Design of a Decimal to IEEE-754 Converter using Verilog

Submitted by
Ahmed Bin Nasser
1505096

Md. Omer Danish
1505053

Supervised by
Dr. Md. Mostofa Akbar

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

20 February 2021

Declaration

This thesis is a presentation of our original research work. Wherever contributions of others are involved, every effort is made to indicate this clearly, with due reference to the literature, and acknowledgement of collaborative research and discussions. The work was done under the guidance of Professor Dr. Md. Mostofa Akbar,Department of Computer Science and EngineeringBangladesh University of Engineering and Technology.

I played a major role in the preparation and execution of the experiment, and the data analysis and interpretation are entirely by own work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this present world of science and technology we handling more and more data.A great amount of this data comes under the umbrella of floating point.To facilitate the calculation of floating point numbers we are designing a Application-specific integrated circuit.We are going to convert floating point decimal numbers to single precision IEEE-754 binary number using Verilog .This should follow RTL rules and standard.

1.1 Problem Definition

We have to convert real numbers to single precision IEEE-754 floating point format using Verilog.The code should follow RTL rules and standard.

Let's suppose 2 floating-point numbers A and B where A=-18.0 and B=9.5
Binary representation of the operands:

A = -10010.0
B = +1001.1

Normalized representation of the operands:

A = -1.001 X 2^4 exponent = 4+127
B = +1.0011 X 2^3 exponent = 3 + 127

IEEE-754 representation of A and B are

Decimal number	Sign Bit	Exponent	Mantissa
A	1	10000011	00100000000000000000000
B	0	10000010	00110000000000000000000

Table 1.1: IEEE 754 bit representation of A , B

.

Principal Tasks

In this this problem scenario we have to perform the following tasks to achieve our

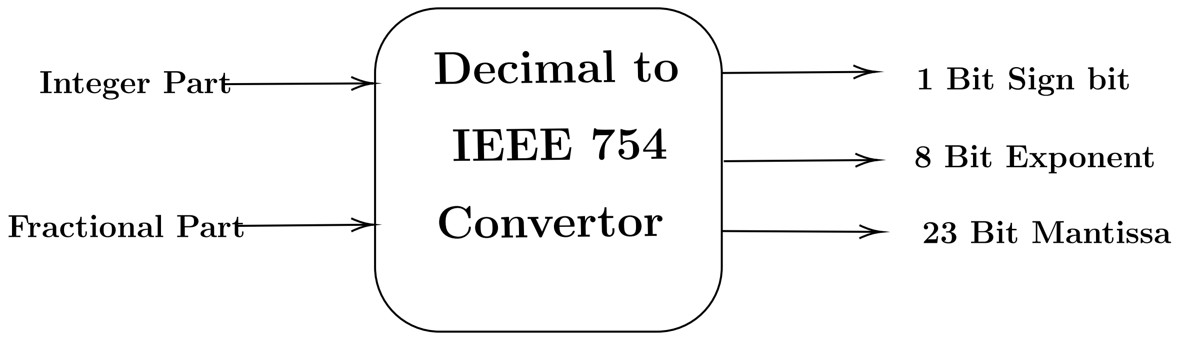


Figure 1.1: Block diagram of total system

goal.

1. Converting fractional part to Binary
2. Mantissa calculation
3. Exponent calculation
4. Sign bit calculation
5. Writing testbenches and synthesizing

1.2 Motivation

With passing time we need more and more fast calculation.But we can not improve CPU performance to a great extent.But we can get huge facility from task specific chip design. In present world we dealing with lots of number every second and floating point numbers are the indispensable part of it. While many applications could benefit from floating-point processing, this technology limitation forces a fixed-point implementation. Some applications are given below

- Wireless communications
- Radar
- Medical imaging
- Motor control
- All could benefit from the high dynamic range afforded by floating-point processing.

1.3 Research Challenges

The format is supported by microprocessor architectures. However, the IEEE 754 format is inefficient to implement in hardware level.Also floating-point processing is not supported in VHDL or Verilog. Newer versions, such as SystemVerilog, allow floating-point variables .But industry-standard synthesis tools do not support floating-point technology. The mantissa includes an implicit 1. Each mantissa digital representation of range $[0 : 0.999..]$, actually maps to a value in the range of $[1 : 1.999..]$.Sign bit is treated separately, rather than using traditional twos complement signed representation. In floating-point processors, the CPU core has special circuits to perform floating point operations because of its complexity. Typical CPUs operate serially, so one or a small number of computational units are used to implement the sequence of software operations. Floating-point results cannot be verified by comparing bit for bit, as is typical in fixed point arithmetic.The reason is that floating-point operations are not associative.

Chapter 2

Background Study

2.1 IEEE 754 Floating Point Standard

IEEE-754 floating point standard is the most common representation today for real numbers on computers. Although there are other representations, it is the most common representation used for floating point numbers.
There are multiple variants of IEEE-754 formats.

- Half precision
- Single precision
- Double precision
- Quadruple precision
- Octuple precision

The larger the precision the bigger and dynamic is the range of number it can represent.

Name	Common Name	Base	Significand Bits	Exponent Bits	Exponent Bias
Binary 16	Half precision	2	11	5	15
Binary 32	Single precision	2	24	8	27
Binary 64	Double precision	2	53	11	1023
Binary 128	Quadruple precision	2	113	15	16383
Binary 256	Octuple precision	2	237	19	262143

Table 2.1: Different formats of IEEE 754

We are working with single precision floating point format.It's total Length is 32 bits.

- Sign bit : 1 bit
- Exponent: 8 bits
- Mantissa: 23 bits
- Exponent Bias : 127
- Range of representation: $1.18X10^{38}$ to $3.4X10^{+}38$

The number in binary, must be normalized. The integer part must always be equal to 1 .The exponent, an integer value, is not represented in 2's-complement, rather in a biased representation. a bias of 127 is added to the exponent. As the value 0 can not be normalized, a special representation is reserved for. All bits to zero. In general, the values 00000000 and 11111111 from the exponent field are reserved for special cases.They are not biased values 00000000 is used for non-normalized values. 11111111 is used for infinity and NaN(not a number)

2.2 Application Specific Integrated Circuit

ASIC stands for Application-Specific Integrated Circuit. It is a microchip which is design to execute hashing algorithm as fast as possible and ASIC is built for a custom single hash algorithm. It has an ability to calculate 100,000 times faster hash then best CPU.
An ASIC (application-specific integrated circuit) is a microchip designed for a special application, such as a particular kind of transmission protocol or a hand-held computer. We can contrast it with general integrated circuits, such as the microprocessor and the random access memory chips in your PC. ASICs are used in a wide-range of applications, including auto emission control, environmental monitoring, and personal digital assistants (PDAs).
An ASIC can be pre-manufactured for a special application or it can be custom manufactured (typically using components from a "building block" library of components) for a particular customer application.

2.3 Verilog

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL).A hardware description language is a language used to describe a digital system.Such as:

- Network switch
- Processor and microprocessor
- Memory etc.

Using Verilog one can describe any digital hardware and also simulate to see result Every design, regardless of complexity, was designed through schematics. They were difficult to verify and error-prone, resulting in long, tedious development cycles of design.The Verilog design cycle is more like a traditional programming one.

2.3.1 Verilog design styles

Bottom up approach:

More traditional approach Each design is performed at the gate-level using the standard gates With more complexity becomes hard to follow.

Top down approach :

It is the desired design-style A real top-down design allows early testing, easy change of different technologies.Most designs are a mix of both methods, implementing some key elements of both design styles.

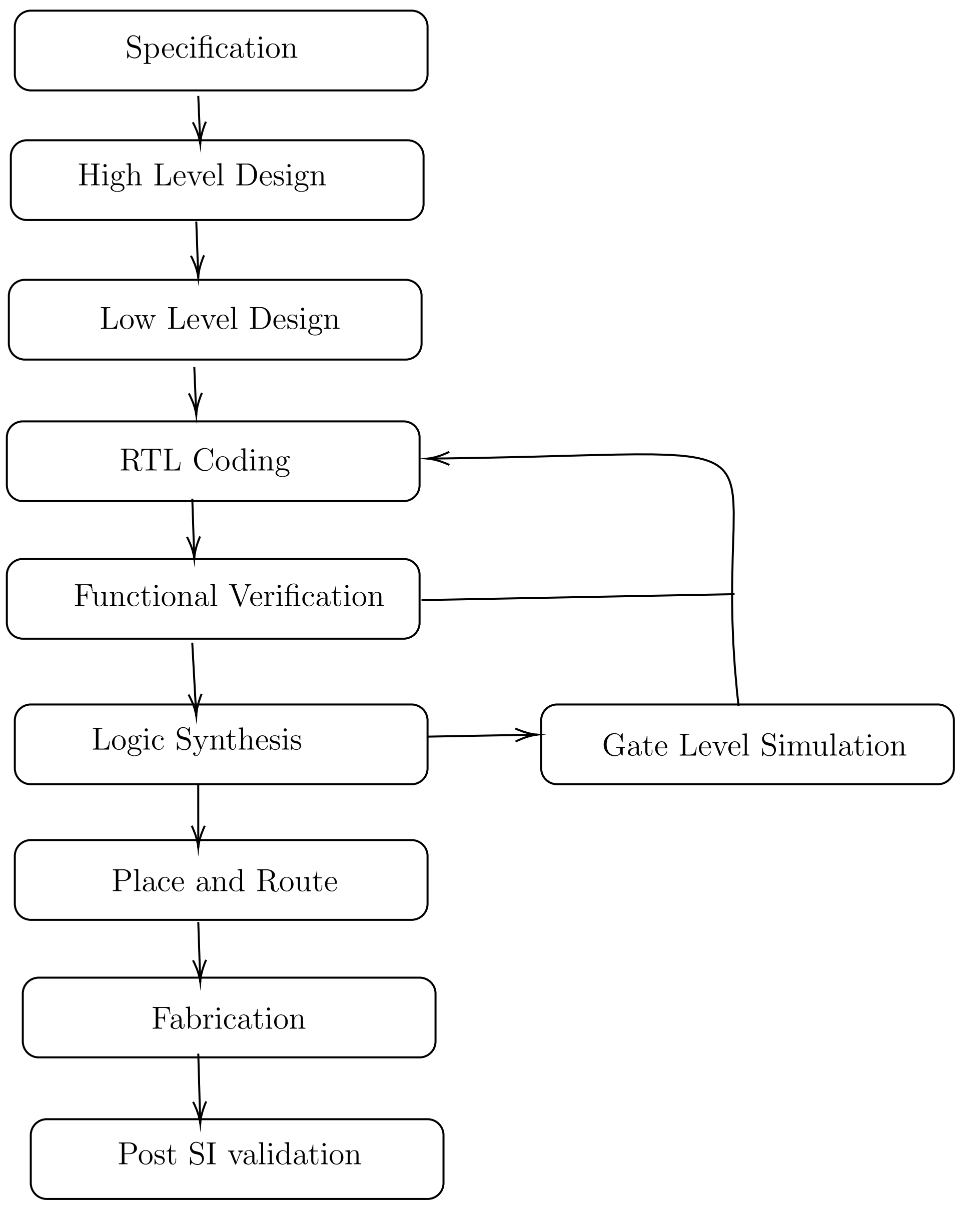


Figure 2.1: Steps of Top down Approach

2.3.2 Verilog Abstraction Level

Verilog supports designing at many different levels of abstraction.
3 of them are very important.

- Behavioral level
- Register-Transfer Level
- Gate Level

Behavioral level Coding:

This level describes a system by concurrent algorithms
Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other.
Functions, Tasks and Always blocks are the main elements.
There is no regard to the structural realization of the design.

Register transfer level Coding :

Designed to specify the characteristics of a circuit by operations and the transfer of data between the registers.An explicit clock is used. RTL design contains exact timing bounds .Any code that is synthesizable is RTL code

Gate level Coding :

Within the logic level the characteristics of a system are described by logical links and their timing properties.Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools which is called Netlist .Netlist is used for gate level simulation and for backend.

2.3.3 Simulation

Simulation is the process of verifying the functional characteristics of models at any level of abstraction.We use simulators to simulate the Hardware models. To test if the RTL code meets the functional requirements of the specification.To achieve this we need to write a testbench, which generates clk, reset and the required test vectors.Normally 60-70 % of time is spent in design verification.

2.3.4 Synthesis

Synthesis is the process in which the RTL code is transformed into hardware design using gates, mux, flip-flops etc.After synthesis we can see how our code looks inn terms of logic gates and physical devices.Synthesis can be done by synthesis tools not compiler. Like: Vivado hls.

2.3.5 Designing FSM

FSM is the main brain of any module.A good designed FSM is essential to code efficiently.A very good understanding of the problem is necessary before designing.An worked out example of the problem is very helpful towards designing an effective FSM.
Care should be given towards designing the default state in case of any anomaly in the program .Be careful about the register lengths and mention their changes in the state diagram. Too much details should be avoided in the diagram.

