Project report

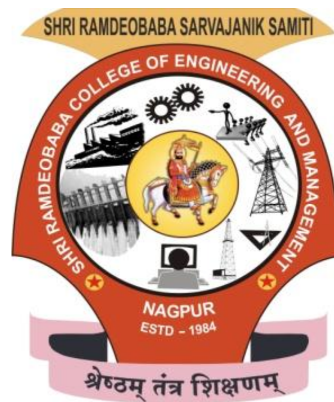
On

Multiplication Algorithms Using VHDL

This project report is submitted

In fulfillment of the requirement for the award of degree of

"B.E. Electronics and Communication"



Submitted by

Chaitanya Kukde (59)
Navoneel Das Gupta (78)
Rituraj Kawale (85)
Yash Patkar (93)

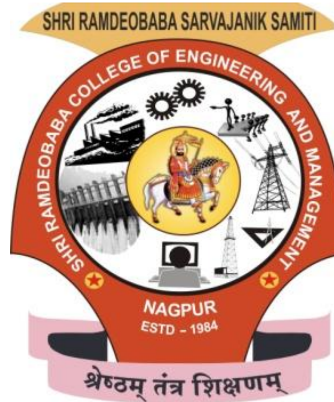
Under the guidance of

Prof.Anish Goel

Department Of Electronics and Communication,
Shri Ramdeobaba College Of Engineering and Management,
Nagpur-440013
(An Autonomous College Of Rashtrasant Tukadoji Maharaj Nagpur University)

2014-2015

Shri Ramdeobaba College Of Engineering and Management, Nagpur-440013
(An Autonomous College Of Rashtrasant Tukadoji Maharaj Nagpur University)
2014-2015



Department Of Electronics And Communication

Certificate

This is to certify that the project report titled **Implementation of Multiplication Algorithms using VHDL** is a bonafide work done by the following students as a partial fulfillment of requirement for the award of Degree in Bachelors Of Electronics and Communications Engineering of R.T.M.N.U University for the year 2013-2014.

Submitted by
Chaitanya Kukde (59)
Navoneel Das Gupta (78)
Rituraj Kawale (85)
Yash Patkar (93)

Prof. Anish Goel	Dr. S. B. Pokle	Dr. R. S. Pande
(Project guide)	(H.O.D-EC)	(Principal)

ACKNOWLEDGEMENT

It is indeed a pleasure for us to express our deep sense of gratitude towards our project guides Prof. Anish Goel whose enthusiasm was a source of inspiration for us . It is because of her , that we could synchronise our efforts in covering the manifold facets of our project .

We express our heartfelt thanks to Dr S. B. Pokle, HOD EC Dept., who ensured that we completed this project smoothly by providing us access to the lab , after college hours , and also helping us with certain technical aspects of the project .

Lastly, our deep regards to all those who directly or indirectly helped us the completion of the project.

PROJECTEES

Chaitanya Kukde
Navoneel Das Gupta
Rituraj Kawale
Yash Patkar

INDEX

Chapter 1: Introduction

1. Introduction Multipliers
2. VHDL
3. Aim and Objective

Chapter 2: Components

1. Half adder
2. Full adder
3. Subtractor

Chapter 3: Software

1.Modelsim
   a. Features
   b. How to use Modelsim
2. Xilinx ISE
   a. User Interface
   b. Simulation
   c. Synthesis

Chapter 4: Algorithms

1. Wallace Tree Multiplier
2. Karatsuba Multiplier

Chapter 5: **VHDL Codes**

1. WALLACE TREE MULTIPLIER
   a. Main Multiplier
   b. Full Adder
   c. Half Adder
2. KARATSUBA TREE MULTIPLIER
   a. Main Multiplier
   b. 4-bit Ripple Carry Adder
   c. Component for concatenating carries
   d. 4-bit Multiplier
   e. 5-bit multiplier
   f. Full Adder
   g. 16-bit adder
   h. 10-bit subtractor

Chapter 6: APPLICATIONS

REFERENCES

ABSTRACT

A Binary multiplier is an integral part of the arithmetic logic unit (ALU) subsystem found in many processors. Integer multiplication can be inefficient and costly, in time and hardware, depending on the representation of numbers. Karatsuba algorithm and others like Wallace-Tree suggest techniques for multiplying signed numbers that works equally well for efficient multiplication.

In this project, we have used VHDL as a HDL and Mentor Graphics Tools (MODEL-SIM) for describing and verifying a hardware design based on Booth's and some other efficient algorithms. Instead of writing TestBenches & Test-Cases we used Wave-Form Analyzer which can give a better understanding of Signals & variables and also proved a good choice for simulation of design.

CHAPTER 1

Introduction

Although computer arithmetic is sometimes viewed as a specialized part of CPU design, still the discrete component designing is also a very important aspect. A tremendous variety of algorithms have been proposed for use in floating-point systems. Actual implementations are usually based on refinements and variations of the few basic algorithms presented here. In addition to choosing algorithms for addition, subtraction .multiplication, and division, the computer architect must make other choices.

Multipliers play an important role in today's digital signal processing and various other applications in high performance systems such as microprocessor, DSP etc addition and multiplication of two binary numbers is fundamental and most often used arithmetic operations. Statics shows that more than 70% instructions in microprocessor and most of DSP algorithms perform addition and multiplication. So, this operation dominates the execution time. That's why there is need of high speed multiplier. The demand of high speed processing has been increasing as a result of expanding computer and signal processing applications. Low power consumption is also an important issue in multiplier design. To reduce significant power consumption it is good to reduce the number of operation thereby reducing dynamic power which is a major part of total power consumption so the need of high speed and low power multiplier has increased. Designers mainly concentrate on high speed and low power efficient circuit design. The objective of a good multiplier is to provide a physically packed together, high speed and low power consumption unit.

Multiplications are very expensive and slow the overall operation. The performances 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.

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-bit adder. 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 the result 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 multi-operand addition is applied to compute the final product. So the adder unit is very important for designing any multiplier

An efficient multiplier should have following characteristics:-

- Accuracy:- A good multiplier should give correct result.
- Speed:- Multiplier should perform operation at high speed.
- Area:- A multiplier should occupies less number of slices and LUTs. Power: - Multiplier should consume less power.

There are different types of multiplier such as:-

1. Booth multiplier.
2. Combinational multiplier.
3. Wallace tree multiplier.
4. Array multiplier.
5. Sequential multiplier.

VHDL

The VHSIC (very high speed integrated circuits) Hardware Description Language (VHDL) was first proposed in 1981. The development of VHDL was originated by IBM, Texas Instruments, and Inter-metrics in 1983. The result, contributed by many participating EDA (Electronics Design Automation) groups, was adopted as the IEEE 1076 standard in December 1987.

VHDL is intended to provide a tool that can be used by the digital systems community to distribute their designs in a standard format. Using VHDL, they are able to talk to each other about their complex digital circuits in a common language without difficulties of revealing technical details.

As a standard description of digital systems, VHDL is used as input and output to various simulation, synthesis, and layout tools. The language provides the ability to describe systems, networks, and components at a very high behavioral level as well as very low gate level. It also represents a top-down methodology and environment.

Simulations can be carried out at any level from a generally functional analysis to a very detailed gate-level wave form analysis.

Many DSP applications demand high throughput and real-time response, performance constraints that often dictate unique architectures with high levels of concurrency. DSP designers need the capability to manipulate and evaluate complex algorithms to extract the necessary level of concurrency.

Performance constraints can also be addressed by applying alternative technologies. A change at the implementation level of design by the insertion of a new technology can often make viable an existing marginal algorithm or architecture. The VHDL language supports these modeling needs at the algorithm or behavioral level, and at the implementation or structural level. It provides a versatile set of description facilities to model DSP circuits from the system level to the gate level. Recently, we have also noticed efforts to include circuit-level modeling in VHDL. At the system level we can build behavioral models to describe algorithms and architectures. We would use concurrent processes with constructs common to many high-level languages, such as if, case, loop, wait, and assert statements. VHDL also includes user-defined types, functions, procedures, and packages." In many respects VHDL is a very powerful, high-level, concurrent programming language.

At the implementation level we can build structural models using component instantiation statements that connect and invoke subcomponents. The VHDL generate statement provides ease of block replication and control. A dataflow level of description offers a combination of the behavioral and structural levels of description. VHDL lets us use all three levels to describe a single component. Most importantly, the standardization of VHDL has spurred the development of model libraries and design and development tools at every level of abstraction. VHDL, as a consensus description language and design environment, offers design tool portability, easy technical exchange, and technology insertion

Aim and Objective

The aim of this project is to design and implement the multiplier that takes less time using fast multiplication techniques. In today's scenario the multipliers with advance features are available that consume less power and less area. There has been extensive work on low-power multipliers at technology, physical, circuit and logic levels. The systems performance is based on speed, power and the area. Hence optimizing the speed is major design issue.

In this project, the implementation of multiplication using FPGA-Based computing platform will be done. Because the highly parallel nature of matrix multiplication it makes an ideal application for using such platform. The computations are done in parallel by multipliers and adders. In our approach, we will adopt the software/hardware co-design. Our multiplier will be modeled in VHDL.

