Self-Time Circuit Size Optimization For An Input Data Distribution

Evelyn Sowells
North Carolina Agricultural and Technical State University

SELF-TIME CIRCUIT SIZE OPTIMIZATION FOR AN INPUT DATA DISTRIBUTION

by

Evelyn R. Sowells

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Department: Electrical and Computer Engineering
Major: Electrical Engineering
Major Professor: Dr. Alvernon Walker

North Carolina Agricultural and Technical State University
Greensboro, North Carolina
2012

ABSTRACT

Sowells, Evelyn R. SELF-TIME CIRCUIT SIZE OPTIMIZATION FOR AN INPUT DATA DISTRIBUTION. **(Major Professor: Dr. Alvernon Walker),** North Carolina Agricultural and Technical State University.

New design techniques with energy-delay characteristics that are superior to that of the synchronous timing and control approach are needed today because the throughput of systems realized with this method is limited by the power dissipation of nanometer scale devices and the power management strategies developed to insure that they do not exceed device thermal constraints. A circuit timing approach that is not dependent only on the propagation delay of the critical path is required to achieve this for a specified technology and supply voltage. Optimized self-timed circuits have this characteristic and therefore outperform synchronous designs for a given energy dissipation. A novel self-timed circuit device sizing approach that is based on the circuit input data distribution and circuit branching effort is proposed in this document. The analysis is based on the Logical Effort (LE). The LE model used in this work was extracted from SPICE simulation for the TMSC 0.18um process. The performance and energy dissipation of circuits implemented with this approach is 13% and 16% respectively better than circuits designed with previously proposed approaches.

School of Graduate Studies

North Carolina Agricultural and Technical State University

This is to certify that the Doctoral Dissertation of

Evelyn R. Sowells

has met the dissertation requirements of
North Carolina Agricultural and Technical State University

Greensboro, North Carolina
2012

Approved by:

Dr. Alvernon Walker Major Professor	Dr. Albert Esterline Committee Member
Dr. Corey Graves Committee Member	Dr. Clinton Lee Committee Member
Dr. Robert Li Committee Member	Dr. John Kelly Department Chairperson

Dr. Sanjiv Sarin
Associate Vice Chancellor for Research and Graduate Dean

DEDICATION

I dedicate my doctoral dissertation work to my loving family and friends. Particularly to my supportive and patient husband, who set the bar for excellence and always encourages me to pursue all of my dreams. I also dedicate this to our wonderful daughter, Maya who inspires me and has grown into a lovely young lady despite having to share me with my other child, research. I must also thank my loving parents Reece and Susan whom always believed in me, even when I did not believe in myself and to my dear friend Eureka, who has been my biggest cheerleader throughout this process. Finally, many thanks to my advisor, Dr. Alvernon Walker, who never gave up on me.

BIBLIOGRAPHY SKETCH

Evelyn R. Sowells was born on November 3, 1971, in Camden, New Jersey. She received the Bachelor of Science degree in Computer Science from North Carolina Agricultural and Technical State University in 2001 and a Master of Science degree from North Carolina Agricultural and Technical State University in 2003. She is a candidate for the Ph.D. in Electrical Engineering.

ACKNOWLEDGEMENTS

I would like to thank everyone who has been a “link in my chain” to helped make this dissertation possible. First, I would like to thank my advisor, Dr. Alvernon Walker, who has given me countless hours of guidance and support. His brilliance and research in Electrical Engineering has been a great inspiration for me. Additionally, I would like to thank my doctoral co-advisor and graduate advisor, Dr. Albert Esterline for suggesting that I pursue a doctorate degree in Electrical Engineering. That suggestion gave me the confidence I needed to start this academic journey. I would also like to thank my committee members Dr. Graves, Dr. Li and Dr. Lee for their helpful insight, recommendations, and comments.

Thank you to my two favorite chairs, Dr. Gloster and Dr. Kelly for providing invaluable support and direction whenever needed and James Griggs for sharing his priceless electrical engineering and Cadence knowledge. Also, many thanks to Juanda Johnson-Taylor who counseled me through my blood, sweat and tears as I climbed the doctoral mountain. Lastly, I would like to thank Greg Hampton who’s computer programming wizardry help me get through some really tough times.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Power conservation without performance penalties have become an increasingly important issue among modern digital circuit designers. As the digital technology evolution continues to produce more complex circuits coupled with ground breaking system performance, the power consumed by these circuits are at record highs. In fact, power dissipation or energy loss in the form of heat is reaching levels comparable to nuclear reactors. The negative affect associated with the power dissipation compromises or in many cases, impair chip reliability and life expectancy.

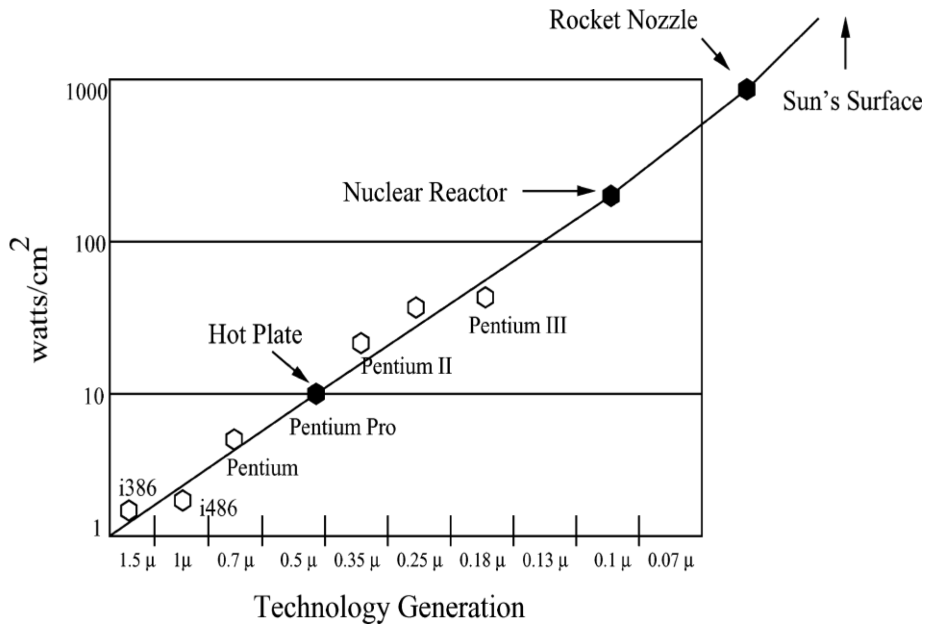


Figure 1.1: Power dissipation vs. scaling technology [2]

1.1 Motivation

Over the past decade, research in this area has eased but not solved this power issue. Many solutions involved increasing chip parameter size to ease the chips density that has lead us to this hot spot. However, as the demand for portable electronic devices rise, scaling technology forces us to deal with this problem. Figure 1.1 shows the power dissipation with respect to technology generation. As illustrated in figure 1.1, scaling technology increases, power dissipation or energy given off in the form of heat also increases. In 1985, Intel's i386 power dissipation was at a minimum less than 3 *watts*/ . However, by November 1995, the Pentium Pro was dissipating heat comparable to a hot plate at 10 *watts*/ . This, of course assuming that there are no power management techniques in place. Hibernation, a power down technique that deactivate idle components of the chip, multiple supply voltages and clock frequencies are the most used power management strategies.

1.2 Moore's Law

Gordon Moore, co-founder of Intel, made an observation in 1965 that the number of transistors per square inch on integrated circuits doubled every year since the integrated circuit was invented. He predicted that this trend would continue for the indefinite future. In the 21st century, the pace slowed down a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore's Law, which Moore himself has named. Figure 1.2 gives a graphical representation of

Moore’s Law. Most experts agree with Gordon Moore expecting this law to hold true for at least another two decades.

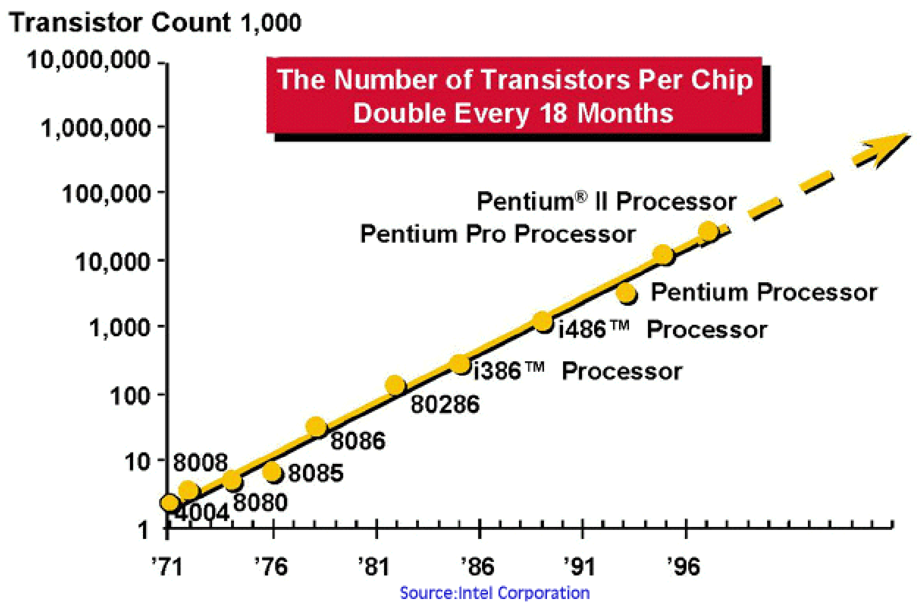


Figure 1.2: Moore’s Law [3]

Figure 1.1 shows the power dissipation increasing in a linear fashion and future technology generations could possibly dissipate power of that comparable to a nuclear reactor. The consumption of heat per is increasing as device scaling increases. This further reinforces the fact that we need more power efficient designs. Remarkably enough, if research does not produced a technique to break through the “power wall”, advancements in circuit technology will have reached its limits because the techniques that are used today may not be effective ten years from now.

1.3 Energy Delay Product

When we consider the energy or power with respect to performance from the prospective of a gate, there are several challenges. As Moore's Law continues to hold, the number of transistors on a chip will double every 18 months, the increasing clock frequencies and chip density have allowed designer to create more desirable architectures which run applications at ground breaking speeds. However, the micro-architecture and logic designs are stressed as frequency has increased faster than scaling. Since clock frequency is a linear function of power dissipation, as we increase the frequency we also increase the power dissipation. Further reducing the number of gate delays per cycle will also be difficult to achieve because the interconnect parasitics associated with the wires of a circuit are starting to dominate the speed or performance of the circuit not the gate. There are several problems that have to be resolved to build faster and more efficient chips: better chip implementation design techniques, better clock system design strategies and a more efficient micro-architecture.

As we increase the supply voltage, the delay of the gate decreases. However, the power dissipation increases, as well. This is called the energy delay product. One of the measures of efficiency for a digital system is the energy delay product, propagation delay multiplied by energy dissipation which is measured in joules. There have been several papers that investigate techniques that explore the possibilities of optimizing the power delay product more in depth [24, 25, 26, 27].

Modern digital designers, most often use synchronous logic to build computer systems because this logic style is more commonly accepted due largely in part to the commercial infrastructure which has already become acclimated. Traditional synchronous system designers often believe that the in order to boost performance one must pay a power penalty or vice versa, which is consider power/performance tradeoff.

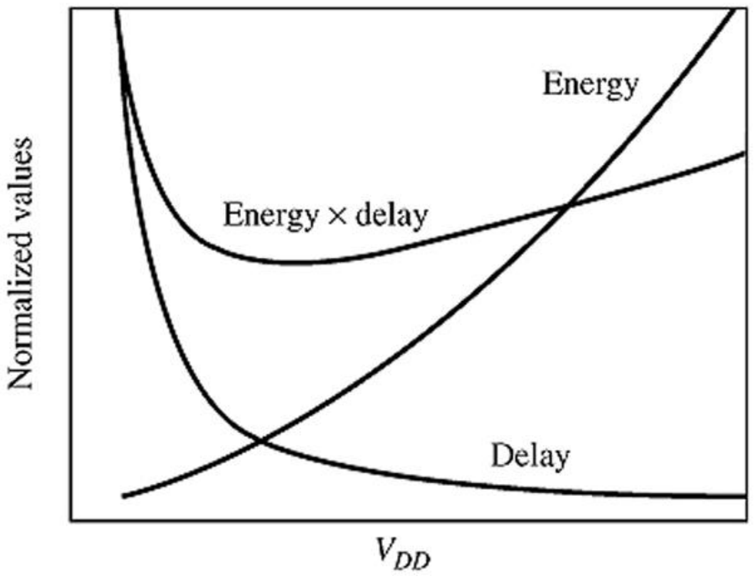
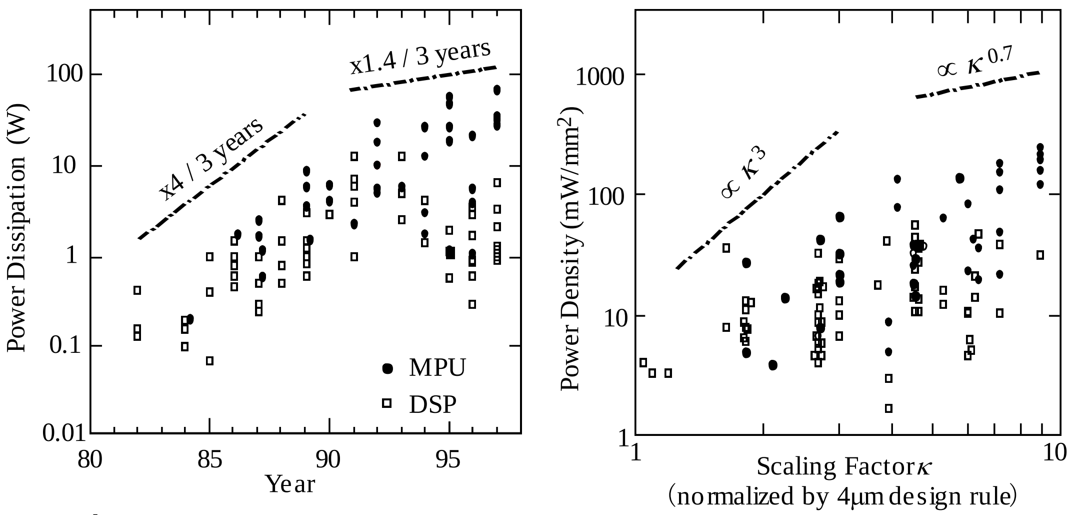


Figure 1.3: Energy Delay Product [4]

Gonzalez and Horowitz demonstrate that the architectural improvements contribute the most to both performance and energy efficiency. For example, their results demonstrate that pipelining is of fundamental importance to processor performance and energy efficiency, but super scalar issue is a lesser contribution [8].

Figure 1.3 shows the ideal energy delay product. Our challenge here is to figure out how to build a gate that is fast and power efficient. Can we increase performance without increasing power dissipation? Is it possible to have a superior power delay product?

Current trends suggest that we can. Let’s take a look at some Multi-processor units (MPS) and Digital Signal Processors (DSP) which are typically the highest performing chips.



(a) Power dissipation vs. year.

(b) Power density vs. scaling factor.

Figure 1.4: Power Dissipation with respect to year and scaling factor [3]

In figure 1.4a, as Moore’s Law remains true, more transistors are packed into a small chip. The initial affects on power dissipation is increasing most rapidly in the 1980’s. This is due in part to technology that was not as power efficient as today’s technology. In the early 80’s the transistor sizes were much larger and the circuit operated at a lower clock frequency. The Intel’s 8088 operated at 4.77 mhz as opposed to today’s personal computers that can operate at 2 Ghz which is about 400 times faster. Then in the early 1990’s as more power efficient architectures were introduced, (e.g.

RISC, pipelining, super scalar, and branch prediction) power dissipation still increased but at a much slower rate. This is demonstrated by the difference in the data lines in the figure 1.4a. The first data line shows a four times increase in power dissipation every three years. While the second data line shows a 1.4 times increase in power dissipation every three years.

The same hold true with respect to scaling, figure 1.4b. As device sizes decrease, the intrinsic time constant is reduced which implies that clock frequencies and power dissipation increase. However, the more efficient architectures had the same effect on the slope to the data line. It did not increase as rapidly. This demonstrates that by building architectures that are more efficient, we get an energy penalty that is less. Furthermore, it is possible to build such systems and that different design strategies can deliver a superior energy delay product. In short, performance is constrained by power. Design choices affect the power efficiency of a circuit and can offer something more in terms of performance. By developing a circuit with a better energy delay product, we can achieve better performance per joule, which gives us new possibilities. One example would be for portable devices, the battery life can be increased and applications can run as long as possible. My goal is to build a system that yields more performance per joule.

1.4 Research Contribution

The central focus of digital system design engineers over the past two decades has been on the trade-offs between the power/energy and performance of the circuits implemented in current and emerging nanometer-scale VLSI technologies. A number of techniques have been developed to address this design challenge; one approach is based on a class of asynchronous pipelined digital circuit structures that are called self-timed [4]. The dynamic power/energy dissipation is reduced in this realization, relative to synchronous implementations, because all clocks are generated locally and circuit timing and control is event driven. The performance of these circuits can exceed synchronous realization because it is based on the average intrinsic timing of the circuit instead of its worst case timing that is used to set the clock frequency in synchronous systems. The circuit design process used to determine the device sizing in self-timed circuits/systems is typically the same as that used for synchronous realizations [6, 7, 8]. However, the input distribution is not considered in this process. A novel self-timed circuit design technique that out performs previously proposed approaches is presented in this dissertation. The input data distribution is used in the proposed technique to optimize the circuit performance for the respective input data set probability distribution.

CHAPTER 2

BACKGROUND

2.1 Power vs. Energy

Energy is related to the total amount of work a system performs over a period of time, while power is the rate at which the computer consumes electrical energy or dissipates it in the form of heat while performing that work. In other words,

$$P = W/T \qquad (1)$$

$$E = P*T \qquad (2)$$

where P = power and is measured in watts, E = Energy and is measured in joules, T = specific time interval in seconds, W= total work performed in each interval [9]. In many cases, energy and power are used interchangeably but as pointed out earlier, they are distinctly different. This is particularly important for system designers because techniques that reduce power do not necessarily reduce energy. Venkatachalam gives an example. The power consumed by a computer may be reduce by halving the clock frequency, but if the computer takes twice as long to run the program, the total energy consumed will be similar. He also states that in some instances, the system designers chose which reduction is most important. For example, when designing for mobile application, energy is more important because of the desire to increase battery life. In other instances, like building mainframes, the temperature is more important because the thermal properties limits, the reduction of instantaneous power is paramount.

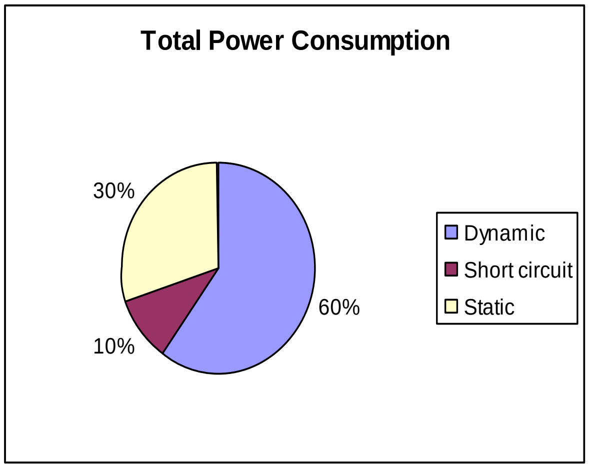


Figure 2.1: Power Dissipation Breakdown of Circuit

There are three sources of power dissipation in digital CMOS circuits which are summarized in this equation: $P_{avg} = P_{switching} + P_{short-circuit} + P_{static}$, where P_{avg} is the total power dissipation and $P_{switching}$ refers to the switched capacitance, power associated with switching circuit gate capacitance. $P_{short-circuit}$ is the circuit power that is due to the direct path current, which arises when both the NMOS and PMOS transistors are simultaneously active. Lastly, P_{static} represents static power dissipation stemming from the leakage current. Figure 2.1 shows the power dissipation breakdown of a circuit.

2.2 Static Power Dissipation

Keeping in mind that leakage current flow from every transistor that is powered on, with increasing die sizes and integration; static power will become a significant part of the total power consumption. The equation for static power dissipation is

$P_{leakage} = V_{dd} N k_{design} I_{leak}$ where N is the number of transistors, K_{design} is a design dependent feature, like the number of transistors on at any time and I_{leak} is a technology dependent characteristic design like threshold voltage.

