**ABSTRACT**

Digital design is an important and wide field. The applications of digital design are present in our daily life, including computers, calculators, video cameras etc. VHDL (VHSIC Hardware Description Language) has become an effective and important language for designers in the world of digital design.

In this project we design a 64 bit ALU with 32 select line operations that can be performed by it in VHDL using VHDL software XILINX. We will also check our designs behavior using MODELSIM PE 5.7 F and by designing its Testbench code in XILINX. We have designed our ALU to perform 32 operations which are arithmetic, logical, comparative, and shifting and many mixed types of them. There are many ways of designing an ALU in VHDL. We have used Behavioral type of modeling in our design. We know that ALU is the core for the microprocessor since it handles its ability to perform any type of mathematical or shifting operation. Here we will take two 64 bit inputs, one 5 bit select line (i.e. 32 operations), a carry output and a 64 bit output line.

We analyze our 64 Bit ALU with 32 operations here using MODELSIM in which we assign clock inputs to our input lines and check for the desired output in the output and carry lines. We will also design a test bench for the assigned code and check its response.

**CHAPTER 1**

**INTRODUCTION**

**1.1 MOTIVATION**

The optimization for lower power dissipation and faster device performance is a matter for great concern in modern day electronics. The characteristics of an ideal design are minimum power consumption, it requires minimum area but has the high speedresponse or we can say output. But the mutually opposing nature of these parameters calls for the formulation of a suitable solution to perform a tradeoff between these parameters. Power optimization may arise at any stage of the digital design flow. However, there are maximized benefits in algorithmic and architectural design level.

In the modern day electrical and computer engineering products digital hardware plays a very important role which is the result of the fast growth in transistor densities and speed of integrated circuits. There has also been lowering of prices in IC's which is the result of extensive advances in micro-electronic implementation technologies which will definitely be further implemented on a far bigger scale in the upcoming future. The "computer revolution" has affected every aspect of society and many problems viewed as being intractable can now be solved. Now days the digital design of new technologies is carried out by groups of engineers consisting of different individuals with different mind sets and different expertise in one or more than one field. Present industry practice has an increased requirement for systems designers with good knowledge and experience in using programmable logic like CPLDs and FPGAs in addition to hardware description languages. VHDL is one of the most popular design applications used by designers in modern digital design.

Modern day microprocessors are designed to operate at maximum speed but consume minimum power at the same time. This is necessary to improve a lot of factors like improved battery life of portable systems, improved reliability and reduced heat removal costs. The ALU, being one of the most computationally intensive modules in a CPU almost always falls in the data path during the execution of an instruction. Hence the power consumption of the ALU should be kept at a minimum.

The Arithmetic Logic Unit (ALU) is a fundamental building block of the Central Processing Unit (CPU) of a Computer. Even the simplest microprocessors do contain at least one ALU for purposes such as maintaining timers and many more. We can also say that ALU is a core component of all central processing unit within in a computer and is an integral part of the execution unit. ALU is capable of calculating the results of an extensive variety of basic arithmetical and logical operations. The ALU takes an input as the data to be operated on (called operands) and a code from the control unit indicating which operation is to perform. The output is the result of the computation. The ALU implemented will perform the following operations: Arithmetic operations (addition, subtraction increment, decrement, transfer) Logic operations (AND, NOT, OR, NAND, NOR, EX-OR, EX-NOR) or combination of these operations like conversion into different forms etc. The output of the circuit is calculated from ALU.

VHDL is a hardware description language that is most widely to model a digital system. It contains elements that can be used to describe the behavior or structure of the digital system, with the provision for specifying its timing explicitly. The language provides support for modeling the system hierarchically and also supports top-down and bottom-up design methodologies. The system and its subsystems can be described at any level of abstraction ranging from the architecture level to gate level. VHDL is a large and verbose language with many complex constructs that have complex semantic meanings and is difficult to understand initially (VHDL is often quoted to be an acronym for Very Hard Description Language). However, it is possible to quickly understand a subset of VHDL which is both simple and easy to use. The emphasis of this text is on presenting this set of simple and commonly used features of the language so that the reader can start writing models in VHDL. These features are powerful enough to be able to model designs of large degrees of complexity.

Precise simulation semantics are associated with all the language constructs, and therefore, models written in this language can be verified using a VHDL simulator. An arithmetic logic unit (ALU) is a digital circuit that performs arithmetic and logical operations. The ALU is a fundamental building block of the central processing unit (CPU) of a computer, and even the simplest microprocessors contain one for purposes such as maintaining timers. The processors used in modern day CPUs and graphics processing units (GPUs) are using very powerful and very complex ALUs i.e. a single component may contain a number of ALUs.

The Behavioral Style architecture contains simultaneous statements with sections of sequential statements that describe the outputs of the circuit at a discrete moment in time given particular inputs. While similar language constructs are often found in Dataflow and Behavioral style architectures, only the latter explicitly exhibit the notions of time and control. This style describes the functions of the circuit at the algorithmic level. The highest level of abstraction is the behavioral level that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them.

Alu’s perform most of the processors operations. In an ALU data is loaded from input registers, and an external Control Unit then points the ALU towards the operation to be performed on that data, and then the ALU stores its result into an output register. The inputs usedin an ALU operation are the data to be operated on also known as operands and a code from the control unit indicating which operation to perform. Its output is the result of the computation.

Designing the ALU in VHDL has become more complex in order to overcome their requirements in terms of number of operations and fast operation and so it has been more efficient over the years and has been a promising area for further improvements.The number of operations that can be performed by an ALU has been consistently increasing. ALU is a core component of all Processor and is an Integral part of the execution unit. The ALU performs the decision making operations (logical) and arithmetic operations. Arithmetic operations consist of several functions such as addition, subtraction etc. There are a number of different techniques to design these functions in VHDL. It is the most complex component with regard to design, amongst all the components of the computer, and it also contributes to most of the delay in the processor.Thus, the design of the ALU is quite critical for the speed of the computer operations. The ALU can also efficiently perform parity check to utilize it in Digital Systems. A faster adder can perform fast addition operation. The ALU is used in Network Interface Card (NIC) to maximize the throughput.

* 1. **OBJECTIVE**

The main objective of our project is to design a 64 bit Arithmetic Logic Unit which is a digital circuit that performs arithmetic and logical operations using VHDL. This project deals with the design and study the 64 bit ALU which can perform 32 arithmetic and logical operations using VHDL (VHSIC hardware description language) i.e. XILINX V6.0 and then verify our design using simulation software i.e. MODELSIM V13.0 and by writing the programs test bench.

**1.2 PROJECT ORGANIZATION**

* The literature survey has been discussed in **chapter 2**.
* The basic concept of Fundamentals of ALU, uses of ALU, history and further scopes in designing and implementation of ALU are given in **chapter 3**.
* **Chapter 4** describes the design and software implementation of the ALU (Arithmetic and Logical Unit). The 64 bit ALU with 32 arithmetic and logical operations is then coded along with its test bench in VHDL (VHSIC hardware description language) language using XILINX V6.0. We will also see an RTL logic circuit of our designed ALU.
* In **chapter 5** simulated our ALU and VHDL code and Test bench code using the simulation software MODELSIM V13.0 using 64 bit values.
* The verification simulation results obtained from the designed ALU and the Test Bench will also be discussed in **chapter 5**. A conclusion has been made by these results and future scope of the thesis work has been discussed.
* REFERENCES
* APPENDIX contains the VHDL and TESTBENCH code.
  1. **TOOLS USED**

**1.3.1. XILINX**

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.

Xilinx Inc. is the world's largest supplier of programmable logic devices, the inventor of FIELD PROGRAMMABLE LOGIC ARRAY (FPGA) and the first semi-conductor company with a fabulous manufacturing model. Founded in Silicon Valley in 1984 and head quartered in San Jose, California, USA, the company has corporate offices throughout North America and Europe. The programmable logic device market has been led by Xilinx since the late 1990s. Over the years, Xilinx has fueled an aggressive expansion to India, Asia and Europe- regions.

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 ISIM or the Modelism logic simulator is used for system-level testing.

**USERINTERFACE**

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.[3] 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.

**1.3.2 MODELSIM**

System-level testing may be performed with ISIM or the ModelSim logic simulator, and such test programs must also be written in HDL languages. Test bench programs may include simulated input signal wavef0orms, or monitors which observe and verify the outputs of the device under test.

ModelSim is a multi-language HDL simulation environment by Mentor Graphics, for simulation of hardware description languages such as VHDL, Verilog and SystemC. ModelSim has a built-in C debugger included in it. ModelSim can be used independently, or in conjunction with Intel Quartus Prime, Xilinx ISE or Xilinx Vivado. Simulation is performed using the graphical user interface (GUI), or automatically using scripts.

ModelSim can also be used with MATLAB/Simulink, using Link for ModelSim. Link for ModelSim is a fast bidirectional co-simulation interface between Simulink and ModelSim. For such designs, MATLAB provides a numerical simulation toolset, while ModelSim provides tools to verify the hardware implementation & timing characteristics of the design.

**LANGUAGE SUPPORT**

ModelSim uses a unified kernel for simulation of all supported languages, and the method of debugging embedded C code is the same as VHDL or Verilog. ModelSim products enable simulation, verification and debugging for the following languages:

* VHDL
* Verilog
* Verilog 2001
* System Verilog
* PSL
* SystemC

**CHAPTER 2**

**LITERATURE SURVEY**

Mohamed EL KHAILI el.al (1) is presented in this paper presents design concept of 4-bit Processing Unit (PU) based on 4-bit Arithmetic and Logic Unit (ALU) for multiprocessor architecture on one FPGA chip. Design methodology has been changing from schematic design to HDL based design. We use a VHDL structural and dataflow level design. Each module of the Processing Unit is divided into smaller modules. All the modules in logical unit and Arithmetic and Logic Unit (ALU) design are realized using VHDL design. Functionalities are validated through synthesis and simulation process. Processing Unit (PU) using VHDL fulfills the needs for different high performance applications such as processor for mono architecture and parallel architecture.

Ankit Chouhan et.al (2) is presented in this paper explains the design and implementation of 16-bit ALU (arithmetic and logic unit) using VHDL by using mixed style of modeling in Xilinx ISE 8.1i. The ALU takes two16-bits numbers and performs different principal arithmetic and logic operations like addition, multiplication, logical AND, OR, XOR, XNOR. The major focus of concern in this ALU is the multiplication operation using radix-4 booth algorithm and bit-pair recording technique which increases the speed of multiplication operation. We had followed modular programming approach so that our ALU is sub divided into smaller logical block. All the modules in arithmetic and logical unit design are realized using VHDL design. The top level design consists of arithmetic unit and logic unit which is implemented by using mixed type of modeling. Designing of this ALU is done by using VHDL and simulated using Xilinx ISE 8.1i.

Disha Malik et.al (3) is presented in this paper involves the construction of 32‐bit ALU (Arithmetic Logical Unit) using VHDL using Xilinx Synthesis tool ISE 9.2i and implementation them on FPGA (Field Programmable Gate Array) using Spartan 3E. The ALU is a fundamental building block of the central processing unit (CPU) of a computer, and even the simplest microprocessors contain one for purposes such as maintaining timers. The processors found inside modern CPUs and graphics processing units (GPUs) accommodate very powerful and very complex ALUs; a single component may contain a number ofALUs. The ALU performs mathematical, logical, and decision operations in a computer and is the final processing performed by the processor.

