ARCHITECTURAL ENERGY-DELAY ASSESSMENT OF ABACUS MULTIPLIER WITH RESPECT TO OTHER MULTIPLIERS

A THESIS SUBMITTED TO

THE BOARD OF GRADUATE PROGRAMS OF

MIDDLE EAST TECHNICAL UNIVERSITY,

NORTHERN CYPRUS CAMPUS

BY

DİDEM GÜRDÜR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR

THE DEGREE OF MASTER OF SCIENCE

IN

SUSTAINABLE ENVIRONMENT AND ENERGY SYSTEMS PROGRAMME

JULY 2013

Approval of the Board of Graduate Programs

_____

Prof. Dr. Erol Taymaz

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____

Assist. Prof. Dr. Ali Muhtaroğlu

Head of Department

This is to certify that we have read this thesis and that, in our opinion, it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____

Assoc. Prof. Dr. Ali Muhtaroğlu

Supervisor

Examining Committee Members

Assist. Prof. Dr. Ali Muhtaroğlu (METU NCC, EE) _____

Assoc. Prof. Dr. Cüneyt Bazlamaçcı (METU, EE) _____

Assist. Prof. Dr. Yeliz Yeşilada (METU, CNG) _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Didem GÜRDÜR

ABSTRACT

ARCHITECTURAL ENERGY-DELAY ASSESSMENT OF ABACUS MULTIPLIER WITH RESPECT TO OTHER MULTIPLIERS

Gürdür, Didem

M.Sc., Sustainable Environment and Energy Systems

Supervisor : Assist. Prof. Dr. Ali Muhtaroğlu

July 2013, 82 pages

This study presents a logic implementation for the recently proposed ABACUS integer multiplier architecture and compares it with other fundamental multipliers. The ABACUS m x n implementation was modeled, simulated, and evaluated using the PETAM (Power Estimation Tool for Array Multipliers) tool developed during this study, against Carry Save Array Multiplier (CSAM), Ripple Carry Array Multiplier (RCAM) and Wallace Tree Multiplier (WTM) for energy-delay performance. The resulting implementation models did not provide as much value in energy-delay as the originally reported crude architectural analysis predicted, especially when the multiplier size is smaller than 32x32. This is due to the fact that threshold detection required by ABACUS “column compression” is not trivial to implement at low cost using standard logic approaches. On the other hand, the proposed logic implementation of ABACUS in this thesis is scalable to any m x n integer multiplier, and demonstrates close to 2x energy-delay product improvement potential compared to scalable RCAM and CSAM logic implementations for 64x64 bits multiplication, and more for larger multipliers.

ÖZ

ABACUS ÇARPICININ MİMARİ AÇIDAN DİĞER ÇARPICILARLA ENERJİ-GECİKME KARŞILAŞTIRMASI

Gürdür, Didem

Fen Bilimleri Yüksek Lisansı., Sürdürülebilir Çevre ve Enerji Sistemleri,

Tez Yöneticisi : Yrd. Doç. Dr. Ali Muhtaroğlu

Temmuz 2013, 82 sayfa

Bu tez yakın zamanda önerilmiş ABAKUS tamsayı çarpım mimarisinin mantıksal uygulamasını sunar. ABAKUS m x n uygulaması, Elde Öngörülü Dizi Çarpıcı (CSAM), Dalgalı Elde Dizi Çarpıcı (RCAM) ve Wallace Ağaç Çarpıcı (WTM) ile araştırma kapsamında geliştirilmiş Tahmini Güç Hesaplama Aracı (PETAM) kullanılarak modellenmiş, doğruluğu ve enerji-gecikme performansı değerlendirilmiştir. Sonuç olarak elde edilen uygulama modeli orjinal yayında kullanılan ham mimariden beklenilen enerji-gecikme performansını özellikle 32x32 uzunluğunun altındaki çarpma işleminde gösterememiştir. Kolon sıkıştırma için kullanılan eşik tesbit etme işleminin standart mantıksal yaklaşımlarla yaratılamaması, bu sonuca başlıca sebep olarak gösterilebilir. Diğer taraftan bu tezde sunulan m x n boyutlarında ölçeklenebilir ABAKUS mantıksal uygulaması, 64x64 ve daha büyük çarpma işlemlerinde CSAM ile RCAM uygulamalarının enerji-gecikme performansinda 2 kat ilerleme göstermektedir.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my mother Zalihe Yazgın; without her love and support I would not be able to accomplish this work.

I show my sincerest thanks to my supervisor Ali Muhtaroğlu, for his guidance, support and valuable contributions throughout my work.

I would also like to thank Tegiye Birey for her understanding, constructive comments and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

CLA: Carry Look-ahead Adder

CMOS: Complementary Metal Oxide Semiconductors

CSLA: Carry Select Adder

CSMA: Conditional Sum Adder

CSAM: Carry Save Array Multiplier

DSP: Digital Signal Processing

FA: Full Adder

FFT: Fast Fourier Transform

HA: Half Adder

NMOS: N-type Metal Oxide Semiconductor

PETAM: Power Estimation Tool for Array Multipliers

PC: Parallel Counter

PP: Partial Product

PMOS: P-type Metal Oxide Semiconductor

RCA: Ripple Carry Adder

RCAM: Ripple Carry Array Multiplier

VHDL: Very High Speed Integrated Circuit Hardware Description Language

WTM: Wallace Tree Multiplier

CHAPTER 1

INTRODUCTION

Technological improvements like CMOS that replace bipolar devices have succeeded in reducing power dissipation. However, these improvements had the primary goals of high integration and increased speed. In recent years, low energy computation has been given equal attention as high performance in architectural design of the embedded systems. Many factors have contributed to this trend. One of the most prominent driving factors has been the increasing demand and growth of embedded systems. Laptop computers, personal digital assistants, remote sensors, mobile phones, and hearing aids have enjoyed significant success among consumers, and the market for these devices is expected to increase in the future [1].

Power dissipation is one of the most critical parameters amongst the modern embedded systems. In the implementation of system architecture, low power design is essential to ensure the sustainability of feature scaling with Moore's law, and to produce consumer electronics with more battery life and less weight. In some applications, embedded systems operate without batteries by scavenging energy from the environment. Some applications still work with batteries, for which battery life is determined by average power consumption. To extend battery life or to enable self-powered systems, it is necessary to use low power operations in these integrated circuits. Furthermore, managing maximum power dissipation is now a key factor in integrated circuit packaging and cooling. Due to large amount of power dissipation of high speed circuits, vast amount of heat generation occurs as a by-product. Consequently, the price, size, weight, battery life, and reliability of these embedded systems are all strongly dependent on power dissipation [2].

Integer multiplication is a vital arithmetic operation for common Digital Signal Processing (DSP) that is used in embedded systems, namely filtering and Fast Fourier Transform (FFT). To achieve high execution speed, parallel array multipliers are widely used but these multipliers can consume more power than other multiplier topologies. Power consumption has become a critical concern in today's

VLSI system design. Hence, power efficient multipliers are needed for the design of low-power DSP systems [3]. Due to signal transitions, in many processors, most of the power dissipation is dynamic power dissipation. However, as transistors become smaller and faster, leakage power known as static power dissipation becomes significant. It is required to reduce both dynamic and static power dissipation to save significant power consumption of embedded systems.

This work presents architectural energy delay assessment of the recently proposed ABACUS integer multiplier architecture with respect to other array multipliers. The ABACUS m x n implementation was modeled and evaluated using the PETAM tool [4], against Carry Save Array Multiplier (CSAM), Ripple Carry Array Multiplier (RCAM) and Wallace Tree Multiplier (WTM) for energy-delay performance. This chapter will discuss the objective of this research and provide information about background research.

1.1 *Objective*

The objective of this research is an attempt to compare ABACUS multiplier architecture with CSAM, RCAM and WTM architectures and to find the place of ABACUS multiplier on the trade-off line. The approach recognizes that power consumption in digital circuits has strong links with activity and physical capacitance. A logic implementation method for the recently proposed ABACUS multiplier architecture [3] is developed in this thesis research, and evaluated for speed, power, and energy-delay product. Specifically, the challenging operation of column threshold detection in the high level architecture described by the author is addressed through variable size parallel counters in this work. The implementation minimizes switching activity for dynamic power reduction. This logic-based study tracks, but does not effectively tackle the multiplier static power dissipation. However, there are opportunities for static power optimization, including partitioning design for supply voltage domains, power gating, using low leakage transistors in circuit design [5], etc. Details about ABACUS multiplier will follow in Chapter 4.

Power estimation is an important issue in the early determination of energy-delay performance in embedded systems. High-level power estimation methods have not yet achieved the maturity necessary to enable rapid evaluation of logic designs [5]. This study, therefore, uses a Power

Estimation Tool for Array Multipliers (PETAM) [4] to calculate the comparative performance of ABACUS against other common implementations.

1.2 Background Research

Digital multipliers are major sources of power dissipation in digital systems. Furthermore, array architecture is a popular typology. An array multiplier is composed of rows of full or half adders for recursive shift and addition operations. As Figure 1 shows, there are two signals, sum and carry, which are generated in the previous rows. These signals are transferred to the next rows as extra inputs. By turning off power to the array elements or by padding the shifted operands with leading zeros, it is possible to reduce switching activity in the unused array elements. Having bypass lines in the partial product array that bypasses nonessential array elements and by feeding partial sum and carry directly to the final carry propagate adder may reduce power. Additional power reduction may be achieved by bypassing the elements of the carry propagate adder.

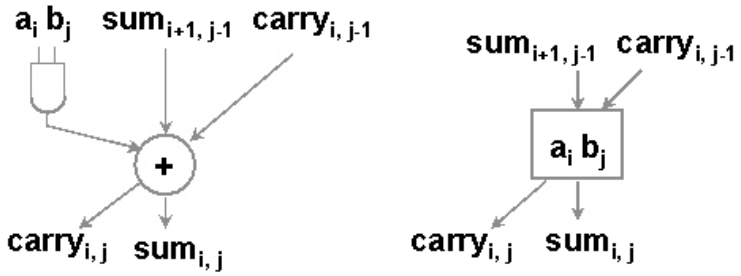


Figure 1. Basic array multiplier.

In 1950, Booth proposed an algorithm to increase the performance of multipliers. Basically the Booth multiplier follows the usual method of summing partial products but it also uses a trick that can reduce the number of operations for multiplication on the average compared to shift add multipliers. Algorithm examines each pair of digits in the multiplier, creates the first pair by appending a dummy 0 at the least significant end. Then,

i. If the pair is 01, add the multiplicand

ii. If the pair is 10, subtract the multiplicand

iii. Otherwise, do nothing

It shifts both the partial product and multiplier to right and examines the next pair of digits. This algorithm repeats as many times as there are digits in multiplier. Booth multipliers are also called radix-2 multiplier. Their main advantage is that they involve no correction cycles for signed terms. One downside of these multipliers is that they become inefficient for alternate zeros and ones as they involve large numbers of adders and subtractors [6].

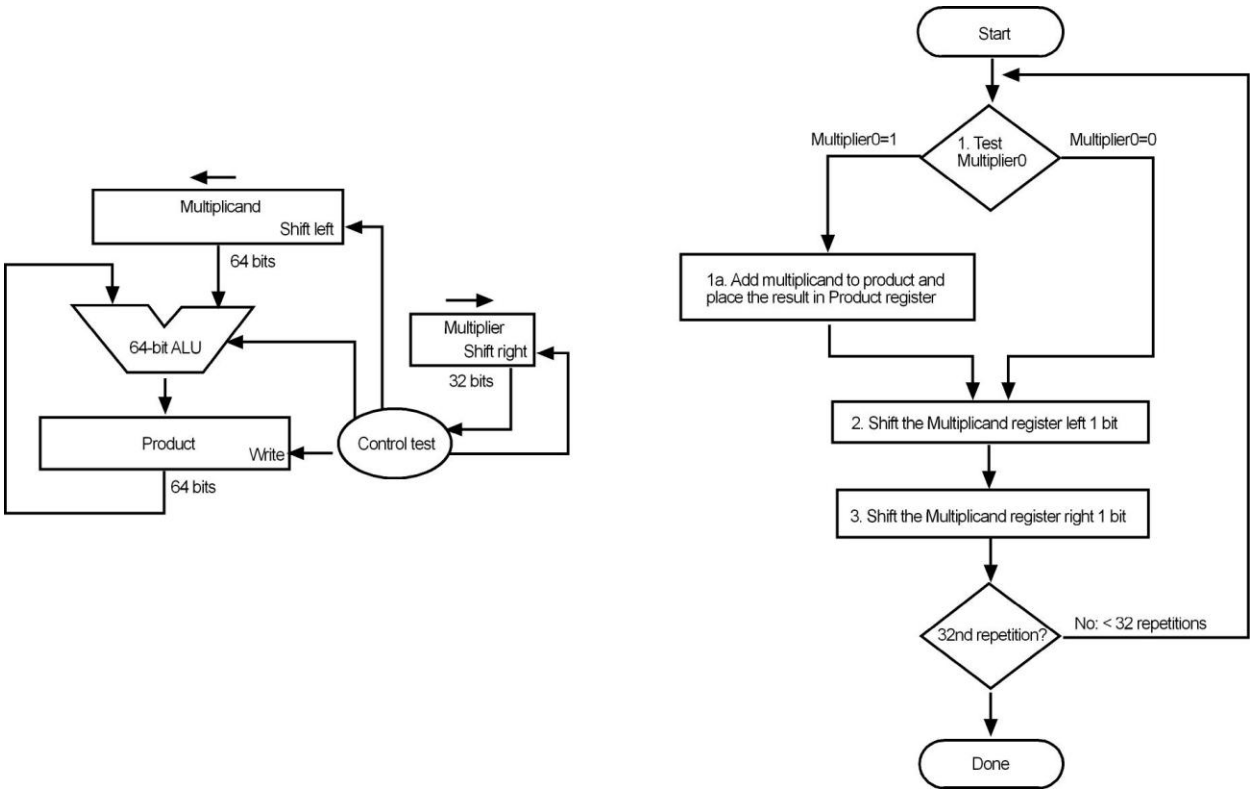


Figure 2. Booth multiplier design and algorithm.

Wallace proposes a tree multiplier architecture, which performs high speed multiplication. This efficient hardware implementation uses full adders and half adders to reduce the partial products matrix. The disadvantage of this multiplier is that it has a high structural irregularity and is not well suited for VLSI implementations which often demand regularity [7]. Section 2.3 will give more information about the implementation of Wallace Tree multiplier and its abstract comparison with proposed ABACUS multiplier.

Dadda presents a similar approach to Wallace. The approach reduces the number of half adders and at smaller scale full adders used in the tree at the expense of a larger adder at the end. Dada tree is optimum in providing the minimum number of full adders. Dadda's work shows that the approach can also work for larger adders. It is also more flexible to manipulate different types of counters and used as the basis of comparisons by a variety of research projects targeting to improve the performance of multiplier [9].

The power comparison between arrays and trees has been examined thoroughly and it is stated that the significant amount of spurious transitions in array multiplier results in higher power dissipation than Wallace tree multiplier. Arrays have almost doubled the amount of transitions for an n-bit multiplier. The high numbers of intermediate, unwanted glitches in array multipliers draw power for charging and discharging intrinsic capacitances for a significant number of times before the circuit settles [10].

Baugh-Wooley multiplier is also an array multiplier, but can perform both unsigned and signed multiplication. Partial products are adjusted in a way that negative sign moves to last step, which in turn maximizes the regularity of the multiplication array. Baugh-Wooley multiplier operates on signed operands with 2's complement representation to make sure that the signs of all partial products are positive. But, it is not suitable for large size operands [11].

The essential principle used for multiplication is to evaluate partial products and add-shifted partial products. To perform this operation, a number of consecutive addition operations are required. Hence, adder is one of the major components required to design an array multiplier. There are adder architectures such as Ripple Carry, Carry Look Ahead, Carry Select, Carry Skip and Carry Save [11, 12, 13]. To analyze the performances of different fast adders, a lot of research work has been done. In 1996, Nagendra et al. performed a study on different parallel and synchronous adders based on their power, delay and area characteristics [14]. This study concentrates on different parallel adders like ripple carry adder, Manchester carry chain adder, carry select adder, carry look-ahead adder, and carry save adder, comparing several types of them and concluding that the ripple carry adder provides reasonable speed for small precisions only.

In 2004, Sertbas and Özbey analyzed the performance for classified binary adder architectures on the basis of VHDL simulations [15]. The results of the study show that CSAM gives better performance in terms of speed and power consumption when compared to the RCAM. Fonseca et al. [16] presented a design of a Radix 2m hybrid array multiplier to handle operands in 2's-complement form by using CSA on partial product lines in 2005. The results also indicate that the multiplier architecture with CSA gives better performance in terms of speed, area and power consumption in comparison with the architecture of RCA [16]. In addition to this, in 2008, Krad and Al-Taie worked on performance analysis of a 32-bit unsigned data multiplier with CLA logic against one with RCA logic, using VHDL. The analysis was done on the basis of speed only, and showed that multiplier architecture using CLA gives better results than does RCA [17]. These studies were mostly based on performance in terms of delay. However, they provide this study with significant base as they have been used to validate the simulation results from the PETAM tool. EDP (of CSAM and RCAM) was calculated by PETAM tool and then compared with Sertbas and Özbey's results. Section 2.4 has more information about the verification of PETAM tool.

Muhtaroğlu [3] proposed crude architectural analysis of ABACUS multiplier, where the energy-delay advantages were analyzed architecturally over the CSAM. The architecture in this study required a special threshold detection block for column compression which was not available for the purposes of analysis in this particular research, which is at a more detailed (lower) abstraction level than what Muhtaroğlu assumed. Chapter 4 includes more details on our implementation of the new ABACUS multiplier architecture.

CHAPTER 2

EVALUATION METHODS & TOOLS

This chapter is dedicated to the evaluation methods, tool descriptions and metrics that are used in the study. In the first section, mathematical proof of the upper limit is described and is followed by an explanation of metrics. Details about the power estimation tool and the verification of the tool are investigated in last two sections.

2.1 Big O Notation

The notations that are used for describing the asymptotic running time of an algorithm are called asymptotic notations. The result of the analysis of an algorithm is usually an equation that gives the amount of time, number of memory access, number of comparisons or the other metric. In this particular case, these formulas are functions of hardware size and delay. This formula is the way of expressing the cost of CSAM, RCAM, WTM and ABACUS multiplier.

One way to express the cost is the Big O notation. The big O notation is used to describe the limiting behavior of a function in mathematics. The argument tendency is headed for a particular value or infinity, usually in terms of a simpler function. The big O notation is a member of Landau notation, which was proposed by Landau and Bachmann. It is an asymptotic notation [18], which expresses the upper bound of an algorithm.. This study uses big O notation to compare different multiplier architectures for the upper limit of the hardware size and the worst case delay.

2.2 Metrics

2.2.1. Static Power Consumption

Static power consumption describes the minimum power used even when there is no activity in the circuit. It is expected that the CMOS components do not have any static power consumption due to no direct paths from the power supply to the ground. Unfortunately, metal oxide semiconductor

transistors in these systems are not perfect switches and in many applications, static power consumption is not zero. The gate of the transistor is closed to the threshold voltage, and some currents still flow through it; this is why there will always be leakage currents in MOS transistors [19]. This current can be negligible. However as transistors get smaller; their effect on power becomes critical. In this study, the static power consumption is taken into account in calculations to get closer EDP estimations.

2.2.2. Dynamic Power Consumption

The dynamic power consumption in CMOS has two sources,

$$P_{dynamic} = P_{short-circuit} + P_{switching}$$

When both P-type metal-oxide-semiconductor (PMOS) and N-type metal-oxide-semiconductor (NMOS) transistors are open at the same time, short circuit happens. Switching delay occurs due to PMOS and NMOS transistor switching. This makes a short circuit between Vcc and ground. This research does not concentrate on short circuit power consumption since it can be adjusted by properly balancing the inputs of the NMOS and PMOS transistors [20].

The other dynamic power source, switching power is used to charge parasitic capacitance in lines between the CMOS cells [19]. When the output of a gate changes from 0 to 1 or 1 to 0 the energy transition occurs, and this is equal to:

$$E_{switching} = \frac{1}{2} cAV^2$$

where V is the power source, A is the activity factor and c is the capacitance per gate.

Since an equal amount of energy is used to charge the circuit for each 0 to 1 or 1 to 0 transitions, it is possible to get an equation for power used in switching. Considering the frequency f of the circuit, and the probability for a 0 to 1 or 1 to 0 switches at the gate, we get the equation:

$$P_{switching} = fcAV^2$$

There are three elements to improve power usage, voltage, physical capacitance and activity.

Although the other sources of power dissipation have increased their share, switching power consumption is still the largest power dissipation source in CMOS today [19]. Therefore, there is a vast body of research for the optimization of switching power. Wei and Horowitz conducted a study about digital power supply controller for variable voltage and frequency circuits. This study shows that the controller has advantages on power efficiency [21].

Power estimation tools are quicker than low level simulation. Moreover, architecturally scalable estimation allows predictions on large systems, which may be very time consuming to design and simulate at circuit level.

The formulas to calculate dynamic energy (E_{dyn}), static energy (E_{stat}) are:

$$E_{dyn} = \frac{1}{2} * c * \#\_\text{of\_gates} * \text{activity\_factor} * V^2 \quad (1)$$

$$E_{stat} = I_{leakage} * \text{V} * [\#\_\text{of\_gates} * (1 - \text{activity\_factor})] * \text{worstcase\_delay} \quad (2)$$

$$E_{total} = E_{dyn} + E_{stat} \quad (3)$$

where c is dynamic gate capacitance, activity_factor is defined in this model as the probability of a gate experiencing an output low-to-high or high to low transition, $I_{leakage}$ is leakage current and V is supply voltage.

This study uses Energy Delay Product (EDP) as a metric to compare each type of multiplier for their ability to give equal importance to energy consumption and delay. Proposed software is designed in a way to accept user defined variables like C, $I_{leakage}$, Vcc to calculate power consumption and EDP of different technologies.

2.3 Tools

In this work, the PETAM (Power Estimation Tool for Array Multipliers) tool [4] was developed for relative comparison, and not for absolute prediction of power and performance.

