Floating Point Unit Design, Verification and Implementation

A Graduation Project Report Submitted to
the Faculty of Engineering at Cairo University
in Partial Fulfillment of the Requirements for the
Degree of Bachelor of Science
in
Electronics and Communications Engineering

By

Dina Mamdouh Mohamed
Sara Nassef Amin
Kirolos Osama Sobhy
Kerollos Samy Aweda
Lidia Adel Adly
Momen Hussein Mohamed

Under supervision of

Dr. Hassan Mostafa

Faculty of Engineering, Cairo University

Giza, Egypt

July 2021

A B S T R A C T

This document describes briefly the flow of design, verification and realization of floating point unit (FPU) that performs the following operations: addition, subtraction and multiplication on two 32-bit representations (single precision and decimal representation decimal format). SIMD instruction is also supported for any of the supported operations with a maximum number of 16 similar operations per SIMD instruction

The project was distributed among four teams

- Design team
  - Developing an algorithm to perform the required operations on the specified representations using MATLAB
  - Implement the algorithm using synthesizable RTL code
  - Perform behavioural simulation, synthesis and post synthesis simulation
  - Improving the design working frequency using pipelining
- System team
  - Developing the host controller interface (HCI) specifications and writing the RTL code of the top level design
  - Building PULPino platform and adding the FPU as a peripheral to it
  - Developing an application and testing it using C coding
- Verification team
  - Building UVM testing environments to perform functional verification on each combinational module separately
  - Building UVM testing environment to test the clocked integrated modules with the host controller interface (HCI)
- Physical design team
  - Synthesize RTL with DC, generate gate level netlist, check STA, and formal verification
  - Place and route generated gate level netlist
  - Generating layout for FPU with speed 35.7 MHz
  - Generating layout with area equal 154122.5275 μm^2

Contents

List of Figures

List of Tables

CHAPTER ONE: INTRODUCTION

A floating-point unit (FPU) is a part of a computer system specially designed to carry out operations on floating-point numbers. Typical operations are addition, subtraction, multiplication, division, and square root.

The advantage of floating-point representation over fixed- point representation is that it can support a much wider dynamic range (the largest and smallest numbers that can be represented). The floating-point format needs slightly more storage (to encode the position of the radix point), floating-point numbers achieve their greater range at the expense of slightly less precision. Floating Point numbers has more flexibility than Fixed-point numbers which has limited or no flexibility. The internal representations of data in floating-point hardware are more exact than in fixed-point, ensuring greater accuracy in the results.

It is also important to consider fixed and floating-point formats in the context of precision – the size of the gaps between numbers. Every time a Digital signal processor (DSP) generates a new number via a mathematical calculation, that number must be rounded to the nearest value that can be stored via the format in use. Rounding and/or truncating numbers during signal processing naturally yields quantization error or 'noise' - the deviation between actual analog values and quantized digital values. Since the gaps between adjacent numbers can be much larger with fixed-point processing when compared to floating-point processing, round-off error can be much more pronounced. As such, floating-point processing yields much greater precision than fixed-point processing, distinguishing floating-point processors as the ideal DSP when computational accuracy is a critical requirement.

The applications of using the floating-point format can be readily seen by contrasting the data set requirements of video and audio applications. Floating Point units are used in high speed objects recognition system and also in high performance computer systems as well as embedded systems and mobile applications. In medical image recognition, greater accuracy supports the many levels of signal input from light, x-rays, ultrasound and other sources that must be defined and processed to create output images with useful diagnostic information. By contrast with these applications, the enormous communications market is better served by floating-point devices. FPUs execute dedicated trigonometric calculations used extensively in real-time applications such as motor control, power management, and communications data management. The graphics processing units (GPUs) today perform most arithmetic operations in the programmable processor cores using IEEE 754-compatible single precision 32-bit floating-point operations, newer GPUs such as the Tesla T10P also support IEEE 754 64-bit double-precision operations in hardware.

The designed floating-point unit (FPU) supports two representations of floating-point numbers according to IEEE754-2019 standard which are binary32 and decimal representation-decimal format, the following arithmetic operations are supported for each of the two representations, addition, subtraction and multiplication between operand A and operand B. SIMD instruction is also supported for any of the supported operations with a maximum number of 16 similar operations per SIMD instruction.

CHAPTER TWO: FLOATING POINT REPRESENTATION

2.1 Floating point representations

Floating point format according to IEEE754 standard-2019 is a way of representing real numbers with a string of digits. It maps the infinite range of real number by a finite subset with limited precision. A floating point number can be characterized by the following:

- Sign: the polarity of the number, either positive (+), or negative (-).
- Radix: the base number for scaling, usually two (binary), or ten (decimal).
- Exponent range: the interval of the maximum and minimum power of the radix.
- Significand: also called Precision or Mantissa, it is a fixed number of significant digits in base format.

In general any floating point number is represented with the following equation:

$$(-\mathbf{1})^{sign} \times \boldsymbol{significand} \times \boldsymbol{radix}^{exponent}$$

2.1.1 Binary interchange format encoding

Each floating-point number has just one encoding in a binary interchange format. To make the encoding unique, the value of the significand m is maximized by decreasing e until either e = emin or m ≥ 1. After this process is done, if e =e min and 0<m<1, the floating-point number is subnormal. Subnormal numbers (and zero) are encoded with a reserved biased exponent value. Representations of floating-point data in the binary interchange formats are uniquely encoded in k bits in the following three fields ordered (1):

a) 1-bit sign S.

b) w-bit biased exponent, E = e + bias.

c) (t = p − 1)-bit trailing significand field digit string, T = d_1 d_2…d_{p-1}; the leading bit of the significand.

d_0, is implicitly encoded in the biased exponent E.

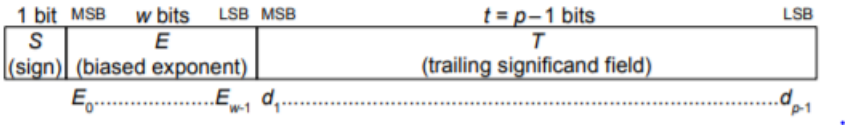

Figure 1: Binary format

To put a number in one of the binary representation the number must be transformed to binary so it can be written as for example:

$$111001 \rightarrow 1.11001 * 2^5$$

So e = 5, m = 11001 and s = 0.

Comparison between the same representations with different number of bits:

Parameter	Binary16	Binary32	Binary64	Binary128
Storage width in bits k	16	32	64	128
Precision in bits p	11	24	53	113
Maximum exponent e_{max}	15	127	1023	16383
Bias E	15	127	1023	16383
Sign bit	1	1	1	1
Exponent width in bits w	5	8	11	15
Significant field width	10	23	52	64

Table 1: comparison between different precisions in Binary representation

2.1.2 Decimal interchange representation:

2.1.2.1 The Need for Decimal Floating Point Arithmetic:

Although binary based computers dominate the world, decimal computations can't be ignored. Decimal numeration system is essential for many applications. Databases belong to 51 commercial and financial organizations were surveyed and investigated, these databases include many financial applications such as banking, billing, inventory control, financial analysis, taxes, and retail sales. There were more than 456,420 columns which contained numeric data and were investigated to extract statistic information. This survey reported that 55% were decimal, and that further 43.7% were integer types which could have been stored as decimal numbers. The results of these applications are required to be accurate and rounded correctly to be committed by human manual calculations and law.

For binary based computers, decimal numbers will be converted to/from binary numbers. Decimal numbers may not be converted exactly, due to the lack of binary system accuracy and finite precision hardware. Most of fraction numbers are not converted to binary numbers properly, let the decimal number X, to convert this decimal number to binary number it will be X that requires infinite number of bits to be represented exactly in binary which is not available so this number will be approximated, the stored value will be X, so any operation using this number will produce inaccurate results although the arithmetic operation is correct. The decimal to/from binary conversion is implemented using software programs with high delays.

In addition to the accuracy problem there is another problem caused by binary arithmetic is the removal of trailing fraction zeroes. For example, binary system can't distinguish between 1.5 and 1.50 because of the normalization nature of binary system. The trailing fraction zeros are essential in the calculation, they are very important for physics measurement for example if it is reported that, the mass of a body is 10.7kg versus 10.700kg, the two measures are not the same as the first one is accurate for 0.1 kg but the second one is accurate for 0.001kg. Hence binary arithmetic units can't be used directly for financial application and decimal arithmetic operations as they produce results not compatible with law and human requirements (2).

2.1.2.2 Decimal interchange format:

Representations of floating-point data in the decimal interchange formats are encoded in k bits in the following three fields, whose detailed layouts and canonical (preferred) encodings are described below.

a) 1-bit sign S.

b) A w+5 bit combination field G encoding classification and, if the encoded datum is a finite number, the exponent q and four significand bits (1 or 3 of which are implied). The biased exponent E is a w+2 bit quantity q+bias, where the value of the first two bits of the biased exponent taken together is either 0, 1, or 2.

c) A t-bit trailing significand field T that contains J ×10 bits and contains the bulk of the significand. When this field is combined with the leading significand bits from the combination field, the format encodes a total of p = 3×J+1 decimal digits (1).

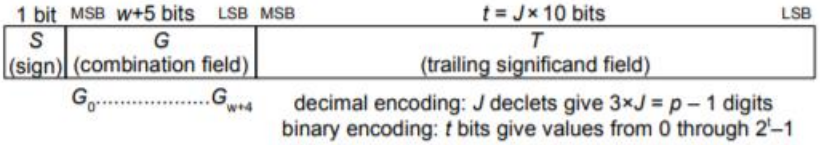

Figure 2: Decimal format

Decimal interchange format contains two ways of encoding they are decimal encoding and binary encoding.

2.1.2.2.1 Decimal interchange decimal encoding

The encoding of the combination field is done using the following table depending on the most significant bit in the mantissa.

COMBINATION FIELD

M4	M3	M2	M1	M0	First 2 bits of E	MSD of the mantissa	Range of the MSD
0	0	a	b	C	00	0abc	MSD <= 7
0	1	a	b	C	01	0abc	MSD <= 7
1	0	a	b	C	10	0abc	MSD <= 7
1	1	0	0	C	00	100c	8 <= MSD <= 9
1	1	0	1	C	01	100c	8 <= MSD <= 9
1	1	1	0	C	10	100c	8 <= MSD <= 9

Table 2: encoding of the combinational field

- If number is infinity, then M4 M3 M2 M1 M0 = 11110
- If the input is NAN, then M4 M3 M2 M1 M0 = 11111

Trailing significand field

Each three digit in the mantissa are encoded as in the following table:

b9	b8	b7	b6	b5	b4	b3	b2	b1	b0	d2	d1	d0	Values encoded
a	b	c	d	e	f	0	g	H	I	0abc	0def	0ghi	(0-7)(0-7)(0-7)
a	b	c	d	e	f	1	0	0	I	0abc	0def	100i	(0-7)(0-7)(8-9)
a	b	c	g	h	f	1	0	L	I	0abc	100f	0ghi	(0-7)(8-9)(0-7)
g	h	c	d	e	f	1	1	0	I	100c	0def	0ghi	(8-9)(0-7)(0-7)
g	h	c	0	0	f	1	1	1	I	100c	100f	0ghi	(8-9)(8-9)(0-7)
d	e	c	0	1	f	1	1	1	I	100c	0def	100i	(8-9)(0-7)(8-9)
a	b	c	1	0	f	1	1	1	I	0abc	100f	100i	(0-7)(8-9)(8-9)
x	x	c	1	1	f	1	1	1	I	100c	100f	100i	(8-9)(8-9)(8-9)

Table 3: encoding of the trailing field

Comparison between decimal representation decimal formats with different number of bits:

Parameter	Binary32	Binary64	Binary128
Storage width in bits k	32	64	128
Precision in digits p	7	16	34
Maximum exponent emax	96	384	6144
Minimum exponent emin	-95	-383	-6143
Bias E	101	398	6176
Sign bit	1	1	1
Combination field in bits	5	5	5
Exponent continuation field in bits	6	8	12
Trailing significand field in bits	20	50	110

Table 4: comparison between different precisions in Decimal representation-decimal format

To put a number in decimal 32 representation decimal format for example:
If the number =1324.25 * 10^5

$$1324.25 * 10^5 \rightarrow 132425.* 10^3$$

Then the most significant digit =1 will be included in the combination field as in table2
The exponent =3+bias=104→ 01101000
So the combination field and the exponent will be: 01-001-101000
The trailing field will be as in table3:011-010-0-100-010-101-0-000
S = 0
So the number will be written by putting the three parts together as:
0 01-001-101000 011-010-0-100-010-101-0-000

2.1.2.2.2 Decimal interchange binary encodings:

If the binary encoding is used for the significand, then:

- if G_0 G_1 is 00, 01, or 10, then E is made up of the bits G_0 to G_{w+1}, and the binary encoding of the significand C is obtained by prefixing the last 3 bits of G (i.e., G_{w+2} G_{w+3} G_{w+4}) to T.
- If G_0 G_1 is 11 and G_2 G_3 is 00, 01 or 10, then E is made up of the bits G_2 to G_{w+3}, and the binary encoding of the significand C is obtained by prefixing 100G_{w+4} to T (1).

For example if we have a number=$1.245678 * 10^{10}$
Then it can be written as $1245678.* 10^{4}$
So E = 4+101→0110 1001
Trailing field=1 0011 0000 0001 1110 1110
The length of trailing field = 21 which is less than 24 then this is the first case:
So the combination field will be = 01101001 001
The trailing field will be = 0011 0000 0001 1110 1110
The sign bit =0
So the number will be written as: 0 01101001-001 00110000000111101110

2.1.3 Special cases

For single precision representation

Single-Format Bit Pattern	Value
0<e<255	$(-1)^s * 2^{e-127} * 1.t$ (normal numbers)
e=0; t≠0 (at least one bit in t is non-zero)	$(-1)^s * 2^{e-127} * 0.t$ (subnormal numbers)
e=0; t=0 (all bits in t are zero)	$(-1)^s * 2^{e-127} * 0.0$ (signed zero)
s=0; e=255; t=0	+INF (positive infinity)
s=1; e=255; t=0	-INF (negative infinity)
s=x; e=255; t≠0 (at least one bit in t is non-zero)	NAN (Not-a-Number)

Table 5: Special cases in single precision

For decimal representation

decimal-Format Bit Pattern	Value
e=x, t=0	Signed zero
0_11110_00000……	+INF (positive infinity)
1_11110_00000……	-INF (negative infinity)
s_11111_00000……	QNAN
S_11111_10000…..	SNAN

Table 6: Special cases in Decimal representation

2.1.4 Exceptions

Exceptions are represented in RTL using flags to note that something abnormal happened to the resulted number

overflow	In case the result of addition or multiplication is bigger than the highest representable number.
underflow	In case the result of subtraction or division is lower than the smallest representable number.
inexact	In case rounding was done to the result or in case of underflow or overflow
Invalid operation	In case that the operation chosen can't be done on the inserted inputs.

Table 7: Exceptions

2.1.5 Rounding

Rounding process is very important after each operation, as all operations produce an intermediate result with infinite precision, so it is required to round this result to finite precision to be suitable for the destination precision format. IEEE 754-2019 standard defines five rounding modes for arithmetic operations as follow (1),

- **RoundTiesToEven:** the absolute result is rounded to nearest number. If tie case occurs the absolute result is rounded to nearest even value.
- **RoundTiesToAway:** the absolute result is rounded to nearest number. If tie case occurs the absolute result is rounded to the larger number.
- **RoundTiesToPositive:** the result is rounded towards positive infinity (if the final result sign is positive then the result is rounded up, else the extra digits are truncated).
- **RoundTiesToNegative:** the result is rounded towards negative infinity (if the final result sign is negative then the absolute result is rounded away from zero, else the extra digits are truncated).
- **RoundTowardZero:** the absolute result is rounded towards zero, (all extra digits are truncated).

CHAPTER THREE: RTL DESIGN

3.1 HIGH LEVEL DESIGN

The logic used in performing each operation on numbers represented in decimal-32 representation or on single precision has been simulated using MATLAB codes, then the results have been tested by comparing them with the results calculated using normal operations in MATLAB.

3.1.1 Decimal-32 representation – Decimal format

A function has been used to extract sign, exponent, and mantissa from the represented numbers to be able to perform the required operation, this function returns the most significand bit as sign bit then it decodes the exponent and the mantissa from the combinational field and trailing significand bits using Table 2 and Table 3.

3.1.1.1 Addition

To perform the addition operation, the two operands must have the same exponent, so first, significand alignment is done by comparing the two exponents to determine which operand has the largest one then calculating the difference between the two exponents and padding the mantissa of smallest operand from the left with a numbers of zeros equal to four multiplied by the calculated difference as the radix of the exponent is ten which means that if the difference equals to one will be equivalent to shifting the mantissa one digit (4 bits), therefore the exponent of the final result will be equal to the exponent of the largest operand.
Second, each 4 bits of the first operand starting from the right has been added to their corresponding in the second operand, if the result is greater than or equal to ten then subtract ten to obtain the corresponding 4 bits of the mantissa of the final results and a carry equals to one will be added to the addition of the next two digits; otherwise the result was putted directly in the mantissa.
Third, normalization is needed when the result of adding the 4 most significand bits is greater than ten; normalization has been done by incrementing the exponent by one and the mantissa will be equal to the most significand seven digits.
Finally, to check the correctness of this logic, different ranges of real numbers has been used, the result was calculated by using MATLAB, and by using the previous logic after putting the real numbers in decimal-32 representations.
As shown in Figure 3 below, the results are approximately equal in both cases.

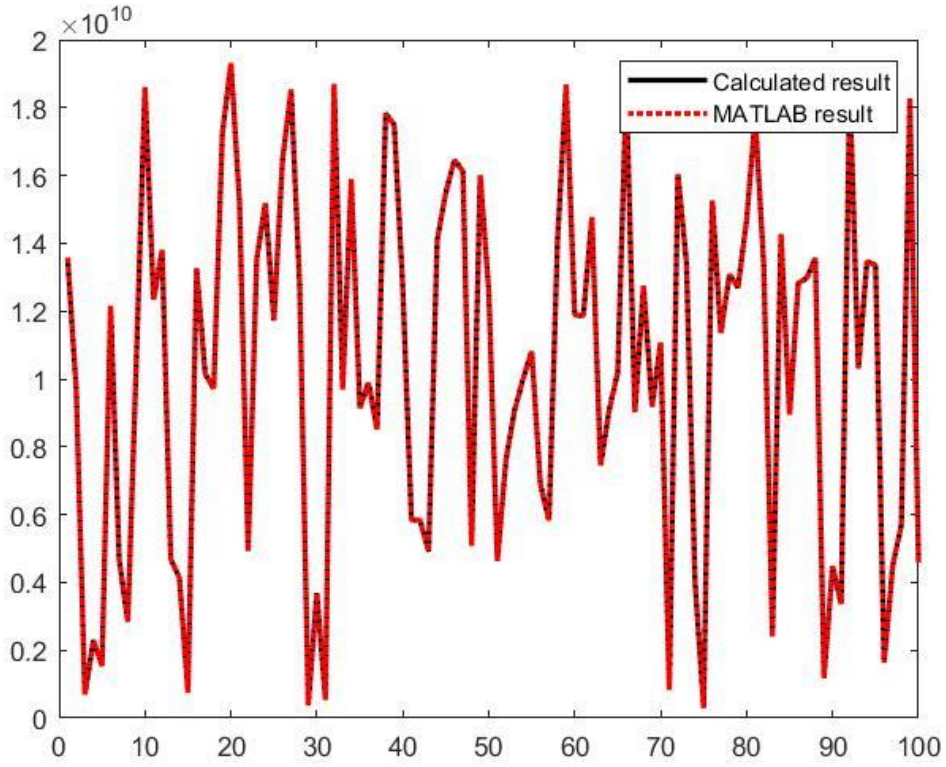

Figure 3: Addition MATLAB results – Decimal format

3.1.1.2 Subtraction

First step, deciding which exponent is the bigger to be the exponent of the final result, then, padding zeros to the mantissa of the number with the smaller exponent by the difference of the two exponents.
Second step, is deciding which mantissa is the bigger if there was no difference in the exponent because we will use the borrow method which needs to subtract the bigger from the smaller to get a right result, so if the second mantissa was the bigger then the final result is negative .
Third step, subtracting the bigger mantissa from the smaller one by taking the last 2 digits from each mantissa an subtract them if their result is negative then we need to add 10 to the result and borrow one from the digit that is before them and continue this operation on the seven digits.
Finally, the results as shown in Figure 4 have been checked as in addition.

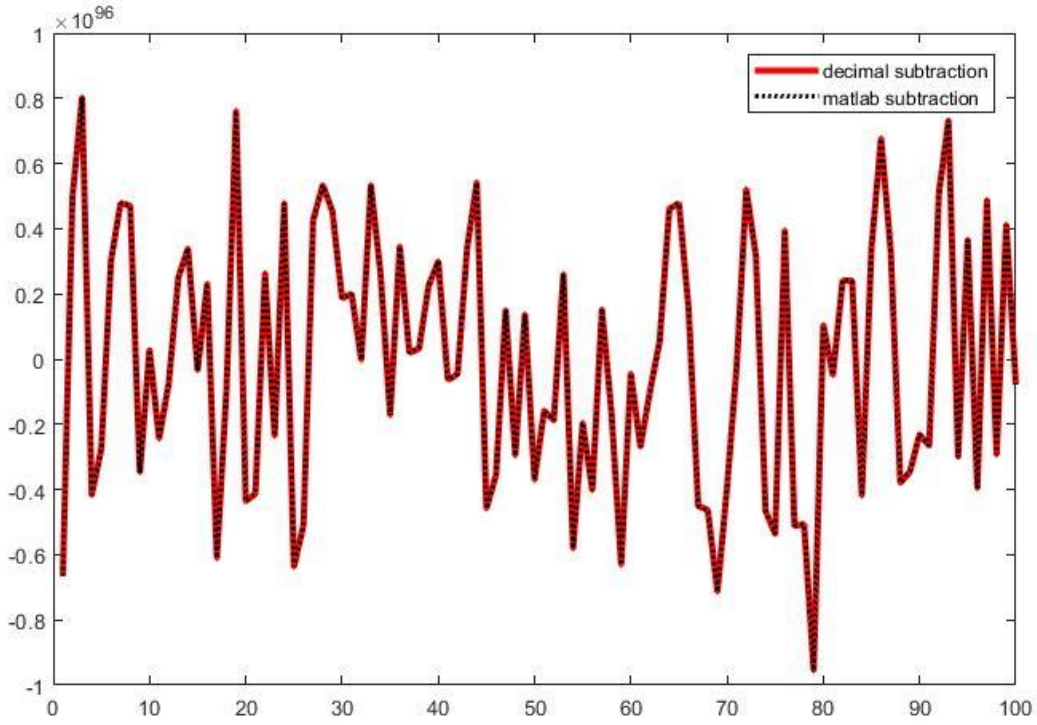

Figure 4: Subtraction MATLAB results – Decimal format

3.1.1.3 Multiplication

First, the exponent of the result has been calculated by adding the exponent of the two operands also, the sign bit has been calculated by xoring the sign bits of the two operands.
Second, the mantissa has been calculated as shown in Figure 5.
Third, the resulted mantissa will be equal to 14 digits, so normalization has been done by taking the most non-zero digits and then the number of remaining digits will be added to the exponent.
Finally, the results as shown in Figure 6 have been checked as in addition.

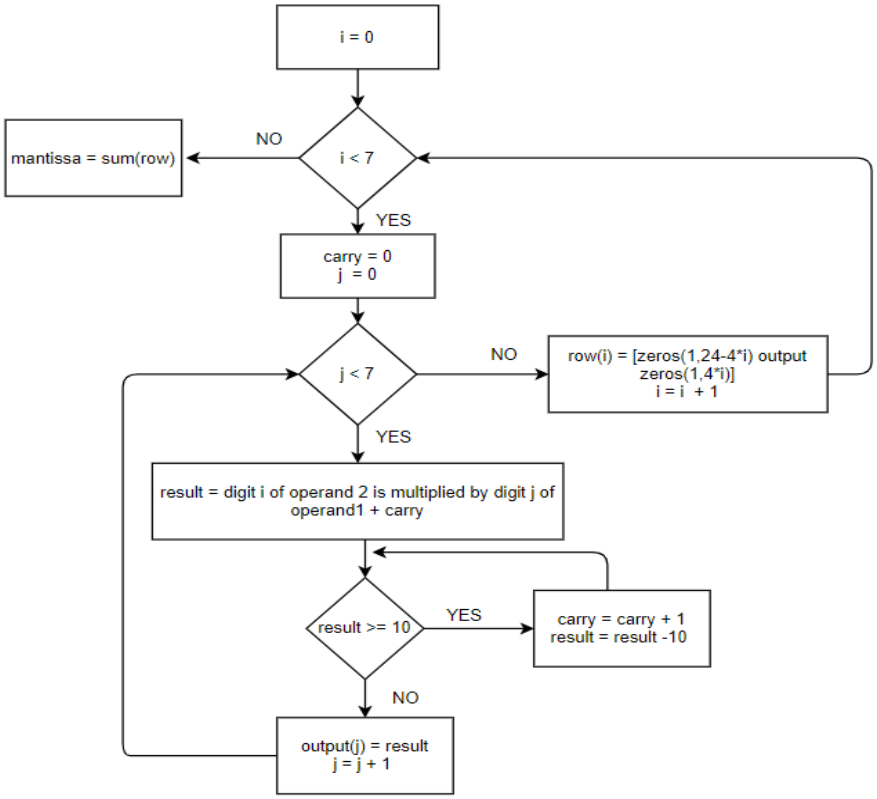

Figure 5: Mantissa calculation in MATLAB

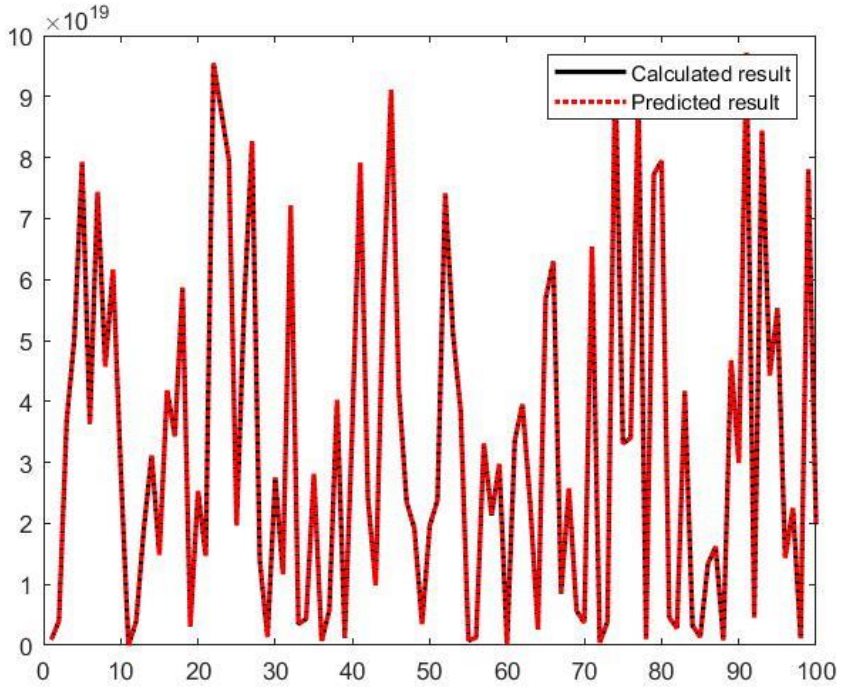

Figure 6: Multiplication MATLAB results – Decimal format

3.1.2 Decimal-32 representation – Binary format

A function has been used to extract sign, exponent and mantissa from the represented number the same as in decimal format.

3.1.2.1 Addition

Same as in decimal format first, significant alignment has been done by dividing the mantissa of the smallest operand by ten because the radix of the exponent is ten and the mantissa is represented in binary format which means two different bases; the number of divisions operation is equal to the difference between the two exponents.
Second, the mantissa of the two operands after alignment has been added directly as normal binary addition.
Finally, different ranges of real numbers have been used to check this logic, by comparing the calculated result with the predicated result as shown in Figure 7.

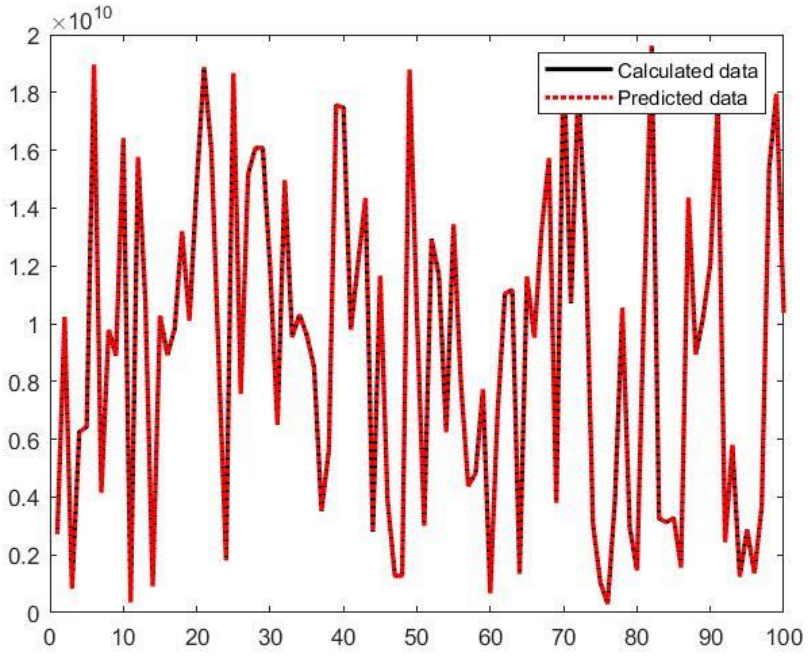

Figure 7: Addition MATLAB result - Binary format

3.1.2.2 Subtraction

Subtraction in binary format is very easy after making significant alignment, so as explained before in the addition the way to make significant alignment is using division.
Then after this step we can make a normal binary subtraction between the normalized mantissa of the two numbers.
Finally, the results as shown in Figure 8 have been checked as before.