Archana Singh Yadav and Pratyush Tripathi et.al (4) is presented in this paper a VHDL structural design for Arithmetic and Logical Unit is proposed. This design of 16 bit ALU consists of two input pins for 16-bit operands along with one input pin for 5 selection lines at the input and at the output a pin for the 32-bit output and a pin for one bit carry or borrow. It may perform all the logical (bitwise) and arithmetic operations such as addition, multiplication, subtraction, division, shift, increment, comparison etc. In this design for 16- bit addition and subtraction, the component used is 16-bit Parallel Adder. This 16-bit parallel design includes a 1-bit full adder. In this design multiplier is made of an algorithm called add and shift algorithm. This design may be made more compact using the statement “generic” in VHDL which also makes the design more flexible. These design units are yet independent but using structural modeling these may be simulated through a single design. In this unit the bitwise operations are also added for different selection lines for the logical operations. In this design the multiplexers may be used to select the appropriate inputs for the arithmetic and logic design units. These multiplexers may be used to perform some simple logical operations like programmable shifter or to invert the operands. In accordance with the selection line we may find out about the operations to be performed on the given input. This VHDL design proves to be very flexible and efficient when it is required to add some new complex operations in it.

Abdul Kareem et.al (5) ispresented in this project will deal with the design of a 16 bit reversible Arithmetic Logic Unit (ALU) with 15 operations is presented by making use of Double Peres gate, Fredkin gate, Toffolli gate, DKG gate and NOT gate. A new VLSI architecture for ALU using reversible logic gates is proposed. ALU is one of the most importantcomponents of CPU that can be part of a programmable reversible computing device such as a quantum computer. A first single bit reversible ALU and second single bit ALU are designed and Then 16 single bit ALU’s arecascaded together taking carry out of ALU performing LSB operation as an input to carry in of ALU performing next LSB operation. Design is implemented and verified in Verilog in modelSim Altera 6.6d.

Geetanjali et.al (6) is presented in this paper presents implementation of a 32-bit Arithmetic Logic Unit (ALU) using VHDL. Here the behavioral VHDL model of ALU is designed to perform 16 operations which includes both logical and arithmetic operations. The VHDL implementation and functionality test of the 32-bit ALU is done by using the Modelsim 5.4a tool.

Mukesh P. Mahajan et.al (7) is presented in this paper ALU is the fundamental unit of a microprocessor which performs all the basic operations based on the control input selection. There are separate units which work independent of the main ALU for performing secondary operations such as address computation. The ALU performs arithmetic functions such as addition, subtraction etc. and logic functions including, logic AND, logic OR, and logic XOR etc. These various functions of the ALU are implemented using a set of functional units each implementing a function. It may also be done using sharing of same hardware with use of certain additional units like multiplexers. In this line of thought, the proposed project deals with design and simulation of 64 bit ALU using VHDL with the help of Xilinx ISE software.

Ankit MItra et.al (8) is presented in this paper how the ALU is one of the most frequently accessed modules in a CPU and is utilized during most instruction executions. Hence the power consumption of the ALU is a major concern. In this paper a low power 16 bit ALU is designed using VHDL. Lower power consumption is achieved by using clock gating technique and the results are compared with conventional ALU design. A carry skip adder with variable block length is used for the arithmetic unit to achieve better performance. The design is then implemented in Xilinx Spartan 3E FPGA. The ALU achieves a maximum frequency of 65.19 MHz with a dynamic power dissipation of 1.98mW when operated at 15 MHz.

Rupali Jarwl et.al (9) is presented in this paper presents design concept of 4-bit arithmetic and logic unit (ALU). Design methodology has been changing from schematic design to HDL based design.She proposed arithmetic and logic unit using VHDL structural and dataflow level design. Each module of ALU is divided into smaller modules. All the modules in arithmetic and logical unit design are realized using VHDL design. Functionalities are validated through synthesis and simulation process. Besides verifying the outputs,the outputs' timing diagram and interfacing signal are also tracked to ensure that they adhere to the design specification.ALU using VHDL fulfils the needs for different high performance application.

Kaushik Chandra Deva Sharma et.al (10) in this paper presents Design and Synthesis of 32-BIT Arithmetic Logic Unit (ALU). The design has been implemented using VHDL Xilinx Synthesis tool ISE 9.1i and targeted for Spartan device. ALU is designed to perform Arithmetic operations such as addition, subtraction, overflow; logical operations such as AND, OR, XOR, XNOR and NOT operations, Parity check, 1’s and 2’s complement operations, compare, etc. The ALU is a fundamental building block of the Central Processing Unit (CPU) of a computer, and even the simplest microprocessors contain one for purposes such as maintaining timers. The processors found inside modern CPUs and Graphics Processing Units (GPUs) accommodate very powerful and very complex ALUs; a single component may contain a number of ALUs. Flags like Zero, Carry and Odd Parity show the status of each Flag for result of the ALU’s operation in each clock cycle. Zero Counter counts number of zeros in the result. The modern ALU must be capable to perform all the binary arithmetic and logical operations to meet the requirements of modern VLSI industry. So, the paper is a forward step to design the ALU and meets the demand of present FPGA based technology. The paper presents a number of new operations (Parity, Overflow, Zero, Zero counter etc.) that an ALU can perform than so far designed ALU in VHDL.

Nazia Khan et.al (11) is presented in this paper presents the construction of 32‐bit ALU (Arithmetic Logical Unit) using VHDL. The main intention to design 32 bit ALU to defeat the area and power of the design. Which is a digital circuit that executes Arithmetic Logic Unit. ALU is a fundamental building block of the central processing unit (CPU) of a computer, and even the simplest microprocessors contain one for purposes such as maintaining timers. The coding will be written in VHDL and verified in I-Sim. After the coding the synthesis of the code was performed using Xilinx-ISE. Synthesis tool ISE 14.7. The ALU executes the desired operation and generates the result consequently. This designed put away very less area and only 193 LUTs occupy out of 10944 LUTs.

Vikesh Ukande et.al (12) presented in that the main objective of project is to design and verify different operations of Arithmetic and Logical Unit (ALU). We have designed an 32 bit ALU which accepts two 32 bits numbers and the code corresponding to the operation which it has to perform from the user. The ALU performs the desired operation and generates the result accordingly. The different operations are arithmetical, the coding was written in VHDL and verified in I-Sim. The waveforms were obtained successfully. After the coding was done, the synthesis of the code was performed using Xilinx-ISE.

Rajeeb Chetia et.al (13) is presented in this paper presents the behavioral Design and synthesis of a 64 bit ALU. 64 bit ALU is basically a multiplexer that operates mainly 16 operations as per select line Bit-permutation. Flags are other important indicators used for specific purpose e.g. if Sign Flag is HIGH then the output of ALU must be a negative number. CLR can reset the output of ALU.

P. Bhanushree et.al (14) is presented in this paper VHDL implementation of 64-bit arithmetic logic unit (ALU) is presented. The design was implemented using VHDL Xilinx Synthesis tool ISE 9.1 and targeted for Spartan device. ALU was designed to perform arithmetic operation and logical operations such as addition, subtraction using 64-bit fast adder, logical operations such as AND, OR, XOR and NOT operations, 1’scomplement, rotate operations and compare. ALU consist of two input registers to hold the data during operation, one output register to hold the result of operation, 64-bit fast adder with 2’s complement circuit to perform subtraction and logic gates to perform logical operation. The maximum propagation delay is 13.588ns and power dissipation is 38mW.

Manish Kumar and Suresh Kumar Jha et.al (15) is presented in this paper that paper basically manages survey for the development of number juggling Logic Unit (ALU) utilizing Hardware Description Language (HDL) utilizing Xilinx Vivado 14.7 and actualizes them on Field Programmable Gate Arrays (FPGAs) to investigate the outline parameters. ALU of advanced PCs is a part of rationale plan with the goal of creating suitable calculations keeping in mind the end goal to accomplish a proficient use of the accessible equipment. Speed, power and use of ALU are the measures of the proficiency of a calculation. In this paper, we have reproduced and joined the distinctive parameters of ALUs by using VHDL on Xilinx Vivado 14.7 and Basys 3 Artix 7 FPGA board.Manish Kumar and Suresh Kumar Jha et.at (15) is presented in this paper that paper basically manages survey for the development of number juggling Logic Unit (ALU) utilizing Hardware Description Language (HDL) utilizing Xilinx Vivado 14.7 and actualize them on Field Programmable Gate Arrays (FPGAs) to investigate the outline parameters. ALU of advanced PCs is a part of rationale plan with the goal of creating suitable calculations keeping in mind the end goal to accomplish a proficient use of the accessible equipment. Speed, power and use of ALU are the measures of the proficiency of a calculation. In this paper, we have reproduced and joined the distinctive parameters of ALUs by using VHDL on Xilinx Vivado 14.7 and Basys 3 Artix 7 FPGA board.

Saumyakaanta Sarangi and Sangita Swain et.al (15) is presented in this paper Digital design is an amazing and very broad field. The applications of digital design are present in our daily life, including computers, calculators, video cameras etc. The VHDL (VHSIC Hardware Description Language) has become an essential tool for designers in the world of digital design. This paper presents implementation of a 4-bit Arithmetic Logic Unit (ALU) using VHDL. ALU of digital computers is an aspect of logic design with the objective of developing appropriate algorithms in order to achieve an efficient utilization of the available hardware. Here the mixed VHDL model of ALU is designed to perform 11 operations which includes both logical and arithmetic operations. The VHDL implementation and functionality test of the 4-bit ALU is done by using the Xilinx ISE 9.2i tool.Priyanka Yadav et.al (16) presented in this paper by increasing the demand of enhancing the ability of processors to handle the more complex and challenging processors has resulted in the integration of a number of processor cores into one chip. Still the load of on the processor is not less in generic system. An Arithmetic Logic Unit (ALU) is the heart of all microprocessors. It is a combinational logic unit that performs its logical or arithmetic operations. ALU is getting smaller and more complex nowadays to enable the development of a more powerful but smaller computer.In this proposed paper a 4 bit ALU chip has been design to the benefits of all the computations are done in parallel and available simultaneously, so no clock resources are wasted. The MUX is then simply used to select the required output.

Suchita Kamble et.al (17) is presented in this paper VHDL implementation of 8-bit arithmetic logic unit (ALU) is presented. The design was implemented using VHDL Xilinx Synthesis tool ISE 13.1 and targeted for Spartan device. ALU was designed to perform arithmetic operations such as addition and subtraction using 8-bit fast adder, logical operations such as AND, OR, XOR and NOT operations, 1’s and 2’s complement operations and compare. ALU consist of two input registers to hold the data during operation, one output register to hold the result of operation, 8-bit fast adder with 2’s complement circuit to perform subtraction and logic gates to perform logical operation. The maximum propagation delay is 13.588ns and power dissipation is 38mW. The ALU was designed for controller used in network interface card.

Shikha Khurana et.al (18) is presented in this paper which primarily deals with the construction of arithmetic Logic Unit (ALU) using Hardware Description Language (HDL) using Xilinx ISE 9.2i and implements them on Field Programmable Gate Arrays (FPGAs) to analyze the design parameters. ALU of digital computers is an aspect of logic design with the objective of developing appropriate algorithms in order to achieve an efficient utilization of the available hardware. The hardware can only perform a relatively simple and primitive set of Boolean & arithmetic operations and are based on a hierarchy of operations that are built by using algorithms employing the hardware. Speed, power and utilization of ALU are the measures of the efficiency of an algorithm. In this paper, we have simulated and synthesized the various parameters of ALUs by using VHDL on Xilinx ISE 9.2i and SPARTAN 3E FPGA board.

Sahil Munjal et.al (19) is presented in this paper, we will learn how to design a low power ALU using VHDL. Advancement in VLSI technology has allowed following Moore’s law for doubling component density on a silicon chip after every three years. An enhancement in the basic function of a transistor has, thus, allowed for designs to be implemented using fewer transistors and reduced interconnections. In published literature, many integrated circuits have been reported which are using multi-input floating gate MOSFETs in standard CMOS process. Thus using the advanced VLSI technology the proposed ALU design is more efficient.

