*A Thesis Report*

*On*

# Design and Implementation of Single Precision Floating Point Multiplier

Submitted in the partial fulfillment of requirement for the award of the

Degree of

**MASTER OF TECHNOLOGY**

**IN**

**VLSI DESIGN AND CAD**

Submitted by

**Sarbjeet Singh**

**60661020**

Under the guidance of

**Dr. Kulbir Singh**

**Assistant Professor, ECED**

**Electronics and Communication Engineering Department**

**Thapar University**

**Patiala-147004 (INDIA)**

**July, 2008**

# CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Design and Implementation of Single Precision Floating Point Multiplier"**, in partial fulfillment of the requirements for the award of degree of **Master of Technology in VLSI Design and CAD** at **Thapar University**, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Kulbir Singh, Assistant Professor, ECED.**

The matter embodied in this thesis has not been submitted for the award of any other degree of this or any other university.

**(SARBJEET SINGH)**

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

**(Dr. KULBIR SINGH)**
Assistant Professor
ECED, Thapar University,
Patiala.

**(Dr. A. K. CHATTERJEE)**
Professor and Head,
Electronics & Communication Engg. Department,
Thapar University,
Patiala- 147004.

**(Dr. R.K. SHARMA)**
Dean of Academic Affairs,
Thapar University
Patiala -147004

# ACKNOWLEDGEMENT

I would like to thank my thesis advisor, **Dr Kulbir Singh, Assistant Professor,** Electronics and Communication Engineering Department at Thapar University, Patiala to provide me the opportunity to work on this thesis, for his guidance, encouragement, support and confidence in me through the course of my studies. Without his motivating discussions and unwavering desire for achieving high research standards, this work would not have been possible. His flawless & forthright suggestions blended with an innate intelligent application have crowned my task with success.

The constant guidance and encouragement received from **Mrs. Alpana Aggrawal**, PG Coordinator, Department of Electronics & Communication Engineering, is acknowledged with reverential thanks.

I am highly obliged to **Prof. A.K.Chatterjee**, H.O.D Electronics and Communication Engineering Department, Thapar University, Patiala for allowing me to carry out my thesis work in this University.I would also like to offer my sincere thanks to all faculty, teaching and non-teaching, of Electronics & Communication Engg. Deptt. (ECED) and staff of central library TU, Patiala for their assistance.

In the end, I would like to thank my parents and God, without their moral support and blessing which helped me completing this thesis successfully.

Sarbjeet Singh

SARBJEET SINGH

Regd. No: 60661020

# ABSTRACT

Floating Point Arithmetic is extensively used in the field of medical imaging, biometrics, motion capture and audio applications, including broadcast, conferencing, musical instruments and professional audio. Many of these applications need to solve sparse linear systems that use fair amounts of matrix multiplication.

The objective of this thesis is to design and implement single precision floating point cores for multiplication. The multiplier conforms to the IEEE 754 standard for single precision. The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely used standard for floating-point computation, and is followed by many CPU and FPU implementations. The standard defines formats for representing floating-point numbers (including negative zero and denormal numbers) and special values (infinities and NaNs) together with a set of floating-point operations that operate on these values. It also specifies four rounding modes and five exceptions. Rounding modes for this multiplier are round to nearest even, round to zero and round to infinity. Exceptions used in this multiplier are invalid operation, inexact, underflow, overflow, infinity, etc.

The design is implemented on ModelSim SE and has been synthesis and simulated on same tool. The thesis pays significant attention to the analysis of the multiplier cores in terms of pipelining and area so as to maximize throughput. In order to perform the floating point multiplication, a simple algorithm is realized. The exponent are added first and then subtracted from 127. The significands are then multiplied and result is determined. The result is then normalized. Any exceptions (like invalid, inexact, zero, infinity, etc) are checked. This multiplier has five stages which are pre normalization, significant multiply, post normalization, except and floating point unit. This multiplier performs 32 bit multiplication of the floating point numbers. Exception and rounding are also considered in this multiplier.

# LIST OF FIGURES

# LIST OF TABLES

# TABLE OF CONTENT

# CHAPTER 1
# INTRODUCTION

## 1.1 General

Many people consider floating-point arithmetic an esoteric subject. This is rather surprising because floating-point is ubiquitous in computer systems. Almost every language has a floating-point data type; computers from PC's to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. There are some aspects of floating point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with numerous examples of how computer builders can better support floating-point.[1]

Every computer has a floating point processor or a dedicated accelerator that fulfills the requirements of precision using detailed floating point arithmetic. The main applications of floating points today are in the field of medical imaging, biometrics, motion capture and audio applications, including broadcast, conferencing, musical instruments and professional audio. Their importance can be hardly over emphasized because the performances of computers that handle such applications are measured in terms of the number of floating point operations they perform per second. [2]

### 1.1.2 Idea of "Floats"

Decimal numbers are also called Floating Points because a single number can be represented with one or more significant digits depending on the position of the decimal point. Since the point floats between the mass of digits that represent the number such numbers are termed Floating Point Numbers. Floating point formats and number representations are discussed in detail in subsequent chapters.

## 1.2 Motivation

Constraints in representation of mathematical values using existing precision bring about the necessity for cores that can manipulate single precision floating point numbers. The single precision cores for multiplication discussed in this thesis.

Implementation of 32-bit multiplier is an important part of this thesis. The implementation uses ModelSim SE and same for simulation of VHDL source code, synthesis.

## 1.3 Floating Point Arithmetic

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely-used standard for floating-point computation, and is followed by many CPU and FPU implementations.[1] The standard defines formats for representing floating-point number (including ±zero and denormals) and special values (infinities and NaNs) together with a set of floating-point operations that operate on these values. It also specifies four rounding modes and five exceptions.

IEEE 754 specifies four formats for representing floating-point values: single-precision (32-bit), double-precision (64-bit), single-extended precision (≥ 43-bit, not commonly used) and double-extended precision (≥ 79-bit, usually implemented with 80 bits). Many languages specify that IEEE formats and arithmetic be implemented, although sometimes it is optional. For example, the C programming language, which pre-dated IEEE 754, now allows but does not require IEEE arithmetic (the C `float` typically is used for IEEE single-precision and `double` uses IEEE double-precision).[18]

## 1.4 Floating Point Multiplication

Given two FP numbers n1 and n2, the product of both, denoted as n, can be expressed as:

$$n = n1 \times n2$$
$$= (-1)^{S1} \cdot p1 \cdot 2^{E1} \times (-1)^{S2} \cdot p2 \cdot 2^{E2}$$

$$= (-1)^{S1+S2} \cdot (p1 \cdot p2) \cdot 2^{E1+E2}$$

This means that the result of multiplying two FP numbers can be described as multiplying their significands and adding their exponents.[14]. The resultant sign S is S1 +S2, the resultant significand p is the adjusted product of p1 ·p2 and the resultant exponent E is the adjusted E1+E2+bias. In order to perform floating-point multiplication, a simple algorithm is realized:

- Add the exponents and subtract 127.
- Multiply the mantissas and determine the sign of the result.
- Normalize the resulting value, if necessary.

## 1.5 ModelSim Overview

ModelSim is a verification and simulation tool for VHDL, Verilog, SystemVerilog, SystemC, and mixed-language designs. ModelSim VHDL implements the VHDL language as defined by IEEE Standards 1076-1987, 1076-1993, and 1076-2002. ModelSim also supports the 1164-1993 Standard Multivalue Logic System for VHDL Interoperability, and the 1076.2-1996 Standard VHDL Mathematical Packages standards. Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with the 1076 specifications.

### 1.5.1 Creating the working library

In ModelSim, all designs are compiled into a library. We start a new simulation in ModelSim by creating a working library called "work". "Work" is the library name used by the compiler as the default destination for compiled design units.

### 1.5.2 Compiling the design

Before the simulate a design, we must first create a library and compile the source code into that library.

1. Create a new directory and copy the design files for this lesson into it. Start by creating a new directory for this exercise.
2. Start ModelSim if necessary.
    - Type **vsim** at a UNIX shell prompt or use the ModelSim icon in Windows. Upon opening ModelSim for the first time, we will see the Welcome to ModelSim dialog. Click **Close**.
    - Select **File > Change Directory** and change to the directory you created in step 1.

3. Create the working library.

    - Select **File > New > Library**.This opens a dialog where you specify physical and logical names for the library. We can create a new library or map to an existing library. We'll be doing the former.
    - Type **work** in the Library Name field if it isn't entered automatically.Click **OK**.

## 1.5.3 Loading the design into the simulator

Load the test_design module into the simulator. Double-click test_design in the Main window Workspace to load the design. It can also load the design by selecting **Simulate > Start Simulation** in the menu bar. This opens the Start Simulation dialog. With the Design tab selected, click the '+' sign next to the work library to see the counter and test_design modules. Select the test_design module and click OK. When the design is loaded, you will see a new tab in the Workspace named *sim* that displays the hierarchical structure of the design. It can navigate within the hierarchy by clicking on any line with a'+' (expand) or '-' (contract) icon. It will also see a tab named *Files* that displays all files included in the design. The Objects pane opens by default when a design is loaded. It shows the names and current values of data objects in the current region (selected in the Workspace). Data objects include signals, nets, registers, constants and variables not declared in a process, generics, parameters, and member data variables of a SystemC module.

### 1.5.4 Running the simulation

With the design compiled, you invoke the simulator on a top-level module (Verilog) or a configuration or entity/architecture pair (VHDL). Assuming the design loads successfully, the simulation time is set to zero, and we enter a run command to begin simulation. Now we will run the simulation.

1. Set the graphic user interface to view the Wave debugging pane in the Main window. Enter **view wave** at the command line. This opens one of several panes available for debugging. To see a list of the other panes, select **View > Debug Windows** from the menu bar. Most debugging windows will open as panes within the Main window. The Dataflow window will open as a separate window. We may need to move or resize the windows to our liking. Panes within the Main window can zoom to occupy the entire Main window or undocked to stand alone.

2 Add signals to the Wave window.

- In the Workspace pane, select the **sim** tab.
- Right-click test_counter to open a popup context menu.
- Select **Add > Add to Wave**.

Three signals are added to the Wave window.

3 Run the simulation.

- Click the Run icon in the Main or Wave window toolbar. The simulation runs for 100 ns (the default simulation length) and waves are drawn in the Wave window.
- Type **run 500** at the VSIM> prompt in the Main window. The simulation advances another 500 ns for a total of 600 ns.
- Click the Run -All icon on the Main or Wave window toolbar. The simulation continues running until you execute a break command or it hits a statement in your code (e.g., a Verilog $stop statement) that halts the simulation.
- Click the Break icon. The simulation stops running.

### 1.5.6 Debugging results

If we don't get the results we expect, then we can use ModelSim's robust debugging environment to track down the cause of the problem.

## 1.6 Objective of Thesis

Based on the discussion above the thesis has following objectives

- To study single precision floating point arithmetic.
- To study ModelSim Tool to implement the single precision floating point multiplier.
- To design and implement the single precision floating point multiplier.

## 1.7 Organization of Thesis

In this thesis, we begin with background on floating point representation continue with a discussion of the IEEE floating point standards and conludes with numerous examples of how computer builders can better support floating point. Chapter two describes floating point arithmetic in which floating point formats, floating point numbers, operation on floating point numbers are covered. This thesis is based on single precision floating point multiplier so that single precision number representation is described in chapter three. In this chapter single precision floating point numbers, denormals, exception, rounding modes etc all are covered. In Chapter 4, ModelSim SE tool is discussed which is used for verification and design of single precision floating point multiplier. Chapter five discusses design and implementation of the single precision floating point multiplier. This chapter also describes the states of the floating point multiplier. In chapter six, verification of the results of the single precision floating point multiplier is discussed. The thesis concludes in Chapter seven and also discusses future scope of work.

# CHAPTER 2

# FLOATING POINT ARITHMETIC

## 2.1 Introduction

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rational, and represent every number as the ratio of two integers. [18]

Floating-point representation - the most common solution - basically represents reals in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as $1.23456 \times 10^2$. In hexadecimal, the number 123.abc might be represented as $1.23abc \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

## 2.2 IEEE 754 Floating Point Standard

IEEE 754 floating point standard is the most common representation today for real numbers on computers. The IEEE (Institute of Electrical and Electronics Engineers) has produced a Standard to define floating-point representation and arithmetic. Although

there are other representations, it is the most common representation used for floating point numbers. The standard brought out by the IEEE come to be known as IEEE 754. The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely-used standard for floating point computation, and is followed by many CPU and FPU implementations.[1] The standard defines formats for representing floating-point numbers including negative numbers and denormal numbers special values i.e. infinities and NAN's together with a set of floating-point operations that operate on these values. It also specifies four rounding modes which are round to zero, round to nearest, round to infinity and round to even and five exceptions including when the exceptions occur, and what happens when they do occur. Dealing with fixed-point arithmetic will limit the usability of a processor. If operations on numbers with fractions (e.g. 10.2445), very small numbers (e.g. 0.000004), or very large numbers (e.g. $42.243x10^5$) are required, then a different one representation is in order is the floating-point arithmetic.[14] The floating point is utilized as the binary point is not fixed, as is the case in integer (fixed-point) arithmetic. In order to get some of the terminology out of the way, let us discuss a simple floating-point number, such as $-2.42x10^3$. The '-' symbol indicates the sign component of the number, while the '242' indicate the significant digits component of the number, and finally the '3' indicates the scale factor component of the number. It is interesting to note that the string of significant digits is technically termed the *mantissa* of the number, while the scale factor is appropriately called the *exponent* of the number. The general form of the representation is the following:

$$\mathbf{(-1)}^{S} * M * 2^{E} \qquad (2.1)$$

*where*

**S** represents the sign bit,

**M** represents the mantissa and

**E** represents the exponent

## 2.3 Scientific Notation of Floating Point Numbers

Floating-point notation can be used conveniently to represent both large as well as small rational and mixed numbers. This makes the process of arithmetic operations on these numbers relatively much easier. Floating-point representation greatly increases the range of numbers, from the smallest to the largest, that can be represented using a given number of digits. Floating-point numbers are in general expressed in the form

$$N = m \times b^e \tag{2.2}$$

where m is the fractional part, called the significand or mantissa, e is the integer part, called the *exponent*, and b is the *base* of the number system or numeration. Fractional part m is a p-digit number of the form (±d.*dddd _ _ _ dd*), with each digit d being an integer between 0 and b – 1 inclusive. If the leading digit of m is nonzero, then the number is said to be normalized.

