Coarse-Grain Reconfigurable Computing

by

Ethan A. Mirsky

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

Master of Engineering in Computer Science and Electrical Engineering

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

Author .....
Department of Electrical Engineering and Computer Science
May 24, 1996

Certified by .....
Thomas F. Knight
Senior Research Scientist
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Coarse-Grain Reconfigurable Computing

by

Ethan A. Mirsky

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Computer Science and Electrical Engineering
and
Bachelor of Science in Computer Science and Engineering

Abstract

All general-purpose computing devices must allocate resources to handling the instructions which tell the devices how to behave. The ways in which these devices allocate their resources determines, to a large part, how efficiently a device will be able to perform a given application. All traditional general-purpose computing devices fix their resource-allocation decisions at fabrication time, making them efficient only on a limited set of applications. This thesis will introduce MATRIX, a novel, reconfigurable computing architecture which allows many of these resource allocation decisions to be made at program-time, allowing it to efficiently yield performance over a wide range of applications. This is made possible by a coarse-grain primitive block that is capable of serving as an instruction store, memory block, control unit, or a computing element, and a unified network capable of carrying both data and instruction information. A multi-level configuration scheme allows a user to deploy these primitive resources in an application-specific manner. A prototype device has been designed, and preliminary estimates indicate that its performance is comparable to modern high-performance computing devices, while maintaining a degree of architectural flexibility unavailable in any other conventional device.

Thesis Supervisor: Thomas F. Knight
Title: Senior Research Scientist

Acknowledgments

I would like to take this all too brief opportunity to give my thanks to those who have given me the greatest support in getting me where I am, and where I'm going.

First, and foremost, to my parents for their endless love and support, and for giving me a foundation from which anything is possible.

To my sister, Naomi, without whose warm friendship and understanding the world would surely be a dark and lonely place.

To all my friends: You opened up new horizons, showing me a world of wonder and beauty, and were always there for me. I will never forget you. Thanks especially to Matt, Burt, Marshal, Alan, Dan, Mike, Mary Beth, Erin and Rachel.

I also wish to give a big thank you to all those whose ideas and efforts have contributed to this work. I am especially grateful to:

Dr. Tom Knight, for giving me the opportunity of a lifetime as well as the advice and support I needed to accomplish it.

André DeHon, without whose brilliant insight and creativity these ideas would never have come to light, and without whose support and encouragement this project would never have even gotten off the ground.

Dan Hartman, an endlessly patient partner and friend, who was always ready to help me over any stumbling-block.

And to Ian Eslick, whose creativity and enthusiasm have and will be an inspiration for this project and beyond.

Thank You!

This research is supported by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

Contents

List of Figures

List of Tables

Chapter 1

Overview

General-purpose computing devices (GPCDs) have been widely used over the past few decades because of their re-usability, commodity applications, and post-fabrication adaptability. This adaptability is controlled by instructions, which are the commands used to tell the device how to behave. These instructions can take a variety of forms. On a microprocessor the instructions are the opcodes issued to the ALU on a cycle-by-cycle basis. On an FPGA, or other traditional reconfigurable computing device, the instruction is the configuration loaded at startup-time which sets the device's behavior for the entire run.

All general-purpose computing devices must address a number of important issues regarding their instructions. These include:

- Granularity
- Size of Instruction Memory
- Number of Instruction Streams
- Coupling of Instruction Streams
- Composition of Instruction Streams

The way in which a particular GPCD addresses these issues distinguishes its architecture from others, and can help classify it as one of the large classes of general-purpose architectures (microprocessors, SIMD, MIMD, VLIW, FPGA, etc). In addition to classification, these decisions play a large part in determining how efficient the device will be on a particular application. Chapter 2 will examine this issues and how they

effect a device's classification.

Modern general-purpose computing devices address these issues and fix their decisions at fabrication time. The consequence of this is that the device will perform well on applications whose needs it addresses, but poorly on those it does not. This thesis will introduce a device, MATRIX, that is capable of changing its choices on the issues listed above *after* fabrication, at program-time. This allows it to be efficient over a much wider range of applications than other GPCDs.

This post-fabrication architectural reconfigurability is made possible in MATRIX through the use of a higher-level configuration. This meta-configuration is used to specify the computing architecture on top of the MATRIX substrate, which can then be programmed as need to support a given application. Chapter 3 describes how this meta-configuration works.

MATRIX itself is composed of an array of 8-bit wide functional units, each of which contains memory, an ALU, and control logic. These blocks are connected through a reconfigurable network which can carry instruction information and data interchangeably. The switches on this network serve as the primary means of meta-configuring network. This basic architecture will be described in detail in Chapters 4, 5, and 6. The details here have been summarized from the more complete MATRIX Micro-Architecture Specification ([12]).

A prototype MATRIX device has be designed for a 0.5μm CMOS process. In this technology the basic array unit has footprint of 1.2mm$\times$1.5mm, and is estimated to run at 100MHz. At this size a MATRIX chip consisting of 10$\times$10 BFUs is easily feasible. Such a device would have peak performance of 10 billion (8-bit) operations per second. Chapter 7 gives more details of the current prototype implementation.

Unlike conventional architectures, MATRIX gives applications the opportunity to optimize the device architecture to best suit their needs. Chapter 8 will go through a detailed example of an application for a MATRIX device, in this case an FIR convolution. Different implementations will be created and compared with conventional devices and architectures.

Because MATRIX doesn't fix its instruction/control decisions at fabrication time,

it doesn't fit in a standard architecture taxonomy. In addition, it is capable of implementing almost any other architectural class. Chapter 9 will go through an architectural taxonomy and compare these conventional architectures with MATRIX implementations of those architectures.

Finally, Chapter 10 will conclude with an evaluation of the MATRIX effort and lessons learned so far and will look ahead to future work.

Appendix A contains working Verilog code for one of the core MATRIX units. It is the main part of a MATRIX simulation model.

Chapter 2

Resource Allocation in General-Purpose Computing Devices

2.1 General-Purpose Computing Devices

General-purpose computing devices (GPCDs) are components that can be programmed to perform any computational task. Although GPCDs typically have a lower performance when compared to application-specific IC (ASICs), they have a large number of advantages. These include:

- GPCDs are **reusable** for different applications. This means that a single piece of hardware can serve many different purposes in its lifetime.
- Because a GPCD can be used by many applications and application domains, the devices become **commodity items**, lowering costs and increasing availability.

The background material presented in this chapter has been summarized from André DeHon's soon to be released PhD thesis [5].

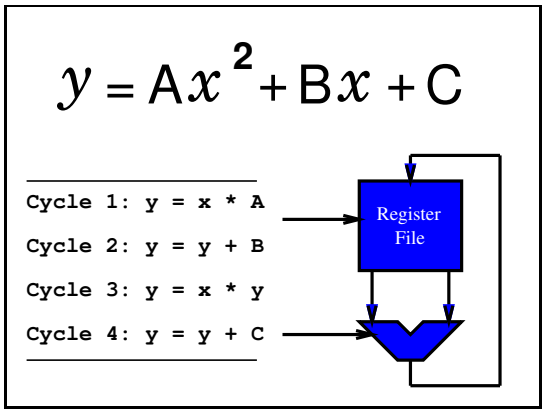

Figure 2-1: Temporal Computing Model

- Systems built with GPCDs are **post-fabrication adaptable**. This means that the algorithms and specifications used by the application can be changed and optimized late in the design process.

2.1.1 Temporal and Spatial Computing

Because it is impossible to provide a hard-wired unit for every possible operation, general-purpose computing devices *compose* complex computations from basic building blocks. Traditional GPCDs compose complex operation either temporally or spatially, although we will see that it is possible create hybrid devices.

Temporal Computing Devices (TCDs) rapidly reuse a single piece of circuitry for many different functions. In these devices, computations are assembled temporally from a usually predetermined set of basic operations. Intermediate data is stored in memory units until needed (Figure 2-1).

Typical temporal computing devices today are microprocessors which re-use their ALUs (Arithmetic-Logic Units) for different operations on every cycle. Modern microprocessors, including SIMD (Single-Instruction Multiple-Data), MIMD (Multiple-Instruction Multiple-Data), and VLIW (Very Long Instruction Word) devices utilize the larger silicon area provided by modern processing technologies to build larger ALUs and put several ALUs on a single chip. How-

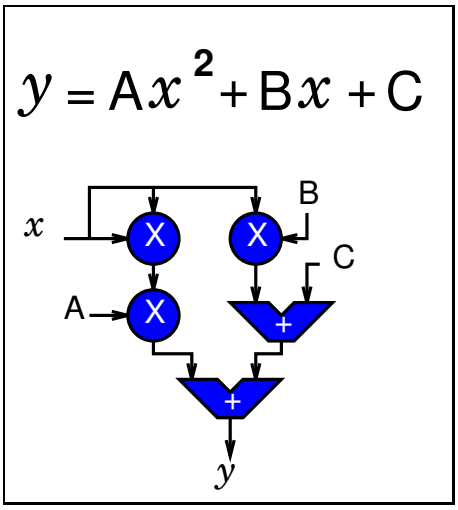

Figure 2-2: Spatial Computing Model

ever, they all still re-use these ALUs in time to compose operations.

Spatial Computing Devices (often referred to as configurable or reconfigurable computing devices (CCDs)) compose operations in space rather than time. These devices generally consist of an array (or other structure) of basic building blocks. In order to create a computation, each block is configured to perform one basic operation. The blocks are then wired together so that intermediate data is stored on wires between blocks rather than in memory units (Figure 2-2).

Typical configurable computing devices today are FPGAs (Field Programmable Gate Arrays) which generally consist of an array of one bit wide basic building blocks that can be configured to perform any logical operation on a small set of inputs. These one-bit blocks are connected through a configurable interconnect.

2.1.2 Instructions

Every GPCD requires a specification input which will tell it how to perform. We will refer to this specification as an **instruction**. The instruction can take a variety of forms. In a microprocessor, the instruction is the sequence of operations issued to the processing units on every cycle. In an FPGA, the instruction is the configuration

loaded into the basic blocks prior to the start of computing. Traditional GPCDs choose one or the other of these methods. As we will see it is possible to mix these styles, creating a hybrid device.

2.2 Design Issues for General-Purpose Computing Devices

When a designer sets out to design a general-purpose computing device, s/he must make a number of decisions, consciously or unconsciously, on how to allocate silicon area to handling instructions. All these issues are interdependent because silicon area resources must be allocated to implement the desired features and there is always a finite amount of area on a die. Improved manufacturing technologies have greatly increased this area, increasing the flexibility afforded to designers in making these choices.

2.2.1 Granularity

Granularity refers the data-width of the operations that can be independently specified by an instruction. In microprocessors this is the size of the datapath - typically 32 or 64 bits in modern microprocessors. In SIMD machines, this is the entire size of the machine because all processors perform the same instruction. In MIMD and VLIW machines, it is the width of each separate datapath. In FPGAs and other CCDs, the granularity is the size of the basic building blocks, typically 1 bit in modern FPGAs.

Coarser-grain datapaths generally simplify the instruction distribution because there are fewer units that need to see a given instruction. This is the reason that microprocessors and other TCDs use large datapaths - a simple instruction distribution is the only possible way to broadcast a new instruction on a cycle-by-cycle basis when the cycle time is very small.

On the other hand, coarse-grain devices are inefficient when working with small data values. A 64-bit datapath will likely be slower than an 8-bit datapath when

working with 8-bit data, and will certainly be much larger. Because many computations do not require large data-words, FPGAs and other CCDs use very fine-grain blocks. The price they pay is that they cannot rapidly change operations because the instruction distribution required would take a great deal of area and time.

2.2.2 Size of Instruction Memory

The size of the on-chip instruction memory determines the number of instructions that can be stored on-chip for rapid use. In microprocessors this is the size on the on-chip instruction cache. In FPGAs and other CCDs, this is the number of configurations that can be stored on-chip.

Large instruction memories are essential for temporal computing devices, because going off-chip for new instructions would greatly slow the rate at which instructions can be issued, and thereby reduce the device's overall performance. For this reason, the instructions on microprocessors and other TCDs tend to be small, selecting from a pre-determined set of operations. Small instructions also require less memory area, and therefore more can be stored on chip.

Because configurable computing devices require an instruction memory for every basic building block, CCDs cannot put many instructions on-chip without using an excessive amount of die area to do so. For this reason modern FPGAs store only one configuration on-chip. As a result, FPGA are not efficient for performing dynamically changing computations - new operations require a long time to configure.

Its important to note that this limit of one in FPGAs is not inherent to CCDs. Experimental devices, such as [20], have put more than one configuration on a CCD, allowing a limited amount of cycle-by-cycle flexibility.

2.2.3 Number of Instruction Streams

The number of instruction streams on a general-purpose computing device refers to the number of operations that can be performed in parallel. Traditional microprocessors have only one instruction stream. SIMD machines also use a single instruction

stream controlling multiple ALUs. MIMD and VLIW machines can have several instruction streams running in parallel. On FPGAs and other CCDs, the number of instruction streams is the same as the number of basic building blocks because each can be programmed differently.

The greater the number of instruction streams, the more parallism the device can exploit, which often means higher performance. On the other hand, each instruction stream requires its own separate memory to store instructions. MIMD and VLIW machines require a separate memory for each ALU, while SIMD machines and traditional microprocessors require only one per chip, and can therefore use larger memories, or put more ALUs on the die. On FPGAs every basic block requires its own memory.

The fact that FPGAs do not share instruction memories between blocks the way SIMD machines do is not fundamental to all CCDs. The MATRIX device described in this thesis is a CCD which can share instruction memories between blocks. However, this is feasible only a coarser granularity than the one-bit blocks used in FPGAs.

2.2.4 Coupling of Instruction Streams

While the number of instruction streams refers to the ratio between the number of instruction memories and ALUs, the coupling of instructions streams refers to the ratio between the number of control units and instruction streams. The best example of this is difference between VLIW and MIMD machines. Both use several different ALUs, each running a separate instruction stream. However, on MIMD machines, each stream is controlled independently so that branches performed on one stream do not necessarily happen on others. On VLIW machines, however, there is only one control unit so that a branch taken on one stream happens on all streams.

Traditional microprocessors and SIMD machines have only one instruction stream and therefore only one control unit. FPGAs typically have *no* control units, because they store only one instruction (configuration) on chip.

2.2.5 Composition of Instruction Streams

Finally, the composition of instruction streams refers to the nature of the instructions in a stream. The more powerful the instruction (the more operations a single instruction can specify), fewer instructions will be needed to complete a computation on a TCD. However, the more powerful the instruction, the larger it is and the more area resources need to be dedicated to distribute and control them.

On microprocessors and other TCDs, the instructions typically select from a set of operations which were fixed at fabrication time. This is generally done to keep the instructions small and easily distributed, as discussed in Section 2.2.2.

FPGAs and other traditional CCDs can be seen as the extreme case of powerful instructions. The CCD configuration is capable of expressing any computation (to the limits of the die area), but is so large, it is extremely difficult to distribute and control.

2.2.6 Architecture Taxonomy

Table 2.1 [1] summarizes the architecture descions made by conventional computing devices. Because conventional devices fix their choices of n, w, m, c at fabrication time, they all can be classified on this table.

2.3 Consequences of Resource Allocation

All general-purpose computing devices must deal with all these issues. However, the performance of applications on a particular GPCD depend greatly on the particular resource allocation choices the GPCD designer made. The reason for this is that every application requires a certain amount of control, has a certain amount of inherent parallism, and has a certain data-size, which will be very different from other applications. Thus different applications require different amounts of the architectural resources discussed above. The closer the match between the application's require-

[1] This table was taken from [5].

Control Threads (PCs)	Instruction Streams per Control Thread	Instruction Memory per Stream	Datapath Granularity	Architecture/Examples
0	0	0	n/a	Hardwired Functional Unit
	n	1	1	FPGA, Programmable Cellular Automata
			w	reconfigurable ALUs Programmable Systolic Datapath Arrays
1	1	c	$n \cdot 1$	bitwise SIMD
			w	Traditional Processors
			$n \cdot w$	Vector Processors
	n	c	1	DPGA [20]
		8	16	PADDI [3]
		c	w	VLIW
m	1	c	$\frac{n}{m} \cdot w$	MSIMD
n	1	c	1	VEGA [11]
		8	16	PADDI-2 [22]
		c	w	MIMD (traditional)

Where:
n is the number of processors
w is the width of a single processor
m is the number of program counters (PCs)
c is the size of the instruction memories

Table 2.1: Instruction/Control Architecture Taxonomy

ment and the device's resource allocation, the more efficient that device will be at running that application. [5] discusses this in more detail.

All modern general-purpose computing devices fix their resource allocation decisions when they are fabricated. As a result, there will be a set of application's whose needs match the choices made by that particular device - and there will be a large number of applications whose needs do not match the device's resources. In order to create a device that will be efficient over a wide range of applications and application requirements we need to be able change the resource allocation of the device *after* fabrication. Chapter 3 suggests a way this can be done.

Chapter 3

Meta-Configurable Architectures

3.1 Meta-Configuration

As discussed in Chapter 2, we would like to create a device who resource allocation choices can be made on a per-application basis, rather than at fabrication time. In order to accomplish this, such a device would need to be given at least two levels of configuration. The most basic level(s) would describe the exact resource allocation and architectural layout an application requires. We will refer to this kind of configuration as a **meta-configuration**. Once the application's desired architecture has been specified, the application itself can be programmed or configured on top of that architecture.

A meta-configuration could be a generic architecture specification, such as a "3 thread, 8-bit, VLIW microprocessor", or could include specific constants, such as "a (3x + 4y) calculator", depending on the flexibility required at run-time.

3.2 Building Blocks

In order to create a meta-configurable architecture, we need to first create a set of basic building blocks. Because we cannot know in advance what requirements applications may have, all the building blocks on the chip should be identical, or at least be spread uniformly across the chip.

There need to be at least several, preferably many, such blocks on a chip because of the possible need to create a spatial computing engine. On the other hand, each block, or a set of blocks, must be able to change its operations rapidly in response to a broadcast operation code, so that temporal computing devices can be created.

Each block, or a uniformly distributed set of blocks, needs to be able to provide any of the four basic resources (datapath compute, instruction distribution, control and memory) on demand. The provided resources should be reasonably high performance, in both speed and area, so that applications running on the device will not suffer when compared with more hardwired structures.

3.3 Granularity

The easiest approach to creating such a block is to create a block that contains a compute unit, a control unit, and a memory unit, and is connected to a switchable network which can carry data, instructions, and control information. An important question that needs to be asked is: how big should the block be?

A small block would allow many such blocks to be built onto a single die, greatly increasing flexibility. A small block, or group of small blocks, could also more closely match the actual data width of any given application than a large block, or set of blocks.

On the other hand, a larger system composed of small uniform elements, where each element is large enough to contain a compute engine, memory and control structures, will be much larger and slower than a device composed of larger basic elements. This is a result of the fact that the wires and switches needed to connect many small elements in a configurable way will require a great deal of area and time, while the larger blocks hardwire more connections so that they require less switching.

In addition, each block must be able to change its function rapidly in response to a broadcast operation. Our flexible substrate is subject to the same problems as are fixed architectures: A fine-grained device requires a great deal of wires, switches, and time to be able to broadcast an operation to all of its elements. All of these factors

argue in favor of a large building block.

The easiest answer is to compromise: create a block large enough that the area required for the switching and wires needed to broadcast instructions and compose the units doesn't completely dominate the block's area, yet small enough that its possible to put a significant number on a single die. We are fortunate that modern manufacturing technologies have reached a point where is this is easily feasible.

3.4 MATRIX

MATRIX (Multiple Alu archiTecure with Reconfigurable Interconnect eXperiment) is a prototype of a meta-configurable architecture. It utilizes a coarse-grain, 8-bit wide basic building block containing a memory, ALU and control unit. It connects these with a unified network which can carry data and instruction information interchangeably. The following chapters discuss the prototype architecture in depth, as well as discuss some of the tradeoffs involved in creating this kind of design.

Chapter 4

MATRIX Architecture Overview I: The BFU

MATRIX consists of an array of 8 bit wide functional blocks called Basic Functional Units (BFUs) connected in a reconfigurable multi-level network. Each block contains a memory, ALU, and a control unit, connected in a configurable manner. The 8-bit granularity of a MATRIX BFU was chosen so that an network line (8 bits wide) could carry a function specification, a memory address into a 256-byte memory, or a data byte. It was believed that a 256-byte memory would be large enough to be interesting, but would not take up the majority of the basic cell. This assumption proved reasonably correct, as we will see in Chapter 7. However, it turned out that 8 bits were not sufficient to fully specify a BFU's cycle-by-cycle operation. The need for more specification lead to the creation of a two-byte function input.

When originally conceived, the block would take in 3 inputs: memory address, data (or a second memory address), and a ALU function select. It would then compute on either incoming data, its own internal memory data, or both, and output a single result. Because of the need for more function specification, the core BFU now requires 4 byte-sized inputs. Figure 4-1 shows the current BFU architecture. The major elements of the BFU will be described below.

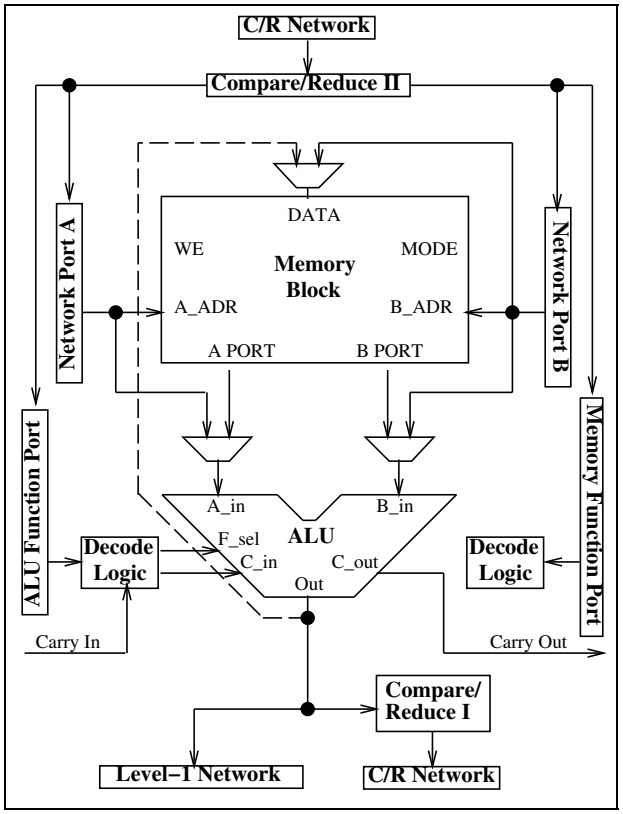

Figure 4-1: MATRIX Basic Functional Unit

4.1 Memory

The main MATRIX memory is a 256 word by 8 bit wide memory, which is arranged to be used in either single or dual port modes. The memory mode is controlled by the Memory/Multiplexor function port (see Section 4.4).

In single port mode, the memory uses the A_ADR port for an address and outputs the selected value to both ports. In dual port mode, the B_ADR port selects a value for the B_PORT separately from the A_PORT. However, in dual-port mode, the memory size is reduced to 128 words in order to be able to perform both read operations without increasing the read latency of the memory.

In both modes this read operation takes place during the first half of the clock cycle and the values are latched for the rest of the cycle. Write operations take place on the second half of the cycle. Writes are always done to the current A_ADR address. If the feedback path (shown in Figure 4-1 as a dashed line) is used, then the BFU is performing "A op B $\rightarrow$ A" in one cycle. Two cycles are needed to perform "A op B $\rightarrow$ C" operations, because there are currently only two memory address ports in BFU. In this case, the feedback is performed by the normal Level-1 network (see Chapter 5).

4.2 ALU

The MATRIX ALU is a basic 8 bit arithmetic logic processing unit. It is capable of performing the following operations:

Input Invert - Prior to performing any of the following operations either, or both of the ALU inputs can be inverted.

Pass - Passes either A or B input to Out. With the input inversion, this operation can be a NOT.

NAND - Performs bitwise operation: (A NAND B). With input inversions this can be an OR.

NOR - Performs bitwise operation: (A NOR B). With input inversions this can be a AND.

XOR - Perform bitwise operation: (A XOR B). With input inversions this can be a XNOR.

Shift - Shifts A or B either left or right one bit.

Add - Performs (A+B+Cin). Cin can be selected from 0, 1, or Cout of an adjacent cell. Combined with the input inversion a subtract can be made: (A-B)=(A + $\bar{B}$ + 1).

Multiply - Performs (A*B). Can also perform (A*B+X) and (A*B+X+Y), where X and Y are special inputs. These operations are needed to create pipelined multiply structures. Multiply operations require two cycles to fully complete. The low byte is available on the first cycle and the high byte is available on the second. The multiply operation will be described in more detail in Section 4.2.2, below.

4.2.1 Multi-BFU Operations

BFUs are designed so that they can be smoothly chained together to form wider-word ALU structures. In order to accomplish this, the user must specify the carry-chain of each of datapath element as it travels through multiple BFUs. In order to accomplish this, part of the meta-configuration needs to specify how the carry-chains are formed. In a BFU this is accomplished by setting the following bits:

LSB - Set to "1" marks the least-significant-byte position.

MSB - Set to "1" marks the most-significant-byte position.

Rightsource - Specifies the direction to the next least-significant-byte. Can also be set to receive a carry from another source (see below).

Leftsource - Specifies the direction to the next most-significant-byte. Can also be set to receive a carry from another source (see below).

The source selection can be one of the following:

North - North BFU.

East - East BFU.

South - South BFU.

West - West BFU.

Local - The local BFU's carry from the previous cycle.

Control Bit - The local Control Bit. See Section 4.3.

Zero - Constant Zero.

One - Constant One.

In addition, pipeline stages can be inserted into the carry chain by specifying another meta-configuration bit, **CarryPipeline**, to be "1". This will register the incoming carry prior to its being used. This is important for addition operations, because the carry-chain is limited by the clock period and the speed of the adder.

Based on this local information, the actual Shift and Add operations have different effects:

Shift

There are two main shift functions: **Left** and **Right**. Left shift moves the bits towards the MSB, and right shifts move the bits towards the LSB. Normally, the carry-in value is used to fill the newly-created opening, but if the cell is an LSB or and MSB the new bit is determined by additional information contained the chosen shift instruction. For Left Shifts the LSB position will be different, while for the Right Shifts it will be the MSB position. The options are:

Force Carry - This option will override the LSB/MSB setting and force the shift to use the carry-in from its designated source (Left/Rightsource). This allows BFU(s) to perform barrel-shift operations on a defined datapath.

Skip Bit - This option will keep the same LSBit/MSBit, essentially duplicating the low/high bit of the shifted number. This allows sign-extension operations.

Insert 0 - This will insert a zero into the LSBit/MSBit.

Insert 1 - This will insert a one into the LSBit/MSBit.

Addition

There are three addition functions: **Add**, **Add-0**, and **Add-1**. Add will perform a normal add-with-carry (A+B+Cin), in all cases. Add-0 will perform a normal add-with-carry, except that the Carry-In of the LSB block will be forced to zero. Add-1 is similar, except that the LSB Carry-In is forced to one.

Note that a "normal" addition operation is usually performed with the Add-0 function. The basic Add operation is primarily intended for performing "block serial" addition - in which addition is performed over multiple cycles on the same set of BFUs. The sequence would be an Add-0, followed by however many Adds are needed to complete the Addition.

Subtracts are performing using the Add-1 operation and inverting the B input value (2's complement subtract).

4.2.2 Multiply

Because many common applications require multiply operations, it was decided to include a multiply operation. As we will see in Section 7, the multiplier took up very little area, and can therefore be considered a good addition to the BFU.

However, the main problem with a hard-wired multiplier is that it produces 16 bits of output, while the datapath it setup for only 8 (or 9, if the carry is considered). When original conceived, the BFU had no mechanism for dealing with all 16 output bits so it was decided to have the multiplier output its result over two cycles: the first cycle outputs the low 8 bits of result and the second cycle outputs the high 8 bits.