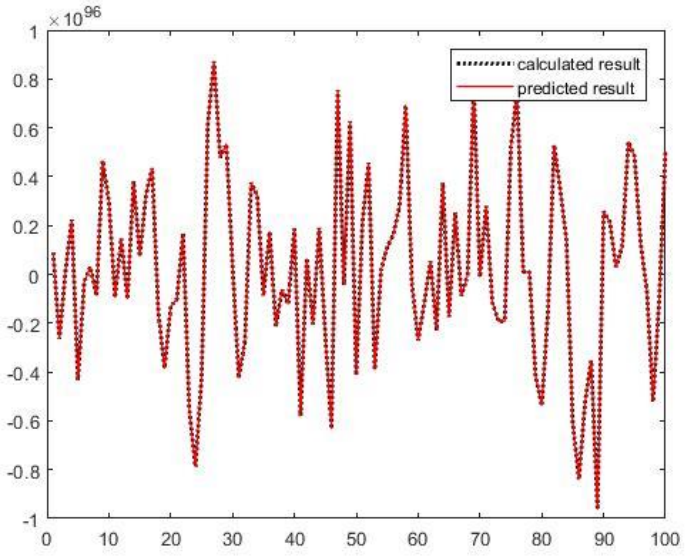

Figure 8: Subtraction MATLAB results - Binary format

3.1.2.3 Multiplication

First, the exponent has been calculated using binary addition of the exponent of the two operand and sign has been calculated same as in decimal format.
Second, the mantissa of the two operands have been multiplied together to calculate the resulted mantissa which will be equal to 48 bits so normalization is required.
To normalize the mantissa, it is required to decide how many digits are presented in the mantissa; this is done by comparing the mantissa with the largest number composed of 14 digits, if it is greater than this number, then add seven to the exponent and divide the mantissa by ten seven times, but if it is smallest than this number, then compare it with the largest number composed of 13 digits, , if it is greater, then add six to the exponent and divide the mantissa by ten six times, and if it is smallest complete the comparing process until the mantissa is normalized to be the most seven non-zeros digits and the numbers of remaining digits in the mantissa is added to the exponent.
Finally, the results as shown in Figure 9 have been checked as before.

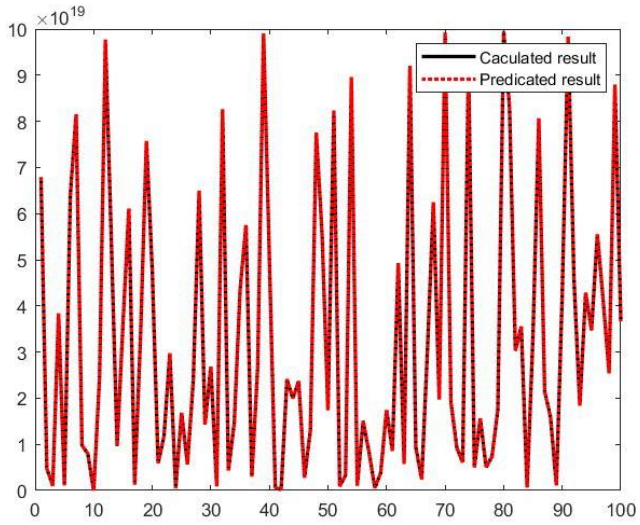

Figure 9: Multiplication MATLAB results – Binary format

3.1.3 Single precision

Sign, exponent and mantissa are extracted directly using bit selection in each operation.

3.1.3.1 Addition

First, significand alignment has been done by calculating the difference between the exponents of the two operands, then shifting the mantissa of the smallest operand with number of zeros equal to the difference calculated, and the exponent of the final result will be equal to the largest exponent.
Second, the mantissa of the two operands after the above modification have been binary added too each other, if there is a carry, then the exponent increased by one and the mantissa will be equal to the carry followed by the most 23 bits of the resulted mantissa.
Finally, different ranges of real numbers have been used to check this logic, by comparing the calculated result with the predicated result as shown in Figure 10.

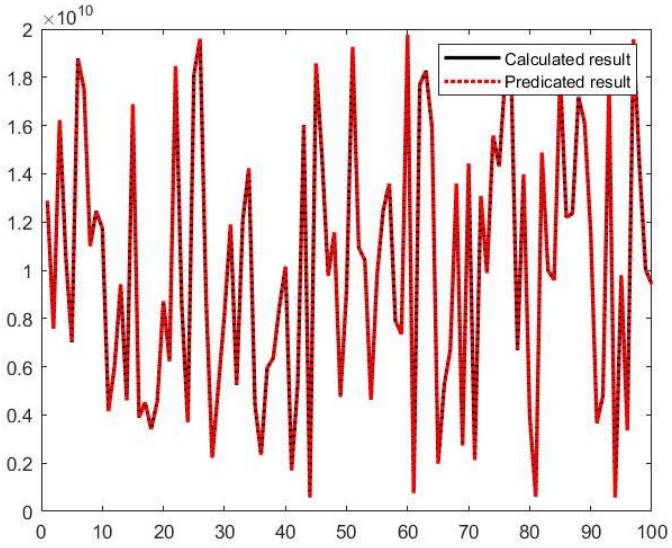

Figure 10: Addition MATLAB result - Single precision

3.1.3.2 Multiplication

First, to calculate the exponents of the result add the exponent of the two operands.
Second, binary multiply the mantissa of the two operands, then the resulted mantissa will be 48 bits, if the most significand bit equal to one then add one to exponent and take the most 24 bits of the resulted mantissa, otherwise normalize the mantissa by decrementing the exponent until reaching the first one.
Finally, the results as shown in Figure 11 have been checked as before.

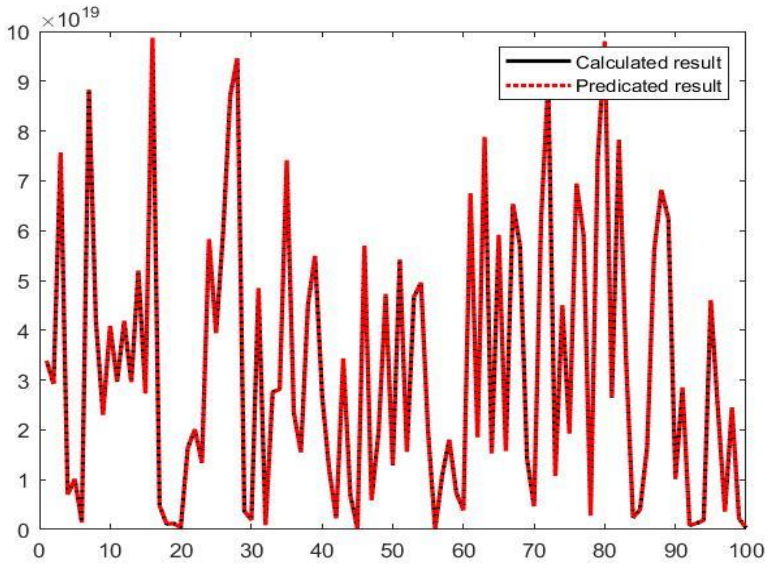

Figure 11: Multiplication MATLAB result - Single precision

3.2 REGISTER TRANSFER LEVEL (VERILOG)

This section describes how the high level design is translated to RTL code, in this project two representations only were chosen for the design phase they are decimal 32 representation decimal format and single precision representation.

3.2.1 Decimal-32 representation – Decimal format

3.2.1.1 Addition

Addition is done using the architecture in figure 12, each block has its role as described below:

1. Conversion from IEEE-754 to sign, exponent and mantissa:

   This block acts as a decoder that decodes the number to extract the sign bit, mantissa represented as a BCD number that is constructed of 28 bits bus each digit is represented in four bits and the exponent in an 8 bit bus.
2. Remove leading zeros:

   The function of this block is to remove the leading zeros in the entered in the representation to not to lose precision or digits in the steps of normalization of rounding, and this is done by checking the number of zeros in the mantissa, subtract this number from the exponent and remove these zeros from the mantissa.
3. Binary subtractor :

   This block is used to determine the exponent of the final result, calculate the difference between the two exponents to make significand alignment in the next block and send a signal called greater to indicate which mantissa needs significand alignment.
4. Significand alignment:

   This block pads the mantissa of the smallest number by zeros their number equals to the difference in the exponent but multiplied by four as each zero is represented in four bits the same as in MATLAB but it keeps the last three removed digits in an 12 bits bus as guard digit, round digit and sticky bit which is the ORing of all the removed digits from the significand alignment, the rounding digit will be used in the rounding module.
5. BCD adder:

   The BCD adder is constructed of 7-4bits binary adders and if the result of each adder is greater than nine then we add six to the result and take the least four bits in the final result and the carry is added in the next adder.
6. Rounding

   This module adds one to the mantissa if the rounding digit is greater than five.
7. Normalization:

   Normalization is used in case of a carry resulted from the addition which means that the result in composed of eight digits and that can't be represented so normalization module is used to take the most seven digits as a result and add one to the exponent of the final result. If the exponent of the final result exceeds 192=8'b1100-0000(the max exponent of the representation) overflow flag and inexact flag are raised.
8. Conversion to IEEE-754 standard

   This module takes the final exponent, the sign bit and the mantissa to encode them and put them in the final representation.

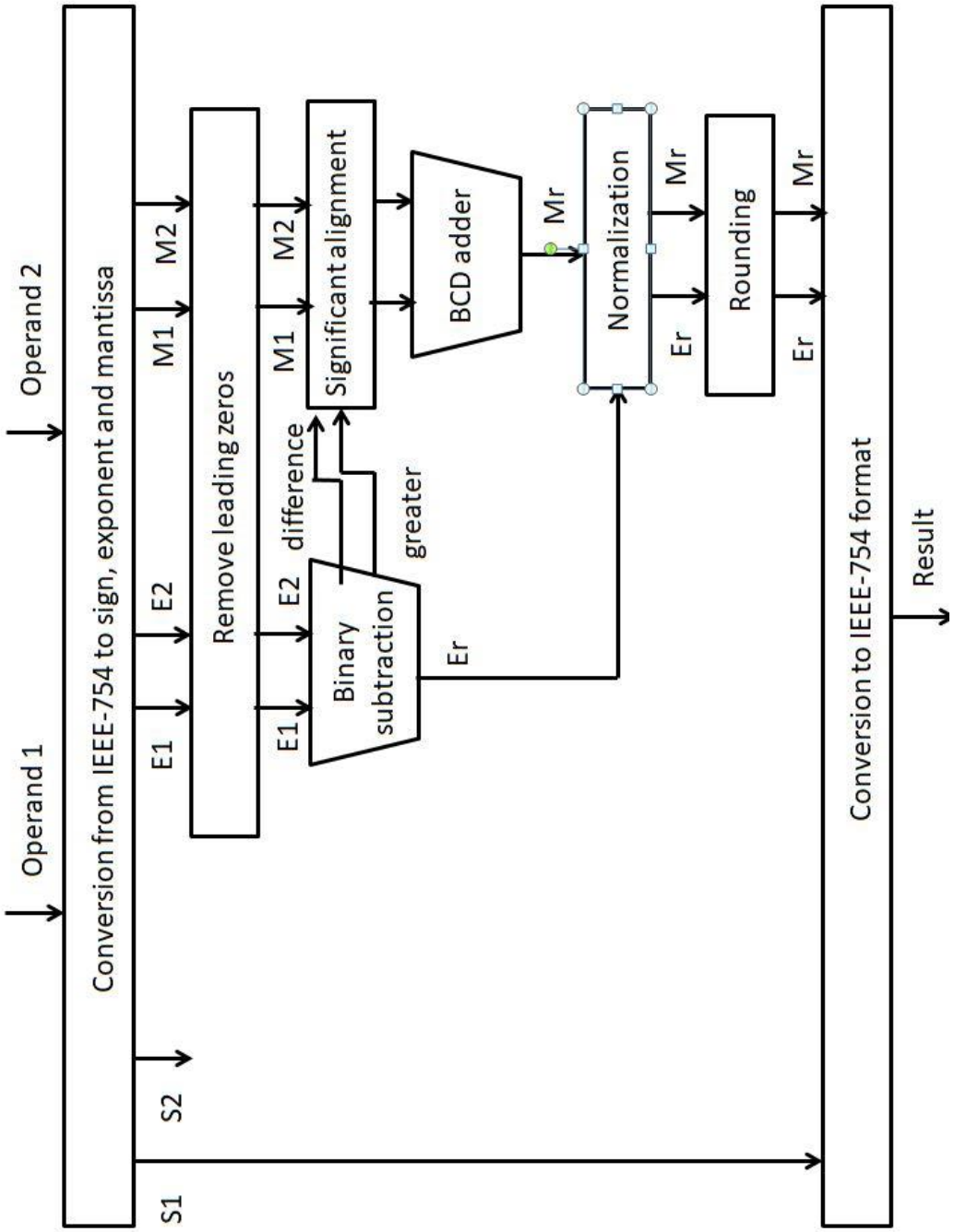

Figure 12: architecture of decimal adder

Behavioral simulation results:

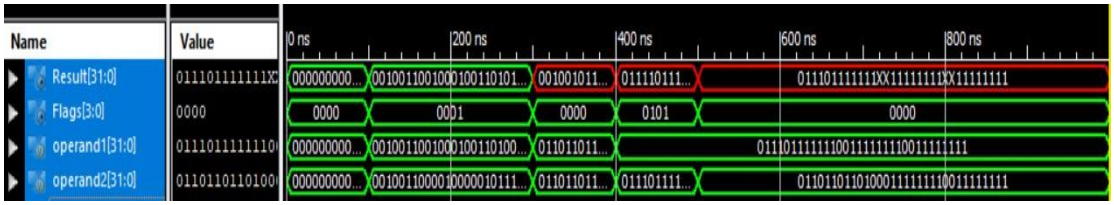

Figure 13: simulation result - decimal adder

Synthesis result:

```
Device utilization summary:
---------------------------

Selected Device : 4vfx12sf363-12

 Number of Slices:                      753  out of   5472    13%
 Number of Slice Flip Flops:             98  out of  10944     0%
 Number of 4 input LUTs:               1468  out of  10944    13%
 Number of IOs:                         103
 Number of bonded IOBs:                 102  out of    240    42%
 Number of GCLKs:                         1  out of     32     3%
```

Figure 14: synthesis result - decimal adder

Post place and route results:

```
Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   25.934|         |         |         |
---------------+---------+---------+---------+---------+
```

Figure 15: minimum clock allowed - decimal adder

3.2.1.2 Subtraction

Subtraction is done using the architecture in Figure 16; each block has its role as described below, repeated blocks are explained above:

1. Conversion form IEEE-754 standard to sign, exponent and mantissa.
   As described above in addition.
2. Remove leading zeros
   As described above in addition.
3. Binary subtractor.
   As described above in addition.
4. Significand alignment
   The same as in addition but the GRS digits are not extracted from the mantissa; they are a part of the mantissa because they will enter in the subtraction process so the output of significand alignment is two buses of fourteen bits.
5. BCD subtractor
   This block subtracts the mantissa of the two operands from each other using the ten's complement, after making the ten's complement of the second operand , the mantissa of the first operand and the ten's complement of the mantissa of the second operand enter in the BCD adder the same as in the addition but check the result of the BCD adder if there is a carry digit then the Result is positive and equal to the digits after the carry digit and if not then the result is negative and the result is the ten's complement of the result.
   The resulted mantissa is the first 28 bits only of the result as they represent seven digit and the next twelve bits are taken for the GRS digits.
6. Normalization
   This block removes the leading zeros resulted from the subtraction, enter the GRS digits instead of these zeros, add zeros to the right of the number to complete the seven digits and subtract the number of the leading zeros from the exponent of the result. if the exponent is less than the number of the leading zeros then remove zeros their number equals to the exponent .
   If the resulted mantissa and exponent equal to zero then underflow flag is raised.
7. Rounding
   As described above in addition.
8. Conversion to IEEE-754 format
   As described above in addition.

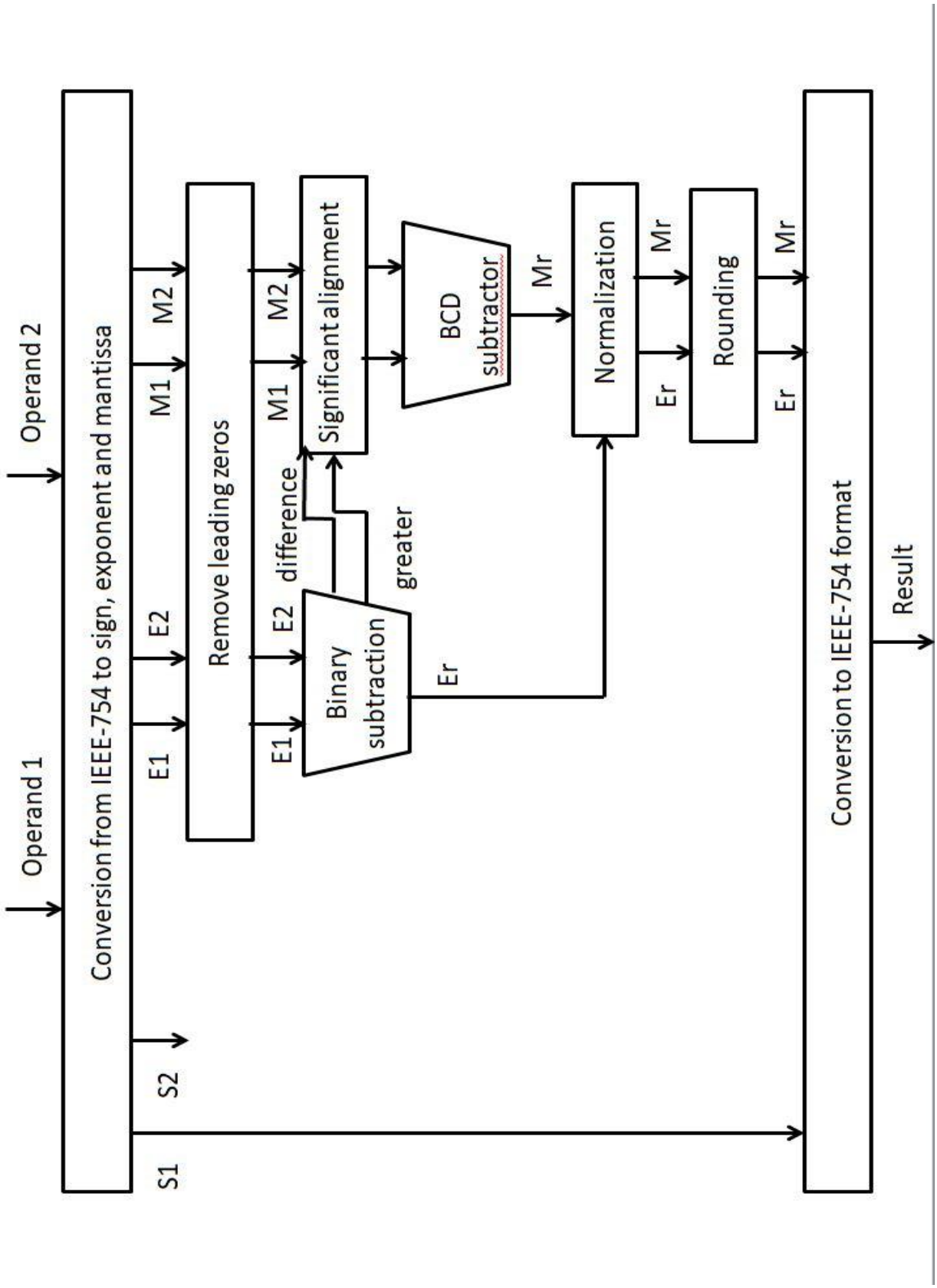

Figure 16: Architecture of subtraction in decimal format

Behavioral simulation results:

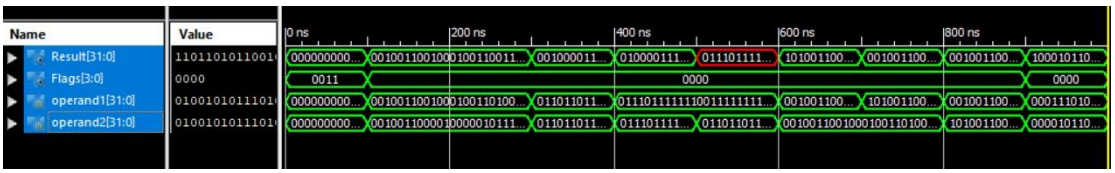

Figure 17: Behavioral simulation result - Decimal subtraction

Synthesis result:

```
Device utilization summary:
---------------------------

Selected Device : 4vfx12sf363-12

 Number of Slices:                     1016  out of   5472    18%
 Number of Slice Flip Flops:             98  out of  10944     0%
 Number of 4 input LUTs:               1980  out of  10944    18%
 Number of IOs:                         103
 Number of bonded IOBs:                 102  out of    240    42%
 Number of GCLKs:                         1  out of     32     3%
```

Figure 18: Area utilization - Decimal subttraction

Post place and route results:

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   34.994|         |         |         |
---------------+---------+---------+---------+---------+
```

Figure 19: Timing report - Decimal subtraction

3.2.1.3 Multiplication

Multiplication has been done using the architecture shown in Figure 20, each block has its role as described below

1. Conversion form IEEE-754 standard to sign, exponent and mantissa

   As described above in addition.
2. Xor gate

   This gate is used to determine the sign of the result.
3. Removing leading zeros

   As described above in addition.
4. Binary adder

   Add the exponent of the two operands, then subtract the bias to calculate the resulted exponent, also this block can raise underflow or overflow signal if the sum is greater or smaller than the available exponent.
5. BCD multiplier

   Multiplication process has been done as explained in the MATLAB work but instead of the "for" loop, use eight different cases to calculate the value of the carry in the next step.
6. Normalization

   Switch case has been used to determine the number of digits resulted from the multiplication operation, then take the first seven non-zeros digits, keep the following three digits to be used in rounding, and add the number of the remaining digits in the exponent, also check the underflow cases as sometimes although the addition of the two exponent is less than available exponent after the normalization the underflow flag can be lowered.
7. Rounding

   If the round digit is greater than five than add one to the mantissa and raise the inexact flag, then check if overflow has occurred.
8. Conversion to IEEE-754 format

   As described above in addition.

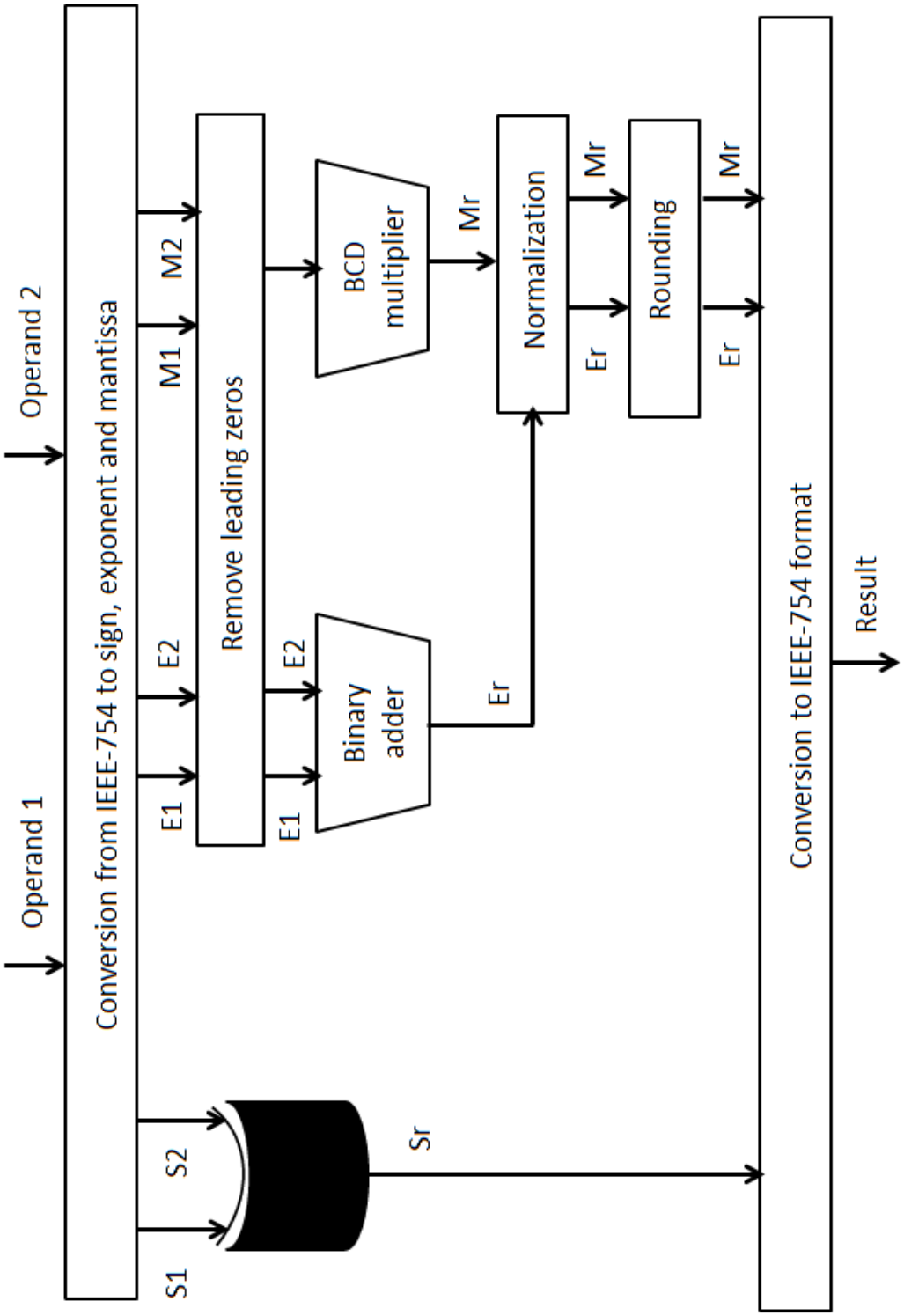

Figure 20: Architecture of Multiplication in decimal representation - Decimal format

Behavioral simulation results:

Figure 21 shows an underflow case, a normal case, an overflow case, an invalid operation case when one of the two operands equal infinity.

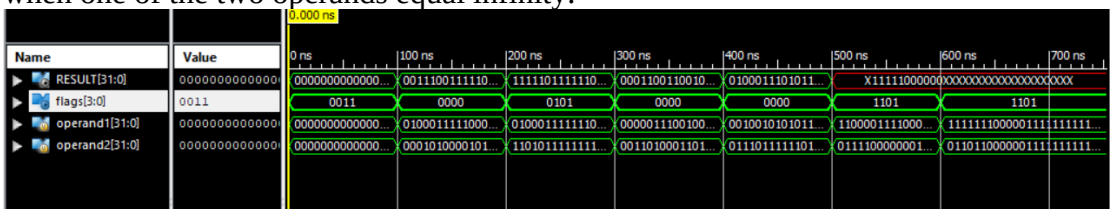

Figure 21: Behavioral simulation result - Decimal multiplication

Synthesis result:

```
Device utilization summary:
---------------------------

Selected Device : 4vfx12sf363-12

 Number of Slices:                     2211  out of   5472    40%
 Number of Slice Flip Flops:            103  out of  10944     0%
 Number of 4 input LUTs:               4293  out of  10944    39%
 Number of IOs:                         103
 Number of bonded IOBs:                 103  out of    240    42%
 Number of GCLKs:                         1  out of     32     3%
```

Figure 22: Area utilization - Decimal mutiplication

Post place and route results:

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   71.897|         |         |         |
---------------+---------+---------+---------+---------+
```

Figure 23: Timing report - Decimal mutiplication

3.2.2 Single precision

3.2.2.1 Addition

Addition is performed using the architecture shown in Figure 24; the role of each block is described below:

1. Conversion from IEEE-754 to sign, exponent and mantissa

   This block has been used to extract the sign, exponent, and mantissa from the represented numbers using bit selection, and then concatenate the implicit bit (1 in case of normal numbers and 0 in case of subnormal numbers).
2. Binary subtractor

   This block compare the two operands, set the exponent of the final result to the largest exponent, calculate the difference between the two exponent, also it has a signal greater indicate which operand is greater.
3. Significand alignment

   This block shift the mantissa of the smallest operand recognized using greater signal, number of shifts equal to the signal difference received from binary subtractor in case of two normal numbers and equal to (difference – 1) in case of normal and subnormal numbers, also it keeps the shifted bits to be used in rounding, the last two shifted bits in the guard and round bits respectively, and the or-ing of the remaining shifted bits in the sticky bit.
4. Binary adder

   This block adds the mantissa of the two operands.
5. Normalization

   This block receives the sum of the two mantissas and checks if there is a carry, then increments the exponent and shifts the resulted mantissa, the shifted bit goes to the guard bit, guard bit to round bit, and sticky bit is equal to the or-ing between the round bit and the sticky bit.
6. Rounding

   This block checks whether the guard bit and last bit in the mantissa are equal to one, or the guard bit, round bit and sticky bit are equal to one, if one of the two cases exist then increment the mantissa by one and raise inexact flag, and finally checks if there exist an overflow.
7. Conversion to IEEE-754 format

   This block takes the sign of the first operand and the final result of the exponent and the mantissa, and then puts the result in single precision representation formats.