The purely software implementation of matrix multiplication will be accomplished. Observation of the matrix multiplication equations shows that the multiplications can be performed concurrently, and then the additions can be performed concurrently. This parallelism can be exploited to increase processing speed.

There are various techniques or we can say algorithms that can be used for fast multiplication process they as follows:-

CHAPTER 2

Components

In electronics, an adder is a digital circuit that performs addition of numbers. In modern computers adders reside in the arithmetic logic unit (ALU) where other operations are performed. Although adders can be constructed for many numerical representations, such as Binary-coded decimal or excess-3, the most common adders operate on binary numbers. In cases where two's complement is being used to represent negative numbers it is trivial to modify an adder into an adder-subtracter

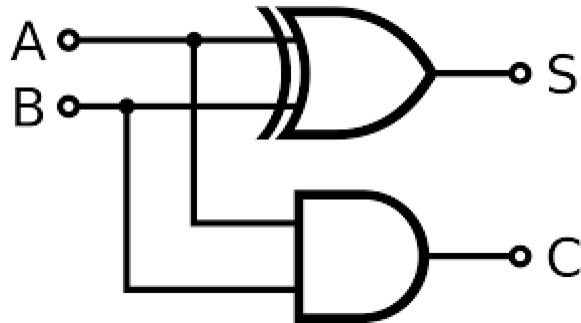
Types of adders

For single bit adders, there are two general types.

A half adder has two inputs, generally labeled A and B, and two outputs, the sum S and carry C. S is the two-bit XOR of A and B, and C is the AND of A and B. Essentially the output of a half adder is the sum of two one-bit numbers, with C being the most significant of these two outputs. The second type of single bit adder is the full adder.

The full adder takes into account a carry input such that multiple adders can be used to add larger numbers. To remove ambiguity between the input and output carry lines, the carry in is labeled Ci or Cin while the carry out is labeled Co or Cout.

Half adder



A half adder is a logical circuit that performs an addition operation on two binary digits. The half adder produces a sum and a carry value which are both binary digits.

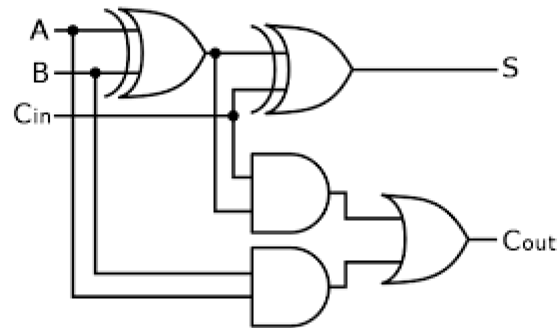
S= A XOR B;

C=A AND B;

Input		Output	
A	*B*	*C*	*S*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth table

Full adder



Inputs: {A, B, Carry In} → Outputs: {Sum, Carry Out}

A full adder is a logical circuit that performs an addition operation on three binary digits. The full adder produces a sum and carries value, which are both binary digits. It can be combined with other full adders (see below) or work on its own.

Input bit for number A	Input bit for number B	Carry bit input C_{IN}	Sum bit output S	Carry bit output C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Note that the final OR gate before the carry-out output may be replaced by an XOR gate without altering the resulting logic. This is because the only discrepancy between OR and XOR gates occurs when both inputs are 1; for the adder shown here, one can check this is never possible. Using only two types of gates is convenient if one desires to implement the adder directly using common IC chips. A full adder can be constructed from two half adders by connecting A and B to the input of one half adder, connecting the sum from that to an input to the second adder, connecting Ci to the other input and or the two carry outputs. Equivalently, S could be made the three-bit xor of A, B, and Ci and Co could be made the three-bit majority function of A, B, and Ci . The output of the full adder is the two-bit arithmetic sum of three one-bit numbers

Subtractor

The full-subtractor is a combinational circuit which is used to perform subtraction of three bits. It has three inputs, X (minuend) and Y (subtrahend) and Z (subtrahend) and two outputs D (difference) and B (borrow).

D=X-Y-Z (don't bother about sign)

B = 1 If X<(Y+Z)

The truth table for the full subtractor is given below.

X	Y	Z	D	B
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

So, Logic equations are:

$$D = (X \oplus Y) \oplus Z$$

$$B = \overline{X} \cdot (Y \oplus Z) + Y \cdot Z$$

CHAPTER 3

SOFTWARE

ModelSim

Mentor Graphics was the first to combine single kernel simulator (SKS) technology with a unified debug environment for Verilog, VHDL, and SystemC. The combination of industry-leading, native SKS performance with the best integrated debug and analysis environment make ModelSim® the simulator of choice for both ASIC and FPGA designs. The best standards and platform support in the industry make it easy to adopt in the majority of process and tool flows.

Features:

Advanced Code Coverage:

ModelSim's advanced code coverage capabilities and ease of use lower the barriers for leveraging this valuable verification resource.

The ModelSim advanced code coverage capabilities provide valuable metrics for systematic verification. All coverage information is stored in the Unified Coverage DataBase (UCDB), which is used to collect and manage all coverage information in a highly efficient database. Coverage utilities that analyze code coverage data, such as merging and test ranking, are available. Coverage results can be viewed interactively, post-simulation, or after a merge of multiple simulation runs. Code coverage metrics can be reported by instance or by design unit, providing flexibility in managing coverage data.

The coverage types supported include:

- Statement coverage: number of statements executed during a run
- Branch coverage: expressions and case statements that affect the control flow of the HDL execution
- Condition coverage: breaks down the condition on a branch into elements that make the result true or false
- Expression coverage: the same as condition coverage, but covers concurrent signal assignments instead of branch decisions
- Focused expression coverage:presents expression coverage data in a manner that accounts for each independent input to the expression in determining coverage results
- Enhanced toggle coverage: in default mode, counts low-to-high and high-to-low transitions; in extended mode, counts transitions to and from X
- Finite State Machine coverage: state and state transition coverage

Mixed HDL Simulation

ModelSim combines simulation performance and capacity with the code coverage and debugging capabilities required to simulate multiple blocks and systems and attain ASIC gate-level sign-off. Comprehensive support of Verilog, System Verilog for Design, VHDL, and SystemC provide a solid foundation for single and multi-language design verification environments. ModelSim's easy to use and unified debug and simulation environment provide today's FPGA designers both the advanced capabilities that they are growing to need and the environment that makes their work productive..

Effective Debug Environment:

The ModelSim debug environment's broad set of intuitive capabilities for Verilog, VHDL, and SystemC make it the choice for ASIC and FPGA design.

ModelSim eases the process of finding design defects with an intelligently engineered debug environment. The ModelSim debug environment efficiently displays design data for analysis and debug of all languages.

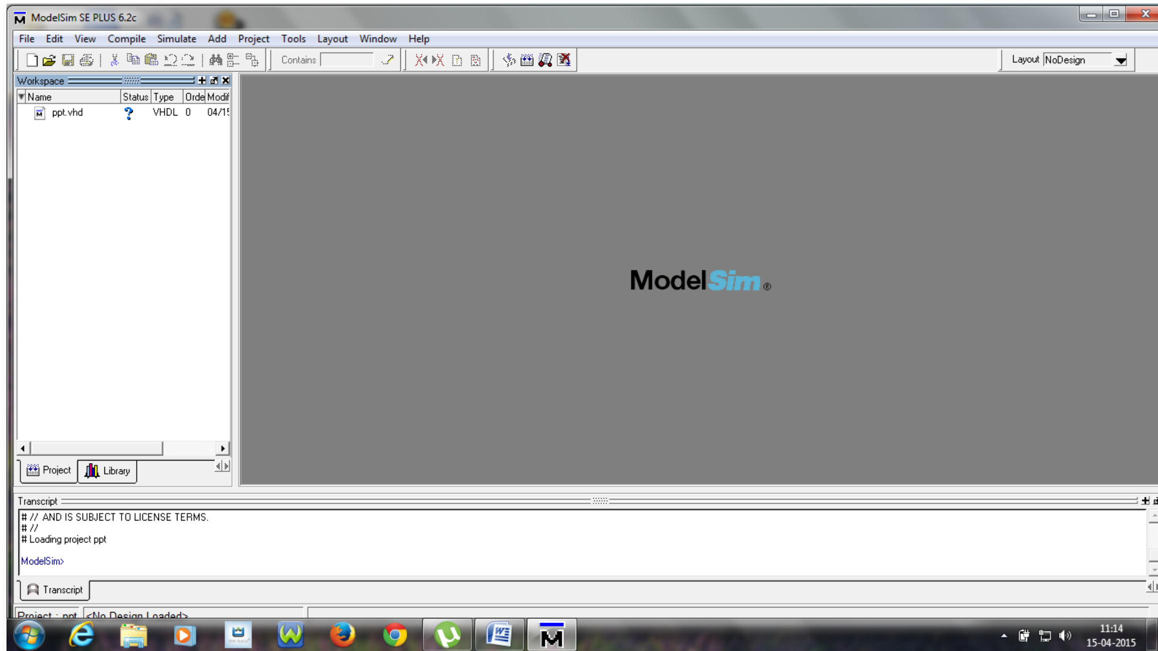
ModelSim allows many debug and analysis capabilities to be employed post-simulation on saved results, as well as during live simulation runs. For example, the coverage viewer analyzes and annotates source code with code coverage results, including FSM state and transition, statement, expression, branch, and toggle coverage.

