RTL Design Document

Project Name: SAP – FPU

Project By: MERL - DSU

Contents

List of Figures

List of Tables

Abstract

In modern-day computations, especially where analog sensors are interfaced with the computation systems, FPU is an inevitable requirement. In earlier ages Integer Unit of the ALU was used to carry out floating-point arithmetic operations, however, this technique is gradually getting old-fashioned and out of practice with the advent of RISC-V ISA, which is the first ever open-source ISA. This was not only the problem of Intel, AMD, or other ASIC design giants but also a matter to look into for new ASIC designers as well. One such ASIC design research laboratory is Microelectronics Research Laboratory.

Microelectronics Research Laboratory (MERL), for whom this project is being designed, up till this date looks toward integrating open source available FPUs in the microprocessors they design however, this usage of open source FPU despite being free, has a cost. All the open source FPUs available are not always in line with the specific requirements of the processor to be designed. Some FPUs do not have multi-standard compatibility, and some are not even equipped with modern-day Floating-Point representation standards such as Bfloat-16, also updating the open source FPU in accordance with the specific design requirements is a tedious task to achieve.

This project aims to develop an FPU with multi-standard compatibility, having all the IEEE-754 (1985) standards and Bfloat-16 standard as well. The implemented design is an execution unit of FPU which is able to perform 22 RISC-V floating-point instructions.

1 Introduction

A floating-point unit (FPU) is a subsection of the Arithmetic Logic Unit inside a processor. This unit is responsible for carrying out the floating-point arithmetic which is instructed to the processor. RISC-V ISA contains extensive support for floating-point instruction for different standards. Figure 1.1 shows the general architecture of a RISC-V processor with the position of the FPU highlighted.

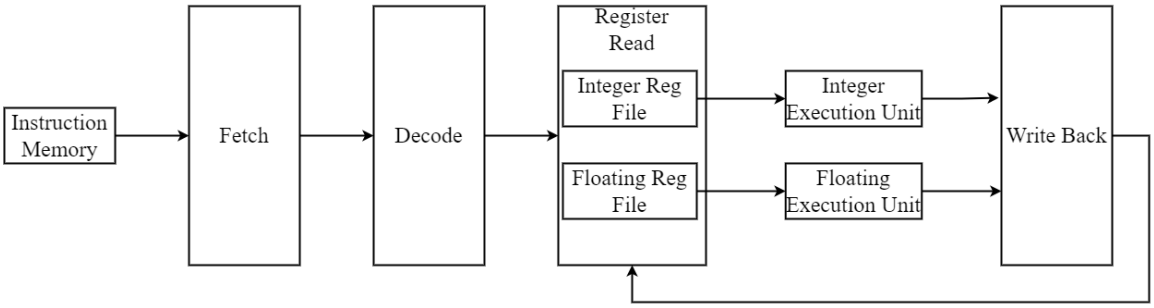

Figure 1.1: Top Level Architecture Of RISC-V Processor

As shown in Figure 1.1, FPU is a part of the execution stage of the processor (ALU). ALU with the addition of FPU has two units, the Integer Execute Unit, responsible for integer arithmetic, and the floating-point unit, responsible for the implementation of floating-point arithmetic. The addition of FPU in the processor also adds a floating-point register file to the processor, which is used to hold the source and destination registers of the floating-point instructions. Traditionally integer unit was the only unit of ALU inside the processors and it was used to implement floating-point instruction as well, however, with increasing needs for analog data computations (such as that in sensor-based automation systems and machine-learned systems), the requirement of a separate unit for floating-point arithmetic has increased because the integer implements floating-point instruction with a major cost of precision loss.

As it is evident from Figure 1.1 that the processor's top module has a few distinct stages such as Instruction memory, Write Back, etc. Therefore, in order to achieve higher clock speeds and at the same time higher throughput of data, the processor is pipelined which briefly means that if an instruction "A" is in the processor; before it completes execution, instruction "B" can also be given. This is done in a way that the data path is broken up into multiple different stages. On every cycle, the data jumps from one stage to the other so if the data of instruction "A" is in the third stage of the processor the data of instruction "B" would be in the second stage of the processor, and flow would continue in this fashion. Since this technique ensures higher throughput of data and ultimately higher clock speed so this technique is replicated inside the FPU as well.

In order to carry out the floating-point arithmetic, floating-point numbers are to be represented in a specific standard. Generally, all floating-point standards have three different fields shown in Table 1.1.

	Number of Bits	**Meaning**
Sign	1 Bit	Sign = 1 represents Negative Numbers Sign = 0 represents Positive Numbers
Exponent	Depends upon the standard used	Represents the Binary equivalent of the Number's exponent
Mantissa	Depends upon the standard used	Represents the Binary equivalent of the Number's mantissa

Table 1.1: General Floating-Point Representation

The most used standard of floating-point representation in ASIC designs is the IEEE-754 stan-

dard. IEEE-754 standard contains different sizes of representation standards that are, Single Precision, Double Precision, Quad, and Half Precision. All these standards are different from each other. In terms of the number of bits, they allocate in different fields of the standard.
One of the emerging floating-point standard which is aimed at carrying out low-precision arithmetic is called the Bfloat-16 standard. This standard was created by GOOGLE with an aim to perform arithmetic for machine learning algorithms as they do not require very high precision arithmetic. This standard is very similar to IEEE single precision standard however, it differs in the mantissa field size. Due to this feature Bfloat-16 standard caters to the same range of floating-point numbers as single precision does but with lesser precision.

The designed FPU is now being implemented in a way that it would be able to cater all the aforementioned floating-point standards. This is achieved by the technique of parameterization.

2 Literature Review

2.1 IEEE-754 Standard

IEEE Standard for Floating-Point Arithmetic (IEEE-754) was developed by The Institute of Electrical Engineering. IEEE-754 Floating-Point Numbers define how to represent a binary number in Floating-Point format. IEEE-754 Floating-Point Numbers have 3 basic components:

Sign It is a single bit that is used to indicate the sign of the number, 1 is used to represent a negative sign while 0 is used to represent a positive sign.

Exponent The size of the exponent in IEEE-754 Floating-Point Numbers depends on the precision that is in use, either half-precision, single-precision, or double-precision [1]. The exponent in all of the precisions mentioned above is biased so that no negative values occur for the exponent.

Mantissa The mantissa is the significand of a number which is in scientific notation. Similar to the exponent's size, the mantissa's size is also different for each precision.
Table 2.1 shows the bit width of mantissa and exponent for different precisions of the IEEE-754 standard and the bias value for each of the precision.

Precision	**Sign Bits Width**	**Exponent Bits Width**	**Mantissa Bits Width**	**Bias**
Half Precision	1	5	10	15
Single Precision	1	8	23	127
Double Precision	1	11	52	1024

Table 2.1: IEEE-754 Standards Representation

IEEE-754 standard also defines how to represent special values such as Infinity, SNaN, QNaN, and Zero.

Infinity In IEEE-754 standard there are two types of infinities, positive infinity and negative infinity. For each of the IEEE-754 precision, infinities are represented by setting the exponent field of the number to all 1 and setting the mantissa field of the number to all zero. Positive and negative infinity differ from each other based on the sign.

SNaN In IEEE-754 standard, for all the precisions SNaN is represented by setting the exponent field of the number to all 1s and placing at least a single 1 in the mantissa at any position other than MSB. The sign bit can be set to 1 or 0.

QNaN In IEEE-754 standard, for all the precisions QNaN is represented by setting the exponent field of the number to all 1s and placing a 1 at the MSB of mantissa. The sign can be set to 1 or 0.

Zero In IEEE-754 standard, for all the precisions positive zero is represented by setting all of the exponent bits to zero, all the mantissa bits to zero, and also the sign bit to 0, while negative zero is represented in the same manner as positive zero, however, the sign is 1 instead of zero.

2.2 Normalization

In floating-point arithmetic operations, one of the most important operations is Normalization. Normalization of operands can be required to be done at the start of the operation, or the end of the operation. If normalization is implemented at the start it is called pre-normalization and if it occurs at the end it is called post-normalization. In normalization, left shifts are made in the mantissa of the operand, until the MSB of the operand becomes non-zero. Based on the number of shifts made

in the mantissa, the exponent is also adjusted. To calculate the number of left shifts to be made in the mantissa different techniques are used. Two of which are

- Leading Zero Detector
- Priority Encoder

2.2.1 Leading Zero Detector

There are several approaches to designing an LZD, one technique is presented in the paper [2]. In the above-mentioned paper, LZD is developed using an algorithmic approach, resulting in a modular design. The algorithm used in the paper is:

1. Form the pair of bits Bi, Bi+1, for i = 0 to N-2
2. Determine P and V bits for each pair, using Table 2.2.

Pattern	Position	Valid
1X	0	yes
01	1	yes
00	X	no

Table 2.2: Two Bit Truth Table For LZD

3. For the next level determine the P and V, using the pair of P (P_{right} and P_{left}) and V (V_{right} and V_{left}) determined in the previous step as:
   $V_g = V_l + V_r$
   if: $V_l = 1$ then $P_g = 0$, P_r where "," designates concatenation.
   else if: $V_r = 1$ then $P_g = 1$, P_r

Repeat step (3) log (N) - 2 times. In SAP-FPU, LZD mentioned above is used.

2.2.2 Advanced Priority Bit Generator

A priority bit generator is a Boolean circuit that is used to indicate the position of the leading one in a data stream of a particular size. The position is indicated by generating a hot-encoded vector for a data stream. In a hot-encoded vector, only the bit corresponding to the location at which MSB is present is set high. The hot-encoded vector generated from the priority bit generator can be further processed using an advanced priority encoder which outputs the location of 1 in the hot-encoded vector. Using the location generated from the priority bit encoder, shifting is done in the mantissa.

2.3 Flags

When it comes to making the FPU compliant with the IEEE standard many aspects have to be taken into note; exceptions and flags in such cases hold great value. Five types of exceptions namely, invalid operation exception, divideByZero exception,overflow exception, underflow exception, and inexact exception are signaled when they occur and along with them comes a corresponding status. [3]

2.3.1 Invalid Condition

The invalid operation exception is signaled if and only if there is no usefully definable result. In these cases the operands are invalid for the operation to be performed.

1. Any general-computational or signaling-computational operation on a signaling NaN, except for some conversions.
2. Multiplication: multiplication (0, ∞) or multiplication (∞, 0).

3. fusedMultiplyAdd: fusedMultiplyAdd(0, ∞, c) or fusedMultiplyAdd(∞, 0, c) unless c is a quiet NaN; if c is a quiet NaN then it is implementation defined whether the invalid operation exception is signaled.

4. Addition or subtraction or fusedMultiplyAdd: magnitude subtraction of infinities, such as: addition ($+\infty$, $-\infty$).

5. Division: division (0, 0) or division (∞, ∞).

6. Square Root if the operand is less than zero.

7. Conversion of a floating-point number to an integer format, when the source is NaN, infinity, or a value that would convert to an integer outside the range of the result format under the applicable rounding attribute.

8. Comparison by way of unordered-signaling predicates listed in, when the operands are unordered.

2.3.2 Division By Zero

The divideByZero exception shall be signaled if and only if an exact infinite result is defined for an operation on finite operands. The default result of divideByZero shall be an Package inputenc Error: Unicode character ∞ correctly signed according to the operation.

- For division, when the divisor is zero and the dividend is a finite non-zero number, the sign of the infinity is the exclusive OR of the operands.

2.3.3 Overflow

The overflow exception shall be signaled if and only if the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. The default result shall be determined by the rounding-direction attribute and the sign of the intermediate result as follows:

1. roundTiesToEven and roundTiesToAway carry all overflows to ∞ with the sign of the intermediate result.

2. roundTowardZero carries all overflows to the format's largest finite number with the sign of the intermediate result.

3. roundTowardNegative carries positive overflows to the format's largest finite number, and carries negative overflows to $-\infty$.

4. roundTowardPositive carries negative overflows to the format's most negative finite number, and carries positive overflows to $+\infty$.

In addition, under default exception handling for overflow, the overflow flag shall be raised and the inexact exception shall be signaled.

2.3.4 Underflow

The underflow exception shall be signaled when a tiny non-zero result is detected.
The implementer shall choose how tininess is detected.

2.3.5 Inexact

Unless stated otherwise, if the rounded result of an operation is inexact—that is, it differs from what would have been computed were both exponent range and precision unbounded—then the inexact exception shall be signaled. The rounded or overflowed result shall be delivered to the destination.

2.4 Rounding

Rounding is one of the important operations in floating-point arithmetic. RISC-V specifications have described 5 rounding modes

1. Round to Nearest tie to even (RNE)
2. Round to Nearest tie to Maximum Magnitude (RMM)
3. Round to positive infinity (RUP)
4. Round to negative infinity (RDN)
5. Round to zero (RTZ)

Round to Nearest tie to even (RNE) In Round to Nearest tie to even (RNE) rounding mode, the number is rounded to the nearest possible number and if it is a tie (similar distance from the two nearest numbers) then the number is rounded to the even number.

Round to Nearest tie to Maximum Magnitude (RMM) In Round to Nearest tie to Maximum Magnitude (RMM) rounding mode, the number is rounded to the nearest possible number and if it is a tie (similar distance from the two nearest numbers) then the number is rounded to the maximum magnitude number.

Round to positive infinity (RUP) In Round to positive infinity (RUP) rounding mode, the number is rounded always towards the positive infinity. This means a positive number is always incremented and a negative number is always truncated.

Round to negative infinity (RDN) In Round to negative infinity (RDN) rounding mode, the number is rounded always towards the negative infinity. This means a negative number is always incremented and a positive number is always truncated.

Round to zero (RTZ) In Round to zero (RTZ) rounding mode, the number is rounded always towards zero. This means both, negative and positive numbers are always truncated.

The decision of whether rounding is to be done is taken on the basis of Guard, Round, and Sticky Bits. The guard bit is the first bit after the useful mantissa bits, the round bit is the first bit after the guard bit, and the sticky bit is the bit-wise OR of all the remaining bits. For all conversion instructions and computational instructions, GRS bits are obtained during the operations and are then used to round the result.

2.5 Bfloat-16

In terms of computation in Machine Learning, 32-bit Floating-point representation was not much considered as it had a few drawbacks which were that the memory would be occupied by 32-bits for each number used and that the hardware had to be specifically made to support 32-bit floating-point computation as it was required in many cases. After consideration the solution for this was to reduce the number of bits therefore, 16-bit representation was further considered but it also had drawbacks such as a limited range of representation of numbers, support from hardware, and quantization error. The issue of the limited range was addressed by increasing the number of exponents and for this, a whole different representation was used called Bfloat-16. In terms of similarity Bfloat-16 and IEEE 16-bit floating-point representation had the same number of bits but differed in the case of exponents.[4] The representation of the standard Bfloat-16 can be seen in Table 2.3.

	Sign	**Exponent**	**Mantissa**
Bfloat-16	1	8	6

Table 2.3: Bfloat-16 Standard Representation

3 Floating-Point Instructions

In SAP-FPU there are 22 types of instructions, all of these instructions are executed in a Single Clock Cycle.

3.1 Single Cycle Instructions

Single cycle instructions are those instructions that take one clock cycle to complete their operation. In SAP-FPU Single cycle instruction can be further divided into 4 sub-categories based on their functionality. The 4 sub-categories are:

1. Conversion Instructions
2. Computational Instructions
3. Non-Computational Instructions
4. Conditional Instructions

3.1.1 Conversion Instructions

One of the types of floating-point instructions are conversion instructions. Conversion instructions are used to convert the data from one format to another. There are two types of conversion instructions in RISC-V ISA which are as follows:

- Float to Integer
- Integer to Float

3.1.1.1 Float to Integer

Instruction Overview Float to Integer instruction is used to convert the data from the floating-point format to the integer format. According to RISC-V specifications, there are two variants of Float to Integer instruction which are as follows:

1. Signed-Float to Integer
2. Unsigned-Float to Integer

Unsigned-Float to Integer variant converts the floating-point number to Unsigned Integer and Signed-Float to Integer variant converts the floating-point number to Signed Integer. As the complete range of all the standards except IEEE-16 standards cannot be converted to an integer. Hence for different types of out-of-range numbers, RISC-V specifications have provided outputs that are produced as per the occurring case, Table 3.1 shows the valid range for input and output that must be produced in case of both the invalid inputs and out-of-range inputs. Out-of-Range numbers are handled by the Exceptional check block, which runs in parallel with the main lane. In the main lane, initially, the input floating-point number is converted to Double Precision format for both variants. From the converted floating-point number integer is extracted. After the extraction of the integer, rounding is implemented as per the provided rounding mode. If the selected variant is Signed-Float to Integer, the Rounded result is converted to signed format depending on the sign of the input. While for Unsigned-Float to Integer, Rounded result is not converted. In case no Exception is detected then the output from the main block is presented at the output port. However, if an Exception is detected then the result coming from the Exception check block is presented at the output port.

	Float to Signed Integer	**Float to Unsigned Integer**
Minimum valid input (after rounding)	2^{31}	0
Maximum valid input (after rounding)	$2^{31} - 1$	$2^{32} - 1$
Output for out-of-range negative input	-2^{31}	0
Output for -ve Infinity	-2^{31}	0
Output for out-of-range positive input	$2^{31} - 1$	$2^{32} - 1$
Output for +ve Infinity or NaN	$2^{31} - 1$	$2^{32} - 1$

Table 3.1: Domains Of Float-To-Integer Conversions And Behaviour For Invalid Inputs.

Flow Chart The complete explanation of the flow of the whole instruction was discussed in the Instruction Overview. A pictorial representation of the flow can be observed in Figure 3.1.

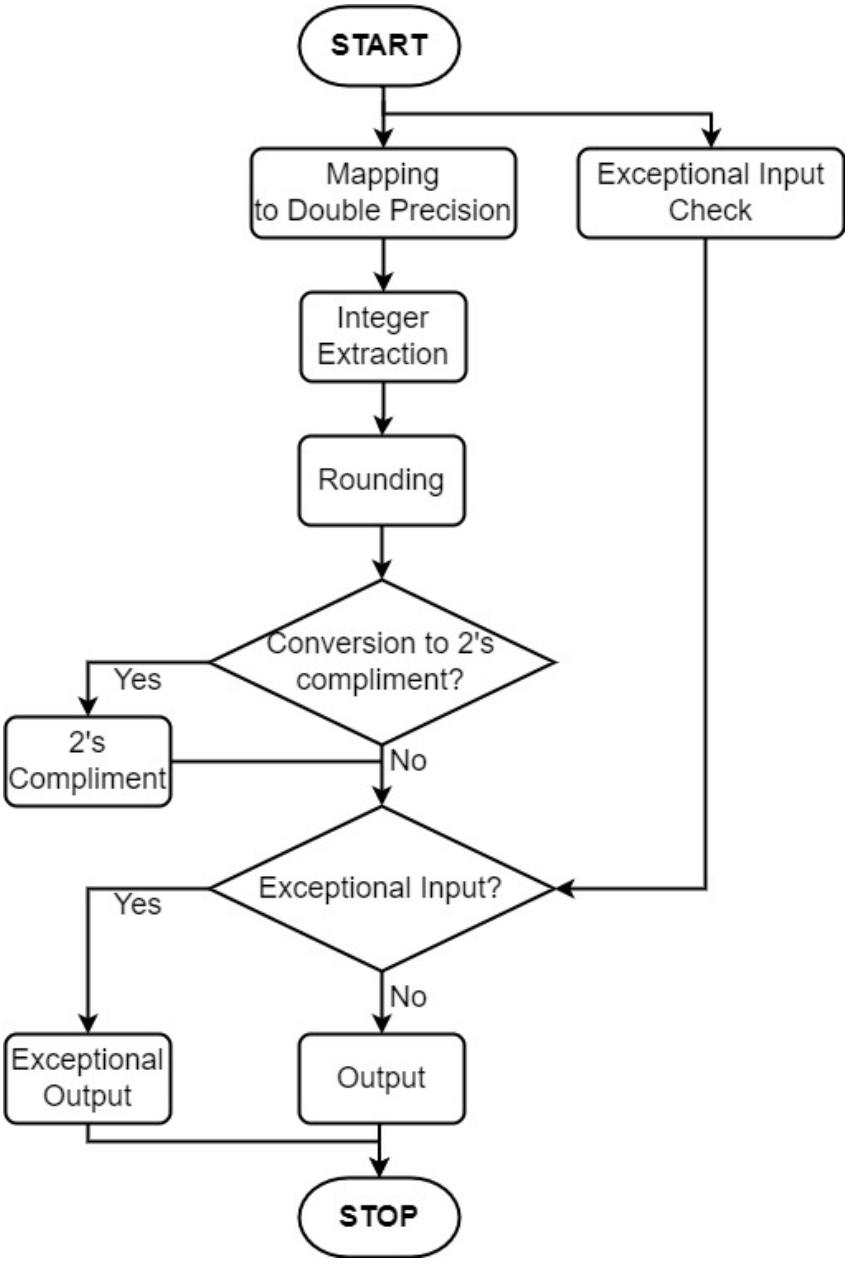

Figure 3.1: Flow Chart Of Float To Integer

Functional Block Diagram The functional block diagram of Float to Integer can be observed in Figure 3.2.

