Compiling and Optimizing Spreadsheets for FPGA and Multicore Execution

by

Amir Hirsch

S.B., E.E.C.S. M.I.T., 2006; S.B., Mathematics M.I.T., 2006

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September 2007

Author_____
Department of Electrical Engineering and Computer Science
September 4, 2007

Certified by_____
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Compiling and Optimizing Spreadsheets for FPGA and Multicore Execution

by

Amir Hirsch

Submitted September 4, 2007 to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

Abstract

A major barrier to developing systems on multicore and FPGA chips is an easy-to-use development environment. This thesis presents the RhoZeta spreadsheet compiler and Catalyst optimization system for programming multiprocessors and FPGAs. Any spreadsheet frontend may be extended to work with RhoZeta's multiple interpreters and behavioral abstraction mechanisms. RhoZeta synchronizes a variety of cell interpreters acting on a global memory space. RhoZeta can also compile a group of cells to multithreaded C or Verilog. The result is an easy-to-use interface for programming multicore microprocessors and FPGAs. A spreadsheet environment presents parallelism and locality issues of modern hardware directly to the user and allows for a simple global memory synchronization model. Catalyst is a spreadsheet graph rewriting system based on performing behaviorally invariant guarded atomic actions while a system is being interpreted by RhoZeta. A number of optimization macros were developed to perform speculation, resource sharing and propagation of static assignments through a circuit. Parallelization of a 64-bit serial leading-zero-counter is demonstrated with Catalyst. Fault tolerance macros were also developed in Catalyst to protect against dynamic faults and to offset costs associated with testing semiconductors for static defects. A model for partitioning, placing and profiling spreadsheet execution in a heterogeneous hardware environment is also discussed. The RhoZeta system has been used to design several multithreaded and FPGA applications including a RISC emulator and a MIDI controlled modular synthesizer.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Throughout my time at MIT I have had the opportunity to learn from many of the greatest minds in my field. I am extremely grateful to Saman Amarasinghe for encouraging me to pursue multidimensional simulated annealing as a method of locality optimization and for his feedback during the development of this work. I also want to thank all of the members of the compilers group for our discussions of research topics and for their excellent papers on software optimizations for reconfigurable computing.

This thesis would not have gotten started without Chris Terman meeting with me weekly while I developed a thesis proposal. I am grateful to Anantha Chandrakasan for starting me down the path of FPGA research and giving me the opportunity to teach 6.111. I would also like to thank Gerry Sussman, Hal Abelson and Chris Hanson for taking me on as a UROP after my freshman year and for teaching me electronic circuit design, constraint propagation, pattern matching, rule systems and circuit simulation. I am also grateful to Professor Arvind, whose class on multithreaded parallelism and guarded atomic actions changed my perspective on automating dynamic dataflow management. Lambda is a powerful wand.

I want to thank Prasanna Sundarajan and Dave Bennett of Xilinx for having me intern on the CHiMPs compiler and introducing me to the idea of RISC emulator pipelining and GOPs/$ as an economic efficiency metric for heterogeneous supercomputing.

I also want to thank my fraternity brothers at Theta Delta Chi for their support; I know you're all sick of hearing me talk about FPGAs already. I especially want to thank Joe Presbrey for knowing how to make computers do things repeatedly, for challenging me to translate the cubes synthesizer to Python, and for staying up late all those nights with me hacking the Playstation 3 and figuring out how to make spreadsheets dispatch a graphics processor.

Finally, I would like to thank my family for being with me through it all. My parents for raising a tinkerer; my brother for being a counterpoint to my geekiness; my grandmother for making me chicken soup and teaching me mathematics when I was too sick to go to school; my grandfather for showing me that it is most important to work hard and be good to people.

Contents

List of Figures

List of Tables

Chapter 1

Reconfigurable Computing: A Paradigm Shift

> What is a von Neumann computer? When von Neumann and others conceived it over thirty years ago, it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed radically, we nevertheless still identify the notion of "computer" with this thirty year old concept.
>
> John Backus, "Can Programming Be Liberated from the von Neumann Style?" Turing Award Lecture, 1977

Reconfigurable computing is a different paradigm from traditional von Neumann computing. In the von Neumann model, a single instruction stream dictates how a global memory space is modified one address at a time. In contrast, each cell in a reconfigurable array changes its state based on the state of other cells. By working concurrently in groups, cells can produce arbitrary functionality with high speed and efficiency. While single-threaded performance will not increase beyond practical clock limits, many algorithms may be redesigned to improve with more and faster cells, and will continue to appreciate speedups as physical cell density increases. This parallel model of computing akin to hardware design, is effectively a spreadsheet and requires a system capable of spawning concurrent processes to assume the behavior of a collection of cells.

Historically, computer programs have been written in one-dimensional instruction stream languages such as C. This programming model matched the von Neumann hardware abstraction: since each instruction is executed one at a time by a central processor unit it makes sense to describe programs in an instruction stream. Modern processors have evolved beyond single-issue

instruction stream processors and more accurately resemble two-dimensional arrays of processing elements. Despite this hardware evolution, parallel hardware has inherited the one-dimensional von Neumann programming model and most programming environments require the programmer to incorporate thread synchronization code into an instruction stream.

A spreadsheet is a better hardware abstraction for modern parallel processors. A spreadsheet programming environment presents parallelism and locality directly to the user. In order to use a spreadsheet to program modern hardware, a number of backend extensions have to be built to compile and optimize spreadsheets for hardware execution. This thesis presents RhoZeta and Catalyst, a spreadsheet compiler and optimization system for FPGA, GPU, and multicore processors. The RhoZeta interpreter and compiler described in chapters two provide a spreadsheet abstraction mechanism and several execution models with simple synchronization semantics. The Catalyst optimization system presented in chapter three dynamically modifies spreadsheet structures by performing guarded atomic actions on a sheet.

1.1.1 An FPGA is a Hardware Spreadsheet

An FPGA is the hardware equivalent of a spreadsheet. Just as a spreadsheet is an array of cells containing formulas and values, FPGAs are physically arrays of cells containing logic functions and memory. Figure 1-1 shows a Mealy state machine cell. A Mealy cell consists of a state register and a next state and output function. When the cell iterates, the next state and output are determined as a combinatorial function of the current state and inputs. The output value reported to a cell's precedents is not necessarily stored in a register and may be a combinatorial function of the state and current inputs. In some cases, a cell can have both registered and combinatorial outputs so dependent cells can read from the current state or the output of the next-state function.

A spreadsheet is a software idealization of the FPGA hardware environment. Where a spreadsheet cell may have an arbitrarily defined function and value type, each cell in hardware has a fixed set of operations and memory data types. A spreadsheet compiler must map a spreadsheet definition to fixed structures available in the hardware array. In addition to dynamic types and arbitrary functions, the idealized spreadsheet has no routing constraints so each cell may refer to other cells at arbitrary distances without additional cost. Physical hardware has strict routing constraints and the overhead required to transmit a signal is strongly dependent on the distance between communicating cells. Though these issues do not affect programs in single core CPUs, with sufficiently many cores, processor arrays resemble their FPGA counterparts.

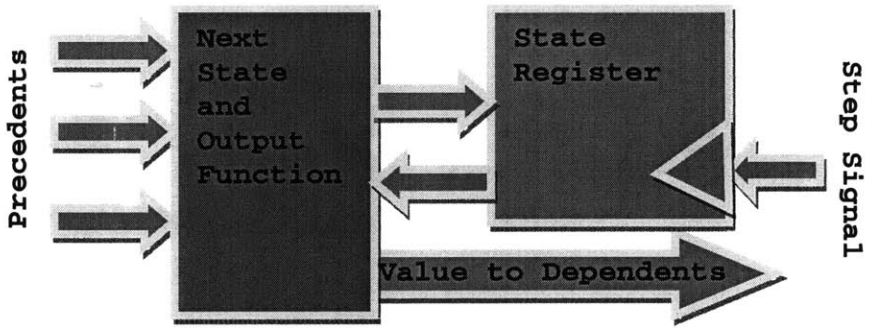


Figure 1-1 A programmable array of Mealy state machines is a model that applies to spreadsheet cells, multiprocessor cells and FPGA cells alike. A spreadsheet cell has dynamic type and arbitrary functionality. An FPGA cell has strictly typed memory and finite functionality.

1.1.2 The Increasing Importance of Locality

A spreadsheet model presents two-dimensional locality directly to the developer. The distance between physical cells is the primary metric for the cost of a function. Currently, toolchains for FPGAs and ASICs optimize a circuit graph for locality to minimize power and increase processing speed. Locality optimizations are not exclusive to ASIC and FPGA domains, they will be increasingly important in chip multiprocessors as the number of cores in such chips increases. In processor arrays, efficient placement of communicating processes will

result in decreased power and time required to transmit information between execution cores. Figure 1-2 shows that the ratio of distance between the worst and best case placement of two communicating threads in a 2-D tile array will grow as the square-root of the number of tiles.

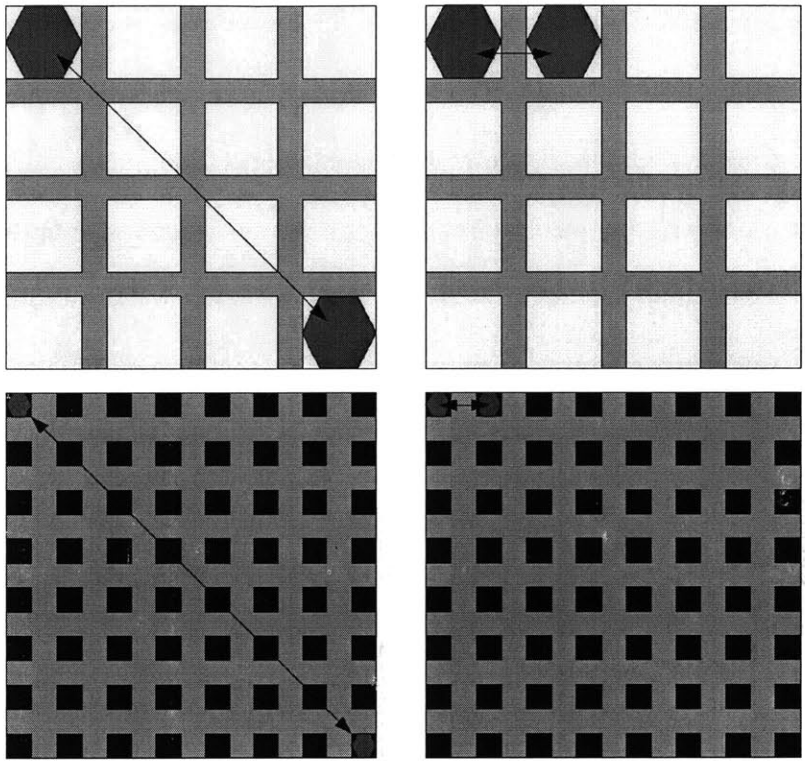


Figure 1-2 The ratio of interconnect length between optimized and worst case placement of two communicating threads increases as the square root of the number of cores in a tiled-array.

Figure 1-2 is the simplest case of two communicating threads; locality issues are compounded by the number of simultaneously communicating cells. Shrinking wires increase in resistance, and thus our ability to create fast connections between ever more cores over long distances is limited [1]. Three-dimensional wafer-stacking technologies increase the transistor density of computer chips and offer a partial solution to the interconnect bottleneck [2]. Locality optimizations for three-dimensional semiconductor devices and issues relating to process locality on multicore chips increase the need for generalized locality optimization. Chapter four provides

a general method of approaching the problem of mapping a process graph to a subset of the integer lattice Z^k.

1.1.3 Fault Tolerance and Granularity

Fault tolerance is built into the structure of a tiled array; a defect may destroy a single cell without destroying the entire system. As we continue to decrease the sizes of transistors and wires, the occurrence and severity of destructive fabrication defects increases [3]. Additionally, cosmic rays are more likely to flip a bit when they hit a smaller transistor so we will continue to see an increase in the number of dynamic faults on a chip [4]. To achieve fault tolerance, error detection and correction techniques may be employed including error correction codes, tracking and managing defective sites and using redundant components that are redundant. The complexity of testing semiconductors for defects increases as the number of components increases. Mass-producing defect tolerant architectures decreases time-to-market, increases yield, and decreases testing costs for new semiconductor technologies. To accommodate for yield costs, the eight-core Cell processor was shipped in the Playstation 3 with only seven functional cores. FPGA manufacturers have developed methods to sell partially-defective chips as ASICs and it is common to sell chips at multiple speed grades due to process variance.

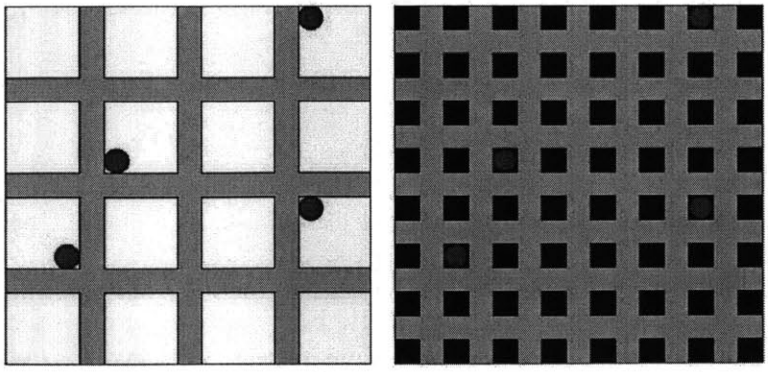


Figure 1-3 Under common defect profiles, the coarse grained array loses a quarter of all of its cores while the fine-grained array loses a sixteenth. An OS should manage component defects.

The granularity of tiles in a physical array strongly determines its susceptibility to defects. Figure 1-3 shows how susceptibility to destructive defects decreases as the number of cells on a chip increases. Each cell in a fine-grained array consumes very little in real-estate; there is little cost in a few defective FPGA cells among a million; software must be able to detect and route around them. In coarse-grained arrays, totally defective components are more costly, so measures must be taken to decrease the impact of defects. It may be possible to ignore a defect simply by disabling certain registers and instructions in a particular CPU. Error detection and correction provide protection from dynamic faults as well as static defects. Chapter three shows how to incorporate fault tolerance into spreadsheet architectures using a macro.

1.1.4 Programming Multicore Processors and FPGAs the Same Way

The structural symmetry and the similar optimizations required for a chip multi-processor and an FPGA suggest that a software design paradigm for multi-core could inherit a methodology from FPGA design. Researchers have investigated stream programming or software pipelining as an effective design paradigm for multicore architectures [5], [6]. Software pipelining is writing software as a pipelined hardware circuit. There have also been several commercial and academic compilers for C-like sequential imperative into a dataflow structure suitable for FPGA execution [7], [8], [9], [10], [11]. These FPGA compilers, like the stream programming counterparts, are generally unable to offer a solution for compiling and running legacy code on parallel architectures. In order to achieve legacy compatibility, RhoZeta interprets and optimizes machine code in a hardware emulator. Section 1.2.3 presents a RISC ISA emulator in a spreadsheet and chapter three discusses methods of compiling and optimizing a spreadsheet containing an ISA emulator and a static instruction ROM into pipelined state machines.

The dataflow paradigm is commonly represented by a spreadsheet containing cells with formulas and values. By using a spreadsheet frontend, we inherit a design tool that has been thoroughly developed for visually creating structural circuits and analyzing data. Both Excel and Calc are integrated with extension tools (COM and UNO respectively) that allow users to extend the interpreter and create transformation macros. Unfortunately, ordinary spreadsheets usually offer one mode to interpret the cells. RhoZeta allows a variety of cell interpretation strategies similar to Verilog's blocking, non-blocking and asynchronous assignments. RhoZeta can also compile spreadsheets for a heterogeneous collection of architectures including multicore processors, FPGAs and GPUs for graphical output. RhoZeta is implemented in Python and inherits its dynamically typed object system, allowing for powerful abstraction mechanisms to be built into a spreadsheet.

1.2 A Few Examples

In order to motivate this system, it is useful to consider a few simple examples. This section presents a simple counter, an IIR filter, and a RISC CPU emulator. The counter provides a simple introduction to circular reference in a spreadsheet. The IIR filter demonstrates how the interpretation method used by the spreadsheet program affects the correctness and efficiency of the system. The RISC emulator demonstrates the universality of this programming paradigm and motivates a spreadsheet to FPGA compiler. The demos in this section run in Excel, but not in Calc since Calc does not natively support circular reference. Each cell's name is in bold above its formulas. Chapter 2 will show how we extend these application frontends with our own spreadsheet interpreter system.

1.2.1 Building a 74LS163 in Excel

Traditionally, circular references in a spreadsheet are problematic and tools exist to detect

and remove them. A counter demonstrates the use of intentional circular reference in a spreadsheet. Table 1-1 shows the formulas for a simple counter. The value in A1 will increment after iteration of the spreadsheet. When the value in cell A1 reaches 100, it will return to 0 on the next iteration. If we repeatedly iterate this spreadsheet then the value in cell A1 is the same after 101 iterations.

Table 1-1. A simple counter demonstrates circular reference in a spreadsheet

	A	B
1	=IF(A1>=B1, 0, A1+1)	100

Table 1-2 extends this simple counter to behave like the 74LS163 synchronous counter with a loadable register and ripple-carry-output. Since this spreadsheet runs in Excel, iteration of the sheet is interpreted as blocking-assignments[1] performed in reading-order (left to right, top to bottom). Due to this interpretation, the series of conditions for determining Qout (Qout is cell A6) are executed in the correct order and the formula for the ripple-carry-out (RCO) must be after Qout. Section 2.1.3 describes the *Execution Policy* abstraction which instructs a *Cell Manager* how to interpret a collection of cells.

[1] A collection of blocking assignments are processed and assigned immediately as they are read. This is in contrast with non-blocking assignments which read their values before any cells are assigned. Transformations between all of the execution policies are provided in 2.1.4.

Table 1-2. 74LS163 implemented in a spreadsheet. The name of each cell is above in bold. In Excel, iterated assignments are executed in reading order from left to right, top to bottom, resulting in "priority-if" statements in the logic for Qout.

	A	B	C	D	E
1	**D**	**LD**	**T**	**P**	**Reset**
2	0	FALSE	TRUE	TRUE	FALSE
3	=IF(AND(T,P),Qout+1,Qout)				
4	=IF(Qout=MAX,0,A3)	**MAX**			
5	=IF(LD,D,A4)	15			
6	**Qout**				**RCO**
7	=IF(Reset,0,A5)				=AND(T,Qout=MAX)

Listing 1-1 is the result of direct translation of the 74LS163 spreadsheet to Verilog. It is provided here to show how to simply translate between Excel and Verilog. The cell MAX is not inferred as a parameter because the direct translation infers cells having no dependencies as inputs and having no dependents as outputs. Unnamed cells become reg objects[2] and acquire the obvious cell name: A3, A4, and A5. A type propagation system can infer that all of the unknown types in the spreadsheet are the same numeric type since Verilog requires assignments and comparison operators to have the same static type. In Chapter two, we will add directives to the spreadsheet compiler to define static types in order to add compatibility with Verilog and C.

2 A Verilog reg is not necessarily a physical registers and is only a container for a value. A synthesis tool must interpret whether a data storage element is a wire, register or latch.

Listing 1-1: Verilog Code for a 74LS163 Synchronous Counter with Synchronous Reset as produced using an Excel VBA macro for transformation. Note that the type of Qout and D is unknown, but are known to be the same numeric type.

```
module counter74LS163(clk,D,LD,T,P,Reset,MAX,Qout,RCO)
input clk;
input <%n1%> D;
input LD;
input T;
input P;
input Reset;
output reg <%n1%> Qout;
input <%n1%> MAX = 15;
output reg RCO;

reg <%n1%> A3;
reg <%n1%> A4;
reg <%n1%> A5;

always @ posedge (clk) begin
      A2 = (T && P) ? Qout + 1 : Qout;
      A3 = (Qout == MAX) ? 0 : A2;
      A4 = LD ? D : A3;
      Qout = Reset ? 0 A4;
      RCO = (T && (Qout == MAX));
end
```

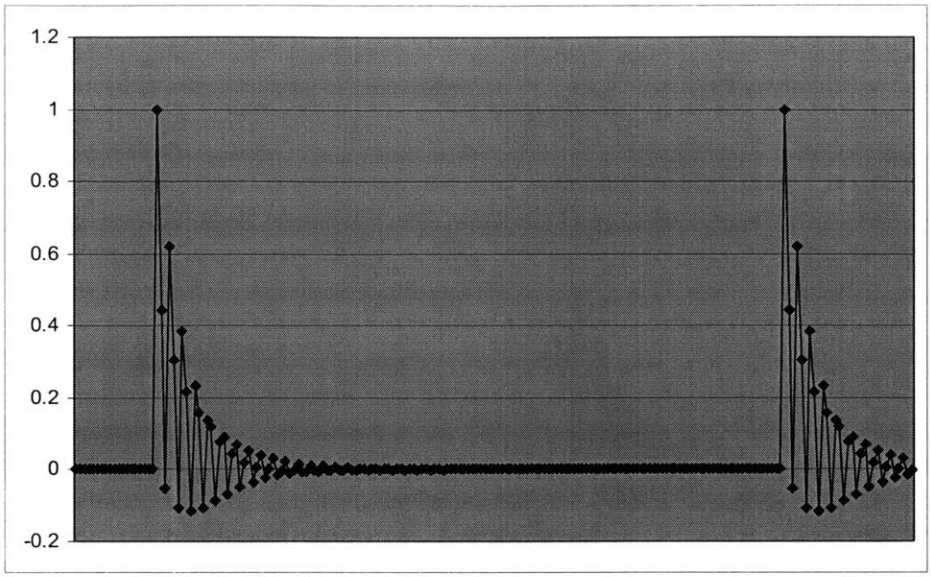
1.2.2 Infinite Impulse Response Filter

An infinite impulse response filter produces an output signal as a linear combination of the previous inputs and outputs. In order to implement an IIR filter, as in Table 1-3, we must have a shift-register structure to store the previous values of the input and output signals. Since Excel iterates in reading order, we can have a cell take on a value from the cell below it or to its right to introduce a one iteration delay between values in the table. If we instead make each cell dependent on the cell above or to the left, after iteration all cells in a chain would have the same value. In the naive software interpretation, each data item would move from one address to another address each cycle, when this structure might be more efficiently implemented in software as a circular buffer. Also, a naive interpreter might execute A1>=B1 more than once per iteration for both cells A1 and A2. Chapter 3 demonstrates how to allocate memory structures with the RhoZeta compiler and how resource sharing is managed.

Table 1-3 An IIR filter in a spreadsheet. By making each cell assume the value below or right of it, the cells behave as a shift register when iterated. Since spreadsheets automatically replicate formulas and update graphs of the data after iteration, this is an easy way to make oscilloscopes and view impulse responses. Cell A2 is an impulse generator based on the counter in the previous section.

	A	B	C	D
1	=IF(A1>=B1, 0, A1+1)	100		=D2
2	=IF(A1>=B1,1,0)			=D3
3	**x[n-2]**	**x[n-1]**	**x[n] = impulse train**	=D4
4	=xn_1	=xn	=A2	=D5
5	**a_2**	**a_1**	**a_0**	=D6
6	.4	-.3	.5	=D7
7	=xn_2*a_2	=xn_1*a_1	=xn*a_0	=D8
8	**y[n-2]**	**y[n-1]**		=D9
9	=yn_1	=yn		=D10
10	**b_2**	**b_1**		=D11
11	-.3	.8	**Yn**	=yn
12	=yn_2*b_2	=yn_1*b_1	=SUM(A7:C7,A12:B12)	

Direct translation of the filter from Excel's reading order blocking assignments to Verilog follows as in the counter demo. The coefficients are again interpreted as inputs with an unknown numeric type since they have no dependencies. To make this synthesizable Verilog, it may also be necessary to define the mathematical operators for your choice of numeric type. The IIR filter was extended in Excel to have coefficients generated by a pole-zero specification and to produce various plots of the system response. Plot of the impulse train response and frequency transfer function are shown in Figure 1-4. The impulse train response is generated by extending column D of Table 1-3. The frequency response is generated by evaluating the Z transform.