S.Kaliamurthy et.al (20) is presented in this paper which involves the design and development of a single chip VHDL FPGA processor which performs all arithmetic and logical functions and the output is displayed by means of LCD interface. This processor can perform 2n number of operations, where n is the number of control bits. In this design, a 5 bit control input is used so that the processor is capable of performing up to 32 operations. The chip is designed to execute 21 operations for different specified functions and 11 more operations can be worked on for improvements and future works. Two data with a size of 8 to 16 bits can be applied as input and the results are obtained on 4 to 8 hexadecimal digits carrying 32 bits in all. A status flag is also designed with the features such as indication of overflow, carry, borrow and zero value. To implement the above design, Very High Speed Description Language simulation is required which can be performed using Altera or Xilinx softwares. Once the program has been developed, the authors demonstrate the feasibility of the proposed design by incorporating it into a FPGA chip and the required hardware can be brought into effect. The state of each output bit is shown by using Light Emitting Diodes. Based on user’s needs, more features can be added to the designed hardware without hindering the implemented one.

**CHAPTER 3**

**BASICS OF ALU**

**3.1. HISTORY OF ALU**

The concept of ALU was proposed by the mathematician John von New in 1945 for foundations of a new computer called the EDVAC in a report given by him. Throughout the early stages of the information age the cost, size, and power consumption of electronic circuitry was relatively high. As a result all serial computers and many early computers, such as the PDP-8, had a simple ALU that operated on one data bit at a time, although they often presented a wider word size to programmers. One of the earliest computers to have multiple discrete single-bit ALU circuits was the 1948 Whirlwind I, which employed sixteen of such "math units" to enable it to operate on 16-bit words.In 1967, Fairchild introduced the first ALU implemented as an integrated circuit, the Fairchild 3800, consisting of an eight-bit ALU with accumulator. Other integrated-circuit ALUs soon emerged, including four-bit ALUs such as the Am2901 and 74181.

These devices were typically "bit slice" capable, meaning they had "carry look ahead" signals that facilitated the use of multiple interconnected ALU chips to create an ALU with a wider word size. These devices quickly became popular and were widely used in bit-slice minicomputers.Microprocessors began to appear in the early 1970s. Even though transistors had become smaller, there was often insufficient die space for a full-word-width ALU and, consequently some early microprocessors employed a narrow ALU that required multiple cycles per machine language instruction. Examples of this are the popular Zilog Z80, which performs eight-bit additions with a four-bit ALU.

In the coming times, transistor geometries shrank further, following Moore's law, and it became feasible to build wider ALUs on microprocessors.Modern integrated circuit (IC) transistors are orders of magnitude smaller than those of the early microprocessors, making it possible to fit highly complex ALUs on ICs. Nowadays, many modern ALUs have wide word widths, and architectural enhancements such as barrel shifters and binary multipliers that allow them to perform, in a single clock cycle, operations that would have required multiple operations on earlier ALUs.

**3.2. ALU (ARITHMETIC AND LOGIC UNIT)**

Microprocessors/Microcontrollers have a single module which is used to perform arithmetic operations or logical operations on integer values (i.e. first converted into binary form then operation is carried out). This is because many of the different arithmetic and logical operations can be performed using similar (if not identical) hardware. This component used to perform the arithmetic and logical operations in the Microprocessors/Microcontrollers is known as the Arithmetic Logic Unit, or ALU.

The ALU is one of the most important components in a microprocessor, and is typically the part of the processor that is designed first. Once the ALU is designed, the rest of the microprocessor is implemented to feed operands and control codes to the ALU. The block diagram of 64 bit ALU is shown in following fig-



Fig3.1: Block Diagram of ALU

The arithmetic and logic unit (ALU) is used to perform all arithmetic operations (addition, subtraction, multiplication, and division) and logic operations (AND, OR, NOT, XOR, XNOR etc.). The logical operations test various conditions experienced during processing and allows for different actions that can be taken according to the results. The data required to perform the arithmetic and logical functions are inputs from the designated CPU registers and are also known as operands. The ALU depends on basic items to perform its operations. These include number systems, data routing circuits (adders/subtractors), timing, instructions, operands, and registers. Fig1 shows a representative block diagram of an ALU. An ALU first loads the data from an input registers (or registers), an external Control Unit then tells the ALU what operation is to be performed on that data, and then the ALU stores its result into an output register. The Control Unit is responsible for moving the processed data between these registers, ALU and memory.The architecture of ALU is shown in the following fig 3.2-

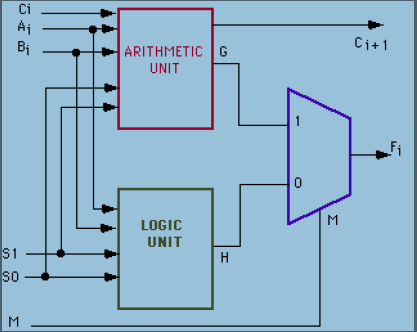


Fig 3.2: Architecture of an ALU

The ALU also process the operand numbers using the same format like all the modern day digital circuits do. The format of modern processors is almost always the two's complement binary number representation. Earlier the computers used a wide variety of number systems, including ones' complement, two's complement sign-magnitude format, and even true decimal systems, with ten tubes per digit.ALUs for each one of these numeric systems had different designs, which made the system to consume more power and space. So these reasons influenced the current preference for two's complement, because this representation made it easier for the ALUs to calculate additions and subtraction.

Most ALUs can perform the following operations:

* Bitwise logic operations (AND, NOT, OR, XOR, NAND, NOR)
* Integer arithmetic operations (addition, subtraction, and sometimes multiplication and division, though this is more expensive).
* Other operations like greater than, equal to, exponential, modulus etc.

**3.3 CIRCUIT OPERATION OF ALU**

An ALU is a combinational logic circuit, which means that its outputs will change asynchronously according to the input changes. In normal operation, stable signals are applied to all of the ALU inputs (as operands) and, when enough time (known as the "propagation delay") passes by for the signals to propagate through the ALU circuitry, the result of the ALU operation appears at the ALU outputs. The external circuitry connected to the ALU is responsible for confirming the stability of ALU input signals throughout the operation and for allowing sufficient time for the signals to propagate through the ALU before sampling the ALU result.

In general, external circuitry controls an ALU by applying signals to its inputs. Typically, the external circuitry employs sequential logic to control the ALU operation, which is paced by a clock signal of a sufficiently low frequency to ensure enough time for the ALU outputs to settle under worst-case conditions. The fig 3.3 shows a simple four-bit ALU -

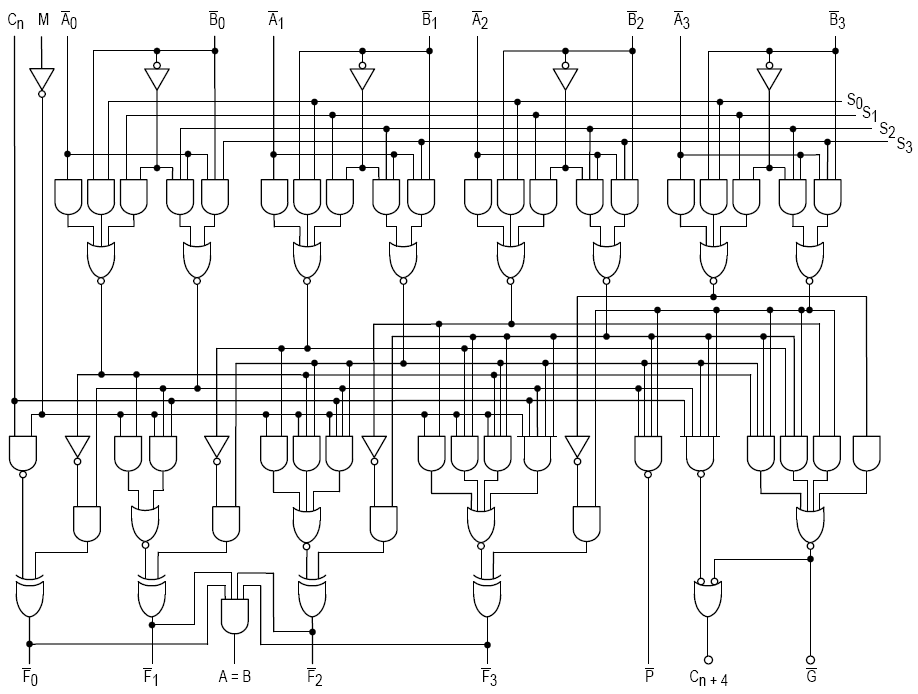


Fig 3.3- The combinational logic circuitry of the 74181 IC i.e. a simple four-bit ALU

For example, a CPU begins an ALU addition operation by routing operands from their sources (which are usually registers) to the ALU's operand inputs, while the control unit simultaneously applies a value to the ALU's opcode input, configuring it to perform addition. At the same time, the CPU also routes the ALU result output to a destination register that will receive the sum. The ALU's input signals, which are held stable until the next clock, are allowed to propagate through the ALU and to the destination register while the CPU waits for the next clock. When the next clock arrives, the destination register stores the ALU result and since the ALU operation has completed, the ALU inputs may be set up for the next ALU operation.

**3.4 FLOWCHART OF AN ALU**

The figure3.4 below shows the flowchart of an ALU:



Fig3.4: Flowchart of an ALU

Hence the above figure 4 gives us a systematic approach of how the data is operated in an ALU which clearly demonstrates the flow of operations in an ALU.

**3.5. VARIOUS ALU OPERATIONS**

The ALU performs three types of operations which are arithmetic operations, logical operations and shifting operations. We can also perform comparison operations like greatest, smallest etc.

Table3.1: Operations Performed By an ALU

|  |  |  |
| --- | --- | --- |
| ARITHMETIC OPERATION | LOGICAL OPERATIONS | SHIFT OPERATIONS |
| Addition | AND | Left Shift |
| Subtraction | OR |
| Multiplication | NOT(Inverter or Compliment) | Right Shift |
| Division | XOR | Left Rotate |
| Comparison between Two operands or inputs | XNOR |
| Combinational Logic Operations i.e. NAND, NOR | Right Rotate |

While designing an ALU we can also use these operations in order to perform other bitwise operations like excess 3 conversions, parity check etc.

**3.6. COMPONENTS OF AN ALU**

The ALU contains three basic units which define theoperations that are about to be performed in the ALU and these three units are :

* Arithmetic Unit
* Logic Unit
* Shifting Unit

The ALU also contains a clock gating circuit which performs the selection of which of these unit is to be used based on the select line signal.

**3.6.1. ARITHMETIC UNIT**

The arithmetic unit is designed in order to perform four operations which are addition, subtraction, increment and decrement. The arithmetic unit's core is a variable block length carry skip adder. The maximum combinational path delay from carry input to carry output has been found to be 22.005 ns which is almost same as that of a ripple carry adder, however, the power dissipation is a little lower than the ripple carry adder. The effect of using carry skip adder with variable block length to minimize carry propagation delay is more pronounced for higher number of bits.