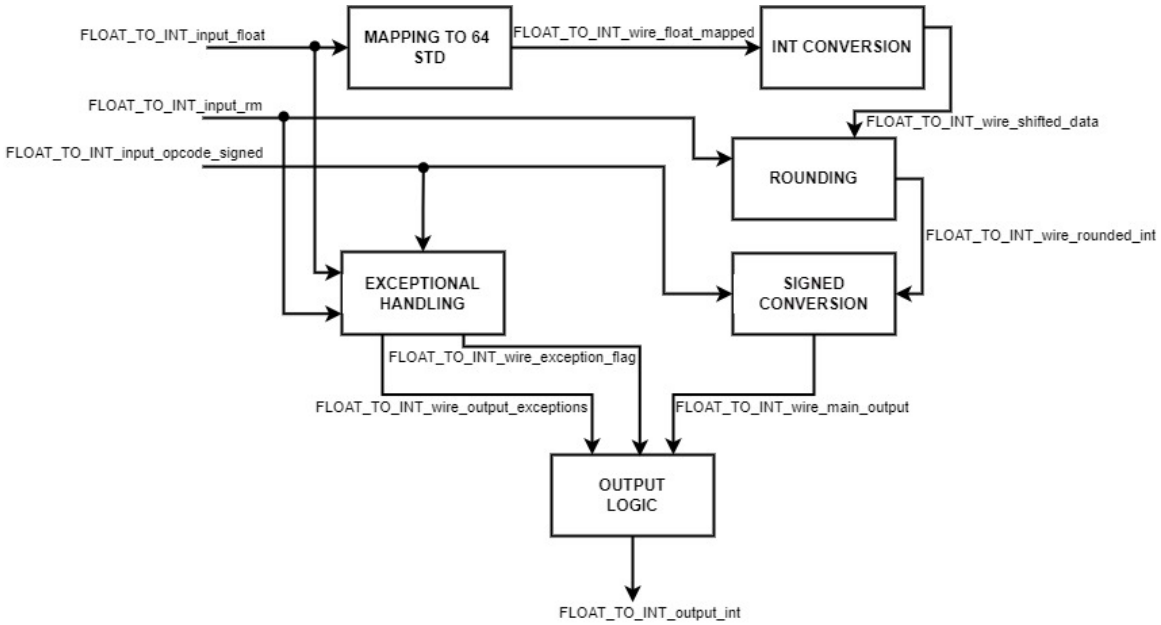

Figure 3.2: Functional Block Diagram Of Float To Integer

I/O List The I/O List of the Float to Integer Module can be seen in Table 3.2.

Port Name	Direction	Size	Description
rst_l	input	1	This port is used for reset
FLOAT_TO_INT_input_opcode_FI	input	1	This port is used for float to integer opcode
FLOAT_TO_INT_input_opcode_signed	input	1	This port is used for signed opcode
FLOAT_TO_INT_input_opcode_unsigned	input	1	This port is used for unsigned opcode
FLOAT_TO_INT_input_rm	input	3	This port is used for input rounding mode
FLOAT_TO_INT_input_float	input	STD	This port is used for input Floating-Point number
FLOAT_TO_INT_output_inexact_flag	output	1	This port is used to output inexact flag
FLOAT_TO_INT_output_invalid_flag	output	1	This port is used to output invalid flag
FLOAT_TO_INT_output_int	output	32	This port is used to output integer number

Table 3.2: I/O List Of Float To Integer Module

3.1.1.2 Integer to Float

Instruction Overview Integer to Float is used to convert the data from the Integer format to floating-point format. According to RISC-V specification, there are two variants of Integer to floating-point instruction which are as follows

1. Signed-Integer to Float
2. Unsigned-Integer to Float

In both variants initially, the modulus of the input integer is calculated which is then passed into LZD [2] circuitry that finds out the location of the leading one in the modulus. Using the location of the leading one, shifts are performed in the modulus to obtain the mantissa for the floating-point number. The exponent is also calculated using the location of the leading one. Mantissa obtained through shifting is then rounded as per the input rounding mode. In the case of Signed-Integer to Float, the sign of the result is determined by whether the input was in 2's complement form or not. While in the case of Unsigned-Integer to Float sign is always positive. Finally, sign, exponent, and rounded mantissa are concatenated and presented at the output.

Flow Chart The complete explanation of the flow of the whole instruction was discussed in the Instruction Overview. A pictorial representation of the flow can be observed in Figure 3.3.

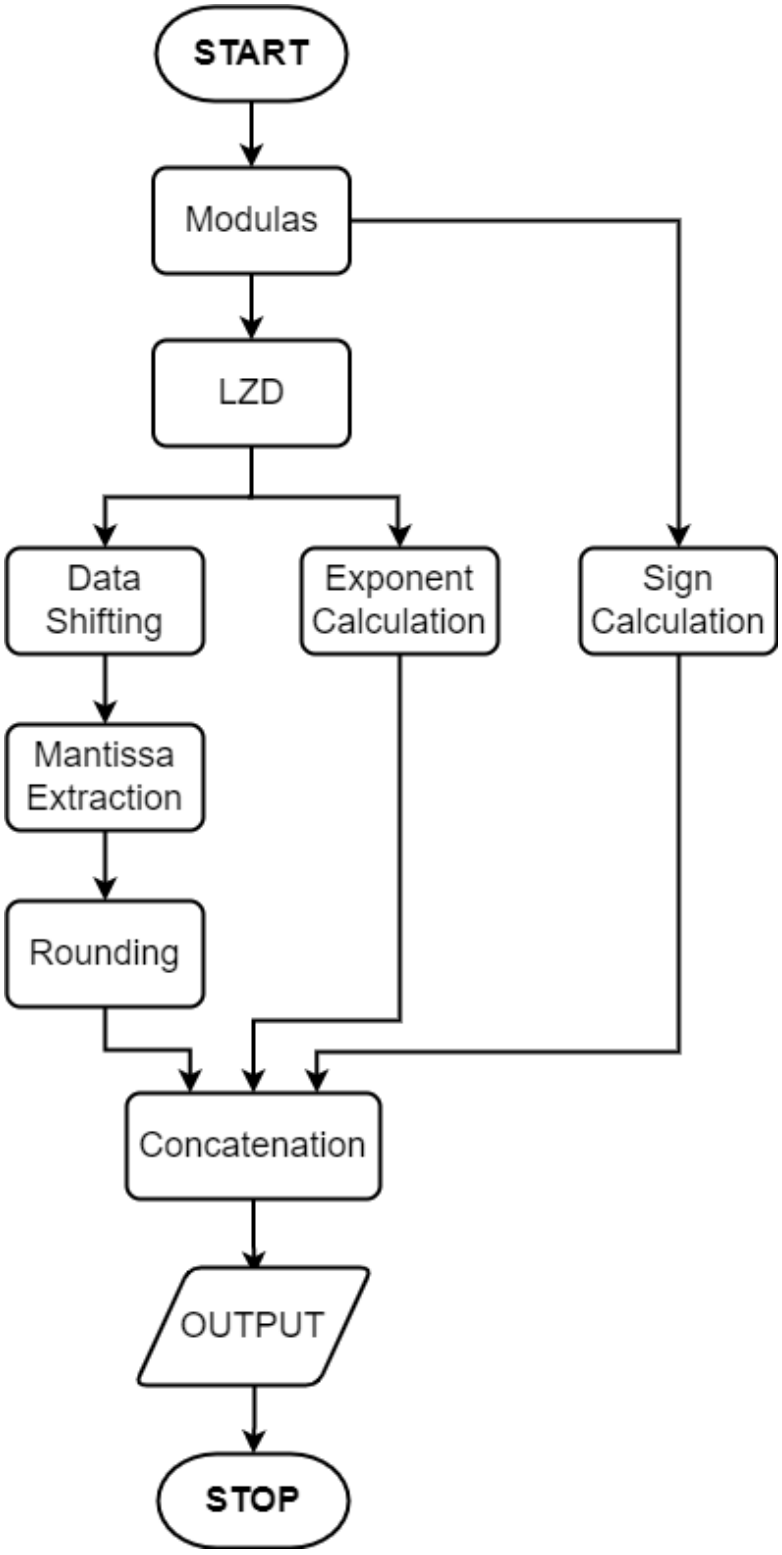

Figure 3.3: Flow Chart Of Integer To Float

Functional Block Diagram The functional block diagram of Integer to Float can be observed in Figure 3.4.

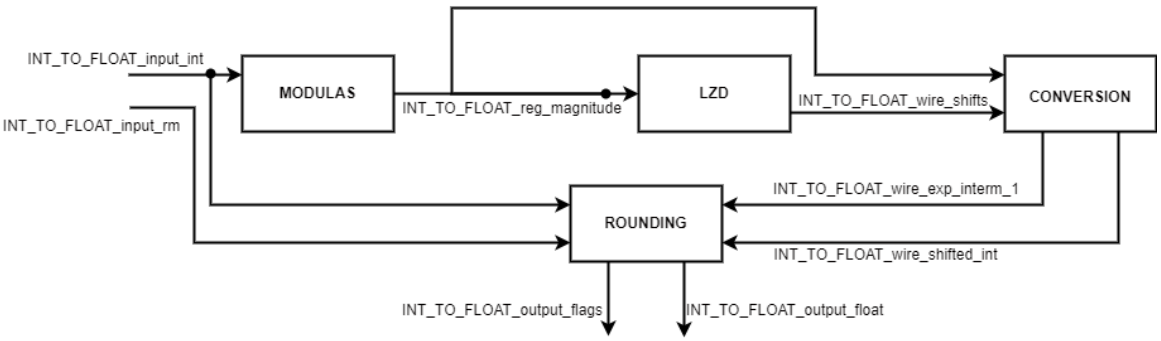

Figure 3.4: Functional Block Diagram Of Integer To Float

I/O List The I/O List of the Integer to Float Module can be seen in Table 3.3.

Port Name	Direction	Bit Width	Description
rst_l	input	1	This port is used for reset
INT_TO_FLOAT_input_opcode_IF	input	1	This port is used for integer to float opcode
INT_TO_FLOAT_input_opcode_signed	input	1	This port is used for signed opcode
INT_TO_FLOAT_input_opcode_unsigned	input	1	This port is used for unsigned opcode
INT_TO_FLOAT_input_rm	input	3	This port is used for rounding mode
INT_TO_FLOAT_input_int	input	32	This port is used for input integer
INT_TO_FLOAT_output_inexact_flag	output	1	This port is used to output inexact flag
INT_TO_FLOAT_output_invalid_flag	output	1	This port is used to output invalid flag
INT_TO_FLOAT_output_float	output	STD	This port is used to output Floating-Point number

Table 3.3: I/O List Of Integer To Float Module

RTL Hierarchy The hierarchy of the modules in the RTL can be seen in Figure 3.5

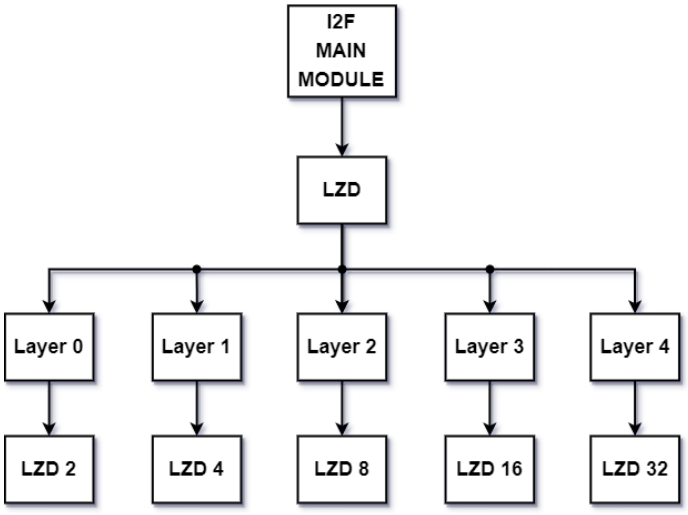

Figure 3.5: RTL Hierarchy Of Integer To Float

3.1.2 Computational Instructions

One of the 4 types of floating-point instruction is Computational instruction. Computational Instructions are used to perform arithmetic instructions. In SAP-FPU there are 7 types of single cycle computational instructions.

1. Fused Multiply Addition
2. Fused Multiply Subtraction
3. Fused Negate Multiply Addition
4. Fused Negate Multiply Subtraction
5. Floating-Point Addition
6. Floating-Point Subtraction
7. Floating-Point Multiplication

3.1.2.1 Floating-Point Multiplication Instruction

Instruction Overview Floating-Point Multiplication instruction performs the arithmetic operation of multiplication on two floating-point numbers. To perform the multiplication of 2 floating-point numbers, the exponent of the numbers are added together in parallel while the mantissa of both the numbers are multiplied together. Added exponent and multiplied mantissa are then passed onto the Post-Normalization block. In the Post-Normalization block, bias is removed from the added exponent (since the exponent in floating-point numbers are biased therefore, exponent obtained from the addition of both the exponent have a double bias in it). While mantissa obtained from the multiplication of both the mantissa is normalized and then rounded as per the desired rounding mode. After rounding if the number is not representable in the selected standard of a floating-point number, the exceptional output is presented at the output port, else the normal result is presented at the output port.

Flow Chart The complete explanation of the flow of the whole instruction was discussed in the Instruction Overview. A pictorial representation of the flow can be observed in Figure 3.6.

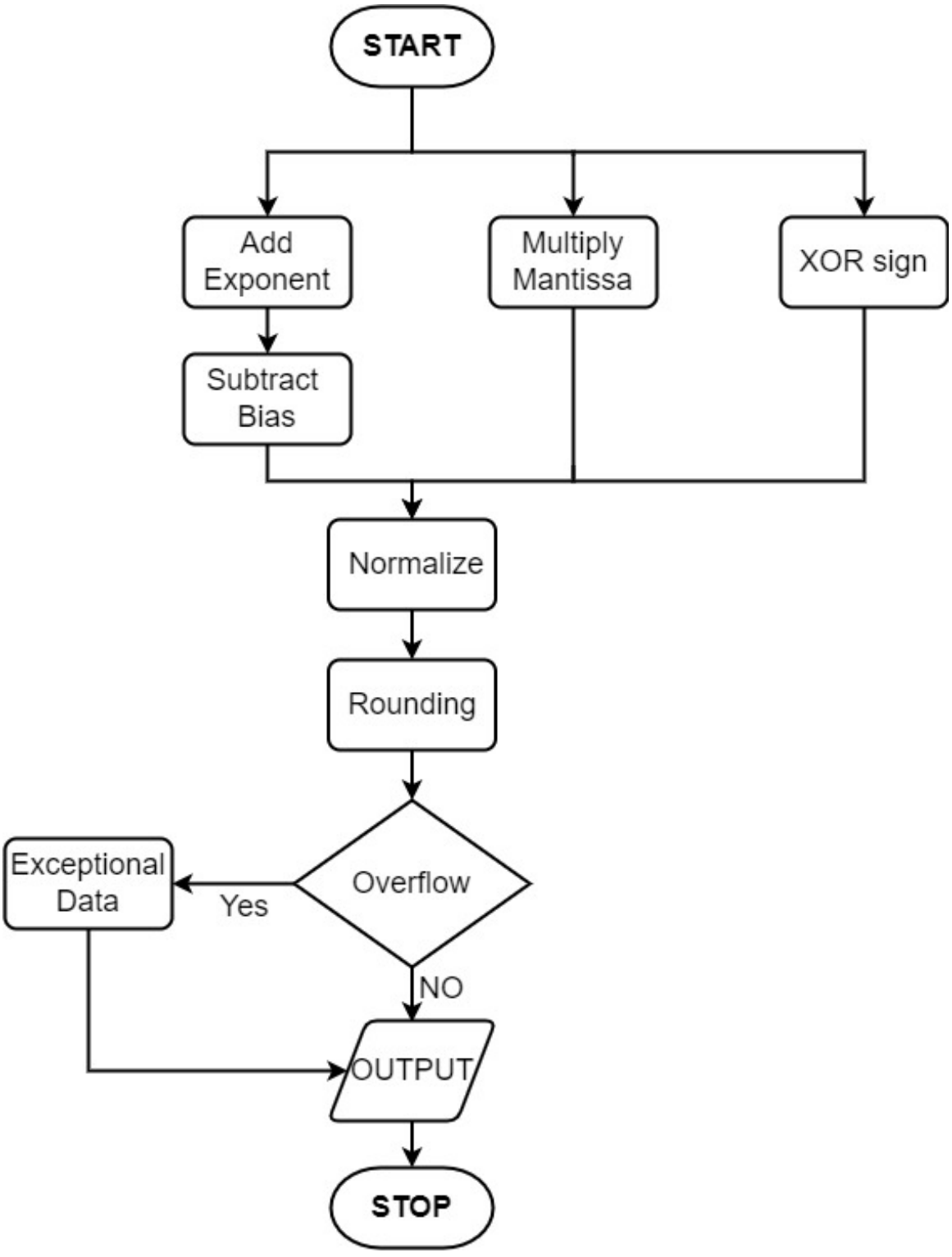

Figure 3.6: Flow Chart Of Floating-Point Multiplication

Functional Block Diagram The functional block diagram of Floating-Point Multiplication can be observed in Figure 3.7.

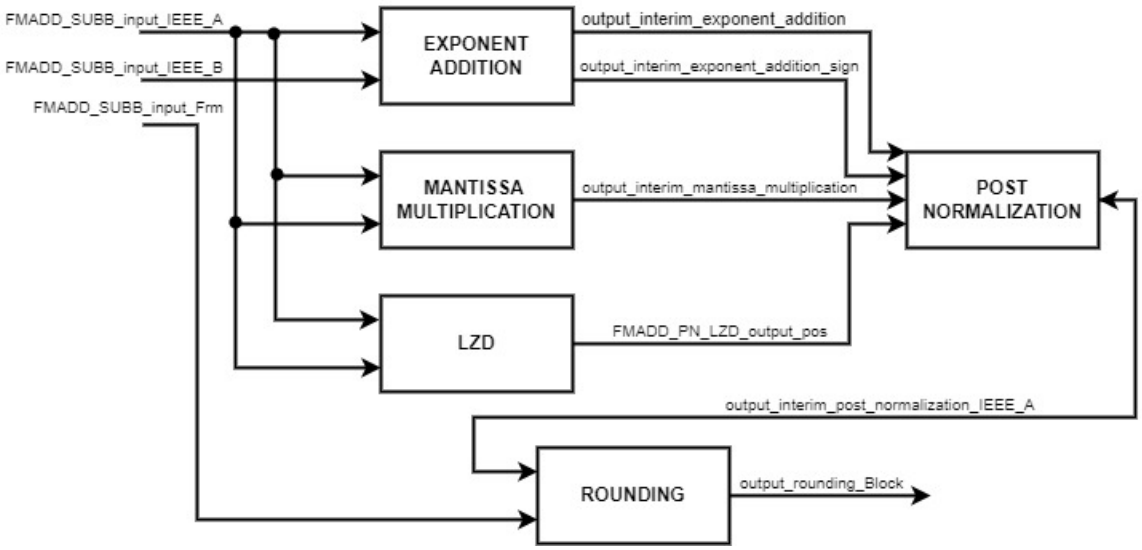

Figure 3.7: Functional Block Diagram Of Floating-Point Multiplication

3.1.2.2 Floating-Point Addition And Subtraction Instruction

Instruction Overview Floating-Point Addition/Floating-Point Subtraction are two different instructions in RISC-V. As the name suggests Floating-Point Addition instruction implements the following mathematical equation,

$$O = A + B \tag{3.1}$$

The Floating-Point Subtraction instruction implements the following mathematical equation,

$$O = A - B \tag{3.2}$$

Both of these instructions are performed in SAP-FPU using the same Lane (Hardware), this is mainly because the steps required for the implementation of these instructions are the same, however, some differences are there but those differences are handled inside the same lane. Another reason for implementing these instructions using the same lane is the fact that there might be a case where the input instruction to FPU is Floating-Point Addition but because of different signs of operands the actual operation required is subtraction and vice versa for Floating-Point Subtraction. In the Floating-Point Addition/Floating-Point Subtraction Lane, the first operation to be carried out is Exponent Matching where exponents are matched and relevant shifts to mantissa are provided, the next step is Mantissa Addition/Subtraction where actual addition or subtraction of mantissa takes place, the results from this step is normalized in Post-Normalization and finally, the results are rounded as per the intended rounding mode in Rounding Block.

Flow Chart The complete explanation of the flow of the whole instruction was discussed in the Instruction Overview. A pictorial representation of the flow can be observed in Figure 3.8.