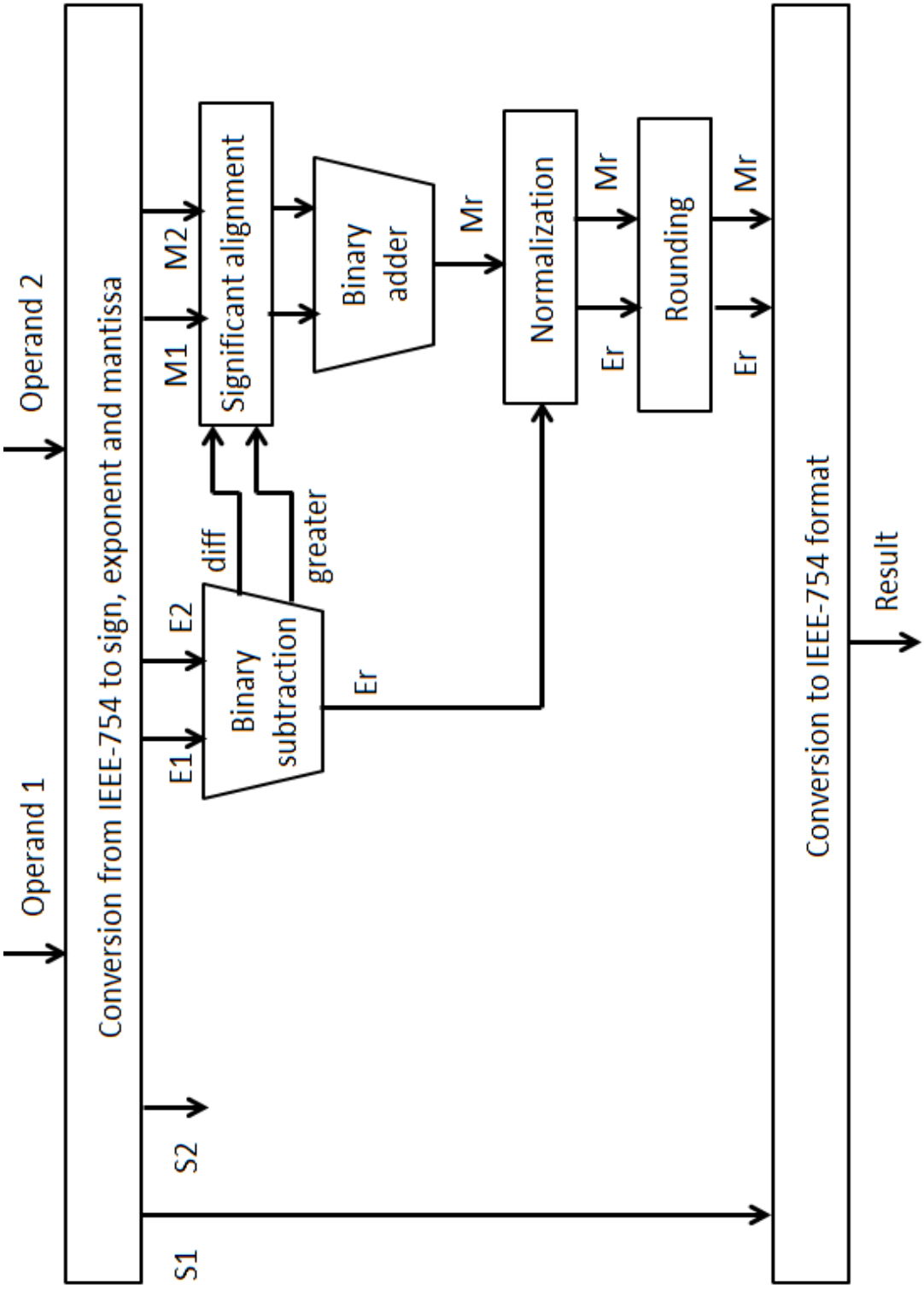

Figure 24: Architecture of addition in Single precision

Behavioral simulation results:

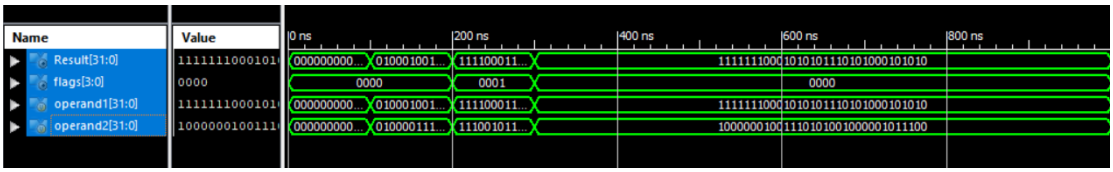

Figure 25: Behavioral simulation result – addition Single precision

Synthesis result:

```
Device utilization summary:
---------------------------

Selected Device : 4vfx12sf363-12

 Number of Slices:                      607  out of   5472    11%
 Number of Slice Flip Flops:             98  out of  10944     0%
 Number of 4 input LUTs:               1180  out of  10944    10%
 Number of IOs:                         103
 Number of bonded IOBs:                 102  out of    240    42%
 Number of GCLKs:                         1  out of     32     3%
```

Figure 26: Area utilization – addition Single precision

Post place and route results:

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   12.997|         |         |         |
---------------+---------+---------+---------+---------+
```

Figure 27: Timing report - addition Single precision

3.2.2.2 Subtraction

Subtraction is done using the architecture in Figure 28; each block has its role as described below, repeated blocks are explained above:

1. Conversion from IEEE-754 to sign, exponent and mantissa

   As described above in addition.
2. Binary subtractor

   As described above in addition
3. Significand alignment

   Same as in addition but it keeps the GRS bits in the shifted mantissa because they will enter in the subtraction process.
4. Binary subtractor

   This block subtract the two normalized mantissa with their GRS bits using two's complement, after making the two'2 complement to the second mantissa a normal binary addition is done and if there is a carry bit then the result is positive else then the result is negative and equals to the two'2 complement of the output of the adder.
5. Normalization

   This block receives the result of subtraction and removes all the leading zeros resulted from the subtraction and take the GRS bits instead of these leading zeros inside the bits that can be represented only if the exponent of the result is bigger than the number of leading zeros so we can subtract their number from the exponent to normalize the number and if not remove a number of leading zeros equal the (exponent -1) and assign the exponent to be equal zero so the number is now a subnormal number.
   If the resulted mantissa and exponent after normalization equal zero then raise the underflow flag.
6. Rounding

   As described above in addition.
7. Conversion to IEEE-754 format

   As described above in addition.

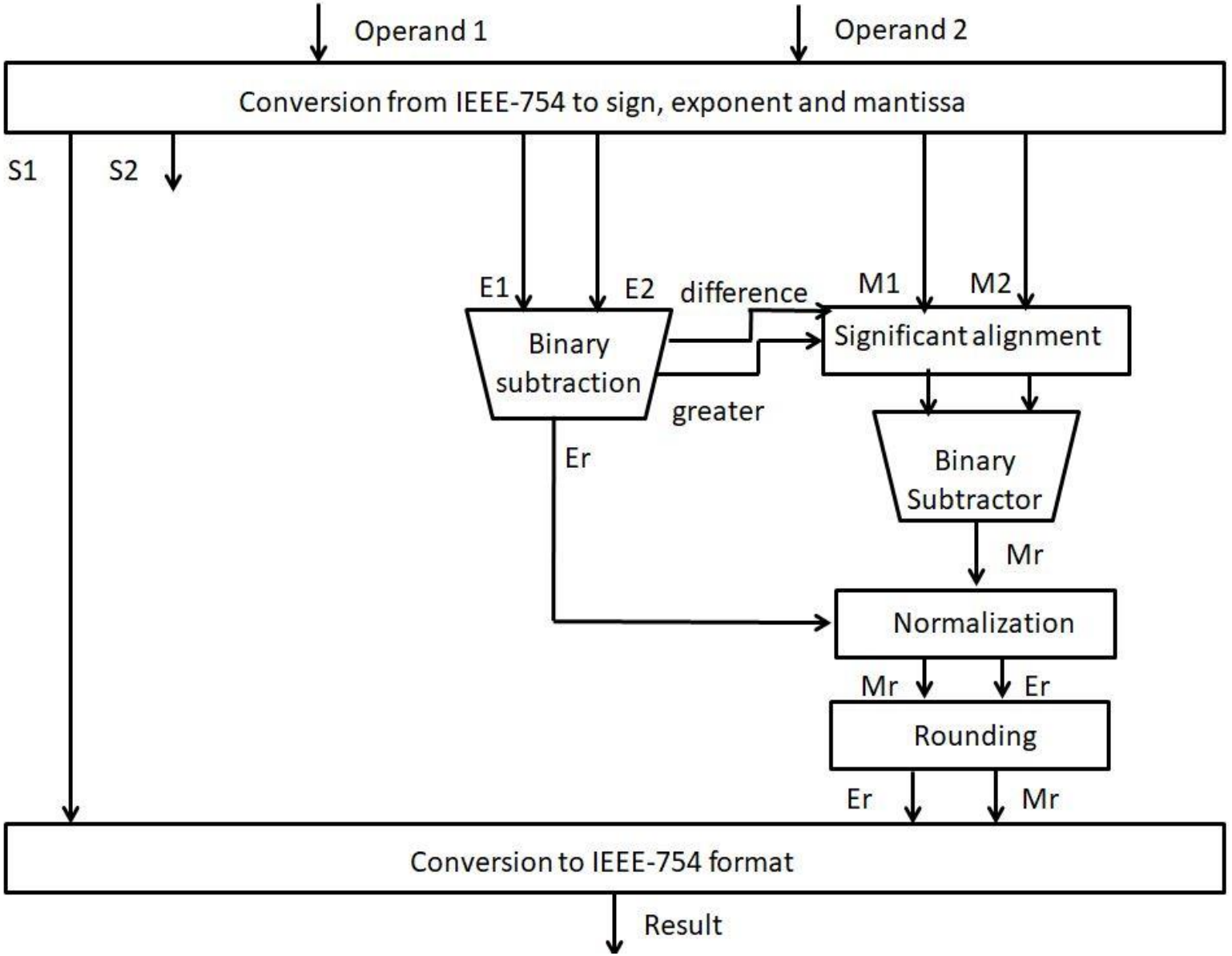
Operand 1
Operand 2
Conversion from IEEE-754 to sign, exponent and mantissa
S1
S2
E1
E2
difference
M1
M2
Binary subtraction
Significant alignment
greater
Er
Binary Subtractor
Mr
Normalization
Mr
Er
Rounding
Er
Mr
Conversion to IEEE-754 format
Result

Figure 28: Architecture of Subtractor in Single precision

Behavioral simulation results:

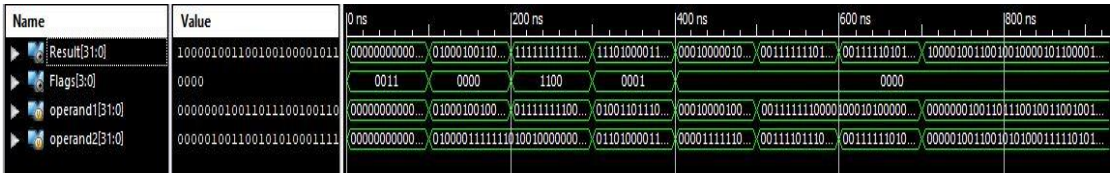

Figure 29: Behavioral simulation result – subtraction Single precision

Synthesis result:

```
Device utilization summary:
---------------------------

Selected Device : 4vfx12sf363-12

 Number of Slices:                      946  out of   5472    17%
 Number of Slice Flip Flops:             99  out of  10944     0%
 Number of 4 input LUTs:               1848  out of  10944    16%
 Number of IOs:                         103
 Number of bonded IOBs:                 102  out of    240    42%
 Number of GCLKs:                         1  out of     32     3%
```

Figure 30: Area utilization – subtraction Single precision

Post place and route results:

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   21.941|         |         |         |
---------------+---------+---------+---------+---------+
```

Figure 31: Timing report - Subtraction Single precision

3.2.2.3 Multiplication

Multiplication is performed using the architecture shown in Figure 32; the role of each block is described below:

1. Conversion form IEEE-754 standard to sign, exponent and mantissa

   As described above in addition.
2. Xor gate

   This gate is used to determine the sign of the result.
3. Binary adder

   This block calculates the result exponent by adding the exponent of the two operands, and then subtracts the bias from the sum, also it can raise the underflow or overflow signal is the result is greater or smaller than the available exponent.
4. Binary multiplier

   This block multiplies the mantissa of the two operands.
5. Normalization

   First, using a switch case this block determine the state of the result according to the numbers of leading zeros, assigns the mantissa to be the first 24 bits starting from the first one from the left, and assigns the following two bits to be guard and round bit respectively and the or-ing of the remaining bits to be sticky bit.
   Second, in each state check the underflow signal if it is raised, then decides whether the underflow can be solved by representing the number as a subnormal number or not and if the underflow signal isn't raised, then treats the number normally the subtracting the number of leading zeros from the exponent.
   Finally, check if there is an overflow occurs or not
6. Rounding

   As described above in addition.
7. Conversion to IEEE-754 format

   As described above in addition.

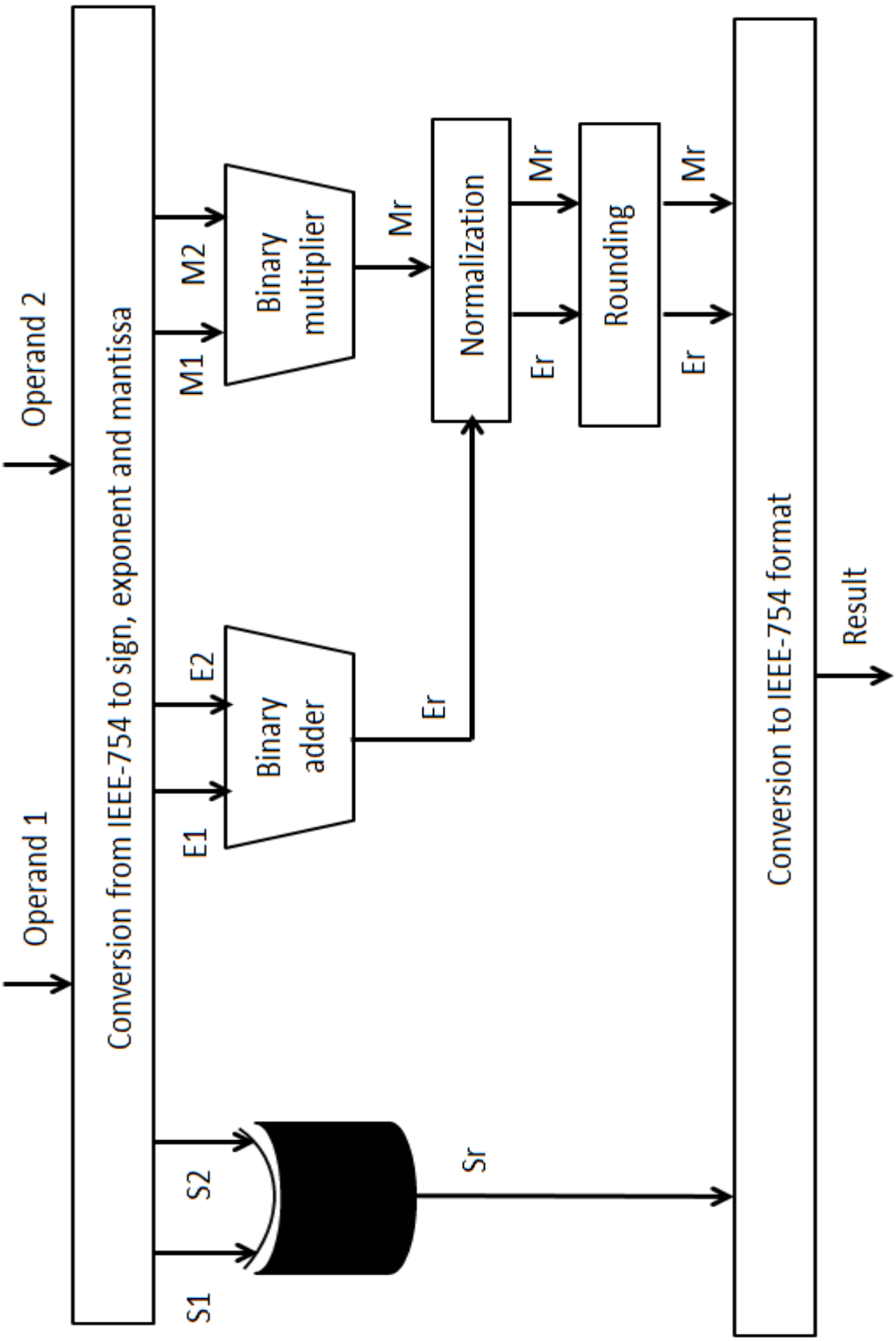

Figure 32: Architecture of multiplication in Single precision

Behavioral simulation results:

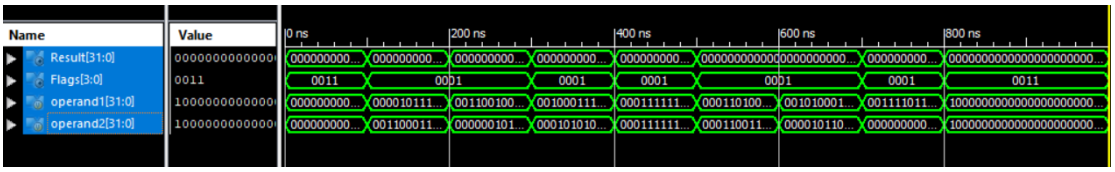

Figure 33: Behavioral simulation result – addition Single precision

Synthesis result:

```
Device utilization summary:
---------------------------

Selected Device : 4vfx12sf363-12

 Number of Slices:                     1511  out of   5472    27%
 Number of Slice Flip Flops:            100  out of  10944     0%
 Number of 4 input LUTs:               2919  out of  10944    26%
 Number of IOs:                         103
 Number of bonded IOBs:                 103  out of    240    42%
 Number of GCLKs:                         1  out of     32     3%
 Number of DSP48s:                        4  out of     32    12%
```

Figure 34: Area utilization – addition Single precision

Post place and route results:

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   25.000|         |         |         |
---------------+---------+---------+---------+---------+
```

Figure 35: Timing report - addition Single precision

3.2.3 Frequency of the design

After designing the six modules a high level module is created to integrate the six modules together, the whole design can work with the frequency of the slowest path that exists in the decimal multiplication as seen in Figure 36 it exists in the path where the BCD multiplier exists. So the design can work with the frequency of the critical path that is equal to 13.8 MHz, so pipelining is used To Increase the frequency of the whole design .
Pipelining is done by dividing the BCD multiplier to two parts as seen in Figure 38, now the design can work with frequency 25.8MHz which is the frequency of the critical path in the decimal subtractor.

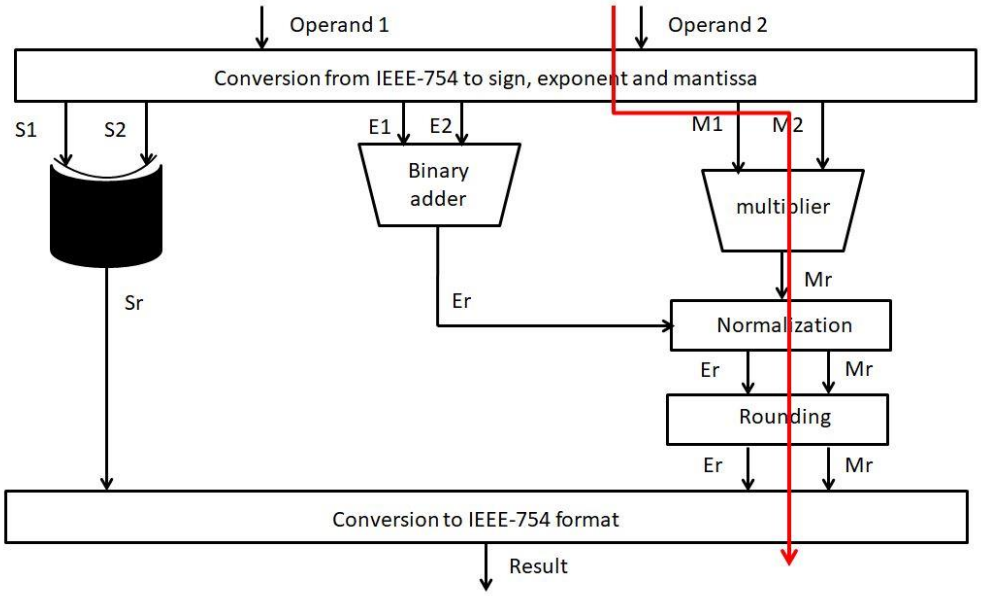

Figure 36: Critical path of decimal multiplication

Post place and route results:

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   31.329|         |         |         |
---------------+---------+---------+---------+---------+
```

Figure 37: Timing report - After pipelining

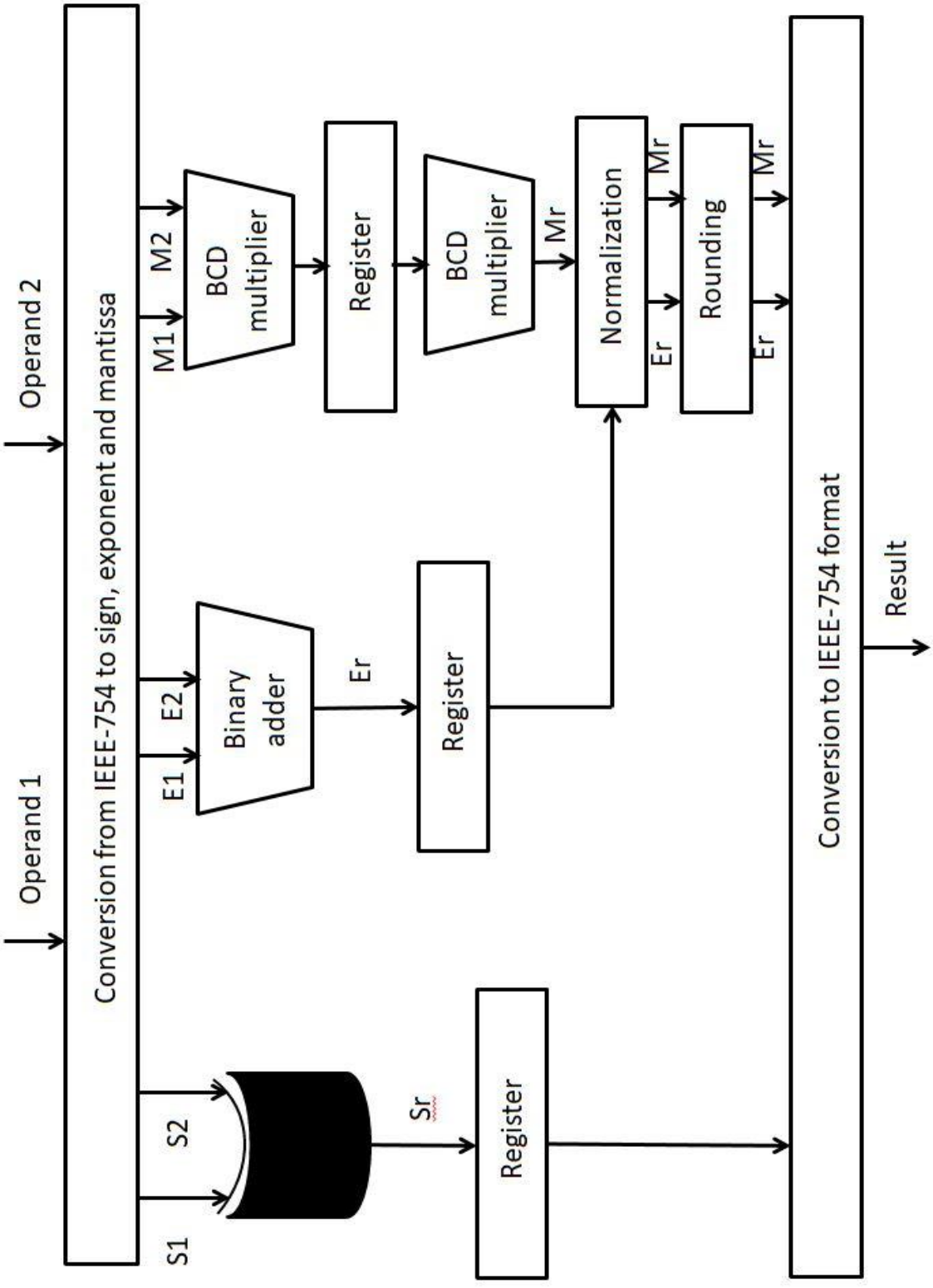

Figure 38: Architecture of the decimal multiplication after pipelining

CHAPTER FOUR: SYSTEM

4.1 HCI (HOST CONTRLLER INTERFACE) SPECIFICATION

4.1.1 Abbreviations

- **Reserved**: These registers/bits are reserved and should be set to zero
- **RO-Read Only**: If a register/bit is read only, this means that only the FPU can write into it, writes by the software have no effect and reads by the FPU return zeros.
- **WO-Write Only**: If a register/bit is write only, this means that only the software can write into it, writes by the FPU have no effect and reads by the software return zeros.
- **R/W-Read/Write**: If a register/bit is read/write, this means that both the software and the FPU can write into it and read from it. Note that individual bits in R/W registers may be RO or WO.
- **Single operation instruction**: instruction where the SIMD of the control bit of the FPU command register is set to zero.
- **SIMD instruction**: instruction where the SIMD of the control bit of the FPU command register is set to one, SIMD is the single instruction, multiple data.

4.1.2 Memory-mapped FPU Host Controller Registers

Configuration offset	Register Set	Number of registers	Register Access
000	FPU Command register	1	R/W
0x004-0x00C	Reserved	3	--
0x010-0x04C	Operand A	16	WO
0x050-0x08C	Operand B	16	WO
0x090-0x10C	Reserved	32	--
0x110-0x11C	FPU Status registers	4	R/W
0x120-0x12C	Reserved	4	--
0x130-0x16C	Output	16	RO
0x170-0x17C	Reserved	4	--

Table 8: Memory-mapped FPU Host Controller Registers

4.1.3 FPU Command register

Bit	Description
22-31	**Reserved**
18-21	**Number of SIMD operations – WO.** • Default 0000b (1 operation). This field identifies the number of SIMD instruction operations, the FPU checks this field if it's a SIMD instruction.

	• The FPU can carry out a maximum of 16 similar operations (value = 1111b) at each SIMD instruction.
17	**SIMD – WO.** • This control bit is used by the software to tell the FPU whether this instruction is a single operation instruction or a SIMD instruction, if the software sets this bit to one then it's a SIMD instruction else if it sets it to zero then this is a single operation instruction.
13-16	**Reserved**
11-12	**Operation – WO.** • Default 00b.This field identifies which operation will be performed. • Values mean: o 00b Addition o 01b Subtraction o 10b Multiplication o 11b Reserved
7-10	**Reserved**
5-6	**Floating-point format – WO.** • Default 00b.This field identifies which floating-point format is to be used. • Values mean: o 00b Binary32 (Single-precision) o 01b Reserved o 10b Decimal representation – Decimal format o 11b Reserved
4	**Reserved**
3	**Interrupt Enable – WO.** • This control bit is set to one by the software to tell the FPU to issue an interrupt by setting the interrupt signal to one when it finishes an operation which is reset to zero when the software raises the clear status bit of the FPU Status register (register 0x110-bit 1) to one. • In case this control bit is set to zero, the interrupt signal is masked (i.e. no interrupt signal is issued by the FPU when it finishes an operation).
2	**Doorbell – R/W.** • This control bit is set to one by software to tell the FPU that there is a new operation. • When the FPU starts the operation, it sets it to zero.
1	**FPU Enable – WO.** • This control bit is used by software to enable the FPU. • The FPU executes the operations as long as this bit is one. When the software sets this bit to zero, the FPU completes the current operation and then halts until the software sets this bit to one again.
0	**FPU Reset – R/W.** • This control bit is used by software to reset the FPU. • When software writes a one to this bit, the FPU terminates any operation in progress. • This bit is set to zero by the FPU when the reset process is complete.

	Software cannot terminate the reset process early by writing a zero to this register.

Table 9: FPU Command register

4.1.4 FPU Status register

4.1.4.1 Register (0x110)

Bit	Description
7-31	**Reserved**
6	**Inexact flag – RO.** • Default 0b. This control bit is set to one by the FPU when an operation delivers a numerical result that signal no other exception and its rounded result differs from what would have been computed were both exponent range and precision unbounded. • More details in IEEE754-2019 standard section 7.6
5	**Underflow flag – RO.** • Default 0b. This control bit is set to one by the FPU when a tiny non-zero result is detected. • More details in IEEE754-2019 standard section 7.5
4	**Overflow flag – RO.** • Default 0b. This control bit is set to one by the FPU 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. • More details in IEEE754-2019 standard section 7.4
3	**Division by zero flag– RO.** • **Reserved** and set to zero as the current FPU doesn't support division or logarithmic operations. • More details in IEEE754-2019 standard section 7.3
2	**Invalid operation flag – RO.** • Default 0b. This control bit is set to one by the FPU if and only if there is no usefully definable result in the cases where the operands are invalid for the operation to be performed. • More details in IEEE754-2019 standard section 7.2
1	**Clear status– WO.** • This control bit is set to one by the software after either receiving an interrupt signal in case the interrupt enable bit of the FPU Command register (bit 3) is set to one or after the software checks the status bit of the FPU Status register (register 0x110-bit 0) to find it set to one, the software sets this bit to one to tell the FPU to reset the status bit of the FPU Status register (register 0x110-bit 0) to zero.
0	**Status– RO.** • This control bit has a default value of zero, it is set to one by the FPU when it terminates an operation and is reset to zero when the software raises the

	clear status bit of the FPU Status register (register 0x110-bit 1) to one.

Table 10: Register (0x110)

Note: The five flags (bits 2 -6) in this register are the output flags in case of a single operation instruction, in case of a SIMD instruction they are the output flags of the first SIMD instruction operation.

4.1.4.2 Register (0x114)

Bit	Description
31	**Reserved**
16-30	**SIMD Division by zero flags– RO.** • **Reserved** and set to zero as the current FPU doesn't support division or logarithmic operations.
15	**Reserved**
0-14	**SIMD Invalid operation flags – RO.** • Each of these 15 bits have a default value of 0b, in case of a SIMD instruction, they are the invalid operation output flags, as explained for bit 1 in FPU status register (0x110), of the second to the 16^{th} SIMD instruction operations in order.

Table 11: Register (0x114)

4.1.4.3 Register (0x118)

Bit	Description
31	**Reserved**
16-30	**SIMD Underflow flags– RO.** • Each of these 15 bits have a default value of 0b, in case of a SIMD instruction, they are the underflow output flags, as explained for bit 4 in FPU status register (0x110), of the of the second to the 16^{th} SIMD instruction operations in order.
15	**Reserved**
0-14	**SIMD Overflow flags– RO.** • Each of these 15 bits have a default value of 0b, in case of a SIMD instruction, they are the overflow output flags, as explained for bit 3 in FPU status register (0x110), of the of the second to the 16^{th} SIMD instruction operations in order.

Table 12: Register (0x118)

4.1.4.4 Register (0x11C)

Bit	Description
15-31	**Reserved**
0-14	**SIMD Inexact flags– RO.** • Each of these 15 bits have a default value of 0b, in case of a SIMD

		instruction, they are the inexact output flags, as explained for bit 5 in FPU status register (0x110), of the of the second to the 16^{th} SIMD instructions operations in order.