Signal values can be annotated in the source window and viewed in the waveform viewer, easing debug navigation with hyperlinked navigation between objects and its declaration and between visited files.

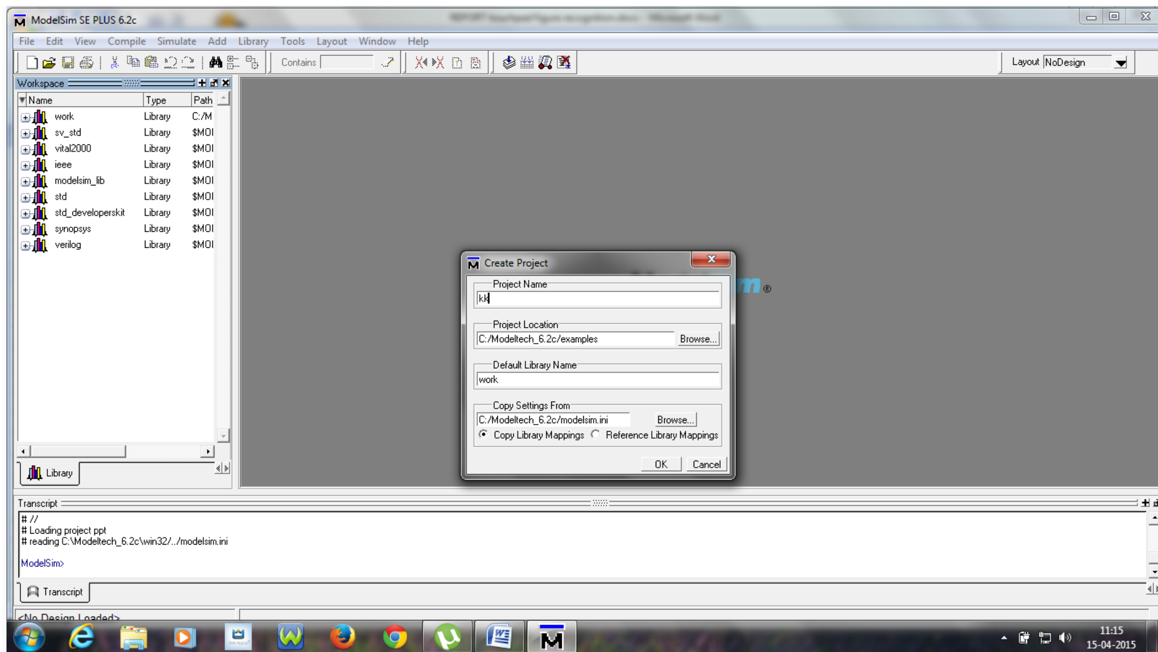
Race conditions, delta, and event activity can be analyzed in the list and wave windows. User-defined enumeration values can be easily defined for quicker understanding of simulation results. For improved debug productivity, ModelSim also has graphical and textual dataflow capabilities.