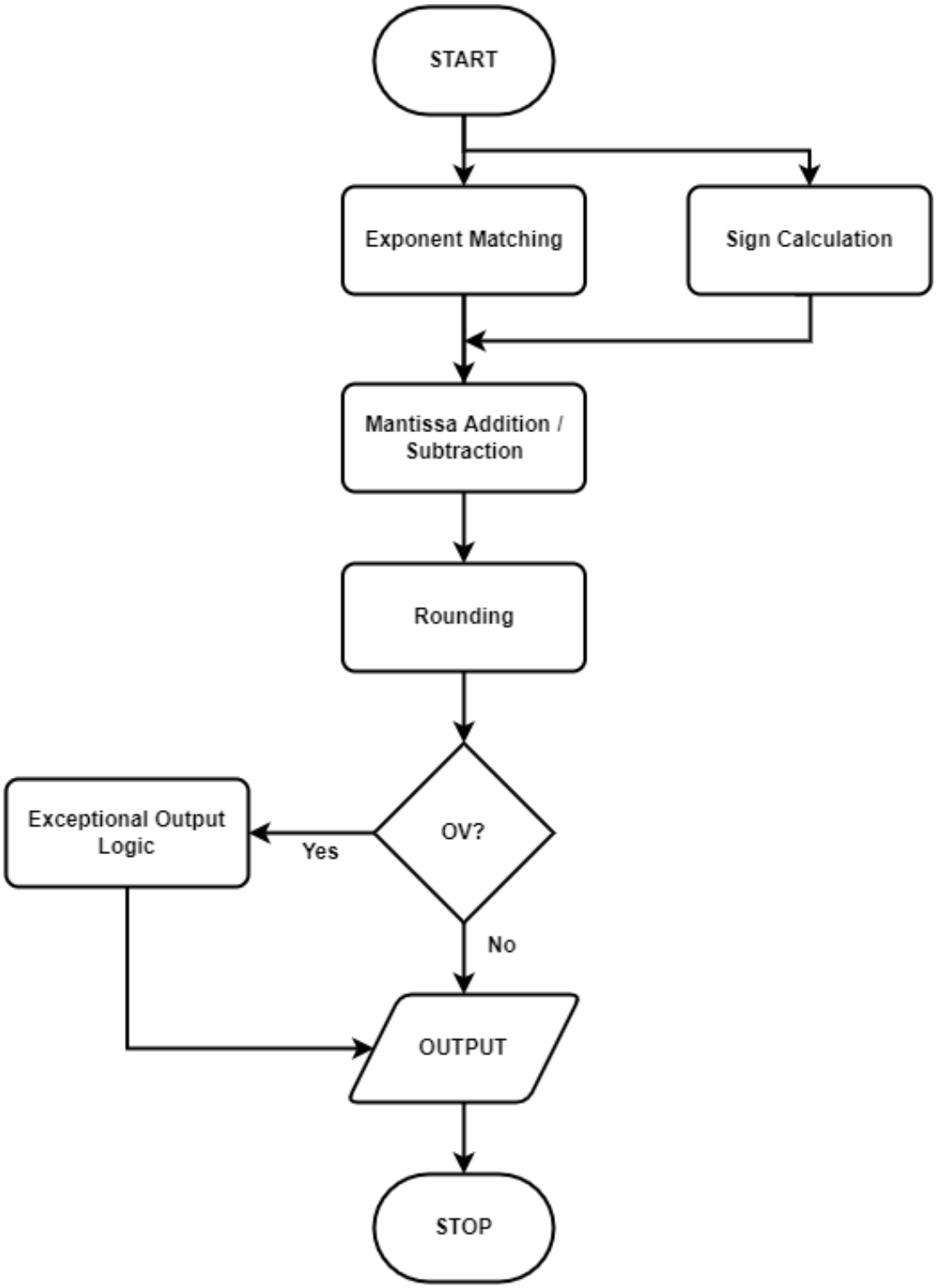

Figure 3.8: Flow Chart Of Floating-Point Addition And Subtraction

Functional Block Diagram The functional block diagram of Floating-Point Addition and Subtraction can be observed in Figure 3.9.

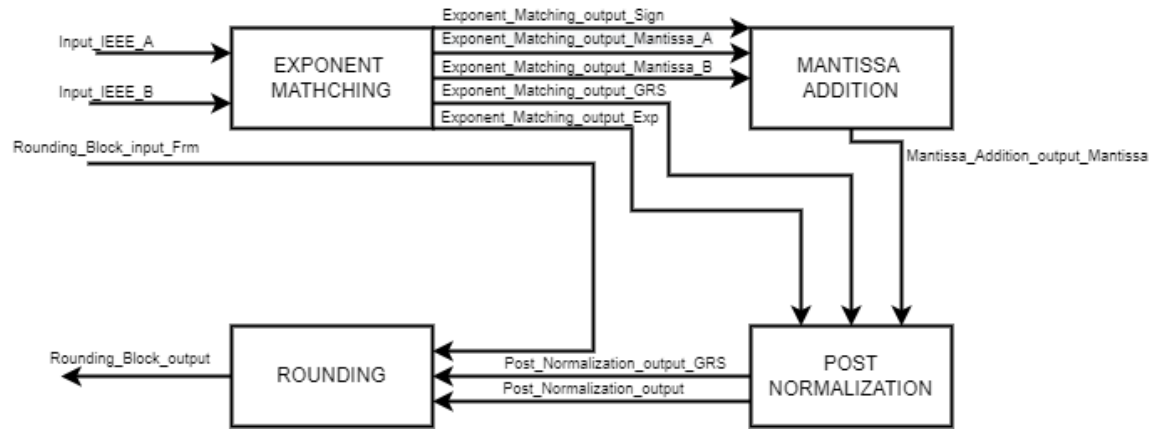

Figure 3.9: Functional Block Diagram Of Floating-Point Addition And Subtraction

I/O List The I/O List of Floating-Point Addition and Subtraction Module can be seen in Table 3.4.

Port Name	**Direction**	**Bit Width**	**Description**
rst_l	input	1	This port is used for reset
FMADD_SUBB_input_opcode	input	2	This port is used for opcode.
FMADD_SUBB_input_IEEE_A	input	STD	This port is used for input operand A
FMADD_SUBB_input_IEEE_B	input	STD	This port is used for input operand B
FMADD_SUBB_output_S_Flags_FMADD	output	3	This port is used for Addition lane Status flags
FMADD_SUBB_output_IEEE_FMADD	output	STD	This port is used to output Addition Lane

Table 3.4: I/O List Of Floating-Point Addition And Subtraction Module

3.1.2.3 Floating-Point Fused Multiply Accumulate Instruction

Instruction Overview Floating-Point Fused Multiply Accumulate instruction has 4 different variants in RISC-V:

1. Fused Multiply Addition

$$O = A \times B + C \tag{3.3}$$

2. Fused Multiply Subtraction

$$O = A \times B - C \tag{3.4}$$

3. Fused Negate Multiply Addition

$$O = A \times B - C \tag{3.5}$$

4. Fused Negate Multiply Subtraction

$$O = -A \times B + C \tag{3.6}$$

As it is evident from the mathematical representations that these 4 variants are naturally the same with fewer differences and they are implementable with the same lane, the other obvious fact is that these instructions are extensions of multiplication and addition/subtraction instructions. Thus, the Floating-Point Fused Multiply Accumulate top module is designed in a way that it caters to 7 of the RISC-V instruction through the same hardware.

1. Fused Multiply Addition
2. Fused Multiply Subtraction
3. Fused Negate Multiply Addition
4. Fused Negate Multiply Subtraction
5. Floating-Point Addition
6. Floating-Point Subtraction
7. Floating-Point Multiplication

Flow Chart The complete explanation of the flow of the whole instruction was discussed in the Instruction Overview. A pictorial representation of the flow can be observed in Figure 3.10.

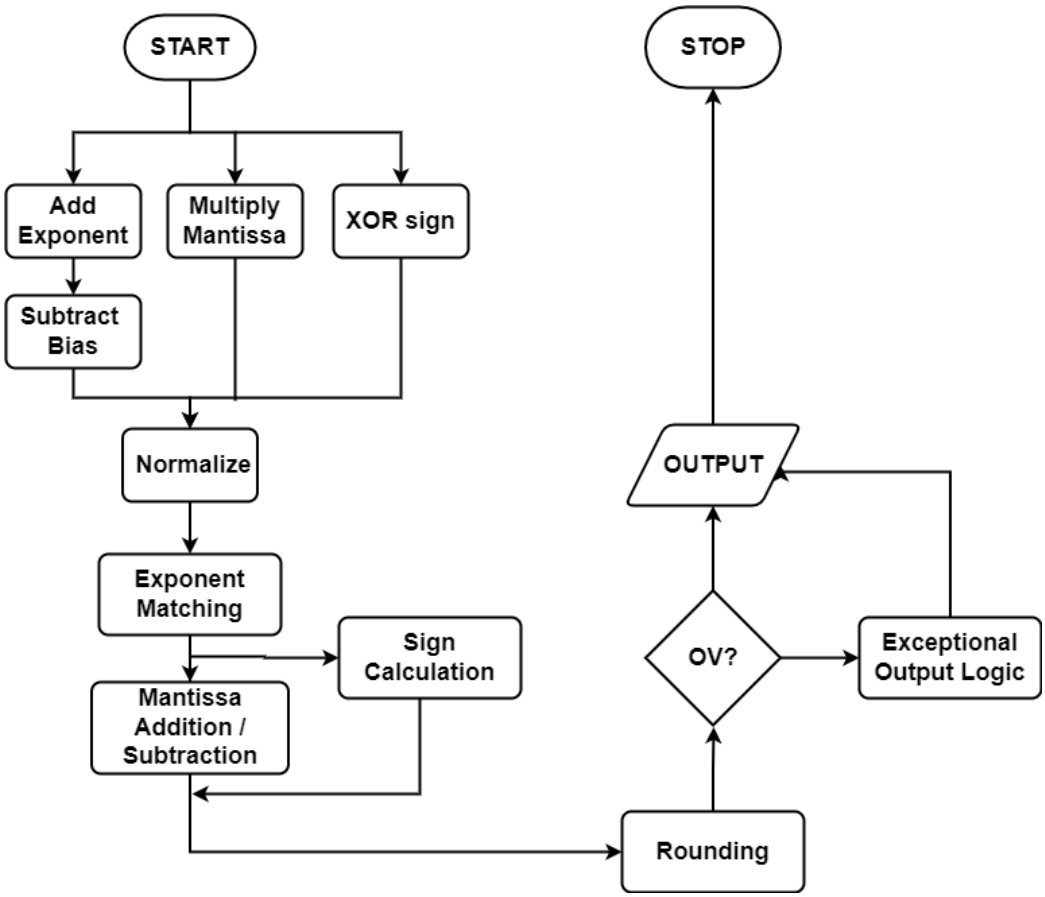

Figure 3.10: Flow Chart Of Floating-Point Fused Multiply Accumulate

Functional Block Diagram The functional block diagram of Floating-Point Fused Multiply Accumulate can be observed in Figure 3.11.

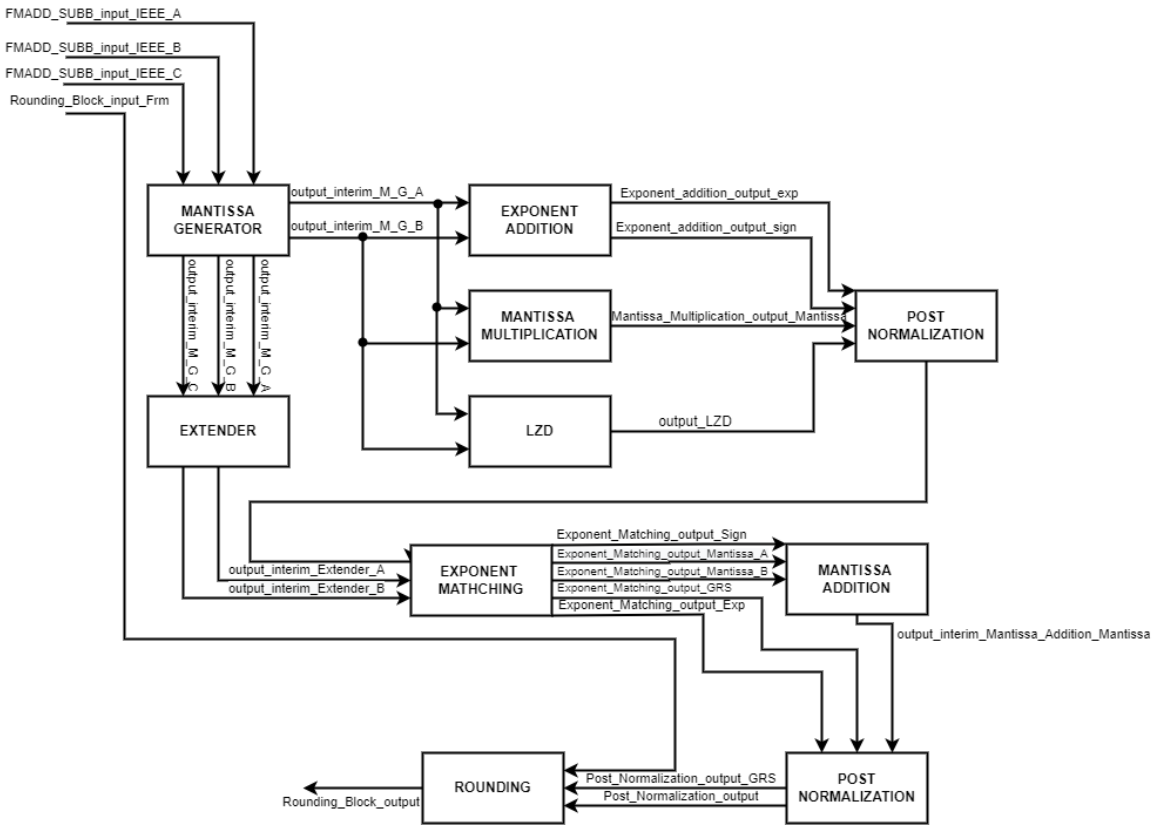

Figure 3.11: Functional Block Diagram Of Floating-Point Fused Multiply Accumulate

I/O List The I/O List of Floating-Point Fused Multiply Accumulate Module can be seen in Table 3.5.

Port Name	Direction	Bit Width	Description
rst_l	input	1	Active low
FMADD_SUBB_input_IEEE_A	input	STD	This port is used for operand A input
FMADD_SUBB_input_IEEE_B	input	STD	This port is used for operand B input
FMADD_SUBB_input_IEEE_C	input	STD	This port is used for operand C input
FMADD_SUBB_input_opcode	input	7	This is a 7 bit one hot encoded signal for opcodes
FMADD_SUBB_output_S_Flags_FMADD	output	3	This port is used for status flags
FMADD_SUBB_output_S_Flags_FMUL	output	3	This is the separate output port for status flags of Multiplication lane
FMADD_SUBB_output_IEEE_FMADD	output	STD	This is the output port for instruction using Add/Sub lane
FMADD_SUBB_output_IEEE_FMUL	output	STD	This is the separate output port for multiplication lane

Table 3.5: I/O List Of Floating-Point Fused Multiply Accumulate Module

RTL Hierarchy The hierarchy of the modules in the RTL can be seen in Figure 3.12.

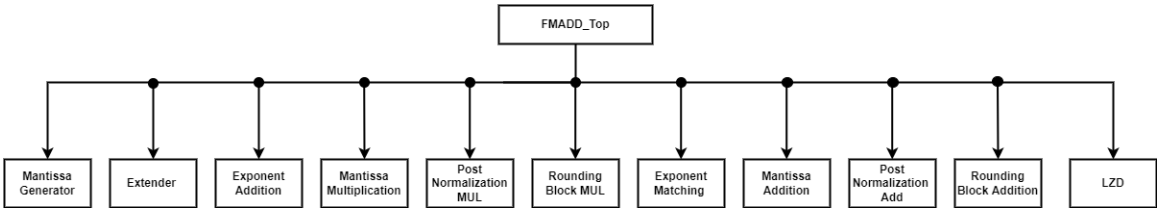

Figure 3.12: RTL Hierarchy Of Fused Multiply Accumulate

3.1.3 Non-Computational Instructions

One of the 3 types of floating-point instructions are Non-Computational instructions. These instructions do not use any sort of arithmetic operation in them.
There are 3 types of Non-Computational instructions in RISC-V specifications.

1. Floating-Point Move Instruction
2. Floating-Point Classification Instruction
3. Floating-Point Sign-Injection Instruction

3.1.3.1 Floating-Point Move Instruction

Instruction Overview Floating-Point Move instruction is used to move data from one register file to another. Floating-Point Move instruction has 2 variants.

1. FMV.X.W
2. FMV.W.X

FMV.X.W variant is used to move data from floating-point register file to integer register file. While FMV.W.X is used to move data from integer register file to floating-point register file. In both variants data bits are not modified while data is being transferred from one register file to another.

I/O List The I/O List of the Floating-Point Move Module can be seen in Table 3.6.

Port Name	Direction	Bit Width	Description
rst_l	input	1	This port is used for reset.
opcode	input	2	This port is used for opcode.
Move_Input_IEEE	input	STD	This port is used for input operand A.
Move_Output_IEEE	output	STD	This port is used to output the data.

Table 3.6: I/O List Of Floating-Point Move Module

3.1.3.2 Floating-Point Classification Instruction

Instruction Overview Floating-Point Classification instruction examines the data in the floating-point register and outputs a 10-bit one-hot-encoded integer data representing the class to which the input data belongs. Table 3.7 shows the data classes that can be identified using Floating-Point Classification instruction and the corresponding output data bits that will be set high on the detection of these classes. Floating-Point Classification instruction does not have any variants.

rd bit	Meaning
0	*rs1* is -∞.
1	*rs1* is a negative normal number.
2	*rs1* is a negative subnormal number.
3	*rs1* is -0.
4	*rs1* is +0.
5	*rs1* is a positive subnormal number.
6	*rs1* is a positive normal number.
7	*rs1* is +∞.
8	*rs1* is a signaling NaN.
9	*rs1* is a quiet NaN.

Table 3.7: Format Of Output Of Floating-Point Classification Instruction

Functional Block Diagram The functional block diagram of Floating-Point Classification Instruction can be observed in Figure 3.13.

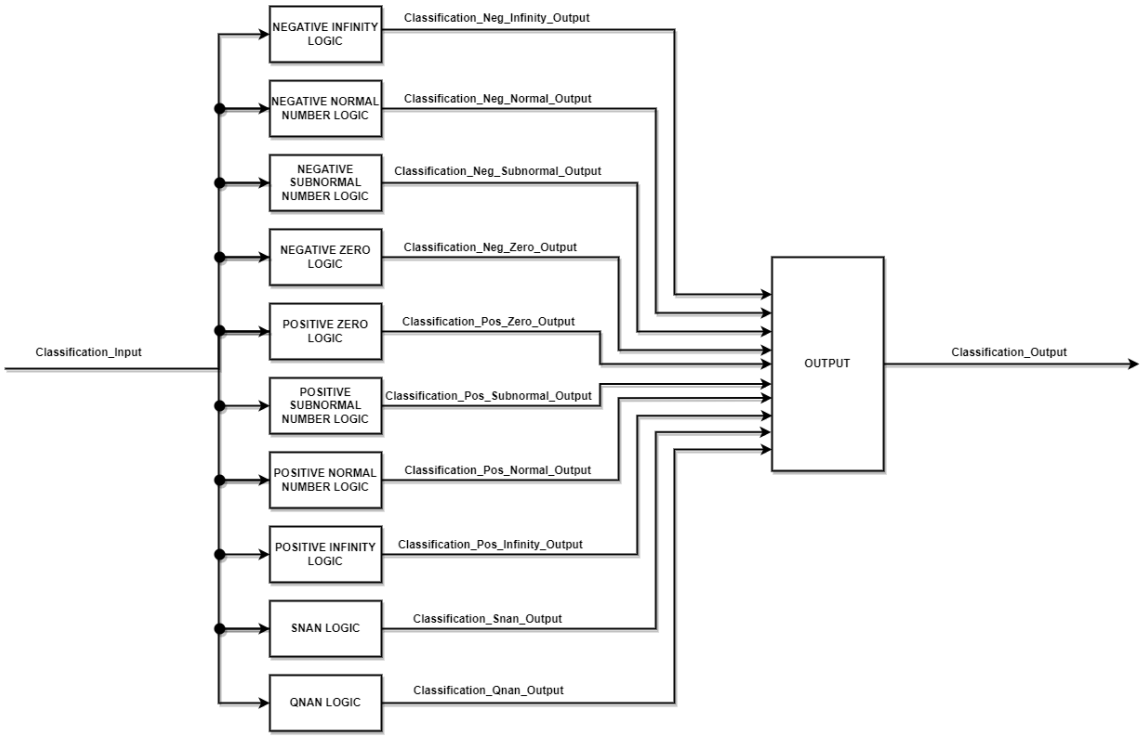

Figure 3.13: Functional Block Diagram Of Floating-Point Classification Instruction

I/O List The I/O List of the Floating-Point Classification Module can be seen in Table 3.8.

Port Name	**Direction**	**Bit Width**	**Description**
rst_l	input	1	This port is used for reset.
opcode	input	1	This port is used for opcode.
Classification_Input	input	STD	This port is used for input floating-point number.
Classification_Output	output	32	This port is used for output integer.

Table 3.8: I/O List Of Floating-Point Classification Module

3.1.3.3 Floating-Point Sign-Injection Instruction

Instruction Overview Floating-Point Sign-Injection instructions are used to generate a floating-point number that takes the exponent and mantissa from one floating-point input and concatenates it with a sign, which is generated either using 2^{nd} floating-point input or both the inputs. Floating-Point Sign-Injection Instructions have 3 variants.

- FSGNJ.S
- FSGNJN.S
- FSGNJX.S

FSGNJ.S variant of the instruction extracts the sign from input B and concatenates the sign with the exponent and mantissa from input A. The concatenated floating-point number is presented at the output.

FSGNJN.S variant of the instruction extracts the sign from input B, inverts the extracted sign of input B, and concatenates it with the exponent and mantissa of input A. The concatenated floating-point number is present at the output.

FSGNJX.X variant of the instruction extracts the sign from both the inputs and XOR them together. Sign generated from the XOR operation is concatenated with the exponent and mantissa of input A. Concatenated floating-point number is presented at the output.

Flow Chart The complete explanation of the flow of the whole instruction as well as its variants was discussed in the Instruction Overview. A pictorial representation of the flow of the whole Floating-Point Sign-Injection Instruction can be observed in Figure 3.14.

