416151

Implementing Three Stage Pipeline ALU on FPGA

A PROJECT REPORT

Submitted by

VADAN H SHAH

210280111024

In partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING

In

Department of Electronics & Communication Engineering

Lalbhai Dalpatbhai College of Engineering, Ahmedabad

Gujarat Technological University, Ahmedabad

2025

Lalbhai Dalpatbhai College of Engineering, Ahmedabad

120, Circular Road, University Area, Ahmedabad, Gujarat 380015

CERTIFICATE

This is to certify that the project report submitted along with the project entitled **Implementing three stage pipeline ALU on FPGA** has been carried out by **VADAN H SHAH** under my guidance in partial fulfilment for the degree of Bachelor of Engineering in Department of Electronics & Communication Engineering, 8th Semester of Gujarat Technological University, Ahmedabad during the Academic year 2024-25.

Prof. S.B. Prajapati
Internal Guide

Dr. C.H. Vithalani
Head of the Department

Lalbhai Dalpatbhai College of Engineering, Ahmedabad

120, Circular Road, University Area, Ahmedabad, Gujarat 380015

DECLARATION

I hereby declare that the Internship report submitted along with the project entitled **Implementing Three Stage Pipeline ALU on FPGA** in partial fulfilment for the degree of Bachelor of Engineering in Electronics & Communication Engineering to Gujarat Technological University, Ahmedabad. Under the supervision of my internal guide Dr S B Prajapati and that no part of this report has been directly copied from any students' reports or taken from any other source, without providing due reference.

Name of the Student Sign of Student

Vadan Shah

ACKNOWLEDGEMENT

The success and final out comes of this internship required a lot of guidance and assistance from many people and we are extremely privileged to have got this all along the completion of my internship. All that we have done is only due to such supervision and assistance and we would not forget to thank them.

I heartily thank my faculty mentor of Electronics and Communication department Dr S B Prajapati for his constant support and encouragement throughout this period.

Lastly I would like to take the opportunity to thank and express my deep sense of gratitude to our Head of Department Dr C H Vithalani and L D College of Engineering for giving me the opportunity to do project .

ABSTRACT

This report details the design and simulation of a single-precision Three Stage Pipeline Arithmetic Logic Unit (ALU) specifically intended for a math coprocessor within a Normal ALU. Traditional integer and fixed-point representations limit the range of values that can be processed. This project addresses this limitation by employing the particular pipeline Architecture for Three Stage Pipeline ALU to handle a much wider spectrum of values. This wider range is essential for DSPs, where high-precision calculations are frequently required during signal processing tasks.

To achieve this functionality, the design leverages the advantages of Field-Programmable Gate Arrays (FPGAs). FPGAs offer several benefits, including low power consumption, numerous input/output pins, and the ability for parallel processing. These features make FPGAs well-suited for various applications demanding intensive mathematical computations. Examples include signal processing, designing artificial neural networks, image processing, and implementing filters.

The chosen programming language for this project is Verilog HDL. The design itself is implemented on the Xilinx SPARTAN-6 FPGA kit .This allows the ALU design to exploit the processing power of the FPGA within a single chip. In essence, this project demonstrates a method for constructing a high-performance ALU specifically designed for DSP applications. The chosen approach utilizes FPGAs and the Pipeline Architecture to achieve the desired level of precision and efficiency.

TABLE OF CONTENTS

CHAPTER 1

1.1 Introduction To Verilog

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using a HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

Verilog supports a design at many levels of abstraction. The major three are −

- Behavioral level
- Register-transfer level
- Gate level

Behavioral level

This level describes a system by concurrent algorithms (Behavioural). Every algorithm is sequential, which means it consists of a set of instructions that are executed one by one. Functions, tasks and blocks are the main elements. There is no regard to the structural realization of the design.

Register−Transfer Level

Designs using the Register−Transfer Level specify the characteristics of a circuit using operations and the transfer of data between the registers. Modern definition of an RTL code is "Any code that is synthesizable is called RTL code".

Gate Level

Within the logical level, the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (basic gates). Gate level modelling may not be a right idea for logic design. Gate level code is generated using tools like synthesis tools and his netlist is used for gate level simulation and for backend.

Comments

There are two forms to represent the comments

- 1) Single line comments begin with the token // and end with carriage return.
- 2) Multiline comments begins with the token /* and end with token */ Numbers.

You can specify a number in binary, octal, decimal or hexadecimal format. Negative numbers are represented in 2's compliment numbers. Verilog allows integers, real numbers and signed & unsigned numbers.

The syntax is given by − <size> <radix> <value>

Size or unsized number can be defined in <Size> and <radix> defines whether it is binary, octal, hexadecimal or decimal.

Operators

Operators are special characters used to put conditions or to operate the variables.

Ex. >, +, ~, &! =.

Verilog Keywords

Words that have special meaning in Verilog are called the Verilog keywords. For example, assign, case, while, wire, reg, and, or, nand, and module. They should not be used as identifiers. Verilog keywords also include compiler directives, and system tasks and functions.

Gate Level Modelling

Verilog has built-in primitives like logic gates, transmission gates and switches. These are rarely used for design work but they are used in post synthesis world for modelling of ASIC/FPGA cells.

Gate Primitives

The basic logic gates using one output and many inputs are used in Verilog. GATE uses one of the keywords - and, nand, or, nor, xor, xnor for use in Verilog for N-number of inputs and 1 output.

Transmission Gate Primitives

Transmission gate primitives include both, buffers and inverters. They have single input and one or more outputs. In the gate instantiation syntax shown below, GATE stands for either the keyword buf or NOT gate.

Example: Not, buf, bufif0, bufif1, notif0, notif1

Not – n outout inverter

Buf – n output buffer

Bufifo – tristate buffer, active low enable

Bufif1 – tristate buffer, active high enable

Notifo – tristate inverter, active low enable

Notif1 – tristate inverter, active high enable

Data Types

Value Set

Verilog consists of, mainly, four basic values. All Verilog data types, which are used in Verilog store these values −

0 (logic zero, or false condition)

1 (logic one, or true condition)

x (unknown logic value)

z (high impedance state)

use of x and z is very limited for synthesis.

Wire

A wire is used to represent a physical wire in a circuit and it is used for connection of gates or modules. Other specific types of wires are −

Wand (wired-AND) − here value of Wand is dependent on logical AND of all the device drivers connected to it.

Wor (wired-OR) − here value of a Wor is dependent on logical OR of all the device drivers connected to it.

Tri (three-state) − here all drivers connected to a tri must be z, except only one (which determines value of tri).

Register

A reg (register) is a data object, which is holding the value from one procedural assignment to next one and are used only in different functions and procedural blocks. Example −

reg c; // single 1-bit register variable

reg [6:0] d, e; // two 7-bit variables

Input, Output, Inout

These keywords are used to declare input, output and bidirectional ports of a task or module.

Integer

Integers are used in general-purpose variables. They are of ‘reg’ type data type. If the integer is not defined at the time of compiling, then the default size would be 32 bits.

Supply0, Supply1

Supply0 define wires tied to logic 0 (ground) and supply1 define wires tied to logic 1 (power).

Parameter

A parameter is defining a constant which can be set when you use a module, which allows customization of module during the instantiation process.

```
Parameter [2:0] param2 = 3’b110;
```

Operators

Arithmetic Operators

These operators is perform arithmetic operations.

+ (addition), −(subtraction), * (multiplication), / (division), % (modulus)

Relational Operators

These operators compare two operands and return the result in a single bit, 1 or 0.

- == (equal to)
- != (not equal to)
- > (greater than)
- >= (greater than or equal to)
- < (less than)
- <= (less than or equal to)

Bit-wise Operators

Bit-wise operators which are doing a bit-by-bit comparison between two operands.

- & (bitwise AND)
- | (bitwiseOR)
- ~ (bitwise NOT)
- ^ (bitwise XOR)
- ~^ or ^~(bitwise XNOR)

Logical Operators

Logical operators are bit-wise operators and are used only for single-bit operands. They return a single bit value, 0 or 1.

- ! (logical NOT)
- && (logical AND)
- || (logical OR)

Reduction Operators

Reduction operators are the unary form of the bitwise operators and operate on all the bits of an operand vector. These also return a single-bit value.

- & (reduction AND)
- | (reduction OR)
- ~& (reduction NAND)
- ~| (reduction NOR)
- ^ (reduction XOR)
- ~^ or ^~(reduction XNOR)

Shift Operators

Shift operators, which are shifting the first operand by the number of bits specified by second operand in the syntax. Vacant positions are filled with zeros for both directions, left and right shifts .

- << (shift left)
- >> (shift right)

Concatenation Operator

The concatenation operator combines two or more operands to form a larger vector.

The operator included in Concatenation operation is − { }(concatenation)

Replication Operator

The replication operator are making multiple copies of an item.

The operator used in Replication operation is − {n{item}} (n fold replication of an item)

Conditional Operator

Conditional operator synthesizes to a multiplexer.

SYNTAX:-

(Condition) ? (Result if condition true) :(result if condition false);

Bit-Selection “x[2]” and Part-Selection “x[4:2]”

Bit-selects and part-selects are used to select one bit and a multiple bits, respectively, from a wire, reg or parameter vector with the use of square brackets “[ ]”.