Table 13: Register (0x11C)

4.1.5 Operands A & B and Output registers:

Each operand has 16 registers and so does the output, the first register of each operand is used in single operations and the result is written in the first output register, the different operations are carried out as follows:

- Addition: $output(0x130) = A(0x010) + B(0x050)$
- Subtraction: $output(0x130) = A(0x010) - B(0x050)$
- Multiplication: $output(0x130) = A(0x010) \times B(0x050)$

In case of SIMD instructions, the Number of SIMD operations field of the FPU command register determines how many similar operations are carried out which also determines the number of registers of each operand and the output that is to be used, operations are carried out on the registers of each operand and the output in order, to illustrate how this works with a simple example, given that the operation is addition and the number of SIMD operations is three, the addition SIMD operations are carried out as follows:

$$output(0x130) = A(0x010) + B(0x050)$$
$$output(0x134) = A(0x014) + B(0x054)$$
$$output(0x138) = A(0x018) + B(0x058)$$

4.1.6 Interrupt signal:

This signal has a default value of zero, it is set to one by the FPU when the interrupt enable bit of the FPU Command register (bit 3) is set to one and the FPU terminates an operation and is reset to zero when the software raises the clear status bit of the FPU Status register (register 0x110-bit 1) to one.

4.2 TOP LEVEL DESIGN

4.2.1 Design without SIMD support

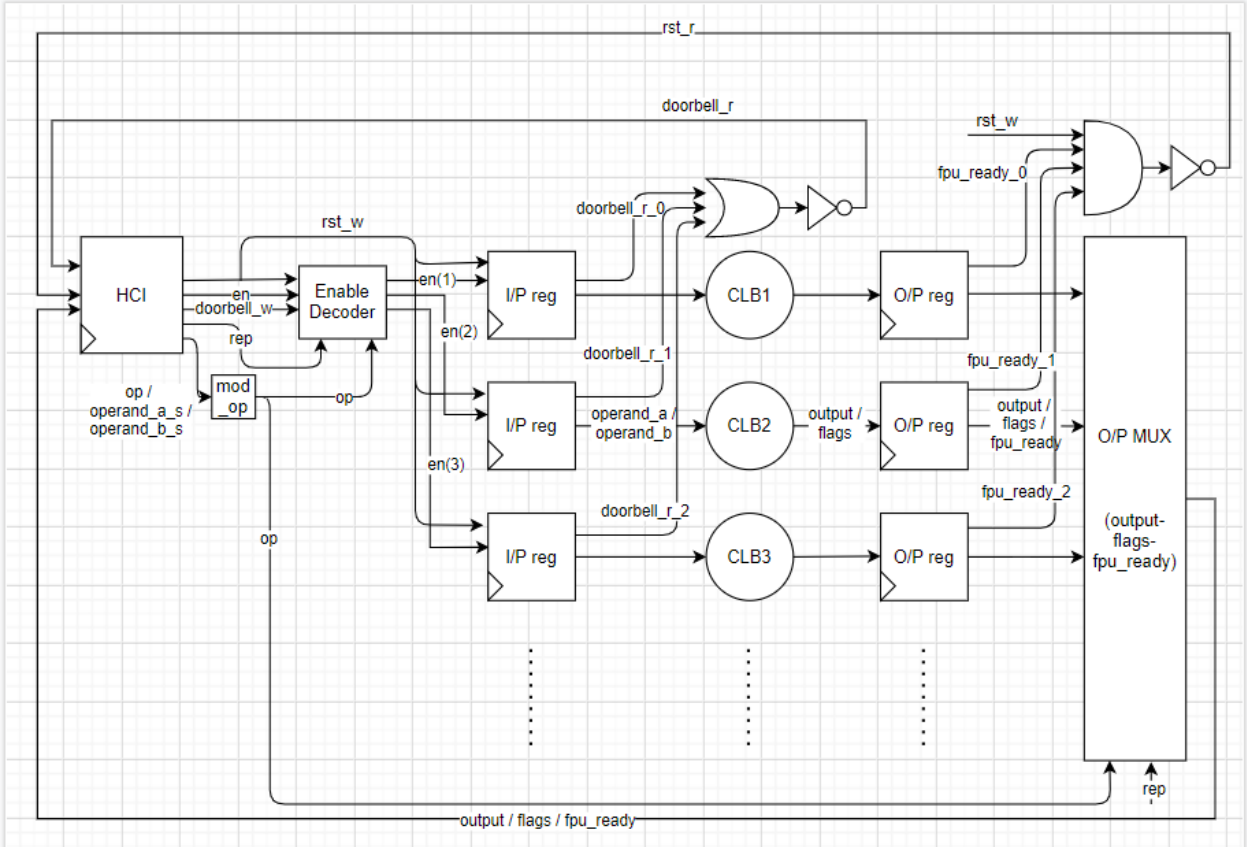

Figure 39: Top level block diagram without SIMD

HCI:

This block represents the interface between the software from one side and the FPU blocks from the other side, from the software side it only sends and receives 32 bits data with certain addresses, from the FPU blocks' side control signals and status signals are sent and received as well as operands and the FPU output as shown in Figure 40.The HCI block extracts the control signals that are sent to the FPU blocks from the data sent from the software as well as uses the status signals to update some bits in the data sent t the software. The HCI block contains some registers which are FPU Command register, Operand A 0x010 register, Operand B 0x050 register, FPU Status register 0x110 and a dataout register, it also contains a multiplexer and a demultiplexer to read from and write into these registers.

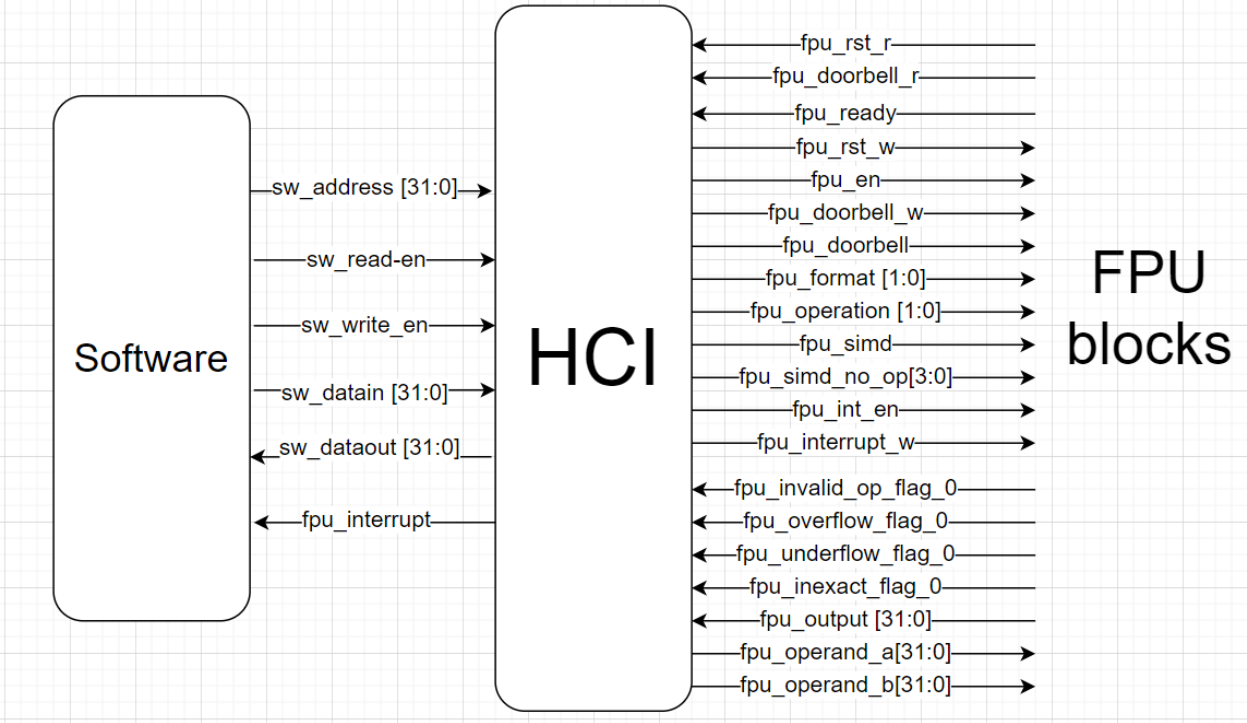

Figure 40: HCI connections

Modified operation:

This block uses three inputs which are the operation and the sign bits (bit number 31) of the two operands to determine the modified operation as shown in Table 14.

Operation	Sign bits of the operands	Modified operation
Adiition	Same sign	Addition
	Different signs	Subtraction
Subtaction	Same sign	Subtraction
	Different signs	Addition
Multiplicaioon	-	Multiplication

Table 14: Modified operation block function

Enable decoder:

This block uses the modified operation together with the representation to enable only one of the input registers to the different CLBs (combinational logic blocks) when there is a new single instruction or new SIMD data. In the SIMD case, the enable decoder operates with each new data because even though the operation and representation don't change, the signs of the operands change with different data which changes the modified operation and therefore the enable of the registers changes.

Input registers:

There are six input registers each connected to one of the CLBs, they register and output the operands to the CLBs, they input new operands when enabled by the enable decoder, when the FPU reset bit is set to one they reset all operands to zero and when enabled they output a signal to the output register of the same CLB to read the outputs and flags after one clock cycle.

CLBs (Combinational Logic Blocks):

The six CLBs are:

1. Decimal Adder
2. Decimal Subtractor
3. Decimal Multiplier
4. Single precision Adder
5. Single precision Subtractor
6. Single precision Multiplier

Input registers:

There are six output registers each connected to one of the CLBs, they register and output the outputs and flags of the CLBs, they input new outputs and flags when enabled by the signal from the input register connected to the same CLB and they also output a ready signal that is set to one whetn it receives the new outputs and flags.

Output multiplexer:

The output multiplexer inputs the outputs, flags and ready signals of the six output registers and outputs the desired according to the modified operation and representation.

Other logic:

The doorbell_r signal is fed back to the HCI to reset the Doorbell bit of the command register to zero indicating that the operands hane been inserted to the CLB.

The rst_r signal is fed back to the HCI to reset the FPU Reset bit of the command register to zero in case it was set to one indicating that the input and output registers have all been reset.

4.2.2 Design with SIMD support

In order to support SIMD instructions some blocks were added and some modifications were made as explained here.

Added blocks are:

- SIMD block
- Operands multiplexer
- Interrupt multiplexer
- Software dataout multiplexer

SIMD:

This block is the main block in supporting SIMD, it interfaces with the software to directly read the operands and register them, it also interfaces with the HCI through some control signals and with other FPU blocks as shown in Figure 41, it contains 16 registers for each of the operands, for the outputs and the flags, it also contains a finite state machine, counters ,multiplexers and demultiplexers that control the operands that are outputted and the outputs and flags that are read, it also achieves pipelining.

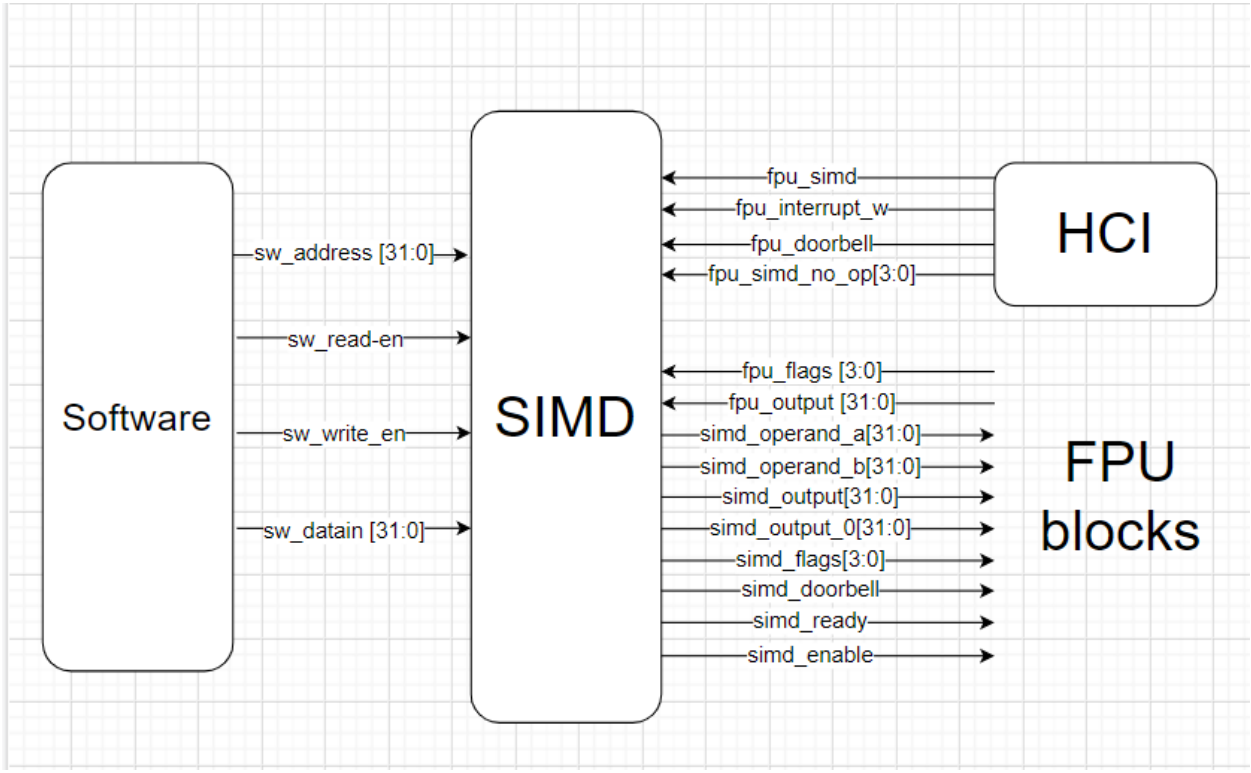

Figure 41: SIMD connections

Operands multiplexer:

This block chooses which operands are to be sent to the input registers either those from the HCI or the SIMD blocks according to whether it's a SIMD operation (SIMD bit of the command register is set to one) or not.

Interrupt multiplexer:

This block's name is misleading, its function is to choose which output, falgs and ready signal are to be sent to the HCI either those from the output multiplexer in case of single instruction or the ouput and flags of the first register in the SIMD block and the simd_ready signal in case of SIMD instruction.

Software dataout multiplexer:

This block interfaces with the software to output the sw_dataoout instead of the HCI, according to the required register data if it's the command register, status register 0x110 or output 0x130, it's read from the HCI output else for the other outputs and status registers, it's read from the SIMD output as their registers are located there.

4.2.3 Top level simulations

4.2.3.1 Single instruction:

Normal operation without interrupt enable:

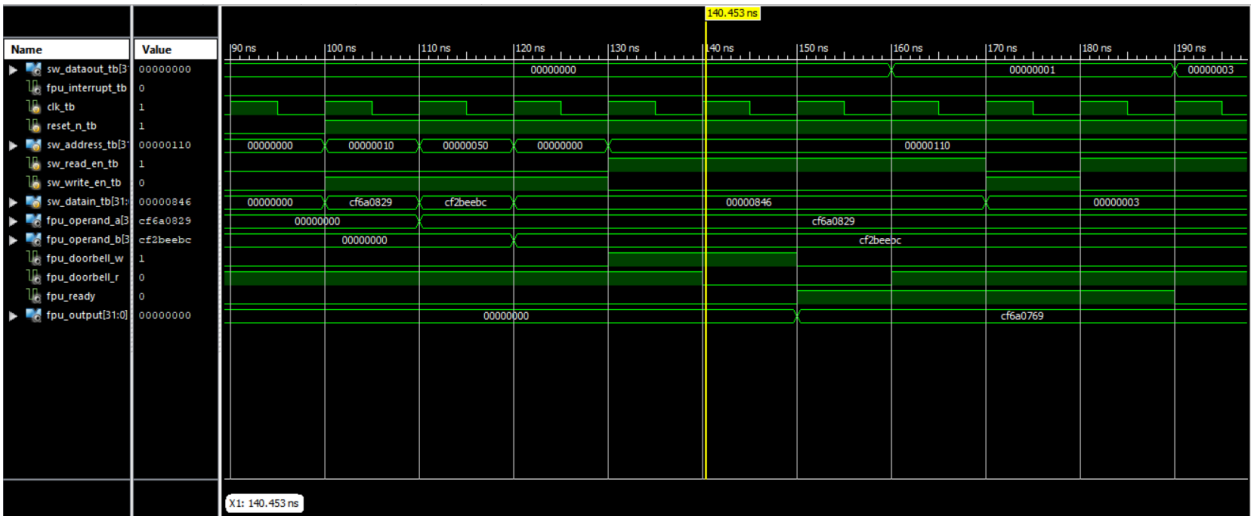

Figure 42: Single instruction simulation (A)

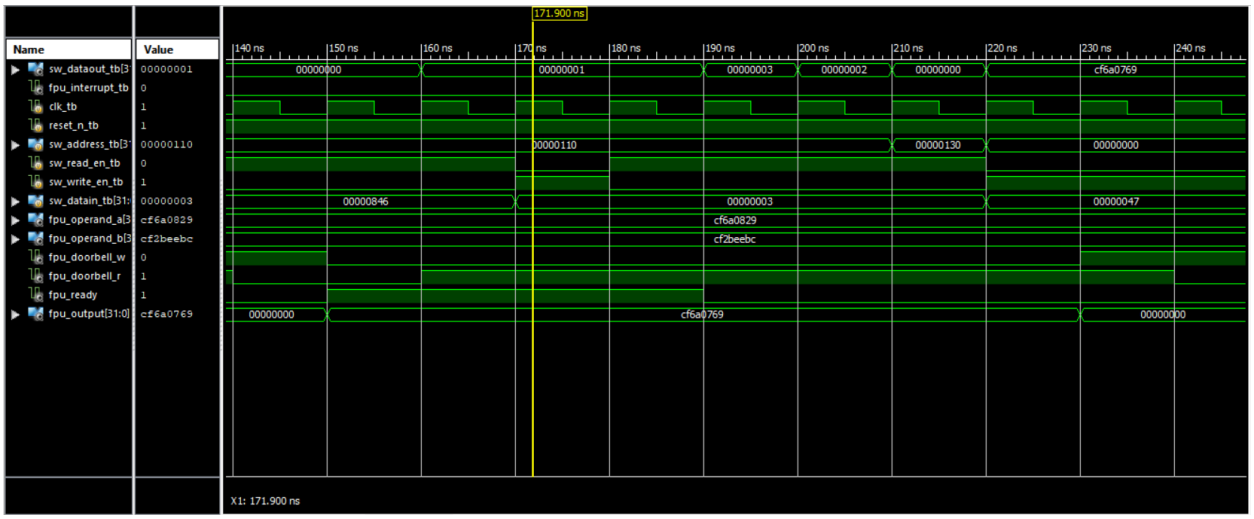

Figure 43: Single instruction simulation (B)

As shown in Figure 42 and Figure 43:

- First, the software sends operand a (0x10) then operand b (0x50), by observing when fpu_operand_a and fpu_operand_b values change it's clear that reading each takes one clock cycle.
- Then, the software sends the FPU command register (0x0) and the fpu_doorbell_w signal is set to 1
- The fpu_doorbell_r signal, output of the input register which is active low, is set to 0 after another clock cycle (this clock cycle is needed to register the inputs of the different representations/operations)
- The fpu_ready signal is set to 1 after another clock cycle (CLB delay).
- After another clock cycle the status bit of the FPU command register (0X110) is set to one and fpu_output is ready which means that the operation takes 3 clock cycles from the negative edge of the sw_write_en signal after reading the FPU command register until the status bit is set to one.
- After that the clear status bit of the FPU command register (0X110) is set to one by the software.
- The fpu_ready signal is reset to zero after one clock cycle.
- After another clock cycle the status bit is reset to zero and in the following clock cycle the clear bit is reset to zero as well, both are reset to zero by the FPU not the software.
- The output (0X130) is then read but it can be read at any clock cycle afte the status bit is set to one.

FPU reset bit:

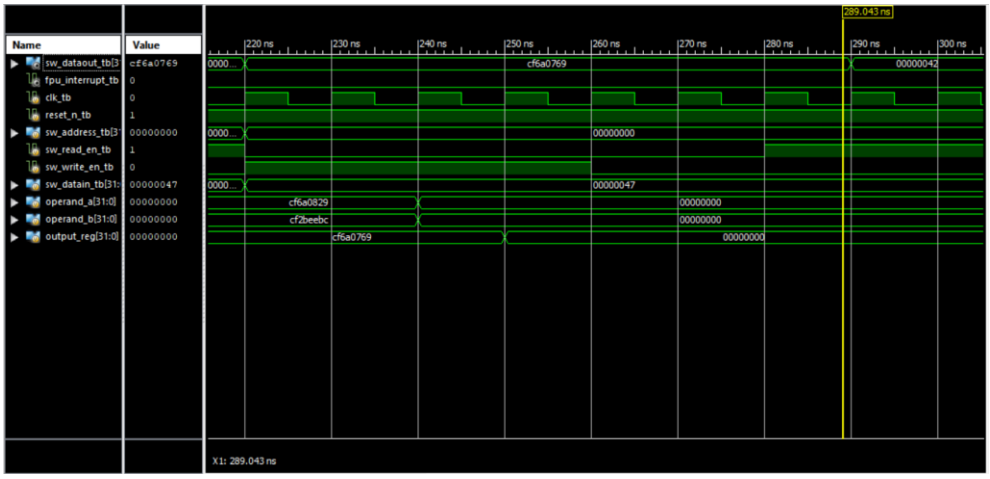

Figure 44: FPU reset bit simulation

Figure 44 shows the case where the FPU reset bit of FPU command register (0x0) is set to one, the operands registered in the input register are set to zero after two clock cycles and the outputs registered in the output register are set to zero after another clock cycle, the software can read the FPU command register (0x0) to find that this signal is reset to 0 after 3 clock cycles from the negative edge of the sw_write_en signal.

FPU enable bit:

Figure 45: FPU enable bit simulation

Figure 45 shows the case where the FPU enable bit is kept zero, the fpu_doorbell_w signal is not set to one and therefore no operation is carried out, trying to read the output, it is zero due to a preceding reset.

FPU interrupt enable bit:

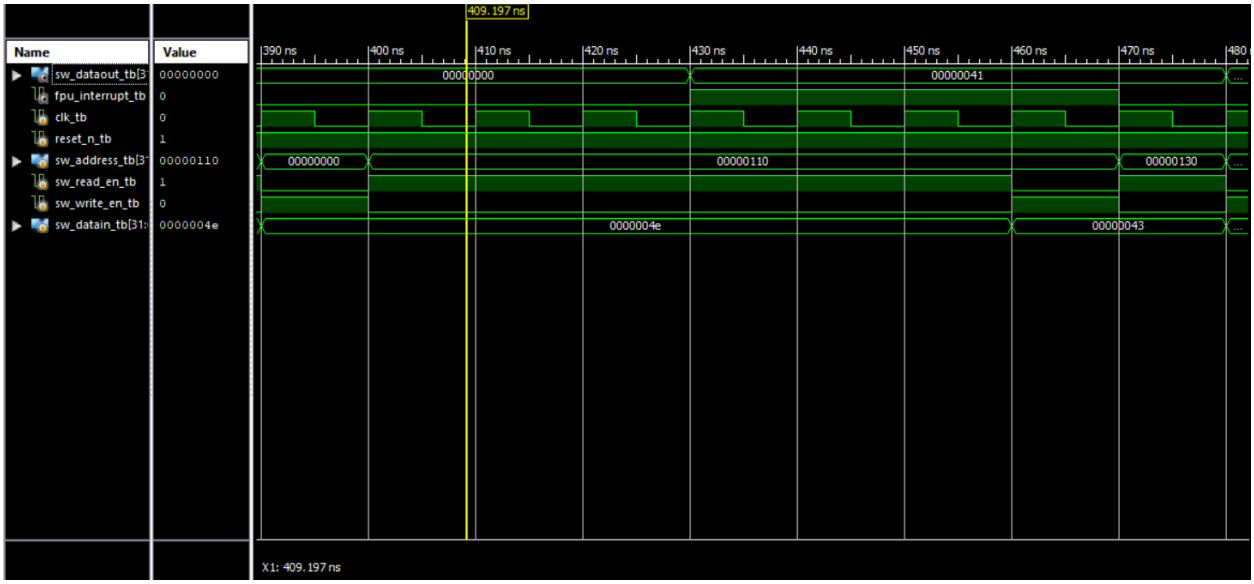

Figure 46: FPU interrupt enable bit simulation

Figure 46 shows the case where the FPU interrupt enable bit is set to one, the fpu_interrupt signal rises to one with the status bit of the FPU status register (0x110) and drops to zero again when the clear status bit is set to one.

4.2.3.2 SIMD instruction:

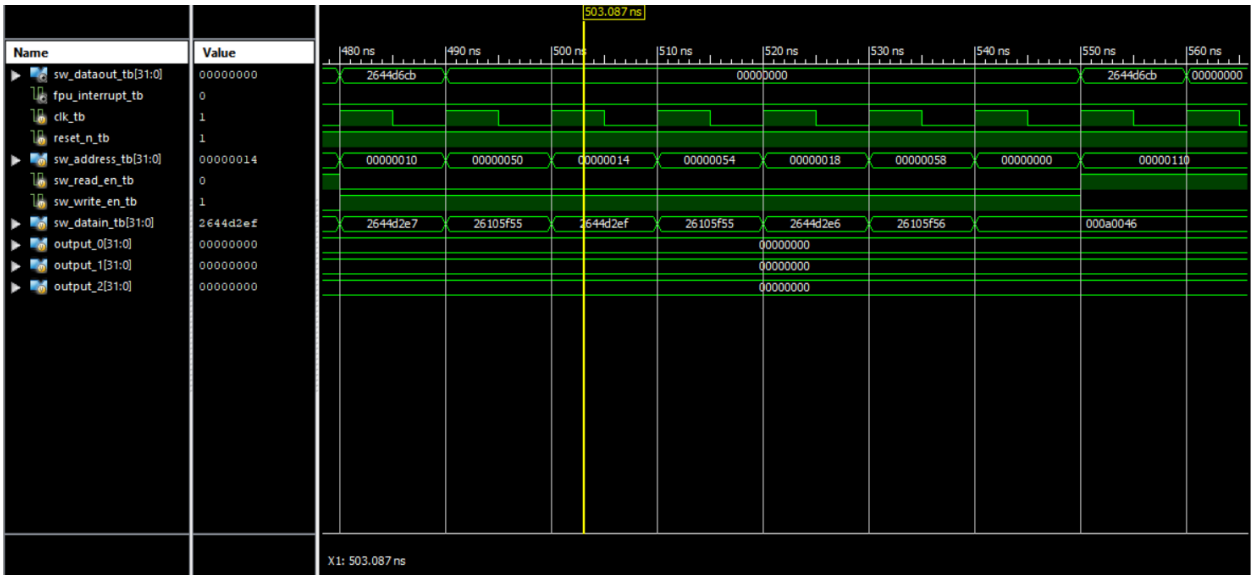

Figure 47: SIMD instruction simulation (A)

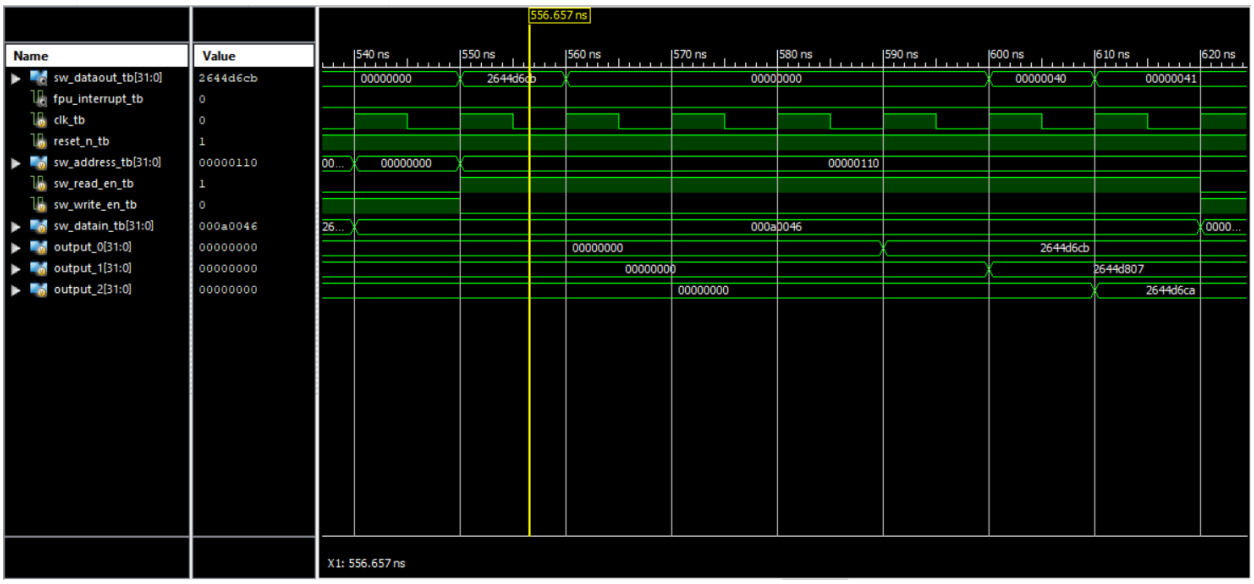

Figure 48: SIMD instruction simulation (B)

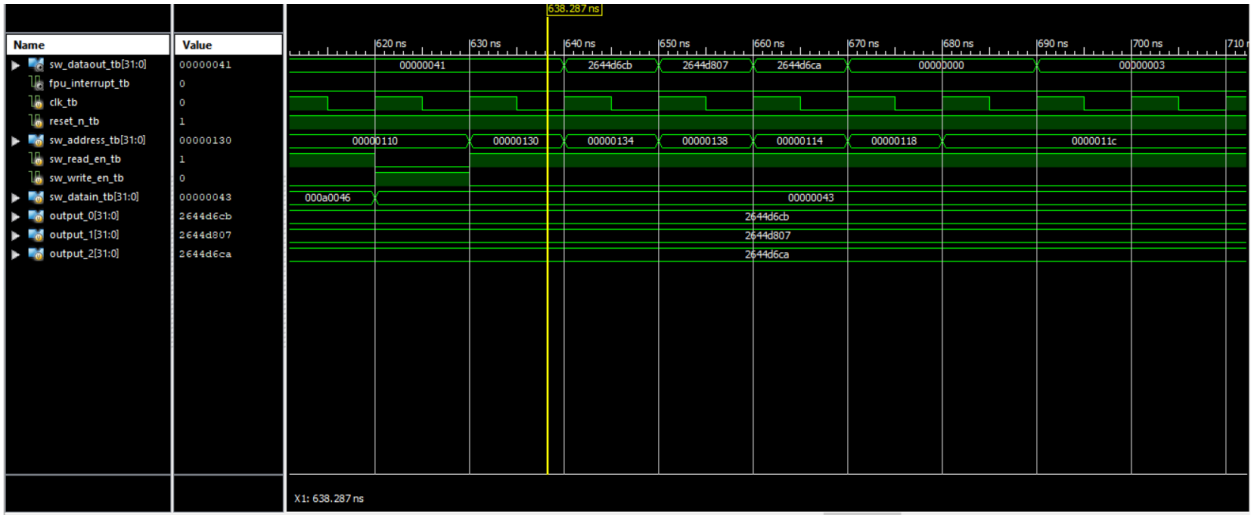

Figure 49: SIMD instruction simulation (C)

As shown in Figure 47, Figure 48 and Figure 49:

- First, all the operands are written.
- Then, the FPU command register (0x0) is written by the software with the simd bit set to one and the number of simd operations bits set to the desired value, here three operations are carried which means that the number of simd operations bits are set to 4'b0010.
- The operation takes 4(latency)+n-1 (n : number of SIMD operations) clock cycles from the negative edge of the sw_write_en signal after reading the FPU command register until the status bit is set to one instead of 3*n
- The outputs and the other status registers (contain flags) can then be read.