In addition to performing a basic multiply, the array multiplier used in in building MATRIX is capable of performing additions into the multiply. It was decided to

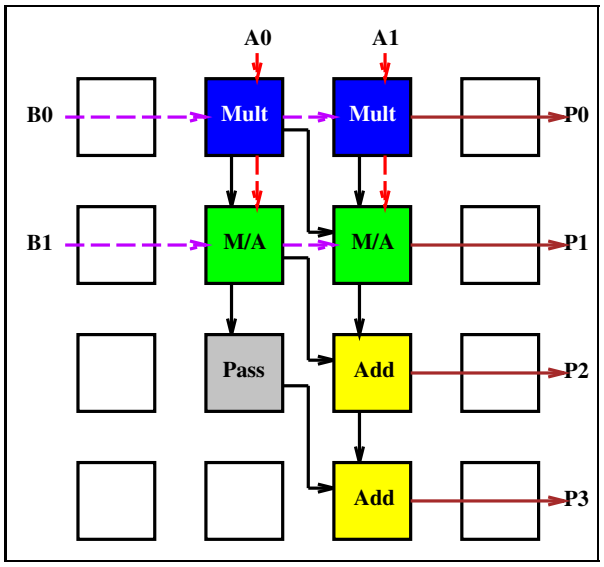

Figure 4-2: 16 Bit Pipelined Multiplier

include this function so that cascading BFU's into larger pipelined multiply structures would be possible (Figure 4-2).

The result is that there are four multiplication functions: **Mult**, **Mult-Add**, **Mult-Add-Add** and **Mult-Cont**. The first three initiate a multiply operation, performing A*B, A*B+X, or A*B+X+Y, respectively. The low byte of the product is available at the end of the current cycle. Mult-Cont is then issued in order to output the high byte. Mult-Cont does not have to be issued, but if it is it must immediately follow a Mult, Mult-Add, or a Mult-Add-Add. The inputs to the multiply are latched on the cycle the Mult, Mult-Add, or Mult-Add-Add is issued, so that the inputs to the BFU may be changed during the Mult-Cont function, without effecting the final value.

The source for X and Y, if used, are special. There are two meta-configuration bits associated with these inputs: **MAdd1source** and **MAdd2source**. If these are set to "0" they hardwire the X and Y inputs for use in pipelined multipliers (Figure 4-2). In this case the X input is connected to the nearest North neighbor (L1_N1), and the Y input is hardwired to the output of the Northwest neighbor (L1_NW) of the previous clock cycle (see Chapter 5 for information on the Level-1 network). If the

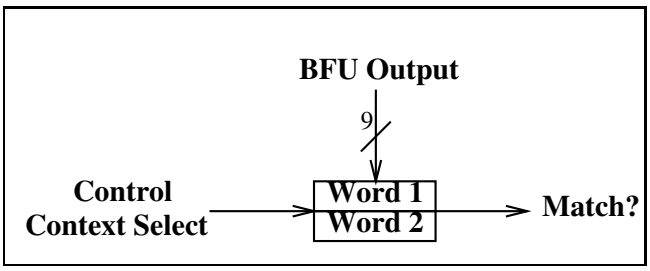

Figure 4-3: Comparison/Reduction Logic

MAddsource bits are set to "1" they allow special network switches called "floating ports" (see Chapter 5) to select the source of the multiply-adds.

Its important to note that this two-cycle output is not inherent in the multiplier design. As we will see in Chapter 5, the BFU can actually output up to 5 bytes of data on every cycle, so it is quite feasible to output all 16 bits simultaneously. It will be worth investigating this possibility for future designs because it is often difficult to create designs that fit within the two-cycle latency of the multiplier (see Chapter 8, for some example designs).

4.3 Compare/Reduce

Compare/Reduce is the first of two forms of control logic built into the MATRIX BFU. The second, a distributed PLA, will be described in Chapter 5. This Compare/Reduce serves as general-purpose "condition codes" of the outputs of a BFU.

Figure 4-3 illustrates what happens in Compare/Reduce I. The 9-bit output of the BFU (data plus carry-out) is compared to one of two programmed words. The Control Context Select (which is part of the ALU function - see Section 4.4, below) determines which word is used. These words can contain "don't care" bits, so it is possible to test any part of the BFU output. For example, a zero-detect function would test all of the data bits for zeros, but ignore the carry, while a sign-check would look only at the 8th (high) bit of the data and ignore the rest.

The result of this comparison is passed to all the BFU's neighbors in the same style as the Level-1 network (see Chapter 5). Figure 4-4 shows an example of a multi-

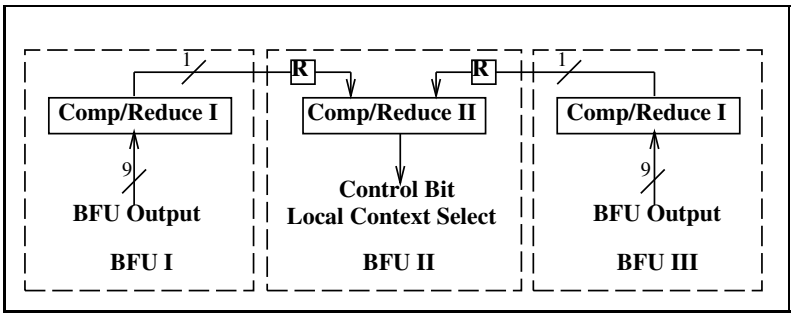

Figure 4-4: Multi-Cell Compare/Reduce Logic

BFU reduction. The Compare/Reduce II block performs a similar reduction on the C/R values from the BFU's neighbors, except that it uses only one comparison word.

The final result of these comparisons is a local Control Bit in each BFU. This control bit is used to change the functionality of the BFU network switches (see Chapter 6). By changing the functionality of the network switches, the Control Bit can be use to select between different BFU operations, such as different data inputs, different ALU functions, or different dataflow structures.

4.4 Input Ports

There are four port into the core BFU (Figure 4-1), each of which is 8 bits wide. The values on ports A and B are used as data for the or addresses into the memory. The selection between how they are used is controlled by the data on the Memory/Multiplexor Function Port, described below.

4.4.1 ALU Function Port

The ALU Function Port (Fa Port) controls the operation of the BFU's ALU, the write enable (WE) for the main memory, and the Compare/Reduce word selection (see Section 4.3). The ALU controller decoding is described below.

The inclusion of the memory write enable in the Fa Port was done because the ALU function port is intended for things that are frequently changed on a cycle-by-

ALU Opcode	Operation
0	Multiply
1	Multiply-Add
2	Multiply-Add-Add
3	Multiply-Cont
4	Shift with Force Carry
5	Shift with Copy Bit
6	Shift with Insert 0
7	Shift with Insert 1
8	Add
9	Add-0
10	Add-1
11	(Add-1) [1]
12	Pass
13	NAND
14	NOR
15	XOR

Table 4.1: ALU Opcodes

cycle basis. The Memory/Multiplexor Function Port (Fm Port - described below) was added to control thing that are not frequently changed, but are not static enough to be included in the meta-configuration. As we will see in Chapter 6, it is possible to statically set the value of a port without consuming network lines. This means that if an application doesn't need to change the Fm port's value (a likely occurrence), it does not need to allocate network lines to supply the value.

Table 4.1 lists the ALU opcodes. In addition to these, two additional control bits are used: **Invert A** and **Invert B**. During normal operation, these bits will perform a bit-wise invert on the A and B ALU input respectively. This is used with the logical operations, as well as with the Adds in order to generate a subtract.

During Shift and Pass operations, however, these bits serve special functions:

Shift Invert-A is used to select the Shift Direction (Left or Right) and Invert-B is used to select the Shift Source (A input or B input). In the current model, there is no way to perform an inversion during a shift operation.

[1] This is an unused opcode but will generate an Add-1 if issued.

Pass Invert-A is used to invert the Pass value and Invert-B is used to select the Pass Source (A or B input).

4.4.2 Memory/Multiplexor Function Port

The Memory/Multiplexor Function Port (Fm Port) controls the less frequently needed parts of the BFU function:

Main Memory Mode Selects between one-ported (256 byte) and two-ported (128 byte) memory mode.

ALU Input Selectors Selects between memory and input data port inputs for the ALU.

Memory Data Select Selects between input data port and write-back data for the main memory write.

Configuration Memory Read/Write Controls writes to the configuration memories.

The last item deserves a little more explanation. The BFU contains a set of configuration memories which store the meta-configuration used by the BFU and network switches. These memories can be written to from the normal network ports, making it possible for the BFU's to reprogram themselves during operation.

When the configuration memory write enable (CWE) is asserted, the BFU takes the A input as address, and the B input as data and writes to the configuration memories rather than the main memory. Similarly, when the configuration memory read enable (CRE) is asserted, the BFU outputs the value in the configuration at the address specified by the value on input port A.[1]

The normal programming methodology will be discussed in Chapter 6.

[1] In the current implementation, the configuration memory value is actually output onto one of the Level-3 network lines.

Chapter 5

MATRIX Architecture Overview II: The Network

5.1 Network Ports

As was described in Chapter 4, the core BFU is connected to the network through 4 ports. The network itself uses 4 additional ports for its own switching. Figure 5-1 shows how all 8 ports are connected. Four switch-ports (Address/Data A and B, Fa and Fm) feed data into the BFU core. Four other switches: Network Switches 1 and 2, and Floating Ports 1 and 2 (FP1, FP2, N1 and N2) feed data into the Level-2 and 3 network drivers.

The mechanism used to implement each of these switches will be described in Chapter 6. The network drivers will be described in Section 5.3.

Each switch/port selects from its inputs to produce a single byte of output. The inputs to each switch are listed in Table 5.1. The Control Byte comes from the distributed PLA, described in Section 5.4. The switches are used uniformly for data, control, and instruction information.

Source	Description
Local	The local BFU
L1_N1	Level-1 Network, From North-1 cell
L1_N2	Level-1 Network, From North-2 cell
L1_NE	Level-1 Network, From NorthEast cell
L1_E1	Level-1 Network, From East-1 cell
L1_E2	Level-1 Network, From East-2 cell
L1_SE	Level-1 Network, From SouthEast cell
L1_S1	Level-1 Network, From South-1 cell
L1_S2	Level-1 Network, From South-2 cell
L1_SW	Level-1 Network, From SouthWest cell
L1_W1	Level-1 Network, From West-1 cell
L1_W2	Level-1 Network, From West-2 cell
L1_NW	Level-1 Network, From NorthWest cell
L2_N1	Level-2 Network, North-1 Line
L2_N2	Level-2 Network, North-2 Line
L2_E1	Level-2 Network, East-1 Line
L2_E2	Level-2 Network, East-2 Line
L2_S1	Level-2 Network, South-1 Line
L2_S2	Level-2 Network, South-2 Line
L2_W1	Level-2 Network, West-1 Line
L2_W2	Level-2 Network, West-2 Line
L3_V1	Level-3 Network, Vertical-1 Line
L3_V2	Level-3 Network, Vertical-2 Line
L3_V3	Level-3 Network, Vertical-3 Line
L3_V4	Level-3 Network, Vertical-4 Line
L3_H1	Level-3 Network, Horizontal-1 Line
L3_H2	Level-3 Network, Horizontal-2 Line
L3_H3	Level-3 Network, Horizontal-3 Line
L3_H4	Level-3 Network, Horizontal-4 Line
CByte	Control Byte
C0	Constant Value 0 (Binary: 00000000)
C1	Constant Value 1 (Binary: 00000001)

Table 5.1: BFU Switch Port Inputs

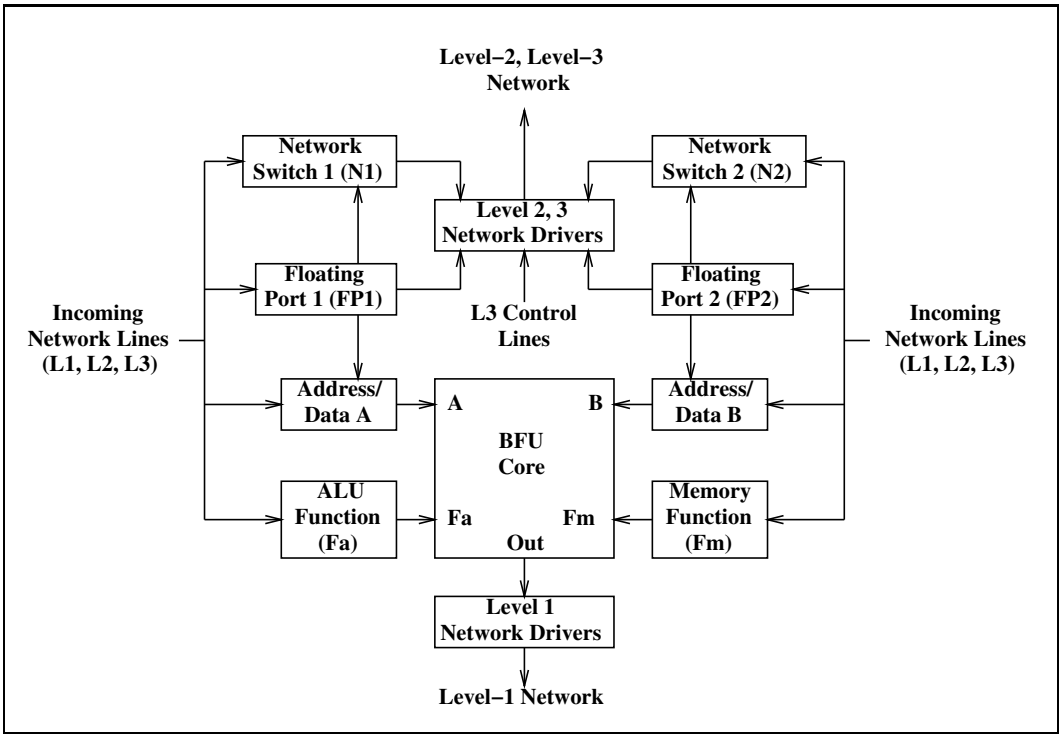

Figure 5-1: MATRIX Network Switch Architecture - BFU Cell

5.1.1 Floating Ports

The BFU's floating ports are special switches because they are used for several different functions. When not being used as network selectors, FP1 and FP2 can serve to control the dynamic switching capability of the A,B,N1 and N2 ports (described in Chapter 6). In addition, FP1 and FP2 can feed data to the control PLA (described in Section 5.4), or can select the source for the Multiply-Adds (Chapter 4).

The reason the floating ports serve so many functions is that every switch included in the BFU significantly increases the size of the BFU (see Chapter 7). Because of this, it is infeasible to dedicate a switch for every possible function. Rather, the floating ports serve many functions which are unlikely to be used in combination. It remains to be seen how seriously this will hurt application designs, if it will effect them at all.

5.2 Network Lines

The MATRIX network is intended to provided high-bandwidth connections between BFUs in a flexible, configurable manner. A three-level interconnect structure, consisting of a regular neighborhood mesh, longer switchable lines, and long broadcast lines was chosen. It was believed that this provided sufficient balance between local broadcasts and long distance connections. However, it turned out that the currently implemented network lines are useful in ways not planned for in the original design. This will be discussed in more detail below.

The current network architecture was designed to be used on chips containing up to 256 (16×16) BFUs. Larger chips would probably benefit from a 4th level between the current L2 and L3 levels, or making the L2 network longer than 4 BFUs.

5.2.1 Level 1

The Level-1 (L1) network was intended to carry data from a BFU to its nearest neighbors. From the beginning it was intended that this communication should happen in the same cycle as the compute, so that the full cycle time looks like:

$$MemoryRead \rightarrow ALUCompute \rightarrow L1NetworkTransition/MemoryWrite$$
$$\rightarrow IncomingAddress/DataLatchedatPorts$$

Being on the critical path, the L1 must be fast. This limits the distance it can traverse. Timing simulations determined that a manhattan distance of 2 would be the maximum distance into order to maintain a reasonable cycle time (100 MHz). Diagonal connections were including, despite the fact that they increased the size of each input switch (Chapter 6) by 4 inputs, because it made it possible to build compact array multipliers and other, inherently diagonal, designs.

Figure 5-2 shows the current Level-1 network structure. The 8-bit output of every BFU is passed a manhattan distance 2 in every direction. As a result every cell receives 12 L1 inputs.

The major drawback to the Level-1 network is the fact that it broadcasts the data to *all* its neighbors on *every* cycle. Because these are high-speed lines, the power

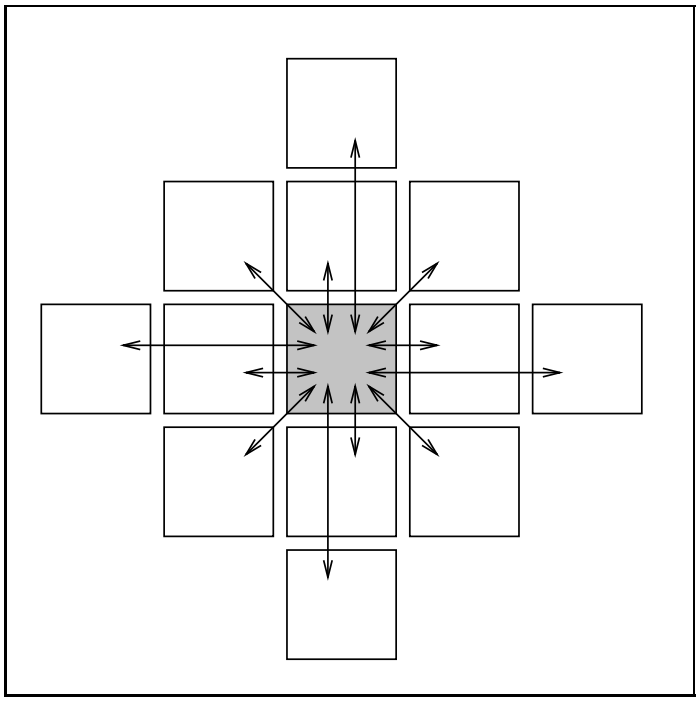

Figure 5-2: Level 1 Network Connections

required to accomplish this becomes quite significant. It was estimated that an array of 64 BFUs would use over 8 watts of power just driving the L1 wires. As a result, it was decided to include a mechanism to turn off network lines that are not being used in a design. This is now part of the meta-configuration of a MATRIX design.

5.2.2 Level 2

The Level-2 (L2) network was intended to carry data intermediate distances (in steps of four) across the chip. It turns out that actual designs have tended to use the L2 network for the fact that it can pipeline data (see below), rather than for its distance communication. Many of the experimental applications that have be mapped to MATRIX require registers for pipelining and retiming that are not easily available anywhere else without sacrificing a complete BFU as a register. Future designs of the L2 network should reflect this change of purpose.

The current Level-2 network uses two drivers in every BFU (see Section 5.3).

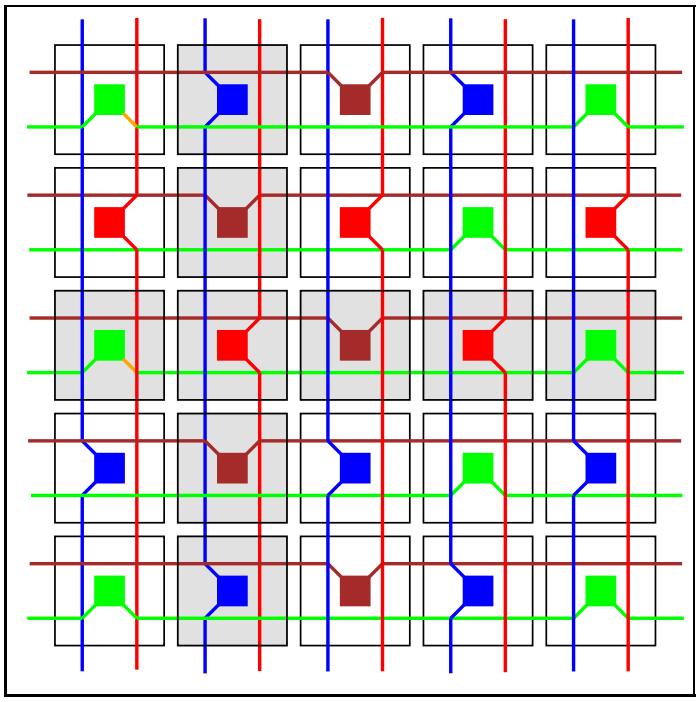

Figure 5-3: Level 2 Network Connections

These broadcast along length-4 (4 BFUs) lines either horizontally or vertically. This results in a checkerboard tiling of BFUs. Figure 5-3 shows this structure. Every colored block in Figure 5-3 represents two Level-2 network switches. Each line shown is a 2-directional broadcast line where the starting switches are the source of the broadcast. Every BFU that a line crosses has access to the data being broadcast on the line.

The checkerboard design was chosen even though it made mapping designs which use the L2 network difficult, because it cut down the size of the BFU. Adding the two additional drivers for each BFU, to complete the symmetry, would add 8 new switch inputs to every BFU, as well as require the additional switches and drivers in each BFU. Given the sizes of the switches (see Chapter 7), this was deemed excessive.

Pipelining on Level-2

Level-2 drivers operate in two modes: Source and Pass. These modes are part of the chip's meta-configuration. In Source mode, the data selected by one of the network

switches is registered and broadcast on the line on the next cycle. The register is used to add pipeline stage in network, because the transit time on the L2 network would exceed the basic cycle time.

In Pass mode, the data is broadcast without the pipeline stage. This allows longer chains of network lines. At some point, a pipeline stage must be inserted (by using a Source-mode switch) to keep the clock period small. The possible number of links in these chains depends on particular implementations of this design as well as the internal clock speed.

The L2 drivers are also capable of being deactivated when not in use to save power, in the same manner as the the Level-1 drivers.

It turned out that the Level-2 network's ability to add a register every 2 BFUs was more useful to many applications than its ability to carry data. Many systolic computing structures require that data be retimed or pipelined across a structure, and the L2 registers are the only current mechanism for accomplishing this.

5.2.3 Level 3

The MATRIX Level-3 (L3) network is intended to carry data long distances as rapidly as possible. It consists of 4 shared network lines spanning every MATRIX row and column. Each BFU cell gets to drive up to 4 inputs onto the L3 network. Section 5.3 describes how this is done. In addition, every BFU has access to every Level-3 line crossing it.

The delay across Level-3 is also one clock cycle per step, except that steps at this level are up to a full-chip long. Thus it is possible to get from any BFU to any other BFU in a MATRIX array in 2 clock cycles.

The control logic, to arbitrate the bus lines, for the L3 network is located at the perimeter of the MATRIX core.

5.3 Network Drivers

There are 2 Level-2 and 8 Level-3 tristate drivers in every BFU. Each uses the

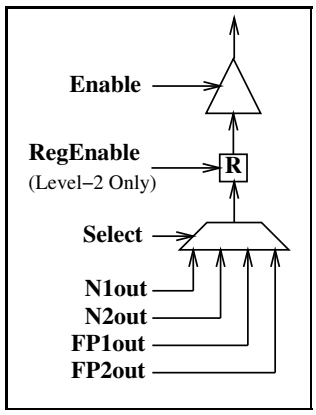

Figure 5-4: Level-2 and Level-3 Network Drivers

Network and Floating Ports (N1,N2,FP1,FP2) to select their inputs on a cycle-by-cycle basic. The assignment of switches to drivers, however, is set by the meta-configuration. Figure 5-4 shows a generic L2 or L3 driver for this network. One of the four switches is configured to drive each line. In the event that the line is not used, it can be completely disabled in the same way as the L1 lines. On the L3 network, these drivers as actually tristate, and are controlled globally. Finally, the Level-2 network contains the optional registers - these set the Source/Pass mode of this L2 driver. On the L3 drivers, the register is mandatory.

This setup allows up to 4 data values to be driven onto the L2 and/or L3 network on every cycle. Including the L1 driver, this gives a BFU up to 5 bytes of output per cycle.

5.4 Distributed PLA

The Compare/Reduce logic, described in Chapter 4, performs fast reduction and control operations if the control is simple. However, this many not be adequate for more complex control operations. In order to handle these cases, a distributed PLA was included in the MATRIX design.

A distributed PLA is a normal PLA where each of the two planes (AND and OR, usually implemented as two NOR planes), are physically scattered across the chip and connected in a configurable manner. Figure 5-5 shows an example of how this

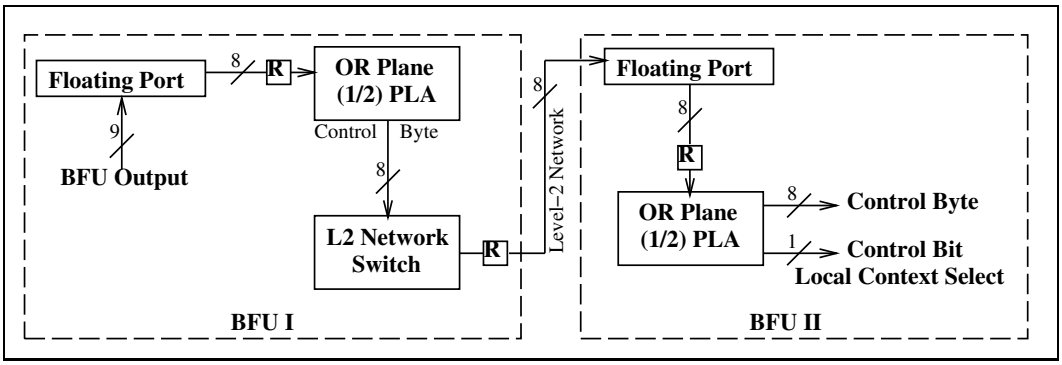

Figure 5-5: Distributed PLA

works.

The BFU output from BFU I gets passed to an OR plane which is used in place of a NOR plane because the inversions can be performed at the inputs to the OR, and at the ALU of the final BFU. The the fact that a Floating Port is used to switch this allows *any* network input to serve as initial data. The register after the floating port provides the necessary pipeline stage if the data used is coming off a long network line.

The OR plane serves as one stage of a multi-level logic function. Therefore its eight outputs can be thought of as product-terms of a standard PLA. These product terms are then passed to a Level-2 or Level-3 network switch.

After the one cycle delay from crossing the network, one of BFU II's floating ports switches the product terms to its OR plane. This plane performs the second stage of the multi-level logic function. If more stages were required, 8 new product terms could be sent to another BFU to continue the operation. In the example shown, only two levels are required.

In the distributed PLA control logic, there are two final outputs. The first is the same as the C/R logic: the local Control Bit used to change the function of the network switches. However, the PLA can also output a Control Byte, which can be inserted into a BFU port or network switch. This allows the control logic to generate specific constants.

Note that the distributed PLA control requires 3 cycles to complete a two-level

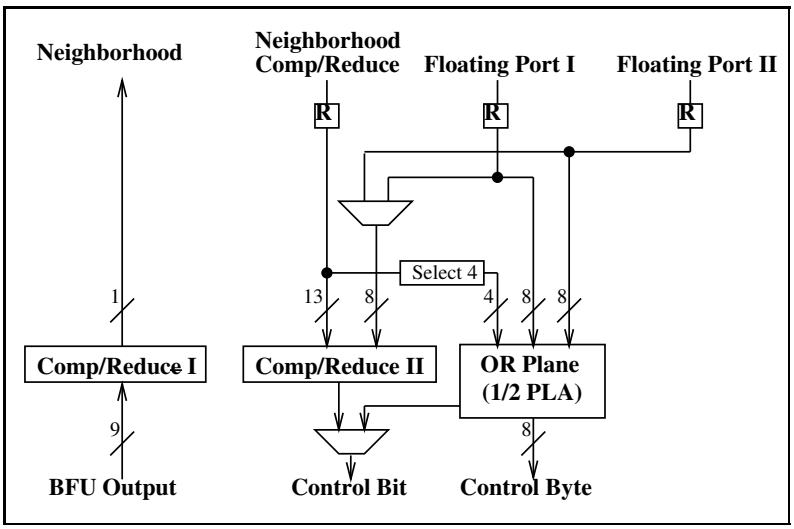

Figure 5-6: BFU Control Logic

logic operation, but is capable of performing complex logic operations as well as distributing this control across large portions of a MATRIX chip (the Level-2 and Level-3 network spans). On the other hand, the C/R logic operates in a single cycle, but is limited in functional complexity and distance.

5.5 Complete Control Logic

Figure 5-6 shows the complete control logic for a single BFU. The Comp/Reduce I is performed just as described in Chapter 4, while the Comp/Reduce II is linked with the OR plane. This connection allows these two styles of control logic to be mixed. For example, the Neighborhood Comp/Reduce can be used as an input to the OR plane, or the floating port outputs can used in the Comp/Reduce II operation.