Function Calls

In the Function calls, the return value of a function is used directly in an expression without the need of first assigning it to a register or wire.

Modules

Module Declaration

In Verilog, A module is the principal design entity. This indicates the name and port list (arguments). The next few lines which specifies the input/output type (input, output or inout) and width of the each port. The default port width is only 1 bit. The port variables must be declared by wire, wand,. . ., reg. The default port variable is wire. Normally, inputs are wire because their data is latched outside the module. Outputs are of reg type if their signals are stored inside.

Example

```
module sub_add(add, in1, in2, out);
input add; // defaults to wire
input [7:0] in1, in2; wire in1, in2;

output [7:0] out; reg out; ... statements ... End module
```

Continuous Assignment

The continuous assignment in a Module is used for assigning a value on to a wire, which is the normal assignment used at outside of always or initial blocks. This assignment is done with an explicit assign statement or to assign a value to a wire during its declaration. Continuous assignment are continuously executed at the time of simulation.

Module Instantiations

Module declarations are templates for creating actual objects. Modules are instantiated inside other modules, and each instantiation is creating a single object from that template. The exception is the top-level module which is its own instantiation. The module's ports must to be matched to those which are defined in the template. It is specified −

By name, using a dot ".template port name (name of wire connected to port)". Or

By position, placing the ports in the same place in the port lists of both of the template and the instance.

Procedural Assignments

Procedural assignments update register, integer, time, and memory variables based on procedural flow constructs. Unlike continuous assignments, they occur within procedural contexts. The left-hand side denotes the variable receiving the assignment, allowing for full or partial updates. Bit-selects and part-selects must have constant indices, while concatenations partition results for assignment.

Blocking Assignments

A blocking procedural assignment statement must be executed before the execution of the statements that follow it in a sequential block. A blocking procedural assignment statement does not prevent the execution of statements that follow it in a parallel block.

Syntax

The syntax for a blocking procedural assignment is as follows −

```
<lvalue> = <timing_control> <expression>
```

Nonblocking (RTL) Assignments

The non-blocking procedural assignment allows you to schedule assignments without blocking the procedural flow. You can use the non-blocking procedural statement whenever you want to make several register assignments within the same time step without regard to order or dependance upon each other.

Syntax

<lvalue> <= <timing_control> <expression>

Conditions

The conditional statement (or if-else statement) is used to make a decision as to whether a statement is executed or not.

```
if (expression) if (expression != 0)
```

if- else- if

The following construction occurs so often that it is worth a brief separate discussion.

Example

```
if (<expression>)
  <statement>
  else if (<expression>)
  <statement>
  else if (<expression>)
  <statement>
  else
  <statement>
```

This sequence of if’s (known as an if-else-if construct) is the most general way of writing a multi-way decision.

The last else part of the if-else-if construct handles the ‘none of the above’ or default case where none of the other conditions was satisfied.

Case Statement

The case statement is a special multi-way decision statement that tests whether an expression matches one of a number of other expressions, and branches accordingly. The case statement is useful for describing, for example, the decoding of a microprocessor instruction. The case statement has the following syntax −

Example

```
<statement>
::= case ( <expression> ) <case_item>+ endcase
||= casez ( <expression> ) <case_item>+ endcase
||= casex ( <expression> ) <case_item>+ endcase <case_item>
::= <expression> <,<expression>>* : <statement_or_null>
||= default : <statement_or_null>
||= default <statement_or_null>
```

Looping Statements

There are four types of looping statements. They provide a means of controlling the execution of a statement zero, one, or more times.

forever continuously executes a statement.

repeat executes a statement a fixed number of times.

while executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.

for controls execution of its associated statement(s)

The following are the syntax rules for the looping statements −

Example

```
forever
begin
   <statement>+
end

repeat ( <expression> ) <statement>
```

```
while ( <expression> )
begin
  <statement>+
end

for ( <assignment> ; <expression> ; <assignment> ) <statement>
```

Delay Control

The execution of a procedural statement can be delay-controlled by using the following syntax −

The following example delays the execution of the assignment by 10 time units −

#10 rega = regb;

Event Control

The execution of a procedural statement can be synchronized with a value change on a net or register, or the occurrence of a declared event, by using the following event control syntax −

Example

```
<statement>
::= <event_control> <statement_or_null>
<event_control>
::= @ <identifier>
||= @ ( <event_expression> )
<event_expression>
::= <expression>
||= posedge <SCALAR_EVENT_EXPRESSION>
||= negedge <SCALAR_EVENT_EXPRESSION>
||= <event_expression> <or <event_expression>>
```

Procedures: Always and Initial Blocks

All procedures in Verilog are specified within one of the following four Blocks. 1) Initial blocks 2) Always blocks 3) Task 4) Function

The initial and always statements are enabled at the beginning of simulation. The initial blocks executes only once and its activity dies when the statement has finished. In contrast, the always blocks executes repeatedly. Its activity dies only when the simulation is terminated. There is no limit to the number of initial and always blocks that can be defined in a module. Tasks and functions are procedures that are enabled from one or more places in other procedures.

Initial Blocks

The example for the initial statement is as follows −

```
Initial
Begin
  Areg = 0; // initialize a register
 End
```

Always Blocks

The ‘always’ statement repeats continuously throughout the whole simulation run. The example for the always statement is given below

The ‘always’ statement, because of its looping nature, is only useful when used in conjunction with some form of timing control.

```
Always@(*)
Begin
areg = ~areg;
end
```

CHAPTER 2

2.1 FPGA

Field Programmable Gate Arrays (FPGAs) are reconfigurable semiconductor devices that offer a flexible and efficient solution for implementing digital circuits. Unlike Application-Specific Integrated Circuits (ASICs), FPGAs can be programmed and reprogrammed to perform different tasks, making them ideal for prototyping, research, and real-time applications. They consist of an array of configurable logic blocks (CLBs), interconnects, and programmable input/output blocks, which enable designers to implement custom hardware architectures. FPGAs are widely used in communication systems, cryptography, signal processing, and embedded systems due to their parallel processing capabilities, low latency, and adaptability. Their ability to integrate high-speed interfaces and hardware-level security features makes them a preferred choice for applications requiring high performance and reliability.

Field Programmable Gate Arrays (FPGAs) play a crucial role in our **3-Stage Pipelined ALU Processor** project by providing a reconfigurable hardware platform for real-time implementation and testing. Unlike traditional microcontrollers, FPGAs offer parallel processing capabilities, allowing multiple pipeline stages to execute simultaneously, reducing latency and enhancing performance. The use of an FPGA enables efficient debugging, hardware-level optimization, and precise control over the ALU operations, making it suitable for complex arithmetic and logical computations. Furthermore, FPGA-based implementation provides flexibility for future modifications, supports high-speed data processing, and ensures better resource utilization. This makes our ALU processor applicable to embedded systems, signal processing, and secure computing applications.

2.2 LITERATURE REVIEW

The design and optimization of Arithmetic Logic Units (ALUs) have been extensively studied in the field of digital electronics and computer architecture. Various approaches have been explored to enhance processing speed, reduce latency, and improve resource utilization. This section reviews key concepts and previous works related to ALU design, pipelining techniques, and FPGA-based implementations.

1. Arithmetic Logic Unit (ALU) Design.
An ALU is a critical component of any processor, responsible for performing arithmetic (addition, subtraction, multiplication, etc.) and logical (AND, OR, XOR, etc.) operations. Traditional ALU designs operate sequentially, limiting execution speed, whereas modern approaches integrate parallel processing and pipelining to increase throughput. Research has demonstrated that pipelined ALUs significantly improve performance by allowing multiple operations to be processed in different pipeline stages simultaneously.

2. Pipelining in Processor Architecture.
Pipelining is a well-established technique in processor design that enhances execution efficiency by dividing tasks into smaller stages, allowing them to be executed in parallel. Previous research on multi-stage pipelined ALUs has shown improvements in clock cycle efficiency and reduced instruction execution time. Studies indicate that a 3-stage pipeline (Fetch, Decode, Execute) balances complexity and performance, making it suitable for resource-constrained environments such as embedded systems.

3. FPGA-Based ALU Implementation.
FPGAs provide a flexible and high-performance platform for implementing digital circuits due to their parallelism, reconfigurability, and real-time execution capabilities. Prior research has explored FPGA-based ALU designs, demonstrating their advantages in terms of reduced power consumption, high-speed computation, and adaptability. Many studies highlight the use of Verilog/VHDL for ALU implementation, utilizing FPGA resources such as Look-Up Tables (LUTs) and DSP blocks for efficient computation.

2.3 METHODOLOGY

The entire design is implemented by the following steps in progression.

