DESIGN OF LOW POWER KARAT SUBA MULTIPLIER USING SELF-CONFIGURATION USING VLSI

A PROJECT REPORT

Submitted by

THANUSH RAJAN .A (950919106020)

In partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

ELECTRONICS AND COMMUNICATION ENGINEERING

HOLYCROSS ENGINEERING COLLEGE, TUTICORIN

ANNA UNIVERSITY:: CHENNAI – 600 025

MAY 2023

ANNA UNIVERSITY: CHENNAI – 600 025

BONAFIDE CERTIFICATE

Certified that this report "**DESIGN OF LOW POWER KARAT SUBA MULTIPLIER USING SELF CONFIGURATION USING VLSI"** is the bonafide work of **"THANUSH RAJAN. A (950919106020)"** who carried out the project work under my supervision.

Mrs.M.Pon Kanagavalli,M.E.,	Mrs.A.Arul Angelin,M.E.,
HEAD OF THE DEPARTMENT	**SUPERVISOR**
Electrical and Electronics Engineering,	Electronics and Communication Engineering,
Holycross Engineering College, Tuticorin-628 851	Holycross Engineering College, Tuticorin-628 851

Submitted for the project on 18.05.2023

ACKNOWLEDGEMENT

First and foremost, we would like to thank **"The Almighty"**, the lord of all creations who by his abundant grace sustained us to work on this project successfully.

We express our deepest gratitude to our founder and managing director **Dr.K.R.Prakash Rajkumar,** for providing us with necessaryfacilities that enabled us to complete our project successfully.

We are very grateful to our correspondent **Mr.D.S.K.Rajarathinam** for providing us with an environment to complete our project successfully.

We wish to express our sincere thanks to our principal **Dr.Arunmozhi Selvi, M.E.,Ph.D.,** for her tremendous contribution and moral support towards the completion of this project.

We express our sincere thanks to the head of the department **Mrs.Pon Kanagavalli,M.E.,** for useful suggestions given during thecourse of the project.

We are very grateful to our guide **Mrs.A.Arul Angelin,M.E.,** assistant professor, for being instrumental in the completion of our project with her complete guidance.

We also thank all the **staff members, our family** and **friendsfor** their Help to make this project a successful one.

ABSTRACT

Multi-digit multiplication is widely used for various applications in recent years, including numerical calculation, chaos arithmetic, primality testing. Systems with high performance and low energy consumption are demanded, especially for image processing and communications with cryptography using chaos. In this project,hardware design of multi-digit multiplication is described, and its VLSI realisation is evaluated in terms of the cost and performance. A configurable Karatsuba multiplier is proposed for low power application. The results obtained by this study will help system designers for applications requiring multi-digit multiplication to select design alternatives including ASIC realisation. Multiplications are very expensive and slows the over all operation. Theperformance of many computational problems are often dominated by the speed at which a multiplication operation can be executed. In this project, a low-error and area-efficient fixed-width Karatsuba multiplier is presented. As compared with the state-of-the-art design, the proposed fixed-width multiplier performs not only with lower compensation error but also with lower hardware complexity, especially as multiplier input bits increase.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS	EXPANSION	PAGE NO.
VLSI	Very Large-Scale Integration	8
HSPICE	Hewlett Simulation Program with Integrated Circuit Emphasis	9
FFT	Fast Fourier Transform	10
SRAM	Static Random-Access Movie	10
CSA	Carry Save Adder	11
BCD	Binary Coded Decimal	12
SMSD	Sign Magnitude Signed Digit	12
DST	Data Scaling Technology	12
FWBM	Fixed Width Booth Multiplier	12
TOSAM	Truncation and Rounding based Scalable Approximate Multiplier	16
LSB	Least Significant Bit	19
KMA	Karatsuba Multiplication Algorithm	23
CSE	Common Subexpression Elimination	28

HDL	Hardware Description Language	30
FPGA	Field Programmable Gate Arrays	36
ASIC	Application Specific Integrated Circuit	37
CLB	Configurable Logic Block	38
DCM	Digital Clock Manager	39
IC	Integrated Circuit	41
EDA	Electronic Design Automation	41
ISE	Integrated Synthesis Environment	61
RTL	Register Transfer Level	62
LUT	LookUp Table	64

Chapter-1

INTRODUCTION

1.1 Multiplier

A multiplier is one of the key hardware blocks in most digital signal processing (DSP) systems. Typical DSP applications where a multiplier plays an important role include digital filtering, digital communications and spectral analysis (Ayman.A (2001)). Many current DSP applications are targeted at portable, battery-operated systems, so that power dissipation becomes one of the primary design constraints. Since multipliers are rather complex circuits and must typically operate at a high system clock rate, reducing the delay of a multiplier is an essential part of satisfying the overall design.

Multiplications are very expensive and slows the over all operation. The performance of many computational problems are often dominated by the speed at which a multiplication operation can be executed. Consider two unsigned binary numbers X and Y that are M and N bits wide, respectively. To introduce the multiplication operation, it is useful to express X and Y in the binary representation

$X = \Sigma\ Xi\ 2i\ i = 0 \text{ to } M$ (1)

$Y = \Sigma\ Yj\ 2j\ j = 0 \text{ to } N$ (2)

With Xi , $Yj \in \{0,1\}$. The multiplication operation is then defined as follows:

$Z = X \times Y = \Sigma\ Zk\ 2k\ k = 0 \text{ to } M + N – 1$ (3)

= (Σ Xi 2i i = 0 to M) (Σ Yj 2j j = 0 to N) (4)

= Σ (Σ Xi Yj 2i+j) i = 0 to M-1, j = 0 to N-1 (5)

The simplest way to perform a multiplication is to use a single two input adder. For inputs that are M and N bits wide, the multiplication tasks M cycles, using an N-bitadder. This shift –and-add algorithm for multiplication adds together M partial products. Each partial product is generated by multiplying the multiplicand with a bit of the multiplier – which, essentially, is an AND operation – and by shifting theresult in the basis of the multiplier bit's position. Similar to the familiar long hand decimal multiplication, binary multiplication involves the addition of shifted versions of the multiplicand based on the value and position of each of the multiplier bits. As a matter of fact, it's much simpler to perform binary multiplication than decimal multiplication. The value of each digit of a binary number can only be 0 or 1, thus, depending on the value of the multiplier bit, the partial products can only be a copy of the multiplicand, or 0. In digital logic, this is simply an AND function.

A faster way to implement multiplication is to resort to an approach similar to manually computing a multiplication. The entire partial product are generated at the same time and organized in an array. A multioperand addition is applied to compute the final product. The approach is illustrated in the figure. This set of operation can be mapped directly into hardware. The resulting structure is called an array multiplier and combines the following three functions: partial product generation, partial-product accumulation and final addition.

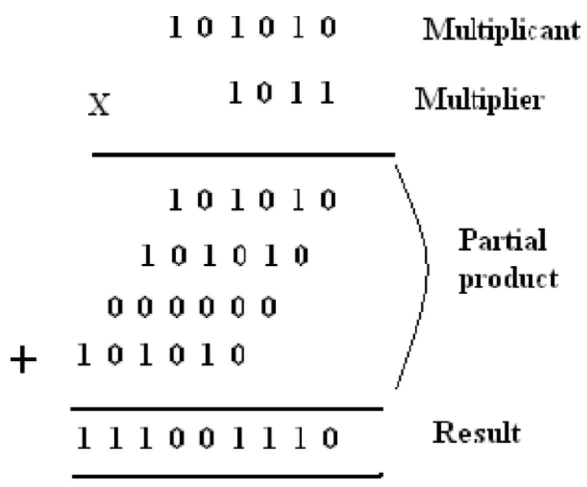


Fig1.1: Example of manual multiplication

So the adder unit is very important for designing any multiplier(John Rabaey (2003)).The different types of adders and their functions were discussed in (Oklobdzija.V.G (1995)),(Pucknell (2004)),(Shalem.R (1999)) and (Zimmermann.R and Fichtner.W (1997)). From the results of (Shalem.R (1999)) ,we came to know that the new improved 14 Transistor full adder cell shows better result in Threshold loss problem, power dissipation and speed by sacrificing MOS transistor count.

1.2 ARRAY MULTIPLIER

The composition of an array multiplier is shown in the figure1.2. There is a one to one topological correspondence between this hardware structure and the manual multiplication shown in figure1.2. The generation of n partial products requires N*Mtwo bit AND gates. Most of the area of the multiplier is devoted to the adding of n partial products, which requires N-1, M-bit adders. The shifting of the partial products for their proper alignment’s performed by simple routing and does not require any logic. The over all structure can be easily be compacted into rectangle, resulting in very efficient layout.

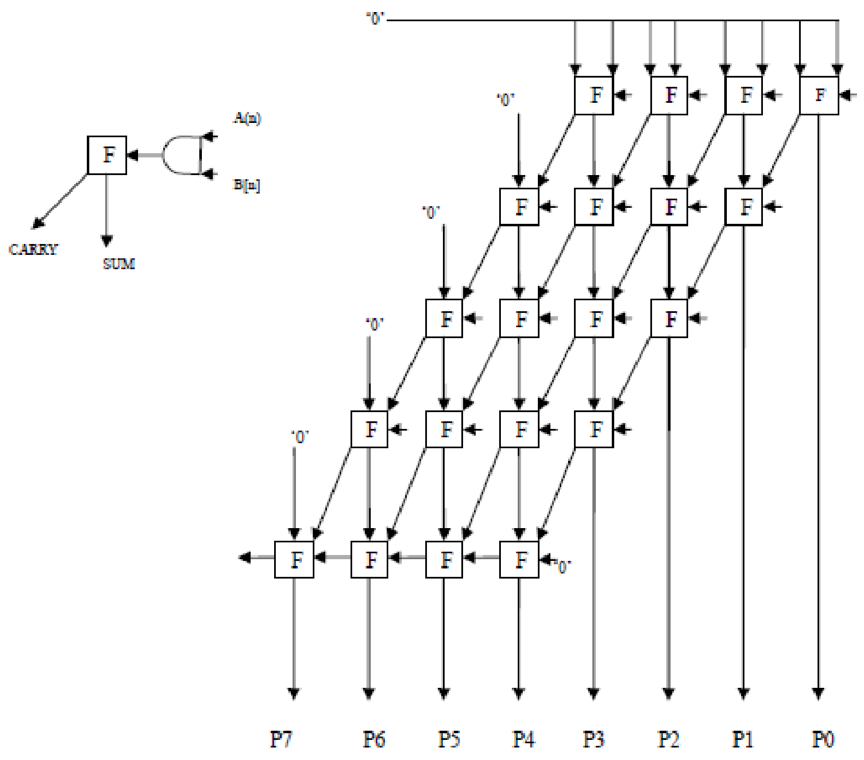


Fig1.2: Array multiplier

1.3 BAUGH WOOLEY MULTIPLIER

Baugh-Wooley algorithm for the unsigned binary multiplication is based on the concept shown in figure1.3. The algorithm specifies that all possible AND terms are created first, and then sent through an array of half-adders and full- adders with the Carry-outs chained to the next most significant bit at each level of addition. Negative operands may be multiplied using a Baugh-Wooley multiplier.

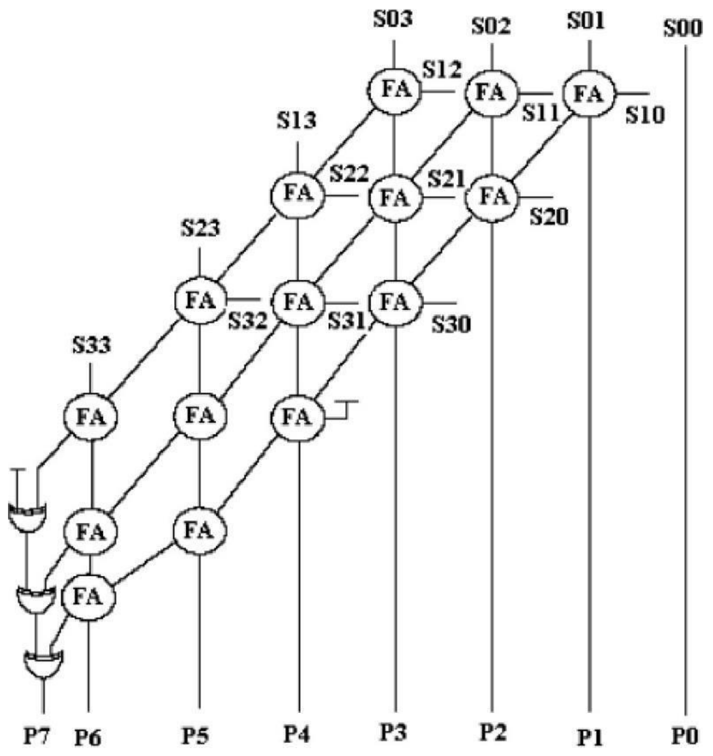


Fig1.3: Baugh Wooley multiplier

1.4 BRAUN MULTIPLIER

The simplest parallel multiplier is the Braun array. All the partial products are computed in parallel, then collected through a cascade of Carry Save Adders. The completion time is limited by the depth of the carry save array, and by the carry propagation in the adder. Note that this multiplier is only suited for positive operands. The structure of the Braun algorithm for the unsigned binary multiplication is shown in figure 1.4.

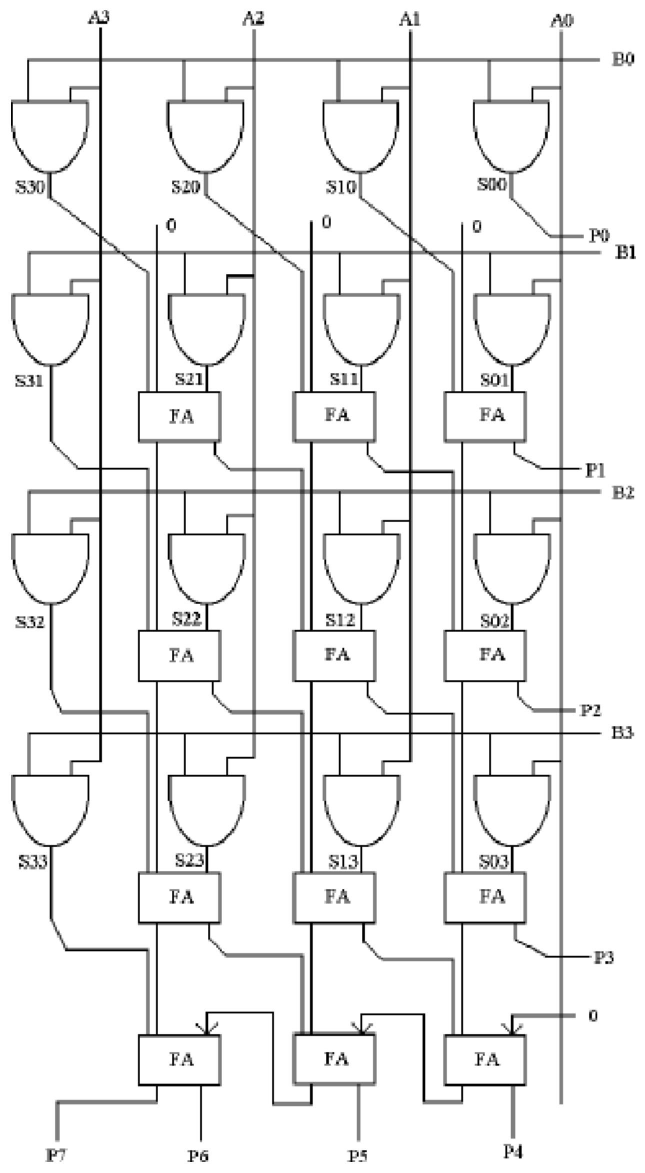


Fig1.4: Braun multiplier

1.5 WALLACE TREE MULTIPLIER

The partial-sum adders can also be rearranged in a tree like fashion, reducingboth the critical path and the number of adder cells needed. The presented structure is called the Wallace tree multiplier and its implementation is shown in figure 1.5.

