A MODULAR PIPELINED MEDIUM GRAIN RECONFIGURABLE PROCESSOR CORE

By

JASON DANIEL VAN DYKEN

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

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

August 2012

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of JASON DANIEL VAN DYKEN find it satisfactory and recommend that it be accepted.

_____

José G. Delgado-Frias, Ph.D., Chair

_____

Partha Pratim Pande, Ph.D.

_____

David E. Bakken, Ph.D.

Acknowledgment

This academic endeavor would not have been possible without the help and support of many people, and has been carried out in conjunction with the High Performance Computer Systems (HiPerCopS) research group and the School of Electrical Engineering and Computer Science at WSU.

Foremost, I would like to thank Professor José Delgado-Frias, who has been my advisor throughout my doctoral and master's research. His encouragement, patience, and enthusiasm throughout this process have been invaluable. Dr. Delgado helped find solutions to the many roadblocks and challenges encountered and his guidance and insights have made this dissertation possible.

In addition I would like to thank Professors Partha Pande and David Bakken, who serve on my committee, for their encouragement and advice both in and out of the classroom. They have pushed me to broaden my base of knowledge and helped to guide my research and academic coursework.

Next, I would like to thank my fellow graduate students in the HiPerCopS research lab for sharing their knowledge and for their camaraderie. Specifically, Mike Turi and Zhe Zhang, have made the many hours spent in lab much more enjoyable.

I would especially like to thank my family for their unending support and encouragement. At no point during this endeavor, even when I doubted my progress toward this goal, did my parents belief in my abilities or conviction that I would reach my goal waiver, which has meant so much to me. Thank you for instilling in me the stubborn determination and strong work ethic that has allowed me to reach this point.

Most importantly, I would like to thank my wife Crystal, for her patience with, and tolerance of, my odd work hours and habits. Her unconditional support and companionship has afforded me the strength I needed to complete this dissertation. She has made my life so much more enjoyable than I could ever have imagined. Crystal keeps me grounded and continuously ensures that I am exposed to new topics outside of my direct interests and research, such as all things yarn related.

A Modular Pipelined Medium Grain Reconfigurable Processor Core

Abstract

By Jason Daniel Van Dyken, Ph.D.
Washington State University
August 2012

Chair: José G. Delgado-Frias

This research focuses on a modular and scalable implementation of a MIPS compliant processor core for a medium-grain reconfigurable hardware. The core of this research is built upon the design of four autonomous modular functional units that provide all the operations required of a MIPS core.

Four novel extremely configurable execution cores have been designed to implement a five-stage processor architecture Each of the cores can be configured for varying path widths and forwarding schemes, which have been evaluated for criteria involving area (cell count), delay, and execution efficiency. A comparative study with other reconfigurable hardware has shown the proposed cores' effective clock rate is 3% above the average for similar cores when utilizing 150nm and 90nm CMOS, and 0.99% below the average Xilinx and Altera soft processor speeds for 65nm and 45nm CMOS technologies. The proposed hardware has no specialized hardwired units such as multipliers or adders that are available in Xilinx and Altera chips. Ongoing research on FinFET technology has shown that a system clock of 5 GHz can reasonably be achieved.

An analysis of the hardware has also been conducted examining issues of hardware design and the energy required for the designs to operate and how the cores would be affected if the hardware were modified. The results of this analysis have shown that module power consumption averages 0.248 mW for 8-bit and 1.855 mW for 32-bit data path widths and that the average energy required for executing a SPEC integer benchmark is 2.06 μJ.

Lastly, designs for implementing a reorder buffer and reservation stations have been completed, which can be configured to track a varying number of instructions. Using these new modules with the previously analyzed components a superscalar core may be built. This core has been analyzed to determine the optimal configuration of the reorder buffer and reservation stations, and undergone the same evaluations as the five-stage cores to determine the best operation configuration and energy requirements. This analysis comparing the superscalar core with a five-stage execution core shows that a speedup of 2.073 can easily be achieved while increasing cell count by only 29%.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction and Background

As computer technology continues to advance the reliance of more devices upon general-purpose processors or microcontrollers continue to grow. This in part is due to the falling costs of digital devices, and the ease with which they can be designed and updated. To help integrate digital systems, designers have increasingly focused on two major design paradigms to implement digital systems, the first being to utilize specialized application specific integrated circuits (ASICs), the other is to focus on a single chip implementation that can be comprised of a System-on-Chip or reconfigurable hardware. All of these options typically rely on some form of a processing unit to execute the code that manages the device's operation.

While ASICs and System-on-Chip designs typically have high performance processing units available, reconfigurable hardware offers promising alternatives to these platforms such as the ability to rapidly prototype systems and the potential for

fault tolerance; faults may be avoided by routing around the faults. One such platform that has been developed is a medium grain reconfigurable architecture [1] based solely on memory cells, which allow for complete flexibility in design placement and fault avoidance since there is no special hardware components that require set design placement to access. To date the hardware has been used for the implementation of DSP algorithms. The primary focus of this work is the implementation of a MIPS compliant processor core for the target reconfigurable hardware that will not only provide reliable general purpose computing, but be easily translated into more complex processing architectures that provide high levels of performance. A secondary goal is the ability to provide a means for encrypting and decrypting data using the Advance Encryption Standard (AES) algorithm since the algorithm is not able to be efficiently implemented in code for standard processors without specialized execution hardware and instructions.

1.1 Background

This research study started with a primary focus of finding a reconfigurable hardware method for securing communication between wireless sensor networks (WSNs). The means of securing communication was not the only critical concern, but special attention had to be given to the energy required to secure the communications since many WSNs rely on low power sensor nodes. The AES algorithm was chosen as the focus because of its prominence in private and symmetric key encryption

algorithms. In the initial study FPGAs were chosen to investigate the implementation issues surrounding AES and how they could affect the power required to secure communications. It became readily apparent that while FPGAs could increase hardware utilization and reduce the physical chip count, the power required to integrate an FPGA would not always be possible or prudent, which will be explained further in Chapter 4. As a result the research shifted focus from FPGAs to a novel Medium-Grain Reconfigurable hardware platform that is expected to be much more power conscious and adaptable to new technology implementations, which will allow maximum performance for minimal power.

In the following paragraphs an overview of the target hardware's architecture is presented. The smallest piece of reconfigurable hardware is termed an element, and similar to most modern reconfigurable hardware designs the core of this hardware is memory. Each element contains 32 1-bit memory locations organized into two 16-bit columns. The memory bank is surrounded by interface components that allow for control of the memory that includes read and write capabilities along with multiplexers for routing data, which also provides the ability to act as a pass-through for other element outputs. The elemental structure is shown in Figure 1.1.

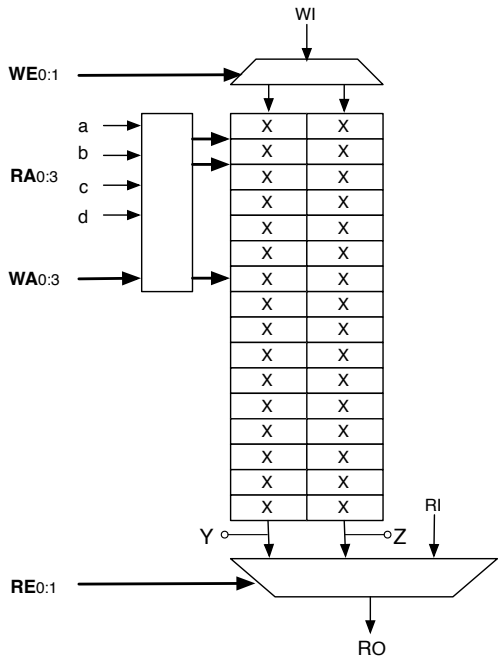


Figure 1.1: Structure of an Element

The reconfigurable hardware is then organized into cells containing a 4×4 grid of elements and is the smallest amount of hardware that can be individually allocated to a design. Each cell can take on one of two configurations, which define how the memory is accessed. The first mode that will be detailed is the Memory mode, and is shown in Figure 1.2. In memory mode the elements are organized into a 128×4-bit memory bank. These structures provide independent read and write control signals, which are used to offer single cycle concurrent read and write operations, to any of the 128 memory locations.

Cells and elements can also be configured to operate in math mode, as shown in Figure 1.3. In math mode the write control signals are disabled turning each element into a 4 input look up table (LUT) with two outputs, which are the Y and Z

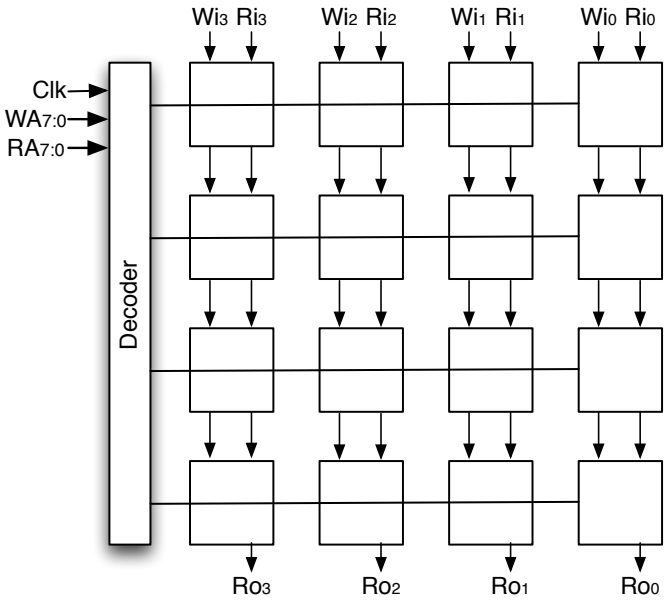


Figure 1.2: Cell in Memory Mode

lines detailed in Figure 1.1. By disabling the read enable mux the cell becomes flexible and dynamic by allowing the elements to more easily communicate with each other, facilitating the implementation of more complex mathematical functions. This capability allows for the math cell's output to become 8-bits wide compared to the 4-bit output in memory mode. The structure of the math cell was chosen to allow for a multiply and accumulate operation to be computed, which is detailed in Chapter 2.

The target hardware contains two parallel communication networks, one being a local mesh network and the other being a global tree network [2]. The local mesh network connects neighboring cells together and allows for any neighboring cell to be communicating data with minimum delay, while the global pipelined communication tree is used to allow cells to communicate with distant cells. Utilizing the global pipelined tree incurs a single cycle of delay for every level traversed; this is

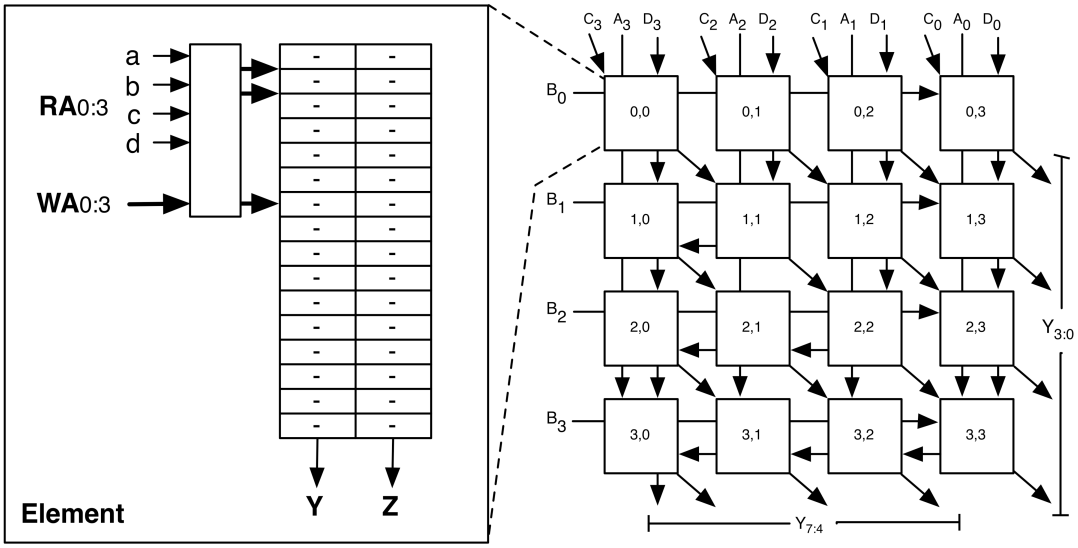


Figure 1.3: Element and Cell in Math Mode

turn leads to a logarithmic delay time with respect to the distance between cells. An additional cycle of delay can be added in each node of the tree to allow for data arrival synchronization.

One of the key attributes that help to differentiate this architecture from traditional reconfigurable hardware platforms is how the clock is defined. Most reconfigurable hardware platforms determine the maximum clock rate by the longest delay in logic between memory elements, however the target hardware defines the maximum clock frequency by the time required for a cell to complete its most complex operation, which can be roughly calculated as one over seven times the elemental delay. This was done so that 4-bit multiply and accumulate operations could be carried out in a single cycle by a cell in math mode. The clock rate is

facilitated by built-in pipelining registers on each of the cell's outputs and pipelined communication tree. This means that the maximum clock rate is technology dependent and not on the mapping of a given design. In simulations it has been shown that using 180 and 90 nm MOSFETs allows for clocks speeds of 267 and 720 MHz, respectively [3]. Scaling these results show that a speed of 2 GHz can be achieved in 45nm or better technology. Ongoing research utilizing future technologies such as FinFETs, similar to Intel's 3D or tri-gate Transistors [4], has shown that speeds of 3 to 5 GHz can reasonably be achieved [5].

1.2 Outline

In the remainder of this study the following topics will be introduced. Chapter 2 will provide an explanation of how the different execution modules that facilitate MIPS compatibility and carry out the computational operations will be presented, including details on implementation. Next in Chapter 3, the proposed schemes for using the modules together as an execution core for a five-stage processor will be introduced, along with the ways in which forwarding can be implemented and optimized. Chapter 4 will provide an analysis of our approach and its results. This will include a more thorough discussion of the work with FPGAs, a performance evaluation of the designs and their custom configuration options, and finally a comparative analysis of this work with other reconfigurable hardware based processors. An investigation of the power consumption characteristics of the

hardware for a five-stage core as well as how the hardware is organized will be presented in Chapter 5. Chapter 6 will focus on the work done to investigate the implementation of a superscalar core and how its performance compares to that of the completed five-stage cores in performance, cell count, and energy required. Lastly, Chapter 7 offers concluding remarks including what knowledge was gained and how this research should continue to proceed in the future.

Chapter 2

Execution Modules

In this chapter the modules required to implement a processor core and facilitate AES encryption and decryption operations will be introduced. In keeping with the separation of primary and secondary goals as outlined in the chapter 1 the first group of modules that will be introduced are those designed to facilitate AES encryption and decryption operations. Next the modules that have been designed for the implementation of a processor core will be introduced. Each module discussion will include its provided operations, cellular implementation, sizing, and layout details. Once all modules have been introduced the means by which they were verified to function as intended will be provided. The actual bit configuration for the modules detailed in this chapter can be found in Appendix A.

2.1 AES Components

In this section the modules that have been designed to facilitate hardware based encryption and decryption of data utilizing the AES algorithm will be introduced. The reason why AES has been chosen is that it has become the standard for symmetric-key cryptography since it became officially adopted and approved of by the US Government in 2002 and is detailed in [6]. The AES algorithm encrypts data in 128-bit blocks that are organized into 4x4 matrices of 8-bit values, called state, which means that modules must operate on 8-bit blocks of data at a minimum rather than the 4-bit blocks used in the processor components. There are four primary transformations that take place during encryption and decryption, three of which will be presented in the following sections. The reason why only three of the transformations are detailed is that one of the transformations, termed the Shift Rows transform, has been demonstrated to provide better performance when implemented in the data routing network and not using logic elements [7] or in the target hardware's case cells. Lastly the key expansion transformation will be detailed along with how it can be implemented using the other components described.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Equation 2.1: S-Box Transform

2.1.1 S-Box Transform

The S-Box, meaning substitution box, transform is what produces non-linearity into the AES cipher. The mathematical operation that is performed in this transform is derived from a multiplicative inverse over the finite field $\{2^8\}$ and is combined with an affine transform to strengthen the transformation against simple algebraic attacks. This transform represented mathematically is shown in Equation 2.1. This computation though is not easily implemented in hardware, but lends itself particularly well to implementation via LUT, because the mapping is 1:1 and thus only 256 8-bit memory locations are needed. By focusing on the LUT implementation the resulting module will be the most ideally optimized for the target hardware since the target hardware is already made up of pure memory and can be used in a efficient manner with minimal delay for computation.

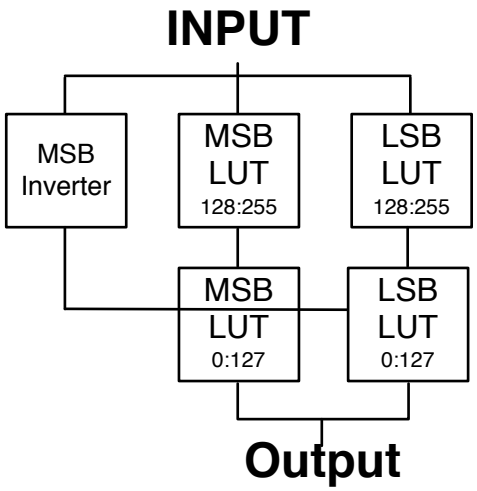


Figure 2.1: S-Box 8-Bit LUT Implementation

The design that was settled on for an 8-bit LUT module is detailed in Figure 2.1, and is built with 5 cells. Since each cell is able to provide 128 4-bit memories two cells are needed in parallel to produce an 8-bit string, two serially connected cells are also needed to provide 256 different memory locations which results in the two parallel cells feeding serially into two other parallel cells. The fifth cell is used as a data read control signal, since the input data is already 8-bits and the most significant bit in the memory cell serves as a read enable all that must be done to utilize the serially connected cells is to invert the MSB of the input. This means that the upper cells contain all the transforms for inputs ranging from 128 to 255, while the lower cells contain the transforms ranging from 0 to 127.

The inverse operation for this transformation known as the Inverse S-Box is implemented the same way as the forward S-Box with the only difference being the contents of the LUTs. These modules require 2 cycles to perform the transform and

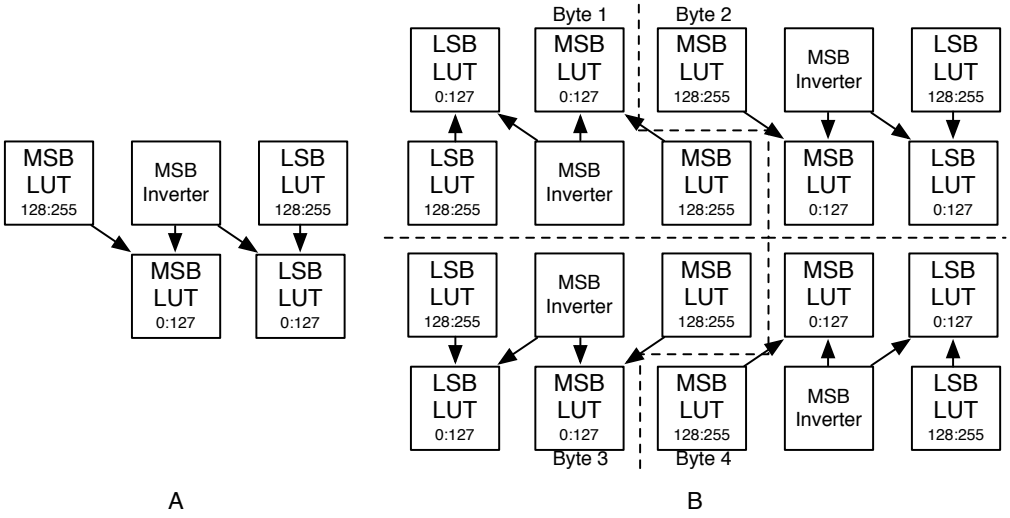


Figure 2.2: S-Box Hardware Layout

require 5 cells for every 8-bits of data that must be concurrently processed. A physical layout for an 8-bit module is shown in Figure 2.2A, and as is shown in Figure 2.2B four transforms allowing for a 32-bit data path can be fit into a 4x5 block of cells without any wasted hardware.

2.1.2 Add Round Key Transform

The Add Round Key transformation is carried out through the XORing of the 128-bit state with one of the round keys. Since the inverse of an XOR function is an XOR function the inverse Add Round Key transformation is itself a simple XOR operation. Although this transform could be carried out using the logic cell presented in section 2.2.1, an 8-bit XOR cell has been designed, shown in Figure 2.3, which reduces the cells needed for adding round keys by half. This translates into a forward

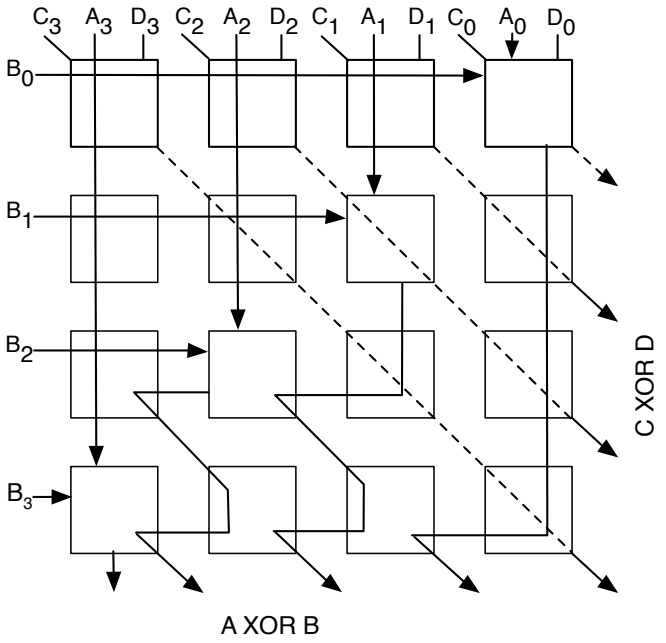


Figure 2.3: 8-Bit Xor Cell

and inverse transformation requiring 1 cell per 8-bits of data being concurrently processed and 1 cycle of delay to complete the designated transform.

It should be noted that when the last round of encryption is being computed the Mix Column transform is omitted from the round. Rather than implement two round blocks in hardware a simple solution is to attach multiplexers to the input of the Add Round Key transformations which can choose the data coming from to Mix Column module or the result of the S-Box transformations via the hard wired Shift Rows transform. This can be seen in Figure 2.4 adds two cells to the Add Round Key module per 8-bits of processed data, and increases the delay by one cycle but greatly reduces overall cell count for most standard AES implementations.

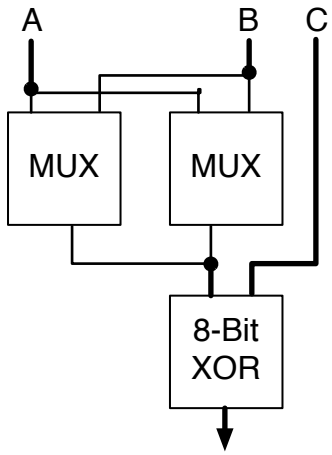


Figure 2.4: Alternate Add-Round-Key

2.1.3 Mix Columns Transformation

The Mix Columns operation is the only operation that requires more than 8-bits of data at once, since the shift row transformations can be carried out through routing. The Mix Column operation and its inverse are shown in Equation 2.2 A and B, and generate new values for each of the values in the state by using a series

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 2 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} = \begin{bmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S'_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S'_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{bmatrix}$$

A: Forward Mix Column Transform

$$\begin{bmatrix} E & B & D & 9 \\ 9 & E & B & D \\ D & 9 & E & B \\ B & D & 9 & E \end{bmatrix} \times \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} = \begin{bmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S'_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S'_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{bmatrix}$$

B: Inverse Mix Column Transform

Equation 2.2: Mix Column and Inverse Mix Column Transformations

weighted summations of each column. The purpose of this transform is to increase entropy in the encrypted data and to ensure that the final output is dependant upon all of the plaintext and intermediate round bits.

To implement the forward transformation each of the column's state values must first be multiplied by two and three, which can be done using only two cells as shown in Figure 2.5. The cells take in the input data and generate their corresponding x2 result that is then output and routed back to the cells input. Once the x2 result is sent back to the cell inputs, it is combined with the original input, delayed one cycle and sent to another input, to generate the x3 result. This means that the two required factors for completing the Mix Column transform can be generated using two cells and two cycles of delay. The designated results of the multiplication blocks along with original data values can then be fed into two parallel cells capable of a four input 4-bit XOR operation to generate each of the new state column values.

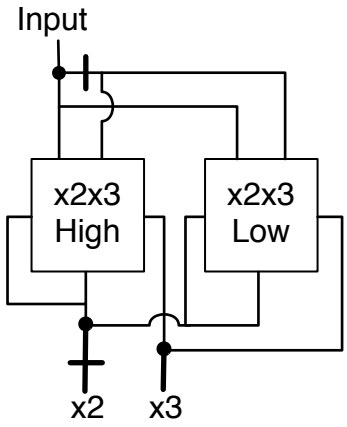


Figure 2.5: Multiply by 2 and 3

This means that 16 cells and 3 cycles of delay are needed to complete the forward Mix Column operations for a single column of the state when using a four input XOR block. There may be one or two cycles of added delay though as signals are routed to more distant cells. The inverse Mix Column transform requires scaling factors of 14, 13, 11, and 9, which can be done through a series of x2 multiplier blocks as detailed in [8], which would increase the cells required to 52 and the delay to 7 cycles. Further optimizations may be possible to reduce cell count for the inverse transform, but this would most likely increase the total delay of the unit.

2.1.4 Key Expansion

Since the AES algorithm is based on repeated rounds of transformation operations the original key must be used to generate a unique key for every round called a round key and is used to ensure maximum entropy in the encryption process. The processes of generating new keys is the largest transformation of the AES algorithm, but can be looked at as a series of 4 smaller transformations. The key expansion process is the same regardless of whether encryption or decryption is being carried out and all that changes is the order in which the round keys are used.

Round keys are generated using the original key or the previous round key and a round constant that can be stored in a LUT. The round key is generated using the following steps and is shown in Figure 2.6:

Rotate Word: The rotate word transform takes the final 32 bits of the state and performs a right rotate of 8 places. This transform like the Shift Rows Transform is best accomplished through direct routing.

Substitute Words: The final 32-bits of the key then undergo a bit substitution to increase differences between round keys. The substitution that is carried out on the bits is the same S-Box substitution previously described in this section 2.1.1 and would require 16 cells and three cycles of delay.

Add Round Constant: A round constant is then added to bits 96 through 103 through an XOR operation. The round constant is best implemented through the use of a LUT and 8-Bit XOR cell, which would require three cells and one cycle to complete the operation.

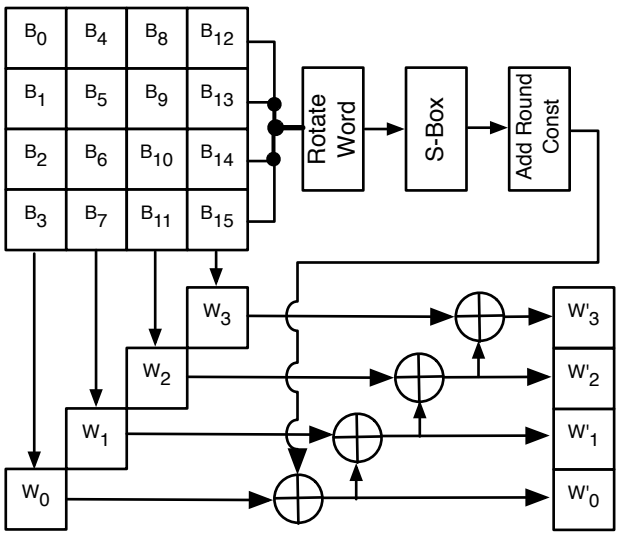


Figure 2.6: Key Expansion

XOR Stage: The result of the previous transformations is then XORed with the first 32-bits of the previous key to generate the first 32-bits of the new round key. The newly generated bits are then XORed with the second 32-bit block from the previous key to generate the next 32-bits of the new round key, which is repeated until the new key reaches 128 bits or 4 words. The method used to implement this process that best balances cell count and delay would include 16 8-bit XOR cells and require 4 cycles of delay.

The cell count and delay requirements of the key expansion process as shown in Figure 2.6 would be 37 cells and 7 cycles, with the round constant LUT adding 2 cells to the total cell count. However it is possible for the round constant to be generated as part of the control logic and eliminate the 2 LUT cells.

2.2 Processor Modules

Before the actual components that make up an ALU or other critical CPU components could be designed the required functionality needed be determined. As previously mentioned it has been decided to provide a set of instructions that would be compatible with the MIPS instruction set as outlined in [9]. Using this as a base line for functionality allows for some flexibility in the design process, if an implementation could offer additional functionality without detrimentally affecting the component's size and delay characteristics than the added functionality would be included. Another goal of the design process has been to ensure that the modules can

scale efficiently and are not locked to a set data path width. In the remainder of this section the designed components are presented, each component discussion includes an overview of how a four-bit operation is carried out along with a description of how the design can be expanded to more than 4-bits. Finally a brief discussion is provided on other support components that are critical for the implementation of a processing unit. An early version of these components has been presented [10].

2.2.1 Bitwise Logic

When implementing a unit for bitwise logic the critical functions that must be included are AND, OR, XOR, and NOT. This fits perfectly with the target hardware since two inputs could be allocated as data inputs {$C_{3:0}$, $D_{3:0}$}, with the other two inputs {A_x, B_0} being allocated for use as a function select. The remaining cells in the design could be used to route the data to either the $Y_{7:4}$ or $Y_{3:0}$ outputs. Using this implementation all of the logic operations contained in the standard MIPs ISA could be provided.

In keeping with the goal of providing extra functionality when there would be little or no cost to the cell count or delay requirements for a execution unit it was decided to use the second row of elements and the B_1 input as an optional inverter. This meant that with the use of 1 extra input bit in an established control signal the logic functions, NAND, NOR, XNOR, and a pass-through could be provided. This is shown in Table 2.1, and Figure 2.7.

TABLE 2.1: LOGIC OPERATIONS AND CONTROL SIGNALS

Control Index (B_0,A_n)	B_1 0	B_1 1
0,0	C and D	C nand D
0,1	C or D	C nor D
1,0	not C	C
1,1	C xor D	C xnor D

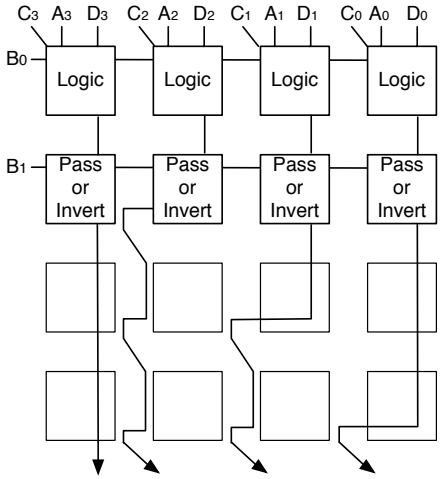


Figure 2.7 Logic Cell Organization

One of the benefits to bitwise logic is that there is no need for any knowledge about neighboring bits or their result to complete the designated operation. This results in a completely parallel computation structure, with the delay remaining unchanged for any path width and the cells required for a given operation to be calculated by simply dividing the path width by four. For a 16-bit unit four cells are required and one cycle is needed to complete any supported operation, similarly a 64-bit unit requires 16 cells and one cycle to complete a given operation.

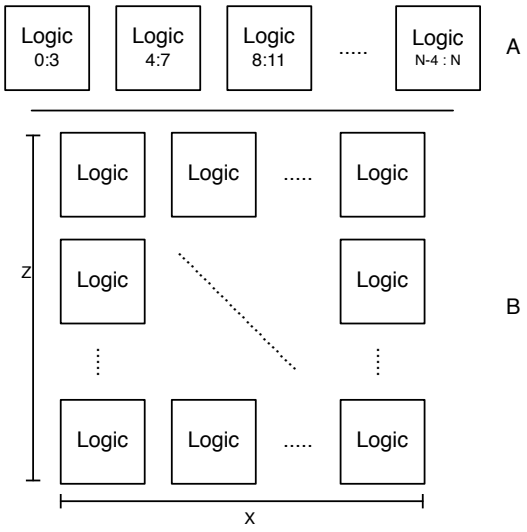


Figure 2.8: Logic Module Layout and Placement

When it comes to layout this unit is the most flexible of all of the units, since there is no need to communicate information about other bits such as their input values or output. This allows for the Module to be placed in as small a region as possible, or spread out and placed in cells that are not being used, but surrounded by cells currently allocated for other modules. The most obvious way to layout the modules is shown in figure 2.8, where in 2.8 A the cells are placed in a straight line. Alternately in Figure 2.9 B the cells are placed in a rectangle where X×Z equals the Path Width divided by 4.

2.2.2 Multiplication, Addition, and Subtraction

Rather than design components for each of these operations it was decided to use a component originally detailed in [11] and for which the math cell was designed.

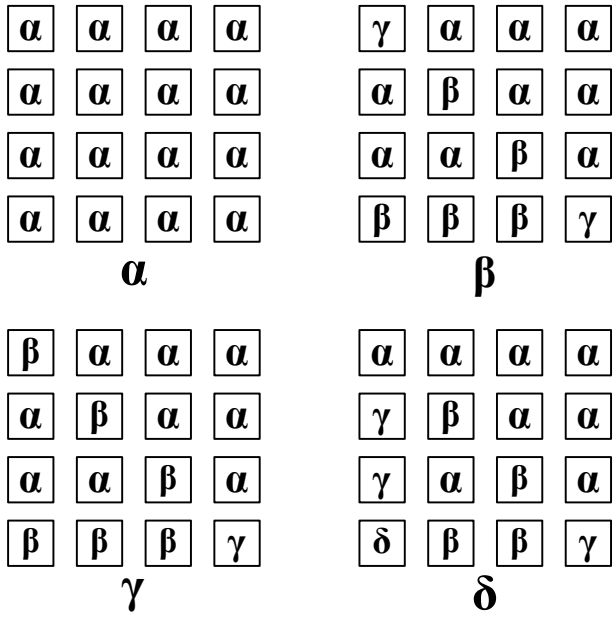
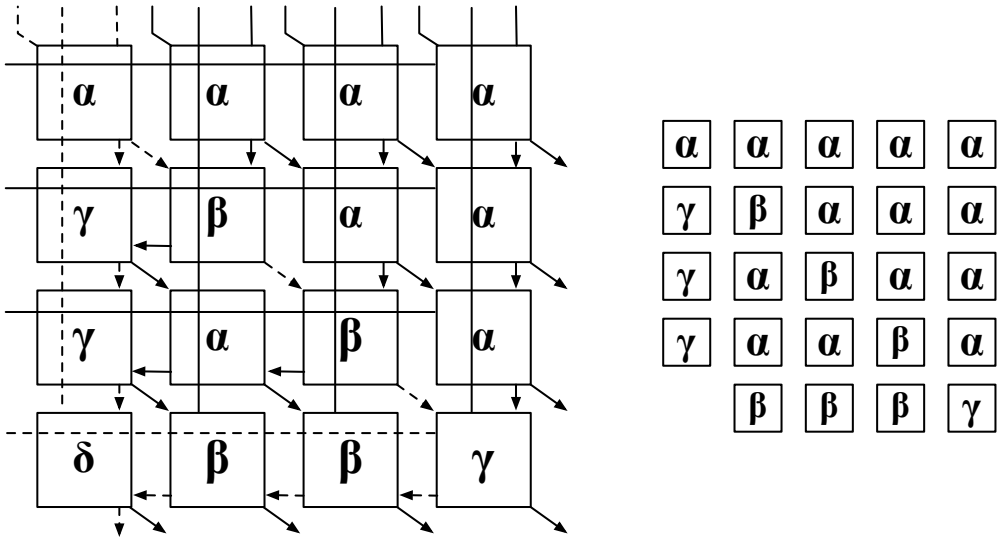


Figure 2.9: Cell Types based on Elemental Configuration

The unit is able to carry out a multiply and accumulate (MAC) operation. When every element in a cell is programmed to implement (2Z + Y) = (A AND B) + C + D, a 4-bit MAC of the form (A×B) + C + D is obtained, with the resulting output being an 8-bit value. This configuration though is only for unsigned numbers, but with minor changes can be adapted to accept signed values, along with varying input widths. The changes that are required to enable the module to accept signed numbers involve changing the configuration of elements internal to each cell. The four elemental configurations that can enable a signed MAC operate are termed Alpha, Beta, Gamma, and Delta, and are chosen based on which if any of its inputs represent a sign bit. Using these elemental configurations there are four separate cellular configurations that are used to make up a MAC unit greater than 4-bits. The four different cell types



A: Standard Signed MAC Unit B: 16-Bit MAC-2 Unit and Layout

Dashed lines represent signed bits

Figure 2.10: 16-Bit MAC Units

are detailed in Figure 2.9 and show how the different elemental configurations are used to implement the cells, which share the same naming scheme as the cells themselves. Figure 2.10A shows the implementation of a 16-bit MAC unit and how signed bits are routed through the MAC unit.

