A MULTI-PARADIGM C++-BASED HARDWARE DESCRIPTION LANGUAGE

A Thesis
Presented to
The Academic Faculty

by

Chad D. Kersey

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2019

A MULTI-PARADIGM C++-BASED HARDWARE DESCRIPTION LANGUAGE

Approved by:

Saibal Mukhopadhyay, Committee Chair
School of Electrical and Computer Engineering
Georgia Institute of Technology

Sudhakar Yalamanchili, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Thomas Conte
College of Computing
Georgia Institute of Technology

Tushar Krishna
School Electrical and Computer Engineering
Georgia Institute of Technology

Hyesoon Kim
School of Electrical and Computer Engineering
Georgia Institute of Technology

Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: 09/30/2019

For Sudha, without whom none of this would have happened.

ACKNOWLEDGEMENTS

The work that eventually became this dissertation started at the end of 2012, though the influences leading to its creation reach back much further, and the number of people whose input shaped it is staggering. It is difficult to work on something for so many years without owing quite a bit of gratitude. I am sure I will not be able to name every contribution here, but I will attempt to name the names who still come most prominently to mind.

Much of the work that would become this dissertation was done in collaboration with the Structural Simulation Toolkit team at Sandia National Laboratories' Computer Science Research Institute and/or funded by the US Department of Energy. Arun Rodriguez especially has been instrumental in his role as a mentor through five separate internships at Sandia, beginning when C++ 11 was still the upcoming C++ 0x standard and ending after C++ 14 had been published. CHDL would have likely been built using another language if it had not been for my work with SST.

My many lab mates in Georgia Tech's Computer Architecture and Systems Lab were helpful and supportive, both in matters technical and mundane. Nawaf Almoosa, Eric Anger, Xinwei Chen, Dhruv Choudhary, Greg Diamos, Naila Farooqui, Minhaj Hassan, Andrew Kerr, Si Li, Adam McLaughlin, Karthik Rao, Ifrah Saeed, William Song, Blaise Tine, Jin Wang, Haicheng Wu, and Hugh Xiao all contributed to making my time at Georgia Tech enjoyable and memorable and participated in technical discussions that strengthened the work presented here. Meghana Gupta in particular contributed important work to the HARP architecture used in Harmonica, including contributing a compiler back-end. Jeff Young has been instrumental in the final stages of writing and preparing this document, and his feedback has been a steady pressure forcing text into what would otherwise be voids.

My advisor throughout nearly all of this work and for years before it was the late Sudhakar Yalamanchili, whose patient guidance pushed me along this program of research. I would be hard pressed to name a single person who has had more influence on my work

or who has shaped my career more.

Finally, I must acknowledge all of the people who contributed much needed community and camaraderie, including Brian Akers, Josh Cowan, Ryan and Emily Curtin, and John Frey. My wife Ashley and our little trembly mutt Sarah, both of whom will surely be quite happy to spend an evening with me in which I am not staring at a laptop screen, have been a dependable, stabilizing force in my life and a great source of emotional support throughout a long program of study, and for that I cannot thank them enough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This dissertation presents a generative hardware description library for C++, the CHDL Hardware Design Library or CHDL, along with a body of supporting libraries and a description of a core design implemented using this library. The supporting libraries extend the level of abstraction covered by CHDL from the solely constructive and generative to a range of hardware description paradigms including the register transfer level (RTL), an implementation of Bluespec-like guarded atomic actions (GAA), and a novel pipeline-oriented HDL providing a high-level synthesis flow from algorithmic descriptions of pipelined hardware. Design input using all of these paradigms is converted by CHDL into an in-memory gate level netlist that may be simulated, emitted as synthesizable Verilog, or technology mapped to a standard cell library for area and energy estimation. Access to this netlist, dubbed "netlist introspection", is provided by the CHDL API, allowing novel optimizations and transformations to be performed by the designer.

CHAPTER I

INTRODUCTION

The end of planar silicon brought with it the last traces of Dennard scaling. Denser logic circuits in the latest process technology nodes no longer provide improvements in energy per operation, precipitating the era of dark silicon. Homogeneous multi-core processors with ever-increasing core counts are a poor choice in this regime since, when faced with the choice of adding cores that must by duty cycled or simply adding cache banks, the cache banks are likely to have a greater marginal impact on performance. Heterogeneous multi-cores containing a variety of different types of accelerator cores or homogeneous multi-cores devoting their dark silicon area to larger and more diverse instruction sets are a more appropriate choice, but increase the demand for design and verification, in turn increasing demand for designer productivity.

Fixed-function accelerators, accelerators implementing specialized instruction sets, and coprocessors adding instruction set features to existing cores are likely to improve performance but must be specified, designed, and verified. While there are tools for the early evaluation of architectures and the transaction-level modeling of systems, once these phases of the design process have been finished, high-fidelity modeling of area and power consumption of accelerators must be performed from a toolchain that can produce a register transfer level description and, ultimately, a gate-level design. The language used as the input to this toolchain, in which the processor architecture is ultimately expressed, is typically a hardware description language. High-level synthesis tools exist to convert designs expressed as programs in high-level programming languages to hardware designs, but the quality-of-results obtained by using such tools for specialized designs such as instruction set processors is often low, while the microarchitecture is specified by the HLS tool and not the designer.

The history of hardware description languages is almost as long as the history of programming languages themselves, and many different languages providing the ability to express hardware at different levels of abstraction employing different paradigms have been created. The industry standard for expressing place-and-route ready netlists is the structural subset of Verilog or simply the SPICE netlist. These are the lowest-level hardware description languages available; below netlists there are only polygons. Synthesizable logic has been expressed for decades using the register transfer level subsets of HDLs such as VHDL and Verilog. The process of converting RTL to high-quality logic that meets timing constraints is not trivial, but it is straightforward and does not entail many design decisions that cannot be efficiently and effectively solved by software-based optimization. The same cannot be said for levels of abstraction above the register transfer level.

The first step beyond the register transfer level is simply generative, like generator loops provided by Verilog or the fundamental structure of constructive HDLs like Chisel [4]. In a generator, register transfer level functions or logic block instances are created by some form of program or loop. Beyond this are dataflow paradigms that have locally RTL-like syntax, but automatically generate structures such as ready and valid signals, beginning to obscure the timing of the generated code from the designer. Above this level of abstraction are systems that automatically generate fairly complex structures such as pipelines. Although its input is simply an annotated dataflow graph, the output of the Sehwa [44] pipeline generator is a pipelined implementation with timing guided by a set of constraints but inscrutable to the designer. Beyond this are high-level synthesis tools that produce customized data paths to implement software expressed in a high-level language originally intended for creating software, including the behavioral dialects of hardware description languages such as VHDL, Verilog, and SystemC.

CHDL, which along with its auxiliary libraries and core designs is the topic of this dissertation, is a C++ library enabling hardware design, using the language-expanding features of C++, such as operator overloading and template metaprogramming, to produce a hardware-oriented domain-specific language. The base CHDL library provides mechanisms to describe combinational and synchronous sequential logic on signals of one or multiple

Contents of `blink.cpp`:

```
#include <fstream>
#include <chdl/chdl.h>

using namespace std;
using namespace chdl;

int main() {
  node x;
  x = !Reg(x);
  OUTPUT(x);

  ofstream vcd("blink.vcd");
  run(vcd, 100);
  return 0;
}
```

Command line:

```
$ c++ -o blink blink.cpp -lchdl
$ ./blink
```

Waveform:

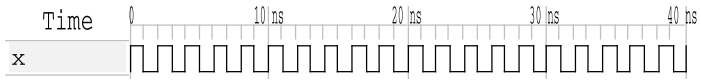


Figure 1: A simple design cycle using CHDL.

bits, to simulate these designs, to technology map these designs to standard cell libraries for analysis, or to output these designs as synthesizable Verilog for further processing by external applications. The decision to restrict sequential logic to synchronous designs was made to simplify the library design as well as formal analysis of the designs made using the library and to guarantee compatibility with contemporary logic synthesis tools. C++ was chosen for a combination of its widespread install and knowledge base combined with its support for features, such as operator overloading and template metaprogramming, enabling the creation of APIs that function as domain-specific languages. The use of C++ for CHDL itself also makes it possible to implement hardware description, synthesis, and simulation all in the same programming language, a feature unique among extensible HDLs. Template metaprogramming in particular allows for compile time type safety for data types representing signals, including structured signals. A mechanism called netlist reflection is provided in which the design is maintained as an in-memory netlist, and functions are provided for reading and manipulating this netlist, allowing programs using CHDL to interact with the simulator and process the in-memory design for optimization and analysis.

The design cycle using CHDL, illustrated in Figure 1 is the software development edit-compile-debug cycle, potentially using a waveform viewer as a part of the debug step. This allows the design and simulation of complex digital hardware using only CHDL and a C++

compiler.

A follow-on library, the CHDL template library, expands CHDL with support for structured signals and interfaces, including a standard interface for memory system components, numeric types including fixed and floating-point real numbers and signed and unsigned integers, and provides a few basic logic elements including FIFOs and stacks. Most designs that have been built using CHDL use CHDL template library data types, as these data types provide convenient ways to represent structured signals containing more than a simple numerically-addressed array of bits. A separate library built on top of the CHDL template library's support for directed and structured signals provides a way to load and store pieces of designs as CHDL netlists, enabling the distribution of IP blocks and the construction and simulation of quite large designs incorporating diverse pieces, with no requirement that the pieces were themselves constructed using CHDL.

The HARP family of instruction sets was created as a family of extensible GPGPU-like RISC instruction sets for systems in which programmable accelerators are treated as first-class cores with the ability to handle both internally-generated exceptions and external interrupts. HARP is designed primarily as a family of instruction sets, with both fixed-point and floating-point real number arithmetic and with a configurable word size, register file size, and degree of SIMT parallelism. As the intent of the CHDL system is to provide a multi-paradigm system for prototyping accelerator architectures, the HARP architectures provide a natural demonstration vehicle.

The combination of CHDL and the template library form a fairly complete register transfer level hardware description language supporting the use of C++ templates for generics and C++ control flow constructs for generators. This was successfully employed to produce Harmonica, an implementation of the HARP instruction set architectures. A runtime library, compilation system, and set of applications were developed to evaluate this family of architectures, and the parameterizability inherently afforded by the template system of C++ as used by CHDL enabled a wide range of Harmonica designs to be evaluated, from versions providing no SIMT parallelism at all, to 1024-thread versions with multi-kilobyte register files.

Harmonica demonstrates the complexity of design that may be undertaken using only the RTL features of CHDL, but rapid development of accelerators and architectures to keep pace with the ever-growing demand for computing performance requires the exploration of higher-level hardware description paradigms. Guarded Atomic Actions (GAA) as popularly implemented in the Bluespec System Verilog [40] hardware description language, is a dataflow paradigm for implementing complex hardware systems as hierarchies of finite state machines, each described using an RTL-like syntax that incorporates automatic generation of "ready" and "valid" signals at the interfaces and maintenance of atomicity for updates to registers. The CHDL GAA library implements guarded atomic actions on top of existing CHDL structures, allowing any CHDL data type to be used as a data type for registers in GAA systems and also allowing any CHDL modules to be instantiated and used from within GAA designs.

This multi-paradigm interoperability is a key feature of CHDL, best illustrated with an example. A common piece of example code for descriptions of GAA libraries in the literature is Euclid's algorithm for finding the greatest common divisor of two integers. This can also be used to find the greatest common divisor of other objects such as polynomials over finite fields. In Chapter 5, an example of an abstract GCD algorithm is provided. This has been used with both the CHDL core library's `bvec<N>` type, which behaves as an unsigned integer and a `gf<N>` type, which behaves as a polynomial. The `gf<N>` code, originally intended for RTL implementations of cryptographic accelerators and implemented before CHDL-GAA was available, may be used in a GAA context with no additional glue code.

The level of abstraction provided by domain specific languages that can be built around CHDL is not limited to structures like guarded atomic actions in which signal names always represent the same value. It is also possible, by returning a context-dependent CHDL signal each time an object representing a variable is referenced, to add additional parallelism through techniques such as pipelining. Cheetah is a pipeline-oriented hardware description language built on top of CHDL, and provides an example of a more abstract paradigm implemented within the CHDL framework.

Cheetah pipelines are described as a set of stages, each of which can be considered

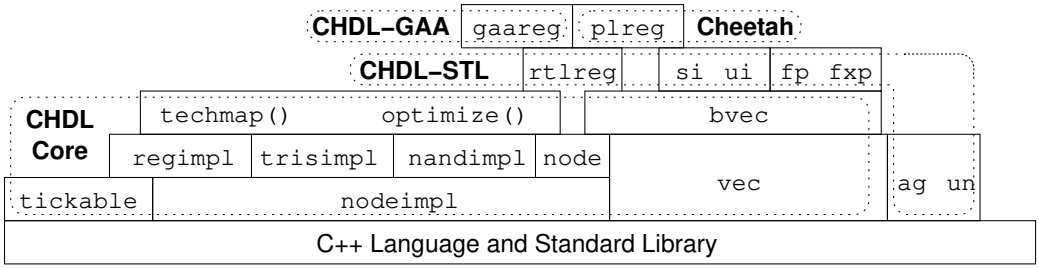


Figure 2: Interfaces within CHDL and its extensions are built up as a permeable hierarchy.

equivalent to a basic block in software. Multiple threads of execution may propagate through the pipeline simultaneously, communicating through non-pipeline-carried signals. In this analogy, the pipeline-carried signals are equivalent to variables accessed by the program and the flow of data between pipeline stages steered by multiplexers is equivalent to control flow. This analogy can be used as a high-level synthesis framework to generate parallel, pipelined fixed-function accelerators implementing specific algorithms, or as a hardware-oriented design paradigm for describing pipelined architectures. Examples of both are explored.

1.1 Summary of Contributions

Performing the kind of low-level description necessary to model accelerators with accurate timing and energy consumption requires the use of hardware description languages. It is not clear at this time that there is a preferred or dominant model for hardware description. From generative approaches such as Chisel [4] to RTL description in traditional HDLs like SystemVerilog to data flow oriented languages such as Bluespec [40] and CAL [20] to high-level synthesis tools such as the pipeline-oriented Sehwa [44], there are many different popular tools and paradigms for describing digital hardware. Given the continued diversity of hardware design tasks and approaches after more than half a century of hardware description language development, it does not seem appropriate to ask what the single ideal hardware description language paradigm should be. A language for expressing digital hardware designs should enable multiple paradigms, from gate-level design through high-level

synthesis, in a manner that preserves interoperability between blocks designed using different approaches. This thesis proposes and evaluates the following hypothesis: *By adopting a general-purpose language with strong support for construction of domain specific languages, such as C++, as a hardware description language and building a layered set of abstractions around a core of simple primitives, we can produce interoperable designs using a diverse set of paradigms, from gate-level description to high-level synthesis.*

The major research contributions presented in this document are:

- The evaluation of a multi-paradigm hardware description, CHDL, from the point of view of designer productivity.
- The evaluation of performance and quality-of-results impact of novel optimizations implemented using CHDL's unique netlist introspection feature.
- The evaluation of a data parallel core designed using this hardware description system.

The major engineering contributions also presented are:

- An open source C++ domain specific language for hardware description, CHDL.
- A novel family of data parallel instruction set architectures and cores implementing these instruction set architectures, HARP and Harmonica.
- Layers implementing higher-level hardware description paradigms on top of CHDL.

The remainder of this document starts with Chapter 2, a chapter describing related work in hardware description languages and high level synthesis, which is followed by a layer-by-layer building-up of the abstractions developed and results gathered for this dissertation. Chapter 3 describes CHDL, a C++ hardware description language including a core library and a template library of common hardware primitives. Included in this is a discussion of the novel netlist reflection technique and a range of techniques developed using this to perform optimization of both elaboration time and quality of results. CHDL and its associated template library build up from a core of gate-level primitives to a register transfer level DSL. Chapter 4 describes the use of RTL CHDL to produce Harmonica; a

family of data parallel core designs evaluated in the role of near-memory accelerators. Much of the evaluation of Harmonica designs was performed using CHDL-provided simulation infrastructure, including a power emulation library leveraging CHDL's netlist introspection technique and technology mapping feature. In Chapter 5 and Chapter 6, libraries supporting higher-level abstractions as further layers on top of CHDL are introduced. These demonstrate the practicality of implementing such higher-level paradigms in a CHDL-style constructive hardware description environment.

CHAPTER II

RELATED WORK

The work presented in this dissertation is primarily concerned with broadening the range of hardware description paradigms that can be used, interoperably, from within a single hardware description language environment. It is necessary to place this work within the context of contemporary and historical hardware description language research. A broad range of hardware description languages has been developed over the past half-century, and a steady stream of research has been published and products have been marketed aiming to make the textual description of hardware more expressive, allow more effective design reuse, enable higher-performance simulation of complex hardware for verification, and integrate reconfigurable hardware with general-purpose computing systems. The present work draws from many sources as it presents a method for unification of multiple paradigms, and is not the first work to do so in one capacity or another.

Research specifically focused on using the layering of abstractions to build systems for RTL and higher-level hardware description systems on top of structural generators has been found among the extant literature [24], but the taxonomy in Table 1 illustrates the greater relative extent of the multi-paradigm support implemented in CHDL. By including a high level pipeline description language among the supported paradigms, this work demonstrates that the full range of abstractions available in hardware description systems can be produced within a generative HDL. In this table, HDL projects discussed in this chapter are taxonomized by the number of hardware description paradigms supported, their extensibility, and the extent of the barriers between these paradigms.

Products are available and research being performed at all corners of this taxonomy. The number of paradigms can range from one as found in high-level synthesis tools exposing a fixed execution model to the programmer to many as found in mainstream HDLs supporting structural, behavioral, and RTL description. Extensibility is a property of generative HDLs

Table 1: A list of hardware description systems discussed academically taxonomized by levels of abstraction supported and extensibility.

System	Language	Structural	RTL	Behavioral	High-Level
Fixed					
PMS/ISP [5]	—	×	×		
Sehwa [44]	Lisp				×
Bambu [46]	C++				×
Gaut [16]	C++				×
Trident [56]	C++				×
LegUp [11]	C++				×
Handel-C [3]	C			×	×
SystemC [42]	C++	×	×	×	×
Extensible					
PamDC [8]	C++	×			
CHDL(2001) [33]	C++	×	×		
Java Gen. [12]	Java	×			
JHDL [7]	Java		×	×	
Chisel [4]	Scala	×	×		
Extended Chisel [24]	Scala	×	×	×	
Cλash [34]	Haskell	×	×		
Bluespec [40]	SystemVerilog	×	×	×	
MyHDL [17] [27]	Python	×	×	×	
PyMTL [35]	Python	×	×	×	
CHDL	C++	×	×	×	×

such as Chisel [4] built in general-purpose programming languages and not of traditional HDLs without support for defining new domain-specific languages.

2.1 Fixed-Paradigm Hardware Description Systems

A subset of hardware description language work has produced fixed, non-extensible languages, in which the set of abstractions available to designers may not be modified by the inclusion of additional libraries or by the designers themselves. A few of these, the majority of which are high-level synthesis tools intended to translate algorithms expressed in software-oriented programming languages into hardware realizations of the same algorithm, only allow hardware to be expressed within a single paradigm. Still more enable some combination of RTL, behavioral, and generative hardware description in addition to any novel, more-abstract hardware description.

2.1.1 Traditional Hardware Description Languages

The genesis of hardware description languages lies in languages providing multiple fixed paradigms. In 1970, Gordon Bell and Alan Newell published the PMS and related ISP systems of hardware description specifically intended to model computing systems [5]. These were developed for and used in their book, *Computer Architecture: Readings and Examples* [6], an early computer architecture text, as a coherent way to formally describe block diagrams of computer system components. PMS was a structural HDL for computer architectures and its cognate RTL language was ISP. At the time these were developed they were perhaps considered to be extensible systems, in the sense that PMS was entirely built from ISP primitives allowing blocks of RTL statements to be combined into units which could be connected together at a higher level, using a graph notation. By allowing modules of RTL statements to be connected together, higher levels of abstraction could be reached simply by constructing more abstract blocks from lower level blocks, and in the case of ISP/PMS there was even a way to compose data types from lower level data types, so all of the examples of e.g. floating point numbers constructed from two integer types and a bit type could be built in this very early system as well. What separates these systems from what is considered extensible in the present document is the level of abstraction that could

be reached in this manner. Raising the level of abstraction of the described system is not the same as raising the level of abstraction provided by the hardware description language itself, and despite the complexity of the systems that could be created from simple primitives through hierarchical construction using a system like ISP/PMS, the HDL paradigm remains structural and certain concepts like complex pipeline controllers remain difficult to express.

The influence of ISP/PMS can be seen in the modern HDLs in most widespread commercial and academic use, VHDL and Verilog. Structured signals and a combination of RTL and structural description underlie both of these language families. Both VHDL and Verilog add to this languages for describing generators, behavioral modules, and a templating or parameter system. The generator system allows procedural code to instantiate submodules and RTL statements, and may use module parameters to modify its behavior. This enables a wide variety of designs to be realized that would require significant repetition otherwise, although criticism of the limitations of the systems for producing generic parameterized modules in both of these languages, when compared to modern templating systems and generative HDLs, is common.

Behavioral descriptions, which were the target of early commercial high-level synthesis tools as described in Section 2.1.3, are now almost entirely relegated to simulation and verification work. These allow the behavior of blocks to be described using a procedural programming language instead of register transfer level. The principal distinction between RTL and behavioral descriptions is that assignments within behavioral descriptions are ordered and may be read and modified multiple times by a procedural program using a full set of control flow constructs within a single time step of the simulator. Timing for behavioral descriptions is also arbitrary and need not conform to the edges of an incoming clock signal.

2.1.2 Data Flow Systems

The actor model, as implemented in the CAL Actor Language [20] is a formalization of parallel algorithms as a networks of communicating units, called actors. This model may be

adopted in the implementation of both hardware and software, and is an eminent dataflow-oriented paradigm. In this model, actors are units possessing state and consuming and producing tokens, which are sent through channels; FIFO links passing tokens which form the edges of the graph of actors. Actors contain actions, which fire if their guard conditions are met, which may be due to the availability of input tokens, the current state of the actor, or both. Actions may both consume inputs and produce outputs, and priorities for actions are explicitly defined. This model, and the CAL Actor Language in particular, has been used to synthesize FPGA configurations, most popularly in the domain of digital signal processing.

A hardware-oriented approach similar to the actor model is the guarded atomic actions (GAA) paradigm as implemented in Bluespec System Verilog [40]. In guarded atomic actions, systems are formed as a hierarchy of modules, where each module contains a set of state values, rules describing how that state evolves, and methods which can be externally-invoked rules leading to state change or reading of state. All of these are protected by guard predicates and rules modifying the same state may not be activated in the same cycle. This is all encapsulated in a superset of System Verilog, Bluespec System Verilog, enabling backward compatibility with existing Verilog RTL and behavioral code. There is not, however, a way to combine this with further extensions to Verilog, and any other system extending System Verilog to add other features would not be aware of Bluespec, limiting the number of paradigms available within the same mutually-compatible system.

AutoPipe and the associated X programming language [21] produces optimized pipelined implementations given a data flow graph. The descriptions in this case are not at the level of hardware, but rather at the level of the flow of data between large units, each implemented either in C++ or in Verilog, with the aim of automatically producing an optimized implementation. The primary disadvantage to this approach when contrasted with the definition of a large design using a system like CHDL is that the interfaces and basic operations on the data that appears at the interfaces have to be defined three times, once for the Verilog representation, once for the C++ representation, and again for the top level data flow language. The advantage is that the scale of systems that could be described

using AutoPipe is potentially much larger than the scale of systems that could be modeled using CHDL, spanning multiple racks of equipment, enabling high-performance simulation of systems that would be impractical to simulate at the gate level.

For the purposes of the taxonomy in Table 1, dataflow paradigms have been lumped in with behavioral models. This is because dataflow paradigms describe a functional behavior in terms of a set of primitives without resorting to register transfer level specifications, and in this sense they are behavioral, but they do not describe an algorithm as a series of steps so they are not precisely a type of high-level synthesis.

2.1.3 High-level Synthesis

There is a large set of contemporary high-level synthesis products, including the GCC-based Bambu [46] and the LLVM/Clang-based Gaut [16], Trident [56], and LegUp [11], as well as a few commercial offerings. This work spans many different problem domains but is all related by the use of a software-oriented compiler as a font-end, producing an intermediate representation that is then converted into a set of hardware primitives, usually represented as some combination of a data path and controller. This approach to design is quite attractive as it allows the automatic generation of hardware architecture to implement a given algorithm within a set of constraints. If the design constraints change during the design process, the high-level synthesis tool can simply be re-run with the new constraints and design work does not have to be replicated.

A compromise approach was taken in Handel-C [3], which provided a hardware-oriented language based on C with explicit concurrency through explicitly parallel loop blocks as well as an interface called the channel. Handel-C expressly forbade writable shared variables between parallel blocks, only allowing inter-block communication through channels providing blocking *get* and *put* operators. This categorically prevented the problem of write conflicts by providing semantics to resolve the case in which multiple get or put operations to the same channel occurred during the same cycle. Despite the use of high-level constructions, it is still possible in many Handel-C programs to determine what the state will be during a specific cycle, a difficult prospect in many high-level synthesis environments.

The SystemC hardware description language takes a similar approach to Handel-C by providing a hardware-oriented superset of an existing general-purpose programming language, but unlike Handel-C provides its simulation environment entirely within the context of a C++ library. SystemC provides support for behavioral, structural, RTL, and transaction-level modeling of hardware using a combination of C++ classes and macros. Commercial high-level synthesis tools are capable of translating SystemC input, including modules containing behavioral components, into synthesizable RTL. SystemC modules can be customized using C++ template metaprogramming and generators, allowing such novelties as a CHDL compatibility layer enabling CHDL designs to use the SystemC simulator, but its purpose is to act as a substitute for Verilog or VHDL in workflows incorporating transaction level modeling.

In cases like video and audio coding where algorithms are quite complex, LegUp provides an automated approach to co-design [11]. This automatically partitions a design between a MIPS core and a set of hardware accelerators at the function granularity. This further automates the process of finding a design to fulfill a set of constraints, as the task of partitioning between hardware and software can itself be automated. The architectures, however, are limited to a very specific format containing a single instruction set processor and a set of accelerators, and in all of these high-level synthesis tools the availability of efficient RTL for particular hardware tasks creates an additional design challenge; there is no automated way to integrate the high-level algorithmic description with existing HDL code. The HLS input is a monolith, only able to communicate with other pieces of the design through narrowly-defined communications semantics.

The FROST framework presented in [18] provides a unified front-end for targeting HLS tools from domain specific languages. This provides a multi-language interface to FPGA-oriented high-level synthesis tools but low-level design is prohibited by the hardware-obscuring nature of the high-level synthesis layer itself, which is provided by commercial HLS tools provided by Xilinx supporting the C, C++, and OpenCL programming languages. These tools, while performing the same function as HLS tools targeting silicon, are designed

to position FPGAs as potential competitors for GPGPUs and other general-purpose accelerators. By supporting OpenCL and its successor SyCL, they allow the explicit thread-level parallelism expressed in applications for inherently multithreaded architectures to be exploited by the resulting hardware implementations. While OpenCL applications provide explicit operations to copy state between accelerators and main memory, SyCL provides a task graph API, effectively providing a single-language interface to two levels of abstraction, much like the early ISP/PMS system, but with compute kernels and data flow graphs instead of RTL implementations of state machines and graphs defining systems composed of them. Doumoukalis et al., in [19], demonstrated the porting of an OpenCL-based FPGA application to the open source TriSyCL implementation of SyCL, preserving much of the performance while automating the task of moving data from host to accelerator.

In any work including a pipeline-oriented hardware description system it would be remiss to fail to mention Sehwa [44], an early high-level synthesis program unique among early HLS work in specifically targeting pipelined designs. From an input data flow graph, Sehwa automatically generates a pipelined implementation meeting a set of specified timing and cost constraints.

The input to Sehwa is a data flow graph that, while acyclic, could contain conditional branches and is wrapped in an implicit outer loop so it could perform some iterative tasks. The pipeline itself is expected to execute multiple independent tasks in parallel. The input format is simple, but Sehwa is a primitive high-level synthesis tool, in the sense that the input is in the form of a higher-level language and expressed an algorithm rather than a specific architecture and resource allocation and timing is handled entirely by the synthesis algorithm, which has specific quality-of-results targets to meet. This is quite different from the approach taken in Cheetah, which is also a tool designed to enable the development of pipelined hardware, but which leaves the specific latencies of hardware implementations of combinational logic blocks to downstream synthesis and retiming optimizations, and which makes selection of pipelined architecture an explicit activity, while allowing and even encouraging looping control flow, even nested loops.

These loops supplant what would be considered to a tool like Sehwa pipeline stages with

multi-cycle latencies or sub-cycle issue rates. This trades machine-generated schedules and manually designed functional units to machine optimized functional units and manually designed pipeline schedules with the help of retiming optimizations. Programs like Sehwa, and broadly, all high-level synthesis tools can be thought of as a series of transformations on an input netlist or program or behavioral description. Cheetah input and CHDL GAA input on the contrary include invocations and instantiations of CHDL functional units, prompting the generation of in-memory low-level representations and are therefore compatible with all CHDL hardware blocks and data types. The GAA and Cheetah `generate()` functions then wire up registers and generate controller logic to implement the appropriate paradigm.

2.2 Extensible Hardware Description Systems

The work discussed so far concerns systems either using custom languages lacking type systems with operator overloading and other features allowing the creation of domain specific languages or are high-level synthesis systems lacking the ability to describe hardware at a lower level. In this section we discuss work concerning systems that use general-purpose languages and allow hardware description in a generative style. These systems are good candidates for extensibility in the same sense as the work presented in this dissertation, although this has not been explored in these HDLs to the extent that it has been explored in CHDL.

2.2.1 Simulation and Verification Oriented Systems

PyMTL [35] is a modeling environment that integrates three levels, dubbed the functional level, cycle level, and register transfer level of architecture modeling. This may all be done from within the same environment, and with the support of a high-performance JIT that accelerates the higher-level models. This enables trade-offs between performance and fidelity to be made; an RTL accelerator model could be validated with behaviorally-modeled memory system components all within the same system.

This can be contrasted with CHDL by the fact that only the register transfer level in PyMTL is synthesizable. The levels of modeling in PyMTL are not levels of abstraction for *describing* hardware, although high level synthesis could presumably be grafted in to

provide this support; they are, rather, levels of abstraction for *modeling* hardware. The focus in the PyMTL system is on developing systems for modeling hardware at multiple levels of fidelity and performance simultaneously. The primary focus of the work presented in this dissertation, however, is the use of a unified framework for describing hardware, in which all of the described hardware is synthesized to a gate-level netlist that can be mapped to hardware.

2.2.2 Generative Hardware Description Languages

As part of the PAM project, an early experimental reconfigurable computing platform, the PamDC hardware description language was developed [8]. This was a C++-based system that used operator overloading and template metaprogramming to enable the structural description of hardware using C++. The level of description allowed was entirely structural, providing a register initialized to zero as a primitive and allowing more complex hardware to be constructed from this and combinational gates. A similar system, CHDL, for C++-based Hardware Description Language, which bears a similar name to the CHDL Hardware Design Library discussed in this dissertation, is a similar project to PAM described in a brief 2-page 2001 paper [33]. This system allowed for both structural and state-based, presumably RTL-like, description in C++. While it does not offer higher levels of description than state-based, and further extension is not mentioned in the paper, the fact that it implements a state-based language on top of a structural description points toward the kinds of extensions implemented in the present work.

In 1998, two systems for writing generators using the Java programming language were reported in the literature. Chu et al. described a simple system mapping Java objects to hardware modules and running generators in constructors [12]. This work relies on Java's metadata interface to allow discovery of the design hierarchy and signals, not requiring an equivalent of CHDL's `TAP()`, or `HIERARCHY_ENTER()`/`HIERARCHY_EXIT()`, described in Section 3.4.5.