1. **To Design simple fixed point ALU**
2. **To make Multiplication and Division using Computer Algorithms**
3. **Understanding a Processor for 3 stage pipeline**
4. **Adding different modules of processor supporting ALU**
5. **Driving 3 stage Pipeline**

In this project, we selected the **ARM (Advanced RISC Machine) architecture** as the basis for our **3-Stage Pipelined ALU Processor** due to its efficiency, power optimization, and widespread use in embedded and high-performance computing. The key reasons for choosing ARM include:
RISC (Reduced Instruction Set Computing) Architecture: ARM follows a simplified instruction set, enabling faster execution and better power efficiency compared to Complex Instruction Set Computing (CISC) architectures like x86.
Pipelining Support: ARM processors inherently support pipelining, making them an ideal reference for our **3-stage ALU pipeline**.
FPGA Compatibility: ARM-based designs can be efficiently implemented on **FPGAs**, allowing real-time execution, simulation, and debugging.
Scalability and Optimization: ARM architectures are modular and scalable, making them suitable for a variety of applications, from low-power embedded systems to high-performance computing.

Applications of ARM-Based Processors
Due to their efficiency, ARM processors are used in a wide range of applications, including:
- Embedded Systems: ARM processors are the backbone of microcontrollers used in IoT devices, industrial automation, and smart appliances.
- Mobile Computing: Almost all smartphones, tablets, and wearables run on ARM-based processors due to their power efficiency.
- Automotive Industry: Used in ADAS (Advanced Driver Assistance Systems), ECU (Engine Control Units), and infotainment systems.
- Defense and Aerospace: ARM-based secure computing platforms are used in avionics, encrypted communication, and real-time signal processing.

- FPGA-Based Prototyping: ARM processors integrated with FPGA platforms help in rapid prototyping of high-speed digital circuits, such as our 3-Stage Pipelined ALU.

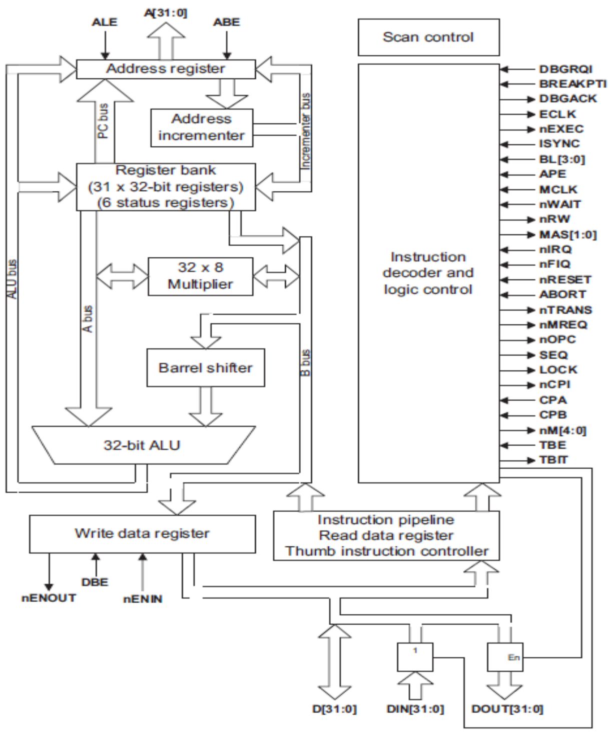


Figure 1 ARM7TDMI core diagram

FIGURE 1: PIPELINE ARCHITECTURE OF AN ARM7.

- **Pipelining**: The three stages allow for overlapping instruction execution, so while one instruction is being executed, another is being decoded, and a third is being fetched. This increases throughput and overall performance.

- **Simplicity**: The ARM7's three-stage pipeline is relatively simple compared to more advanced multi-stage pipelines, making it efficient for embedded systems where power consumption and complexity need to be minimized.

The **ARM7** architecture, specifically the **ARM7TDMI** core, is a microprocessor used in many embedded systems. It supports a **three-stage pipeline** for executing instructions. This pipeline is a crucial part of how the processor fetches, decodes, and executes instructions efficiently, helping improve performance by allowing overlapping execution of instructions.

Here's an overview of the **three-stage pipeline** in the context of ARM7 architecture:

1. Fetch (IF - Instruction Fetch)

- **Description**: In this stage, the ARM7 processor fetches the instruction from memory (typically from the instruction cache or main memory) using the current Program Counter (PC). This step involves reading the instruction and preparing it for decoding.
- **Key Tasks**:
  - The instruction located at the address specified by the Program Counter (PC) is fetched from memory.
  - The PC is incremented to point to the next instruction (or updated in the case of branch instructions).

2. Decode (ID - Instruction Decode)

- **Description**: After the instruction is fetched, the ARM7 processor decodes it to determine what action needs to be taken. This stage involves decoding the instruction's operands and determining the type of operation (e.g., arithmetic, load/store, branch).
- **Key Tasks**:
  - The instruction is decoded into its components (e.g., opcode, registers, and immediate values).
  - The necessary operands (values) for the instruction are fetched from the registers or memory.
  - The control logic prepares the pipeline for the execution stage based on the instruction type (e.g., setting flags for condition codes, determining whether the instruction requires a read/write to memory).

3. Execute (EX - Execution)

- **Description**: In the final stage, the ARM7 executes the instruction. The actual operation specified by the decoded instruction is carried out, such as performing arithmetic, loading/storing data from/to memory, or branching.
- **Key Tasks**:
  - For arithmetic and logical instructions, the ALU (Arithmetic Logic Unit) performs the operation.
  - For load and store instructions, data is transferred between registers and memory.
  - For branch instructions, the Program Counter (PC) is updated to a new value based on the branch condition.

How the Pipeline Works:

In a pipelined processor like ARM7, multiple instructions can be at different stages of the pipeline simultaneously. For example:

- While one instruction is being decoded, another instruction might be fetched, and a third might be executing. This overlapping allows the processor to execute instructions more efficiently and at a higher rate.

Pipeline Hazards:

Although the pipeline helps increase throughput, there are potential issues (called **pipeline hazards**) that can arise, such as:

- **Data Hazards**: Occur when an instruction depends on the result of a previous instruction still being processed in the pipeline.
- **Control Hazards**: Happen when there are branch instructions, and the processor needs to figure out the next instruction to execute.
- **Structural Hazards**: Arise when hardware resources (such as the memory or ALU) are needed simultaneously by multiple instructions.

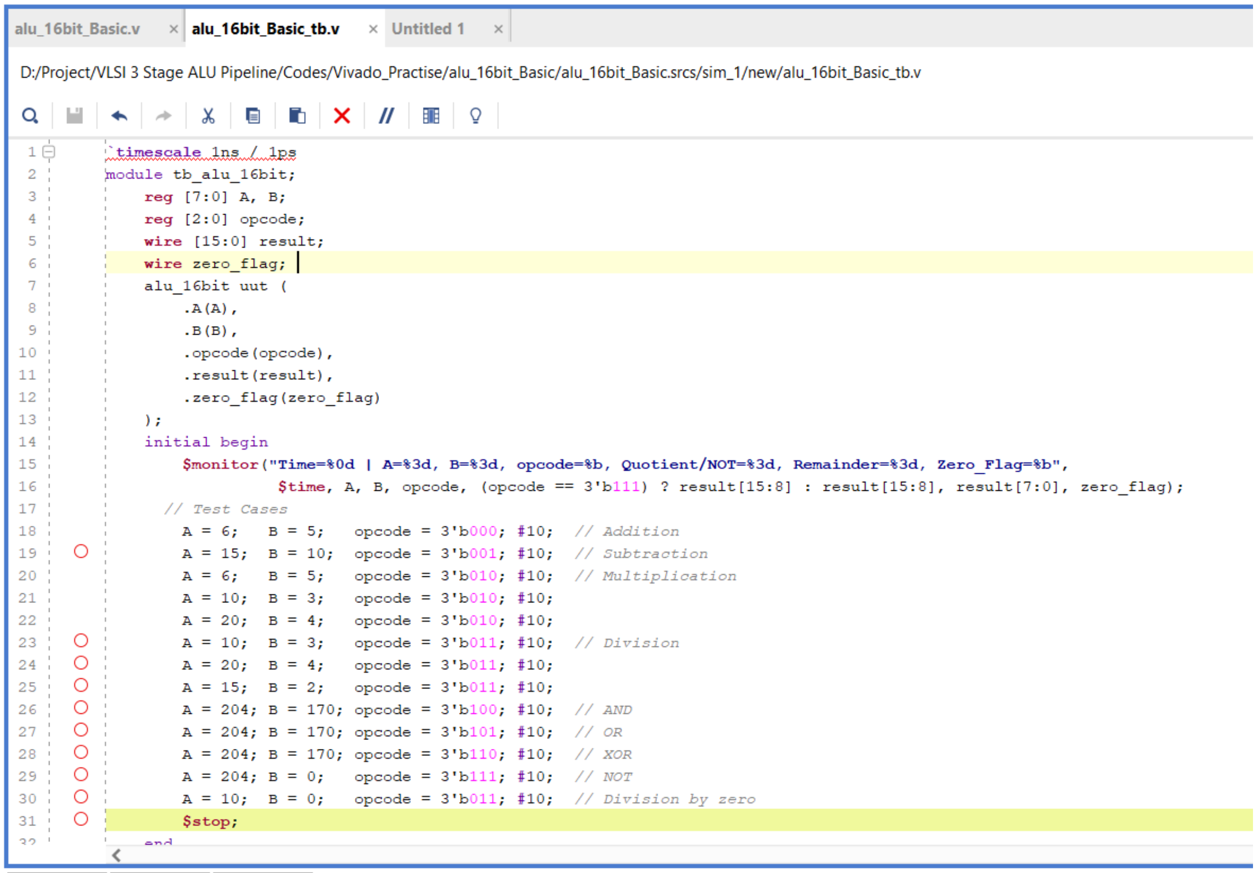
In ARM7, the three-stage pipeline is relatively simple compared to more modern processors with deeper pipelines (e.g., 5-stage or 10-stage pipelines). However, it allows for efficient execution of simple embedded tasks while maintaining low power consumption.