The tree multiplier realizes substantial hardware savings for larger multipliers. The propagation delay is reduced as well. In fact, it can be shown that the propagation delay through the tree is equal to O (log3/2 (N)). While substantially faster than the carry-save structure for large multiplier word lengths, the Wallace multiplier has the disadvantage of being vary irregular, which complicates the task of an efficient layout design.

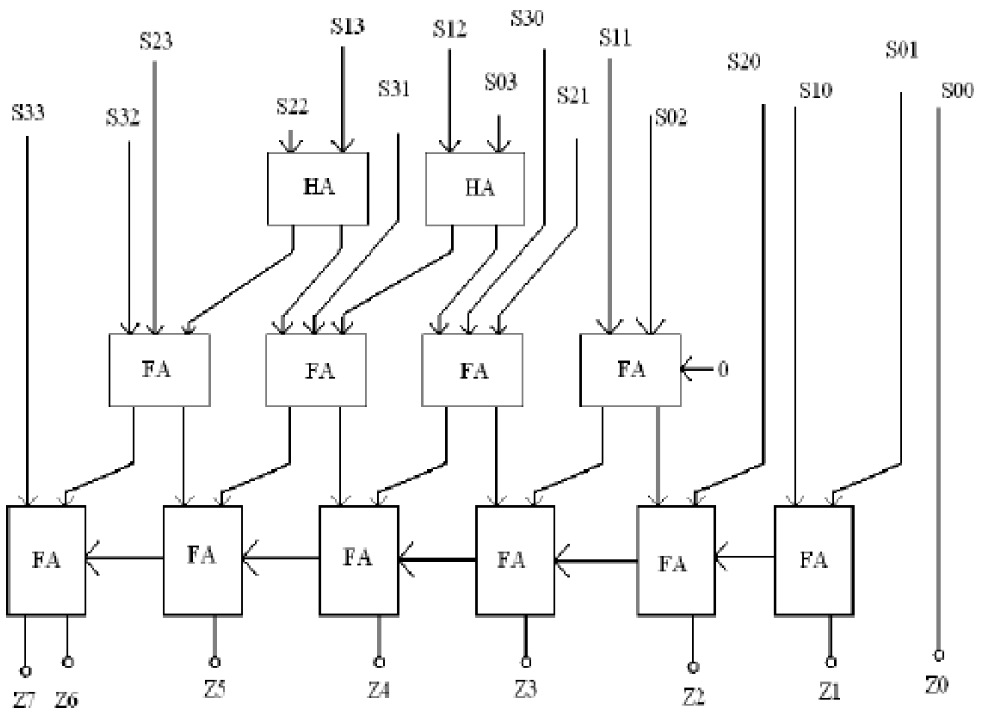


Fig1.5: Wallace tree multiplier

In this project proposes the objective of good multiplier to provide a physically compact high speed and low power consumption unit. Being a core part of arithmetic processing unit multipliers are in extremely high demand on its speed and low power consumption. To reduce significant power consumption of multiplier design it is a good direction to reduce number of operations thereby reducing a dynamic power which is a major part of total power dissipation. In the past considerable effort were put into designing multiplier in VLSI in this direction.

CHAPTER-2

LITERATURE SURVEY

Shrestha presents an area-efficient low-power architecture for configurable booth multiplier. It is synthesized and post-layout simulated using 90 nm CMOS process and it occupies 9511 µm2 and consumes 1.73 mW at 167 MHz. Comparatively, the proposed multiplier architecture requires 43.12% and 75.65% lower area and power, respectively, in comparison with the state of the art work

Selvakumar proposes a low error and power optimized architecture for the multiplier based on multiplexers that aims for an optimized truncated product and power. The design of efficient truncation scheme with minimum truncation error and low power multiplier is essential for VLSI implementation of Signal Processing Devices. Various conventional array and parallel multipliers have been used and many of them boost the speed of the device at the cost of large VLSI area and high power dissipation. A novel design for multiplier based on multiplexer has been proposed in this paper considering the existing multiplexer based architecture.The proposed low error and power optimized multiplexer based truncated Multiplier was implemented in HSPICE environment in TSMC 180 nm library technology files. The results obtained are tabulated in the simulation result section, and it is observed that the proposed truncated multiplier architecture consumes approximately 35% reduction in dynamic power with minimum error. It also reduces the number of transistors by 37% when compared to the existing multiplexer based multiplier the conventional multiplexer based multiplier for 8 X8 bit multiplication operation

Eriksson introduce a multiplier test-vector generation method that has the ability to exercise such long carry propagation paths. Through extensive circuit simulation and static timing analysis, we evaluate the quality of the test vectors that result from the new method. Especially for fast multipliers with a pronounced carry propagation, the timing-critical vectors manage to stimulate a path, which has a delay that comes close to the true worst case delay. We investigate the complexity and run-time for the test-vector generation, and derive timing-critical vectors up to a factor word length of 54 bits.

Shanmuga design an extended multiplier low power consumption adders to minimize complexity and reduce power consumption compared to other designs. This can increase the speed and reduce the dissipation of power. A newer GDI (Gate Diffusion Input) transmission gate was used to implement the adder configuration. This technology allows reducing the power consumption, delay and area of low-power digital circuits thus dealing with low logic design difficulties

Wang presents the design of a power- and areaefficient high-speed 768 000-bit multiplier, based on fast Fourier transform multiplication for fully homomorphic encryption operations. A memory-based in-place architecture is presented for the FFT processor that performs 64 000-point finite-field FFT operations using a radix-16 computing unit and 16 dual-port SRAMs. By adoptinga special prime as the base of the finite field, the radix-16 calculations are simplified to requiring only additions and shift operations. A two-stage carry-look- ahead scheme is employed to resolve carries and obtain the multiplication result. The multiplier design is validated by comparing its results with the GNU Multiple Precision (GMP) arithmetic library. The proposed design has been synthesized using 90-nm process technology with an estimated die area of 45.3 mm2. At 200 MHz, the large-number multiplier offers roughly twice the performance of a previous implementation on an NVIDIA C2050 graphics processor unit and is 29 times faster than the Xeon X5650 CPU, while at the same time consuming a modest 0.97

Fritz present the domains in terms of speed and area in which each multiplier is preferable, in which the choice must be left up to the application- specific cost function. Circuits designed to perform binary multiplication are widely used in many settings. An optimal design for any computer arithmetic circuit may not always be the fastest one. Instead, a tradeoff between speed and chip area is important. Thus, we find that the Interlaced Partition multiplier, which is able to greatly reduce the lengths of the long carry chains in multiplier circuits, has a place as a tradeoff between the slow but cheap array multipliers and the speedy but large Dadda or Wallace Tree multipliers.

Young proposed a new architecture of multiplier-and-accumulator (MAC) for high-speed arithmetic. By combining multiplication with accumulation and devising a hybrid type of carry save adder (CSA), the performance wasimproved. Since the accumulator that has the largest delay in MAC was merged into CSA, theoverall performance was elevated. The proposed CSA tree uses 1's-complement- based radix-2 modified Booth's algorithm (MBA) and has the modified array for the sign extension in order to increase the bit density of the operands. The CSA propagates the carries to the least significant bits of the partial products and generates the least significant bits in advance to decrease the number of the input bits of the final adder. Also, the proposed MAC accumulates the intermediate results in the type of sum and carry bits instead of the output of the final adder, which made it possible to optimize the pipeline scheme to improve the performance. The proposed architecture was synthesized with 250, 180 and 130 m, and 90 nm standard CMOS library. Based on the theoretical and experimental estimation, we analyzed the results such as the amount of hardware resources, delay, and pipelining scheme. We used Sakurai's alpha power law for the delay modeling. The proposed MAC showed the superior properties to the standard design in many ways and performance twice as much as the previous research in the similar clock frequency. We expect that the proposed MAC can be adapted to various fields requiring high performance such as the signal processing areas

Gorgin propose a parallel 16 × 16 radix-10 BCD multiplier, where 17 partial products are generated with [−6, 6] SMSD representation. Some innovationsof this paper and use of previous techniques, as listed below, have led to marginal 1.5% less area consumption and 10% less power dissipation, at 4.8-ns latency, with respect to the fastest previous work [4]. The least possible delay for the latter is 4.8ns, while the proposed design leads the synthesis tool to meet the 4.4-ns time constraint (i.e., 9% faster). In other words, the advantage is that the proposed design can operate in 9% higher frequency and dissipate up to 13% less power withno claim in area improvement

Chen propose a data scaling technology (DST) for use in a low-error fixed-width Booth multiplier (FWBM) to reduce truncation errors. The proposed DST reduces the number of redundant bits in the multiplicand, yielding more efficient bits in low-error FWBMs. The truncation errors in FWBMs are reduced by adding a circuit incorporating the proposed DST to them as well as an error- compensation circuit. We found that the signal-to-noise ratio of the proposed DSTFWBM (1 bit) was more than 1.05 dB higher than that of an FWBM without the DST circuit. Long-width DST-FWBMs achieved an accuracy closely approaching the ideal value of a post-truncated multiplier. To verify its performance in a VLSI chip, we implemented the DST-FWBM in a 0.18-µm CMOS process. The proposed DST method was shown to considerably improve the accuracy of FWBMs, rendering this technology suitable for use in digital signal processing technique

Abdulrahman present new hybrid-double multiplication architectures. To the best of our knowledge, this is the first time such a hybrid multiplier structure using the polynomial basis is proposed. Prototypes of the presented serial-out bit-level schemes and the proposed hybrid-double multiplication architectures (10 schemes in total) are implemented over both GF(2163) and GF(2233), and experimental results are presented. he Serial-out bit-level multiplication scheme is characterized by an important latency feature. It has an ability to sequentially generate an output bit of the multiplication result in each clock cycle. However, the computational complexity of the existing serial-out bit-level multipliers in GF(2m) using normal basis representation, limits its usefulness in many applications; hence, an optimized serialout bit-level multiplier using polynomial basis representation is needed. In this paper, we propose new serial-out bit-level Mastrovito multiplier schemes

Wey propose a reliable low-power multiplier design by adopting algorithmic noise tolerant (ANT) architecture with the fixed-width multiplier to build the reduced precision replica redundancy block (RPR). The proposed ANT architecture can meet the demand of high precision, low power consumption, and area efficiency. We design the fixed-width RPR with error compensation circuit viaanalyzing of probability and statistics. Using the partial product terms of input correction vector and minor input correction vector to lower the truncation errors, the hardware complexity of error compensation circuit can be simplified. In a 12 ×

12 bit ANT multiplier, circuit area in our fixed-width RPR can be lowered by 44.55% and power consumption in our ANT design can be saved by 23% as compared with the state-of-art ANT design

Tsoumanis introduce an architecture of pre-encoded multipliers for Digital Signal Processing applications based on off-line encoding of coefficients. To this extend, the Non-Redundant radix-4 Signed-Digit (NR4SD) encoding technique, which uses the digit values {−1, 0, +1, +2} or {−2, −1, 0, +1}, is proposed leading to a multiplier design with less complex partial products implementation. Extensive experimental analysis verifies that the proposed pre-encoded NR4SD multipliers, including the coefficients memory, are more area and power efficient than the conventional Modified Booth scheme

Li proposed an efficient MK bit-parallel multiplier for irreducible Type II pentanomial. Our proposal requires only one more TX delay compared with the fastest bit-parallel multiplier of the same type know to data, but saves about 1/4 logic gates. Meanwhile, we also present new formulae for modular reductions, which can be utilized to construct associated Mastrovito matrices directly. Applying a similar formula, we demonstrate that Type I pentanomials have more complicated modular reduction under the same SPB/Montgomery parameter. Thus,associated SPB bit-parallel multipliers would be less efficient than Type II pentanomials. Our next work is developing MK multiplier for Type C.1 and C.2 pentanomials using GP

Cui a proposed new RB modified partial product generator (RBMPPG)
; it removes the extra ECW and hence, it saves one RBPP accumulation stage. Therefore, the proposed RBMPPG generates fewer partial product rows than a conventional RB MBE multiplier. Simulation results show that the proposed

RBMPPG based designs significantly improve the area and power consumption when the word length of each operand in the multiplier is at least 32 bits; these reductions over previous NB multiplier designs incur in a modest delay increase (approximately 5%). The power-delay product can be reduced by up to 59% using the proposed RB multipliers when compared with existing RB multiplier

Vahdat presented a scalable approximate multiplier, called truncation- and rounding-based scalable approximate multiplier (TOSAM), which reduces the number of partial products by truncating each of the input operands based on their leading one-bit position. In the proposed design, multiplication is performed by shift, add, and small fixed-width multiplication operations resulting in large improvements in the energy consumption and area occupation compared to those of the exact multiplier. To improve the total accuracy, input operands of the multiplication part are rounded to the nearest odd number. Because input operands are truncated based on their leading one-bit positions, the accuracy becomesweakly dependent on the width of the input operands and the multiplier becomes scalable. Higher improvements in design parameters (e.g., area and energy consumption) can be achieved as the input operand widths increase. To evaluate the efficiency of the proposed approximate multiplier, its design parameters are compared with those of an exact multiplier and some other recently proposed approximate multipliers. Results reveal that the proposed approximate multiplier with a mean absolute relative error in the range of 11%-0.3% improves delay, area, and energy consumption up to 41%, 90%, and 98%, respectively, compared to those of the exact multiplier. It also outperforms other approximate multipliers in terms of speed, area, and energy consumption. The proposed approximate multiplier has an almost Gaussian error distribution with a near-zero mean value. We exploit it in the structure of a JPEG encoder, sharpening, and classification applications. The results indicate that the quality degradation of the output is negligible. In addition, we suggest an accuracy configurable TOSAM where the energy consumption of the multiplication operation can be adjusted based on the minimum required ac- uracy.