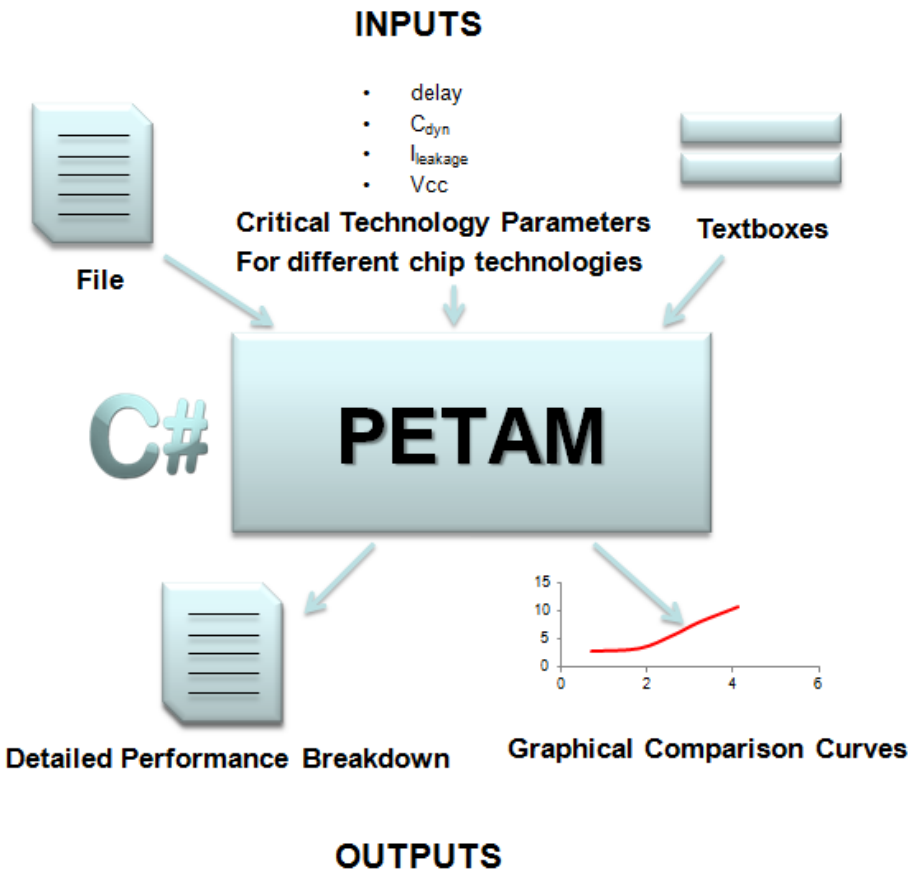


Figure 3. Inputs and outputs of PETAM.

Physical implementation, especially of large systems, is very time consuming and is not scalable. Spreadsheets, which provide means to calculate power dissipation, can be slow when parameterized, and were not preferred for this particular case. It is preferred to use C# environment due to the author's experience on coding. Industrial CAD tools may offer more accuracy, and are widely used in the implementation phase. However, they have some significant limitations: They can in general only be applied to some of the available device technologies, are very hard to configure to quickly report energy-delay performance across a variety of activity assumptions, and are very expensive.

PETAM [4] is developed with C# .Net Framework to calculate EDP of RCAM, CSAM and ABACUS multipliers. It was later enhanced to support WTM as well, as will be elaborated in later sections. The tool simulates the architecture of all multiplier topologies implemented using Full-Adders as building blocks, and calculates multiplication result, delay, power, activity factor and EDP. Figure 3 and Figure 4 show inputs and outputs, and screenshot of the tool respectively.

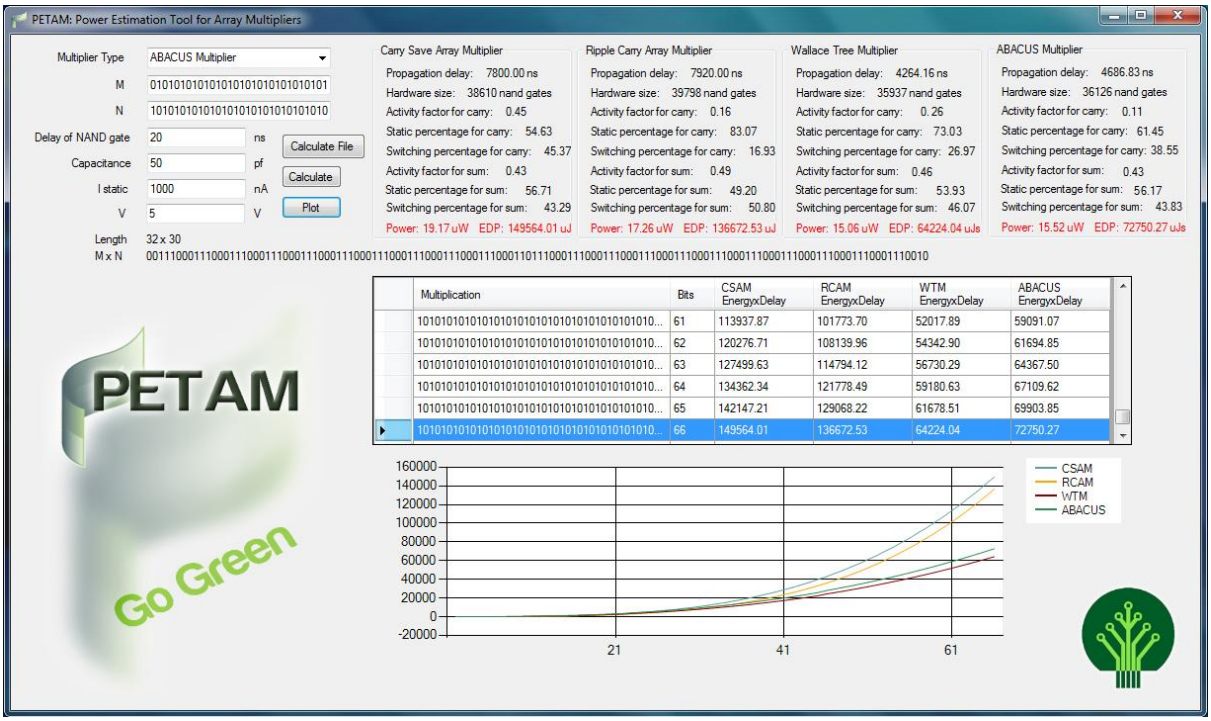


Figure 4. Result screen of PETAM.

PETAM could make single multiplication calculation using data entered in text boxes, or read an input text file to process a large set of multiplications in batch mode. PETAM first reads the bit length of multiplication; it checks the multiplier and multiplicand size and finds out the bit length of them by cancelling any zeros at the higher bits. Then PETAM creates necessary number of full adders, and connects inputs and outputs properly as algorithm dictates. The tool simulates the propagation of bits from inputs to outputs for each multiplication, and calculates activity factor at each internal node.

PETAM works in a scalable manner, and there is no limitation on the input or output size. Delay, capacitance, leakage current and supply voltage are input variables, not constants, because integrated circuit (IC) technologies vary, and PETAM has the ability to calculate results for different IC technologies. PETAM generates a table that shows the multiplication results, bit sizes, and EDPs for all multiplier topologies. The tool also has the ability to produce plots of EDP against multiplier size to compare multipliers easily across a variety of sizes [4].

Four different types of input vectors have been used during research: Checkerboard pattern, all 1's pattern, 2nd type checkerboard pattern and random pattern. Checkerboard pattern is repetition of "01" or "10", all 1's pattern is repetition of 1's, 2nd type checkerboard pattern is repetition of "0011"or "1100", and random is randomly generated vectors by the tool. Appendix D contains these input files. Different types of multiplier and multiplicand patterns were utilized to investigate the impact on EDP for each multiplier architecture.

2.4 Verification

All multiplier architectures were modeled in ALTERA Quartus II 9.0 for the basic verification of the implemented logic in PETAM with the help of waveform files. The design verification methodology consist of 3 steps; design creation, correctness verification, delay verification. Verification of the power consumption was not possible with available software resources.

Appendix A contains the design files of multiplier architectures. These files were used to simulate the multipliers to acquire the delay information through the waveform files. Even though the delay results were not the same as PETAM due to differing circuit technology assumptions, the relative trends across different architectures were the same. The relative delay performance of CSAM and RCAM as predicted by PETAM simulations and ALTERA Quartus II simulations were consistent when compared with the results from the study of Sertbas and Özbey [15].

The vector dependent power and delay parameters associated with the NAND gate can easily be characterized in the lab environment, as the building block for all multipliers. To establish this characterization method, this experiment was done with selected HD74HC00 IC components, and

gate level delay, capacitance, and current were measured. Appendix C contains the measurement details of that experiment.

CHAPTER 3

MULTIPLIER IMPLEMENTATIONS

This chapter provides an overview of the array and tree multipliers. The implementation of selected multiplier topologies is described, and each multiplier's architecture is studied. In addition, the algorithm used to model each multiplier topology in a scaleable manner in PETAM tool is discussed.

Binary multiplication operation can be realised using repeated binary addition, once partial products are generated. If we define two binary numbers called X and Y that are m and n bits wide, their binary representation with X_i, Y_j Є {0.1}|:

$$X = \sum_{i=0}^{M-1} X_i 2^i$$

$$Y = \sum_{j=0}^{N-1} Y_j 2^j$$

The multiplication of X and Y is called Z and is defined as:

$$Z = X * Y = (\sum_{i=0}^{M-1} X_i 2^i) * (\sum_{j=0}^{N-1} Y_j 2^j)$$

Multiplication operation has two main stages; one is partial product generation, and the other one is addition of those partial products. Figure 5 shows the partial products and their addition. Each dot represents one partial product. Chapter 4 has a more detailed explanation about the implementation of partial product generation.

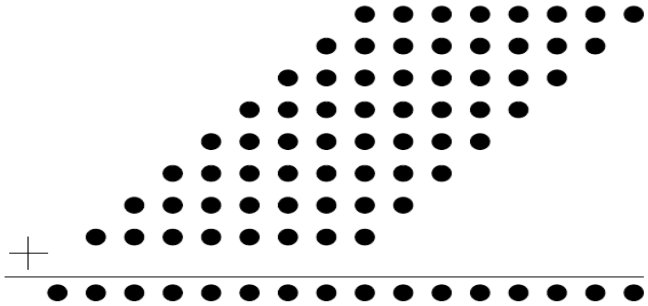


Figure 5. Addition of partial products.

The multiplication topologies differ from each other because of the way partial products are added. Kumar and Saravanan [22] suggested using full adders as building blocks to compare various types of adders in 2012. Their study shows that the use of full adders instead of half adders reduces the critical path, and the usage of energy efficient adder designs increases the performance [22]. Thus full adder is used as the building block of all selected multiplier implementations in this study. There are different implementation methods for full adders. Shanthala and Kulkarni [23] compared different ways of realizing the full adder in their research. Their research shows that only NAND gate implementation of full adder has less delay and lower power when compared with XOR, AND, OR implementations. Figure 6 shows the implementation of NAND based full adder that has been extensively used in this research.

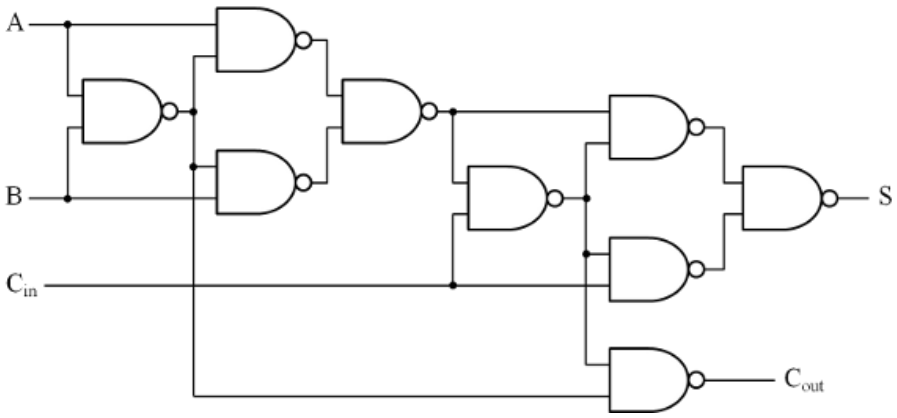


Figure 6. Full Adder design with NAND gates.

An array multiplier is composed of rows of full or half adders for recursive shift and addition operations. There are two signals, sum and carry, which are generated in the previous rows. These signals are transferred to the next rows as extra inputs. Array multiplier has the advantage of a regular structure. This reduces the design time of an array multiplier when compared with a tree multiplier. Another advantage of the array multiplier is its ease of design for a pipelined architecture. On the other hand, the main disadvantage of the array multiplier is the worst case delay. The speed will be low for a very wide multiplier. Adder is one of the major components required to design an array multiplier. Many researchers carried out analyses of array multipliers based on their power, delay and area characteristics. Next two sections will give details about two selected array multiplier topologies; carry save array multiplier and ripple carry array multiplier.

3.1 Carry Save Array Multiplier

CSAM is one of the simplest of all multipliers. The basic idea behind it is doing paper and pencil style multiplication, which means addition of partial products. CSAM is usually designed as a square-shaped array to minimize capacitance and sustain regularity for ease of design. Figure 7 shows the CSAM architecture modeled in PETAM, in 4x4 size as an example. The modeling algorithm is based on the principle that carry outputs are sent diagonally to the next stage of addition. The last stage has a vector merging adder to merge the carry and sum outputs. As it is seen from the figure, the carry output of a full adder is fed back to the carry input of the neighbor full adder. Almost all fast multipliers make use of a Carry Save Adder technique [7, 8, 9].

An n x m bit CSA multiplier has n x m AND gates, m half adders, and ((n-1).(m-1))-1=n.m-n-m full adders. Depending on the design of the full adders, there could be 22-32 transistors inside.

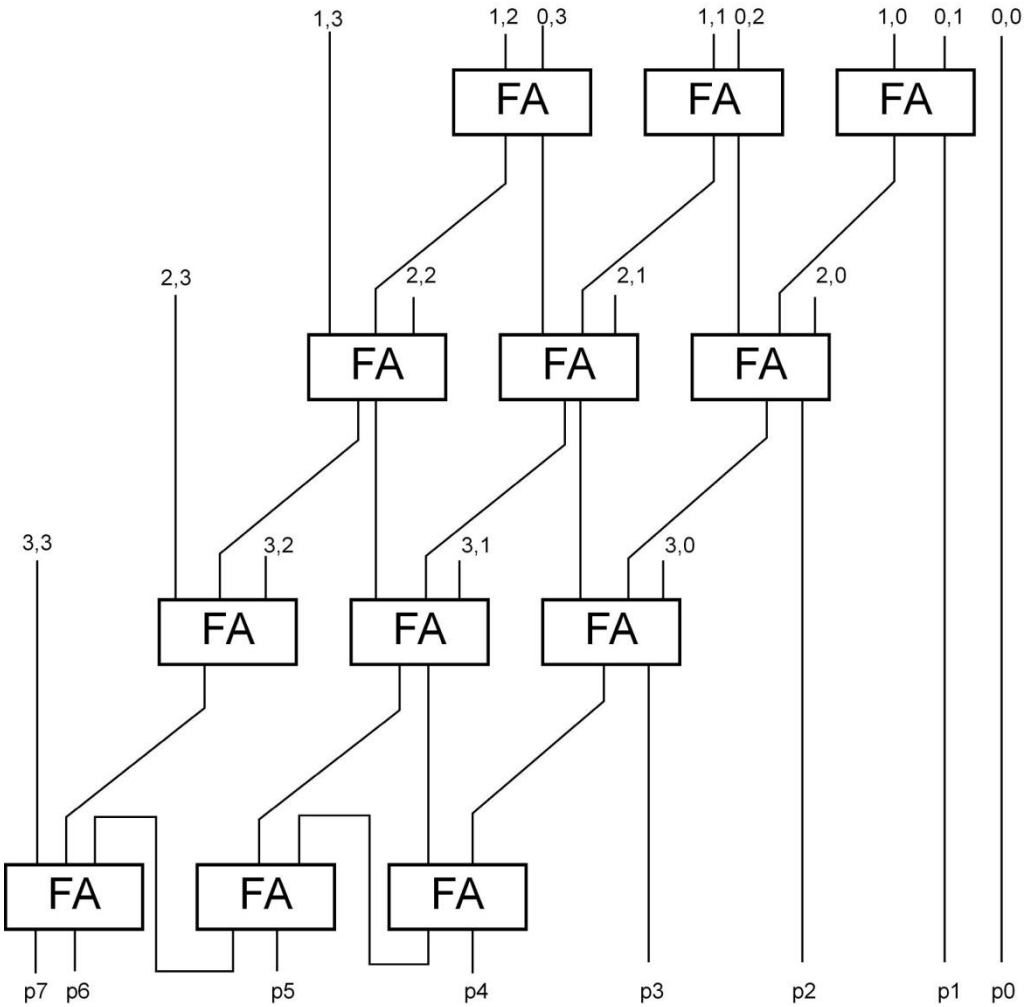


Figure 7. 4-bit Carry Save Array multiplier.

CSAM has hardware size $O(n^2)$ and delay O(n) as an order of growth property where n is the size of each of multiplicand and multiplier. The detailed description of the order of growth can be found in in Chapter 2.

3.1.1 Pseudocode to Simulate CSAM

This is the pseudocode to simulate the signal propagation in the PETAM model for the CSAM.

Initialize; *;Initialization Section.*

m=get_multiplier_size *;read multiplier from textbox and save size of it to m*.

n=get_multiplicand_size *;read multiplicand from textbox and save size of it to n.*

c= get_capacitance *;read capacitance of a NAND gate from textbox and save it to c.*

v= get_voltage ;*read voltage of a NAND gate from textbox and save it to v.*

I= get_leakage_current ;*read leakage of a NAND gate current from textbox and save it to I.*

d= get_delay; ;*read delay of a NAND gate from textbox and save it to d.*

Partial product generation ;*Partial Product Generation Section.*

for each bit of multiplier

for each bit of multiplicand

PP=m AND n ;*create partial products array with using AND gates.*

end

end

for each full adder

create_FA_inputs_A=PP[..] ;*connect first input of full adder with suitable partial product.*

create_FA_inputs_B=PP[..] ;*connect second input of full adder with suitable partial product.*

create_FA_inputs_Cin=PP[..] ;*connect third input of full adder with suitable partial product.*

calculate_Cout ;*Cout=A*Cin+B*Cin+A*B*

calculate_S ;*S=Cin XOR A XOR B*

compare Cout with earlier Cout ;*compare signals with earlier ones and calculate activity factor*

compare Sum with earlier Sum ;*if only sum or Cout changed switching=0.5,*

calculate_switching ;*if both changed switching=1, if none changed switching=0*

;*switching activity calculated with the comparison of each full adders earlier sum and Cout with current ones. If only one of them changes then the activity factor is 0.5, if both changes it is 1 and if there is no change on both of them then the activity factor is 0.*

end

product=selected_Couts_Sums ;*final product is equal to Cout or Sum of some full adders.*

Activity_Factor=total_switching /m*n ;*calculate activity factor of multiplication.*

Hardware_Size=n*m*9 ;*total number of gates, each FA has 9 gates*

Delay=(m+n-1)*3*d; ;*worst case delay for each gate*

Edyn=0.5*c* Hardware_Size*Activity_Factor*v^2 ;*calculate dynamic energy*

Estat=I*V* Hardware_Size*(1- Activity_Factor)*Delay *;calculate static energy*

E=Edyn+Estat *;calculate total energy*

EDP=E*Delay *;calculate EDP*

The following formulas are used to calculate upper bound of hardware size and worst case delay for n x n multiplication.

Hardware Size = n*(n-1) *Full Adders*

= n*(n-1)*9 *NAND gates*

Worst Case Delay = 2*(n-1)* $t_{delay\_of\_one\_FA}$ *Full Adders*

= 2*(n-1)*3* $t_{delay\_of\_one\_NAND}$ *NAND gates*

CSAM complexity is $O(n^2)$ for hardware size and O(n) for worst case delay.

3.2 Ripple Carry Array Multiplier

RCA multiplier accepts an n x m bit multiplication and uses an array of cells to calculate the bit products. As Figure 8 shows for 4x4 case, after the parallel calculation of bit products, architecture adds them together in a proper way to yield the final product. Meier et. al.'s [24] study shows that RCAs are always smaller than other array multipliers due to emphasis on local wiring. RCA multipliers are thus the slowest, but use the least energy compared to other multipliers. This multiplier implementation forces each full adder to wait until the previous carry output is calculated in order to start calculation of its carry and sum outputs. Since the carry has to propagate through every row in the column, the critical path is very long [24].

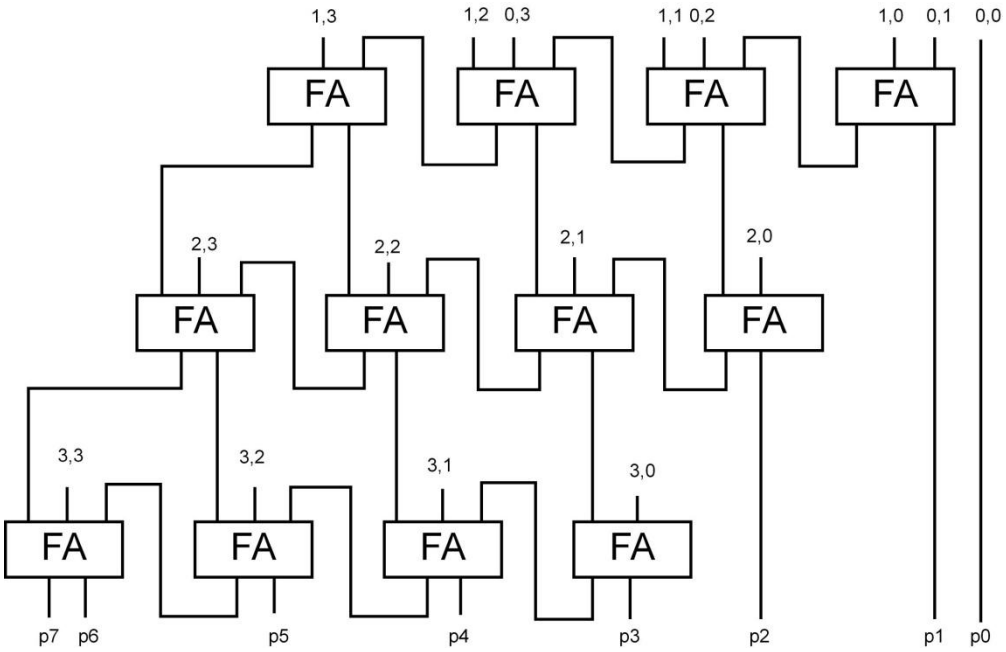


Figure 8. 4-bit Ripple Carry Array multiplier.