2.4 VERILOG CODE

```
 `timescale 1ns / 1ps

module alu_16bit (
    input [7:0] A, B,
    input [2:0] opcode,
    output reg [15:0] result,
    output reg zero_flag
);
    reg [7:0] Q, M;
    reg [8:0] A_reg;  // 9-bit to handle sign
    reg Qm1;
    integer i;

    always @(*) begin
        case (opcode)
            3'b000: result = A + B;   // Addition
            3'b001: result = A - B;   // Subtraction

            3'b010: begin  // Booth's Multiplication
                A_reg = 9'b0;         // A = 0 (9-bit for sign extension)
                Q = A;                // Q = Multiplier
                M = B;                // M = Multiplicand
                Qm1 = 0;              // Q-1 = 0

                for (i = 0; i < 8; i = i + 1) begin
                    case ({Q[0], Qm1})
                        2'b10: A_reg = A_reg - M; // A = A - M
                        2'b01: A_reg = A_reg + M; // A = A + M
                    endcase

                    // Arithmetic Right Shift (ARS) {A_reg, Q, Qm1}
                    {A_reg, Q, Qm1} = {A_reg[8], A_reg, Q};
    end

    result = {A_reg[7:0], Q};  // Final result in {A, Q}
end

3'b011: begin  // Restoring Division
    A_reg = 8'b0; // A = 0
    Q = A;        // Q = Dividend
    M = B;        // M = Divisor

    if (B == 0) begin
        result = 16'b0;  // Division by zero returns 0
    end
    else begin
        for (i = 0; i < 8; i = i + 1) begin
            {A_reg, Q} = {A_reg, Q} << 1;
            A_reg = A_reg - M;

            if (A_reg[7] == 1) begin
                A_reg = A_reg + M;
                Q[0] = 0;
            end else begin
                Q[0] = 1;
            end
        end
        result = {Q, A_reg[7:0]};  // Quotient in upper 8 bits, Remainder in lower 8 bits
    end
end

3'b100: result = A & B;   // AND
3'b101: result = A | B;   // OR
3'b110: result = A ^ B;   // XOR
3'b111: result = ~A;      // NOT

default: result = 16'b0;
```

```
        endcase

        zero_flag = (result == 16'b0) ? 1 : 0;
    end
endmodule
```

Testbench

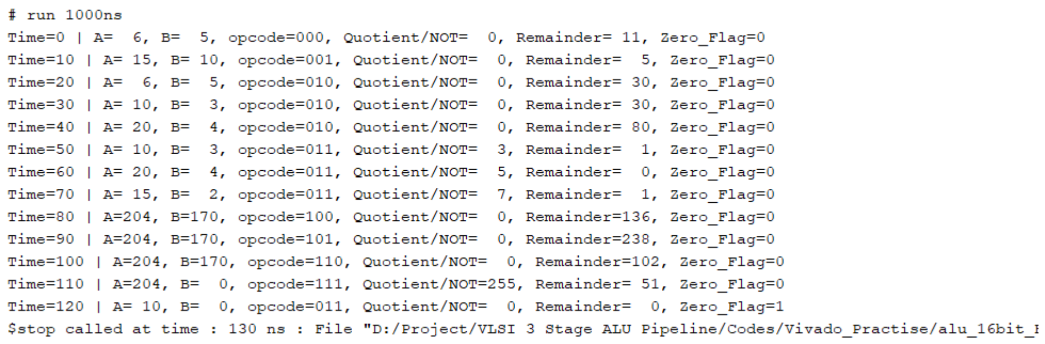


alu_16bit_Basic.v × alu_16bit_Basic_tb.v × Untitled 1 ×

D:/Project/VLSI 3 Stage ALU Pipeline/Codes/Vivado_Practise/alu_16bit_Basic/alu_16bit_Basic.srcs/sim_1/new/alu_16bit_Basic_tb.v

```
`timescale 1ns / 1ps
module tb_alu_16bit;
    reg [7:0] A, B;
    reg [2:0] opcode;
    wire [15:0] result;
    wire zero_flag;
    alu_16bit uut (
        .A(A),
        .B(B),
        .opcode(opcode),
        .result(result),
        .zero_flag(zero_flag)
    );
    initial begin
        $monitor("Time=%0d | A=%3d, B=%3d, opcode=%b, Quotient/NOT=%3d, Remainder=%3d, Zero_Flag=%b",
                 $time, A, B, opcode, (opcode == 3'b111) ? result[15:8] : result[15:8], result[7:0], zero_flag);
      // Test Cases
        A = 6;   B = 5;   opcode = 3'b000; #10;  // Addition
        A = 15;  B = 10;  opcode = 3'b001; #10;  // Subtraction
        A = 6;   B = 5;   opcode = 3'b010; #10;  // Multiplication
        A = 10;  B = 3;   opcode = 3'b010; #10;
        A = 20;  B = 4;   opcode = 3'b010; #10;
        A = 10;  B = 3;   opcode = 3'b011; #10;  // Division
        A = 20;  B = 4;   opcode = 3'b011; #10;
        A = 15;  B = 2;   opcode = 3'b011; #10;
        A = 204; B = 170; opcode = 3'b100; #10;  // AND
        A = 204; B = 170; opcode = 3'b101; #10;  // OR
        A = 204; B = 170; opcode = 3'b110; #10;  // XOR
        A = 204; B = 0;   opcode = 3'b111; #10;  // NOT
        A = 10;  B = 0;   opcode = 3'b011; #10;  // Division by zero
        $stop;
    end
```

Output



```
# run 1000ns
Time=0 | A=  6, B=  5, opcode=000, Quotient/NOT=  0, Remainder= 11, Zero_Flag=0
Time=10 | A= 15, B= 10, opcode=001, Quotient/NOT=  0, Remainder=  5, Zero_Flag=0
Time=20 | A=  6, B=  5, opcode=010, Quotient/NOT=  0, Remainder= 30, Zero_Flag=0
Time=30 | A= 10, B=  3, opcode=010, Quotient/NOT=  0, Remainder= 30, Zero_Flag=0
Time=40 | A= 20, B=  4, opcode=010, Quotient/NOT=  0, Remainder= 80, Zero_Flag=0
Time=50 | A= 10, B=  3, opcode=011, Quotient/NOT=  3, Remainder=  1, Zero_Flag=0
Time=60 | A= 20, B=  4, opcode=011, Quotient/NOT=  5, Remainder=  0, Zero_Flag=0
Time=70 | A= 15, B=  2, opcode=011, Quotient/NOT=  7, Remainder=  1, Zero_Flag=0
Time=80 | A=204, B=170, opcode=100, Quotient/NOT=  0, Remainder=136, Zero_Flag=0
Time=90 | A=204, B=170, opcode=101, Quotient/NOT=  0, Remainder=238, Zero_Flag=0
Time=100 | A=204, B=170, opcode=110, Quotient/NOT=  0, Remainder=102, Zero_Flag=0
Time=110 | A=204, B=  0, opcode=111, Quotient/NOT=255, Remainder= 51, Zero_Flag=0
Time=120 | A= 10, B=  0, opcode=011, Quotient/NOT=  0, Remainder=  0, Zero_Flag=1
$stop called at time : 130 ns : File "D:/Project/VLSI 3 Stage ALU Pipeline/Codes/Vivado_Practise/alu_16bit_
```

CHAPTER 3
Introduction

Multiplication Algorithm Used in ALU

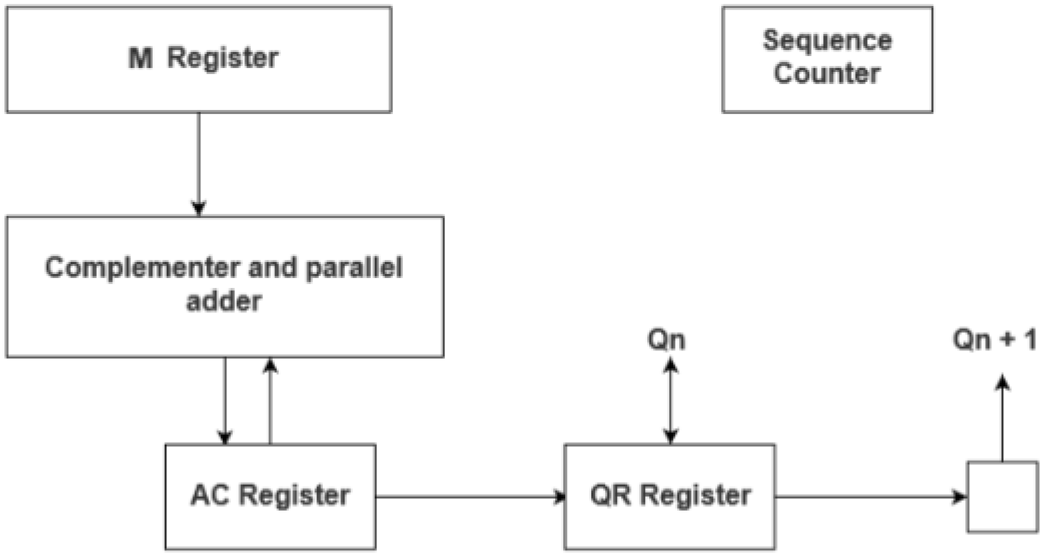
The Booth Algorithm is a widely used technique for multiplying binary integers particularly in signed 2’s complement representation. It was proposed by Andrew D in 1951. This algorithm is an efficient way to perform multiplication, minimizing the number of additions and subtractions. The algorithm optimizes the multiplication process to identify the patterns in binary representation of numbers. It makes highly efficient hardware implementations such as processors and digital signal processors (DSPs).
In this article, we will explore Booth’s algorithm, its working, advantages, limitations, and applications.

