Design of a C- Testable Carry Save Array Multiplier using VHDL.

by

Mohammad Masud Uddin Bhuiyan

A project submitted to the Department of Electrical and Electronic Engineering, BUET in partial fulfillment of the requirements for the degree of Master of Engineering in ELECTRICAL AND ELECTRONIC ENGINEERING

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DHAKA-1000, BANGLADESH

February15, 2005

The project titled "Design of a C-Testable Carry Save Array Multiplier using VHDL" submitted by Mohammad Masud Uddin Bhuiyan, Roll no:100106220, Session: Oct,2001 has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical and Electronic Engineering on February 15, 2005.

Board of Examiners

1. _____

Dr. A.B.M Harun-ur-Rashid — **Chairman**
Associate Professor — (Supervisor)
Department of EEE
BUET, Dhaka-1000

2. _____

Dr.M.M Shahidul Hassan — **Member**
Professor
Department of EEE
BUET, Dhaka-1000

3. _____

Dr. Md. Quamrul Huda — **Member**
Associate Professor
Department of EEE
BUET, Dhaka-1000

Declaration

It is hereby declared that this project or any part of it has not been submitted elsewhere for the award of any degree or diploma.

15.2.05

Mohammad Masud Uddin Bhuiyan

DEDICATION

To my parents

Acknowledgement

It is a matter of great pleasure on the part of the author to acknowledge his profound gratitude to his supervisor Dr. A.B.M Harun-ur-Rashid for his support, continuous guiding, suggestions, advice, assistance and wholehearted supervision throughout the progress of this project work.

The author wishes to acknowledge thankfully the discussions made with Masud Hyder, Podder and others who have been directly or indirectly related to this work for their support and encouragement.

Finally the author is grateful to Almighty Allah for giving him the strength and courage to complete this project.

Contents

List of figures

List of Tables

List of abbreviations

CPLD	Complex Programmable Logic device
FA	Full Adder
FPGA	Field Programmable Gate Array
CSA	Carry save array
MCSA	Modified Carry save array
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale integration
PLD	Programmable Logic device
ASIC	Application specific integrated circuit
HDL	Hardware Description Language
AHDL	Altera Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineer SOC
SOC	System on a chip
DUT	Device under Test
ILA	Iterative Logic Array
ANSI	American National Standardization Institute

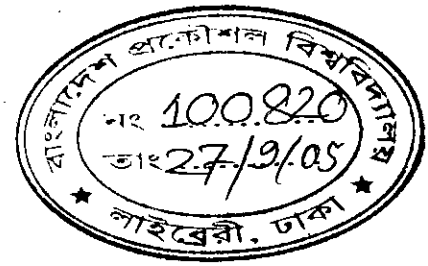
Abstract

A C-testable and easily synthesizable MXN bit carry save array multiplier architecture is developed using Very High Speed Integrated Circuit Hardware Description Language(VHDL). The architecture is translated in efficient VHDL code and the designed hardware code can be used as an Intellectual Property (IP) core in designing VLSI system on a chip.

The basic cell of the multiplier has four inputs. So we can test the cell with sixteen input combinations. In order to minimize the test pattern (input combinations) special care is taken during multiplier testing. We can divide the testing scheme into two parts. Basic cell consists of a full adder with an *AND* gate. At first *AND* gate is tested which needed four test patterns. Then keeping *AND* gate output '0' applying various combinations of *c,d* and keeping *AND* gate out '1' applying various combinations of *c,d* we need another four test patterns. Out of these twelve test patterns two test patterns are common. With the remaining ten test patterns multiplier is C- testable.

Moreover the test pins Dodd, Deven, Codd, Ceven, Dia_odd, Dia_even, FA_x, FA_Cin, Test are fixed, don't vary with the operand sizes. '0' is given to these input pins during multiplications. Care is taken to minimize the total numbers of test pins. For multiplications of signed numbers the result will be in twos complement form.

Introduction



1.1 Motivation

The advancement of Very Large Scale Integration (VLSI) technology has made it possible to build System on a Chip (SOC). Multipliers are one of the key elements in single chip Digital Signal Processor, microcontroller, microprocessor etc [1]-[4]. Depending upon application, these SOC devices need multiplier with various operand sizes. It is desirable that the design of the various components of SOC devices such as the multiplier should be reusable in order to reduce the design cycle of the chip. Array multiplier due to their high regularity, are efficiently designed as parts of complex VLSI devices [5]. However due to the increasing complexity of VLSI circuits, it is becoming more and more difficult and costly to test them [6]-[7].Specially the embedded multipliers in SOC devices have low controllability and observability, making the use of appropriate testing scheme a necessity. Some work on C-testable array multiplier has been done in the past [8]. However to make the design reusable it is necessary to use process independent design approach. It is also desirable that the design should be parameterizable such that multipliers of arbitrary operand size could be designed without any external effort. The design core should also be generalized such that both signed and unsigned multiplication can be accomplished.

1.2 Objective

The objective of this work is to design an easily testable and parameterizable Carry save array multiplier using Very High Speed Integrated Circuit Hardware Description Language(VHDL). Because of parameterizable approach the design code could be executed for any number of arrays (operands) and this property will make the multiplier code reusable. Also process independent design approach will be ensured by VHDL based design approach. The designed multiplier would be exhaustively testable for single stuck-at fault with fixed number of test patterns irrespective of its size with minimum number pf additional pin added. It will also be able to perform multiplication for both signed and unsigned multiplicand and multiplier.

Issues in VLSI Design

2.1 Test and testability

Once we have designed our logic, we must develop tests to allow manufacturing to separate faulty chips from good ones. A fault is a manifestation of a manufacturing defect; fault may be caused by small imperfections in starting material, processing steps, or in photo masking which may results in bridged connections or missing features. It is the aim of test procedure to determine the fault as early as possible. Testing a chip can occur:

- at the wafer level.
- at the packaged chip level.
- at the board level.
- at the system level.
- in the field.

By detecting a malfunctioning chip at an earlier level, the manufacturing cost may be kept low. A relationship, known as 'the rule of ten', tends to apply as far as test costs are concerned. This rule is concisely put as follows:

If the chip test cost = $x, then once that chip is soldered into a p.c. board with other components, test cost rises to $10x. Further, once that board is integrated into a system/equipment, then the cost escalates by a further factor of ten results in test cost of

;100x. Finally, a factor which is often overlooked is that test cost may escalate by a ;urther factor of ten when the equipment is in service in the field. It is thus essential to est at the chip level as comprehensively as possible. Design of testability (DFT) is an -essential part of good design. The requirements of testing must be considered at the outset and a satisfactory and sufficient measure of testability built into the architecture.

2.2 Benefits of testability

The time and money saved by designing for testability are the obvious major advantages - - the more efficiently and accurately we test the more profitable the product but there are many other advantages as well.

Designing for testability:

- Reduces the time required to pass the design to manufacturing
- Lowers the cost of manufacturing
- Minimizes the design engineer's involvement in production set up
- Improves cross-functional communication and cooperation between design, engineering, and manufacturing .
- Lowers both initial and life cycle costs
- Decreases test times and virtually eliminate harrowing production delays
- Guarantees more efficient diagnosis and repair in the field
- Provides more accurate diagnostics to the part and pin level

2.3 An overview of VLSI testing

The role of testing in the VLSI device realization process is illustrated in fig2.1. Test planning begins with specifications. Device specifications include functional (input-output behavior, frequency, timing, etc.), environmental (power, temperature, humidity, noise, etc.), and reliability (incoming quality, failure rate, etc.) specifications.

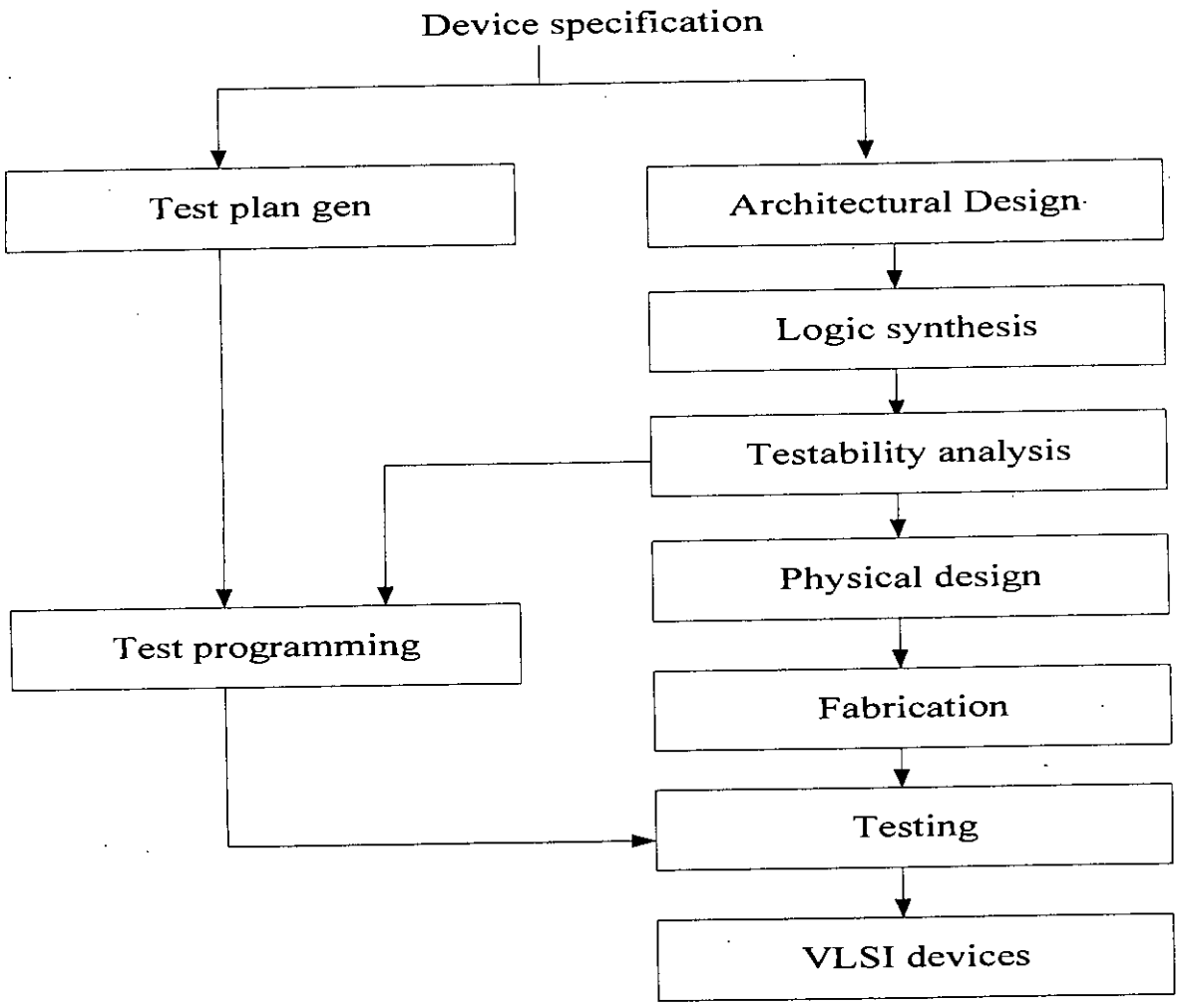


Fig.2.1 Test functions in VLSI device realization

Test activities are interwoven with design. Architectural design consists of portioning of the VLSI chip into realizable functional blocks. The next step, logic design, includes several test activities. Either the logic should be synthesized in a testable form, or the synthesized logic should be analyzed (and improved) for testability

After logic synthesis, test vectors are generated and evaluated for their effectiveness. The next test activity takes place after physical design (layout, timing verification, mask generation) and fabrication (wafer processing). Using test specifications and test vectors, we can develop a test program for the test equipment to be employed.

2.4 Method of testing

Testing in the digital systems is defined by the process by which a defect of a system can be exposed. The defect can occur at time of manufacture or when the system is in the field.

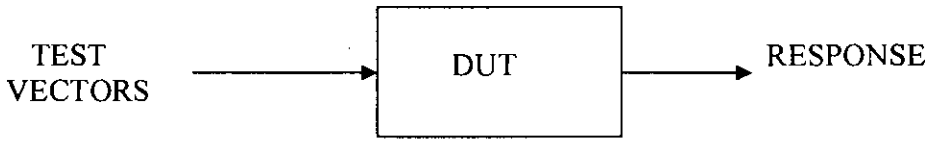


Fig: 2.2 Testing of a device

To find the actual fault in the device, a device under test (DUT) is shown above. Here test vectors are applied to the device. The resulting response is monitored and compared with the expected response. If we have the knowledge of correct response then we can know that the device under test has a defect or not.

2.5 Types of testing

Tests may fall into two main categories:

2.5.1 Functionality test

These tests assert that all the gates in the chip, acting together to achieve the desired function. These tests are usually used early in the designed cycle to verify the functionality of the circuit. Functionality tests are usually the first tests a designer might construct as part of the design process. Does this adder add? Does this counter count? Does this state-machine yield the right outputs at the right clock cycles?

For the most systems, functionality tests involve proving that the circuit is functionally equivalent to some specification. That specification might be a verbal description, a plain-language textual specification, a description in some high level computer language or in a hardware-description language such as VHDL, Verilog, or simply a table of inputs and required outputs. Functional equivalence involves running a simulator at some level on the two descriptions of the chip (say, one at the gate level and one at functional level) and ensuring for all inputs applied that outputs are equivalent at some convenient check-points in time. The most detailed check might be on a cycle-by-cycle basis.