The obvious method for providing signed and unsigned operations in a processor core is to utilize 2 MAC units one configured for each type of input. However, this is not a responsible use of hardware, so a different method had to be found. The solution that was derived and termed the MAC-2 is based on implementing an optional sign extension module, which could be placed in the

instruction decode section of a processor, and then to implement a MAC unit that was capable of computing a multiple and accumulate operation of N+4 bit, and then ignoring the highest 8 bits of output. The benefit of this method is detailed in Table 2.2. As Table 2.2 shows the savings is minimal in terms of cell count for 8 bit data paths, but rapidly increases to almost 50% for larger data path widths. Additionally by utilizing the MAC-2 any standard arithmetic function can be completed except for division by controlling which input the data is routed to. As is shown in Table 2.3 multiplications can be computed by sending the data values to the A and B MAC-2 inputs. Subtraction operations can be accomplished by multiplying the subtrahend by -1 and adding it to the minuend. Addition operations can be completed by simply routing the two addends to the C and D inputs, but to minimize input usage and hardware routing paths the operation will be computed by multiplying the augend by 1 and adding the other addend to its result. Furthermore if a specific application required incrementing operations by set amounts the MAC-2's unused input could be adapted for that purpose, much like the ARM processors use barrel shifters in conjunction with adders for quick multiplications [12].

TABLE 2.2: MAC UNIT CELL COUNT COMPARISONS

Path Width	Area in Cells			MAC-2 Savings over Dual MAC
	MAC	Dual MAC	MAC-2	
8	4	10	8	20.00%
16	16	36	24	33.33%
32	64	136	80	41.18%
64	256	528	288	45.45%

As is demonstrated in Table 2.2 the MAC-2 unit scales at a rate of $(N/4)^2$-1, while the delay required for operation scales at a rate of N/4 cycles for N-bit operations or N/2 cycles for 2N-bit operations. This translates to a 16-bit MAC-2 unit requiring 15 cells and 4 or 16 cycles and a 64-Bit MAC-2 unit requiring 255 cells and 8 or 32 cycles for N and 2N-bit operations respectively. One drawback to the MAC units is that they require a set placement pattern that allows for built in pipelining to be utilized if needed and to eliminate extra delay when communicating carryout or borrow-in information. Figure 2.10B shows the required implementation for a 16-bit MAC-2 unit.

2.2.3 Shifting and Rotating

When designing the shift module the main operations that needed to be included were the shift left logical, shift right logical and shift right arithmetic operations. After examining different methods for implementing the unit a shifter has been chosen that would reduce delay and hardware when compared to more popular barrel shifters like those detailed in [13]. The shifter is made up of a single shifting

TABLE 2.3: MAC-2 INPUT ORGANIZATION

MAC Fuction	Input A	B	C	D
X × Y	X	Y	-	-
X + Y	1	Y	-	X
X - Y	-1	Y	-	X

cell, which can carry out shifts of 0, 1, 2, 3, or 4 bits and relies on inputs ordered {$C_{3:0}$, $D_{3:0}$}. Left and right shifts require separate, but identical configuration, cells that can be connected to a MUX to choose the appropriate output.

The basic shift cell is shown in Figure 2.11 and as detailed in the figure shows the top row of elements are used as bit select blocks or basic two input multiplexers. The remaining elements in the cell are configured to act as a router and based on the $A_{3:1}$ and $B_{3:1}$ inputs can route the selected bits to any block for output. This configuration allows for any continuous sequence of 4 bits to be output from the cell from the original input of {$C_{3:0}$, $D_{3:0}$}. When being used for left moving operations the input should be ordered with the original bits being sent to the $C_{3:0}$ input and the shift in bits occupying the $D_{3:0}$ input. The right moving operations require the opposite input ordering with the original bits being connected to the $D_{3:0}$ input and the fill bits tied to the $C_{3:0}$ input.

To facilitate operands greater than 4 bits, cells must be used in parallel with bits being shifted into one block being the primary bits in the neighboring block. Additionally shifts of greater than four bits can be implemented in one of two ways. The first method is through repeated use of the cells broken up into 4 or fewer bit shifts. An alternative method, which is used for this work, is based on feeding the output of the shifter into another shifter allowing for a total shift of 8 bits, which can be repeated to offer shifting of any number of bits. This method also allows for one

shifting cell to be eliminated for every row of shifters, because if a shift of greater than four is taking place, one cell will output four of the fill bits and this will not change no matter how many more places are shifted. This is detailed in figure 2.12, which shows a 16-bit right shifter capable of shifting up to 16 places.

A rotate operation may be implemented simply by modifying the output MUX, which chooses the left or right shifter, to compute an OR operation of the two shifted values representing left and right shifts. This is only valid if the two shifters shift by complimentary amounts that when added equal the total path width. An example of this is that 8-bit value 10010110 when shifted left by 2 becomes 01011000, and when ORed with the same value shifted right by 6 (00000010), becomes a left rotate by 2 or 01011010. Since this required no additional hardware and only minor

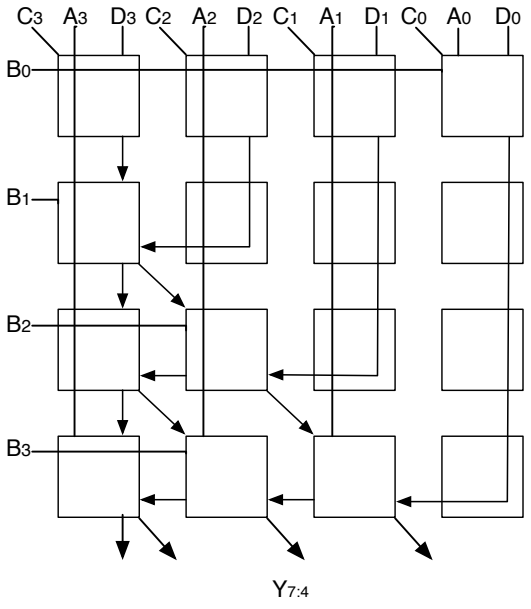


Figure 2.11: Shift Cell

modifications of the output mux and control logic it was added to the standard functions offered in our MIPS ISA compliant architecture.

In addition to the shift cells and output mux it was decided that the control logic should be made internal to the shift module. By doing this the modules inputs could be reduced to the 3 inputs: data, shift amount, and function, eliminating the control signals needed for each row of shift cells as inputs. To generate the control signals two memory cells are needed for each row of unidirectional shifts and must be used to generate the control signals a cycle before the data arrives at the controlled cells. Additionally a cell for modifying the input shift value to a memory index is needed for each shift direction. This creates a delay overhead of 2 cycles and a sizeable increase in the total cell count requirements but ensures a self-sufficient

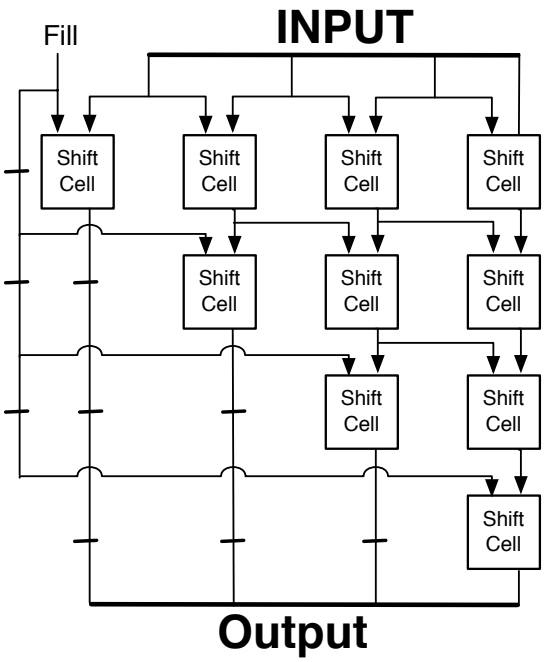


Figure 2.12: 16-Bit Right Shift Unit

module that can easily be integrated into any type of processor or device requiring shift or rotate functionality.

A diagram of the shifter with the control logic is shown in Figure 2.13. Like the MAC-2, the shift module relies not only on neighboring bits, but also on distant bits within a word, if the shift/rotate amount is significantly large, and serial computations. This means that rather than slowing increasing the delay in the component as the path width increases the delay increases linearly and can be calculated by computing 3+N/4. The cell count requirement for shifting and selecting an output are not easily defined by a single equation, but for a four bit shift requires 3×(N/4), and for every for extra 4-bits of shifting 2×((N/4)-*i*) cells are required, where *i* is the distance from the first shifting row of cells. This leads to a 16-bit shifter requiring 24 cells for shifting, 20 cells for control logic and 7 cycles for computation,

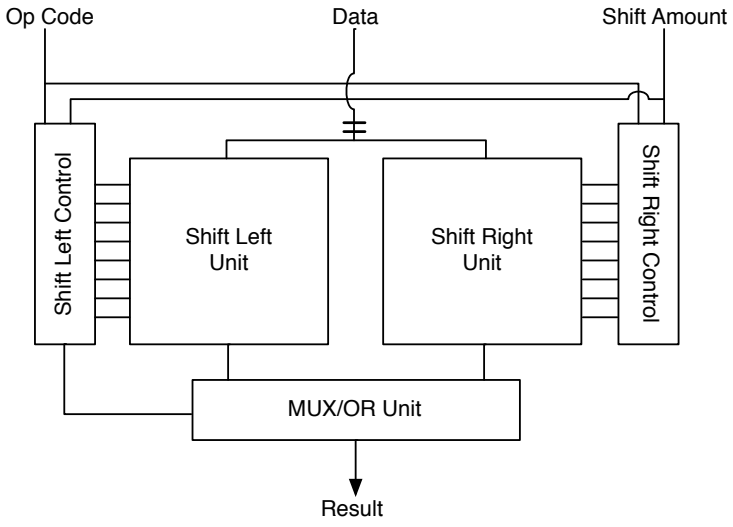


Figure 2.13: Shift Module

similarly a 64-bit shift unit would require 288 cells for shifting, 68 cells for control logic and 19 cycles for the operation to be completed.

2.2.4 Comparator

Comparators in modern processors are depended upon for two primary instruction types, the first being branching where the decision to branch is based on whether a number is equal to zero or to another number. The second major use for comparators in processors is "set on" instructions which can range from set on: $<$, $>$, $\leq$, $\geq$, $=$ or $\neq$. These instructions allow for the processor to make decisions on what to do based on the state of a register or registers. In the target reconfigurable hardware the comparator is one of the more challenging components to implement because unlike other operations there are three pieces of information needed to make a decision that are: has a decision been made, are the bits equal, or is the comparison bit greater than the reference bit. This causes one of the limitations of the math cell layout to come to light, which is that the last cell in each row can only transmit 1 bit of information about itself to the rest of the cell. To overcome this issue the comparator is broken up into 2 cells, the first being an information generation cell and the second being a decision-making cell.

The information generation cell or top cell is a very simple cell. As shown in Figure 2.14, the top row of elements in the cell output whether the bits are equal on one output and whether the comparison bit is greater than reference bit. The equal

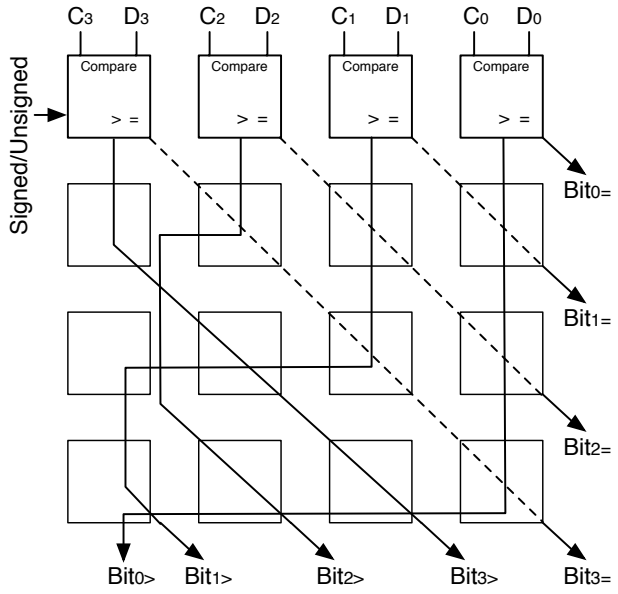


Figure 2.14: Comparator Top Cell

bits are then routed to the $Y_{3:0}$ outputs representing the bit order {$B_{3=}$, $B2_{=}$, $B_{1=}$, $B_{0=}$}. Similarly the greater than bits are routed to the $Y_{7:4}$ outputs, except that the bit order is reversed to represent {$B_{0>}$, $B_{1>}$, $B_{2>}$. $B_{3>}$}, which allows for the complexity of the decision cell to be reduced. If the input values are in 2's compliment form all that must be done is to have the most significant bit's information generation modified to output a greater than if the comparison bit is zero while the reference bit is 1 or vice versa, otherwise all information generation operations are the same. To include this functionality the B_0 input of a cell can be used to indicate signed comparison and that standard behavior should be modified.

The decision cell or bottom cell, as shown in figure 2.15 is actually quite simple and elegant since the bits representing greater than comparisons have been

reversed in the information generation cell, otherwise the decision cell would have become much more complex. As is shown in Figure 2.15 the cell operates on two different data paths. The first path is dedicated to the greater than comparisons and routes data from the upper left corner down to the lower right. While data is moving from element to element the greater than comparisons are merged into a single greater than decision, if a more significant bit is found to be greater than its reference than the previous decision is set aside and replaced with the most significant bit. The second data path is dedicated to equal detection and transmits data in a tree pattern, from the top of the cell to the bottom. With each additional level the equal information incorporates an additional bit of information so that by the time the data reaches the bottom of the cell, the left most three cells know if the original 4 input bits are equal or not. Using the greater than and equal bits the output $Y_{4:0}$ can be set

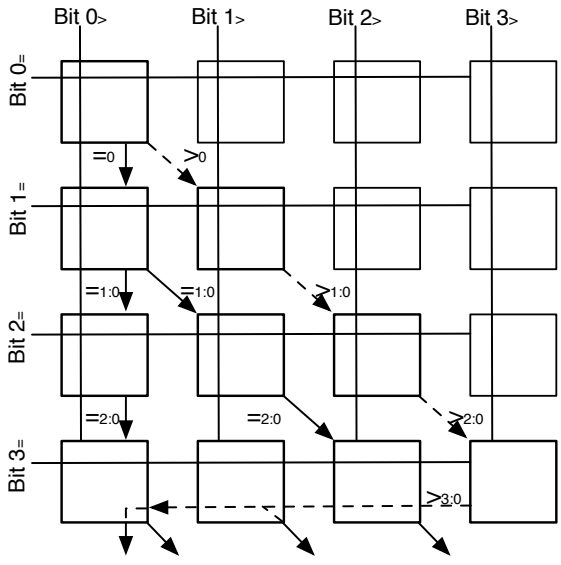


Figure 2.15: Comparator Bottom Cell

to show the result of any of the comparison operations on any or all of the outputs, as detailed in Table 2.4. To make the comparator more adaptable to expansion the output bits are mapped as follows: {>, =, >, =}.

Rather than rely on the sequential nature of comparisons and chain the output of one 4-bit comparison into the input of another, a parallel implementation was adopted to reduce delay while minimally increasing the required number of cells. The final cell that needed to be designed to complete the module is a collector cell, which is shown in figure 2.16. The cell is divided into four parts with the cell's four inputs being ordered {A, B, C, D} for order of significance. Part J is used to merge the C and D inputs to make a decision on the eight least significant bits' comparison. Part K takes the resulting decision from Part J and using the A and B inputs makes a decision for the entire 16 bits on whether the comparison input is greater than the reference and passes the result to Part M. Part L checks if each of the 4-bit segments are equal and if so sets an equal bit that is sent to Part M. Finally Part M takes the output from Parts K and L and can output any of the comparative results on the $Y_{7:4}$ output. To

TABLE 2.4: Comparator Output Generation

Operation	Logic Derivation
>	>
≥	> or =
<	!(> or =)
≤	!>
=	=
≠	!=

ensure the module is highly scalable Part M is set to output the results in the form {>. =. >. =}, which can then be sent to another collector cell if needed. Additionally output of the final collector cell can be modified to show the result of any supported comparison operations. A Diagram of the final module structure is shown in figure 2.17 A.

Using the collector cell allows for comparison operations to be computed in parallel, so that while the path width and cell count are increasing at a rate close to (N/4)×2+N/16+N/64… the delay required to complete the operation is increasing at a much slower rate, closely following 2+N/16+N/64... This translates to a 16-bit comparator requiring 9 cells and 3 cycles of delay while a 64-bit comparator requires 37 cells and only 4 cycles of delay. An added benefit of relying on parallel

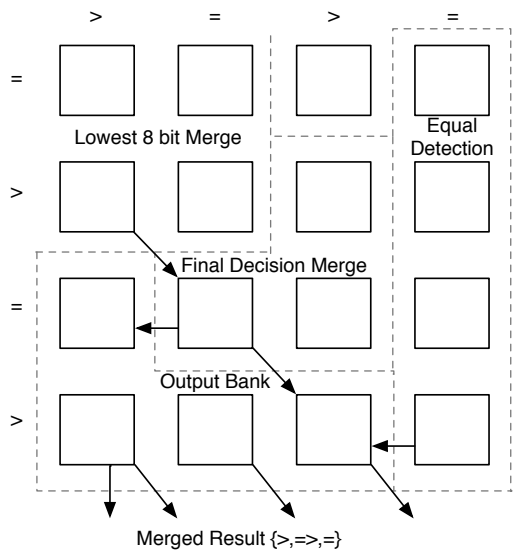


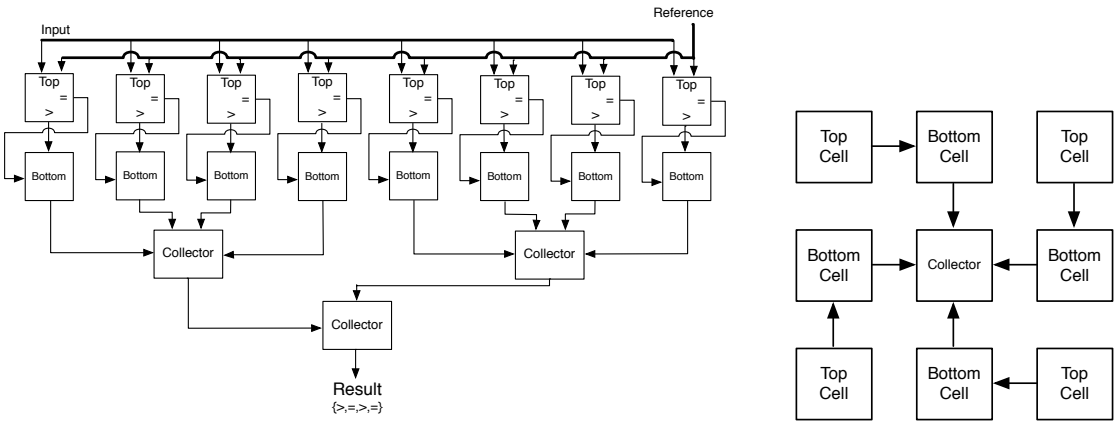
Figure 2.16: Comparator Collector Cell

computations is that the comparator can be made to fit in a relatively small number of cells on the chip, as is shown in the layout of a 16-bit comparator in Figure 2.17 B.

2.2.5 Other Components

Other key support units/components that allow the execution units to be integrated into a processor core are presented below. The implementation of these components is briefly described along with their basic sizing details.

- ***Register File.*** The register file is simply a block of memory that can concurrently read two values, while writing to another. This can be built by allocating two parallel groups of cells in memory mode and connecting the write inputs to the same source. Using this model 2 cells are needed for every 4-bits in the data path and will provide up to 128 register locations. Typically there are 32 general-purpose registers plus the Hi/Lo registers for



A: 32-Bit Module Organization B: 16-Bit Comparator Layout

Figure 2.17: Comparator Oganization and Layout

multiplication and division operations, by partitioning these memory locations accordingly a processor supporting 2 threads can easily be realized.

- ***Program Counter.*** Since the program counter is meant to increment the PC value by four or another constant value a scaled down MAC unit can be used which simply takes the addends and computers next PC value. The optimized MAC is essentially the far right row in Figure 13.b and requires N/4 cells placed serially and requires N/4 cycles to generate the new PC value.

- ***TriMux.*** Due to the architecture of the basic math cell a 2:1 mux can be implemented that only utilizes 25% of the elements in a cell. To increase cell utilization a 3:1 mux has been designed that can also set, reset and invert the chosen input bits. This unit makes data routing much more efficient, since building a 4:1 mux requires 2 cells rather than 3 cells. The TriMux requires N/4 cells for implementation and has no specific placement requirements since cells contain no interdependencies.

- ***Pipeline Registers.*** Pipeline registers can be implemented by using one input of a cell in math mode as feedback from the output, which allows for the current data to be saved for multiple cycles. The remaining 3 inputs can be used for new data input, set, and reset signals. This means that one cell is needed for every 4-bits of input and can be places in nonadjacent locations.

2.3 Module Functional Simulation

One important aspect of the module design process that has not yet been touched on is the method for ensuring that the designs function as intended. It is critical that each cell and module will work as expected, without exception, because they will serve as the core of a processor or coprocessor that must be reliable in order to be utilized. In this section the method used to verify the modules' functionality will first be detailed, this will include a detailed description of the method used as well as an explanation for why the method was chosen. Lastly the module functional simulation results will be described.

When attempting to verify that modules and their cells were functioning properly there were three possible choices. The first method was to use hand calculations to prove functionality; this method was quickly abandoned though due to the size and complexity of some of the modules along with the ease with which human error could be introduced. The second method considered was to utilize a simulator developed by another student, which included basic cell placement capabilities. This option was appealing but in the end could not be used due to the simulator's inability to track signals internal to modules and size constraints when implementing the larger modules such as the MAC-2 and shift modules. The final method that was considered and ultimately chosen was to use VHDL modules to

implement the basic memory and math cells on an elemental level and then through component mappings connect the cells together to form each of the modules.

The VHDL approach was particularly appealing since it allowed for signals not just within a module to be tracked during simulation, but also signals within a cell, and since VHDL is a common hardware definition language it also meant that there were many software suites that could be used to complete the simulation process without having to write custom simulation software. Similarly test benches could be made to use VHDL's file IO features so that expanding a test set required only adding the additional test to a text file. This method also allowed for modules of any size to be designed, and for modules to be made of other modules if needed.

The process to ensure each of the modules started at a cellular level; first each cell type in a given module was tested to ensure it reacted properly to the given stimulus. Once each of the cells had been verified any groups of cells that were used together to carry out part of a modules operation were then tested. Finally the module was built for a 32-bit path-width and tested with a sufficient number of inputs for each provided operation to ensure that everything was functioning correctly.

Each of the processor modules as well as the support modules has been thoroughly verified to function properly through this simulation process. An example of the verification output is shown in Figure 2.18 and 2.19, which

demonstrates that the Shift and MAC-2 modules consistently output the correct results. Each of the modules designed to carry out encryption transformations were also verified to ensure proper functionality. Decryption transformation modules did not undergo the same thorough testing because they work identically to their encrypting transformations with only modified memory values.

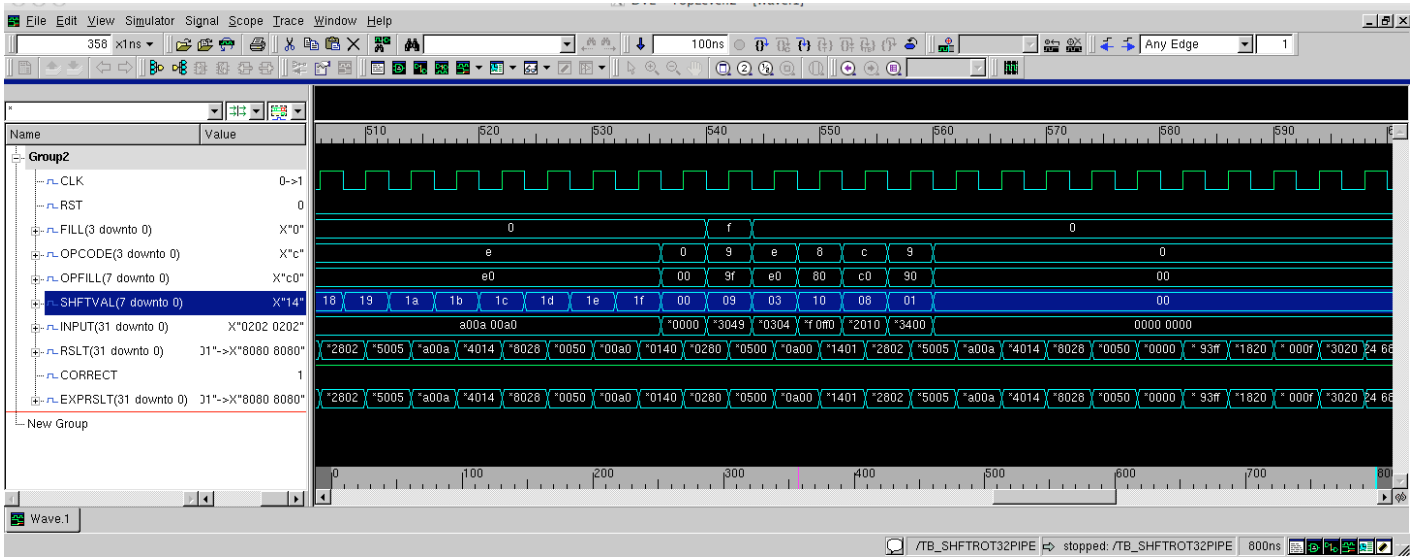


Figure 2.18: 32-Bit Shift Module Functional Simulation

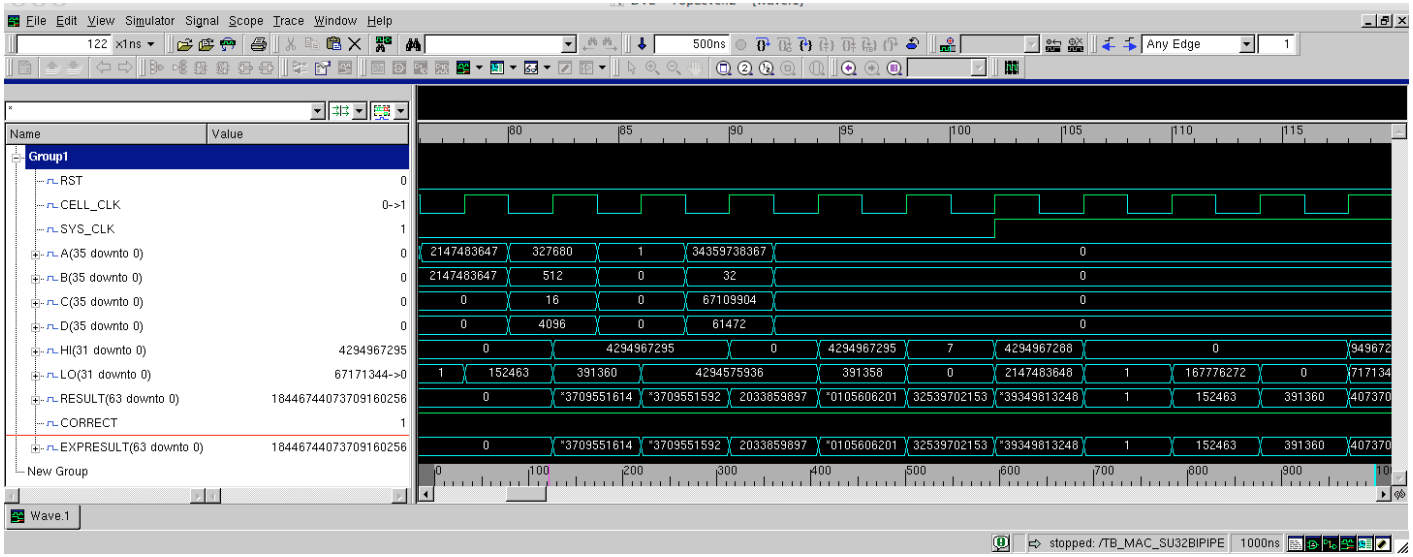


Figure 2.19: 32-Bit MAC-2 Module Functional Simulation

Chapter 3

Execution Cores and Forwarding

One of the major benefits to using the modular processing components is that execution cores can be optimized for the most common instruction in a given program or for the behavior characteristics desired. Design considerations and components such as clocking, the core architecture and the methods for implementing data forwarding that are required to build the processor's execution core are presented in this chapter. By utilizing the methods for configuring and designing the execution cores presented here, the design of a five-stage processing architecture as detailed in [9] may be obtained, that is not only MIPS compliant, but can extend the instruction set.

The remainder of this chapter will proceed as follows. A discussion of how data processed and output by each of the modules will first be provided. The introduction of the execution cores themselves will then be discussed including a

brief discussion of the mechanisms that allow for branch and jump instructions to be completed. Lastly, the three different data forwarding schemes that have been devised will be presented, along with details regarding their cell count and delay requirements.

3.1 Five-Stage Architecture Adaptation

The first step in implementing the execution core for a pipelined processor is to determine how best to adapt the standard five-stage model for the target hardware. The most obvious solution to this task is to utilize the hardware's built in pipeline and serially connect all the necessary components. There are two downsides to this approach the first being to calculate PC + 4 requires up to eight cycles and would require generating new PC values in parallel to ensure maximum throughput. This leads to the second and most complex problem, which is managing the deep pipeline. The control logic required to clear the pipeline for a missed branch prediction or to service an interrupt would become very complex.

An alternative approach that has been developed in this study is to utilize a predefined window of clock cycles to represent the time required for an instruction to be serviced in a given stage to implement a standard five-stage processor. This allows for the core to become more deterministic in terms of execution rate and can simplify the control logic. The window size as defined by the number of cycles it spans has been termed the Control Synchronization Factor (CSF). If the CSF is 1 then every

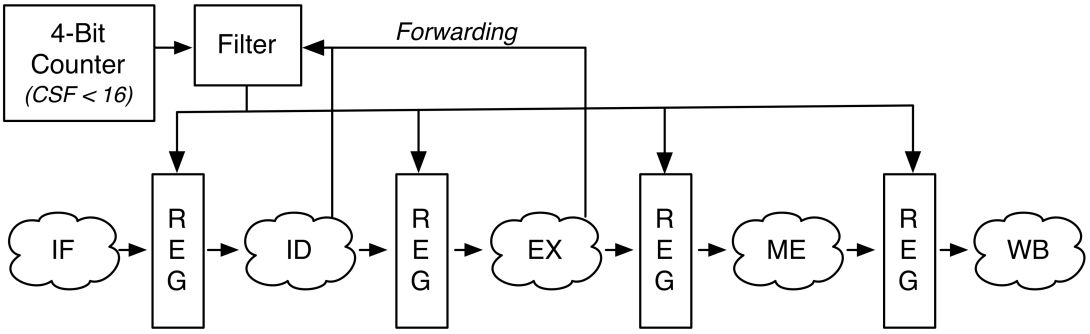


Figure 3.1: Five-Stage Core Architecture Adaptation

cycle the control logic must determine if the operation has been completed, similarly a CSF of 10 means every tenth cycle the control logic would determine if the output was valid or not. It should be noted that when using the CSF values greater than 1 the system clock remains the same, but the effective clock rate of the execution core becomes the clock rate divided by the CSF.

The standard five-stage processor utilizes boundary registers to save the results of one stage and hold the values constant as the input to the following stage, the clock rate is then defined by the maximum delay of any of the five stages. To adapt this to the target hardware the CSF is used and when implemented as shown in Figure 3.1 can provide an equivalent architecture to the standard five-stage architecture. The primary difference is that the counter via the filter block controls an instruction's transition from stage to stage. The filter block is used to dynamically increase the window size and allow extra CSF windows to be used for data forwarding while minimizing the number of unused cycles when forwarding is not needed. A brief

explanation of how the instruction decode stage and control logic may be implemented can be found in Appendix B.

3.2 Execution Cores

Four different execution cores have been proposed, designed and evaluated. Each execution core has been developed around a specific module or design goal ensuring that each core could ideally fit an application highly dependant on a given instruction type or behavior. The first core is presented in detail to give a better understanding of how the proposed cores function and scale with varying data path sizes. The description of the three remaining cores is focused on their differences from the first core and what niche they are attempting to fulfill. Lastly, two methods for providing branch and jump operations are presented, which can work with any of the 5-stage cores.

3.2.1 Half-MAC Core

The first core is referred to as the half-MAC (or ½MAC) core and was originally proposed in [14], since it is designed around the MAC unit and moving its results to the output as rapidly as possible once the computations are completed. The term half is used since the unit's 64-bit output is treated as 2 distinct outputs, one for the lower 32-bits and one for the upper 32-bits. As is shown in Figure 3.2, the logic, comparator, and shift/rotate units' outputs are all tied to a single TriMux. The output

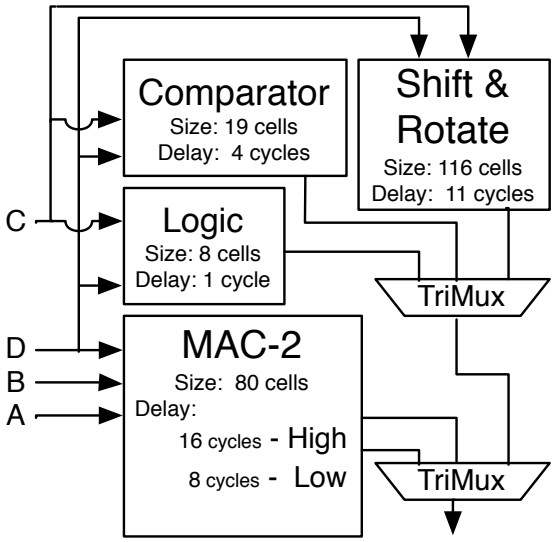


Figure 3.2: ½MAC Execution Core

of this TriMux is then tied to the third input of the lower TriMux, which is shared with both of the MAC-2 unit's outputs, to produce the final execution stage output. It should be noted that using this method requires two cycles to commit the results of 64-bit MAC operations, but in keeping with MIPS functionality this matches the Hi and Lo register segregation and allows for fewer cells to be used in the implementation of the register file. This core is best used for general-purpose programs where the MAC unit would be heavily relied upon.

The width of the data path plays a critical role when determining core delay and size. As shown in Table 3.1 the modules can easily scale with the average a size increase approximated by multiplying the current size by 2.38 when the data path width doubles, the average delay increase is kept to a scaling factor of 1.34. This shows that the ½MAC core and all other cores utilizing the modules from Chapter 2

TABLE 3.1: Core and Module Sizing Based on Path Width

Module/Core	Data Path Size							
	8-bit		16-Bit		32-Bit		64-Bit	
	# of Cells	Delay	# of Cells	Delay	# of Cells	Delay	# of Cells	Delay
MAC-2 (N-bit)	8	2	24	4	80	8	288	16
MAC-2 (2N-bit)	8	4	24	8	80	16	288	32
Shifter	20	5	44	7	116	11	356	19
Logic	2	1	4	1	8	1	16	1
Comparator	5	3	9	3	19	4	37	4
TriMUX	2	1	4	1	8	1	16	1
1/2MAC*	39	3	89	5	239	9	729	17

* Delay is the tuned delay for the N-BIT MAC-2 operation.

scale in a very similarly fashion. The size scales at an average rate of 2.67 and the delay increases at a rate of 1.79 cycles for every doubling in path width. When clocking this unit an ideal CSF factor is 9, because it allows for 32-bit operations (excluding shift operations) to be completed and output in a single control window while minimizing the number of wasted cycles for other operations. Additionally the 64-bit operations and shift operations can be completed in the second CSF window. For the remainder of this study comparisons will be based on 32-bit path widths unless otherwise noted.