What is the Booth Algorithm in Computer Organization?

Booth's algorithm is a technique used for multiplying binary numbers. It reduces the number of operations required for multiplication by encoding the multiplier in a specific way. When compared to traditional multiplication algorithms, it treats both positive and negative numbers in a consistent manner, which makes it highly efficient for computer systems where signed numbers are commonly used.
In Booth’s Algorithm, the multiplication process involves:

- Inspecting two consecutive bits of the multiplier at a time.
- Adding, subtracting, or leaving unchanged the current product based on these bits.
- Shifting the product right after each operation gradually forms the final result.

Implementation of Booth's Algorithm



How the Booth's Algorithm Works?

Booth's algorithm operates by examining consecutive pairs of bits in the multiplier and performing arithmetic operations based on these pairs. It uses the 2's complement representation of numbers to handle both positive and negative values. Below is the step-by-step procedure for Booth's Multiplication Algorithm in Computer Architecture. Let's take an example of the numbers such as m = 5 (multiplicand) and r = −3 (multiplier)

Step 1: Initialize the registers

We first convert both numbers to their binary forms.

Now, m = 5 (binary: 0101), r = −3 (binary: 1101) in 4-bit 2's complement representation, as we are working with 4-bit registers. We will assume we are using 4 bits for the multiplicand and multiplier.

- Multiplicand M = 0101
- Multiplier Q =1101

Now we initialize the following registers:

- AC (Accumulator) = 0000 (4-bit register)
- QR (Multiplier Register) = Q =1101

- Qn+1 = 0 (additional bit, initialized to 0)
- SC (Sequence Counter) = 4 (since the multiplier has 4 bits)

Step 2: Start the algorithm We need to check the last two bits of the multiplier Qn and Qn+1 and follow Booth's algorithm rules: **First iteration (SC = 4):** Qn = 1 (from QR), Qn+1 = 0 (initialized to 0) The last two bits are 10, so we perform a subtraction of the multiplicand from AC:

- AC = AC−M
- AC = 0000−0101 = 1011 (in 2's complement representation of 4 bits)

Now perform the arithmetic right shift operation on AC and QR:

- **Before shift:** AC = 101, QR = 1101, Qn+1 = 0
- **After the right shift:** AC = 1101, QR = 1110, Qn+1 = 1

Decrement the sequence counter SC = 3. **Second iteration (SC = 3)**: Qn = 0, Qn+1 = 1The last two bits are 01, so we perform an addition of the multiplicand to AC:

- AC = AC+M
- AC = 1101+0101 = 0010 (in 4-bit 2's complement representation)

Perform the arithmetic right shift operation on AC and QR:

- **Before shift:** AC = 0010, QR = 1110, Qn+1 = 1
- **After right shift:** AC = 0001, QR = 1111, Qn+1 = 0

Decrement the sequence counter SC = 2. **Third iteration (SC = 2):** Qn = 1, Qn+1 = 0The last two bits are 10, so we perform a subtraction of the multiplicand from AC:

- AC=AC−M
- AC = 0001−0101=1011 (in 2's complement)

Perform the arithmetic right shift operation on AC and QR:

- **Before shift:** AC = 1011, QR = 1111, Qn+1 = 0
- **After right shift:** AC = 1101, QR = 1111, Qn+1 = 1

Decrement the sequence counter SC = 1. **Fourth iteration (SC = 1):** Qn = 1, Qn+1 = 1The last two bits are 11, so we do not perform any addition or subtraction. We just perform the arithmetic right shift:

- **Before shift:** AC = 1101, QR = 1111, Qn+1 = 1
- **After right shift:** AC = 1110, QR = 1111, Qn+1 = 1

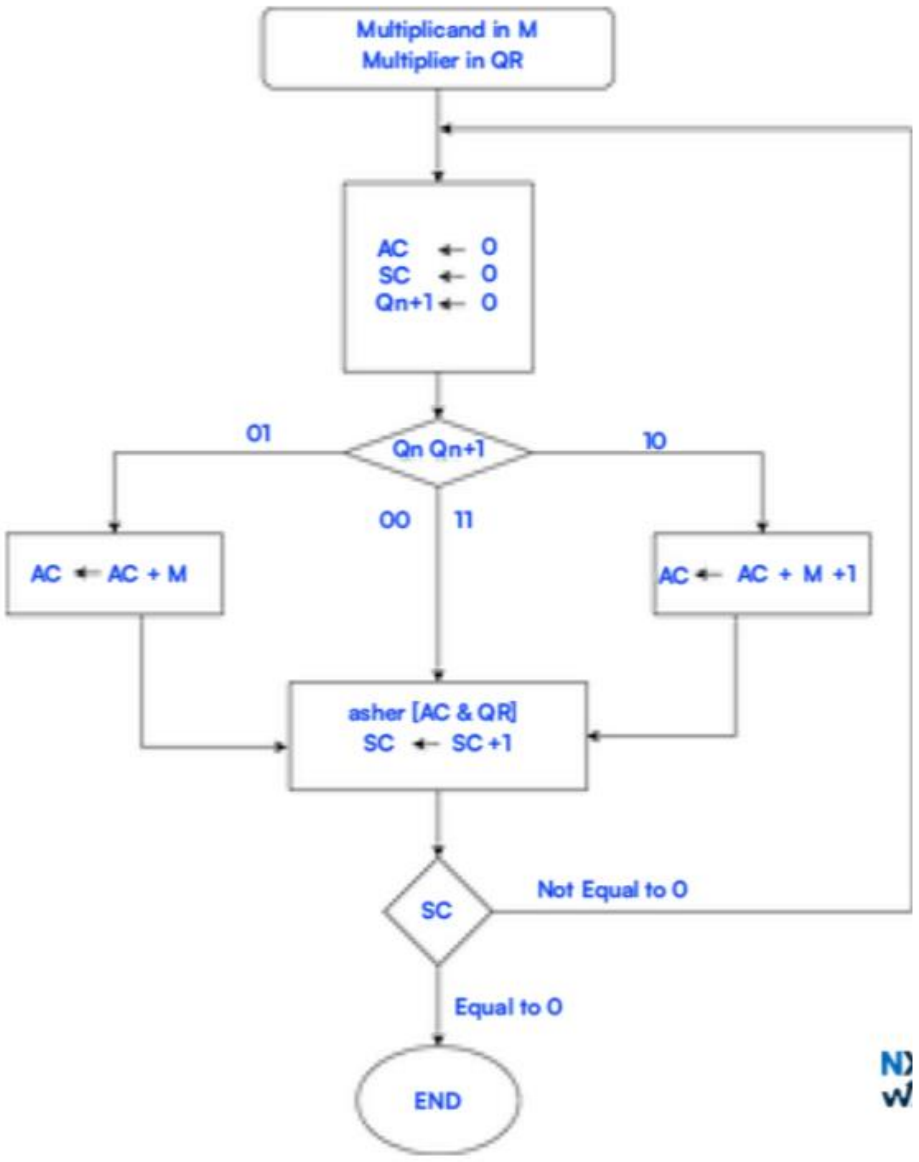
Decrement the sequence counter SC = 0. Step 3: Result At the end of the algorithm, the result of the multiplication will be stored in the AC and QR registers. Where, AC = 1110, QR = 1111To interpret this result:

- The AC and QR together form the final product in 8-bit 2's complement representation: 11101111.

- This is the 8-bit 2’s complement representation of -15, which is the product of 5× (−3) = −15

  Final Answer

  The product of 5× (−3) using Booth's algorithm is -15.



What are the Advantages of Booth’s Multiplication Algorithm?

The advantages of Booth’s multiplication algorithm are:

- This algorithm is used to multiply signed binary numbers, handling both positive and negative numbers effectively without requiring separate logic for negative numbers.
- The algorithm minimizes the number of additions and subtractions required during

multiplication, leading to a faster computation compared to traditional methods.

