possible. The hardware execution is prescripted so that every computational element is configured to execute at specific times. In some cases, the schedule can be parameterizable to take varying data sizes into account, but this is the extent of the flexibility. It is my belief that such systems are ignoring some of the fundamental issues of reconfigurable computing, and as a result are restricted to producing solutions for an extremely small set of regular applications.

In the general case, static timing analysis is incredibly complex, but nevertheless, ASIC design is typically done this way. This is possible because the design phase of ASIC design is substantially longer than the design time for most RCS systems, and the behavior of the ASIC is fixed. An ASIC is designed over a period of many months, and the timing behavior of the circuit can be manually analyzed, tuned, tested, and adjusted ad infinitum. By contrast, reconfigurable systems attempt to provide a robust, reliable solution in minutes, and must be able to operate automatically with a minimum

of human intervention. Additionally, RCS systems are, by definition, more dynamic than ASICs, and are not expected to be as efficient. Using an ASIC design methodology for RCS systems is very restrictive, and inappropriate for general solutions. In order to produce a completely static timing model automatically, the set of source programs must be significantly reduced to describe programs with analyzable structure.

6.2.1 MMAlpha

ALPHA [48, 49, 50, 51] is a single-assignment language, developed in France at IRISA, which uses systems of affine recurrence equations to define algorithms. "Algorithms may be represented at a very high level in ALPHA, similar to how one might specify them mathematically. The ALPHA language is somewhat restrictive in the class of algorithms it can easily represent, but is useful for programming algorithms which have a high degree of regularity and parallelism such as linear algebra and digital signal processing applications."[52].

MMAlpha [53, 54] is a tool created in Mathematica and C which accepts algorithms described in ALPHA, and applies transformations on them to derive equivalent descriptions which may be suitable for implementation on processors, parallel systems, or RCS platforms. One of the features of the MMAlpha tool is its ability to convert a subset of the ALPHA language into systolic array descriptions, which are then implemented in VHDL. Because the source language (which is a subset of the already restrictive ALPHA language) and the target platform (Systolic Arrays) are so limited, static timing is possible. The generated VHDL implementation contains simple state machines which control the systolic array, combined with systolic control signals throughout the array. MMAlpha produces efficient systolic designs for the set of algorithms which it can implement, but that set is quite small.

6.2.2 PICO

The PICO [55] system, developed by Hewlett Packard, attempts to convert loop-nest functions in C programs into hardware accelerators in VHDL. The authors describe this

process as "Program-In-Chip-Out," hence the name "PICO." PICO has strict limitations on the types of acceptable functions it can implement. Specifically, it is restricted to loops with constant bounds, affine array references, and uniform data-dependences. Such loops are transformed into systolic array implementations, which can then be implemented in hardware. Like MMAlpha, the restricted nature of the source language and target model make static timing possible.

6.3 Control-Path Solutions

Many solutions are available which use a simplified form of handshaking which operates in the same direction as the data flows. In this unidirectional handshaking model, the elements in the computation pass a token between them which identifies when they should execute. The token follows a control-path through the graph. Many such solutions treat the control-path token as a one-hot value, meaning that only one such token can exist in the control path at any time. This leads to sequential execution, and as such, is most often used for systems which are programmed using a sequential language. Some variations on this method are able to provide pipeline parallelism, or independent parallel sections by allowing multiple control-path tokens to exist simultaneously. What these solutions all have in common is that no element in the design is able to refuse the token when it is offered. A computational element is able to pass its token to another element, but until it is passed, the receiving element must stall. A producer is able to make the consumer stall, but a consumer is not able to make the producer stall. In this way, it is unidirectional handshaking. Care must be taken to make sure that areas of the computation which operate at a slower rate than the preceding area do not become overwhelmed by input arriving too quickly.

6.3.1 PipeRench

PipeRench [56] was a project developed at Carnegie Mellon University to create a reconfigurable chip to exploit pipeline parallelism. It utilizes a control-path solution where multiple tokens along the same path are permissible (and even desired). PipeRench was

subsequently licensed by Rapport Inc., and is the basis of the Kilocore [57] chip, which is being jointly developed by IBM. The PipeRench chip is much more coarse-grained than an FPGA, and enforces a unidirectional dataflow. PipeRench is programmed using a dataflow language called DIL, which is then mapped directly to the chip. The chip itself contains a pipeline of SIMD processors, which implement the described algorithms. The architecture of the PipeRench is limited only to implementing programs which exhibit Pipeline and SIMD parallelism. The chip uses unidirectional, synchronous dataflow. The program must be mapped so that it contains no bottlenecks which would cause the processing elements on the chip to stall.

6.3.2 Celoxica DK

The Celoxica DK design suite [58] utilizes the one-hot control-path method for synchronization, with exceptions to allow parallel sections to occur. The control-path token can be split to trigger several independent computations in parallel. The multiple tokens are then merged together again after each section has completed. The behavior is quite dynamic, but does not provide control for pipeline parallelism. It is based on Communicating Sequential Processes (CSP) [59]. Their approach follows from the fact that the Handel-C language is based on C (which is inherently sequential), and Occam [60] (which provides a method of sharing data between multiple processes executing in parallel). To identify parallel sections, the language allows user supplied annotations which control the parallelism explicitly. This information is based on the Celoxica DK1, because information regarding the subsequent DK2 and DK3 systems is unavailable. However, because the Handel-C language is still sequential by nature, it cannot have deviated very far from the initial DK1 design methodology.

6.3.3 SRC MAP-C

Like Celoxica and Handel-C, the SRC-6 [61] system allows FPGA co-designs to be generated from a variant of C called MAP-C. However, its synchronization model uses a hybrid approach to extract parallelism. Its inner loops rely heavily on pipeline paral-

lelism, while outer loops use a one-hot control-path method for synchronization. The control-path is used to trigger pipelined basic blocks but, internally, the components within the basic blocks execute using dataflow semantics to allow limited parallelism. The basic blocks of the innermost loops are combined and optimized to exploit pipeline parallelism. These inner loops are initiated when the control-path token arrives at the top of the loop. When the loop completes execution, it passes the control-path token to the next basic block. Streams are available which allow multiple processes to execute concurrently in a producer-consumer fashion. Typically, streams are used to connect processes which operate completely independently from one another, allowing them to operate at their own rate. This is not the case for SRC streams. Once a consumer process has begun reading from a stream, it must continue to read a value in every clock cycle where one is available. It is not allowed to refuse an element from the stream. The producer may choose whether to enqueue an element in the stream or not but, once it has read the first element, the consumer has no choice but to continue dequeuing. In this way, the behavior of the producer may cause the consumer to stall, but not vice versa.

6.3.4 Forge

The Forge [62] project at Xilinx aims to compile Java source code to hardware. The Forge compiler shares many similarities to the SA-C* compiler, including many of the same optimizations (Constant Propagation, Bit-Width Analysis, Loop Unrolling, Pipelining), and strategies. However, since SA-C* is a language with parallel semantics, and Java has sequential semantics, Forge must attempt to extract the parallelism from the code, where the SA-C* compiler often has more parallelism available than can be exploited in the hardware given finite chip area. Forge uses a one-hot control-path model. They call the control-path token a "Go" signal. The Go signal propagates through the design in the same fashion as the data. Each component accepts a Go signal, and produces a "Done" signal, to indicate that it has finished its execution. The Go signal cascades through the design sequentially from component to component. When parallelism has

been identified by the compiler, the token is split to follow multiple paths simultaneously, and then merged together after the parallel sections are complete. A major assumption of the Forge design is that the elements of the design are simple atomic operations which can begin and finish a computation in each time unit. Each operation is assumed to have a one-to-one correspondence between its inputs and outputs. Because of this restriction, there is no need for full bidirectional handshaking, and the unidirectional Go signal is used instead.

Chapter 7

Conclusions and Future Work

This chapter will discuss the conclusions and insight gained through this work, and will provide some potential directions where future work may be done. It is unlikely that the Cameron Project, and the SA-C* compiler will continue to be developed, but similar projects may expand or build upon it. For this reason, no explicit future work within the context of the SA-C* compiler is discussed. Instead, potential areas of research for future projects will be suggested based on the lessons learned in this work.

The AHAHA model and its VHDL implementation have many attractive features which makes it a desirable approach to implementing FPGA-based reconfigurable computing systems. It provides a powerful blend of performance and flexibility which make it attractive for implementing high-level languages on hardware. It has been shown that its overhead is low, and its benefits are substantial.

Prior approaches to reconfigurable computing often rely heavily on simplifying assumptions. For example, many systems require that the building blocks used in the hardware design have completely known runtime behavior (such as Forge, SRC, and Celoxica), so that they may be fully analyzed and timed. Others assume that the computations being implemented are very regular (PICO, and MMAlpha). The AHAHA approach can implement the same designs as these other approaches, but is more robust, and is able to execute efficiently when the runtime behavior of the design is more dynamic. For example, operations with complex I/O behaviors can make static analysis difficult or impossible, such as the behaviors described by AHAHA nodes with complex

firing rules; the FIFO_UNPACK, SHIFT_REGISTER, MERGE, and SPLIT nodes all would complicate static analysis, as would memory operations with possible contention, and all stream-based operations.[1] The AHAHA allows the compiler to optimize the execution, but still provides a mechanism for handling these dynamic behaviors. The AHAHA also allows the compiler to interrogate the components in the design, allowing it to make decisions to further tune the performance.[2] The AHAHA model can provide implementation details to the compiler, such as frequencies, and area utilization estimates of each node. The compiler can use these estimates to perform limited automatic design space exploration, automated pipelining, and to provide utilization estimates.

In order to operate in a dynamic fashion, the AHAHA uses handshaking to synchronize the computational elements. The overhead for this handshaking is quite minimal, when compared against the flexibility that it affords. The AHAHA handshaking methodology allows programs to exploit pipeline parallelism in loops. It also provides fine-grained parallelism in the form of sections (each node in a single section executes concurrently). Coarse-grained parallelism can be obtained through partial loop unrolling, and process level parallelism via streams. Because the AHAHA execution environment provides its own scheduling, the compiler does not need to provide a complete end-to-end timing solution. The compiler can choose to optimize the schedule as much as it is able, to provide efficient execution with minimal loop slowdown. However, in the event that some portion of the program cannot be pipelined, or gives rise to a highly dynamic runtime behavior, the compiler is able to rely on the AHAHA model to schedule it correctly at runtime. Systems with more rigid scheduling rules simply will not be able to produce a solution in some situations. For example, MMAlpha and PICO programs without an affine schedule will not produce VHDL code. Portions of Java code whose

---

[1]The term "stream" here refers to streams which may become full, and require a producer to stall, not to the SRC interpretation in which the consumer must read continuously whenever data is available.

[2]The SA-C* compiler does not take advantage of this feature, but support for it is present in the AHAHA framework.

dataflow structure cannot be completely analyzed will not be implemented in hardware by Forge. In some cases, these irregularities in a program structure are unavoidable and a fully pipelined, pre-scheduled implementation will not be possible. In these situations, a dynamically scheduled design is preferable to no design at all.

The dynamic AHAHA scheduling allows the compiler to do analysis of a graph using a simplified model of the nodes. For example, consider a `FIFO_PACK` node which is parameterized to pack 8-bit values into 32-bit words. In the steady state, its behavior is quite regular and predictable: It will produce an output token for every 4 input tokens. However, when considered at a more detailed level, its behavior can be quite complex. For example, when a last-flag arrives, its behavior and I/O characteristics depend on the status of its internal buffer. If the internal buffer is empty, the node can immediately produce its own last-flag, and accept new input tokens immediately. However, if the buffer is partially full, it must first produce the partial word it has assembled, and then follow it with a last-flag. This delay may cause the data path which feeds the node to stall, or it may not, if buffering is present in the graph. If the compiler were to statically analyze this, these minute details would cause unneeded complexity. If instead, the compiler simply optimizes the loop taking into account only the steady state behavior of each node, the details of the execution will be handled by the dynamic scheduling, but the compiler can still optimize performance. The AHAHA scheduling allows the compiler to optimize for performance without needing to schedule each individual operation and data transfer.

However, great care must be taken not to depend too heavily on dynamic scheduling. One-hot control-path solutions are highly dynamic, but are contrary to hardware execution paradigms. They are able to implement parallelism as an exception and, by default, are sequential. In order to outperform processor-based solutions, hardware design must leverage massive parallelism to overcome their slower clock frequencies. Loops should be analyzed by the compiler to expose as much parallelism as possible to the underlying implementation. The AHAHA model allows a good trade-off between performance

and dynamic execution. When the compiler constructs an AHAHA graph which is fully pipelineable, the AHAHA will dynamically pipeline it, obtaining the same performance as a statically timed solution with minimal overhead. But when a fully pipelined implementation is unavailable, the AHAHA will dynamically identify the best schedule possible, taking the dataflow characteristics into account, and will often outperform a one-hot control-path solution.

The AHAHA methodology requires great discipline on the part of the compiler writer and the user. Just because something is possible to implement in the AHAHA framework, does not imply that it will be efficient. For example, if one were to implement floating-point operations using this model, sequential operators would be available as well as pipelined ones. Both would be supported using the handshaking model, but the performance and area characteristics would differ greatly. A pipelined solution would have much higher throughput and area, while the sequential implementation would have smaller area and performance. One of the advantages of the AHAHA model is that the compiler is not forced into one implementation or the other, but care must be taken to select the best one for the given circumstances. For example, in a situation where a floating-point operation is present within a pipelineable innermost loop, a pipelined operator is preferable. However, in an outer loop which executes at a slower rate, a sequential solution may be tolerated without slowing down the overall computation. In situations where the implementation decisions cannot be made entirely automatically, feedback from the compiler to the user can be used to request some user intervention.

One area of possible research is to identify situations where the full AHAHA handshaking model is not required. In these situations, it may be possible to modify the structure of the handshaking control signals to more closely resemble a unidirectional control-path. This will remove some of the handshaking overhead, but the potential for misuse is great. By allowing the compiler to manipulate the handshaking signals, the AHAHA is no longer able to guarantee that the execution of the graph will follow dataflow semantics, and tokens may be lost. Alternatively, the entire AHAHA graph could be

analyzed as a whole, and if no nodes are present which may lead to dynamic behavior, alternate, simplified handshaking may be employed. However, it is my assertion that the overhead for the handshaking is so minimal, and situations where completely static scheduling is possible are so rare, that this analysis is not worth the effort.

The AHAHA model of execution provides a robust solution, so that nearly any algorithm can be implemented in hardware. It also provides many opportunities for optimization, and methods to extract performance. I believe that the AHAHA architecture is an ideal solution for implementing reconfigurable computing systems.

Appendix A

AHAHA Node Details

For a full and detailed understanding of the design decisions made in the AHAHA framework and the AHAHA to VHDL translator, detailed descriptions of the AHAHA nodes are included in this appendix.

A.1 Special Nodes

Special nodes are nodes which provide information to the AHAHA to VHDL translator to influence the generated hardware, but do not have any VHDL descriptions directly associated with them. They either provide supplemental information about the graph, or provide an interface to the host.

A.1.1 INPUT

Input nodes are used to get information from the host interface into the graphs. Input nodes have no inputs, and a single output which can be of any bit-width. The output is supplied by the on-chip register file which contains the runtime constants written into the FPGA by the host. The node is implemented by a signal assignment from the appropriate cell of the register file to the output of the `INPUT` node. For example, `INPUT` nodes may be used to supply the following.