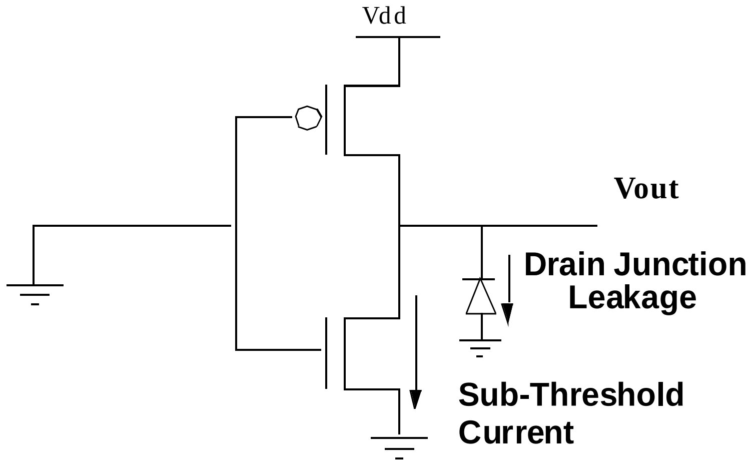


Figure 2.2: Static Power Dissipation

2.2.1 Power management Strategies for Static Power Dissipation

Figure 2.2 gives an illustration of the leakage current which has several components: Reverse biased pn junction—the diode leakage that occurs when a transistor is turned off, and sub-threshold leakage which occurs when the gate source voltage has exceeded the weak inversion point but is still below the threshold voltage. The aforementioned are the most important components of leakage currents. Gate induced drain leakage, punch through and gate tunneling are the other components of leakage currents. Most popular techniques for reducing static power dissipation are: (1) Reduce circuit size which decreases total power consumed by dynamically cutting power of idle components. The major disadvantage is unpredictable, overhead for clock gating. (2)

Reduce temperature which decreases sub-threshold leakage. The circuit is faster because lower temperatures have less resistance. It increases the life expectancy of the chip but is more expensive to build. (3) Increase threshold voltage which causes the sub-threshold leakage current drops exponentially.

2.3 Dynamic Power Dissipation

The circuit power associated with switching circuit device capacitance and short-circuit are two components of dynamic power dissipation in a digital CMOS circuit. Figure 2.3 gives an illustration of dynamic power dissipation. Switched capacitance is the largest component of total power consumed accounting for sixty percent of power used. As capacitors charge and discharge at the output of the circuit, electrical energy is used and heat is given off.

The equation for dynamic power dissipation is $P_{dyn} = \alpha C_l Vdd^2 f$ where, α is the activity factor of a system, C_L is the total load capacitance, *Vdd* is the supply voltage, and is the operation frequency. The reduction of one or more of the previous factors is needed to lower power dissipation of a system.

2.3.1 Power management Strategies for Dynamic Power Dissipation

There are four methods to reduce this type of power loss. The first method is to reduce the physical capacitance or stored electrical charge of a circuit. This can be done by changing design parameters: reducing the size of transistors and wires, layout optimizations where signals that have high switching activity assigned to short wires and signals that have low switching activity assigned to longer wires. However, the designer must deal with the risk of reducing system performance.

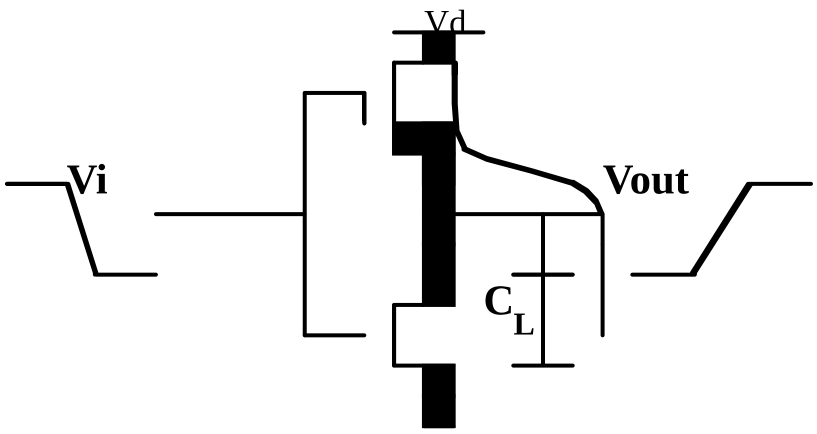


Figure 2.3: Dynamic Power Dissipation

The second method is reducing the switching activity. One approach is Algorithmic Optimization, which includes Technology Mapping that minimizes the number of operations by using a genetic algorithm to find an energy efficient way to arrange gates & signals. Architecture Optimization is another approach which uses clever glitch-free circuits which also includes transistor reordering. Logic gate restructuring focuses on the circuit’s topology (Tree vs. Chain) and uses path balancing, shorter logic depth, and fewer spurious transitions. Clock gating, power down or hibernation is also a popular technique that the Pentium 4 uses to reduce switching activity which stops the clock signal from reaching idle functional unit. This approach is advantageous because the clock network consumes a lot of power. One disadvantage is the latency involved with starting functional unit back up. However, it is inherent to self-timed logic since power consuming transitions only occur when requested. The circuit optimization technique examines Dynamic Logic which has fewer transistors, N + 2 as opposed to 2N for its counterpart, faster switching speeds and no short circuit or spurious transitions, while Static Logic has no pre-charge or power downs and low level of complexity to build. Synchronous circuits are more commonly accepted in the computer industry and easier the build. However, the maximum performance is not achieved since the clock runs at worst case in critical path and larger circuits have to overcome the clock skew problem. Asynchronous logic are low power, generally faster, has average case performance, immunity from meta-stable states, only critical path is optimized and idle functional units decreases dynamic power consumption, as well. On the down side, extra overhead is needed for completion signal.

The third method for reducing power loss is reducing the clock rate. If the frequency is reduced, less power dissipates and parallel architectures and/or pipelining is introduced to increase performance. The tradeoff to consider is a more complex circuit, slower performance, and larger silicon area. Reducing clock frequency will lessen the system performance and should only be used for applications where speed is not a top priority. The final method used to reduce power dissipation of a circuit is to reduce voltage supply. This technique increases gate delays which are offset by a slower clock frequency to allow the circuitry to work properly. The disadvantages are worsening performance by increasing gate delay, which may cause erroneous data and if delay is too long, data hazards are introduce.

2.3.2 Short Circuit Power Dissipation

The second source of dynamic power loss is short-circuit current which account for about 10% of total power consumed and is illustrated in figure 2.4. It is defined as $P_{short-circuit} = I_{SC}Vdd$ where I_{SC} is short circuit current and *Vdd* is supply voltage. Figure 8 illustrates the short circuit current. During the switching of a transistor, there is a brief moment when both the NMOS and PMOS are simultaneously on which creates a short circuit for the source to the ground. This particular area of power loss has had the least amount of progress for several reasons. The amount of power that is lost is so small that

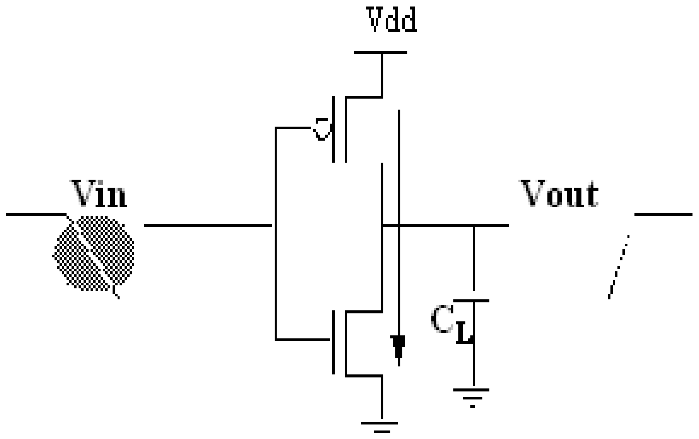


Figure 2.4: Short Circuit Power Dissipation

it is almost neglectible and current research has not found a way to reduce it that without significantly reducing the performance of the transistor. One rule of thumb that keeps this power loss at a minimum is to insure that the rise and fall time of the transistor gates are equal.

2.4 High Speed Digital System Realization

There are several techniques that system designers use to boost performance. Perhaps the most popular techniques is parallelism, operations are carried out simultaneously or concurrently. It is the backbone of high performance computing. The theory behind it is the more work that a system is able to do per clock cycle, the energy consumed in not going to be as great. One type of parallelism is Multi-core architectures which boost performance and minimizes heat output by integrating two or more processor cores in a single processor socket. Intel has a 50-Core processor named Knights Corner which is a super computer at University of Texas at Austin used for research. Pipelining is the most popular performance enhancement technique that increases the throughput of a system by processing data in stages like an assembly line. Superscalar in very similar to pipelining but it deals with instruction level parallelism that issues multiple instructions multiple data (MIMD). Multithreading is used to run multiple threads on the hardware at one time.

Another performance enhancement technique is to reducing the data execution time. This is mainly done by having a high clock frequency. Since power dissipation is a linear function of the clock frequency, it is also increased. Clock skew can also be introduced where the clock signal reaches different components at different times. As the clock rate of a circuit increases, timing becomes more critical and less variation can be tolerated if the circuit is to function properly. Single-Cycle Instruction Set Architecture also helps to reduce the execution time of data. Reduced Instruction Set Architecture

(RISC) operates on a fixed length instruction and the hardware is simple, fast and uses less energy. The truth is that modern digital system designers use a combination of all of the aforementioned techniques to boost performance.

2.5 Circuit Design Methodologies

When building a circuit, designers must choose a methodology that compliments the circuit's logic and system design. For the most part, circuit designs in the industry are built with synchronous logic; small blocks of combinatorial logic separated by synchronously clocked registers. Figure 2.5 gives an illustration of a synchronous system. As its name suggest, synchronous circuits use a clock to synchronize each transition. In other words, change in the circuit happens at the same rate and occur at the same time. The biggest advantage of this logic style is the ease in determining the maximum clock frequency of a design by finding and calculating the longest delay path between registers in a circuit. Another advantage of synchronous design is hazard avoidance. Static logic can introduce hazards through spurious transitions meaning that some flip flops have internal meta-stable transition before the settle to their final logic. If the signal is used before the final logic state, the wrong signal may be forwarded. Synchronous logic eliminates this hazard because the clock insures that these glitches have been worked out before transitioning to next state. One major disadvantage of synchronous design is the unused clock cycle time. Even if the gate has finished transitioning, the signal cannot go to the next state until the clock signals the transition. More power is used because the clock uses energy whether gates transition or not. Clock skew is another problem that synchronous systems encounter. This is the difference in time that the clock signal arrives throughout the circuit. It is even further exaggerated as we scale systems because wire delay does not scale the same as transistor switching speed.

Since synchronous systems have dominated the circuit design industry, there are a small number of available CAD tools for design, simulation and testing of asynchronous circuits. However, as the semiconductor industry wrestle with mounting problems trying to achieve higher performances and lower power consumption without significant

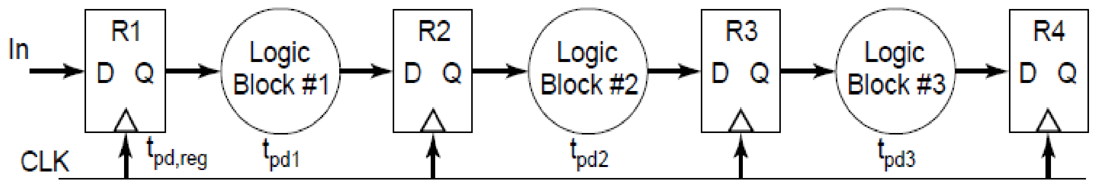


Figure 2.5: Synchronous Three Stage Pipeline [4]

increases in fabrication costs, developers are turning to asynchronous alternatives to solve these problems. Over the past few years, universities and established asynchronous companies have focused their research on developing Electronic Design Automation (EDA) tools and design flows that can be integrated into the custom and semi-custom methods now used by the industry for synchronous design. This paradigm shift has opened the door for unprecedented advances in the circuit design industry. [20, 21, 22, 23] all investigate the possible benefits of self-timed system design. Asynchronous logic works extremely well on power dissipation reduction. At 40% activity, an asynchronous system will dissipate 50% less power than its synchronous counterparts [2].

Asynchronous circuits have several other possible benefits. No clock skew – the difference in arrival time of clock signals to different parts of the circuit. Since asynchronous circuits have no clock, there is no clock skew. Speed is another area where these circuits shine. The timing of an asynchronous circuit depends on the structure of the transistor network, the delay of its signals and the length of the signal paths. Worst case performance of traditional synchronous systems is replaced by average case since performance is dependent on only the current active path. Better technology migration potential and automatic adaptation to physical properties- fabrication, temperature and power supply voltage.

Modern synchronous digital systems are limited by power dissipation of nanometer scaled devices and power management strategies developed to insure that they do not exceed circuit thermal constraints. Traditional optimization techniques are base on synchronous digital systems that use a global clock network which consume a considerable amount of the systems power. 50% of Dynamic Power is consumed by clock circuitry [11]. Furthermore, significant power can be wasted in transitions within blocks, even when their output is not needed. Global clock signals are particularly affected by scaling technology in that the long interconnect wires have increasing different times which must be manage to produce valid output. System designers have dealt with the power challenges by clock gating, which saves power by adding logic gates to a circuit in order to disable, portions of the clock tree when not needed. Even though clock gating reduces the power dissipation, it is more effectively implemented on a macro level as opposed to the circuit level.

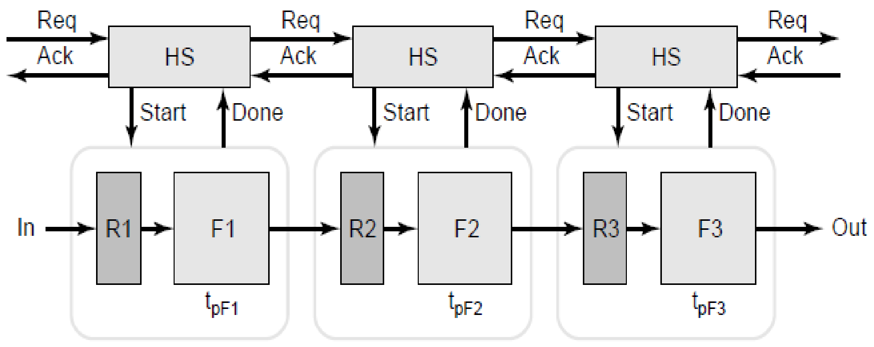


Figure 2.6: Asynchronous Three Stage Pipeline [4]

The handshake protocol shown in figure 2.6 regulates the flow of information through the self-timed pipeline. Input arrives and a Request to F1 is raised. If F1 is inactive, it transfers the data and acknowledges this fact to the input buffer which can then fetch the next input. Next F1 is enabled by raising the Start signal. The Done signal goes high after the completion of the computation. A Request is issued to F2. If it is free, an Acknowledgement is raised and the output value is sent to R2. After which, the process can repeat itself.

2.6 Floating Point Adder

A little known fact is that floating point arithmetic is an essential component in computer systems for several reasons. Almost every computer language has floating point data type and accelerators. Compilers and operating systems are capable of processing information in the floating point format. Even more importantly, is how essential the floating point unit is to high performance computing (HPC), mobile applications and embedded systems. Computer system's performance is measured in Floating Point Operations per Second; more commonly known as FLOPs.

The overall performance of HPC system or any other computing system is greatly affected by the Floating Point Unit design; thus, the architecture can affect overall performance and power dissipation [28, 29, 30]. Within the floating point unit are several components: Adder/Subtractor, Multiplier and Divider. As their names suggest, they each have a specific computational roles. However, the Adder is the single most commonly used component in this unit. According to data Pappalardo et al in [13], signal processing algorithms require on average, 40% multiplication and 60% addition operations; once again, reinforcing the importance of FPA.

Due to the emerging field of computational science and the widespread use of high performance computers dealing with application such as computational fluid dynamics, the floating point unit is now consuming more power than ever. In fact, a major portion of the systems power is used to maintain these floating point units. Therefore, a reduction of power usage for these unit will decrease the overall power dissipation of the system. For these reasons, I have chosen to focus my research on reducing the amount of power used in the Floating Point Adder. More specifically, a Ripple Carry Adder.

2.7 Related Works

There has been a plethora of works and research geared towards improvement of the FPA. Many of which investigate techniques that optimize the latency factors that are in large part dependant on the circuit topology. The key component of this unit is the adder type. The Ripple Carry Adder was chosen because of its simplicity in design. There are design performance/power/area tradeoffs which must be addressed. In most cases, system designers use architectural optimizations to develop a more efficient RCA design [14, 15, 16]. However, since synchronous system designers are limited to constraints associated with trading energy for performance in CMOS circuits, their designs are application based and are not robust. The designs focus on reducing the latency of the circuit, while paying a significant area penalty.

There have been two papers that have chosen to exploit the performance and power reduction aspects of the dynamic circuit and use asynchronous logic. N-PMOS logic and DRCA was implemented in [17 and 18]. While the DRCA in [18] was proven to be a superior logic style for the application, it has not resolved the race conditions that were created and under certain conditions, could produce erroneous data.

CHAPTER 3

ASYNCHRONOUS SYSTEM REALIZATION

3.1 Self-timed

Traditional synchronous optimization approaches have accepted the notion that there must be some tradeoff between power and performance. Asynchronous systems offer us something more in terms of speed and power dissipation which allows designers to exploit these properties to produce a superior power delay product. In order to realize these systems, we should examine which particular type of asynchrony we would like to implement. There are several types of asynchronous styles. Burst-mode design begins with a state-machine specification, somewhat like conventional synchronous state-machine synthesis methods. However, the transitions in the machine are governed by the inputs themselves, not by a clock. Self-timed design's structure and behavior are very similar to synchronous thus they are much easier to implement.

3.2 Handshake Protocol

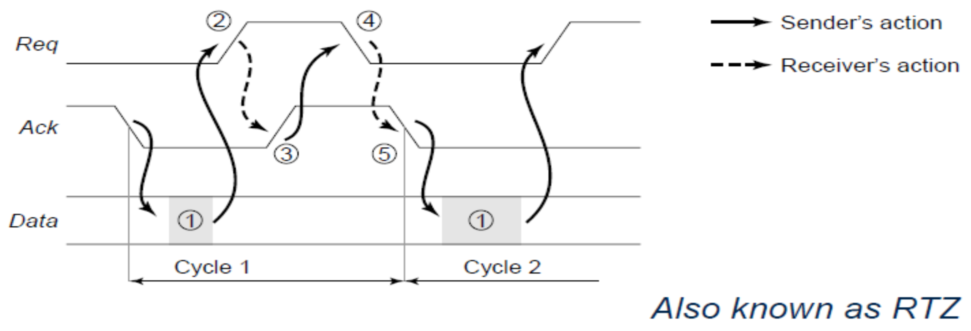


Figure 3.1: Four Phase Handshake Protocol [4]

In asynchronous or self-timed systems, handshake signals, more commonly known as Request, which initiate an action and Acknowledge which signals completion of that action are used to regulate the flow of information in the system. [19] Shows the fundamental building block of the handshake family. The four phase handshake protocol or return-to-zero is illustrated in figure 3.1. This type of signaling approach requires that all control signals be brought back to their original values before the next cycle can begin. Both the Req and Ack are initially low. When new input is placed on bus (1), the Req is raised high (2) and control is given to the receiver. The receiver then raises Ack high (3). After which Req is returned to low (4) and Ack is returned as well (5).

3.3 Dynamic Logic

Now we can move from the system level to the gate level. Self-timed circuits are sensitive to glitches, an undesired transition that occurs before the signal settles to its intended value. Therefore self-timed systems must be realized with a glitch free logic style that does not produce any static or dynamic hazards. A dynamic gate alleviates these hazards because during the evaluation phase, there is at most one transition. Figure 3.2 illustrates a dynamic logic gate. It is also great for fast and complex gates. Dynamic gates are composed of a n-type logic gate, Pull Down Network (PDN) and transistors that regulate the mode of operation: Pre- charge and evaluate. During the pre-charge phase, the clock = 0, and the output node is charged to *Vdd* by the PMOS transistor. At that time, the NMOS is off and therefore the PDN is disabled which also eliminates the static current. When the clock = 1, the evaluate phase, the PMOS is turned off and the NMOS is turned on. If the inputs are such that the NMOS conducts, then a path between out and ground exist and the output is discharged to ground. Since the PMOS is turned off, the pre-charge value remains stored on the output capacitance. During this phased, the only path that exist between output and *Vdd* is ground.

