A versatile, scalable, and open memory architecture in CMOS 0.18 µm

Karl Leboeuf
University of Windsor

A Versatile, Scalable, and Open Memory Architecture in CMOS $0.18\mu m$

by

Karl Leboeuf

A Thesis
Submitted to the Faculty of Graduate Studies through the Department of Electrical and Computer Engineering in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science at the University of Windsor

Windsor, Ontario, Canada
2008

Your file Votre référence
ISBN: 978-0-494-47093-0
Our file Notre référence
ISBN: 978-0-494-47093-0

Declaration of Originality

Abstract

A lookup table is a permanent memory storate element in which every stored value corresponds to a unique address. Range addressable lookup tables differ in that every stored value corresponds to a *range* of addresses. This type of memory has important applications in a recently proposed central processing unit which employs a multi-digit logarithmic number system that is well suited for digital signal processing applications.

This thesis details the work done to improve range addressable lookup tables in terms of operating speed and area utilization. Two range addressable lookup table designs are proposed. Ideal design parameters are determined. An integrated circuit test platform is proposed to determine the real-world ability of these lookup tables. A case study exploring how non-linear functions can be approximated with range addressable lookup tables is presented.

This work is dedicated to many teachers in my life who have helped shape me into what I am today.

Acknowledgments

I would like to express my sincere thanks and appreciation to Dr. Roberto Muscedere for his support and guidance throughout the progress of this thesis. In addition, I would also like to thank Mr. Ashkan Hosseinzadeh Namin, Ms. Christine Koncan and my family for their unending help and support.

Contents

List of Figures

List of Tables

List of Abbreviations

ASIC	Application Specific Integrated Circuit
ATPG	Automatic Test Pattern Generator
BIST	Built-In Self Test
CAD	Computer Aided Design
CMC	Canadian Microelectronics Corporation
CMOS	Complimentary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
DRC	Design Rule Check
DSP	Digital Signal Processing
FET	Field-Effect Transistor
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
LFSR	Linear Feedback Shift Register
LUT	Lookup Table
MDLNS	Multi-Dimensional Logarithmic Number System
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MSE	Mean Squared Error
MUX	Multiplexer
NAND	Not-AND
NMOS	n-Channel MOSFET
PMOS	p-Channel MOSFET
RALUT	Range Addressable Lookup Table
RAM	Random Access Memory
ROM	Read Only Memory
SPICE	Simulation Program with Integrated Circuit Emphasis
TLNS	Two-Digit Logarithmic Number System
TSMC	Taiwan Semiconductor Manufacturing Company
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Chapter 1

Introduction

Since the first integrated circuit was successfully created in September of 1958, fabrication technology has been constantly advancing; transistors become increasingly small, allowing for faster designs at lower costs. The steady progress of miniaturization has continued almost unimpeded for fifty years until recently. As transistor sizes approach atomic sizes, numerous problems begin to arise and researchers must look elsewhere for performance improvements. Investigation into new types of digital computer architectures is one approach researchers are taking to continue to advance the state of the art of the integrated circuit.

Among these new architectures are processors which employ exotic number systems that excel in performing certain mathematical operations, such as multiplication, division and exponentiation [4], [5]. These are important operations for many digital signal processing applications, such as in a hearing aid processor, and in digital filtering [6], [15], [7]. The multi-digit logarithmic number system has recently been proposed for such purposes, and a processor employing this number system has been designed; the two-digit logarithmic number system CPU [2]. This processor is able to quickly and efficiently perform digital signal processing instructions, however it is reliant on the use of range addressable lookup tables to perform certain crucial operations, including conversion to and from binary [14].

Lookup tables, or LUTs, are a common form of permanent memory used in many applications. Every LUT functions by giving it an input address, causing it to output a particular stored value. Every value stored in the LUT corresponds to a unique input address.

Range addressable lookup tables, or RALUTs, function similarly to LUTs, with one key difference. Every value that is stored in the RALUT corresponds to a *range* of input addresses. This difference allows the table size to be significantly reduced for many applications, particularly when approximating non-linear functions.

Consider, for example, the hyperbolic tangent function in Figure 1.1. It can be approximated with the use of a lookup table. The input to the lookup table is the quantized x-axis of the function, while the corresponding y-axis values are stored into the LUT, acting as outputs. Any degree of precision is possible, however more precision will require a larger table size. An example of a hyperbolic tangent function approximated by a LUT is shown in Figure 1.2. It is approximated with 8 values, and it can be seen that this is a poor approximation with very large error, particularly in the points close to the x-axis' origin. Notice that in a LUT, the stored values are evenly spaced across in input range.

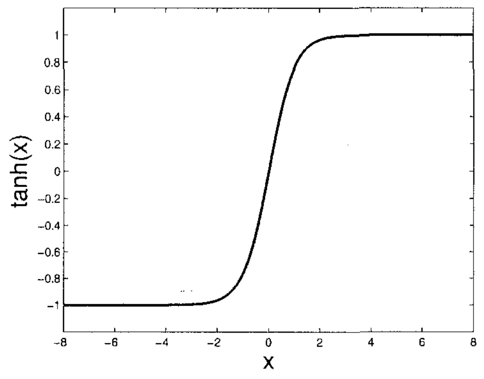

Figure 1.1: The Hyperbolic Tangent Function

RALUTs were designed to excel at this type of task. Similar to the LUT approximation, they require the input address and stored output values to be quantized. RALUTs possess an important advantage in approximating this type of function, however; the stored y-values can be placed wherever the hardware designer wishes. This allows greater accuracy to be achieved with the same amount of points, as in Figure 1.3. This figure shows the same function being approximated with the same number of points, however the points are placed in a way such that the maximum error is minimized. The points are placed closely together when the function is changing rapidly near the origin, and further apart along the extremities where the function is hardly changing at all. Alternately, if the error in the LUT approximation was acceptable, an RALUT implementation with the same maximum error could be used with as few as 4 stored values, cutting the table size in half,

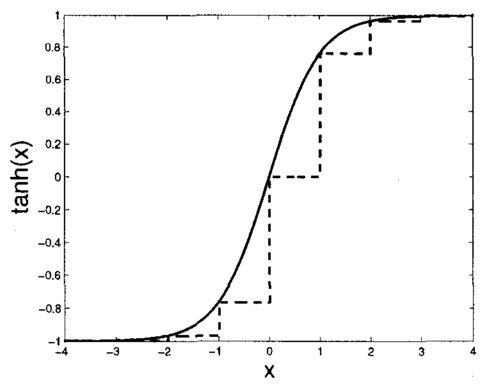

Figure 1.2: Lookup Table Approximation of tanh(x) with Eight Points

achieving a major area savings.

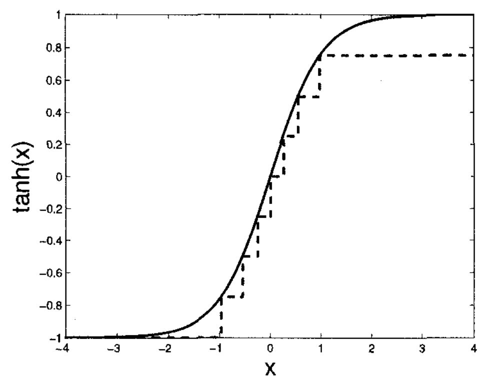

Figure 1.3: Range Addressable Lookup Table Approximation of tanh(x) with Eight Points

The focus of this thesis is to advance the state of the art of range addressable lookup tables. To achieve this, an existing RALUT design is rescaled to use a newer fabrication process, and then further enhanced, reducing its area utilization and increasing its operating speed. A test platform is proposed to allow real-world performance data to be collected. Finally, a new application for RALUTs is proposed in the area of artificial neural networks.

This thesis consists of seven chapters. In this first chapter, the issue being addressed is presented, and the structure of the remaining chapters is laid out. Chapter two provides pertinent background information. Chapter three presents the RALUT architecture in detail, while chapter four proposess two different RALUT designs along with their performance results. Chapter five details the creation of a test platform for RALUT memory designs, and chapter six describes the utility of RALUTs in the area of artificial neural networks. In the final chapter, concluding remarks, and suggestions for possible directions in future research in this are are made.

Chapter 2

Background

This chapter provides the reader important background information regarding lookup tables, a brief review of static CMOS, domino logic, as well as range addressable lookup tables.

2.1 Lookup Tables

Lookup tables, or LUTs are a form of non-volatile, read-only memory. They are often used in hardware design to store functions, and are desirable in applications that require high operating speeds. As shown in Figure 2.1, LUTs have two I/O ports, an input address bus and an output bus.

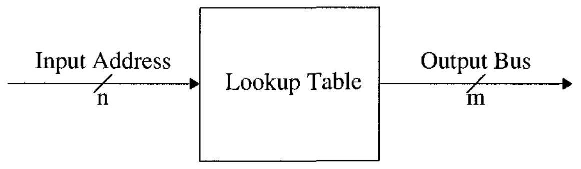

Figure 2.1: Lookup Table Block Diagram

In literature and in practice, LUTs are generally measured with two dimensions: address space and word size. The address space is defined by Equation 2.1; it is simply the number of different addresses that can be referred to by the input bus. For example, a LUT with a 10-bit input bus can refer to 1024 unique addresses.

$$\text{Total Addresses} = 2^{\text{Address Bits}} \tag{2.1}$$

The other defining parameter of the LUT, word size, is the width in bits of the output bus. Every stored word in memory is referred to by a unique address. Together, these two dimensions summarize the storage ability of the LUT, and the expression "address space × word size" will be referred to as the size of the LUT.

2.1.1 Lookup Table Implementation

ROM Lookup Table Implementation

Different techniques for implementing LUTs exist. One of the most common methods is ROM implementation, where the input addresses and output values are permanently stored into a hardware array. Important advantages of this approach are simplicity and predictability; given that the address space × word size parameters of the LUT remain constant, the specific words being stored into the LUT do not affect its area utilization or maximum operating speed. Additionally, it is worth mentioning that the only practical limitation of LUT size when using a ROM implementation is silicon area. These are a highly desirable qualities when designing digital systems that make use of LUTs.

One disadvantage of using a ROM array is that a proprietary tool called a "memory compiler" must be used in order to implement them in hardware. Such tools are expensive, and closed-source, meaning that the hardware designer does not have access to the internals of the ROM design. Furthermore, memory compilers are not necessarily versatile; a compiler that works for a CMOS $0.35\mu m$ process may not function with a CMOS $90nm$ process.

Another problem with ROM implementation is that they consume a very large area as the number of address bits increases. Figure 2.2 shows the internal workings of the ROM implementation of the LUT. It consists of an address decoder, which scales in size with the number of input bits, and the word lines, which scale in size with the number of output bits. Thus, the total area of a ROM LUT scales approximately with Equation 2.2. As the number of input bits increases, the area utilization increases dramatically, possibly rendering the ROM implementation of LUTs impractical for very large addresses.

$$LUT_{Area}\ \alpha\ 2^{n} \times (n+m) \tag{2.2}$$

Despite these shortcomings, skillful hardware designers have found a place for the ROM array implementation of LUTs in many different devices, including FPGAs, microcontrollers, and microprocessors.

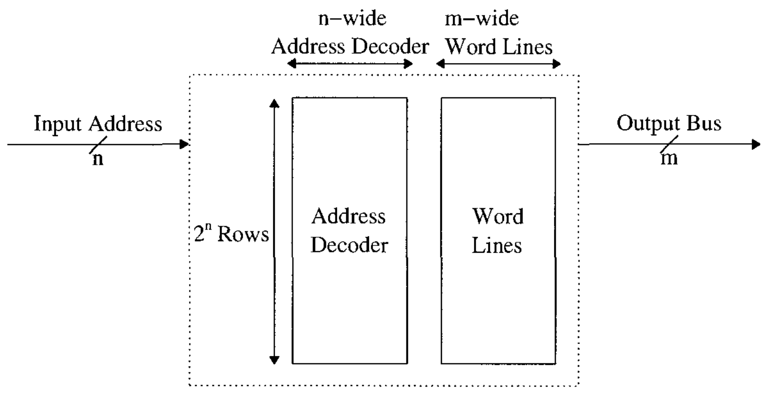

Figure 2.2: Lookup Table Internal Block Diagram

Logic Synthesizer and Logic Gate Implementation

An alternative approach to implementing LUTs is to use a logic synthesizer, sometimes called a hardware compiler, to take the LUT's I/O characterisitcs, and implement it using logic gates. The benefit of implementing LUTs as a series of simple logic gates is that there is a high probability that the design can be simplified, yielding a large area reduction. The reason for this is that the hardware compiler carefully examines the specific I/O behaviour of a particular and "optimizes away" redundant logic. To demonstrate how a hardware compiler can optimize a design, the following explanation will refer to Figure 2.3.

Suppose a designer wanted, for whatever reason, to create a 256×4 lookup table where every even address would place the bit pattern "1111" on the output bus, and "0000" for every odd address. Given this specific design, a hardware compiler would most likely only use the least significant bit to determine if the address were even or odd, and simply connect this signal to an inverter. Referring to Figure 2.3 part (a), when the address is even, the least significant bit is '0', and the address passes through an inverter and the first word line is enabled. Since the second word line receives the signal of '0', its contents are not placed on the output bus. The alternate situation occurs when an odd address is used and the least significant bit is '1'. This hardware compiler implementation would require (approximately) a few dozen transistors, while occupying a very tiny area, and operate at very high speeds. As seen in Figure 2.3 part (b), a ROM implementation would fill alternating word lines with these two patterns, occupying the entire 256×4 ROM array, which is vastly more area

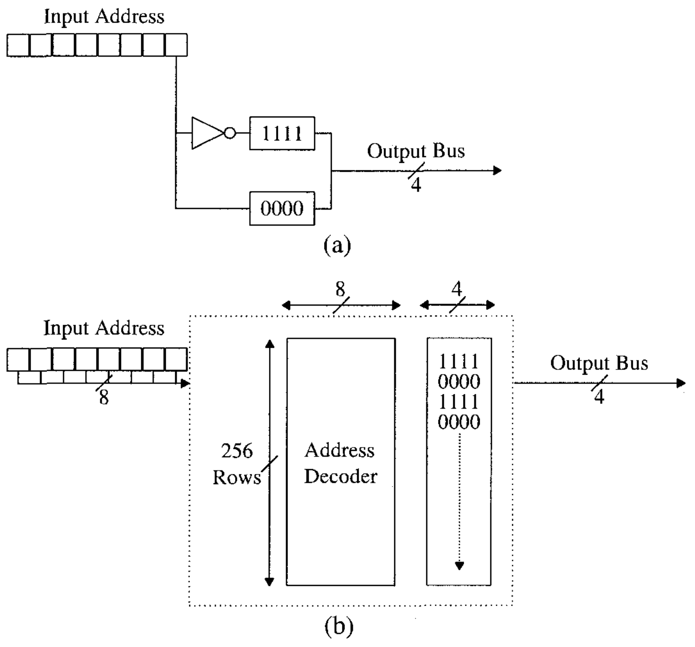

Figure 2.3: Hardware Compiler Result (a) Compared with ROM Implementation (b)

than the hardware compiler version.

While this expample is an ideal case, it does demonstrate the capability of the hardware compiler. Under most scenarios, such dramatic reductions are not possible, however area utilization *is* typically significantly less than with ROM implementations. The exact area utilization and operating speed depend heavily on the exact bit patterns used in the word lines of the LUT. This is a disadvantage for hardware designers, as precise timing and area information are unknown until the design is synthesized, and it is possible that small design changes made to the word lines will greatly affect these LUT attributes. Another disadvantage is that this approach is not feasible for very large LUT sizes. The processor and memory requirements of the logic synthesizer will increase to the point where a single workstation equipped with a large amount of RAM still requires days or even weeks to determine a gate-level design for the LUT.

This approach does not use a memory compiler, however it does require a standard cell library, and a logic synthesizer tool: a proprietary library and a commercial tool. It is a scalable design, in that any number of input bits, output bits, and rows can be used, with the only theoretical limit being the area utilization. Currently there are no known open source standard cell libraries, and a license for a hardware compiler is extremely expensive, however most digital hardware designers do have access to both of these.

2.2 A Brief Review of Domino Logic

Many different logic families exist for implementing logic gates; the building blocks of digital circuit design. The designs presented in this work make use of static CMOS, and domino-logic, a type of dynamic CMOS logic. The goal of this section is to provide a brief overview of these logic styles, and to impress the reader with a fundamental understanding of their mechanics, advantages, and disadvantages.

2.2.1 Static CMOS Logic

Static CMOS is a very common logic style; it is used in almost every type of design [24]. In static CMOS, a direct, low impedance path exists from the output of the gate to either VDD or VSS. PMOS transistors act as the pull-up network, while NMOS transistors form a pull-down network. When the appropriate inputs arrive at the transistors' gates, the circuit evaluates, and the output node is either connected directly to either VDD or VSS.

A Static CMOS 2-input NAND Gate

For example, a static CMOS 2-input NAND gate is shown in Figure 2.4. It implements the function described by Table 2.2.1. The NMOS transistors connect the output directly to ground when both inputs A and B are equal to *logic 1*, forming the pull-down network. Similarly, when either (or both) of A or B are at *logic 0*, the PMOS transistors forming the pull-up network connect the output node directly to VDD.

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.1: The NAND Function Input and Output Behaviour

Static CMOS Properties

Static CMOS logic gates are relatively easy to implement, and are not overly sensitive to loading. This is a highly desirable property, as it allows different static CMOS gates to be combined together

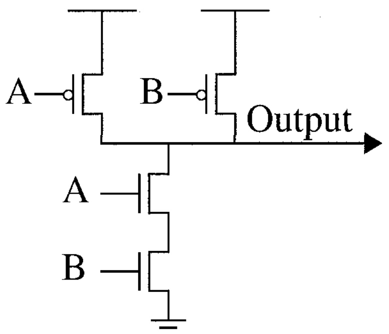

Figure 2.4: Schematic for a 2-Input, Static CMOS NAND Gate

with ease to form larger circuits. For this reason, standard cell libraries are composed of this type of logic gate.

Another important feature of static CMOS is near-zero static power consumption. Static power consumption refers to the power being consumed while the device is not switching. In other words, as long as the inputs to the logic gate remain constant, very little power is consumed. The reason why a small amount of power is still being consumed during this operating state is due to charge leakage; a physical phenomenon in which some of the charge carriers are able to "leak" through the transistor's gate oxide. This is not a major issue for fabrication processes larger than *90nm* due to the relatively large oxide thickness, however when using fabrication technology at the *90nm* node and beyond, this may become a greater concern.

Static CMOS gates do, on the other hand, consume switching power. This is due to the fact that when the gate's output is switching from *logic 0* to *logic 1*, or vice-versa, both the pull-up network and pull-down network will be conducting current for a very short time interval. In other words, for a short instant (on the order of picoseconds), a short circuit from VDD to VSS is available. In addition to consuming power, this can generate noise, which may be an issue if there are analog circuits operating nearby.

One drawback to static CMOS, is the reliance on PMOS transistors to form the pull-up network. PMOS transistors rely on "holes", rather than electrons as their charge carriers, which are much slower [10]. It is for this reason that PMOS transistors must be significantly larger than an NMOS transistor in order to possess equivalent current drive capability. This results in greater area utilization and slower operating speeds compared to a logic style that relies more heavily on NMOS transistors.

2.2.2 Domino Logic

A dynamic logic gate is one in which the output is only valid for a short amount of time after the result is produced. [18] Athough this sounds quite restrictive, dynamic CMOS networks are useful for high-speed system design. Dynamic logic encompasses several different logic families, including domino logic.

Domino logic uses a clock signal to "precharge" a node, and later "evaluate" the node via an NMOS pull-down network. It is best illustrated via an example, as in Figure 2.5.

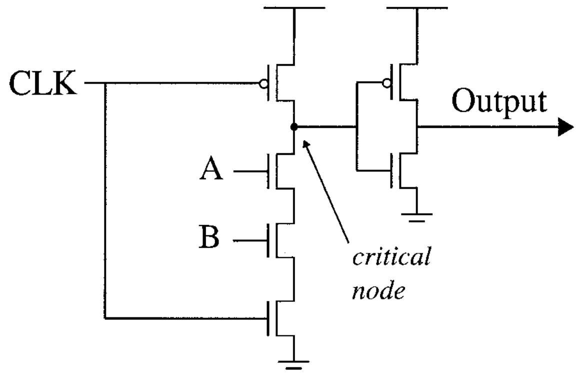

Figure 2.5: A Domino Logic 2-Input NAND Gate

In this schematic, when the clock signal is at *logic 0*, the PMOS or "precharge" transistor in this case, pulls the critical node to *logic 1*. This node connects to the gate of the inverter, and the output of the gate at this point is *logic 0*. Also notice that at this time, the NMOS transistor connected to the clock signal, the "evaluate" transistor, is not currently conducting, eliminating any path to ground that the critical node may have had.

As time elapses, the clock makes the transition to *logic 1*, the precharge transistor stops conducting, while the evaluate transistor opens a path to ground through the NMOS pull-down network. At this point, one of two events may occur. If inputs A and B are both at *logic 0*, the pull-down network completes the path to ground from the critical node to the evaluate transistor, discharging the critical node, and bringing the gate output to *logic 1*. Alternately, if either of A or B are low, a path to ground does not exist, and the charge on the critical node remains. The gate output stays at *logic 0*.

Domino logic's many advantages over static CMOS stem from several facets of its design. First, only the faster NMOS transistors are used to evaluate the circuit, and the lack of a large pull-up network greatly reduces parasitic capacitance, significantly enhancing operating speed. Second, power reduction is possible; there is never a short circuit from VDD to VSS as there is in static CMOS. Another advantage is the reduced area utilization made possible by only implementing the pull-down network as opposed to both pull-down and pull-up networks. For example, a 4-input NAND gate would require only two additional NMOS transistors than 2-input gate in Figure 2.5, whereas a static CMOS 4-input NAND gate would require four additional transistors: two NMOS and two PMOS.

Despite these advantages, domino logic design presents a separate set of challenges. Domino gates are sensitive to charge leakage and charge sharing, and suffer from these effects. As described in the previous section, charge leakage is the physical phenomenon in which some of the charge leaks through the transistor's gate oxide. In addition to dissipating power and creating heat, this is particularly problematic in domino gates; if the charge at the critical node dissipates too rapidly, the output will become invalid. To eliminate this concern, an additional transistor is placed between VDD and the critical node, and controlled by the gate's output, as in Figure 2.6

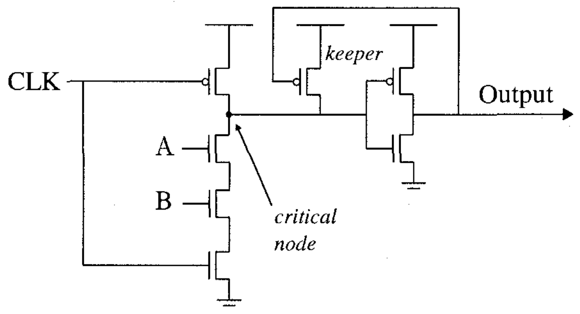

Figure 2.6: A Domino Logic 2-Input NAND Gate, with Keeper Transistor

This transistor is referred to as the "keeper". Its role is to maintain charge on the critical node that would otherwise bleed away over time due to charge leakage [20]. It is a very weak transistor; it is deliberately sized so that it possesses low current drive. This is done to ensure that when the circuit legitimately attempts to discharge the critical node, the keeper does not overpower the pull-down network, reducing operating speeds.

The other domino logic concern, charge sharing, is the effect of all transistors attached to a common node contributing to the charge stored there. A larger amount of charge will require larger transistors, and more time to dissipate during the evaluate phase, reducing circuit performance. Hardware designers must be aware of this phenomenon, and carefully plan their designs around this problem. It is due to domino logic's sensitivity to this effect that limits its use to hand-designed circuits, rather than standard cell libraries.

2.2.3 Range Addressable Lookup Tables

RALUTs were originally proposed in [14] as an efficient way to implement certain non-linear, discontinuous functions used for number conversion as well as addition and subtraction in a multi-dimensional logarithmic number system (MDLNS) [16] The MDNLS number system is able to perform the multiplication, exponentiation, and devision operations with extreme efficiency, rendering its use extremely beneficial in certain applications such as DSP, cryptography, and multimedia processing. A primary concern of implementing a processor that employs this number system in hardware is number conversion to and from the binary number system, which is traditionally used extensively throughout most hardware designs. The conversion process is relatively time consuming unless some special hardware techniques are used. Lookup tables were proposed, however it was shown that they become very large in size as greater conversion accuracy is needed and a larger address space is required.

A block diagram showing the main components of the RALUT is shown in Figure 3.1. The architecture is divided into two main sections, the address decoder and the word lines. The input address is connected to the address decoder, and a single word line is enabled and palced on the output bus. There are only k rows, whereas in the LUT there are 2^n rows. As will be shown, the number of rows in a RALUT is not dependent on the number of bits in the input address. Finally, it is worth noting the presence of a clock signal. Although the RALUT functions like a combinational logic circuit, due to its domino logic implementation it will require a clock.

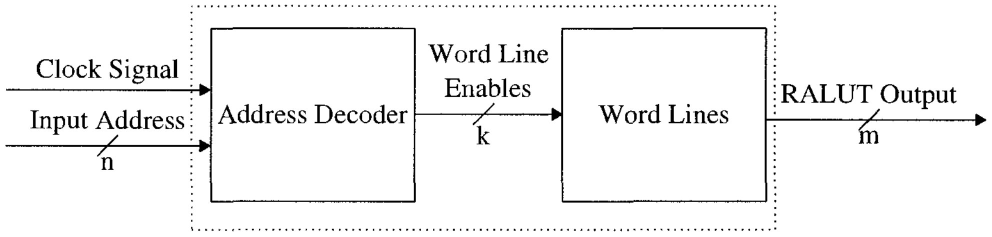

Figure 2.7: Block Diagram of the RALUT, with n Address Bits, m Output Bits, and k Rows

Range addressable lookup tables, or RALUTs, function very similarly to the LUTs described in the previous section. The key difference is that every stored value in a RALUT is referred to by a *range* of addresses, as opposed to a single, unique address, as in Figure 2.8. As shown in the figure, for a RALUT, every address is compared to the values stored in the address decoder. If the input address is larger than a given row, but smaller than the next, that word line is activated.

This architecture allows for a tremendous area savings when implementing specific types of non-

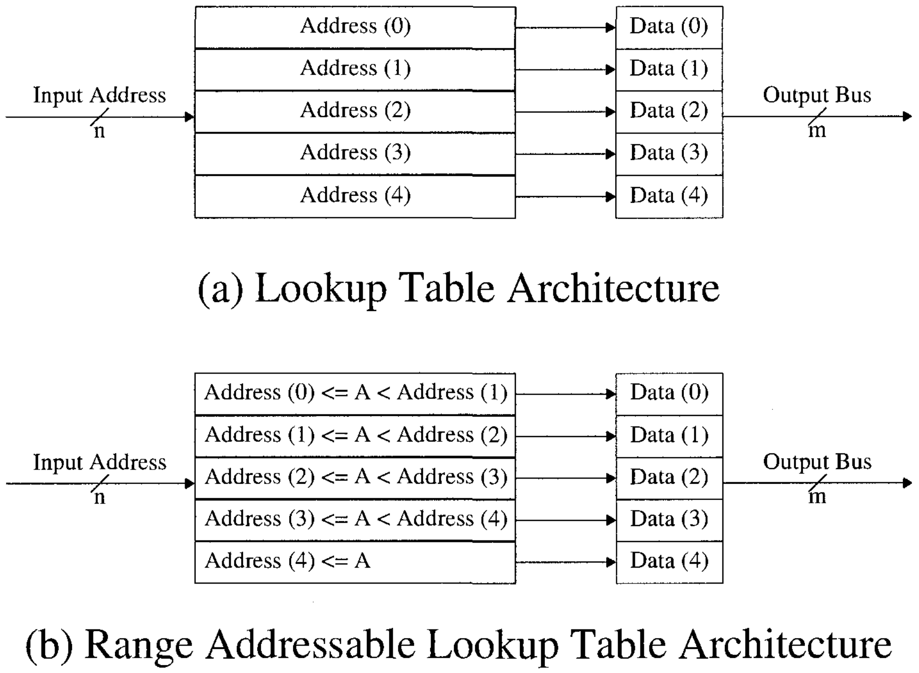

Figure 2.8: RALUT (a) and LUT (b) Architectures

linear and/or discontinuous functions. It is the RALUT's ability to span a large address space, while only using as many rows as are required that allows it to minimize area utilizaton and optimize speed. Equation 2.3 describes how the RALUT will scale in size with the design parameters n, m, and k.

$$RALUT_{Area} \; \alpha \; k \times (n + m) \tag{2.3}$$

Chapter 3

The Range Addressable Lookup Table Architecture

This chapter presents a detailed review of the architecture originally proposed in [14]. It begins by giving an overview of the design, and then expands on the individual components of which it is composed.

3.1 RALUT Architecture Overview

The RALUT is composed of two main parts; the address decoder, and the output rows. As shown in Figure 3.1, the RALUT uses three external signals. The n-bit wide input address and clock signal enter the address decoder portion of the architecture, which is responsible for triggering one of the k word lines. The word lines connect to the output rows, placing an output value on the m bit wide RALUT output bus.

3.2 The Address Decoder

At the heart of the RALUT is the address decoder. The address decoder architecture determines which output row to enable. This is performed by comparing the input address bits with the values that are permanently stored into the RALUT's address decoding array. The input address, and

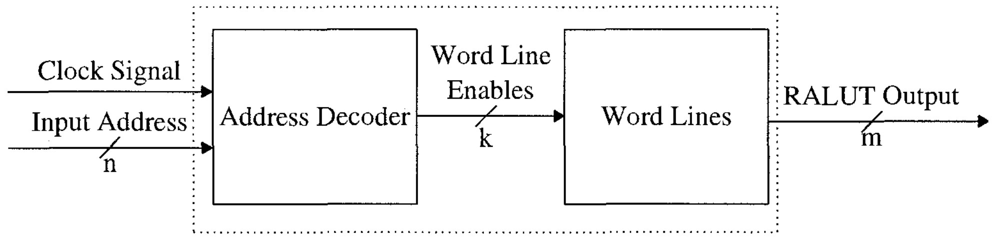

Figure 3.1: Block Diagram of the RALUT, with n Address Bits, m Output Bits, and k Rows

consequently the decoding array, is divided up into groups and compared in stages. This is done to minimize the length of the the domino logic NMOS pulldown network, as long NMOS chains significantly reduce circuit performance. The number of stages used depends on the width of the input address, as well as the number of bits being evaluated by each stage. A block diagram of a five row, five stage RALUT address decoder is shown in Figure 3.2. Omitted for clarity are the input address lines connecting to every row, rather than only the last row as shown in the diagram.