CHAPTER-3

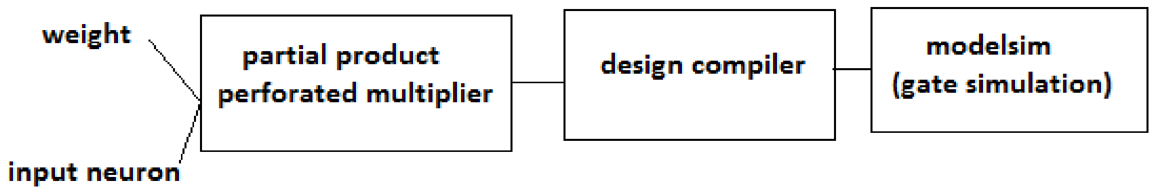
EXISTING SCHEME

We target the design of power–error efficient multiplication circuits. We differ fromthe previous works by exploring approximation on the generation of the partial products. The proposed method can be easily applied in any multiplier architecture without the need for a special design, in contrast to related works. In addition, the error imposed by perforation depends only on the configuration parameters and, in contrast to existing work, can be analytically calculated without the need for exhaustive simulations.

$$A \times B = \sum_{i=0}^{n-1} A b_i 2^i, \;\; b_i \in \{0, 1\}.$$

$$A \times B|_{j,k} = \sum_{\substack{i=0, \\ i \notin [j, j+k)}}^{n-1} A b_i 2^i, \;\; b_i \in \{0, 1\}.$$

the partial product perforation method for the design of approximate hardware multipliers is described. Consider two *n*-bit numbers *A* and *B*. The result of their multiplication *A* × *B* is obtained after summing all the partial products *Abi*, where *bi* is the *i* th bit of *B*. Thus The partial product perforation technique omits the generation of *k* successive partial products starting from the *j* th one. A perforated partial product is not inserted in the accumulation tree, and hence *n* full adders can be eliminated. Applying the product perforation with *j* and *k* configuration values on the multiplication, *A* × *B* produces the approximate result For each architecture, the dot diagrams of the accurate and the respective perforated tree are presented. The dots represent the bits of the partial products that have to be accumulated, while the stages represent the delay of the reduction process followed by each tree. The dashed boxes with four dots are 4:2 compressors, those with three are full adders and those with two are either full- or half-adders. Through the proposed approximation technique, the power, area, and delay of the multiplication circuit are decreased, making, though, the computation imprecise. The higher the order of a perforated partial product, the greater the error imposed at the final result. In addition, since the addition is an associative and commutative operation, when more than one partial products are perforated, the total error results from the addition of the errors produced from the perforation of each partial product separately For each architecture, the dot diagrams of the accurate and the respective perforated tree are presented. The dots represent the bits of the partial products that have to be accumulated, while the stages represent the delay of the reduction process followed by each tree. The dashed boxes with four dots are 4:2 compressors, those with three are full adders and those with two are either full- or half-adders. Through the proposed approximation technique, the power, area, and delay of the multiplication circuit are decreased, making, though, the computation imprecise. The higher the order of a perforated partial product, the greater the error imposed at the final result. In addition, since the addition is an associative and commutative operation, when more than one partial products are perforated, the total error results from the addition of the errors produced from the perforation of each partial product separately that produces specific least significant bits (LSBs) of the accumulation tree, while the perforation skips the generation of partial products and thus decreases the number of operands to be accumulated. For example, in an 8-bit array multiplier, perforating a partial product removes eight full adders from the accumulation tree and reduces its delay. In order to attain similar circuit reduction using truncation, 6 LSB have to be truncated. However, truncating 6 LSB does not offer any delay reduction. Moreover, in this example, the truncation delivers, in all the cases, incorrect results, whereas the outputs of perforation are 50% correct. Finally, perforating one partial product (out of eight) results in a 12.5% loss of information while truncating 6 LSB (out of 16) results in a 37.5% information loss


weight
input neuron
partial product perforated multiplier
design compiler
modelsim (gate simulation)

```
        10100000
        01100100

        00000000
       00000000
      10100000
     00000000
    00000000
   10100000
  10100000
 00000000
0 111110100000000
```

Above example for both multiplier and multiplicand more number of one is presentin MSB when you truncate LSB bits in partial product do not affect output value majorly .based on statistical data in multiplier input number of half adders and full adders will be reduced.

3.1 Design of Power and Area Efficient Approximate Multipliers

Suganthi Venkatachalam and Seok-Bum Ko deals with a new design approach for approximation of multipliers. The partial products of the multiplier are altered to introduce varying probability terms. Logic complexity of approximation is varied for the accumulation of altered partial products based on their probability. The proposed approximation is utilized in two variants of 16-bit multipliers. Synthesis results reveal that two proposed multipliers achieve power savings of 72% and 38%, respectively, compared to an exact multiplier. They have better precision when compared to existing approximate multipliers. Mean relative error figures are as low as 7.6% and 0.02% for the proposed approximate multipliers, which are better than the previous works. Performance of the proposed multipliers is evaluated with an image processing application, where one of the proposed models achieves the highest peak signal to noise ratio.

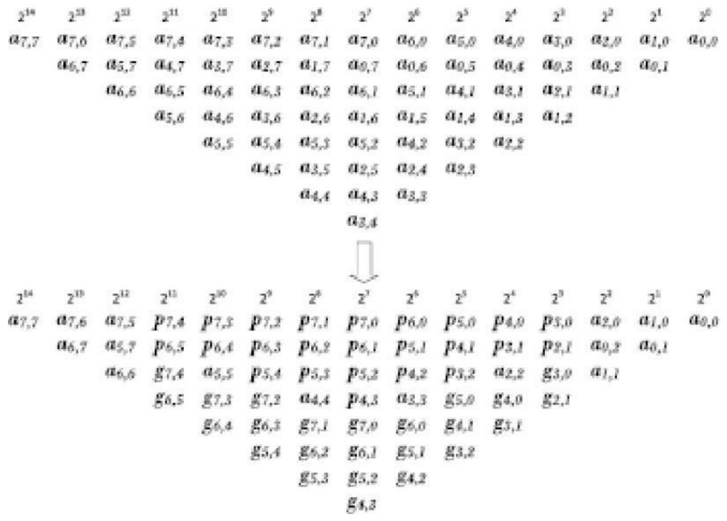


Fig3.1: Transformation of generated partial products into altered partial products.

Implementation of multiplier comprises three steps: generation of partial products, partial products reduction tree, and finally, a vector merge addition to produce final product from the sum and carry rows generated from the reduction tree. Second step consumes more power.

In this project, approximation is applied in reduction tree stage. A 8-bit unsigned1 multiplier is used for illustration to describe the proposed method in approximation of multipliers. Consider two 8-bit unsigned input operands $\alpha = \sum_{m=0}^{7} \alpha_m 2^m$ and $\beta = \sum_{n=0}^{7} \beta_n 2^n$

The partial product *am,n* = *αm* ・ *βn* in Fig3.1 is the result of AND operation between the bits of *αm* and *βn*.

From statistical point of view, the partial product *am,n* has a probability of 1/4 of being 1. In the columns containing more than three partial products, the partial products *am,n* and *an,m* are combined to form *propogate* and *generate* signals as given. The resulting *propogate* and *generate* signals form altered partial products *pm,n* and *gm,n*. From column 3 with weight 23 to column 11 with weight 211, the partial products *am,n* and *an,m* are replaced by altered partial products *pm,n* and *gm,n*. The original and transformed partial product matrices are shown in Fig3.2

pm,n = *am,n* + *an,m*

gm,n = *am,n* ∗*an,m.*

The probability of the altered partial product *gm,n* being one is 1/16, which is significantly lower than 1/4 of *am,n*. The probability of altered partial product *pm,n* being one is 1/16 + 3/16 + 3/16 = 7/16, which is higher than *gm,n*. These factors are considered, while applying approximation to the altered partial product matrix.

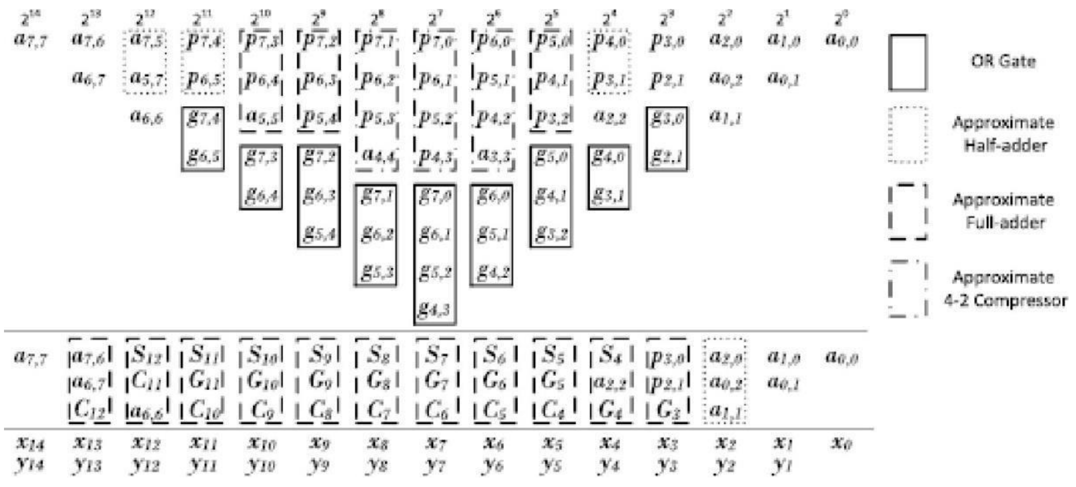


Fig3.2 Reduction of altered partial products.

Fig3.2 shows the reduction of altered partial product matrix of 8 × 8 approximate multiplier. It requires two stages to produce sum and carry outputs for vector merge addition step. Four 2-input OR gates, four 3-input OR gates, and one 4-input OR gates are required for the reduction of *generate* signals from columns 3 to 11. The resultant signals of OR gates are labeled as *Gi* corresponding to the column *I* with weight 2*i* . For reducing other partial products, 3 approximate half-adders, 3 approximate full-adders, and 3 approximate compressors are required in the first stage to produce *Sum* and *Carr y* signals, *Si* and *Ci* corresponding to column *i* . The elements in the second stage are reduced using 1 approximate half-adder and 11 approximate full-adders producing final two operands *xi* and *yi* to be fed to ripple carry adder for the final computation of the result.

CHAPTER-4

PROPOSED SYSTEM

The Karatsuba formula is used to speed-up the multiplication of large numbers by splitting the operands in two parts of equal length. The proposed configurable KMA is shown in Figure4.1.

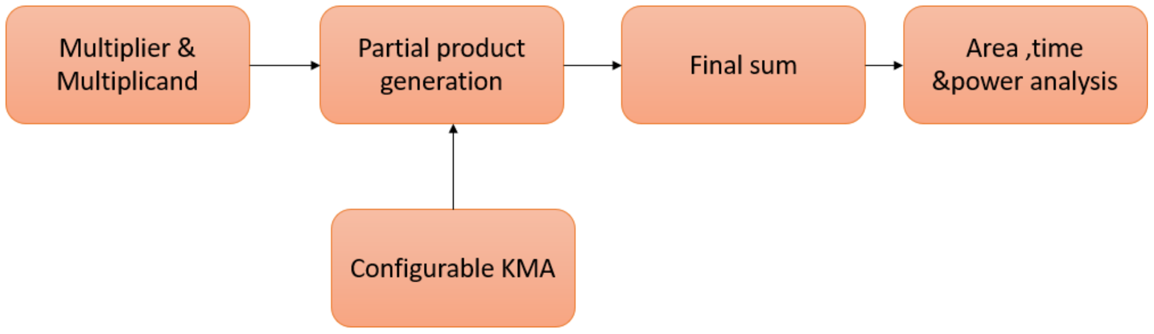

Multiplier & Multiplicand
Partial product generation
Final sum
Area ,time &power analysis
Configurable KMA

Figure4.1 : Proposed multiplier

The standard procedure for multiplication of two n-digit numbers requires a number of elementary operations proportional to n2, or o(n2) in big-O notation. Andrey Kolmogorov conjectured that the classical algorithm was asymptotically optimal, meaning that any algorithm for that task wouldrequire n2 elementary operations.

In 1960, Kolmogorov organized a seminar on mathematical problemsin cybernetics at the Moscow State University, where he stated the Ωn2conjecture and other problems in the complexity of computation. Within a week, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in o(log2) elementary steps, thus disproving the conjecture. Kolmogorov was very agitated about the discovery; he communicated it at the next meeting of the seminar, which was then terminated. Kolmogorov did some lectures on the Karatsuba result at the conferences all over the world (see, for example, "Proceedings of the international congress of mathematicians 1962", pp. 351–356, and also "6 Lectures delivered at the International Congress of Mathematicians in Stockholm, 1962") and published the method in 1962, in the Proceedings of the USSR Academy of Sciences. The article had been written by Kolmogorov and contained two results on multiplication, Karatsuba's algorithm and a separate result by Yuri Ofman it listed "A. Karatsuba and Yu. Ofman" as the authors. Karatsuba only became aware of the paper when he received the reprints from the publisher.

Let x and y be represented as n-digit strings in some base B. For any positive integer m less than n, one can write the two given numbers as

$$x = x_1 B^m + x_0,$$
$$y = y_1 B^m + y_0,$$

where x_0 and y_0 are less than B^m. The product is then

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0),$$
$$xy = z_2 B^{2m} + z_1 B^m + z_0,$$

These formulae require four multiplications and were known to Charles Babbage.Karatsuba observed that xy can be computed in only three multiplications, at the cost of a few extra additions. With z0 and z2 as before one can calculate

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0,$$

which holds, since

$$z_1 = x_1 y_0 + x_0 y_1,$$
$$z_1 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0.$$

An issue that occurs, however, when computing z1 is that the above computation of (x1+x0) and (y1+y0) may result in overflow which require a multiplier having one extra bit. This can be avoided by noting that

$$z_1 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0,$$
$$z_1 = (x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0) - x_1 y_1 - x_0 y_0,$$
$$z_1 = (-x_1 y_1 + x_1 y_0 + x_0 y_1 - x_0 y_0) + x_1 y_1 + x_0 y_0,$$
$$z_1 = -(x_1 y_1 - x_1 y_0 - x_0 y_1 + x_0 y_0) + x_1 y_1 + x_0 y_0,$$
$$z_1 = -(x_1 - x_0)(y_1 - y_0) + x_1 y_1 + x_0 y_0,$$
$$z_1 = (x_0 - x_1)(y_1 - y_0) + x_1 y_1 + x_0 y_0.$$

This computation of (x0-x1) and (y1-y0) will produce a result in the range of -bm<result<bm. This method may produce negative numbers, which require one extra bit to encode signedness, and would still require one extra bit for the multiplier.