The JHDL [7] system was also described at this time, but took a very different approach. Instead of building up designs from a low level, combinational or synchronous circuits were

described at the register transfer level, as executable Java code, within specially designated Java functions. Since these functions are programmer-visible as Java bytecode, they could be directly netlisted from the Java program itself. This allowed very tight integration with partially-reconfigurable hardware and allowed JHDL to be used as a platform for reconfigurable computing. In addition to RTL, classes could also simply be structurally defined as combinations of other classes. Since this could conceivably include arrays of other classes strung together by elaborate constructors, JHDL structural classes had the capacity to be quite complex generators, although this was never used to extend JHDL beyond RTL.

2.3 Extensible Generative Systems

The MyHDL system is a system for performing RTL design and building behavioral simulation models using the general-purpose Python programming language [17]. The original version of MyHDL is built around Python's support for continuations using the `yield` keyword. Each of these functions `yield`s a sensitivity list and its execution is allowed to continue by the simulator when one of the conditions in this sensitivity list is met. This allows for the simulation of rather complex behavioral models, as very complex procedural programs can execute cycle-to-cycle, one `yield` to the next. These kinds of behavioral models are not synthesizable, although a register transfer level and structural subset is provided that is.

The feature that makes the MyHDL system extensible is the fact that the simulated object hierarchy is constructed entirely at run time, and this can be performed entirely under the direction of a Python program, allowing the construction of domain specific languages producing synthesizable hardware as output. It has primarily been discussed in hobbyist circles, although MyHDL has received some academic attention as of 2015 [27]. This work described the addition of essential functionality to MyHDL, allowing structured data types with signals, including other structured data types, as member variables to be used in the generation of MyHDL logic.

The Cλash HDL exploits the concepts of functional programming to enable the elegant

and compact description of synchronous hardware [34]. The Cλash system is specifically designed for enabling the description of Mealy machines with single clocks, performing simulation by repeatedly executing a transformation function on a series of inputs and a machine state, producing a final machine state and a series of outputs. Also included is a compiler that processes the Core intermediate format provided by the Glasgow Haskell Compiler to transform these functions to synthesizable VHDL. This operation could be classified as high-level synthesis in a very broad sense, but since the transformation is performed entirely on combinational logic and does not enable procedural execution or, equivalently, arbitrary recursion depth, it is better to think of as a system enabling generators, as arbitrary recursion is definitely supported by any higher-order functions involved in the production of this state transition function.

The examples given for the Cλash language are quite simple and the logic generated is very simple, but this is not a fundamental limitation of the system. Because higher-order functions, essentially generators, are supported a wide variety of paradigms could be implemented.

In 2012, the first description of the Chisel system was published [4]. This work emphasized the fact that Chisel is a system for describing generators, using features of the Scala language such as operator overloading to provide a natural interface for describing hardware using generators. In this work, some possibility of extensibility beyond generators suggested. Chisel introduces a keyword `when` that enables the construction of RTL-like statements using Scala's support for passing blocks of code as function arguments. This has not been extended to the possibility of nested `when` statements or to higher levels of abstraction, but there is no inherent limitation in Chisel itself that prevents the kinds of constructs that have been implemented as part of CHDL from being ported, and some work to this effect has been done, including an implementation of Bluespec-like guarded atomic actions.

2.3.1 Extending Generative HDLs

Generative HDLs are the best candidates for receiving ports of CHDL's multi-paradigm features. A generative HDL could support description at multiple levels of abstraction in the same way as CHDL as long as it was implemented in a general-purpose programming language with good domain-specific language support, generation was performed by execution of this code, and references to signals could be stored and assigned later. These three traits enable the creation of generators that implement register-transfer-level and high-level paradigms on top of generative HDLs and, while specific and excluding several existing tools, are not unique to CHDL.

Work by David Greaves at the University of Cambridge has demonstrated a variety of higher-level paradigms being implemented on top of the Chisel system [24]. This includes an implementation of register-transfer-level semantics including nested `if` statements, the concurrency model from Handel-C, an implementation of Bluespec-like Guarded Atomic Actions, and an implementation of transaction-level modeling, the high-level inter-module communication model supported by SystemC. This work introduced the possibility of a rich set of hardware description paradigms implemented as domain specific languages on top of a generative HDL. Among its contributions is a system for interoperability between the method interfaces of Bluespec and the transactional interface of transaction-level modeling, pairing and outputs method invocations into single transactions to enable full interoperability among the paradigms presented.

The present work provides a similar RTL and GAA layer on top of a generative hardware description language, but also extends this to include support for Cheetah, a pipeline-oriented HDL providing automatic insertion of pipeline registers. This effectively raises the level of abstraction to that of high-level synthesis, cementing the assertion that any hardware description paradigm may be implemented as a library or DSL on top of a generative HDL.

2.4 Motivation for Present Work

Literature concerning hardware description languages has principally focused on designer productivity, quality-of-results, and simulation performance in one sense or another. All of

the systems described here had some type of productivity-related goal in mind, whether it was simply enabling direct comparison of computer architectures like PMS/ISP, enabling design reuse like all of the generative languages, or enabling the development of hardware accelerators by programmers instead of hardware designers like much of the high-level synthesis work. The present work is a continuation of this thread that focuses on improving designer productivity in two ways: first raising the level of abstraction by enabling a collection of DSLs suited to specific design tasks and second by enabling design reuse by lowering the barrier to integrating designs created using different paradigms into a single design.

The decision to implement CHDL using C++ instead of using Python, Scala, Haskell, Scheme, of any of the other appropriate languages comes down to practical considerations about its ubiquity and the performance requirements of software in the hardware design toolflow. The performance of design elaboration within hardware description languages cannot be entirely overlooked, but it is primarily simulation and synthesis performance that define the performance of an HDL-based design flow. The decision to use C++ for HDL work enables a single language to be used for both hardware description itself, where performance is not as critical, and related CAD algorithms, which are highly performance-sensitive, without the duplication of effort required when crossing language boundaries.

The principal way C++ enables the development of performant software is through its templating system, allowing reusable structures to be described that compile to high-performance, specialized machine code, which is an important enabler for CAD algorithms. The use of template metaprogramming also enables type safety by allowing algorithmic type checking and expansion, which is a feature useful for hardware description.

The wide install base of C++ compilers is also a relevant consideration. All widespread desktop, laptop, SBC and microcontroller platforms support development using the C++ programming language and every major platform used for development can run a C++ compiler, typically available as a free download. This has led to the number of developers familiar with C++ being quite large and knowledge of the language being widespread, making C++ arguably the most widely known language with the features required for implementing an extensible HDL.

CHAPTER III

DESIGN OF CHDL

3.1 Introduction

CHDL is a C++ library for hardware description. CHDL programs are first run through a C++ compiler, producing executable *generators* that in turn produce an in-memory netlist describing hardware blocks using CHDL API calls. Because of the expressiveness of C++, the code describing these generators may take the form of domain-specific languages implementing higher-level paradigms, but ultimately the modeled hardware is converted into an in-memory gate-level netlist that may be written out as synthesizable Verilog or simulated.

The key contribution of CHDL is that a diverse collection of hardware description paradigms may be employed within a single framework with a small set of core primitives. Designs implemented in CHDL may range from the gate level, to the register transfer level, to designs using the CHDL implementation of guarded atomic actions, to designs using Cheetah, a pipeline-based HDL also implemented as a library on top of CHDL. Each of these pieces is designed in such a way that they it does not hide the layer of abstraction beneath it, preserving interoperability between pieces designed using different paradigms.

To use a concrete example, say we create a templated N-bit saturating integer data type called `satint<N>` using CHDL, along with a set of operator overloads to support basic arithmetic using this saturating integer data type. Operations on `satint` types all generate combinational logic, so they are just as easily expressed in C as they are in synthesizable Verilog or VHDL or another RTL-oriented HDL. Say we have a pipelined accelerator core that is implemented using Cheetah. Because Cheetah extends the ordinary CHDL types to represent signals, we can still use `satint` without any modification. This is reasonable behavior; combinational logic in an automatically-generated pipelined datapath still performs the same function as the same logic in a datapath expressed as RTL. In this

chapter, the implementation of CHDL and its included template library is described, from the basic primitives allowing the description and simulation of logic to the first level of abstraction beyond that, which enables the register transfer level description of hardware designs, and a technique enabled by CHDL called *netlist introspection*, in which programs generating hardware use the APIs exposed by CHDL to extend CHDL itself, is highlighted.

3.2 Synchronous Design

CHDL intrinsically embraces the concept of *synchronous* design; a CHDL design is restricted to only D flip-flops and combinational logic. Stated equivalently, CHDL designs are directed graphs of gates in which all cycles necessarily pass through at least one D flip-flop. SRAM, both synchronous and asynchronous, is available as well, but the path of a read port belonging to an asynchronous SRAM is considered to be combinational logic. The behavior of simultaneously, within the same clock cycle, reading and writing the same memory location is undefined. Bypass paths for asynchronous SRAM must be explicit and these are counted toward the no-cycle rule. As a result of this, many common structures such as D latches, SR latches, and dynamic logic cannot be explicitly modeled. For the kinds of computation-oriented logic CHDL is designed for, these structures are considered superfluous. This refusal to allow free-form asynchronous sequential logic design shapes the focus of CHDL away from these designs. Despite this it is CHDL's stated goal to be universal. Adoption of a synchronous-only approach is therefore the equivalent of rejecting asynchronous clocked designs wholesale.

In the context of computation, synchronous designs are more testable, more straightforward to formally verify and synthesize, and more immune to subtle design errors. Latches require fewer transistors than flip-flops per bit, especially scan flip-flops, and this may lead to area or power reduction. This is true, but in storage and register file cases, SRAM saves even more area and in general pipelined computation cases, cycle-stealing latch based designs may be converted to D flip-flop designs with no loss of design content and little increase in area.

Perhaps the most valuable feature enabled by synchronous logic is scan-based test,

including built-in self test, in which all bits of state in a design can be arranged into a set of shift registers, allowing the full state to be clocked in or out. One of the reasons for embracing scan-based testing is the tractability of automatically generating test patterns. ATPG-produced externally-transferred test patterns and BIST strategies, perhaps including test point insertion, are essential for reliability in deep submicron technologies nodes.

Scan chains have further use during debug, characterization, and failure analysis: providing access to and allowing modification of a design's state. Much like in-circuit debugging of microcontrollers, this enables a wide range of techniques that would not otherwise be practical. This can be illustrated with a simple example: a BCD counter with a divide-by-10 input is damaged and this has led to a field return. You wish to validate some down-stream circuitry that is only actuated when the counter reaches a specific value. Due to the failure, this value is never reached. By loading the value into the scan chain the downstream or external feature may be examined despite the fact that the requisite conditions are otherwise unreachable.

Beyond such rare and special circumstances, the scan chain enables the use of silicon as an extraordinarily fast simulator and FPGAs as very fast simulators. Through the scan chain, states simulating a wide range of conditions may be reached and a logic design thoroughly characterized in a way that may not be practical even with the fastest simulator. The answering of questions about what happens in trillions of cycles in designs containing megabits of state becomes tractable.

Given the complexity of digital designs, the benefits of the scan chain alone are enough to recommend synchronous design. There is also, however, the matter of correctness. It is quite straightforward to formally express and evaluate synchronous designs as Mealy state machines, and quite easier to ensure that they retain their functional correctness through a wide range of circuit transformations that may, e.g. impact timing. The same cannot be said of asynchronous designs containing a large number of derived clocks, a problem that has led to the inability to simulate gate-level silicon asynchronous designs described in Verilog on FPGAs.

Much of asynchronous design is already difficult to express in a language like Verilog,

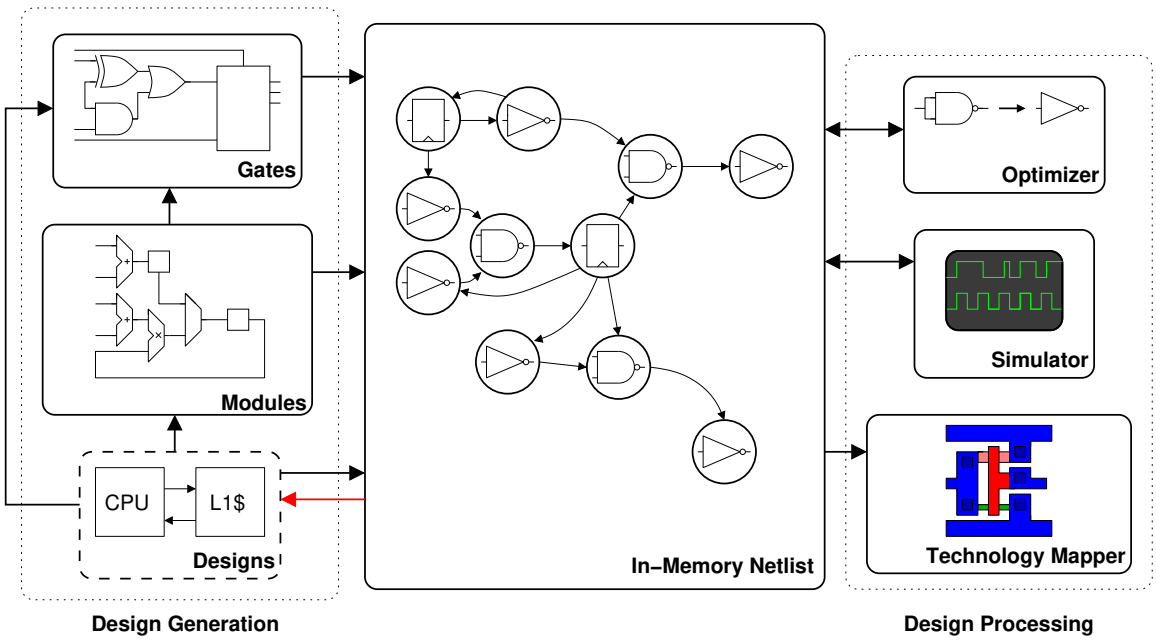


Figure 3: Overview of dataflow in CHDL.

with the exception of simple latch structures. CHDL simply takes this to its logical conclusion by providing no provision for any novel latching structures to be described at the gate level, preventing timing errors by forbidding derived clocks entirely. This dependence on synchronous design is a fundamental characteristic of CHDL.

This is not to say that it is impossible to model asynchronous designs at all within CHDL. While the intent of CHDL is that it be used to model processor designs and other complex synchronous logic, there are other types of circuits that may benefit from description and simulation within a C++ environment. Gate-level models have been constructed using CHDL, quantizing propagation delay using clock cycles to represent fixed time increments. Further possibilities for a future approach incorporating hooks for an entire in-memory suite of CAD tools, including timing-accurate gate-level modeling, is discussed in Section 7.2.

3.3 Structure of CHDL

Programs written using CHDL express generators for hardware, which construct an in-memory netlist. This may then be optimized, simulated, or technology mapped to standard cell libraries using function calls available in CHDL. Figure 3 illustrates the flow of data in CHDL. All of the "design processing" pieces as well as all of the functions used to manipulate

the in-memory netlist are provided by CHDL or its template library, CHDL-STL, though APIs are provided to allow external code to provide additional functionality as well.

3.3.1 Template Metaprogramming

The design of CHDL is meant to take advantage of C++ template metaprogramming to enable compile-time signal type checking. It is feasible to perform run-time checking of signal types and to create dynamically-allocated C++ objects representing signals, but the use of template metaprogramming to construct operations in CHDL enables the C++ type system to be used to ensure that operations are semantically valid at compile time. A simple example that illustrates the use of template metaprogramming in CHDL is a library function for performing a population count. This makes heavy use of functions provided in the CHDL core library, described in detail in the next section.

If we were to use the dynamically-allocated C++ STL types to represent our vectors of bits, we may create a single recursive population count function that adds the population counts of two half-vectors, with special base cases for vectors of length zero and one. The length of the result of this operation cannot be known at compile time. Using template metaprogramming, the population count operation may be implemented as:

```
bvec<0> PopCount(bvec<0> x) { return x; }
bvec<1> PopCount(bvec<1> x) { return x; }

template<unsigned N> bvec<CLOG2(N+1)> PopCount(bvec<N> x) {
  return Zext<CLOG2(N+1)>(PopCount(x[range<0,N/2-1>()])) +
         Zext<CLOG2(N+1)>(PopCount(x[range<N/2,N-1>()]));
}
```

Using the template metaprogramming approach, all of the quantities representing dimensions can be compile-time constant.

3.4 CHDL Core Library

3.4.1 Basic Hardware Manipulation

- `CLOG2()`

The operation $\lceil \log_2 N \rceil$ is so ubiquitous throughout hardware design that, in CHDL, it is given its own function; one of very few non-hardware-generating utility functions in CHDL. Using the C++ `constexpr` keyword, this function is evaluated at compile time, meaning that it may be used within the declarations of constants and template arguments. This is, in fact, its primary use, as the general operator for the number of bits needed to select one of N possibilities. It shows up in the declarations of many functions throughout the CHDL core library, template library, and hardware designs using CHDL.

- `reset();`

A core decision around which many of the design choices in CHDL were made was the choice to use a stateful API. By having, at any time during program execution, an implicitly addressed global state, variable declarations and function invocations are shrunk in size, though the number of situations in which CHDL designs may be elaborated is also diminished. The choice to use global variables hidden from the user to contain this state specifically sacrifices thread safety in the elaboration stage, in order to make variable declarations cleaner since they do not have to be pointed to a CHDL context.

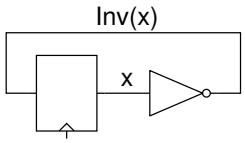
The CHDL design accumulates in memory as CHDL calls are made and simulation and optimization are performed on the in-memory netlist. In many cases, a single program may wish to generate more than one CHDL hardware design. In these cases, the `reset()` function must be invoked. The `reset()` function is connected to each piece of CHDL that stores state. Each component uses the `REGISTER_RESET()` macro to instantiate the `reset_func<T>` template class, whose constructor registers the call in a global list. Many of these pieces of global state are pointers initialized to `NULL` which can be reset by deleting the heap-allocated object they point to and re-initializing to `NULL`. A convenience function, `delete_on_reset()`, is provided for configuring this common case.

This may seem like internal CHDL plumbing, but in addition to the `reset()` function, all of these hooks are provided as part of the user-facing CHDL API in case there are also user-allocated objects, e.g. for high-level simulation, that must be cleared along with the generated design.

- `node` and `nodeimpl`

Another fundamental design decision affecting CHDL is the decision to split the fundamental object, `node`, representing a net or logic signal in a circuit, into two pieces, `node` the designer-facing handle representing but not actually containing the circuit node and `nodeimpl`, the behind-the-scenes structure that stores the logical netlist, including relationships between nodes. This distinction allows a single `node` type to represent all of the available logic gates in CHDL without further extension and enables assignment operators to behave reasonably even in cases where multiple `node` objects ultimately wind up representing the same circuit node, and in cases where the consumer of a value occurs after its producer. Consider the following simple example:

```
node x;
x = Reg(Inv(x));
```



In this case and many others like it a `node` must be evaluated before its value has been assigned. The use of a `nodeimpl` to represent the node allows us to keep a list of all `node` objects pointing to the same `nodeimpl` and, when the assignment operator is called on any `node` object pointing to a given `nodeimpl` to update all of the `nodes` pointing to that `nodeimpl` to point to the new `nodeimpl`. In the following example:

```
node a, b;
a = b;
```

`a` and `b` initially refer to two different entities, but at the end they both point to the entity originally referred to by `b`. The `nodeimpl` pointed to by `a` in the beginning becomes inaccessible, to be reclaimed by the dead node elimination optimization when the time comes.

A subtle consequence of the behavior of the `node` object is the fact that `node`'s assignment operator is a `const` member function of `node`. It does not, after all, modify the `node` object itself through the `this` pointer. It modifies all of the `node` objects pointing to a given `nodeimpl` through the `nodeimpl`'s array of non-`const` pointers to `node` objects. This is similar to the counter-intuitive fact that `delete` may be called on a `const` pointer in C++ because the memory manager is not invoking the destructor through the `this` pointer but through its own internal data structures. Assignment in CHDL is transitive and does not depend on program order, thereby allowing information to flow up the page as in the following example:

```
node a, b, c;
a = b;
b = c;
c = Lit(1);
```

which causes all three `node` objects to point to the same literal logic 1 value.

In addition to the C++ classes `node` and `nodeimpl`, a circuit node in CHDL may be referred to by an integer value, using the type `nodeid_t`. A node's ID is simply the index into the global vector of `nodeimpl` pointers, `nodes`, at which a node's implementation is stored. This type is used when the assignment behavior of `node` is not desired, but is invalidated by optimizations.

A member function of `node`, `check()`, is provided to ensure that a given `node` points to a valid node ID. Failure of this check results in a controlled program crash instead of access into invalid memory. A huge hurdle to the use of CHDL that has been mitigated in many ways such as this is the basic problem of designing hardware with a type-unsafe systems programming language like C++. Unless care is taken through insertion of checks and features that make it difficult to express invalid designs in syntactically valid ways, the program may simply violate memory access protections and crash, or leak memory, or suffer from any of a number of other software errors before we have even begun to debug the hardware itself.

A special template class that is not meant to be instantiated, `sz<T>` is specialized for every CHDL signal type. It has a single constant integer member, `value`. The constant `sz<node>::value` is one. Other types in CHDL that aggregate multiple types together, such as the `vec` and `ag` set the value of `sz<T>::value` to be the total size of all of their constituents so that this can always be used to obtain the number of physical wires needed to carry a signal or bits of storage needed to store it.

- `printable`

One of the stated goals of CHDL is the ability to emit synthesizable netlists and act as a front-end to commercial or vendor-specific tools. The ability to dump the in-memory circuit representation to a variety of formats is therefore important. Previously there was an ad-hoc mechanism in place for doing this with functions called `print()` and `print_vl()` being provided by all `nodeimpl` sub-types to emit native netlists and synthesizable Verilog respectively. This has been deprecated by a system in which each object that may be dumped in any format inherits from a class called `printable` and overloads a set of functions:

- `register_print_phase(language, phase)`
- `is_initial(language, phase)`
- `predecessors(language, phase, set<printable*> &)`
- `print(ostream &, language, phase)`
- `print_design(ostream &, lang)`

This leads to the presence of another global list, the list of printable objects. These printable objects are ordered by the `is_initial()` and `predecessors()` functions in a directed acyclic graph. This graph may be different for each language supported and for each phase of printing. At the moment, the supported languages are Verilog and CHDL's own netlist format, but more are expected in the future including some abstract formats such as Graphviz `.dot`, currently supported through the CHDL visualization API call `print_dot()`.

- `cdomain`, `tickable`

Simulation of synchronous designs requires both a way to evaluate combinational logic and the concept of a clock edge, the temporal basis for synchronous logic. In CHDL, "derived" clocks are not considered for simulation, only clock domains constructed around periodic clocks are allowed. When netlists are exported for further processing each clock domain becomes a separate clock input. During generation of designs, a "current" clock domain is maintained along with a stack of previous clock domains allowing individual design modules to use their own clocks. Functions `push_clock_domain()` and `pop_clock_domain()` are used on entry and exit of portions of the design using clock domains other than the default. This has enabled the description, e.g., of asynchronous FIFOs surrounding independently-clocked sub-blocks. Each clock domain contains a subset of the clocked objects, `tickable`s. Within CHDL each tickable has four separate, independent function calls during each step of the simulation clock: `pre_tick()`, `tick()`, `tock()`, and `post_tock()`.

The basic clockable objects in CHDL, D flip-flops, read their inputs during the "tick" phase and update their outputs on the "tock" phase. This subdivision of the clock enables zero-propagation-delay ideal flip-flops without any need for ordering consideration to prevent violation in simulation of the 0 hold time by the 0 propagation delay flip-flops, e.g. in shift registers. These subdivisions provide a virtual set-up and hold that can be considered vanishingly small but large enough to ensure correct synchronous behavior.

- `cdomain_handle_t`

During simulation, `tickable`s do not have to share a single clock and when written to Verilog there may be multiple clock signals. CHDL supports multiple clock domains and has been used to implement structures such as asynchronous FIFOs used to bridge clock domain boundaries.

Since the CHDL API is stateful, it can manage clock domains and does so using internal tables. The index into these tables used to specify a clock domain is `cdomain_handle_t`; this is an integer type associated with every `tickable` as the simulator progresses clock

edges are generated in clock domains, updating every `tickable` in these domains by calling their `pre_tick()`, `tick()`, `tock()`, and `post_tock()` functions.

- `sim_time(id);`

A user-visible function, `sim_time()`, provides an interface producing a count of cycles simulated in a given clock domain. Since CHDL has no concept of the passage of time beyond the count of clock edges, this is a reasonable way to manage time in simulation.

- `print_time();`

The primary output of the CHDL simulation engine is a VCD waveform file showing the evolution of selected node values over time. VCD, or "value change dump" was a file format developed for storing waveform data produced by Verilog simulations. Higher-level interfaces have been built using e.g. the CHDL console and the `egress()` function, but the raw debugging value of waveform files is difficult to discount.

The `print_time()` function prints a VCD-compatible time-step and could be used as part of a a customized waveform writer; it is also used by CHDL's internal VCD waveform writer.

- `advance();`

Advancing the clock one cycle for a given clock domain is the basic operation performed in simulation of a CHDL design, and the central function of the simulation portion of the CHDL library. It is possible to use callbacks through the `egress()` interface and other tricks, but the usual main loop of a complex CHDL simulation providing interactions between procedural C++ code and CHDL designs repeatedly calls `advance()`, instead of using the otherwise more-common `run()` function.

The `advance()` function simply calls the four functions: `pre_tick()`, `tick()`, `tock()`, and `post_tock()` of each `tickable` object in the given clock domain, or if called with no clock domain it advances the global tick count and advances any clock domain due for a tick.

- `run();`

There are two versions of the `run()` function; one takes a cycle count as an argument and a more general version takes a function returning a C++ `bool` as an argument. This allows, e.g., CHDL nodes to be used to trigger the end of the simulation. One of the arguments to `run()` is a C++ output stream, providing a place to write a VCD waveform. This may be opened using a waveform viewer such as GTKWave [10] and provides a highly valuable source of debugging information for CHDL designs.

Internally, `run()` simply calls `print_vcd_header()`, followed by `advance()`, followed by `print_time()` and `print_taps()` to print the relevant node values. Nodes get added to the list of taps by the `tap()` function, discussed in detail in Section 3.4.5.

- `finally()`, `call_final_funcs()`

Frequently, it is desirable to perform some action, such as printing statistics or writing a report file, at the end of a simulation. The `finally()` function enables this; functions can be registered using `finally()` that will be called when `call_final_funcs()` is invoked, either at the end of a simulation managed by `run()` or manually at the end of a user-defined simulation loop.

- `TruthTable`, `LLRom`

ROM lookup tables are efficiently substituted with combinational logic in many cases, yielding a synthesizable standard cell design that can be represented as a collection of logic gates. This is the purpose of `LLRom()`, a "low-level" i.e. logic gate, model of ROM. Originally this was implemented as a set of Lit values used as inputs into a multiplexer controlled by an address. The optimizations in synthesis tools handled these just as well as any other representation of the same function, and for repetitive content with the stride of the repetition a small multiple of a power of two the CHDL built-in optimizations are quite effective. For a bit of additional compute time, however it is possible to represent such look-up tables with fewer nodes, leading to smaller representations and less time expended on later optimizations. `TruthTable()` uses the well-known method of prime implicants

Table 2: Utility functions for instantiating memory.

Function	Rd. Ports	Wr. Ports	Synchronous
`Memory()`	N	M	Either
`Syncmem()`	N	1	Yes
`Syncmem()`	1	1	Yes
`Syncrom()`	1	0	Yes
`Rom()`	1	0	No

to represent lookup table contents, also taking commonalities between output bits into account. It uses an algorithm designed for performance instead of optimum output, simply combining adjacent prime implicants that differ by the value of a single bit. Selecting the prime implants to reduce in random order provides a way to quickly produce a result that, while not optimal, is potentially more compact than attempting to reduce prime implicants in bit order.

- `Ram()`, `Rom()`, `SyncMem()`

While it is technically possible to implement array structures in CHDL using combinational logic and D flip-flops, it is difficult to identify arrangements of these elements and convert them into structures in the synthesizable Verilog output that can be inferred as RAM structures by FPGA synthesis tools. If instead, the FPGA toolchains attempt to use D flip-flops and combinational logic for these arrays, performance and area suffer greatly. If the target is silicon, popular tools do not automatically convert arrays into instances of memory macro cells and instead this must be handled using Verilog sub-modules. Furthermore, simulation performance of CHDL memory is much higher than simulation performance of the equivalent set of logic gates.

Memory in CHDL is implemented with a `nodeimpl` subclass called `qnodeimpl` for all of the data outputs. For asynchronous memories, each of these outputs depends on all of the address bits, data input bits, and write signals to prevent these signals from being optimized away. In addition to the `qnodeimpl` class, there is a `memory` class that is a subclass of `tickable`. This contains arrays listing all of its address bits, data bits, and write signals for each read or write port. For synchronous ports, inputs are read during the

`tick()` phase of simulation and outputs are written during the `tock()` phase, just like D flip-flops.

The `memory` object is relatively hidden from CHDL users and is instantiated through a set of simplified template functions instantiating synchronous and asynchronous ROMs and RAMs. These functions are listed in Table 2. The most-general `Memory()` function can instantiate any type of memory but requires more arguments than the more specific `Syncmem()` functions or the specialized overloads of the `Memory()` functions. The general `Memory()` function also requires that the read ports be provided as arrays of addresses and data outputs. The single-ported `Memory()` function simply provides the read port data as a return value and takes an address as a `bvec`.

3.4.2 Optimization

The low-engineering-effort fast-executing algorithm chosen for `TruthTable()` is an example of the approach to optimization problems is taken everywhere optimization is required in CHDL. It is not a design goal of CHDL to produce optimal gate-level implementations of its input, but given the intended use of functions and operators producing combinational logic in CHDL, e.g. the use of multi-bit carry lookahead adders to represent circuits for incrementing inputs by fixed values, it is necessary to perform some optimization to prevent the netlists output by CHDL from being very large and simulation times from being unbearably long. The intent of this is not to provide world-class synthesis algorithms competitive with commercial synthesis tools. The optimizations of CHDL, modestly, aim to only enable this "lazy" and intuitive approach to hardware design with the knowledge that this kind of design shorthand will not impact simulation time, Verilog file size, or intermediate representation content.

As an example, suppose we wish to invert every second bit of an input value. To make our intent more clear we perform an exclusive or instead of doing a loop alternately instantiating inverters. We are confident that our synthesis tools downstream will catch this but (1) we are concerned about simulation time (2) we are concerned with the size of our intermediate Verilog file, and (3) we may want to perform some area estimation by

technology mapping our in-memory netlist and do not wish to have the accuracy of this unnecessarily burdened by space-consuming exclusive-or gates.

The built-in optimizations in CHDL are suitable for these kinds of light-duty goals, folding in constants, eliminating redundant expressions, and removing hardware with no path to an output or observable internal variable.

- `node_sweep()`, `opt_dead_node_elimination()`

One exception to the assertion that optimizations in `opt.h` are meant to be run once the entire design has been elaborated is `node_sweep()`. While the design is being elaborated a large number of residual nodes with no successors may build up. This is not usually a problem and repeated runs of `opt_dead_node_elimination()` by `optimize()`, called after the design is fully elaborated, will take care of dead nodes. However, in designs that, e.g., frequently algorithmically re-assign nodes, dead nodes may build up. The `node_sweep()` function simply removes all `nodeimpl` objects not referenced by a `node` object repeatedly, until only currently-valid `node`s and their predecessors remain. The final `opt_dead_node_elimination()` optimization called by `optimize()` simply re-runs this optimization.

- `opt_contract()`

"Contraction" in CHDL parlance is what *keyhole optimizations* are to compiler writers. These simple rule-based optimizations replace networks of logic gates with equivalent, smaller implementations of the same functions. The search for reducible patterns is performed, repeatedly until none are left. The contraction rules are depicted in Figure 4.

Most of these operations involve literals so it would also be fair to say that CHDL contraction is a constant folding operation; indeed this is a core purpose e.g. when performing a multiply by a constant instead of a combination of shifts and additions.