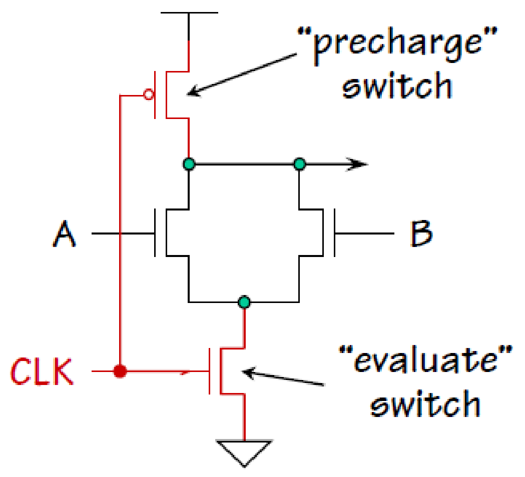


Figure 3.2: Dynamic Logic Gate

Therefore, once out is discharged, it can only be charged again during the next pre-charge phase. Inputs can only make at most, one transition during the evaluation phase.

There are several advantages to the logic style. Fewer transistors are used; 2+N as opposed to 2N for standard CMOS gates which allows for a smaller implementation area. There is also no static current between *Vdd* and ground since one part of the circuit is always turned off. Dynamic gates also have faster switching speeds. This is due in part to the reduced load capacitance because they are driving fewer transistors; one as opposed to two. Also, since all of the input capacitance is dedicated to the falling transitions, and not the slow PMOS transistors, they require a reduced logic effort. For example: a two input NOR gate only requires to be sized 2/3 vs. 5/3 for the static logic.

3.4 Domino Logic

For the circuit that I am using in this dissertation, I used Domino Logic which is a shown in figure 3.3. The structure is a N-type dynamic gate followed by a static inverter. During the pre-charge phase, the output of the N-type dynamic gate id charged up to *Vdd* and the output of the inverter is set to zero. In the evaluation phase, the dynamic gate is conditionally discharged and the output of the inverter makes a conditional transition from 0 -> 1. All of the inputs of domino gates are outputs of other domino gates which ensure that all inputs are set to zero at the end of the pre-charge phase. Therefore, during the evaluation phase, only 0 -> 1 transitions are made. The inverter is used as a buffer which (1) increases noise immunity, (2) reduces the capacitance of the output node by separating the internal and load capacitance, and (3) is used as a keeper to reduce the leakage and charge redistribution.

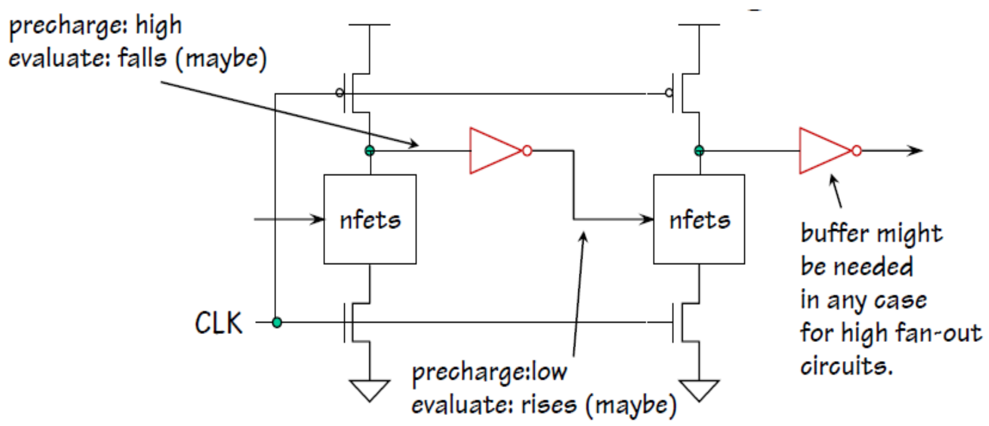


Figure 3.3: Domino Logic

On the norm, domino logic is 1.5 to 2 times faster than static CMOS logic because dynamic gates present much lower input capacitance for the same output current and have lower switching thresholds [12]. In static gates, much of the input capacitance is wasted on the slower PMOS that are not even used during a falling transition. Other reason dynamic gates are a good choice is because they have lower switching threshold. The dynamic gate will begin to switch as soon as the inputs rise to *Vt,* as opposed to *Vdd/2* for the static. [1]

Only non-inverting logic structures are possible because of the presence of inverting static buffer.   For this reason, many designer stay away from this complex logic style. However, if you are an experienced designer and performance is important, dual rail logic may be implemented to produce the signal and its’ complement with an area penalty.

CHAPTER 4

RESEARCH

4.1 Logic Gate Delay

Now that we understand how self-timed circuits are realized, let's review how we model the timing process. The delay in a logic gate is determined by the topology of the gate (fan in) and the capacitive load that the logic gate drives (fan out). Logical effort is a term coined by Ivan Sutherland and Bob Sproull in 1991 which is a method that is used to model the delay of a single logic gate. Logical effort method provides a technique to determine the most efficient transistor sizing on the critical path to minimize the delay, as well as, providing an estimation of that delay. The delay of a logic gate using logical effort is given as:

$$d = f + p \qquad (4)$$

where *p* is the parasitic delay which is the intrinsic delay of the gate driving no load, and *f* is the stage effort. The stage effort is defined as:

$$f = gh \qquad (5)$$

(6)

where g is logical effort which is the ratio of the input capacitance of a given gate to that of an inverter capable of delivering the same output current and h is effective fan out cout/cin. The dependency is demonstrated in figure 4.1.

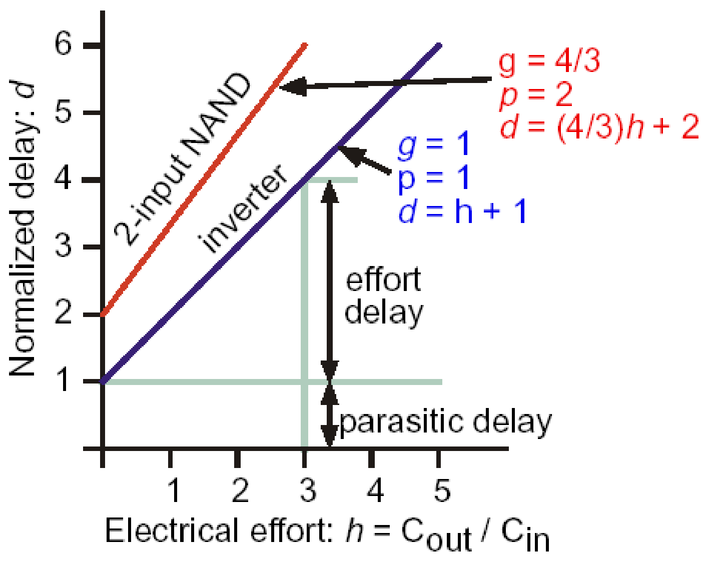


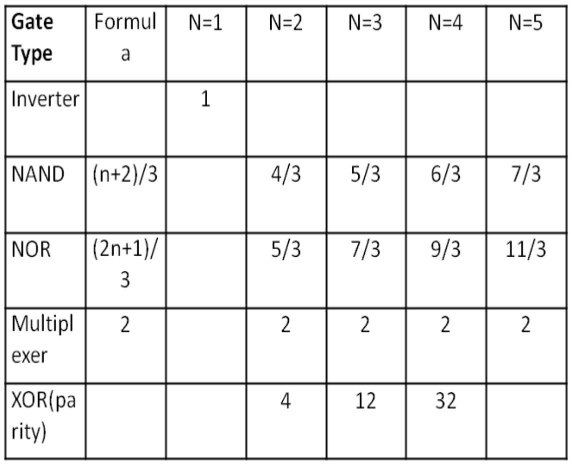
Figure 4.1: Delay expressed in terms of a minimal sized inverter [1]

The delay is a function of electrical effort of and inverter for a two input NAND gate. The slope of each line is the logical effort and the y-intercept is the parasitic delay. As shown, we can adjust the total delay by adjusting the electrical effort or by choosing a logic gate with a different logical effort [1].

4.2 Logical Effort

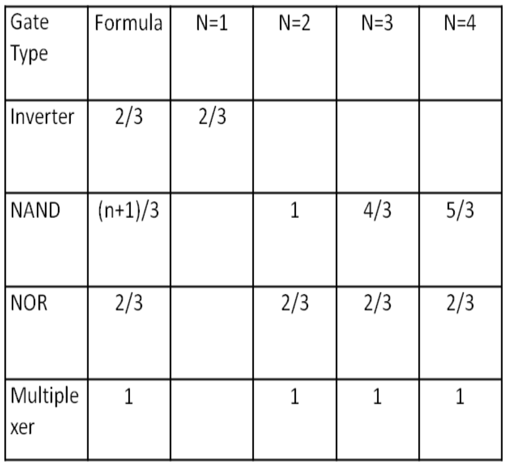
The tables 4.1a and 4.1b below are a representation of the logical effort for static gates and dynamic gates. Clearly, the dynamic logic style allows for smaller sizing which partially explains why dynamic gates are faster than static gates. In static gates, much of the input capacitance is wasted on slow PMOS transistors that are not even used during a falling transition. [4] From table 4.1 we see that the dynamic inverter has a logical effort of 1/3 less than the static inverter. Since logical effort is used for sizing estimations of each component, I have included the table below where N=number of inputs.

Table 4.1: Logical effort per input of (a) and (b)



Gate Type	Formula	N=1	N=2	N=3	N=4	N=5
Inverter		1				
NAND	(n+2)/3		4/3	5/3	6/3	7/3
NOR	(2n+1)/3		5/3	7/3	9/3	11/3
Multiplexer	2		2	2	2	2
XOR(parity)			4	12	32	

(a) static CMOS gate

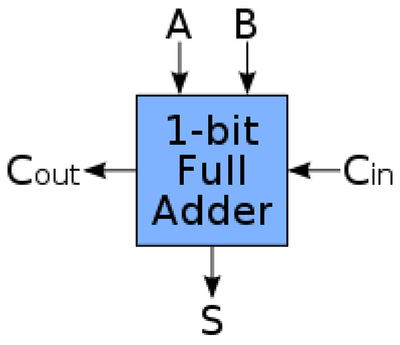


Gate Type	Formula	N=1	N=2	N=3	N=4
Inverter	2/3	2/3			
NAND	(n+1)/3		1	4/3	5/3
NOR	2/3		2/3	2/3	2/3
Multiplexer	1		1	1	1

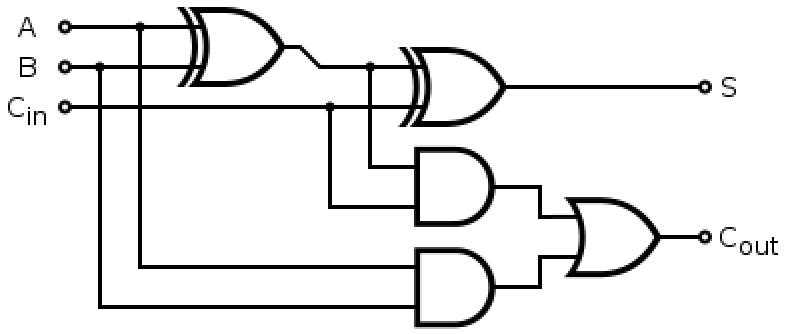
(b) dynamic CMOS gates

4.3 Self-timed Ripple Carry Adder Circuit Design

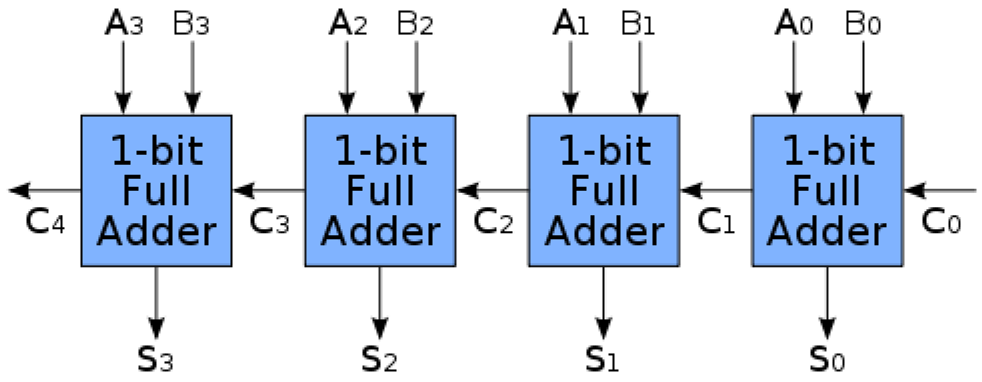
In digital electronics, an adder is a digital circuit that performs addition and in normally located in the arithmetic logic unit. There are many different types of adders (Ripple Carry Adder, Carry Look ahead, Carry Select, Conditional Sum, ect.) which designers carefully choose according to the design application. For the purpose of this dissertation, we will examine the RCA.



(a)



(b)



(c)

Figure 4.2: (a) Full Adder Schematic, (b) Logic diagram, and (c) 4 bit RCA

The foundation of a RCA is a full adder since it is possible to create a logical circuit using more than one full adder to add N-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous full adder. This type of adder is a ripple carry adder, since each C_{out} bit "ripples" to the next full adder.

A full adder adds three one-bit binary numbers, often written as *A*, *B*, and C_{in}; *A* and *B* are the operands, and C_{in} is a bit carried in (in theory from a past addition). The circuit produces a two-bit output sum typically represented by the signals C_{out} and *S*. The equations to implement the logic for figure 4.2 is:

$$S = A \oplus B \oplus C_{in} \tag{7}$$

$$C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B)) \tag{8}$$

which is represented in the truth table 4.2 below.

Since, in the worst case, the carry can propagate from the least significant bit position to the most significant bit position, the addition time of an N-bit RCA is *O(X)*. The RCA is typically slower than other adders; however the ease of design makes it attractive to many.

The dynamic power dissipation depends primarily on the number of transitions per unit area. As a result, the average number of logic transitions can serve as the basis for comparing the efficiency of a variety of adder designs [10].

Table 4.2: Truth table for full adder

Inputs			Outputs	
A	B			S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 4.3: Average Number of Logic Transitions per Addition [10]

ADDER TYPE	ADDER SIZE (BITS)		
	16	32	64
Ripple Carry	90	182	366
Carry Lookahead	100	202	405
Carry Skip	108	220	437
Carry Select	161	344	711
Conditional Sum	218	543	1323

From table 4.3, we can see that the Ripple Carry Adder uses the least amount of logic transitions per addition. Even though, the propagation delay is higher than the Carry Lookahead Adder, another reason it uses less power is it has a lower transistor count. For example, a four bit Ripple Carry Adder uses 120 transistors as opposed to 170 used by its counterpart, the Carry Lookahead Adder. The transistor count directly affects the capacitance stored by the circuit. Minimizing the transistor count reduces the physical capacitance or stored electrical charge of a circuit. This in turn, reduces power dissipation. The increased propagational delay is offset by using domino logic.

Figure 4.3 illustrates the one bit ripple carry adder that was designed to boost performance and decrease power dissipation. The data path is designed twice for the signal and its' complement. Domino logic is used, the non-inverting logic followed by a static inverter, to implement the logic. The done signal is used as a completion detection signal which is used in the four phase handshake protocol. The geometry of the transistors within each of these individual gates was defined using the approach defined for "logical effort". All components in this dissertation are implemented in an 180nm TMSC process where lambda is 90nm and the devise dominions are based on the design rules of this process. The figures below show the gate and transistor level realization of all the components used to create the one bit RCA. The intrinsic time constant and parasitic delays of the CMOS components are determined with a SPICE3 simulation for the TSMC 180nm process with the specifications *Vdd* of 1.8 Volts, at a temperature of 27C.

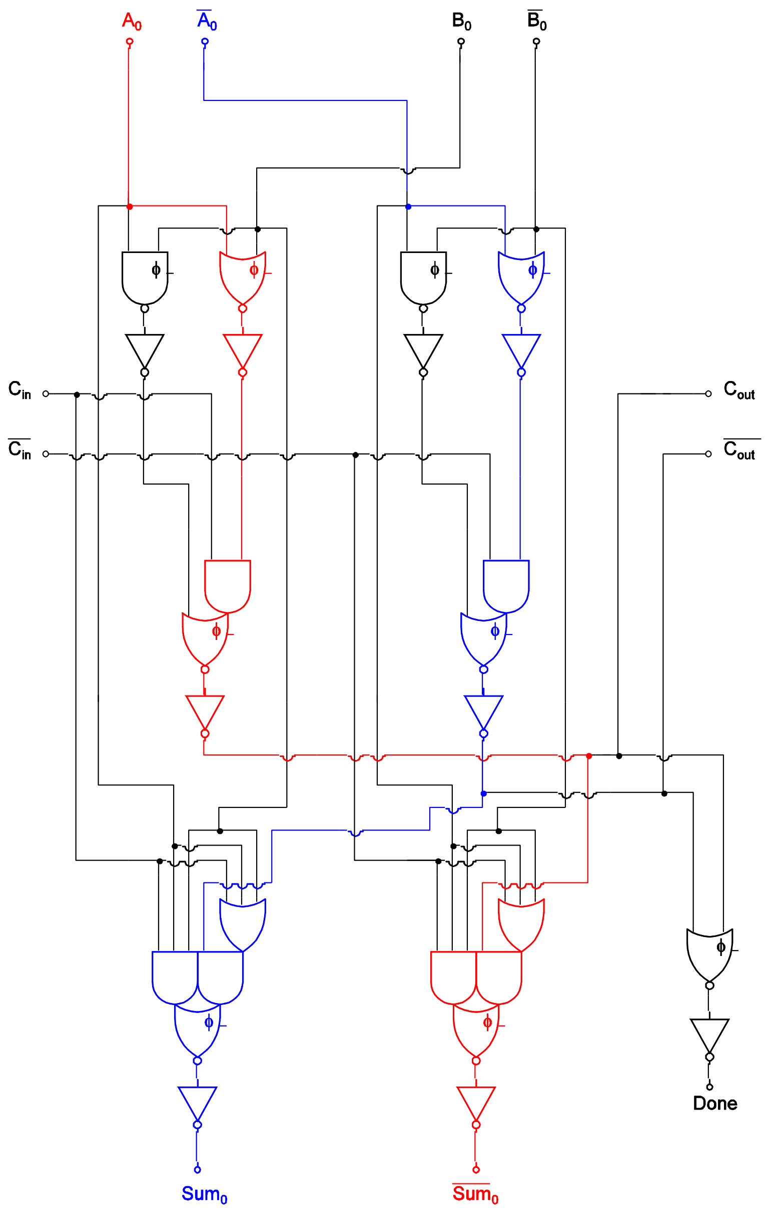


Figure 4.3: Domino Logic Realization of One bit Adder

4.4 1- Bit RCA Sub-circuit Parameters

The static high skew CMOS inverters are used in this design. The circuit level realization and associated device geometry is shown figure 4.4.

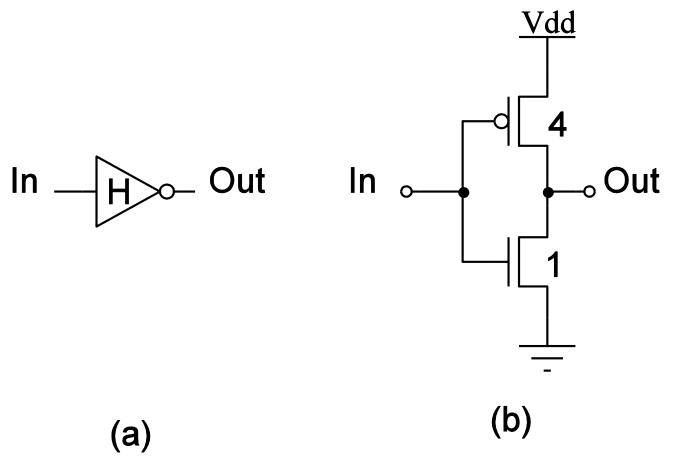


Figure 4.4: High-Skew Inverter gate (a) and transistor (b) level schematic

The simulated voltage transfer curve for the high skew and standard inverter is shown in figure 4.5 below.