How To Use Modelsim

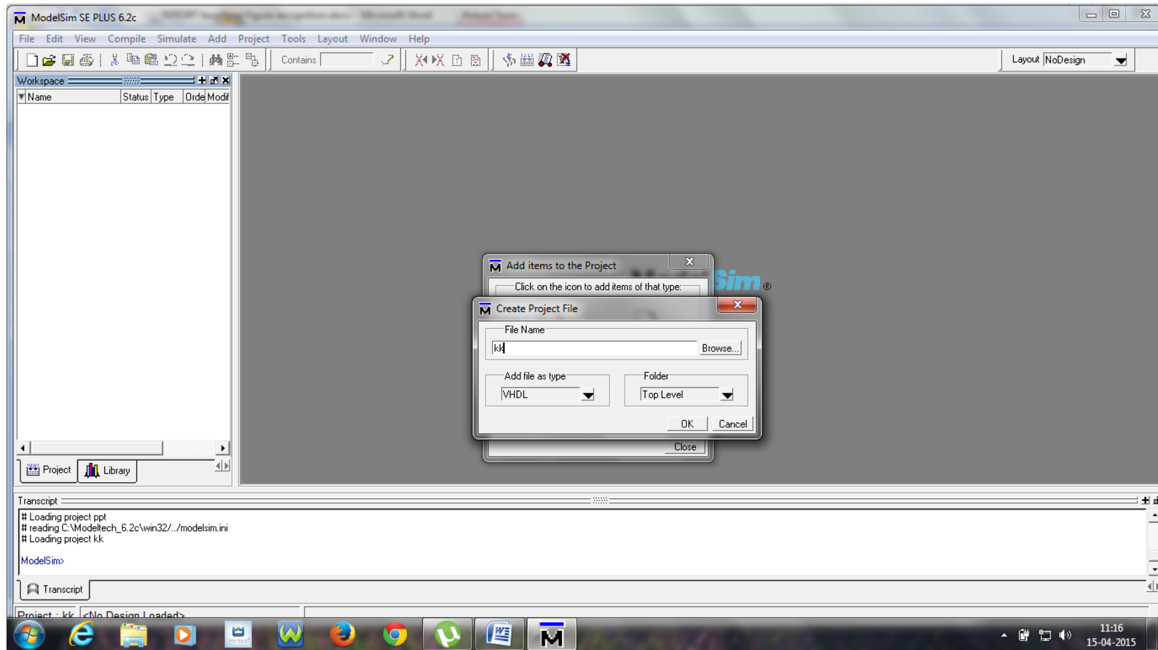
1.Open modelsim software



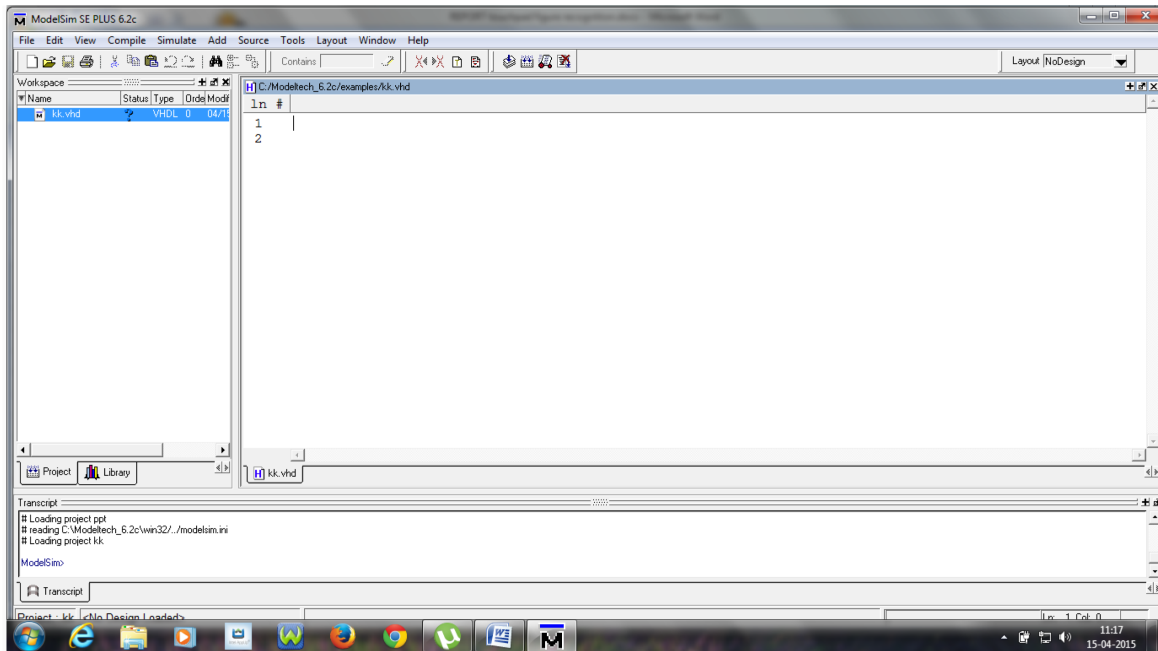
2.Creat New project



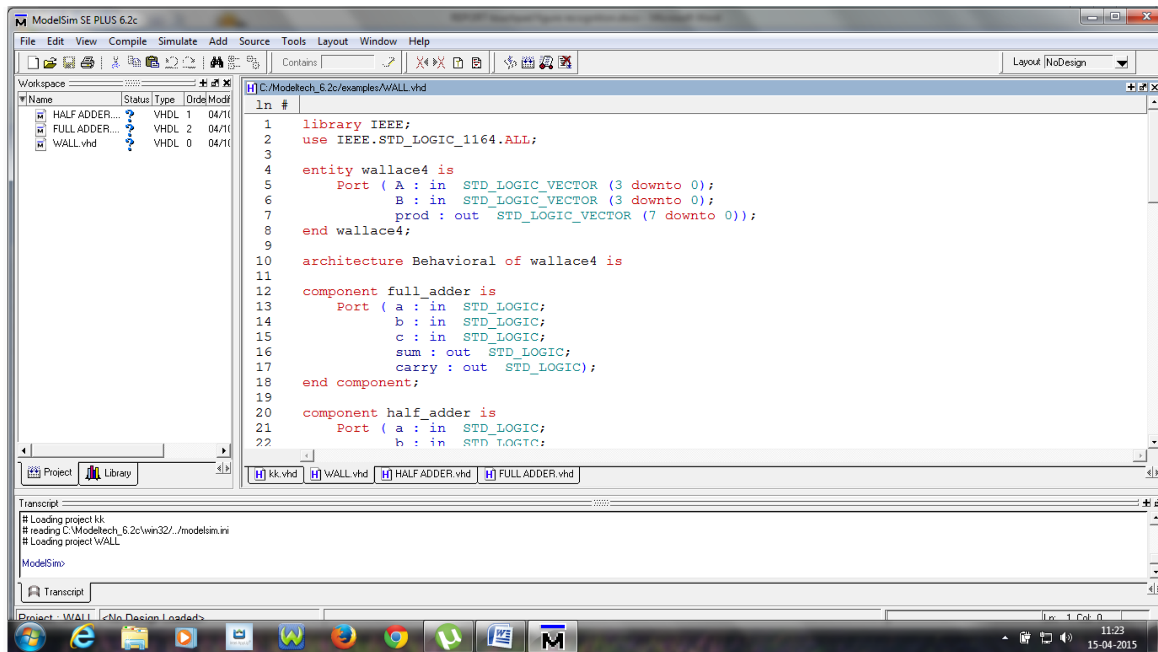
3.Creat project file



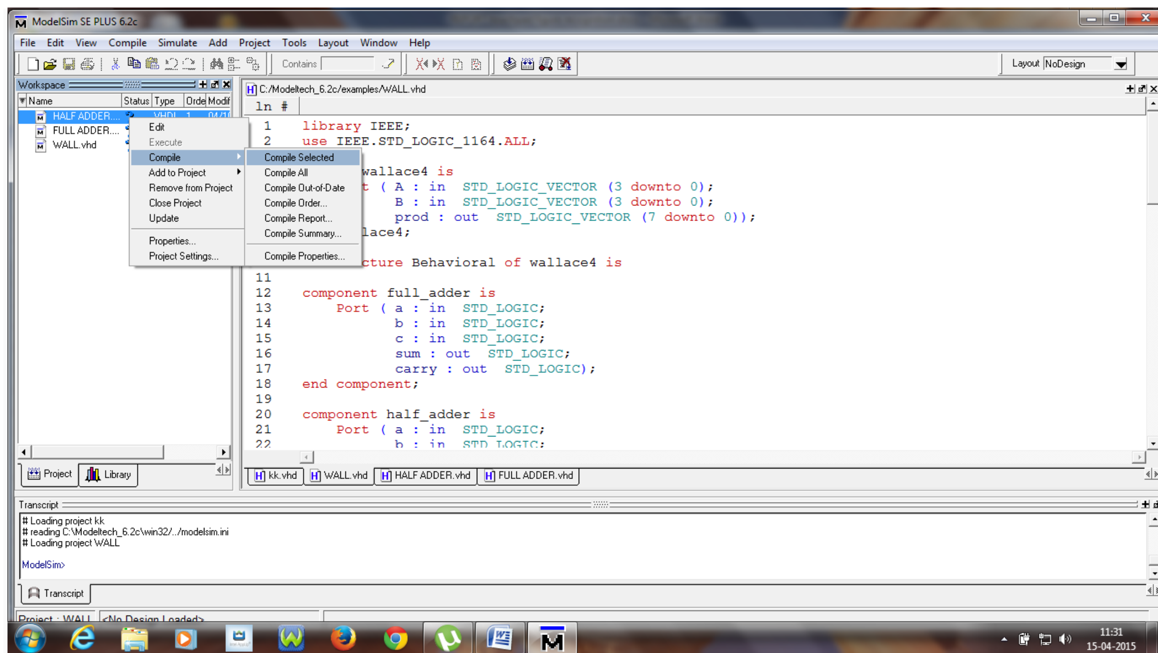
4.Double click on file name,it will open work space



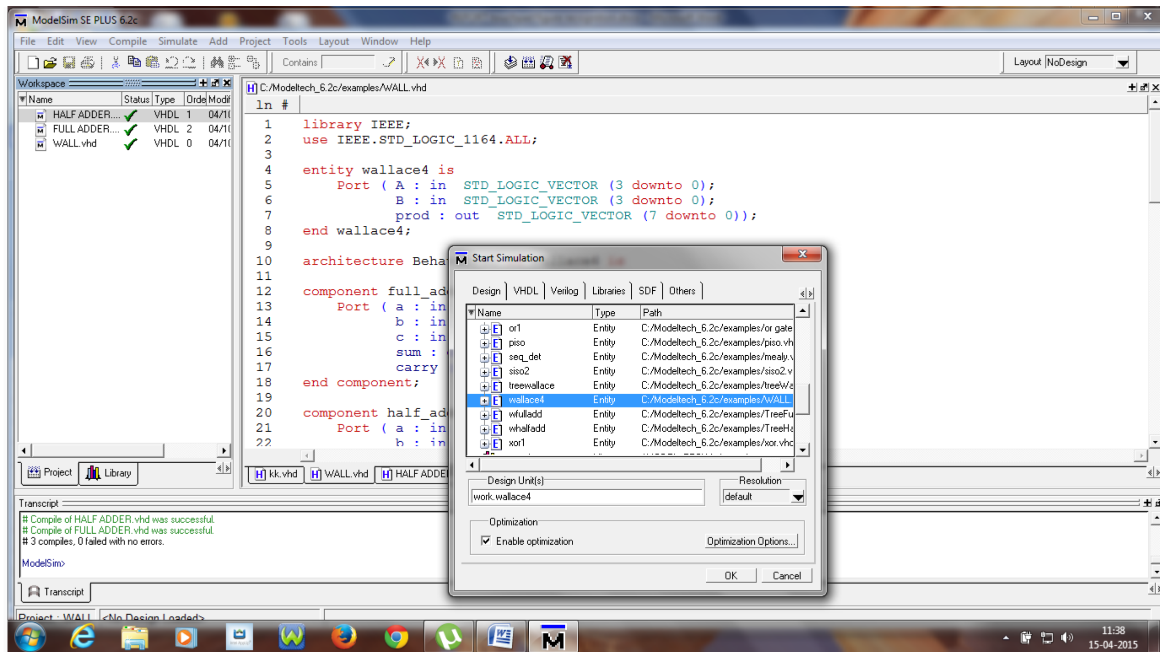
5.Write the codes



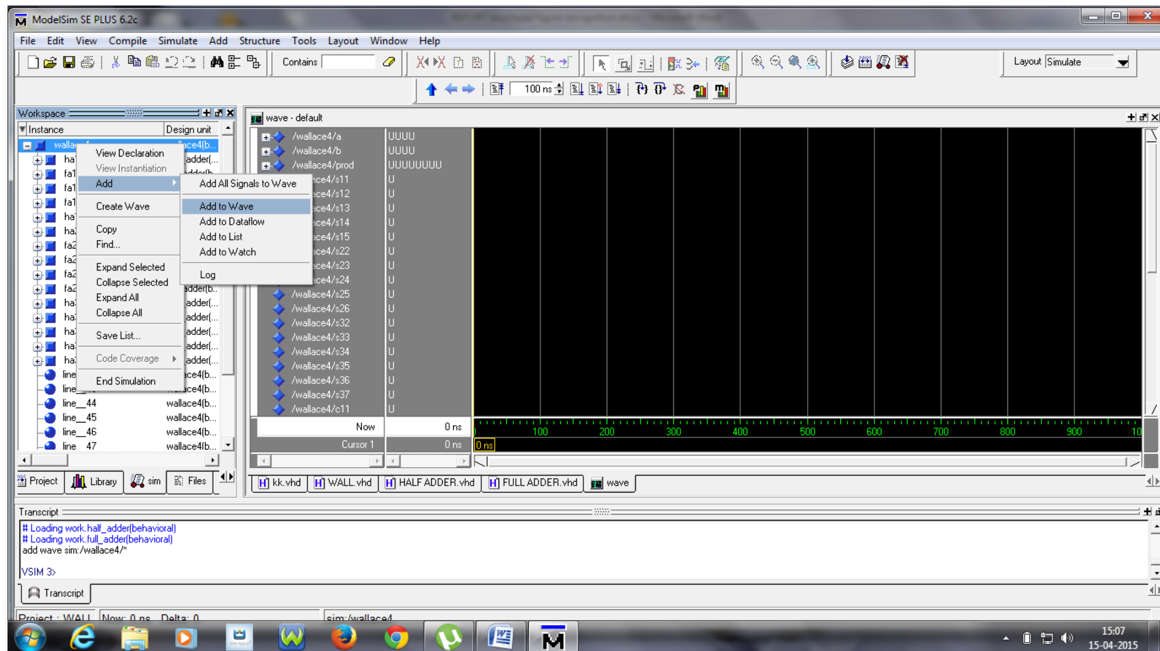
6.Compile the file: Select the file :Right click on it: Go to compile and select compile