FSM Coding in Verilog :

It is a bad practice to use loops in RTL Coding, most cases it is unsupported and avoided. Instead FSM are designed and Coded for RTL Coding.Normally in any FSM there are two parts
(1)Sequential Logic part :
Sequential logic's output depends not only on the present value of its input signals but on the sequence of past inputs and the input history as well.It is executed sequentially.This part calculates both Register value changes and Output calculation
(2)Combinational Logic part :
Combinational Logic Circuits are memory less digital logic circuits whose output at any instant in time depends only on the combination of its inputs.It is normally the calculation of next state.It is very important that any asynchronous signal to the next state logic be synchronized before being fed to the FSM.This means waiting for sequential logic part to be completed before feeding the next state logic ultimaely synchronizing the whole process.

2.3.6 Writing Test bench

Writing a test bench is as important and complex as writing the RTL code itself.Main purpose of the test bench is to check outputs(validation).Before writing test bench a good understanding of different outputs the program might generate is necessary.Determine different test scenarios should be generated.
Here we mainly write behavioral code for the program (it is similar to 'C' language) and compute the output.In other part we pass the input to RTL program and wait for its Completion . At last check both Outputs.We try to perform test on wide range of input.
Test bench are also written for checking the outputs manually. Three different functions can be used to observe the outputs (1) $display (2) $monitor (3) $strobe

Test bench Coding :

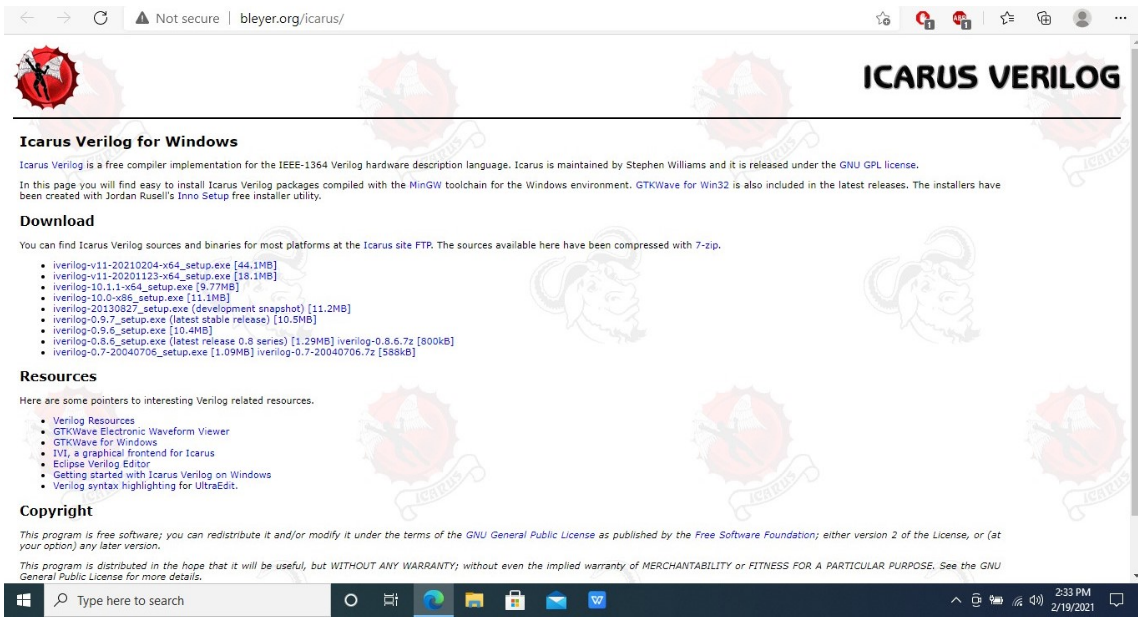
First declare all the output and input variables needed for the testing program.Then we generate the clock and declare the modules with appropriate input and output. Then write the reset logic FSM should start or stop working depending on the reset value it is normally in an initial block.After that start passing input and check how many clock cycles it needs to complete task.
We check the output manually using $monitor function at first.If it is ok then write c-like code for the program and inside a loop run the both c-like and RTL code and automatically match their output and display result.The c-like code can generate output instantly but RTL takes some time() .So maintain delay accordingly before checking both outputs.

2.4 Verilog Compiler (Icarus)

There are many Compiler to compile verilog for free Icarus is one of them which supports latest verilog constructs It can also compile both RTL and Behavioral codes

2.4.1 Where to download

Icarus is also low in size can be easily used from command prompt. GTKWAVE is a software to observe the waveforms of variables and clock which is included with the Icarus .Icarus can be downloaded from `http://bleyer.org/icarus/`



Icarus Verilog for Windows

Icarus Verilog is a free compiler implementation for the IEEE-1364 Verilog hardware description language. Icarus is maintained by Stephen Williams and it is released under the GNU GPL license.

In this page you will find easy to install Icarus Verilog packages compiled with the MinGW toolchain for the Windows environment. GTKWave for Win32 is also included in the latest releases. The installers have been created with Jordan Rusell's Inno Setup free installer utility.

Download

You can find Icarus Verilog sources and binaries for most platforms at the Icarus site FTP. The sources available here have been compressed with 7-zip.

- iverilog-v11-20210204-x64_setup.exe [44.1MB]
- iverilog-v11-20201123-x64_setup.exe [18.1MB]
- iverilog-10.1.1-x64_setup.exe [9.77MB]
- iverilog-10.0-x86_setup.exe [11.1MB]
- iverilog-20130827_setup.exe (development snapshot) [11.2MB]
- iverilog-0.9.7_setup.exe (latest stable release) [10.5MB]
- iverilog-0.9.6_setup.exe [10.4MB]
- iverilog-0.8.6_setup.exe (latest release 0.8 series) [1.29MB] iverilog-0.8.6.7z [800kB]
- iverilog-0.7-20040706_setup.exe [1.09MB] iverilog-0.7-20040706.7z [588kB]

Resources

Here are some pointers to interesting Verilog related resources.

- Verilog Resources
- GTKWave Electronic Waveform Viewer
- GTKWave for Windows
- IVI, a graphical frontend for Icarus
- Eclipse Verilog Editor
- Getting started with Icarus Verilog on Windows
- Verilog syntax highlighting for UltraEdit.

Copyright

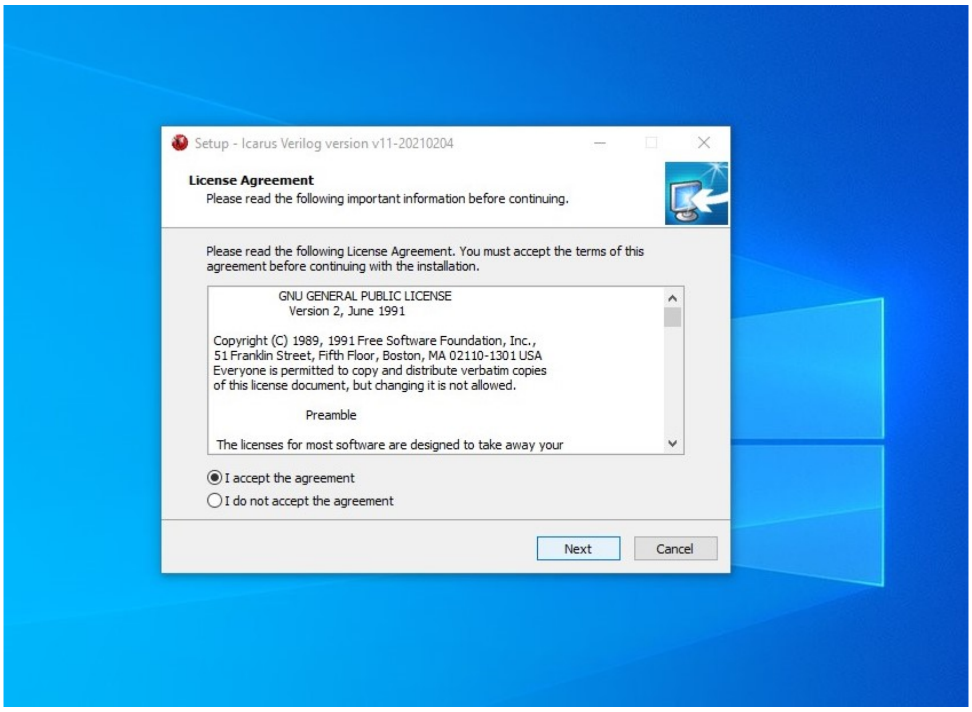
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Figure 2.2: Icarus Compiler home page

2.4.2 How to setup environment

After downloading Icarus it have has to be installed properly.Video demonstration for Icarus environment setup is available here `https://www.youtube.com/watch?v=3Xm6fgKAO94`



License Agreement
Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

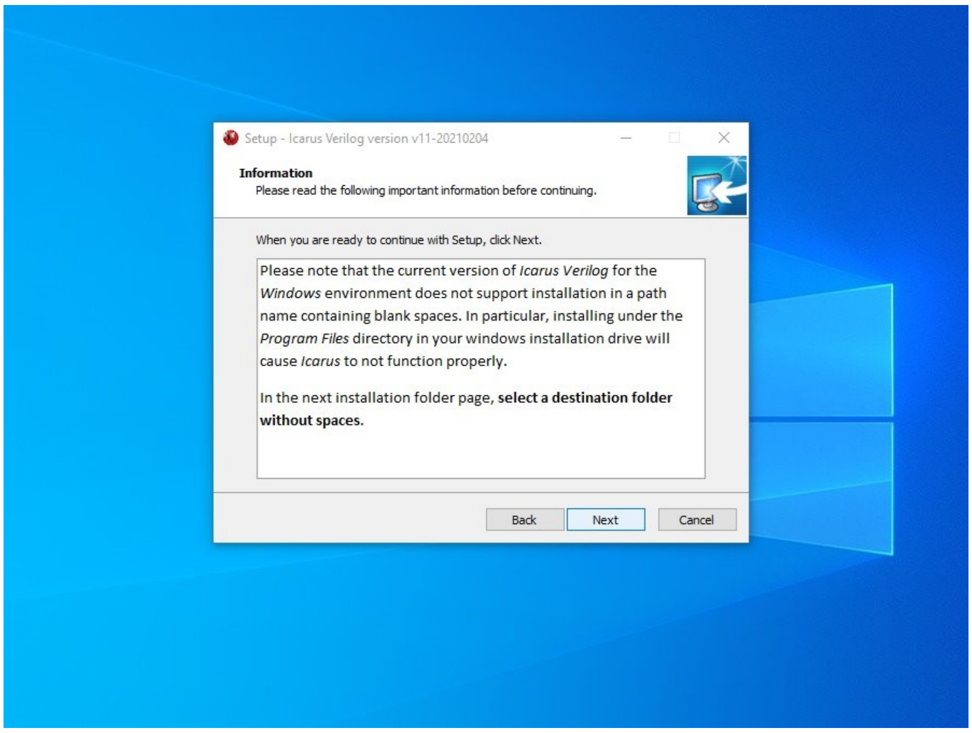
Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your

[✓ I accept the agreement]
I do not accept the agreement

Figure 2.3: Icarus Compiler installation step 1



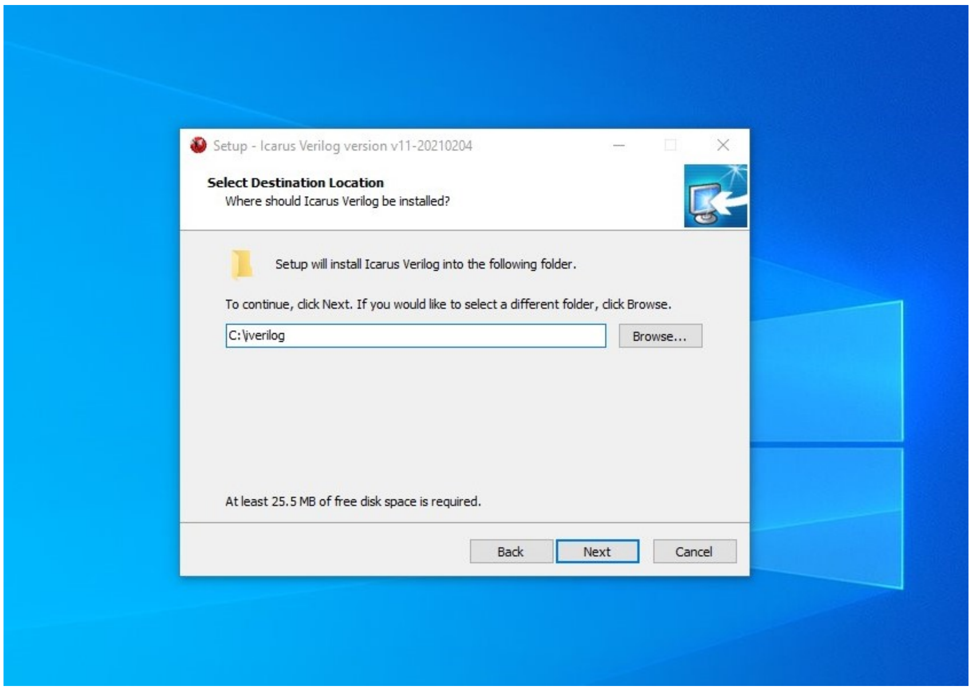
Information
Please read the following important information before continuing.