In order to reduce the size of these reduction operations, a number of pre-selections are made on the incoming data. Comp/Reduce II operates on all 13 C/R inputs, but can only include one of the Floating Port values. The OR Plane takes both Floating Ports (so that it can combine the outputs), but only takes 4 bits of the C/R inputs. Any 4 can be selected as part of the design's meta-configuration.

One final bit of meta-configuration selects the source of the Control Bit: C/R II or OR plane.

Chapter 6

MATRIX Architecture Overview III: The Switches

As was described in Chapter 5, the BFU's are connected to the interconnect network through a set of eight switches. The architecture of these switches is unique, because they provide the mechanism by which MATRIX can be meta-configured.

6.1 Switch Architecture

MATRIX switches operate in three modes: **Static Value**, **Static Source**, or **Dynamic Source**. Each of these will be explained in more detail below. Figure 6-1 shows the architecture of a MATRIX switch. Each switch takes in values from the 30 network lines crossing a BFU (these are listed in Table 5.1).

The switch is controlled by a 10 bit configuration word. This word contains 8 bits of data, and 2 bits which determine how the switch will interpret that data. This combination constitutes the basis of MATRIX's meta-configuration, as we will see here and in later chapters.

6.1.1 Static Value

In Static Value mode, the switch passes the 8-bit data byte directly to its output, as shown by the dark line on Figure 6-2. This allows port value to be set without

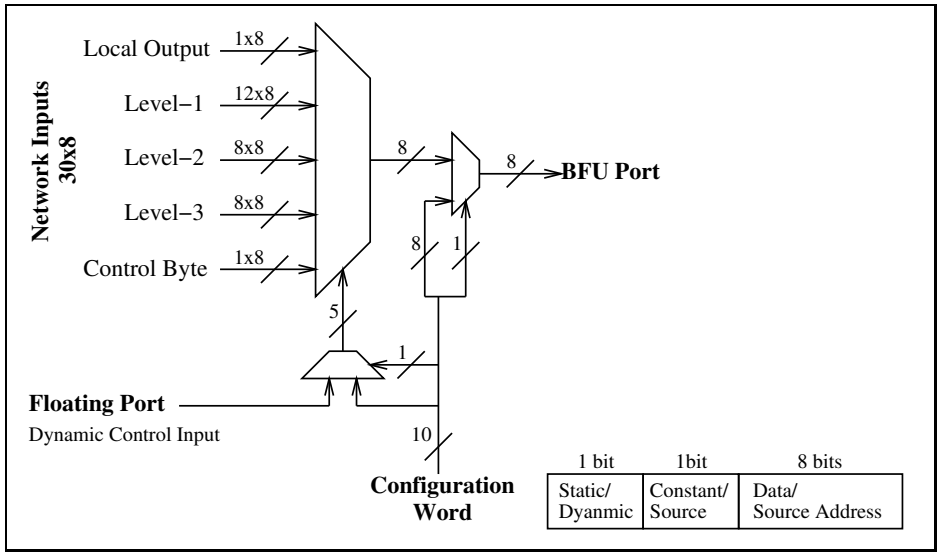

Figure 6-1: MATRIX Dynamic Switch Architecture

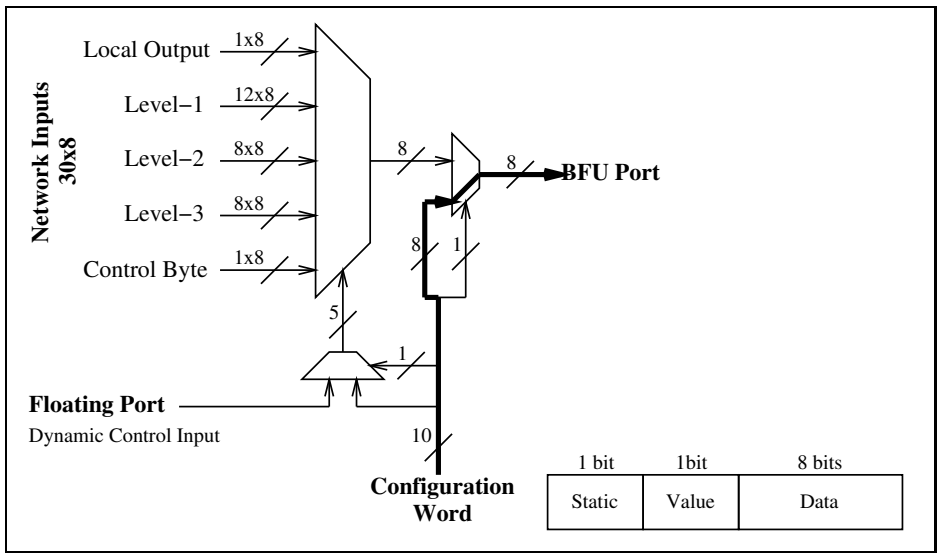

Figure 6-2: MATRIX Switch in Static Value Mode

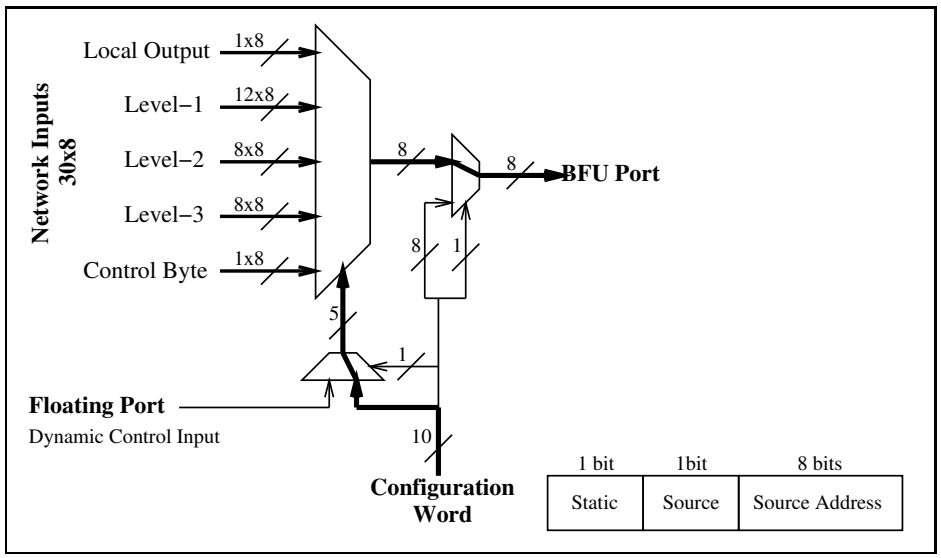

Figure 6-3: MATRIX Switch in Static Source Mode

consuming network wires. For example, if a BFU is always performing add operations, the add instruction would be programmed into the configuration word for the Fa Port's switch, and the switch set to static value mode. This will fix the ALU operation to add without consuming network lines to broadcast that instruction. This is especially useful for the Fm Port because, as was discussed in Chapter 4, the functions controlled by this port are often constant during normal operation.

In addition to fixed instructions, this mode can be used to assert constant memory addresses or insert specific constants into the BFU data ports, or onto the the network.

6.1.2 Static Source

In Static Source mode, the switch uses 5 bits of the data byte to select one of the incoming network lines to pass its data onto the BFU port. Figure 6-3 shows the paths used in this mode. This mode allows the data, instruction, and control paths through the network to be statically set as part of the meta-configuration.

6.1.3 Dynamic Source

In Dynamic Source mode, the switch allows an outside source to control the cycle-

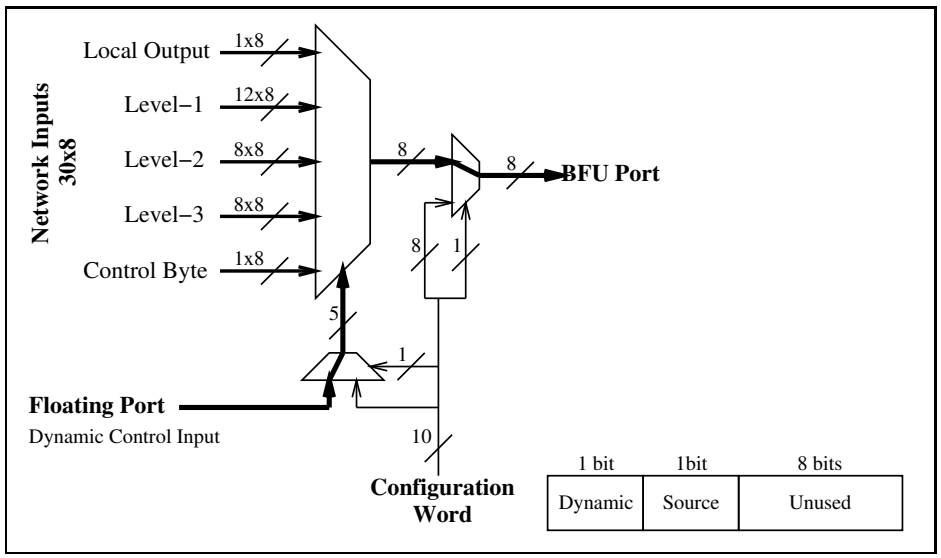

Figure 6-4: MATRIX Switch in Dynamic Source Mode

by-cycle selection of incoming network lines. This presents the 30→1 multiplexor to the programmer who can use it as part of the meta-configured design. The MATRIX switches use the floating ports to generate (select the source for) this dynamic control.

6.2 BFU Switches

The actual eight switches in every BFU are all variants of the switch described above. The main differences are that the function ports (Fa and Fm) and the Floating Ports (FP1 and FP2) do not support the dynamic source mode. This was done to simplify the design because dynamic control for all ports seemed excessive.

In addition, the four BFU core ports (Fa, Fm, A, and B) all have registers attached, in order to establish the pipeline stage on every BFU operation.

6.2.1 The Control Bit

As discussed in Chapters 4 and 5, the Control Bit generated by the control logic changes the function of the ports. Figure 6-5 shows how this is done. Every switch actually has two independent configuration words, and the Control Bit selects between them. This mechanism allows the Control Bit to change the ALU operation, an

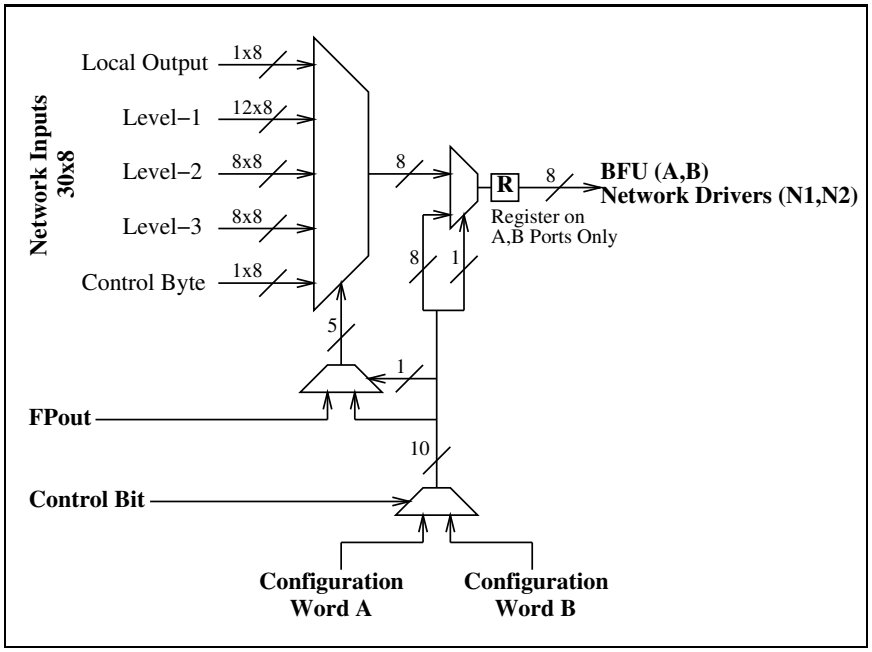

Figure 6-5: Switch Architecture with Control Bit

input constant, a memory address, or even the datapath/control flow. If no change is desired, the same data can be programmed in both configurations. Its important to remember that a BFU's Control Bit changes the operation of *all* eight switches simultaneously.

6.3 Configuration Memories and Programming

Chapter 4 described how the BFU's configuration memories were programmed through the Configuration Memory Read/Write Enable bits in the Fm Port. The difficulty with this system is that it requires the port configuration to exist in a known state at startup, so it is possible to route the address/data pairs, as well as the enables themselves, to the BFUs. In MATRIX this is accomplished by giving all of the configuration memories on the chip several **Global Contexts**.

In the current prototype there are four such contexts, as shown in Figure 6-6. Two of these (Contexts 2 and 3) are programmable, while the other two (Contexts 0 and 1) are hardwired.

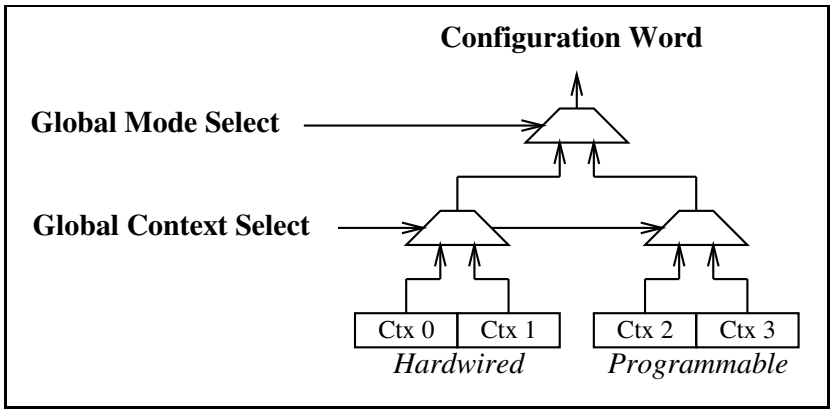

Figure 6-6: Configuration Memory Structure

The hardwired contexts used to bootstrap the chip. When set to Context 0, a MATRIX chip looks like a memory chip in write mode, so an external device can generate address/data pairs to program both the configuration memories as well as the main memories. Context 1 sets to the chip to act a memory in read mode, so that a configuration state can be offloaded. More sophisticated uses of the hardwired contexts are possible, such as a machine that will automatically load configurations from a passive memory off-chip, or even complete designs to manage system-level startup issues. For the sake of simplicity, these were not implemented in the initial prototype.

The programmable contexts are the ones used to hold meta-configurations for user applications. When originally conceived, MATRIX was intended to have a "background-load" feature. This would allow a second meta-configuration to be loaded while another was in use. The new design could then be swapped into operation in a single cycle, allowing MATRIX to change algorithms or even entire designs rapidly. This turned out to be too complicated for the initial prototype. The current design still allows designs to be rapidly swapped out, only now they cannot be re-loaded without interrupting a running design.

Chapter 7

Prototype Implementation

The MATRIX prototype is being implemented in a 0.5μm, 3 metal layer CMOS process. A complete BFU has been designed and floorplanned, and results of this will be described below. However, timing analysis of the circuitry has not yet been completed, so timing results are not available at this writing. Initial estimates suggest that this prototype will be able to achieve a 10ns (100Mhz) cycle time.

One of the main goals of the layout was to keep the BFU as small as possible. As we will see in Chapter 8, the performance of a MATRIX chip of a given size is directly related to the number of BFUs that can be fit on it. The original targeted BFU size was 1mm$\times$1mm. As we will see, this turned out to be too difficult to accomplish on this first pass design.

7.1 Floorplan

Figure 7-1 shows the floorplan design for a MATRIX BFU. The design evolved from the BFU block diagram (Figure 4-1), with the main memory in the top center and the switches feeding addresses and data in from the sides.

Figure 7-2 shows how the network wires travel across the BFU. The blue lines (dark grey) are horizontal, red lines (light grey) represent the vertical wires. Data on the wires is switches in the eight switches, and travels down to the registers. From the registers the data is passed to the rest of the BFU, including the memory and

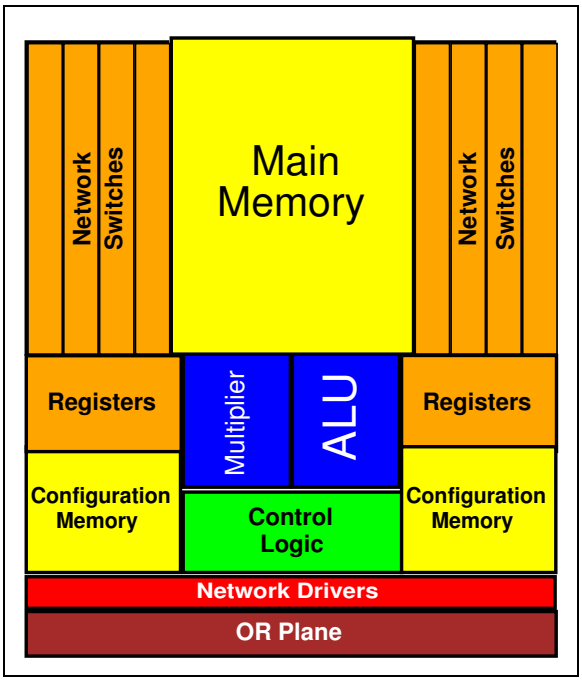

Figure 7-1: BFU Floorplan

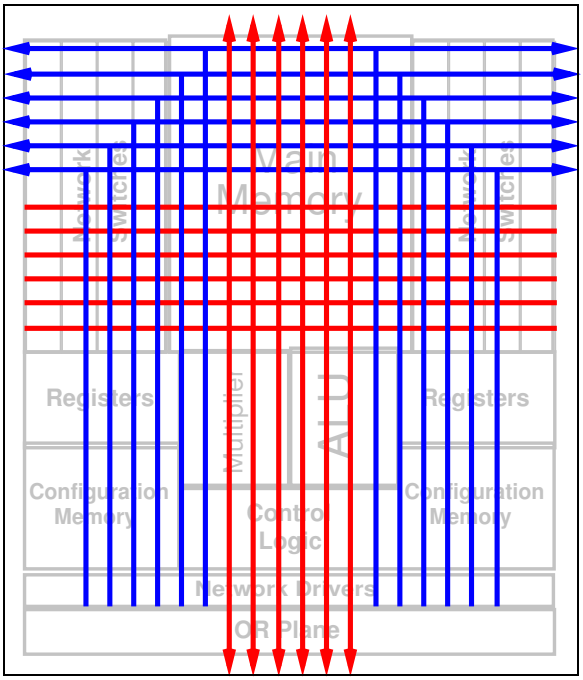

Figure 7-2: Network Wires Over A BFU

Component	Dimensions (μm)	Area (μm^2)	Percentage
BFU	1500×1200	1.8M	100%
MainMem	755×620	468,100	26%
ALU	230×265	60,950	3.4%
Multiplier	215×265	56,975	3.2%
Switches	(700×67)×8	375,200	20.8%
Switch Config	(284×67)×8	152,224	8.5%
Registers	(186×67)×8	99,696	5.5%
ORplane	1050×100	105,000	5.8%
C/R I	(60×50)×9	27,000	1.5%
C/R II	(60×50)×21	63,000	3.5%
Drivers, Misc Logic, Unused Area			21.8%

Table 7.1: BFU Area Results

ALU.

The floorplan shown in Figure 7-1 is certainly not an optimal layout. For example, if the switches, registers and switch configuration memories were built together, the overall structure would be smaller, and probably faster, due to the large amount of wiring currently used to connect these units.

However, even the current BFU design does not result in poor performance. A BFU of 1.2mm×1.5mm allows a MATRIX chip consisting of a 10×10 array of BFUs to be fabricated in a reasonable die size. A MATRIX chip of this size would have a raw performance of 10 billion (8-bit) operations per second.

7.2 Area Results

Table 7.1 shows the breakdown in area usage of the BFU components. The whole BFU is approximately 1.2mm×1.5mm. Some of the significant results of this are:

- The main memories account for 26% of the whole BFU area. This means that the 256x8 bit size is not too large for this BFU. However, if the BFU were to become smaller, this size will become quite significant. in a 1mm×1mm BFU, this memory would consume nearly 47% of the area. Under those circumstances,

either a tighter memory design, or a smaller memory size would be required.

- The switches, and their associated configuration memories and registers account for 34.8% of the current BFU. This amounts to over 4% per switch. If the BFU had been its targeted size, this would be nearly 63%, or nearly 8% per switch. Clearly, a tighter layout would be required in order to make the BFU smaller.

  Another possibility would be to reduce the size of the switches themselves. If each switch took 15 inputs, instead of 30, the switches would fall to under 25% of the current BFU size, or 44% of the 1mm$\times$1mm BFU. If this option is chosen, the network would have to be redesigned (which may be a good thing), because there would be a large degree of asymmetry in which network lines were visible to the BFU.

- The actual computing logic is only 6.6% of the current BFU, or almost 12% of a 1mm$\times$1mm BFU. Compared to the other component this is almost insignificant. However, even though more functionality may balance the situation, its probably more advantageous for the whole chip to reduce the size of the other components instead of increasing the ALU.

- The multiplier essentially doubles the size of the current ALU, but since the ALU is so small (in comparison with everything else), the addition of the multiplier was a net win.

Chapter 8

MATRIX Application Example: FIR

In order to illustrate how MATRIX works, this chapter will examine a simple application in depth. The application used is a Finite Impulse Response (FIR) convolution, a common primitive in signal processing. The problem is to take a set of k weights $\{w_1, w_2, \ldots w_k\}$ and a sequence of samples $\{x_1, x_2, \ldots\}$, and compute a sequence of results $\{y_1, y_2, \ldots\}$ according to:

$$y_i = w_1 \cdot x_i + w_2 \cdot x_{i+1} + \cdots + w_k \cdot x_{i+k-1}$$

where each $w_k \cdot x_i$ is called a **Filter TAP**. These examples are based on 8-bit sample data (x_i) and a 16-bit accumulate (y_i).

8.1 Comparison Benchmark

In order to compare the efficiency and performance of MATRIX designs with more conventional architectures, we will employ the metric of **functional density**, similar to the one used in [5]. Functional density measures the **capacity** per unit area of

---

I am indebted to André DeHon for working through the details of these examples. This material first appeared in [13].

a device. The capacity of a device is roughly the number of unit operations it can perform in a unit of time.

In this case, we will measure the number of filter TAPs per unit area. We will use the second (s) as the unit of time. The unit area will be in terms of λ^2, where λ is one-half the minimum feature size of the process used (a 1μm process would have a λ of 0.5μm). This will help to eliminate advantages due to superior process technologies so we can evaluate devices in terms of their architectures only. The resulting benchmark looks like:

$$\frac{Filter\ TAPs}{\lambda^2 \cdot s}$$

The advantage of using a capacity-based measurement, especially when process technology variations have been normalized out, is that we can compare the efficiency with which an architecture utilizes its silicon area for a the given task. Because the amount of normalized silicon area used is directly related to the raw cost of fabricating a chip, this measurement can be viewed as a kind of price/performance benchmark rather than one of the maximum performance benchmarks traditionally used.

For all these examples we will be assuming a MATRIX BFU is 1.2mm×1.5mm, in a 0.5μm process (as shown in Chapter 7), giving it a size of $\approx 29M\lambda^2$. We will assume a clock rate of 100MHz, giving a 10ns cycle.

8.2 Systolic - Spatial FIR

8.2.1 Implementation

Figure 8-1 shows a purely systolic or spatial implementation of the FIR filter, with eight TAPs ($k = 8$). Every block in the array is a BFU configured to act as labeled. The top row simply acts a staged pipeline, carrying the input down the row at one cycle per step. The multiply cells perform the 8×8 multiplication against hardcoded weights. The lower two rows accumulate the 16-bit results of the multiply operations.

This example uses the BFU ports in Static Value mode to set the function of

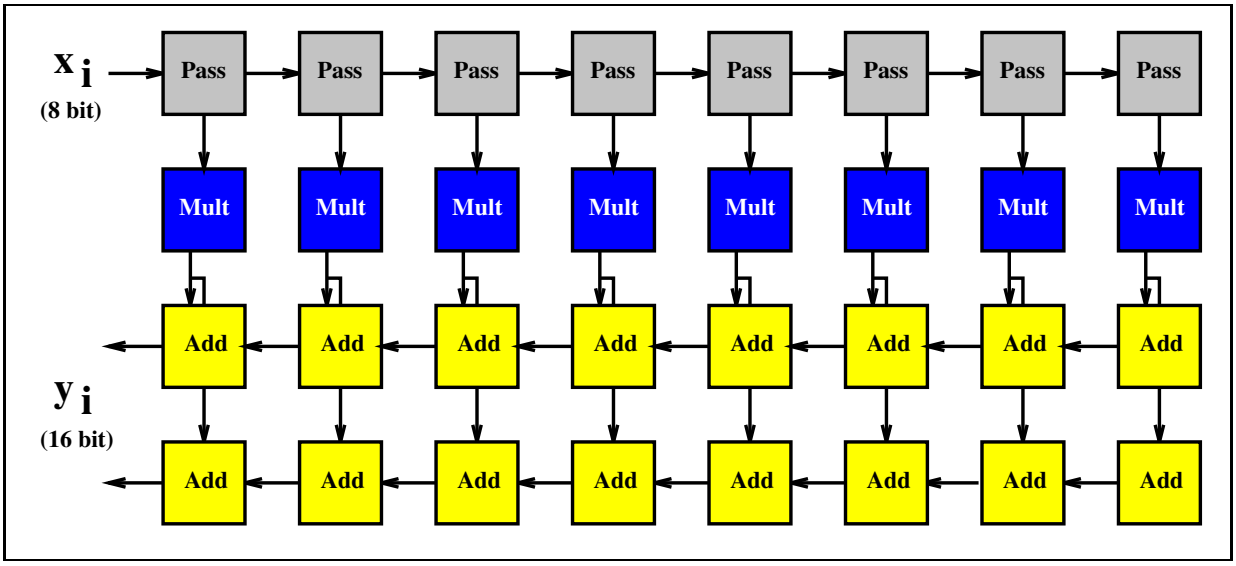

Figure 8-1: Systolic FIR Implementation

each of the BFUs as well as the weights on the multiply operations. In addition, Static Source mode is used on the data inputs to define the datapaths (the arrows in Figure 8-1).

The performance limiting step in this case is the multiply operation because it takes two cycle to complete the 16 bits of result. As a result, new inputs can only be fed in every other cycle, giving a throughput of 50MHz.

The implementation in Figure 8-1 uses 4 BFUs per filter TAP, but a more involved implementation could:

- Use the horizontal Level-2 lines for pipelining the inputs, removing the need for the top row of BFUs simply to carry sample values through the pipeline.
- Use both the horizontal and vertical Level-2 lines to retime the data flowing through the add pipeline so that only a single BFU adder is needed per filter tap stage. This is an example of an unplanned use of the Level-2 lines
- Use three I-stores and a program counter (PC) to control the operation of the multiply and add BFUs, as well as the advance of samples along the sample pipeline. This design would be a hybrid between the systolic implementation and the microcoded example in Section 8.3.

Device	MATRIX	FPGA (XC4K)	
Reference		ICSPAT93 [4]	App. Note [8]
Size	2 BFUs/TAP	100 CLBs/TAP	67 CLBs/16-TAPs
	29Mλ^2/BFU	1.25Mλ^2/CLB	
Speed	20 ns cycle	100 ns cycle	184 ns cycle
Density $TAPs/\lambda^2 \cdot s$	$0.87/\lambda^2 \cdot s$	$0.08/\lambda^2 \cdot s$	$1.0/\lambda^2 \cdot s$ (symmetric)

Table 8.1: Systolic FIR Performance Density Comparison

Thus, the k-weight filter can be implemented with only $2k + 4$ cells in practice.

8.2.2 Performance Density

Table 8.1 shows a density comparison between the MATRIX systolic FIR implementation and two other systolic FIR implementations, both done on Xilinx 4000-series FPGAs. The second FPGA design is restricted to symmetric weights, while both MATRIX and the first FPGA design can use fully flexible weights.

Using an average of 2 BFUs per TAP, at 50MHz, MATRIX compares favorably even to the symmetric-weight FPGA design, and is a factor of 10 more dense than a more typical FPGA design. In addition it has a much higher overall throughput.

