Abstract

Most computers today support binary floating-point in hardware. While suitable for many purposes, binary floating-point arithmetic should not be used for financial, commercial, and user-centric applications or web services because the decimal data used in these applications cannot be represented exactly using binary floating-point [1].

The problems of binary floating-point can be avoided by using base 10 (decimal) exponents and preserving those exponents where possible. So, in order to overcome this problem, we introduce a decimal floating-point adder subtractor based on the final version of the IEEE Standard for Floating-Point Arithmetic P754r which was published in August 2008.

The previously mentioned standard is the revised version of IEEE 754-85 which is the IEEE standard for the Binary floating-point arithmetic that was published in 1985.

The design performs addition and subtraction on 64-bit operands in a single path adder with exception handling fulfilling the released standard and it can easily be extended to also support operations on 128-bit decimal floating-point numbers.

We introduced 2 different implementations for the BCD-subtractor internal design. The tens complement and the nines complement. We found out that in case we should complement the output the rippling of the carry in case of tens-complement makes it much slower than the nines complement. So, we tried another architecture in which we added another BCD-subtractor block for which we interchanged the 2 operands so that in case we need to complement the output all we have to do is -with the aid of an extra multiplexer- we select either the first or second BCD-subtractor so we won't wait for the carry rippling. This implementation enhanced the speed but on the other hand the

area is also increased. Regarding both the area and speed, we found out that the nines complement is more suitable for our design for both area and speed

The internal design of the BCD-adder is the carry-ripple adder which is known by its small area, we introduced another implementation for the BCD-adder which is the carry look-ahead adder and we used the nine's complement for subtraction. We found out that the speed is enhanced and the area is a increased (as expected).

We compared the overall performance of the decimal adder from the point of view of area and speed for the same FPGA families. We synthesized the design for 2 families of Xilinx, Spartan II and Vertix II. And we got the previously mentioned results.

Complete test and verification is performed on all the design versions fulfilling 3063 test vectors supplied by IBM Corp. and supporting 7 rounding modes (5 stated by the standard and 2 proposed by IBM) with exception handling for overflow, inexact and invalid operations.

Contents

List of Tables

List of Figures

Definitions

Quiet operation: Any of the operations specified by this standard that never generate an exception.

Biased exponent: The sum of the exponent and a constant (bias) are chosen to make the biased exponent's range nonnegative.

Binary floating-point number: A floating-point number with radix two.

Cohort: In a given format, the set of floating-point representations with the same numerical value.

Decimal floating-point number: A floating-point number with radix ten.

Declet: An encoding of three decimal digits into ten bits using the densely
packed decimal encoding scheme. Of the 1024 possible declets, 1000
canonical declets are produced by computational operations, while 24 noncanonical declets are not produced by computational operations, but are accepted in operands

Exception: An event that occurs when an operation has no outcome suitable for every reasonable application.

Exponent: The component of a binary floating-point number that normally signifies the integer power to which the radix two is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent.

Floating-point number: A bit-string encoding characterized by three components: a sign, a signed exponent, and a significand. Its

numerical value, if any, is the signed product of its significand and its radix two rose to the power of its exponent. In this standard a bit-string is not always distinguished from a number it may represent.

NaN: Not a Number, a symbolic entity encoded in floating-point format. There
are two types of NaNs , quiet and signaling. quiet NaNs propagate through almost every arithmetic operations without signaling exceptions, while signaling NaNs signal the invalid operation exception whenever they appear as operands.

Signal: When an operation has no outcome suitable for every reasonable application, that operation might signal one or more exceptions by invoking the default or user-specified alternate handling. Note that “exception” and “signal” are defined in diverse ways in different programming environments.