When you are ready to continue with Setup, click Next.

Please note that the current version of *Icarus Verilog* for the *Windows* environment does not support installation in a path name containing blank spaces. In particular, installing under the *Program Files* directory in your windows installation drive will cause *Icarus* to not function properly.

In the next installation folder page, **select a destination folder without spaces.**

Figure 2.4: Icarus Compiler installation step 2



Setup - Icarus Verilog version v11-20210204

Select Destination Location

Where should Icarus Verilog be installed?

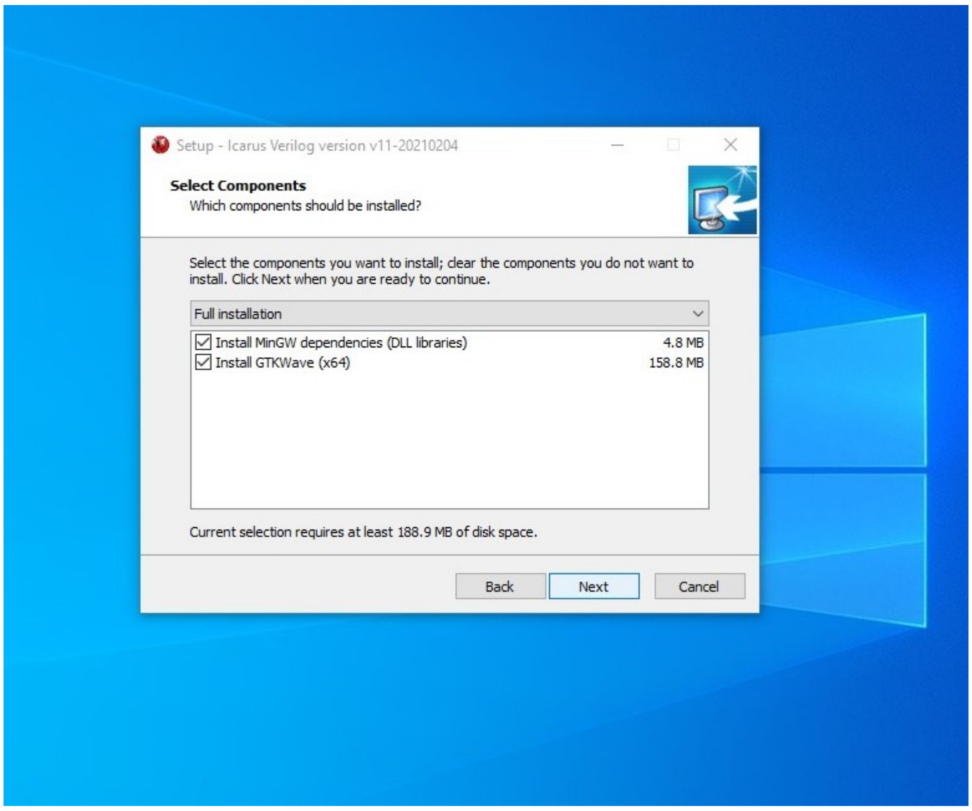
Setup will install Icarus Verilog into the following folder.

To continue, click Next. If you would like to select a different folder, click Browse.

C:\iverilog

At least 25.5 MB of free disk space is required.

Figure 2.5: Icarus Compiler installation step 3



Setup - Icarus Verilog version v11-20210204

Select Components

Which components should be installed?

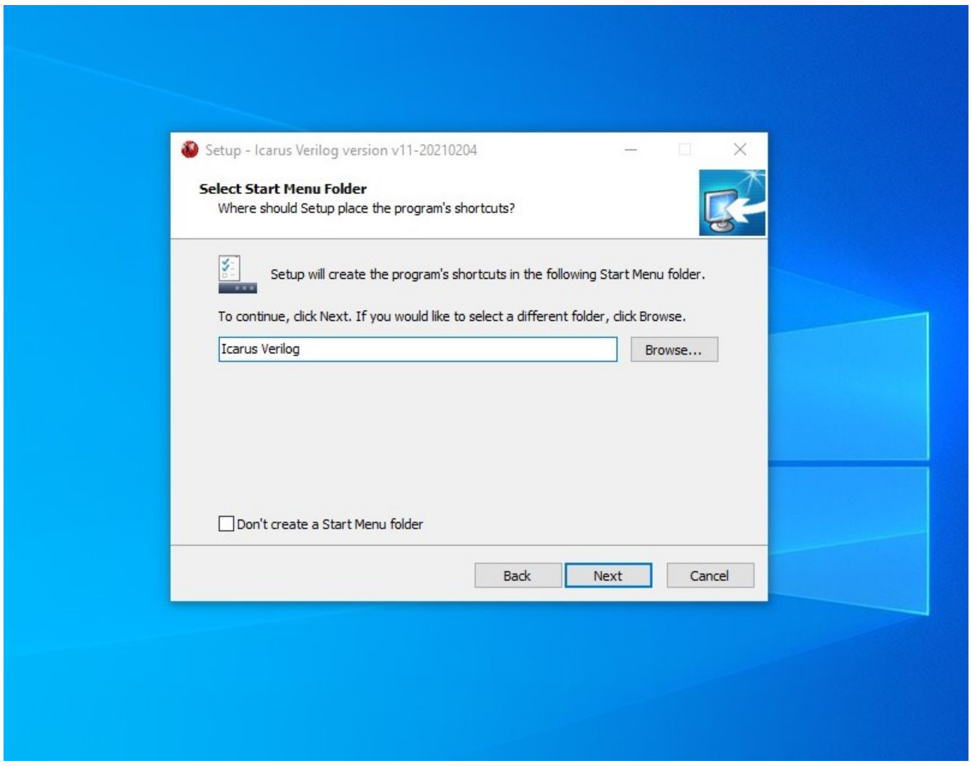
Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

Full installation

☑ Install MinGW dependencies (DLL libraries)	4.8 MB
☑ Install GTKWave (x64)	158.8 MB

Current selection requires at least 188.9 MB of disk space.

Figure 2.6: Icarus Compiler installation step 4


Setup - Icarus Verilog version v11-20210204

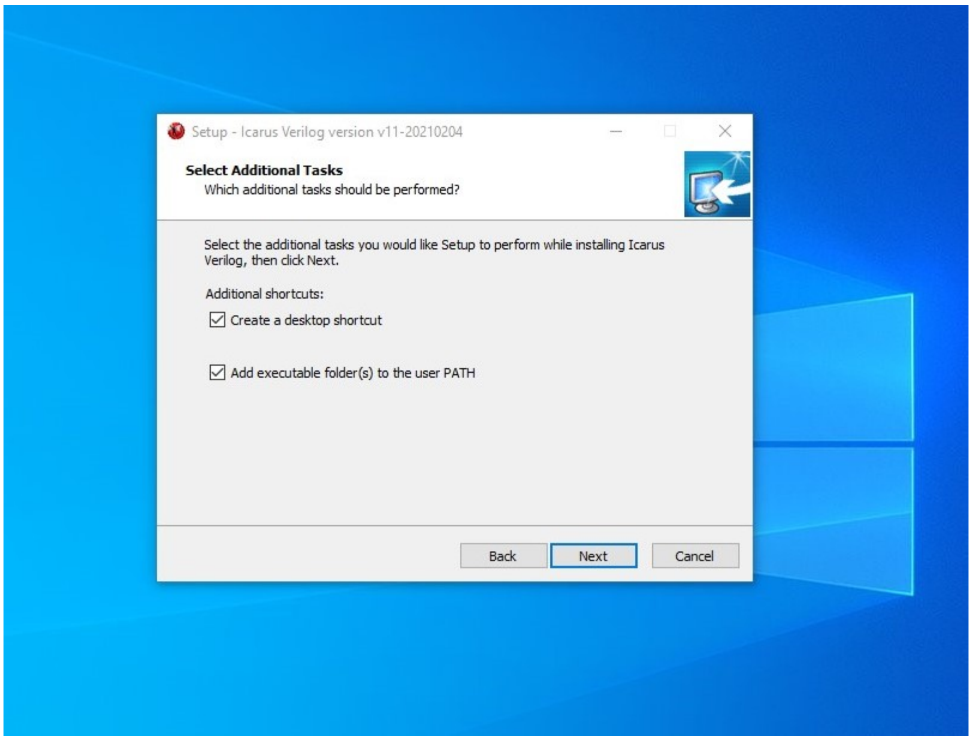
Select Start Menu Folder
Where should Setup place the program's shortcuts?

Setup will create the program's shortcuts in the following Start Menu folder.

To continue, click Next. If you would like to select a different folder, click Browse.

Icarus Verilog

Figure 2.7: Icarus Compiler installation step 5


Setup - Icarus Verilog version v11-20210204

Select Additional Tasks
Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing Icarus Verilog, then click Next.

Additional shortcuts:

☑ Create a desktop shortcut

☑ Add executable folder(s) to the user PATH

Figure 2.8: Icarus Compiler installation step 6

Chapter 3

Design Methodology

Floating-Point Design-Flow Methodology in verilog: Hardware is typically designed using an HDL, either Verilog or VHDL. These languages are fairly low level. This languages also requires the designer to specify all data widths at each stage and specify each register level.Although some higher level languages support foat variables but they does not support the synthesis of floating-point arithmetic operations.
To implement the IEEE-754 using HDL would be very arduous. Because of this the use of floating-point processing in hardware-based designs has been greatly discouraged.

3.1 Design Principles and Conditions

To design a floating point converter we have kept the following principles in mind.

1. Collaborate on the design: Our goal is to design a floating point converter. which could be translated into hardware later on added onto a embedded system.

2. Adhere to industry standards: The design and the code should follow industry standards. The design also should follow hardware standards for lower level systems.

3. Synchronous : The design should be synchronous to one or more clocks.

4. Modular : The design should be modular. So that it could be integrated as a whole system or any of its module could be integrated to any system.

5. Continuous : The design should be able to take continuous inputs and give output with minimum delay.

6. RTL Code: The design should easily be translated into RTL codes

3.2 Design of the Floating Point Converter

The whole converter design can be divided into 4 parts.

1. Design of fractional decimal to Binary Converter
2. Design of sign bit module
3. Design of Exponent module
4. Design of Exponent module

3.3 Design of fractional decimal to Binary Converter

Real numbers can be divided into 2 parts. Integer and Fractional part. Integer part can easily be converted to binary. But for fractional part a new algorithm should be devised.

3.3.1 Inputs and Outputs

As verilog cant handle floats, Here input is the integer of the fractional value. Output will be 8 bit binary of the fractional value

3.3.2 Numerical Examples

An worked out Example:

$value = .35$
$Programinput = 35$
$Result = [0] * 8$

$Start:$
$i = 0$
$Data = 35$
$Data = 35 * 2 = 70$
$Result[i] = 70/100 = 0$
$Data = 70\%100 = 70$

$i = 1:$
$Data = 70$
$Data = 70 * 2 = 140$
$Result[i] = 140/100 = 1$
$Data = 140\%100 = 40$

$i = 2:$
$Data = 40$
$Data = 40 * 2 = 80$
$Result[i] = 80/100 = 0$
$Data = 80\%100 = 80$

$i = 3:$
$Data = 80$
$Data = 80 * 2 = 160$
$Result[i] = 160/100 = 1$
$Data = 160\%100 = 60$

$i = 4:$
$Data = 60$
$Data = 60 * 2 = 120$
$Result[i] = 120/100 = 1$
$Data = 120\%100 = 20$

3.3.3 Algorithm

Algorithm 1 shows how to convert fractional decimal to fractional binary.