To compute the product of 12345 and 6789, where B = 10, choose m = 3. Then we decompose the input operands using the resulting base (Bm = 1000), as:

12345 = 12 · 1000 + 345

6789 = 6 · 1000 + 789

Only three multiplications, which operate on smaller integers, are used to compute three partial results:

z2 = 12 × 6 = 72

z0 = 345 × 789 = 272205

z1 = (12 + 345) × (6 + 789) − z2 − z0 = 357 × 795 − 72 − 272205 = 283815 − 72 − 272205 = 11538

We get the result by just adding these three partial results, shifted accordingly (and then taking carries into account by decomposing these three inputs in base 1000 like for the input operands):

result = z2 · (Bm)2 + z1 · (Bm)1 + z0 · (Bm)0,

result = 72 · 10002 + 11538 · 1000 + 272205 = 83810205.

Note that the intermediate third multiplication operates on an input domain which is less than two times larger than for the two first multiplications, its output domain is less than four times larger, and base-1000carries computed from the first two multiplications must be taken into account when computing these two subtractions.

4.1 Recursive application

If n is four or more, the three multiplications in Karatsuba's basic step involve operands with fewer than n digits. Therefore, those products can be computed by recursive calls of the Karatsuba algorithm. The recursion can be applied until the numbers are so small that they can (or must) be computed directly.

In a computer with a full 32-bit by 32-bit multiplier, for example, one could choose B = 231 = 2147483648, and store each digit as a separate 32-bit binary word. Then the sums x1 + x0 and y1 + y0 will not need an extra binary word for storing the carry-over digit (as in carry-save adder), and the Karatsuba recursion can be applied until the numbers to multiply are only one-digit long.

4.2 Pseudocode

```
procedure karatsuba(num1, num2)
  if (num1 < 10) or (num2 < 10)
  return num1*num2
  /* calculates the size of the numbers */
  m  = max(size_base10(num1), size_base10(num2))
  m2 = floor(m/2)
  /* split the digit sequences in the middle */
  high1, low1 = split_at(num1, m2)
  high2, low2 = split_at(num2, m2)
  /* 3 calls made to numbers approximately half the size */
  z0 = karatsuba(low1, low2)
  z1 = karatsuba((low1 + high1), (low2 + high2))
  z2 = karatsuba(high1, high2)
  return (z2 * 10 ^ (m2 * 2)) + ((z1 - z2 - z0) * 10 ^ m2) + z0
```

4.3 Modified Multiplier

Common subexpression elimination (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value

The possibility to perform CSE is based on available expression analysis (a data flow analysis). An expression b*c is available at a point p in a program if: every path from the initial node to p evaluates b*c before reaching p, and there are no assignments to b or c after the evaluation but before p.

The cost/benefit analysis performed by an optimizer will calculate whether the costof the store to tmp is less than the cost of the multiplication; in practice other factors such as which values are held in which registers are also significant.

Compiler writers distinguish two kinds of CSE:

local common subexpression elimination works within a single basic block

global common subexpression elimination works on an entire procedure,

Both kinds rely on data flow analysis of which expressions are available at which points in a program.

The benefits of performing CSE are great enough that it is a commonly used optimization.

In simple cases like in the example above, programmers may manually eliminate the duplicate expressions while writing the code. The greatest source of CSEs are intermediate code sequences generated by the compiler, such as for array indexing calculations, where it is not possible for the developer to manually intervene. In some cases language features may create many duplicate expressions. For instance, Cmacros, where macro expansions may result in common subexpressions not apparent in the original source code.

Compilers need to be judicious about the number of temporaries created to hold values. An excessive number of temporary values creates register pressure possibly resulting in spilling registers to memory, which may take longer than simply recomputing an arithmetic result when it is needed.

CHAPTER 5

SOFTWARE DESCRIPTION

5.1 VERILOG

5.1.1 Introduction

In electronics, a hardware description language (HDL) is a specialized computer language used to program the structure, design and operation of electronic circuits, and most commonly, digital logic circuits.

A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis, simulation, and simulated testing of an electronic circuit. It also allows for the compilation of an HDL program into a lower level specification of physical electronic components, such asthe set of masks used to create an integrated circuit.

A hardware description language looks much like a programming language such as C, it is a textual description consisting of expressions, statements and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time.

HDLs form an integral part of electronic design automation (EDA) systems, especially for complex circuits, such as microprocessors.

5.1.2 Motivation

Due to the exploding complexity of digital electronic circuits since the 1970s (see Moore's law), circuit designers needed digital logic descriptions to be performed at a high level without being tied to a specific electronic technology, such as CMOS or BJT. HDLs were created to implement register-transfer level abstraction, a model of the data flow and timing of a circuit.

There are two major hardware description languages: VHDL and Verilog. There are different types of description in them "dataflow, behavioral and structural".

5.1.3 Structure of HDL

HDLs are standard text-based expressions of the structure of electronic systems and their behaviour over time. Like concurrent programming languages, HDL syntax and semantics include explicit notations for expressing concurrency. However, in contrast to most software programming languages, HDLs also include an explicit notion of time, which is a primary attribute of hardware. Languages whose only characteristic is to express circuit connectivity between a hierarchy of blocks are properly classified as netlist languages used in electric computer-aided design (CAD). HDL can be used to express designs in structural, behavioral or register-transfer-level architectures for the same circuit functionality; in the latter two cases the synthesizer decides the architecture and logic gate layout.

HDLs are used to write executable specifications for hardware. A program designed to implement the underlying semantics of the language statements and simulate the progress of time provides the hardware designer with the ability to model a piece of hardware before it is created physically. It is this executability that gives HDLs the illusion of being programming languages, when they are more precisely classified as specification languages or modeling languages. Simulators capable of supporting discrete-event (digital) and continuous-time (analog) modeling exist, and HDLs targeted for each are available.

5.1.4 Comparison with Control-Flow Languages

It is certainly possible to represent hardware semantics using traditional programming languages such as C++, which operate on control flow semantics as opposed to data flow, although to function as such, programs must be augmented with extensive and unwieldy class libraries. Generally, however, software programming languages do not include any capability for explicitly expressing time, and thus cannot function as hardware description languages. Before the introduction of System Verilog in 2002, C++ integration with a logic simulator wasone of the few ways to use object-oriented programming in hardware verification. System Verilog is the first major HDL to offer object orientation and garbage collection.

Using the proper subset of hardware description language, a program called a synthesizer, or logic synthesis tool, can infer hardware logic operations from the language statements and produce an equivalent netlist of generic hardware primitives[jargon] to implement the specified behaviour.[citation needed] Synthesizers generally ignore the expression of any timing constructs in the text. Digital logic synthesizers, for example, generally use clock edges as the way to time the circuit, ignoring any timing constructs. The ability to have a synthesizable subset of the language does not itself make a hardware description language.

5.2 HISTORY

The first hardware description languages appeared in the late 1960s, looking like more traditional languages. The first that had a lasting effect was described in

1971 in C. Gordon Bell and Allen Newell's text Computer Structures. This text introduced the concept of register transfer level, first used in the ISP language to describe the behavior of the Digital Equipment Corporation (DEC) PDP-8.
The language became more widespread with the introduction of DEC's PDP-16 RT-Level Modules (RTMs) and a book describing their use. At least two implementations of the basic ISP language (ISPL and ISPS) followed. ISPS was well suited to describe relations between the inputs and the outputs of the design and was quickly adopted by commercial teams at DEC, as well as by a number of research teams both in the USAand among its NATO allies. The RTM products never took off commercially and DEC stopped marketing them in the mid-1980s,as new techniques and in particular very-large-scale integration (VLSI) became more popular. Separate work done about 1979 at the University of Kaiserslautern produced a language called KARL, which included design calculus language features supporting VLSI chip floor planning [jargon] and structured hardware design. This work was also the basis of KARL's interactive graphic sister language ABL. ABL was implemented in the early 1980s by the Centro Laboratory Tele-communication (CSELT) in Torino, Italy, producing the ABLED graphic VLSI design editor. In the mid-1980s, a VLSI design framework was implementedaround KARL and ABL by an international consortium funded by the Commissionof the European Union.

By the late 1970s, design using programmable logic devices (PLDs) became popular, although these designs were primarily limited to designing finite state machines. The work at Data General in 1980 used these same devices to design the Data General Eclipse MV/8000, and commercial need began to grow for a language that could map well to them. By 1983 Data I/O introduced ABEL to fill that need.

As design shifted to VLSI, the first modern HDL, Verilog, was introduced by Gateway Design Automation in 1985. Cadence Design Systems later acquired the rights to Verilog-XL, the HDL simulator that would become the de facto standard of Verilog simulators for the next decade. In 1987, a request from the U.S. Department of Defense led to the development of VHDL (VHSIC Hardware Description Language). VHDL was based on the Ada programming language, as well as on the experience gained with the earlier development of ISPS. Initially, Verilog and VHDL were used to document and simulate circuit designs already captured and described in another form (such as schematic files). HDL simulation enabled engineers to work at a higher level of abstraction than simulation at the schematic level, and thus increased design capacity from hundreds of transistors to thousands.

The introduction of logic synthesis for HDLs pushed HDLs from the background into the foreground of digital design. Synthesis tools compiled HDL source files (written in a constrained format called RTL) into a manufacturable netlist description in terms of gates and transistors. Writing synthesizable RTL files required practice and discipline on the part of the designer; compared to a traditional schematic layout, synthesized RTL netlists were almost always larger inarea and slower in performance[citation needed]. A circuit design from a skilled engineer, using labor-intensive schematic-capture/hand-layout, would almost always outperform its logically-synthesized equivalent, but the productivity advantage held by synthesis soon displaced digital schematic capture to exactly those areas that were problematic for RTL synthesis: extremely high-speed, low- power, or asynchronous circuitry.

Within a few years, VHDL and Verilog emerged as the dominant HDLs in the electronics industry, while older and less capable HDLs gradually disappeared from use. However, VHDL and Verilog share many of the same limitations: neitheris suitable for analog or mixed-signal circuit simulation; neither possesses language constructs to describe recursively-generated logic structures. Specialized HDLs (such as Confluence) were introduced with the explicit goal of fixing specific limitations of Verilog and VHDL, though none were ever intended to replace them. Over the years, much effort has been invested in improving HDLs. The latest iteration of Verilog, formally known as IEEE 1800-2005 SystemVerilog, introduces many new features (classes, random variables, and properties/assertions) to address the growing need for better test bench randomization, design hierarchy, and reuse. A future revision of VHDL is also in development, and is expected to match SystemVerilog's improvements.

5.3 DESIGN USING HDL

As a result of the efficiency gains realized using HDL, a majority of modern digital circuit design revolves around it. Most designs begin as a set of requirements or a high-level architectural diagram. Control and decision structures are often prototyped in flowchart applications, or entered in a state diagram editor. The process of writing the HDL description is highly dependent on the nature of the circuit and the designer's preference for coding style. The HDL is merely the 'capture language', often beginning with a high-level algorithmic description such as a C++ mathematical model. Designers often use scripting languages such as Perl to automatically generate repetitive circuit structures in the HDL language. Special text editors offer features for automatic indentation, syntax-dependent coloration, and macro-based expansion of entity/architecture/signal declaration.

The HDL code then undergoes a code review, or auditing. In preparation for synthesis, the HDL description is subject to an array of automated checkers. The checkers report deviations from standardized code guidelines, identify potential ambiguous code constructs before they can cause misinterpretation, and check for common logical coding errors, such as dangling[jargon] ports or shorted outputs. This process aids in resolving errors before the code is synthesized.

In industry parlance, HDL design generally ends at the synthesis stage. Once the synthesis tool has mapped the HDL description into a gate netlist, the netlist is passed off to the back-end stage. Depending on the physical technology (FPGA, ASIC gate array, ASIC standard cell), HDLs may or may not play a significant role in the back-end flow. In general, as the design flow progresses toward a physically realizable form, the design database becomes progressively more laden with technology-specific information, which cannot be stored in a generic HDL description. Finally, an integrated circuit is manufactured or programmed for use.

5.4 SPARTAN 3

The Spartan-3 family of Field-Programmable Gate Arrays is specifically designed to meet the needs of high volume, cost-sensitive consumer electronic applications. The eight-member family offers densities ranging from 50,000 to five million system gates. The Spartan-3 family builds on the success of the earlier Spartan-IIE family by increasing the amount of logic resources, the capacity of internal RAM, the total number of I/Os, and the overall level of performance as well as by improving clock management functions.

These Spartan-3 FPGA enhancements, combined with advanced process technology, deliver more functionality and bandwidth per dollar than was previously possible, setting new standards in the programmable logic industry.

Because of their exceptionally low cost, Spartan-3 FPGAs are ideally suited to a wide range of consumer electronics applications including broadband access, home networking, display/ projection and digital television equipment. The Spartan-3 family is a superior alternative to mask programmed ASICs. FPGAs avoid the high initial cost, the lengthy development cycles, and the inherent inflexibility of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary, an impossibility with ASICs.

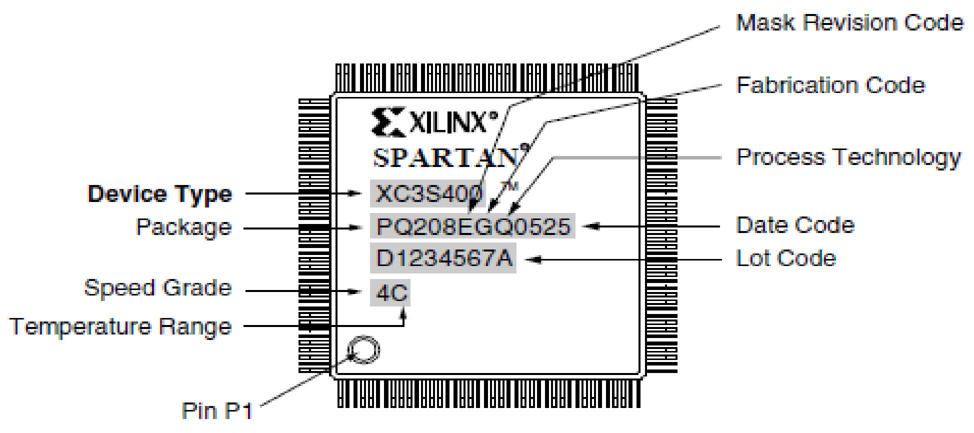


Figure 5.1 Spartan-3 Package XC3S400-4PQ208C

5.5 ARCHITECTURAL OVERVIEW

The Spartan-3 family architecture consists of five fundamental programmable functional elements:

1. Configurable Logic Blocks (CLBs) contain RAM-based Look-Up Tables (LUTs) to implement logic and storage elements that can be used as flip-flops or latches. CLBs can be programmed to perform a wide variety of logical functions as well as to store data.