- **Array Locations**: Addresses where arrays are stored in the memories present on the RCS board. These addresses are allocated by the host process at runtime. The host then supplies these addresses to the AHAHA graph (via an `INPUT` node)

so that the nodes which access these arrays can locate the regions in the memory where they are stored.

- **Iteration Counts**: Loops come in 2 types: for-loops and while-loops. While-loops are slightly less efficient than for-loops because they require (at least) 1 additional clock cycle to evaluate the test of the while-loop before the body executes. For-loops, on the other hand, can be analyzed to determine exactly how many times the inner-loop body will need to be executed. The number of iterations of a for-loop is often dependent on the size of an array supplied at runtime. In these cases, the iteration count of the loop is supplied by an `INPUT` node.

- **Word Counts**: For loops, which are driven by a "for substructure in structure" type of for-loop, the hardware must read in the words which contain the larger structure from the memory. By supplying the number of words in the structure (via an `INPUT` node), it is possible to generate efficient hardware which will read an entire region of memory sequentially.

- **Runtime Constants**: Oftentimes, the SA-C* code will contain a small runtime constant which is passed into the hardware portion of the code. When this happens, the value is provided by an `INPUT` node.

A.1.2 OUTPUT

`OUTPUT` nodes are used to signal the host when the hardware execution is completed. Each AHAHA graph must contain exactly 1 `OUTPUT` node. The output node has a single 1-bit input, and no outputs. This node is implemented by attaching its input to the host interface foreman. A 1 arriving on the input indicates that the graph has finished executing, and the host interrupt should be triggered. It also causes the foreman to stop counting clock cycles, so that the clock-cycle value read by the host will reflect the total runtime of the graph. A 0 arriving on the edge is ignored.

A.1.3 STREAM_CREATE_INPUT

The STREAM_CREATE_INPUT nodes do not describe a computational hardware component, or have operational semantics like most AHAHA nodes. Instead, they parameterize the graph.[1]. The existence of this node creates an input stream, which is written to by a process (or file) on the host, and read by the AHAHA graph. The AHAHA graph must not contain any stream operations which write to this stream. This node causes both the stream itself to be created, as well as a stream bridge, which is parameterized to allow write operations from the host but not read operations. STREAM_CREATE_INPUT nodes have 4 inputs. The first 3 inputs must be compile-time constants:

0. **Stream ID**: This indicates which stream the node is declaring. The stream ID is used by the nodes which perform stream operations to identify which stream they are operating upon. Beyond providing a method mapping back and forth between the stream and the nodes which use it, this value does not have a physical impact on the hardware construction.

1. **Depth**: Declares the requested depth of the stream. This indicates the minimum number of elements that the stream must be able to store at one time. This value may be supplied by user instructions in the SA-C* code. If the user does not specify it, the compiler must provide a default (currently 16). The actual stream may be able to store more than the requested number of values. This value is also used to determine what type of storage will be used to implement the stream. Streams containing less than 512 values[2] are implemented using FIFOs created from SRLC16E components, which use the LUTs in the FPGA as a 16-

[1]For consistency, the information provided by the STREAM_CREATE_INPUT, STREAM_CREATE_OUTPUT and STREAM_CREATE_LOCAL nodes should have been implemented by a pragma on the top-level wrapper node which encapsulates the AHAHA graph. However, the decision to use these nodes instead of pragmas was made by the SA-C* compiler developer, who was not available to alter it.

[2]This value is "USE_BRAM_STREAMS" #defined at the top of ahaha_to_vhd.c, and can be changed. 512 was chosen empirically as the point where SRL streams begin to become impractical.

bit selectable shift register. FIFOs larger than 16 are implemented by cascading multiple SRLC16Es together and selecting the correct value with a MUX. Streams larger than 512 are implemented with BlockRAMs, used as circular queue. Because there is a single clock cycle of latency between a BlockRAM read request and when the data is available, the "next" value in the FIFO is pre-fetched into a double buffer, to provide instant access as is expected by the stream operators.

2. **Width**: Declares the bit-width of the elements which are stored in the stream.

3. **Unknown**: This value comes from an INPUT node. It is always a 32-bit wide value of all 0s. It is always ignored.[3]

STREAM_CREATE_INPUT nodes have no outputs, or pragmas associated with them.

A.1.4 STREAM_CREATE_LOCAL

Like STREAM_CREATE_INPUT nodes, STREAM_CREATE_LOCAL nodes are used to declare the existence of a stream.[1] In this case, however, it is a local stream, which is both written and read inside the AHAHA graph, and does not need an associated stream bridge (because the host is not involved). It is used to connect multiple communicating processes on the hardware. It allows the processes to run independently, with data flowing between them in a non-strict manner. STREAM_CREATE_LOCAL nodes have the same inputs as STREAM_CREATE_INPUT nodes in the same order, no outputs, and no pragmas.

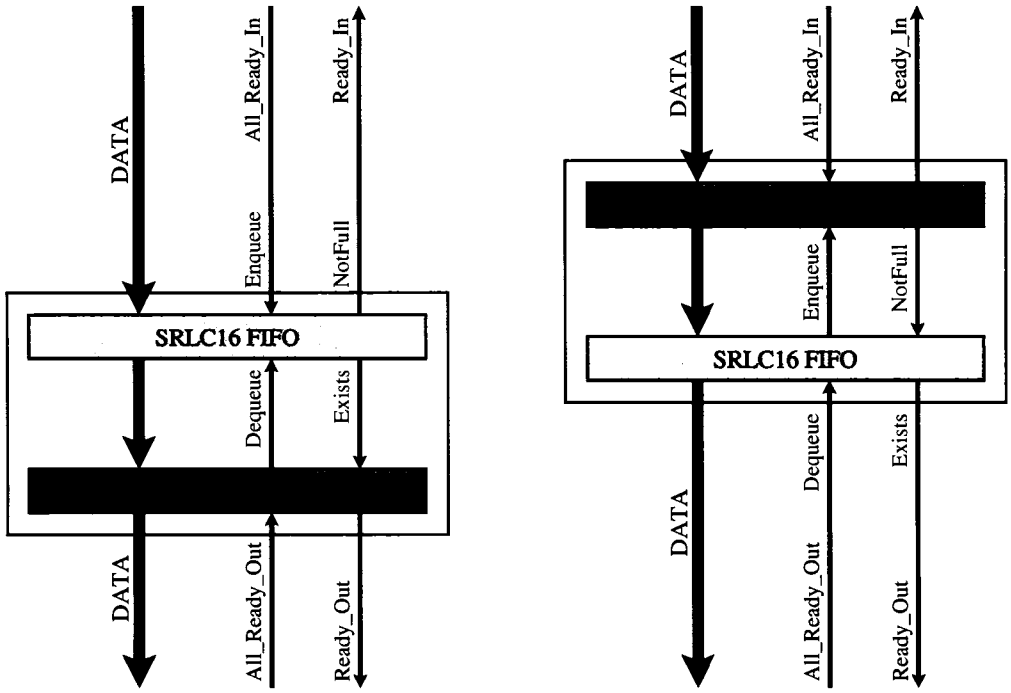
A.1.5 STREAM_CREATE_OUTPUT

Like the other STREAM_CREATE nodes, STREAM_CREATE_OUTPUT nodes declare a stream[1]. In this case, it is an output stream which is written to by the AHAHA graph and read by the host. The stream bridge associated with it allows read operations by the host but not write operations. STREAM_CREATE_OUTPUT nodes have the same inputs as STREAM_CREATE_INPUT nodes in the same order, no outputs, and no pragmas.

---

[3]This value is apparently needed in the higher levels of the compiler in order to enforce a data dependence, but is not needed (or used) at the AHAHA level.

A.2 Clocked Nodes



a: FIFO on Input ports.

b: FIFO on Output ports.

Figure A.1: SRL16E FIFOs embedded in clocked nodes.

There are 32 clocked nodes available in the AHAHA framework. They are the workhorses of the AHAHA graphs. They control the handshaking, contain all of the state in the graph, and create all of the more complex behaviors. When designing clocked nodes, several details must be taken into account.

First, there must not be any combinational paths from the handshaking signals of any input clusters and the handshaking signals of any output clusters, or vice versa. If such a path exists, it may cause the firing of one section to be dependent on the firing of another section. This would defeat the motivation of sections. In addition, if a large number of nodes contain such paths, then the possibility of extremely long path delays

in the handshaking logic becomes greater, and they may become the critical path of a design, as they did in the old ad hoc AHA model.

Second, whenever possible, the nodes should be designed so that the input and output sections fire simultaneously, once steady state has been reached. These 2 ideas seem to contradict each other, but it can usually be achieved by adding a small SRL16 FIFO to either the input ports or output ports, as shown in Figure A.1. This adds an extra clock cycle of latency, which is usually not a significant issue, due to software pipelining. As an extra benefit, the internal FIFO allows some internal buffering in the node so that when a node in the graph stalls momentarily, there is less impact on the overall performance..

Most clock nodes utilize "last-flags" in some form. Nodes use the ports `Last_In` and `Last_Out` to communicate the flag between nodes. When the arriving flag is 1, it indicates that the data arriving with the flag in that clock cycle (if any) is not valid data, and instead represents the end of a sequence. It instructs the node to treat the current value differently (usually by ignoring it), and to flush any internal state the node may have and return to its initial state. Most nodes also produce a single garbage value, with the `Last_Out` output set to 1 to indicate that the data that a node is producing is not valid data, but instead represents the end of the preceding sequence, so that the nodes below can also reset to their initial condition.

Any nodes which perform stream operations have an additional flag, called the "stream-flag" which is used to serialize a sequence of stream operations. Stream nodes use the ports `Flag_In` and `Flag_Out` to communicate the flag between nodes. Strictly speaking, because stream operations side effect the stream, all stream operations must be executed in the order in which they appeared in the initial SA-C* code. This would seem to indicate that streaming codes are largely sequential. However, in practice, much of this serialization may be relaxed. Each stream may have its stream operations serialized independently, because an operation on one stream will not have an affect on any other stream. Likewise, each group of stream operations may be divided into 2 groups which may be serialized independently. The first group consists of operations

which affect the input of the stream (put, finalize, and close), while the second group contains operations which effect the output of the stream (get, peek, and is_open). As long as the operations in each group occur in the correct order, the program will perform according to the SA-C* semantics. The flag originates with a `STREAM_INIT_FLAG` node (which simply produces a constant 0). The flag typically goes into a `CIRCULATE` node (or one of its derivatives), and then is threaded through all of the stream operations which are to be serialized, and then back into the circulate node to complete the loop. When the flag is 0, the node is permitted to perform an operation on its stream; when the flag is 1, it indicates that the stream should not be touched. In this way, the stream-flag is similar to a last-flag. In addition to the stream-flag, all stream operation nodes contain an input that identifies the stream it operates on, which must be a constant value[4] The preamble of the VHDL file must copy this constant value to the variable `stream_id`, so that the AHAHA to VHDL translator can attach the correct stream to the node. Other than assigning it to `stream_id`, the value must not be used inside the stream operation itself.

A.2.1 ADDR_CALC

`ADDR_CALC` nodes are used to generate sequences of memory addresses in a very particular way. They are given a Base address (B), a Step (S) and a Depth (D), and produce a sequence of D addresses ($A_i := B + (i \cdot S), i := 0 \ldots D - 1$). This allows addresses to be generated in column major order, instead of the more typical row major order which requires only linear addresses from a `COUNTER`. In SA-C*, these nodes most often arise as a result of a pragma requesting vertical stripmining of a 2-D loop. The addresses from the `ADDR_CALC` are intended to be fed into a `READ_WORD_MEMORY` node, but could easily be used for other purposes.

---

[4]In retrospect, this value should have been defined using a pragma instead of an input, but no negative repercussions resulted from this design decision, so it has not been changed.

ADDR_CALC nodes have 3 inputs and 2 outputs.

Inputs:

0. **Base_In**: B in the equation above.

1. **Step_In**: S in the equation above. If this value is a compile time constant (which it often is), then the VHDL implementation will take this into account, and optimize the node to remove registers for holding it, and use an adder with a constant value. If the low bits of the constant are 0s, the synthesis tools will further optimize the adder to remove the low bits.

2. **Last_In**: Last-flag, described in Section A.2.

Outputs:

0. **Addr_Out**: The sequence of addresses being generated; A_i in the equation above.

1. **Last_Out**: Last-flag, described in Section A.2.

ADDR_CALC nodes have 1 mandatory pragma:

- **DEPTH**: D in the equation above.

A.2.2 BUFFERX

The BUFFERX node is perhaps conceptually the simplest of the clocked nodes. It has only 1 input and 1 output, named "Data_In" and "Data_Out," respectively. The node is simply a FIFO. The BUFFERX node is most often used for buffer balancing the AHAHA graphs. It has 1 mandatory pragma called "DEPTH" which declares the minimum depth of the FIFO. There are 4 styles of this node. The first, called "buffered1" uses a sequence of SRL16E FIFOs connected together in sequence; the buffer depth is rounded up to the nearest multiple of 16. The latency is the number of SRL16E FIFOs needed, since it takes 1 clock cycle for a value to transition from one FIFO to the next. Buffered1 is intended for Virtex chips which support SRL16Es but not the SRLC16Es. The second style, "buffered2," uses SRLC16Es to create FIFOs deeper than 16 elements which still

only have a latency of 1 clock cycle. The third style, "buffered3," is only used when the DEPTH is 1, and consists of a simple double buffer constructed from registers. In steady state, it can only hold 1 item, but when stalling, it can hold 2. This style is primarily intended for use in pipelining. The final style, "clocked," is no longer in use. It uses more chip area, has a higher latency, and slower maximum frequency than any of the other buffered styles. The clocked style contains the type of buffer used in old AHA graphs. It is included in AHAHA only for historic reasons, and should not be used.

A.2.3 CIRCULATE

The `CIRCULATE` node was discussed briefly in Section 3.2. It is used to allow circularity in the graph. Specifically, it is intended for nextified loop variables, while-loop condition variables, and accumulation operators such as sum, or max. The `CIRCULATE` node has 2 styles "clocked" and "buffered." The behavior of the clocked style is described in the state machine in Figure A.2.

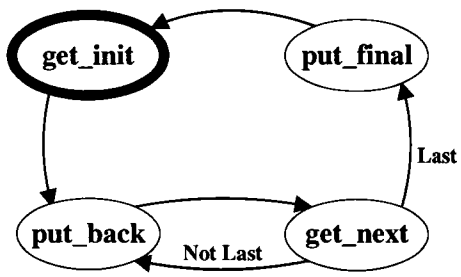


Figure A.2: FSM of a CIRCULATE node.

In the **get_init** state, the node waits for input cluster 0 to be ready to fire, stores the `Init_In` value, and moves to the **put_back** state. In the **put_back** state, when output cluster 1 is ready to fire, it drops its current value on the `Back_Out` port, and moves to **get_next**. In the **get_next** state, when input cluster 1 fires, if the `Last_In` is true, then it ignores the `Val_In`, and moves to the **put_final** state; otherwise it will move to the **put_back** state (accepting a new value from `Val_In` if `Dummy_In` is false).

The buffered style is an optimization which makes the `CIRCULATE` node more efficient when the node is used in a situation where there is actually no circularity present in the

graph. The node contains 2 FIFOs on the outputs of the node (as in Figure A.1 b), one FIFO for the circulated value, and one for the final value. The FSM is simplified to have only a **get_init** and **get_next** which take the init value or next value, respectively, and enqueue them into the appropriate FIFO. When there is a circularity in the graph, this style is not any less efficient, and the buffers will only contain a single value. When there is no circularity, this node allows more efficient pipelining than the clocked style. The state machine returns to the **get_init** at the end of each sequence.