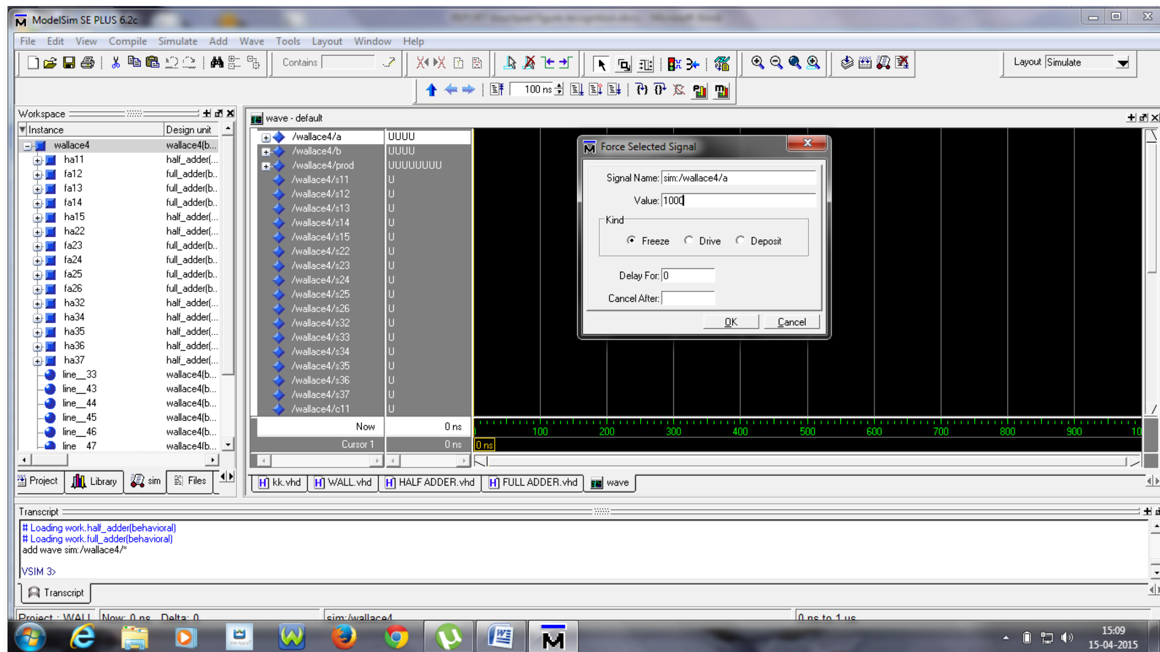
7 Go to Simulate. And click on Start simulation



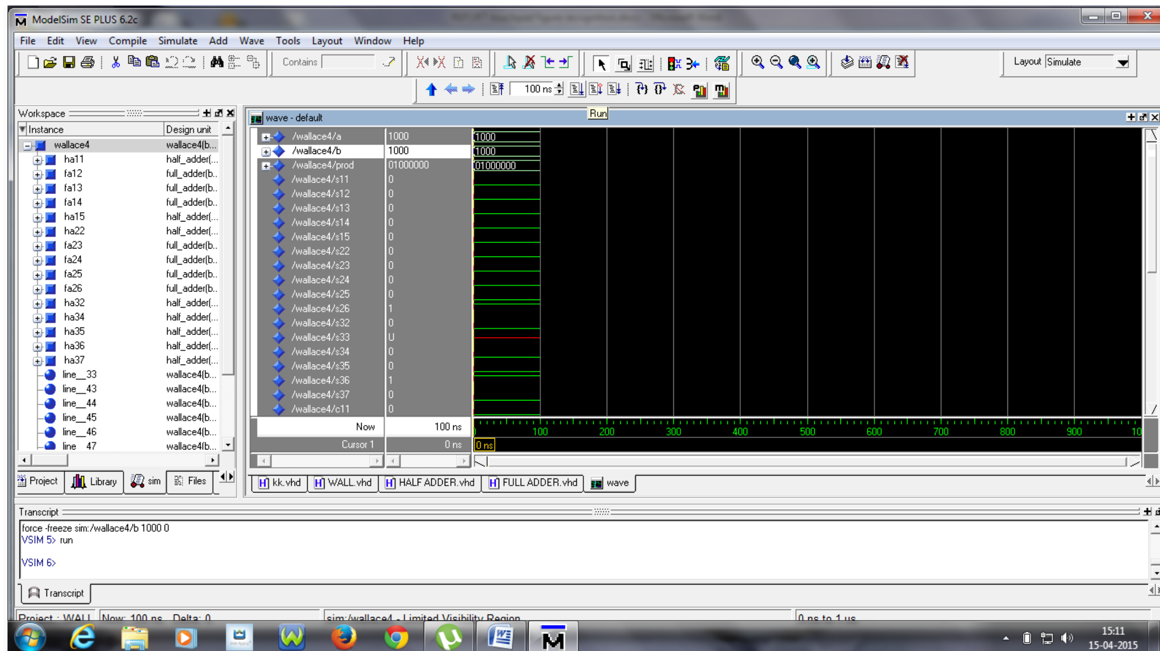
8.Adding wave to signal



9.Now Force the input signal value.



10.Click on Run and get the output



XILINX ISE

Xilinx ISE (Integrated Synthesis Environment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.

The Xilinx ISE is a design environment for FPGA products from Xilinx, and is tightly-coupled to the architecture of such chips, and cannot be used with FPGA products from other vendors.The Xilinx ISE is primarily used for circuit synthesis and design, while the ModelSim logic simulator is used for system-level testing. Other components shipped with the Xilinx ISE include the Embedded Development Kit (EDK), a Software Development Kit (SDK) and ChipScope Pro.

User Interface

The primary user interface of the ISE is the Project Navigator, which includes the design hierarchy (Sources), a source code editor (Workplace), an output console (Transcript), and a processes tree (Processes).

The Design hierarchy consists of design files (modules), whose dependencies are interpreted by the ISE and displayed as a tree structure.For single-chip designs there may be one main module, with other modules included by the main module, similar to the main() subroutine in C++ programs. Design constraints are specified in modules, which include pin configuration and mapping.

The Processes hierarchy describes the operations that the ISE will perform on the currently active module. The hierarchy includes compilation functions, their dependency functions, and other utilities. The window also denotes issues or errors that arise with each function.

The Transcript window provides status of currently running operations, and informs engineers on design issues. Such issues may be filtered to show Warnings, Errors, or both.

Simulation

System-level testing may be performed with the ModelSim logic simulator, and such test programs must also be written in HDL languages. Test bench programs may include simulated input signal waveforms, or monitors which observe and verify the outputs of the device under test.

ModelSim may be used to perform the following types of simulations:

- Logical verification, to ensure the module produces expected results
- Behavioural verification, to verify logical and timing issues
- Post-place & route simulation, to verify behaviour after placement of the module within the reconfigurable logic of the FPGA

Synthesis

Xilinx's patented algorithm for synthesis allow designs to run upto 30% faster than competing programs, and allows greater logic density which reduces project costs.

Also, due to the increasing complexity of FPGA fabric, including memory blocks and I/O blocks, more complex synthesis algorithms were developed that separate unrelated modules into *slices*, reducing post-placement errors.

IP Cores are offered by Xilinx and other third-party vendors, to implement system-level functions such as digital signal processing (DSP), bus interfaces, networking protocols, image processing, embedded processors, and peripherals.Xilinx has been instrumental in shifting designs from ASIC-based implementation to FPGA-based implementation.

CHAPTER 4

ALGORITHMS IMPLEMENTED

Wallace Tree Multiplier

Introduction

A Wallace tree is an efficient hardware implementation of a digital circuit that multiplies two integers, devised by Australian Computer Scientist Chris Wallace in 1964.

The Wallace tree has three steps:

1. Multiply (that is - AND) each bit of one of the arguments, by each bit of the other, yielding n^2 results. Depending on position of the multiplied bits, the wires carry different weights, for example wire of bit carrying result of $a_2 b_3$ is 32 (see explanation of weights below).

2. Reduce the number of partial products to two by layers of full and half adders.

3. Group the wires in two numbers, and add them with a conventional adder.

The second phase works as follows. As long as there are three or more wires with the same weight add a following layer:

- Take any three wires with the same weights and input them into a full adder. The result will be an output wire of the same weight and an output wire with a higher weight for each three input wires.

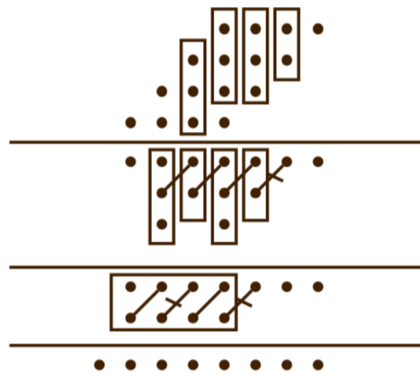
- If there are two wires of the same weight left, input them into a half adder.

- If there is just one wire left, connect it to the next layer.

The benefit of the Wallace tree is that there are only $O(\log n)$ reduction layers, and each layer has $O(1)$ propagation delay. As making the partial products is $O(1)$ and the final addition is $O(\log n)$, the multiplication is only $O(\log n)$, not much slower than addition (however, much more expensive in the gate count). Naively adding partial products with regular adders would require $O(\log^2 n)$ time. From a complexity theoretic perspective, the Wallace tree algorithm puts multiplication in the class NC.

