CAD TOOL EMULATION FOR A TWO-LEVEL RECONFIGURABLE CELL ARRAY FOR DIGITAL SIGNAL PROCESSING

By

JONATHAN KARL LARSON

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2005

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of JONATHAN KARL LARSON find it satisfactory and recommend that it be accepted.

Chair

ACKNOWLEDGMENTS

This research was done in conjunction with the High Performance Computer Systems (HiPerCopS) research group under the direction of Dr. José Delgado-Frias at Washington State University. I would particularly like to thank Mitchell Myjak, who developed the foundational designs that the CAD tools were developed for. I would also like to thank Mitchell for all his help and support. Finally, I would like to thank Dr. José Delgado-Frias for his guidance and support.

CAD TOOL EMULATION FOR A TWO-LEVEL RECONFIGURABLE CELL

ARRAY FOR DIGITAL SIGNAL PROCESSING

Abstract

by Jonathan Karl Larson, M.S.
Washington State University
December 2005

Chair: José Delgado-Frias

The use of reconfigurable hardware has been increasing in recent years; the need for robust Computer Aided Design (CAD) tools has risen accordingly. For a reconfigurable system, CAD tools enable developers to create, modify, simulate, and estimate the performance of synthesized designs. Furthermore, once a design is realized, the CAD tools can be used to generate a configuration file to program the hardware.

This research deals with the design of comprehensive CAD tools for a medium-grain reconfigurable cell array. This architecture has been developed by the High Performance Computer Systems (HiPerCopS) research group at Washington State University to accelerate digital signal processing (DSP). The CAD tools include a full-featured designer utility that allows users to map sophisticated DSP algorithms onto the array. A hardware-level simulator verifies these designs and produces results for benchmarking. Finally, an array of supporting structures, such as designer history and part libraries, are built around the core tools to provide a full-featured user experience.

The CAD tools have been used to evaluate the reconfigurable architecture itself. The results from the simulated benchmarks help verify the performance and functionality of the system, and suggest potential areas for improvement. As a result, a more powerful interconnection network has been designed to simplify the mapping process and to improve performance. Thus, the CAD tools provide a powerful platform for future research into medium-grain reconfigurable architectures.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Dedication

This thesis is dedicated to Jesus Christ, my Lord and Savior. Also I would like to dedicate this to my parents, who always encouraged me in everything I did.

CHAPTER 1

Introduction

Digital signal processing (DSP) is used in numerous digital systems today. Embedded devices such as cellular phones, satellite radios, and video cards have permeated everyday life. However, DSP places great demands on the processing power of the underlying hardware. As technology continues to advance, reconfigurable hardware has become a well-accepted option for implementing DSP. This alternative balances the flexibility of a microprocessor with the performance of dedicated hardware. Sophisticated computer aided design (CAD) tools allow developers to synthesize algorithms onto the reconfigurable platform.

Traditional fine-grain devices such as field-programmable gate arrays (FPGAs) can implement arbitrary logic equations. However, binary arithmetic such as multiplication creates a bottleneck when mapped onto fine-grain cells. Many FPGAs have incorporated dedicated multipliers for this reason. On the other hand, researchers have proposed new coarse-grain architectures that provide inherent support for DSP computations [1]. These alternatives include the one-dimensional RaPiD array [2], the two-dimensional KressArray [3], and the heterogeneous Pleiades [4] and MONTIUM [5] architectures that combine both fine-grain and coarse-grain components. While coarse-grain devices offer enhanced performance, their functionality may be limited to basic operations of a predefined word length.

As a third alternative, medium-grain architectures attempt to balance performance and flexibility. Each cell may only work with 4-bit or 8-bit data, so a module such as a 16-bit multiplier would require several cells. However, cells typically can support a wider variety of operations. To this end, the High Performance Computing Systems (HiPerCopS) research group at Washington State University has developed a novel medium-grain architecture that strives for efficient circuit-level implementation [6]. Each cell can perform mathematical functions or memory operations.

Mapping DSP onto reconfigurable hardware requires a sophisticated set of software tools. This software may include place and route functionality, timing analysis, and circuit simulations. As an initial step, we have created CAD tools that allow users to map algorithms onto the HiPerCopS architecture by hand. A hardware-level simulator performs verification and testing of the designs. We have used the software to implement several DSP benchmarks and evaluate the performance of the system. Based on the results obtained from these designs, we propose to modify the interconnection network in the HiPerCopS architecture. We have developed a second version of the CAD tools to allow for comparisons between the two alternatives.

1.1 Background

The HiPerCopS reconfigurable architecture integrates an array of medium-grain cells with a pipelined interconnection network [7]. DSP algorithms are divided into modules, such as multipliers and adders, and mapped onto blocks of cells. Each cell handles a 4-bit operation within the module. Cells can perform mathematics functions or implement a small memory. The outputs of each cell are sent to the inputs of the next cell via a two-tiered interconnect structure.

Conceptually, the HiPerCopS architecture contains three layers:

1. Element Layer – Each cell consists of a 4x4 matrix of lookup tables, known as elements. The matrix of elements can be configured into two structures [8]. In mathematics mode, shown in Figure 1, each element stores a lookup table for a mathematics function. In memory mode, shown in Figure 2, the elements collectively implement a random-access memory. More details about the design of basic functions appear in the reference.

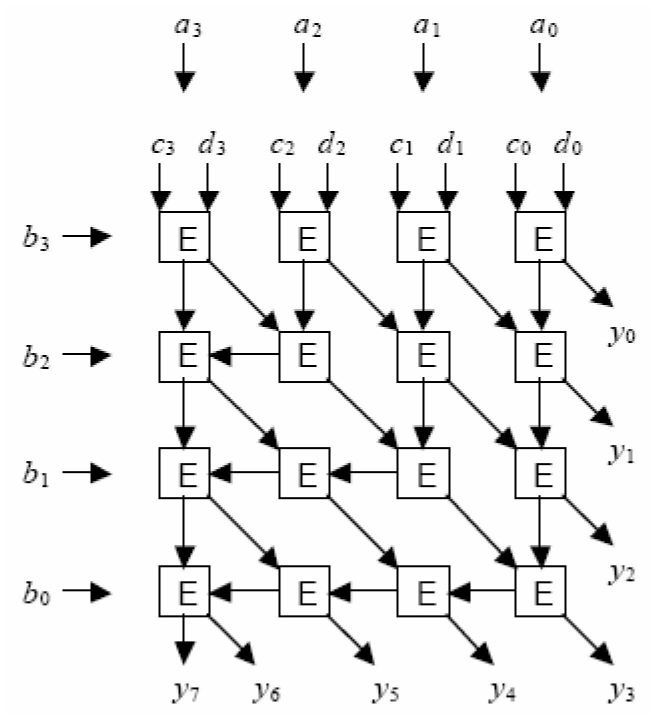


Figure 1. Cell in mathematics mode

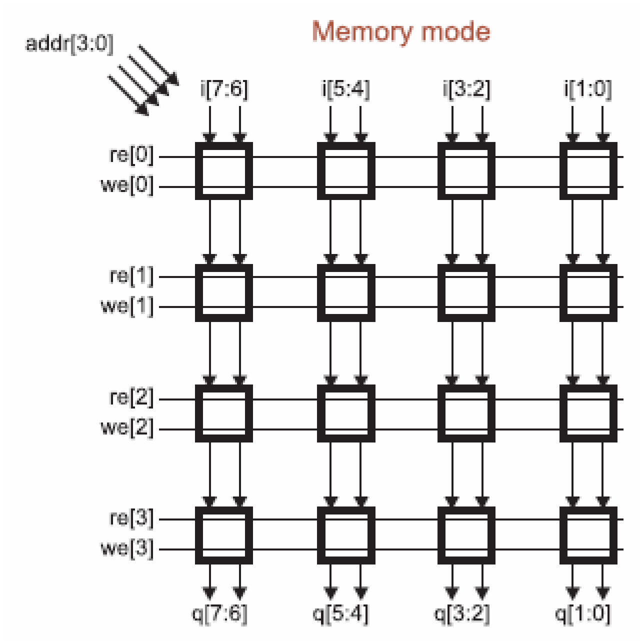


Figure 2. Cell in memory mode

2. Local Network Layer – This layer transfers data in 4-bit units between two cells in the same module. The structure consists of a mesh of busses that connect adjacent cells. Cells also contain two internal switches to connect the matrix of elements to the local network: one for the input side, and one for the output side. All outputs are pipelined to maintain high throughput.
3. Global Network Layer – This layer connects the outputs of one module with the inputs of the next module. The global network overlays on top of the local network, but can transfer data quickly across the entire array. The structure is modeled after a binary tree and is described in detail in [7]. Each level of the tree contains pipeline latches. Figure 3 shows the lowest level of the global network and how it interfaces with the cells.

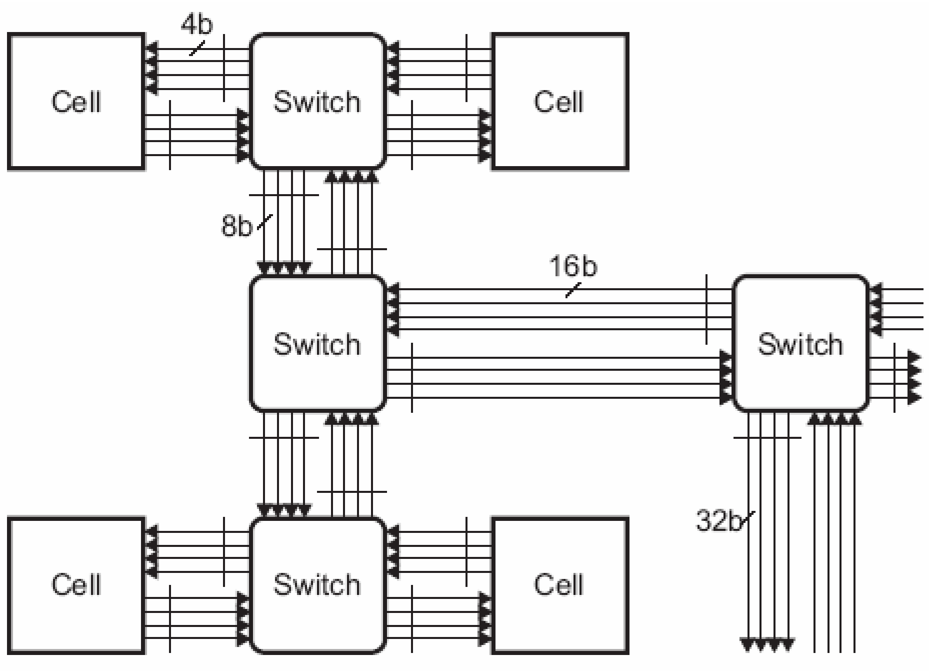


Figure 3. Structure of global network [9]

The original HiPerCopS architecture utilized a mesh of busses on the local network. A screenshot showing this structure appears in Figure 4. As shown, each cell has four local switches adjacent to it. The cell uses these switches to route data to the cell on the other side of the switch. Notice that the lowest level of the global network feeds directly into the local network.

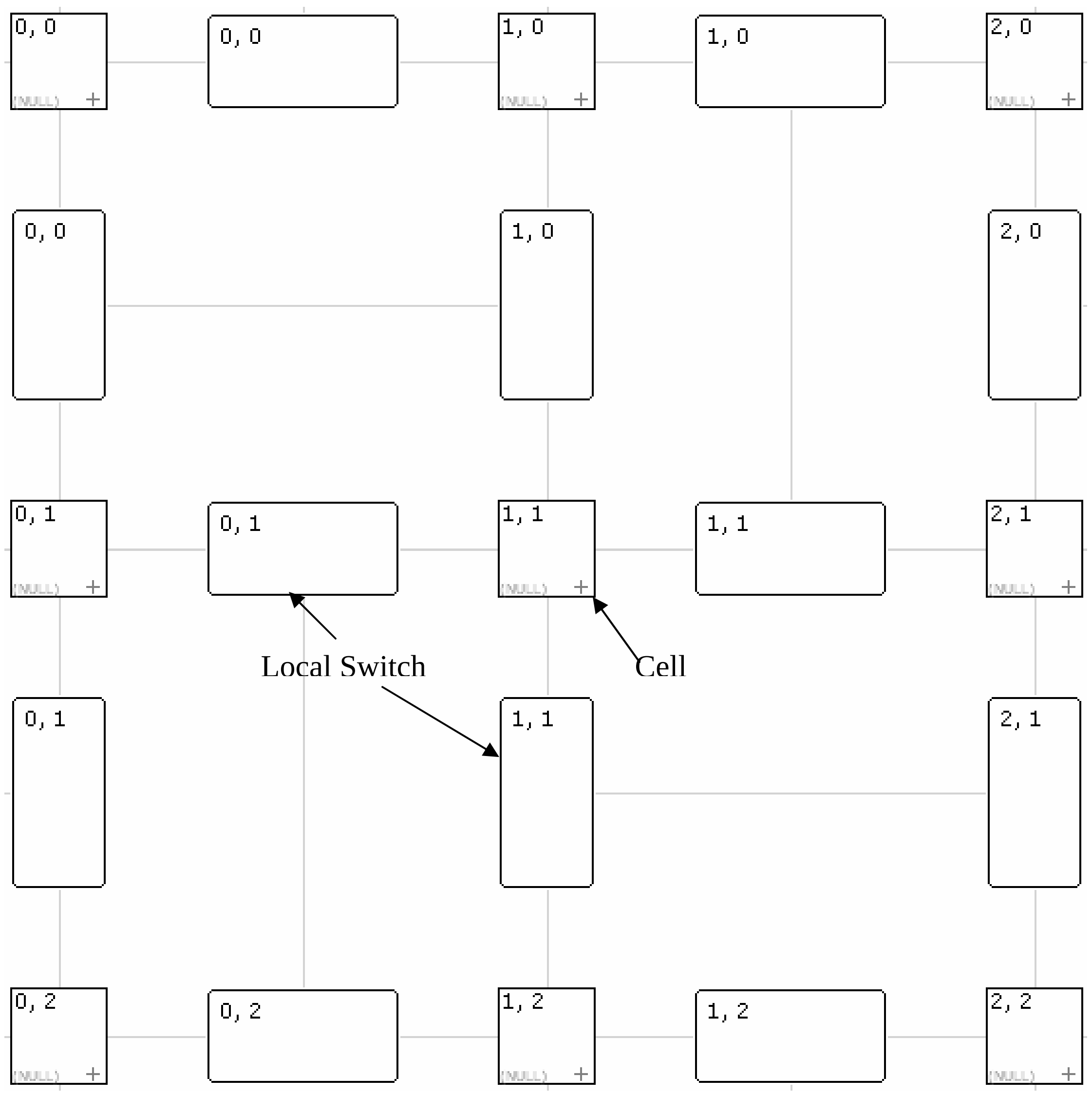


Figure 4. Screenshot of local network in Manhattan architecture

After mapping some DSP algorithms onto the HiPerCopS architecture, we discovered that using a different interconnection network might improve the overall performance. This alternative eliminates the local switches, so each cell interfaces directly with other cells or global switches. The local network also contains diagonal connections between cells rather than strictly horizontal and vertical lines. Figure 5 contains a screenshot showing this structure. Due to the absence of local switches, an additional layer of the global network is needed to interface with the cells. In effect, this doubles the bandwidth of the global network.