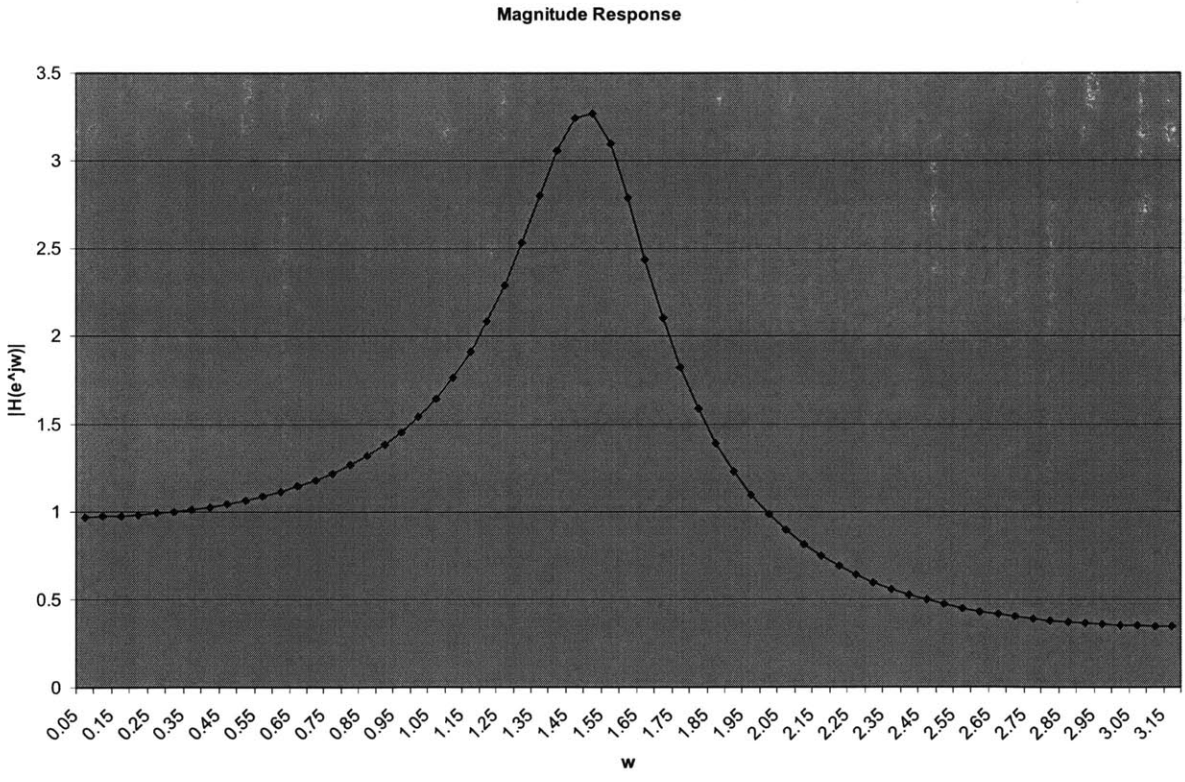


Figure 1-4: The impulse train response (above) and magnitude response (below) of the IIR filter. The magnitude of the frequency response is computed by evaluating the Z-transform of the system along the unit circle from 0 to pi radians using complex math functions.

1.2.3 A RISC CPU

Table 1-4 is a spreadsheet that executes a reduced instruction set. The first row contains the RESET signal and a constant spacebar character. The instruction decoder (A2:F4, light blue) contains a program counter, an instruction fetch from memory and an argument decoder. The argument decoder finds space characters in the instruction text string in order to parse a string into the left and right side of the space. In order to avoid parsing errors for short instructions (those with less than three spaces), spaces are concatenated to the instruction when it is read. The ReadArg block (B5:D6, red) associates register names with their value using the VLOOKUP function which can be thought of as generating a "case" statement. The ALU (C7:D14 orange) also uses VLOOKUP to associate the opcode with a function in the ALU. Each register (C15:D24, grey) is set to zero on reset and writes back the ALU output if its address is specified in arg1 of the instruction. Memory (E5:F24, green) is implemented the same way as the register file except that when it is reset it assumes the value of the cell immediately to the right. This memory structure would be inefficient if the interpreter evaluated every cell each cycle. An efficient memory macro will be introduced in chapter two as well as a number of other interpreter extensions.

The spreadsheet in Table 1-4 is designed around Excel's function set and is not meant to compile to a hardware description language. This example demonstrates a useful testing tool for CPU design and assembly level coding. Since the spreadsheet can be iterated while designed, such circuits can be built interactively. Adding opcodes and registers is trivial, though VLOOKUP requires the ALU table be sorted. It is also possible to pipeline in this structure by altering the order of the cells. Copying the sheet 10 times creates a multi-core processor model and the ALU can easily be made into a SIMD or VLIW unit too. This example motivates the

need for functions in the spreadsheet language intended for building and compiling such structures to physical hardware and FPGA. To provide FPGA compatibility for programs written for instruction stream executers we can emulate them. Chapter three will demonstrate how to create a pipeline of instruction set emulators.

Table 1-4 A reduced instruction set CPU designed to run in Excel. This system cannot compile into an HDL directly. The spreadsheet language must be extended to support a better type and object system to make this easier to design and synthesize.

	A	B	C	D	E	F
1	**Reset:**	0	**Space:**		*(D1 has a " ")*	
2	**PC**	**Inst**	**arg0**	**arg1**	**arg2**	**arg3**
3	=IF(reset, 0,nextpc)	=CONCATENATE(OFFSET(MEM, PC, 0), space, space, space, space)	=LEFT(inst, SEARCH(space, inst) - 1)	=RIGHT(inst, LEN(inst) - SEARCH(space, inst))	=RIGHT(inst, LEN(D3) - SEARCH(space, D3))	=RIGHT(inst, LEN(E3) - SEARCH(space, E3))
4				=LEFT(D3, SEARCH(space, D3) - 1)	=LEFT(E3, SEARCH(space, E3) - 1)	=LEFT(F3, SEARCH(space, F3) - 1)
5		**ReadArg1**	**ReadArg2**	**ReadArg3**	**ST**	**STC**
6		=VLOOKUP(arg1, regfile, 2)	=VLOOKUP(arg2, regfile, 2)	=VLOOKUP(arg3, regfile, 2)	=ReadArg1	=ReadArg1
7		**ALU:**	***ADD***	=ReadArg2 + ReadArg3	**Staddr**	**stcaddr**
8			***LD***	=OFFSET(MEM, ReadArg2, 1)	=ReadArg2	=arg2
9			***LDC***	=OFFSET(MEM,arg2,1)	**Write**	**Waddr**
10			***MOVC***	=arg2	=VALUE(IF(arg0 ="ST",st, IF(arg0 = "STC", stc, 0)))	=VALUE(IF(arg0 = "ST", staddr, IF(arg0 = "STC", stcaddr,0)))
11	**JMP**	**JMPC**	***MUL***	=ReadArg2 * ReadArg3	**START**	
12	=arg0 = "JMP"	=arg0="JMPC"	***ST***	=ReadArg1	=row()+2	
13	**Dest**	**DestC**	***STC***	=ReadArg1	**MEM**	**ResetValues**
14	=ReadArg1	=arg1	**ALUOUT:**	=VLOOKUP(arg0, ALU, 2)	0	0
15	**NextPC**	**RegFile**	**RegWr**	**RegWrAd**	=IF(reset,F15, IF(waddr=ROW()-start,write,MEM))	MOVC R1 443
16	=PC+1		=aluout	=arg1	=IF(reset,F16, IF(waddr=ROW()-start,write,E16))	MOVC R2 23451
17	=IF(jmp, dest, B26)		R0	=IF(reset, 0, IF(regwrad = C17, regwr, D17))	=IF(reset,F17, IF(waddr=ROW()-start,write,E17))	MOVC R3 12312
18	=IF(jmpc, destc, B27)		R1	=IF(reset, 0, IF(regwrad = C18, regwr, D18))	=IF(reset,F18, IF(waddr=ROW()-start,write,E18))	ADD R4 R2 R3
19			R2	=IF(reset, 0, IF(regwrad = C19, regwr, D19))	=IF(reset,F19, IF(waddr=ROW()-start,write,E19))	ADD R5 R1 R2
20			R3	=IF(reset, 0, IF(regwrad = C20, regwr, D20))	=IF(reset,F20, IF(waddr=ROW()-start,write,E20))	MUL R0 R4 R5
21			R4	=IF(reset, 0, IF(regwrad = C21, regwr, D21))	=IF(reset,F21, IF(waddr=ROW()-start,write,E21))	STC R0 32
22			R5	=IF(reset, 0, IF(regwrad = C22, regwr, D22))	=IF(reset,F22, IF(waddr=ROW()-start,write,E22))	JMPC 1
23			R6	=IF(reset, 0, IF(regwrad = C23, regwr, D23))	=IF(reset,F23, IF(waddr=ROW()-start,write,E23))	
24			R7	=IF(reset, 0, IF(regwrad = C24, regwr, D24))	=IF(reset,F24, IF(waddr=ROW()-start,write,E24))	

1.3 Design Principles and Efficiency Economics

The design of a complex system must be driven by a set of principles. Spreadsheet tools dictate the syntax required for point-and-click formula replication, but we are free to modify the interpreter as we please. Ordinary spreadsheets do not allow abstraction mechanisms within the sheet. We will implement a behavioral lambda which allows the user to capture the behavior of a set of cells. Spreadsheets do not ordinarily support higher order procedures, but using Python gives us these for free. Compiling higher order procedures to Verilog follows a substitution model and requires structural recursion to be resolved before attempting to program a device. RhoZeta does not aim for elegant style in the emitted C and Verilog, but rather behavioral correctness and predictable timing performance for a given spreadsheet. Transformational macros presented in chapter 3 and 4 are useful for improving the efficiency and fault tolerance of the emitted code, but ultimately, optimizing a piece of code is complicated by the various architectural tradeoffs between power, area, and speed. Chapter four explores the efficiency metric "Giga-Ops per Dollar" or GOps/$ as a metric of computational efficiency as well as a motif for analyzing and optimizing architectures under various cost functions.

1.3.1 Typing: Dynamic When Possible, Static When Necessary

Dynamic types permit rapid prototyping and fast design space exploration, but lower level languages often require strictly casted types. Spreadsheets have a somewhat dynamic type system with some strange quirks. Since we are reading the spreadsheet into Python, we immediately inherit its object system and dynamic typing. Dynamically typed spreadsheet cells allow RhoZeta to store lambda objects in a cell so that we can apply cell A1 to B1 and C1 as in Table 1-5. We may also create a structure and declare its output behavior as a function of its input cells, as shown by cell A4 of Table 1-5. By adding lambda to the language we have the ability to create sheets that contain state machine macros defined within the sheet. To compile spreadsheets, translation macros convert a sheet to a type sensitive language. By constraining some cell type declarations and propagating the effect through the

dataflow network, we can quickly explore the performance of various numeric types.

Table 1-5. Adding lambda to spreadsheets. A1(B1,C1) means apply SUM to the cells B1 and C1. Cell A2 contains the factorial function defined as a recursive function. Cell A4 contains the function f(x,y) = x*x + y so for example A4(4,4) = 20. Section 2.1.4 explains abstraction methods more in depth.

	A	B	C	D
1	SUM	1	2	=A1(B1,C1)
2	=lambda((x),if(x=0,1,x*A2(x-1)))			=A2(D1)
3				
4	=lambda((B4,C4),D5)	2	2	=B4*B4
5				=D4+C4

1.3.2 Optimizing VLIW Architectures

For some systems it may be necessary to map a spreadsheet to a constrained area, but it may be the case that a one clock-cycle per iteration direct translation of a spreadsheet will not fit in an FPGA. To solve this, an FPGA will be configured as a multicore-VLIW processor with a finite set of pipelined functional blocks and memories. Even though iteration will take many clock cycles, pipelining results in a clock speed that is often much faster than is possible using a direct translation. If the instruction sequence to be performed by a core is static, then all unnecessary arithmetic and control hardware may be removed and the core reduces to a simpler state machine. At a more fundamental level than configuring an FPGA as a VLIW, the configuration file of an FPGA can be thought of as a single long instruction specifying a sequence of look-up-table operations to be performed and stored every clock cycle. Many FPGAs are limited by the rate and methods at which their very-long "instructions" can change, though some allow for partial, dynamic, and self-reconfiguration.

1.3.3 Static Optimization

Never dedicate hardware to perform optimizations that could be done in software.

If a computer is designed to execute long instruction words specifying a set of operations to dispatch each cycle, then we could get rid of scheduling optimizations in hardware. Out-of-order

superscalar dispatches, branch predicting program counter logic, reorder buffers and any other complex instruction decoding are unnecessary for performing functions with a static execution schedule. Such controllers could be replaced with simpler state machines that can be optimized for the branching schedule of a given program. In cases where it makes sense, we can replace branching state machines with speculative pipelines and a multiplexer. In cases where complex instruction dispatching is necessary, the logic of even the most complex super-scalar scoreboard could be emulated. If we design functions with deterministic system timing, then we can maximize computational area and minimize scheduling overhead by leaving as much of the scheduling to the compiler as possible.

1.3.4 Contribution

The major contribution of this thesis is a simple model for parallel computing based on a spreadsheet. This model is easy to comprehend and allows for rapid development of dynamic dataflow applications. To extend the spreadsheet as a more complete development environment this thesis contributes:

- An abstraction mechanism to capture the behavior of a set of cells
- Compilation of a spreadsheet to Python, Verilog, and C/OpenGL
- Conversion macros between non-blocking, blocking and asynchronous spreadsheets
- A spreadsheet optimization system which achieved a nearly 45% reduction in delay for a serially-defined 64-bit leading zero-counter
- A model for automatic pipelining of arbitrary instruction set emulators
- Fault and defect tolerance macros
- An economic model for heterogeneous load-balancing and locality optimization

1.3.5 Roadmap

This chapter introduced some of the ideas in the remaining chapters and provides examples motivating a spreadsheet compiler for FPGA and chip multiprocessors. Chapter two will explain the

details of the RhoZeta interpreter and compiler and how it works in conjunction with Calc and Excel. Chapter three will explore various optimization macros including how a state machine model can be constructed and optimized from an emulator with a static set of instructions as well as demonstrate various methods for detecting and tolerating faults. Chapter four will explain the GOPs/$ metric for computational efficiency and discuss issues related to locality optimization and heterogeneous partitioning of spreadsheet computation.

Chapter 2

RhoZeta: Compiling Spreadsheets to Multicore and FPGA

Programs must be written for people to read, and only incidentally for machines to execute.
--Abelson and Sussman SICP

This chapter presents the RhoZeta spreadsheet interpreter and compiler. Section 2.1 provides an overview of the Python objects used to describe sheet interpreters, behavioral lambdas and macro transformations. Section 2.2 will expand on the premise and show how to read sheets into C and compile to multithreaded code. By binding cells to MIDI, Audio and OpenGL graphics objects we construct a synthesizer called Cubes. Section 2.3 will demonstrate how to produce a Verilog structure from the same sheet. Section 2.4 will discuss a model of low-level compilation and management for statically typed tiled arrays and will demonstrate a design of a crossbar programmable self-reconfigurable array in a spreadsheet.

Computation on spreadsheet structures has been widely studied. There have already been a Fourier synthesizer [12] and a drum machine [13] built in a spreadsheet. Previous works have discussed LISP-extended, object-oriented, and Python sheets [14], [15], [16]. My experience with SIAG [17] (scheme-in-a-grid) was a motivating tool for this work and led to my developing my own spreadsheet to allow me to make modular synthesizers. An early implementation of RhoZeta was developed in Scheme by extending the circuit simulator in SICP 3.3.4 [18] to allow for dynamically typed symbols to travel on wires. Metaprogramming with spreadsheets has been done before too [19] and there has also been an example of using spreadsheets for FPGA compilation [20]. RhoZeta extends the spreadsheet metaprogramming model with a behavioral

abstraction mechanism and compiles to multiple types modern parallel hardware. Representing recursive behavioral lambdas in a spreadsheet is a non-trivial task [21]. The current implementation of RhoZeta presented in sections 2.1-2.3 is good for rapidly designing dataflow functions in a spreadsheet, though it does not yet have the low-level completeness of the hardware OS model presented in section 2.4. The thread synchronization methodology presented in 2.2.3 is based on a master-slave multiprocessor architecture built for a sample-based synthesizer [22].

2.1 RhoZeta Interpreter Overview

The RhoZeta system consists of a client spreadsheet application bridged to the RhoZeta server which interprets sheets of cells. Since OpenOffice Calc provides an interface to Python via the "Universal Network Objects" (UNO) interface, it is straightforward to create a networked Python controller for a Calc spreadsheet. Bindings to RhoZeta from Excel are built using win32com and an XMLRPC client to connect to the server. Similar bindings to RhoZeta have been made from a Javascript/HTML frontend. Figure 2-1 shows a high-level block diagram of the client and the partitioning of the server. The block diagram shows a set of hardware devices controlled through a layer of macros and scripts built beneath the main Python interpreter. In addition to the statically compiled sheets, the interpreter can also dynamically interpret spreadsheets and perform transformation macros to allow rapid prototyping and design exploration. It is generally not possible for the synchronizer to trace the state of cells in threads running in a statically compiled mode since cells may map to inaccessible system registers though OpenGL bindings will allow us to render data as graphics in a GPU.

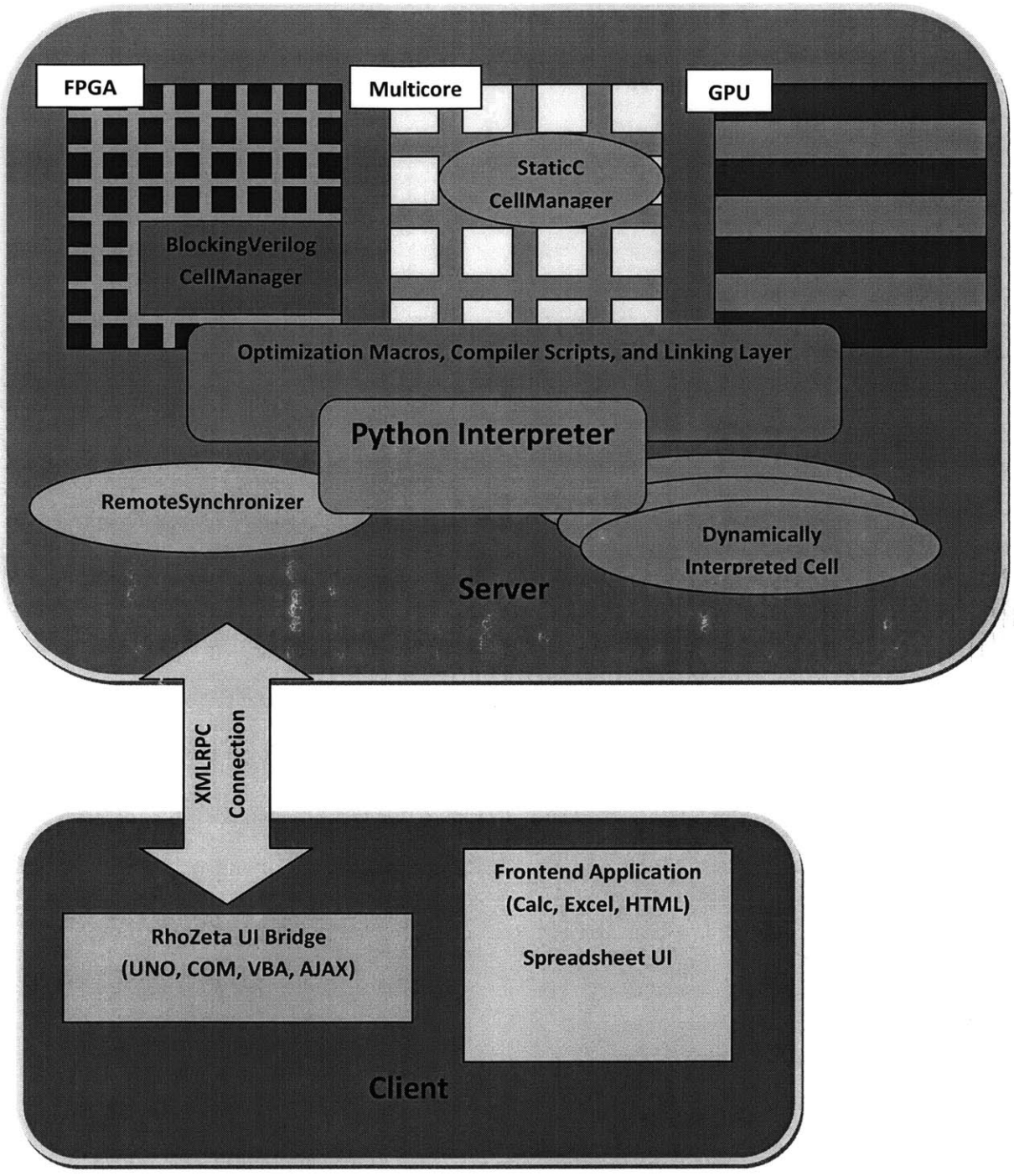


Figure 2-1 An overview of the RhoZeta spreadsheet system. The client application is responsible for managing a spreadsheet user interface and maintaining a socket connection with the server. The server interprets spreadsheets and compiles them with various hardware specific macros. When the client requests, the server synchronizes cell information with a client UI.

2.1.1 Frontend Application and UI Bridge

Any spreadsheet application may be a client for RhoZeta. The duty of the UI Bridge in the client is to notify the server of spreadsheet events invoked by the user and to synchronize the spreadsheet user interface with the values computed on the server. The client requests updates from the server and notifies the server of any modifications to cells made by the user. When the server receives an update request for a region of the spreadsheet from a client, a Remote Synchronizer object reports the current value of the requested cells. The client may transmit event notifications and request remote synchronization of a range of cells on a timer callback or when the user invokes an event. Usually, the frontend synchronizer informs the client of cells that are visible in the users' window. The User-Interface (UI) Bridge has been built on top of UNO for Calc and COM for Excel. A UI Bridge was also built using an HTML coded browser-based Emacs editor[1] connected via AJAX to a Python command line interpreter. The OpenGL demos in Section 2.2 shows how a UI can be designed in a spreadsheet and rendered into a window on a GPU.

2.1.2 Cell Managers: Threads for Interpreting Spreadsheets

The server stores a document model with sheets of Mealy state machine cell objects containing a state register and a next-state and output function. The server will compute the state and output values of the cells. All data entered by the user is parsed and evaluated as a formula (not just stored as a text string), a leading = sign will be used to indicate a next state computation. The parser translates ranges entered as "A3:B4" into lists of cell names. When a cell manager is initialized it must be provided with a list of cells and a mode of interpreting the

[1] DistributedAjax.com. An Emacs UI has splitting panes and an AJAX link to a command-line interpreter on the server (enabling remote Scheme scripting). The server interpreter can also pipe responses into the Javascript interpreter in the client's browser to display stuff or farm spare cycles at high traffic websites (hence distributed).

user-entered formulas. All cells with next state computations will be initialized to have null value. An execution policy specifies how to compute and commit the next values for a table of cells. Cell managers respond to a step signal by computing and storing cell values according to their execution policy. Step signals are akin to an activity lists in Verilog and may be bound to the cell value-change event of any set of cells. This forms a structure for creating event callbacks, since cells may be bound to a keyboard, mouse, or even specific pins on an FPGA.