Figure 3.2 also shows the signals emanating from each of the stages. These are used to control the evaluation of subsequent stages. Whenever possible, subsequent stages are prevented from evaluating in order to reduce power consumption due to transistor switching. There are two ways in which this is achieved. First, the EQ_out and GT_out signals act as clock signals for subsequent stages by controlling the precharge and evaluate transistors, later evaluation stages in the same row may be disabled. If, for example, EQ_out does not make the logical transition from *logic 0* to *logic 1* in a given *beginning* stage, the subsequent stage's *EQ* circuit will not enter into an evaluation mode. The second technique employed to limit power consumption is the use of feedback from other rows. By having every row (except the last) use feedback from the next immediate row in the form of the nGT_out_comp signal. If the input address is greater than the stored value in the next, higher-addressed, row, it stands to reason that the input address must be greater than the current row, and that fully evaluating this entire row is redundant. By preventing as much redundant evaluation as possible, transistor switching and thus power consumption is reduced.

It is important to note that only the first stage of the address decoder is driven by the clock. Additional stages are driven by the EQ_out and GT_out signals, which will be further explained in the example at the end of this chapter.

3.2.1 Overview of the Beginning Stage

A block diagram of the *beginning* stage is shown in Figure 3.3.

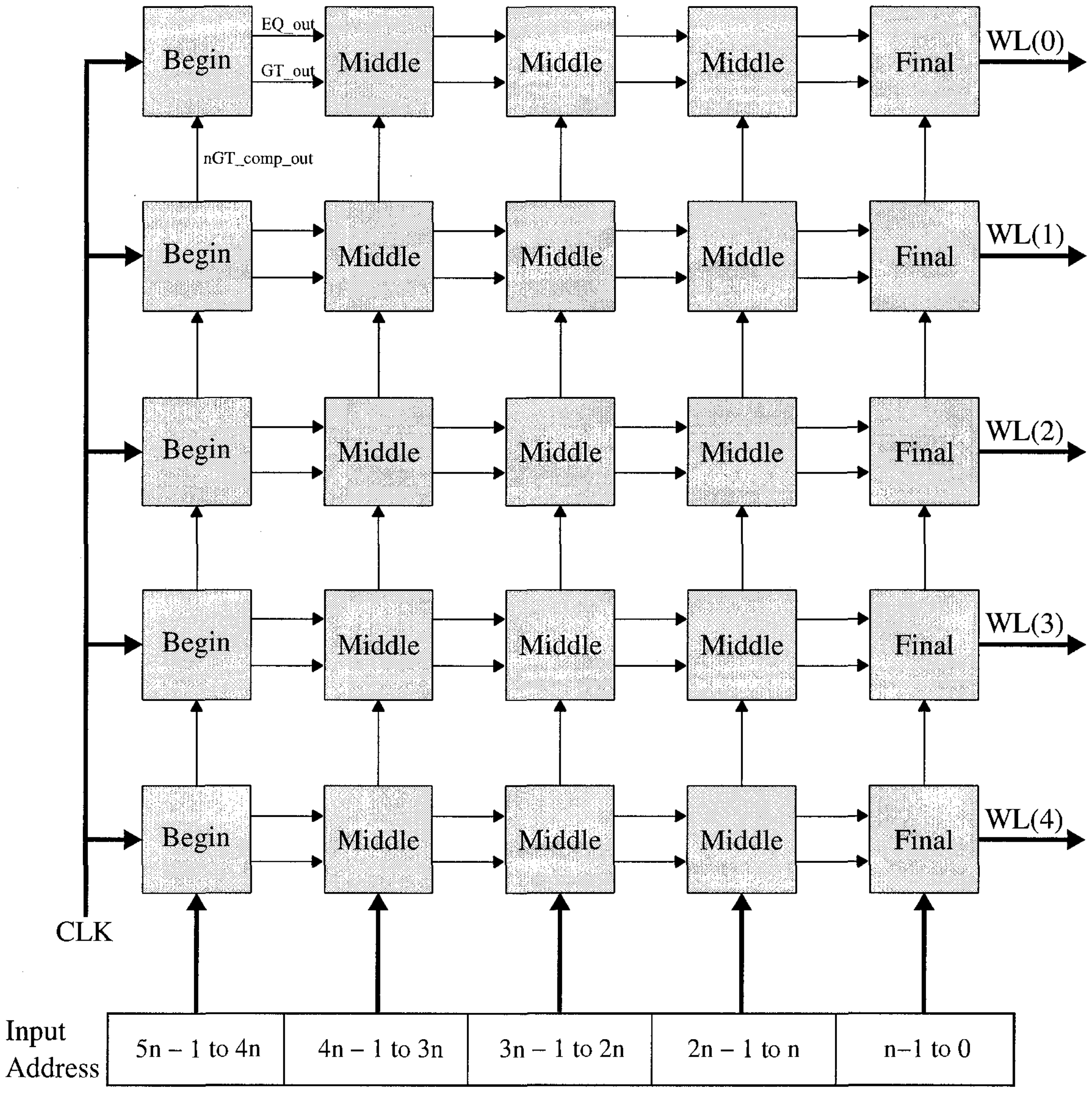

Figure 3.2: Block Diagram of a Five Row, Five Stage RALUT Address Decoder

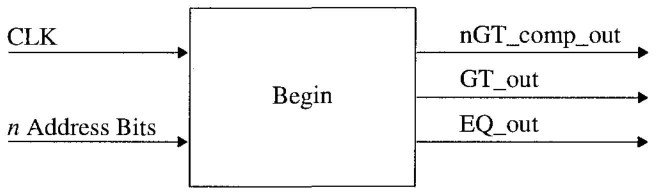

Figure 3.3: Block Diagram of the Beginning Stage

The most significant n bits of the address are passed to every row of this stage. For every row, the *beginning* stage computes if the address bits being compared are greater than, or equal to its stored value. It then continues on to generate the following signals depending on how the circuit evaluated.

- EQ_out evalues to *logic 1* if the input address is exactly equal to the value stored in that particular stage, and *logic 0* otherwise
- GT_out evaluates to *logic 1* if the input address is greater than the stage's stored value, and *logic 0* otherwise
- nGT_out_comp is simply the complement of GT_out

Note that the *beginning* stage is the only stage in this architecture to be driven by the clock signal. Looking closer into the *beginning* architecture, shown in Figure 3.4, is the transistor-level design, showing that the circuit is essentially divided into two parts; one to evaluate the EQ_out signal, and the other to evaluate the GT_out and nGT_comp_out signals.

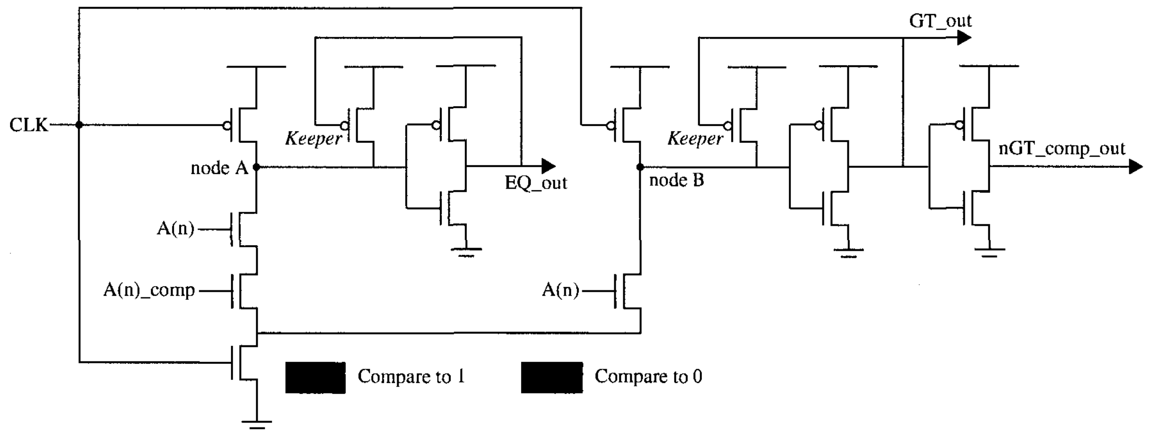

Figure 3.4: Schematic of the *Begin* Stage

For the *beginning* stage, both of these sub-circuits have a very similar, standard domino-logic gate style architecture. The only major difference is the additional inverter added after the GT_out signal to generate nGT_comp_out. When the clock is low, these circuits precharge the critical nodes A and B, meaning the outputs of this stage will be $EQ\_out = logic0$, $GT\_out = logic0$, and $nGT\_comp\_out = logic1$ during this time period. As time elapses, and the clock rises to *logic 1*, the direct connections between these nodes and VDD are severed, and the evaluate transistor conducts, opening a path to ground at the end of the pull-down network. If the input address is equal to the value that this *beginning* stage compares to, a direct path to ground exists for node A, discharging it, bringing the output of EQ_out to *logic 1*. Similarly, if the input address is greater than the compare value, node B discharges, setting GT_out high and nGT_comp_out low.

The way in which EQ and GT are evaluated depends principally on the pull-down network. In Figure 3.4, the pull-down network shows what combination of transistors are used for comparing a bit of the address to '0', and '1' (shown in green and red, respectively). The pull-down network is explained further in the next section.

3.2.2 The Address Compare Pull-Down Network

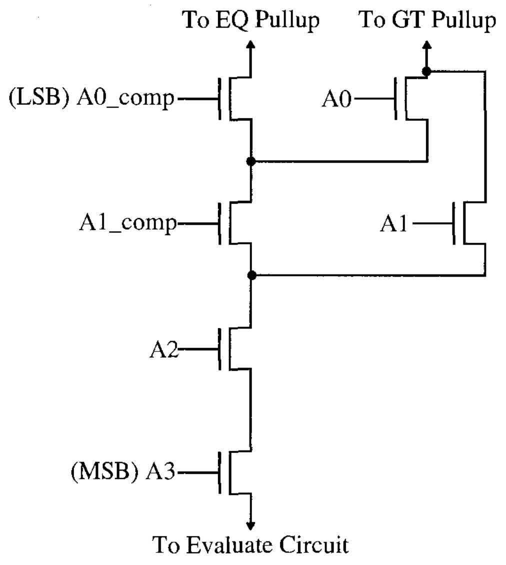

Figure 3.5: An Example Pull-Down Network for a 4-Bit Address Decode Stage Comparing to "1100"

An example of a 4-bit pull-down network used to evaluate the EQ and GT signals is shown in Figure 3.5. Note that the most significant address bit that will be compared is $A3$, the transistor closest to the evaluate circuit, while the least significant address bits connect to the pull-up circuits. In the case of the *beginning* stage, the EQ pullup is node A, and the GT pullup is node B. Also note that when comparing the input address to a '1', there is no transistor in the GT chain. This is due to the fact that in a binary number system, it is impossible to determine if a number is greater than one using a single digit, rather the next, more significant bit must be examined.

3.2.3 Overview of the Middle Stage

A block diagram showing the input and output signals of the *middle* stage is shown in Figure 3.6. The major differences between this and the *beginning* stage are that this stage uses EQ_in and GT_in in place of the clock signal, and nGT_comp_in used in the evaluate chain.

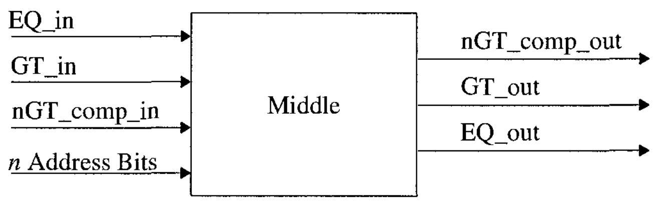

Figure 3.6: Block Diagram of the Middle Stage

The schematic for the *middle* stage is shown in Figure 3.7. Once again, this stage differs only slightly from *beginning*. The EQ circuit uses in_EQ from the previous stage as a clock, and an additional transistor, controlled by in_nGT_comp, is added to the evaluate path. This extra transistor is responsible for disabling the evaluate stage of the EQ circuit in the event that the input address is greater than the next row's value. The GT circuit uses both the in_EQ and in_GT signals to work the precharge and evaluate transistors. For this part of the *middle* stage, note that neither the EQ or GT circuits will evaluate if the input address is greater than the next row due to the additional transistor in the evaluate path of the pull-down network. Additionally, if the previous stage's out_GT signal is at *logic 1*, and the input address is not yet known to be greater than the next row, the GT_out signal of this stage will automatically propogate due to the additional transistors added to the GT circuit's parallel pull-down network. This is done to further reduce the amount of switching in order to improve power consumption.

3.2.4 Overview of the Final Stage

The *final* stage is shown in block diagram form in Figure 3.8. This stage makes use of EQ_in, GT_in, and nGT_comp_in, however its only output is a word line enable signal, WL.

The schematic for the *final* stage is shown in Figure 3.9. This circuit is essentially the same as the *middle* stage, with the exception that the GT and EQ subcircuits have been combined. The reason for this is that if the input address is not yet known to be greater than the next row's compare value, this stage must determine if it is to be enable its row's output word line.

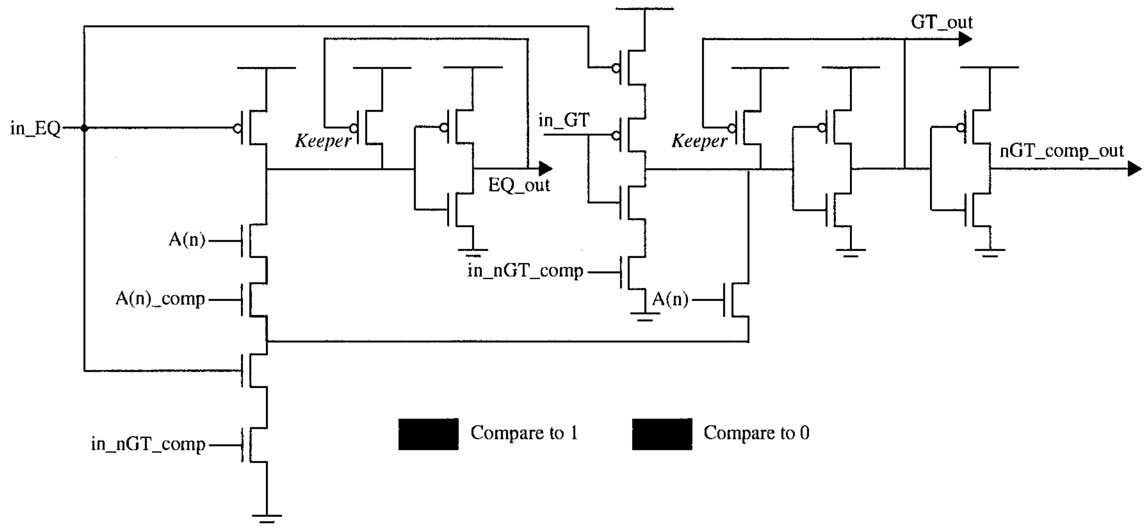

Figure 3.7: Schematic of the *Middle* Stage

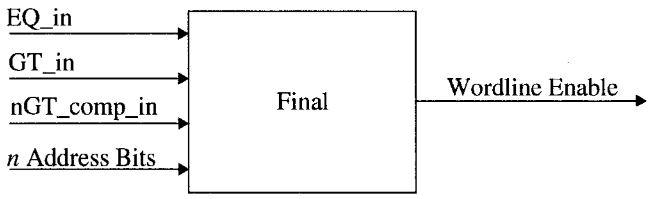

Figure 3.8: Block Diagram of the Final Stage

3.2.5 Detailed Example of the RALUT Address Decoder

This example will refer extensively to the five row, four stage, 12-bit RALUT address decode circuit shown in Figure 3.10. Omitted for clarity are the input address lines going to every row, rather than only the last row. Also omitted are the address and clock buffers. This figure is colour coded to indicate weather a signal is *logic 1* (green), *logic 0*, (red), and if a stage evaluates (green), or if it is disabled to save power (grey).

This RALUT address decoder can enable one of five different word line output rows, shown as WL(0) through WL(4). Evaluation begins as follows. When the CLK signal is at *logic 0*, the *beginning* stage enters its pre-charge state. During this time, the input address may change without affecting the RALUT's output. On the rising edge of CLK, the *beginning* stage of the circuit begins to evaluate.

The most significant three bits of the input address are compared with the *beginning* stage of each row. If the result is equal, as in rows 3 and 4, EQ_out changes from *logic 0* to *logic 1*, acting as

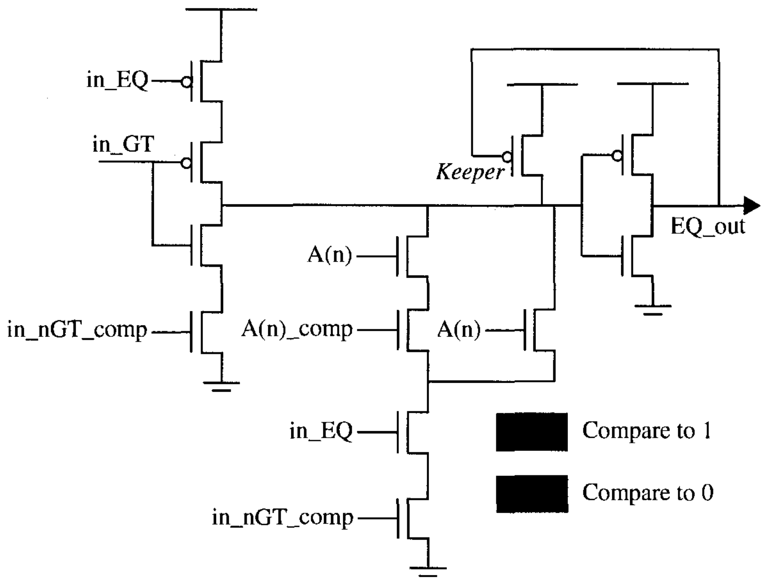

Figure 3.9: Schematic of the Final Stage

clock signal for the next stage's EQ_out circuit. If the result is not equal however, EQ_out does not change logic levels and the next stage's EQ_out circuit remains dormant, saving power. Similarly, GT_out evaluates to *logic 1* if the input address is greater than the stored value. This signal, in turn, acts as the clock for the next stage's GT_out circuit, similar to EQ_out. The last row's *beginning* stage's stored value is greater than the first three bits of the input address. Due to this, both the EQ_out and GT_out lines remain low, and the remainder of the final row does not evaluate, saving power.

The nGT_comp_out signal connects to the *beginning* stage of the previous row. This signal is simply the complement of that *beginning* stage's GT_out, and is used to disable the evaluation of the previous row in order to reduce power consumption. In this example, once the first three bits of the input address have been evaluated by the *beginning* stages, it is apparent that the first row will not require further evaluation since the input address is greater than the second row.

The *middle* stages perform similar to the *beginning* stage, with two differences. First, the *middle* stages are not attached to a clock signal, rather they employ the previous stage's EQ_out and GT_out as a pre-charge and evaluate mechanism. Second, the *middle* stages have been modified to accept an additional input, the nGT_comp_out signal, to disable their evaluate chains when it is at *logic 0*.

Once the previous stages have evaluated, the *final* stage determines which of the word lines to enable. Note that this architecture will, regardless of input addresses and stored decoder values,

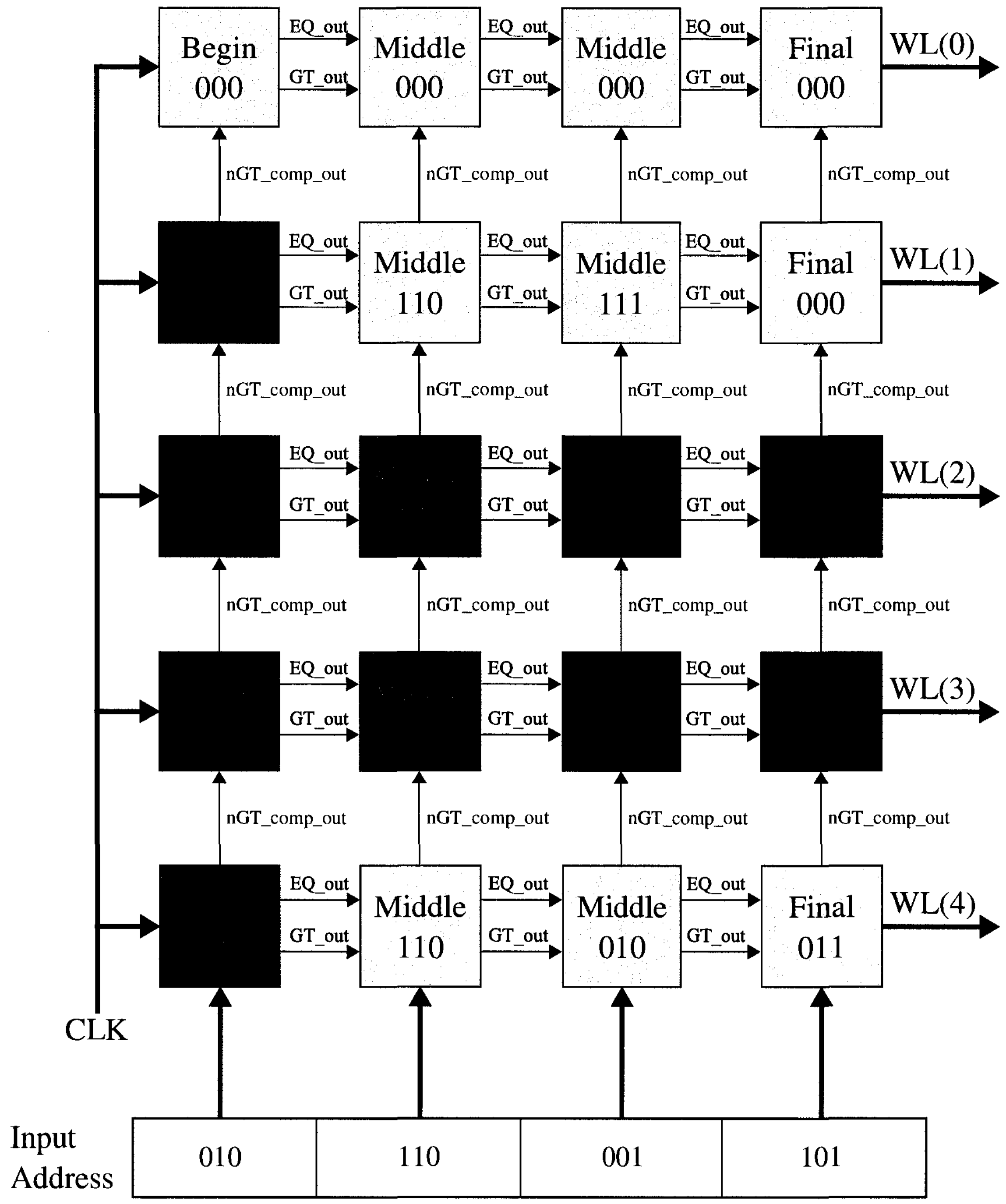

Figure 3.10: Example of a 5 Row, 12-bit, 4 Stage RALUT Address Decoder Evaluating

fully evalute two entire rows at the most. In this example, rows 3 and 4 are evaluated to the end. The *final* stage will compare the final, and least significant, three bits of the input address with its stored value, as well as the previous stage's EQ_out and GT_out signals. Based on these, a single output word line is enabled. In this example, the input address is larger than the value stored in the third row of the address decoder, but smaller than the fourth, and the third word line is enabled. If the fourth row's *final* stage would have had the bit pattern 101 stored, the row would have been exactly equal to the input address, and that world line would have been enabled instead.

Once all stages have evaluated, the word line remains valid until the negative edge of the clock signal, CLK.

3.3 Overview of the Word Lines

The word lines are simple in function; given a line enable signal, they simply place the correct output bit pattern on the output bus. The line enable signal connects to a series of buffer, or line drivers, which then connect directly to NMOS and PMOS transistors which either pull-up or pull-down the RALUT output bits depending how they have been configured.

3.4 Address and Clock Buffering Overview

The clock signal and input address lines must be sufficiently buffered so that as the RALUT scales in size, these signals can be driven without incident. For example, without buffering, the incoming clock signal would have to drive every *beginning* stage. With smaller designs this might be acceptable, however when using a design with hundreds of rows the rise and fall times of the clock signal will be very high, if the signal is able to even drive the circuit at all.

Buffering is implemented with a simple tree structure. Every input signal enter a single buffer, which branches off to a series of additional buffers, and so on, until the signal reaches the address decoder.

Chapter 4

Proposed VLSI Implementations in CMOS $0.18\mu m$

This chapter discusses the design goals, methodology, and results in creating two proposed designs, both of which are in CMOS $0.18\mu m$. The first is a rescaling of the existing $0.35\mu m$ design, in which all layout cells were recreated in the more advanced $0.18\mu m$ node, however only two different transistor sizes were used: one for NMÓS transistors, and the other for PMOS transistors. This was a very rapid approach to rescaling the design, and was used to meet a fabrication deadline for the test platform outlined in the next chapter. The second proposed design involves carefully resizing individual transistors, and further reducing area utilization to produce a high-performance RALUT. This approach proved to be much more time consuming, however simulation results prove to be optimal.

This chapter is organized as follows. It begins with an overview of the CMOS $0.35\mu m$ design, followed by an explanation why the CMOS $0.18\mu m$ technology node was chosen for the new designs. A brief discussion regarding the rescaling of CMOS designs ensues. Next, detailed explanations of the proposed CMOS $0.18\mu m$ and high performance $0.18\mu m$ designs are presented. Finally, the chapter ends with a comparison of the results and some summarizing remarks.

4.1 Existing CMOS 0.35μm Design

This work advances the contributions made in [14] towards a high performance, full-custom RALUT design. As such, an existing full-custom design in an CMOS 0.35μm process existed, however many improvements could be made. The existing CMOS 0.35μm design consisted of the following items:

1. A full-custom cell library, including beginning, middle, and final stages of the address decoder, as well as the input and clock buffers, output bits and output linedriver cells

2. A CAD tool designed in SKILL, used in the Cadence software environment to automatically place and configure the design cells based on a user-generated file containing the desired bit patterns

While this work consists of a solid base, many improvements were possible. Originally used in 1995, CMOS 0.35μm is a dated technology. Many modern processors are currently designed with 90nm technology, and as of 2007 Intel has been fabricating some of their ICs using a 45nm process. Clearly, it is advantageous to advance the RALUT design to a more recent technology node, increasing its utility. In addition to porting to a more recent technology, the RALUT can be further optimized by carefully resizing its transistors. The CMOS 0.35μm design uses two different transistor sizes: one for PMOS and one for NMOS transistors. While this may greatly simplify layout design, it does not yield optimal performance.

4.1.1 Selecting an Updated Technology Node

The term "technology node" refers to a generation of process technology used to fabricate integrated circuit chips. The name of the node itself refers to the smallest possible transistor channel width that can be fabricated with that process. CMOS 0.18μm, for example, allows the creation of transistors with a minimum channel width of 0.18μm. As new fabrication techniques are discovered, the creation of smaller devices is possible, enabling faster operating speeds and reduced power consumption.

Currently, several different technology nodes are available for researchers to fabricate devices, including CMOS 0.35μm, 0.18μm, 0.13μm, and 90nm. When selecting which design technology to implement, and later fabricate the RALUT architecture, the following considerations were made:

- The fabrication technology design kit must be made available to the University of Windsor through CMC

- In addition to the analog design tools, the process' design kit must include standard library cells for digital designs

- It is preferable to use a mature design kit in which designs have been successfully fabricated in the past
- Assuming the previous criteria are met, the most recent process technology should be selected to ensure a high-performance design that compares well with current competing design alternatives

The University of Windsor currently has the 0.35μm, 0.18μm, 0.13μm, 90nm, and 65nm CMOS design kits, while fabrication services made available from CMC. Of these, the 0.35μm, 0.18μm, and 90nm kits have digital standard cell libraries. The 90nm design kit is currently considered quite "bleeding edge", and at the time of this writing, the kit is incomplete; it lacks several important elements such as timing libraries for the standard cells, which are crucial for timing-driven placement and routing. With CMOS 0.18μm and 0.35μm to choose from, 0.18μm was selected. CMOS 0.18μm was first used in 2000; it is a proven process, and significantly more recent than the 0.35μm node, which was first available in 1995.

4.2 Design Rescaling

Every fabrication technology possesses a set of design rules that, among other things, define the minimum distances that must separate certain layout elements to ensure that the integrated circuit can be fabricated. [CMC's cmosp18/cmosp35 documents] Unfortunately, the majority of these design rules do not scale simply with the technology. For example, as shown in Table 4.1, in advancing to CMOS 0.18 from CMOS 0.35, the minimum transistor widths and lengths do not scale at the same rate [26], [25]. Due to these uneven scaling factors, a full-custom layout cannot be simply rescaled when advancing to a newer technology node.

Technology Node	CMOS 0.35μm	CMOS 0.18μm	Scaling Factor
Transistor Length	0.35μm	0.18μm	0.51
Transistor Width	0.40μm	0.22μm	0.55

Table 4.1: Transistor Length and Width Scaling Factors

Maintaining the same transistor width-to-length ratio is also insufficient when rescaling the design, as the supply voltage also changes from 3.3V to 1.8V from CMOS 0.35μm to 0.18μm. For these reasons, rescaling the design is not simply a matter of shrinking the design cells. Rather, every cell must be redrawn by hand to ensure that the design rules are adhered to while maintaining a compact and efficient design.

4.3 Proposed CMOS $0.18\mu m$ Implementation

The CMOS $0.18\mu m$ implementation consists of rescaling the existing CMOS $0.35\mu m$ design to the CMOS $0.18\mu m$ process. All of the design cells must be redrawn and rescaled according to the CMOS $0.18\mu m$ design rules. Similar to the CMOS $0.35\mu m$ design, NMOS and PMOS transistors are both given a width parameter, allowing these broad categories of transistors to be easily resized. While it is not ideal to use the same size for all transistors of a particular type, it simplifies the layout, and reduces design time. It was highly desirable to fabricate an integrated circuit to test the RALUT; due to this a shortened design cycle was important, as only four months were available from the commencement of this work until the fabrication deadline.