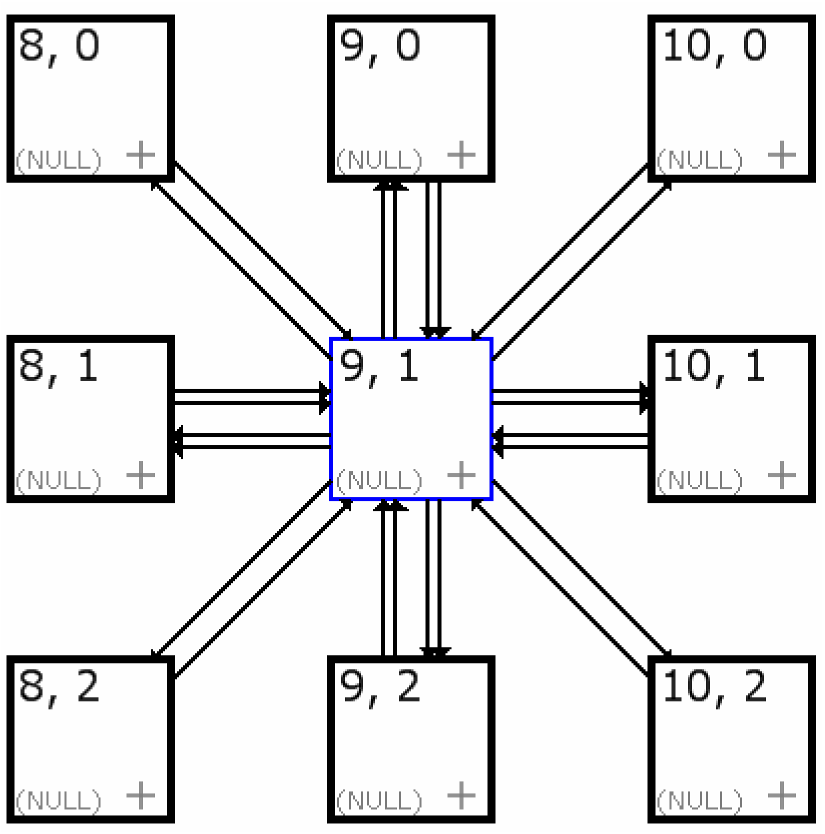


Figure 5. Screenshot of local network in Washington architecture

In this thesis, the original version of the reconfigurable architecture is named HiPerCopS Manhattan (for its grid-like nature), whereas the new version is named HiPerCopS Washington (for its diagonal routing). The Washington architecture offers the flexibility of the Manhattan architecture, while improving upon bus capacities. Just as the two architectures differ fundamentally on the local network, the software CAD tools also reflect these design changes.

1.2 Development Platform

The CAD tools were developed using Microsoft Visual Studio .NET 2003. All of the code was developed using the free student version of Microsoft Developer Network's Academic Alliance software. C# was the primary language used for the development as it provides a rapid prototyping environment. Lower-level languages were considered, but C# offered much shorter development time. The main graphics engines within the CAD software used GDI+, though the components could be expanded later to support hardware rendering. The designs and tests were run on a 3.2-GHz P4 HT system with 2 GB of RAM and a 320-GB performance-striped RAID rack.

1.3 Outline

The remainder of this thesis gives an overview of the software design of the CAD tools developed for the two HiPerCopS reconfigurable architectures. Chapter 2 covers the basic software structures needed to map DSP algorithms. Chapter 3 describes the basic design of the simulator. Chapter 4 deals with the underlying support structures for the software. Chapter 5 looks at the performance of the CAD tools. Chapter 6 compares HiPerCopS Manhattan with HiPerCopS Washington. Finally, Chapter 7 concludes the thesis and looks ahead into future work.

CHAPTER 2

CAD Tool Design

2.1 Architecture

2.1.1 Designer

The primary purpose of the CAD tools centers around the ability to model the HiPerCopS reconfigurable architectures. Using the CAD tools, developers can map DSP algorithms onto the array of cells and ensure that these designs adhere to the applicable constraints. Future extensions of the tools could translate the design into a configuration file for programming the hardware. Because of this, the designer was built with the intention of modeling the hardware as closely as possible. Thus, the designer stores the fine-grain details of the synthesized design, such as the individual truth tables used in the cell elements. However, the component does abstract some principles, such as switch programming.

The abstract layout of the data structures that applies to both architectures to represent the hardware is as follows:

- Cell Array
- Local Switch Array
- Global Switch Array

2.1.2 Impact of different architectures

Programming CAD tools for two different architectures that use the same basic array of cells allowed for the re-use of many code modules. Following the development of HiPerCopS Washington, all of the simulator and most of the designer were leveraged into a shared code base. This was possible because the simulator does not actually perform routing itself, but uses an underlying support structure that is described in Chapter 4.

2.2 Cells

The core component to both architectures is the cell. Internally, the cell is responsible for producing the core calculations that are then propagated to other cells. As such, the software model of the cell includes the following:

- The interface between the cell and the interconnection network
- The lookup tables inside each element

Each of these parts will be discussed in detail in the following sections.

2.2.1 Cell Routing Tables

Each cell contains a pair of switches that connect the matrix of elements to the interconnection network. Figure 6 depicts these two internal switches. The input switch routes data from the interconnection network to the eight inputs of the matrix of elements, which are named as follows: a, b, c, d, w, x, y, z. The output switch routes the eight outputs of the matrix of elements back to the interconnection network. The Manhattan architecture allows cells to interface with eight input lines and eight output lines. The Washington architecture doubles this number to sixteen.

The designer represents the input and output switches with routing tables that contain one entry per switch input. Each table entry is a bitmap that specifies whether that switch input connects to any one of the switch outputs. This method allows one source line to drive multiple destination lines. Table 1 shows the format of an entry, and Table 2 describes the fields inside the bitmap.

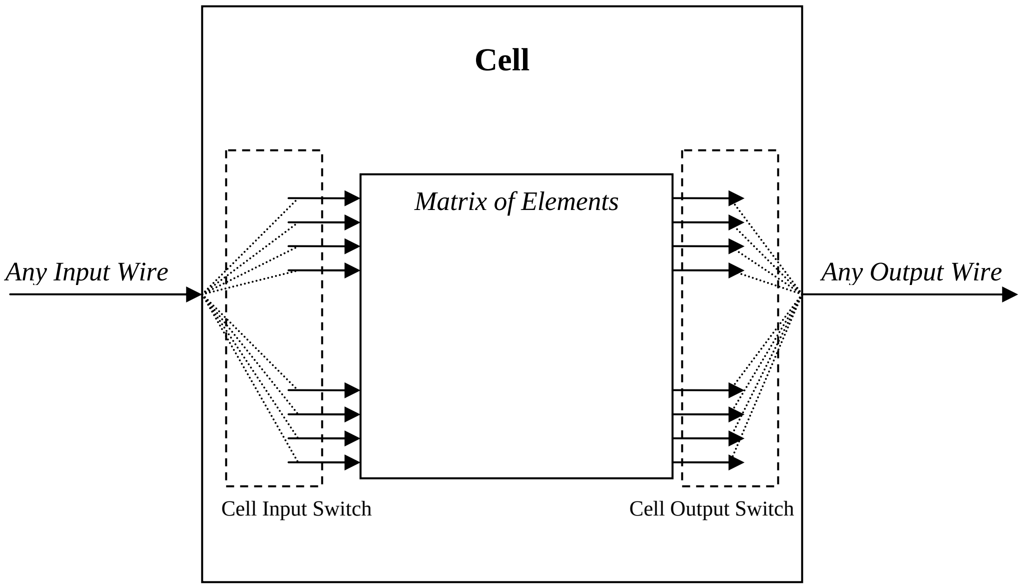


Figure 6. Diagram of cell showing internal switches

Table 1. Routing table entry for cell

Cell Routing Table Bitmap by bit position

A	D1	D0	r	r	r	r	r	r	r	r	r	r	r	r	r
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

G	G	G	G	W	W	W	W	Y	Y	Y	Y	Y	Y	Y	Y
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 2. Fields in routing table entry for cell

Field	Definition
A	Specifies whether the input line is active.
D1, D0	Delay bits. (Only applies to the output routing tables.) Specifies how many latches to send the output through. This is used to create pipeline delays for data synchronization elsewhere in the array. Default of 1 delay when D0 and D1 are 0. Can delay up to 3 extra cycles.
r	Reserved bits not used for actual design specifications. Can be used in the user interfaces for faster tracing, wire coloring, or other various functions.
G	Only applies to the output routing tables in HiPerCopS Washington. Denotes a connection directly to the adjacent global switch.
W	Only applies to the output routing tables in HiPerCopS Washington. Denotes a connection in a diagonal direction to an adjacent cell. See Figure 9 for more details on the diagonal wiring present in the Washington Architecture.
Y	These bits are handled differently depending on whether the bitmap is in an output routing table or an input routing table. *Input routing tables:* Map the incoming wire to the processing core. Bits are mapped to the matrix of elements as shown in Table 3 . **Table 3. Input mapping** (below) *Output Tables:* Map the outbound processor core outputs to the corresponding wires around the cell.

Table 3. Input mapping

Input	a	b	c	d	w	x	y	z
Bit	7	6	5	4	3	2	1	0

2.2.2 Cell Mapping

Each component of the designer—cells, local switches, and global switches—assigns numbers to its input and output lines based on position. To propagate data from one component to the next, the CAD tools must define a mapping between the numbering schemes used by these components. This section outlines these standards.

For the Manhattan architecture, cell wires map to the local network as shown in Figure 7. Horizontally-oriented switches have a different mapping than vertically-oriented switches (VSwitches). Table 4 shows the mappings between cells and switches in tabular form.

Table 4. Cell-to-switch mapping in Manhattan architecture

Cell	Switch	VSwitch
0	-	6
1	-	7
2	0	-
3	1	-
4	-	0
5	-	1
6	6	-
7	7	-

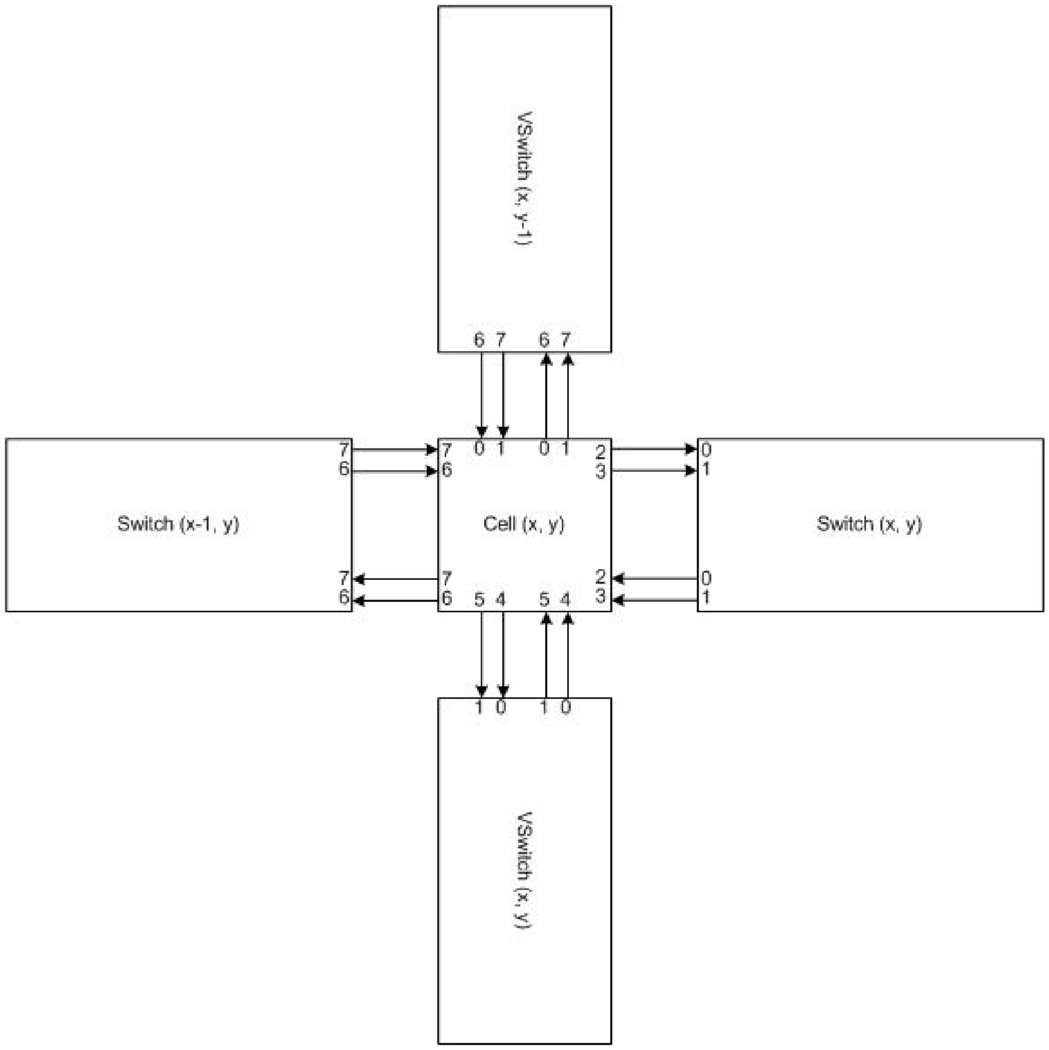


Figure 7. Cell-to-switch mapping in Manhattan architecture

For the Washington architecture, cells map directly to one another as shown in Figure 8. and Figure 9. Extracting this figure into a table yields the list in Table 5.

.

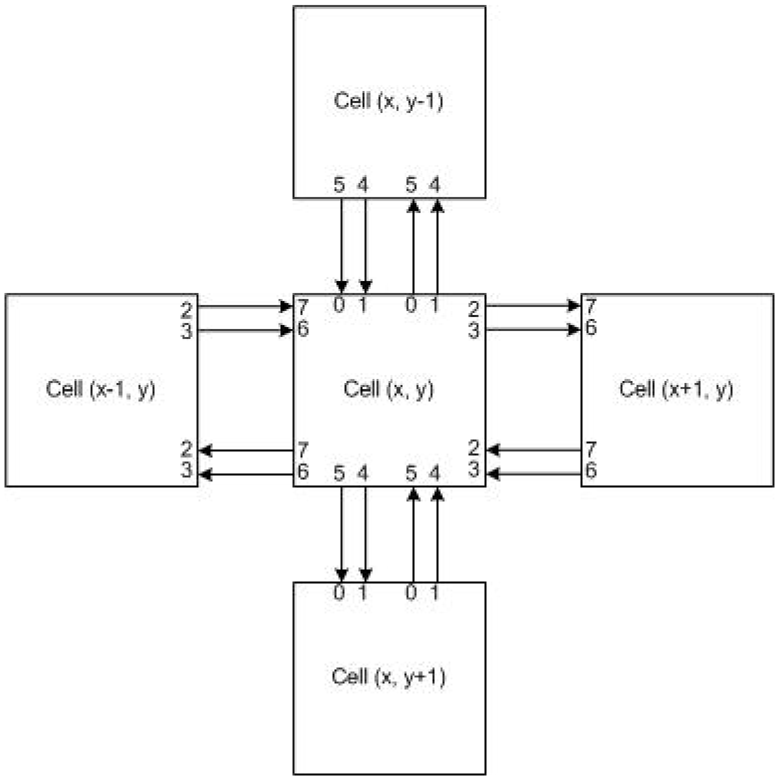


Figure 8. Orthogonal cell-to-cell mapping in Washington architecture

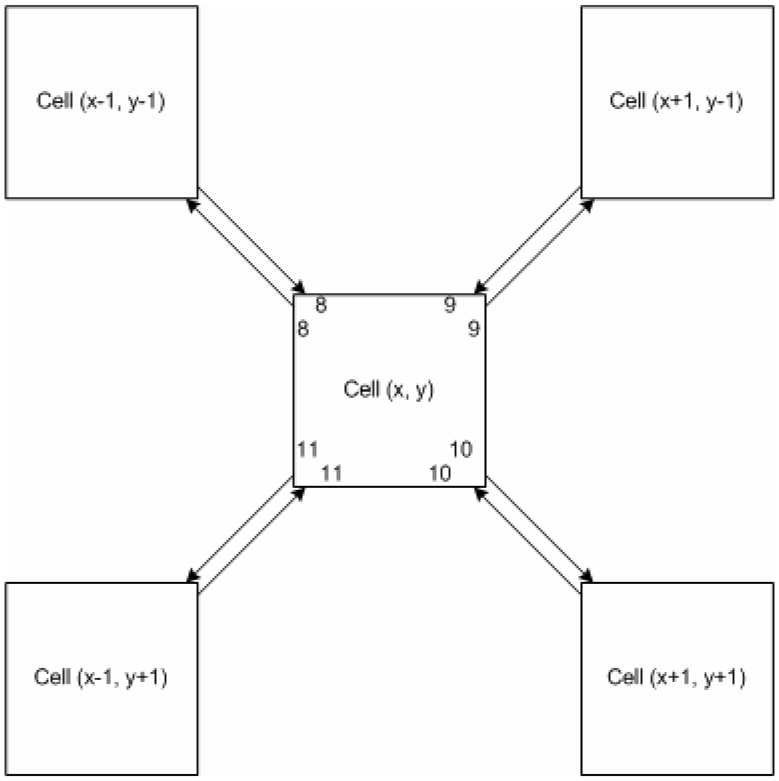


Figure 9. Diagonal cell-to-cell mapping in Washington architecture

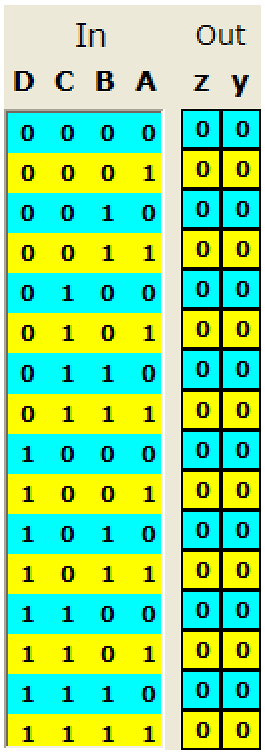
Table 5. Cell-to-cell mapping in Washington architecture