Functional equivalence may be carried out at various level of design hierarchy. If the description is in a behavioral language, the behavior at a system level may be verifiable. For instance, in the case of microprocessor, the operating system might be booted and key program might be run for the behavioral description. However, this might be impractical (due to long simulation times) for a gate level model and extremely impractical for a transistor level model. The way out of this impasse is to use the hierarchy inherent within a system to verify chips and modules within chips. That,

combined with well-defined modular interfaces, goes a long way in increasing the likelihood that a system composed of many VLSI chips will be first time functional.

2.5.2 Manufacturing test

These tests are used after the chip is manufactured to verify the function of the chip as a whole. It verifies that every gate and register in the chip functions correctly. The need to do this arises from a number of manufacturing defects that might occur during chip fabrication or during accelerated life testing (where the chip is stressed by over-voltage and over-temperature operation). Typical defects include:

- layer-to-layer shorts(i.e. metal to metal).
- discontinuous wires(i.e. metal thins when crossing vertical topology jumps).
- thin-oxide shorts to substrate or well.

These in turn lead to particular circuit maladies, including:

- nodes shorted to power or ground.
- nodes shorted to each other.
- inputs floating/outputs disconnected.

Test is required to verify that each gate and register is operational and has not been compromised by a manufacturing defect. Tests are normally carried out at the wafer level to cut out bad die, and then on the packaged parts.

Apart from the verification of internal gates, I/O integrity is also tested through completing the following tests:

- I/O-level test (i.e. checking the noise margin for TTL, ECL, or CMOS I/O pads).
- Speed test.

- I_{DD} test.

The last of the tests checks the leakage if the circuit is composed of complementary logic. Any value markedly above the expected value for a given wafer tests may be done at high speed or low speed due to possible power and ground bounce effects that may be present in older testers.

In general, manufacturing-test generation assumes that the circuit/chip functions correctly, and ways of exercising all gate inputs and of monitoring all gate outputs are required.

2.6 System partitioning

The problems of testing, particularly at the prototype stage, are generally eased if the system is sensibly partitioned into subsystems, each of which is as self-contained and independent as possible. For production items, also, it helps greatly if the subsystems can be checked out individually by providing the appropriate additional inlet/outlet pads for the test purposes. The test requirements for exhaustive testing of large digital systems are quite prohibitive if the system is tested as a whole.

2.7 Reset/Initialization

One simple but very effective aid to testing and testability is to design a reset facility into all digital systems of any complexity. This has the considerable advantage if setting all internal states to known values, and testing may then at least proceed from known conditions.

2.8 Design for testability

There are two key concepts to ensure that the designer considers the provision of means of setting or resetting key nodes in the system and of observing the response at key points.

The effects of testability or lack of it are such that it has been predicted that testability will soon become the main design criterion for VLSI circuits. The alternative is to save area by ignoring testability, but the penalties are such that even for modest complexity (i.e. 10,000 gates per chip) the test costs could rise by a factor of five to ten, compared with the same system designed for testability. Given that test is already a significant component of LSI chip costs, the effects will be quite dramatic and could well cause the test costs to exceed all other production cost by significant factor.

The inputs to the DUT are subjected to a test pattern (or test vector) which supplies a set of binary values, in combination and/or in sequence, to detect faults. The specification of the test vector sequences must involve the designer, while the generation and application of test patterns to a DUT are the problem faced by the test engineer. Test pattern generation is assisted by using automatic test pattern generation (ATPG), but they are complicated to use properly and ATPG costs tend to rise rapidly with the circuit size.

Once the application of a test pattern has revealed a fault, the process of diagnosis must be invoked to localize the fault.

2.9 Different types of fault

We are starting with the assumption that logically, the system performs its desired function, and that any faults occurring will be due to electrical problems associated with one or more of its component parts.

Two key concepts are of interest here, these are:

- Controllability
- Observability

During the design of a system, the designer must ensure that the test engineers have the means to set or reset key nodes in the system, that is, to control them. Equally as important is the requirement that the response to this control will be observable, that is, that we will be able to see clearly the effects of the test patterns applied.

- Controllability - Being able to set up known internal states.
- Combinatorial Testability - Being able to generate all states to fully exercise all combinations of circuit states.
- Observability - Being able to observe the effects of a state change as it occurs (preferably at the system primary outputs).

A fault is an actual defect that occurs in the device. When the test vector is applied to faulty device then incorrect response is produced. A fault may change the logic value from 0 to 1 or vice versa is called "Logical fault ". On the other hand if the fault causes some parameters of the circuit to change then it is termed as "Parametric fault".

A fault may also be categorized on the basis of duration for which it last. The categories are:

2.9.1Transient

Transient fault exits for a small duration. This fault is dominant cause of system failure. These type of fault is caused by -particle radiation, power supply fluctuation etc, but no permanent damage is done by this fault. As it exists for short duration, it is difficult to detect.

2.9.2 Intermittent

These type of fault appears regularly but not present continuously. Environmental effects like temperature and humidity variations cause this fault.

2.9.3 Permanent

If fault presents continuously. These faults are easy to detect. Shorts and opens of VLSI circuits cause them.

2.10 Fault modeling

Failures may occur in VLSI chips throughout their life cycle. Failures are caused by design errors, material defects, process defects, and extremes in operational environment, deterioration due to length of operation or age, and so on. Phenomena causing failures can be physical or chemical in nature. However, to analyze the faulty behavior and develop techniques to detect and locate failures, we use abstract fault models. Fault model allow cost-effective development of test stimuli that will identify failed chips and, if necessary, diagnose the failure.

Fault models also limit the number of tests as opposed to applying all possible inputs. Practical fault models depend upon the chip model, the technology, and, in some cases, the particular phase in the life cycle of the chip where analysis is conducted. By the chip we mean how the chip is described. Typically, one describes a VLSI chip at the following levels: specification, behavioral, functional, logic, circuit and layout. The translation between levels may be manual or automatic, but the complexity of the description grows in details as we move specification towards layout. Some of the causes of faults, which occur at any level, are errors in specification, errors in the translation from one level to another, errors in the manufacturing process, or material failures. Fault model must mimic the effects of these errors, yet they should easily analyzable. Compromises are often necessary to balance the complexity of a fault model necessary for accuracy against

the tractability of analysis. The guiding principle in arriving at a good compromise is to model most probable failures. The percentage of chips found faulty in the field is used as a measure to certify the adequacy of the fault model used in testing. In the sequel we will describe several fault models currently in use.

2.10.1 Stuck-at-faults

The most commonly used fault model to represent failures in logic circuits is the stuck at fault. A line in a logic circuit is an input or output of logic gate, and by fault "line l stuck-at-0" we mean that line l in the faulty circuit remains in the logic 0 state independent of the input to the circuit.

Similarly when any line of the circuit is at 1 then that is called stuck-at-1 fault. Another class of faults is the transistor-stuck faults. A 'stuck on' fault in a transistor represents a failure that causes the transistor to permanently turned on. A 'stuck open' fault represents a failure causing the transistor to remain permanently in the open or off state.

Since the only thing the line fault needs is a logical model, and most digital circuits, irrespective of the specific technology, can be modeled at this level, these faults are often called logical faults or technology independent faults. In contrast, the transistor-stuck faults are specific to the MOS technology and are often referred to as non-classical faults; stuck faults are regarded as the classical model.

2.10.2 Shorts and open

The diminishing feature size allows increased circuit density. Certain failures in high density VLSI chips require fault models that are different from the stuck model. A short or Bridging fault is defined as an electrical short circuit between two nodes that are supposed to be electrically isolated. Actually a short fault causes those two nodes to have

the same voltage at all times. An open fault represents a failure that causes a line or wire in the circuit to be broken.

2.10.3 Delay fault

Even a circuit is free from all structural defects; it may not propagate a signal in the proper time allowed. This is termed as delay fault. The voltage in the delay line could either be slow-to-rise (STR) or slow-to-fall (STF). Two types of delay fault models are generally used:

- Gate delay model
- Path delay model

In the Gate delay model, delay defects at the inputs or outputs of a gate. On the other hand, the Path delay model models those defects which cause cumulative propagation delays along the circuit path to exceed the specified value. Path delay model needs increasing number of test generation and test generation time however gate delay model does not have this problem.

2.10.4 Functional faults

It often happens to the user of a complex VLSI chip that he/she does not have access to a detailed circuit or logic level description of the chip. In those situations, functional faults that model failures at the register level or processor instruction level can be used. Such model developed for microprocessors, have proved effective in deriving tests that cover a high percentage of gate level stuck faults. Functional faults have also been used for cellular logic arrays and sequential machines. For example, an m input cell in a logic array may be assumed to realize any other m-input function, or the state table of a sequential machine is assumed to have an erroneous entries.

Chapter- 3

Multiplier Algorithm and Architecture

3.1Introduction

Multiplier design starts with the elementary school algorithm for multiplication. At each step, we multiply one digit of the multiplier by the full multiplicand; we add the result, shifted by the proper number of bits, to the partial product. When we run out of digits, we are done. The computation of partial products and their accumulation into the complete product can be optimized in many ways. We have chosen the Carry Save Array architecture for implementing a C-testable and easily synthesizable M X N bit multiplier.

3.2Carry save Array Multiplier

The elementary multiplication algorithm suggests a logic and layout structure for a multiplier which is surprisingly well suited to VLSI implementation-the array multiplier. The logic structure is shown in the parallelogram form both to simplify the drawing of wires between stages and also to emphasize the relationship between the array and the basic multiplication steps shown in fig 3.1. As when multiplying by hand, partial products are formed in rows and accumulated in columns, with partial products shifted by the appropriate amount. The partial products are called summand. In a carry save array multiplier the summands are collected through a carry save adders. At the bottom of the array an adder is used to convert the carry save form to the required form of output.

Fig.3.1shows the proposed carry save array multiplier. For simplicity 4X4 bit version is shown in the figure and the basic cell by which the multiplier is constructed is shown in the left side of the diagram.

The Boolean function for the basic cell output is:

x = (a and b) *xor* c *xor* d;

y = ((a *and* b) *and* c) or (c *and* d) or ((a *and* b) *and* d);

The proposed multiplier has two parts. One is for converting the operands in two's complement form when the operand is a signed number. The operand a propagate vertically and the operand b propagate horizontally. When the operand A is a negative number then the signal Asgnm is 1. At the 1st row of the multiplier the cell inputs a and b are shorted together and both are the same i.e only one bit of operand a. Asgnm signal is given to c input and d input is grounded. so the Boolean equation for x becomes

x = a1(i) *xor* 1

Hence output of the 1st row will results in 1's complement of the inputs a1(i).here input to the 2's complement block is assumed as a1(i) and input to the main multiplier after 2nd row is a(i) and in the 2nd row only 1 is added to the 1st row's results. So after the 2nd row we get the 2's complement of the a1(i). But a1(i) will remain same if Asgnm is 0 i.e when the

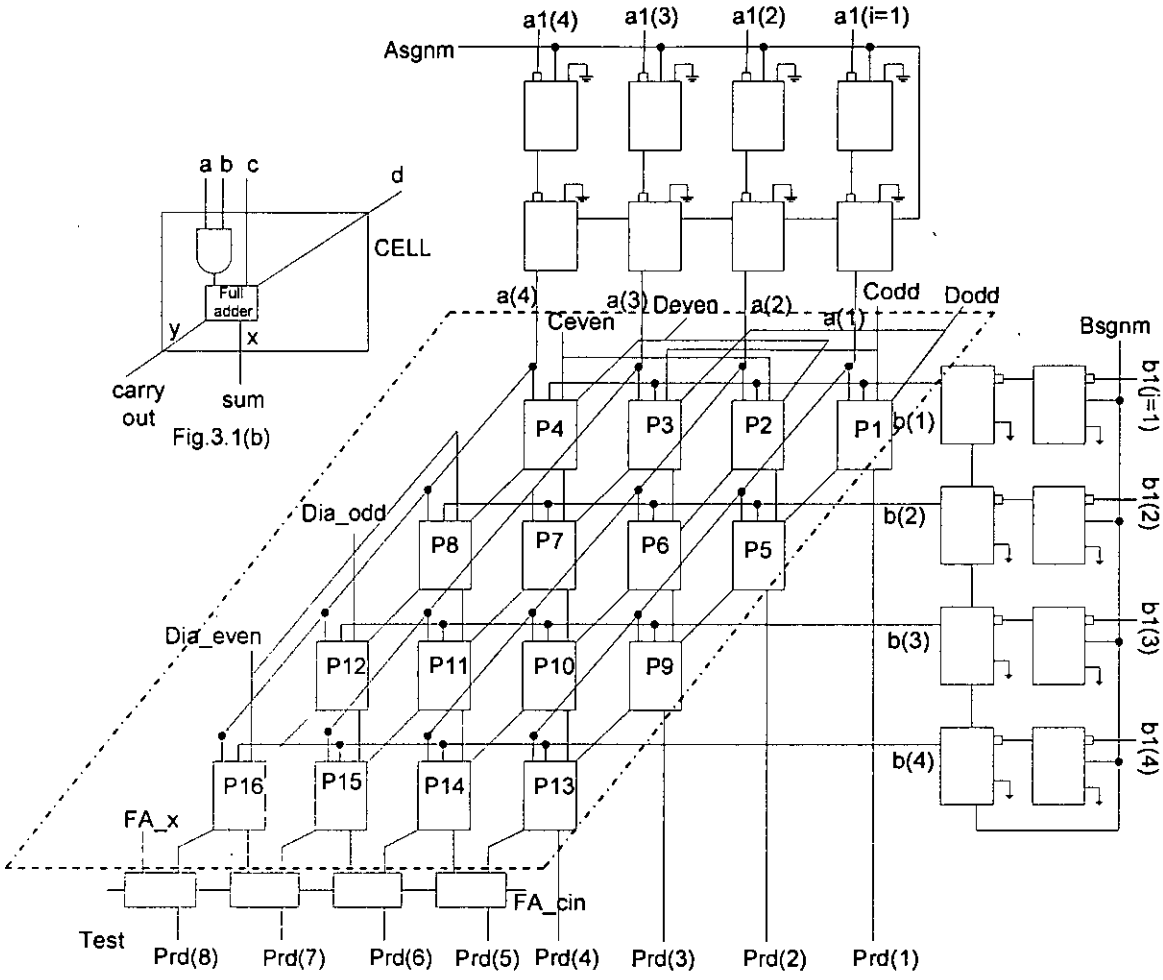


Fig.3.1(a)

Fig: 3.1 (a)A 4x4 Carry Save Array Multiplier (for signed and unsigned numbers), (b) Basic cell

operand A is an unsigned number. The same happens for the operand B.