Equation (2.2) in the case of decimal, hexadecimal and binary number systems will be written as follows:

Decimal system

$$N = m \times 10^e \tag{2.3}$$

Hexadecimal system

$$N = m \times 16^e \tag{2.4}$$

Binary system

$$N = m \times 2^e \tag{2.5}$$

For example, decimal numbers 0.0003754 and 3754 will be represented in floating-point notation as $3.754 \times 10^{-4}$ and $3.754 \times 10^3$ respectively. A hex number 257.ABF will be represented as $2.57ABF \times 16^2$. In the case of normalized binary numbers, the leading digit, which is the most significant bit, is always ‘1’ and thus does not need to be stored explicitly. Also, while expressing a given mixed binary number as a floating-point number, the radix point is so shifted as to have the most significant bit immediately to the

right of the radix point as a '1'. Both the mantissa and the exponent can have a positive or a negative value. The mixed binary number $(110.1011)_2$ will be represented in floating-point notation as $.1101011 \times 2^3 = .1101011e^{+0011}$. Here, .1101011 is the mantissa and $e^{+0011}$ implies that the exponent is +3. As another example, $(0.000111)_2$ will be written as $.111e^{-0011}$, with .111 being the mantissa and $e^{-0011}$ implying an exponent of −3. Also, $(-0.00000101)_2$ may be written as $-.101 \times 2^{-5} = -.101e^{-0101}$, where −.101 is the mantissa and $e^{-0101}$ indicates an exponent of −5. If we wanted to represent the mantissas using eight bits, then .1101011 and .111 would be represented as .11010110 and .11100000.

The standard specifies:

- Basic and extended floating-point number formats
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversions between integer and floating-point formats
- Conversions between different floating-point formats
- Conversions between basic format floating-point numbers and decimal strings
- Floating-point exceptions and their handling, including non numbers (Nan's)

When it comes to their precision and width in bits, the standard defines two groups: **basic-** and **extended format**. The basic format is further divided into **single-precision** format with 32-bits wide, and **double-precision** format with 64-bits wide.[12] The three basic components are the sign, exponent, and mantissa.[4]

## 2.4 IEEE 754 Floating Point Formats

IEEE 754 specifies four formats for representing floating-point values:

1. Single-precision (32-bit)
2. Double-precision (64-bit)
3. Single-extended precision (≥ 43-bit, not commonly used)
4. Double-extended precision (≥ 79-bit, usually implemented with 80 bits).

Table 2.1 shows the basic constituent parts of the single and double-precision formats. As shown in the table, the floating-point numbers, as represented using these formats, have three basic components including the sign, the exponent and the mantissa. A '0' denotes a positive number and a '1' denotes a negative number. The n-bit exponent field needs to represent both positive and negative exponent values. To achieve this, a bias equal to $2^{n-1}-1$ is added to the actual exponent in order to obtain the stored exponent. This equals 127 for an eight-bit exponent of the single-precision format and 1023 for an 11-bit exponent of the double-precision format. The addition of bias allows the use of an exponent in the range from −127 to +128, corresponding to a range of 0–255 in the first case, and in the range from −1023 to +1024, corresponding to a range of 0–2047 in the second case. A negative exponent is always represented in 2's complement form. The single-precision format offers a range from $2^{-127}$ to $2^{+127}$, which is equivalent to $10^{-38}$ to $10^{+38}$. The figures are $2^{-1023}$ to $2^{+1023}$, which is equivalent to $10^{-308}$ to $10^{+308}$ in the case of the double-precision format.[5]

**Table 2.1:** Characteristic parameters of IEEE-754 formats

| Precision | Sign (bits) | Exponent (bits) | Mantissa (bits) | Total length (bits) | Decimal digits of precision |
|---|---|---|---|---|---|
| Single | 1 | 8 | 23 | 32 | > 6 |
| Single-extended | 1 | ≥ 11 | ≥ 32 | ≥ 44 | > 9 |
| Double | 1 | 11 | 52 | 64 | > 15 |
| Double-extended | 1 | ≥ 15 | ≥ 64 | ≥ 80 | > 19 |

### 2.4.1 Single Precision Floating Point Numbers

The single-precision number is 32 bit wide. The single-precision number has three main fields that are sign, exponent and mantissa. The 24-bit mantissa (the leading one is implicit) can approximately represent a 7-digit decimal number, while an 8-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. Thus, a total of 32

bits is needed for single-precision number representation. To achieve a bias equal to $2^{n-1}-1$ is added to the actual exponent in order to obtain the stored exponent. This equals 127 for an eight-bit exponent of the single-precision format. The addition of bias allows the use of an exponent in the range from −127 to +128, corresponding to a range of 0–255 for single precision number. The single-precision format offers a range from $2^{-127}$ to $2^{+127}$, which is equivalent to $10^{-38}$ to $10^{+38}$

**Sign**: 1-bit wide and used to denote the sign of the number i.e. 0 indicate positive number and 1 represent negative number.

**Exponent:** 8-bit wide and signed exponent in excess-127 representation.

**Mantissa:** 23-bit wide and fractional component.

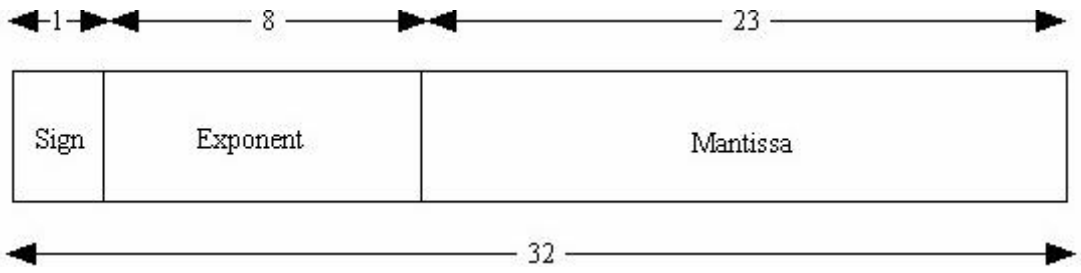


**Figure 2.1**: Single-precision floating-point number representation

The excess-127 representation mentioned when discussing the exponent portion above, is utilized to efficiently compare the relative sizes of two floating point numbers. Instead of storing the exponent (*E*) as a signed number, we store its unsigned integer representation (*E' = E* +127). This gives us a range for *E'* of 0 <= *E'* <= 255. While the 0 and 255 end values are used to represent special numbers (exact 0, infinity and denormal numbers), the operating range of *E0* becomes 1 <= *E'*<=· 254, thus, limiting the range of *E* to -126 <= *E* <= 127. In double-precision numbers, an excess-1023 representation is utilized.

## 2.4.2 Double Precision Floating Point Numbers

Figure shows the double precision floating point number representation. The double precision number is 64 bit wide. The double precision number has three main fields which are sign, exponent and mantissa. The 52-bit mantissa (the leading one is implicit), while an 11-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. Thus, a total of 64 bits is needed for single-precision number representation. To achieve a bias equal to $2^{n-1}-1$ is added to the actual exponent in order to obtain the stored exponent. This equal 1023 for an 11-bit exponent of the double-precision format. The addition of bias allows the use of an exponent in the range from −1023 to +1024, corresponding to a range of 0–2047 for double precision number. The double precision format offers a range from $2^{-1023}$ to $2^{+1023}$, which is equivalent to $10^{-308}$ to $10^{+308}$

**Sign**: 1-bit wide and used to denote the sign of the number i.e. 0 indicate positive number and 1 represent negative number.
**Exponent:** 11-bit wide and signed exponent in excess- 1023 representation.

**Mantissa:** 52-bit wide and fractional component.

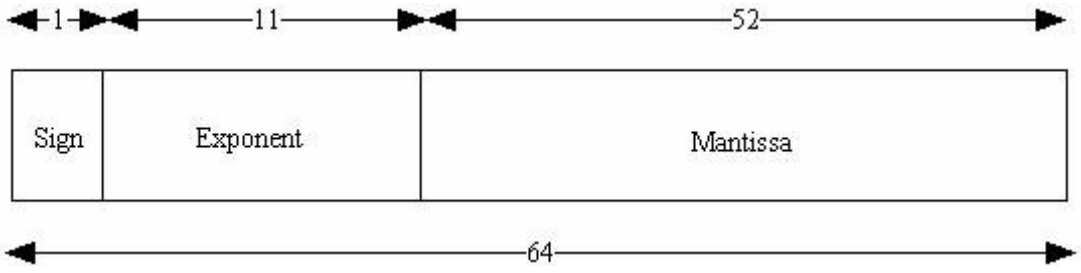


**Figure 2.2**: Double-precision floating-point number representation

In IEEE standard is that the mantissa component is always normalized. The latter implies that the decimal point is placed to the right of the first (nonzero) significant digit. Hence, the 23 bits stored in the M field actually represent the fractional part of the mantissa, that is, the bits to the right of the binary point. As aforementioned, the most significant bit of

the mantissa is always equal to 1, due to binary normalization. Revisiting our original formula representation, let us now normalize the 32-bit single-precision floating-point number representation as:

$$(-1)\ S * (1 + M) * 2^{E-127} \quad (2.6)$$

S represents the sign (1-bit),
M represents the mantissa (23-bit) and
E represents the exponent (8-bit)

## 2.5 Number Representation using Single Precision Format

Let us try and represent the decimal number $(-0.75)_{10}$ in IEEE floating-point format. First of, we notice that $(-0.75)_{10} = (-3/4)_{10}$ .
In binary notation, we have $(-0.11)_2 = (-0.11)_2 \times 20 = (-1.1)_2 \times 2^{-1}$.
Referring to equation (2), we can represent our number as:
$(-1)1 * (1 + .10000000000000000000000_2) * 2^{126-127}$ Thus, our single-precision representation of the number is given as $(10111111010000000000000000000000)_2$,where The sign bit is $(1)_2$,for negative numbers; The exponent is $(01111110)_2$, to represent $(126)_{10}$, The mantissa is $(10000000000000000000000)_2$, to represent the fractional part $(.1)_2$.

## 2.6 Ranges of Floating-Point Numbers

Let's consider single-precision floats for a second. Note that we're taking essentially a 32-bit number and re-jiggering the fields to cover a much broader range. Something has to give, and it's precision. For example, regular 32-bit integers, with all precision centered on zero, can precisely store integers with 32-bits of resolution. Single-precision floating-point, on the other hand, is unable to match this resolution with its 24 bits. It does, however, approximate this value by effectively truncating from the lower end. For example:

11110000 11001100 10101010 00001111 -- 32-bit integer

= +1.1110000 11001100 10101010 x $2^{31}$ -- Single-Precision Float

= 11110000 11001100 10101010 00000000 -- Corresponding Value

This approximates the 32-bit value, but doesn't yield an exact representation. On the other hand, besides the ability to represent fractional components (which integers lack completely), the floating-point value can represent numbers around $2^{127}$, compared to 32-bit integers maximum value around $2^{32}$. [15]

The range of positive floating point numbers can be split into normalized numbers which preserve the full precision of the mantissa, and denormalized numbers which use only a portion of the fractions's precision.

**Table 2.2:** Range of single precision and double precision float numbers

| | **Denormalized** | **Normalized** | **Approximate Decimal** |
|---|---|---|---|
| **Single Precision** | $\pm 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$ | $\pm 2^{-126}$ to $(2-2^{-23})\times 2^{127}$ | $\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$ |
| **Double Precision** | $\pm 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$ | $\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$ | $\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$ |

Since the sign of floating point numbers is given by a special leading bit, the range for negative numbers is given by the negation of the above values.

There are five distinct numerical ranges that single-precision floating-point numbers are not able to represent:

- Negative numbers less than $-(2-2^{-23}) \times 2^{127}$ (negative overflow)
- Negative numbers greater than $-2^{-149}$ (negative underflow)

- Zero
- Positive numbers less than $2^{-149}$ (positive underflow)
- Positive numbers greater than $(2-2^{-23}) \times 2^{127}$ (positive overflow)

Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers. Underflow is a less serious problem because is just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

Table 2.3 gives the effective range (excluding infinite values) of IEEE floating-point numbers.

**Table 2.3:** Effective Range of single precision and double precision float numbers

| | **Binary** | **Decimal** |
|---|---|---|
| **Single** | $\pm (2-2^{-23}) \times 2^{127}$ | $\sim \pm 10^{38}$ |
| **Double** | $\pm (2-2^{-52}) \times 2^{1023}$ | $\sim \pm 10^{308}$ |

Note that the extreme values occur (regardless of sign) when the exponent is at the maximum value for finite numbers ($2^{127}$ for single-precision, $2^{1023}$ for double), and the mantissa is filled with 1s (including the normalizing 1 bit).

## 2.7 Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

### 2.7.1 Zero

As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

### 2.7.2 Denormalized

If the exponent have all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a *denormalized* number, which does *not* have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where $s$ is the sign bit and $f$ is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.

### 2.7.3 Infinity

The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

### 2.7.4 Not a Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN). They have the following format, where s is the sign bit:

QNaN = s 11111111 10000000000000000000000
SNaN = s 11111111 00000000000000000000001

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined. An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage. Semantically, QNaN's denote indeterminate operations, while SNaN's denote invalid operations.

### 2.7.5 Special Operations

Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Other operations are as follows:

**Table 2.4:** Table represents the special result for some special inputs.

| Operation | Result |
|---|---|
| n ÷ ±Infinity | 0 |
| ±Infinity × ±Infinity | ±Infinity |
| ±nonzero ÷ 0 | ±Infinity |
| Infinity + Infinity | Infinity |
| ±0 ÷ ±0 | *NaN* |
| Infinity - Infinity | *NaN* |
| ±Infinity ÷ ±Infinity | *NaN* |
| ±Infinity × 0 | *NaN* |

# CHAPTER 3

# SINGLE PRECISION NUMBER REPRESENTATION

## 3.1 A Simple Example of Single Precision Number

Let us try and represent the decimal number $(-0.75)_{10}$ in IEEE floating-point format. First of, we notice that $(-0.75)_{10} = (-3/4)_{10}$ .
In binary notation, we have $(-0.11)_2 = (-0.11)_2 \times 20 = (-1.1)_2 \times 2^{-1}$.