These computations only consider gate delays and don't deal with wire delays, which can also be very substantial.

The Wallace tree can be also represented by a tree of 3/2 or 4/2 adders. It is sometimes combined with Booth encoding.



Wallace Tree: 2 carry-save levels, 5 FA, 3 HA, 4-bit CPA

Generic Example

For $n = 4$,

multiplying $a_3a_2a_1a_0$ by $b_3b_2b_1b_0$:

1. First we multiply every bit by every bit:

    - weight 1 - a_0b_0
    - weight 2 - a_0b_1 , a_1b_0
    - weight 4 - a_0b_2, a_1b_1, a_2b_0
    - weight 8 - a_0b_3 , a_1b_2, a_2b_1, a_3b_0
    - weight 16 - a_1b_3, a_2b_2, a_3b_1
    - weight 32 - a_2b_3, a_3b_2
    - weight 64 - a_3b_3

2. Reduction layer 1:

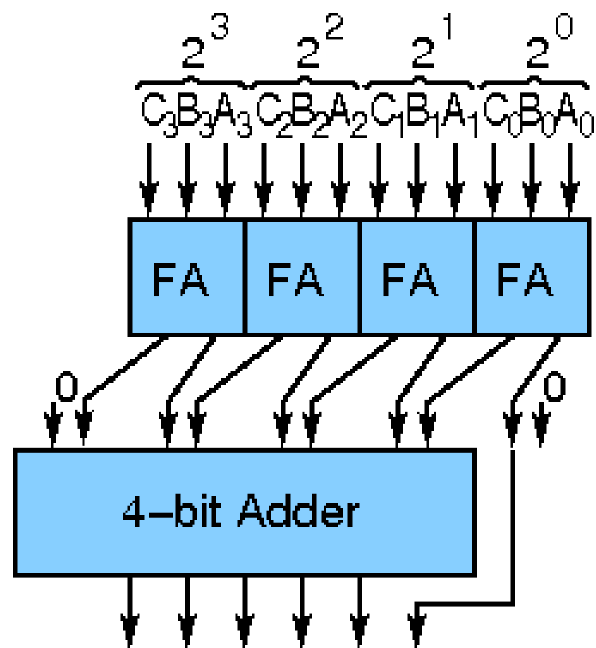
    - Pass the only weight-1 wire through, output: 1 weight-1 wire
    - Add a half adder for weight 2, outputs: 1 weight-2 wire, 1 weight-4 wire
    - Add a full adder for weight 4, outputs: 1 weight-4 wire, 1 weight-8 wire
    - Add a full adder for weight 8, and pass the remaining wire through, outputs: 2 weight-8 wires, 1 weight-16 wire
    - Add a full adder for weight 16, outputs: 1 weight-16 wire, 1 weight-32 wire

- Add a half adder for weight 32, outputs: 1 weight-32 wire, 1 weight-64 wire
- Pass the only weight-64 wire through, output: 1 weight-64 wire

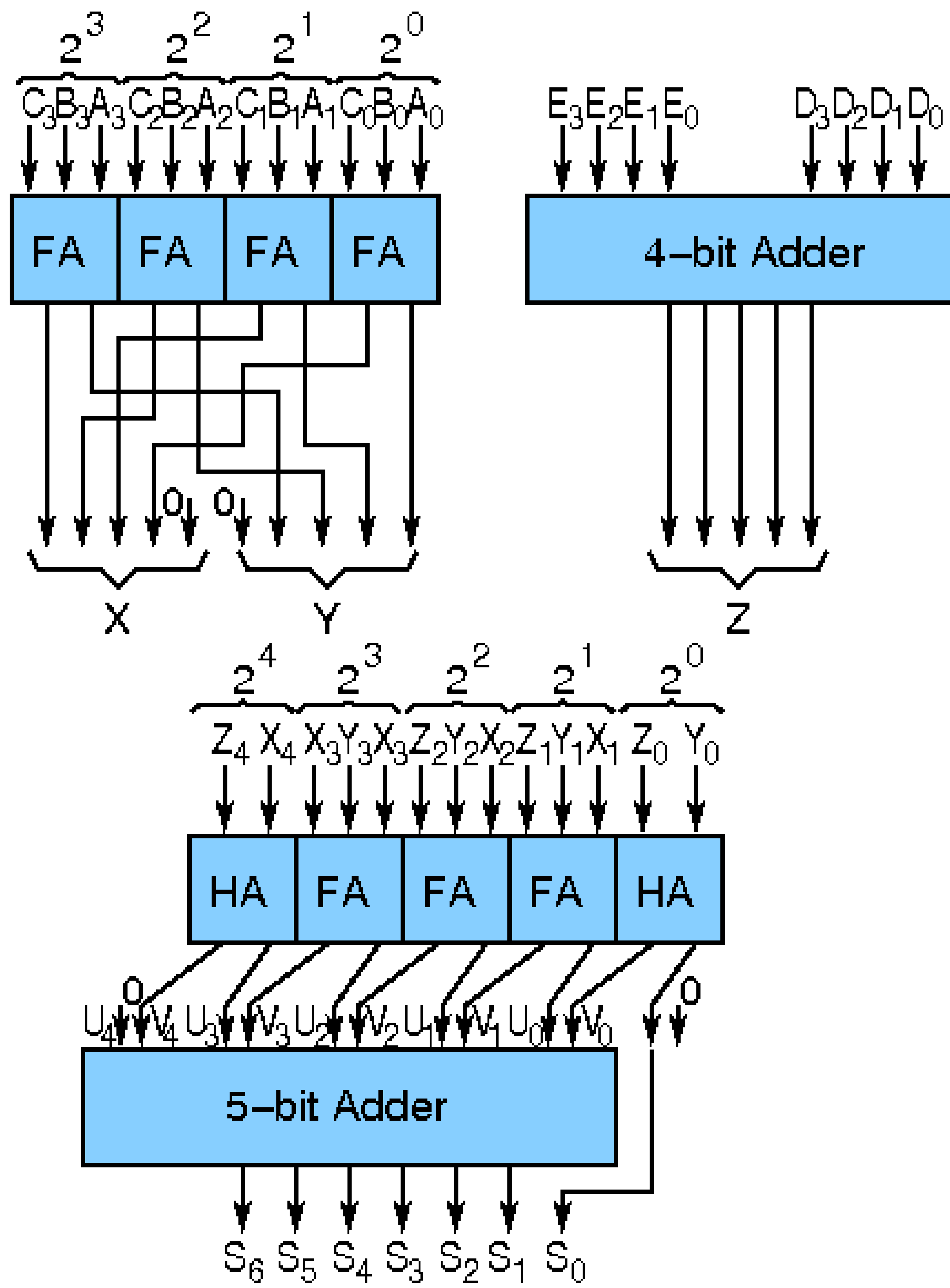
3. Wires at the output of reduction layer 1:
   - weight 1 - 1
   - weight 2 - 1
   - weight 4 - 2
   - weight 8 - 3
   - weight 16 - 2
   - weight 32 - 2
   - weight 64 - 2
4. Reduction layer 2:
   - Add a full adder for weight 8, and half adders for weights 4, 16, 32, 64
5. Outputs:
   - weight 1 - 1
   - weight 2 - 1
   - weight 4 - 1
   - weight 8 - 2
   - weight 16 - 2

- weight 32 - 2
- weight 64 - 2
- weight 128 - 1

6. Group the wires into a pair integers and an adder to add them



Wallace Multiplier Intermediate Stage



Final Stage of Wallace Tree Multiplier

Karatsuba Multiplier

The Karatsuba algorithm is a fast multiplication algorithm. It was discovered by Anatolii Alexeevitch Karatsuba in 1960 and published in 1962. It reduces the multiplication of two *n*-digit numbers to at most $n^{\log_2 3} \approx n^{1.585}$ single-digit multiplications in general (and exactly $n^{\log_2 3}$ when *n* is a power of 2). It is therefore faster than the classical multiplication algorithm, which requires n^2 single-digit products.

For example, the Karatsuba algorithm requires 3^{10} = 59,049 single-digit multiplications to multiply two 1024-digit numbers ($n = 1024 = 2^{10}$), whereas the classical algorithm requires $(2^{10})^2$ = 1,048,576.

The Karatsuba algorithm was the first multiplication algorithm asymptotically faster than the quadratic "grade school" algorithm. The Toom–Cook algorithm is a faster generalization of Karatsuba's method, and the Schönhage–Strassen algorithm is even faster, for sufficiently large *n*.

Basic steps of algorithm

The basic step of Karatsuba's algorithm is a formula that allows us to compute the product of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and digit shifts.

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

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

where

$$z_2 = x_1 y_1$$
$$z_1 = x_1 y_0 + x_0 y_1$$
$$z_0 = x_0 y_0$$

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 z_0 and z_2 as before we can calculate

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

which holds since

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

A more efficient implementation of Karatsuba multiplication can be set as

$$xy = (b^2 + b)x_1 y_1 - b(x_1 - x_0)(y_1 - y_0) + (b + 1)x_0 y_0,$$

where b is the power where the split occurs of x_1.