Every individual block is a ripple carry adder. The carry generated in each block enters the ripple carry logic along with the carry generated in previous block. A bitwise XOR operation between the operands is done in the skip logic block and the results are AND’ed together to form the propagation bit. If the propagation bit is 1, it shows that no carry has been generated in the block and the previous carry input is sent directly to the next block. If the propagation bit is 0, a carry has been generated or killed in the block and this is sent to the next block. Variable block length gives better performance in terms of delay with almost 40 percent faster operation than fixed block design. The fig 3.5 shows structure of variable block length carry skip adder-

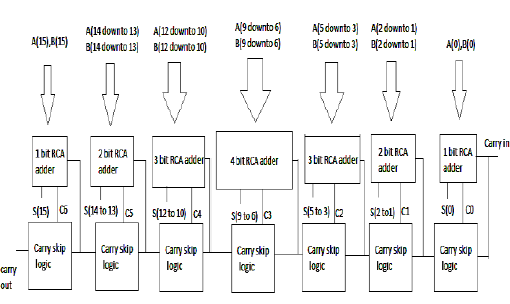


Fig 3.5: Structure of variable bock length carry skip adder

The operand A is applied directly to the Carry skip adder. The operand B goes through a multiplexer and is selected based on select inputs S1 and S0(or more). The nature of the output is based on the signals S0, S1 and carry input. The arithmetic unit can perform seven exclusive operations. The entry of input operands A and B are controlled through registers clocked by the gated clock from the clock gating circuit. Hence, the arithmetic unit is brought into operation only when required and remains inactive during other operations of ALU.

**3.4.2. LOGICAL UNIT**

The operands A and B enter the logic unit through registers controlled by the gated clock, like the arithmetic unit, giving it exclusivity of operation only when required by the ALU. We always design a logic unit that is capable of performing the four basic logic operations: OR, AND, XOR and Complement, because from these four operations, all other logic operations can be derived.The logic unit consists of four gates and a 4:1 multiplexer. The outputs of the gates are applied to the data inputs of the multiplexer. Using to selection lines S0 and S1 one of the data inputs of the multiplexer is selected as the output. For a logic unit of 64-bit, the output will be of 65-bit with 65th bit to be High-impedance

**3.6.3. SHIFTING UNIT**

Shifter unit is used to perform logical shift micro-operation. The shifting of bits of a register can be in either the direction- left or right. The content of a register that has to be shifted first placed onto common bus. This circuit uses no clock pulse. For a shift unit of 64-bit, the output will be of 65-bit with 65th bit to be the carry bit.When the shifting unit is activated the register is shifted left or right according to the selection unit.

**3.6.4. CLOCK GATING CIRCUIT**

The clock gating circuit takes in the clock input and generates a gated clock based on a control signal S2. The gated clock signal is used to activate the arithmetic or logic unit. Preventing unnecessary charging and discharging of the clock signal in inactive modules leads to lower dynamic power dissipation.

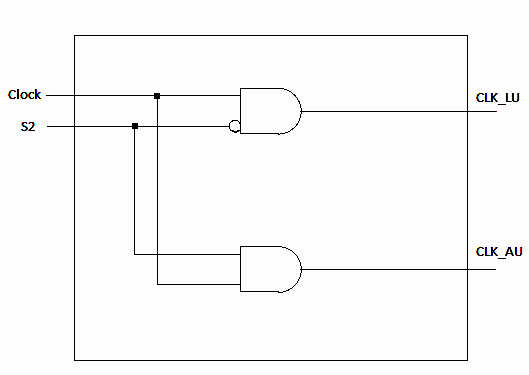


Fig 3.6.: Clock Gating Circuit

**3.7. EXECUTION IN AN ALU**

For executing an instruction in a microprocessor, the instruction is fetched in the first clock cycle using the instruction pointer and then it is decoded.Then the Control Unit (CU) calculates the memory addresses of the operands using ALU by using the offset which has already been obtained from the instruction, loads the address onto the address buses of the CPU and then fetches the operands from their respective locations in memory and feeds them as inputs to the ALU through the system buses.During the next clock cycle the ALU operates on the operands which it has received on the data buses of the CPU and then it produces the result. Also depending on the result of the operation the flag register is set by the ALU. Next the result of operation is either stored to a register or written to the memory. Finally the address of next instruction is calculated and execution proceeds in a similar manner as in the above operation.

**3.8. DESIGN OF ALU IN VHDL**

The ALU design that we are going to do here is going to be in the behavioral modeling of the VHDL in which we first define the operands i.e. the inputs to the ALU and an output line which gives us the output of the operation performed by the ALU on the operands. We also define a Carry output which gives us weather the operation that is being performed gives us a carry output or not. We define basic libraries in the code in order to use the standard functions. A select line is also defined which directs the ALU to the operation that are to be performed on the operands depending on the bit pattern of the select line (since more than one operations are performed in the ALU). We perform this coding on the XILINX ISE 6.0 PROJECT NAVIGATOR and then simulate it on the MODELSIM PE7.5F simulation software.

**3.8.1 SIMULATION OF ALU**

Basically modelsim is a simulation and verification software through which we can see the nature of output that we get from our code and check for the errors in our code by comparing the inputs to the outputs. The design created in theprevious code is simulated on the modelsim software which is linked with the project navigator. We then assign different Clock Pulses to the operands so that the get various bit values to be operated on them and then be verified.

**3.8.2 TEST BENCH OF A VHDL DESIGN**

Test bench is a model code which is used to exercise and verify the correctness of a hardware model that we have designed using the VHDL code. The expressive power of the VHDL language provides us with the capability of writing test bench models also in the same language. A test bench has three main purposes:

* To generate stimulus for simulation (waveforms),
* To apply this stimulus to the entity under test and to monitor the output responses,
* To compare output responses with expected known values.

Again, the language provides a large number of ways to write a test bench.A typical format of a test bench that drives an entity under test is–

entity testbench\_ent is

end entity testbench\_ent;

architecture testbench\_arch of testbench\_ent is

signal declarations

component declarations

begin

component instantiations

stimuli (test vectors)

end architecture testbench\_arch;

The testbench is a specification in VHDL that plays the role of a complete simulation environment for the analyzed system (unit under test, UUT). A testbench contains both the UUT as well as stimuli for the simulation.The UUT is instantiated as a component of the testbench and the architecture of the testbench specifies stimuli for the UUT's ports, usually as waveforms assigned to all output and bidirectional ports of the UUT.The entity of a testbench does not have any ports as this serves as an environment for the UUT. All the simulation results are reported using the assert and report statements.

**CHAPTER 4**

**DESIGN OF 64 BIT ALU WITH 32 OPERATIONS IN VHDL**

In this section we will study about the design of the 64 bit ALU using VHDL about the implementation of the selection lines and the operations we have used in our design. We will also see the RTL view of the designed ALU for the flow of data and the selection lines. We will also write the test bench model for our designed ALU which will help in the verification of the design.Now 64 bits ALU means it will process a 64 bit data and provide output of also 64 bits. So in the entity section a and b are two inputs of 64 bits and outalu is the output i.e. also of 16 bits & sel is a selection line of 5 bit which is required to perform various function on the inputs a and b. So for different value of sel different operations are performed on the two inputs like when sel = "0000” a and b are passed through an adder and an output is generated on outalu. In the architecture case statement is used in which according to changing value of d different operations are performed on inputs a and b.

**4.1 DESIGN OF 64 BIT ALU IN VHDL**

First when we initiate our design after creating the desired VHDL Module file we define the inputs of the design A, B which are vectors of size 0 to 63 (since 64 bit ALU). We also define aSelect Line i.e. also input type with the size 0 to 4. These select lines are direct the operands towards the operation i.e. to be performed by the ALU on them. Since there are 5 bits in the select line input which means that there are 32 operations that our designed ALU will perform on the operands. These 32 operations will be the combination of Logical, Arithmetic, Shifting, Comparative and Mixed Operations. We use case statement in order to assign different operations at different select lines. We also declare two output ports of the ALU as outalu and carryout. The outalu is the output port i.e. a vector of size 0 to 63 and carryout is the output that holds the output with the carry of the result and is a vector of size 0 to 64. The default output of the design will be a summing operation of operands.

**4.1.1.32 OPERATIONS OF THE DESIGNED 64 BIT ALU**

The following Table shows the 32 select lines and the operations associated with them. These operations are the operations that the particular select line will make the ALU do on the operands when selected.

Table1: Select Lines and there corresponding operations

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **SL/NO** | **SELECT LINE** | | | | | **OPERATION** | **DISCRIPTION** |
| **S4** | **S3** | **S2** | **S1** | **S0** |
| 1. | 0 | 0 | 0 | 0 | 0 | A+B | Addition of A and B |
| 2. | 0 | 0 | 0 | 0 | 1 | A-B | Subtraction of B from A |
| 3. | 0 | 0 | 0 | 1 | 0 | B-A | Subtraction of A from B |
| 4. | 0 | 0 | 0 | 1 | 1 | A-1 | Decrement A by 1 |
| 5. | 0 | 0 | 1 | 0 | 0 | A+1 | Increment A by 1 |
| 6. | 0 | 0 | 1 | 0 | 1 | B-1 | Decrement B by 1 |
| 7. | 0 | 0 | 1 | 1 | 0 | B+1 | Increment Bby 1 |
| 8. | 0 | 0 | 1 | 1 | 1 | Left Shift A | Shift 1 BIT to left |
| 9. | 0 | 1 | 0 | 0 | 0 | Right Shift A | Shift 1 BIT to right |
| 10. | 0 | 1 | 0 | 0 | 1 | Left Rotate A | Rotate 1 Bit to left |
| 11. | 0 | 1 | 0 | 1 | 0 | Right Rotate A | Rotate 1 Bit to right |
| 12. | 0 | 1 | 0 | 1 | 1 | Left Shift B | Shift 1 BIT to left |
| 13. | 0 | 1 | 1 | 0 | 0 | Right Shift B | Shift 1 BIT to right |
| 14. | 0 | 1 | 1 | 0 | 1 | Left Rotate B | Rotate 1 Bit to left |
| 15. | 0 | 1 | 1 | 1 | 0 | Right Rotate B | Rotate 1 Bit to right |
| 16. | 0 | 1 | 1 | 1 | 1 | A AND B | Anding b/w A and B |
| 17. | 1 | 0 | 0 | 0 | 0 | A OR B | Oring b/w A and B |
| 18. | 1 | 0 | 0 | 0 | 1 | NOT A | Compliment A |
| 19. | 1 | 0 | 0 | 1 | 0 | NOT B | Compliment B |
| 20. | 1 | 0 | 0 | 1 | 1 | A XOR B | Exoring of A and B |
| 21. | 1 | 0 | 1 | 0 | 0 | A XNOR B | Exnoring of A and B |
| 22. | 1 | 0 | 1 | 0 | 1 | A NOR B | Noring of A and B |
| 23. | 1 | 0 | 1 | 1 | 0 | A NAND B | Nanding of A and B |
| 24. | 1 | 0 | 1 | 1 | 1 | A | Buffer A |
| 25. | 1 | 1 | 0 | 0 | 0 | 2’s Compliment of A | |
| 26. | 1 | 1 | 0 | 0 | 1 | 2’s Compliment of B | |
| 27. | 1 | 1 | 0 | 1 | 0 | A+3 | Excess 3 of A |
| 28. | 1 | 1 | 0 | 1 | 1 | B+3 | Excess 3 of B |
| 29. | 1 | 1 | 1 | 0 | 0 | Comparison | A>B OR A<B |
| 30. | 1 | 1 | 1 | 0 | 1 | Comparison | A=B or A not =B |
| 31. | 1 | 1 | 1 | 1 | 0 | Buffer B | Buffer B |
| 32. | 1 | 1 | 1 | 1 | 1 | Gray to Binary Conversion | Gray to Binary Conversion of A |

**4.1.2. DESCRIPTON OF THE SELECT LINE & ITS OPERATION**

* **Select Line = 00000 = Addition of A and B**

Here the arithmetic unit adds the 64 bit operand A with the other 64 bit operand B hence the output of addition operation will be saved in the 64 bit output outalu. Now if the result contains a carry then it will be saved in carryout as 1 otherwise 0.

* **Select Line = 00001 = Subtraction of B from A**

Here the arithmetic unit subtracts the 64 bit operand B from the other 64 bit operand B hence the output of subtraction operation will be saved in the 64 bit output outalu. Now if the result contains borrow at the MSB then it will be saved in carryout as 1 otherwise 0.