4.3.1 Transistor Sizing for the CMOS $0.18\mu m$ Implementation

In order to determine transistor sizing for this design, a parametric analysis was performed, and the NMOS/PMOS transistor widths which provided optimal results were selected. Transistor lengths were all set to 0.18 μm, the minimum channel length allowable for this technology node. In this case, using a PMOS width of 0.6 μm, and an NMOS width of 0.39 μm provided the best results. RALUT parameters such as the number of bits per address decode stage, number of output bits per linedriver, and the maximum number of rows per buffer were all configured to be the same as in the existing CMOS $0.35\mu m$ design. Implementation results for this proposed design are shown at the end of this chapter in Section 4.5.

4.4 The High Performance CMOS $0.18\mu m$ Implementation

Once the $0.35\mu m$ RALUT was rescaled to the $0.18\mu m$ process, and the test IC sent off for fabrication, work continued on further improving the $0.18\mu m$ version. This proposed design will be referred to as the "high performance CMOS $0.18\mu m$ implementation". This section details the work done to create this high performance design.

4.4.1 High Performance CMOS 0.18μm Implementation Design Goals

Carefully sizing individual transistors should be able to increase operating speeds without dramatically affecting area utilization and layout complexity. The effects of proper keeper transistor sizing, and the pull-down network chain scaling should also be investigated to determine what performance gains can be made. Additionally, optimal design parameters for the maximum number of address bits per decode stage, amount of address buffering, and the number of output bits per linedriver are not known for the CMOS 0.18μm design. It is worthwhile to determine ideal values for these parameters to maximize performance.

To summarize, the design goals of the high performance implmentation are as follows:

1. Optimize the transistor sizing for the address decode stages, clock and address buffers, output bits, and output linedrivers

2. Determine ideal design parameters for the number of address bits per address decode stage, ideal amount of input address and clock buffering, and the maximum number of output bits per linedriver circuit

3. Report simulation performance data to serve as a guide for future hardware designers

4. Redraw cell layouts, making any possible area and performance optimizations

4.4.2 Transistor Channel Length

In digital circuit design, the transistor length, or channel length, is typically set to the minimum size allowed by the fabrication process —in this case 0.18 μ m. This is done to maximize the transistor's conduction current, which is governed by Equation 4.1 for NMOS devices and 4.2 for PMOS devices. This equation describes the transistor's maximum current drive ($I_{D,max}$) in terms of the transistor's dimensions, width (W) and length (L), a process-specific constant, the gate oxide capacitance (C_{ox}), the gate-to-source and threshold voltages (V_{GS} and V_{TH}), and either the hole or electron drift velocity, μ_n and μ_p, respectively. As shown in the equations, reducing L will increase $I_{D,max}$, resulting in, using qualitative terms, a "stronger" transistor. In short, using smaller channel lengths will result in smaller channel widths for the same conducting current, while reducing the total amount of charge that must be displaced. This reduces area utilization and power consumption, while increasing the operating speed.

$$I_{D,max} = \frac{1}{2}\mu_n C_{ox}\frac{W}{L}(V_{GS} - V_{TH})^2 \tag{4.1}$$

$$I_{D,max} = \frac{1}{2}\mu_p C_{ox}\frac{W}{L}(V_{GS} - V_{TH})^2 \tag{4.2}$$

4.4.3 Keeper Widths

Keeper transistors are used to minimize the effect of charge leakage. The keeper must be correctly sized to ensure that the critical node remains at *logic 1* when it is charged, however it must not overpower the node if it is legitimately attempting to discharge during the evaluation phase. A keeper sizing scheme was described in [20], and was used as a starting point. To size the keeper in this way, the NMOS $\frac{W}{L}$ aspect ratios in the pull-down network are summed, and multiplied by a constant less than one. This constant is then experimented with until simulation results prove optimal. For this work, the keeper was computed using this approach, and then tuned to yield optimal results. Although different keeper sizes were considered for use in the various address decode stages of the RALUT, simulation results indicated a negligeable difference. Different keeper sizes were also tested when using 4, 5, and 6 input bits per stage. Once again, operating speeds among the different keeper sizes were negligeable. Due to the minimal performance gains in sizing the keepers differently depending on the number of address bits, and among the different address decode stages, the same keeper width of 250nm was used throughout the design to simplify the layouts.

4.4.4 NMOS Chain Scaling

NMOS chain scaling is a circuit design technique employed to improve speed performance in domino logic [18]. It consists of sizing each of the transistors in the NMOS pull-down network such that the transistors closest to the critical node are smaller, while the transistors closer to the ground connection get larger. The reason for this is that when the domino logic gate enters evaluation mode, and a valid path to ground exists via the pull-down network, the charge from the transistor closest to the critical node must pass through the next transistor in the pull-down network, and the charge from both of those must past through the next, and so on. Thus the last transistor in the chain must conduct all the charge from the ones before it, and as such, modest performance increases can be expected if the chain is resized in this way.

Chain scaling was simulated on the schematic level for this design, and it was determined that a only a negligeable performance increase was possible. Unfortunately, the added layout complexity, in addition to the increased area utilization, outweighed the benefits of the increased operating speed. The final design does not make use of chain scaling for these reasons.

4.4.5 Transistor Widths

With many of the transistor dimensions determined as the previous sections explained, relatively few transistor widths need to be determined. At this point it is possible to perform a parametric analysis to heuristically determine transistor sizing. This approach consists of running a simulation circuit for many different combinations of transistor widths, and selecting the best results. Many iterations are repeated, each time resizing different sets of transistors, until circuit-wide performance is maximized.

The test circuits used during these simulations are described in the following section, while results are presented at the end of this chapter.

4.4.6 High Performance CMOS 0.18μm RALUT Test Circuits

In order to determine the ideal transistor sizing, test circuits were developed to load each of the address decode stages appropriately, and to test performance with a variety of bit patterns.

Address Compare Bits

Each address decode stage can compare its fraction of the input address to any of 2^n bit patterns, where n is the number of bits per address decode stage. It is impractical to exhaustively test and analyze every bit pattern for every address decode stage. A more reasonable approach is to determine the worst-case bit pattern or patterns, and then to use those when evaluating performance. This is an acceptable alternative, as the most important measure of speed performance is the maximum delay, rather than the average.

Three different address compare values will be considered when evaluating the decode stage delay: all ones, all zeroes, and the most significant bit set to one, with the remaining bits set to zero, as shown in Figure 4.1. The first of these, all ones, is expected to be a best-case scenario. With only one transistor directly attached to the critical node, a minimum amount of charge sharing is present in this configuration. Next, the 'all zero' configuration has been selected as a test pattern to determine performance when there is a maximal amount of charge sharing at the critical node. With this configuration every transistor in the GT evaluate path connects directly to the critical node and a very large amount of charge is being shared as the number of compare bits increases.

Finally, when this same pattern is used, with the most significant compare bit changed to a '1', the worst performance is usually observed. When this compare value is given an input address of all ones, every '0' transistor conducts, and all this charge must pass through the single '1' transistor along the evaluate path.

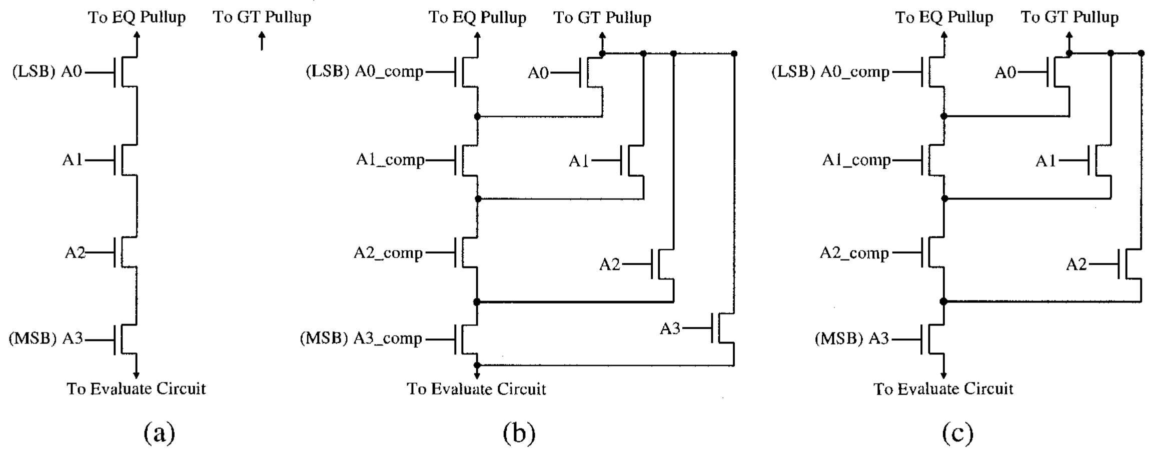

Figure 4.1: Pull-Down Networks Used in Test Circuits: (a) '1111', (b) '0000', (c) '1000'

Beginning Stage Test Circuit

In the *beginning* stage test circuit, a single *beginning* stage is attached to a set of ideal address inputs, as well as an ideal clock signal. The stage's outputs are appropriately loaded with two *middle* stages, the EQ_out line attaches to the in_EQ port on the *middle* stage, while the nGT_comp_out of the same *beginning* stages connects to a second *middle* stage's nGT_comp_in port. This was done such that the *beginning* stage could be simulated under typical loads.

This test circuit determines the latency of a single *beginning* stage by changing the input address while the clock is low and the circuit is in its precharge stage, and then measuring the time between the rising edge of the clock, and the change (if any) in each the stage's three outputs. Once the clock returns to *logic 0*, the input is changed, and this cycle continues. This is repeated for every one of the possible 2^n input combinations, where n is the number of that stage's compare bits, to ensure that the circuit behaves properly (provides the correct results) under all input conditions. A sample simulation waveform for the *beginning* stage with the 6-bit address compare value of '100000' is shown in Figure 4.2, in Section 4.4.7, followed by a table summarizing the results.

Middle Stage Test Circuit

Similar to the previous stage's test circuit, the *middle* stage test circuit is loaded with two additional *middle* stages, and driven by ideal inputs. Two seperate simulations were run in order to determine this stage's performance; one in which the EQ signal is changing, and the second where the GT signal is changing. This is done to determine the performance of the *middle* stage's delay when either of its EQ or GT subcircuits evaluate.

Final Stage Test Circuit

This test circuit is once again driven by ideal inputs, however it is loaded with two inverters connected in series to the word-line enable output. Similar to the *middle* stage test circuit, it is simulated in two separate runs, one using the EQ signal, the other using GT, in order to isolate and optimize the stage's delay for both subcircuits.

Buffer Test Circuit

The buffer test circuit differs from the address decode test circuits in that it is much more simple, as it only needs to drive other buffers in the buffer tree, as well as the input address lines going to the address compare bits. The following criteria had to be determined in order to achieve an optimal buffer design:

1. Buffer transistor sizing
2. Number of stages per buffer
3. Optimal buffer loading

The first of these goals is relatively easy to determine, a buffer is nothing more than an even number of inverters connected in parallel, meaning very few transistors exist. A parametric analysis quickly reveals which transistor combinations perform well. The number of stages per buffer refers to the number of inverters connected together to form the buffer. More inverters are better suited to drive larger loads, at the expense of increased area and delay. Finally, optimal buffer loading is simply the drive capability of the buffer, or in other words, the number of circuits that it can drive. As more buffers are used, the area utilization increases significantly, rendering the buffer loading parameter very important in the efficient implementation of the RALUT in hardware.

The test circuit to determine these parameters is effective and simple. First, a one-stage buffer was used, and its output was connected in parallel to the inputs of several inverters. A multi-dimensional parametric analysis followed, in which the number of inverters was varied along with the buffer transistor widths. This strategy allowed the ideal transistor width and the optimal amount of loading with relative ease. Once ideal parameters were determined for a single-stage buffer, the experiment was repeated with a two-stage buffer, in order to determine its performance characteristics.

Simulation waveforms and test circuit results are presented and discussed in Section 4.4.7.

Linedriver Test Circuit

The linedriver is similar in function to the buffer, except that it is exclusively used to drive output bits and one additional linedriver stage, rather than the input address lines. The linedriver test circuit consists of two *final* decode stages which are configured to enable their word lines one after another as the input address increments. One of the output lines consists of all ones, or all PMOS transistors, while the second output row consists of all NMOS transistors. This is done to test the buffers with maximal loading and charge sharing. Results of this simulation, in additition to a discussion of ideal number of output bits per linedriver stage, are presented in Section 4.4.7.

4.4.7 High Performance Implementation Test Circuit Results and Final Transistor Sizing

Simulation Environment

Currently, there are several different SPICE tools available, the most popular of which are Avanti HSPICE, Cadence Spectre, Mentor Eldo, and Silvaco SmartSpice. HSPICE and Spectre are available for use, and both were tested for use in this work. Results from each of these tools were typically within less than a perecent of each other. HSPICE typically evaluated faster, however for certain circuits, it experienced difficulty in converging to a solution. Spectre, on the other hand, performed better in this aspect, and few, if any, convergence aids were required to compute simulation results. Additionally, Spectre is better integrated with the other Cadence tools, such as Analog Environment, as they are both developed by the same company. For these reasons, Spectre was used almost exclusively throughout this work, and all of the reported results are from this netlist simulator.

Measurements

Measurements shown in the following tables were determined as follows:

- Rise time is measured as the time elapsed between 10% and 90% of the maximum voltage, in other words, the time required for a signal to raise from 0.18V to 1.62V
- Fall time is measured between 90% and 10%, or 1.62V and 0.18V
- Propagation Delay is measured as the time between the 50% value (0.9V) of the input signal and the 50% (0.9V) value of the output signal

Beginning Stage Test Circuit Results and Discussion

Pictured in Figure 4.2 are the *beginning* stage simulation waveforms.

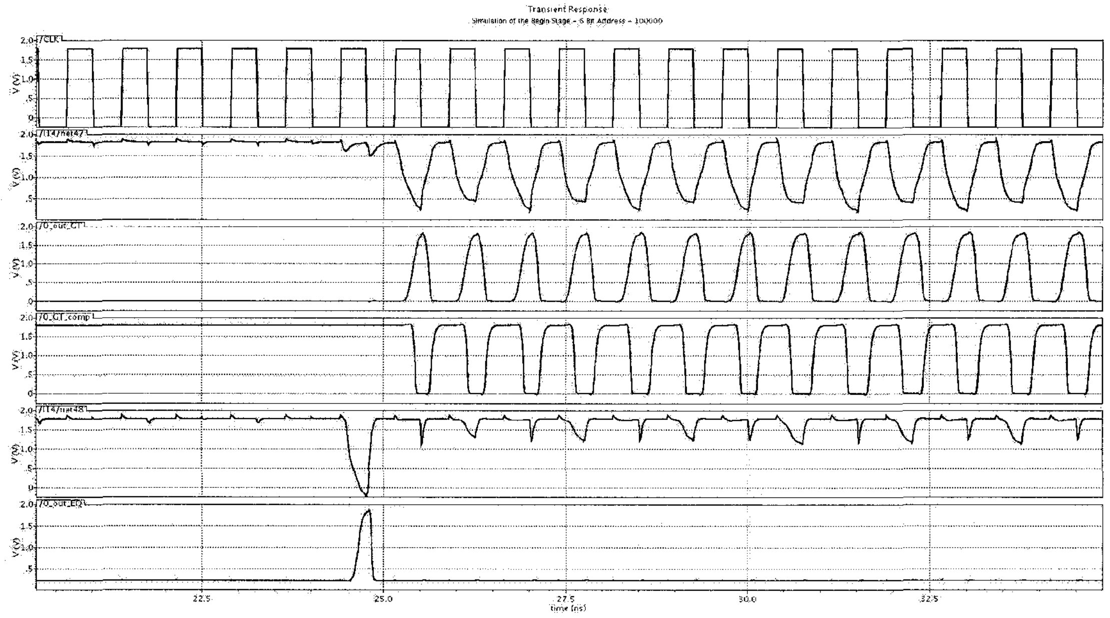

Figure 4.2: Simulation Waveforms for the Beginning Address Decode Stage. From top to bottom: clock signal, GT Critical Node, GT_out, GT_out_comp, EQ Critical Node, EQ_out

In this figure, from top to bottom, the signals are: The clock, GT circuit critical node, GT_out, GT_out_comp, the EQ circuit critical node, and EQ_out. Also note that for clarity, the input address signals are not shown.

These waveforms demonstrate that the properly sized *beginning* stage operates correctly at very high clock frequencies; in this case the clock is set to approximately $1.3GHz$. Although the addresses are not explicitly shown, it is easy to tell when the input address is '100000', as the EQ_out signal raises to *logic 1*. The EQ circuit critical node shows some signs of charge sharing problems. Even though a keeper transistor is used, its diminuitive size causes issues at such high operating speeds. Unfortunately, using a larger keeper will negatively impact the critical node's fall time, resulting in a greater worst-case delay for this stage. The sizing used in this situation is shown to function correctly at very high operating speeds using an address chain length of 6 bits.

Table 4.2 displays the complete results for every *beginning* stage simulation. Signal delay, rise and fall times were recorded for the *beginning* stage when using 4, 5, and 6 input address bits, and with the bit patterns described in section 4.4.6.

Bits	Value	EQ_out			GT_out			GT_comp_out		
		Delay	Rise	Fall	Delay	Rise	Fall	Delay	Rise	Fall
4	0000	252	105	65	187	138	70	165	75	43
	1000	230	107	67	233	135	69	199	73	45
	1111	209	100	66	-	-	-	-	-	-
5	00000	259	127	77	191	130	70	164	82	51
	10000	279	115	88	252	125	75	234	81	51
	11111	203	123	69	-	-	-	-	-	-
6	000000	298	120	71	323	150	75	293	77	45
	100000	334	122	74	304	141	73	281	77	47
	111111	207	115	62	-	-	-	-	-	-

Table 4.2: Beginning Stage Test Circuit Simulation Results in Picoseconds (ps)

In order to determine the ideal number of address bits to use per stage, the worst-case delay of the stage must be determined every time a different number of address bits are used. As shown in the table, the worst case delay for the 4-address-bit *beginning* stage arises when the bit pattern '0000' is used, however for both the 5, and 6-bit stages, the worst case bit pattern is '10000' and '100000', respectively. The worst case delay for these stages are shown in Table 4.3, along with the delay per address bit.

Rise and fall times remain relatively unaffected throughout the various simulation results, with the exception of the GT_out signal when the bit pattern '000000' is used. In this case, the large amount of charge sharing at the critical node begins to cause the GT_out rise time to suffer, requiring 150ps to rise from 0.18V to 1.62V. A brief simulation using a wider pull-down network determiend that the rise time increases faster as the width of the pull-down network grows. It is for this reason that a maximum of 6 address bits per *beginning* stage was considered.

It is desirable to compare as many address bits as possible per address decode stage in order

Bit Pattern	Worst Case Delay	Delay Per Bit
0000	252	63
10000	279	55.8
100000	334	55.7

Table 4.3: Summary of Beginning Stage Worst Case Delay and Delay Per Address Bit, Results in Picoseconds (ps)

to reduce area utilization. However, operating speed is of critical importance, and as such it is preferable to minimize the delay per address bit. For these reasons, the *beginning* stages employed in this design will make use of six address bits per stage, and cost approximately 55.7 picoseconds per decoded bit.

Final transistor sizing for the *beginning* stage is shown in Appendix A.

Middle Stage Test Circuit Results and Discussion

Figures 4.3 and 4.4 show sample waveforms for a 6 input address *middle* stage comparing against the bit pattern '100000'. The first figure shows the performance of the EQ circuit, while the second presents waveforms when the GT circuit of the *middle* stage is in operation.

In both figures, from top to bottom, the waveforms are: clock, the GT critical node, GT_out, GT_out_comp, the EQ critical node, and EQ_out. In this case, the *middle* stage is simulating at 1.1 GHz, and there is little noticeable effect due to charge sharing in either of the critical nodes. A complete summary of delay, rise time, and fall time for the *middle* stage is shown in Table 4.4, for various combinations of bit patterns and input address lengths.

This stage's worst-case delay for both the 4-bit and 6-bit addresses occurs when '1000' and '100000', respectively trigger the GT_out signal when the stage is being driven by the EQ_in signal. For the 5-bit address, '10000' also triggers the worst case, except this time it is GT_comp_out when the circuit is driven by EQ_in. These worst-case delays, and the delays per bit, are summarized in Table 4.5

Once the *middle* stage's transistors were properly sized, the best delay per bit was achieved is $53.4ps$. For this stage, the best performance was achieved while using five address bits, however 6 bits also performed comparably with a delay per bit of $55.2ps$. In the interest of minimizing area utilization it is therefore advantageous to use the 6-bit addressing, as fewer *middle* stages will be required in the overall design.

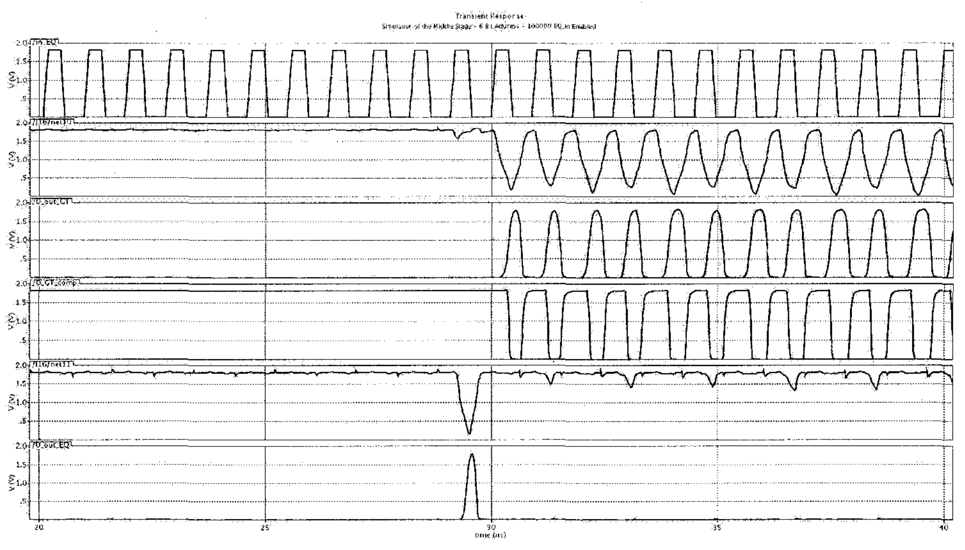

Figure 4.3: Simulation Waveforms for the Middle Address Decode Stage with EQ Enabled. From top to bottom: clock signal, GT Critical Node, GT_out, GT_out_comp, EQ Critical Node, EQ_out

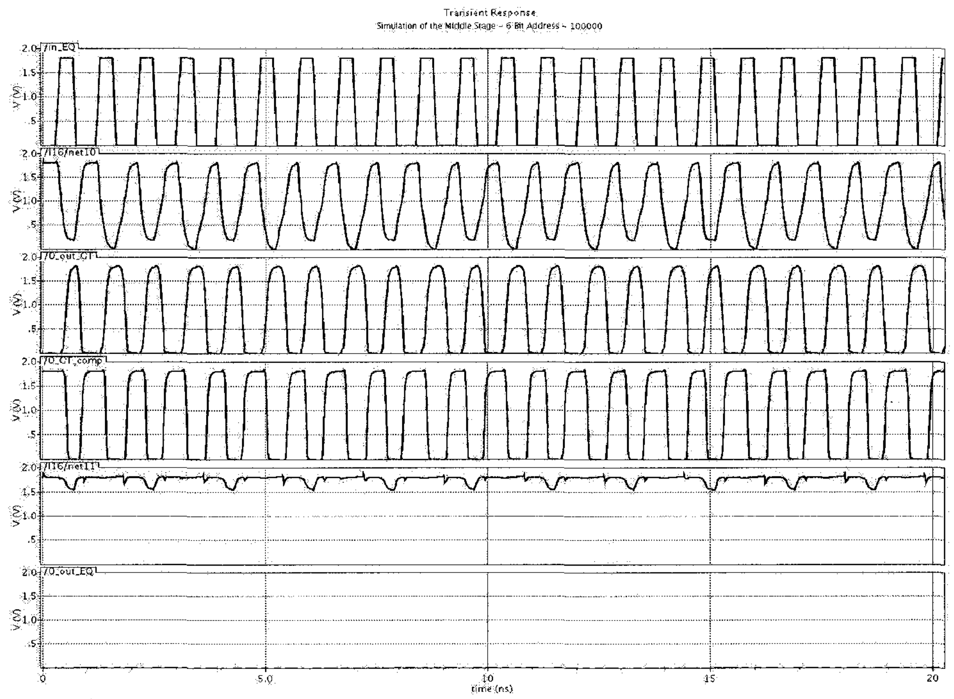

Figure 4.4: Simulation Waveforms for the Middle Address Decode Stage with GT Enabled. From top to bottom: clock signal, GT Critical Node, GT_out, GT_out_comp, EQ Critical Node, EQ_out

Bits	Value	EQ_out			GT_out				GT_comp_out			
		Delay	Rise	Fall	Delay		Rise	Fall	Delay		Rise	Fall
					EQ	GT			EQ	GT		
4	0000	171	114	47	183	191	129	99	159	170	84	48
	1000	190	112	42	244	194	125	94	187	167	84	51
	1111	171	106	47	-	148	107	54	-	126	80	42
5	00000	236	130	56	225	204	135	87	196	163	88	50
	10000	260	120	60	260	194	127	64	267	168	92	44
	11111	192	121	63	-	131	110	54	-	110	76	47
6	000000	298	133	77	318	224	140	93	281	156	84	50
	100000	328	128	78	331	200	136	92	300	184	98	53
	111111	235	131	51	-	134	108	57	-	140	82	44

Table 4.4: Middle Stage Test Circuit Simulation Results in Picoseconds (ps)

Bit Pattern	Worst Case Delay	Delay Per Bit
1000	244	61.0
10000	267	53.4
100000	331	55.2

Table 4.5: Summary of Middle Stage Worst Case Delay and Delay Per Address Bit, Results in Picoseconds (ps)

A schematic with final transistor sizing is shown in Appendix A.

Final Stage Simulation Circuit Results and Discussion

Simulation waveforms for the *final* stage of the address decoder are shown in Figures 4.5 (with EQ_in enabled), and 4.6 (with GT_in enabled). Both sets of waveforms are for circuits which compare the input to the 6-bit pattern '100000'. The *final* stage is simulating at 833 MHz; it is the slowest of the three address decode stages, however the dalay will be shown to be approximately the same as the *middle* stage.

From top to bottom, both figures show the clock signal, and the wordline enable output. Once again, the input address is omitted for clarity; it is incrementing by one with every clock pulse. Figure 4.5 shows that the EQ circuit struggles to pull-up the signal when operating at 833 MHz and above; the first peak is at 1.6V, rather than 1.8V. This voltage level is high enough to properly drive the attached logic, however it at any speeds higher than this it may not. Enlarging the PMOS transistors will allow the *final* stage to work at higher clock speeds, however this will negative impact the delay time. Since the operating speed is still very high, the delay is much more important at this point. For these reasons, the final transistor sizing shown in Appendix A presents optimal results for this stage of the address decoder.

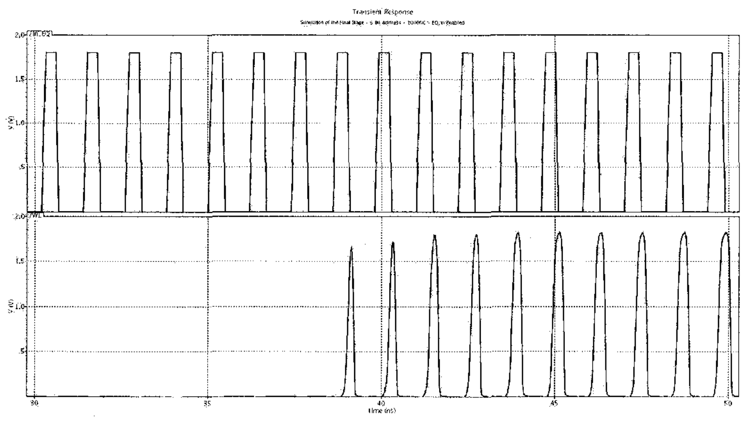

Figure 4.5: Simulation Waveforms for the Final Address Decode Stage with EQ Enabled. From top to bottom: clock signal, GT Critical Node, GT_out, GT_out_comp, EQ Critical Node, EQ_out

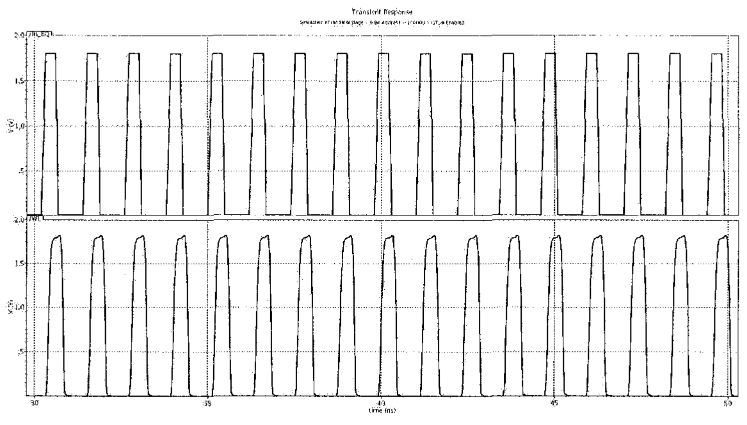

Figure 4.6: Simulation Waveforms for the Final Address Decode Stage with GT Enabled. From top to bottom: clock signal, GT Critical Node, GT_out, GT_out_comp, EQ Critical Node, EQ_out

Worst case delay, rise, and fall times are presented in Table 4.6. As shown, the worst case delay for the all three address bit lengths occurs when the in_EQ signal is driving the decode stage, and the pattern '1000', '10000', or '100000' is compared. All of the results which rely on the in_EQ signal are worse than when the circuit is driven by the in_GT signal. This is due to the additional transistors that must be discharged when in_EQ is driving the circuit; in this case the pull-down network must evaluate, whereas this is not required when in_GT drives this decode stage. Table 4.7 displays a summary of the worst case delay for every address bit length, and the delay per address bit. Once again, 6 address bits proves optimal. The performance gain in going from four to five address bits is a significant reduction of 7.5 ps per address bit, however there is a performance loss of 3 ps per address bit. The *final* address decode stage is not repeated throughout the design like the *middle* stage, so this tiny performance loss is acceptable in the interest of reducing area utilization.