Cell Output	**Cell Input**
0	5
1	4
2	7
3	6
4	1
5	0
6	3
7	2

2.2.3 Cell Elements

The designer must program the matrix of elements inside each cell as well as the routing around it. Because different cells may use the same configuration of elements, the CAD tools feature a part library, described in Chapter 4. Each part stores a configuration for the lookup table (LUT) inside each element. These configurations may then be used to program cells within the array, at which point the cells will be labeled accordingly.

In mathematics mode, the lookup table for each element is represented in memory by a raw 32-bit bitmap. The upper 16 bits refer to the 'Y' portion of the truth table, whereas the lower 16 bits refer to 'Z' portion. This raw bitmap is abstracted through the ConnectionDS class and is exposed through the TruthTable user control that appears in the element editor. A screenshot of this user control is shown in Figure 10.



In				Out	
D	C	B	A	z	y
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

Figure 10. Truth Table User Control

In memory mode, the designer does not model each element individually, but rather abstracts the entire matrix of elements to a random-access memory. The memory data is stored as metadata and has the ability to persist data across cycles of simulation. If a cell is configured into memory mode, that cell will contain a pointer off to the memory segment for which it is responsible. This solution has proved much more useful than treating the cell as a 4x4 matrix of LUTs, and is an example of simulator behavioral abstraction, which is discussed in Chapter 3.4.

Cell data for simulation is stored as a collection of registers. Figure 11 shows the basic data structure used for this purpose. CycleData is a two-dimensional array that holds the current contents of the registers for each cell. The IsDefined property is a bitmask that tells the simulator whether a specified input is valid. The simulator skips over a cell unless all applicable bits are active—a feature that can be useful in debugging. Finally, the MemData structure is a linked list of all of the memory cells.

All data in the system is input driven. Accordingly, CycleData contains all the data coming into the inputs of the cell for the given cycle. The cell latches in a typical simulation cycle will consume approximately 8 KB of memory for a 64x64 array of cells (not accounting for memory mode cells and their metadata).

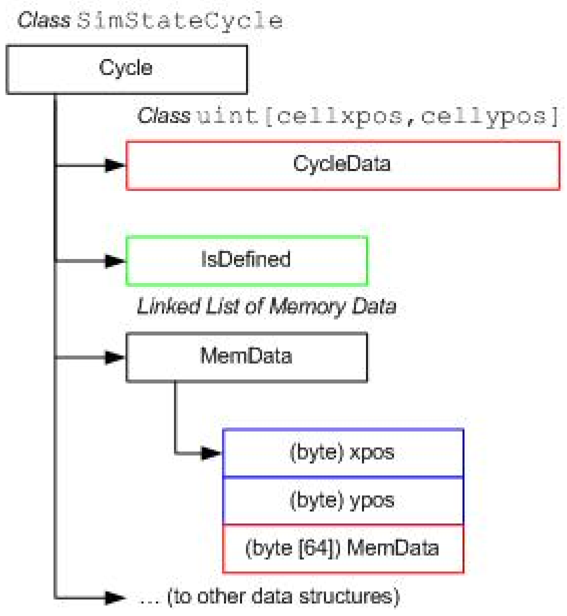


Figure 11. Cell Data Structure

2.3 Global Switches

Global switches are used to connect cells that are far away from each other. In terms of connections between lines, all global switches share a common structure. However, switches on higher levels route data in larger word units than switches on lower levels. Global switches only connect with other global switches or directly to cells. In fact, given the coordinates of a cell, one can easily determine the coordinates and location of the global switch that connects to it. As with local switches, the global switches do not perform any operation on the data other than routing. However, global switches do contain pipeline latches between each layer.

2.3.1 Global Switch Instantiation

To support a 64x64 grid of cells, the CAD tools instantiate ten layers of global switches. The number given to each layer corresponds to the number of clock cycles that it would take for data to travel from the global switch to the local network. Every layer has two groups of eight busses connecting to the previous layer, and one group of sixteen busses connecting to the next layer. Table 6 shows the number of global switches in each layer for a Manhattan 64x64 grid of cells and the bus width for each. The Washington Architecture introduces another layer of 64x32 switches beneath layer 0.

Table 6: Global switches in Manhattan architecture

Layer	Number of Global Switches	Bits on bus to previous layer	Bits on bus to next layer
0	32x32	4	8
1	16x32	8	16
2	16x16	16	32
3	8x16	32	64
4	8x8	64	128
5	4x8	128	256
6	4x4	256	512
7	2x4	512	1024
8	2x2	1024	2056
9	1x2	2056	4096
10	1x1	4096	-

2.3.2 Global Switch to Local Switch Network Locations

Level 0 global switches (which bridge between the global network and local network) are located inside a square of local switches and cells, directly between two horizontal local switches which have an even x coordinate. There is no global switch between horizontal local switches that have an odd x coordinate. Figure 12 shows the positioning of the global switch with respect to the local network.

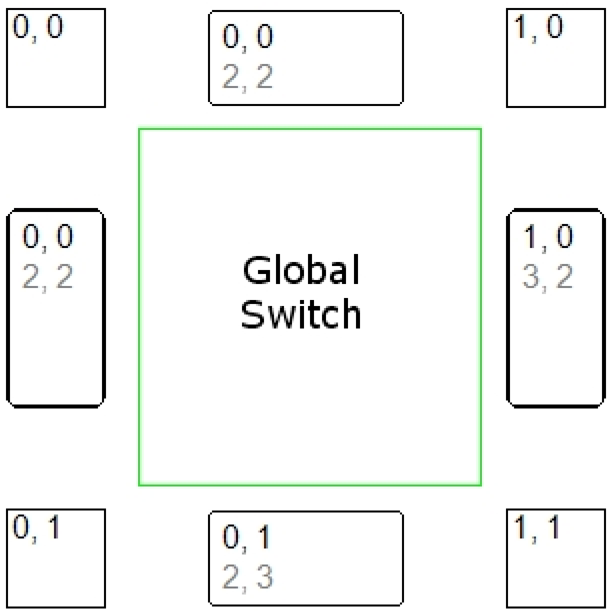


Figure 12. Global Switch Positioning Overlay

2.3.3 Global Switch Routing:

Even though the global switches differ in bus width, the number of busses on each switch remains the same. The designer only needs to know the direction in which the data is routed. The simulator uses this routing to track data through the simulation stages. The global switch routing is represented by input-based bitmaps (similar to the internal cell mapping and local switch routing). The input and output wiring is shown in Figure 13, Figure 14, Figure 15, and Figure 16. Note that two wire bundles will either be on the right or left of the global switch—not both—as represented by the dotted lines.

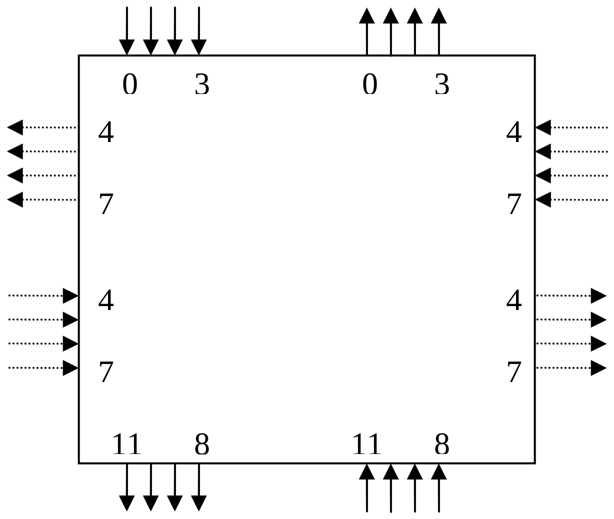


Figure 13. Global switch wiring (oriented in vertical plane)

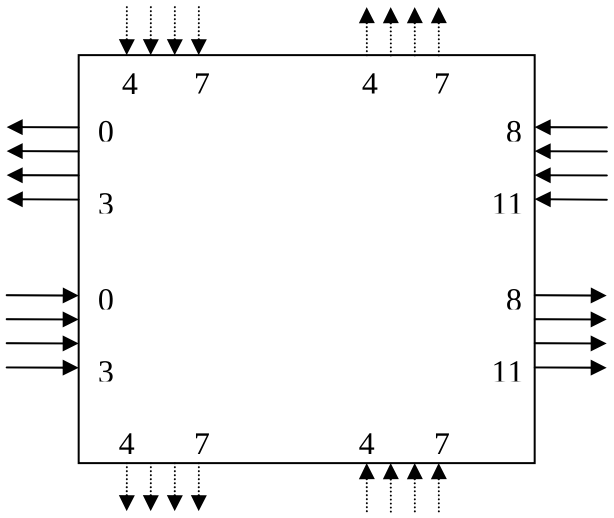


Figure 14. Global switch wiring (oriented in horizontal plane)

The positions of these wires directly correspond to the positions used with each other and for local switches. For more information on this mapping, see Figure 17, which has global switch to global switch mappings. To interface with the local network in the

Manhattan architecture, wires 0-3 or 8-11 communicate with wires G0-G3 of a local switch. The Washington Architecture connects the global switches directly to cells using bits 12-15 of the cell's input/output bitmap.

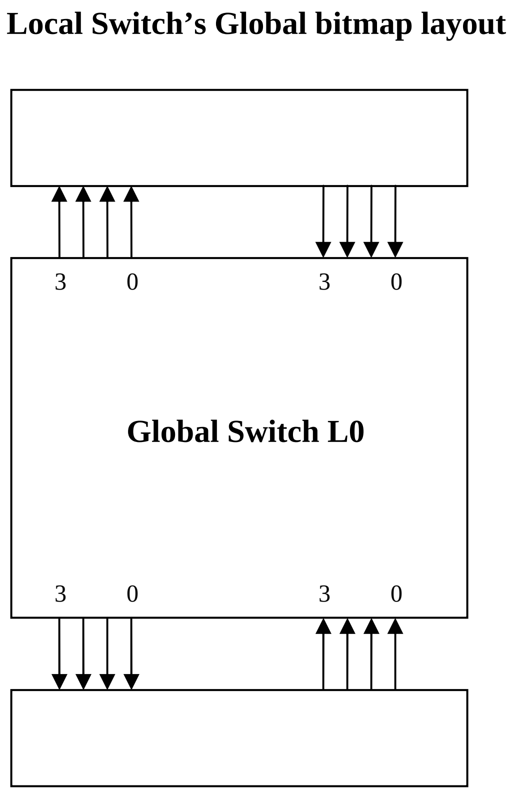


Figure 15. Interface from local switch to global switch in Manhattan architecture

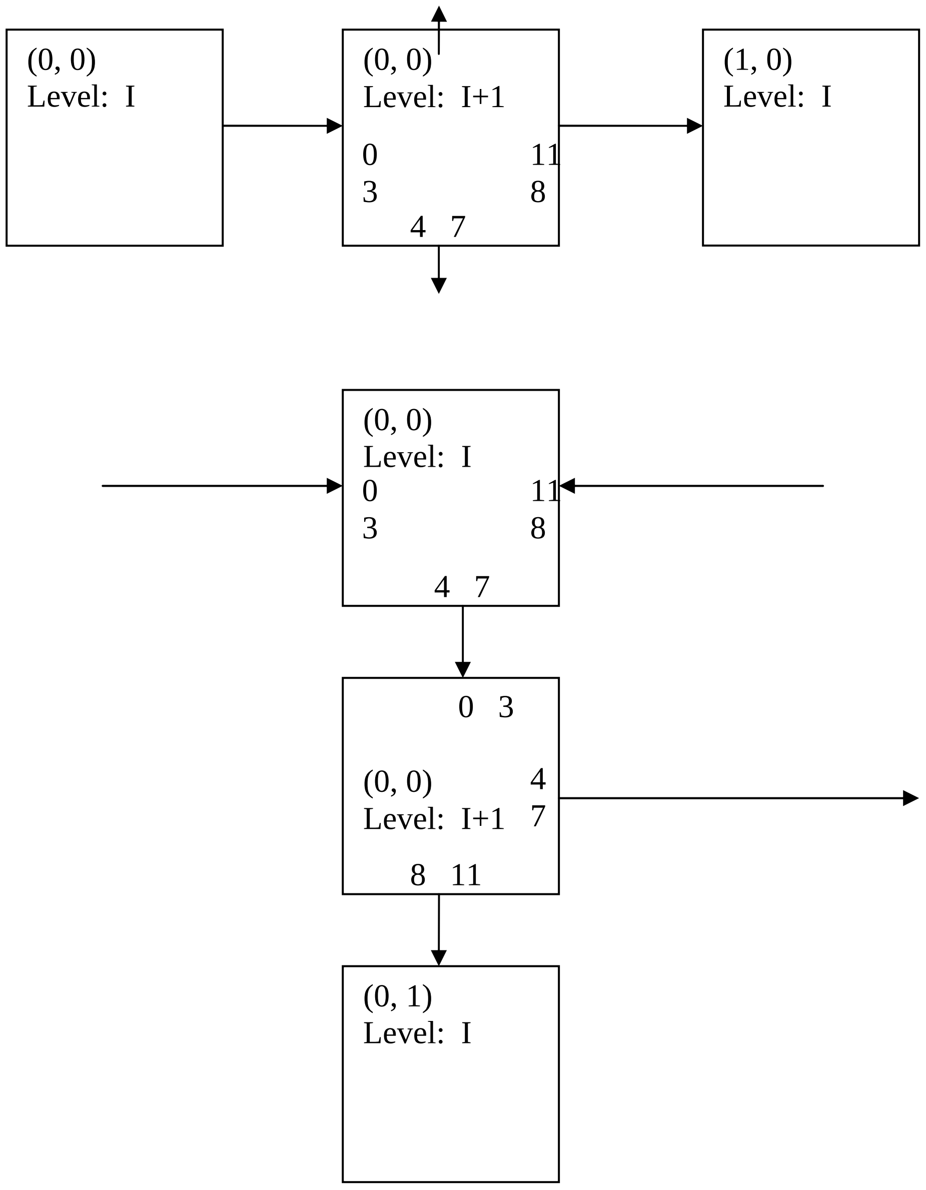


Figure 16. Relational layout of global switches

Global switches are similar to local switches in the way their routing tables are built. Because a switch has no processing capabilities within the software, the routing tables are the only logic applied to the data. There are 12 routing table entries for any global switch.