* **Select Line = 00010 = Subtraction of A from B**

Here the arithmetic unit subtracts the 64 bit operand A from the other 64 bit operand B hence the output of subtraction operation will be saved in the 64 bit output outalu. Now if the result contains borrow at the MSB then it will be saved in carryout as 1 otherwise 0.

* **Select Line = 00011 = Decrement by 1 in A**

Here the arithmetic unit subtracts 1 from the 64 bit operand A and the output of decrement operation will be saved in the 64 bit output outalu. Now if the result contains borrow at the MSB then it will be saved in carryout as 1 otherwise 0.

* **Select Line = 00100 = Increment by 1 in A**

Here the arithmetic unit adds 1 to the 64 bit operand A and the output of increment operation will be saved in the 64 bit output outalu. Now if the result contains carry at the MSB then it will be saved in carryout as 1 otherwise 0.

* **Select Line = 00101 = Decrement by 1 in B**

Here the arithmetic unit subtracts 1 from the 64 bit operand B and the output of decrement operation will be saved in the 64 bit output outalu. Now if the result contains borrow at the MSB then it will be saved in carryout as 1 otherwise 0.

* **Select Line = 00110 = Increment by 1 in B**

Here the arithmetic unit adds 1 to the 64 bit operand B and the output of increment operation will be saved in the 64 bit output outalu. Now if the result contains carry at the MSB then it will be saved in carryout as 1 otherwise 0.

* **Select Line = 00111 = Shift 1 bit of A to left**

Here the Shifting Unit shifts one bit of the right end i.e. A [0] and the preceding bits shift on place to their righti.e. A[n] shifts to A[n-1] and at the MSB 1 is added on its place .

* **Select Line = 01000= Shift 1 bit of A to right**

Here the Shifting Unit shifts one bit of the left end i.e. A [63] and the preceding bits shift on place to their left i.e. A[n-1] shifts to A[n] and at the MSB 1 is added on its place .

* **Select Line = 01001= Rotate 1 bit of A to left**

Here the Shifting Unit rotates one bit of the left end i.e. A [63] and then puts it at A [0] and then shifts all the terms from A [0] to A [62] to one bit to their left.

* **Select Line = 01010= Rotate 1 bit of A to right**

Here the Shifting Unit rotates one bit of the right end i.e. A [0] and then puts it at A[63] and then shifts all the terms from A [0] to A [62] to one bit to their right.

* **Select Line = 01011 = Shift 1 bit of B to left**

Here the Shifting Unit shifts one bit of the right end i.e. B [0] and the preceding bits shift on place to their right i.e. B[n] shifts to B[n-1] and at the MSB 1 is added on its place .

* **Select Line = 01100 = Shift 1 bit of B to right**

Here the Shifting Unit shifts one bit of the left end i.e. B [63] and the preceding bits shift on place to their left i.e. B[n-1] shifts to B[n] and at the MSB 1 is added on its place .

* **Select Line = 01101 = Rotate 1 bit of B to left**

Here the Shifting Unit rotates one bit of the left end i.e. B [63] and then puts it at B[0] and then shifts all the terms from B [0] to B[62] to one bit to their left.

* **Select Line = 01110 = Rotate 1 bit of B to right**

Here the Shifting Unit rotates one bit of the right end i.e. A [0] and then puts it at A[63] and then shifts all the terms from A [0] to A [62] to one bit to their right.

* **Select Line = 01111 = Anding of A and B operands**

Here there is a bitwise anding operation between the two operands A and B i.e. A AND B, and the output of this operation is stored in the output outalu.

* **Select Line = 10000 = Oring of A and B operands**

Here there is a bitwise Oring operation between the two operands A and B i.e. A OR B, and the output of this operation is stored in the output outalu.

* **Select Line = 10001 = Compliment of A**

In this operation the output is the compliment of A i.e. 1 is changed with 0 and 0 is changed to 1.

* **Select Line = 10010 = Compliment of B**

In this operation the output is the compliment of B i.e. 1 is changed with 0 and 0 is changed to 1.

* **Select Line = 10011 = Exoring of A and B**

Here there is a bitwise exoring operation between the two operands A and B i.e. A XOR B, and the output of this operation is stored in the output outalu.

* **Select Line = 10100 = Exnoring of A and B**

Here there is a bitwise exnoring operation between the two operands A and B i.e. A XNOR B, and the output of this operation is stored in the output outalu.

* **Select Line = 10101 = Noring of A and B**

Here there is a bitwise noring operation between the two operands A and B i.e. A NOR B, and the output of this operation is stored in the output outalu.

* **Select Line = 10110 = Nanding of A and B**

Here there is a bitwise nanding operation between the two operands A and B i.e. A NAND B, and the output of this operation is stored in the output outalu.

* **Select Line = 10111 = Buffer A**

In this operation the output is the same as the input i.e. 1 is 1 and 0 is 0. In this the same operand is copied at the result.

* **Select Line = 11000 = 2’s Compliment of A**

In this operation the output is the sum of A’s compliment and 1 and also known as the 2’s compliment of A.

* **Select Line = 11001 = 2’s Compliment of A**

In this operation the output is the sum of B’s compliment and 1 and also known as the 2’s compliment of B.

* **Select Line = 11010 = Conversion of binary to excess 3 of A**

Here the operation performed by this select line is that it will add the operand A with 3 and store the result in outalu. So the value stored at the output of ALU is bits larger than A.

* **Select Line = 11011 = Conversion of binary to excess 3 of B**

Here the operation performed by this select line is that it will add the operand B with 3 and store the result in outalu. So the value stored at the output of ALU is bits larger than B.

* **Select Line = 11100= Greater Comparison between A and B**

In this operation the ALU compares both the operands A and B and then the value stored in output 1 if A>B and 0 otherwise. So the output stored at outalu is either 1 or 0.

* **Select Line = 11101 = Equality Comparison between A and B**

In this operation the ALU compares both the operands A and B and then the value stored in output 1 if A=B and 0 if they are not. So the output stored at outalu is either 1 or 0.

* **Select Line = 11110 = Buffer B**

In this operation the output is the same as the input i.e. 1 is 1 and 0 is 0. In this the same operand is copied at the result.

* **Select Line = 11111 = Grey to Binary Conversion of A**

Here the operand A is converted from grey to binary in sel‘11111’ and the result is stored at the output outalu.

**4.2 RTL LOGIC OF THE DESIGNED 64 BIT ALU**

Here we are going to be seeing the RTL logic of the ALU we have designed i.e. this will tell us about the placement of MUX’s and their connections. The following figure 4.1 showsthe RTL logic of our design-

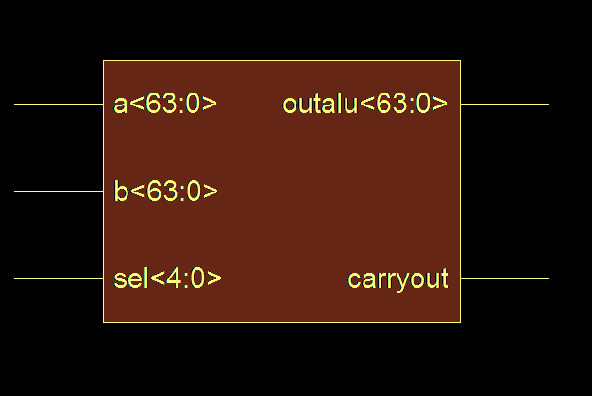


Figure 4.1: The RTL block of the designed ALU

The above diagram shows the RTL Block diagram of the 64 bit ALU and a and b are 64 bit input lines, sel is a 32 select line , outalu is the 64 bit output line and carryout is the one which carries the carry of the 64 bit output. The internal circuit of this block can be seen in the next pages which is quite large and complex as can be seen since the normal ALU’s designed in VHDL have maximum of 8 to 15 operations which is the extensive work we have done in this paper i.e. we have designed a 64 bit ALU with 32 operations which makes our circuit quite large and complex but has an extensive field of operation.

Theabove RTL logic circuit is merely the uotline or we can say it is just a block diagram of the designed ALU. The internal RTL circuit of the abover ALU bock is given below in fig 4.2 to 4.10-