CIRCULATE nodes have 4 inputs, in 2 clusters. Input 0 is in cluster 0, the remaining inputs are in cluster 1. This is because Input 0 only arrives once at the beginning of each sequence, and inputs 1, 2, and 3 arrive once for every circulation:

0. **Init_In**: The initial value to produce to begin the loop. This value is produced immediately on the Back_Out port.

1. **Val_In**: The value to use in the next iteration. Ignored if Last_In or Dummy_In is true.

2. **Dummy_In**: Indicates that the value on Val_In should be ignored. However, it still advances to the **put_back** state, causing the node to produce the most recent value a second time. Ignored if Last_In is true.

3. **Last_In**: (Cluster 1) last-flag, described in Section A.2. Causes the node to produce the most recent value on the Final_Out port.

Each of the 2 outputs of the CIRCULATE are in their own cluster. This is because output 0 only produces 1 token per sequence, and output 1 produces 1 for every circulation.

0. **Final_Out**: The final value of the sequence. Used primarily for reduction operators.

1. **Back_Out**: The value used in the next iteration. This edge is the edge which causes the circularity; it provides a value to a higher point in the graph.

CIRCULATE nodes do not have any pragmas associated with them.

A.2.4 CIRCULATE_CONST

The CIRCULATE_CONST node behaves similarly to the CIRCULATE node, except that the Init_In has been removed and replaced with a pragma (instead of a port, because it must always be a constant, as described in Section 3.4.1). This allows several small optimizations to the node. The node also has 2 styles "clocked" and "buffered." The state machine for the clocked style is shown in Figure A.3.

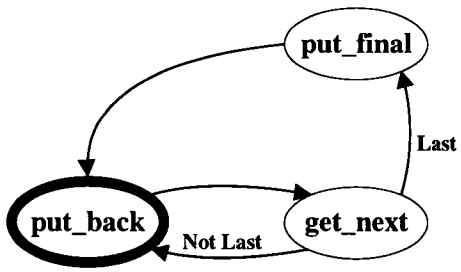


Figure A.3: FSM of a CIRCULATE_CONST node.

The buffered style is similar to the buffered style of the CIRCULATE, except that the node only ever executes the **get_init** state once, since the node is able to enqueue the final value, and the constant init value at the same time. This makes the node more efficient between sequences.

The number of input clusters is reduced from 2 to 1, since all of the ports in the first cluster have been removed. This allows the node to simplify its logic slightly, and remove at least 1 wasted cycle between the end of one sequence and the beginning of another.

The CIRCULATE_CONST node has 3 inputs:

0. **Val_In**: Same as in the CIRCULATE node.

1. **Dummy_In**: Same as in the CIRCULATE node.

2. **Last_In**: Same as in the CIRCULATE node.

The outputs of the CIRCULATE_CONST node are identical to those of the CIRCULATE node, but the CIRCULATE_CONST node has a single mandatory pragma:

- **INIT**: The constant initial value to produce at the beginning of each new loop sequence.

A.2.5 CIRCULATE_LOCAL

The CIRCULATE_LOCAL node is another variant on the CIRCULATE node where it is assumed that input cluster 1 and output cluster 1 are participating in the same section (if this is not the case, the node will malfunction). This assures that a new value will be arriving at the same time it is producing a value. This node has both a "clocked" and "buffered" style. The simplified state machine for the the clocked style is shown in Figure A.4.

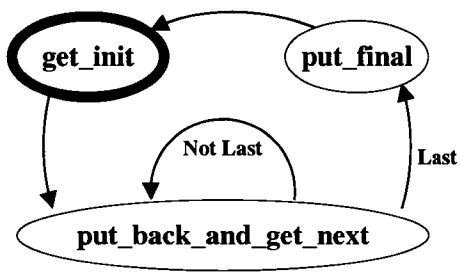


Figure A.4: FSM of a CIRCULATE_LOCAL node.

The buffered style of this node is identical to the CIRCULATE buffered style. In fact, it is implemented in the same .vhd file, since the _LOCAL optimization has no effect on the buffered style.

This optimization allows a loop to execute in a minimum of 1 clock cycle per body, instead of a minimum of 2 clock cycles for a CIRCULATE node.

The inputs, outputs, and pragmas are identical to the CIRCULATE node.

A.2.6 CIRCULATE_LOCAL_CONST

The CIRCULATE_LOCAL_CONST node is a combination of the optimizations present in the CIRCULATE_CONST and CIRCULATE_LOCAL nodes. It is the most optimized of the

CIRCULATE variants. It has "buffered" and "clocked" styles. Its state machine of the clocked style is shown in Figure A.5.

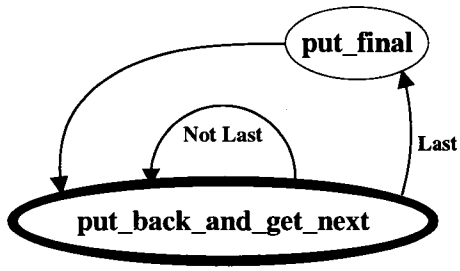


Figure A.5: FSM of a CIRCULATE_LOCAL_CONST node.

The buffered style of this node is identical to the CIRCULATE_CONST buffered style. In fact, it is implemented in the same .vhd file, since the _LOCAL optimization has no effect on the buffered style.

The inputs, outputs, and pragmas of the CIRCULATE_LOCAL_CONST node are identical to the CIRCULATE_CONST node.

A.2.7 COUNTER

The COUNTER node is used to produce a sequence of values, similar to the ADDR_CALC node, except that the number of values in each sequence is not a compile time constant. It is given an initial value (B), an increment value (S), and produces a sequence of values ($A_i := B + (i \cdot S)$). The end of a sequence (and the beginning of another) is identified using the Last_In port. The behavior of the COUNTER node is described in the state machine in Figure A.6.

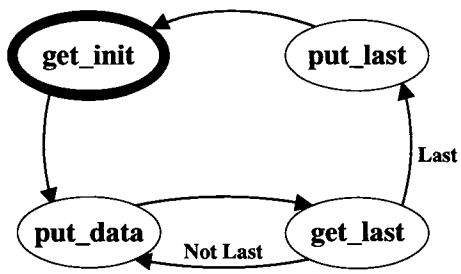


Figure A.6: FSM of a COUNTER node.

In the **get_init** state, the node waits for input cluster 0 to be ready to fire, stores the `Init_In`, and `Incr_In` values, and moves to the **put_data** state. In the **put_data** state, when output cluster 1 is ready to fire, it drops its current value on the `Data_Out` port, increments the counter, and moves to **get_last**. In the **get_last** state, when input cluster 1 fires, if the `Last_In` is true, then it moves to the **put_last** state; otherwise it moves to the **put_data** state.

`COUNTER` nodes have 3 inputs in 2 clusters. Inputs 0 and 1 are in cluster 0, and input 2 is in cluster 1. This is because inputs 0 and 1 only arrive once at the beginning of a sequence, and input 2 arrives once for every element in the sequence:

0. **`Init_In`**: B in the equation above.

1. **`Incr_In`**: S in the equation above. If this value is a compile time constant (which it often is), then the VHDL implementation will take this into account, and optimize the node to remove registers for holding it, and use an adder with a constant value.

2. **`Last_In`**: Last-flag, described in Section A.2.

`COUNTER` nodes have 2 outputs in a single cluster.

0. **`Data_Out`**: The sequence of values being generated; A_i in the equation above.

1. **`Last_Out`**: Last-flag, described in Section A.2

`COUNTER` nodes have no pragmas associated with them.

A.2.8 COUNTER_CONST

The `COUNTER_CONST` node behaves similarly to the `COUNTER` node, except that the `Init_In` and `Incr_In` have both been removed and replaced with pragmas. This allows several small optimizations to the node. The state machine is simplified as shown in Figure A.7.

The number of input clusters is reduced from 2 to 1, since all of the ports in the first cluster have been removed. This allows the node to simplify its logic slightly, and remove at least 1 wasted cycle between the end of one sequence and the beginning of another.

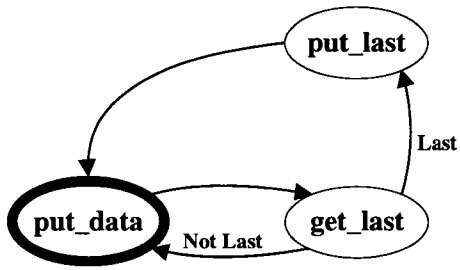


Figure A.7: FSM of a COUNTER_CONST node.

The COUNTER_CONST node has only 1 input:

0. **Last_In**: Last-flag, described in Section A.2.

The outputs of the COUNTER_CONST node are identical to those of the COUNTER node, but the COUNTER_CONST node has 2 mandatory pragmas:

- **INIT**: The constant initial value to produce at the beginning of each sequence (B in the equation above).
- **INCR**: The constant increment value to add to each value in the sequence (S in the equation above).

A.2.9 COUNTER_LOCAL

The COUNTER_LOCAL node is another variant on the COUNTER node where it is assumed that input cluster 1 and output cluster 0 are participating in the same section (if this is not the case, the node will malfunction). This assumption assures that a new last-flag will be arriving at the same time the node produces a value. It is also allowed to pass the Last_In flag directly through to the Last_Out port, since this will not result in a combinational path between 2 sections (both ports are in the same section). The state machine is simplified to the one shown in Figure A.8. The **running** state is the combination of the **put_data** and **get_last** states.

This optimization changes the behavior so that a counter is able to produce a new value every clock cycle instead of every other clock cycle for a COUNTER node.

The inputs, outputs, and pragmas are identical to the COUNTER node.

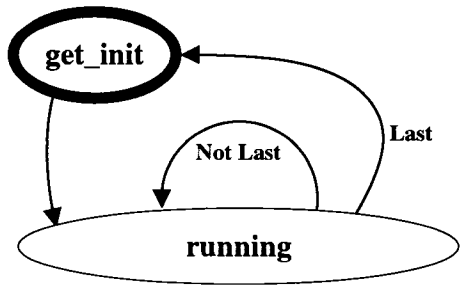


Figure A.8: FSM of a COUNTER_LOCAL node.

A.2.10 COUNTER_LOCAL_CONST

The COUNTER_LOCAL_CONST node is a combination of the optimizations present in the COUNTER_CONST and COUNTER_LOCAL nodes. It is the most optimized of the COUNTER variants. Its state machine would be reduced to a single state, and so is omitted all together.

The inputs, outputs, and pragmas of the COUNTER_LOCAL_CONST node are identical to the COUNTER_CONST node.

A.2.11 DONE

The DONE node is used to identify the end of a loop sequence, and does not operate on any data directly. It has a single input (Last_In) and a single output (Last_Out), which work as described in Section A.2. In its initial state, it continuously reads the Last_In port until it becomes 1, and then drops a single 1 on the Last_Out port, and then returns to normal operation again. This node is used to identify the end of a loop, and is used to trigger the next loop, or outer loop, or host, as is necessary. It can be thought of as the inverse of a TOK_GEN node which will be discussed later.

A.2.12 FIFO_PACK

The FIFO_PACK node is used to collect n values serially and pack them into a single (larger) word. Its most common use is as part of a collector node. For example, if a loop body is generating a sequence of 8-bit pixels, they need to be packed in groups of 4 into a single 32-bit word before they can be written to the memory (by a WRITE_WORD_MEMORY

node). The 0th value received is stored in the least significant bits of the resulting word, and the $n-1$th value is stored in the most significant bits. This node has 4 available implementations, called "clocked1," "clocked2," "buffered1," and "buffered2."

In the "clocked1" implementation, the register for the final word is broken into cells, and as each value arrives, it is directly stored in the appropriate cell. This has the advantage that when the `Last_In` flag arrives, the partially constructed word is guaranteed to be in the correct state, and can be dropped (if it is not empty) as is, followed by a garbage value with the last-flag. The disadvantage of this implementation is that in the hardware implementation, the arriving value needs to have a path to every cell, which may require significant resources.

In the clocked2 implementation, the incoming value is always stored in the most significant cell, and the rest of the cells shift 1 cell toward the least significant cell. This has the advantage that the incoming value only has 1 target; however, it may require a few extra cycles to finish shifting if a `Last_In` flag arrives in the middle of constructing a word.

In both the "clocked1" and "clocked2" implementations, no new values may be read after the $n-1$th value arrives, until the fully constructed word is accepted by the consuming section below. This results in a minimum of $n+1$ clock cycles to pack each word (n to read the values, and 1 to output the packed word). To fix this inefficiency, the "buffered1" and "buffered2" styles use a single SRL16E FIFO to buffer the inputs (as was shown in Figure A.1 b), but otherwise work exactly like the clocked1 and clocked2 styles, respectively. The FIFO allows the body of the node to read the first input value in the same clock cycle that it is producing the output word without creating a combinational path between the handshaking signals of 2 sections. Logically, the FIFO could have just as easily been placed on the outputs instead of the inputs (as in Figure A.1 a), however, in this node the inputs are always smaller values than the outputs, so we save a factor of n by buffering inputs rather than outputs.

The clocked1 and clocked2 styles are not used any longer, because the buffered styles are given priority in the `node_weights` file over the clocked styles. They are only retained because the buffered styles require slightly more chip area due to the FIFOs. However, in practice, the minuscule increase in area is acceptable when faced with a significant performance degradation as the only alternative (25% in loops writing 8-bit values, and 50% in loops writing 16-bit values).

All 4 styles of this node have 2 inputs and 2 outputs.

Inputs:

0. **Data_In**: The values to be packed into each word.

1. **Last_In**: Last-flag, described in Section A.2.

Outputs:

0. **Data_Out**: The output word containing n values from `Data_In`.

1. **Last_Out**: Last-flag, described in Section A.2.

`FIFO_PACK` nodes have a single mandatory pragma:

- **DEPTH**: The value of n in the above discussion.

A.2.13 FIFO_PACK_FULL

The `FIFO_PACK_FULL` node is identical in function to the `FIFO_PACK` node (it even has the same 4 styles available, with the same behavior in each). The inputs, outputs, and pragmas are the same as well. The only difference is that it is guaranteed by the compiler that the `Last_In` flag will only arrive after an integer multiple of n values. This ensures that there is no final shifting required in the "clocked2" and "buffered2" styles, and that a partially filled word will never need to be produced. This allows the VHDL to be optimized to use less hardware resources.

A.2.14 FIFO_UNPACK

The FIFO_UNPACK node is the inverse of the FIFO_PACK_FULL node; it receives as input a large word which contains n sub-values and produces them serially on its output. It only has 2 styles: "clocked" and "buffered." The clocked style receives a packed word, and stores it in a shift register (similar to the shift register in the clocked2 style of the FIFO_PACK node). Like the clocked styles of the FIFO_PACK, $n+1$ clock cycles are required to pack each word. The buffered style introduces a FIFO on the outputs (Figure A.1 b), to reduce the time to n cycles. The outputs are buffered instead of the inputs, because the outputs are narrower in this node. As with the FIFO_PACK node, the clocked style is deprecated in preference of the more efficient buffered style.

Both styles of this node have 2 inputs and 2 outputs.

Inputs:

0. **Data_In**: The value to be unpacked.

1. **Last_In**: Last-flag, described in Section A.2.

Outputs:

0. **Data_Out**: The n values which were packed in the Data_In word.