4.3 RISC-V

The RISC-V is an open-source ISA that was originally developed in the **Computer Science Division** of the EECS Department at the **University of California, Berkeley**.

4.3.1 RISC-V features

- An open-source ISA (Instruction Set Architecture) without the financial burden of licensing fees
- Simple
  - Far smaller than other commercial ISAs
  - Clean slate design
- A modular ISA
  - Small standard base ISA (I)
  - Multiple standard extensions (M A F D G C)
- Designed for Extensibility/Specialization
  - Variable-length instruction encoding
  - Vast opcode space available for instruction-set extensions
- Stable
  - Base and standard extensions are frozen
  - Addition via optional extensions, not new versions
- Engineers can choose to go big, small, powerful or lightweight with their designs.

4.3.2 RISC-V processors

A Survey of different implementations of the RISC-V processor was carried out and a comparison between them was made.
First, some cores were excluded, some examples for reasons for excluding cores are:

- Language (not Verilog or SystemVerilog)
- Not open source (e.g. Andes- Nuclei)
- No debugger (e.g. PicoRV32)
- GitHub Star (e.g. starsea_riscv-0 star)
- License (e.g. RV12)

Then, three SoCs (System on Chips) were chosen SweRVolf, PULPino and PULPissimo.

- SweRVolf
  - Core: EH1
    - Language: SystemVerilog
    - RV32IMC
    - On-chip debugger (OpenOCD)
- PULPino and PULPissimo
  - Cores:
    1. Zero-risky
       - Language: SystemVerilog

- *RV32IMC*
- *Dubugger: RISC-V debug specification 0.13*

2. RI5CY
   - Language: SystemVerilog
   - RV32IM[F]C 32-bit
   - Optional full support for RV32F Single Precision Floating Point Extensions (Floating-point support in the form of IEEE-754 single precision)
   - Dubugger: RISC-V debug specification 0.13

SweRVolf	PULPino	PULPissimo
		uDMA Subsystem
UART	UART	UART
SPI	SPI Master	SPI Master
	I2C	I2C
		I2S
		CAMIF
GPIO	GPIO	GPIO
RISC-V timer	Timer	Timer
	Event/Interrupt Unit	Event/Interrupt Unit
	FLL	FLL
System controller	SoC Control	SoC Control
DMI	Debug Port	Debug Unit
		SoC Event Generator
		Advanced Timer

Table 15: Peripherals of SweRVolf, PULPino and PULPissimo

Work was done on the three SoCs trying to build and run simulation examples on them and then trying to modify these examples to later build our own application, only SweRVolf and PULPino were successfully built and a simulation example was run on each, PULPino example was easily modified by changing in C codes unlike SweRVolf, therefore PULPino was chosen for integration.

4.4 PULPino

PULPino is an open-source single-core microcontroller system, based on 32-bit RISC-V cores developed at ETH Zurich. PULPino is configurable to use either the RISCY or the zero-riscy core.

4.4.1 Features

- Processor (Open-source RISC-V ISA processor).
- Ultra-low-power and ultra-low-area constraints.
  Most of PULPino blocks are gated by clock (to turn off any useless block during operations so it can save more power).
  The peripherals connected to APB bus that is less power consumption than AXI bus.
- RI5CY or zero-riscy core.
  The two cores have the same external interfaces and are thus plug-compatible.
  The difference between RI5CY and zero-riscy is that the RI5CY core support more ALU ISA extensions and complex operations than zero-riscy.
  We are working with the RI5CY core which is enabled by default.
- Contains a broad set of peripherals: I2C SPI UART
- Available for FPGA (Synthesizable written in System Verilog)

4.4.2 PULPino Architecture

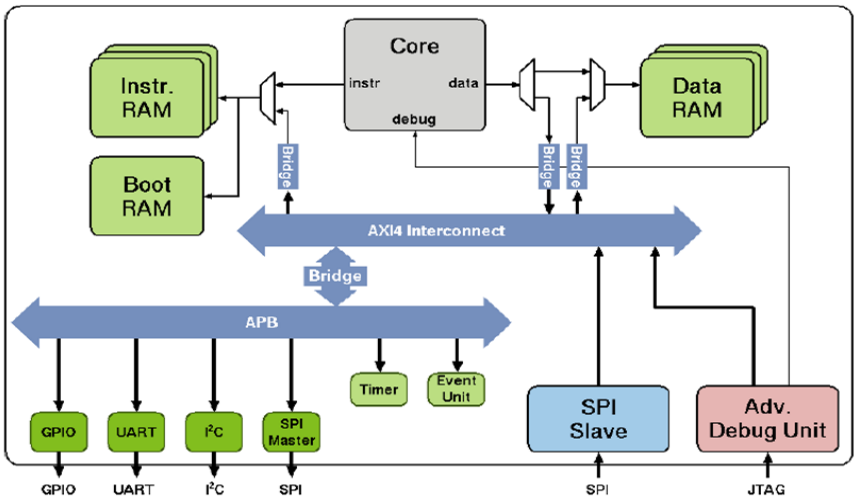

Figure 50: PULPino block diagram

The SoC uses a AXI as its main interconnect with a bridge to APB, both the AXI and the APB buses feature 32 bit wide data channel, all peripherals in PULPino are connected to the APB bus except the SPI slave which is a very special peripheral and not intended to be used from the core itself. (3)
The core uses a very simple data and instruction interface to talk to data and instruction memories directly.

4.4.3 Memory map

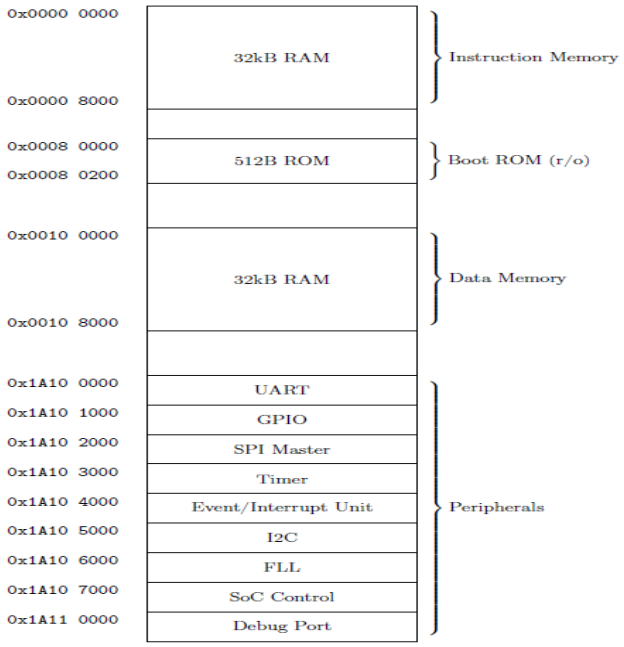

Figure 51: PULPino memory map

4.4.4 PULPino environment

In order to build Pulpino platform, Ubuntu 18.04 Linux VM image on VMware was used.

4.4.4.1 Pulpino requirements

1. ModelSim in reasonably recent version

   Modelsim-Intel FPGA Lite (Free) Edition for Linux (release: 19.1) was used, but since Intel only supports Red Hat-based distros like CentOS Linux, at first Modelsim didn't work, but by looking for some hacks and scripting some edits and changes as shown in Figure 52, Modelsim worked on the used Ubuntu and the installation path to the bin was added to the ".bashrc" path. (4)

```
# Start with ModelSimSetup-*-linux in your home directory

# Open a terminal in your home directory

# Update your system
sudo apt update; sudo apt upgrade

# Run the installer
./ModelSimSetup-*-linux.run

# Make the vco script writable
chmod u+w /home/ubuntu/intelFPGA/*.*/modelsim_ase/vco

# Make a backup of the vco file
(cd /home/ubuntu/intelFPGA/*.*/modelsim_ase/ && cp vco vco_original)

# Edit the vco script manually, or with these commands:
sed -i 's/linux\_rh[[:digit:]]\+/linux/g' \
    /home/ubuntu/intelFPGA/*.*/modelsim_ase/vco
sed -i 's/MTI_VCO_MODE:-""/MTI_VCO_MODE:-"32"/g' \
    /home/ubuntu/intelFPGA/*.*/modelsim_ase/vco
sed -i '/dir=`dirname "$arg0"`/a export LD_LIBRARY_PATH=${dir}/lib32' \
    /home/ubuntu/intelFPGA/*.*/modelsim_ase/vco

# Check that the correct lines have changed
diff /home/ubuntu/intelFPGA/*.*/modelsim_ase/vco \
    /home/ubuntu/intelFPGA/*.*/modelsim_ase/vco_original

# Download the 32-bit libraries and build essentials
sudo dpkg --add-architecture i386
sudo apt update
sudo apt install build-essential
    sudo apt install gcc-multilib g++-multilib lib32z1 \
    lib32stdc++6 lib32gcc1 libxt6:i386 libxtst6:i386 expat:i386 \
    fontconfig:i386 libfreetype6:i386 libexpat1:i386 libc6:i386 \
    libgtk-3-0:i386 libcanberra0:i386 libice6:i386 libsm6:i386 \
    libncurses5:i386 zlib1g:i386 libx11-6:i386 libxau6:i386 \
    libxdmcp6:i386 libxext6:i386 libxft2:i386 libxrender1:i386

cd
# Download the old 32-bit version of libfreetype
wget download.savannah.gnu.org/releases/freetype/freetype-2.4.12.tar.bz2
tar xjf freetype-2.4.12.tar.bz2

# Compile libfreetype
cd freetype-2.4.12/
./configure --build=i686-pc-linux-gnu "CFLAGS=-m32" \
    "CXXFLAGS=-m32" "LDFLAGS=-m32"
make clean
make

cd /home/ubuntu/intelFPGA/*.*/modelsim_ase/
mkdir lib32
cp ~/freetype-2.4.12/objs/.libs/libfreetype.so* lib32/
```

Figure 52:Commands for making Modelsim work

2. python2 >= 2.6

   The installed version is 2.7.17 in addition to installing python yaml.

3. CMake >= 2.8.0

   Used script is shown in Figure 53 and then the bin was added to the ".bashrc" path

```
sudo apt-get install build-essential
wget http://www.cmake.org/files/v3.2/cmake-3.2.2.tar.gz
tar xzf cmake-3.2.2.tar.gz
cd cmake-3.2.2
./configure
make
```

Figure 53: Script for installing and making Cmake

4. riscv-toolchain

   ri5cy_gnu_toolchain was used, errors arose at first while running make, by searching I reached a way that by making some changes in some files and rerunning make, it finished successfully and the installation path to the bin was added to the ".bashrc" path. (5)

   Changes to build ri5cy_gnu_toolchain:

   - git clone https://github.com/pulp-platform/ri5cy_gnu_toolchain.git
   - cd ri5cy_gnu_toolchain
   - Run make. It will download some files, encounter an error and stop.
   - cd build/src/newlib-gcc/gcc/cp
   - Open cfns.gperf in your favorite text editor and remove lines below the first comment (starting at line 19, inclusive) up until the line containing "%}" without the quotation marks. After that's done, right after the comment ends with "*/" without quotation marks, the next line should be "%}" without quotation marks.
   - gperf -o -C -E -k '1-6,$' -j1 -D -N libc_name_p -L C++ --output-file cfns.h cfns.gperf
   - Open except.c and on line 1043 add "Perfect_Hash::" without quotation marks exactly in front of "libc_name_p" without quotation marks.
   - cd back to ri5cy_gnu_toolchain and run make. It should not encounter any errors and should finish successfully.

4.4.4.2 Running simulations

To run simulation of hello world example in Modelsim console, the script in Figure 54 were used and the output is shown in Figure 55

```
git clone https://github.com/pulp-platform/pulpino.git
cd pulpino
./update-ips.py
cd pulpino/sw
mkdir build
cp cmake_configure.riscv.gcc.sh ./build
cd build
sh ./cmake_configure.riscv.gcc.sh
make vcompile
make helloworld.vsim
```

Figure 54: Script for running hello world example

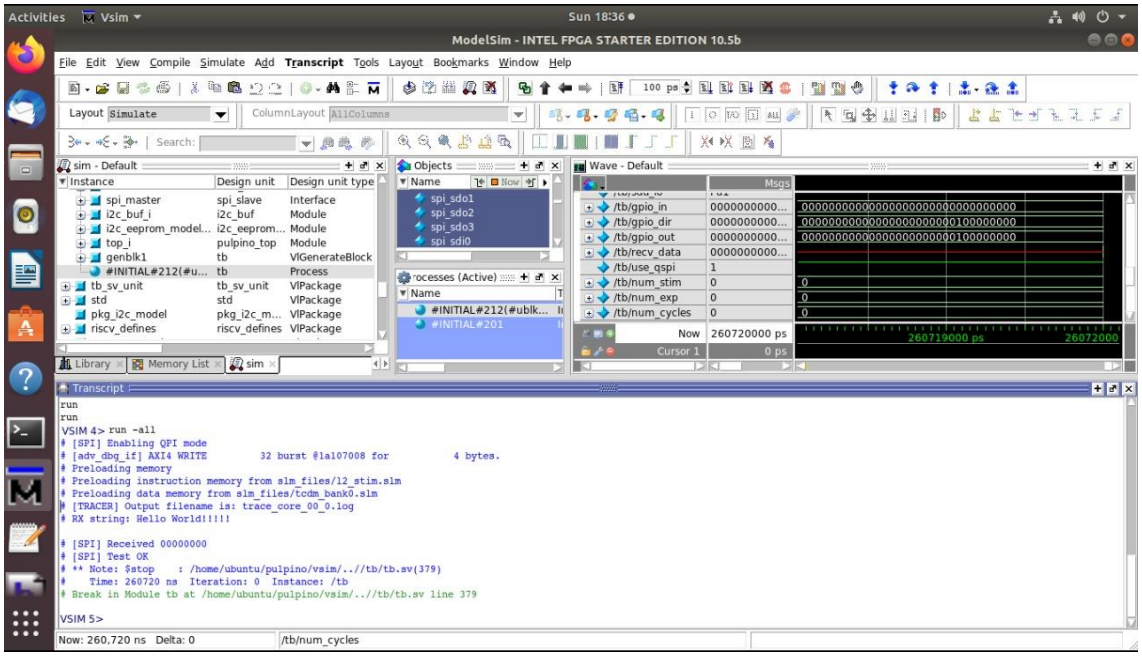

Figure 55: Output of hello world example

4.5 FPU-RISC V INTEGRATION

4.5.1 Integration methods

There were three methods to integrate the FPU with the RISC-V:

1. To connect it through the UART as an intermediate interface between both RISC-V and FPU.
2. To replace one of the peripherals with the FPU to be directly connected to the APB.
3. To connect the FPU directly to the APB as a new peripheral.

The first method is not preferred because it requires unnecessary time and more complex applications to handle the data between two different peripherals

The second method was the one used, the I2C peripheral is the one chosen to be replaced since

- The size of its memory suits that specified in the HCI memory map specification as shown in Table 8, Pulpino memory map after this replacement is shown in Figure 56.
- The FPU registers slightly resemble those specified in the HCI specifications.

The third method is more practical as in practical one would want to extend or add new peripherals to existing ones rather than replace an exsisting one but since it needs more modifications in PULPino files than the second case and there wouldn't be a difference in functionality, the second case was chosen yet this case is a better practice.

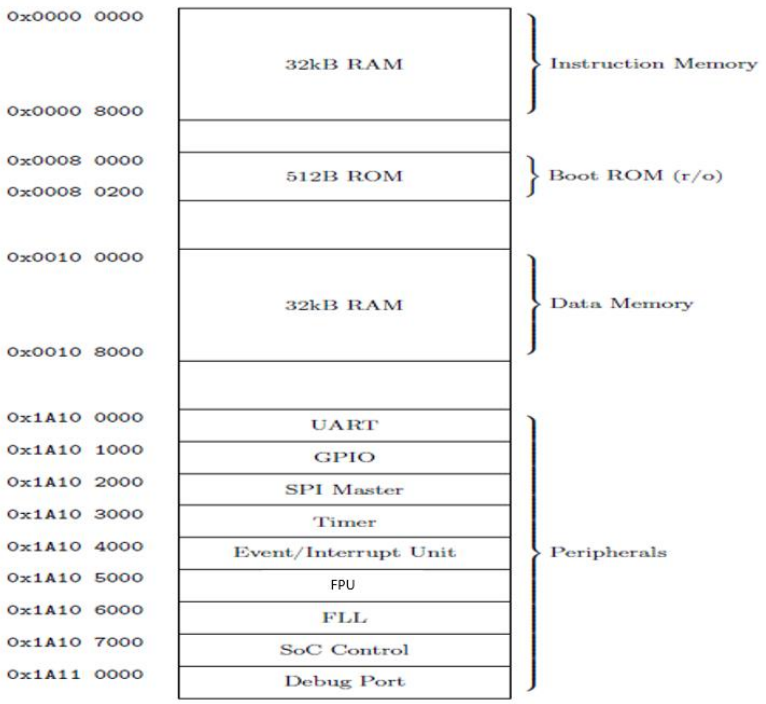

Figure 56: Pulpino memory map after replacement of I2C by FPU

4.5.2 Integrating the FPU with RISC-V via a Bridge

An apb_fpu_bridge was designed to integrate the FPU with the APB since the FPU have input/output signals different from that of the APB and a different address register size, the bridge is a combinational block written in Verilog and it interfaces the APB from the left and the HCI of the FPU from the right, the input and output signals on both sides are shown in Figure 57

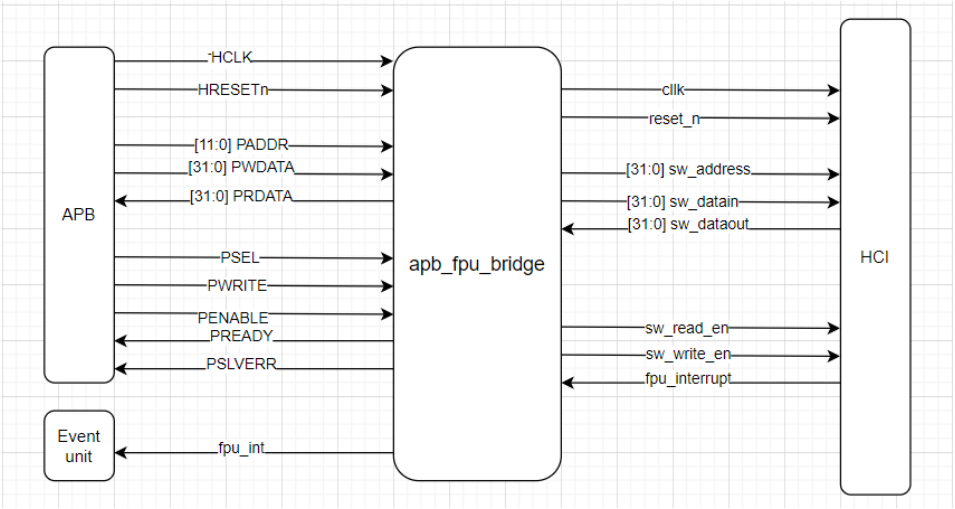

Figure 57: apb_fpu_bridge input/output signals

4.5.2.1 Bridge-APB interface

The used APB signals' description according to the AMBA APB Protocol (Version: 2.0) are shown in Table 16: APB used signals descriptionTable 16 (6)

Signal	Desciption
PCLK	Clock. The rising edge of PCLK times all transfers on the APB.
PRESETn	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal.
PADDR	Address. This is the APB address bus. It can be up to 32 bits wide (here 12 bits wide) and is driven by the peripheral bus bridge unit
PWDATA	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when PWRITE is HIGH. This bus can be up to 32 bits wide (here 32 bits wide)
PRDATA	Read Data. The selected slave drives this bus during read cycles when PWRITE is LOW. This bus can be up to 32-bits wide (here 32 bits wide)
PSELx	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a PSELx signal for each slave.
PWRITE	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PENABLE	Enable. This signal indicates the second and subsequent cycles of an APB transfer
PREADY	Ready. The slave uses this signal to extend an APB transfer.
PSLVERR	This signal indicates a transfer failure. APB peripherals are not required to support the PSLVERR pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW

Table 16: APB used signals description

4.5.2.2 Bridge-FPU interface

Signal	Desciption
Clk	Clock
reset_n	Asynchronous reset
sw_address	Address which is 32 bits wide
sw_datain	Write data which is 32 bits wide
sw_dataout	Read Data which is 32-bits wide
sw_read_en	Read enable one bit which enables a read operation
sw_write_en	Write enable one bit which enables a write operation
fpu_interrupt	Interrupt

Table 17: FPU signals description

4.5.2.3 Bridge-Event unit interface

The fpu_int signal is connected to PULPino lightweight event and interrupt unit.

4.5.2.4 Integration steps

1) Replace I2C rtl files with the FPU and the bridge rtl files.
2) Replace the I2C instantiation by the FPU instantiaton (peripherals.sv).
3) Remove inputs/outputs of the I2C (pulpino_top.sv).
4) Replace I2C rtl files directories in vsim vcompile scripts by those of the FPU (vcompile_apb_i2c.csh).
5) Compile and load design.

4.6 FPU APPLICATION AND TESTING

4.6.1 FPU application

The FPU application was written in C programming language, four functions were added to the I2C.c and I2C.h files which are:

1. FPU_Single_Instruction
2. FPU_SIMD_Instruction
3. Compare
4. FPU_get_status

4.6.1.1 FPU Single Instruction

```
int FPU_Single_Instruction(int operand_a,int operand_b,char op, char* rep, int *flags)
```

Figure 58: FPU_Single_Instruction header

Function Inputs:

- Two operands
- Operation
- Representation

Function Outputs:

- FPU output
- Flags

Function code flow:

1. Wait until status bit is set to zero by reading the command register in a loop.
2. Write operands.
3. Write command register
    - FPU Enable and Doorbell are set to one, Interrupt enable is set to zero.
    - Floating-point format and operation are set according to the floating-point representation and operation sent.

4. Wait until status bit is set to one by reading the command register in a loop.
5. Reset status bit to zero by writing to the command register the same value read fom it (to not affect the flags as they'll be read later) but eith the clear bit set to one.
6. Read output.
7. Read flags.

```
# RX string: FPU

# RX string: Decimal addition Output = 2644d6cb

# RX string: Invalid operation flag = 0

# RX string: Division by zero flag = 0

# RX string: Overflow flag = 0

# RX string: Underflow flag = 0

# RX string: Inexact flag = 1
```

Figure 59: FPU_Single_Instruction output

4.6.1.2 FPU SIMD Instruction

```
int * FPU_SIMD_Instruction(int* operand_a,int* operand_b,char op, char* rep, int num, int simd_flags[])
```

Figure 60: FPU_SIMD_Instruction header

Function Inputs:

- Two arrays for operands
- Operation
- Representation
- Number of SIMD operations

Function Outputs:

- Array of FPU outputs
- Array of flags

Function code flow:

1. Wait until status bit is set to zero by reading the command register in a loop.
2. Write operands in a loop according to number of SIMD operations.
3. Write command register
    - FPU Enable and Doorbell are set to one, Interrupt enable is set to zero.
    - Floating-point format and operation are set according to the floating-point representation and operation sent.
    - SIMD bit is set to one.
    - Number of SIMD operations bits are set according to the required number of operations.
4. Wait until status bit is set to one by reading the command register in a loop.

5. Reset status bit to zero by writing to the command register the same value read fom it (to not affect the flags as they'll be read later) but eith the clear bit set to one.
6. Read the four status registers then extract from them the flags of each operarion and insert them in the array of flags.
7. Read outputs in a loop and insert them in the array of outputs.

```
# RX string: FPU

# RX string: Decimal addition Output_1 = 2644d6cb

# RX string: Invalid operation flag= 0

# RX string: Division by zero flag = 0

# RX string: Overflow flag = 0

# RX string: Underflow flag = 0

# RX string: Inexact flag = 1

# RX string: Decimal addition Output_2 = 26105f54

# RX string: Invalid operation flag= 0

# RX string: Division by zero flag = 0

# RX string: Overflow flag = 0

# RX string: Underflow flag = 0

# RX string: Inexact flag = 0

# RX string: Decimal addition Output_3 = 2644d6cb
```

Figure 61: FPU_SIMD_Instruction output

4.6.1.3 Compare

```
bool compare(int fpu_output,int reference_output,int fpu_flags,int reference_flags)
```

Figure 62: compare header

Function Inputs:

- FPU output.
- FPU flags.
- Reference output.
- Reference flags.

Function Outputs:

- Boolean.

Function code:

The output is true if

1. FPU output is equal to Reference output.
2. All FPU flags are equal to Reference flags.

Else it's false

4.6.1.4 FPU get status

```
int FPU_get_status()
```

Figure 63:FPU_get_status header

This function reads the status register and resurns it's value, it's called in the FPU_Single_Instruction and th FPU_SIMD_Insruction functions.

4.6.2 FPU testing

In order to test the integration of the FPU with the RISC-V core as well as the C functions developed, a set of data for different operands, operations, representations and the expected outputs and flags was used, the cases shown in Figure 64 were caried out.

The cases are divided into 2 groups:

1. Cases to test the single instruction function which are futher divided into two sub-groups:
    - One to test that the representation and operation are correctly decoded and this was carried out by inserting the same opeands to all the possible combinations of the representation and operation (the designed FPU have 6 different cmbinations) as shown in Figure 65, where each of those combinations have a different expected output so by comparing the expected outputs with the generated ones, we can guarantee that the representation and operation are decoded correctly.
    - The other to test that the flags are extracted from the status register and read correctly, four cases were tested one for each case as shown in Figure 66, in the cases of overflow and underflow two flags are risen in each case, the flag representing either overflow or underflow in addition to the inexact flag.
2. Cases to test the SIMD instruction function, due to the large number of possibilities of this instruction and the difficulty of covering all it's cases by designing specific test cases, testing of this instruction was done using 16 different operands for each of the 6 different combinations of the operation and representation, then the number of SIMD operations for each case was looped on to cover all it's pssible cases, at the first iteration the first operands are taken, then in the second iteration the first and the second operands are taken, then in the third one the first three are taken and so on until the last operation where all the sixteen operands are taken as shown in Figure 67, also the used operands with each operation and representation were chosen and disttibuted in a way so that different flags are risen in different locations of the SIMD array with each of the 6 different combinations to check the flags' decoding.

The outputs of all test cases were zero errors as shown in Figure 68

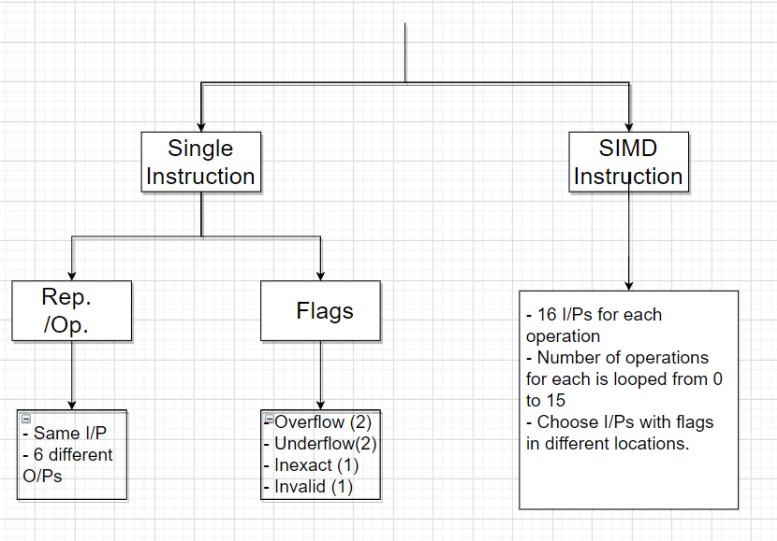

Figure 64: Integration test cases

```
#define num 6
int main()
{
  printf("FPU\n");


int operand_A [num]      = {0x267692e7,0x267692e7,0x267692e7,0x267692e7,0x267692e7,0x267692e7};
int operand_B [num]      = {0x267492e7,0x267492e7,0x267492e7,0x267492e7,0x267492e7,0x267492e7};
int fpu_output[num];
int fpu_flags [num];
int ref_output[num]      = {0x26f592e7,0x23000000,0x0d6b9182,0x2a7b24b4,0x22720000,0x26fc8821};
int ref_flags [num]      = {0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000};
char op       [num]      = {'+','-','*','+','-','*'};
char rep      [num] [20] = {"Single precision","Single precision","Single precision","Decimal","Decimal","Decimal"};
bool comp [num];
int i;
int n = 0;
int flags;
for ( i=0 ; i < num ; i++){
        fpu_output[i]= FPU_Single_Instruction(operand_A[i],operand_B[i],op[i],rep[i],&flags);
        fpu_flags[i] = flags;
        comp[i] = compare(fpu_output[i],ref_output[i],flags,ref_flags[i]);
        }
for ( i=0 ; i < num ; i++){
        if (comp[i] == false){
                printf("Error in = %d\n",i);
                printf("Expected O/P = %x\n",ref_output[i]);
                printf("O/P = %x\n",fpu_output[i]);
                printf("Expected flags = %x\n",ref_flags[i]);
                printf("Flags = %x\n",fpu_flags[i]);
                n = n + 1;
                }
        }
printf("Number of errors = %d\n",n);
```

Figure 65: Single instruction operation/representation test case

```
int operand_A [num]      = {0x00000996,0x7f4658d6,0x52c54628,0xffffffff};
int operand_B [num]      = {0x006b25f4,0x6d0f1c41,0x005e220c,0xffffffff};
int fpu_output[num];
int fpu_flags [num];
int ref_output[num]      = {0x00000000,0x7f800000,0x139113ff,0xffffffff};
int ref_flags [num]      = {0x00000018,0x00000014,0x00000010,0x00000015};
char op       [num]      = {'*','*','*','*'};
char rep      [num] [20]= {"Single precision","Single precision","Single precision","Single precision"};
bool comp [num];
int i;
int n = 0;
int flags;

for ( i=0 ; i < num ; i++){
        fpu_output[i]= FPU_Single_Instruction(operand_A[i],operand_B[i],op[i],rep[i],&flags);
        fpu_flags[i] = flags;
        comp[i] = compare(fpu_output[i],ref_output[i],flags,ref_flags[i]);
        }
for ( i=0 ; i < num ; i++){
        if (comp[i] == false){
                printf("Error in = %d\n",i);
                printf("Expected O/P = %x\n",ref_output[i]);
                printf("O/P = %x\n",fpu_output[i]);
                printf("Expected flags = %x\n",ref_flags[i]);
                printf("Flags = %x\n",fpu_flags[i]);
                n = n + 1;
                }
        }
printf("Number of errors = %d\n",n);
```

Figure 66: Single instruction flags test case

```
for ( j=0 ; j < 6 ; j++){
        n = 0
        for ( i=0 ; i < 16 ; i++){
                fpu_output= FPU_SIMD_Instruction(operand_A[j],operand_B[j],op[j],rep[j],(i+1),fpu_flags);
                for ( k=0 ; k < (i+1)  ; k++){
                        comp = compare(fpu_output[k],ref_output[j][k],fpu_flags[k],ref_flags[j][k]);
                        if (comp == false){
                                printf("Error in = [%d][%d][%d]\n",j,i,k);
                                printf("Expected O/P = %x\n",ref_output[j][k]);
                                printf("O/P = %x\n",fpu_output[k]);
                                printf("Expected flags = %x\n",ref_flags[j][k]);
                                printf("Flags = %x\n",fpu_flags[k]);
                                n = n + 1;
                                }
                        }

                }
        printf("Number of errors = %d\n",n);
}
```

Figure 67: SIMD instruction test case

```
# RX string: FPU

# RX string: Number of errors = 0
```

Figure 68: Test cases output

CHAPTER FIVE: VERIFICATION

In this chapter we're going to discuss the verification phase in this project, we're required to build a testing environment to perform functional verification on the RTL code, firstly we built separate testing environments for each combinational module then an environment to test the integrated floating point unit modules with the host controller interface (HCI)

5.1 DESIGN UNDER TEST SPECIFICATIONS

The designed FPU performs 3 operations (Addition, subtraction and multiplication) on two different 32 bits representations (single precision and decimal format decimal encoding, also the designed FPU supports single instruction multiple data (SIMD) operations so it can perform the same operation on different operands up to 16 operands

The output of the FPU has 32-bit result with the same representation of the two input operands and 4-bits flags where the four flags are:

- Invalid operation: raised when the input operation is not one of the three specified operations.
- Overflow flag: raised when the result is greater than the maximum representable number.
- Underflow flag: raised when the result is smaller than the maximum representable number.
- Inexact flag: raised when the result is rounded up.