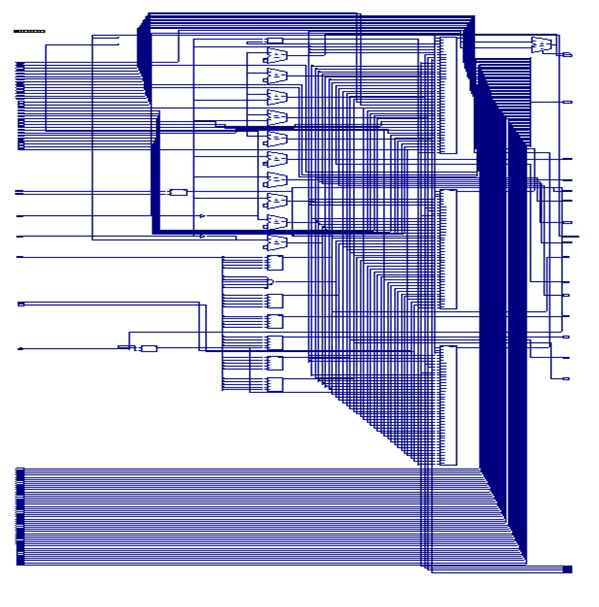


Fig 4.2: Internal RTLcircuit 1

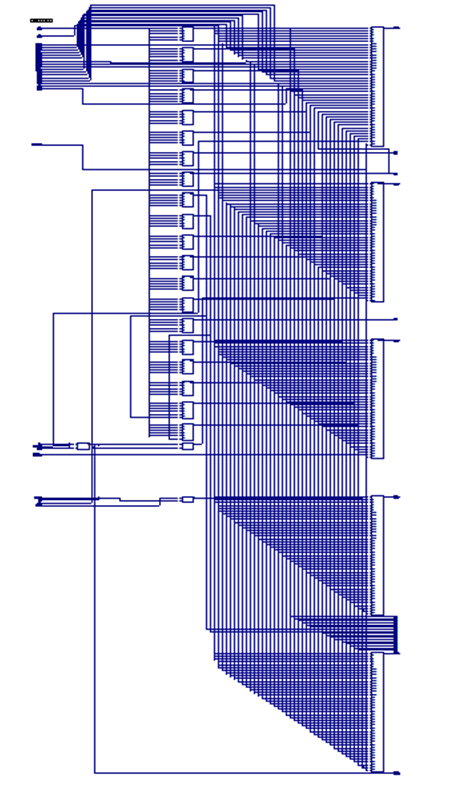


Fig 4.3: Internal RTLcircuit 2

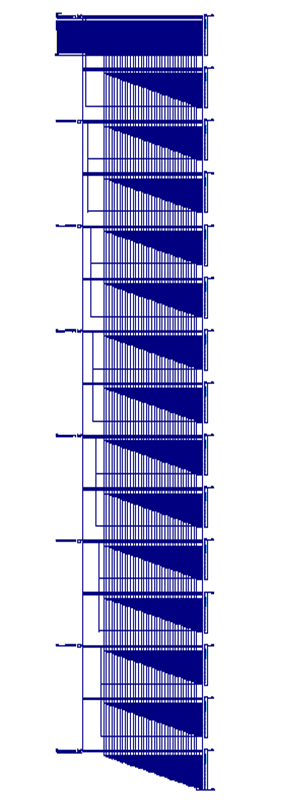


Fig 4.4: Internal RTLcircuit 3

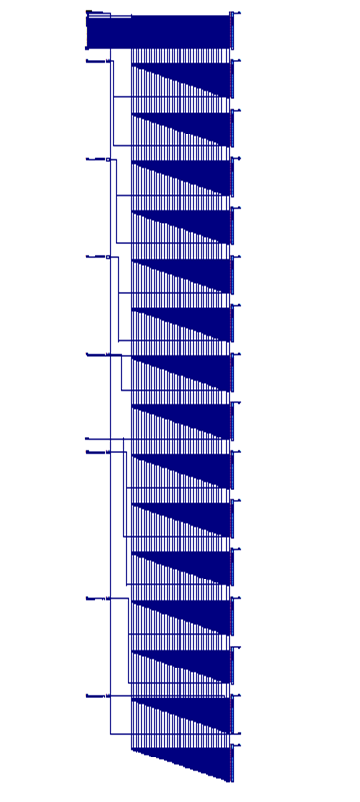


Fig 4.5: Internal RTLcircuit 4

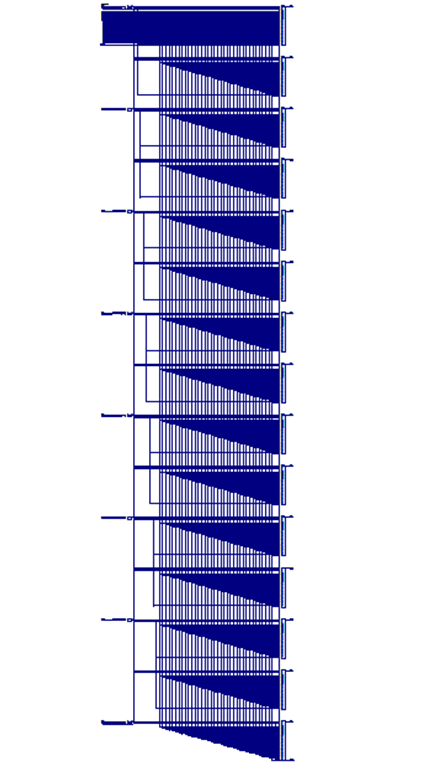


Fig 4.6: Internal RTLcircuit 5

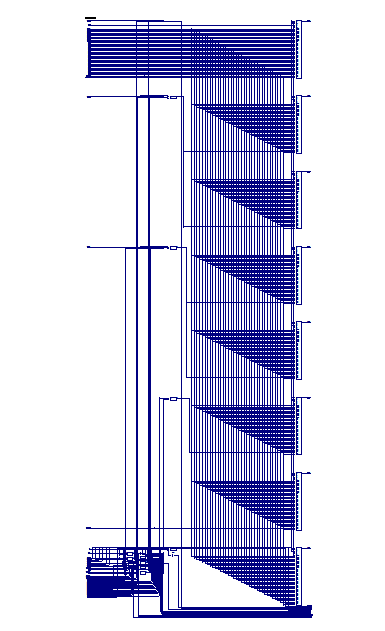


Fig 4.7: Internal RTLcircuit 6

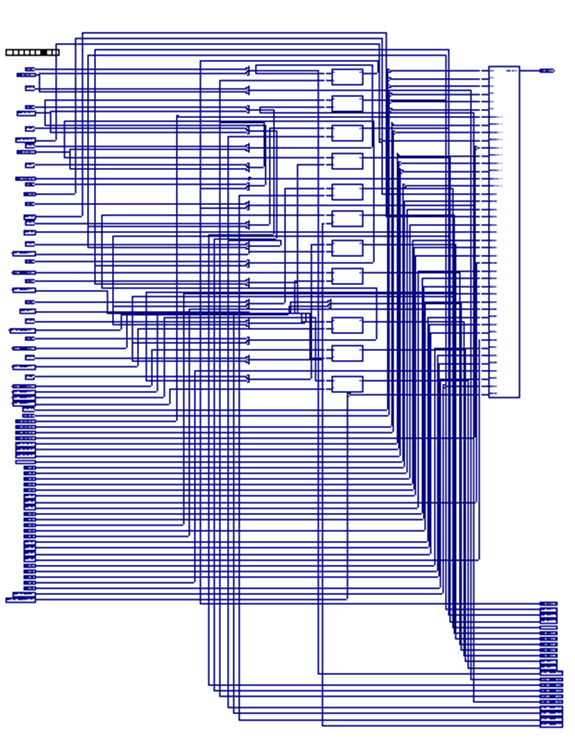


Fig 4.8: Internal RTLcircuit 4.7

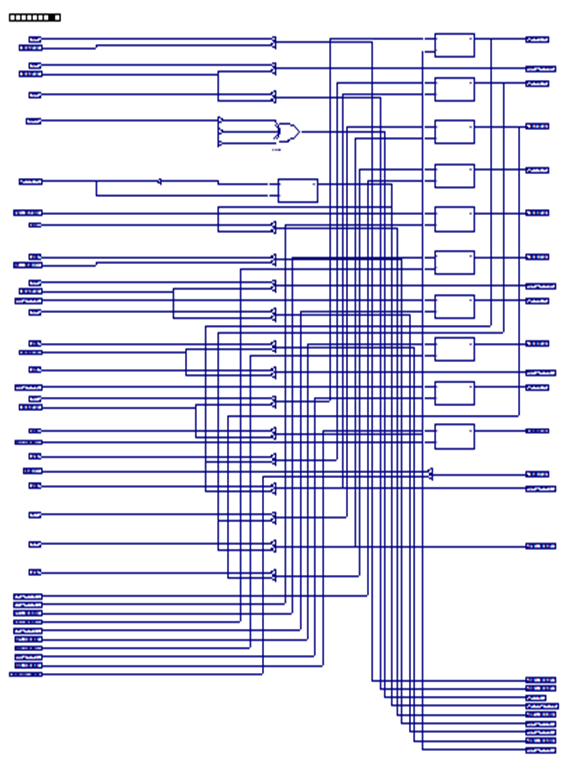


Fig 4.9: Internal RTLcircuit 4.8

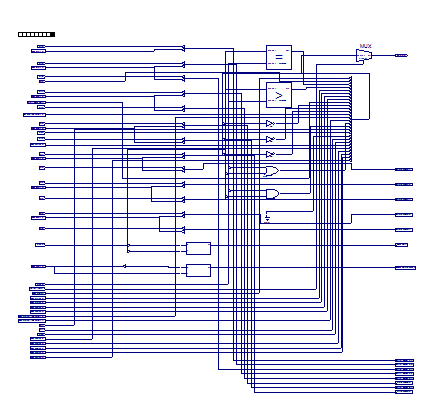


Fig 4.10: Internal RTLcircuit 4.9

**4.3. TESTBENCH MODEL OF THE DESIGNED ALU**

Once we finished writing our code for our ALU design, the next step we perform is that we perform the testing of our ALU VHDL code. One method of testing our design is by writing a testbench code for it. A testbench is used for testing the design and making sure that the design works as per the specified functionalities of the code that we have written. Using a testbench, we can pass inputs of our choice to the design i.e. the ALU. The outputs coming out of our design can be viewed on a simulation waveform or text file or even on console screen.

In order to write a good testbench we need to first understand the design. At least we should know what kind of errors might be there and what kind of input combinations can bring out these errors. Now just to summarize the ways of designing a testbench we note the following points:

* The entity port list of a testbench is always empty.
* All the designs which you want to test, declare them as components in the testbench code.
* The clock process part in the code, is only required for testing designs with a clock (sequential designs).
* Declare all the inputs and outputs in the design to be tested. Make sure you initialize all the inputs to zero, in the system.
* Instantiate the design by any one of the Instantiation methods. Note that this instantiation is written after the begin statement.

The testbench we have created we left the entity section empty and define the ports are defined as signals. After that we create the port map of the ALU we have designed. Then we initialize the inputs with a = x”0000000A” and b = x“00000002” and the select line as sel = “0000”. Now the wait statement changes the input signals after 100ns from high to low.

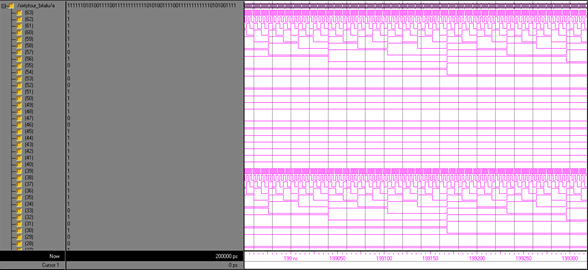
**CHAPTER 5**

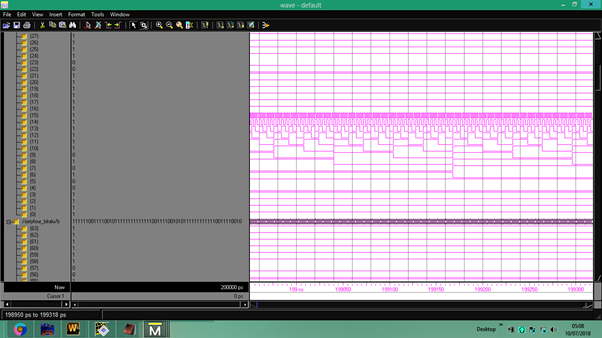
**SIMULATION OF 64 BIT ALU VHDL CODE AND TESTBENCH**

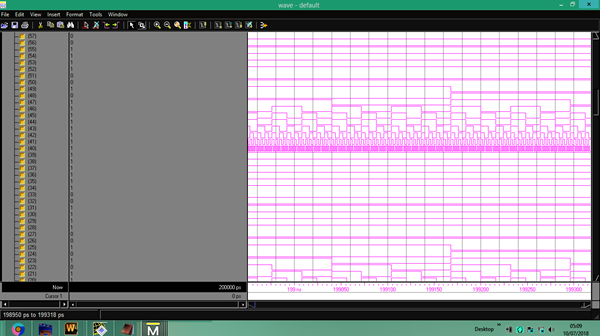
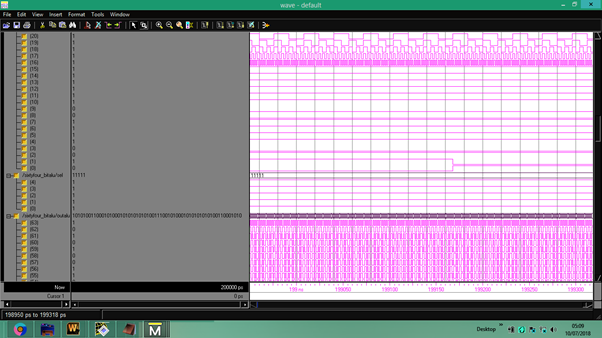
Modelsim is a program created by Mentor Graphics used for simulating your VHDL and Verilog designs. It is the most widely use simulation program in business and education.Simulation is an important step of designing FPGAs and ASICs. Simulation allows the designer to stimulate his or her design and see how the code that they wrote reacts to the given inputs and clock pulses. A great simulation will exercise all possible states of the design to ensure that all input scenarios will be handled appropriately i.e. weather you forget an if statement somewhere or did you remember to give every possible case statement assignment or missed any of them. These are the types of errors that are very easy to make when you do not simulate your design.