8.2.3 Conclusions

As we will see, systolic designs can achieve the highest raw performance density of any design styles on a general-purpose computing architecture. However, there are two main drawbacks to systolic designs. First of all, they require resource to be allocated for every operation to be performed. In this case, 2 BFUs are required for every TAP desired, regardless of the clock rate. This sets a minimum area for these designs which can grow to be quite large for many TAPs.

The second drawback is also related to the minimum area. If the application's required throughput is lower than that provided by the design, the design's *yielded* performance density *decreases* from the peak performance. For example, if an appli-

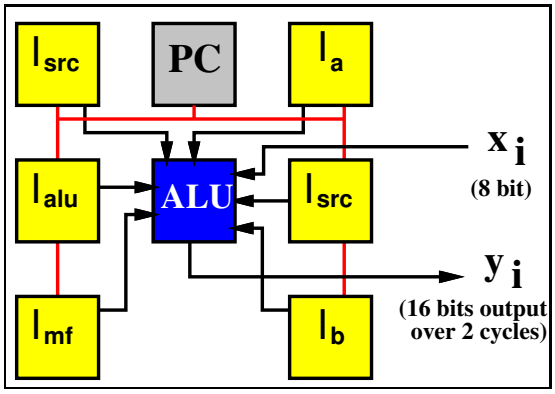

Figure 8-2: Microcoded FIR Implementation

cation required an FIR at 25MHz, the MATRIX design would yield at $\approx 0.4 \cdot \lambda^2 \cdot s$.[1] Systolic designs have no means of taking advantage of the extra time alloted.

8.3 Microcoded - Temporal FIR

8.3.1 Implementation

Figure 8-2 shows a microcoded design of an FIR filter. Rather than dedicating BFU to performing dedicated functions, one BFU is being used to perform *all* the required operations (the blue (dark grey) BFU marked "ALU"). The BFU's main memory is being used as a register file to store the coefficient weights (w_k) as well as six intermediate variables.

Six additional BFUs are used as instruction stores to hold the microcoded program. These BFUs use their main memories as 256 byte blocks, and do not use their ALUs for any computation. The purpose of each are as follows:

I_a and I_b are used to store the A and B register addresses.

I_{alu} and I_{mf} are used to store the ALU operation, and Fm Port function, respectively.

The I_{src} memories control the dynamic behavior of the A and B ports (through the FP1 and FP2 ports). This is an example of ports being used in Dynamic Source

[1] It may be worth noting that neither FPGA implementations could achieve this rate all.

Label	ALU Op	PC
newsample	Rxp $\leftarrow$ Rxp + 1 ; Match $(k+1)$ (6 bits)	BNE xpcont1
	$<$ Rxp $>$ $\leftarrow$ new x_i	(pipelined branch slot)
	Rxp $\leftarrow$ 65	
xpcont1	$<$ Rxp $>$ $\leftarrow$ new x_i	
	Rs $\leftarrow$ $<$ Rxp $>$	
	Rwp $\leftarrow$ 1	
	Rw $\leftarrow$ $<$ Rwp $>$	
	Rs $\leftarrow$ Rs $\times$ Rw	
	Rw $\leftarrow$ $\times$-continue	
	Rl $\leftarrow$ Rs; Match false	BNE enterloop
	Rh $\leftarrow$ Rw	(pipelined branch slot)
innerloop	Rs $\leftarrow$ Rs $\times$ Rw	
	Rw $\leftarrow$ $\times$-continue	
	Rl $\leftarrow$ Rs + Rl	
	Rh $\leftarrow$ Rw +-continue Rh	
enterloop	Rxp $\leftarrow$ Rxp + 1 ; Match $(k+1)$ (6 bits)	BNE xpcont2
	Rs $\leftarrow$ $<$ Rxp $>$	(pipelined branch slot)
	Rxp $\leftarrow$ 65	
	Rs $\leftarrow$ $<$ Rxp $>$	
xpcont2	Rwp $\leftarrow$ Rwp + 1 ; Match $(k+1)$ (6 bits)	BNE innerloop
	Rw $\leftarrow$ $<$ Rwp $>$	(pipelined branch slot)
last	read Rl ; Match false	BNE newsample
	read Rh	(pipelined branch slot)

Table 8.2: Microcode for FIR Computation

mode. During normal operation, the A and B ports are being used to supply the register addresses for the internal calculation. However, in order to load a new sample value, these ports must switch to loading data from an external source. This is accomplished using the dynamic port mode on the A and B ports.

Finally, a single BFU has been configured to supply the lookup address for the I-stores. This Program Counter (PC) is setup to either increment its counter, or load a new counter value from its internal memory, based on the current operational step.

Table 8.2 shows the microcode for the FIR computation. The 8 BFUs shown in Figure 8-2 produce a new 16-bit result every $8k + 9$ cycles (k is the number of filter TAPs). The result is output over two cycles. In this example, $k \leq 61$ because of the limited space in the ALU's register file memory. Larger FIRs could be supported using additional BFUs to store the extra sample and coefficient values.

Device	MATRIX	MIPS-X MSTEP	NEC VSP8 32b mpy	1996 Alpha 64b mpy
Reference		ISSCC87 [10]	[14]	ISSCC96 [9]
Size	8 BFUs	1 die	1 die	1 die
Area	$29M\lambda^2$/BFU	$68M\lambda^2$	$1.2G\lambda^2$	$6.8G\lambda^2$
Clock Rate	10 ns cycle	50 ns cycle	10 ns cycle	2.3 ns cycle
Throughput	8 cycles/TAP	10+ cycles/TAP	4 cycles/TAP	1 cycle/TAP
Density $TAPs/\lambda^2 \cdot s$	$0.054/\lambda^2 \cdot s$	$0.029/\lambda^2 \cdot s$	$0.022/\lambda^2 \cdot s$	$0.064/\lambda^2 \cdot s$

Table 8.3: Microcoded FIR Performance Density Comparison

8.3.2 Performance Density

Table 8.3 compare the performance density of the microcoded MATRIX design with three modern microprocessor architectures. Of these, the NEC VSP8 and the DEC Alpha have hardwired multipliers, while the MIPS-X has only a multiply-step operation. As we can see, MATRIX compares very well to these designs, beating out all but the Alpha.

The main reason the microprocessors performed poorly against the MATRIX design, is the fact that they tied up their entire chip performing the simple 8-bit FIR, while MATRIX was able to free whatever space remained on the die to performing other operations. This demonstrates the one of the main inefficiencies of traditional microprocessor architectures: unneeded on-chip resources are wasted when not in use. In modern microprocessors, a large amount of chip area is dedicated to caches designed to handle rapidly changing, random instruction streams. On regular applications such as FIR this area goes mostly unused. In addition the large datapath widths (64 bits on the Alpha) are unneeded in this 8-bit FIR example.[1]

8.3.3 Conclusions

Microcoded designs provide a mechanism for reusing functional blocks to perform multiple operations in time. As a result they are very useful when there is a lim-

[1]This is not an unreasonably small size - most signal processing applications require FIRs no larger than 16 bits.

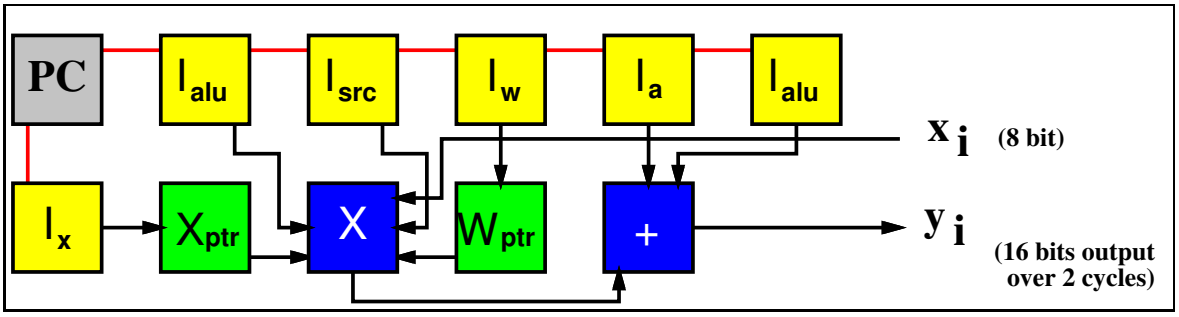

Figure 8-3: Custom VLIW FIR Implementation

ited amount of space available. In fact, microprocessor architectures were originally conceived as a method of efficiently using the very limited (at the time) silicon area available. However, modern microprocessors do not suffer this restriction, yet continue to follow the same design methodology. As a result, their performance density has suffered in comparison to other architectures.

The major drawback for microcoded designs is that due to the need to perform all operations on a single unit, every application will require a minimum amount of time to run. One possible solution to this was mentioned in Section 8.2 - create a hybrid systolic/microcoded design. In such a design, the systolic logic would handle the high required throughput portions of the application, while the microcoded logic would handle the control and infrequently needed functions. The combination of the two would be smaller than a pure systolic array, but faster than a pure microcoded design.

Another possible solution is presented in Section 8.4, below.

8.4 Custom VLIW FIR

8.4.1 Implementation

Figure 8-3 shows a custom VLIW implementation of an FIR filter. This example takes advantage of the parallism inherent in the FIR computation to construct application-specific datapaths, while maintaining a temporal computing style.

As shown in Figure 8-3, there are four BFU allocated to performing computation:

Label	Xptr unit	Wptr unit	MPY unit	+-unit
firstsample	Xptr←64	Wptr←0		
	output Xptr	output Wptr	< Xptr > ← new x_i	
nextsample	Xptr++ mod k | 64	Wptr++	< Xptr > × < Wptr >	
	output Xptr	output Wptr	×-continue	Rlow ← MPY-result
	Xptr++ mod k | 64	Wptr++	< Xptr > × < Wptr >	Rhigh ← MPY-result
	output Xptr	output Wptr	×-continue	Rlow ← Rlow + MPY-result
innerloop	Xptr++ mod k | 64	Wptr++; Match k	< Xptr > × < Wptr >	Rhigh ← Rhigh + MPY-result
	output Xptr	output Wptr	×-continue	Rlow ← Rlow + MPY-result
last	output Xptr	output Wptr	< Xptr > × < Wptr >	Rhigh ← Rhigh + MPY-result
	Xptr++ mod k | 64	Wptr←0; Match false	×-continue	**Rlow** ← Rlow + MPY-result
	output Xptr	output Wptr	< Xptr > ← new x_i	**Rhigh** ← Rhigh + MPY-result

Table 8.4: VLIW Microcode for FIR Computation

one each for the multiply and add operation, one to manipulate the sample pointer (Xptr), and one to manipulate the coeffient pointer (Wptr). There are also six BFUs allocated as instruction stores, and one BFU for serve as a program counter. This arrangement makes it possible to reduce the inner loop of the FIR computation to two steps, as shown in Table 8.4. The boxed values in last column are the pair of y_i output bytes at the end of each convolution.

As shown in Figure 8-3, this implementation requires 11 BFUs and produces a new 16-bit result every $2k + 1$ cycles. As in the microcoded example the result is output over two cycles on the ALU output bus. The number of weights supported is limited to $k \leq 64$ by the space in the ALU's memory.

Most of the I-stores used in this design contain only a few instructions. With clever use of the control PLA and configuration words, the number of I-stores can be cut in half making this implementation no larger than the microcoded implementation, while still being four times faster.

8.4.2 Performance Density

Table 8.5 compares the performance density of the MATRIX VLIW FIR implementation with a modern DSP chip. The DSP uses a similar VLIW approach to performing FIR computations. In this case, the DSP's slower clock rate and larger area gave it a significantly lower performance density. While DSPs have tailored their datapath to performing signal processing operations, they include many more hardwired functional units, most of which are not needed for a given application.

Design	MATRIX	Toshiba 16b DSP
Reference		CICC92 [17]
Size	11 BFUs	1 die
Area	$29M\lambda^2$/BFU	$275M\lambda^2$
Clock Rate	10 ns cycle	50 ns cycle
Throughput	2 cycles/TAP	1 cycle/TAP
Density $TAPs/\lambda^2 \cdot s$	$0.16/\lambda^2 \cdot s$	$0.072/\lambda^2 \cdot s$

Table 8.5: VLIW FIR Performance Density Comparison

8.4.3 Conclusions

This example demonstrates the advantages of customizing a datapath to an individual application. The VLIW approach to a problem improves the performance of temporal designs with a usually minimal area cost.

8.5 Hybrid FIR Architectures

Microcoded and VLIW designs allow MATRIX to take advantage of a lower desired throughput to reduce the chip area required for the computation. This saved area could be used to perform other computations, or could be used to perform the same computation in parallel. For example, a MATRIX chip with 64 BFUs could, theoretically, perform 8 microcoded, or 5 VLIW, FIR computations in parallel simply by dedicating a separate microcoded design to each FIR.

If each FIR computation is running at a different rate, or in different time-steps, this is the best that can be done. However, if the FIRs can be run in lock-step, a drastic improvement can be made. Figure 8-4 shows a Multiple-SIMD/VLIW hybrid FIR Implementation. A single VLIW control structure, running the code shown in Table 8.4, controls 6 parallel FIR computations. The whole structure requires 21 BFUs which is one-third the size of 6 independent VLIW designs.

Many other hybrids are possible, depending on the flexibility and requirements of specific applications. Hardware optimizations like these are only possible on MATRIX-like architectures which allow users to *completely* define the computing

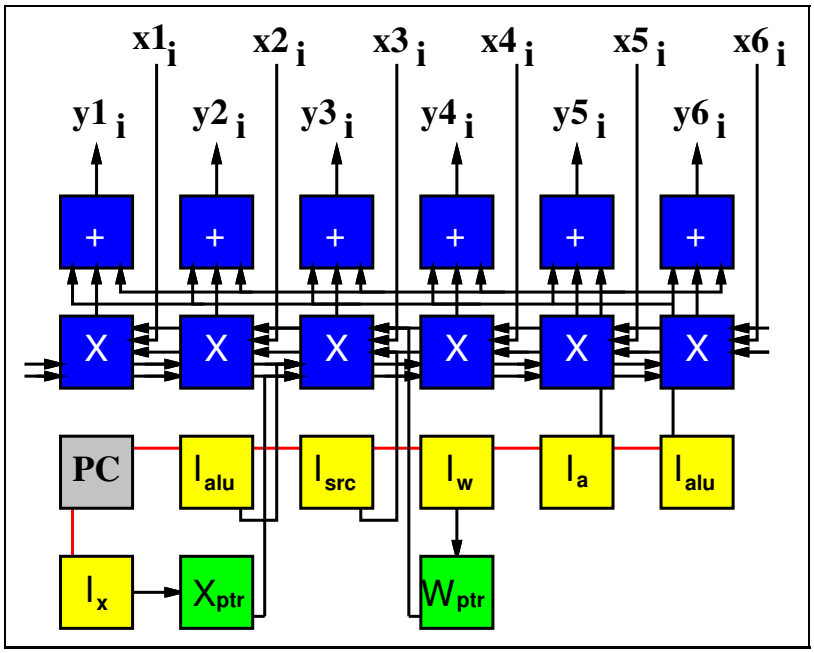

Figure 8-4: VLIW/MSIMD Hybrid FIR Implementation

structure that ideally suits the problem.

8.6 Summary

Table 8.6 shows the performance density results for the FIR example running on several different architectures. The "XC4K" is a Xilinx 4000-series FPGA. A CLB is a 4→1 combinational logic block, the basic unit of Xilinx FPGAs. PADDI2 is an experimental MIMD device with 16-bit execution units (EXUs). Two fully custom FIR chips have been included for comparison.

As we can see, MATRIX designs are comparatively dense, or even better, than similar architectures. In addition, as we have seen, MATRIX has the ability to change its design to match application requirements and flexibility, giving it a robust performance density across a wide range of applications.

Architecture	Reference	Area and Time	Filter TAPs $\frac{}{\lambda^2 \cdot \mathbf{s}}$
16b DSP	ISSCC86 [21]	125 ns/TAP	0.090
	CICC92 [17]	50 ns/TAP	0.072
32b RISC MSTEP	MIPS-X [10]	50+ ns/TAP	0.029
32b RISC/DSP	VSP8 [14]	40 ns/TAP	0.022
64b RISC	1996 Alpha [9]	2.3 ns/TAP	0.064
systolic		2 BFUs, 20ns/TAP	0.87
MATRIX microcode		8 BFUs, 80ns/TAP	0.054
VLIW		11 BFUs, 20ns/TAP	0.16
XC4K	App. Note [8]	64 CLBs, 184 ns/16-TAPs†	1.0
	ICSPAT93 [4]	400 CLBs, 100ns/4-TAPs	0.080
PADDI2	ISSCC95 [22]	5 EXUs, 20ns/TAP	0.93
Full Custom	JSSC89 [16]	45ns/64-TAPs‡	6.1
	JSSC90 [7]	33ns/16-TAPs	3.5

† – symmetric filter; ‡ – 24-bit accum.

Table 8.6: FIR Survey - 8×8 multiply, 16-bit Accumulate

Chapter 9

Relationship to Conventional Computing Devices

As we have seen, MATRIX is capable of changing its architectural structure in order to match application needs. This makes the task of comparing it to other conventional architectures which cannot change their structure difficult. Table 2.1 classified conventional architectures by the instruction/control allocation choices they made (Chapter 2). This chapter will examine these architectures and compare the choices they made with MATRIX implementations of those architectures.

9.1 Systolic Architectures

As discussed in Chapter 2, systolic architecture compute spatially, and therefore do not have any control threads. Table 2.1 list three different kinds of systolic architectures:

Hardwired Functional Units are included here as a special case since they are not general-purpose architectures. Hardwired units fix all of their functionality choices at fabrication time, and are not programmable.

FPGAs are fine-grain programmable systolic arrays. Due to their granularity, they typically have a large number of basic units on a die (large n). Due to its

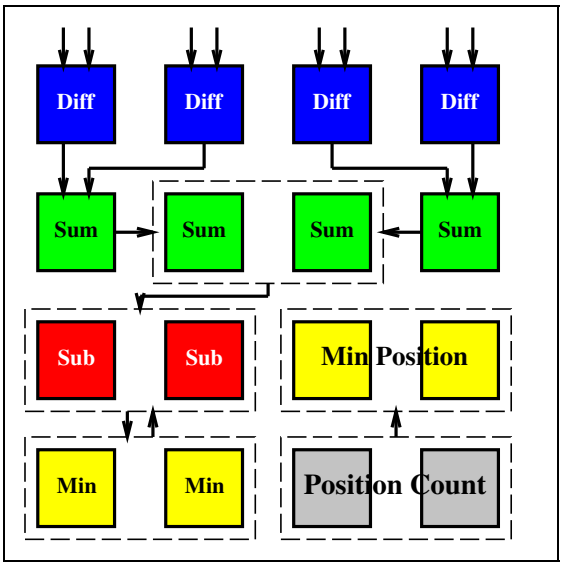

Figure 9-1: Best Match Detector - Systolic Array

basic 8-bit granularity MATRIX cannot implement these architectures, but will generally perform much better against FPGAs implementations of coarser-grain computations.

Reconfigurable ALUs are coarse-grain programmable systolic arrays. At first glance, a MATRIX array looks a great deal like these devices, especially when programmed to perform systolic operations. Figure 9-1 shows an example of a MATRIX array acting as an array of reconfigurable ALUs. The computation shown is a best-match detector used in video compression applications.

While MATRIX can act like an array of reconfigurable ALUs (at a multiple-of-8 granularity), it is not a systolic array because of its ability to temporally reuse its resources. As we saw in Chapter 8, this ability allows MATRIX to take advantage of application flexibility in ways traditional systolic arrays cannot.

9.2 Traditional and SIMD Processors

Moving down on Table 2.1, we begin to see devices that temporally reuse their resources. The first three types of these are traditional microprocessors and SIMD

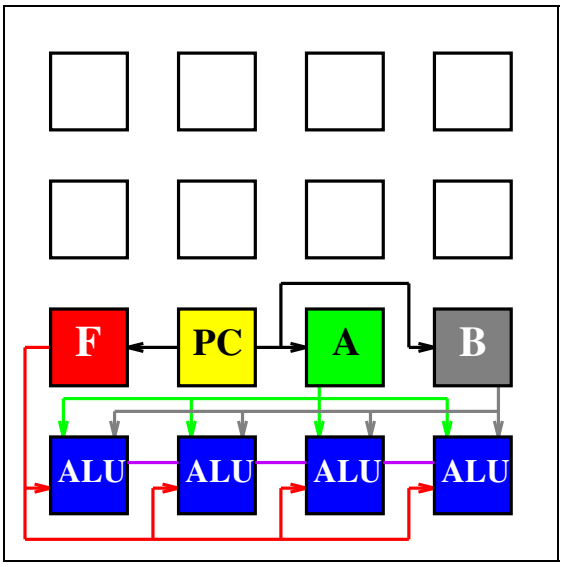

Figure 9-2: 32 Bit Microprocessor

(Single Instruction, Multiple Data) or Vector (essentially wide-datapath SIMD) processors. These three are similar in that they each have only one instruction stream on the device.

Traditional processors use a single, usually very coarse grain ($w = 32$ or 64 bit) ALU. Figure 9-2 shows an example of a (simple) traditional microprocessor architecture implemented on MATRIX. The MATRIX implementation composes the 32-bit datapath from 4 BFUs by creating a carry-chain. Three other BFUs store the 24-bit instruction ($[A \text{ op } B \rightarrow A]$ style operations), and one BFU serves as the program counter.

In a similar manner, MATRIX can implement a SIMD or Vector machine. A single program counter and set of instruction stores can control any number of processors, as shown in Figure 9-3. MATRIX cannot emulate a bitwise SIMD array due to the 8-bit granularity of the BFU, but it can implement a SIMD or Vector machine built from multiple-of-8 datapaths. Datapaths are assembled by composing BFUs through carry-chains. We saw an application-specific example of this processing style in Chapter 8.

A number of other architectures have been proposed or built which can also adjust

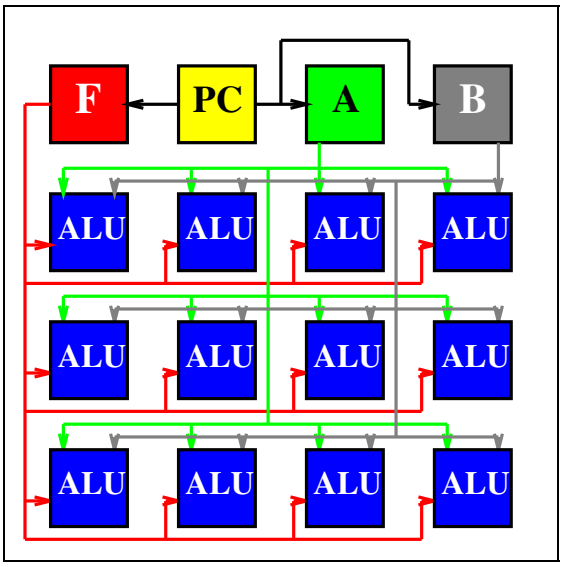

Figure 9-3: SIMD System

to different datapath granularity. Typically, this is accomplished through segmentable datapaths (*e.g.* [19] [1]). These generally exhibit SIMD instruction control for the datapath, but can be reconfigured to treat the n bit datapath as k, $\frac{n}{k}$-bit words, for certain, restricted, values of k. Modern multimedia processors (*e.g.* [18] [6]) allow the datapath to be treated as a collection of 8, 16, 32, or 64 bit words.

All of these architectures give users the ability to choose an appropriate granularity for their computation. However, they all control these datapaths in a SIMD manner. MATRIX allows not only flexible data-widths, but flexible control, as we will see in the following sections.

9.3 Multi-Context Gate Arrays and VLIW Machines

Another group of device that have a single on-chip instruction thread are multi-context gate arrays and VLIW (Very Long Instruction Word) machines. These devices are categorized by having multiple instruction stream operating under a single thread of control.

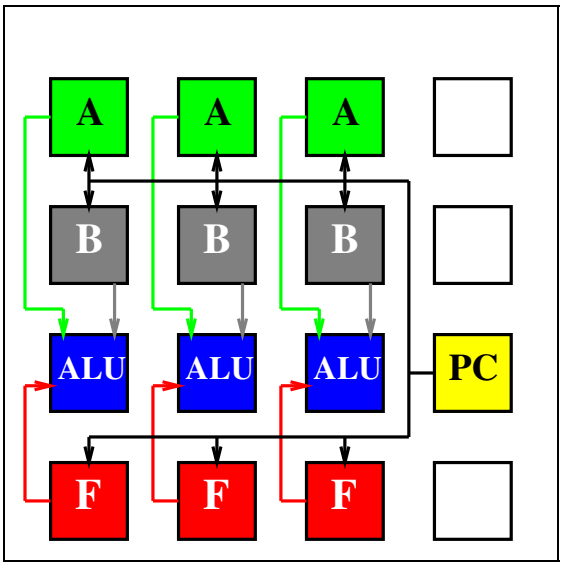

Figure 9-4: VLIW System

Multi-context gate arrays are FPGA-like devices which store multiple instruction (configurations) on-chip. Several designs have been proposed including: the DPGA [20] which provides a small number of instructions per basic look-up table (4 in the current prototype), and VEGA [11] which provides 2048 instructions.[1] These devices all fix the number instructions on-chip at fabrication time, making it hard select the "correct" size of the instruction memories. While MATRIX cannot match the fine-grain datapaths of the DPGA and VEGA, it can flexibly deploy instruction memories (in 256 instruction chunks, on the current prototype) to more closely match an application's requirements.

VLIW machines are essentially coarse-grain versions of multi-context gate arrays (or that the multi-context gate arrays are fine-grain VLIW machines). Figure 9-4 shows a generic VLIW machine implemented on MATRIX. A single program counter (PC) controls three separate instruction streams. As discussed in Chapter 2, these designs generally provide more processing power per unit area than MIMD machines, but do not have the same control flexibility.

Various architectures, such as PADDI [3] choose a granularity (16 for PADDI),

[1] VEGA actually has multiple program counters and therefore functions as a MIMD machine.

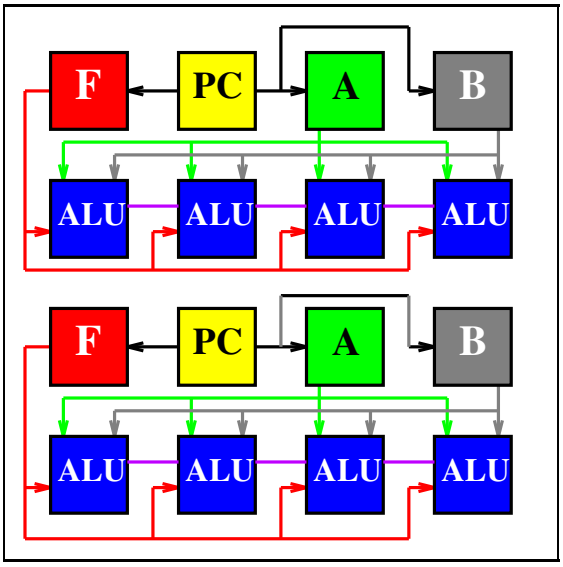

Figure 9-5: 32 Bit MIMD System

and a instruction memory size (8 for PADDI). MATRIX allows a designer to choose the VLIW architecture which best suits the application to the extent that the 8-bit BFU allows. We saw an example of this in Chapter 8.

9.4 MIMD Machines

Devices utilizing more than one program counter (control unit) per die are considered MIMD (Multiple-Instruction, Multiple-Data) machines. Figure 9-5 shows an generic 2-PC, 32-bit MIMD machine implemented on MATRIX. Just as in the VLIW case, a variety of devices, such as PADDI-2 [22], have chosen a specific data point, while MATRIX gives a designer the option of changing those choices.