A schedule sheet for each RhoZeta document specifies how each cell is interpreted and dictates the synchronization between cell managers. The FIFO scheduler shown in Table 2-1, spawns cell managers and listens to the setValue() method for each step signal. When setValue() changes the value of a step signal cell, a notification for the set of dependent cell managers is inserted in the FIFO. If a step notice for a cell manager is already in the FIFO then no new notice will be added. When multiple cell managers respond to the same step signal, the scheduler dispatches threads for the compute and commit phases of the execution policy separately. This is so synchronized cell managers can read from the same set of values before any values are overwritten. This simple FIFO is efficient for sequentially interpreting asynchronously scheduled events with equal priority and for synchronizing multiple cell managers. Asynchronously evaluated cells are their own cell manager with a step signal attached to all precedents and a step function assigning its own value. An asynchronous circuit with this FIFO scheduler has the property that propagation delay is strictly determined by the number of cells in the worst case signal path. To provide a more precise semantic for synchronized events we will use execution policies.

Table 2-1: A scheduler sheet. The FIFOSchedule function takes in a cell manager specification table and tells all cells attached to step signals to insert a step request into a queue when their value changes.

	A	B	C	D
1	=FIFOSchedule(A3:D5)			
2	**Cell Manager ID**	**Cell Range**	**Execution Policy**	**Step Signal**
3	ShiftRegister	Table2!A1:B10	NonBlocking	Table3!A1
4	ClockGenerator	Table3	Asynchronous	N/A
5	Scheduler	Table1!A1	ReadingOrderBlocking	Table1

2.1.3 Execution Policies: How to Interpret a Group of Cells

An execution policy specifies the mode of interpretation for a cell manager to apply to a set of cells. We have already examined the execution policy of Excel, which iterates in reading order, as though committing assignments immediately after reading and processing operands. Listing 2-1 provides Python code for the reading order assignment interpretation. Another useful mode is non-blocking assignments in which all values are computed using values from the previous state before any next state is committed. Non-blocking assignments provide simple semantics for concurrent actions and they are mobile in the sense that their spatial arrangement does not affect the sheet behavior. This will be important when we perform locality optimizations to minimize communication overhead between cells. An implementation of a non-blocking assignment interpreter is presented in Listing 2-2. In order to synchronize events when multiple execution policies respond to the same events and have overlapping data dependency, we have partitioned the step function into compute and commit phases so that we may dispatch multiple threads simultaneously. We may also translate sheets between execution polices and merge multiple sheets together. This will be explained further in section 2.1.5.

Listing 2-1: Reading Order Assignments Execution Policy. The cells are sorted in reading order when the execution policy is initialized and the NextValue dictionary is used to resolve values belonging to the cell manager while each cell is computed to emulate being assigned as they are computed. We cannot actually assign values as we are computing them since we must commit in a synchronous manner during a compute-commit cycle. We may step in the obvious way.

```
class ReadingOrderAssignments(ExecutionPolicy):
	def __init__(self,cells):
		self.Cells = ReadingOrderSort(cells);

	def compute(self):
		for cell in self.Cells:
			cell.setNextValue(eval(cell.Formula, self.Cells.NextValueDict))
	def commit(self):
		self.Cells.setValues(self.Cells.NextValues)
	def step(self):
		for cell in self.Cells
			cell.setValue(eval(cell.Formula))
```

Listing 2-2: Non-blocking Assignments. Non-blocking assignments compute next values from current registered values only. The NextValues list is computed by a list comprehension which is a syntactic convenience for map.

```
class NonBlockingAssignments(ExecutionPolicy):
	def __init__(self,cells):
		self.Cells = cells
	def compute(self):
		self.Cells.setNextValues([eval(cell.Formula) for cell in self.Cells])
	def commit(self):
		for cell in self.Cells:
			cell.setValue(cell.NextValues)
	def step(self):
		self.Cells.setValues([eval(cell.Formula) for cell in self.Cells])
```

Table 2-2: A non-blocking execution policy reads all values before assignments are made, so Row 2 would be a shift register. A blocking assignment interpretation results in Row 2 not working as a shift register.

	A	B	C	D
1	=B1	=C1	=D1	=IF(Reset,0,D1+1)
2	=IF(Reset,0,A2+1)	=A2	=B2	=C2

A useful execution policy for running Python scripts is Reading Order Execution, which is similar to reading order assignments, except that the evaluation: `eval(cell.Formula)` is replaced by execution `exec(cell.Formula)`. This allows us to execute Python macros right in the spreadsheet. Table 2-3 shows how to define this execution policy within a spreadsheet. Calling exec modifies state beyond just cells in our spreadsheet and violates of the

functional dataflow style, though it demonstrates the meta-programming methodology we will use to connect to lower-level compiled objects and to develop optimization macros. Another way to achieve the same effect as reading order execution is to write cells into a file and execute the script. This is precisely what we will do to create and run Verilog and C.

Table 2-3: The reading order execution policy runs a spreadsheet as a thread of Python code. This does not support the compute-commit synchronization interface and is not intended for synchronized execution, though running multiple reading order execution threads on a spreadsheet allows for a visual sandbox to test multithreaded algorithms.

	A	B	C	D
1	`class ReadingOrderExecution(ExecutionPolicy):`			
2		`def __init__(self,cells):`		
3			`self.Cells = ReadingOrderSort(cells)`	
4		`def step(self):`		
5			`for cell in self.Cells:`	
6				`exec('\t'*cell.Column+cell.Formula+'\n')`

The asynchronous event-based execution policy is the most traditional way to think of a spreadsheet: a cell recalculates when its precedents change. This is implemented by creating an individual cell manager for every cell with step signal bound to all direct precedents of that cell. Asynchronous cells are their own managers. Their execution order is entirely dependent on the scheduler. Without circular reference, an asynchronous execution policy is equivalent to a combinatorial circuit. Of course there isn't any reason there couldn't be a circular reference for example, cross coupled NAND gates can make an SR-latch, or two muxes make a register as in Table 2-4. Figure 2-2 demonstrates the order in which the asynchronous cells are evaluated. Better yet, non-convergent circular reference can be used to create ring oscillators to clock our register. Like non-blocking assignments, asynchronous assignments are mobile and can be moved anywhere in the spreadsheet without affecting the implied system behavior.

Table 2-4: Using the asynchronous execution policy we can produce a ring-oscillator and two registers. A1 is our clock. Row 2 is falling-edge triggered, row 3 is rising-edge triggered. Section 3.4 will show how an asynchronous ring oscillator and a counter can detect faulty circuits and measure the communication lags between points in a hardware topology.

	A	B	C	D
1	=AND(CE,NOT(C1))	=NOT(A1)	=NOT(B1)	CE
2	0	=MUX(A1,A2,B2)	=MUX(NOT(A1),B2,C2)	1
3	0	=MUX(NOT(A1),A3,B3)	=MUX(A1,B3,C3)	

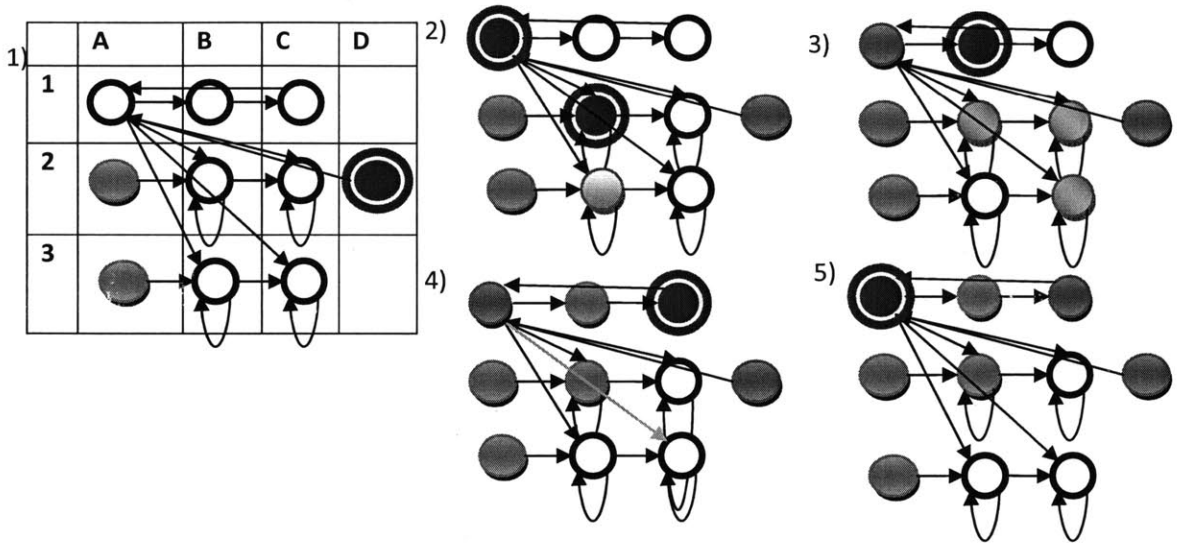


Figure 2-2 Demonstrating the evaluation of the circularly referent asynchronous clock and register. Green cells are 0, Blue cells are 1, white cells have null value, red cells are currently being evaluated, red cells with a red border take on a new value. To start the circuit, a clock enable signal, CE and register inputs, are set in step 1. Cells A1, B2 and B3 react in phase 2. Assuming A1 evaluates first in phase 2 then B2 and C2 will not be notified by A1 since they are already in the queue for phase 2. In phase 2, Cell A1 is assigned to 1 and B2 is assigned to 0 so all of their dependents (B1, B2, C2 C3,) will evaluate in phase 3. Only B1 is assigned a new value in phase 3, so C1 is evaluated in phase 4. In phase 5 the clock will fall and a similar chain reaction will ensue.

2.1.4 Behavioral Lambdas in a Spreadsheet

In chapter one, we introduced the notion of assigning a cell to a lambda object without fully explaining the mechanism for resolving closures or managing internal state. Our lambda operator is used to capture the behavior of a Mealy state machine expressed in a set of cells.

Lambda merges the behavior of a set of cells to a behavior than fits in a single cell. Thus RhoZeta's lambda is like a Verilog module abstraction with a single output expression per module (though the output may be a list of multiple outputs). The lambda function requires a list of names to rebind and an expression to evaluate in place of the lambda definition. For example, the impulse generator in Table 2-5 has "Period" as its argument and "counter=0" as its lambda expression; when the ImpulseGen macro is expanded in row 4, preserving reading order semantics, Period is bound to "10" and the lambda cell's formula is substituted with "counter=0" with "counter" bound to the spawned copy in A4. Evaluation of ImpulseGen in C3 causes a step of row 4 and returns the value of C4.

Table 2-5: An ImpulseGen lambda is created in cell C2. When the lambda is applied in C3, it creates hidden state, represented as the italic, bold formulas in row 4. Evaluating ImpulseGen(10) steps row 4 and returns the value of C4.

	A	B	C	D
1	**Counter**	**Period**	**ImpulseGen**	RESET
2	=IF(RESET,0, IF (Counter = Period, 0, Counter+1)	900	=lambda((Period),if(Counter=0,0,1))	0
3			=ImpulseGen(10)	
4	***=IF(RESET,0, IF(A4=B4, 0 A4+1))***	***10***	***=A4=0***	

Lambda application is thought of as cloning the cell manager at the lambda definition, modifying the bound cells' formulas to the passed arguments, stepping the execution policy whenever the application is evaluated and returning the value in the cell containing the lambda expression. This interpretation of lambda allows for structural recursion as in the delay_by example of Table 2-6. As delay_by unfolds, new hidden states are created until the base case is reached. Whenever a lambda definition contains a parameter name that does not correspond to a cell, then a cell will be spawned before everything else in the reading order. If the delay length

parameter, n, in C5 were to increase, then the lambda would continue to unfold and no state would be lost; if C5 were to decrease, then an IF statement at a lesser delay would resolve to a value and we would lose the state deeper in the pipeline. IF statements with lambdas as conditional inputs remove the internal lambda states when they resolve to a non-lambda value. Recursive lambda invocation must be bound within an IF statements or it will unroll infinitely.

Table 2-6: The "delay_by" lambda is created using a recursive call. Rows 6,7 and 8 are the hidden states spawned by each call. Recursive macros like this must be unrolled entirely to compile to C or Verilog.

	A	B	C	D
1	**delayedby1**	**In**	**delay_by(n,in)**	
2	=In	0	=lambda((n, In), if (n=0, In, delay_by(n-1, delayedby1)))	
3			=ImpulseGen(10)	
4	***=IF(Table5!RESET,0, IF(A4=B4, 0 A4+1))***	***10***	***=A4=0***	
5	n		2	=delay_by(C5,C3)
6	***=C5-1***	***=C6***	***=C5***	***=delay_by(A6,B6)***
7	***=C6-1***	***=C7***	***=B6***	***=delay_by(A7,B7)***
8	***=C7-1***	***=C8***	***=B7***	***= IF(B5-2=0, C8, ...)***

If no state dependencies exist in the formula for the lambda expression, then a combinatorial function is inferred. This is the case with stateless blocking assignments that have no dependencies on a previous iteration (no-down-right dependencies). Free variables retain a binding to the cells where the lambda is defined. For example, the RESET signal of the impulse generator of Table 2-6 is still bound to the RESET cell of Table 2-5 where ImpulseGen was defined. Free variables may be bound like any Python function call with optional arguments; cell A4 of Table 2-7 assigns the MAX parameter of a counter to 31. This is a natural way to create objects or functions with default parameters.

Table 2-7: An example of mixing execution policies and using lambda across mixed modes. Row 2 is reading order blocking and row 4 is non-blocking. The counter is abstracted in a lambda object, which takes an increment parameter and returns the CNT of a counter.

	A	B	C	D	E
1	INC	RESET	CNT	MAX	COUNTER
2	0	0	=IF(RESET, 0, IF (INC, IF(CNT=MAX, 0, CNT+1) , CNT))	15	=lambda((INC), CNT)
3	1				
4	=COUNTER(A3, MAX= 31)	=A4	=B4	=C4	=D4

If a lambda is used in a different execution policy than its definition environment, as in Table 2-7 (a shift register going to the right is the tell-tale sign of non-blocking mode), then we need a natural way to resolve the issue of stepping the internal state when applying a lambda defined in a different execution policy. One implementation of this requires construction of a new cell manager and modification of its state during lambda evaluation. Even though it is invoked within a non-blocking assignment, the counter from Table 2-7 is captured as reading order blocking assignments so that the reference to CNT in the lambda expression, in cell E2, refers to the value of CNT evaluated at that point during a step. When A4 is first evaluated, a cell manager with a reading order execution policy is spawned with all of the lambda's ancestors in the same reading order as the lambda definition. The MAX signal is assigned an optional parameter and the Reset signal remains bound to the place the function was defined. After the initial construction, the cell manager is stepped each time the COUNTER() call is evaluated and the count is returned.

Table 2-8: The expansion of the COUNTER call in Table 2.7. The third row is allocated to store the state required for cell A4. The cells in the third row evaluate in reading order and return E3 whenever A4 is evaluated in its non-blocking execution policy.

	A	B	C	D	E
1					
2	***INC***	***RESET***	***CNT***	***MAX***	***RETURN***
3	***1***	***=Table7!Reset***	***=IF(RESET, 0, IF (INC, IF(CNT = MAX, 0, CNT+1), CNT))***	***31***	***=CNT***
4	={ Reading Order Blocking Step (A3:E3) and return the value of E3}	=A4	=B4	=C4	=D4

By constructing and modifying state when we apply a lambda, we have implemented a monadic lambda in Python. We have defined a class called Lambda with a __call__ function interface. When a lambda is defined, all of the ancestors of the lambda expression are collected into a prototype cell manager and the execution policy is recorded. When the call method is invoked, the calling cell is asked to step the cell manager for its internal state. A new cell manager is copied from the prototype if the cell manager does not exist. This cell manager is stepped and the output value is returned. Figure 2-3 shows a diagram of this process.

For an asynchronously defined lambda, stepping the spawned cell manager will evaluate and return the lambda expression, though the cell manager will respond asynchronously to changes in its inputs and thus will settle to a value without requiring this additional step signal. Asynchronously defined lambdas still have propagation delays, if we use such lambdas in non-blocking or blocking assignments, we will need to disable the step signal of the assignment until the asynchronous cells are idle and thus the combinatorial circuit has stabilized. The idle()

function applied to a cell or set of cells returns true when the cells have no ancestors in the FIFO schedule queue. When we invoke an asynchronous lambda, we can use idle to control a clock enable signal of an asynchronous ring oscillator as in Table 2-4 thus allowing the asynchronous cell function to settle before a non-blocking step occurs.

This implementation of lambda is useful and convenient; though it has the unfortunate property that the state of the created cell manager is updated between and during cell evaluation cycles. For compilation of such structures to static code, we shall require hidden cell manager objects to be flattened-out and converted to the same execution policy as the invoking environment so that there is no longer hidden state. To do this we will require transformations among asynchronous, blocking and non-blocking modes.

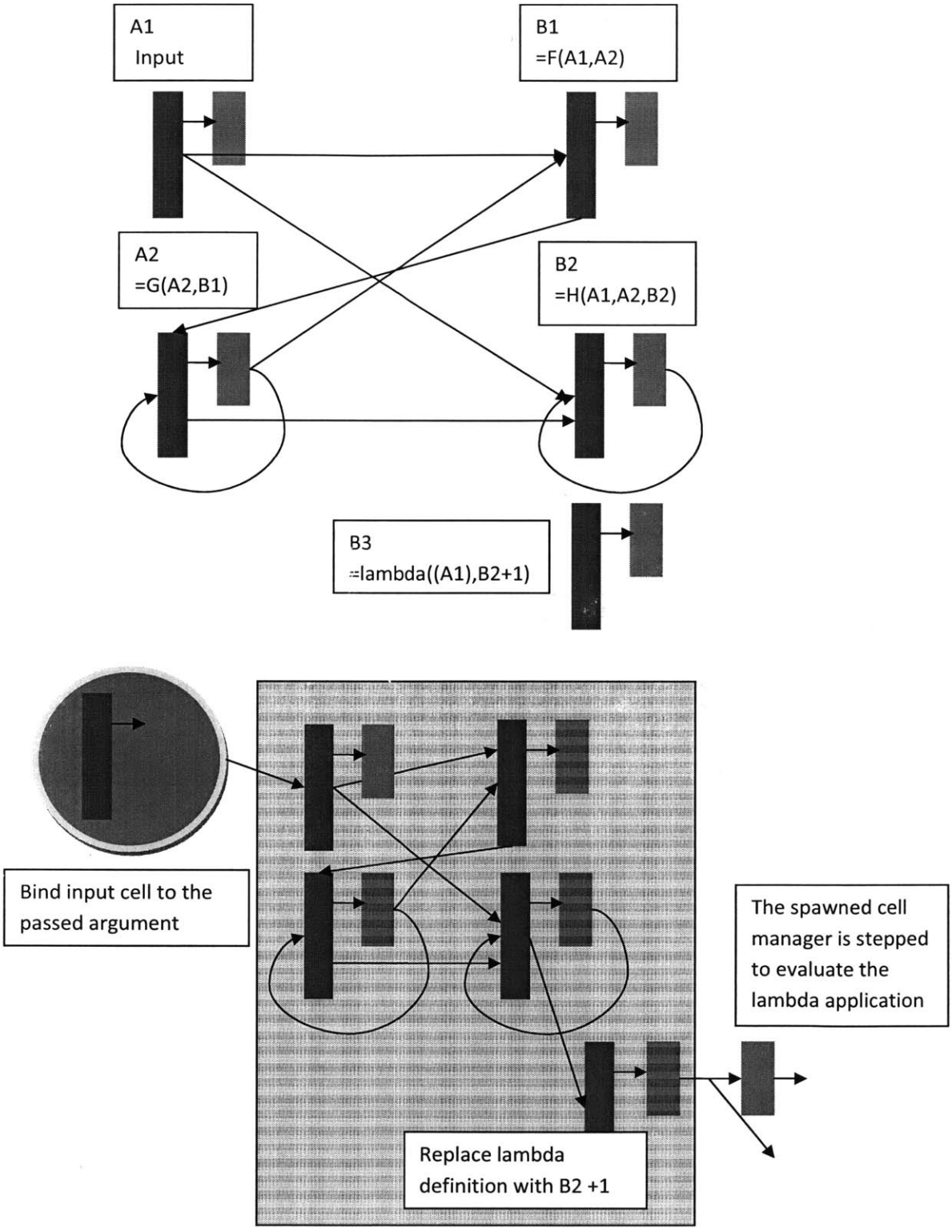


Figure 2-3: State machine diagram of lambda definition and application. Applying a lambda replicates the cell manager of the lambda definition and rebinds the input cell. Evaluation of lambda requires the internal cell manager to be stepped.

2.1.5 Translating and Combining Execution Policies

When we introduced the non-blocking execution policy, we showed it implemented by storing a list of next values to assign to the cells. Another way to implement multiple non-blocking and blocking cell managers sharing a step signal is by merging sheets together while preserving synchronization semantics. Figure 2-4 shows a pair of blocking-assigned cells as a state machine with a state register and a combinatorial next state formula. In blocking assignments, whenever a cell reads from a cell before it in the reading order, it is reading values from the output of the next state logic function. Whenever a cell reads from itself or later in the reading order, it is reading the data stored in the state register from the previous iteration.

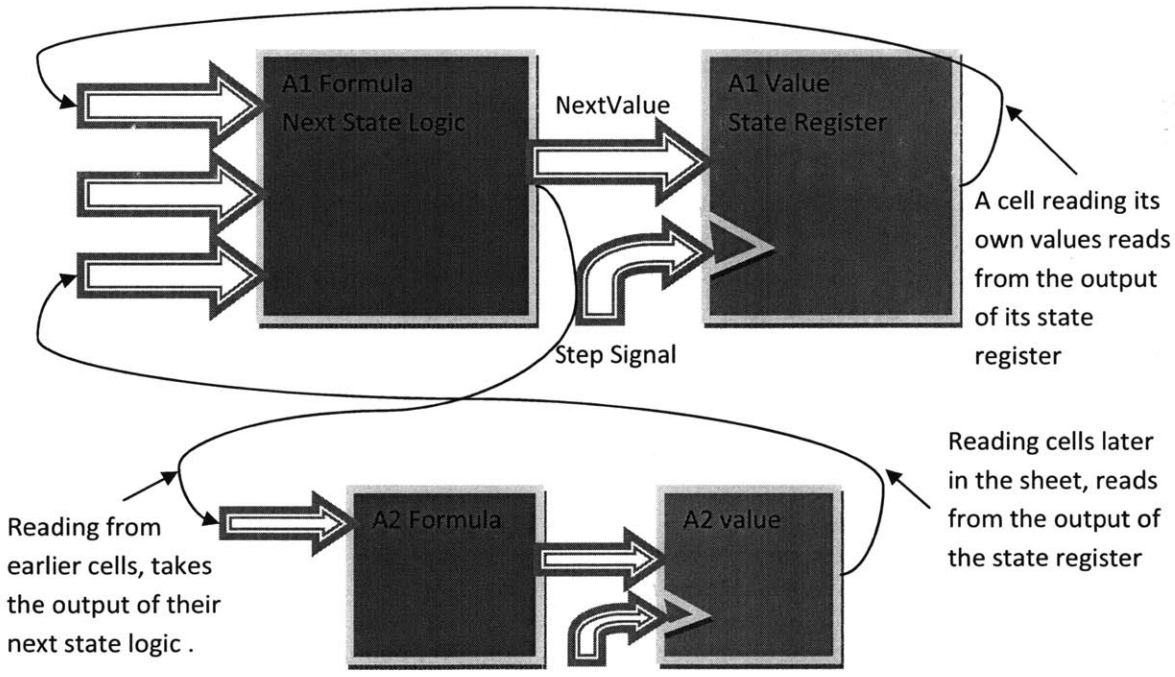


Figure 2-4 Understanding blocking assignments. In a blocking interpretation, whenever a cell refers to itself or cells after it in the reading order, it reads the cell value out of its state register. Whenever a cell refers to cells earlier in the reading order, it will read its value from the output of its combinatorial next state logic. In non-blocking assignment interpretation all cells read from the stored value output of the state registers. Converting blocking to non-blocking will merge together formulas so that they are only read from the output of state registers.

Conversion of the blocking assignment of Figure 2-4 to a non-blocking policy requires that we replicate the A1 formula and compose it with the A2 formula. This conversion is shown in Figure 2-5. Merging the two combinatorial formulas into a new combinatorial formula is done using argument substitution to produce a cell with an equivalent step response, though following a non-blocking sheet interpretation.