2. Input/ Output Blocks (IOBs) control the flow of data between the I/O pins and the internal logic of the device. Each IOB supports bidirectional data flow plus 3-state operation. Twenty-six different signal standards, including eight high-performance differential standards are available. Double Data-Rate (DDR) registers are included. The Digitally Controlled Impedance (DCI) feature provides automatic on-chip terminations, simplifying board designs.

3.Block RAM provides data storage in the form of 18-K bit dual-port blocks.

4. Multiplier blocks accept two 18-bit binary numbers as inputs and calculate the product.

5. Digital Clock Manager (DCM) blocks provide self-calibrating, fully digital solutions for distributing, delaying, multiplying, dividing and phase shifting clocksignals.

A ring of IOBs surrounds a regular array of CLBs. The XC3S50 has a single column of block RAM embedded in the array. Those devices ranging from the XC3S200 to the XC3S2000have two columns of block RAM. The XC3S4000 and XC3S5000 devices have four RAM columns. Each column is made up of several 18-Kbit RAM blocks; each block is associated with a dedicated multiplier.

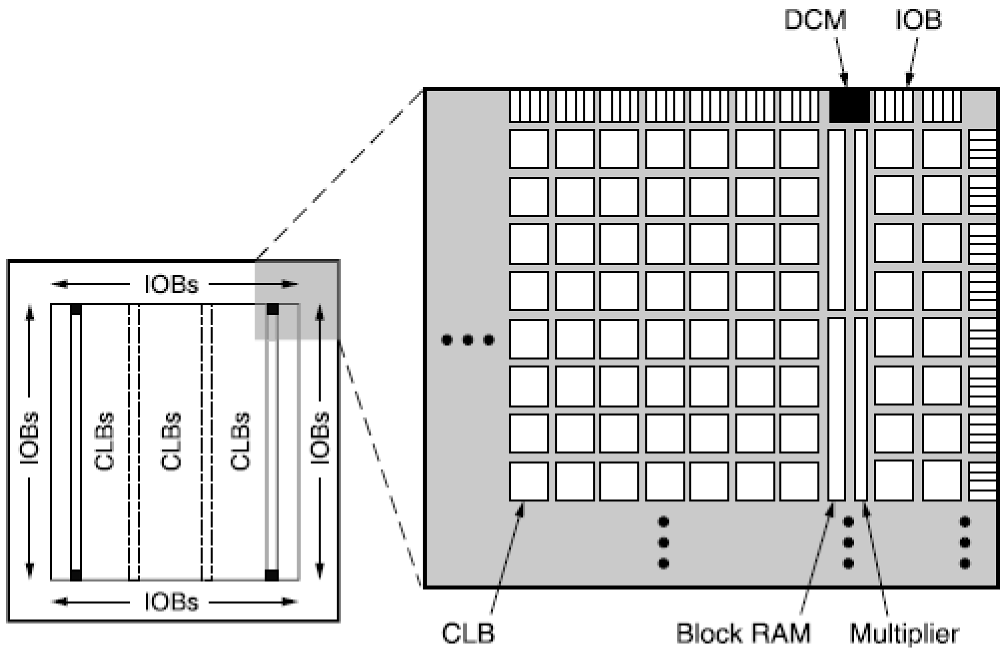


Figure 5.2 Spartan-3 Architecture

The DCMs are positioned at the ends of the outer block RAM columns. The Spartan-3 family features a rich network of traces and switches that interconnect all five functional elements, transmitting signals among them. Each functional element has an associated switch matrix that permits multiple connections to the routing.

5.6 CONFIGURABLE LOGIC BLOCK

The Configurable Logic Blocks (CLBs) constitute the main logic resourcefor implementing synchronous as well as combinatorial circuits. Each CLB comprises four interconnected slices as shown in Fig.5.3. These slices are grouped in pairs. Each pair is organized as a column with an independent carry chain.

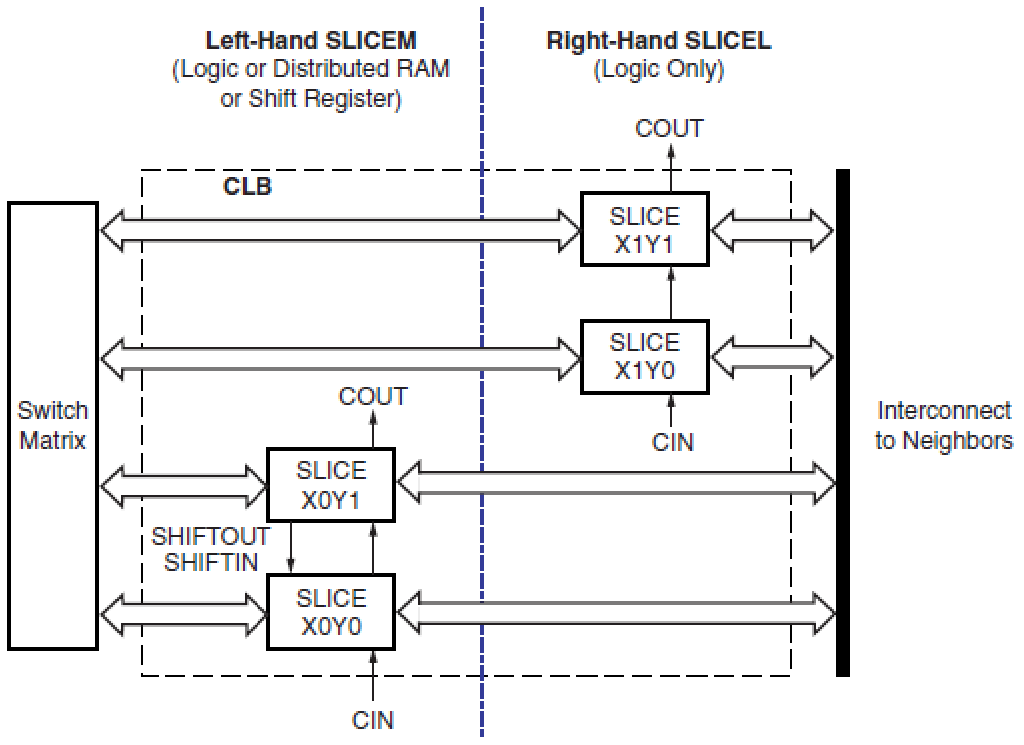


Figure 5.3 Arrangement of Slices within the CLB

The nomenclature that the FPGA Editor part of the Xilinx development software uses to designate slices is as follows: The letter ‘X’ followed by a number identifies columns of slices. The ‘X’ number counts up in sequence from the left side of the die to the right. The letter ‘Y’ followed by a number identifies the position of each slice in a pair as well as indicating the CLB row. The ‘Y’ number counts slices starting from the bottom of the die according to the sequence: 0, 1, 0, 1 (the first CLB row); 2, 3, 2, 3 (the second CLB row); etc. Fig. 5.3 shows the CLBlocated in the lower left-hand corner of the die. Slices X0Y0 and X0Y1 make up the column-pair on the left where as slices X1Y0 and X1Y1 make up the column- pair on the right. For each CLB, the term “left-hand” (or SLICEM) indicates the pair of slices labeled with an even ‘X’ number, such as X0, and the term “right- hand” (or SLICEL) designates the pair of slices with an odd ‘X’ number, e.g., X1.

CHAPTER-6

DOMAIN DESCRIPTION

Very-large-scale integration (VLSI) is the process of creating an **integrated circuit**(IC) by combining thousands of **transistors** into a single chip. VLSI began in the 1970s when complex **semiconductor** and **communication** technologies were being developed. The **microprocessor** is a VLSI device.

Before the introduction of VLSI technology, most ICs had a limited set of functions they could perform. An **electronic circuit** might consist of a **CPU**, **ROM**, **RAM** and other **glue logic**. VLSI lets IC designers add all of these into one chip.

The electronics industry has achieved a phenomenal growth over the last few decades, mainly due to the rapid advances in large scale integration technologies and system design applications. With the advent of very large scale integration (VLSI) designs, the number of applications of integrated circuits (ICs) in high- performance computing, controls, telecommunications, image and video processing, and consumer electronics has been rising at a very fast pace.

6.1 VLSI Design Flow

The VLSI IC circuits design flow is shown in the figure6.1 below. The various levels of design are numbered and the blocks show processes in the design flow. Specifications comes first, they describe abstractly, the functionality, interface, andthe architecture of the digital IC circuit to be designed.

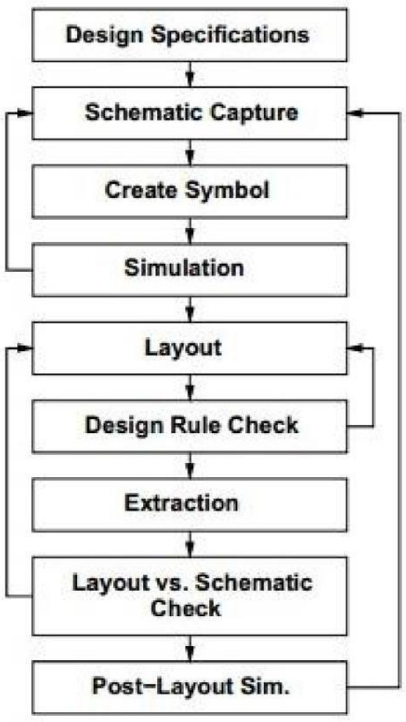


Fig6.1: Simplified VLSI Design Flow

Behavioral description is then created to analyze the design in terms of functionality, performance, compliance to given standards, and other specifications.

RTL description is done using HDLs. This RTL description is simulated to test functionality. From here onwards we need the help of EDA tools. RTL description is then converted to a gate-level netlist using logic synthesis tools. A gate-level netlist is a description of the circuit in terms of gates and connections between them, which are made in such a way that they meet the timing, power and area specifications. Finally, a physical layout is made, which will be verified and then sent to fabrication.

6.2 Y Chart

The Gajski-Kuhn Y-chart is a model, which captures the considerations in designing semiconductor devices. The three domains of the Gajski-Kuhn Y-chart are on radial axes. Each of the domains can be divided into levels of abstraction, using concentric rings. At the top level (outer ring), we consider the architecture ofthe chip; at the lower levels (inner rings), we successively refine the design into finer detailed implementation: Creating a structural description from a behavioral one is achieved through the processes of high-level synthesis or logical synthesis. Creating a physical description from a structural one is achieved through layout synthesis.

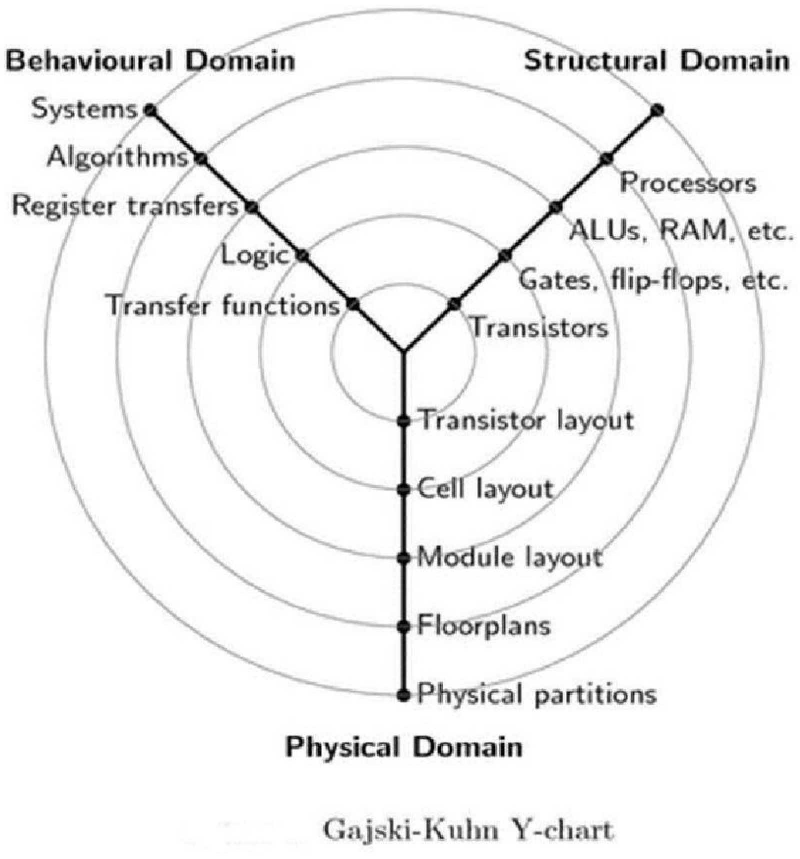


Fig 6.2: Y Chart

6.3 Design Hierarchy-Structural

The design hierarchy involves the principle of "Divide and Conquer." It is nothing but dividing the task into smaller tasks until it reaches to its simplest level. This process is most suitable because the last evolution of design has become so simple that its manufacturing becomes easier. We can design the given task into the design flow process's domain (Behavioral, Structural, and Geometrical). To understand this, let's takean example of designing a 16-bit adder, as shown in the figure6.3 below

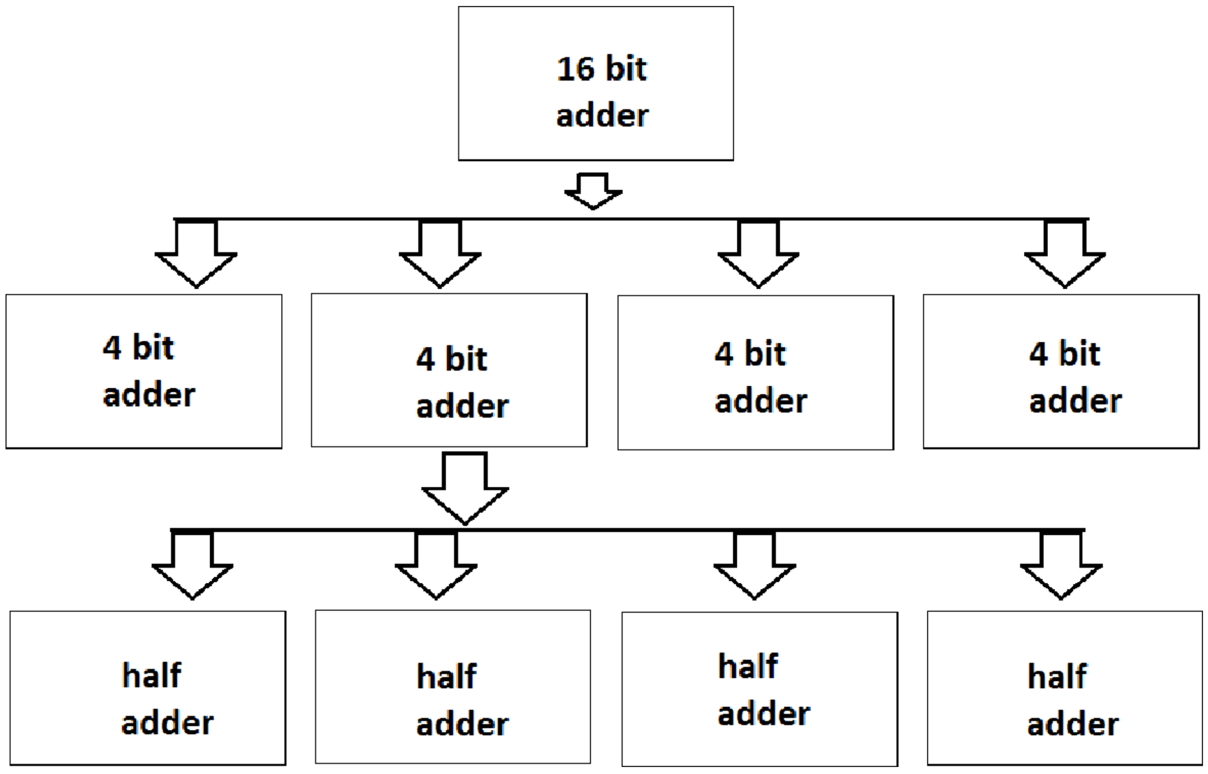


Fig 6.3: Structural hierarchy of 16 bit adder circuit

Here, the whole chip of 16 bit adder is divided into four modules of 4-bit adders. Further, dividing the 4-bit adder into 1-bit adder or half adder.

