Architectural Implications of Bit-level Computation in Communication Applications

by

David Wentzlaff

B.S.E.E., University of Illinois at Urbana-Champaign 2000

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2002

BARKER
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 1 8 2002
LIBRARIES

© Massachusetts Institute of Technology 2002. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
September 3, 2002

Certified by....... ....
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by........ ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Architectual Implications of Bit-level Computation in Communication Applications

by

David Wentzlaff

Submitted to the Department of Electrical Engineering and Computer Science
on September 3, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

In this thesis, I explore a sub domain of computing, *bit-level communication processing*, which has traditionally only been implemented in custom hardware. Computing trends have shown that application domains previously implemented only in special purpose hardware are being moved into software on general purpose processors. If we assume that this trend continues, we must as computer architects reevaluate and propose new superior architectures for current and future application mixes. I believe that *bit-level communication processing* will be an important application area in the future and hence in this thesis I study several applications from this domain and how they map onto current computational architectures including microprocessors, tiled architectures, FPGAs, and ASICs. Unfortunately none of these architectures is able to efficiently handle *bit-level communication processing* along with general purpose computing. Therefore I propose a new architecture better suited to this task.

Thesis Supervisor: Anant Agarwal
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Anant Agarwal for advising this thesis and imparting on me words of wisdom. Chris Batten was a great sounding board for my ideas and he helped me collect my thoughts before I started writing. I would like to thank Matthew Frank for helping me with the mechanics of how to write a thesis. I thank Jeffrey Cook and Douglas Armstrong who helped me sort my thoughts early on and reviewed drafts of this thesis. Jason E. Miller lent his photographic expertise by taking photos for my appendix. Walter Lee and Michael B. Taylor have been understanding officemates throughout this thesis and have survived my general crankiness over the summer of 2002. Mom and Dad have given me nothing but support through this thesis and my academic journey. Lastly I would like to thank DARPA, NSF, and Project Oxygen for funding this research.

Contents

1 Introduction **13**

1.1 Application Domain . . . 14

1.2 Approach . . . 16

2 Related Work **19**

2.1 Architecture . . . 19

2.2 Software Circuits . . . 20

3 Methodology **21**

3.1 Metrics . . . 21

3.2 Targets . . . 23

3.2.1 IBM SA-27E . . . 23

3.2.2 Xilinx . . . 25

3.2.3 Pentium . . . 27

3.2.4 Raw . . . 29

4 Applications **33**

4.1 802.11a Convolutional Encoder . . . 33

4.1.1 Background . . . 33

4.1.2 Implementations . . . 37

4.2 8b/10b Block Encoder . . . 43

4.2.1 Background . . . 43

4.2.2 Implementations . . . 44

5 Results and Analysis **49**

5.1 Results 49

5.1.1 802.11a Convolutional Encoder 49

5.1.2 8b/10b Block Encoder 53

5.2 Analysis 56

6 Architecture **59**

6.1 Architecture Overview 59

6.2 Yoctoengines 63

6.3 Evaluation 67

6.4 Future Work 68

7 Conclusion **71**

A Calculating the Area of a Xilinx Virtex II **73**

A.1 Cracking the Chip Open 74

A.2 Measuring the Die 78

A.3 Pressing My Luck 80

List of Figures

1-1 An Example Wireless System 15

3-1 The IBM SA-27E ASIC Tool Flow 24
3-2 Simplified Virtex II Slice Without Carry Logic 26
3-3 Xilinx Virtex II Tool Flow 28
3-4 Raw Tool Flow 30

4-1 802.11a Block Diagram 34
4-2 802.11a PHY Expanded View taken from [18] 35
4-3 Generalized Convolutional Encoder 36
4-4 802.11a Rate 1/2 Convolutional Encoder 37
4-5 Convolutional Encoders with Feedback 38
4-6 Convolutional Encoders with Tight Feedback 38
4-7 Inner-Loop for Pentium *reference* 802.11a Implementation 39
4-8 Inner-Loop for Pentium *lookup table* 802.11a Implementation 40
4-9 Inner-Loop for Raw *lookup table* 802.11a Implementation 41
4-10 Inner-Loop for Raw *POPCOUNT* 802.11a Implementation 42
4-11 Mapping of the *distributed* 802.11a convolutional encoder on 16 Raw tiles 42
4-12 Overview of the 8b/10b encoder taken from [30] 45
4-13 8b/10b encoder *pipelined* 46
4-14 Inner-Loop for Pentium *lookup table* 8b/10b Implementation 46
4-15 Inner-Loop for Raw *lookup table* 8b/10b Implementation 47

5-1 802.11a Encoding Performance (MHz.) 51
5-2 802.11a Encoding Performance Per Area (MHz./mm^2.) 52
5-3 8b/10b Encoding Performance (MHz.) 54
5-4 8b/10b Encoding Performance Per Area (MHz./mm^2.) 55

6-1 The 16 Tile Raw Prototype Microprocessor with Enlargement of a Tile 60
6-2 Interfacing of the Yoctoengine Array to the Main Processor and Switch 62
6-3 Four Yoctoengines with Wiring and Switch Matrices 64
6-4 The Internal Workings of a Yoctoengine 65

A-1 Aren't Chip Carriers Fun 74
A-2 Four Virtex II XC2V40 Chips 75
A-3 Look at all Those Solder Bumps 75
A-4 The Two Parts of the Package 76
A-5 Top Portion of the Package, the Light Green Area is the Back Side of the Die 77
A-6 Removing the FR4 from the Back Side of the Die 78
A-7 Measuring the Die Size with Calipers 79
A-8 Don't Flex the Die 80

List of Tables

3.1 Summary of Semiconductor Process Specifications 22

5.1 802.11a Convolutional Encoder Results 50

5.2 8b/10b Encoder Results 54

6.1 Yoctoengine Instruction Coding Breakdown 67

Chapter 1

Introduction

Recent trends in computer systems have been to move applications that were previously only implemented in hardware into software on microprocessors. This has been motivated by several factors. Firstly microprocessor performance has been steadily increasing over time. This has allowed more and more applications that previously could only be done in ASICs and special purpose hardware, due to their large computation requirements, to be done in software on microprocessors. Also, added advantages such as decreased development time, ease of programming, the ability to change the computation in the field, and the economies of scale due to the reuse of the same microprocessor for many applications have influenced this change.

If we believe that this trend will continue, then in the future we will have one computational fabric that will need to do the work that is currently done by all of the chips inside of a modern computer. Thus we will need to pull all of the computation that is currently being done inside of helper chips onto our microprocessors. We have already seen this being done in current computer systems with the advent of all software modems.

Two consequences follow from the desire to implement all parts of a computer system in one computational fabric. First, the computational requirements of this one computational fabric are now much higher. Second, the mix of computation that it will be doing is significantly different from applications that current day microprocessors are optimized for. Thus if we want to build future architectures that can handle this new application mix, we need to develop architectural mechanisms that efficiently handle conventional applications, SpecInt and SpecFP [8], multimedia applications, which have been the focus of significant research recently, and the before mentioned applications which we will call software circuits.

In modern computer systems most of the helper chips are there to communicate with different devices and mediums. Examples include sound cards, Ethernet cards, wireless communication cards, memory controllers and I/O protocols such as SCSI and Firewire. This research work will focus on the subset of software circuits for communication systems, examples being Ethernet cards (802.3) and wireless communication cards (802.11a, 802.11b). Communication systems are chosen as a starting point for this research for two reasons. One, it is a significant fraction of the software circuits domain. Secondly, if communication bandwidth is to continue to grow as is foreseen, the computation needed to handle it will become a significant portion of our future processing power. This is mostly due to the fact that communication bandwidth is on a steep exponentially increasing curve. This research will further focus on bit-level computation contained in communication processing. Fine grain bit-level computation is an interesting sub-area of communications processing, because unlike much of the rest of communications processing, it is not easily parallelizable on word oriented systems because very fine grain, bit-level, communication is needed. Examples of this type of computation include error-correcting codes, convolutional codes, framers, and source coding.

1.1 Application Domain

In this thesis, I investigate several kernels of communications applications that exhibit non-word-aligned, bit-level, computation that is not easily parallelized on word-oriented parallel architectures such as Raw [29, 26]. Figure 1-1 shows a block diagram of an example software radio wireless communication system. The left dotted box contains computation which transforms samples from an analog to digital converter into a demodulated and decoded bit-stream. This box typically does signal process-

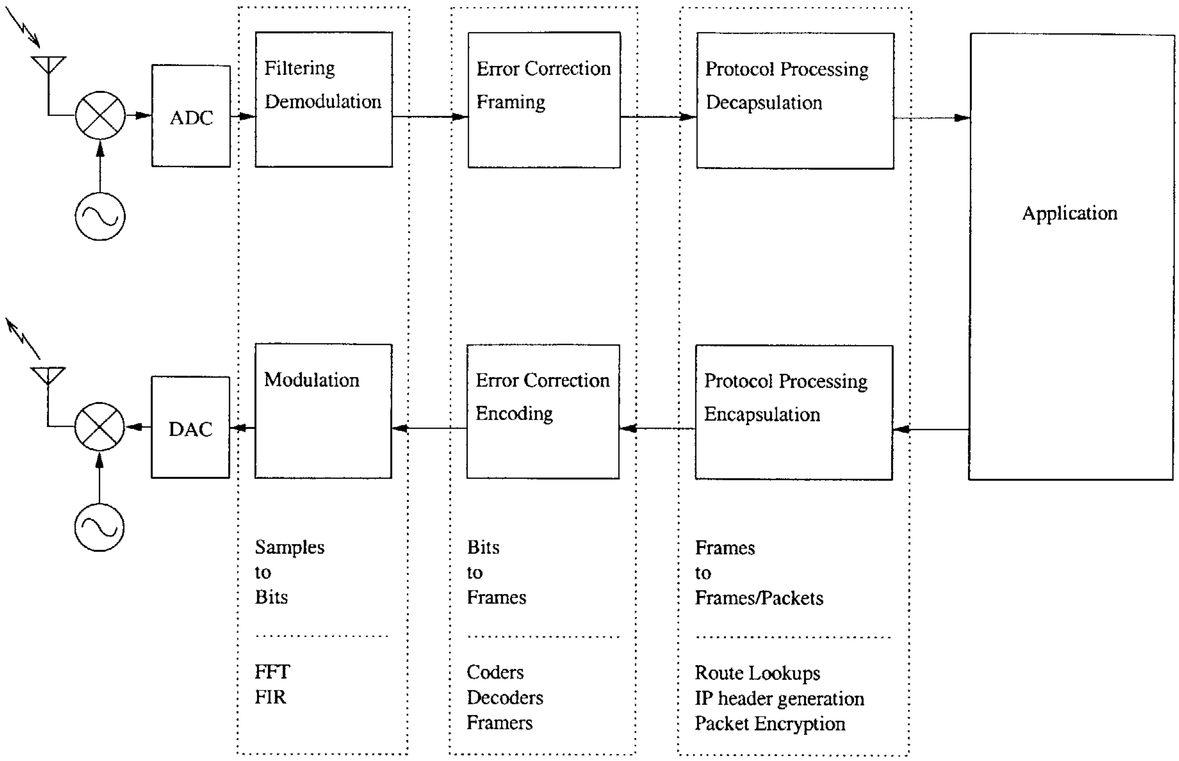
ADC
Filtering
Demodulation
Error Correction
Framing
Protocol Processing
Decapsulation
Application
DAC
Modulation
Error Correction
Encoding
Protocol Processing
Encapsulation
Samples
to
Bits
FFT
FIR
Bits
to
Frames
Coders
Decoders
Framers
Frames
to
Frames/Packets
Route Lookups
IP header generation
Packet Encryption

Figure 1-1: An Example Wireless System

ing on samples of data. In signal processing, the data is formatted in either a fixed point or floating point format, and thus can be efficiently operated on by a word-oriented computer. In the right most dotted box, protocol processing takes place. This transforms frames received from a framer into higher level protocols. An example of this is transforming Ethernet frames into IP packets and then doing an IP checksum check. Once again frames and IP packets can be packed into words without too much inefficiency. This thesis focuses on the middle dotted box in the figure. It is interesting to see that the left and right boxes contain word-oriented computation, but the notion of word-aligned data disappears in this box. This middle processing step takes the bit-stream from the demodulation step and operates on the bits to remove encoding and error correction. This processing is characterized by the lack of aligned data and is essentially a transformation from one amorphous bit-stream to a different bit-stream. This thesis will focus on this unaligned bit domain. We will call this application domain *bit-level communication processing.*

The main reason that these applications are not able to be easily parallelized on parallel architectures is because the parallelism that exists in these applications is inside of one word. Unfortunately for current parallel architectures composed of word-oriented processors, operations that exist on word-oriented processors are not well suited for exploiting unstructured parallelism inside of a word.

Two possible solutions exist to try to exploit this parallelism in normal parallel architectures yet they fall short. One approach is to use a whole word to represent a bit and thus a word-oriented parallel architecture can be used to exploit parallelism. Unfortunately this type of mapping is severely inefficient, due to the fact that a whole word is being used to represent a bit, and if any of the computation has communication, the communication delay will be high because the large word-oriented functional units require more space than bit-level functional units. Thus they physically have to be located further apart. Finally, if all of these problems are solved, there are still problems if sequential dependencies exist in the computation. If the higher communication latency is on the critical path of the computation, the throughput of the application will be lowered. Thus for these reasons, simply scaling up the Raw architecture is not appropriate to speedup these applications.

The second possible parallel architecture that could exploit parallelism in these applications is sub-word SIMD architectures such as MMX [22]. Unfortunately, if applications exhibit intra-word communication, these architectures are not suited because they are set up to parallelly operate on and not communicate between sub-words.

1.2 Approach

To investigate these applications I feel that first we need a proper characterization of these applications on current architectures. To that end, I coded up two characteristic applications in 'C', Verilog, and assembly for conventional architectures (x86), Raw architectures, the IBM ASIC flow, and FPGAs and compared relative speed and area tradeoffs between these different approaches. This empirical study has allowed me quantify the differences between the architectures. From these results I was able to quantitatively determine that tiled architectures do not efficiently use silicon area for *bit-level communication processing.* Lastly I discuss how to augment the Raw tiled architecture with an array of finer grain computational elements. Such an architecture is more capable of handling both general purpose computation efficiently and *bit-level communication processing* in an area efficient manner.

Chapter 2

Related Work

2.1 Architecture

Previous projects have investigated what architectures are good substrates for bit-level computation in general. Traditionally many of these applications have been been implemented in FPGAs. Examples of the most advanced commercial FPGAs can be seen in Xilinx's product line of Virtex-E and Virtex-II FPGAs [33, 34]. Unfortunately for FPGAs, the difficulty of programming and inability of virtualization have stopped them from becoming a general purpose computing platform. One attempt made to use FPGAs for general purpose computing was made by Babb in [2]. In Babb's thesis, he investigates how to compile 'C' programs to FPGAs directly using his compiler named deepC. Unfortunately, due to the fact that FPGAs are not easily virtualized, large programs cannot currently be compiled with deepC.

Other approaches that try to excel at bit-level computation, but still be able to do general purpose computation include FPGA-processor hybrids. They range widely on whether they are closer to FPGAs or modern day microprocessors. Garp [6, 7] for example is a project at The University of California Berkeley that mixes a processor along with a FPGA core on the same die. In Garp, both the FPGA and the processor can share data via a register mapped communication scheme. Other architectures put reconfigurable units as slave functional units to a microprocessor. Examples of this type of architecture include PRISC [25] and Chimaera [13].

The PipeRench processor [11] is a larger departure from a standard microprocessor with a FPGA put next to it. PipeRench contains long rows of flip-flops with lookup tables in between them. In this respect it is very similar to a FPGA but has the added bonus that it has a reconfigurable data-path that can be changed very quickly on a per-row, per-cycle basis. Thus it is able to be virtualized and allows larger programs to be implemented on it.

A different approach to getting speedup on bit-level computations is by simply adding the exact operations that your application set needs to a general purpose microprocessor. In this way you build special purpose computing engines. One example of this is the CryptoManiac [31, 5]. The CryptoManiac is a processor designed to run cryptographic applications. To facilitate these applications, the CryptoManiac supports operations common in cryptographic applications. Therefore if it is found that there are only a few extra instructions needed to efficiently support bit-level computation, a specialized processor may be the correct alternative.

2.2 Software Circuits

Other projects have investigated implementing communication algorithms that previously have only been done in hardware on general purpose computers. One example of this can be seen in the SpectrumWare project [27]. In the SpectrumWare project, participants implemented many signal processing applications previously only done in analog circuitry for the purpose of implementing different types of software radios. This project introduced an important concept for software radios, temporal decoupling. Temporal decoupling allows buffers to be placed between different portions of the computation thus increasing the flexibility of scheduling signal processing applications.

Chapter 3

Methodology

One of the greatest challenges of this project was simply finding a way to objectively compare very disparate computing platforms which range from a blank slate of silicon all the way up to a state of the art out-of-order superscalar microprocessor. This section describes the architectures that were mapped to, how they were mapped to, how the metrics used to compare them were derived, and what approximations were made in this study.

3.1 Metrics

The first step in objectively comparing architectures is to choose targets that very nearly reflect each other in terms of semiconductor process. While it is possible to scale between different semiconductor feature sizes, not everything simply scales. Characteristics that do not scale include wire delay and the proportions of gate sizes. Thus a nominal feature size of 0.15μm drawn was chosen. All of the targets chosen with the exception of the Pentium 4 (0.13μm.) and Pentium 3 (0.18μm.) are fabricated with this feature size. Because everything is essentially on the same feature size this thesis will use area in mm^2. as one of its primary metrics. One assumption that is made in this study is that voltage does not significantly change performance of a circuit. While the targets are essentially all on the same process, they do have differences when it comes to nominal operational voltages. Table 3.1 shows the process

Target	Foundry	L_{drawn} (μm.)	$L_{effective}$ (μm.)	Nominal Voltage (Volts)	Layers of Metal	Type of Metal
IBM SA-27E ASIC	IBM	0.15	0.11	1.8	6	Cu
Xilinx Virtex II	UMC	0.15	0.12	1.5	8	Al
Intel Pentium 4 Northwood 2.2GHz.	Intel	0.13	0.07	1.75	6	Cu
Intel Pentium 3 Coppermine 993MHz.	Intel	0.18	Unknown	1.7	6	Al
Raw 300MHz.	IBM	0.15	0.11	1.8	6	Cu

Table 3.1: Summary of Semiconductor Process Specifications

parameters for the differing targets used in this thesis.

To objectively compare the performance of the differing platforms the simple metric of frequency was used. The applications chosen have clearly defined rates of operation which are used as a performance metric. For the targets that are software running on an processor, the testing of the applications was done over a randomly generated input of several thousand samples. This was done to mitigate any variations that result from differing paths through the program. This method should give an average case performance result. In the hardware designs, performance was measured by looking at the worse case combinational delay of the built circuit. In order for the the delay to the pins and other edge effects from accidentally being factored into the hardware timing numbers, registers were placed at the beginning and end of the design under test. These isolation registers were included in the area calculations for the hardware implementations. Lastly the latency through any of the software or hardware designs was not measured and deemed unimportant because all of the designs have relatively low latency and the applications are bandwidth sensitive not latency sensitive.

3.2 Targets

3.2.1 IBM SA-27E

Overview

The IBM SA-27E ASIC process is a standard cell based ASIC flow. It has a drawn transistor size of $0.15\mu m$ and 6 layers of copper interconnect [16]. It was chosen primarily to show what the best case that can be done when directly implementing the applications in hardware looks like. While using a standard cell approach is not as optimal as a full-custom realization of the circuits, it is probably characteristic of what would be done if actually implementing the applications, because full-custom implementations are rarely done due to time to market constraints. Conveniently this tool flow was available for this thesis because the Raw research group is using this target for the Raw microprocessor.