- Booth’s algorithm is well-suited for hardware implementation because it reduces the complexity of arithmetic operations, which is crucial for devices with limited resources like embedded systems.
- The algorithm efficiently handles sequences of 0’s or 1’s in the multiplier, leading to fewer shifts and operations.
- It is widely used in processors, digital signal processors (DSPs), and other hardware accelerators to optimize the speed of multiplication operations.

What are the Disadvantages of Booth’s Multiplication Algorithm?

The disadvantages of Booth’s Multiplication Algorithm are:

- It can be more difficult to understand and implement compared to traditional multiplication methods. It requires careful handling of sign bits and conditional additions or subtractions.
- It is primarily designed for signed binary numbers. To use it for unsigned numbers, modifications are required, making it less versatile for all types of binary multiplication.
- The algorithm involves multiple iterations of addition, subtraction, and shifting, which may lead to higher latency compared to simpler multiplication methods.
- Although it reduces the number of operations, the algorithm may still consume more power due to the multiple cycles involved in shifting and adding or subtracting the multiplicand.

Applications of Booth’s Multiplication Algorithm

The applications of Booths’ Multiplication Algorithm:

- This algorithm is often integrated into the Arithmetic Logic Unit (ALU) of microprocessors to perform efficient binary multiplication.
- In DSP applications, where the algorithm helps to reduce computation time for multiplication operations such as filtering and convolution.
- The systems often require efficient exponentiation and multiplication of large numbers.
- This is used in custom hardware accelerators to speed up specific tasks, such as image processing or machine learning, where large-scale multiplications are frequently performed.

Conclusion

In conclusion, Booth’s algorithm is an essential technique in modern computer

organization, designed to efficiently multiply signed binary numbers. While it offers substantial performance benefits in terms of reduced operations and hardware efficiency, it also comes with challenges such as complexity and increased latency.
better results in terms of speed performance, is used than standard collection algorithms.

Division Algorithm Used in ALU

In this project, the Restoring Division Algorithm has been implemented for performing division operations in the 3-Stage Pipelined ALU Processor. This algorithm is widely used in hardware-based division due to its structured approach and suitability for FPGA-based arithmetic logic units (ALUs).

Justification for Using the Restoring Division Algorithm

- Hardware Simplicity:
  The algorithm follows a sequential subtraction method, making it easier to implement in digital logic compared to complex division methods like Newton-Raphson or Goldschmidt's algorithm, which require floating-point arithmetic.
- Efficient for Integer Division:
  The Restoring Division Algorithm is well-suited for integer arithmetic, allowing efficient computation of quotients and remainders without the need for complex floating-point calculations.
- Ease of FPGA Implementation:
  FPGA-based designs effectively utilize shift and subtract operations, which align well with the Restoring Division Algorithm since it primarily involves bitwise shifts, subtractions, and comparisons rather than multiplications or approximations.
- Balanced Performance and Resource Utilization:
  While alternative methods like Non-Restoring Division eliminate the restoring step, the Restoring Division Algorithm ensures a balance between hardware complexity and execution speed, making it a good fit for the 3-stage pipelined ALU design.

Working Principle of the Restoring Division Algorithm

The Restoring Division Algorithm follows an iterative subtraction-based approach, where the divisor is repeatedly subtracted from the dividend while shifting bits to compute the quotient. The process is as follows:

1) Initialize the dividend in the remainder register and shift it left by one bit.
   Subtract the divisor from the remainder.
2) If the result is negative, restore the remainder by adding the divisor back and shift a 0 into the quotient.
3) If the result is positive, shift a 1 into the quotient.
4) Repeat this process until the quotient is completely determined.

Conclusion

The Restoring Division Algorithm was chosen because it provides a simple, hardware-efficient, and FPGA-friendly approach to integer division, ensuring an optimal balance between performance and resource utilization in the 3-Stage Pipelined ALU Processor

CONCLUSION

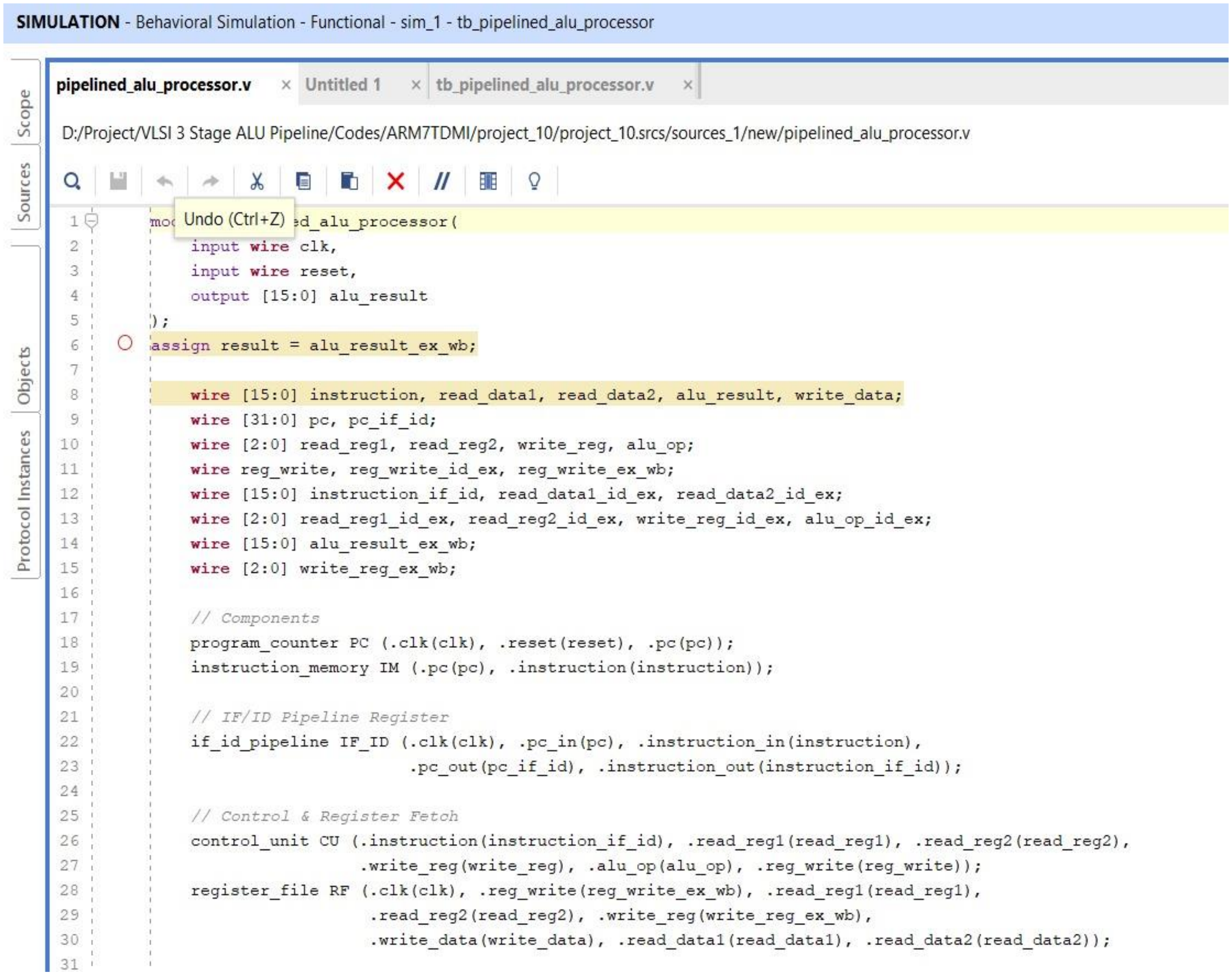
The 3-Stage Pipelined ALU Processor designed in this project successfully demonstrates the principles of pipelined processing for efficient arithmetic and logical operations. By implementing a pipeline architecture, the processor achieves improved instruction throughput while maintaining efficient resource utilization. The integration of fundamental arithmetic operations such as addition, subtraction, multiplication, and division ensures a well-rounded functionality suitable for embedded and FPGA-based applications.

The project has been implemented on the Xilinx Spartan-6 FPGA, which provides a flexible and reconfigurable hardware platform for real-time testing and validation. The Spartan-6 series is well-suited for high-performance arithmetic operations due to its optimized logic resources, DSP slices, and efficient power consumption, making it ideal for FPGA-based ALU design. The use of FPGA technology allows for hardware-level parallelism, resulting in faster execution and improved efficiency compared to traditional sequential processing architectures.