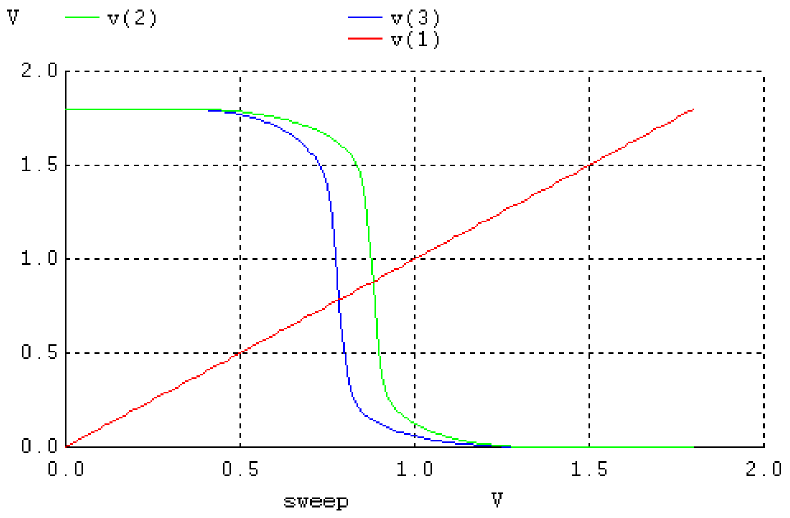


Figure 4.5: VTC of high-skew inverter [i.e. V(2)] and standard inverter [i.e. V(3)]

The SPICE simulation of the high-skew inverter calibration for the input and output voltage is illustrated in figure 4.6 below.

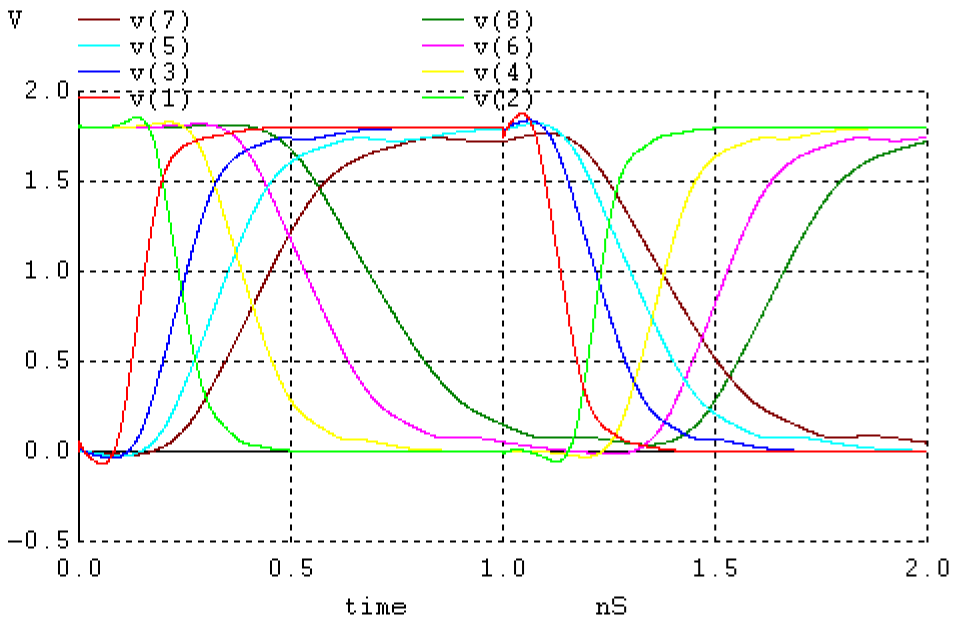


Figure 4.6: High-skew Inverter Calibration Input and Output Voltage

The logical effort and parasitic delay associated with this logic function is shown in table 4.4.

Table 4.4: High-skew inverter Logical Effort

High-skew Inverter		
Output transition	Logical Effort	Parasitic Effort
High-to-Low	1.403510	1.639791
Low-to-high	1.455118	1.303958

The AND Gate used to implement the 1-bit RCA is shown below. The gate schematic is shown in figure 4.7 with the compound gate symbol used in the adder schematic. The device sizing used to realize the gate is also shown in this figure. The sizing is based on the 1x scaling in the adder schematic and 3-12 sizing used for the high skew inverter. Dynamic AND gates are used in this design. The circuit level realization and associated device geometry is shown figure 4.7.

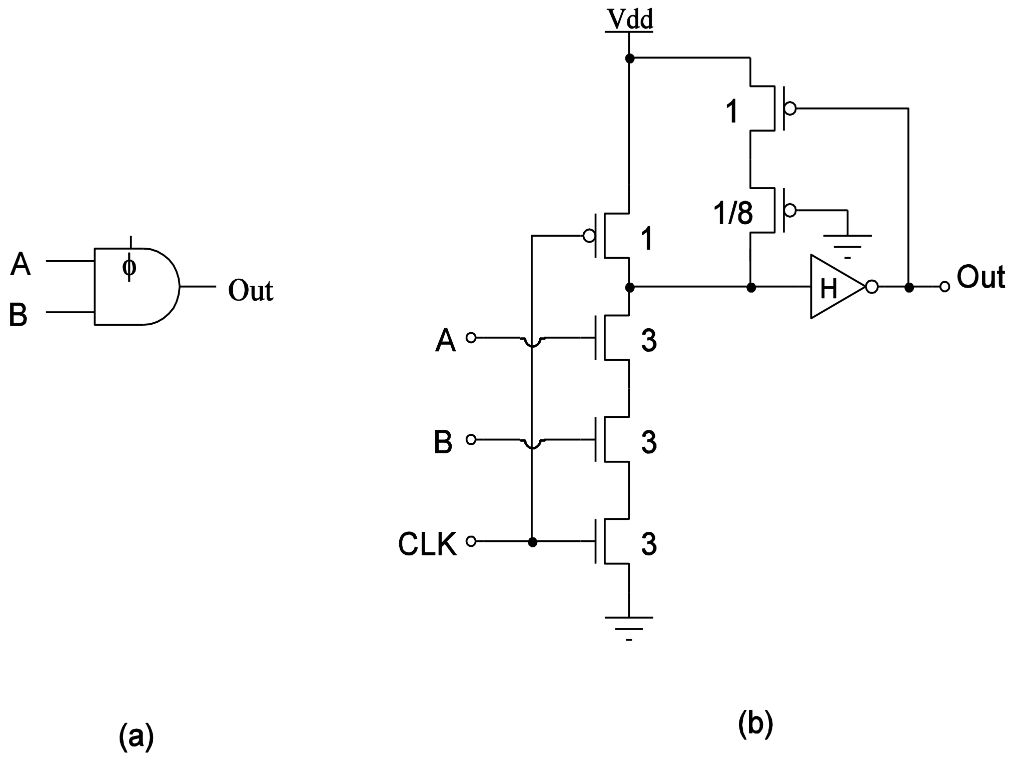


Figure 4.7: Dynamic AND gate (a) and transistor (b) level schematic

Figure 4.8 shows the SPICE simulation that was used to compute the logical and parasitic effort of the AND gate. The simulation was done without the output inverter in the signal path.

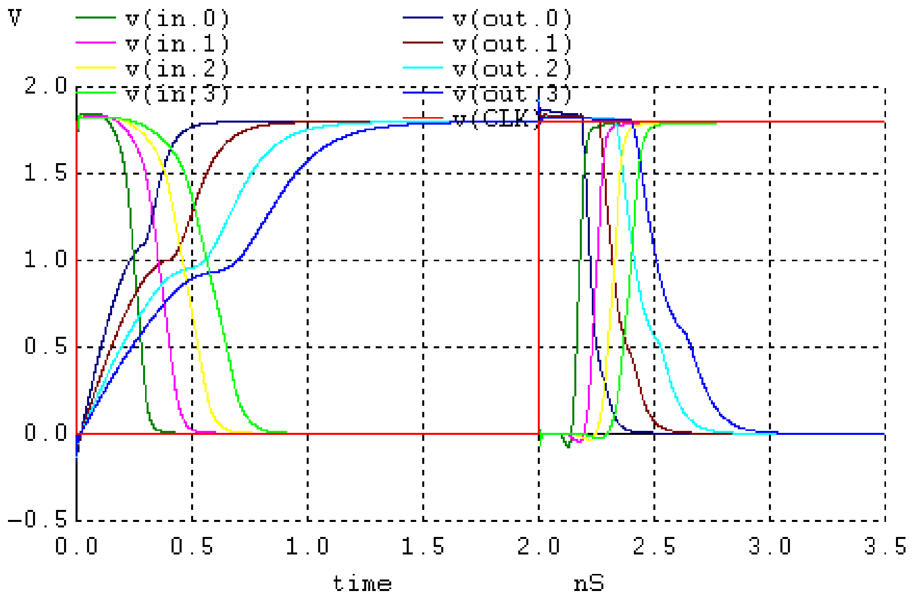


Figure 4.8: Dynamic AND gate embedded NAND gate input and output voltage

The normalized logical and parasitic effort of this gate is shown in table 4.5.

Table 4.5: Embedded 2-input NAND gate Logical Effort

2-input dynamic NAND gate with keeper		
Input	Logical Effort	Parasitic Effort
A	0.800326	1.02319
B	0.753106	1.40450

All of these entries are normalized with respect to the average propagation delay of a minimum sized inverter (i.e. 17.52 picoseconds). Table 4.6 below illustrates NAND gate input capacitance.

Table 4.6: Embedded 2-input NAND gate input capacitance

2-input dynamic NAND gate input capacitance	
Input	Capacitance
A	1.62fF
B	1.62fF

The parasitic effort of the dynamic AND is:

$$P_{AND} = P_{low-to-High} + P_{NAND2} + g_{NAND2} \cdot \left(\frac{5}{3}\right) + g_{inv} \cdot \left(\frac{1}{5}\right) = 1.665108 + 1.02319 + 1.333876 + 0.3075876$$

Table 4.7 shows the logical effort and parasitic effort of the 2-input AND gates.

Table 4.7: 2-input AND gate Logical Effort.

2-input dynamic AND gate with keeper		
Input	Logical Effort	Parasitic Effort
A	1.455118	3.952149
B	1.455118	4.254657

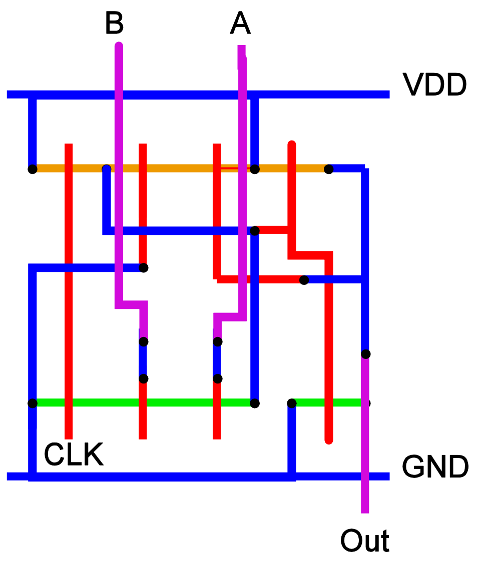


Figure 4.9: Dynamic AND stick diagram

Figure 4.9 above represent the stick diagram of a dynamic AND gate. Stick diagrams are used by designers to determine how to layout a VLSI realization. It is used as a tool to create a preliminary guess on how to lay the circuit out for fabrication without the worry of device parameters rules. Once the stick diagram has been created, it is used as the blue print for the layout which is illustrated in figure 4.10.

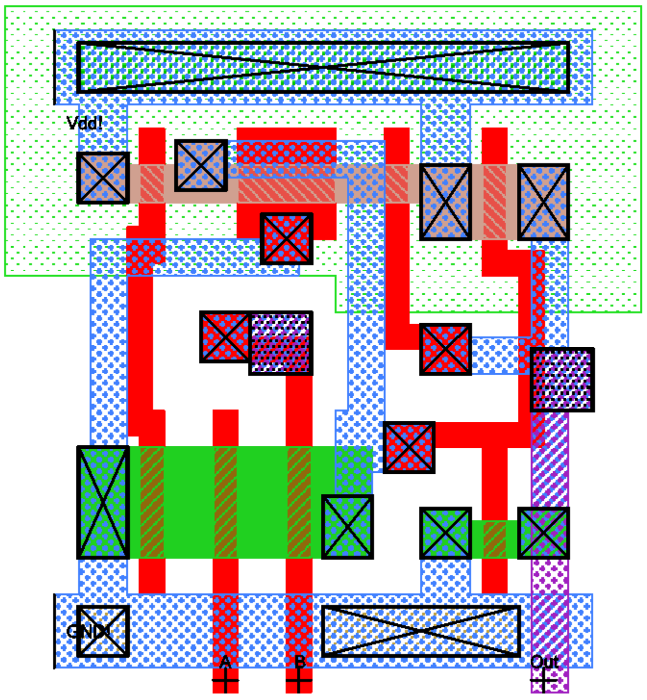


Figure 4.10: Dynamic 2-input AND gate Layout

The OR Gate used to implement the 1-bit RCA is shown below. The gate schematic is shown in figure 4.11 with the compound gate symbol used in the adder schematic. The device sizing used to realize the gate is also shown in this figure. The sizing is based on the 1x scaling in the adder schematic and 3-12 sizing used for the high-skew inverter.

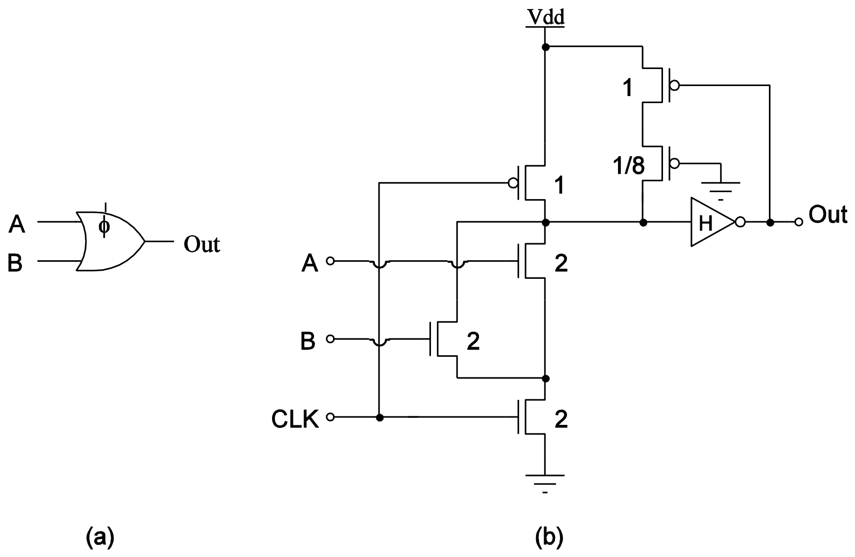


Figure 4.11: Dynamic 2-input OR gate (a) and transistor (b) level schematic

Figure 4.12 shows the SPICE simulation that was used to compute the logical and parasitic effort of the OR gate. The simulation was done without the output inverter in the signal path.

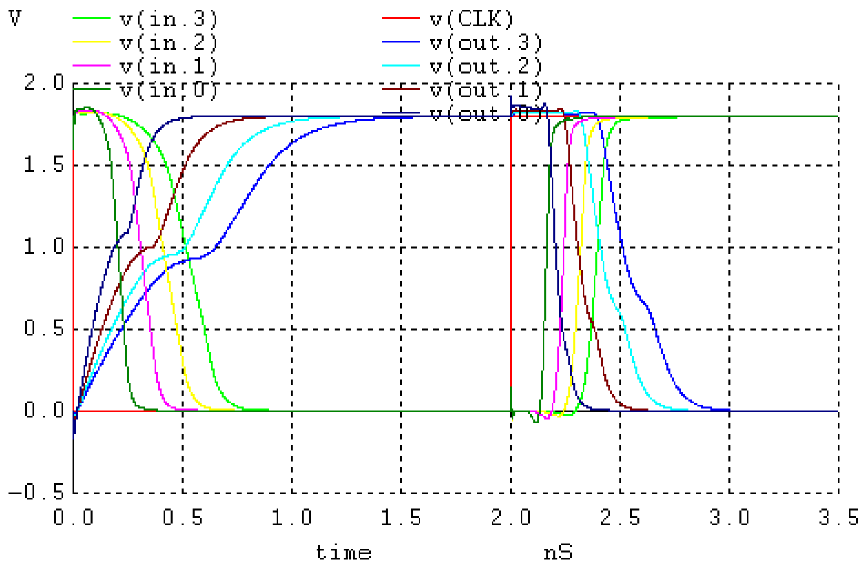


Figure 4.12: Dynamic 2-input OR gate input and output voltage

The normalized logical and parasitic effort of this gate is shown in table 4.8.

Table 4.8: Embedded 2-input NOR gate Logical Effort

2-input dynamic NOR gate with keeper		
Input	Logical Effort	Parasitic Effort
A	0.851715	0.988586
B	0.851715	0.988586

All of these entries are normalized with respect to the average propagation delay of a minimum sized inverter (i.e. 17.52 picoseconds). Table 4.9 below illustrates NOR gate input capacitance.

Table 4.9: Embedded 2-input NOR gate input capacitance

2-input dynamic NOR gate input capacitance	
Input	Capacitance
A	1.08fF
B	1.08fF

Parasitic Effort of Dynamic NOR is:

$$P_{NOR} = P_{low-to-High} + P_{NOR2} + g_{NOR2} \cdot \left(\frac{5}{2}\right) + g_{inv} \cdot \left(\frac{1}{5}\right) = 1.665108 + 0.988586 + 2.129287 + 0.3075876$$

The calculated logical effort for the 2-input OR gate is shown in table 4.10 below.

Table 4.10: 2-input OR gate Logical Effort

2-input dynamic OR gate with keeper		
Input	Logical Effort	Parasitic Effort
A	1.455118	4.712854
B	1.455118	4.712854

Again, stick diagrams are used in figure 4.13 as a preliminary model for the layout of the circuit in figure 4.14.

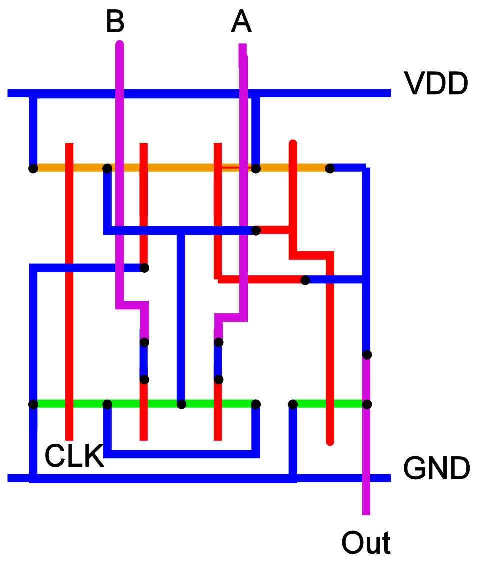


Figure 4.13: Dynamic 2-input OR stick diagram

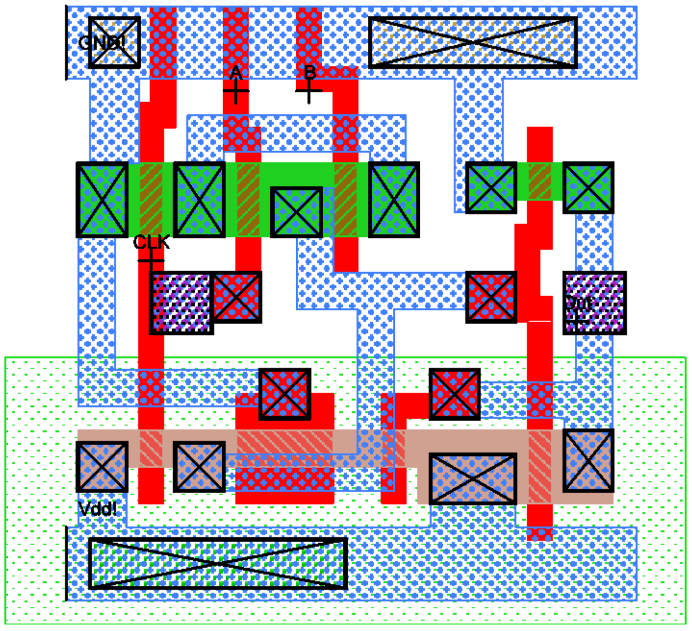


Figure 4.14: Dynamic 2-input OR gate layout

The AOI21 Gate used to implement the 1-bit RCA is shown below. The gate schematic is shown in figure 28 with the compound gate symbol used in the adder schematic. The device sizing used to realize the gate is also shown in this figure. The sizing is based on the 1x scaling in the adder schematic and 3-12 sizing used for the high-skew inverter.