- `opt_combine_literals()`

Each literal in CHDL is unique. This is vital to its function. If assigning a different

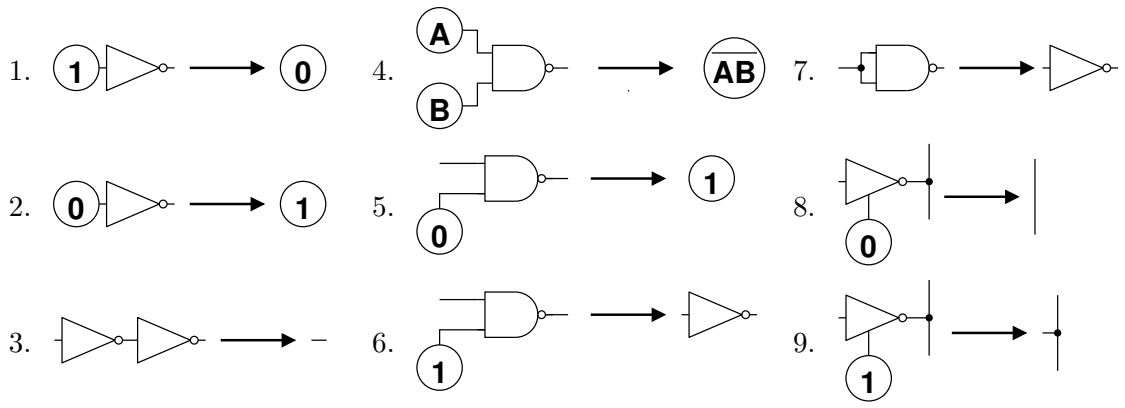


Figure 4: Contraction rules implemented by CHDL, most of which serve to implement constant folding and strength reduction type optimizations.

value to a literal zero assigned that same value to every literal zero, CHDL elaboration/-generation would not be intuitive. The `combine_literals()` optimization provides a way to dramatically reduce the space taken up by these literals in the `nodeimpl` vector and therefore the synthesizable Verilog output, at the cost of making further assignments to nodes in the design unpredictable. This is why it is typically performed after the design is elaborated, reducing the number of `litimpl`s in the design to at most 2.

- `opt_tristate_merge()`

In some cases, but not universally, it is desirable to replace bus accesses with a single tri-state driver and multiplexer. This is what `opt_tristate_merge()` does to every tristate node. Since the desirability of this transformation is not universal it is not run as part of the `optimize()` function.

- `opt_assoc_balance()`

The fundamental gates of CHDL can be thought of as 1 and 2-input logical NAND operations. Constructing logical functions having more than 2 inputs relies on structures built from these primitives. Logical operations having the associative property such as logical AND, OR, and XOR, may be assembled into combinations with an arbitrary number of inputs simply by connecting sub-blocks implementing the 2-input version of the function

output-to-input in any arrangement. Tree-like structures have less propagation delay but are less efficient with respect to routing resources. Linear structures are quite easy to route but have the longest possible propagation delay. The number of 2-input units required is the same no matter what arrangement is used.

The `opt_assoc_balance()` function arranges these units in the higher-performance configuration. Since it does not impact the compactness of the Verilog produced by CHDL it is not called by the `optimize()` function and must be invoked explicitly.

- `opt_limit_fanout()`

A late addition to the CHDL optimizations, `limit_fanout()` is included to make estimates of area based on CHDL's technology mapping algorithm more realistic by buffering the outputs of gates with fan-outs greater than a given threshold. As this does not make the generated Verilog more compact, it is not included in the set of optimizations invoked by `optimize()`.

- `opt_set_dontcare()`

The CHDL simulator does not use 6-state logic and does not include the concept of an undefined value, but it is possible to call the `Lit()` function with a literal ASCII '`x`' as its argument, and this produces a `litimpl` object having no explicitly defined value. This may be used when representing a function as a `vec` containing a truth table followed by a multiplexer to enable further optimization.

The `opt_set_dontcare()` optimization evaluates all of these `Lit('x')` nodes to literal 0 or 1, making an attempt to eliminate as many gates as possible by doing so. This is a single-pass algorithm in which each undefined `node` is visited once in turn and the locally-hardware-minimizing decision is made.

- `optimize()`

The function from the CHDL optimizations most likely to be invoked by CHDL code outside of the library itself is simply called `optimize()`. This invokes a series of optimization functions and prints, after each, the number of remaining CHDL `nodeimpl`

objects.

- `techmap()`

The CHDL technology mapping feature is intended to be used for area and power estimation and its primary advantage over synthesizing the Verilog produced by CHDL is that it does not tie up costly logic synthesis tool licenses. It does not take timing or placement into account or generate a clock distribution network and would not therefore be useful for the generation of high-performance circuit implementations of CHDL designs. It does, however, provide a valid implementation of a CHDL in-memory netlist using logic cells from a specified library, providing a usable upper bound for area and lower bound for performance in a given technology.

The algorithm employed by `techmap()` is a simple greedy tree matching algorithm taking a set of (1) logic cells expressed as trees of NAND, invert, D flip-flop, and tristate functions and (2) a CHDL in-memory netlist as inputs and writing as output a textual representation of the technology mapped netlist.

3.4.3 Node Types

- `litimpl`, `Lit()`

Arguably the simplest component available to the designer of logic circuits is the tie to a constant logic level. In CHDL, subclasses of `nodeimpl` are used to provide logic functionality, each providing its own implementation of the `eval()` function used during simulation. The `eval()` function of `litimpl` simply returns a fixed 1 or 0 value and provides a way to mark constants to be propagated during optimization.

- `invimpl`, `nandimpl`

The inverter and NAND gate implementations simply implement their associated combinational logic functions. Their `eval()` functions simply return the logical NAND of their predecessors. Perhaps `Nand()` and `Inv()`, and even `Lit(1)` could have been generalized into a single n-input NAND gate, but the separation into a NAND-inverter graph containing

only the one and two-input versions of the NAND function simplifies the process of finding matches during technology mapping, when cells available in a logic library are mapped to the generic in-memory netlist. A performance enhancing feature used during simulation implemented by both `invimpl` and `nandimpl` is memoization. During a given simulation clock period, as discussed in the following discussion of `cdomain` and `tickable`, the value of a `nandimpl` or `invimpl` is unchanging. This is cached along with the most recent time step at which an evaluation occurred. If the value is recent, the inputs are not re-evaluated, saving a potentially exponential-in-logic-layers number of calculations that quickly renders even the simplest of simulations impossible.

- `regimpl`

The basic sequential logic unit in CHDL is the D flip-flop. As described in Section 3.2 there are many advantages to synchronous logic design and while it is not the only paradigm used, especially in highly-specialized high-performance structures, the advantages of a synchronous design methodology in testability and verifiability makes it at least a compelling default. Because of this, CHDL is designed with only synchronous logic in mind, and in many designs, the D flip-flop is the only sequential logic element present.

A CHDL D flip-flop is both a `tickable` object and a `nodeimpl`, providing an output that depends on the value of an input in a prior cycle. Since this dependency crosses a temporal boundary, the input node's ID is not held in the `pred` array of the `nodeimpl` base class but as a special member variable `d`. This distinction holds for all sequential logic so that, e.g., combinational logic cycle detection can be performed without knowledge of the particulars of sequential logic in CHDL.

As mentioned in the Section 3.4.1, the inputs to `regimpl` are evaluated during the `tick` phase and the outputs are updated during the `tock` phase. This ensures that any other logic depending on the `regimpl`'s output during a clock cycle reads the appropriate value. Failure to do so can be interpreted as a simulated hold violation, where the 0 propagation delay D flip-flop is too fast for the 0 hold time D flip flop that depends on it and the new value inadvertently gets propagated through. This may lead to such strange effects as

stages in shift registers being skipped. Subdividing `tick` into four steps is like imposing infinitesimal propagation delays on CHDL elements and serves to eliminate this kind of simulated hold time violation.

- `tristatenode` and `tristateimpl`

Unlike more circuit-oriented logic simulators supporting six-level logic, values in the CHDL simulator are restricted to Boolean values. In the CHDL simulator nodes cannot be weakly or strongly pulling up or down and cannot be high-impedance or undefined. Such characterization is important in some domains but in the synchronous world of CHDL it is possible to simulate the relevant behavior of a wide range of structures and describe their structure in a synthesizable manner without resorting to six-level logic. Tristate structures like buses are one area where it may be counterintuitive that they can be described in CHDL, simulated, and synthesized from CHDL descriptions with only support for Boolean values. The key observation is that, while not every participant in a tri-state net is driving it all the time, in any correctly functioning system employing tri-state elements any time a try-state net is being read, a value is being driven on that net by at least one driver. Taken in aggregate a tri-state net could be modeled as a single logic function including every driver and each driver's enable signal.

It is, of course, possible to model this behavior as a simple network of basic logic gates and not involve a CHDL concept of tri-state nodes at all. Unfortunately the most important use of tri-state modules in CHDL is not *within* modules; whether to use a traditional multiplexer design or a tri-state design internally is more of a logic synthesis question than a basic modeling question. Instead, the most important use of tri-state nodes is *between* modules, in the definition of interfaces.

Since tristate nodes must be able to be exposed as tristate by CHDL's synthesizable Verilog generator, a `trisimpl` node type is provided by CHDL. It has a variable, even number of inputs alternating between input signals and their corresponding enable signals. When set up as an output a `trisimpl` node will generate a Verilog `inout` port. A `connect()` member function provides a way to add new `trisimpl` inputs without directly

Table 3: Logical operations and overloads.

Function	Operator	Description
`Inv()`	`!`	Logical NOT
`Nand()`		Logical NAND
`Nor()`		Logical NOR
`And()`	`&&`	Logical AND
`Or()`	`||`	Logical OR
`Xor()`	`!=`	Exclusive OR
	`==`	Exclusive NOR

manipulating the `pred` vector, though this is merely for internal convenience.

Unlike other node types in CHDL, it is necessary for `trisimpl` to have its own corresponding `tristatenode`. This type includes a member function, `connect(x, en)`, that connects an input to the `tristatenode` through a tri-state buffer with an enable signal.

3.4.4 Logic Functions Library

- Logical Operations and Operators

For the CHDL `node` object, all of the usual logical operations are provided. By convention in CHDL, functions that act as generators of hardware are named in camel case, i.e. with the beginning of words within the function name denoted by a capital letter and not an underscore or other separator, with an initial capital. Table 3 provides a full list of these functions and operator overloads. The basic logic functions, `Inv()` and `Nand()` instantiate `invimpl`s and `nandimpl`s respectively. The others build their operations out of these basic operations.

- `vec<N, T>`: Fixed-length vectors

Much of the utility of RTL representations over gate-level netlists comes from the ability to represent groups of signals with single names, e.g. as arrays and pass them around as aggregates. CHDL provides a data type `vec<N, T>` that combines N instances of type `T` and allows them to be indexed both with integers and instances of class `range<A, B>`

Table 4: Bit vector operations and overloads.

Function	Operator	Description
`Not()`	`~`	Bitwise NOT
`And()`	`&`	Bitwise AND
`Or()`	`|`	Bitwise OR
`Xor()`	`^`	Bitwise XOR
`AndN()`		AND-reduce
`OrN()`		OR-reduce
`XorN()`		Odd parity (XOR reduce)
`EqDetect()`	`==`	Equality
	`!=`	Complemented equality

to select sub-vectors. Due to its ubiquity, the type `vec<N, node>` is aliased to the name `bvec<N>`. The `bvec` is used throughout CHDL and related libraries whenever a collection of bits is needed.

The constant `sz<vec<N,T>>::value` is simply defined as N times `sz<T>::value`, so `sz<bvec<N>>::value` is simply N. This potentially-recursive description allows the sizes of multi-dimensional arrays to be computed at compile time.

- Basic bit vector operations

It would be highly inconvenient to need to break signals out of bit vectors one-by-one in order to perform operations on them, so basic functions and, where available, corresponding operator overloads are available for basic operations on bit vectors. These are summarized in Table 4. Arithmetic operations and shifts are also available, but as these require the generation of nontrivial combinational logic they are covered separately.

A feature of C++ added within the past decade that is used by the `vec<N, T>` types is list initialization; the ability to initialize collections by passing a list of initial contents to the constructor. With CHDL vectors, this feature allows the vector's value to be assigned without manually assigning each entry, saving optimization time by avoiding the creation of unused `Lit(0)` objects.

- `Cat()` and `Flatten()`

Multi-dimensional arrays are useful structures, as are the aggregate types explored in Section 3.5, but sometimes it is preferable just to have a flat one-dimensional array of bits. The `Flatten()` function for CHDL `vec` types takes any `vec` type and recursively calls `Flatten<T>()` on its members. The `Flatten()` function for `node` returns a `bvec<1>`. The end result is that `Flatten()` called on an object of size `T` bits returns a `bvec<sz<T>::value>`.

A set of utility functions for concatenating vectors together, as well as adding single elements to vectors, are all four called `Cat()`. These are templated functions that take two arguments, either of types `vec<N, T>` and `vec<M, T>` or `vec<N, T>` and `T` and concatenate them together into a single vector. There is also a single-argument version of `Cat()` that returns a special `concatenator<T>` object. This is a functor, meaning it can be called as a function, allowing long concatenations to be built with a syntax like "`Cat(a)(b)(c)...`". The final result from this, a `concatenator`, may be automatically cast to its result type.

Because the `node` type in CHDL has the unique property that its assignment operator is `const`, since assignments do not modify the `node`s themselves but the internal connections between `node`s and `nodeimpl`s, non-reference return values from functions can be used as the target of assignments. This means that, in CHDL, it is common to see such constructions as "`Cat(a, b) = c;`" and "`Flatten(a) = b;`".

- `Zext<N>()`, `Sext<N>()`

A zero extension and sign extension function are provided for `bvec`s. These utility functions allow bit vectors to be expanded as needed. If passed an argument smaller than the size of the input `bvec` they also function as truncation functions.

- `Shifter()`, `<<`, and `>>`

The first function mentioned in this section, `CLOG2()` is vital for shifters, as the number of bits needed to specify the shift amount is equal to $\lceil \log_2 N \rceil$, where N is the number of bits in the input and output. CHDL provides both overloads for the C++ `<<` and `>>`

operators that operate like their unsigned counterparts and a general `Shifter()` function that implements a barrel shifter with arithmetic, logic, and rotate modes. These operations are truncating; the width of the output is the same as the width of the input. This is to accommodate typical usage in datapaths of fixed width.

- `Adder()`, `Mult()`, `Div()`, `+`, and `-`, `*`, and `%`

The `Adder()` function instantiates a Kogge-Stone carry lookahead adder with carry inputs and outputs. This is used by operator overloads for the binary `+` and `-` operators to generate an adder; in this case only the sum is returned. The `Mult()` function and equivalent * operator generates a Wallace tree multiplier. The `Div()` function produces an integer divider formed by performing up to one subtraction per bit of input. This is a quite a low-performance operation and is intended primarily for division by constants, a scenario in which most of the logic required for general division is optimized away. The arithmetic operations truncate their results and the result has the same number of bits as the operands to accommodate typical usage in a fixed-size data path.

- `Mux()`

CHDL provides two classes of multiplexers, a 2-to-1 multiplexer with a single `node` as a select, and a general N-to-1 multiplexer with a `bvec<`$\lceil \log_2 N \rceil$`>` select signal. Both of these are implemented as templated functions, building multi-bit multiplexers using the `Flatten()` function and a the version of the function for `node`. The N-to-1 multiplexer is built recursively from $\frac{N}{2}$-to-1 multiplexers, with a base version for the degenerate case of a 1-to-1 multiplexer taking a 0-bit select signal.

- `Decoder()`, `Enc()`, `PriEnc()`, `Log2()`

CHDL provides a function, `Decoder()`, implementing a general N-to-2^N encoder, with an optional enable input. Encoders are also provided, including an encoder that expects a one-hot input, producing meaningless results if more than one input is high, and two versions of priority encoder. The two priority encoders differ only in the priority order of

their inputs, with `PriEnc()` prioritizing bits with smaller indices and `Log2()` prioritizing bits with higher-numbered indices, thereby having an output whose value is $\lfloor \log_2 x \rfloor$ for input x.

3.4.5 Inputs, Outputs, and Simulation Interfaces

- `tap()`, `TAP()`, and `OUTPUT()`

As C++ does not provide an automatable way to expose the names of variables within a program to that program during execution, naming the signals tracked during simulations and naming the ports of generated synthesizable Verilog must be done through alternate means. The mechanism for this is the tap system. The `tap()` function identifies signals of interest for waveform output and also, when passed `true` as its default-false final argument, identifies signals that should be made outputs of the generated Verilog. It is intended to be overloaded for aggregate types so a single call to TAP exposes all of the constituent signals of these. These are held in an array that is traversed during netlist generation and during every cycle of simulation. A flag indicating whether it is an output is stored with each entry. CHDL `nodeimpl`s and vectors thereof marked as taps have their value output, if it changes, into the `.vcd` waveform file on each cycle of the simulation.

To add a signal to the tap array, the `tap()` function is called with a string representing a signal name and a signal; either a `node` or a `vec` as its arguments. Macros `TAP()` and `OUTPUT()` are provided for convenience, using the stringification feature of the C++ preprocessor to report the signal name as it appears in the macro invocation.

Taps are important to CHDL optimizations because they are used to mark nodes as being visible. Nodes that are not visible from taps may be eliminated by the dead node elimination operation. Because of this, a special category for nodes that are used internally during simulation but are not visible when directly following combinational logic from outputs, the ghost tap, is also available.

- `Input()` and `Input<N>()`

Outputs to the generated Verilog module are simply special cases of taps; identifying

signals that already exist as the outputs of already-declared logic. Inputs, however, are different. A special subclass of `nodeimpl` which cannot be simulated, `inputimpl`, is needed for inputs to the module. The functionality provided is a basic set of inputs that may be either single `node`s or `bvec`s; one-dimensional arrays of nodes. This is expanded on in the template library's support for directed structured signals, described in Section 3.5.2, where inputs and outputs consisting of multi-dimensional arrays may be declared, but all of this support is built using the simple `Input()` and `Input<N>()` functions and each sub-component of a structured signal or multi-dimensional array becomes a differently-named single input or output.

- Design Hierarchy

Sometimes it is useful, for debugging or design analysis, to have a concept of design hierarchy. CHDL includes functions `hierarchy_enter()` and `hierarchy_exit()` and their macro equivalents `HIERARCHY_ENTER()` and `HIERARCHY_EXIT()` that allow the designer to tag levels of design hierarchy. Traditionally these are placed at the beginning and end of functions and the macro versions automatically tag a region of the design with the name of the function from which they are invoked. Since it is convention in CHDL to use functions to define design modules, this provides a reasonable hierarchy, and all CHDL functions defining hardware modules, from `Div()` to `And()` include calls to `HIERARCHY_ENTER()`/`HIERARCHY_EXIT()`. Position within the hierarchy is maintained as a property of `nodeimpl` objects and the optimizations preserve this value.

- `Submodule()`

When `print_verilog()` is called, the entire CHDL in-memory state is written to a single synthesizable Verilog netlist. If the intent is to use CHDL to design a block that *contains* IP provided with a Verilog interface, e.g. hard macros or soft macros, the `Submodule()` interface is used to instantiate these blocks. The only argument to `Submodule()` is the name of the module type. This is followed, through a functor interface much like the single-argument `Cat()` function, by a list of input signals, a list of output signals, and a

list of bi-directional signals into and out of the instantiated module. This creates a `module` object, and all of the outputs of this object are new `mnodeimpl` objects.

The `Submodule()` interface provides an illustration of why CHDL's interoperability between different hardware description paradigms and levels of abstraction is valuable. The use of external IP written in Verilog requires that IP to be wrapped in a CHDL interface. If, for example, that IP provides some floating point operations, any operators performing floating point operations would have to be overloaded for the specific subset of floating point data types supported by the available IP catalogue. This interface layer must be written and maintained separately from both the designs in CHDL that use it and the IP itself, leaving it independently vulnerable to errors and in need of maintenance. If both of these pieces were described in the same language, no glue layer would be needed. As a multi-paradigm language, CHDL hopes to reduce the number of such language boundary crossings needed in the typical design.

- `Ingress()` and `Egress()`

The simulator state is available to code outside of CHDL through the `Ingress()` and `Egress()` functions. These functions allow variables and functions external to the CHDL simulator to be updated by the CHDL simulator as the simulation progresses. The simplest versions of the `Ingress()` and `Egress()` functions simply call functions passed in as arguments and call them with values from the CHDL simulator as arguments. C++ lambdas may be used as a simple mechanism to give these functions access to global and local variables. Utility functions `IngressInt()` and `EgressInt()` enable integer types to be updated from within the CHDL simulator.

The purpose of these functions is to allow, while using the `advance()` function to step the simulator through single cycles, interactions between external code and CHDL-defined hardware designs. The domain in which this is most potentially high-impact is the interoperation between the CHDL simulator and simulation frameworks like the Structural Simulation Toolkit [47], described in Section 3.6.1.3. By allowing external models for memory system components to be used, the CHDL SST component allows the evaluation of

architectures developed with CHDL in configurations that would be difficult to simulate entirely at the bit level.

The functions `ConsoleIn()` and `ConsoleOut()` use `Ingress()` and `Egress()` to allow simulated hardware to be connected directly to console input and output. This is designed to provide a simplified way to evaluate core designs running benchmark and test programs that produce textual output or require textual input.

- `Assert()`

Much of the validation and verification support provided in traditional hardware description languages by non-synthesizable behavioral code is provided in CHDL mostly within the logic of the simulated hardware itself. The `Assert()` function and accompanying `ASSERT()` macro provide a way to stop the simulator and generate an error message containing the line number of the failed assertion when a predicate becomes false. This increases error exposure in simulation without adding additional hardware to synthesis output.

3.5 CHDL Template Library

While the CHDL core library provides a set of logic functions and a vector data type in addition to its basic netlist manipulation features, the set of included modules is fairly basic. Features and pre-designed modules not included in the CHDL core library that are considered universal enough to be widely useful are included in the CHDL-STL instead. This includes support for structured data types, directed data types, pseudo-random number generation, and RTL-style conditional assignment expressions.

3.5.1 Aggregate Data Types: `ag` and `un`

A C++ `struct` or `class` containing CHDL datatypes is an appropriate choice for many types of structured data that may be used in CHDL, especially in cases where there are combinational functions of the elements used frequently that may be generated by member functions. By assigning bits from members to other members in the constructor, fields of such a data structure can be made to refer to the same data, similar to a C `union` data type. The disadvantage to this type of approach to structured data is that the names of

the fields and the fact that they are arranged into a structure are invisible to the simulator and netlist generator.

A template metaprogramming technique known as the type list was applied to the problem of combining multiple fields into single structured data types in the CHDL template library. This technique and a technique for converting strings into C++ types were employed to create a system of structured signals in CHDL, supporting both `ag`, types, in which the fields are concatenated end-to-end in the flattened representation and `un` types in which the fields all map to the same set of `node`s, occupying the space of the largest member. The use of a type list allows template metaprograms to operate on `ag` and `un` types, iterating through elements at compile time. This is used by the `sz<ag<NAME,T,NEXT>>` template class specialization to compute the size of aggregate types and the `Flatten()` function template to recursively concatenate aggregate types into a single `bvec`.

3.5.2 Directed Data Types: `directed`, `in`, `out`, `inout`

Aggregate types provide a convenient way to bundle signals together into groups that may be manipulated as a single unit and passed through generators provided by the CHDL core library including `Reg()` and `Mux()`. A shortcoming of aggregates alone is that their fields do not specify direction. Calling the `OUTPUT()` macro on an `ag` or `un` creates outputs out of all of the fields. The class template `directed<T, DIR>` and its aliases `in<T>`, `out<T>`, and `inout<T>` provide a way to designate signals as inputs or outputs to their respective blocks. A `reverse<T>` template metaprogram is available to produce the complementary directed type, allowing both side of an interface to be declared without replicating code. Signals of a directed type and its complement may be connected using a `Connect()` function that is also provided as part of the directed signals library.

3.5.3 Loadable Module Support

Directed aggregate types allow the signal structure, if not the protocol, of interfaces for modules to be expressed in a single data type. This is exploited by the CHDL module loader, a separate feature from the external Verilog module support provided by the CHDL core library. The CHDL module loader provides an `Expose()` function that creates inputs

and outputs based on the direction of each signal. A `LoadModule()` function is provided that parses netlists with `Expose()`-created interfaces and allows them to be connected to aggregates with complementary interfaces. This allows the distribution of IP as loadable modules and the combination of pre-optimized IP into larger-scale single modules. This is the preferred method for IP distribution in CHDL, as it allows simulation within CHDL unlike Verilog sub-modules.

3.5.4 IP Block Generators

- Linear-Feedback Shift Registers

Linear feedback shift registers produce a string of bits with a nearly 50% 1/0 balance with a period of 2^N for N bits. This makes them useful for a variety of pseudo-random number generation and counting applications. The LFSR implementation in CHDL takes the polynomial and number of bits generated per cycle as template arguments.

- Sorting Networks

Sorting networks are an abstract mathematical formulation of the problem of parallel sorting, in which sorting algorithms are constructed from basic compare-and-swap operations. It is known that optimal sorting networks can be discovered that have depth and thus, in hardware, time complexity $O(\log N)$ of these operations, but discovering these and even proving that they are sorting networks is computationally hard. There are, however, efficient algorithms for producing sorting networks of depth $O(N \log N)$ and one of these, Batcher's even-odd merge sort is used to produce CHDL's sorting network implementation. CHDL's implementation of this is templated so that it may be used on any signal type providing a comparison operator.

- Hardware FIFOs and Stacks

Templated functions generating both stacks and FIFO queues, called `Stack()` and `Queue()` are provided as part of the template library. These simple utility functions are included because queues are needed by the memory system components where they are used as buffers and stacks only require a simple modification to these structures.

Table 5: Memory system components provided by the CHDL template library.

Component	Description
`Scratchpad()`	Scratchpad SRAM with `memreq` interface
`Rom()`	Pre-programmed ROM with `memreq` interface
`ExtAddr()`	Extend or truncate address
`Share()`	Share single interface among multiple cores
`SizeAdaptor()`	Adapt narrow core interfaces to wider interfaces

Table 6: Numeric types defined in the CHDL template library.

Component	Description
`si<N>`	Signed integer of length N
`ui<M>`	Unsigned integer of length N
`fxp<W,F>`	Fixed point, W bits whole and F bits fractional
`fp<E,M>`	Floating point, E-bit exponent and M-bit mantissa

3.5.5 Memory System Interfaces and Components

Core and accelerator designs in CHDL share a tagged memory system interface defined by basic memory request and response types, provided by the CHDL template library. In addition to the types themselves, `memreq<B, N, A, I>` and `memresp<B, N, I>`, where

- `B` is the width of a single "byte", the minimum writable unit, in bits,
- `N` is the number of bytes in a single transfer "word",
- `A` is the length of the word address, and
- `I` is the length of the request ID/tag in bits.

there are a number of IP blocks for memory system components, enumerated in Table 5.

3.5.6 Numeric Types and Operations

A variety of numeric types are included in the template library, along with functions generating arithmetic operations and operator overloads for those functions, and a pair of functions, `IToF()` and `FToI()`, for converting between integral and floating point types. The CHDL numeric types are enumerated in Table 6.

```
const NN(2*CLOG2(N + 1));

rtl_reg<bvec<N> > a;
rtl_reg<bvec<NN> > i(Lit<NN>(2)), j;
rtl_reg<node> init(Lit(1)), find, mark;

IF(init) {
  IF(i == Lit<NN>(N)) {
    i = Lit<NN>(2);  init = Lit(0);  find = Lit(1);
  } ELSE {
    a[Zext<CLOG2(N)>(i)] = Lit(1);  i += Lit<NN>(1);
  } ENDIF;
} ELIF(find) {
  IF(i * i >= Lit<NN>(N)) {
    find = Lit(0);
  } ELIF(Mux(Zext<CLOG2(N)>(i), a)) {
    find = Lit(0);  mark = Lit(1);  j = i * i;
  } ELSE {
    i += Lit<NN>(1);
  } ENDIF;
} ELIF(mark) {
  IF(j >= Lit<NN>(N)) {
    i += Lit<NN>(1);  find = Lit(1);  mark = Lit(0);
  } ELSE {
    a[Zext<CLOG2(N)>(j)] = Lit(0);  j += i;
  } ENDIF;
} ENDIF;
```

Figure 5: Example design using CHDL-RTL; an implementation of the Sieve of Eratosthenes.

3.5.7 RTL Description in CHDL

The most universally-accepted method for providing designs to synthesis tools targeting silicon and FPGAs is register transfer level description in the synthesizable subset of Verilog or VHDL. CHDL as presented so far enables the construction of complex designs but assignments are continuous and registers are instantiated as hardware blocks. The presentation of designs as a set of assignment statements that occur when triggered by the combination of a clock rising edge and a logical predicate described as a set of nested conditions is not allowed by the CHDL core library which, to borrow terms from Verilog, allows `wire`s but not `reg`s. A CHDL register is more akin to instantiating a group of D flip-flop primitives in such a hardware description language than it is to an RTL representation. Due to the ability of the `Reg()` function in CHDL to be generic this is not as much a limitation as

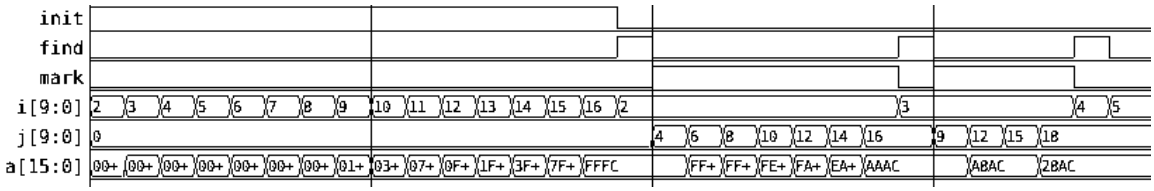


Figure 6: Waveforms for example from Figure 5, given N is 16.

it may sound, but for more complex state machines the ability to have potentially-nested RTL statements would be preferred.

For this reason, CHDL-RTL was added to the CHDL template library. This provides a templated data structure, `rtl_reg<T>`, that may take zero or one assignments per clock cycle from an assignment statement, predicated on whether that assignment statement would be "reached" in its corresponding position in a nested block of `rtl_if()`, `rtl_elif()`, and `rtl_else()` calls. These calls may be wrapped in optional macros to replace the longer names like "`rtl_if`" with shorter names like an all-caps `IF`, as used in the example in Figure 5.

3.6 Introspection

The CHDL libraries provide both low-level and high-level forms of hardware description. With the core library and the template library, it has been shown that it is possible to describe designs from the gate level through the register transfer level, and higher-level options are presented in later chapters. CHDL not only encompasses multiple levels of abstraction, but it also stands in for several phases in the design tool chain. It is not uncommon for the same program to generate a design, produce simulation results, and output a standard cell netlist. This vertically integrated design coupled with the APIs provided by CHDL allowing the reading and modification of netlists enables a novel technique called *netlist introspection*; interaction with the design structure by the same program that generates it. In the remainder of this section, it is shown that:

- Netlist introspection can be used to implement a range of performance, design quality, and utility enhancing software modules.

- With netlist introspection as a part of the CHDL API, these software modules may be implemented as part of CHDL, as external libraries, or alongside hardware designs.

This key feature is one of the aspects that distinguishes the CHDL environment from similar tools like Chisel [4] and SystemC [42]. While these bring the power of a general-purpose programming language to bear on hardware design problems, they do not include as part of their API a low-level interface enabling netlist introspection. This could potentially have impact on design costs for accelerator architectures by reducing the critical path of the design cycle and improving support for rapid prototyping and evaluation.

3.6.1 Applications of Netlist Introspection

The exposure of the `nodes` array and the `nodeimpl` subclass declarations to the library user enables netlist introspection. This feature enables a host of novel techniques, including those described in this section, bringing down synthesis, optimization, and simulation times and enabling more rapid design space exploration and novel architectures.