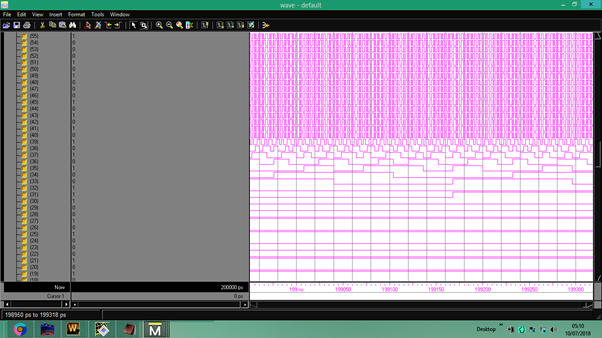
**5.1. SIMULATION OF THE 64 BIT ALU DESIGNED IN VHDL**

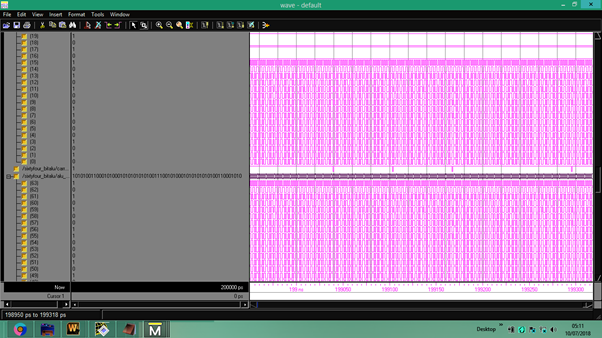
Here we simulate the designed VHDL code by assigning the inputs and select lines their respective clock pulses. The simulation of this design needs many clock pulses since there are 64 bits used hence we shuffle the input pulse to get the desired output of the given design. Since the no. of bits and select lines are too many the graphs has to be seen in many windows at a time. Here we take the complete output for a select line 11111 which performs the grey to binary operation conversion. The result of applied clock pulse is given in fig 1 to fig 10 -

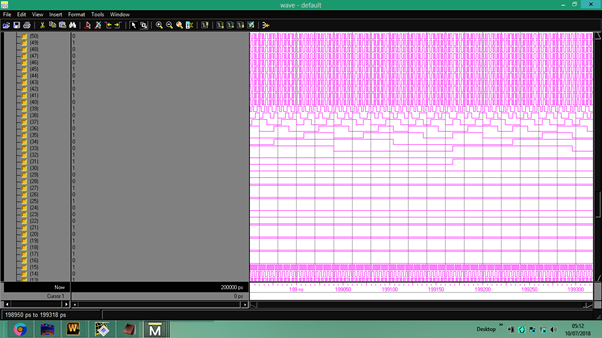


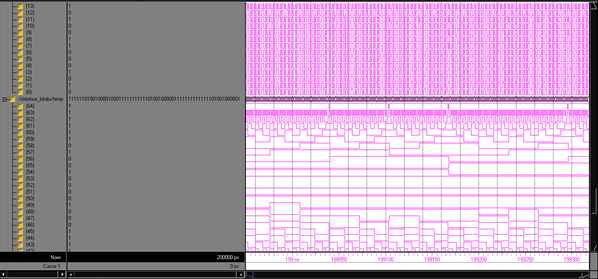


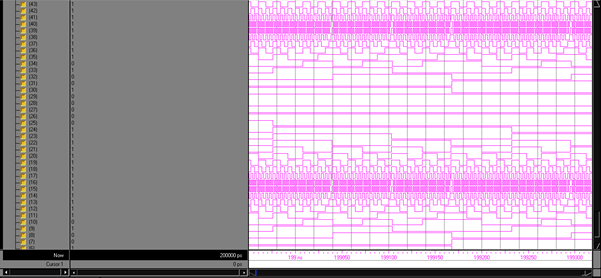
Figure5.1 to 5.4: The above 4 graphs show the input a, b and the select lines in which select line is set at 11111





Figure 5.4 to 5.7: These 3 graphs give the output outalu and carry





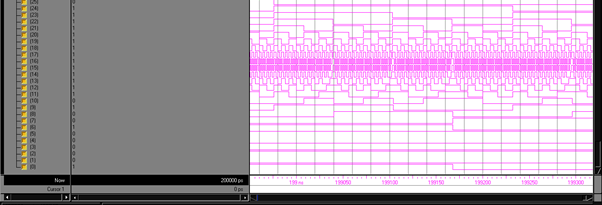
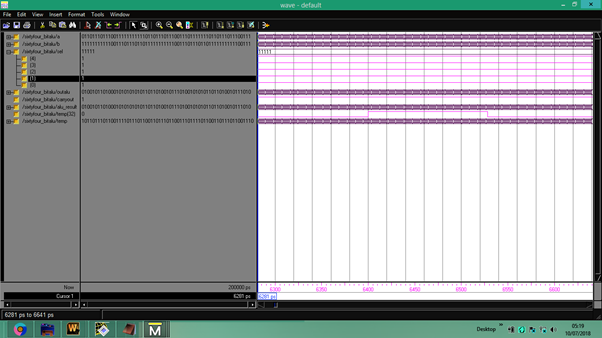


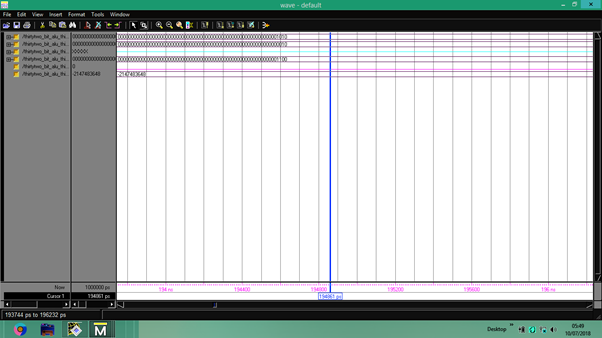
Figure 5.8 to 5.10:The temporary output which is used to find carry of the ALU output

The output of the ALU can be shortened by combining these waveforms but due to high number of bit operations this makes the graph indistinguishable on the basis of pulses. This can be shown in fig 4.11:

Figure 5.11: This shows that output waveform of the simulation when the waves are combined together for select line 11111

**5.2. SIMULATION OF THE 64 BIT ALU TEST BENCH DESIGNED**

The result of the test bench shows weather our designed system is following our desired design or not or it has some errors in it. The graph below shows our test bench result in the combined form and shows the values associated to the output, input, etc nodes when the test bench is simulated is shown in fig 4.12 -

Figure 5.12: This graph shows the test bench result

**CHAPTER-6**

**CONCLUSION AND FUTURE SCOPE**

**6.1 CONCLUSION**

The design of 64 bit ALU with 32 operations in VHDL is done here which shows us that though the circuit formed is a bit complex but this opens up a wide area of improvement in the ALU design and in the select line organization. The ALU that we have designed in this project along with its testbench has also been used verified using simulation software MODELSIM in the above section.

**6.1 FUTURE SCOPE**

We have designed a 64 bit ALU with 32 operations and this ALU can be used in the microprocessors or microcontrollers. We can also increase the nUMBER of bit and operations in the ALU using different types of coding aspects which can be more efficient by consuming less area on the chip, power and having a quick response and decreased delay time.

**REFERENCES**

1. Mohamed EL KHAILI “4-Bit Processing Unit Design Using VHDL Structural Modelling For Multiprocessor Architecture” Laboratory of SSIDA, ENSET, Mohammedia Hassan II University of Casablanca, Morocco.
2. Ankit Chouhan, Srishti dubey, Asst. Prof Arvind Pratap Singh, "16-Bit Arithmetic And Logic Unit Design Using Mixed Type of Modeling In VHDL"Dept. of electronics and communication / RGPV,India.
3. Disha Malik, Richa Singh Rathore,"32‐bit Arithmetic Logical Unit (ALU) using VHDL"1M. Tech Scholar, Department of ECE, Jayoti Vidyapeeth Women’s University, Rajasthan, INDIA.
4. Archana Singh Yadav,Student, Kanpur Institute of Technology, Kanpur M.Tech, Pratyush Tripathi, Asst. Professor, KIT, Kanpur "A VHDL Implementation of a Flexible 16-Bit
5. Abdul Kareem, Mrs. E Kavitha, PG Student [M.Tech in Electronics], "Design and Implementation 16 BIT REVERSIBLE LOGIC ALU with 15-Operations" Department Electronics & Communication Engineering, HKBK College of Engineering, Bangalore, India.
6. Geetanjali and Nishant Tripathi "VHDL Implementation of 32-Bit Arithmetic Logic Unit (ALU)" International Journal of Computer Science and Communication EngineeringIJCSCE Special issue on “Emerging Trends in Engineering” ICETIE 2012.
7. Mukesh P. Mahajan, P. G. Salunke, Y. M. Gaikwad, V. P. Jagtap "Design And Simulation of 64 bit ALU" Department of Electronics & Telecommunication Engineering Sandip Institute of Technology & Research centre, Mahiravani, Nashik, International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)Volume 4, Issue 4, January 2015
8. Ankit Mitra, "Design and implementation of low power 16 bit ALU with clock gating", ISSN: 2278 – 1323, International Journal of Advanced Research in Computer Engineering & Technology (IJARCET), Volume 2, Issue 6, June 2013
9. Rupali Jarwal and Ulka Khire, Microelectronics and VLSI Design, "4-Bit Arithmetic And Logic Unit Design Using Structural Modelling In VHDL", Electronics & Instrumentation Engineering department, SGSITS, Indore, M.P., India, International Journal of Engineering Research & Technology (IJERT), Vol. 2 Issue 3, March - 2013, ISSN: 2278-0181.
10. Kaushik Chandra Deva Sarma, Amlan Deep Borah, Lalan Kumar Mishra "Design And Synthesis Of 32 BIT ALU Using Xilinx ISE V9.1i" (Department of ECE, Central Institute of Technology, Kokrajhar-783370), International Journal of Engineering Research & Technology (IJERT), Vol. 2 Issue 5, May - 2013 ISSN: 2278-0181.
11. Nazia Khan, M. Tech. Scholar, Pankaj Soni,"Design and Implementation of Low Power 32 Bit Arithmetic Logic Unit" Professor Technocrats Institute of Technology- Advance Bhopal (M.P.), International Journal of Computer Applications (0975 – 8887) Volume 166 – No.7, May 2017.
12. Vikesh Ukande, M.Tech Scholar, Ankit Pandit,Assistant Professor ,"High Speed Low Power 32 Bit Alu Implementation" Department of ECE, AISECT University, Institute of science and technology, Bhopal,INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY, ISSN: 2277-9655, CODEN: IJESS7.
13. Rajib Chetia, Asst. Professor, CIT, Kokrajhar, Kaushik Chandra Deva Sarma,Asst. Professor, CIT, Kokrajhar, Gaurab Baruah M.Tech student .in ECT, Gauhati University, "Behavioral Design and Synthesis of 64 BIT ALU using Xilinx ISE", IOSR Journal of Electronics and Communication Engineering (IOSR-JECE)e-ISSN: 2278-2834,p- ISSN: 2278-8735. Volume 7, Issue 4 (Sep. - Oct. 2013),
14. P Bhanusree, Assistant Professor, Y Ashwanth Kumar, K Sravan Kumar, U.G Students, Department of Electronics and Instrumentation Engineering, Vignan Institute of Technology & Science, Deshmukhi, Nalgonda,G Bhargav Sai,Supply Chain Operation Trainee (SCOT), Hindustan Coca-Cola Beverages Pvt Ltd, Hyderabad, "VHDL Implementation Of 64-bit ALU", IOSR Journal of Electronics and Communication Engineering (IOSR-JECE) e-ISSN: 2278-2834,p- ISSN: 2278-8735. Volume 7, Issue 4 (Sep. - Oct. 2013).
15. Manish Kumar, Suresh Kumar Jha, Ravindra Sharma, Jodhpur Institute of Engineering & Technology, Jodhpur, A Case Study: Design of 16 bit Arithmetic and Logical unit using Xillinx 14.7 and Implementation on FPGA Board, International Journal of Engineering Technology Science and Research, IJETSR, ISSN 2394 – 3386, Volume 4, Issue 9, September 2017.
16. Priyanka Yadav, Assistant Professor Department of Electronics & Communication HITM,Agra, Gaurav Kumar, Research Analyst ONE97 Communication LTD. Noida, Sumita Gupta, Assistant Professor Department of Electronics & Communication HITM,Agra, "Design and Implementation of 4-Bit Arithmetic and Logic Unit Chip with the Constraint of Power Consumption", IOSR Journal of Electronics and Communication Engineering (IOSR-JECE) e-ISSN: 2278-2834,p- ISSN: 2278-8735.Volume 9, Issue 3, Ver. V (May - Jun. 2014)
17. Shikha Khurana, M.Tech Student, KIIT Gurgaon, Kanika Kaur, Research Scholar, JJTU, Rajasthan, "IMPLEMENTATION OF ALU USING FPGA", International Journal of Emerging, Trends & Technology in Computer Science (IJETTCS), Volume 1, Issue 2, July – August 2012 ISSN 2278-6856
18. Shikha Khurana, M.Tech Student, KIIT Gurgaon, Kanika Kaur, Research Scholar, JJTU, Rajasthan, India, "IMPLEMENTATION OF ALU USING FPGA", International Journal of Emerging Trends & Technology in Computer Science (IJETTCS), Volume 1, Issue 2, July – August 2012 ISSN 2278-6856
19. Sahil Munjal, Sahil Bhardwaj, Sachin Malik, Dept. of Computer science and Engineering Dronacharya College of Engineering, Gurgaon, Haryana, India, "ALU USING VHDL DESIGNING A 4-BIT LOW POWER ALU USING VHDL", International Journal For Technological Research In Engineering, Volume 2, Issue 8, April-2015 ISSN: 2347 – 4718
20. Prof.S.Kaliamurthy , Ms.U.Sowmmiya, Anna University, Chennai, India, "VHDL Design of FPGA Arithmetic Processor", Global Journal of researches in engineering, Electrical and electronics engineering, Volume 11 Issue 6 Version 1.0 November 2011, Global Journals Inc. (USA), Online ISSN: 2249-4596 & Print ISSN: 0975-5861