Table 7 shows the format of a routing table entry. Table 8 provides the description of the fields in this bitmap. Finally, Table 9 depicts the container holding the bitmaps.

Table 7. Routing table entry for global switch

A	R	R	R	R	R	R	R	B	B	B	B	B	B	B	B
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
B	B	B	B	O	O	O	O	O	O	O	O	O	O	O	O
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 8. Fields in routing table entry for global switch

Field	Definition
A	Specifies whether the input line is active.
R	Reserved for future use.
O	Maps the input line to the output wire that is the same number as the bit position of this output bit.
B	Paired bit with the output bit. If flipped, transfers to upper half of the output bus rather than the lower half.

Table 9. Data structure of global switch

Data Field	Comment
Wires [0-11]	16-bit bitmap for inputs into the global switch.
Unmapped	Unmapped outputs.
Type	WiresLeft = 0, WiresRight = 1. Specifies whether the Right/Left bundle for this global switch are to its right or left.

2.3.4 Simulation Design

The simulator's instantiation of the global network adheres to the same principles as the cell simulator. Wherever a latch exists in hardware, memory must be allocated in software to mimic the hardware. For a single cycle, this means that the single highest level global latch in a Washington Architecture will consume 4 KB alone. Because the Manhattan architecture uses one less layer of global switches, its highest level global latch only consumes 2 KB. The global network is responsible for a large amount of the memory usage during simulation. This is unavoidable as the simulations mimic the hardware's behavior. Extra caution should be used if the global network were to expand to fit a larger sized array.

2.4 Local Switches (Manhattan Architecture Only)

HiPerCopS Manhattan uses a local network that incorporates local switches for routing data between neighboring cells. These switches do not perform any operation on the data other than routing it to the correct destination. The following section specifies how local switches are built in the designer and how they route wires from one location to another.

2.4.1 Switch Wiring

The wiring of a local switch is defined as shown in Figure 17. The numbering of these wires directly corresponds to the numbering used for cells.

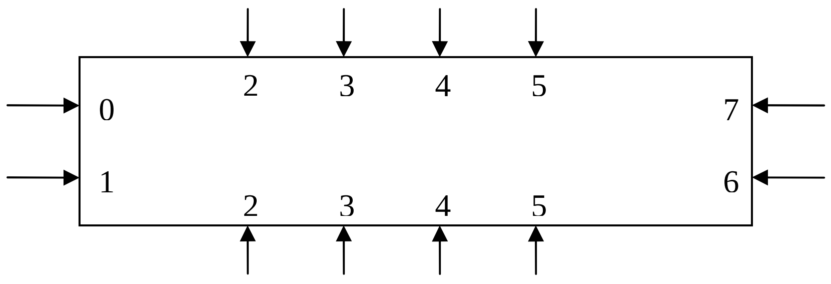


Figure 17. Wiring of local switch

Local switches work in a similar manner to the routing tables used in cells. However, because a switch has no internal logic, there are only nine entries in the routing table.

Furthermore, the designer uses an additional bitmap to keep track of unconnected wires. This is known as the Unmapped Outputs bitmap.

Table 11. Fields in routing table entry for local switch Table 10 shows the routing table entry used for wires coming into the switch. Table 11 describes the fields in this bitmap. Finally, Table 12 shows the container data structure that holds the bitmaps.

Table 10. Routing table entry for local switch

a	r	r	r	G3	G2	G1	G0	o	o	O	o	o	o	o	o
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 11. Fields in routing table entry for local switch

Field	Definition
a	Specifies whether the input line is active.
r	Reserved for future use.
o	Maps the input line to the output wire that is the same number as the bit position of this output bit.
G0-G3	Specifies an output to the global switch connected to this local switch. Not used for switches that cannot connect to the global network.

Table 12. Data structure for local switch

Data Field	Comment
Local [0-7]	16-bit bitmap for inputs from local network.
Unmapped	Unmapped outputs.
Type	Horizontal = 0, Vertical =1.
Global [0-3]	***Pointer*** to four bitmaps that let wires connect to the global network.

2.4.2 Local Switch to Global Switch Mapping:

The designer must perform two types of mapping when connecting a local switch to a global switch. First, the software must determine which global switch is connected to the local switch. This value can be calculated using the local switch's coordinates (*Lx*, *Ly)* using the algorithm in Figure 18. Second, the software must map the wire positions on the local switch to the wire positions on the level 0 global switch. Figure 19 defines this mapping.

```
//Lx = Local Switch x coordinate
//Ly = Local Switch x coordinate
//Glblx = Local Switch x coordinate
//Glbly = Local Switch x coordinate

if(Lx%2!=0)
{
  //No global switches connect up to
  return -1;
}
//Find the x,y for the global switch
Glblx = Lx/2;
Glbly = Ly/2;
```

Figure 18. Code for Local Switch to Global Switch Mapping

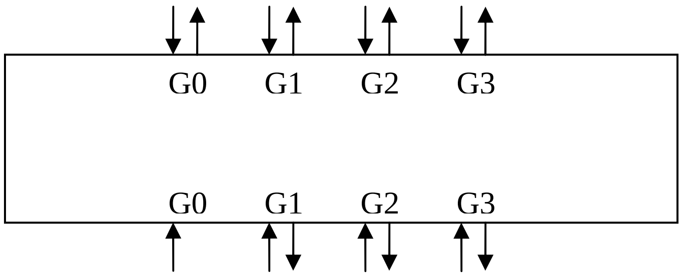


Figure 19. Local Switch to Global Switch Mapping

2.4.3 Simulation Design

Though the local switches exist in hardware, the simulator effectively bypasses these components because they contain no latches. Data leaving a cell is instantly propagated to its destination latch during simulation.

CHAPTER 3

Simulator Design

There are two major areas of simulation that take place in the HiPerCopS CAD tools. The first is the cell-level simulation, which pertains to the 4x4 matrix of elements within a cell. Here, the simulator drives the inputs of the elements and computes the required memory and mathematics functions. The second is the system-level simulation, which is responsible for propagating the simulated data to appropriate registers elsewhere in the array. The following sections cover both of these concepts in detail.

3.1 Cell Level Simulation

As described in Chapter 1, cells have two operating modes: mathematics mode and memory mode. Both modes take in four 4-bit inputs and generate two 4-bit outputs.

3.1.1 Mathematics Mode

True element-level simulation takes place for mathematics mode. The elements for mathematics mode are wired as shown in Figure 20.

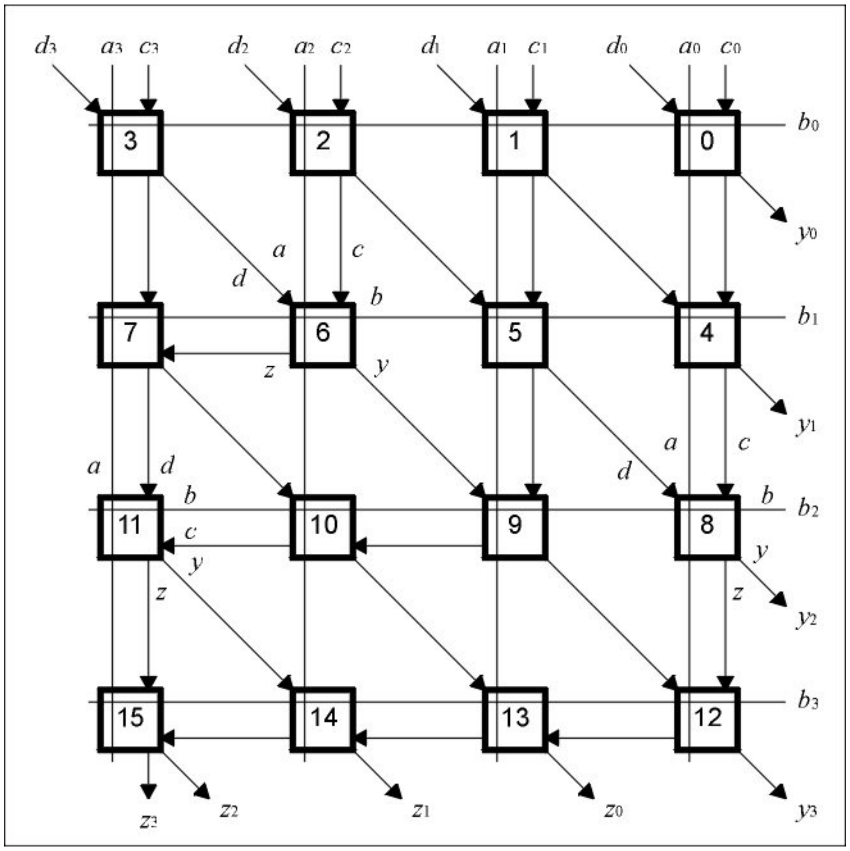


Figure 20. Matrix of elements in mathematics mode

The primary driver function used for simulating a basic cell uses fast computational procedures wherever possible, such as shifting and other basic logic functions. The basic algorithm used is as follows:

1. Bitmask off each of the 4-bit inputs – break inputs off to truth table patterns.
2. Simulate each truth table for every element. Start with Element 0 and iterate through 15.
3. Propagate data to next stage in the array before simulation of a dependent stage.
4. Bitmask and OR the outputs together into a single byte.

The important element of the above process is verifying the connections between the elements and simulating the elements in the correct order. For example, in the figure above, element 5 cannot be simulated until elements 1 and 2 have both finished and have a result ready. Furthermore, because the system has to run this simulation for every cell that has valid inputs, it is imperative that these simulations run at the highest speed possible.

3.1.2 Memory Mode:

Because of memory mode's conventional behavior, it is simulated on the behavioral level. In other words, its function was abstracted to the cell level. The truth tables were expanded into a single 64-entry truth table that can take a byte of data for each entry. The layout of a memory cell is shown in Figure 21.

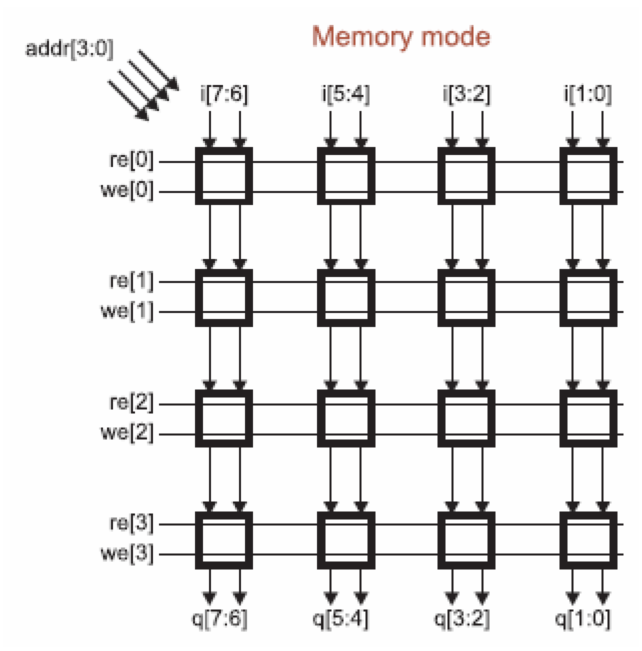


Figure 21. Matrix of elements in memory mode

3.2 System Level Simulation

There are several stages to simulating a full system design. The system level simulator works as a large state machine that calls upon the cell-level simulator object. The basic stages to simulation are described in Figure 22.

High Level View of Simulation

1. Initialization
    a. Allocate cell output pipeline latches
    b. Transfer memory mode data into latches
    c. Set up initial data input feed
2. Simulation (*NS = Next State; PS = Prev. State)*
    a. MemCell(*NS*) = MemCell(*PS + Input Logic*)
    b. MathCell(*NS*) = MathCell(*PS + Input Logic*)
        i. References to design structure for truth tables
        ii. Math cell emulation performed
3. Propagation
    a. Trace output to destination
    b. Translate output wire to input wire
    c. Copy data to input latch of next logic/control element
    d. Global network latch propagation

As each output is propagated, the IsDefined is propagated along with the data.

Figure 22. Simulation procedures

Stage 1 is performed at the beginning of every cycle to set up all of the necessary data structures for the current simulation. This step includes mid-simulation re-programming that may need to be done on the fly. Stage 2 is performed almost entirely at the cell simulator level. Stage 3 is done in conjunction with the Trace support structure. Once the simulated data is obtained, its trace can be called to instantly transport that data to its destination spot in memory for the next cycle.

3.3 Pipeline Latches

In a pipelined architecture such as the HiPerCopS architectures, it is imperative that data be synchronized at certain stages within a DSP operation. To satisfy this requirement, all cell outputs have an outgoing latch which can be used to delay a 4-bit result for up to 3 cycles. Every latch that is allocated on a cell output requires memory to be allocated for it. The simulator mimics the hardware with respect to the timing of clock cycles when dealing with these latches.

3.4 Behavioral Simulation

If the logic of a particular block of cells can be abstracted to a given math function, the simulation could be performed on the behavioral level. Currently, this ability is not available in the CAD tools, but the interfaces currently exist to allow for this feature to be added. For example, if a block of cells behaved like a multiplier and had a known propagation delay, the simulator could be programmed to simulate the block of cells behaviorally. This abstraction would greatly speed up the simulation process.

CHAPTER 4

Software Support Structures

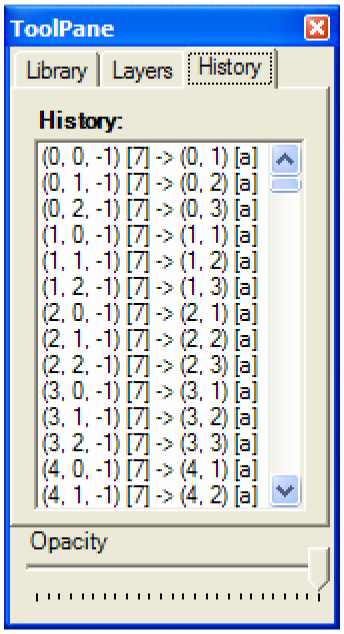
The software support structures are necessary features that assist in the software's necessary functions. These structures interface with both the back end and the front end to add required functionality to the tools.

4.1 Back End

Developing the CAD tools to map designs onto the reconfigurable architecture created a need for various other support structures within the software. These structures help expedite simulations, keep the code clean, and add higher-level functionality for the user. There are two main support structures within the CAD tools: Traces and Part Libraries. The first, Traces, tracks every user action into a history, which can be used for wire deletion and other undo functionality. The second, Part Libraries, provides a way to program one cell and apply that programming to other cells within the array, creating the concept of "parts" within the CAD tools. The part library provides a basic structure that can be programmed by the user and stored for later use.