3.6.1.1 Module Caching

CHDL is able to handle designs at scales ranging from single arithmetic functions to billions of logic gates. At the larger end of this scale, the amount of time required to build a design can become quite large, limiting opportunities for design iteration and improvement. This build time comes in several stages:

- Compilation; running the C++ compiler to convert design `.cpp` files into executable generators.
- Elaboration; executing the generators and building an initial representation of the design.
- Optimization; reducing redundancy and improving performance in the internal design representation.
- Simulation and validation; producing result vectors and validation results.
- Synthesis; producing a netlist suitable for use by an FPGA or ASIC design flow.

Build systems are a widely-accepted solution for decreasing compile times. By dividing designs into multiple source files and automatically re-building only the sources that change between iterations, compilation time can be drastically reduced. Similarly, elaboration and optimization time can both be reduced through module caching. With module caching, a CHDL function is executed only if a corresponding cache file does not already exist on disk. Otherwise, the cached module is loaded from the disk in lieu of generating it. Caching can be enabled for large pieces of the design, leading to considerable performance improvement for elaboration following the initial run, even in cases where parts of the design outside of the cached portion are modified. Performance can be increased further by pre-optimizing the cached module files, decreasing their size and eliminating redundant optimization.

The module loader and writer used for caching are the same as the user-visible module loader mentioned in Section 3.4, except inputs and outputs are not declared using directed aggregates. Both inputs and outputs are defined collectively and flattened down into a single large I/O vector. This enables compatibility with both directed and non-directed aggregates, vectors, nodes, and user-defined types, as long as they conform to CHDL conventions. The performance improvement experienced by a simple floating-point design, a Mandelbrot set visualizer, is shown in Figure 7. The modules being cached in this case are the floating point adder and multiplier. Elaboration and optimization performance collectively doubled and elaboration performance alone more than quintupled in the double precision version of that design. Naturally, as the logic functions being cached comes to represent a larger portion of the total design size, the amount of performance improvement experienced increases.

3.6.1.2 Mixed-level Microarchitecture Simulation

Implementation and simulation of microarchitectures occupy two very distinct realms of activity, despite the fact that simulators are, in a sense, implementations of relevant features. A large part of the problem is simulation speed. A gate level implementation of a processor is a very high-fidelity model, but runs much more slowly than a comparable higher-level simulation. In the example, a simple accumulator architecture processor model was taken

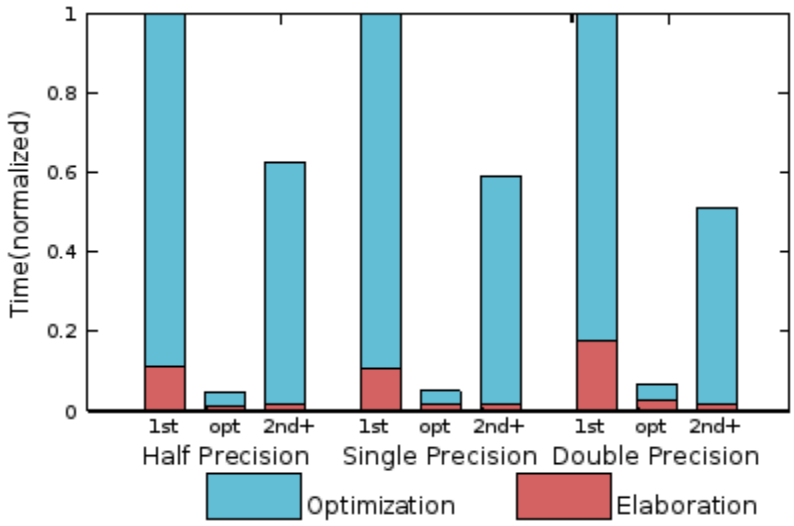


Figure 7: Caching and pre-optimization of floating point operation implementations improves both elaboration and optimization time for a Mandelbrot sample design. Bars show run-time of first design elaboration, optimization command run, and subsequent runs with pre-cached floating point operation netlists.

and all instances of processor words and their equivalent `node`s and `nodeimpl`s were replaced with a higher-level modeling construct, a full processor `word` with an associated set of arithmetic operations and a `nodeimpl` subclass for interoperating with gate-level portions of the design. Using the `nodeimpl` subclass, a way has been built to access individual bits when needed, while replacing all gate level modeling of arithmetic operations on these words with integer operations. This accelerates elaboration, optimization, and simulation and is easily disabled for synthesis with a single `#define`.

Figure 8 presents the results of an analysis in which the bit width of the accumulator-based variable-width Harvard architecture integer processor was swept from 16 bits to 64 bits and the elaboration, optimization, and simulation time of both the gate-level and mixed-level models was recorded. As expected, the higher-level model outperformed the lower-level model significantly, and this rift in performance only increased with the data path width.

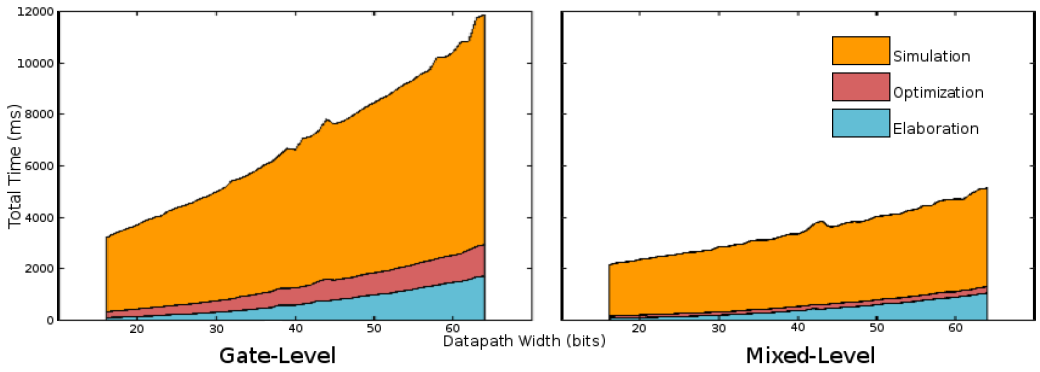


Figure 8: Significant speedup in all stages of design can be achieved through the use of mixed-level simulation of microarchitecture models.

3.6.1.3 Integration with the Structural Simulation Toolkit

Mixed-level modeling has also been implemented by packaging the CHDL simulation environment as a component for the Structural Simulation Toolkit [47]. This makes use of the introspective `ingressimpl` node type and the `Egress()` function to provide input and output over SST communications links for CHDL components. This allows high-performance modeling of potentially many memory system components and cores while enabling low-level designs implemented in CHDL to be simulated in low-level detail and debugged. This integration was used successfully to debug the load-linked and store-conditional instruction implementations provided by the Iqyax RISC core described in Section 4.1 in multi-core operation, providing a valuable mechanism for debugging this support within SST's library of memory system components. For this purpose, multiple Iqyax cores were simulated using the CHDL simulation environment, in conjunction with a simulated coherent cache and network-on-chip.

3.6.1.4 User-Defined Optimization: Retiming

The optimizations mentioned in Section 3.4 are implemented using the same interfaces as netlist introspection. There are no additional internal APIs used by these optimizations; they could just as easily have been implemented outside of the CHDL library itself.

One transformation in particular has proven useful in this work, e.g. in the FPU of the Harmonica soft core [30], but is not used frequently enough to justify inclusion in

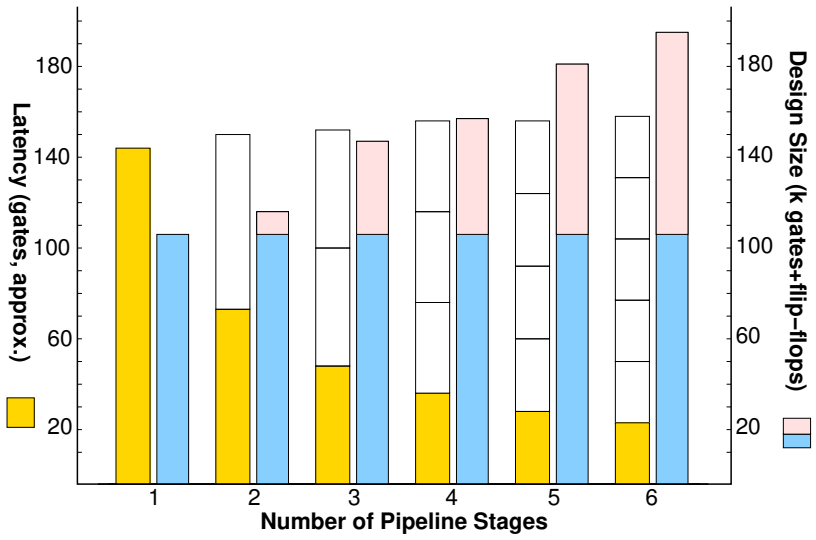


Figure 9: The retiming transformation automatically pipelines functional units. In this example, a floating point multiplier is retimed to between 2 and 6 stages, leading to an increase in achievable throughput with a slight overhead in latency and design size.

the main CHDL library, and this is the automatic generation of pipelined functional units from non-pipelined, combinational implementations. Anticipating a reconfigurable fabric as a target, in which adding D flip-flops to complex logic is not very costly, this algorithm simply organizes the relevant gates into layers, finds edges in the netlist that cross the layer boundaries, and inserts additional registers on these edges. This simple algorithm, implemented in 160 lines of C++, drastically increases the throughput that can be achieved when using complex functional units as illustrated in Figure 9. Implementing this algorithm using netlist introspection allows it to be contained within the source for the design that requires it, not externally added to the tool flow as a separate program, reducing workflow complexity and eliminating a potential source of error.

3.6.1.5 User-Defined Analysis: Power Emulation

Power emulation [14] is a class of techniques for modeling the power consumption of a hardware system using an FPGA. Using CHDL, a power emulation framework that takes advantage of netlist introspection to augment the in-memory netlist with a high-performance activity-based energy model has been prepared. This energy model assumes that the energy required to toggle the state of any one gate's output, including driving any interconnect

following the gate, is invariant, intentionally ignoring temperature and state dependence. The total energy expended during a given cycle is modeled as the sum of toggling energies for all gates that toggle during a given cycle, or:

$$E_{\text{cyc}} = \sum_{i=0}^{\#\text{gates}} E_i T_i$$

where E_i is the dynamic energy expended during the cycle if the node toggles and T_i is 1 if the node toggles in a given cycle and 0 otherwise. E_i is computed within the power model for a given node as the sum of the intrinsic toggle energy for the gate producing its value and the input energies for successors:

$$E_i = E_{\text{int}} + \frac{1}{2} V_{\text{dd}}^2 \sum_{\text{inputs}} C_i$$

where C_i is the capacitance for gate input i, a function of the types of the successor gates, and E_{int} is the intrinsic energy expended by the gate type during toggling; a function of the gate type. This is computed statically based on the post-technology-mapping netlist and is an intrinsic part of the model; it does not change over time and is hard-wired into the energy model produced. It does not directly take interconnect into account, since it is not based on a physical placement of the gates, and for all analyses performed using this tool, a constant scaling factor for interconnect overhead has been used.

The challenge, ordinarily, of running such a model, is the time required to compute T_i, whether each gate toggles, for each cycle in the program. By running the netlist in an FPGA instead of performing traditional gate-level simulation, this issue is side-stepped. The primary disadvantage of fully instrumenting such a design running in an FPGA is a significant overhead in FPGA area. This model is flexible in that it allows instrumentation of a smaller number of gates to decrease area overhead at the cost of accuracy. A subset of gates, a *static sample* can be selected instead. Controlling the size of this sample allows a trade-off of accuracy to be made for FPGA area overhead and performance, and perfectly reasonable models can be created by instrumenting only 10% of the logic gates in a design, as discussed in the evaluation.

This energy model generator is based on the activity detector circuit shown in Figure 10.

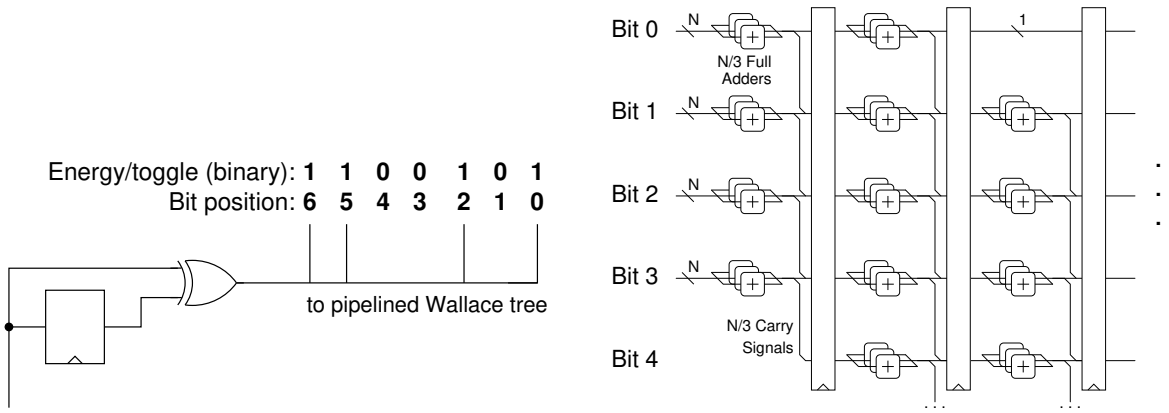


Figure 10: The basic gate-level instrumentation and the pipelined Wallace tree into which it feeds.

This is connected to each instrumented gate and its outputs are connected to a design-specific adder tree to perform the summation step. The output of this adder tree is the module's `cycle_energy`.

The adder tree is a pipelined Wallace tree. This structure for adding N numbers in $O(\log N)$ time and $O(N)$ space is traditionally used for building multipliers by adding shifted partial sums from N adders. In this case, this structure is used to sum the outputs of all of the activity detectors, connected in such a way as to evaluate to E_i if the gate toggles and 0 if it does not. This is done by connecting each activity detector's output to a subset of Wallace tree inputs as shown in Figure 10.

The input to the Wallace tree can be thought of as a vector v of sets of bits, where each element v_j of the vector (each set of bits) corresponds to the set of 1 bits in the jth bit position the E_i values of all instrumented gates. The Wallace Tree achieves its $O(\log N)$ performance by performing a division by a constant factor of $\frac{3}{2}$ of the number of bits in each layer. This is done by feeding sets of 3 bits in each layer into full adders, reducing them to 2 bits in the next layer, the sum bit at the same bit position as the inputs and the carry bit at the next bit position up. This is done until the last layer, where there are only 2 or fewer entries left in every bit position, at which point an ordinary addition is performed.

The summation tree is pipelined by placing D flip-flops at fixed intervals along the tree, limiting path lengths and thereby limiting potential impact on achievable clock rates. The right-hand side of Figure 10 illustrates the regular placement of D flip-flops between layers

of adders.

Static RAM Static RAM arrays frequently appear in digital circuits, even ones that are not memory-oriented. Of the benchmark netlists, for example, all of the processors contain SRAMs. These are used as register files in `harmonica2`, a GPU-like core, as BTB tables and the register file in the 5-stage RISC-like `Iqyax`, both discussed in Chapter 4. They are also used as a small scratchpad memory in the tiny microcontroller called `t-core` in this evaluation. These are modeled similarly to gates, except the exclusive-or based activity detectors are replaced with circuitry to detect reads and writes. Read and write energies for each type of RAM present in the system must be provided. These can be generated through the use of SRAM models like Cacti [55], which was used in this evaluation.

Other Sources This technique covers the potential sources of dynamic energy dissipation fairly well, and static energy dissipation can be analyzed offline, but there are other potential sources of power dissipation that must be considered. In addition to the cost of toggling gate outputs and the internal gate energy dissipation already covered, energy dissipated overcoming wire capacitance is quite important. While this is not accounted for in this analysis, a parameter extraction technique capable of producing accurate estimates of wire capacitance could certainly be used in place of, or in addition to, the model currently in place. The power dissipation on toggle could simply be increased by $\frac{1}{2}CV_{\mathrm{dd}}^2$, for each instrumented gate. Sources of power dissipation that are not covered by this model, or any activity-based model in general, include power dissipation caused by uneven path lengths; glitch power. This is typically minimized in the technology mapping and optimization phases by path length optimizations, but is still expected to be present and is a source of error in all activity-based energy models.

Theoretical Overhead Because the Wallace tree has $\mathrm{O}(N)$ space efficiency, each instrumented gate has a fixed-size activity detector, and the fraction of gates that must be instrumented for a given accuracy can be assumed to be roughly constant, the overhead in gate count, and therefore, roughly, FPGA area, is $\mathrm{O}(N)$. Equivalently, the FPGA area requirement of an instrumented model is the FPGA area requirement of an uninstrumented model times a constant factor.

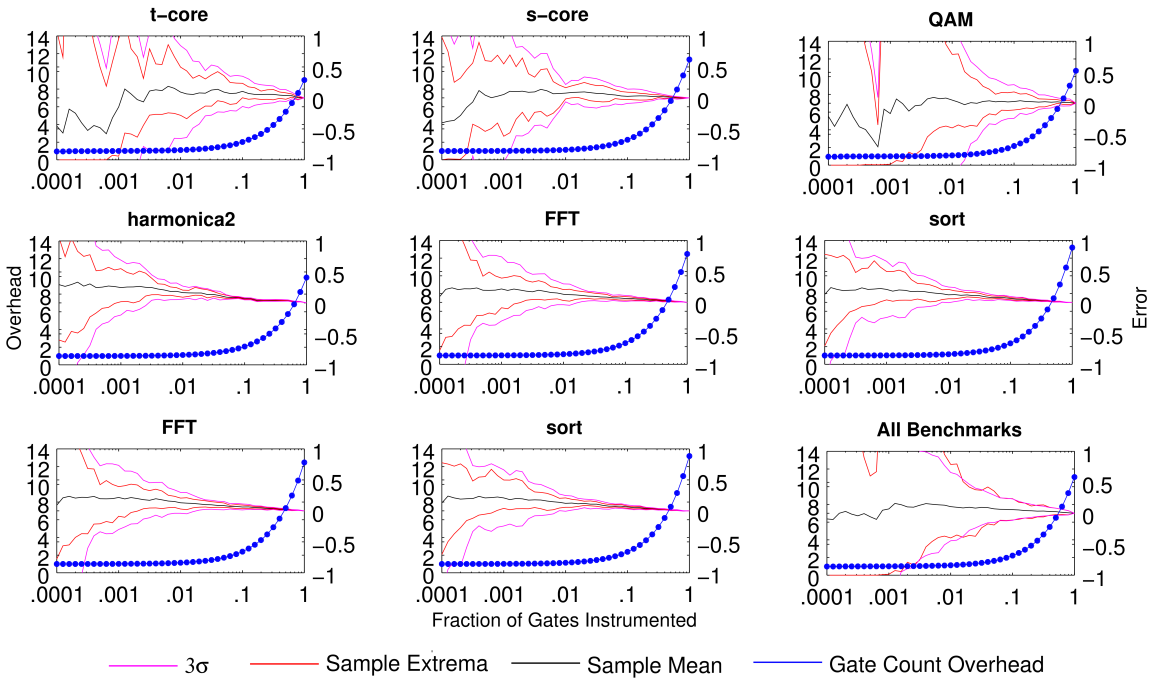


Figure 11: Gate count overhead and error for levels of instrumentation from one gate in ten thousand to all gates.

Accuracy vs. Overhead The power estimation error experienced in simulations using these static samples is roughly normally distributed, with a mean near 0 and a variance that decreases with increasing amounts of instrumentation, measured by the fraction of gates that are instrumented. For each benchmark, a sweep of the fraction of gates instrumented was performed, trying 11 different samples of each size in 41 steps spaced evenly along a logarithmic scale from 1 in ten-thousand of the gates present to all of the gates being instrumented. These were evaluated for accuracy with a brief, 10k-cycle simulation, and compared to the fully-instrumented example to quantify error.

The benchmarks used were:

- `t-core`– a minimalistic 16-bit accumulator architecture microcontroller intended for tightly-embedded applications.
- `Iqyax`– a single-issue pipelined RISC core, including branch prediction.
- `harmonica2`– [30] a GPGPU-like single-instruction, multiple-thread core, 4 lanes by

16 warps.

- `sort`– even-odd merge sort, 32 elements by 32 bits.
- `fft`– fully combinational fast Fourier transform, fixed point, 16 elements by 12 bits per element.
- `qam`– quadrature-amplitude modulator, DDS with 8-bit precision, 64-point constellation.

The results of this sweep are shown in Figure 11. Error is reported as a distribution, with both the minimum and maximum sample value reported as well as the values three standard deviations from the mean on both the upper and lower side. Since the results for static sampling fall on a normal distribution, these can be seen as the bounds of a 99.7% confidence interval on the range of the results.

Instrumenting 10% of gates, with an average expected error of less than 13%, the gate count overhead is approximately 2 times, while instrumenting all gates leads to no error and an overhead of 11 times. The overhead is estimated as number of gates in the output compared to the number of gates in the input. Given the nature of the power model implemented, this tends to under-estimate the overhead, but the trends remain accurate.

3.7 Conclusions

In this chapter, the CHDL core library and template library were described in detail, showing that it is possible, simply by creating a library to perform simulation and manipulation of gate-level logic circuits, to support domain-specific languages capable of describing hardware at the register transfer level. Extensions to the functionality of CHDL can be implemented either as libraries or as part of user programs themselves, enabling the implementation of highly-specific optimizations which may generate and even alter the in-memory netlist. In the next chapter, a specific accelerator architecture that has been implemented with CHDL is discussed, and the chapters following it continue to add layers of abstraction on top of CHDL.

CHAPTER IV

THE HARMONICA DATA PARALLEL CORE

4.1 Iqyax RISC Core

The CHDL hardware design environment described in the previous chapter was designed to enable the development of accelerators including instruction set processors. An early example of this is the Iqyax RISC core, which strives for compatibility with MIPS 1, but also provides the load linked/store conditional instructions from the MIPS 2 instruction set, enabling efficient implementation of synchronization primitives in multi-core designs. Iqyax provides optional support for a register scoreboard, MSHRs to enable support for non-blocking caches, and a branch target buffer to enable branch prediction.

Iqyax was created to provide a textbook single-issue pipelined core that could run multithreaded benchmark programs compiled by GCC on the CHDL component for the Structural Simulation Toolkit (SST), which is further explained in Section 3.6.1.3. Both Iqyax and the core described in the remainder of this chapter, Harmonica, use the memory request and response interface formats provided by the CHDL template library, which is translated by the CHDL SST component into a format compatible with SST memory system components. While the Iqyax RISC core provides configuration of performance/area trade-offs including support for non-blocking memory accesses, it does not provide the kind of high-throughput data parallel execution common in modern accelerator cores. This would require a fundamental change of architecture. The remainder of this chapter describes a family of such architectures as well as an implementation.

4.2 Introduction

Recently, there has been a re-emergence of research on moving processing to memory to reduce data movement energy, mitigate the impact of poor locality, and increase memory bandwidth available to computations. While research in the 1990s addressed the idea of placing processing in memory (PIM), the continued evolution of Moore's Law and related

architectural advances and market forces of the day precluded the need for such architectures. Further, the integration of logic in a DRAM process presented its own challenges. However, with the slowing down of Moore's Law, the emergence of 3D packaging provides a vehicle for integrating logic-optimized and DRAM-optimized dice thereby providing a means to sustain performance scaling by pushing back the memory bandwidth and power walls. However, placement of compute accelerators in memory is subject to distinct constraints and tradeoffs between power, performance, and area. Our goal is to design a SIMT architecture that can be tuned to match different 3D memory organizations.

In this chapter we define a family of lightweight single instruction multiple thread (SIMT) architectures referred to as the Heterogeneous Architecture Research Prototype (HARP). The HARP infrastructure defines a family of instruction set architectures parameterized by aspects such as the word size, the number of general-purpose and predicate registers, and the size and number of synchronously executed thread blocks. Their simplified design realizes high bandwidth, latency tolerance, low area, and low power. The execution model is a generalization of the bulk synchronous parallel (BSP) model supported by commodity general purpose GPUs and languages such as OpenCL and CUDA. Our refinements to the BSP model add support for dynamic parallelism and exceptions. An implementation of the HARP instruction sets named Harmonica has been produced using an open-source C++-based hardware design library called CHDL.

CHDL is used to produce both a simulation model and a gate-level netlist, using a 15nm standard cell library, for the Harmonica core design, which is configured in terms of these parameters. A system deploying one of these architectures in combination with a stacked DRAM could size the cores so they fit in a tiled multicore arrangement along with per-core caches and independent DRAM channels. In our analysis we run a set of analytics and data warehousing oriented benchmarks. A companion toolchain provides an assembler, linker, emulator and gate and cycle level simulators also using the same set of parameters. We use this environment to generate, explore, and evaluate SIMT accelerators for integration with 3D DRAM memories.

4.3 Background and Overview

By permitting some operations to be performed on data in-package, (1) the amount of data movement between the processor chip and the memory can be reduced, (2) energy efficiency improved, (3) use of pin bandwidth optimized, and (4) memory access latency substantially reduced, thereby mitigating the performance effects of poor reference locality. The past impediments to combining DRAM and logic in a single process are now precluded by the emergence of 3D stacked DRAM permitting logic-optimized dice and DRAM optimized dice to be integrated within a single package interconnected by through-silicon vias (TSVs).

This may now be considered to be a mature technology. Manufacturers of NAND flash memory have been using 3D die stacking for years and several 3D stacked DRAM technologies are now commercially available. Wide I/O-2 is designed for mobile platforms by stacking conventional DRAM on embedded processors (3D interface) while High Bandwidth Memory (HBM), which incorporates a stack of 4 DRAM dice and a single logic die, is designed for high performance processors [54]. In our work we utilize a third class of 3D memories based on the Hybrid Memory Cube from Micron [28], which is organized into many vertical channels called *vaults*.

4.3.1 Hybrid Memory Cube (HMC)

The Hybrid Memory Cube standard [15] defines an interface and a device architecture for 3D-stacked memory. It is composed of multiple stacked DRAM dice and a single base logic die interconnected with through silicon vias (TSVs). Each DRAM die is divided into partitions in a 2D grid and the corresponding partitions aligned vertically and connected with TSVs form a vertical DRAM channel called a vault. Each vault has an independent DRAM controller on the logic die; therefore each cycle, an access may be initiated in a ready bank in each vault.

These vault controllers are interconnected by an on-chip network consisting of two levels of crossbar switches to high speed external serial links. A typical HMC 2 device provides 8GB of memory organized as 32 vaults, with 4 layers and 4 banks per layer in each vault,

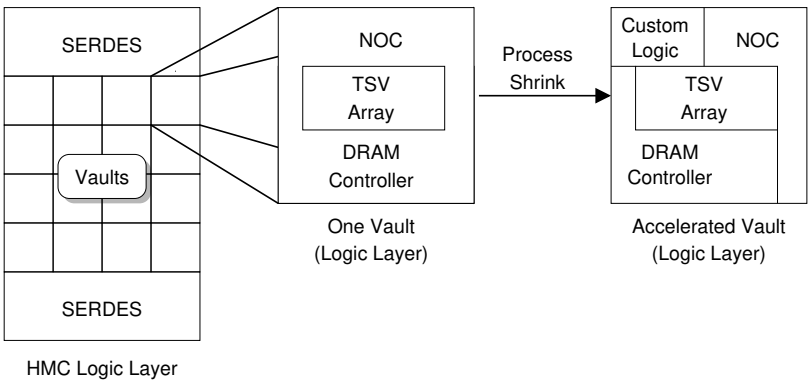


Figure 12: Hypothetical floor-plan of current-generation HMC compared to accelerated vault. At least 1.5mm^2 per vault is expected to be made available for accelerators by moving from a 28nm to a 15nm process.

or 512 total banks. The logic and memory dice can be fabricated in different process technologies. For example, DRAM dice could be 50nm and logic die is fabricated in 28nm [15]. Compared to HBM or Wide I/O-2, the HMC architecture provides highly parallel access to the memory optimized for random accesses; a large number of vaults with one channel per vault and multiple banks per vault, leading to an unprecedented level of parallelism in the memory system. These devices were designed to provide very high bandwidth for random accesses by exploiting this memory-level parallelism and the density of TSVs.

4.3.2 Architectural Constraints

In a system employing per-vault accelerators, as shown in Figure 12 the accelerator footprint must fit within the vault footprint, including the sizes of the vault controller and router for the on-chip network. If we assume that with the current 28nm process technology, the entire 68mm^2 HMC die area is consumed by the vault memory controller and network interface, then we can estimate that by shrinking to a more modern 15nm process, 1.5mm^2 per vault becomes available for accelerators. Additionally, the proximity of compute and DRAM places tight constraints on the thermal design power (TDP) of the logic die, estimated in some studies at 10W [36]. The near-memory accelerators must also be able to effectively utilize the available memory bandwidth especially for those applications plagued by poor spatial and temporal locality.

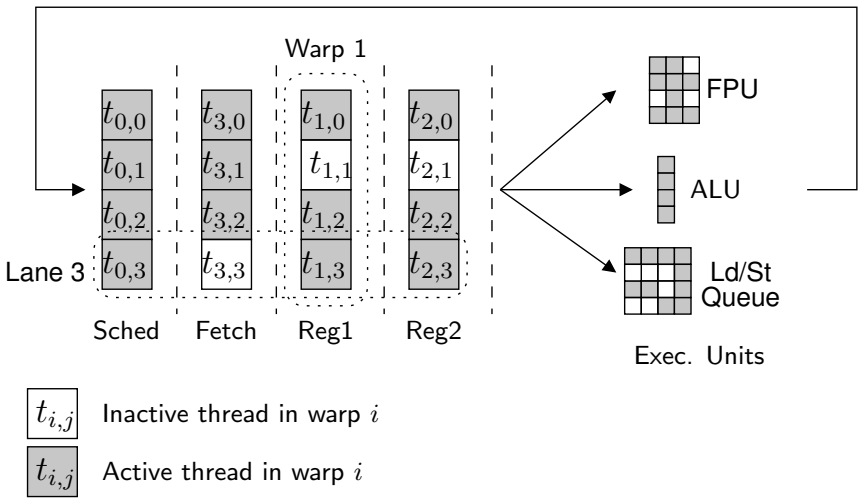


Figure 13: Simplified diagram of an SIMT pipeline.

4.4 The HARP Architectures

To conform to the power constraints, area constraints and memory bandwidth opportunity within a vault, we selected a single instruction multiple thread (SIMT) architecture, parameterized to fit within the constrained environment of the vault-level accelerator in the logic layer. The parameters that can be adjusted include the size of the general-purpose and predicate register files, the width of the parallel execution units (number of threads per *warp*), and the number of thread blocks or warps. In the following sections we describe a family of such compute architectures, the toolchain for generating architectural instances, and unique features that were adopted to meet the characteristics of emerging applications. Due to the similarities in their SIMT designs, the terminology used in this section is the same terminology used by NVIDIA when discussing their line of SIMT GPGPUs.

The parameterized family of instruction set architectures (ISAs) we have developed was originally developed as part of a project to create a Heterogeneous Architecture Research Prototype, and is therefore referred to as the family of HARP instruction sets. The HARP ISAs are SIMT architectures that can be described by parameters for the word size, instruction encoding, the number of general-purpose and predicate registers, and the two-dimensional thread count; the number of warps and *lane*s, or threads per warp. Figure 13 illustrates the way in which warps and lanes are grouped logically in a simplified pipelined

HARP implementation.

The Heterogeneous Architecture Research Prototype (HARP) defines a space of instruction set architectures that implement a single instruction multiple thread (SIMT) programming and execution model. HARP is composed of two main elements. The first is a parameterized family of instruction set architectures (ISAs) for the SIMT model. An instance of a HARP ISA can be described by parameters for (1) the word size, which must be a multiple of eight bits, (2) instruction encoding, (3) the number of general-purpose and predicate registers, (4) the size of a *warp* which is a synchronously executed block of threads and therefore determines the width of the parallel execution units, and (5) the number of warps supported by an architecture implementation.