9.5 Hybrid Architectures

Certainly not all applications fall into one of the traditional computing realms discussed above. In order to efficiently deal with these cases a number of architectures, including MSIMD (*e.g.* [2], [15]) have been developed. These devices allocate control units among a set of processing units. Like MATRIX, these devices can deploy control

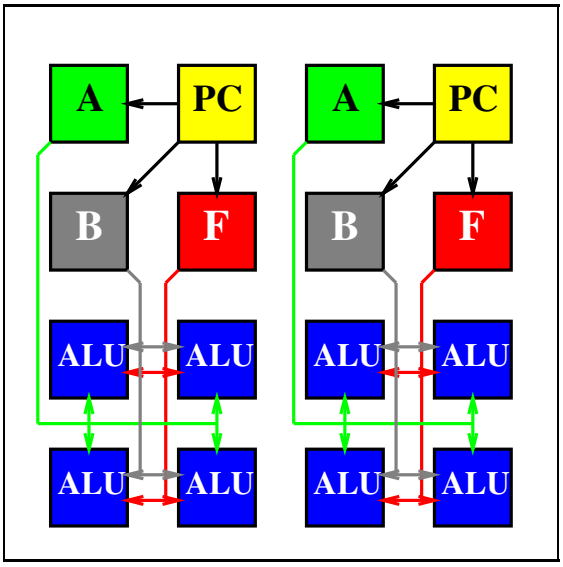

Figure 9-6: MSIMD System

units as applications require. Unlike MATRIX, the control and processing units are not the same, nor do data and control travel over the same network. This limits the structure and flexibility of any resource allocation. Figure 9-6 shows an example of one possible MSIMD design on a small MATRIX array.

Many hybrid architectures are possible on MATRIX. We saw an application-specific example of one in Chapter 8.

9.6 Summary

As we have seen in these examples, MATRIX covers nearly all the architectural possibilities listed in Table 2.1, as well as many that are not listed there. All the devices listed on the table fix their place in the taxonomy at fabrication time. MATRIX, on the other hand, can use its meta-configurability to implement nearly any of those structures, and therefore cannot be fit into that classification.

Chapter 10

Conclusions

10.1 Results

The MATRIX prototype demonstrates the possibilities for meta-configurable architectures. These include:

- **High Performance** – The prototype architecture can support designs that achieve a similar performance density to conventional commercial and academic devices with similar computing styles.
- **Flexibility** – The MATRIX architecture can implement nearly any kind of traditional architecture, while not fitting into any traditional classification.
- **Architectural Advantages** – MATRIX achieves its performance and flexibility without relying on exotic manufacturing technologies, and can therefore ride the process technology curve along with all other architectures.

On the downside, MATRIX is so different from traditional architectures that standard methods of programming do not easily apply. MATRIX allows the programmer to optimize the architecture while optimizing a program and the algorithm. This multi-dimensional space is very difficult to search.

It is always possible to implement a conventional architecture on MATRIX, then program it normally. However, this does not allow the application to take advantage of

MATRIX's meta-configurability and its performance will suffer accordingly. In order for meta-configurable architectures to come into widespread use, a new programming methodology is needed.

10.2 Future Work

Meta-configurable architectures open up a very large space of architectures and systems that has not yet been explored. The MATRIX prototype is simply a single datapoint in this space. Some avenues for future exploration include:

- **Different Granularities** – While meta-configuration does not make sense at a very small or very large granularity, there is still a wide range of basic granularities that might work better than the 8 bits chosen for MATRIX.

- **Different Internal Arrangement of the BFU** – Perhaps a more flexible BFU structure would be able to better deploy some of its resources without consuming others. For example, a different BFU structure might make the ALU available for computation even when the memory has been deployed as an I-store.

- **Different Network Structures** – Network structures that more accurately reflect the needs of applications, such as the inclusion of pipeline/retiming registers, will certainly improve the usability of these devices

- **Hybrid Architectures** – Coarse-grain blocks, such as the MATRIX BFU, are not well suited for fine-grain control logic, and the distributed PLA is a poor substitute for real fine-grain logic. Perhaps combining a MATRIX-like architecture with FPGA-like fine-grain blocks would make creating control logic much easier.

  On the other hand, traditional microprocessor-like structures are very well-suited for handing random control manipulations. Instead of FPGA-like logic, perhaps the inclusion of a small microprocessor on a MATRIX array would prove worthwhile.

- **Programming Tools** – As mentioned above, the programming methodologies required to create very high-performance designs on a meta-configurable architecture are very different from traditional methods. A whole set of tools needs to be developed so that designs can be quickly and easily mapped to meta-configurable architectures like MATRIX.

10.3 Summary

All general-purpose computing devices can perform any operation, based on their instruction stream. However, traditional general-purpose computing devices cannot adapt the way in which they handle these instruction to match the application's requirements. Because of this, these devices are efficient only on a specific set of applications. MATRIX, on the other hand, supports a meta-configuration layer which allows applications to create a computing architecture which more closely matches their requirements. This is accomplished through:

- **Parallel, Configurable Dataflow** – Datapaths can be wired up in an application-specific manner allowing data to be delivered directly to their destinations, rather than requiring special, load/store-like operations to move the data.

- **As much Dynamic Control as Needed** – Whenever an application needs to change instructions or data on a cycle-by-cycle basis, resources can be allocated to do so. On the other hand, when these things do not need to change, they can be configured so they do not consume control or network resources, freeing these resources for other needs.

- **As much Regularity as Exploitable** – Every instruction can be broadcast to any number of functional units, so that regular operations do not consume extra instruction memories and control units.

- **Deployable Resources** – Each BFU and network switch has unified instruction control, datapath, and memory resources, which can be deployed as needed

in an application. High-bandwidth applications, or parts of applications, can be composed spatially, while low-bandwidth applications can save space by composing application temporally.

These features allow MATRIX to yield high performance across a wide range of applications.

Appendix A

BFU Model

This appendix contains the Verilog code for a MATRIX BFU model. The code has been debugged and tested on a wide range of input vectors. It will be part of a complete MATRIX chip model, which will be used to run applications until the prototype chips are in.

A.1 Top Level BFU Module

```
/***************************/
/* Specifications for BFU.v */
/***************************/

/*  BFU is the model of a MATRIX BFU.  See TN130 for details on its operation
    and I/O ports.  (Hopefully all the names here will be the same as listed
    in TN130)

    The modules contained in this module are:

    BFUcore, CompReduceI, Config, Control, Fport, L1drivers, L2driver,
    L3decode, L3driver, MAdd, NAport, TSregister.

    A BFU has the following parameters:

    X, Y : The BFU's X and Y address in the array.

*/


/*****************/
/* Include Files */
```

```
/*****************/
`include "BFUcore.v"
`include "CarryDecode.v"
`include "Config.v"
`include "Control.v"
`include "Fport.v"
`include "L1drivers.v"
`include "L2driver.v"
`include "L3decode.v"
`include "L3driver.v"
`include "MAdd.v"
`include "NAport.v"

`ifdef CompReduce_defined
  `else
    `include "CompReduce.v"
`endif

`ifdef TSregister_defined
  `else
    `include "TSregister.v"
`endif


/**************/
/* Module BFU */
/**************/

module BFU(CLK, Gctx, start,
           L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE,
           L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
           CR_N1, CR_N2, CR_NE, CR_E1, CR_E2, CR_SE,
           CR_S1, CR_S2, CR_SW, CR_W1, CR_W2, CR_NW,
           Cin_N, Cin_E, Cin_S, Cin_W,
           L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
           L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4,
           L3_V1en, L3_V2en, L3_V3en, L3_V4en,
           L3_H1en, L3_H2en, L3_H3en, L3_H4en,
           L1_Nout, L1_Eout, L1_Sout, L1_Wout, CRout, Cout,
           L2_1out, L2_2out);


  /**************/
  /* Parameters */
  /**************/

  parameter X = 0, Y = 0;

  /********************/
  /* I/O Declarations */
  /********************/

  input [1:0] Gctx;
  input CLK, start;
```

```
input [7:0] L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE;
input [7:0] L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW;

input CR_N1, CR_N2, CR_NE, CR_E1, CR_E2, CR_SE;
input CR_S1, CR_S2, CR_SW, CR_W1, CR_W2, CR_NW;

input Cin_N, Cin_E, Cin_S, Cin_W;

input [7:0] L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2;

inout [7:0] L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4;

input [3:0] L3_V1en, L3_V2en, L3_V3en, L3_V4en;
input [3:0] L3_H1en, L3_H2en, L3_H3en, L3_H4en;

output [7:0] L1_Nout, L1_Eout, L1_Sout, L1_Wout, L2_1out, L2_2out;
output Cout, CRout;


/******************/
/* Internal Wires */
/******************/

/* Port Outputs */
wire [7:0] Aout, Bout, FAout, FMout, N1out, N2out, FP1out, FP2out;
wire [7:0] Aout_reg, Bout_reg, FAout_reg, FMout_reg;

/* Decoded Carries */
wire LeftCarry, RightCarry;

/* Special Input to Carry Decode */
wire AddSig;

/* BFUcore Output */
wire [7:0] BFUcore_out;

/* BFU Output */
wire [7:0] BFU_out;

/* Control Context */
wire CtrlCtx;

/* Control Outputs */
wire CtrlBit;
wire [7:0] CtrlByte;

/* L3 Enables */
wire V1en, V2en, V3en, V4en, H1en, H2en, H3en, H4en;

/* MAdd Values */
wire [7:0] MAdd1, MAdd2;

/* Config Read/Write Enable */
```

```
wire Conf_RE, Conf_WE;

/* Config Data Outputs */
wire [7:0] Main_Config, OR_Config;
wire [7:0] Config_Out, N1special;

/* Configuration Words */
wire LSB, MSB, CarryPipeline, TSenable, MAdd1source, MAdd2source;
wire [2:0] LeftSource, RightSource;
wire [8:0] Fa_a, Fa_b, Fm_a, Fm_b;
wire [9:0] A_a, A_b, B_a, B_b;
wire [9:0] N1_a, N1_b, N2_a, N2_b;
wire [8:0] FP1_a, FP1_b, FP2_a, FP2_b;
wire [3:0] L1_Enable;
wire [1:0] L2_1_Enable, L2_2_Enable;
wire [1:0] L2_1c, L2_2c;
wire [1:0] L3_V4c, L3_V3c, L3_V2c, L3_V1c;
wire [1:0] L3_H4c, L3_H3c, L3_H2c, L3_H1c;
wire [3:0] TS_A, TS_B, TS_Fa, TS_Fm, TS_FP1;
wire [3:0] TS_FP2, TS_WE, TS_CR, TS_MAdd1, TS_MAdd2;
wire [17:0] CRI_a, CRI_b;
wire [41:0] CRII;
wire [3:0] CRsel_1, CRsel_2, CRsel_3, CRsel_4;
wire CRIIsel, CtrlBitsel;


/*************************************/
/* Module Declarations and Connections */
/*************************************/

/* Configuration Block */

Config #(X, Y)
    Configuration(start, Gctx, Aout_reg, Bout_reg, Conf_WE, Conf_RE, CLK,
                  Main_Config,
                  LSB, MSB, RightSource, LeftSource, TSenable,
                  MAdd1source, MAdd2source, CarryPipeline,
                  Fa_a, Fa_b, Fm_a, Fm_b, A_a, A_b, B_a, B_b,
                  N1_a, N1_b, N2_a, N2_b, FP1_a, FP1_b, FP2_a, FP2_b,
                  L1_Enable, L2_1_Enable, L2_2_Enable,
                  L2_1c, L2_2c,
                  L3_V4c, L3_V3c, L3_V2c, L3_V1c,
                  L3_H4c, L3_H3c, L3_H2c, L3_H1c,
                  TS_A, TS_B, TS_Fa, TS_Fm, TS_FP1, TS_FP2, TS_WE, TS_CR,
                  TS_MAdd1, TS_MAdd2, CRI_a, CRI_b, CRII,
                  CRsel_1, CRsel_2, CRsel_3, CRsel_4, CRIIsel, CtrlBitsel);

/* Config Output */

Sel2 #(8) Config_sel(Main_Config,OR_Config,Config_Out,Aout_reg[6],start);
Sel2 #(8) N1_sel(N1out,Config_Out,N1special,Conf_RE,start);


/* Ports */
```

```
Fport Fa(BFUcore_out, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE,
         L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
         L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
         L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CtrlByte,
         Fa_a, Fa_b, CtrlBit, FAout, start);
Fport Fm(BFUcore_out, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE,
         L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
         L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
         L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CtrlByte,
         Fm_a, Fm_b, CtrlBit, FMout, start);

Fport FP1(BFUcore_out, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE,
          L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
          L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
          L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CtrlByte,
          FP1_a, FP1_b, CtrlBit, FP1out, start);
Fport FP2(BFUcore_out, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE,
          L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
          L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
          L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CtrlByte,
          FP2_a, FP2_b, CtrlBit, FP2out, start);

NAport A(BFUcore_out, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE,
         L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
         L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
         L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CtrlByte,
         A_a, A_b, FP1out, CtrlBit, Aout, start);
NAport B(BFUcore_out, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE,
         L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
         L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
         L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CtrlByte,
         B_a, B_b, FP2out, CtrlBit, Bout, start);

NAport N1(BFUcore_out, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE,
          L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
          L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
          L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CtrlByte,
          N1_a, N1_b, FP1out, CtrlBit, N1out, start);
NAport N2(BFUcore_out, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE,
          L1_S1, L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
          L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
          L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CtrlByte,
          N2_a, N2_b, FP2out, CtrlBit, N2out, start);


/* Port Registers */

TSregister #(8) FAreg(FAout, L3_H4[3:0], FAout_reg, TSenable, TS_Fa,
                      CLK, start);
TSregister #(8) FMreg(FMout, L3_H4[3:0], FMout_reg, TSenable, TS_Fm,
                      CLK, start);

TSregister #(8) Areg(Aout, L3_H4[3:0], Aout_reg, TSenable, TS_A,
```

```
                    CLK, start);
TSregister #(8) Breg(Bout, L3_H4[3:0], Bout_reg, TSenable, TS_B,
                    CLK, start);


/* Control Stuff */

CompReduceI #(9) CRI({Cout,BFUcore_out}, CRI_a, CRI_b, CtrlCtx,
                    CRout, start);

Control Ctrl({CRout,CR_N1,CR_N2,CR_NE,CR_E1,CR_E2,CR_SE,
              CR_S1, CR_S2, CR_SW, CR_W1, CR_W2, CR_NW},
             FP1out, FP2out, CtrlBit, CtrlByte, CLK, Gctx,
             Aout_reg, Bout_reg, Conf_WE, Conf_RE, start,
             OR_Config,
             CRII, CRIIsel, CRsel_1, CRsel_2, CRsel_3, CRsel_4, CtrlBitsel,
             TSenable, L3_H4[3:0], TS_CR, TS_FP1, TS_FP2);


/* Network Drivers (and Decoder) */

L3decode #(X,Y) L3decoder(L3_V1en, L3_V2en, L3_V3en, L3_V4en,
                          L3_H1en, L3_H2en, L3_H3en, L3_H4en,
                          V1en, V2en, V3en, V4en, H1en, H2en, H3en, H4en,
                          start);


L1drivers L1out(BFUcore_out, L1_Enable, start,
                L1_Nout, L1_Eout, L1_Sout, L1_Wout);

L2driver L2_1(N1out, N2out, FP1out, FP2out, L2_1out, L2_1c,
              L2_1_Enable, CLK, start);
L2driver L2_2(N1out, N2out, FP1out, FP2out, L2_2out, L2_2c,
              L2_2_Enable, CLK, start);

L3driver V1(N1out, N2out, FP1out, FP2out, L3_V1, L3_V1c,
            V1en, CLK, start);
L3driver V2(N1out, N2out, FP1out, FP2out, L3_V2, L3_V2c,
            V2en, CLK, start);
L3driver V3(N1special, N2out, FP1out, FP2out, L3_V3, L3_V3c,
            V3en, CLK, start);
L3driver V4(N1out, N2out, FP1out, FP2out, L3_V4, L3_V4c,
            V4en, CLK, start);
L3driver H1(N1out, N2out, FP1out, FP2out, L3_H1, L3_H1c,
            H1en, CLK, start);
L3driver H2(N1out, N2out, FP1out, FP2out, L3_H2, L3_H2c,
            H2en, CLK, start);
L3driver H3(N1out, N2out, FP1out, FP2out, L3_H3, L3_H3c,
            H3en, CLK, start);
L3driver H4(N1out, N2out, FP1out, FP2out, L3_H4, L3_H4c,
            H4en, CLK, start);


/* MAdd Cell */
```

```
  MAdd MAdd_decode(L1_N1, L1_NW, FP1out, FP2out, MAdd1source, MAdd2source,
                   TSenable, L3_H4[3:0], TS_MAdd1, TS_MAdd2, CLK,
                   start, MAdd1, MAdd2);


  /* Carry Decoder */
  CarryDecode Cdecode(CLK, CarryPipeline, LeftSource, RightSource, Cin_N,
                      Cin_E, Cin_S, Cin_W, Cout, CtrlBit, LeftCarry,
                      RightCarry, AddSig, start);

  /* BFU Core */

  BFUcore Core(Aout_reg, Bout_reg, FAout_reg, FMout_reg, LeftCarry,
               RightCarry, CLK, MAdd1, MAdd2, BFUcore_out, Cout, LSB, MSB,
               TSenable, TS_WE, L3_H4[3:0], CtrlCtx, Conf_WE, Conf_RE,
               AddSig, start);

endmodule
```