Significand: A component of an unencoded binary or decimal floating-point number containing its significant digits. The significand may be thought of as an integer, a fraction, or some other fixed-point form, by choosing an appropriate bias. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

Chapter 1

Introduction

Background

Although most people use decimal arithmetic when performing manual calculations, computers typically only support binary arithmetic in hardware. This is primarily due to there being only two logic values, zero and one, that are represented in modern computers. While it is possible to use these two logic values to represent decimal numbers, doing so is wasteful in terms of storage space and is also less efficient. For example, in binary, four bits can represent sixteen values; while in binary coded decimal (BCD), four bits only represent ten values. Since most computer systems do not provide hardware support for decimal arithmetic, numbers are typically input in decimal, converted from decimal to binary, processed using binary arithmetic, and then converted back to decimal for output.

In spite of the current dominance of hardware support for binary arithmetic, there are several motivations that encourage the provision of support for decimal arithmetic. First, applications that deal with financial and other real-world data often have errors introduced, since many common decimal numbers cannot be represented exactly in binary. For example, the decimal number “0.1” is a repeating fraction when represented in binary. Second, people typically think about computations in decimal, even when using computers that operate only on binary representations, and therefore may experience what is perceived as incorrect behavior when processing decimal values. Third, converting between binary and decimal floating-point numbers is computationally intensive and may take thousands of cycles on modern processors.[2]

Decimal data permeates society, as humans most commonly use numbers in base-ten. An increasing demand for decimal real number computations across a wide range of exponents has spurred the IEEE 754R Working Group to include specifications for Decimal Floating-Point (DFP) arithmetic in the new IEEE P754 Draft Standard for Floating-point Arithmetic [11]

Decimal Floating-Point (DFP) computations are critical for many financial and commercial applications. With trends towards globalization, many laws and standards require decimal calculations. For example, the European Union requires currency conversion to and from the euro to be calculated to six decimal places. One study estimates that a large telephone billing system can accumulate errors of up to $5 million per year, if using binary floating-point arithmetic, rather than decimal. Both hardware and software solutions for DFP arithmetic are being developed to remedy these problems [11].

Also, another important question is why do we need to replace the existing software conversion from decimal to BCD than back to decimal into hardware. An interesting study [14] shows that application can realize performance improvements ranging from about 10% (for applications whose respective DFP routines consumes 10% of the execution time) to nearly 1000% (for applications whose respective DFP routines consumes 90% of the execution time)

Due to the rapid growth in financial, commercial, and Internet-based applications, there is an increasing desire to allow computers to operate on both binary and decimal floating-point numbers. Consequently, specifications for decimal floating-point arithmetic are being added to the IEEE-754 Standard for Floating-Point Arithmetic which was published in 1985. In this thesis, we present the design and implementation of a decimal floating-point adder/subtractor that is compliant with the final revision of the IEEE-754r Standard. The adder supports operations on 64-bit (16-digit) decimal floating-

point operands. We provide 2 different architectures for the adder/subtractor and 2 different internal designs for the subtractor in accordance with 2 different internal designs for the adder. Synthesis results indicating the area usage and the clock frequency with 2 Xilinx FPGA families, Spartan II and Vertix II for our design were introduced. Also, comparison with other designs is introduced.

Problem description

Binary floating-point cannot exactly represent decimal fractions, so if binary floating-point is used it is not possible to guarantee that results will be the same as those using decimal arithmetic. This makes it extremely difficult to develop and test applications that use exact real-world data, such as commercial and financial values [4].

Here are some specific examples:

1. Taking the number 9 and repeatedly dividing by ten yields the following results shown in Table 1.1:

Decimal	Binary
0.9	0.9
0.09	0.089999996
0.009	0.0090
0.0009	9.0E-4
0.00009	9.0E-5
0.000009	9.0E-6
9E-7	9.000000E-7
9E-8	9.0E-8
9E-9	9,0E-9
9E-10	8.999999E-10

Table 1.1: Binary versus Decimal division.

2. Here, the left hand column shows the results delivered by decimal floating-point arithmetic (such as the BigDecimal class for Java or the decnumber C

package), and the right hand column shows the results obtained by using the Java float data type. The results from using the double data type are similar to the latter (with more repeated 9s or 0s).