4.4.1 Execution Model

A simplified HARP core pipeline is illustrated in Figure 13. The ISA enables the execution of multithreaded kernels where each kernel is composed of a set of synchronously executed blocks of threads referred to as *warps*. Each warp is composed of a number of threads that execute in lock step and where the number of threads in a warp is equal to number of lanes in an core. In Figure 13, threads are given numbers $t_{i,j}$ where i is the warp ID and j is the position of the thread within the warp. Since kernels executing on SIMT cores must span multiple warps in order to achieve high performance and latency tolerance, a hardware barrier mechanism is provided to synchronize between these warps. The availability of a large number of warps provides for effective latency hiding, while the concurrent execution of threads within warps can generate significant memory traffic that can be satisfied with the high internal memory bandwidth and concurrency within a vault.

4.4.2 Parameterization

HARP is distinguished by its parameterizability, and implementations of HARP can take on a wide range of performance and energy characteristics, as seen in the evaluation of our implementation called Harmonica in Section 4.7.

A succinct string representation is used to identify HARP architectures. A `4w32/32/8/4` architecture, for example, has four byte (32-bit) machine words, word-oriented instruction

Table 7: Selected instructions from the HARP instruction sets.

Instruction	Operands	Description
`clone %rD`	`%rD`: dest. lane	Copy register state to destination thread within warp.
`jalis %rL, %rN, DEST`	`%rL`: link register `%rN`: number of lanes `DEST`: dest. label	Jump-and-link-and-spawn; call multithreaded function.
`jmprt %rD`	`%rD`: dest. address	Jump-and-terminate; return from multithreaded function.
`@pX ? jmpi DEST`	`@pX`: predicate register `DEST`: dest. label	Branches are implemented as predicated jumps.
`split @pX`	`@pX`: predicate register	Handle divergent control flow: diverge.
`join`	–	Handle divergent control flow: reconverge.
`bar %rID, %rN`	`%rID`: Barrier ID. `%rN`: Warps count.	Synchronize warps with barrier.

encodings, 32 general-purpose and predicate registers, eight lanes per warp and supports four warps, for a total of 32 hardware threads. Valid HARP architectures have either word-encoded (`w`) or byte-encoded (`b`) instructions and between 1 and 256 predicate registers, registers, warps, and lanes. This format is used by the open-source utility *Harptool*, a monolithic assembler, linker, and emulator utility written in C++, to specify HARP architectures.

4.4.3 Instruction Set Features

In addition to its parameterization the HARP instruction set provides several unique features in order to implement SIMT semantics and efficiently support application behaviors. These are (1) full predication, (2) `split` and `join` instructions for handling divergent control flow within warps, (3) a barrier instruction and dedicated functional unit, and (4) warp-level exceptions. Some of the more interesting and relevant instructions are listed in Figure 7.

Predication SIMT cores keep threads within a warp at the same program counter in

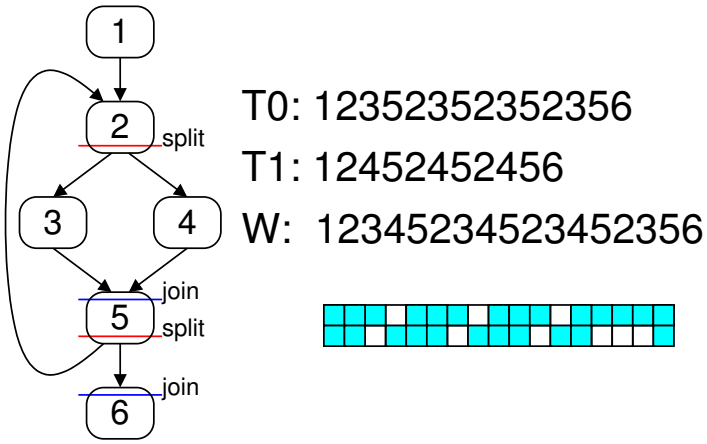


Figure 14: Split and join instructions are used to manage control flow divergence.

order to reduce pipeline control overhead. Throughput is maximized by keeping control flow paths in all threads within a warp the same. When the control flow of threads within a warp diverge, such as when a branch in one lane's thread is taken while a branch in another lane's thread is not taken, a subset of the threads are masked out and not executed; these are shown as the white squares in Figure 13. Two mechanisms are used to serialize the instruction stream between divergent threads. The first, predication, simply uses the value of a 1-bit register to determine whether an instruction will execute. This way, simple `if/then/else` types of statements can be translated to simple assembly code. In HARP, for example, the C-like pseudocode "`if (%r1) %r2++ else %r2--;`" becomes:

```
      rtop @p0, %r1     // reg. to predicate
@p0 ? addi %r2, %r2, #1 // if @p0, add
      notp @p0, @p0     // complement @p0
@p0 ? subi %r2, %r2, #1 // if @p0, subtract
```

A stack-based algorithm, discussed in the next section, is used in cases where it is not possible to use predication e.g. due to looping behavior, or cases where it is not desirable to use predication for performance reasons. All predicated instructions encountered incur the full instruction execution penalty in our implementation, and this is even true when the predicated instruction is not being executed by a single lane.

Split/Join The immediate post dominator algorithm (IPDOM) first described by NVIDIA

in [39] is a straightforward way to handle the serialization of control flow in cases where predication is not appropriate. IPDOM works by maintaining a per-warp mask of running threads. When a divergent branch occurs, IPDOM pushes a copy of the current thread mask to a special hardware stack, as well as a mask for the not-taken direction of the branch, then sets the mask to contain only the threads that took the branch. At a compiler-selected reconvergence point, which for true IPDOM is the first instruction of the immediate post-dominator of the taken and not-taken paths, the next entry is popped off of the stack and the mask and PC set accordingly.

In the HARP instruction sets, selection of divergence and convergence points is performed explicitly with a pair of instructions. The `split` instruction indicates the point of (possible) divergence, and the `join` instruction indicates the reconvergence point. The example from the previous section, "`if (%r1) %r2++ else %r2--;`", is handled using a divergent branch instead of predication, becoming:

```
              rtop @p0, %r1
        @p0 ? split
        @p0 ? jmp then
              subi %r2, %r2, #1
              jmp cont
  else:       addi %r2, %r2, #1
  cont:       join
```

when `split` and `join` are used.

Barrier One of the principles of the bulk synchronous parallel (BSP) execution model used with SIMT processors is that synchronization using barriers is inexpensive. HARP provides inter-warp barrier synchronization through a single instruction. Since threads within warps execute in lock-step, barrier synchronization at every point within a warp is implicit. The barrier instruction, `bar`, takes as operands the number of warps participating in the barrier and the barrier ID. Any number of barrier table entries may be provided by the implementation, allowing for multiple simultaneous barriers to be active. No warp that

has executed a `bar` will execute until the number of threads provided in the count operand have also executed `bar` instructions with the same barrier ID. The way this is implemented in hardware is implementation specific, but in the implementation discussed in the next section, involves table structures containing warp information in a special functional unit.

4.4.4 Kernel Launch Model

HARP is designed to run independently, minimizing the need for host processor interaction, and as such the HARP instruction sets include support for spawning new warps, and spawning threads within warps. For the benchmarks used in our evaluation, the entire launch and run of the kernel is performed by a single Harmonica core on data that is already present in memory without any interaction with a host processor. Because of these features, HARP can be said to offer a generalization of the BSP execution model and to provide instruction set level hooks for dynamically spawning threads.

When execution on a HARP processor begins, a single warp is running. Executing the `wspawn` instruction spawns a single warp, and copies a given register value into a register of the new warp. This can be used to pass in a pointer to warp parameters and bootstrap the warp spawning process. Within warps, a `clone` instruction is provided to copy register state, and jump-and-link-and-spawn and jump-register-and-terminate instructions are provided to initiate and terminate multi-lane execution, respectively. This allows the parallel portion of execution within a warp to be treated as a simple procedure call.

The `wspawn`, `clone`, and thread control instructions are not privileged instructions and can be executed by user code, enabling the dynamic parallelism as described in [29] in implementations. These features are made available to the programmer through the HARP runtime library, a lightweight set of functions exposing the ability to spawn warps and call multi-lane functions to the programmer. The purpose of the HARP runtime library is to provide a C API encapsulating the HARP instruction set's thread management features, including control flow divergence and the spawning of threads.

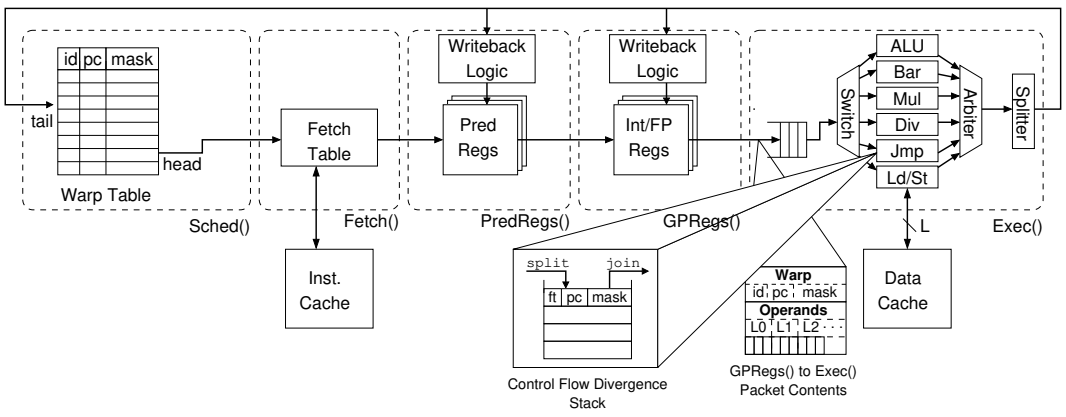


Figure 15: The Harmonica pipeline.

4.4.5 Exception Support

Rare among SIMT architectures and accelerator-oriented architectures is HARP's ability to handle exceptions through a simple hardware-provided mechanism. Exception handlers in HARP architectures run only in lane 0, and are called in response to events such as page faults and external hardware interrupts. These are implemented in our Harptool emulator but are not supported by the current version of our implementation discussed in the next section, nor are they evaluated in Section 4.7.

4.5 The Harmonica Microarchitecture

Harmonica is an implementation of the HARP family of SIMT architectures and ISAs, implemented using CHDL, a hardware description library for C++, designed to enable the creation of hardware using highly templated generators. Harmonica is a small low-power design intended to meet the unique challenges of near-memory computing; a single-issue implementation of a subset of the HARP parameter space. Using the notation described in Section 4.4.2, `XwN/N/W/L` architectures are supported by Harmonica, requiring the use of word-oriented encoding, and that the number of general-purpose and predicate registers be the same. This core design achieves the requirements enumerated in Section 4.3.2 through a variety of techniques:

- Small area and low power through a simple, customizable design that may, when

appropriate, eschew energy-and-area demanding features such as floating point. Our analysis focuses on fixed point Harmonica cores.

- Tolerance to memory latency through the exploitation of thread-level parallelism. A surplus of warps is maintained that can be executed while other warps await the completion of memory operations.

- The highest possible utilization of memory bandwidth through the use of a multi-lane SIMT design. Each cycle, multiple memory requests can be made simultaneously. Note that parameterization and tool chain makes it possible to generate a single lane Harmonica implementation.

4.5.1 Pipeline Detail

The design of Harmonica is structured as a modular pipeline, illustrated in Figure 15, in which warps flow through like packets in a network. The semantics of the architecture allow these warps to be completely independent, executing a single instruction on each active lane in each trip through the pipeline.

Scheduler The scheduler is a simple queue of warp states. Since this is a single issue design, a single warp per cycle proceeds from the scheduler to the fetch stage. Warps are scheduled in a FIFO manner, leading to a simple circular buffer design for the warp table, which dominates the area for this stage but is relatively small given typical numbers of threads per warp.

Instruction Fetch and Register Access Harmonica uses the CHDL memory interface for both the link to the data cache and instruction cache. This is an arbitrary-width request/response interface with arbitrary word size, which matches requests and responses using tags. Support for atomic operations is provided by the memory interface format through load-linked/store-conditional operations. The memory interface descriptions are provided as part of the CHDL template library making them the standard memory interface used by CHDL designs.

In Harmonica, the warp ID is used as the tag value for both loads and instruction fetches, and the warp state is stored in a table while the operation completes. Warps leave the fetch

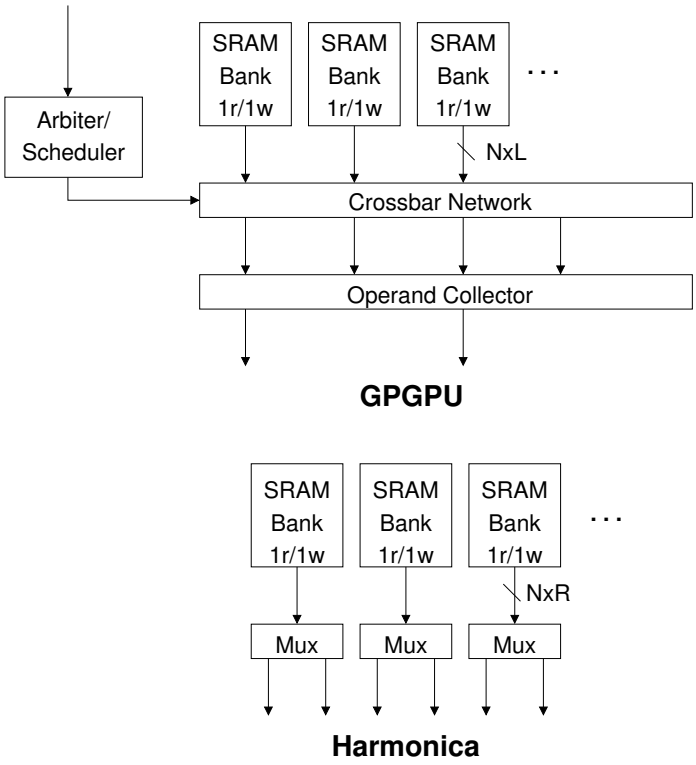


Figure 16: The Harmonica register file design is somewhat simplified compared to that of commercially-available SIMT GPUs.

stage in the order in which their responses return from the instruction cache, and continue to the two register read pipeline stages. Like the scheduler stage, this stage is dominated in area and gate count by this warp table structure.

Register Files The use of a single-issue design enables a simplification of the register file, shown in Figure 16, as compared to comparable high-performance GPU designs. Due to the highly-parallel design, typical GPUs require multi-ported register files in order to achieve design performance. Because of the large area penalty of multi-ported memory structures, these are implemented using banking schemes and a large crossbar network. Because the Harmonica design issues only a single warp at a time, its register file design can be greatly simplified, using an SRAM structure with a single read port and a single write port, but foregoing both the bank steering logic and the operand crossbar, saving area and energy at the cost of throughput.

Execute Once the warps have gathered their operands from the register stages, execution can be performed. Harmonica dispatches warps to functional units by opcode with a simple router. Warps which have finished executing their instructions are returned to the schedule stage, with their program counters and thread masks updated as required by the instruction executed. Concurrently with this their results are written back to the register file.

Because warps share no state, execution units are independent, with an interface consisting entirely of an input and output with a single `ready` bit for backpressure; a FIFO interface. This simplifies the implementation and inclusion of functional units, and as a result, enables customization of Harmonica SMs. For example, all functional units are optional, and some, such as the single-cycle multiply and divide, can be replaced with lower cost bit-serial multi-cycle versions to trade performance, and even, in the case of floating point units, accuracy, for size and power.

Among the functional units present in the execute stage is a simple load/store unit that, much like the fetch unit, sends a request using the CHDL memory request/response interface for each operation, and stores the warp state in a table, with the requirement that there is a memory interface per lane and for loads, all lanes' responses must return before the warp is allowed to advance. This is intended to interface with a small, multi-ported cache with one logical port per SIMT lane.

4.6 Tool Flow

A set of tools and benchmarks has been developed around the HARP instruction set architectures and Harmonica implementation, providing a complete flow allowing benchmarks to be compiled to machine code representations and the Harmonica core itself to be elaborated into synthesizable Verilog netlists and simulated or run by an FPGA. Figure 17 provides an abbreviated view of this tool flow, including a 2-stage compilation flow for C code allowing it to run on HARP cores, the *harptool* instruction set multitool providing an assembler, linker, and emulator, and a variety of ways to execute applications on simulated or synthesized Harmonica hardware. At the start of both of these flows is a HARP configuration string, as

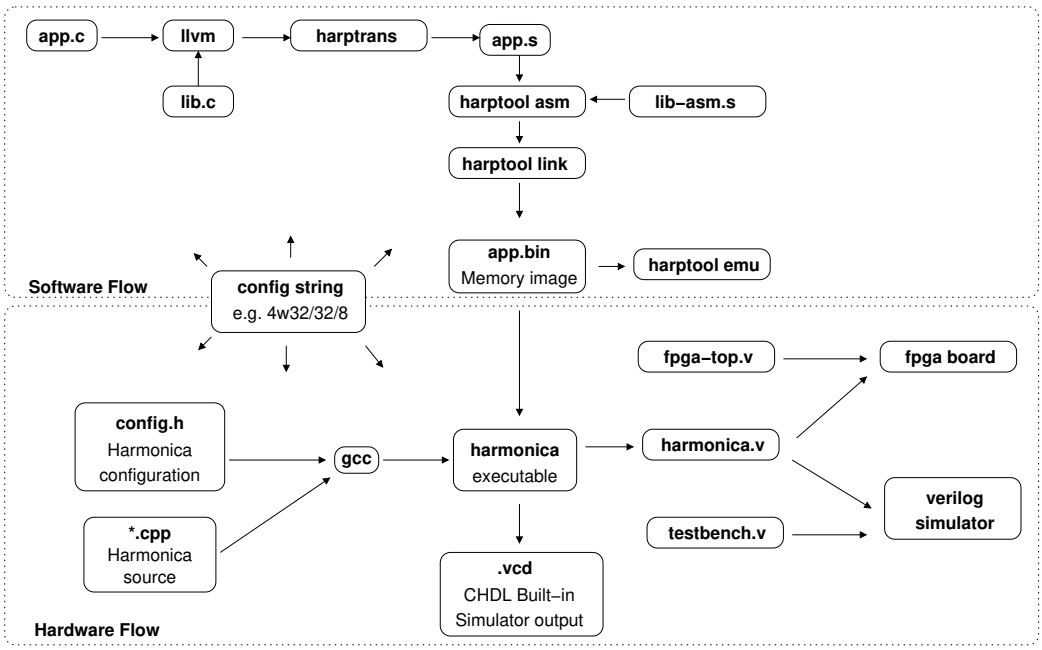


Figure 17: Complete software and hardware flows, including examples of evaluation by CHDL simulation, Verilog simulation, and execution in FPGA.

described in Section 4.4.2. This provides the parameters needed to select the appropriate register file dimensions, instruction set representation, and pre-defined constant values in the runtime library.

4.6.1 Software Flow

The HARP software flow is built around a lightweight runtime library, the LLVM compiler framework, a translation utility called Harptrans, and the HARP architecture assembler, linker, and emulator called Harptool. An LLVM-based compiler backend has also been developed [26], but since it was not compatible with the full suite of benchmark applications across all possible core parameters it was not used at all in the evaluation of the Harmonica core to avoid obscuring trends in the results.

- Simple runtime library

While Harmonica and the HARP architectures were created with accelerator architectures in mind, it is not the purpose of the evaluation flow to solve such problems as the spawning of kernels on accelerator cores or communication between main and accelerator

Table 8: Some of the calls in the HARP runtime library.

Function	Description
`call_par()`	Multi-lane function call.
`spawn_warp()`	Spawn a warp.
`split()`	Split instruction. Start divergent region.
`join()`	Join instruction. End divergent region.
`barrier()`	Barrier instruction.
`lane_or()`	Lane-reduce logical or.
`lane_and()`	Lane-reduce logical and.
`lane_min()`	Lane-reduce min.
`lane_max()`	Lane-reduce max.

cores. Instead, the focus of this tool flow is on simply running programs on the Harmonica cores itself, providing mechanisms for such events as control flow divergence. The runtime library, therefore, assumes that the memory has already been loaded with an executable image and any necessary data, with execution starting at address zero. The library provides enough bootstrap code to begin execution at address zero. At program entry, there is only a single thread running in a single warp.

Some of the functions in the runtime library are listed in Table 8. Many of them simply provide access to the parallelism-oriented aspects of the Harmonica instruction set, such as the barrier, split, and join instructions. Others, such as `lane_or()` and `lane_max()` perform parallel operations involving inter-lane communication that would be difficult to perform directly using a high-level language. The runtime instructions highlight an aspect of the programming interface presented for HARP. The `split()` and `join()` operations are explicit. There is no automatic insertion of these instructions when this approach is used, and this is not supported by the LLVM-to-HARP translator Harptrans, requiring low-level control flow decisions usually automated to be made by the programmer.

- LLVM-to-HARP translator

Harptrans provides rudimentary support for converting LLVM intermediate representation to HARP instructions, including register allocation. This includes a greedy linear

sweep algorithm for register allocation and a basic instruction selection scheme. The table-based generator employed by LLVM is more powerful and allows a wider range of keyhole optimizations and future work will deprecate Harptrans for the compiler infrastructure described in [26].

- Harptool assembler, linker, and emulator

A single source code base is used to provide all of the fundamental instruction set adjacent software related to HARP, including an assembler, linker, and emulator. This program, called Harptool acts as a reference implementation of the instruction set, providing a valuable debugging tool both for implementations of the instruction set, e.g. Harmonica, and software targeting the instruction set, e.g. Harptrans and the HARP LLVM back-end. The harptool emulator provides a fast execution environment for Harmonica-compatible binaries and has been an invaluable tool in the evaluation of the Harmonica core design.

4.6.2 Hardware Flow

The hardware flow is centered around the CHDL core library and template library. The C++ source code for Harmonica is compiled with a set of parameters encoding the configuration string; including the register counts, number of threads per warps and maximum number of supported warps. This is translated at compile time into an executable generator that produces a synthesizable Verilog netlist which may be simulated, executed on an FPGA, or technology mapped to hardware. The area and energy estimates later in this chapter were produced by analysis of the output of CHDL's built-in technology mapping algorithm.

4.7 Performance Analysis

We have developed a set of open source tools for exploring the HARP design space. Figure 18 shows that the tool flow is bifurcated into two paths; along the bottom is the software path used to compile benchmarks, and along the top is the hardware toolchain used to produce and evaluate Harmonica cores at the gate-level. The software flow includes the benchmarks themselves, their shared runtime library, an LLVM-based compilation infrastructure, and

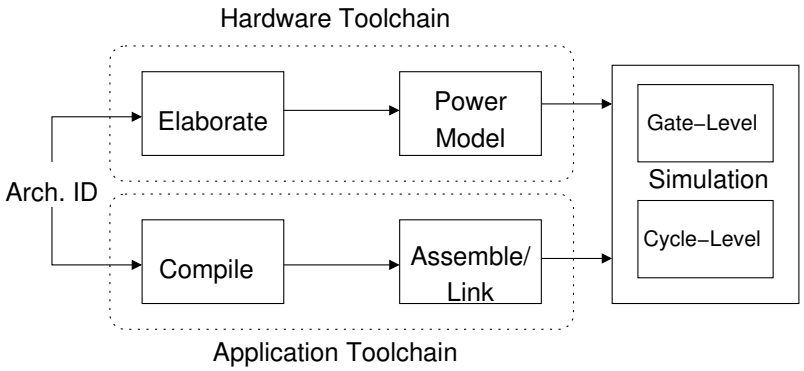


Figure 18: Simplified diagram of tool flow used to evaluate Harmonica design.

the aforementioned Harptool, which functions both as an assembler/linker program, and as the emulation front-end for our cycle-level simulator. The details of the subset of the hardware simulation flow used to produce each category of results are presented along with those results, but generally a cycle-accurate execute-at-execute Harmonica simulator was produced using the emulator provided in Harptool, in order to quickly produce results. For detailed modeling of power consumption, workloads long enough to get to steady state of the first application phase were run on a gate-level simulation of the Harmonica design synthesized for a 15nm standard cell library instrumented with a counter tree for energy consumption estimation.

Our analysis is performed in the context of a single accelerated vault, considering kernels that operate on data contained entirely within the local vault. While applications operating on larger datasets may experience slow-down due to contention in the on-chip network, there will always be less traffic than the same application would experience running on an off-chip processor. By focusing on the single-vault case, we provide an evaluation of the Harmonica core design disentangled from the orthogonal problems of network-on-chip design and execution models for many-core SIMT processors.

4.7.1 Benchmarks

Workloads were chosen with an emphasis on memory-bound analytics and database applications that must traverse large in-memory data structures. It was felt both that these kinds of workloads will become increasingly important as the sizes of data sets grow, and

Table 9: The benchmarks used in our evaluation of the Harmonica core design. Cited papers are sources for algorithms used.

Name	Description	Data	Size
`bfs`	Breadth-first search. [38]	Pennsylvania road network.	1090920 nodes, 3083796 edges
`radix`	Radix sort. [50]	Random 32-bit integers.	1048576 elements
`binsearch`	Binary search.	Random 32-bit integers.	1048576 elements, 1048576 lookups
`hashtable`	Hash table lookup.	Random 32-bit integers.	1048576 elements, 1048576 lookups
`vecsum`	Sum integer vector.	Random 32-bit integers.	16777216 elements
`select`	Select from table.	Random Boolean values.	1048576 elements, 1037940 matching rows

that these are the applications for which in-memory processing provides the most benefit. The workloads themselves, listed in Figure 9, are kernels representing common functions that could reasonably be expected to be offloaded by an application to the memory system. These are all fixed-point benchmarks as well. A floating point unit is supported as an optional add-on for Harmonica cores, but is beyond the scope of this work, in which the workloads do not benefit from fast floating point computation.

4.7.2 Results

The following represents an analysis of the execution of the benchmarks on an Harmonica core that supports integer operations.

Area A gate-level representation of a range of Harmonica designs has been produced using the technology mapping algorithm provided by CHDL, including a collection of gates from the Open Cell Library for the FreePDK15 15nm PDK [37] and a set of descriptions of SRAM banks. SRAM area was estimated at a bank granularity using a conservative projection of SRAM array area based on values provided by Cacti [55] for the 32nm node, and logic area was estimated as $\frac{\sum_{g \in \text{Gates}} A_g}{0.90}$, the area required for a placement solution with 90% area utilization.

We study a wide range of Harmonica implementations. As seen in Figure 19, many

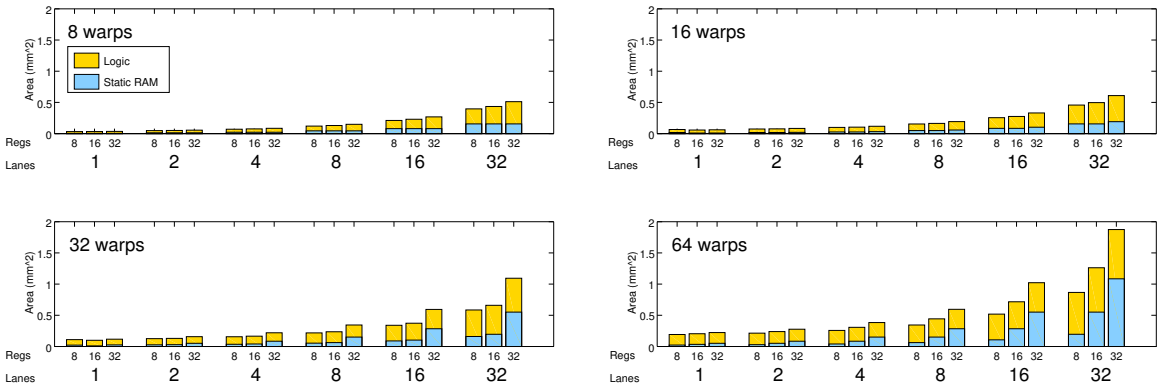


Figure 19: Area estimates for Harmonica cores, not including L1 caches, based on synthesis.

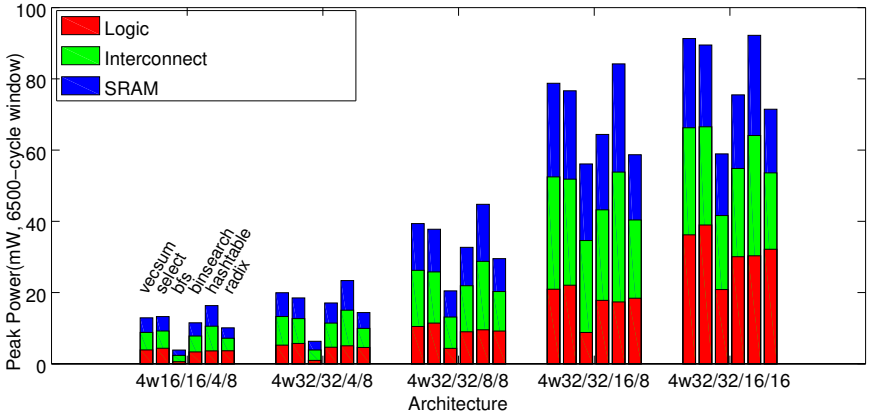


Figure 20: Average power for 1ms of each application running at 650MHz.

implementations occupy less than a square millimeter; small enough to fit with a 1kB instruction cache and a 32kB data cache in the empty space freed up by shrinking the existing Hybrid Memory Cube logic layer design from 28nm to 15nm. Note that this assessment does not assume any available area in the current HMC logic layer design, a conservative assumption since the HMC vault pitch is likely determined by the DRAM banks, the current generation may have free area in the logic layer already. There is likely area to accommodate multiple Harmonica cores or larger Harmonica cores in a vault.

Power As a complement to CHDL's technology mapping functionality, a power emulation library has been built that allows the generation, as CHDL hardware, of activity-based energy models. Using the extracted capacitances, and internal switching energies provided

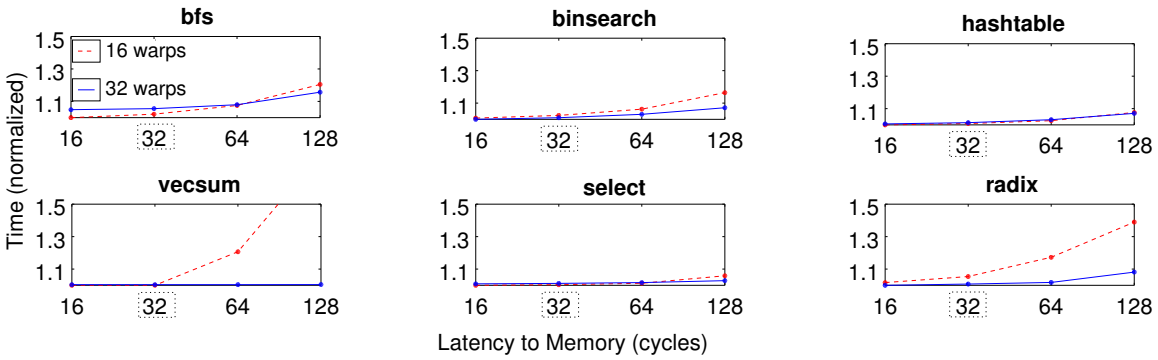


Figure 21: Relative application performance in simulation as a function of memory access latency for `4w32/32/16` architectures with 16 or 32 warps. 32 cycles highlighted as it represents the most realistic latency estimate for local vault accesses.

along with the Open Cell Library, a set of netlists augmented with energy models was created that could be run through a gate-level simulator. These were simulated at the gate level for long enough to reach steady-state in our set of applications. This produced a set of logic power traces in terms of energy per cycle and SRAM access counts. These were multiplied by Cacti-derived access energy estimates and added to the logic power to produce the final results, which are expressed in milliwatts, assuming a 650MHz clock rate, which is according to [15] the rate at which at least some logic in the HMC operates.

Figure 20 shows the breakdown for power consumption for 1ms of operation at steady state for each of our six applications. Even the largest cores simulated, 16 lanes wide with 16 warps, operate well within the 300mW per core (1 core for each of 32 vaults, approximately 10 watts total power) limit. Of note is the fact that one of the applications, breadth-first search, consistently expends less power than the remaining five. This is because the irregular nature of breadth-first search leads it to have higher than average control flow divergence and warp load imbalance, limiting the average number of active SIMT lanes and pipeline stages.