1. **Last_Out**: Last-flag, described in Section A.2.

FIFO_UNPACK nodes have a single mandatory pragma:

- **DEPTH**: The value of n in the above discussion.

A.2.15 INSERT_LAST

The INSERT_LAST node is used to create a sequence of length 1, out of every element in an input sequence. That is, for every input value, it replicates it on the output, with Last_Out false, and then again, with Last_Out true. This node is only used when a loop is writing a single value to memory after the loop terminates, such as the result of a reduction. Because this node always lies outside of the innermost loop, there is no need to make a "buffered" style, so only a "clocked" style exists.

The INSERT_LAST node has only a single input, and 2 outputs.

Input:

0. **Data_In**: The values in the input sequence.

Outputs:

0. **Data_Out**: The values copied from Data_In.

1. **Last_Out**: Last-flag, described in Section A.2.

INSERT_LAST nodes have no pragmas associated with them.

A.2.16 MASK

The MASK node is used to selectively remove elements from a sequence. It simply copies data from its inputs to its outputs, using a boolean input to suppress certain values. Last-flags are always copied to the output regardless; other values are ignored whenever the boolean is true, and copied when false. This node has a "clocked" and a "buffered" style. In the clocked style, the node contains a register to hold the value (and the last-flag) as well as a presence bit to indicate whether the register contains a valid value. When the presence bit indicates the register is empty, it accepts values from the inputs, and sets the presence bit based on the last-flag and the boolean value. When the presence bit indicates the register is full, it outputs its value, and clears the presence bit. This behavior requires 2 clock cycles for a value which is copied, and 1 clock cycle for a value which is ignored. By contrast, the buffered style is much simpler, and employs a FIFO in the same way as other buffered nodes. It is difficult to say whether the FIFO in the buffered style is contained in the top or the bottom of the node, as the logic of the node is entirely replaced with the FIFO. A small boolean expression is used to identify when to enqueue. This allows increased performance which permits a new value to be accepted in every clock cycle.

The MASK node has 3 inputs, and 2 outputs.

Inputs:

0. **Data_In**: The values in the input sequence.

1. **Mask_In**: The boolean flag indicating whether the Data_In value should be removed. (1 indicates the value should be masked, or ignored; 0 causes the value to be copied as is.)

2. **Last_In**: Last-flag, described in Section A.2.

Outputs:

0. **Data_Out**: The values copied from Data_In.

1. **Last_Out**: Last-flag, described in Section A.2.

MASK nodes have no pragmas associated with them.

A.2.17 MERGE

The MERGE node is the counterpart of a SPLIT node. They are used together to implement conditional execution in areas where side effecting is an issue, as discussed in Section 4.1. In a MERGE which merges n conditional branches, there are $n+1$ input clusters, and only 1 output cluster. MERGE nodes have both a "clocked" and a "buffered" style.

The clocked style uses a finite state machine with $n+2$ states. State n is the initial state. In this state, the node stalls and waits for the control signal (Where_In) to indicate which of the branches will be executing next. Based on the Where_In signal, the node transitions to one of the first n states $(0 \cdots n-1)$. Each of these corresponds to a branch of the conditional. When in one of these states, the node waits for the cluster containing the indicated branch to fire, and accepts data from it, and then moves to state $n+1$. In state $n+1$, the node produces the value it received from the executed branch. The clocked style requires 3 clock cycles each time the conditional executes, and does not allow the conditional to be pipelined. This is because the input cluster which provides

the control signal cannot fire in any state other than state n. This prevents the logic which is generating the control signal from executing until the conditional has completed.

The buffered style addresses many of the inefficiencies of the clocked style. It uses an input FIFO (Figure A.1 a) to buffer the control signal, as well as an output FIFO (Figure A.1 b) to buffer the output. These FIFOs allow the node to read the control signal from the input buffer, and copy values from the incoming branches to the output buffer simultaneously, removing the need for a state machine. The buffering of the control signal allows the conditional to be pipelined. The logic generating the control signal may continue operating as the conditional executes. Due to the pipelining effect, the buffered style of the `MERGE` node is capable of producing a new value every clock cycle in steady state. The only circumstance where the clocked style outperforms the buffered style is when no pipelining is possible, and the last-flag arrives. In the clocked style, the `MERGE` node is able to produce output 1 clock cycle after the last-flag arrives, where the buffered node requires 2 clock cycles after the arrival of the flag (1 for the input buffer, and 1 for the output buffer).

The `MERGE` node has $n + 2$ inputs and 1 output.

Inputs:

0. **`Where_In`**: The control signal which indicates which conditional branch is being executed.

1. **`Last_In`**: Last-flag, described in Section A.2. Unlike other clocked nodes, `MERGE` drops a 1 (padded with 0s on the high order bits) on its `Data_Out` line when a last token is received. Because the value which is produced with a last-flag is ignored as garbage, this should not be necessary, however the SA-C* compiler expects (and depends upon) this subtle behavior. Because reworking the SA-C* compiler was not an option, the `MERGE` node was modified to conform with it.

2. **`Data_In`**: The data from all of the conditionals concatenated.

Output:

0. **Data_Out**: The values copied from the conditional branches as they execute.

MERGE nodes have no pragmas associated with them.

A.2.18 READ_WORD_BRAM

The READ_WORD_BRAM node is used to read data from a BlockRAM. It accepts a stream of addresses, and produces the stream of values in those addresses. For example, the BlockRAM can be represented as an array A, if it accepts the sequence $4, 6, 2$, then it produces the sequence $A[4], A[6], A[2]$. The pragma on the top-level wrapper node which created the BlockRAM also defines the width, and depth of it, and the bit-widths used in the node must match. For example, if the BlockRAM being accessed contains 16-bit values, then the node must take this into account. Only a "clocked" style is available for READ_WORD_BRAM nodes, but this is somewhat misleading, because it does utilize SRL16 FIFOs. However, they are used inside the node to provide basic functionality, not at the ports to enhance efficiency.

Because the BlockRAMs have a single clock cycle of latency, a buffer is required to accept the value arriving back from the memory in the event that the output section is not ready to fire when the data is arriving. In order to handle the last-flag correctly, 2 FIFOs are used in parallel, 1 for holding the last-flag, and 1 for holding the actual data arriving back from the memory. Every time an address to read arrives, the last-flag is stored into the last-FIFO, and the read request is made to the BlockRAM to retrieve the appropriate cell. In the following clock cycle, the value arrives from the memory and is enqueued into the data-FIFO. Together, the 2 FIFOs imitate a single FIFO attached to the output port. Because the last-FIFO always contains at least as many elements as the data-FIFO, if the last-FIFO is not full, neither is the data-FIFO. Likewise, if the data-FIFO has at least 1 element available to be dequeued, so does the last-FIFO. When a last-flag arrives, the BlockRAM is not instructed to perform a read access, but

the (garbage) value it returns in the next clock cycle is enqueued into the data-FIFO anyway.

The READ_WORD_BRAM node has 2 inputs, and 2 outputs.

Inputs:

0. **Address_In**: The address to read from the BlockRAM. The bit-width of this port must correspond to the number of bits required to address the full depth of the BlockRAM.

1. **Last_In**: Last-flag, described in Section A.2.

Outputs:

0. **Data_Out**: The values which were read from the BlockRAM. The bit-width of this port must match the bit-width of the elements stored in the BlockRAM.

1. **Last_Out**: Last-flag, described in Section A.2.

READ_WORD_BRAM nodes have no pragmas associated with them, since all of the required information about the BlockRAM geometry can be gleaned from the port widths.

A.2.19 READ_WORD_MEMORY

The READ_WORD_MEMORY node is nearly identical to the READ_WORD_BRAM except that it is used to interface with external memories (such as the ZBT memories on the Alpha Data board) instead of BlockRAMs. Like the READ_WORD_BRAM node, READ_WORD_MEMORY nodes have only a "clocked" style, but use SRL16-based FIFOs in the same manner as READ_WORD_BRAM nodes. The delay between a request made to the memory and the response is variable depending on the memory to which the node is attached. For this reason, a shift register is used to delay the signal to enqueue into the data-FIFO instead of a register. The node is capable of handling memories with a latency from 1 to 16 clock cycles.

The READ_WORD_MEMORY node has the same 2 inputs and the same 2 outputs as the READ_WORD_BRAM node. It has a single mandatory pragma MEMORY associated with it.

The pragma is used to identify which memory it is connected to in order to prepare the memory-related variables (see the memory_ variables in Table 3.6). These variables are pre-inserted into the symbol table of the node before the preamble is parsed to allow the VHDL to be parameterized.

A.2.20 REPLICATOR_FLAG_EXTRA

The REPLICATOR_FLAG_EXTRA node is used to provide a value from outside a loop into its body. The name of the node is somewhat misleading since it carries its history from AHA, but the behavior is perfectly consistent with the AHAHA model.[5] Initially, in the AHA environment there were 2 nodes called REPLICATOR and REPLICATOR_FLAG. The REPLICATOR was used to repeat a value a given number of times, as in a for-loop with known bounds. It would accept, for example, the pair [5, 3], and produce a sequence of three 5s. The REPLICATOR_FLAG was used to repeat a given value based on a boolean flag. It was used when the number of times to replicate was not known beforehand, as in a while-loop. For example, in order to recreate the same sequence of three 5s, it would accept the 5, followed by 3 0s and a 1. The boolean flag in this node corresponds loosely to the last-flag in the AHAHA model except that no garbage value is associated with a value of 1. Finally, the REPLICATOR_FLAG_EXTRA node was created which would repeat the value 1 extra time at the end of the sequence of flags before receiving a new value to repeat. This behavior is completely consistent with the last-flag in the AHAHA model. The REPLICATOR and REPLICATOR_FLAG nodes were not needed in the AHAHA framework because they clashed with the AHAHA paradigm, and the formerly "special" behavior associated with the REPLICATOR_FLAG_EXTRA is now normal. The node has 2 input clusters and a single output cluster. The first input cluster provides the value to replicate, and the second contains the last-flag.

---

[5] Arguably, this node should have been renamed when it was implemented in the AHAHA framework. The decision to leave it with the old name was made to minimize the changes required in the SA-C* compiler.

The REPLICATOR_FLAG_EXTRA node has 2 inputs and 1 output.

Inputs:

0. **Data_In**: The value to be replicated.

1. **Last_In**: Last-flag, described in Section A.2.

Output:

0. **Data_Out**: The repeated value.

REPLICATOR_FLAG_EXTRA nodes have no pragmas associated with them.

A.2.21 SHIFT_REGISTER

The SHIFT_REGISTER node is the primary method of implementing sliding window operations. It accepts a sequence of values and produces contiguous subsequences of its history. The length, and stride of the window is variable, and may be selected based on the context of the node. The SHIFT_REGISTER has both a "clocked" and a "buffered" style.

There are several special cases for this node which are outside its intended use. For example, with a window depth of 1, and a stride of n, it can be used to produce every nth value of a sequence. This behavior is better accomplished with a MASK node. When the depth and stride are the same, and greater than 1, this node behaves similarly to a FIFO_PACK, which would be a better choice. When the depth and stride are both 1, this node has no effect whatsoever, and should be removed, or replaced with a BUFFERX, to preserve the section partitions. When the greatest common divisor (GCD) of the depth and stride is greater than 1, then the node can be simplified (or *factored*). This involves dividing both the window and stride by the GCD, and multiplying the input bit-width by the GCD. This increases the maximum throughput of the node. This may require some changes upstream from the input of the node to create larger data elements.

The "clocked" style of the SHIFT_REGISTER uses a simple state machine with only 2 states: **get_values** and **put_value**, a shift register for holding the values in the window,

and a counter to know how many elements of data have been accumulated for the next window. Initially, the state machine is in the **get_values** state, and the counter is 0. Each time an input value is received, it is shifted into the window (into the low-order bits), the oldest value is shifted out of the window and lost, and the counter is incremented. When the counter reaches the depth of the window, it transitions into the **put_value** state. In this state, it produces the entire window, concatenated together, decrements the counter by the window stride, and transitions back to the snget_values state. Note that it is possible that the stride will be greater than the window size, and in this case the counter must be able to hold negative values.[6] Partial windows are not valid; if a last-flag arrives in the middle of filling a window, that (garbage) window is immediately produced with an outgoing last-flag, and it will not be used in the computation.

The "buffered" style of the SHIFT_REGISTER is identical in function to the "clocked" style, except that it uses an input FIFO (Figure A.1 a) to buffer the incoming values. This buffer allows the node to accept an incoming value in the same clock cycle that it is producing output. This behavior removes a dead clock cycle so that the node can accept a new value in every clock cycle in the steady state. It is tempting to move the buffer to the output port instead of the input port. The output is at least as large as the input, and most often is substantially larger. Buffering the output would allow the the SHIFT_REGISTER to pre-compute some windows when the graph begins to stall. However, in practice, this condition does not occur often enough to warrant the larger buffer. In the best of circumstances, this node requires a number of clock cycles equal to the stride of the window in order to produce a new window, which is why factoring the node as described above may be a useful practice.

---

[6]Because the typical behavior of the XST VHDL synthesis tool is to initialize integer-typed values with their lowest possible value at configuration time (which would be negative in this case), it is vital that the SHIFT_REGISTER is reset using the AHAHAReset signal before its use.

The SHIFT_REGISTER node has 2 inputs, and 2 outputs.

Inputs:

0. **Data_In**: The values in the input sequence.

1. **Last_In**: Last-flag, described in Section A.2.

Ouptuts:

0. **Data_Out**: The constructed window, concatenated into 1 large value, which may then be accessed using an UNPACK node.

1. **Last_Out**: Last-flag, described in Section A.2.

SHIFT_REGISTER nodes have 2 mandatory pragmas:

- **DEPTH**: The number of elements which should be accumulated in the window before it is produced.
- **STEP**: The stride between window positions.

A.2.22 SPLIT

The SPLIT node is the counterpart of a MERGE node. SPLIT and MERGE nodes are used together to implement conditional execution in areas where side effecting is an issue, as discussed in Section 4.1. In a SPLIT which merges n conditional branches, there are n output clusters, and only 1 input cluster. SPLIT nodes have both a "clocked" and a "buffered" style.

The clocked style uses a finite state machine with $n + 1$ states. State n is the initial state. In this state, the node stalls and waits for the data (Data_In) and control signal (Where_In) to arrive. Based on the Where_In signal, the node transitions to one of the first n states ($0 \cdots n - 1$). Each of these corresponds to a branch of the conditional. When in one of these states, the node sends the data into the associated branch, and then moves back to state n. The clocked style requires 2 clock cycles each time the conditional executes.

The buffered style addresses the inefficiencies of the clocked style. It uses an input FIFO (Figure A.1 a) to buffer the data and control signal. This buffer allows the node to read its inputs and write values to the output simultaneously, removing the need for the state machine. The buffered style of the SPLIT node is capable of accepting a new value every clock cycle in steady state.

The SPLIT node has 3 inputs, and n outputs.

Inputs:

0. **Where_In**: The control signal which indicates which conditional branch is being executed.

1. **Last_In**: Last-flag, described in Section A.2. Unlike other clocked nodes, SPLIT ignores the last condition completely, and sends no information into any of the branches when it occurs. The corresponding MERGE node (which receives the same last-flag) compensates for this on the other end of the conditional.

2. **Data_In**: The data to be passed into the active conditional.

Outputs:

0. **Data_Out**: The output to all n of the branches concatenated together.

SPLIT nodes have no pragmas associated with them.