The main architectural block of the CSA is shown in the dotted box. Then in the cell block the output of each cell is x term and y term. x term propagate to the next stage vertically and y term propagate diagonally to the next stage. The n x n Carry Save Array multiplier receives two n-bit operands and produces a 2n-bit product consisting n^2 cells

arranged as n rows with n cell each. There are n rows and 2n-1 diagonals in this arrangement. A cell is denoted (i, j) if it is in the i^{th} row and j^{th} diagonal.

The primary inputs to the CS array multiplier consists of following five n-bit vectors

$a = (a_{n-1}, a_{n-2}, \ldots\ldots\ldots a_1, a_0)$

$b = (b_{n-1}, b_{n-2}, \ldots\ldots\ldots b_1, b_0)$

$c = (c_{n-1}, c_{n-2}, \ldots\ldots\ldots c_1, c_0)$

$d = (d_{n-1}, d_{n-2}, \ldots\ldots\ldots d_1, d_0)$

$c_{n-1} = (c_{n-1\,n-1}, c_{n-2\,n-1}, \ldots\ c_{0\,n-1})$

Where a and b are main inputs to the array multiplier, i.e. multiplicand and multiplier. The rest of them are used only for testing purpose. So during multiplication these inputs are kept '0'.Each test pattern in CSA multiplier is denoted by T_{cs} [a, b, c, d, c_{n-1}]. Each cell-input vector can be represented by a binary 4-tuple <a, b, c, d>. The pattern V_k stands for test pattern T_k <a, b, c, d >. The other pin c, d, c_{n-1} is used for testing purpose. During multiplication these input are kept at 0.

The A and B pins are used for multiplicand and the multiplier with the bit size of n. The main advantages of symmetrical input test vectors are utilized in this program algorithm. All c, d, c_{n-1} inputs of basic cells are divided into two categories. Codd and Ceven for c, Dodd and Deven for d and Dia_odd and Dia_even for C_{n-1} inputs. FA_Cin and FA_x pins are used to control the inputs of Full Adder block during testing the multiplier.

Testability of the proposed multiplier

4.1 C-testability

A carry-save array multiplier is a regularly structured circuit. So it is easily testable. An iterative logic array (ILA) is a logic circuit that consists of regular array of identical cell. For testing the structured circuit it is necessary to test all the cells in a general ILA. If the test pattern needed for testing the entire circuit is independent of size of the array, then the circuit is said to be C-testable.

For example a full adder can be fully tested with only eight test patterns. Because a full adder has only three input signal. The basic building block or cell used for most of the array multiplier is a full adder with a 2-input *and* gate connected to one of the three inputs of the full adder. The basic cell is shown in Fig 4.1. The cell has four inputs, labeled as *a*, *b*, *c*, and *d* and two outputs, labeled as x and y. x and y stand for sum and carry output respectively of the full adder.

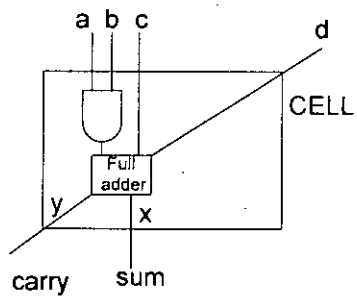


Fig.4.1 Basic cell structure

The principle aim of testing approach is to exhaustively test each cell in an array. But it is impractical to test exhaustively the entire array, as it is time consuming. By taking the

advantage of the structure of ILA, every cell in the array can be simultaneously tested.

Fault model in this thesis assumes:

a) The fault is permanent fault (i.e. the fault permanently changes the circuit's logic characteristics)

b) The fault may alter the cell's output functions in any arbitrary way, as long as the faulty cell remains on the combinational circuit.

With these assumptions for exhaustively testing each cell in the array multiplier requires all possible inputs patterns and observing all outputs.

- Conditions for an array multiplier to be testable:

1) All possible input vectors (2^4) must be applied to every cell in the array.

2) For each cell input vector applied to a cell under test any faulty signal produced at the output of the cell can be propagated to primary output of the array.

- Conditions for an array multiplier to be C-testable:

An array multiplier is C-testable if it is testable and number of test pattern required is constant and independent of the size of the array multiplier.

4.2 Test Vector (*Test pattern*) Generation

In VLSI circuits, we have a high ratio of logic gates to pins on the device, there is generally no way of accessing most of the logic, so we cannot directly probe the internals of the device. Because of this problem, we need a way of generating tests which, when applied to the inputs of a circuit, give a set of signals which indicate whether or not the device is good or faulty. The set of stimulus input and expected output pattern is called a "*Test Vector*". The test vectors distinguish between the good machine and the faulted machine.

The n x n Carry Save array multiplier receives two n-bit operands and produces a 2n-bit product consisting n^2 cells arranged as n rows with n cell each. During multiplication the c and d inputs of the top row and the c inputs of the leftmost diagonal are kept at 0. But during testing these inputs are available at primary inputs. There are n rows and 2n-1 diagonals in this arrangement. A cell is denoted (i, j) if it is in the ith row and jth diagonal. A 4 x 4 Carry Save array multiplier is shown in fig. 3.1 in chapter-3.

As we need to exhaustively test each cell, the test pattern must be such that for each pattern, every cell in the array has same input. So if there is no fault then each cell will have same inputs as well as same outputs. Test pattern for our proposed array multiplier can be denoted by $T_i[a,b,c,d,c_{n-1}]$. T_i represents the test pattern for input $V_i[a,b,c,d]$. In this case as the basic cell of carry-save array multiplier has four input signals, so sixteen input combinations are possible, we will first test the *and* gate by applying various input combinations of *a* and *b* which are [00,01,10,11] then from testability point of view, inputs to the basic cell are now reduced to three i.e< out put of the *and* gate,c,d>. While testing the *and* gate '0' is given to both *c* and *d* inputs. For testing the *and* gate the required test patterns are as follows:

1. T0 [0000]
2. T4 [0100]
3. T8 [1000]
4. T12 [1100]

After the and gate is tested then various combinations of the inputs are given to the other two inputs c and d. With all of these combinations of inputs if the results of the array are

correct then we can consider that the multiplier has no fault. Depending on the response test patterns are categorized in the following three groups.

Group-a:

Each test patterns of this group is denoted by T_i [a, b, c, d, c_{n-1}]. T_i represents the test pattern for input v_i .which means that input and output of the every cell of every row is same. Following four test patterns can be applied to every cell in the array.

1) T_0 = [<00...00> <00...00> <00...00> <00...00> <00...00>]

2) T_2 = [<00...00> <00...00> <11...11> <00...00> <11...11>]

3) T_4 = [<00...00> <11...11> <00...00> <00...00> <00...00>]

4) T_{15} = [<11...11> <11...11> <11...11> <11...11> <11...11>]

Input and output vectors for the test pattern T_{15} and T_0 of this group are shown in fig 4.2 (a) and Fig 4.2(b) respectively. Here if any vector is like <0101>which means that every odd position bit is 1 and every even position bit is 0. and position of LSB is considered here '1'.

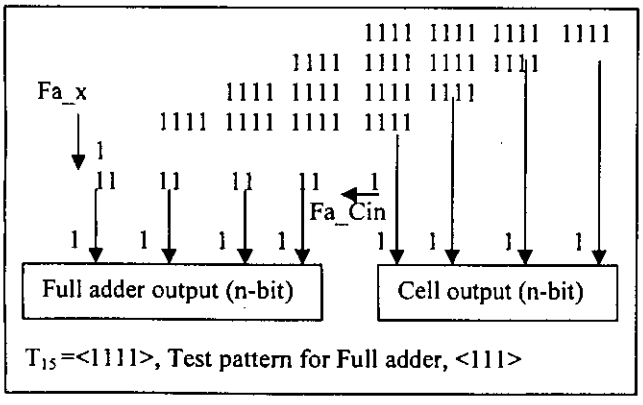


Fig. 4.2(a) Bit conditions for T_{15}

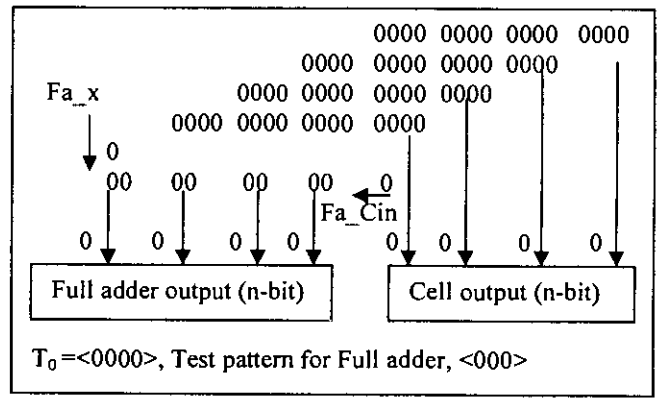


Fig. 4.3(b) Bit conditions for T_0

Group-b:

Each test pattern in this group is denoted by $T_{i,j}$,which represents one input vector v_i ,to one half in the array and another input vector v_j ,to another half in the array. Each test pattern $T_{i,j}$ has a companion test pattern $T_{j,i}$ which applies v_j (v_i) those cell having input vector v_i (v_j) under $T_{i,j}$ test pattern. There are two test pattern in this group.

1) $T_{9,14}$ = [<11...11> <10...10> <00...00> <11...11> <10...10>]

2) $T_{14,9}$ = [<11...11> <01...01> <11...11> <00...00> <01...01>]

Input and output vectors of array multiplier for these two patterns are shown in Fig.4.3(a) and Fig. 4.3 (b)

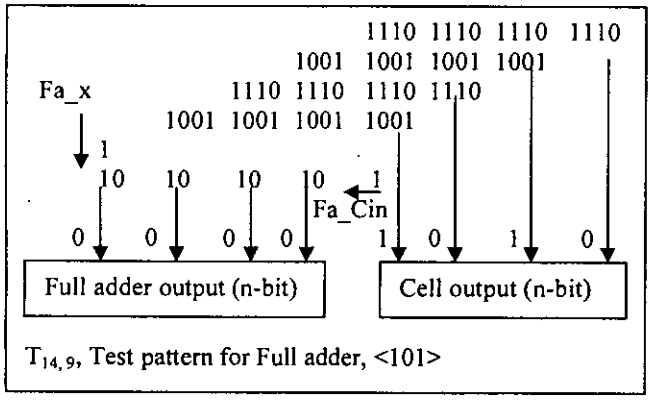


Fig.4.3 (a) Bit conditions for $T_{14,9}$

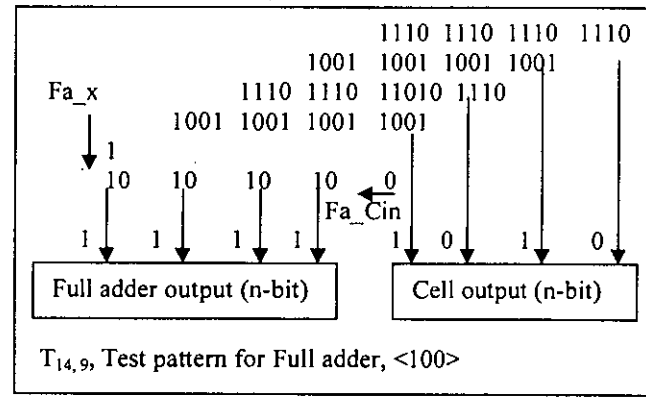


Fig.4.3 (b) Bit conditions for $T_{14,9}$

Group-c:

The test pattern of this group is denoted by $T_{i,j,k,l}$,which applies four input vector v_i , v_j , v_k ,v_l to one-fourth in the array. $T_{j,k,l,i}$,$T_{k,l,i,j}$,$T_{l,i,j,k}$ together with $T_{i,j,k,l}$ can be used to apply four input vector v_i , v_j , v_k ,v_l in the array. In this group there are four test patterns.

1) $T_{8,11,12,13}$ = [<11...11> <01...01> <00...00> <01...01> <00...00>]

2) $T_{12,13,8,11}$ = [<11...11> <10...10> <01...01> <01...01> <00...00>]

3) $T_{11,8,13,12}$ = [<11...11> <01...01> <00...00> <10...10> <00...00>]

4) $T_{13,12,11,8}$ = [<11...11> <10...10> <10...10> <10...10> <01...01>]

Input and output vectors of array multiplier for the test pattern $T_{13,12,11,8}$ are shown in Fig.4.4.