Latency Tolerance To measure latency tolerance, we used a cycle accurate simulator built around the Harptool emulation library. This includes a highly-banked 32kB 4-way set associative cache for data and a 32kB 4-way set associative cache for instructions. For these results, no attempt was made to optimize the data cache address mapping to improve

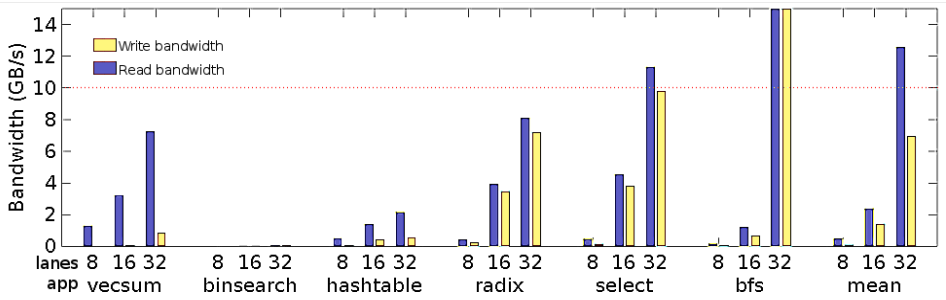


Figure 22: Bandwidth as a function of application and Harmonica core size. The number of warps and the number of lanes are the same in this case, and the number of GPRs is held at 32.

hit rates, and hit rates were typically quite low in the D-cache. Meanwhile, all kernels fit entirely into the I-cache.

The DRAM timing model used to produce the latency tolerance results, summarized in Figure 21, is a simple fixed-latency model, since the purpose is observing limits on application performance caused only by limits in latency, without consideration for bandwidth or the address dependence of latency. Bandwidth is given a complete treatment later in this section and the impact of vault bank conflicts and caching is examined in Section 4.7.3. The cores studied all had 32 general-purpose registers per thread and 32 SIMT execution lanes, though the values of these had little impact on relative performance.

Latency tolerance in SIMT cores in general and Harmonica in particular is achieved by having a large number of available warps that can be scheduled, so that a large number of memory operations can be in flight simultaneously, taking advantage of high-bandwidth, high-latency memory like the DRAM used in our HMC-based vaults. The number of available warps is the most obvious factor impacting a Harmonica core's tolerance to memory latency, and as Figure 21 shows, increasing the number of warps effectively increases the tolerance to memory latency. For the applications studied, even the 16-warp Harmonica core is capable of handling the estimated 32-cycle latency of the HMC vault DRAM with negligible performance degradation, and a 32-warp version is capable of handling four times that latency.

Bandwidth Demand Figure 22 shows the bandwidth achieved by the Harmonica core

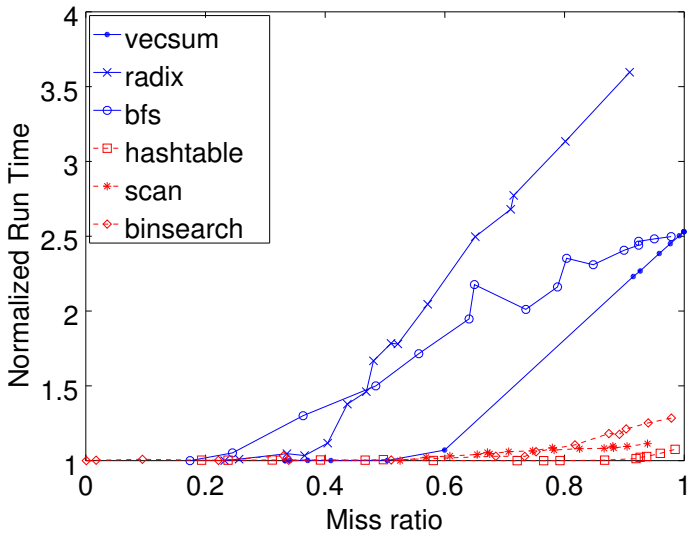


Figure 23: Plot of miss ratio vs. slowdown for various data cache associativities and capacities from 64 bytes to 16 kilobytes, simulated with a realistic vault model.

for various design sizes on our applications. For this simulation, in order to evaluate the peak bandwidth demand, the latency of a DRAM access was fixed at 32 cycles and the bandwidth was left unbounded, with no bank or channel model in place. In this simulation, the Harmonica core with 32 lanes and 32 warps running at 650MHz was able to achieve quite high utilization of bandwidth; greater than two thirds of the DRAM vault's available 10GB/s on 4 of the 6 applications. The remaining two, hashtable and binsearch, despite their relatively random access patterns to their data stores, use quite a few thread-local variables and thus achieve surprisingly high cache hit rates, leading to lower-than-expected bandwidth utilization.

4.7.3 Impact of Cache

Given the small size of the benchmark kernel code, less than 5kB on average and 10kB maximum (for breadth-first search), it is unsurprising that the only instruction cache misses in our evaluation are compulsory misses; the kernels all fit in our 32kB I-cache. While more complex applications may necessitate future exploration of instruction cache designs for SIMT cores, the looping nature of most programs and the fact that each thread in a low-divergence application only performs $\frac{1}{W}$, where W is the number of warps per thread,

instruction cache accesses per instruction, de-emphasizes instruction caches in SIMT accelerator design.

Despite the cores' tolerance to memory latency, the performance of Harmonica cores *is* dependent on the presence and performance of the data cache. This effect is entirely due to the cache's ability to reduce DRAM queuing latency caused by bank conflicts. Figure 23 shows, for a 32-GPR, 16-lane, 32-warp Harmonica core, the impact of cache miss rate on application performance. The cache sweep was performed using an accurate vault performance model, thereby emphasizing the effects of bank conflicts and TSV bandwidth limits.

4.8 Context

The use of multithreading for memory latency tolerance and simplified datapath control can be traced back to pioneering "barrel processors" such as the Tera MTA. This technique was notably adopted in commercial SIMT architectures first described in documentation from NVIDIA alongside a thread-based vector processing approach [39].

An interest in processing-in-memory in the 1990s led to work including [23] and [41], which ultimately never led to the production of a PIM product. The development in 3D die stacking has led to renewed interest, and near-memory accelerators for a diverse range of applications including general-purpose computing [58], graph processing [1] [59], sparse matrix operations [49], and machine learning [31]. SIMT and other GPU-like processors have been discussed in the near-memory role, including [58], [43], and [45].

Several open source GPUs have been developed for various applications, including FlexGrip [2] and Nyuzi [9]. These processors are similar to Harmonica in that they are, to some extent, parameterizable, but this parameterization does not extend to instruction set aspects such as the number of general-purpose registers. These designs are not specifically targeted toward in-memory applications, and have not been evaluated in detail for physical or performance characteristics that would qualify them as suitable for the study of PIM applications. To the authors' knowledge this is the first work presenting an in-depth gate-level evaluation of a SIMT core performed entirely using open-source tools and designs and

the first low-level evaluation of a SIMT core for near-memory computing.

Our work with the open source CHDL hardware description environment parallels recent work on the Chisel open source hardware description language and framework [4]. Much like CHDL, Chisel provides an environment for writing generators for hardware using a general-purpose programming language; in the case of Chisel this language is Scala instead of C++. There is quite a bit of work now on accelerator architectures in general, and fixed-function accelerators in particular have received newfound attention as a response to near-memory computing and dark silicon, with tools like Aladdin [51] providing ways to quickly analyze sets of accelerators without performing, as we have for Harmonica, low-level hardware design. Reconfigurable accelerators have also made a comeback in the domain of near-memory processing, with architectures like HRL [22] providing reconfigurable fabrics in the logic layer. Accelerators can be produced for these by a variety of methods, including automatic generation from software source code as demonstrated with the Delite HDL [32]. HARP is described as a customizable architecture in the sense that it is parametric. While this is true, it differs significantly from the kind of low-level architecture generation that is performed in work such as [13].

4.9 Conclusion

A Harmonica core with 32 lanes and 32 warps fits in the logic layer of a Hybrid Memory Cube style vault, can nearly saturate the available bandwidth for many applications while tolerating the latency of directly accessing DRAM, and assuming that trends from our analysis of smaller cores hold, fits within the 10 watt thermally-imposed power constraint of such near-memory cores. As Hybrid Memory Cube failed to capture market share and has been displaced by high-density 3D-stacked DRAMs such as HBM lacking a layer implemented in a logic process, the direction forward is re-evaluating such a core in the context of silicon or ceramic interposers or PCB technology and larger, less random access oriented DRAM banks. Since the Harmonica core design is intended to maximize throughput and bandwidth, it is likely to perform well no matter what bonding technology is used to deliver that bandwidth to the core, though further work is needed to evaluate that quantitatively.

CHAPTER V

GUARDED ATOMIC ACTIONS IN CHDL

One of the primary advantages of CHDL as an environment for implementing digital hardware is its extensibility through the use of C++ programming language features. Since C++ is a general-purpose programming language with features for making multiple paradigms realizable through features such as operator overloading and template metaprogramming, while on the surface CHDL is a system for implementing hardware generators, it is possible to piggyback other paradigms on top of CHDL. This chapter explores a specific example of this kind of use of CHDL, an implementation of guarded atomic actions (GAA) in CHDL, implemented as a C++ library and allowing the use of existing CHDL data types, including much of the CHDL template library. The ability to compose library functions in CHDL across multiple paradigms enables design reuse and repurposing far beyond the ability to wire together simple modules.

Guarded atomic actions, of which the Bluespec implementation [40] hardware description languages are the only popular embodiment, is a paradigm for expressing digital logic as a set of registers and rules; collections of register assignments invoked, or *fired*, when a set of associated guard predicates are true. These sets of registers and rules are combined together into *module*s, which can themselves contain instances of other modules. Each module exports an interface of actions and values. Values may be read at any time and represent the module state, and actions must be invoked by rules and include additional guard predicates that may block the invoking rule from firing. The GAA paradigm provides both a guarantee of fairness; all rules that may fire will fire, and a guarantee of atomicity; no two rules affecting the same data will fire during the same clock cycle. Among the benefits of guarded atomic actions as a design paradigm is the fact that power, area, and performance may be traded given the same input design.

A simple implementation of guarded atomic actions has been created using CHDL,

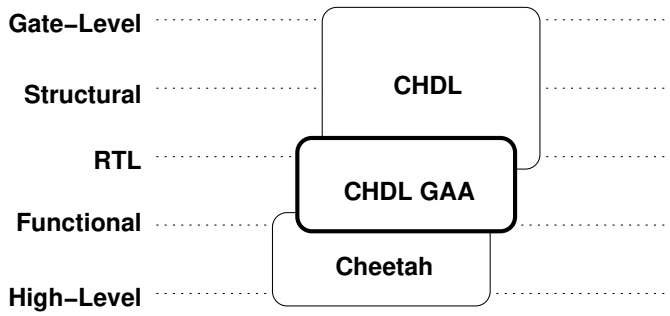


Figure 24: Guarded atomic actions operates at a level of abstraction more concrete than high-level synthesis but more abstract than register-transfer level.

providing both a case study in adapting other paradigms to CHDL in a composable fashion and an open source C++ implementation of GAA providing a path to hardware netlists. It also provides another path to a richer set of instruction set based accelerators implemented within the CHDL ecosystem. Early work with GAA, such as [52], focused on its use to describe instruction set processors, and this application remains relevant.

It should be noted that this is not the first open source implementation of the GAA paradigm. David Greaves of Cambridge produced a complete open source implementation of Bluespec [25] providing support for the language functionality as well as the paradigm. This does not, however, provide a stand-alone open source path to gate-level netlists, nor does it function as library for a generative HDL as Greaves's own later work using the Chisel HDL does [24].

The CHDL GAA library demonstrates the flexibility afforded by the use of C++ for hardware design by providing a higher level of abstraction than the structural and register transfer level description provided by CHDL itself, as illustrated graphically in Figure 24. The guarded atomic actions paradigm is more abstract than register transfer level in that a cycle-accurate simulation cannot be produced simply by analyzing a GAA description of a hardware unit. GAA is, however, more concrete than high-level synthesis because the input is an explicitly parallel hardware-oriented description and not an algorithm presented in a procedural language. This illustrates that, while CHDL ultimately generates an in-memory netlist, it is not restricted to netlist-oriented design paradigms, as already demonstrated by

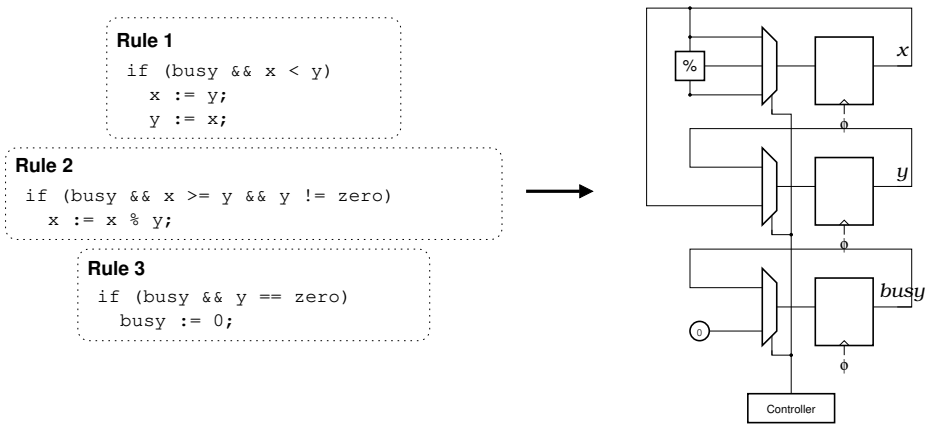


Figure 25: The CHDL implementation of guarded atomic actions performs the transformation of GAA variables into registers which are assigned based on the values of guard predicates.

the CHDL RTL API and further evidenced by the pipeline description language Cheetah described in the next chapter.

5.1 The GAA Paradigm

The essence of the guarded atomic actions paradigm is the ability to design hardware as a hierarchy of state machines, with each component providing an abstract interface through a set of exposed methods, including actions, which modify state, and values, which simply read state. The behavior of these hardware units is governed by a set of rules which are activated when a set of conditions is satisfied. These rules act on internal registers and may also trigger actions in other modules, which are themselves rules that assign their registers and may trigger further actions further down the hierarchy of modules.

5.2 An Implementation of GAA in CHDL

Guarded atomic actions can be seen as an extension of the register transfer level paradigm adding implicit predicates to prevent rules writing the same register from firing simultaneously and enabling the implementation of the method call ready/valid signal interface. The same implementation strategy used in the CHDL RTL implementation can be used here; a custom register type may be introduced along with an API amounting to a domain specific language for implementing GAA designs.

Because the features of CHDL afford extensibility with a minimum of boilerplate code, the CHDL GAA library weighs in at only 214 lines of C++ code. Two data structure definitions totaling 40 lines of C++, and six additional utility functions totaling 128 more lines form the entirety of the CHDL implementation of guarded atomic actions, the bulk of which is in the two scheduler generation algorithms. The data type system and all of the logic primitives are provided by CHDL itself leaving the GAA as simply a way to declare rules and an implementation of a method for storing and updating state. The fact that CHDL-GAA does not introduce a completely new HDL and does not require the porting of a library of combinational primitives or existing functional units is one of the most attractive features of implementing paradigms such as GAAs as libraries for existing generative HDLs instead of creating new HDLs in their own right.

An implementation of guarded atomic actions must provide three basic data structures: a *register* type describing a storage location for data, a *rule* type describing a transformation that can be made on these registers if conditions are met, and some kind of *module* for packaging related registers, the rules that act on them, and the interface they provide to other modules. In the CHDL implementation, software structures have been created explicitly to handle registers and rules, and a convention established to provide the equivalent of a module. Once the design has been described, a final `gaa_generate()` function is called to produce a concrete CHDL logic implementation of the rule scheduler and clean up the heap-allocated rules.

5.2.1 `gaareg<T>`: A Templated GAA Register Type

The CHDL implementation of GAA provides registers using the `gaareg` class, a simple template class that may be wrapped around any CHDL signal data type including numeric types, aggregates, or single nodes. The register class includes a unique identifier for use in scheduler generation and a member function `gen()` that generates a CHDL register for the register contents as well as the multiplexer and write signal used to assign this register when one of its associated rules fires. This `gen()` function is called by the destructor of the `gaareg<T>` class, guaranteeing that any rules using the register will have been

declared before this generator is called. This does impose the restriction that code using guarded atomic actions must either explicitly call this `gen()` function or allow all `gaareg` objects to go out-of-scope before performing GAA scheduler generation, logic optimization, simulation, and printing of HDL output. The `gaareg<T>` class includes the definition of a cast operator to convert its contents to type `T` simply by returning the CHDL register's output signal.

5.2.2 `Rule()`: Expressing GAA Rules

Rules are declared by calling a function, `Rule(p)`, where `p` is a CHDL `node` specifying the rule's predicate. This function's return value is a reference to a heap-allocated `rule_t` object, whose member functions also return a reference to the same, allowing a call to `Rule()` to be followed by a string of invocations of its member functions, which define its behavior. These member functions are `Assign(reg, val)`, which, when the rule fires, assigns a value of type `T` to a `gaareg<T>` and `Call(obj, func, arg1, ...)`, which will invoke an action on a given object, which is defined within a member function. This action invocation is guarded by the action's guard predicate, and the rule will be prevented from firing unless this predicate is also true.

5.2.3 C++ Classes as Modules

The module concept has a ready analog in C++ classes and structs. A class may have member variables which are themselves instances of other classes, allowing the hierarchy of modules that is central to the design of Bluespec to be implemented. Since `gaareg<T>` is itself a C++ structure, registers may also be instantiated as member variables. Classes also have constructors which are run on instantiation, in which all of the calls to `Rule()` may be placed.

Methods are provided by member functions. Value methods may simply be functions that return CHDL variables when they are called. To support action methods, a function called `Action()` is provided. It simply allows the continuation of the rule that is currently being built in the called function, and takes as its argument a guard predicate. The predicate of the rule invoking the function becomes the logical and of the rule's explicit predicate and

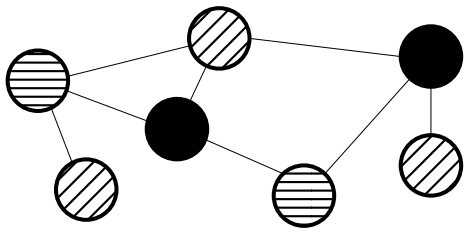


Figure 26: Example of a colored graph. No adjacent nodes have the same label, here represented by a pattern.

this guard predicate.

5.2.4 Scheduler Generator

Two scheduler options are provided by the CHDL GAA implementation: a lightweight static scheduler and a potentially more heavyweight dynamic scheduler. Both guarantee fairness and atomicity with no requirements that rule predicates be mutually-exclusive. The static scheduler pre-selects groups of rules known to be executable in parallel using graph coloring and maintains fairness by selecting one of these groups to execute each cycle, allowing other individual rules to execute if there are no conflicts. The dynamic scheduler finds groups of simultaneously-executable rules based on the predicates values in each cycle and maintains fairness by rotating the priority order of all rules each cycle.

When multiple rules may update the same register during the same cycle, it is necessary to serialize this behavior in a way that preserves the guarantees of atomicity and fairness. Since in the CHDL implementation each rule fully executes within a single clock cycle the only requirement for atomicity is that write conflicts are avoided. To avoid write conflicts, each rule is provided with, in addition to its predicate signal, a stall signal generated by a central scheduler. This scheduler is generated by the `gaa_generate()` function that must be called after all rules are declared in every CHDL function using the GAA library.

We may formulate the problem of write conflicts using a graph with vertices representing rules with true guard predicates during a given cycle and edges representing destination register conflicts. As long as no rules corresponding to neighboring vertices in this graph fire, write conflicts are prevented and atomicity is preserved. Unnecessary stalls, while

```
 1  -- inputs: a set of rules, each having
 2  --           pred[i] a predicate
 3  --           dest[i] set of destination register IDs
 4  V := 0..N_rules-1;
 5  E := i, j : rule i and j share a destination register;

 6  -- color graph returns a vertex->color map and a
 7  -- number of colors for a given graph
 8  [C, N] := color_graph(V, E);

 9  ctr := gen counter from 0 to N-1;

10  for i in V:
11    ocpred := gen Or(preds for all rules j : C[j] != C[i]);
12    cstall := gen ocpred && (ctr != C[i]);
13    stall[i] := gen cstall && Or(preds rules j : (i,j) in E);
```

Figure 27: The static scheduling algorithm option for the CHDL GAA implementation relies on graph coloring. Bold-face type is used to represent the names of variables describing signals in the generated hardware.

reducing performance, will not have a detrimental impact on correctness. Graph coloring, as illustrated in Figure 26 is the process of labeling nodes in a graph so that no two adjacent nodes have the same label. Higher-quality colorings of a graph use fewer unique labels. If we expand our graph to include vertices for rules not firing during the cycle, we can use a graph coloring algorithm to statically identify sets of rules that may always safely fire simultaneously. This is our static scheduling algorithm, described in Figure 27.

In schedulers produced by this algorithm, fairness is maintained by using a counter that repeatedly iterates through the rule color IDs, identifying active rule color. During a cycle, if any predicate with the active rule color is active, that rule is allowed to fire. Another rule color may fire only if no predicates for any rules of any other color, including the active color, are true. This way, the performance of mutually exclusive rules operating on the same set of registers is not impeded, but fairness is enforced.

The dynamic scheduling algorithm, provided as pseudocode in Figure 28, represents each predicate individually as an element in a matrix of nodes. In order to provide fairness, the rows of this matrix are rotated to place the current highest-priority rule in the top position. Since the degree of rotation is set by a counter, no rule spends more than a cycle

```
 1  -- inputs: a set of rules, each having
 2  --           pred[i] a predicate
 3  --           dest[i] set of destination register IDs

 4  P[i,j] := pred[i] if rule i assigns reg j,
 5            0 otherwise;

 6  ctr := gen counter from 0 to max rule ID;

 7  R := gen rotate rows of P by ctr;
 8  S := gen for each column of R find most-significant
 9       set bit and set all smaller bits;
10  T := gen rotate rows of S by ctr
11                direction reverse of line 7;

12  A[i,j] := gen literal 1 if rule i assigns reg j,
13            0 otherwise.
13  stall[i] := Or(A[i] & T[i]);
```

Figure 28: The dynamic scheduling algorithm option for the CHDL GAA implementation relies on the generation of relatively expensive logic. Bold-face type is used to represent the names of variables describing signals in the generated hardware.

in the top-priority position until every rule has had a chance. There is no possibility that the highest-priority rule will be stalled, but any rule attempting to write a register written by a higher-priority rule is stalled. Since this scheme is scheduled on a per-rule basis instead of per statically-assigned scheduling block, it requires considerably more hardware to implement but may lead to more efficient execution.

5.3 Applications Implemented Using CHDL GAA

The example applications for the CHDL GAA library are a hardware implementation of the dining philosopher's problem illustrating the fairness of the schedulers implemented, an implementation of Euclid's algorithm for finding the GCD of two numbers, an implementation of the Sieve of Eratosthenes providing its results as a vector of bits, and a hardware block that projects 3D points onto a 2D plane for the purpose of e.g. driving vector displays. These are provided to illustrate the expressiveness of the GAA paradigm, the fairness of the CHDL GAA scheduler, and the flexibility provided by the CHDL type system within the CHDL GAA implementation to allow different fixed, floating point, and non-numeric

Table 10: Applications implemented using the CHDL implementation of guarded atomic actions.

Name	Lines	Description
Philosophers	14	Dining philosophers fairness demonstration.
GCD	31	Greatest common divisor.
Sieve	48	Sieve of Eratosthenes.
Project	54	Project 3D points on plane.

data types to be used.

5.3.1 GCD: Euclidean Algorithm

To provide an example of the syntax afforded by the CHDL GAA implementation, we may examine an implementation of Euclid's algorithm for finding the greatest common divisor. This is the algorithm used to introduce the GAA paradigm in much of the literature on guarded atomic actions, including [48]. It is a useful example because it performs a productive task with a small number of rules:

- Given two values x and y.
- While x is not zero:
  - if $x > y$: $x \leftarrow x \mathrm{mod} y$
  - otherwise swap x and y

This algorithm is not limited to numbers and may be used to determine if, for instance, two polynomials are co-prime. The extension to polynomials is made readily in $GF(2^m)$ and a CHDL `gf<M>` type has been implemented to realize this as well.

The CHDL GAA implementation of the Euclidean algorithm is written as a single module represented by a templated `struct`. The single template argument represents the type on which a given instance of `gcd<T>` operates. This allows the data and rules to be packaged together and allows external inspection of member variables, which while potentially dangerous for maintaining abstractions may be useful for debugging.

```
template <typename T> struct gcd {
```

Three registers are used by the GCD algorithm. A state variable, `busy`, remains asserted while the algorithm is being performed and clear when it has finished. The variables `x` and `y` are the state of the algorithm. When it has finished, the result appears in `x` and `y` contains zero.

```
gaareg<node> busy;
gaareg<T> x, y;
```

All of the rules are declared in the constructor for the `gcd` struct, guaranteeing that the same set of rules will be generated for any instance of the `gcd` object and allowing the rule declarations to be co-located with the member variable declarations. The three rules declared here implement the algorithm fully, alternately swapping `x` and `y` and replacing `x` with the remainder of `x` divided by `y`. Most published versions of a GAA implementation of GCD substitute the remainder operation for the less-expensive subtraction operation, but in addition to requiring significantly more clock cycles to produce its result, this would not be applicable to objects, like $GF(2^m)$ polynomials, for which repeated subtraction is not equivalent to division.

```
gcd() {
  T zero;
  Lit(zero, 0);

  Rule(busy && x < y).
    Assign(x, y).
    Assign(y, x);

  Rule(busy && x >= y && y != zero).
    Assign(x, x % y);

  Rule(busy && T(y) == zero).
    Assign(busy, Lit(0));
}
```

Initialization of the algorithm is performed by an external invocation of the `Load()` action. The inclusion of the `!busy` predicate ensures that this may only occur while the previous GCD is not being computed.

```
  void Load(const T &a, const T &b) {
    Action(!busy).
      Assign(x, a).
      Assign(y, b).
      Assign(busy, Lit(1));
  }
};
```

A CHDL module representing a complete demonstration of the GCD algorithm is below. This takes two values and immediately on start-up loads them into the GCD unit, then permanently sets `called`, preventing any future invocations.

```
template <typename T>
  void GcdDemo(const T &x_in, const T &y_in)
{
  gcd<T> g;

  gaareg<node> called;
  Rule(!called).
    Call(g, &gcd<T>::Load, x_in, y_in).
    Assign(called, Lit(1));

  gaa_generate_color();
}
```

5.3.2 Other Examples

Dining Philosophers An implementation of the dining philosophers problem was produced to demonstrate both the fairness of the CHDL GAA scheduling algorithms and the

ability to make progress despite resource contention. Generally, dining philosophers describes a thought experiment used to illustrate concepts in concurrent computing. Seated at a table are N philosophers, who alternate between thinking and eating. Between them are N chopsticks. While eating, each philosopher needs two chopsticks, but while talking none are needed.

In GAA, the philosophers map to rules and the chopsticks to variables. During a given cycle, each register may have a single writer and rules for which all registers are not available must stall. The CHDL-GAA implementation instantiates N registers containing counters and N rules incrementing these counters, each with an always-satisfied predicate. Both schedulers allow these rules to advance in alternating sets each containing half of the rules, the optimal pattern that neither deadlocks nor starves any subset of the rules.

Sieve of Eratosthenes The sieve of Eratosthenes is an algorithm to find prime numbers by eliminating multiples of known prime numbers until only prime numbers remain. The CHDL-GAA implementation does this in phases, finding the least-significant set bit, then clearing its multiples. Its structure is quite similar to the GCD module.

Projection onto Display Coordinates The final example performs a bit of fixed-point arithmetic to perform a task that might show up in a graphics pipeline, using a set of four rules to iterate through a set of triangles one vertex per cycle, projecting these vertices onto 2D display coordinates. This example was included to provide something a bit more computationally intensive than the other examples that also took advantage of CHDL's extensive mathematical libraries, since it may be used with either fixed-point of floating-point numeric types.

5.4 Conclusions

The GAA implementation in CHDL demonstrates that it is possible, within the context of CHDL designs, to move the level of abstraction up from gate-level representations and RTL descriptions to popular higher-level paradigms and maintain a natural syntax while doing so. As shown in the following chapter, which describes a novel paradigm and implementation for describing pipelined hardware, a C++-based HDL like CHDL built around a structural

core and allowing manipulation and transformation of the in-memory netlist is inherently multi-paradigm. The CHDL GAA implementation can be considered to be a domain-specific language built on top of CHDL, exploiting the features of CHDL to provide a natural means of expressing GAA designs.

CHAPTER VI

CHEETAH: A PIPELINED HLS ENVIRONMENT WITHIN CHDL

The C++ programming language does not allow for true *reflection*, the ability to access the original syntax tree of the program as data. Because of this, implementing a fully self-contained high level synthesis system using CHDL would be a difficult proposition. However, while high-level synthesis provides a path to code reuse and designer productivity, its drawbacks, primarily in terms of designer control over the timing behavior or structure of produced logic, have led to the continued prevalence of lower-level paradigms, such as GAA, as discussed in the previous chapter. Another such paradigm is provided by Cheetah, a C++ library providing a pipeline description language implemented on top of CHDL. Cheetah automates the process of inserting pipeline flip-flops and managing valid signals and ready signals for pipelined designs, including pipelined designs with divergent and convergent data flow. This can be seen as providing an analogy to multi-threaded execution and an approach approaching the convenience of high-level synthesis while retaining cycle-level control of the data path and bit-level control of the representations of data and hardware units used to process data.

By exploiting the equivalence between multithreaded software and pipelined hardware, we can quickly construct, model, and analyze a range of both fixed function and instruction set accelerators well-suited to the energy constraints of modern architectures. This is to be reached by:

- Realization of a domain specific language that provides for the high-productivity, high-performance modeling of pipelined accelerators by exploiting the equivalence of these accelerators with multithreaded software execution.
- Implementation of a range of fixed-function and general purpose accelerators.
- Automatically-generated area, energy, and fault models of these accelerators.

- Evaluation of these accelerators in the context of near-memory processing.

The proliferation of accelerator cores of various types, both fixed-function and general-purpose, has led to a simulation gap in microarchitecture research. Accurately modeling the performance, energy, area, and fault tolerance of every component of an accelerator-rich architectures is a is a daunting task. Much work has been done recently on creating integrated simulation platforms bringing together the best tools for modeling each component of potentially-heterogeneous many-core architectures, [47] [57] [53] but for many classes of accelerators, the models do not yet exist.

It is difficult to produce credible area, and power estimates without either an example of the modeled component in silicon, a silicon-ready HDL design, or at least sketch of the design in hardware at the level of detail needed to infer an HDL design. Producing models of accelerators using hardware description languages like Verilog and VHDL and even more modern HDLs like Chisel is a labor-intensive process compared to writing cycle-level models in a high-level language like C++. Given the variety of accelerators a system might be expected to contain, the necessity of producing low-level models, and complexity of producing low-level models, an approach must be found that reduces the burden placed on the simulator writer.

High-level synthesis (HLS), simply transcompiling code written in a language such as C to a synthesizable HDL, is one such approach, and for many types of accelerators may be the best option. HLS approaches greatly reduce the burden on the designer to manage the details of how operations in a program are mapped onto physical devices and how these resources are scheduled. HLS approaches are a win for designer productivity, but they take a great hit in terms of expressiveness. HLS solutions map a program to an HDL with no regard to the physical structure of the resulting hardware. Channels other than the input source code must be used to describe physical constraints, and these only guide the generation of the HDL code. For this reason, there are specific optimizations that HLS solutions may miss, and specific designs that are difficult to express using HLS. Given the source code for an instruction set emulator, an HLS solution cannot be reasonably expected to produce a high-performance general-purpose processor.

To address this lack, an intermediate-level synthesis tool is needed. Instead of producing HDL code based on a language designed for programming general-purpose processors, a high-level HDL is needed that can express arbitrary hardware, but which simplifies common tasks associated with building high-performance accelerators and processor cores, such as pipeline control, moving that burden on the designer to the toolchain. Ideally, the output generated from this pipeline-oriented HDL would be targetable to both a synthesis flow, for the production of area, power, and fault models, and a high performance simulation flow, for integration into system-level models.