4.1.1 Traces

Taking cues from other design tools, we added a history feature to the software. This history serves several purposes: allowing users to undo their actions, abstracting the routing between cells to a higher level, and increasing the performance of the simulator. The Trace class encapsulates a linked list that tracks every action that the user has taken during the mapping process. These actions are also serialized and saved into the files for permanent storage along with the actual design, so that a user can load the history and design together from file. Each action is recorded in the history log as it occurs, so the most recent events appear at the bottom while the earliest events occur at the top. A screenshot of the History window is shown in Figure 23.



```
(0, 0, -1) [7] -> (0, 1) [a]
(0, 1, -1) [7] -> (0, 2) [a]
(0, 2, -1) [7] -> (0, 3) [a]
(1, 0, -1) [7] -> (1, 1) [a]
(1, 1, -1) [7] -> (1, 2) [a]
(1, 2, -1) [7] -> (1, 3) [a]
(2, 0, -1) [7] -> (2, 1) [a]
(2, 1, -1) [7] -> (2, 2) [a]
(2, 2, -1) [7] -> (2, 3) [a]
(3, 0, -1) [7] -> (3, 1) [a]
(3, 1, -1) [7] -> (3, 2) [a]
(3, 2, -1) [7] -> (3, 3) [a]
(4, 0, -1) [7] -> (4, 1) [a]
(4, 1, -1) [7] -> (4, 2) [a]
```

Figure 23. History window

The history itself is composed of a list of Trace objects. Figure 24 shows a graphical representation of how the history is managed:

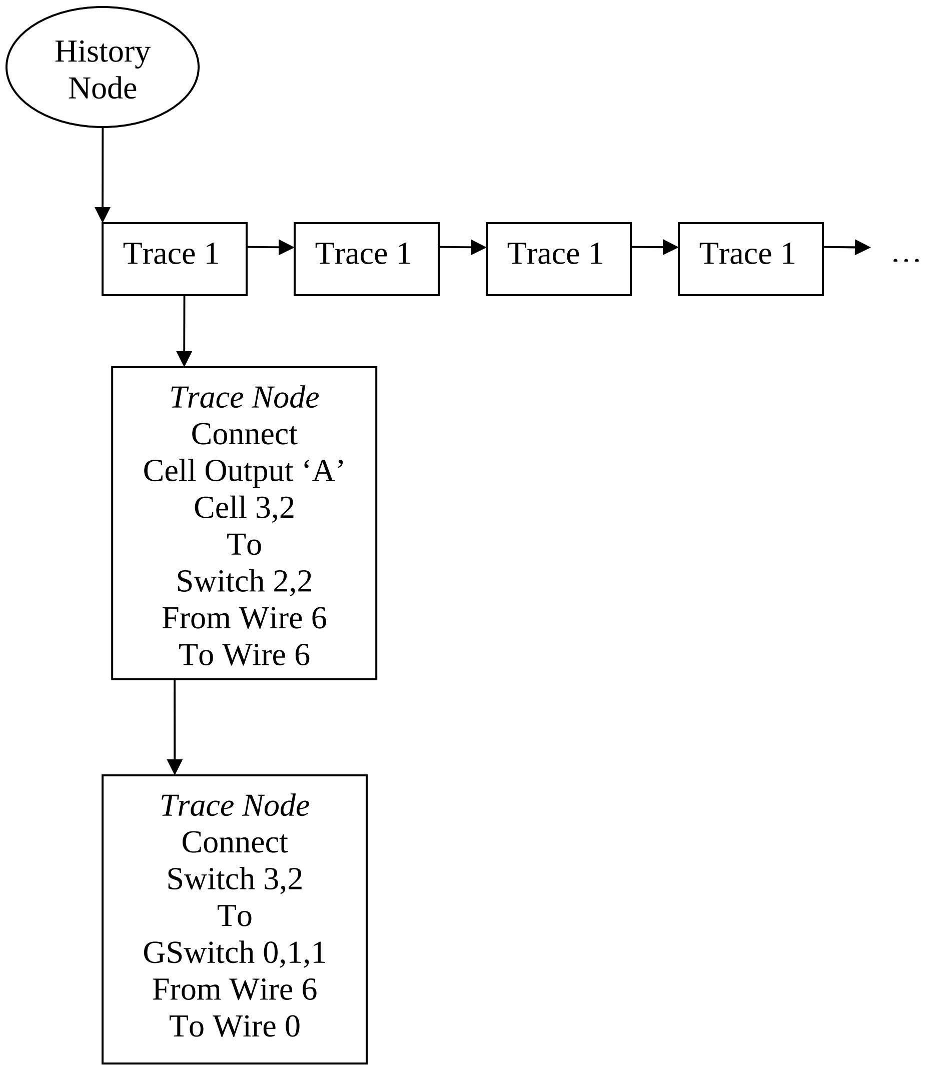


Figure 24. Example of history, traces, and trace nodes

Notice that traces always end on a data latch. Thus, the simulator can simply skip to the end of a trace to propagate the data from source to destination. This optimization greatly accelerates the performance of the simulator. In the Manhattan architecture, a trace could jump across three connections before reaching its final destination, as shown in Figure 25. Hence, the simulator would have to perform a series of table lookups to translate the wire position to the next component—two translations per cell and one per switch. These actions are recorded within the Trace object.

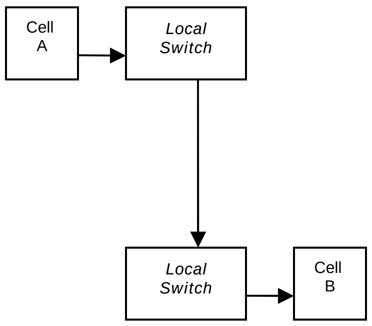


Figure 25. Worst-case connection in the Manhattan architecture

In the Washington architecture, the implementation of traces has less value, because the simulator can compute the destination with only three table lookups for a cell-to-cell connection. However, because the new architecture was based off of the old architecture, much of the code base was inherited by the new architecture. The ability to delete wires easily was required in both architectures regardless of simulation needs.

The downside to using these traces are memory usage and navigating through the traces when there are dense sets of wires existing on the schematic. For future work and expediting the implementation of traces, it would be good to add a reference to the trace from within the originating component (cell or global switch). This would greatly speed up the performance over the current model, which uses one master history list.

The Trace class has become a very important class within the designs of both architectures. It allows the easy movement across a wire from beginning to end and keeps a record of everything done on the wire. The delete and undo features find their roots through the traces. It proved so useful that it was used in the early version of the simulator for tracing outputs to their destination latch.

4.1.2 Part Libraries

The part window is familiar for any CAD tool users as it provides the elemental components with which to build a system. Similarly, the HiPerCopS CAD tools have such a window for the storage of the cell configurations. A screenshot of the window is shown in Figure 26. The cell configurations are stored in a linked list that record complete cell configurations down to the element lookup tables. The configurations are then programmed into cells on the array, which effectively copies the configuration onto that cell. Because of this copy relationship, the cell can be further modified after programming (to add extra pipeline latches, for example) without affecting other cells of the same type.

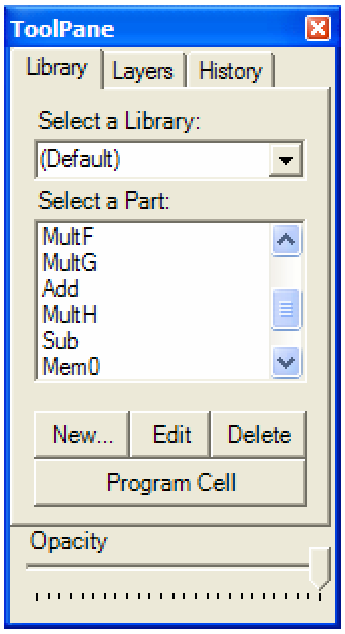


Figure 26. Screenshot of Part Library

Future versions of the software should support parts at a much larger scale. Currently, one can build pieces of a multiplier or pieces of any generic part. Eventually, the software should also allow for the design of parts incorporating a set of cells. In this way, a designer could place an entire multiplier and all the routing and element programming would be complete. This functionality could also be extended to simulations, as a part such as a multiplier could have a simulation behavior associated with it to speed up simulation. The primary concern with this upgrade is the positioning of the global network with respect to the parts, and defining the part inputs and outputs. Furthermore, if the part is modified in any way after placement, all expedited simulation models would have to be discarded.

4.2 Front End

This section is dedicated to the user interfaces developed for the HiPerCopS CAD tools.

4.2.1 Rendering Engines

The basic structure for the CAD tools is shown in Figure 27. Though each version of the CAD tools uses a completely different rendering method, the basic responsibility for the engine is the same for each version. The rendering engine is responsible for displaying the routing data stored in the instantiated cell array object in an informative and usable fashion. It is important to balance all of the following performance aspects in the rendering engine:

- Render Speed: The rate at which the rendering engine can respond to user input.

- Visual Comprehension: The interface must make intuitive sense and not be over-cluttered. It must also display all information about the design of the cell array.
- Usability: The interface must provide efficient entry points and use case scenarios for the design of the system.

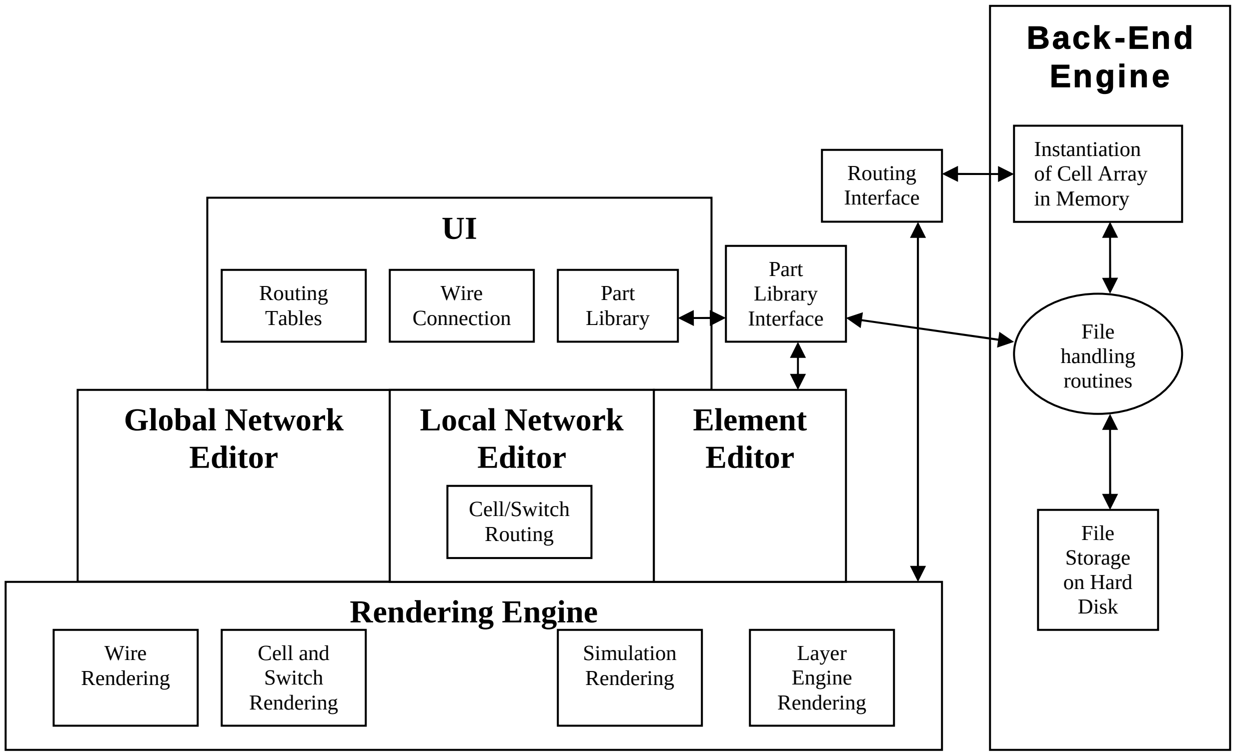


Figure 27. Diagram of system interfaces

4.2.1.1 Shift Array Rendering Engine

The first rendering engine used in the CAD tools was built from an array of compiled User Controls. This was done for the sake of rapid development and for code re-usability. The Manhattan architecture's local network editor is the only rendering engine that uses this method. It provides the fastest redraw speeds because the operating system optimizes its graphics draw routines on the user controls, however, this rendering engine provides the least in terms of flexibility. This is because every user control is responsible for its own drawn area. It becomes very difficult to overlay images that traverse across a portion of a user control. Drawing an image across several user controls presents a large processor heavy challenge.

4.2.1.2 Basic GDI+ Global Rendering Engine

After the development of the Shift Array Rendering Engine, a better engine was developed to handle the cases where wires would need to be drawn over top of current cells and routing wires. This rendering engine was developed using the basic features of GDI+ and ran through the cell array instantiation drawing everything onto a back buffer, which was then flashed to screen. This proved to be a much slower rendering agent, but a lot more flexible.

4.2.1.3 Washington Rendering Engine

Finally, when the development of the Washington Architecture was chartered, a final rendering engine was developed. Based off of some of the principles used in the Basic GDI+ engine, this much more object-oriented engine also incorporated features such as matrix transformations and layered architecture. Matrix transformations allow for the array view to be resized on demand, while the layered architecture views the drawing as a series of drawings put on top of one another. These offer additional flexibility and assist in the design-making process. Unfortunately, this is currently the slowest rendering engine as it has a lot of drawing overhead. Future work could look at ways to optimize this engine to increase its performance.

4.2.2 Software Differences

There are several areas in which HiPerCopS Washington differs from HiPerCopS. First, the new interconnection network incorporates an additional layer of global switches, since the local switches are effectively removed. This extra layer causes an extra 2-cycle delay for all communication through the global network. It also effectively doubles the bandwidth of the global network, allowing for much more complex designs. Furthermore, the switching mechanisms in the newer architecture are more complex, and incur additional delay versus the same switches in the older architecture. However, the absence of the local switches reduces the overall complexity of the system.

Migrating the software to the newer platform highlighted all of these differences. First, the local switches had to be removed from the local network. A change this drastic warranted a re-design of the graphics engine driving the CAD tools. Instead of using an array of dynamically updating user-controls, a blank draw surface was used. To display a design on the screen for the new layout, the CAD tools draw the representation in layers on a back buffer which is then sent through a transformation matrix before being flushed to the screen. This also creates a flicker-free double buffered environment similar to the environment used in the old architecture.

CHAPTER 5

Performance Analysis

5.1 Layout of DSP on Cell Array

With all the foundation built and the CAD tools operational, we implemented several DSP benchmarks on the HiPerCopS Washington and HiPerCopS Manhattan architectures. These benchmarks included a 12-tap Finite Impulse Response (FIR) filter and a 256-point Fast Fourier Transform (FFT). Each algorithm was designed from the modular level down to the individual elements.

5.1.1 Twelve Tap FIR Filter

The first DSP algorithm we implemented was a FIR filter, which appears in Figure 28. As can be seen in the diagram, the filter is comprised of multipliers and adders operating in parallel. A diagram showing a suitable layout for the HiPerCopS CAD Tools is shown in Figure 29.

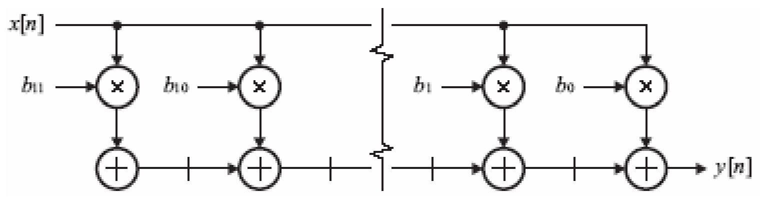


Figure 28. Diagram of FIR filter [10]

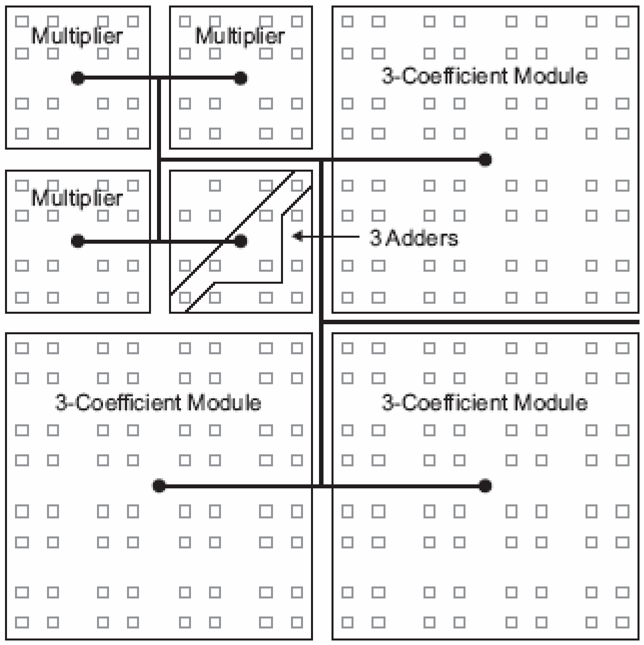


Figure 29. Modular implementation of FIR Filter [10]

To create a higher-order filter, one could simply add more modules. The pipelined architecture of the design will also produce outputs in the same order that the inputs are fed. Finally, extra pipeline latches within the design allow for time synchronization on all of the outputs. A sample local network view of the whole FIR Filter as built in the HiPerCopS Washington architecture appears in Figure 30. A standalone 3-Coefficient Module also appears in Figure 31.