5.2 WORK FLOW

We started by studying systemVerilog language for verification from the reference ("SystemVerilog for verification A guide to learning the testbench language features third edition") and the UVM basics from ("The UVM Primer An Introduction to the Universal Verification Methodology by Ray Salemi"). (7) (8)

We have built testing environment for each combinational module then we built the integrated environment to test the integrated modules with the (HCI).

The three testing environments for the decimal representation are built using UVM transactions & UVM_TLM ports, while the single precision and the integrated environments built using sequence_item and UVM_sequencer.

5.3 VERIFICATION ENVIRONMENTS

All the environments are built using UVM methodology which have more features than the OOP environments and make the testing environment more reusable and editable

The UVM allows us to use

- Dynamically-generated objects that allow you to specify tests and test bench architecture without recompiling
- A hierarchical testbench organization that includes Drivers, Monitors, and Bus Functional Models
- Transaction-level communication between objects
- Testbench stimulus (UVM Sequences) separated from the testbench structure

5.3.1 Decimal representation testing environment

It's a UVM transaction based environment

5.3.1.1 Environment architecture

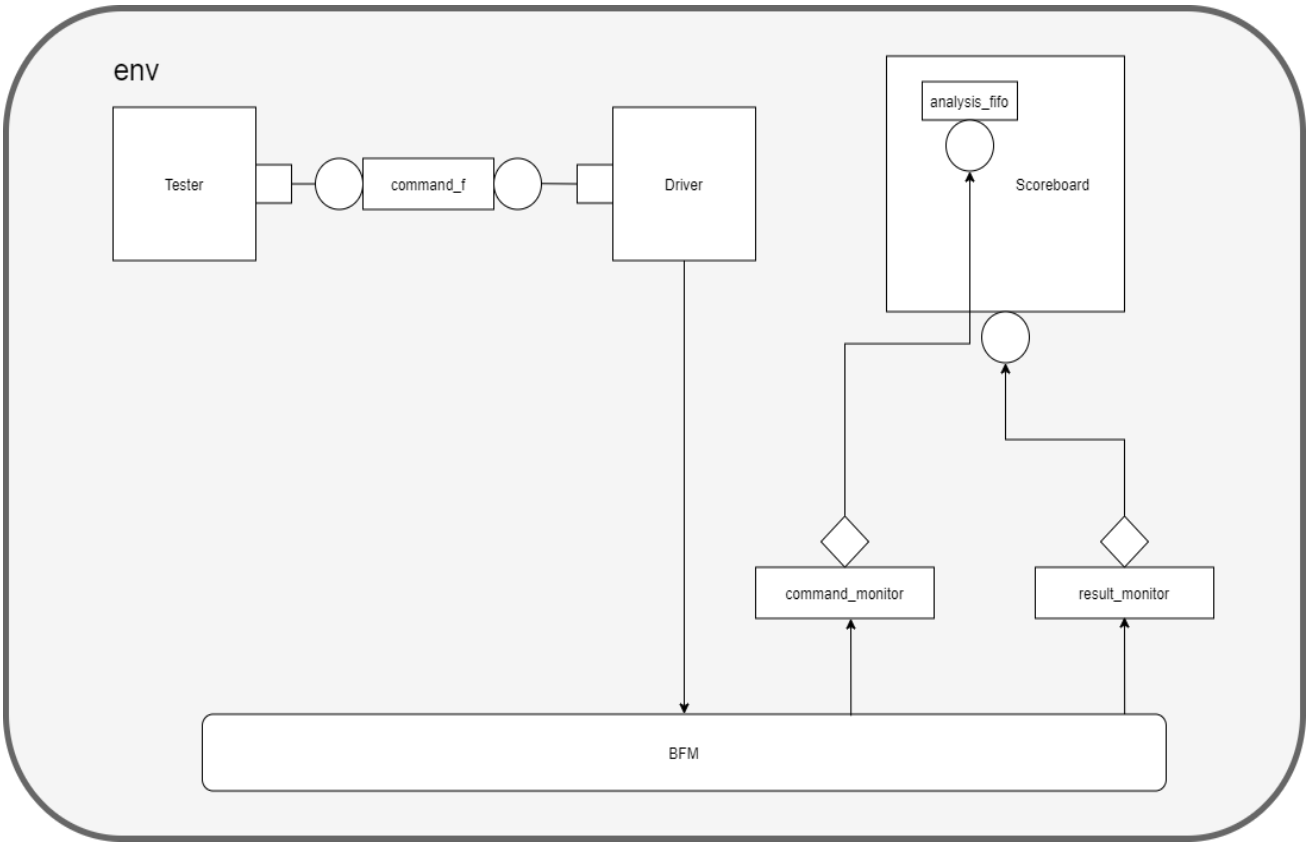

Figure 69: Decimal representation testing environment

5.3.1.2 Environment's transaction

The transaction is a UVM object that extends the UVM_transaction, Transactions encapsulate both data and all the operations we can do to that data, In our environment we have two transactions which are the command transaction and the result transaction

5.3.1.2.1 Command transaction

The command transaction class contains four members which are:

- 32-bit (a_rep) for operand(1) represented in the decimal format
- 32-bit (b_rep) for operand(2) represented in the decimal format
- Real (a_dec) for operand(1) in real format
- Real (b_dec) for operand(2) in real format

And it contains two functions which are ("random") and ("dec"):

Function ("random")

```
14  function void random ();
15    rand_num num ;
16    num = new();
17    num.randomize();
18    a_rep = represent(num);
19    num.randomize();
20    b_rep = represent(num);
21  endfunction
```

Figure 70: DE function ("random")

This function is used to randomize the two operands of the transaction, firstly we create an object of the class "rand_num ", this class contains 3 data members:

- Sign bit which represents the sign of the operand (0= positive number, 1=negative number)
- Unsigned integer ("C") which represent combination field of the number
- Integer ("exp") which represents the exponent of the number

And one function member ("gen_num"), this function return a real number from the class members by using the formula shown in Figure 71

```
15  function real gen_num();
16    return ((-1.0)**$itor(sign))*$itor(c)*(10.0**$itor(exp));
17  endfunction
```

Figure 71: DE function ("gen_num")

Then these class members are randomized with certain constraints that will be discussed later, after the randomization in order to store the randomized operands in 32-bit decimal format we used the function ("represent") which takes an object of the class "rand_num" as an argument and return 32-bits in the decimal format, this process is repeated to generate the second operand.

Function ("dec")

```
23  function void dec ();
24    rand_num num;
25    num = new();
26    num = decode(a_rep);
27    a_dec = num.gen_num();
28    num = decode(b_rep);
29    b_dec = num.gen_num();
30  endfunction
```

Figure 72: DE function ("dec")

This function is used to store the two randomized numbers in the real format in the two transaction members ("a_dec") and ("b_dec") using the function ("decode") which takes the 32-bit represented number in decimal format and return the corresponding real value using the member function ("gen_num") of class ("rand_num").

5.3.1.2.2 Result transaction

This transaction is used to store the result and then transmits it from the result monitor to the scoreboard and its data members are:

- 32-bit ("Result") which represents the result in the decimal format
- Real ("result_dec") which represents the corresponding real value of the result
- 4-bit ("flags") which represents the result flags

And a function ("dec") which takes the 32-bit represented result as an argument and return the corresponding real value.

5.3.1.3 Environment's components description

5.3.1.3.1 Tester

The tester block is responsible for generating the test cases which will propagate through the environment, the test cases are generated using constrained randomization in the form of transactions, the tester has a UVM_put_port which takes the data of transaction type to deliver the test cases to the driver.

The tester generates the UVM command transactions then it calls the function (".random") to randomize the transaction components, then the transaction is decoded using the function (".dec") to have the random operands in real form which is needed to perform further operations, then the transaction is put in the tester port then this process is repeated to generate another test case and so on.

```
19      repeat (200000) begin
20         command = command_transaction::type_id::create("command");
21         command.random();
22         command.dec();
23         command_port.put(command);
24       end
```

Figure 73: DE generating random transactions

5.3.1.3.2 Command_f

It's an UVM analysis FIFO of type transaction which delivers the test cases transactions from the tester to the driver, the FIFO takes the transaction through its put port in the tester then it blocks the tester from putting new transactions in the FIFO till the first added transaction is get by the driver

5.3.1.3.3 Driver

The driver block extends UVM_component and has a UVM_get_port, also we instantiate a virtual BFM in the driver so it can communicate with the DUT through the BFM

The UVM_get_port is used to get the test cases transactions from the tester through the FIFO then it calls a built task (".send_op") which takes the two operands represented in the 32-bit decimal format to be sent to the DUT through the BFM, the other task is (".write_to_monitor") which takes two arguments which are the two operands in the real format to be sent to the command monitor through the BFM

The run phase of the driver is inside a forever loop however the test ends when the loop used in the tester to generate the test cases finishes because the phase objection is dropped after generating the test cases and it was the only raised objection in the whole testing environment.

```
18    task run_phase(uvm_phase phase);
19       command_transaction    command;
20       forever begin : command_loop
21         command_port.get(command);
22         bfm.send_op(command.a_rep, command.b_rep);
23         bfm.write_to_monitor(command.a_dec , command.b_dec);
24       end : command_loop
25    endtask : run_phase
```

Figure 74: DE driver run phase

5.3.1.3.4 Bus functional model (BFM)

The BFM has two interfaces. On one side is a functional interface that accepts transactions and on the other side is a pin interface that is connected to the DUT. The functionality of the BFM is to bridge those two interfaces

In our BFM we have some data members which represent the inputs and outputs of the design under test:

- 32-bit ("operand1") models the represented first operand in the decimal format
- 32-bit ("operand2") models the represented second operand in the decimal format
- 32-bit ("Result") models the output of the DUT in the decimal format
- 4-bit ("Flags") models the result flags of the DUT

It also has instances of the command monitor and the result monitor which are used to send the two operands and the DUT outputs to the scoreboard, and it has two member tasks ("send_op") and ("write_to_monitor").

Task ("send_op")

This task is called in the driver and takes two arguments which are the two operands in the decimal format representation then they are assigned to the BFM members ("operand1") and ("operand2") which are connected to the DUT, the task has a delay of 10 ns to model the propagation delay of the operands through the combinational DUT

```
9   task send_op (bit [31:0] a_rep , bit [31:0] b_rep);
10    operand1 = a_rep;
11    operand2 = b_rep;
12    #10;
13  endtask
```

Figure 75: DE task ("send_op")

Task ("write_to_monitor")

This task is called in the driver and takes two arguments which are the two operands in the real format and then they are passed to the command monitor by calling the member function of the command monitor instance ("write_to_monitor"), the output of the DUT which are the 32-bit represented result and the 4-bit flags are passed to the result monitor also by calling the member function of the result monitor instance ("write_to_monitor").

```
18  task write_to_monitor (real a_dec , real b_dec);
19    command_monitor_h.write_to_monitor(a_dec,b_dec);
20    result_monitor_h.write_to_monitor(Result,Flags);
21  endtask
```

Figure 76: DE BFM task("write_to_monitor")

5.3.1.3.5 Command monitor

The command monitor block extends the UVM _component, it has a virtual instance of the BFM and has an analysis port of type ("command _transaction") called ("cm_port") which is used to send the two operands to the scoreboard, In the build phase, this command monitor is connected to the command monitor instance in the BFM

The command monitor has a member function ("write_to_monitor") (which is called in the BFM), the function has two arguments of type real ("a_dec") and ("b_dec"), firstly we create an object of ("command_transaction") then we assign the two arguments to the corresponding members of the command transaction, then the transaction is sent through the port to the scoreboard.

```
20  function void write_to_monitor(real a_dec, real b_dec);
21     command_transaction cmd;
22     cmd = new("cmd");
23     cmd.a_dec = a_dec;
24     cmd.b_dec = b_dec;
25     cm_port.write(cmd);
26   endfunction : write_to_monitor
```

Figure 77: DE command monitor function ("write_to_monitor")

5.3.1.3.6 Result monitor

The result monitor block extends the UVM _component, it has a virtual instance of the BFM and has an analysis port of type (“result _transaction”) called (“ap_port”) which is used to send the result and flags to the scoreboard, In the build phase, this result monitor is connected to the result monitor instance in the BFM

The result monitor has a member function (“write_to_monitor”) (which is called in the BFM), the function has two arguments 32-bit (“Result”) and 4-bit (“Flags”), firstly we create an object of (“result_transaction”) then we assign the two arguments to the corresponding members of the result transaction, we call the member function of the result transaction (“dec”) to put the real format of the result in the member of the result transaction (“result_dec”), then the transaction is sent through the port to the scoreboard.

```
19  function void write_to_monitor(bit [31:0] Result , bit [3:0] flags);
20    result_transaction result_t;
21    result_t = new("result_t");
22    result_t.Result = Result ;
23    result_t.flags = flags ;
24    result_t.dec();
25    ap.write(result_t);
26   endfunction : write_to_monitor
```

Figure 78: DE result monitor function ("write_to_monitor")

5.3.1.3.7 Scoreboard

UVM scoreboard is a verification component that contains checkers and verifies the functionality of a design. It usually receives transaction level objects captured from the interfaces of a DUT via TLM Analysis Ports.

In our environment the scoreboard extends uvm_subscriber for the result monitor to be connected to the analysis port of the result monitor and receive the result transaction, it has a UVM_tlm_analysis_fifo of type command transaction which is connected to the command monitor analysis port, it has two member functions (“predict_result”) and (“write”)

Function (“predict_result”)

This function has one argument of type command transaction and return result transaction with the predicted result and flags.

Firstly we create an object of result transaction then we store the predicted real format value of the result in (“result_dec”) member of the result transaction by carrying out the required operation on the two operands in real format (“a_dec”) and (“b_dec”) which are members of the command transaction passed through the argument

The operation applied on the two operands is changed according to the design under test so that the same environment can be used for the addition, subtraction and multiplication designs for decimal format representation just by changing the operator in the scoreboard.

```
15 function result_transaction predict_result(command_transaction cmd);
16    result_transaction predicted;
17    predicted = new("predicted");
18    predicted.result_dec = cmd.a_dec + cmd.b_dec;
19    return predicted;
20 endfunction : predict_result
```

Figure 79: DE function ("predict_result") for decimal addition

Function ("write")

This function is called automatically when the result monitor write the data in its analysis port and it takes one argument of type result transaction ("t") which have the same data written by the result monitor in the port.

We create two objects of type command transaction and result transaction ("cmd"), ("predicted") then we use the function ("try_get") to get the data from the FIFO and store it in the command transaction ("cmd"), this transaction is passed as an argument to the function ("predict_result") and return a result transaction which is the predicted result for the given operands, then the predicted flags are calculated as will be discussed to be compared with the DUT flags.

To calculate the overflow flag, the predicted result is compared to the maximum representable numbers (positive or negative), if the result is greater than the maximum positive number or smaller than the maximum negative number the overflow flag and the inexact flag are raised, then the predicted flags are compared with the DUT flags and decide whether the test case pass or fail, without comparing the result as they're not checked in case of overflow.

```
154      if(predicted.result_dec > 9999999e90 || predicted.result_dec < -9999999e90)
155        begin
156          predicted.flags[2] = 1 ;
157          predicted.flags[0] = 1 ;
158          if( t.flags != predicted.flags)
159            begin
160              `uvm_error("SELF CHECKER", "FAIL" );
161              $display ("OVERFLOW case")  ;
162              $display("predicted flags = %b " , predicted.flags) ;
163              $display ("dut flags = %b" , t.flags) ;
164            end
165          else
166            `uvm_info ("SELF CHECKER", {"PASS"}, UVM_HIGH)
167          return;
168        end
```

Figure 80: DE overflow condition

The minimum representable number is (1×10^{-101}) so that if the absolute result is less than that number, the number will be not representable and the underflow flag will be raised, but if the exponent is between -95 and -101 it may be underflow or not according to the precision of the number for example:

If the exponent is -96, and the result is 1.23456×10^{-96} so the number will be representable in the form 123456×10^{-101} but if the number is $1.234567 \times 10^{-96} = 1234567 \times 10^{-102}$ which is an underflow case, special cases is made for each exponent between -95 and -101 to calculate the precision of the result and decide if it's underflow or not.

```
45      if(predicted.result_dec < 1e-95 && predicted.result_dec > -1e-95
46        && predicted.result_dec != 0 )
47        begin
48          predicted_result_uf = predicted.result_dec;
49          exp_uf = 0 ;
50          while(predicted_result_uf < 1)
51            begin
52              predicted_result_uf = predicted_result_uf*10.0;
53              exp_uf = exp_uf-1;
54            end
55          if(exp_uf == -96)
56            begin
57              predicted_result_uf = predicted_result_uf*1e5;
58              if(!((predicted_result_uf-$rtoi(predicted_result_uf) > 0.99999999
&&
59                  predicted_result_uf-$rtoi(predicted_result_uf) < 1)
60                  ||
61                  (predicted_result_uf-$rtoi(predicted_result_uf) < 0.00000001 &&
62                   predicted_result_uf-$rtoi(predicted_result_uf) >= 0) ))
63                begin
64                  predicted.flags[1] = 1 ;
65                  predicted.flags[0] = 1 ;
66                end
67            end
```

Figure 81: DE underflow condition and special case

The inexact flag is risen when there is rounding up in the last digit of the result (the 7th digit) or if it's an underflow or overflow case, this approximation is done according to rounding digit which is the 9th digit of the result, where if this digit is greater than or equal 5 the result is rounded up otherwise no rounding occur and this is done by the algorithm shown in Figure 82

```
177       predicted_result_2 = predicted.result_dec;
178       if(predicted_result_2 < 0 )
179         predicted_result_2 = -1.0 * predicted_result_2 ;
180       if(predicted_result_2 > 1 )
181         begin
182           while(predicted_result_2 > 10)
183             predicted_result_2 = predicted_result_2/10.0 ;
184         end
185       else
186         begin
187           while(predicted_result_2 < 1)
188             predicted_result_2 = predicted_result_2*10.0 ;
189         end
190           predicted_result_2 = predicted_result_2*1e7;
191           diff_2 = predicted_result_2 - $itor(int'(predicted_result_2)) ;
192
```

Figure 82: DE rounding according to 9th digit

There is a special case in the addition and subtraction operations where if the exponent difference between the two operands is 14 or more so the result will be the larger operand and the inexact flag will never be raised but this is not applied for the multiplication case

The result is compared by calculating the difference between the predicted result and the result from the DUT, if this difference is more than a certain threshold (stated due to that the operation done on real type in the testing environment has more precision of the decimal format

representation) or the predicted flags aren't equal the DUT flags the test case is considered a failure.

```
249 ////////////////////////////////////result check
250       if(t.result_dec > predicted.result_dec)
251         diff = t.result_dec - predicted.result_dec ;
252       else
253         diff = predicted.result_dec - t.result_dec ;
254
255       result_num = decode(t.Result);
256
257       if (diff > (10.0**$itor(result_num.exp)) || t.flags != predicted.flags)
258         begin
259           `uvm_error("SELF CHECKER", "FAIL" )
260         end
261       else
262         `uvm_info ("SELF CHECKER", {"PASS"}, UVM_HIGH)
```

Figure 83: DE result check algorithm

5.3.1.3.8 Class ("env")

The env class extends UVM_env, this class has instances of all the environments components discussed above which are created in its build phase, then in the connect phase these components are connected together where the driver and the tester are connected to the ports of the FIFO ("command_f"), also the command and result monitors are connected to the ports of the scoreboard, this class is instantiated in ("random_test") class which extends UVM_test.

```
27    function void connect_phase(uvm_phase phase);
28      driver_h.command_port.connect(command_f.get_export);
29      tester_h.command_port.connect(command_f.put_export);
30      command_monitor_h.cm_port.connect(scoreboard_h.cmd_f.analysis_export);
31      result_monitor_h.ap.connect(scoreboard_h.analysis_export);
32    endfunction : connect_phase
```

Figure 84: DE env ("connect_phase") function

5.3.1.3.9 Top module

In the top module ("env_pkg") which is a package that include all the environment components flies, also import the ("UVM_pkg"), then the BFM and the DUT are instantiated and connected, the BFM is given to ("UVM_config_db") to be instantiated easier in the other components, then ("run_test") function is called with the argument the name of the UVM_test file ("random_test") to start the test.

```
module top;

   import uvm_pkg::*;
  `include "uvm_macros.svh"
   import env_pkg::*;

  bfm  bfm();

  decimal_adder DUT (.operand1(bfm.operand1), .operand2(bfm.operand2),
                     .Result(bfm.Result),
                     .Flags(bfm.Flags));

  initial begin
    uvm_config_db #(virtual bfm)::set(null, "*", "bfm", bfm);
    run_test("random_test");
  end


endmodule
```

Figure 85: DE top module

5.3.2 Single precision representation testing environment

It's a UVM sequence based environment, this environment has some improvements on the transaction based environment to make it more reusable as we separate the test case generation from the environment structure so we could run different sequences on the same environment.

5.3.2.1 Environment's architecture

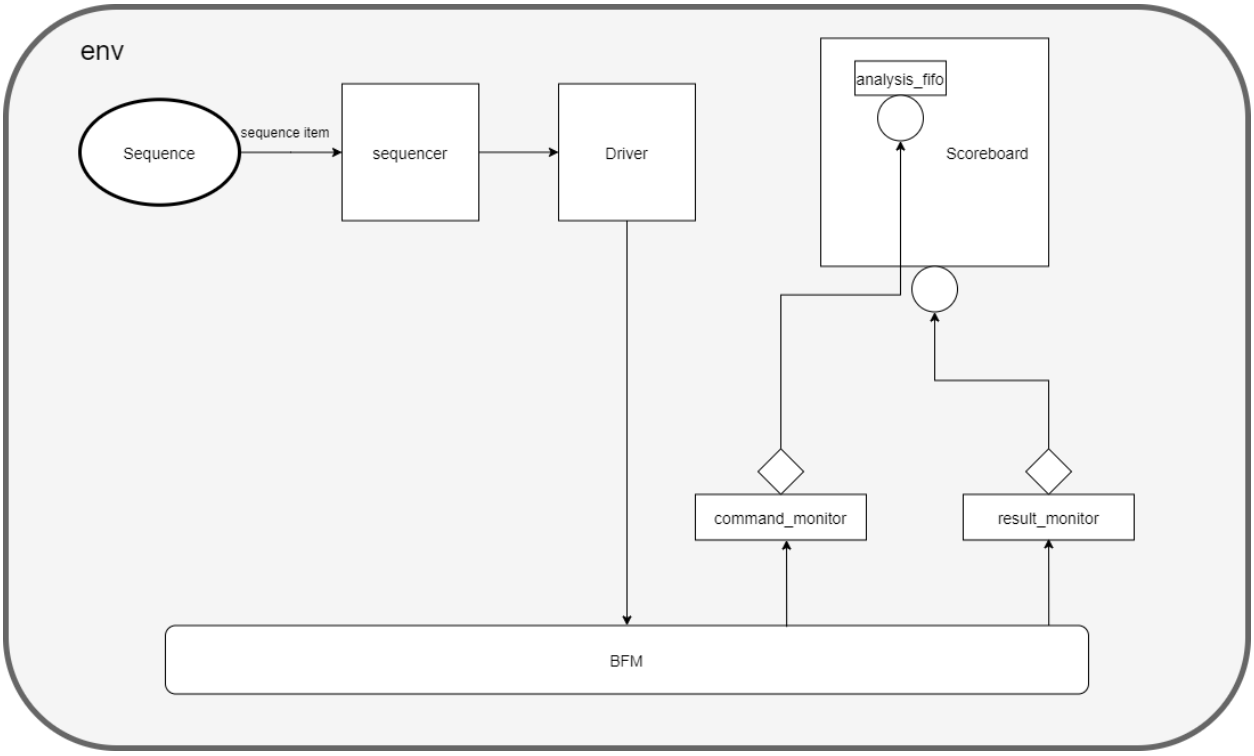

Figure 86: Single precision representation testing environment

5.3.2.2 Environment's sequence item

The sequence item class contains six members which are the fields of the single precision representation for each operand:

- 23-bit ("mantissa_1") represents the significand of operand(1)
- 8-bit ("exp_1") represents the exponent of operand(1)
- bit ("sign_1") represents the sign bit of operand(1)
- 23-bit ("mantissa_2") represents the significand of operand(2)
- 8-bit ("exp_2") represents the exponent of operand(2)
- bit ("sign_2") represents the sign bit of operand(2)

These members will be randomized with certain constraints which will be discussed later.

5.3.2.3 Environment's components description

The top module, BFM, command monitor and result monitor are almost the same as the UVM transaction based environment previously discussed.

5.3.2.3.1 Sequence

In our environment this class is called ("random_sequence") which extends UVM_sequence of type ("sequence_item"), this sequence contains the testing scenario by creating a sequence item object and randomize it then send it to the sequencer using the functions ("start_item") and ("finish_item").

```
10    task body();
11      repeat (200000) begin : random_loop
12         command = sequence_item::type_id::create("command") ;
13         start_item(command)                                 ;
14         command.randomize()                                 ;
15         finish_item(command)                                ;
16       end : random_loop
17    endtask : body
```

Figure 87: SP task ("body") of ("random_sequence")

5.3.2.3.2 Sequencer

The sequencer is automatically deliver the sequence_item from the sequence to the driver, it has no special functions, so it's defined in the ("env_pkg") using ("typedef") and will be instantiated in the ("env") class, the sequencer has a built in port which will be connected to the driver.

5.3.2.3.3 Driver

The driver class extends UVM_driver of type ("sequence_item") the UVM_driver has a built in port called ("seq_item_port") which will be connected to the sequencer, the driver create an object of type sequence item to store the data generated by using the function ("get_next_item"), then sends this data to the BFM using function ("send_op") and call the

function (“write_to_monitore”) which have been discussed previously, the function (“item_done”) is called after sending the data which declares that the driver is ready to get another sequence item.

```
11    task run_phase(uvm_phase phase);
12       sequence_item cmd;
13       forever begin : cmd_loop
14         seq_item_port.get_next_item(cmd) ;
15         bfm.send_op({cmd.sign_1,cmd.exp_1,cmd.mantissa_1} ,
16                     {cmd.sign_2,cmd.exp_2,cmd.mantissa_2});
17         bfm.write_to_monitor ();
18         seq_item_port.item_done()        ;
19       end : cmd_loop
20    endtask : run_phase
```

Figure 88: SP driver (“run_phase”)

5.3.2.3.4 Scoreboard

The scoreboard extends UVM_subscriber of type (“result_transaction”) which is the same class in the decimal encoding environment previously discussed, it has a (“UVM_tlm_analysis_fifo”) of type sequence item to get the command data from the command monitor, it also has two function members (“write”) and (“predict_result”).