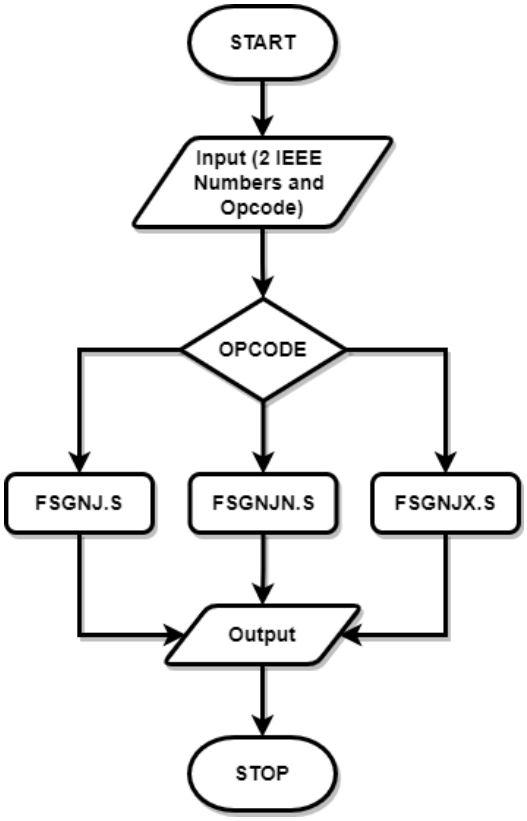

Figure 3.14: Top Flow Chart Of Floating-Point Sign-Injection Instruction

The pictorial representation of the flow of the Floating-Point FSGNJ.S variant can be observed in Figure 3.15.

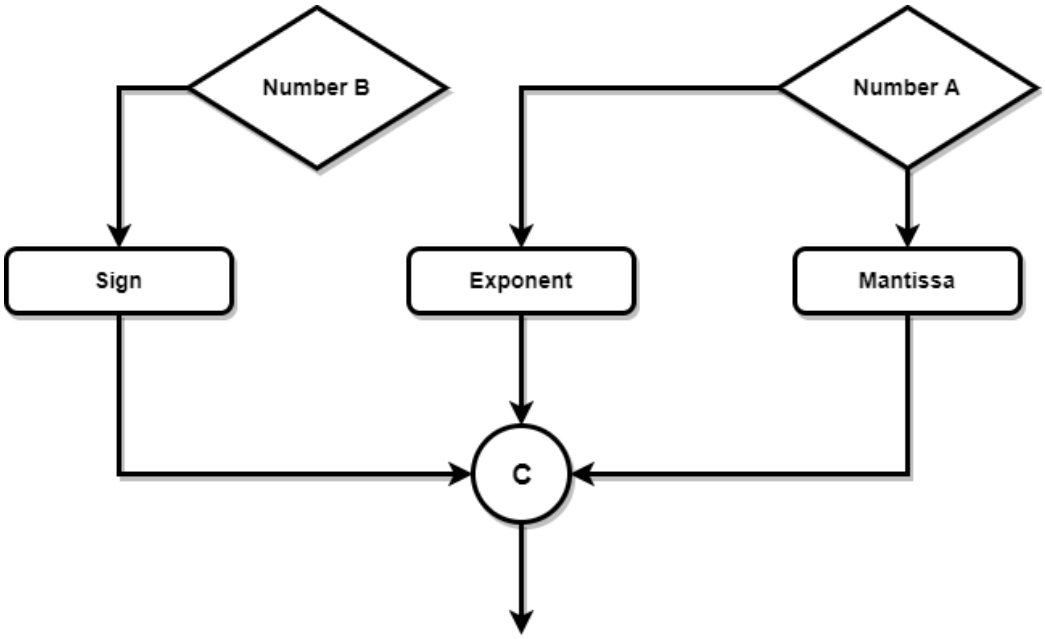

Figure 3.15: Flow Chart Of Floating-Point FSGNJ.S Variant

The pictorial representation of the flow of the Floating-Point FSGNJN.S variant can be observed in Figure 3.16.

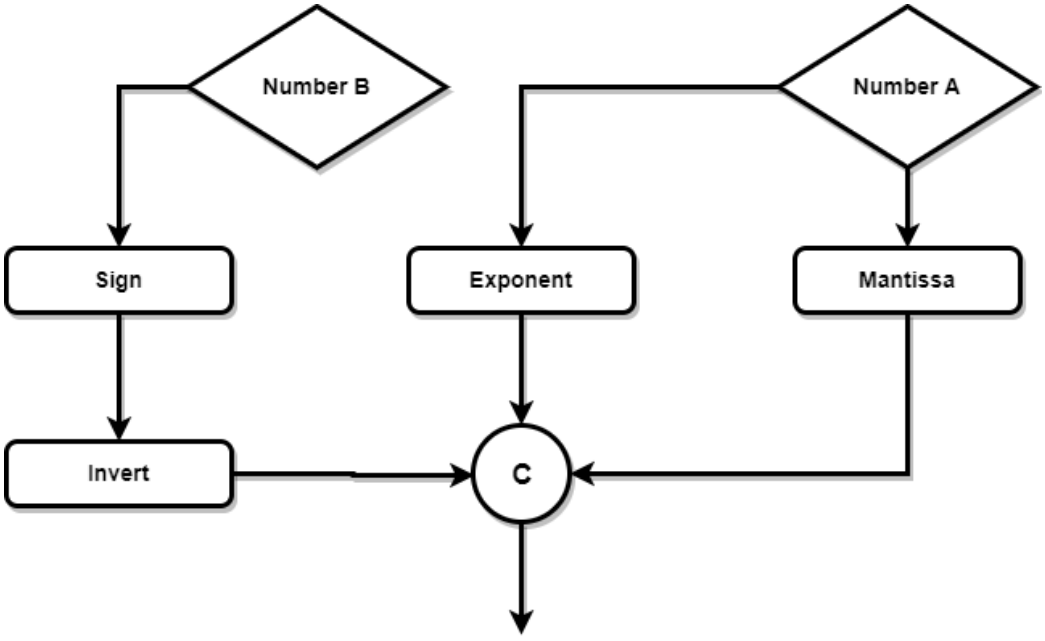

Figure 3.16: Flow Chart Of Floating-Point FSGNJN.S Variant

The pictorial representation of the flow of the Floating-Point FSGNJX.S variant can be observed in Figure 3.17.

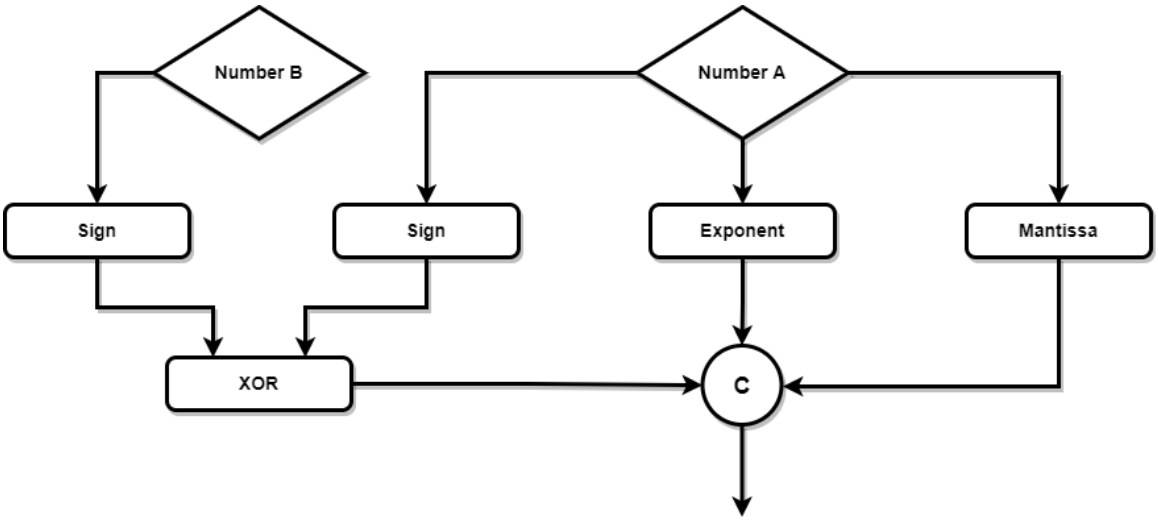

Figure 3.17: Flow Chart Of Floating-Point FSGNJX.S Variant

I/O List The I/O List of Floating-Point Sign-Injection Module can be seen in Table 3.9.

Port Name	**Direction**	**Bit Width**	**Description**
rst_l	input	1	This port is used for reset.
op	input	3	This port is used for opcode.
IEEE_A	input	STD	This port is used for input operand A.
IEEE_B	input	STD	This port is used for input operand B.
IEEE_out	output	STD	This port is used to output the data.

Table 3.9: I/O List Of Floating-Point Sign-Injection Module

3.1.4 Conditional Instructions

One of the types of floating-point instructions are Conditional instructions. Conditional instructions are either used to get the boolean result based on the data or to get the required result based on the condition. The Conditional instructions are separated into two categories which are as follows:

- Floating-Point Comparison Instruction
- Floating-Point Minimum and Floating-Point Maximum Instruction

3.1.4.1 Floating-Point Comparison Instruction

Instruction Overview There are three basic Floating-Point Comparison Instructions in the RISC-V ISA. All Floating-Point Comparison Instructions have 2 floating-point inputs and 1 integer output. No flags are produced in any of these instructions.

1. Feq.s: Floating Equal
2. Fle.s: Floating Less than Equal
3. Flt.s: Floating Less than

Feq.s Feq compares both the incoming operands through the comparison of sign, mantissa, and exponent separately. If both the inputs are the same the 32-bit integer, 1 is presented at the output. In the case of both the operands being different the output is 32-bit integer 0.

Fle.s Fle checks whether Input A is lesser than or equal to Input B or not. When the condition is true the output is 1 and when the condition is false the output is 0.

Flt.s Fle checks whether Input A is lesser than Input B or not. When the condition is true the output is 1 and when the condition is false the output is 0.

Flow Chart The complete explanation of the flow of the whole instruction was discussed in the Instruction Overview. A pictorial representation of the flow can be observed in Figure 3.18.

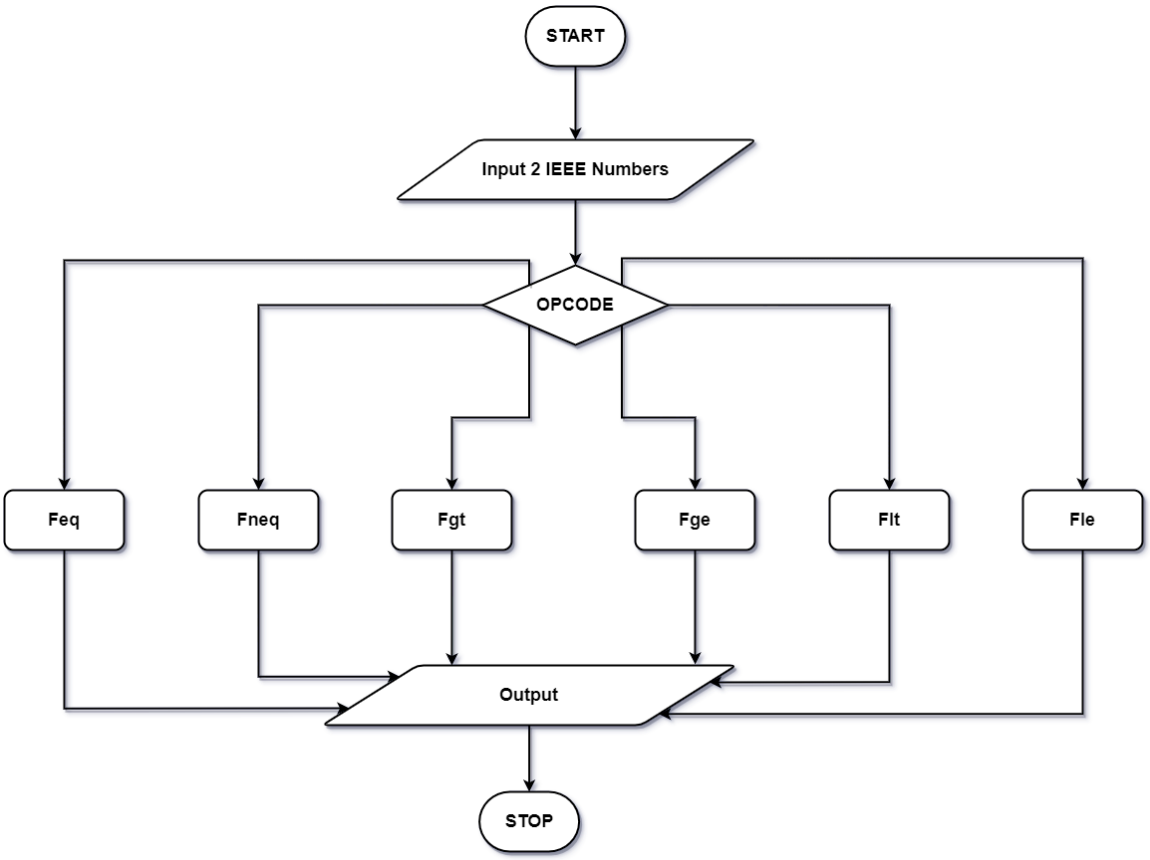

Figure 3.18: Flow Chart Of Floating-Point Comparison

Functional Block Diagram The functional block diagram of Floating-Point Comparison Instruction can be observed in Figure 3.19.

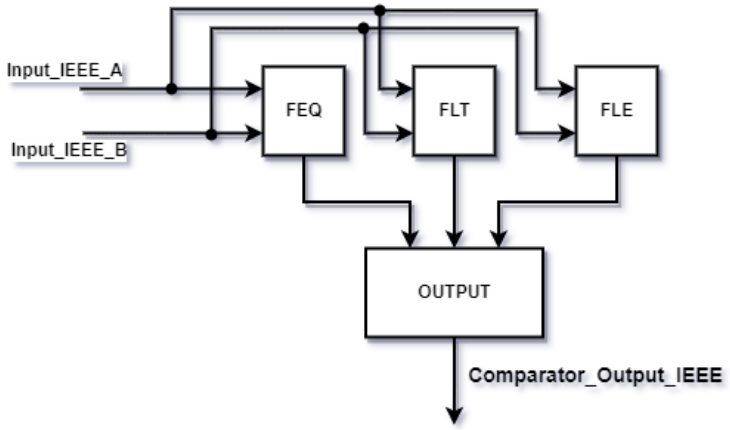

Figure 3.19: Functional Block Diagram Of Floating-Point Comparison

I/O List The I/O List of Floating-Point Comparison Module can be seen in Table 3.10.

Port Name	**Direction**	**Bit Width**	**Description**
rst_l	input	1	This port is used for reset.
opcode	input	8	This port is used for the opcode.
Comparator_Input_IEEE_A	input	STD	This port is used for input operand A.
Comparator_Input_IEEE_B	input	STD	This port is used for input operand B.
Comparator_Output_IEEE	output	32	This port is used to output the result.

Table 3.10: I/O List Of Floating-Point Comparison Module

3.1.4.2 Floating-Point Minimum And Floating-Point Maximum Instruction

Floating-Point Minimum and Floating-Point Maximum Instruction are used to compare two operands for the condition of minimum or maximum.

Instruction Overview

Floating-Point Minimum Instruction The Floating-Point Minimum Instruction has two inputs and one output. Instruction is used to check which one of the inputs is smaller than the other. If Input A is smaller than Input B then input A is presented at the output and vice versa.

Floating-Point Maximum Instruction The Floating-Point Maximum Instruction has two inputs and one output. Instruction is used to check which one of the inputs is maximum. If Input A is greater than Input B then input A is presented at the output and vice versa.

Flow Chart The complete explanation of the flow of the whole instruction was discussed in the Instruction Overview. A pictorial representation of the flow can be observed in Figure 3.20.

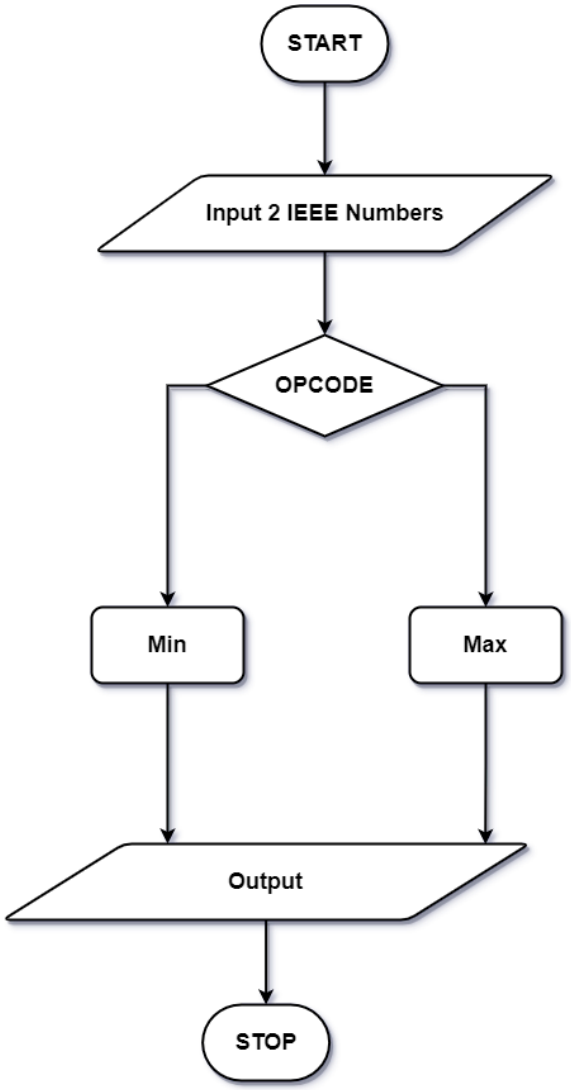

Figure 3.20: Flow Chart Of Floating-Point Minimum And Floating-Point Maximum Instruction

Functional Block Diagram The functional block diagram of Floating-Point Minimum and Floating-Point Maximum Instruction can be observed in Figure 3.21.

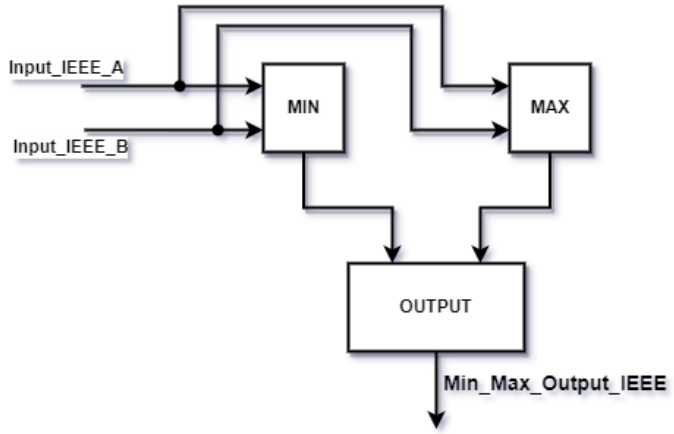

Figure 3.21: Functional Block Diagram Of Floating-Point Minimum And Floating-Point Maximum Instruction

I/O List The I/O List of Floating-Point Minimum and Floating-Point Maximum Module can be seen in Table 3.11.

Port Name	**Direction**	**Bit Width**	**Description**
rst_l	input	1	This port is used for reset.
opcode	input	8	This port is used for the opcode.
Comparator_Input_IEEE_A	input	STD	This port is used for input operand A.
Comparator_Input_IEEE_B	input	STD	This port is used for input operand B.
Min_Max_Output_IEEE	output	STD	This port is used to output the result.

Table 3.11: I/O List Of Floating-Point Minimum And Floating-Point Maximum Module

4 Model Overview

4.1 Model One Overview

In the initially developed version of the SAP-FPU, 15 RISC-V instructions were implemented, and all of the 15 implemented instructions did not have any rounding mode embedded in them. In the initial model, 10 operations were pipelined in nature while the other 5 operations were single-stage instructions. Table 4.1 shows the operations that had pipelining implemented and also the operations that did not have pipelining implemented in them. The table also shows how many stages the pipeline was distributed for each operation. RTL of the initial model was developed using Behavioral modeling and instead of a modular design, the initial model was developed using functional programming.

Instructions	**Cycles per output**
Float to Integer, Integer to Float, Floating-Point Comparison Instruction (6 Variants)	1
Addition, Subtraction, Multiplication, Division, Square root	4
Fused Multiply Add, Fused Multiply Subtract	6

Table 4.1: Cycles Per Output For Each Instruction

Table 4.2 shows the algorithms used for different operations in the initial model, algorithms for Floating-Point Conversion Instruction and Floating-Point Comparison Instructions were completely self-made. As there are no general algorithms available for Floating-Point Conversion Instruction and Floating-Point Comparison Instructions. While for the arithmetic operation of Addition, Subtraction, Multiplication, Fused Multiply Addition, and Fused Multiply Subtraction a combination of self-made algorithms plus general algorithm were used. The algorithms developed for arithmetic operations follow the same general flow as described in the literature. However, the algorithms to execute the flow and perform the operations were completely self-made. For example, to implement addition, the initial model of the SAP-FPU followed the same 3 steps/blocks as described in the literature, exponent matching, mantissa addition, and post-normalization. However, algorithms to implement all these blocks were completely self-made since no detail is available in the literature regarding their implementation. In the initial model, for Division and Square root, the multiplicative algorithm of Newton-Raphson was used. Newton-Raphson was preferred over other algorithms as it is faster due to the use of fast multipliers, unlike subtractive algorithm as it depends on subtracting circuitry. One important step in the operation of floating-point numbers is normalization (making MSB 1 in mantissa), for normalization in the initial model an enhanced priority bit generator capable of calculating the number of leading zeros in a number was designed, developed, and used.

Instructions	**Algorithms**
Float to Integer, Integer to Float, Floating-Point Comparison Instruction (6 Variants)	Self Made
Addition, Subtraction, Multiplication, Fused Multiply Add, Fused Multiply Subtract	General Algorithms used for FP numbers
Division, Square root	Newton-Rapshon (Multiplicative Algorithm)
Leading Zero Detection	Enhanced Self Made Priority Encoder

Table 4.2: Algorithms Used For Each Instruction

The initial model also used an Input validation block, every input that was applied to the FPU first went to the input validation block, in the input validation block it was checked if the input is exceptional data or not. In case any exceptional data was detected, the IV block stopped the data from passing any further and generated the output corresponding to the selected operation and the exception detected and present the output at the output port. Input validation block also handles arithmetic operation cases in which the answer is predetermined. The table in Figure 4.1 shows the inputs that were handled by the input validation block in model 1.