Algorithm 1 Algorithm for Fractional Binary conversion

```
$data \leftarrow binaryof(input)$
$count \leftarrow 0$
$result \leftarrow' 00000000'$
while $count < 8$ do
    $data \leftarrow data * 2$
    $result[count] \leftarrow data/100$
    $data \leftarrow data\%100$
    if $data == 0$ then
        $Break$
    else
        $count \leftarrow count + 1$
    end if
end while
```

3.3.4 Finite State Machine

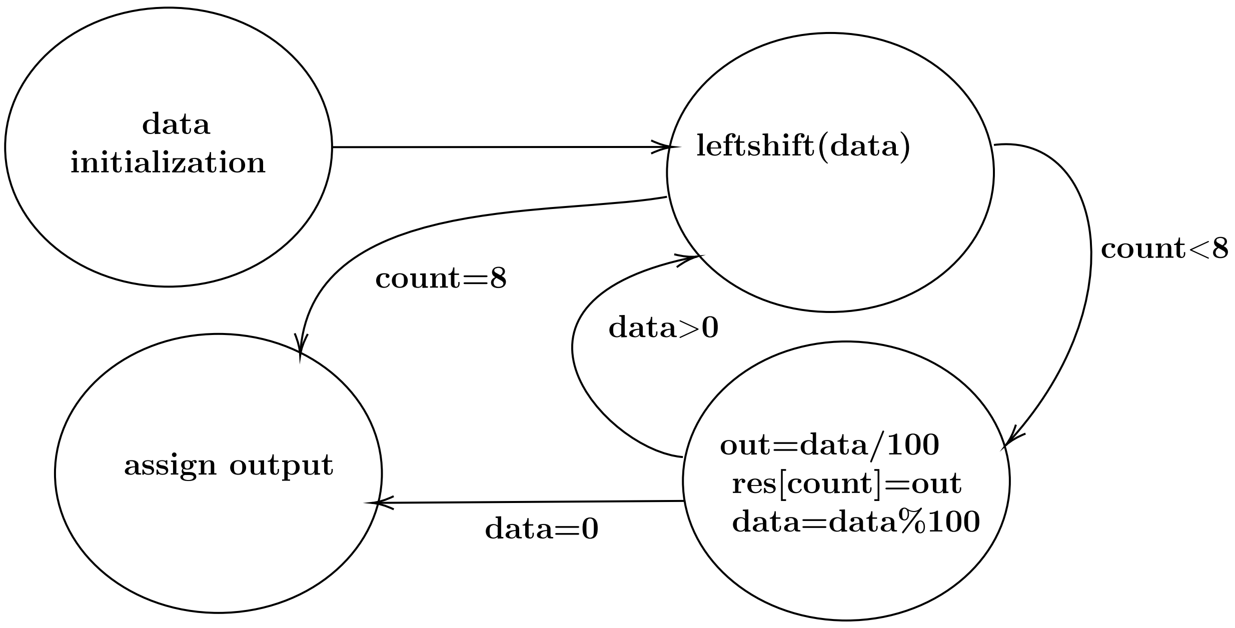


Figure 3.1: State diagram of fractional decimal to binary conversion

3.4 Design of sign bit module

Sign bit is the first bit of IEEE-754 representation. it is of 1 bit length. if the number is positive then the value is 1, otherwise the value becomes 0.

3.4.1 Inputs and Outputs

Here input is the real number and output is sign bit. which is very easy to calculate.

3.4.2 Numerical Example

An Worked out Example:
$real\_number = 3.5$
$integer\_part = 3$
$fractional\_part = 50$
$integer\_part > 0$
$so, \ sign\_bit = 0$

3.4.3 Algorithm

Algorithm 2 Algorithm for Sign Bit Calculation

```
if $integer\_part > 0$ then
    $Sign\_bit \leftarrow 0$
else
    $Sign\_bit \leftarrow 1$
end if
```

Algorithm 2 shows sign bit calculation.

3.4.4 Finite State Machine

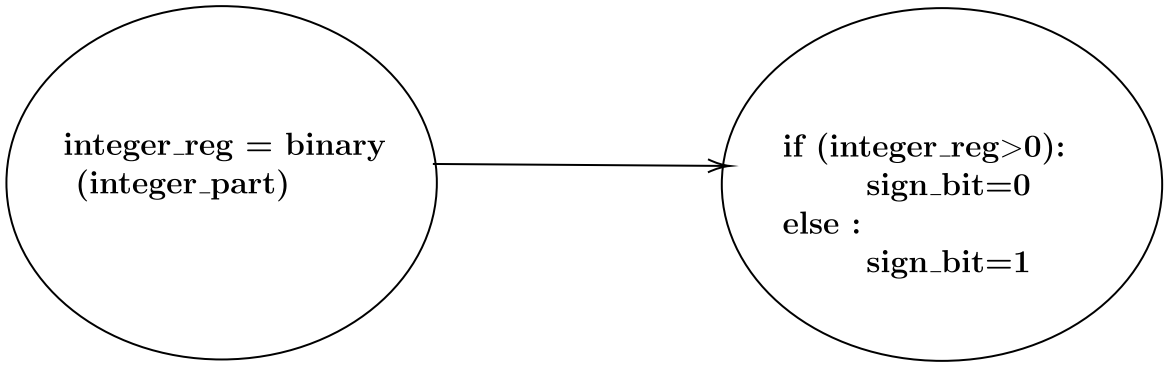


Figure 3.2: State diagram for sign bit calculation

3.5 Design of the Mantissa Module

Mantissa is the largest part which is of 23 bits long.

3.5.1 Inputs and Outputs

Here, the input will be the real number and output will be 23 bit mantissa.

3.5.2 Numerical Example

An Worked out Example:
$real\_number = 9.39$
$binary = 1001.01100011$
$first\ 1\ is\ situated\ at\ 4th\ place\ from\ the\ decimal$
$so\ leftshift\ (4-1) = 3\ times$
$after\ left\ shift\ binary = 1.00101100011000000000000$
$mantissa = 00101100011000000000000$

3.5.3 Algorithm

Algorithm 3 Algorithm for mantissa calculation

```
$integer \leftarrow binary(integer\_part)$
$fraction \leftarrow binary(fractional\_part)$
$Mantissa \leftarrow \{integer, fraction\}$
while $mantissa[0] \neq 1$ do
    $mantissa \leftarrow leftshift(mantissa)$
    $i \leftarrow i + 1$
end while
$output \leftarrow leftshift(mantissa)$
```

Algorithm 3 shows mantissa calculation.

3.5.4 Finite State Machine

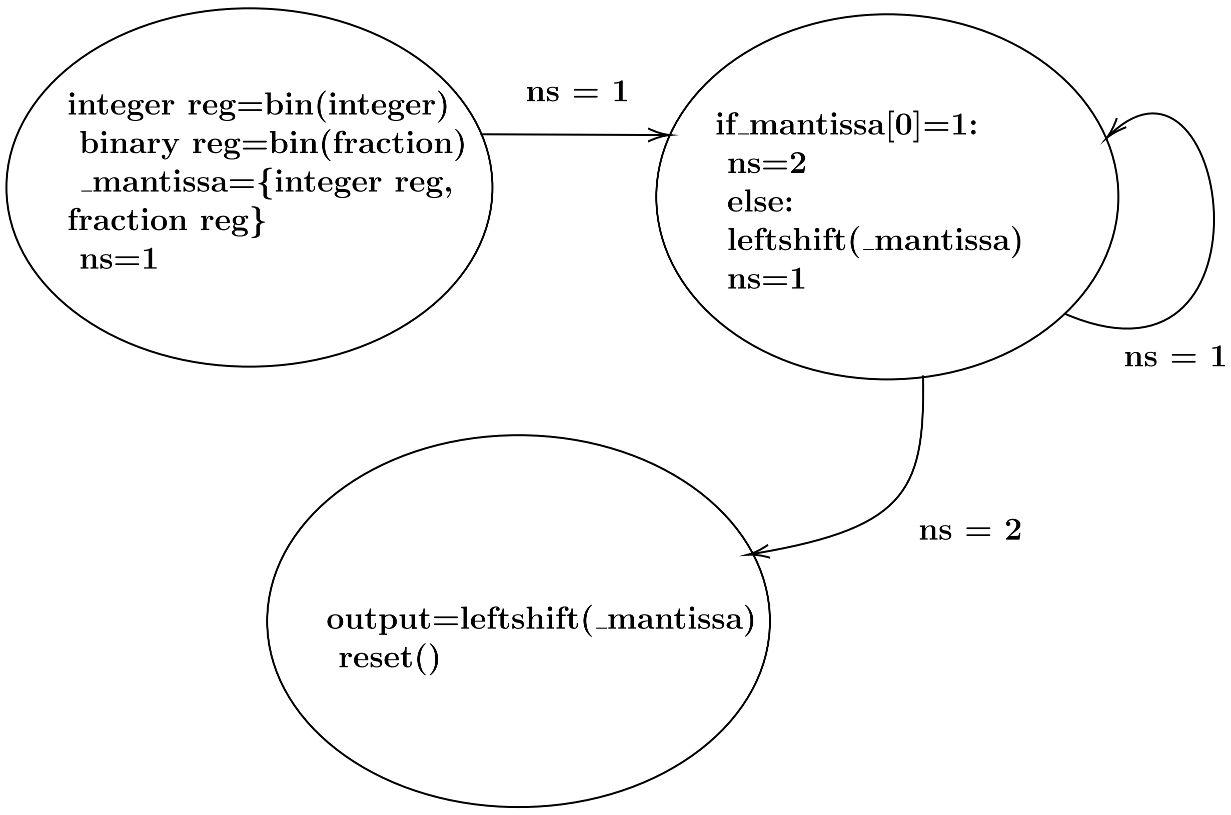


Figure 3.3: state diagram of mantissa calculation

3.6 Design of the Exponent Module

Exponent is of 8 bit which can be calculated from either integer part or fractional part.

3.6.1 Inputs and Outputs

Here, the input will be the real number and output will be 8 bit exponent.

3.6.2 Numerical Example

An Worked Out Example:

$real\_number = 9.39$
$binary = 1001.01100011$
$first\ 1\ is\ situated\ at\ 4th\ place\ from\ the\ decimal$
$so\ leftshift\ (4-1) = 3\ times$
$after\ left\ sihift\ binary = 1.00101100011 * 2^3$
$exponent = 127 + 3 = 130$

3.6.3 Algorithm

Algorithm 4 Algorithm for Exponent Calculation

```
$count \leftarrow 0$
**if** $integer\_part \neq 0$ **then**
    **while** $integer\_part[count] \neq 1$ **do**
        $count \leftarrow count + 1$
    **end while**
    $exponent \leftarrow 127 + count$
**else**
    **while** $fractional\_part[count] \neq 1$ **do**
        $count \leftarrow count + 1$
    **end while**
    $exponent \leftarrow 127 - count - 1$
**end if**
```

Algorithm 4 shows how to calculate exponent.