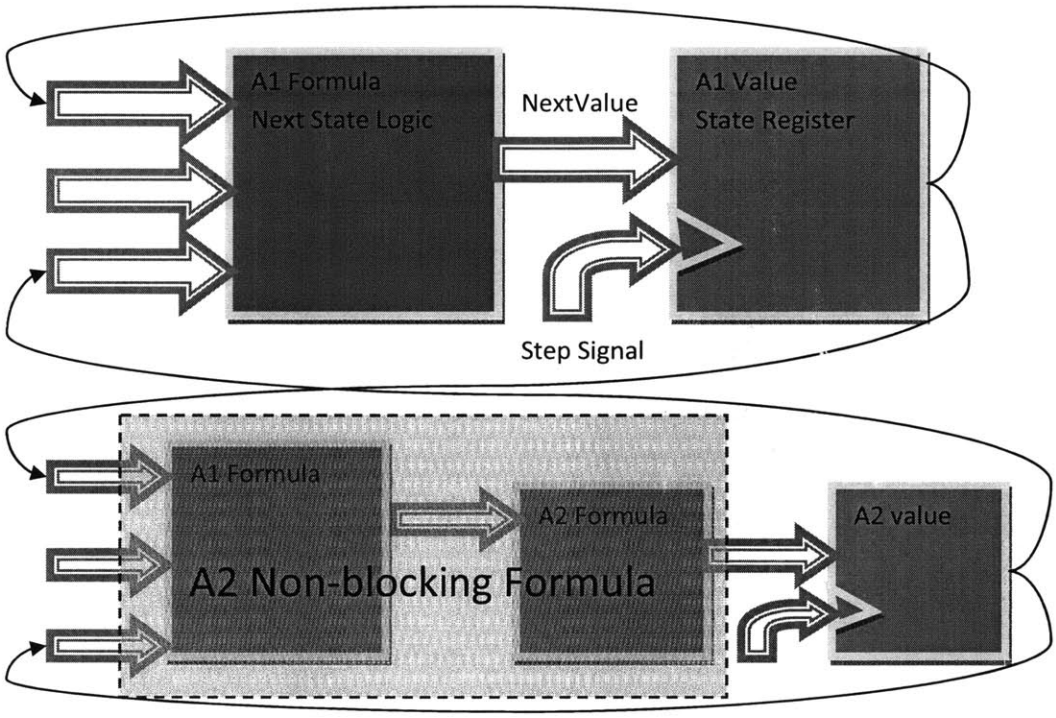


Figure 2-5 Conversion of the blocking assignments in Figure 2.3 to a non-blocking mode. A2 was dependent on the NextValue signal, so the blocking formula for A1 is replicated in A2s non blocking formula so that all dependencies are on the output of state registers.

To perform the inverse conversion, converting non-blocking assignments into reading order blocking assignments, we create cells to hold the current value and compute the next values from these. Table 2-9 demonstrates this conversion. The first row of Table 2-9 is the non-blocking interpreted shift register. The third through sixth row produces equivalent step behavior when following a reading order blocking interpretation. Earlier in Table 2-2, we showed the same shift register could be implemented in blocking mode using assignments from

right to left and thus using only four cells. We will have to apply optimizations to these assignment chains to recover the efficiency of having fewer cells. These optimizations will be explored further in chapter three and four.

Table 2-9: Conversion from non-blocking assignments to blocking assignments. Row 1 is a non-blocking shift-register. Row 3 to 6 is the same shift register transformed to blocking assignments. A "current" and a "next" cell are created for each cell of the original. The next cell assignments on row 4 are computed before the current values are assigned on row 6. The next formulas are the original equations from row 1, with all references changed to read the current cell value. All current values are assigned in row 6 to the result of the next calculation.

	A	**B**	**C**	**D**
1	=IF(Reset,0,A1+1)	=A1	=B1	=C1
2				
3	**A1Next**	**B1Next**	**C1Next**	**D1Next**
4	=IF(Reset,0,A1Current+1)	=A1Current	=B1Current	=C1Current
5	**A1Current**	**B1Current**	**C1Current**	**D1Current**
6	=A1Next	=B1Next	=C1Next	=D1Next

Converting non-blocking and blocking execution policies to asynchronous requires state registers to be implemented as sequential circuits with circular reference as in the registers of Table 2-4. The step signal from the non-blocking or blocking policies can be used to clock the asynchronous register. Ordinarily our step signals are not edge-sensitive since they may not even be Boolean typed. A dual-edged register can be used to maintain behavioral equivalence. Alternatively, posedge() and negedge() functions can divide a clock signal by two so that steps occur only on rising edges, though posedge(clock) creates a different step signal than clock, so we must use only one if we want to maintain synchronization of cell managers as built into the scheduling FIFO. It is also possible to modify the scheduler to manage predicated step sensitivity. We shall explore the use of such "guarded atomic actions" in the next chapters.

The conversion from asynchronous execution to blocking or non-blocking mode is trivial: an asynchronous sheet may iterate in any order and step when any cell changes. This conversion introduces different delay semantics than the scheduler so non-convergent circular referent structures may translate awkwardly, such as the ring oscillator of Table 2-4. Purely combinatorial circuits without circular reference are directed acyclic graphs and can be directly converted to a sheet of blocking assignments with no previous-iteration dependencies by using a topological sort instead of a reading-order sort. By topologically sorting and converting sheets to blocking assignments, we may extract pure-functional, stateless, lambdas out of sheets defined as pipelines. This allows us to avoid the use of the idle() predicate to enable a clock for combinatorial asynchronous circuits.

In order to merge two sheets together that share a step signal, we must take care to preserve synchronization semantics. Since the interpretation of non-blocking and asynchronous policies is independent of cell locality, it is possible to simply merge two sheets in any order. Synchronized blocking assignment cell managers have a dependency on reading order so merging two blocking cell managers with shared dependencies requires care. Whenever blocking assignments have a dependency in another blocking cell manager, these values must be read from the previous iteration to preserve the synchronization semantics of independent blocking assignments. Any such dependency may be resolved by preserving a copy of the previous state values using the "current/next" transformations of Table 2-7. By creating a new cell to hold the value from the previous iteration, we can merge separate blocking assignment cell managers into one blocking assignment.

2.1.6 Conclusions on the RhoZeta Spreadsheet Interpreter

The RhoZeta spreadsheet interpreter is designed to be an easy to use simulator of synthesizable Verilog with options for interpreting a set of cells. So far we have introduced event-based asynchronous, non-blocking and blocking assignments and showed how execution policies can be transformed to other execution policies. We have also introduced the lambda operator which captures the behavior of a set of cells and showed how to resolve lambda application by creating internal cell managers. Transformations between blocking, non-blocking and asynchronous assignment modes allow us to convert the internal state-machine spawned by a lambda and merge all sheets into one sheet for compilation. In the next two sections we will explore execution policies which dispatch compilation scripts for C and Verilog and execute multithreaded executable code on a dual-core and an audio callback in a Cell SPE. In section 2-4 we will explore a motif for building structures with static cells.

2.2 Cubes: a MIDI-Controlled Modular Synthesizer in a Spreadsheet

Before there was a spreadsheet compiler there was a synthesizer called cubes. Cubes was originally a C program built in Linux using JACK for Audio, ALSA for MIDI, and OpenGL for GPU graphics. The structure of cubes was gutted piece by piece until the only C code remaining was a dynamic linker controlled from a Python script. The first execution policy compiled a spreadsheet to C as blocking assignments and dynamically linked the callback functions for the JACK and OpenGL threads. A generic pthread dispatcher for synchronized blocking assignments was also built using the same technique. When the Playstation 3 came out with Linux compatibility, a port was made to execute the blocking assignment callback in a Cell SPE thread to perform the audio callback. The Cell processor does not use its SPE's with the pthread

structure and an efficient global memory synchronization protocol is still not established. Compiling RhoZeta cell managers for the Cell will be discussed in 2.2.4.

Cubes was a simple square-wave software-synthesizer. In addition to producing sound from MIDI events, the MIDI controllers can be linked to the variables of an OpenGL scene consisting of a 2-D array of cubes each with a modifiable height parameter. Every variable in the program was eventually replaced with a global variable linked to a cube height value and all of the code looked like as in Listing 2-3. The audio callback occurs 48000 times per second, but the render callback occurs as rapidly as possible. Thus the cube height values at any time update much more rapidly than they are rendered. The height of the cubes in the array controls the parameters of the program: filter parameters, oscillator effects, echo and even the color, position, rotation and lighting of the cubes. Using a MIDI controller or a keyboard, the user can select and modify the height of a cube or link it to a MIDI controller. Note-on and note-off MIDI signals produce a procedurally generated square wave.

Listing 2-3: Square Wave Generator: (cubes.c). The original decomposition of the cubes synthesizer resembled this structure. The audio callback executes this code and reads cubes[3][0] for playback. All cubes are floating point typed.

```
void squarewave (){
if (cubes[1][0]) { cubes[1][2] = 0 }            //if reset  i = 0
if (cubes[1][2] > cubes[1][1])                  //if i > period
      cubes[1][2] = cubes[1][2] - cubes[1][1];  //i = i - period
else
      cubes[1][2] = cubes[1][2] + 1;            //else i = i + 1
if (cubes[1][2] >= (cubes[1][1] * cubes[1][3])) // if i >= period*pw
      cubes[3][0] = 1;                          // sq = 1
else
      cubes[3][0] = -1;                         // else sq = -1
}
```

2.2.1 Sound Synthesis in a Spreadsheet

Every synthesizer starts with a MIDI thread and an audio callback. A MIDI event polling thread processes MIDI events into a global arrays of controller values (CV[num][chan]) and note

velocities (NV[num][chan]). Note on and note off signals add or remove note numbers from the global activenotes linked-list. Without dynamic recursion we cannot spawn an oscillator for each active note, so the synthesizer must have fixed polyphony. An audio callback provides a buffer of input samples and requires output to be written to a buffer of output samples. The JACK server is invoked with the sample rate, the number of buffers and the number of samples per buffer. Usually a buffer is 32, 64 or 256 samples, and 2 to 4 buffers provide data to a sound card. At a sample rate of 48 kHz, a 256 sample latency introduced by 4 frames of 64 samples is only 5.3 ms, and is unnoticeable. The JACK client must provide a callback function that is passed a variable with the number of samples needed to fill the output buffer. The type of the inputs and outputs are PCM samples of floating point or integer formats.

Table 2-10: A square wave, a saw wave, and a sin wave. This implementation of these oscillators will generate alias noise. The execution policy of Audio_Out captures the finite state machine lambda(()Saw) and compiles it to a synthesizer callback that gets dynamically linked to a running JACK client.

	A	B	C	D
1	Reset	Period	I	Pw
2	0	50	=IF(reset,0, IF(i >=period,i-period,i+1))	0.75
3	Sq	Saw	Sin	
4	=IF(i >= period * pw, 1,-1)	=2*(i/(period) -0.5)	=SIN(2*PI * i / (period))	
5	3.14159		=Audio_Out(Sq)	

The buffer callback model is too high-level from a sound synthesis perspective. We would prefer to think of our synthesis callback like a cell manager that is stepped every sample period and whose outputs are written to the audio buffer. We create a global state-modifying function that is iterated to provide the number of samples required. An example of such a stateful oscillator with square, saw and sin waves, is shown in Table 2-10. A graph of these waveforms is provided in Figure 2-6. When the Audio_Out function call is executed the implied

instruction is to "compile in reading-order, as floating point cells and return sq to audio callback." An audio_out execution call can be added to the FIFOSchedule and be set to update whenever the audio sheet is changed.

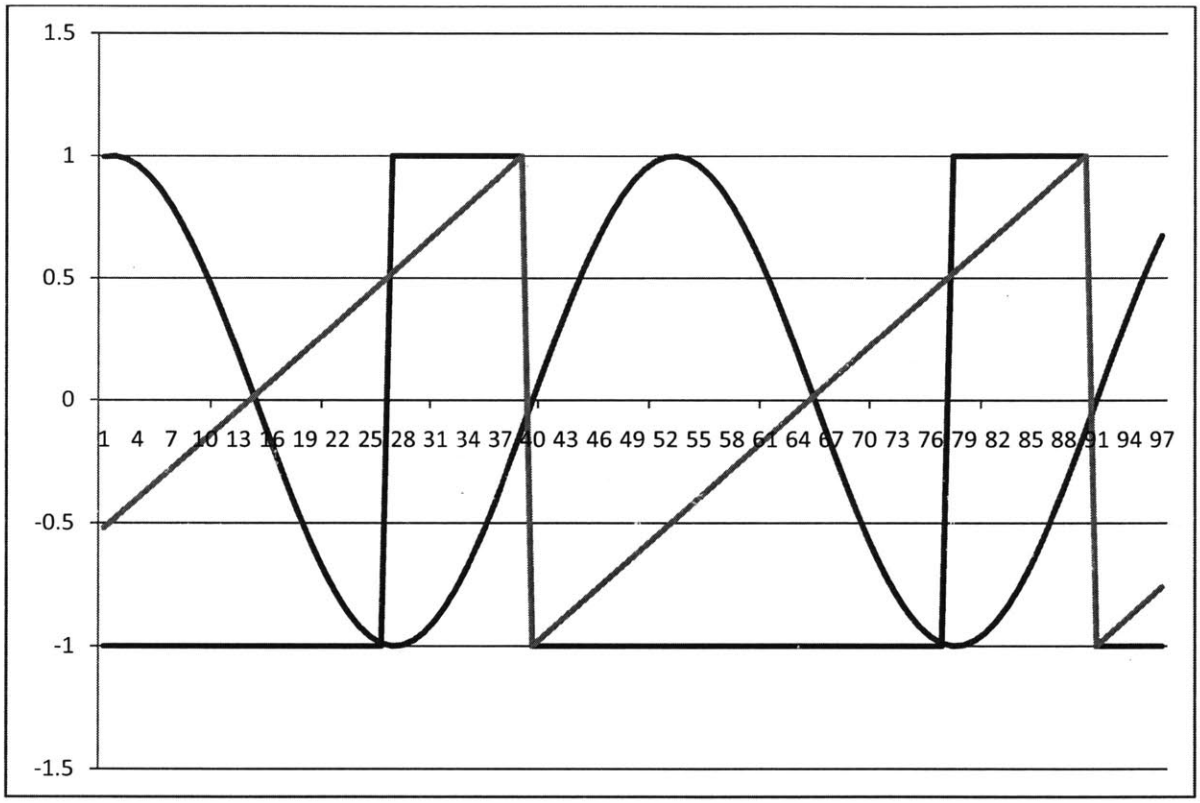


Figure 2-6: The square, saw, and sin waves generated in Table 2-8 as rendered by Excel.

The code emitted by compiling audio_out of Table 2-8 creates the same function as Listing 2-3, except that a global state pointer (void * cubes) is passed as an argument to the function and the function returns a floating point output corresponding to the square wave output. The pointer must be passed in because the dynamically linked object is compiled without a global context so to resolve the variable cubes we must provide it with a pointer to the cubes array. Cell names are used instead of "cubes[x][y]" references in the actual function. A "#define" preprocessor macro assigns names to locations in the cubes array (this was easier than

parsing and translating all the variable names in the code). Cell formulas are converted to a semi-colon delimited listing of cell assignments. A preprocessor in Python finds "IF" functions and converts them to the proper C syntax (for example, = becomes ==). Before compiling and dynamically linking the code it is wrapped with the necessary preprocessor directives, function prelude and coda. Listing 2-4 shows what this code structure looks like.

Listing 2-4: An example of the emitted code from Audio_Out(sq). I have added spaces and tabs to make it readable.

```
#define reset cubes[0][0]
#define i cubes[1][0]
...
float * audio_out (float * cubes){
if (reset) { i = 0 }
      else { if (i>period) { i = i - period; }
                  else { i = i + 1; }}
if (i>=period*pw)) { sq = 1; }
      else { sq = -1; }
return sq; }
```

A reading order C execution policy was created for the OpenGL rendering callback in addition to the audio callback so that a list of rendering instructions and transformation matrices could be built into a spreadsheet and dynamically linked to the GPU render callback. OpenGL also provides global variables for mouse and keyboard events. Physics equations can be built in so that a spring-like response to note velocity can be observed and used to envelope the filter and the audio-out can be graphed as an oscilloscope as in Figure 2-7.

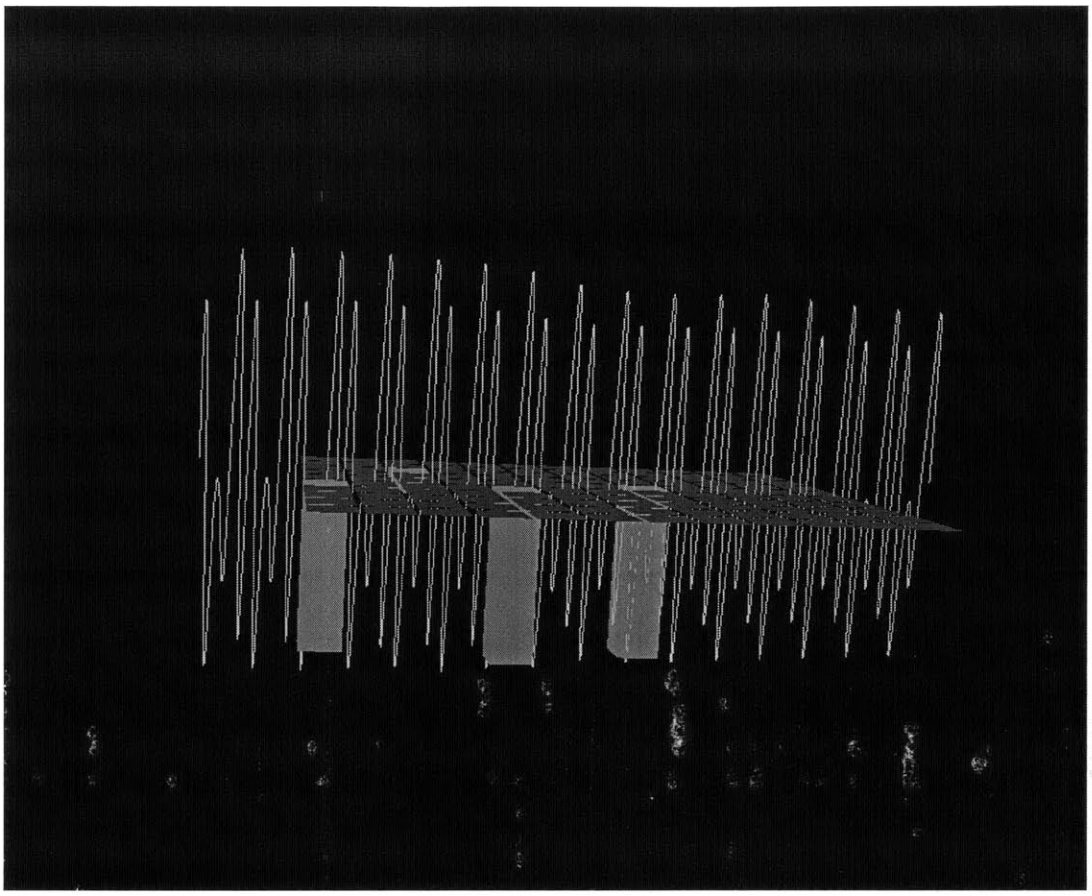


Figure 2-7 Cubes synthesizer with an oscilloscope. Each rendered cell's height corresponds to a MIDI Note Velocity. The oscilloscope is generated using a memory macro.

Currently, there is no RhoZeta execution policy capable of dynamic recursion[2] in C or Verilog compiled code since this generally requires dynamic state management which is the duty of an OS. Application of a lambda in the current C implementation issues a fault which must be handled by Python. All lambda applications must be flattened using the transformations from the previous sections and recompiled in C. A function compiler execution policy, though not yet developed, could produce a C functions out of GCD in Table 2-11; such code in a spreadsheet is

[2]When we evaluate a lambda we must spawn a new set of cells if they don't already exist. The current cubes cell abstraction does not permit a cell to contain this hidden state. The current attempt traps to a dynamic allocation function which allocates cells for lambda application from a global cell space. This flattens all stateful lambdas upon application. We must also free this state when the recursive path is no longer followed.

equivalent to a C function under change of syntax. For any C execution policy, types may be specified as in the square and GCD function definition with a preprocessor statically typing the bindings. The audio_out and openGL execution policies implicitly bind variables automatically for us. Without such type definitions, every variable is interpreted as a void pointer and type casts are required in the cell equations in order to use floating point arithmetic. This structure has a limitation that void pointers on a 32-bit architecture cannot cast adjacent cells to 64-bit floating point types. A better type system could simply infer everything using type propagation.

Table 2-11: Type declarations are processed in the obvious way. Currently square flattens out correctly, but GCD does not yet compile to C since we reject any recursive function invocations in static mode. Type properties could also be inferred from the spreadsheet interface or the Python object system. To propagate types through monadic lambdas, Haskell's type system could be used.

	A	B	C
1	setType(x, float)	X	Square
2	setType(square, float)	0	=lambda((x),x*x)
3	setInt(A4,B4,GCD)		GCD
4	10	8	=if(B4=0,A4, gcd(B4, A4%B4))
5			=lambda((A4,B4),C4)

2.2.2 Changing the Circuit with the Power On

It is possible to put the execution policy responsible for compiling and linking OpenGL and audio callback functions in the FIFOscheduler and having it re-execute whenever the tables are modified, for example when an IIR filter is added to the audio signal path, or when links to MIDI controllers are added. Since the compilation and dynamic linking of short code executes fast, it has the effect of modifying the synthesizer circuit in the spreadsheet while the power is on. However, since we must flatten all stateful recursion into one sheet when we compile, this model for dynamic update does not scale well as our audio pipelines get more complex. An implementation of the FIFO event scheduler and lambda application are required to have a truly

self-contained dynamic C or Verilog environment. The question is where to draw the boundaries between dynamic and static execution. Ultimately if high performance pipelined functions executing on multiprocessors is the goal, static threads with bounded space and synchronized timing are necessary. These statically compiled cell managers should be thought of as static dataflow cores or kernels, and an OS must manage the global and local context each core is processing. The thread dispatcher described in the next section was a first attempt to use pthread_join to synchronize multiple sheets.

2.2.3 Multithreaded Execution in C

In addition to audio and OpenGL callbacks, a multi-threaded execution policy in C was created to synchronize multiple blocking assignments. The current multithreaded framework uses a dynamically linked thread dispatcher with a global memory space. Local state for each cell manager thread is allocated and freed when the dispatcher is linked or closed. A compute thread for each cell manager is spawned using pthread_create to compute the next state function in local space for each cell manager. When all threads are terminated (checked with pthread_join), the dispatch issues another thread to commit the local state to the global current values. This is an exact translation to C of how Python creates threads for the compute and commit phases of the synchronized assignments. The cell manager knows it's offset in global memory and commits its local data there. The current system does not respond to events and the compute-commit dispatcher loops forever until Python re-links it. Additional cell managers can be added by re-linking the dispatch function. Simple tests have been done to make sure that synchronized blocking assignment semantics are preserved by multithreading and that there are no memory leaks.

By requiring structural recursions to be unrolled, we have a fixed number of cells to execute. One may think the emitted C code has the property that it executes in static space. However, since our compilation macro is a syntactic effect, it is possible to have a cell contain a malloc call to allocate memory and write to it using memcpy as in Table 2-12. A lambda macro captures this state-machine for reuse. This was used to create an echo effect and the oscilloscope shown in the screenshot in Figure 2-7. Before the audio callback and OpenGL callbacks are updated to reference the allocated space, a delay buffer must be allocated and a pointer stored in a specific cell by executing a C thread as shown in Listing 2-5. The audio callback fills the delay buffer with a sample and the OpenGL buffer renders a point for the last 300 samples in the delay buffer. This approach could be used to generate state structures for dynamic stateful lambda recursion, though this is a work-around rather than a solution to the "memory/cell allocation" problem.