Inputs	Exception Type				
	Infinity	Zero	QNAN	SNAN	No-computation Required
0+0		✓			
A*0		✓			
A*∞	✓				
A+ ∞	✓				
∞ + ∞	✓				
∞ - ∞			✓		
Over/Under Flow				✓	
A+0/A-0					✓
A-A		✓			
A/0	✓				
0/0			✓		
0/A		✓			

Figure 4.1: Input Validation Conditions And Outputs

Top Architecture of SAP-FPU Model One is shown in Figure 4.2.

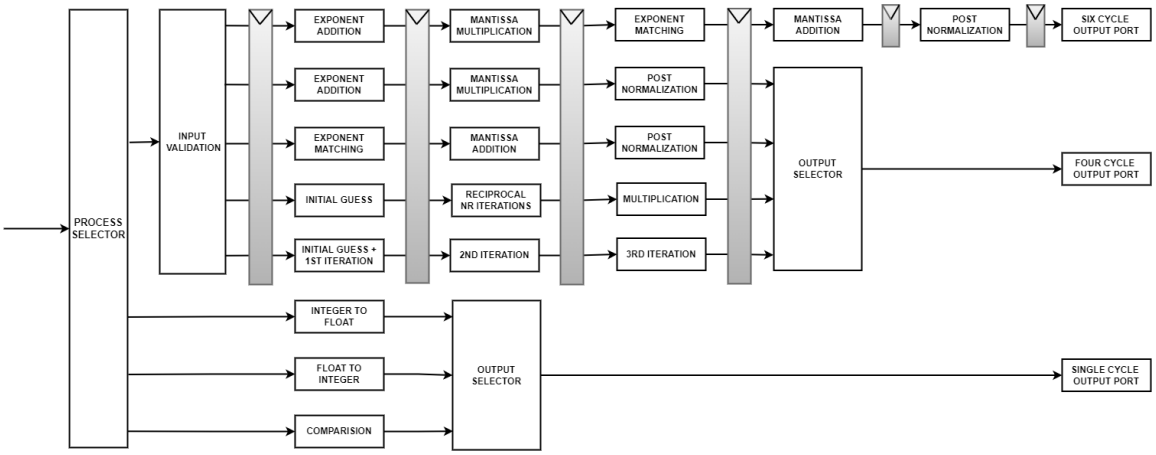

Figure 4.2: Top Architecture SAP-FPU Model One

4.2 Model Two Overview

After feedback from the synthesis of "Model 1" and detailed analysis, it was soon realized that SAP-FPU Model 1 had some serious problems. Some of those problems were of such stature that they caused redesigning of SAP-FPU from scratch. These problems were:

1. In SAP-FPU Model 1 every asynchronous portion of the logic in SAP-FPU was designed via behavioral modeling and since the development was carried out in Verilog thus the ambiguities of registers and latches occurred. Therefore, when SAP-FPU was synthesized some unwanted latches were created and this for sure was negative for the practical usage of FPU.

2. As it is evident from Figure 4.2 that every instruction is broken up into its sub-modules, in SAP-FPU Model 1 each of these sub-modules were consuming 1 clock cycle which means, for a single instruction like Floating-Point Addition, Floating-Point Subtraction, or Floating-Point Multiplication SAP-FPU was consuming 4 Clock cycles. This was a problem as while working with a core SAP-FPU would have faced issues of data dependencies or data hazards because of the Bypass stage of the processor.

3. In the development of SAP-FPU Model 1, all the instructions were designed without a sub-normal number range incorporated in the algorithms.

4. In SAP-FPU Model 1, some instructions which had similar data paths were not combined to have a single/common data path. This would have cost a lot of area in the synthesis phase of the design thus this was a problem to be dealt with.

All these problems had to be dealt with as they were a hurdle in the practical usage of the FPU thus a new model of SAP-FPU namely Model 2 was developed which had remedies to all these problems in the following ways:

1. All the asynchronous logic (instruction data path) was redesigned using data flow modeling in Verilog to avoid the creation of latches in synthesis.

2. All the instructions in SAP-FPU were designed to consume only one operational clock cycle.

3. The subnormal or De-normal numbers range was incorporated in the SAP-FPU, this caused redesigning of each and every instruction as algorithms had to go under major changes.

4. All the instructions with common data paths were designed to have common/combined hardware to ensure resource sharing and make the area consumption of SAP-FPU optimized.

The input validation block is common to all the sub-modules in Model 1 and 2 however in Model 2 some of the conditions which the input validation block deals with are updated along with its controlling technique for exceptions of a different sort.

Input validation block deals with special cases and raises a flag named Exceptional Flag which indicates that the incoming input to the FPU is either capable of causing an exceptional situation such as A/0, QNAN, –QNAN or it is an input which does not require any computations such as A*1, A/1 or A-A, Input validation block also generates the relevant output for such cases namely Exceptional Result. In either case, a high Exceptional Flag assures that the output selector selects the output coming from the Input Validation block for flopping on the output register.

The Top Architecture of SAP-FPU Model Two is as follows,

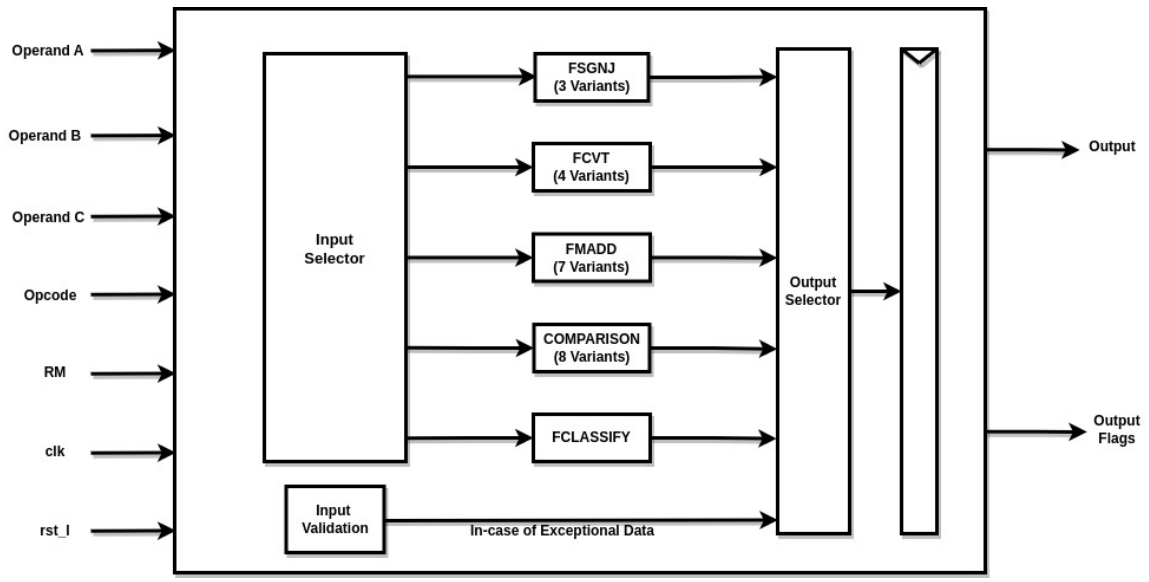

Figure 4.3: Top Architecture Of SAP-FPU Model Two

I/O List The I/O List of SAP-FPU Model Two can be seen in Table 4.3.

Port Name	Direction	Bit Width	Description
Clk	Input	1	This port is used for clock
rst_l	Input	1	This is the port for global reset
Frm	Input	3	This port is used for input of desired rounding mode
Operand_A	Input	STD	This is the operand A (IEEE-754) input port
Operand_B	Input	STD	This is the operand B (IEEE-754) input port
Operand_C	Input	STD	This is the operand C (IEEE-754) input port
FPU_resultant	Output	STD	This is the output (IEEE-754) port for floating register file
FPU_resultant_rd	Output	32	This is the output (IEEE-754) port for Integer register file
S_Flags	Output	5	This I the output port for status flags
Exceptional Flag	Output	1	This port is used for exceptional Flag output
Interrupt Pin	Output	1	This port is used for indicating that the operation inside FPU has to trigger an interrupt

Table 4.3: I/O List Of SAP-FPU Model Two

RTL Hierarchy The RTL Hierarchy of SAP-FPU Top Model Two can be seen in Figure 4.4.

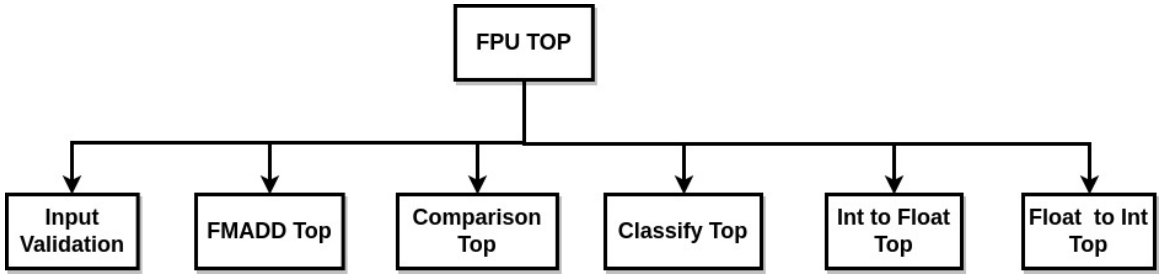

Figure 4.4: RTL Hierarchy Of SAP-FPU Top Model Two

5 Functional Verification

5.1 Software Simulation

The next phase after the completion of RTL is its functional verification, this is carried out by the following two means:

1. Open Loop Testing
2. Close Loop Testing

5.1.1 Open Loop Testing

The verification through this method is carried out by means of writing Verilog test benches. These test benches have the Top modules of the design under test instantiated in them. Test vectors are provided as a stimulus to the design and the results are stored in the VCD file which is then viewed via GTKWAVE in Figure 5.1.

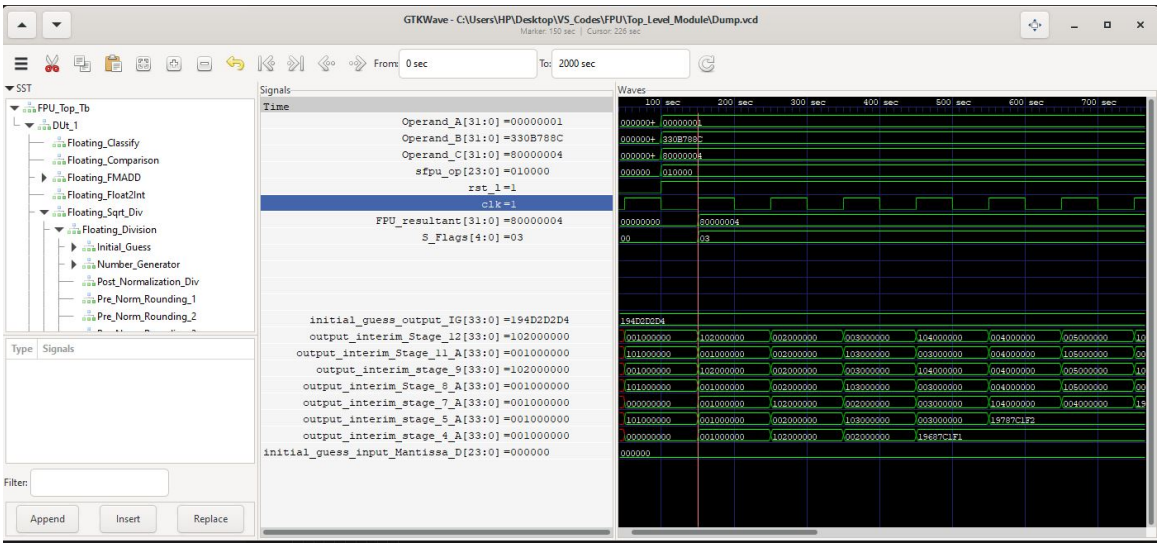

Figure 5.1: GTKWAVE Window

The verification in this method is carried out manually, that is, the test vectors which are applied in the test bench as a stimulus are provided to a golden model manually. Then the results are compared after viewing the design under the test's VCD file in GTKWAVE. This may serve as a basic verification method or a method that could be used for debugging purposes however the tediousness in verification through this method makes it unfit for formal design verification.

5.1.2 Close Loop Testing

Open loop testing may serve as the basic testing mechanism for testing out the functionality however, by no means is it the formal verification methodology. This is mainly because the verification in this method is carried out manually, which is a tedious task, and thus it is not humanly possible to carry out an adequate number of tests on specific designs. Another reason for close loop testing is that in the open loop method the stimulus to the design under test is not a randomly generated number therefore testing environment may fail to catch probable corner cases where an error may occur.

For Close loop testing python's co-routine, COCOTB is used. In the course of completion of this project, the testing environment of close loop testing was under continuous upgradations as per evolving testing requirements. These upgrades are defined as follows.

5.1.2.1 Initial Model

This model being the first is the simplest implementation of Close loop testing (from now on referred to as CLT). The general signal flow for this model is understandable in Figure 5.2.

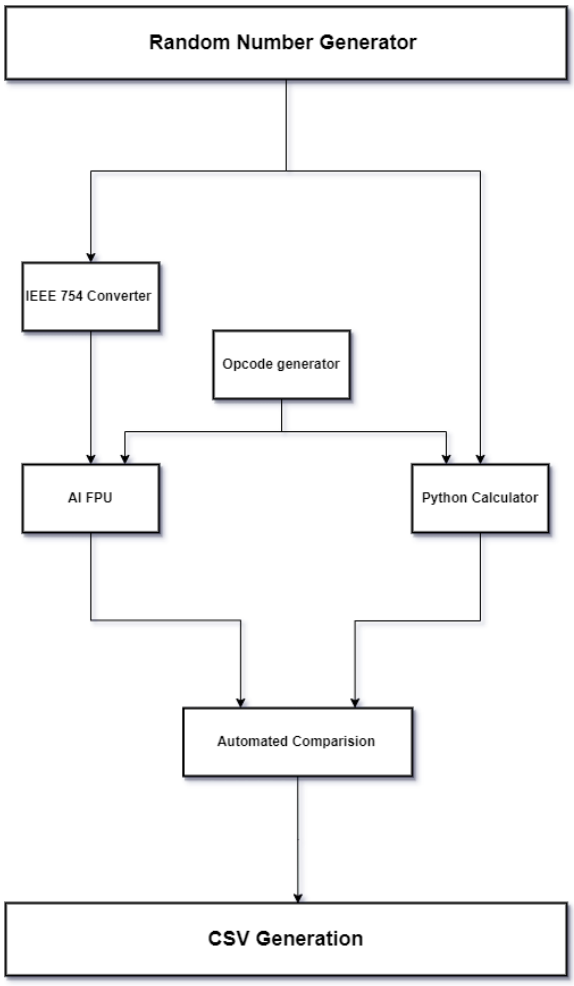

Figure 5.2: Block Diagram Of CLT Initial Model

At first random Floating-Point numbers (in decimal number system) were generated using random.Uniform() function. These numbers were then converted into their IEEE-754 equivalence using a custom python Function. Opcode generator function was used to generate a controlling bit which indicated which kind of instruction is to be tested for the generated data. Both these inputs (Randomly generated number, opcode) were then given as a stimulus to SAP-FPU. In parallel the randomly generated number and opcode were also fed to a python calculator where the calculation was carried out using simple arithmetic operators using Numpy library, the arithmetic in this function was carried out in a decimal number system. Python calculator is a custom function that served as a "Golden Model" in this initial model of CLT thus, results from SAP-FPU and the golden model were then automatically compared using python's built-in constructs. This process was repeated for thousands of times using Loops and all the data for each iteration of the loop was stored in a data frame. At the termination of this loop, the data frame was converted into a CSV file and hence a log of these random tests was created.

5.1.2.2 Model 1

Initial model was crude in its state, this was mainly because of the following reasons:

1. Random number generation with python's built-in construct random.Uniform() was not as random as desired.
2. SAP-FPU from its initial days had PULP FPU to compete with as a benchmark however the golden model in "Initial Model" was a python-based calculator.
3. Besides results, SAP-FPU was also designed to produce correct status flags, for this purpose testing mechanism of the "Initial Model" was inadequate again for the same reason of the golden model being a python-based calculator rather than an FPU.
4. SAP-FPU was also to be tested for correct functionality with all the possible rounding modes available in RISC-V thus this was a shortcoming in the initial model as the python calculator could not implement arithmetic as per the rounding modes of RISC-V.

Keeping these two issues in mind "Model 1" of CLT was developed with upgradations. The general signal flow of Model 1 is understandable by Figure 5.3.

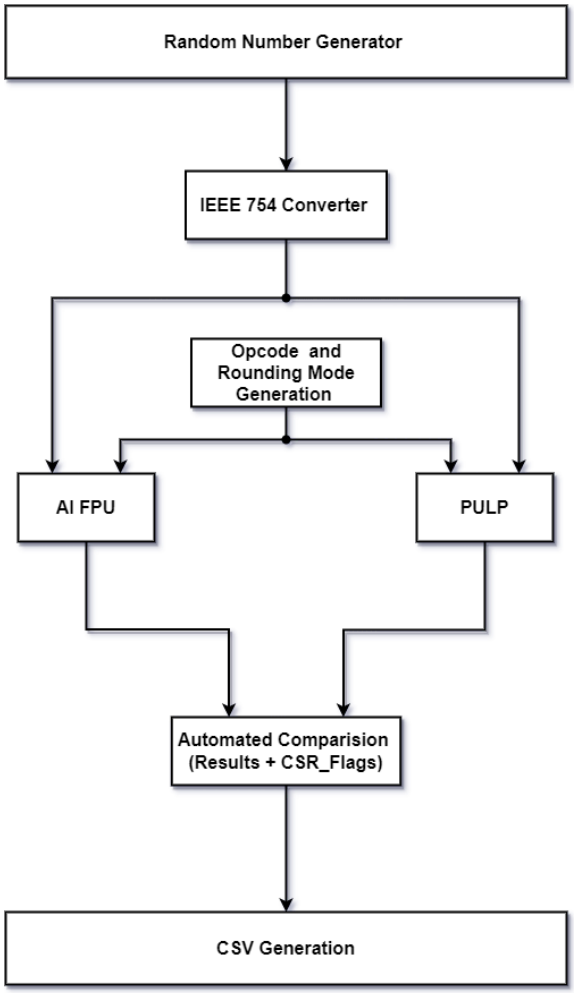

Figure 5.3: Block Diagram Of CLT "MODEL 1"

The first phase of the testing framework was the same that is, Random Number generation, however in this model contrary to the previous model the random number generation was updated

in order to fulfill the evolving requirements of more wide random number generation range. For this purpose, a custom random number generator in python was made.

Random Number Generator The maximum number that can be represented using IEEE-754 Single Precision Standard is 3.4028235E38 while the minimum number that can be represented is 1E-45 (The range of minimum number is greater than the maximum number due to subnormal format). The designed Random Number Generator generates exponent randomly from the range 0 to 45, to incorporate the fractional numbers sign of the exponent is then also randomly decided and attached to the exponent, Figure 5.4 shows the frequency of the exponent both the positive and negative exponents (in case the number is not representable in IEEE standards the converter outputs infinity or underflow depending on whether the number was too large or too small).

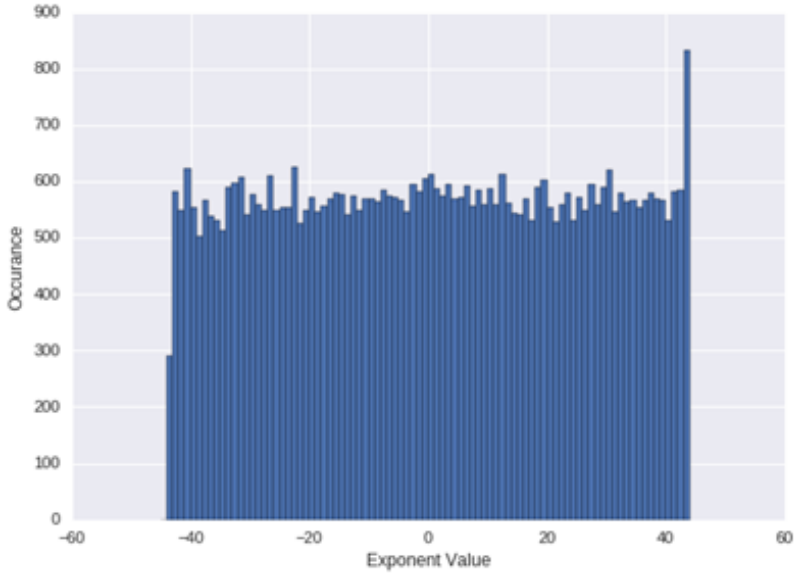

Figure 5.4: Frequency Of Exponent Values

The significand of the numbers that are representable in IEEE-754 Single Precision, when written in decimal format can go as high as 8 and as low as 1, therefore as visible in Figure 5.5 significand generated are within the range of 1 to 8. Similar to the sign of exponent, the sign of significand is also randomly decided and attached to it.