Bits	Value	in_EQ			in_GT		
		Delay	Rise	Fall	Delay	Rise	Fall
4	0000	206	116	77	152	103	74
	1000	221	117	75	164	110	72
	1111	191	113	49	134	91	52
5	00000	220	126	82	150	109	93
	10000	239	132	85	148	105	79
	11111	205	130	75	107	91	51
6	000000	246	126	90	161	111	87
	100000	331	139	85	152	110	80
	111111	263	138	82	110	92	47

Table 4.6: Final Stage Test Circuit Simulation Results

Bit Pattern	Worst Case Delay	Delay Per Bit
1000	221	55.3
10000	239	47.8
100000	305	50.8

Table 4.7: Summary of Final Stage Worst Case Delay and Delay Per Address Bit, Results in Picoseconds (ps)

Buffer Test Circuit Results and Discussion

Results of the buffer test circuit are shown in Table 4.8, and waveforms for one and two stage buffers driving 8 rows is shown in Figures 4.7 and 4.8.

	Rows per Buffer	Rising		Falling	
		A	A_{comp}	A	A_{comp}
One Stage	1	66	32	54	54
	2	77	43	67	71
	4	103	65	88	74
	8	162	103	123	94
	10	206	135	142	109
Two Stages	1	142	109	121	130
	2	156	119	129	136
	4	171	125	135	147
	8	234	163	177	172
	10	270	188	520	201

Table 4.8: Buffer Test Circuit Results in Picoseconds (ps)

Table 4.8 shows that the single-stage buffer is able to adequately drive up to 10 stages, with delay increasing approximately linearly with the number of stages. While this may be true, it is important to consult the simulation waveforms. Figures 4.7 and 4.8 are both driving 8 stages, and although the delay characterisitcs remain acceptable, the waveforms begin to show problems. Rather than stay high the entire time, the level of the buffers' outputs fluctuate significantly, and if noise is introduced into the system, whatever they are attached to may switch errantly. For this reason, it is recommended to use no less than one buffer for every 8 decode stages. Additionally, there is little signal improvement from the single stage to the double stage buffer, thus it is recommended to use the single-stage buffer unless the RALUT is to be implemented in a high-noise environment, or if the additonal area utilization is not a concern.

Linedriver Test Circuit Simulation Results and Discussion

Results of simulating the linedriver circuit with varying lengths of output chains are shown in Table 4.9. As shown in the table, the delay per bit appears to continuously improve as the output chains grow in length until it is approximately 28-30 bits long at which point the improvement narrows. While this seems to imply that a single large linedriver is well-suited to drive every output bit, it is worth investigating the quality of the linedriver signal, as shown in Figure 4.9

Figure 4.9 shows the waveforms of two linedrivers and an output line. One of the linedrivers is

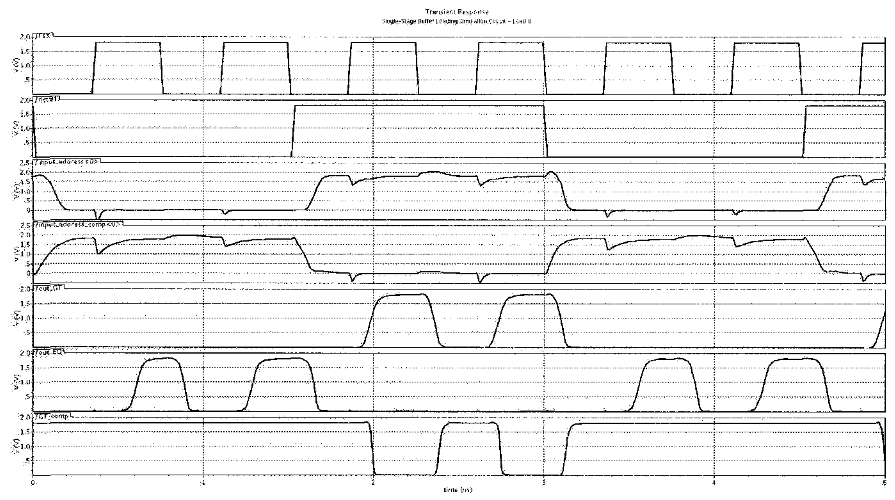

Figure 4.7: Single-Stage Buffer Driving 8 Rows From Top to Bottom: Clock, Buffer Input, Buffer Output, Buffer's Complemented Output, Stage's Output Signals EQ_out, GT_out, and nGT_out_comp

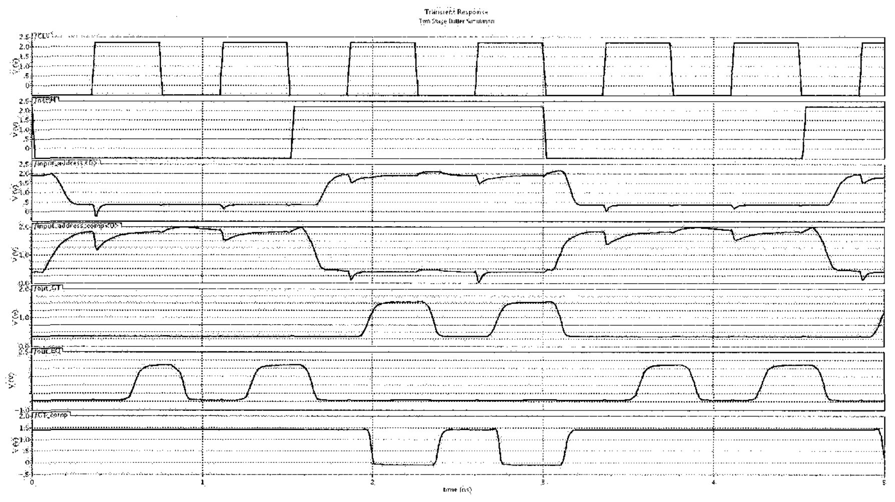

Figure 4.8: Two-Stage Buffer Driving 8 Rows From Top to Bottom: Clock, Buffer Input, Buffer Output, Buffer's Complemented Output, Stage's Output Signals EQ_out, GT_out, and nGT_out_comp

driving all PMOS transistors, while the second is driving all NMOS transistors. It can be seen that when driving 48 output bits, the linedrivers are unable to rise above 1.5V. This is less than 90% of the maximum voltage of 1.8V, and may have the negative effect of not being able to properly drive the output bits. Parasitic capacitance is not taken into account for this simulation, meaning

Chain Length	Positive Transistion Delay	Negative Transition Delay	Delay Per Output Bit
4	90	95	23.8
8	108	118	14.8
12	126	143	11.9
16	142	168	10.5
20	158	195	9.8
24	174	220	9.2
28	189	241	8.6
32	203	265	8.3

Table 4.9: Output Bit Chain Length Test Circuit Simulation Results in Picoseconds (ps)

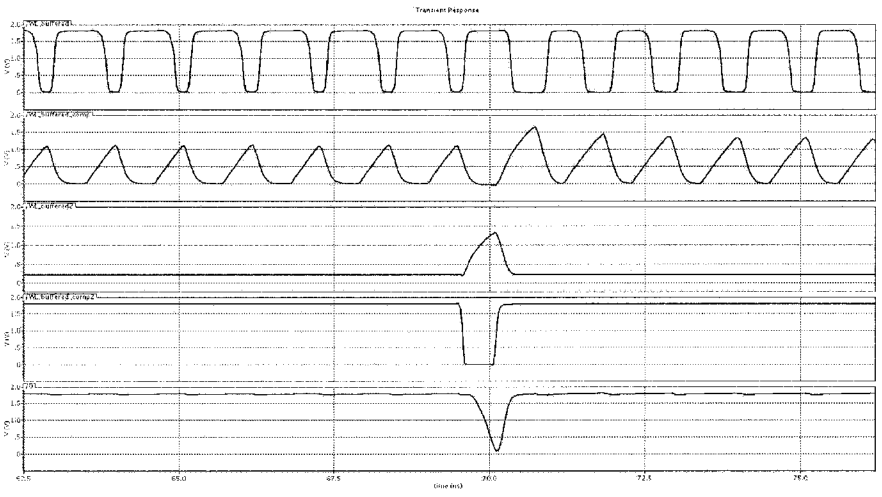

Figure 4.9: Simulation Waveforms for the Linedriver Driving 48 Bits, From Top to Bottom: Row 1 Enable Signal, Row 1 Enable Signal Comp, Row 2 Enable Signal, Row 2 Enable Signal Comp, Sample Output Line

layout results will be far worse. For this reason, it is inadvisable to use more than 16 output bits per linedriver.

4.4.8 Proposed High Performance Design Layout Improvements

After designing the high performance RALUT implementation in CMOS 0.18μm, the layouts were further scrutinized in order to determine what additional performance gains can be made. This section details the work done to further reduce area utilization and improve performance.

Area Reduction Improvements

Several aspects of the RALUT design were carefully examined in order to find ways to reduce area utilization. The most frequently occuring cells were of particular interest, namely the address decode bits, and the output bits. Unfortunately, the address decode bits are already tightly packed, and it was determined that no further optimizations were possible without restructuring the entire design. The output bits, on the other hand yielded some improvement.

It was possible to pack these layouts closer together, reducing the RALUT width. In order to do this, it was necessary to push two bits together into a single layout, and modify the placement CAD tool to work correctly with this change. The original output bit had two possible combinations, it could either be a 'zero' or a 'one'. The modified output bits, being pushed together are now either a 'zero zero', 'zero one', 'one zero' or 'one one'. The output bits of the CMOS $0.18\mu m$ design were $1.5\mu m$ wide, and after this modification they are $2.25/2 = 1.125$, representing a width reduction of 25%. Since the output bits repeat many times, this translates into a large area savings when many output bits are used.

Another major area reduction improvement was achieved in superimposing the power and ground rails over the RALUT design, rather than having them isolated at the top and bottom. This presented many signal routing difficulties, however in the end it was possible to reduce cell height for all design cells to 3.685 from 5.46, representing a cell height reduction of 33%.

A final area reduction was possible in increasing the fanout of the buffer tree. The existing CAD tool had a parameter which specified the maximum number of rows per buffer row, and created a buffer tree structure based on this information. The initial buffer would split into two additional buffers, and so on. This was revised so that rather than splitting into two additional buffers, it will split into n buffers. Given the performance data gathered in the previous chapter, using a fan out of 8 rather than 2 is acceptable, and will greatly reduce area utilization.

Reduced Parasitic Capacitance

Parasitic capacitance is the term given to the unwanted or undesirable capacitance in a circuit that is often a result of components being placed closely together [8]. These capacitances are unwanted because they increase the charge capacity at various nodes in the circuit; a node with greater charge capacity will require more time to charge and discharge, negatively affecting circuit performance. To appreciate the effect they have on a circuit, simulations should be performed with, and without, parasitic capacitances. The sensitivity of domino logic designs accentuates their effect on performance; a well designed domino-logic cell will often suffer a 15-25% maximum operating speed decrease once parasitic capacitances have been considered.

It is useful to understand how parasitic capacitances arise in IC design; they principally come from closely placed wires, or overlapping wires on different metal layers. When arranged in either of these configurations, the wires essentially form parallel plate capacitors. This type of capacitance can be approximately modeled by Equation 4.3, where A is the overlapping area of the parallel plates, d is their separation distance, k is a process constant, and ϵ_0 is the free space permittivity constant.

$$C = \frac{k\epsilon_0 \times A}{d} \tag{4.3}$$

In order to reduce parasitic capacitances, several approaches were taken. By examining Equation 4.3, it is apparent that either the area of the capacitor must be reduced, or the distance increased. For intersecting wires on different metal layers, the separation distance can only be changed by changing the metal layers the wires reside on. This is possible, however it may create other problems as more vias are required, and the total wire length increases. For adjacent wires on the same metal layer, however, simply spacing them further apart whenever possible will greatly reduce the capacitance. This technique was used with little success; unfortunately the cramped nature of the cells left almost no room to space routing wires apart more than they already were.

Optimizing the other parameter, the overlapping plate area, can also greatly reduce parasitic capacitances, and proved to be more advantageous. Reducing wire sizes to the minimum allowable widths, and shifting them apart as much as possible proved to be very effective. This is illustrated in Figure 4.10, where two metals, shown in blue and yellow, overlap. In part (b), the overlapping area is greatly reduced. During layout optimization this was carried out wherever possible by minimizing wire widths. Finally, in part (c), the ideal situation is shown, where the metal layers no longer overlap. Once again, this was also implemented wherever possible, however the level of optimization in (b) was typically more feasible.

4.5 Results of the CMOS $0.18\mu m$ and High Performance CMOS $0.18\mu m$ Designs

The $0.35\mu m$ design, as well as the proposed CMOS $0.18\mu m$ design layouts are shown in Figure 4.11. It presents a comparison of a 29 row, 16-bit input, 52-bit output RALUT design in CMOS $0.35\mu m$,

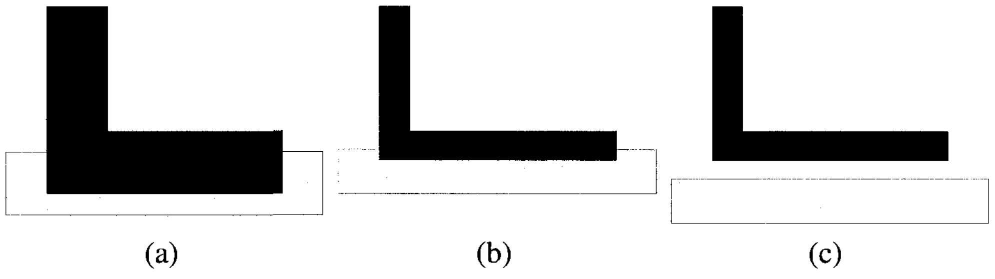

Figure 4.10: Overlapping Wires Creating Parasitic Capacitances: (a) Original Placement, (b) Reduced Overlap Area, (c) Ideal Placement

CMOS 0.18μm, and the high performance CMOS 0.18μm design. The exact area reduction and speed results are shown in Table 4.10.

Design	Width	Height	Area	Delay	Area × Delay
CMOS 0.35μm[17]	420 μm	260 μm	68460 μm^2	4.45 ns	3.06×10^{-10}
Proposed CMOS 0.18μm	240 μm	163 μm	39120 μm^2	2.70 ns	1.06×10^{-10}
Proposed High-Performance CMOS 0.18μm	210 μm	126 μm	26460 μm^2	1.8 ns	4.76×10^{-9}

Table 4.10: Area and Critical Path Delay Comparison for a 16-bit Input, 52-bit Output, 29 Row RALUT

Table 4.10 displays the width, height, area, and delay of the 16-bit input, 52-bit output, and 29 row RALUT. Using *Area* × *Delay* as a performance metric, Table 4.10 shows that the CMOS 0.18μm design is 65.34% more efficient than the CMOS 0.35μm design presented in [17]. It is also shown that the proposed high performance CMOS 0.18μm design is 84.44% more efficient than the design in [17]. This is a significant improvement in terms of both area utilization and delay.

4.6 Summary

The CMOS 0.18μm technology node was selected to design a rescaled version of the existing CMOS 0.35μm design, as well as a high performance implementation of the RALUT. The rescaled version was quickly designed, allowing for it to be included in test IC in time for fabrication, while offering 65% better results than the 0.35μm design. The high-performance design demonstrates excellent results with an 84% improvement over the CMOS 0.35μm design.

The superior results achieved in the high performance design were made possible due to the

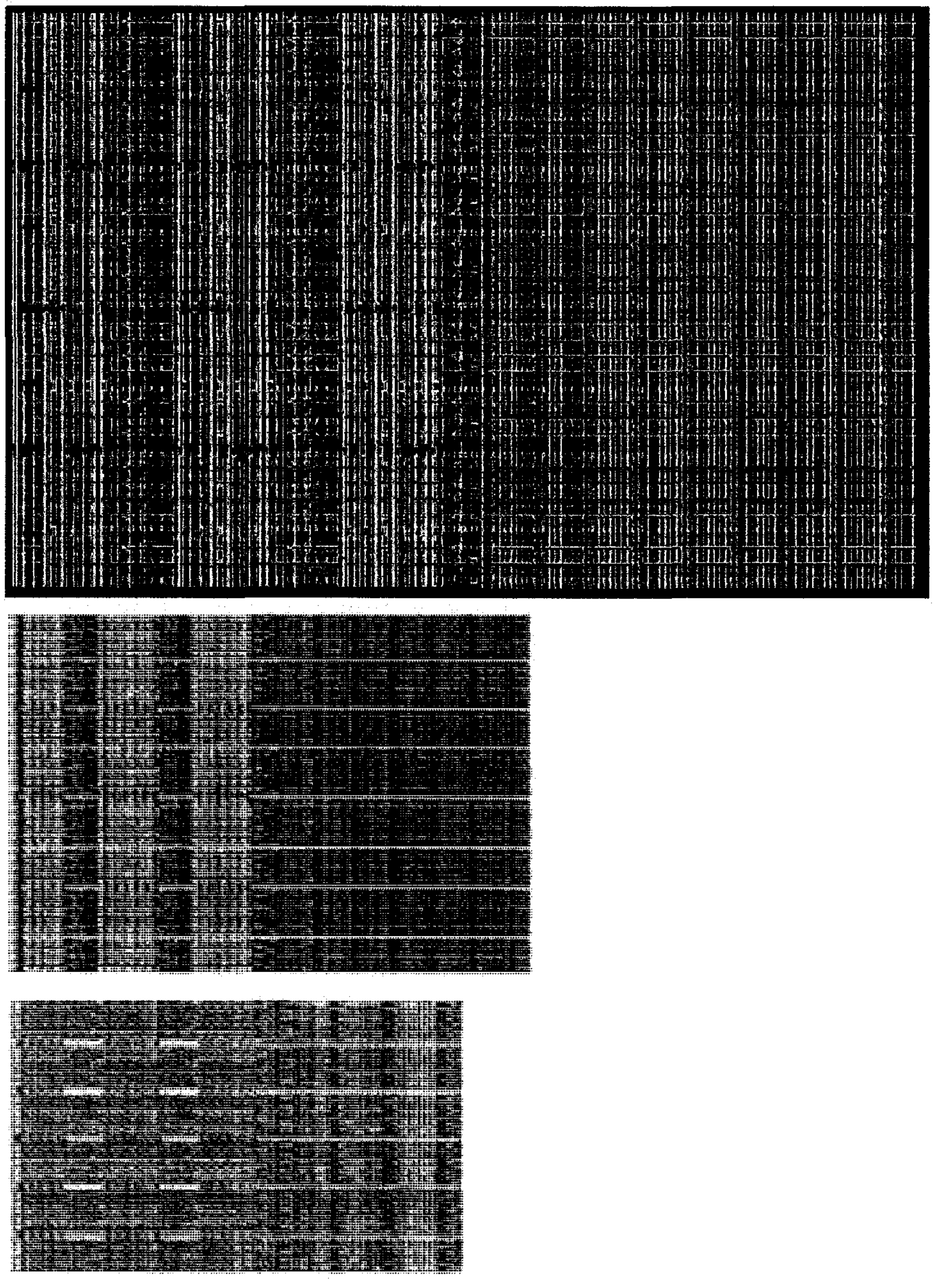

Figure 4.11: RALUT Layout Comparison for CMOS $0.35\mu m$ design (top), CMOS $0.18\mu m$ design (middle), and area-reduced, high-performance CMOS $0.18\mu m$ design (bottom)

properly sized transistors and optimal RALUT design parameters. The ideal number of address bits per decode stage was determined to be 6, every linedriver should drive no more than 16 output bits, and no fewer than one buffer should be used for every 8 rows of the design.

Also, due to the performance data collected, it is now possible to estimate the delay of the high performance RALUT design. With knowledge of the delay per bit for every address decode stage, in addition to the delay for the output bits, this can be calculated as in Equation 4.4.

$$Delay \approx 55.7 \times B_{bits} + 55.2 \times M_{bits} + 50.8 \times F_{bits} + 20.5 \times I_{bits} + 10.5 \times O_{bits}(ps) \tag{4.4}$$

Where B_{bits} is the number of address bits in the *beginning* stage (up to 6), M_{bits} is the number of bits in the *middle* stage, F_{bits} is the number of bits in the final stage, I_{bits} is the number of total input bits, and O_{bits} is the number of output bits. It is important to note that this is only an approximation of the delay, however it will give designers an excellent basis when considering the use of the high performance RALUT design.

Chapter 5

Integrated Circuit Test Platform Design

While simulations are critical to the succes of any design, it is always preferable to rely on physical test data. Currently, no such information exists for the RALUT. Additionally, many things are difficult to account for, such as switching noise, sensitivity to temperature and process variation, as well as overlooked design flaws and limitations. While critical path delay and power consumption can be approximated via simulation, there is no guarantee that this will be the actual case given a physical manifestation of the design. Physical test data would further prove the utility of the RALUT, as well as provide hardware designers with realistic performance expectations.

To achieve this, an integrated circuit chip is proposed to test the design in real-world conditions.

The main goal of the proposed RALUT test IC is to determine maximum operating speed and the power consumption of the RALUT. Additionally, it is desirable to compare the full-custom domino logic implementation of the RALUT with a semi-custom HDL implementation of the same design. To achieve this, the following system was designed.

This chapter begins with an explanation of the general testing strategy that the IC employs, followed by an overview of the major components. Subsystems are fully detailed, and the IC layout, cell placement, and routing is explained. The chapter concludes with several figures of the complete test IC, and some concluding remarks.

5.1 Test IC Overview and Testing Strategy

RALUTs are designed to operate at very high speeds. Although the RALUT critical path delay depends heavily on the number of input and output bits, as well as the number of rows, when a typical RALUT design is implemented in CMOS $0.18\mu m$ the delay is expected to be approximately two to ten nanoseconds. Unfortunately, due to pin capacitance, it is not possible to communicate with the IC beyond approximately 50 MHz, as it takes a finite amount of time to charge and discharge the device's pins. This creates some interesting challenges when designing an IC to determine the design's maximum operating speed. Primarily, it is not possible to supply the IC with an external clock frequency greater than 40-50 MHz. Additionally, input addresses cannot be sent, nor can the device's output be verified at such speeds.

Keeping these design constraints in mind, the following test IC is proposed. The test IC will consist of four major components pictured in Figure 5.1.

1. The clock controller circuit
2. The control unit
3. The test circuit
4. The output select circuit

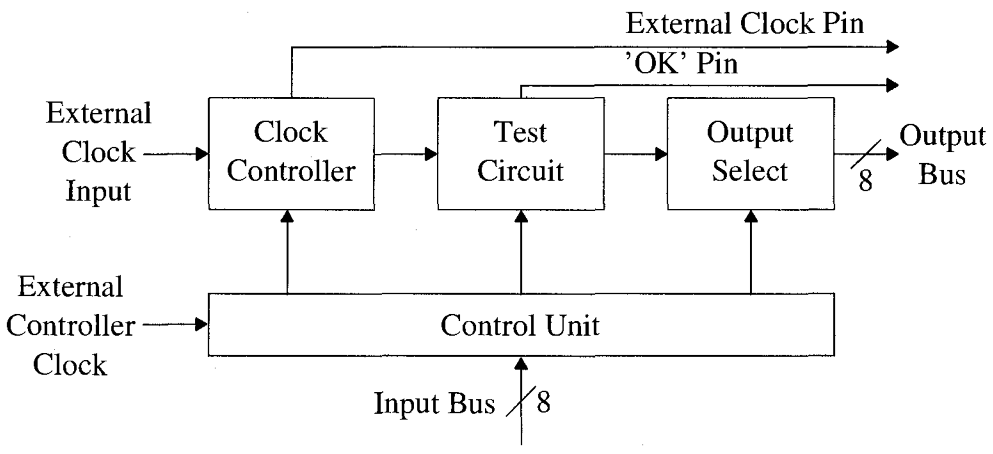

Figure 5.1: Block Diagram of the IC Subsystems

5.2 The Clock Controller Circuit

The clock signal is extremely important in any synchronous logic system. The proposed test IC provides a variety of clock modes to allow testing under a variety of conditions. A block diagram of the clock controller circuit is shown in Figure 5.2.

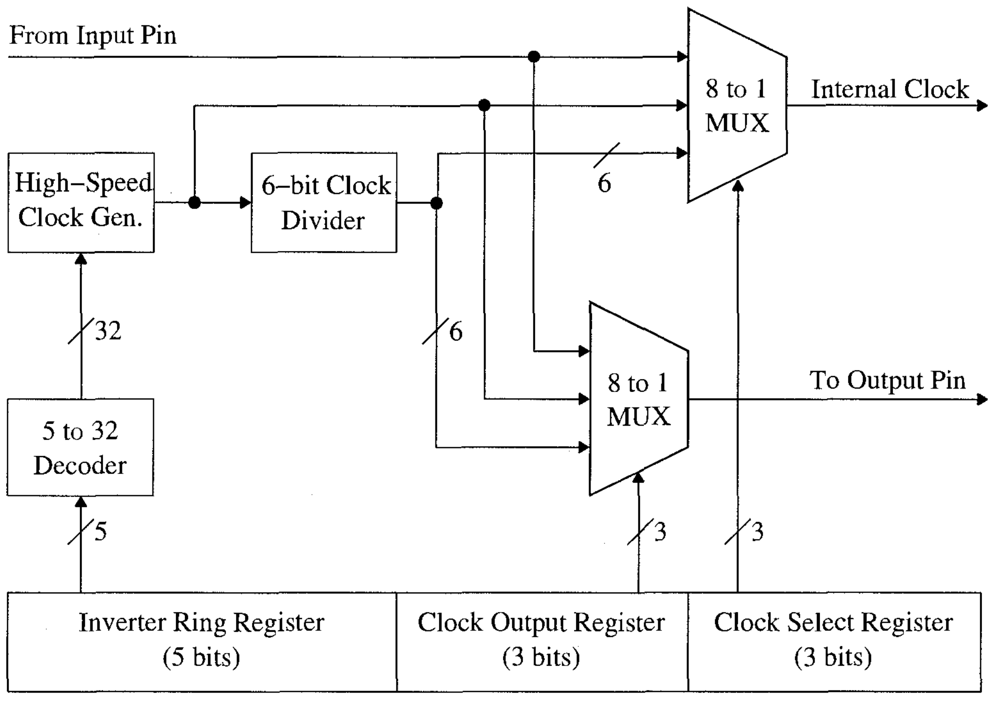

Figure 5.2: The Clock Selection Circuit Block Diagram

A series of registers in the control unit provide the control signals which dictate the functionality of the clock select circuit. A 5-bit clock generator control signal enters a 5-to-32 decoder, which enables one of the clock generator's modes. The 6-bit clock divider is an up counter circuit used to divide the frequency of the clock generator by 2, 4, 8, 16, 32, and 64. The external clock pin's signal, as well as high-speed clock generator and it's frequency-reduced signals, are connected to a set of 8-to-1 multiplexers. This allows for the selection of an internal clock signal, as well as an external clock signal; both sets of MUX selection lines are connected to the control unit's registers. The internal signal is used to drive all of the IC's test-circuit sequential logic including the RALUT design, while the external clock is connected to an output pin to allow for clock feedback.

In addition to allowing the high-frequency clock generator's output to be more carefully controlled for interal IC use, the clock divider may be used to scale the frequency low enough such that it be properly monitored on the output pin. For example, if the clock generator is configured to generate a clock signal of $350MHz$, the frequency can be divided by 16, reducing it to approximately $22MHz$, which is a low enough frequency for the output pin to handle. This feedback will allow the exact operating frequency of the clock generator to be measured.

5.3 Internal High-Speed Clock Generation Circuit

Since it is not possible to provide an externally driven clock signal greater than 40-50 Mhz, in order to test the RALUT design at higher operating speeds, an on-chip clock must be available. Two solutions were explored, phase-locked loops and inverter rings.

5.3.1 Phase-Locked Loops

A phase-locked loop (PLL) is one of the most common methods used to generate high-frequency internal clock signals. In short, PLLs multiply the frequency of a reference clock to generate a higher-frequency internal clock signal. A properly designed PLL is able to reliably generate whatever clock frequencies are required by the designer. The circuits used to create a PLL are relatively complex, and require careful analog circuit design. A literature review on PLLs quickly revealed that their design is an entire topic all on its own, with entire textbooks dedicated to them. Due to time constraints, it is not feasible to implement a PLL in the proposed design.

5.3.2 Inverter Ring

A much simpler alternative is the inverter ring. The schematic for an inverter ring is shown in Figure 5.3. It consists of an odd number of digital CMOS inverters, connected together in a ring, such that the output of the last inverter connects to the input of the first, creating an oscillator.

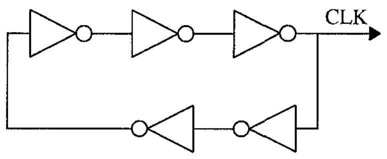

Figure 5.3: A 5-Stage Inverter Ring

While this is a simple solution, it does introduce its own set of challenges. Due to the large amount of switching, the inverter ring will consume a large amount of power. The inverter ring does not possess any feedback mechanism; process and temperature variations will affect the clock frequency, causing its real-world behaviour to diverge significantly from simulation results.

Despite these inconveniences, the inverter ring will be used, as it is much more feasible than designing a PLL given design-time constraints.

5.3.3 Inverter Ring Design

An inverter ring is used in the test IC for high-speed clock signal generation. It is an important design goal for the test IC to be able to verify correct operation of the chip at a variety of different speeds. This issue can either be resolved by creating a series of different inverter rings with varying amounts of delay, or by creating an inverter ring with selectable delay. The latter is the preferred solution, as multiple inverter rings will occupy a much larger area, particularly for low frequency designs which will require a larger number of inverters.

A block diagram of the proposed selectable-delay inverter ring is presented in Figure 5.4. It consists of two main sub-components: the *delay* block and the *switch*. The delay block, consisting of a group of serially connected inverters as in Figure 5.5, will be broken up by a series of switches, which are able to divert the output of a given delay block to either the next delay block in series, or to a return path, completing the 'ring' and connecting to the first delay block. This will allow for the run-time selection of the number of delay stages.

Figure 5.6 will be referred to, in order to further explain the functionality of the delay-select scheme. In this case, the series of four selection lines are given the control word "0010". This causes the first, second, and last switches to forward their inputs the the next delay stage in the chain. The third switch, on the other hand, has its select line enabled, which causes its input to drive the return line, connecting with the first delay block, completing the ring. In this case, a total of three delay blocks will contribute to the clock signal's delay.

It is also important to note that this inverter ring design will require one-hot encoding on its select lines, meaning that only one select signal should be at *logic 1* at any given time. For this reason, it is recommended to control the select lines of the proposed inverter ring design with a decoder. In addition to ensuring the one-hot condition, it will also reduce the amount of I/O required to control this design element.