As described in Chapter 1, ripple adder is the slowest but also lowest power adder implementation. A vast body of research has been done about adder designs and most of them go beyond RCA. These architectures focused on cutting down the carry propagation delay. Nevertheless, while the speed of the adder is increased, the area of the implementation often increased undesirably.

3.2.1 Pseudocode to Simulate RCAM

The following is the pseudocode to simulate the signal propagation in the PETAM model of the RCAM:

Initialize;	*;Initialization Section.*
m=get_multiplier_size	*;read multiplier from textbox and save size of it to m*.
n=get_multiplicand_size	*;read multiplicand from textbox and save size of it to n.*
c= get_capacitance	*;read capacitance of a NAND gate from textbox and save it to c.*
v= get_voltage	*;read voltage of a NAND gate from textbox and save it to v.*
I= get_leakage_current	*;read leakage of a NAND gate current from textbox and save it to I.*
d= get_delay;	*;read delay of a NAND gate from textbox and save it to d.*

Partial product generation *;Partial Product Generation Section.*

for each bits of multiplier

for each bits of multiplicand

PP=m AND n *;create partial products array with using AND gates.*

end

end

for each full adder

create_FA_inputs_A=PP[..] *;connect first input of full adder with suitable partial product.*

create_FA_inputs_B=PP[..] *;connect second input of full adder with suitable partial product.*

create_FA_inputs_Cin=PP[..] *;connect third input of full adder with suitable partial product.*

calculate_Cout *;Cout=A*Cin+B*Cin+A*B*

calculate_S *;S=Cin XOR A XOR B*

compare Cout with earlier Cout *;compare signals with earlier ones and calculate activity factor*

compare Sum with earlier Sum *;if only sum or Cout changed switching=0.5,*

calculate_switching *;if both changed switching=1, if none changed switching=0*

;switching activity calculated with the comparison of each full adders earlier sum and Cout with current ones. If only one of them changes then the activity factor is 0.5, if both changes it is 1 and if there is no change on both of them then the activity factor is 0.

end

product=selected_ Couts_Sums *;final product is equal to Cout or Sum of some full adders.*

Activity_Factor=total_switching /m*n *;calculate activity factor of multiplication.*

Hardware_Size=n*m*9 *;total number of gates, each FA has 9 gates*

Delay=(m+n)*3*d; *;worst case delay for each gate*

Edyn=0.5*c* Hardware_Size*Activity_Factor*v^2 ;*calculate dynamic energy*

Estat=I*V* Hardware_Size*(1- Activity_Factor)*Delay *;calculate static energy*

E=Edyn+Estat *;calculate total energy*

EDP=E*Delay *;calculate EDP*

Formulas below are used to calculate the upper bound of hardware size and worst case delay for n x n multiplication.

Hardware Size = n*(n+1) *Full Adders*

= n*(n+1)*9 *NAND gates*

Worst Case Delay = $2*n*t_{delay\_of\_one\_FA}$ *Full Adders*

= $2*n*3*t_{delay\_of\_one\_NAND}$ *NAND gate*

RCAM complexity is $O(n^2)$ for hardware size and O(n) for worst case delay.

3.3 Wallace Tree Multiplier

The most time consuming operation in multiplication is the carry propagation. In 1964 Wallace proposed a method to avoid using carry propagate addition. This method uses an adder tree, which is constructed by full adders and half adders. In this technique it is possible to decrease any numbers of partial products to two numbers without carry propagate addition. These two numbers can be added in the last stage [10]. It is shown in Figure 9 how full adders and half adders reduce the bits. The carry output is shifted to the left by one bit so its weight is doubled.

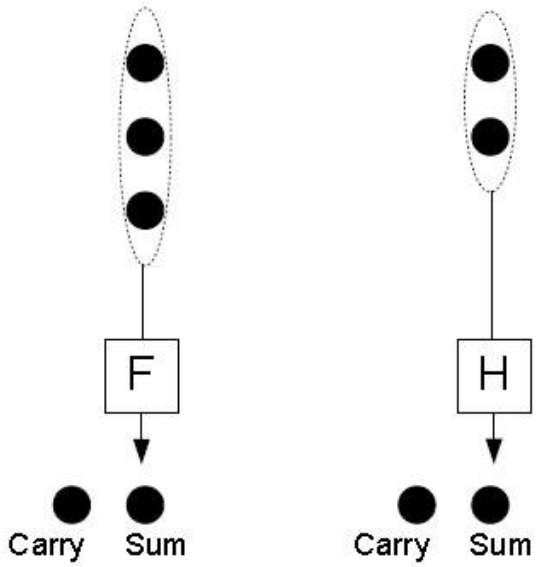


Figure 9. Full adder and half adder bit reduction.

Wallace tree creates a structure for parallel addition, which removes the need to wait for all the earlier stages to complete, and therefore, has less delay. It parallelizes the carry save operations and makes the delay time shorter than the array's sequential series of operations.

Addition of partial products occurs as follows [10]:

1) Split partial products into groups of three and input each group into individual sets of (3,2) counters.

2) Split the resulting bits from step (1) into groups of three and input each group into sets of (3,2) counters or FAs.

3) Repeat by combining into groups of three, and adding with sets of (3,2) counters until two numbers remain.

4) Add the final two numbers using a carry propagation adder to get the final product.

The difference between carry save procedure and proposed procedure is that the carry save procedure takes three inputs and reduces the number of bit vectors to two at each stage but parallel methods takes sets of 3 vectors and reduce them to sets of 2 vectors. Hence the delay of the parallel method will be O(log3/2 n), whereas the delay of sequential carry cave procedure is O(n) [10]. This parallel method is called WTM and it reduces the delay of partial products addition stage substantially. The downside of WTMs is their irregular layout, which results in potentially greater wire loads.

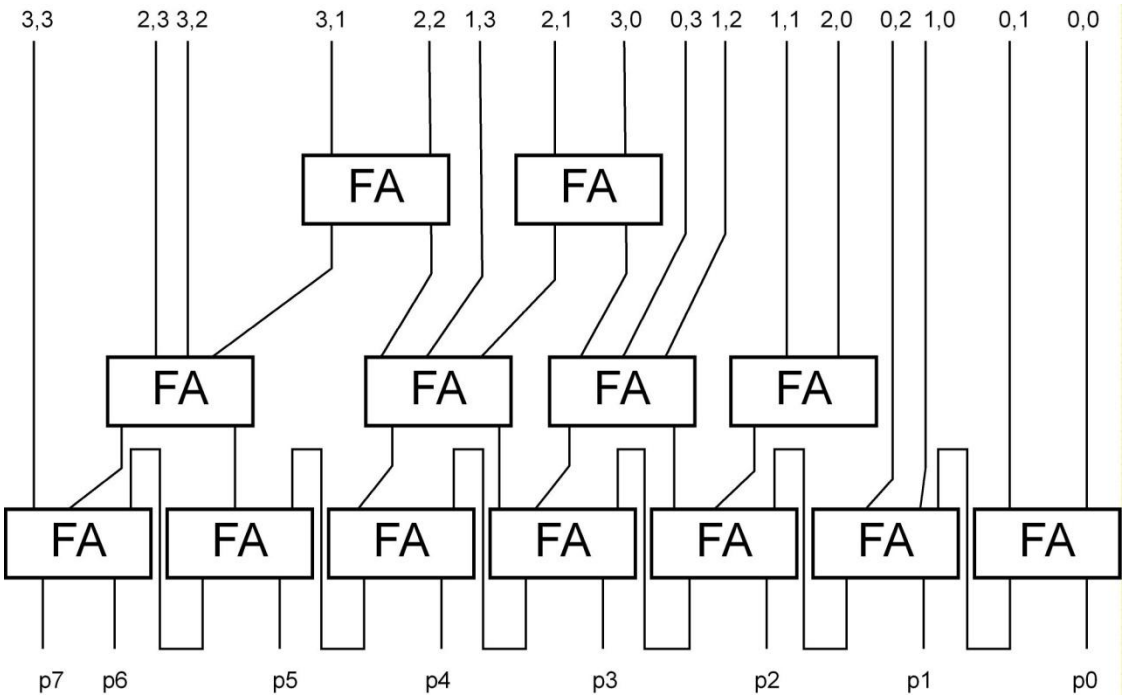


Figure 10. 4-bit Wallace Tree multiplier [35].

3.3.1 Pseudocode to Simulate WTM

The following is the pseudocode to simulate the signal propagation in the PETAM model of the corresponding RCAM.

Initialize; *;Initialization Section.*

m=get_multiplier_size *;read multiplier from textbox and save size of it to m*.

n=get_multiplicand_size *;read multiplicand from textbox and save size of it to n.*

c= get_capacitance *;read capacitance of a NAND gate from textbox and save it to c.*

v= get_voltage *;read voltage of a NAND gate from textbox and save it to v.*

I= get_leakage_current *;read leakage of a NAND gate current from textbox and save it to I.*

d= get_delay; *;read delay of a NAND gate from textbox and save it to d.*

Partial product generation *;Partial Product Generation Section.*

for each bits of multiplier

for each bits of multiplicand

PP=m AND n *;create partial products array with using AND gates.*

end

end

stage_number= log(m/2)/log(4/3) *;calculate number of stages needed to complete multiplication*

Form (3,2) compressors with full adders;

for y=1 to y=stage_number

while (maximum_column_size>2)

if column_size==1 *;the first column where there is only one element*

propogate_to_next_stage

if column_size==2 || column_size==3 *;next columns*

create_FA_inputs_A=PP[..] *;connect first input of full adder with suitable partial product.*

create_FA_inputs_B=PP[..] *;connect second input of full adder with suitable partial product.*

create_FA_inputs_Cin=PP[..] *;connect third input of full adder with suitable partial product.*

calculate_Cout and propogate_to_next_column *;Cout=A*Cin+B*Cin+A*B*

calculate_Sum and propogate_to_next_stage *;S=Cin XOR A XOR B*

compare Cout with earlier Cout *;compare signals with earlier ones and calculate activity factor*

compare Sum with earlier Sum *;if only sum or Cout changed switching=0.5,*

calculate_switching *;if both changed switching=1, if none changed switching=0*

;switching activity calculated with the comparison of each full adders earlier sum and Cout with current ones. If only one of them changes then the activity factor is 0.5, if both changes it is 1 and if there is no change on both of them then the activity factor is 0.

else

divide column size to 2 or 3 *; divide column to smaller 2 or 3 inputs* and check again.

end

product=selected_Couts_Sums *;final product is equal to Cout or Sums of some full adders.*

Activity_Factor=total_switching /m*n *;calculate activity factor of multiplication.*

Hardware_Size=n*m*9 *;total number of gates, each FA has 9 gates*

Delay=(m+n-1)*3*d; *;worst case delay for each gate*

Edyn=0.5*c* Hardware_Size*Activity_Factor*v^2	;*calculate dynamic energy*
Estat=I*V* Hardware_Size*(1- Activity_Factor)*Delay	*;calculate static energy*
E=Edyn+Estat	*;calculate total energy*
EDP=E*Delay	*;calculate EDP*

WT architecture decreases the order of growth of delay to O(lg(n)), and the complexity of hardware size remains $O(n^2)$.

CHAPTER 4

ABACUS MULTIPLIER

In this chapter, the new ABACUS multiplier architecture is presented, and an algorithm is reviewed for modeling it in PETAM. This study introduces a logic design and verification of the ABACUS multiplier. Energy dissipation is the first concern of the proposed multiplier architecture. After calculation of the partial products with the use of standard AND gates, partial products require to be aligned such that all digits with the same binary weight has to be in the same vertical column. As Figure 11 shows, the ABCUS multiplier will have an isosceles triangle shape that has a long side at the bottom [3].

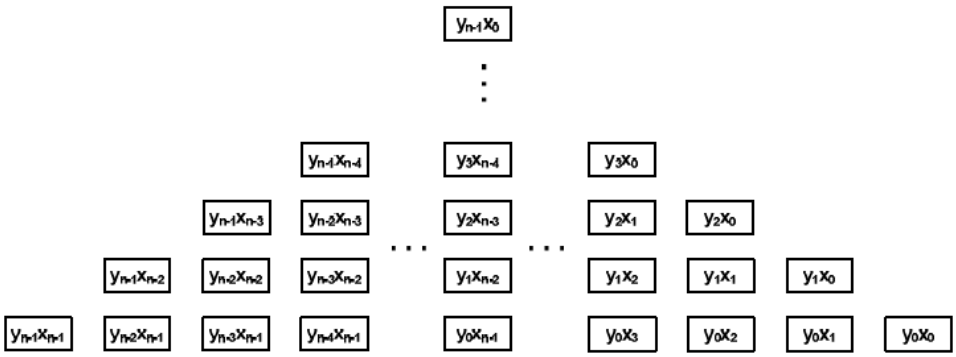


Figure 11. ABACUS partial product alignment.

Muhtaroğlu proposes a threshold function for the implementation of many parallel carry operations in ABACUS multiplier in addition to a set of rules for carry operations [3]. This approach requires consecutive compression and carry cycles, after which a result is obtained in the bottom row. The initial analysis by the author, which was done using some general assumptions about the ‘cost’ of a

carry and compression cycle as compared to add and carry cycles in a CSA based architecture, indicates up to an order of magnitude better opportunity in energy-delay product. However, it is quickly observed that the described high level architecture is untrivial to implement in a scalable manner. Logic implementations need to be completed before the actual advantage can be evaluated.

In this work, CSAM, RCAM and ABACUS have all been implemented with full adders (FA) as building blocks, for scalability and ease of comparison across architectures without indulging in the circuit design details. All FAs were implemented with NAND gates for the same reasons. In addition to FAs, the ABACUS multiplier implementation has parallel counters that decide on the position of carry. Since parallel counter approach does not require a separate column-wise compression, the implementation combines the compression and carry cycles reported in [3].

Parallel counters are categorized as (3, 2), (7, 3), (15, 4), (31, 5) and (63, 6). The architectural design of ABACUS multiplier principally uses FAs to calculate addition of 2 and 3 bits. As a result, the first, second and third columns do not need any parallel counters for addition. Other columns with more than 3 elements need parallel counters to propagate carries to the next stage. A multiplier in this framework thus consists of various stages of full adders, either stand-alone or within parallel counters, each stage adding up to the total delay. The scaleable ABACUS logic was thus modeled in PETAM, based on the developed rules for implementation and scaling. After reading the provided multiplier and multiplicand, the algorithm develops the columns for m x n multiplication. The sizes and the numbers of the parallel counters per column are determined next. Figure 12 shows the 4 bit ABACUS multiplier design. In the first stage of architecture, 7 FAs are used, 3 of which are for the (4,3) parallel counter. The second stage uses 5 FAs, and finally last stage requires only 2 FAs to generate the result. This approach minimizes carry operations and results in decreased delay. Scalability of ABACUS multiplier comes from the repeated structure for larger multiplications. For instance, the multiplication (m+1) x (m+1) has m+1 height column in the middle and two extra parallel counters (m, round(lg(m)+1)) and (m+1, round(lg(m+1)+1) in addition to the hardware of m x m multiplication.

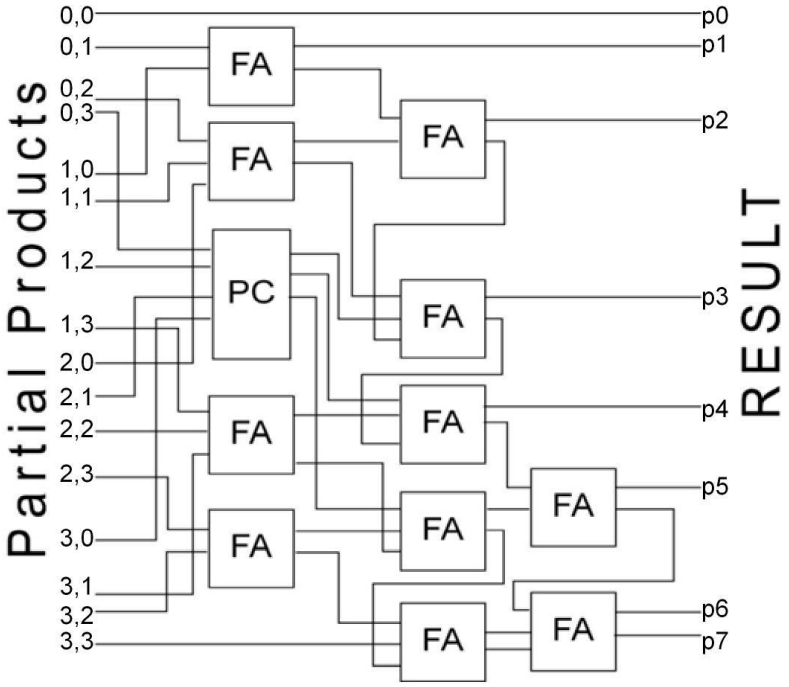


Figure 12. 4-bit ABACUS Multiplier design.

PETAM was used to design and verify correct functionality of any m x n multiplication. The main delay path is the propagation of the carry out from the previous stage to the next stage. ABACUS multiplier has the advantage of logarithmic delay where both in CSAM and RCAM the delay is linear. ABACUS multiplier architecture with hardware size $O(n^2)$ and delay=$O(\log(n)^2)$ has advantage when compared with CSAM and RCAM. Because CSAM has hardware size $O(n^2)$, delay=O(n), and RCAM hardware size $O(n^2)$, delay=O(n). The circuit layout is predicted to have complexities even though the speed of the operation is high, since the routing resource requirement is expected to be high, and this will result in increase in both energy and delay. Such physical layout disadvantages of ABACUS are outside the scope of this architectural study, and left out for future work.

4.1 Partial Products

Dividing the multiplication into smaller parts and addition of the results of the smaller multiplications is a method that is used often. This method is used in hardware as dividing multiplication down to one bit, and multiplying one bit with another, which can be done with a regular AND gate. These one bit multiplications are called a Partial Product (PP). The addition of all PPs yields the multiplication result.

Figure 13 shows the partial product generation process of 4x4 multiplication. Each bit from X is multiplied with each bit of Y. Additionally Figure 14 shows the hardware implementation for the same process.

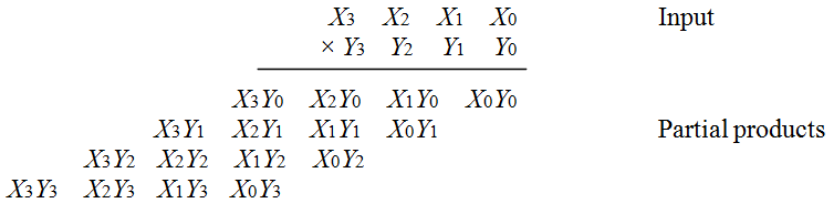


Figure 13. Basic bit level multiplication.

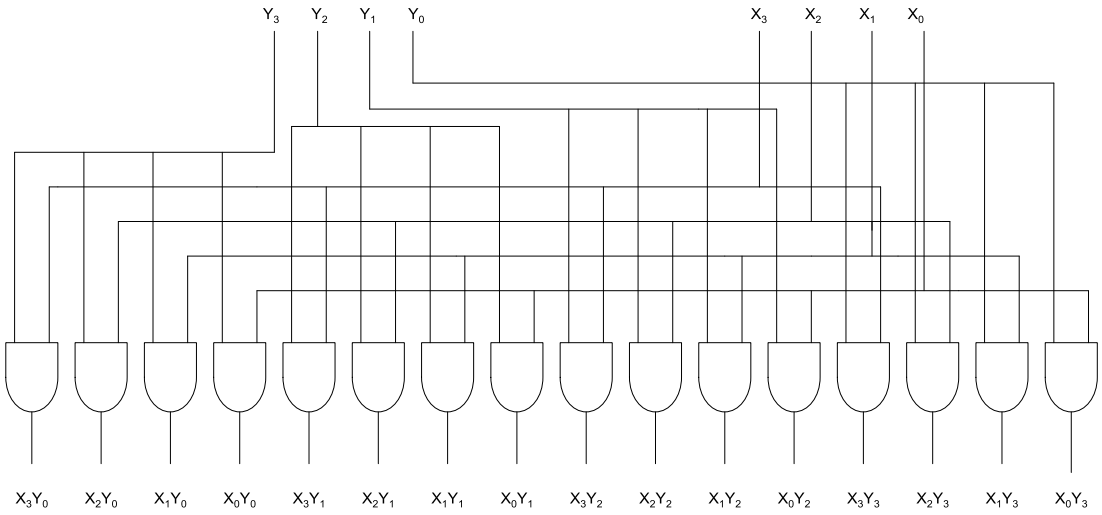


Figure 14. Partial product generation of 4x4 multiplication.

4.2 Parallel Counters and Compression

The hardware that counts the number of logic ones of m-bit inputs is called Parallel Counter (PC). The PCs are different from compressors. PCs do not have carry inputs and outputs where compressors have these in addition to regular inputs and outputs. Full adders are the most widely used parallel counters. FAs are (3,2) counters and HAs are (2,2) counters. Larger parallel counters are useful in the implementation of signal processing elements such as multipliers [7].