One of the many advantages of an HDL-oriented microarchitecture research flow is that it permits the creation, when available, of FPGA-based prototypes. The use of such prototypes, while limited in scope to relatively small systems, allows speeds unmatched by purely software solutions, and accordingly allows validation of simulation results with longer-running applications. The energy consumption of a prototype running on an FPGA is not necessarily related to that of the same design running on silicon. It is necessary, to take full advantage of this workflow, to develop low-overhead energy models that can run alongside the custom hardware on the FPGA.

The proposed work is then, in summary, to:

- Demonstrate the feasibility and productivity of a pipeline-oriented HDL in the domain of microarchitecture modeling for a variety of fixed-function and general-purpose accelerators.
- Develop technique for deriving fast power and fault models from the HDL.
- Demonstrate the feasibility of using accelerators so generated in FPGA-based prototypes, including the incorporation of FPGA-based power models.

6.1 Pipeline-Oriented Hardware Description Language

The basic approach taken is one of creating analogies between pipelined hardware and multi-threaded executions of software, and building on those analogies to encompass a range of common hardware techniques. This allows for a natural programming model similar

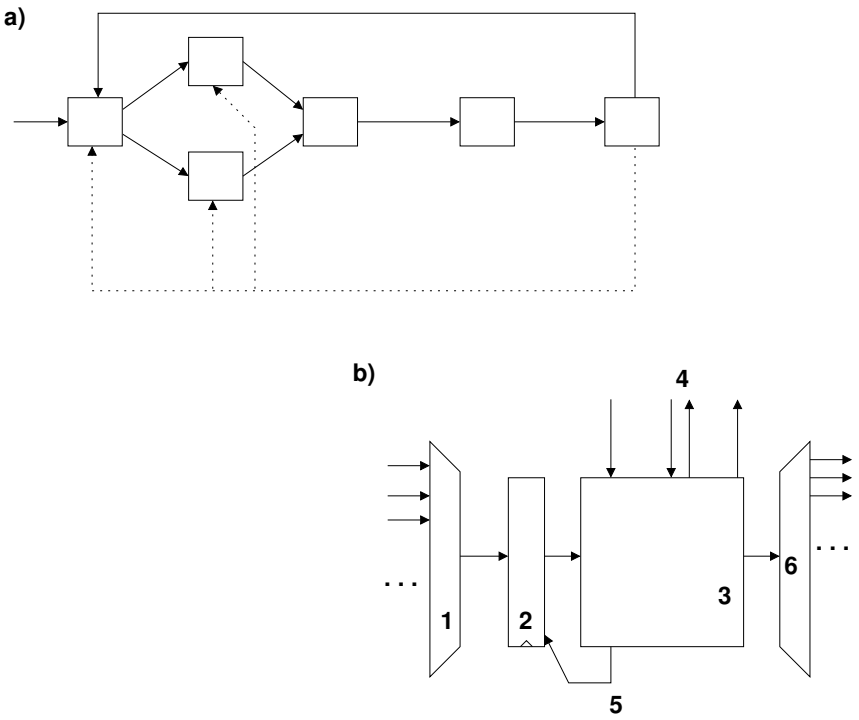


Figure 29: An example (**a**) of a pipeline design that fits the model used and (**b**) the corresponding generalized pipeline stage design.

to that of high-level synthesis tools, but with a well-defined one-to-one mapping between source code and generated hardware structures. Arithmetic and logic instructions represent fixed hardware units. Basic blocks represent pipeline stages, with a pipeline latch or buffer holding all of the signals of the live input set and an arbiter selecting between predecessor blocks. The complete pipeline model is explained in Section 6.2 Conditional branches are dispatch operations, where a successor functional unit is addressed based on a signal. A key difference from software is that it is possible for any number of successors to be branched to simultaneously.

6.2 Pipeline Model

It is necessary, before describing the features of the proposed language, to describe the model it uses to represent pipelined hardware. At their most basic, pipelines simply divide high latency tasks into a linear sequence of lower-latency steps, allowing a higher clock rate and therefore higher throughput for the area overhead of a set of pipeline latches. Practical

pipelined designs may have stages with multiple successors and predecessors, communication between stages that does not pass through pipeline registers, shared SRAM arrays, and a need to stall computation in individual stages and insert bubbles in order to maintain correctness. These features taken together constitute the pipeline model, as illustrated in Figure 29. For the purposes of this document, a *pipeline* is an arbitrary graph, not necessarily linear, of pipeline *stage*s, each of which consists of, as labeled in Figure 29, (**1**) an input arbiter, (**2**) a pipeline latch or FIFO, (**3**) the stage's logic, (**4**) connections to any external *broadcast* variables or SRAM arrays, (**5**) logic for generating pipeline stalls, and (**6**) output logic for selecting successors.

In the degenerate case, this corresponds to the simple linear pipeline described above; each stage has an input latch, some logic, and an output, and the arbiter and successor selection are optimized away. In more complex designs, the output selection circuitry and input arbiter allow for cases like the dispatching of instructions to functional units in a processor pipeline, the stalling or deflection of requests waiting for input to become ready, or the forwarding networks of modern processors.

It is assumed that there may be some interfaces to external hardware, and that interaction with this is done through an interface that can be modeled as a request and response interface, in which a request containing some information is issued and after an unknowable number of cycles a response is delivered. Systems with a one-way request-only interface can be modeled as this simply by assuming the response is returned immediately. Sections of pipeline can be encapsulated within this request/response interface, provided that live values are preserved from request to response.

The pipeline stages described here can be considered analogous to the basic blocks of software, and the request-response interface as a procedure call, but an accelerator implemented using this model only achieves its theoretical performance when multiple pipeline stages are active simultaneously. To continue the software analogy, this is a multi-threaded execution. It is by exploiting the equivalence between multithreaded software and pipelined hardware, we can quickly construct, model, and analyze a range of both fixed-function and instruction set accelerators especially well-suited to the massively parallel, low power

Table 11: Cheetah functions for defining pipeline stages.

Function	Description
`PlLabel("name")`	Start a new named pipeline stage.
`PlStage()`	Start a new unnamed stage.
`PlStall()`	Current stage's stall signal.
`PlValid()`	Current stage's valid signal.

environment of near-memory computation.

6.3 Pipeline Description Language API

Cheetah can be thought of as a way to extend the basic CHDL API with the concept of a *pipeline stage*, analogous to a basic block with an optional *label*, a *jump* function for controlling the flow of data between basic blocks, a *stall* function for implementing pipeline stalls and the concept of a *pipeline variable*, which will be carried between blocks through a set of automatically-generated pipeline registers. Stall signals may propagate throughout a pipeline and represent a potential performance bottleneck. The insertion of multi-entry buffers instead of simple D flip-flops allows the decoupling of pipeline stages to mitigate the impact of large many-stage pipelines. To enable the insertion of buffers to solve this problem and other general pipeline stage decoupling problems, the concept of a *buffer* is provided as well.

6.3.1 Pipeline Stages

In a traditional pipelined design, a pipeline stage can be considered a step in an activity that takes place during a single clock cycle. Stages are performed concurrently, improving throughput but complicating designs when communication must take place between pipeline stages. Cheetah provides the `PlLabel()` and `PlStage()` functions to make declaring a stage roughly equivalent to creating a label or line number in a traditional programming language.

This may seem limited, in that CHDL does not provide a way to declare a pipelined arithmetic operation. For instance, using the CHDL STL floating point addition operation creates a single unit of combinational logic. The use of `PlStage()` repeatedly following

Table 12: Cheetah functions for control flow between pipeline stages.

Function	Description
`PlJmp("dest", x);`	Conditionally jump to named stage.
`PlStall(x)`	Conditionally stall pipeline.
`PlEnd()`	Terminate pipelined control flow.
`PlBuf(`n`)`	Insert n-entry pipeline buffer.

Table 13: Important member functions of the `plvar` type.

Function	Description
`plvar.get()`	Get value in current stage.
`plvar.get("stage")`	Get value in another stage.
`plvar.set(x)`	Set value at this stage's output to x.
`plvar.set(x, cond)`	Conditionally set value.
`PlValid("stage")`	Valid signal of another stage.

such an operation merely creates empty pipeline stages. Cheetah relies on subsequent retiming operations to achieve performance on arithmetic operations. By declaring empty pipeline stages following complex combinational operations, a Cheetah user may leave room for such transformations to operate, thereby increasing throughput.

6.3.2 Pipeline Control Flow

In the simplest possible linearly pipelined design, work items propagate through a pipeline stage-by-stage, entering unprocessed and emerging at the end completed. Modern in-order processor pipeline designs, however, include a dispatch stage in which a single instruction may be dispatched to one of a number of pipelined functional units, each of which has a different pipeline length.

Using our multi-threaded execution analogy, dispatch operations like this may be seen as equivalent to a series of conditional branch instructions, each with a different pipeline stage as a destination.

6.3.3 Pipeline-Carried Values

A templated data type, `plvar<T>` is introduced to describe data that flows through a pipeline. Any CHDL type may be encapsulated into a `plvar<T>` and accessed with the

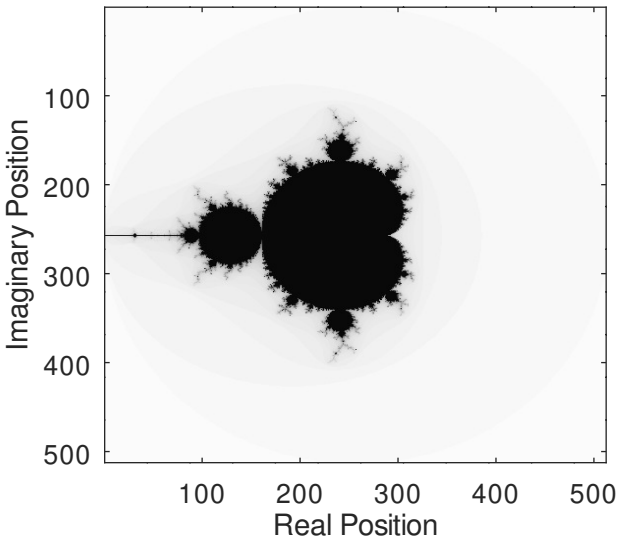


Figure 30: Mandelbrot set visualization produced by simulation of CHDL/Cheetah design. Intensity represents number of iterations required to prove divergence.

`get()` and `set()` member functions. One of the key advantages of a pipeline-oriented hardware description language is that there is no need to manually, as was done in the Iqyax and Harmonica cores described in Chapter 4, describe the contents of each pipeline register and keep track of which values are needed in which stages. Instead, a technique borrowed from compilers called liveness analysis is used to keep track of which variables must be available both at the inputs and outputs of each pipeline stage.

6.4 An Example: Mandelbrot Set Visualization

Having introduced the basic components of the Cheetah API, it is worthwhile to consider a more concrete example. The Mandelbrot set is a mathematical curiosity comprised of all complex numbers c for which the iterated function $f_{i+1}(c) = f_i(c)^2 + c$ converges. Finding points likely to be in the Mandelbrot set is a matter of evaluating this expression repeatedly. Values with a norm greater than 2 are guaranteed to diverge, providing a terminating condition, however a maximum iteration count is provided at the high end to bound execution time. Visualizations of the Mandelbrot set typically provide a visual indicator of the number of iterations required to diverge, producing infinitely complex and visually striking

patterns.

Visualizing points in the Mandelbrot set is a classic example of an embarrassingly-parallel compute-bound application. Iterations on individual points are serial, but many points can be evaluated simultaneously independently of one-another, with no communication or storage requirements.

The Cheetah implementation of this algorithm is 127 lines long; shorter than a non-pipelined plain CHDL version, although that version spends tens of additional lines implementing a complex arithmetic library not used in the Cheetah version. The Cheetah version, however, provides a variable length pipeline executing multiple iterations in parallel, providing nearly-linear scaling with the number of parallel iterations despite each iteration occupying three pipeline stages itself.

6.4.1 Constant Declarations

The CHDL numeric library, part of the CHDL STL, discussed in Section 3.5, establishes a convention of declaring literal values for representations of real numbers using the `Lit()` function. All constants used are declared at the beginning of the main function.

```
word_t x_inc, y_inc, x0, y0, four, two;
Lit(x_inc, XINC);
Lit(y_inc, YINC);
Lit(x0, XMIN);
Lit(y0, YMIN);
Lit(four, 4.0);
Lit(two, 2.0);
```

6.4.2 Pipeline Variable Declarations

After the constants, all of the pipeline-carried values are declared. It is not necessary to declare these all at the beginning of a CHDL pipeline generator program, or for their scope to persist. The only reason these are declared at the beginning is to clearly demonstrate the amount of pipeline-carried state.

The first two variables, `i` and `j` contain the integer coordinate of the point.

```
plvar<bvec<20> > i, j;
```

The type `word_t` has already been declared to be equivalent to `fp16_t`, a 16-bit half-precision floating point type, also from the CHDL numeric library. The variables `x` and `y` contain the real and imaginary components, respectively, of the point being evaluated. This is the fixed value, the c in $f(z) = z^2 + c$. The result of this iterated evaluation is stored in `zx` and `zy`.

```
plvar<word_t> x, y, zx, zy;
```

The iteration count is stored in `iter`. At 255 iterations, a point is considered to likely be contained in the Mandelbrot set and the loop is exited.

```
plvar<bvec<8> > iter;
```

The `ovfl` flag is set when the point is proven to not be a member of the Mandelbrot set, i.e. $|z| > 2$ for any iteration.

```
plvar<node> ovfl;
```

6.4.3 Spawn Loop

A thread must be spawned for each point to be processed, and an ordinary CHDL state machine consisting of two counters, `ictr` and `jctr` is used to do this. These counters are stalled, through the use of the write signal on the `Wreg()`, during all pipeline stalls. These non-pipelined variables enter the pipeline through the calls to the `set()` functions of the `plvar`s.

```
PlLabel("main"); {
  bvec<20> ictr, jctr;
```

```
    node iexp(ictr == Lit<20>(XSIZE-1)),
         jexp(jctr == Lit<20>(YSIZE-1));
    PlSpawn(ictr < Lit<20>(XSIZE) && jctr < Lit<20>(YSIZE));
    i.set(ictr);
    j.set(jctr);

    ictr = Wreg(PlValid() && !PlStall(),
                Mux(iexp, ictr + Lit<20>(1), Lit<20>(0)));
    jctr = Wreg(PlValid() && !PlStall() && iexp,
                jctr + Lit<20>(1));

    word_t new_x(x0 + x_inc*IToF<5,10>(ictr)),
           new_y(y0 + y_inc*IToF<5,10>(jctr));

    x.set(new_x);
    y.set(new_y);
    zx.set(new_x);
    zy.set(new_y);
  }
```

6.4.4 Room for Retiming

The floating point intensive generation of `new_x` and `new_y` is best spread over a few pipeline stages. Since the only dependencies in the `"main"` stage are on the previous values of simple binary counters, the computation of `new_x` and `new_y` will be spread over all of the new stages created here by retiming operations.

```
  for (unsigned ii = 0; ii < 4; ++ii) PlStage();
```

6.4.5 Main Iteration Logic

The constant integer `STAGES` configures the number of simultaneous iterations being evaluated. The `"main_loop"` stages is the beginning of a long loop of repeated iterations, each

itself made of `ISTAGES` individual pipeline stages. When a terminating condition, either an overflow or a iteration count limit, is reached, control flow is transferred to the `"end"` stage.

```
PlLabel("main_loop"); {
  for (unsigned ii = 0; ii < STAGES; ++ii) {
    ovfl.set(zx.get()*zx.get() + zy.get()*zy.get() > four);

    zx.set(zx.get()*zx.get() - zy.get()*zy.get() + x.get());
    zy.set(two*zx.get()*zy.get() + y.get());
    iter.set(iter.get() + Lit<8>(1));

    for (unsigned jj = 1; jj < ISTAGES; ++jj) PlStage();

    PlJmp("end", (ovfl.get() || iter.get() == Lit<8>(255)), ii);
    if (ii != STAGES-1) PlStage();
  }

  PlJmp("main_loop", Lit(1), 0);
  PlDft();
}
```

6.4.6 Serialization of Results

The stage labeled `"end"` provides a way to gather the output and pass t out of the functional unit. In this implementation, `x_out` and `y_out` are `bvec<20>` parameters to the function containing the pipeline, and `iter_out` is a `bvec<8>`. The call to `PlEnd()` is made so that, in the absence of a successor stage, threads reaching this point will not stall indefinitely.

```
PlLabel("end"); {
  x_out = i.get();
  y_out = j.get();
```

```
    iter_out = iter.get();
    valid_out = PlValid();
    PlEnd();
  }
```

6.5 Applicability to Instruction Set Processors

While the Mandelbrot set example is a simple but nontrivial example of a high-performance pipelined design, the natural application for this kind of utility is the creation of instruction set processors. As an example of this type of application, a stream processor implementing a simple instruction set and using scoreboarding to avoid pipeline hazards has been implemented.

6.5.1 Example Processor Design

With the aim of providing a concrete example of an instruction set processor implemented using the Cheetah pipeline description language, a simple example was created. This in-order core contains variable-latency functional units and uses a scoreboarding scheme for hazard detection. Its simple instruction set contains only a few floating point operations and branches. To avoid focusing attention on addressing modes or memory system implementation, this instruction set only extracts data from one FIFO buffer and places it into another FIFO buffer.

Opcode	Mnemonic	Description
04	jmp	Unconditional jump.
10	put	Put word in output FIFO.
21	jz	Jump if zero.
22	jn	Jump if negative.
23	jp	Jump if positive.
40	ldi	Load immediate integer.
42	get	Get word from input FIFO.
71	add	Floating point add.
72	sub	Floating point subtract.
73	mul	Floating point multiply.

6.5.2 Signals

This processor design presumes that all branches are not taken, so the `take_branch` signal can be interpreted as an indication of a branch misprediction, with the corrected program counter contained in `branch_addr`. Note that neither of these signals are `plvar`s, but instead they are non-pipelined signals. This can be thought of as communication between different simultaneous threads of execution, where a branch instruction tells a set of misspeculated instructions to terminate and the program counter to update.

The remaining variables: `iaddr`, `ienc`, `a`, `b`, and `result` are all pipeline variables. They travel through the pipeline along with instructions, and correspond to each instruction's address in instruction memory, machine language encoding, operands, and result, respectively. As seen below, each of these is `get()` and `set()` at appropriate times in the pipeline implementation.

```
node take_branch;
bvec<IA> branch_addr;
plvar<bvec<IA>> iaddr;
plvar<bvec<32>> ienc;
```

```
plvar<word_t> a, b, result;
```

6.5.3 Instruction Fetch

Two pipeline stages are devoted to instruction fetch. The first computes the program counter and the second reads the instruction from instruction memory. A ROM is provided in this example as an instruction memory; implementations employing other forms of instruction storage would be similar. Both of these stages will flush their contents on a taken branch; `fetch1` will not spawn a pipeline thread unless there is not a taken branch, and `fetch2` will not jump to the next stage, `reg`, in the event of a taken branch. The call to `PlEnd()` in `fetch2` ensures that, in the event a successor stage is not named, which in this case means a branch is taken and the instruction is to be flushed, the pipeline thread corresponding to that instruction will simply be terminated instead of stalling until a valid successor is available.

```
PlLabel("fetch1"); {
  bvec<IA> next_pc, pc(Reg(next_pc));
  Cassign(next_pc).
    IF(!PlStall() && !take_branch, pc + Lit<IA>(1)).
    IF(!PlStall() && take_branch, branch_addr).
    ELSE(pc);
  iaddr.set(pc);
  PlSpawn(!take_branch);
}

PlLabel("fetch2"); {
  ienc.set(Rom<IA, 32>(iaddr.get(), "irom.hex"));
  PlJmp("reg", !take_branch);
  PlEnd();
}
```

6.5.4 Register File and Scoreboard

The `reg` stage contains both accesses to the register file and the primary instruction scheduling structure: a *scoreboard*. This is simply a vector containing a single bit for each register indicating whether this register is ready. In this implementation, a logical 0 corresponds to a register ready to be read and a logical 1 corresponds to a register whose contents will be updated by an instruction which is still in flight through the pipeline. The scoreboard bit corresponding to an instruction's destination is cleared when the instruction passes through this stage. The call to `PlStall()` causes instructions to stall in this stage until the scoreboard bits for all source registers are clear.

Register writeback is handled in the register file, despite the fact that the writeback pipeline stage is described later. The values of the `ienc` and `result` variables in the pipeline stage are used to update the scoreboard and register file contents. These values are accessed within the `reg` stage by passing the pipeline stage name argument to the `get()` member function of `plvar`. This technique is frequently used since, while many structures such as pipelined functional units are completely confined within a single pipeline stages, many are accessed across multiple pipeline stages and must be described using signals from several pipeline stages. It is possible to use ordinary CHDL signals to communicate between pipeline stages, but it is often, as in the case of `result.get("writeback")`, clearer to specify the pipeline stage alongside the variable.

```
PlLabel("reg"); {
  // Scoreboard
  node set_scoreboard(PlValid() && !PlStall());
  bvec<32> next_scoreboard, scoreboard(Reg(next_scoreboard));

  bvec<5> wbreg = ienc.get("writeback")[range<16,20>()];
  node wb(PlValid("writeback") && ienc.get("writeback")[30]);

  node rd_wait, rs1_wait, rs2_wait;
```

```
    rd_wait = Mux(ienc.get()[range<16,20>()], scoreboard)
                && ienc.get()[30];
    rs1_wait = Mux(ienc.get()[range<8,12>()], scoreboard)
                && ienc.get()[29];
    rs2_wait = Mux(ienc.get()[range<0,4>()], scoreboard)
                && ienc.get()[28];

    next_scoreboard =
      (scoreboard & ~Decoder(wbreg, wb)) |
      Decoder(ienc.get()[range<16,20>()],
                ienc.get()[30] && set_scoreboard);

    PlStall((rd_wait || rs1_wait || rs2_wait) && !take_branch);

    // Register file
    vec<32, word_t> r;
    vec<32, bvec<sz<word_t>::value> > rf;
    Flatten(rf) = Flatten(r);

    for (unsigned i = 0; i < 32; ++i)
      r[i] = Wreg(wb && wbreg == Lit<5>(i),
                  result.get("writeback"));
    a.set(Mux(ienc.get()[range<8,12>()], r));
    b.set(Mux(ienc.get()[range<0,4>()], r));

    PlJmp("dispatch", !take_branch);
    PlEnd();
  }
```

6.5.5 Dispatch and Branch Resolution

The next pipeline stage performs two distinct tasks. To keep the branch misprediction penalty short, branches are all resolved prior to the dispatch to pipelined functional units.

This is possible because floating point arithmetic operations are quite expensive compared to determining whether the value of a floating point number is positive, negative, or zero.

The first half of the dispatch stage, therefore, computes a branch destination address and determines whether the instruction at this stage is a taken branch.

```
  PlLabel("dispatch"); {
    // Decide branches.
    bvec<32> af(Flatten(a.get()));
    bvec<8> opcode(ienc.get()[range<24,31>()]);
    node z(!OrN(af[range<0,30>()])),
         pos(af[31] && !z),
         neg(!af[31] && !z);
    take_branch = (((opcode == Lit<8>(0x21)) && z) ||
                   ((opcode == Lit<8>(0x22)) && pos) ||
                   ((opcode == Lit<8>(0x23)) && neg) ||
                   (opcode == Lit<8>(0x04)))
                     && PlValid() && !PlStall();
    branch_addr = iaddr.get()+Sext<IA>(ienc.get()[range<0,7>()]);
```

Non-branch instructions are dispatched to the appropriate pipelined functional units. To perform this dispatch operation, calls to `PlJmp()` are used, selecting a successor stage based on the value of the op-code.

```
    // Dispatch all other instructions.
    PlJmp("put", opcode == Lit<8>(0x10));
    PlJmp("ldi", opcode == Lit<8>(0x40));
    PlJmp("get", opcode == Lit<8>(0x42));
    PlJmp("add", opcode == Lit<8>(0x71));
    PlJmp("sub", opcode == Lit<8>(0x72));
    PlJmp("mul", opcode == Lit<8>(0x73));
    PlEnd();
  }
```

6.5.6 Pipelined Functional Units

The pipelined functional units themselves rely entirely on the CHDL template library implementations of floating point operations. As such, their actual instantiations in our instruction set processor are fairly bare, consisting of only labeled pipeline stages followed by a number of dummy stages designed to have the latency of the arithmetic operation performed in the name stage distributed over them by retiming optimizations.

```
PlLabel("add"); {
  result.set(a.get() + b.get());
  for (unsigned i = 0; i < ADD_STAGES; ++i) PlStage();
  PlJmp("writeback");
}

PlLabel("sub"); {
  result.set(a.get() - b.get());
  for (unsigned i = 0; i < SUB_STAGES; ++i) PlStage();
  PlJmp("writeback");
}

PlLabel("mul"); {
  result.set(a.get() * b.get());
  for (unsigned i = 0; i < MUL_STAGES; ++i) PlStage();
  PlJmp("writeback");
}

PlLabel("ldi"); {
  result.set(IToF<8, 23>(ienc.get()[range<0,15>()]));
  for (unsigned i = 0; i < LDI_STAGES; ++i) PlStage();
  PlJmp("writeback");
}
```

6.5.7 FIFO Input/Output

A traditional memory hierarchy represents a complex piece of hardware beyond the scope of this demonstration design and as such has been omitted. The memory interface of our example instruction set processor uses two instructions: `get` and `put` which receive a word of data from a FIFO, here represented as a ROM, and place a word onto an output FIFO, here represented as an external, non-stallable FIFO interface, respectively.

```
PlLabel("put"); {
  node stream_out_valid(PlValid() && !PlStall());
  bvec<32> stream_out(Flatten(b.get()));
  OUTPUT(stream_out_valid);
  OUTPUT(stream_out);
  PlEnd();
}

PlLabel("get"); {
  node inc_sp(PlValid() && !PlStall());
  bvec<10> stream_ptr;
  word_t stream_in;
  Flatten(stream_in) = Rom<10,32>(stream_ptr, "drom.hex");
  result.set(stream_in);
  stream_ptr = Wreg(inc_sp, stream_ptr + Lit<10>(1));
  for (unsigned i = 0; i < GET_STAGES; ++i) PlStage();
  PlJmp("writeback");
}
```

6.5.8 Register Writeback

Finally, the register writeback stage needs to be declared. Since all of the logic of register writeback is already implemented in the register file, this is a vestigial stage and needs only a label and, to allow instructions to finish without stalling despite the fact that no successor stage is available, a call to `PlEnd()`.

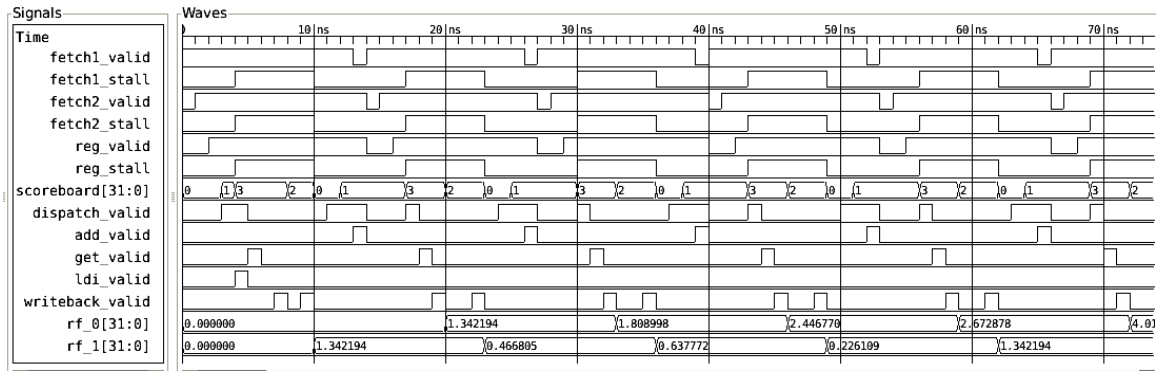


Figure 31: Waveform of operation of pipelined instruction set processor example with loop of repeated `get` and `add` instructions.

```
PlLabel("writeback"); {
  PlEnd();
}
```

6.5.9 Operation

Figure 31 shows the waveform of operation for a core using the given architecture running the two-instruction loop body "`get $r1; add $r0, $r0 $r1;`" repeatedly. This is a low-performing instruction stream due to the repetition of dependent instructions and as can be seen the core spends more time stalling for register contents to become available than executing instructions.

6.5.10 Applicability to Harmonica and Other SIMT Core Designs

The Harmonica design is inherently pipelined, and much of its code is devoted to the flow of signals through the pipeline. This is what motivated the creation of Cheetah in the first place. The technique used here to produce an example floating point oriented instruction set processor could be expanded to implement a HARP-like instruction set, forming the basis for the successor to the present generation of Harmonica implementations.

CHAPTER VII

CONCLUSIONS

Large digital design projects require high levels of intra-design and inter-design reuse in order to meet designer productivity goals and deadlines. One of the ways this can be achieved is by raising the level of abstraction in the language used to input the design. RTL is still the dominant hardware description paradigm used by digital designers targeting both silicon and FPGA, but there are mature options among high-level synthesis, dataflow, and generator-based paradigms.

Many of the HDLs and utilities introduced to support the diverse set of new hardware description paradigms suffer from a lack of mutual and backward compatibility. Given that one of the principal goals of raising the level of abstraction in hardware description is increasing potential for design reuse, the lack of mutual compatibility between these tools is a sticking point for adoption. In order to validate and synthesize designs containing pieces written using multiple paradigms, it becomes necessary to write shim layers and use the least-common-denominator language as an intermediate representation, usually some form of RTL representation. It has been shown in the literature that it is possible to combine multiple hardware description paradigms into a single mutually-compatible system in the context of generative hardware description languages. This work has demonstrated the extension of this compatibility to incorporate a full range of hardware description paradigms, from gate-level to the high-level synthesis of pipelined digital hardware from procedural descriptions.

7.1 Thesis Contributions Revisited

Modern generative HDLs provide an interface to allow the generation of RTL or other suitably low-level representation from within a general-purpose programming language. CHDL,

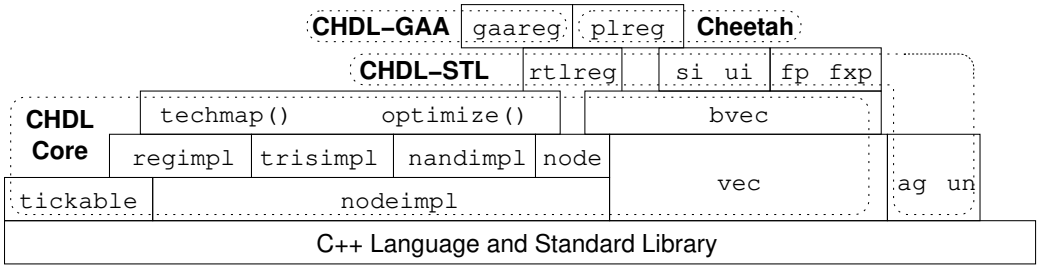


Figure 32: Copy of Figure 2. The components of CHDL described in this dissertation are outlined in dashed lines.

the vehicle for this dissertation's research, is a C++ library to enable the creation of generators, producing an in-memory netlist from the execution of a C++ program. The components and interfaces of CHDL and the related libraries discussed in this thesis are illustrated in Figure 32. This in-memory netlist is suitable for simulation, output as synthesizable Verilog, or technology mapping to a gate-level netlist suitable principally for high-level design evaluation. Natively, CHDL exports a generative, structural design paradigm, enabling the construction of hardware modules from lower-level hardware modules connected together with structured signals. Modules of CHDL designs are represented by C++ functions and instantiations of modules by calls to these functions. This lends itself very well to design reuse through the adoption of C++ templates for most of the standard library functions, allowing the use of common hardware blocks with arbitrary data types. While its native hardware description paradigm is generative, the multi-paradigm capabilities of CHDL and other generative HDLs was demonstrated with an implementation of the RTL paradigm using a custom templated register data type `rtlreg<T>`. The CHDL API, uniquely among modern generative HDLs, supports reading and modifying of the generated netlist as it is generated, allowing hardware designs to incorporate novel transformations, including optimizations and evaluation methods.