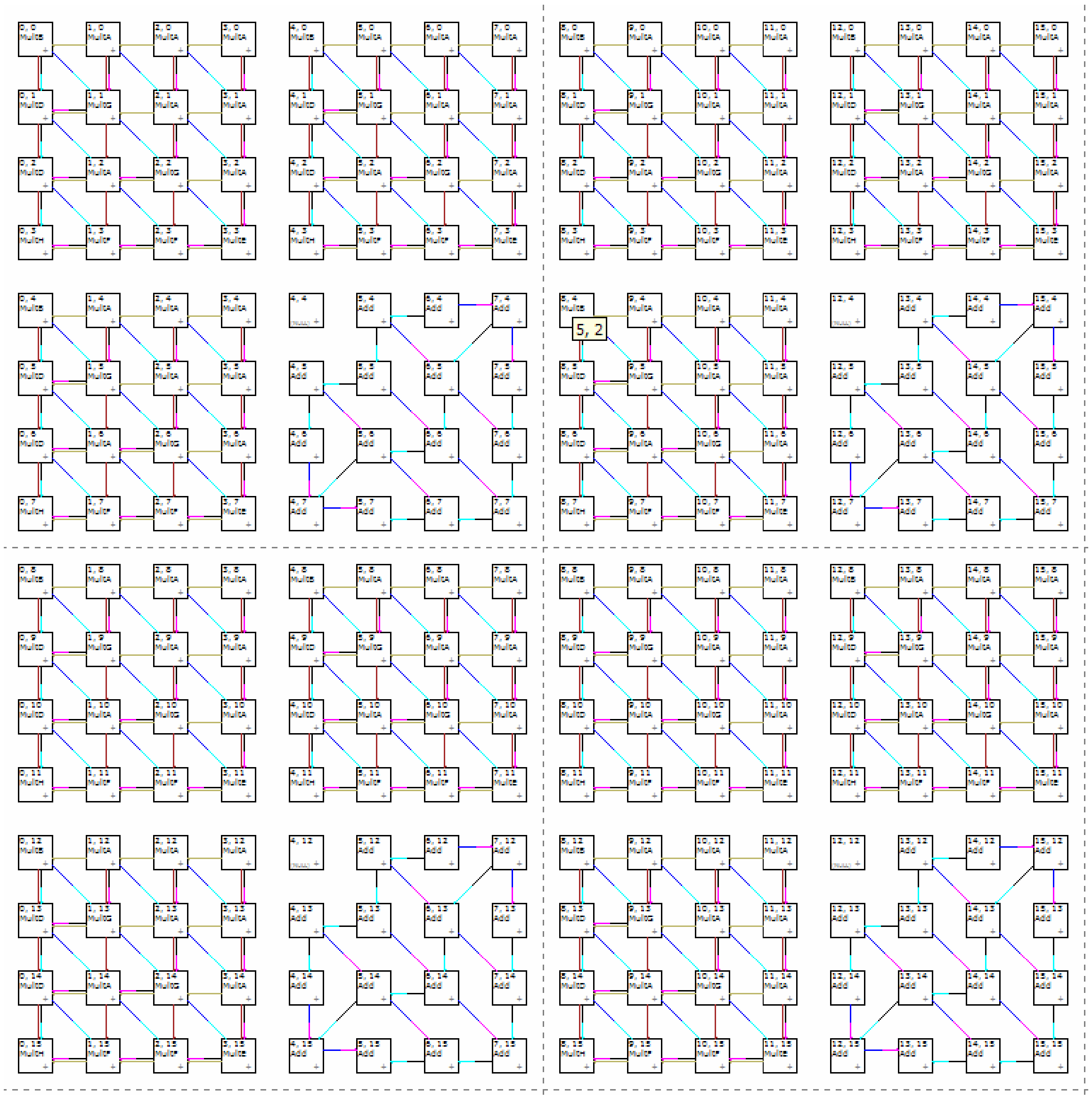


Figure 30. Screenshot of Local Network of FIR Filter Implementation

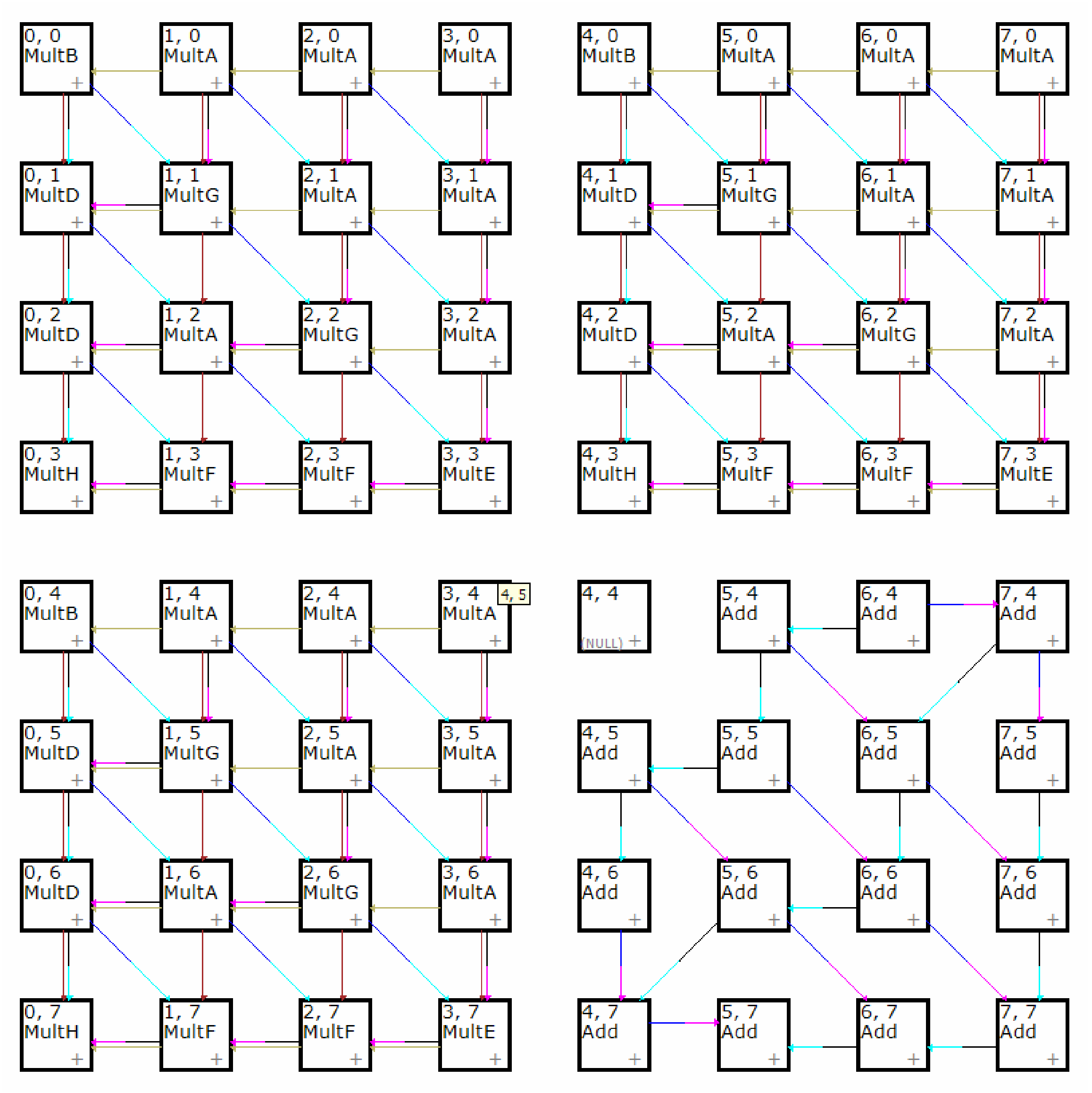


Figure 31. Screenshot of Local Network of a 3-Coefficient Module for FIR Filter Implementation

5.1.2 256 Point FFT

The second benchmark algorithm that we implemented was a 256 Point FFT, which appears in Figure 32.

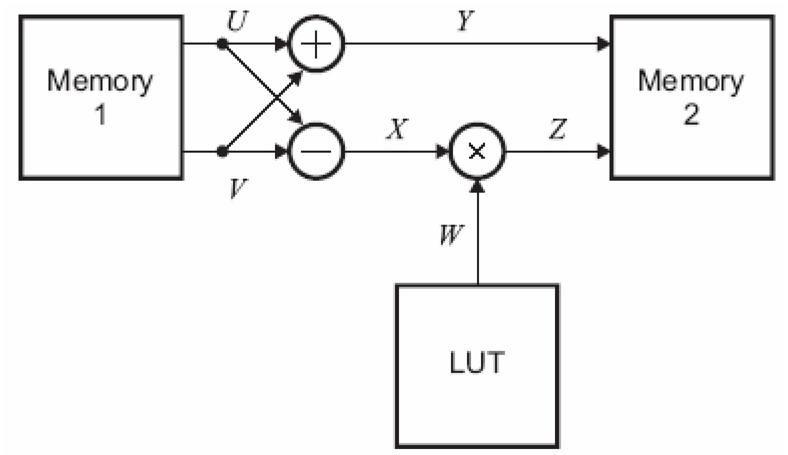


Figure 32. Diagram of FFT [10]

This algorithm is more complex and more diverse in its layout as it uses cells in memory mode. The FFT demonstrates the implementation of several major structures, such as a multiplier, subtractor, adder, and its memory core. It also heavily uses the global interconnect, as many busses have to be transported across the array. In fact, the design led to the development of the Washington architecture, which doubles the capacity of the global network. The FFT was implemented onto the array of cell as shown in Figure 33.

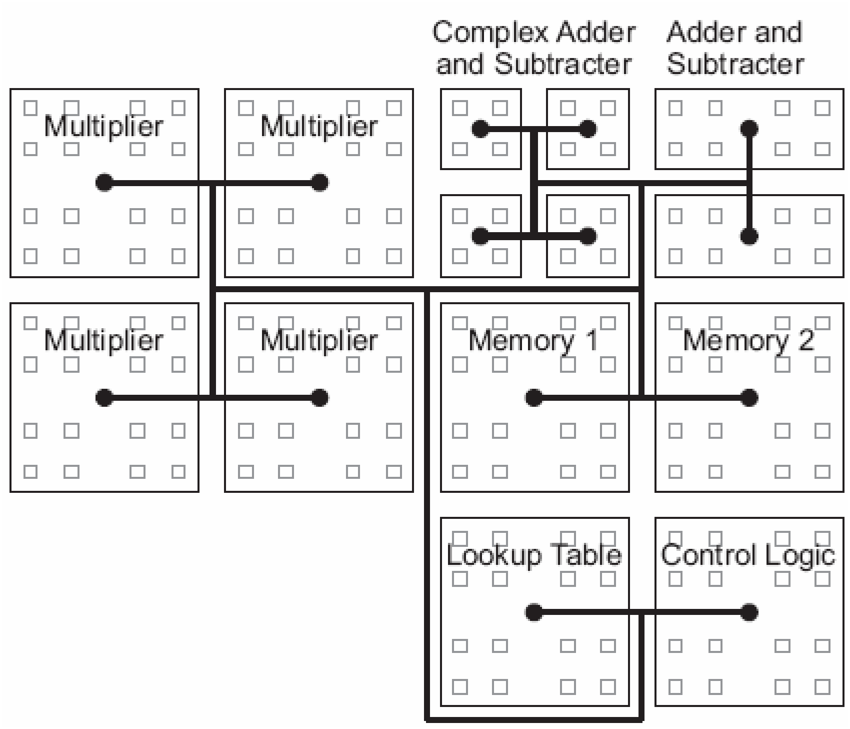
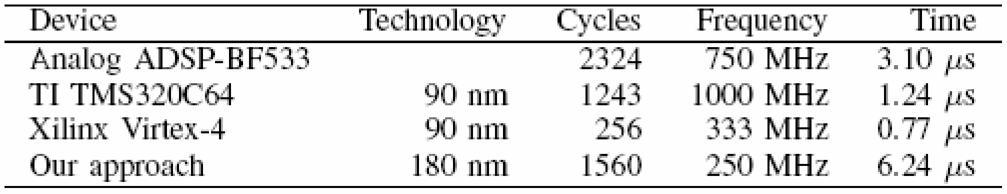


Figure 33. Implementation of FFT

After simulation, we discovered that our architecture has a latency of 57 cycles between the two memory units. Each processing stage handles 128 pairs of samples, for a total of 185 cycles. Taking the runtime reconfiguration, the FFT requires 1560 cycles, or 6.24 micro-seconds. This execution time is greater than other implementations as shown in Table 13, but these other approaches use a more sophisticated radix-4 technique to reduce the number of computations required. We estimate that a radix-4 FFT would require four times the area but lower the execution time to less than 2 micro-seconds [10].

Table 13. Execution Time of 256-Point FFT



Device	Technology	Cycles	Frequency	Time
Analog ADSP-BF533		2324	750 MHz	3.10 μs
TI TMS320C64	90 nm	1243	1000 MHz	1.24 μs
Xilinx Virtex-4	90 nm	256	333 MHz	0.77 μs
Our approach	180 nm	1560	250 MHz	6.24 μs

5.2 Software Performance

5.2.1 Memory Usage

Memory usage is always a critical statistic when setting software requirements. Basically, the CAD tools allocate a base set of memory for the engine and underlying design data structure, which varies between the Washington and Manhattan architectures. Either way, this memory usage does not change very much and stays at a reasonable value. What is a much greater consideration to memory is the simulation results. Every cycle of simulation requires a large amount of memory from the system resources. The base memory required for simulating a single cycle at debug runtime is 696 KB. This value does not include cells in memory mode or pipeline latches. An additional 66 bytes should be added for every memory mode cell and additional 2 bytes for every additional pipeline latch. Extending these numbers off to a typical system comprised of 25% memory cells and 10% additional pipeline latching, the total memory usage per cycle is shown in Figure 34.

696KB + 67.584KB + 1.6384KB = 765.2224KB
(System) (Memory) (Pipeline)

Figure 34. Memory usage per cycle in a normal system

765 KB may not seem like a significant amount of memory. However, one must take into consideration that to sufficiently test a DSP algorithm, hundreds to tens of thousands of cycles must be simulated to come up with an accurate result. Simulating a thousand cycles on this typical system and storing every bit of data for every one of those cycles would use up 765 MB of RAM.

Currently, the CAD tools store a complete snapshot of the entire cycle that it simulates and stores it into an array. To improve on performance and cut down on memory usage, one could only store data selected to watch. This would force the user to select certain wires that they would want to monitor over the course of the simulation. Each wire would only be on the order of 4 bits, so a selection of 20 wires would only cost the user a total of 10 bytes of memory per cycle. The downside to using a watch feature as described is that if the user wanted to “watch” any other wire that they hadn’t selected, the simulation would have to be redone in its entirety.