The Switch Block

This sub-section presents more information on the design of the *switch* block used in 5.4. In order to achieve a simple switch structure, transmission gates (also known as T-gates) were used [13]. A

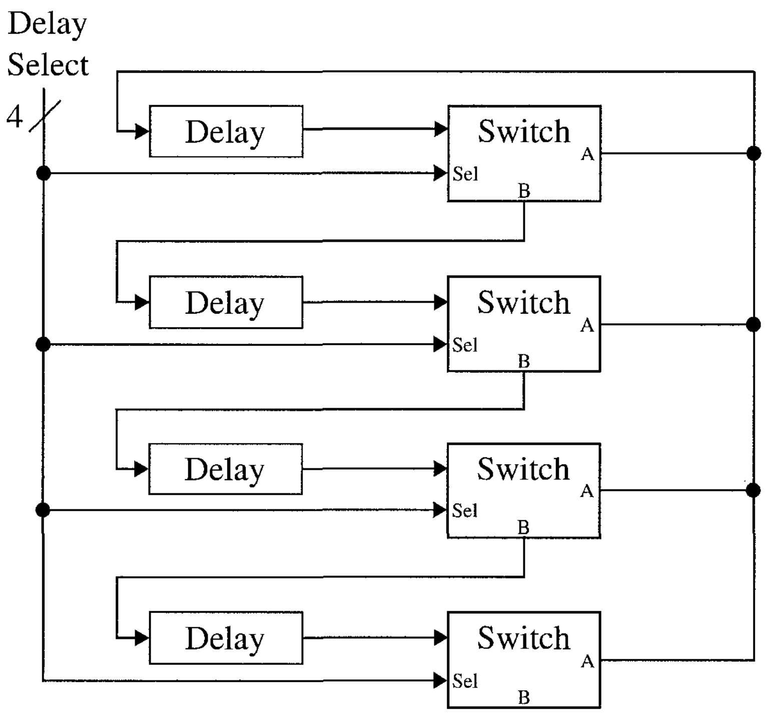

Figure 5.4: Inverter Ring with Four Delay Settings

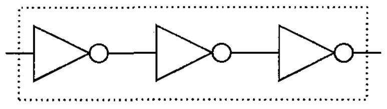

Figure 5.5: A Three-Inverter Delay Block

T-gate schematic is presented in Figure 5.7. This is a simple pass-gate which allows bidirectional signal propogation given that the transistors are conducting. In the case of Figure 5.8, this condition is met when signal A is driven to *logic 1*.

While T-gates only serve to control if a signal is to drive an output or not, two T-gates along with an inverter can be used to create a switch, as in Figure 5.7. The selection signal and its complement are connected to the transistors of one T-gate, while the opposite arragement is made for the second T-gate, as in Figure5.8. Finally, the layout used to implement the switch in hardware is shown in Figure 5.9.

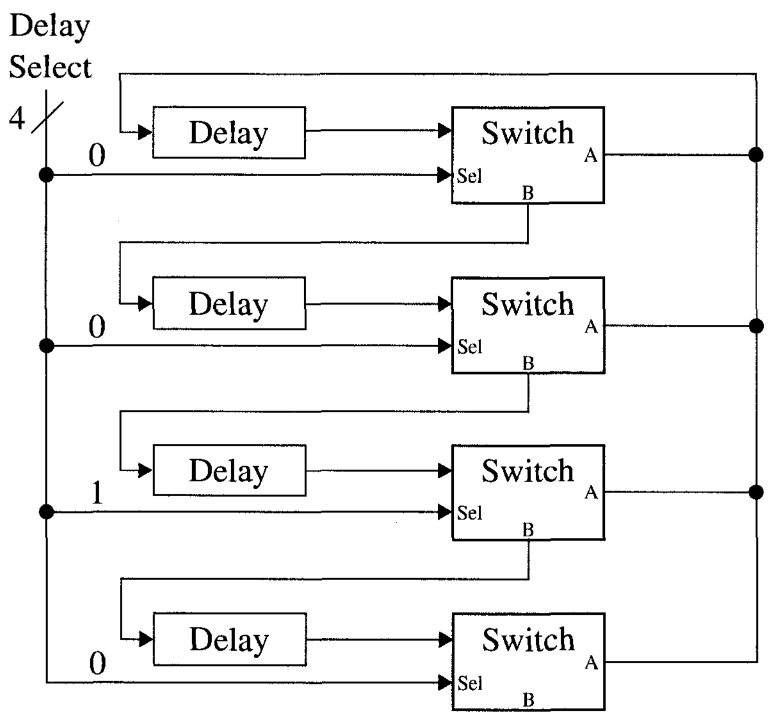

Figure 5.6: Inverter Ring Example Using Control Word "0010"

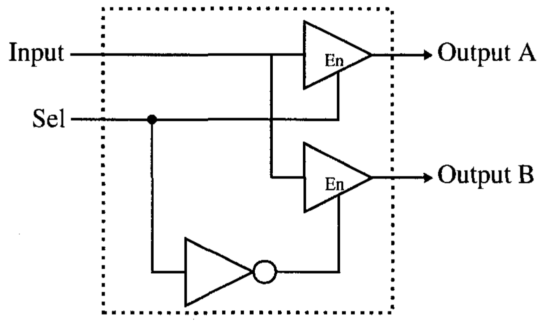

Figure 5.7: Schematic for the Switch Block

5.3.4 Proposed Inverter Ring Design Specifications and Simulation Results

The ring was designed to generate a series of 32 different clock frequencies, spanning the range from approximately $75MHz$ to $350MHz$. Greater frequencies will not be needed; even if the RALUT design is able to perform at or above $350MHz$, the static CMOS test circuitry will experience difficulty keeping up. Switches were inserted into the inverter ring as required to allow the number

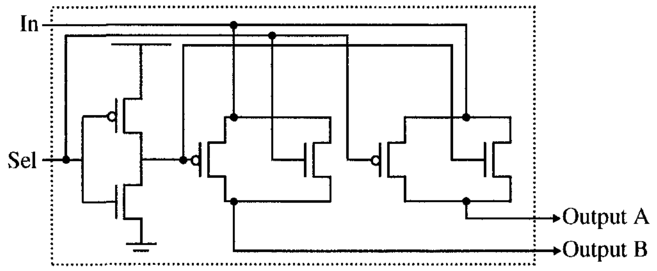

Figure 5.8: Transistor Schematic for the Switch Block

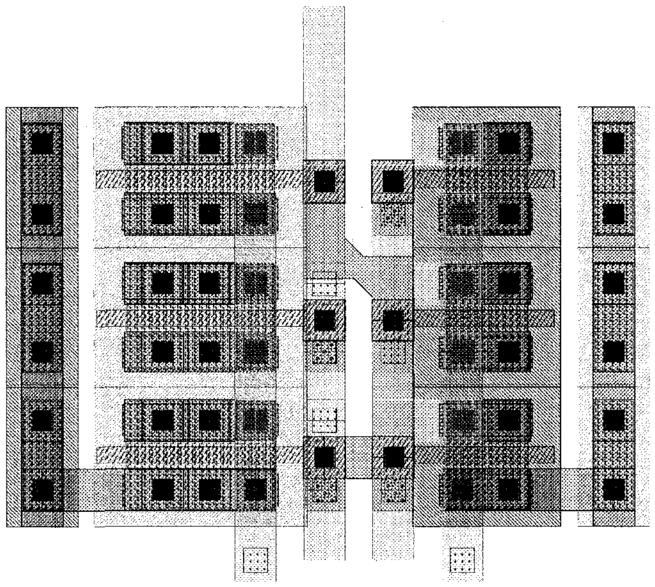

Figure 5.9: Layout for the Switch Block

of stages to be user-selectable during the test stages. To ensure that only one switch is enabled at any given time, a standard-cell 5-to-32 decoder will be responsible for generating the inputs for this design.

To ensure that the inverter ring behaves as designed as parasitic capacitances are introduced, a complete simulation was performed, testing all 32 clock modes. Sample simulation waveforms of the inverter ring's output at 350, 200, and 75 MHz are shown in Figures 5.10, 5.11, and 5.12, respectively.

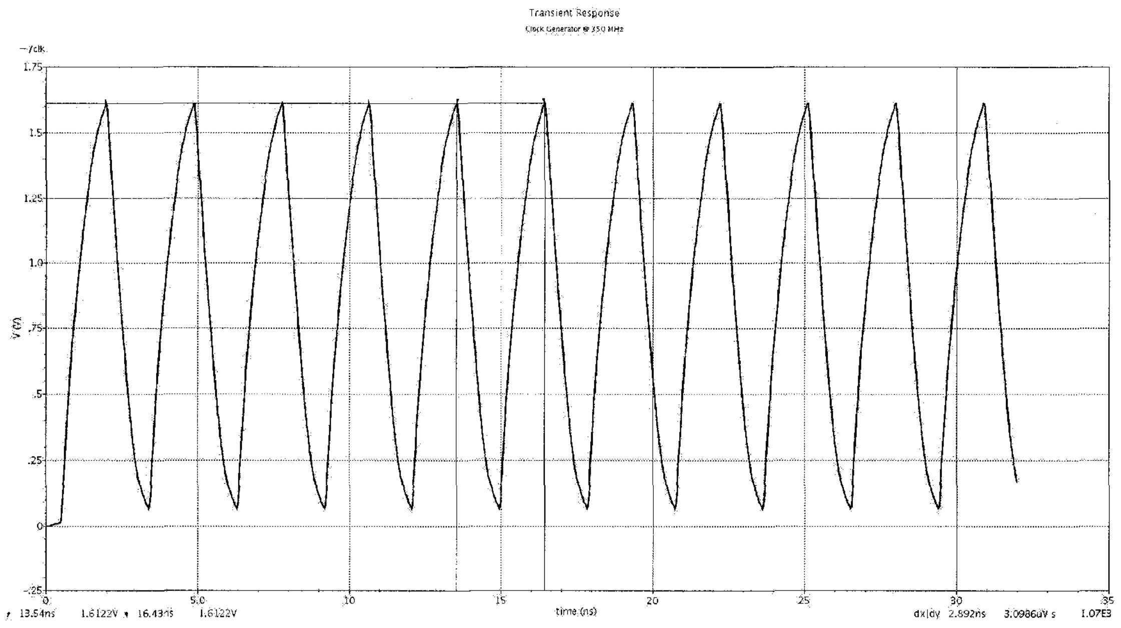

Figure 5.10: Simulation Waveform for the Ring Oscillator @ 350 MHz

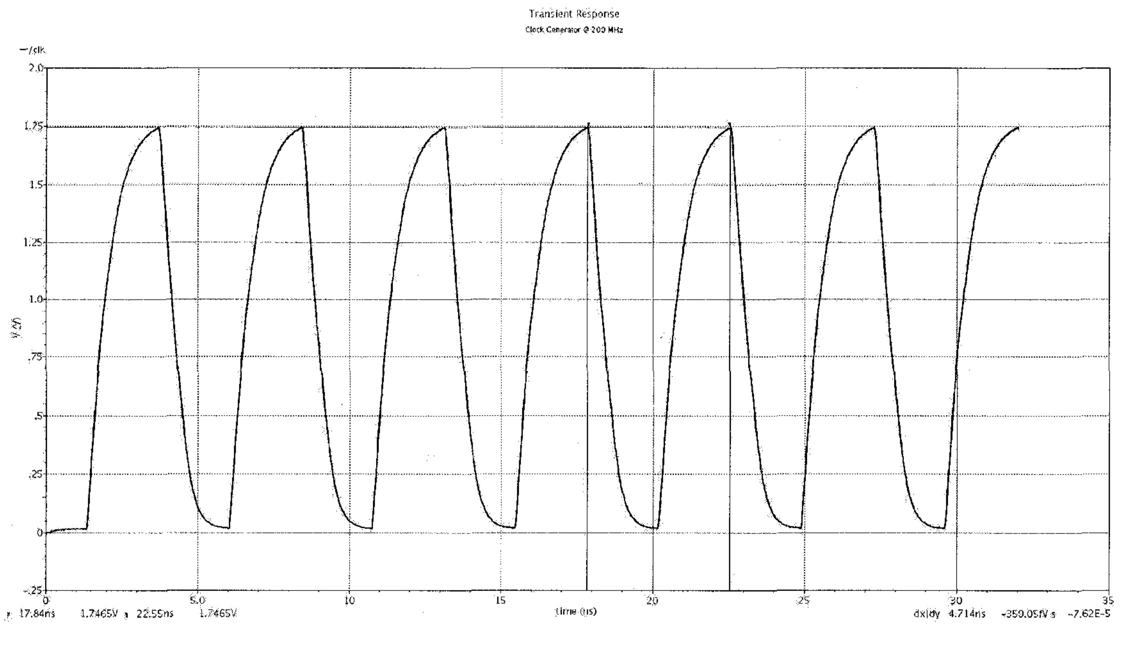

Figure 5.11: Simulation Waveform for the Ring Oscillator @ 200 MHz

5.4 Test, and Output Select Circuits

Ideally it would be more convenient and more flexible to simply provide input patterns to the IC's pins, and verify its outputs, however, once again due to the limited slew rate of the I/O pins, this is

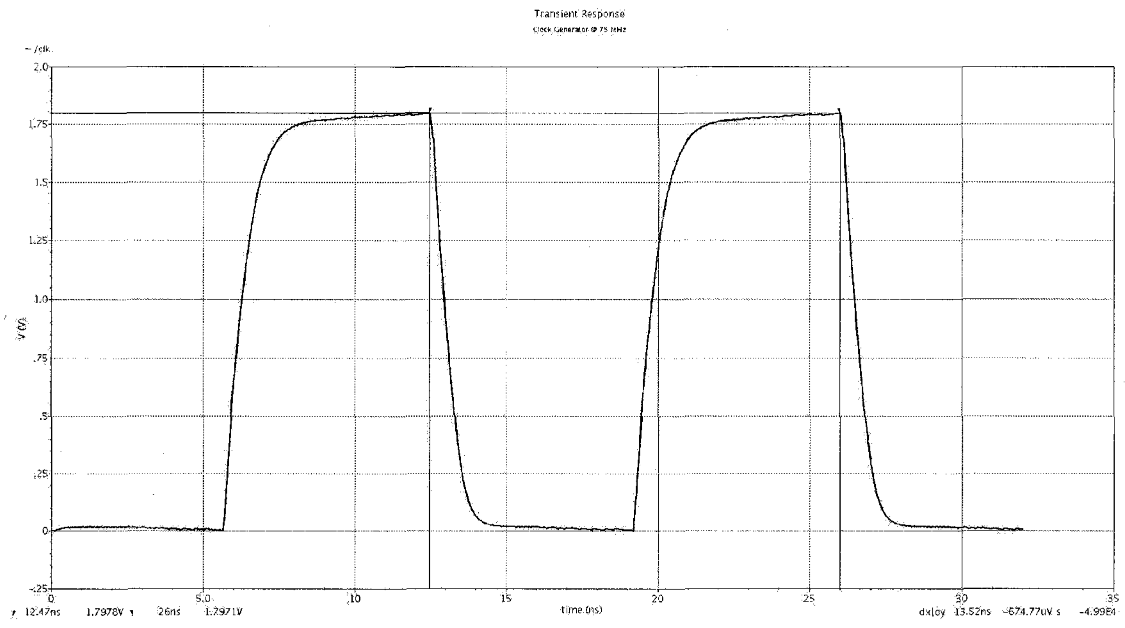

Figure 5.12: Simulation Waveform for the Ring Oscillator @ 75 MHz

not possible. The test circuit is at the heart of the test IC, and its block diagram is shown in Figure 5.13. It is responsible for generating the test patterns and verifying the results, and makes use of the following sub-components:

- Automatic test pattern generator (ATPG)
- Full-custom RALUT block
- Standard-cell HDL RALUT block
- Compare circuit
- Output select circuit
- Pipelining registers

The goal of the test circuit is to determine the correct operation of the RALUT at the designated clock speed. This is a difficult task, as the full-custom RALUT design is expected to work at very high speeds. In order to determine if it is functioning correctly at such speeds, a set of known 'correct' input/output values must be stored on the same IC to verify the full-custom design's results. A classic LUT would be ideal for this purpose, unfortunately the available silicon area is only $1\mu m \times 1\mu m$. It is preferable to maximize the area available to the RALUT, and a LUT

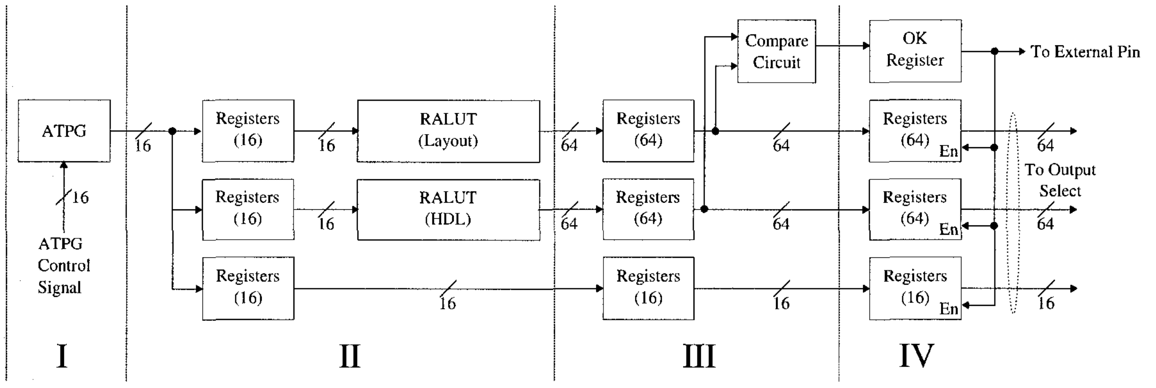

Figure 5.13: Test Circuit Block Diagram With Pipelining

implementing the same functionality will not be able to fit on the IC. The proposed test circuit design makes use of another, standard-cell RALUT to perform this task.

This second RALUT is designed using the verilog hardware description language (HDL), and synthesized into a gate-level netlist with Synopsys. As will be shown in subsequent sections, this design can be compiled to also function at relatively high speeds, while occupying minimal area, allowing it to adequately test the full-custom design up to at least moderate speeds.

It is important to remark that this design is pipelined. The various stages of the pipeline are shown in Figure 5.13, and are denoted by I, II, III, and IV. The pipelined approach was taken to ensure that the operating time afforded to the RALUT is easily determined by the clock frequency; by registering the RALUT inputs and outputs, only the RALUT itself contributes to the critical path delay, rather than the RALUT in addition to input generation and output verification logic delays.

The complete test circuit functionality works as follows:

1. The automatic test pattern generator (ATPG) places a 16-bit pattern on its output, and is captured by the input registers in II

2. The input patterns proceed through both the full-custom and standard cell RALUT designs, generating 64 bit outputs; they are saved into the registers in III
   The original input pattern responsible for generating the RALUT outputs propagates through the pipeline

3. The compare circuit proceeds to evaluate if the outputs of both designs are equal, if they are, the result is *logic 1*, which is saved into the OK register, otherwise the OK register is set to

logic 0

Both the 64-bit output patterns that are currently being compared, as well as the original input that generated them, propagate through the pipeline registers in IV

4. If the OK register is high, the test-circuit continues to operate, if it is low the write-enables on the last series of pipeline registers in IV are disabled

5. The OK signal is connected to an external pin which should be monitored by a microcontroller or other device; if it is low, the values of the registers in IV can be placed on the output bus to determine what failed and why

Anoter important remark is that this design employs negative edge triggered flip-flops in its registers. This allows for a smooth integration of the domino logic RALUT design; latching flip-flops on the negative edge ensures a maximal amount of time for the domino gates to evaluate. Finally, the test-circuit has a series of reset signals going to every one of its sequential logic components. This is done to allow the device to begin working as power is enabled and the reset pin triggered, to aid in debugging the IC in a physical testbench.

The remainder of this section goes into further detail regarding the sub-components of the test circuit, including the selection of the RALUT, the ATPG design, and the output select circuit.

5.4.1 Range Addressable Lookup Table Selection

It is desirable to test the largest possible design in order to better determine the RALUT operating performance, wherein a variety of different bit patterns are used for the many address decode stages. The IC core for this work (the silicon area of the IC without considering bonding pads) is $1mm \times 1mm$. For these reasons, a RALUT consisting of 128 rows, a 16-bit input, and 64-bit output was used; its dimensions are $307\mu m \times 749\mu m$. The quickly-designed, rather than the high-performance RALUT described in the previous chapter was used. Although using the high-performance design would have been preferred, it was not ready in time for the fabrication deadline. This design occupies the majority of the available vertical space, while allowing approximately 250 μm for power rings and I/O lines. This amount of space was left deliberately to ensure that the routing tool would be able to connect the design with the rest of the circuit. Although enough space remains on the IC core to expand the number of input and output bits of the proposed RALUT design, 16 input bits and 64 output bits were used to simplify testing, and also because these are practical values that may be used in future work.

5.4.2 Automatic Test Pattern Generator

While operating at speeds in excess of 50 MHz, using external pins to supply the input address is no longer feasible. Again, this is due to the limitation of the I/O pads, which, due to their capacitance, require several microseconds to charge and discharge. To overcome this issue, the proposed design employs internal automatic test pattern generator (ATPG) circuitry to from input address test vectors.

The most straight forward approach for this design is to use a binary counter. They are easy to implement, and will cycle through every input address, allowing for the verification of every input/output pair. A concern with using a sequential counter, however, is that only the lower order input address bits will change at much higher frequencies than the most significant address bits. This will only test the speed limitations of the final stage of the RALUT address decoder. Since the goal of this IC design is to fully test every element of the RALUT design, particularly every stage of the address decoder, this is not an acceptable solution.

To address this problem, the use of a linear feedback shift register (LFSR) as described in [23] is proposed. A LFSR is similar to a counter, except that the output patterns are pseudo-random. Every possible output combination will be generated out of order, in a predictable, repeating sequence. As an example, the output of a 4-bit LFSR is presented in Table 5.1.

The pattern shown in the table repeats, which is why the LFSR is said to generate a pseudo-random output rather than a truly random output.

5.5 The Control System

The test IC is designed to interface well with microcontrollers for easy use on a physical testbench. As such, it posseses an 8-bit input bus, so that it can interface with common I/O ports. This input controls the entire functionality of the test IC design. As shown in Figure 5.14, the input word is divided up into two parts, 4 bits of data and 4 address bits. The address bits pass through a 4-to-16 decoder, enabling one of 16 different sets of 4-bit registers, where the data component of the input whill be stored. This scheme allows for a maximum of 64 bits of signals to be controlled, however only 36 are used. Also worth noting, is that the control registers are driven by a separate external clock signal, and not shown on the diagram is a reset signal which places the control registers in an initialized state.

A complete list of the control unit signals and their locations in the control register follows in Table 5.2

Time	Output Pattern
0	0001
1	0011
2	0111
3	1111
4	1110
5	1101
6	1010
7	0101
8	1011
9	0110
10	1100
11	1001
12	0010
13	0100
14	1000
15	0001

Table 5.1: 4-Bit LFSR Output States

Control Register Bits	Control Signal Description
0 - 2	External Clock Signal Select
3	Clock Enable
4 - 6	Internal Clock Signal Select
7	Enable Clock to Full-Custom RALUT
8 - 12	High-Speed Internal Clock Mode Select
13	Enable Clock to HDL RALUT
14	Reset Test Circuit
15 - 19	Select Output Address
20 - 35	ATPG Initial Value

Table 5.2: Control Unit Signals

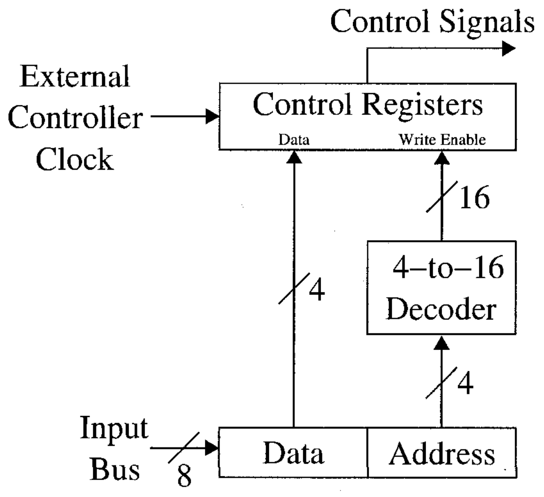

Figure 5.14: The IC Register-Based Control System and Input Word

5.6 Hardware Synthesis

With the exception of the inverter ring and full-custom RALUT, HDL code was written for every one of the components described in this chapter. These components were all designed using verilog; the HDL code, as well as test benches are available in Appendix B. Existing code was available for the RALUT, it is also available in Appendix B.

Synopsys was used with Artisan standard cells for gate-level synthesis. Synthesis parameters were optimized for speed, particularly in the case of the HDL RALUT. Scripts indicating the exact parameters are in Appendix E.1.

5.7 Simulation

Simulation is an important step in any hardware design methodology. It is relatively easy and inexpensive to correct issues if they are found during simulations, and it is impossible to modify a VLSI IC once it is fabricated.

Simulating the IC proved to be a significant challenge. The workstations available in the RCIM lab at the University of Windsor possessed a maximum of 2GB of RAM, greatly limiting the size of the designs that can be simulated in a reasonable amount of time. Another major simulation limitation is that only "black box" versions of the standard cells used in the HDL design portions of this work are available. These cells show the locations of their I/O pins, and their analog modelling is available, however the cell layouts are not known. Without knowledge of the locations of the various metal layers inside the cells, it is impossible to determine parasitic capacitances. In other words, it is impossible to simulate the entire circuit with parasitic capacitances taken into account.

Due to the available workstations' limitations, a divide-and-conquer approach to simulating the design was taken. The test IC was separated into its components and tested individually to ensure correct functionality.

Digital Simulation

The digital components of this design were all extensively tested using verilog test benches. The test bench code for each of the components is available in Appendix B.2. Individual components were tested to ensure functionality, integrated into larger designs, and then these larger designs were tested. Small errors were found and corrected; this iterative bottom-up approach to simulation and verification proved effective.

Once the digital designs were determined to be working correctly, the gate level code was synthesized and imported into Cadence so that it could be simulated in an analog environment. Unfortunately the workstations currently available were unable to simulate the entire gate-level code at once, and it had to be further subdivided in order to simulate correctly.

The digital design was separated into the control block and the clock controller, and these two elements were successfully simulated for several clock cycles. Once again, due to the limited computational resources, it is only feasible to simulate a limited number of clock cycles, as each of these may require many hours of processing time.

Analog Simulation

The full-custom layouts of the RALUT and inverter ring were simulated using Spectre in Cadence Analog Environment. Section 5.3.4 details the simulation results of the inverter ring. A sample simulation waveform of the RALUT is shown in Figure 5.15, the entire simulation waveform has been omitted for clarity, as it would span nearly 100 rows. Also omitted for clarity are the address lines.

This waveform shows the clock signal at the top, followed by four output lines: 3, 2, 1, and 0 from top to bottom. The outputs are switching appropriately with different input addresses, although some jitter can be seen from time to time. While this may appear to be a serious fault at first, it is important to remember that the output of a domino logic circuit is not valid while the clock is at *logic 0*, as is the case during these glitches.

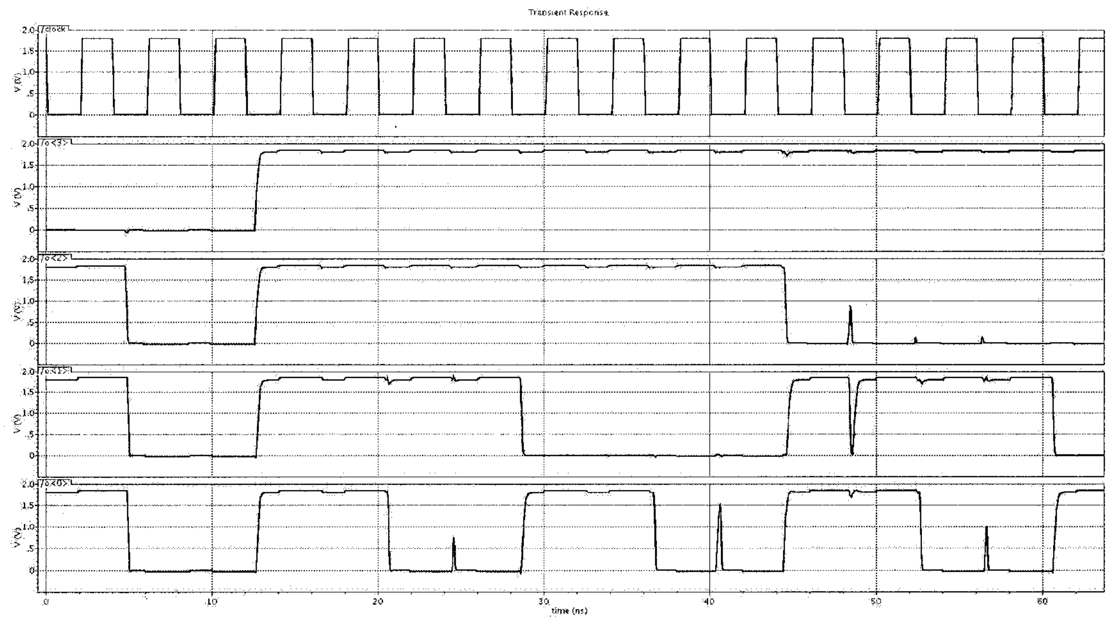

Figure 5.15: Simulation Waveforms for the RALUT, from Top to Bottom: Clock Signal, Output Line 3, 2, 1, and 0

5.7.1 Design Rule Check

Design Rule Checks, or DRCs, are a set of checks that are performed to confirm that the design does not violate any of the fabrication process parameters. Examples of the parameters that are checked are maximum transistor width, minimum gate overlap, and minimum metal1 separation. DRCs for this work were performed using Diva as well as Calibre. During early design stages, specifically while individual cell layouts were being created, Diva was used to perform DRCs. It possesses fast execution times for smaller layouts, and is well-integrated with the Cadence tools. Calibre was used to DRC the entire IC; it is a more robust tool, and properly detects certain violations that Diva does not, such as antenna violations. Additionally, final DRC checks are also performed by Calibre. The University of Windsor does not currently have access to full cell-views, meaning the DRC software cannot determine if there are any violations occuring in the black box standard library cells. In order to determine the legitimacy of a standard cell design, it must be uploaded to CMC's DRC server, where Calibre is run locally on their system, and the results are made available for download.