3. Some problems like this can be partly hidden by rounding, but this confuses users. Errors accumulate unseen and then surface after repeated operations.

4. For example, Consider the calculation of a 5% sales tax on an item (such as a $0.70 telephone call), which is then rounded to the nearest cent. Using double binary floating-point, the result of 0.70 x 1.05 is 0.73499999999999998667732370449812151491641998291015625; the result should have been 0.735 (which would be rounded up to $0.74) but instead the rounded result would be $0.73 (using Banker's rounding). Which will introduce an error of 1 cent per telephone call.

5. Even a single operation can give much unexpected results. For example:

    - Similarly, the result of 1.30 x 1.05 using binary is 1.365000000000000213162820728030055761337280273437 5; this would be rounded up to $1.37. However, the result should have been 1.365 – which would be rounded *down* to $1.36 (using Banker's rounding).

      Taken over a million transactions of this kind, as in the 'telco' benchmark, these systematic errors add up to an overcharge of more than $20. For a large company, the million calls might be two-minutes-worth; over a whole year the error then exceeds $5 million.

    - Using binary floating-point, calculating the remainder when 1.00 is divided by 0.10 will give a result of exactly 0.09999999999999995003996389186795568093657493591308593 7

Even if rounded this will still give a result of 0.1, instead of 0, the result obtained if decimal encoding and arithmetic are used.

Related work

The decimal-encoded formats and arithmetic described in the new IEEE 754-2008 standard now have many implementations in hardware and software including:

- The hardware decimal floating-point unit in the IBM Power6 processor, the firmware (with assists) in the IBM System z9 (mainframe) processor, and the hardware decimal floating-point unit in the IBM System z10 mainframe which is the first mainframe with hardware support for the DFP format in the IEEE 754-2008 floating-point standard. It joins the IBM POWER6 processor-based System p 570 server as the only hardware support available for this format [13].

- Benchmark suite of financial Decimal Floating-Point (DFP) applications. The benchmark suite includes a banking benchmark, a euro conversion benchmark, a risk management benchmark, a tax preparation benchmark, and a telephone billing benchmark. The benchmark suite is being made publicly available [11].

- SilMind's Decimal Floating Point Arithmetic hardware IP Cores Family. Two hardware implementations are introduced for decimal floating-point adder that is compliant with the IEEE 754-2008 Standard; one for High-Speed applications and the other for Low Power/Area ones [12].

- IBM XL C/C++ for AIX, Linux and z/OS, DB2 for z/OS, Linux, UNIX, and Windows, and Enterprise PL/I for z/OS; IBM is also adding support to many other software products including z/VM V5.2, System i/OS, the dbx debugger, and. Debug Tool Version 8.1

- SAP NetWeaver 7.1, which includes the new DECFLOAT data dtype in ABAP, with support for hardware decimal floating-point on Power6
  - GCC 4.2 was released in July 2007; this is the first GCC release with support for the proposed ISO C extensions for decimal floating point.

Also, some related work on decimal arithmetic includes designs for fixed-point decimal adders and floating-point decimal arithmetic units. An extensive bibliography of support for decimal arithmetic is presented in [1].

The proposed decimal floating-point adder differs from previous decimal adders in that it is compliant with the final version of the revised IEEE-754 Standard.

Thesis outline

The following chapters provide detailed information about the IEEE 754-2008 standard for floating-point Arithmetic, architecture and implementation for our 64-bit decimal floating point adder/subtractor compliant with the standard with extensive testing according to IBM test suite.

Chapter Two: Overview of the final IEEE 754-2008 standard for floating-point arithmetic with focus on the decimal part of it from the point of view of the format, encoding, rounding modes and exception handling.

Chapter Three: Architecture and Implementation which gives detailed information for our 64-bit adder/subtractor discussing the internal design of each block and its hierarchal levels as well (if any).

Chapter Four: Verification and Testing for the design, the problem we faced during testing and how we solve it. Also, synthesis results are discussed in details.