3.2.2 Shift Centric Core

The Shift Centric core is built around the shift/rotate unit. Similar to the ½MAC unit the Logic and MAC-2's outputs are fed into a TriMux, which is connected to the output TriMux that is shared with the shift/rotate and comparator units. This Unit is shown in Figure 3.3 and is ideally suited for programs in which

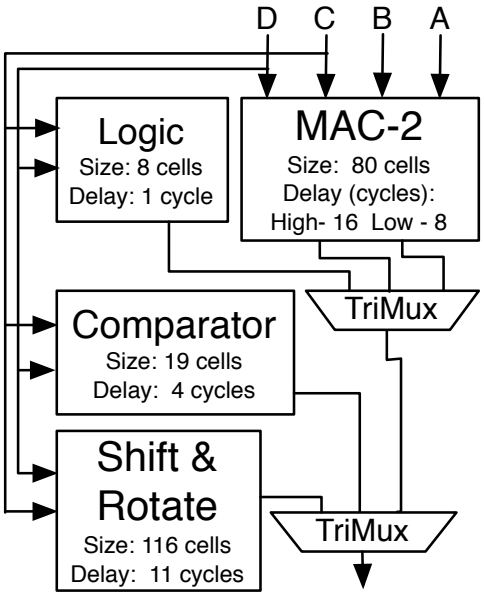


Figure 3.3: Shift Centric Core

division is carried out algorithmically through shift and comparison operations. Choosing a CSF for this unit is not as straight forward as other units because the shift operation requires the greatest amount of delay for a 32-bit output and unless the program being executed requires a large number of shifts a significant number of wasted machine cycles will result. An ideal CSF of 12 is best for shift dependant programs, while best performance would be achieved with a CSF of 10 if a program were not dominated by shift operations.

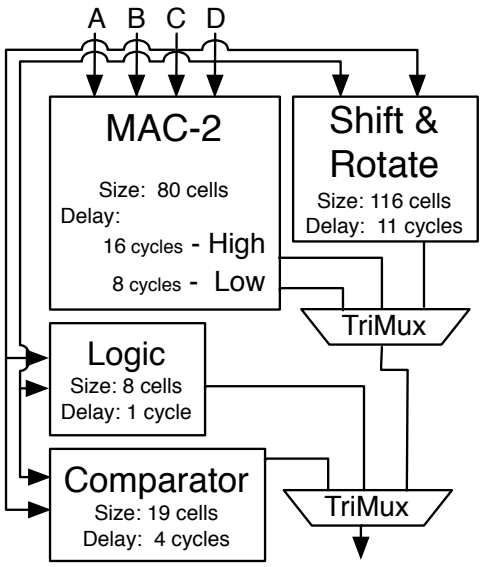


Figure 3.4: Comparator Centric Core

3.2.3 Comparator Centric Core

The Comparator Centric core is built around the comparator and logic execution units. The primary units are tied to the output TriMux along with the output of a TriMux connected to the MAC2 and shift/rotate unit, as shown in Figure 3.4. This module is best used for systems that want to maintain a small CSF and are not concerned with complexity of control logic. This is due to the core being designed around the 2 units that require the fewest machine clocks to carry out their designated operations, and can be tuned to run with a CSF of 5 while minimizing wasted machine cycles. This core is particularly well suited for executing pre-organized code blocks that rely heavily on logic and comparisons operations rather

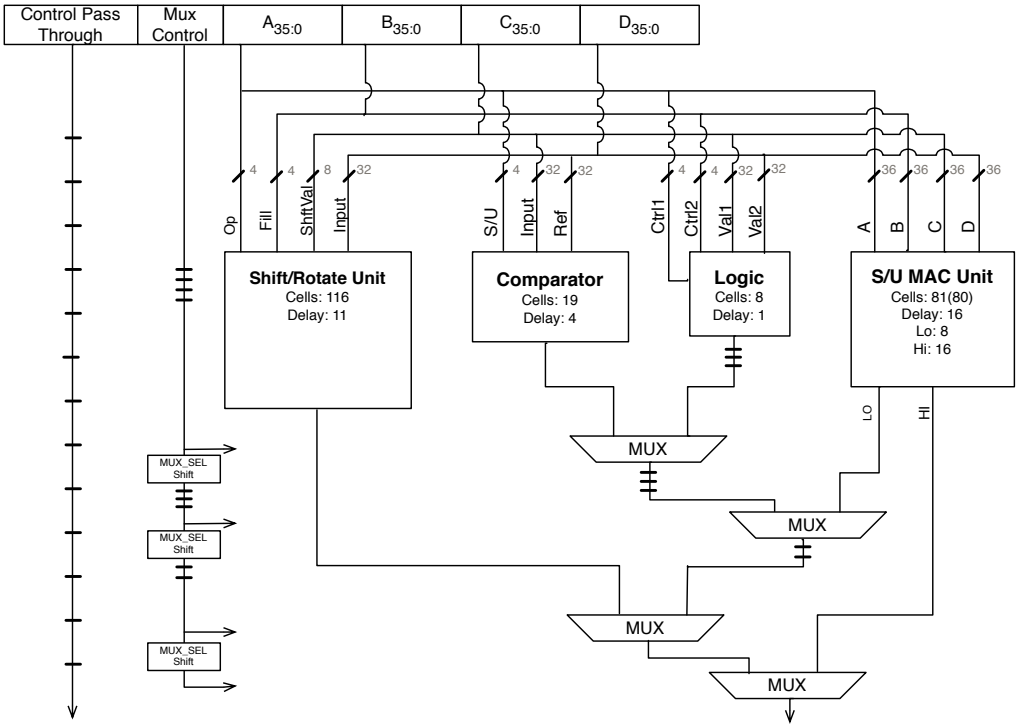


Figure 3.5: Original ExCore Layout

than branching and arithmetic intensive code blocks and could be optimal for use as a microcontroller or monitoring the output of an A/D.

3.2.4 ExCore

The ExCore design is significantly different from the other cores in design philosophy, in that it was architected to balance the unit delays allowing for single clock execution time for all 32-bit operations and 2 system cycles for 64-bit operations. This was particularly useful when the design was originally implemented, because the core was still utilizing 2:1 multiplexers and this increased the delay for some modules' outputs in reaching the stage output, as shown in figure 3.5. After having transitioned to utilizing TriMux multiplexers the implementation became very

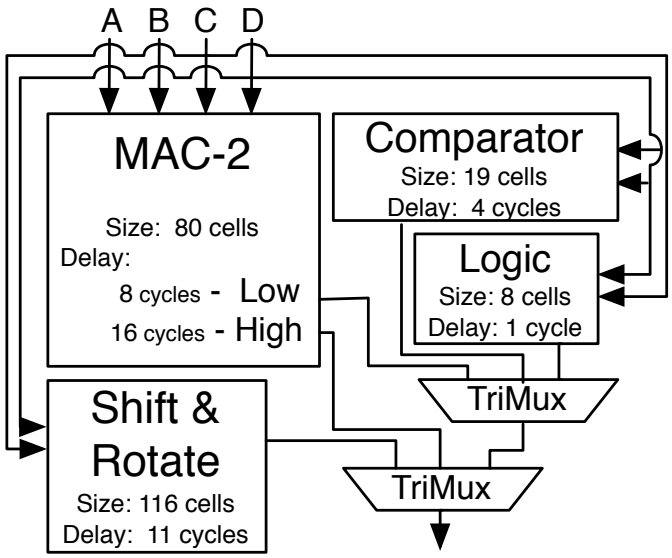


Figure 3.6: ExCore

similar to the shift core. The primary difference is that the MAC2's outputs are split between the primary output TriMux and the upper TriMux, which is also shared with the logic and comparator units, as shown in Figure 3.6. When appropriate delays are added between execution units and their primary TriMux the design is best run tuned with a CSF of 12, however if few shift operation are required then a CSF of 9 is more ideal with shift operations also requiring 2 system cycles and transforming the core into a variant of the ½MAC core.

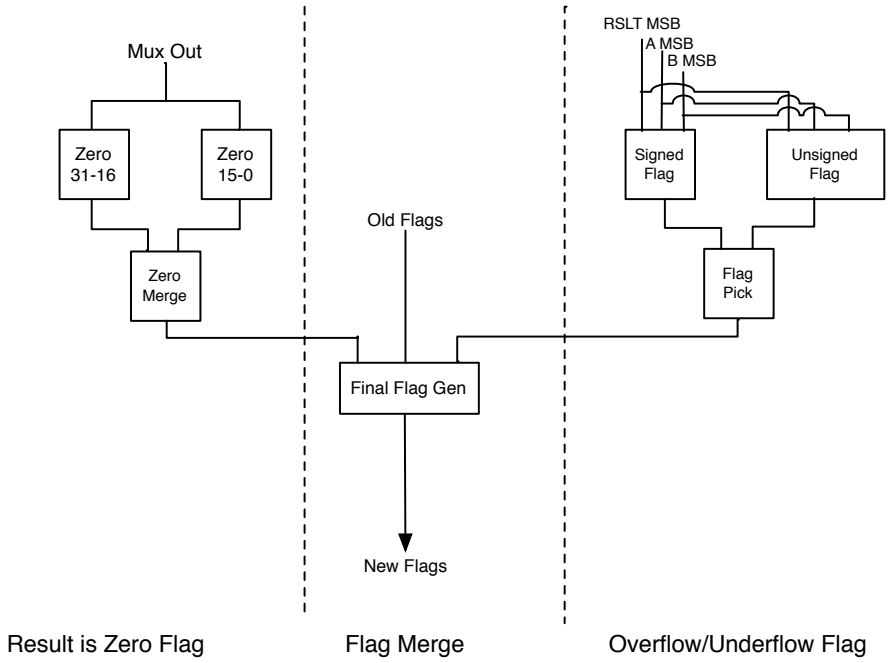


Figure 3.7: Flag System

3.2.5 Branch and Jumps

Branch and jump instructions are vital functions of any processor; typically a zero detection bit controls the PC replacement mux that facilitates branch instructions and replaces the current next PC value with the branch/jump target address. There are two methods for implementing the zero detect signal in the cores, the first relies on connecting the comparator unit's output to the zero detection output and along with an enable bit controlling the PC replacement mux. This method does not require any additional hardware or delay to be added to the core. The other option shown in figure 3.7 is to implement a flag system, which has been

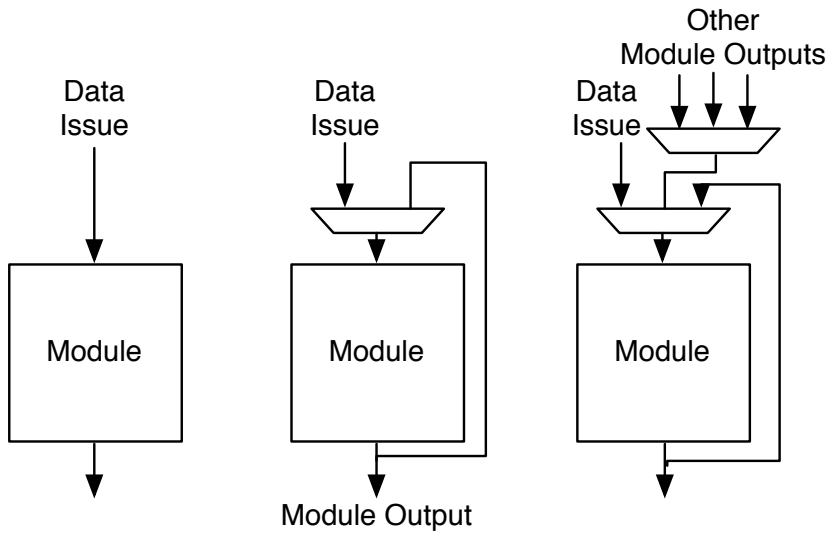
designed and includes overflow/underflow flags, but requires 13 cells and adds 3 to 4 cycles to the total execution stage delay depending on the core being used.

3.3 Forwarding Mechanisms

To accommodate data forwarding and reduce system stalls, three separate forwarding mechanisms have been implemented for use with the proposed processing cores. In the remainder of this chapter each mechanism's primary functions will be detailed as well as the hardware requirements to implement the forwarding mechanism.

Module Forwarding: The Module Forwarding (MF) scheme, shown in Figure 3.8A and is the simplest of the forwarding mechanisms in that each module's output can be sent directly back to the instruction decode stage of the processor for use in the next issued instruction. Since the instruction decode hardware is generally a network of LUTs and muxes to move data to the correct location and generate control signals. For the purposes of comparisons this forwarding scheme is considered to have no hardware or delay overhead for the execution stage of the processor.

Module with Local Forwarding: The Module with Local Forwarding (MwLF) method, shown in Figure 3.8B, is very similar to the module forwarding scheme, except that an input multiplexer is added to each of the inputs of the processing units and connected to the given units output. This allows for processing



(A) MF Scheme (B) MwLF Scheme (C) M2MwLF Scheme

Figure 3.8: Module Forwarding Schemes

units to be used repeatedly with the same data without incurring large delays when the data is at the unit's output. The hardware and time requirements of the MwLF scheme compared to MF scheme are 48 cells and 1 machine cycle, but this could be reduced if the units that typically do not run sequentially on the same data like the comparator were not allocated this extra forwarding hardware.

Module-to-Module with Local Forwarding: The Module-to-Module with Local Forwarding (M2MwLF) scheme, shown in Figure 3.8C, builds further upon the MwLF scheme in that each module is able to directly forward its output to any of the other modules. This is done by adding an additional TriMux to the MwLF scheme that can select as an input the output of any of the three other processing units. When compared to the MF scheme this method required the addition of 96

cells and 2 machine cycles or double that of the MwLF scheme. Like the MwLF scheme the hardware and time overhead could also be reduced by selectively eliminating local and module-to-module forwarding where the performance benefits do not outweigh the additional hardware.

Chapter 4

Five-Stage Core and Forwarding Scheme Performance Evaluation

This chapter will focus on the evaluation of our approach and the effectiveness of the designed cores. The evaluation will be divided into three components, the first being the lessons learned from early research, which will provide an overview of what was learned from the FPGA research and a critique of the target hardware when compared to FPGAs. The performance evaluation will focus primarily on the effects of different control synchronization factors (CSFs) and forwarding schemes and how they effect the cycles required to execute a SPEC integer benchmarks. Finally a comparative analysis will be presented that focuses on how our approach differs from other reconfigurable hardware based processors in approach, performance, and adaptability. Using these evaluations in combination a clear picture will be given

TABLE 4.1: AES Implementation Comparisons

Scheme	Slices	Minimum Clock period (ns) (in MHz)	Throughput (Mbps)	Throughput per Slice (Kbps/Slice)
NIST	5964	50.8 (19.69)	252.0	42.3
OPEN	*6910*	66.4 (15.06)	*192.8*	*27.9*
Gaj et al.	2902	38.6 (25.91)	331.5	114.2
Dandali et al.	5673	**36.3 (27.55)**	**353.0**	62.2
Elbrit et al.	3528	43.5 (22.99)	294.2	83.4
DOR	2448	37.2 (26.88)	344.1	140.6
DOR+K	**2439**	36.6 (27.32)	349.7	**143.4**
Dual Stage	3475	*73.2 (13.66)*	349.7	100.6

showing the strengths and promise of the target hardware as well as the versatility of our approach in developing highly configurable execution cores.

4.1 Past Lessons

In [15], three AES encryption cores, termed DOR, DOR+K, and Dual-Stage, were proposed that included various optimization to the standard encryption core including custom routed Shift Row transformations, key storage registers, optimized Mix Column transforms and multi-round hardware implementations. These designs were then compared to similar work from [16], [17], and [18], which showed that the proposed schemes provided comparable levels of throughput while consuming the least amount of power, which is detailed in Table 4.1. Additionally the DOR and DOR+K schemes provided the best throughput vs. area metric of the compared designs, although some of the designs used for comparison included decryption hardware.

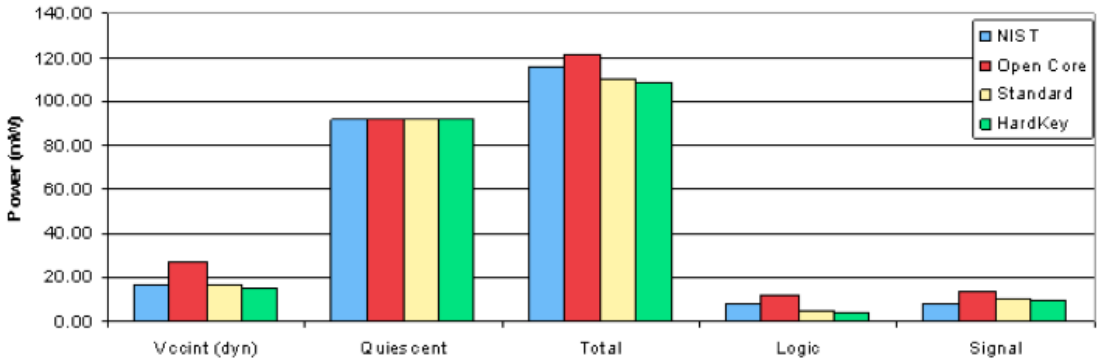


Figure 4.1: Power Consumption of FPGAs

When the proposed encryption designs along with a design obtained from NIST [19] and the Open Cores project [20] were analyzed using the Xilinx's XPower software a large insight into the performance and power characteristics of FPGAs was obtained. As shown in Figure 4.1 when operating at a speed of 5 MHz or an average rate of 25% of the maximum clock speed, quiescent power becomes the dominating power consumption component. This is not unexpected though because until recently FPGAs have typically been designed to provide the fastest performance possible without regards to the power consumption rates. Further research in [7] showed that regardless of design or operating speed the quiescent power required by an FPGA did not change, which shows that FPGAs are not able to disable unallocated parts of the device to reduce base power requirements and that this component is entirely dependant upon implementation choices and technology. This shows that FPGA choice is of more importance than design optimizations when it comes to reducing power consumption in certain environments regardless of the technology size, since

FPGAs utilizing the same transistor sizes can have vastly different power characteristics.

Examining the items learned from the FPGA research helped to incentivize the move to the medium-grain reconfigurable hardware. It should be noted that the AES modules introduced in Chapter 2.1 are capable of being used to implement any of the designs used in [7] on the target hardware. Additionally some designs that were not used for comparison because they focused on minimal area like [21] could also be implemented using the proposed modules and the hardware's integrated pipeline. One important characteristic of the reconfigurable hardware is that the clock is defined by the hardware implementation and technology, not by the design as with FPGAs. This is possible in large part because of the built in pipelining that allows for modules to continuously process data compared to FPGAs where chained logic blocks without buffer registers often limit the maximum clock rate. One example of this is that the DOR and DOR+K scheme were able to run up to speeds of 28 MHz, while the AES modules presented in section 2 can run at speeds reaching 720 MHz when utilizing the built in pipeline of the target hardware and do not require allocating additional hardware for buffer registers. If the natural pipeline were not to be used the speeds of the AES module would when implementing the DOR+K scheme be conservatively approximated at 72 MHz for a chip using the same technology as the Spartan 3E.

Another benefit of the target hardware is the ease with which new technologies can be adopted and fully utilized, since everything is based on a memory cell with no specialized hardware needing to be redesigned. As shown in [5] the power reduction in leakage current alone of FinFETs versus standard CMOS technologies is appealing and can increase clock speeds by more than 100% when compared to 45nm CMOS circuits. Similar research examining future implementations utilizing CNFETs (Carbon-nanotube FETs) has shown that a static power reduction of up to 98% and a 43% reduction in memory access delay may be achieved for 8-T memory cells when compared to standard CMOS technologies. Additionally the target hardware is divided into groups of reconfigurable elements that if not being used can be disconnected from the power supplies to eliminate idle leakage power.

4.2 Performance Evaluation

In this section the proposed schemes for flexible processing units are evaluated. These evaluations include different forwarding methods for performance and size, as well as the effects of varying the CSF. This discussion proceeds as follows: The method used to analyze the designs is first presented. Next the results of the analysis are detailed. Lastly an analysis of what the results show is presented.

Before the evaluation process is detailed it should be noted that the range across which the CSF will be varied is 10 to 20. The reason for this is due to the time

it takes to generate a new PC value using an addition only MAC unit. The modified unit requires 8 cycles to compute a 32-bit addition and another cycle would be required to store the data into a register. A method could be devised in which multiple new PC values are generated at once allowing for CSFs less than 10, but for this evaluation 10 will be the minimum considered CSF value.

To accomplish this evaluation a program has been written that computes the number of cycles a program takes to execute. By using the number of clock cycles required, comparisons may be made across core architectures, forwarding methodologies and CSF values without the need to translate data from one technology to another. A MIPS trace file (in machine code) is the input to the program, which takes each instruction and calculates the number of cycles required to execute the instruction, as well as any forwarding delays to communicate the current instruction's results to the subsequent instructions. If the resulting number of cycles does not match a multiple of the CSF then extra cycles are added to synchronize with the control system. The program continues to process instructions until the end of the trace file. In the event that an instruction is not supported a penalty is added to the total cycle count equivalent to a 32-bit MAC operation. It should be noted that the forwarding delay is computed using the forwarding scheme's cycle overhead plus the time to route data from one component to another using a

best-case placement. This analysis method is similar to the approach used in [22] for the design of a programmable instruction set computer.

The MIPS trace files that are used for this analysis are the SPEC Li (Lisp), EQNTOTT (Equation to Truth Table), ESPRESSO (Boolean Function Simplification), COMPRESS (Text Compression Algorithm), and SC (Spreadsheet Application) integer benchmarks [23]. These test benchmarks range in size from 720K to 804K instructions and total 3.85 million instructions, with at most 0.07% of the instructions not being supported, which is made up mostly of coprocessor instructions. However, each benchmark does require a small number of division operations (.04% maximum); these instructions currently need be implemented algorithmically in software. It is estimated that these instructions would take 20 times the standard 64-bit MAC-2 instruction time or 320 machine cycles. The simulator also has been modified to replace the standard MIPS NOP instruction that is a left shift of zero by zero with a logical ANDing of zero with zero, since the shift unit requires the most cycles to compute its results.

TABLE 4.2: Optimal Performance Characteristics for Cores and Forwarding Schemes

			Benchmark										All Benchmarks		
			Compress		EQNTOTT		Espresso		Li		SC				
Core	Forwarding Type	Best CSF	Clks per Inst	CSF per Inst.	Clks per Inst	CSF per Inst.	Clks per Inst	CSF per Inst.	Clks per Inst	CSF per Inst.	Clks per Inst	CSF per Inst.	Machine Cycles ($x10^6$)	Clks per Inst	CSF per Inst.
1/2 MAC	MF	10	16.42	1.64	15.07	1.51	15.96	1.60	14.07	1.41	14.79	1.48	58.699	15.26	1.53
	MwLF	10	14.85	1.49	13.57	1.36	14.44	1.44	13.20	1.32	13.37	1.34	53.397	13.88	1.39
	M2MwLF	10	**14.24**	1.42	**13.07**	1.31	**13.52**	1.35	**12.76**	1.28	**12.66**	1.27	**50.932**	**13.24**	1.32
Shift Core	MF	10	16.42	1.64	15.07	1.51	15.96	1.60	14.07	1.41	14.79	1.48	58.699	15.26	1.53
	MwLF	11	15.98	1.45	14.73	1.34	15.51	1.41	14.45	1.31	14.52	1.32	57.815	15.03	1.37
	M2MwLF	11	15.30	**1.39**	14.22	**1.29**	14.51	**1.32**	13.97	**1.27**	13.82	**1.26**	55.204	14.35	**1.30**
Comp-arator Core	MF	10	16.42	1.64	15.07	1.51	15.96	1.60	14.07	1.41	14.79	1.48	58.699	15.26	1.53
	MwLF	11	15.98	1.45	14.73	1.34	15.51	1.41	14.45	1.31	14.52	1.32	57.815	15.03	1.37
	M2MwLF	11	15.30	**1.39**	14.22	**1.29**	14.51	**1.32**	13.97	**1.27**	13.82	**1.26**	55.204	14.35	**1.30**
ExCore	MF	10	16.42	1.64	15.07	1.51	15.96	1.60	14.07	1.41	14.79	1.48	58.699	15.26	1.53
	MwLF	11	15.98	1.45	14.73	1.34	15.51	1.41	14.45	1.31	14.52	1.32	57.815	15.03	1.37
	M2MwLF	11	15.30	**1.39**	14.22	**1.29**	14.51	**1.32**	13.97	**1.27**	13.82	**1.26**	55.204	14.35	**1.30**

TABLE 4.3: Cycles (in Millions) Needed to Complete 5 Benchmarks Totaling 3.85M Instructions

		CSF Factor	Core Type 1/2MAC	Shift Centric	Comparator Centric	ExCore
Forwarding Scheme	MF	10	*58.699*	*58.699*	*58.699*	*58.699*
		11	63.688	63.688	63.688	63.688
		12	67.906	67.366	67.906	67.366
		13	72.980	72.980	72.980	72.980
		14	77.165	77.165	77.165	77.165
		15	77.140	77.140	77.140	77.140
		16	80.687	80.687	81.060	80.687
		17	85.451	84.600	84.720	84.441
	MwLF	10	*53.397*	67.966	67.965	67.965
		11	57.815	*57.815*	*57.815*	*57.815*
		12	63.027	63.027	63.027	63.027
		13	65.139	64.553	65.139	64.553
		14	69.506	69.507	69.506	69.506
		15	72.486	72.487	72.486	72.486
		16	74.162	74.162	74.536	74.162
		17	77.956	77.668	77.787	76.947
	M2MwLF	10	***50.932***	65.500	65.500	65.500
		11	55.204	***55.204***	***55.204***	***55.204***
		12	60.210	60.211	60.210	60.210
		13	62.115	61.529	62.115	61.529
		14	66.262	66.262	66.262	66.262
		15	70.865	70.865	70.865	70.865
		16	73.656	73.713	74.087	73.656
		17	77.415	77.126	77.250	76.405

Minimum Cycles/Instructions for Forwarding type
Minimum Cycles/Instructions for Design

Evaluation results are shown in Table 4.2 and 4.3 and Figure 4.2 and 4.3. As is shown in Table 4.2 the more advanced forwarding schemes are able to provide a sizeable reduction in total cycles, the MwLF scheme averages 6.27% and the M2MwLF scheme averages 9.29%. This translates to an average savings of 6.28 and a

maximum of 7.84 million cycles for the MwLF scheme and an average of 8.38 and a maximum of 10.9 million cycles for the M2MwLF scheme.

It can be observed that the Shift Core, Comparator Core and ExCore actually experience a decrease in performance when using the more advanced forwarding schemes and a CSF of 10. The main reason for this is that the 32-bit MAC2's forward

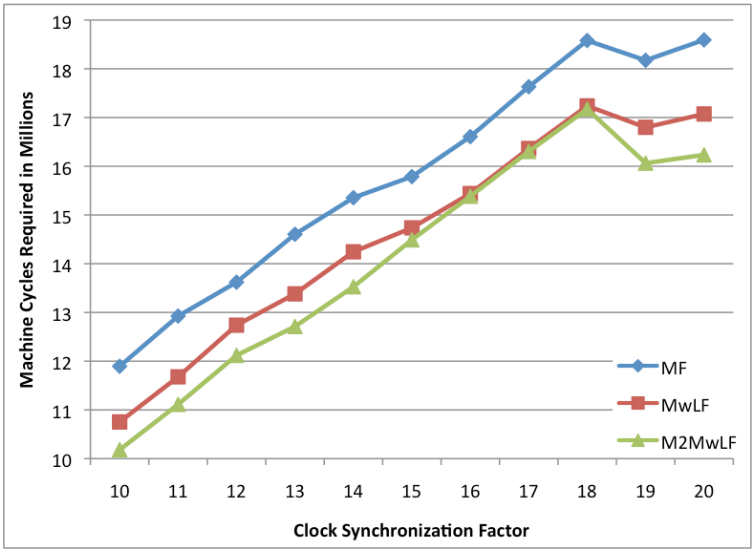


Figure 4.2: ½MAC Performance for SC Benchmark

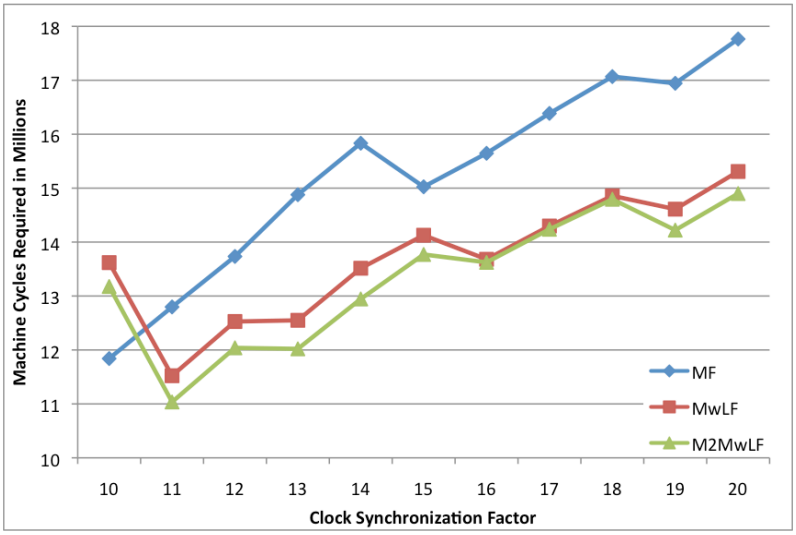


Figure 4.3: ExCore Performance for Compress Benchmark

data path in these instances becomes optimized for 11 cycles due to the input mux on each execution unit. This is more evident in Figure 4.2 and 4.3, which show the machine cycles for the ½MAC and ExCore cores for the SC and Compress benchmarks respectively. The ½MAC Core scales in an almost linear fashion until a CSF of 19 is reached where a relative minimum occurs. The ExCore design exhibits a very different response to increasing the CSF; there is virtually no change between 12 and 13 machine cycles per system cycle, and relative minimums at 11, 16, and 19. Additionally the ExCore, which was optimized for a CSF of 12, exhibits the most consistent performance of the cores regardless of forwarding mechanism for CSFs greater than 12.

When using these results it becomes apparent that the core can be easily optimized for any type of program with only a small amount of foresight as to the types of operations that a program will be executing. There are some trade-offs that must be considered: a 10.74% reduction in machine cycles requires a 20% increase in cell count, or a 14.89% reduction in machine cycles incurs a 40% increase in cell count. If the designer is striving for a pure general purpose platform, where all instructions must be completed as fast as possible, then the size of a design becomes the deciding factor in choosing which core and forwarding mechanism to utilize. In this case the ½MAC offers the best performance when utilizing the MwLF and

M2MwLF schemes, but offers marginally poorer performance than the ExCore unit with MF for varying CSFs.

4.3 Comparative Analysis

In this section a comparison with other reconfigurable hardware based processors is presented. There are a number of approaches that can be taken when implementing reconfigurable hardware based processors; four approaches have been selected that represent a significant portion of the research done. A brief discussion of each approach is presented along with specific implementations utilizing the approach. Once each approach has been detailed, a comparison of these approaches with each other and the proposed designs is then presented.

When considering design area for a comparison metric Xilinx characterizes their design in terms of slices, four input LUTs, and block RAMs, while Altera primarily uses logic elements. These two metrics can be approximated to provide fair comparisons between designs based on different FPGAs. However, there is no reliable way to compare them with a cell from our medium-grain reconfigurable hardware. Thus, to avoid unfair area comparison this analysis does not consider area as a metric. This leaves implementation speed as the only fair metric for offering valid comparison across devices and technology.

The first approach for utilizing reconfigurable hardware for processors is to use a soft processor developed by another party for a specific platform. Xilinx has developed and maintained the MicroBlaze and PicoBlaze series of soft processors [24], which can be configured to offer tuned performance for a given design's specifications and can reach speeds of 85 and 154 MHz on the Spartan 3 and 6, respectively. Similarly Altera has provided the NIOS II [25] soft processor core for its FPGAs that can be configured for fast, standard or economic implementations, with the standard implementation maintaining speeds up to 110 and 145 MHz on the Cyclone II and III FPGAs. While these options provide a streamlined mechanism to implement processors on reconfigurable hardware, these implementations usually provide low performance when pared with other custom hardware. This is why Xilinx started to embeds Power PC cores and Altera embeds ARM cores into their high-end FPGAs.

The second approach builds off of existing soft processors and modifies their organization/architecture through the use of the reconfigurable components to improve performance. Gonzalez et al. [26] added reconfigurable encryption coprocessors to the MicroBlaze soft processors that could be dynamically reconfigured on a Spartan 3 FPGA. The work showed that by using a reconfigurable coprocessor three different encryption ciphers may be sped up in hardware without dramatically increasing area requirements, but were only able to maintain a speed of 65 MHz for verification. Similarly Lysecky and Vahid [27] focused on making loop execution

more efficient, by identifying repeated code blocks and using reconfigurable hardware to replace the repeated instructions. In their work they found that the MicroBlaze was able to operate at speeds of 85 MHz while the reconfigurable coprocessor could reach speeds of 250 MHz. Both of these examples are able to boost the performance of their target processors, but require a prior knowledge of the code being executed or the addition of hardware that can analyze code and initiate reconfigurations for possible optimization. This combined with the hardware required to dynamically reconfigure parts of a chip during execution are not trivial and come with an increase in design area and execution time caused by reconfiguration delays.

Other researchers have looked to unique architectures that not only adapt well to specific tasks, but to implementation on reconfigurable hardware. In [28] a parallel associative processor was built with multi-comparand multi-search operations that could reach speeds of 104 MHz on a Spartan 3. This type of processor is particularly well suited for highly parallel application including graph theory and matrix/list computations, and could be scaled based on the desired block size. Similarly a double-issue processor was built in [29] that operates much like a VLIW processor, since 2 instructions of differing types can be issued concurrently. While the instructions per cycle factor would increase with multi-issue CPUs, the processor speed suffered, as it was only able to reach a speed of 63.3 MHz. Both of these processors faced the same

problems though, which is one of size and resource utilization. As the size of the processor increased for larger data paths and free space on the device decreased the speeds started to drop dramatically because of the overhead of routing data around the target device and trying to fit parts of the processors close to specialized hardware blocks.

Researchers have also focused on schemes to implement MIPS processors utilizing reconfigurable hardware. Li et. al. [30] proposed an economic MIPS design that strived to minimize design area by taking full advantage of embedded components such as multipliers while operating at speeds up to 64 MHz on a Spartan 3. Similarly Ramdas et. al [31] designed an integer MIPS processor using Handel-C that reached speeds of 45 MHz on Virtex-II for facial recognition. Lastly in [32] researchers attempted to build a low-power MIPS processor and by utilizing a modified pipeline to increase performance were able to operate at an effective clock rate of 100 MHz on a Spartan 3E. Power however is a particularly important area for FPGA research due to their poor power consumption characteristics, detailed in [33, 34]. Each of these groups were able to successfully implement a MIPS processor for their intended use, but only [32]'s performance stood out and that design required a significant modification to the pipeline and when running at maximum speed would not efficiently consume power as the designers intended, which has been demonstrated in [7].

Now having presented some of the various design focuses, we turn our attention to the use of specialized hardware on chip such as embedded multipliers and memory to increase speed and performance. While the efficient use of available resources on the target chip is critical, it also locks an implementation to a specific placement on a chip, and if a fault occurs in or near the specialized unit the device can become unusable. This is one of the primary appeals of our approach in that the platform offers no specialized hardware, which means that the design can be placement agnostic and the most efficient way to route a design can be used regardless of the technology used. The number and word-lengths of the units may also be tailored to the target application's requirements using the proposed reconfigurable hardware. Additional performance gains can also be achieved by utilizing the built in hardware pipelining, which allows for increased clock rates, without a substantial area penalty. A potential drawback of utilizing the pipeline hardware is the increased complexity for: control logic, clearing the pipeline, and the time it takes to repopulate it.