5.2.2 Simulation Performance

It is difficult to derive a metric for simulation performance as every design will produce a different simulation behavior and timing scheme. Not only do different designs produce different simulation speeds, but simulation speeds vary at every cycle due to the optimizations programmed into the simulator. In particular, the simulator uses the IsDefined bitmap associated with cell latches to bypass simulation if one of the necessary inputs is undefined. Because large portions of the array may not be used during the first few cycles of the simulation, there is no need to simulate garbage data in these unused portions. Furthermore, to expedite memory mode cells, a separate part of the simulator was developed just for the simulation and update of the memory tables. All memory cells are thrown into a list at the beginning of a simulation cycle. During simulation, the simulator runs through the list and updates the memory values for all the memory cells in one quick traversal.

CHAPTER 6

Comparison of Architectures

6.1 Manhattan Architecture

The Manhattan architecture has provided a solid basis for testing design principles and routing mechanisms. From an architectural standpoint, the Manhattan architecture's primary limitation was found to be its global network. Extremely complex designs requiring a lot of dense routing cannot be built easily on such an architecture. However, this model does provide the benefit of fast propagation across the array of cells.

Looking at the basic local network model, the cells transfer data through local switches which does not exist in the Washington architecture. One of the factors taken into account for determining the minimum clock cycle period in the Manhattan architecture is the worst-case delay between latches in the architecture. Because of the presence of the local network, the worst-case delay scenario, shown in Figure 35, has four traversals to complete during a single clock cycle. The output signal must propagate through the output switch of the source cell, two local switches, and the input switch of the destination cell. This means that each of the four switches must have an average propagation delay of 1 ns to provide a clock rate of 250 MHz.

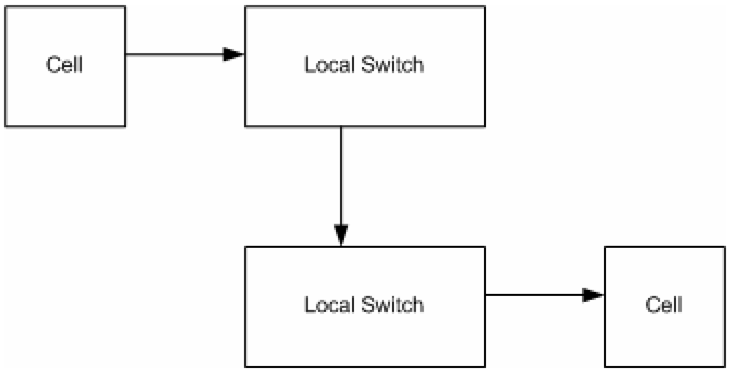


Figure 35. Worst-case delay scenario for Manhattan architecture

From a software standpoint, the implementation of the Manhattan architecture was far more complex than that of the Washington architecture. The need for a local network created the need for many different routing routines and tables that were not needed in the other alternative. However, because the local network was not latched with the clock, the simulator was not affected by the presence of the local network.

6.2 Washington Architecture

The Washington architecture has several advantages over the Manhattan architecture as it offers more routing features. In particular, one more layer was added to the global network and the local network was removed. This effectively doubled the available lines in the global network. To account for the loss of the local network switches, four additional routes were added to allow "diagonal" routing between cells. Furthermore, the cells are connected directly to the global network. This impacts the design because the input/output switches of the cells are two times larger than their equivalent switches found in the Manhattan architecture. For example, the input/output switch for a

Manhattan Cell is show in Figure 36. The equivalent input switch for a Washington architecture cell is shown in Figure 37.

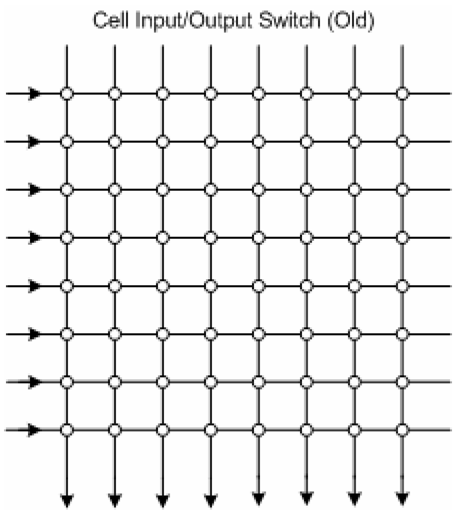


Figure 36. Internal switch used in Manhattan architecture

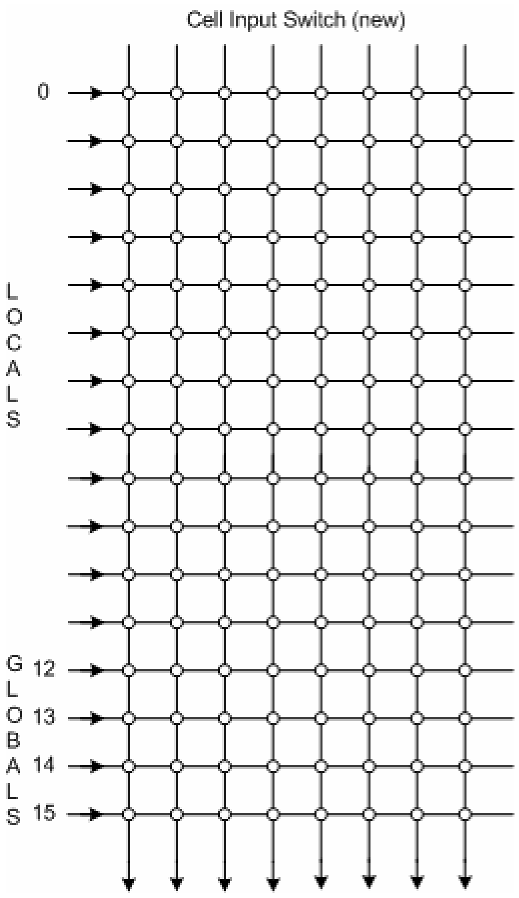


Figure 37. Internal switch used in Washington architecture

CHAPTER 7

Conclusion

This thesis has presented CAD tools for the design of DSP on the HiPerCopS two-level reconfigurable architecture. A robust simulator for both the Manhattan and Washington architecture were built and tested. The designs in software allow for optimizations in design and create a viable workspace for synthesizing DSP algorithms. Furthermore, the simulations provide sufficient information to qualify benchmarks and give a reasonable idea as to the performance of the hardware at clock-level granularity.

7.1 Contributions

Several full-featured systems were built during the course of this research to support the reconfigurable architectures tested:

- *Developed flexible mapping tool designer.* A novel CAD tool was developed for both the Manhattan and Washington architectures. This tool allows for schematics to be designed in a 64x64 reconfigurable cell array. The software structure is flexible enough to allow for new architectures to be implemented in the future.
- *Developed a reconfigurable array simulator.* The simulator works with the designer to test and verify designs made in the software. The simulator takes a state-based approach to the models and runs each simulation as it should occur in hardware. Useful data and metrics, such as the execution time, can be obtained from the simulator.

- *Mapped and simulated two benchmarks.* We verified the software's functionality by successfully implementing an FIR filter and an FFT in the CAD tools. Each algorithm was mapped onto the cell array and then simulated. Each of these benchmarks was verified by successfully running actual data through the array of programmed cells and viewing the results.
- *Proposed a new interconnection structure to improve performance.* After viewing the results of the Manhattan architecture, the Washington interconnect structure was developed. This alternative improves upon performance and has significantly improved bus capabilities. The software tools were migrated to support this new architecture as well.

7.2 Future Work

Further research on the development of the CAD tools will focus on several areas:

- *Hardware Assisted Rendering.* Currently the graphics engine for the CAD tools is done without any hardware support, which can lead to slow draw times when displaying the whole cell array. Implementing draw routines by utilizing a hardware graphics card will greatly improve performance.
- *Auto Place and Route.* Another useful feature that we are looking to implement in future version is automated routing. Extending this feature further, we are looking to develop a format for downloading designs into hardware.
- *Component Level Design.* Extending the part library to be able to design whole multipliers, adders, and other parts is a pivotal feature for future development. A set of standard parts could be built and distributed with the software so that designers could design with an object oriented approach.

- DSP Mapping Translation: Perhaps the most complex feature that we are looking to develop is a script translator. The ability for a designer to simply write a script that will automatically program the cell array with the correct DSP would be very valuable.

References

[1] R. Hartenstein, “Coarse grain reconfigurable architectures,” in *Proc. 6th Asia South Pacific Design Automation Conference*, Yokohama, Japan, pp. 564-570, 2001.

[2] C. Ebeling, D. Cronquist, P. Franklin, and C. Fisher, “RaPiD—a configurable computing architecture for compute-intensive applications,” *University of Washington Department of Computer Science & Engineering Tech Report TR-96-11-03*, Nov. 1996.

[3] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, “Using the KressArray for reconfigurable computing,” *Proc. SPIE*, vol. 3526, pp. 150-161, Oct. 1998.

[4] H. Zhang et al, “A 1-V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing,” *IEEE J. Solid-State Circuits*, vol. 35, iss. 11, pp. 1697-1704, Feb. 2000.

[5] P. Heysters and G. Smit, “Mapping of DSP algorithms on the MONTIUM architecture,” in *Proc. International Parallel and Distributed Processing Symposium*, pp. 180-185, Apr. 2003.

[6] M. Myjak, “A two-level reconfigurable cell array for digital signal processing,” M.S. thesis, Washington State University, May 2004.

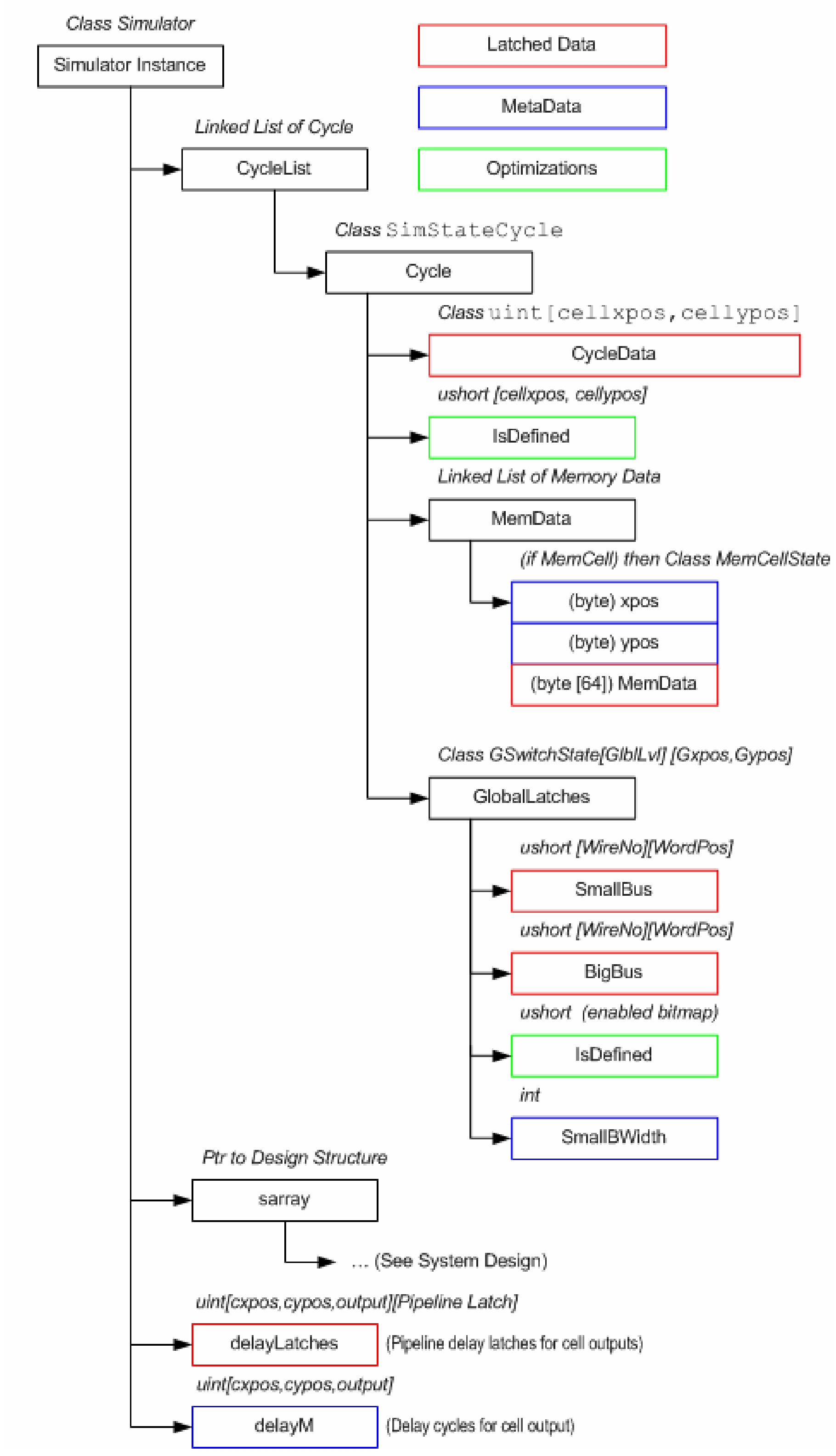
[7] M. Myjak, F. Anderson, and J. Delgado-Frias, “H-tree interconnection structure for reconfigurable DSP hardware,” in *Proc. 2004 International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, pp. 170-176, Jun. 2004.

[8] M. Myjak and J. Delgado-Frias, “A two-level reconfigurable architecture for digital signal processing,” in *Proc. 2003 International Conference on VLSI*, Las Vegas, NV, pp. 21-27, Jun 2003.

[9] M. Myjak and J. Delgado-Frias, “Superpipelined reconfigurable hardware for DSP,” *IEEE International Symposium on Circuits and Systems*, May 2006, submitted.

[10] J. Larson, M. Myjak, and J. Delgado-Frias, “Mapping and performance of DSP benchmarks on a medium-grain reconfigurable architecture,” *IEEE International Symposium on Circuits and Systems*, May 2006, submitted.