5.8 IC Design

Once the design was determined to be functioning correctly in simulation results, Encounter was used to perform cell placement and routing, power ring and stripe placement. Results of this step are pictured in Figures 5.16, and 5.17. In these figures. routing layers for metals 4, 5, and 6 have been removed to better show the chip's internals. On the left is the full-custom RALUT, and in the bottom-right corner is the inverter ring. The remaining area is mostly filled with the standard cells making up the chip's testing and control circuitry, as well as the HDL RALUT design.

5.9 Test IC Summary and Results

The test IC described in this chapter will be able to fully simulate the RALUT design such that every input combination can be tested, operating speeds can be determined, and correct functionality determined. The robust clock controller circuit allows for simulation at a wide variety of speeds, while the simple 8-bit input and output ports of this design will allow it to easily interface with a microcontroller for testing. In the future, this design framework could be used to test other memory architectures, greatly reducing the design time in creating a custom built-in self-test module.

The test IC has been fabricated, and has been tested. Unfortunately, it is failing to respond to even basic tests. This suggests that there was a problem with the fabrication process, and CMC is currently being contacted in order to further investigate the issue.

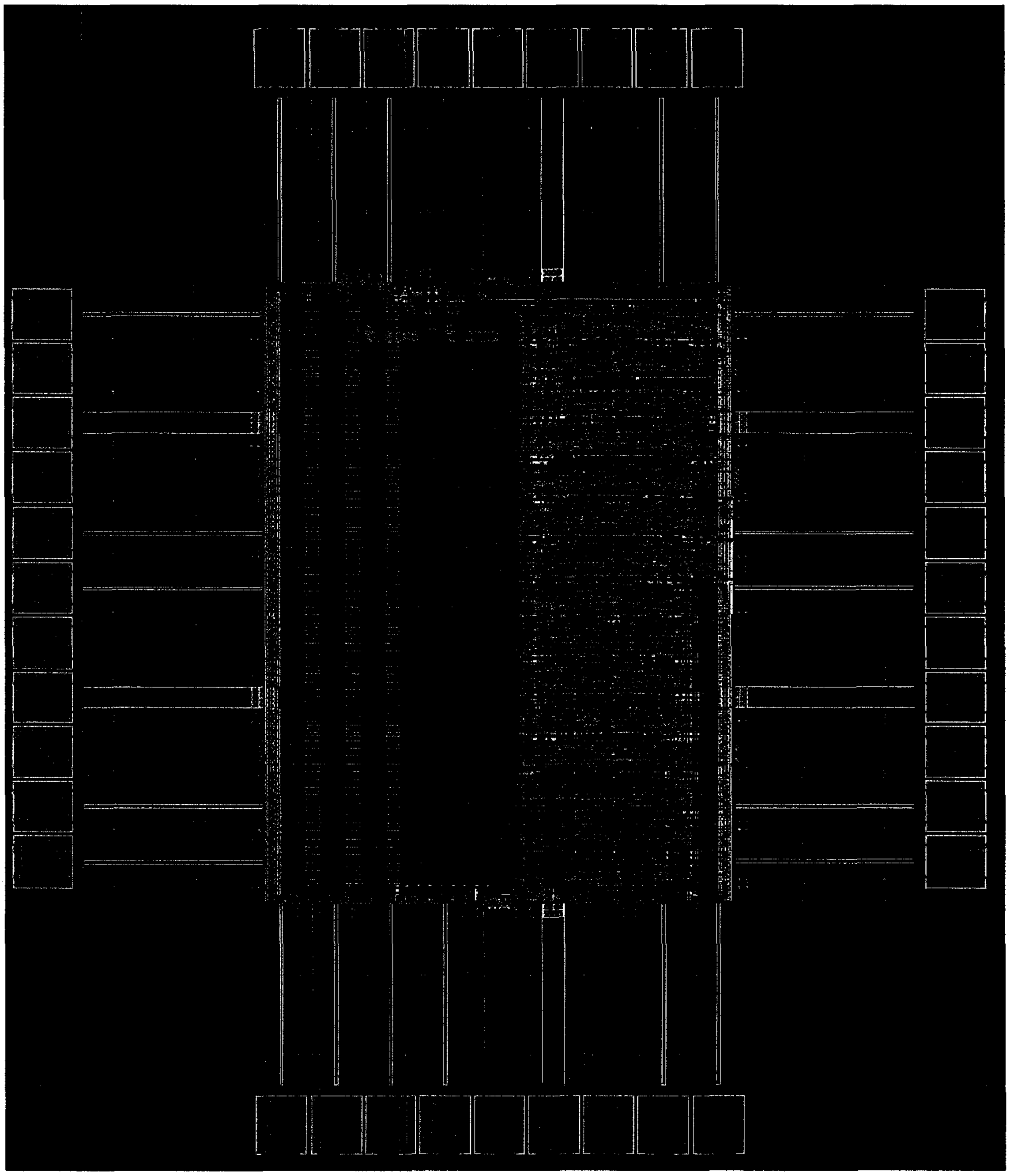

Figure 5.16: Complete IC Layout

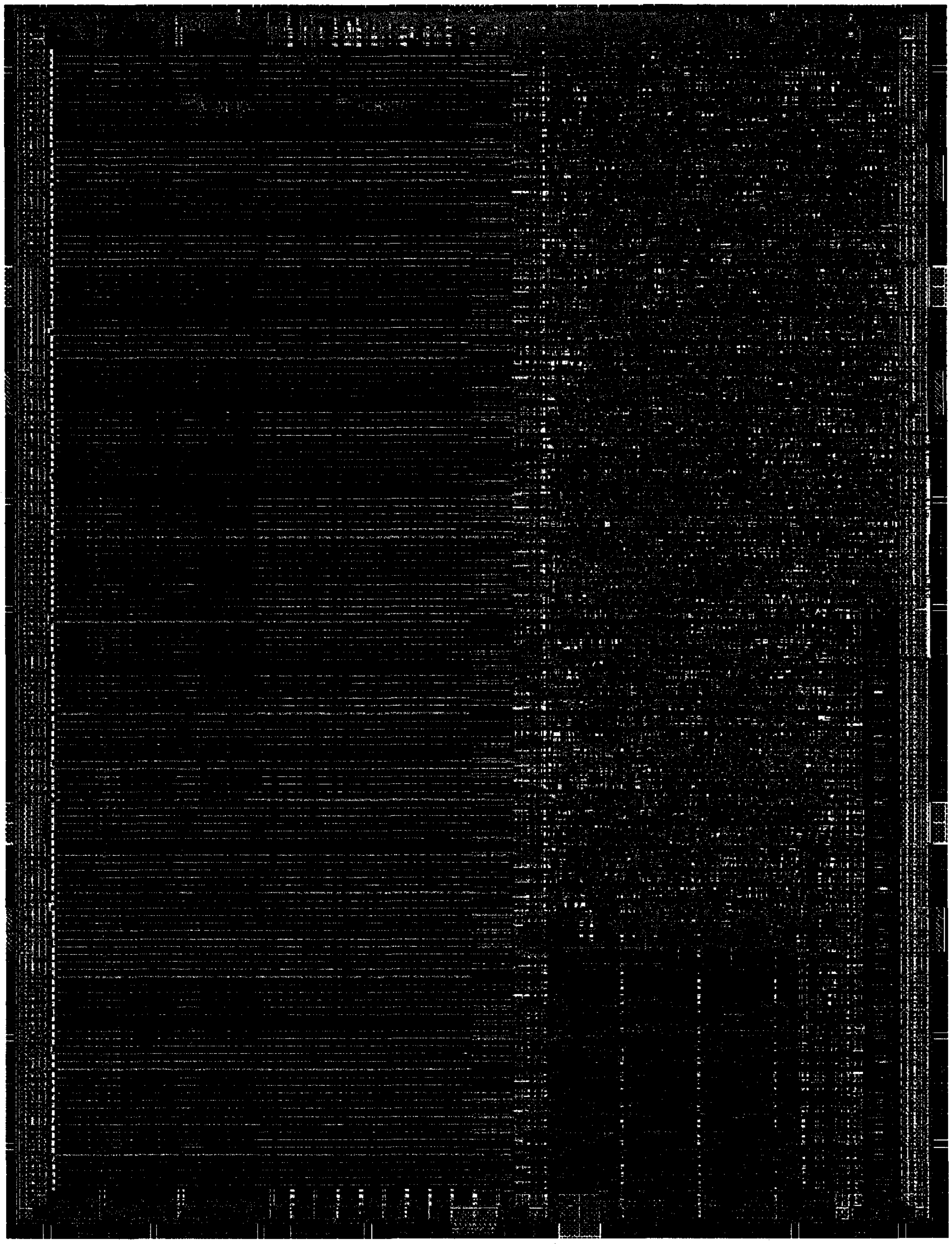

Figure 5.17: Close-Up View of the IC Core

Chapter 6

Case Study: Range Addressable Lookup Tables in Artificial Neural Networks

In this chapter, a RALUT implementation of the hyperbolic tangent function is presented. Hardware implementation results show that a RALUT implementation was significantly faster and smaller than a recently published picewise linear (PWL) approximation method, while possessing the same level of accuracy. Hardware designs were implemented using a digital CMOS $0.18\mu m$ process; the same technology node used by the PWL implementation used in the comparison. Additional comparisons are made between the RALUT implementation and a series of other PWL methods implemented on an FPGA, further demonstrating the RALUT's superior performance.

The rest of this chapter is organized as follows. Section 6.2 briefly reviews previous work on hyperbolic tangent function implementation. Section 6.3 discusses a LUT-based approach, while section 6.4 examines the RALUT approach to implement the activation function. In section 6.6, a complexity comparison between several different methods is presented, and section 6.3 presents additional comparisons to published work that emply FPGAs. Finally, section 6.8 summarizes the results.

6.1 Artificial Neural Networks and Activation Functions

Artificial neural networks (ANNs) are currently employed for many diverse purposes, ranging from image classification to motor control [9, 21]. Since ANN systems are computationally intensive, they require large execution times in software implementations. Hardware implementations can eliminate this issue. One of the challenges presented when designing a hardware-based ANN system is the implementation of the activation function. There are several different activation functions available including, but not limited to, the sigmoid, hyperbolic tangent, and step functions [9, 21]. An important property of the activation function is that a continuous derivative exists, which is desirable when performing backpropagation-based learning. These functions are used to threshold the output of every artificial neuron; increasing the speed of the activation function will improve the entire system's performance.

The hyperbolic tangent function is among the most widely used activation functions in ANNs. As it is shown in Fig. 6.1, this function produces a sigmoid curve, which is a curve having an "S" shape. Its variation is limited outside the range of $(-2, 2)$.

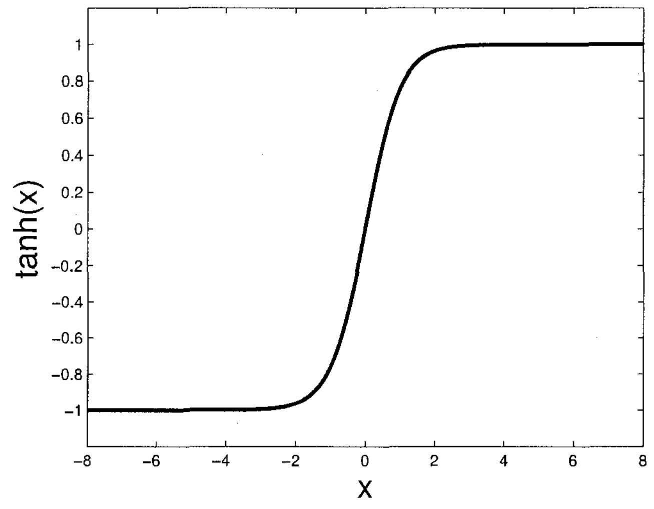

Figure 6.1: The Hyperbolic Tangent Activation Function

Currently, there are several different approaches for the hardware implementation of the activation function. Piecewise linear approximation (PWL), lookup tables (LUTs), and hybrid methods have been widely used for this purpose [3, 19, 27]. With the use of current hardware synthesizers, LUTs are not only faster, but also occupy less area than piecewise linear approximation methods.

In this work, range addressable lookup tables are proposed as a solution that offers advantages compared to simple LUT implementation in terms of speed and area utilization.

This type of table was originally proposed in [16] to implement highly nonlinear, discontinuous functions, and it will be shown to be suitable for implementing the hyperbolic tangent activation function. Depending on the desired accuracy, ranges of inputs will have the same output, which could be implemented more efficiently using RALUTs rather than a regular LUT.

6.2 A Brief Review of Different Hyperbolic Tangent Function Implementations

Efficient implementation of the activation function is an important part of designing an ANN system in hardware. The activation function is typically unsuitable for direct implementation since it is formed of an infinite exponential series. In practice, approximations of the function are used, as opposed to the function itself.

Currently, there are three main approaches used to approximate and implement the hyperbolic tangent function in hardware; lookup table (LUT) approximation, piece-wise linear (PWL) approximation, and hybrid methods, which are essentially a combination of the former two. Following is a brief overview of each of these methods.

6.2.1 Piecewise Linear Approximation

Piecewise linear schemes use a series of linear segments to approximate a function [3]. The number and location of these segments are chosen such that error, processing time, and area are minimized. This approach usually requires several clock cycles and the use of multipliers, which are expensive in terms of area. A piecewise linear approximation of the hyperbolic tangent function with five segments is shown in Fig. 6.2.

6.2.2 Lookup Table Approximation

In this method, the function is approximated with a limited number of points [19]. The points are uniformly distributed across the entire input range. There is a direct relation between the number of bits used to represent the address (input) and output, and as such, care must be taken to ensure enough are used to minimize the error. A LUT approximation of the hyperbolic tangent function with eight points (a three bit input representation) is shown in Fig. 6.3.

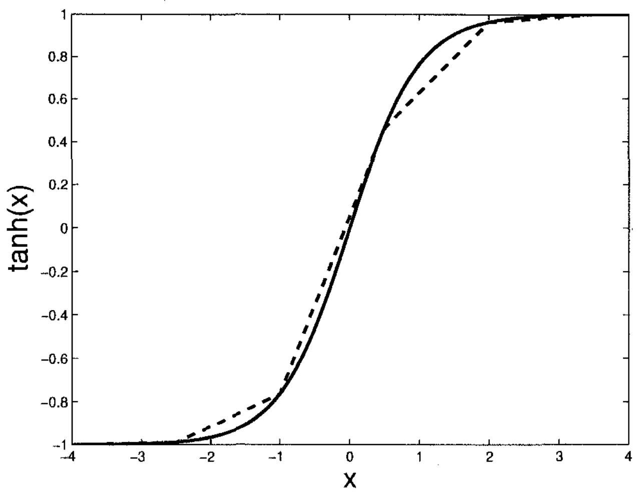

Figure 6.2: Piecewise Linear Approximation of tanh(x) with Five Segments

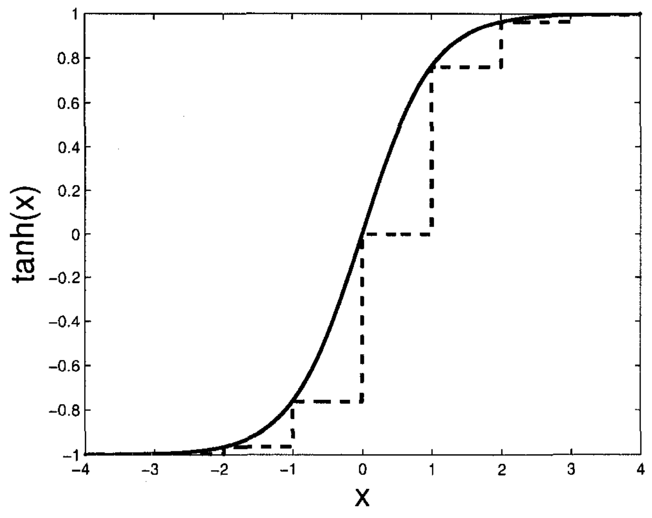

Figure 6.3: Lookup Table Approximation of tanh(x) with Eight Points

6.2.3 Hybrid Methods

Hybrid methods use a combination of look-up tables and other hardware to generate the result of a function [27]. They typically take several clock cycles, however they do not employ multipliers, which significantly increases their speed.

6.3 Lookup Table Implementation of the Hyperbolic Tangent Function

The major advantages of using a LUT is its high operating speed, particularly when compared to PWL approximation which uses multipliers in its design. The are two different ways to implement a lookup table in hardware. The first is to use a ROM. The main drawback of this method is that no further optimization can be done after the exact input/output bit patterns are known.

The second method is to use a logic synthesizer to implement the table as a purely combinational circuit. This works well because the synthesizer excels in optimizing away large amounts of logic.

In the implementation, MATLAB code was generated to determine the number of input and output bits, as well as the output bit patterns themselves, for a table with a specified maximum error. For a maximum error of 0.04, 9 bits were used for both the input and output, whereas 10 bits were required to keep the maximum error below 0.02.

Once the input/output characteristics of the table were determined, HDL code employing them was written, and a hardware design was synthesized using Synopsys' Design Compiler. Virtual Silicon standard library cells for a TSMC CMOS $0.18\mu m$ process were used for this design, and synthesis parameters were chosen to maximize operating speed. Hardware implementation results with a maximum error of 0.04 and 0.02 are summarized in the second row of tables 6.1 and 6.2 respectively.

6.4 Range Addressable Lookup Table Implementation of the Hyperbolic Tangent Function

A range addressable lookup table, originally proposed in [16] to accurately approximate non-linear, discontinuous functions, shares many aspects with the classic LUT with a few notable differences. In LUTs, every data point stored by the table corresponds to a unique address. In RALUTs, every data point corresponds to a range of addresses. This alternate addressing approach allows for a large reduction in data points, particularly in situations where the output remains constant over a range. An example of this is the hyperbolic tangent function, where the output changes only slightly outside the range of $(-2, 2)$. Rather than store every individual point, a single point is used to represent an entire range.

To implement the hyperbolic tangent function, MATLAB code was written to select the minimum number of data points, while keeping the maximum error beneath a specified threshold. The

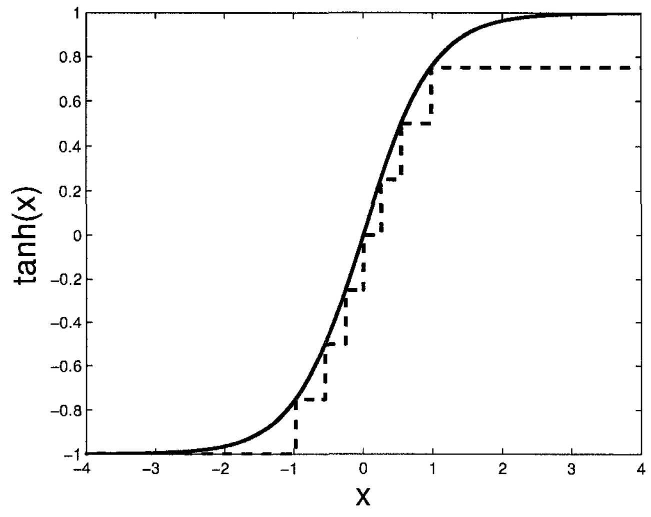

Figure 6.4: Range Addressable Lookup Table Approximation of tanh(x) with Eight Points

MATLAB code is available in Appendix C. It was possible to represent the activation function with 61 points using 9 bits for the inputs and outputs, with an error below 0.04 . Using a 10 bit representation, only 127 were needed to maintain a maximum error below 0.02 . The required number of points for these levels of maximum error using classic LUTs were 512 and 1024, respectively. This large reduction in stored values is what drives the RALUT approach to achieve better results than a LUT implementation of the same function.

6.5 Results and Comparison

Architectures	Max-Error	AVG-Error	Area	Delay	Area $\times$ Delay
Scheme-1 [11]	0.0430	0.0078	32069.83 μm^2	903 ns	2.895×10^{-5}
Proposed-LUT	0.0365	0.0040	9045.94 μm^2	2.15 ns	1.944×10^{-11}
Proposed-RALUT	0.0357	0.0089	7090.40 μm^2	1.85 ns	1.311×10^{-11}

Table 6.1: Complexity comparison of different implementations for 0.04 maximum error

Both sets of data points were passed on to HDL code, and the designs were synthesized with Synopsys Design Compiler using CMOS 0.18μm technology. Design parameters were chosen to maximize operating speed. Implementation results are shown on the last row of tables 6.1 and 6.2.

Architectures	Max-Error	AVG-Error	Area	Delay	Area × Delay
Scheme-2 [11]	0.0220	0.0041	$83559.17\mu m^2$	1293 ns	1.080×10^{-4}
Proposed-LUT	0.0180	0.0020	17864.24 μm^2	2.45 ns	4.376×10^{-11}
Proposed-RALUT	0.0178	0.0057	11871.53 μm^2	2.12 ns	2.516×10^{-11}

Table 6.2: Complexity comparison of different implementations for 0.02 maximum error

6.6 Comparison of Different Hardware Implementations

Comparisons of hardware implementations for a maximum error of 0.04 and 0.02 are shown in tables 6.1 and 6.2. In table 6.1, the first row represents results from "Scheme-1", which is an isosceles triangular approximation of the hyperbolic tangent function. In table 6.2, the same row shows results from "Scheme-2", which is a PWL approximation of the hyperbolic tangent function. Both Scheme-1 and Scheme-2 designs were implemented using CMOS $0.18\mu m$ technology; the same used by the proposed implementations. Also note that all designs accept an input in the range of $(-8, 8)$.

The proposed RALUT design was able to improve over the LUT implementation in both cases. With a maximum error of 0.04, the RALUT was 13% faster, and occupied 21.6% less area than the classic LUT approach. When the maximum error threshold was reduced to 0.02, the RALUT maintained a speed improvement of 13.5%, and area was further reduced by 33.5% compared to the LUT.

As can be seen from the tables, the LUT and RALUT designs prove to be significantly faster than the work recently presented in [11]. This is largely because this approach uses combinational logic exclusively, allowing results to appear after a single clock cycle, whereas multiple clock cycles are needed by the other designs. The "*area* × *delay*" was calculated as a performance metric to compare the overall efficiency of the designs. It is shown in the last column of tables 6.1 and 6.2.

6.7 Comparison to FPGA Implementations

While Section 6.6 outlines a direct comparison of two ASIC VLSI designs using CMOS $0.18\mu m$ technology, other designs are available which target FPGA platforms. In an effort to broaden the proposed design's basis of comparison, it will be compared with FPGA implementations of ANN activation functions. First, however, it is important to understand the some key differences between FPGAs and ASIC designs.

Rather than use a library of cells as building blocks, FPGAs make use of regularly patterned groups of logic that typically contain a combination of small look up tables, multiplexers, and flip flops to implement their designs. The FPGA implementations in Table 6.3 are a reproduction of the results published by [22], and were implented on a Xilinx Virtex-II XC2V40 device. Virtex-II FPGA devices make use of 'slices', which is a term used by the Xilinx corporation to describe a unit of logic elements. Also, FPGAs do not use um^2 of silicon as an area metric; they refer to how many slices a design occupies. According to their datasheet [28], a slice contains the following:

- Two function generators
- Two storage elements
- Arithmetic logic gates
- Large multiplexers
- Wide function capability
- Fast carry look-ahead chain
- Horizontal cascade chain (OR gate)

It is due to this large mix of resources on a slice that renders it difficult to form a direct comparison between FPGA and ASIC area utilization.

Another major factor affecting the fairness of comparison is the fabrication technology used to fabricate both the ASIC and FPGA. The Virtex-II FPGA used in this comparison as described as "$0.15\mu m$ / $0.12\mu m$ CMOS 8-layer metal process with high-speed transistors", meaning it is at a more advanced technology node than the proposed CMOS $0.18\mu m$ design. Unfortunately, the University of Windsor does not possess a design kit for either of the $0.15\mu m$ or $0.12\mu m$ CMOS processes, and as such a direct performance comparison is also not possible. Luckily, however, the FPGA is not several generations ahead in technology, so this will only slightly skew the comparison in favour of the results reported by [22].

Table 6.3 displays how the proposed work compares with other designs which also report a maximum error of approximately 0.02. The critical path delay of the proposed RALUT design is significantly lower than all other results, including the FPGA implementations. This is due to the fact that the FPGA designs require multiple clock cycles to determine their results, whereas the RALUT is purely combinational. Although not essential, it is worth remarking that the average error of the proposed design is also lower than the FPGA implementations. Also compared in the table is a high-performance full-custom RALUT design. It is larger than the ASIC designs, however it possesses the minimum critical path delay. This design would be ideal in a situation requiring the fastest function approximation possible.

Architectures	Max-Error	AVG-Error	Area	Delay	Platform
Zhang et al. [12]	0.0216	0.0077	176 Slices	15.06 ns	FPGA
Alippi et al. [1]	0.0189	0.0087	36 Slices	15.58 ns	FPGA
CRI (q=3) [3]	0.0206	0.0085	65 Slices	86.21 ns	FPGA
CRI (q=4) [3]	0.0197	0.0084	65 Slices	114.94 ns	FPGA
LUT	0.0180	0.0020	17864.24 μm^2	2.45 ns	ASIC
RALUT	0.0178	0.0057	11871.53 μm^2	2.12 ns	ASIC
High Performance RALUT	0.0178	0.0057	39442 μm^2	1.60 ns	Custom

Table 6.3: Complexity comparison of different implementations for 0.02 maximum error, including FPGA implementations

6.8 Summary

The hyperbolic tangent function is commonly used as the activation function in artificial neural networks. In this work, two different hardware implementations for this function are proposed. The first method uses a classic LUT to approximate the function, while the second method uses a RALUT to do so. Hardware synthesis results show that proposed methods perform significantly faster, and use less area compared to other similar methods with the same amount of error. On average, the speed was improved by 13%, while area was reduced by 26% when using the second method compared to first in implementing a hyperbolic tangent function. A comparison with FPGAs was carried out to show that the propsed design also performs well against these approaches, particularly in terms of critical path delay. The full custom, high-performance RALUT was also compared, and performed the best in terms of critical path delay, however its area was larger than the standard cell ASIC designs. The proposed designs can be used in the hardware implementation of ANNs.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The CMOS $0.18\mu m$ technology node was selected to design a rescaled version of the existing CMOS $0.35\mu m$ design, as well as a high performance implementation of the RALUT. The rescaled version was quickly designed, allowing for it to be included in test IC in time for fabrication, while offering 65% better results than the $0.35\mu m$ design. The high-performance design demonstrates excellent results with an 84% improvement over the CMOS $0.35\mu m$ design for a typical RALUT size.

The superior results achieved in the high performance design were made possible due to the properly sized transistors and optimal RALUT design parameters. The ideal number of address bits per decode stage was determined to be 6, every linedriver should drive no more than 16 output bits, and no fewer than one buffer should be used for every 8 rows of the design.

Also, due to the performance data collected, it is now possible to estimate the delay of the high performance RALUT design.

The integrated circuit test platform will be able to fully simulate the RALUT design, such that every input combination can be tested, operating speeds can be determined, and correct functionality determined. The robust clock controller circuit allows for simulation at a wide variety of speeds, while the simple 8-bit input and output ports of this design will allow it to easily interface with a microcontroller for testing. In the future, this design framework could be used to test other memory architectures, greatly reducing the design time in creating a custom built-in self-test module.

A case study of how RALUTs can be used to approximate non-linear functions was carried out on the activation function of artificial neural networks. Two different hardware implementations for this function were proposed. The first method uses a classic LUT to approximate the function, while the second method uses a RALUT to do so. Hardware synthesis results show that proposed methods perform significantly faster, and use less area compared to other similar methods with the same amount of error. On average, the speed was improved by 13%, while area was reduced by 26% when using the second method compared to first in implementing a hyperbolic tangent function. A comparison with FPGAs was carried out to show that the propsed design also performs well against these approaches, particularly in terms of critical path delay. A full-custom RALUT design was also proposed, and while it yielded the best performance, the area utilization was greater than the ASIC RALUT design. The proposed designs can be used in the hardware implementation of ANNs.

7.2 Future Work

The proposed RALUT design requires a greater amount of delay as the number of input and output bits increases. By inserting registers in between address decode stages, this design could easily be modified to make use of pipelining in order to boost throughput.

Another research area is in determining the suitability of this RALUT architecture in more advanced technology nodes. The open nature of this design should easily allow minor design changes, such as the insertion of clock gating blocks to reduce leakage power in fabrication processes which are known to experience deep submicron effects.