Another characteristic that differentiates our approach from the others is that by not utilizing specialized hardware on chip; the reconfigurable hardware can be transitioned very quickly to other technology types since everything is memory based. As is shown in Table 4.4 the proposed hardware and execution cores are able to provide competitive levels of performance with the other designs. At 90 nm the

TABLE 4.4: Design Technology Size and Processor Clock in MHz

	Technology		
	150 & 90 nm CMOS	65 & 45 nm CMOS	21 nm FinFET
Target Hardware			
½MAC *(csf)*	720(10)	2000(10)	5000(10)
ExCore (csf)			
Module Forwarding	720(10)	2000(10)	5000(10)
MwLF Forwarding	720(11)	2000(11)	5000(11)
M2MwLF Forwarding	720(11)	2000(11)	5000(11)
Xilinx MicroBlaze			
Spartan 3	85	-	-
Spartan 6	-	154	-
Virtex 6	-	307	-
Altera NIOS II/s			
Cyclone II	110	-	-
Cyclone III	-	145†	-
Custom MIPS Soft Processors			
[30] on Spartan 3	64	-	-
[31] on Virtex-II	45†	-	-
[32] on Spartan 3E	100	-	-

† Denotes Larger Technology Size

½MAC and ExCore schemes' clock speeds are comparable to the other designs' clock speeds, but when transitioned to a 45 nm process the scheme offers some of the best performance of the designs and without any modifications required. It should be pointed out that the machine clock for our designs is 720 MHz and 2GHz for the 90nm and 45nm technologies. The ongoing work being done to implement our reconfigurable hardware with FinFET technology [3] has shown an even greater performance increase while using conservatively calculated machine clock speeds. Our FinFET designs show that the machine clock can easily reach 5GHz. Furthermore the power consumption of FinFET devices is expected to be much lower than standard CMOS devices, which has contributed to Intel utilizing similar technology in their current generation of Ivy Bridge processors [35].

Chapter 5

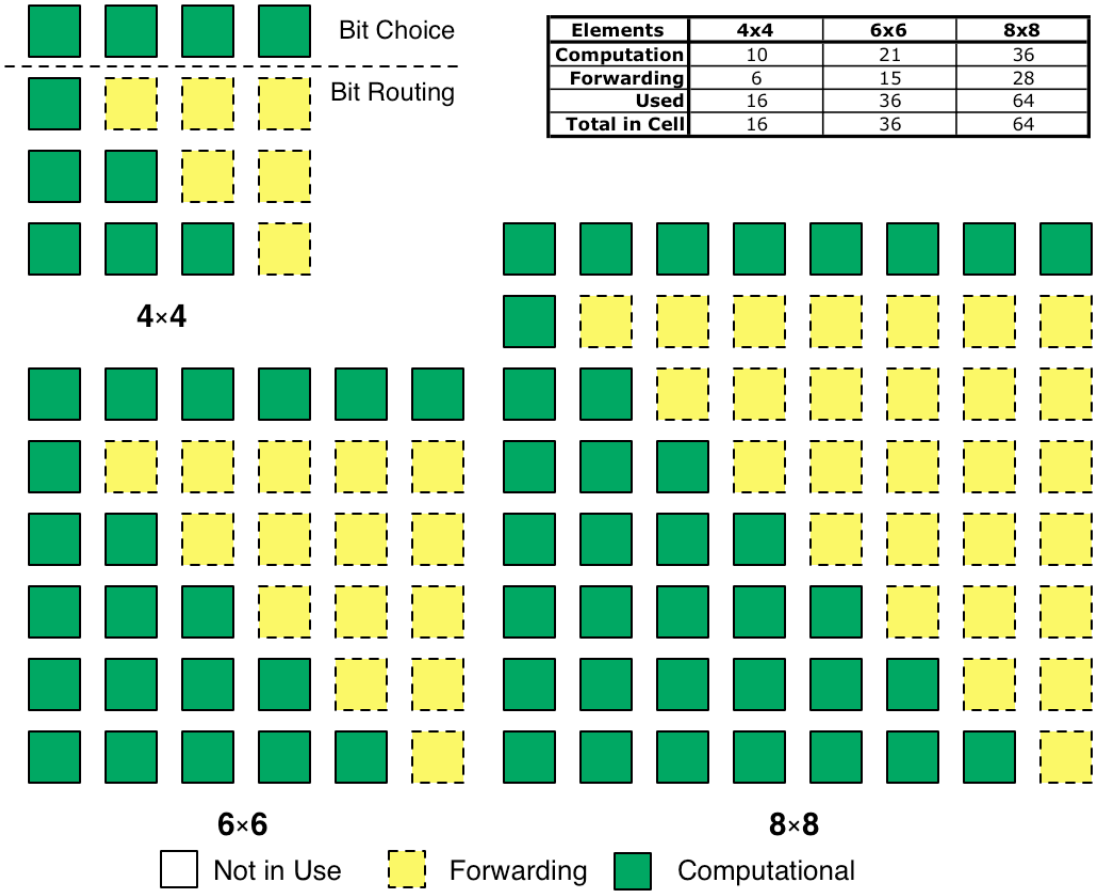
Hardware Analysis and Design Characterization

Having characterized the performance of the designed modules and execution cores based on the target hardware's performance characteristics, the attention of this study will shift focus to the cost for this performance and a more thorough evaluation of the target hardware. In this chapter the energy consumption characteristics of the modules and cores are evaluated to gain an insight into the cost of utilizing the target hardware. The first component of this evaluation is a look at how efficiently the hardware is utilized in terms of cell utilization, which will focus on how cell sizing would affect the area (cell and elemental count) and delay characteristics of the designs detailed in Chapters 2 and 3. The focus will then shift to the energy consumption rates of the components and a characterization of the individual

module's power consumption rates. Next the execution core characterization for each core type will be offered, including the effects of the varying forwarding schemes and CSFs. Lastly, the energy consumption and performance results from chapter 4 are combined to provide a picture of how the different cores perform in comparison to each other.

5.1 Hardware Design Analysis

The primary reason for the evaluation of the physical hardware design in this investigation is that the hardware development is currently ongoing and if a more optimal cellular configuration could be found the hardware may still be adapted to take advantage of the more optimal configuration. As a result this analysis will focus on the size of the cell and how efficiently the different modules utilize the included components, this approach provides an obvious metric for comparison and that is elemental utilization. This utilization can then be broken down into the following distinct categories: not in use, forwarding and computational. Not in use elements are elements contained in used cells but not serving any purpose, but have been allocated as part of a cell in use. Forwarding elements are elements that act as repeaters forwarding one or two of their inputs to either of its outputs. Lastly computational elements are any allocated element that does more than simply echo data from one input to an output, this could be to simply AND two of the inputs or compute the logic function ((A AND B) XOR C) NOR D.



Elements	4x4	6x6	8x8
Computation	10	21	36
Forwarding	6	15	28
Used	16	36	64
Total in Cell	16	36	64

Figure 5.1: Shift Cell Scaling

To complete this analysis the methodology chosen was to vary the number of elements in a cell, but keep the basic configuration of four N-bit inputs feeding into an N×N grid of elements. Each of the modules detailed in Chapter 2 could then be scaled to the new cell configurations. Additionally using timing data from [35] and [37] an estimate of the new clock speed can be realized and used for comparison.

The chosen configurations for examination are: 4×4, 6×6, 8×8, 16×16, and 32×32. An example of how the modules are scaled is shown Figure 5.1 and details how the shift cell is scaled to the 6×6 and 8×8 cell configurations. Using this same

TABLE 5.1: Cell Utilization for Varying Configurations

		4x4	6x6	8x8	16x16	32x32
Total Cells		209	122	75	33	17
Total Elements		3344	4392	4800	8448	17408
Usage	**Computational**	2540	2816	2868	3668	5304
	%	75.96%	64.12%	59.75%	43.42%	30.47%
	Forwarding	669	1244	1409	2873	5364
	%	20.01%	28.32%	29.35%	34.01%	30.81%
	Not in Use	135	332	523	1907	6740
	%	4.04%	7.56%	10.90%	22.57%	38.72%
Est. Speed (MHz)		2000	1429	1111	588	303

approach, each of the cells from Chapter 2 have been scaled to take advantage of the different cell configurations, the results of which are detailed in Appendix C, table 5.1 shows the computational and forwarding utilization for a 32-bit ½MAC core. What these results show is it that 6×6 and 8×8 cellular configurations may offer a comparable percentage overall utilization. But the original 4×4 configuration offers the highest computational usage while minimizing the number of not in use elements as well as the forwarding elements and is shown in Figure 5.2.

5.2 Hardware Energy Analysis

Before introducing the energy consumption characteristics of the different execution cores and modules the method by which the numbers were obtained must be presented. The chosen method for characterizing the energy consumption of the designs is a simulative approach utilizing a hierarchical simulation technique similar to that of [38] and [39]. The Hierarchical model is based on using two separate components, one to characterize switching activity and one to combine the activity

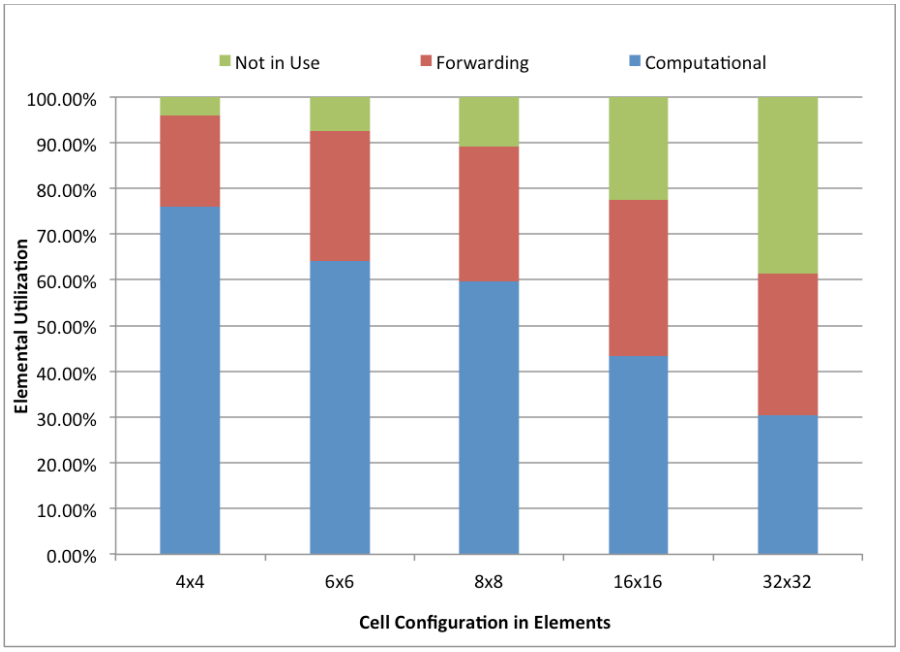


Figure 5.2: Cellular Utilization Characterization

models with energy numbers to determine the estimated energy consumption like in [40]. This is very similar to the methods used by Xilinx with their XPower tool [41] and other researchers [42].

To apply the hierarchical technique to the target hardware a program had to be made that calculated the probability of reading a 1 or 0 from memory based on the cellular configurations and input probabilities, as well as storing 1's and 0's in the output flip-flops. The input probabilities were calculated by having all data values and operation considered equally likely and if one cell is connected to another its output probability becomes the input probability of the succeeding cell(s). Once the memory access probabilities are determined the numbers can then be combined with energy consumption data from [35] and [37] which target 21nm FinFETs using a shorted gate configuration, to minimize leakage, and a 10 GHz clock to characterize each cell's

TABLE 5.2: Module Power Consumption

Module	Path Width (power in mW)								
	8-Bit			16-Bit			32-Bit		
	Cycles	Power Cycle	Static Power	Cycles	Power Cycle	Static Power	Cycles	Power Cycle	Static Power
Logic	1	0.0716	0.0002	1	0.1432	0.0004	1	0.2863	0.0008
Comparator	3	0.1132	0.0005	3	0.2158	0.0009	4	0.4412	0.0020
Shift/Rotate	5	0.4360	0.0021	7	1.0465	0.0046	11	3.1081	0.0122
MAC-2	4	0.3729	0.0008	8	1.0886	0.0025	16	3.5830	0.0084
Tri-Mux	1	0.0682	0.0002	1	0.1365	0.0004	1	0.2730	0.0008
2:1 Mux	1	0.0405	0.0002	1	0.0810	0.0004	1	0.1620	0.0008

expected energy and power consumption. Currently the memory decoder attached to each cell has yet to be implemented in the ongoing work focusing on FinFETs, but a pessimistic view of the energy required for memory access (5x increase for each of the 32 banks) should compensate for this and provide a reasonable characterization.

The first step in characterizing each core's benchmark energy consumption is to determine the power required for each of the execution modules, which is shown in Table 5.2 for path widths of 8, 16, and 32-bits. As expected the modules that increase linearly in area also increased linearly in power consumption, this is shown by the logic module's power requirement scaling from 0.07 mW for 8-bit path widths to 0.29 mW for 32-bit path widths. Similarly the shift and MAC-2 modules, which scale at a larger rate than the other modules, increases at an acceptable rate considering how many cells are required to work together for operation, as shown by the MAC-2 requiring 0.37 mW and 3.58 mW for 8 and 32-bit path widths respectively.

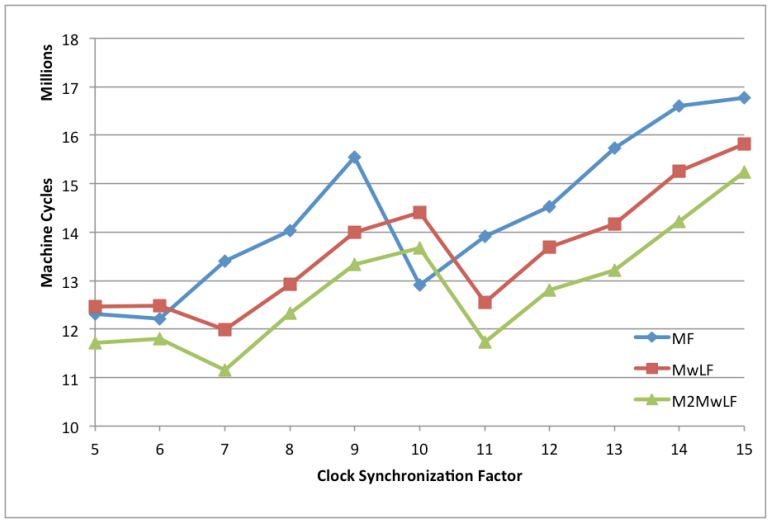


Figure 5.3: ExCore Cycles for the Espresso Benchmark

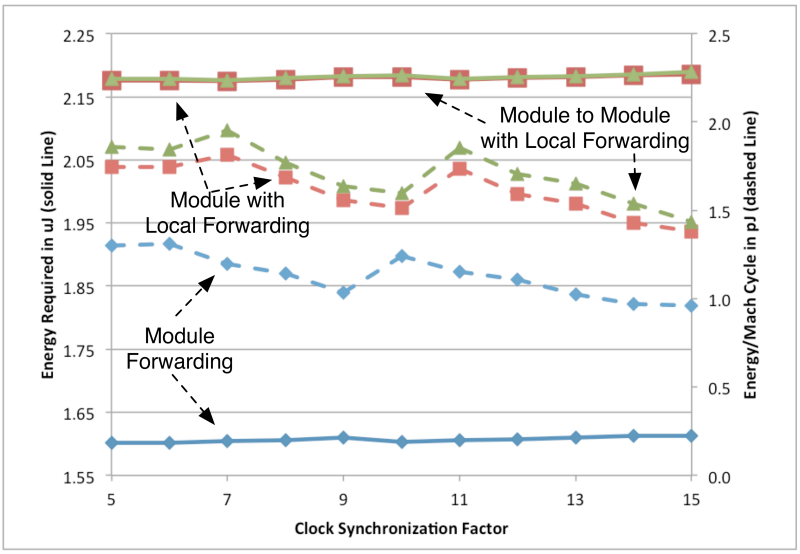


Figure 5.4: ExCore Energy Characteristics for the Espresso Benchmark

Utilizing the power characteristics of each module and component it became possible to determine the energy consumption for each of the cores while executing the SPEC integer benchmarks described in Section 5.1. The analysis of the execution core behaviors for various CSF and forwarding schemes found in the previous section yielded a series of figures, two of which are shown in Figure 5.3 and Figure 5.4 and

TABLE 5.3: Benchmark Energy Consumption

Execution Core	Forwarding Scheme	CSF	Benchmark Energy Consumption in µJ				
			Compress	EQNTOTT	ESPRESSO	Li	SC
1/2MAC	MF	10	1.6726	1.6438	1.4138	1.5568	1.3746
	MwLF	10	2.3014	2.2618	1.9559	2.1474	1.9032
	M2MwLF	10	2.3036	2.2641	1.9575	2.1496	1.9050
Comparator Centric	MF	10	1.8662	1.8310	1.5493	1.7255	1.5021
	MwLF	11	2.5095	2.4628	2.0974	2.3272	2.0358
	M2MwLF	11	2.5119	2.4653	2.0991	2.3297	2.0379
Shift Centric	MF	10	1.8447	1.8095	1.5278	1.7041	1.4806
	MwLF	11	2.4875	2.4407	2.0754	2.3052	2.0137
	M2MwLF	11	2.4898	2.4433	2.0770	2.3077	2.0158
ExCore	MF	10	1.9200	1.8844	1.6027	1.7793	1.5558
	MwLF	11	2.5889	2.5418	2.1764	2.4066	2.1150
	M2MwLF	11	2.5912	2.5443	2.1781	2.4091	2.1172

demonstrate common trend in results. What these charts demonstrate is that as the CSFs increased and the number of machine cycles grew rapidly the energy required very slowly increased due to the efficiency of shorted gate FinFET designs in minimizing leakage currents. The numeric results of this analysis are shown in Table 5.3 for the CSF values that resulted in the fewest machine cycles. The most interesting item contained in the table is that even though the M2MwLF scheme while requiring a sizable increase in area over the MwLF schemes does not greatly affect the dynamic power consumed, usually by less than 0.1% of the difference from the MF power scheme and both the MwLF and M2MwLF schemes require 35.9% more energy than the MF scheme configurations on average.

5.3 Combined Energy and Performance Analysis

The final step in the core characterizations and hardware analysis is to combine the performance and energy characterizations into a single set of data points that could help to further differentiate the schemes and pinpoint the best

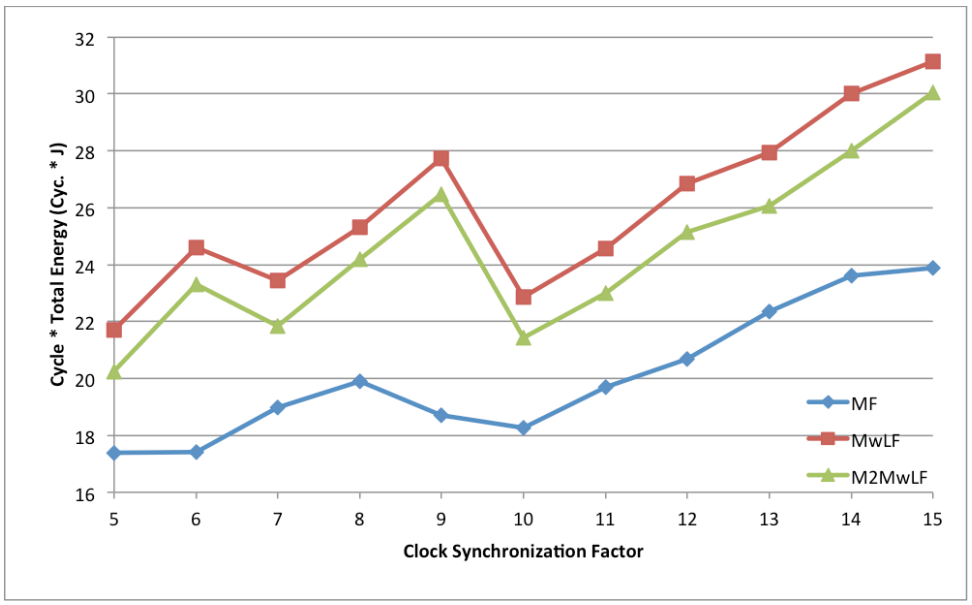


Figure 5.5: Cycles × Energy vs. CSF for Espresso Benchmark on 1/2MAC Core

combination of core organization, forwarding scheme and CSF. The settled upon metric for completing the combination of the two characterizations was to simply multiply the number of cycles with the total energy required for execution of a benchmark. When looking at a single core type with varying forwarding mechanisms the results were all very similar regardless of core, with the ½MAC's result being shown for the Espresso benchmark in Figure 5.5. The results show that the MwLF scheme increases the Cycles x Energy metric by an average of 29.1% with a minimum and maximum difference of 23.0% and 48.4% respectively. Similarly the M2MwLF scheme increases the metric by an average of 22.5% with minimum increase of 15.1% and a maximum increase of 41.6%. Although increasing the CSF can offset some of the increase it can be concluded that unless minimal execution time is the ultimate goal the cores with simple module forwarding offer the best balance of performance

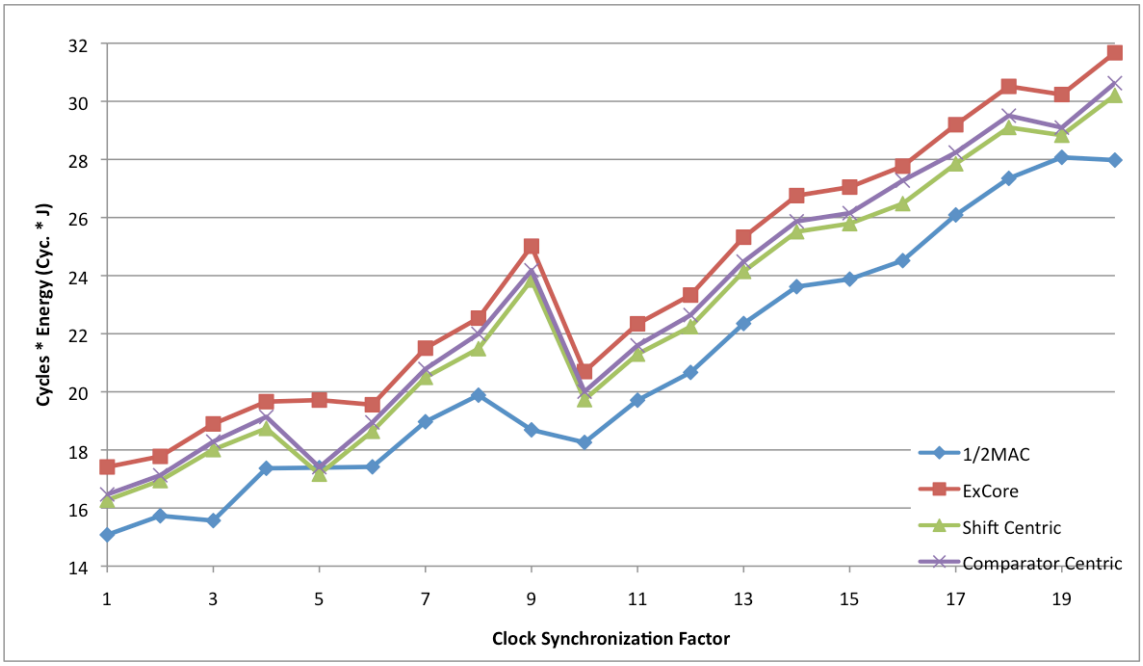


Figure 5.6: Core Cycles × Energy Characterization for Espresso Benchmark Utilizing Module Forwarding

and energy consumption. The results also show that the only time the MwLF scheme should be used is if the designer does not have the area to implement the M2MwLF scheme, but still requires the lowest possible execution time.

Lastly, as shown in Figure 5.6 and Figure 5.7 the execution cores themselves have been compared using the two most promising forwarding mechanisms, MF and M2MwLF. These figures when used together show that for the Espresso Benchmark the ½MAC core provides the best balance of performance regardless of forwarding type. The more advanced forwarding schemes help to narrow the performance gap, but it is never fully overcome in the range of usable CSFs, being 9 and higher. While the ExCore, Shift Centric, and Comparator Centric cores still provide alternatives for programs utilizing specific instructions, be it shifts, comparisons, or logic, more

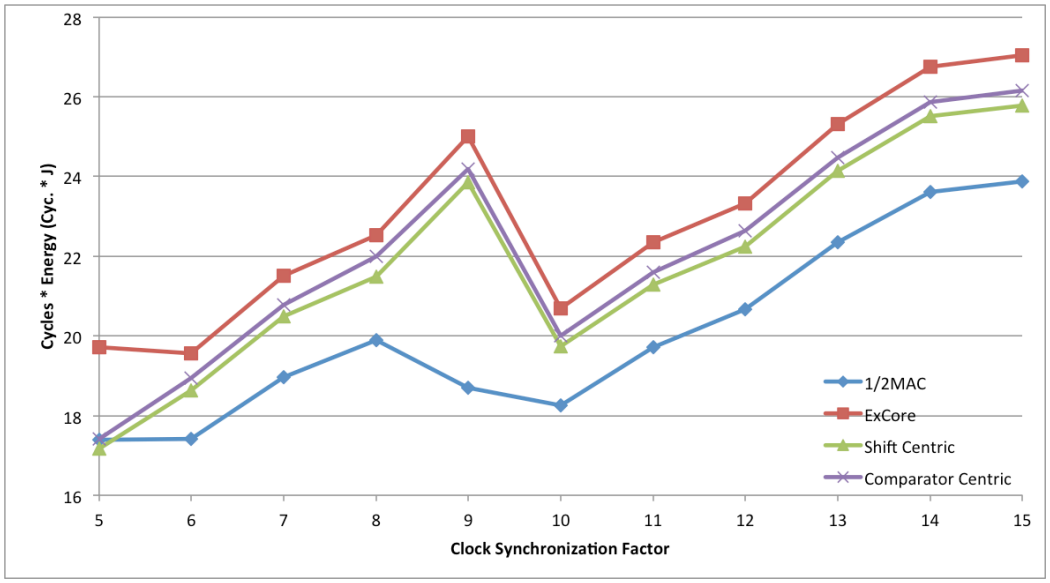


Figure 5.7: Core Cycles × Energy Characterization for Espresso Benchmark Utilizing Module to Module with Local Forwarding

heavily than general-purpose computing applications the ½MAC core should be considered the primary core for further development and optimizations. These conclusions are further supported when examining Figure 5.8, which shows the total energy required versus the total cycles required for execution for all of the benchmarks when using the optimal CSF from Table 4.3 and shows that the ultimate cost of reducing execution cycles by 13.2% is an increase in energy consumption of 38.1% for the 5-stage execution cores.

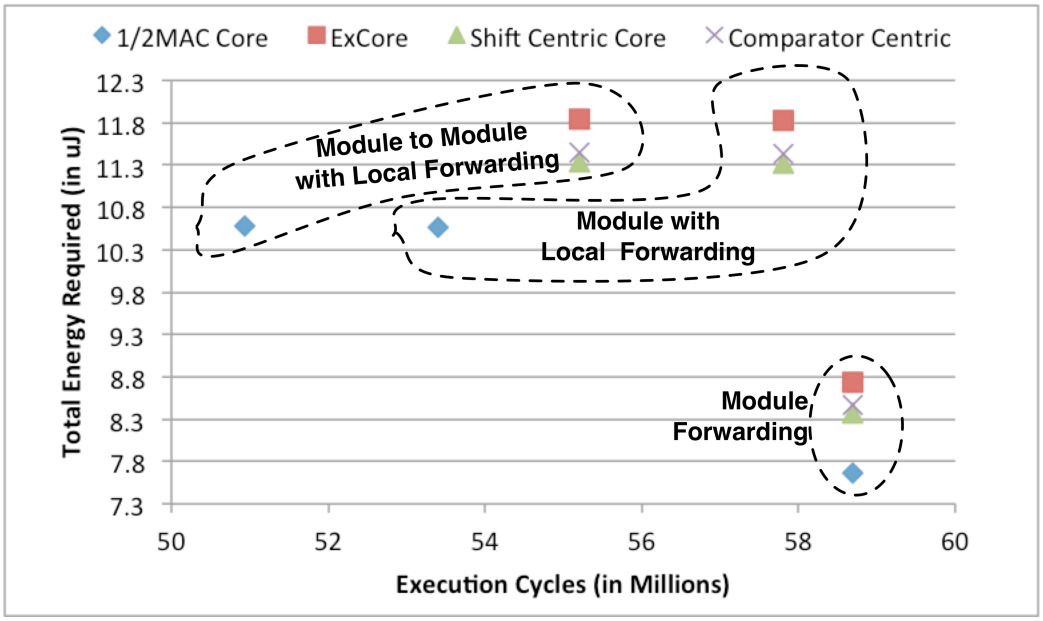


Figure 5.8: Total Energy vs. Total Cycles

Chapter 6

Superscalar Architecture

Having investigated and characterized the designs for a five-stage execution core the focus of this research shifts to investigating alternative architectures for reducing execution cycles. As shown in Chapters 4 and 5 a measurable performance increase can be achieved through better forwarding techniques, but at a steep cost in cell count and substantial energy increase. This factor along with the hardware analysis helps to narrow the focus of this pursuit to architectural changes that better utilize the hardware and provide higher levels of performance. The architectures that have been considered for this initiative are Very Long Instruction Word (VLIW), Vector, and Superscalar paradigms. VLIW processors utilize large instruction sizes that can hold multiple execution unit specific instructions to parallelize processing, but require the code to be compiled for the specific processor configuration. Vector processors are known for processing large amounts of data very rapidly and are very

well suited for applications involving a set operation manipulating an array of data values, but are not an optimal solution for general purpose processing. Superscalar processors focus on maintaining instruction level compatibility with five-stage processors and improving overall performance through dynamic instruction scheduling. Since the goal of this research has been directed toward a MIPS compliant general-purpose processing core the focus of this chapter will be the development of a superscalar execution core [43].

As Smith and Sohi [44] pointed out one of the major benefits to superscalar architectures is that it parallelizes linear code and can eliminate many of the data hazards that the five-stage pipeline introduced. Furthermore the superscalar architecture can allow for such large increases in performance that modern general-purpose processors have stopped using the cycles per instruction (CPI) metric and begun using instructions per cycle (IPC). In this chapter the efforts to adapt the modules from Chapter 2 into a superscalar execution core will be presented. This will commence with an explanation of the new modules that had to be designed to allow for a superscalar configuration, including detailed performance, cell count, and delay characteristics. Next the execution core that has been designed will be introduced and will include a description of its configurability options. An analysis of the new core's performance will be given similar to the evaluation in Chapter 4, focusing on how the different configuration options affect performance. Then an examination of the power and energy consumption characteristics will be offered similar to Chapter 5. Lastly

this evaluation will be used to provide a comparison to the five-stage execution cores for gauging the cost vs. performance increase that the superscalar core should offer.

6.1 Additional Components

In the design of the superscalar core it is necessary to identify the new components that need to be designed to allow for its implementation. One benefit to dynamic instruction execution is that it helps to eliminate stalls in hardware through out-of-order execution (OOE) processing and register renaming. Both of these methods require specialized hardware to be utilized, which include a reorder buffer for OOE and a set of reservation stations for tracking and forwarding renamed registers and their data. In this section the method for implementing these two critical components will be detailed. Each description will start with an explanation of the functionality that is required from the module and be followed by an overview of the new modules design.

6.1.1 Reorder Buffer

A critical component that needs to be designed is the reorder buffer (RoB). The purpose of the RoB is simply to insure instructions are committed in the proper order to avoid hazards and maintain precise interruptions. The ability to buffer instruction commitments also is what allows for OOE to be possible. This component is actually relatively simple in structure though because it is a queue that monitors the oldest instruction issued and commits that instruction once it has been completed.

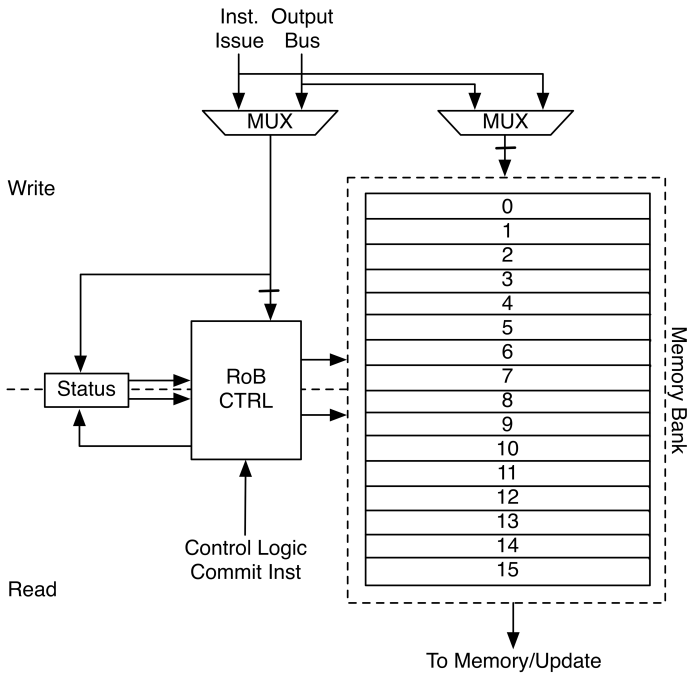


Figure 6.1: Reorder Buffer Structure

When implementing the reorder buffer the key requirements are the ability to commit a value and read a value, additionally the ability of the module to also know that status of each instruction is useful because it would reduce the complexity of the control logic interacting with the module. The final structure of the module is shown in Figure 6.1, and is divided into two halves, one for reading and one for writing with a memory bank for storing the computed values. As is shown in Figure 6.1, two muxes on the write inputs allow for the RoB to be initialized for new instructions, or to commit the computed results from the data bus. To commit an instruction's results back to memory the RoB receives a signal including the index of the oldest instruction and outputs the data through a read from the memory bank. On a cellular level the RoB is quite straightforward to implement because cells can be configured as

a 128x4-bit memory bank, which can be used to implement both the status registers as well as the memory bank of results. The control blocks can then be implemented with a group of cells in math mode that based on the status and the transmitted address generates the proper read/write control signals.

Although the RoB offers a straightforward implementation it is not without its critical design choices. The most critical choice is how many slots to support, which translates to the number of instructions that can be tracked at any given time. For any number less than 16 every slot can be identified in a single 4-bit bundle, larger number will require more 4-bit bundles as well as more cells to complete the comparisons required to determine where the result is destined and if it should be stored. A 16 slot RoB requires 5 cells for status storage and update along with reading from and writing to the memory bank, which requires N/4 cells where N is the Path Width. The RoB currently takes 3 cycles to save the results of an instruction and to output the results of an instruction, although this is pipelined so if needed commits can occur with a latency of 1 cycle.

6.1.2 Reservation Stations

The other module that needs to be designed to accommodate a superscalar core is the reservation station (RS), which utilizes register renaming to maximize the use of the execution modules and facilitate dynamic instruction scheduling. RSs store the type of operation to be executed by the functional unit, operand values or RoB

pointers, and RoB entry where the instruction is stored. RSs must be able to initialize a slot for a new instruction, update its status, store newly computed source data and issue the operation when ready data is in the RS entry. The RS is essentially made up of a series of memory components that hold the operation to be computed including: the renamed location of the register that is the target for source data, source data, a register to keep track of the validity of each memory unit and whether the instruction is ready to be processed. This fits very well with the target hardware, which is essentially all memory, and allows for a cell to be used to store multiple RS's source data.

The general structure of the reservation station is shown in Figure 6.2. Each reservation station slot has its own source address, op code, ID, and status registers to minimize overall delay, but the data memory is shared between all of the execution unit's reservation stations. The current structure allows for issuing instructions to be committed at one time utilizing delays in the communication tree, however when data is being forwarded the data must be held for the number of cycles that match the number of reservation stations. This is because the reservation station is constantly checking if the data on the bus matches one of the reservation stations renamed dependencies and commits the data if so. This process is very similar to polling and helps to reduce the complexity and delay of the RS design.