Appendix A: Simulator Diagram



Appendix B: Washington Cell to Cell Connection Source Code

```
/// Subroutines for the new architecture
/// </summary>
/// <param name="srccellx">Source Cell X Coordinate</param>
/// <param name="srccelly">Source Cell Y Coordinate</param>
/// <param name="destcellxpos">Destination Cell X Coordinate</param>
/// <param name="destcellypos">Destination Cell Y Coordinate</param>
/// <param name="CellOutput">Source Cell Ouput Line</param>
/// <returns>(ushort)0x0 if success</returns>

     //Subroutine used to connect a cell to a cell
     public ushort ConnectCellToCell(int srccellx, int srccelly, int
destcellx, int destcelly, WireToCell CellOutput, WireToCellInput CellInput)
     {
        int CellWire, availWire;


        //To supress compiler concerns
        availWire = 0xffff;

        //8 directions that we can be wiring to
        //Handle the primary 4 to start with
        //Check Horizontal
        if(srccellx == destcellx)
        {
           //Check vertical
           if(srccelly == destcelly+1)
           {
              //Connecting Upwards

              //Get an available wire
              availWire = CheckAvailInputs(destcellx, destcelly, 4, 5);
           }
           else if(srccelly == destcelly-1)
           {
              //Connecting Downwards
              //Get an available wire
              availWire = CheckAvailInputs(destcellx, destcelly, 0, 1);
           }

        }
           //Check vertical
        else if(srccelly == destcelly)
        {
           //Check vertical
           if(srccellx == destcellx+1)
           {
              //Connecting Left
              //Get an available wire
              availWire = CheckAvailInputs(destcellx, destcelly, 2, 3);
           }
           else if(srccellx == destcellx-1)
```

```
  {
    //Connecting Right
    //Get an available wire
    availWire = CheckAvailInputs(destcellx, destcelly, 6, 7);
  }
}
//Check diagonal
if(srccellx == destcellx-1 && srccelly == destcelly-1)
{
  //Connecting to lower right

  //Get an available wire
  availWire = CheckAvailInputs(destcellx, destcelly, 8, 8);
}
else if(srccellx == destcellx+1 && srccelly == destcelly-1)
{
  //Connecting to lower left

  //Get an available wire
  availWire = CheckAvailInputs(destcellx, destcelly, 9, 9);
}
else if(srccellx == destcellx-1 && srccelly == destcelly+1)
{
  //Connecting to upper right

  //Get an available wire
  availWire = CheckAvailInputs(destcellx, destcelly, 11, 11);
}
else if(srccellx == destcellx+1 && srccelly == destcelly+1)
{
  //Connecting to upper left

  //Get an available wire
  availWire = CheckAvailInputs(destcellx, destcelly, 10, 10);
}
//Actually make the connection now
//Make sure we had a wire available
if(availWire==0xFFFF)
{
  //CheckAvailWires() failed.  Exit out with an error.
  return 0xFFFF;
}

//If we get here, we change the cell's bitmap by
//1) Adding a cell output
//2) Adding a cell input
//3) Adding the trace to the history

CellWire=TranslateCellInputToCellOutput(availWire);

//1) Connect the output line
//Activate the output line and
this.cell[srccellx, srccelly].OutRouteTable[(int)CellOutput] |=
                              (uint)(0x80000000 | (0x1 << CellWire));
//2) Connect the input line
this.cell[destcellx, destcelly].InRouteTable[availWire] =
                         (uint)(0x80000000 | (0x1 << (int)CellInput));
    //3) Finally add the wire traces
    //Start trace from cell
    Trace.CreateTrace(srccellx, srccelly, (int)CellOutput, (uint)(0x1 <<
                                                               CellWire));
    //End trace from cell
    Trace.EndTrace(destcellx, destcelly, availWire, (uint)(0x1 <<
                                                      (int)CellInput));

    //Write the trace out
    return (ushort)0x0;
}
```

Appendix C: Manhattan Cell to Cell Connection Source Code

```
     //Subroutine used to connect a cell to a switch
     public ushort ConnectCellToSwitch(int cellxpos, int cellypos, int
switchxpos, int switchypos, DrawModeStatus.SOrientation SType, ushort
WirePosition, WireToCell CellOutput)
     {
        //Input Parameters
        //cellxpos = Cell's X coordinate
        //cellypos = Cell's Y coordinate
        //switchxpos = Switch's X coordinate
        //switchypos = Switch's Y coordinate
        //Type = Switch's orientation {Horizontal | Vertical}
        //WirePosition = The position of the last wire connected
        //CellOutput = The output line of the processing core of the cell that
          we will be updating

        bool CellIsOnRightAbove;
        int WireNum;
        ushort CellWire, scratch, availWire;
        /*
         * Return Codes:
         * !0 - Success - Returns the SWITCH's last connected input/output
         * -1 - Insufficient wires
         * */

        //Is the Cell on the Right | Above the switch?
        CellIsOnRightAbove=(cellxpos != switchxpos || cellypos !=switchypos);

        //Get an available wire for the switch
        //availWire contains a bitmap which maps out a single bit and returns
          the wire number to use
        //availWire returns the wire open from the SWITCH's point of view
        availWire=AllocateSwitchWire(switchxpos, switchypos,
                                                   CellIsOnRightAbove, SType);

        //Make sure we had a wire available
        if(availWire==0xFFFF)
        {
          //AllocateSwitchWire() failed.  Exit out.
          return 0xFFFF;
        }

        //If we get here, we change the cell's bitmap by
        //1) Adding a cell output
        //2) Cleaning up the unconnected wire bitmap on the switch after we
             are done
        //3) Adding the wire to the switch that is being connected

        CellWire=TranslateSwitchToCell(availWire, SType);

        //Connect the Wire to the processor core
        //Activate the line by OR'ing in a 0x80000000 with the wire position
```

```
//Map to the CellWire that we get after translating the switch's wires
//The bits that we are OR'ing into the OutRouteTable
this.cell[cellxpos,cellypos].OutRouteTable[(int)CellOutput]=
               (uint)(this.cell[cellxpos,cellypos].
               OutRouteTable[(int)CellOutput] |
               (uint)(0x80000000) | CellWire);

WireNum = TranslateBitmapToInt(availWire);
if(WireNum==-1)
{
  //Sent an invalid bitmap in
  return 0xFFFF;
}

//Change the Switch's input lines to connect this line
//1. Activate the line 0x8000
//2. Connect it to the Previous wire we used if there was one

if(SType==DrawModeStatus.SOrientation.Horiz)
{
  this.switches[switchxpos, switchypos].LocalBitmap[WireNum] =
                                    ((ushort)(0x8000 | WirePosition));
}
else
{
  this.vswitches[switchxpos, switchypos].LocalBitmap[WireNum] =
                                    ((ushort)(0x8000 | WirePosition));
}
//Clear the Switch's last unconnected bit
//Create an inverted bitmap with the position we need to turn off set
  to 0
//WirePosition will be zero if we drawing in the forward direction
  (which doesn't need any wire cleanup)

scratch = (ushort)(WirePosition ^ 0xFFFF);
if(SType==DrawModeStatus.SOrientation.Horiz)
{
  //And the bitmap into the UnMapped Outputs to clear the bit if it
  was set
  this.switches[switchxpos, switchypos].UnmappedBitmap =
      ((ushort)(scratch &  this.switches[switchxpos,
    switchypos].UnmappedBitmap));
}
else
{
  //And the bitmap into the UnMapped Outputs to clear the bit if it
  was set
  this.vswitches[switchxpos, switchypos].UnmappedBitmap =
    ((ushort)(scratch &  this.vswitches[switchxpos,
  switchypos].UnmappedBitmap));
}

//We are starting a new wire.  Create a new trace
Trace.CreateTrace(cellxpos, cellypos, (int) CellOutput,
                                                 (uint)CellWire);
//Add the switch to the trace
```

```
    Trace.AppendTrace(switchxpos, switchypos, WireNum,
                                          (ushort)(WirePosition), SType);

    //Return the wire number that we just wired up
    return availWire;
}
```

Appendix D: Math Cell Simulation Source Code

```
public uint SimulateMathCell(uint Inputs, HiperCopsControls.Cell clInput)
     {
        //This function takes 16 bits of inputs (4 - 4 bit inputs) and outputs
2 - 4 bit outputs
        uint scratch, temp, test, y, z, a, b, c, d, ain, bin, cin, din, TTIn,
ret;

        uint[] outputy = new uint[16];
        uint[] outputz = new uint[16];

        //Run through in math mode
        ain=bin=cin=din=a=b=c=d=y=z=TTIn=0;
        //uint "Input" format
        //--------------------
        //  D    C    B    A
        //0000|0000|0000|0000
        //(MSB)          (LSB)
        //^         ^
        //D[3]         B[0]

        //Start with Element 0's Truth table
        //First calculate output on Y

        //Element Layout
        //
        //| 0| 1| 2| 3|
        //| 4| 5| 6| 7|
        //| 8| 9|10|11|
        //|12|13|14|15|
        //

        //Input Map:
        //A - Bits 15-12
        //B - Bits 11-8
        //C - Bits 7-4
        //D - Bits 3-0

        //Mask out a[0], b[0], c[0], d[0]
        a=this.GetA(Inputs);
        b=this.GetB(Inputs);
        c=this.GetC(Inputs);
        d=this.GetD(Inputs);

        //Run through the truth table
        //D is MSB, A is LSB in truth table
        //Ordering of final 4 bits fed to element:
        //D C B A
        #region 1stRow
        #region Element0
        //Now resolve Element 0's Truth Table
        TTIn = LoadTTIn(a, b, c, d, 0, 0, 0, 0);
```

```
//Get the results
outputz[0] = FindZ(clInput.cellelements[0].TTable, TTIn);
outputy[0] = FindY(clInput.cellelements[0].TTable, TTIn);
#endregion
#region Element1
//Now resolve Element 1's Truth Table
TTIn = LoadTTIn(a, b, c, d, 1, 0, 1, 1);
//Get the results
outputz[1] = FindZ(clInput.cellelements[1].TTable, TTIn);
outputy[1] = FindY(clInput.cellelements[1].TTable, TTIn);
#endregion
#region Element2
//Now resolve Element 2's Truth Table
TTIn = LoadTTIn(a, b, c, d, 2, 0, 2, 2);
//Get the results
outputz[2] = FindZ(clInput.cellelements[2].TTable, TTIn);
outputy[2] = FindY(clInput.cellelements[2].TTable, TTIn);
#endregion
#region Element3
//Element 3 (upper left)

//Load up TTIn selection a[3], b[3], c[3], d[3]
TTIn = LoadTTIn(a, b, c, d, 3, 0, 3, 3);

//At this point, TTIn contains the value to lookup in the truth table
//Select the bit of the truth table to select
//Load up Output Z first
outputz[3] = FindZ(clInput.cellelements[3].TTable, TTIn);
outputy[3] = FindY(clInput.cellelements[3].TTable, TTIn);
#endregion
#endregion


#region 2ndRow
#region Element4
//Now resolve Element 2's Truth Table
TTIn = LoadTTIn(a, b, 0, 0, 0, 1, 0, 0);
//Load the two incoming dependent wires
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[0] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputy[1] << 3 );
//Get the results
outputz[4] = FindZ(clInput.cellelements[4].TTable, TTIn);
outputy[4] = FindY(clInput.cellelements[4].TTable, TTIn);
#endregion
#region Element5
//Now resolve Element 2's Truth Table
TTIn = LoadTTIn(a, b, 0, 0, 1, 1, 0, 0);
//Load the two incoming dependent wires
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[1] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputy[2] << 3 );
//Get the results
outputz[5] = FindZ(clInput.cellelements[5].TTable, TTIn);
outputy[5] = FindY(clInput.cellelements[5].TTable, TTIn);
```

```
#endregion
#region Element6
//Now resolve Element 1's Truth Table
TTIn = LoadTTIn(a, b, 0, 0, 2, 1, 0, 0);
//Load the two incoming dependent wires
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[2] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputy[3] << 3 );
//Get the results
outputz[6] = FindZ(clInput.cellelements[6].TTable, TTIn);
outputy[6] = FindY(clInput.cellelements[6].TTable, TTIn);
#endregion
#region Element7
//Element 0 (upper left)

//Load up TTIn selection a[3], b[2]
TTIn = LoadTTIn(a, b, 0, 0, 3, 1, 0, 0);
//Work here!
TTIn = TTIn | (outputz[6] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[3] << 3 );
//At this point, TTIn contains the value to lookup in the truth table
//Select the bit of the truth table to select
//Load up Output Z first
outputz[7] = FindZ(clInput.cellelements[7].TTable, TTIn);
outputy[7] = FindY(clInput.cellelements[7].TTable, TTIn);
#endregion
#endregion

#region 3rdRow
#region Element8
//Now resolve Element 2's Truth Table
TTIn = LoadTTIn(a, b, 0, 0, 0, 2, 0, 0);
//Load the two incoming dependent wires
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[4] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputy[5] << 3 );
//Get the results
outputz[8] = FindZ(clInput.cellelements[8].TTable, TTIn);
outputy[8] = FindY(clInput.cellelements[8].TTable, TTIn);
#endregion
#region Element9
//Now resolve Element 2's Truth Table
TTIn = LoadTTIn(a, b, 0, 0, 1, 2, 0, 0);
//Load the two incoming dependent wires
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[5] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputy[6] << 3 );
//Get the results
outputz[9] = FindZ(clInput.cellelements[9].TTable, TTIn);
outputy[9] = FindY(clInput.cellelements[9].TTable, TTIn);
#endregion
#region Element10
//Now resolve Element 1's Truth Table
```

```
TTIn = LoadTTIn(a, b, 0, 0, 2, 2, 0, 0);
//Load the two incoming dependent wires
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[9] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputy[7] << 3 );
//Get the results
outputz[10] = FindZ(clInput.cellelements[10].TTable, TTIn);
outputy[10] = FindY(clInput.cellelements[10].TTable, TTIn);
#endregion
#region Element11
//Element 0 (upper left)

//Load up TTIn selection a[3], b[2]
TTIn = LoadTTIn(a, b, 0, 0, 3, 2, 0, 0);

//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[10] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[7] << 3 );

//At this point, TTIn contains the value to lookup in the truth table
//Select the bit of the truth table to select
//Load up Output Z first
outputz[11] = FindZ(clInput.cellelements[11].TTable, TTIn);
outputy[11] = FindY(clInput.cellelements[11].TTable, TTIn);
#endregion
#endregion


#region 4thRow
#region Element12
//Now resolve Element 2's Truth Table
TTIn = LoadTTIn(a, b, 0, 0, 0, 3, 0, 0);
//Load the two incoming dependent wires
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[8] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputy[9] << 3 );
//Get the results
outputz[12] = FindZ(clInput.cellelements[12].TTable, TTIn);
outputy[12] = FindY(clInput.cellelements[12].TTable, TTIn);
#endregion
#region Element13
//Now resolve Element 2's Truth Table
TTIn = LoadTTIn(a, b, 0, 0, 1, 3, 0, 0);
//Load the two incoming dependent wires
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[12] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputy[10] << 3 );
//Get the results
outputz[13] = FindZ(clInput.cellelements[13].TTable, TTIn);
outputy[13] = FindY(clInput.cellelements[13].TTable, TTIn);
#endregion
#region Element14
//Now resolve Element 1's Truth Table
```

```
TTIn = LoadTTIn(a, b, 0, 0, 2, 3, 0, 0);
//Load the two incoming dependent wires
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[13] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputy[11] << 3 );
//Get the results
outputz[14] = FindZ(clInput.cellelements[14].TTable, TTIn);
outputy[14] = FindY(clInput.cellelements[14].TTable, TTIn);
#endregion
#region Element15
//Element 0 (upper left)

//Load up TTIn selection a[3], b[2]
TTIn = LoadTTIn(a, b, 0, 0, 3, 3, 0, 0);

//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[14] << 2 );
//Shift by 2 to stick in C's spot.  3 for D.
TTIn = TTIn | (outputz[11] << 3 );

//At this point, TTIn contains the value to lookup in the truth table
//Select the bit of the truth table to select
//Load up Output Z first
outputz[15] = FindZ(clInput.cellelements[15].TTable, TTIn);
outputy[15] = FindY(clInput.cellelements[15].TTable, TTIn);
#endregion
#endregion
y = 0;
y = y|outputy[0];
y = y|(outputy[4]<<1);
y = y|(outputy[8]<<2);
y = y|(outputy[12]<<3);

z = 0;
z = z|outputy[13];
z = z|(outputy[14]<<1);
z = z|(outputy[15]<<2);
z = z|(outputz[15]<<3);

//Combine Y and Z into output uint.  Lower 4 bits represent y.  Upper
4 bits represent z.
ret = 0;
ret = y;
ret = ret | (uint)(z<<4);
//Shift over further
ret = ret << 24;
//uint "Input" format
//--------------------
//  D    C    B    A
//0000|0000|0000|0000
//(MSB)          (LSB)
//This will send inputs X -> A through to the output
ret = ret | Inputs & 0x00FFFFFF;
//OR in the inputs
//     clInput.cellelements[0].TTable;
```

```
    //Upper 16 bits refer to Output Y
    //Lower 16 bits refer to Output Z
    //TTable = 0xAAA7FFFF;
    //Lowest bit of grouping refers to DCBA = 0000
    return ret;
}
```