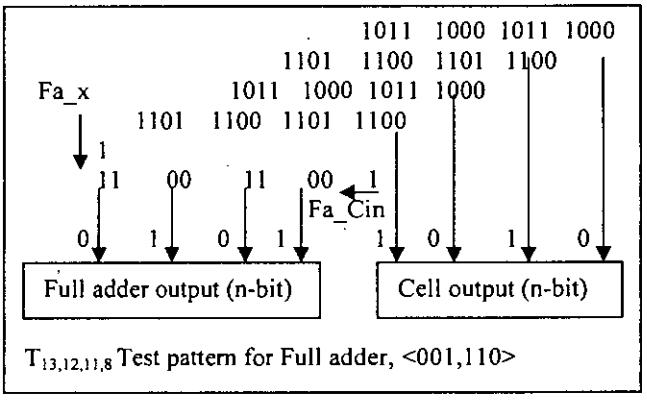


Fig.4.4 Bit conditions for patterns $T_{13,12,11,8}$

So using these ten test patterns our proposed Carry Save Array Multiplier can be tested independent on the size of the array.

4.3 Test patterns for testing the multiplier

These following test patterns are identical for both odd and even n.

Table 4.3: Patterns for both odd and even n

Test Pattern	a (n- bit)	b (n-bit)	c (n-bit)	d (n-bit)	c_{n-1} (n-bit)
T_0	<00…00>	<00…00>	<00…00>	<00…00>	<00…00>
T_2	<00…00>	<00…00>	<11…11>	<00…00>	<11…11>
T_4	<00…00>	<11…11>	<00…00>	<00…00>	<00…00>
T_{15}	<11…11>	<11…11>	<11…11>	<11…11>	<11…11>
$T_{9,14}$	<11…11>	<10…10>	<00…00>	<11…11>	<10…10>
$T_{14,9}$	<11…11>	<01…01>	<11…11>	<00…00>	<01…01>

For this following group, test pattern differs for different values of n.

Table 4.4: Test patterns for EVEN-n

Test Pattern	a (n-bit)	b (n-bit)	c (n-bit)	d (n-bit)	c_{n-1} (n-bit)
$T_{8,11,12,13}$	<11…11>	<01…01>	<00…00>	<01…01>	<00…00>
$T_{12,13,8,11}$	<11…11>	<10…10>	<01…01>	<01…01>	<00…00>
$T_{11,8,13,12}$	<11…11>	<01…01>	<00…00>	<10…10>	<00…00>
$T_{13,12,11,8}$	<11…11>	<10…10>	<10…10>	<10…10>	<01…01>

Table 4.5: Test patterns for ODD-n

Test Pattern	a (n-bit)	b (n-bit)	c (n-bit)	d (n-bit)	c_{n-1} (n-bit)
$T_{8,11,12,13}$	<11…11>	<10…01>	<00…00>	<10…01>	<01…10>
$T_{12,13,8,11}$	<11…11>	<01…10>	<10…01>	<10…01>	<01…01>
$T_{11,8,13,12}$	<11…11>	<10…01>	<00…00>	<01…10>	<00…00>
$T_{13,12,11,8}$	<11…11>	<01…10>	<01…10>	<01…10>	<00…00>

Test pattern applied to Full adder for exhaustively testing:

Table 4.6: Test patterns for EVEN-n

Test pattern for FA	Control bit for FA testing		Test pattern for CSA T_{cs}
	FA_Cin	FA_x	
000	0	0	T_0, T_4
001,110	1	1	$T_{13,12,11,8}$
010	0	0	$T_{9,14}$
011	1	0	$T_{9,14}$
100	0	1	$T_2, T_{14,9}$
101	1	1	$T_2, T_{14,9}$
110,001	0	0	$T_{12,13,8,11}$
111	1	1	T_{15}

Table 3.7: Test patterns for ODD-n

Test pattern for FA	Control bit for FA testing		Test pattern for CSA T_{cs}
	FA_Cin	FA_x	
000	0	0	T_0, T_4
001,110	1	0	$T_{11,8,13,12}$
010	0	0	$T_{14,9}$
011	1	0	$T_{14,9}$
100	0	1	T_2
101	1	1	T_2
110,001	0	1	$T_{8,11,12,13}$
111	1	1	T_{15}

The test pattern applied to full adder for exhaustively testing comes from the test pattern applied to the multiplier circuit. So we need not to be worried about the test pattern for the full adder. But we have to control the full adder input in such a way that each full adder in the full adder block gets the same input. For this task we need only two control bits, FA_x and FA_Cin, where FA_Cin is the Carry input for the first full adder and FA_x is the x input of the left most full adder in the full adder block. The table 3.6 and 3.7 cover the test pattern needed for both Even and Odd n. With the application of test pattern in the multiplier and for different combination of Full adder control bits, different output pattern is shown in the Fig.4.2, Fig. 4.3 and Fig. 4.4 for the operand *a* and *b* size of 4.Output patterns for both even and odd n for ten test patterns applied is shown in the table below:

Table 4.7 Output patterns for the applied test patterns for the n x n Carry save Array Multiplier

Test pattern T_{cs}	n-Even		Output pattern (2n-bit)		n-Odd		Output pattern (2n-bit)	
	Fa_Cin	Fa_x	FA out (n-bit)	Cell out (n-bit)	Fa_Cin	Fa_x	FA out (n-bit)	Cell out (n-bit)
T_0	0	0	<00..00>	<00..00>	0	0	<00..00>	<00..00>
T_2	0	1	<11..11>	<11..11>	0	1	<11..11>	<11..11>
	1		<00..00>		1		<00..00>	
T_4	0	0	<00..00>	<00..00>	0	0	<00..00>	<00..00>
T_{15}	1	1	<11..11>	<11..11>	1	1	<11..11>	<11..11>
$T_{14,9}$	0	1	<11..11>	<10..10>	0	0	<11..11>	<01..10>
	1		<00..00>		1		<00..00>	
$T_{9,14}$	0	0	<11..11>	<01..01>	0	1	<11..11>	<10..01>
	1		<00..00>		1		<00..00>	
$T_{8,11,12,13}$	0	0	<01..01>	<00..00>	0	1	<01..10>	<00..00>
$T_{12,13,8,11}$	0	0	<01..10>	<00..00>	0	0	<10..01>	<00..00>
$T_{11,8,13,12}$	0	0	<10..10>	<01..01>	1	0	<10..01>	<10..01>
$T_{13,12,11,8}$	1	1	<01..01>	<10..10>	0	0	<01..10>	<01..10>

In the output pattern, FA out and Cell out means the output bits of the Full adder block and Cell output respectively. As the operand size is n for both *a* and *b*, so the output of the full adder block will be n-bits and output of the cell will also be n-bits.

Different signals are as follows:

FA_cin: Carry input of rightmost fulladder connected with last row of multiplier cells.

FA_x: x input of leftmost full adder connected with last row of multiplier cells.

Codd: All odd 'c' input of each cells are ANDed together to get this input pin.

Ceven: All even 'c' input of each cells are ANDed together to get this input pin.

Dodd: All odd 'd' input of each cells are ANDed together to get this input pin.

Deven: All even 'd' input of each cells are ANDed together to get this input pin.

a: Give 'a' input vector like <000…0> or <1111…1>, Here all bits should be same.

bodd: all odd positioned bits should be of the same type.

beven: all even positioned bits should be of the same type. If bodd=1 and beven=0 then 'b' input vector is<…010101>

Dia_even: All even 'd' input of each cells of leftmost diagonal are ANDed together to get this input pin.

Dia_ odd: All odd 'd' input of each cells of leftmost diagonal are ANDed together to get this input pin.

VHDL overview

5.1 Introduction

VHDL is a general-purpose programming language as well as a hardware description language, so it is possible to create VHDL simulation programs ranging in abstraction from gate level to system. VHDL has a rich verbose syntax that makes its models appear to be long and verbose. However, VHDL models are relatively easy to understand once one is used to the syntax. VHDL stands for VHSIC (Very high-speed integrated circuits) Hardware description language. It is advantageous to express complex digital design & system for both simulation and synthesis.

VHDL has the features to create re-usable circuit building blocks in larger circuits. VHDL fits into overall electronic design process, particularly as the process relates to FPGA, PLD and ASIC design problem.

VHDL includes a rich set of control and data representation features. One of the important applications of VHDL is to test the performance of a circuit in the form of Testbench. By using Test bench along the description of the circuit, the expected output of the circuit can be verified over a period of time.VHDL was developed by committee

intended for documenting digital hardware behaviorally. The intent for the language was solely for the explicit purpose of documentation. A documentation language for its digital designs has provided the initial momentum. In an effort to focus its use on practical applications and to expand beyond the use as simply a documentation language, it has been through numerous iterations. During this refinement process it has become IEEE standard 1076 in 1987.

5.2 History of VHDL

In past the language to describe the Hardware was Verylog. In the early 1980, VHDL was developed, funded by U.S department of Defense. During the VHSIC program, researchers were confronted with the daunting task of describing circuits of enormous scale and managing very large circuit design problems that involved multiple teams of engineers. With only gate-level design tools available, it soon became clear that better, more structured design methods and tools would be needed.

To meet the challenge, a team of engineers from three companies – IBM, Texas and Intermetrics were contracted by the department of Defense to complete the specification and implementation of a new, language-based design description method in July 1983.

- In August 1985, the final version of the language under government contract was released : VHDL version 7.2
- In December 1987, VHDL became IEEE standard 1076-1987 and in 1988 an ANSI standard.

- In September 1993, VHDL was restandardized to clarify and enhance the language.
- VHDL has been accepted as a Draft International Standard by the IEC.

5.3 Benefits of VHDL

- Allows for various design methodologies.
- Provides technology independence.
- Describes a wide variety of digital hardware.
- Eases communication through standard language.
- Allows for better design management.
- Provides a flexible design language.
- Has given rise to derivatives standards:

  □ WAVES, VITAL, Analog VHDL

5.4 Compiler used for VHDL language

One of the easiest and most sophisticated compilers to use VHDL system is MAX+PLUS II software. This software will run on several different types of computer system like Windows-95, Windows-98 or Windows NT. Using this software design of logic circuits can be performed by using schematic capture, writing VHDL code and using a truth table.

Other software for compiling the VHDL system is Peak Accolade VHDL, Active-HDL etc.

The program in this thesis paper has been written in VHDL code in MAX+PLUS II (Version 9.23) software environment. Any VHDL file in this environment must be saved with the extension of vhd. If several leaf cells are used in designing the digital systems then all VHDL file must saved in the same directory. In the MAX+PLUS II software environment the VHDL code written in the text editor and can also be served as a default extension of tdf, which stands for TEXT DESIGN FILE. It is used for files that contains the source code written in the Altera Hardware Description Language (AHDL), which is another language supported by MAX+PLUS II system.

MAX+PLUS II software is a fully integrated, architecture-independent package for designing logic with Altera programmable logic devices, including Classic™, MAX® 5000, MAX 7000, MAX 9000, FLEX® 6000, FLEX 8000, and FLEX 10K devices. MAX+PLUS II offers a full spectrum of logic design capabilities: three design entry methods for hierarchical designs; floorplan editing; powerful logic synthesis; design partitioning; functional, timing, and board-level-type linked simulation; detailed timing analysis; automatic error location; and device programming and verification. MAX+PLUS II also reads standard EDIF netlist files, VHDL netlist files, Verilog HDL netlist files, OrCAD Schematic Files, and Xilinx Netlist Format Files, and writes EDIF, VHDL, and Verilog HDL netlist files, including VITAL-compliant files, for a convenient interface to other industry-standard CAE software. In addition, MAX+PLUS II for UNIX workstations allows running the Synopsys Design Compiler and FPGA Compiler automatically, which allows processing both VHDL and Verilog HDL designs. The MAX+PLUS II Compiler ensures that a design--called a project in MAX+PLUS II--fits into the device architecture in the most efficient way possible.

5.4.1 MAX+PLUS II Highlights

MAX+PLUS II offers rich graphical user interface complemented with an illustrated, easy-to-use on-line help system. The complete MAX+PLUS II system includes ten fully integrated applications that take through every step from design entry to device programming.

Many features and commands are shared by the different MAX+PLUS II applications. For example, using identical commands in each MAX+PLUS II application to open files, to assign project devices, and to begin compiling the current project. The design editors in MAX+PLUS II--the Graphic, Text, and Waveform Editors--and the auxiliary editors--the Floorplan and Symbol Editors--also share numerous design entry tools and features. Each editor allows performing similar tasks, such as assigning a pin, in the same way.

We can open multiple design files and transfer information between them, while simultaneously compiling or simulating another project. We can view an entire hierarchy of design files and move smoothly from one hierarchical level to another. As we open a design files, MAX+PLUS II automatically starts the appropriate design editor.

The MAX+PLUS II Compiler lies at the heart of the MAX+PLUS II system, providing powerful design processing to customize to achieve the best possible silicon implementation of our project. Automatic error location and extensive documentation on error and warning messages make design modifications as simple as possible. We can create output files in a variety of formats for simulation, timing analysis, and device

programming, including EDIF, Verilog HDL, and VHDL files for use with other industry-standard EDA tools. At every step in the design process, MAX+PLUS II software makes it easy for us to focus on our design--not on how to use the software.

The superb integration of MAX+PLUS II software improves our efficiency and productivity, putting us in control of our logic design environment.