Tool Flow

For this target, the applications were written in the Verilog Hardware Description Language (HDL). Verilog is a HDL which has similar syntax to 'C' and allows one to behaviorally and structurally describe circuits. The same code base was shared between this target and the Xilinx target. The applications were written in a mixture of behavioral and structural Verilog.

To transform the Verilog code into actual logic gates a Verilog synthesizer is needed. Synopsys's Design Compiler 2 was utilized. Figure 3-1 shows the flow used. Verilog with VPP [1] macros is first processed into plain Verilog code. Next, if the circuit is to be simulated, the Verilog is passed onto Synopsys's VCS Verilog simulator. If synthesis is desired, the Verilog along with the IBM SA-27E technology libraries are fed into Synopsys's Design Compiler 2 (DC2). Design Compiler 2 compiles, analyzes, and optimizes the output gates. A design database is the output of DC2 along with area and timing reports. To get optimal timing, the clock speed was slowly

---

[1] VPP is a Verilog Pre-Processor which expands the language to include auto-generating Verilog code, much in the same way that CPP allows for macro expansion in 'C'.

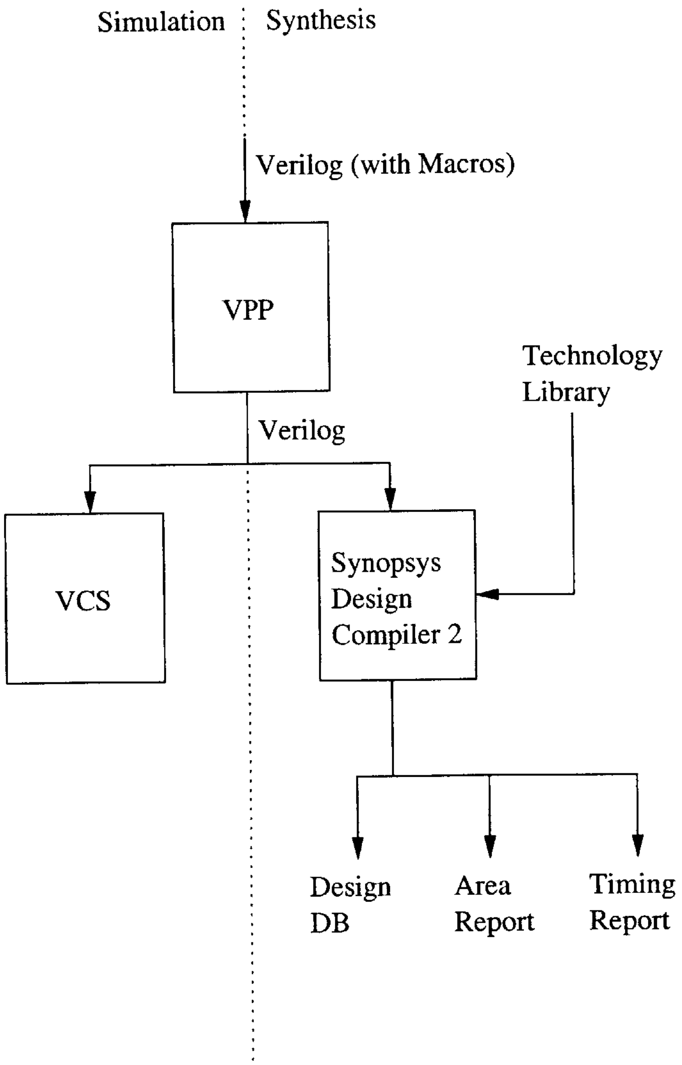
Simulation
Synthesis
Verilog (with Macros)
VPP
Technology Library
Verilog
VCS
Synopsys Design Compiler 2
Design DB
Area Report
Timing Report

Figure 3-1: The IBM SA-27E ASIC Tool Flow

dialed up until the synthesis failed. This was done because it is well known that Verilog optimizers don't work their best unless tightly constrained. This target's area numbers used in this report are simply the area of the circuit implemented in this process, not including overheads such as pins.

3.2.2 Xilinx

Overview

This target is Xilinx's most recent Field Programmable Gate Array (FPGA), the Virtex II [34]. A FPGA such as the Virtex II is a fine grain computational fabric built out of SRAM based lookup tables (LUTs). The primary computational element in a Virtex II is a Configurable Logic Block (CLB). CLBs are connected together by a statically configurable switch matrix. On a Virtex II, a CLB is composed of four Slices which are connected together by local interconnect and are connected into the switch matrix. A Slice on a Virtex II is very similar to what a CLB looked like Xilinx's older 4000 series FPGAs. Figure 3-2 shows a simplified diagram of a Slice. This LUT based structure along with the static interconnect system allows the easy mapping of logic circuits onto FPGAs. Also LUTs can be reconfigured so that they can be used as small memories. The configuration of a SRAM based FPGA is a slow process of serially shifting in all of the configuration data.

Tool Flow

The application designs for the Xilinx Virtex II share the same Verilog code with the IBM SA-27E design. The tool flow for creating FPGAs is the most complicated out of all of the explored targets. Figure 3-3 shows the tool flow which has some in commonality with the IBM flow. The basic flow starts by feeding the Verilog code to a FPGA specific synthesis tool, FPGA Compiler 2 which generates a EDIF file. The EDIF file is transformed to a form readable by the Xilinx tools, NGO. Then the NGO is fed into NGD build which is similar to a linking phase for a microprocessor. The NGO is then mapped into LUTs, placed, and routed. At this point the tools are

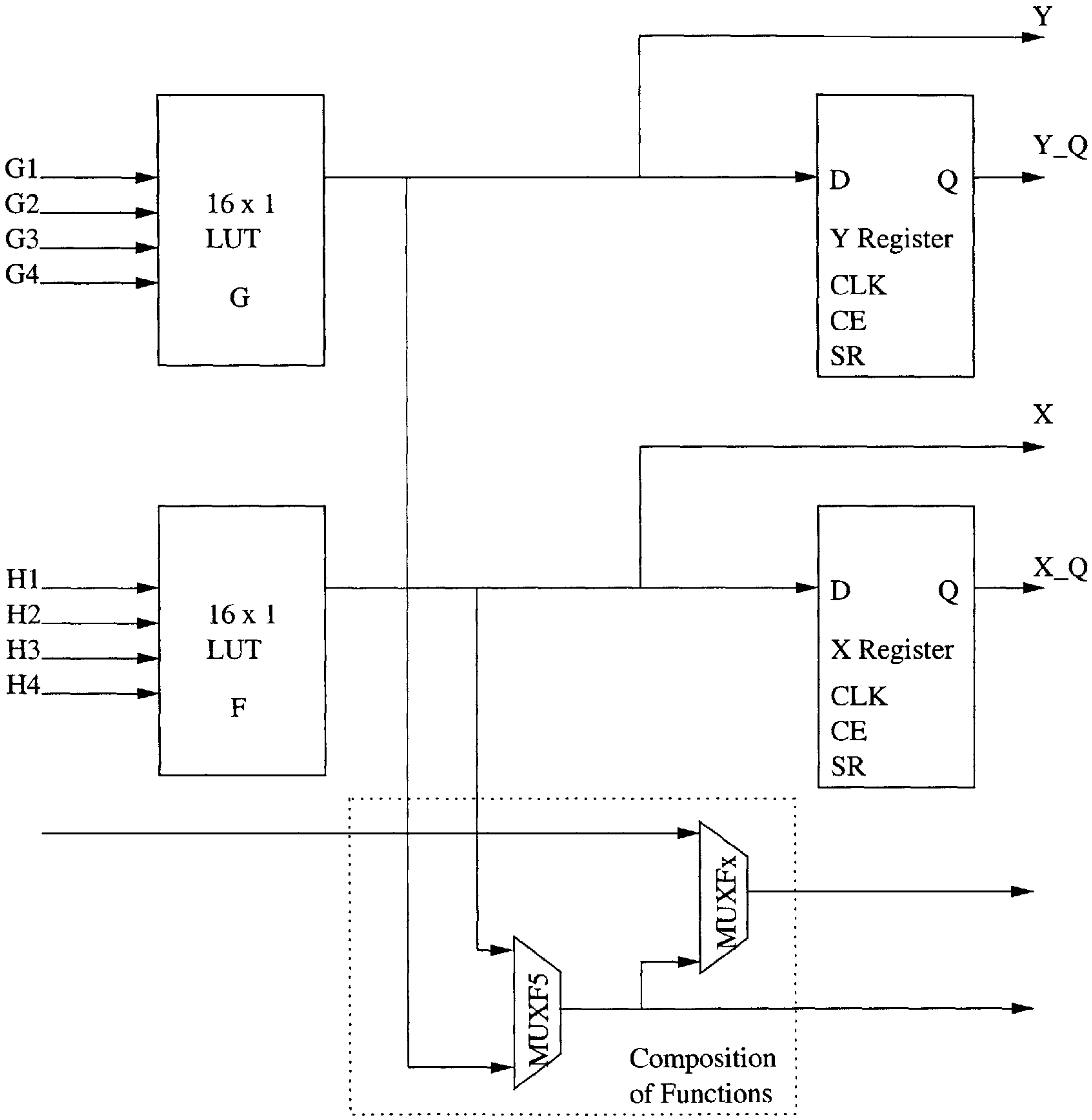
Y
Y_Q
X
X_Q
G1
G2
G3
G4
16 x 1
LUT
G
D
Q
Y Register
CLK
CE
SR
H1
H2
H3
H4
16 x 1
LUT
F
D
Q
X Register
CLK
CE
SR
MUXFx
MUXF5
Composition
of Functions

Figure 3-2: Simplified Virtex II Slice Without Carry Logic

able to generate timing and area reports. BitGen can also be run which generates the binary file to be loaded directly into the FPGA.

One interesting thing to note is that the area reports that the Xilinx tools generate are only in terms of number of flip-flops and Slices. Converting these numbers into actual mm^2. area numbers was not as simple as was to be expected. The area of a Slice was reverse engineered by measuring the die area of an actual Virtex II. Appendix A describes this process. The area used in this thesis is the size of one slice multiplied by the number of slices a particular application used.

3.2.3 Pentium

Overview

This target is a Pentium 4 microprocessor [15]. The Pentium 4 is Intel's current generation of 32-bit x86 compatible microprocessor. It is a 7 wide out-of-order superscalar with an execution trace cache. The same applications were also run on an older Pentium 3 for comparison purposes.

Tool Flow

The tool flow for the Pentium 4 is very simplistic. All of the code is written is 'C' and compiled with gcc 2.95.3 with optimization level of -O9. To gather the speed at which that applications run, the time stamp counter (TSC) was used. The TSC is a 64-bit counter on Intel processors (above a Pentium) which monotonically increases on every clock cycle. With this counter, accurate cycle counts can be determined for execution of a particular piece of code. To prevent memory hierarchy from unduly hurting performance, all source and result arrays were touched to make sure they were within the level of cache being tested before the test's timing commenced. To calculate the overall speed, the tests were run over many iterations of the loop and the overall time as per the TSC was divided by the iterations completed normalized to the clock speed to come up with the resultant performance. To calculate the area size, the overall chip area was used.

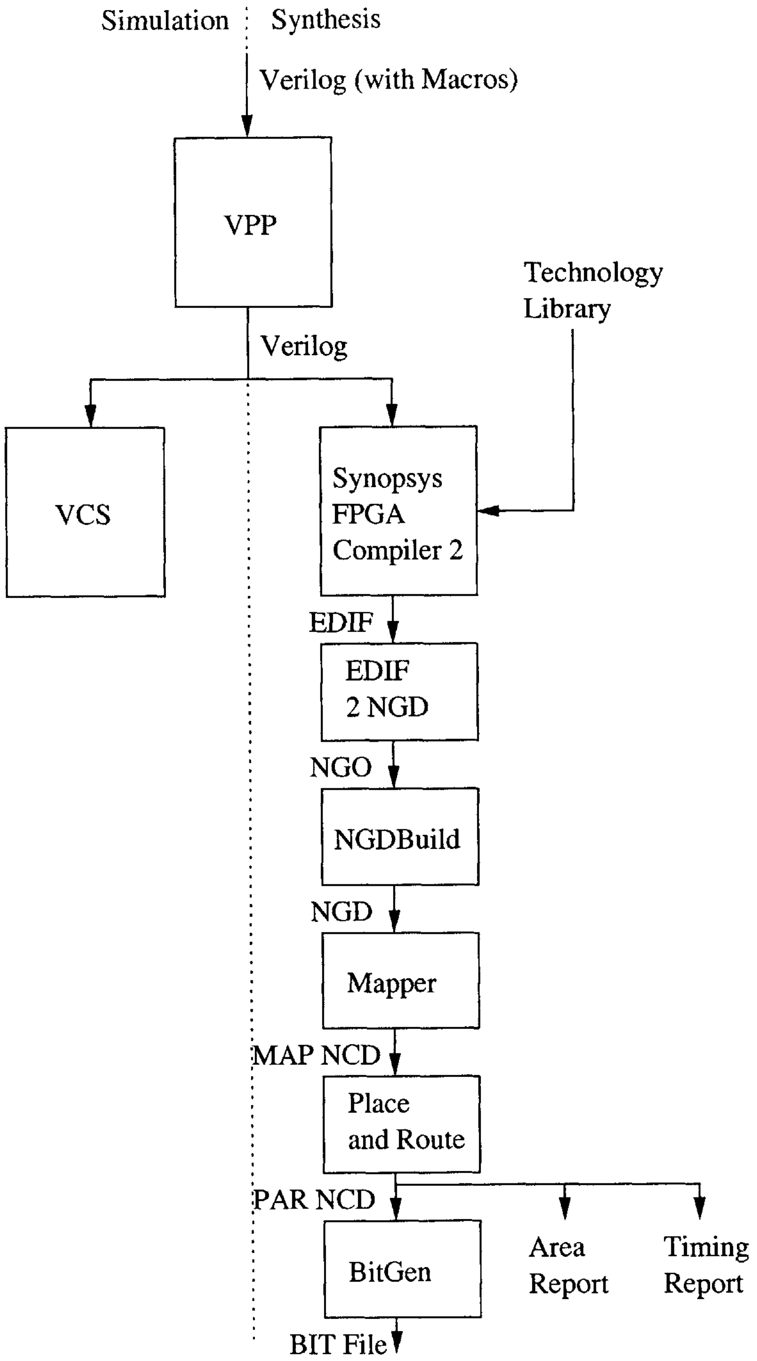
Simulation
Synthesis
Verilog (with Macros)
VPP
Technology Library
Verilog
VCS
Synopsys FPGA Compiler 2
EDIF
EDIF 2 NGD
NGO
NGDBuild
NGD
Mapper
MAP NCD
Place and Route
PAR NCD
BitGen
Area Report
Timing Report
BIT File

Figure 3-3: Xilinx Virtex II Tool Flow

3.2.4 Raw

Overview

The Raw microprocessor is a tiled computer architecture designed in the Computer Architecture Group at MIT. The processor contains 16 replicated tiles in a 4x4 mesh. A tile consists of a main processor, memory, two dynamic network routers, two static switch crossbars and a static switch processor. Tiles are connected to each of their four nearest neighbors via register mapped communication by two sets of static network interconnect and two sets of dynamic network interconnect. Each main processor is similar to a MIPS R4000 processor. It is an in-order single issue processor that has 32KB of data cache and 8K instructions of instruction store. The static switch processor handles compile time known communication between tiles. Each static switch processor has 8K instructions of instruction store.

Because of all of the parallel resources available and the closely knit communication resources, many differing computational models can be used to program Raw. For instance instruction level parallelism can be mapped across the parallel compute units as is discussed in [21]. Another parallelizing programming model is a stream model. Streaming exploits course grain parallelism between differing filters which are many times used in signal processing applications. A stream programming language for the Raw chip is being developed to compile a new programming language named StreamIt [12]. The applications in this study for Raw were written in 'C' and assembly and were hand parallelized with communication happening over the static network for peak performance.

The Raw processor is being built on IBM's SA-27E ASIC process. It is currently out to fab and chips should be back by the end of October 2002. More information can be found in [26].

Tool Flow

Compilation for the Raw microprocessor is more complicated than a standard single stream microprocessor due to the fact that there can be 32 (16 main processor

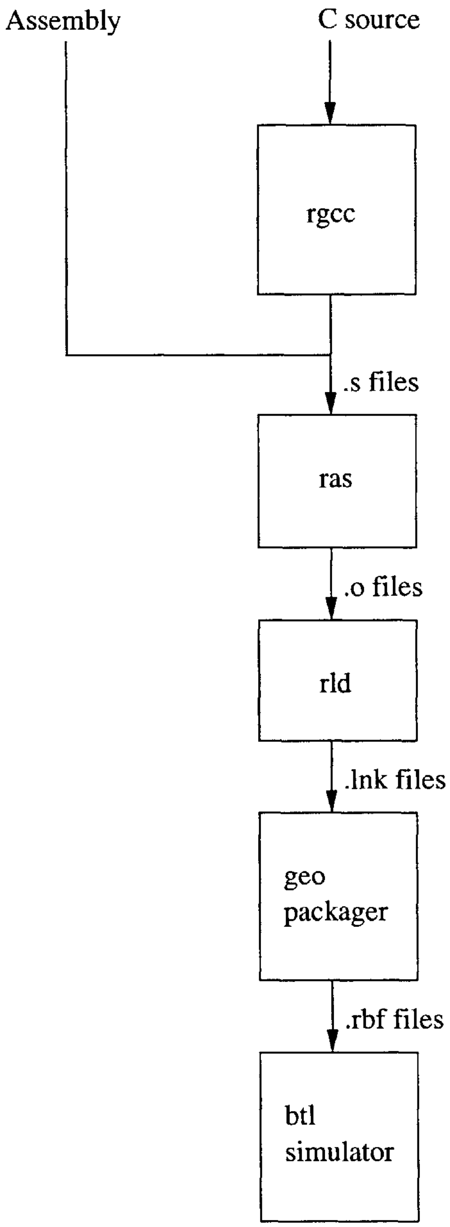
Assembly
C source
rgcc
.s files
ras
.o files
rld
.lnk files
geo packager
.rbf files
btl simulator

Figure 3-4: Raw Tool Flow

streams, 16 switch instruction streams) independent instruction streams operating at the same time. To solve the complicated problem of building binaries and simulating them on the Raw microprocessor, the Raw group designed the Starsearch build infrastructure. Starsearch is a group of common Makefiles that the Raw group shares that understands how to build and execute binaries on our various differing simulators. This has saved countless hours for new users to the Raw system by preventing them from having to set up their own tool flow.

Figure 3-4 shows the tool flow used in this thesis for Raw. This tool flow is conveniently orchestrated by Starsearch. The beginning of the tool flow is simply a port of the GNU compilation tool flow of gcc and binutils. geo is a geometry management tool which can take multiple binaries, one for each tile, and create self booting binaries using Raw's intravenous tool. This is the main difference from a standard microprocessor with only one instruction stream. Lastly the rbf file is booted on a cycle accurate model of the Raw processor called btl.

To collect performance statistics for the studied applications on Raw a clock speed of 300MHz. was assumed. Applications were simulated on btl and timed using Raw's built in cycle counters, CYCLE_HI and CYCLE_LO, which provide 64-bits worth of accuracy. Unlike on the Pentium, preheating the cache was not needed. This was because the data that was to be operated and the results all came in and left on the static network. To accomplish this, bC [2] models were written that streamed in and collected data from the static networks on the Raw chip thus preventing all of the problems that are caused by memory hierarchy.