Appendix A

Final Transistor Sizing

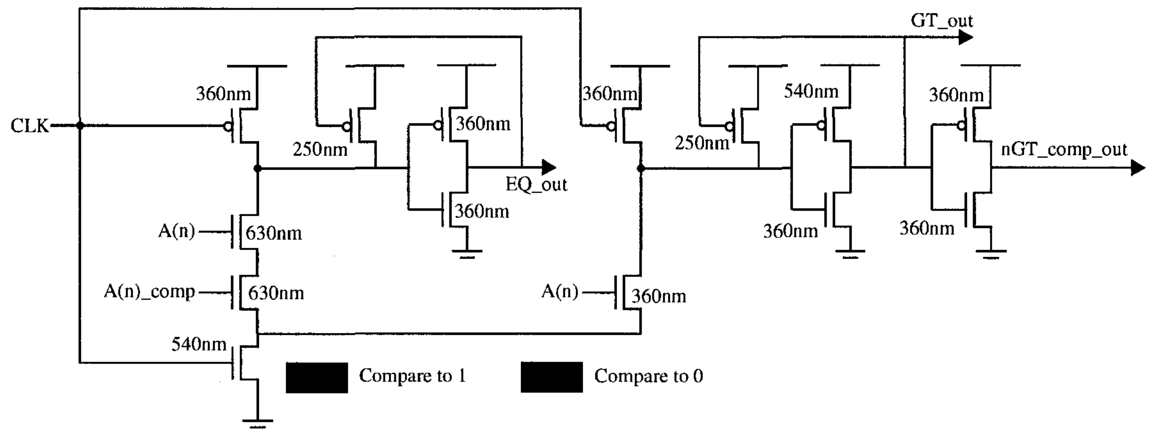

Figure A.1: Beginning Stage Final Transistor Sizing

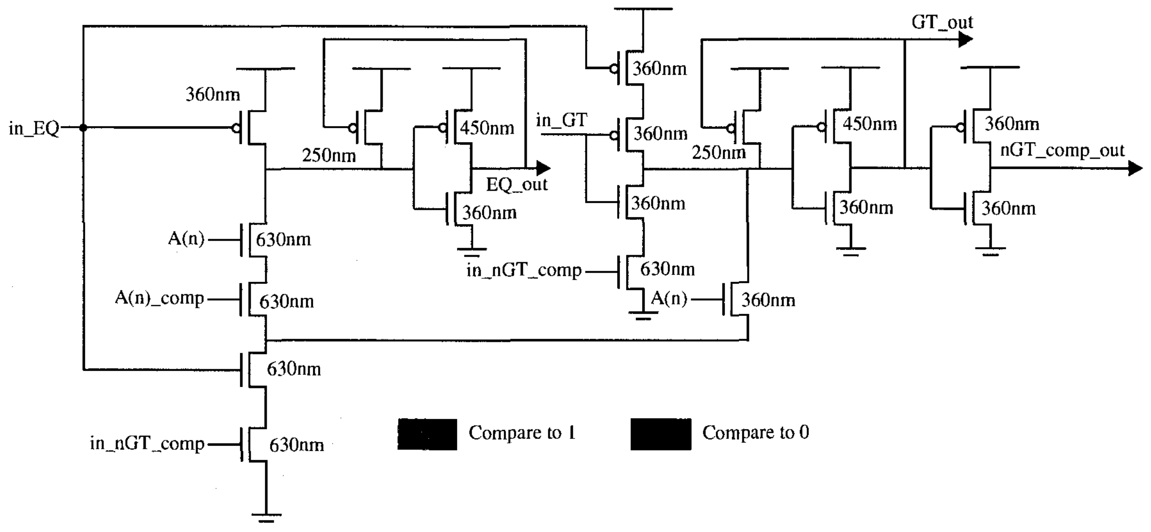

Figure A.2: Middle Stage Final Transistor Sizing

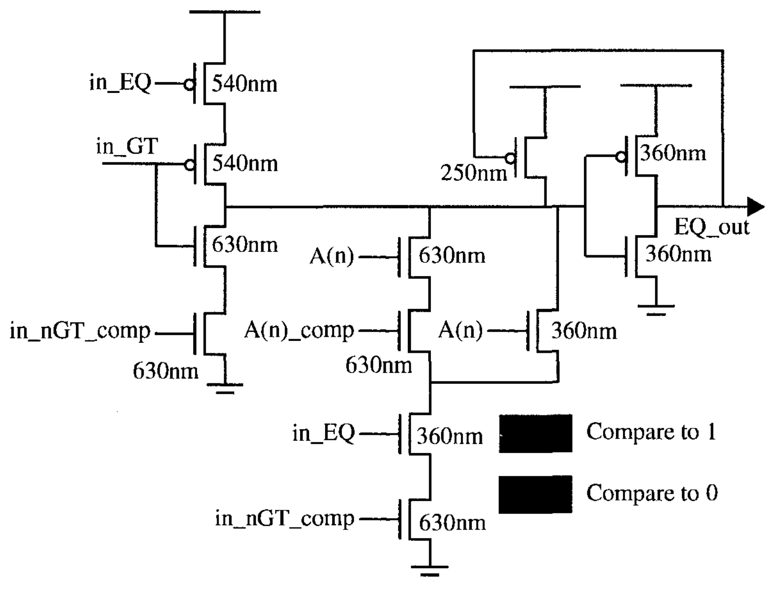

Figure A.3: Final Stage Final Transistor Sizing

Appendix B

Verilog Code

B.1 Verilog Modules

B.1.1 Automatic Test Pattern Generator

```
module atpg16 (clk, reset, seed, atpg_out);

        integer N;
        parameter [15:0] taps = 16'b 10000000_00010110;

        input clk, reset;
        input [15:0] seed;
        wire clk, reset;
        wire [15:0] seed;

        output [15:0] atpg_out;
        wire [15:0] atpg_out;

        reg bits, feedback;
        reg [15:0] lfsr_reg, next_lfsr_reg;

always @(negedge clk)
        begin
                if (reset)
                        lfsr_reg = seed;
                else
                        lfsr_reg = next_lfsr_reg;
        end

always @(lfsr_reg)
        begin
                bits = ~| lfsr_reg[14:0];
                feedback = lfsr_reg[15] ^ bits;
                for (N = 15; N >= 1; N = N - 1)
```

```
                        if( taps[N-1] == 1 )
                                next_lfsr_reg[N] = lfsr_reg[N-1] ^ feedback;
                        else
                                next_lfsr_reg[N] = lfsr_reg[N-1];
                next_lfsr_reg[0] = feedback;
        end

        assign atpg_out = lfsr_reg;

endmodule
```

B.1.2 Clock Wrapper

```
module clockwrapper ( ext_clk , reset , ctrl_decoder , ctrl_clk ,
ctrl_clk_out , clkgen , clk , clk_out , decoder_out);


input ext_clk , reset , clkgen;
input [2:0] ctrl_clk , ctrl_clk_out;
input [4:0] ctrl_decoder;
wire ext_clk , reset , clkgen;
wire [2:0] ctrl_clk , ctrl_clk_out;
wire [4:0] ctrl_decoder;

output clk , clk_out;
output [31:0] decoder_out;
wire clk , clk_out;
wire [31:0] decoder_out;

wire [5:0] counter;


decoder_n #(5, 32) DECODER0( .decoder_in(ctrl_decoder),
.decoder_out(decoder_out) );
counter_n #(6) COUNTER0( .clk(clkgen), .reset(reset),
.counter_out(counter) );
nto1_mux #(8, 3) MUX0( .mux_in({ext_clk , clkgen , counter}),
.select(ctrl_clk), .mux_out(clk) );
nto1_mux #(8, 3) MUX1( .mux_in({ext_clk , clkgen , counter}),
.select(ctrl_clk_out), .mux_out(clk_out) );


endmodule
```

B.1.3 Compare Module

```
module compare2 (ralut , lut , compare);

input [51:0] ralut , lut;
wire [51:0] ralut , lut;

output compare;
reg compare;

//assign compare = &(ralut ^~ lut);
always @(ralut or lut)
begin
        if (ralut == lut)
                compare = 1'b1;
        else
```

```
                compare = 1'b0;
end
endmodule
```

B.1.4 Controller

```
module controller (mcu_clk, reset ,ctrl_clk_out , ctrl_clk , ctrl_decoder , ctrl_reset ,
ctrl_clk_en , ctrl_en_a , ctrl_en_b , ctrl_atpg_seed , data_in , ctrl_datasel);

input mcu_clk, reset;
input [7:0] data_in;
wire mcu_clk, reset;
wire [7:0] data_in;

output [2:0] ctrl_clk_out , ctrl_clk;
output [4:0] ctrl_decoder;
output ctrl_clk_en , ctrl_en_a , ctrl_en_b;
output ctrl_reset;
output [15:0] ctrl_atpg_seed;
output [4:0] ctrl_datasel;

wire [2:0] ctrl_clk_out , ctrl_clk;
wire [4:0] ctrl_decoder;
wire ctrl_reset;
wire [15:0] ctrl_atpg_seed;
wire ctrl_clk_en , ctrl_en_a , ctrl_en_b;
wire [15:0] address_decode;

wire [4:0] ctrl_datasel;

//output [2:0] extra;
//wire [2:0] extra;

decoder_n #(4, 16) DATAREGS_DECODER( .decoder_in(data_in[7:4]),
.decoder_out(address_decode));
controller_mem #(4) MEM_00( .mcu_clk(mcu_clk), .reset(reset),
.write_enable(address_decode[0]), .data_in(data_in[3:0]),
.data_out({ctrl_clk_out , ctrl_clk_en}) );
controller_mem #(4) MEM_01( .mcu_clk(mcu_clk), .reset(reset),
.write_enable(address_decode[1]), .data_in(data_in[3:0]),
.data_out({ctrl_clk , ctrl_en_a}) );
controller_mem #(4) MEM_02( .mcu_clk(mcu_clk), .reset(reset),
.write_enable(address_decode[2]), .data_in(data_in[3:0]),
.data_out(ctrl_decoder[3:0]) );
controller_mem #(4) MEM_03( .mcu_clk(mcu_clk), .reset(reset),
.write_enable(address_decode[3]), .data_in(data_in[3:0]),
.data_out({ctrl_datasel[0], ctrl_reset , ctrl_en_b , ctrl_decoder[4]}) );
controller_mem #(4) MEM_04( .mcu_clk(mcu_clk), .reset(reset),
.write_enable(address_decode[4]), .data_in(data_in[3:0]),
.data_out(ctrl_datasel[4:1]) );
controller_mem #(4) MEM_05( .mcu_clk(mcu_clk), .reset(reset),
.write_enable(address_decode[5]), .data_in(data_in[3:0]),
.data_out(ctrl_atpg_seed[3:0]) );
controller_mem #(4) MEM_06( .mcu_clk(mcu_clk), .reset(reset),
.write_enable(address_decode[6]), .data_in(data_in[3:0]),
.data_out(ctrl_atpg_seed[7:4]) );
controller_mem #(4) MEM_07( .mcu_clk(mcu_clk), .reset(reset),
.write_enable(address_decode[7]), .data_in(data_in[3:0]),
.data_out(ctrl_atpg_seed[11:8]) );
controller_mem #(4) MEM_08( .mcu_clk(mcu_clk), .reset(reset),
.write_enable(address_decode[8]), .data_in(data_in[3:0]),
.data_out(ctrl_atpg_seed[15:12]) );

endmodule
```

B.1.5 n-bit Counter

```
module counter_n( clk, reset, counter_out);

parameter width = 6;
input clk, reset;
output [width - 1:0] counter_out;

wire clk, reset;
reg [width - 1:0] counter_out;

always @(posedge clk or posedge reset)
begin
        if (reset == 1)
                counter_out = 1;
        else
                counter_out = counter_out + 1;
end
endmodule
```

B.1.6 Data-out Selector

```
module data_out_select (sel, data_in, data_out);

        input [4:0] sel;
        input [255:0] data_in;
        wire [4:0] sel;
        wire [255:0] data_in;

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

        always @(sel or data_in)
                case (sel)
                        0 : data_out = data_in[7:0];
                        1 : data_out = data_in[15:8];
                        2 : data_out = data_in[23:16];
                        3 : data_out = data_in[31:24];
                        4 : data_out = data_in[39:32];
                        5 : data_out = data_in[47:40];
                        6 : data_out = data_in[55:48];
                        7 : data_out = data_in[63:56];
                        8 : data_out = data_in[71:64];
                        9 : data_out = data_in[79:72];
                        10: data_out = data_in[87:80];
                        11: data_out = data_in[95:88];
                        12: data_out = data_in[103:96];
                        13: data_out = data_in[111:104];
                        14: data_out = data_in[119:112];
                        15: data_out = data_in[127:120];
                        16: data_out = data_in[135:128];
                        17: data_out = data_in[143:136];
                        18: data_out = data_in[151:144];
                        19: data_out = data_in[159:152];
                        20: data_out = data_in[167:160];
                        21: data_out = data_in[175:168];
```

```
                    22: data_out = data_in[183:176];
                    23: data_out = data_in[191:184];
                    24: data_out = data_in[199:192];
                    25: data_out = data_in[207:200];
                    26: data_out = data_in[215:208];
                    27: data_out = data_in[223:216];
                    28: data_out = data_in[231:224];
                    29: data_out = data_in[239:232];
                    30: data_out = data_in[247:240];
                    31: data_out = data_in[255:248];
                    default :    data_out = data_in[7:0];
            endcase

endmodule
```

B.1.7 n-bit Decoder

```
module decoder_n( decoder_in , decoder_out );

parameter in_size = 5, out_size = 32;
input [in_size -1:0] decoder_in;
output [out_size -1:0] decoder_out;

reg [out_size -1:0] decoder_out;
integer i;

always @(decoder_in)
begin
        for (i = 0; i < out_size; i  = i + 1)
                if (decoder_in == i)
                        decoder_out[i] = 1;
                else
                        decoder_out[i] = 0;
                end
endmodule
```

B.1.8 Input Module

```
input_table[0]=16'b0000000000000000;
input_table[1]=16'b0000011000110111;
input_table[2]=16'b0000011111111000;
input_table[3]=16'b0000111001100001;
input_table[4]=16'b0001011011001100;
input_table[5]=16'b0001110110010001;
input_table[6]=16'b0010011001110101;
input_table[7]=16'b0010110110011011;
input_table[8]=16'b0011011011111111;
input_table[9]=16'b0011111010001101;
input_table[10]=16'b0100000010101110;
input_table[11]=16'b0100100001111000;
input_table[12]=16'b0101001010110010;
input_table[13]=16'b0101101011101011;
input_table[14]=16'b0110010110111000;
input_table[15]=16'b0110111001101000;
input_table[16]=16'b0111100111010000;
input_table[17]=16'b1000001011111101;
input_table[18]=16'b1000111100001001;
input_table[19]=16'b1001100010111010;
input_table[20]=16'b1001101101110110;
input_table[21]=16'b1010010101110100;
```

```
input_table[22]=16'b1011001010010011;
input_table[23]=16'b1011110100100000;
input_table[24]=16'b1100101011111100;
input_table[25]=16'b1101011000100001;
input_table[26]=16'b1110010011000100;
input_table[27]=16'b1111000010001010;
input_table[28]=16'b1111001111011100;
```

B.1.9 Memory Module

```
module main_mem (clk, reset, write_enable, data_in, data_out);

parameter size = 52;


input clk, reset, write_enable;
input [size-1:0] data_in;
wire clk, reset, write_enable;
wire [size-1:0] data_in;

output [size-1:0] data_out;
reg [size-1:0] data_out;

always @(negedge clk)
begin
        if(reset)
                data_out = 0;
        else if(write_enable)
                data_out = data_in;

end

endmodule
```

B.1.10 n-to-1 Multiplexer

```
module nto1_mux(mux_in, select, mux_out);

        parameter mux_size = 8;
        parameter select_lines = 3;
        integer i;

        input [mux_size - 1:0] mux_in;
        input [select_lines - 1:0] select;
        output mux_out;
        reg mux_out;

        always @(mux_in or select)
        begin
                mux_out = mux_in[0];
                for(i = 0; i < mux_size; i = i + 1)
                        begin
                                if(select == i)
                                        mux_out = mux_in[i];
                        end
        end
endmodule
```

B.1.11 n-wide n-to-1 Multiplexer

```
module data_out_select (sel , data_in , data_out);

        input [4:0] sel;
        input [255:0] data_in;
        wire [4:0] sel;
        wire [255:0] data_in;

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

        always @(sel or data_in)
        begin
                case (sel)
                        0 : data_out = data_in[7:0];
                        1 : data_out = data_in[15:8];
                        2 : data_out = data_in[23:16];
                        3 : data_out = data_in[31:24];
                        4 : data_out = data_in[39:32];
                        5 : data_out = data_in[47:40];
                        6 : data_out = data_in[55:48];
                        7 : data_out = data_in[63:56];
                        8 : data_out = data_in[71:64];
                        9 : data_out = data_in[79:72];
                        10: data_out = data_in[87:80];
                        11: data_out = data_in[95:88];
                        12: data_out = data_in[103:96];
                        13: data_out = data_in[111:104];
                        14: data_out = data_in[119:112];
                        15: data_out = data_in[127:120];
                        16: data_out = data_in[135:128];
                        17: data_out = data_in[143:136];
                        18: data_out = data_in[151:144];
                        19: data_out = data_in[159:152];
                        20: data_out = data_in[167:160];
                        21: data_out = data_in[175:168];
                        22: data_out = data_in[183:176];
                        23: data_out = data_in[191:184];
                        24: data_out = data_in[199:192];
                        25: data_out = data_in[207:200];
                        26: data_out = data_in[215:208];
                        27: data_out = data_in[223:216];
                        28: data_out = data_in[231:224];
                        29: data_out = data_in[239:232];
                        30: data_out = data_in[247:240];
                        31: data_out = data_in[255:248];
                  default : data_out = data_in[7:0];
                endcase
        end

endmodule
```

B.1.12 OK Signal Indicator

```
module ok(clk , reset , in , out);

input clk , reset , in;
wire clk , reset , in;

output out;
reg out;
```

```
always @(negedge clk)
begin
        if(reset)
                out = 1'b1;
        else
                out = in;
end

endmodule
```

B.1.13 Output Controller

```
output_table[0]=52'b0000000000000000000000110001101110000000111010000001011;
output_table[1]=52'b0000011000110111000000111111110000111010101010111 0100;
output_table[2]=52'b0000011111111000000011100110000110101000111010001111;
output_table[3]=52'b0000111001100001000101101100110000011110000111110011;
output_table[4]=52'b0001011011001100000111011001000111000001101001101110;
output_table[5]=52'b0001110110010001001001100111010100110110110111010010;
output_table[6]=52'b0010011001110101001011011001101111011010011001001101;
output_table[7]=52'b0010110110011011001101101111111101001111100110110001;
output_table[8]=52'b0011011011111111001111101000110111110011001000101100;
output_table[9]=52'b0011111010001101010000001010111001100100110110010101;
output_table[10]=52'b0100000010101110010010000111100010011000011010110000;
output_table[11]=52'b0100100001111000010100101011001000001101101000010100;
output_table[12]=52'b0101001010110010010110101110101110110001001010001111;
output_table[13]=52'b0101101011101011011001011011100000100110010111110011;
output_table[14]=52'b0110010110111000011011100110100011001001111001101110;
output_table[15]=52'b0110111001101000011110011101000000111111000111010010;
output_table[16]=52'b0111100111010000100000101111110111100010101001001101;
output_table[17]=52'b1000001011111101100011110000100101010111110110110001;
output_table[18]=52'b1000111100001001100110001011101011111011011000101100;
output_table[19]=52'b1001100010111010100110110111011001101101000110010101;
output_table[20]=52'b1001101101110110101001010111010010100000101010110000;
output_table[21]=52'b1010010101110100101100101001001100010101111000010100;
output_table[22]=52'b1011001010010011101111010010000010111001011010001111;
output_table[23]=52'b1011110100100000110010101111110000101110100111110011;
output_table[24]=52'b1100101011111100110101100010000111010010001001101110;
output_table[25]=52'b1101011000100001111001001100010001000111010111010010;
output_table[26]=52'b1110010011000100111100001000101011101010111001001101;
output_table[27]=52'b1111000010001010111100111101110001011100100110110110;
output_table[28]=52'b1111001111011100000000000000000001001000000101101 0001;
```

B.1.14 HDL Ralut Module

```
module ralut(ralut_in, ralut_out);

        parameter in_size = 16, out_size = 52, rows = 29;
        integer i;

//      input clk;
        input [in_size - 1:0] ralut_in;
        output [out_size -1:0] ralut_out;

        reg [out_size - 1:0] ralut_out;
        reg [in_size - 1:0] input_table [rows - 1:0];
        reg [out_size - 1:0] output_table [rows - 1:0];
/* for simulation, initialize the tables in "initial" block */
`ifdef SYNTHESIS
`else
        initial
        begin
`include "input.v"
`include "output.v"

        end
`endif
        always @(ralut_in)
        begin
/* For synthesis , initialize the tables in an always block */
`ifdef SYNTHESIS
`include "input.v"
`include "output.v"
`endif
        ralut_out = output_table[0];
                for(i = 0; i < rows; i = i + 1)
                begin
                        if( i < rows - 1)
                        begin

        if( (ralut_in >= input_table[i]) && (ralut_in < input_table[i+1]) )
                                        ralut_out = output_table[i];
                        end
                        else if ( i == rows - 1)
                                begin
                                if(ralut_in >= input_table[i])
                                        ralut_out = output_table[i];
                                end
                end
        end

endmodule
```

B.1.15 Test Circuit

```
module testblock2( clk, ctrl_clk_en, ctrl_en_a, ctrl_en_b, ctrl_reset, ok,
ralut_clk, ralut_atpg, ralut_v_atpg, ralut_out, ralut_v_out, ctrl_datasel,
data_out, ctrl_atpg_seed);

input clk, ctrl_clk_en, ctrl_en_a, ctrl_en_b, ctrl_reset;
wire clk, ctrl_clk_en, ctrl_en_a, ctrl_en_b, ctrl_reset;

input [63:0] ralut_out, ralut_v_out;
wire [63:0] ralut_out, ralut_v_out;

input [4:0] ctrl_datasel;
wire [4:0] ctrl_datasel;

input [15:0] ctrl_atpg_seed;
wire [15:0] ctrl_atpg_seed;

output ok, ralut_clk;
wire ok, ralut_clk;

output [15:0] ralut_atpg, ralut_v_atpg;
wire [15:0] ralut_atpg, ralut_v_atpg;

output [7:0] data_out;
wire [7:0] data_out;

wire [15:0] atpg;
wire [15:0] atpgp1, atpgp2, atpgp3;
wire [63:0] ralutl2 , ralutv2 , ralutl3 , ralutv3;

wire compared;

main_mem #(16) ATPG_L( .clk(clk), .reset(ctrl_reset), .write_enable(ctrl_en_a),
.data_in(atpg), .data_out(ralut_atpg) );
main_mem #(16) ATPG_V( .clk(clk), .reset(ctrl_reset), .write_enable(ctrl_en_b),
.data_in(atpg), .data_out(ralut_v_atpg) );
main_mem #(16) ATPG_P1( .clk(clk), .reset(ctrl_reset), .write_enable(1'b1),
.data_in(atpg), .data_out(atpgp1) );


main_mem #(16) ATPG_P2( .clk(clk), .reset(ctrl_reset), .write_enable(1'b1),
.data_in(atpgp1), .data_out(atpgp2) );
main_mem #(16) ATPG_P3( .clk(clk), .reset(ctrl_reset), .write_enable(1'b1),
.data_in(atpgp2), .data_out(atpgp3) );

toggle RALUT_CLK_TOGGLE( .clk_in(clk), .enable(ctrl_clk_en),
.clk_out(ralut_clk) );

main_mem #(64) RALUT_L2( .clk(clk), .reset(ctrl_reset), .write_enable(ok),
.data_in(ralut_out), .data_out(ralutl2) );
main_mem #(64) RALUT_V2( .clk(clk), .reset(ctrl_reset), .write_enable(ok),
.data_in(ralut_v_out), .data_out(ralutv2) );

main_mem #(64) RALUT_L3( .clk(clk), .reset(ctrl_reset), .write_enable(ok),
.data_in(ralutl2), .data_out(ralutl3) );
main_mem #(64) RALUT_V3( .clk(clk), .reset(ctrl_reset), .write_enable(ok),
.data_in(ralutv2), .data_out(ralutv3) );

data_out_select DATAOUT_SEL( .sel(ctrl_datasel),
.data_in({112'bx, ralutl3, ralutv3, atpgp3}), .data_out(data_out) );

compare COMPARE1( .ralut(ralutl2), .lut(ralutv2), .compare(compared) );

ok OK1(.clk(clk), .reset(ctrl_reset), .in(compared), .out(ok) );

atpg16 ATPG16_1( .clk(clk), .reset(ctrl_reset), .seed(ctrl_atpg_seed),
.atpg_out(atpg) );


endmodule
```

B.1.16 Power Toggle

```
module toggle( clk_in, enable, clk_out);

input clk_in, enable;
wire clk_in, enable;

output clk_out;
wire clk_out;

assign clk_out = (clk_in & enable);

endmodule
```

B.1.17 System Wrapper

```
// IO WRAPPER

`timescale 1ns / 10ps

module wrapper( mcu_clk_wrapper , ext_clk_wrapper , reset_wrapper , ok_wrapper ,
clk_out_wrapper , data_in_wrapper , data_out_wrapper );

input mcu_clk_wrapper , ext_clk_wrapper , reset_wrapper;
input [7:0] data_in_wrapper;
wire mcu_clk_wrapper , ext_clk_wrapper , reset_wrapper;
wire [7:0] data_in_wrapper;

output ok_wrapper , clk_out_wrapper;
output [7:0] data_out_wrapper;
wire ok_wrapper , clk_out_wrapper;
wire [7:0] data_out_wrapper;

wire mcu_clk , ext_clk , reset , ok, clk_out;
wire [7:0] data_in , data_out;

assembled2 U0( .ext_clk(ext_clk) , .mcu_clk(mcu_clk) , .data_in(data_in),
.reset(reset) , .data_out(data_out) , .ok(ok) , .clk_out(clk_out) );

PDIDGZ PAD_MCU_CLK( .PAD(mcu_clk_wrapper) , .C(mcu_clk) );
PDIDGZ PAD_EXT_CLK( .PAD(ext_clk_wrapper) , .C(ext_clk) );
PDIDGZ PAD_RESET( .PAD(reset_wrapper) , .C(reset) );
PDIDGZ PAD_DATA_IN_0( .PAD(data_in_wrapper[0]) , .C(data_in[0]) );
PDIDGZ PAD_DATA_IN_1( .PAD(data_in_wrapper[1]) , .C(data_in[1]) );
PDIDGZ PAD_DATA_IN_2( .PAD(data_in_wrapper[2]) , .C(data_in[2]) );
PDIDGZ PAD_DATA_IN_3( .PAD(data_in_wrapper[3]) , .C(data_in[3]) );
PDIDGZ PAD_DATA_IN_4( .PAD(data_in_wrapper[4]) , .C(data_in[4]) );
PDIDGZ PAD_DATA_IN_5( .PAD(data_in_wrapper[5]) , .C(data_in[5]) );
PDIDGZ PAD_DATA_IN_6( .PAD(data_in_wrapper[6]) , .C(data_in[6]) );
PDIDGZ PAD_DATA_IN_7( .PAD(data_in_wrapper[7]) , .C(data_in[7]) );


PDO08CDG PAD_OK( .I(ok) , .PAD(ok_wrapper) );
PDO08CDG PAD_CLK_OUT( .I(clk_out) , .PAD(clk_out_wrapper) );
PDO08CDG PAD_DATA_OUT_0( .I(data_out[0]) , .PAD(data_out_wrapper[0]) );
PDO08CDG PAD_DATA_OUT_1( .I(data_out[1]) , .PAD(data_out_wrapper[1]) );
PDO08CDG PAD_DATA_OUT_2( .I(data_out[2]) , .PAD(data_out_wrapper[2]) );
PDO08CDG PAD_DATA_OUT_3( .I(data_out[3]) , .PAD(data_out_wrapper[3]) );
PDO08CDG PAD_DATA_OUT_4( .I(data_out[4]) , .PAD(data_out_wrapper[4]) );
PDO08CDG PAD_DATA_OUT_5( .I(data_out[5]) , .PAD(data_out_wrapper[5]) );
PDO08CDG PAD_DATA_OUT_6( .I(data_out[6]) , .PAD(data_out_wrapper[6]) );
PDO08CDG PAD_DATA_OUT_7( .I(data_out[7]) , .PAD(data_out_wrapper[7]) );
endmodule
```

B.2 Verilog Test Benches

B.2.1 Compare Module Test Bench

```
module compare_tb;

reg [51:0] ralut , lut;
wire compare;

compare2 U0(.ralut(ralut) , .lut(lut) , .compare(compare) );
```

```
initial
begin
#0      $monitor( "ralut = %d, lut = %d, compare = %b", ralut, lut, compare);
#10     ralut = 0;
        lut = 0;

#20     ralut = 5;
#50     lut = 25;
#70     lut = 5;
end

endmodule
```

B.2.2 Clock Wrapper Test Bench

```
module clockwrapper_tb();


reg ext_clk, reset, clkgen;
reg [2:0] ctrl_clk, ctrl_clk_out;
reg [4:0] ctrl_decoder;

wire clk, clk_out;
wire [31:0] decoder_out;


clockwrapper U0( .ext_clk(ext_clk), .reset(reset), .ctrl_decoder(ctrl_decoder),
.ctrl_clk(ctrl_clk), .ctrl_clk_out(ctrl_clk_out), .clkgen(clkgen), .clk(clk),
.clk_out(clk_out), .decoder_out(decoder_out) );

initial
begin
$monitor("reset = %b, ext_clk = %b, clkgen = %b, ctrl_clk = %b,
ctrl_clk_out = %b, ctrl_decoder = %b, clk = %b, clk_out = %b, decoder_out = %b",
reset, ext_clk, clkgen, ctrl_clk, ctrl_clk_out, ctrl_decoder, clk,
clk_out, decoder_out);
#0      reset = 0;
        ext_clk = 0;
        clkgen = 0;
        ctrl_clk = 0;
        ctrl_clk_out = 0;
        ctrl_decoder = 0;

#30     reset = 1;
#10     reset = 0;
#1      ext_clk = 1;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 0;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 0;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 0;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
```

```
#1      ext_clk = 0;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 1;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 1;
        clkgen = 1;

#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 1;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 1;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 1;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 1;
        clkgen = 1;
#1      ext_clk = 0;
        clkgen = 0;
#1      ext_clk = 1;
        clkgen = 1;

#1      ctrl_decoder = 0;
#1      ctrl_decoder = 1;
#1      ctrl_decoder = 2;
#1      ctrl_decoder = 3;
#1      ctrl_decoder = 4;
#1      ctrl_decoder = 5;
#1      ctrl_decoder = 6;
#1      ctrl_decoder = 7;
#1      ctrl_decoder = 8;
#1      ctrl_decoder = 9;
#1      ctrl_decoder = 10;
#1      ctrl_decoder = 11;
#1      ctrl_decoder = 12;
#1      ctrl_decoder = 13;
#1      ctrl_decoder = 14;
#1      ctrl_decoder = 15;
#1      ctrl_decoder = 16;
#1      ctrl_decoder = 17;
#1      ctrl_decoder = 18;
#1      ctrl_decoder = 19;
#1      ctrl_decoder = 20;
#1      ctrl_decoder = 21;
#1      ctrl_decoder = 22;
#1      ctrl_decoder = 23;
#1      ctrl_decoder = 24;
#1      ctrl_decoder = 25;
#1      ctrl_decoder = 26;
#1      ctrl_decoder = 27;
#1      ctrl_decoder = 28;
#1      ctrl_decoder = 29;
```

```
#1      ctrl_decoder = 30;
#1      ctrl_decoder = 31;

end
endmodule
```