Referring to equation (2), we can represent our number as:
$(-1)1 * (1 + .10000000000000000000000_2) * 2^{126-127}$ Thus, our single-precision representation of the number is given as $(10111111010000000000000000000000)_2$,where The sign bit is $(1)_2$, for negative numbers; The exponent is $(01111110)_2$, to represent $(126)_{10}$, The mantissa is $(10000000000000000000000)_2$, to represent the fractional part $(.1)_2$.

Step-by-step transformation of $(23)_{10}$ into an equivalent floating-point number in single-precision IEEE format is as follows:

- $(23)_{10} = (10111)_2 = 1.0111e^{+0100}$.
- The mantissa = 0111000 00000000 00000000.
- The exponent = 00000100.
- The biased exponent = 00000100 + 01111111 = 10000011.
- The sign of the mantissa = 0.
- Therefore $(+23)_{10}$ = 01000001 10111000 00000000 00000000.
- Also, $(-23)_{10}$= 11000001 10111000 00000000 00000000

## 3.2 Storage Layout

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit. The exponent base (2) is implicit and need not be stored. [13]

The following figure shows the layout for single (32-bit) and doubles (64-bit) precision floating-point values. The number of bits for each field is shown (bit ranges are in square brackets)

**Table 3.1:** Range of single and double floating point number

| | **Sign** | **Exponent** | **Fraction** | **Bias** |
|---|---|---|---|---|
| **Single Precision** | 1 [31] | 8 [30-23] | 23 [22-00] | 127 |
| **Double Precision** | 1 [63] | 11 [62-52] | 52 [51-00] | 1023 |

### 3.2.1 Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

### 3.2.2 Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a *bias* is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. The exponent is biased by $(2^{e-1}) - 1$, where $e$ is the number of bits used for the exponent field (e.g. if e=8, then $(2^{8-1}) - 1 = 128 - 1 = 127$). Biasing is done because exponents have to be signed values in order to be able to represent tiny and huge values, but2's complement, the usual representation for signed values, would make comparison harder.

To solve this, the exponent is biased before being stored by adjusting its value to put it within an unsigned range suitable for comparison.For double precision, the exponent field is 11 bits, and has a bias of 1023.

### 3.2.3 Mantissa

The *mantissa*, also known as the *significant*, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

In order to maximize the quantity of represent able numbers, floating-point numbers are typically stored in *normalized* form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as $5.0 \times 10^0$.

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

- The sign bit is 0 for positive, 1 for negative.
- The exponent's base is two.
- The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
- The first bit of the mantissa is typically assumed to be 1.*f*, where *f* is the field of fraction bits.

The storage layout for single precision floating point number is shown below:

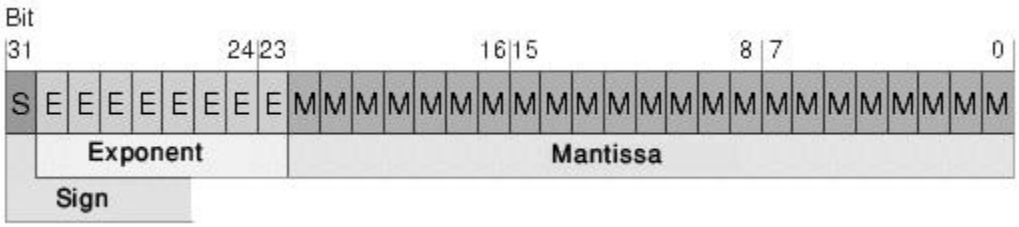


**Figure 3.1:** Storage Layout for Single Precision Floating Point Number

The number represented by the single-precision format is:

Value $= (-1)^{s} 2^{e} \times 1.f$ (normalized) when E > 0 else

$= (-1)^{s} 2^{-126} \times 0.f$ (denormalized)

where

$f = (b_{23}^{-1} + b_{22}^{-2} + b_i^n + \ldots + b_0^{-23})$ where $b_i^n = 1$ or 0

s = sign (0 is positive; 1 is negative)

E = biased exponent; $E_{max} = 255$ , $E_{min} = 0$. E=255 and E=0 are used to represent special values.

e =unbiased exponent; e = E – 127(bias)

A bias of 127 is added to the actual exponent to make negative exponents possible without using a sign bit. So for example if the value 100 is stored in the exponent placeholder, the exponent is actually -27 (100 – 127). Not the whole range of E is used to represent numbers. As we may have seen from the above formula, the leading fraction bit before the decimal point is actually implicit (not given) and can be 1 or 0 depending on the exponent and therefore saving one bit. Below is a table with the corresponding values for a given representation to help better understand what was explained above.

**Table 3.2:** Corresponding Value for Single Precision Floating Point Number

| Sign | Exponent (e) | Fraction | Value |
|---|---|---|---|
| 0 | 00000000 | 00000000000000000000000 | +0<br>(Positive Zero) |
| 1 | 00000000 | 00000000000000000000000 | -0<br>(Negative Zero) |
| 1 | 00000000 | 10000000000000000000000 | $-2^{0-127} \times 0.2^{-1}$<br>$= 2^{0-127} \times 0.5$ |
| 0 | 00000000 | 00000000000000000000001 | $+2^{0-127} \times 0.2^{-23}$<br>(Smallest Value) |
| 0 | 00000001 | 01000000000000000000000 | $+2^{1-127} \times 1.2^{-2}$<br>$= 2^{1-127} \times 1.25$ |
| 0 | 10000001 | 00000000000000000000000 | $2^{129-127} \times 1.0$<br>$= 4$ |
| 0 | 11111111 | 00000000000000000000000 | + Infinity |
| 1 | 11111111 | 00000000000000000000000 | - Infinity |
| 0 | 11111111 | 10000000000000000000000 | Not a Number<br>(NaN) |
| 1 | 11111111 | 10000100001000000001100 | Not a Number<br>(NaN) |

In this example we determine whether single-precision number does the following 32-bit word represent.

1 10000001 01000000000000000000000

Considered as an unsigned number, the exponent field is 129, making the value of the exponent 129 −127 = 2. The fraction part is .012 = .25, making the significand 1.25. Thus, this bit pattern represents the number −1.25 *22 = −5.

The fractional part of a floating-point number (.25 in the example above) must not be confused with the significand, which is 1 plus the fractional part. The leading 1 in the significand 1 f does not appear in the representation; that is, the leading bit is implicit. When performing arithmetic on IEEE format numbers, the fraction part is usually unpacked, which is to say the implicit 1 is made explicit.

It shows the exponents for single precision to range from –126 to 127; accordingly, the biased exponents range from 1 to 254. The biased exponents of 0 and 255 are used to represent special values. When the biased exponent is 255, a zero fraction field represents infinity, and a nonzero fraction field represents a NaN. Thus, there is an entire family of NaNs. When the biased exponent and the fraction field are 0, then the number represented is 0. Because of the implicit leading 1, ordinary numbers always have a significand greater than or equal to 1. Thus, a special convention such as this is required to represent 0. Denormalized numbers are implemented by having a word with a zero exponent field represent the number 0. $f$*$2^{E\min}$.

**Table 3.3:** Representations of special values.

| Exponent | Fraction | Represents |
|---|---|---|
| $e = E_{\min} - 1$ | $f = 0$ | $\pm 0$ |
| $e = E_{\min} - 1$ | $f \neq 0$ | $0.f \times 2^{E_{\min}}$ |
| $E_{\min} \leq e \leq E_{\max}$ | — | $1.f \times 2^{e}$ |
| $e = E_{\max} + 1$ | $f = 0$ | $\pm \infty$ |
| $e = E_{\max} + 1$ | $f \neq 0$ | NaN |

The primary reason why the IEEE standard, like most other floating-point formats, uses biased exponents is that it means nonnegative numbers are ordered in the same way as

integers. That is, the magnitude of floating-point numbers can be compared using an integer comparator. Another (related) advantage is that 0 is represented by a word of all 0's. The downside of biased exponents is that adding them is slightly awkward, because it requires that the bias be subtracted from their sum.

## 3.3 Floating-Point Multiplication

The simplest floating-point operation is multiplication, so we discuss it first. A binary floating-point number *x* is represented as a significand and an exponent,

$$x = s * 2^{e}.$$

The formula

$$(s1 * 2^{e1}) \bullet (s2 * 2^{e2}) = (s1 \bullet s2) * 2^{e1+e2}$$

Shows that a floating-point multiply algorithm has several parts. The first part multiplies the significands using ordinary integer multiplication. Because floating point numbers are stored in sign magnitude form, the multiplier need only deal with unsigned numbers (although we have seen that Booth recoding handles signed two's complement numbers painlessly). The second part rounds the result. If the significands are unsigned *p*-bit numbers (e.g., $p = 24$ for single precision), then the product can have as many as $2p$ bits and must be rounded to a *p*-bit number. The third part computes the new exponent. Because exponents are stored with a bias, this involves subtracting the bias from the sum of the biased exponents. [9]

Let us take one example how does the multiplication of the single-precision numbers proceed in binary. The single precision number are given below:

$$1\ 10000010\ 000\ldots = -1 * 2^{3}$$

$$0\ 10000011\ 000\ldots = 1 * 2^{4}$$

When unpacked, the significands are both 1.0, their product is 1.0, and so the result is of the form

1 ???????? 000. . .

To compute the exponent, use the formula

$$\text{Biased exp}(e1 + e2) = \text{biased exp}(e1) + \text{biased exp}(e2) - \text{bias}$$

The bias is $127 = 01111111_2$, so in two's complement $-127$ is $10000001_2$. Thus the biased exponent of the product is

$$\begin{array}{r} 10000010 \\ 10000011 \\ +\ 10000001 \\ \hline 10000110 \\ \hline \end{array}$$

Since this is 134 decimal, it represents an exponent of $134 - \text{bias} = 134 - 127 = 7$, as expected. The interesting part of floating-point multiplication is rounding. [16]. Since the cases are similar in all bases, the figure uses human-friendly base 10, rather than base 2. There is a straightforward method of handling rounding using the multiplier with an extra sticky bit. If $p$ is the number of bits in the significand, then the A, B, and P registers should be $p$ bits wide. Multiply the two significands to obtain a $2p$-bit product in the (P,A) registers Using base 10 and $p = 3$, parts (a) and (b) illustrate that the result of a multiplication can have either $2p - 1$ or $2p$ digits, and hence the position where a 1 is added when rounding up (just left of the arrow) can vary.

a)

```
  1.23
 *6.78_______
      8.3394    r= 9 >5 so round up rounds to 8.34.
```

b)

```
   2.83
   *4.47
   _______
      12.6501  r=5 and following digit is not equal to zero so round up rounds to
```

$1.27 \text{ X } 10^3$

P and A contain the product, case1 $x_0$=0 shift needed, case2 $x_0$=1 increment exponent.

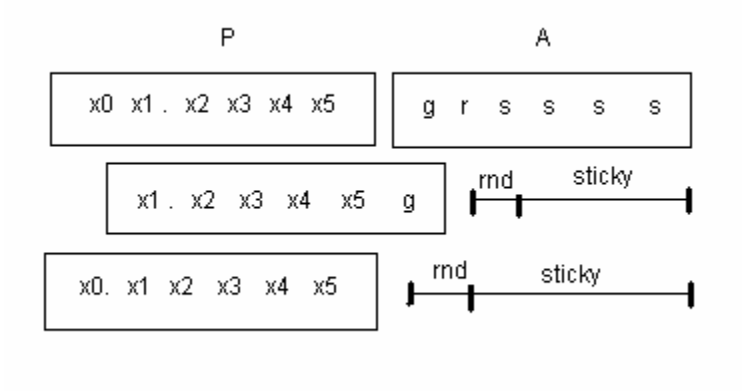


**Figure 3.2**: Rounding method

The top line shows the contents of the P and A registers after multiplying the significands, with p = 6. In case (1), the leading bit is 0, and so the P register must be shifted. In case (2), the leading bit is 1, no shift is required, but both the exponent and the round and sticky bits must be adjusted. The sticky bit is the logical OR of the bits marked s. During the multiplication, the first p −2 times a bit is shifted into the A register, OR it into the sticky bit. This will be used in halfway cases. Let *s* represent the sticky bit, *g* (for guard) the most-significant bit of A, and r (for round) the second most-significant bit of A.

There are two cases:

- The high-order bit of P is 0. Shift P left 1 bit, shifting in the *g* bit from A. Shifting the rest of A is not necessary.
- The high-order bit of P is 1. Set s= s.v.r and r = *g*, and add 1 to the exponent.

Now if $r = 0$, P is the correctly rounded product. If $r = 1$ and $s = 1$, then P + 1 is the product (where by P + 1 we mean adding 1 to the least-significant bit of P). If $r = 1$ and $s = 0$, we are in a halfway case, and round up according to the least significant bit of P. After the multiplication, P = 126 and A = 501, with $g = 5$, $r = 0$, $s = 1$. Since the high-order digit of P is nonzero, case (2) applies and $r := g$, so that $r = 5$, as the arrow indicates in Figure H.9. Since $r = 5$, we could be in a halfway case, but $s = 1$ indicates that the result is in fact slightly over 1/2, so add 1 to P to obtain the correctly rounded product. Note that P is nonnegative, that is, it contains the magnitude of the result.

Here is an example of rounding of number. In binary with $p = 4$, show how the multiplication algorithm computes the product −5 *10 in each of the four rounding modes. n binary, −5 is $-1.0102*2^2$ and 10 = $1.0102*2^3$. Applying the integer multiplication algorithm to the significands gives 011001002, so P = 01102, A = 01002, $g = 0$, $r = 1$, and $s = 0$. The high-order bit of P is 0, so case (1) applies. Thus P becomes $1100_2$ and the result is negative.

| | | |
|---|---|---|
| round to -∞ | $1101_2$ | add 1 since $r.v.s = 1/0 = \text{TRUE}$ |
| round to +∞ | $1100_2$ | |
| round to 0 | $1100_2$ | |
| round to nearest | $1100_2$ | no add since $r \wedge p0 = 1 \wedge 0 = \text{FALSE}$ and $r \wedge s = 1 \wedge 0 = \text{FALSE}$ |

The exponent is 2 + 3 = 5, so the result is $-1.100_2 *2^5 = -48$, except when rounding to −∞, in which case it is $-1.101_2*2^5 = -52$.