A.2.23 STREAM_CLOSE

The STREAM_CLOSE node is used to close a stream. This indicates that no further values will be enqueued into the stream. After the first time a stream has been closed, no further enqueue operations are permitted, and attempting to do so will cause a runtime error in simulation and a malfunction in the hardware execution. This node has only a "clocked" style. Buffering inputs or outputs on this node is not required since it is not intended to execute in a pipelined fashion, and usually appears at the end of a process or loop.

This node has 2 inputs, and 1 output.

Inputs:

0. **Stream ID**: The stream ID, discussed in Section A.2.

1. **Flag_In**: Stream-flag, described in Section A.2.

Output:

0. **Flag_Out**: Stream-flag, described in Section A.2.

This node has no pragmas associated with it.

A.2.24 STREAM_FINALIZE

The STREAM_FINALIZE node identical in nearly every way to the STREAM_CLOSE node except that its Flag_Out port is always 1. That is, when the node is allowed to execute (incoming flag is 0), it produces an outgoing flag of 1 so that subsequent operations on the stream will be disabled.

A.2.25 STREAM_GET

The STREAM_GET node retrieves a value from a stream, which must be open, otherwise the node will malfunction. If the stream is empty, the node will stall until a value arrives. It has a "clocked" style, as well as a "buffered" style. The need for a buffered style when stream operations are serialized may not seem clear. However, in situations where only a single stream operation is performed (as in the program in Figure 5.9), the graph can be optimized (as in Figure 5.15). In these situations, the stream operations may actually be pipelined, and the buffered style will outperform the clocked one.

This node has 2 inputs and 2 outputs.

Inputs:

0. **Stream ID**: The stream ID, discussed in Section A.2.

1. **Flag_In**: Stream-flag, described in Section A.2.

Outputs:

0. **Data_Out**: The value which was dequeued from the stream.

1. **Flag_Out**: Stream-flag, described in Section A.2.

This node has no pragmas associated with it.

A.2.26 STREAM_IS_OPEN

The STREAM_IS_OPEN node returns a boolean indicating whether the stream has reached the end (0) or if it has more values (1). The node must stall if the stream is empty and has not been closed, because the stream may become closed (in which case it should produce a 0), or it may have a value enqueued into it (in which case it should produce a 1). Typically, this node is used in the test of a while-loop which operates on stream values. This node has only a "clocked" style.

This node has 2 inputs and 2 outputs.

Inputs:

0. **Stream ID**: The stream ID, discussed in Section A.2.

1. **Flag_In**: Stream-flag, described in Section A.2.

Outputs:

0. **Data_Out**: A boolean value indicating the status of the stream.

1. **Flag_Out**: Stream-flag, described in Section A.2.

This node has no pragmas associated with it.

A.2.27 STREAM_PEEK

The STREAM_PEEK node retrieves the value at the head of the stream, which must be open, otherwise the node will malfunction. The value is not dequeued; the same value will be available to subsequent peek or get operations on the stream. If the stream is empty, the node will stall until a value arrives. If 2 STREAM_PEEK nodes are to be executed in serial,

they may be merged since both nodes will return the same value, without side effecting the stream. Likewise, if a STREAM_PEEK is followed by an unconditional STREAM_GET, they may be merged into a single STREAM_GET node. This node has "clocked" and "buffered" styles.

This node has 2 inputs and 2 outputs.

Inputs:

0. **Stream ID**: The stream ID, discussed in Section A.2.

1. **Flag_In**: Stream-flag, described in Section A.2.

Outputs:

0. **Data_Out**: The value at the head of the stream.

1. **Flag_Out**: Stream-flag, described in Section A.2.

This node has no pragmas associated with it.

A.2.28 STREAM_PUT

The STREAM_PUT node conditionally enqueues a value into a stream, which must be open, otherwise the node will malfunction. The node has a boolean input which indicates whether the value should be enqueued or not. If the stream is full, the node will stall until it has space. This node has both a "clocked" and a "buffered" style. See the discussion for the STREAM_GET node above for a discussion of why the "buffered" style is needed.

This node has 4 inputs, and 2 outputs.

Inputs:

0. **Data_In**: The data to be enqueued into the stream.

1. **Mask_In**: A boolean flag indicating whether the enqueue should be executed or skipped.

2. **Stream ID**: The stream ID, discussed in Section A.2.

3. **Flag_In**: Stream-flag, described in Section A.2.

Outputs:

0. **Data_Out**: The value which was dequeued from the stream.

1. **Flag_Out**: Stream-flag, described in Section A.2.

This node has no pragmas associated with it.

A.2.29 TOK_GEN

The TOK_GEN node is responsible for driving for-loops which have runtime static bounds. It is the source of the last-flags which are used in the inner loop. It receives a value n, and produces n 0s followed by a single 1. It also accepts a last-flag from the outer loop, which when it is 1, causes the node to override the incoming n value to 0. In the event that there is no outer loop, this value can be safely assigned a constant 0. The node only has a "clocked" style.

This node has 2 inputs and 1 output.

Inputs:

0. **Data_In**: The number of 0s to produce before the 1.

1. **Last_In**: Last-flag from the outer loop, described in Section A.2.

Output:

0. **Last_Out**: Last-flag, described in Section A.2.

This node has no pragmas associated with it.

A.2.30 WRITE_WORD_BRAM

The WRITE_WORD_BRAM node is used to write a given value into a given cell of a Block-RAM. It accepts a stream of address, value pairs, and writes the value into the indicated address. Only a "clocked" style is available for READ_WORD_BRAM nodes, but this is somewhat misleading, because it does utilize an SRL16 FIFO. However, it is used to provide

functionality, rather than being present expressly to improve efficiency (although it does also provide efficiency). The node is effectively just a small FIFO for the last flag, which has a side effect of writing a value into the BlockRAM every time a 0 is enqueued. The purpose of the FIFO is to delay the last-flag by at least 1 clock cycle so that the value will have actually been written into the BlockRAM when the last-flag is produced.

The `WRITE_WORD_BRAM` node has 3 inputs and 1 output.

Inputs:

0. **`Address_In`**: The address to write to in the BlockRAM. The bit-width of this port must correspond to the number of bits required to address the full depth of the BlockRAM.

1. **`Data_In`**: The value which will be written to the BlockRAM. The bit-width of this port must match the bit-width of the elements stored in the BlockRAM.

2. **`Last_In`**: Last-flag, described in Section A.2.

Output:

0. **`Last_Out`**: Last-flag, described in Section A.2.

`WRITE_WORD_BRAM` nodes have no pragmas associated with them since all of the required information about the BlockRAM geometry can be gleaned from the port widths.

A.2.31 WRITE_WORD_MEMORY

The `WRITE_WORD_MEMORY` node is similar to the `WRITE_WORD_BRAM` except that it is used to interface with external memories (such as the ZBT memories on the Alpha Data board) instead of BlockRAMs.

It is modeled after the `READ_WORD_MEMORY` node, except that the data-FIFO is used only for synchronization purposes, and stores degenerate 0-bit values. The node is capable of handling memories with a latency from 2 to 16 clock cycles.

The `WRITE_WORD_MEMORY` node has the same 3 inputs and the same output as the `WRITE_WORD_BRAM` node. It has a single mandatory pragma `MEMORY` associated with it.

The pragma is used to identify to which memory it is connected in order to prepare the memory-related variables (see the `memory_` variables in Table 3.6). These variables are pre-inserted into the symbol table of the node before the preamble is parsed to allow the VHDL to be parameterized.

A.3 Unclocked Nodes

Unclocked nodes are much simpler than clocked nodes. They do not participate in handshaking, utilize last signals, or store any internal state. They simply contain combinational logic. However, they are the computational workhorses of the graph, and perform nearly all of the arithmetic operations. None of the unclocked nodes have any pragmas.

A.3.1 BIT_AND, BIT_EOR, BIT_OR

The `BIT_AND`, `BIT_EOR`, `BIT_OR` nodes are simple binary operators which perform simple boolean logic. These nodes may be used to perform logic on values longer than 1-bit, in which case each bit is operated on independently. These nodes are similar to the `&`, `|`, and `^` operators in C respectively. All the inputs and outputs must have the same bit-width. These nodes have 2 styles: "unclocked1" and "unclocked2." They differ only in that the operands are swapped in the second style. This is possible because the operators are commutative. It is unlikely that there will be a difference between the hardware implementations of the 2 styles, but they are included only for completeness.

These nodes have 2 inputs and 1 output.

Inputs:

0. **`A`**: The first operand.

1. **`B`**: The second operand.

Output:

0. **`R`**: The result value.

A.3.2 BIT_COMPL, NEG

The BIT_COMPL node performs bitwise negation, and the NEG node performs signed arithmetic negation. When using the BIT_COMPL node, the input and output must be the same size, but the NEG node contains an implicit CHANGE_WIDTH_SE on its output. BIT_COMPL has only an "unclocked" style, since there is only 1 reasonable implementation of it. The NEG node, however has 2 styles: "unclocked1" and "unclocked2." The "unclocked1" style simply allows the VHDL to compute $-X$, where the "unclocked2" style uses the definition of two's-compliment numbers; it negates all of the bits in the same way as the BIT_COMPL node, and then adds 1 to the result. Again, it is unlikely that there will be a difference between the hardware implementations of the 2 styles, but they are included only for completeness.

These nodes have 1 input, and 1 output.

Input:

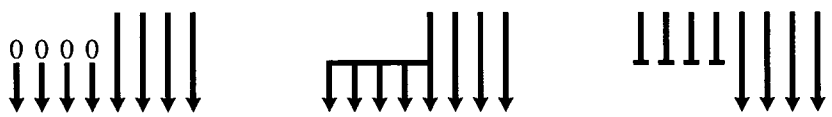
0. **A**: The operand.

Output:

0. **R**: The result.

A.3.3 CHANGE_WIDTH, CHANGE_WIDTH_SE

These nodes are used to change the bit-width of unsigned and signed numbers respectively (the SE stands for Sign-Extend). When the result is larger than the operand, they behave as in Figures A.9 a and A.9 b respectively. When the result is smaller than the



a: Extend. b: Sign-Extend. c: Truncate.

Figure A.9: Implementations of CHANGE_WIDTH and CHANGE_WIDTH_SE nodes.

operand, they both truncate the result as in Figure A.9 c.

Because both of these nodes contain only routing, and no logic, these nodes use the `skip_compile` variable to prevent wasted compilation for the purpose of estimation.

These nodes have 2 inputs and 1 output.

Inputs:

0. **A**: The operand.

1. **New Size**: The size that to which the operand will be resized. This value must be a constant, and must match the bit width of the output port **R**.

Output:

0. **R**: The resized result.

A.3.4 IADD, UADD, ISUB, USUB, IMUL, UMUL

These nodes are used for adding signed values, adding unsigned values, subtracting signed values, subtracting unsigned values, multiplying signed values, and multiplying unsigned values, respectively. Because addition and multiplication are commutative, the `IADD`, `UADD`, `IMUL` and `UMUL` nodes have 2 styles "unclocked1" and "unclocked2"; `ISUB` and `USUB` only have an "unclocked" style. All of these nodes contain implicit `CHANGE_WIDTH`s or `CHANGE_WIDTH_SE`s on their output ports depending on whether they are performing signed or unsigned arithmetic. For this reason, there is no constraint on the bit-widths of the result value. Likewise, the operands may also be of differing sizes. Note that division is not supported because it is not implemented by the underlying VHDL synthesis tools.

These nodes have 2 inputs and 1 output.

Inputs:

0. **A**: The first operand.

1. **B**: The second operand.

Output:

0. **R**: The result value.

A.3.5 IEQ, IGE, IGT, ILE, ILT, INEQ, UEQ, UGE, UGT, ULE, ULT, UNEQ

All of these nodes perform comparisons on either signed or unsigned values. All of the variants which begin with "I" treat their operands as signed values; the ones which begin with "U", treat them as unsigned. The nodes, their standard mathematical symbol, and their description are shown in Table A.1.

Node	Symbol	Description
IEQ	$=$	Signed equal to
IGE	$\geq$	Signed greater than or equal to
IGT	$>$	Signed greater than
ILE	$\leq$	Signed less than or equal to
ILT	$<$	Signed less than
INEQ	$\neq$	Signed not equal to
UEQ	$=$	Unsigned equal to
UGE	$\geq$	Unsigned greater than or equal to
UGT	$>$	Unsigned greater than
ULE	$\leq$	Unsigned less than or equal to
ULT	$<$	Unsigned less than
UNEQ	$\neq$	Unsigned not equal to

Table A.1: Description of comparison nodes.

All of these nodes have 4 styles ("unclocked1," "unclocked2," "unclocked3," "unclocked4"), which provide 4 ways to implement them. The "unclocked1" style is the most straightforward, treating the operands in order, and using the VHDL comparison operator that is implied by their name. The "unclocked2" style reverses the order of the operands, and adjusting the operator accordingly. For example, an IGT node is logically equivalent to an ILT node with the operands reversed. The "unclocked3" style inverts the sense of the operand to convert it into its logical opposite. For example, an IGT node (producing a 1 when **A** is greater than **B**, and 0 otherwise) is behaviorally equivalent to producing a 0 when **A** is less than or equal **B**, and 1 otherwise. The final style, "unclocked4", combines these 2 ideas, swapping the order of the operands, and inverting the sense of the operator. The rationale for using 4 styles is that these nodes are often

used with constant inputs, which might cause different hardware to be generated. The input bit-widths need not be the same. The result is always 1-bit.

All of these nodes have 2 inputs and 1 output.

Inputs:

0. **A**: The first operand.

1. **B**: The second operand.

Output:

0. **R**: The boolean result value.

A.3.6 LEFT_SHIFT_CONST, RIGHT_SHIFT_CONST

These nodes are used to perform logical left or right shifts. The first operand is the value to be shifted, and the second argument (which may not be 0) is an unsigned compile time constant which indicates the distance of the shift. In the case of a left shift, the result value grows by the size of the left shift. For example, an 8-bit value left shifted by 4 bits will produce a 12-bit result, with the low 4 bits all 0. When right shifting, the result is shortened by the size of the shift; an 8-bit value right shifted by 4 bits will produce a 4-bit result, with the lowest 4 bits of the operand dropped. In order to obtain a shift which is similar in function to the C shift operators, the shift node must be followed by a `CHANGE_WIDTH` or `CHANGE_WIDTH_SE` node to return the operand to the correct size. In a right shift, the size of the shift must be less than the bit-width of the operand. Because both of the shift nodes contain only routing, and no logic, these nodes use the `skip_compile` variable to prevent extra compilation.

As the names of these nodes may imply, there was once an intention to build the non-constant versions `LEFT_SHIFT` and `RIGHT_SHIFT`, which would perform runtime shifts of size n using $log(n)$ shifts by powers of 2 with MUXes. Such a node would produce results that are the same size as its operands. However, in practice, such nodes were never needed. If at some point they became required, their implementation would be simple, as would roll-left and roll-right operators.

These nodes have 2 inputs and 1 output.

Inputs:

0. **A**: The operand to be shifted.

1. **B**: The constant input which indicates the size of the shift.

Output:

0. **R**: The result value.

A.3.7 PACK

The PACK node is used to concatenate several values into one, similar to the & operator in VHDL. This is similar to filling in a C struct. Because this node is implemented with out the need for any logic cells, it uses the skip_compile variable to prevent compilation.

The pack node has any number (greater than 1) of inputs, which are concatenated into a single output, whose bit-width must be the sum of the input bit-widths. Input 0 becomes the most significant bits of the result, and the last input becomes the least significant bits.