A.2 Main BFU Modules

```
/***********************************/
/* Specifications for CarryDecode.v */
/***********************************/

/* This module handes the selection of the carry-in for the ALU.  It takes in
   the local configuration information (CarryPipeline, LeftSource,
   RightSource) and the six possible sources (N,E,S,W,Local, and Control) and
   selects the appropriate one (or a constant) for Cin_left and Cin_right.

   In addition, this module impliments logic that, on an add operation using
   a local carry (right), the pipeline is always used.
*/

/*****************/
/* Include Files */
/*****************/

`ifdef selector_defined
  `else
    `include "Selector.v"
`endif


/**********************/
/* CarryDecode Module */
/**********************/

module CarryDecode (CLK, CarryPipeline, LeftSource, RightSource, Cin_N,
                    Cin_E, Cin_S, Cin_W, Cin_Local, CtrlBit, Cin_left,
                    Cin_right, AddSig, start);

  input [2:0] LeftSource, RightSource;
  input Cin_N, Cin_E, Cin_S, Cin_W, Cin_Local, CtrlBit;
  input CLK, CarryPipeline, AddSig, start;

  output Cin_left, Cin_right;


  /**********************/
  /* Internal Registers */
  /**********************/

  reg Left, Right;  /* Selected Carries */
  reg Left_reg, Right_reg;  /* Registered Carries */

  /******************/
  /* Begin Decoding */
  /******************/

  always @(LeftSource or start)
```

```
  begin
    case (LeftSource)
      2'd0:  /* North */
          assign Left = Cin_S;
      2'd1:  /* East */
          assign Left = Cin_W;
      2'd2:  /* South */
          assign Left = Cin_N;
      2'd3:  /* West */
          assign Left = Cin_E;
      2'd4:  /* Local */
          assign Left = Cin_Local;
      2'd5:  /* Control Bit */
          assign Left = CtrlBit;
      2'd6:  /* Constant Zero */
          assign Left = 1'b0;
      2'd7:  /* Constant One */
          assign Left = 1'b1;
      default
          assign Left = 1'bz;
    endcase
  end /* Left Decoding */

always @(RightSource or start)
  begin
    case (RightSource)
      2'd0:  /* North */
          assign Right = Cin_S;
      2'd1:  /* East */
          assign Right = Cin_W;
      2'd2:  /* South */
          assign Right = Cin_N;
      2'd3:  /* West */
          assign Right = Cin_E;
      2'd4:  /* Local */
          assign Right = Cin_Local;
      2'd5:  /* Control Bit */
          assign Right = CtrlBit;
      2'd6:  /* Constant Zero */
          assign Right = 1'b0;
      2'd7:  /* Constant One */
          assign Right = 1'b1;
      default
          assign Right = 1'bz;
    endcase
  end /* Right Decoding */


/*********************/
/* Output Assignment */
/*********************/

always @(posedge(CLK) or start)
  begin
```

```
      Left_reg = Left;
      Right_reg = Right;
    end

  Sel2 #(1) LeftSel(Left, Left_reg, Cin_left, CarryPipeline, start);
  Sel2 #(1) RightSel(Right, Right_reg, Cin_right,
                     (CarryPipeline || ((RightSource === 4'd4) &&
                                        (AddSig === 1'b1))),
                     start);


endmodule
```

```
/*******************************/
/* Specifications for Config.v */
/*******************************/

/*  The Config module contains two configuration memories, and the logic to
    deocde those memories into the actual configuration words (listed below).
    Also in this module are the constant configurations.

    This module does not handle the OR plane.

    Note that the high bit of ProgAdd selects between the two programmable
    contexts for programming.

    The Hardwired Contexts:

    Ctx0: Write Context
         L3_V1 : <WEmain (1 bit), 5'b0, WEconfig (1 bit), 1'b0>
                   Fed into BOTH function ports.
         L3_V2 : <2'b0, RowAddress (3 bits), ColAddress (3 bits)>
                   Decoded intro a control bit by C/R II, through FP1.
         L3_V3 : Memory Address (A port)
         L3_V4 : Data (B port)

         In this context, the L3_H lines would be driven with their
         matching L3_V line (ie: L3_H1 = L3_V1), on a one-cycle delay.
         On a real chip, these lines would be driven in at least one row,
         so that the north-side inputs (programming) are available to the
         other sides.


    Ctx1: Read Context
    Uses L3_V1 as the memory address, and
         L3_V2 as the main memory/config memory selection (Fm Port).

         L3_V3 is the output line.
         Row selection is performed by the perimeter L3 controllers.

    Inputs:
    Gctx : Global Context Select (2 bits)
    ProgAdd:  Programming Address (8 bits)
    ProgDara: Programming Data (8 bits)
    PWE: Programming Write Enable (1 bit)
    PRE: Programming Read Enable (1 bit)
    CLK: Global Write Clock (1 bit)
    start : The model-specific initalization forces this module to re-read its
       files.

    Outputs:

    MSB, LSB, CarryPipeline (1 bit each) :  BFUcore configuration
    TS_Enable, MAdd1source, MAdd2source : BFUcore configuration
    RightSource, LeftSource (3 bits each) : BFUcore configuation

    Fa_a, Fa_b (9 bits each) : ALU Function Port
```

```
    Fm_a, Fm_b (9 bits each) : MEM Function Port

    A_a, A_b (10 bits each) : A Address Port
    B_0, B_0 (10 bits each) : B Address Port

    N1_a, N1_b (10 bits each) : Network Port 1
    N2_a, N2_b (10 bits each) : Network Port 2

    FP1_a, FP1_b (9 bits each) : Floating Port 1
    FP2_a, FP2_b (9 bits each) : Floating Port 2

    L1_Enable (4 bits): Level-1 Driver Enables
    L2_1_Enable (2 bits): Level-2 Driver 1 Enables
    L2_2_Enable (2 bits): Level-2 Driver 2 Enables

    L2_1 (2 bits): Level-2 Driver 1 Selector
    L2_2 (2 bits): Level-2 Driver 2 Selector

    L3_V4, L3_V3, L3_V2, L3_V1 (2 bits each): Level-3 Driver Selectors
    L3_H4, L3_H3, L3_H2, L3_H1 (2 bits each): Level-3 Driver Selectors

    TS_A, TS_B, TS_Fa, TS_Fm (4 bits each): Time-Switch Register Values
    TS_FP1, TS_FP2, TS_WE, TS_CR (4 bits each): Time-Switch Register Values
    TS_MAdd1, TS_MAdd2 : Time-Switch Register Values

    CRI_a, CRI_b (18 bits each) : Compare/Reduce I Configuration
    CRII (42 bits) : Compare/Reduce II Configuration

    CRsel_1, CRsel_2, CRsel_3, CRsel_4 (4 bits each): NOR Plane Input Selectors
    CRIIsel (1 bit) : Compare/Reduce II Input Selector
    CtrlBitsel (1 bit) : Control Bit Selector

*/


/*****************/
/* Module Config */
/*****************/

module Config(start, Gctx, ProgAdd, ProgData, PWE, PRE, CLK,
              DataOut,
              LSB, MSB, RightSource, LeftSource, TS_Enable,
              MAdd1source, MAdd2source, CarryPipeline,
              Fa_a, Fa_b, Fm_a, Fm_b, A_a, A_b, B_a, B_b,
              N1_a, N1_b, N2_a, N2_b, FP1_a, FP1_b, FP2_a, FP2_b,
              L1_Enable, L2_1_Enable, L2_2_Enable,
              L2_1, L2_2,
              L3_V4, L3_V3, L3_V2, L3_V1,
              L3_H4, L3_H3, L3_H2, L3_H1,
              TS_A, TS_B, TS_Fa, TS_Fm, TS_FP1, TS_FP2, TS_WE, TS_CR,
              TS_MAdd1, TS_MAdd2, CRI_a, CRI_b, CRII,
              CRsel_1, CRsel_2, CRsel_3, CRsel_4, CRIIsel, CtrlBitsel);

  /**************/
```

```
/* Parameters */
/**************/

/* X and Y position of the BFU.
   These are used for the constant configuations */
parameter X = 0;
parameter Y = 0;


/********************/
/* I/O Declarations */
/********************/

input [1:0] Gctx;
input [7:0] ProgAdd, ProgData;
input PWE, PRE, CLK, start;

output [7:0] DataOut;
reg [7:0] DataOut;

output LSB, MSB, TS_Enable, MAdd1source, MAdd2source, CarryPipeline;
output [2:0] RightSource, LeftSource;
reg LSB, MSB, TS_Enable, MAdd1source, MAdd2source, CarryPipeline;
reg [2:0] RightSource, LeftSource;

output [8:0] Fa_a, Fa_b, Fm_a, Fm_b;
reg [8:0] Fa_a, Fa_b, Fm_a, Fm_b;

output [9:0] A_a, A_b, B_a, B_b;
reg [9:0] A_a, A_b, B_a, B_b;

output [9:0] N1_a, N1_b, N2_a, N2_b;
reg [9:0] N1_a, N1_b, N2_a, N2_b;

output [8:0] FP1_a, FP1_b, FP2_a, FP2_b;
reg [8:0] FP1_a, FP1_b, FP2_a, FP2_b;

output [3:0] L1_Enable;
output [1:0] L2_1_Enable, L2_2_Enable;
reg [3:0] L1_Enable;
reg [1:0] L2_1_Enable, L2_2_Enable;

output [1:0] L2_1, L2_2;
output [1:0] L3_V4, L3_V3, L3_V2, L3_V1;
output [1:0] L3_H4, L3_H3, L3_H2, L3_H1;
reg [1:0] L2_1, L2_2;
reg [1:0] L3_V4, L3_V3, L3_V2, L3_V1;
reg [1:0] L3_H4, L3_H3, L3_H2, L3_H1;

output [3:0] TS_A, TS_B, TS_Fa, TS_Fm, TS_FP1;
output [3:0] TS_FP2, TS_WE, TS_CR, TS_MAdd1, TS_MAdd2;
reg [3:0] TS_A, TS_B, TS_Fa, TS_Fm, TS_FP1;
reg [3:0] TS_FP2, TS_WE, TS_CR, TS_MAdd1, TS_MAdd2;
```

```
output [17:0] CRI_a, CRI_b;
output [41:0] CRII;
reg [17:0] CRI_a, CRI_b;
reg [41:0] CRII;

output [3:0] CRsel_1, CRsel_2, CRsel_3, CRsel_4;
output CRIIsel, CtrlBitsel;
reg [3:0] CRsel_1, CRsel_2, CRsel_3, CRsel_4;
reg CRIIsel, CtrlBitsel;


/************************************/
/* Define the Configuration Memories */
/************************************/

reg[7:0] Ctx2[45:0];
reg[7:0] Ctx3[45:0];


/**********************/
/* Handle Programming */
/**********************/

always @(posedge(CLK) && (PWE === 1'b1))
  begin
    if (ProgAdd[6] === 1'b0)
      begin
        if (ProgAdd[7] === 1'b0)
            Ctx2[ProgAdd[5:0]] = ProgData;
        else
            Ctx3[ProgAdd[5:0]] = ProgData;
      end
  end


/******************/
/* Handle Reading */
/******************/

always @(posedge(CLK) && (PRE === 1'b1))
  begin
    if (ProgAdd[7] === 1'b0)
        DataOut = Ctx2[ProgAdd[5:0]];
    else
        DataOut = Ctx3[ProgAdd[5:0]];
  end


/**********************/
/* Temporary Register */
/**********************/
/* Note that this are required because verilog does not support
   bit-selects of memory elements */
```

```
reg [7:0] TempReg;


/*************************/
/* Handle Context Changes */
/*************************/

always @(start or Gctx or negedge(PWE))
  begin
    if (Gctx === 2'd0)
      begin
        MSB = 1'b0;
        LSB = 1'b0;
        RightSource = 3'd0;
        LeftSource = 3'd0;

        MAdd2source = 1'b0;
        MAdd1source = 1'b0;
        TS_Enable = 1'b0;
        CarryPipeline = 1'b0;

        Fa_a = {1'b0, 8'd0};
        Fa_b = {1'b1, 8'd21};
        Fm_a = {1'b0, 8'd0};
        Fm_b = {1'b1, 8'd21};

        A_a = {2'b11, 8'd23};
        A_b = {2'b11, 8'd23};
        B_a = {2'b11, 8'd24};
        B_b = {2'b11, 8'd24};

        N1_a = {2'b11, 8'd24};
        N1_b = {2'b11, 8'd24};
        N2_a = {2'b11, 8'd23};
        N2_b = {2'b11, 8'd23};

        FP1_a = {1'b1, 8'd22};
        FP1_b = {1'b1, 8'd22};
        FP2_a = {1'b0, 8'd21};
        FP2_b = {1'b0, 8'd21};

        L1_Enable = 4'b0000;

        L2_1 = 2'd0;
        L2_2 = 2'd0;

        L2_1_Enable = 2'b00;
        L2_2_Enable = 2'b00;

        L3_V1 = 2'd0;
        L3_V2 = 2'd0;
        L3_V3 = 2'd0;
        L3_V4 = 2'd0;
        L3_H1 = 2'd3;
```

```
    L3_H2 = 2'd2;
    L3_H3 = 2'd1;
    L3_H4 = 2'd0;

    TS_A = 4'd0;
    TS_B = 4'd0;
    TS_Fa = 4'd0;
    TS_Fm = 4'd0;
    TS_FP1 = 4'd0;
    TS_FP2 = 4'd0;
    TS_CR = 4'd0;
    TS_WE = 4'd0;
    TS_MAdd1 = 4'd0;
    TS_MAdd2 = 4'd0;

    CRI_a = {2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0};
    CRI_b = {2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0};

    CRII = {2'd3, 2'd3,Num2CR(Y),Num2CR(X),
            2'd3,2'd3,2'd3,2'd3,2'd3,2'd3,2'd3,2'd3,
            2'd3,2'd3,2'd3,2'd3,2'd3,2'd3};

    CRsel_1 = 4'd0;
    CRsel_2 = 4'd0;
    CRsel_3 = 4'd0;
    CRsel_4 = 4'd0;

    CtrlBitsel = 1'b0;
    CRIIsel = 1'b0;
  end

if (Gctx === 2'd1)
  begin
    MSB = 1'b0;
    LSB = 1'b0;
    RightSource = 3'd0;
    LeftSource = 3'd0;

    MAdd2source = 1'b0;
    MAdd1source = 1'b0;
    TS_Enable = 1'b0;
    CarryPipeline = 1'b0;

    Fa_a = {1'b0, 4'b0000, 4'd12};
    Fa_b = {1'b0, 4'b0000, 4'd12};
    Fm_a = {1'b0, 8'd22};
    Fm_b = {1'b0, 8'd22};

    A_a = {2'b11, 8'd21};
    A_b = {2'b11, 8'd21};
    B_a = {2'b00, 8'd30};
    B_b = {2'b00, 8'd30};

    N1_a = {2'b11, 8'd0};
```

```
    N1_b = {2'b11, 8'd0};
    N2_a = {2'b00, 8'd30};
    N2_b = {2'b00, 8'd30};

    FP1_a = {1'b0, 8'd30};
    FP1_b = {1'b0, 8'd30};
    FP2_a = {1'b0, 8'd30};
    FP2_b = {1'b0, 8'd30};

    L1_Enable = 4'b0000;

    L2_1 = 2'd0;
    L2_2 = 2'd0;

    L2_1_Enable = 2'b00;
    L2_2_Enable = 2'b00;

    L3_V1 = 2'd0;
    L3_V2 = 2'd0;
    L3_V3 = 2'd0;
    L3_V4 = 2'd0;
    L3_H1 = 2'd0;
    L3_H2 = 2'd0;
    L3_H3 = 2'd0;
    L3_H4 = 2'd0;

    TS_A = 4'd0;
    TS_B = 4'd0;
    TS_Fa = 4'd0;
    TS_Fm = 4'd0;
    TS_FP1 = 4'd0;
    TS_FP2 = 4'd0;
    TS_CR = 4'd0;
    TS_WE = 4'd0;
    TS_MAdd1 = 4'd0;
    TS_MAdd2 = 4'd0;

    CRI_a = {2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0};
    CRI_b = {2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0,2'd0};

    CRII = {2'd3,2'd3,2'd3,2'd3,2'd3,2'd3,2'd3,
            2'd3,2'd3,2'd3,2'd3,2'd3,2'd3,2'd3,
            2'd3,2'd3,2'd3,2'd3,2'd3,2'd3,2'd3};

    CRsel_1 = 4'd0;
    CRsel_2 = 4'd0;
    CRsel_3 = 4'd0;
    CRsel_4 = 4'd0;

    CtrlBitsel = 1'b0;
    CRIIsel = 1'b0;
  end

if (Gctx === 2'd2)
```

```
begin
  TempReg = Ctx2[0];
  MSB = TempReg[7];
  LSB = TempReg[6];
  RightSource = TempReg[5:3];
  LeftSource = TempReg[2:0];

  TempReg = Ctx2[1];
  MAdd2source = TempReg[3];
  MAdd1source = TempReg[2];
  TS_Enable = TempReg[1];
  CarryPipeline = TempReg[0];

  TempReg = Ctx2[6];
  Fa_a = {TempReg[0],Ctx2[2]};
  Fa_b = {TempReg[1], Ctx2[3]};
  Fm_a = {TempReg[2], Ctx2[4]};
  Fm_b = {TempReg[3], Ctx2[5]};

  TempReg = Ctx2[11];
  A_a = {TempReg[1:0], Ctx2[7]};
  A_b = {TempReg[3:2], Ctx2[8]};
  B_a = {TempReg[5:4], Ctx2[9]};
  B_b = {TempReg[7:6], Ctx2[10]};

  TempReg = Ctx2[16];
  N1_a = {TempReg[1:0], Ctx2[12]};
  N1_b = {TempReg[3:2], Ctx2[13]};
  N2_a = {TempReg[5:4], Ctx2[14]};
  N2_b = {TempReg[7:6], Ctx2[15]};

  TempReg = Ctx2[21];
  FP1_a = {TempReg[0], Ctx2[17]};
  FP1_b = {TempReg[1], Ctx2[18]};
  FP2_a = {TempReg[2], Ctx2[19]};
  FP2_b = {TempReg[3], Ctx2[20]};

  TempReg = Ctx2[22];
  L1_Enable = TempReg[3:0];

  TempReg = Ctx2[23];
  L2_1 = TempReg[1:0];
  L2_2 = TempReg[3:2];

  TempReg = Ctx2[24];
  L2_1_Enable = TempReg[1:0];
  L2_2_Enable = TempReg[3:2];

  TempReg = Ctx2[25];
  L3_V1 = TempReg[1:0];
  L3_V2 = TempReg[3:2];
  L3_V3 = TempReg[5:4];
  L3_V4 = TempReg[7:6];
```

```
    TempReg = Ctx2[26];
    L3_H1 = TempReg[1:0];
    L3_H2 = TempReg[3:2];
    L3_H3 = TempReg[5:4];
    L3_H4 = TempReg[7:6];

    TempReg = Ctx2[27];
    TS_A = TempReg[3:0];
    TS_B = TempReg[7:4];

    TempReg = Ctx2[28];
    TS_Fa = TempReg[3:0];
    TS_Fm = TempReg[7:4];

    TempReg = Ctx2[29];
    TS_FP1 = TempReg[3:0];
    TS_FP2 = TempReg[7:4];

    TempReg = Ctx2[30];
    TS_CR = TempReg[3:0];
    TS_WE = TempReg[7:4];

    TempReg = Ctx2[31];
    TS_MAdd1 = TempReg[3:0];
    TS_MAdd2 = TempReg[7:4];

    TempReg = Ctx2[36];
    CRI_a = {TempReg[1:0], Ctx2[33], Ctx2[32]};
    CRI_b = {TempReg[3:2], Ctx2[35], Ctx2[34]};

    TempReg = Ctx2[42];
    CRII = {TempReg[1:0], Ctx2[41], Ctx2[40],
            Ctx2[39], Ctx2[38], Ctx2[37]};

    TempReg = Ctx2[43];
    CRsel_1 = TempReg[3:0];
    CRsel_2 = TempReg[7:4];

    TempReg = Ctx2[44];
    CRsel_3 = TempReg[3:0];
    CRsel_4 = TempReg[7:4];

    TempReg = Ctx2[45];
    CtrlBitsel = TempReg[0];
    CRIIsel = TempReg[1];
  end

if (Gctx === 2'd3)
  begin
    TempReg = Ctx3[0];
    MSB = TempReg[7];
    LSB = TempReg[6];
    RightSource = TempReg[5:3];
    LeftSource = TempReg[2:0];
```

```
TempReg = Ctx3[1];
MAdd2source = TempReg[3];
MAdd1source = TempReg[2];
TS_Enable = TempReg[1];
CarryPipeline = TempReg[0];

TempReg = Ctx3[6];
Fa_a = {TempReg[0],Ctx3[2]};
Fa_b = {TempReg[1], Ctx3[3]};
Fm_a = {TempReg[2], Ctx3[4]};
Fm_b = {TempReg[3], Ctx3[5]};

TempReg = Ctx3[11];
A_a = {TempReg[1:0], Ctx3[7]};
A_b = {TempReg[3:2], Ctx3[8]};
B_a = {TempReg[5:4], Ctx3[9]};
B_b = {TempReg[7:6], Ctx3[10]};

TempReg = Ctx3[16];
N1_a = {TempReg[1:0], Ctx3[12]};
N1_b = {TempReg[3:2], Ctx3[13]};
N2_a = {TempReg[5:4], Ctx3[14]};
N2_b = {TempReg[7:6], Ctx3[15]};

TempReg = Ctx3[21];
FP1_a = {TempReg[0], Ctx3[17]};
FP1_b = {TempReg[1], Ctx3[18]};
FP2_a = {TempReg[2], Ctx3[19]};
FP2_b = {TempReg[3], Ctx3[20]};

TempReg = Ctx3[22];
L1_Enable = TempReg[3:0];

TempReg = Ctx3[23];
L2_1 = TempReg[1:0];
L2_2 = TempReg[3:2];

TempReg = Ctx3[24];
L2_1_Enable = TempReg[1:0];
L2_2_Enable = TempReg[3:2];

TempReg = Ctx3[25];
L3_V1 = TempReg[1:0];
L3_V2 = TempReg[3:2];
L3_V3 = TempReg[5:4];
L3_V4 = TempReg[7:6];

TempReg = Ctx3[26];
L3_H1 = TempReg[1:0];
L3_H2 = TempReg[3:2];
L3_H3 = TempReg[5:4];
L3_H4 = TempReg[7:6];
```

```
        TempReg = Ctx3[27];
        TS_A = TempReg[3:0];
        TS_B = TempReg[7:4];

        TempReg = Ctx3[28];
        TS_Fa = TempReg[3:0];
        TS_Fm = TempReg[7:4];

        TempReg = Ctx3[29];
        TS_FP1 = TempReg[3:0];
        TS_FP2 = TempReg[7:4];

        TempReg = Ctx3[30];
        TS_CR = TempReg[3:0];
        TS_WE = TempReg[7:4];

        TempReg = Ctx3[31];
        TS_MAdd1 = TempReg[3:0];
        TS_MAdd2 = TempReg[7:4];

        TempReg = Ctx3[36];
        CRI_a = {TempReg[1:0], Ctx3[33], Ctx3[32]};
        CRI_b = {TempReg[3:2], Ctx3[35], Ctx3[34]};

        TempReg = Ctx3[42];
        CRII = {TempReg[1:0], Ctx3[41], Ctx3[40],
                Ctx3[39], Ctx3[38], Ctx3[37]};

        TempReg = Ctx3[43];
        CRsel_1 = TempReg[3:0];
        CRsel_2 = TempReg[7:4];

        TempReg = Ctx3[44];
        CRsel_3 = TempReg[3:0];
        CRsel_4 = TempReg[7:4];

        TempReg = Ctx3[45];
        CtrlBitsel = TempReg[0];
        CRIIsel = TempReg[1];
      end
  end /* Context Changes */

/*******************************/
/* Number to CRconfig Converter */
/*******************************/

function [5:0] Num2CR;
  input [2:0] Value;
  reg [1:0] Out[2:0];
  integer i;
    begin
      for (i=0;i<3;i=i+1)
        begin
          if (Value[i] == 1'b0)
```

```
                Out[i]=2'b01;
            else
                Out[i]=2'b10;
          end
        Num2CR = {Out[2],Out[1],Out[0]};
      end
  endfunction

endmodule
```

```
/*******************************/
/* Specifications for Control.v */
/*******************************/

/* Control.v is an assembly of most of the control logic in a BFU cell.  It
   includes everything except Comp/Reduce I.

   This module takes in all the parameters of the NORplane as well.

   The inputs to Control are:

   CRin : The 13 neighborhood Compare/Reduce values.
   FP1  : The output of Floating Port 1.
   FP2  : The output of Floating Port 2.

   CLK   : A clock.
   start : The standard simulator reset.
   Gctx  : The global context selection.
   ProgAdd  : Programming Address (OR plane)
   ProgData : Programming Data (OR plane)
   PWE      : Programming WE (OR plane)
   PRE      : Programming RE (OR plane)

   CRIIconfig : The configuration for Comp/Reduce II.
   CRIIsel : The input selector for Comp/Reduce II.
   CRsel1, CRsel2, CRsel3, CRsel4 : Selector configurations for the NOR array.
   CrtlBitsel : The selector for the control bit.

   TSenable : Enable for Time-Switch Registers.
   TScycle : Incoming Time-Switch Cycle.
   TS_CRconf, TS_FP1conf, TS_FP2conf : Configuation for Time-Switch Registers.

   The outputs of Control are:

   CtrlBit  : The Control Bit
   CtrlByte : The Control Byte

   OR_Config : The read-out data from the OR plane configuration

*/

/*****************/
/* Include Files */
/*****************/
`include "ORplane.v"

`ifdef selector_defined
  `else
    `include "Selector.v"
`endif

`ifdef TSregister_defined
  `else
    `include "TSregister.v"
```

```
`endif

`ifdef CompReduce_defined
  `else
    `include "CompReduce.v"
`endif


/******************/
/* Module Control */
/******************/

module Control (CRin, FP1, FP2, CtrlBit, CtrlByte, CLK, Gctx,
                ProgAdd, ProgData, PWE, PRE, start,
                OR_Config,
                CRIIconfig, CRIIsel, CRsel1, CRsel2, CRsel3, CRsel4,
                CtrlBitsel, TSenable, TScycle, TS_CRconf, TS_FP1conf,
                TS_FP2conf);


  /********************/
  /* I/O Declarations */
  /********************/

  input [12:0] CRin;
  input [7:0] FP1, FP2;
  input CLK, start;
  input [1:0] Gctx;

  input [7:0] ProgAdd, ProgData;
  input PWE, PRE;

  input [41:0] CRIIconfig;
  input CRIIsel, CtrlBitsel;
  input [3:0] CRsel1, CRsel2, CRsel3, CRsel4;

  input TSenable;
  input [3:0] TScycle, TS_CRconf, TS_FP1conf, TS_FP2conf;

  output [7:0] OR_Config;

  output [7:0] CtrlByte;
  output CtrlBit;


  /******************/
  /* Internal Wires */
  /******************/

  /* The registered inputs */
  wire [12:0] CRin_reg;
  wire [7:0] FP1_reg, FP2_reg;

  /* The input to C/R II */
```

```
  wire [7:0] CRIIin;

  /* The 4 selected CR for the OR array */
  wire CR_OR1, CR_OR2, CR_OR3, CR_OR4;

  /* The Bit outputs of the OR and CRII */
  wire CRIIout, ORout;


  /************************/
  /* Time-Switch Registers */
  /************************/

  TSregister #(13) TS_CR(CRin, TScycle, CRin_reg, TSenable, TS_CRconf,
                         CLK, start);
  TSregister #(8) TS_FP1(FP1, TScycle, FP1_reg, TSenable, TS_FP1conf,
                         CLK, start);
  TSregister #(8) TS_FP2(FP2, TScycle, FP2_reg, TSenable, TS_FP2conf,
                         CLK, start);


  /*******************/
  /* Input Selectors */
  /*******************/

  Sel2 #(8) CRIIin_sel(FP1_reg, FP2_reg, CRIIin, CRIIsel, start);

  Sel16 CR_OR1_sel({4'd0,CRin_reg}, CR_OR1, CRsel1, start);
  Sel16 CR_OR2_sel({4'd0,CRin_reg}, CR_OR2, CRsel2, start);
  Sel16 CR_OR3_sel({4'd0,CRin_reg}, CR_OR3, CRsel3, start);
  Sel16 CR_OR4_sel({4'd0,CRin_reg}, CR_OR4, CRsel4, start);


  /****************/
  /* Main Modules */
  /****************/

  CompReduce #(21) CRII({CRIIin, CRin_reg}, CRIIconfig, CRIIout, start);

  ORplane ORarray({CR_OR1, CR_OR2, CR_OR3, CR_OR4, FP2_reg, FP1_reg},
                   Gctx, CLK, {ORout, CtrlByte}, ProgAdd, ProgData,
                   PWE, PRE, OR_Config, start);


  /*******************/
  /* Output Selector */
  /*******************/

  Sel2 #(1) CtrlBit_sel(CRIIout, ORout, CtrlBit, CtrlBitsel, start);

endmodule
```

```
/*********************************/
/* Specifications for NORplane.v */
/*********************************/

/*  ORplane is the model of the MATRIX control logic OR plane.  The inputs
    to this module are the the 20 main inputs to the OR plane.  Internally,
    they will each be inverted, and then both polarities are fed into the
    OR plane itself.  The configuration of the OR plane is read of a file,
    and is senstive to the global context changes.

    The inputs of ORplane:
    Data : The 20-bit wide input vector.
    Gctx : The two bit global context.
    CLK  : The global CLK.
    start : The standard simulation reset.

    ProgAdd  : Programming Address
    ProgData : Programming Data
    PWE      : Programming WE
    PRE      : Programming RE

    The outputs of ORplane is the 9-bit output vector and
    OR_Config, the output of the configuration memories during a read.

    */

/*****************/
/* Include Files */
/*****************/

`ifdef selector_defined
  `else
    `include "Selector.v"
`endif


/*******************/
/* Module ORplane */
/*******************/

module ORplane(Data, Gctx, CLK, Out, ProgAdd, ProgData, PWE, PRE,
               OR_Config, start);

  /*******************/
  /* I/O Declarations */
  /*******************/

  input [19:0] Data;
  input [1:0] Gctx;
  input [7:0] ProgAdd, ProgData;
  input CLK, PWE, PRE, start;

  output [7:0] OR_Config;
  reg [7:0] OR_Config;
```

```
output [8:0] Out;


/********************/
/* Output Registers */
/********************/

reg [8:0] OR_out;  /* Output of the OR plane */


/***********************************/
/* Define the Configuration Memory */
/***********************************/

reg[7:0] ReadMem[44:0];  /* This accepts data */
reg[39:0] ORMem[8:0];  /* This is basis for the OR array */


/****************/
/* Handle Reads */
/****************/

always @(posedge(CLK) && (PRE === 1'b1))
  begin
    OR_Config = ReadMem[ProgAdd[5:0]];
  end


/********************/
/* Program ReadMem */
/********************/

always @(posedge(CLK) && (PWE === 1'b1))
  begin
    if (ProgAdd[6] === 1'b1)
      begin
        ReadMem[ProgAdd[5:0]] = ProgData;

        /********************/
        /* Setup the ORMem */
        /********************/

        ORMem[0] = {ReadMem[4],ReadMem[3],ReadMem[2],
                    ReadMem[1],ReadMem[0]};
        ORMem[1] = {ReadMem[9],ReadMem[8],ReadMem[7],
                    ReadMem[6],ReadMem[5]};
        ORMem[2] = {ReadMem[14],ReadMem[13],ReadMem[12],
                    ReadMem[11],ReadMem[10]};
        ORMem[3] = {ReadMem[19],ReadMem[18],ReadMem[17],
                    ReadMem[16],ReadMem[15]};
        ORMem[4] = {ReadMem[24],ReadMem[23],ReadMem[22],
                    ReadMem[21],ReadMem[20]};
        ORMem[5] = {ReadMem[29],ReadMem[28],ReadMem[27],
```

```
                        ReadMem[26],ReadMem[25]};
          ORMem[6] = {ReadMem[34],ReadMem[33],ReadMem[32],
                        ReadMem[31],ReadMem[30]};
          ORMem[7] = {ReadMem[39],ReadMem[38],ReadMem[37],
                        ReadMem[36],ReadMem[35]};
          ORMem[8] = {ReadMem[44],ReadMem[43],ReadMem[42],
                        ReadMem[41],ReadMem[40]};
        end
    end


  /***********************/
  /* Define the OR Plane */
  /***********************/

  initial
    begin
      $async$or$array(ORMem,
                      {~Data[19],Data[19],~Data[18],Data[18],
                       ~Data[17],Data[17],~Data[16],Data[16],
                       ~Data[15],Data[15],~Data[14],Data[14],
                       ~Data[13],Data[13],~Data[12],Data[12],
                       ~Data[11],Data[11],~Data[10],Data[10],
                       ~Data[9],Data[9],~Data[8],Data[8],
                       ~Data[7],Data[7],~Data[6],Data[6],
                       ~Data[5],Data[5],~Data[4],Data[4],
                       ~Data[3],Data[3],~Data[2],Data[2],
                       ~Data[1],Data[1],~Data[0],Data[0]},
                      {OR_out[8],OR_out[7],OR_out[6],OR_out[5],
                       OR_out[4],OR_out[3],OR_out[2],OR_out[1],
                       OR_out[0]});
    end


  /*******************/
  /* Output Selector */
  /*******************/

  Sel2 #(9) OutSel(9'b0,OR_out,Out,Gctx[1],start);

endmodule
```

```
/* The following is necessary because this file may be read from many inlcude
   statements and should be ignored on all but the first */

`define CompReduce_defined

/*********************************/
/* Specifications for CompReduce.v */
/*********************************/

/* CompReduce module models the Comparison/Reduction operation of the MATRIX
   control logic.  It takes in a (parameterized-length) input word, and
   compares it to a configuration word, which can include "don't care" and
   "fail" bits.  If all bits pass their test, the output of the module is
   high, otherwise the output is low.  See TN130 for more details including
   the actual bit-encoding.

   The inputs to CompReduce are:

   Data : The Data input.
   Config : The configuration word.  Twice as large as the Data input.
   start : The standard simulator reset.

   The output is Match.

   There are actually two modules in this file.  The basic CompReduce is
   described above.  The second is CompReduceI, which adds a double-context
   configuration word to the basic CompReduce.
*/


/*****************/
/* Include Files */
/*****************/

`ifdef selector_defined
  `else
    `include "Selector.v"
`endif


/*********************/
/* Module CompReduce */
/*********************/

module CompReduce (Data, Config, Match, start);

  /* Define the size of the Data and Config words */
  parameter size = 1;

  input [size-1:0] Data;
  input [(2*size)-1:0] Config;
  input start;

  output Match;
```

```
  reg Match;


  /**********************/
  /* Internal Variables */
  /**********************/

  integer i;


  /***************/
  /* Begin Model */
  /***************/

  always @(Data or Config or start)
    begin

      /* Initialize Match to true, then update with comparisons to each bit */

      Match = 1'b1;

      for (i=0; i<size; i=i+1)
        begin
          if (Data[i]===1'b0)
              Match = Match && Config[i*2];
          else
              Match = Match && Config[(i*2)+1];
        end

    end
endmodule


/**********************/
/* Module CompReduceI */
/**********************/

module CompReduceI (Data, ConfigA, ConfigB, Ctrl, Match, start);

  /* Define the size of the Data and Config words */
  parameter size=1;

  input [size-1:0] Data;
  input [(2*size)-1:0] ConfigA, ConfigB;
  input Ctrl, start;

  output Match;


  /*****************/
  /* Internal Wire */
  /*****************/

  wire [(2*size)-1:0] Config;
```

```
  /*************************/
  /* Configuration Selector */
  /*************************/

  Sel2 #(size*2) CtxSel(ConfigA, ConfigB, Config, Ctrl, start);


  /************************/
  /* The CompReduce Module */
  /************************/

  CompReduce #(size) CR(Data, Config, Match, start);

endmodule
```

```
/*****************************/
/* Specifications for Fport.v */
/*****************************/

/*   An Fport is the switch used to feed data to the BFU function and floating
     ports.  It consists of a NetSwitch, and a few selectors which are used to
     configure the port, based on incoming configuration data.  No register
     appears at this level of the simuluation.  See TN130 for a block diagram
     of the port's operation.

     The inputs to an Fport (in addition to the main network inputs)

     Config_a, Config_b :  The configuration Contexts.
         Each of these is a 9-bit value.

     Ctrl   : Local Control Bit

     And, of cource:
     Input start is a model-specific initialization input, used to force the
       module to evaluate its inputs.

*/


/*****************/
/* Include Files */
/*****************/

`ifdef selector_defined
  `else
    `include "Selector.v"
`endif

`ifdef netswitch_defined
  `else
    `include "NetSwitch.v"
`endif


/****************/
/* Module Fport */
/****************/

module Fport(Local, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE, L1_S1, L1_S2,
             L1_SW, L1_W1, L1_W2, L1_NW,
             L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
             L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CByte,
             Config_a, Config_b, Ctrl, Out, start);

  input [7:0] Local, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE, L1_S1, L1_S2;
  input [7:0] L1_SW, L1_W1, L1_W2, L1_NW;
  input [7:0] L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2;
  input [7:0] L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CByte;
```

```
  input [8:0] Config_a, Config_b;
  input Ctrl;

  output [7:0] Out;

  input start;


  /******************/
  /* Internal Wires */
  /******************/

  wire [8:0] Config; /* Final configuration word */


  /*****************************/
  /* Delacare the major modules */
  /*****************************/

  NetSwitch switch(Local, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE, L1_S1,
                   L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
                   L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
                   L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4,
                   CByte,
                   Config[4:0], Config[7:0], Config[8], Out, start);

  Sel2 #(9) Ctx_Sel(Config_a, Config_b, Config, Ctrl, start);

endmodule
```

```
/*******************************/
/* Specifications for NAport.v */
/*******************************/

/*   An NAport is the switch used to feed data to the BFU L2 and L3 network
     drivers and address ports.  It consists of a NetSwitch, and a few
     selectors which are used to to configure the port, based on incoming
     configuration data.  No registers are included at this level of
     simulation.  See TN130 for a block diagram of the port's operation.

     The inputs to an NAport (in addition to the main network inputs)

     Config_a, Config_b :  The configuration Contexts.
         Each of these is a 10-bit value.

     FPout  : Alternate SourceSel (5 bits)
     Ctrl   : Local Control Bit

     And, of cource:
     Input start is a model-specific initialization input, used to force the
       module to evaluate its inputs.

*/


/*****************/
/* Include Files */
/*****************/

`ifdef selector_defined
  `else
    `include "Selector.v"
`endif

`ifdef netswitch_defined
  `else
    `include "NetSwitch.v"
`endif


/*****************/
/* Module NAport */
/*****************/

module NAport(Local, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE, L1_S1, L1_S2,
              L1_SW, L1_W1, L1_W2, L1_NW,
              L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
              L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CByte,
              Config_a, Config_b, FPout, Ctrl, Out, start);

  input [7:0] Local, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE, L1_S1, L1_S2;
  input [7:0] L1_SW, L1_W1, L1_W2, L1_NW;
  input [7:0] L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2;
  input [7:0] L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CByte;
```

```
  input [9:0] Config_a, Config_b;
  input [7:0] FPout;
  input Ctrl;

  output [7:0] Out;

  input start;


  /******************/
  /* Internal Wires */
  /******************/

  wire [4:0] SourceSel;  /* Final Source Selector */
  wire [9:0] Config; /* Final configuration word */


  /*****************************/
  /* Delacare the major modules */
  /*****************************/

  NetSwitch switch(Local, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE, L1_S1,
                   L1_S2, L1_SW, L1_W1, L1_W2, L1_NW,
                   L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
                   L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4,
                   CByte,
                   SourceSel, Config[7:0], Config[8], Out, start);

  Sel2 #(10) Ctx_sel(Config_a, Config_b, Config, Ctrl, start);

  Sel2 #(5) source_sel(Config[4:0], FPout[4:0], SourceSel, ~Config[9], start);

endmodule
```

```
/*********************************/
/* Specifications for L1drivers.v */
/*********************************/

/* L1drivers represent the drivers that enable the BFU's output
   Level 1 lines.  When enabled, the drive the BFU output along the
   appropriate wires.  When disabled, they drive the lines to ground.

   The inputs to L1drivers are:

   BFUout : The output of the BFUcore
   Enables : The 4 enable bits in the order N,E,S,W (msb->lsb)

   start : The standard simualtion reset signal

   The outputs of L1drivers:

   L1_N, L1_E, L1_S, L1_W : The appropriate Level 1 output lines.

*/

/********************/
/* Module L1drivers */
/********************/

module L1drivers (BFUout[7:0], Enables[3:0], start,
                  L1_N[7:0], L1_E[7:0], L1_S[7:0], L1_W[7:0]);

  input [7:0] BFUout;
  input [3:0] Enables;
  input start;

  output [7:0] L1_N, L1_E, L1_S, L1_W;
  reg [7:0] L1_N, L1_E, L1_S, L1_W;


  /* Handle the Enables */
  always @(Enables or start)
    begin

      if (Enables[3]===1'b1)
          assign L1_N = BFUout;
      else
          assign L1_N = 8'd0;

      if (Enables[2]===1'b1)
          assign L1_E = BFUout;
      else
          assign L1_E = 8'd0;

      if (Enables[1]===1'b1)
          assign L1_S = BFUout;
      else
          assign L1_S = 8'd0;
```

```
      if (Enables[0]===1'b1)
          assign L1_W = BFUout;
      else
          assign L1_W = 8'd0;
    end

endmodule
```

```
/*********************************/
/* Specifications for L2driver.v */
/*********************************/

/* An L2driver drives the Level-2 network lines.  When enabled it drives one
   of its input onto its output.  When disabled, it drives a zero.
   The driver also includes an optional register.

   The inputs to L2driver are:

   N1out, N2out, FP1out, FP2out : The incoming signals that can be driven.
   DRsel_A : Two-bit configuration word for the input selector.
   Enable : A two-bit word containing <driver enable, register enable>
   CLK : A clock
   start : The standard simulator reset signal

   The output of L2driver is Out, the final output of the driver.

   */


/*****************/
/* Include Files */
/*****************/

`ifdef selector_defined
  `else
    `include "Selector.v"
`endif


/*******************/
/* Module L2driver */
/*******************/

module L2driver (N1out, N2out, FP1out, FP2out, Out, DRsel,
                 Enable, CLK, start);

  input [7:0] N1out, N2out, FP1out, FP2out;
  input [1:0] DRsel, Enable;
  input CLK, start;

  output [7:0] Out;
  reg [7:0] Out;

  /*******************************/
  /* Internal Registers and Wires */
  /*******************************/

  wire [7:0] data_A, data_B;  /* Intermediate selector data */
  wire [7:0] SelData; /* The selected data */

  wire [7:0] FinalData; /* The final, selected and registered data */
```

```
  reg [7:0] pipeline; /* The optional register */


  /*************/
  /* Selectors */
  /*************/

  Sel2 #(8) Sel_A(N1out, N2out, data_A, DRsel[0], start);
  Sel2 #(8) Sel_B(FP1out, FP2out, data_B, DRsel[0], start);
  Sel2 #(8) Sel_Data(data_A, data_B, SelData, DRsel[1], start);

  Sel2 #(8) Sel_Final(SelData, pipeline, FinalData, Enable[0], start);


  /*********************************/
  /* Maintain the Pipeline Register */
  /*********************************/

  always @(posedge(CLK) or start)
    begin
      pipeline = SelData;
    end


  /**********************/
  /* The Actual Driver */
  /**********************/

  always @(Enable or start)
    begin
      if (Enable[1]===1'b1)
          assign Out = FinalData;
      else
          assign Out = 8'd0;
    end

endmodule
```

```
/*********************************/
/* Specifications for L3decode.v */
/*********************************/

/* L3decode decodes the incoming Level-3 Network enable lines and outputs
   the actual enables for the L3drivers.

   Parameters:  (X,Y) BFU address

   Inputs:

   L3_V1en : Level-3 Enable, Vertical-1 Line. (4 bits)
   L3_V2en : Level-3 Enable, Vertical-2 Line. (4 bits)
   L3_V3en : Level-3 Enable, Vertical-3 Line. (4 bits)
   L3_V4en : Level-3 Enable, Vertical-4 Line. (4 bits)
   L3_H1en : Level-3 Enable, Horizontal-1 Line. (4 bits)
   L3_H2en : Level-3 Enable, Horizontal-2 Line. (4 bits)
   L3_H3en : Level-3 Enable, Horizontal-3 Line. (4 bits)
   L3_H4en : Level-3 Enable, Horizontal-4 Line. (4 bits)

   start : a model-specific initialization input, used to force the
       module to evaluate its inputs.


   Ouputs: (one bit each)

   V1en : Enable L3_V1.
   V2en : Enable L3_V2.
   V3en : Enable L3_V3.
   V4en : Enable L3_V4.
   H1en : Enable L3_H1.
   H2en : Enable L3_H2.
   H3en : Enable L3_H3.
   H4en : Enable L3_H4.

*/


/*******************/
/* Module L3decode */
/*******************/

module L3decode(L3_V1en, L3_V2en, L3_V3en, L3_V4en,
                L3_H1en, L3_H2en, L3_H3en, L3_H4en,
                V1en, V2en, V3en, V4en, H1en, H2en, H3en, H4en,
                start);


  /* Default Parameters */
  parameter X=0;
  parameter Y=0;

  /* I/O specifications */
  input [3:0] L3_V1en, L3_V2en, L3_V3en, L3_V4en;
```

```
input [3:0] L3_H1en, L3_H2en, L3_H3en, L3_H4en;

input start;

output V1en, V2en, V3en, V4en, H1en, H2en, H3en, H4en;
reg V1en, V2en, V3en, V4en, H1en, H2en, H3en, H4en;


/******************/
/* Begin Decoding */
/******************/

always @(start or L3_V1en)   /* Vertical 1 */
  begin
    if (L3_V1en===Y)
        V1en = 1'b1;
    else
        V1en = 1'b0;
  end

always @(start or L3_V2en)   /* Vertical 2 */
  begin
    if (L3_V2en===Y)
        V2en = 1'b1;
    else
        V2en = 1'b0;
  end

always @(start or L3_V3en)   /* Vertical 3 */
  begin
    if (L3_V3en===Y)
        V3en = 1'b1;
    else
        V3en = 1'b0;
  end

always @(start or L3_V4en)   /* Vertical 4 */
  begin
    if (L3_V4en===Y)
        V4en = 1'b1;
    else
        V4en = 1'b0;
  end

always @(start or L3_H1en)   /* Horizontal 1 */
  begin
    if (L3_H1en===X)
        H1en = 1'b1;
    else
        H1en = 1'b0;
  end

always @(start or L3_H2en)   /* Horizontal 2 */
  begin
```

```
      if (L3_H2en===X)
          H2en = 1'b1;
      else
          H2en = 1'b0;
    end

  always @(start or L3_H3en)   /* Horizontal 3 */
    begin
      if (L3_H3en===X)
          H3en = 1'b1;
      else
          H3en = 1'b0;
    end

  always @(start or L3_H4en)   /* Horizontal 4 */
    begin
      if (L3_H4en===X)
          H4en = 1'b1;
      else
          H4en = 1'b0;
    end

endmodule
```

```
/*********************************/
/* Specifications for L3driver.v */
/*********************************/

/* An L3driver drives the Level-3 network lines.  It is a true tristate
   driver. The driver also includes a register.

   The inputs to L2driver are:

   N1out, N2out, FP1out, FP2out : The incoming signals that can be driven.
   DRsel : Two-bit configuration words for the input selector.
   Enable : The one-bit driver enable
   CLK : A clock
   start : The standard simulator reset signal

   The output of L3driver is Out, the final output of the driver.

   */


/*****************/
/* Include Files */
/*****************/

`ifdef selector_defined
  `else
    `include "Selector.v"
`endif

`ifdef trireg_defined
  `else
    `include "Trireg.v"
`endif


/*******************/
/* Module L3driver */
/*******************/

module L3driver (N1out, N2out, FP1out, FP2out, Out, DRsel,
                 Enable, CLK, start);

  input [7:0] N1out, N2out, FP1out, FP2out;
  input [1:0] DRsel;
  input Enable;
  input CLK, start;

  output [7:0] Out;


  /*******************************/
  /* Internal Registers and Wires */
  /*******************************/
```

```
  wire [7:0] data_A, data_B;  /* Intermediate selector data */
  wire [7:0] SelData; /* The selected data */


  /*************/
  /* Selectors */
  /*************/

  Sel2 #(8) Sel_A(N1out, N2out, data_A, DRsel[0], start);
  Sel2 #(8) Sel_B(FP1out, FP2out, data_B, DRsel[0], start);
  Sel2 #(8) Sel_Data(data_A, data_B, SelData, DRsel[1], start);


  /*******************/
  /* Tristate Driver */
  /*******************/

  Trireg #(8) Driver(SelData, Enable, CLK, start, Out);

endmodule
```

```
/*****************************/
/* Specifications for MAdd.v */
/*****************************/

/* MAdd is the decoder for the Multiplier-Add inputs to a BFU.  See TN130 for
   complete description of this part of MATRIX.

   Inputs to MAdd:

   HW1, HW2 : Hardwired input for MAdd1 and MAdd2.  For the current revision
              these are assume to be L1_N1, and L1_NW, respectively.
   FP1, FP2 : Outputs of the Floating Port.
   Source1, Source2 : Source selector configuration.
   TSenable : Enable Time-Switching
   TScycle  : Current Time-Switch Cycle
   TS_MAdd1, TS_MAdd2 : Time-Switch Configutation.
   CLK : A clock
   start : The simulator reset.


   Outputs of MAdd:
   MAdd1, MAdd2 : The final MAdd values.

*/


/*****************/
/* Include Files */
/*****************/
`ifdef selector_defined
  `else
    `include "Selector.v"
`endif

`ifdef TSregister_defined
  `else
    `include "TSregister.v"
`endif


/***************/
/* Module MAdd */
/***************/

module MAdd(HW1, HW2, FP1, FP2, Source1, Source2, TSenable, TScycle,
            TS_MAdd1, TS_MAdd2, CLK, start, MAdd1, MAdd2);

  input [7:0] HW1, HW2, FP1, FP2;
  input Source1, Source2, CLK, start;
  input TSenable;
  input [3:0] TScycle, TS_MAdd1, TS_MAdd2;

  output [7:0] MAdd1, MAdd2;
```

```
  /*******************************/
  /* Internal Wires and Registers */
  /*******************************/

  reg [7:0] HW2reg;

  wire [7:0] MAdd1sel_out, MAdd2sel_out;


  /*************/
  /* Selectors */
  /*************/

  Sel2 #(8) MAdd1sel(HW1, FP1, MAdd1sel_out, Source1, start);
  Sel2 #(8) MAdd2sel(HW2reg, FP2, MAdd2sel_out, Source2, start);


  /************************/
  /* Time-Switch Registers */
  /************************/

  TSregister #(8) TSreg1(MAdd1sel_out, TScycle, MAdd1, TSenable, TS_MAdd1,
                         CLK, start);
  TSregister #(8) TSreg2(MAdd2sel_out, TScycle, MAdd2, TSenable, TS_MAdd2,
                         CLK, start);



  /*****************************/
  /* Maintain Internal Register */
  /*****************************/

  always @(posedge(CLK) or start)
    begin
      HW2reg = HW2;
    end

endmodule
```