Overflow occurs when the rounded result is too large to be represented. In single precision, this occurs when the result has an exponent of 128 or higher. If $e1$ and $e2$ are the two biased exponents, then $1 \leq e_i \leq 254$, and the exponent calculation $e1 + e2 - 127$ gives numbers between 1 + 1 −127 and 254 + 254 −127, or between −125 and 381. This range of numbers can be represented using 9 bits. So one way to detect overflow is to perform the exponent calculations in a 9-bit adder.

## 3.4 Denormals: Some Special Cases

Checking for underflow is somewhat more complex because of denormals.[7]. In single precision, if the result has an exponent less than −126, that does not necessarily indicate underflow, because the result might be a denormal number. For example, the product of $(1*2^{-64})$ with $(1*2^{-65})$ is $1 *2^{-129}$, and −129 is below the legal exponent limit. But this result is a valid denormal number, namely, $0.125 *2^{-126.}$ In general, when the unbiased exponent of a product dips below −126, the resulting product must be shifted right and

the exponent incremented until the exponent reaches −126. If this process causes the entire significand to be shifted out, then underflow has occurred.

**Table 3.4:** Rounding Mode's

| Rounding mode | Sign of result ≥ 0 | Sign of result < 0 |
|---|---|---|
| $-\infty$ | | +1 if $r \vee s$ |
| $+\infty$ | +1 if $r \vee s$ | |
| 0 | | |
| Nearest | +1 if $r \wedge p_0$ or $r \wedge s$ | +1 if $r \wedge p_0$ or $r \wedge s$ |

When one of the operands of a multiplication is denormal, its significand will have leading zeros, and so the product of the significands will also have leading zeros. If the exponent of the product is less than –126, then the result is Denormals, so right-shift and increment the exponent as before. If the exponent is greater than –126, the result may be a normalized number. In this case, left-shift the product (while decrementing the exponent) until either it becomes normalized or the exponent drops to –126.

Denormal numbers present a major stumbling block to implementing floating-point multiplication, because they require performing a variable shift in the multiplier, which wouldn't otherwise be needed. Thus, high-performance, floating-point multipliers often do not handle denormalized numbers, but instead trap, letting software handle them. A few practical codes frequently underflow, even when working properly, and these programs will run quite a bit slower on systems that require denormals to be processed by a trap handler.

Handling of zero operands can be done either testing both operands before beginning the multiplication or testing the product afterward. Once you detect that the result is 0, set the biased exponent to 0. The sign of a product is the XOR of the signs of the operands, even when the result is 0.

## 3.5 Precision of Multiplication

In the integer multiplication, the designers must decide whether to deliver the low-order word of the product or the entire product. A similar issue arises in floating-point multiplication, where the exact product can be rounded to the precision of the operands or to the next higher precision. In the case of integer multiplication, none of the standard high-level languages contains a construct that would generate a “single times single gets double” instruction. The situation is different for floating point. Many languages allow assigning the product of two single-precision variables to a double-precision one and numerical algorithms can also exploit the construction. The best-known case is using iterative refinement to solve linear systems of equations.

## 3.6 Exceptions

The IEEE standard defines five types of exceptions that should be signaled through a one bit status flag when encountered. The multiplier cannot always determine a result by simply doing a multiplication. There are certain inputs that require the multiplier to take special action. The multiplier performs this action in parallel with the regular multiplication, and chooses this special result in cases in which it is required.

Not a number (NaN) - The IEEE 754 Standard specifies that an implementation will return a NaN that is given to it as input, or either one if both inputs are NaN's. The multiplier can be configured to return either the first NaN or the higher of the two. Infinity - Nearly anything multiplied by infinity is properly signed infinity, with the exception of NaN, described above, and zero, described below.

Infinity and zero - The result of the multiplication of infinity and zero is undefined. The multiplier will therefore return a predefined NaN.

### 3.6.1 Invalid Operation

Some arithmetic operations are invalid, such as a division by zero or square root of a negative number. The result of an invalid operation shall be a NaN. There are two types

of NaN, quiet NaN (QNaN) and signaling NaN (SNaN). They have the following format, where s is the sign bit:

QNaN = s 11111111 10000000000000000000000
SNaN = s 11111111 00000000000000000000001

The result of every invalid operation shall be a QNaN string with a QNaN or SNaN exception. The SNaN string can never be the result of any operation, only the SNaN exception can be signaled and this happens whenever one of the input operand is a SNaN string otherwise the QNaN exception will be signaled. The SNaN exception can for example be used to signal operations with uninitialized operands, if we set the uninitialized operands to SNaN. However this is not the subject of this standard.
The following are some arithmetic operations which are invalid operations and that give as a result a QNaN string and that signal a QNaN exception:

a. Any operation on a NaN
b. Addition or subtraction: $\infty + (-\infty)$
c. Multiplication: $\pm 0 \times \pm \infty$

### 3.6.2 Inexact

This exception should be signaled whenever the result of an arithmetic operation is not exact due to the restricted exponent and/or precision range.

### 3.6.3 Underflow

Two events cause the underflow exception to be signaled, tininess and loss of accuracy. Tininess is detected after or before rounding when a result lies between $\pm 2^{Emin}$. Loss of accuracy is detected when the result is simply inexact or only when a de normalization loss occurs. Subnormal numbers, also called “De normalized” allow underflow to occur. The implementer has the choice to choose how these events are detected. They should be the same for all operations. The implemented FPU core signals an underflow exception whenever tininess is detected after rounding and at the same time the result is inexact.

### 3.6.4 Overflow

The overflow exception is signaled whenever the result exceeds the maximum value that can be represented due to the restricted exponent range. It is not signaled when one of the operands is infinity, because infinity arithmetic is always exact. Division by zero also doesn't trigger this exception.

### 3.6.5 Infinity

This exception is signaled whenever the result is infinity without regard to how that occurred. This exception is not defined in the standard and was added to detect faster infinity results.

### 3.6.6 Zero

This exception is signaled whenever the result is zero without regard to how that occurred. This exception is not defined in the standard and was added to detect faster zero results.

## 3.7 Rounding Modes

Since the result precision is not infinite, sometimes rounding is necessary. To increase the precision of the result and to enable round-to-nearest-even rounding mode, three bits were added internally and temporally to the actual fraction: *guard*, *round*, and *sticky* bit. While guard and round bits are normal storage holders, the sticky bit is turned '1' when ever a '1' is shifted out of range. [16].

As an example we take a 5-bits binary number: 1.1001. If we left-shift the number four positions, the number will be 0.0001, no rounding is possible and the result will no be accurate. Now, let's say we add the three extra bits. After left-shifting the number four positions, the number will be 0.0001 101 (remember, the last bit is '1' because a '1' was shifted out). If we round it back to 5-bits it will yield: 0.0010, therefore giving a more accurate result.

The product of two n-bit numbers has the potential of being 2(n+1) bits wide. The result of floating point multiplication, however, must fit into the same n bits as the multiplier and the multiplicand. This, of course, often leads to loss of precision. The IEEE standard attempted to keep this loss as minimal as possible with the introduction of standard rounding modes. When all are enabled, the multiplier supports all IEEE rounding modes: round to nearest even, round to zero, round to positive infinity, and round to negative infinity. The multiplier determines rounding mode based on the two-bit control signal provided as input. The encoding for the rounding modes is shown in Table 3.5.

**Table 3.5**: Rounding mode encodings

| **Rounding Mode** | **Abbr.** | ***Encoding*** |
|---|---|---|
| Nearest even | RN | 00 |
| Zero | RZ | 01 |
| Positive infinity | RP | 10 |
| Negative (minus) infinity | RM | 11 |

The standard specifies four rounding modes:

### 3.7.1 Round to nearest even

This is the standard default rounding. The value is rounded up or down to the nearest infinitely precise result. If the value is exactly halfway between two infinitely precise results, then it should be rounded up to the nearest infinitely precise even.

For example:

**Table 3.6:** Example of rounding to nearest even

| Unrounded | Rounded |
|---|---|
| 3.4 | 3 |
| 5.6 | 6 |
| 3.5 | 4 |
| 2.5 | 2 |

### 3.7.2 Round-to-Zero

Basically in this mode the number will not be rounded. The excess bits will simply get truncated, e.g. 3.47 will be truncated to 3.4.

### 3.7.3 Round-Up

The number will be rounded up towards +∞, e.g. 3.2 will be rounded to 4, while -3.2 to -3.

### 3.7.4 Round-Down

The opposite of round-up, the number will be rounded up towards -∞, e.g. 3.2 will be rounded to 3, while –3.2 to -4.

# CHAPTER 4

# MODELSIM TOOL

## 4.1 General

ModelSim is a verification and simulation tool for VHDL, Verilog, SystemVerilog, SystemC, and mixed-language designs. This chapter provides a brief conceptual overview of the ModelSim simulation environment. ModelSim VHDL implements the VHDL language as defined by IEEE Standards 1076-1987, 1076-1993, and 1076-2002. ModelSim also supports the 1164-1993 Standard Multivalue Logic System for VHDL Interoperability, and the 1076.2-1996 Standard VHDL Mathematical Packages standards. Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with the 1076 specifications. ModelSim Verilog implements the Verilog language as defined by the IEEE Standard 1364-1995 and 1364-2001. ModelSim Verilog also supports a partial implementation of SystemVerilog 3.1, Accellera's Extensions to Verilog. The Open Verilog International Verilog LRM version 2.0 is also applicable to a large extent. Both PLI (Programming Language Interface) and VCD (Value Change Dump) are supported for ModelSim PE and SE users.[19]

### 4.1.1 Code Coverage

Testbenches have become an integral part of the design process, enabling to verify the HDL model is sufficiently tested before implementing your design and helping you automate the design verification process. It is essential, therefore, that we have confidence our testbench is thoroughly exercising our design. Collecting code coverage statistics during simulation helps to ensure the quality and thoroughness of our tests.

### 4.1.2. Compiler Optimizations

To increase simulation speed, ModelSim can apply a variety of optimizations to the design. These include, but are not limited to, mergingprocesses, pulling constants out of

loops, clock suppression, and signal collapsing. We control the level of optimization by specifying certain switches when you invoke the compiler.

### 4.1.3. ModelSim Debug GUI

ModelSim provides an Integrated Debug Environment that facilitates efficient design debug for SoC and FPGA based designs. This GUI has continuously evolved to include new windows and support for new languages. This aims to give an introduction to the ModelSim 6.0 debug environment. This environment is trilingual supporting designs based on VHDL, Verilog (all standards including SystemVerilog, Verilog 2001 and Verilog 1995), and SystemC. ModelSim will enable even more debug capabilities supporting higher levels of abstractions for verification and modeling in SystemVerilog and SystemC. In ModelSim, the GUI has been enhanced and is based on Multiple Document Interface (MDI) layout standard. In addition, the debug windows have been re-organized in such a way as to display design data and simulation results in an intuitive manner.

### 4.1.4. SystemC

The SystemC verification with ModelSim, starting with setting up the simulation environment, and followed by a setup of the use model. Information on using SystemC across Verilog and VHDL boundaries is represented after setup of the model.

### 4.1.5. SystemVerilog

SystemVerilog (SV) is the next generation of the popular Verilog language. As an extension to the IEEE 1364-2001 Verilog standard (referred to hereafter as Verilog) SV has been carefully designed to be 100 percent backward compatible. Under Accellera (the standards body responsible for defining SystemVerilog ), SV has evolved in recent years.

### 4.1.6. Using C in a Mixed Language Environment

C and C++ languages have an important role to play in ASIC design, and using these languages can significantly increase designer productivity. However, C and C++ cannot be used entirely alone; they must work together with conventional HDLs (VHDL and Verilog), to create a mixed HDL and C/C++ environment. The key to success when employing a mixed environment in ASIC design is to choose and use the language that offers the most effective abstraction level for the task at hand.

### 4.1.7. Using the Valgrind Tool with ModelSim

It describes the use of the open source tool, Valgrind, to detect and track down the memory usage errors in user-created C/C++ source files in ModelSim simulation. The Valgrind tool is limited to use with x86/linux based platforms.

### 4.1.8. Verilog 2001

Support for the Verilog-2001 standard (IEEE 1364-2001) in Model Technology's ModelSim® simulator was rolled out over four different releases. Initial support began in March of 2001 with the release of version 5.5. With the latest release of ModelSim, version 5.8, Model Technology is the first EDA Company to complete its implementation of the entire Verilog-2001 standard.

## 4.2. Simulation flow in ModelSim

The diagram shows the basic steps for simulating a design in ModelSim.

### 4.2.1. Creating the working library

In ModelSim, all designs are compiled into a library. We start a new simulation in ModelSim by creating a working library called "work". "Work" is the library name used by the compiler as the default destination for compiled design units.

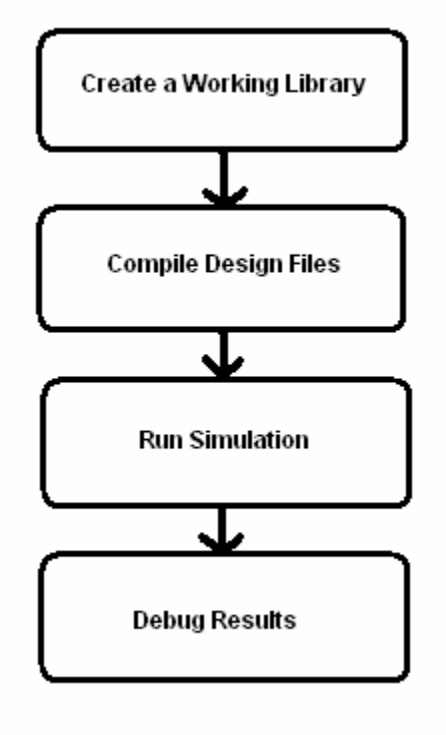


**Figure 4.1:** Basic Simulation Flow Diagram

### 4.2.2 Compiling the design

Before the simulate a design, we must first create a library and compile the source code into that library.

1. Create a new directory and copy the design files for this lesson into it. Start by creating a new directory for this exercise.
2. Start ModelSim if necessary.
    - Type **vsim** at a UNIX shell prompt or use the ModelSim icon in Windows. Upon opening ModelSim for the first time, we will see the Welcome to ModelSim dialog. Click **Close**.
    - Select **File > Change Directory** and change to the directory you created in step 1.

3. Create the working library.
    - Select **File > New > Library**.This opens a dialog where you specify physical and logical names for the library. We can create a new library or map to an existing library. We'll be doing the former.
    - Type **work** in the Library Name field if it isn't entered automatically Click **OK**.