The base CHDL language is suitable for use as a platform for microarchitecture exploration as demonstrated by the Harmonica family of data parallel core designs. Harmonica is an implementation of the HARP family of instruction set architectures, a range of single-instruction-multiple-thread instruction sets that expose control flow divergence operations

as explicit instructions and allow the use of SIMT cores as both accelerators and stand-alone cores. The Harmonica architecture has been technology mapped to the FreePDK15 and FreePDK45 standard cell libraries and simulated at the gate level using the CHDL simulation environment and run on modern FPGAs.

The highest-level description to be found in the Harmonica design is register transfer level, but following the creation of Harmonica other hardware description paradigms have been implemented using CHDL as well, to further illustrate the ability of generative hardware description languages to be expanded to multiple paradigms. An example of one such paradigm is guarded atomic actions, popularized by the Bluespec System Verilog hardware description language. This paradigm extends the concept of register transfer level with the concept of a rule, a set of assignments with guaranteed atomicity and fairness, and the concept of a method, a guarded atomic interface to a module. The use of GAA provides a convenient abstraction around the proliferation of ready and valid signals ordinarily found in complex RTL designs.

The highest level of abstraction provided is found in Cheetah, which takes as input a procedural description of a pipelined design in which pipeline stages representing basic blocks are connected by what are effectively control flow branches. Liveness analysis is performed on the set of pipeline-carried values and pipeline registers are inserted wherever they are needed, effectively allowing the synthesis of hardware for highly-parallel execution of specified algorithms while retaining full compatibility with the library of data types and logic modules available within CHDL.

This level of multi-paradigm capability, while not yet demonstrated to this extent outside of CHDL, could be ported to a variety of other generative HDLs. Candidates for which this would be particularly enticing include Chisel and MyHDL, though the unique netlist introspection capabilities provided by CHDL enable modeling and optimization techniques not yet available in these candidates, the potentially multi-paradigm character of generative HDLs, also known as hardware construction languages, provides an important argument for their widespread adoption and use.

7.2 Future Directions

While the pipeline-oriented language provided by Cheetah is procedural, there remains quite a bit of room for high-level synthesis of both fully pipelined and state machine driven designs using generative HDLs. An area not yet explored that sounds very promising for future work is the potential for the use of *homoiconic* languages as hardware description languages. Languages like Java and Python that provide some *reflection*, i.e. ability to enumerate members of data structures, have been useful as HDLs due to their ability to trace signals without explicit declaration of each signal to be traced. It was the lack of reflection support that led to the CHDL `ag` and `un` types being used instead of C++ structs for structured data within CHDL.

MyHDL and JHDL both make use of the fact that some access is provided by Java and Pyhton to an intermediate representation based on the source code. In homoiconic languages by comparison, the abstract syntax tree itself is available as a data structure in that language's native data structure format. The potential for high-level synthesis is quite enticing. With a homoiconic language as the basis for a generative HDL it should be possible to introduce complete high-level synthesis of any executable code as a library function. This is one area where it would be difficult to take a language like CHDL, as there is no straightforward, portable way within C++ to process code as a data structure completely within that code. These can be approximated with macros and C++ functions, like `IF()` provided by the CHDL RTL support, but the gap between generation language and hardware-oriented language remains. Implementing a CHDL-like hardware description system in a homoiconic language would provide a path to clearing this final hurdle, dissolving the boundary between generative hardware description languages and high-level synthesis, with the drawback that there are not many very popular homoiconic languages, although homoiconicity could be shoehorned into any language with the right preprocessor.

Despite the promising directions promised by using radically different languages or significant extensions to C++ for hardware description, the value of the decision to use C++ for CHDL cannot be understated. Because C++ is an attractive or at least adequate language for implementing both CAD algorithms such as synthesis and simulation as well as hardware

description, CHDL makes it possible to integrate synthesis, logic simulation, placement and routing, timing simulation, even circuit simulation, all within the same framework. This may take the form of expanding the existing technology mapping algorithm, creating a separate in-memory format for circuit-level netlists and cell libraries, providing a discrete event simulation kernel, and adding support for physical design automation. An expanded framework library or set of libraries containing CHDL could potentially encapsulate an entire open-source hardware design platform within a widely-available, somewhat familiar, C++ compilation environment. This has great appeal in education, where the adoption of widely-installed, familiar, and accessible tools is perhaps more important than achieving the best performance money can buy.

7.3 Concluding Remarks

It is not clear where the future of hardware description lies. The present state of bifurcation between high-level synthesis and RTL design with relatively few adopting specialized HDLs or the construction of front-end scripts for RTL tools could continue indefinitely. If there is, however, as there has been in system software, a convergence toward open source tools that starts on the low end and eventually grows to encompass the industry, there may be renewed interest in interoperability and this may finally push toward the widespread adoption of one or more of the open source generative HDLs. If that day comes, it may be the last time in the era of digital computers that a new HDL needs to be adopted, as a generative language written in a modern programming language can support the full spectrum of hardware description paradigms.

APPENDIX A

HARP INSTRUCTION SET

A.1 Introduction

HARP is two things, a multi-year Heterogeneous Architecture Research Project, and an implementation of a specific Heterogeneous Architecture Research Prototype. It is for the latter that the HARP instruction set architectures have been created. This is a space of SIMT(GPU) oriented RISC-like instruction sets with the following properties:

- Full predication
- Assembly language level compatibility
- SIM[DT] parallelism
- Little endianness
- 8-bit byte size
- Customizability

The customizability of the HARP ISAs is illustrated by facts missing from this list of features. The data path width, instruction encodings, number of registers (general purpose and predicate) are all left up to the implementation. Harptool; the HARP assembler, linker, emulator, and disassembler, is passed information about the ISA through an architecture identifier string, or `ArchID`. An `ArchID` uniquely identifies a HARP ISA.

A.2 Architecture Identifier String (`ArchID`)

The best way to understand the multifaceted parameterizablity of the HARP ISAs is to study the architecture identifier strings used to uniquely identify a single HARP instruction set architecture. We'll start by breaking down Harptool's default `ArchID`: `8w32/32/8/8`:

Field	Meaning
8	8-byte (64-bit) registers and addresses
w	Word-based (64-bit) fixed-width instruction encoding
32	32 general-purpose registers per lane
32	32 predicate registers per lane
8	8 SIMD lanes
8	8 warps (thread groups)

All ArchIDs have a similar format, although the final two fields can be omitted, as object files are still fully compatible even if the dimensions of the core change.

A.3 HarpTool

The assembler/linker/emulator/disassembler program for HARP is called HarpTool. It is a multiple-function executable, its function selected with the first command line argument. When run with no command line arguments the HarpTool executable prints a help message explaining the available command line arguments.

All of the HARP utilities can take an archID as a command line parameter. If none is provided, a default will be assumed.

A.3.1 Assembler

The assembler converts assembly files to HOF, the Harp Object Format.

A.3.2 Linker

The linker combines HOF files and produces raw RAM images for use by the emulator. An intended future use is the conversion of multiple HOF files to statically-linked HOF executables.

A.3.3 Disassembler

The disassembler is used to convert HOF files to equivalent assembly files. One of its intended uses is the conversion of HOF object files between different HARP ISAs, say from 8w32/32 to 8b32/32.

A.3.4 Test Programs

In the `harptool/test` directory there is a set of test programs. The makefile in this directory assembles, links, and emulates them, placing the output in plain text files.

A.3.4.1 `hello.s`

The simplest example prints a message and exits.

A.3.4.2 `2thread.s`

`2thread` performs a vector addition across two threads.

A.3.4.3 `sieve.s`

`sieve` performs the Sieve of Eratosthenes in a single thread and prints the results, including the count of total prime numbers found.

A.4 Instruction Encoding

There are two currently-supported types of instruction encoding, but they all share a similar basic structure. The opcodes and types of fields required by each instruction are identical, differentiated only by the number of bits available for each type of field and the way predication is specified.

A.4.1 Argument Classes

Instructions can be broadly categorized by the types of arguments they require. The bit fields in the instruction encodings depend heavily on this quality.

Argument Class	Description	Example
AC_NONE	No arguments	`di;`
AC_2REG	2 GPRs, 1 in, 1 out	`neg %r1, %r2;`
AC_2IMM	1 immediate in, 1 GPR out	`ldi %r1, #0xff;`
AC_3REG	3 GPRs, 2 in, 1 out	`add %r1, %r2, %r2;`
AC_3PREG	3 pred. regs, 2 in, 1 out	`andp @p0, @p0, @p1;`
AC_3IMM	GPR in, imm. in, GPR out	`andi %r1, %r3, #3;`
AC_3REGSRC	3 GPRs in	`tlbadd %r0, %r1, %r2;`
AC_1IMM	1 imm in	`jmpi label;`
AC_1REG	1 reg in	`jmpr %r2`
AC_3IMMSRC	2 GPRs in, 1 imm. in	`st %r1, %r2, #10;`
AC_PREG_REG	GPR in, pred. reg. out	`iszero @p0, %r3;`
AC_2PREG	2 pred. regs, 1 in, 1 out	`notp @p0, @p0;`

A.4.2 Opcode/Instruction Class Table

```
00  "nop"    NONE      01  "di"       NONE      02  "ei"     NONE
03  "tlbadd" 3REGSRC   04  "tlbflush" NONE      05  "neg"    2REG
06  "not"    2REG      07  "and"      3REG      08  "or"     3REG
09  "xor"    3REG      0a  "add"      3REG      0b  "sub"    3REG
0c  "mul"    3REG      0d  "div"      3REG      0e  "mod"    3REG
0f  "shl"    3REG      10  "shr"      3REG      11  "andi"   3IMM
12  "ori"    3IMM      13  "xori"     3IMM      14  "addi"   3IMM
15  "subi"   3IMM      16  "muli"     3IMM      17  "divi"   3IMM
18  "modi"   3IMM      19  "shli"     3IMM      1a  "shri"   3IMM
1b  "jali"   2IMM      1c  "jalr"     2REG      1d  "jmpi"   1IMM
1e  "jmpr"   1REG      1f  "clone"    1REG      20  "jalis"  3IMM
21  "jalrs"  3REG      22  "jmprt"    1REG      23  "ld"     3IMM
24  "st"     3IMMSRC   25  "ldi"      2IMM      26  "rtop"   PREG_REG
27  "andp"   3PREG     28  "orp"      3PREG     29  "xorp"   3PREG
2a  "notp"   2PREG     2b  "isneg"    PREG_REG  2c  "iszero" PREG_REG
2d  "halt"   NONE      2e  "trap"     NONE      2f  "jmpru"  1REG
```

```
30  "skep"   1REG      31  "reti"     NONE      32  "tlbrm"  1REG
33  "itof"   2REG      34  "ftoi"     2REG      35  "fadd"   3REG
36  "fsub"   3REG      37  "fmul"     3REG      38  "fdiv"   3REG
39  "fneg"   2REG      3a  "wspawn"   3REG      3b  "split"  NONE
3c  "join"   NONE      3d  "bar"
```

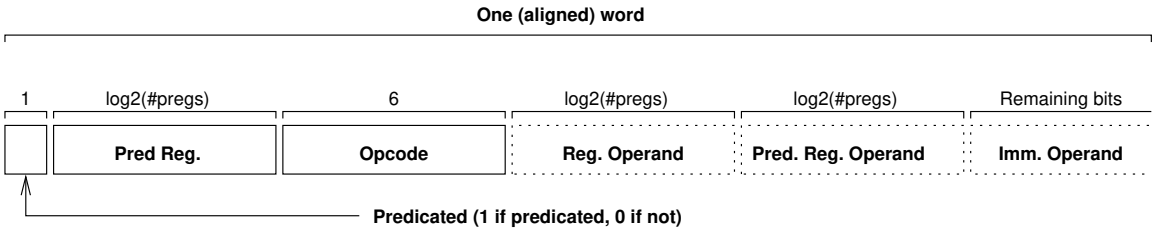
A.4.3 Word Encoding

Word-based instruction encodings all share the initial fields:

- The most-significant bit is 1 if the instruction is predicated and 0 otherwise.
- The next log_2(#pred_regs) specify the predicate register.
- The next 6 bits are used for the opcode.

After this, the operands of the instruction are ordered corresponding to their ordering in the assembly language, sized according to the following rules:

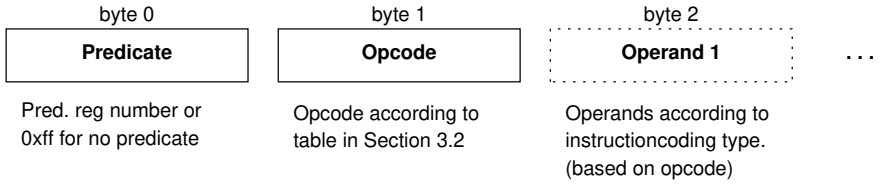
- Register operands are log_2(#GPRs) bits long, or just enough bits to uniquely identify a register.
- Predicate register operands are log_2(#pred_regs) bits long, or just enough bits to uniquely identify a predicate register.
- Immediate fields are always the last field and occupy the remaining bits of the instruction. All immediate fields are sign extended to the length of a machine word.



A.4.4 Byte Encoding

In the byte encoding, each field of the instruction (predicate, opcode, operands) occupies a byte, with the exception of immediates, which occupy an unaligned word. All instructions

have a predicate and opcode byte. The predicate byte is all ones if the instruction is not predicated; otherwise the predicate byte contains the predicate register number used to predicate the instruction. Just like the word-based instruction encoding, registers appear in the same order as the assembly language, destination-first.



A.5 Assembly Language

The assembly language is fairly easy to pick up from the Harptool examples. It is RISC-like, and written destination register first (in this it differs from Unix assembly syntax). Registers names are prefixed with the percent sign (%) and predicate register names with the at symbol (@). Predicated instructions are prefixed with the predicate register name and a question mark:

```
@p0 ? addi %r7, %r1, #1
```

A small set of directives is provided to express non-instruction data:

Directive	Use
`.align 256`	Align next symbol to a multiple of 256 bytes.
`.word 0x1234`	Insert a word with the value `0x1234`.
`.byte 0xff`	Insert a byte with the value `0xff`.
`.def SYM 123`	Replace SYM with 123 in immediate operands.
`.entry`	Make the next label the HOF executable entry point.
`.global`	Give the next label global (external) linkage.
`.perm rw`	Set HOF permissions of the next label to read/write.
`.string "Str"`	Create a null terminated string.

A.6 Instruction Set

A.6.1 Trivial Instruction

Instruction	Description
`nop`	No operation.

A.6.2 Privileged Instructions

Instruction	Description
`ei`	Enable interrupts.
`di`	Disable interrupts.
`skep` %addr	Set kernel entry point.
`tlbadd` %virt, %phys, %flags	Add an entry to the TLB.
`tlbrm` %virt	Remove entry corresponding to virt. address from TLB.
`tlbflush`	Remove all but default entry from TLB.
`jmpru` %addr	Jump indirect and switch to user mode.
`reti`	Return from interrupt.
`halt`	Halt CPU until next interrupt.

The flags register used by `tlbadd` stores, in its least-significant four bits, in order from most to least significant:

Bit	Meaning
`kx`	Kernel can execute.
`kw`	Kernel can write.
`kr`	Kernel can read.
`ux`	User can execute.
`uw`	User can write.
`ur`	User can read.

A.6.3 Memory Loads/Stores

Instruction	Description
`st` %val, %base, #OFFSET	Store.
`ld` %dest, %base, #OFFSET	Load.

A.6.4 Predicate Manipulation

Instruction	Description
`andp` @dest, @src1, @src2	Logical and.
`orp` @dest, @src1, @src2	Logical or.
`xorp` @dest, @src1, @src2	Exclusive or.
`notp` @dest, @src	Complement.

A.6.5 Value Tests

Instruction	Description
`rtop` @dest, %src	Set @dest if %src is nonzero.
`isneg` @dest, %src	Set @dest if %src is negative.
`iszero` @dest, %src	Set @dest if %src is zero.

A.6.6 Immediate Integer Arithmetic/Logic

Instruction	Description
`ldi` %dest, #Imm	Load immediate.
`addi` %dest, %src1, #Imm	Add immediate.
`subi` %dest, %src1, #Imm	Subtract immediate.
`muli` %dest, %src1, #Imm	Multiply immediate.
`divi` %dest, %src1, #Imm	Divide immediate.
`modi` %dest, %src1, #Imm	Modulus immediate.
`shli` %dest, %src1, #Imm	Shift left immediate.
`shri` %dest, %src1, #Imm	Shift right immediate.
`andi` %dest, %src1, #Imm	And immediate.
`ori` %dest, %src1, #Imm	Or immediate.
`xori` %dest, %src1, #Imm	Xor immediate.

A.6.7 Register Integer Arithmetic/Logic

Instruction	Description
`add` %dest, %src1, %src2	Add.
`sub` %dest, %src1, %src2	Subtract.
`mul` %dest, %src1, %src2	Multiply.
`div` %dest, %src1, %src2	Divide.
`mod` %dest, %src1, %src2	Modulus.
`shl` %dest, %src1, %src2	Shift left.
`shr` %dest, %src1, %src2	Shift right.
`and` %dest, %src1, %src2	And.
`or` %dest, %src1, %src2	Or.
`xor` %dest, %src1, %src2	Xor.
`neg` %dest, %src1	Two's complement.
`not` %dest, %src1	Bitwise complement.

A.6.8 Floating Point Arithmetic

These operations operate on real numbers in an implementation-determined format, which can be fixed point or floating point.

Instruction	Description
`itof` %dest, %src	Signed integer to floating point.
`ftoi` %dest, %src	Floating point to signed integer.
`fneg` %dest, %src	Negate (complement sign bit).
`fadd` %dest, %src1, %src2	Floating point add.
`fsub` %dest, %src1, %src2	Floating point subtract.
`fmul` %dest, %src1, %src2	Floating point multiply.
`fdiv` %dest, %src1, %src2	Floating point divide.

A.6.9 Control Flow

Instruction	Description
`jmpi` #RelDest	Jump to immediate (PC-relative).
`jmpr` %addr	Jump indirect.
`jali` %link, #RelDest	Jump and link immediate.
`jalr` %link, %reg	Jump and link indirect.

A.6.10 SIMD Control

Instruction	Description
`clone` %lane	Clone register state into specified lane.
`jalis` %link, %n, #RelDest	Jump and link immediate, spawning N active lanes.
`jalrs` %link, %n, %dest	Jump and link indirect, spawning N active lanes.
`jmprt` %addr	Jump indirect, terminate execution on all but lane 0.
`split`	Control flow diverge.
`join`	Control flow reconverge.

A.6.11 Warp Control

Instruction	Description
`wspawn` %dest, %pc, %src	Create new warp, copying %src in current warp to to %dest in new warp.
`bar` %id, %n	Barrier of %n warps. Identified by %id.

A.6.12 User/Kernel Interaction

Instruction	Description
`trap`	User-generated interrupt.

A.7 Interrupts

The HARP interrupt mechanism is simple. For SIMD lane 0, there is a shadow register file, program counter, and active lane count. When an interrupt occurs, the state of lane zero is saved into these shadow registers, and execution resumes at the kernel entry point. The type of interrupt is specified by the value placed in register 0 at this time, according to the following table:

Number	Description
0	Trap (user-generated interrupt)
1	Page fault due to absence from TLB
2	Page fault due to permission violation
3	Invalid/unsupported instruction
4	Divergent branch
5	Numerical domain (divide by zero)
6-7	(reserved for future exceptions)
8	Console input

The first eight interrupt numbers are reserved for internal CPU-generated exceptions, and all of the remaining numbers are free for use by hardware.

A.8 Application Binary Interface

The ABI assumes a set of at least four general purpose registers. The frame pointer is optional and can be stored on the stack itself if necessary. The stack pointer and link register, in this order, are always the two highest-numbered registers. If 8 or more registers are available, the frame pointer may be the register one less than the register number of the stack pointer.

- The lower-numbered half of the registers are caller-saved (temporary).
- The upper-numbered half are therefore callee-saved.
- The callee is responsible for adjusting the stack and frame pointers, if such adjustment is required.
- The stack grows toward smaller addresses (subtract to push, add to pop).
- Pointer function arguments and numerical arguments that can fit in a single register are passed through temporary registers, starting with register `%r0`. If more registers are required than there are temporary registers available, stack space at addresses less than the stack pointer is used.
- Record (struct) return values and numerical return values larger than the word size are always passed on the stack. The caller is responsible for allocating the necessary space. The stack pointer at the time of call is a pointer to the returned structure. All other return values are returned in `%r0`.

A.9 SIMD Operation

The HARP ISAs are inherently SIMD. In addition to designs with a single set of functional units and architectural registers, designs are allowed that replicate these while retaining a single front-end and memory system. This allows for multiple threads executing the same stream of instructions to simultaneously occupy multiple "lanes" of the processor. When a predicated control flow instruction occurs without unanimous agreement among predicate

registers, a divergent branch has occurred. The current response to this is to trap to the operating system (interrupt number 4).

A.9.1 Instructions for SIMD Operation

The `clone`, `jalis`, `jalrs`, and `jmprt` instructions form the basis of SIMD context control in the HARP instruction set. Context is created using `clone`, the waiting threads are spawned using `jalrs` or `jalis`, "jump-and-link immediate/register and spawn", and finally the parallel section returns using `jmprt`, "jump register and terminate", best thought of as "return and terminate."

There are times when a control flow operation will need to be predicated, going one direction on some lanes and the other direction on other lanes. For this, the HARP instruction set provides the `split` and `join` instructions. When a predicated `split` is first encountered, only the lanes for which the `split`'s predicate are true are allowed to continue. The other lanes are masked out until the corresponding `join` is encountered. The first time `join` is reached, control flow returns to the instruction following the corresponding `split` with the set of masked-out lanes complemented. The second time the same `join` is reached, control flow falls through and the original lane mask is restored. A hardware stack is maintained to keep track of nested `split`s.

A.10 Default I/O Devices

The emulator currently only supports a single I/O device, simple console I/O. Writing to the address `0x800...0` (an address with its MSB set and all other bits cleared) causes text to be written to the display. Input on this console interface causes an interrupt (number 8).

REFERENCES

[1] Ahn, J., Hong, S., Yoo, S., Mutlu, O., and Choi, K., "A scalable processing-in-memory accelerator for parallel graph processing," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 105–117, IEEE, 2015.

[2] Andryc, K., Merchant, M., and Tessier, R., "Flexgrip: A soft gpgpu for fpgas," in *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 230–237, IEEE, 2013.

[3] Aubury, M., Page, I., Randall, G., Saul, J., and Watts, R., "Handel-c language reference guide," *Computing Laboratory. Oxford University, UK*, 1996.

[4] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K., "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 1216–1225, ACM, 2012.

[5] Bell, C. G. and Newell, A., "The pms and isp descriptive systems for computer structures," in *Proceedings of the May 5-7, 1970, spring joint computer conference*, pp. 351–374, ACM, 1970.

[6] Bell, C. G. and Newell, A., "Computer structures: Readings and examples," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1971.

[7] Bellows, P. and Hutchings, B., "Jhdl-an hdl for reconfigurable systems," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pp. 175–184, IEEE, 1998.

[8] Bertin, P. and Touati, H., "Pam programming environments: Practice and experience," in *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pp. 133–138, IEEE, 1994.

[9] Bush, J., Khasawneh, M. A., Mahmoud, K. Z., and Miller, T. N., "Nyuzi-raster: Optimizing rasterizer performance and energy in the nyuzi open source gpu," in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pp. 204–213, IEEE, 2016.

[10] Bybell, T., "Gtkwave electronic waveform viewer," 2010.

[11] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T., "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36, ACM, 2011.

[12] Chu, M., Weaver, N., Sulimma, K., DeHon, A., and Wawrzynek, J., "Object oriented circuit-generators in java," in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No. 98TB100251)*, pp. 158–166, IEEE, 1998.

[13] Clark, N., Blome, J., Chu, M., Mahlke, S., Biles, S., and Flautner, K., "An architecture framework for transparent instruction set customization in embedded processors," in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pp. 272–283, IEEE, 2005.

[14] Coburn, J., Ravi, S., and Raghunathan, A., "Power emulation: a new paradigm for power estimation," in *Proceedings of the 42nd annual Design Automation Conference*, pp. 700–705, ACM, 2005.

[15] Consortium, H. M. C. and others, "Hybrid memory cube specification 1.0," 2013.

[16] Coussy, P., Lhairech-Lebreton, G., Heller, D., and Martin, E., "Gaut–a free and open source high-level synthesis tool," in *IEEE DATE*, 2010.

[17] Decaluwe, J., "Myhdl: a python-based hardware description language," *Linux journal*, vol. 2004, no. 127, p. 5, 2004.

[18] Del Sozzo, E., Baghdadi, R., Amarasinghe, S., and Santambrogio, M. D., "A unified backend for targeting fpgas from dsls," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 1–8, IEEE, 2018.

[19] Doumoulakis, A., Keryell, R., and O'Brien, K., "Sycl c++ and opencl interoperability experimentation with trisycl," in *Proceedings of the 5th International Workshop on OpenCL*, IWOCL 2017, (New York, NY, USA), pp. 31:1–31:8, ACM, 2017.

[20] Eker, J. and Janneck, J., "Cal language report: Specification of the cal actor language," 2003.

[21] Franklin, M. A., Tyson, E. J., Buckley, J., Crowley, P., and Maschmeyer, J., "Auto-pipe and the x language: A pipeline design tool and description language," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pp. 10–pp, IEEE, 2006.

[22] Gao, M. and Kozyrakis, C., "Hrl: efficient and flexible reconfigurable logic for near-data processing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 126–137, Ieee, 2016.

[23] Gokhale, M., Holmes, B., and Iobst, K., "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[24] Greaves, D. J., "Layering rtl, safl, handel-c and bluespec constructs on chisel hcl," in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 108–117, IEEE, 2015.

[25] Greaves, D. J., "Research note: An open source bluespec compiler," *arXiv preprint arXiv:1905.03746*, 2019.

[26] Gupta, M., *Code generation and adaptive control divergence management for light weight SIMT processors*. PhD thesis, Georgia Institute of Technology, 2016.

[27] Jaic, K. and Smith, M. C., "Enhancing hardware design flows with myhdl," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, (New York, NY, USA), pp. 28–31, ACM, 2015.

[28] Jeddeloh, J. and Keeth, B., "Hybrid memory cube new dram architecture increases density and performance," in *2012 Symposium on VLSI Technology (VLSIT)*, 2012.

[29] Jones, S., "Introduction to dynamic parallelism," in *GPU Technology Conference Presentation S*, vol. 338, 2012.

[30] Kersey, C., Yalamanchili, S., Kim, H., Nigania, N., and Kim, H., "Harmonica: An fpga-based data parallel soft core," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pp. 171–171, IEEE, 2014.

[31] Kim, D., Kung, J., Chai, S., Yalamanchili, S., and Mukhopadhyay, S., "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 380–392, IEEE, 2016.

[32] Koeplinger, D., Delimitrou, C., Prabhakar, R., Kozyrakis, C., Zhang, Y., and Olukotun, K., "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 115–127, IEEE Press, 2016.

[33] Kornmesser, K., Kugel, A., and Manner, R., "The fpga development system chdl," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pp. 271–272, March 2001.

[34] Kuper, J., Baaij, C., Kooijman, M., and Gerards, M., "Exercises in architecture specification using cλash," in *2010 Forum on Specification & Design Languages (FDL 2010)*, pp. 1–6, IET, 2010.

[35] Lockhart, D., Zibrat, G., and Batten, C., "Pymtl: A unified framework for vertically integrated computer architecture research," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 280–292, Dec 2014.

[36] Loh, G. H., "Computer architecture for die stacking," in *VLSI Design, Automation, and Test (VLSI-DAT), 2012 International Symposium on*, pp. 1–2, IEEE, 2012.

[37] Martins, M., Matos, J. M., Ribas, R. P., Reis, A., Schlinker, G., Rech, L., and Michelsen, J., "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ISPD '15, (New York, NY, USA), pp. 171–178, ACM, 2015.

[38] Merrill, D., Garland, M., and Grimshaw, A., "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 117–128, ACM, 2012.

[39] Moy, S. and Lindholm, J., "Method and system for programmable pipelined graphics processing with branching instructions," Sept. 20 2005. US Patent 6,947,047.

[40] Nikhil, R., "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pp. 69–70, IEEE, 2004.

[41] Off, I., "Computational ram: A memory-simd hybrid and its application to dsp," 1992.

[42] Panda, P. R., "Systemc-a modeling platform supporting multiple design abstractions," in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, pp. 75–80, IEEE, 2001.

[43] Panda, R., Eckert, Y., Jayasena, N., Kayiran, O., Boyer, M., and John, L. K., "Prefetching techniques for near-memory throughput processors," in *Proceedings of the 2016 International Conference on Supercomputing*, p. 40, ACM, 2016.

[44] Park, N. and Parker, A. C., "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 3, pp. 356–370, 1988.

[45] Pattnaik, A., Tang, X., Jog, A., Kayiran, O., Mishra, A. K., Kandemir, M. T., Mutlu, O., and Das, C. R., "Scheduling techniques for gpu architectures with processing-in-memory capabilities," in *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pp. 31–44, IEEE, 2016.

[46] Pilato, C. and Ferrandi, F., "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1–4, IEEE, 2013.

[47] Rodrigues, A. F., Hemmert, K. S., Barrett, B. W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., and others, "The structural simulation toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[48] Rosenband, D. L. and others, "Hardware synthesis from guarded atomic actions with performance specifications," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pp. 784–791, IEEE, 2005.

[49] Sadi, F., Pileggi, L., and Franchetti, F., "3d dram based application specific hardware accelerator for spmv," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pp. 1–1, IEEE, 2016.

[50] Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D., and Dubey, P., "Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 351–362, ACM, 2010.

[51] Shao, Y. S., Reagen, B., Wei, G.-Y., and Brooks, D., "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 97–108, IEEE, 2014.

[52] Shen, X. and others, "Using term rewriting systems to design and verify processors," *IEEE Micro*, vol. 19, no. 3, pp. 36–46, 1999.

[53] Song, W. J., Mukhopadhyay, S., and Yalamanchili, S., "Kitfox: multi-physics libraries for integrated power, thermal, and reliability simulations of multicore microarchitecture," *IEEE Transactions on Component, Packaging, and Manufacturing Technology*, vol. 5, pp. 1590–1601, Oct. 2015.

[54] Standard, J., "High bandwidth memory (hbm) dram," *JESD235*, 2013.

[55] Thoziyoor, S., Muralimanohar, N., Ahn, J. H., and Jouppi, N. P., "Cacti 5.1," tech. rep., Technical Report HPL-2008-20, HP Labs, 2008.

[56] Tripp, J. L., Gokhale, M. B., and Peterson, K. D., "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, 2007.

[57] Wang, J., Beu, J., Bheda, R., Conte, T., Dong, Z., Kersey, C., Rasquinha, M., Riley, G., Song, W., Xiao, H., and others, "Manifold: A parallel simulation framework for multicore systems," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 106–115, IEEE, 2014.

[58] Zhang, D., Jayasena, N., Lyashevsky, A., Greathouse, J. L., Xu, L., and Ignatowski, M., "Top-pim: Throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, (New York, NY, USA), pp. 85–98, ACM, 2014.

[59] Zhang, J., Khoram, S., and Li, J., "Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), pp. 207–216, ACM, 2017.

VITA

Chad Kersey was born in Valdosta, Georgia and spent his formative years living in Macclenny, Florida, graduating in 2005 from both Baker County High School in Glen St. Mary, Florida and Lake City Community College, now known as Florida Gateway College. He completed his B.S. in Computer Engineering in 2008 at Georgia Tech and remained at Georgia Tech to pursue a Ph.D. When he is not trying to advance the frontiers of of digital electronics, Chad can often be found exploring, discussing, and trying to preserve its past.