A.3 BFUcore Modules

```
x
/*******************************/
/* Specifications for BFUcore.v */
/*******************************/

/*  A BFU core is the assembly of a main MATRIX memory and ALU.  It does not
    contain any of the network port/switches or control logic.

    The modules included in a BFUcore are:
    ALU, ALUdecode, MEM, MEMdecode, CarryDecode, WEdecode, and Selector

    The inputs to BFUcore are:
    A,B    : 8-Bit Address/Data Ports
    Fa,Fm  : 8-Bit Function Ports (ALU/Memory)
    RightCarry  : Carry from LSB direction
    LeftCarry   : Carry from MSB direction
    CLK    : A Clock
    Madd1  : Multiplier-Add data 1 (special data input)
    Madd2  : Multiplier-Add data 2 (special data input)
    LSB, MSB        : Configuration Data
    TSenable    : Configuration Data
    TS_WE       : Configuration Data
    TimeStep    : Global broadcast timestep

    The Outputs of BFUcore are:
    Out   : 8-Bit output bus
    Cout  : Carry-Out
    CCS   : Control Context Select - Used by the control block outside this
            module
    AddSig : Signals an ADD op - Used by the CarryDecoder

    WEconf : Write Configuration Memory - Used outside this module
    REconf : Read Configuration Memory - Used outside this module

    Input start is a model-specific initialization input, used to force the
       module to evaluate its inputs.

*/


/*****************/
/* Include Files */
/*****************/

`include "ALU.v"
`include "ALUdecode.v"
`include "MEM.v"
`include "MEMdecode.v"
`include "TSand.v"
```

```
/* The following prevents Selector from getting re-compiled many times */
`ifdef selector_defined
  `else
    `include "Selector.v"
`endif

/******************/
/* Module BFUcore */
/******************/

module BFUcore(A[7:0], B[7:0], Fa[7:0], Fm[7:0], LeftCarry, RightCarry, CLK,
               Madd1[7:0], Madd2[7:0], Out[7:0], Cout, LSB, MSB, TSenable,
               TS_WE, TimeStep, CCS, WEconf, REconf, AddSig, start);

  input [7:0] A, B, Fa, Fm;
  input LeftCarry, RightCarry;
  input [7:0] Madd1, Madd2;
  input CLK;
  input LSB, MSB, TSenable;
  input start;
  input [3:0] TS_WE, TimeStep;

  output [7:0] Out;
  reg [7:0] Out;

  output Cout, CCS;
  output WEconf, REconf;

  output AddSig;
  reg AddSig;


  /******************/
  /* Internal wires */
  /******************/

  /* ALU I/Os */
  wire [7:0] ALU_A, ALU_B;
  wire [7:0] ALUout;

  /* These connect the ALUdecoder to the ALU (and Memory) */
  wire A_Pass, B_Pass, NAND, NOR, XOR, ShiftAR, ShiftAL, ShiftBR;
  wire ShiftBL, ADD, MULT, MULTA, MULTAA, MULTcont, InvertA, InvertB;
  wire ALU_Cin, WE;

  /* Memory I/Os */
  wire [7:0] mem_data, mem_A, mem_B;
  wire WEmem;

  /* These connect the Fm_decoder to the things it controls */
  wire Mode, Ain_sel, Bin_sel, Data_sel;


  /************************************/
```

```
  /* Declarations for the major modules */
  /************************************/

  MEM memblock(mem_data, A, B, mem_A, mem_B, Mode, WEmem, CLK);

  MEMdecode Fm_decode(Fm, Mode, Ain_sel, Bin_sel, Data_sel, WEconf, REconf);

  ALU alublock(ALU_A, ALU_B, ALU_Cin, Madd1, Madd2, ALUout, Cout,
               A_Pass, B_Pass, NAND, NOR, XOR, ShiftAR, ShiftAL, ShiftBR,
               ShiftBL, ADD, MULT, MULTA, MULTAA, MULTcont, InvertA, InvertB,
               start,CLK);

  ALUdecode Fa_decode(Fa, ALU_A, ALU_B, LeftCarry, RightCarry, LSB, MSB,
                      start, ALU_Cin, A_Pass, B_Pass, NAND, NOR, XOR,
                      ShiftAR, ShiftAL, ShiftBR, ShiftBL, ADD, MULT, MULTA,
                      MULTAA, MULTcont, InvertA, InvertB, CCS, WE);

  TSand #(1) WE_timeswitch(WE, TimeStep, WEmem, TSenable, TS_WE, start);

  /*********************************/
  /* Delcarations for the Selectors */
  /*********************************/

  Sel2 #(8) A_sel(A, mem_A, ALU_A, Ain_sel, start);
  Sel2 #(8) B_sel(B, mem_B, ALU_B, Bin_sel, start);
  Sel2 #(8) D_sel(B, ALUout, mem_data, Data_sel, start);


  /*****************************/
  /* Maintain the Output Ports */
  /*****************************/

  initial
    begin
      assign Out = ALUout;
      assign AddSig = ADD;
    end

endmodule
```

```
/************************/
/* Specifications: ALU.v */
/************************/

/* This module emulates the basic combinational ALU without control or
   I/O logic.

   This module does not include scan/reduce logic.

   Note that this model differs from expected silicon behavior as follows:
       In the real ALU, the high-byte of the multiply will be available ONLY
       on the cycle after the multiply is performed.  In this model, it
       stays around until a new multiply is performed.

   A few I/O specs:
      The default Cout is "0"

   Input start is a model-specific initialization input, used to force the
   module to evaluate its inputs.
*/

/* Modified 6 May 1996 by spon - passB corrected */


/**************/
/* ALU Module */
/**************/

module ALU (Ain[7:0], Bin[7:0], Cin, Madd1[7:0], Madd2[7:0], Out[7:0], Cout,
            A_Pass, B_Pass, NAND, NOR, XOR, ShiftAR, ShiftAL, ShiftBR,
            ShiftBL, ADD, MULT, MULTA, MULTAA, MULTcont, InvertA, InvertB,
            start, CLK);

  /* The main data inputs */
  input [7:0] Ain, Bin;
  input Cin;

  /* Input data for the multiplier adds */
  input [7:0] Madd1, Madd2;

  /* Initialization */
  input start;

  /* Clock */
  input CLK;

  /* The outputs */
  output [7:0] Out;
  output Cout;
  reg [7:0] Out;
  reg Cout;

  /* Function Select Inputs.  For this verilog model, they are assumed to
     be one-hot encoded. */
```

```
input A_Pass, B_Pass, NAND, NOR, XOR, ShiftAR, ShiftAL, ShiftBR, ShiftBL;
input ADD, MULT, MULTA, MULTAA, MULTcont;

/* Additional Control Inputs */
input InvertA, InvertB;

/* Some internal "wires" */
reg [7:0] A, B;  /* Internal (maybe inverted) A and B inputs */
reg [8:0] ADDresult; /* The adder result */
reg [15:0] MULTresult;  /* The multiplier result */
reg [7:0] TempShift;  /* An interum shift result */

/* Internal Register */

reg [7:0] MCONTreg;  /* Register for Multiply Continue */

/******************/
/* BEGIN MODELING */
/******************/


/* Maintain A and B */
always @(InvertA or start)
  begin
    if (InvertA)
        assign A = ~Ain[7:0];
    else
        assign A = Ain[7:0];
  end
always @(InvertB or start)
  begin
    if (InvertB)
        assign B = ~Bin[7:0];
    else
        assign B = Bin[7:0];
  end


/* Begin to test for, and handle, each function.  Because they are assumed
   to be one-hot, one and only one will activate at a time. */

always @(A_Pass or B_Pass or NAND or NOR or XOR or ADD or ShiftAR or
         ShiftAL or ShiftBR or ShiftBL or MULT or MULTA or MULTAA or
         MULTcont or start)
  begin
    if (A_Pass)
      begin
        assign Out = A[7:0];
        assign Cout = 1'b0;
      end
    if (B_Pass)
      begin
        assign Out = B[7:0];        /* Fixed spon 6 May 1996 */
        assign Cout = 1'b0;
```

```
  end

if (NAND)
  begin
    assign Out = ~(A[7:0] & B[7:0]);
    assign Cout = 1'b0;
  end
if (NOR)
  begin
    assign Out = ~(A[7:0] | B[7:0]);
    assign Cout = 1'b0;
  end
if (XOR)
  begin
    assign Out = (A[7:0] ^ B[7:0]);
    assign Cout = 1'b0;
  end

if (ADD)
  begin
    assign ADDresult = (A[7:0] + B[7:0] + Cin);
    assign Out = ADDresult[7:0];
    assign Cout = ADDresult[8];
  end

if (ShiftAR)
  begin
    assign TempShift = (A[7:0] >> 1);
    assign Out = {Cin, TempShift[6:0]};
    assign Cout = A[0];
  end
if (ShiftAL)
  begin
    assign TempShift = (A[7:0] << 1);
    assign Out = {TempShift[7:1], Cin};
    assign Cout = A[7];
  end
if (ShiftBR)
  begin
    assign TempShift = (B[7:0] >> 1);
    assign Out = {Cin, TempShift[6:0]};
    assign Cout = B[0];
  end
if (ShiftBL)
  begin
    assign TempShift = (B[7:0] << 1);
    assign Out = {TempShift[7:1], Cin};
    assign Cout = B[7];
  end

if (MULT)
  begin
    assign MULTresult = (A[7:0] * B[7:0]);
    assign Out = MULTresult[7:0];
```

```
          assign Cout = 1'b0;
        end
      if (MULTA)
        begin
          assign MULTresult = (A[7:0] * B[7:0]) + Madd1[7:0];
          assign Out = MULTresult[7:0];
          assign Cout = 1'b0;
        end
      if (MULTAA)
        begin
          assign MULTresult = (A[7:0]*B[7:0])+Madd1[7:0]+Madd2[7:0];
          assign Out = MULTresult[7:0];
          assign Cout = 1'b0;
        end
      if (MULTcont)
        begin
          assign MULTresult = (A[7:0]*B[7:0]);
          assign Out = MCONTreg[7:0];
          assign Cout = 1'b0;
        end
    end /* Functions */


  /* Maintain MCONTreg */
  always @(posedge(CLK) or start)
    begin
      #1;
      MCONTreg = MULTresult[15:8];
    end

endmodule
```

```
/*******************************/
/* Specifications: ALUdecode.v */
/*******************************/

/* This module represents the decoder logic for the ALU function port of a
   BFU Cell.  It takes the 8 bit function input and decodes it to the ALU
   functions.

   Verilog Code Specification:

   Input Fin[7:0] is the function port to be decoded.
   ALU_A, and ALU_B are the raw inputs to the ALU.  There are used here to
      help detemine the carry.
   Inputs CinL and CinR are the "left" and "right" carries from adjacent cells.
      Exactly which direction is determined outside this cell.
   Inputs LSB and MSB are static configuration bits which define the
      beginning and end of datapaths.

   Input start is a model-specific initialization input, used to force the
   module to evaluate its inputs.

   Output CinALU is the Cin that the ALU will actually use.

   The function outputs (A_Pass, B_Pass, NAND, NOR, XOR, ShiftAR, ShiftAL,
   ShiftBR, ShiftBL, ADD, MULT, MULTA, MULTAA, MULTcont)
   are 1-hot (one on at a time).
         Brief description of non-obvious names:
              ShiftAR: Shift input A to the right (MSB->LSB)
              ShiftAL: Shift input A to the left (MSB<-LSB)
              ShiftBR: Shift input B to the right (MSB->LSB)
              ShiftBL: Shift input B to the left (MSB<-LSB)
                  (note that all shifts use carry in and out)
              MULT:    A*B                \
              MULTA:   (A*B)+Madd1         > Low byte out here
              MULTAA:  (A*B)+Madd1+Madd2  /
              MULTcont:  Continue previous cycle multiply.  Output high byte.

   Outputs InvertA and InvertB are additional control signals for the ALU.

   Output CCS is the control context select.

   Output WE is the write enable line for the memory.  A local bit will
      determine which port's (Fa,Fm) WE is actually used.

   Outputs Latch_Madd1 and Latch_Madd2 are signals to latch the multiplier
   adds.  Madd1 is latched from the NW cell diagonal connection.
*/


/********************/
/* ALUdecode Module */
/********************/

module ALUdecode (Fin[7:0], ALU_A, ALU_B, CinL, CinR, LSB, MSB, start, CinALU,
```

```
                A_Pass, B_Pass, NAND, NOR, XOR, ShiftAR, ShiftAL,
                ShiftBR, ShiftBL, ADD, MULT, MULTA, MULTAA, MULTcont,
                InvertA, InvertB, CCS, WE);

/* Inputs */
input [7:0] Fin; /* This is the function port */
input [7:0] ALU_A, ALU_B;  /* The ALU inputs */
input CinL, CinR, LSB, MSB;  /* Carry logic */
input start; /* Initialization */

/* Output Carry */
output CinALU;
reg CinALU;

/* These are the ALU functions */
output A_Pass, B_Pass, NAND, NOR, XOR, ShiftAR, ShiftAL, ShiftBR, ShiftBL;
output ADD, MULT, MULTA, MULTAA, MULTcont;
reg A_Pass, B_Pass, NAND, NOR, XOR, ShiftAR, ShiftAL, ShiftBR, ShiftBL;
reg ADD, MULT, MULTA, MULTAA, MULTcont;

/* Additional Control signals */
output InvertA, InvertB;
reg InvertA, InvertB;

/* Control Context Select */
output CCS;
reg CCS;

/* Memory Write Enable */
output WE;
reg WE;

/****************/
/* Decode Logic */
/****************/

/* Assign the fixed bits */
initial
  begin
    assign CCS = Fin[6];
    assign WE = Fin[7];
  end

always @(Fin[5:0] or start)
  begin

    /* Start by clearing value of the one-hot outputs. */
    A_Pass = 1'b0;
    B_Pass = 1'b0;
    NAND = 1'b0;
    NOR = 1'b0;
    XOR = 1'b0;
    ShiftAR = 1'b0;
    ShiftAL = 1'b0;
```

```
ShiftBR = 1'b0;
ShiftBL = 1'b0;
ADD = 1'b0;
MULT = 1'b0;
MULTA = 1'b0;
MULTAA = 1'b0;
MULTcont = 1'b0;

assign CinALU = 1'b0;   /* Default Cin */

/* Decode the function */
case (Fin[3:0])
  4'd0:
    begin
      MULT = 1'b1;
      assign InvertA = Fin[4];
      assign InvertB = Fin[5];
    end
  4'd1:
    begin
      MULTA = 1'b1;
      assign InvertA = Fin[4];
      assign InvertB = Fin[5];
    end
  4'd2:
    begin
      MULTAA = 1'b1;
      assign InvertA = Fin[4];
      assign InvertB = Fin[5];
    end
  4'd3:
    begin
      MULTcont = 1'b1;
      assign InvertA = Fin[4];
      assign InvertB = Fin[5];
    end

  4'd4: /* Shift with Force-Carry */
    begin
      assign InvertA = 1'b0;
      assign InvertB = 1'b0;
      case(Fin[5:4])
        2'b00:
          begin
            ShiftAR = 1'b1;
            assign CinALU = CinL;
          end
        2'b10:
          begin
            ShiftBR = 1'b1;
            assign CinALU = CinL;
          end
        2'b01:
          begin
```

```
          ShiftAL = 1'b1;
          assign CinALU = CinR;
        end
      2'b11:
        begin
          ShiftBL = 1'b1;
          assign CinALU = CinR;
        end
    endcase
  end
4'd5: /* Shift with Skip-Bit */
  begin
    assign InvertA = 1'b0;
    assign InvertB = 1'b0;
    case(Fin[5:4])
      2'b00:
        begin
          ShiftAR = 1'b1;
          assign CinALU = ((MSB && ALU_A[7]) || (~MSB && CinL));
        end
      2'b10:
        begin
          ShiftBR = 1'b1;
          assign CinALU = ((MSB && ALU_B[7]) || (~MSB && CinL));
        end
      2'b01:
        begin
          ShiftAL = 1'b1;
          assign CinALU = ((LSB && ALU_A[0]) || (~LSB && CinR));
        end
      2'b11:
        begin
          ShiftBL = 1'b1;
          assign CinALU = ((LSB && ALU_B[0]) || (~LSB && CinR));
        end
    endcase
  end
4'd6: /* Shift with Insert 0 */
  begin
    assign InvertA = 1'b0;
    assign InvertB = 1'b0;
    case(Fin[5:4])
      2'b00:
        begin
          ShiftAR = 1'b1;
          assign CinALU = ((MSB && 1'b0) || (~MSB && CinL));
        end
      2'b10:
        begin
          ShiftBR = 1'b1;
          assign CinALU = ((MSB && 1'b0) || (~MSB && CinL));
        end
      2'b01:
        begin
```

```
          ShiftAL = 1'b1;
          assign CinALU = ((LSB && 1'b0) || (~LSB && CinR));
        end
      2'b11:
        begin
          ShiftBL = 1'b1;
          assign CinALU = ((LSB && 1'b0) || (~LSB && CinR));
        end
    endcase
  end
4'd7: /* Shift with Insert 1 */
  begin
    assign InvertA = 1'b0;
    assign InvertB = 1'b0;
    case(Fin[5:4])
      2'b00:
        begin
          ShiftAR = 1'b1;
          assign CinALU = ((MSB && 1'b1) || (~MSB && CinL));
        end
      2'b10:
        begin
          ShiftBR = 1'b1;
          assign CinALU = ((MSB && 1'b1) || (~MSB && CinL));
        end
      2'b01:
        begin
          ShiftAL = 1'b1;
          assign CinALU = ((LSB && 1'b1) || (~LSB && CinR));
        end
      2'b11:
        begin
          ShiftBL = 1'b1;
          assign CinALU = ((LSB && 1'b1) || (~LSB && CinR));
        end
    endcase
  end

4'd8: /* Add */
  begin
    assign InvertA = Fin[4];
    assign InvertB = Fin[5];
    ADD = 1'b1;
    assign CinALU = CinR;
  end
4'd9: /* Add-0 */
  begin
    assign InvertA = Fin[4];
    assign InvertB = Fin[5];
    ADD = 1'b1;
    assign CinALU = ((LSB && 1'b0) || (~LSB && CinR));
  end
4'd10: /* Add-1 */
  begin
```

```
            assign InvertA = Fin[4];
            assign InvertB = Fin[5];
            ADD = 1'b1;
            assign CinALU = ((LSB && 1'b1) || (~LSB && CinR));
          end
        4'd11: /* Unusued Opcode - Treat as an Add-1 */
          begin
            assign InvertA = Fin[4];
            assign InvertB = Fin[5];
            ADD = 1'b1;
            assign CinALU = ((LSB && 1'b1) || (~LSB && CinR));
          end

        4'd12:  /* Pass */
          begin
            if (Fin[5]==1'b1)
              begin
                B_Pass = 1'b1;
                assign InvertA = 1'b0;
                assign InvertB = Fin[4];
              end
            else
              begin
                A_Pass = 1'b1;
                assign InvertA = Fin[4];
                assign InvertB = 1'b0;
              end
          end
        4'd13:
          begin
            NAND = 1'b1;
            assign InvertA = Fin[4];
            assign InvertB = Fin[5];
          end
        4'd14:
          begin
            NOR = 1'b1;
            assign InvertA = Fin[4];
            assign InvertB = Fin[5];
          end
        4'd15:
          begin
            XOR = 1'b1;
            assign InvertA = Fin[4];
            assign InvertB = Fin[5];
          end
      endcase
    end /* Decode */

endmodule
```

```
/************************/
/* Specifications: MEM.v */
/************************/

/* This module emuates the 256x8 memory block which the main MATRIX BFU
   memory.

   Reads happen during the first half of the clock cycle.
   In mode=1, it looks like two 128x8 memories, controlled by the two
      address ports.
   In mode=0, it looks like a single 256x8 memory outputing to both output
      port and controlled by addr_A.
*/

/**************/
/* MEM Module */
/**************/

module MEM(data[7:0], addr_A[7:0], addr_B[7:0], Aout[7:0], Bout[7:0],
                 mode, WE, clk);

  /* Clock */
  input clk;

  /* Data and Address inputs */
  input [7:0] data, addr_A, addr_B;

  /* Write Enable */
  input WE;

  /* Mode select */
  input mode;

  /* Output Ports */
  output [7:0] Aout, Bout;
  reg [7:0] Aout, Bout;

  /* Define the two 128x8 memory blocks */
  reg [7:0] A_block[127:0];
  reg [7:0] B_block[127:0];


  /*********/
  /* Reads */
  /*********/

  always @(posedge clk)
    begin
      if (mode===1'b0)  /* Mode=0 (256 byte block) */
        begin
          if (addr_A[7]===1'b0)
            begin
              #1 Aout = A_block[addr_A[6:0]];
              #1 Bout = A_block[addr_A[6:0]];
```

```
            end
          else
            begin
              #1 Aout = B_block[addr_A[6:0]];
              #1 Bout = B_block[addr_A[6:0]];
            end
        end

      else  /* Mode=1 (2x128 byte block) */
        begin
          #1 Aout = A_block[addr_A[6:0]];
          #1 Bout = B_block[addr_B[6:0]];
        end
    end  /* reads */


  /**********/
  /* Writes */
  /**********/

  always @(negedge clk)
    begin
      if (WE)
        begin
          if (addr_A[7]===1'b0)
              A_block[addr_A[6:0]]=data[7:0];
          else
              B_block[addr_A[6:0]]=data[7:0];
        end
    end /* writes */

endmodule
```

```
/*********************************/
/* Specification for: MEMdecode.v */
/*********************************/

/* This module represents the decoder logic for the MEM/MUX function port of
   a BFU cell.  Its takes the the 8 bit input and decodes it to the memory
   and mux control lines.

   Input Fin[7:0] is the function port input.

        Bit 7: Unused
        Bit 6: Unused
        Bit 5: Mode
        Bit 4: ALU A Select (Ain_sel)
        Bit 3: ALU B Select (Bin_sel)
        Bit 2: Memory Data Select (Data_sel)
        Bit 1: Configuration Memory Write-Enable (WEconf)
        Bit 0: Configuration Memoru Read-ENable (REconf)

*/

/*******************/
/* MEMdecode module */
/*******************/

module MEMdecode (Fin[7:0], Mode, Ain_sel, Bin_sel, Data_sel, WEconf, REconf);

  input [7:0] Fin;

  output Mode, Ain_sel, Bin_sel, Data_sel, WEconf, REconf;
  reg Mode, Ain_sel, Bin_sel, Data_sel, WEconf, REconf;

  /*****************/
  /* Begin Decoding */
  /*****************/

  /* Assign the inputs appropriately */
  initial
    begin
      assign Mode = Fin[5];
      assign Ain_sel = Fin[4];
      assign Bin_sel = Fin[3];
      assign Data_sel = Fin[2];
      assign WEconf = Fin[1];
      assign REconf = Fin[0];
    end /* Decoding */

endmodule
```

```
/*********************************/
/* Specifications for WEdecode.v */
/*********************************/

/*  WEdecode decodes the Write Enable for the BFU memory.  Its inputs are:

    WE_Fa, WE_Fm : Write Enables from the ALU and Memory function ports.
    WEsource     : Selects WE source
    TS_Enable    : Enables Time-Switch Logic
    TS_WE        : TimeStep configuration data
    TimeStep     : Global timestep

    The output of WEdecode is the final Write Enable

    Input start is a model-specific initialization input, used to force the
       module to evaluate its inputs.
*/

/*******************/
/* Module WEdecode */
/*******************/

module WEdecode(WE_Fa, WE_Fm, WEsource, TS_Enable, TS_WE, TimeStep, WE, start);

  input WE_Fa, WE_Fm, WEsource;
  input TS_Enable;
  input [3:0] TS_WE, TimeStep;
  input start;

  output WE;
  reg WE;

  always @(start or TS_Enable or TS_WE or TimeStep)
    begin

      if (TS_Enable) /* Use TimeStep */
        begin
          if (TS_WE != TimeStep)  /* TimeStep doesn't match */
            assign WE = 1'b0;
          else
            begin             /* TimeStep Matches */
              if (WEsource)
                  assign WE = WE_Fm;
              else
                  assign WE = WE_Fa;
            end
        end

      else /* Do not use TimeStep */
        begin
          if (WEsource)
              assign WE = WE_Fm;
          else
              assign WE = WE_Fa;
```

```
        end

    end

endmodule
```