5.5 Standard Logic 1164

- The LIBRARY statement is used to reference a group of previously defined VHDL design units

  (other entities or groups procedures/functions known as 'packages'.
- The USE statement specifies what entities or packages to use out of this library; in this case

  'USE IEEE.std_logic_1164.all' imports all procedures/functions in the *std_logic_1164* package.
- The *std_logic_1164* package defines a multi-valued logic system, which will be used as the data types for the signals defined in our examples.
- The VHDL language definition had a built-in bit type which only supported two values, '1' and '0' which was insufficient for modeling and synthesis applications.
- The 1164 standard defines a 9-valued logic system; only 4 of these have meaning for synthesis:

  '1', '0', 'Z' (high impedance), '-' (don't care).
- The 1164 single bit type *std_logic* and vector type *std_logic_vector* (for busses) will be used for all signal types in the tutorial examples.

5.6 Primary VHDL constructs we will use for synthesis

- signal assignment
  Nextstate <= HIGHWAY_GREEN

- comparisons
  = (equal), /= (not equal),
  > (greater than), < (less than)
  <= (less than or equal), >= (greater than or equal)

- logical operators
  (and, xor, or, nand, nor, xnor, not)

- 'if' statement
  if (presentstate = CHECK_CAR) **then** ....
  end if | **elsif** ....

- 'for' statement (used for looping in creating arrays of elements)

- Other constructs are **'when else'**, 'case', **'wait '**. Also **": ="** for variable assignment.

5.7General Comments on VHDL Syntax

Most syntax details will be introduced on an 'as-needed' basis. The full syntax of a statement type including all of its various options will often NOT be presented.

Generalities:

- VHDL is not case sensitive.

- The semicolon is used to indicate termination of a statement.

- Two dashes ('--') are used to indicate the start of a comment.

- Identifiers must begin with a letter, subsequent characters must be alphanumeric or '_' (underscore).

- VHDL is a strongly typed language. There is very little automatic type conversion; most operations have to operate on common types. Operator overloading is supported in which a function or procedure can be defined differently for different argument lists.

5.8 Model Template

Entity *model_name* is
Port
(
    list of inputs and outputs
);

end *model_name*;

architecture *architecture_name* of *model_name* is

begin

    VHDL concurrent statements

end *architecture_name* ;

5.8.1 Logic diagram of full adder

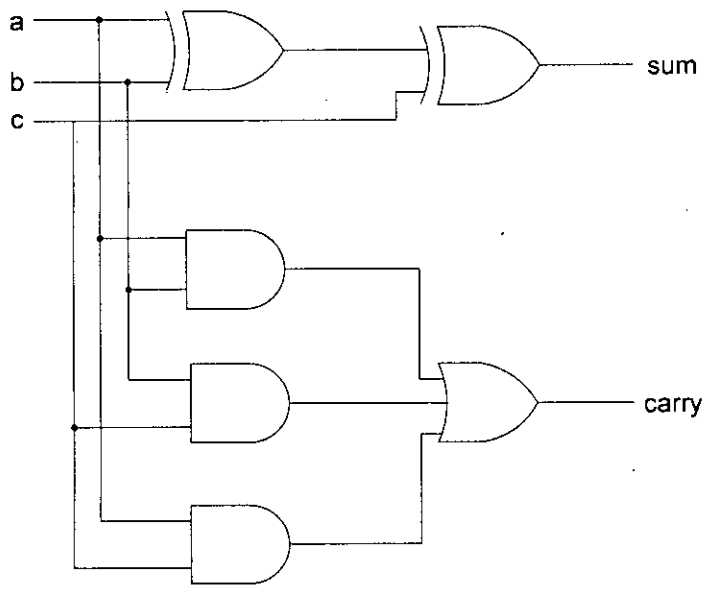


Fig. 5.1 Logic diagram of full adder

VHDL code for full adder:

```
library ieee;
use ieee.std_logic_1164.all;
entity fulladder is
port(X,Y,Cin:in std_logic;
        Cout,Sum:out std_logic);
end fulladder;
architecture concurrent of fulladder is
begin
Sum<=X xor Y xor Cin;
Cout<=(X and Y) or (X and Cin) or (Y and Cin);
end concurrent;
```

5.8.2 Logic diagram of basic cell:

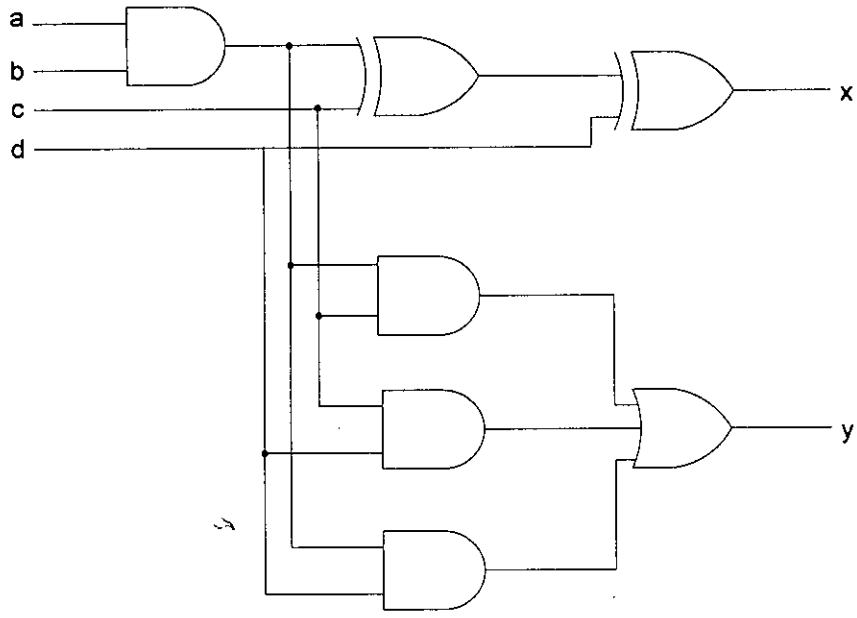


Fig.5.2 logic diagram of basic cell

VHDL code for basic cell:

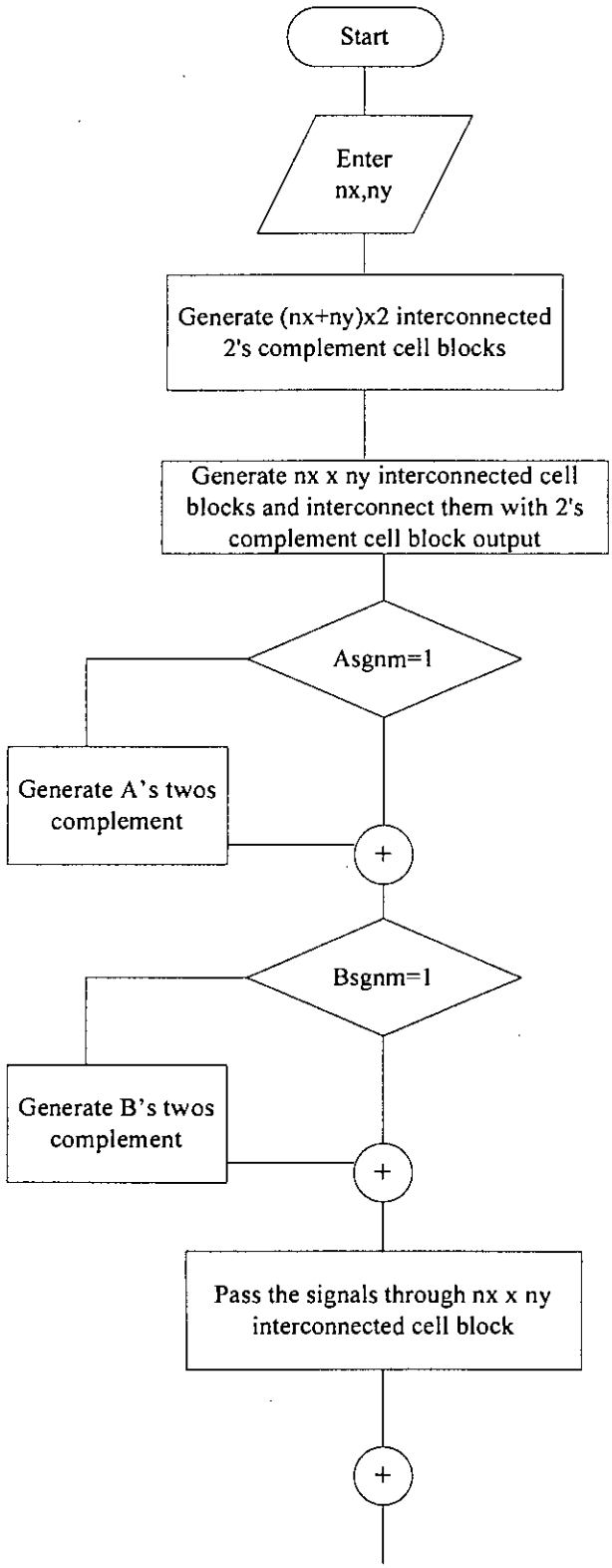
```
architecture structure of cell is
component fulladder
port(X,Y,Cin:in std_logic;
   Cout,Sum:out std_logic);
end component;
signal andout:std_logic;
signal faout:std_logic;
signal modsig: std_logic;
begin
andout<=a and b;
```

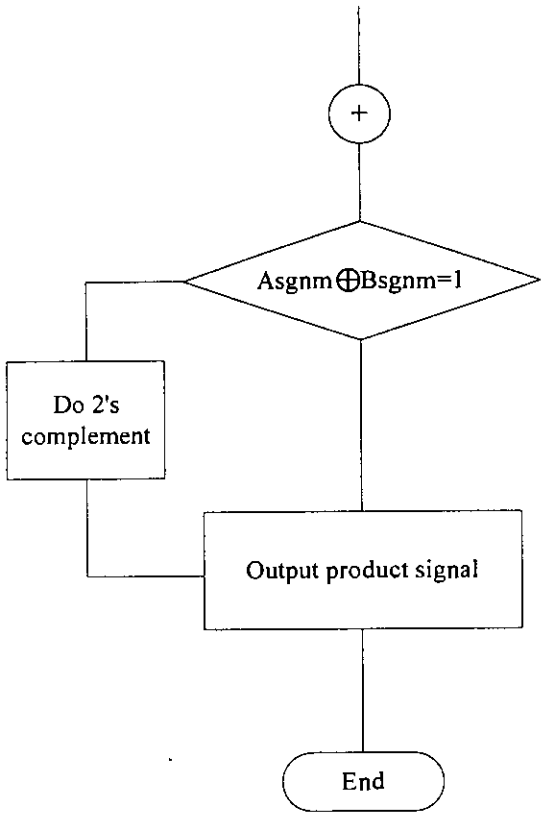
```
fulladder1:fulladder port map (X=>andout,Y=>c,Cin=>d,Cout=>faout,Sum=>x);

y<=faout;

end structure;
```