Chapter Five: Similar work comparison, which is a review of what has been done as hardware implementation for decimal adder/subtractor from companies as well as universities.

Chapter Six: Illustrates the conclusions and offers suggestions for future work.

References

Chapter 2

Overview of the standard

History

The first IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985) set the standard for floating-point computation for 23 years. It became the most widely-used standard for floating-point computation, and is followed by many CPU and FPU implementations. Its binary floating-point formats and arithmetic are preserved in the new IEEE 754-2008 standard which replaced it.

The 754-1985 standard defines formats for representing floating-point numbers and special values (infinities and NaNs) together with a set of floating-point operations that operate on these values. It also specifies four rounding modes and five exceptions (including when the exceptions occur, and what happens when they do occur).

The draft version of the standard including the decimal part was first issued on 12 Feb 2001 and finally released in August 2008.

We started by following the DRAFT Standard for Floating-Point Arithmetic P754/D0.10.4 2005 March 14 16:43 and then after the publishing of the standard we made the required modification so that the current design is now following the final version.

Scope

This standard specifies formats and methods for binary and decimal floating-point arithmetic in computer programming environments: standard and extended functions in 32-, 64-, and 128-bit basic formats single, double, quad, and extended precision formats, and recommends formats for data interchange. Exception conditions are defined and default handling of these conditions

An implementation of a floating-point system conforming to this standard can be realized entirely in software, entirely in hardware, or in any combination of software and hardware. For operations specified in the normative part of this standard, numerical results and exceptions are uniquely determined by the values of the input data, sequence of operations, and destination formats, all under user control.

Keywords: computer, floating-point, arithmetic, rounding, format, interchange, number, binary, decimal, subnormal, NaN, significand, exponent.

Purpose

This standard provides a discipline for performing floating-point computation that yields results independent of whether the processing is done in hardware, software, or a combination of the two. For operations specified in this standard, numerical results and exceptions are uniquely determined by the values of the input data, sequence of operations, and destination formats, all under user control.

Formats

This standard defines five basic floating-point formats and two storage floating-point formats, in two radices, two and ten. Binary basic format lengths are 32, 64, and 128 bits; the binary storage format length is 16 bits.

Decimal basic format lengths are 64 and 128 bits; the decimal storage format length is 32 bits. A programming environment conforms to this standard, in a particular radix, by providing one or more of the basic formats for that radix.

Binary floating-point formats are indicated for:

• supporting scientific computation

• Applications for which the input data is not known exactly

• Applications for which arithmetic time dominates time spent in conversion between internal floating-point formats and external decimal formats

• Applications for which maximum performance is critical – binary is either faster or cheaper than decimal of the same fixed word size

• Applications for which maximum accuracy is critical – binary packs more precision in a fixed word size and the change in roundoff is less extreme at powers of the radix

Decimal floating-point formats are indicated for:

• The bulk of casual numerical applications for which ease of debugging is the most important numerical quality

• Supporting business applications especially those with financial data

• Applications for which the input data is known exactly in decimal

• Applications for which time spent in conversion between internal floating-point formats and external decimal formats dominates arithmetic time

Many applications work well with data and computation in 64-bit formats. 128-bit formats are useful as native formats for computations in which roundoff error would otherwise dominate accuracy of results, and as evaluation formats for complicated expressions involving 64-bit formats.

Binary32 is useful as a computational format for applications which consume or produce much low-precision data, especially if that data is in binary16 storage format. If those computations perform few operations per

datum, then binary32 may be a satisfactory expression evaluation format; otherwise binary64 is good for complicated expression evaluation.

Basic Decimal Format Encodings

Unlike basic binary floating-point formats, a representable number may have multiple representations in a basic decimal format. The set of floating-point representations a number maps to is called the number's cohort; the members of a cohort are distinct representations of the same number. For example, if c is a multiple of 10 and q is not its maximum, (s, q, c) and (s, q+ 1, c ÷ 10) are two representations for the same number and are members of the same cohort.