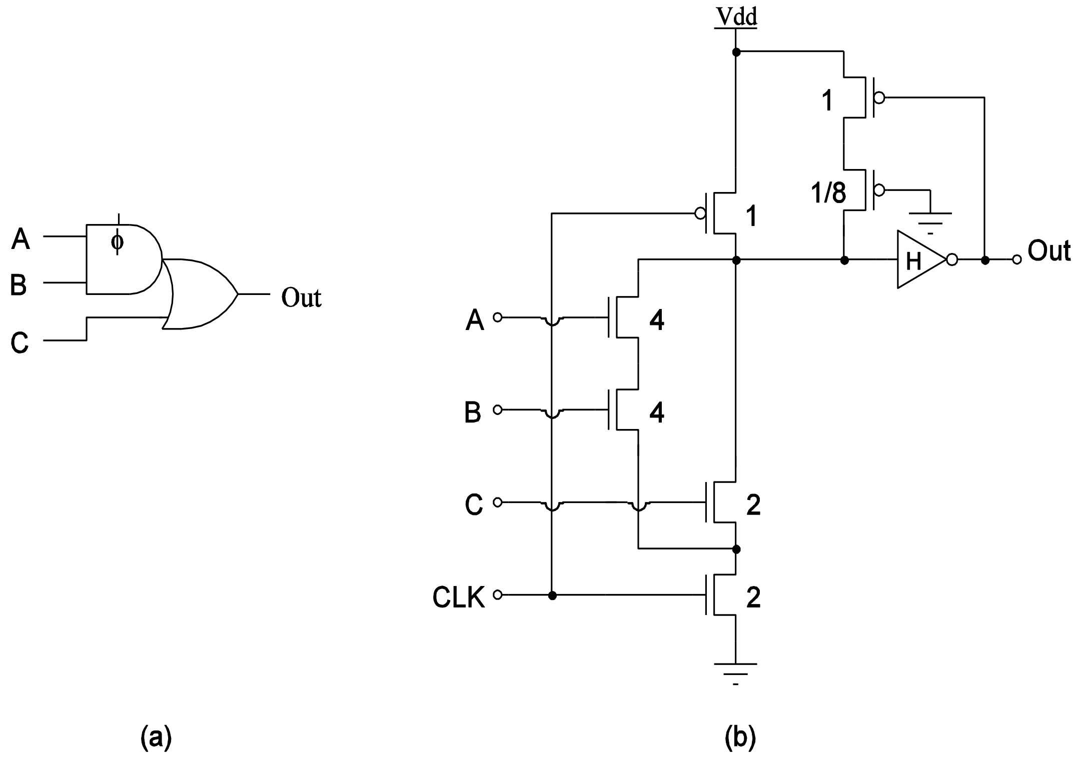


Figure 4.15: Dynamic AO21 gate (a) and transistor (b) schematics

Figure 4.16 shows the SPICE simulation that was used to compute the logical and parasitic effort of the AOI21 gate. The simulation was done without the output inverter in the signal path. The Boolean function implemented by this compound gate is *F(A,B,C) = A* B+ C.*

The logical effort of this gate with reference to sized devices is shown in table 4.11. These values were computed with a calibration circuit and SPICE simulation. The input capacitance of this gate is shown in table 4.12.

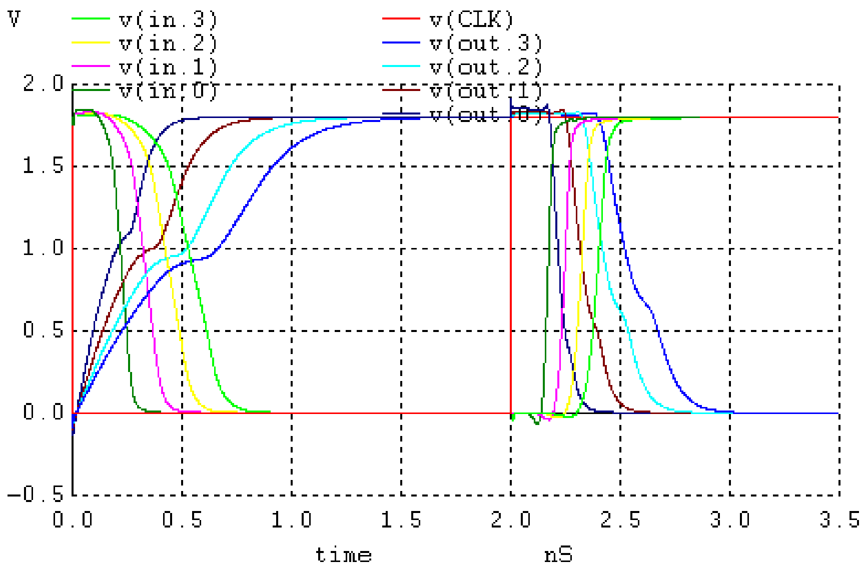


Figure 4.16: Dynamic AO21 gate input and output voltage

Table 4.11: Embedded AOI21 gate logical effort

Dynamic OAI21 gate with keeper		
Input	Logical Effort	Parasitic Effort
A	0.764290	1.23525
B	0.725933	1.82991
C	0.851006	1.12045

Table 4.12: Embedded AO21 gate input capacitance

2-input Dynamic AOI21 gate input capacitance	
Input	Capacitance
A	2.16fF
B	2.16fF
C	1.08fF

Parasitic Effort of Dynamic AOI21 is:

$$P_{AO21_c} = P_{low-to-High} + P_{AO21_c} + g_{AO21_c} \cdot \left(\frac{5}{2}\right) + g_{inv} \cdot \left(\frac{1}{5}\right) = 1.665108 + 1.12045 + 2.127515 + 0.3075876$$

Once again, a stick diagram is used in figure 4.17 as a preliminary blue print for the layout of the circuit in figure 4.18.

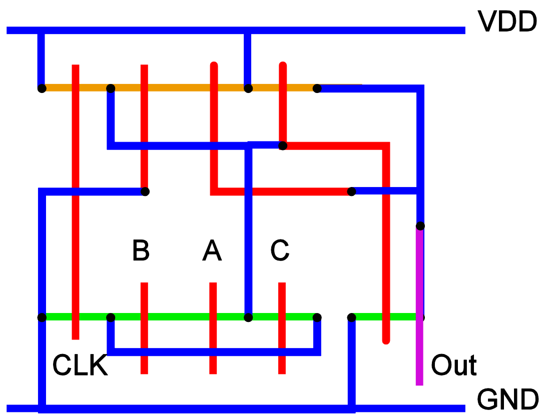


Figure 4.17: Dynamic AO21 gate stick diagram

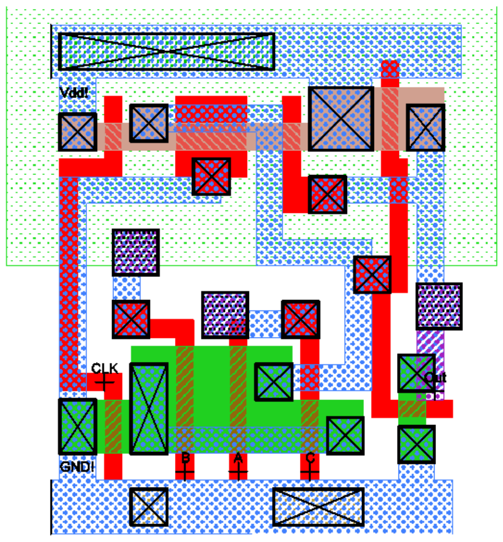


Figure 4.18: Dynamic AO21 gate layout

The Sum Gate used to implement the 1-bit RCA is shown below. The gate schematic is shown in Figure 4.19 with the compound gate symbol used in the adder schematic. The device sizing used to realize the gate is also shown in this figure. The sizing is based on the 1x scaling in the adder schematic and 3-12 sizing used for the high-skew inverter.

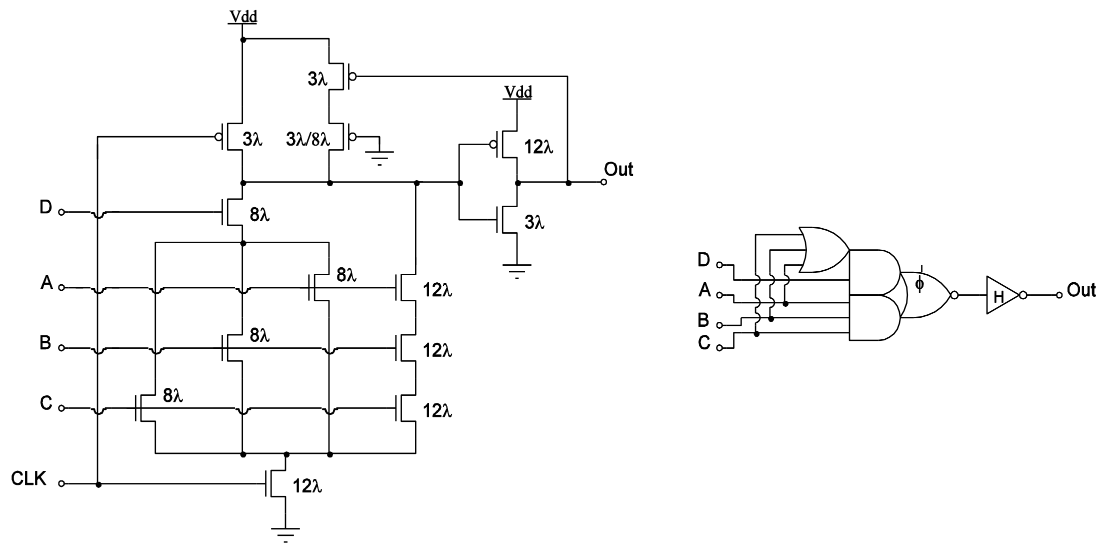


Figure 4.19: Dynamic Sum gate and transistor schematics

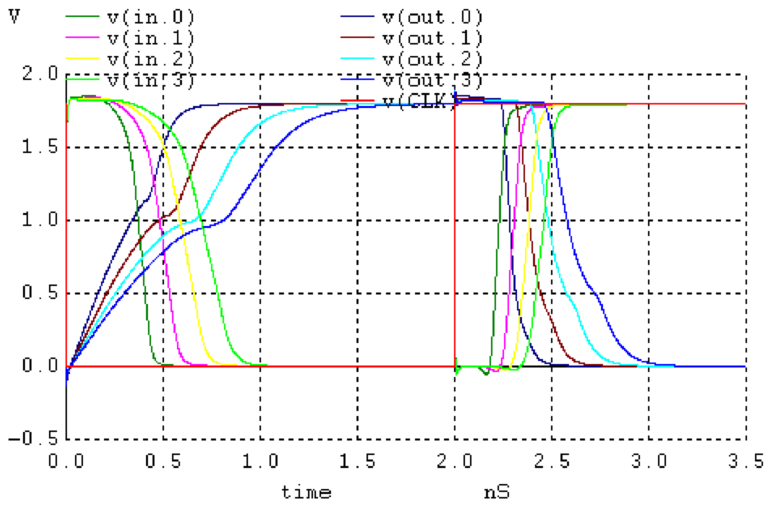


Figure 4.20: Dynamic not Sum gate calibration circuit input and output voltage

Figure 4.20 above shows the SPICE simulations used to compute the logical and parasitic effort of the sum gate. The simulation was done without the output inverter in the signal path. Tables 4.13, 4.14 and 4.15 show the logical effort of the not sum gate (complement of the signal), input capacitance and logical effort of the sum gate, respectively.

Table 4.13: Embedded not Sum gate logical effort

Dynamic Majority gate with Keeper			
Input	Logical Effort	Parasitic Effort	ABCD
A	0.762905	1.47823	-110
B	0.730713	1.98016	1-10
C	0.714673	2.26256	11-0
D	0.833278	1.27067	100-
A*	0.778093	2.19541	-001
B*	0.766942	1.81375	0-01
C*	0.756949	2.12740	00-1

Table 4.14: Embedded Sum gate input capacitance

Dynamic Sum gate input capacitance	
Input	Capacitance
A	3.6fF
B	3.6fF
C	3.6fF
D	1.44fF

Table 4.15: Sum gate logical effort

Dynamic Majority gate with Keeper			
Input	Logical Effort	Parasitic Effort	ABCD
A	0.762905	1.47823	-110
B	0.730713	1.98016	1-10
C	0.714673	2.26256	11-0
D	0.833278	1.27067	100-
A*	0.778093	2.19541	-001
B*	0.766942	1.81375	0-01
C*	0.756949	2.122740	00-1

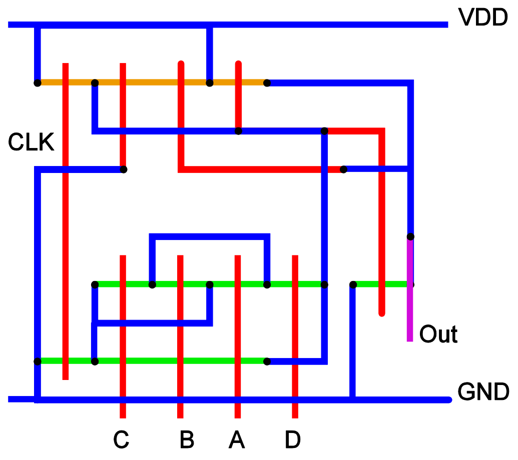


Figure 4.21: Dynamic Sum gate stick diagram

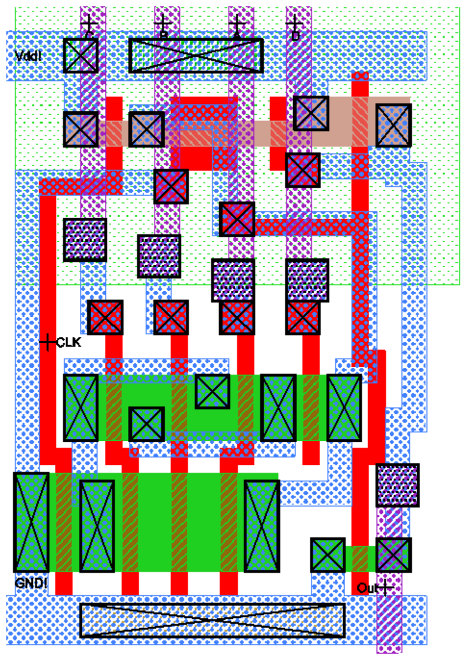


Figure 4.22: Dynamic Sum gate Layout

Again, stick diagrams are used in figure 4.21 as a preliminary model for the layout of the circuit in figure 4.22, while figure 4.23 is the layout of the whole circuit.

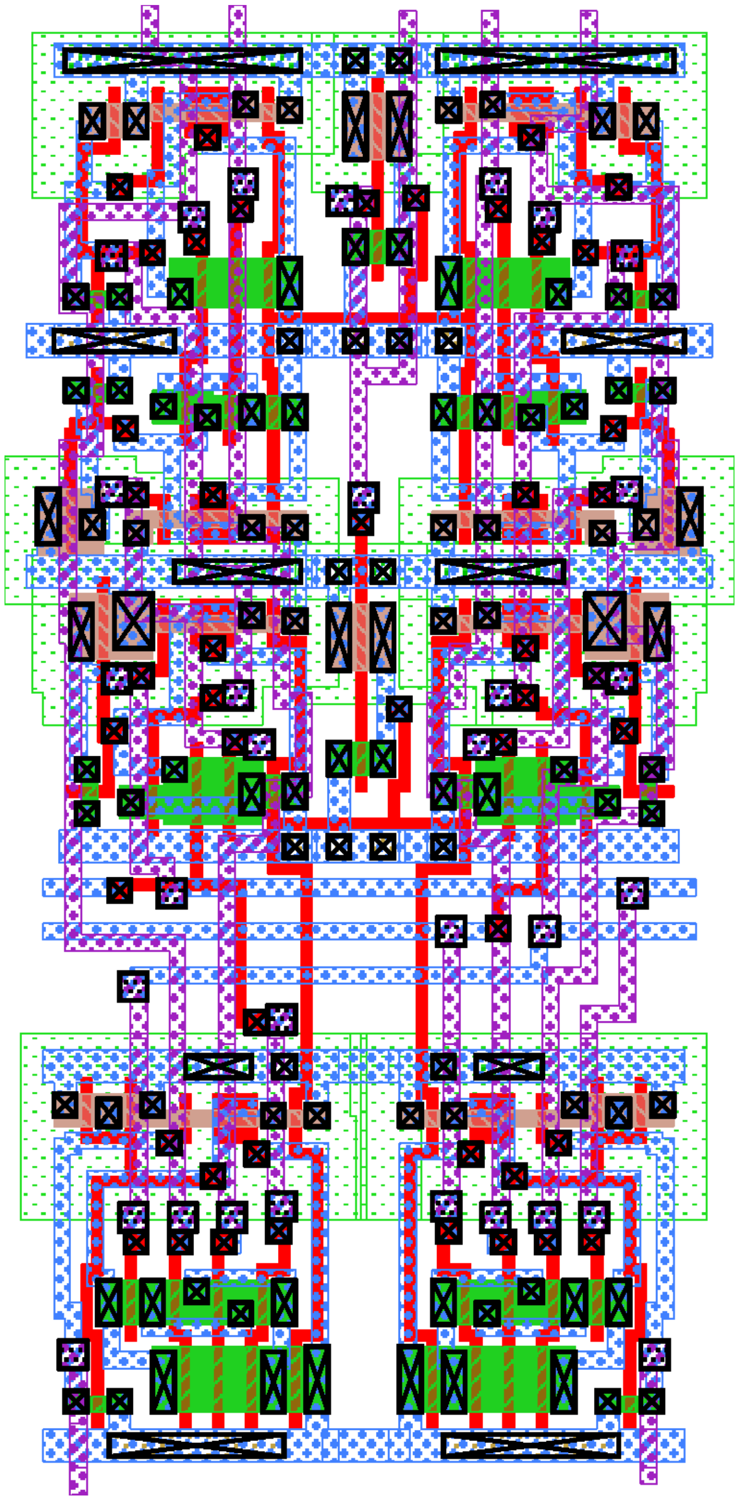


Figure 4.23 Layout of One Bit Adder

4.5 Input Distribution in Self-timed Circuits

To achieve high performance and manage power loss, designers should consider non-traditional levels of abstraction, in particularly, input data profiling. Since the switching activity of a logic gate is a strong function of the input signal statistics, system designers can use this knowledge to exploit power delay capabilities of a circuit.   In this dissertation, a pipelined architecture that intersects the timing function of the circuit itself and the data that it is processing is utilized. Using input data distribution to increase self-timed circuit performance and decrease energy dissipation is novel because the timing is determined locally, which is a function of the circuit and the input data.

A few advantages of this proposed technique is the decreased circuit area. This is realized when the probability of a path being used is very low then the transistors on the path will be sized smaller. There is also an increase average circuit performance because when you include data profiling, performance is even better than self-timed alone. The average energy dissipation is decreased since energy is only consumed when and event happens. The decrease circuit noise is due in part by the fact that fewer transistors are used which decreases circuit activity. The local clock distribution alleviates the greedy global clock network and hazards that can be introduced by clock skew. This technique is less sensitive to changes to process variation because timing is generated locally. Figure 4.25 gives a graphical illustration of a one bit self-timed RCA circuit path activation probability with eight different input distributions (0-7) and four different activation or critical paths illustrated by the different colors along the path.