A.3.8 ROMREF

The ROMREF node is used to implement a lookup table. It is not implemented in a VHDL entity like most other nodes. Instead, the .vhd file which implements it simply gathers information about the node and stores it in a series of variables. Instead of an entity/architecture pair, the .vhd file contains the command:

```
--> SPECIAL_CASE_IMPLEMENTATION "ROMREF";
```

which invokes the special case code contained in ahaha_special_romref.c. This code generates the VHDL implementation using the variables declared in the preamble. The variables which are required by the special case code are shown in Table A.2. The contents of the variables is type-checked by the special case code, and verified to ensure that they are not contradictory.

Variable	Type	Description
SPECIAL_ROMREF_VALUES_IN	Int List	The input port numbers used to construct the ROM table. Defined to be "[0 ~ $num_inputs - 2]."
SPECIAL_ROMREF_INDEX_IN	Int List	The input port number which is used as the index. The list must be 1 element long. Defined to be "[$num_inputs - 1]."
SPECIAL_ROMREF_DATA_OUT	Int List	The output port number which receives the result. The list must be 1 element long. Defined to be "[0]."
SPECIAL_ROMREF_WIDTH_IDX	Int	The bit-width of the index value. Defined to be "sizeof(in, $SPECIAL_ROMREF_INDEX_IN)."
SPECIAL_ROMREF_WIDTH	Int	The bit-width of result value (and of constants in the table). Defined to be "sizeof(out, $SPECIAL_ROMREF_DATA_OUT)."
SPECIAL_ROMREF_DEPTH	Int	The number of elements in the ROM table. Defined to be "$num_inputs - 1."

Table A.2: Variables used in the ROMREF special case implementation code.

A constant array containing the result values of the lookup table is dynamically generated before the begin statement of the top-level architecture. The lookup table is indexed using an integer in the body of the VHDL. The synthesis tools are able to recognize this as a ROM table and convert it into logic. The complexity of this node is directly dependent upon the values used to fill in the table. For example, a ROMREF which implements a bit-reversal of an 8-bit input would be similar to the following VHDL:

```
ROMREF_romty is array (natural range <>) of std_logic_vector(7 downto 0);
constant ROMREF_rom :  ROMREF_romty(255 downto 0) := (
       "11111111",
       "01111111",
       "10111111",
       "00111111",
          ...
       "11000000",
       "01000000",
       "10000000",
       "00000000");
```

The "..." in the above code is the reason that this node must be dynamically generated using a special case. The length and contents of the table must be tailored for each ROMREF node that is created.

This node has a number of inputs which must be 1 more than a power of 2. All inputs except the last must be constants used to construct the table. The final input is treated as an unsigned value and is used to index the table. The output bit-width must be the same as the bit-widths of the constant inputs, which must all be identical.

A.3.9 SELECTOR

The `SELECTOR` node is a special case node used to implement an n-input priority MUX (or selector). It is not implemented in a VHDL entity like most other nodes. Instead, the **.vhd** file which implements it simply gathers information about the node and stores it in a series of variables. Instead of an entity/architecture pair, the **.vhd** file contains the command:

```
--> SPECIAL_CASE_IMPLEMENTATION "SELECTOR";
```

which invokes the special case code contained in **ahaha_special_selector.c**. This code generates the VHDL implementation using the variables declared in the preamble. The variables which are required by the special case code are shown in Table A.3. The variables are type-checked and verified to ensure that they are not contradictory.

The `SELECTOR` node which is used to implement an n-input priority MUX, has $(2n)+2$ inputs and a single output. The first input (input 0) contains the selector control signal. The next $2n$ inputs are in the form n index/value pairs. The value describes what the value of the MUX should be when the index matches the control signal. A "match" is defined as being logically equivalent to an unsigned equal-to comparison; therefore, the bit-width of the low nonzero bits of the indexes must be smaller than or equal to the bit width of the control signals. If this is not the case, it is impossible to select the corresponding value. While this degenerate behavior is not disallowed by the special case code, it will result in warnings, and the graph will contain dead code which has not been trimmed. The synthesis tools will typically dissolve such logic. All of the indexes must be compile time constants. The final input contains the default value, which will

Variable	Type	Description
SPECIAL_SELECTOR_INDEX_IN	Int List	The input port number which provides the selector control signal. Defined to be "[0]."
SPECIAL_SELECTOR_VALUES_IN	Int List	The input port numbers which contain the index/value pairs. Defined to be "[1 ~ $num_inputs - 2]."
SPECIAL_SELECTOR_DEFAULT_IN	Int List	The input port number which contains the default. Defined to be "[$num_inputs - 1]."
SPECIAL_SELECTOR_DATA_OUT	Int List	The output port number which is to receive the result. Defined to be "[0]."
SPECIAL_SELECTOR_WIDTH_IDX	Int	The width of the selector control signal. Defined to be "sizeof(in, $SPECIAL_SELECTOR_INDEX_IN)."
SPECIAL_SELECTOR_WIDTH	Int	The width of the values which are being selected (and the output result). Defined to be "sizeof(out, $SPECIAL_SELECTOR_DATA_OUT)."
SPECIAL_SELECTOR_DEPTH	Int	The number of values that the SELECTOR selects between. Defined to be "(($num_inputs - 2) / 2) + 1."

Table A.3: Variables used in the SELECTOR special case implementation code.

be used if none of the index values match the control logic. The output bit-width must be the same as the bit-widths of the selected values, which must all be identical.

The priority MUX is implemented using a VHDL *when* statement of the form:

```
Result  <=  Value0 when Control = Index0 else
            Value1 when Control = Index1 else
            ...
            ValueN when Control = IndexN else
            Default;
```

The "..." in the above code is the reason that this node must be dynamically generated using a special case. The number of clauses in the when statement must be tailored for each SELECTOR node that is created.

A.3.10 STREAM_INIT_FLAG

The STREAM_INIT_FLAG node is used to provide the initial flag which begins a sequence of stream operations.[7] The node has no inputs, and a single 1-bit output which is a constant 0.

A.3.11 UNPACK

The UNPACK node is the counterpart to the PACK node. It is used to separate portions of a larger value into smaller segments. This is similar to accessing elements inside a C struct. Because this node is implemented without the need for any logic cells, it uses the `skip_compile` variable to prevent compilation.

The pack node has a single input and any number (greater than 1) of outputs. The bit-width of the input must be the sum of the output bit-widths. Output 0 will contain the most significant bits of the input, and the last output receives the least significant bits.

A.4 Semiclocked Nodes

Semiclocked nodes are most similar to unclocked nodes, except that they contain a register. There is currently only 1 semiclocked node (DELAY_REGISTER), however it would be possible to treat the COUNTER_CONST node as a semiclocked node in some circumstances.[8] Semiclocked nodes receive the **All_Ready** signal of the section they reside in, but do not contribute a **Ready** signal into it.

A.4.1 DELAY_REGISTER

The DELAY_REGISTER node is used to carry values from 1 iteration of a loop into the following iteration. It contains a single register (initialized to 0) which it provides as its

---

[7] This node could easily be replaced with a constant 1-bit value of 0 in the graphs. It only exists to identify the beginning of the stream-flag chain.

[8] The SA-C* compiler is not able to exploit the benefits afforded by a semiclocked COUNTER_CONST node, so the node is clocked instead.

single output. Each time the enclosing section fires, it replaces the value in its register with the value provided on its input. The DELAY_REGISTER is used by the SA-C* compiler to implement ragged sliding windows. This technique may cause dummy iterations in the loop which must be handled by additional graph logic.

Appendix B

Co-simulation

Verification of the AHAHA layer, including all of its node implementations and implicit structures, is a significant task. Verification is complicated by several factors:

- The space of designs which may be generated using the AHAHA model is practically infinite. There are several nodes in the AHAHA model, which may be combined in a wide variety of ways. Each node may be implemented in any one of several styles. Further, nodes may be parameterized, which may add more dimensions to the design space.

- The VHDL that is generated from the AHAHA graphs is intended to run on an FPGA. Debugging code behavior while it is executing inside the FPGA is quite difficult. The Xilinx Chipscope [63] tool allows a developer to view a portion of the signals in the design as waveforms. In order to use Chipscope, however, the design itself must be modified prior to compilation to include a component to monitor and record signals as they change. The signals which are monitored, and the conditions under which monitoring should occur must be set before the design is compiled. Communication between the debugging tools and the Chipscope component in the FPGA is done via a JTAG port. This requires that the developer has physical access to the hardware. Unfortunately, debugging AHAHA graphs with Chipscope is impractical. There is no way to know beforehand which signals to monitor, or under which circumstances they should be recorded. The number of signals in an

AHAHA graph is too large to simply select them all, and it is difficult to restrict the selection to a smaller number. Lastly, the hardware being debugged is not necessarily accessible by the developer. Since it is impractical to use Chipscope to debug AHAHA-generated designs, the FPGA must be treated as a black box. Black box testing and debugging is quite difficult.

- Technically, the SA-C* compiler generates a "Co-Design," not a stand-alone FPGA bitstream. The AHAHA graphs are not intended to be executed in isolation. They require a communication pathway to the host machine before and after execution of the graph. In many circumstances, they communicate during the execution as well. Many stream programs require host intervention while the FPGA executes. Additionally, SA-C* provides a mechanism to partition programs so that a portion of them executes on the host, and a portion executes on the FPGA. Such programs require an active communication pathway between the host and the FPGA. Co-verification (the task of testing a co-design) is difficult because, typically,' components of the design cannot be tested independently since the behavior is defined by their interactions. These parts are very different in nature; they can be software (ie, host code, FPGA configuration code, board drivers) or hardware (ie, PCI bus, memory, localbus controller).

- Testing a compiler which generates co-designs is a meta-problem. Apart from proving the compiler correct, the only practical way to test it is to use regression testing: A large suite of programs with known outputs are compiled and executed. Their outputs are compared with previously known good results. Proving the SA-C compiler correct is impractical as it contains over a quarter of a million lines of code. When the regression testing identifies an error, there is no indication as to the location or nature of the bug. The only information provided is that a particular program produced incorrect results.

- The accepted method for debugging errors in VHDL code involves creating an application-specific testbench to provide stimulus to the component being tested. Such an approach is impractical because the compiler test suite contains thousands of programs, each of which would require its own handmade testbench. Additionally, the behavior of the hardware is often dependent on the behavior of the host, which may in turn, be dependent upon the hardware. This is possible because the host may invoke the hardware multiple times, with multiple hardware configurations. There is no way to easily simulate this behavior using a static testbench.

To overcome these difficulties, a system for testing AHAHA graphs has been developed which is incorporated directly into the compiler. It is based on automated co-simulation (concurrent execution of the host program, and a simulation of the reconfigurable computing hardware). This system can be utilized by compiler developers to simulate the RCS portion of the SA-C program as it interacts with the host, and other components present on the reconfigurable computing board. The co-simulation tool can be used in conjunction with portions of the Xilinx ISE[64] tools to create cycle accurate and phase accurate simulations of the FPGA during execution.

B.1 Pre-simulation

Before the co-simulation tool can be used, the SA-C* compiler should be verified to ensure that the AHAHA graph that it has generated is correctly formed, and self-consistent. The precise details of this process are outside of the scope of this dissertation, but are outlined briefly here. Some of this information is a review from Chapter 2.

The ability to compile SA-C* programs to a host-only implementation allows the user to verify the correctness of the SA-C* code as early as possible. At this point, no complex optimizations have been applied, and only simple translations have been performed. The ability to confirm the program correctness this early in the process allows the assumption in the later stages that the initial SA-C* code is correct.

Because the SA-C* compiler contains several layers (SA-C*, DDCF, DFG, AHAHA, VHDL), when an error occurs, it is necessary to identify which translation phase is the source of the error. The program being compiled is executable at each layer. The DDCF can be verified by generating C code from the DDCF, and executing it on the host. The remaining layers (DFG, AHAHA, VHDL) can be executed using runtime system (RTS) libraries. Each of these layers have their own RTS library. When the DDCF graphs are partitioned into host and hardware portions, RTS library calls are inserted into the DDCF. When the modified DDCF is converted to C, it will contain calls to the RTS, which allows the host to communicate with the hardware (or a simulation of it) during program execution. Different RTS libraries (with identical APIs) can be used to communicate with the various simulators, or with the FPGA board. Figure B.1 shows

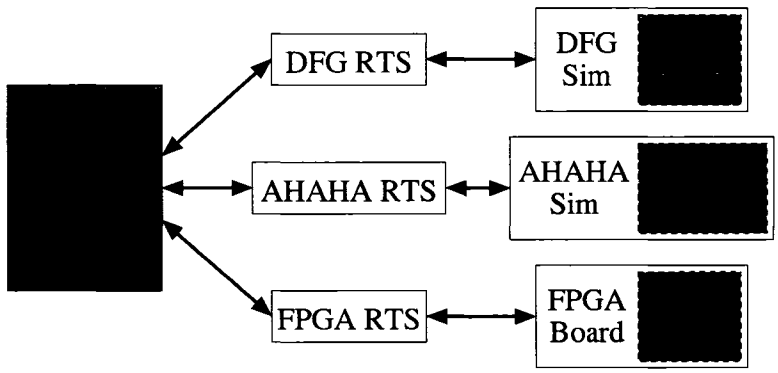


Figure B.1: Runtime Systems.

the various runtime systems and communication paths to the execution models; areas in gray are the program which is being executed and areas in white are fixed. Only one RTS library can be used at a time. The RTS libraries provide a communications layer between the host and the underlying execution model. The host code is the same for all simulations, as well as for the final execution on the FPGA. DFGs are tested using the DFG simulator, and AHAHA graphs are tested using the AHAHA simulator. Trace files from the simulators aid debugging by visualizing the graph execution and allowing comparisons between layers.

B.2 Simulation

To be consistent with the testing model, the tool for verifying the VHDL must be in the form of a new RTS library, and use the same Host C code and API as the other RTS libraries. A co-simulation RTS is more complex than the other RTS libraries because 2 independent platforms are involved (FPGA board and the host processor), as well as the interconnection between them (ie, board drivers and PCI Bus). The portion of the program executing on the board consists of several interacting components, such as memories, clocks, and bus controllers. Since the FPGA code will not operate correctly without it, a simulation of the entire board, including the non-FPGA components is required. Additionally we require a method for communicating between the simulation and the Host C Code. To implement this, ModelSim is used to simulate a VHDL board wrapper which encompasses the ADM-XRC-II board, including the ZBT memories, the localbus, clocks, and the generated FPGA design. The board wrapper contains non-synthesizable VHDL, such as statements which simply print information, so that the RTS can monitor the status of the board. A conduit from the host directly into the simulated localbus controller is created using UNIX FIFOs. Figure B.2 shows the co-simulation RTS library. As before, gray areas represent the program which is being executed and areas in white are constant regardless of the program. ModelSim saves a trace file of every signal in the design during execution, which can be examined later.