3.6.4 Finite State Machine

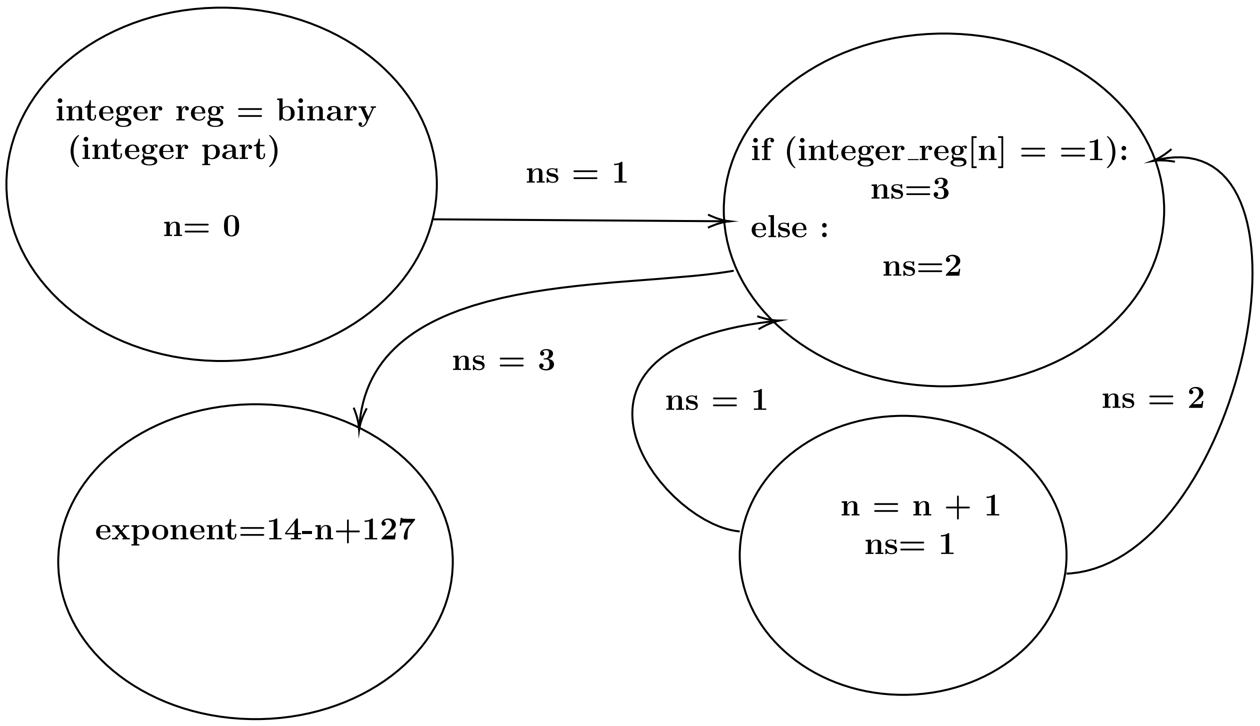


Figure 3.4: State diagram of exponent calculation when integer part is not 0

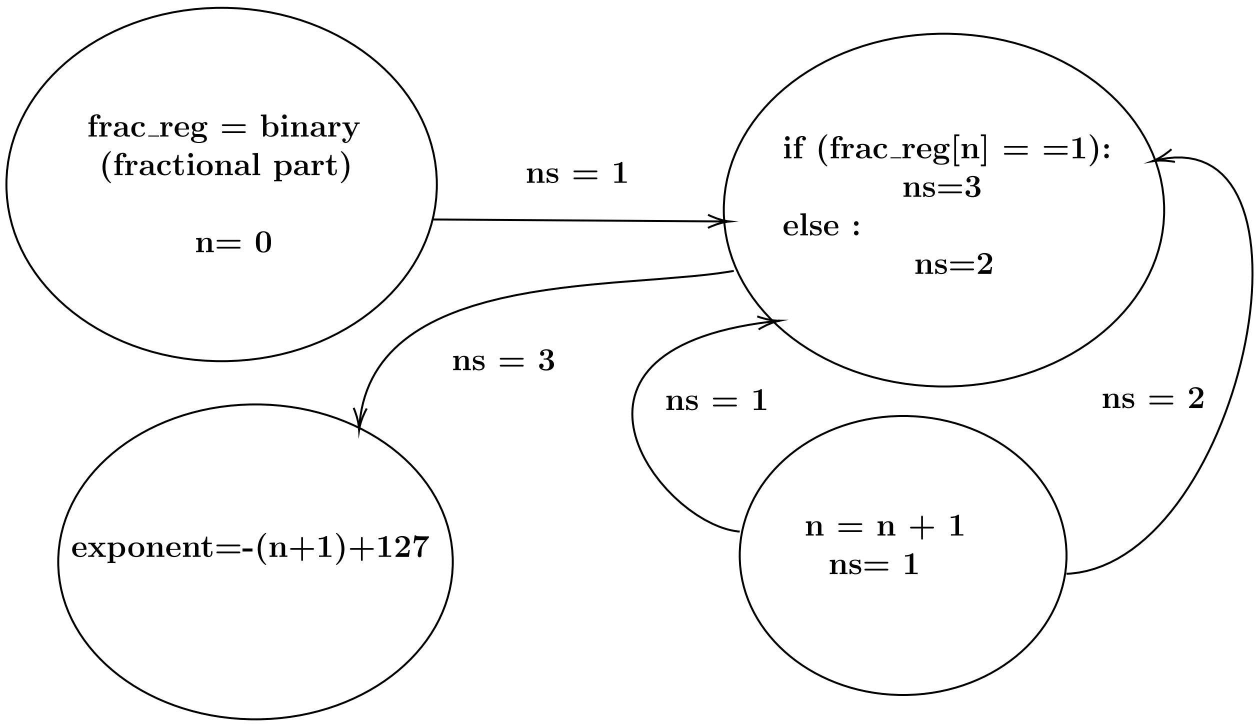


Figure 3.5: State diagram of exponent calculation when integer part is 0

Chapter 4

Implementation and Results

4.1 Converting fractional part to Binary

This module will convert fractional part of the real number into binary. Here our Input range is .01 to .99.

Input is an integer of the fractional value.

- For .5 input is 50
- For .05 input is 5

Output is 8 bit binary of fractional value

4.1.1 Pseudo-code

Here is the Pseudo-code for converting Fractional numbers into Binary for our module

```
$data \leftarrow binary(input)$
$count \leftarrow 0$
$result[0:7] \leftarrow 8'b00000000$
while $count < 8$ do
    $data \leftarrow data * 2$
    $result[count] \leftarrow data/100$
    $data \leftarrow data\%100$
    if $data == 0$ then
        $Break$
    else
        $count \leftarrow count + 1$
    end if
end while
```

4.1.2 Input Output Port Declaration

This Module has 3 input ports and 2 output ports. Details of them are given below.

- **Clock:** Input Clock signal
- **1 bit input valid:** Input Check whether the input is valid or invalid. If valid module starts conversion.
- **8 bit fraction part :** Input Binary of integer value of the fractional part
- **8 bit binary of fraction value:** Output binary of decimal fraction
- **1 bit output done flag:** This bit is set when calculation is completed and output is ready.

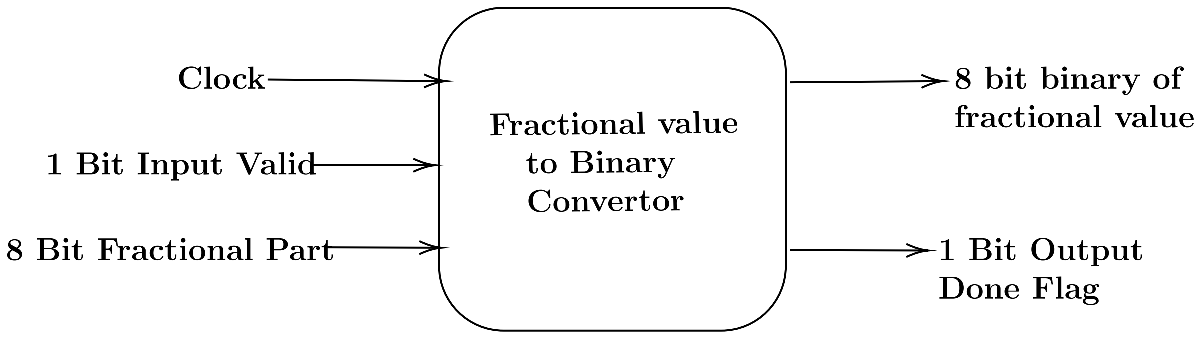

Clock
1 Bit Input Valid
8 Bit Fractional Part
Fractional value to Binary Convertor
8 bit binary of fractional value
1 Bit Output Done Flag

Figure 4.1: Fractional to binary module

4.1.3 Code

Module Declaration

```
/* this module converts data register value
to equivalent fractional value in binary */

module frac_bin
(
    input clk,
    input input_valid,
    input [7:0] data, // fractional part integer value
    output [0:7] frac,
    output output_valid
 );
```

Data Initialization

```
reg [1:0] ns; // temporary holds next state value
reg [1:0] next_state; // holds next state value for FSM
reg result; // 1 bit
reg [0:7] result_reg; // temp output
reg [0:7] output_reg; // output register
reg [7:0] data_reg; // input register
reg [3:0] count; // 4 bit
reg output_valid_reg; // output done flag
reg input_valid_reg; // input taken flag

// for reset we are initializing values
always @(posedge input_valid) begin
    output_valid_reg = 1'b0;
    input_valid_reg = 1'b1;
    data_reg = data; // input reg
    ns = 2'b00; // next state
end
```

FSM

```
always @(ns) begin // state transition
    next_state = ns;
end

always @(posedge clk) begin
    if(input_valid_reg)begin
        case (next_state)
            2'b00:begin
                result_reg = 8'b0; // result register initialization
                count = 0; // count
                ns = 2'b01; // next_state
            end
            2'b01:begin
            // output is 8 bit so maximum 8 times
                if (count<8) begin
                    data_reg = {data_reg << 1};
                    // data = data * 2
                    count = count + 1;
                    ns = 2'b10; // next state
                end
                else begin
                    ns = 2'b11; // transition to output state
                end
            end
            2'b10:begin
                result = data_reg / 100;
                result_reg[count-1] = result;
                data_reg = data_reg % 100;
                if (data_reg == 0)begin // if 0, go to output state
                    ns = 2'b11;
                end
                else
                    ns = 2'b01;
            end
            2'b11:begin
                output_reg = result_reg;
                output_valid_reg = 1'b1;
                input_valid_reg = 1'b0; // output state
                ns = 2'bxx;
            end
            2'bxx:begin
            end
        endcase
    end
end
```

Output assignment

```
// output assignment
assign frac = output_reg;
assign output_valid = output_valid_reg;
endmodule
```

4.1.4 Test-bench

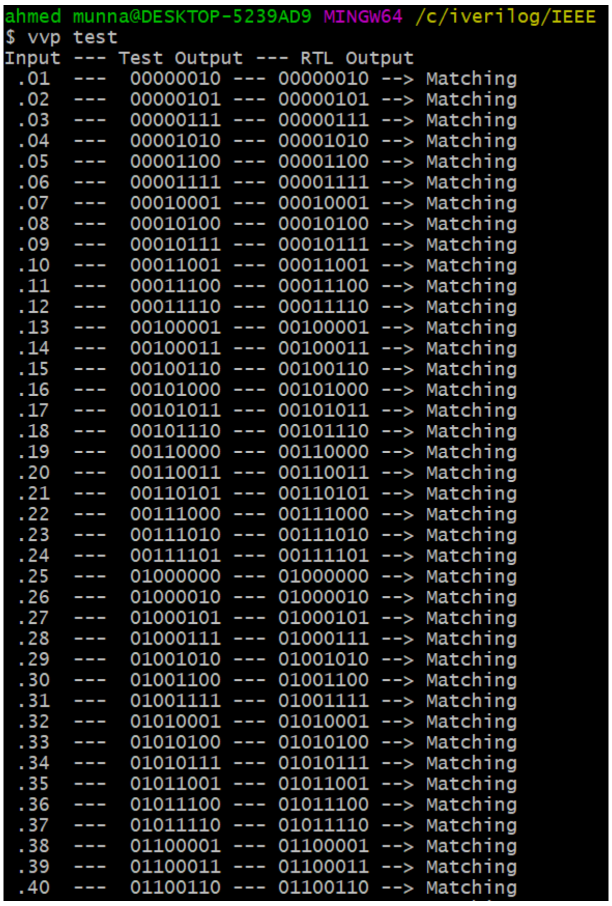
This test bench only Test this module to check its accuracy and performance. The Code Runs the algorithm with C- like code and RTL Code Then matches the both outputs and shows result.

```
// test bench module for frac_bin module

`include "bin2.v"

module frac_bin_tb(); // testbench
reg clock , input_valid;
reg [7:0] data = 0; // fractional part input
wire [0:7] frac;
wire output_valid; // output

// loop variables
integer i;
integer j;

reg [0:7] test_output;
reg [7:0] test_data;
reg matched; // for checking both outputs
```

```
initial begin
  clock =  1;

  $display("Input␣---␣Test␣Output␣---␣RTL␣Output");

  // code for test output
  for (i = 1; i<100; i = i + 1 ) begin
      test_output = 8'b0;
      test_data = i;
      for(j = 0; j<8; j = j + 1)begin
        test_data = test_data * 2;
        test_output[j] = test_data / 100;
        test_data = test_data % 100;
      end

      // Running Rtl module
      #2 input_valid = 0;
      #1 data = data + 1;
      #2 input_valid = 1;
      // waiting for rtl to finish
      #100
      //displaying and comparing both outputs
      if (test_output == frac) matched = 1'b1;
      else matched = 1'b0;
      if(i<10)
        if(matched)
          $display("␣.0%0d␣␣---␣␣%b␣---␣%b␣-->␣Matching",
          i,test_output,frac);
        else
          $display("␣.0%0d␣␣---␣␣%b␣---␣%b␣-->␣Not␣Matching",
          i,test_output,frac);
      else
        if(matched)
          $display("␣.%0d␣␣---␣␣%b␣---␣%b␣-->␣Matching",
          i,test_output,frac);
        else
          $display("␣.%0d␣␣---␣␣%b␣---␣%b␣-->␣Not␣Matching",
          i,test_output,frac);
  end
  #100 $finish;
end

always
#2 clock = ~clock;

frac_bin U_frac_bin(
  clock , // Clock
```

```
    input_valid  , // reset
    data,   // input value
    frac, // output reg
    output_valid
);
endmodule
```

4.1.5 Result and Output



```
ahmed munna@DESKTOP-5239AD9 MINGW64 /c/iverilog/IEEE
$ vvp test
Input --- Test Output --- RTL Output
 .01  ---  00000010 --- 00000010 --> Matching
 .02  ---  00000101 --- 00000101 --> Matching
 .03  ---  00000111 --- 00000111 --> Matching
 .04  ---  00001010 --- 00001010 --> Matching
 .05  ---  00001100 --- 00001100 --> Matching
 .06  ---  00001111 --- 00001111 --> Matching
 .07  ---  00010001 --- 00010001 --> Matching
 .08  ---  00010100 --- 00010100 --> Matching
 .09  ---  00010111 --- 00010111 --> Matching
 .10  ---  00011001 --- 00011001 --> Matching
 .11  ---  00011100 --- 00011100 --> Matching
 .12  ---  00011110 --- 00011110 --> Matching
 .13  ---  00100001 --- 00100001 --> Matching
 .14  ---  00100011 --- 00100011 --> Matching
 .15  ---  00100110 --- 00100110 --> Matching
 .16  ---  00101000 --- 00101000 --> Matching
 .17  ---  00101011 --- 00101011 --> Matching
 .18  ---  00101110 --- 00101110 --> Matching
 .19  ---  00110000 --- 00110000 --> Matching
 .20  ---  00110011 --- 00110011 --> Matching
 .21  ---  00110101 --- 00110101 --> Matching
 .22  ---  00111000 --- 00111000 --> Matching
 .23  ---  00111010 --- 00111010 --> Matching
 .24  ---  00111101 --- 00111101 --> Matching
 .25  ---  01000000 --- 01000000 --> Matching
 .26  ---  01000010 --- 01000010 --> Matching
 .27  ---  01000101 --- 01000101 --> Matching
 .28  ---  01000111 --- 01000111 --> Matching
 .29  ---  01001010 --- 01001010 --> Matching
 .30  ---  01001100 --- 01001100 --> Matching
 .31  ---  01001111 --- 01001111 --> Matching
 .32  ---  01010001 --- 01010001 --> Matching
 .33  ---  01010100 --- 01010100 --> Matching
 .34  ---  01010111 --- 01010111 --> Matching
 .35  ---  01011001 --- 01011001 --> Matching
 .36  ---  01011100 --- 01011100 --> Matching
 .37  ---  01011110 --- 01011110 --> Matching
 .38  ---  01100001 --- 01100001 --> Matching
 .39  ---  01100011 --- 01100011 --> Matching
 .40  ---  01100110 --- 01100110 --> Matching
```

Figure 4.2: Comparison between RTL and Non RTL ouput

Figure 4.2 shows a Comparison between RTL and Non-RTL Code outputs. Here We can see that the the module gives a 100% correct result.