Function (“predict_result”)

It has one argument of type sequence_item , a result_transaction object is created to store the predicted result and flags, each operand is converted to short real type using the built in function (“$bitstoshortreal”) then the operation is carried out and the result is converted back to the single precision representation using the built in function (“$shortrealtobits”), we are using the type short real as it’s stored as bits in the form of single precision representation, so it’s easier to switch between the two formats

The same function is used for all the operations by changing only the operator.

```
14  function result_transaction predict_result(sequence_item cmd);
15    result_transaction predicted;
16    predicted = new("predicted") ;
17    predicted.result = $shortrealtobits(
18      $bitstoshortreal({cmd.sign_1, cmd.exp_1, cmd.mantissa_1})-
19      $bitstoshortreal({cmd.sign_2, cmd.exp_2, cmd.mantissa_2}));
20    return predicted ;
21  endfunction
```

Figure 89: SP function ("predict_result")

Function (“write”)

In this function the predicted flags are calculated then the DUT result and flags are compared to the predicted to decide whether the test case pass or fail

Since the result is stored in short real type which have the same ranges as the single precision representation, in the overflow case the result will be infinity which will be represented

as all ones in the exponent field and all zeros in the mantissa field (“32’b11111111000000000000000000000000”)

```
40      if(predicted.result[30:0] == 31'b11111111_00000000_00000000_0000000)
41         begin
42           predicted.flags[2] = 1 ;
43           predicted.flags[0] = 1 ;
44           if( t.flags != predicted.flags)
45              `uvm_error("SELF CHECKER", "FAIL" )
46           else
47               `uvm_info ("SELF CHECKER", {"PASS"}, UVM_LOW)
48           return;
49        end
```

Figure 90: SP predicted overflow flag

In the underflow case the result is all zeros.

```
51      if(predicted.result[30:0] == 31'b0 )
52         begin
53           predicted.flags[1] = 1 ;
54           predicted.flags[0] = 1 ;
55           if( t.flags != predicted.flags)
56              `uvm_error("SELF CHECKER", "FAIL" )
57           else
58               `uvm_info ("SELF CHECKER", {"PASS"}, UVM_LOW)
59           return;
60        end
```

Figure 91: SP predicted underflow flag

To calculate the predicted inexact flag, the operation to get the predicted result is carried out again but with storing the result in a real data type so that it has a higher precision, then to decide if rounding occurred or not we get the difference between the mantissa of the predicted result stored in short real and that stored in real.

In the case of the subnormal number the mantissa of the higher precision needs to be normalized so that it can be subtracted from the single precision mantissa this normalization is done by adding the bias of the higher precision (1024) to the exponent of the higher precision and subtraction the bias of the lower precision format (128), then the mantissa of the higher precision format is shifted right by the result of the previously described operation then the mantissa is ready to be subtracted from the single precision mantissa

```
61
62      if(predicted.result[22:0]-predicted_2[51:29] == 1'b1)
63       predicted.flags[0] = 1 ;
64     if(predicted.result[30:23] == 8'b0)
65       begin
66         counter = 896 - predicted_2[62:52] ;
67         predicted_3 = predicted.result ;
68         predicted_3[23-counter-1] = 0 ;
69         predicted_2[51:0] = predicted_2[51:0] >> counter+10 ;
70         if ( predicted_3 - predicted_2[51:20] == 1'b1)
71           predicted.flags[0] = 1 ;
72       end
```

Figure 92: SP predicted inexact flag

5.3.2.3.5 Class ("env")

In this class all the environment components are instantiated, in its connect phase the driver is connected directly to the sequencer and the command and result monitor are connected to the scoreboard as shown in Figure 93

```
25  function void connect_phase (uvm_phase phase);
26    driver_h.seq_item_port.connect(sequencer_h.seq_item_export) ;
27    command_monitor_h.ap.connect (scoreboard_h.cmd_f.analysis_export);
28    result_monitor_h.ap.connect ( scoreboard_h.analysis_export);
29  endfunction
30
```

Figure 93: SP env connect phase

5.3.2.3.6 Class ("base_test")

It's the base test which extends the UVM_test and then any other test with its sequence will extends this base test, in this base test an object from the ("env") and the sequencer classes are created then this sequencer object is connected to the sequencer inside the env object

```
2  class base_test extends uvm_test;
3
4    env       env_h       ;
5    sequencer sequencer_h ;
6
7    function new (string name , uvm_component parent ) ;
8      super.new (name , parent) ;
9    endfunction
10
11   function void build_phase (uvm_phase phase);
12     env_h       = new("env_h" , this)       ;
13   endfunction
14
15   function void end_of_elaboration_phase (uvm_phase phase) ;
16     sequencer_h = env_h.sequencer_h ;
17   endfunction
18
19 endclass
```

Figure 94: SP class base_teste

5.3.2.3.7 Class ("random_test")

This class extends ("base_test") class ,firstly it creates an object from the ("sequence") class then the built_in function start after raise the uvm_phase objection this function takes one argument which is the sequencer object then the objection is dropped after the test scenario is done.

```
class random_test extends base_test ;
  `uvm_component_utils (random_test)

  function new (string name , uvm_component parent);
    super.new( name , parent );
  endfunction

  task run_phase (uvm_phase phase);
    random_sequence random ;
    random     = new("random");
    phase.raise_objection(this)          ;
    random.start(sequencer_h)            ;
    phase.drop_objection(this)           ;
  endtask

endclass
```

Figure 95: SP ("random_test") class

5.3.3 The integrated environment

The integrated environment tests the integrated sequential module which have the FPU (all 6 modules) and the HCI, it also tests the SIMD (Single Instruction Multiple Data) which performs the same operation on array of data up to 16 entries, the environment is a sequence_item based environment with the same structure as the single precision environment previously described

5.3.3.1 Environment sequence_item

The sequence item of the integrated environment has combined data member from both single precision sequence item and decimal encoding command transaction, all this members are in the form of arrays of 16 elements it also has control data members to decide the operation to be done, the representation and the number of the simd operations, the result transaction data members are also in the form of arrays.

```
7  ////////////////////// Single prescision ////////////////////////////
8    rand bit [22:0]  mantissa_sp_1 [16] ;
9    rand bit [7:0]   exp_sp_1      [16] ;
10   rand bit         sign_sp_1     [16] ;
11   rand bit [22:0]  mantissa_sp_2 [16] ;
12   rand bit [7:0]   exp_sp_2      [16] ;
13   rand bit         sign_sp_2     [16] ;
14
15 ////////////////////// Decimal encoding ////////////////////////////////////
16   real a_dec[16]                 ;
17   real b_dec[16]                 ;
18   rand int unsigned  c_de_1 [16] ;
19   rand int exp_de_1         [16] ;
20   rand bit sign_de_1        [16] ;
21   rand int unsigned  c_de_2 [16] ;
22   rand int exp_de_2         [16] ;
23   rand bit sign_de_2        [16] ;
```

Figure 96: integrated sequence_item data members

```
25 ////////////////////// Control signals ////////////////////////////////////
26
27   rand bit       rep          ; //1 ==> single precision   0 ==> decimal encoding
28   rand bit [1:0] op           ;
29   rand int       simd         ;
30   rand bit       interrupt_en ;
```

Figure 97: integrated sequence_item control signals

5.3.3.2 Environment's components description

The sequence, sequencer, command and result monitors are almost the same as the single precision environment described above

5.3.3.2.1 Class ("driver")

The driver class extends the UVM_driver class, it has an instance from the ("bfm"), in its run_phase an object of the sequence_item is created to store the test case which is read from the sequencer, then the test case is put in the right representation according to the ("rep") bit in the sequence item, this done in a for loop so that each operation is sent in the case of simd operations (using concatenation for single precision and function decode for the decimal encoding representation), then the function ("send_op") is called given the arguments which are the represented operands and the control signals, also the function ("write_to_monitor") is called with the same arguments

5.3.3.2.2 BFM

The BFM class is connected to the DUT whose input and output data members are shown in Figure 98

```
5   ////////// Inputs
6
7   bit        clk        ;
8   bit        reset_n    ;
9   bit [31:0] sw_address ;
10  bit        sw_read_en ;
11  bit        sw_write_en;
12  bit [31:0] sw_datain  ;
13  ////////// Outputs
14  bit [31:0] sw_dataout    ;
15  bit        fpu_interrupt ;
16
```

Figure 98: integrated environment BFM data members

The inputs and outputs are registered where we can give the address of the required register through ("sw_address") and write the data through ("sw_datain") or read the output from ("sw_dataout").
The BFM has 3 tasks which are ("reset"), ("send_op"), ("write_to monitor"), and an initial block for clock generation

Task ("send_op")

The operands is sent to the DUT following a certain procedure corresponding to the specifications of the HCI design, for synchronization we used the negative edge clock in the environment to write the data on the DUT, then the DUT is sampling the data at the next positive edge, by this method the testing environment is immune to the clock skew between the clock generation in the BFM and the DUT and monitoring the result the flow of writing is as follow:

1. The DUT is reset at the start of the testing sequence.
2. The operands are written to registers with reserved addresses.
3. Then the control data (which contains the operation, representation, number of simd operations and FPU enable) is written to the command register.

```
31  ////////////////////// Write operands
32
33      for(int i = 0 ; i <= simd ; i++ )
34        begin
35          sw_write_en = 1'b1      ;
36          sw_address = address_a ;
37          sw_datain = operand1[i];
38          @(negedge clk);
39          sw_address = address_b ;
40          sw_datain = operand2[i];
41          @(negedge clk);
42          address_a = address_a + 4 ;
43          address_b = address_b + 4 ;
44        end
```

Figure 99: writing operands to the BFM

Task ("write_to_monitor")

This task reads the results from the DUT and creates the result transaction that will be sent by the result monitor to the scoreboard and at the same time the command monitor send to the scoreboard the corresponding sequence item which includes the test case data

1. Firstly, we wait on the status bit (which is raised when the FPU finish the required operations)
2. Then the result is read from the output registers
3. Then the flags are read from the flag registers
4. Then the result is send to the result monitor and the command is sent to the command monitor using the function ("write_to_monitor")
5. Then the clear bit is raised which makes the DUT ready to accept a new test case and set the status bit to zero.

5.3.3.2.3 Scoreboard

The scoreboard extends the UVM_subcriber with the type ("result_transaction "), as the previous testing environments it has two tasks ("predict_result") and ("write")

Task ("predict_result")

The function ("predict_result") takes one argument of the type ("sequence_item") it is used to calculate the predicted result according to the representation and the operation of the test case:

- In the case of the decimal encoding format (rep bit is zero) the predicted result is calculated by performing the required operation on the ("a_dec"), ("b_dec") data members of the sequence item and the predicted result is stored in the ("result_dec") data member of the result transaction.

```
67      else ////////////////////// Decimal representation
68        begin
69          case (cmd.op)
70            2'b00 : ////// Addition (DE)
71              begin
72                for(int i = 0 ; i <= cmd.simd ; i++)
73                  begin
74                    predicted.result_dec[i] = cmd.a_dec[i] + cmd.b_dec[i];
75                  end
76              end
77
```

Figure 100: DE predicted result

- In the case of the single precision format (rep bit is one) the predicted result is calculated by performing the required operation on the operands after converting them by using the built-in function ("$bitstoshortreal") then the result is converted back to bits by using the built-in function ("$shortrealtobits")

```
19      if(cmd.rep) ////////////////////// Single Prescision
20        begin
21          case (cmd.op)
22            2'b00 : ////// Addition (SP)
23              begin
24                for(int i = 0 ; i <= cmd.simd ; i++)
25                  begin
26                    predicted.result[i] = $shortrealtobits(
27                          $bitstoshortreal({cmd.sign_sp_1[i], cmd.exp_sp_1[i],
28                                            cmd.mantissa_sp_1[i]}) +
29                          $bitstoshortreal({cmd.sign_sp_2[i], cmd.exp_sp_2[i],
30                                            cmd.mantissa_sp_2[i]})) ;
31                  end
32              end
```

Figure 101: SP predicted result

Task ("write")

Task write is responsible for reading the result transaction (which contains the result and flags calculated by the DUT) from the result monitor and calculate the predicted flags is calculated then the result and flags is compared to decide if the test case will pass or fail

The predicted flags are calculated according to the representation and operation bits in the sequence item that is read from the command monitor the flags are calculated inside a for loop that loops on the simd operation number, which perform the same logic as the previous environments according to the test case representation.

5.4 TESTING RANGES DISTRIBUTION

These ranges distribution is done by applying constraints on the data members to be randomized. The test is carried out with random seed for each run and each run generates about 200000 test case in combinational modules and about 20000 test case for the integrated DUT where SIMD is also randomized which can have upto 16 operation in each test case

5.4.1 Single precision representation

Single precision representation has 32 bits, 1 sign bit, 8-bit exponent, 23-bit mantissa, and it has 2 categories of numbers normal and subnormal numbers. When the 8 bits of the exponent are all zeros it means that the number is a subnormal number (its decimal exponent less than -38),

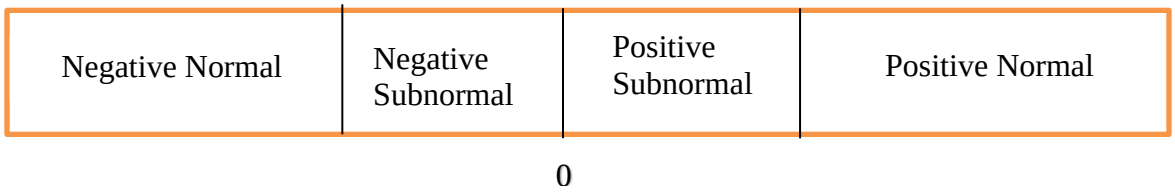

Figure 102: Single precision ranges

Since there are edges between positive and negative subnormal numbers (around zero) and between subnormal and normal numbers (in both positive and negative cases) and the boundaries of the range, so the corner cases are near these edges and boundaries, we distributed the weights of the range to:

1) Large positive normal
2) Small positive normal
3) Large positive subnormal
4) Small positive subnormal

Other than the ordinary cases of the range between these numbers, Same for the negative numbers.
The cases generated where the randomized operands are different combinations of these ranges have a higher probability to cross the boundaries.

```
37  constraint num {
38    foreach(exp_sp_1[i])
39    {exp_sp_1[i],mantissa_sp_1[i]}
40      dist
41    {
42      [31'h00000000 : 31'h00000fff] :/10 , // small subnormal
43      [31'h00001000 : 31'h004fEfff] :/10 , // ordinary subnormal
44      [31'h004ff000 : 31'h007fffff] :/15 , // large subnormal
45      [31'h00800000 : 31'h03ffffff] :/15 , // small normal
46      [31'h00400000 : 31'hff3fffff] :/40 , // ordinary normal
47      [31'hff400000 : 31'hff7fffff] :/10   // large normal
48    };
```

Figure 103: Single precision constraints

5.4.2 Decimal encoding representation

By the same criteria used in the single precision we divided the entire positive (Same for the negative) range into 5 parts:

1) Extremely large numbers (with max exponent of the range “90”)
2) Large numbers
3) Extremely small numbers (with min exponent of the range “-101”)
4) Small numbers
5) Ordinary numbers

```
62  ////////////////////// Constraints DE ////////////////////////////////
63
64   constraint num_constraint {
65     foreach(c_de_1[i])    c_de_1[i] < 9999999  ;
66     foreach(c_de_2[i])    c_de_2[i] < 9999999  ;
67     foreach(exp_de_1[i])
68       exp_de_1[i] dist { 90 :/5          ,
69                          -101 :/5        ,
70                          [-100:-80] :/15 ,
71                          [70:89]    :/15 ,
72                          [-80:70]   :/60
73                        };
74     foreach(exp_de_2[i])
75       exp_de_2[i] dist { 90 :/5          ,
76                          -101 :/5        ,
77                          [-100:-80] :/15 ,
78                          [70:89]    :/15 ,
79                          [-80:70]   :/60
80                        };
81   }
```

Figure 104: decimal encoding representation constraints

5.5 BUGS

MODULE	BUGS	ACTIONS TAKEN BY RTL TEAM	CURRENT STATUS
DECIMAL ADDER	Incorrect result when one of the operands has precision of 6 digits	Create a new block to remove the leading zeros	Fixed
DECIMAL SUBTRACTOR	predicted result not equal to calculated at some cases exponent of the operands)	Use the GRS bits during normalization	Fixed
	Rounding error due to error in GRS bits	Change the rounding condition in the file (“rounding”)	Fixed
	Rounding error (when the exponent of the result is different from the	Change the length of BCD subtractor	Fixed
	Incorrect rounding and result when the difference between the operands is very small		Open Bug
DECIMAL MULTIPLIER	Overflow flag is always raised	Increase the length of the variable storing the sum of the exponent of the two operands in the file (“binary adder”)	Fixed
SINGLE PRECISION ADDER	No errors		

SINGLE PRECISION SUBTRACTOR	Inexact flag isn't raised correctly	Change the length of binary subtractor	Fixed
	Incorrect result and inexact flag in case of operands with small difference in exponent	During normalization use GRS bits, Switch the order between rounding and normalization	Fixed
	Inexact flag is not raised correctly in case of subnormal operands	Handle subnormal numbers as special case during normalization	Fixed
	Incorrect result when rounding occur	Remove GRS bits from the mantissa before rounding	Fixed
SINGLE PRECISION MULTIPLIER	Underflow flag is raised at subnormal result($\exp < -38$)	Modifying ("normalization") file to consider the subnormal numbers as results without raising the underflow flag	Fixed
	Incorrect result when one of the operands is subnormal	("exponent addition") module handles the case of multiplication between normal and subnormal operands as a special case	Fixed
	Overflow flag isn't raised correctly	Modifying the overflow condition	Fixed
INTEGRATED FPU	Output is always zero at first add/subtract operation after reset where operands are not of the same sign	Take into consideration the operand's sign for the selection of output multiplexer	Fixed
	Status bit isn't raised in simd instructions	Modifying ("simd") module and including finite state machine to control the module	Fixed
	Simd operation is t the same for all the operands as the first operation regardless the sign of the other operands	Updating the operation for each simd operation depending on the operand's sign	Fixed
	Flags in the first simd operation are not correctly risen	Creating special cases for small number of simd operations to directly connect the flags to the output	Fixed

Table 18: BUGS

CHAPTER SIX: SYNTHESIS AND FORMAL VERIFICATION

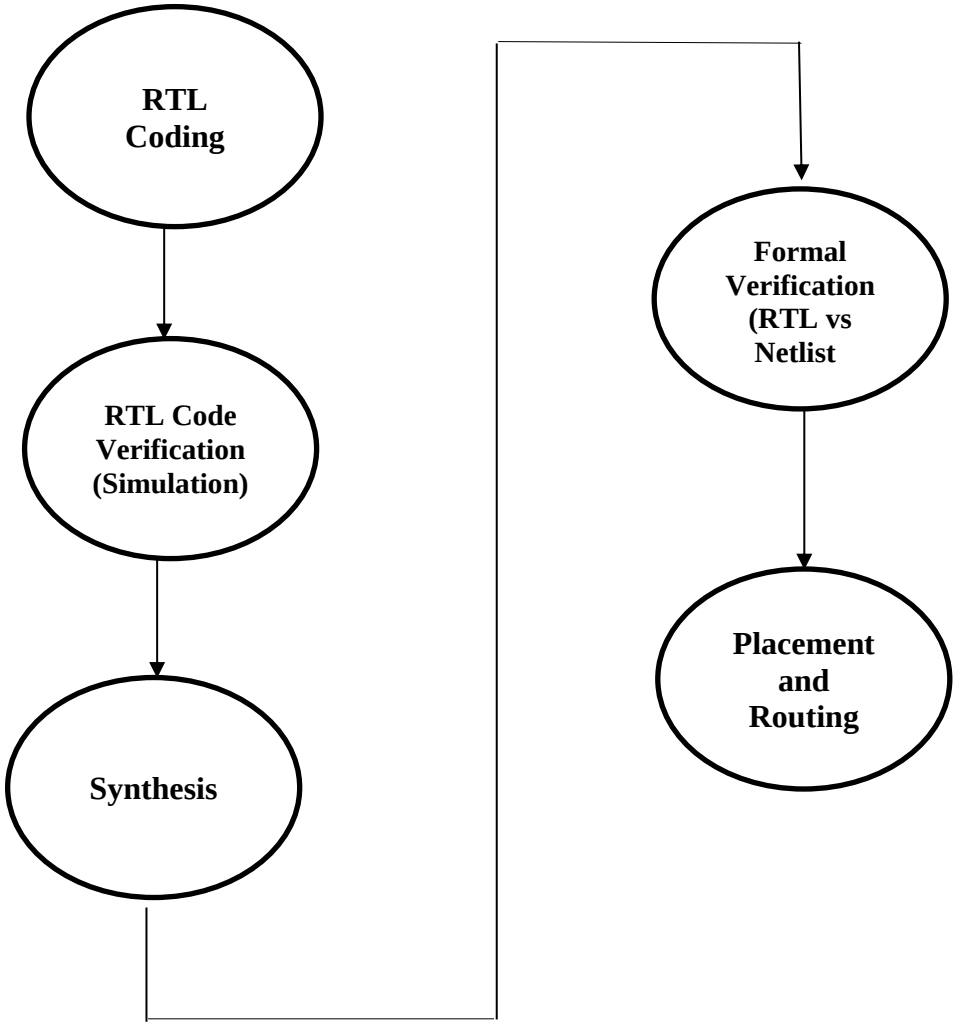

Figure 105: Design Flow Block Diagram

6.1 Synthesis

When synthesizing any design, we have certain considerations to take. We need to set the libraries to be used due to allowable fabrication technology and design techniques, and then we need to analyze our designs and sub-designs. Moreover, we need to define Performance figures like speed and power optimization constraints according to our needs. Furthermore, we need to

specify technology constraints like size and space (area). Finally, we need to compile the design according to the specified constraints, in a top-down or bottom-up strategy, afterwards we obtain reports about whether the constraints we set were satisfied or not, and the area, speed and power consumption of our design (9).

6.1.1 Flow Chart of the Synthesis Process

The steps of the synthesis process are done in a sequential manner as seen below in Figure 106. This sequence may be modified slightly to suit the design process of each designer.

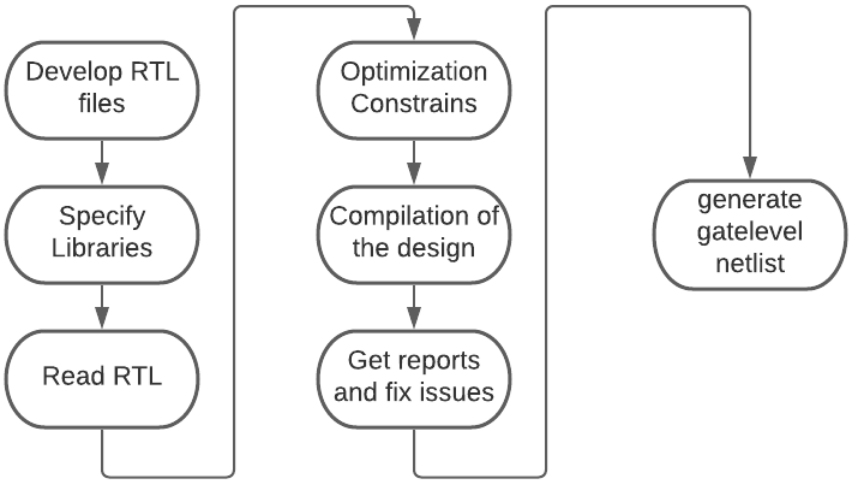

Figure 106: Synthesis process flow chart.

We will refer to the synthesis flow using the Synopsys Design Compiler tool in the following discussion because that is the tool we use in our practice.

6.1.2 Setting the Libraries

It is vital to define the technology library to which the design will be mapped so that the synthesis tool knows how to map the design. There are multiple library types, each contains specific information about the cells and the technology itself, they are as follows (9):

- **Target library:** contains all the logic cells that should used for mapping during synthesis. In other words, the tool during synthesis maps a design to the logic cells present in this library.

• **Link library:** contains information on the logic gates in the synthesis technology library. The tool uses this library solely for reference but does not use the cells present in it for mapping as in the case of target_library.

In order to specify the Technology library we set both the target library and the link library in our design to "NangateOpenCellLibrary_ss0p95v125c.db" as it has highest temperature, lowest voltage and slow-slow process, also notice that link library setting is a list that contains the technology library as well as an asterisk, which indicates that DC should resolve references by searching the memory (designs that have been analysed prior to this design) and then if it cannot find the reference in memory it will look in the technology library. If DC does not find the reference in either, it looks in the search path. The `search_path` is just a variable that tells DC where to look in order to resolve references that have not been found in the link library.

6.1.3 Reading in the Design

After specifying our libraries, we need to allow DC to read in the design. This phase consists of checking and analyzing the RTL for syntax errors, resolving references, mapping the design to technology-independent implementation (GTECH) before building the generic logic for the design. DC offers us with two options for accomplishing this. The `read_file` technique is the first, while the `analyse` and `elaborate` approach is the second. The `analyse` command also stores the result of the translation in the specified design library that maybe used later. So a design analyzed once need not be analyzed again and can be merely elaborated, thus saving time. Conversely read command performs the function of analyze and elaborate commands but does not store the analyzed results, therefore making the process slow by comparison, so we use `analyse` and `elaborate`.

6.1.4 Optimization Constrains

In order to properly optimize our design to give minimum area and highest speed, we must provide DC with constrains. This involves setting drive characteristics for input ports, setting loads on input and output ports.

6.1.4.1 Clock Characteristics

1. It is important for prelayout phase where clock tree are incompelete to specify transition time at register clock pins as it might be pessimistic. We can specify this using command `set_clock_transition`.

   The command `set_clock_transition` places clock_rise_transition and clock_fall_transition attribures on the clock_list we take it in our design with "0.1".

2. `set_clock_uncertainity` This command can specify either interclock uncertainty or simple uncertainty (skew characteristics). It has been set with 0.08 from period.

3. The clock network latency is the time it takes a clock signal to propagate from the clock definition point to a register clock pin. Design Compiler assumes ideal clocking but specifying clock network latency provides an estimate of the clock tree for pre-layout, so we used this command `set_clock_latency` with value 2.

6.1.4.2 Maximum Capacitance

We use the `set_max_capacitance` command to define the maximum total capacitive load that an output port can drive. Capacitance is specified in units consistent with technology library definition.

DC must check that the capacitive load of driven nets and interconnects is less than the `max_capacitance` attribute of the driving pin.

6.1.4.3 Speed

DC deals with timing constraints for speed optimization in a very specific way. Generally, the tool classifies timing paths into 3 categories as follows:

- **Path category 1**: From input to register (this path is constrained according to the input delay using `set_input_delay` command) and we set it with 0.4 from our period.
- **Path category 2**: From register to register (this path is constrained according to the clock period, using the `create_clock` command) and here we set period with $28ns$ as after some iterations of compilation we found it gives positive slack $= 0.26ns$.
- **Path category 3**: From register to output (this path is constrained according to output delay, using the `set_output_delay` command) and we set it with 0.4 from our period.

6.1.4.4 Area

In order to constrain the area of the design we use the `set_max_area` command and provide DC with the maximum area constraint. Setting `max_area` attribute to a value of zero means we want DC to optimize the area to the smallest possible size.

6.1.5 Compiling the Design

The design is mapped onto technology-specific gates at this step, and the design is also optimised at this time. We ask DC to map and optimise the design based on these limitations and environment settings after we've defined all of our constraints and environment variables. This is accomplished on three levels: the architectural, logic, and gate levels. “compile_ultra” is command that has been used.

DC performs high-level synthesis activities at the architectural level, such as reordering operators, sharing sub-expressions and resources, and picking other more optimal DesignWare implementations.

In the logic-level phase, DC is still working on GTECH implementation; here is where DC deals with the hierarchy in the design.

DC works on the netlist created by logic-level synthesis to create a technology-specific implementation in the gate-level phase. The actual technology-specific mapping, as well as delay and area optimization (according to restrictions) and any design rule constraint violations are completed in this step.

For large hierarchical designs consisting of many sub-circuits there are several strategies that we may use to compile the design, but we use Top-Down Strategy.

Top-Down Strategy

In this strategy, we set the constraints for the top level module only. We read all lower level modules, but we do not compile them separately. After all the modules have been read, and the top level constraints have been defined we compile the top level only, and DC infers the constraints required for lower level modules in order to satisfy the top level constraints, and thus it maps all modules accordingly.

At first we use `compile_ultra -retime -timing_high_effort_script` to give more effort to timing violations then we `set_critical_range 2` which specify that next optimization will be applied to critical path and paths which has till negative slack `2,` then we `compile_ultra -incremental` to optimize more or paths which may be violated but we found that there is a hold violation, and to solve it we `set_fix_hold [all_clocks]` and then `compile_ultra -retime -timing_high_effort_script` and `compile_ultra -incremental` and it gives good results without violations (10).

6.1.6 Report Analysis

Design Compiler makes it easy for us to generate a range of reports to check the accuracy and quality of our implementation. The time report, the area report, the quality of outcomes report, and the constraint report are the most significant reports. We'll go over each of them in detail below (9).

6.1.6.1 Timing Reports

Design Compiler has a built-in static timing analyzer called DesignTime. Static Timing Analysis can determine if a circuit meets timing constraints without dynamic simulation which is an advantage when it comes to saving time.

DesignTime works by breaking down our design into a set of timing paths, each has a startpoint and an endpoint.

Below in Figure 107. Notice that the time units are units consistent with those defines in the technology library, which is nanosecond in our case.

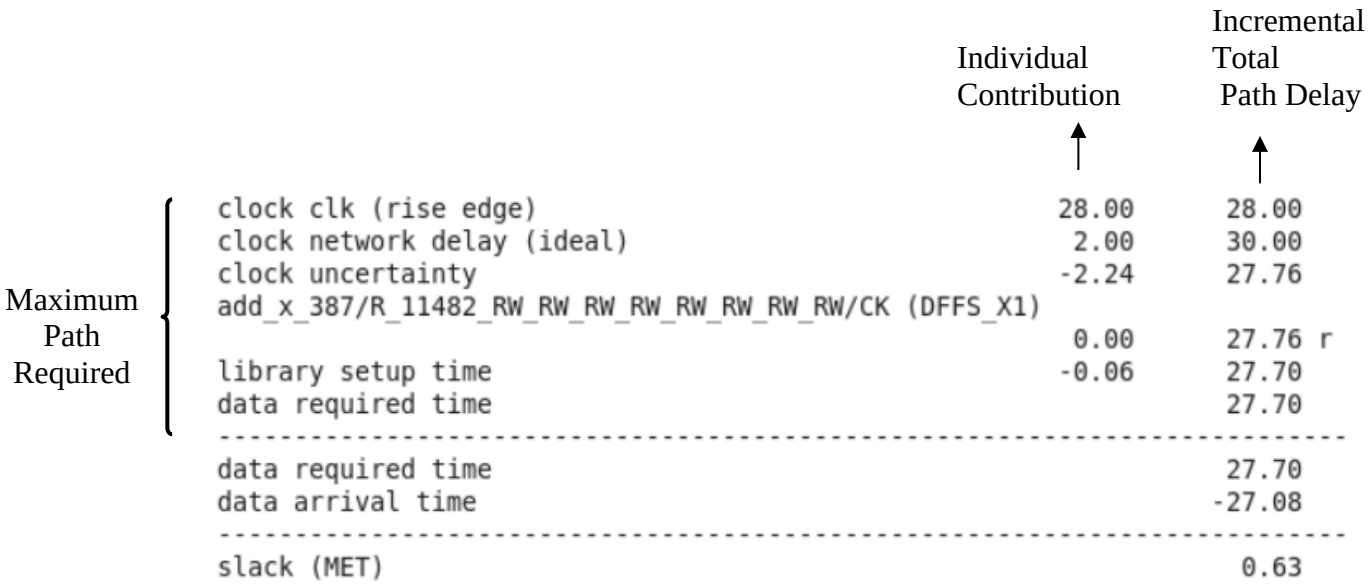

Figure 107: Part of Timing report example

As shown in following histogram that worst slack is 0.268ns which means that there is no setup time violation after synthesis.