1 bit addition is the simplest designing process and its internal circuit is also easy to fabricate on the chip. Now, connecting all the last four adders, we can design a 4-bit adder and moving on, we can design a 16-bit adder.

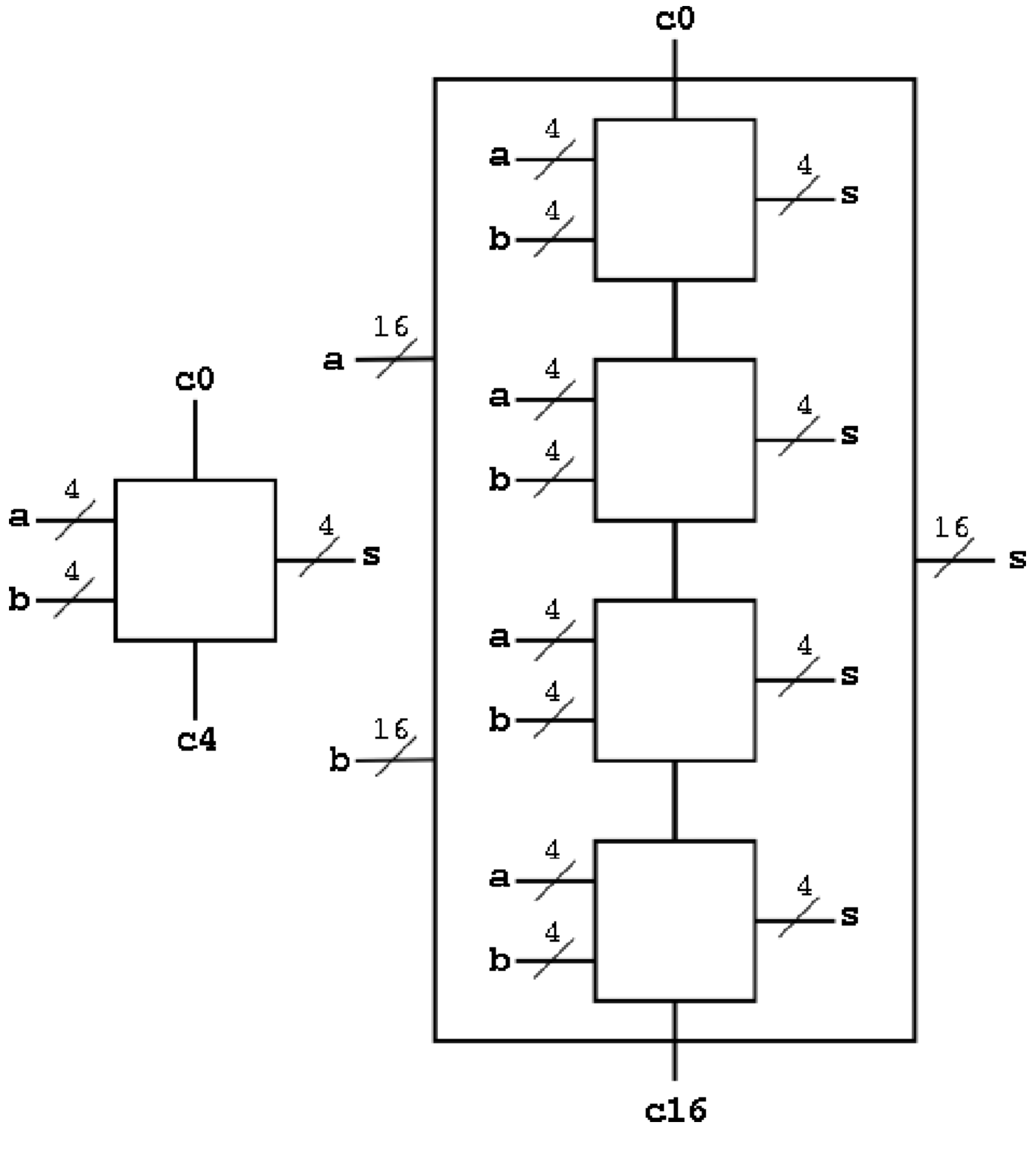


Fig 6.4: Decomposition of a 4 bit adder

CHAPTER 7

SOFTWARE PROGRAM

7.1 KARAT SUBA MULTIPLIER DESIGN CODING

```
module filteriir(clk,rst,x1,y);
input [15:0]x1;
input clk,rst;
output [15:0]y;
wire [15:0]dataout1;
wire [15:0]d11,d12,d13;
wire [15:0]m11,m12,m13,m14,m15;
wire [15:0]d111,d112,d113,d114;

wire [15:0]do11,do12,do13;
wire   [15:0]ee12,ee13,ee14,ee15;wire   [15:0]e111,e112,e113,e114;

parameter h10=16'b0000000000000001; //parametr replacd by regparameter
h11=16'b0000000000000001;
parameter h12=16'b0000000000000001;
parameter h13=16'b0000000000000001;
parameter  h14=16'b0000000000000001;

//parametr replacd by reg
parameter e11=16'b0000000000000001;
parameter e12=16'b0000000000000001;
parameter e13=16'b0000000000000001;
parameter  e14=16'b0000000000000001;

//assign m11=x1*h10;      //always@(*)
mul g1(x1,h10,m11);
dff u12(clk,rst,x1,d111);
//assign m12=d111*h11;
mul g2(d111,h11,m12);
assign d11=m11+m12;

dff u14(clk,rst,d111,d112);
//assign m13=d112*h12;
mul g3(d112,h12,m13);
```

```
assign d12=d11+m13;

dff u16(clk,rst,d112,d113);
//assign m14=d113*h13;
mul g4(d113,h13,m14);
assign d13=d12+m14;

dff u18(clk,rst,d113,d114);
//assign m15=d114*h14;
mul g5(d114,h14,m15);
assign   dataout1=d13+m15;

//assign m11=x1*h10;      //always@(*)
dff u19(clk,rst,dataout1,e111);
assign ee12=e111*e11;
assign do11=e111+ee12;

dff u20(clk,rst,e111,e112);assign ee13=e112*e12; assign do12=do11+ee13;
dff u21(clk,rst,e112,e113);assign ee14=e113*e13; assign do13=e12+ee14;
dff u22(clk,rst,e113,e114);assign ee15=e114*e14; assign y=do13+ee15;

endmodule

module dff(clk,rst,d,q);// sub module d flipflop
input clk,rst;
input [15:0]d;
output [15:0]q;
reg [15:0]q;
always@(clk)
begin
if(rst==1)
begin
q=0;   //d11=0end
else
begin
q=d;   //d111=x1;
end
end

endmodule
module mul(i1,i3,y);
```

```
input [15:0]i1,i3;

output [31:0]y;
reg [31:0]y;
reg [31:0]r1,r2,r3,t1,t2,t3;
reg [31:0]z2,z1,z0;
always @(*)
begin

t1=i1/4'b1000;
t2=t1*10'b1111101000;
t3=i1-t2;

r1=i3/4'b1000;
r2=r1*10'b1111101000;
r3=i3-r2;

z2=t1*r1;
z0=t3*r3;
z1=(t1+t3)*(r1+r3)-z2-z0;
y=z2*(20'b11110100001001000000)+z1*10'b1111101000+z0;
end


endmodule
```

7.2 MULTIPLIER LOGIC FUNCTION CODING

```
module filteriir(clk,rst,x1,y);
input [15:0]x1;
input clk,rst;
output [15:0]y;
wire [15:0]dataout1;
wire [15:0]d11,d12,d13;
wire [15:0]m11,m12,m13,m14,m15;
wire [15:0]d111,d112,d113,d114;
wire  [15:0]do11,do12,do13;
wire [15:0]ee12,ee13,ee14,ee15;
wire  [15:0]e111,e112,e113,e114;
```

```
parameter h10=16'b0000000000000001; //parametr replacd by regparameter
h11=16'b0000000000000001;
parameter h12=16'b0000000000000001;parameter h13=16'b0000000000000001;
parameter h14=16'b0000000000000001;

//parametr replacd by reg
parameter e11=16'b0000000000000001;parameter e12=16'b0000000000000001;
parameter  e13=16'b0000000000000001;parameter e14=16'b0000000000000001;

//assign m11=x1*h10;      //always@(*)mul g1(x1,h10,m11);
dff u12(clk,rst,x1,d111);
//assign m12=d111*h11;mul g2(d111,h11,m12); assign d11=m11+m12;

dff u14(clk,rst,d111,d112);
//assign m13=d112*h12;mul g3(d112,h12,m13); assign d12=d11+m13;

dff u16(clk,rst,d112,d113);
//assign m14=d113*h13;mul g4(d113,h13,m14); assign d13=d12+m14;

dff u18(clk,rst,d113,d114);
//assign  m15=d114*h14;  mul  g5(d114,h14,m15);  assign  dataout1=d13+m15;
//assign m11=x1*h10;      //always@(*)dff u19(clk,rst,dataout1,e111);
assign ee12=e111*e11; assign do11=e111+ee12;

dff u20(clk,rst,e111,e112);assign ee13=e112*e12; assign do12=do11+ee13;
dff u21(clk,rst,e112,e113);assign ee14=e113*e13; assign do13=e12+ee14;
dff u22(clk,rst,e113,e114);assign ee15=e114*e14; assign y=do13+ee15;

endmodule

module dff(clk,rst,d,q);// sub module d flipflopinput clk,rst;
input [15:0]d; output [15:0]q;reg [15:0]q; always@(clk) begin if(rst==1) begin
q=0;   //d11=0end
else begin
q=d;   //d111=x1;

end
end
endmodule
```

```
module mul(a,b,z);input [15:0]a,b; output [15:0]z; reg [31:0]z;
reg [15:0]tem1,tem2;reg [15:-1]x,y;
reg [15:0]u;
integer t1,t2,t3,t4;integer t5,t6,t7,t8;

reg  [63:0]pp1,pp2,pp3,pp4;reg  [63:0]pp5,pp6,pp7,pp8;
reg [63:0]shift1,shift2,shift3;reg [63:0]shift4,shift5,shift6;reg [63:0]shift7;

always @(*)begin

tem1=a[15]^a[14]+a[13]^a[12]+a[11]^a[10]+a[9]^a[8]^a[7]^a[6]+a[5]^a[4]+a[3
]^a[2]+a[1]^a[0];
tem2=b[15]^b[14]+b[13]^b[12]+b[11]^b[10]+b[9]^b[8]^b[7]^b[6]+b[5]^b[4]+b[3
]^b[2]+b[1]^b[0];

if(tem1>tem2)begin

x[-1]=0;
x[0]=a[0];
x[1]=a[1];
x[2]=a[2];
x[3]=a[3];
x[4]=a[4];
x[5]=a[5];
x[6]=a[6];
x[7]=a[7];

x[8]=a[8];
x[9]=a[9];
x[10]=a[10];
x[11]=a[11];
x[12]=a[12];
x[13]=a[13];
x[14]=a[14];
x[15]=a[15];

y[-1]=0;
y[0]=b[0];
y[1]=b[1];
y[2]=b[2];
y[3]=b[3];
```

```
y[4]=b[4];
y[5]=b[5];
y[6]=b[6];
y[7]=b[7];

y[8]=b[8];
y[9]=b[9];
y[10]=b[10];
y[11]=b[11];
y[12]=b[12];
y[13]=b[13];
y[14]=b[14];
y[15]=b[15];

end

else begin

x[-1]=0;
x[0]=b[0];
x[1]=b[1];
x[2]=b[2];
x[3]=b[3];
x[4]=b[4];
x[5]=b[5];
x[6]=b[6];
x[7]=b[7];

x[8]=b[8];
x[9]=b[9];
x[10]=b[10];
x[11]=b[11];
x[12]=b[12];
x[13]=b[13];
x[14]=b[14];
x[15]=b[15];

y[-1]=0;
y[0]=a[0];
y[1]=a[1];
y[2]=a[2];
```

```
y[3]=a[3];
y[4]=a[4];
y[5]=a[5];
y[6]=a[6];
y[7]=a[7];

y[8]=a[8];
y[9]=a[9];
y[10]=a[10];
y[11]=a[11];
y[12]=a[12];
y[13]=a[13];
y[14]=a[14];
y[15]=a[15];

endend
always @(*)begin
case({y[1],y[0],y[-1]})3'b000:
begint1=0;
end 3'b001:
begint1=1;
end 3'b010:
begint1=1;
end 3'b011:
begint1=2;
end 3'b100:
begin t1=-2;
end 3'b101:
begin t1=-1;
end 3'b110:
begin t1=-1;
end 3'b111:
begint1=0;
end endcase

case({y[3],y[2],y[1]})3'b000:
begint2=0;
end 3'b001:
begint2=1;
end 3'b010:
begint2=1;
```

```
end 3'b011:
begint2=2;
end 3'b100:
begin t2=-2;
end 3'b101:
begin t2=-1;
end 3'b110:
begin t2=-1;
end 3'b111:
begin
t2=0;
end endcase

case({y[5],y[4],y[3]})3'b000:
begint3=0;
end 3'b001:
begint3=1;
end 3'b010:
begint3=1;
end 3'b011:
begint3=2;
end 3'b100:
begin t3=-2;
end 3'b101:
begin t3=-1;
end 3'b110:
begin t3=-1;
end 3'b111:
begint3=0;
end endcase

case({y[7],y[6],y[5]})3'b000:
begint4=0;
end 3'b001:
begint4=1;
end 3'b010:
begint4=1;
end 3'b011:
begin
t4=2;
end 3'b100:
```

```
begin t4=-2;
end 3'b101:
begin t4=-1;
end 3'b110:
begin t4=-1;
end 3'b111:
begint4=0;
end endcase

case({y[9],y[8],y[7]})3'b000:
begint5=0;
end 3'b001:
begint5=1;
end 3'b010:
begint5=1;
end 3'b011:
begint5=2;
end 3'b100:
begin t5=-2;
end 3'b101:
begin t5=-1;
end 3'b110:
begin t5=-1;
end  3'b111:
begint5=0;

end
endcase
case({y[11],y[10],y[9]})3'b000:
begint6=0;
end 3'b001:
begint6=1;
end 3'b010:
begint6=1;
end 3'b011:
begint6=2;
end 3'b100:
begin t6=-2;
end 3'b101:
begin t6=-1;
end 3'b110:
```

```
begin t6=-1;
end 3'b111:
begint6=0;
end
endcase

case({y[13],y[12],y[11]})3'b000:
begint7=0;
end 3'b001:
begint7=1;
end 3'b010:
begin
t7=1;
end 3'b011:
begint7=2;
end 3'b100:
begin t7=-2;
end 3'b101:
begin t7=-1;
end 3'b110:
begin t7=-1;
end 3'b111:
begint7=0;
end endcase

case({y[15],y[14],y[13]})3'b000:
begint8=0;
end 3'b001:
begint8=1;
end 3'b010:
begint8=1;
end 3'b011:
begint8=2;
end 3'b100:
begin t8=-2;
end 3'b101:
begin t8=-1;
end 3'b110:
begin t8=-1;
end 3'b111:
begint8=0;
```