4.2 Sign Bit Calculation

This module compares the integer part value and calculates the sign bit. If the integer part is a negative number than it also convert it to a positive number which is required for our later calculations.

4.2.1 Pseudo-code

Here is the Pseudo-code for calculation of sign bit for the module.

```
if $integer\_part > 0$ then
    $Sign\_bit \leftarrow 0$
else
    $Sign\_bit \leftarrow 1$
    $integer\_part \leftarrow integer\_part * -1$
end if
```

4.2.2 Input Output Port Declaration

This Module has 3 input ports and 3 output ports. Details of them are given below.

- **Clock:** Input Clock signal
- **1 bit input valid:** Input Check whether the input is valid or invalid. If valid module starts conversion.
- **15 Bit integer :** 15 bit Binary of integer part
- **1 bit sign bit:** Output sign bit.
- **15 bit positive integer:** Positive value of the input integer part.
- **1 bit output done flag:** This bit is set when calculation is completed and output is ready.

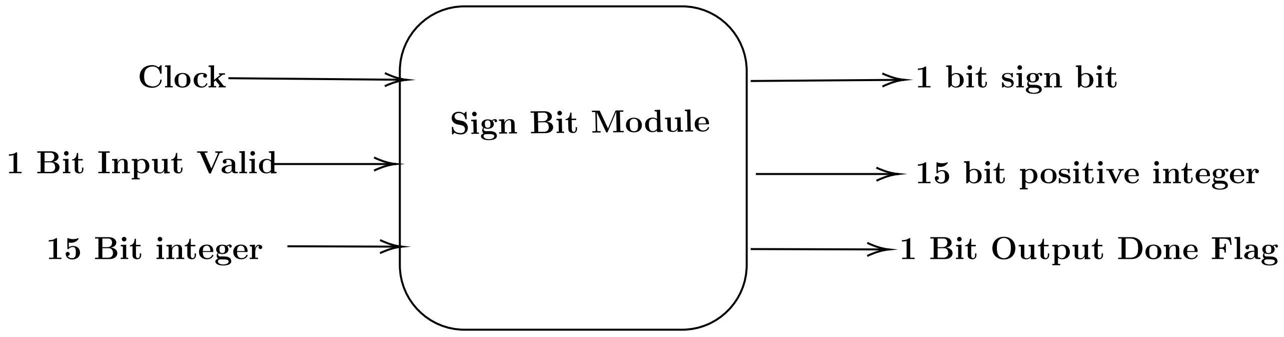


Figure 4.3: Fractional to binary module

4.2.3 Code

Module Declaration

```
module sign(
    input clk,
    input [14:0] intg,
    input input_valid,
    output sign_bit,
    output [14:0] intg_out,
    output output_valid
);

// registers for inputs and outputs
reg [14:0] intg_reg;
reg [14:0] intg_reg_out;

reg sign_bit_reg;
reg output_valid_reg = 1'b0;
```

Fsm and Output

```
// output calculation
always @(posedge input_valid) begin
  intg_reg = intg;
  // checking msb to determine sign

  if(intg[14]==0)begin
    sign_bit_reg = 1'b0;
    intg_reg_out = intg_reg;      //  absolute value of integer part
  end

  else begin
    sign_bit_reg = 1'b1;
    intg_reg_out = intg_reg * -1; // absolute value of integer part
  end

  output_valid_reg = 1'b1;

end


// output assignment
assign intg_out = intg_reg_out;
assign sign_bit = sign_bit_reg;
assign output_valid = output_valid_reg;


endmodule //
```

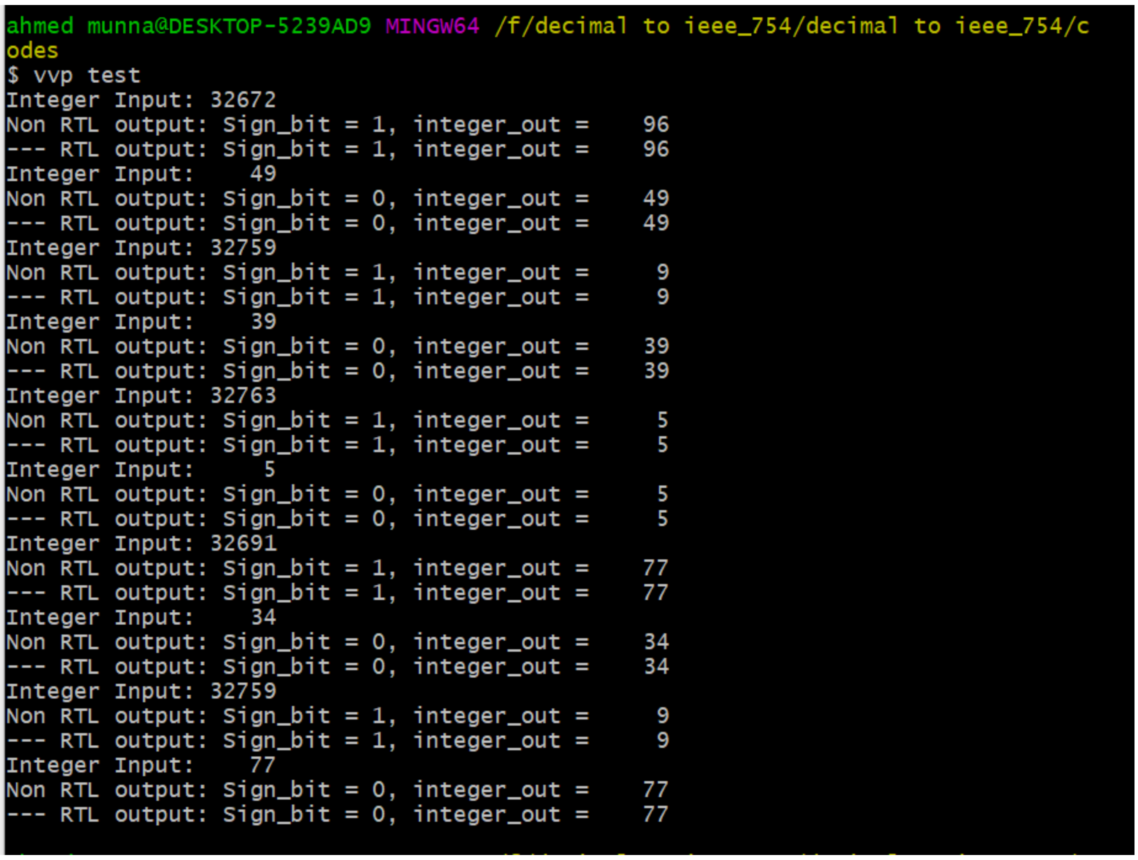
4.2.4 Test-bench

This test bench only Test this sign bit module to check its accuracy and performance. The Code Runs the algorithm with C- like code and RTL Code Then matches the both outputs and shows result.

```
`include "sign_bit.v"
module sign_bit_tb();
integer    seed,i,j;
reg clock , input_valid;
reg [14:0] intg;
reg [14:0] intg_f;

wire [14:0] intg_out;
wire sign_bit;
wire output_valid; // output


initial begin
    clock = 1;

    for (i=0; i<10; i=i+1)
    begin
        intg=$urandom%100;
        if (i%2 == 0) begin
            intg = intg * -1;
        end
        $display("Integer Input: %d",intg);
        if (intg[14]==0) begin
          $display("Non RTL output: Sign_bit = 0,
          integer_out = %d", intg);
        end
        else begin
          intg_f = intg * -1;
          $display("Non RTL output: Sign_bit = 1,
          integer_out = %d", intg_f);
        end
        #2 input_valid = 0;
        #2 input_valid = 1;
        #200
        $display("---- RTL output: Sign_bit = %d,
        integer_out = %d",sign_bit, intg_out);
    end
    $finish;
end
```

```
always
#2 clock = ~clock;

sign U_sign(
     clock,
     intg,
     input_valid,
     sign_bit,
     intg_out,
     output_valid
);
endmodule
```

4.2.5 Result and Output



```
ahmed munna@DESKTOP-5239AD9 MINGW64 /f/decimal to ieee_754/decimal to ieee_754/c
odes
$ vvp test
Integer Input: 32672
Non RTL output: Sign_bit = 1, integer_out =    96
--- RTL output: Sign_bit = 1, integer_out =    96
Integer Input:    49
Non RTL output: Sign_bit = 0, integer_out =    49
--- RTL output: Sign_bit = 0, integer_out =    49
Integer Input: 32759
Non RTL output: Sign_bit = 1, integer_out =     9
--- RTL output: Sign_bit = 1, integer_out =     9
Integer Input:    39
Non RTL output: Sign_bit = 0, integer_out =    39
--- RTL output: Sign_bit = 0, integer_out =    39
Integer Input: 32763
Non RTL output: Sign_bit = 1, integer_out =     5
--- RTL output: Sign_bit = 1, integer_out =     5
Integer Input:     5
Non RTL output: Sign_bit = 0, integer_out =     5
--- RTL output: Sign_bit = 0, integer_out =     5
Integer Input: 32691
Non RTL output: Sign_bit = 1, integer_out =    77
--- RTL output: Sign_bit = 1, integer_out =    77
Integer Input:    34
Non RTL output: Sign_bit = 0, integer_out =    34
--- RTL output: Sign_bit = 0, integer_out =    34
Integer Input: 32759
Non RTL output: Sign_bit = 1, integer_out =     9
--- RTL output: Sign_bit = 1, integer_out =     9
Integer Input:    77
Non RTL output: Sign_bit = 0, integer_out =    77
--- RTL output: Sign_bit = 0, integer_out =    77
```

Figure 4.4: Comparison between RTL and Non-RTL Code outputs for sign bit module

Figure 4.4 shows a Comparison between RTL and Non-RTL Code outputs. Here We can see that the the module gives a 100% correct result.

4.3 Mantissa Calculation

This module calculates the mantissa from the Integer and fractional part. The mantissa has 23 bits of length.

4.3.1 Pseudo-code

Here is the Pseudo-code for calculation of mantissa for the module.

```
$integer[0:14] \leftarrow binary(integer\_part)$
$fraction[0:7] \leftarrow binary(fractional\_part)$
$Mantissa[0:22] = \{integer, fraction\}$
while $mantissa[0] \neq 1$ do
  $mantissa \leftarrow leftshift(mantissa)$
  $i \leftarrow i + 1$
end while
$output \leftarrow leftshift(mantissa)$
```

4.3.2 Input Output Port Declaration

- **Clock:**
  Input Clock signal
- **1 bit input valid:**
  Input Check whether the input is valid or invalid. If valid module starts conversion.
- **8 bit fraction part :**
  Input Binary of integer value of the fractional part.
- **15 bit integer part:**
  Input binary of integer value
- **23 bit Mantissa:**
  Output Mantissa
- **1 bit output done flag:**
  This bit is set when calculation is completed and output is ready.