B.2.1 Interface to ModelSim

When the co-simulation RTS is initialized, it starts a console-only ModelSim session with stdin and stdout attached via pipes to the RTS. This allows the RTS to send instructions to ModelSim. The first commands instruct ModelSim to compile and execute the VHDL in the design. Additional commands allow the RTS to monitor the progress of the simulation, and detect errors and warnings produced by the simulation. Additional instructions can be sent to ModelSim after the simulation has begun. These instructions may instruct ModelSim to report detailed information back to the RTS which can be used

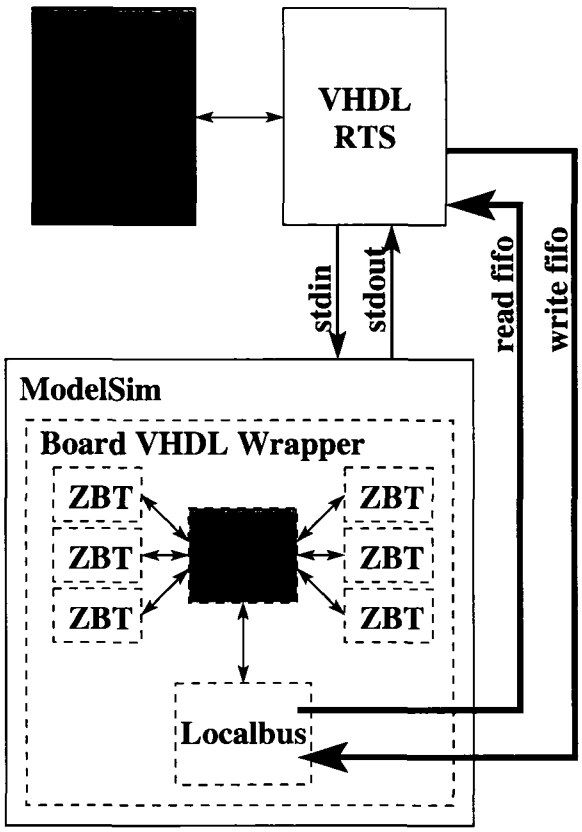


Figure B.2: Co-simulation RTS Structure.

to generate a trace file similar to the DFG and AHAHA simulator. When the RTS needs to read or write data, it issues commands to ModelSim to force control signals which are monitored by the simulated localbus controller. The transfer of actual data (in either direction) is done by the use of the FIFOs on the underlying UNIX file-system. The VHDL simulation of the localbus and the RTS are both designed to read and write data on these FIFOs. This allows the VHDL simulation and the C code to run simultaneously.

B.2.2 Board Memories

Behavioral simulations of the ZBT memories on our board were obtained from the Free Model Foundry (FMF)[65]. The FMF is a free source for simulatable VHDL models of common hardware components. The FMF model which most closely emulates the ZBT memories on the ADM-XRC-II is the "idt71v65603.vhd" model. It is a generic model of a pipelined ZBT memory, which is intended to emulate the behavior of several different ZBT chips, including the one used on our board. Internally, it stores the memory in an array, whose contents can be inspected in ModelSim.

B.2.3 Localbus Controller

The localbus controller consists of several VHDL procedures which were adapted from a testbench provided by Alpha Data for testing their example programs. Some procedures were added which read and write using the named FIFOs, instead of predefined constants. A finite state machine monitors the signals which are forced by the RTS, calls the appropriate procedures, and responds to the RTS with print statements.

B.2.4 FPGA

The VHDL which describes the FPGA behavior is generated from an AHAHA graph, and is inserted as a component into the wrapper. Because the FPGA will implement the behavior in the VHDL file exactly, no modifications to the generated VHDL are required; it is the same synthesizable VHDL which is used to generate the FPGA bitstream. In ModelSim, the simulated FPGA can be viewed separately from the wrapper and diagnosed with all of the signal names intact.

B.2.5 Testing Approach

Using the co-simulation environment, the design can be tested without the need for handmade testbenches. Effectively, the host code functions like a dynamic testbench, providing stimulus to the simulated hardware. The optional trace file which is generated by the RTS allows the developer to compare the dataflow of the FPGA with the trace file from the AHAHA simulator. The data flowing through the top level VHDL entity must be equivalent to the data in the AHAHA simulation trace file, so each clock cycle can be directly compared to identify discrepancies. For debugging purposes, the ability to relate and compare the VHDL, AHAHA, and DFG trace files is vital. Once a difference has been identified, the waveforms from ModelSim can be reopened using an interactive ModelSim session and examined using the GUI. This allows the developer to view the execution in detail, and to debug the low-level implementation of the nodes, their interactions, and other implicit structures.

B.3 Additional Benefits

The testing method described above can be used in conjunction with various Xilinx tools to create more detailed simulations. The Xilinx tools include a utility called "netgen." Netgen is able to generate VHDL from the intermediate forms that are used while creating the BIT file for the FPGA. Depending on which intermediate form is used, the resulting simulation can contain various levels of granularity. Earlier intermediate forms contain high-level components which are structurally similar to the AHAHA generated VHDL. The later forms are made up of FPGA resource primitives, which closely mimic the electrical behavior of the internals of the FPGA, including propagation delays through the logic cells, flip-flops, routing, etc. Figure B.3 shows the Xilinx pipeline of transformations with all of the intermediate files, as well as all of the areas where netgen can be used to generate VHDL.

In the first step, the "`XST`" tool synthesizes the VHDL code generated from the AHAHA graph("a" in Figure B.3), and creates an NGC file. An NGC file is a netlist

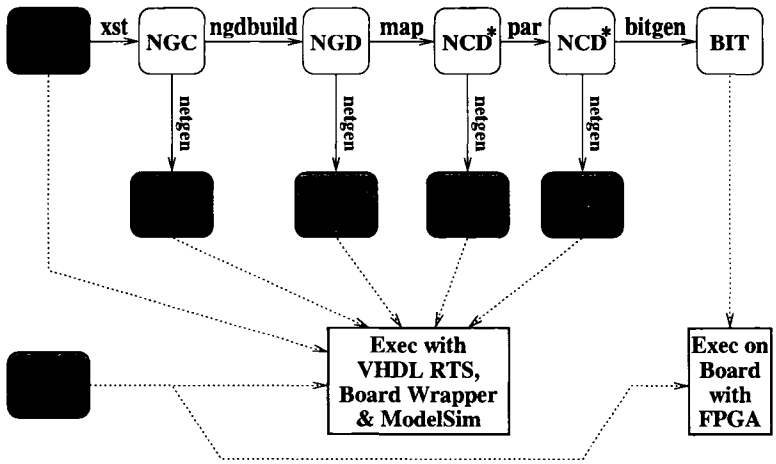


Figure B.3: Simulating at lower levels with the Xilinx tools.

which contains logical design data as well as FPGA design constraints. The NGC file is then converted to an NGD file using the "ngdbuild" tool. The NGD file contains both a logical description of the design expressed in terms of FPGA primitives as well as a description of the original hierarchy contained in the NGC file. The "map" program converts the NGD file into an NCD which contains a physical representation of the design mapped onto FPGA resources. The resources in the NCD are then placed and routed using the "par" tool, and the a new NCD file is created which has the placement and routing annotations included. Finally, the placed and routed NCD file is converted to a BIT file using "bitgen" which can be used to configure the FPGA.

In each of the 4 intermediate forms netgen can be used to generate a VHDL description of the current form of the design. Since all 5 forms of the VHDL (including the initial form) represent the same chip behavior, any one of them can be used by the co-simulation RTS without modification. All that is required is to use the desired VHDL description of the FPGA in the wrapper. When the SA-C* compiler is instructed to produce a co-simulation, it uses the Xilinx tools to produce all of the available intermediate forms, and all 5 VHDL descriptions are saved for potential simulation later. VHDL

configuration blocks provide the ability to select at runtime which of the 5 forms of the VHDL should be used:

a. The VHDL description generated directly from the AHAHA graph can be used to verify that the AHAHA to VHDL translation is correct. This is the most efficient of the 5 models to simulate, since it is the highest level description; this VHDL contains synthesizable processes which are often written in a behavioral style for clarity. The simulator is able to execute these portions as is, which makes it quite efficient. Additionally, numeric operations in the VHDL are treated as high-level operators, instead of bits and circuits. When attempting to identify the source of a bug, this is the first place to check. Bugs that are identified here are typically logical errors in the top level generated VHDL, or in the VHDL implementation of an AHAHA node .

b. The VHDL translation of the NGC file from `XST` can be used to verify that `XST` has interpreted the VHDL in the way it was intended. This is an issue because the synthesizable portion of VHDL is a small subset of the language, and every tool which accepts VHDL as input (`XST`, ModelSim, Synplify, Synopsis, etc.) implements a slightly different subset of the language. When the VHDL deviates from the subset that is implemented, the behavior may become unpredictable. This layer is therefore useful in verifying that the subset of the language which the SA-C* compiler is generating lies within the portion that `XST` implements. Simulation at this level is still reasonably efficient because numeric values are still treated as such; however, the behavioral processes have been converted into logically equivalent circuits by `XST`, which makes the simulation slightly slower than in "a" above. Errors uncovered in this layer can usually be repaired by a slight change in VHDL coding style.

c. When the NGD file from ngdbuild is converted to VHDL, the resulting code is composed entirely of primitives which can be created easily out of FPGA hardware,

such as MUXes, n-bit flip-flops, adder/subtracters, and shift registers. This layer isn't very useful for debugging or analysis, but is included for completeness.

d. At this level, the VHDL generated from the post-map NCD file has been mapped onto the set of primitives that are available on the FPGA such as 4-input 1-output lookup tables, 1-bit flip-flops, BlockRAMs, and SRL16s. In addition, netgen provides information about the propagation delay through each one of these primitives. Since the design has not been placed and routed yet, there is no timing information available regarding the delays on the wires between the primitives, so the delays are best-case. This simulation can be used to identify critical paths which lower the clock frequency of the design. Using this information, the compiler developers can identify nodes which have an abnormally long internal path delay, and repair them to achieve higher clock frequencies. This would be complicated by the routing in the next phase because it would be difficult to tell if the delays are due to the routing, or to the structure of the design itself. Unfortunately, this simulation is rather slow to execute because each bit is handled independently. A 32-bit adder can no longer be implemented arithmetically; its function needs to be created from the FPGA primitives, and 32 separate 1-bit wires will be used for each numerical value.

e. The lowest level available is the VHDL generated from the post-par NCD. This shows the most precise simulation of the behavior inside the FPGA. Because this simulation is phase accurate, any timing glitches in the FPGA configuration will also be visible in this layer and, therefore, can be debugged. If this simulation behaves correctly, and running on the board does not, then we can be confident that there is a physical defect in one of the components on the ADM-XRC-II board, the PCI bus, or the board drivers. Because of the granularity of this design, this is also the most time consuming simulation of the 5.

To summarize: Option a is used to debug logical errors in the SA-C* generated VHDL. Option b is used to find discrepancies in coding style between the SA-C* generated VHDL and what is expected by `XST`. Option c is not useful for debugging because the design is described in terms of components which are not FPGA primitives. Option d is used to understand the timing behavior of the design and to locate critical paths. Option e is used to simulate the exact behavior of the FPGA.

B.4 Related work

There have been many projects which supply environments for simultaneous hardware/software simulation and debugging. Many of these approaches are similar to the one described in this appendix. However, none of them utilize a ModelSim simulation in the same way, or allow simulation of the designs at intermediate stages of synthesis. Many rely on proprietary solutions, or allow only a single level of simulation. Debugging environments which are similar to the AHAHA debugging solution are described here.

B.4.1 JHDL

The development framework for JHDL [66, 67, 68] provides a unified environment for both simulation and hardware execution. They use a proprietary GUI for debugging hardware as well as simulation. When a simulation is used, the behavior of the hardware is emulated by the host and displayed in the GUI. When the hardware is being executed, a readback of the FPGA is used to extract the runtime state in the middle of its execution. The extracted state information is then displayed using the same user interface. Effectively, they are able to simulate at a high-level with the JHDL simulator, or to execute at the lowest level (hardware), and debug at either level using the same tool. Performing a readback from the FPGA is quite expensive, and single-stepping through a hardware execution using this method is impractical. To address this problem, they combine the two methods. The environment is able to extract the state from a running FPGA (via a readback), and import it into the simulator, allowing the execution to continue within the simulator from that point. They also claim the process can be

done in reverse. That is, using the simulator to augment the state stored in a bitstream which was read back from the FPGA, and then re-download it to allow the FPGA to continue from that point. This approach could be quite useful to debug situations where the FPGA is physically damaged, or malfunctioning at the circuitry level. However, for functional debugging, this feature is not required. The AHAHA debugging environment only addresses functional debugging, and is unable to debug physical hardware defects. However, by simulating using intermediate representations of the program as it is compiled by the `ISE` tool suite, it removes the need for debugging the functionality of the program on the hardware.

B.4.2 SystemC

SystemC [69] is a hardware description language similar to VHDL or Verilog, except that it is implemented in C++. SystemC descriptions can be compiled using a C++ compiler to generate a register transfer level simulator for the described design. However, SystemC can also be synthesized using a variety of HDL synthesis tools such as ones available from Celoxica [70], Synopsys [71], or SystemCrafter. [72] If the SA-C compiler were redesigned to generate SystemC instead of VHDL, the SystemC could be compiled to create a simulator which could then be linked with the host C. The resulting design could then be debugged using standard C and C++ debugging tools. There would be no need for a VHDL simulation tool such as ModelSim. Additionally, the SystemC description could be used to generate a netlist which could then be used by the Xilinx tools to generate an FPGA bitstream. However, this technique would remove the current ability to debug using the intermediate program representations during compilation of the design. The Xilinx ISE tools are able to produce VHDL descriptions of their intermediate forms, but are not able to generate SystemC.

B.4.3 Celoxica

The Celoxica DK design suite [58] provides an "HW/SW Co-Verification" solution for its Handel-C language [73, 74, 75]. Their verification tools allow the hardware description

(in Handel-C) and host code (in C) to run and interact during the simulation. Because the Handel-C used in the simulation is executed at a high-level, it is impossible to identify low-level errors in the execution, including timing glitches or FPGA level implementation details. The DFG or AHAHA simulators are similar to this type of debugging.

B.4.4 Verilog PLI

A common approach to co-verification is to use concurrently running C code and Verilog simulation, which communicate via BSD sockets. The Verilog simulation directly accesses the sockets via the Verilog Programming Language Interface (PLI). From the hardware simulation, the sockets function like hardware FIFOs; the C code uses standard socket communication routines.

B.4.4.1 Becker et al.

Becker, Singh, and Tell [76], used this method to perform co-simulations of their Network Interface Uint (NIU). The NIU is a communication interface for a gigabit network. It contains several hardware and software components which communicate via a backplane. Some of the software components are firmware, which execute within the device, and some are used externally to monitor and control it. The software and hardware components of the NIU system were designed concurrently and debugged using the PLI/Sockets-based method. This approach allowed the various components to operate independently with simple communication pathways between them. This mirrors the intended behavior of the final design. They explored other alternatives before committing to the PLI/Sockets implementation. They determined that incorporating the software components into the Verilog simulation, as user extensions, may cause them to interfere with each other, and permit a tighter integration between the components than is intended. They also explored the possibility of using a behavioral model of the firmware processor (a Cypress CY7C611 SPARC). This would allow the Verilog simulation to execute many of the software components indirectly. However, such processor models are expensive, and would not address the issue of the external monitoring software. The

PLI/Sockets method addressed all of these issues, and allowed the 3 execution models (hardware, firmware, and software) to communicate while executing separately.

B.4.4.2 Coumeri and Thomas

A similar approach was used by Coumeri and Thomas [77] to test hardware-software co-designs. They used the PLI/Sockets method because speed was a primary concern, and this approach allows the C host program and the Verilog simulation to execute on separate hosts connected with a network. Using separate machines leverages the parallelism in the simulation to speed up the process. In the model described, there is no physical hardware, only a simulation, so the C code running on the host simply uses BSD sockets directly, and no attempt is made to emulate a hardware interface. This is quite different than the simulation model used for SA-C*, in which the host C code is designed to communicate with the hardware, and the simulations emulate the hardware API. This allows the same host code to be used for all levels of the simulation, as well as hardware execution.