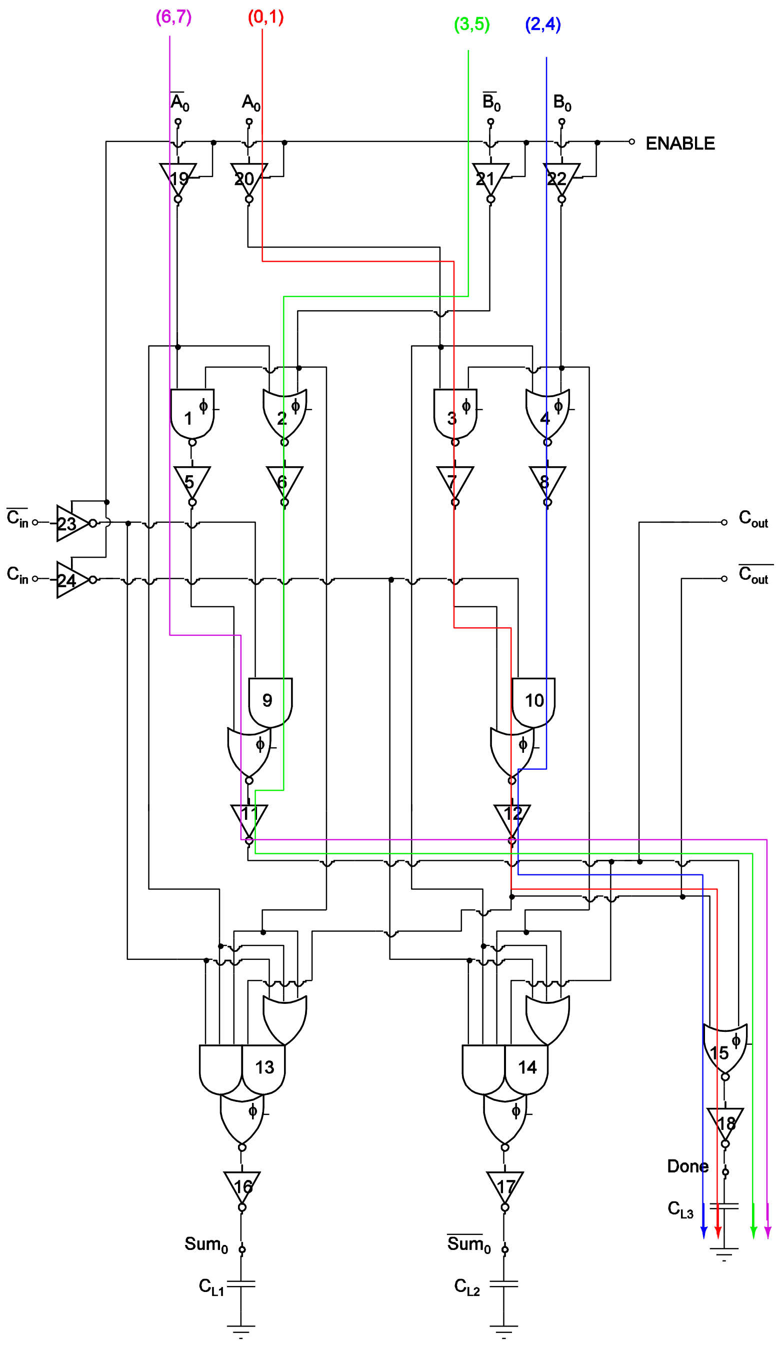


Figure 4.24: Circuit Path Activation Probability

There are a few disadvantages. There are very few Computer Aided Design development tools for design a verification. Sensitive to charge sharing is another concern that is just the nature of dynamic logic which can be offset by circuit design that is sized to minimize the effect.

The performance and energy dissipation of synchronous and asynchronous digital system is determined in part by the geometry of the devices used to realize the system embedded gates. The device geometry is set in the design process to minimize the propagation delay along all the paths in the systems. This approach maximizes the performance of synchronous systems because the propagation delay of the circuit critical path is also minimized. However the performance of asynchronous circuits is not maximized because the average propagation delay is not minimized. The performance and energy dissipation of asynchronous circuits that are optimized for the average delay of the completion detection circuit are maximized and minimized respectively. The proposed technique achieves this because it is based on the average completion circuit propagation delay and the circuit input data distribution.

A novel self-timed circuit device sizing approach is presented in the dissertation. The analysis used to develop the approach is covered in section 4.7. The performance and energy dissipation of the proposed approach is compared to circuits that were designed with device sizing method that are used for synchronous circuits in section 4.8. The conclusion is presented in section 4.9.

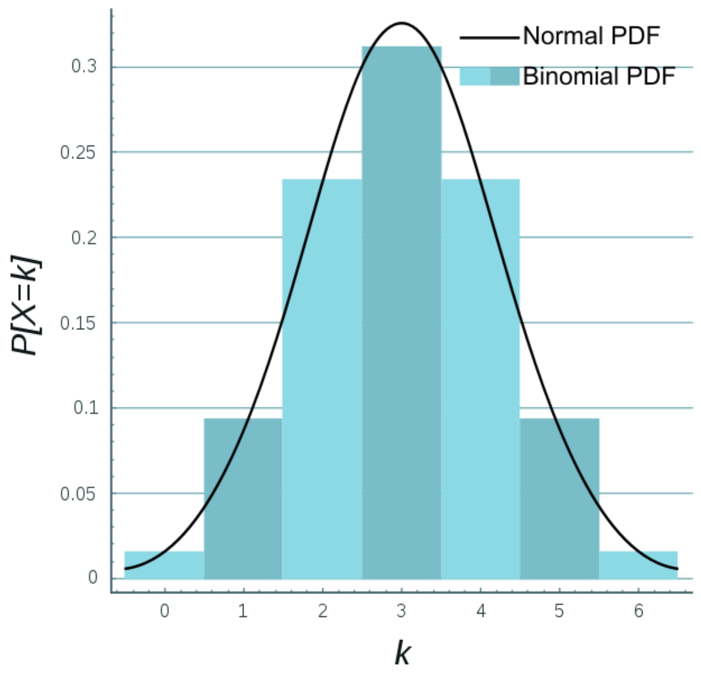


Figure 4.25 Gaussian (Normal) and Discrete (Binomial) distribution

Figure 4.25 shows two distributions, Gaussian or normal and binomial which apply to discrete numbers for digital system. We see that they a very similarity to Gaussian which is by definition continuous. The distributions show the probability that the input appears at the input of the ripple carry adder. If we assume the given distribution, then three is more likely to occur at the input and zero and six are less likely to occur. Therefore, transistors on the green path would be sized larger and transistors on the purple and red path would be sized smaller. Let take a closer look at the fundamental principles of this proposed approach.

4.6 Approach

A novel self-timed circuit device sizing approach is presented in this section that is based on the optimization of circuit device size for a specified input distribution to minimize circuit average completion time.

4.6.1 Circuit Device Sizing with Input Distribution Data without branching effort

The performance of the circuits realized with circuit device sizing with input distribution data without branching effort approach outperforms previously proposed self-timed circuits for the specified input distribution. This due in-part to the fact that the circuit input distribution is not used to size circuit devices. The device sizing approach presented here is based on the Newton-Raphson algorithm which is a root finding algorithm for solving non-linear equations[11] and generally converges rapidly for a given circuit and input distribution. A self-timed full adder is used in this section to demonstrate the proposed device sizing approach. The adder is implemented with domino logic and dynamic input latches. It is shown in fig. 4.24.

The time between the start signal (i.e. self-timed circuit local clock) rising transition and the rising transition on the *Done* node in fig. 4.24 is defined as the completion time of the adder. It is a function of the execution time of the self-timed circuit/system functional block. It depends on the circuit inputs and therefore it is the average of all the active critical path delays for the circuit input space. The active critical path delay is the propagation delay along the longest signal path for a given circuit input over the valid input combinations of a self-timed circuit with *n* primary input bits. The circuit in fig. 4.24 contains four active critical paths. The circuit four active critical paths from the primary inputs (i.e. to the output of the completion detection circuit (i.e. node *Done*) are shown in fig. 4.24 with the respective inputs that activate the paths. The bits that define the numbers in fig. 4.24 are organized as follows: where is the MSB. The normalized propagation delay along the critical path that is activated for input 000 is shown in equation (9). This equation is normalized with respect to the average intrinsic time constant, i.e. $\tau = 17.527$ pSec for TSMC process, of a CMOS process. The propagation delay along the critical paths activated by input 001, 010 (or 100), 011 (or 101) and 110 is shown in equation (10). Finally equation (11) is the normalized delay associated with the path activated by the input word 111. The delay associated with this path and that activated by input 000 (i.e. and) is a piecewise function because the active critical path propagation time is determines by the minimum delay on the path that contains the NAND gate, high-skew inverter and AOI21 gate or NOR gate, high-skew inverter and AOI21 gate.

Recall the formula that was used to calculate the delay, Shown below in equations are the estimated delay associated with the four active paths for input distributions, where,

is input capacitance of AOI21 gate on input B, labeled 10 in fig. 4.24,

- logical effort of AOI21 gate from input C,

- NOR gate parasitic effort and

- probability circuit input is 000,

- probability circuit input is 001,
- probability circuit input is 010,
- probability circuit input is 011,
- probability circuit input is 100,
- probability circuit input is 101,
- probability circuit input is 110,
- probability circuit input is 111.

The expected completion time of the full adder is the average of the active critical path delays , , , and . It equals equation (9). The unknown parameters in Fig. 4.25 related to the device geometry is , , , , , and . The average is:

(9)

completion time of the adder is minimized if these values are set such that,

The Newton-Raphson method is used to find the circuit parameters (i.e. unknown capacitances above) when the expressions in the equation above vanish.

The primary problem encountered with this device sizing approach was the convergence problems. This problem is due to numerical problems in the Newton-Raphson algorithm that is used to solve the non-linear system of equations. Due to problem, the convergence of this technique was very sensitive to the input data distribution. An example of this is the case where it worked well the bimodal input distribution and failed for Gaussian, binomial and uniform distributions.

4.6.2 Circuit Device Sizing with Input Distribution Data with branching effort

This is a new research approach that is based on adjusting the branching effort. The circuit in fig. 4.24 will be used to demonstrate the proposed approach.

Let,

$$B_0 = \frac{C_{NAND3} + C_{NOR4} + C_{SUMA14}}{C_{NAND3}} = \frac{C_{NAND3} + C_{NOR4} + C_{SUMA14}}{\left(C_{NAND3} + C_{NOR4}\right)\left(1 - S\right)}$$

$$B_0^{'} = \frac{C_{NAND3} + C_{NOR4} + C_{SUMA14}}{C_{NOR4}} = \frac{C_{NAND3} + C_{NOR4} + C_{SUMA14}}{\left(C_{NAND3} + C_{NOR4}\right)S}$$

$$B_1 = \frac{C_{NOR15} + C_{SUMD13}}{C_{NOR15}}$$

The stage effort is:

$$f = \sqrt[7]{g_{LD(en)} \cdot g_{NAND} \cdot g_{iinvh}^{3} \cdot g_{AOI21C} \cdot g_{NOR} \cdot B_0 \cdot B_1 \cdot \left(\frac{C_{L3}}{C_{LD(EN)}}\right)}$$

The propagation delay along the path is:

$$D_{110(111)} = 7f + P_{LD(EN)} + P_{NAND} + P_{NOR} + P_{AOI21C} + 3P_{invh} \quad \text{(path though NAND gate 1)}$$

The load at the input of high-skew inverter 12 is:

$$C_{invh12} = \frac{C_{L3} \cdot g_{invh}^{2} \cdot g_{NOR} \cdot B_1}{f^{3}}$$

The propagation delay through the NOR gate is:

$$f^{'} = \sqrt[4]{g_{LD(EN)} \cdot g_{NOR} \cdot g_{invh} \cdot g_{AOI21B} \cdot B_0^{'} \cdot \left(\frac{C_{invh12}}{C_{LD(EN)}}\right)}$$

$$D_{010(100)} = 3f + 4f^{'} + P_{LD(EN)} + 2P_{NOR} + P_{AOI21B} + 3P_{invh} \quad \text{(path though NOR gate 2)}$$

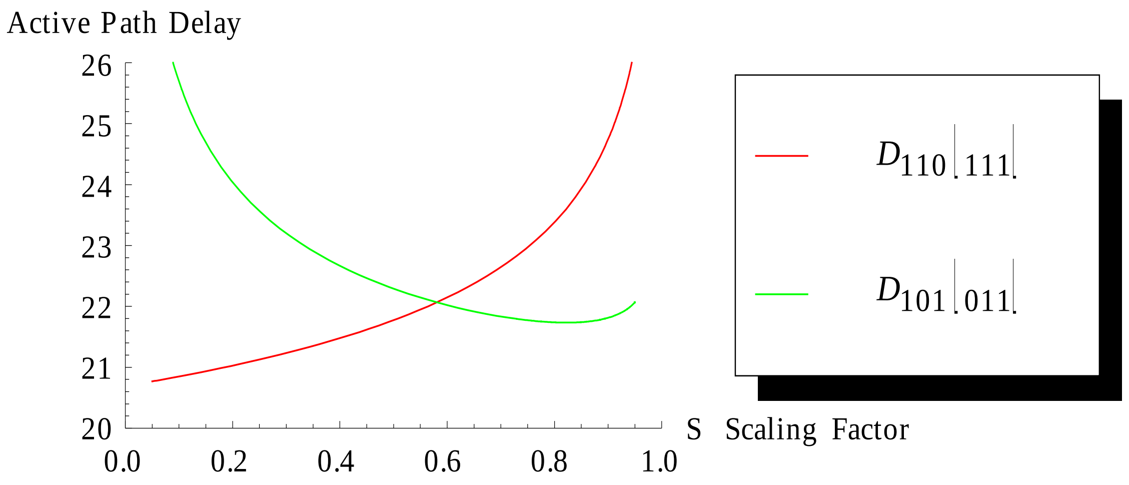


Figure 4.26: Trading delay in one path for delay in another

The inverting logic in the full adder shown in fig. 4.24 is a mirror image of the un-inverted logic. If the input probability distribution is not symmetrically distributed then the delay associated with each side of the adder should be different. This is achieved in the proposed approach by adjust the input capacitance of the sum gates. The branching effort in the circuit associated with path is:

$$B_0 = \frac{C_{NAND} + C_{NOR} + C_{SUMA}}{C_{NAND}}$$

$$B_1 = \frac{C_{NOR} + 2sC_{SUMD}}{C_{NOR}}$$

$$B_1^{'} = \frac{C_{NOR} + 2(1-s)C_{SUMD}}{C_{NOR}}$$

The stage effort of the left and right path in fig. 4.24 is:

$$f_{Left} = \sqrt[7]{g_{LDu} \cdot g_{NAND} \cdot g_{invh}^{3} \cdot g_{AOI21C} \cdot g_{NOR} \cdot B_0 \cdot B_1 \left(\frac{C_{L3}}{C_{LD(EN)}} \right)}$$
$$f_{right} = \sqrt[7]{g_{LDu} \cdot g_{NAND} \cdot g_{invh}^{3} \cdot g_{AOI21C} \cdot g_{NOR} \cdot B_0 \cdot B_1^{'} \left(\frac{C_{L3}}{C_{LD(EN)}} \right)}$$

The path delay of the left and right sides is:

$$D_{Left(001)} = 7 f_{Left} + P_{LD(EN)} + P_{NAND} + P_{AOI21C} + P_{NOR} + 3P_{invh}$$
$$D_{Right(110)} = 7 f_{Right} + P_{LD(EN)} + P_{NAND} + P_{AOI21C} + P_{NOR} + 3P_{invh}$$

The delay associate with each of these paths as *x* is swept from 0.1 to 0.9 is shown in fig. 4.27.

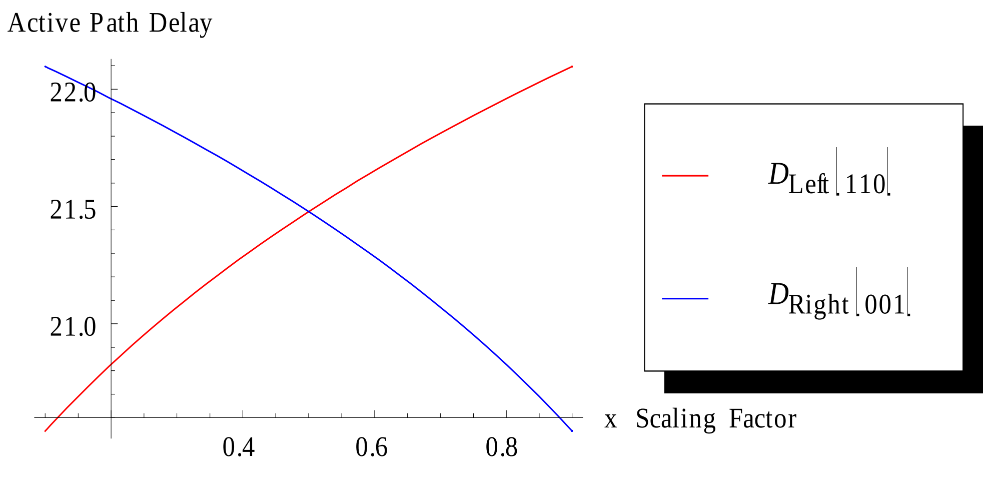


Figure 4.27: Left and Right circuit propagation delay for scaling factor x

Now let's optimize the scaling factors for the circuit shown in fig. 4.24 for the following input distribution. The input distribution is shown in fig. 4.28.

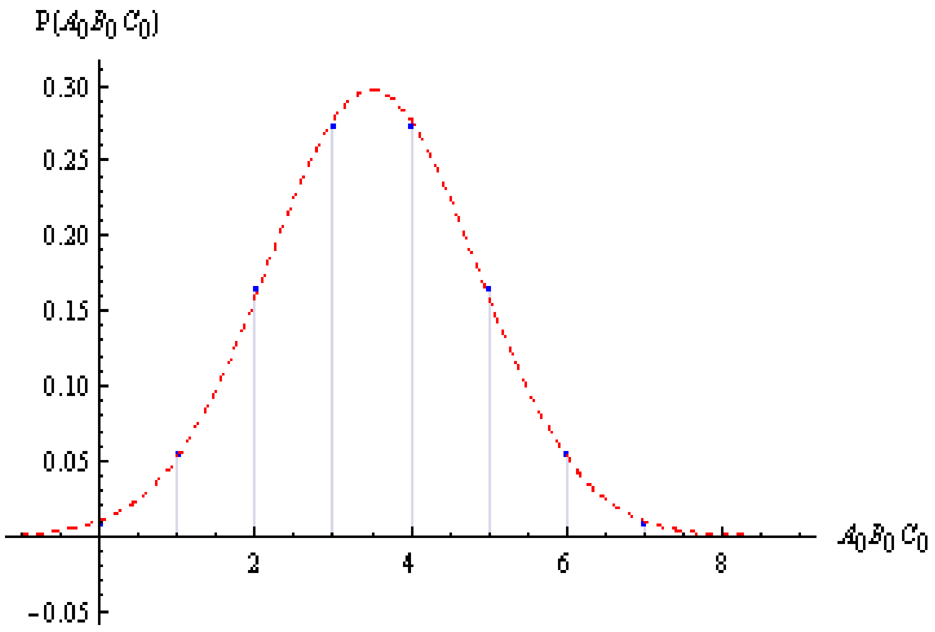


Figure 4.28: Full Adder input Distribution

Therefore, the expected completion time of the full adder is the average of the active critical path delays , and . It equals equation (10). The unknown parameters in fig. 4.25 related to the device geometry is:

and .

$$D_{avg}=W_1D_{001}+(W_0+W_2+W_4)D_{010}+W_6D_{110}+(W_5+W_3+W_7)D_3 \qquad (10)$$

The average completion time of the adder is minimized if these values are set such that,

$$0 = \frac{\partial D_{avg}}{\partial C_{NAND1}}$$

$$0 = \frac{\partial D_{avg}}{\partial C_{NOR2}}$$

$$0 = \frac{\partial D_{avg}}{\partial C_{NAND3}} \qquad (11)$$

$$0 = \frac{\partial D_{avg}}{\partial C_{invh18}}$$

The Newton-Rapson method is used to find the circuit parameters (i.e. unknown capacitances above) when the expression in equation (11) vanishes.

4.7 Results

Table 4.16 below shows the results of device sizing with branching effort.

Table 4.16: Device capacitance in terms of transistor width

Full Adder Side	Branching Effort	Nominal	Bimodal[]	Binomial[]
Left-Side	B0	3.889	9.22128	2.7882
	B0’	5.8335	3.12388	14.3051
Right-Side	B2	3.889	9.22128	2.7882
	B2’	5.8335	3.12388	14.3051
	B1	2.3335	2.3335	2.3335
	B1’	2.3335	2.3335	2.3335
	Average Propagation Delay		21.7258 (22.9488)	21.2617 (24.0977)
	Speedup		5.629%	13.39%
	Energy % Reduction		11.567%	16.78%

4.8 Conclusions

The performance and energy dissipation of self-timed circuits/systems depend on the circuit gate-level implementation, device sizing and input distribution. The device sizing approach used in previously proposed self-timed circuits is identical to that used for synchronous realizations. Therefore it is only optimized to minimize the propagation delay of all circuit signal paths. The performance and energy dissipation, i.e. average completion time and energy dissipation, of the proposed approach for a self-timed circuit is optimized, with respect to device sizing, for a given input distribution. It is less than realizations that do not considered this feature of the input space. This design process causes the active critical path delay of the circuit paths with the highest probability of being active to be less than the path delay in a realization that does not use input data. It also generates delay paths with larger propagation delay than that in previously proposed self-timed circuits design for path that are rarely used, i.e. paths associated with low probability. Both the performance and energy dissipation of self-timed circuits are reduced if the device sizing is optimized for the input distribution.