ModelSim creates a directory called *work* and writes a specially formatted file named *_info* into that directory. The *_info* file must remain in the directory to distinguish it as a ModelSim library. Do not edit the folder contents from your operating system; all changes should be made from within ModelSim. ModelSim also adds the library to the list in the Workspace and records the library mapping for future reference in the ModelSim initialization file (*modelsim.ini*). When you pressed OK in step b above, several lines were printed to the Main window Transcript pane:

```
vlib work
vmap work work
# Copying C:\modeltech\win32/../modelsim.ini to modelsim.ini
# Modifying modelsim.ini
# ** Warning: Copied C:\modeltech\win32/../modelsim.ini to
modelsim.ini.
# Updated modelsim.ini.
```

The first two lines are the command-line equivalent of the menu commands you invoked. Many menu-driven functions will echo their command-line equivalents in this fashion. The other lines notify you that the mapping has been recorded in a local ModelSim initialization file. After creating the working library, compile the design units into it. The ModelSim library format is compatible across all supported platforms. We can simulate your design on any platform without having to recompile your design. With the working library created, we are ready to compile your source files. We can compile by using the menus and dialogs of the graphic interface, as in the Verilog or VHDL. The example below shows for VHDL, or by entering a command at the ModelSim> prompt.

1. Compile counter.vhd and tcounter.vhd

- Select **Compile > Compile**.This opens the Compile Source Files dialog. If the Compile menu option is not available, probably have a project open. If so, close the project by selecting **File > Close** when the Workspace pane is selected.
- Select counter.vhd, hold the <Ctrl> key down, and then select tcounter.vhd.
- With the two files selected, click **Compile**. The files are compiled into the *work* library.
- Click **Done**.

2. View the compiled design units.

- On the Library tab, click the '+' icon next to the *work* library and we will see two design units. It can also see their types (Modules, Entities, etc.) and the path to the underlying source files (scroll to the right if necessary).

### 4.2.3 Loading the design into the simulator

Load the test_counter module into the simulator. Double-click test_counter in the Main window Workspace to load the design. It can also load the design by selecting **Simulate > Start Simulation** in the menu bar. This opens the Start Simulation dialog. With the Design tab selected, click the '+' sign next to the work library to see the counter and test_counter modules. Select the test_counter module and click OK. When the design is loaded, you will see a new tab in the Workspace named *sim* that displays the hierarchical structure of the design. It can navigate within the hierarchy by clicking on any line with a'+' (expand) or '-' (contract) icon. It will also see a tab named *Files* that displays all files included in the design. The Objects pane opens by default when a design is loaded. It shows the names and current values of data objects in the current region (selected in the Workspace). Data objects include signals, nets, registers, constants and variables not declared in a process, generics, parameters, and member data variables of a SystemC module.

### 4.2.4 Running the simulation

With the design compiled, you invoke the simulator on a top-level module (Verilog) or a configuration or entity/architecture pair (VHDL). Assuming the design loads successfully, the simulation time is set to zero, and we enter a run command to begin simulation. Now we will run the simulation.

1. Set the graphic user interface to view the Wave debugging pane in the Main window. Enter **view wave** at the command line. This opens one of several panes available for debugging. To see a list of the other panes, select **View > Debug Windows** from the menu bar. Most debugging windows will open as panes within the Main window. The Dataflow window will open as a separate window. We may need to move or resize the windows to our liking. Panes within the Main window can zoom to occupy the entire Main window or undocked to stand alone.

2 Add signals to the Wave window.

- In the Workspace pane, select the **sim** tab.
- Right-click test_counter to open a popup context menu.
- Select **Add > Add to Wave**.

Three signals are added to the Wave window.

3 Run the simulation.

- Click the Run icon in the Main or Wave window toolbar. The simulation runs for 100 ns (the default simulation length) and waves are drawn in the Wave window.
- Type **run 500** at the VSIM> prompt in the Main window. The simulation advances another 500 ns for a total of 600 ns.
- Click the Run -All icon on the Main or Wave window toolbar. The simulation continues running until you execute a break command or it hits a statement in your code (e.g., a Verilog $stop statement) that halts the simulation.
- Click the Break icon. The simulation stops running.

### 4.2.5 Debugging results

If we don't get the results we expect, then we can use ModelSim's robust debugging environment to track down the cause of the problem.

## 4.3 Steps for Design Simulation

There are five steps which are used to stimulate the design using ModelSim which are given below.

### 4.3.1 Step 1 — Collecting Files and Mapping Libraries

Files needed to run ModelSim for the design:

- design files (VHDL, Verilog, and/or SystemC), including stimulus for the design
- libraries, both working and resource
- modelsim.ini (automatically created by the library mapping command

**Providing Stimulus to the Design**

Designer provide stimulus to your design in several ways:

- Language based testbench
- Tcl-based ModelSim interactive command, force
- VCD files / commands

### 4.3.2 Step 2 — Compiling the Design (vlog, vcom, sccom)

Designs are compiled with one of the three language compilers.

### 4.3.2.1 Compiling Verilog (vlog)

ModelSim's compiler for the Verilog modules in the design is vlog. Verilog files may be compiled in any order, as they are not order dependent

#### 4.3.2.2 Compiling VHDL (vcom)

ModelSim's compiler for VHDL design units is vcom. VHDL files must be compiled according to the design requirements of the design.

#### 4.3.2.3 Compiling SystemC (sccom)

ModelSim's compiler for SystemC design units is sccom, and is used only if we have SystemC components in our design.

### 4.3.3 Step 3 — Loading the Design for Simulation

#### 4.3.3.1 vsim top Level Module

The design is ready for simulation after it has been compiled. We may then invoke vsim with the names of the top-level modules (many designs contain only one top-level module). For example, if your top-level modules are "testbench" and "globals", then invoke the simulator as follows: vsim testbench globals. After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references. We can optionally optimize the design with vopt.

#### 4.3.3.2. Using SDF

We can incorporate actual delay values to the simulation by applying SDF back-annotation files to the design.

### 4.3.4. Step 4 — Simulating the Design

Once the design has been successfully loaded, the simulation time is set to zero, and we must enter a **run** command to begin simulation. The basic simulator commands are:

- . add wave
- . force
- . bp

- run
- step

### 4.3.5. Step 5 — Debugging the Design

Numerous tools and windows useful in debugging our design are available from the ModelSim GUI. In addition, several basic simulation commands are available from the command line to assist us in debugging your design:

- describe
- drivers
- examine
- force
- log
- checkpoint
- restore
- show

## 4.4 Design Libraries

### 4.4.1. What is a Library?

A library is a location where data to be used for simulation is stored. Libraries are ModelSim's way of managing the creation of data before it is needed for use in simulation. It also serves as a way to streamline simulation invocation. Instead of compiling all design data each and every time you simulate, ModelSim uses binary pre-compiled data from these libraries. So, if you make a change to a single Verilog module, only that module is recompiled, rather than all modules in the design.

### 4.4.2 Working and Resource Libraries

Design libraries can be used in two ways:

- as a local working library that contains the compiled version of your design;
- as a resource library.

The contents of your working library will change as you update your design and recompile. A resource library is typically unchanging, and serves as a parts source for your design. Examples of resource libraries might be: shared information within your group, vendor libraries, packages, or previously compiled elements of your own working design. You can create your own resource libraries, or they may be supplied by another design team or a third party (e.g., a silicon vendor). For more information on resource libraries and working libraries, see Working Library Versus Resource Libraries, Managing Library Contents, Working with Design Libraries, and Specifying Resource Libraries.

### 4.4.3. Creating the Logical Library (vlib)

Before you can compile your source files, you must create a library in which to store the compilation results. You can create the logical library using the GUI, using **File > New > Library** (see Creating a Library), or you can use the vlib command. For example, the command: vlib work creates a library named **work**. By default, compilation results are stored in the **work** library.

### 4.4.4. Mapping the Logical Work to the Physical Work Directory

VHDL uses logical library names that can be mapped to ModelSim library directories. If libraries are not mapped properly, and you invoke your simulation, necessary components will not be loaded and simulation will fail. Similarly, compilation can also depend on proper library mapping. By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library. You can use the GUI (Library Mappings with the GUI, a command (Library Mapping from the Command Line), or a project (Getting Started with Projects to assign a logical name to a design library. The format for command line entry is:

vmap <logical_name> <directory_pathname>

This command sets the mapping between a logical library name and a directory.

### 4.4.5 Design Libraries

A design library is a directory or archive that serves as a repository for compiled design units. The design units contained in a design library consist of VHDL entities, packages, architectures, and configurations; Verilog modules and UDPs (user-defined primitives); and SystemC modules. The implementation of a design library is not defined within standard VHDL or Verilog. Within ModelSim, design libraries are implemented as directories and can have any legal name allowed by the operating system, with one exception; extended identifiers are not supported for library names. When we create a project, ModelSim automatically creates a working design library. If you don't create a project, you need to create a working design library before you run the compiler. This can be done from either the command line or from the ModelSim graphic interface. From the ModelSim prompt or a UNIX/DOS prompt, use this **vlib** command.

vlib <directory_pathname>

To create a new library with the ModelSim graphic interface, select **File > New > Library**. The options in this dialog are described under "Create a New Library dialog". When you click **OK**, ModelSim creates the specified library directory and writes a specially-formatted file named *_info* into that directory. The *_info* file must remain in the directory to distinguish it as a ModelSim library. The design units are classified as follows:

• **Primary design units**

Consist of entities, package declarations, configuration declarations, modules, UDPs, and SystemC modules. Primary design units within a given library must have unique names.

• **Secondary design units**

Consist of architecture bodies, package bodies, and optimized Verilog modules. Secondary design units are associated with a primary design unit. Architectures by the same name can exist if they are associated with different entities or modules.

### 4.4.5.1 Design unit information

The information stored for each design unit in a design library is:

- retargetable, executable code
- debugging information
- dependency information

## 4.4.6 Working library versus resource libraries

Design libraries can be used in two ways:

- As a local working library that contains the compiled version of your design;
- As a resource library.

The contents of our working library will change as we update our design and recompile. A resource library is typically static and serves as a parts source for our design. We can create your own resource libraries, or they may be supplied by another design team or a third party (e.g., a silicon vendor). Only one library can be the working library. In contrast any number of libraries can be a resource library during a compilation. We specify which resource libraries will be used when the design is compiled, and there are rules to specify in which order they are searched. A common example of using both a working library and a resource library is one where your gate-level design and testbench are compiled into the working library, and the design references gate-level models in a separate resource library.

## 4.4.7 The library named "work"

The library named "work" has special attributes within ModelSim; it is predefined in the compiler and need not be declared explicitly (i.e. **library work**). It is also the library name used by the compiler as the default destination of compiled design units (i.e., it doesn't need to be mapped). In other words the **work** library is the default *working* library. By default, design libraries are stored in a directory structure with a sub-directory for each design unit in the library. Alternatively, it can configure a design library to use archives. To create an archive, use the **-archive** argument to the **vlib** command.

## 4.5 Specifying the resource libraries

### 4.5.1 Verilog resource libraries

ModelSim supports separate compilation of distinct portions of a Verilog design. The **vlog** compiler is used to compile one or more source files into a specified library. The library thus contains pre-compiled modules and UDPs that are referenced by the simulator as it loads the design.

### 4.5.2 VHDL resource libraries

Within a VHDL source file, you use the VHDL **library** clause to specify logical names of one or more resource libraries to be referenced in the subsequent design unit. The scope of a **library** clause includes the text region that starts immediately after the **library** clause and extends to the end of the declarative region of the associated design unit. It does not extend to the next design unit in the file. Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation. The **vcom** command adds compiled design units to the current working library. By default, this is the library named **work**. To change the current working library, we can use **vcom -work** and specify the name of the desired target library.

### 4.5.3 Predefined libraries

Certain resource libraries are predefined in standard VHDL. The library named **std** contains the packages **standard** and **textio**, which should not be modified. The contents of these packages and other aspects of the predefined language environment are documented in the IEEE Standard VHDL Language Reference Manual, Standard 1076. A VHDL **use** clause can be specified to select particular declarations in a library or package that are to be visible within a design unit during compilation. A **use** clause references the compiled version of the package—not the source. By default, every VHDL design unit is assumed to contain the following declarations:

```
LIBRARY std, work;
USE std.standard.all
```

To specify that all declarations in a library or package can be referenced, add the suffix *.all* to the library/package name. For example, the **use** clause above specifies that all declarations in the package *standard*, in the design library named *std*, are to be visible to the VHDL design unit immediately following the **use** clause. Other libraries or packages are not visible unless they are explicitly specified using a **library** or **use** clause.

Another predefined library is **work**, the library where a design unit is stored after it is compiled as described earlier. There is no limit to the number of libraries that can be referenced, but only one library is modified during compilation.

### 4.5.4 Alternate IEEE libraries supplied

The installation directory may contain two or more versions of the IEEE library:

• ieeepure

Contains only IEEE approved packages (accelerated for ModelSim).

• ieee

Contains precompiled Synopsys and IEEE arithmetic packages which have been accelerated by Model Technology including math_complex, math_real, numeric_bit, numeric_std, std_logic_1164, std_logic_misc, std_logic_textio, std_logic_arith, std_logic_signed, std_logic_unsigned, vital_primitives, and vital_timing. We can select which library to use by changing the mapping in the modelsim.ini file. The modelsim.ini file in the installation directory defaults to the ieee library.

## 4.6 Compiling VHDL files

### 4.6.1 Creating a design library

Before we can compile your source files, we must create a library in which to store the compilation results. Use **vlib** to create a new library. For example: vlib work. This creates a library named **work**. By default, compilation results are stored in the **work** library. The **work** library is actually a subdirectory named *work.* This subdirectory contains a special file named *_info*. Do not create libraries using UNIX, MS Windows, or DOS commands always use the **vlib** command.

### 4.6.2 Invoking the VHDL compiler

ModelSim compiles one or more VHDL design units with a single invocation of **vcom**, the VHDL compiler. The design units are compiled in the order that they appear on the command line. For VHDL, the order of compilation is important – we must compile any entities or configurations before an architecture that references them. We can simulate a design containing units written with 1076 -1987, 1076 -1993, and 1076-2002 versions of VHDL. To do so we will need to compile units from each VHDL version separately. The **vcom** command compiles using 1076 -2002 rules by default; use the **-87** or **-93** argument to **vcom** to compile units written with version 1076-1987 or 1076 -1993, respectively. We can also change the default by modifying the VHDL93 variable in the modelsim.ini file.