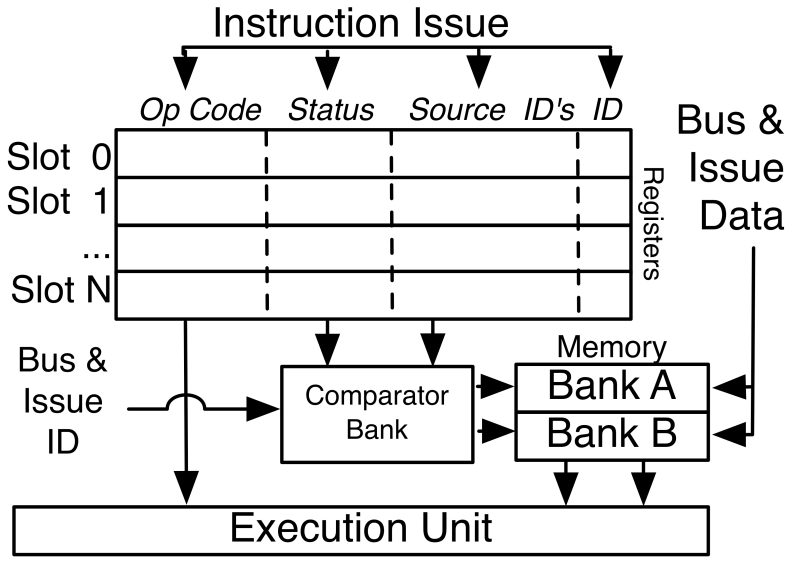


Figure 6.2: Reservation Station Structure

Figure 6.3 provides a cellular description of the RSs and how a 2 slot RS would be structured. The Q_j and Q_k slots are the renamed registers, which indicate where the source data is coming from if a dependency is present. The values stored in the Q register are then compared to the current ID for the data on the incoming data bus in the Valid blocks, which are essentially comparators, and if the values match and the instruction status is waiting for data a write signal is sent to the Memory count filters. The memory count filters are essentially continuous counters that if the input matched to a given counter value is enabled the filter enables a write to occur to the memory bank, which are named V_j and V_k. For RSs with more than two slots multiple tiers of memory count filters can be used as shown in Figure 6.4 for a 4 slot RS.

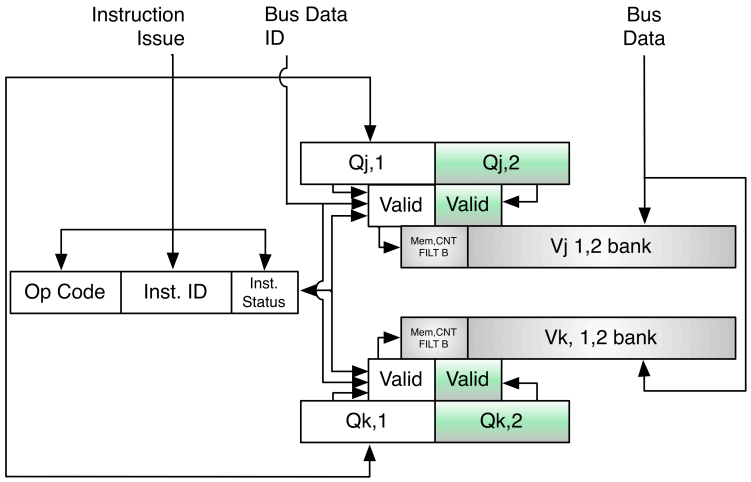


Figure 6.3: Two Slot Reservation Station Cellular Layout

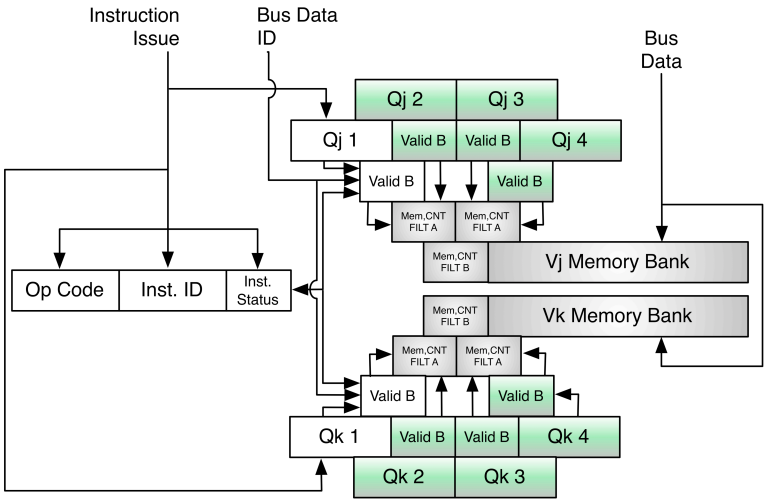


Figure 6.4: Four Slot Reservation Station Cellular Layout

Like the execution modules the reservation station unit can support data widths of any multiple of 4. For a 32-bit data path with 2 RS slots the implementation requires 34 cells and 2 cycles to update data, similarly for 4 RS slots the cell count is increased to 54 cells with 4 cycles to update the data memory. An 8 location RS

similarly requires 8 cycles to update memory, but due to the increasing number of comparators requires 94 cells to implement.

6.2 Superscalar Core

Having designed the RSs and the RoB the next step in this pursuit is to design the actual core itself. Superscalar cores typically follow one of two paths, the first being to utilize multiple ALUs and/or floating point modules so that any operation that may be computed by any available and compatible module like in [45]. The other approach which is used in this study is to separate the execution modules or functional units (FUs) from each other so that their use can be maximized and have a set of reservation stations for each FU. This approach also allows for multiple FUs of the same type to be included without wasting cells for other unneeded duplicate FUs.

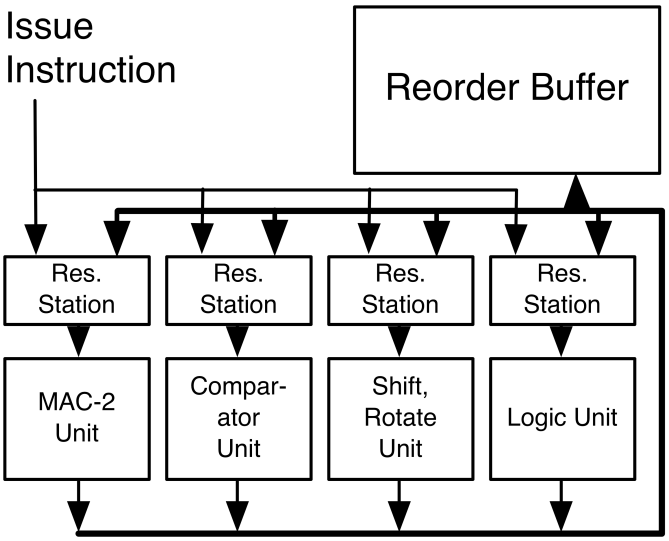


Figure 6.5: Superscalar Core

The basic structure of the core is shown in Figure 6.5. Each execution module or FU is allotted its own set of RSs, allowing FUs to execute operations simultaneously. Furthermore, the shared bus allows for minimal hardware to be used in forwarding. The only drawback is that the data must be held constant for the number of cycles identical to the number of RS slots to allow for every slot to be updated. Similarly to the five-stage cores proposed in [14] this core will work for all path widths that are a multiple of 4 and scales in a reasonable manner for the number of cells required. A 32-bit core that uses 2-slot RSs and a 16 location RoB requires 374 cells, while a 4-slot RS and 16 location RoB configuration requires 484 cells.

6.3 Performance Evaluation

Once the basic structure of the core had been determined the next step is the evaluation of a superscalar core; this evaluation involves different configurations, similar to those in Chapter 4 for the five-stage cores. This means that the configuration of both the RSs and RoBs are varied to determine how RS and RoB size affect the number of execution cycles and cell count. In addition to these standard metrics RoB utilization will also be examined, because it dictates the number of instructions that can be in the process of being executed at a given time, but also has implications on cell count because RoBs larger than 16 entries will require two 4-bit data bundles to address the instruction and increase the cells required for the comparators needed in the RSs.

To complete this evaluation a program has been written that similar to the five-stage simulator take a MIPS trace file and calculates the number of cycles required to execute all the instructions in a benchmark trace file. Some changes were made to the original approach to more accurately calculate the total execution cycles. The first change is that a floating-point unit was added to the core based on designs from [3] for the 32 (.00083%) floating point instructions in the 5 benchmarks. The other change made was the addition of a yet to be designed division module that requires 320 cycles to compute (identical to the delay penalty of the five-stage cores), this then allows the other modules to keep processing data but causes a delay when the division instruction becomes the oldest instruction in the RoB. It should be noted that when making cell count comparisons the size of the floating-point and Division units are not included because they are not present in the five-stage models and would increase the required number of cells for all cores by equal amounts.

Before the results can be detailed the range of configurations must be determined for both RS and RoB sizes. The key factor when coming up with configurations for testing is that the size of the RoB directly affects the size of the RSs' instruction ID, renamed register pointers, and comparator components. To minimize the impact on the RS modules and ensure that only 1 cycle was needed to read or write from the RoB memory bank the range of RoB size was chosen to be 16, 32, and 64 slots. Similarly the RS configurations were chosen to include 2, 4, and 8 instruction slots, which minimize the time a modules output has to be transmitted over the

output bus. The first evaluation is the number of execution cycles required to execute the SPEC integer benchmarks. This is followed by an analysis of the cell counts and resource utilization and how they are related to the number of execution cycles required.

The required number of cycles for execution of the varying core configurations is shown in Table 6.1. The base configuration has RS = 2 and RoB = 16; larger RS and RoB configurations are compared to this one.

It can be observed that the performance of the core actually decreases for RS sizes of 8, ranging from a 7.08% to 9.56% decrease in performance. This is due to the time and output is required to be held on the output bus. Conversely when RS slots are increased to four this provides a measurable performance boost, averaging a speed up of 1.06. While the increase in performance of the configurations with 4 RS slots is

TABLE 6.1: Superscalar Configuration Evaluation

	Superscalar Configuration							
	RS = 2, RoB = 16		RS = 4, RoB = 16		RS = 4, RoB = 32		RS = 4, RoB = 64	
Benchmark	Total Cycles	Speedup	Total Cycles	Speedup	Total Cycles	Speedup	Total Cycles	Speedup
Compress	5,683,490	1	5,383,975	1.056	5,382,054	1.056	5,381,959	1.056
EqnToTT	5,928,251	1	5,786,252	1.025	5,779,396	1.026	5,777,131	1.026
Espresso	5,092,658	1	4,792,109	1.063	4,659,238	1.093	4,658,571	1.093
Li	5,272,303	1	5,090,213	1.036	4,940,909	1.067	4,884,149	1.079
SC	5,203,138	1	4,839,317	1.075	4,777,612	1.089	4,745,700	1.096
All	27,179,840	1	25,891,866	1.050	25,539,209	1.064	25,447,510	1.068
Savings	-	-	1,287,974	0.050	1,640,631	0.064	1,732,330	0.068
%	0		0.0474		0.0604		0.0637	
	RS = 8, RoB = 16		RS = 8, RoB = 32		RS = 8, RoB = 64			
	Total Cycles	Speedup	Total Cycles	Speedup	Total Cycles	% used		
Compress	5,992,270	0.948	5,963,620	0.953	5,963,553	0.953		
EqnToTT	6,573,501	0.902	6,360,647	0.932	6,344,539	0.934		
Espresso	5,849,828	0.871	5,741,785	0.887	5,741,041	0.887		
Li	5,686,983	0.927	5,562,222	0.948	5,490,046	0.960		
SC	5,674,351	0.917	5,602,805	0.929	5,566,173	0.935		
All	29,776,933	0.913	29,231,079	0.930	29,105,352	0.934		
Savings	-2,597,093	-0.087	-2,051,239	-0.070	-1,925,512	-0.066		
%	-0.0956		-0.0755		-0.0708			

not a huge increase it leads to the question of which configuration best utilizes the hardware.

To determine the configuration that best utilizes the hardware an analysis of the usage of the RoB had to be undertaken. To determine how the RoB was used the simulator was modified to monitor how many slots were reserved in the RoB and then report the average and maximum usage. The results are shown in Table 6.2 and Figure 6.6, which show that RoB utilization is highest when large RSs are paired with the smaller RoBs. The reason behind this is that most of the benchmark instruction relies on the MAC-2 unit so the more space there is to issue instructions to that unit the more the RoB will be used. For RoB sizes of 32 and 64 the best utilization occurs for RS sizes of 8, but the cellular increase for RSs of 8 slots versus 4 is 240 cells.

Overall when examining Table 6.2 it can be observed that the highest average use is obtained with RS sizes of 8, but come with a large overhead to the cell count. The increased RoB utilization does not necessarily mean better performance as shown in Figure 6.7. An optimal core when it comes to performance and utilization is when the RoB and RSs contain 16 and 4 slots, respectively. The next best core is actually the 16 slot RoB and 2 slot RS since the cells required is much smaller when compared to other cores.

TABLE 6.2: Reorder Buffer Usage and Core Area

RoB Size	RS slots	Area	% Area Increase	RoB Average Use	% RoB Utilization	Maximum RoB Slots Used
16	2	372	-	6.27	39.21%	16
16	4	452	21.51%	10.08	63.01%	16
16	8	612	64.52%	10.37	64.80%	16
32	4	532	43.01%	11.39	35.59%	32
32	8	772	107.53%	17.91	55.98%	32
64	4	532	43.01%	11.64	18.19%	48
64	8	772	107.53%	18.75	29.29%	51

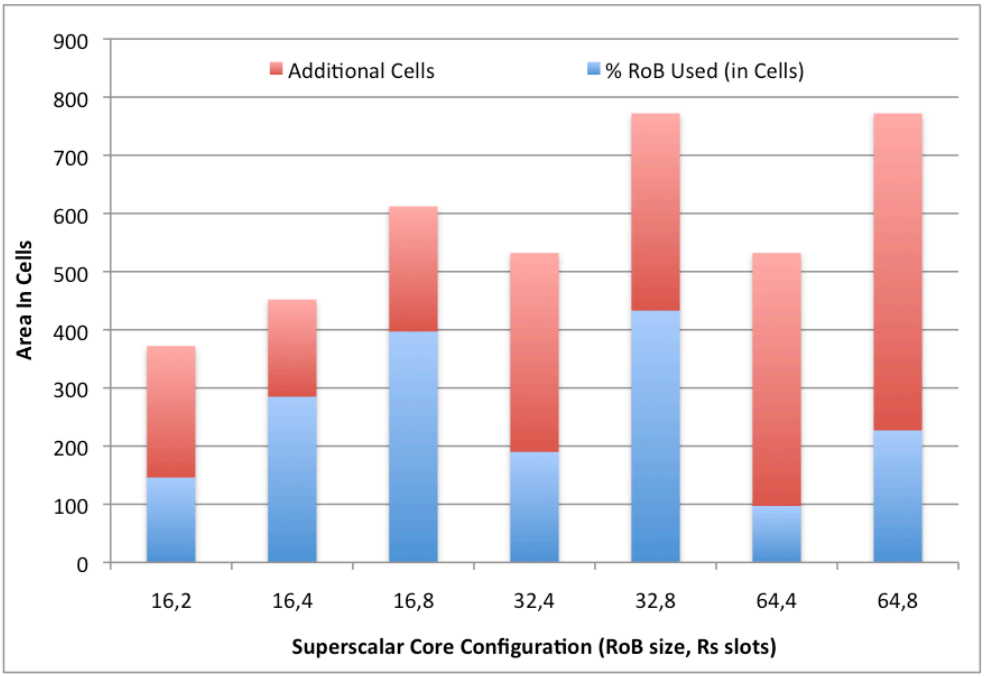


Figure 6.6: Reorder Buffer Average Usage by Area Required

6.4 Power and Energy Evaluation

The next aspect of the superscalar core and its various configurations that has been examined is the energy consumption. Similar to the work done in Chapter 5 the approach followed to characterize the power consumed in a superscalar core was a hybrid approach first focused on characterizing the modular power consumption and

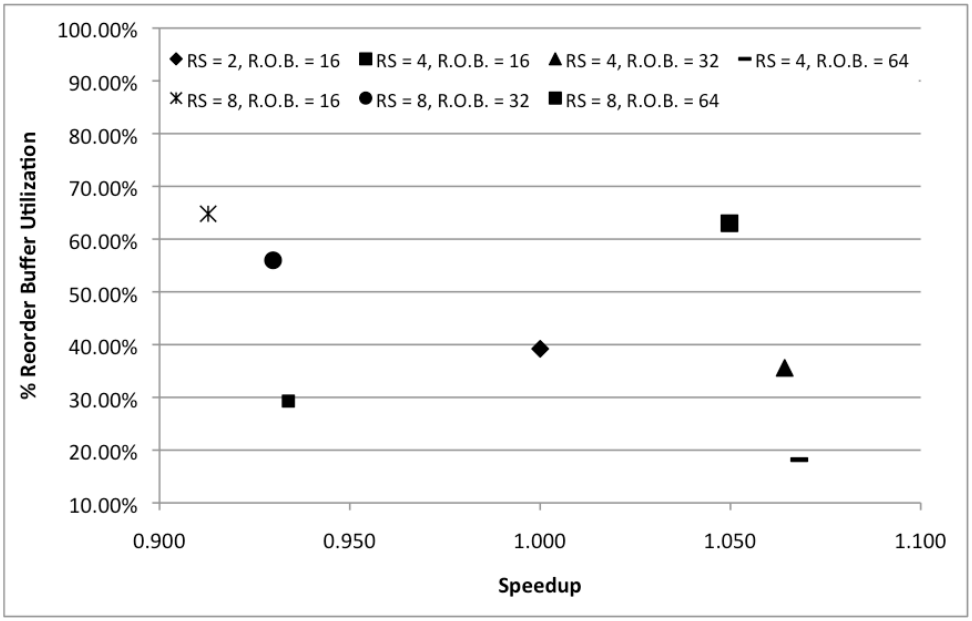


Figure 6.7: Reorder Buffer Usage vs. Speedup

then on how the varying benchmarks would consume power based on instruction type. Using the power required for each module would then allow for the energy consumed when executing a benchmark to be calculated.

To determine the appropriate energy consumption rates for the reservation station and reorder buffer blocks, pessimistic estimates of switching activity were made that would equate to comparators requiring the same energy as the worst performing logic and comparator blocks from Section 5.2. This was done to account for the continual processing of random data as both modules ensure the up to date statuses of each instruction. The results of this characterization is detailed in Table 6.3 and Figure 6.8 and show that energy required for operation increases more rapidly as the RS size increases (62%), than as when the RoB increases (54%), when transitioning from RS sizes of 4 to 8.

TABLE 6.3: Cycle and Energy Requirements for Varying Reservation Station and Reorder Buffer Configurations

RS, RoB	Update Cycles	Energy per Cycle (fJ)	Static Energy (fJ)
2,16	2	286.330	0.725
4,16	4	447.913	1.145
8,16	8	771.080	1.985
4,32	4	706.613	1.764
8,32	8	1238.420	3.109
4,64	4	706.613	1.764
8,64	8	1238.420	3.109

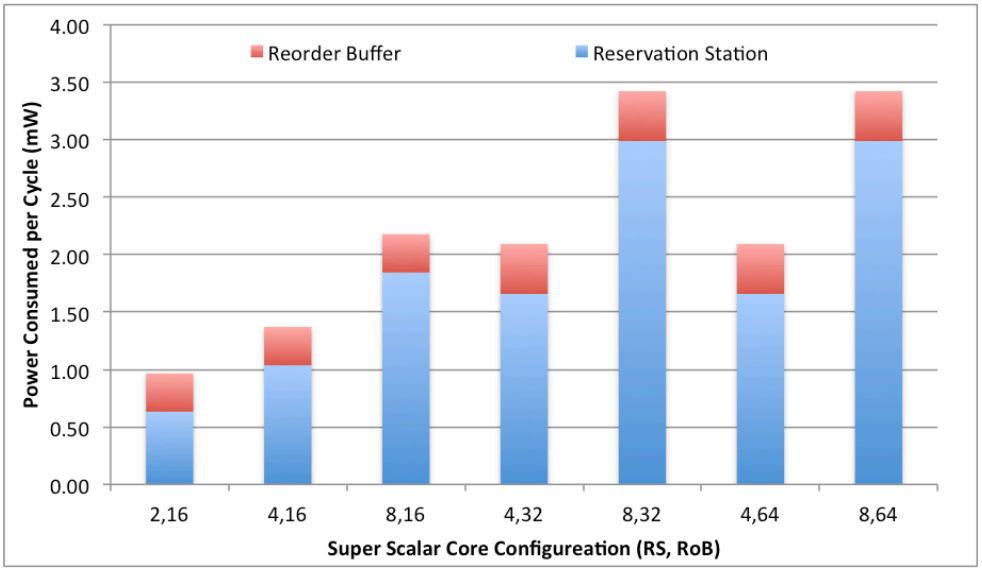


Figure 6.8: Power Requirements for Varying Reservation Station and Reorder Buffer Configurations

Using these energy consumption rates and the execution modules (functional units) consumption rates from section 5.2 the energy to execute any of the previously described benchmark trace files could be calculated. The result of this evaluation is shown in Figure 6.9 and Table 6.4 for the execution of all of the benchmarks or roughly 3.85 million instructions. The results demonstrate that RS modules of 2 or 4

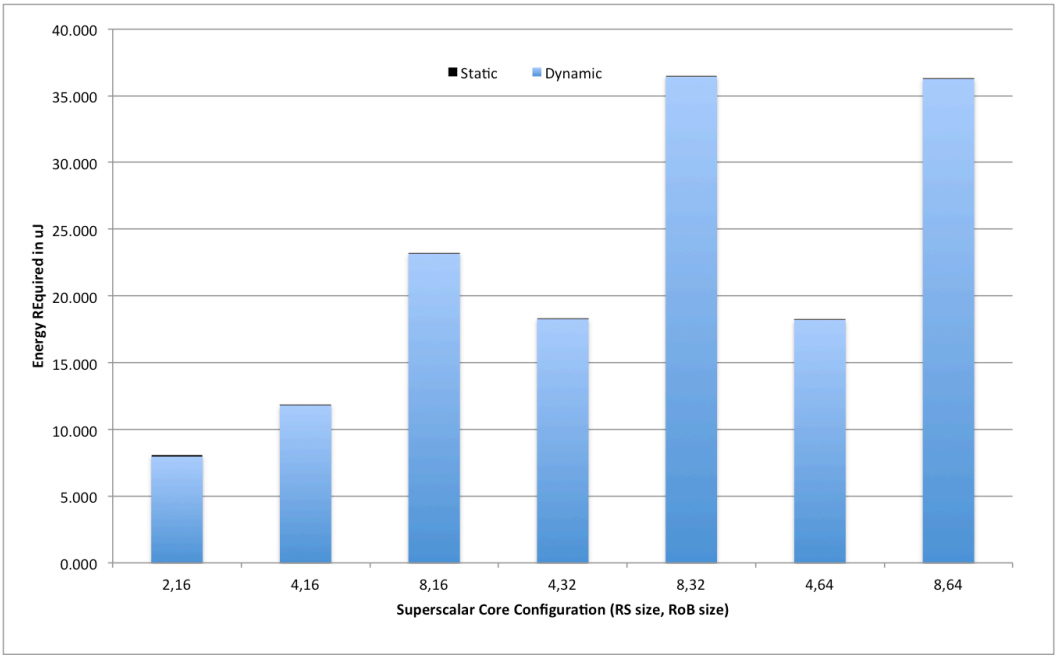


Figure 6.9: Energy Required for Executing 3.85 Million Instructions

TABLE 6.4: Superscalar Energy and Cycle Requirements

RS Size	RoB Size	Execution Cycles	% Cycle Decrease	Dynamic Energy (uJ)	Static Energy (nJ)	Total Energy (uJ)	% Energy Increase
2	16	2.72E+07	-	7.984	63.695	8.048	-
4	16	2.59E+07	*4.74%*	11.799	60.677	11.860	*47.36%*
8	16	2.98E+07	*-9.56%*	23.162	69.781	23.232	*188.67%*
4	32	2.55E+07	*6.04%*	18.248	59.850	18.308	*127.48%*
8	32	2.92E+07	*-7.55%*	36.402	68.502	36.471	*353.16%*
4	64	2.54E+07	*6.37%*	18.184	59.635	18.243	*126.68%*
8	64	2.91E+07	*-7.08%*	36.247	68.207	36.315	*351.22%*

slots offer much better energy consumption characteristics than with 8 slots and that the RoB utilizing 16 locations. More specifically an increase in RS size from 2 to 4 offers a 4.74% improvement in execution time while increasing the energy by 47%. Similarly as the RoB increases in size the energy consumption more than doubles while realizing increasingly smaller improvements in performance.

6.5 Comparison with Five-Stage Cores

Having investigated how the varying configurations of the superscalar core affect its performance, overall hardware utilization and energy consumption this section focuses on how the superscalar and five-stage cores compare. To do this the most appealing cores from each approach will be used, including the superscalar core with a 16 slot RoB and RS sizes of 2, and 4. The five-stage cores that have been chosen for comparison are the ½MAC core and ExCore, with the majority of the focus being on the ½MAC core utilizing the MwLF scheme. The metrics that will be examined are: cell count requirements, CPI and energy consumed for execution. The reason why CPI is used here instead of the execution cycles is that the nop instructions were eliminated from the superscalar core's benchmark trace files because the majority of the nops in the trace files were used to fill branch delay slots, which in the current configuration do not serve a purpose.

The CPI for each of the cores selected for comparison is shown in Table 6.5. As is shown the five-stage cores best case CPI is 13.24, while the optimal configuration that minimizes the number of cells is 13.88 for the ½MAC and 15.03 for the ExCore. Comparatively the optimal superscalar core configurations provide a near two times increase in performance yielding a CPI of 8.1 and 7.72 for configurations with a RS size of 2 and 4 respectively and a 16 location RoB. This comparison shows that if performance is critical then the only acceptable core should be a superscalar core.

TABLE 6.5: Five-Stage and Superscalar Core Comparisons

Scheme	Area	% Inc.	CPI	Speedup	Speedup / Cell Inc.
ExCore - MwLF	287	-	15.029	-	-
1/2MAC - MwLF	287	0.00%	13.881	1.083	-
1/2MAC - M2MwLF	335	16.72%	13.240	1.135	0.0236
Superscalar RS - 2, ROB - 16	374	30.31%	8.104	1.855	0.0213
Superscalar RS - 4, ROB - 16	484	68.64%	7.720	1.947	0.0099

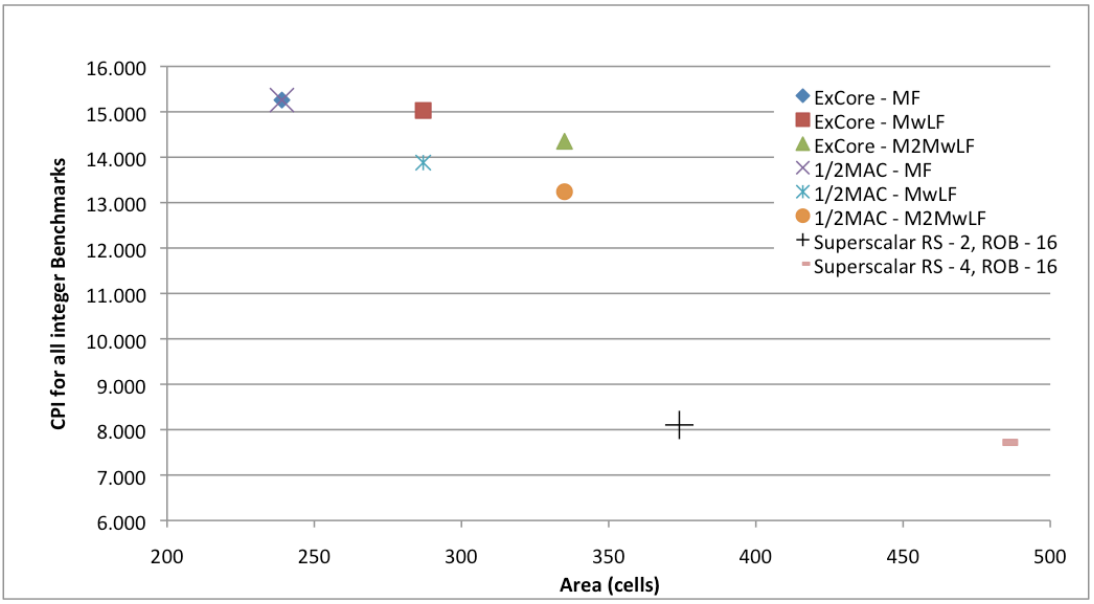


Figure 6.10: Superscalar and Five-Stage CPI vs Area

When examining the size of the varying cores there is a very noticeable overhead for the superscalar architecture that far surpasses the overhead for the more advance M2MwLF scheme. The M2MwLF scheme requires that 48 more cells be added to the optimal five-stage core, while the superscalar core requires at least 87 more cells and that increases to 197 cells for the most optimal superscalar configuration. When evaluating the cores solely by cells required, the five-stage cores may be optimized while still maintaining a significantly smaller footprint than the

superscalar cores and provide a reasonable level of performance for a small number of cells required.

The next comparison that must be examined is the energy consumed by each core during execution. Figure 6.11 shows how the total energy consumed versus the CPI are related. The figure shows that although the superscalar cores consume a comparable amount of energy to execute the same benchmarks, 22% less than average for RSs of 2 and 14% more for RSs of 4, they provide a much higher level of performance by reducing the CPI by an average 45% compared to the five-stage cores. The figure also includes a data point for the superscalar core configured for a RS size of 4 with 32 RoB slots, and shows that the power increase versus performance

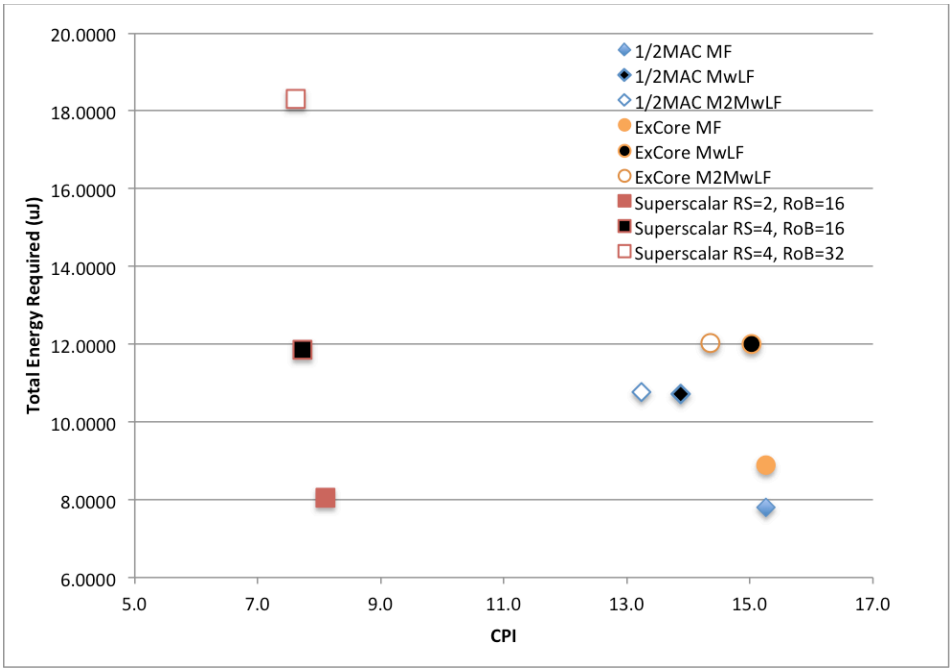


Figure 6.11: Superscalar and Five-Stage Energy vs. CPI

increase is not a reasonable tradeoff, which would also eliminate all configurations with 8 RSs.

Lastly, by using the execution cores' energy characterizations and their total execution cycles, along with the clock cycle time of .2 ns for 21 nm FinFETs from Table 4.4, a calculation of the energy–delay product may be computed. The results of this computation are shown in Figure 6.12 for the 3.85 million instructions contained in the benchmarks. When examining the figure it becomes apparent that although the superscalar cores use more energy on average than the five-stage cores, the energy is used much more effectively to reduce the overall delay. This can be seen by

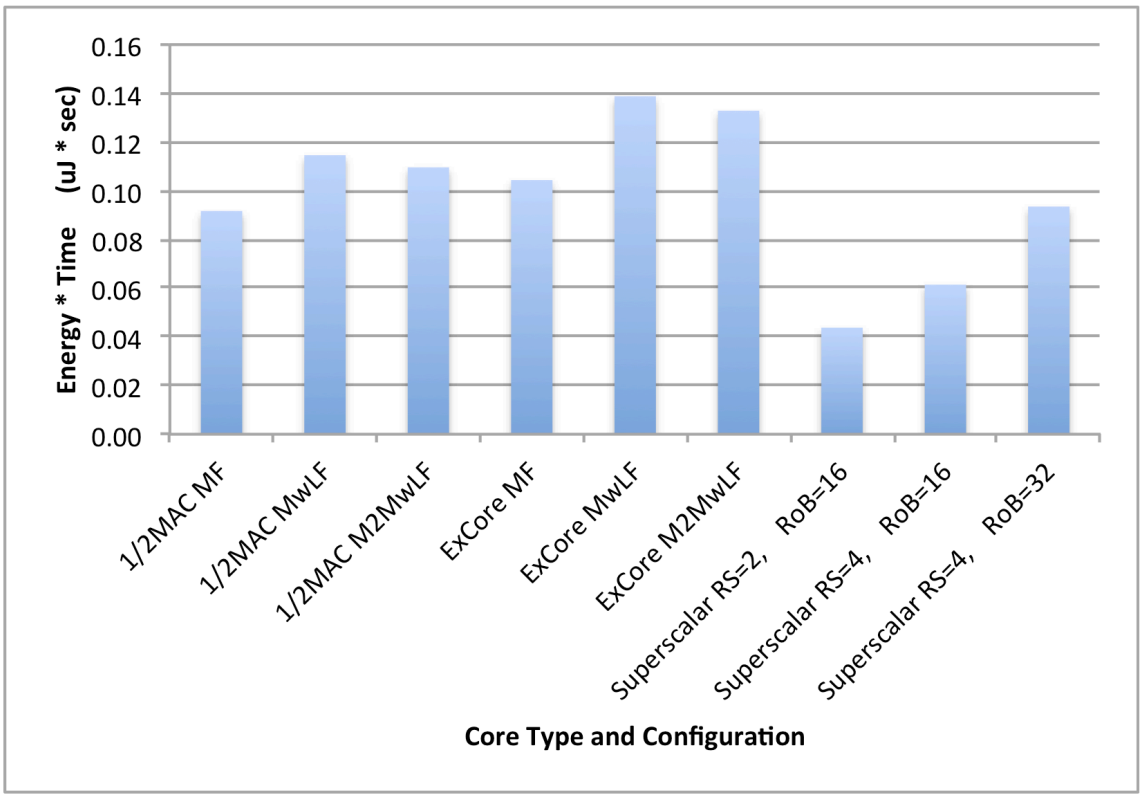


Figure 6.12: Superscalar and Five-Stage Energy-Delay Product

comparing the cores with the lowest energy-delay product for each architecture, the 2 slot RS and 16 location RoB superscalar core's product is 0.044 compared to the ½MAC core utilizing MF's product of 0.092 even thought it requires 0.24 μ J more energy.

This study provides a design space where designers of reconfigurable processor cores can choose a core that fits the computational, power, and cell count requirements. If a designer is looking for a good balance of area, performance, and power, Figures 6.10 and 6.11 can be used to find this core. The figures show the average CPI versus both the required cells and energy consumed for all of the ½MAC and ExCore configurations, as well as the superscalar cores utilizing 2 and 4 RS sizes with a 16 location RoB. Using these figures along with Figure 6.12 it can be observed that the best processor to balance performance, cell count, and energy is the Superscalar core with 2 RS slots and a 16 location RoB, because it offers a speedup of 1.71 compared to the optimal ½MAC core, while offering a performance level within 6.4% of the best performing superscalar configurations. This core only requires 30% more area (that is 87 additional cells) than the optimal five-stage cores, while using 23% (110) fewer cells than the best superscalar core. Lastly, the selected core consumes only 3% more energy than the execution core with the lowest energy requirements and requires just 8.05 μ J for 3.85 million instructions to be computed.

Chapter 7

Concluding Remarks

This dissertation has focused on implementing an execution core for a novel medium-grain reconfigurable hardware. The research began seeking to characterize FPGA power consumption for the AES encryption algorithm when used for wireless sensor networks, but found the power demands too onerous for use. As a result the study transitioned to focus on the homogenous medium-grain reconfigurable hardware, which offers a promising architecture for more than just the DSP applications it was designed and includes: integrated pipelining, rapid technology adoptions, faster clock speeds, and flexible four input, two output cells.