Table 2-12: A memory state machine. Running this thread allocates a buffer for 96000 floating point samples and stores it in cell A2. Cell A1 is a state register and frees memory on reset. The implementation of free returns 0 after executing the traditional free and ignores null pointers. Care must be taken to avoid segmentation faults and memory must be freed to avoid memory leaks. Read before write semantics apply and the written data may be modified after read, though addresses must change because of reading order interpretation.

	A	B	C
1	=IF(RESET,State=0,IF(State=0,1,2))	0	96000
2	=IF(State=0,free(A2),IF(State=1,(float *) malloc(C1*sizeof(float)),A2)	0	
3	=IF(State=2,A2[B1],0)	0	**Memory Macro** (for reuse)
4	=IF(State=2, memcpy(B3,A2+B2,1),0)		=lambda((C1,B1,B2,B3), A3)

Listing 2-5: The cell manager for this buffer turned into the C code. Formatting and comments were added.

```
#define RESET cubes[0][0] //global closure
#define Row   10
#define Col   10     //where the application flattened to in the global sheet
#define Rows 4
#define Cols 3
#define State local[0][0] //local state for everything that is written
#define B1    cubes[0][1] //input of lambda application read from global space
...
void * MemoryCompute(void * cubes, void * local) { //function prelude
if (RESET) { State = 0; } else { if (State==0) { State = 1; } else { State = 2; }}
if (State==0) { A2 = free(A2); }
else { if (State ==1) { A2 = (float *)malloc(C1 * sizeof(float)); } else { A2 = A2; }
...
pthread_exit(NULL) //thread coda }

void * MemoryCommit(void * cubes, void * local){
int i,j;
for (i = 0; i < Rows; i++) {
       for (j = 0; j < Cols; j++) {
              cubes[Row+i][Col+j] = local[i][j];}}}
```

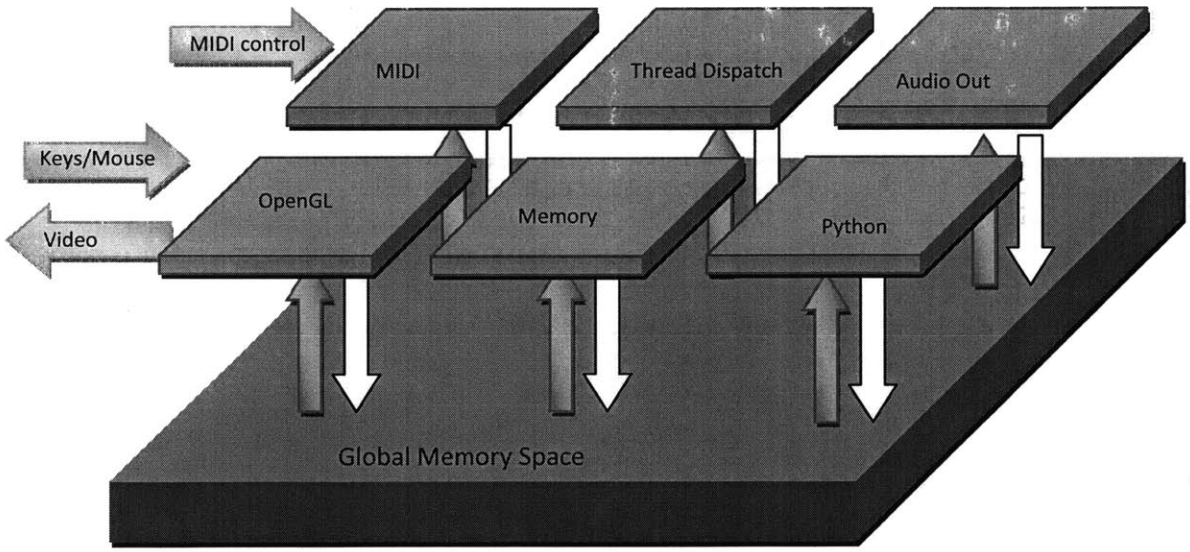


Figure 2-8: The synchronization model used for multiple blocking assignments requires values to be committed to shared global memory after each commit step. The cell manager threads should be thought of as stateless combinatorial functions acting on global memory.

This synchronization protocol requires a thread to commit local state to global memory each commit phase. This is the fundamental semantic of the language, and allows the compute phase to be "stateless" as if the local state just represented values on a wire and commit

represented the synchronized latching of a set of registers in a global address space as depicted in Figure 2-8. The effect of committing local state to global memory may not be costly if a memory subsystem is designed to distribute global memory shared by multiple cores efficiently. It may be necessary to synchronize data with off-chip memory when multiple processors must communicate over shared global memory. An efficient system is required to handle memory synchronization between threads in the Cell processor in which direct communication between cores is optimal.

2.2.4 Compilation of Spreadsheet Cells for Cell SPE

A script to produce SPE cores from the oscillator cells in Table 2-8 was produced while learning the SPE thread API[3]. The SPE uses mfc_get to get the nextvalue array, iterates the audio generation function and produce several samples of audio output. The audio output and the current cell state is transferred back to the JACK callback in the PPE using mfc_put before the SPE thread terminates. A similar framework for launching and synchronizing threads from a spreadsheet on the Cell is still under development.

Committing all internal state of a SPE core to global memory in the PPE each step is wasteful in chip multiprocessor arrays. If a cell manager only synchronized values for cells with external dependencies, then we would be more efficient with global memory bandwidth. On the cell processor, it is possible to use the element interconnect bus (EIB) to transfer data directly between SPEs. When we write code like Table 2-13, an OS should locate the square wave generator and the IIR filter in separate SPEs directly streaming data from core to core over the EIB as shown in Figure 2-9. The EIB is a ring, so for most acceleration applications on the Cell

[3] The SPE code was derived from the demos of the IAP Playstation 3 programming course.

it is more efficient for each core to stream packets of data directly to other cores over the EIB as soon as the data is ready and independent of global data synchronization.

Table 2-13 Specifying a Synthesizer with a sawtooth wave oscillator and an IIR Filter.

	A	B	C
1	=SawTooth(ActiveNotes)	=IIRFilter(A1,Cutoff=MIDICV[1][1])	=Audio_Out(B1)

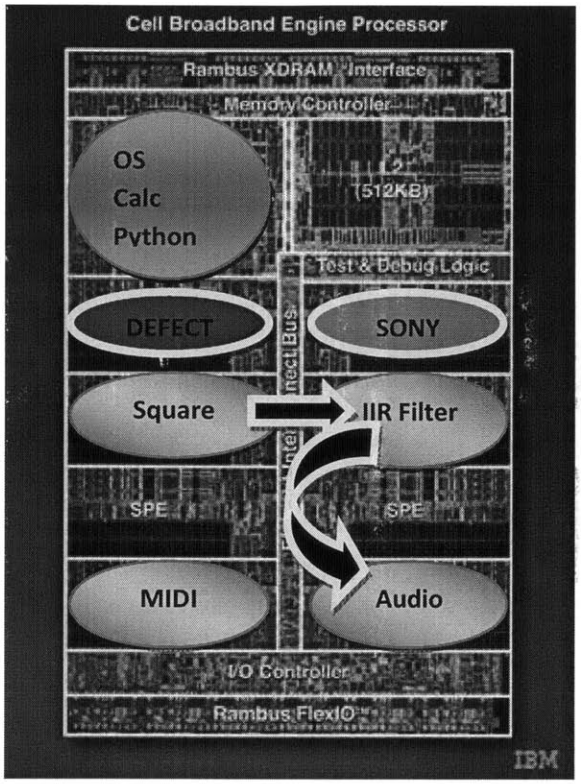


Figure 2-9 Stepping a square wave oscillator 64 times in a SPE takes under 200ns at 3.2 GHz. If properly vectorized, four oscillators can be stepped at once in one SPE. We extrapolate that it is possible to generate polyphony of over 26000 square oscillators per SPE at 48 kHz sample rate. (Cell Graphic Source: Wikipedia)

For most multiprocessor architectures it is good to think in terms of streaming kernels executing some component of a larger non-blocking assignment pipeline with coarse-grained synchronization. If a pipeline is a directed acyclic graph with latency slack as in the synthesizer

in Table 2-11, then the pipeline may be buffered without affecting the pipeline output. This allows globally asynchronous pipelines to share data in larger bursts that occur less frequently. Buffering ports is possible even when a pipeline block has internal feedback requiring finer-grained synchronization, such as the IIR filter in our synthesizer. By incorporating acceptable latency we minimize the complexity of scheduling communication between cores on a shared bus. Buffered pipelines also allow cores to dynamically decrease their voltage and frequency if there is pipeline slack due to load imbalance.

Synchronization rate tends to be the inverse of functional granularity. Larger function blocks must be synchronized at a lower rate. Instead of thinking of 6 SPEs performing 128-bit vector operations at 3.2 GHz, it is often better to think of the Cell processor spatially as an array of 6400 simple vector pipelines each operating at 1 MHz for I/O bound multithreading, or 600 megapixel instructions[4] iterating at 30 Hz for graphics applications. With buffered pipeline communication between cores, it may be possible to achieve very close to this kind of performance. Metaprogramming shells are good for hiding the programming overhead required to develop synchronization protocols.

The current C compilation strategy in the RhoZeta interpreter works well for developing generic blocking assignment threads on a global memory space. A metaprogramming script for producing multithreaded kernels out of synchronized blocking assignment is very easy to work with, though much more work is required to develop a general system for managing and optimizing dynamic stateful lambda application and synchronization. In the next section a Verilog compilation of RhoZeta will be demonstrated in which all unexpanded lambda

[4] Instructions that operated on 1 million 128 bit pixel vectors

applications compile to clock disabling traps. Section 2.4 will discuss a reductionist model for an OS based on dynamically reconfigurable cells as a means of lambda expansion.

2.3 Compilation to Verilog

Compilation of a spreadsheet to Verilog for an FPGA is much easier than compilation to C. The Python execution policies described in section 2-1 are a framework for a dynamically typed interpreter for Verilog and transformation of blocking, non-blocking and asynchronous cell managers are directly translatable to "always @ ([step signals])" blocks. Additional parser macros allow Verilog-like bit-string reference "A[1:0]" to override the Python meaning. Types can be specified as in C, though a Verilog type is specified by using a bit width for the type with additional directives for "signed" or "unsigned." Many functions have been added to the Verilog framework and many modules have been tested on Virtex-2, Virtex-5 and Spartan boards[5]. In order to specify cells as parameters, inputs, inouts and outputs, a few additional functions were added. If unspecified, all cells with no dependencies become inputs, all cells with no dependents become outputs.

Depending the on the types you wish to use, implementations of operators may need to be added to the language. Currently supported types for converting from Python to Verilog include bit strings, integer numbers and lists of bit strings and numbers. Floating point types are not native to most Verilog implementations and are not fully yet supported in RhoZeta either, though a PDP-11 floating point adder has been built. Ordinary Verilog cannot handle structural recursion, but we have added a global trap whenever a dynamic invocation is required. The leading zero counter (LZC) shown in Table 2-14 is used to normalize floating point numbers and

[5] The 6.111 Lab assignments are good benchmarks. I have also incorporated MIDI and AC97 audio modules. These boards have all been coupled to my CPU via USB and I have not been able to test high performance hybrid multiprocessing yet. Ultimately the target application for a spreadsheet to FPGA, GPU and Multicore is for modeling and simulation applications.

was a testing benchmark for recursive macros. This implementation of LZC is not expressed in the optimal parallel way. The way it is expressed requires N steps to execute, where N is the number of bits of the LZC input bit string.

Table 2-14 A serialized expression of the leading-zero-counter. This code is intentionally bad for Verilog and compiles to a long series of if statements. Rest is dependent on Verilog bit-string interpretation of the A[a:b] syntax. We will later attempt to automate optimizations which reduce this O(N) serial operation to the correct O(lg N) parallel algorithm for LZC.

	A	B
1	00010010011011	
2		**Rest**
3	**LZC**	=lambda((A1),A1[(Len(A1)-2):0]
4	=lambda((A1),IF(Len(A1)=1,1, IF((msb(A1))=1,Len(A1)-1,LZC(rest(A1))	
5	00000100101010010101	setType(A6,5)
6	=A5	=LZC(A6)
7	***00001001010010101***	***=LZC(A7)***
8	***0001001010010101***	***=LZC(A8)***
9	***001001010010101***	***=LZC(A9)***
10	***01001010010101***	***=LZC(A10)***
11	***1001010010101***	***=LZC(A11) = 14***

A hardware trap was added to Verilog so that any cell manager may be compiled to static Verilog even if it contains dynamic lambda invocations. Since dynamic lambda invocations are always encapsulated by conditional expression, a Verilog translation raises an interrupt flag and disables the clock whenever a dynamic recursion is required. Listing 2-6 demonstrates how this works for LZC from Table 2-14. When we tell RhoZeta to construct Verilog from the LZC function call in B6, our Verilog compiler will collect only the cell managers that exist in Python. Since the LZC happened to expand internally in Python to a certain number of levels, only these objects are flattened in the Verilog. If we flatten this code entirely for a 64 bit LZC, the Xilinx tools compile this to a propagation delay of 7 ns on a Virtex 5 with full optimization. We will

demonstrate spreadsheet macros in the next chapters that transform this sheet to better Verilog achieving 3.9 ns propagation delay on the same chip.

Listing 2-6: Leading Zero Counter emitted from Table 2-13

```
module sheet1(clk, A5, B6, lambda_req);
parameter B11_TRAP_ID = 1;
input [19:0] A5;
output [4:0] B6;
output [16:0] lambda_req;
reg [19:0] A6;
reg [18:0] A7;
reg [17:0] A8;
reg [16:0] A9;
reg [15:0] A10;
reg [14:0] A11;

always @ (posedge clk) begin
if (A6[19] == 1)
      B6 = 18;
else begin
      A7 = A6[18:0]
      if  (A7[18] == 1)
            B6 = 17;
...
else begin
A11 = A10[14:0]
if (A11[14] == 1)
B6 = 13
else begin
Lambda_Req[16] = 1;                  //make lambda request
Lambda_Req[15:0] = B11_TRAP_ID       //trap id!
...
```

So far this lambda_req trap has not been used for much more than simple testing purpose. This is a way of making a request for reconfiguration; currently there is nothing to answer that request. In the C compiled code, the trap to Python could cause the interpreter to unroll more C code for execution. Since Verilog synthesis, mapping, place, route and device programming takes a long time for large designs, it is not practical for a device to request this sort of reconfiguration. A lower level model of reconfigurability must be built into the language to manage stateful recursion when it is necessary.

2.4 Dynamic Low Level Compilation and Reflection

The goal of this section is to construct a model for describing low-level compilation to reconfigurable tiled arrays. A low level macro editor for configuration layers should support dynamic dispatch to hardware; rather than reverse-engineer existing FPGA architectures' bit-streams, I have built my own FPGA models within a spreadsheet[6]. Configurations are stored in memory and we are able to modify them dynamically like any other data. Once we have a low-level model for a tile we will implement a crossbar model to reconfigure the control bits. We can attach the crossbar reconfiguration controller to the actual programmable environment thus achieving reflection. We will also attach a reconfigurable array to a RISC processor so we can use a software model for FPGA control. When a user alters a spreadsheet cell representing the configuration data at a specific chip location, the RISC reconfiguration controller controls the crossbar array to implement the change in hardware. I have yet to successfully compile a spreadsheet to run in an FPGA expressed inside a spreadsheet.

2.4.1 Primitive Languages for Reconfigurable Arrays

So far we have used inherited similarities between Python, Verilog and C to create syntax transformation macros. What is necessary is to close the gap to reconfigurable computing on spatially assorted tiles is a primitive language and a means for reflecting dynamic cell allocation into the hardware. In the dynamically typed spreadsheet we used a stateful lambda evaluation function to take care of issuing step calls to internal state. In a low level compiler, we need some way to allocate cells and modify formulas of a statically typed array. What we want is to have the ability for a few special cells inside of the spreadsheet to control the formulas in other cells.

[6] One result of a spreadsheet programming environment is that designing an FPGA takes 10 minutes.

Once we have access to an internal configuration ports then we have a self-reconfigurable architecture capable of handling lambda application via structural recursion. The Xilinx FPGAs used to prototype RhoZeta have internal configuration ports though the Xilinx documentation informs you of their existence with a warning [23].

Before attempting automation, developers must have hands-on understanding of what they are automating. To build a model for a low level compiler for tiled arrays, we will consider how to characterize the types of primitive tiles that make up a logic array. There are a number of different types of universal cells and many devices support multiple different kinds of primitive operations and routing modes for connectivity. For an FPGA, the most primitive language is to think of tile configuration bits as specifying look-up-tables or connection bits operations for a router. Rather than worrying about the routing constraints or primitive configurations we assume that an architectures' primitive operation have constrained addressing modes that limit us to a certain range of cells for operations and that an auto-router is required to determine whether or not a low level spreadsheet meets routing constraints. Additionally, cell assignments may be clocked or asynchronous and this also needs to be incorporated into the addressing mode of a primitive language.

It is possible to have a purely functional array without any routing. For example, each logic block may be defined as four 4-LUTs taking a single bit of input from each of its four neighbors and producing a single bit of output for each of its neighbors. This structure is shown in Table 2-15. The LUT4 function takes as arguments a 16-bit string and 4 inputs and converts returns the bit indexed by the bit inputs. By making the LUT configurations reconfigurable as we have made loadable registers, we can produce a reconfigurable array. If we constrain our implementation to associative operators we can reduce the lookup table to 3 bits each.

Table 2-15: A look-up-table based programmable logic array with a static address mode, each cell output is strictly a function of its neighboring cells. By replicating this structure and defining the LUTs we produce a universal Turing machine. Cell C3 demonstrates how to make the lookup table reconfigurable. It is useful to create a separate sheet storing recon_addr and recon_data.

	A	B	C	D
1			=LUT4(C2, D2,C4,A3,B1)	
2	=LUT4(B2, B1,D2,C4,A3)	1111000011110000	1001001000101011	
3		0001000000000000	=IF(reset,0000000000000000, if(recon_addr='C3' recon_data, C3)	=LUT4(C3, C4,A3,B1, D2)
4		=LUT4(B3, A3,B1,D2,C4)		

Alternatively it is possible to construct reconfigurable arrays in which the cell function is fixed and programmability created by modifying routing. Table 2-16 demonstrates a "MUX" array tile in which the only function is a MUX and the routing may specify a set of offsets for reading the inputs. These primitive array structures are easy to experiment with and may have potential for interesting nanometer scale assembly techniques, as well as optical systems.

Table 2-16: An OFFSET-MUX tile. By specifying the offset for the mux operation we create a multiplexer with programmable inputs. This structure can be implemented using adiabatic logic with power clocks and transmission gates.

	A	B	C	D
1	Down	Right		
2	0	0	=OFFSET(D3,4*A2,4*B2)	
3	0	0	=OFFSET(D3,4*A3,4*B3)	=mux(C2,C3,C4)
4	0	0	=OFFSET(D3,4*A4,4*B4)	

2.4.2 Re-Configuration Macros and Reflection

In addition to primitive tiles structures there are a number of modes for reconfiguration. For example, Xilinx Virtex chips have their configuration shifted in serially as a column. A

common feature of spreadsheet solutions is that the solution is usually reduced to a single replicated formula. Thus an interesting reconfiguration method I have experimented with uses a shift register to drive a crossbar row and column select signals for a reconfiguration buffer. This allows us to select a group of cells, and then select all cells shifted by some offset. First we create a crossbar structure as in Table 2-17. In a table we will create another symmetric array which checks if its column and a row are selected and if the reconfiguration signal is true. If so, the cell loads a new configuration from a specified value. Algorithms will often require multiple common structures replicated regularly; by using this reconfiguration control structure to select rows and columns we can spawn multiple copies of the same structures simultaneously.

Table 2-17 A 3x3 cross-bar is used to select a row and column for configuration. Row 1 and column A are shift registers so we can create multiple copies of the same thing simultaneously. Interpret this table with non-blocking semantics.

	A	B	C	D
1		=IF(ShR,Rin,B1)	= IF(ShR,B1,C1)	=IF(ShiftR,C1,D1)
2	=IF(ShC,Cin,A2)	=AND(B1,A2)	=AND(C1,A2)	=AND(D1,A2)
3	=IF(ShC,A2,A3)	=AND(B1,A3)	=AND(C1,A3)	=AND(D1,A3)
4	=IF(ShC,A3,A4)	=AND(B1,A4)	=AND(C1,A4)	=AND(D1,A4)
5				
6	**ShC**	**ShR**	**ReconEn**	**ReconData**
7	0	0	1	1111000011110000
8	**Cin**	**Rin**		
9	0	0		

Once we have a means to control the configuration bits of a tiled array, we can make the signals responsible for controlling these bits attached internally to signals in the tiled array. This reflection allows us to define a state-machine in the language of the underlying array that is capable of reconfiguring itself. Hooking up our FPGA to a RISC processor like the one from section 1.2.3 allows for even more interesting hybrid architecture exploration in which a processor uses a reconfigurable array as a co-processor. A diagram is of this architecture is

shown in Figure 2-10. We can control the configuration and data ports of the reconfigurable array as if they were registers of the RISC CPU and can consider a model for incorporating dynamic reconfigurable pipelines into RISC architecture. A low level FPGA Control API in the RISC processor allows macro control of the FPGA device and a shared memory communication scheme is used to share data between the two.

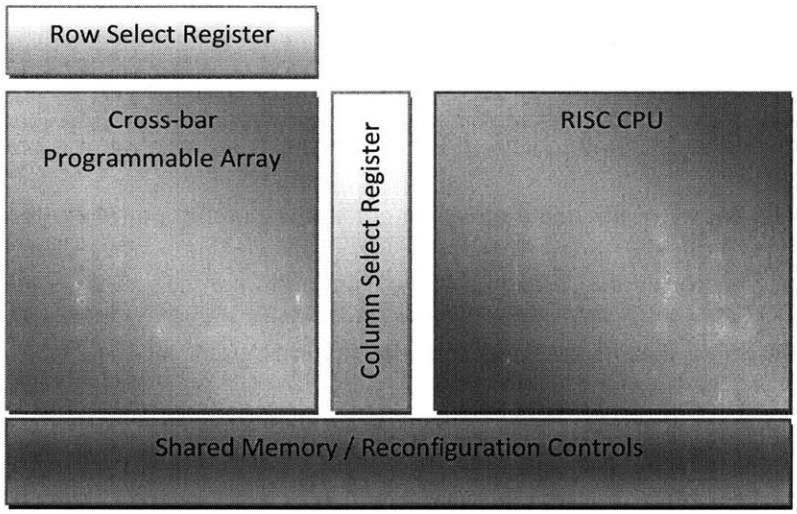


Figure 2-10 A hybrid reconfigurable architecture prototyped in a spreadsheet. The RISC CPU controls the cross-bar programmable array using row select and column select registers.

2.4.3 Conclusions and Looking Forward

Structural lambdas in a spreadsheet make it substantially easier to prototype a tiled array and design its API. Once we have a model for specifying and reconfiguring the primitive cells, we will want to develop a set of macros that allow us to allocate and manage cells for lambda expansion. While a dynamic low-level compiler for reconfigurable computing on FPGAs is non-trivial it is at least feasible. Again for FPGAs as with multiprocessors (where a good dynamic allocator is also an open problem) static optimization is critical to performance. A properly

compiled system will not require any sort of dynamic elaboration or will produce a trap to some higher level interpreter to handle reconfiguration. A generic static optimizer and profiler for parallel computations in a spreadsheet is still an interesting problem, and can be examined in an architecturally independent manner.

Architecture optimizations are possible even within the functionality of the Verilog and C compiler we already have. The next chapters will construct architectural optimization macros for spreadsheet defined architectures independent of the low level physical API implementation. Within our dynamically typed language we will construct a rule based interpreter which allows sheet polymorphism as a guarded atomic action. By establishing optimization macros as an atomic action on a sheet we will establish a means of automating architectural exploration. Behaviorally invariant transformation rules will allow us to optimize the structure of ISA emulators and apply resource sharing and fault tolerance macros.