4.3.3 Code

Module Declaration

```
/*
    this module computes 23 bit mantissa from 15 bit binary of
    integer part and 8 bit binary of fractional part.
```

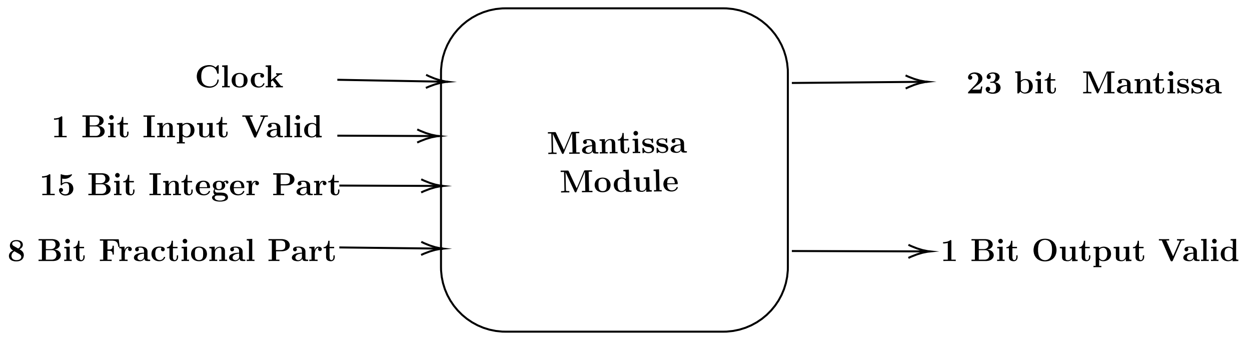


Figure 4.5: Mantissa module

```
*/

module mantissa
(
    input clk,
    input input_valid, // input taken flag
    input [14:0] intg, // integer part binary
    input [7:0] frac, // fractional part binary
    output [22:0] mantissa, // 23 bit mantissa
    output output_valid  // output done flag
 );
```

Data Initialization

```
reg [1:0] ns; // temporary holds next state value
reg [1:0] next_state;
reg [14:0] reg_intg; // integer part binary value
reg [7:0] reg_frac; // fractional part binary value
reg input_valid_reg; // input_valid flag


// reg for outputs
reg [22:0] reg_mantissa;
reg [22:0] out_mantissa;
reg output_valid_reg;

// for input valid we are initializing values
always @(posedge input_valid) begin
    input_valid_reg = 1'b1;
    reg_intg = intg;
    reg_frac = frac;
    ns = 2'b00; // next state
end
```

FSM

```
always @(ns) begin // state transition
    next_state = ns;
    // $display("%b———%b", ns,reg_mantissa);
end

// always @(mantissa) begin
//      $display("output: %b", mantissa); // displaying Output
// end

always @(posedge clk) begin
    if(input_valid_reg)begin
        case (next_state)
            2'b00:begin
                reg_mantissa = {reg_intg ,reg_frac};
                // $display("reg_mantissa: %b",reg_mantissa);
                ns = 2'b01; // next_state
            end
            2'b01:begin
                if (reg_mantissa[22] == 1)begin
                    ns = 2'b11; // next_state
                end
                else begin
                    reg_mantissa = {reg_mantissa << 1};
                    ns = 2'b10; // next_state
                end
            end
            2'b10:begin
                ns = 2'b01; // next_state
            end
            2'b11:begin
                // storing outputs
                out_mantissa = {reg_mantissa << 1};
                output_valid_reg = 1'b1;
            end
            2'bxx:begin
            // default case
            end
        endcase
    end
end
```

Output Assignment

```
// output assignment
assign mantissa = out_mantissa;
assign output_valid = output_valid_reg;
endmodule
```

4.4 Exponent Calculation

This module will calculate the exponent which is 8 bit of length from the input real number.

4.4.1 Pseudo-code

Here is the Pseudo-code for calculation of exponent for the module.

```
$count \leftarrow 0$
**if** $integer\_part \neq 0$ **then**
    **while** $integer\_part[count] \neq 1$ **do**
        $count \leftarrow count + 1$
    **end while**
    $exponent \leftarrow 127 + 14 - count$
**else**
    **while** $fractional\_part[count] \neq 1$ **do**
        $count \leftarrow count + 1$
    **end while**
    $exponent \leftarrow 127 - count - 1$
**end if**
```

4.4.2 Input Output Port Declaration

- **Clock:**
  Input Clock signal
- **1 bit input valid:**
  Input Check whether the input is valid or invalid. If valid module starts conversion.
- **8 bit fractional part :**
  Input Binary of integer value of the fractional part.
- **15 bit integer part:**
  Input binary of integer value
- **8 bit Exponent:**
  Output Exponent
- **1 bit output done flag:**
  This bit is set when calculation is completed and output is ready.

Figure 4.6 shows the input output declaration for the exponent module.

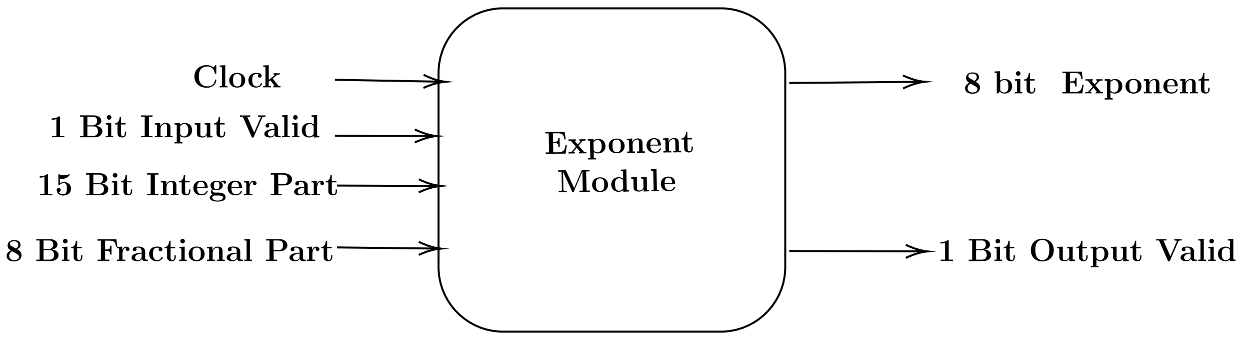


Figure 4.6: Exponent Module

4.4.3 Code

Module Declaration

```
/*
    this module computes 8 bit exponent from 15 bit binary
    of integer part and 8 bit binary of fractional part.
*/

module exponent
(
    input clk,
    input input_valid, // input taken flag
    input [14:0] intg, // integer part  binary
    input [7:0] frac, // fractional part binary
    output [7:0] exponent_out, // 8 bit exponent
    output output_valid  // output done flag
 );
```

Data Initialization

```
reg [1:0] ns; // temporary holds next state value
reg [1:0] next_state;
reg [14:0] reg_intg; // integer part binary value
reg [7:0] reg_frac; // fractional part binary value
reg input_valid_reg;


// registers for outputs
reg [7:0] reg_exponent;
reg output_valid_reg;
reg process;
reg [3:0] count;

// exponent_out = 0;
// for input valid we are initializing values
```

```
always @(posedge input_valid) begin
     output_valid_reg = 0;
     input_valid_reg = 1'b1;

     if(intg == 0)begin
        reg_frac = frac;
        process = 0;
     end

     else begin
         reg_intg = intg;
         process = 1;
     end

     ns = 2'b00; // next state
end
```

FSM

```
always @(ns) begin // state transition
     next_state = ns;
     // $display("%b———%b", ns,exponent);
end



// executes when integer part is greater than zero
always @(posedge clk) begin
     if(input_valid_reg && process)begin
          case (next_state)
               2'b00:begin
                    count = 0;
                    ns = 2'b01; // next_state
               end
               2'b01:begin
                    if (reg_intg[14-count] == 1)begin
                         ns = 2'b11; // next_state
                    end
                    else begin
                         ns = 2'b10; // next_state
                    end
               end
               2'b10:begin
                    count = count + 1;
                    ns = 2'b01; // next_state
               end
               2'b11:begin
                    reg_exponent = 127 + 14 - count;
```

```
                $display("%d, %b", count, reg_intg);
                input_valid_reg = 0;
                output_valid_reg = 1;
            end
            2'bxx:begin
            end
        endcase
    end
end


// executes when integer part is equal to zero
always @(posedge clk) begin
    if(input_valid_reg && (!process))begin
        case (next_state)
            2'b00:begin
                count = 0;
                ns = 2'b01; // next_state
            end
            2'b01:begin
                if (reg_frac[7-count] == 1)begin
                    ns = 2'b11; // next_state
                end
                else begin
                    ns = 2'b10; // next_state
                end
            end
            2'b10:begin
                count = count + 1;
                ns = 2'b01;  // next_state
            end
            2'b11:begin
                reg_exponent = 127 - count - 1;
                // $display("%d", count);
                input_valid_reg = 0;
                output_valid_reg = 1;
            end
            2'bxx:begin
            end
        endcase
    end
end
```

Output Assignment

```
// output assignment
assign exponent_out = reg_exponent;
assign output_valid = output_valid_reg;
```

endmodule

4.5 Test Bench and Result

For testing purpose an automated test bench was written to test out RTL code output and non RTL code output. Inputs were taken randomly using the verilog random function.

4.5.1 Code of the Test Bench

```
`include "sign_bit.v"
`include "bin2.v"
`include "mantissa.v"
`include "exponent.v"

module ieee_754_converter();
reg clock, test_Sign_bit;
reg [14:0] intg_reg;
reg [14:0] test_intg;
reg [22:0] mantissa_out_test;
reg [7:0] exponent_out_test;

reg [7:0] frac_inp;
reg [0:7] test_frac_out;
reg [7:0] test_frac;

wire sign_bit;
wire [14:0] intg_out;
wire [0:7] frac_out;
wire [7:0] exponent_out;
wire [22:0] mantissa_out;

reg input_valid_sign;
reg input_valid_frac_bin;
reg input_valid_mantissa;
reg input_valid_exponent;

wire output_valid_sign;
wire output_valid_frac_bin;
wire output_valid_mantissa;
wire output_valid_exponent;

integer j, count, i;

initial begin
  clock = 1;
```

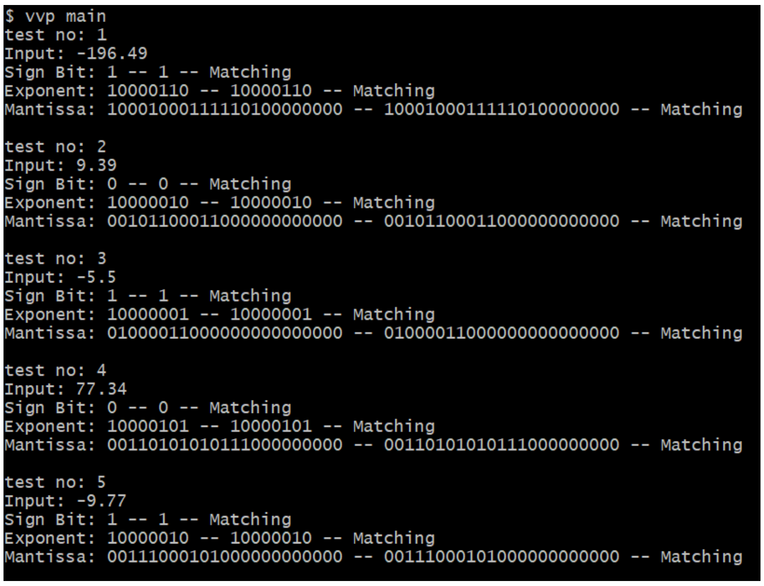
```
for (i=0; i<20; i=i+1)
begin
// $random(10000);
intg_reg = $urandom%200;
frac_inp = $urandom%100;

$display("test␣no:␣%d", i);
if(i % 2 == 0) intg_reg = intg_reg * -1;
$display("inputs:␣%d,␣%d", intg_reg, frac_inp);

test_frac = frac_inp;

if (intg_reg[14]==0) begin
  test_intg = intg_reg;
  test_Sign_bit = 0;
end
else begin
  test_Sign_bit = 1;
  test_intg = intg_reg * -1;
end


for(j = 0; j<8; j = j + 1)begin
  test_frac = test_frac * 2;
  test_frac_out[j] = test_frac / 100;
  test_frac = test_frac % 100;
end
$display("%b,%b,%b",test_Sign_bit, test_intg, test_frac_out);

// Mantissa
mantissa_out_test = {test_intg, test_frac_out};
while (mantissa_out_test[22]!= 1) begin
  mantissa_out_test = {mantissa_out_test<<1};
end
mantissa_out_test = {mantissa_out_test<<1};
// $display("%b", mantissa_out_test);

// Exponent Calculation
if(test_intg == 0)begin
  count = 0;
  while(test_frac_out[count]== 0)begin
    count = count + 1;
  end
  exponent_out_test = 127 - count - 1;
end

else begin
  count = 0;
```

```
      while(test_intg[14-count]== 0)begin
        count = count + 1;
      end
      exponent_out_test = 127 + 14 - count;
    end

    // $display("%d", exponent_out_test);

    input_valid_sign = 0;
    input_valid_frac_bin = 0;
    #1 input_valid_sign = 1;
    #1 input_valid_frac_bin = 1;
    #100
    intg_reg = intg_out;
    frac_inp = frac_out;
    // $display("%b,%b,%b", sign_bit, intg_reg, frac_inp);
    // $display("Sign Bit: %b", sign_bit);
    input_valid_exponent = 0;
    input_valid_mantissa = 0;
    #5
    input_valid_exponent = 1;
    input_valid_mantissa = 1;
    #750
    if(sign_bit==test_Sign_bit)
      $display("Sign␣Bit:␣%d␣--␣%d␣--␣Matching", sign_bit,
      test_Sign_bit);
    else
      $display("Sign␣Bit:␣%d␣--␣%d␣--␣Not␣Matching", sign_bit,
      test_Sign_bit);
    if(exponent_out==exponent_out_test)
      $display("Exponent:␣%d␣--␣%d␣--␣Matching", exponent_out,
      exponent_out_test);
    else
      $display("Exponent:␣%d␣--␣%d␣--␣Not␣Matching",
      exponent_out_test, exponent_out_test);
    if(mantissa_out==mantissa_out_test)
      $display("Mantissa:␣%b␣--␣%b␣--␣Matching",
      mantissa_out, mantissa_out_test);
    else
      $display("Mantissa:␣%b␣--␣%b␣--␣Not␣Matching",
      mantissa_out, mantissa_out_test);
    // #100 ;
    end
    #100
    $finish;
end
```

```
always
#2 clock = ~clock;




sign u_sign(
    clock,
    intg_reg,
    input_valid_sign,
    sign_bit,
    intg_out,
    output_valid_sign
);



frac_bin U_frac_bin(
  clock , // Clock
  input_valid_frac_bin , // reset
  frac_inp ,   // input value
  frac_out , // output reg
  output_valid_frac_bin
);


exponent U_exponent(
  clock , // Clock
  input_valid_exponent , // reset
  intg_reg ,   // input value
  frac_inp ,
  exponent_out ,
  output_valid
);


mantissa U_mantissa(
  clock , // Clock
  input_valid_mantissa , // reset
  intg_reg ,   // input value
  frac_inp ,
  mantissa_out ,
  output_valid_mantissa
);

endmodule //
```