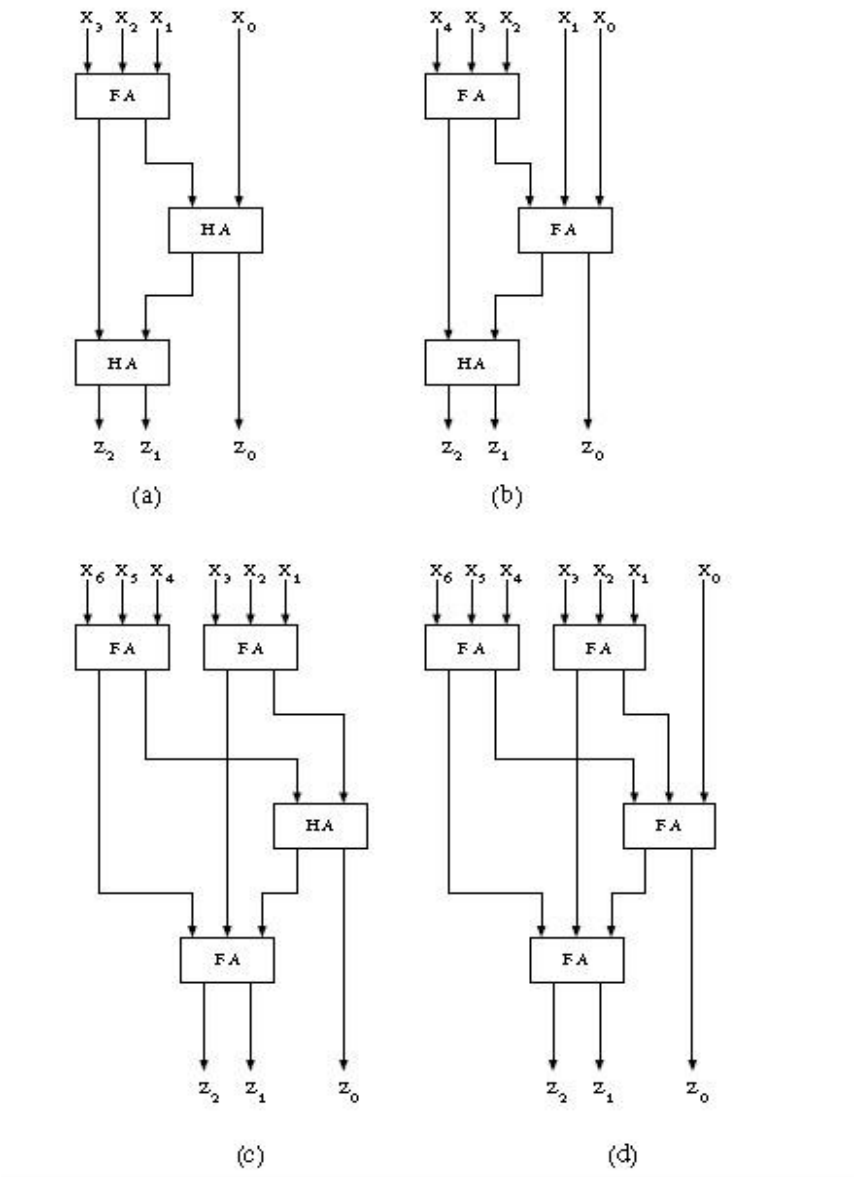


Figure 15. Parallel counter designs (a)(4,3), (b)(5,3), (c)(6,3), (d)(7,3).

A parallel *i*-to-*o* counter is denoted as (*i*, *o*), where *i* is the number of inputs, *o* is the number of outputs and o= $\lceil \log_2 i \rceil$. It is also called population counter. Parallel counters are extensively used in many different areas such as digital neural networks [25], triggering in multichannel high energy spectrometers [26], multiple input adders, parallel multipliers and various arithmetic multipliers [8].

Table 1. Hardware Size and Delay for 4-to-64 bit parallel counters.

Bits	Number of FA	Delay
4	3	3
5	4	4
6	4	4
7	5	5
8	6	5
9	7	5
10	8	6
11	9	6
12	9	6
13	10	6
14	11	7
15	12	7
16	13	7
17	14	7
18	15	7
19	16	7
20	17	8
21	18	8
22	19	8
23	19	8
24	20	8
25	21	8
26	22	8
27	23	9
28	24	9
29	25	9
30	26	9
31	27	9
32	28	9
33	29	9
34	30	9
35	31	9
36	32	9
37	33	9
38	34	9
39	35	10
40	36	10
41	37	10
42	38	10
43	39	10
44	40	10
45	41	10
46	41	10
47	42	10
48	43	10
49	44	10
50	45	10
51	46	10
52	47	10
53	48	10
54	49	11
55	50	11
56	51	11

57	52	11
58	53	11
59	54	11
60	55	11
61	56	11
62	57	11
63	58	11
64	59	11

The traditional approach for the architecture of parallel counters consists of a tree of HAs and FAs. Figure 15 shows the schematics of (4, 3), (5, 3), (6, 3) and (7, 3) PCs. In this research FAs are used as the building blocks, including the implementation of parallel counters. This approach does not yield the most efficient method and affects the total efficiency of ABACUS multiplier; however, the aim of the study is to implement ABACUS multiplier in a way that allows a fair comparison with other multiplier topologies. The counter level improvements have a big impact on the performance of larger arithmetic circuits [8].

The total cost of a (i, o) PC with this approach is i-o FAs. On the other hand, the delay is expected to be $d=2*\lceil \log_2 i \rceil -1$. The delay and hardware size graphs of PCs are shown in Figure 16 and Figure 17 respectively. It is shown in Figure 16 that the rate of delay change decreases as the size of PCs increases. On the other hand the hardware size increases linearly. This in turn affects the efficiency of ABACUS multiplier for small and large size multiplications.

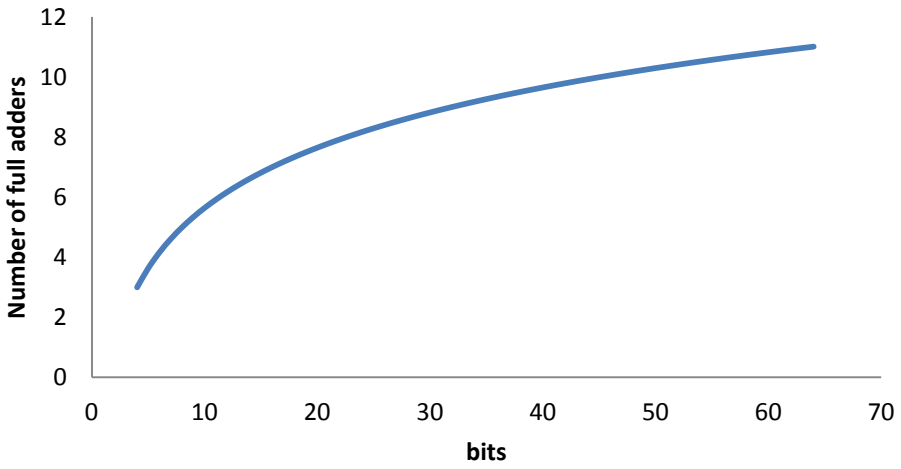


Figure 16. Delay of parallel counters.

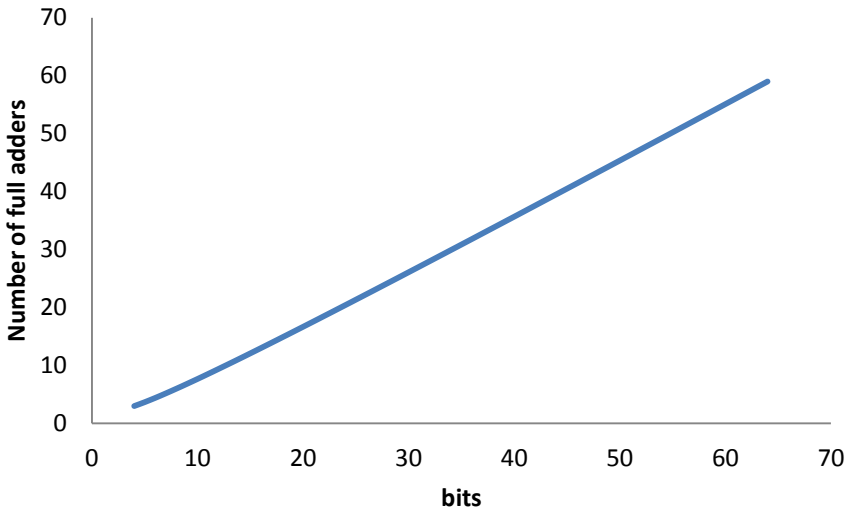


Figure 17. Hardware size of parallel counters.

There are other approaches for the implementation of parallel counters where the switching activity of each intermediate output is lower. One of these methods called 1-out-of-n is proposed by McIlhenny and Ercegovac in 1996 and results shows that this new approach reduces the delay and switching activity, and produces less glitching at the outputs, which significantly reduces the average power consumption [27].

Column compression multipliers are mostly designed for performance improvements. Research done by Callaway in 1993 shows that the column compression multipliers are more power efficient than array multipliers [28]. On the other hand, the delay and power characteristic of these multipliers need to be well understood, and overall improvements need to be clear before the implementation.

As described earlier, Wallace proposed a method for column compression tree multipliers for reduction of propagation delay. Afterwards, Dada worked on Wallace's method and defined a counter placement strategy that reduces the counters in partial production stage [8]. There are other partial product reduction methods like a new technique for column compression multipliers [29] and the Windsor [30] methods focused on the utilization of FAs and HAs. These techniques are more area efficient and have shorter interconnections. Oklobdzija, et al. proposed an algorithm for partial product reduction based on understanding the unequal delay paths through counters and compressors.

Oklobdzija's method sorts inputs and outputs then connects these inputs and outputs for signal paths and critical paths to tolerate the increase in delay. This technique connects fast inputs and outputs in the critical delay paths and assigns slow inputs and outputs to signal [31].

The study in this thesis uses gate level implementation models to identify the EDP performance of the ABACUS multiplier. Column compression part of the ABACUS multiplier is implemented with parallel counters, and is modeled by PETAM. The compression cycles of each multiplication depends on the carry propagation from parallel counters. PETAM algorithm for ABACUS implementation checks column width in an infinite loop until it reaches two.

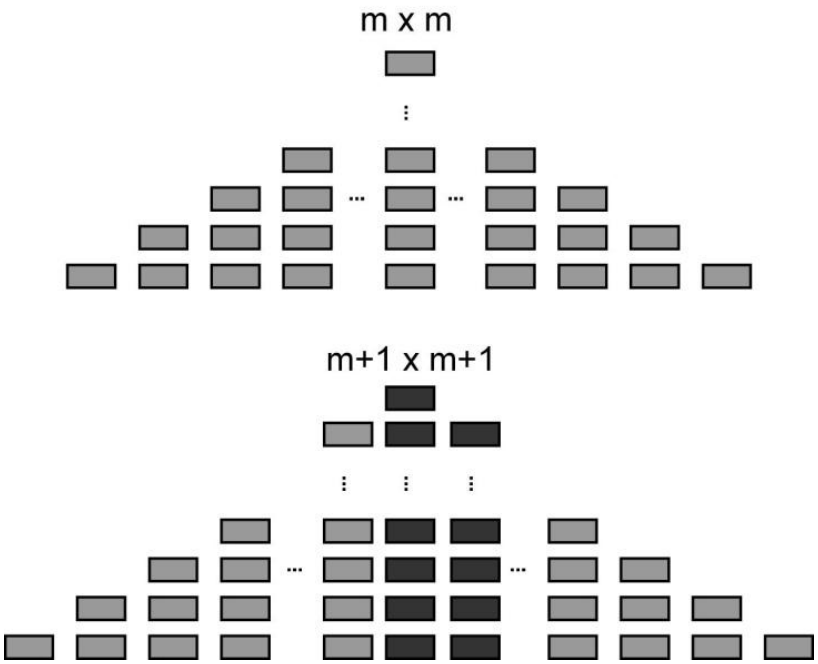


Figure 18. mxm and (m+1)x(m+1) multiplications with ABACUS
extra hardware same hardware

Figure 18 shows the hardware difference of two differently sized multiplications. This study uses higher order parallel counters for the column compression. ABACUS multiplier hardware size increases by the multiplicand and multiplier size increase. The difference between the m x m multiplier and m+1 x m+1 multiplier consists of two parallel counters in size (m+1, $\lceil \log_2(m+1) \rceil$) and (m,

$\lceil \log_2 m \rceil$). This difference between two different size multiplications linearly increases in number of FAs. However, ABACUS multiplier uses PCs. This decreases total hardware size and also carry cycles.

4.3 Final Adder

The final stage in ABACUS multiplier architecture takes place when only two rows remain. There are different adder implementation options for this purpose. Since the final addition time adds directly to the critical path of the multiplier it is important to choose fastest implementation. On the other hand, it is necessary to use an easy to implement and scalable method for PETAM to work efficiently across multiplier sizes.

The choices for the adder consist of conditional sum adder (CSMA), carry select adder (CSLA) carry-look-ahead adder (CLA) and ripple carry adder (RCA). The CSLA uses less hardware than the CSMA and can be considered as a subset of it. In 1989 Yeung and Yu suggested that the difference in speed between these adders are strongly dependent on the profile of input as it changes from uniform to non-uniform [32]. Even though CSLA is slower than CSMA, it is still more popular than CSMA due to its simplicity when compared with CSMA. Smaller size and uncomplicated implementation of CSLA reduces the speed advantage of CSMA. On the other hand, CLA has become popular due to its speed and modularity. There are different variations of CLA for especially larger addition operations, which have significant delay decrease [14].

For most parallel multiplier implementations, simple ripple carry adder is used as a choice of final adder. The study by Negandra examined several parallel adder implementations. This study showed that RCA is suitable for small width summation [14]. The early research on ABACUS multiplier suggested that the EDP of the architecture was mostly dominated by the power dissipation due to hardware size instead of delay. Using unnecessarily powerful and complex architectures for the final adder stage was therefore not deemed necessary in ABACUS implementation. Ripple carry adder was used for final addition stage to balance the delay and power. A small width multiplication operation has larger delay in ABACUS implementation and RCA has advantages on small size operations so it is the appropriate choice for the final addition stage. On the other hand, there is still

room for further research on different parallel adders and their effect on total EDP of multiplication. In this particular research the main idea was implementing ABACUS multiplier and comparing it with selected multipliers so the final addition stage is common, just like partial production stage for all multiplier implementations and the real comparison across selected multipliers is about the compression part. Next chapter discusses results and gives more detail about comparison of multipliers. The difference on EDP mentioned here is not about partial products generation or final addition stages.

4.4 Pseudocode to Simulate ABACUS

The following is the pseudocode for the model in PETAM to simulate the signal propagation in the ABACUS architecture.

```
Initialize;                 ;Initialization Section.
m=get_multiplier_size       ;read multiplier from textbox and save size of it to m.
n=get_multiplicand_size     ;read multiplicand from textbox and save size of it to n.
c= get_capacitance          ;read capacitance of a NAND gate from textbox and save it to c.
v= get_voltage              ;read voltage of a NAND gate from textbox and save it to v.
I= get_leakage_current      ;read leakage of a NAND gate current from textbox and save it to I.
d= get_delay;               ;read delay of a NAND gate from textbox and save it to d.


first_stage(n)=first_stage(n-1)+(n-lg(n)+1)+(n-1-lg(n-1)+1)   ;calculate hardware size of first stage

stage_number= log(m)+1      ;calculate number of stages needed to complete multiplication


Partial product generation  ;Partial Product Generation Section.
for each bits of multiplier
        for each bits of multiplicand
                PP=m AND n          ;create partial products array with using AND gates.
        end
end
```

Form Parallel Counters for each column with full adders;

for y=1 to y=stage_number

while (maximum_column_size>2)

if column_size==1 *;the first column where there is only one element*

propogate_to_next_stage

if column_size==2 || column_size==3 *;next columns*

create_FA_inputs_A=PP[..] *;connect first input of full adder with suitable partial product.*

create_FA_inputs_B=PP[..] *;connect second input of full adder with suitable partial product.*

create_FA_inputs_Cin=PP[..] *;connect third input of full adder with suitable partial product.*

calculate_Cout and propogate_to_next_column *;Cout=A*Cin+B*Cin+A*B*

calculate_Sum and propogate_to_next_stage *;S=Cin XOR A XOR B*

compare Cout with earlier Cout *;compare signals with earlier ones and calculate activity factor*

compare Sum with earlier Sum *;if only Sum or Cout changed switching=1/2=0.5,*

calculate_switching *;if both changed switching=1, if none changed switching=0*

;switching activity calculated with the comparison of each full adders earlier sum and Cout with current ones. If only one of them changes then the activity factor is 0.5, if both changes it is 1 and if there is no change on both of them then the activity factor is 0.

else

while (column_size>=n)

create_PC_nlg(n)_inputs[n] *;connect inputs of parallel counters with suitable partial product.*

create_PC_ nlg(n)_outputs[lg(n)] *;connect outputs of full adder with suitable full adders*

result=sum *;output which is saved for next stage*

connect_each_output_ as_carry_to_next_columns *;outputs which are propagated for next stage.*

compare Cout with earlier Cout *;compare signals with earlier ones and calculate activity factor*

compare Sum with earlier Sum *;if only sum or Cout changed switching=0.5,*

calculate_switching *;if both changed switching=1, if none changed switching=0*

product=selected_Couts_Sums *;final product is equal to Cout or Sum of some full adders.*

Activity_Factor=total_switching /m*n	*;calculate activity factor of multiplication.*
Hardware_Size= first_stage(n)+((lg(n)+1)*(n-2))*9	*;total number of gates, each FA has 9 gates*
Delay=2*lg(n)*(lg(n+1))*3*d;	*;worst case delay for each gate*
Edyn=0.5*c* Hardware_Size*Activity_Factor*v^2	;*calculate dynamic energy*
Estat=I*V* Hardware_Size*(1- Activity_Factor)*Delay	*;calculate static energy*
E=Edyn+Estat	*;calculate total energy*
EDP=E*Delay	*;calculate EDP*

The algorithm of ABACUS starts with initialization of multiplier size, multiplicand size, capacitance, voltage, leakage current and delay variables like other algorithms. Total stage number in ABACUS is important to use in loops as stop criteria. Afterwards, algorithm calculates the partial products and creates FAs and parallel counters. For more than 3 inputs instead of FAs PCs are used. Each FA has A, B, Cin as inputs and, Cout and S as outputs. And each parallel counter has (n, lg(n)) inputs and outputs. The calculation of each FA's and PC's output occurs in loops and at the end of this process product variable has the result. For each low to high or high to low transition, switching variable is increased and activity factor is calculated as a probability. Total energy is calculated by the summation of dynamic and static energy; the explanation and details of each formula could be seen in Chapter 2.

CHAPTER 5

RESULTS

In this chapter, the new ABACUS multiplier architecture is compared with CSAM, RCAM and WTM architectures. The comparisons are made in terms of delay, power and EDP for different multiplication bit lengths. Also analysis of new ABACUS architecture is shown in this chapter.

The performance of multipliers has been studied extensively, still in an empirical block based design. Some factors need to be taken into consideration: One of these factors is wiring effect. Earlier research focused on and tried to reduce the arrival times to make zero wire delays [30]. The delays from input pins to output pins of the gates are not equal. The gate delays are also functions of loading capacitance and input signal slopes, which cannot be estimated well without detailed physical information. Due to these difficulties, full adders are used in this research.

Table 2 is the average of output files from PETAM. It shows the results of 2x2 to 64x64 bits multiplications for 4 different pattern input files which are read from text file for CSAM, RCAM, WTM and ABACUS multipliers where Vcc=5V, C=50pF, $I_{leakage}$=1000nA and delay=20ns for each NAND gate. It is assumed that HD74HC00 Hitachi Integrated Circuit CMOS – Quad 2 Input NAND Buffer/Driver chip is used at 25 °C [33].

Table 2. DELAY, POWER, AND EDP COMPARED ACROSS CSAM, RCAM, ABACUS

Multiplier Topologies												
SIZE	***CSAM***			***RCAM***			***WT***			***ABACUS***		
m x n	**Delay(ns)**	**Power(uW)**	**EDP(uJs)**	**Delay(ns)**	**Power(uW)**	**EDP(uJs)**	**Delay(ns)**	**Power(uW)**	**EDP(uJs)**	**Delay(ns)**	**Power(uW)**	**EDP(uJs)**
4 x 4	360.00	0.01	3.18	480.00	0.02	7.54	360.00	0.01	2.08	540.00	0.03	17.43
8 x 8	840.00	0.08	66.54	960.00	0.08	80.41	960.00	0.10	98.85	1200.00	0.14	170.15
12 x 12	1320.00	0.27	352.34	1440.00	0.22	318.68	1422.00	0.31	436.00	1697.00	0.36	612.68
16 x 16	1800.00	0.58	1040.72	1920.00	0.44	836.56	1800.00	0.61	1100.00	2100.00	0.67	1414.19
20 x 20	2280.00	1.02	2315.38	2400.00	0.75	1810.29	2121.00	1.00	2142.00	2440.00	1.08	2642.07
24 x 24	2760.00	1.61	4433.93	2880.00	1.19	3438.37	2402.00	1.50	3818.65	2737.00	1.69	4618.04
28 x 28	3240.00	2.35	7623.81	3360.00	1.76	5923.99	2653.00	2.24	5952.48	3001.00	2.36	7077.49
32 x 32	3720.00	3.26	12120.08	3840.00	2.49	9553.48	2880.00	3.02	8710.00	3240.00	3.16	10235.22
36 x 36	4200.00	4.35	18258.59	4320.00	3.39	14623.64	3087.00	3.90	12073.00	3458.00	4.07	14065.00
40 x 40	4680.00	5.62	26316.72	4800.00	4.46	21430.33	3278.00	4.90	16145.00	3658.00	5.11	18685.29
44 x 44	5160.00	7.09	36584.30	5280.00	5.75	30366.27	3456.00	6.07	20985.00	3844.00	6.28	24159.95
48 x 48	5640.00	8.77	49468.05	5760.00	7.26	41826.00	3623.00	7.16	27206.00	4018.00	7.77	31226.21
52 x 52	6120.00	10.67	65298.87	6240.00	9.01	56203.42	3779.40	8.94	33805.68	4181.00	9.24	38654.92
56 x 56	6600.00	12.97	84425.09	6720.00	11.01	73992.28	3927.00	10.52	41308.00	4335.00	10.86	47084.03
60 x 60	7080.00	15.16	107316.87	7200.00	13.29	95686.50	4066.00	12.23	49754.00	4481.00	12.62	56555.57
64 x 64	7560.00	17.77	134362.34	7680.00	15.86	121778.49	4200.00	14.09	59180.00	4620.00	14.53	67109.62

Table 2 is shorter version of detailed breakdown file where only multitude of 4 is listed. Appendix E has whole breakdown file where all size of multiplications are listed for chessboard pattern. On the other hand Table 2 has the average results of all patterns. The data used to draw the comparison graphics is derived from this table.

5.1 Comparison of Multiplier Architectures

This research compares the energy and performance metrics of fundamental array multipliers with ABACUS multiplier. For EDP estimations mainly PETAM was used and comparisons were made for each multiplication. These comparisons have 3 different matrices as power, delay and EDP. Each metric shows different properties, and indicate the similarities and dissimilarities across the multiplier architectures under analysis.

5.1.1 Delay and Power Comparison

Architectural delay and power comparison of CSAM, RCAM, WTM and ABACUS multiplier are shown in Figure 19 and Figure 20 respectively. It is observed that the maximum delay occurs for RCAM for multiplications larger than 20x20. Less delay occurs in WTM and ABACUS multipliers when compared with CSAM and RCAM for the same range.