5.9 Program algorithm





5.10 VHDL code for the multiplier:

VHDL code for carry save array multiplier

```
----------------------------------------

library ieee;

use ieee.std_logic_1164.all;

entity MCSA_Mult is

generic(nx:integer:=4;ny:integer:=4);

port (    A1        :in std_logic_vector(nx downto 1);

          B1        :in std_logic_vector(ny downto 1);

          Codd,Ceven    :in std_logic;
```

```
    Dodd,Deven    :in std_logic;
    Asgnm,Bsgnm   :in std_logic;
----------------------
   Dia_odd,Dia_even :in std_logic;
   FA_Cin,FA_x    :in std_logic;
    Test      :out std_logic;
    Sumn      :out std_logic_vector(nx+ny downto 1));
end MCSA_Mult;
architecture MCSA of MCSA_Mult is
------------------------
component cell
port(a,b,c,d:in std_logic;
        x,y:out std_logic);
end component;
------------------------
component fulladder
port(X,Y,Cin:in std_logic;
        Cout,Sum:out std_logic);
end component;
-------------------------
constant high:std_logic:='1';
constant low:std_logic:='0';
signal Coutm : std_logic_vector(nx downto 1);
signal temp1 : std_logic_vector(nx downto 1);
```

```
signal temp2 : std_logic_vector(nx downto 1);
signal temp3 : std_logic_vector(nx downto 1);
signal Coutmy : std_logic_vector(ny downto 1);
signal temp4 : std_logic_vector(ny downto 1);
signal temp5 : std_logic_vector(ny downto 1);
signal temp6 : std_logic_vector(ny downto 1);
signal  A    : std_logic_vector(nx downto 1);
signal  B    : std_logic_vector(ny downto 1);
----------
signal test1 : std_logic;
signal test2,test3 : std_logic;
signal high1 : std_logic;
signal  c0   : std_logic_vector(nx-1 downto 1);
signal  x    : std_logic_vector(nx*ny downto 1);
signal  y    : std_logic_vector(nx*ny downto 1);
signal  prd  : std_logic_vector(nx+ny+1 downto 1);
signal  sgn  : std_logic;
signal  cxor : std_logic_vector(nx+ny+1 downto 1);
signal onecom: std_logic_vector(nx+ny downto 1);
begin
test1<=Asgnm;
test3<=Bsgnm;
test2<=low;
```

```
M:
for i in 1 to nx generate
M1:if i<=nx generate
M2:cell port map(A1(i),A1(i),test1,test2,temp1(i),temp2(i));
--A(i)<=temp1(i);
end generate;
end generate;
M3:
for i in 1 to nx generate
M4:if i=1 generate
M5:cell port map(temp1(i),temp1(i),test2,test1,A(i),temp3(i));
Coutm(i)<=temp3(i);
end generate;
M6:if i>1 generate
M7:cell port map(temp1(i),temp1(i),test2,Coutm(i-1),A(i),temp3(i));
Coutm(i)<=temp3(i);
end generate;
end generate;
---------- For y ----------
yM:
for i in 1 to ny generate
yM1:if i<=ny generate
yM2:cell port map(B1(i),B1(i),test3,test2,temp4(i),temp5(i));
--A(i)<=temp1(i);
```

```
end generate;
end generate;
yM3:
for i in 1 to ny generate
yM4:if i=1 generate
yM5:cell port map(temp4(i),temp4(i),test2,test3,B(i),temp6(i));
Coutmy(i)<=temp6(i);
end generate;
yM6:if i>1 generate
yM7:cell port map(temp4(i),temp4(i),test2,Coutmy(i-1),B(i),temp6(i));
Coutmy(i)<=temp6(i);
end generate;
end generate;
------------------------
label1:
   for j in ny downto 1 generate
label2:
   for i in nx downto 1 generate
--------------------------------
c1:if j=1 and i=1 generate
        P1:cell port map(A(i),B(j),Codd,Dodd,x(j),y(i+(j-1)*ny));
   prd(j)<=x(j);
end generate;
--------------------------------
```

```
c2a:if j=1 and i>1 and i-2*(i/2)=1 generate
        P2:cell port map(A(i),B(j),Codd,Dodd,x(i+(j-1)*ny),y(i+(j-1)*ny));
end generate;
c2b:if j=1 and i>1 and i-2*(i/2)=0 generate
        P2:cell port map(A(i),B(j),Ceven,Deven,x(i+(j-1)*ny),y(i+(j-1)*ny));
end generate;
--------------------------------
c3:if j>1 and i=1 and j<ny generate
        P3:cell port map(A(i),B(j),x(i+1+(j-2)*ny),y(i+(j-2)*ny),x(i+(j-1)*ny),y(i+(j-
1)*ny));
prd(j)<=x(i+(j-1)*ny);
end generate;
--------------------------------
c4:if j>1 and j<ny and i<nx and i>1 generate
        P4:cell port map(A(i),B(j),x(i+1+(j-2)*ny),y(i+(j-2)*ny),x(i+(j-1)*ny),y(i+(j-
1)*ny));
end generate;
--------------------------------
c5a:if j>1 and j<ny and i=nx and j-2*(j/2)=1 generate
        P5:cell port map(A(i),B(j),Dia_odd,y(i+(j-2)*ny),x(i+(j-1)*ny),y(i+(j-1)*ny));
end generate;
c5b:if j>1 and j<ny and i=nx and j-2*(j/2)=0 generate
        P5:cell port map(A(i),B(j),Dia_even,y(i+(j-2)*ny),x(i+(j-1)*ny),y(i+(j-1)*ny));
end generate;
```

```
--------------------------------

c6:if j=ny and i=1 generate

        P6:cell port map(A(i),B(j),x(i+1+(j-2)*ny),y(i+(j-2)*ny),x(i+(j-1)*ny),y(i+(j-
1)*ny));

        prd(i+j-1)<=x(i+(j-1)*ny);

end generate;

--------------------------------

c7:if j=ny and i=2 and nx>2 generate

        P7:cell port map(A(i),B(j),x(i+1+(j-2)*ny),y(i+(j-2)*ny),x(i+(j-1)*ny),y(i+(j-
1)*ny));

        FA1:fulladder port map(x(i+(j-1)*ny),y(i+(j-1)*ny-1),FA_Cin,c0(i-1),prd(i+j-1));

end generate;

--------------------------------

c8:if j=ny and i>2 and i<nx generate

        P8:cell port map(A(i),B(j),x(i+1+(j-2)*ny),y(i+(j-2)*ny),x(i+(j-1)*ny),y(i+(j-
1)*ny));

        FA2:fulladder port map(x(i+(j-1)*ny),y(i+(j-1)*ny-1),c0(i-2),c0(i-1),prd(i+j-1));

end generate;

--------------------------------

c9a:if j=ny and i=nx and j-2*(j/2)=1 generate

        P9:cell port map(A(i),B(j),Dia_odd,y(i+(j-2)*ny),x(i+(j-1)*ny),y(i+(j-1)*ny));

        FA3:fulladder port map(x(i+(j-1)*ny),y(i+(j-1)*ny-1),c0(i-2),c0(i-1),prd(i+j-1));

end generate;

c9b:if j=ny and i=nx and j-2*(j/2)=0 generate
```

```
        P9:cell port map(A(i),B(j),Dia_even,y(i+(j-2)*ny),x(i+(j-1)*ny),y(i+(j-1)*ny));
        FA3:fulladder port map(x(i+(j-1)*ny),y(i+(j-1)*ny-1),c0(i-2),c0(i-1),prd(i+j-1));
end generate;
--------------------------------
   end generate;
   end generate;
--------------------------------
FA4:fulladder port map(Fa_x,y(nx*ny),c0(nx-1),prd(nx+ny+1),prd(nx+ny));
--------------------------------
PAR:
for i in nx+ny+1 downto 1 generate
cxor(i)<=prd(i);
end generate;
sgn<=Asgnm xor Bsgnm;
process(cxor,sgn)
variable sig,sig1,sig2:std_logic;
begin
for i in 1 to nx+ny loop
onecom(i)<=sgn xor cxor(i);
if i=1 then
sig:=sgn;
else
sig:=sig1;
end if;
```

```
Sumn(i)<=onecom(i) xor sig;
sig1:=(onecom(i) and sig);
end loop;
---test1<= not(prd xor Sumn);
if sig1='0'then
sig2:=high;
high1<=sig2;
Test<=high1;
else
Test<=sig1;
end if;
end process;
end MCSA;
---------------------------------
configuration MCSA_2scom of MCSA_Mult is
for MCSA
end for;
end MCSA_2scom;
```

5.11 Simulation result For multiplication of unsigned numbers

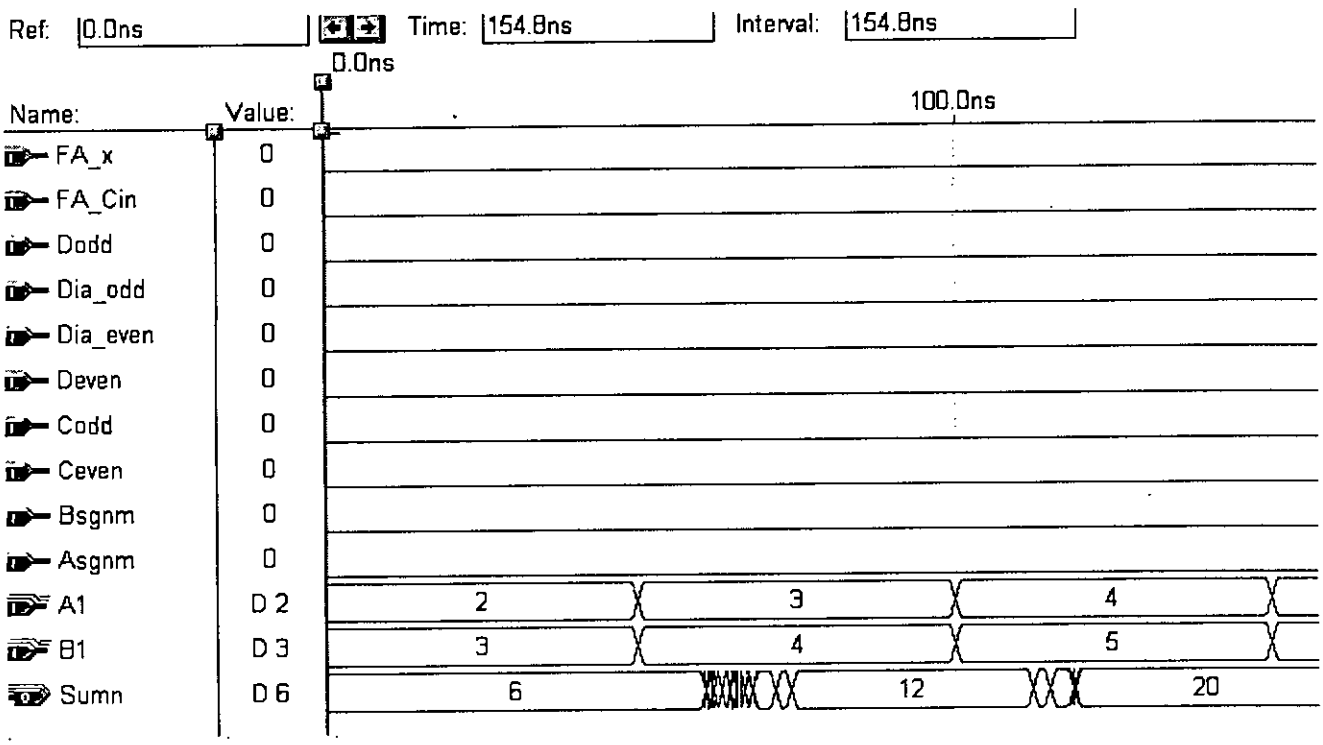


Fig: 5.3 when n = 3 (A and B are decimal number)

In the above figure simulation result for multiplication of two positive decimal numbers(A1,B1 with bit size of n=3) is shown. results is shown in the sumn row. One number A1 starting from 2 and incremented by 1 after every 50 ns . Another number B1 starting from 3 incremented by '1' after every 50 ns. In the first block A1=2 and B1=3 and the result is 6.

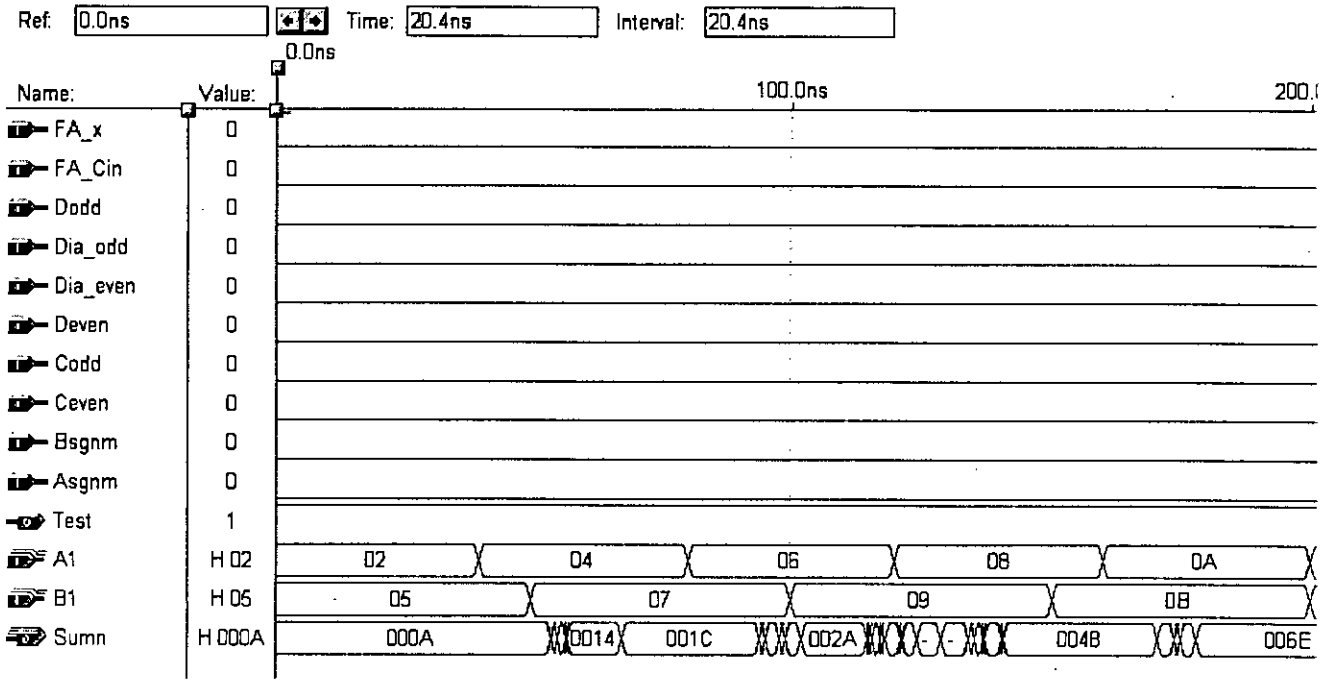


Fig: 5.4 when n=8(A and B are hexadecimal numbers)

In the Fig.5.4 multiplication of two positive hexadecimal numbers is shown with the operand size is n=8

5.12 For multiplication of signed numbers

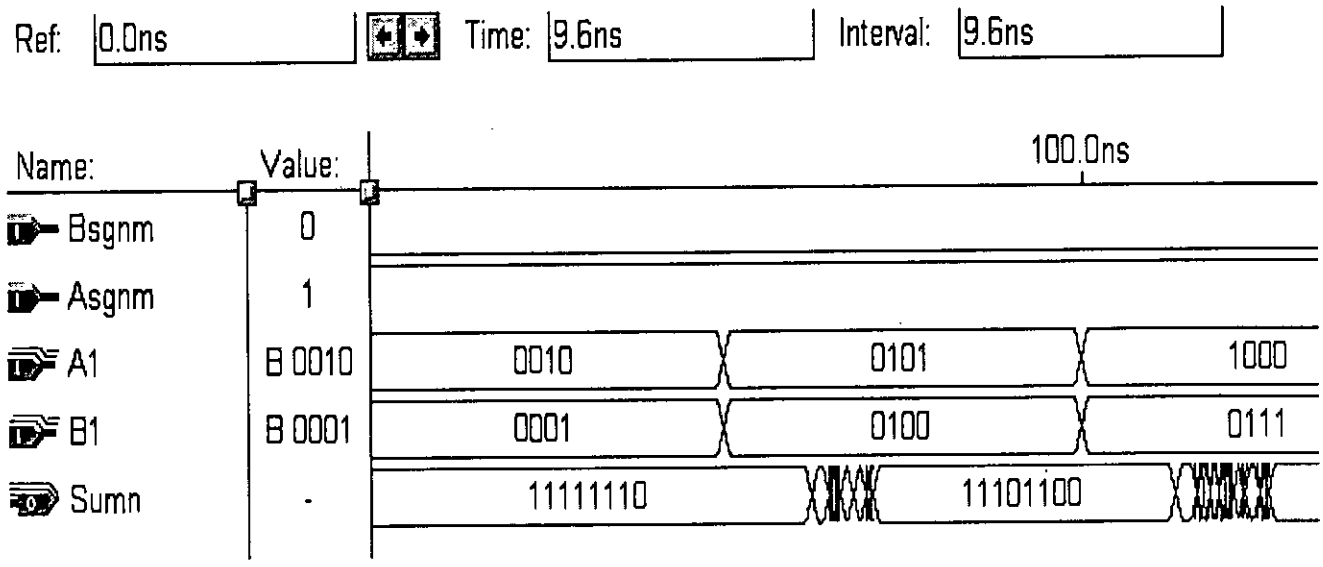


Fig.5.5 Simulation results for n=4 when A is negative number and B is positive

In the Fig.5.5 simulation result for multiplication of signed number is shown. Here A is negative number and B is positive binary number. As the number A is negative so the signal Asgnm is '1'. thats makes the input A to the multiplier block dashed block shown in the Fig.3.1 is 2's complement of the original input. That why we get the result in 2's complement form.