Chapter 3

Catalyst: Optimization of Spreadsheet Architectures

"Simple things should be simple, complex things should be possible" – Alan Kay

This chapter addresses the problem of optimizing a behavior defined in a spreadsheet for power, speed, area and fault-tolerance. We shall express graph isomorphism and dynamic reconfiguration as guarded atomic actions on spreadsheet cells. Section 3.1 describes the Catalyst rule-system which dynamically optimizes functions in a spreadsheet and demonstrates how combinatorial reductions, redundant resource sharing and speculation macros can be used to automatically optimize a serially-defined leading-zero-counter. Section 3.2 will explain how an analysis of static code can often optimize out most of the architecture from an instruction set emulator and provide a model for automatic pipelining of arbitrary instruction set architectures. Section 3.3 shows how pipeline resource sharing in this context allows multiple non-blocking pipelines to share physical resources. Section 3.4 will demonstrate simple macros to automate and test for fault tolerance.

The Catalyst rule-system presented in Section 3.1 is designed to optimize a spreadsheet architecture using behaviorally invariant graph-rewriting macros on a spreadsheet. Graph-rewriting on a spreadsheet is a very useful general purpose problem solving tool and gives a UI to guarded atomic-action rule systems. The general topic of dynamic dataflow optimization has been widely studied and incorporated into a number of development environments. My implementation of a dynamic transformation system executes graph-rewrite rules as atomic-actions on spreadsheet architectures. The guarded atomic action model used to preserve

dynamism is derived from Bluespec [24]. A previous implementation of Bluespec in Scheme was used to perform pipeline resource sharing [25]. This previous work in Scheme served as the basis for Catalyst, which uses guarded atomic actions to modify dataflow architectures expressed in a spreadsheet. Section 3.2 describes how to unroll a pipeline of arbitrary instruction set emulators and is inspired by the CHiMPs compiler from Xilinx [11]. By using rule-based optimization threads to act on spreadsheet typed objects in a behaviorally-invariant way, we can express dynamic architecture transformation strategies that guarantee the functionality of our system. By applying redundant resource sharing and Boolean reduction rules we will transform a worst-case linear sequentially specified leading-zero-counter to a logarithmic parallel version.

3.1 Behavioral Invariance

We use the term behavior to mean the combinatorial function determining a cell's next state and outputs. The topological space of behaviors is enumerable: a look-up-table can specify arbitrary n-to-m combinatorial logic functions. However, instead of operating on huge somewhat meaningless numbers for large behaviors, we define all behaviors in terms of a graph of simpler behaviors. In this section we will define a rule system to perform topological transformations that preserve behavior. Our behaviorally invariant macros will execute atomically while the RhoZeta interpreter is running allowing for dynamic optimization. Reduction rules will be used to optimize a serially specified 64-bit leading zero counter.

3.1.1 Catalyst: Guarded Atomic-Actions for Graph Rewriting

In Chapter 2 when we presented the event-based FIFO thread dispatcher, we discussed predicated step signals. Without predicated events, we could not conditionally dispatch step

functions under certain conditions of the step signal[1]. A different asynchronous event scheduling abstraction is guarded atomic-actions. Instead of using guarded atomic-actions to modify cell values we will perform predicated optimization on a sheet's formula. The result is a rule-based framework for dynamically executing atomic actions on spreadsheet. While guarded atomic actions operating on cell values could be used as a framework to implement compute-commit ExecutionPolicies for RhoZeta, the metaprogramming approach for dynamic compilation is easier to implement for immediate integration. Instead we use guarded atomic actions to act on cell formulas and construct some simple optimizations.

The Catalyst algorithm is my implementation of atomic graph rewriting. It incorporates a pattern matching framework and a means for concurrent graph transformations as atomic actions. Multiple graph-walking threads can be used to identify or act upon property patterns in the spreadsheet process graph (properties are represented in the cell color, border, font size, or any other arbitrarily defined associations). A variety of search algorithms can be used to dictate the graph traversal order: a depth-first traversing thread can be written in 10 lines of Python. A thread may spawn new and different threads or it may allocate new cells to add formulas to the graph. Every catalyst thread may act atomically on the spreadsheet and multiple catalysts may simultaneous operate in the same cells if they do not modify the same properties. Atomicity is managed with locking and a thread-priority conflict resolution system. By allocating empty cells with specific formulas, we can form connections between distant cells of a graph.

The optimization rules I have developed are generic fine-grained Boolean optimizations though they have been specifically designed to optimize a serial chain of multiplexors into a more efficient combinatorial network to produce an optimal Leading Zero Counter. The premise

[1] This is not really an issue with lambda since we can predicate the expansion of a functional lambda.

for this optimization is based on analogy of guarded atomic actions as a pool of chemicals and enzymes (global memory and guarded rules) that may modify other chemicals. Optimizations may be performed by the teamwork of multiple types of simpler Catalyst threads. For example, an indicator for a graph optimization walks a cell precedent graph and attaches a color property to cells that match a predicate condition, for example, we can mark all IF statements green and all constant 1's and 0's blue and red. When a transformation thread finds cells with the correct promoter color pattern it performs an atomic action to copy and optimizes a sheet. Another thread can evaluate the output of a cost function and determine over time when a transformation is worse than the original. This system is biologically inspired with the intent of evolving rules; though I have only incorporated rudimentary cost-function indicators for feedback.

The learning curve is extremely low for developing spreadsheet graph traversing functions with win32com, VBA or UNO using cell properties to mark graph properties. A good way to develop these optimization threads is by using the macro recording feature in a spreadsheet as you perform the optimization and then modify the code to generically walk the dependency graph. By associating cell styles with cell predicate matches, it is possible to create macros and watch them run. Figure 3-1 shows a number of graph reduction optimizations that propagate static values and merge associative and commutative operations. These optimizations are almost always beneficial reductions. It is usually necessary to predicate atomic actions on whether or not they lower a cost function.

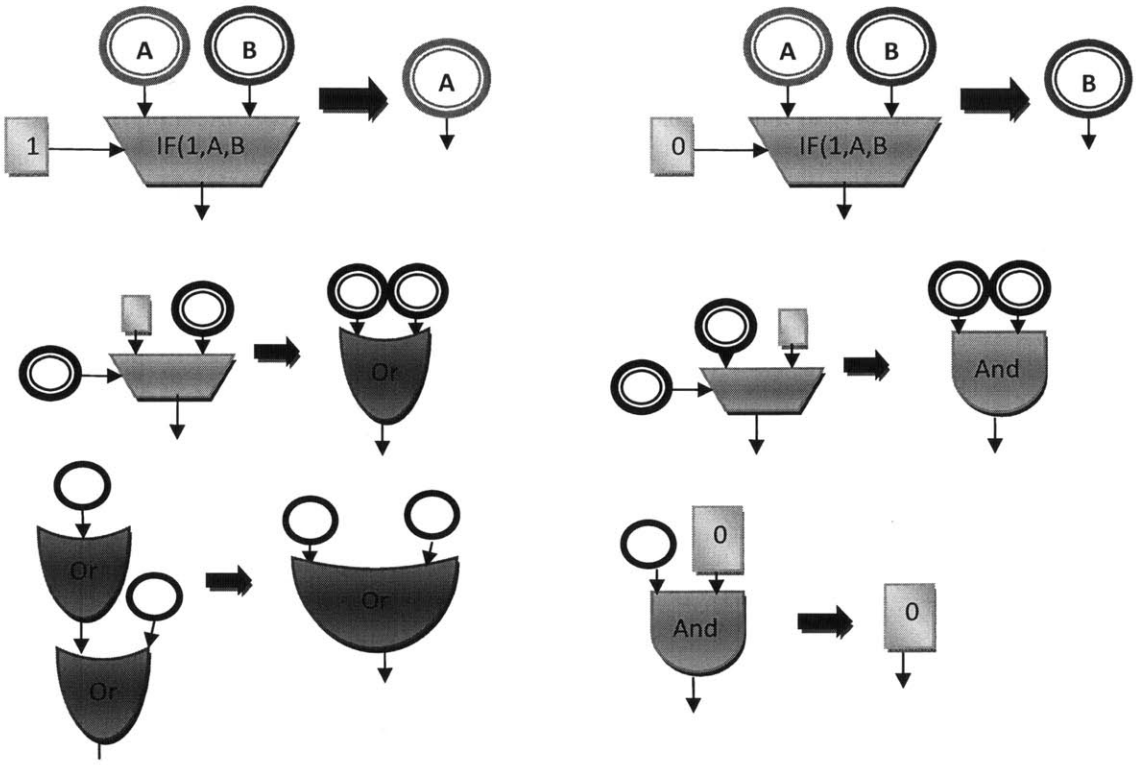


Figure 3-1: A few simple reduction rules for multiplexors and Boolean logic. These are programmed as threads that search for patterns in the precedent graph and atomically modify the graph.

We can use macros to generate cost functions out of a process graph. These allow us to track some metrics as our sheet calculates and provide feedback to a cost-function minimization thread. For example a useful macro for analyzing the average switching rate of signals is shown in Table 3-1. The switching score macro creates new cells to store the value of a cell from the previous iteration and tracks each transition. Temperature simulations can be performed by putting this activity data into a resistive lattice temperature model as shown in Figure 3-2

Table 3-1: A macro spawns cells to track the switching activity of every cell in two adder benchmarks. The top benchmark sums serial connected counters so the activity of each successive bit is half of the previous bit. This power macro is useful for creating objective functions to direct an optimization.

Nested Counters								
Word1	0.9991	0.499325	0.249437	0.124719	0.062134	0.031067	0.015308	0.007654
Word2	0.003602	0.001801	0.0009	0.00045	0	0	0	0
Carry	0.461054	0.34579	0.201711	0.110311	0.05493	0.027015	0.013507	0.006303
Sum	0.995498	0.501126	0.250338	0.125169	0.062584	0.031067	0.015308	0.007654
Random Inputs							**Average**	0.162652
Word1	0.507429	0.501576	0.505628	0.502927	0.470959	0.497974	0.494822	0.496623
Word2	0.489419	0.484016	0.521837	0.511031	0.496173	0.499325	0.505178	0.522738
Carry	0.377758	0.464205	0.489419	0.493021	0.483566	0.495272	0.500675	0.512832
Sum	0.496173	0.506979	0.492121	0.501126	0.488519	0.497524	0.518235	0.497073
							Average	0.494442

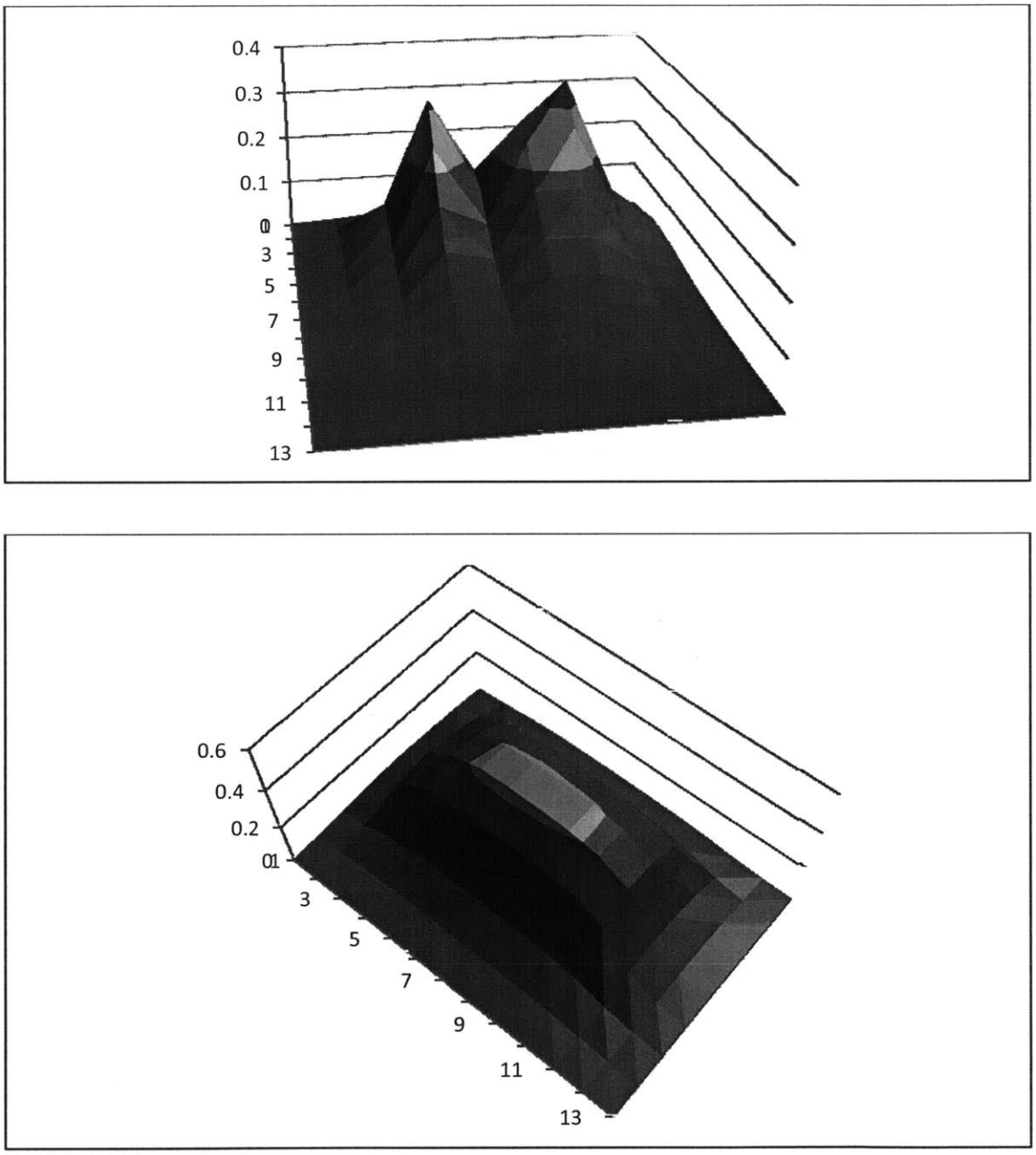


Figure 3-2: Relative temperature simulations of two adder benchmarks (with made up thermal equation constants). This simulation is performed directly in the spreadsheet using a function of the computed activity values. The edges are fixed at a constant temperature 0. Thermal capacitance and resistance between nodes is the same for each tile. This simulation is based only on the activity in a cell and does not take into account specific wire connections. The LSB for the counter (top) is easily identifiable as a "hot spot."

The goal for the LZC optimization macros was to see if using a set of combinatorial circuit transformation rules for Boolean logic we could transform a linear combinatorial circuit into a logarithmic one. As a starting point the circuit consisted of 64 serial MUXes for each of 6 output bits. A "combinatorial path" indicator measures the worst-case combinatorial graph distance from an input to a node of a circuit[2] and a "total area" indicator measures the total number of cells used for a circuit. The speculation and resource sharing macros presented in the next two sections were useful to explore the automatic parallelization of a serially-defined Verilog architecture and achieved positive results for the serially defined leading-zero-counter circuit. The method runs in many massively parallel threads each atomically modifying color or formula properties in the spreadsheet graph. The threads can visualize locking, allocating, and atomically modifying cells by coloring them and changing fonts[3].

3.1.2 Speculation as a Parallelization Primitive

Speculation in the context of an instruction set executer is the process of conditionally executing code simultaneously in parallel before the conditional predicate expression is resolved. The performance boost comes from the fact that we can start to compute both possibilities while we are determining the predicate. Speculation presents a direct method for improving performance with parallelism. For example, consider the expression IF(P,A,B). Suppose that A and B require 5 ns to compute while P requires 4 ns. In a sequentially executing processor, resolving this conditional expression requires 9 ns. If we have a 3 parallel processors, we may simultaneously schedule P, A and B to execute simultaneously. When P resolves we may halt

[2]The worst-case combinatorial graph distance is not the worst-case delay since false-paths may exist. Detecting false paths is hard.

[3] Making a spreadsheet visualize multithreaded locking and graph-walking should be the first homework assignment for a parallel compilers class and takes few lines of Python code to achieve multithreaded gratification that walks the contents of cells and colors them and so forth.

the unnecessary operation and resolve the entire conditional expression in 5 ns. If we only have 2 parallel threads, then we can only speculate one of A or B and we would want to pick the more likely of the two outcomes. This is what speculating branch predictors do.

In dataflow executers like FPGAs, the Verilog compiler will automatically make MUXes out of all conditional statements (except in the case of a conditional lambda application, which should be thought of as a conditional dispatching a trap). Even when we are already speculatively executing all inputs to a multiplexer, there is another form of dataflow speculation which is useful for transforming decision bound structures into speculative parallel structures. Consider the formula F(X,(IF(P,A,B))). Suppose now that P resolves in 4 ns but A and B resolve in 1 ns. Suppose further that F(X,A) and F(X,B) require 3 ns. We may transform F(X,(IF(P,A,B)) into IF(P,F(X,A),F(X,B)). The former requires 7 ns if executed with full speculation but the latter only requires 4 ns. This transformation rule is shown in Figure 3-3. Note that F may have any number of other inputs and this transformation is still valid.

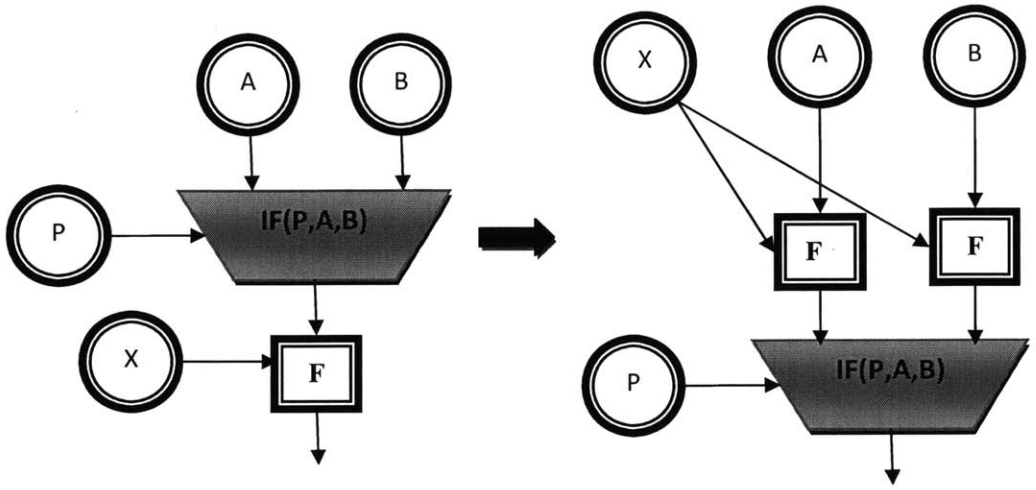


Figure 3-3 We may speculatively apply F to A and B before determining the predicate P. If A or B resolve before P, then this can speed up the circuit. If F(X,A) and F(X,B) share common substructure then we can use a redundant resource sharing macro to minimize wasted resources.

Speculation does not improve our serial leading zero counter circuit, and could potentially cause a cancer: an infinite number of multiplexers may be created. In order to prevent this from happening we have had to alter the indicator for speculation so that it only occurs when the predicate has the largest combinatorial delay of all of the MUX inputs. Figure 3-4 shows a circuit diagram of the serial LZC as it was defined in Chapter 2 and demonstrates what happens on one step of speculation. This example provided an important lesson on the need for feedback and regulation in automated graph rewriting systems.

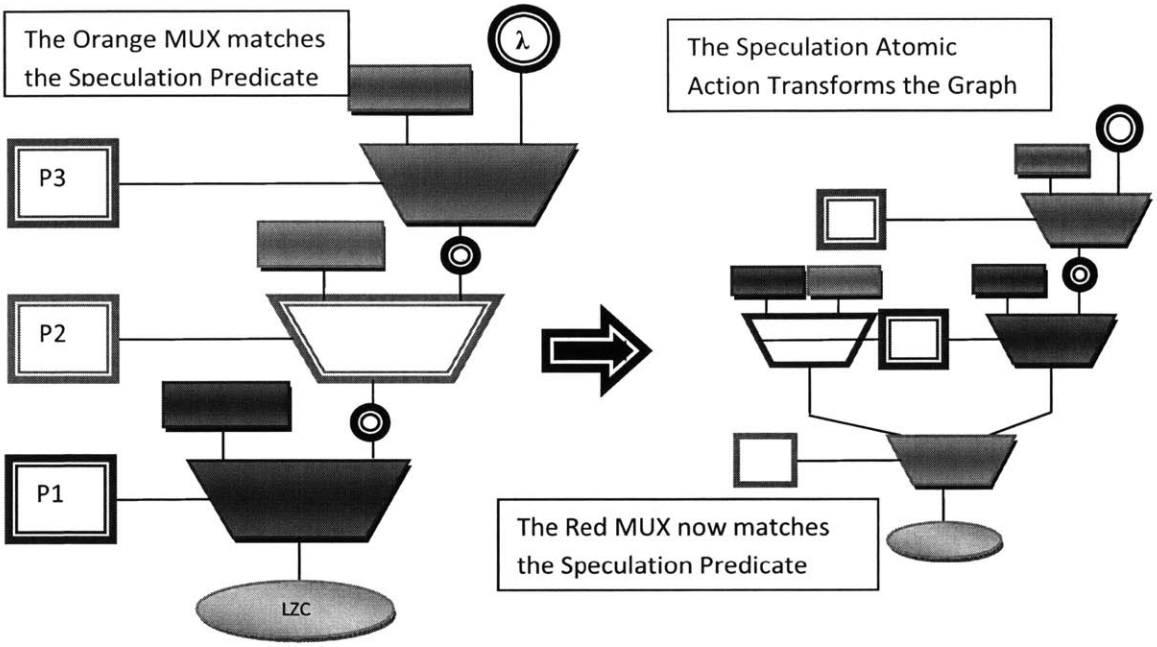


Figure 3-4: Out-of-control speculation does not decrease the combinatorial path of a serial LZC and may create an infinite number of MUX circuits if not properly guarded.

3.1.3 Redundant Resource Sharing

After the combinatorial reductions propagate constants into the LZC circuit an independent circuit for each bit remains. Whenever a cell has two dependents computing the same formula, those two formulas can be merged. This is often necessary to eliminate common substructures emerging from other transforms. When reduction rules on the leading zero counter occur in full, a number of redundant AND and OR gates emerge computing on various subsets of the input vector. Without Xilinx resource sharing, static propagation alone yields a highly redundant circuit requiring a lot of wire delay. Resource sharing optimizations identify when redundant operations exist and must split redundant AND and OR logic into multiple disjoint subsets as shown in Figure 3-5.

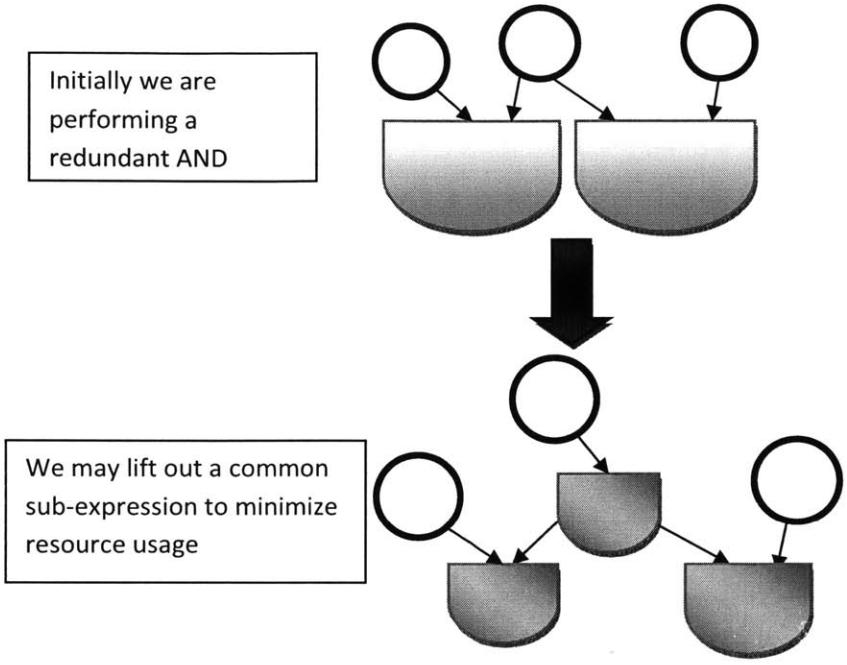


Figure 3-5: A redundant resource sharing macro minimizes the size and simplifies the routing of redundant expressions.