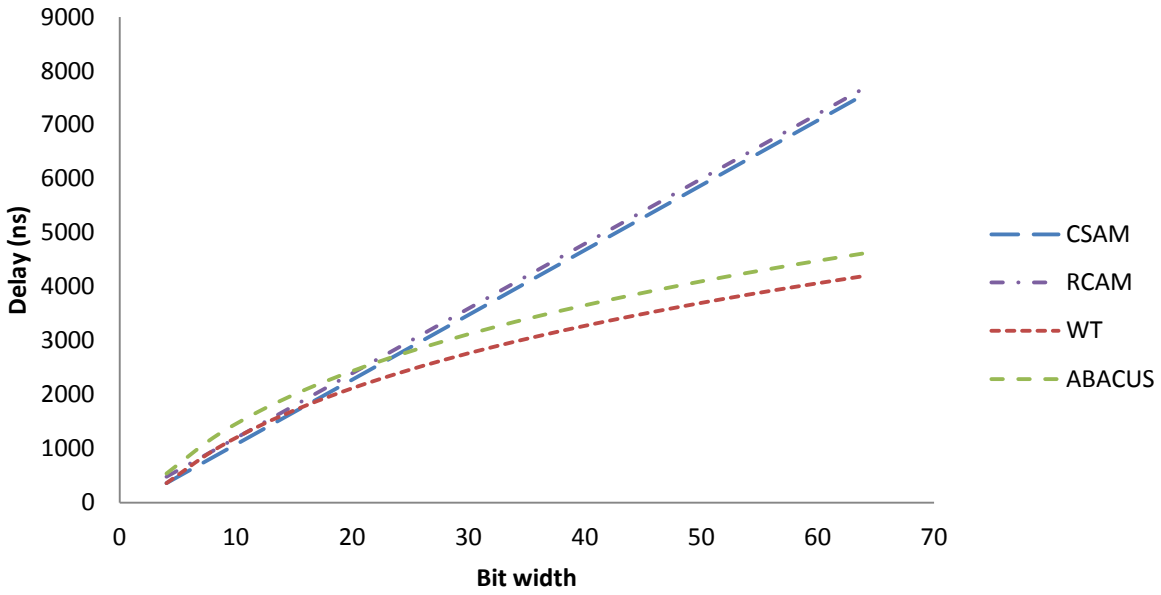


Figure 19. Architectural delay comparison of CSAM, RCAM, WT and ABACUS.

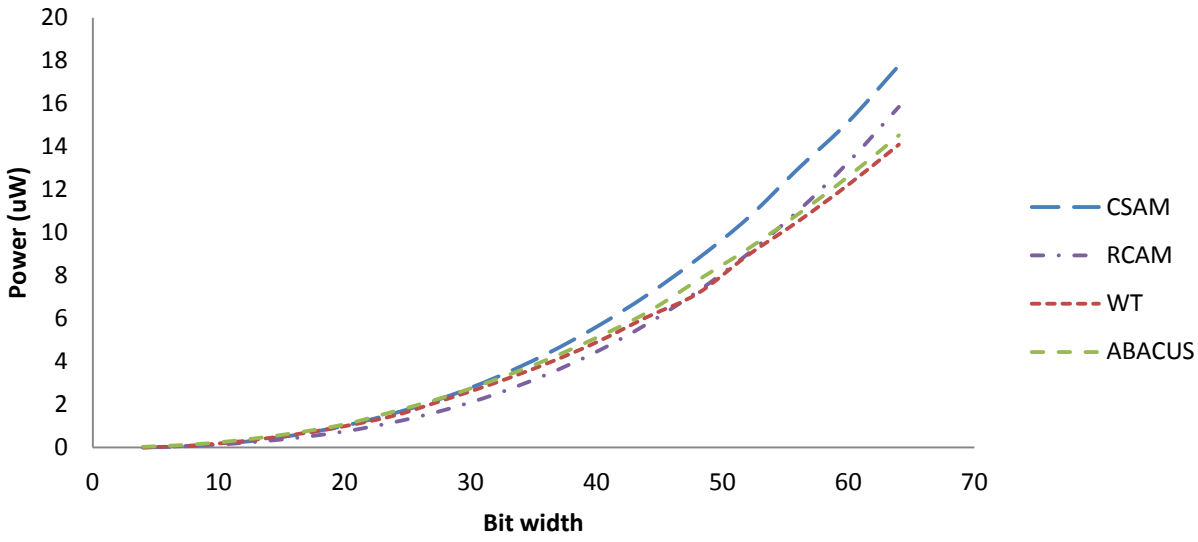


Figure 20. Architectural power comparison of CSAM, RCAM, WT and ABACUS.

WTM and ABACUS multiplier need more power for multiplication less than 32x32 bits when compared with array multipliers. Due to large hardware size of WTM and ABACUS multipliers, higher power dissipation occurs for small size multiplications. However, WTM and ABACUS multipliers have lower power for larger multiplications.

5.1.2 Energy Delay Product Comparison

Figure 21 shows the EDP vs. Bit width graph of m x n multiplications from Table 2. Due to the logarithmic order of growth property, WT and ABACUS have slightly higher EDP for small size multiplications, but are predicted to be far superior for larger multiplication designs. The resulting implementation does not provide advantage in EDP when the multiplier size is smaller than 32x32. The reason behind this is larger hardware size of ABACUS multiplier when compared with CSAM, RCAM and WT. Parallel counter based multiple carry operations do not provide a significant advantage for small size multiplications, but increase the total hardware size.

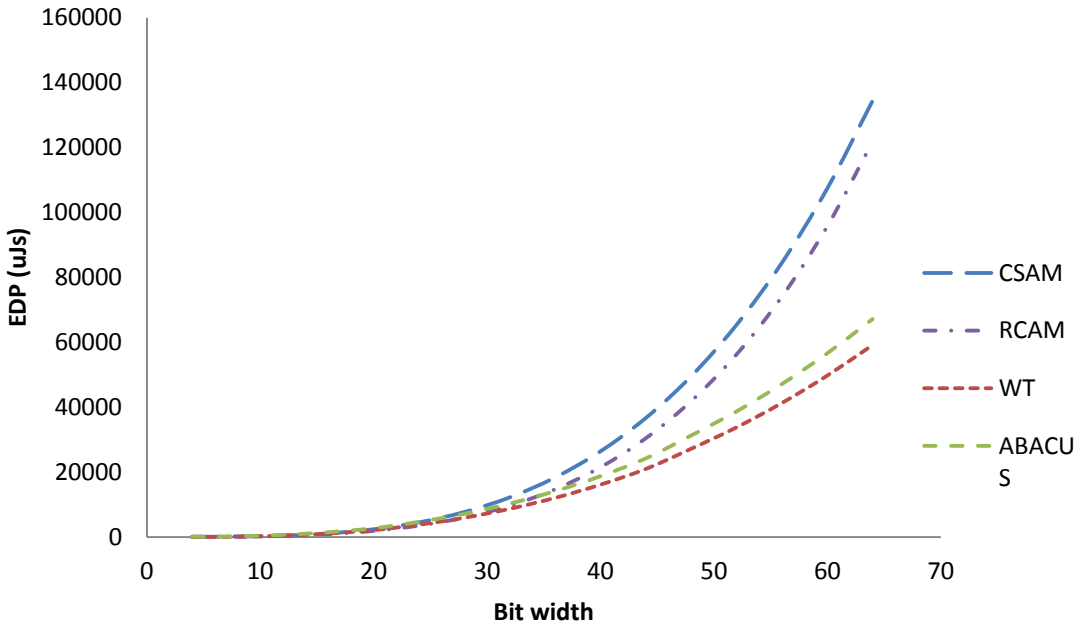


Figure 21. Architectural EDP comparison of CSAM, RCAM, WT and ABACUS.

It is observed that there is 5-40% difference between the EDP of selected 4 patterns. The highest activity factor difference between selected 4 patterns is around 0.4 and it is between checkerboard pattern and all1's pattern. This pattern effect makes gaps smaller or larger between different multiplier topologies and changes the threshold multiplication size. However the relative trend of EDP line for each architecture does not significantly change. Reported results are averaged over the full input vector set. If we compare architectures for each vector type the differences of EDP between architectures are changing but it is very small and there is no change in relative trends.

CHAPTER 6

CONCLUSION

In this study, a viable logic implementation is introduced for the ABACUS multiplier architecture for the first time, and is compared with CSAM and RCAM, using input library component parameters. These parameters may come from circuit simulations or measurements, but datasheet numbers were used from discrete NAND components in this work, only for relative comparison purposes. Based on the results, it was concluded that FA based ABACUS multiplier implementation gives better performance for large size multiplications in terms of energy-delay product when compared to the FA based CSAM and RCAM architectures. Although the ABACUS multiplier power consumption is more than CSAM and RCAM due to hardware size (and no specific circuit optimization for static power), great decrease on delay due to the parallel carry algorithm makes ABACUS multiplier attractive. The resulting implementation models did not offer as much advantage in energy-delay as the originally reported crude architectural analysis, especially when the multiplier size is smaller than 32x32. This is due to the fact that threshold detection required by ABACUS architecture parallel carry operations is not trivial to implement at low cost using standard logic approaches. On the other hand, the proposed logic implementation of ABACUS in this paper is scalable to any m x n integer multiplier, and demonstrates close to 2x EDP improvement potential compared to scalable RCAM and CSAM logic implementations for 64x64 bits multiplication, and more for larger multipliers. The next steps involve building of the simulated multiplier circuits using the discrete standard components in order to extract any miscorrelation factors between predictions and measurements.

It was recognized after the completion of the design that the WTM architecture implementation has similarities with the ABACUS multiplier logic implementation as described in this study. Compressors in WTM are built from parallel counters that help the multiplier delay to scale logarithmically as in the case of ABACUS, but these increase the irregularity of the topology [34]. It is predicted that the FA based implementation would make ABACUS slower than WT. The main difference between ABACUS multiplier and WT multiplier is the size of parallel counters. The bit width of multiplication determines the biggest parallel counter size in ABACUS multiplier where in WTM architectures the compressor

size is typically constant (4, 3), and algorithm includes rules to divide column size to necessary number of compressors. This algorithm of WT architecture decreases the order of growth of delay to O(lg(n)) and WT has the ability to calculate multiplication faster than ABACUS with $O(lg(n)^2)$. Yet it increases the irregularity of topology.

This study shows that the new ABACUS multiplier architecture has advantages in comparison with array multipliers even without making any more optimization to the proposed architecture. On the other hand, WTM has better EDP profile when compared with proposed logic implementation of ABACUS multiplier. The optimization of ABACUS multiplier may be possible by using pipeline architecture or different logic implementation of parallel counters or without FA based modeling. For 64x64 bits multiplication with ABACUS multiplier, largest PC consists of 59 FAs in the middle of the first stage. Total hardware size for PCs in the middle is totally 175 FAs or 1575 NAND gates. The total difference between WTAM and ABACUS multiplier hardware size is 189 NAND gates. It is possible for ABACUS to reach WTM EDP with 15-20% less hardware for these 3 PCs. With 20% less hardware size on only these 3 PCs ABACUS multiplier would have 126 NAND gates less than WTM and reach same EDP. The results of this study prove that the ABACUS multiplier is promising in the sense of low power multiplier architecture especially for multiplications for use in encryption/decryption or similar areas where the operand width is greater than 64 bits.

6.1 Future Work

This study mainly focused on the comparison of ABACUS multiplier with other fundamental array and tree type multipliers. It was necessary to use the same building block and PETAM software to be consistent and fair for the comparison of other multipliers. On the other hand, assumptions like using NAND gates only for FA implementation or the placement of wiring strategies have large impact on power and weren't considered in this research due to the primary aims of the research. Even though it was very helpful to have PETAM especially for testing and controlling the effects of algorithm, it performs first order estimate of design metrics. In the future, research including detailed analysis of empirical model would be more compelling. In particular, choosing fewer bit widths and completing a detailed design in lab environment would provide more trustworthy comparison results.

REFERENCES

[1] Landman, Paul E. *Low-power architectural design methodologies*. Diss. University of California, 1994.

[2] Zhu, Changyun, et al. "Towards an ultra-low-power architecture using single-electron tunneling transistors." *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*. IEEE, 2007.

[3] Muhtaroğlu, Ali. "ABACUS: A novel array multiplier-accumulator architecture for low energy applications." *IEEE International Conference on Energy Aware Computing (ICEAC).* 2010.

[4] Gurdur, Didem, and Ali Muhtaroğlu. "PETAM: Power estimation tool for array multipliers. "*IEEE International Conference on Energy Aware Computing (ICEAC).* 2012.

[5] Horowitz, Mark, Thomas Indermaur, and Ricardo Gonzalez. "Low-power digital design." *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*. 1994.

[6] Ravi, Nirlakalla, et al. "A New Design for Array Multiplier with Trade off in Power and Area." *arXiv preprint arXiv:1111.7258*, 2011.

[7] Arechabala, J., et al. "Full systolic binary multiplier." *Circuits, Devices and Systems, IEE Proceedings G*. Vol. 139. No. 2. IET, 1992.

[8] Booth, Andrew D. "A signed binary multiplication technique." *The Quarterly Journal of Mechanics and Applied Mathematics*. 1951, pp 236-240.

[9] Dadda, Luigi. "On parallel digital multipliers" *Alta FrequenzaVol. 45*. 1976, pp 574-580.

[10] Wallace, Christopher S. "A suggestion for a fast multiplier." *IEEE Transactions on Electronic Computers*. 1964, pp 14-17.

[11] Baugh, Charles R., and Bruce A. Wooley. "A two's complement parallel array multiplication algorithm." *IEEE Transactions on Computers*. 1973, pp 1045-1047.

[12] Parhami, Behrooz. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc., 2009.

[13] Brown, Stephen D., and Zvonko G. Vranesic. *Fundamentals of Digital Logic with VHDL Design*. Vol. 6. New York: McGraw-Hill, 2000.

[14] Nagendra, Chetana, Mary Jane Irwin, and Robert Michael Owens. "Area-time-power tradeoffs in parallel adders." *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*.1996, pp 689-702.

[15] Sertbaş, Ahmet., and R. Selami ÖZBEY "A performance analysis of classified binary adder architectures and the VHDL simulations." *IU-Journal of Electrical & Electronics Engineering*. 2011.

[16] Fonseca, Marcelo, et al. "Design of a radix-2 m hybrid array multiplier using carry save adder format." *Proceedings of the 18th annual symposium on Integrated circuits and system design*. ACM, 2005.

[17] Krad, Hasan, and Aws Y. Al-Taie. "Performance Analysis of a 32-bit Multiplier with a Carry-look-ahead Adder and a 32-bit Multiplier with a Ripple Adder Using VHDL." *Journal of Computer Science*. 2008, pp 305.

[18] Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2001.

[19] Rabaey, Jan M., and Massoud Pedram, eds. *Low power design methodologies*. Kluwer academic publishers, 1996.

[20] Veendrick, Harry JM. "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits." *IEEE Journal of Solid-State Circuits*. 1984, pp 468-473.

[21] Wei, Gu-Yeon, and Mark Horowitz. "A low power switching power supply for self-clocked systems." *IEEE International Symposium on Low Power Electronics and Design*. 1996.

[22] Kumar, Jaya C., and R. Saravanan. "VLSI Design for Low Power Multiplier using Full Adder." *European Journal of Scientific Research*. 2012, pp 5-16.

[23] Shanthala, S., and S. Y. Kulkarni. "VLSI Design and Implementation of Low Power MAC Unit with Block Enabling Technique." *European Journal of Scientific Research*. 2009, pp 620-630.

[24] Meier Pascal C.H., Rob A. Rutenbar, L. Richard Carley, “Exploring Multiplier Architecture and Layout for Low Power” *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference*.1996, pp 513-516.

[25] D. Zhang, G. A. Jullien, W. C. Miller, and Earl E. Swartzlander. “Arithmetic for digital neural networks.” *10th Symposium on Computer Arithmetic*. 1991, pp 58-63.

[26] Nikityuk, N. M. "Use of parallel counters for triggering." *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*. 1992, pp 571-582.

[27] McIlhenny, Robert, and Miloš D. Ercegovac. "On Using 1-out-of-n Codes for (p, q) Counter Implementations” *IEEE Conference Record of the Thirtieth Asilomar Conference on Signals, Systems and Computers* Vol. 1. 1996.

[28] Callaway, Thomas K., and Earl E. Swartzlander Jr. "Optimizing multipliers for WSI." *Proceedings., Fifth Annual IEEE International Conference on Wafer Scale Integration.* 1993.

[29] Bickerstaff, K’Andrea C., Michael J. Schulte, and Earl E. Swartzlander, Jr., “Reduced Area Multipliers,” *Proceedings of the 1993 International Conference on Application Specific Array Processors*, 1993, pp. 478-489.

[30] Wang, Zhongde, Graham A. Jullien, and William C. Miller. "A new design technique for column compression multipliers." *IEEE Transactions on Computers.* 1995, pp 962-970.

[31] Oklobdzija, Vojin G., David Villeger, and Simon S. Liu. "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach." *IEEE Transactions on Computers.* 1996, pp 294-306.

[32] Yeung, A. K. W., and R. K. Yu. "A self-timed multiplier with optimized final adder." *Final Report for CS* 2921. Univ. of California at Berkeley. 1989.

[33] HD74HC00, QUAD. 2 input NAND Gates Datasheet, Hitachi LTD. 1999.

[34] Flynn, Michael J., and Steven S. Oberman. Advanced computer arithmetic design. John Wiley & Sons, Inc., 2001.

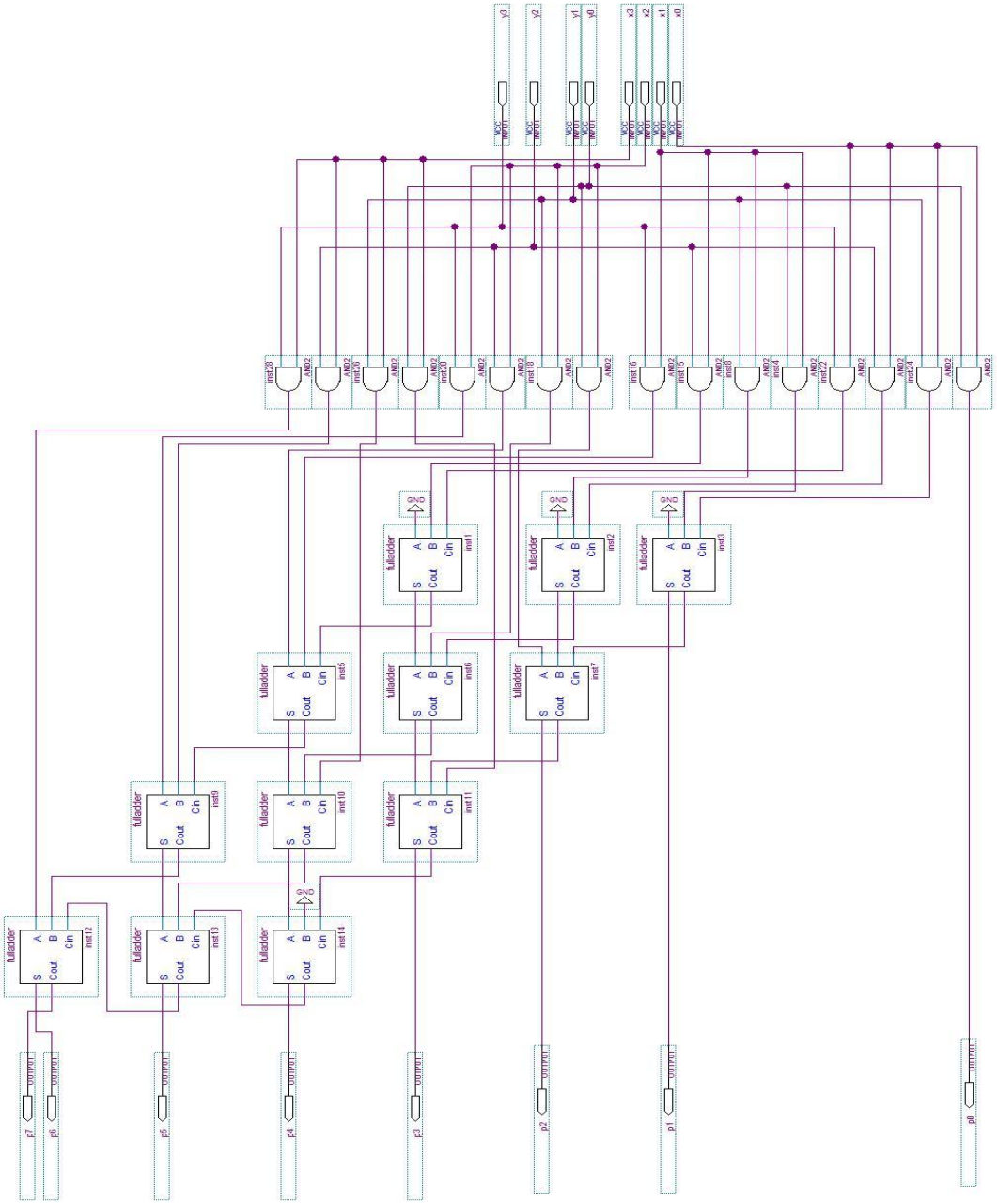
[35] Smith, Michael John Sebastian. *Application-specific integrated circuits*. Vol. 7. Reading: Addison-Wesley, 1997.