A.4 Helper Modules

```
/* The following is necessary because this file may be read from many inlcude
   statements and should be ignored on all but the first */

`define netswitch_defined

/*********************************/
/* Specifications for NetSwitch.v */
/*********************************/

/*  A NetSwitch is the primary network swiching mechanism in MATRIX.  It consists
    of a 30->1 selector followed by another 2->1 selector, both 8-bits wide.
    (See TN130 for block diagrams).

    The inputs (all 8-bits) to the main switch, in order (0-29):

    0  : Local : The local BFU.

    1  : L1_N1 : Level-1 Network, From North-1 cell.
    2  : L1_N2 : Level-1 Network, From North-2 cell.
    3  : L1_NE : Level-1 Network, From NorthEast cell.
    4  : L1_E1 : Level-1 Network, From East-1 cell.
    5  : L1_E2 : Level-1 Network, From East-2 cell.
    6  : L1_SE : Level-1 Network, From SouthEast cell.
    7  : L1_S1 : Level-1 Network, From South-1 cell.
    8  : L1_S2 : Level-1 Network, From South-2 cell.
    9  : L1_SW : Level-1 Network, From SouthWest cell.
    10 : L1_W1 : Level-1 Network, From West-1 cell.
    11 : L1_W2 : Level-1 Network, From West-2 cell.
    12 : L1_NW : Level-1 Network, From NorthWest cell.

    13 : L2_N1 : Level-2 Network, North-1 Line.
    14 : L2_N2 : Level-2 Network, North-2 Line.
    15 : L2_E1 : Level-2 Network, East-1 Line.
    16 : L2_E2 : Level-2 Network, East-2 Line.
    17 : L2_S1 : Level-2 Network, South-1 Line.
    18 : L2_S2 : Level-2 Network, South-2 Line.
    19 : L2_W1 : Level-2 Network, West-1 Line.
    20 : L2_W2 : Level-2 Network, West-2 Line.

    21 : L3_V1 : Level-3 Network, Vertical-1 Line.
    22 : L3_V2 : Level-3 Network, Vertical-2 Line.
    23 : L3_V3 : Level-3 Network, Vertical-3 Line.
    24 : L3_V4 : Level-3 Network, Vertical-4 Line.
    25 : L3_H1 : Level-3 Network, Horizontal-1 Line.
    26 : L3_H2 : Level-3 Network, Horizontal-2 Line.
    27 : L3_H3 : Level-3 Network, Horizontal-3 Line.
    28 : L3_H4 : Level-3 Network, Horizontal-4 Line.

    29 : CByte : Control Byte.
```

```
    30 : Constant 0
    31 : Conatant 1

    A few definitions are in order:

    On the Level-2 network, "1" and "2" lines are defined as the distance to the
        the broadcasting L2 switch, divided by 2.  Therefore the "1" line could
        either come from 1 or 2 cells away, and the "2" line could come from
        eitehr 3 or 4 cells away.

    On the Level-3 network, Verical is defined as North-South and Horizontal is
        defined as East-West.  Since they apply uniformly to the entire chip,
        the numberings (from 1-4) are arbitrary.


    Other inputs to the NetSwitch are:

    SourceSel (5 bits)  : Source selector - selects from the 30 main inputs.
    StaticByte (8 bits) : The alternate data.
    StaticSel (1 bit)   : Selects between main and alternate data inputs.

    Note that if SourceSel is greater than 29, the main data will be zero.

    All of this produces a single, 8-bit output.

    And, of cource:
    Input start is a model-specific initialization input, used to force the
       module to evaluate its inputs.
*/


/*****************/
/* Include Files */
/*****************/

/* The following prevents Selector from getting re-compiled many times */
`ifdef selector_defined
  `else
    `include "Selector.v"
`endif


/********************/
/* Module NetSwitch */
/********************/

module NetSwitch(Local, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE, L1_S1, L1_S2,
                 L1_SW, L1_W1, L1_W2, L1_NW,
                 L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
                 L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CByte,
                 SourceSel, StaticByte, StaticSel, Out, start);

  input [7:0] Local, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE, L1_S1, L1_S2;
  input [7:0] L1_SW, L1_W1, L1_W2, L1_NW;
```

```
  input [7:0] L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2;
  input [7:0] L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CByte;

  input [4:0] SourceSel;
  input [7:0] StaticByte;
  input       StaticSel;

  output [7:0] Out;

  input start;


  /******************/
  /* Internal Wires */
  /******************/

  wire [7:0] Main; /* the main input, after selection */


  /************************/
  /* Define the Selectors */
  /************************/

  Sel32 MainSel(Local, L1_N1, L1_N2, L1_NE, L1_E1, L1_E2, L1_SE, L1_S1, L1_S2,
                L1_SW, L1_W1, L1_W2, L1_NW,
                L2_N1, L2_N2, L2_E1, L2_E2, L2_S1, L2_S2, L2_W1, L2_W2,
                L3_V1, L3_V2, L3_V3, L3_V4, L3_H1, L3_H2, L3_H3, L3_H4, CByte,
                8'd0, 8'd1,
                Main, SourceSel, start);

  Sel2 #(8) S_Sel(StaticByte, Main, Out, StaticSel, start);


  /* And thats all there is! */
endmodule
```

```
/* The following is necessary because this file may be read from many inlcude
   statements and should be ignored on all but the first */

`define selector_defined

/*********************************/
/* Specifications for Selector.v */
/*********************************/

/*  A selector is is used to choose one input from a set of inputs and pass
    this value to the output (a multiplexor).  There are three types of
    selectors in this file.  Sel2 is a 2 input selector, Sel8 is an 8 input
    selector, Sel16 is a 16 input selector, and  Sel32 is a 32 input
    selector.  Sel2 and Sel8 are parameterized in the size of the inputs
    and output, while Sel16 is fixed at 1-bit and Sel32 are fixed at
    8-bit I/O.

    Unfortunately, due to some problems (probably bugs) with VerilogXL it was
    impossible to construct a single device with parameterized inputs and
    bus-size, or even parameterized inputs.  So, this is what was possible.

    Input start is a model-specific initialization input, used to force the
       module to evaluate its inputs.
*/


/***************/
/* Module Sel2 */
/***************/

module Sel2(In0, In1, Out, ctrl, start);

  /* Define the Bus Width parameter*/
  parameter BusW=1;

  /* The I/O sizes are set accordingly: */
  input [BusW-1:0] In0, In1;
  input ctrl;

  output [BusW-1:0] Out;
  reg [BusW-1:0] Out;

  input start;

  /******************/
  /* Begin Selector */
  /******************/

  always @(ctrl or start)
    begin
      if (ctrl===0)
          assign Out = In0;
      else
          assign Out = In1;
```

```
    end

endmodule


/***************/
/* Module Sel8 */
/***************/

module Sel8(In0,  In1,  In2,  In3,  In4,  In5,  In6,  In7,
            Out, ctrl, start);

  parameter BusW = 8;

  input [(BusW-1):0] In0,  In1,  In2,  In3,  In4,  In5,  In6,  In7;
  input [2:0] ctrl;

  output [(BusW-1):0] Out;
  reg [(BusW-1):0] Out;

  input start;

  /******************/
  /* Begin Selector */
  /******************/

  always @(ctrl or start)
    begin
      case(ctrl)
        5'd0: assign Out=In0;
        5'd1: assign Out=In1;
        5'd2: assign Out=In2;
        5'd3: assign Out=In3;
        5'd4: assign Out=In4;
        5'd5: assign Out=In5;
        5'd6: assign Out=In6;
        5'd7: assign Out=In7;
        default assign Out=8'd0;
      endcase
    end

endmodule


/****************/
/* Module Sel16 */
/****************/

module Sel16(In, Out, ctrl, start);


  /* Note:  For reasons I cannot understand (probably a bug), you need
     to put in a constant when concatinating inputs.  Therefore this
     input is large to accomodate a constant in the high bit position */
```

```
  input [16:0] In;
  input [3:0] ctrl;

  output Out;
  reg Out;

  input start;

  /******************/
  /* Begin Selector */
  /******************/

  always @(ctrl or start)
    begin
      assign Out = In[ctrl];
    end

endmodule


/****************/
/* Module Sel32 */
/****************/

module Sel32(In0,  In1,  In2,  In3,  In4,  In5,  In6,  In7,  In8,  In9,
             In10, In11, In12, In13, In14, In15, In16, In17, In18, In19,
             In20, In21, In22, In23, In24, In25, In26, In27, In28, In29,
             In30, In31, Out, ctrl, start);

  input [7:0] In0,  In1,  In2,  In3,  In4,  In5,  In6,  In7,  In8,  In9;
  input [7:0] In10, In11, In12, In13, In14, In15, In16, In17, In18, In19;
  input [7:0] In20, In21, In22, In23, In24, In25, In26, In27, In28, In29;
  input [7:0] In30, In31;

  input [4:0] ctrl;

  output [7:0] Out;
  reg [7:0] Out;

  input start;

  /******************/
  /* Begin Selector */
  /******************/

  always @(ctrl or start)
    begin
      case(ctrl)
        5'd0: assign Out=In0;
        5'd1: assign Out=In1;
        5'd2: assign Out=In2;
        5'd3: assign Out=In3;
        5'd4: assign Out=In4;
        5'd5: assign Out=In5;
```

```
        5'd6: assign Out=In6;
        5'd7: assign Out=In7;
        5'd8: assign Out=In8;
        5'd9: assign Out=In9;
        5'd10: assign Out=In10;
        5'd11: assign Out=In11;
        5'd12: assign Out=In12;
        5'd13: assign Out=In13;
        5'd14: assign Out=In14;
        5'd15: assign Out=In15;
        5'd16: assign Out=In16;
        5'd17: assign Out=In17;
        5'd18: assign Out=In18;
        5'd19: assign Out=In19;
        5'd20: assign Out=In20;
        5'd21: assign Out=In21;
        5'd22: assign Out=In22;
        5'd23: assign Out=In23;
        5'd24: assign Out=In24;
        5'd25: assign Out=In25;
        5'd26: assign Out=In26;
        5'd27: assign Out=In27;
        5'd28: assign Out=In28;
        5'd29: assign Out=In29;
        5'd30: assign Out=In30;
        5'd31: assign Out=In31;
        default assign Out=8'd0;
      endcase
    end

endmodule
```

```
/* The following is necessary because this file may be read from many inlcude
   statements and should be ignored on all but the first */

`define TSregister_defined


/*********************************/
/* Specifications for TSregister.v */
/*********************************/

/* A TSregister is a Time-Switch register.  It is basically a normal clocked
   register (always enabled), except that it can optionally enabled by
   comparing an incoming cycle value to a stored configuration word.  See
   TN130 for more details of this.

   TSregister is parameterized to the width of the Data.

   The TSregister structure also contains a reset signal which can force the
   register to load zeros.

   The inputs to TSregister are:

   Data : The data input
   Cycle : The current cycle
   TSenable : Enables the Time-Switch Logic
   Config : The configuration word

   CLK : A clock
   start : The simulation reset signal

   Out is the only output.
*/


/*********************/
/* Module TSregister */
/*********************/

module TSregister (Data, Cycle, Out, TSenable, Config, CLK, start);


  /* Set default Data width */
  parameter Width = 8;

  /* Set I/O accordingly */
  input [Width-1:0] Data;
  input [3:0] Cycle, Config;
  input TSenable, CLK, start;

  output [Width-1:0] Out;
  reg [Width-1:0] Out;


  /******************/
```

```
  /* Internal State */
  /******************/

  reg Enable;  /* The result of the Cycle-Config comparison */


  /********************/
  /* Maintain Enable */
  /********************/

  initial
    begin
      assign Enable = (Cycle == Config);
    end


  /****************/
  /* Handle Start */
  /****************/

  always @(start)
    begin
      #1 Out = Data;
    end


  /********************/
  /* Everything Else */
  /********************/

  always @(posedge(CLK))
    begin
      if (Enable || ~TSenable)
        begin
          #1 Out = Data;
        end
    end

endmodule
```

```
/**********************************/
/* Specifications for TSand.v */
/**********************************/

/* A TSand is a Time-Switch AND gate.  It takes a single input (parameterized
   width) and bit-wise ANDs it with the result of a Time-Switch comparison.
   If TSenable is off, the comparison is always true.

   The inputs to TSregister are:

   Data : The data input
   Cycle : The current cycle
   TSenable : Enables the Time-Switch Logic
   Config : The configuration word

   start : The simulation reset signal

   Out is the only output.
*/


/****************/
/* Module TSand */
/****************/

module TSand (Data, Cycle, Out, TSenable, Config, start);


  /* Set default Data width */
  parameter Width = 1;

  /* Set I/O accordingly */
  input [Width-1:0] Data;
  input [3:0] Cycle, Config;
  input TSenable, start;

  output [Width-1:0] Out;
  reg [Width-1:0] Out;


  /****************/
  /* Maintain Out */
  /****************/

  always @(start or TSenable or Cycle or Config)
    begin
      if (TSenable === 0)
          assign Out = Data;
      else if (Cycle === Config)
          assign Out = Data;
      else
          assign Out = 0;
    end
```

```
endmodule
```

```
/* The following is necessary because this file may be read from many inlcude
   statements and should be ignored on all but the first */

`define tribuf_defined

/*******************************/
/* Specifications for Tribuf.v */
/*******************************/

/* Tribuf is a non-clocked tristate buffer, which passes on ctrl=1.

   Tribuf takes the bit-widths of the data lines as a parameter.

   Inputs:

   In    : The Input
   Ctrl  : The control bit
   start : A model-specific initialization input, used to force the
               module to evaluate its inputs.

   Out : The Output

*/


/*****************/
/* Module Tribuf */
/*****************/

module Tribuf(In, Ctrl, start, Out);

  /* Set the default parameter */
  parameter size = 1;

  input [size-1:0] In;
  input Ctrl, start;

  output [size-1:0] Out;
  reg [size-1:0] Out;

  integer i;

  /***************/
  /* Begin model */
  /***************/

  always @(Ctrl or start)
    begin
      case (Ctrl)
        1'b0 :
          begin
            #1;
            deassign Out;
            for (i=0; i<size; i=i+1)
```

```
                Out[i]=1'bz;
          end
        1'b1 : #1 assign Out = In;
        default
          begin
            #1;
            deassign Out;
            for (i=0; i<size; i=i+1)
                Out[i]=1'bz;
          end
      endcase
    end

endmodule
```

```
/* The following is necessary because this file may be read from many inlcude
   statements and should be ignored on all but the first */

`define trireg_defined

/******************************/
/* Specifications for Trireg.v */
/******************************/

/* TriReg is a clocked tristate buffer (register), which passes on ctrl=1.

   TriReg takes the bit-widths of the data lines as a parameter.

   Inputs:

   In    : The Input
   Ctrl  : The control bit.
   CLK   : A clock
   start : A model-specific initialization input, used to force the
               module to evaluate its inputs.

   Out : The Output

*/


/*****************/
/* Module Trireg */
/*****************/

module Trireg(In, Ctrl, CLK, start, Out);

  /* Set the default parameter */
  parameter size = 1;

  input [size-1:0] In;
  input Ctrl, CLK, start;

  output [size-1:0] Out;
  reg [size-1:0] Out;

  integer i;

  /***************/
  /* Begin model */
  /***************/

  always @(posedge(CLK) or start)
    begin
      case (Ctrl)
        1'b0 :
          begin
            #1;
            for (i=0; i<size; i=i+1)
```

```
                Out[i]=1'bz;
          end
        1'b1 : #1 Out = In;
        default
          begin
            #1;
            for (i=0; i<size; i=i+1)
                Out[i]=1'bz;
          end
      endcase
    end

endmodule
```

Bibliography

[1] Michael Bolotski, Thomas Simon, Carlin Vieri, Rajeevan Amirtharajah, and Thomas F. Knight Jr. Abacus: A 1024 processor 8ns simd array. In *Advanced Research in VLSI 1995*, 1995.

[2] Timothy Bridges. The gpa machine: A generally partitionable msimd architecture. In *Proceedings of the Third Symposium on The Frontiers for Massively Parallel Computations*, pages 196–202. IEEE, 1990.

[3] Dev C. Chen and Jan M. Rabaey. A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, December 1992.

[4] Chi-Jui Chou, Satish Mohanakrishnan, and Joseph B. Evans. Fpga implementation of digital filters. In *International Conference on Signal Processing Applications and Technology*, 1993.

[5] André DeHon. *Reconfigurable Architectures for General-Purpose Computing.* PhD dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1996. Draft version - expected completion: July, 1996.

[6] Dave Epstein. Chromatic raises the multimedia bar. *Microprocessor Report*, 9(14):23 ff., October 23 1995.

[7] Carla Golla, Fulvio Nava, Franco Cavallotti, Alessandro Cremonesi, and Giulio Casagrande. 30-msamples/s programmable filter processor. *IEEE Journal of Solid-State Circuits*, 25(6):1502–1509, December 1990.

[8] Greg Goslin and Bruce Newgard. *16-TAP, 8-Bit FIR Filter Applications Guide*. Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, November 1994. `http://www.xilinx.com/appnote/fir_filt.pdf`.

[9] Paul Gronowski, Peter Bannon, Michael Bertone, Randel Blake-Campos, Gregory Bouchard, William Bowhill, David Carlson, Ruben Castelino, Dale Donchin, Richard Fromm, Mary Gowan, Anil Jain, Bruce Loughlin, Shekhar Mehta, Jeanne Meyer, Robert Mueller, Andy Olesin, Tung Pham, Ronald Preston, and Paul Robinfeld. A 433mhz 64b quad-issue risc microprocessor. In *1996 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 222–223. IEEE, February 1996.

[10] Mark Horowitz, John Hennessy, Paul Chow, Glenn Gulak, John Acken, Anant Agarwal, Chorng-Yeung Chu, Scott McFarling, Steven Przybylski, Steven Richardson, Arturo Salz, Richard Simoni, Don Stark, Peter Steenkiste, Steven Tjiang, and Malcom Wing. A 32b microprocessor with on-chip 2k byte instruction cache. In *1987 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 30–31. IEEE, February 1987.

[11] David Jones and David Lewis. A time-multiplexed fpga architecture for logic emulation. In *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pages 495–498. IEEE, May 1995.

[12] Ethan Mirsky. Matrix micro-architecture. Transit Note 130, MIT Artificial Intelligence Laboratory, November 1995.

[13] Ethan Mirsky and André DeHon. Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[14] Kouhei Nadehara, Miwako Hayashida, and Ichiro Kuroda. *A Low-Power, 32-bit RISC Processor with Signal Processing Capability and its Multiply-Adder*, volume VIII of *VLSI Signal Processing*, pages 51–60. IEEE, 1995.

[15] Gary J. Nutt. Microprocessor implementation of a parallel processor. In *Proceedings of the Fourth Annual International Symposium on Computer Architecture*, pages 147–152. ACM, 1977.

[16] Peter Ruetz. The architectures and design of a 20-mhz real-time dsp chip set. *IEEE Journal of Solid-State Circuits*, 24(2):338–348, April 1989.

[17] M. Shiraishi, M. Koizumi, A. Yamaguchi, and H. Hoike. User programmable 16bit 50ns dsp. In *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pages 6.4.1–6.4.4. IEEE, May 1992.

[18] Michael Slater. Microunity lifts veil on mediaprocessor. *Microprocessor Report*, 9(14):11 ff., October 23 1995.

[19] Lawrence Snyder. An inquiry into the benefits of multigauge parallel computation. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 488–492. IEEE, August 1985.

[20] Edward Tau, Ian Eslick, Derrick Chen, Jeremy Brown, and André DeHon. A first generation dpga implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pages 138–143, May 1995.

[21] Jef van Meerbergen, Frank Welten, Frans van Wijk, Jan Stoter, Jos Huisken, Antoine Delaruelle, and Karel Van Eerdewijk. An 8 mips cmos digital signal processor. In *1985 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 84–85. IEEE, February 1986.

[22] Alfred K. Yeung and Jan M. Rabaey. A 2.4 gops data-drivern reconfigurable multiprocessor ic for dsp. In *Proceedings of the 1995 IEEE International Solid-State Circuits Conference*, pages 108–109. IEEE, February 1995.