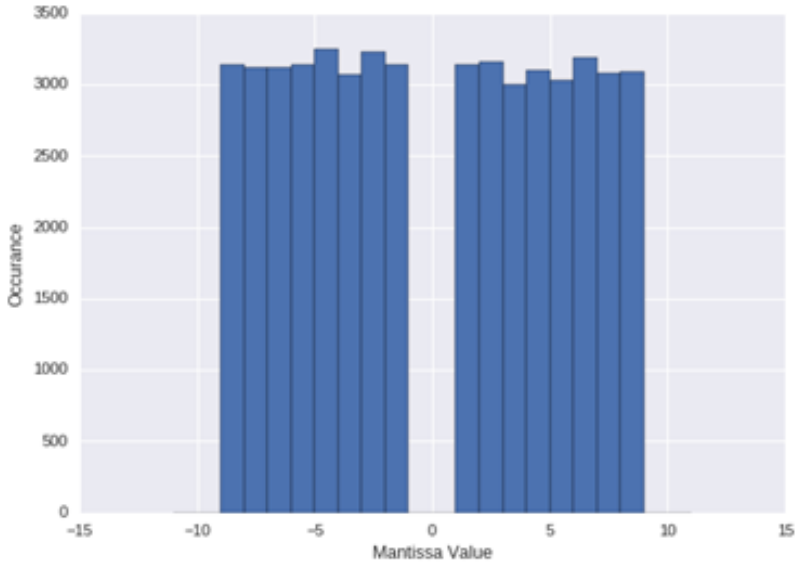

Figure 5.5: Histogram Of Randomly Generated Mantissa Value

Visual Analysis of the histograms shown in Figure 5.5, and Figure 5.4 shows that the exponent and mantissa of all the weights are generated with approximately the same frequency which ensures that the generation of numbers is uniformly distributed over the entire range of floating-point standard (Single Precision) instead of numbers forming a cluster for smaller range.
After the random generation of numbers (Data Points / Operands) the next step in the testing framework was to pass this data as a stimulus to SAP-FPU and PULP at the same time. In other inputs, Instruction opcode and rounding modes were passed to both the FPUs as well. The results coming from both the FPUs were compared in IEEE-754 standard (for Resultant) and Hexadecimal numbers (for status Flags). This process was repeated again for numerous times and results were logged in the data frame which was finally converted into CSV format.

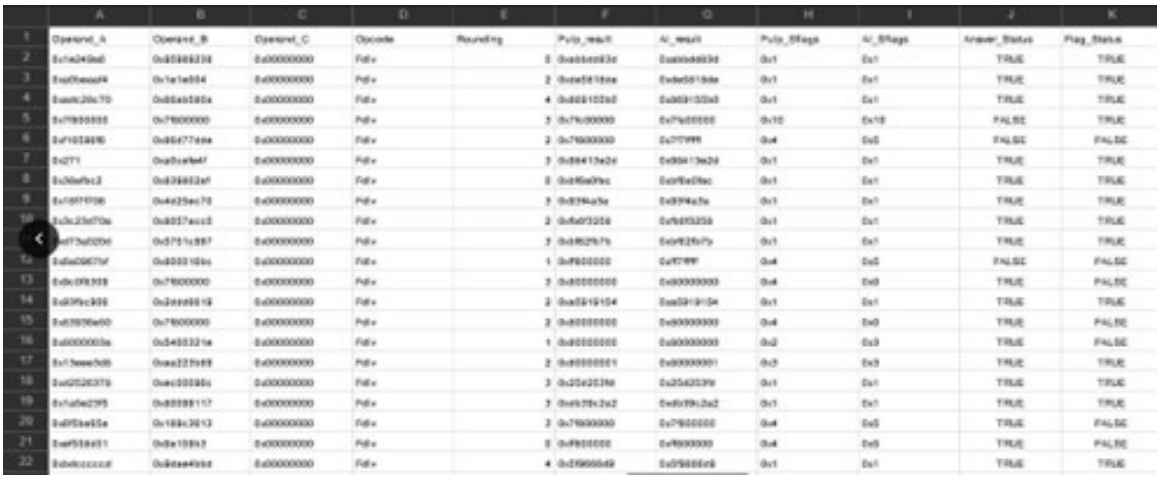

Figure 5.6: Test Data Logging In CSV

5.2 Hardware Emulation

In order to perform the functional testing of a design-under-test hardware emulation is a frequently used technique. Hardware emulation is a technique that incorporates a hardware design into a reconfigurable (for example, FPGA-based) prototyping platform. In a realistic performance environment, both hardware and software running on the hardware can be tested in this manner. Hardware emulation is generally used to debug and verify ASIC designs. This step is generally the last step of verification before finalising the ASIC design for physical layout (APR).

5.2.1 FPGA

The Altera® Cyclone® V SoC Development Kit offers a quick and simple approach to develop custom ARM® processor-based SOC designs accompanied by Altera's low-power, low-cost Cyclone V FPGA fabric. This kit supports a wide range of functions, such as:

- Processor and FPGA prototyping and power measurement

FPGA I/O interfaces

- x4 push buttons
- x4 LEDs

Power

- Laptop DC input 14—20 V adapter

Clocking

- Four-output programmable clock generator for FPGA reference clock inputs.
- 50 MHz single-ended oscillator for FPGA and MAX V FPGA clock input.

5.2.2 Schematic Of The Cyclone V SX SoC Development Board

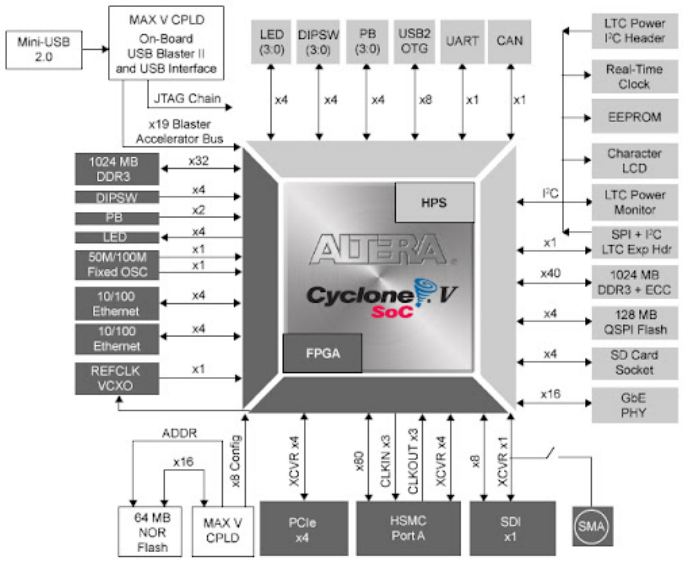

Figure 5.7: Functionality Of The DE1 Altera Cyclone V SoC FPGA

The above Figure 5.7 is showing all the functionality of the DE1 Altera Cyclone V SoC FPGA. The Figure 5.8 shows the I/O diagram of the FPGA used for testing of SAP-FPU. The clock is used to synchronise all the blocks, LEDs are used to indicate the output level, Push button is used to reset the whole program running on the hardware and there are six Hex seven segment display units which are used to display the output. The software used for burning the codes on FPGA is Quartus Prime Lite Intel.

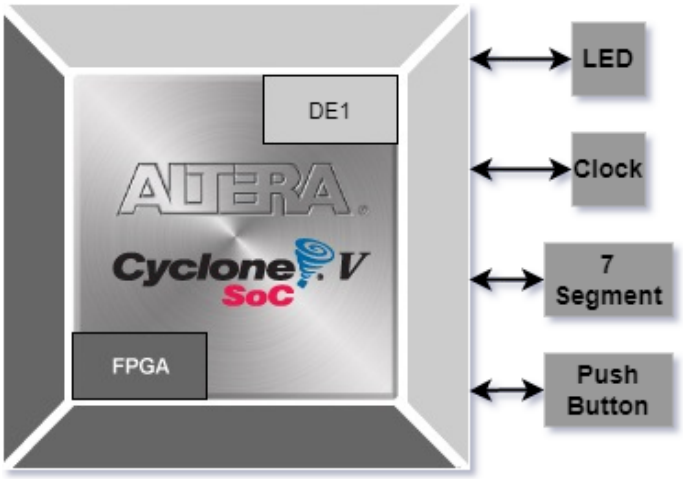

Figure 5.8: I/O Diagram Of FPGA

The SAP-FPU is a standalone execution unit however, in order to make the SAP-FPU compatible with the standards of selection for MPW6 program a controlling logic was required to be made around the stand alone execution unit (SAP-FPU). Thus for this purpose the SAP-FPU was packaged inside a core, which had all the standard CPU stages that are, fetch, decode, execute, and write back.

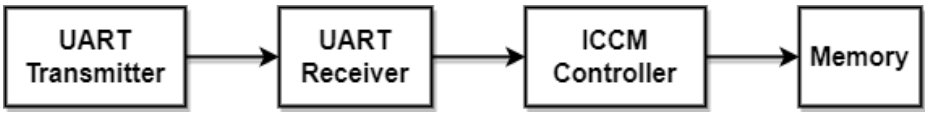

Figure 5.9: Block Diagram Of Idle Stage

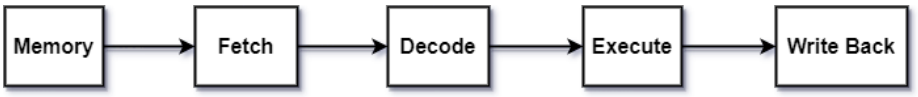

Figure 5.10: Block Diagram Of Core

As mentioned earlier SAP-FPU was a part of the execution unit of the core built. In order to control the data path of the core a Finite State Machine (FSM) was designed, as per the design requirements.

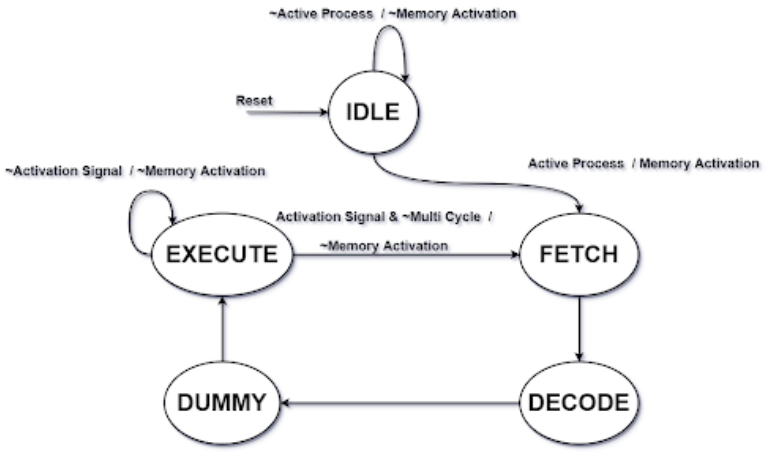

Figure 5.11: Designed FSM For Data Path Control

The designed RTL was ready to be burnt on FPGA however few configurations were required. As the results coming from the process were to be visually analysed, on seven segment displays, thus a clock of 50 MHZ (Fpga default clock speed) was not appropriate as then it would have been physically impossible to analyse results, thus need of clock divider arose as a potential solution to this problem, clock divider was designed in RTL and burnt on FPGA in order to achieve lesser clock rates and ensure smooth testing environment. The other challenge was to use/design a memory which would store the Hex file of the program which is to run on the designed test core. Since the results were to be displayed on seven segment LEDs thus the requirement of BCD decoder arose which would take 32bit binary form result from SAP-FPU and convert it to Hexadecimal form to make it displayable on seven segment displays.
Thus, the final design ,as shown in Figure 5.12, has the following, At first the FPGA clock is divided to lower clock rates through clock divider, The resulting clock from this module is a common clock for all the stages of the core designed (Fetch, Decode, execute) also the same cock is given to the BCD decoder and memory (containing the hex file of the program to run). In terms of data flow, the program hex file is saved in the memory through the UART controller, memory contains RISC-V instructions, these instructions are fed in to core (SAP-FPU core) and the results are synchronously transmitted to BCD decoder from where these results are displayed on seven segment displays. Three precision levels (Single, Half, Bfloat-16) are verified through FPGA (hardware emulation), in order to display all the 32 bits if single precision 8 more simple LEDs are used in combination with the 6 available seven segment displays .

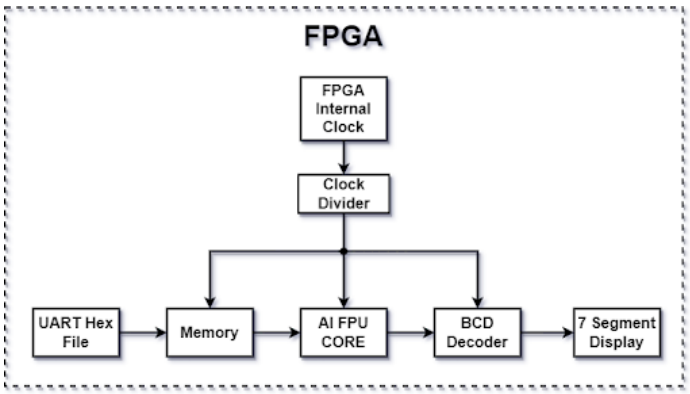

Figure 5.12: Block Diagram Of FPGA

Architecture The whole architecture can be seen in Figure 5.13

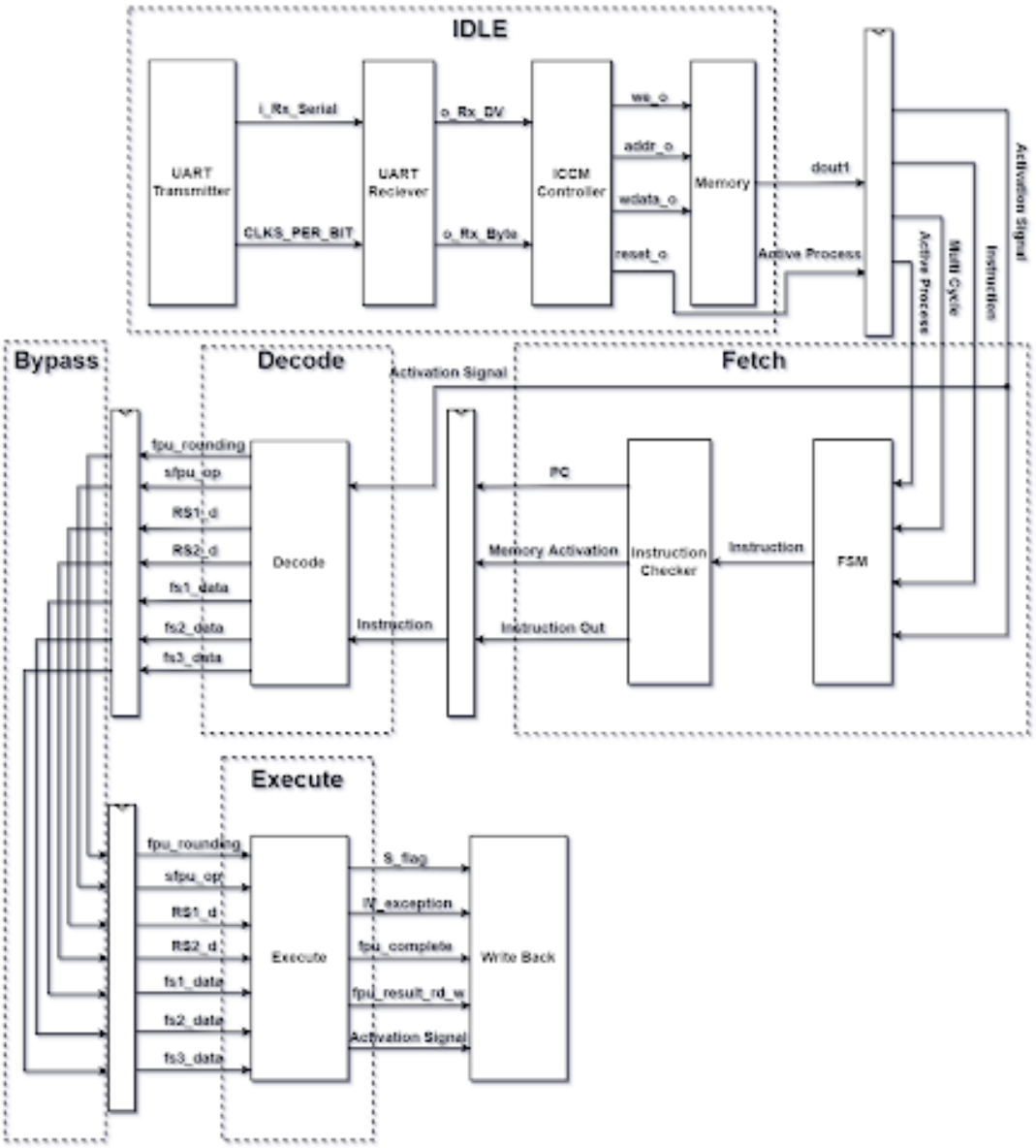

Figure 5.13: Architecture Of Core

6 APR

6.1 Physical Design Overview

After the completion of RTL designing and verification the next step which is essential for any design is the physical designing.
Physical Design of any design is divided into following steps:

1. PDK and Process Details
2. Synthesis, Mapping and Static Timing Analysis (STA)
3. Floor plan
4. Placement
5. CTS
6. Routing
7. Checks
8. Final GDS

PDK and Process Details is the first step in every physical design. Some parameters must be defined/decided before starting the flow which include the technology, number of metal layers, PVT, etc.
The second step of physical design is **Synthesis, Mapping and Static timing analysis (STA)**. Synthesis is a process by which the RTL or the logical design is converted into a circuit using the components of standard cell library (SCL). This results in a gate level netlist. The netlist is the actual implementation of the behavioural model. It is the description of the physical connections between the IOs of different standard cells.
Mapping is basically the placement of standard cells of a particular library in the netlist. The mapping is also done while synthesising the code.
Static timing analysis is a methodology to determine and validate the timing performance through extensive checks of every path for timing violation. This step is executed multiple times throughout the APR flow. For the design to be deemed ready for tape out, the timing violations of the design must be brought to zero. STA is only applicable on synchronous sequential designs.
The third step of physical design is **Floor plan**. Floor planning mainly deals with assigning the die area, core area and number of rows in the design space. The design chip consists of three major sections:

1. Core Area
2. IO Pads
3. Die Area/Package

Core Area It is the space where all the standard cells are placed. All the design logic is implemented using the standard cells in this area. It is the innermost area of the chip.

IO Pads IO Pads Area is the surrounding space around the Core Area.

Die Area/Package It is the complete package with Core Area + IO Pads. Hence the Die Area is always greater than the Core Area of the design chip.

At the floor plan step, the Core Area + Die Area is decided which remains constant throughout the process till the GDSII file is generated. However, there is another very important parameter that is decided in this stage. In the Core Area, the tool makes a number of rows in which the standard cells would be placed.
The fourth step of physical design is **Placement**. In this step the flow starts placing the standard cells inside the core area. The placement is divided into 2 steps:

1. Global Placement
2. Detailed Placement

Global Placement inserts all the standard cells into the core area haphazardly. There is no sequence or order, some standard cells might even overlap each other.
However, in the detailed placement each and every cell is placed properly inside the rows. As the height of the standard cell is constant for a technology while the width of the cells might differ with respect to each other. Hence the standard cells are placed in alignment with the rows.

The fifth step of physical design is **CTS**. This step mainly deals with sequential circuits which involve the usage of clocks. If the design is a purely combinational circuit with no clock, then CTS can be skipped. After placement the flow would then go to the routing stage.
So far, we have been assuming an ideal clock being fed to every register as shown in the figure below.

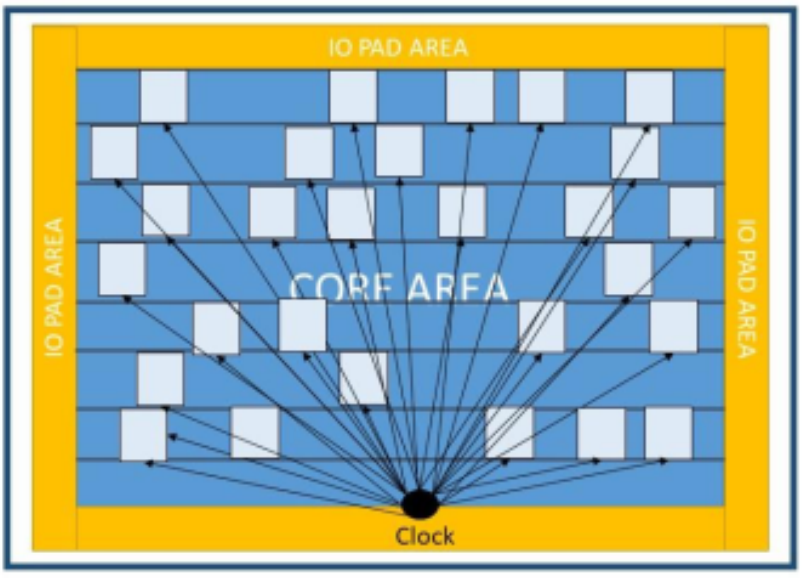

Figure 6.1: Bad Clock Connection

The above connection looks simple and good. However, treating the clock connection like a net connection is not feasible due to a plethora of reasons, some of which are mentioned below:

- Timing
  (a) Skew – The highest concern in clock networks.
- Power
  (a) The clock is the major consumer of power because it switches at every cycle.

- Signal Integrity

    (a) For long routes, the clock is prone to noise creating distortion in the signal.

Skew Clock skew is the maximum difference in the arrival time of a clock signal at two different flops.
The sixth step in physical design is **Routing**. Routing is a step followed by the placement of standard cells. In OpenLane flow, routing is executed automatically through scripts. The task of the router is to precisely define the paths on the layout surface enabling conductors to carry electrical signals. The conductors are responsible for interconnecting the pins and the standard cells on the layout and thus forming a routing grid. Since the routing grid is quite large, routing is performed using a divide and conquer approach.
In light of the divide and conquer approach, routing is performed in two stages in an OpenLane flow:

1. Global Routing
2. Detailed Routing

During Global routing the routing guides are roughly generated to outline the implementation of actual routes whereas the detailed routing enables the wires to follow those routing guides and build interconnects.