APPENDICES

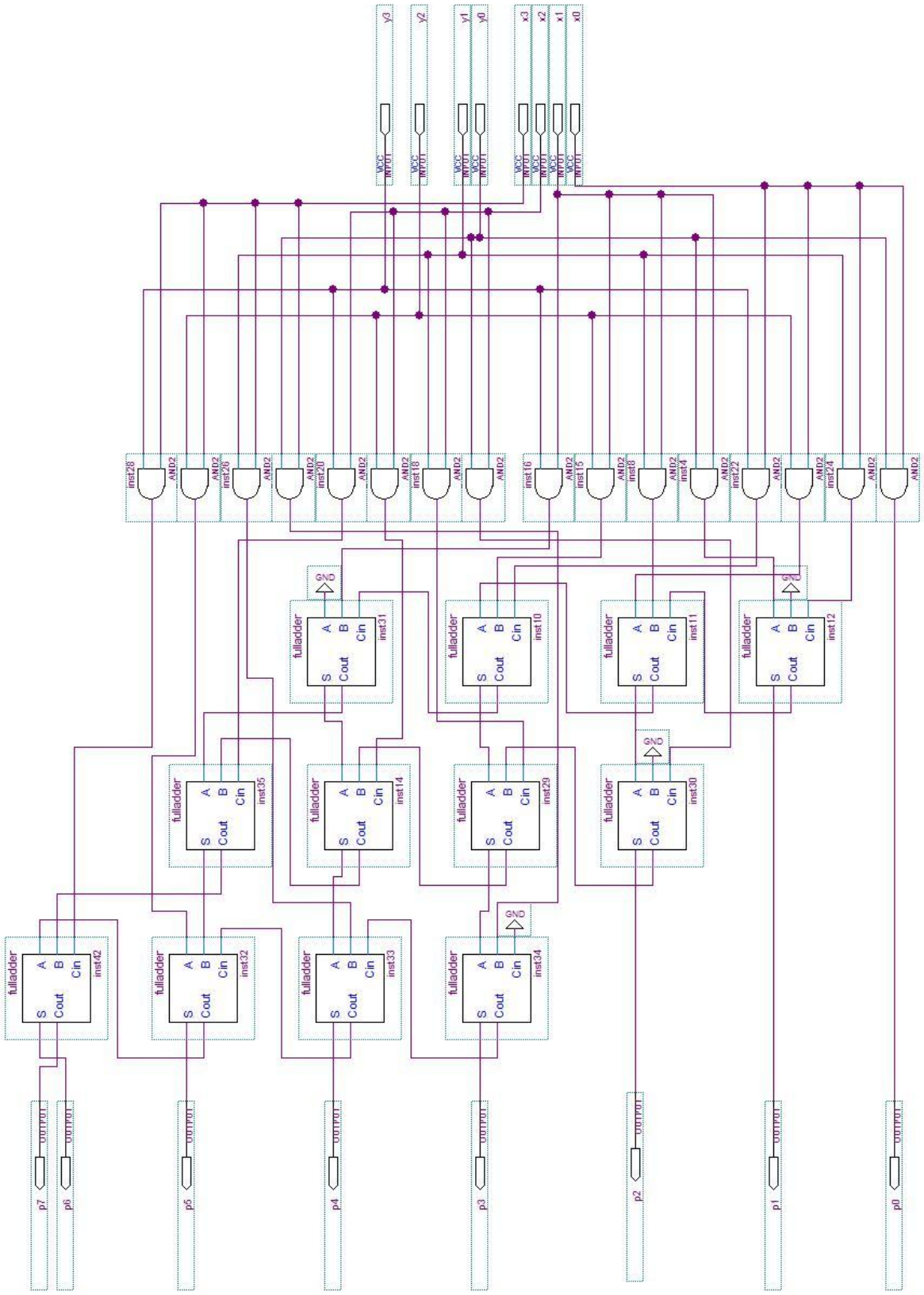
APPENDIX A

A. QUARTUS DESIGNS

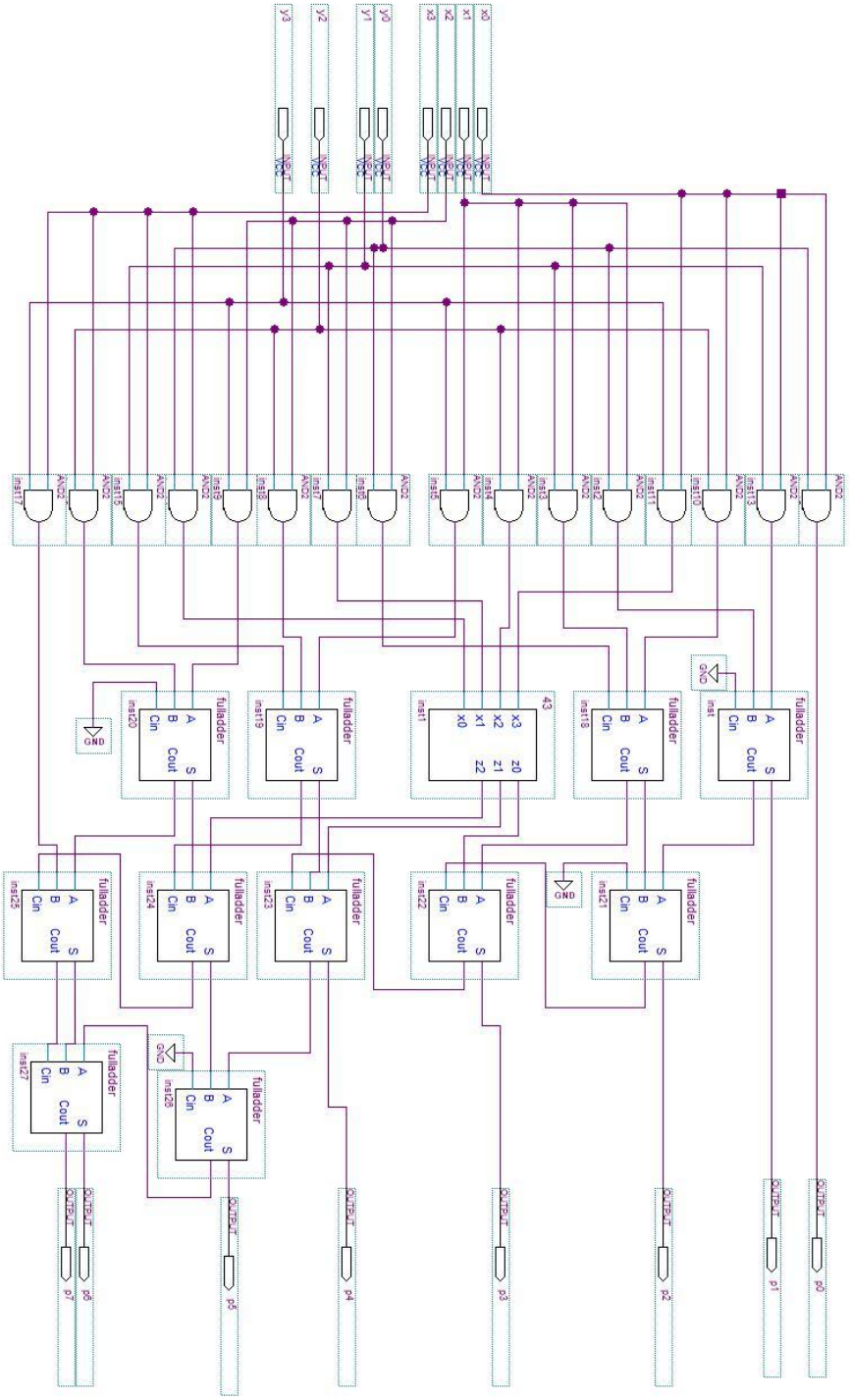
a. CSAM



b. RCAM


y3
y2
y1
y0
x3
x2
x1
x0
VCC
INPUT
AND2
fulladder
A
B
Cin
S
Cout
p7
p6
p5
p4
p3
p2
p1
p0
OUTPUT

c. ABACUS


y3
y2
y1
y0
x3
x2
x1
x0
fulladder
Cin
Cout
A
B
S
GND
p7
p6
p5
p4
p3
p2
p1
p0

APPENDIX B

B. C# CODES

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.IO;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Windows.Forms.DataVisualization.Charting;

namespace Multipliers
{

    public partial class Form1 : Form
    {
        int n1,n2,flag=0,count=0;

        public Form1()
        {
            InitializeComponent();
        }

        public void CSAM(int[] m,int[]q)
        {
            int[] cin = new int[n1 * (n1 + 1)];
            int[] cout = new int[n1 * (n1 + 1)];
            int[] s = new int[n1 * (n1 + 1)];
            int[] A = new int[n1 * (n1 + 1)];
            int[] B = new int[n1 * (n1 + 1)];
            int[] p = new int[n1 * 2];
            double propdel = 0;
            float static_carry = 0, switching_carry = 0, static_sum = 0, switching_sum = 0, AF_carry = 0,
AF_sum = 0;


            //////////////////////////////////////////////////////
            B[n1 * (n1 + 1) - 1] = 0;

            for (int i = 0; i < n1 + 1; i++)
            {
                for (int j = 0; j < n1; j++)
                {
                    if (i == n1 && j == 0)
                        A[(n1 * i) + j] = 0; //s[(n1*i)+j-n1];
                    else if (i == n1 && j > 0)
                        A[(n1 * i) + j] = cout[(n1 * i) + j - 1];
                    else
                        A[(n1 * i) + j] = m[j] * q[i];
                    if ((n1 * i) + j < n1)
                        B[(n1 * i) + j] = 0;
                    else if (((n1 * i) + j) % (((i + 1) * n1) - 1) != 0)
```

```
            B[(n1 * i) + j] = s[(n1 * i) + j - (n1 - 1)];
        if ((n1 * i) + j < n1)
            cin[(n1 * i) + j] = 0;
        else if ((n1 * i) + j > (n1 * i) + j - n1)
            cin[(n1 * i) + j] = cout[(n1 * i) + j - n1];
        else
            cin[(n1 * i) + j] = cin[i - 1];
        s[(n1 * i) + j] = A[(n1 * i) + j] ^ B[(n1 * i) + j] ^ cin[(n1 * i) + j];
        cout[(n1 * i) + j] = (A[(n1 * i) + j] * B[(n1 * i) + j]) + cin[(n1 * i) + j] * (A[(n1 * i) + j] ^ B[(n1 * i) + j]);
    }
}

for (int i = 1; i < n1 * (n1 + 1); i++)
{
    if (cout[i] == cout[i - 1])
        static_carry++;
    else
        switching_carry++;
    if (s[i] == s[i - 1])
        static_sum++;
    else
        switching_sum++;
}


AF_carry = switching_carry / ((n1 * (n1 + 1)) - 1);
AF_sum = switching_sum / ((n1 * (n1 + 1)) - 1);
int HS = n1 * (n1 - 1) * 9; //one fulladder has 9 nand gates
propdel = Convert.ToDouble(textBox3.Text) * 2 * (n1 - 1) * 3;


label11.Text = propdel.ToString("F2") + " ns";

label12.Text = HS.ToString() + " nand gates";
label13.Text = AF_carry.ToString("F2");
float st = static_carry * 100 / (static_carry + switching_carry);
label14.Text = st.ToString("F2");
float sw = switching_carry * 100 / (static_carry + switching_carry);
label15.Text = sw.ToString("F2");
label19.Text = AF_sum.ToString("F2");

float sts = static_sum * 100 / (static_sum + switching_sum);
label18.Text = sts.ToString("F2");
float sws = switching_sum * 100 / (static_sum + switching_sum);
label20.Text = sws.ToString("F2");

for (int i = 0; i < n1; i++)
{
    p[i] = s[n1 * i];
}

for (int i = 0; i < n1 - 1; i++)
{
    p[n1 + i] = s[n1 * n1 + i];
}
p[(2 * n1) - 1] = cout[(n1 * (n1 + 1)) - 1];

int[] array = new int[(n1 * 2) - 1];
label5.Text = "";

for (int num = (n1 * 2) - 1; num >= 0; num--)
{
```

```
            label5.Text += p[num].ToString();
        }

        double Edyn, Estat, EDP;


        Edyn = (Convert.ToDouble(textBox4.Text) *HS* AF_carry * Convert.ToDouble(textBox6.Text) *
Convert.ToDouble(textBox6.Text))/2;
        Estat = Convert.ToDouble(textBox5.Text) * Convert.ToDouble(textBox6.Text) * propdel * HS* (1 -
AF_carry);

        EDP = ((Edyn) + (Estat*0.00001)) * propdel*(0.000001);

        double Pow = EDP / propdel;

        label73.Text = Pow.ToString("F2") + " uW";
        label49.Text = EDP.ToString("F2") + " uJs";

        string z=label49.Text;
        if(z.Length >= 3)
        {
            z = z.Remove(z.Length - 4);
        }

        var builderm = new StringBuilder();
        Array.ForEach(m, x => builderm.Append(x));
        var resm = builderm.ToString();

        var builderq = new StringBuilder();
        Array.ForEach(q, x => builderq.Append(x));
        var resq = builderq.ToString();

        if (flag == 0)
        {
            DataGridViewRow row = (DataGridViewRow)dataGridView1.Rows[0].Clone();
            row.Cells[0].Value = resm + 'x' + resq;
            row.Cells[1].Value = n1.ToString();
            row.Cells[2].Value = z;
            dataGridView1.Rows.Add(row);
        }
        //dataGridView1.CurrentRow.Selected = true;

    }

    public void RCAM(int[] m, int[] q)
    {
        int[] cin = new int[n1 * (n1 - 1)];
        int[] cout = new int[n1 * (n1 - 1)];
        int[] s = new int[n1 * (n1 - 1)];
        int[] A = new int[n1 * (n1 - 1)];
        int[] B = new int[n1 * (n1 - 1)];
        int[] p = new int[n1 * 2];
        double propdel = 0;
        float static_carry = 0, switching_carry = 0, static_sum = 0, switching_sum = 0, AF_carry = 0,
AF_sum = 0;


        A[n1 - 1] = 0;
        cin[0] = 0;

        for (int i = 0; i < n1 - 1; i++)
        {
            for (int j = 0; j < n1; j++)
```

```
    {
        if (i == 0 && j != n1 - 1)

            A[(n1 * i) + j] = m[i] * q[j + 1];
        if (i == 0)
            B[(n1 * i) + j] = m[i + 1] * q[j];

        if (i > 0)
        {
            A[(n1 * i) + j] = s[((n1 * i) + j) - n1 + 1];
            B[(n1 * i) + j] = m[i + 1] * q[j];
        }
        if (i >= 0 && j != 0)
            cin[(n1 * i) + j] = cout[(n1 * i) + j - 1];

        if (i > 0 && j == n1 - 1)
            A[(n1 * i) + n1 - 1] = cout[(n1 * i) - 1];
        if (i >= 0 && j == 0)
            cin[(n1 * i) + j] = 0;



        s[(n1 * i) + j] = A[(n1 * i) + j] ^ B[(n1 * i) + j] ^ cin[(n1 * i) + j];
        cout[(n1 * i) + j] = (A[(n1 * i) + j] * B[(n1 * i) + j]) + cin[(n1 * i) + j] * (A[(n1 * i) + j] ^ B[(n1 * i) + j]);
    }
}
for (int i = 1; i < n1 * (n1 - 1); i++)
{
    if (cout[i] == cout[i - 1])
        static_carry++;
    else
        switching_carry++;
    if (s[i] == s[i - 1])
        static_sum++;
    else
        switching_sum++;
}


AF_carry = switching_carry / ((n1 * (n1 + 1)) - 1);
AF_sum = switching_sum / ((n1 * (n1 + 1)) - 1);
int HS = n1 * (n1 + 1) * 9;
propdel = Convert.ToDouble(textBox3.Text) * (2 * n1)* 3;


label67.Text = propdel.ToString("F2") + " ns";
label66.Text = HS.ToString() + " nand gates";
label65.Text = AF_carry.ToString("F2");
float st = static_carry * 100 / (static_carry + switching_carry);
label64.Text = st.ToString("F2");
float sw = switching_carry * 100 / (static_carry + switching_carry);
label63.Text = sw.ToString("F2");
label61.Text = AF_sum.ToString("F2");

float sts = static_sum * 100 / (static_sum + switching_sum);
label62.Text = sts.ToString("F2");
float sws = switching_sum * 100 / (static_sum + switching_sum);
label60.Text = sws.ToString("F2");

p[0] = q[0] * m[0];

for (int i = 1; i < n1; i++)
{
```

```
            p[i] = s[(n1 * (i - 1))];
        }

        for (int i = 0; i < n1 - 1; i++)
        {
            p[n1 + i] = s[n1 * (n1 - 2) + 1 + i];
        }
        p[(2 * n1) - 1] = cout[(n1 * (n1 - 1)) - 1];

        int[] array = new int[(n1 * 2) - 1];
        label5.Text = "";

        for (int num = (n1 * 2) - 1; num >= 0; num--)
        {

            label5.Text += p[num].ToString();
        }

        double Edyn, Estat, EDP;


        Edyn = (Convert.ToDouble(textBox4.Text) * HS * AF_carry * Convert.ToDouble(textBox6.Text) *
Convert.ToDouble(textBox6.Text))/2;
        Estat = Convert.ToDouble(textBox5.Text) * Convert.ToDouble(textBox6.Text) * propdel * HS * (1-
AF_carry);

        EDP = ((Edyn) + (Estat * 0.00001)) * propdel * (0.000001);

        double Pow = EDP / propdel;

        label79.Text = Pow.ToString("F2") + " uW";
        label51.Text = EDP.ToString("F2") + " uJs";

        DataGridViewRow row1 = (DataGridViewRow)dataGridView1.Rows[0].Clone();
        row1.Cells[2].Value = label51.Text;
        dataGridView1.Rows.Add(row1);
        string y=label51.Text;
        if(y.Length >= 3)
        {
            y = y.Remove(y.Length - 4);
        }
        if (flag == 0)
        {
            dataGridView1.CurrentRow.Cells[3].Value = y;
        }
    }


    public void WTM(int[] m, int[] q)
    {

        int[] cin = new int[n1 * n1];
        int[] cout = new int[n1 * (n1 - 1)];
        int[] s = new int[n1 * (n1 - 1)];
        int[] A = new int[n1 * (n1 - 1)];
        int[] B = new int[n1 * (n1 - 1)];
        int[] p = new int[n1 * 2];
        double propdel = 0;
        float static_carry = 0, switching_carry = 0, static_sum = 0, switching_sum = 0, AF_carry = 0,
AF_sum = 0;
        double d = 2.0;
        double lg = Math.Log(n1, d);
        int loop_numb = (int)Math.Round(lg, MidpointRounding.AwayFromZero) + 1;
```

```
        int[] first_stage = new int[n1 + 1];
        first_stage[0] = 2;
        int[] numb_FA = new int[n1 + 1];
        numb_FA[0] = 2;
        int delay = (int)(Math.Log(n1, d));
        for (int i = 1; i < n1; i++)
        {
            numb_FA[i] = i + 1 - ((int)(Math.Log(i + 1, d)) + 1);
            first_stage[i] = first_stage[i - 1] + numb_FA[i] + numb_FA[i - 1];
        }

        A[n1 - 1] = 0;
        cin[0] = 0;

        for (int i = 0; i < n1 - 1; i++)
        {
            for (int j = 0; j < n1; j++)
            {
                if (i == 0 && j != n1 - 1)

                    A[(n1 * i) + j] = m[i] * q[j + 1];
                if (i == 0)
                    B[(n1 * i) + j] = m[i + 1] * q[j];

                if (i > 0)
                {
                    A[(n1 * i) + j] = s[((n1 * i) + j) - n1 + 1];
                    B[(n1 * i) + j] = m[i + 1] * q[j];
                }
                if (i >= 0 && j != 0)
                    cin[(n1 * i) + j] = cout[(n1 * i) + j - 1];

                if (i > 0 && j == n1 - 1)
                    A[(n1 * i) + n1 - 1] = cout[(n1 * i) - 1];
                if (i >= 0 && j == 0)
                    cin[(n1 * i) + j] = 0;



                s[(n1 * i) + j] = A[(n1 * i) + j] ^ B[(n1 * i) + j] ^ cin[(n1 * i) + j];
                cout[(n1 * i) + j] = (A[(n1 * i) + j] * B[(n1 * i) + j]) + cin[(n1 * i) + j] * (A[(n1 * i) + j] ^ B[(n1 * i) + j]);
            }
        }
        for (int i = 1; i < n1 * (n1 - 1); i++)
        {
            if (cout[i] == cout[i - 1])
                switching+0.5;
            if (s[i] == s[i - 1])
                switching +0.5;
        }


        AF_carry = switching_carry / ((n1 * (n1 + 1)) - 1);
        AF_sum = switching_sum / ((n1 * (n1 + 1)) - 1);
        int HS = (int)(first_stage[n1 - 2] + ((Math.Round(lg, MidpointRounding.AwayFromZero) + 1) * (n1 -
5))) * 9;
        propdel = (int)Convert.ToDouble(textBox3.Text) * (Math.Log(n1, d) + 1) * (2 * Math.Log(n1, d) - 2) *
3;


        label90.Text = propdel.ToString("F2") + " ns";
        label89.Text = HS.ToString() + " nand gates";
        label88.Text = delay.ToString("F2");
```

```
            float st = static_carry * 100 / (static_carry + switching_carry);
            label87.Text = loop_numb.ToString("F2");
            float sw = switching_carry * 100 / (static_carry + switching_carry);
            label86.Text = sw.ToString("F2");
            label84.Text = AF_sum.ToString("F2");

            float sts = static_sum * 100 / (static_sum + switching_sum);
            label85.Text = sts.ToString("F2");
            float sws = switching_sum * 100 / (static_sum + switching_sum);
            label83.Text = sws.ToString("F2");

            p[0] = q[0] * m[0];

            for (int i = 1; i < stage_num; i++)
            {
                p[i] = s[(n1 * (i - 1))];
            }

            for (int i = 0; i < stage_num - 1; i++)
            {
                p[n1 + i] = s[n1 * (n1 - 2) + 1 + i];
            }
            p[(2 * n1) - 1] = cout[(n1 * (n1 - 1)) - 1];

            int[] array = new int[(n1 * 2) - 1];
            label5.Text = "";

            for (int num = (n1 * 2) - 1; num >= 0; num--)
            {

                label5.Text += p[num].ToString();
            }

            double Edyn, Estat, EDP;


            Edyn = (Convert.ToDouble(textBox4.Text) * HS * 0.5 * Convert.ToDouble(textBox6.Text) *
Convert.ToDouble(textBox6.Text)) / 2;
            Estat = Convert.ToDouble(textBox5.Text) * Convert.ToDouble(textBox6.Text) * propdel * HS * (1 -
0.5);

            EDP = ((Edyn) + (Estat * 0.00001)) * propdel * (0.000001);

            double Pow = EDP / propdel;

            label75.Text = Pow.ToString("F2") + " uW";
            label78.Text = EDP.ToString("F2") + " uJs";