Throughout the research the development of autonomous modular execution units have been detailed, which are not only scalable but provide all the functionality needed to ensure MIPS compatibility. These units have served as the foundation upon which the research has been built to characterize how the target hardware could be

best utilized for the implementation of a general-purpose processing core, and how to balance the performance, energy, and cellular requirements of the cores. The remainder of this chapter will discuss the contributions of this research and propose future directions for continuing this research.

7.1 Contributions

This research has offered many insights into how to harness the target hardware through the building of MIPS compatible execution cores that include:

- **Modular Design:** Modular execution units, termed the shift, comparator, logic, and MAC-2 modules, have been designed to provide the needed operations for a MIPS ISA compliant processor. These execution units can extend the instruction set to include operations such as logical NOR, NAND and XNOR, as well as left and right rotate. By utilizing these modular and autonomous designs the components may even be used in superscalar and vector processors that requires multiple cores for a given instruction type without having to duplicate the other unneeded units.
- **Four Novel Execution Cores:** Four executions cores have been proposed, three of which, ½MAC, Shift Centric, and Comparator Centric, are optimized for a specific instruction type, while the fourth, the ExCore, has been designed to provide a consistent level of performance to all of the instructions. These

cores provide a building block that can serve as the core component of a standard five-stage pipeline, multi-cycle or single cycle processor.

- **Five-Stage Architecture:** In addition to detailing the organization of the execution stage a method for adapting the five-stage architecture to the target hardware has been detailed for use with the execution cores. This includes the ability to dynamically resize the CSF window size to eliminate wasted cycles and allow for optimized forwarding between execution modules.
- **Highly Interchangeable cores and forwarding schemes:** Using the four execution cores and three forwarding schemes an in-depth analysis of the performance characteristics could be carried out including how different forwarding mechanisms affect the size and performance of a core when executing different SPEC benchmarks. This study shows that 11.55% of the total cycles could be saved at a 20% increase in cell count or that a 15.69% cycle savings at a 40% cell count increase could be achieved. This also provided means to investigate how partitioning the clock through the control synchronization factor would affect the cores in terms of minimizing the number clock cycles required for completing a program execution and eliminating wasted cycles. As shown in Table 3, a CSF increase from 10 to 11 for some cores resulted in a savings of almost 10.3 million cycles or a reduction in execution time of roughly 15.72%.

- **Scalability:** A 32-bit processor has been studied in-depth. However, using the modular nature of the medium-grain reconfigurable hardware cores and the parallel nature of many operations, the proposed designs can be easily extended to accommodate any data path width that is a multiple of 4. For example, a 16-bit core requires 89 cells while a 32-bit core requires 239 cells; even so the difference in total delay changes by less than a factor of 2.
- **High system clock frequency:** One of the most appealing aspects of the target hardware is the ease with which it can move from one technology to another. This leads to steady increases in performance without the need for significant design modification, and is shown in that the ½MAC can run at speeds of 2GHz on 45 nm CMOS technology, but when the same configuration is used with FinFET technology [35] clock speeds in excess of 5 GHz can be conservatively realized.
- **Comparable or Better Performance than existing soft-processors:** The proposed designs (using similar CMOS technology) offer comparable levels of performance with soft processors offered by Xilinx and Altera while out pacing other custom designed MIPS processors for FPGAs when using similar implementation technologies.
- **Hardware Evaluation:** An evaluation of the hardware's cellular architecture has also been completed. This showed that as the number of elements in a cell is increased the execution time for complex operations

would see a small decrease, but this resulted in a sharp increase in unused elements within a cell, which would lead to less than optimal power consumption characteristics.

- **Power Characterization:** Using a hierarchical model, that combines switching activities and data from ongoing research in transitioning the hardware to FinFET technology, a characterization of how the five-stage cores would consume power was developed. The model showed that as the execution modules scaled to service larger path widths at worst the cell count increased by 2.5×, while the best-case modules scaled at the same rate as the path width. It was also found that to utilize the MwLF or M2MwLF schemes with a five-stage execution core an increase in energy of 36% is required over the MF scheme.
- **Superscalar Execution:** An investigation into the ability of the execution modules and hardware to be used for a superscalar-processing core was completed. The results show that with the addition of the configurable reservation station and reorder buffer modules the CPI of a five-stage core can almost be halved while minimizing the cell count increase to 30.3% for a superscalar core. This investigation also demonstrated that pairing a reservation station utilizing 2 or 4 slots and a reorder buffer of 16 slots provided the highest levels of speedup while minimizing the cellular increase and maximizing hardware utilization.

Using the contributions of this research the core of any general-purpose processor, the execution core, may be designed to best fit the parameters required. This can take into account the number of cells available, maximum power consumption, performance characteristics, and processor type. This work also helps to increase the number and diversity of application that take advantage of the target hardware.

7.2 Future Directions

To continue this work there are a number of areas that should be investigated. The most obvious of which is the implementation of the control logic and the stages that surround the execution core. This includes memory controllers to connect to external devices and multi-level cache architectures that can take advantage of the target hardware to provide an optimal balance of size and delay.

It is also important that the model used to characterize energy consumption be refined. This should include using the most up-to-date numbers from the current FinFET and CNFET research. Once the cell decoder is implemented it will also be important to refine how the energy is calculated since energy characterizations may become less pessimistic as more of the hardware is modeled and accurately described.

Lastly, the execution modules should be examined and evaluated for use in a vector-processing unit that could become a co-processor. As previously mentioned the architecture of vector processors is particularly well suited to take full advantage

of the target hardware and its built-in pipeline, but is not well suited for general-purpose processors. Uses for this research may be applied to high throughput encryption/decryption modules, SIMD co-processors for multimedia applications, and matrix mathematics.

Bibliography

[1] M.J. Myjak, and J.G. Delgado-Frias, "Medium-Grain Cells for Reconfigurable DSP Hardware," in *IEEE Transactions on Circuits and Systems, I: Fundamental Theory and Applications*, vol. 54, no 6, June 2007, pp. 1255-1265

[2] M.J. Myjak, et al, "H-tree interconnection structure for reconfigurable DSP hardware," *in Proceedings International Conference of Engineering Reconfigurable Systems Algorithms (ERSA)*, Las Vegas, NV, Jun. 2004, pp. 170–176.

[3] M.J.Myjak, J.G. Delgado-Frias, "A Medium-Grain Reconfigurable Architecture for DSP: VLSI Design, Benchmark Mapping, and Performance, Very Large Scale Integration (VLSI) Systems," *IEEE Transactions on VSLI Systems,* vol.16, no.1, pp.14-23, Jan. 2008

[4] B.S. Doyle et al, "High Performance Fully-Depleted Tri-Gate CMOS Transistors," in *IEEE Electron Device Letters*, Volume 24, Issue 4, April 2003, pp. 263 - 265

[5] J.G. Delgado-Frias, Z. Zhang, M.A. Turi, "Low power SRAM cell design for FinFET and CNTFET technologies," 2010 International *Green Computing Conference,* pp.547-553, 15-18 Aug. 2010

[6] National Institute of Standards and Technology, *Advanced Encryption Standard, FIPS 197*, November 2001.

[7] J. Van Dyken, and J.G. Delgado-Frias, "FPGA schemes for minimizing the power-throughput trade-off in executing the Advanced Encryption Standard algorithm." in *Journal of Systems Architecture*, 2-3 February 2010, pp. 56

[8] H. Li, Z. Friggstad, "An efficient architecture for the AES mix columns operation," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'05)*, 2005, pp. V 4637–V 4640.

[9] D. Patterson, and J. Hennessy, "Computer Organization and Design, The Hardware/Software Interface 4th ed.", Elsevier Morgan Kaufmann, 2009

[10] J. Van Dyken, and J.G. Delgado-Frias, "A medium-grain reconfigurable processing unit," in *2010 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 1-4 Aug. 2010, pp. 729-732

[11] M.J. Myjak, and J.G. Delgado-Frias, "Pipelined multipliers for reconfigurable hardware," in *Parallel and Distributed Processing Symposium,* 2004. Proceedings. 18th International, 26-30 April 2004, pp. 150-155

[12] ARM Ltd. www.arm.com

[13] H.C. Neto, and M.P. Vestias, "Architectural Tradeoffs in the Design of Barrel Shifters for Reconfigurable Computing," in 2008 4th Southern Conference on *Programmable Logic*, 26-28 March 2008, pp. 31-36

[14] J. Van Dyken, and J.G. Delgado-Frias, "A medium-grain reconfigurable processor organization," in *2011 54th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 7-10 Aug. 2011

[15] J. Van Dyken, et al, "FPGA Schemes with Optimized Routing for the Advanced Encryption Standard," in *ERSA'2008 The International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, USA, July 14 - 17, 2008

[16] K. Gaj, and P. Chodowiec, "Comparison of the hardware performance of the AES candidates using reconfigurable hardware", in *Third Advanced Encryption Standard (AES3) Candidate Conference*, New York, 2000.

[17] P. Chodowiec, and K. Gaj, "Very compact FPGA implementation of the AES algorithm," in *Proceedings of the Fifth International Workshop on Cryptographic Hardware and Embedded Systems – CHES 2003*, Cologne Germany, September 2003, LNCS 2779, Springer, 2003, pp. 319–333.

[18] A. Dandalis, et al, "A comparative study of performance of AES final candidates using FPGAs," in *Cryptographic Hardware and Embedded Systems Workshop (CHES 2000)*, Worcester, Massachusetts, 2000.

[19] National Institute of Standards and Technology (NIST), 2001, AES, <csrc.nist.gov/archive/aes/index.html>, accessed 8.9.11.

[20] H. Satyanarayana, AES128, OpenCores, Online, Available from Internet, <opencores.com/projects.cgi/web/aes_crypto_core/overview>, 2004, accessed 8.9.11.

[21] Chi-Jeng Chang, et al, "8-bit AES FPGA Implementation using Block RAM," in *Industrial Electronics Society, 2007. IECON 2007*. 33rd Annual Conference of the IEEE, pp.2654-2659, 5-8 Nov. 2007

[22] R. Razdan, M.D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on*, 30 Nov. 2 Dec. 1994, pp. 172- 180

[23] SPEC, http://www.spec.org/cpu92/

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

[25] Altera Corp. http://www.altera.com.

[26] I. Gonzalez, et al, "Self-Reconfigurable Embedded Systems on Low-Cost FPGAs," in IEEE *Micro*, vol.27, no.4, July-Aug. 2007, pp. 49-57

[27] R. Lysecky, F. Vahid, "A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning," in *Proceedings of the conference on Design, Automation and Test in Europe*, 2005, *DATE 2005*, Vol. 1, 7-11 March 2005, pp. 18- 23

[28] Z. Kokosinski, B. Malus, "FPGA Implementations of a Parallel Associative Processor with Multi-Comparand Multi-Search Operations," in *International Symposium on Parallel and Distributed Computing, 2008. ISPDC '08*, 1-5 July 2008, pp. 444-448

[29] R. A. L. Tyson, et al, "A pipelined double-issue MIPS based processor architecture," in *International Symposium on Intelligent Signal Processing and Communication Systems, ISPACS 2009*, 7-9 Jan. 2009, pp. 583-586

[30] X. Li, T. Li, "ECOMIPS: an economic MIPS CPU design on FPGA," in *4th IEEE International Workshop on System-on-Chip for Real- Time Applications*, 19-21 July 2004, pp. 291-294

[31] T. Ramdas, et al, "FPGA implementation of an integer MIPS processor in Handel-C and its application to human face detection," *TENCON 2004. 2004 IEEE Region 10 Conference*, Vol. 1, 21-24 Nov. 2004, pp. 36- 39

[32] P. Gautham, et al, "Low-power pipelined MIPS processor design," *12th International Symposium on Integrated Circuits*, ISIC '09, 14-16 Dec. 2009, pp. 462-465

[33] L. Shang, et al, “Dynamic power consumption in Virtex-II FPGA family,” in Proceedings of *ACM International Symposium on FPGAs*, Feb. 2002, pp. 157–164

[34] S.E. Esmaeili, N.I. Khachab, “Efficiency Of Components' Region-Constrained Placement To Reduce FPGA's Dynamic Power Consumption,” in *14th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2007*. 11-14 December, 2007, pp. 857-860

[35] M.A. Turi, and J.G. Delgado-Frias, “Performance-Power Tradeoffs of 8T FinFET SRAM Cells,” in 54th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 1-4, 7-10 Aug. 2011

[36] Intel, Corp. http://newsroom.intel.com/community/intel_newsroom/blog/2011/05/04/intel-reinvents-transistors-using-new-3-d-structure

[37] P.M. Munson, J.G. Delgado-Frias, “A performance-power evaluation of FinFET flip-flops under process variations”, in *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pp.1-4, 7-10 Aug. 2011

[38] M. Pedram, “Design Technologies for Low Power VLSI.” in: *Encyclopedia of Computer Science and Technology*, Vol. 36. Marcel Dekker, Inc., (1997) 73–96

[39] P. Van Oostende, et al, “Estimation of typical power of synchronous (CMOS) circuits using a hierarchy of simulators,” in *IEEE Journal of Solid-State Circuits 28*, 1 (Jan.), 26 –39, 1993

[40] B.J. George, et al, “Power analysis for semicustom design.” in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 249–252.1994.

[41] Xilinx XPower. http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm

[42] K. Arshak, et al, “Power Testing of an FPGA based System Using Modelsim Code Coverage capability,” in IEEE Workshop on *Design and Diagnostics of Electronic Circuits and Systems*, 2007. DDECS '07, vol., no., pp.1-4, 11-13 April 2007

[43] J.L. Hennessy , David Patterson. “Computer Architecture: A Quantitative Approach (Fourth Edition).” Morgan Kaufmann. 2006.

[44] J.E. Smith, G.S. Sohi, “The microarchitecture of superscalar processors,” *Proceedings of the IEEE,* vol.83, no.12, pp.1609-1624, Dec 1995

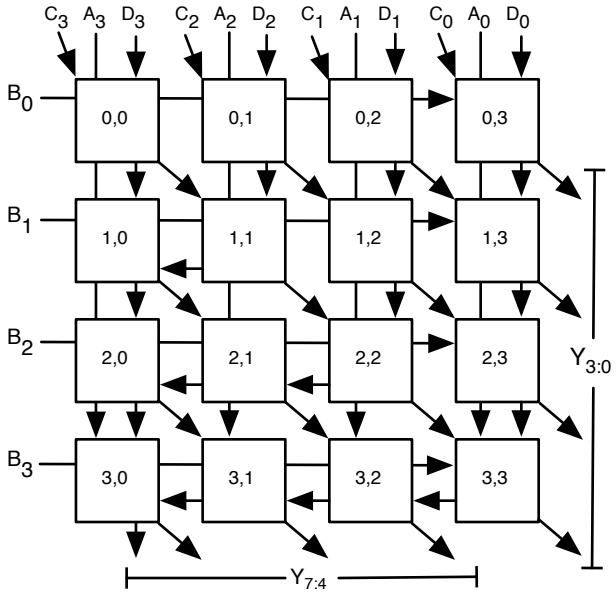
[45] R.M. Tomasulo. “An efficient algorithm for exploiting multiple arithmetic units.” In *IBM Journal of Research and Development*, 11(1):25–33, 1967.

Appendix A

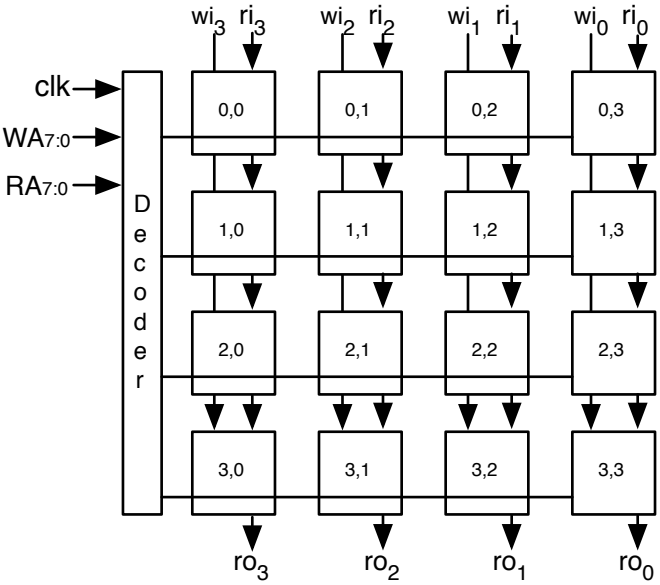
Cellular Configurations

In this appendix the cellular configurations used to implement the modules presented in chapter 2 will be detailed. Modules that require more than a single cell being used in parallel will first have the module's structure detailed with a diagram detailing the cellular organization. Each cell configuration will include a diagram detailing what elements are used for computation and how data flows through the cell, along with a written description of the cell's function and input/output usage and assignment. Each cell configuration will also include a table detailing the elemental configurations indexed by row from the top left to bottom right, which includes an explanation of the element's use, and memory configuration.

Cell Elemental Naming Convention

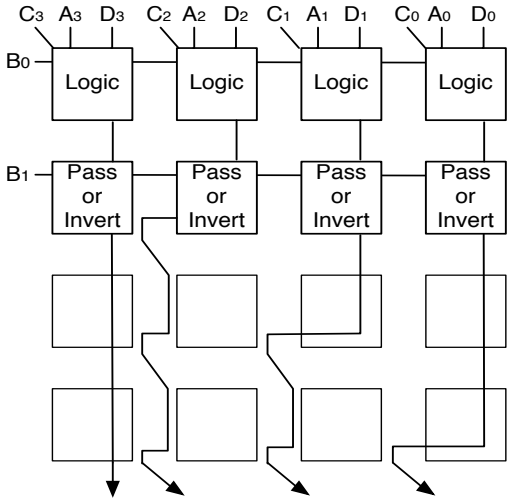


MATH CELL



MEMORY CELL

Bitwise Logic Cell



Description

This cell computes the bitwise logic functions of: AND, NAND, OR, NOR, XOR, XNOR, and NOT.

Ctrl		I	Operation	
A_n	B_0	I	B_1=0	B_1=1
0	0	I	AND	NAND
0	1	I	OR	NOR
1	0	I	NOT	*Pass*
1	1	I	XOR	XNOR

Inputs

$A_{3:0}$ - Control Signal

$B_{1:0}$ - Control Signals

$C_{3:0}$ - Input 1

$D_{3:0}$ - Input 2

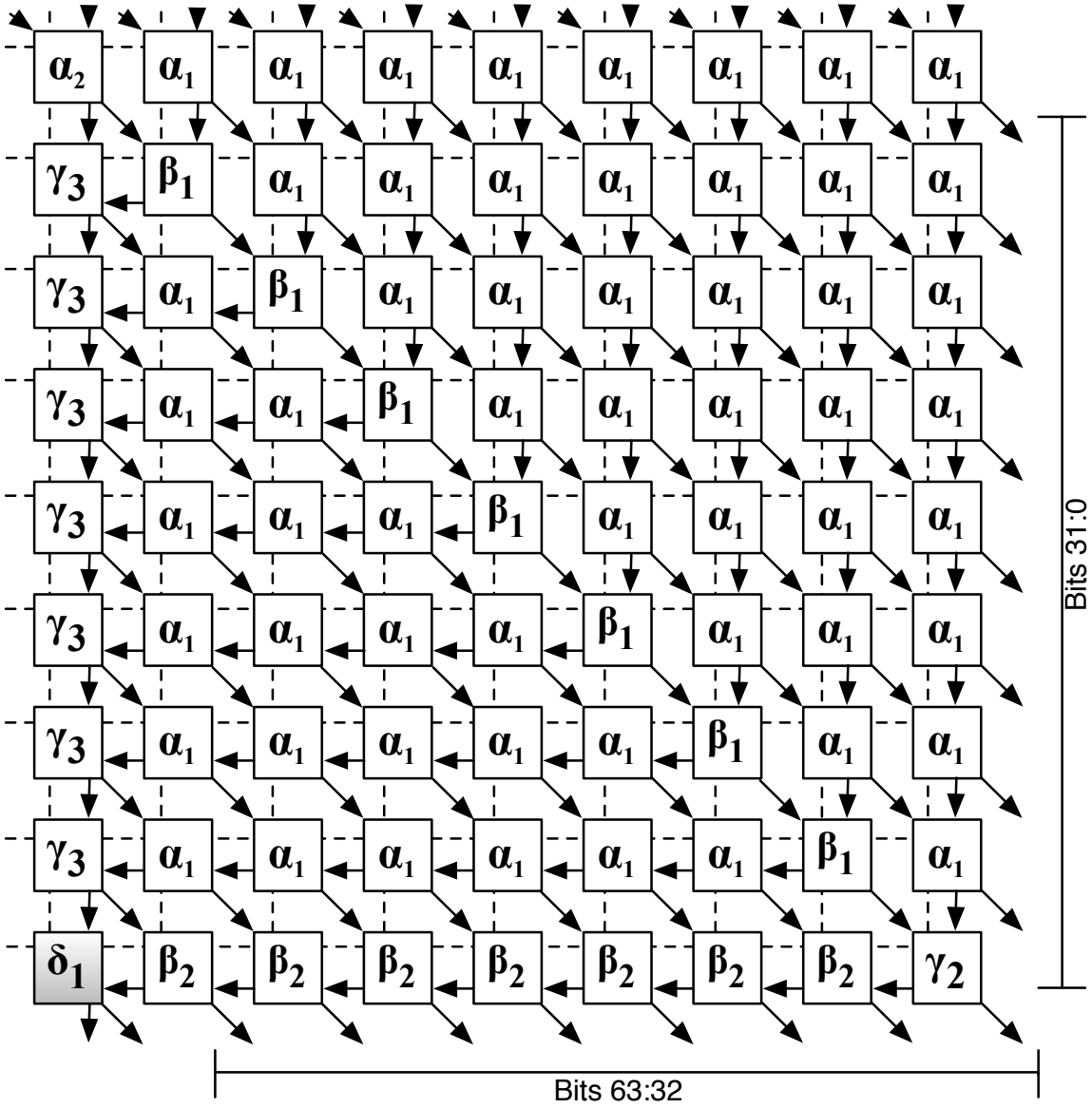
Outputs

$Y_{3:0}$ - unused

$Y_{7:4}$ - Result

Element ID	Description or Function	Y Config	Z Config
0,0	Bitwise Block	0000	63E8
0,1	Bitwise Block	0000	63E8
0,2	Bitwise Block	0000	63E8
0,3	Bitwise Block	0000	63E8
1,0	Pass or Invert	AAAA	3C3C
1,1	Pass or Invert	0000	5A5A
1,2	Pass or Invert	0000	5A5A
1,3	Pass or Invert	0000	5A5A
2,0	Route	AAAA	CCCC
2,1	Route	AAAA	CCCC
2,2	Route	0000	AAAA
2,3	Route	0000	AAAA
3,0	Route	AAAA	CCCC
3,1	Route	AAAA	CCCC
3,2	Route	AAAA	CCCC
3,3	Route	0000	AAAA

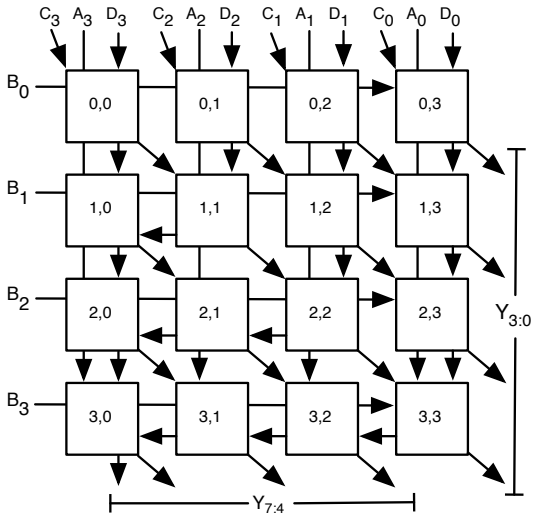
MAC-2 Module



With sign extension or zero padding this unit computes a 32-bit Multiply and accumulate function of the form: A*B+C+D

The Delta cell in the lower left corner is not needed

Alpha 1 Cell



Description

This Cell is used to build signed or unsigned MAC units. If the unit is unsigned all cells are alpha cells. The functional output is:

$(A*B)+C+D \rightarrow Y_{7:4} = +, Y_{3:0} = +$

Inputs

$A_{3:0}$ - Multiplicand

$B_{3:0}$ - Multiplicand

$C_{3:0}$ - Addend

$D_{3:0}$ - Addend

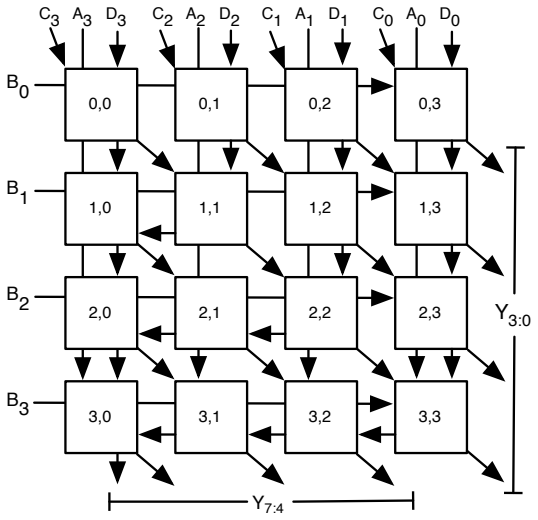
Outputs

$Y_{3:0}$ - 4 LSB of result

$Y_{7:4}$ - 4 MSB of result

Element ID	Description or Function	Y Config	Z Config
0,0	Alpha Element	9666	E888
0,1	Alpha Element	9666	E888
0,2	Alpha Element	9666	E888
0,3	Alpha Element	9666	E888
1,0	Alpha Element	9666	E888
1,1	Alpha Element	9666	E888
1,2	Alpha Element	9666	E888
1,3	Alpha Element	9666	E888
2,0	Alpha Element	9666	E888
2,1	Alpha Element	9666	E888
2,2	Alpha Element	9666	E888
2,3	Alpha Element	9666	E888
3,0	Alpha Element	9666	E888
3,1	Alpha Element	9666	E888
3,2	Alpha Element	9666	E888
3,3	Alpha Element	9666	E888

Alpha 2 Cell



Description

This Cell is used to build signed MAC units. The functional output is:

(-A*B)-C-D -> $Y_{7:4}$ = -, $Y_{3:0}$ = -

Inputs

$A_{3:0}$ - Multiplicand

$B_{3:0}$ - Multiplicand

$C_{3:0}$ - Addend

$D_{3:0}$ - Addend

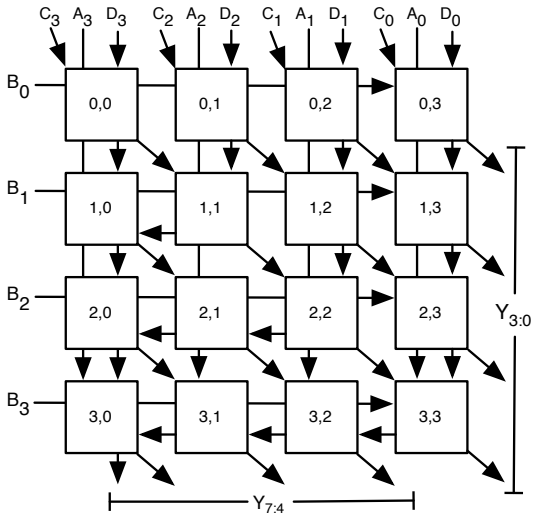
Outputs

$Y_{3:0}$ - 4 LSB of result

$Y_{7:4}$ - 4 MSB of result

Element ID	Description or Function	Y Config	Z Config
0,0	Alpha Element	9666	E888
0,1	Alpha Element	9666	E888
0,2	Alpha Element	9666	E888
0,3	Alpha Element	9666	E888
1,0	Gamma Element	9666	D444
1,1	Beta Element	9666	B222
1,2	Alpha Element	9666	E888
1,3	Alpha Element	9666	E888
2,0	Gamma Element	9666	D444
2,1	Alpha Element	9666	E888
2,2	Beta Element	9666	B222
2,3	Alpha Element	9666	E888
3,0	Gamma Element	9666	D444
3,1	Alpha Element	9666	E888
3,2	Alpha Element	9666	E888
3,3	Beta Element	9666	B222

Beta 1 Cell



Description

This Cell is used to build signed MAC units. The functional output is:

(A*B)-C+D -> $Y_{7:4}$ = +, $Y_{3:0}$ = -

Inputs

$A_{3:0}$ - Multiplicand

$B_{3:0}$ - Multiplicand

$C_{3:0}$ - Addend

$D_{3:0}$ - Addend

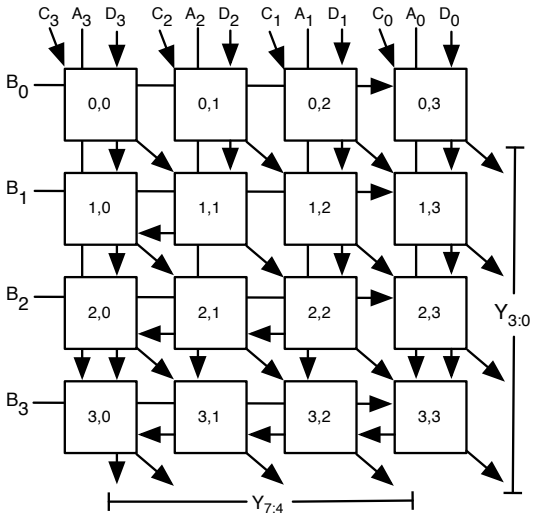
Outputs

$Y_{3:0}$ - 4 LSB of result

$Y_{7:4}$ - 4 MSB of result

Element ID	Description or Function	Y Config	Z Config
0,0	Beta Element	9666	B222
0,1	Alpha Element	9666	E888
0,2	Alpha Element	9666	E888
0,3	Alpha Element	9666	E888
1,0	Alpha Element	9666	E888
1,1	Beta Element	9666	B222
1,2	Alpha Element	9666	E888
1,3	Alpha Element	9666	E888
2,0	Alpha Element	9666	E888
2,1	Alpha Element	9666	E888
2,2	Beta Element	9666	B222
2,3	Alpha Element	9666	E888
3,0	Alpha Element	9666	E888
3,1	Alpha Element	9666	E888
3,2	Alpha Element	9666	E888
3,3	Beta Element	9666	B222

Beta 2 Cell



Description

This Cell is used to build signed MAC units. The functional output is:

(A*-B)+C-D -> $Y_{7:4}$ = -, $Y_{3:0}$ = +

Inputs

$A_{3:0}$ - Multiplicand

$B_{3:0}$ - Multiplicand

$C_{3:0}$ - Addend

$D_{3:0}$ - Addend

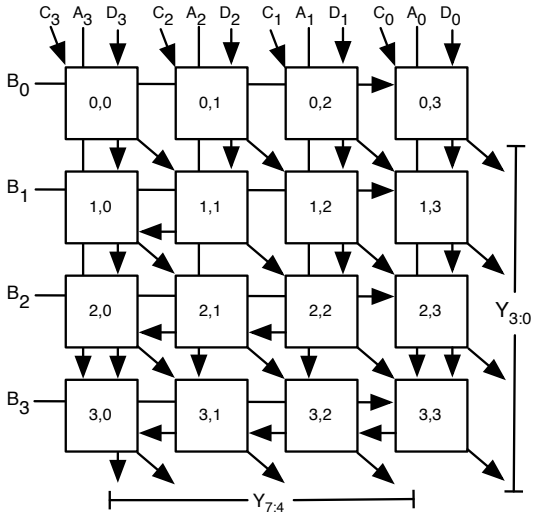
Outputs

$Y_{3:0}$ - 4 LSB of result

$Y_{7:4}$ - 4 MSB of result

Element ID	Description or Function	Y Config	Z Config
0,0	Gamma Element	9666	D444
0,1	Alpha Element	9666	E888
0,2	Alpha Element	9666	E888
0,3	Alpha Element	9666	E888
1,0	Alpha Element	9666	E888
1,1	Beta Element	9666	B222
1,2	Alpha Element	9666	E888
1,3	Alpha Element	9666	E888
2,0	Alpha Element	9666	E888
2,1	Alpha Element	9666	E888
2,2	Beta Element	9666	B222
2,3	Alpha Element	9666	E888
3,0	Beta Element	9666	B222
3,1	Beta Element	9666	B222
3,2	Beta Element	9666	B222
3,3	Gamma Element	9666	D444

Gamma 2 Cell



Description

This Cell is used to build signed MAC units. The functional output is:

(A*-B)-C+D -> $Y_{7:4}$ = -, $Y_{3:0}$ = +

Inputs

$A_{3:0}$ - Multiplicand

$B_{3:0}$ - Multiplicand

$C_{3:0}$ - Addend

$D_{3:0}$ - Addend

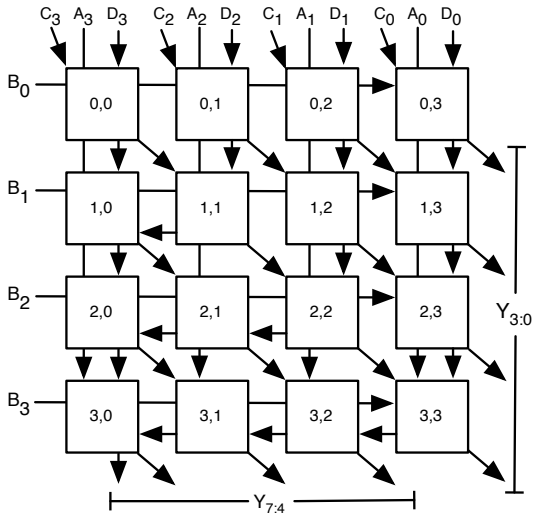
Outputs

$Y_{3:0}$ - 4 LSB of result

$Y_{7:4}$ - 4 MSB of result

Element ID	Description or Function	Y Config	Z Config
0,0	Beta Element	9666	B222
0,1	Alpha Element	9666	E888
0,2	Alpha Element	9666	E888
0,3	Alpha Element	9666	E888
1,0	Alpha Element	9666	E888
1,1	Beta Element	9666	B222
1,2	Alpha Element	9666	E888
1,3	Alpha Element	9666	E888
2,0	Alpha Element	9666	E888
2,1	Alpha Element	9666	E888
2,2	Beta Element	9666	B222
2,3	Alpha Element	9666	E888
3,0	Beta Element	9666	B222
3,1	Beta Element	9666	B222
3,2	Beta Element	9666	B222
3,3	Gamma Element	9666	D444

Gamma 3 Cell



Description

This Cell is used to build signed MAC units. The functional output is:

(-A*B)-C+D -> $Y_{7:4}$ = -, $Y_{3:0}$ = +

Inputs

$A_{3:0}$ - Multiplicand

$B_{3:0}$ - Multiplicand

$C_{3:0}$ - Addend

$D_{3:0}$ - Addend

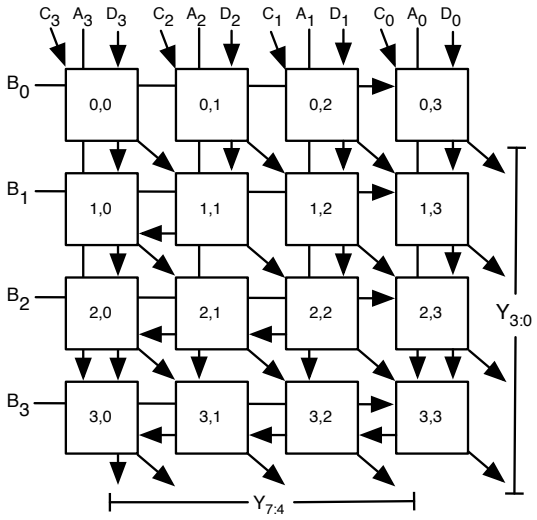
Outputs

$Y_{3:0}$ - 4 LSB of result

$Y_{7:4}$ - 4 MSB of result

Element ID	Description or Function	Y Config	Z Config
0,0	Gamma Element	9666	D444
0,1	Alpha Element	9666	E888
0,2	Alpha Element	9666	E888
0,3	Alpha Element	9666	E888
1,0	Gamma Element	9666	D444
1,1	Alpha Element	9666	E888
1,2	Alpha Element	9666	E888
1,3	Alpha Element	9666	E888
2,0	Gamma Element	9666	D444
2,1	Alpha Element	9666	E888
2,2	Alpha Element	9666	E888
2,3	Alpha Element	9666	E888
3,0	Gamma Element	9666	D444
3,1	Alpha Element	9666	E888
3,2	Alpha Element	9666	E888
3,3	Alpha Element	9666	E888