Area calculations are easily made for the Raw chip because the Raw group has completed layout of the Raw chip. A tile is 4mm. x 4mm. This study assumes that as tiles are added the area scales linearly, which is a good approximation, but is not completely accurate because there is some area around a tile reserved for buffer placement, and there are I/O drivers on the periphery of the chip.

---

[2] bC is btl's extension language which is very similar to 'C' and allows for rapid prototyping of external devices to the Raw chip.

Chapter 4

Applications

4.1 802.11a Convolutional Encoder

4.1.1 Background

As part of the author's belief that benchmarks should be chosen from real applications and not simply synthetic benchmarks, the first characteristic application that will be examined is the convolutional encoder from 802.11a. IEEE 802.11a is the wireless Ethernet standard [17, 18] used in the 5GHz. band. In this band, 802.11a provides for transmission of information up to 54Mbps. Other wireless standards that have similar convolutional encoders to that of 802.11a include IEEE 802.11b [17, 19], the current WiFi standard and most widely used wireless data networking standard, and Bluetooth [3], a popular standard for short distance wireless data transmission.

Before we dive into too much depth of convolutional encoders we need to see where they fit into the overall application. Figure 4-1 shows a block diagram of the transmit path for 802.11a. Starting with IP packets, they get encapsulated in Ethernet frames. Then the Ethernet frames get passed into the Media Access Control (MAC) layer of the Ethernet controller. This MAC layer is typically done directly in hardware. The MAC layer is responsible for determining when the media is free to use and relaying physical problems up to higher level layers. The MAC layer then passes the data onto the Physical Layer (PHY) which is responsible for actually transmitting the

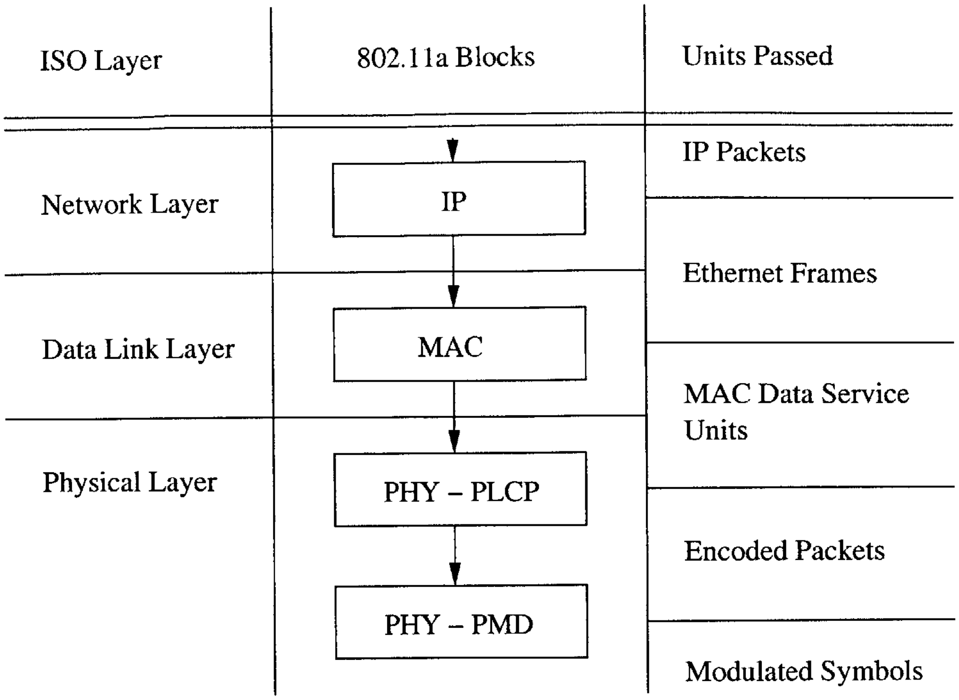
ISO Layer
802.11a Blocks
Units Passed
Network Layer
IP
IP Packets
Ethernet Frames
Data Link Layer
MAC
MAC Data Service Units
Physical Layer
PHY - PLCP
Encoded Packets
PHY - PMD
Modulated Symbols

Figure 4-1: 802.11a Block Diagram

data over the media, in this case radio waves. In wireless Ethernet the PHY has two sections. One that handles the data as bits, the physical layer convergence procedure (PLCP), and a second part that handles the data after it has been converted into symbols, the physical medium dependent (PMD) system. 802.11a uses orthogonal frequency division multiplexing (OFDM) which is a modulation technique that is more complicated than something like AM radio.

Figure 4-2 shows an expanded view of the PHY path. The input to this pipeline are MAC service data units (MSDUs), the format at which the MAC communicates with the PHY and the output is a modulated signal broadcast over the antenna. As can be seen in the first box, all data which passes over the airwaves, needs to first pass through forward error correction (FEC). In the 802.11a, this FEC comes in the form of a convolutional encoder. Because all of the data passes through the convolutional encoder, it must be able to operate at line speed to maintain proper throughput over the wireless link. It is advantageous to pass all data that is to be transmitted over a wireless link through a convolutional encoder because it provides some resilience to electro-magnetic interference. Without some form of encoding, this interference

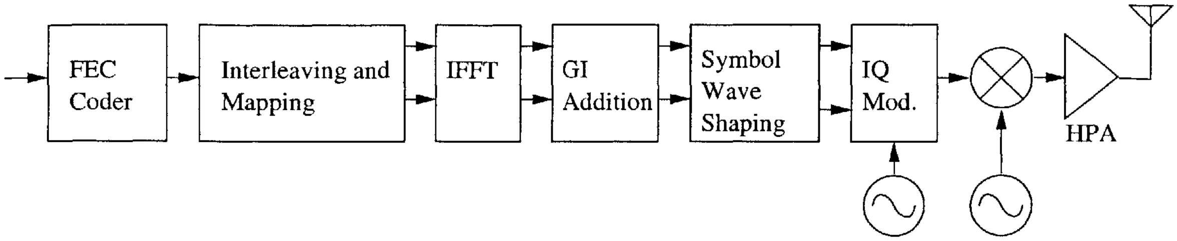

Figure 4-2: 802.11a PHY Expanded View taken from [18]

would cause corruption in the non-encoded data.

There are two main ways to encode data to add redundancy and prevent transmission errors, block codes and convolutional codes. A basic block code takes a block of n symbols in the input data and encodes it into a codeword in the output alphabet. For instance the addition of a parity bit to a byte is an example of a simple block code. The code takes 8 bits and maps into 9-bit codewords. The parity operation maps as follows, have all of the first 8 bits stay the same as the input byte and have the last bit be the XOR of all of the other 8 bits. After that one round this block code takes the next 8 bits as input and do the same operation. Convolutional codes in contrast do not operate on a fixed block size, but rather operate a bit at a time and contain memory. The basic structure of a convolutional encoder has k storage elements chained together. The input bits are shifted into these storage elements. The older data which is still stored in the storage elements shift over as the new data is added. The shift amount s can be one (typical) or more than one. The output is computed as a function of the state elements. A new output is computed whenever new data is shifted in. In convolutional encoders, multiple functions are many times computed simultaneously to add redundancy. Thus multiple output bits can be generated per input bit. Figure 4-3 shows a generalized convolutional encoder. The boxes with numbers in them are storage elements that shift over by s bits every encoding cycle. The function box, denoted with f, can actually represent multiple differing functions.

As can be seen from the previous description, a convolutional [1] encoder essentially

---

[1] Convolutional codes get their name from the fact that they can be modeled as the convolution of polynomials in a finite field, typically the extended Galois field GF(p^r). This is typically done by modeling the input stream as the coefficients of a polynomial and the tap locations as the coefficients

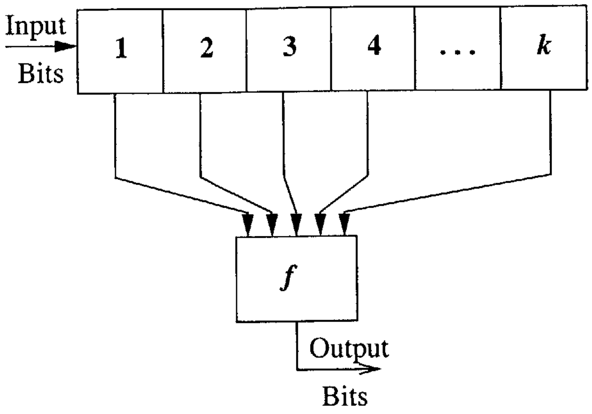

Figure 4-3: Generalized Convolutional Encoder

smears input information across the output information. This happens because, assuming that the storage shifts only one bit at a time ($s = 1$), one bit effects k output bits. This smearing helps the decoder detect and many times correct one bit errors in the transmitted data. Also, because it is possible to have multiple output bits for each cycle of the encoder, even more redundancy can be added. The rate of input bits compared to the output bits is commonly referred to the rate of the convolutional encoder. More mathematical groundings about convolutional codes can be found in [23], and a good discussion on convolutional codes, block codes and their decoding can be found in [24].

This application study uses the default convolutional encoder that 802.11a uses in poor channel quality. It is a rate 1/2 convolutional encoder and contains seven storage elements. 802.11a has differing encoders for different channel quality with rates of 1/2, 3/4, and 2/3. The shift amount for this convolutional encoder is one ($s = 1$). Figure 4-4 is a block level diagram of the studied encoder. This encoder has two outputs with differing tap locations, and uses XOR as the function it computes. The generator polynomials used are $g_0 = 133_8$ and $g_1 = 171_8$.

---

of a second polynomial. Then if you convolve the two polynomials modulo a prime polynomial and evaluate the resultant polynomial with $x = 1$ in the extended Galois field, you get the output bits.

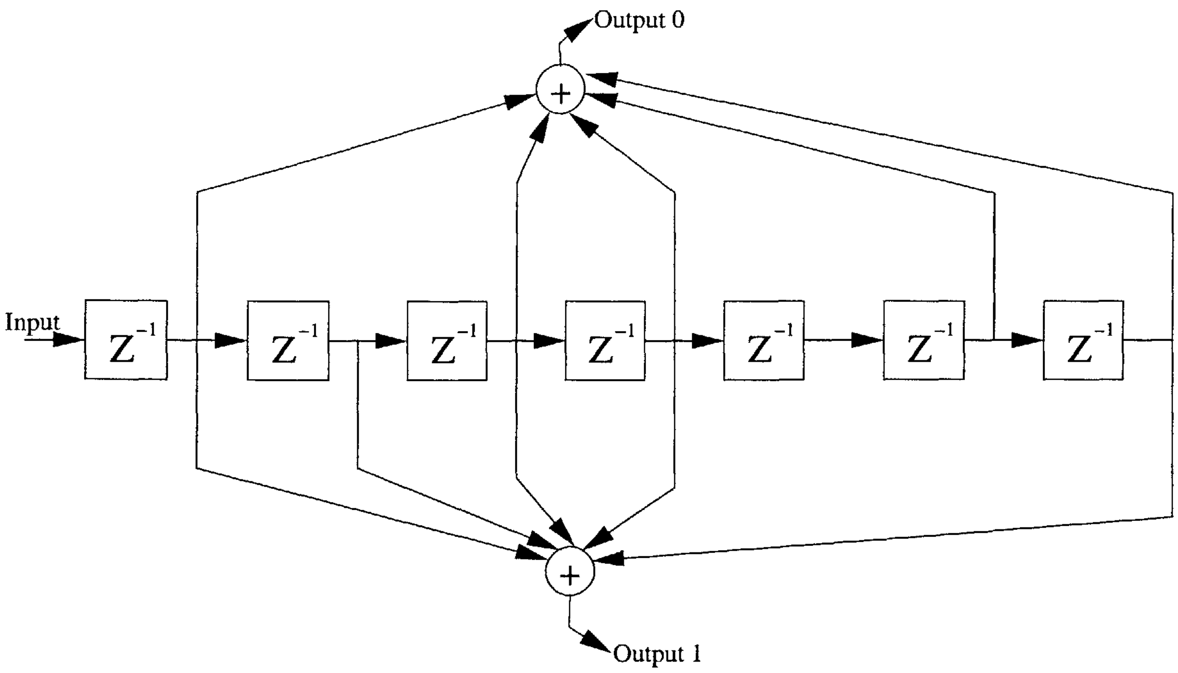

Figure 4-4: 802.11a Rate 1/2 Convolutional Encoder

4.1.2 Implementations

Verilog

This convolutional encoder and other applications similar to it, which include linear feedback shift registers, certain stream ciphers and other convolutional encoders all share similar form. This form is amazingly well suited for simple implementation into hardware. The basic form of this circuit is as chain of flip-flops serially hooked together. Several outputs of this chain are then logically XORed together producing the outputs of the circuit. For this design, the same code was used for synthesis to both the IBM SA-27E ASIC and Xilinx targets. While this design was not pipelined more than the trivial implementation, if needed, convolutional encoders that lack feedback can be arbitrarily pipelined. They can be pipelined up to the speed of the delay through one flip-flop and one gate but this is probably not efficient due to too much latch overhead.

Encoders with feedback such as those in Figures 4-5 and 4-6, cannot be arbitrarily pipelined. They can be pipelined up to the point where the data is first used to compute the feedback. In this case it is the location of the first tap. Thus for the

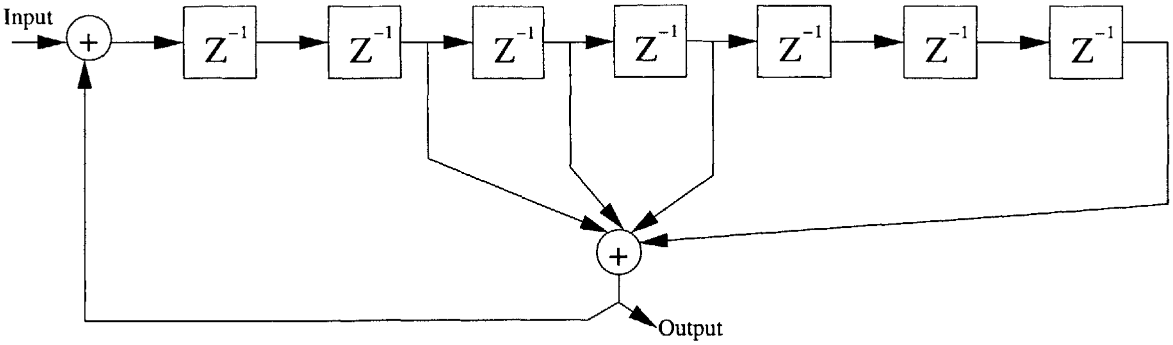

Figure 4-5: Convolutional Encoders with Feedback

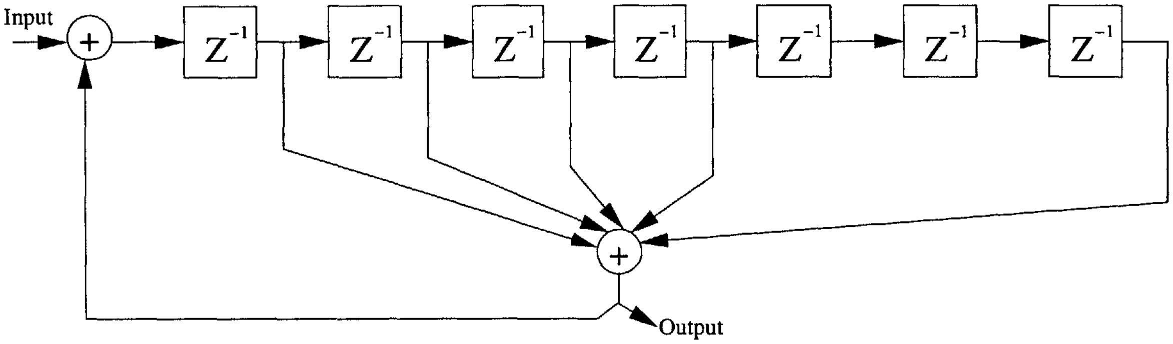

Figure 4-6: Convolutional Encoders with Tight Feedback

encoder shown in Figure 4-5 the circuit can be pipelined two deep, where the first tap occurs. But, the encoder in Figure 4-6 cannot be pipelined any more than the naive implementation provides for.

Not too surprisingly from the fact that these applications were designed to be implemented in hardware, the Verilog implementation of this encoder was the easiest and most straight forward to implement. While the implementation still contains a good number of lines of code, this is not an indication of the difficulty to express this design but rather is due to the inherent code bloat associated with Verilog module declarations.

Pentium

While this application is easily implemented in hardware, when it comes to a software implementation it is less clear what the best implementation is. Thus two implementations were made, one which is a naive '*reference*' implementation which calculates

```
void calc(unsigned int dataIn, unsigned int * shiftRegister,
    unsigned int * outputArray0, unsigned int * outputArray1)
{
    *outputArray0 = dataIn ^ (((*shiftRegister)>>4) & 1) ^
        (((*shiftRegister)>>3) & 1) ^ (((*shiftRegister)>>1) & 1)
        ^ ((*shiftRegister) & 1);
    *outputArray1 = dataIn ^ (((*shiftRegister)>>5) & 1) ^
        (((*shiftRegister)>>4) & 1) ^ (((*shiftRegister)>>3) & 1)
        ^ ((*shiftRegister) & 1);
    *shiftRegister = (dataIn << 5) | ((*shiftRegister) >> 1);
}
```

Figure 4-7: Inner-Loop for Pentium *reference* 802.11a Implementation

the output bits using logical XOR operations and a '*lookup table*' implementation which computes a table of 2^7 entries which contains all of possibilities of data stored in the shift register. Then the contents of the shift register are used as a index into the table. Both of these implementations are written in 'C'. Figure 4-7 shows the inner loop of the *reference* code.

As can be seen in the *reference* code, the input bit is passed into the `calc` function as `dataIn`, the two output bits are calculated into the locations `*outputArray0` and `*outputArray1`. And lastly the state variable `*shiftRegister` is shifted over and the new data bit is added for the next iteration of the loop. This inner loop is rather expensive especially considering how many shifts and XORs need to be carried out in the inner loop.

To make the inner loop significantly smaller, other methods were investigated to put as much as is possible out of the inner loop of this applications. Unfortunately, the operations that needed to be done were not simply synthesizable out of the operations available on a x86 architecture. Hence the fastest way to implement this convolutional encoder was to use a lookup table. The *lookup table* implementation of this encoder uses the same loop from the *reference* implementation to populate a lookup table with all of the possible combinations of shift register entries and then in its inner loop, the only work that needs to be done is indexing into the array and shifting of the shift register. Figure 4-8 shows the inner loop for the *lookup table* version of this encoder. Note that `lookupTable[]` contains both outputs as a performance optimization and

```
void calc(unsigned int dataIn, unsigned int * shiftRegister,
    unsigned int * outputArray0, unsigned int * outputArray1)
{
    unsigned int theValue;
    unsigned int theLookup;
    theValue = (*shiftRegister) | (dataIn<<6);
    theLookup = lookupTable[theValue];
    *shiftRegister = theValue >> 1;
    *outputArray0 = theLookup & 1;
    *outputArray1 = theLookup >> 1;
}
```

Figure 4-8: Inner-Loop for Pentium *lookup table* 802.11a Implementation

as such when assigning to the outputs some extra work must be done to demultiplex the outputs.

Raw

On the Raw tiled processor, three different versions of the encoder were made. They were all implemented in Raw assembly. Two of the implementations, *lookup table* and *POPCOUNT*, use one tile and the third implementation, *distributed*, uses the complete Raw chip, 16 tiles. One of the main differences between the Pentium versions of these codes and the Raw versions is the input/output mechanisms. On the Pentium, the codes all encode from memory (cache) to memory (cache), while Raw has a inherently streaming architecture. This allows the inputs to be streamed over the static network to their respective destination tiles and the encoded output can be streamed over the static network off the chip. The inputs and outputs all come from off chip devices in the Raw simulations.