            //DataGridViewRow row1 = (DataGridViewRow)dataGridView1.Rows[0].Clone();
            //row1.Cells[2].Value = label75.Text;
            //dataGridView1.Rows.Add(row1);
            string y = label78.Text;
            if (y.Length >= 3)
            {
                y = y.Remove(y.Length - 4);
            }
            if (flag == 0)
            {
                dataGridView1.CurrentRow.Cells[4].Value = y;
            }
        }

        public void ABACUS(int[] m, int[] q)
```

```
        {

            int[] cin = new int[n1 * n1];
            int[] cout = new int[n1 * (n1 - 1)];
            int[] s = new int[n1 * (n1 - 1)];
            int[] A = new int[n1 * (n1 - 1)];
            int[] B = new int[n1 * (n1 - 1)];
            int[] p = new int[n1 * 2];
            double propdel = 0;
            float static_carry = 0, switching_carry = 0, static_sum = 0, switching_sum = 0, AF_carry = 0,
AF_sum = 0;
            double d = 2.0;
            double lg = Math.Log(n1, d);
            int loop_numb = (int)Math.Round(lg, MidpointRounding.AwayFromZero) + 1;
            int[] first_stage = new int[n1 + 1];
            first_stage[0] = 2;
            int[] numb_FA = new int[n1 + 1];
            numb_FA[0] = 2;
            int delay = (int)(2 * Math.Log(n1, d) - 1);
            for (int i = 1; i < n1; i++)
            {
                numb_FA[i] = i + 1 - ((int)(Math.Log(i + 1, d)) + 1);
                first_stage[i] = first_stage[i - 1] + numb_FA[i] + numb_FA[i - 1];
            }

            A[n1 - 1] = 0;
            cin[0] = 0;

            for (int i = 0; i < n1 - 1; i++)
            {
                for (int j = 0; j < n1; j++)
                {
                    if (i == 0 && j != n1 - 1)

                        A[(n1 * i) + j] = m[i] * q[j + 1];
                    if (i == 0)
                        B[(n1 * i) + j] = m[i + 1] * q[j];

                    if (i > 0)
                    {
                        A[(n1 * i) + j] = s[((n1 * i) + j) - n1 + 1];
                        B[(n1 * i) + j] = m[i + 1] * q[j];
                    }
                    if (i >= 0 && j != 0)
                        cin[(n1 * i) + j] = cout[(n1 * i) + j - 1];

                    if (i > 0 && j == n1 - 1)
                        A[(n1 * i) + n1 - 1] = cout[(n1 * i) - 1];
                    if (i >= 0 && j == 0)
                        cin[(n1 * i) + j] = 0;



                    s[(n1 * i) + j] = A[(n1 * i) + j] ^ B[(n1 * i) + j] ^ cin[(n1 * i) + j];
                    cout[(n1 * i) + j] = (A[(n1 * i) + j] * B[(n1 * i) + j]) + cin[(n1 * i) + j] * (A[(n1 * i) + j] ^ B[(n1 * i) + j]);
                }
            }
            for (int i = 1; i < n1 * (n1 - 1); i++)
            {
                if (cout[i] == cout[i - 1])
                    switching+0.5;
                if (s[i] == s[i - 1])
                    switching +0.5;
```

```
        }

        AF_carry = switching_carry / ((n1 * (n1 + 1)) - 1);
        AF_sum = switching_sum / ((n1 * (n1 + 1)) - 1);
        int HS = (int)(first_stage[n1 - 2] + ((Math.Round(lg, MidpointRounding.AwayFromZero) + 1) * (n1 -
2))) * 9;
        propdel = (int)Convert.ToDouble(textBox3.Text) * (Math.Log(n1, d) + 1) * (2 * Math.Log(n1, d) - 1) *
3;


        label35.Text = propdel.ToString("F2") + " ns";
        label34.Text = HS.ToString() + " nand gates";
        label33.Text = delay.ToString("F2");
        float st = static_carry * 100 / (static_carry + switching_carry);
        label32.Text = loop_numb.ToString("F2");
        float sw = switching_carry * 100 / (static_carry + switching_carry);
        label32.Text = sw.ToString("F2");
        label29.Text = AF_sum.ToString("F2");

        float sts = static_sum * 100 / (static_sum + switching_sum);
        label30.Text = sts.ToString("F2");
        float sws = switching_sum * 100 / (static_sum + switching_sum);
        label28.Text = sws.ToString("F2");

        p[0] = q[0] * m[0];

        for (int i = 1; i < stage_num; i++)
        {
            p[i] = s[(n1 * (i - 1))];
        }

        for (int i = 0; i < stage_num - 1; i++)
        {
            p[n1 + i] = s[n1 * (n1 - 2) + 1 + i];
        }
        p[(2 * n1) - 1] = cout[(n1 * (n1 - 1)) - 1];

        int[] array = new int[(n1 * 2) - 1];
        label5.Text = "";

        for (int num = (n1 * 2) - 1; num >= 0; num--)
        {

            label5.Text += p[num].ToString();
        }

        double Edyn, Estat, EDP;


        Edyn = (Convert.ToDouble(textBox4.Text) * HS * 0.5 * Convert.ToDouble(textBox6.Text) *
Convert.ToDouble(textBox6.Text)) / 2;
        Estat = Convert.ToDouble(textBox5.Text) * Convert.ToDouble(textBox6.Text) * propdel * HS * (1 -
0.5);

        EDP = ((Edyn) + (Estat * 0.00001)) * propdel * (0.000001);

        double Pow = EDP / propdel;

        label55.Text = Pow.ToString("F2") + " uW";
        label53.Text = EDP.ToString("F2") + " uJs";

        //DataGridViewRow row1 = (DataGridViewRow)dataGridView1.Rows[0].Clone();
        //row1.Cells[2].Value = label51.Text;
```

```
    //dataGridView1.Rows.Add(row1);
    string y = label53.Text;
    if (y.Length >= 3)
    {
        y = y.Remove(y.Length - 4);
    }
    if (flag == 0)
    {
        dataGridView1.CurrentRow.Cells[5].Value = y;
    }
}


private void Form1_Load(object sender, EventArgs e)
{
    this.dataGridView1.Columns.Add("Multiplication", "Multiplication");
    this.dataGridView1.Columns.Add("Bits", "Bits");
    this.dataGridView1.Columns.Add("CSAM", "CSAM EnergyxDelay");
    this.dataGridView1.Columns.Add("RCAM", "RCAM EnergyxDelay");
    this.dataGridView1.Columns.Add("WTM", "WTM EnergyxDelay");
    this.dataGridView1.Columns.Add("ABACUS", "ABACUS EnergyxDelay");
    DataGridViewColumn column = dataGridView1.Columns[0];
    column.Width = 260;
    DataGridViewColumn column2 = dataGridView1.Columns[1];
    column2.Width = 45;
}

private void label2_Click(object sender, EventArgs e)
{

}

private void textBox3_TextChanged(object sender, EventArgs e)
{

}

private void label5_Click(object sender, EventArgs e)
{

}

private void textBox1_TextChanged(object sender, EventArgs e)
{
    n1 = textBox1.Text.Length;
    label16.Text = n1.ToString();
}

private void textBox2_TextChanged(object sender, EventArgs e)
{
    n2 = textBox2.Text.Length;
    label16.Text = n1.ToString() + " x " + n2.ToString();
}

private void button1_Click(object sender, EventArgs e)
{

    string mi = textBox1.Text;
    int[] m = new int[mi.Length];
    for (int i = mi.Length - 1; i >= 0; --i)
    {
        m[mi.Length - 1 - i] = Convert.ToInt32(mi[i].ToString());
    }
```

```
            string qi = textBox2.Text;
            int[] q = new int[qi.Length];
            for (int i = qi.Length - 1; i >= 0; --i)
            {
                q[qi.Length - 1 - i] = Convert.ToInt32(qi[i].ToString());
            }
            //MessageBox.Show(qi.ToString());
            int bm = Convert.ToInt32(mi, 2);
            int qm = Convert.ToInt32(qi, 2);



            if (comboBox1.SelectedItem.ToString() == "Carry-Save Multiplier")
            {

                CSAM(m,q);
                int ans = bm * qm;
                //string ansi = Convert.ToString(ans, 2).PadLeft(2 * n1);

                if (Convert.ToInt32(label5.Text,2) != ans)
                    MessageBox.Show("WRONG ANSWER!!");
                comboBox1.SelectedItem = "Ripple Carry Multiplier";

                count++;

            }
            if (comboBox1.SelectedItem.ToString() == "Ripple Carry Multiplier")
            {
                dataGridView1.CurrentCell = dataGridView1.Rows[count-1].Cells[2];
                RCAM(m,q);


                int ans = bm * qm;
                //string ansi = Convert.ToString(ans, 2).PadLeft(2 * n1);

                if (Convert.ToInt32(label5.Text, 2) != ans)
                    MessageBox.Show("WRONG ANSWER!!");

            }
            if (count > 1)
            {
                if (dataGridView1.Rows[count-1].Cells[2].Value == dataGridView1.Rows[count -
2].Cells[2].Value)
                {
                    dataGridView1.Rows[count-1].Cells[3].Value =
(Convert.ToDouble(dataGridView1.Rows[count-1].Cells[3].Value) +
Convert.ToDouble(dataGridView1.Rows[count - 2].Cells[3].Value))/2;
                    dataGridView1.Rows[count-1].Cells[4].Value =
(Convert.ToDouble(dataGridView1.Rows[count-1].Cells[4].Value) +
Convert.ToDouble(dataGridView1.Rows[count - 2].Cells[4].Value))/2;
                    dataGridView1.Rows[count - 1].Cells[5].Value =
(Convert.ToDouble(dataGridView1.Rows[count - 1].Cells[4].Value) +
Convert.ToDouble(dataGridView1.Rows[count - 2].Cells[5].Value)) / 2;
                    dataGridView1.Rows.Remove(dataGridView1.Rows[count - 2]);
                    count--;
                }
            }
            }


        private void button2_Click(object sender, EventArgs e)
        {
            int irow;
```

```
        for (irow = 0; irow < dataGridView1.Rows.Count-1; irow++)
        {
            chart1.Series["CSAM"].Points.AddXY
                        (dataGridView1.Rows[irow].Cells[1].Value,
System.Convert.ToDouble(dataGridView1.Rows[irow].Cells["CSAM"].Value));
            chart1.Series["RCAM"].Points.AddXY
                        (dataGridView1.Rows[irow].Cells[2].Value,
System.Convert.ToDouble(dataGridView1.Rows[irow].Cells["RCAM"].Value));
            chart1.Series["WTM"].Points.AddXY
                        (dataGridView1.Rows[irow].Cells[3].Value,
System.Convert.ToDouble(dataGridView1.Rows[irow].Cells["WTM"].Value));
            chart1.Series["ABACUS"].Points.AddXY
                        (dataGridView1.Rows[irow].Cells[4].Value,
System.Convert.ToDouble(dataGridView1.Rows[irow].Cells["ABACUS"].Value));
        }


        chart1.Series["CSAM"].ChartType =
                        SeriesChartType.FastLine;
        //chart1.Series["CSAM"].Color = Color.Chartreuse;
        chart1.Series["RCAM"].ChartType =
              SeriesChartType.FastLine;
        //chart1.Series["RCAM"].Color = Color.Blue;
        chart1.Series["WTM"].ChartType =
              SeriesChartType.FastLine;
        //chart1.Series["WTM"].Color = Color.Red;
        chart1.Series["ABACUS"].ChartType =
              SeriesChartType.FastLine;
        //chart1.Series["ABACUS"].Color = Color.DarkOrange;
    }

    private void button3_Click(object sender, EventArgs e)
    {


    /// <summary>
    /// Converts a given delimited file into a dataset.
    /// Assumes that the first line
    /// of the text file contains the column names.
    /// </summary>
    /// <param name="File">The name of the file to open</param>
    /// <param name="TableName">The name of the
    /// Table to be made within the DataSet returned</param>
    /// <param name="delimiter">The string to delimit by</param>
    /// <returns></returns>
        string File = "C:\\Users\\user\\Documents\\MS\\mult.txt"; string TableName = "MyNewTable"; string
delimiter = "\x2c";
    {

        //The DataSet to Return
        DataSet result = new DataSet();

        //Open the file in a stream reader.
        StreamReader s = new StreamReader(File);

        //Split the first line into the columns
        string[] columns = s.ReadLine().Split(delimiter.ToCharArray());

        //Add the new DataTable to the RecordSet
        result.Tables.Add(TableName);

        //Cycle the colums, adding those that don't exist yet
        //and sequencing the one that do.
```

```
foreach (string col in columns)
{
    bool added = false;
    string next = "";
    int i = 0;
    while (!added)
    {
        //Build the column name and remove any unwanted characters.
        string columnname = col + next;
        columnname = columnname.Replace("#", "");
        columnname = columnname.Replace("'", "");
        columnname = columnname.Replace("&", "");

        //See if the column already exists
        if (!result.Tables[TableName].Columns.Contains(columnname))
        {
            //if it doesn't then we add it here and mark it as added
            result.Tables[TableName].Columns.Add(columnname);
            added = true;
        }
        else
        {
            //if it did exist then we increment the sequencer and try again.
            i++;
            next = "_" + i.ToString();
        }
    }
}

//Read the rest of the data in the file.
string AllData = s.ReadToEnd();

//Split off each row at the Carriage Return/Line Feed
//Default line ending in most windows exports.
//You may have to edit this to match your particular file.
//This will work for Excel, Access, etc. default exports.
string[] rows = AllData.Split("\r\n".ToCharArray());

//Now add each row to the DataSet
foreach (string r in rows)
{
    //Split the row at the delimiter.
    string[] items = r.Split(delimiter.ToCharArray());

    //Add the item
    if (items[0]!=""&&items[1]!="")
    result.Tables[TableName].Rows.Add(items);
}

//Return the imported data.
DataSet ds = result;
int p;
for (p = 0; p < result.Tables[TableName].Rows.Count - 1; p++)
{
    //dataGridView1.Rows.Add();
    string length = ds.Tables["MyNewTable"].Rows[p].ItemArray[0].ToString();
    n1 = length.Length;

    string mi = ds.Tables["MyNewTable"].Rows[p].ItemArray[0].ToString();
    int[] m = new int[mi.Length];
    for (int i = mi.Length - 1; i >= 0; --i)
    {
        m[mi.Length - 1 - i] = Convert.ToInt32(mi[i].ToString());
```

```
            }
            string qi = ds.Tables["MyNewTable"].Rows[p].ItemArray[1].ToString();
            int[] q = new int[qi.Length];
            for (int i = qi.Length - 1; i >= 0; --i)
            {
                q[qi.Length - 1 - i] = Convert.ToInt32(qi[i].ToString());
            }

            CSAM(m,q);
            dataGridView1.CurrentCell = dataGridView1.Rows[p].Cells[2];
            RCAM(m,q);
            dataGridView1.CurrentCell = dataGridView1.Rows[p].Cells[3];//bakmak lazim
            WTM(m, q);
            dataGridView1.CurrentCell = dataGridView1.Rows[p].Cells[4];//bakmak lazim
            ABACUS(m, q);
            dataGridView1.CurrentCell = dataGridView1.Rows[p].Cells[5];
            if (p != 0)
            {
                if (dataGridView1.Rows[p].Cells[2].Value == dataGridView1.Rows[p - 1].Cells[2].Value)
                {
                    dataGridView1.Rows[p].Cells[3].Value =
(Convert.ToDouble(dataGridView1.Rows[p].Cells[3].Value) + Convert.ToDouble(dataGridView1.Rows[p -
1].Cells[3].Value))/2;
                    dataGridView1.Rows[p].Cells[4].Value =
(Convert.ToDouble(dataGridView1.Rows[p].Cells[4].Value) + Convert.ToDouble(dataGridView1.Rows[p -
1].Cells[4].Value))/2;
                    dataGridView1.Rows[p].Cells[5].Value =
(Convert.ToDouble(dataGridView1.Rows[p].Cells[5].Value) + Convert.ToDouble(dataGridView1.Rows[p -
1].Cells[5].Value)) / 2;
                    dataGridView1.Rows.Remove(dataGridView1.Rows[p - 1]);
                }
            }


            /*dataGridView1.Rows[p].Cells[0].Value = ds.Tables["MyNewTable"].Rows[p].ItemArray[0] + "x" +
ds.Tables["MyNewTable"].Rows[p].ItemArray[1];
            string length = ds.Tables["MyNewTable"].Rows[p].ItemArray[0].ToString();
            dataGridView1.Rows[p].Cells[1].Value = length.Length;

            dataGridView1.DataSource = new BindingSource(result.Tables["MyNewTable"], null);*/
        }
    }
    }

    private void dataGridView1_CellContentClick(object sender, DataGridViewCellEventArgs e)
    {
        string mi1 = dataGridView1.SelectedRows[0].Cells[0].Value.ToString();

        string[] mi2 = mi1.Split('x');

        string mi = mi2[0];
        int[] m = new int[mi.Length];
        for (int i = mi.Length - 1; i >= 0; --i)
        {
            m[mi.Length - 1 - i] = Convert.ToInt32(mi[i].ToString());
        }
        string qi = mi2[1];
        int[] q = new int[qi.Length];
        for (int i = qi.Length - 1; i >= 0; --i)
        {
            q[qi.Length - 1 - i] = Convert.ToInt32(qi[i].ToString());
        }
```

```
            if (mi.Length != qi.Length)
                MessageBox.Show("SIZE PROBLEM");
            n1 = mi.Length;
                flag = 1;
            Array.Reverse(m);
            Array.Reverse(q);
                CSAM(m, q);
                RCAM(m, q);
                ABACUS(m, q);
                WTM(m, q);
                flag = 0;
        }