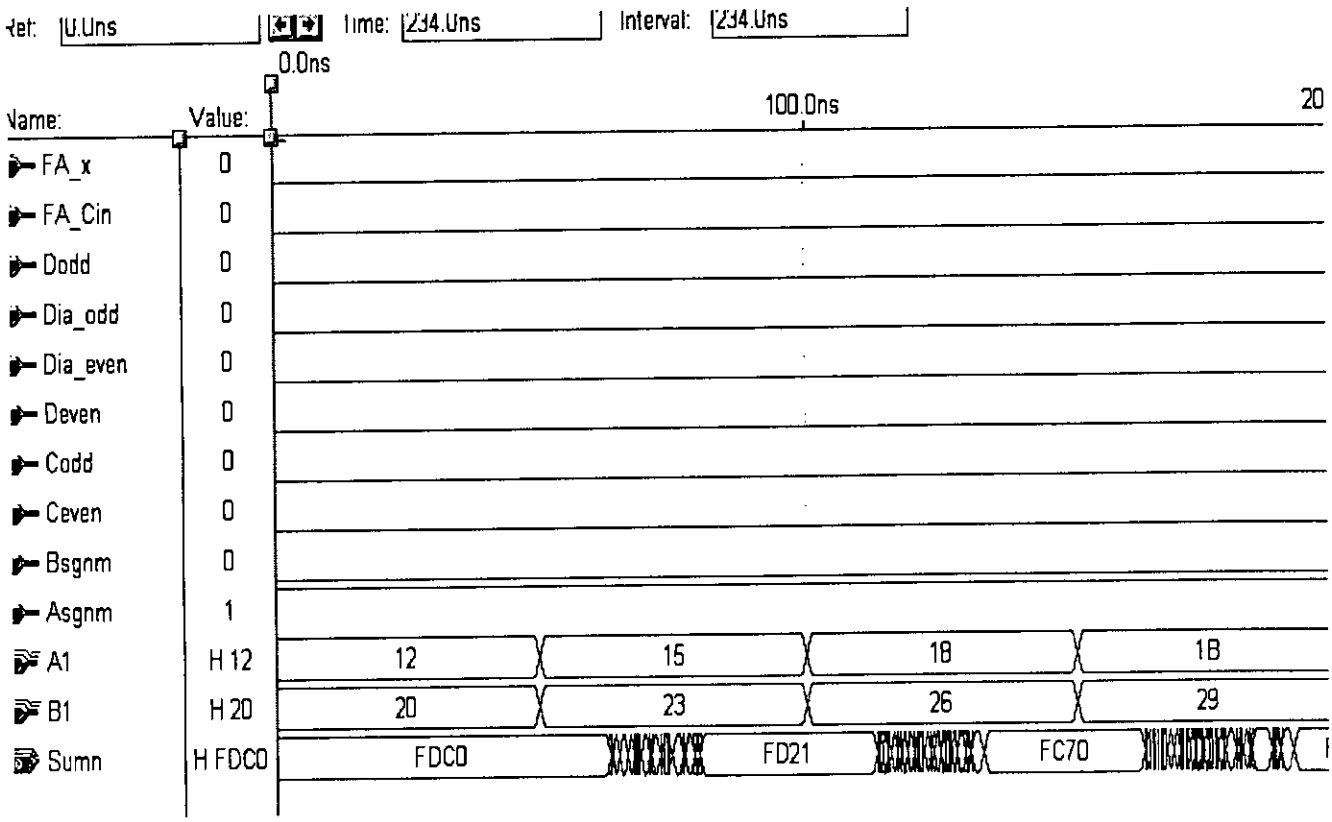


Fig. 5.6 when n = 8 (A is negative and B is positive hexadecimal numbers)

Here multiplication results for one signed hexadecimal and one unsigned hexadecimal numbers is shown.

5.13 For testing the multiplier

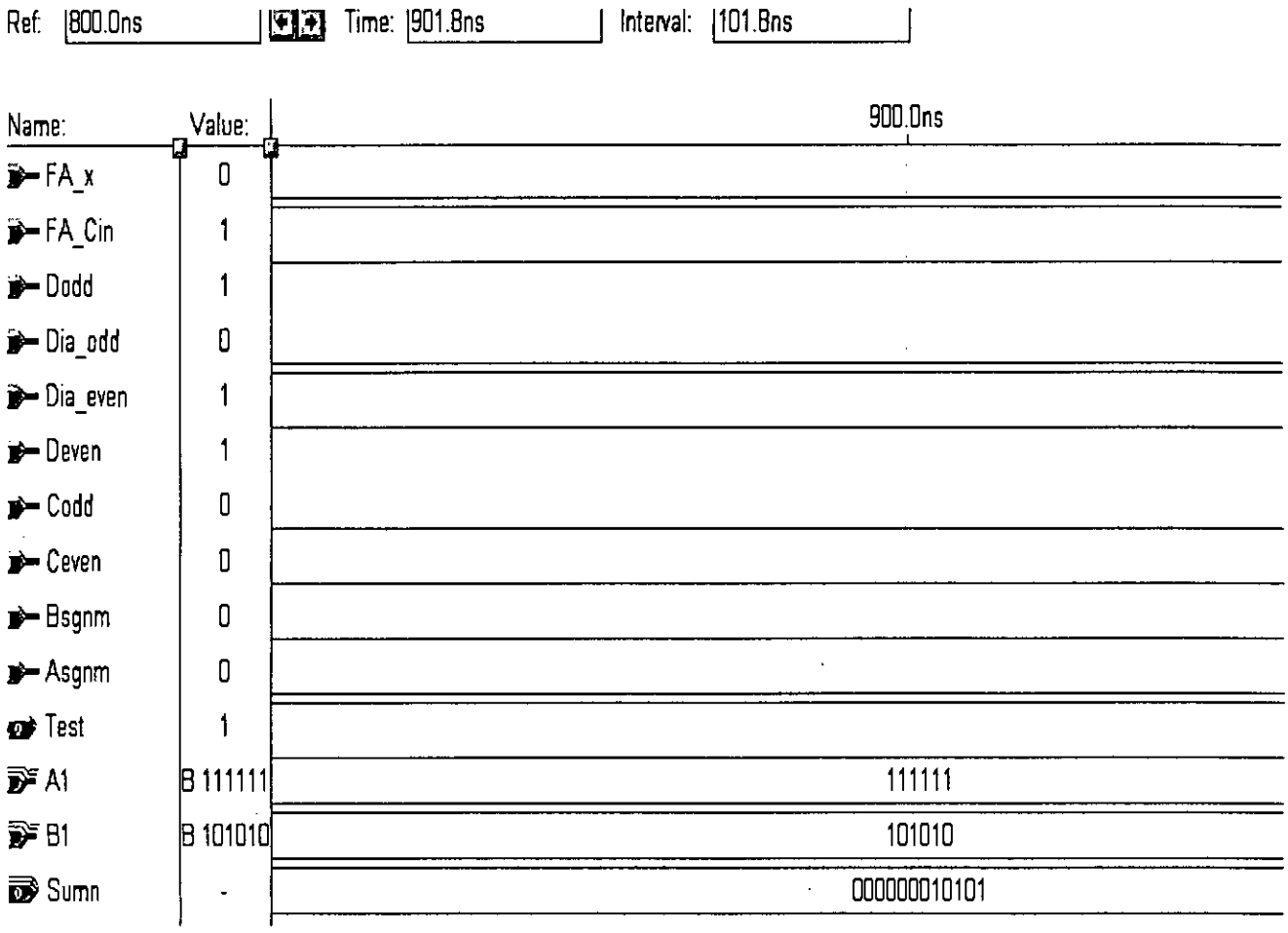


Fig.5.7 When n = 6 and test pattern is $T_{9,14}$

In the Fig.5.7 output patterns for test pattern $T_{9,14}$ is shown.

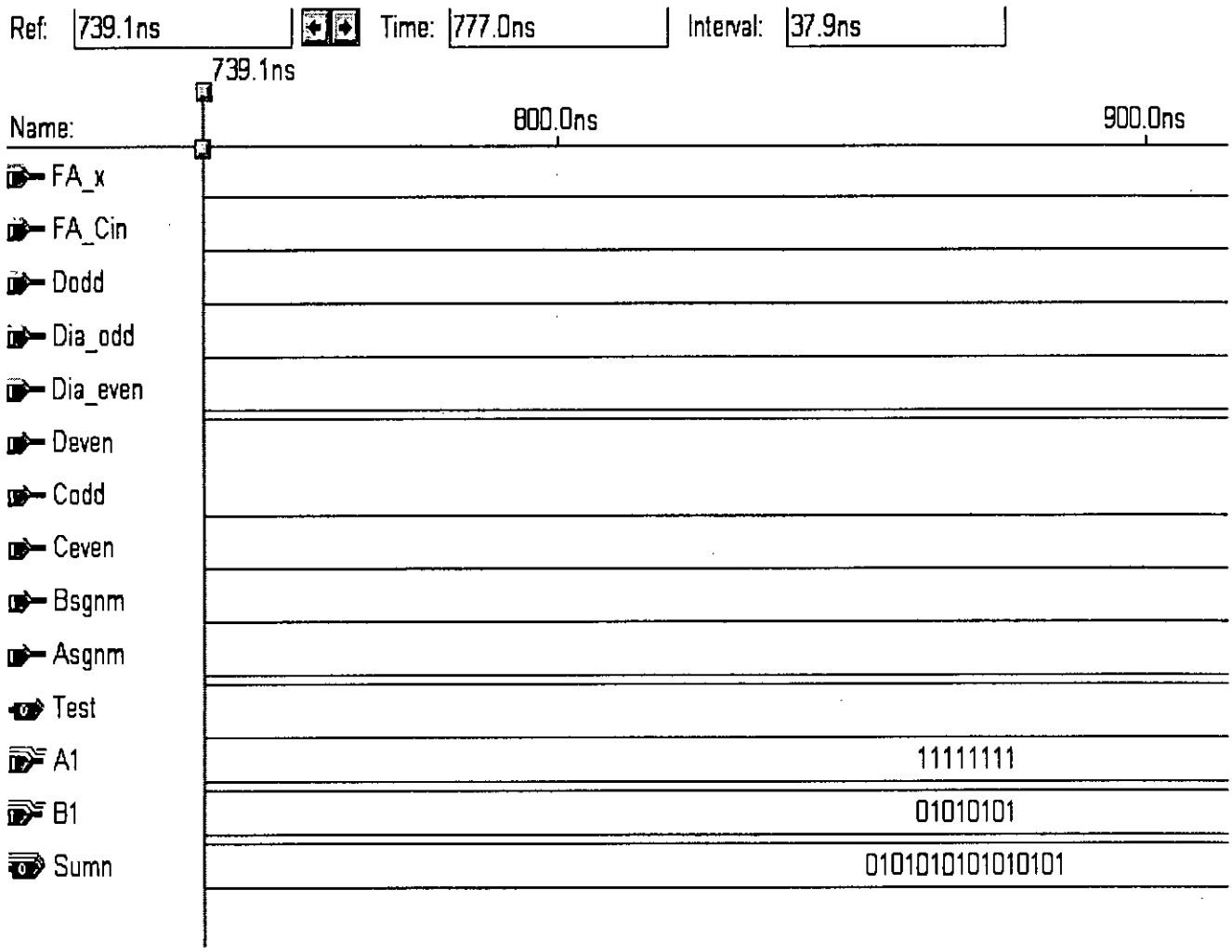


Fig. 5.8 When n = 8 and test pattern is $T_{11,8,13,12}$

In the Fig.5.8 output patterns for test pattern $T_{11,8,13,12}$ is shown.