The *lookup table* implementation for Raw is very similar to the Pentium version, with the exception that it is written in Raw assembly code, and it uses the static network as input and output. Figure 4-9 shows the inner loop. In the code, register \$8 contains the state of the shift register and `$csti` is the network input and `$csto` is the static network output. The loop has been unrolled twice and software pipelined to hide the memory latency of going to the cache.

```
loop:
    # read the word
    sll $9, $csti, 6
    or $8, $8, $9
    # do the lookup here
    sll $11, $8, 2
    lw $9, lookupTable($11)
    srl $8, $8, 1
    sll $10, $csti, 6
    or $8, $8, $10
    sll $12, $8, 2
    lw $10, lookupTable($12)
    andi $csto, $9, 1
    srl $csto, $9, 1
    srl $8, $8, 1
    andi $csto, $10, 1
    srl $csto, $10, 1
    j loop
```

Figure 4-9: Inner-Loop for Raw *lookup table* 802.11a Implementation

The Raw architecture has a single cycle POPCOUNT [2] instruction whose assembly mnemonic is popc. The lowest ordered bit of the result of popc is the parity of the input. This is convenient for this application because calculation of the outputs are a mask operation followed by a parity operation. The code of the inner loop of the *POPCOUNT* implementation can be seen in Figure 4-10. This implementation does not have a significantly different running time than the *lookup table* version, but it has no memory footprint.

The last and most interesting implementation for the Raw processor is the *distributed* version which uses all 16 tiles. This design exploits the inherent parallelism in the application and the Raw processor. In this design tile computes subsequent outputs. Thus if the tile nearest to the output is computing output x_n, the tiles further back in the chain are computing newer output values x_{n+1} to x_{n+6}.

This implementation contains two data paths and output streams which correspond to the two output values of the encoder. In this design all of the input data streams past all of the computational tiles. As the data streams by on the static

---

[2] POPCOUNT returns the number of ones in the input operand.

```
    # this is the mask for output bit 0
    li $11, (1<<6)|(1<<4)|(1<<3)|(1<<1)|(1)
    # this is the mask for output bit 1
    li $12, (1<<6)|(1<<5)|(1<<4)|(1<<3)|(1)
loop:
    # read the word
    sll $9, $csti, 6
    or $8, $8, $9
    and $9, $8, $11
    and $10, $8, $12
    popc $9, $9
    popc $10, $10
    andi $csto, $9, 1
    andi $csto, $10, 1
    srl $8, $8, 1
    j loop
```

Figure 4-10: Inner-Loop for Raw *POPCOUNT* 802.11a Implementation

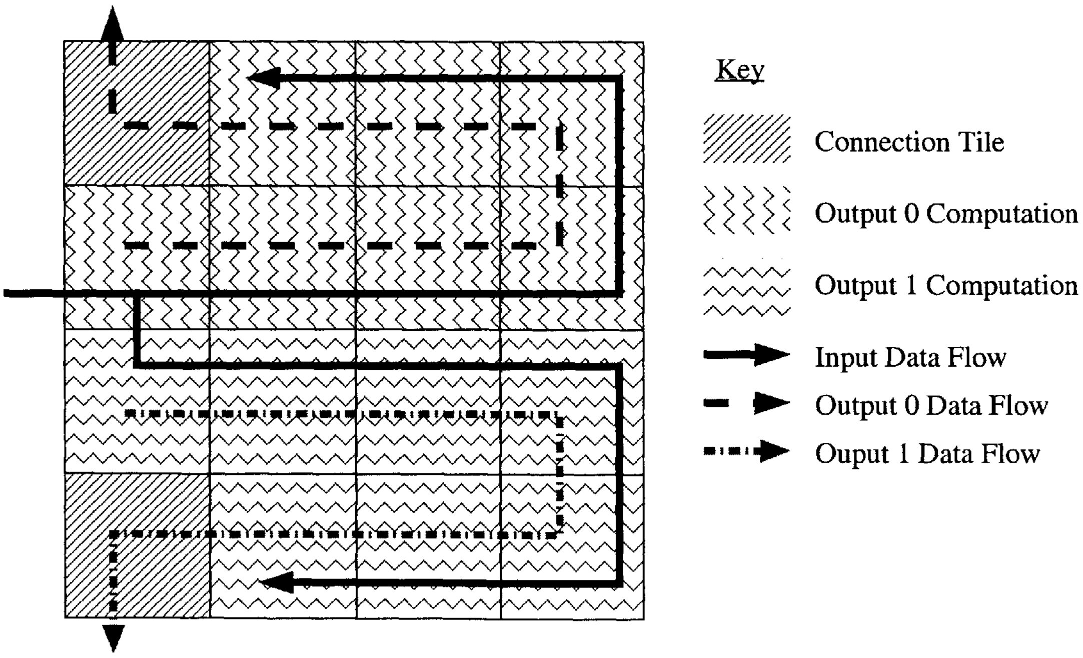
Key
Connection Tile
Output 0 Computation
Output 1 Computation
Input Data Flow
Output 0 Data Flow
Ouput 1 Data Flow

Figure 4-11: Mapping of the *distributed* 802.11a convolutional encoder on 16 Raw tiles

network, the respective compute tiles take in only the data that they need. Each tile only needs to take in five pieces of data because there are only five taps in this application, but they need to have all of the data flow past them because of the way that the data flows across the network. Once outputs are computed, they are injected onto the second static network for their trip to the output of the chip. The input and output paths can be seen in Figure 4-11. The connection tiles are needed to bring the output data off-chip because it was not possible to have the output data get onto the first static network without them. Lastly, all of the tiles have the same main processor code which consists of one move, four XORs, and a jump to get to the top of the loop. The static switch code determines which data gets tapped off to be operated on.

While this design might look scalable, it is not linearly scalable by simply making the chains longer. This is because if you make the chains longer, you will find that each tile has to let more data that it doesn't care to see pass by them. The current mapping does as good as a longer chain does because it is properly balanced. Every tile has to have 7 bits of data pass it, but it only cares to look at 5 of those input values, but because the application is 7 way parallel, it properly matched. If you make the chain longer, the useful work begin done will go from 5/7 to $5/n$ where n is the length of the chain, and hence no speedup is attained.

4.2 8b/10b Block Encoder

4.2.1 Background

The second characteristic application that this thesis will explore is IBM's 8b/10b block encoder. It is a byte oriented binary transmission code which translates 8 bits at a time into 10-bit codewords. This particular block encoder was designed by Widmer and Franszek and is described in [30] and patented in [9]. This encoding scheme has some nice features such as being DC balanced, detection of single bit errors, clock recovery, addition of control words and commas, and ease of implementation in hardware.

One may wonder why 8b/10b encoding is important. It is important because it is a widely used line encoder for both fiber-optic and wired applications. Most notably it is used to encode data right before it is transmitted in fiber optic Gigabit Ethernet [20] and 10 Gigabit Ethernet physical layers. Also, because it is used in such high speed applications, it is a performance critical application.

This 8b/10b encoder is a partitioned code, meaning it is made up of two smaller encoders, a 5b/6b and a 3b/4b encoder. This partitioning can be seen in Figure 4-12. To achieve the DC balanced nature of this code, the Disparity control box contains one bit of state which is the running disparity of the code. It is this state which lets the code change its output codewords such that the overall parity of the line is never more than ±3 bits, and the parity at sub-word boundaries is always either +1 or −1. The coder changes its output codewords by simply complementing the individual subwords in accordance with the COMPL6 and COMPL4 signals. All of the blocks in Figure 4-12 with the exception of the Disparity control simply contain feed forward combinational logic therefore this design is somewhat pipelinable. But due to the tight parity feedback calculation it is not inherently parallelizable otherwise. Tables are provided in [30] which show the functions implemented in these blocks, along with a more minimal combinational implementation.

4.2.2 Implementations

Verilog

Two implementations were made of this 8b/10b encoder in Verilog and they were shared between the IBM ASIC process and the Xilinx FPGA flows. One which we will call *non-pipelined* is a simplistic reference implementation following the proposed implementation in the paper. One thing to note about this implementation is that in the paper, two state elements were, and two-phased clocks were used while both of these were not needed. A good portion of the time designing this circuit was spent figuring out how to retime the circuit not to use multi-phase clocks. Unfortunately

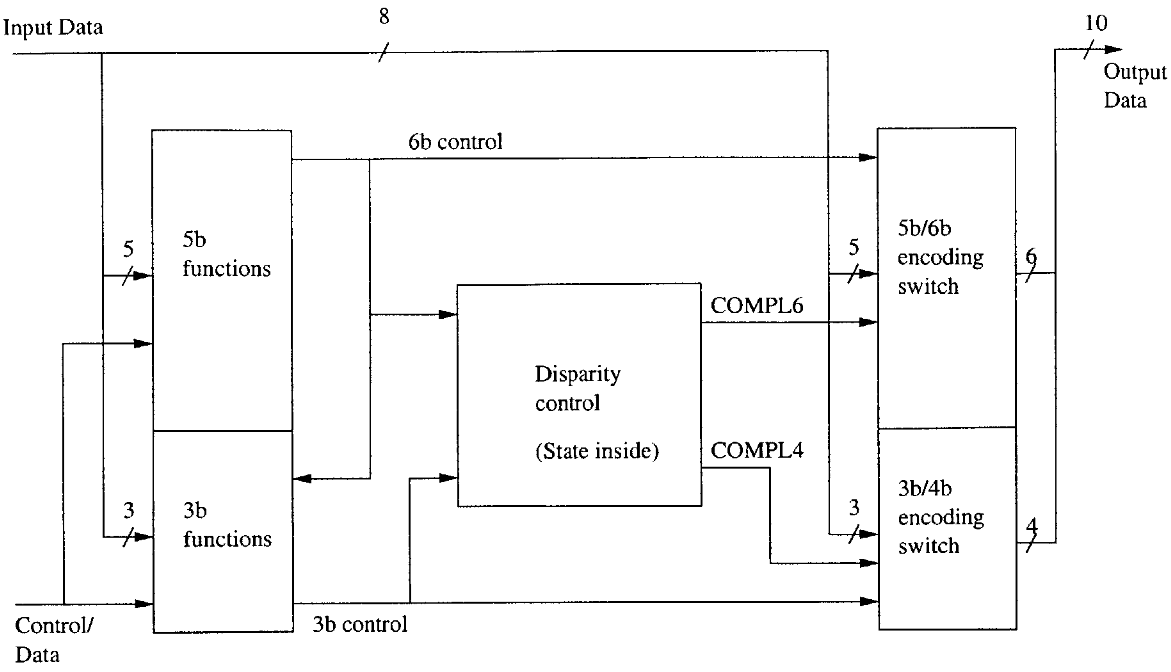
Input Data
8
10
Output Data
6b control
5
5b functions
5
5b/6b encoding switch
6
COMPL6
Disparity control
(State inside)
COMPL4
3
3b functions
3
3b/4b encoding switch
4
Control/ Data
3b control

Figure 4-12: Overview of the 8b/10b encoder taken from [30]

due to the retiming, a longer combinational path was introduced in the Disparity control box. This path exists because the parity calculation for the 3b/4b encoder is dependent on the parity calculation for the 5b/6b encoder.

To solve the timing problems of the *non-pipelined* design, a *pipelined* design was created. Figure 4-13 shows the locations of the added registers to the design. This design was conveniently able to reuse the Verilog code by simply adding the shown registers. The *pipelined* design was pipelined three deep. This is not the maximal pipelining depth, but rather this design can be pipelined significantly more, up to the point of the feedback in the parity calculation which cannot be pipelined.

Pentium

This application has only one feasible implementation on a Pentium. This implementation is as a *lookup table.* This was written in 'C' and the lookup table was created by hand from the tables presented in [30]. To increase performance, one large table was made thus providing a direct mapping from 9 bits (8 of data and one bit to denote control words) to 10 bits. The inner loop of the *lookup table* implementation can be

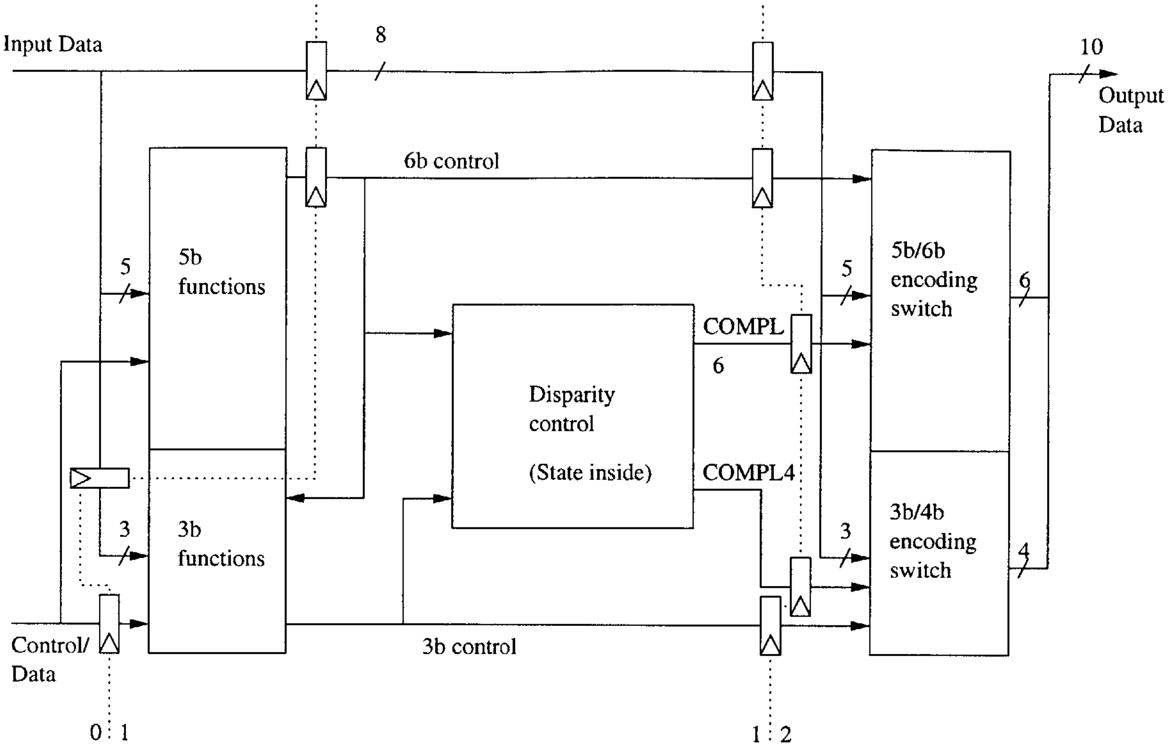
Input Data
8
10
Output Data
6b control
5
5b functions
5
5b/6b encoding switch
6
COMPL
6
Disparity control
(State inside)
COMPL4
3
3b functions
3
3b/4b encoding switch
4
Control/ Data
3b control
0 : 1
1 : 2

Figure 4-13: 8b/10b encoder *pipelined*

```
unsigned int bigTableCalc(unsigned int theWord, unsigned int k)
{
    unsigned int result;
    result = bigTable[(disparity0<<9)|(k<<8)|(theWord)];
    disparity0 = result >> 16;
    return (result&0x3ff);
}
```

Figure 4-14: Inner-Loop for Pentium *lookup table* 8b/10b Implementation

seen in Figure 4-14. The code constructs the index from the running disparity and the input word, does the lookup, and then saves away the running disparity for the next iteration.

Raw

On Raw, a *lookup table* implementation was made and benchmarked. This design was written in 'C' for the non-critical table setup, and Raw assembly for the critical portions. It is believed that a 'C' inner-loop implementation on Raw could do just as good, but currently it is easier to write performance critical Raw code that uses the

```
loop:
    # read the word
    or $9, $csti, $8        # make our index
    sll $10, $9, 2          # get the word address
    lw $9, bigTable($10)    # get the number
    or $0, $0, $0 # stall
    or $0, $0, $0 # stall
    andi $csto, $9, 0x3ff   # send lower ten bits out to net
    srl $8, $9, 7
    andi $8, $8, 0x200
    j loop
```

Figure 4-15: Inner-Loop for Raw *lookup table* 8b/10b Implementation

networks in assembly. Figure 4-15 shows the inner-loop of the lookup table. `bigTable` contains the lookup table. The two instructions marked with `stall` are there because there is an inherent load-use penalty on the Raw processor. Unfortunately due to the tight feedback via the parity bit, software pipelining is not able to remove these stalls.

Some thought was given to trying to make a pipelined design for Raw much in the same way that this application was pipelined in Verilog. Unfortunately, the speed at which the application would run would not change because the pipeline stage which contains the disparity control would have to do a read of the disparity, a table lookup, and save off of the the disparity variable which has an equivalent critical path to the *lookup table* design. It would save table space though by making the 512 entry table into three smaller tables, but this win is largely offset by the fact that it would require more tiles and the 512 word table already fits inside of the cache of one tile.

Chapter 5

Results and Analysis

5.1 Results

As discussed in Chapter 3, generating a fair comparison between so largely varied architectures is quite difficult. This thesis hopes to make a fair comparison with the tools available to the author. The area and frequency numbers in this section are normalized to a $0.15\mu m.$ L_{drawn} process. This was done by simply quadratically [1] scaling the area and linearly scaling the frequency used. No attempt was made to scale performance between technologies to account for voltage differences. This linear scaling is believed to be a relatively good approximation considering that all of the processes are all very similar $0.18\mu m.$ vs. $0.15\mu m.$ vs. $0.13\mu m.$

5.1.1 802.11a Convolutional Encoder

Table 5.1 shows the results of the convolutional encoder running on the differing targets. The area and performance columns are both measured metrics while the performance per area column is a derived metric. Figure 5-1 shows the absolute, normalized to $0.15\mu m.$, performance. This metric is tagged with the units of MHz. which is the rate at which this application produces one bit of output. The trailing letters on the Pentium identifiers denote which cache the encoding matrices fit in.

---

[1] This should be quadratically scaled because changing the feature size scales the area in both directions thus introducing a quadratic scaling term.

Target	Implementation	Area (mm^2) (Normalized to 0.15μm. L$_{drawn}$)	Performance (MHz.) (Normalized)	Performance per Area (MHz./mm^2.)
IBM SA-27E ASIC		0.0016670976	1250	749806
Xilinx Virtex II		0.77	364	472.7
Intel Pentium 4 Northwood 2.2GHz. Normalized to 1.907GHz.	*reference* L1	194.37	50.1713	0.2581
	reference L2	194.37	48.9	0.2515
	reference NC	194.37	46.5	0.2393
	lookup table L1	194.37	119.2	0.6131
	lookup table L2	194.37	95.3	0.4905
	lookup table NC	194.37	76.3	0.3924
Intel Pentium 3 Coppermine 993MHz. Normalized to 1191.6MHz.	*reference* L1	73.61	27.1	0.3678
	reference L2	73.61	24.312	0.3303
	reference NC	73.61	18.6	0.2530
	lookup table L1	73.61	74.5	0.1.0117
	lookup table L2	73.61	62.7	0.8519
	lookup table NC	73.61	25.2	0.3423
Raw 1 tile 300 MHz.	*lookup table*	16	31.57	1.973
	POPCOUNT	16	30	1.875
Raw 16 tiles 300 MHz.	*distributed*	256	300	1.172

Table 5.1: 802.11a Convolutional Encoder Results

"L1" represents that all of data fits in the L1 cache, "L2" the L2 cache, and "NC" represents no cache, or the data set size is larger than the cache size.