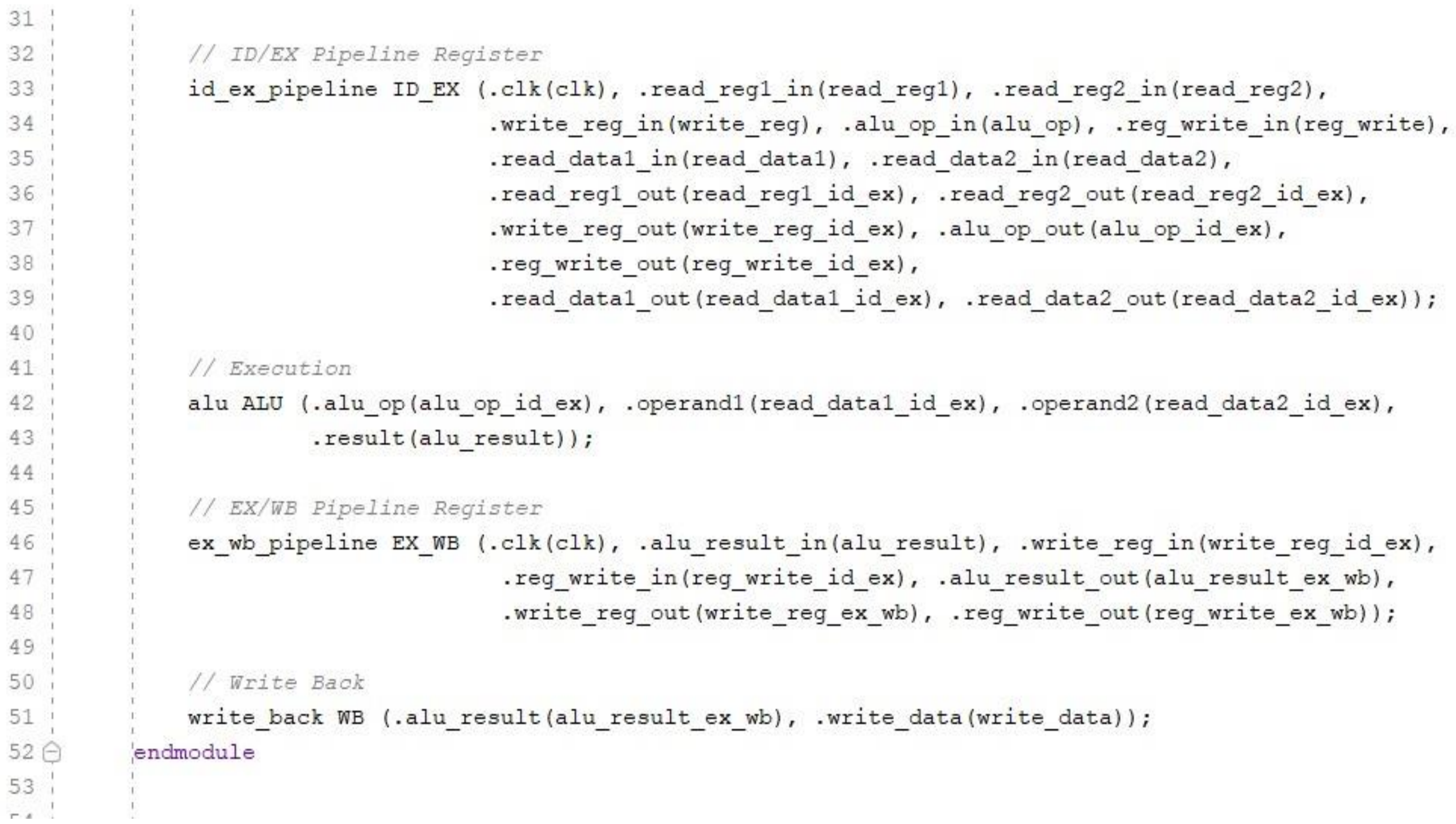
Additionally, the Restoring Division Algorithm was chosen for integer division due to its hardware-friendly nature and ease of implementation on the Spartan-6 FPGA. The FPGA’s configurable logic blocks (CLBs) and DSP slices enable an efficient hardware mapping of the arithmetic operations, ensuring smooth execution of ALU functions.

Overall, the developed pipelined ALU processor on the Xilinx Spartan-6 FPGA serves as a strong foundation for further enhancements in digital arithmetic units. It highlights the advantages of pipelined processing, FPGA-based computation, and efficient arithmetic logic design, making it a valuable contribution to the field of digital system design and FPGA-based computing architectures.

Code :-

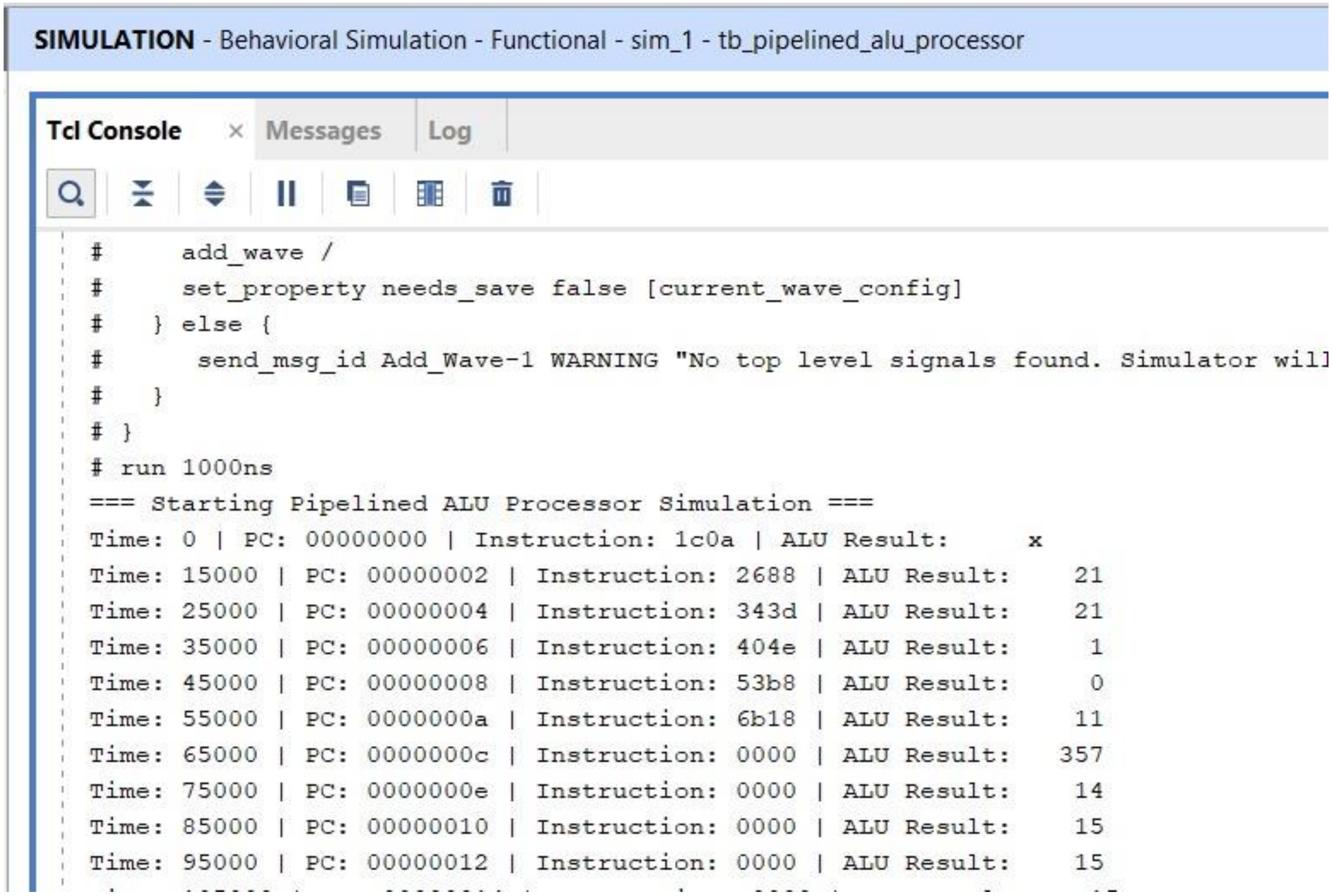

SIMULATION - Behavioral Simulation - Functional - sim_1 - tb_pipelined_alu_processor
pipelined_alu_processor.v
Untitled 1
tb_pipelined_alu_processor.v
D:/Project/VLSI 3 Stage ALU Pipeline/Codes/ARM7TDMI/project_10/project_10.srcs/sources_1/new/pipelined_alu_processor.v
Scope
Sources
Objects
Protocol Instances
Undo (Ctrl+Z)

```
moc Undo (Ctrl+Z) ed_alu_processor(
    input wire clk,
    input wire reset,
    output [15:0] alu_result
);
assign result = alu_result_ex_wb;

    wire [15:0] instruction, read_data1, read_data2, alu_result, write_data;
    wire [31:0] pc, pc_if_id;
    wire [2:0] read_reg1, read_reg2, write_reg, alu_op;
    wire reg_write, reg_write_id_ex, reg_write_ex_wb;
    wire [15:0] instruction_if_id, read_data1_id_ex, read_data2_id_ex;
    wire [2:0] read_reg1_id_ex, read_reg2_id_ex, write_reg_id_ex, alu_op_id_ex;
    wire [15:0] alu_result_ex_wb;
    wire [2:0] write_reg_ex_wb;

    // Components
    program_counter PC (.clk(clk), .reset(reset), .pc(pc));
    instruction_memory IM (.pc(pc), .instruction(instruction));

    // IF/ID Pipeline Register
    if_id_pipeline IF_ID (.clk(clk), .pc_in(pc), .instruction_in(instruction),
                          .pc_out(pc_if_id), .instruction_out(instruction_if_id));

    // Control & Register Fetch
    control_unit CU (.instruction(instruction_if_id), .read_reg1(read_reg1), .read_reg2(read_reg2),
                     .write_reg(write_reg), .alu_op(alu_op), .reg_write(reg_write));
    register_file RF (.clk(clk), .reg_write(reg_write_ex_wb), .read_reg1(read_reg1),
                      .read_reg2(read_reg2), .write_reg(write_reg_ex_wb),
                      .write_data(write_data), .read_data1(read_data1), .read_data2(read_data2));