    }
}
```

APPENDIX C

C. EXPERIMENT CIRCUIT

The experimental circuit is shown in Figure C.1, 2 HD74HC00 chips used and 4 NAND gates connected to each other for delay, static current, dynamic current and capacitance measurements.

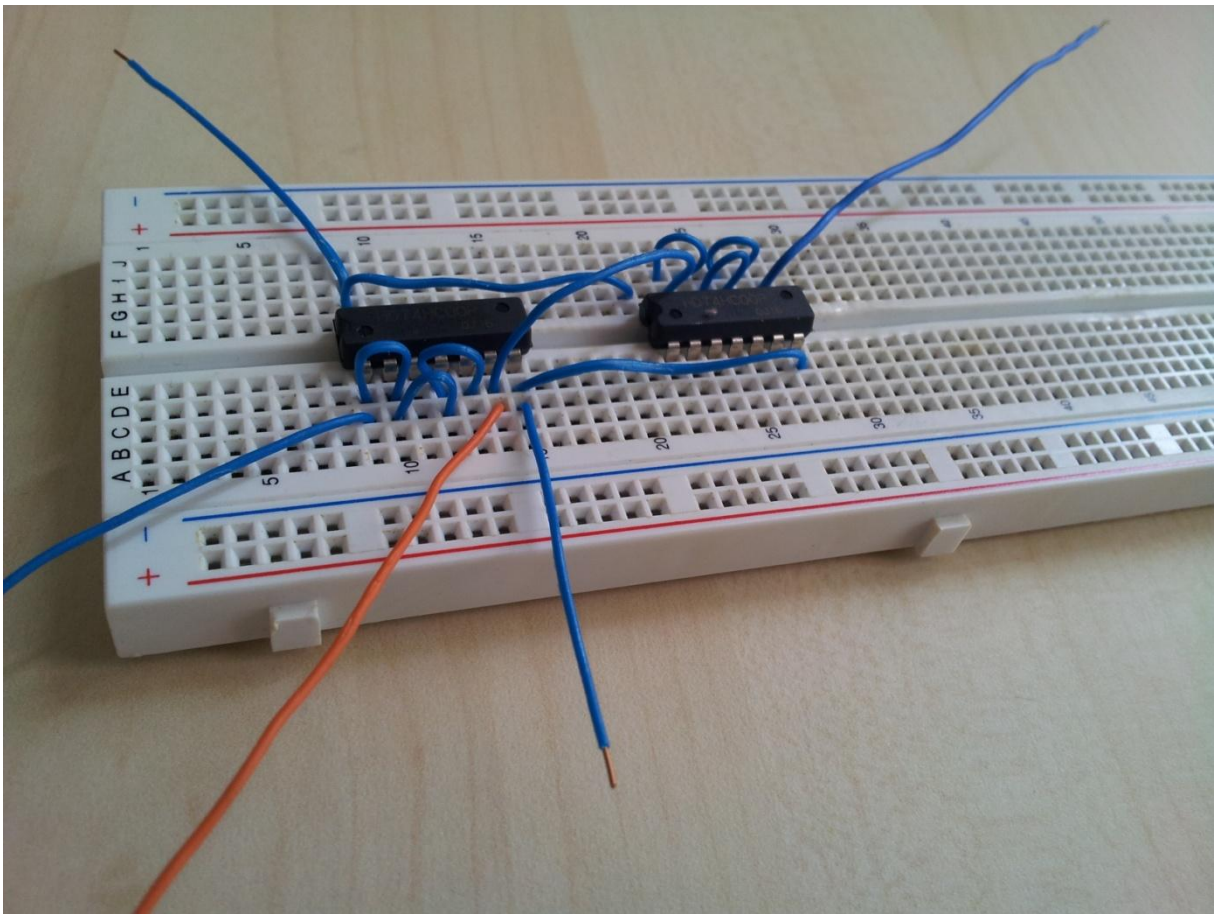


Figure C. 1 Experimental Circuit.

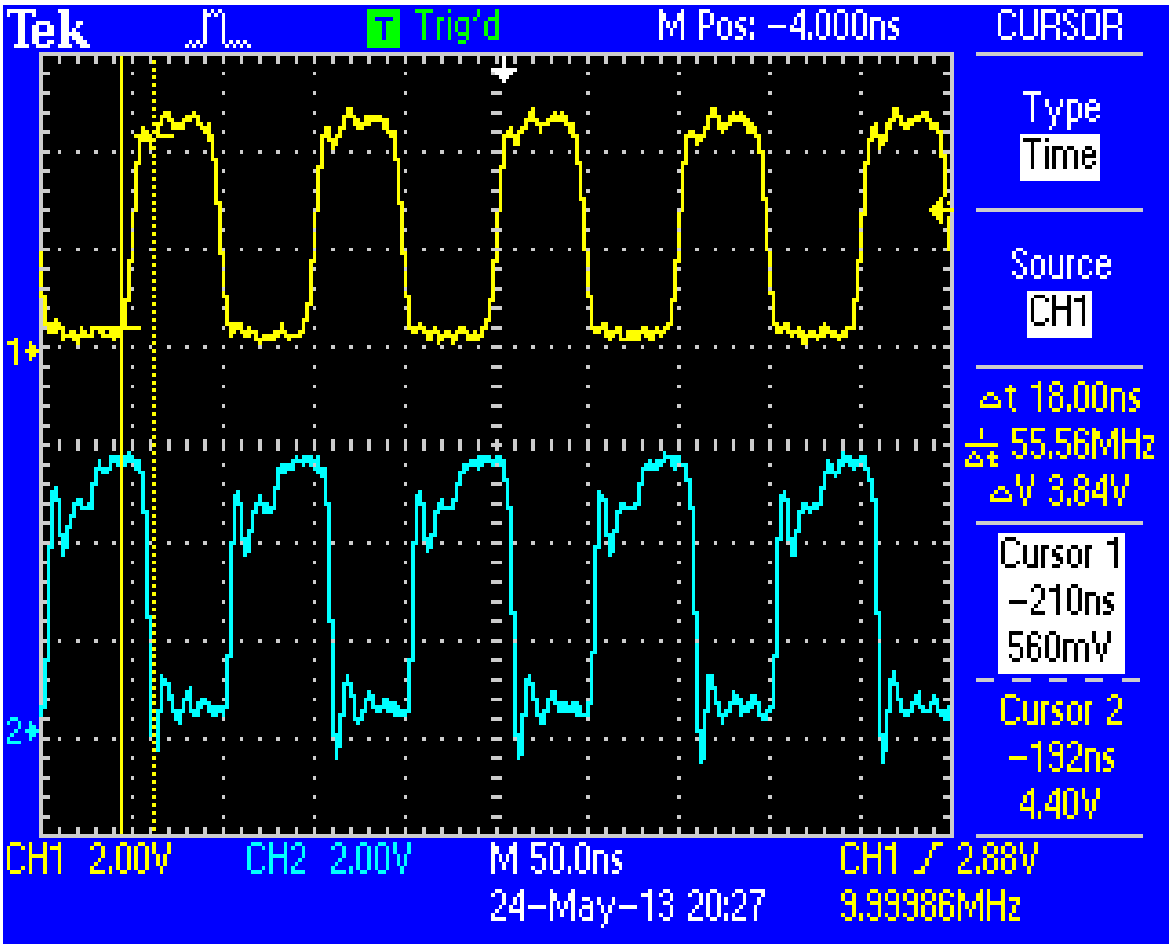


Figure C. 2 Delay measurement.

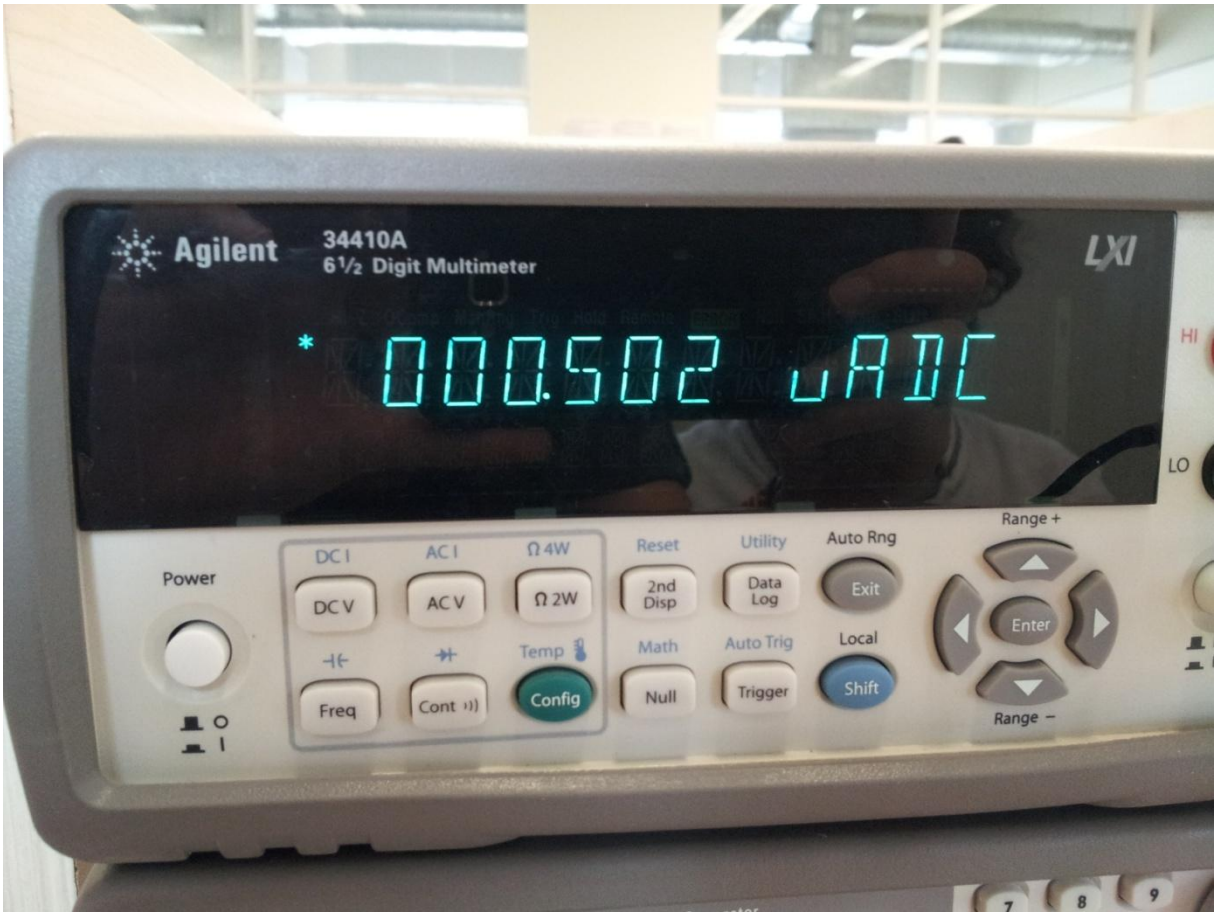


Figure C. 3 Static current measurement.

Figure C. 4 Dynamic current measurement.

APPENDIX D

D. MULT.TXT FILES

1. Chess Board

The chess board file contains multiplier and multiplicand of each multiplication. The left side of comma symbol (,) is multiplicand and the right side of it is multiplier. The txt file contains 64 multiplications in 1 bit x 1 bit to 64 bit x 64 bit.

1,0

01,10

101,010

0101,1010

10101,01010

010101,101010

1010101,0101010

01010101,10101010

101010101,010101010

0101010101,1010101010

10101010101,01010101010

010101010101,101010101010

1010101010101,0101010101010

01010101010101,10101010101010

101010101010101,010101010101010

0101010101010101,1010101010101010

10101010101010101,01010101010101010

010101010101010101,101010101010101010

1010101010101010101,0101010101010101010

01010101010101010101,10101010101010101010

101010101010101010101,010101010101010101010

0101010101010101010101,1010101010101010101010

10101010101010101010101,01010101010101010101010

010101010101010101010101,10101010101010101010101010

1010101010101010101010101,0101010101010101010101010

01010101010101010101010101,1010101010101010101010101010

10101010101010101010101010101,01010101010101010101010101010

010101010101010101010101010101,101010101010101010101010101010

1010101010101010101010101010101,0101010101010101010101010101010

01010101010101010101010101010101,10101010101010101010101010101010

101010101010101010101010101010101,010101010101010101010101010101010

0101010101010101010101010101010101,1010101010101010101010101010101010

10101010101010101010101010101010101,01010101010101010101010101010101010

010101010101010101010101010101010101,101010101010101010101010101010101010

1010101010101010101010101010101010101,0101010101010101010101010101010101010

01010101010101010101010101010101010101,10101010101010101010101010101010101010

101010101010101010101010101010101010101,010101010101010101010101010101010101010

0101010101010101010101010101010101010101,1010101010101010101010101010101010101010

10101010101010101010101010101010101010101,01010101010101010101010101010101010101010

010101010101010101010101010101010101010101,101010101010101010101010101010101010101010

1010101010101010101010101010101010101010101,0101010101010101010101010101010101010101010

01010101010101010101010101010101010101010101,10101010101010101010101010101010101010101010

101010101010101010101010101010101010101010101,010101010101010101010101010101010101010101010

0101010101010101010101010101010101010101010101,1010101010101010101010101010101010101010101010

10101010101010101010101010101010101010101010101,01010101010101010101010101010101010101010101010

010101010101010101010101010101010101010101010101,101010101010101010101010101010101010101010101010

1010101010101010101010101010101010101010101010101,0101010101010101010101010101010101010101010101010

01010101010101010101010101010101010101010101010101,10101010101010101010101010101010101010101010101010

101010101010101010101010101010101010101010101010101,010101010101010101010101010101010101010101010101010

0101010101010101010101010101010101010101010101010101,1010101010101010101010101010101010101010101010101010

10101010101010101010101010101010101010101010101010101,01010101010101010101010101010101010101010101010101010

010101010101010101010101010101010101010101010101010101,101010101010101010101010101010101010101010101010101010

1010101010101010101010101010101010101010101010101010101,0101010101010101010101010101010101010101010101010101010

01010101010101010101010101010101010101010101010101010101,10101010101010101010101010101010101010101010101010101010

101010101010101010101010101010101010101010101010101010101,010101010101010101010101010101010101010101010101010101010

0101010101010101010101010101010101010101010101010101010101,1010101010101010101010101010101010101010101010101010101010

10101010101010101010101010101010101010101010101010101010101,01010101010101010101010101010101010101010101010101010101010

010101010101010101010101010101010101010101010101010101010101,101010101010101010101010101010101010101010101010101010101010

2. ALL 1's

1,1

11,11

111,111

1111,1111

11111,11111

111111,111111

1111111,1111111

11111111,11111111

111111111,111111111

1111111111,1111111111

11111111111,11111111111

111111111111,111111111111

1111111111111,1111111111111

11111111111111,11111111111111

111111111111111,111111111111111

1111111111111111,1111111111111111

11111111111111111,11111111111111111

111111111111111111,111111111111111111

1111111111111111111,1111111111111111111

11111111111111111111,11111111111111111111

111111111111111111111,111111111111111111111

1111111111111111111111,1111111111111111111111

11111111111111111111111,11111111111111111111111

111111111111111111111111,111111111111111111111111

1111111111111111111111111,1111111111111111111111111

11111111111111111111111111,11111111111111111111111111

111111111111111111111111111,111111111111111111111111111

1111111111111111111111111111,1111111111111111111111111111

11111111111111111111111111111,11111111111111111111111111111

111111111111111111111111111111,111111111111111111111111111111

1111111111111111111111111111111,1111111111111111111111111111111

11111111111111111111111111111111,11111111111111111111111111111111

111111111111111111111111111111111111,111111111111111111111111111111111111

1111111111111111111111111111111111111,1111111111111111111111111111111111111

11111111111111111111111111111111111111,11111111111111111111111111111111111111

111111111111111111111111111111111111111,111111111111111111111111111111111111111

1111111111111111111111111111111111111111,1111111111111111111111111111111111111111

11111111111111111111111111111111111111111,11111111111111111111111111111111111111111

111111111111111111111111111111111111111111,111111111111111111111111111111111111111111

1111111111111111111111111111111111111111111,1111111111111111111111111111111111111111111

11111111111111111111111111111111111111111111,11111111111111111111111111111111111111111111

111111111111111111111111111111111111111111111,111111111111111111111111111111111111111111111

1111111111111111111111111111111111111111111111,1111111111111111111111111111111111111111111111

11111111111111111111111111111111111111111111111,11111111111111111111111111111111111111111111111

111111111111111111111111111111111111111111111111,111111111111111111111111111111111111111111111111

1111111111111111111111111111111111111111111111111,1111111111111111111111111111111111111111111111111

11111111111111111111111111111111111111111111111111,11111111111111111111111111111111111111111111111111

111111111111111111111111111111111111111111111111111,111111111111111111111111111111111111111111111111111

1111111111111111111111111111111111111111111111111111,1111111111111111111111111111111111111111111111111111

11111111111111111111111111111111111111111111111111111,11111111111111111111111111111111111111111111111111111

111111111111111111111111111111111111111111111111111111,111111111111111111111111111111111111111111111111111111

1111111111111111111111111111111111111111111111111111111,1111111111111111111111111111111111111111111111111111111

11111111111111111111111111111111111111111111111111111111,11111111111111111111111111111111111111111111111111111111

111111111111111111111111111111111111111111111111111111111,111111111111111111111111111111111111111111111111111111111

1111111111111111111111111111111111111111111111111111111111,1111111111111111111111111111111111111111111111111111111111

3. 2^{nd} Type Chessboard

0,0

00,00

001,001

0011,0011

00110,00110

001100,001100

0011001,0011001

00110011,00110011

001100110,001100110

0011001100,0011001100

00110011001,00110011001

001100110011,001100110011

0011001100110,0011001100110

00110011001100,00110011001100

001100110011001,001100110011001

0011001100110011,0011001100110011

00110011001100110,00110011001100110

001100110011001100,001100110011001100

0011001100110011001,0011001100110011001

00110011001100110011,0011001100110011001100110011

001100110011001100110,001100110011001100110011001100110

0011001100110011001100,0011001100110011001100110011001100

00110011001100110011001100110,0011001100110011001100110011001100110

001100110011001100110011001100110011,0011001100110011001100110011001100110011

0011001100110011001100110011001100110011001,0011001100110011001100110011001100110011001100110011

00110011001100110011001100110011001100110011001100110011,0011001100110011001100110011001100110011001100110011001100110011

001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011

0011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011

00110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011001100110011

0011001100110011001100110011001100110011001,0011001100110011001100110011001100110011001

00110011001100110011001100110011001100110011,00110011001100110011001100110011001100110011

001100110011001100110011001100110011001100110,001100110011001100110011001100110011001100110

0011001100110011001100110011001100110011001100,0011001100110011001100110011001100110011001100

00110011001100110011001100110011001100110011001,00110011001100110011001100110011001100110011001

001100110011001100110011001100110011001100110011,001100110011001100110011001100110011001100110011

0011001100110011001100110011001100110011001100110,0011001100110011001100110011001100110011001100110

00110011001100110011001100110011001100110011001100,00110011001100110011001100110011001100110011001100

001100110011001100110011001100110011001100110011001,0011001100110011001100110011001100110011001100110
01

0011001100110011001100110011001100110011001100110011,001100110011001100110011001100110011001100110011
0011

00110011001100110011001100110011001100110011001100110,00110011001100110011001100110011001100110011001
100110

001100110011001100110011001100110011001100110011001100,0011001100110011001100110011001100110011001100
11001100

0011001100110011001100110011001100110011001100110011001,001100110011001100110011001100110011001100110
0110011001

00110011001100110011001100110011001100110011001100110011,00110011001100110011001100110011001100110011
001100110011

001100110011001100110011001100110011001100110011001100110,0011001100110011001100110011001100110011001
10011001100110

0011001100110011001100110011001100110011001100110011001100,001100110011001100110011001100110011001100
1100110011001100

00110011001100110011001100110011001100110011001100110011001,00110011001100110011001100110011001100110
011001100110011001

001100110011001100110011001100110011001100110011001100110011,0011001100110011001100110011001100110011
00110011001100110011

0011001100110011001100110011001100110011001100110011001100110,001100110011001100110011001100110011001
1001100110011001100110

00110011001100110011001100110011001100110011001100110011001100,00110011001100110011001100110011001100
110011001100110011001100

001100110011001100110011001100110011001100110011001100110011001,0011001100110011001100110011001100110
01100110011001100110011001

0011001100110011001100110011001100110011001100110011001100110011,001100110011001100110011001100110011
0011001100110011001100110011

4. Random

1,0

01,10

110,001

1110,1000

01010,01110

011000,101100

1010101,1011101

11010101,10010100

011010011,010101001

0101011101,0001101100

01000100100,10101001010

110101010100,100110101010

1101011111111,1011110101111

10011100101110,11111101110111

110100011111011,100101101110101

1011111001101000,1111111101111111

11010111011111111,11110101111011101

11010101101110111,111000010010110000

110101010111101111,1111111011111011111

110011111000000101010,11001101010100001000

11010101111111010101010,110111111011110101111

111100101010100101111,11010101010101011111000

101101010001010000011110,100110101010101111100111

110011001011000111000011,110001110011111101100111

100100110000111101100111,111110111011000100011100

111110101010001011100011111,101010001010101011110000

111001100101010101111101101,111100110011000100101110011

111101111110110000111011011,111011111011110110000011100

110101110111010101110111011101,101010111010101101010101001

110101010101010101010101111011,111010101110110101110111100111

110000011111111100000000000001111,111111000000111111000000001111111

111111011110101011111101110111110,111110111101111010111110110101101

100000000111111111000000001111111,111000000000111111110000000000110

110100101010101010001010111010010 1,110101110111100111100001100010011 1

111101111110111101010101111111101011,111010101110111010111100010101110 11

111101010111110101111110000011111110,111010111100000011010001111011110010

100000011111111010101111111110010111110,100110110111111100111111110000001110111

110101010111111010000011110011010100111,110111001111111011100111011011111101110

111010111010101110100111000111011011111,111100011110000110101000110100111110001

100000000111111111000001000110000111111,111000000000110111111110000000011100000110

110100101010101010001010111001111110100101,110110001110111011110011110000110001001 11

111101111110111101010101111110111110101011,111010101110111010110111011110001010111011

111101010111110101111110000011000111111110,111010111100010001110001101000111101111010

100000011111110101011111111001011110000010101111 0,100110110111110000111110011111110000001110111

110100101010101010001010111001111101001001101,110110111000111011101111001111000011000100111

111101111110111101010101111110111110101010110101,111010101110111010110101011110111110001010111101

101001010101001010101010101010101010101010100101010,111101010111001010101111010010101000100011000010

111101010111110101111110000011000111111110101110,111010111100010001110001101000101011111011110010

100000011111110101011111111001011110000010101111 0,100110110111110000111110011111110000001110111

110100101010101010001010111001111101001010110101 01,111010111010011000111011101111001111000011000100111

111101010111110101111110000011000111111011110101110,110101111101000100011100011010001010111110111100101

100000011111110101011111111001011110000010101010110111,100110110101101101111100001111100111111110000011110111

110100101010101010001010111001111101001010111010101,111011100110100110001110111011110011110000110001001111

101110101011010101010111011010100111001101010111010101,110001110001110010101111110001100101000001010100001111

100111111101101010101010101011011110101000111010111101 1,110101001011001110110111010111011111011110101111101000

111101111011101111110111011111010101000011111111101111 01,110101010111111011110110111111010101010111110101110110

111011111011101111011011010111011100000100110110000011110,101110111111101011111011101111111011010111111101001001

111011101111100111100001111101011111100000001110101110101 1,110101111011001010111110101010101011010111010101110110

100001101001111010101011101011111010101010101110101011111111000,110000111000111101010111100010111100111011110100011000111110

111111110001101010010101100001010101010110101011100101010001001,101010111111111111111111000000010101111111110000101

111101010111110101111110000011000111100000000111111011110101110,111010111100010001110001101000101011111010111100110010

100011110100111010101111101011111100000110001001111011010111111110,100011101111010001110101111000100011100001101000111010111100100010

APPENDIX E

E. OUTPUT.TXT FILE

Output.txt file lists each multiplication and corresponding EDP for all multiplier architectures for chessboard pattern.

Bits	CSAM EnergyxDelay	RCAM EnergyxDelay	WTM EnergyxDelay	ABACUS EnergyxDelay
2	0.01	0.12	0.18	0.45
3	0.10	0.42	0.89	5.21
4	2.57	6.26	1.75	14.53
5	7.69	15.15	8.92	29.01
6	15.14	27.94	23.78	59.05
7	27.80	45.09	47.83	94.52
8	55.83	67.54	83.03	142.93
9	99.27	96.26	128.25	202.63
10	147.13	142.64	182.03	274.23
11	222.70	199.36	254.32	362.54
12	295.97	267.69	363.75	514.65
13	402.46	349.05	476.37	649.14
14	526.85	444.97	607.05	805.31
15	706.87	557.12	753.99	984.49
16	872.97	702.71	928.33	1187.92
17	1103.28	869.79	1102.02	1410.74
18	1323.91	1060.37	1316.37	1653.25
19	1614.09	1276.57	1549.31	1922.42
20	1942.29	1520.64	1791.83	2219.34
21	2357.31	1794.96	2073.94	2545.12
22	2726.74	2121.78	2382.58	2900.76
23	3261.56	2485.30	2837.53	3447.84
24	3698.65	2888.23	3205.67	3879.15
25	4329.52	3333.42	3602.36	4343.63
26	4928.26	3823.84	4040.45	4842.19
27	5705.25	4362.59	4502.76	5375.72
28	6399.17	4976.15	5000.98	5945.09
29	7312.97	5645.67	5522.20	6551.14
30	8126.29	6374.58	6089.87	7194.71
31	9163.24	7166.43	6681.83	7876.60
32	10162.33	8024.92	7311.81	8597.58
33	11433.81	8953.87	7972.99	9348.61
34	12546.29	9983.22	8661.49	10129.85

35	14023.51	11091.84	9380.41	10951.69
36	15342.28	12283.86	10142.43	11814.89
37	16967.27	13563.56	10937.22	12720.16
38	18490.69	14935.35	11771.46	13662.24
39	20391.25	16403.78	12645.79	14657.84
40	22104.04	18001.48	13560.86	15695.64
41	24241.32	19705.78	14515.29	16772.33
42	26165.67	21521.56	15510.72	17900.57
43	28525.31	23453.81	16547.74	19071.03
44	30730.81	25507.67	17626.96	20293.36
45	33414.31	27688.37	18748.98	21560.20
46	35865.83	30030.48	20354.54	23411.56
47	38836.11	32510.59	21580.04	24795.98
48	41553.16	35134.29	22851.29	26230.02
49	44802.42	37907.30	24167.84	27711.26
50	47857.85	40835.47	25530.25	29246.30
51	51489.93	43924.78	26940.04	30830.74
52	54851.05	47210.87	28393.77	32470.13
53	58825.16	50670.42	29897.95	34161.05
54	62511.76	54309.73	31450.10	35901.07
55	66824.43	58135.24	33049.75	37698.73
56	70913.08	62153.52	34693.40	39549.59
57	75672.16	66371.24	36396.55	41456.18
58	80130.18	70824.42	38143.70	43417.04
59	85291.69	75490.54	39944.34	45433.69
60	90145.17	80376.66	41793.96	47506.68
61	95705.81	85489.91	43695.03	49636.50
62	101032.44	90837.57	45648.04	51823.67
63	107097.69	96427.06	47653.44	54068.70
64	112861.37	102293.93	49711.73	56372.08