Figure 5-1 shows trends that one would expect. The ASIC implementation provides the highest performance at 1.25GHz. The FPGA is the second fastest at approximately 4 times slower. Of the microprocessors, the Pentium 4 shows the fastest non-parallel implementation. The Raw processor provides interesting results. It is significantly slower than the Pentium 4 using a single tile, which is to be expected considering that the Raw processor runs at 300MHz. versus 2GHz. and is only a single issue in-order processor. But by using Raw's parallel architecture a parallel mapping can be made which provides 10x the performance of one tile when using all 16 tiles. Note that this shows sub-linear scaling of this application. These performance numbers show for a real world application how much more adept an ASIC is than an FPGA and how much more adept a FPGA is than a processor at bit level

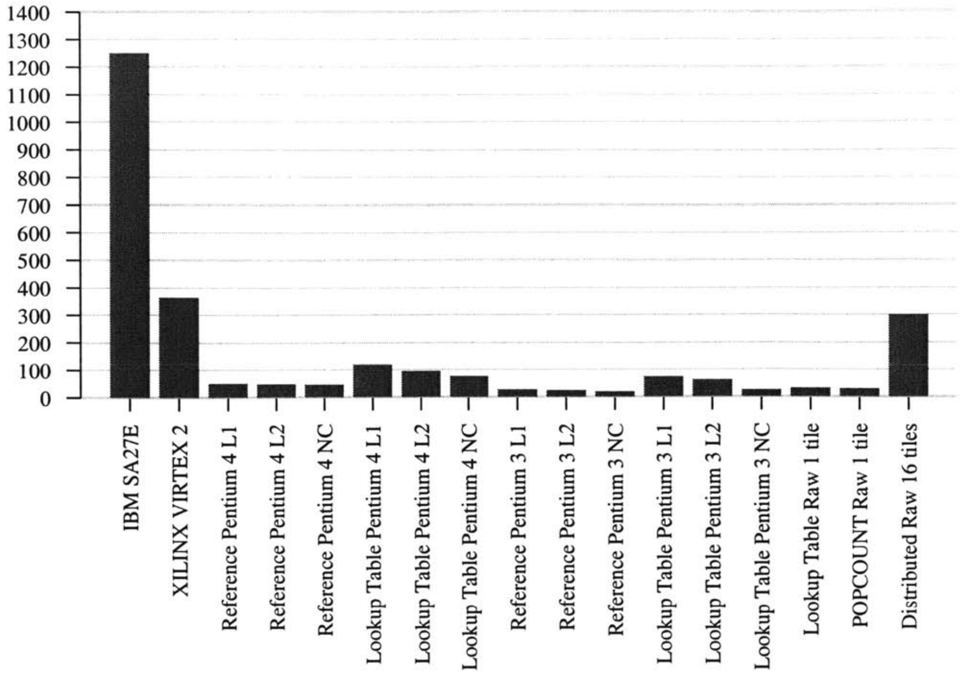

Figure 5-1: 802.11a Encoding Performance (MHz.)

computation.

Figure 5-2 shows a much more interesting metric. It plots the performance per area for all of the differing targets. This metric measures how efficiently each target uses the silicon area for convolutional encoding. One would want to use a metric like this if an application was fully parallel and you wanted to build the most efficient design for a given silicon area. Also, this metric gives an idea of how to compare architectures when it comes down to area comparison and helps answer the age old question of quantitatively how much better is a FPGA or ASIC compared to architecture "X". As can be seen from Figure 5-2, the ASIC is 5-6 orders of magnitude better than processor implementations on this bit-level application. FPGAs are 2-3 orders of magnitude more efficient than any of the tested processors. With respect to the processors, they are all within an order of magnitude of each other. It is interesting to see that the Pentium 4 is less efficient than Pentium 3 for performance per area while it does have better peak performance. This is because the area used by the Pentium 4 is proportionally more than the gained performance when compared to a

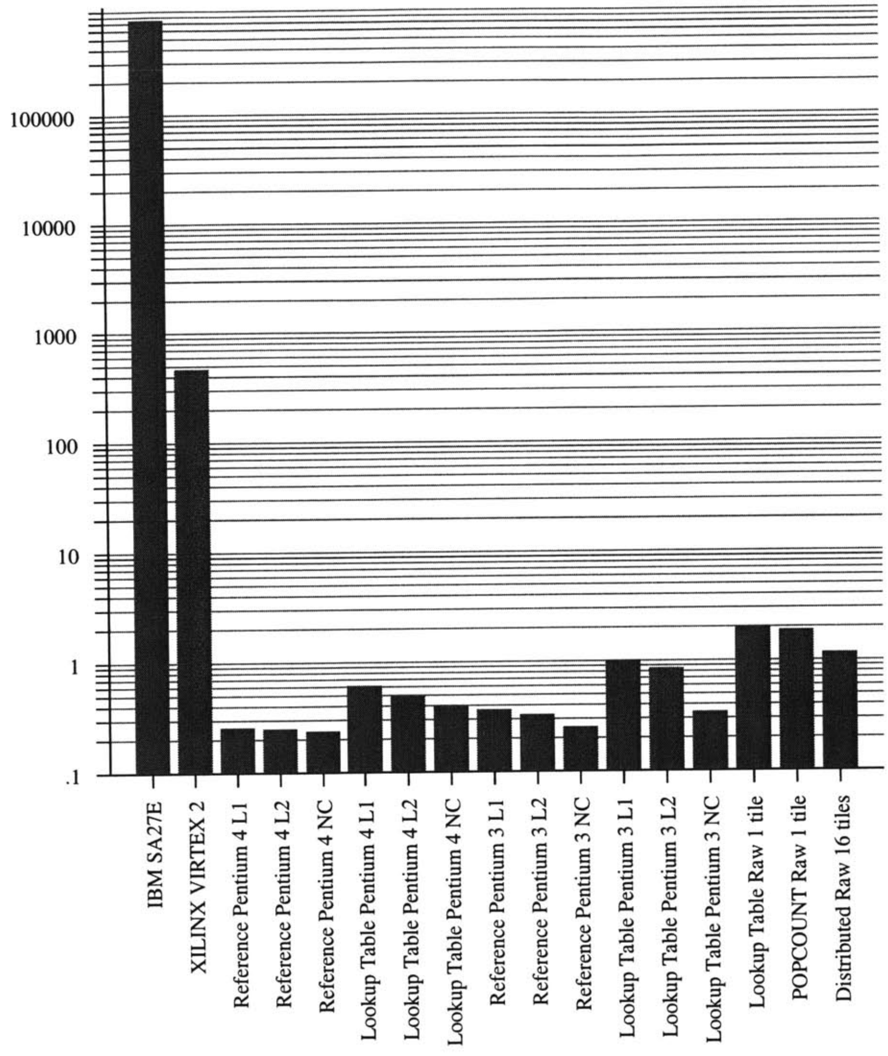

Figure 5-2: 802.11a Encoding Performance Per Area (MHz./mm^2.)

Pentium 3. Likewise, because of Raw's simpler data-path, its grain size more closely matches the application's grain size and thus it gets a smaller area punishment. Lastly, it is interesting to note that the parallelized 16-tile *distributed* Raw version has lower performance per area than the single tile Raw implementations. This is due to the sub-linear scaling of this application. While this implementation is able to realize 10x the performance of the single tile implementation, it uses 16x the area and thus if absolute performance is not a concern a single tile implementation is more efficient with respect to area.

5.1.2 8b/10b Block Encoder

The trends of this thesis's second characteristic application, 8b/10b encoding, are similar to the first application. Table 5.2 contains full results for all implementations. Figure 5-3 shows the absolute encoding performance. Note, that one cannot easily compare this to Figure 5-1 because the units are different. The performance metric used in Figure 5-3 is the rate at which 10-bit output blocks are produced, while Figure 5-1 is the rate at which bits are produced for a totally different application. As can be seen from the performance chart, the *pipelined* ASIC implementation is approximately 3x faster than the FGPA implementation, and the FPGA is 3x faster than a software implementation. As is to be expected due to clock speed, the Pentium 4 is faster than the Pentium 3 which is faster than a single Raw tile in absolute performance.

Figure 5-4 shows the performance per area for 8b/10b encoding. These results corroborate the results for the convolutional encoder. The ASIC provides 5 orders of magnitude better area efficiency than a microprocessor. Also, a FPGA has two orders of magnitude better area efficiency than a software implementation on a processor. Likewise the efficiency of the processors parallels the grain size of the differing architectures. One interesting thing that is not totally intuitive about Figure 5-4 is how pipelining this applications has differing effects on different targets. In performance per area, pipelining the ASIC implementation is worth it as can be seen from the first two bars. This means that the added performance outpaced the added area

Target	Implementation	Area (mm^2) (Normalized to 0.15μm. L$_{drawn}$)	Performance (MHz.) (Normalized)	Performance per Area (MHz./mm^2.)
IBM SA-27E ASIC	*non-pipelined*	.005117952	570	111372.67
	pipelined	.00756464032	860	113695.98
Xilinx Virtex II	*non-pipelined*	1.4706	159.109	108.1933
	pipelined	3.1514	272.554	86.4866
Intel Pentium 4 Northwood 2.2GHz. Normalized to 1.907GHz.	*lookup table* L1	194.37	111.8	0.5752
	lookup table L2	194.37	105.7	0.54340
	lookup table NC	194.37	111.8	0.5752
Intel Pentium 3 Coppermine 993MHz. Normalized to 1191.6MHz.	*lookup table* L1	73.61	62.7	0.8519
	lookup table L2	73.61	49.7	0.6745
	lookup table NC	73.61	28.4	0.3854
Raw 1 tile 300 MHz.	*lookup table*	16	27.273	1.7046

Table 5.2: 8b/10b Encoder Results

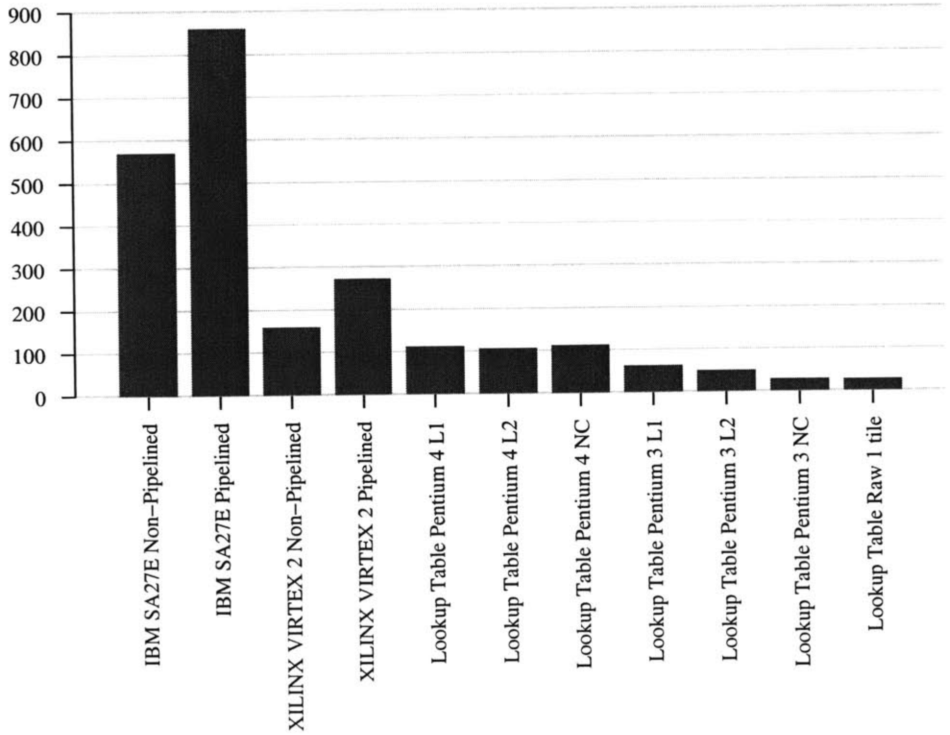

Figure 5-3: 8b/10b Encoding Performance (MHz.)

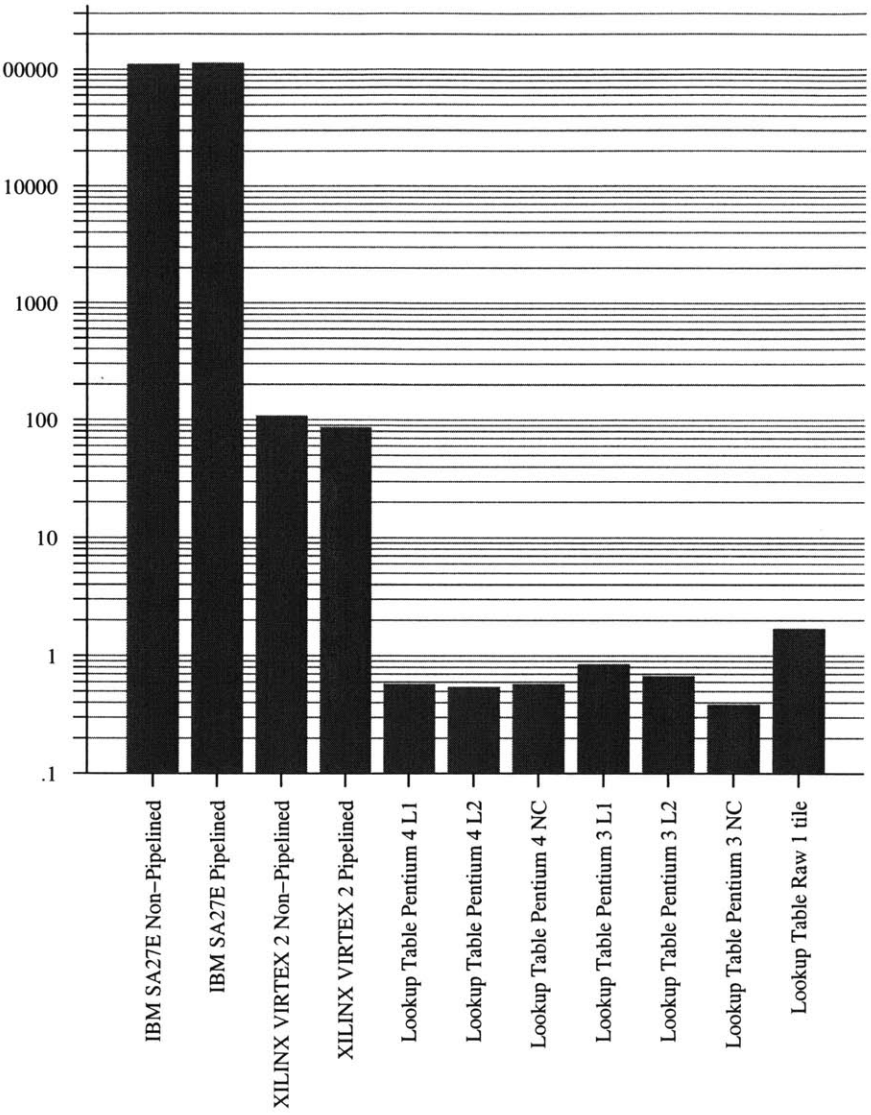

Figure 5-4: 8b/10b Encoding Performance Per Area (MHz./mm^2.)

of extra pipeline flip-flops. But the opposite story is true for the Xilinx Virtex II. While a 1.7x performance gain was achieved by pipelining this application, the area for this application was changed by a factor of 2.14. This result shows off the relative differences between flip-flop costs between these two targets. In an FPGA, because of the relative sparseness of flip-flops and the larger flip flops due to all of the added reconfiguration complexities added by an FPGA, the cost of pipelining an application is much higher than in an ASIC where the flip-flops cost less in both direct area, and they restrict the placement of the circuit far less than in a FPGA.

5.2 Analysis

When one looks as the results of this thesis, there are a couple of quantitative Rules of Thumb that present themselves with respect to *bit-level communications processing.*

1. ASICs provide a 2-3x absolute performance improvement over a FPGA implementation.
2. FPGAs provide a 2-3x absolute performance improvement over microprocessor implementation.
3. ASICs provide 5-6 orders of magnitude better performance per area than software implementation on a microprocessor.
4. FPGAs provide 2-3 orders of magnitude better performance per area than software implementation on a microprocessor.

These Rules of Thumb at first may be relatively surprising. One question that people may wonder about is why is the absolute performance of a FPGA compared to a microprocessor only 2-3 times? And why is the performance of an ASIC only 4-9 times as much as in software? Many people may think that the absolute performance of ASICs and FPGAs should be higher than that shown here. There are two reasons that the performance difference is not larger. One, when using a microprocessor as a lookup table it does a surprisingly good job of running bit level applications.

Second, this study uses simplistic implementations in hardware so as not to bloat the area of the designs. The hardware implementations could have used more complex implementations but this would have been at the cost of design complexity and silicon area.

Another question that these results inspire is why if this is simply a grain size problem are architectures that have a one-bit grain size more than 32 times as efficient when compared to an architecture with a 32-bit data-path? This can be reposed as asking why is using a 32-bit processor as a one-bit processor more than 32 times area inefficient? There are several factors at work here. One reason why smaller grain size applications can actually have super-linear performance speed up is that the relative cost of communication is cheaper. An example of this can be seen in the Raw processor which is a 32-bit processor. If this processor was shrunk to a 16-bit processor, we will assume roughly half the area, the distance that is needed to be traversed to communicate with the nearest other tile will not simply stay constant. Rather, the distance that is traversed will decrease to roughly 70% [2] of the 32-bit example's distance. Secondly, both ASIC and FPGA technology gain performance increases due to datapath specialization. Why build complicated general purpose structures when all you need is something small and specific? This is one of the typical arguments used in literature about why reconfigurable architectures are good. Data-path specialization helps increase clock rate by having the custom dataflow needed to match the computation and it also uses less area by simply not needing all of the complexity of a microprocessor such as instruction fetch, register renaming, reorder buffers, etc. One problem with the metric of performance per area metric that should be acknoledged, is the fact that it does not properly credit the ability of a microprocessor to time multiplex its hardware area. In a mircoprocessor, instructions are stored in a very dense instruction memory or possibly in and even denser main memory store, DRAM. It is via this time multiplexed use of the hardware area resources that the performance per area efficiency goes up. This is pretty difficult to quantize though becasue it requires total knowledge of all of the applications that are ever going to be run on the computational target. But, if all that is going to be run on a target, in a small period of time, is one application, the performance per area metric is a worthwhile metric.

---

[2] This can easily be calculated if you know that the area is decreasing by 1/2 this corresponds to scaling in each direction by $\sqrt{1/2} = 0.707$.

Finally, the most important thing that these results point to is the fact that computational grain size is very important. If an architecture only has one grain size, emulation of a differing grain size can in many cases be emulated relatively efficiently with respect to performance. An example of this is how a processor can creatively use a lookup tables to emulate random logic. But, when it comes down to efficient area utilization, proper grain size is critical.

Chapter 6

Architecture

This chapter proposes a new architecture, motivated by the results of this thesis, which is one possible solution of how a computer system can handle both *bit-level communication processing* and general purpose computation. This section is largely a proposal and as such can be thought of as a future work section as it brings forth many unanswered questions.

6.1 Architecture Overview

The main lesson that should be learned from this thesis is that supporting *bit-level communication processing* is an issue of efficient area utilization and not of absolute performance. While finer grain architectures can provide higher absolute performance due to data-path specialization and shorter distances to travel between parallel computational elements, the performance improvements are less than a factor of 10. This is in contrast to the area efficiency wins which can be more like 2-3 orders of magnitude conservatively over standard microprocessors or tiled architectures. While area efficiency may not be critical if area is free in the future, power can roughly be thought of as being proportional to area and it is unlikely that power will be free anytime soon especially with the advent of more and more devices being run off of batteries.