Delta Cell



Description

This Cell is used to build signed MAC units. The functional output is:

(-A*-B)-C-D -> $Y_{7:4}$ = -, $Y_{3:0}$ = +

Inputs

$A_{3:0}$ - Multiplicand

$B_{3:0}$ - Multiplicand

$C_{3:0}$ - Addend

$D_{3:0}$ - Addend

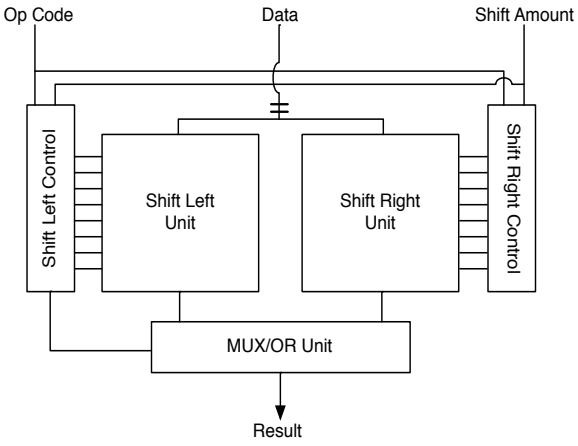
Outputs

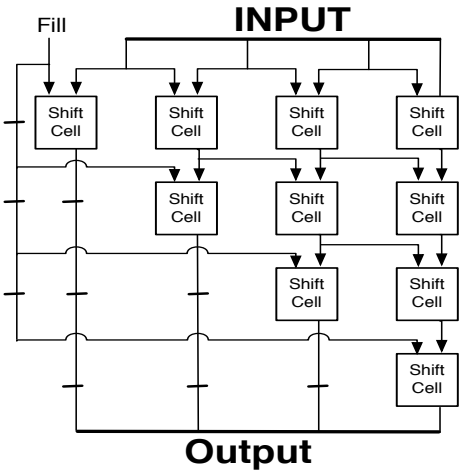
$Y_{3:0}$ - 4 LSB of result

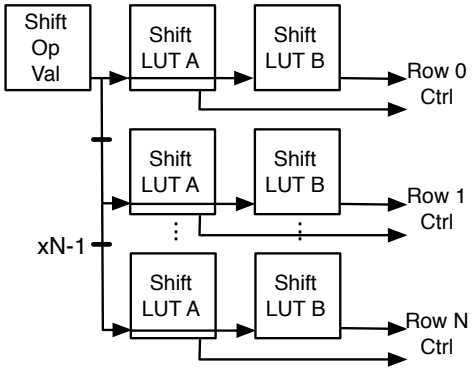
$Y_{7:4}$ - 4 MSB of result

Element ID	Description or Function	Y Config	Z Config
0,0	Alpha Element	9666	E888
0,1	Alpha Element	9666	E888
0,2	Alpha Element	9666	E888
0,3	Alpha Element	9666	E888
1,0	Gamma Element	9666	D444
1,1	Beta Element	9666	B222
1,2	Alpha Element	9666	E888
1,3	Alpha Element	9666	E888
2,0	Gamma Element	9666	D444
2,1	Alpha Element	9666	E888
2,2	Beta Element	9666	B222
2,3	Alpha Element	9666	E888
3,0	Delta Element	9666	8EEE
3,1	Beta Element	9666	B222
3,2	Beta Element	9666	B222
3,3	Gamma Element	9666	D444

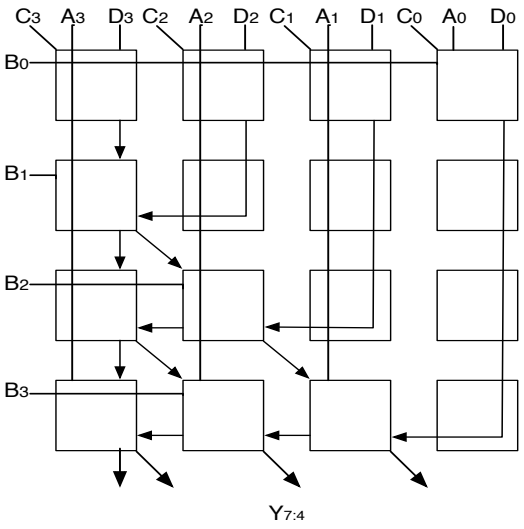
Shift Module







Shift Cell



Description

This Cell implements 4-bit Shifting

SHFT	A	B	Output
0	0000	0000	$\{C_3, C_2, C_1, C_0\}$
1	0001	1000	$\{D_0, C_3, C_2, C_1\}$
2	1111	0000	$\{D_1, D_0, C_3, C_2\}$
3	1111	1111	$\{D_2, D_1, D_0, C_3\}$
4	0000	0001	$\{D_3, D_2, D_1, D_0\}$

Inputs

$A_{3:0}$ - Control

$B_{3:0}$ - Control

$\{D_{0:3}, C_{0:3}\}$ - Data

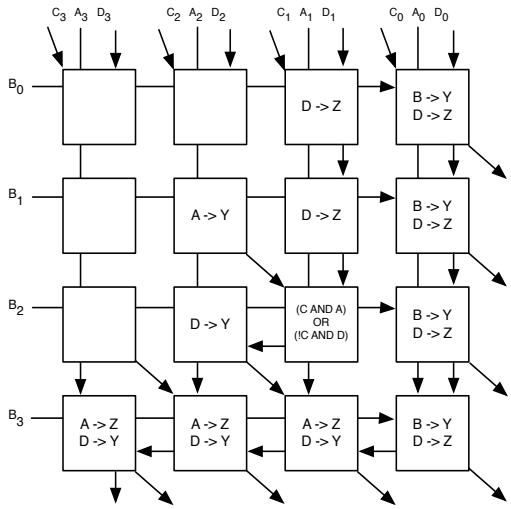
Outputs

$Y_{3:0}$ - unused

$Y_{7:4}$ - Result of Shift

Element ID	Description or Function	Y Config	Z Config
0,0	Bit Choice	0000	CCAC
0,1	Bit Choice	0000	ACAC
0,2	Bit Choice	0000	AAAC
0,3	Bit Choice	0000	AAAC
1,0	Routing	CA0A	AC0C
1,1	Routing	0000	AAAA
1,2	Routing	0000	AAAA
1,3	Routing	0000	AAAA
2,0	Routing	AC0A	CA0C
2,1	Routing	CC0A	AA0C
2,2	Routing	0000	AAAA
2,3	Routing	0000	AAAA
3,0	Routing/Output	AACA	CCAC
3,1	Routing/Output	ACCA	CAAC
3,2	Routing/Output	CCCA	AAAC
3,3	Routing	0000	AAAA

Shift Op Val Cell



Description

This unit take in the Op and Shift Amount signals and generates the LUT control signals for the shift/rotate module.

Inputs

$A_{3:0}$ - Operation Control

- *A_3 - Enable (0/1 - off/on)*
- *A_2 - Rotate Enable (0/1 - off/on)*
- *A_1 - Rotate Dir. (0/1 - right/left)*
- *A_0 - Shift Dir. (0/1 - right/left)*

$B_{3:0}$ - Shift Input Lower 4-bits

$C_{3:0}$ - unused

$D_{3:0}$ - Shift Input High 4-bits

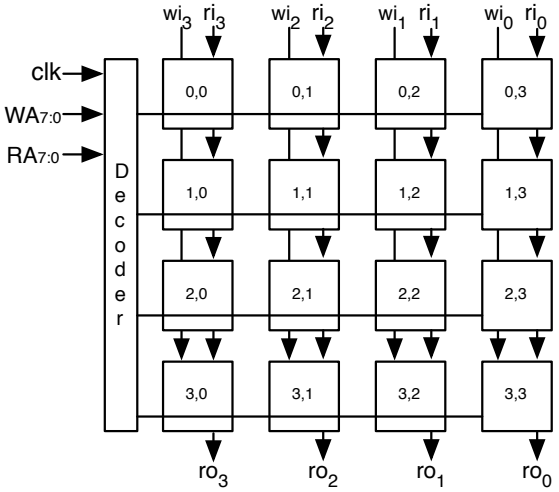
Outputs

$Y_{3:0}$ - $LUT\_Control_{3:0}$

$Y_{7:4}$ - $LUT\_Control_{7:4}$

Element ID	Description or Function	Y Config	Z Config
0,0	unused	0000	0000
0,1	unused	0000	0000
0,2	Data Forward	0000	AAAA
0,3	Ctrl Output/Data Forward	F0F0	AAAA
1,0	unused	0000	0000
1,1	Forward Rotate Enable	FF00	0000
1,2	Data Forward	0000	AAAA
1,3	Ctrl Output/Data Forward	F0F0	AAAA
2,0	unused	0000	0000
2,1	Data Forward	AAAA	0000
2,2	Choose Between Shift or Rotate	0000	EE22
2,3	Ctrl Output/Data Forward	F0F0	AAAA
3,0	Ctrl Output	AAAA	FF00
3,1	Ctrl Output	AAAA	FF00
3,2	Ctrl Output	AAAA	CCCC
3,3	Ctrl Output/Data Forward	F0F0	AAAA

Shift Control LUTs



Function

The LUT configurations serve to generate the control signals for the Shift cell Modules.

Inputs

$WA_{7:0}$ - disabled

$RA_{7:0}$ - Shift Op Val output

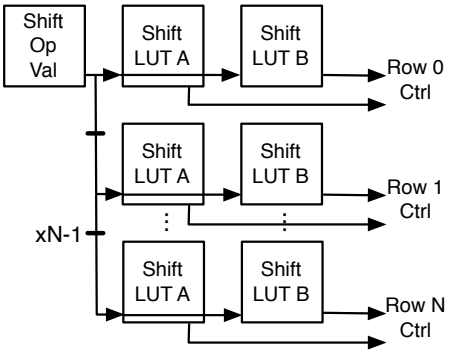
Merge of Shift Amount and Op

$wi_{3:0}$ - unused

$ri_{3:0}$ - unused

Outputs

$ro_{3:0}$ - 1 of 2 Shift Row Control Signals



Shift Control LUTs

Element ID	Left A: Level 0		Left B: Level 0		Right A: Level 0		Right B: Level 0	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	0000	0006	0000	000A	0000	000C	0000	000A
0,1	0000	0006	0000	0002	0000	000C	0000	0008
0,2	0000	0006	0000	0002	0000	000C	0000	0008
0,3	0000	000E	0000	0003	0000	000E	FFFF	FFF8
1,0	0000	0000	0000	0000	0000	0000	0000	0000
1,1	0000	0000	0000	0000	0000	0000	0000	0000
1,2	0000	0000	0000	0000	0000	0000	0000	0000
1,3	0000	0000	0000	0000	0000	0000	0000	0000
2,0	C000	0000	A000	0000	0000	000C	0000	000A
2,1	C000	0000	8000	0000	0000	000C	0000	0008
2,2	C000	0000	8000	0000	0000	000C	0000	0008
2,3	E000	0000	8000	0000	0000	000E	FFFF	FFF8
3,0	0000	0006	0000	000A	6000	0000	A000	0000
3,1	0000	0006	0000	0002	6000	0000	2000	0000
3,2	0000	0006	0000	0002	6000	0000	2000	0000
3,3	0000	000E	0000	0003	E000	0000	3FFF	FFFF

Element ID	Left A: Level 1		Left B: Level 1		Right A: Level 1		Right B: Level 1	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	0000	0060	0000	00A0	0000	00C0	0000	00A0
0,1	0000	0060	0000	0020	0000	00C0	0000	0080
0,2	0000	0060	0000	0020	0000	00C0	0000	0080
0,3	0000	00E0	0000	003F	0000	00E0	FFFF	FF80
1,0	0000	0000	0000	0000	0000	0000	0000	0000
1,1	0000	0000	0000	0000	0000	0000	0000	0000
1,2	0000	0000	0000	0000	0000	0000	0000	0000
1,3	0000	0000	0000	0000	0000	0000	0000	0000
2,0	0C00	0000	0A00	0000	0000	00C0	0000	00A0
2,1	0C00	0000	0800	0000	0000	00C0	0000	0080
2,2	0C00	0000	0800	0000	0000	00C0	0000	0080
2,3	0E00	0000	F800	0000	0000	00E0	FFFF	FF80
3,0	0000	0060	0000	00A0	0600	0000	0A00	0000
3,1	0000	0060	0000	0020	0600	0000	0200	0000
3,2	0000	0060	0000	0020	0600	0000	0200	0000
3,3	0000	00E0	0000	003F	0E00	0000	03FF	FFFF

Element ID	Left A: Level 2		Left B: Level 2		Right A: Level 2		Right B: Level 2	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	0000	0600	0000	0A00	0000	0C00	0000	0A00
0,1	0000	0600	0000	0200	0000	0C00	0000	0800
0,2	0000	0600	0000	0200	0000	0C00	0000	0800
0,3	0000	0E00	0000	03FF	0000	0E00	FFFF	F800
1,0	0000	0000	0000	0000	0000	0000	0000	0000
1,1	0000	0000	0000	0000	0000	0000	0000	0000
1,2	0000	0000	0000	0000	0000	0000	0000	0000
1,3	0000	0000	0000	0000	0000	0000	0000	0000
2,0	00C0	0000	00A0	0000	0000	0C00	0000	0A00
2,1	00C0	0000	0080	0000	0000	0C00	0000	0800
2,2	00C0	0000	0080	0000	0000	0C00	0000	0800
2,3	00E0	0000	FF80	0000	0000	0E00	FFFF	F800
3,0	0000	0600	0000	0A00	0060	0000	00A0	0000
3,1	0000	0600	0000	0200	0060	0000	0020	0000
3,2	0000	0600	0000	0200	0060	0000	0020	0000
3,3	0000	0E00	0000	03FF	00E0	0000	003F	FFFF

Shift Control LUTs

Element ID	Left A: Level 3		Left B: Level 3		Right A: Level 3		Right B: Level 3	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	0000	6000	0000	A000	0000	C000	0000	A000
0,1	0000	6000	0000	2000	0000	C000	0000	8000
0,2	0000	6000	0000	2000	0000	C000	0000	8000
0,3	0000	E000	0000	3FFF	0000	E000	FFFF	8000
1,0	0000	0000	0000	0000	0000	0000	0000	0000
1,1	0000	0000	0000	0000	0000	0000	0000	0000
1,2	0000	0000	0000	0000	0000	0000	0000	0000
1,3	0000	0000	0000	0000	0000	0000	0000	0000
2,0	000C	0000	000A	0000	0000	C000	0000	A000
2,1	000C	0000	0008	0000	0000	C000	0000	8000
2,2	000C	0000	0008	0000	0000	C000	0000	8000
2,3	000E	0000	FFF8	0000	0000	E000	FFFF	8000
3,0	0000	6000	0000	A000	0006	0000	000A	0000
3,1	0000	6000	0000	2000	0006	0000	0002	0000
3,2	0000	6000	0000	2000	0006	0000	0002	0000
3,3	0000	E000	0000	3FFF	000E	0000	0003	FFFF

Element ID	Left A: Level 4		Left B: Level 4		Right A: Level 4		Right B: Level 4	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	0006	0000	000A	0000	000C	0000	000A	0000
0,1	0006	0000	0002	0000	000C	0000	0008	0000
0,2	0006	0000	0002	0000	000C	0000	0008	0000
0,3	000E	0000	0003	FFFF	000E	0000	FFF8	0000
1,0	0000	0000	0000	0000	0000	0000	0000	0000
1,1	0000	0000	0000	0000	0000	0000	0000	0000
1,2	0000	0000	0000	0000	0000	0000	0000	0000
1,3	0000	0000	0000	0000	0000	0000	0000	0000
2,0	0000	C000	0000	A000	000C	0000	000A	0000
2,1	0000	C000	0000	8000	000C	0000	0008	0000
2,2	0000	C000	0000	8000	000C	0000	0008	0000
2,3	0000	E000	FFFF	8000	000E	0000	FFF8	0000
3,0	0006	0000	000A	0000	0000	6000	0000	A000
3,1	0006	0000	0002	0000	0000	6000	0000	2000
3,2	0006	0000	0002	0000	0000	6000	0000	2000
3,3	000E	0000	0003	FFFF	0000	E000	0000	3FFF

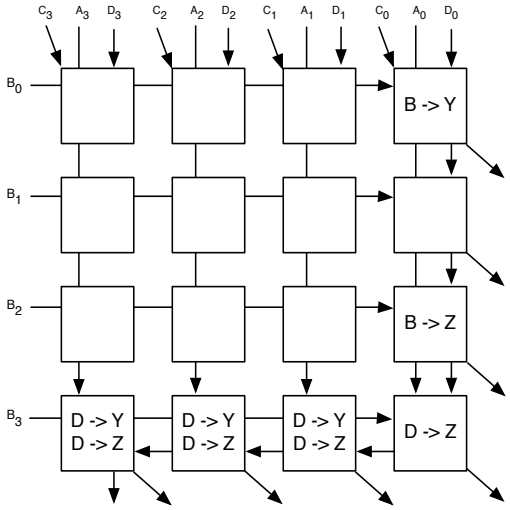
Element ID	Left A: Level 5		Left B: Level 5		Right A: Level 5		Right B: Level 5	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	0060	0000	00A0	0000	00C0	0000	00A0	0000
0,1	0060	0000	0020	0000	00C0	0000	0080	0000
0,2	0060	0000	0020	0000	00C0	0000	0080	0000
0,3	00E0	0000	003F	FFFF	00E0	0000	FF80	0000
1,0	0000	0000	0000	0000	0000	0000	0000	0000
1,1	0000	0000	0000	0000	0000	0000	0000	0000
1,2	0000	0000	0000	0000	0000	0000	0000	0000
1,3	0000	0000	0000	0000	0000	0000	0000	0000
2,0	0000	0C00	0000	0A00	00C0	0000	00A0	0000
2,1	0000	0C00	0000	0800	00C0	0000	0080	0000
2,2	0000	0C00	0000	0800	00C0	0000	0080	0000
2,3	0000	0E00	FFFF	F800	00E0	0000	FF80	0000
3,0	0060	0000	00A0	0000	0000	0600	0000	0A00
3,1	0060	0000	0020	0000	0000	0600	0000	0200
3,2	0060	0000	0020	0000	0000	0600	0000	0200
3,3	00E0	0000	003F	FFFF	0000	0E00	0000	03FF

Shift Control LUTs

Element ID	Left A: Level 6		Left B: Level 6		Right A: Level 6		Right B: Level 6	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	0600	0000	0A00	0000	0C00	0000	0A00	0000
0,1	0600	0000	0200	0000	0C00	0000	0800	0000
0,2	0600	0000	0200	0000	0C00	0000	0800	0000
0,3	0E00	0000	03FF	FFFF	0E00	0000	F800	0000
1,0	0000	0000	0000	0000	0000	0000	0000	0000
1,1	0000	0000	0000	0000	0000	0000	0000	0000
1,2	0000	0000	0000	0000	0000	0000	0000	0000
1,3	0000	0000	0000	0000	0000	0000	0000	0000
2,0	0000	00C0	0000	00A0	0C00	0000	0A00	0000
2,1	0000	00C0	0000	0080	0C00	0000	0800	0000
2,2	0000	00C0	0000	0080	0C00	0000	0800	0000
2,3	0000	00E0	FFFF	FF80	0E00	0000	F800	0000
3,0	0600	0000	0A00	0000	0000	0060	0000	00A0
3,1	0600	0000	0200	0000	0000	0060	0000	0020
3,2	0600	0000	0200	0000	0000	0060	0000	0020
3,3	0E00	0000	03FF	FFFF	0000	00E0	0000	003F

Element ID	Left A: Level 7		Left B: Level 7		Right A: Level 7		Right B: Level 7	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	6000	0000	A000	0000	C000	0000	A000	0000
0,1	6000	0000	2000	0000	C000	0000	8000	0000
0,2	6000	0000	2000	0000	C000	0000	8000	0000
0,3	E000	0000	3FFF	FFFF	E000	0000	8000	0000
1,0	0000	0000	0000	0000	0000	0000	0000	0000
1,1	0000	0000	0000	0000	0000	0000	0000	0000
1,2	0000	0000	0000	0000	0000	0000	0000	0000
1,3	0000	0000	0000	0000	0000	0000	0000	0000
2,0	0000	000C	0000	000A	C000	0000	A000	0000
2,1	0000	000C	0000	0008	C000	0000	8000	0000
2,2	0000	000C	0000	0008	C000	0000	8000	0000
2,3	0000	000E	FFFF	FFF8	E000	0000	8000	0000
3,0	6000	0000	A000	0000	0000	0006	0000	000A
3,1	6000	0000	2000	0000	0000	0006	0000	0002
3,2	6000	0000	2000	0000	0000	0006	0000	0002
3,3	E000	0000	3FFF	FFFF	0000	000E	0000	0003

Mux/OR Control Cell



Description

This Cell Takes a four bit input and generates the 6-bits (2x4-bit) inputs to control the Mux/OR cell.

Inputs

$A_{3:0}$ - unused

$B_{3:0}$ - Original Control Bits

(B_0 = Mux Select, B_2 = OR enable)

$C_{3:0}$ - unused

$D_{3:0}$ - unused

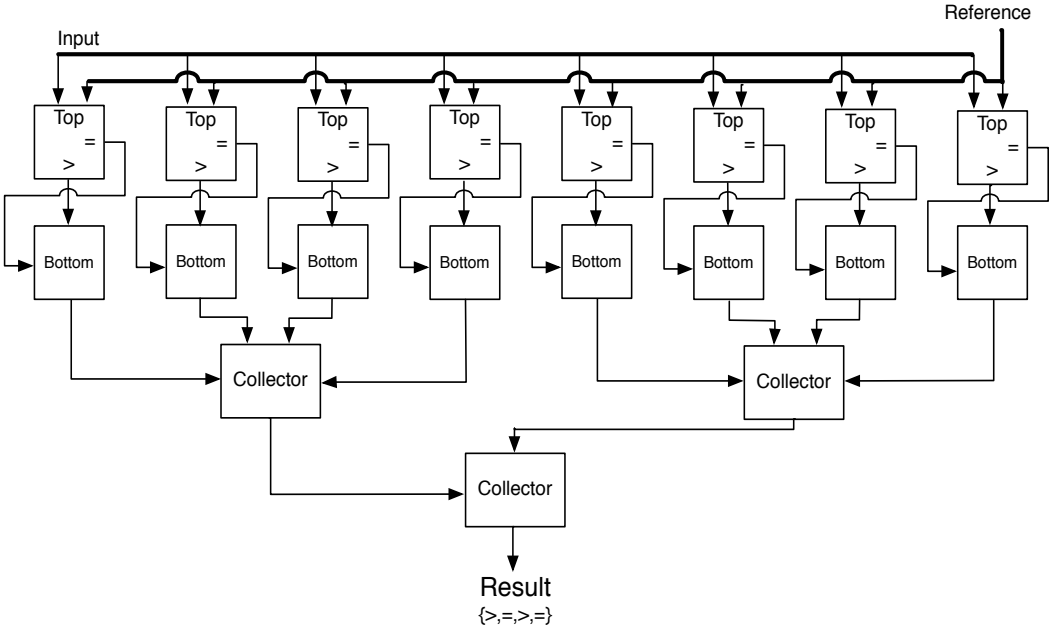
Outputs

$Y_{3:0}$ - B input of MuxOR Cell

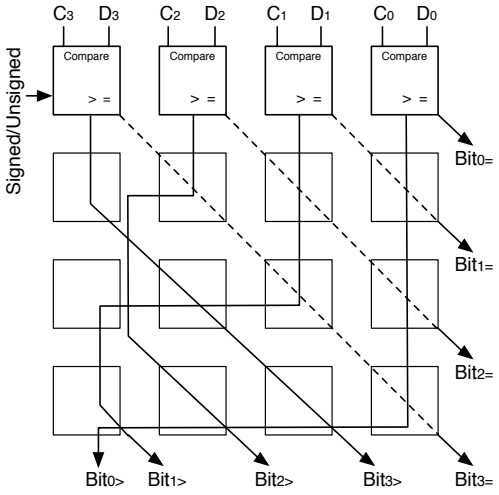
$Y_{7:4}$ - A input of MuxOR Cell

Element ID	Description or Function	Y Config	Z Config
0,0	unused	0000	0000
0,1	unused	0000	0000
0,2	unused	0000	0000
0,3	Output Select	F0F0	0000
1,0	unused	0000	0000
1,1	unused	0000	0000
1,2	unused	0000	0000
1,3	unused	0000	0000
2,0	unused	0000	0000
2,1	unused	0000	0000
2,2	unused	0000	0000
2,3	unused	0000	F0F0
3,0	OR Enable	AAAA	AAAA
3,1	OR Enable	AAAA	AAAA
3,2	OR Enable	AAAA	AAAA
3,3	Or Enable Forward	0000	AAAA

Comparator Module



Top (Information) Cell



Description

This is the cell generates a > and = comparison for each of the 4 input bits for use with the decision cell.

Inputs

$A_{3:0}$ - unused

$B_{3:0}$ - Signed/Unsigned Compare

B_0 - 0/1 - unsigned/signed

$C_{3:0}$ - Value being Compared

$D_{3:0}$ - Reference Value

Outputs

$Y_{3:0}$ - Bitwise C = D comparison

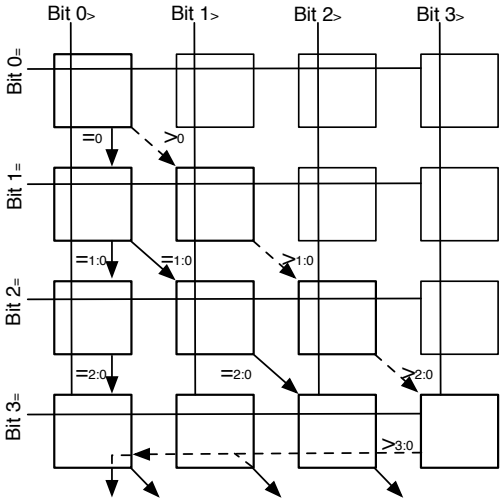
$Y_{7:4}$ - Bitwise C > D comparison

Reversed Order

{Bit_0, Bit_1, Bit_2, Bit_3}

Element ID	Description or Function	Y Config	Z Config
0,0	Comparison Cell	9999	2424
0,1	Comparison Cell	9999	4444
0,2	Comparison Cell	9999	4444
0,3	Comparison Cell	9999	4444
1,0	Routing	CCCC	AAAA
1,1	Routing	CCCC	AAAA
1,2	Routing	CCCC	AAAA
1,3	Routing	CCCC	AAAA
2,0	Routing	CCCC	AAAA
2,1	Routing	CCCC	AAAA
2,2	Routing	CCCC	AAAA
2,3	Routing	CCCC	AAAA
3,0	Routing	CCCC	AAAA
3,1	Routing	CCCC	AAAA
3,2	Routing	CCCC	AAAA
3,3	Routing	CCCC	AAAA

Bottom (Decision) Cell



Description

This cell takes the output of the information cell and combines the bitwise comparisons into a single 4-bit comparison.

Inputs

$A_{3:0}$ - Bitwise > comparisons from information cell (reversed order)

$B_{3:0}$ - Bitwise = comparison from information cell

$C_{3:0}$ - unused

$D_{3:0}$ - unused

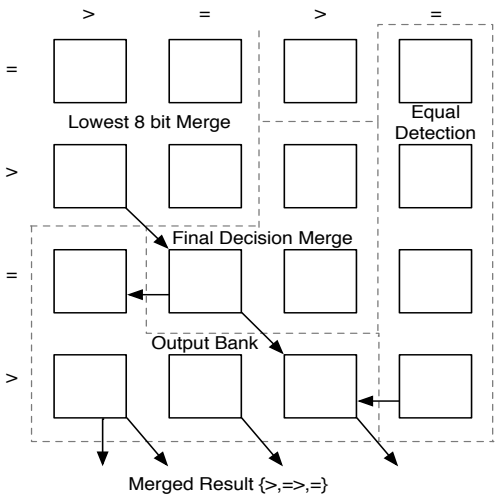
Outputs

$Y_{3:0}$ - unused

$Y_{7:4}$ - 4-bit comparison result formated {>, =, >, =}

Element ID	Description or Function	Y Config	Z Config
0,0	Bit 0 comparisons	0F00	F0F0
0,1	unused	0000	0000
0,2	unused	0000	0000
0,3	unused	0000	0000
1,0	Bits 1:0 Equal Comparison	C0C0	C0C0
1,1	Bits 1:0 Greater Comparison	0FC0	0000
1,2	unused	0000	0000
1,3	unused	0000	0000
2,0	Bits 2:0 Equal Comparison	C0C0	C0C0
2,1	Bits 2:0 Equal Comparison	C0C0	0000
2,2	Bits 2:0 Greater Comparison	0FC0	0000
2,3	unused	0000	0000
3,0	Bits 3:0 Equal and Output	C0C0	AAAA
3,1	Bits 3:0 Equal and Output	AAAA	AAAA
3,2	Bits 3:0 Equal and Output	C0C0	AAAA
3,3	Bits 3:0 Greater Comparison	0000	CFC0

Collector Cell



Description

This cell takes the output of up to four of the decision cells or other collector cells and determine the result of larger comparisons. The input order of precedence is A, B, C, then D.

Inputs

$A_{3:0}$ - Input with 1st highest precedence

$B_{3:0}$ - Input with 2nd highest precedence

$C_{3:0}$ - Input with 3rd highest precedence

$D_{3:0}$ - Input with 4th highest precedence

Outputs

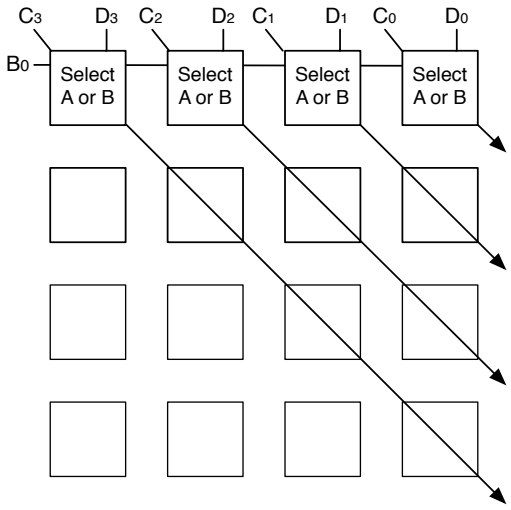
$Y_{3:0}$ - unused

$Y_{7:4}$ - Comparison Result formated {>, =, >, =}

Element ID	Description or Function	Y Config	Z Config
0,0	Lowest 8-bit Merge	AAAA	CCCC
0,1	Lowest 8-bit Merge	FF00	CCCC
0,2	unused	0000	0000
0,3	Equal Detection	0000	8000
1,0	Lowest 8-bit Merge	EEEE	0000
1,1	Lowest 8-bit Merge	0000	8888
1,2	Final Decision Merge	0000	FFC0
1,3	Equal Forwarding	0000	AAAA
2,0	Output Bank	AAAA	AAAA
2,1	Final Decision Merge	0000	EAAA
2,2	Final Decision Merge	0000	AAAA
2,3	Equal Forwarding	0000	AAAA
3,0	Output Bank	AAAA	CCCC
3,1	Output Bank	CCCC	AAAA
3,2	Output Bank	AAAA	AAAA
3,3	Equal Forwarding	0000	AAAA

Support Modules

Mux Cell



Description

This cell implements a 2:1 multiplexer.

Inputs

$A_{3:0}$ - unused

$B_{3:0}$ - Control

(B_0=0 -> D, B_1=0 -> C)

$C_{3:0}$ - Input 1

$D_{3:0}$ - Input 2

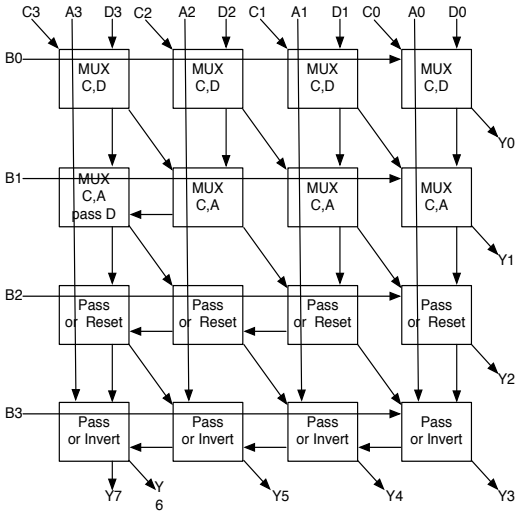
Outputs

$Y_{3:0}$ - Selected Data

$Y_{7:4}$ - unused

Element ID	Description or Function	Y Config	Z Config
0,0	Bit Select	CACA	0000
0,1	Bit Select	CACA	0000
0,2	Bit Select	CACA	0000
0,3	Bit Select	CACA	0000
1,0	Unused	0000	0000
1,1	Routing	CCCC	0000
1,2	Routing	CCCC	0000
1,3	Output	CCCC	0000
2,0	Unused	0000	0000
2,1	Unused	0000	0000
2,2	Routing	CCCC	0000
2,3	Output	CCCC	0000
3,0	Unused	0000	0000
3,1	Unused	0000	0000
3,2	Unused	0000	0000
3,3	Output	CCCC	0000

TriMux Cell



Description

This cell is a 3:1 mux, with the ability to set, reset, and invert the chosen input.

Inputs

A, C, D - Inputs

B - Control Signals

- B_0 - Select C or D
- B_1 - Select A or B_0 Mux Out
- B_2 - Reset
- B_3 - Invert (set if B_2 & B_3)

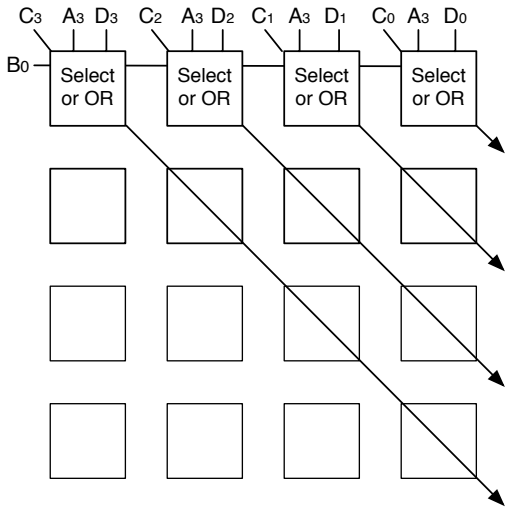
Outputs

$Y_{3:0}$ - unused

$Y_{7:4}$ - Result of Muxing/Reset/Invert/Set

Element ID	Description or Function	Y Config	Z Config
0,0	MUX C,D	CACA	0000
0,1	MUX C,D	CACA	0000
0,2	MUX C,D	CACA	0000
0,3	MUX C,D	CACA	0000
1,0	MUX C,A PASS D	AAAA	FC0C
1,1	MUX C,A	0000	FC0C
1,2	MUX C,A	0000	FC0C
1,3	MUX C,A	0000	FC0C
2,0	Pass or Reset, PASS D - !B AND C	AAAA	OCOC
2,1	Pass or Reset, PASS D - !B AND C	AAAA	OCOC
2,2	Pass or Reset	0000	0A0A
2,3	Pass or Reset	0000	0A0A
3,0	Pass or Invert	AAAA	FCFC
3,1	Pass or Invert	AAAA	FCFC
3,2	Pass or Invert	AAAA	FCFC
3,3	Pass or Invert	0000	FAFA

MUX/OR Cell



Description

This cell is a 2:1 mux that can also OR the two inputs.