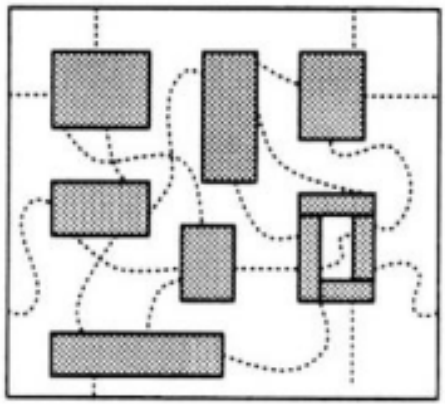

Figure 6.2: Global Routing

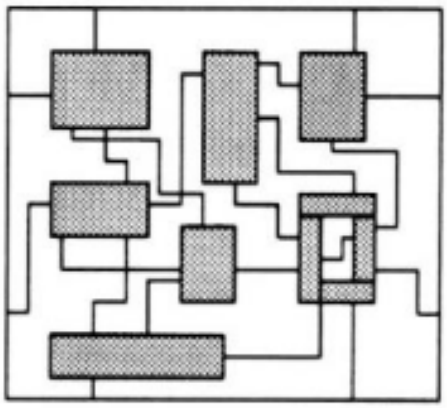

Figure 6.3: Detailed Routing

The seventh step in physical design is **Checks**. Checkers usually involve checking the generated physical design for any design rule violations. This is usually to check if the router and placer have correctly routed the grid and placed the cells. The design is checked for any overlapping cells or short circuits and inspects any Layout Vs Schematic (LVS) error that may include any unmatched pins or short/open circuits between nets that should have been connected.
After performing all the steps finally we move towards the eighth step of physical design which is **Final GDS**.
For the MPW-6 shuttle program three variants of SAP-FPU were submitted from which two variants got selected. The three variant which were submitted are as follows:

1. Single Precision (SP)
2. Half Precision (HP)
3. Bfloat-16 (BP)

6.2 Physical Design Of Single Precision (SP)

Single precision is one of the floating-point standards that have been defined in the IEEE-754 floating-point standard. In a single precision standard, a floating-point number is distributed into three components, Sign, Exponent, and Mantissa. The Sign in single-precision standard is of a single bit, the exponent is 8-bit wide, and the mantissa is 23-bit wide. The exponent represented in the IEEE-754 single-precision standard is represented with a bias of 127.

PDK and Process Details The parameters chosen for APR can be seen in Table 6.1

PARAMETERS CHOSEN FOR APR		
Technology	Sky130n	
Operating Voltage	1.8V	
Number of Tracks	9	
Pitch of Metal 2	0.46um	
Height of Standard Cell	2.72um	
Liberty File	sky130_fd_sc_hd ss_100C_1v60.lib	
	sky130_fd_sc_hd ff_n40C_1v95.lib	
	sky130_fd_sc_hd tt_025C_1v80.lib	
	sky130_sram_1kbyte_1rw1r_32x256_8_TT_1p8V_25C.lib	
Process [Corner]	Slow	
	Fast	
	Typical	
Voltage	1.6V	
	1.95V	
	1.8V	
Temperature	100C	
	-40C	
	25C	
P.V.T For Hold	Fast | 1.95V | -40C Typical | 1.8V | 25C	
P.V.T For Setup	Slow | 1.6V | 100C Typical | 1.8V | 25C	
Number of Metal Layers	6 Metal Layers	
Targeted Frequency	76.9230 MHz	
Input delay: min | max	1.3ns | 6.5ns	
Output delay: min | max	1.3ns | 6.5ns	
Memory	1kb	
Base Model	1kb	
Number of base models	1	
Memory Size	X	Y
	479.78um	397.5um

Table 6.1: Parameters Chosen For APR Of Single Precision

Synthesis, Mapping and Static Timing Analysis (STA) The initial settings for synthesis are stated below in Table 6.2:

INITIAL SETTINGS	
Syn generic effort	High
Syn map effort	High
Syn opt effort	High
Cost group used	Yes
Physical synthesis applied	No
Multi corner flow used	Yes
Captable used	No
QRC used	No

Table 6.2: Initial Settings For Synthesis Of Single Precision

The results after synthesis are stated below in Table 6.3:

RESULT AFTER SYNTHESIS	
Total number of gates	25002
Number of Sequential instances	2532
Number of combinational instances	22470
Number of hierarchical instances	1
Cell Area	407965.913 μm^2
Max fanout	2533 (clk)
Min fanout	0
Average fanout	2.5
Terms to net ratio	3.5744
Terms to instance ratio	3.6149
Total number of paths [SETUP]	3000
Number of Failing paths [SETUP]	413
Number of passing paths [SETUP]	2587
Setup slack [WNS]	-17.1930ns
Setup slack [TNS]	-1155.4586ns

Table 6.3: Result After Synthesis Of Single Precision

Timing Histogram Of Setup The timing histogram of the setup can be seen in Figure 6.4.

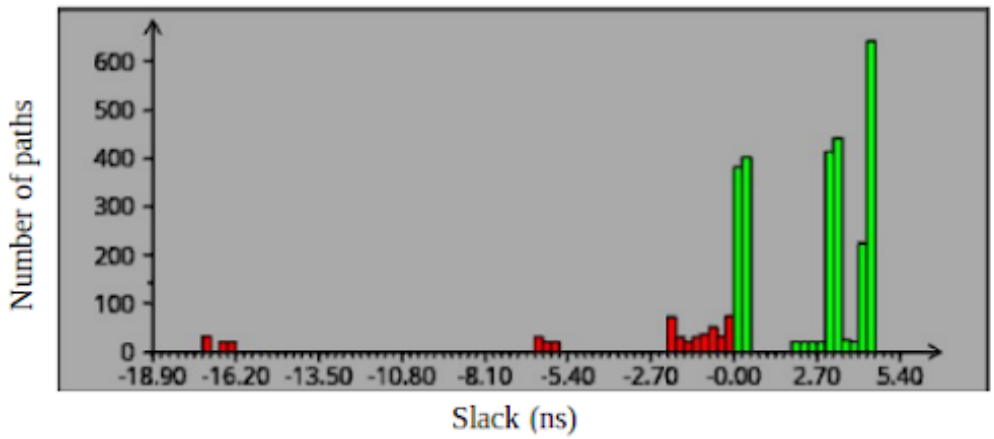

Figure 6.4: Timing Histogram Of Setup For Single Precision

Gate Details The gate area details can be observed from Table 6.4.

GATE COUNT WITH AREA PERCENTAGE			
TYPE	**INSTANCES**	**AREA μm^2**	**AREA %**
Timing Model	1	190712.550	46.7
Sequential	2531	62533.725	15.3
Inverter	3477	15867.718	3.9
Buffer	579	3911.251	1.0
Logic	18414	134940.699	33.1
Physical Cell	0	0.000	0.0
Total	25002	407965.913	100.0

Table 6.4: Gate Area Details Of Single Precision

Placement After placing standard cells

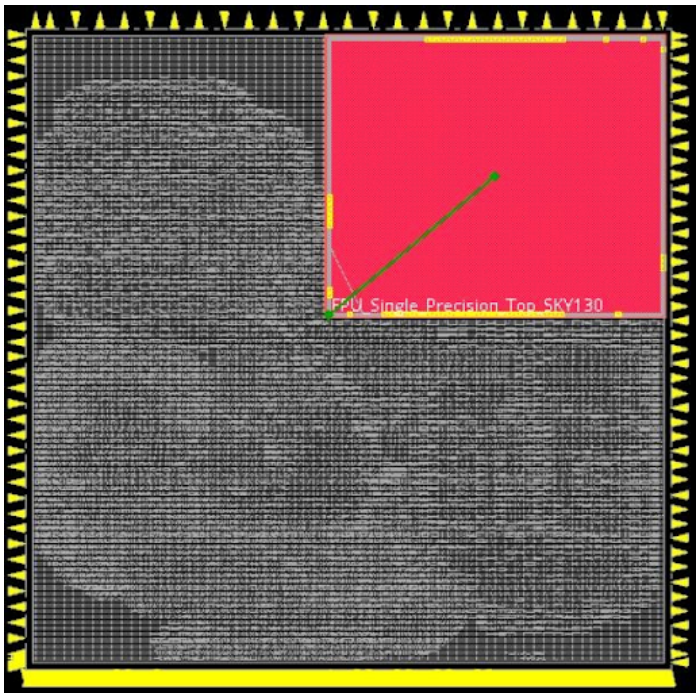

Figure 6.5: Placement Of Standard Cells For Single Precision

Final GDS The Final GDS is

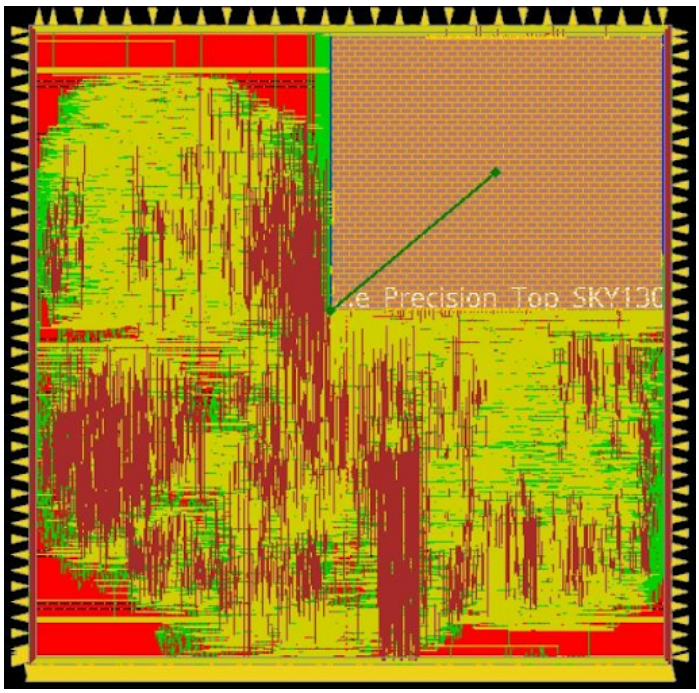

Figure 6.6: Final GDS Of Single Precision

Final Result Table The Final Results can be seen below in Table 6.5.

Technology	**Sky130n**
Operating Voltage	**1.8V**
Number of Tracks	9
Pitch of Metal 2	0.46um
Height of Standard Cell	2.72um
P.V.T For Hold	Fast | 1.95V | -40C Typical | 1.8V | 25C
P.V.T For Setup	Slow | 1.6V | 100C Typical | 1.8V |25C
Number of Metal Layers	6 Metal Layers
Targeted Frequency	76.9230 MHz
Achieved Frequency	18.4624 MHz
Input delay: min | max	1.3ns | 6.5ns
Output delay: min | max	1.3ns | 6.5ns
Memory	1kb
Base Model	1kb
Number of base models	1
Die Area	855533.024 μm^2
Number of DRCs	0
Number of LVS	Not Checked
Hold slack [WNS]	0.110ns
Hold slack [TNS]	0.000ns
Setup slack [WNS]	-41.1640ns
Setup slack [TNS]	-12122.2148ns

Table 6.5: Final Result Table Of Single Precision

Final Layout The Final Layout can be seen below in Figure 6.7.

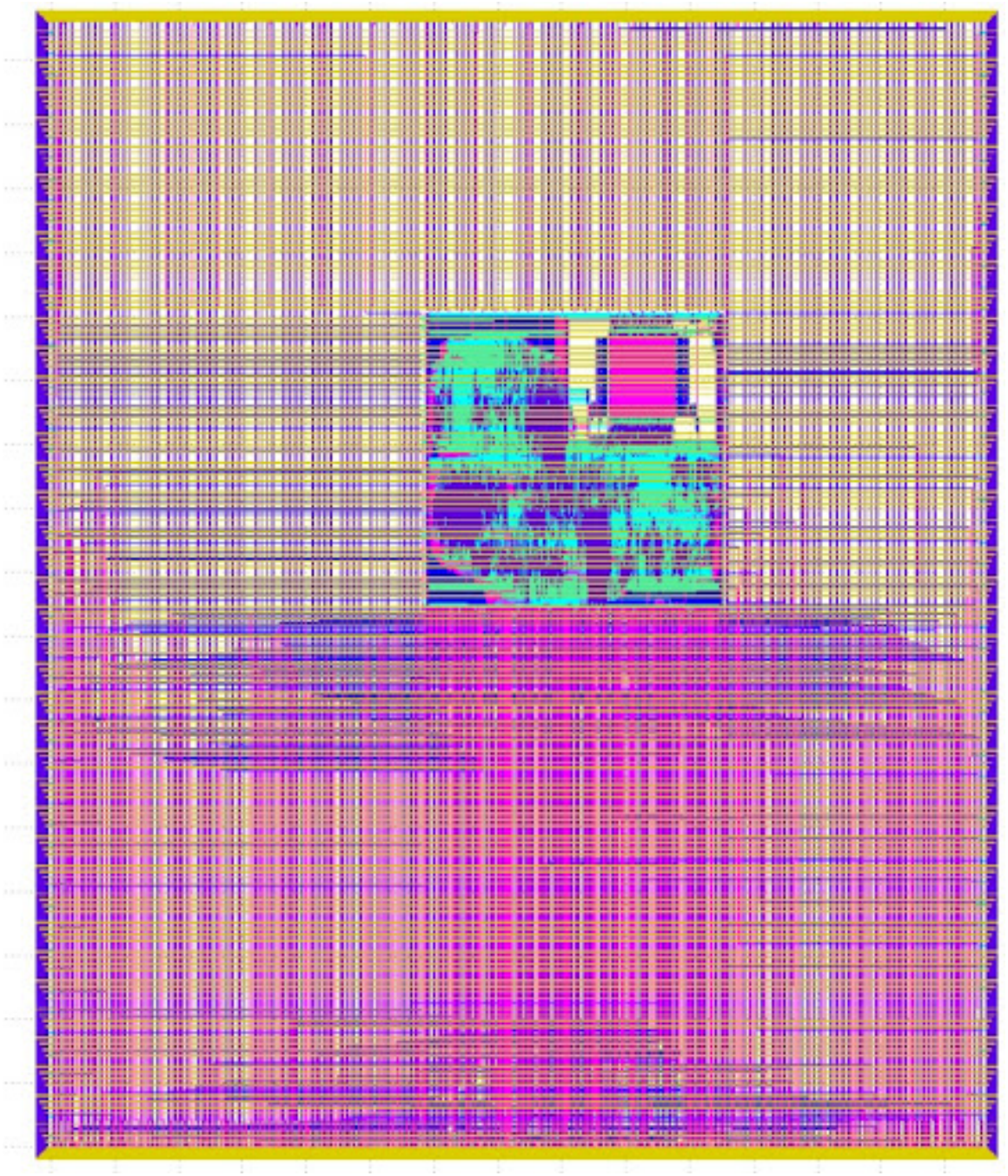

Figure 6.7: Final Layout Of Single Precision

6.3 Physical Design Of Half Precision (HP)

Half precision is one of the floating-point standards that have been defined in the IEEE-754 floating-point standard. Similar to single precision, in half precision standard, a floating-point number is also distributed into three components, Sign, Exponent, and Mantissa. The Sign in single-precision standard is of a single bit, the exponent is 5-bit wide, and the mantissa is 10-bit wide. The exponent represented in the IEEE-754 half-precision standard is represented with a bias of 15.

PDK and Process Details The parameters chosen for APR can be seen in Table 6.6

PARAMETERS CHOSEN FOR APR		
Technology	Sky130n	
Operating Voltage	1.8V	
Number of Tracks	9	
Pitch of Metal 2	0.46um	
Height of Standard Cell	2.72um	
Liberty File	sky130_fd_sc_hd ss_100C_1v60.lib	
	sky130_fd_sc_hd ff_n40C_1v95.lib	
	sky130_fd_sc_hd tt_025C_1v80.lib	
	sky130_sram_1kbyte_1rw1r_32x256_8_TT_1p8V_25C.lib	
Process [Corner]	Slow	
	Fast	
	Typical	
Voltage	1.6V	
	1.95V	
	1.8V	
Temperature	100C	
	-40C	
	25C	
P.V.T For Hold	Fast | 1.95V | -40C Typical | 1.8V | 25C	
P.V.T For Setup	Slow | 1.6V | 100C Typical | 1.8V | 25C	
Number of Metal Layers	6 Metal Layers	
Targeted Frequency	76.9230 MHz	
Input delay: min | max	1.3ns | 6.5ns	
Output delay: min | max	1.3ns | 6.5ns	
Memory	1kb	
Base Model	1kb	
Number of base models	1	
Memory Size	X	Y
	479.78um	397.5um

Table 6.6: Parameters Chosen For APR Of Half Precision

Synthesis, Mapping and Static Timing Analysis (STA) The initial settings for synthesis are stated below:

INITIAL SETTINGS	
Syn generic effort	**High**
Syn map effort	High
Syn opt effort	High
Cost group used	I2C C2O C2C
Physical synthesis applied	Generic
Multi corner flow used	Slow Fast Typical
Captable used	No
QRC used	No

Table 6.7: Initial Settings For Synthesis Of Half Precision

The results after synthesis are stated below in Table 6.8:

RESULT AFTER SYNTHESIS	
Total number of gates	**14226**
Number of Sequential instances	1944
Number of combinational instances	12282
Number of hierarchical instances	0
Cell Area	325281.708 μm^2
Max fanout	1945 (clk)
Min fanout	1 (FPU_Half_Precision_Top_Decoder_CSR_Read_Data_r1[0])
Average fanout	2.6
Terms to net ratio	3.6831
Terms to instance ratio	3.7488
Total number of paths [SETUP]	3000
Number of Failing paths [SETUP]	246
Number of passing paths [SETUP]	2754
Setup slack [WNS]	-12.5040ns
Setup slack [TNS]	-425.0613ns

Table 6.8: Result After Synthesis Of Half Precision

Timing Histogram of Setup The timing histogram of the setup can be seen in Figure 6.8.

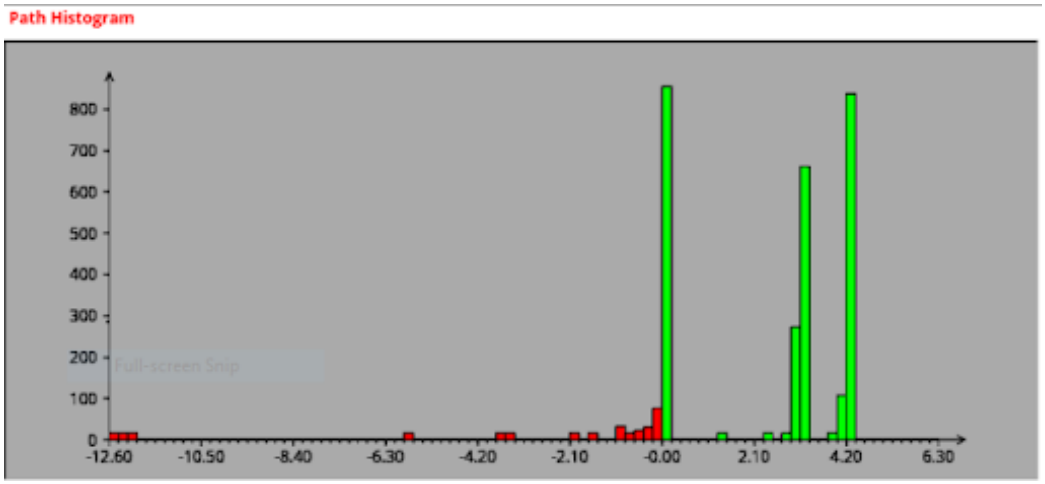

Figure 6.8: Timing Histogram Of Setup For Half Precision

Gate Details The gate area details can be observed from Table 6.9.