In short, performance is restricted by power and as chip density and frequency increase, synchronous designers try to figure out ways to deal with power/performance tradeoff. Can we get a better Energy Delay Product? Asynchronous designers do not have to deal with this tradeoff because of the nature of the logic design; we can use less transistors and operate at faster speeds.

Using self-timed circuits coupled with data profiling, I can exploit the natural properties --faster speeds, less transistors and path sizing– to optimize power dissipation and performance. This gives us a superior Energy Delay Product. This technique is novel because there has been no research that alters the LE formula by manipulation the branching effort to trade delay in one part of the circuit for another. We can essentially control the flow of data by allowing highly probable paths to be sized larger and vice versa. With a 13% increase in performance and 16% decrease in power dissipation.

REFERENCES

1. Sutherland, I., Sproull, B. and Harris D.: "Logical Effort: Designing Fast CMOS Circuits," (Morgan Kaufmann Publishers, San Francisco, CA, 1999)

2. Pollack, F. 1999. New microarchitecture challenges in the coming generations of CMOS process technologies. *International Symposium on Microarchitecture.*

3. Moore, Gordon. "Cramming More Components onto Integrated Circuits," *Electronics Magazine* Vol. 38, No. 8 (April 19, 1965).

4. Rabaey, J.M., Chandrakasan, A. and Borivoje N.:"Digital Integrated Circuits: A Design Perspective, 2nd Ed.,"(Prentice Hall, Upper Saddle River, N J, 2003).

5. Gonzales, R., AND Horowitz, M. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits 31*, 9 (September 1996), 1277-1284.

6. Balasubramanian, P.; Edwards, D.A.; Brej, C.; , "Self-timed full adder designs based on hybrid input encoding," *Design and Diagnostics of Electronic Circuits & Systems, 2009. DDECS '09. 12th International Symposium on,* vol., no., pp.56-61, 15-17 April 2009 doi: 10.1109/DDECS.2009.5012099.

7. I-Chyn Wey; Hwang-Cherng Chow; You-Gang Chen; An-Yeu Wen; , "A fast and power-saving self-timed Manchester carry-bypass adder for Booth multiplier-accumulator design," *Advanced System Integrated Circuits 2004. Proceedings of 2004 IEEE Asia-Pacific Conference on,* vol., no., pp. 50- 53, 4-5 Aug. 2004 doi: 10.1109/APASIC.2004.1349402.

8. Brej, C.; Garside, J.D.; , "A quasi-delay-insensitive method to overcome transistor variation," *VLSI Design, 2005. 18th International Conference on* , vol., no., pp. 368- 373, 3-7 Jan. 2005 doi: 10.1109/ICVD.2005.30.

9. V. Venkatachalam and M. Franz Power Reduction Techniques For Microprocessor Systems, ACM Computing Surveys, Vol. 37, No. 3, September 2005.

10. D. Soudris, C. Piguet, C. Goutis, Designing CMOS Ciruits for low Power, Kluwer Academic Publishers, 1995.

11. Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P.:”Numerical Recipes in C: The Art of Scientific Computing, 2nd Ed.,”(Cambridge University Press, Cambridge, UK, 1992).

12. Harris, D, and Horowitz, “Skew Tolerant Domino Circuits.” IEEE J. Solid-State Circuits, 31(11): 1687-1697, Nov. 1997.

13. F. Pappalardo, G. Visalli, and M. Scarana, “An application-oriented analysis of power/precision tradeoff in fixed and floating-point arithmetic units for VLSI processors,” in Proc. IASTED Conf. Circuits, Signals, and Systems, Dec. 2004, pp. 416–421.

14. Chang, T.-Y. and Hsiao, M.-J., “Carry-select adder using single ripple-carry adder”. Electronics Letters. Volume: 34, Issue: 22,29 Oct. 1998, Pages: 2101 - 2103

15. Youngjoon Kim and Lee-Sup Kim. “A low power carry select adder with reduced area”. Circuits and systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on, Volume: 4,2001. Pages:218~221.

16. Youngjoon Kim and Lee-Sup Kim, “64-bit carry-select adder with reduced area”. Electronics Letters, Volume: 37 Issue: 10 , 10May2001.Pages:614-61j.28.

17. J. M. Rabay, Digital Integrated Circuits, A Design perspective. 1st edition Englewood Cliffs. NI: Prentice-hall. 1996, pp. 392-393.

18. Gustavo A. Ruir, “Evaluation of three 32-bit CMOS adders in DCVS logic for self-timed circuits”. IEEE Journal of Solid- State Circuits, Vol. 33. NO. 4, April 1998.

19. G. Birtwistle and K. S. Stevens, "The family of 4-phase latch protocols," in *14th International Symposium on Asynchronous Circuits and Systems.* IEEE, 2008, pp. 71-82.

20. C. H. Lau, “Self: Self-timed system design technique”. Electronics Letters. Volume: 23, Issue: 6 ,12 Mar. 1987, Pages: 269 -270.

21. Y. K. Tan, Y. C. Lim, “Self-timed system design technique”. Electronics Letters. Volume: 26, Issue: 5 ,1 Mar. 1990, Pages: 284 -286.

22. J. Yang, E. Brunvand, “Self-timed design with dynamic domino circuits”. Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI’03) 2003.

23. N.P. Singh, “A design methodology for self-timed systems,” MIT Computer Science Laboratory Tech. Report. TR-258, Feb. 1981.

24. V. Oklobdzija, B. Zeydel, H. Dao, S. Mathew, R. Krishnamurthy, “Energy-Delay Estimation Technique for High-Performance Microprocessor VLSI Adders,” Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH’03) 2003.

25. H. Zhang, P. Mazumder, “Design of a new sense amplifier flip-flop with improved power delay product,” IEEE International Symposium on circuits and System, ISCAS 2005, p. 1262- 1265, Volume 2, 2005.

26. M. Ghasemzar, M. Pedram, “Power delay product minimization in high-performance 64-bit carry-select adders,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 12, Issue 3, p. 235-244, 2004.

27. J. Choi, K, Lee, Design of CMOS tapered buffer for minimum power-delay product,” IEEE Journal of Solid-State Circuits. Volume 29, Issue 9, p. 1142-1145. 1994.

28. S.F. Oberman, H. Al-Twaijry, and M.J. Flynn, ªThe SNAP Project: Design of Floating Point Arithmetic Units, Proc. 13th IEEE Symp. Computer Arithmetic, pp. 156-165, Asilomar, Calif., July 1997.

29. N.T. Quach and M.J. Flynn, “An Improved Floating Point Addition Algorithm,” Technical Report CSL-TR-90-442, Stanford Univ., June 1990. (available at http://umunhum.stanford.edu/main. html).

30. N.T. Quach and M.J. Flynn, “Design and Implementation of the SNAP Floating-Point Adder,” Technical Report CSL-TR-91-501, Stanford Univ., Dec. 1991. (available at http://umunhum.stanford.edu/main. html).

APPENDIX

5.1 C++ Code for Single Precision, Round to Even Floating Point Adder

```
 #include <iostream.h>

#include <fstream.h>

#include <iomanip.h>

#include <cmath>


int main()

{

float   num1 = 3.99, num2 = 3.99,  largest  = 0.0, smallest =0.0, sum = 0.0, Num1, Num2;

 int    num1_exp =1, num2_exp =1, diff =0, Larger_exp = 0, shift = 1, x=10, larger = 0,
smaller =0, sticky;

char   sign;

cout.setf(ios::fixed, ios::floatfield); //set up floating point

cout.setf(ios::showpoint); //output format



Num1 = num1;

Num2 = num2;

// compare exponents

  if ( num1_exp >=  num2_exp)

{
    larger  = num1_exp;
            smaller  = num2_exp;
    diff = larger - smaller;

while ( diff > 0)
      {
         shift = shift * x;
         diff --;
       }

            num2 = num2 / shift;

            {
                        if  (num1 >= num2)
                        {
                        largest  = num1;
                        smallest  = num2;
                        }
                        else if (num1 < num2)
                        {
        largest  = num2;
                        smallest  = num1;
                        }
            }
```

```
}
 if (num2_exp > num1_exp)
{

    larger  = num2_exp;
            smaller  = num1_exp;
    diff = larger - smaller;
while ( diff > 0)
      {
         shift = shift * x;
         diff --;
       }
            num1 = num1 / shift;

            {
                        if  (num1 >= num2)
                        {
                        largest  = num1;
                        smallest  = num2;
                        }
                        else if (num1 < num2)
                        {
        largest = num2;
                        smallest = num1;
                        }
            }
}

Larger_exp = larger;
sum = largest + smallest;  // add significands
if (sum < 0)
   sign = ‘-‘;
else
  sign = ‘+’;

//Exception Handling
if ((sum > 9.9999) || (sum < 1))
                        sticky = 1;
    Else
                        sticky = 0;

// Normalizing
while ( fabs (sum) > 9.9999)
{
    sum = sum / x;
            Larger_exp ++;
            cout << "Overflow" << endl;
```

```
}


while ( fabs(sum) < 1)

{

   sum = (sum * x);

   Larger_exp --;

   cout << "Underflow" << endl;


}


//Rounding


cout << setw(5) << setprecision(2) <<Num1 << " x 10 ^" << num1_exp << " + " <<
Num2 << " x 10^"  << num2_exp << " = " << sum << "  x 10^" << Larger_exp << endl;

return 0;

}
```