Simulation result

6.1 Device information after simulation

Project Information

c:\max2work\vhdl\mcsa_mult.rpt

MAX+plus II Compiler Report File

Version 9.3 7/23/1999

Compiled: 12/25/2004 23:41:51

Any megafunction design, and related net list (encrypted or decrypted),support information, device programming or simulation file, and any otherassociated documentation or information provided by Altera or a partnerunder Altera's Megafunction Partnership Program may be used only to program PLD devices (but not masked PLD devices) from Altera. Any otheruse of such megafunction design, net list, support information, device programming or simulation file, or any other related documentation orinformation is prohibited for any other purpose, including, but notlimited to modification, reverse engineering, de-compiling, or use withany other silicon devices, unless such use is explicitly licensed undera separate agreement with Altera or a megafunction partner. Title tothe intellectual property, including patents, copyrights, trademarks, trade secrets, or maskworks, embodied in any such megafunction design, net list, support information, device programming or simulation file, or any other related documentation or information provided by Altera or a megafunction partner, remains with Altera, the megafunction partner, or their respective licensors. No other

***** Project compilation was successful

MCSA_MULT

** DEVICE SUMMARY **

Chip/ POF	Device	Input Pins	Output Pins	Bidir Pins	LCs	% Utilized
mcsa_mult						
8282ALC84-2		26	17	0	172	82 %
User Pins:		26	17	0		

Project Information

c\max2work\vhdl\mcsa_mult.rpt

** FILE HIERARCHY **

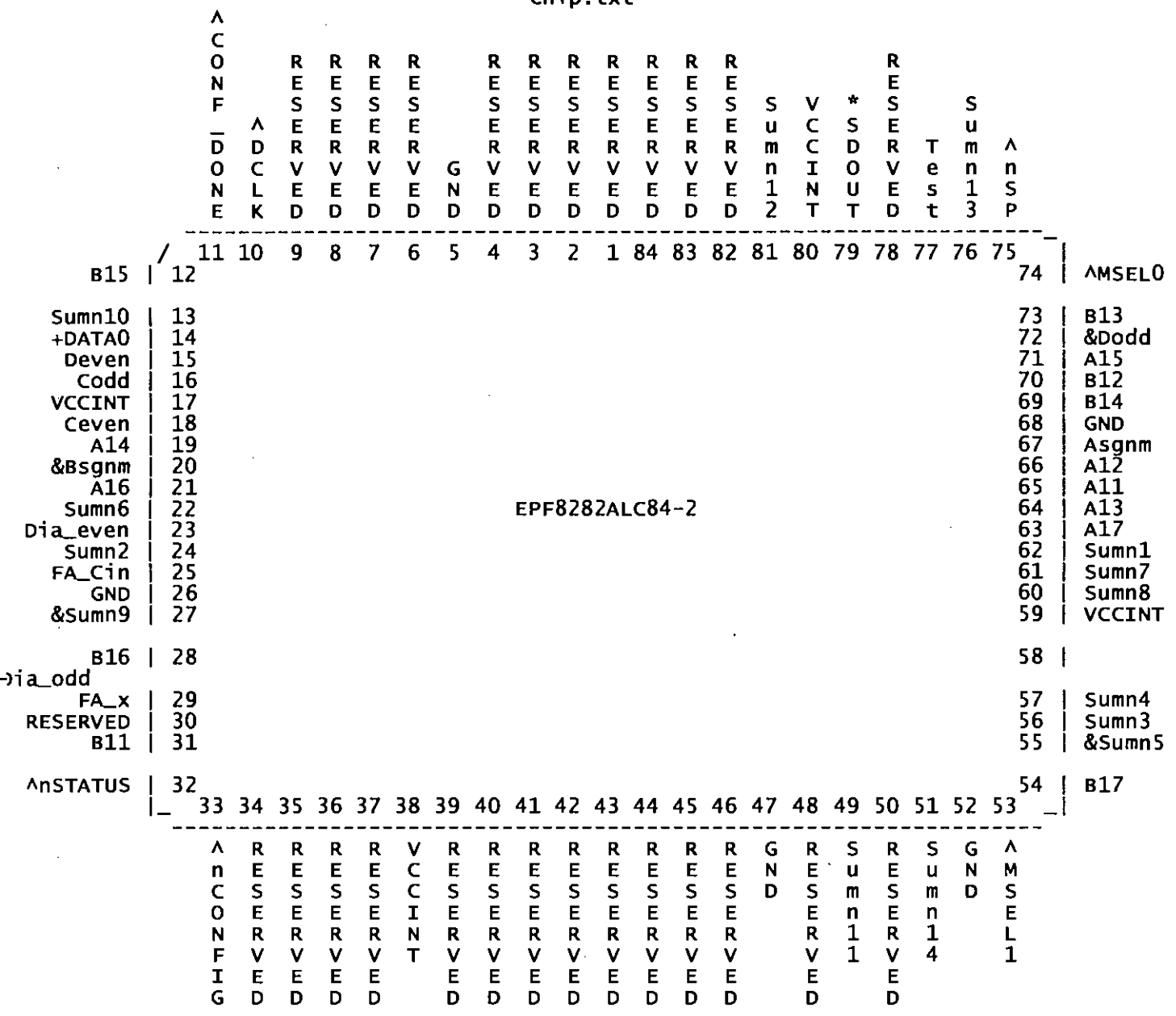
Device-Specific Information:

c:\max2work\vhdl\mcsa_mult.rpt

mcsa_mult

***** Logic for device 'mcsa_mult' compiled without errors.

Device: EPF8282ALC84-2


^CONF_DONE
^DCLK
RESERVED
GND
Sumn12
VCCINT
*SDOUT
Test
Sumn13
^nSTATUS
B15
Sumn10
+DATA0
Deven
Codd
VCCINT
Ceven
A14
&Bsgnm
A16
Sumn6
Dia_even
Sumn2
FA_Cin
GND
&Sumn9
B16
Dia_odd
FA_x
RESERVED
B11
^nSTATUS
EPF8282ALC84-2
^MSEL0
B13
&Dodd
A15
B12
B14
GND
Asgnm
A12
A11
A13
A17
Sumn1
Sumn7
Sumn8
VCCINT
B17
Sumn4
Sumn3
&Sumn5
^nCONFIG
RESERVED
VCCINT
GND
^MSEL1
Sumn14
Sumn11

N.C. = No Connect. This pin has no internal connection to the device.

VCCINT = Dedicated power pin, which MUST be connected to VCC (5.0 volts).

VCCIO = Dedicated power pin, which MUST be connected to VCC (5.0 volts).

GND = Dedicated ground pin or unused dedicated input, which MUST be connected to GND.

RESERVED = Unused I/O pin, which MUST be left unconnected.

Device-Specific Information:

c:\max2work\vhdl\mcsa_mult.rpt

mcsa_mult

Total dedicated input pins used:	4/4	(100%)
Total I/O pins used:	41/64	(64%)
Total logic cells used:	172/208	(82%)
Average fan-in:	3.83/4	(95%)
Total fan-in:	660/832	(79%)

Total input pins required:	26
Total input I/O cell registers required:	0
Total output pins required:	17
Total output I/O cell registers required:	0
Total buried I/O cell registers required:	0
Total bidirectional pins required:	0
Total reserved pins required	2
Total logic cells required:	172
Total flipflops required:	0
Total logic cells in carry chains:	0
Total number of carry chains:	0
Total logic cells in cascade chains:	0
Total number of cascade chains:	0

Device-Specific Information:
c:\max2work\vhdl\mcsa_mult.rpt
mcsa_mult

Total dedicated input pins used:	4/4	(100%)
Total I/O pins used:	37/64	(57%)
Total logic cells used:	137/208	(65%)
Average fan-in:	3.81/4	(95%)
Total fan-in:	523/832	(62%)

Total input pins required: 24
Total input I/O cell registers required: 0
Total output pins required: 15
Total output I/O cell registers required: 0
Total buried I/O cell registers required: 0
Total bidirectional pins required: 0
Total reserved pins required 2
Total logic cells required: 137
Total flipflops required: 0
Total logic cells in carry chains: 0
Total number of carry chains: 0
Total logic cells in cascade chains: 0
Total number of cascade chains: 0

Synthesized logic cells: 8/ 208 (3%)

** INPUTS **

					Fan-In	Fan-Out			
Pin	LC	Row	Col	Primitive	Code	INP	FBK	OUT	FBK Name
67	-	A	--	INPUT	0	0	0	1	Asgnm
65	-	A	--	INPUT	0	0	0	14	A11
66	-	A	--	INPUT	0	0	0	14	A12
64	-	A	--	INPUT	0	0	0	14	A13
19	-	A	--	INPUT	0	0	0	14	A14
71	-	A	--	INPUT	0	0	0	14	A15
21	-	A	--	INPUT	0	0	0	14	A16
63	-	B	--	INPUT	0	0	0	14	A17
20	-	A	--	INPUT	0	0	0	1	Bsgnm
31	-	-	--	INPUT	0	0	0	14	B11
70	-	A	--	INPUT	0	0	0	14	B12
73	-	-	--	INPUT	0	0	0	14	B13
69	-	A	--	INPUT	0	0	0	14	B14
12	-	-	--	INPUT	0	0	0	14	B15
28	-	B	--	INPUT	0	0	0	14	B16
54	-	-	--	INPUT	0	0	6		Ceven
16	-	A	--	INPUT	0	0	0	8	Codd
15	-	A	--	INPUT	0	0	0	6	Deven
23	-	B	--	INPUT	0	0	0	6	Dia_even
58	-	B	--	INPUT	0	0	0	6	Dia_odd
72	-	A	--	INPUT	0	0	0	8	Dodd
25	-	B	--	INPUT	0	0	0	2	FA_Cin
29	-	B	--	INPUT	0	0	0	1	FA_x

Code:

s = Synthesized pin or logic cell
+ = Synchronous flipflop
/ = Slow slew-rate output

! = NOT gate push-back
r = Fitter-inserted logic cell

Device-Specific Information:
c:\max2work\vhdl\mcsa_mult.rpt
mcsa_mult

** OUTPUTS **

Pin	Fed By LC	Row	Col	Primitive	Fan-In Code	Fan-Out INP	FBK	OUT	FBK	Name
84	- -	09		OUTPUT	0	1	0	0		Sumn1
81	- -	11		OUTPUT	0	1	0	0		Sumn2
49	- -	11		OUTPUT	0	1	0	0		Sumn3
4	- -	05		OUTPUT	0	1	0	0		Sumn4
41	- -	05		OUTPUT	0	1	0	0		Sumn5
48	- -	10		OUTPUT	0	1	0	0		Sumn6
9	- -	01		OUTPUT	0	1	0	0		Sumn7
34	- -	01		OUTPUT	0	1	0	0		Sumn8
55	- B	--		OUTPUT	0	1	0	0		Sumn9
60	- B	--		OUTPUT	0	1	0	0		Sumn10
56	- B	--		OUTPUT	0	1	0	0		Sumn11
30	- B	--		OUTPUT	0	1	0	0		Sumn12
61	- B	--		OUTPUT	0	1	0	0		Sumn13
25	- B	--		OUTPUT	0	1	0	0		Sumn14
57	- B	--		OUTPUT	0	1	0	0		Sumn15
28	- B	--		OUTPUT	0	1	0	0		Sumn16
29	- B	--		OUTPUT	0	1	0	0		Test

Code:

s = Synthesized pin or logic cell

+ = Synchronous flipflop

/ = Slow slew-rate output

! = NOT gate push-back

r = Fitter-inserted logic cell

Compilation Times

Compiler Netlist Extractor	00:00:01
Database Builder	00:00:00
Logic Synthesizer	00:00:01
Partitioner	00:00:01
Fitter	00:00:01
Timing SNF Extractor	00:00:00
Assembler	00:00:00
---	---
Total Time	00:00:04

Memory Allocated

Peak memory allocated during compilation = 9,838K

6.2 Conclusion

Here architecture for the multiplication of signed and unsigned numbers is proposed and VHDL code for the proposed architecture is written. Our design has the capability in designing parameterizable architecture for any target process. This program can be used to implement various sizes of multipliers on FPGA without any modification. For the multiplier testing the total number of test pins are fixed i .e don't vary with the operand size. For signed operation the output is obtained in two's complement form. In future this architecture can be implemented with Booth algorithm for better performance.

References

[1] E.Koutroulis, K.Kalaitzakis, and N.C. Voulgaris, "Development of a Microcontroller-Based, Photovoltaic Maximum power point Tracking Control System", IEEE transactions on Power Electronics, Vol-16, Jan 2001, pp. 46-54

[2] A.A.Hiasat, "New Efficient Structure for a Modular Multiplier for RNS", IEEE transactions on Computers, Vol-49, Feb 2000, pp.170-174.

[3] A. V. Opperheir and R. W. schafer, "Discrete signal processing", Prentice-Hall, 1994.

[4] V. Belaguli and A.K.S. Bhat, "A Hybrid resonant Converter Operated as a Low Harmonic Rectifier with and without Active Control", IEEE transactions on Power Electronics, Vol-14, July 1999, pp. 730-742.

[5] J. P. Shen and F. Joel Ferguson, "The design of easily testable VLSI array multipliers", IEEE transactions on computers, Vol. C-33, June 1984, pp. 554-560.

[6] Jien-Chunglo, "Online current testing", IEEE Design and Test of computers, October-December 1998, pp. 49-56.

[7] T. Williams and K Parker, "Design for testability- a survey", IEEE transactions on computers, Vol. C-31, Jan.1982, pp.2-15.

[8] K. O. Boateng, H. Takahashi, and Y. Takamatsu " Design of a C- Testable multipliers Based on the modified Booth Algorithm", 5th Asian Test Symposium, 17-18 November,1997, Akita, Japan, pp.121-123