GATE COUNT WITH AREA PERCENTAGE			
TYPE	**INSTANCES**	**AREA** μm^2	**AREA %**
Timing Model	1	190712.550	58.6
Sequential	1943	48465.232	14.9
Inverter	1731	7946.371	2.4
Buffer	321	2324.730	0.7
Logic	10230	75932.826	23.3
Physical Cell	0	0.000	0.0
Total	14226	325381.708	100.0

Table 6.9: Gate Area Details Of Half Precision

Placement After placing standard cells

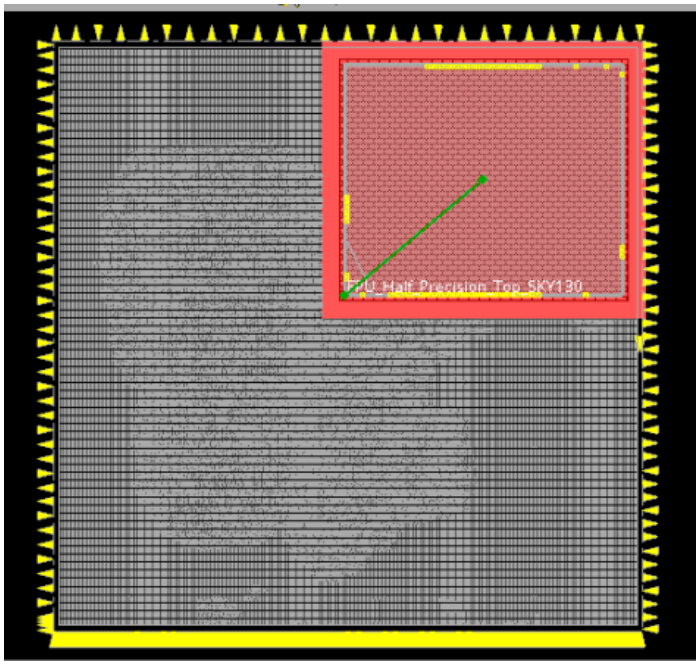

Figure 6.9: Placement Of Standard Cells For Half Precision

Final GDS The Final GDS is

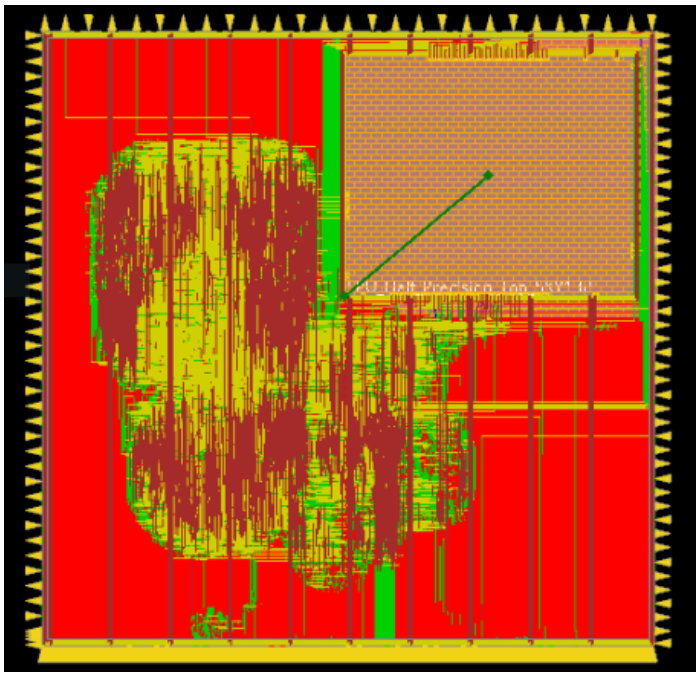

Figure 6.10: Final GDS Of Half Precision

Final Result Table The Final Results can be seen below in Table 6.10.

Technology	**Sky130n**
Operating Voltage	**1.8V**
Number of Tracks	9
Pitch of Metal 2	0.46um
Height of Standard Cell	2.72um
P.V.T For Hold	Fast | 1.95V | -40C Typical | 1.8V | 25C
P.V.T For Setup	Slow | 1.6V | 100C Typical | 1.8V | 25C
Number of Metal Layers	6 Metal Layers
Targeted Frequency	76.9230 MHz
Achieved Frequency	29.87 MHz
Input delay: min | max	1.3ns | 6.5ns
Output delay: min | max	1.3ns | 6.5ns
Memory	1kb
Base Model	1kb
Number of base models	Fast | 1.95V | -40c | Typical | 1.80V | 25c
Die Area	1,040,338,7048 um 2
Number of DRCs	0
Number of LVS	Not Checked
Hold slack [WNS]	0.00
Hold slack [TNS]	0.00ns
Setup slack [WNS]	-47.05
Setup slack [TNS]	-52119.48

Table 6.10: Final Result Table Of Half Precision

Final Layout The Final Layout can be seen below in Figure 6.11.

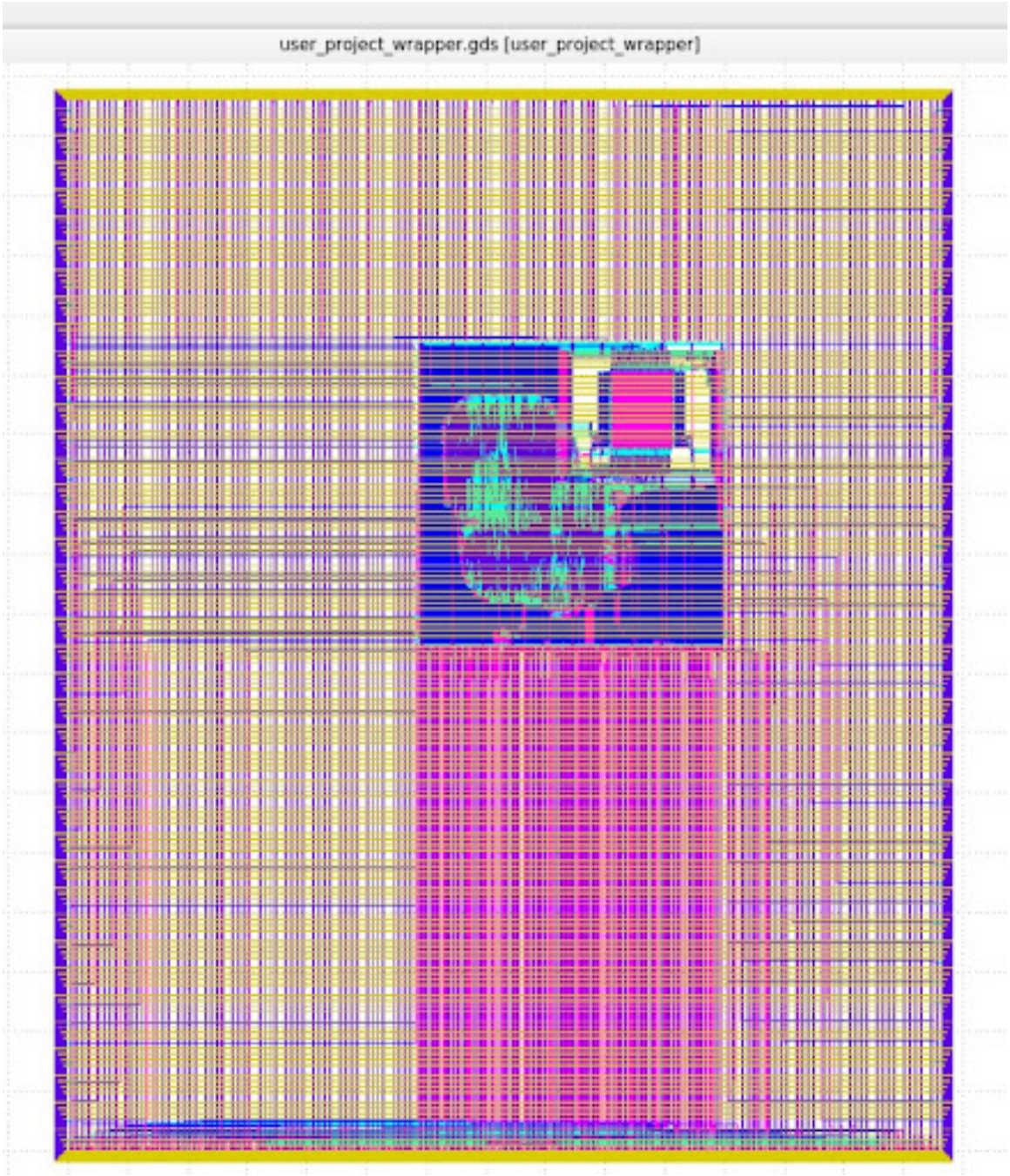

Figure 6.11: Final Layout Of Half Precision

6.4 Physical Design Of Bfloat-16 (BP)

Bfloat-16 is a standard developed by GOOGLE to speed up machine learning. The standard is similar to the IEEE-754 half precision in terms of bit but the distribution of sign, exponent, and mantissa is slightly different from the 16-bit standard. The Sign in Bfloat-16 standard is of a single bit, the exponent is 8-bit wide, and the mantissa is 7-bit wide.

PDK and Process Details The parameters chosen for APR can be seen in Table 6.11

PARAMETERS CHOSEN FOR APR		
Technology	**Sky130 nm**	
Operating Voltage	1.80	
Number of Tracks	9	
Pitch of Metal 2	0.46	
Height of Standard Cell	2.72	
Liberty File	sky130_fd_sc_hd__tt_025C_1v80.lib	
	sky130_fd_sc_hd__ss_100C_1v60.lib	
	sky130_fd_sc_hd__ff_n40C_1v95.lib	
	sky130_sram_1kbyte_1rw1r_32x256_8_TT_1p8V_25C.lib	
Process [Corner]	typical	
	slow	
	fast	
Voltage	1.80	
	1.60	
	1.95	
Temperature	25C	
	100C	
	-40C	
P.V.T For Hold	Fast | 1.95V | -40 C Typical | 1.80V | 25C	
P.V.T For Setup	Slow | 1.60V | 100 C Typical | 1.80V | 25C	
Number of Metal Layers	6	
Targeted Frequency	12ns 83.3MHZ	
Input delay: min | max	1.2ns | 6ns	
Output delay: min | max	1.2ns | 6ns	
Memory	1kb	
Base Model	1kb	
Number of base models	1	
Memory Size	X	Y
	32	256

Table 6.11: Parameters Chosen For APR Of Bfloat-16

Synthesis, Mapping and Static Timing Analysis (STA) The initial settings for synthesis are stated below:

INITIAL SETTINGS	
SYN_EFF	**high**
MAP_EFF high	high
OPT_EFF high	high
Multi Corner	yes
Physical synthesis	no
CAPTABLE	no
QRC	no
SYN_EFF	high

Table 6.12: Initial Settings For Synthesis Of Bfloat-16

The results after synthesis are stated below in Table 6.13:

RESULT AFTER SYNTHESIS	
Total number of gates	**13403**
Number of Sequential instances	1956
Number of combinational instances	11447
Number of hierarchical instances	0
Cell Area	321113.865
Max fanout	1957 (clk)
Min fanout	0 (FPU_Bfloat-16_Precision_Top_ Excecution_Unit_FPU_Execution_ Floating_Top_Oper and_Int[24])
Average fanout	2.6
Terms to net ratio	3.684
Terms to instance ratio	3.8133
Total number of paths [SETUP]	2000
Number of Failing paths [SETUP]	258
Number of passing paths [SETUP]	1742
Setup slack [WNS]	-11.8280
Setup slack [TNS]	-510.4608

Table 6.13: Result After Synthesis Of Bfloat-16

Timing Histogram Of Setup The timing histogram of the setup can be seen in Figure 6.12.

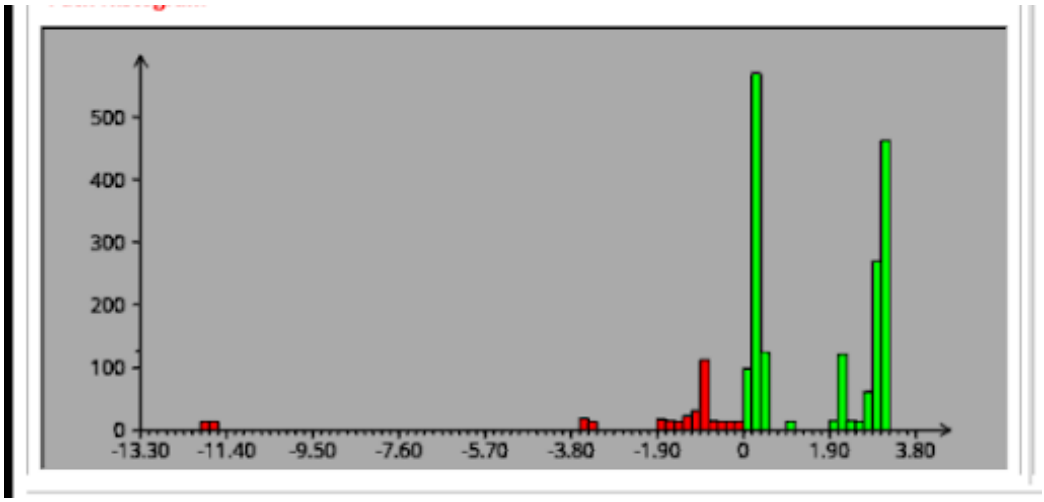

Figure 6.12: Timing Histogram Of Setup For Bfloat-16

Gate Details The gate area details can be observed from Table 6.14.

GATE COUNT WITH AREA PERCENTAGE			
TYPE	**INSTANCES**	**AREA μm^2**	**AREA %**
Timing_Model	1	190712.550	59.4
Sequential	1955	49303.536	15.4
Inverter	1374	6301.043	2.0
Buffer	264	1904.326	0.6
Logic	9809	72892.410	22.7
Physical_Cell	0	0	0
Total	13403	321113.865	100.0

Table 6.14: Gate Area Details Of Bfloat-16

Placement After placing standard cells

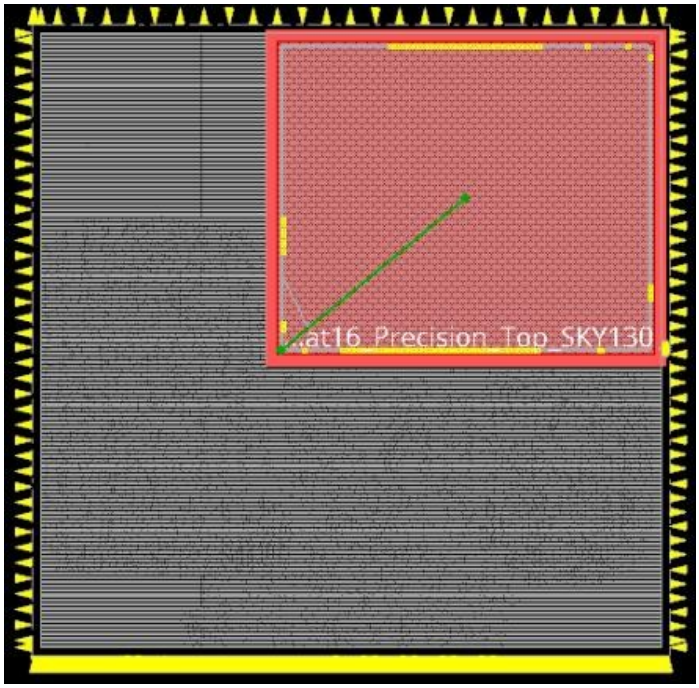

Figure 6.13: Placement Of Standard Cells For Bfloat-16

Final GDS The Final GDS is

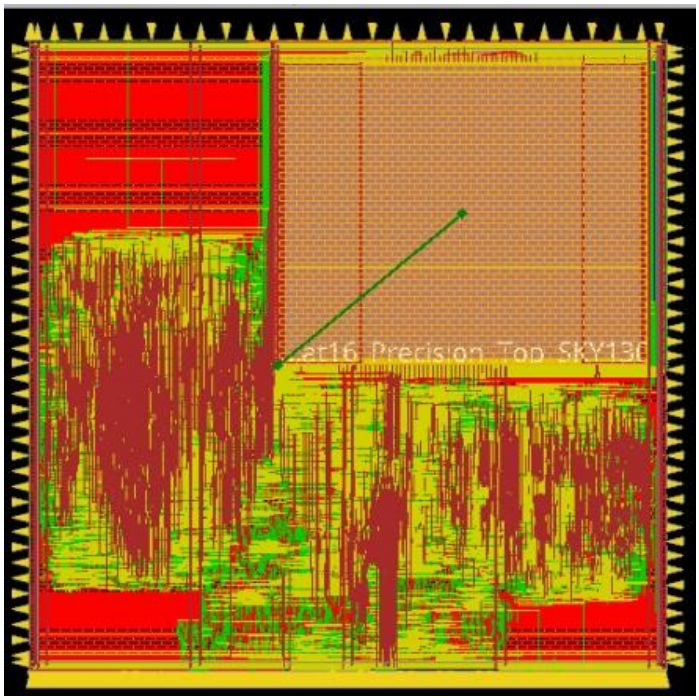

Figure 6.14: Final GDS Of Bfloat-16

Final Result Table The Final Results can be seen below in Table 6.15.

Technology	**130**
Operating Voltage	**1.80**
Number of Tracks	9
Pitch of Metal 2	0.46
Height of Standard Cell	2.72
P.V.T For Hold	Fast | 1.95V | -40 C | Typical | 1.80V | 25C
P.V.T For Setup	Slow | 1.60V | 100 C | Typical | 1.80V | 25C
Number of Metal Layers	6
Targeted Frequency	83.3MHZ
Input delay: min | max	1.2ns | 6ns
Output delay: min | max	1.2ns | 6ns
Memory	1kb
Base Model	1kb
Number of base models	1
Die Area	680157.95mm^2
Number of DRCs	0
Number of LVS	Not checked
Hold slack [WNS]	0
Hold slack [TNS]	0
Setup slack [WNS]	-25.2930
Setup slack [TNS]	-11528.7363
Technology	130

Table 6.15: Final Result Table Of Bfloat-16

Final Layout The Final Layout can be seen below in Figure 6.15.

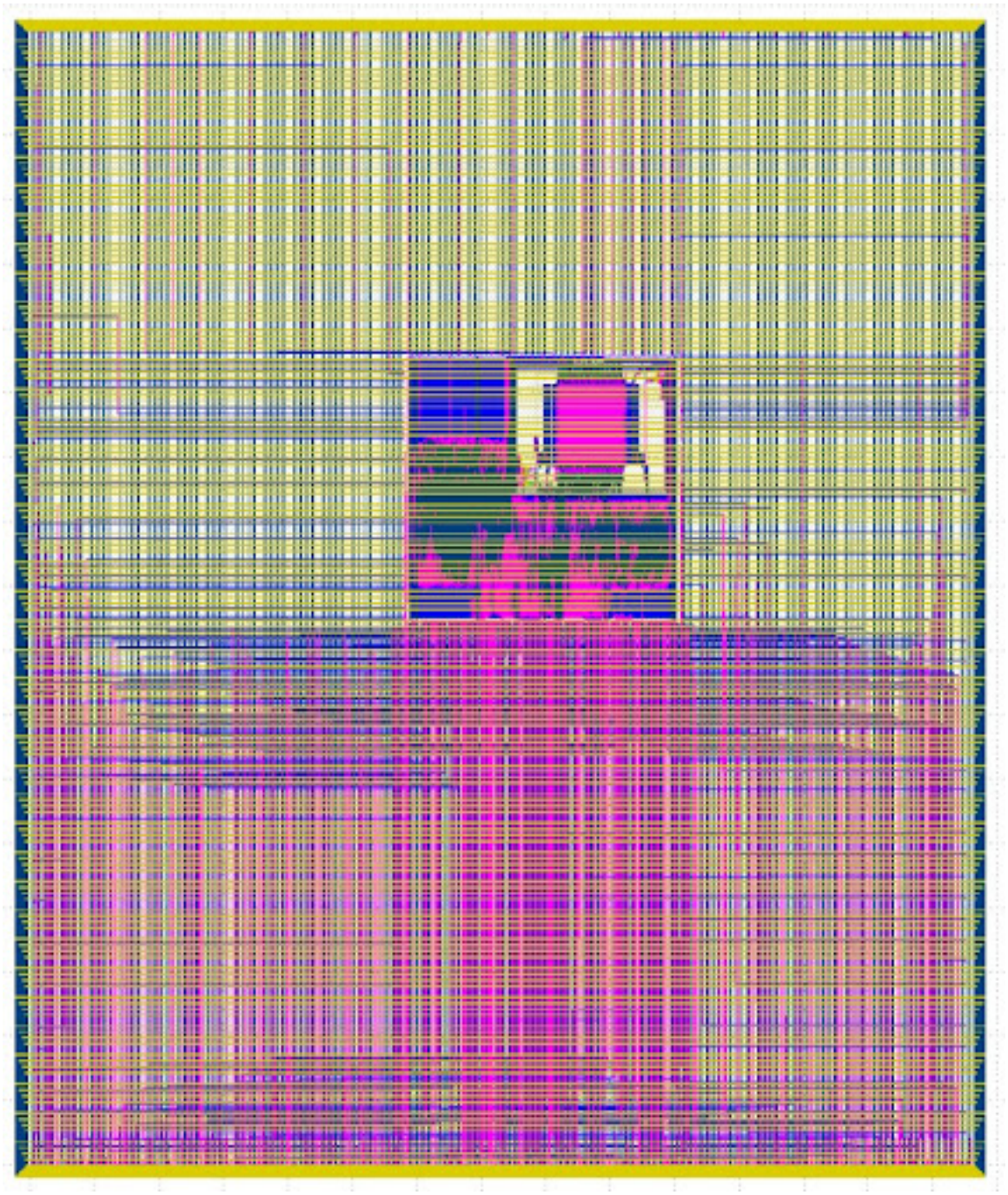

Figure 6.15: Final Layout Of Bfloat-16

7 Conclusion

The report covers all the design aspects of version 1 and version 2 of the SAP-FPU, which is the first ever designed and developed FPU in Pakistan. In SAP-FPU we have implemented 22 RISC-V instructions. All of the instructions are implemented as single cycle instructions. To functionally verify SAP-FPU, we have developed a Close Loop Testing (CLT) environment. Through the CLT environment millions of randomly generated numbers, op-codes, and rounding modes can be applied to the design. Two versions of the Close loop Testing environment were developed. In the second version, we used PULP FPU as the golden model, against which SAP-FPU results were compared. SAP-FPU passed all the tests run on it through the CLT environment. Results generated by SAP-FPU matched all the results generated by PULP FPU ISS. Along with functional verification of SAP-FPU through CLT, SAP-FPU was also tested through hardware emulation. To perform hardware emulation SAP-FPU was integrated with a core and the entire core was tested on FPGA. SAP-FPU also passed all the tests performed on it during hardware emulation.

References

[1] V. Rajaraman, "Ieee standard for floating point numbers," *Resonance*, vol. 21, no. 1, pp. 11–30, 2016.

[2] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, pp. 124–128, 1994.

[3] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo *et al.*, "Ieee standard for floating-point arithmetic," *IEEE Std*, vol. 754, no. 2008, pp. 1–70, 2008.

[4] C. D. Sa, "Low-precision machine learning," *CS4787 - Principles of Large-Scale Machine Learning Systems, Lecture 23*, February, Spring 2019.