```



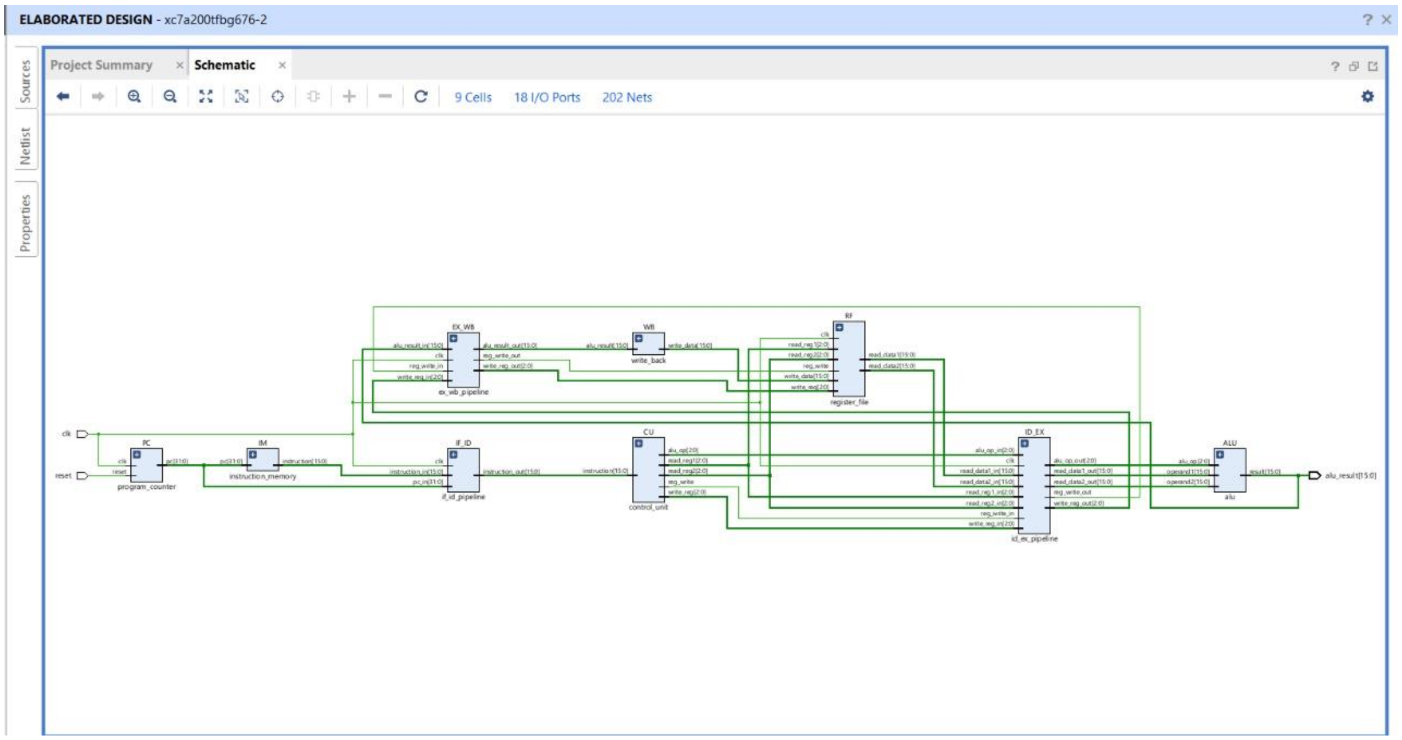
```

    // ID/EX Pipeline Register
    id_ex_pipeline ID_EX (.clk(clk), .read_reg1_in(read_reg1), .read_reg2_in(read_reg2),
                          .write_reg_in(write_reg), .alu_op_in(alu_op), .reg_write_in(reg_write),
                          .read_data1_in(read_data1), .read_data2_in(read_data2),
                          .read_reg1_out(read_reg1_id_ex), .read_reg2_out(read_reg2_id_ex),
                          .write_reg_out(write_reg_id_ex), .alu_op_out(alu_op_id_ex),
                          .reg_write_out(reg_write_id_ex),
                          .read_data1_out(read_data1_id_ex), .read_data2_out(read_data2_id_ex));

    // Execution
    alu ALU (.alu_op(alu_op_id_ex), .operand1(read_data1_id_ex), .operand2(read_data2_id_ex),
             .result(alu_result));

    // EX/WB Pipeline Register
    ex_wb_pipeline EX_WB (.clk(clk), .alu_result_in(alu_result), .write_reg_in(write_reg_id_ex),
                           .reg_write_in(reg_write_id_ex), .alu_result_out(alu_result_ex_wb),
                           .write_reg_out(write_reg_ex_wb), .reg_write_out(reg_write_ex_wb));

    // Write Back
    write_back WB (.alu_result(alu_result_ex_wb), .write_data(write_data));
endmodule

```

OUTPUT:-


SIMULATION - Behavioral Simulation - Functional - sim_1 - tb_pipelined_alu_processor
Tcl Console
Messages
Log

```
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will
#   }
# }
# run 1000ns
=== Starting Pipelined ALU Processor Simulation ===
Time: 0 | PC: 00000000 | Instruction: 1c0a | ALU Result:     x
Time: 15000 | PC: 00000002 | Instruction: 2688 | ALU Result:    21
Time: 25000 | PC: 00000004 | Instruction: 343d | ALU Result:    21
Time: 35000 | PC: 00000006 | Instruction: 404e | ALU Result:     1
Time: 45000 | PC: 00000008 | Instruction: 53b8 | ALU Result:     0
Time: 55000 | PC: 0000000a | Instruction: 6b18 | ALU Result:    11
Time: 65000 | PC: 0000000c | Instruction: 0000 | ALU Result:   357
Time: 75000 | PC: 0000000e | Instruction: 0000 | ALU Result:    14
Time: 85000 | PC: 00000010 | Instruction: 0000 | ALU Result:    15
Time: 95000 | PC: 00000012 | Instruction: 0000 | ALU Result:    15
```

SCHEMATIC:-


ELABORATED DESIGN - xc7a200tfbg676-2
Sources
Netlist
Properties
Project Summary
Schematic
9 Cells
18 I/O Ports
202 Nets
clk
reset
PC
clk
reset
pc[31:0]
program_counter
IM
pc[31:0]
instruction[15:0]
instruction_memory
EX_WB
alu_result_in[15:0]
clk
reg_write_in
write_reg_in[2:0]
alu_result_out[15:0]
reg_write_out
write_reg_out[2:0]
ex_wb_pipeline
WB
alu_result[15:0]
write_data[15:0]
write_back
IF_ID
clk
instruction_in[15:0]
pc_in[31:0]
instruction_out[15:0]
if_id_pipeline
CU
instruction[15:0]
alu_op[2:0]
read_reg1[2:0]
read_reg2[2:0]
reg_write
write_reg[2:0]
control_unit
RF
clk
read_reg1[2:0]
read_reg2[2:0]
reg_write
write_data[15:0]
write_reg[2:0]
read_data1[15:0]
read_data2[15:0]
register_file
ID_EX
alu_op_in[2:0]
clk
read_data1_in[15:0]
read_data2_in[15:0]
read_reg1_in[2:0]
read_reg2_in[2:0]
reg_write_in
write_reg_in[2:0]
alu_op_out[2:0]
read_data1_out[15:0]
read_data2_out[15:0]
reg_write_out
write_reg_out[2:0]
id_ex_pipeline
ALU
alu_op[2:0]
operand1[15:0]
operand2[15:0]
result[15:0]
alu
alu_result[15:0]

REFERENCES

1) https://www.ccbp.in/blog/articles/booth-multiplication-algorithm-in-computer-organization https://www.h-schmidt.net/FloatConverter/IEEE754.html

2) https://www.tutorialspoint.com/vlsi_design/vlsi_design_verilog_introduction.html

3) Design and Simulation of Arithmetic Logic Unit using VerilogHDL

   International Research Journal of Engineering and Technology (IRJET)

   www.irjet.net

APPENDIX

Abbreviations

ASIC	Application Specific Integrated Circuits
FPGA	Field Programmable Gate Array
VLSI	Very Large-Scale Integration
RTL	Register Transfer Level
IC	Integrated Circuits
SoC	System on Chip
OS	Operating System
HDL	Hardware Description Language
FSM	Finite State Machine
DUT	Design Under Test
UUT	Unit Under Test
EDA	Electronic Design Automation
ALU	Arithmetic Logic Unit