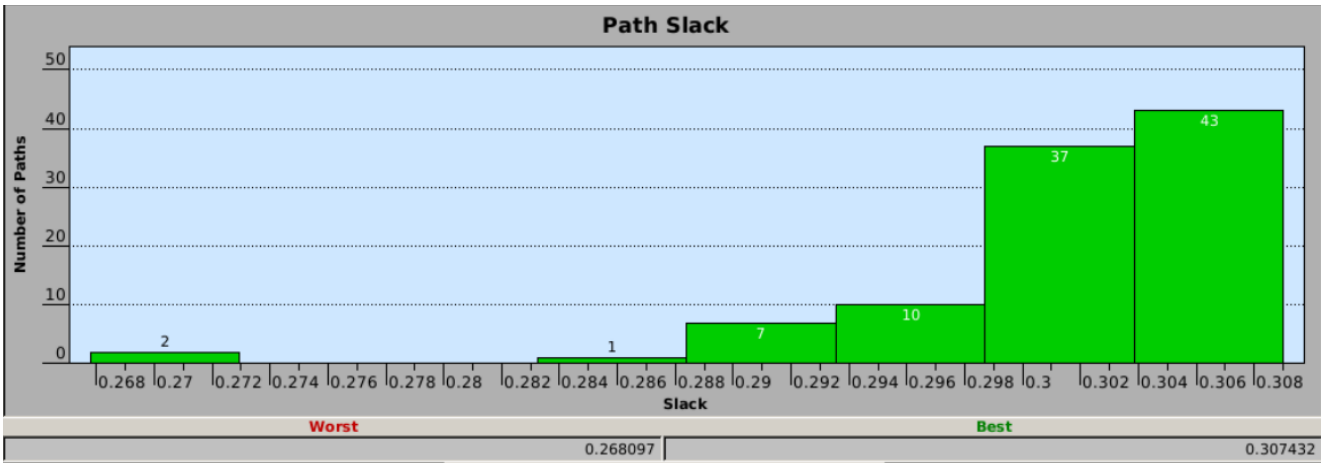

Figure 108: Path Slack histogram

6.1.6.2 Area and QoR Reports

This report gives us the total area of the design. It calculates this area by adding the area attributes of the gates from the technology library. Usually the area units are specified in the technology library as well, or in an associated document. In our case the area units were μm^2. Figure 109: Area report example shows an example of the area report. Notice that the interconnect area and the total area are reported as "undefined" because pre-layout we do not have an idea of interconnect area.

```
Number of ports:                        101
Number of nets:                       35898
Number of cells:                      34978
Number of combinational cells:        33600
Number of sequential cells:            1374
Number of macros:                         0
Number of buf/inv:                    26195
Number of references:                    76

Combinational area:       81509.314916
Buf/Inv area:             55600.382645
Noncombinational area:    17677.828569
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          99187.143485
Total area:                 undefined
1
```

Figure 109: Area Report example

```
Timing Path Group 'clk'
-----------------------------------
Levels of Logic:              146.00
Critical Path Length:          25.64
Critical Path Slack:            0.01
Critical Path Clk Period:      28.00
Total Negative Slack:           0.00
No. of Violating Paths:         0.00
Worst Hold Violation:           0.00
Total Hold Violation:           0.00
No. of Hold Violations:         0.00
-----------------------------------


Cell Count
-----------------------------------
Hierarchical Cell Count:            4
Hierarchical Port Count:          560
Leaf Cell Count:                93655
Buf/Inv Cell Count:             70971
CT Buf/Inv Cell Count:              0
Combinational Cell Count:       90334
Sequential Cell Count:           3321
Macro Count:                        0
-----------------------------------
```

```
Area
-----------------------------------
Combinational Area:    81509.314916
Noncombinational Area: 17677.828569
Buf/Inv Area:          55600.382645
Net Area:                  0.000000
-----------------------------------
Cell Area:             99187.143485
Design Area:           99187.143485


Design Rules
-----------------------------------
Total Number of Nets:         95884
Nets With Violations:             0
Max Trans Violations:             0
Max Cap Violations:               0
-----------------------------------


Hostname: localhost.localdomain

Compile CPU Statistics
----------------------------------------
Resource Sharing:                   0.0
Logic Optimization:                 3.2
Mapping Optimization:            5035.5
----------------------------------------
Overall Compile Time:            5053.4
Overall Compile Wall Clock Time: 30793.1
```

Figure 110: Summary of QoR Report

6.1.7 Design challenges

At first hold time violation was “-1.88ns”as we used compile_ultra only. We tried to fix this violation with:

1. Compile with -only hold time.
2. Specify timing optimization options in PnR script.
3. Opening worst path and insert more buffers in it or use larger driving buffers in it (this method works but there were large number).

Finally we solve it using specific types of compile as illustrated in section 6.1.5.

6.2 FORMAL VERIFICATION

Formal verification is a method to verify a design without running Simulations, thus saving simulation time. It works by comparing the "implementation" design against a "reference", golden model design, that has already been simulated (or proofed by formal verification against a previous reference design).

6.2.1 Basic Definitions:

- **Reference Design:** The golden model against which we verify the implementation design
- **Implementation Design:** The design we want to verify, here it is the synthesized netlist.
- **Container**: A container is like a "bucket" that carries the design library as well as the technology library info related to the design in the container. Conventionally, we have one container for the reference design and one for the implementation design.
- **Logic Cones:** is a cluster of combinational logic starting from a design object (like: primary outputs, internal registers, black box input pins and nets having multiple drivers where at least one driver is a port/black box) and spreading backward to terminate at certain design object outputs. Formality uses the origin points of logic cones to create compare points (when a logic cone boils down from multiple termination points to a single origin point, Formality compares the logic cone at this single origin point).
- **Compare Points:** The points of origin of logic cones at which formality compares the entire logic cones between reference and implementation designs.
- **Matching Compare Points:** finding the analogous logic cones between the reference and implementation designs. The analogous cones are matched by several techniques, performed as follows (in this order):
    - Exact-name matching
    - Name filtering
    - Topological equivalence
    - Signature analysis
    - Compare point matching based on net names
    - After compare points are matched, each pair of compare points is verified against the other for logic equivalence.

- **Verification:** Checking that matched compare point in the implementation design is logically equivalent to its peer in the reference design.

We used Formality tool by Synopsys and we got 4 unmatched points like in following Figure 111, but after checking with RTL team and system team we found that those 4 bits are redundant as they will always lead to logic1 or logic 0, and also when we verify design all points passed.

```
4 Unmatched points (4 reference, 0 implementation):

  Ref  DFF0X      r:/WORK/top/output_register_decimal_adder/flags_reg_reg[1]

  Ref  DFF0X      r:/WORK/top/output_register_decimal_subtractor/flags_reg_reg[2]

  Ref  DFF0X      r:/WORK/top/output_register_single_prec_adder/flags_reg_reg[1]

  Ref  DFF        r:/WORK/top/simd/simd_enable_reg
```

Figure 111: Unmatched Points

```
********************************* Verification Results *********************************
Verification NOT RUN
(Equivalence checking aborted due to errors)
-----------------------------------------
 Reference design: r:/WORK/top
 Implementation design: i:/WORK/top
--------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop   BBNet     Cut    Port     DFF     LAT   TOTAL
--------------------------------------------------------------------------------------
Passing (equivalent)           0       0       0       0      33    2449       0    2482
Failing (not equivalent)       0       0       0       0       0       0       0       0
**************************************************************************************
1
```

Figure 112: Verification Report

CHAPTER SEVEN: PHYSICAL DESIGN, PLACEMENT AND ROUTING STAGES

After the completion of the synthesis phase of a design, we can then move on to the next step, which is placement and routing of the netlist. There are several stages in the placement and routing (PnR) process.

7.1 BASIC PHYSICAL DESIGN FLOW USING IC COMPILER

The goal of physical design is to convert the synthesized netlist into a GDSII file that is manufacturable (9). The main steps of PnR flow can be seen in the figure below.

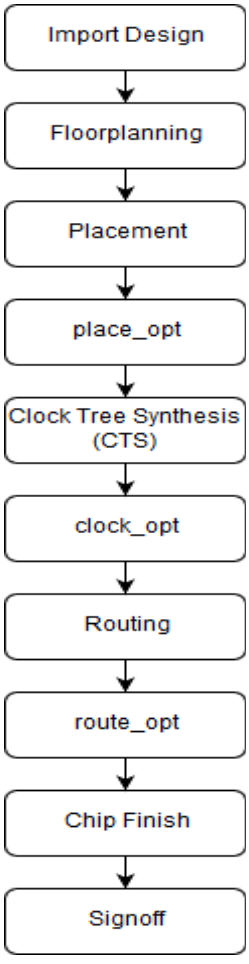

Figure 113: Basic Physical Design Flow

- Libraries and Files Used During the Physical Design Process:

1- Logical Libraries
   a. Provide timing and functionality information for all standard cells
   b. Provide timing information for hard macros
   c. Define drive/load design rules:
      Max fanout
      Max transition
      Max/Min capacitance
   d. usually the same ones used by Design Compiler during synthesis
   e. Are specified with variables:
      target_library
      link_library

2- Physical Libraries
   a. Contain physical information of standard, macro and pad cells, necessary for placement and routing.
   b. Define placement unit tile like Height of placement rows, Minimum, width resolution, Preferred routing directions and Pitch of routing tracks
   c. Are specified with the command:
      create_mw_lib –mw_reference_library

3- Technology Files
   A technology file is provided by the technology vendor. Technology file is unique for each technology and contains the information related to metal/vias information such as
   a. Units & precision for electrical units (V, I and power)
   b. Define colors and patterns of layers for displays
   c. Number & name designations for each metal/vias
   d. Physical & electrical characteristics of each metal/via
   e. Define design rules such as min. wire width & min. wire to wire spacing
   f. Contains ERC rules, Extraction rules, LVS rules
   g. Provide parameterized cells for MOS capacitance
   h. Create menus and commands

4- RC module files (TLU+)
   TLU is a Synopsys specific format which contains the R and C (Resistor and capacitance) values of nets used for routing .These R and C values will be required at the time of calculating a net delay which is nothing but product of these R and C of a net.In our design we use following TLU files :
   set tlupmax "$sc_dir/tech/rcxt/FreePDK45_10m_Cmax.tlup"
   set tlupmin "$sc_dir/tech/rcxt/FreePDK45_10m_Cmin.tlup"

```
set tech2itf "$sc_dir/tech/rcxt/FreePDK45_10m.map"
```

7.2 FLOORPLANNING

Floorplan is one the critical & important step in Physical design. Quality of your Chip / Design implementation depends on how good the floorplan is.

A floorplanning is the process of placing blocks/macros in the chip/core area, thereby determining the routing areas between them. Also, it determines the size of die and creates wire tracks for placement of standard cells, creates power ground (PG) connections, and determines the I/O pin/pad placement.

Parameter	Value
Aspect Ratio (AR)	1.0 (square)
Maximum Core Utilization	0.6 (60%)

Table 19: Floorplanning Parameters

7.3 PLACEMENT

After the floorplanning and power-planning stage, we need to begin placing the standard cells in uniform rows inside the core area and fix the obtained placement of the macros as well. This stage can greatly influence the timing parameters of our design, as it specifies the finalized placement of blocks and standard cells, thus providing a more accurate estimate of interconnect lengths and thus delays.

Keeping the above in mind, we need to be vigilant during our checks in this stage to ensure that the rest of the flow will go as smoothly as possible. Here we start to fix hold violation using "`set_buffer_opt_strategy -effort high`" which introduces buffers and inverters to fix timing.

The placement stage is done using the `place_opt` command and it has several sub-steps. There are several options for configuring the flow of this stage according to the needs of our

design. For example, we may invoke `place_opt` with `-congestion` to encourage the tool to place cells with the goal of minimizing congestion or with `-area_recovery` which enables buffer removal and cell downsizing of non critical paths, and in our design we use them both with `-effort high.`

At end of this stage we `check_legality` sure that all the cells are placed in row with no overlaps.

After Placement we use "`report_timing -delay max -max_paths 20 > output/top_place.setup.rpt`" and "`report_timing -delay min -max_paths 20 > output/top_place.hold.rpt`" to check violations and there was a hold violation with -2.01ns which will be fixed in next steps.

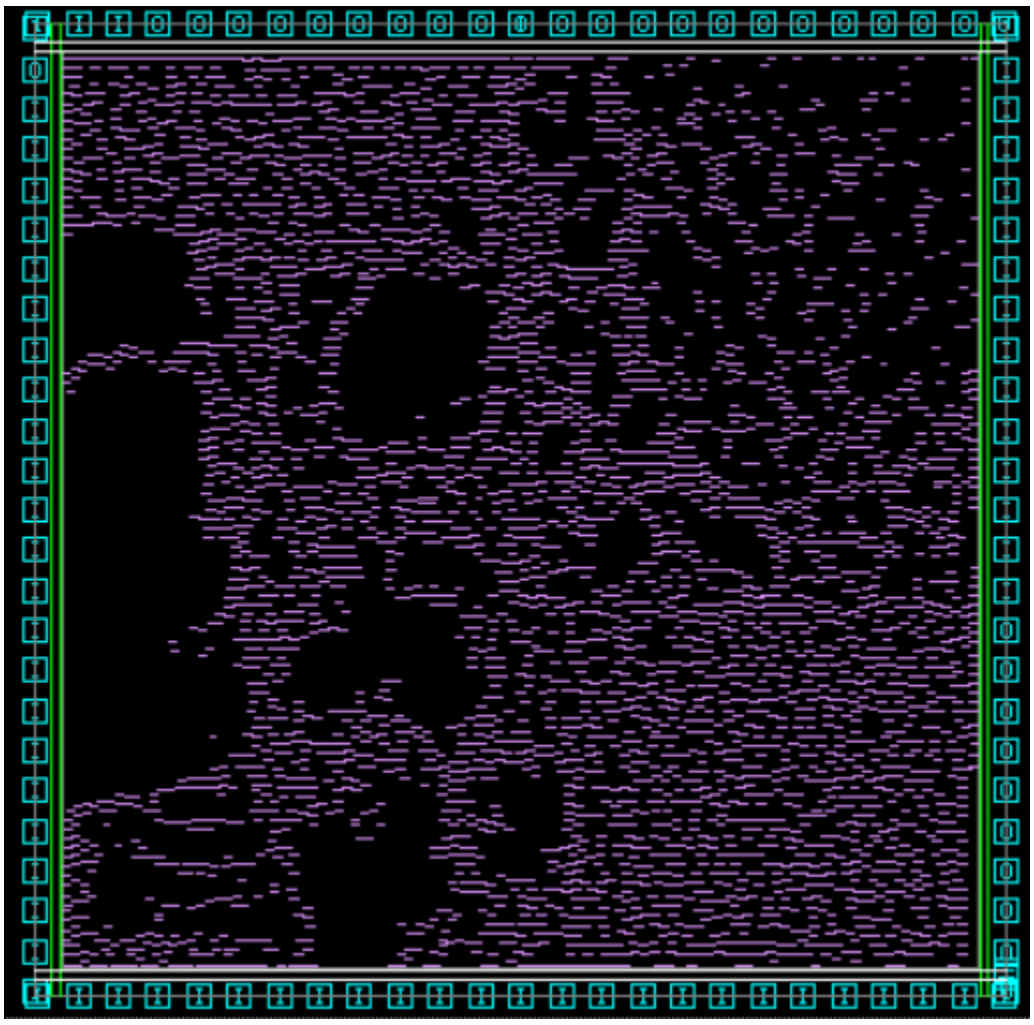

Figure 114: Floorplan and power rings Placement of FPU

Figure 115: Zoomed in view of power rings and floorplanning placement

7.4 CLOCK TREE SYNTHESIS (CTS)

We can go on to the Clock Tree Synthesis (CTS) stage after completing the placement stage with acceptable timing and estimations of congestion/power usage. We deal with the clock nets that were previously viewed as optimal throughout this stage. The parts that follow describe the step and what is done during CTS, as well as the inputs to the stage and the desired outputs or goals.

CTS is essentially the insertion of buffers along the clock paths in the design in order to balance the skew (differences in clock signal delay between clock inputs) and satisfy the required insertion delay (time taken by clock signal to traverse from clock definition point to the sink of the clock). The balancing of clock skew is done by building a buffer tree, as illustrated in following figure, below. The handling of insertion delay is done by adding delay lines, as illustrated too.

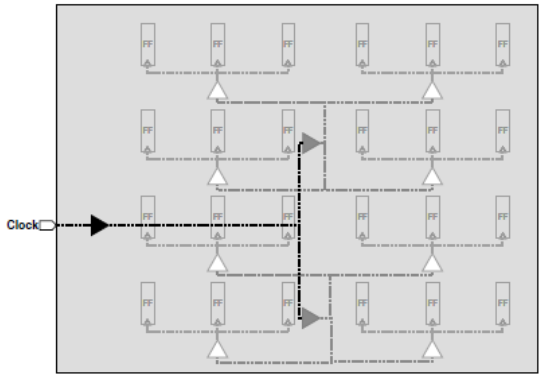

Figure 116:Balancing of Clock Skews

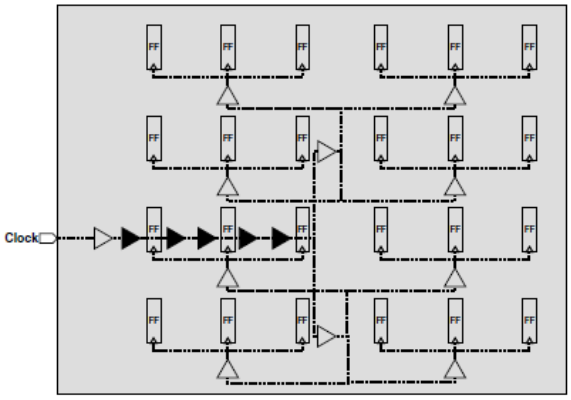

Figure 117:Handling Insertion Delay

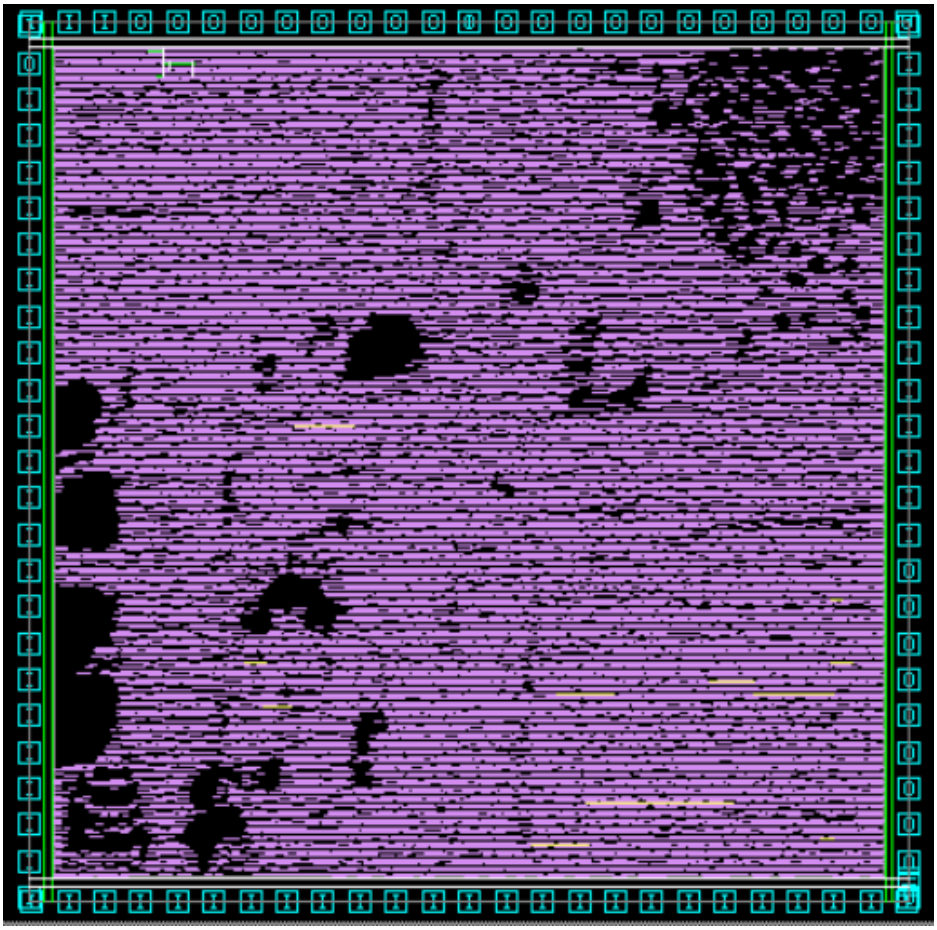

Figure 118: Layout after CTS

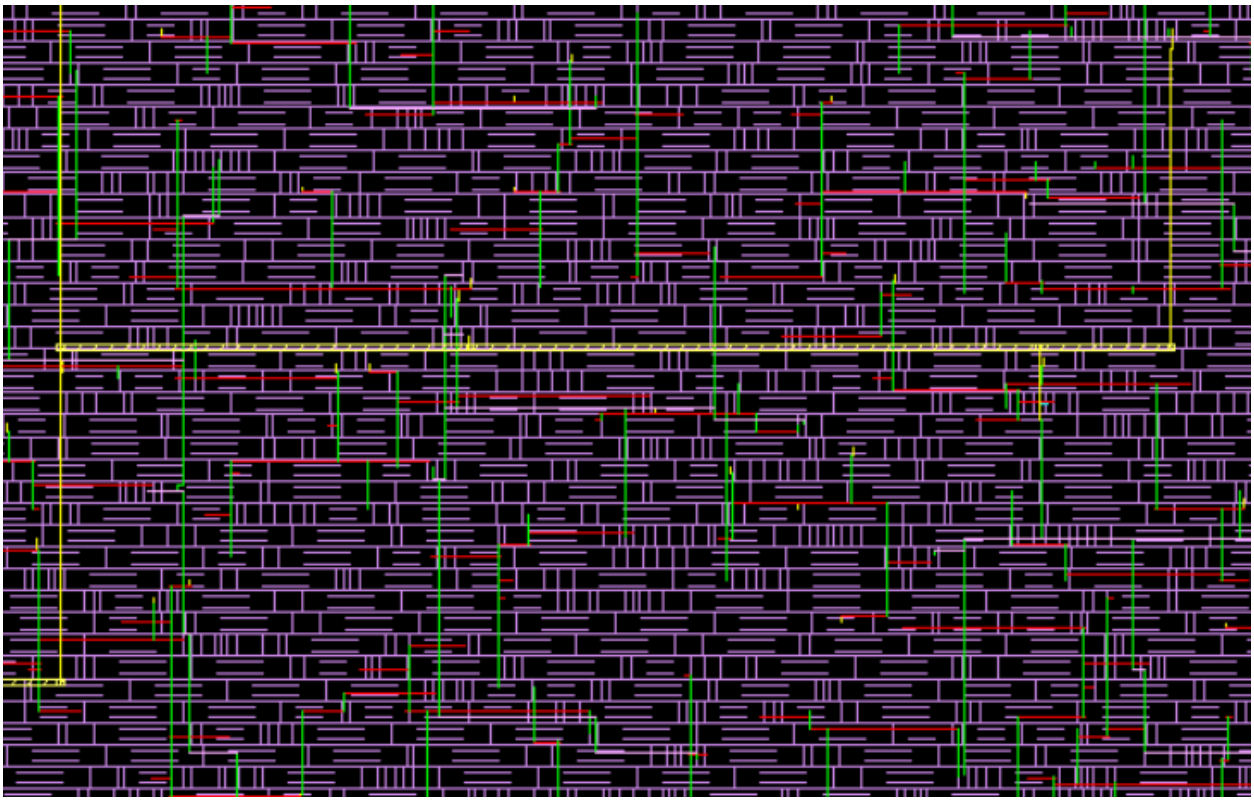

Figure 119: Zoomed in view after CTS

7.5 ROUTING

After the CTS stage is completed with satisfactory skew-balancing and no hold (or setup) timing violations, we may proceed to the routing stage. In this stage the design undergoes detailed routing, where the actual path of interconnects across different metal layers and in different geometric configurations is determined, so as expected area increased a lot as shown inFigure 120.

```
Area
-----------------------------------
Combinational Area:   136444.964880
Noncombinational Area: 17677.562569
Buf/Inv Area:         110497.462605
Net Area:                  0.000000
Net XLength        :      382402.66
Net YLength        :      409925.59
-----------------------------------
Cell Area:            154122.527449
Design Area:          154122.527449
Net Length         :      792328.25
```

Figure 120: Summary of final area

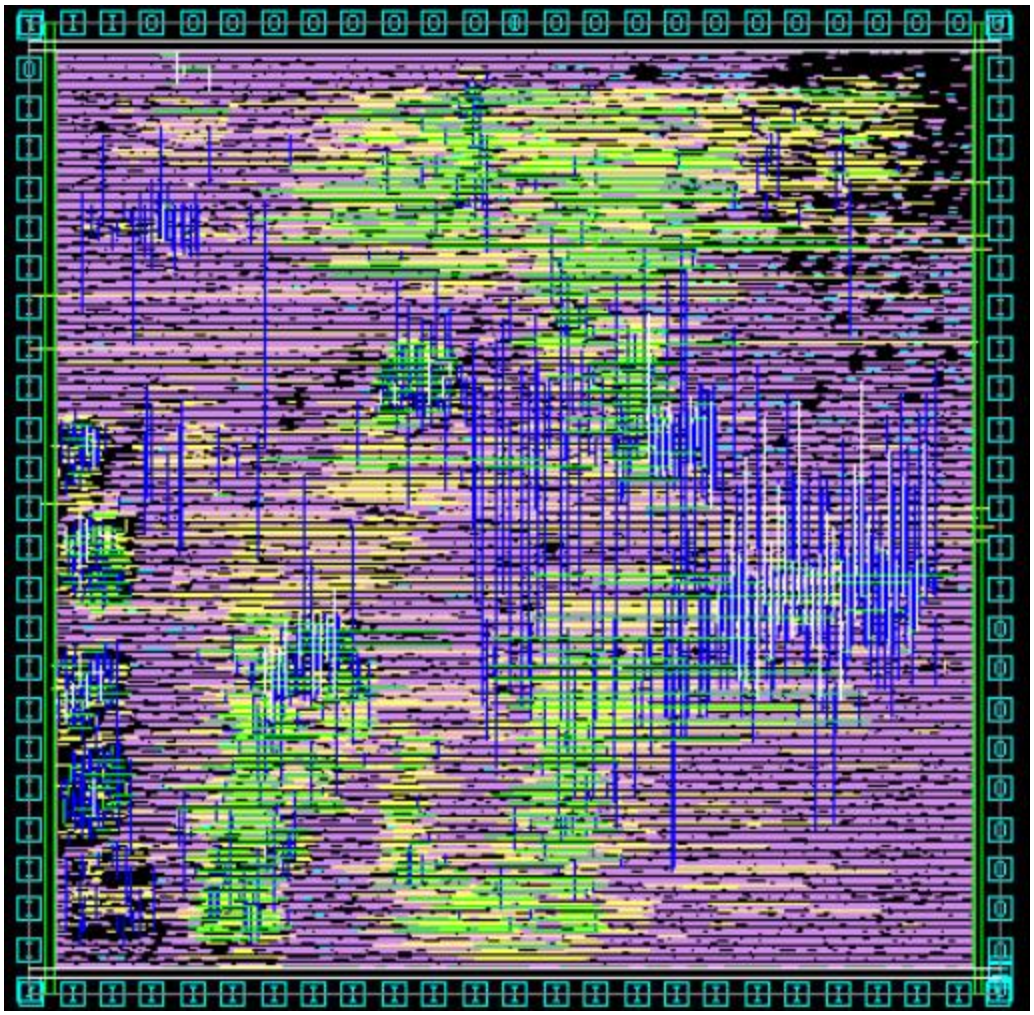

Figure 121: Final Layout of FPU

PROJECTS CODE LINKS

INTEGRATED FPU RTL CODE

https://drive.google.com/drive/u/0/folders/1JWLysGlZydS-aQwMg3v_szmyj6i19_vM

SYSTEM CODE

https://drive.google.com/drive/u/0/folders/1NannWNHSFAEaqcV2o_RvE4jhI5tWiKcX

TESTING ENVIRONMENT LINK ON EDA PLAYGROUND

https://www.edaplayground.com/x/JvFm

PHYSICAL DESIGN SCRIPTS

https://drive.google.com/drive/folders/1-FJvMNPa0Mv9naBD6y11cY8nzqXDsSQ6

BIBLIOGRAPHY

1. *IEEE Standard for Floating-Point.* 2019. IEEE-754.
2. *A. V. Alvarez, "High-performance Decimal Floating-Point Units," University of Santiago de Compostela, Jan. 2009.*
3. https://github.com/pulp-platform/pulpino. [Online]
4. https://vhdlwhiz.com/modelsim-quartus-prime-lite-ubuntu-20-04/. [Online]
5. https://github.com/pulp-platform/pulpino/issues/196?fbclid=IwAR1Ofd2zRy1yiL3_7S055Dw8uw8JK1VbiqZ3xxVTnJhGKsOmXR1FtXO8quY. [Online]
6. https://www.eecs.umich.edu/courses/eecs373/readings/IHI0024C_amba_apb_protocol_spec.pdf. [Online]
7. *SystemVerilog for Verification A Guide to Learning the Testbench Language Features Third Edition.*
8. *The UVM Primer An Introduction to the Universal Verification Methodology by Ray Salemi .*
9. *ASIC Design Flow Tutorial Using Synopsys tools.*
10. *Design Compiler Optimization Reference Manual .*