Results for compiled Verilog emitted before and after reduction rules and redundant resource sharing are shown in Table 3-2. Catalyst's fully optimized LZC outperforms a fully reduced implementation without resource sharing when both are compiled with Xilinx's resource sharing optimizations. Multithreaded graph rewriting as atomic actions are is a powerful mechanism for dynamic architecture optimization. There is potential for reflection in a system like this to self-optimize the rule system execution reflexively. The method of using guarded atomic rewrite rules on a spreadsheet is underexplored and has potential to vastly simplify creating and understanding heuristic optimization systems for a variety of hard problems. These algorithms run slowly when they must communicate with the spreadsheet visualization directly. It could be possible to use the RhoZeta frontend synchronizer framework to decouple the algorithm from the visualization of properties.

Table 3-2: Results for 64-bit leading-zero-counter under speculation and resource sharing optimizations as produced by the Xilinx Verilog synthesis tool. Note the huge amount of routing resources consumed under full reduction without resource sharing. The serial LZC fully optimized by Catalyst before Xilinx synthesis and optimization outperforms all other optimizations by 1 ns (20.5% faster, the macros used were specifically designed to do LZC really well of course).

	Slices	Levels of Logic	Total Delay (ns)	Logic Delay	Routing Delay
Serial LZC with Xilinx Full	**74**	**13**	**7.016**	**1.576**	**5.440**
Reduced LZC without Xilinx Optimization	**45**	**15**	**9.483**	**1.964**	**7.519**
Reduced LZC with Xilinx Full	**45**	**8**	**4.873**	**1.404**	**3.469**
Reduced LZC with Catalyst Sharing	**45**	**6**	**3.883**	**1.130**	**2.753**

3.2 Optimizing Legacy Emulators

Emulating an instruction set in a tiled array is a strategy for legacy system compatibility especially where highly domain specific software systems exist such as industrial machinery[4] or mechanical controllers. As FPGAs and RAW chips enter the mainstream computing market, a model for legacy software compatibility will be crucial to their adoption. Instruction set emulation provides a route to such compatibility. Nearly all FPGA manufacturers offer a soft-microprocessor core with a C compiler suite in order to provide a traditional development environment for microcontroller applications. Many FPGAs also include fixed hardware microprocessors such as a PowerPC or an ARM. The possibility of a reconfigurable computing system executing arbitrary legacy emulators is the goal of RhoZeta and inspired the name.

There are number of ways to consider optimizing generic emulator structures. Suppose we have built a model of a general purpose CPU core, like the RISC core of section 1.2.3. If we

[4]Often legacy hardware is better understood and easier to emulate than upgrading domain specific software to a new platform.

dispatch our core to execute a static thread that requires no dynamic allocation, and no multiplier, then our memory space is too big and we've wasted space for our multiplier. We can optimize out these structures from the core. Given an ISA emulator with a static instruction ROM that does not perform dynamic code modification, it is possible to perform a static code analysis to identify a complete set of instructions and registers used by the function. Once a generic multicore architecture is built, generation macros can parameterize all of the system components and a manifest of required cores and functional units can be determined by static instruction analysis.

After creating a multicore architecture for a sample-based-synthesizer [22], I employed this technique to reduce the area for the DSP components to 30% of the original area. By pruning cores to a specific application, we can fit more of them in an FPGA which is useful for algorithms that can scale this way. Using generic multicore frameworks and performing architecture optimization for a specific core task is a generalized way to maintain a compatibility layer between FPGA and Multicore architectures. For example, an auto-pruning SPE-compatible FPGA core could be used as an accelerator to the Cell processor with exactly the same API.

Another option for legacy compatibility is instruction stream emulator pipelining. Suppose now that we implement an ISA emulator as a function of a set of instructions and some initial state of every cell. The CPU lambda will execute a single instruction each cycle and combinatorially produce the next state of the processor. Thus, we can place "=CPU(InstructionSet,InputSystemState)" into a reading order blocking pipeline of cells to capture the ILP behavior of multiple CPU steps or use non-blocking interpretation to make a multicycle pipeline. This works even in Excel and can be tested by copying the RISC

architecture sheet from Section 1.2.3 and making each sheet read registers from the previous sheet's state in the pipeline. Figure 3-6 shows a diagram of this process.

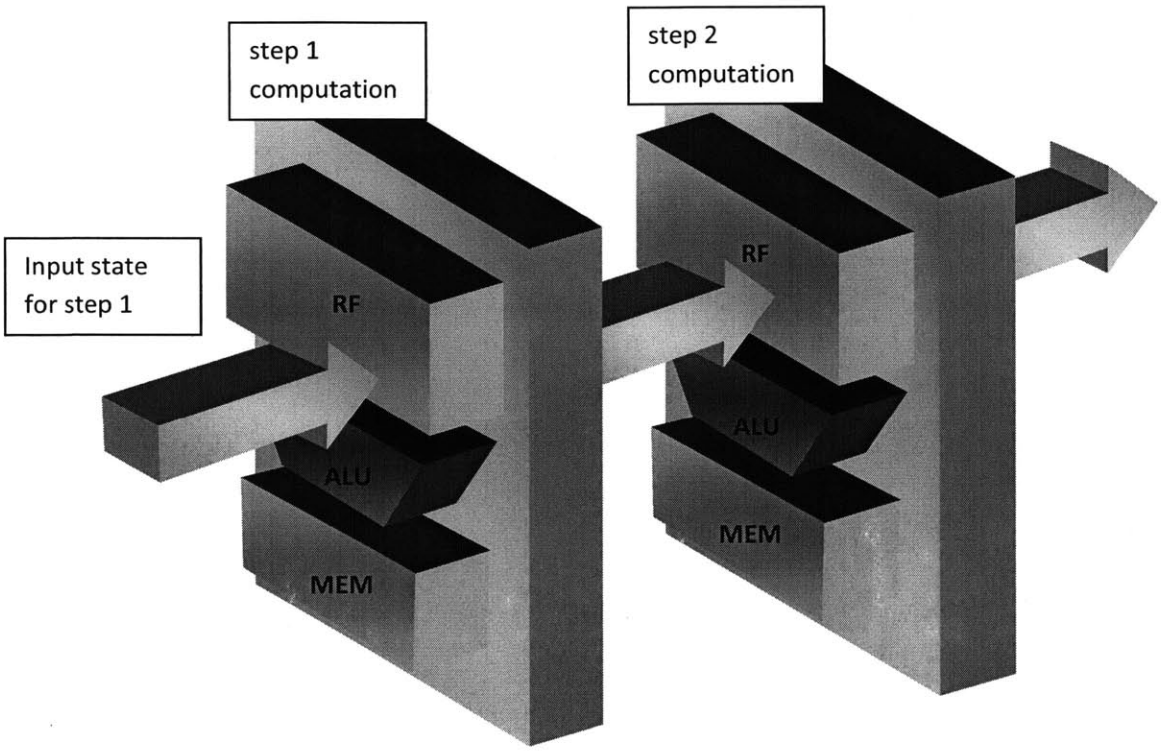


Figure 3-6: Each RISC core is used to compute the next input state of the next stage in a pipeline. With functional unit pruning macros, this method could be used to auto-pipeline arbitrary instruction set emulator code

Instead of using a single ISA emulator with its next state attached to its current state, we can spawn a new ISA emulator as a continuation for each instruction and pass the entire emulator state to the newly spawned architecture. Whenever we reach a conditional branch, we can either speculate, backtrace to a previous stage or dynamically spawn a new emulator for the continuation of a thread. These options are shown in Figure 3-7.

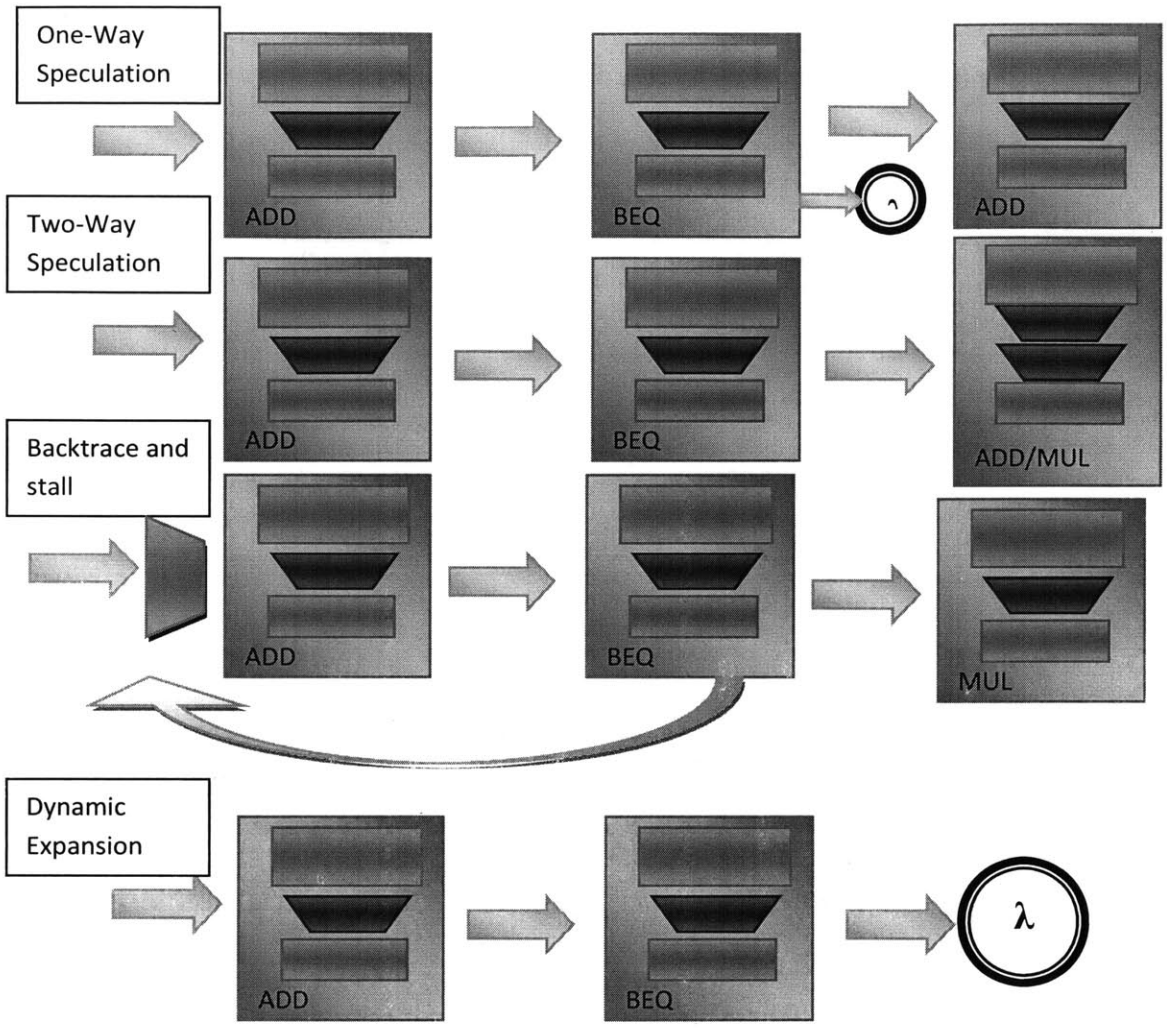


Figure 3-7: Call-with-current-continuation meets reconfigurable computing. A number of different strategies can be used to handle branching in the unrolled-emulator computer. We may speculate with one or two supported next state possibilities. We may backtrace to share a previously spawned stage and possibly stall its input pipeline. Alternatively we may wait until the branch resolves and invoke a lambda trap.

When architecture pruning rules are combined with dynamic pipelining we can imagine a system in which each function is laid out in front of the data as it reaches the execution point. Since this is reduced to a spreadsheet macro, it can be generally applied to arbitrary ISA emulators. A complete implementation of such a system provides a framework for legacy

compatibility. A global memory model for this kind of dataflow architecture is extremely non-trivial and any interrupting behavior must be handled by a lambda trap.

3.3 Pipeline Resource Sharing

Computers can be thought of as a finite array of statically typed resources in a graph with a strict interconnection schema. Resource consumption in the idealized spreadsheet application is theoretically infinite with no routing constraints. Sometimes fast physical resources or large macro structures can be placed in a gate array and must be shared by multiple processes as shown in Figure 3-8. Our spreadsheet models can often exceed the physical resources available for execution of all cells. Parallel threads can share a pipelined resource to minimize the amount of physical area required to perform a behavior.

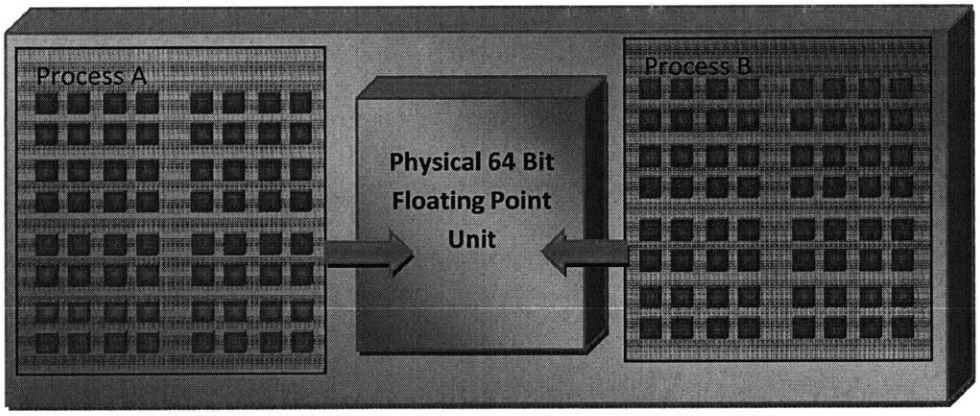


Figure 3-8: Two processes in a reconfigurable fabric share a large 64 bit floating point unit. The Floating Point unit may be a physical structure or a macro. Managing resource sharing in a reconfigurable array is a difficult problem.

The decision to share often comes at a tradeoff. Maintaining behavioral invariance often requires faster pipeline units which require more power and may require a mechanism to stall a pipeline. This overhead means that two processes sharing do not necessarily halve the area.

Whenever resource sharing is used, some extra hardware is spawned to manage the sharing relationship. For example, Table 3-3 shows two square root operations before and after resource sharing transformation. A round-robin resource sharing macro chooses one of two inputs to square-root each step. Figure 3-9 shows how this transformation saves area. As an alternative to round-robin selection, a priority dispatcher may use the leading-zero-counter from Section 3.1 to select one of many requestors with priority.

Table 3-3: The square-root resource can be shared in a round robin setup. Columns A and B are two separate cores with two separate square-rooters. Columns C and D share the operator

	A	**B**	**C**	**D**
1	=Input(A)	=Input(B)	=IF(C1=0,1,0)	
2	=Sqrt(A1)	=Sqrt(B1)	=IF(A3,Input(A) ,Input(B))	=Sqrt(A5)

Regulating resource sharing macros requires a complex optimization system to evaluate various sharing possibilities. Resource sharing macros must take into account temporal and spatial locality of physical processes. For example, in system designed for dynamic reconfiguration to run unrolled emulator structures, we will want to schedule processes that recycle common resources when threads terminate to minimize the amount of reconfiguration. There are also tradeoffs related to heterogeneous partitioning of processes. These topics will be explored further in Chapter four.

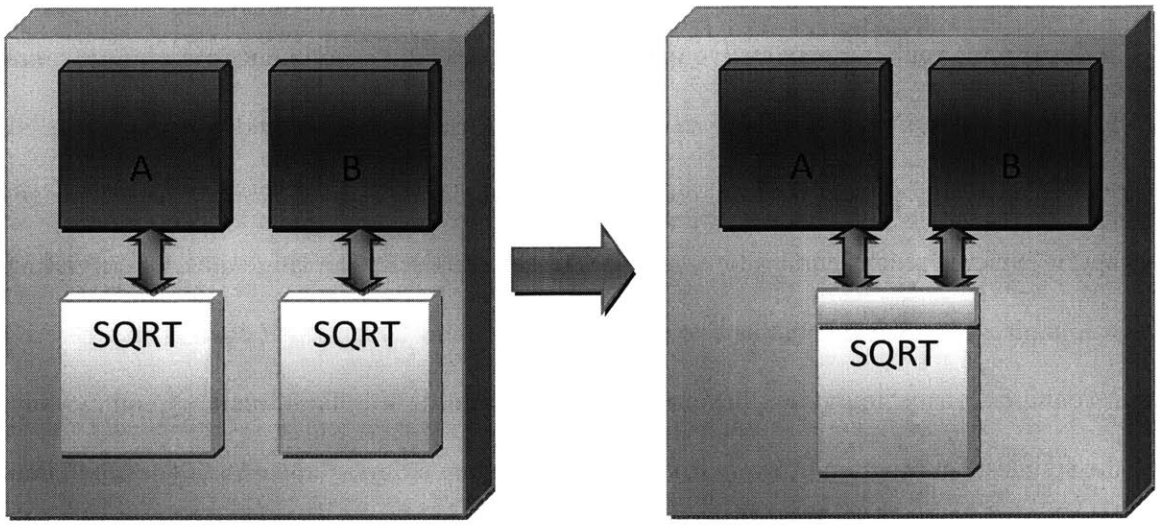


Figure 3-9: Resource sharing reduces the overall area required for complex pipeline elements.

3.4 Fault and Defect Tolerance

Incorporating fault and defect tolerance into an operating system for reconfigurable chips has potential to decrease time-to-market for new fabrication technologies and to increase yield of semiconductor devices. Randomized actors can simulate a fault or defect. A simple macro for fault and defect tolerance performs n-way modular redundancy and a guarded atomic action system can detect and manage spreadsheet formula defects by reassigning formulas from a guaranteed source.

As technology progresses to smaller geometries, static and dynamic faults will occur with increased frequency. There are well established mechanisms for analyzing of error correction codes and susceptibility to faults. There is a proof that the parity function requires at least logarithmic redundancy [26]. FPGA vendor tools incorporate fault tolerance macros for configuration memory [27] and DRAM and hard disk systems commonly incorporate error-

correction codes. There is also potential to use dynamic reconfigurability to recover from a fault [28]. From a programmer's perspective there are relatively few development environments that address the issue. In most languages, there is no a simple mechanism to randomly flip the variables in a running system without multiple compile and test iterations. Since there is no concept of dynamic reconfigurability in previous hardware description languages, incorporating dynamic fault recovery mechanisms into an FPGA system must use vendor macros and may require complete reconfiguration. A generalized strategy for testing fault tolerance can extend into the spreadsheet programming environment with random atomic actors. A self-testing circuit can manage even with static defects within the array.

3.4.1 Randomly Guarded Atomic Actors

Most languages lack a simple mechanism for testing fault tolerance. Using a thread to randomly move around the graph and modify cells, we can inject faults into the system to simulate dynamic defects or randomly place an actor in a cell to simulate an unknown static defect. Faults may modify any cell property like formulas, values or colors in the UI. The atomic action optimization system could have strange behavior due to coloring faults. It is possible to create evolution where a random actor modifies the guard of an optimizer actor and mistakenly permutes the dataflow graph in a non- invariant way. Since atomic actors will ultimately be implemented in a fault-tolerant dynamic dataflow environment it is possible to ignore these effects.

Formulas and values represent two different components of a reconfigurable computing system. The formulas are configuration data in the cells and the values are the processed data flowing through the system. Both components of a reconfigurable system are susceptible to

defects though we do not necessarily have the same mechanisms to access both. N-way modular redundancy (NMR) with majority voting can detect and correct both kinds of faults at a large area overhead cost. Table 4-1 shows how a majority-vote NMR is implemented in a spreadsheet. Figure 4-1 shows NMR detecting a data fault and a configuration fault. For high-reliability systems it is necessary to incorporate this sort of fault detection at multiple levels of hierarchy.

Table 3-4: An NMR macro copies the IIRFilter 3 times and instantiates a majorityvoter. NMR can be used to detect faults in the configuration or in the data flowing through the system.

	A	B	C
1	=Input(A1)	=IIRFilter(A1)	
2		=IIRFilter(A1)	
3		=IIRFilter(A1)	=MajorityVote(B1:B3,errorhandle())

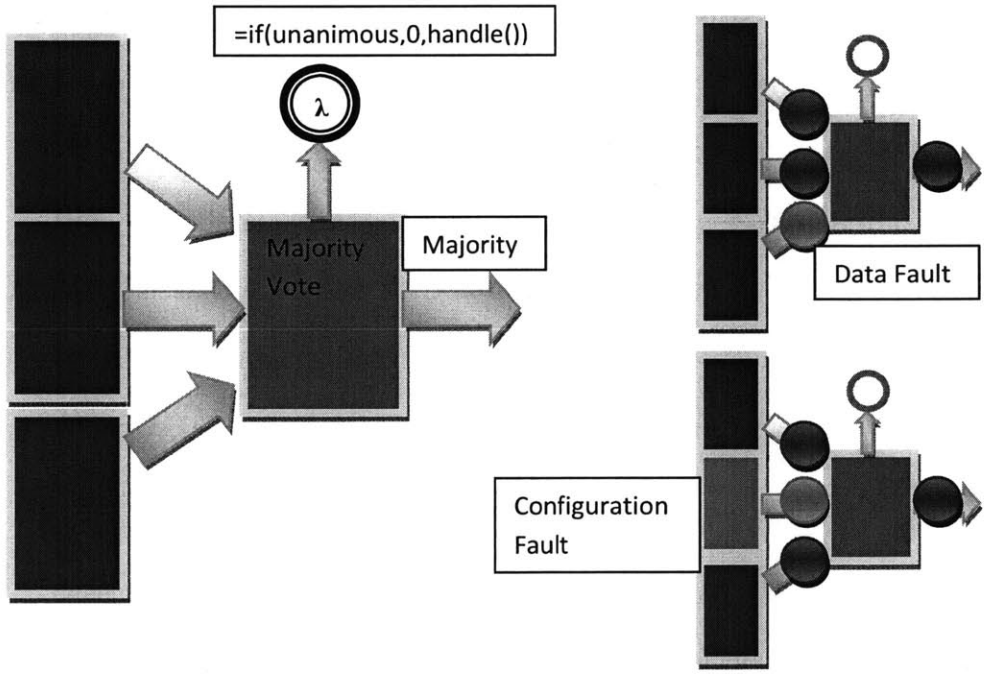


Figure 3-100: NMR with a majority voter can detect and handle configuration faults or data faults. The handle() function executes an atomic action. NMR alone does not guarantee against silent faults in the configuration.

To handle a fault, we must detect if it is a configuration fault, a data fault or a static defect. Configuration faults can be behaviorally invariant; it is possible for a random event to modify the don't-care conditions of a look-up-table. NMR is not sufficient for detecting such silent configuration faults and there is a possibility that a series of silent errors can accumulate across a system's configuration before a fault is observed. Modern FPGAs provide configuration check-sum mechanisms that can detect configuration faults even with don't-cares. The optimal method for handling a detected error is to identify and record the source of the error so that we can track for static defects and partially reconfigure the erroneous region.

3.4.2 Error Correction as a Data Type

Error correction codes (ECC) are an optimal method to increase the reliability of stored data. It is possible to incorporate error correction macros in an FPGA that can read back its own configuration and check it against an error correction code, though this is wasteful if we can just reconfigure the location again. Error correction codes can be used generically throughout an electronic system to correct data stored in RAM or any other long term data storage though ECC codes do always not make sense for the data moving through the dataflow architecture. Harmful effects may still result when a fault occurs in the dataflow architecture and so ECC must be coupled with functional redundancy to actively detect and correct dataflow faults.