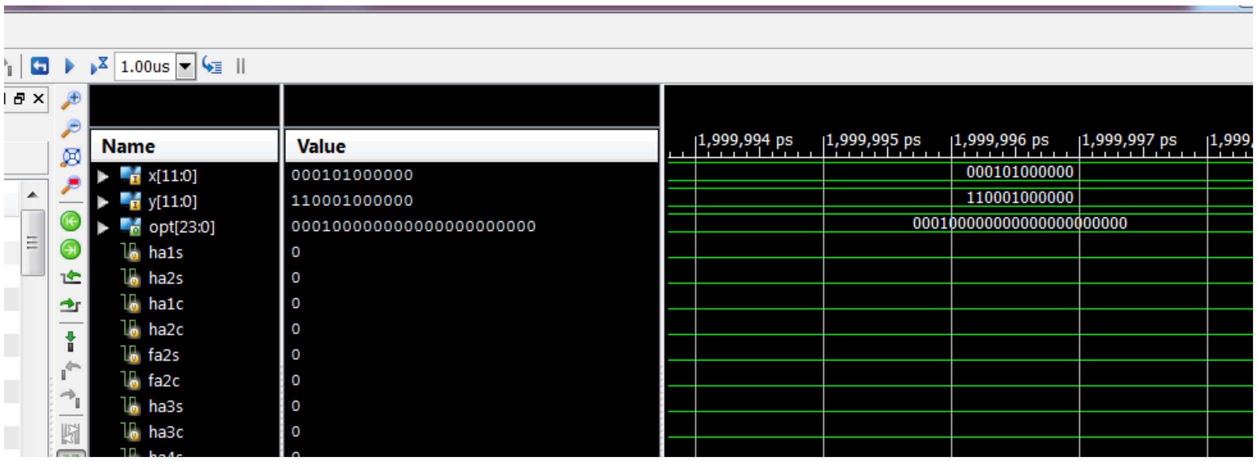
```
end endcase
end

always @(*)begin u[0]=x[0];
u[1]=x[1];
u[2]=x[2];
u[3]=x[3];
u[4]=x[4];
u[5]=x[5];
u[6]=x[6];
u[7]=x[7];

u[8]=x[8];
u[9]=x[9];
u[10]=x[10];
u[11]=x[11];
u[12]=x[12];
u[13]=x[13];
u[14]=x[14];
u[15]=x[15];
if(t1==0)begin pp1=0; end
else if(t1==1)begin
pp1=u;end
else if(t1==-1)begin
pp1=-u;end
else if(t1==2)begin
pp1=2*u;end
else if(t1==-2)begin
pp1=-2*u;
end

if(t2==0)begin pp2=0; end
else if(t2==1)begin
pp2=u;end
else if(t2==-1)begin
pp2=-u;end
else if(t2==2)begin
pp2=2*u;end
else if(t2==-2)begin
pp2=-2*u;end
```

```
if(t3==0)begin pp3=0; end
else if(t3==1)begin
pp3=u;end
else if(t3==-1)begin
pp3=-u;end
else if(t3==2)begin
pp3=2*u;end
else if(t3==-2)begin
pp3=-2*u;end

if(t4==0)begin pp4=0; end
else if(t4==1)begin
pp4=u;end
else if(t4==-1)begin
pp4=-u; end
else if(t4==2) begin
pp4=2*u; end
else if(t4==-2) begin
pp4=-2*u; end

if(t5==0) begin pp5=0; end
else if(t5==1) begin
pp5=u; end
else if(t5==-1) begin
pp5=-u; end
else if(t5==2) begin
pp5=2*u; end
else if(t5==-2) begin
pp5=-2*u; end
if(t6==0) begin pp6=0; end
else if(t6==1) begin
pp6=u; end
else if(t6==-1) begin
pp6=-u; end
else if(t6==2) begin
pp6=2*u; end
else if(t6==-2) begin
pp6=-2*u;
end
if(t7==0)begin pp7=0; end
else if(t7==1)begin
```

```
pp7=u;
end

else if(t7==-1)begin
pp7=-u;end
else if(t7==2)begin
pp7=2*u;end
else if(t7==-2)begin
pp7=-2*u;end
if(t8==0)begin pp8=0;

end
else if(t8==1)begin
pp8=u;end
else if(t8==-1)begin
pp8=-u;end
else if(t8==2)begin
pp8=2*u;end
else if(t8==-2)begin
pp8=-2*u;end
end

always @(*)
begin
shift1=pp2<<2;
shift2=pp3<<4;
shift3=pp4<<6;
shift4=pp5<<8;
shift5=pp6<<10;
shift6=pp7<<12;
shift7=pp8<<14;
z=pp1+shift1+shift2+shift3+shift4+shift5+shift6+shift7;

end

endmodule
```

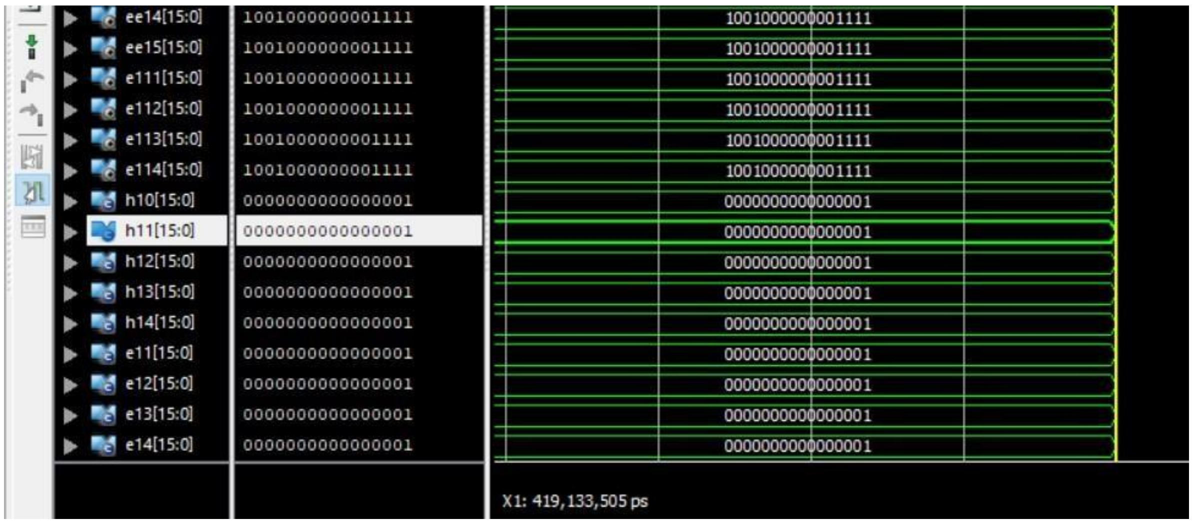
CHAPTER 8

SIMULATION RESULTS



```
Name        Value
x[11:0]     000101000000
y[11:0]     110001000000
opt[23:0]   000100000000000000000000
ha1s        0
ha2s        0
ha1c        0
ha2c        0
fa2s        0
fa2c        0
ha3s        0
ha3c        0
```

Figure 8.1: Product output



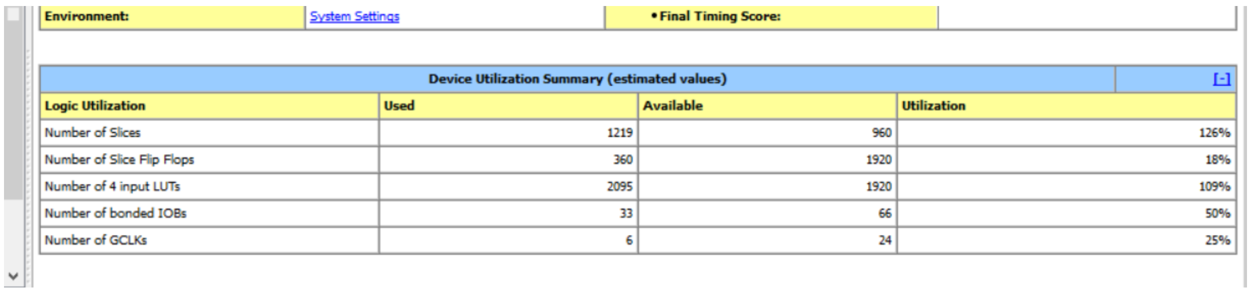
```
ee14[15:0]   1001000000001111
ee15[15:0]   1001000000001111
e111[15:0]   1001000000001111
e112[15:0]   1001000000001111
e113[15:0]   1001000000001111
e114[15:0]   1001000000001111
h10[15:0]    0000000000000001
h11[15:0]    0000000000000001
h12[15:0]    0000000000000001
h13[15:0]    0000000000000001
h14[15:0]    0000000000000001
e11[15:0]    0000000000000001
e12[15:0]    0000000000000001
e13[15:0]    0000000000000001
e14[15:0]    0000000000000001

X1: 419,133,505 ps
```

Figure 8.2: Final output

CHAPTER 9

XILINX SYNTHESIS REPORT

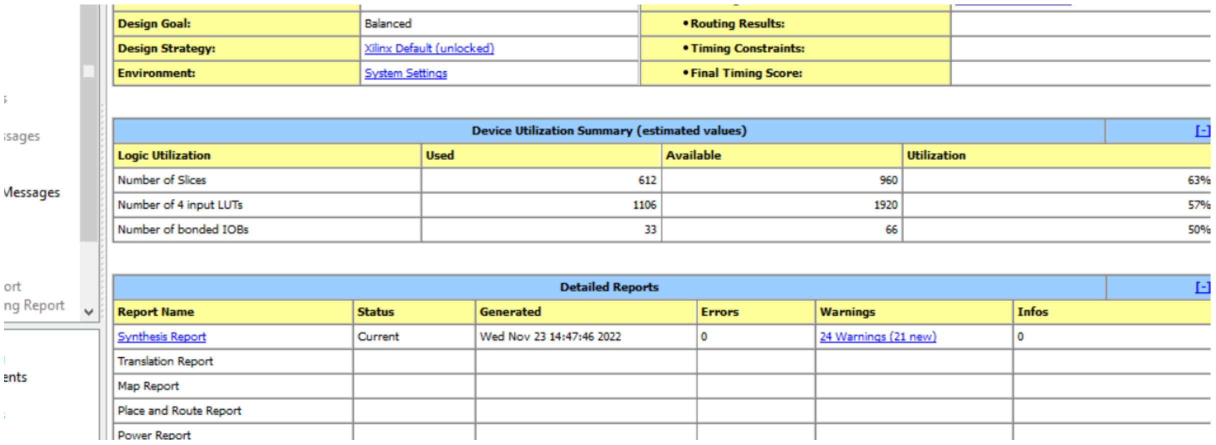
Existing system:



Environment:	System Settings	**• Final Timing Score:**	

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1219	960	126%
Number of Slice Flip Flops	360	1920	18%
Number of 4 input LUTs	2095	1920	109%
Number of bonded IOBs	33	66	50%
Number of GCLKs	6	24	25%

Proposed System:



Design Goal:	Balanced	**• Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	
Environment:	System Settings	**• Final Timing Score:**	

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	612	960	63%
Number of 4 input LUTs	1106	1920	57%
Number of bonded IOBs	33	66	50%

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Wed Nov 23 14:47:46 2022	0	24 Warnings (21 new)	0
Translation Report					
Map Report					
Place and Route Report					
Power Report					

The proposed has been simulated and the synthesis report can be obtained by using Xilinx ISE 12.1i. The various parameters used for computing existing and proposed systems with Spartan-3 processor are given in the table.

Table:1

s.no	Parameter	Existing	Proposed
1	Slice	1219	2095
2	LUT	612	1106

CHAPTER 10

RTL SCHEMATIC

After performing the synthesize process, the RTL schematic has been created automatically based on the functionality. The comparison between the different cells can be viewed clearly by this schematic.