B.2.3 Controller Test Bench

```
module controller_tb();

reg mcu_clk, reset;
reg [7:0] data_in;


wire [2:0] ctrl_clk_out, ctrl_clk;
wire [4:0] ctrl_decoder;
wire ctrl_reset;
wire [15:0] ctrl_atpg_seed;
wire ctrl_clk_en, ctrl_en_a, ctrl_en_b;
wire [4:0] ctrl_datasel;

always #20 mcu_clk = ~ mcu_clk;

controller U0( .mcu_clk(mcu_clk), .reset(reset), .ctrl_clk_out(ctrl_clk_out),
.ctrl_clk(ctrl_clk), .ctrl_decoder(ctrl_decoder), .ctrl_reset(ctrl_reset),
.ctrl_atpg_seed(ctrl_atpg_seed), .ctrl_clk_en(ctrl_clk_en), .ctrl_en_a(ctrl_en_a),
.ctrl_en_b(ctrl_en_b), .data_in(data_in), .ctrl_datasel(ctrl_datasel) );

initial
begin
$monitor("output = %b %b %b %b %b %b %b %b %b, reset = %b, mcu_clk = %b,
data_in = %b", ctrl_atpg_seed, ctrl_datasel, ctrl_reset, ctrl_en_b,
ctrl_decoder, ctrl_clk, ctrl_clk_en, ctrl_clk_out, ctrl_clk_en, reset,
mcu_clk, data_in);
#0 mcu_clk = 0;
#0 reset = 0;
#0 data_in = 8'b0000_0000;
#40 reset = 1;
#40 reset = 0;
#40 data_in = 8'b0000_1111;
#40 data_in = 8'b0001_1111;
#40 data_in = 8'b0010_1111;
#40 data_in = 8'b0011_1111;
#40 data_in = 8'b0100_1111;
#40 data_in = 8'b0101_1111;
#40 data_in = 8'b0110_1111;
#40 data_in = 8'b0111_1111;
#40 data_in = 8'b1000_1111;
#40 data_in = 8'b1001_1111;
#40 data_in = 8'b1010_1111;
#40 data_in = 8'b1011_1111;
#40 data_in = 8'b1100_1111;
#40 data_in = 8'b1101_1111;
#40 data_in = 8'b1110_1111;
#40 data_in = 8'b1111_1111;

#40 data_in = 8'b1111_0000;
#40 data_in = 8'b1110_0000;
#40 data_in = 8'b1101_0000;
#40 data_in = 8'b1100_0000;
#40 data_in = 8'b1011_0000;
```

```
#40 data_in = 8'b1010_0000;
#40 data_in = 8'b1001_0000;
#40 data_in = 8'b1000_0000;
#40 data_in = 8'b0111_0000;
#40 data_in = 8'b0110_0000;
#40 data_in = 8'b0101_0000;
#40 data_in = 8'b0100_0000;
#40 data_in = 8'b0011_0000;
#40 data_in = 8'b0010_0000;
#40 data_in = 8'b0001_0000;
#40 data_in = 8'b0000_0000;


#40 reset = 1;
#40 reset = 0;
end
always #2000 $finish;
endmodule
```

B.2.4 OK Signal Test Bench

```
module ok_tb;

reg clk, reset, in;
wire out;

ok U0(clk, reset, in, out);

initial
begin
$monitor("clk = %b, reset = %b, in = %b, out = %b", clk, reset, in, out);
#0      clk = 0;
        reset = 0;
        in = 0;

#20     in = 1;
#20     reset = 1;
#20     reset = 0;
#20     in = 0;
#40     in = 1 ;
end
always #500 $finish;
always #5 clk = ~ clk;


endmodule
```

B.2.5 Test Circuit Test Bench

```
module testblock2_tb();
reg clk, ctrl_clk_en, ctrl_en_a, ctrl_en_b, ctrl_reset;

reg [63:0] ralut_out, ralut_v_out;

reg [4:0] ctrl_datasel;

reg [15:0] ctrl_atpg_seed;

wire ok, ralut_clk;
wire [15:0] ralut_atpg , ralut_v_atpg;

wire [7:0] data_out;

testblock2 U0( .clk(clk), .ctrl_clk_en(ctrl_clk_en), .ctrl_en_a(ctrl_en_a),
.ctrl_en_b(ctrl_en_b), .ctrl_reset(ctrl_reset), .ok(ok), .ralut_clk(ralut_clk),
.ralut_atpg(ralut_atpg), .ralut_v_atpg(ralut_v_atpg), .ralut_out(ralut_out),
.ralut_v_out(ralut_v_out), .ctrl_datasel(ctrl_datasel), .data_out(data_out),
.ctrl_atpg_seed(ctrl_atpg_seed) );


always #20 clk = ~ clk;

initial
begin
$monitor("clk␣=␣%b,␣reset␣=␣%b,␣data_out␣=␣%b,␣ralut_atpg␣=␣%b,␣ralut_v_atpg␣=␣%b,
ok␣=␣%b,␣ralut_clk␣=␣%b,␣ctrl_datasel␣=␣%b", clk, ctrl_reset, data_out, ralut_atpg,
ralut_v_atpg, ok, ralut_clk, ctrl_datasel);

#0      ralut_out = 3;
        ralut_v_out = 3;
        clk = 0;
        ctrl_reset = 0;
        ctrl_datasel = 0;
        ctrl_atpg_seed = 0;
#40     ctrl_reset = 1;
#40     ctrl_reset = 0;
#40     ctrl_en_a = 1;
        ctrl_en_b = 1;
#40     ctrl_reset = 1;
#40     ctrl_reset = 0;
#40     ctrl_clk_en = 1;
#40     ctrl_clk_en = 0;
#100    ctrl_clk_en = 1;
#1      ctrl_datasel = 0;
#1      ctrl_datasel = 1;
#1      ctrl_datasel = 2;
#1      ctrl_datasel = 3;
#1      ctrl_datasel = 4;
#1      ctrl_datasel = 5;
#1      ctrl_datasel = 6;
#1      ctrl_datasel = 7;
#1      ctrl_datasel = 8;
#1      ctrl_datasel = 9;
#1      ctrl_datasel = 10;
#1      ctrl_datasel = 11;
#1      ctrl_datasel = 12;
#1      ctrl_datasel = 13;
#1      ctrl_datasel = 14;
#1      ctrl_datasel = 15;
#1      ctrl_datasel = 16;
#1      ctrl_datasel = 17;
#20     ralut_out = 64'b11111111_00000000_11111111_00000000_11111111_
        00000000_11111111_00000000;
        ralut_v_out = 64'b00000000_11111111_00000000_11111111_00000000_
        11111111_00000000_11111111;
#60     ctrl_datasel = 0;
#1      ctrl_datasel = 1;
#1      ctrl_datasel = 2;
#1      ctrl_datasel = 3;
```

```
#1      ctrl_datasel = 4;
#1      ctrl_datasel = 5;
#1      ctrl_datasel = 6;
#1      ctrl_datasel = 7;
#1      ctrl_datasel = 8;
#1      ctrl_datasel = 9;
#1      ctrl_datasel = 10;
#1      ctrl_datasel = 11;
#1      ctrl_datasel = 12;
#1      ctrl_datasel = 13;
#1      ctrl_datasel = 14;
#1      ctrl_datasel = 15;
#1      ctrl_datasel = 16;
#1      ctrl_datasel = 17;
#20     ctrl_reset = 1;
#20     ctrl_reset = 0;
#20     ralut_out = 64'b01010101_01010101_01010101_01010101_01010101_01010101_
        01010101_01010101_01010101;
        ralut_v_out = 64'b10101010_10101010_10101010_10101010_10101010_10101010_
        10101010_10101010_10101010;

#60     ctrl_datasel = 0;
#1      ctrl_datasel = 1;
#1      ctrl_datasel = 2;
#1      ctrl_datasel = 3;
#1      ctrl_datasel = 4;
#1      ctrl_datasel = 5;
#1      ctrl_datasel = 6;
#1      ctrl_datasel = 7;
#1      ctrl_datasel = 8;
#1      ctrl_datasel = 9;
#1      ctrl_datasel = 10;
#1      ctrl_datasel = 11;
#1      ctrl_datasel = 12;
#1      ctrl_datasel = 13;
#1      ctrl_datasel = 14;
#1      ctrl_datasel = 15;
#1      ctrl_datasel = 16;
#1      ctrl_datasel = 17;

end
always #3000 $finish;

endmodule
```

B.2.6 Power Toggle Test Bench

```
module toggle_tb;

reg clk;
reg enable;
wire out;

toggle U0(.clk_in(clk), .enable(enable), .clk_out(out));

initial
begin
#0      clk = 0;
        enable = 0;
#10     enable = 1;
#10     enable = 0;
```

```
#40     enable = 1;
#200    enable = 0;
#300    enable = 1;
end
initial $monitor("clk␣=␣%b,␣enable␣=␣%b,␣out␣=␣%b", clk, enable, out);
always #10 clk = ~ clk;
always #1000 $finish;

endmodule
```

Appendix C

Matlab Code

C.1 Matlab .m Files

C.1.1 RALUT Point Generator

```
clc
clear all;
close all;

%0.033
%0.01568
eps = 0.25;
start = -8;
stop = 8;


acc = 0.0005;
x_cont = start:acc:stop;
y_cont = user_function(x_cont);


j=1;
x_ralut(j) = start;
y_ralut(j) = user_function(start);

for i=start:acc:stop
   if( abs(y_ralut(j) - user_function(i) ) >= eps )
        j = j + 1;
        x_ralut(j) = i;
        y_ralut(j) = user_function(i);
   end
end

x_ill = x_ralut;
```

```
y_ill = y_ralut;
%figure
plot(x_ralut, y_ralut, 'bo', x_cont, y_cont, 'r-')
%plot(x_cont, y_cont, 'k-', 'LineWidth', 5)
%     xlabel('x', 'FontSize', 20)
%     ylabel('tanh(x)', 'FontSize', 20)
%     grid on
%     axis([-8 8 -1.2 1.2])

error(1) = 0;
i = 1;
x_pos = start;
for j=2:1:size(x_cont,2)

    %get the "true" floating point y coordonate
    cont_y = user_function(x_pos);


    quant_y = y_ralut(i);

    error(j) = abs(cont_y - quant_y);


    if( i < size(x_ralut,2))
         if(x_pos + acc > x_ralut(i+1))
            i = i + 1;
        end
    end
    x_pos = x_pos + acc;
end

max_error = max(error);
mean_error = mean(error);

for i = 1:1:size(x_ralut,2) - 1
    delta_x(i) = x_ralut(i+1) - x_ralut(i);
end

lut_bits = ceil(log2((stop-start) / min(delta_x)))
x_ralut = x_ralut + stop;
x_ralut = x_ralut ./ 16 .* 2^lut_bits;
y_ralut = y_ralut + 1;
y_ralut = y_ralut ./2 .* 2^lut_bits;
a = dec2bin(x_ralut,lut_bits);
b = dec2bin(y_ralut,lut_bits);

for i=1:1:size(x_ralut,2)
    spacer(i,1) = '␣';
end


c = [a spacer b]
clear error;
x_lut_from_binary = bin2dec(a);
y_lut_from_binary = bin2dec(b);

x_lut_from_binary = x_lut_from_binary .* 16 ./ 2^lut_bits - stop;
y_lut_from_binary = y_lut_from_binary .* 2 ./ 2^lut_bits - 1;


error(1) = 0;
i = 1;
```

```
x_pos = start;
for j=2:1:size(x_cont,2)

    %get the "true" floating point y coordonate
    cont_y = user_function(x_pos);
    %get nearest x point in the table
  % previous = abs(x_pos - x_lut_from_binary(i) );
  % if( i < 2^lut_bits )
  %     next = abs (x_pos - x_lut_from_binary(i+1) );
  % end
  %
  %
    %if( previous <= next )
    %     nearest_x = i;
    %else
    %     if( i < 2^lut_bits)
    %         nearest_x = i+1;
    %     else
    %         nearest_x = i;
    %     end
    %end

    quant_y = y_lut_from_binary(i);

    error(j) = abs(cont_y - quant_y);


    if( i < size(x_lut_from_binary,1))
         if(x_pos >= x_lut_from_binary(i+1))
            i = i + 1;
        end
    end
    x_pos = x_pos + acc;
end

max_error = max(error);
mean_error = mean(error);
```

C.1.2 Sigmoid Function

```
function s = user_function(x)
  %s = 1 ./ ( 1 + exp(-x) );
  s = (exp(x) - exp(-x) ) ./ (exp(x) + exp(-x));
end
```

Appendix D

Layouts for the $0.35\mu m$***,*** $0.18\mu m$***, and High Performance*** $0.18\mu m$ ***Designs***

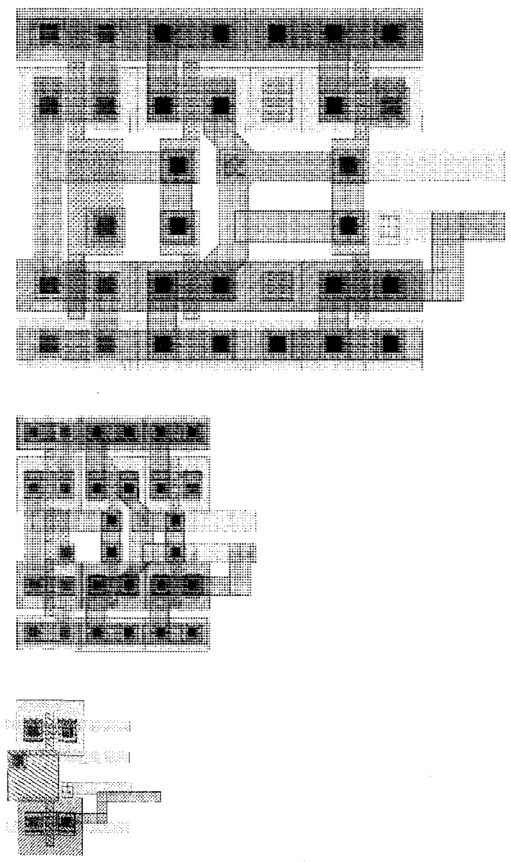

Figure D.1: Begin Address Decode Stage Layouts

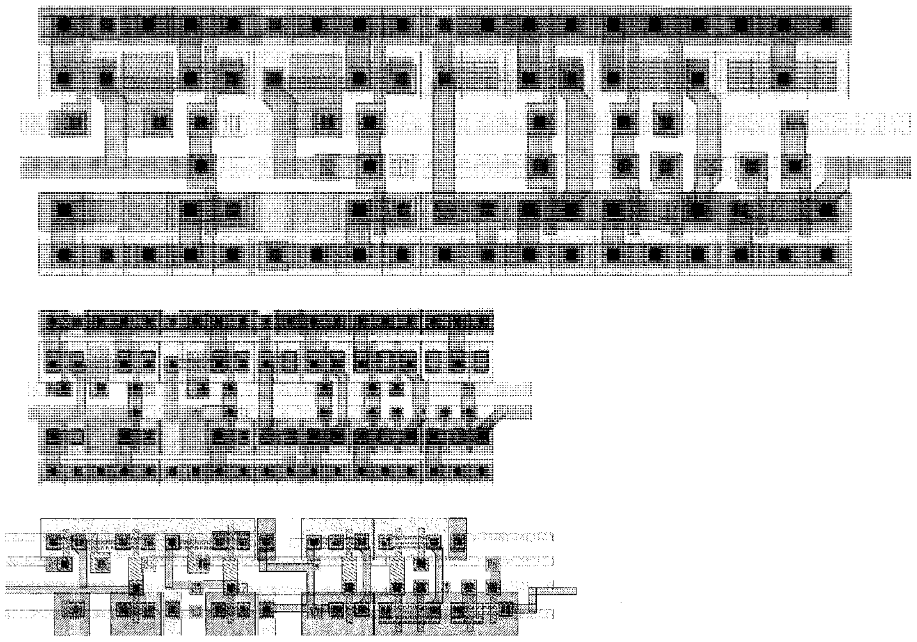

Figure D.2: Middle Address Decode Stage Layouts

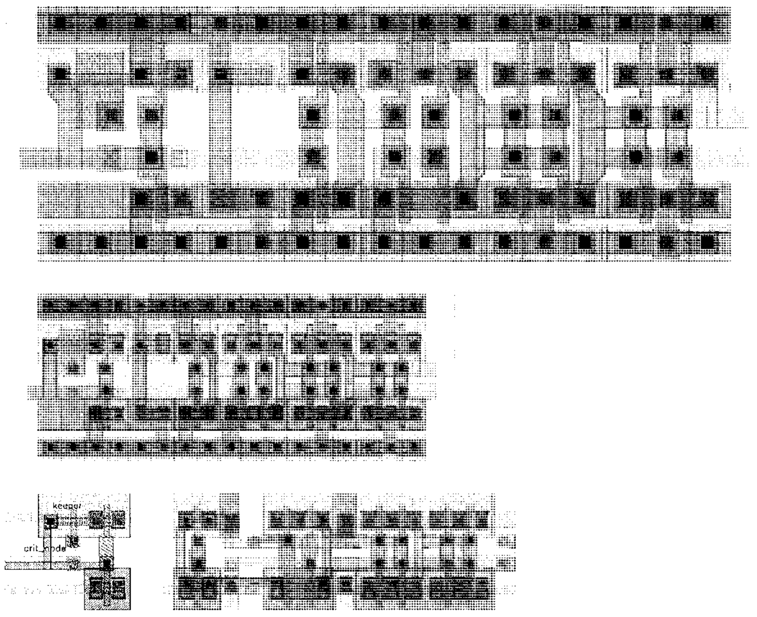

Figure D.3: Final Address Decode Stage Layouts

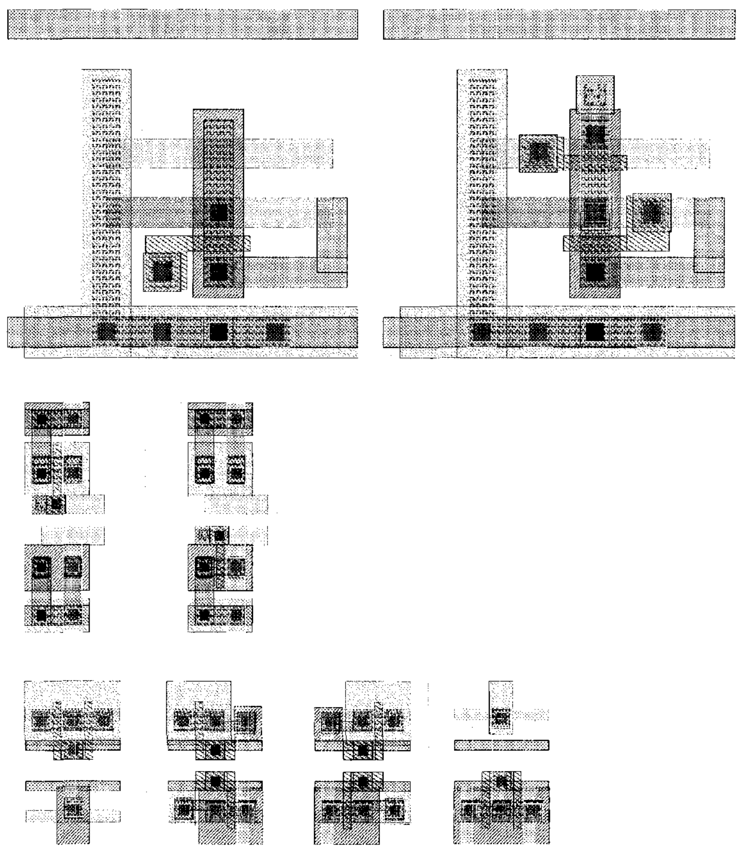

Figure D.4: Output Bits Layouts, First Row: '0', '1' Second Row: '0', 1' Third Row: '00', '01', '10', '11'

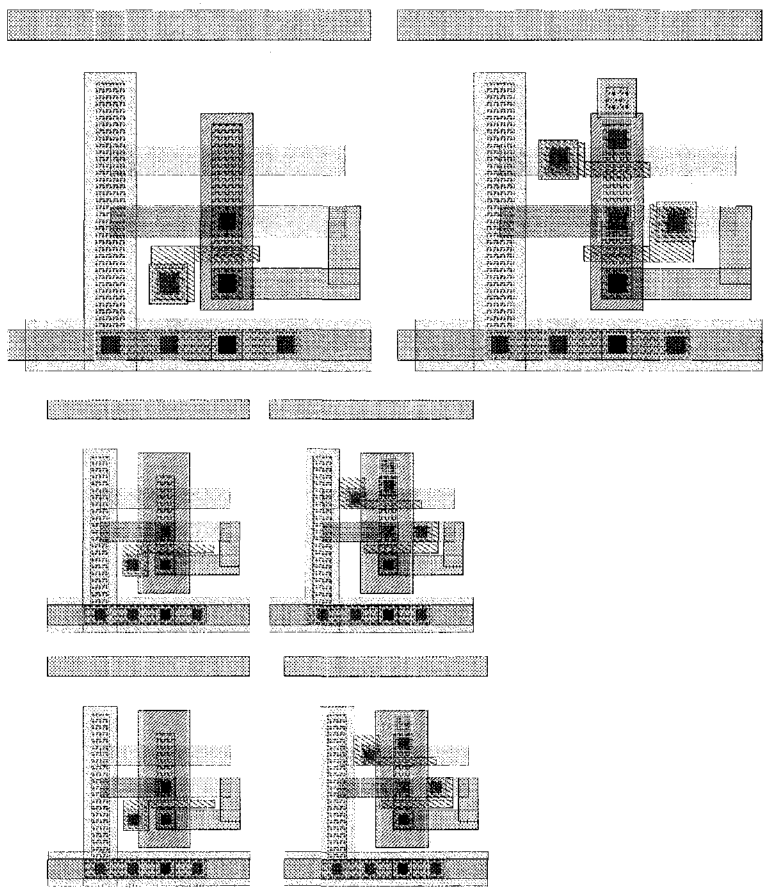

Figure D.5: Address Compare Bits From Left to Right: '0' and ''1

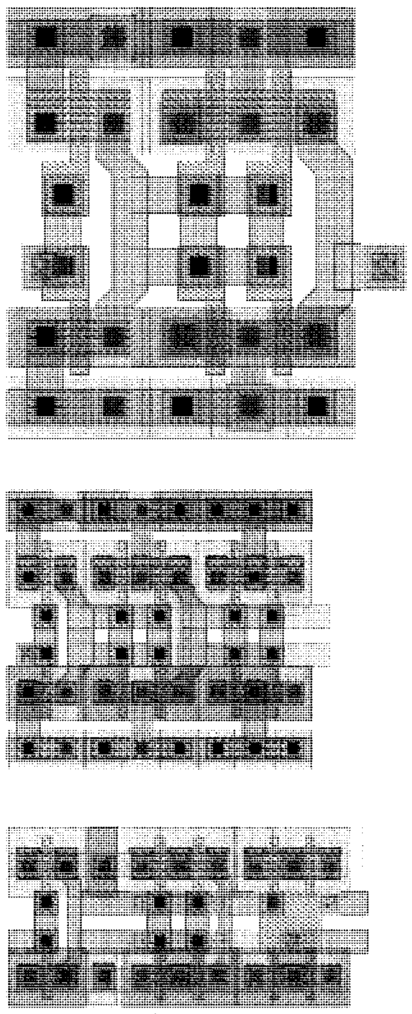

Figure D.6: Linedriver Layouts

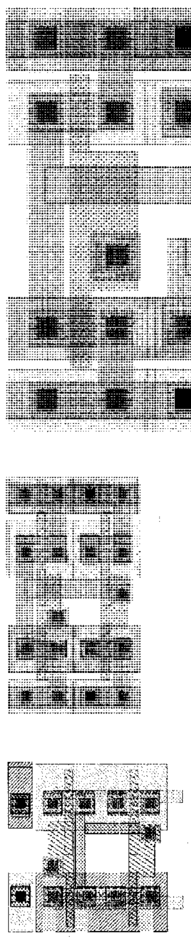

Figure D.7: Buffer Layouts

Appendix E

Synopsys Files

E.1 Verilog .v Files

E.1.1 Synopsys .dc Setup

```
set search_path "./src"
set search_path "$search_path + $synopsys_root + /libraries/syn"
set search_path "$search_path + $synopsys_root +
/CMC/kits/artisan/FE/fe_TSMCHOME_tpz973g_240c/digital/
Front_End/timing_power/tpz973g_240c +
/CMC/kits/artisan/FE/aci/sc/synopsys +
/CMC/kits/artisan/FE/aci/sc/symbols/synopsys"
set link_library "tpz973gwc.db slow.db dw_foundation.sldb *"
set target_library "tpz973gwc.db slow.db"
set synthetic_library "dw_foundation.sldb"
set symbol_library "tsmc18.sdb"

define_design_lib work -path work

set verilogout_no_tri  "true"
define_name_rules preview -allowed "A-Za-z0-9_"


set hdlin_enable_presto "false"
set hdlin_enable_vpp "true"

set hdlin_enable_vpp true
```

E.1.2 Clock Controller Script

```
analyze -format verilog {nto1_mux.v}
analyze -format verilog {decoder_n.v}
analyze -format verilog {counter_n.v}
analyze -format verilog {clockwrapper.v}
elaborate clockwrapper -architecture verilog
create_clock clkgen -period 2
link
uniquify
propagate_constraints
```

E.1.3 Test Circuit Script

```
analyze -library WORK -format verilog {toggle.v}
analyze -library WORK -format verilog {atpg16.v}
analyze -library WORK -format verilog {main_mem.v}
analyze -library WORK -format verilog {ok.v}
analyze -library WORK -format verilog {compare.v}
analyze -library WORK -format verilog {data_out_select.v}
analyze -library WORK -format verilog {testblock.v}
elaborate testblock -architecture verilog
create_clock clk -period 1.5
link
uniquify
propagate_constraints
compile -map high
report_timing
write -f verilog -out testblock_gates.v -hier
write_sdc testblock_gates.sdc
```

E.1.4 RALUT Wrapper Script

```
analyze -library WORK -format verilog {ralut3.v}
analyze -library WORK -format verilog {ralut2.v}
analyze -library WORK -format verilog {ralut.v}
analyze -library WORK -format verilog {ralut_wrap.v}
elaborate ralut_wrap -architecture verilog -library DEFAULT
create_clock -name "CK" -period 4 -waveform { "0" "2" } { "CK" }
set_max_dynamic_power 1.12e-6
compile -map high -power high
propagate_constraints
characterize ss
write -f verilog -out gates.v -hier
write_sdc gates.sdc
```

References

[1] C. Alippi and G. Storti-Gajani. *Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning.* 1991.

[2] M. Azarmehr. *A Multi-Dimensional Logarithmic Number System based Central Processing Unit.* MASc. Thesis,, University of Windsor, 2007.

[3] K. Basterretxea, J.M. Tarela, and I. Del Campo. *Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons*, volume 151. February 2004.

[4] V. S. Dimitrov and G. A. Jullien. *A New Number Representation with Applications*, volume Second Quarter. 2003.

[5] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. *Theory and Applications of the Double-Base Number System*, volume 48. October 1999.

[6] V. S. Dimitrov, G. A. Jullien, and K. Walus. *Digital Filtering Using the Multidimensional Logarithmic Number System*, volume 4791. December 2002.

[7] V. S. Dimitrov M. Ahmadi H. Li, G. A. Jullien and W. C. Miller. *A 2-digit multidimensional logarithmic number system filterbank for a digital hearing aid architecture*, volume 2. 2002.

[8] A. Hastings. *The Art of Analog Layout.* Pearson Prentice Hall, New Jersey, 2006.

[9] S. Haykin. *Neural networks : a comprehensive foundation.* Prentice Hall, July, 1998.

[10] S. O. Kasap. *Electronic Materials and Devices.* McGraw-Hill, New York, 2001.

[11] C. Lin and J. Wang. *A digital circuit design of hyperbolic tangent sigmoid function for neural networks.* May 2008.

[12] S. Vassiliadis M. Zhang and J. G. Delgrade-Frias. *Sigmoid generators for neural computing using piecewise approximations*, volume 45. August 1996.

[13] M. M. Mano. *Digital Design.* Prentice Hall, New Jersey, 2002.

[14] R. Muscedere. *Difficult Operations in the Multi-Dimensional Logarithmic Number System.* Ph.D. Thesis,, University of Windsor, 2003.

[15] R. Muscedere, V. Dimitrov, G. Jullien, and W. Miller. *A Low-Power Two-Digit Multi-dimensional Logarithmic Number System Filterbank Architecture for a Digital Hearing Aid*, volume 18. 2005.

[16] R. Muscedere, V. Dimitrov, G.A. Jullien, and W.C. Miller. *Efficient techniques for binary-to-multidigit multidimensional logarithmic number system conversion using range-addressable look-up tables*, volume 54. March 2005.

[17] R. Muscedere and K. Leboeuf. *A dynamic address decode circuit for implementing range addressable look-up tables.* May 2008.

[18] A. Pua P. Srivastava and L. Welch. *Issues in the Design of Domino Logic Circuits.* February 1998.

[19] F. Piazza, A. Uncini, and M. Zenobi. *Neural networks with digital LUT activation functions*, volume 151. February 2004.

[20] B. Razavi. *Design of Analog CMOS Integrated Circuits.* McGraw-Hill, New York, 2007.

[21] D.E. Rumelhart, J.L. McClelland, and the PDP Research Group. *Parallel Distributed Processing, Vol. 1: Foundations.* The MIT Press, July, 1987.

[22] M. A. Iachino S. Marra and F. C. Morabito. *High Speed, Programmable Implementation of a Tanh-like Activation Function and Its Derivative for Digital Neural Networks.* August 2007.

[23] D. J. Smith. *HDL Chip Design.* Doone Publications, Madison, AL, 2000.

[24] Sedra / Smith. *Microelectronic Circuits.* Oxford, New York, 2004.

[25] TSMC. Tsmc 0.35 um mixed signal polycide 3.3v/5v design rule. Product Specification TA-1098-4003 Rev. 2.2, Taiwan Semiconductor Manufacturing Co., LTD, July 1999.

[26] TSMC. Tsmc 0.18 um logic 1p6m salicide 1.8v/3.3v design rule. Product Specification T-018-LO-DR-001 Ver. 2.6, Taiwan Semiconductor Manufacturing Co., LTD, May 2006.

[27] S. Vassiliadis, Ming Zhang, and J.G. Delgado-Frias. *Elementary function generators for neural-network emulators*, volume 11. November 2000.

[28] Xilinx. Virtex-ii platform fpgas: Complete data sheet. Product Specification DS031 v3.5, Xilinx, Inc., November 2007.

VITA AUCTORIS

Karl was born in Windsor, Ontario, Canada. He received his Bachelor of Applied Science degree in Electrical Engineering from the University of Windsor in 2006. He is a student member of the IEEE, and is currently working towards a doctorate in Electrical Engineering at the University of Windsor. His primary research interests are VLSI design, analog design, cryptography, memory design, image processing and artificial neural networks.