To solve the problem of achieving better area efficiency and maintain the ability to run general purpose computations, I propose the extension of the Raw tiled ar-

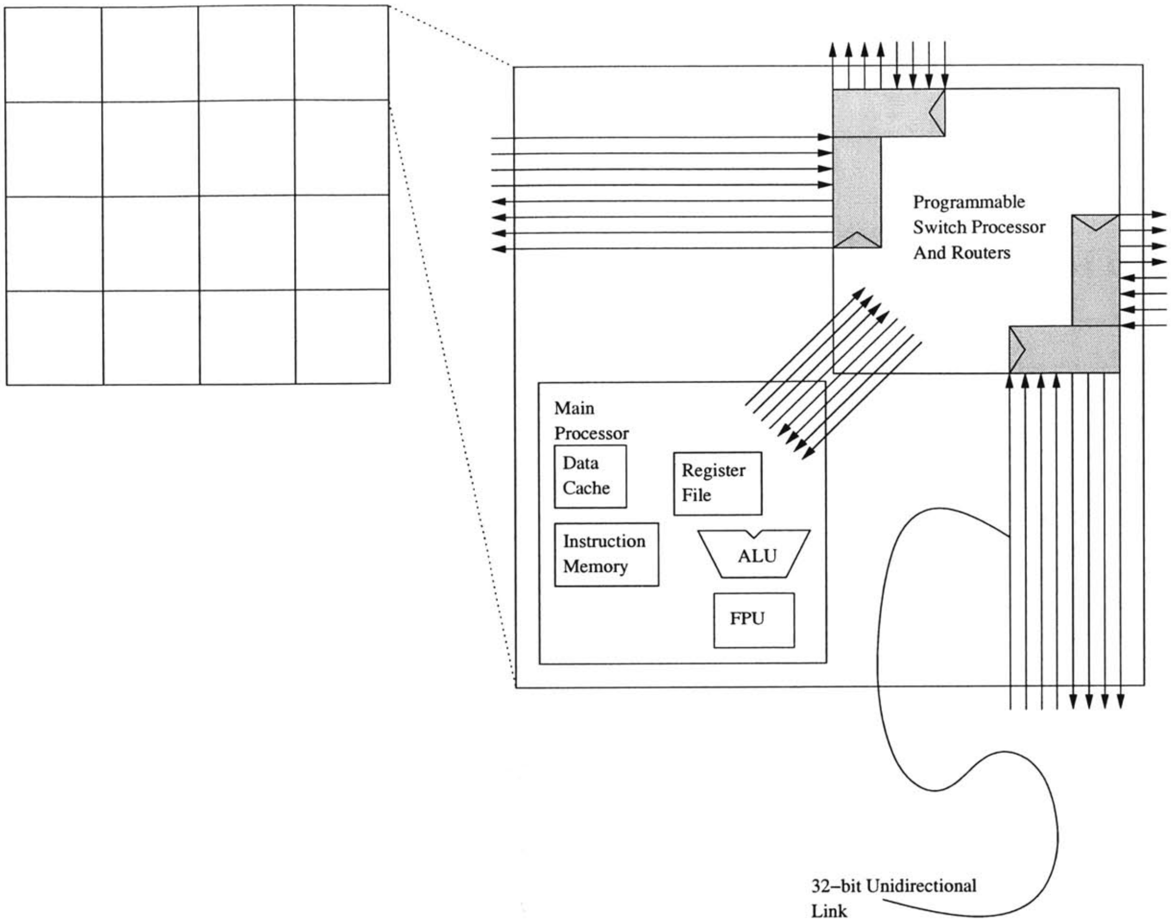
Programmable
Switch Processor
And Routers
Main
Processor
Data
Cache
Register
File
Instruction
Memory
ALU
FPU
32–bit Unidirectional
Link

Figure 6-1: The 16 Tile Raw Prototype Microprocessor with Enlargement of a Tile

chitecture. Raw is already able to effectively handle both scalar and parallel codes through extensive compiler support. But, as shown in the results section of this thesis, when it comes to efficiently using area for *bit-level communication processing*, it is not as efficient as finer grain computational fabrics. Thus I propose adding a fine grain computational fabric inside of each Raw tile.

Other approaches include heterogeneous tiled mixes or simply connecting a course grain chip next to a fine grain chip. The motivation for a homogeneous mix of tiles is that it is believed that to efficiently run *bit-level communication processing* together with other applications, a very close knit, low-latency, high bandwidth communications channel is needed. Also, homogeneity provides other niceties such as being a better compiler target and begin simpler to design.

To provide a reference frame about which all details are known, I propose using the Raw tile as it is embodied in the Raw prototype processor, with the same sizing of memories and same networks. Figure 6-1 shows the 16 tile Raw prototype processor with an enlargement of one tile. The main addition to Raw will be an array of yoctoengines [1] which are laid out in a mini-mesh inside of each tile. The contents of a yoctoengine and internal mesh communications are described in Section 6.2.

The yoctoengine array is a 2D mesh of yoctoengines. This is similar to the way that Raw tiles are tiled but on a much smaller grain. Therefore, one way to envision this architecture is as a tiled architecture with two levels of tiling, one for large Raw tiles, and inside of the Raw tiles there are yoctoengines tiled together. While I do propose a particular implementation of fine grain computation in a Raw tile, it is believed that any form of fine grain computational fabric, such as a FPGA, inside of a Raw tile would help increase the area efficiency of Raw on *bit-level communication processing.*

There are two problems that need to be solved when it comes to interfacing the yoctoengine array into a Raw tile. One, what is the interface, and two, if an application needs to use more than one tile's worth of yoctoengine array space, how does it efficiently use more? To interface the array with Raw's main processor, I decided that the use of register mapped communication provides the most efficient mechanism. This register mapped communication will take two general purpose registers from the main processor. Reading and writing to these registers read and write from two sets of network input blocks [2] (NIBs) which provide some buffer space between the yoctoengine array and the processor. The NIBs provide a speed gasket between the array and the main processor, because it is assumed that the array will run off of a faster clock than the main processor. The NIBs also provide flow control between the array and the main processor. If the processor reads from an empty NIB it will stall and if it writes to a full NIB it will stall until the NIB's blocking condition is cleared. On the array side, the whole array stalls if a write is about to occur to a NIB that is full, or if a read is about to occur and a NIB is empty. Completely stalling

---

[1] yocto is the SI prefix for 10^{-24}.

[2] A NIB is basically a FIFO with forward and backward flow control.

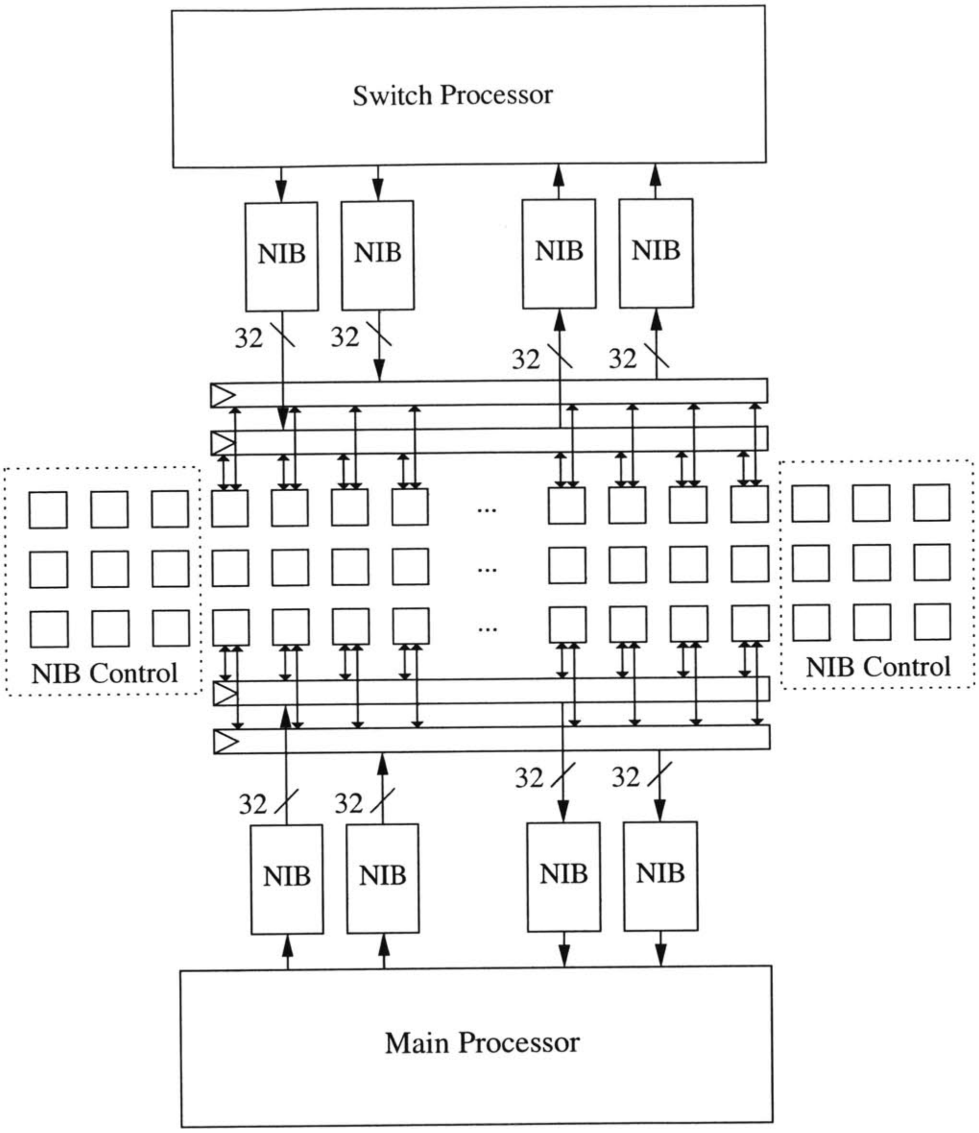
Switch Processor
NIB
NIB
NIB
NIB
32
32
32
32
...
...
...
NIB Control
NIB Control
32
32
32
32
NIB
NIB
NIB
NIB
Main Processor

Figure 6-2: Interfacing of the Yoctoengine Array to the Main Processor and Switch

the array is needed because inside of the array, there is no flow control because flow control would be expensive.

To be able to easily gang multiple of the yoctoengine arrays together, the array is also connected directly onto both static networks. Figure 6-2 shows this interface. This allows Virtual Wires [1] style communication between yoctoengine arrays. This addition requires an additional port on both of the static switch crossbars. The yoctoengine array can also be used as a filter on data being sent out the static network from the main processor by using a flow-through approach.

A preliminary sizing of the array calls for a 38 x 3 grid of yoctoengines. This allows for a 32-bit wide array to match data coming from 32-bit aligned sources. There are three additional columns on either side of the 32 x 3 array which can be used for control of the interface with the NIBs. Between the NIBs and the array, there are 32-bit wide registers that can be loaded from the NIBs or the array. These registers can also be shifted. The loading of these registers is connected to the control of the NIBs and the shifting of these registers is controlled by the NIB control yoctoengines.

6.2 Yoctoengines

The following description of a yoctoengine is a preliminary architecture and is open to change as more experience with fine grain computation is gained and as more application areas are mapped to it.

The yoctoengine array can either be viewed as an array of small Raw tiles or as an array of sequenceble FPGA LUTs with localized control. Each yoctoengine contains an instructions store, two two-input lookup tables, two eight entry stacks, a program counter, and a count register. Basically each yoctoengine is a sequenceble LUT similar to DPGA [4], but with differences including localized control, a stack for storage, and some per cycle reconfigurable interconnect like the Raw static network.

Yoctoengines are arranged in a 2D mesh. To communicate with each other, there are two switched point-to-point nearest neighbor networks that are reconfigurable on a per cycle basis. We will call these the cycle reconfigurable network (CRN). There is also an abundance of other routing resources which are similar to the routing resources on a FPGA. This network is based off of buses with switch points. The difference between this network and a FPGA's interconnect is that before this network enters into a yoctoengine, it has to pass through a flip-flop. This is done to force a fixed cycle time which is in contrast to a FPGA's variable cycle time. The FPGA style network (FSN) is not reconfigurable on a per cycle basis, but rather can only be reconfigured via an expensive operation. The use of these two networks provides the advantage of ease of routing that FPGAs enjoy while also allowing the

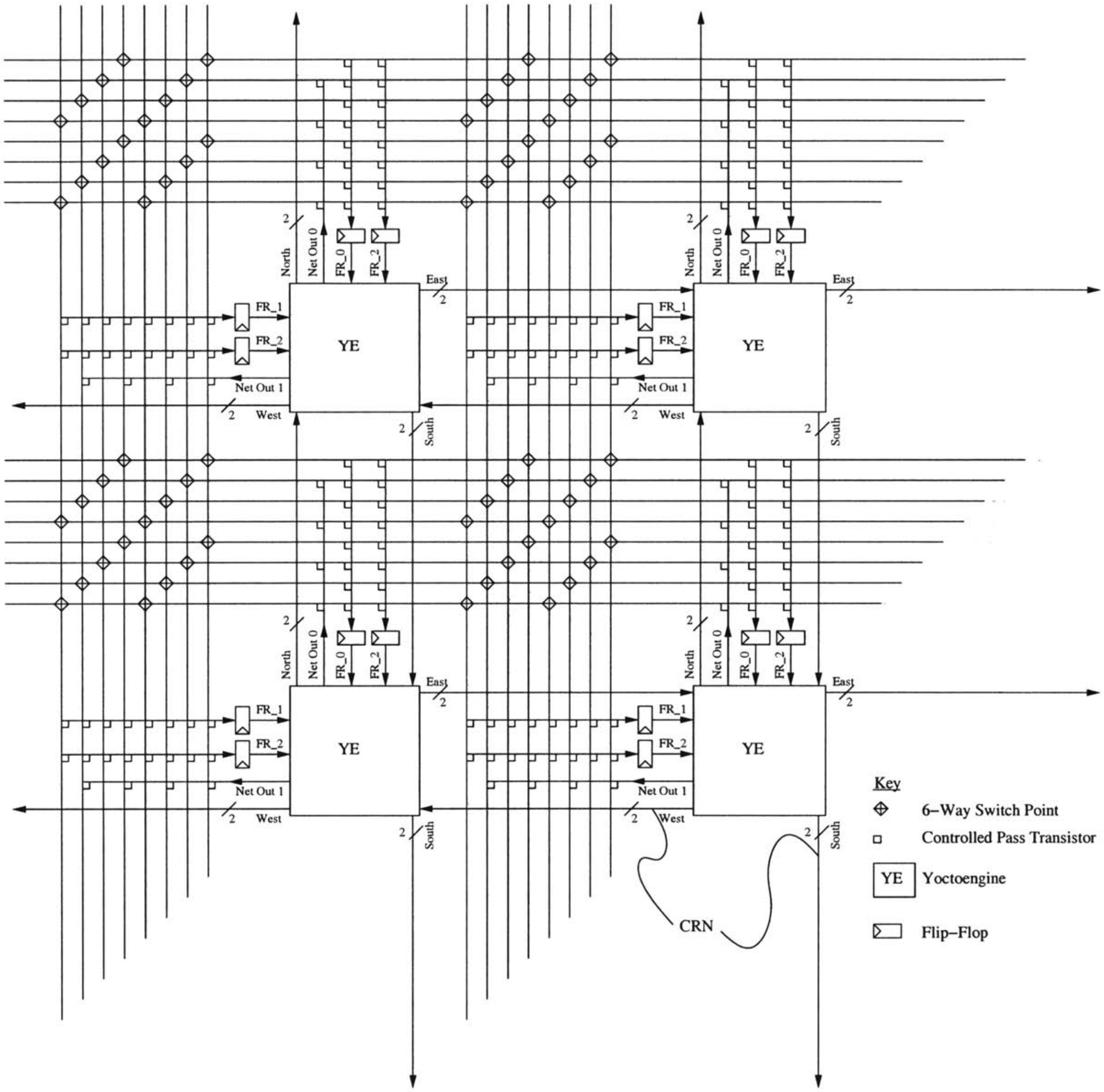
North
Net Out 0
FR_0
FR_2
East
2
FR_1
FR_2
YE
Net Out 1
West
South
CRN
Key
6–Way Switch Point
Controlled Pass Transistor
YE Yoctoengine
Flip–Flop

Figure 6-3: Four Yoctoengines with Wiring and Switch Matrices

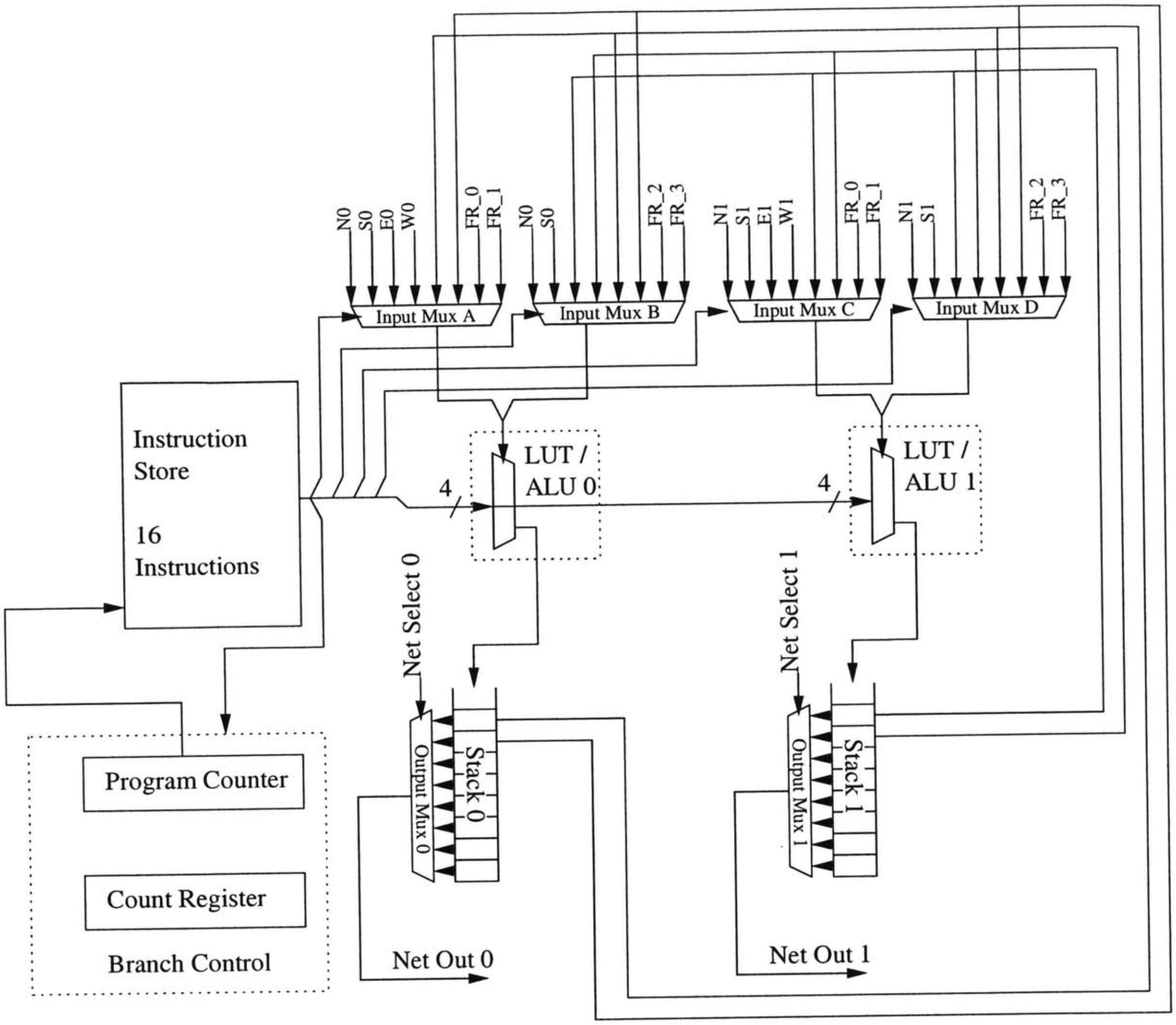
N0
S0
E0
W0
FR_0
FR_1
N0
S0
FR_2
FR_3
N1
S1
E1
W1
FR_0
FR_1
N1
S1
FR_2
FR_3
Input Mux A
Input Mux B
Input Mux C
Input Mux D
Instruction Store
16 Instructions
LUT / ALU 0
LUT / ALU 1
4
4
Net Select 0
Net Select 1
Output Mux 0
Output Mux 1
Stack 0
Stack 1
Program Counter
Count Register
Branch Control
Net Out 0
Net Out 1

Figure 6-4: The Internal Workings of a Yoctoengine

reconfigurability on a per cycle basis that Raw enjoys. Figure 6-3 shows the wiring inside of the yoctoengine array. The CRN can be seen as two-bit nearest neighbor communication labeled North, South, East, and West. The FSN routing is essentially the same as the distance-one interconnect found on Xilinx's 4000 series FPGAs [32]. One difference is that each switch matrix contains twice as many 6-way switch points to alleviate routing congestion. In Figure 6-3, each small square represents a flip-flop connected to a pass transistor which selectively connects the two associated wires. The diamonds represent 6-way switch points which are made out of 6 pass transistors and six controlling flip-flops which allow all possibilities of connection or isolation of the connected wires.