5.2 VHDL Code for Simple, Single Precision, Round to Even Floating Point Adder

```
Library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_arith.all;

use ieee.std_logic_unsigned.all;

--use 16 bits for now SEEEEEEEEMMMMMMMM

entity Thirty_Two_Bit_FP_ADD is

   Port ( x : in STD_LOGIC_vector(31 downto 0);--first number

          y : in  STD_LOGIC_vector(31 downto 0);--second number

         final_Man: out std_logic_vector(23 downto 0);--final sig

         Final_Exponent: out std_logic_vector(7 downto 0);--final exp

         comp: out std_logic_vector(8 downto 0);-- tester

         result : out std_logic_vector (31 downto 0);--answer

         getflag: out std_logic;

         mout: out string(32 downto 1);--message out

         afteradd4 : out std_logic_vector (27 downto 0));--tester

end Thirty_Two_Bit_FP_ADD;


architecture structural of Thirty_Two_Bit_FP_ADD is

Signal  Temp,Big_Exp, twos_comp_Temp,twos_comp_y,
ShiftSmallSig,changeBigExp,changebigexp2, NewExp, original_exp,newdiff,
temp_exp1, temp_exp, tempdiff, twos_comp_tempdiff, updated_exp,
add2exp,add2exp2,final_exp1 , stky, xexp,  yexp, final_exp2, stky2,  tempexp,tempexp1,
tempexp2, twos_comp_grstky, final_exp_EX,
final_Exp_EX1,Small_Exp,Final_Exp_EX2: std_logic_vector ( 7 downto 0);
```

```
Signal  afterAdd,twos_comp_Afteradd,  grstky,
Tempsig,twos_comp_tempSig,final_man_EX: std_logic_vector(23 downto 0);

Signal prependZero, comp1,pickSmallSig,xorofsign,
temp3,BigSigSign,Temp1,temp2,temp6, temp7,temp8,temp9,temp10,temp12,
CO,ovfl,ovefl, udfl, NAN,Post_shift,updateexp2,check_exp, guard_bit, updateExp,
guard_bit1, guard_bit2, guard_bit3, guard_bit4, guard_bit5,s, Rd2n, sticky,Addl,flag,
fflag, fflag2,fflag3, setflag, dflag, nflag, inflag,underflag, underflag2, bpflag, zflag,
zflag2, getAns, gflag, getMessage: std_logic;

Signal Ready2Add, eflag  : std_logic_vector(8 downto 0);

Signal Ready2Add2,Ready2add3, BigSig2,twos_comp_bigsig,temp4,
BigSigready,int_part: std_logic_vector(23 downto 0);

signal BigSig, Small_Sig,xman,yman, Final_man_EX1,final_man2,Final_Man_EX2 :
std_logic_vector(22 downto 0);

signal smallsigready,afteradd3,twos_comp_Ready2Add3,Small_Sig2,afteradd2:
std_logic_vector(26 downto 0);

signal message, message2,  final_message:string (32 downto 1) ;

signal afteradd6, rounded_num,afteradd12,afteradd13: std_logic_vector(24 downto 0);

signal sel:std_logic_vector(1 downto 0);

signal afteradd1, afteradd0:std_logic_vector (27 downto 0);


component Eight_bit_subtractor is

Port(A, B: in std_logic_vector (7 downto 0);

cout: out std_logic;

Sum: out std_logic_vector(7 downto 0));

End component;
```

```
component Mux is
   Port ( a : in  STD_LOGIC_vector (7 downto 0);
          b : in  STD_LOGIC_vector (7 downto 0);
          sel : in  STD_LOGIC;
          e : out std_logic_vector (7 downto 0));
end component;

component Mux2 is
   Port ( a2 : in  STD_LOGIC;
          B2 : in  STD_LOGIC;
          Sel2 : in  STD_LOGIC;
          e2: out std_logic);
end component;

component  Mux3 is
   Port ( a : in  STD_LOGIC_vector (8 downto 0);
          b : in  STD_LOGIC_vector (8 downto 0);
          sel : in  STD_LOGIC;
          e : out std_logic_vector (8 downto 0));
end component;

component Mux24 is
   Port ( a : in  STD_LOGIC_vector (23 downto 0);
          b : in  STD_LOGIC_vector (23 downto 0);
          sel : in  STD_LOGIC;
          e : out std_logic_vector (23 downto 0));
end component;

component  Mux23 is
   Port ( a : in  STD_LOGIC_vector (22 downto 0);
          b : in  STD_LOGIC_vector (22 downto 0);
          sel : in  STD_LOGIC;
          e : out std_logic_vector (22 downto 0));
end component;

component  Mux26 is
   Port ( a : in  STD_LOGIC_vector (26 downto 0);
          b : in  STD_LOGIC_vector (26 downto 0);
          sel : in  STD_LOGIC;
          e : out std_logic_vector (26 downto 0));
end component;

component  Mux31 is
   Port ( a : in  STD_LOGIC_vector (31 downto 0);
          b : in  STD_LOGIC_vector (31 downto 0);
          sel : in  STD_LOGIC;
          e : out std_logic_vector (31 downto 0));
end component;
```

```
component twos_comp is

Port ( B: in std_logic_vector (7 downto 0);

        AplusB1: out std_logic_vector (7 downto 0));

End component;


component twos_comp2 is

Port ( B: in std_logic_vector (8 downto 0);

     AplusB1: out std_logic_vector (8 downto 0));

End component;


component twos_comp3 is

Port ( B: in std_logic_vector (9 downto 0);

     AplusB1: out std_logic_vector (9 downto 0));

End component;


component twos_comp24 is

Port ( B: in std_logic_vector (23 downto 0);

     AplusB1: out std_logic_vector (23 downto 0));

End component;


component twos_comp27 is

Port ( B: in std_logic_vector (26 downto 0);

     AplusB1: out std_logic_vector (26 downto 0));

 End component;
```

```
component Eight_bit_Adder is

Port(A, B: in std_logic_vector(8 downto 0);

cout: out std_logic;

Sum: out std_logic_vector(8 downto 0));

End component;


component Twenty4_Bit_Adder is

Port(A, B: in std_logic_vector (23 downto 0);

        cout: out std_logic;

       Sum: out std_logic_vector(23 downto 0));

End component;


component Twenty5_Bit_Adder is

Port(A, B: in std_logic_vector (24 downto 0);

        cout: out std_logic;

       Sum: out std_logic_vector(24 downto 0));

End component;


component CIE is

Port(A, B: in std_logic_vector (7 downto 0);

        cout: out std_logic;

       Sum: out std_logic_vector(7 downto 0));

end component;
```

```
component  Mux4 is
    Port ( a : in  STD_LOGIC_vector (9 downto 0);
           b : in  STD_LOGIC_vector (9 downto 0);
           sel : in  STD_LOGIC;
           e : out std_logic_vector (9 downto 0));
end component;

Begin
  xExp <= x(30 downto 23);
  yExp <= y(30 downto 23);
  xMan <= x(22 downto 0);
  yMan <= y(22 downto 0);

  Process (xexp, yexp,xman, yman,stky, flag)
      begin
   If (xExp = "00000000") then
    if (xMan = "00000000000000000000000") then  stky(0) <= '1'; flag <= '1';
      else stky(1) <= '1'; setflag <= '1'; --renomalize
  end if;
  end if;

  If (xExp = "11111111")  then
   if (xMan = "00000000000000000000000") then stky(6) <= '1'; flag <= '1';--INFN
    else stky(7) <= '1'; flag <= '1';--NAN
```

```
end if;

end if;


   If (yExp = "11111111")  then

    if (yMan = "00000000000000000000000") then stky(2) <= '1';flag <= '1';--INFN

     else stky(3) <= '1'; flag <= '1';--NAN

 end if;

 end if;


   If (yExp = "00000000") then

    if (yMan = "00000000000000000000000") then  stky(4) <= '1'; flag <= '1';

     else stky(5) <= '1'; setflag <= '1';--renomalize

 end if;

 end if;


end process;


TC3: twos_comp -- getting negative for addition

Port map (B(7 downto 0) => y(30 downto 23), AplusB1 (7 downto 0) => twos_comp_y(7
downto 0));


--SUB EXP X-Y= DIFF/TEMP

s0: Eight_bit_subtractor
```

```
Port map (A(7 downto 0) => X(30 downto 23), B(7 downto 0) => twos_comp_Y(7
downto 0), Cout => Temp2, sum(7 downto 0) => Temp(7 downto 0));


Temp1 <= not temp(7);


--Select big exponent


M0: Mux

Port map (A(7 downto 0) => X(30 downto 23), B(7 downto 0) => Y(30 downto 23), e(7
downto 0) => Big_Exp(7 downto 0), Sel => temp(7));


--Select small exponent

M11: Mux

Port map (A(7 downto 0) => X(30 downto 23), B(7 downto 0) => Y(30 downto 23), e(7
downto 0) => Small_Exp(7 downto 0), Sel => temp1);


--select Big_Sig

M1: Mux23

Port map (A(22 downto 0) => X(22 downto 0), B(22 downto 0) => Y(22 downto 0), e(22
downto 0) => BigSig(22 downto 0), Sel => temp(7));



--2'S COMP OF DIFF IN EXP

TC1: twos_comp

Port map (B(7 downto 0) => temp(7 downto 0), AplusB1 (7 downto 0) =>
twos_comp_Temp(7 downto 0));
```

```
--select THE DIFF OF COMPLEMENT Or DIFF FOR SHIFT AMOUNT

M4: Mux

Port map (A(7 downto 0) => temp(7 downto 0), B(7 downto 0) => twos_comp_Temp (7
downto 0), e(7 downto 0) => ShiftSmallSig(7 downto 0), Sel  =>  temp(7));

--24 bit shift?

guard_bit1 <= (shiftsmallsig(7) or shiftsmallsig(6) or shiftsmallsig(5));

guard_bit2 <= (shiftsmallsig(4) AND shiftsmallsig(3));

guard_bit3 <= (shiftsmallsig(2) or shiftsmallsig(1) or shiftsmallsig(0));

guard_bit4 <= (guard_bit2 and guard_bit3);

guard_bit5 <= (guard_bit1 or guard_bit4);


fflag2 <= '0' when guard_bit5 = '1' else '1';


--PREPEND "1" TO BIG SIG

--prependZero <= (big_exp(6) and big_exp(5) and big_exp(4) and big_exp(3) and
big_exp(2) and big_exp(1) and big_exp(0));

BigSig2 <= '0' & bigsig when big_exp ="00000000" else '1' & BigSig;


--get sign of big sig

M26: Mux2

Port map (A2 => x(31), B2 => y(31), e2 => BigSigSign, Sel2 =>  temp(7));
```

```
--select small sig

M2: Mux23

Port map (A(22 downto 0) => X(22 downto 0), B(22 downto 0) => Y(22 downto 0), e(22
downto 0) => Small_Sig(22 downto 0), Sel  =>  Temp1);


-- PREPEND "1" TO SMALL SIG

Ready2Add2 <= '0' & small_sig when small_exp = "00000000" else '1' & small_sig;


--SHIFT SMALL SIG


Ready2Add3 <=  Ready2Add2 when ("00000000" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 24) when
("00011000" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 23) when
("00010111" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 22) when
("00010110" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 21) when
("00010101" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 20) when
("00010100" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 19) when
("00010011" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 18) when
("00010010" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 17) when
("00010001" = ShiftSmallSig) else
```

```
                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 16) when
("00010000" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 15) when
("00001111" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 14) when
("00001110" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 13) when
("00001101" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 12) when
("00001100" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 11) when
("00001011" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 10) when
("00001010" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 9) when
("00001001" = ShiftSmallSig) else

                                        to_stdlogicvector(to_bitvector(Ready2Add2) srl 8) when
("00001000" = ShiftSmallSig) else

            to_stdlogicvector(to_bitvector(Ready2Add2) srl 7) when ("00000111" =
ShiftSmallSig) else

            to_stdlogicvector(to_bitvector(Ready2Add2) srl 6) when ("00000110" =
ShiftSmallSig) else

            to_stdlogicvector(to_bitvector(Ready2Add2) srl 5) when ("00000101" =
ShiftSmallSig) else

            to_stdlogicvector(to_bitvector(Ready2Add2) srl 4) when ("00000100" =
ShiftSmallSig) else

            to_stdlogicvector(to_bitvector(Ready2Add2) srl 3) when ("00000011" =
ShiftSmallSig) else
```

```
             to_stdlogicvector(to_bitvector(Ready2Add2) srl 2) when ("00000010" =
ShiftSmallSig) else

             to_stdlogicvector(to_bitvector(Ready2Add2) srl 1) when ("00000001" =
ShiftSmallSig) else

            "000000000000000000000000";



 --Generate Guard, Round and Sticky

 --GRSTKY(23 downto 0)<= "000000000000000000000000";

 Process(Ready2add2,shiftsmallsig, GRSTKY,Ready2Add3)

 begin

 if  ("00000001" = ShiftSmallSig) then GRSTKY<= Ready2add2(0) &
"00000000000000000000000";-- Ready2Add3 <=
to_stdlogicvector(to_bitvector(Ready2Add2) srl 1);

 elsif ("00000010" = ShiftSmallSig) then GRSTKY <= Ready2add2(1 downto 0) &
"0000000000000000000000";-- Ready2Add3
<=to_stdlogicvector(to_bitvector(Ready2Add2) srl 2);

 elsif ("00000011" = ShiftSmallSig) then GRSTKY<= Ready2add2(2 downto 0) &
"000000000000000000000";-- Ready2Add3
<=to_stdlogicvector(to_bitvector(Ready2Add2) srl 3);

 elsif ("00000100" = ShiftSmallSig) then GRSTKY <= Ready2add2(3 downto 0) &
"00000000000000000000";-- Ready2Add3
<=to_stdlogicvector(to_bitvector(Ready2Add2) srl 4);

 elsif ("00000101" = ShiftSmallSig) then GRSTKY<= Ready2add2(4 downto 0) &
"0000000000000000000";-- Ready2Add3
<=to_stdlogicvector(to_bitvector(Ready2Add2) srl 5);

 elsif ("00000110" = ShiftSmallSig) then GRSTKY<= Ready2add2(5 downto 0) &
"000000000000000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2)
srl 6);
```

```
 elsif ("00000111" = ShiftSmallSig) then GRSTKY<= Ready2add2(6 downto 0) &
"00000000000000000";-- Ready2Add3 <= to_stdlogicvector(to_bitvector(Ready2Add2)
srl 7);

 elsif ("00001000" = ShiftSmallSig) then GRSTKY<= Ready2add2(7 downto 0) &
"0000000000000000";-- Ready2Add3 <= to_stdlogicvector(to_bitvector(Ready2Add2)
srl 8);

 elsif ("00001001" = ShiftSmallSig) then GRSTKY<= Ready2add2(8 downto 0) &
"000000000000000";-- Ready2Add3 <= to_stdlogicvector(to_bitvector(Ready2Add2) srl
9);

 elsif ("00001010" = ShiftSmallSig) then GRSTKY<= Ready2add2(9 downto 0) &
"00000000000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl
10);

 elsif ("00001011" = ShiftSmallSig) then GRSTKY<= Ready2add2(10 downto 0) &
"0000000000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl
11);

 elsif ("00001100" = ShiftSmallSig) then GRSTKY<= Ready2add2(11 downto 0) &
"000000000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 12);

 elsif ("00001101" = ShiftSmallSig) then GRSTKY<= Ready2add2(12 downto 0) &
"00000000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 13);

 elsif ("00001110" = ShiftSmallSig) then GRSTKY <= Ready2add2(13 downto 0) &
"0000000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 14);

 elsif ("00001111" = ShiftSmallSig) then GRSTKY<= Ready2add2(14 downto 0) &
"000000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 15);

 elsif ("00010000" = ShiftSmallSig) then GRSTKY<= Ready2add2(15 downto 0) &
"00000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 16);

 elsif ("00010001" = ShiftSmallSig) then GRSTKY<= Ready2add2(16 downto 0) &
"0000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 17);

 elsif ("00010010" = ShiftSmallSig) then GRSTKY<= Ready2add2(17 downto 0) &
"000000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 18);

 elsif ("00010011" = ShiftSmallSig) then GRSTKY<= Ready2add2(18 downto 0) &
"00000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 19);
```

```
 elsif ("00010100" = ShiftSmallSig) then GRSTKY<= Ready2add2(19 downto 0) &
"0000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 20);

 elsif ("00010101" = ShiftSmallSig) then GRSTKY<= Ready2add2(20 downto 0) &
"000";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 21);

 elsif ("00010110" = ShiftSmallSig) then GRSTKY<= Ready2add2(21 downto 0) &
"00";-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 22);

 elsif ("00010111" = ShiftSmallSig) then GRSTKY<= Ready2add2(22 downto 0) & '0';--
Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 23);

 elsif ("00011000" = ShiftSmallSig) then GRSTKY(23 downto 0) <= Ready2add2(23
downto 0);-- Ready2Add3 <=to_stdlogicvector(to_bitvector(Ready2Add2) srl 24);

 else   GRSTKY<= "000000000000000000000000";-- Ready2Add3 <=  Ready2Add2;

 end if;

 end process;


 -- get sticky

 S<= (grstky(21)or grstky(20)or grstky(19) or grstky(18) or grstky(17) or grstky(16) or
grstky(15) or grstky(14) or grstky(13) or grstky(12) or grstky(11) or grstky(10) or
grstky(9) or grstky(8) or grstky(7) or grstky(6) or grstky(5) or grstky(4) or grstky(3) or
grstky(2) or grstky(1) or grstky(0));



--2?s comp of small sig if it is negative


TCT0: twos_comp24

Port map (b(23 downto 0) => Ready2add3(23 downto 0),  AplusB1 (23 downto 0) =>
twos_comp_Ready2Add3(23 downto 0));
```

```
TCT53: twos_comp

port map (b(7 downto 3) => "00000", b(2 downto 1) => grstky (23 downto 22),b(0) => s,
AplusB1(7 downto 0) => twos_comp_grstky(7 downto 0));

--get sign of small sig

picksmallsig <= x(31) xor y(31);


smallsigready <= Ready2add3(23 downto 0) & GRSTKY(23 downto 22) & S;


--SELECTS SMALL SIG OR 2'S COMP OF SMALL SIG

M57: Mux26

Port map (A(26 downto 3) =>  Ready2add3(23 downto 0), A(2 downto 1) =>
GRSTKY(23 downto 22), A(0) => S, B(26 downto 3) => twos_comp_Ready2Add3(23
downto 0), B(2 downto 0) => twos_comp_grstky(2 downto 0), e(26 downto 0) =>
Small_Sig2(26 downto 0), Sel  =>  PickSmallSig);


--ADD BIG SIG TO SMALL SIG

FA11: Twenty4_Bit_Adder

Port map (A(23 downto 0) => BigSig2(23 downto 0), B(23 downto 0) => Small_Sig2(26
downto 3), Cout => ovfl, sum(23 downto 0) => Tempsig(23 downto 0));


--twos comp of big&small sig addition

TCT10: twos_comp24

Port map (b(23 downto 0) => tempsig(23 downto 0), AplusB1 (23 downto 0) =>
twos_comp_tempSig(23 downto 0));


Co <= not ovfl;
```

```
comp1 <= (picksmallsig and (tempsig(23)) and Co);


--select tempsig or twos comp to tempsig


M15: Mux24

Port map (A(23 downto 0) => tempsig(23 downto 0), B(23 downto 0) =>
twos_comp_tempSig(23 downto 0), e(23 downto 0) => afteradd(23 downto 0), Sel  =>
comp1);


--shift afteradd to normalize?


updateExp<= ovfl when picksmallsig = '0' else

            '0';

afteradd0<= updateExp & afteradd & small_sig2(2 downto 0);


--afteradd1<= afteradd0 (27 downto 0) when picksmallsig = '1' else

           --ovfl & afteradd0 (26 downto 0);

 afteradd1 <= to_stdlogicvector(to_bitvector (Afteradd0) srl 1) when afteradd0(27) = '1'
else afteradd0;

afteradd2 <= afteradd1 (26 downto 0);


--increment exp?

s5: Eight_bit_subtractor

Port map (A(7 downto 0) => Big_exp(7 downto 0), B(7 downto 1) => "0000000", B(0)
=> updateExp, Cout => Temp2, sum(7 downto 0) => TempExp(7 downto 0));
```

```
--Overflow?

Eflag(0) <= '1' when temp2 = '1' else '0';

Eflag(1) <= '1' when tempExp= "11111111" else '0';


--Underflow?

Eflag(2) <= '1' when tempExp = "00000000" else '0';


 --afteradd2 <= afteradd1 & small_sig2(2 downto 0)


   --Normalize


process( afteradd2,afteradd3, changeBigExp, underflag,TempExp1 )

   begin


    if afteradd2(26)= '1' then afteradd3 <= afteradd2; changeBigExp <=
"00000000";underflag <= '0';


    elsif afteradd2(25)= '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
1);changeBigExp <= "11111111"; underflag <= '0';

    elsif afteradd2(24)= '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
2);changeBigExp <= "11111110"; underflag <= '0';

    elsif afteradd2(23)= '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
3);changeBigExp <= "11111101"; underflag <= '0';

    elsif afteradd2(22)= '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
4);changeBigExp <= "11111100"; underflag <= '0';
```

```
   elsif afteradd2(21)= '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
5);changeBigExp <= "11111011"; underflag <= '0';

   elsif afteradd2(20)= '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
6);changeBigExp <= "11111010"; underflag <= '0';

   elsif afteradd2 (19)= '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
7);changeBigExp <= "11111001"; underflag <= '0';

   elsif afteradd2 (18) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2)
sll 8);changeBigExp <= "11111000"; underflag <= '0';

   elsif afteradd2 (17) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2)
sll 9);changeBigExp <= "11110111"; underflag <= '0';

   elsif afteradd2 (16) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2)
sll 10);changeBigExp <= "11110110"; underflag <= '0';

   elsif afteradd2 (15) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2)
sll 11);changeBigExp <= "11110101"; underflag <= '0';

   elsif afteradd2 (14) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2)
sll 12);changeBigExp <= "11110100"; underflag <= '0';

   elsif afteradd2 (13) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2)
sll 13);changeBigExp <= "11110011"; underflag <= '0';

   elsif afteradd2 (12) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2)
sll 14);changeBigExp <= "11110010"; underflag <= '0';

   elsif afteradd2 (11) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2)
sll 15);changeBigExp <= "11110001"; underflag <= '0';

   elsif afteradd2 (10) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2)
sll 16);changeBigExp <= "11110000"; underflag <= '0';

   elsif afteradd2 (9) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
17);changeBigExp <= "11101111"; underflag <= '0';

   elsif afteradd2 (8) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
18);changeBigExp <= "11101110"; underflag <= '0';
```

```
   elsif afteradd2 (7) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
19);changeBigExp <= "11101101"; underflag <= '0';

   elsif afteradd2 (6) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
20);changeBigExp <= "11101100"; underflag <= '0';

   elsif afteradd2 (5) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
21);changeBigExp <= "11101011"; underflag <= '0';

   elsif afteradd2 (4) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
22);changeBigExp <= "11101010"; underflag <= '0';

   elsif afteradd2 (3) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
23);changeBigExp <= "11101001"; underflag <= '0';

   elsif afteradd2 (2) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
24);changeBigExp <= "11101000"; underflag <= '0';

   elsif afteradd2 (1) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
25);changeBigExp <= "11100111"; underflag <= '0';

   elsif afteradd2 (0) = '1' then afteradd3 <= to_stdlogicvector(to_bitvector(Afteradd2) sll
26);changeBigExp <= "11100110"; underflag <= '0';

   else underflag <= '1'; afteradd3 <= "000000000000000000000000000";  TempExp1 <=
"00000000"; changeBigExp <= "00000000";

       end if;


    end process;

--inflag <= '0' when nflag = '1' else '1';

fflag <= '0' when flag ='1' else '1';

dflag <= '1' when setflag = '1' else '0';

underflag2 <= '1' when underflag = '1' else '0';

tempexp2 <= tempexp1 when underflag2 = '1' else tempexp;

bpflag <= '0' when (stky(4) or stky(0)) = '1' else '1';
```

```
fflag3 <= not fflag2;

zflag <= underflag2 or fflag3;

zflag2 <= '0' when zflag = '1' else '1';

  C1: CIE

   Port map (A(7 downto 0) => TempExp2(7 downto 0), B(7 downto 0) =>
ChangeBigExp(7 downto 0), Cout => Temp6, sum(7 downto 0) => Temp_Exp(7 downto
0));


  --Overflow?

  Eflag(3) <= '1' when temp6 = '1' else '0';

  Eflag(4) <= '1' when temp_Exp= "11111111" else '0';


  --Underflow?

  Eflag(5) <= '1' when temp_Exp = "00000000" else '0';


  --Rounding logic

  Sticky <= ((afteradd3(1)) or (afteradd3(0)));

  Rd2N <= ((afteradd3(3)) or (sticky));

  Addl <= ((Rd2N) and (afteradd3(2)));

  --RNI  Rd2N <= ((afteradd3(2)) or (sticky));

  --RNI  Addl <= ((Rd2N) and (not(BigSigSign)));


--Add one to odd sig
```

```
--C3: CIE

--ADD ulp

 afteradd6 <="000000000000000000000000" &  ADDl;

FA3: Twenty5_Bit_Adder

Port map (A(24 downto 0) => afteradd3(26 downto 2), B(24 downto 0) => afteradd6(24
downto 0), Cout => Post_shift, sum(24 downto 0) => rounded_num (24 downto 0));


updateExp2<= '0' when Post_shift  = '0' else

            '1';


--increment exp?

s22: Eight_bit_subtractor

Port map (A(7 downto 0) => Temp_Exp(7 downto 0), B(7 downto 1) => "0000000", B(0)
=> updateExp2, Cout => Temp12, sum(7 downto 0) => Final_Exp1(7 downto 0));


--Overflow?

eflag(6) <= '1' when temp12 = '1' else '0';

eflag(7) <= '1' when Final_Exp1= "11111111" else '0';


--Underflow?

eflag(8) <= '1' when Final_Exp1 = "00000000" else '0';


 process(stky, stky2)

     begin
```

```
if stky(0) = '1' then stky2(0) <= '1'; else stky2 (0) <='0'; end if;

if stky(1) = '1' then stky2(1) <= '1'; else stky2 (1)<='0'; end if;

if stky(2) = '1' then stky2(2) <= '1'; else stky2 (2)<='0'; end if;

if stky(3) = '1' then stky2(3) <= '1'; else stky2 (3)<='0'; end if;

if stky(4) = '1' then stky2(4) <= '1'; else stky2 (4)<='0'; end if;

if stky(5) = '1' then stky2(5) <= '1'; else stky2 (5)<='0'; end if;

if stky(6) = '1' then stky2(6) <= '1'; else stky2 (6)<='0'; end if;

if stky(7) = '1' then stky2(7) <= '1'; else stky2 (7)<='0'; end if;


end process;


process (stky2, final_exp2, final_man2, message)
   begin
Case stky2 is
   when "00000001" => final_exp2 <= y(30 downto 23);final_man2 <= y(22 downto 0);
   when "00000010" => message <= "first number is subnormal       ";
   when "00000100" => message <= "Second Num/Result equal Infinity";
   when "00001000" => message <= "Second Number/Result equals NAN-";
   when "00010000" => final_exp2 <= x(30 downto 23);final_man2 <= x(22 downto 0);
   when "00100000" => message <= "Second number must be normalized";
   when "01000000" => message <= "First Num/Result equals Infinity";
   when "10000000" => message <= "First Number/Result equals NAN- ";
   when "01000100" => message <= "Result equals Infinity         -";
   when "01000001" => message <= "Result equals Infinity         -";
```

```
    when "00010100" => message <= "Result equals Infinity         -";
    when "00010001" => message <= "Result equals NAN              ";
    when "00000101" => message <= "Second Num/Result equal Infinity";
    when "01010000" => message <= "First Num/Result equals Infinity";
    when "01001000" => message <= "Result equals NAN              ";
    when "10000100" => message <= "Result equals NAN              ";
    when "00001001" => message <= "Result equals NAN              ";
    when "10010000" => message <= "Result equals NAN              ";

    when others => message <=     "                                " ;
  end case;
  end process;



process (eflag, message2)
    begin
Case eflag is
    when "000000001" => message2 <= "Result equals NAN - Overflow    ";
    when "000000010" => message2 <= "Result equals NAN - Overflow    ";
    when "000000100" => message2 <= "Result equals NAN - Underflow 1 ";
    when "000001000" => message2 <= "Result equals NAN - Overflow    ";
    when "000010000" => message2 <= "Result equals NAN - Overflow    ";
    when "000100000" => message2 <= "Result equals NAN - Underflow 2 ";
    when "001000000" => message2 <= "Result equals NAN - Overflow    ";
    when "010000000" => message2 <= "Result equals NAN - Overflow    ";
    when "100000000" => message2 <= "Result equals NAN - Underflow 3 ";
    when "010010000" => message2 <= "Result equals NAN - Overflow    ";
    when "000100100" => message2 <= "Result equals NAN - Underflow 4 ";
    when "100100100" => message2 <= "Result equals NAN - Underflow 5 ";
    when "000000011" => message2 <= "Result equals NAN - Overflow    ";
    when "000001011" => message2 <= "Result equals NAN - Overflow    ";
    when "000011011" => message2 <= "Result equals NAN - Overflow    ";
    when "001011011" => message2 <= "Result equals NAN - Overflow    ";
    when "010010010" => message2 <= "Result equals NAN - Overflow    ";
    when others =>      message2 <= "                                ";
  end case;
  end process;



M299: Mux23
Port map (A(22 downto 0) => final_man2(22 downto 0), B(22 downto 0) => y (22 downto 0), e(22 downto 0) => Final_man_EX(22 downto 0), Sel  =>  fflag);
M298: Mux
Port map (A(7 downto 0) => final_exp2(7 downto 0), B(7 downto 0) => y(30 downto 23), e(7 downto 0) => final_exp_EX(7 downto 0), Sel => fflag);
getAns <= (bpflag and fflag);
M279:Mux23
```

```
Port map (A(22 downto 0) => final_man_EX(22 downto 0), B(22 downto 0) =>
rounded_num(23 downto 1), e(22 downto 0) => Final_man_EX1(22 downto 0), Sel  =>
getans);

M278: Mux

Port map (A(7 downto 0) => final_exp_EX(7 downto 0), B(7 downto 0) => Final_Exp1(7
downto 0), e(7 downto 0) => final_exp_EX1(7 downto 0), Sel => getans);



--M289: Mux23

--Port map (A(22 downto 0) => final_man_EX(22 downto 0), B(22 downto 0) =>
Final_man_EX2(22 downto 0), e(22 downto 0) => Final_man_EX1(22 downto 0), Sel
=>  getAns);

--M288: Mux

--Port map (A(7 downto 0) => final_exp_EX(7 downto 0), B(7 downto 0) =>
Final_Exp_EX2(7 downto 0), e(7 downto 0) => final_exp_EX1(7 downto 0), Sel =>
getAns);


getMessage <= (eflag(8) or eflag(7) or eflag(6) or eflag(5) or eflag(4) or eflag(3) or
eflag(2) or eflag(1) or eflag(0));


final_message <= message when (fflag ='0' or dflag ='1') else message2;


final_Man <=  '1' & Final_man_EX1;

Final_Exponent <= final_Exp_EX1;

--tmpcomp <= final_exp1;

comp <= Eflag;

afteradd4 <=  afteradd2 & '0';
```

```
getflag <= getmessage;

mout <= final_message;

result <=  BigSigSign & final_Exp_EX1 & Final_man_EX1;

--z<= New_binout;



end structural;
```