### 4.6.3 Dependency checking

Dependent design units must be reanalyzed when the design units they depend on are changed in the library. **vcom** determines whether or not the compilation results have changed. For example, if we keep an entity and its architectures in the same source file and we modify only an architecture and recompile the source file, the entity compilation results will remain unchanged and we will not have to recompile design units that depend on the entity.

### 4.6.4 Simulating VHDL designs

After compiling the design units, we simulate your designs with **vsim.** The VHDL design can be simulate from the UNIX or Windows/DOS command line. We can also use a project to simulate or the **Simulate** dialog box. For VHDL invoke **vsim** with the name of the configuration, or entity/architecture pair. Note that if you specify a configuration you may not specify an architecture. This example invokes **vsim** on the entity **my_asic** and the architecture **structure**:

vsim my_asic structure **vsim** is capable of annotating a design using VITAL compliant models with timing data from an SDF file. We can specify the min:typ:max delay by invoking **vsim** with the **-sdfmin**, **-sdftyp**, or **-sdfmax** option. Using the SDF file *f1.sdf* in the current work directory, the following invocation of **vsim** annotates maximum timing values for the design unit *my_asic*: vsim -sdfmax /my_asic=f1.sdf my_asic By default, the timing checks within VITAL models are enabled. They can be disabled with the **+notimingchecks** option. For example: vsim +notimingchecks topmod

### 4.6.5 Default binding

By default ModelSim performs default binding when we load the design with **vsim.** The advantage of performing default binding at load time is that it provides more flexibility for compile order. Namely, entities don't necessarily have to be compiled before other entities/architectures which instantiate them. However, we can force ModelSim to perform default binding at compile time. This may allow we to catch design errors (e.g., entities with incorrect port lists) earlier in the flow. Use one of these two methods to change when default binding occurs:

• Specify the **-bindAtCompile** argument to **vcom.**
• Set the BindAtCompile variable in the modelsim.ini to 1

# CHAPTER 5

# DESIGN AND IMPLEMENTATION

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely-used standard for floating-point computation, and is followed by many CPU and FPU implementations. n1 and n2 are the two Floating Point numbers, the product of both, denoted as n, can be expressed as:

$$n = n1 \times n2$$
$$= (-1)^{S1} \cdot p1 \cdot 2^{E1} \times (-1)^{S2} \cdot p2 \cdot 2^{E2}$$
$$= (-1)^{S1+S2} \cdot (p1 \cdot p2) \cdot 2^{E1+E2}$$

This means that the result of multiplying two FP numbers can be described as multiplying their significands and adding their exponents. The resultant sign S is S1 +S2, the resultant significand p is the adjusted product of p1 ·p2 and the resultant exponent E is the adjusted E1+E2+bias. In order to perform floating-point multiplication, a simple algorithm is realized:

- Add the exponents and subtract 127.
- Multiply the mantissas and determine the sign of the result.
- Normalize the resulting value, if necessary.

## 5.1 Algorithms and Design of Floating Point Multiplication

Figure 5.1 shows how the floating point multiplication is performed. Floating-point multiplication is inherently easier to design than floating-point addition or subtraction.

Multiplication requires integer addition of operand exponents and integer multiplication of significands that facilitate normalization when multiplying normalized significands.

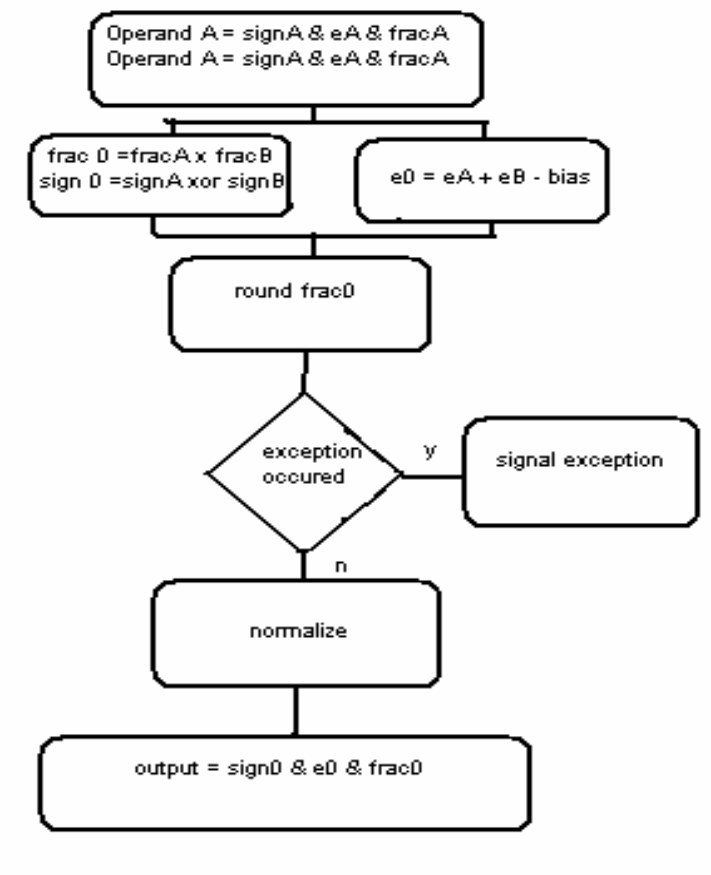


**Figure 5.1:** Basic design of Floating Point Multiplier

These independent operations within a multiplier make it ideal for pipelining. In floating point multiplication the following three steps can be done:

- Unpack the operands, re-insert the hidden bit, and check for any exceptions on the operands (such as zeros or NaNs).
- Multiplication of the significands, calculation of the sign of the two significands, and addition of the exponents take place.

- The final result needs to be normalized and the exponent adjusted before packing and removing the hidden bit.

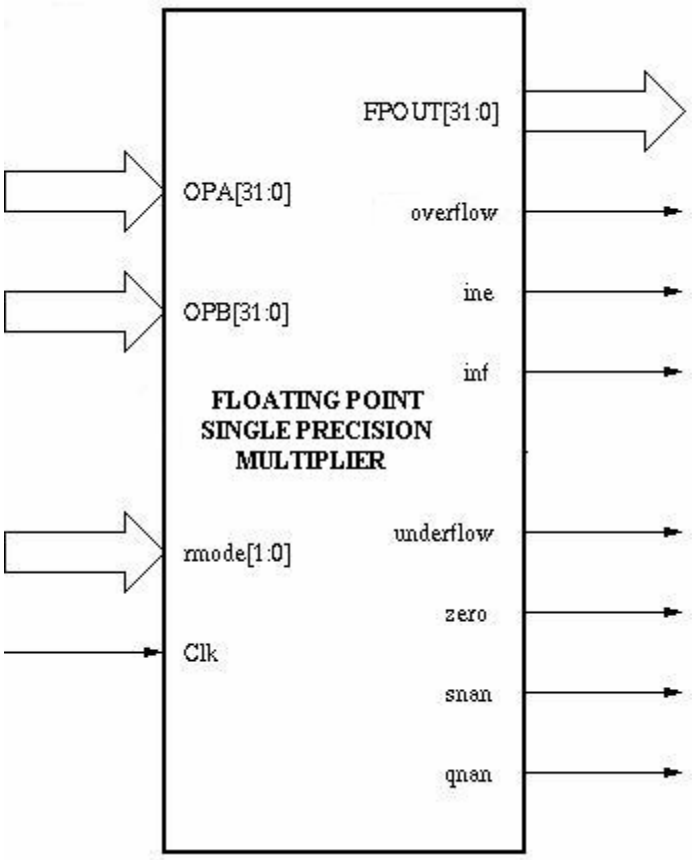


**Figure 5.2**: Entity Block Diagram of Single Precision Multiplier

To demonstrate the basic steps, let's say we want to multiply two 5-digits FP numbers:

$2^{100} \times 1.1001$

X $2^{110} \times 1.0010$

**Step 1**: Multiply fractions and calculate the result exponent.

1.1001

x 1.0010

---

1.11000010

so frac0= 1.11000010 and $e_0 = 2^{100+110-bias} = 2^{83}$

**Step 2**: Round the fraction to nearest-even

frac0= 1.1100

**Step 3**: Result

$2^{83} \times 1.1100$

Multiplication does not require shifting of the significands or adjustment of the exponents as in the adder unit until the final stage for normalization purposes. For the basic summation of partial products in a floating-point multiplication represented in scientific notation (significand multiplied by the radix to some power), one multiplies the two significands and adds the two radix powers. Normalization of the significand ensures the decimal point of the significand has an exactly one significant digit to the left of it which may or may not need to be done.

For example, multiplying **A** = 5.436 x $10^8$ by **B** = 8.995 x $10^{-4}$ would result in **C** = 48.89682 x $10^4$ and 4.889682 x $10^5$ with normalization. Similar operations are performed on binary numbers as well. The format chosen for this design follows the basic steps as described above to perform multiplication. The designed floating-point multiplier consists of the 13 stage pipelined integer multiplier, pipeline delay elements, and an adder/subtractor unit to handle the exponent computation. The multiplier cannot directly accept 32-bit floating-point operands as for operand B. Operand A may be issued directly to the multiplier, however. The operand B mantissa must be given to the multiplier on a partial basis but may be interleaved with other operand B values entering the multiplier pipeline. Since the multiplier must be issued static coefficients, an interleaved pattern may be formed for operand B to make implementation and design much easier. The exponent, however, must be calculated before and after the mantissa multiplication. Before entering the exponent pipeline delay, the exponents of operand A and B are added together. Once the mantissa product completes, the most significant bit must be examined to ensure the implied one is properly placed. The final stages of the pipelined floating-point multiplier ensure the biasing of the exponent is properly done which relies on the most significant bit of the integer multiplication product. The floating-point multiplier block diagram can be seen in Figure 5.3.

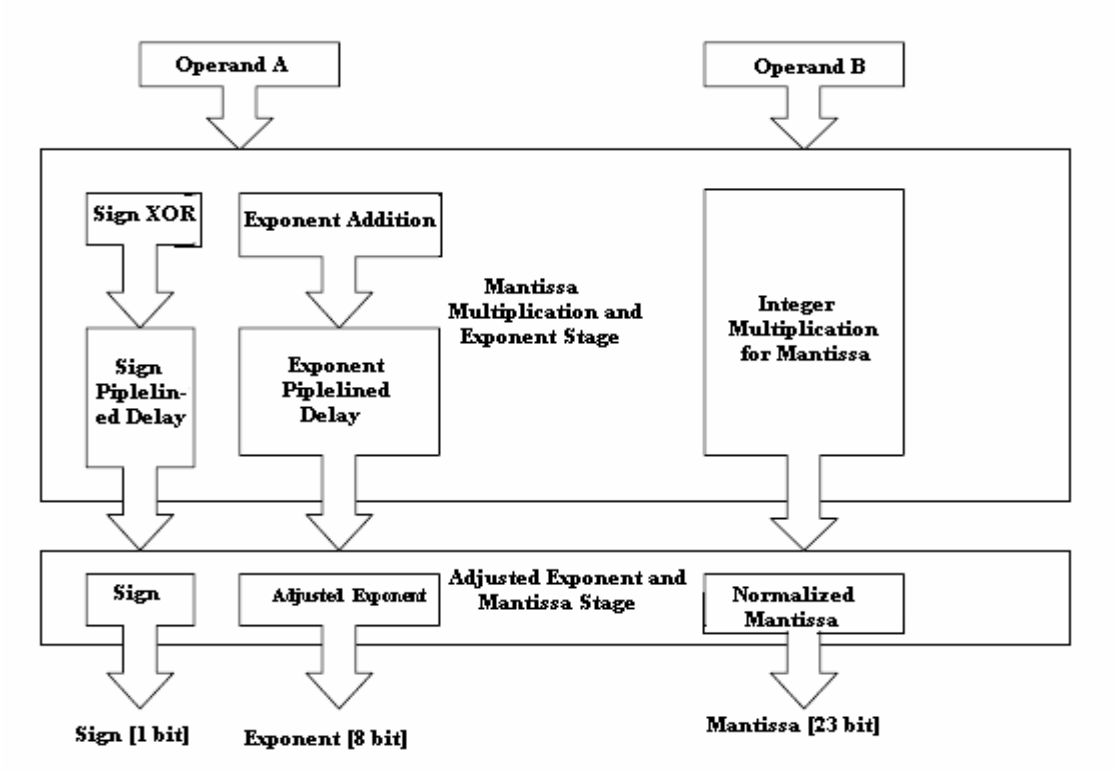


**Figure 5.3**: Pipelined Multiplier Block Diagram

## 5.2 24-bit Pipelined Integer Multiplier

The pipelined floating-point multiplier generates a product every clock when the pipeline has completely filled, and has a latency of 13 cycles. The pipeline stages for the multiplier are much simpler in comparison to the adder stages. Twelve of the 13 stages are used for the computation of the integer multiply. By simply relying upon VHDL, synthesis tools for the creation of the multiplication produced a design which was deemed unacceptable considering the planned resource budget. An alternative integer multiplier was created using a parameterized multiplier generation program. The generated 24X24 integer multiplier utilizes Booth recoding and pipeline stages to preserve routing, timing, and size of the multiplier. Two bits of the multiplier are issued at a time for twelve consecutive clock cycles, starting with the lowest two bits.

Figure 5.3 illustrates the pipeline multiplier stages for the floating-point multiplier. The exponent and mantissa operations can be performed concurrently until the final stage where normalization takes place. In floating-point multiplication, the exponents must be added together as they are in this implementation during the first stages. The result from the exponent addition continues through a pipeline delay until the mantissa result completes. Carry-out logic from the mantissa multiplication informs the control logic not to perform a 1-bit shift since the implied one exists. Note that the exponent must continue through several pipeline delays that require registered logic.