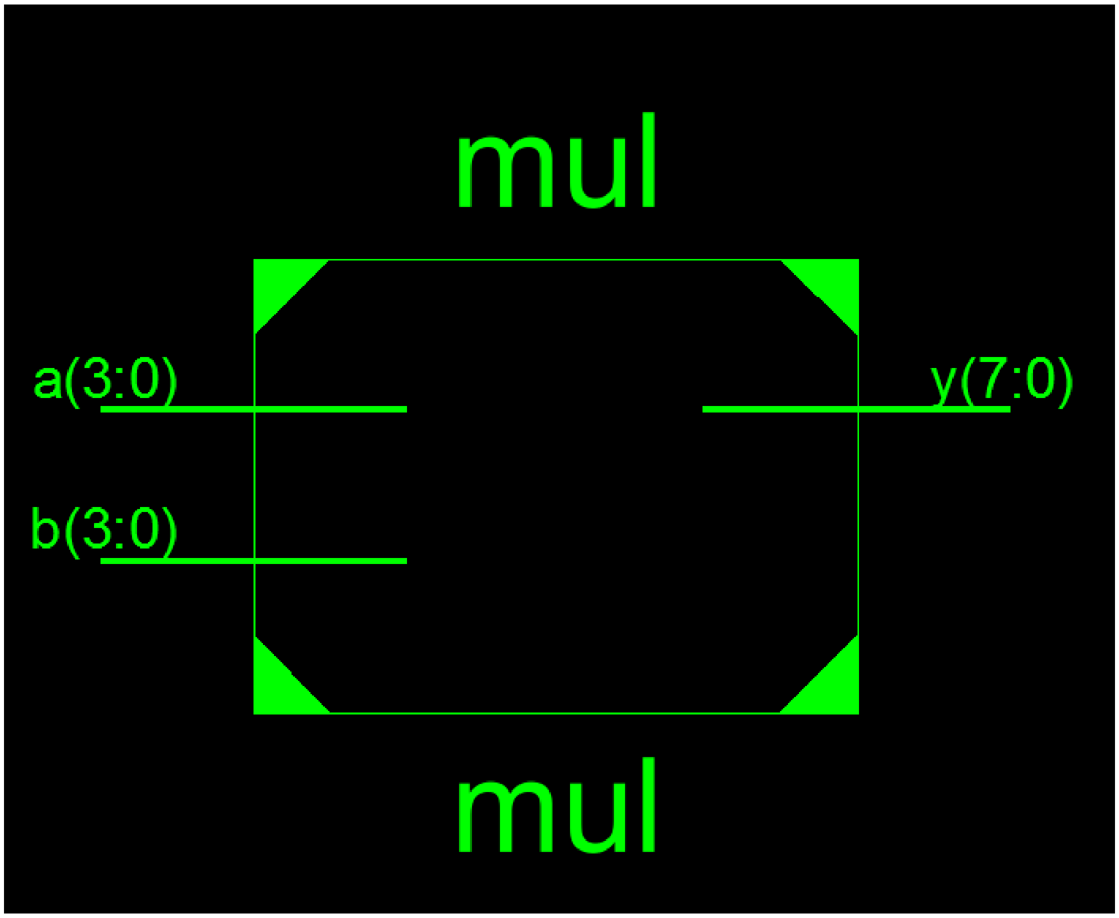


Figure 10.1: RTL Schematic

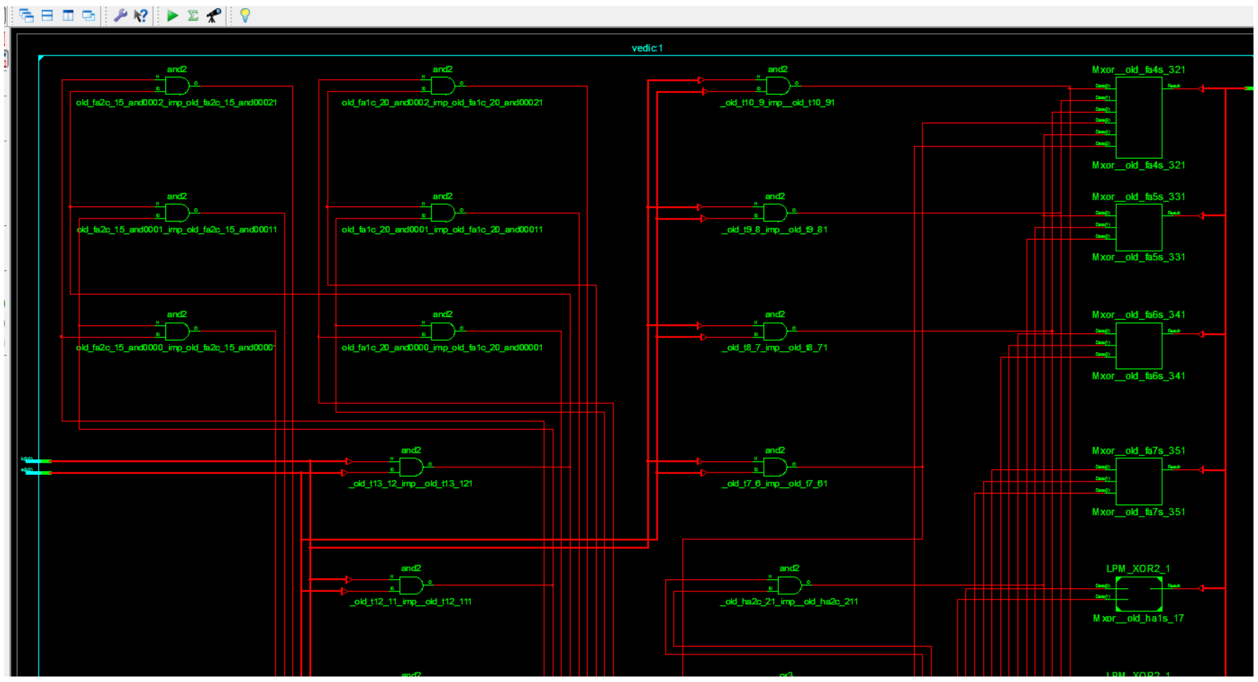

vedic:1
and2
old_fa2c_15_and0002_imp_old_fa2c_15_and00021
and2
old_fa1c_20_and0002_imp_old_fa1c_20_and00021
and2
_old_t10_9_imp__old_t10_91
Mxor__old_fa4s_321
old_fa2c_15_and0001_imp_old_fa2c_15_and00011
old_fa1c_20_and0001_imp_old_fa1c_20_and00011
_old_t9_8_imp__old_t9_81
Mxor__old_fa5s_331
old_fa2c_15_and0000_imp_old_fa2c_15_and00001
old_fa1c_20_and0000_imp_old_fa1c_20_and00001
_old_t8_7_imp__old_t8_71
Mxor__old_fa6s_341
_old_t13_12_imp__old_t13_121
_old_t7_6_imp__old_t7_61
Mxor__old_fa7s_351
_old_t12_11_imp__old_t12_111
_old_ha2c_21_imp__old_ha2c_211
LPM_XOR2_1
Mxor__old_ha1s_17
or3

Fig10. 2: Gatelevel Netlist

CHAPTER 11

PERFORMANCE ANALYSIS

The Figure11.1 given below is shown that there is a considerable reduction in time and area based on the implementation results which have been done by using Spartan-3 processor. The proposed algorithm significantly reduces area consumption when compared to the existing system.

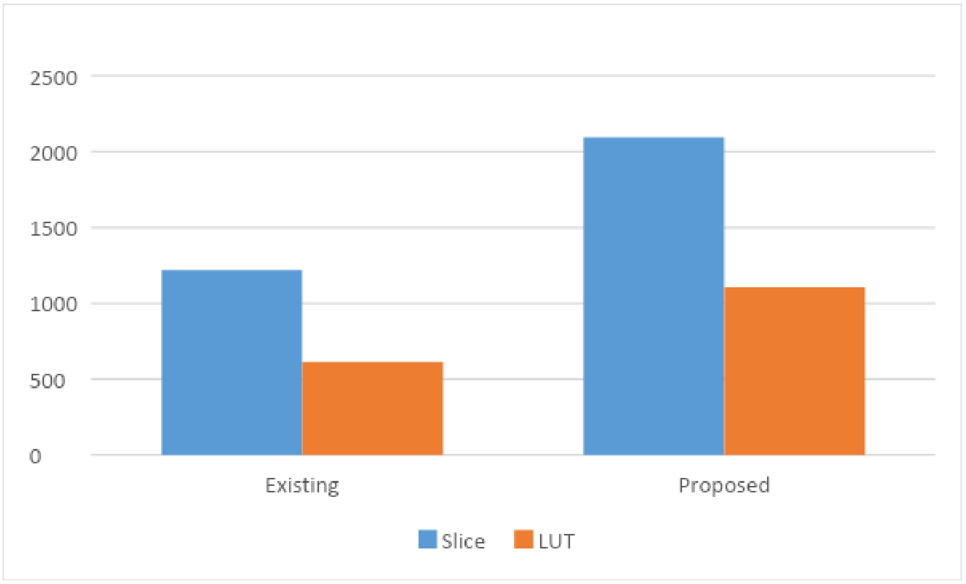


Figure 11.1 : Performance analysis

CHAPTER 12

CONCLUSION

In this project, a low-error and area-efficient fixed-width Karatsuba multiplier is presented. As compared with the state-of-the-art design, the proposed fixed-width multiplier performs not only with lower compensation error but also with lower hardware complexity, especially as multiplier input bits increase. The proposed multiplier circuit is implemented in Xilinx 12.1 The area and slice count in the proposed design are only 53% of the full-length multiplier

REFERENCES

AmritaNanda,”Design and Implementation of Urdhva-Tiryakbhyam Based Fast 8×8 Vedic Binary Multiplier”IJERT, ISSN: 2278-0181,Vol. 3 Issue 3, March -2014Poornima M, Shivaraj Kumar Patil, Shivukumar, ShridharKP,Sanjay H, “Implementation of Multiplier Using Vedic Algorithm”, JITEE, ISSN:-2278-3075,Volume-2, Issue-6,May-2013.

Premananda B.S, Samarth S. Pai, Shashank B, ShashankS.Bhat, “Design and Implementation of 8-bit Vedic Multiplier”, IJAREEIE, Vol.2, Issue 12, ISSN: 2320-3765, Dec-2013.

Anju& V.K. Agrawal,”FPGA Implementation of Low Power and High Speed Vedic Multiplier using Vedic Mathematics”, IOSR-JVSP , e-ISSN: 2319 – 4200 ,2, Issue 5 (May. – Jun. 2013), PP 51-57 Booth, A.D., “A signed binary multiplication technique,” Quarterly Journal of Mechanics and Applied Mathematics, vol. 4, pt. 2, pp. 236– 240, 1951 Jagadguru,Swami Sri Bharath, KrsnaTirathji, “Vedic Mathematics or Sixteen Simple Sutras From The Vedas”, MotilalBanarsidas, Varanasi(India),1986

Mrs. M. Ramalatha, Prof. D. Sridharan, “VLSI Based High Speed KaratsubaMultiplier for Cryptographic Applications Using Vedic Mathematics”, IJSCI, 2007.

L. Ciminiera and A. Valenzano, "Low cost serial multipliers for high speed specialised processors," Computers and Digital Techniques, IEEE Proc., vol. 135.5,1988, pp. 259-265.

P. Verma, K.K. Mehta, "Implementation of an efficient multiplier based on Vedic Mathematics using EDA Tool", International Journal of Engineering and Advance Technology (IJEAT) ISSN : 1 (5), 2012, 2249-8958.

Harpreet Singh Dhillon and AbhijitMitra, “A Reduced– BitMultipliction Algorithm for Digital Arithmetics”, International Journal of Computational and Mathematical Sciences 2.2 @ www.waset.orgSpring2008

P. D. Chidgupkar and M. T. Karad, “The Implementation of Vedic Algorithms in Digital Signal Processing”, Global J. of Engg. Edu, Vol.8, No.2, 2004, UICEE Published in Australia

Vahdat, Shaghayegh; Kamal, Mehdi; Afzali-Kusha, Ali; Pedram, Massoud (2019). *TOSAM: An Energy-Efficient Truncation- and Rounding-Based Scalable Approximate Multiplier. IEEE Transactions on Very Large Scale Integration (VLSI)*
Systems, (), 1–13. doi:10.1109/TVLSI.2018.2890712

cui, xiaoping; Liu, Weiqiang; chen, Xin; Swartzlander, Earl; Lombardi, Fabrizio (2015). *A Modified Partial Product Generator for Redundant Binary Multipliers. IEEE Transactions on Computers, (), 1–1.*

Li, Yin; Zhang, Yu; He, Wei (2020). *Fast Hybrid Karatsuba Multiplier for Type II Pentanomials. IEEE Transactions on Very Large Scale Integration (VLSI) Systems,28(11), 2459–2463.*

Tsoumanis, Konstantinos; Axelos, Nikos; Moschopoulos, Nikos;Zervakis, Georgios; Pekmestzi, Kiamal (2015). *Pre-Encoded Multipliers Based on Non-Redundant Radix-4 Signed-Digit Encoding. IEEE Transactions on Computers,(), 1–1.*

Wey, I-Chyn; Peng, Chien-Chang; Liao, Feng-Yu (2015). *Reliable Low-Power Multiplier Design Using Fixed-Width Replica Redundancy Block. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 23(1), 78–87.*

E. Abdulrahman; A. Reyhani-masoleh (2015). *High-Speed Hybrid-Double Multiplication Architectures Using New Serial-Out Bit-Level Mastrovito Multipliers. , (), –.*

Chen, Yuan-Ho (2020). *Improvement of Accuracy of Fixed-Width Booth Multipliers Using Data Scaling Technology. IEEE Transactions on Circuits and Systems II: Express Briefs, (), 1–1.*

Gorgin, Saeid; Jaberipur, Ghassem (2016). *Sign-Magnitude Encoding for Efficient VLSI Realization of Decimal Multiplication. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, (), 1–13.*

Young-Ho Seo; Dong-Wook Kim (2010). *A New VLSI Architecture of Parallel Multiplier–Accumulator Based on Radix-2 Modified Booth Algorithm. , 18(2),201–208.*

Fritz, Christopher; Fam, Adly (2015). *The Interlaced Partition Multiplier. IEEE Transactions on Computers, (), 1–1.*

Wang, Wei; Huang, Xinming; Emmart, Niall; Weems, Charles (2014). *VLSI Design of a Large-Number Multiplier for Fully Homomorphic Encryption. IEEETransactions on Very Large Scale Integration (VLSI) Systems, 22(9), 1879–1887.*

Shrestha, Rahul; Rastogi, Utkarsh (2016). *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID) - Design and Implementation of Area-Efficient and Low-Power Configurable Booth-Multiplier. , (), 599–600.*

Selvakumar, J.; Bhaskar, V.C. (2012). *IET Chennai 3rd International Conference on Sustainable Energy and Intelligent Systems (SEISCON 2012) - Low power and area optimized truncated multiplier architecture. , (), 166–171.*

Eriksson, H.; Larsson-Edefors, P.; Eckerbert, D. (2006). *Toward architecture-based test-vector generation for timing verification of fast parallel multipliers. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 14(4), 370–379.*

Shanmuga Priya, N; Radha, N (2020). *[IEEE 2020 International Conference on Inventive Computation Technologies (ICICT) - Coimbatore, India (2020.2.26-2020.2.28)] 2020 International Conference on Inventive Computation Technologies (ICICT) - 4x4 Multiplier Implementation using Gate Diffusion Input. , (), 960–965.*