**APPENDIX**

**\*\*\*\*\*\*VHDL CODE FOR 64 BIT ALU WITH 32 OPERATIONS\*\*\*\*\*\***

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are

-- provided for instantiating Xilinx primitive components.

--library UNISIM;

--use UNISIM.VComponents.all;

entity sixtyfour\_bitalu is

generic(

constant N: natural:= 1);

Port ( a : in std\_logic\_vector(63 downto 0);

b : in std\_logic\_vector(63 downto 0);

sel : in std\_logic\_vector(4 downto 0);

outalu : out std\_logic\_vector(63 downto 0);

carryout : out std\_logic);

end sixtyfour\_bitalu;

architecture Behavioral of sixtyfour\_bitalu is

signal ALU\_result:std\_logic\_vector(63 downto 0);

signal temp:std\_logic\_vector(64 downto 0);

begin

process(a,b,sel)

begin

case sel is

when "00000" =>

ALU\_result <= a + b;--addition

when "00001" =>

ALU\_result <= a - b;--substraction b from a

when "00010" =>

ALU\_result <= a - b;--subtraction a from b

when "00011" =>

ALU\_result <= a - 1;--decrement 1 in a

when "00100" =>

ALU\_result <= a + 1;--increment 1 in a

when "00101" =>

ALU\_result <= b - 1;--decrement 1 in b

when "00110" =>

ALU\_result <= b + 1;--increment 1 in b

when "00111"=>

ALU\_result <= to\_stdlogicvector(to\_bitvector(a(63 downto 0)) sll 1); --left shift a

when "01000"=>

ALU\_result <= to\_stdlogicvector(to\_bitvector(a(63 downto 0)) srl 1); --right shift a

when "01001"=>

ALU\_result <= to\_stdlogicvector(to\_bitvector(a(63 downto 0)) rol 1); --left rotate a

when "01010"=>

ALU\_result <= to\_stdlogicvector(to\_bitvector(a(63 downto 0)) ror 1); --left rotate a

when "01011" =>

ALU\_result <= to\_stdlogicvector(to\_bitvector(b(63 downto 0)) sll 1); --left shift b

when "01100"=>

ALU\_result <= to\_stdlogicvector(to\_bitvector(b(63 downto 0)) srl 1); --right shift b

when "01101"=>

ALU\_result <= to\_stdlogicvector(to\_bitvector(b(63 downto 0)) rol 1); --left rotate b

when "01110"=>

ALU\_result <= to\_stdlogicvector(to\_bitvector(b(63 downto 0)) ror 1); --left rotate b

when "01111" =>

ALU\_result <= a and b;--and gate

when "10000" =>

ALU\_result <= a or b;-- or gate

when "10001" =>

ALU\_result <= not a ;--not gate

when "10010" =>

ALU\_result <= not b;--not b

when "10011" =>

ALU\_result <= a xor b ;--xor gate

when "10100" =>

ALU\_result <= a xnor b ;--xnor gate

when "10101" =>

ALU\_result <= a nor b ;--nor gate

when "10110" =>

ALU\_result <= a nand b ;--nand gate

when "10111" =>

ALU\_result <= a ; -- a buffer

when "11000" =>

ALU\_result <= not(a)+ x"0000000000000001";-- 2's compliment

when "11001" =>

ALU\_result <= not(b)+ x"0000000000000001";-- 2's compliment

when "11010" =>

ALU\_result <= a + x"0000000000000011"; -- excess 3 (a)

when "11011" =>

ALU\_result <= b + x"0000000000000011"; -- excess 3 (b)

when "11100" =>

if(a>b) then

ALU\_result <= x"0000000000000001";

else

ALU\_result <= x"0000000000000000" ; --grater comparison

end if;

when "11101" =>

if(a=b) then

ALU\_result <= x"0000000000000001";

else

ALU\_result <= x"0000000000000000" ; --equal comparison

end if;

when "11110" =>-- b buffer

ALU\_result <= b ;

when"11111"=> --gray to binary

ALU\_result(63)<= a(63);

ALU\_result(62)<= a(63)xor a(62);

ALU\_result(61)<= a(63)xor a(62)xor a(61);

ALU\_result(60)<= a(60)xor a(61)xor a(62) xor a(63);

ALU\_result(59)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59);

ALU\_result(58)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59)xor a(58);

ALU\_result(57)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57);

ALU\_result(56)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56);

ALU\_result(55)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55);

ALU\_result(54)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54);

ALU\_result(53)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53);

ALU\_result(52)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52);

ALU\_result(51)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51);

ALU\_result(50)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50);

ALU\_result(49)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49);

ALU\_result(48)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48);

ALU\_result(47)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47);

ALU\_result(46)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46);

ALU\_result(45)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45);

ALU\_result(44)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44);

ALU\_result(43)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43);

ALU\_result(42)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42);

ALU\_result(41)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41);

ALU\_result(40)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40);

ALU\_result(39)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39);

ALU\_result(38)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38);

ALU\_result(37)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37);

ALU\_result(36)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36);

ALU\_result(35)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35);

ALU\_result(34)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34);

ALU\_result(33)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33);

ALU\_result(32)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32);

ALU\_result(31)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31);

ALU\_result(30)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(31);

ALU\_result(29)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29);

ALU\_result(28)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28);

ALU\_result(27)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27);

ALU\_result(26)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26);

ALU\_result(25)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25);

ALU\_result(24)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24);

ALU\_result(23)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23);

ALU\_result(22)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22);

ALU\_result(21)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21);

ALU\_result(20)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20);

ALU\_result(19)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19);

ALU\_result(18)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18);

ALU\_result(17)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17);

ALU\_result(16)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16);

ALU\_result(15)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15);

ALU\_result(14)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14);

ALU\_result(13)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13);

ALU\_result(12)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12);

ALU\_result(11)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11);

ALU\_result(10)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10);

ALU\_result(9)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9);

ALU\_result(8)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9)xor a(8);

ALU\_result(7)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9)xor a(8)xor a(7);

ALU\_result(6)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9)xor a(8)xor a(7)xor a(6);

ALU\_result(5)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9)xor a(8)xor a(7)xor a(6)xor a(5);

ALU\_result(4)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9)xor a(8)xor a(7)xor a(6)xor a(5)xor a(4);

ALU\_result(3)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9)xor a(8)xor a(7)xor a(6)xor a(5)xor a(4)xor a(3);

ALU\_result(2)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9)xor a(8)xor a(7)xor a(6)xor a(5)xor a(4)xor a(3)xor a(2);

ALU\_result(1)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9)xor a(8)xor a(7)xor a(6)xor a(5)xor a(4)xor a(3)xor a(2)xor a(1);

ALU\_result(0)<= a(63)xor a(62)xor a(61) xor a(60) xor a(59) xor a(58)xor a(57)xor a(56)xor a(55)xor a(54)xor a(53)xor a(52)xor a(51)xor a(50)xor a(49)xor a(48)xor a(47)xor a(46)xor a(45)xor a(44)xor a(43)xor a(42)xor a(41)xor a(40)xor a(39)xor a(38)xor a(37)xor a(36)xor a(35)xor a(34)xor a(33)xor a(32)xor a(31)xor a(30)xor a(29)xor a(28)xor a(27)xor a(26)xor a(25)xor a(24)xor a(23)xor a(22)xor a(21)xor a(20)xor a(19)xor a(18)xor a(17)xor a(16)xor a(15)xor a(14)xor a(13)xor a(12)xor a(11)xor a(10)xor a(9)xor a(8)xor a(7)xor a(6)xor a(5)xor a(4)xor a(3)xor a(2)xor a(1)xor a(0);

when others =>

ALU\_result <= a + b ;

NULL;

end case;

end process;

outalu <= ALU\_result ;

temp <= ('0' & a) + ('0' & b);

carryout <= temp(64);

end Behavioral;

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END OF CODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*TESTBENCH CODE FOR 64 BIT ALU WITH 32 OPERATIONS\*\*\***

-- VHDL Test Bench Created from source file thirtytwo\_bit\_alu.vhd -- 12:31:33 05/10/2018

-- Notes:

-- This testbench has been automatically generated using types std\_logic and

-- std\_logic\_vector for the ports of the unit under test. Xilinx recommends

-- that these types always be used for the top-level I/O of a design in order

-- to guarantee that the testbench will bind correctly to the post-implementation

-- simulation model.

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

USE ieee.std\_logic\_unsigned.ALL;

USE ieee.numeric\_std.ALL;

ENTITY thirtytwo\_bit\_alu\_thirtytwo\_bit\_alutb\_vhd\_tb IS

END thirtytwo\_bit\_alu\_thirtytwo\_bit\_alutb\_vhd\_tb;

ARCHITECTURE behavior OF thirtytwo\_bit\_alu\_thirtytwo\_bit\_alutb\_vhd\_tb IS

COMPONENT thirtytwo\_bit\_alu

PORT(

a : IN std\_logic\_vector(63 downto 0);

b : IN std\_logic\_vector(63 downto 0);

sel : IN std\_logic\_vector(4 downto 0);

outalu : OUT std\_logic\_vector(63 downto 0);

carryout : OUT std\_logic

);

END COMPONENT;

SIGNAL a : std\_logic\_vector(63 downto 0);

SIGNAL b : std\_logic\_vector(63 downto 0);

SIGNAL sel : std\_logic\_vector(4 downto 0);

SIGNAL outalu : std\_logic\_vector(63 downto 0);

SIGNAL carryout : std\_logic;

SIGNAL i:integer;

BEGIN

uut: thirtytwo\_bit\_alu PORT MAP(

a => a,

b => b,

sel => sel,

outalu => outalu,

carryout => carryout

);

-- \*\*\* Test Bench - User Defined Section \*\*\*

stim\_proc: PROCESS

BEGIN

a <=x"000000000000000A";

b <= x"0000000000000002";

sel <= "00000";

for i in 31 downto 0 loop

sel <= sel + "00001";

wait for 100 ns;

end loop;

a <= x"00000000000000F6";

b <= x"000000000000000A";

wait; -- will wait forever

END PROCESS;

-- \*\*\* End Test Bench - User Defined Section \*\*\*

END;

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END OF CODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***