## 5.3 Implementation in VHDL

The multiplier VHDL consists of several different components that rely on a clocked process and registered signals. The components consist of a pipeline delay element, a 9-bit adder and a 24X24 pipelined integer multiplier. The VHDL clocked process provides much of the glue logic for the components used in order to ensure signals to each component are registered properly and to avoid timing hazards. The pipelined integer multiplier, for instance, requires that the inputs be registered for expected results. The inputs to the floating-point multiplier need to be checked for a possible zero outcome and assert a flag through the pipeline to indicate a zero value be given as the result during the last stage in the pipeline. [11]. The VHDL code provides concurrent operations for some of the initial stages. As the mantissa undergoes integer multiplication, calculations on the exponent are done and passed through a pipeline delay to remain synchronized with the integer multiplier data. The VHDL used in the multiplier differs from the 32-bit floating-point pipelined adder in that no state machines are required for the multiplier. Instead, the VHDL provides minimal control logic to ensure the components are given data on the correct cycles.

### 5.3.1 Mantissa Multiplication and Exponent Addition

The multiplier undergoes two separate, parallel operations during the first 12 stages. One of the operations includes multiplying the two 24-bit mantissa values using the 24X24 pipelined integer multiplier. The multiplier generates the result on the thirteenth clock cycle. During the mantissa calculation, the exponent addition takes place using the 9-bit integer adder component. Nine bits, instead of eight, are used to handle carry-out situations. The carry-out bit provides important information used in the final stage of the floating-point multiplier to handle exponent biasing adjustments. Since the exponent calculation does not require more than a clock cycle, a pipeline delay component delays the calculated exponent result until the last stage when the bias adjustments are ready to be done. In addition, two smaller logic operations take place. The first determines if either of the input operands are zero. If so, a special zero-flag needs to be set. The second uses XOR logic to determine the resulting sign bit of the two input operands. The zero-flag and sign bit need to be delayed as well until the last stage in the floating-point multiplier. All data going through the pipelined delay must continue to be synchronized with operand B going through the pipelined integer multiplier.

### 5.3.2 Exponent Adjustment and Product Assembly Stage

The last stage receives the data from the pipelined integer multiplier and the other pipeline delay elements. The stage logic checks the zero-flag bit to see if the output is simply a zero. Otherwise, a one in the most significant bit of the mantissa indicates the resulting mantissa value has already been normalized. If not, one and the mantissa output shifted by one must adjust the exponent. The exponent undergoes subtraction to remove an extra biasing factor from the addition of an earlier stage in the pipeline. Depending on the most significant bit of the mantissa, different values are subtracted from the exponent. The final stage assigns the resulting values to the output signals of the floating-point multiplier.

## 5.4 Stages in Single Precision Floating Point Multiplier

The standard floating point multiplier has several stages:

- Prenormalizing,
- Multiplying,
- Postnormalizing,
- Shifting
- Rounding

All of the stages involve multiple steps, but some stages are more complex than others. Each is described in its own section. [14]

### 5.4.1 Prenormalizing

Recall that, in IEEE 754 format, normalized numbers have an implicit leading 1, and that denormalized numbers do not. Additionally, recall that denormalized numbers use leading zeros to increase the range of the exponent. To keep the multiplication stage simple, both inputs are converted into the same form. In the prenormalization stage, the multiplier produces two numbers, C and D, defined as follows:

$$SC = SA$$

$$xc = \begin{cases} 0 & \text{if } A \text{ is normalized} \\ \text{amount } MA \text{ must be shifted until ufp} & \text{if } A \text{ is denormalized} \end{cases}$$

$$ec = ea - xc$$

$$MC = \begin{cases} (1, MA) & \text{if } a \text{ is normalized} \\ (0, MA << xc) & \text{if } a \text{ is denormalized} \end{cases}$$

with similar definitions for D.

### 5.4.2 Multiplying

The multiplication stage involves three parts: multiplying *mc* and *md*, adding *ec* and *ed*, and detecting tininess. Multiplying the mantissas – The two mantissas are already in a standard form due to the prenormalization stage. The multiplication part produces two new values, *mf* and the "two or more" flag, *t*:

$$mf = mc \times md$$

$$t = \begin{cases} 1 & mf \geq 2 \\ 0 & otherwise \end{cases}$$

Adding the exponents – *ec* and *ed* include an implicit *bias*[1] and *ef* must have the same bias:

$$ef = (ec\text{-}bias) + (ed\text{-}bias) + bias$$

This can easily be simplified to:

$$ef = ec + ed - bias$$

which would be the result if *mf* never overflowed the range [1,2). Because it does overflow that range, however, the actual result is:

$$ef = ec + ed - bias + t$$

Detecting tininess – The result is said to be *tiny* if it smaller than can be represented by a normalized number.

$$tiny = \begin{cases} 1 & ef \le 0 \\ 0 & otherwise \end{cases}$$

### 5.4.3 Postnormalizing

In the postnormalization stage, the multiplier normalizes the product and returns MN:

$$MN = \begin{cases} MF << 0 & tiny \bullet t \\ MF << 1 & tiny \oplus t \\ MF << 2 & otherwise \end{cases}$$

### 5.4.4 Shifting

If the product is a denormalized number, then it might need to be shifted to the right to provide the appropriate number of leading zeros to indicate the correct exponent. This stage outputs MQ and EQ:

$$MQ = \begin{cases} MN & en > 0 \\ MN >> (-en) & otherwise \end{cases}$$

$$eq = \begin{cases} en & en > 0 \\ 0 & otherwise \end{cases}$$

### 5.4.5 Rounding

The product of two n-bit numbers has the potential of being 2(n+1) bits wide. The result of floating point multiplication, however, must fit into the same n bits as the multiplier and the multiplicand. This, of course, often leads to loss of precision. The IEEE standard attempted to keep this loss as minimal as possible with the introduction of standard rounding modes. [16]

When all are enabled, the multiplier supports all IEEE rounding modes: round to nearest even, round to zero, round to positive infinity, and round to negative infinity. [12]

The multiplier determines rounding mode based on the two-bit control signal provided as input. The encoding for the rounding modes is shown inTable 5.1.

**Table 5.1:** Rounding mode encodings

| **Rounding Mode** | **Abbr.** | **Encoding** |
|---|---|---|
| Nearest even | RN | 00 |
| Zero | RZ | 01 |
| Positive infinity | RP | 10 |
| Negative (minus) infinity | RM | 11 |

MQ, also called the "infinitely precise result," is split into an upper (MU) and lower (ML) portion. Recalling that the mantissa of a single-precision floating-point value is twenty-three bits long,

MU = MQ [MSB : MSB-22] (the 23 most significant bits)

ML = MQ [MSB-23 : 0] (everything else)

Additionally, the multiplier produces the “rounded up” version of MU, MV:

MV = MU + ulp

The choice between *mu* and *mv* is called the rounding decision, R, and can differ according to the rounding mode:

Round nearest even:

$$R = \begin{cases} 1 & (mv - mq) < (mu - mq) \\ 1 & (mv - mq) = (mu - mq) \quad AND \quad mv \bmod 2 = 0 \\ 0 & otherwise \end{cases}$$

Round to infinity:

$$R = \begin{cases} 1 & OR(ML)\ OR\ shiftloss \\ 0 & otherwise \end{cases}$$

Round to zero (also called "truncate" mode):

$$R = 0$$

The result of the rounding stage, and of the entire standard path, p, is then determined:

$$p = \begin{cases} u & R = 0 \\ v & R = 1 \end{cases}$$

### *5.4.6 Special Case Path*

The multiplier cannot always determine a result by simply doing a multiplication. There are certain inputs that require the multiplier to take special action. The multiplier performs this action in parallel with the regular multiplication, and chooses this special result in cases in which it is required.

Not a number (NaN) - The IEEE 754 Standard specifies that an implementation will return a NaN that is given to it as input, or either one if both inputs are NaN's. The multiplier can be configured to return either the first NaN or the higher of the two. The Intel Pentium series returns the higher of the two NaN’s and, as this multiplier was tested using a processor from that series, the multiplier is by default set to do the same.

Infinity - Nearly anything multiplied by infinity is properly signed infinity, with the exception of NaN, described above, and zero, described below.

Infinity and zero - The result of the multiplication of infinity and zero is undefined. The multiplier will therefore return a predefined NaN. If none of these cases apply, the special case path signals that the result of the standard path should be chosen.

# CHAPTER 6

# RESULTS AND VERIFICATION

## 6.1 Results

This design has been implemented, simulated on ModelSim and synthesized for VHDL. The HDL code uses VHDL 2001 constructs that provide certain benefits over the VHDL 95 standard in terms of scalability and code reusability. Simulation based verification is one of the methods for functional verification of a design. In this method, test inputs are provided using standard test benches. The test bench forms the top module that instantiates other modules. Simulation based verification ensures that the design is functionally correct when tested with a given set of inputs. Though it is not fully complete, by picking a random set of inputs as well as corner cases, simulation based verification can still yield reasonably good results.

The following snapshots are taken from ModelSim after the timing simulation of the floating point multiplier core.

Consider the inputs to the floating point multiplier are :

A = 1 10000010 00000000000000000000000

B = 0 10000011 00000000000000000000000

The output of the multiplier should be:

1 10000110 00000000000000000000000

And flags outputs of this multiplier

ine = 0; inf = 0; overflow = 0; qnan = 0; snan = 0; underflow = 0; zero = 0;

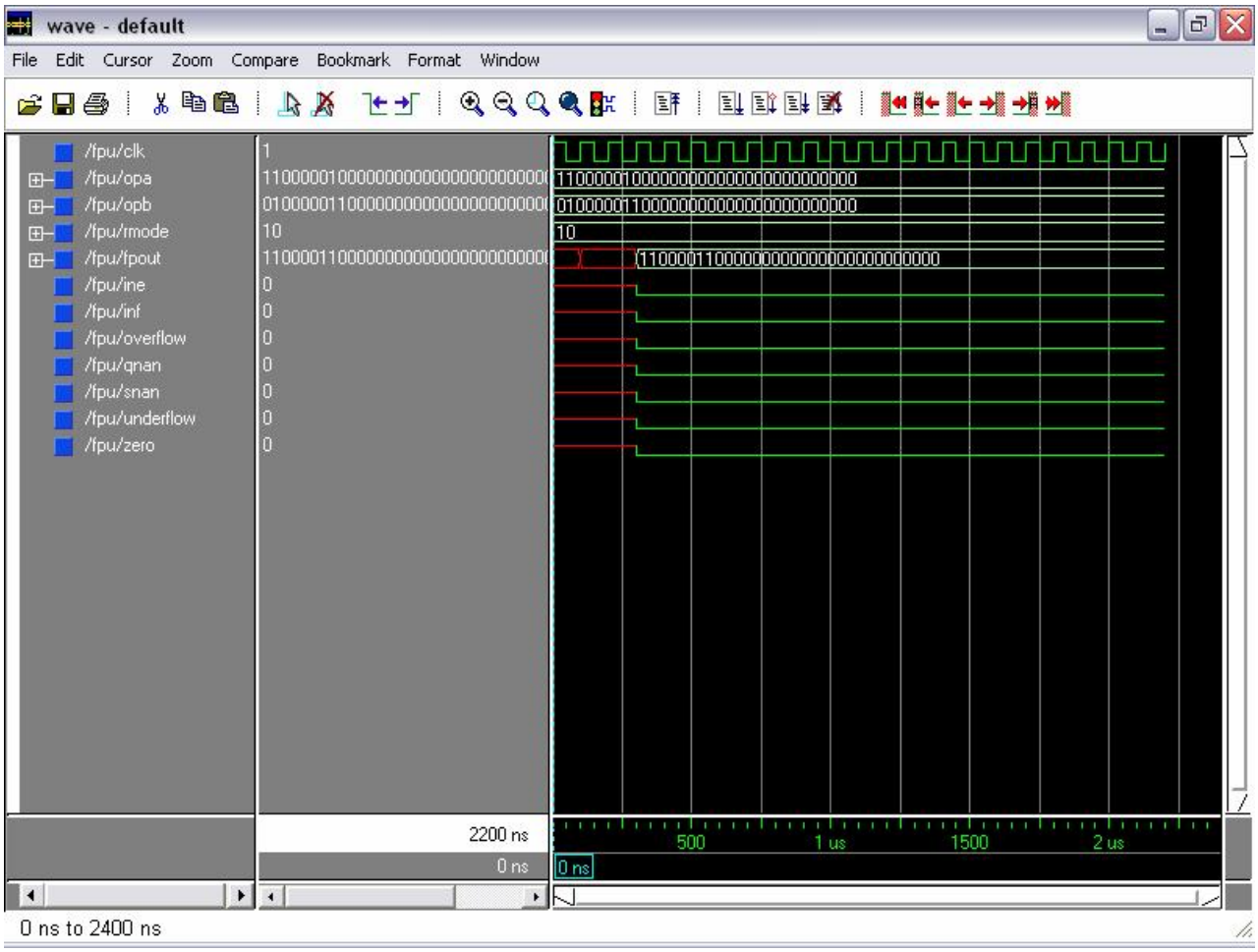


**Figure 6.1**: Output of Single Precision Floating Point Multiplier when above input's are given

The following snapshots are taken from ModelSim after the timing simulation of the floating point multiplier core.