Figure 6-4 shows the internals of a yoctoengine. The heart of it is two muxes which act as the ALUs. They both compute the same function of 2-bits, which requires 4-bits of state stored in the instruction store. The select on the muxes come from the A, B, C, and D muxes. These muxes implement the switching portion of the CRN. Instead of simply choosing between the nearest neighbors, North, East, South, and West, the muxes can also choose fixed routes (FR) and the top two elements of the yoctoengine's stack. They can also choose from the top two elements of the other pipeline's stack. After the data flows through the ALUs/LUTs, the outputs feed into an 8 element stack. The stack can be pushed onto, popped from, held, or not shifted with only the top element being replaced. The stack controls are independent and each require 2 bits of state to control. The branch control provides for simple branching. Options include no branch, branch and decrement, branch if the top of Stack 0 == 0, branch if the top of Stack 1 == 0, and load the count register. The branch target or count load value is taken selectively from either the A/B Mux control bits, or the C/D Mux control bits. When branching, it is not advisable to be use the pipe that has its mux control being used as the branch target. Table 6.1 shows the breakdown of how bits are used in the yoctoengine's 24-bit instruction word. The instruction store hold 16 of these instructions. The muxes situated next to the stack allow the stack to serve a dual use as both a computational stack and as a fixed length FIFO which goes out to the network. The length of the FIFO is Net Select 0 and Net Select 1, which are reconfigurable when the whole yoctoengine array is reconfigured. This FIFO usage was inspired by the use of FIFOs for retiming in the HSRA project [28].

Reconfiguration of the yoctoengine array is an expensive operation that can take on the order of thousands of main processor cycles. To reconfigure, the main processor writes to a special purpose register which causes the data written to it to be shifted into the reconfigurable state. The reconfigurable state of the yoctoengine array is on 32 independent shift registers. This is similar to how the reconfigurable state on a FPGA is designed to be loaded as a large shift register. The reconfigurable state includes, Net Select 0, Net Select 1, the yoctoengines instruction store, and the

Function	Bit Width
A Mux select	3
B Mux select	3
C Mux select	3
D Mux select	3
LUT Function	4
Stack 0 Control	2
Stack 1 Control	2
Branch Control	4
Total	24

Table 6.1: Yoctoengine Instruction Coding Breakdown

control on the switch points of the FSN.

6.3 Evaluation

Now that we have gone through the gory details of an yoctoengine, lets take a step back and see just how it can be used. Trivially it can be seen that this architecture can be used as a FPGA. It has LUTs and a similar switch matrix as that found in a FPGA. The only real difference is the fact that before entering a yoctoengine, data coming from the FSN must go through a flip-flop so that the array can use a fixed clock cycle.

The added bonus of the yoctoengine array comes from the ability to do both temporal LUT reuse and temporal link reuse. The ability to temporally reuse LUTs is also achieved by DPGA, but the hybrid network of the yoctoengines takes this reuse one step further via link reuse much in the same way that Virtual Wires allows for time multiplexing of pins. This link reuse comes in two flavors. One, the CRN allows for generalized routing such as that which occurs on the Raw static network. The other flavor is used by having one yoctoengine connected to multiple other yoctoengines in a multicast manner via the FSN. This allows links to have different data destined for different destinations from one source simply by time multiplexing the multicast channel and having the destinations only listening at select times. This effectively lets links inside of a FPGA to be temporally reused and even allows them to be temporally reused to send data to different receivers. This link reuse also makes it such that the FSN needs less connectivity to effectively let arbitrary connections be made.

I currently do not have enough information to compare the yoctoengine array with a standard FPGA. It is thought that through the temporal link reuse, this architecture will provide a much more LUT dense environment with less area devoted to wiring than a typical FPGA. This added area efficiency comes at the cost of time multiplexing of the LUTs, which at first inspection seems to mean that it will take many more cycles than an FPGA to compute a given function, but rather all this does is cause pipelining where on a normal FPGA gates would be sitting unused for large portions of a clock cycle.

6.4 Future Work

One of the first things that needs to be done with any new architecture is to do a preliminary mapping of an application to it. Unfortunately for this thesis an actual mapping of the two sample applications has not been done. To be able to complete this a simulator is needed which I plan to write. Also a sensitivity study should be done to determine the exact grain size that is appropriate for the desired application mix. Virtualization support of the instruction stores would be convenient to be able to support larger programs on each yoctoengine.

Looking into the future, ultimately compiler support is needed to be able to take any arbitrary circuit and efficiently map it onto the described augmented Raw. I think that a possible programming model would include a mixture of 'C' interfaced with a synchronous sub-set of Verilog. I also believe that the described yoctoengines here or a slightly beefed up version could be used in the future to run streaming applications with incredible overall performance and performance per area improvements. The idea here is that the yoctoengines are a massively parallel 2D MIMD array. This would allow word oriented computations to occur on the bit-level yoctoengines, but very slowly. But this slowdown can be easily offset by parallelism that can be achieved by having many simpler processors which have higher clock rates. This argument is similar to those by the Connection Machine [14] and Terasys [10] with the main difference being that the array would be MIMD so that it can exploit the parallelism in streaming applications.

Power efficiency needs to be studied closer as I believe that this is the real motivation to use finer grain computational fabrics. Unfortunately, the tools to explore power make it difficult to make fair comparisons or to draw any conclusions as I learned in this thesis.

Lastly while this thesis focuses on *bit-level communication processing*, I would like to look into broader applicability of bit-level computation. Possible areas of improvement include efficient state machine implementation and more typical multimedia applications.

Chapter 7

Conclusion

This thesis has studied how different architectures are able to handle *bit-level communication processing.* To study this, two characteristic applications were selected and experimentally mapped onto common computational structures of differing grain size including microprocessors, tiled architectures, FPGAs, and ASICs. From these results it can be concluded that for these applications, fine grain computational fabrics (FPGAs) can provide a 2-3x absolute performance improvement over a best case microprocessor in the same fabrication process. And more importantly a fine grain computational fabric is able to provide 2-3 orders of magnitude better performance per area than software on a microprocessor.

From these results we conclude that it is usually possible to use one grain size to run an application with a smaller grain size with not too large of an absolute performance degradation through some emulation mechanism. In this case this emulation mechanism is the ability to use a microprocessor's cache as a lookup table to substitute for custom logic. But, unfortunately these forms of grain size mismatch cause large, multiple orders of magnitude, inefficiencies when it comes to area utilization. Thus we need to study integration of fine grain computational structures with architectures that have larger grain size such as word-based microprocessors to be able to support *bit-level communication processing* and general purpose computation in an area efficient manner. This thesis proposes one possible architecture to fill this gap based off of the Raw tiled microprocessor with the addition of a time multiplexed

LUT array. It is hoped that the results of this thesis will spur further development in this area and new architectures to meet the challenge of supporting both software circuits and general purpose computation on the same silicon.

Appendix A

Calculating the Area of a Xilinx Virtex II

In this thesis, I decided to use area as one of the main metrics for comparison. This seemed like a relatively easy thing to figure out. Most companies publish at least **some** area info about their chips, right? Wrong. I wanted to use a Xilinx Virtex II as one of architectures that I was mapping to but as much as I searched I couldn't find any area information. The information that I really wanted was how big a CLB/Slice was along with rough interconnection information. After not having any luck finding area information I started to get desperate. I posted to `comp.arch.fpga`, the Usenet group dealing with FPGAs asking if anybody had any area information about the Virtex II line from Xilinx. The responses I received all said that Xilinx was being very tight lipped about the area of their newest FPGAs and had not released any useful area information. I was down but not out.

I thought for a bit and then realized that I could just open up a Virtex II and figure out the information I needed. So my next step was to order some Virtex II's. I ordered four of the smallest Virtex II parts, XC2V40's[1] in the FG256 package.

---

[1] The part that I dissect is the Engineering Sample version of the chip because commercial parts for XC2V40's are not out yet. Presumably the area will be the same in the commercial version.

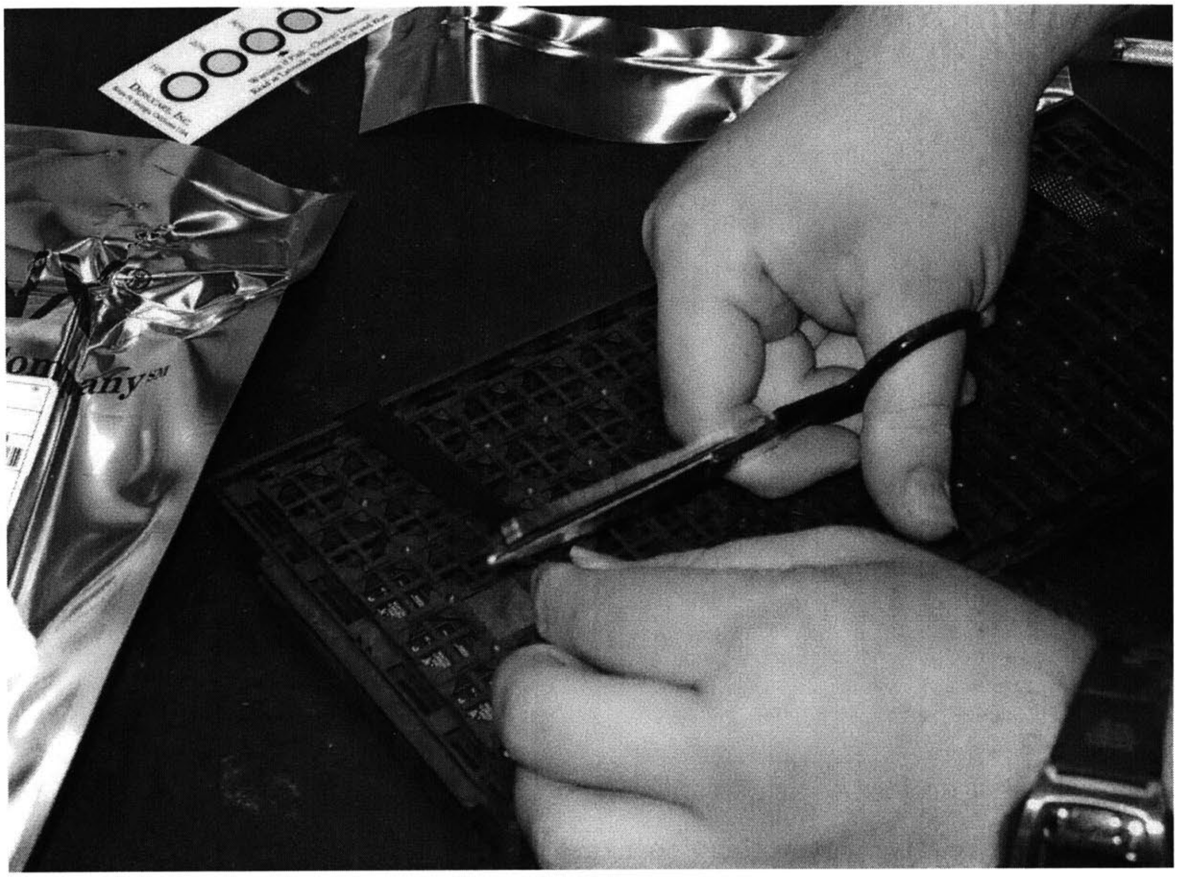

Figure A-1: Aren't Chip Carriers Fun

A.1 Cracking the Chip Open

I took a lot of pictures of the dissection process and thought I would share some of them with the reader. Figure A-1 shows me anxiously opening up the package that the chips came in. Figures A-2 and A-3 show the chips in their chip carriers.

Next we had to chop the chips open. Conveniently the FR4 [2] layer was easily pealed away from the epoxy/plastic top. A little bit of wedging of an Exacto knife in and some flexing of the FR4 allowed me to pop the two parts apart as can be seen in Figure A-4. At this point I was getting excited because I could see the actual die. If you look in Figure A-5, you can see a zoom in on the top half (the part with the lettering, not the solder balls) of the package, and in the middle there is a light green area with some scratched off area which is the actual die.

---

[2] FR4 is a fiberglass epoxy used as a dielectric in printed circuit boards.

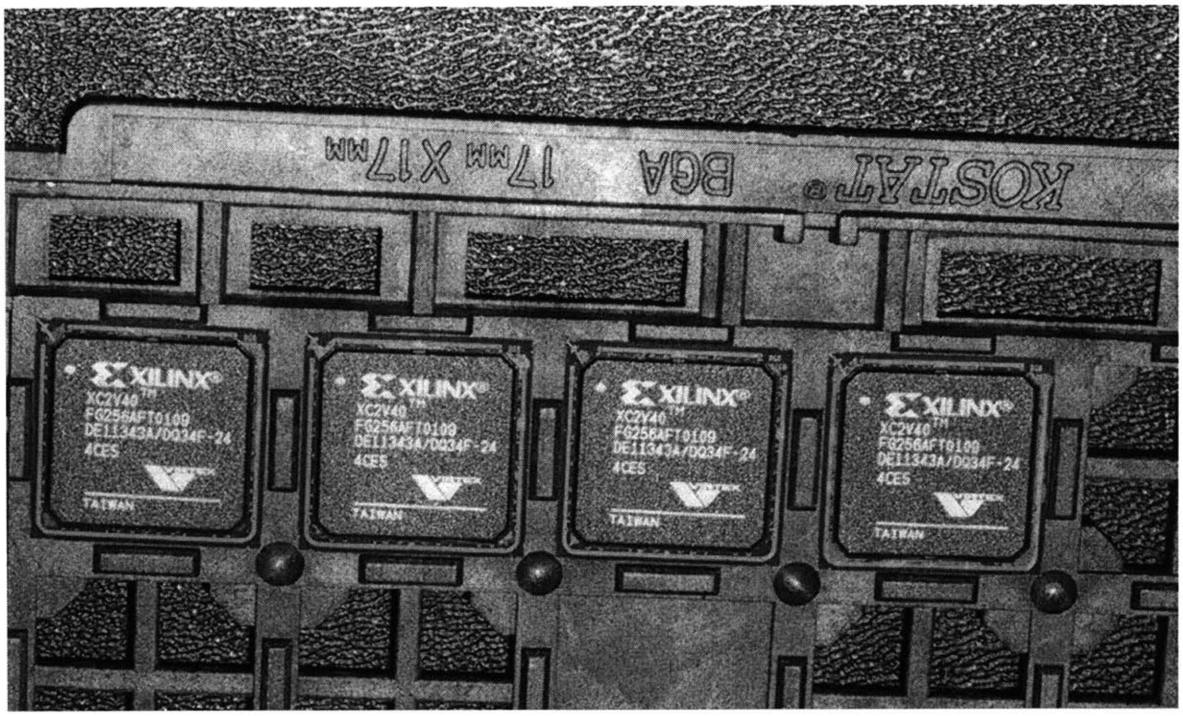
Figure A-2: Four Virtex II XC2V40 Chips

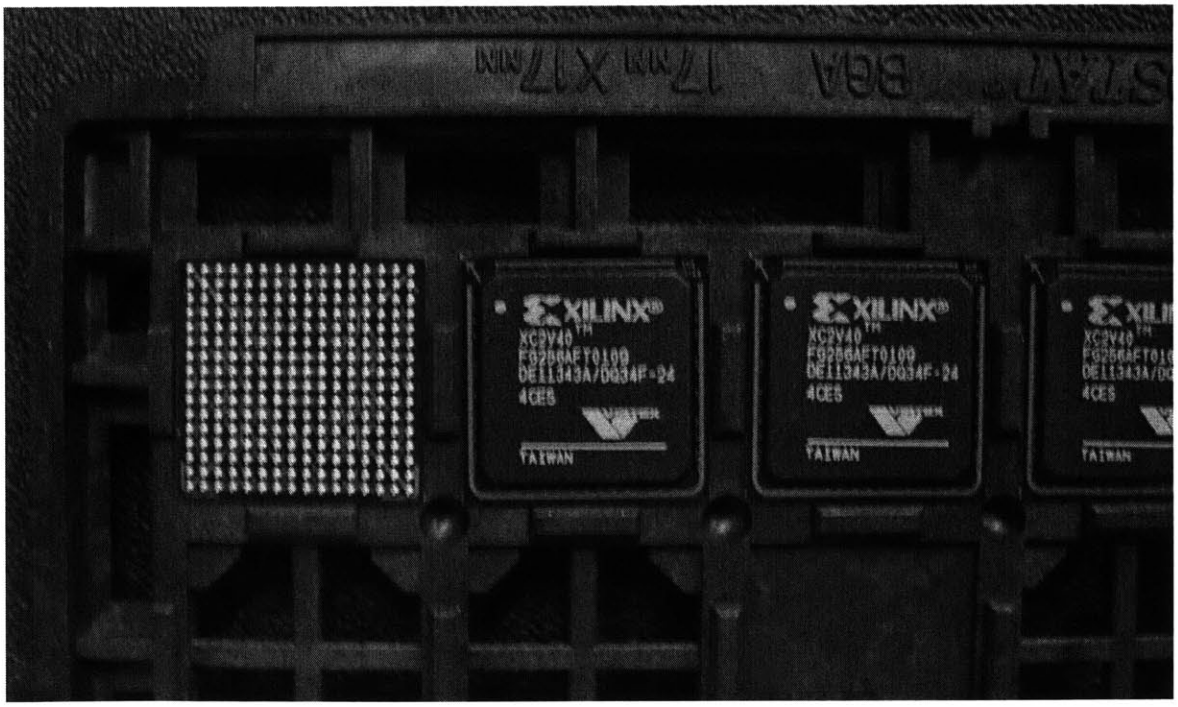
Figure A-3: Look at all Those Solder Bumps