Inputs

$A_{3:0}$ - OR enable

$B_{3:0}$ - Output Select

(B_0=0 -> D, B_1=0 -> C)

$C_{3:0}$ - Input 1

$D_{3:0}$ - Input 2

Outputs

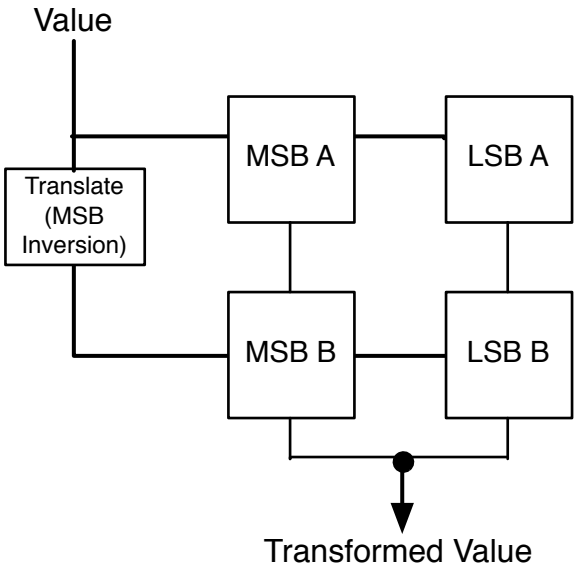
$Y_{3:0}$ - Result

$Y_{7:4}$ - unused

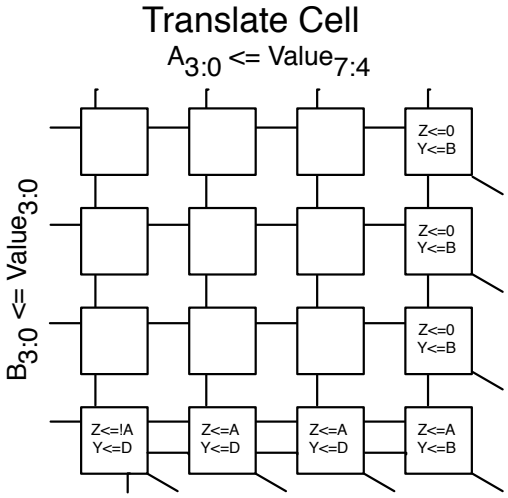
Element ID	Description or Function	Y Config	Z Config
0,0	Bit Select / OR	EECA	0000
0,1	Bit Select / OR	EECA	0000
0,2	Bit Select / OR	EECA	0000
0,3	Bit Select / OR	EECA	0000
1,0	Unused	0000	0000
1,1	Routing	CCCC	0000
1,2	Routing	CCCC	0000
1,3	Output	CCCC	0000
2,0	Unused	0000	0000
2,1	Unused	0000	0000
2,2	Routing	CCCC	0000
2,3	Output	CCCC	0000
3,0	Unused	0000	0000
3,1	Unused	0000	0000
3,2	Unused	0000	0000
3,3	Output	CCCC	0000

AES Modules

S-BOX Module


Value
MSB A
LSB A
Translate
(MSB
Inversion)
MSB B
LSB B
Transformed Value

S-BOX MSB Inversion Cell



Description

This cell is used to facilitate the 256 location 8-bit memory bank that completes the substitution or inverse substitution transform. This is done by inverting the most significant bit.

Inputs

$A_{3:0}$ - Upper 4 bits of Input

$B_{3:0}$ - Lower 4 bits of Input

$C_{3:0}$ - unused

$D_{3:0}$ - unused

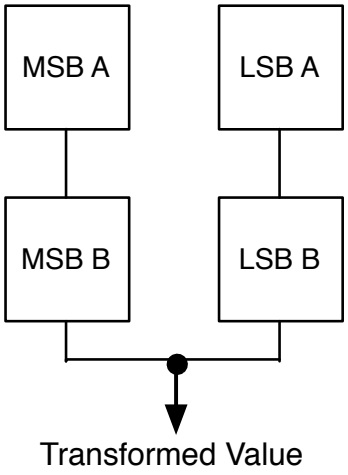
Outputs

$Y_{3:0}$ - LSB Ctrl bits for LSB LUT

$Y_{7:4}$ - MSB Ctrl bits for LSB LUT

Element ID	Description or Function	Y Config	Z Config
0,0	unused	0000	0000
0,1	unused	0000	0000
0,2	unused	0000	0000
0,3	LSBs Output	F0F0	0000
1,0	unused	0000	0000
1,1	unused	0000	0000
1,2	unused	0000	0000
1,3	LSBs Output	F0F0	0000
2,0	unused	0000	0000
2,1	unused	0000	0000
2,2	unused	0000	0000
2,3	LSBs Output	F0F0	0000
3,0	MSB Inversion and MSBs Output	AAAA	00FF
3,1	MSBs Output	AAAA	FF00
3,2	MSBs Output	AAAA	FF00
3,3	LSBs Output	F0F0	FF00

S-BOX LUTs



Function

The use of 4 memory cells with two in parallel connected serially to two other cells forms the core of the S-BOX transform. The inverse transform can be done by simply changing the LUT values.

Inputs

$WA_{7:0}$ - unused

$RA_{7:0}$ - Input value being transformed or Output of S-BOX LUT CTRL

$wi_{3:0}$ - unused

$ri_{3:0}$ - Connects MSB LUTS to LSB LUTs

Outputs

$ro_{3:0}$ - Output of LUTs

S-BOX LUT CONFIG

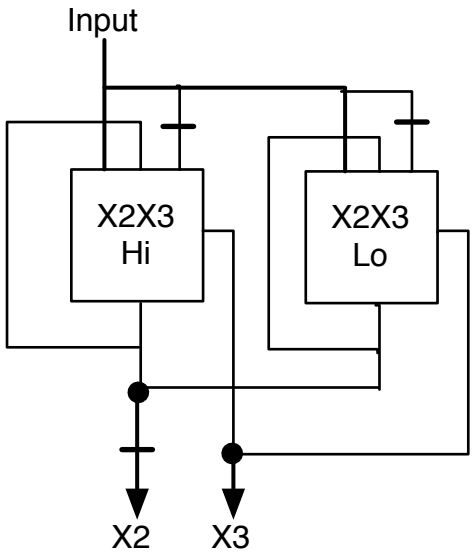
Forward Transform

Element ID	MSB BITS 255:128		MSB BITS 127:0		LSB BITS 255:128		LSB BITS 127:0	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	96CA	0329	BF97	7090	F6AC	6C1B	193D	586A
0,1	B30D	B559	C2FD	B4FF	3B0C	3FFB	3B48	B4C6
0,2	0631	9E08	6D98	DD7F	F334	86B4	4C53	FC7D
0,3	BC48	ECB4	52B8	B11E	C006	EAB5	096C	6EED
1,0	4CB3	7701	5CAA	2EC7	E9DA	849C	1090	20A2
1,1	3F6B	CB91	980A	3CC2	577D	64E0	A163	87FB
1,2	7D8D	CC47	F804	5F7B	6A45	0B2E	7BAE	007D
1,3	2624	B286	C2B0	F977	10BD	B210	B14E	DE67
2,0	B844	E3E1	866A	AC82	C892	FB1B	FFA8	527D
2,1	2559	1782	F3AB	2560	D6CE	2EFC	A428	C424
2,2	0B4F	256F	0D78	7AA4	94EA	D8A9	9781	6F7A
2,3	CE47	2E53	4CE3	0F58	6F24	7A04	8ACB	7A13
3,0	5237	9DE7	E7BA	C28F	4E9D	DB76	2568	EA2E
3,1	21E0	B833	E485	1B3B	AC39	B6C0	23A8	69A2
3,2	54B2	4813	6BC2	AA4E	C870	9740	E61A	4C5E
3,3	F210	A3AE	F7F1	7A49	4F1E	AD39	68AB	4BFA

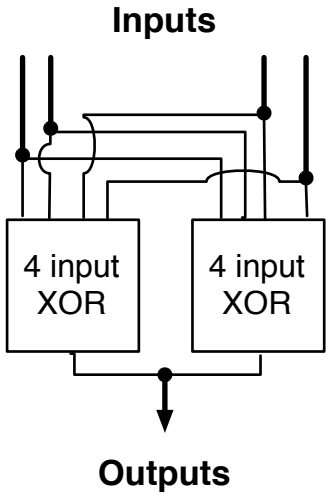
Inverse Transform

Element ID	MSB BITS 255:128		MSB BITS 127:0		LSB BITS 255:128		LSB BITS 127:0	
	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config	Y Config	Z Config
0,0	4BB3	7FC2	F2DA	FD48	DA22	0CD1	E275	8986
0,1	EB14	DEF8	FC43	E2OD	F4F7	6D70	3BE1	4968
0,2	AF7E	F2A1	4167	A5F4	C519	CFB1	A6FA	ED25
0,3	7645	B347	2155	E9B9	66F0	853E	C4F6	F54A
1,0	AF31	52C2	64A4	6534	7B4D	F9B4	3A33	AB82
1,1	FE7B	054B	5B28	F323	A817	4B51	914A	8795
1,2	98C5	572A	95DE	21DA	4B3E	DF05	278A	F97A
1,3	DB67	E21E	43A0	248F	2248	83FB	FA24	4CC2
2,0	FA28	6156	F0F0	CB56	EDDC	C817	61F5	1C62
2,1	A647	C842	0DBF	3D2F	CBA4	D063	FF31	7F9C
2,2	872D	518C	1073	622F	9C44	9269	1D80	C095
2,3	5C36	8F8B	4BDA	D5C7	BBBE	99EB	7A70	3000
3,0	0150	57D3	4719	3377	C21A	4F3C	C67E	14B6
3,1	9B68	A34A	8111	4742	D4ED	0858	066E	CB30
3,2	ABBA	8EF7	8622	3324	08FB	3634	EAFC	A1C4
3,3	9479	6CC4	2425	3563	BB23	F64C	BF68	6944

Mix Column Module

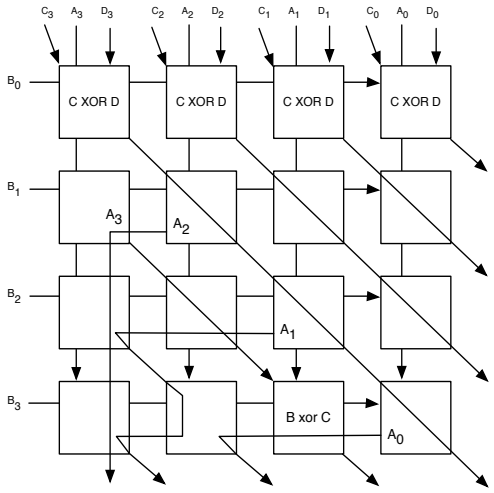


Scaling Component



Summation Component

x2x3 High Cell



Description

This module generates the upper 4-bits of the x2 and x3 factors for the Mix Column Transform.

Inputs

$A_{3:0}$ - Original Value$_{7:4}$

$B_{3:0}$ - Original Value$_{3:0}$

$C_{3:0}$ - Original Value$_{7:4}$

Delayed by 1 cycle

$D_{3:0}$ - x2 Result

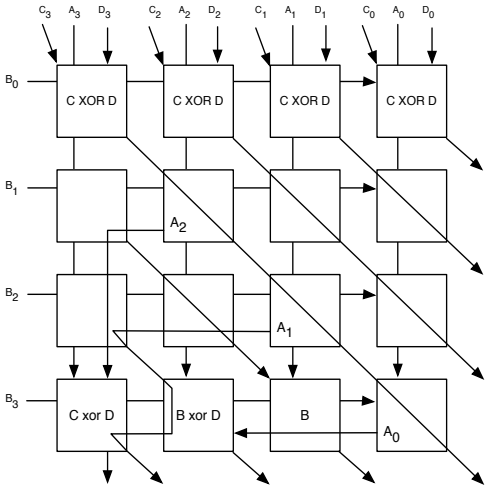
Outputs

$Y_{3:0}$ - x3 Result

$Y_{7:4}$ - x2 Result

Element ID	Description or Function	Y Config	Z Config
0,0	Generate x3	6666	0000
0,1	Generate x3	6666	0000
0,2	Generate x3	6666	0000
0,3	Generate x3	6666	0000
1,0	Forward A3 and A2	FF00	AAAA
1,1	Forward A2	CCCC	FF00
1,2	Route Data	CCCC	0000
1,3	Route Data	CCCC	0000
2,0	Route Data	AAAA	CCCC
2,1	Route Data	CCCC	AAAA
2,2	Forward A1	CCCC	FF00
2,3	Route Data	CCCC	0000
3,0	Output A2 and A1	AAAA	CCCC
3,1	Output A0	AAAA	CCCC
3,2	Output A3 XOR B3	3C3C	AAAA
3,3	Forward A0	CCCC	FF00

x2x3 Lower Cell



Description

This module generates the lower 4-bits of the x2 and x3 factors for the Mix Column Transform.

Inputs

$A_{3:0}$ - Original Value$_{3:0}$

$B_{3:0}$ - Original Value$_{7:4}$

$C_{3:0}$ - Original Value$_{7:4}$

Delayed by 1 cycle

$D_{3:0}$ - x2 Result

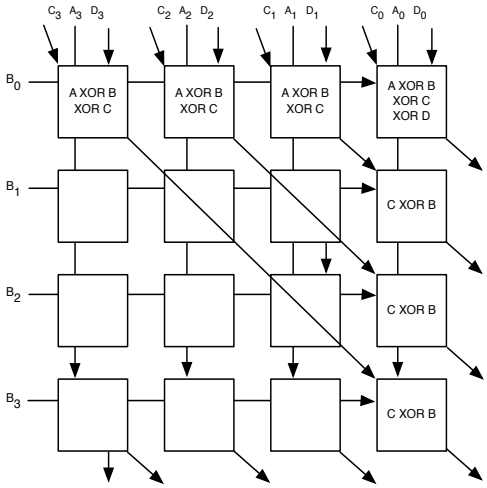
Outputs

$Y_{3:0}$ - x3 Result

$Y_{7:4}$ - x2 Result

Element ID	Description or Function	Y Config	Z Config
0,0	Generate x3	6666	0000
0,1	Generate x3	6666	0000
0,2	Generate x3	6666	0000
0,3	Generate x3	6666	0000
1,0	Route Data	0000	AAAA
1,1	Forward A2	CCCC	FF00
1,2	Route Data	CCCC	0000
1,3	Route Data	CCCC	0000
2,0	Route Data	AAAA	CCCC
2,1	Route Data	0000	AAAA
2,2	Forward A1	CCCC	FF00
2,3	Route Data	CCCC	0000
3,0	Output (A2 xor B3), and A1	AAAA	3C3C
3,1	Output A0 xor B3	5A5A	CCCC
3,2	Output B3	F0F0	AAAA
3,3	Forward A0	CCCC	FF00

4 Input 4-bit XOR



Description

This cell computes A xor B xor C xor D.

Inputs

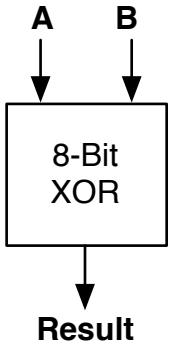
$A_{3:0}$ - Input 1
$B_{3:0}$ - Input 2
$C_{3:0}$ - Input 3
$D_{3:0}$ - Input 4

Outputs

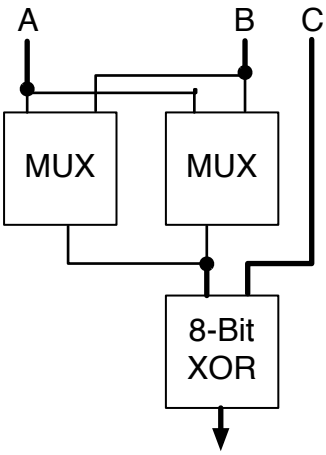
$Y_{3:0}$ - unused
$Y_{7:4}$ - Result

Element ID	Description or Function	Y Config	Z Config
0,0	Generate x3	9966	0000
0,1	Generate x3	9966	0000
0,2	Generate x3	9966	0000
0,3	Generate x3	6996	0000
1,0	Route Data	0000	0000
1,1	Forward A2	CCCC	0000
1,2	Route Data	CCCC	0000
1,3	Route Data	3C3C	0000
2,0	Route Data	0000	0000
2,1	Route Data	0000	0000
2,2	Forward A1	CCCC	0000
2,3	Route Data	3C3C	0000
3,0	Output (A2 xor B3), and A1	0000	0000
3,1	Output A0 xor B3	0000	0000
3,2	Output B3	0000	0000
3,3	Forward A0	3C3C	0000

Add Round Key Module
(8-bit XOR)

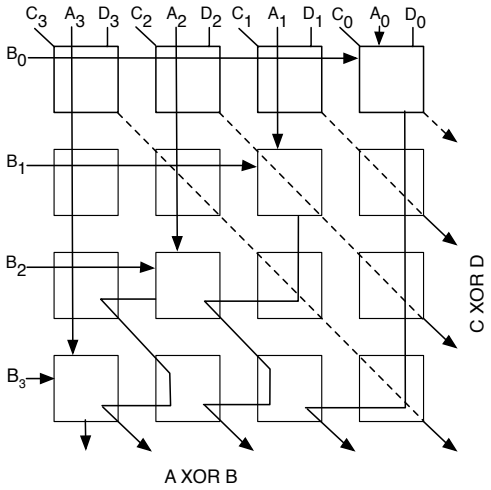


Module



Module With Mix Column Bypass

8-Bit XOR



Description

This module computes an 8-bit XOR operation.

Inputs

$A_{3:0}$ - MSBs of Input 1

$B_{3:0}$ - MSBs of Input 2

$C_{3:0}$ - LSBs of Input 1

$D_{3:0}$ - LSBs of Input 2

Outputs

$Y_{3:0}$ - LSBs of Result

$Y_{7:4}$ - MSBs of Result

Element ID	Description or Function	Y Config	Z Config
0,0	XOR	6666	0000
0,1	XOR	6666	0000
0,2	XOR	6666	0000
0,3	XOR	6666	0FF0
1,0	unused	0000	0000
1,1	Routing	CCCC	0000
1,2	XOR and Routing	CCCC	0FF0
1,3	Routing	CCCC	AAAA
2,0	Routing	AAAA	0000
2,1	XOR and Routing	AAAA	0FF0
2,2	Routing	CCCC	AAAA
2,3	Routing	CCCC	AAAA
3,0	XOR and Routing	AAAA	0FF0
3,1	Routing	AAAA	CCCC
3,2	Routing	AAAA	CCCC
3,3	Routing	AAAA	CCCC

Appendix B

Control Logic

In Chapters 4 and 5 the focus has been directed primarily toward the execution modules and their organization into the execution core of a five-stage processor and how the five-stage architecture could be adapted to the target hardware. During this research the practice of partitioning multiple clock cycles into windows during which the operations could be completed and termed the control synchronization factor, or CSF, has been investigated. In chapter 5 analyses have been provided detailing to how the different CSFs could change the performance of a processor. It also mentions that different CSFs would affect the area requirements of a processor by affecting the size and complexity of the control logic. In the remainder of this appendix an overview of how the control logic could be implemented is presented, which would work for CSFs greater than 9.

B.1 Control Logic at ID Stage

The first aspect of the control logic that will be presented is the core of the ID stage, which ensures data is transmitted to the correct location and that the proper control signals are generated. The processes for routing data to its target module can easily be carried out using a network of multiplexers surrounding the register memory bank, which have previously been discussed. The critical component of the ID stage then becomes the control signal generation because it also allows for data to be transmitted through the multiplexer network to the correct module and for the execution modules to be instructed to carry out the designated function. Control signal generation can be accomplished using a series of LUTs, which can be easily implemented in the target hardware by using a group of cells in memory mode and disconnecting the write inputs. A scheme for doing this is shown in Figure B.1 and would require four cycles of delay for the data and control signals to be finalized, which is fewer than the minimum number of cycles required to implement the PC+4 operation ensuring that the ID stage would not increase the minimum CSF factor or delay for the processor.

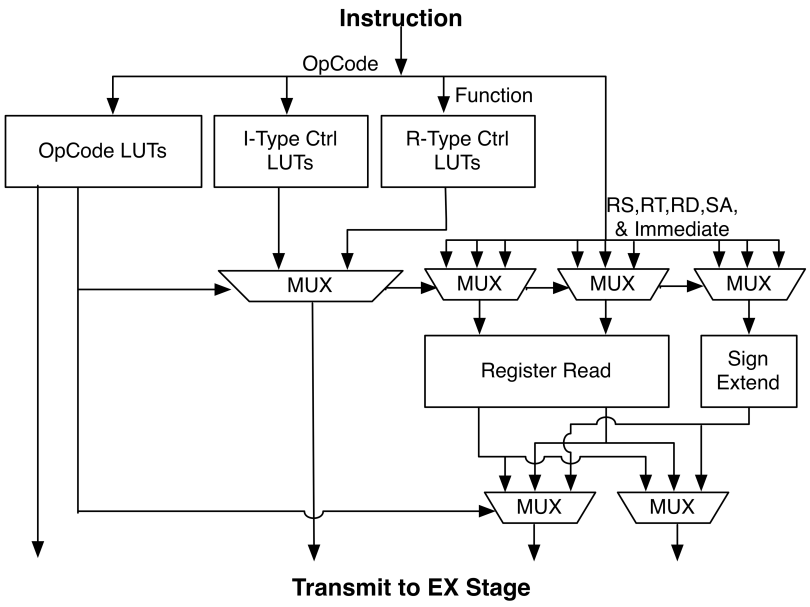


Figure B.1: Control Logic and ID Stage

B.2 Forwarding control logic

Another important control logic component is forwarding. In this subsection a method by which forwarding can be managed is presented. This will allow for the filter block in Figure 3.1 to dynamically increase the CSF window size and control any necessary multiplexers used for forwarding. The most straightforward method for implementation is shown in Figure B.2. Using this architecture the two possible source registers are compared to the past 2 destination registers and if a match is found the forward detect block identifies and communicates the forwarding to occur to the LUT components. The LUT then generates the forwarding mux control signals and alerts the filter block to increase the CSF window size. This method requires only 3 cycles to complete and can be computed in parallel with the other control logic

once the instruction type is known and the sources have been identified, which occurs after the second machine cycle. This means that the forwarding control signals will be generated after the fifth machine cycle and can be computed during any of the examined CSF windows.

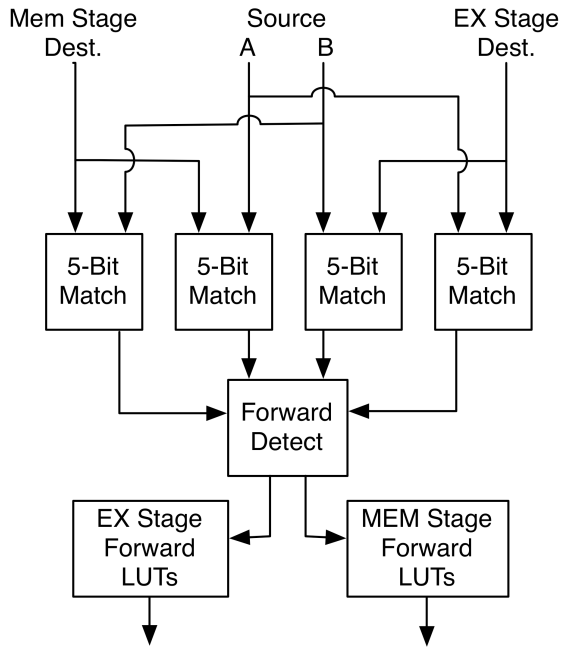


Figure B.2: Forward Detection and Control

Appendix C

Cell Utilization

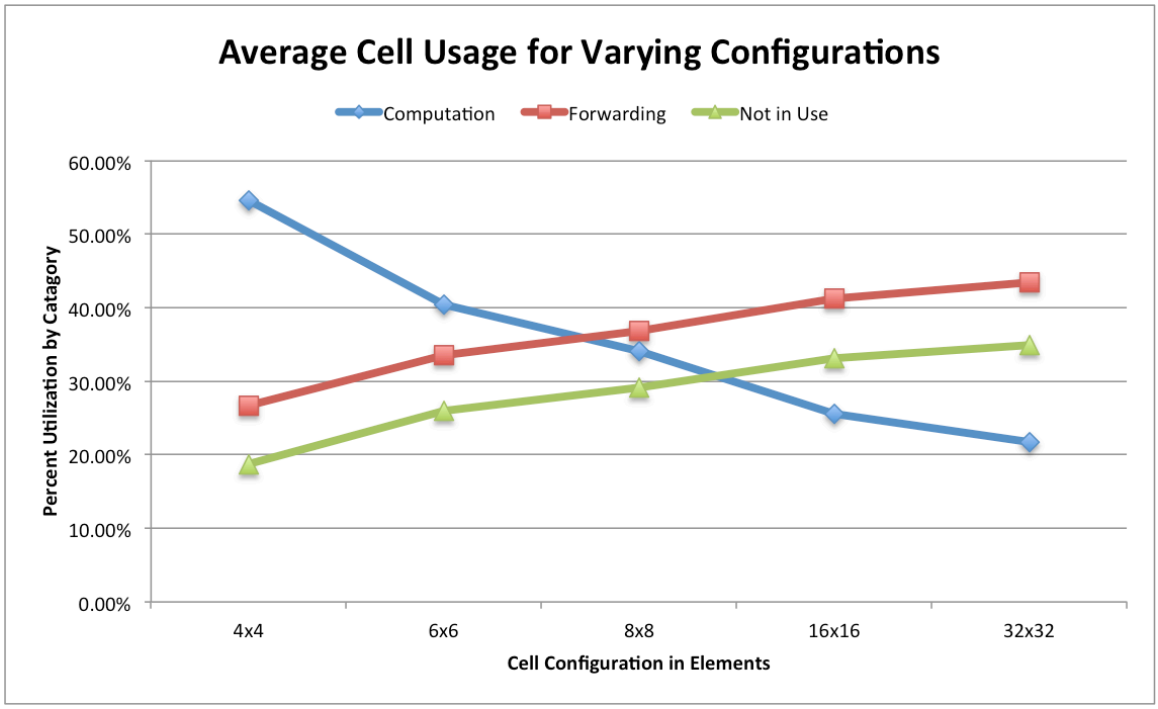
In the course of completing the hardware analysis in Chapter 5 the utilization of each of the execution modules had to be determined to compute the overall five-stage core utilization. Chapter 5 detailed only the overall core utilization, so in this appendix the cell and module utilization is detailed for the cells configured in math mode. The first table, which spans 2 pages, shows the computational, forwarding, and total utilization for the different cell types that are the core of the execution modules for varying cell configurations ranging from 4×4 to 32×32 elements. A figure displaying the average cell usage characteristics then is provided which further illustrates that as the cell size is increased the computational utilization rapidly declines while the forwarding and not in use allocations increase in a similar manner.

Lastly a Table and Figure detailing the operational delay for the execution modules is provided and as expected most of the module execution times increased for

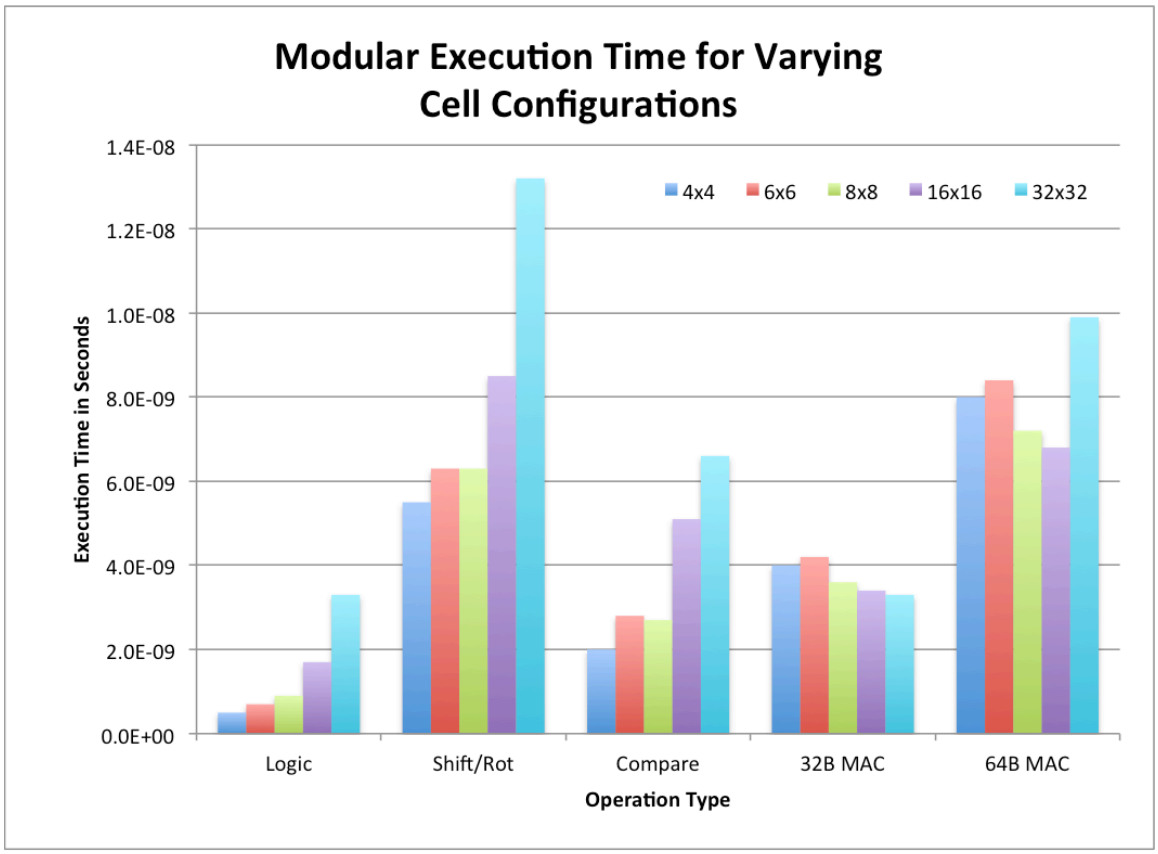
larger cell configurations. The logic, shift, and comparator module all increase their processing time by over a 100% as the cell size grew. The MAC-2 unit was the only execution module not to see a decrease in execution time, but yielded a 17.5% maximum reduction in operation delay for 32-bit operations or a 15% maximum reduction in 64-bit operational delay.

Cell Configuration and Type		Element Allocation					
		Computation		Forwarding		Total	
		Used	%	Used	%	Used	%
4x4	Logic	8	50.0%	8	50.0%	16	100.0%
	MAC Cells	16	100.0%	0	0.0%	16	100.0%
	Shift Cell	10	62.5%	6	37.5%	16	100.0%
	Op Val Cell	8	50.0%	4	25.0%	12	75.0%
	MUX/OR Ctrl	4	25.0%	2	12.5%	6	37.5%
	Info Cell	4	25.0%	12	75.0%	16	100.0%
	Decision Cell	10	62.5%	0	0.0%	10	62.5%
	Collector Cell	12	75.0%	3	18.8%	15	93.8%
	MUX Cell	4	25.0%	6	37.5%	10	62.5%
	TriMux Cell	16	100.0%	0	0.0%	16	100.0%
	MUX/OR Cell	4	25.0%	6	37.5%	10	62.5%
6x6	Logic	12	33.3%	24	66.7%	36	100.0%
	MAC Cells	36	100.0%	0	0.0%	36	100.0%
	Shift Cell	21	58.3%	15	41.7%	36	100.0%
	Op Val Cell	12	33.3%	6	16.7%	18	50.0%
	MUX/OR Ctrl	4	11.1%	2	5.6%	6	16.7%
	Info Cell	6	16.7%	30	83.3%	36	100.0%
	Decision Cell	21	58.3%	0	0.0%	21	58.3%
	Collector Cell	12	33.3%	14	38.9%	26	72.2%
	MUX Cell	6	16.7%	15	41.7%	21	58.3%
	TriMux Cell	24	66.7%	12	33.3%	36	100.0%
	MUX/OR Cell	6	16.7%	15	41.7%	21	58.3%
8x8	Logic	16	25.0%	48	75.0%	64	100.0%
	MAC Cells	64	100.0%	0	0.0%	64	100.0%
	Shift Cell	36	56.3%	28	43.8%	64	100.0%
	Op Val Cell	16	25.0%	8	12.5%	24	37.5%
	MUX/OR Ctrl	4	6.3%	2	3.1%	6	9.4%
	Info Cell	8	12.5%	56	87.5%	64	100.0%
	Decision Cell	36	56.3%	0	0.0%	36	56.3%
	Collector Cell	12	18.8%	29	45.3%	41	64.1%
	MUX Cell	8	12.5%	28	43.8%	36	56.3%
	TriMux Cell	32	50.0%	32	50.0%	64	100.0%
	MUX/OR Cell	8	12.5%	28	43.8%	36	56.3%

Cell Configuration and Type		Element Allocation					
		Computation		Forwarding		Total	
		Used	%	Used	%	Used	%
16x16	Logic	32	12.5%	224	87.5%	256	100.0%
	MAC Cells	256	100.0%	0	0.0%	256	100.0%
	Shift Cell	136	53.1%	120	46.9%	256	100.0%
	Op Val Cell	32	12.5%	16	6.3%	48	18.8%
	MUX/OR Ctrl	4	1.6%	2	0.8%	6	2.3%
	Info Cell	16	6.3%	240	93.8%	256	100.0%
	Decision Cell	136	53.1%	0	0.0%	136	53.1%
	Collector Cell	12	4.7%	129	50.4%	141	55.1%
	MUX Cell	16	6.3%	120	46.9%	136	53.1%
	TriMux Cell	64	25.0%	192	75.0%	256	100.0%
	MUX/OR Cell	16	6.3%	120	46.9%	136	53.1%
32x32	Logic	64	6.3%	960	93.8%	1024	100.0%
	MAC Cells	1024	100.0%	0	0.0%	1024	100.0%
	Shift Cell	528	51.6%	496	48.4%	1024	100.0%
	Op Val Cell	64	6.3%	32	3.1%	96	9.4%
	MUX/OR Ctrl	4	0.4%	2	0.2%	6	0.6%
	Info Cell	32	3.1%	992	96.9%	1024	100.0%
	Decision Cell	528	51.6%	0	0.0%	528	51.6%
	Collector Cell	12	1.2%	521	50.9%	533	52.1%
	MUX Cell	32	3.1%	496	48.4%	528	51.6%
	TriMux Cell	128	12.5%	896	87.5%	1024	100.0%
	MUX/OR Cell	32	3.1%	496	48.4%	528	51.6%



Operation Timing (sec)										
	4x4		6x6		8x8		16x16		32x32	
	Cyc.	*Time*	*Cyc.*	*Time*	*Cyc.*	*Time*	*Cyc.*	*Time*	*Cyc.*	*Time*
Logic	1	5.0E-10	1	7.0E-10	1	9.0E-10	1	1.7E-09	1	3.3E-09
Shift/Rot	11	5.5E-09	9	6.3E-09	7	6.3E-09	5	8.5E-09	4	1.3E-08
Compare	4	2.0E-09	4	2.8E-09	3	2.7E-09	3	5.1E-09	2	6.6E-09
32B MAC	8	4.0E-09	6	4.2E-09	4	3.6E-09	2	3.4E-09	1	3.3E-09
64B MAC	16	8.0E-09	12	8.4E-09	8	7.2E-09	4	6.8E-09	3	9.9E-09



Appendix D

Publications

This section list the papers published and submitted for publication during the course of this research.

D.1 Journal Papers

[1] J. Van Dyken, and J.G. Delgado-Frias, "FPGA schemes for minimizing the power-throughput trade-off in executing the Advanced Encryption Standard algorithm." in *Journal of Systems Architecture*, 2-3 February 2010, pp. 56

[2] J. Van Dyken, and J.G. Delgado-Frias, "A Medium-Grain Modular Processor: Execution Cores and Forwarding Schemes." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, *submitted May 2012.*

D.2 Conference Papers

[1] J. Van Dyken, J.G. Delgado-Frias, and S. Medidi. FPGA Schemes with Optimized Routing for the Advanced Encryption Standard. *ERSA'2008 The International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, USA, July 14 - 17, 2008

[2] J. Van Dyken, and J.G. Delgado-Frias. A medium-grain reconfigurable processing unit, *2010 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 1-4 Aug. 2010

[3] J. Van Dyken, and J.G. Delgado-Frias. A medium-grain reconfigurable processor organization*, 2011 54th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 7-10 Aug. 2011

[4] J. Van Dyken, and J.G. Delgado-Frias. "A Superscalar Processor for a Medium-Grain Reconfigurable Hardware." *2012 55th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 5-8 Aug. 2012, Accepted for Publication.