Example

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

12345 = **12** · *1000* + **345**

6789 = **6** · *1000* + **789**

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

z_2 = **12** × **6** = 72

z_0 = **345** × **789** = 272205

z_1 = (**12** + **345**) × (**6** + **789**) − z_2 − z_0 = 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 = $z_2 \cdot B^{2m} + z_1 \cdot B^m + z_0$, i.e.

result = 72 · 1000^2 + 11538 · *1000* + 272205 = **83810205**.

Note that the intermediate third multiplication operates on an input domain which is less than twice larger than for the two first multiplications, its output domain is less than four times larger, and base-*1000* carries computed from the first two multiplications must be taken into account when computing these two subtractions; but note also that this partial result z_1 cannot be negative: to compute these subtractions, equivalent

additions using complements to 1000^2 can also be used, keeping only the two least significant base-*1000*digits for each number:

$z_1 = 283815 - 72 - 272205 = (283815 + 999928 + 727795)$ **mod** $1000^2 =$ 2011538 **mod** $1000^2 = 11538$.

CHAPTER 5

VHDL CODES FOR MULTIPLIER

WALLACE TREE MULTIPLIER

MAIN MULTIPLIER

```
library ieee;
use ieee.std_logic_1164.all;

entity treeWallace is
   Port ( A, B : in  STD_LOGIC_VECTOR (3 downto 0);
          PROD : out  STD_LOGIC_VECTOR (7 downto 0));
end treeWallace;


architecture multiplier of treeWallace is
--component Half Adder for instances where two partial products are to be
added
component WFULLADD is
Port( a, b, cin : in STD_LOGIC;
           sum, carry : out STD_LOGIC);
end component;


--component Full Adder for instances where more than two partial products
are to be added
component WHALFADD is
```

```
Port(a, b : in STD_LOGIC;
     sum, carry : out STD_LOGIC);
end component;



signal     s11,s12,s13,s14,s15,s22,s23,s24,s25,s26,s32,s34,s35,s36,s37     :
STD_LOGIC;

signal    c11,c12,c13,c14,c15,c22,c23,c24,c25,c26,c32,c34,c35,c36,c37    :
STD_LOGIC;

signal p0,p1,p2,p3 : STD_LOGIC_VECTOR(3 downto 0);



begin

process(A,B)
begin
--partial products generation stage
--here each bit of each binary multiplicand is multiplied by the other
--thus we have n^2 bits
for i in 0 to 3 loop
p0(i)<=A(i) and B(0);
p1(i)<=A(i) and B(1);
p2(i)<=A(i) and B(2);
p3(i)<=A(i) and B(3);
end loop;
end process;
```

```
--first partial products reduction stage
ha11 : WHALFADD port map(p0(1),p1(0),s11,c11);
fa12 : WFULLADD port map(p0(2),p1(1),p2(0),s12,c12);
fa13 : WFULLADD port map(p0(3),p1(2),p2(1),s13,c13);
fa14 : WFULLADD port map(p1(3),p2(2),p3(1),s14,c14);
ha15 : WHALFADD port map(p2(3),p3(2),s15,c15);

--second partial products reduction stage
ha22 : WHALFADD port map(c11,s12,s22,c22);
fa23 : WFULLADD port map(p3(0),c12,s13,s23,c23);
fa24 : WFULLADD port map(c13,c32,s14,s24,c24);
fa25 : WFULLADD port map(c14,c24,s15,s25,c25);
fa26 : WFULLADD port map(c15,c25,p3(3),s26,c26);

--third partial products reduction stage
ha32 : WHALFADD port map(c22,s23,s32,c32);
ha34 : WHALFADD port map(c23,s24,s34,c34);
ha35 : WHALFADD port map(c34,s25,s35,c35);
ha36 : WHALFADD port map(c35,s26,s36,c36);
ha37 : WHALFADD port map(c36,c26,s37,c37);

---final mapping
PROD(0)<=p0(0);
PROD(1)<=s11;
PROD(2)<=s22;
PROD(3)<=s32;
PROD(4)<=s34;
PROD(5)<=s35;
```

```
PROD(6)<=s36;
PROD(7)<=s37;

end multiplier;
```

FULL ADDER

```
library ieee;
use ieee.std_logic_1164.all;

entity WFULLADD is
port(a,b,cin: in STD_LOGIC;
  sum,carry: out STD_LOGIC);
end WFULLADD;

architecture fulladd of WFULLADD is

  begin

  sum <= (a AND b) OR (b AND cin) OR (a AND cin);
  carry <= a XOR b XOR cin;

end fulladd;
```

HALF ADDER

```
library ieee;
use ieee.std_logic_1164.all;

entity WHALFADD is
port(a,b: in STD_LOGIC;
  sum,carry: out STD_LOGIC);
end WHALFADD;

architecture halfadd of WHALFADD is

begin

  sum <= a XOR b;
  carry <= a AND b;

end halfadd;
```

KARATSUBA TREE MULTIPLIER

MAIN MULTIPLIER

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity karatsuba_multiplier_even is

port (
  a, b: in std_logic_vector(7 downto 0);
  d: out std_logic_vector(15 downto 0)
);
end karatsuba_multiplier_even;


architecture simple of karatsuba_multiplier_even is

  component Multiplier_VHDL is
  port
   (
     Nibble1, Nibble2: in std_logic_vector(3 downto 0);

     Result: out std_logic_vector(7 downto 0)
   );
  end component;
```

```
  component Multiplier_VHDL1 is
    port
   (
     Nibble1, Nibble2: in std_logic_vector(4 downto 0);

     Result: out std_logic_vector(9 downto 0)
   );
end component;

  component test1 is
       port(a,b:in std_logic_vector(3 downto 0);
           s:out std_logic_vector(4 downto 0)
           );
  end component;

 component subtr is
  port(A,B : in std_logic_vector(9 downto 0);
     RES : out std_logic_vector(9 downto 0));
  end component;

  component adr16 is
  port(A,B : in std_logic_vector(15 downto 0);
     RES : out std_logic_vector(15 downto 0));
end component;

  signal x0y0,x1y1: std_logic_vector(7 downto 0);
  signal x01y01: std_logic_vector(9 downto 0);
```

```
  signal x0_p_X1, y0_p_y1: std_logic_vector(4 downto 0);
  signal x0y0a, x1y1a, res1, res2: std_logic_vector(9 downto 0);
  signal h, i, l, res3, res4: std_logic_vector(15 downto 0):=(others=>'0');

begin

  mult1: Multiplier_VHDL
          port map(a(3 downto 0), b(3 downto 0), x0y0);

  mult2: Multiplier_VHDL
          port map(a(7 downto 4), b(7 downto 4), x1y1);

  mult3: Multiplier_VHDL1
          port map(x0_p_X1, y0_p_y1, x01y01);

  add1: test1
          port map(a(7 downto 4),a(3 downto 0), x0_p_X1);
  add2: test1
          port map(b(7 downto 4),b(3 downto 0), y0_p_y1);

   x1y1a <= "00"&x1y1;
   x0y0a <= "00"&x0y0;

  sub1: subtr
          port map(x01y01,x1y1a,res1);
  sub2: subtr
          port map(res1,x0y0a,res2);
```

```
    h(15 downto 8) <= x1y1;
    i(13 downto 4) <= res2;
    l(7 downto 0) <= x0y0;

  adr16a: adr16
        port map(h, i, res3);
  adr16b: adr16
        port map(res3, l, res4);

  d <= res4;

  end simple
```

4-BIT RIPPLE CARRY ADDER

```
library ieee;
    use ieee.std_logic_1164.all;
    entity rca4 is
        port(a,b:in std_logic_vector(3 downto 0);
            cin:in std_logic;
            s:out std_logic_vector(3 downto 0);
            cout:out std_logic);
        end rca4;

architecture struct of rca4 is
    signal c1,c2,c3:std_logic;
    component full is
```

```
port(a,b,cin:in std_logic;
    s,cout:out std_logic);
end component;
begin
    fa1:full port map(a(0),b(0),cin,s(0),c1);
    fa2:full port map(a(1),b(1),c1,s(1),c2);
    fa3:full port map(a(2),b(2),c2,s(2),c3);
    fa4:full port map(a(3),b(3),c3,s(3),cout);
end struct;
```

COMPONENT FOR CONCATENATING CARRIES

```
library ieee;
   use ieee.std_logic_1164.all;
   entity test1 is
      port (a,b:in std_logic_vector(3 downto 0);
         s:out std_logic_vector(4 downto 0);
      end test1;

architecture struct of test1 is
 signal c1:std_logic_vector(3 downto 0);
 signal c2:std_logic;
 component rca4 is
      port(a,b: in std_logic_vector(3 downto 0);
         cin: in std_logic;
         s: out std_logic_vector(3 downto 0);
         cout: out std_logic);
 end component;
```

```
  begin
       rca1: rca4 port map(a,b,'0',c1,c2);
       s <= c2&c1;
    end struct;
```

4-BIT MULTIPIERS

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Multiplier_VHDL is
   port
   (
      Nibble1, Nibble2: in std_logic_vector(3 downto 0);

      Result: out std_logic_vector(7 downto 0)
   );
end entity Multiplier_VHDL;

architecture Behavioral of Multiplier_VHDL is
begin

   Result <= std_logic_vector(unsigned(Nibble1) * unsigned(Nibble2));

end architecture Behavioral;
```