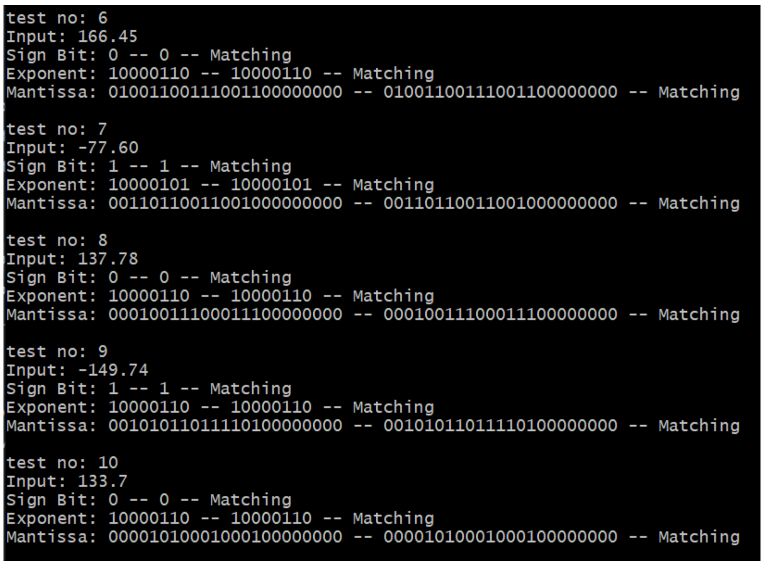
4.5.2 Result

For testing purpose multiple random numbers were generated and through the testbench the outputs were tested. It was seen that it was giving correct outputs for the inputs.The whole algorithm was written in behavioral code(non-rtl) in the test bench. Then outputs were collected and matched against the RTL outputs. Both of them were similar. Although RTL code follows a exclusive clock it took much greater time to complete compared to Non-RTL code.



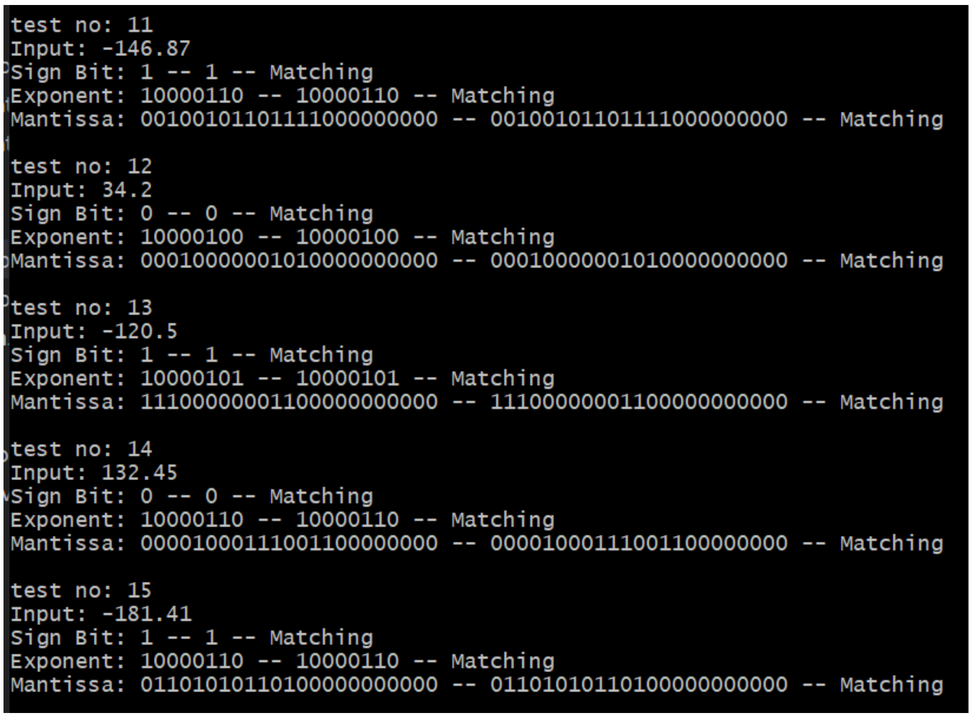
```
$ vvp main
test no: 1
Input: -196.49
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 10001000111110100000000 -- 10001000111110100000000 -- Matching

test no: 2
Input: 9.39
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000010 -- 10000010 -- Matching
Mantissa: 00101100011000000000000 -- 00101100011000000000000 -- Matching

test no: 3
Input: -5.5
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000001 -- 10000001 -- Matching
Mantissa: 01000011000000000000000 -- 01000011000000000000000 -- Matching

test no: 4
Input: 77.34
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000101 -- 10000101 -- Matching
Mantissa: 00110101010111000000000 -- 00110101010111000000000 -- Matching

test no: 5
Input: -9.77
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000010 -- 10000010 -- Matching
Mantissa: 00111000101000000000000 -- 00111000101000000000000 -- Matching
```

Figure 4.7: Test bench result(sample 1-5)



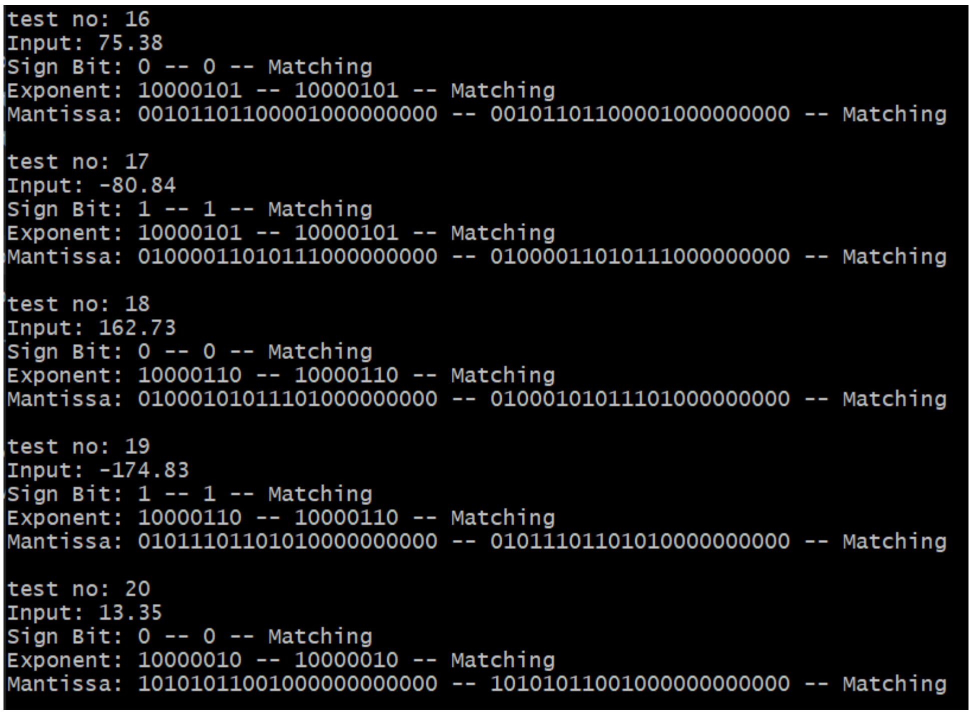
```
test no: 6
Input: 166.45
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 01001100111001100000000 -- 01001100111001100000000 -- Matching

test no: 7
Input: -77.60
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000101 -- 10000101 -- Matching
Mantissa: 00110110011001000000000 -- 00110110011001000000000 -- Matching

test no: 8
Input: 137.78
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 00010011100011100000000 -- 00010011100011100000000 -- Matching

test no: 9
Input: -149.74
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 00101011011110100000000 -- 00101011011110100000000 -- Matching

test no: 10
Input: 133.7
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 00001010001000100000000 -- 00001010001000100000000 -- Matching
```

Figure 4.8: Test bench result(sample 6-10)



```
test no: 11
Input: -146.87
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 00100101101111000000000 -- 00100101101111000000000 -- Matching

test no: 12
Input: 34.2
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000100 -- 10000100 -- Matching
Mantissa: 00010000001010000000000 -- 00010000001010000000000 -- Matching

test no: 13
Input: -120.5
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000101 -- 10000101 -- Matching
Mantissa: 11100000001100000000000 -- 11100000001100000000000 -- Matching

test no: 14
Input: 132.45
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 00001000111001100000000 -- 00001000111001100000000 -- Matching

test no: 15
Input: -181.41
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 01101010110100000000000 -- 01101010110100000000000 -- Matching
```

Figure 4.9: Test bench result(sample 11-15)



```
test no: 16
Input: 75.38
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000101 -- 10000101 -- Matching
Mantissa: 00101101100001000000000 -- 00101101100001000000000 -- Matching

test no: 17
Input: -80.84
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000101 -- 10000101 -- Matching
Mantissa: 01000011010111000000000 -- 01000011010111000000000 -- Matching

test no: 18
Input: 162.73
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 01000101011101000000000 -- 01000101011101000000000 -- Matching

test no: 19
Input: -174.83
Sign Bit: 1 -- 1 -- Matching
Exponent: 10000110 -- 10000110 -- Matching
Mantissa: 01011101101010000000000 -- 01011101101010000000000 -- Matching

test no: 20
Input: 13.35
Sign Bit: 0 -- 0 -- Matching
Exponent: 10000010 -- 10000010 -- Matching
Mantissa: 10101011001000000000000 -- 10101011001000000000000 -- Matching
```

Figure 4.10: Test bench result(sample 16-20)

Chapter 5

Conclusion

5.1 Summary of the Work

We are converting fractional decimal number to IEEE-754 single precision binary number.To implement this task we have broken down our task in five major portion. (1)Converting fractional part to Binary (2)Mantissa calculation (3) Exponent calculation (4) Sign bit calculation (5) Writing test benches and synthesizing.For each of these except last one we followed a common approach.At first Pseudo-code is written down after that input output ports for module are declared and lastly module is implemented using Verilog code.

In Pseudo-code what logic are working for a specific module are presented.The input output port refers to what are input-output ports ,their types and length of each port.In the coding section firstly the ports are declared ,secondly these ports are initialized and lastly module is implemented according to finite state machine (FSM).During the implementation RTL rules are followed.

Test benches create testing scenario through which module can be verified whether the module is working as desired or not.Behavioral code is used to write test benches.That means test bench codes are not bound by RTL rule.Some random inputs are fed and output are analyzed.

5.2 Proposed Future Work

There are multiple variants of IEEE-754 formats.The larger the precision the bigger and dynamic is the range of number it can represent.We have done this conversion task for single precision. In future we will extend our design for formats having larger precision.

Besides this to make our design more efficient we will try to reduce clock count .System will perform better with reducing clock count.

After improving floating point precision we will compare performance of our floating point converter with processors having different cores.This experiment will motivate and give direction to optimize our design.Lastly we will synthesize our design and perform fabrication task .

Chapter 6

Reference

1. https://en.wikipedia.org/wiki/IEEE_754#Formats
2.https://whatis.techtarget.com/definition/ASIC-application-specific-integrated-circu