In order to have a unified perspective of fault tolerance, we would like to program in terms of guaranteed execution and have NMR, ECC and other fault-tolerance mechanisms incorporated in the OS. A fault tolerance guarantee should be like a dial which can allow the developer to turn up the number of zeros guaranteed by the system in a provable manner. The gap between our ability to model faulty systems and our ability to apply fault tolerant mechanism

means that advanced techniques for fault tolerance are generally unexplored. If fault-tolerance is reduced to a data-type in the language and a property of the OS, then structures like fault-tolerant arithmetic and logical units [29] can be incorporated as sub-systems to provide a fault-tolerance guarantee without requiring substantial developer burden.

3.4.3 Static Defect Detection and Self-Testing

So far we have addressed the problem of detecting and managing dynamic data and configuration faults via NMR and ECC by tracking and reconfiguration. Since an NMR fault tolerance system can track the location of faults when it invokes a handler, we can determine if a particular physical location has regularly occurring faults. If a chip is suspected to have a static physical defect in a particular configuration bit, it can be found using multiple orthogonal self-testing configurations to narrow down the potential locations of a physical defect. Once a defect is found and characterized, a software system can manage it.

It is also possible to characterize a reconfigurable chip for process variations by placing self-timed ring-oscillators around the circuit and comparing their counts with counters fixed to a clock. A thorough investigation of this method is provided in [30]. A diagram of this auto-characterization is shown in Figure 4-2. Using a low-level FPGA API it is possible to test various interconnect paths around a circuit to determine variations in the interconnect delay between various points. If self-test characterization of hardware is built into an operating system, it is possible reduce the hardware test costs and simply manage defective components or optimization specifically design for the characterization of a part.

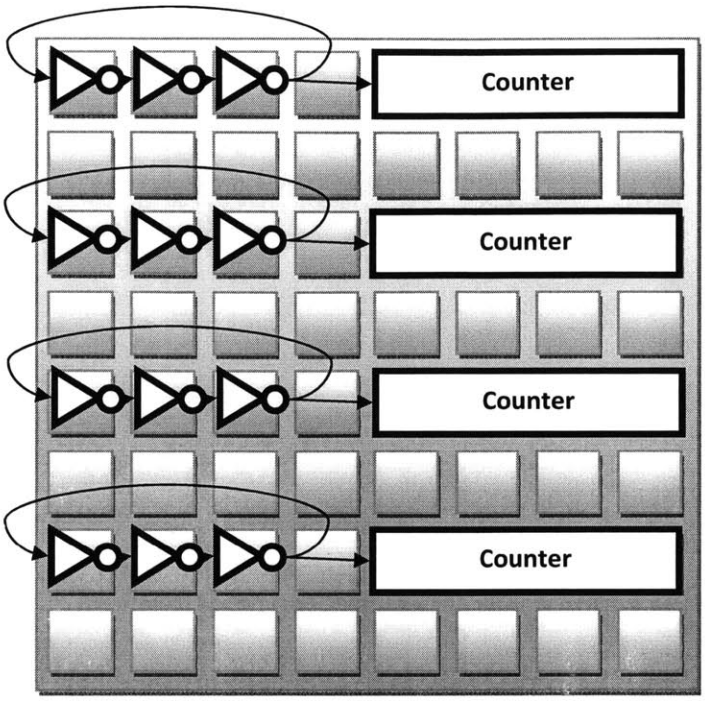


Figure 3-11: Ring Oscillator structures can be used to auto-characterize an FPGA for interconnect variance

In this chapter we introduced the Catalyst system which uses guarded atomic actions in a spreadsheet graph rewriting system and explored potential macro strategies for optimizing spreadsheet dataflow circuits. Since Catalyst graph-rewriting macros modify cell formulas atomically, the optimizations occur while the system is running without affecting the system's dataflow behavior. We examined how cost metrics allow us to regulate the optimization system and used combinatorial reduction and resource sharing to optimize a leading-zero-counter for Verilog compilation. We explored the potential to use macros to automatically optimize arbitrary emulators for dataflow execution and discussed the potential for resource sharing macros to reduce area. We also showed a set of macros which allow us to randomly simulate faults on a spreadsheet and modify sheets to handle such faults.

Chapter 4

Efficiency, Heterogeneity and Locality Optimization

Wherever you have an efficient government you have a dictatorship.
-- Harry S. Truman

This chapter discusses the general problem of profiling and optimizing applications in heterogeneous environments consisting of multiple types of processing elements. Section 4.1 discusses information entropy throughput per dollar as a metric of computational throughput efficiency in an application specific environment. Section 4.2 explores how heterogeneous management can potentially minimize the cost of computational throughput by locating computational pipelines in particular hardware. Section 4.3 suggests a multi-dimensional simulated annealing approach to generalized locality optimization and explains the use of projection to gradually introduce dimensionality constraints into a placement optimization.

We have shown that spreadsheet iteration is a sufficient model for expressing arbitrary computation. In as much as there is demand for computation there is demand for spreadsheet iteration. The rate and cost of spreadsheet iteration can provide a measurable efficiency unit for utility computing. If there is a hardware agnostic API for standard spreadsheet iterations, a general cost reduction strategy may be to partition parts of spreadsheet iteration to the lowest cost supplier of a particular function. A unified cost model is required to profile spreadsheet iteration across a plurality of options. Cost functions may depend highly on application specific constraints. When high-throughput is paramount, power efficiency may be less of a concern. For

applications that can achieve lower power consumption through increased parallelism, the amortized cost of real-estate must be weighed against power benefits.

4.1 GOPs per Dollar

We have already considered switching activity, temperature and combinatorial delay as indicators of module's cost. In this section we will generalize computational throughput efficiency and examine GOPs/$ as a metric for computational efficiency. GOPs is short for "billions of operations per second" and is dependent on the application. If our unit of an application specific computation is a spreadsheet, then GOPs is a measure of one billion iterations of that spreadsheet in a second. Teraflops, for contrast, is a measure of floating-point throughput. The cost function for a specific application is also dependent both on the type of operation being performed and the mapping to a particular computational fabric. An application's cost function may incorporate multiple factors including power, latency, area and temperature.

To quantify information processing I will use Shannon's concept of information entropy. If a Boolean random variable is has equal probability of being 1 or 0, then we gain 1 bit of information from reading its value. Logic functions affect the entropy between the input and output signals by projecting a domain into a smaller range. For example, a two input NOR gate with evenly distributed independent inputs has output probabilities has p(0) = .75 and p(1) = .25. Storing the string of outputs of the NOR gate provides us with .75*lg(.75) + .25*lg(.25) = .811 bits of information (lg means base-2 logarithm). If this gate iterates 1 Billion times, we expect a lower bound of 811 Megabits required to store the output string.

Consider now if both of the inputs to the NOR gate are independent and have p(1) = .99 and p(0) = .01. The output of our NOR gate now has p(1) = .0001 and p(0) = .9999. Resolving the NOR gate only provides us with .0015 bits of information, substantially less than before. Storing the outputs of 1 Billion iterations should only require 1.5 Kilobits of information. Yet the circuitry providing the data from the NOR gate output to its dependents still must transmit 1 Gigabit of data. Thus there is a huge amount of symbolic inefficiency from the quantization required for discrete states. Whenever symbolic inefficiency occurs, it is likely that interconnect efficiency can be improved through serialization and multi-word encodings.

Information entropy of an input provides a physical basis for measuring an operation's usefulness. If GOPs is operational throughput, then its physical unit is bits/second or (information entropy / time) and GOPS/$ = information entropy / (time * cost) is a measure of throughput efficiency subject to an application's cost-function. If the information entropy of a variable is small, then it may not be worth the cost of implementing the hardware that reads its value and instead invoking a lambda trap to the OS.

For example, consider an adder whose inputs have high probability of being small. The entropy of the output bits is very low for the high order bits. Depending on the costs and probabilities of handling exceptions, it may be worthwhile to use an 8 bit adder instead of a 32 bit adder. If there is some finite probability of inputs being larger than 8 bits then we will need a lambda trap as shown in Figure 4-1. This adds a fixed area cost to the circuitry and a probabilistic time-cost dependent on the probability and delay of invoking a trap. If we compare cost functions across a variety of topologies we can deduce an optimal bit width for our adder.

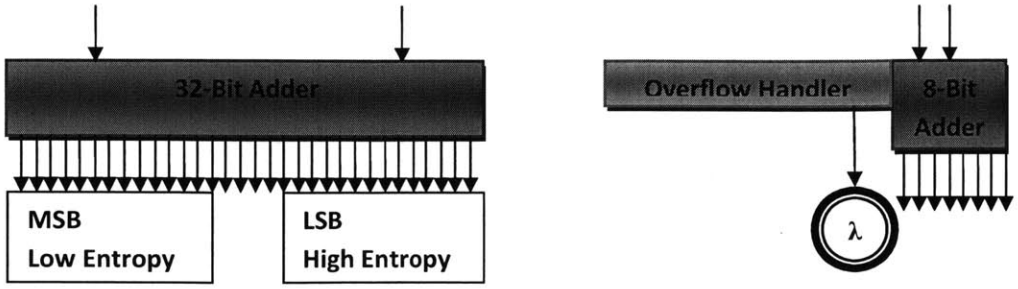


Figure 4-1: Since our 32-Bit Adder has low entropy in the MSBs we can remove the computation for the higher-order bits. By using an 8-bit adder and an overflow handler we can save area. Since we require detection circuitry and a trap in overflow cases, there is overhead to this method.

In cases where the entropy of a signal between successive pipeline stages is small, it may be possible to perform compression on the data between stages to minimize the data transmission overhead between streaming cores. For example, consider the pipelines in Figure 4-2: a transformation and filter pipeline produce low entropy data for an analysis engine to interpret and the analysis engine is separated from the transformational filters by a substantial latency. A common example of such a setup may be an image recognition algorithm which first performs a series of transformations and filters on an image processor and ships the reduced data to a database engine for comparative analysis against a dataset. The output of the image filter may be represented by a bitmap with substantially lower information content. An encoder and decoder can substantially compress the symbol space before transmission to decrease the burden on long distance interconnects. In many cases it is possible to propagate the encoder and decoder into the pipeline and operate directly on compressed data streams. For example, if a bit-serialized transmission feeds into a parallel adder, we can use a bit-serial adder instead.

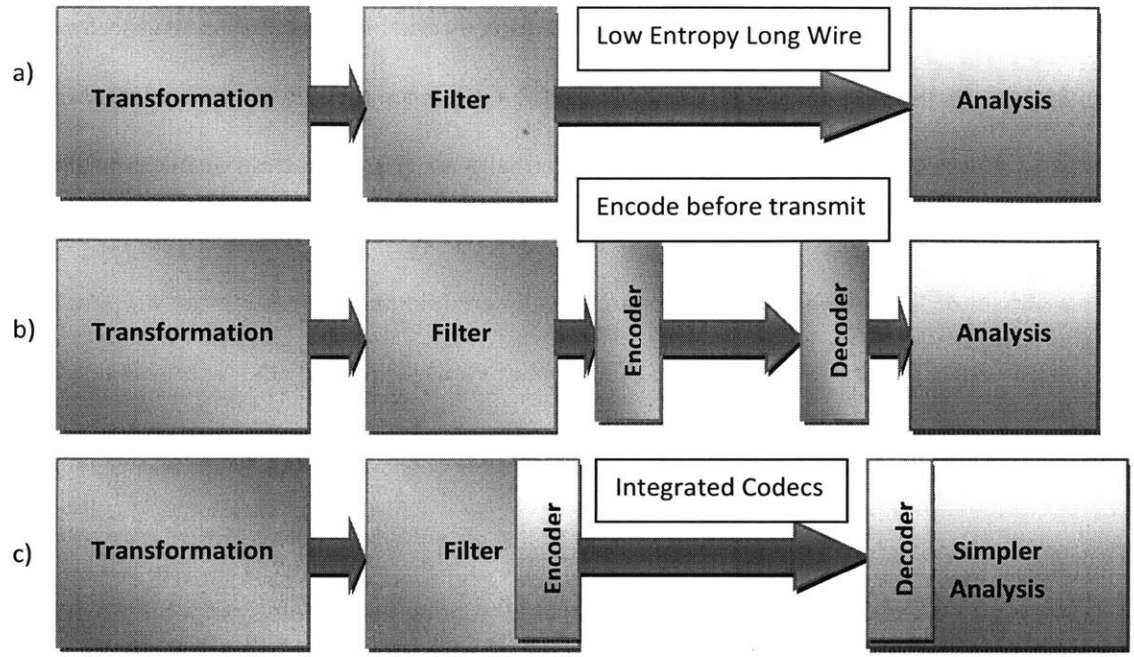


Figure 4-2 Initially the signal transmitted over a long wire has low entropy. If we encode the stream before we transmit the data to the analysis unit we save transmission overhead. It is often possible to integrate the encoder and decoder into the filter and analysis to simplify these modules.

4.2 Heterogeneous Load Balancing

In our pipeline example, we made a decision to encode data prior to long-range transmission between separate processors. Partitioning and profiling an application across multiple co-processors still remains as a task left to developers. Partitioning decisions are non-trivial: we will pose an example of partitioning N FFT operations of size S and we have the option of using a CPU, a GPU or an FPGA. Let's suppose our computation begins in the CPU and we only wish to perform 1 FFT of size 128. In this case it may not be worth the overhead of offloading the process to the GPU or the FPGA since we only need to perform a single operation. As a result, GOPS/$ is maximized by using the CPU to compute the FFT.

Consider now that we have 128 FFT operations of size 128. In this case, the throughput benefits associated with offloading the process amortizes the cost of doing so. We may offload

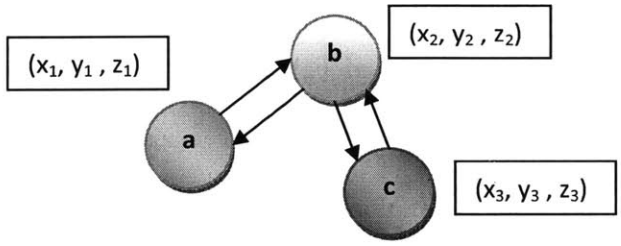
the task to either the FPGA or the GPU. If the FPGA already has FFT circuitry configured and assuming it performs FFTs substantially better than the GPU, then the task should be performed in the FPGA. However, if the FPGA is not configured with an FFT, then for a problem of this size the overhead associated with configuring the FPGA may preclude using it for this operation. Thus we will use the GPU if the FPGA does not already contain an FFT core. Now suppose that we want to perform 2048 FFTs of size 2048. The cost of configuring the FPGA for this task is amortized by the size of the job and thus it will always be beneficial to perform the FFT on the FPGA.

The result of this discourse is that choosing an execution methodology in a heterogeneous reconfigurable fabric may be a runtime consideration depending on the size of the operation to be performed and the configuration of the system. A load-balancing subsystem will need to simplify the task of profiling an application by determining some high-dependency variables. To keep the overhead associated with a run-time load balancer low, we will generate a condition set at profile-time and link each condition with a particular configuration and methodology.

4.3 Generalized Locality Optimization

Both the heterogeneous partitioning problem and thread placement in a multicore or FPGA can be generalized as locality issues. The general problem is how to map a high dimensional graph of cells to a low dimensional lattice of hardware in which the lattice exhibits communication constraints due to a distance function. I have experimented with a multidimensional variation of simulated annealing based on a simple premise: a graph of N nodes can be mapped to N dimensions such that all nodes are equidistant from one another. This can be easily understood by placing each node initially at (1,0,0…0), (0,1,0,…0), (0,0,1,…0) to form an N by N placement matrix which is initially the identity. The power cost of each node is

its activity times its Manhattan distance to each of its precedents. This cost is represented in an N x N activity matrix. The cost of a placement is the product of the activity matrix and the placement matrix. By taking the absolute value of the activity-placement product matrix we can compute the wire distance times the activity for each dimension. By multiplying the product matrix by dimensionality communication factors we can reduce the cost to a scalar quantity. Figure 4-3 demonstrates a graph and equations for its cost function. This generalized cost function can easily be programmed into a spreadsheet.



$$Cost = [f_1 \quad f_2 \quad f_3] abs\left(\begin{bmatrix} -a & a & 0 \\ b & -2b & b \\ 0 & c & -c \end{bmatrix}\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}\right)\begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

Figure 4-3: A very simply case of 3 nodes and the cost function for them. By modulating the dimensionality factors in the cost we can project a graph from N dimensions to 2 dimensions. We use the absolute value of each element of the cost matrix and placement product to compute the activity times the Manhattan distance of each arc.

Simulated annealing is a general optimization strategy in which random modifications are attempted and accepted probabilistically dependent on the net effect of the permutation on the cost function. If a modification increases the cost function, there is a finite probability dependent on a temperature parameter, that the permutation will be accepted. In this case, a cell is allowed to randomly move in any direction in the integer lattice provided there is no collision. In order to project the precedent graph to a two-dimensional lattice the dimensionality is reduced gradually by modulating the cost factor for high dimensional communication as the temperature parameter of the simulated anneal decreases. When T=0 any circuit placement that contains higher

dimensional terms will have infinite cost and only permutations that reduce the cost function will be accepted. Knots in a circuit placement can be untied by increasing the temperature and lifting the circuit graph into a higher dimension at various stages of the simulated anneal. In order to create extra space for cells to move in, the placement matrix is may be multiplied by a constant factor.

I have achieved positive results for this algorithm on a few obvious graphs by producing the temperature schedule by hand. The simple cases tested include a 4 x 4 graph in which all nodes are connected with their immediate neighbors and a linear pipeline. These two test cases were chosen because they have an obvious global optimum. This multidimensional simulated annealing approach apart from being mathematically interesting is currently very slow and takes several minutes and multiple user-interactions to converge to the global optimum even for these simple cases. Simulated annealing tends to work as a brute-force optimization mechanism and there seems to be an endless number of ways to tweak the algorithm that could potentially improve performance.

4.4 Conclusions and Future Work

This thesis has presented a number of methods and tools for the design and development of dynamic dataflow in a spreadsheet. Emerging reconfigurable architectures obviate the need for a new software development paradigm. The current toolset for multicore and FPGA development is inadequate for the needs of software developers. Spreadsheets obviate the parallelism in a system design and provide a simple visual development environment with minimal time-to-gratification. The complexities of spreadsheet-driven dynamic reconfiguration are still unsolved. Many layers of abstraction still separate the simple programming model of a

spreadsheet to compilation on an FPGA or Multicore. These multiple compiler layers ultimately slow down the dynamic programming model. For multicore and GPU chips, vendor specific global memory optimization mechanisms will probably require dynamic compilation. Since our programming environment conveniently matches the hardware environment a better low-level programming environment can hopefully be built within a spreadsheet.

Bibliography

[1] M. Bohr, "Interconnect Scaling - The Real Limiter to High Performance ULSI," International Electron Devices Meeting, p. 241, (1995).

[2] S. Das, A. Chandrakasan, and R. Reif. "Calibration of Rent's-Rule Models for Three-Dimensional Integrated Circuits." IEEE Trans. on VLSI Systems, vol. 12, no. 4, pp. 359-366, Apr. 2004.

[3] I.Koren, Z. Koren "Defect tolerance in VLSI circuits: techniques and yield analysis ," Proceedings of the IEEE , vol.86, no.9, pp.1819-1838, Sep 1998

[4] C. Constantinescu, "Trends and Challenges in VLSI Circuit Reliability," IEEE Micro ,vol. 23, no. 4, pp. 14-19, July/August, 2003.

[5] W. Thies, M. Karczmarek, and S. Amarasinghe. "StreamIt: A Language for Streaming Applications"

[6] W.J. Dally, U.J. Kapasi, B. Khailany, J. H. Ahn, A. Das. Stream Processors: Programmability with Efficiency ACM Queue, Vol. 2, No. 1, March 2004, pages 52-62.

[7] I. Page. Constructing hardware-software systems from a single description . Journal of VLSI Signal Processing, 12(1), pp. 87-107, 1996.

[8] Impulse Accelerated Technologies Corporate Website. http://www.impulseC.com

[9] O. Storaasli, D. Strenski. Exploring Accelerating Science Applications with FPGAs

[10] D. Lau, O. Pritchard, "Rapid System-On-AProgrammable-Chip Development and Hardware Acceleration of ANSI C Functions," in Proc. 16th International Conference on Field Programmable Logic and Applications (FPL 2006), (Madrid, Spain, August 28-30, 2006).

[11] D. Bennett, E. Dellinger, J. Mason, P. Sundarajan, An FPGA-oriented target language for HLL compilation. http://gladiator.ncsa.uiuc.edu/PDFs/rssi06/presentations/13_Dave_Bennett.pdf

[12] E. Neuwirth. Realtime Fourier synthesis – Sound generation http://sunsite.univie.ac.at/Spreadsite/fourier/fourtone.htm

[13] http://www.milezero.org/index.cgi/music/tools/excel/the_beat_goes_on.html

[14] R. Gradwohl, R. Fateman. Lisp and Symbolic Functionality in an Excel Spreadsheet: Development of an OLE Scientific Computing Environment

[15] Fuller, D. A., Mujica, S. T., and Pino, J. A. 1993. The design of an object-oriented collaborative spreadsheet with version control and history management. In Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice

[16] R. Hettinger. Python Cookbook : Spreadsheet http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/355045

[17] U. Eriksson http://siag.nu/siag/

[18] Abelson, H. and Sussman, G. J. 1996 Structure and Interpretation of Computer Programs. 2nd. MIT Press.

[19] R.M. Snyder. A Metaprogramming Pattern for Creating Java Class Functions Using a Spreadsheet. http://sais.aisnet.org/2006/Synder-SAIS2006-paper.pdf

[20] Lew, A. and Halverson, R. 1995. A FCCM for dataflow (spreadsheet) programs. In Proceedings of the IEEE Symposium on Fpga's For Custom Computing Machines (April 19 - 21, 1995).

[21] A. Yoder, D. Cohn. Domain-specific and general-purpose aspects of spreadsheet languages www-sal.cs.uiuc.edu/~kamin/dsl/papers/yoder.ps

[22] A. Hirsch, A. Leiserson. A MIDI Controlled Sample-Based Synthesizer. http://web.mit.edu/6.111/www/s2004/PROJECTS/1/index.htm

[23] Xilinx Inc. Virtex-5 Libraries Guide for Schematic Designs. http://toolbox.xilinx.com/docsan/xilinx82/books/docs/v5lsc/v5lsc.pdf

[24] http://www.Bluespec.com

[25] K. Kelley, A. Hirsch, D. Qumsiyeh, M.M. Tobenin. Sequentializing Bluespec. http://fpgaos.com/bs/SequentializingBluespec.pdf

[26] N. Pippenger, G.D. Stamoulis, J.N. Tsitsiklis. On a Lower Bound for the Redundancy of Reliable Networks with Noisy Gates. http://dspace.mit.edu/bitstream/1721.1/3177/1/P-1942-21258897.pdf

[27] B. Bridgford, C. Carmichael, C.W. Tseng. XAPP779 Correcting Single-Event Upsets in Virtex-II Platform FPGA Configuration Memory. Xilinx Inc.

[28] K. Kwiat, W. Debany, S. Hariri, "Software Fault Tolerance Using Dynamically Reconfigurable FPGAs," glsvlsi, p. 0039, 6th Great Lakes Symposium on VLSI, 1996

[29] Alderighi, M.; D'Angelo, S.; Metra, C.; Sechi, G.R., "Novel fault-tolerant adder design for FPGA-based systems," *On-Line Testing Workshop, 2001. Proceedings. Seventh International* , vol., no., pp.54-58, 2001

[30] M. Ruffoni, and A. Bogliolo, "Direct Measures of Path Delays on Commercial FPGA Chips," in *Proc. of IEEE Workshop on Signal Propagation on Interconnects*, 2002