B.4.4.3 PipeRench

Prior to 2002, the PipeRench [78] project was forced to use a simulator of their hardware because the hardware had not yet been fabricated. Likewise, the PCI interface to the chip was implemented in a Verilog simulation, but did not exist physically. In order to develop the system, a simulation of the chip and its PCI interface were required. The PipeRench chip is accessed using an API to configure, monitor, and communicate with the device. Two implementations of the same API are available, one to communicate with a Verilog Simulation using PLI/Sockets, and another to communicate to the hardware via the PCI bus. The simulation API provides a layer of abstraction, so that the host processor which controls the PipeRench does not need to take into account whether or not the hardware is physical, or simulated.

B.4.4.4 SRC

SRC [61] provides a comprehensive debugging environment, involving 4 execution modes, which closely mimic the execution modes for SA-C*. In all cases, the same host code is used, and different libraries are invoked to supply the various modes of hardware execution. The lowest level is "hardware" in which FPGAs are used. In addition to this, there are 3 modes used for debugging and verification. In the "debug" mode, the hardware portion of the program is compiled and executed directly on a standard processor using software-based functional emulation of the hardware components. This is similar to compiling SA-C* code to run on the host CPU. Next, the "dataflow emulation" mode of execution mirrors the DFG and AHAHA simulation. It is a token-driven dataflow simulation to verify the correctness of the dataflow graphs. The "simulation" mode used at SRC is the most similar to the co-simulation model used for the AHAHA. It utilizes the VCS Verilog simulator from Synopsis.[1] Interestingly, the board components outside the FPGA are not included within the simulation, but are handled in software, via the Verilog PLI. The simulation communicates with the host portion of the program using shared memory and semaphores.

B.4.4.5 Comparison to AHAHA

The PLI/Sockets technique is quite similar to the method employed in the AHAHA Co-simulation environment. Because the AHAHA uses FIFOs and pipes instead of sockets, it is limited to point-to-point communication involving only two clients. The PLI/Sockets method, however, would allow the design to be decomposed into smaller independent units (such as in the work by Becker et al. [76]), but may also introduce some nondeterminism, which may be undesirable for a debugging environment. Because the SA-C* model only uses two partitions (Software and Hardware) this is not an issue. Another major difference is that the hardware in the AHAHA simulation system is treated as a

[1] ModelSim can also be used as the hardware simulation engine, although VCS is preferred.

slave of the host. The host (specifically, the VHDL Simulation RTS) controls the hardware simulation at a very low level, advancing the simulation explicitly, and monitoring its execution continuously. In the PLI/Sockets method, the network is treated as a simple FIFO, and the hardware runs concurrently. Neither the host, nor the hardware is considered to be master or slave.

B.5 Conclusions

The method of testing the AHAHA generated VHDL is based on a co-simulation technique using UNIX FIFOs. The host code is able to communicate with an FPGA board simulated in ModelSim as if it were physical hardware. The board simulation contains VHDL descriptions of all of the board components including the FPGA itself. The co-simulation approach described allows testing and analysis of all parts of the complete co-design. In essence it allows the compiler to perform automated co-verification for any SA-C* program, at many levels.

In combination with the Xilinx ISE tools (in particular, netgen) it is possible to use the simulation environment at varying levels. At the highest level of simulation, it allows functional verification of the VHDL generated by the compiler. At the lowest level of detail, the FPGA simulation is phase accurate and mimics the exact hardware behavior down to the individual CLB.

Compared with the simulation solutions discussed in Section B.4, the support of simulation in the SA-C* suite is more complete and flexible. It allows simulation at many levels: Host-Code, DFG, AHAHA, and 5 varying levels of VHDL simulation. It allows debugging of the user code, as well as the various intermediate stages of the compiler. Despite its strengths, there are two features which may be useful in debugging which are not implemented in the VHDL co-simulation environment. First, a system based on the Xilinx Chipscope tool may provide a useful method of debugging the hardware when its simulation does not match the physical execution. While no such event has occurred which would require this level of debugging, faulty drivers or hardware may cause such a

feature to be useful. Likewise, a method of debugging based on FPGA readback would provide an even lower level of debugging which would allow verification of the physical integrity of the chip. Utilizing readback for debugging purposes requires extra work in order to be able to relate the FPGA bitstream back to the registers in the design, and is quite difficult, though possible. It is debatable as to whether or not the benefits afforded by this approach would outweigh the effort to implement such a system.

REFERENCES

[1] MicroBlaze. Microblaze - from wikipedia. Information about the MicroBlaze processor is available from Wikipedia, `http://en.wikipedia.org/wiki/MicroBlaze`.

[2] Xilinx, Inc. *MicroBlaze Processor Reference Guide: Embedded Development Kit, EDK 8.1i*, Oct. 2005. `www.xilinx.com`.

[3] Picoblaze. Picoblaze - from wikipedia. Information about the PicoBlaze processor is available from Wikipedia, `http://en.wikipedia.org/wiki/Picoblaze`.

[4] Xilinx, Inc. *PicoBlaze 8-bit Embedded Microcontroler User Guide: for spartan-3, Virtex-II, and Virtex-II Pro FPGAs*, Jun. 2004. `www.xilinx.com`.

[5] Nios. Nios - from wikipedia. Information about the Nios processor is available from Wikipedia, `http://en.wikipedia.org/wiki/Nios_(computer_processor)`.

[6] Altera, Inc. *Nios Embedded Processor: 16-Bit Programmer's Reference Manual*, Jan. 2004. `www.altera.com`.

[7] Nios-II. Nios-II - from wikipedia. Information about the Nios-II processor is available from Wikipedia, `http://en.wikipedia.org/wiki/Nios_II`.

[8] Altera, Inc. *Nios II Processor Reference Handbook*, May 2006. `www.altera.com`.

[9] FPGA. Fpga - from wikipedia. Information about FPGAs is available at from Wikipedia, `http://en.wikipedia.org/wiki/FPGA`.

[10] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, 1993.

[11] Xilinx, Inc., San Jose, CA. *The Programmable Logic Databook*, 1998. `www.xilinx.com`.

[12] Patterson C. High performance des encryption in virtex fpgas using jbits. In *IEEE Symposium on Field-programmable Custom Computing Machines*, April 2000.

[13] W. Böhm and B. Draper. The cameron project. Information about the Cameron Project, including several publications, is available at the project's web site, `www.cs.colostate.edu/cameron`.

[14] J. Hammes, R. Rinker, W. Böhm, W. Najjar, B. Draper, and R. Beveridge. Cameron: High level language compilation for reconfigurable systems. In *Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[15] W. Najjar, B. Draper, W. Böhm, and J. R. Beveridge. The Cameron Project: High-level programming of image processing applications on reconfigurable computing machines. In *PACT '98 - Workshop on Reconfigurable Computing*, pages 83–88, Oct. 1998.

[16] J. Hammes and W. Böhm. *The SA-C Language - Version 1.0*, 1999. Document available from `www.cs.colostate.edu/cameron`.

[17] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. In *Supercomputing Journal 21*, pages 117–130, 2002.

[18] W. Böhm, R. Beveridge, B. Draper, C. Ross, and M. Chawathe. SA-C: Single assignment C. high-level, high-speed FPGA programming. In *Dr. Dobbs Journal*, pages 60–64, May 2003.

[19] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. Najjar, and W. Böhm. An automated process for compiling dataflow graphs into reconfigurable hardware. In *IEEE Transactions on VLSI Systems*, pages 130–139, 2001.

[20] R. Rinker. *Compiling Dataflow Graphs Into Hardware.* PhD thesis, Colorado State University, 2005.

[21] C. Ross. A VHDL runtime system for dataflow execution on reconfigurable systems, April 2000.

[22] M. Chawathe. *Hardware Compilation of Streams and Processes.* PhD thesis, Colorado State University, 2006.

[23] G. Kahn. The semantics of a simple language for parallel processing. In *IFIP Congress*, pages 471–475, 1974.

[24] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, Amsterdam, Holland, 1977.

[25] Xilinx, Inc. *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*, Oct. 1999. `www.xilinx.com`.

[26] Annapolis Micro Systems Website, 2005. `www.annapmicro.com`.

[27] Annapolis Micro Systems, Inc., Annapolis, MD. *STARFIRE Reference Manual*, 1999. `www.annapmicro.com`.

[28] Annapolis Micro Systems, Inc., Annapolis, MD. *WILDSTAR Reference Manual*, 1999. `www.annapmicro.com`.

[29] Alpha Data Website, 2005. `www.alpha-data.com`.

[30] Alpha Data, Inc., San Jose, CA. *ADM-XRC-II Reference Manual*, 2005. `www.alpha-data.com`.

[31] *8Mb: 512K x 18, 256K x 32/36 Pipelined ZBT SRAM datasheet*, 2001. `www.micron.com`.

[32] PCI 9656BA Data Book, 2003. `www.plxtech.com`.

[33] Xilinx, Inc. *XST User Guide*, 2005. `www.xilinx.com`.

[34] IA-64. IA-64 - from wikipedia. Information about the IA-64 instruction set architecture is available from Wikipedia, `http://en.wikipedia.org/wiki/IA-64`.

[35] Itanium. Itanium - from wikipedia. Information about the Itanium processor is available from Wikipedia, `http://en.wikipedia.org/wiki/Itanium`.

[36] Jeffrey M. Arnold. Software configurable processors. In Nikitas Dimopoulos and Sanjay Rajopadhye, editors, *Application-Specific Systems, Architectures, and Processors*, pages 45–49, Steamboat Springs, CO, 2006. IEEE Computer Society Press.

[37] Stretch. Information about Stretch, including their S5 engine is available at their web site, `www.stretchinc.com`.

[38] Wim Böhm. *CWI Tract 6: Dataflow Computation.* Centre for Mathematics and Computer Science, Amsterdam, Netherlands, 1984.

[39] Block cipher modes of operation. Block cipher modes of operation - from wikipedia. Information about CBC encryption/decryption can be found at `http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation`.

[40] E. Rijpkema, B. Kienhuis, and E. Deprettere. Compilation from matlab to process networks, 1999.

[41] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. pages 13–17.

[42] T. Harriss, R. Walke, B. Kienhuis, and E. Depettere. Compilation from matlab to process networks realized in fpga, 2001.

[43] Paul L. Master. Reconfigurable hardware and software architectural constructs for the enablement of resilient computing systems. In Nikitas Dimopoulos and Sanjay Rajopadhye, editors, *Application-Specific Systems, Architectures, and Processors*, pages 50–55, Steamboat Springs, CO, 2006. IEEE Computer Society Press.

[44] ElementCXI. Information about ElementCXI, including the ECA is available at their web site, `www.elementcxi.com`.

[45] C. Ebeling D. C. Green and P. Franklin. RaPiD – reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, Darmstadt, Germany, 1996. Springer-Verlag.

[46] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping applications to the rapiD configurable architecture. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 106–115, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[47] Carl Ebeling. Configurable computing platforms - promises, promises. In Nikitas Dimopoulos and Sanjay Rajopadhye, editors, *Application-Specific Systems, Architectures, and Processors*, pages 3–4, Steamboat Springs, CO, 2006. IEEE Computer Society Press.

[48] D. Wilde. The ALPHA language. Technical Report PI 827, IRISA, Rennes, France, May 1994. Also published as INRIA Research Report 2295.

[49] C. Mauras. *ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.

[50] H. Le Verge. *Un environnement de transformations de programmmes pour la synthèse d'architectures régulières.* PhD thesis, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, Oct 1992.

[51] F. Dupont De Dinechin, P. Quinton, and T. Risset. Structuration of the alpha language. In W.K Giloi, S. Jahnichen, and B.D. Shriver, editors, *Massively Parallel Programming Models*, pages 18–24. IEEE Conmputer Society Press, 1995.

[52] Alpha homepage, 2006. Information about the ALPHA language is available online at `www.ece.byu.edu/faculty/wilde/Alpha`.

[53] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies, 1986.

[54] A.-C. Guillou, F. Quiller, P. Quinton, S. Rajopadhye, and T. Risset. Hardware design methodology with the alpha language. In *FDL 2001: Forum on Design Languages*, September 2001.

[55] R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators, 2000.

[56] S. C. Goldstein and H. Schmit. Instant hardware: Fast compilation to scalable reconfigurable hardware, Oct. 1999. Presentation made to DARPA ACS-PI Meeting, San Juan, Puerto Rico.

[57] Rapport INC. The kilocore chip. `www.rapportincorporated.com`.

[58] Celoxica, Ltd., Abingdon, Oxfordshire, UK. *Celoxica DK Design Suite Datasheet*, 2005. `www.celoxica.com`.

[59] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[60] David May. Occam. *SIGPLAN Not.*, 18(4):69–79, 1983.

[61] SRC. `www.srccomputers.com`.

[62] Don Davis. Forge: High performance hardware from high-level software, September 2002.

[63] Xilinx, Inc., San Jose, CA. *Chipscope Software Manuals*, 2006. `www.xilinx.com/literature/literature-chipscope.htm`.

[64] Xilinx, Inc., San Jose, CA. *Software documentation for ISE 7.1i*, 2005. `www.xilinx.com`.

[65] Free Model Foundry. `www.freemodelfoundry.com/welcome.html`.

[66] Peter Bellows and Brad Hutchings. JHDL – an HDL for reconfigurable systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[67] Brent E. Nelson. The mythical CCM in search of useable (and reusable) FPGA-based general computing machines. In Nikitas Dimopoulos and Sanjay Rajopadhye, editors, *Application-Specific Systems, Architectures, and Processors*, pages 5–11, Steamboat Springs, CO, 2006. IEEE Computer Society Press.

[68] Brad Hutchings, Peter Bellows, Joseph Hawkins, Scott Hemmert, Brent Nelson, and Mike Rytting. A CAD suite for high-performance FPGA design. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–24, Los Alamitos, CA, 1999. IEEE Computer Society Press.

[69] SystemC. SystemC homepage. `www.systemc.org/`.

[70] Celoxica, Ltd., Abingdon, Oxfordshire, UK. *Agility Compiler Datasheet*, 2005. `www.celoxica.com`.

[71] Synopsys System Studio, 2005. `www.synopsys.com`.

[72] Systemcrafter SC, 2005. `www.systemcrafter.com`.

[73] Handel-C. Information about Handel-C is available at the project's web site, `www.celoxica.com/tech/handel-c/default.asp`.

[74] Celoxica, Ltd., Abingdon, Oxfordshire, UK. *Handel-C Language Reference Manual*, 2005. `www.celoxica.com`.

[75] O.H.C.Group. OXFORD hardware compiler group, the Handel Language. Technical report, Oxford University, 1997.

[76] David Becker, Raj K. Singh, and Stephen G. Tell. An engineering environment for hardware/software co-simulation. In *Design Automation Conference*, pages 129–134, 1992.

[77] S. L. Coumeri and D. E. Thomas. A simulation environment for hardware-software codesign. In *International Conference on Computer Design*, October 1995.

[78] Ronald Laufer, R. Reed Taylor, and Herman Schmit. PCI-PipeRench and the SwordAPI: A system for stream-based reconfigurable computing. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 200–208, Los Alamitos, CA, 1999. IEEE Computer Society Press.