Consider the inputs to the floating point multiplier are :

A = 0 11111111 00000000000000000000000

B = 1 11111111 00000000000000000000000

The output of the multiplier should be:

1 11111111 00000000000000000000000

And flags outputs of this multiplier

ine = 0; inf = 1; overflow = 0; qnan = 1; snan = 0; underflow = 0; zero = 0;

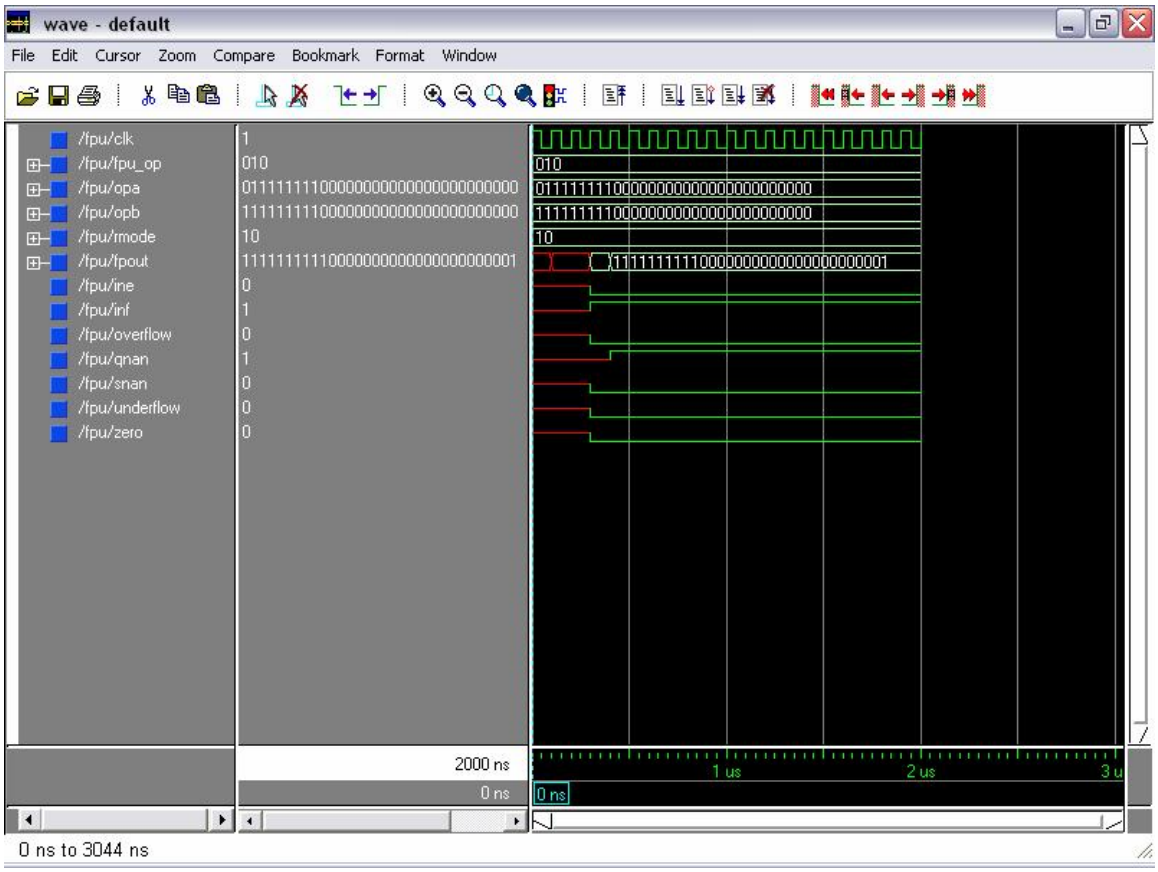


**Figure 6.2**: Output of Single Precision Floating Point Multiplier when above input's are given

The following snapshots are taken from ModelSim after the timing simulation of the floating point multiplier core.

Consider the inputs to the floating point multiplier are :

A = 0 11111111 00000000000000000000000

B = 0 00001110 00111111000001111100110

The output of the multiplier should be:

0 11111111 00000000000000000000000

And flags outputs of this multiplier

ine = 0; inf = 1; overflow = 0; qnan = 0; snan = 0; underflow = 0; zero = 0;

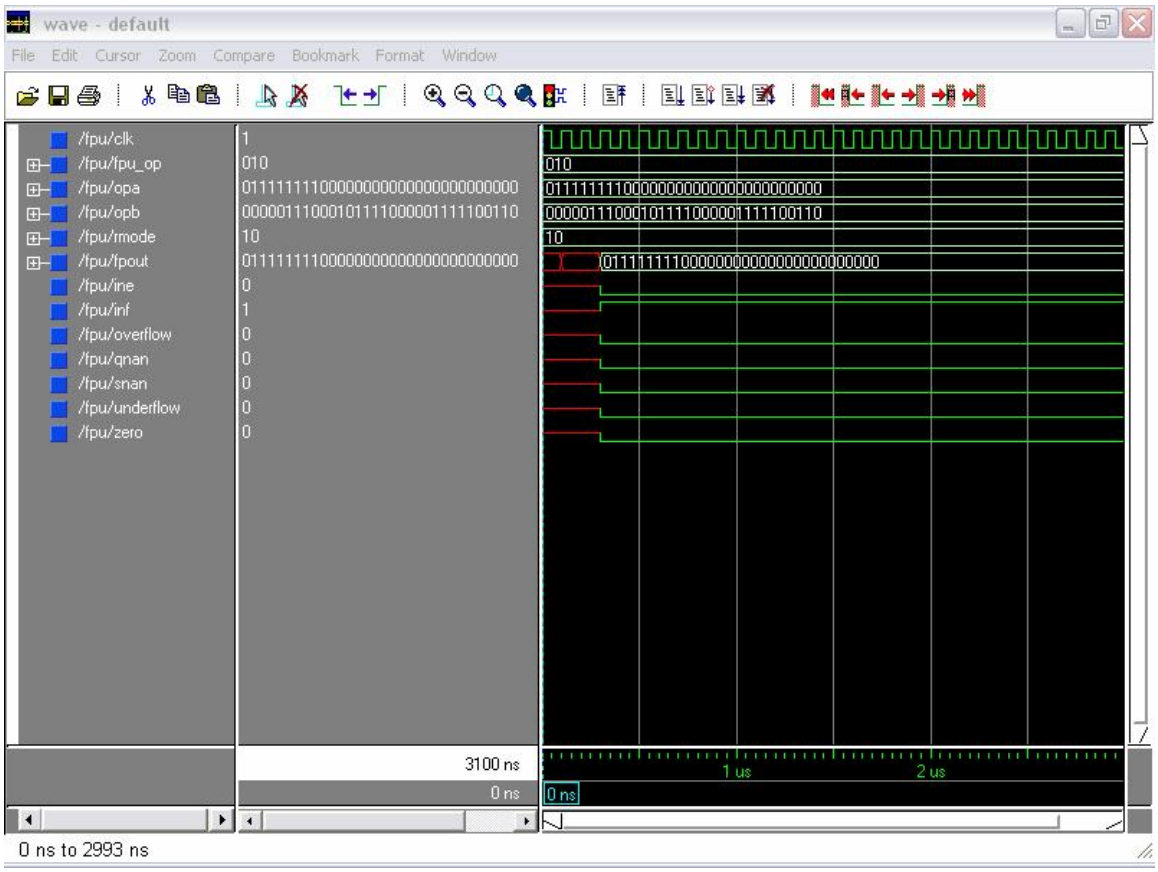


**Figure 6.3**: Output of Single Precision Floating Point Multiplier when above input's are given

The following snapshots are taken from ModelSim after the timing simulation of the floating point multiplier core.

Consider the inputs to the floating point multiplier are:

A = 1 10000010 00000000000000000000000

B = 0 10000011 00000000000000000000000

The output of the multiplier should be:

1 10000110 00000000000000000000000

And flags outputs of this multiplier

ine = 1; inf = 0; overflow = 0; qnan = 0; snan = 0; underflow = 1; zero = 0;

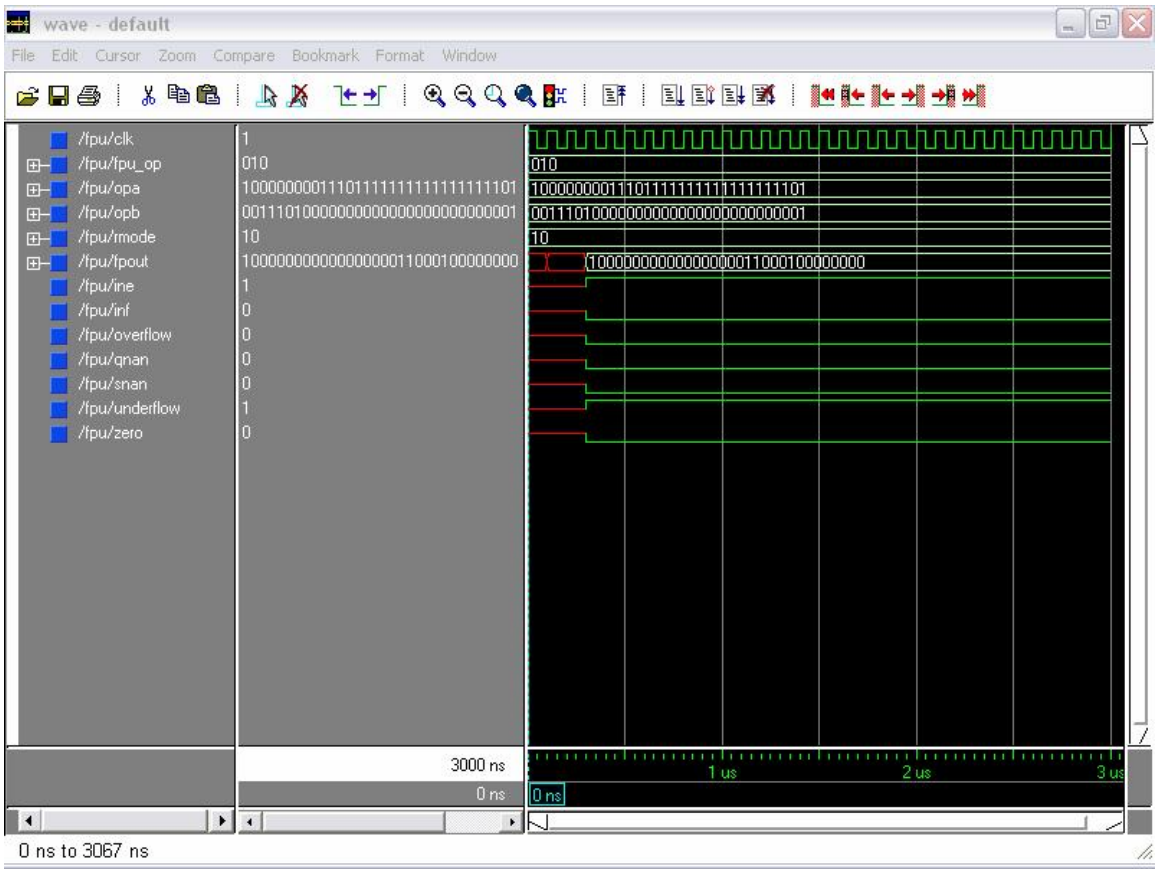


**Figure 6.4**: Output of Single Precision Floating Point Multiplier when above input's are given

## 6.2 Verification

Design verification is defined as the reverse process of design. It takes in an implementation as an input and confirms that the implementation meets the specifications. Though design verification includes functional verification, timing verification, layout verification and electrical verification, functional verification is by default termed design verification. [13]

Two popular forms of verification are the simulation based approach and the formal verification approach. The most important difference between the two approaches is that simulation based approaches need input vectors while formal verification approaches do

not. In the former, we generate input vectors and derive reference outputs from them. However a formal verification approach differs in that it predetermines what output behavior is desirable and uses the formal checker to see if it agrees or disagrees with the desired behavior. This shows that simulation based approach is input driven while formal approach is output driven. Since formal verification methodology operates on an input space as against chosen vectors, it can be more complete. Simulation based approach takes a point in the input space at a time and therefore samples few points only. However, this can be justified due to the extensive use of memory and long runtime that formal verification uses. Besides when memory overflow is encountered, the tools are at a loss to show what are the right problem and its fix. [17]

This design has been verified using a simulation based approach. In order to verify the functionality of the cores, it is mandatory to have inputs with signal combination. Since an explicit two's complement determination unit is used, the cores must be able to distinguish between negative number and positive number. The design is capable of handling both normalized inputs and denormalized inputs. A check for denormalized inputs is already embedded in the design. However we need to verify if the accelerator gracefully terminates with zero outputs when such inputs are given. The Normalizer is the main part of the module that prepares the multiplier for rounding. This module needs to be completely verified. Hence inputs are so designed that each case in the priority encoder gets exercised. Verification of this module covers a large number of values from the input space. Lastly, the design is verified for overflow and underflow cases. A value of exponent greater than 127 should set the result to infinity and a value of zero is set when the exponent is zero and the mantissa is also zero.

# CHAPTER 7

# Conclusion and Future Scope of Work

## 7.1 Conclusion

Single precision floating point multiplier is designed and implemented using ModelSim in this thesis. The designed multiplier conforms to IEEE 754 single precision floating point standard. In this implementation exceptions (like invalid, inexact, infinity, etc) are considered. In this implementation rounding modes like round to positive infinity, round to negative infinity, round to zero and round to even. The designed is verified using fpu_test test bench. The design is also verified for overflow and underflow cases.

## 7.2 Future Scope of Work

- Double Precision Floating Point multiplier can be implemented.
- Speed improvement using advanced algorithms like booth algorithm etc.
- ALU can be designed for DSP application

# REFERENCES


1. IEEE standard for binary-floating point arithmetic, ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronic Engineers Inc., New York, August 1985.

2. David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic, 1991.

3. I. Koren, Computer Arithmetic Algorithms, Second Edition, prentice Hall, 2002.

4. An ANSI/ IEEE Standard for Radix-Independent Floating-Point Arithmetic, Technical Committee on microprocessor of IEEE computer society, October, 1987.

5. Steve Hollasch, IEEE Standard 754 Floating Point Numbers, February 2005.

6. BROWN, Stephen D. Fundamentals of Digital Logic with VHDL designs. Boston: McGraw-Hill, 2000.

7. John L Hennesy & David A. Patterson "Computer Architecture A Quantitative Approach" Second edition; A Harcourt Publishers International Company

8. J. Bhasker**, *A VHDL Primer*,** Third Edition, Pearson, 1999.

9. M. Ercegovac and T. Lang, Digital Arithmetic, Morgan Kaufmann Publishers, 2004

10. John. P. Hayes, "Computer Architecture and Organization", McGraw Hill, 1998.

11. Peter J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers, 95 Inc., 1996.

12. Prof. W. Kahan, Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic. Link: www.cs.berkeley.edu/~wkahan/ieee754status.html

13. Wikipedia, the free encyclopedia, IEEE 754-1985.
Link: http://en.wikipedia.org/wiki/floating _point.html

14. Behrooz Parhami, Computer Arithmetic, Algorithms and Hardware Design Oxford University Press.2000.

15. IEEE Floating Point Representation of Real Number, Fundamentals of Computer Science. Link: http://www.math.grin.edu/~stone/courses/fundamentals/ieee-reals.html

16. M. J. Flynn and S. F. Oberman, Advanced Computer Arithmetic Design, John Wiley and Sons, 2001.

17. N. Weste, D. Harris, CMOS VLSI Design, Third Edition, Addison Wesley, 2004.

18. Beebe, H.F.Nelson, Floating Point Arithmetic, Computation in Modern Science & Technology, December,2007.

19. P. Karlstrom, A. Ehliar, High Performance Low Latency Floating Point Multiplier, November 2006

20. www.ieee.org

21. www.modelsim.com