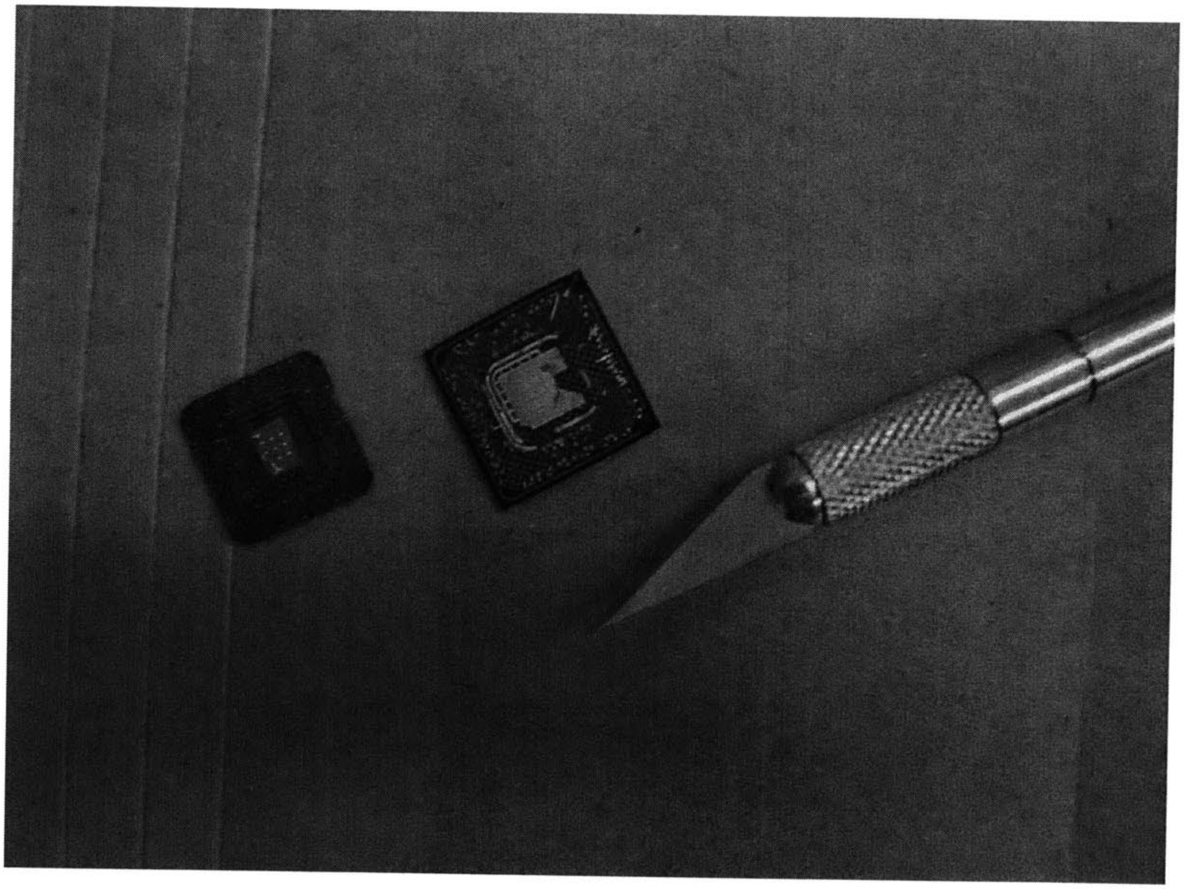

Figure A-4: The Two Parts of the Package

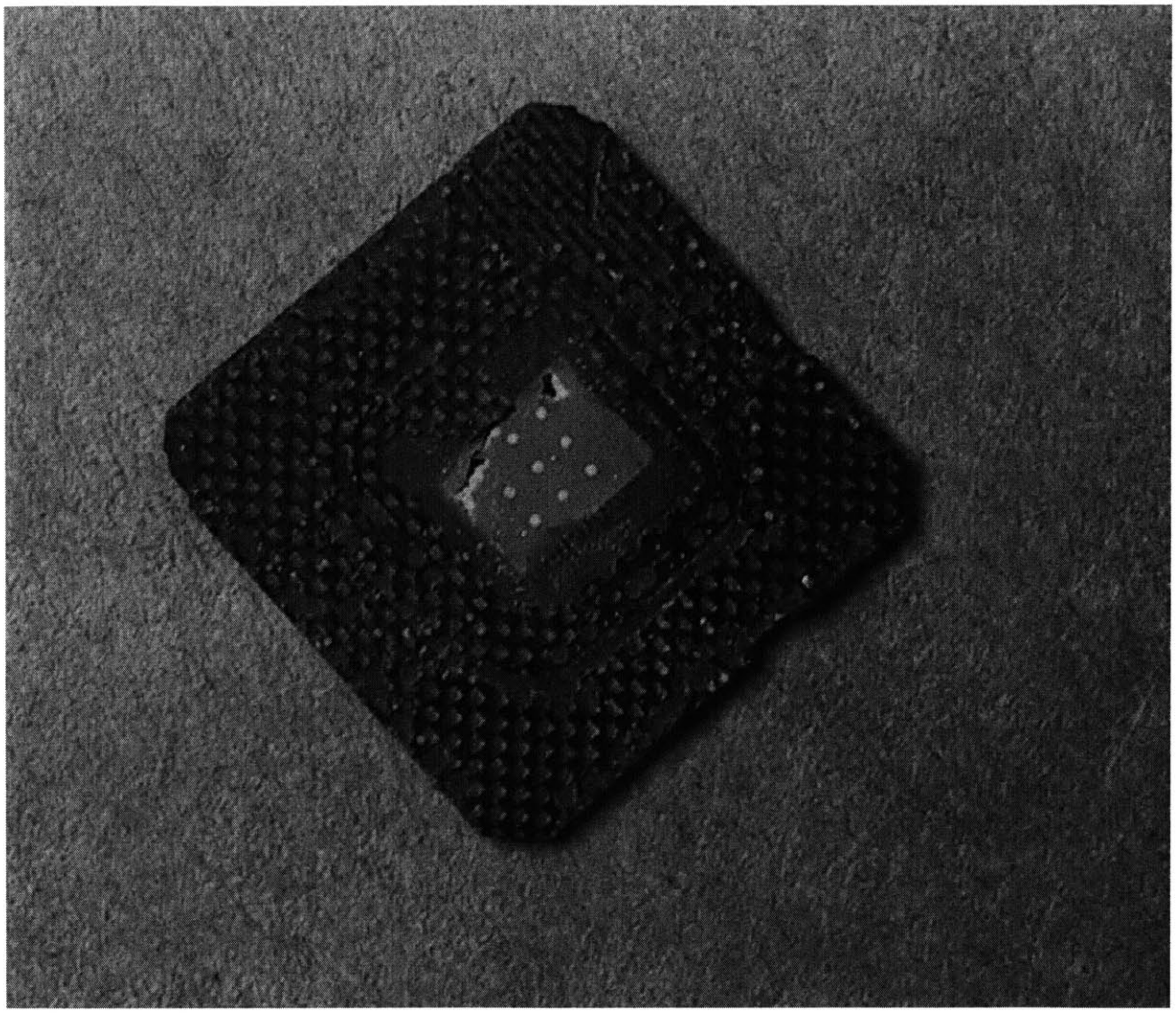

Figure A-5: Top Portion of the Package, the Light Green Area is the Back Side of the Die

Figure A-6: Removing the FR4 from the Back Side of the Die

I wanted to get a better look at the die so I took an Exacto knife and scrapped away the FR4 from the back side of the die. Unfortunately the top side of the die is on the other side and is embedded in epoxy. I know this for two reasons, one, as I scraped I didn't see any structure and contacts, and two this package is wire-bond, not flip chip and typically those have the die right side up in the package and have the wires come around from the package to the top of the die and package. This was later confirmed when I found a wire-bond wire going to the top of the die. Figure A-6 shows me removing the fiberglass from the back of the die.

A.2 Measuring the Die

Finally I got to the part that I was waiting for. I could clearly see the extents of the die. I pulled out my handy dandy set of calipers and measured the die size as shown

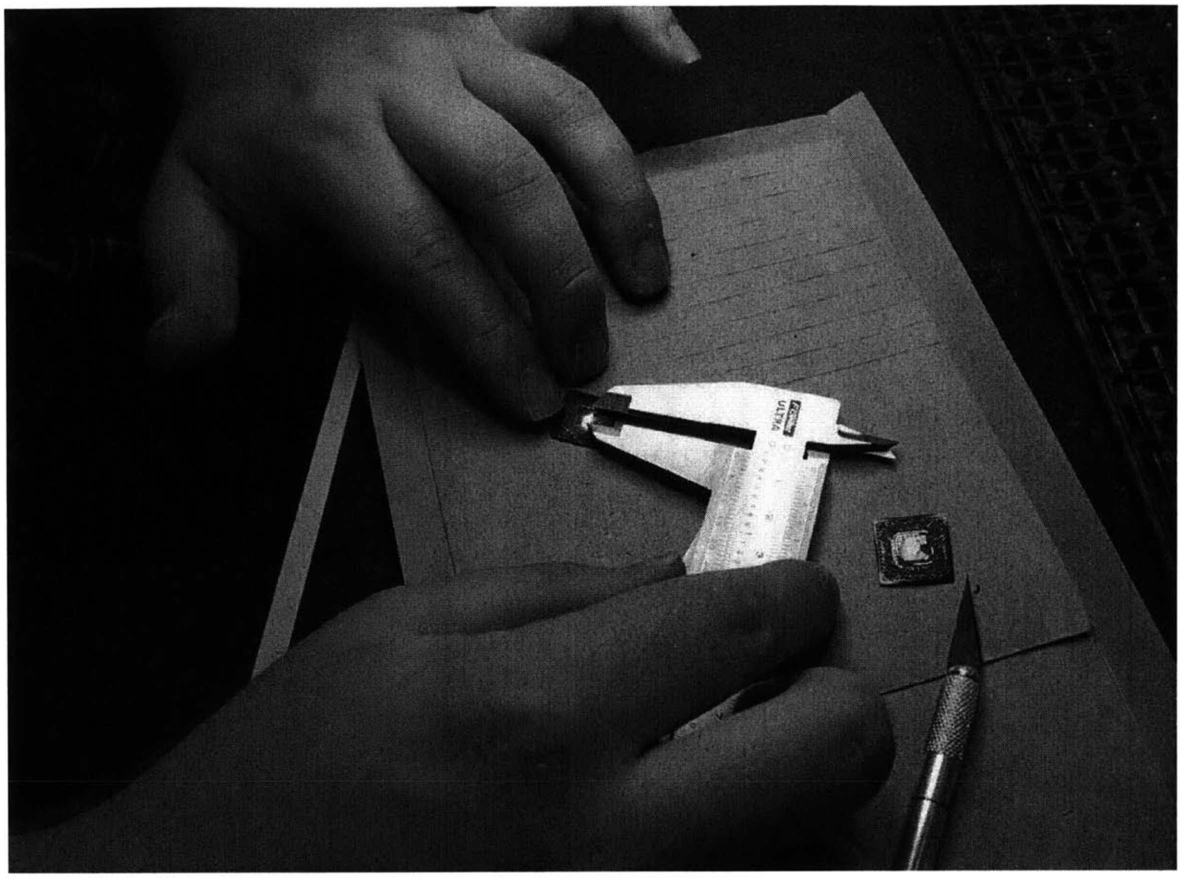

Figure A-7: Measuring the Die Size with Calipers

in Figure A-7. The die ended up being 4.98mm. x 3.60mm. Note that this was as accurate as I could get and it is possible that it is off by a bit, but I don't think it is off by very much. So now armed with total die area lets see if we can deduce the size of a CLB and the size of a Slice.

Taking a look at the Xilinx Virtex II Datasheet [34], we see that a XC2V40 contains an array of 8x8 CLBs, 4 multiplier blocks, 4 digital clock management units, and 72 Kbits of block RAM. This comes out to be 256 Slices. Because I was not able to find out how big the block RAMs, I/Os and multiplier blocks I will make a crude over-approximation and simply divide the overall die area by the number of Slices to figure out size of one Slice. Doing this, we come out with 0.07003125 mm^2. per Slice. This approximation is easily within a factor of two, but is definitely an overestimation which includes I/O and block RAM area unfortunately.

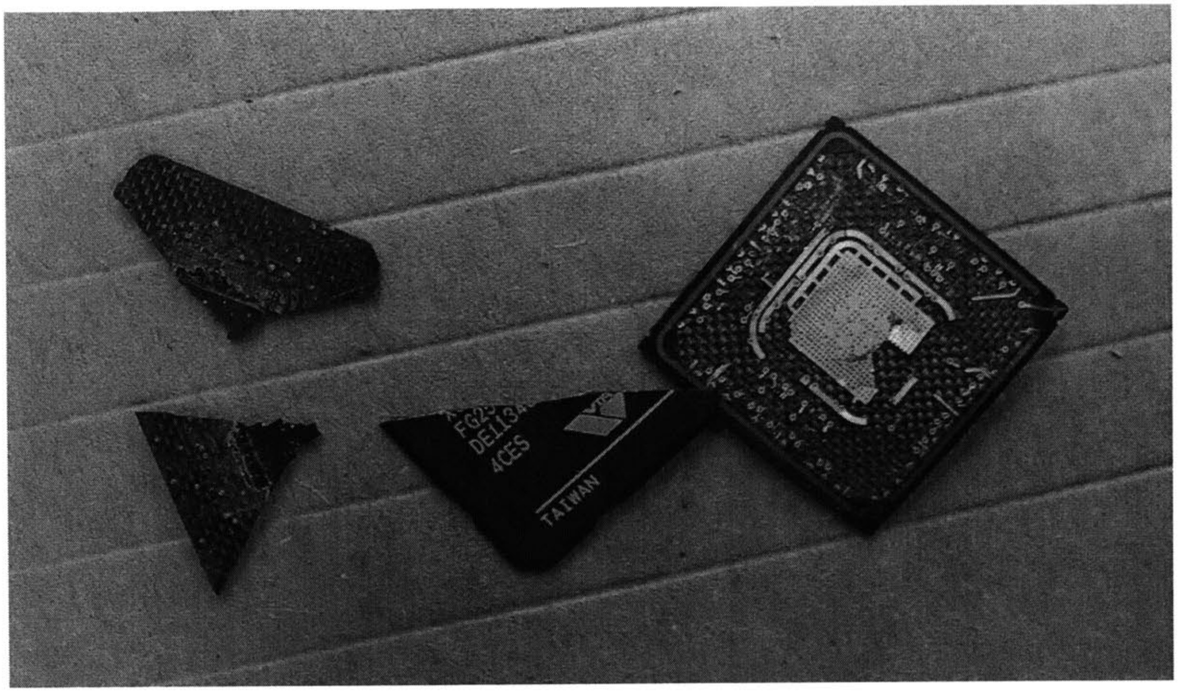

Figure A-8: Don't Flex the Die

A.3 Pressing My Luck

To solve the problem of overestimation of die size, I decided to try to see some large structure on the top of the chip and see if I could at least measure the CLB array. I first tried to cut out the die, but that wasn't going so well, so next I tried to peel the die out of its encased epoxy. Little did I know that this was an exceedingly poor idea. As soon as I flexed the die just a little bit, it cracked into two pieces. Figure A-8 shows what not to do. I still have three chips intact and hope to dissolve the epoxy lids off in my spare time so I can see some structure, but that is for another day.

Bibliography

[1] Jonathan William Babb. Virtual wires: Overcoming pin limitations in FPGA-based logic emulation. Master's thesis, Massachusetts Institute of Technology, November 1993.

[2] Jonathan William Babb. *High Level Compilation for Gate Reconfigurable Architectures*. PhD thesis, Massachusetts Institute of Technology, September 2001.

[3] Bluetooth Special Interest Group. *Specification of the Bluetooth System: Core*, 1.1 edition, February 2001.

[4] Michael Bolotski, André DeHon, and Thomas Knight. Unifying fpgas and simd arrays. In *Proceedings of the International Workshop on Field-Programmable Gate Arrays*, February 1994. MIT Transit Note Number 95.

[5] Jerome Burke, John McDonald, and Todd Austin. Architectural support for fast symmetric-key cryptography. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[6] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The garp architecture and c compiler. *IEEE Computer*, 33(4):62–69, April 2000.

[7] Timothy J. Callahan and John Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, San Jose, CA, 2000. ACM.

[8] Standard Performance Evaluation Corporation, 2002. http://www.spec.org/.

[9] Peter A. Franaszek and Albert X. Widmer. Byte oriented DC balanced (0,4) 8b/10b partitioned block transmission code. US Patent, December 1984. US Patent Number 4,486,739.

[10] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The Terasys massively parallel PIM array. *IEEE Computer*, 28(4):23–31, April 1995.

[11] Seth Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the International Symposium on Computer Architecture*, pages 28–39, 1999.

[12] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[13] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffery P. Kao. The Chimaera reconfigurable functional unit. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 87–96, April 1997.

[14] W. Daniel Hillis. *The Connection Machine*. PhD thesis, Massachusetts Institute of Technology, 1985.

[15] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Douglas Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, February 2001.

[16] IBM. *ASIC SA-27E Databook*, 2000.

[17] IEEE. *IEEE Standard 802.11-1997 Information Technology- telecommunications And Information exchange Between Systems-Local And Metropolitan Area Networks-specific Requirements-part 11: Wireless Lan Medium Access Control (MAC) And Physical Layer (PHY) Specifications*, November 1997.

[18] IEEE. *IEEE Standard 802.11a-1999 Supplement to IEEE standard for Information Technology- telecommunications And Information exchange Between Systems-Local And Metropolitan Area Networks-specific Requirements-part 11: Wireless Lan Medium Access Control (MAC) And Physical Layer (PHY) Specifications*, 1999.

[19] IEEE. *IEEE Standard 802.11b-1999 Supplement to IEEE standard for Information Technology- telecommunications And Information exchange Between Systems-Local And Metropolitan Area Networks-specific Requirements-part 11: Wireless Lan Medium Access Control (MAC) And Physical Layer (PHY) Specifications*, 1999.

[20] IEEE. *IEEE Standard 802.3-2000 IEEE standard for Information Technology- telecommunications And Information exchange Between Systems-Local And Metropolitan Area Networks-specific Requirements-part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, 2000.

[21] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–54, October 1998.

[22] Alex Peleg and Uri Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[23] Phillipe Piret. *Convolutional Codes.* MIT Press, 1988.

[24] John G. Proakis. *Digital Communications.* McGraw-Hill, fourth edition, 2001.

[25] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the International Symposium on Microarchitecture*, pages 172–80, November 1994.

[26] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose prgrams. *IEEE Micro*, pages 25–35, March 2002.

[27] David L. Tennenhouse and Vanu G. Bose. Spectrumware: A software-oriented approach to wireless signal processing. In *Proceedings of the International Conference on Mobile Computing and Networking*, pages 37–47, November 1995.

[28] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhanu, Varghese George, John Wawrzynek, and André DeHon. Hsra: High-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 125–134, February 1999.

[29] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, September 1997.

[30] Albert X. Widmer and Peter A. Franaszek. A DC-balanced, partitioned-block, 8b/10b transmission code. *IBM Journal of Research and Development*, 27(5):440–451, September 1983.

[31] Lisa Wu, Chris Weaver, and Todd Austin. CryptoManiac: A fast flexible architecture for secure communication. In *Proceedings of the International Symposium on Computer Architecture*, pages 110–119, July 2001.

[32] Xilinx Corporation. *XC4000E and XC4000X Series Field Programmable Gate Arrays Data Sheet*, May 1999.

[33] Xilinx Corporation. *Virtex-E Data Sheet*, November 2001.

[34] Xilinx Corporation. *Virtex-II Data Sheet*, November 2001.