5-BIT MULTIPLIER

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Multiplier_VHDL1 is
  port
  (
    Nibble1, Nibble2: in std_logic_vector(4 downto 0);

    Result: out std_logic_vector(9 downto 0)
  );
end entity Multiplier_VHDL1;

architecture Behavioral of Multiplier_VHDL1 is
begin

  Result <= std_logic_vector(unsigned(Nibble1) * unsigned(Nibble2));

end architecture Behavioral;
```

FULL ADDER

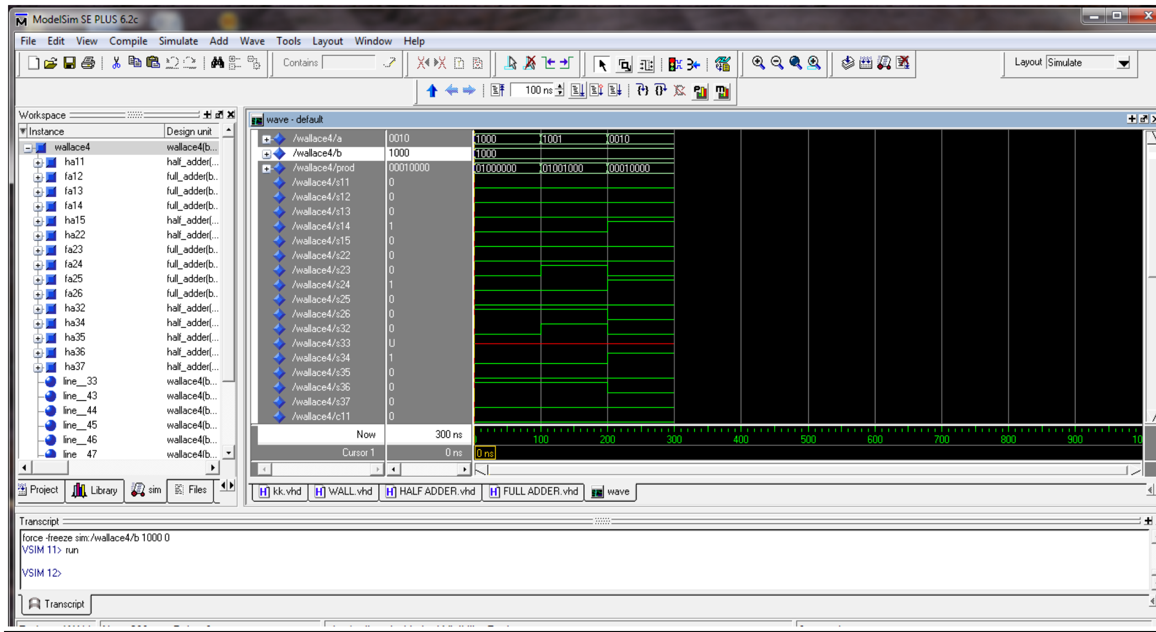
```
library ieee;
use ieee.std_logic_1164.all;
entity full is
    port(a,b,cin:in std_logic;
         s,cout:out std_logic);
     end full;

 architecture FA of full is
   begin
       s<= a xor b xor cin;
       cout<=(a and b)or(a and cin)or(b and cin) ;
   end FA;
```

16-BIT ADDER

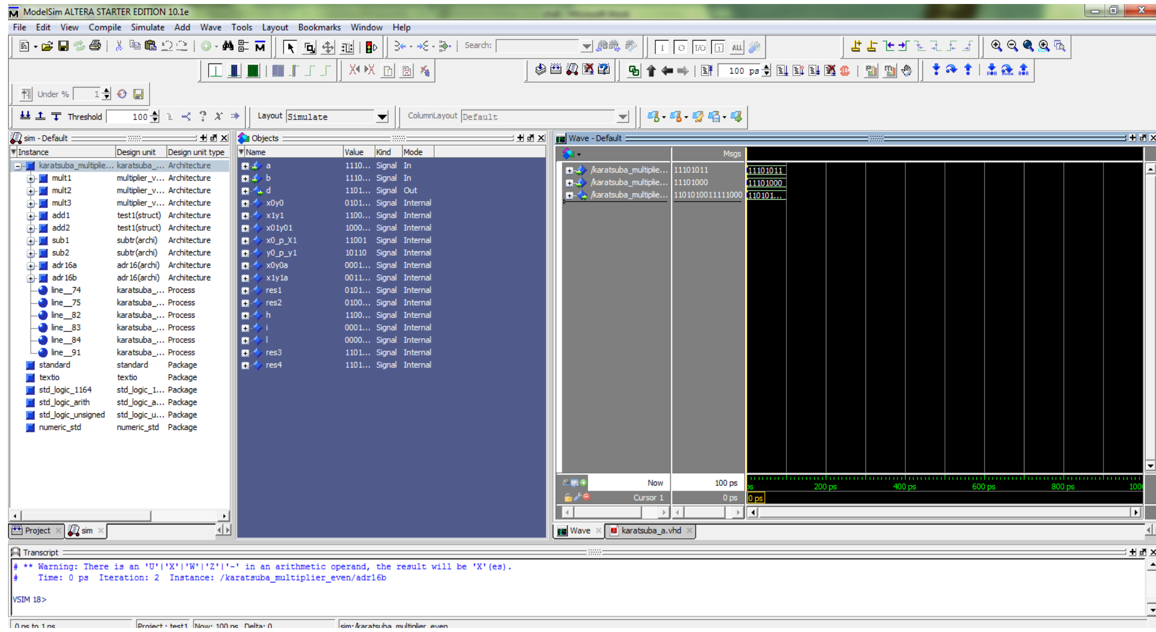
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity adr16 is
  port(A,B : in std_logic_vector(15 downto 0);
      RES : out std_logic_vector(15 downto 0));
end adr16;
architecture archi of adr16 is
  begin
    RES <= A + B;
end archi;
```

10-BIT SUBTRACTOR

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity subtr is
  port(A,B : in std_logic_vector(9 downto 0);
      RES : out std_logic_vector(9 downto 0));
end subtr;
architecture archi of subtr is
  begin
    RES <= A - B;
end archi;
```



Simulation output: Wallace Tree Multiplier



Simulation output: Karatsuba Multiplier

CHAPTER 6

APPLICATIONS

The potential usages of proposed design are -

• High Speed Signal Processing that includes DSP based applications.

• DWT and DCT transforms used for image and wide signal processing.

• FIR and IIR Filters for high speed, low power filtering applications.

• Multirate signal processing applications such as digital down converters and up converters.

Computers are extremely capable in two broad areas: (1) data manipulation, such as word processing and database management, and (2) mathematical calculation, used in science, engineering, and Digital Signal Processing. However, most computers are not optimized to perform both functions. In computing applications such as word processing, data must be stored, sorted, compared, moved, etc., and the time to execute a particular instruction is not critical, as long as the program's overall response time to various commands and operations is adequate enough to satisfy the end user. Occasionally, mathematical operations may also be performed, as in a spreadsheet or database program, but speed of execution is generally not the governing factor. In most general purpose computing applications there is no concentrated attempt by software companies to make the code efficient. Application programs are loaded with "features" which require more memory and faster processors with every new release or upgrade.

On the other hand, digital signal processing applications require that mathematical operations be performed quickly, and the time to execute a given instruction must be known precisely, and it must be predictable. Both code and hardware must be extremely efficient to accomplish this. As has been shown in the last two sections of this book, the most fundamental mathematical operation or kernel in all of DSP is the sum-of-products (or dot-product). Fast execution of the dot product is critical to fast Fourier transforms (FFTs), real time digital filters, matrix multiplications, graphics pixel manipulation, etc.

Multiplication is an important fundamental function in arithmetic operations. Multiplication-based operations such as Multiply and Accumulate(MAC) and inner product are among some of the frequently used computation Intensive Arithmetic Functions(CIAF) currently implemented in many Digital Signal Processing (DSP) applications such as convolution, Fast Fourier Transform(FFT), filtering and in microprocessors m its arithmetic and logic unit. Since multiplication dominates the execution tune of most DSP algorithms, so there is a need of high speed multiplier. Currently, multiplication time is still the dominant factor in determining the instruction cycle time of a DSP chip. The demand for high speed processing has been increasing as a result of expanding computer and signal processing applications. Higher throughput arithmetic operations are important to achieve the desired performance in many real-time signal and image processing applications.

References

Text References

- C. S.Wallace, “A suggestion for fast multipliers,” IEEE Trans. Electron. Comput., no. EC-13, pp. 14–17, Feb. 1964.
- J. Bhasker, A VHDL Primer, Third Edition, Pearson, 1999.
- John. P. Hayes, “Computer Architecture and Organization”, McGraw Hill, 1998.

Internet Sources

- http://www.researchgate.net/
- http://www.ece.rochester.edu/~parihar/pres/Report_FP-Multipliers.
- http://www.edaboard.com/
- http://stackoverflow.com/
- https://en.wikipedia.org/