Numbers in the decimal formats are encoded in the following four fields ordered as shown in table 2-1:

1. 1-bit sign S

2. 5-bit combination field G encoding classification, two leading exponent bits whose value together is 0, 1, or 2, and one leading significand digit

3. w-bit following exponent field F which, when combined with the two leading exponent bits from the combination field, provides a w+2-bit biased exponent $E = q + bias$

4. t-bit trailing significand field $T = J^1 \ldots J^J$. There are $J = t \div 10$ groups J^j; each these groups of ten bits is a declet encoding three decimal digits. When the declets are combined with the leading significand digit from the combination field, the format has a total of $p = 1 + 3\,J$ decimal digits. Computational operations produce only 1000 canonical declets, but also accept 24 noncanonical declets in operands according to Tables 2-3 and 2-4.

Width	1 Bit	5 Bits	W Bits	t=10 *J* bits=3 *J* digits
Field	Sign S	Combination G	Following Exponent F	Trailing signficand T Containing *J* declets
Most/least significant bit		Most.....….Least G0……G4	Most..Least *F2*……..F_{w+1}	Most……..Least d_1…..…..…..…d_{3J} j_1…..…..…..…..j_J

Table 2.2: Basic Decimal Floating-Point Format

The values of w, bias, and t for the basic decimal formats are listed in Table 2-2.

Basic Decimal Format Encoding Parameters			
Format Name	**Decimal32**	**Decimal64**	**Decimal 128**
Storage Width	32	64	128
Trailing significand field width t	20	50	110
Following exponent field width w	6	8	12
Combination field width	5	5	5
emax	96	384	6144
Exponent bias	101	398	6176

Table 2.3: Decimal Encodings

The floating point representation r and representable entity v are inferred from the constituent fields, thus:

1. If G is 11111, then r is qNaN or sNaN and v is NaN regardless of S. The values of F and T distinguish various NaNs. If F_2, the most significant bit of F, is 1, then r is sNaN; otherwise r is qNaN. [This allows the all-1 bit pattern to be a decimal signaling NaN. However, the all-1 bit pattern might not be propagated; A canonical NaN representation has bits F_3 to F_{w+1} zero, and trailing significand declets are all canonical.

2. If G is 11110, then r and v = $(-1)^S \infty$. The values of F and T are ignored. The two canonical infinity representations have F = 0, T = 0.

3. For finite numbers, r is (S, E–bias, c) and v = $(-1)^S 10^{E-bias} c$; the decimal digit string $d_0\ d_1 ... d_{p-1}$ of the significand c is encoded in the combination and trailing significand fields, while the biased exponent E is encoded in the combination and following exponent fields:

• When the combination field G is 110xx or 1110x, the leading significand digit d_0 is $8+G_4$, a value 8 or 9, and the leading exponent bits are $2G_2+G_3$, a value 0, 1, or 2.

• When the combination field G is 0xxxx or 10xxx, the leading significand digit $d_0$is $4G_2+2G_3+G_4$, a value in the range 0..7, and the leading exponent bits are $2G_0+G_1$, a value 0, 1, or 2. Consequently if T is 0 and G is 00000, 01000, or 10000, then v = $(-1)^S$ 0.

The trailing significand field T contains J declets, groups of ten bits each encoding three decimal digits using the densely packed decimal encoding scheme described in Cowlishaw, M.F., “Densely Packed Decimal Encoding,”

IEE Proceedings - Computers and Digital Techniques, ISSN 1350-2387, Vol. 149, No. 3, pp102-104, May 2002.

A canonical number representation has only canonical declets – see Tables 2-3 and 2-4.

$b_{(6)}$, $b_{(7)}$, $b_{(8)}$, $b_{(3)}$, $b_{(4)}$	$d_{(1)}$	$d_{(2)}$	$d_{(3)}$
0 x x x x	$4b_{(0)} + 2b_{(1)} + b_{(2)}$	$4b_{(3)} + 2b_{(4)} + b_{(5)}$	$4b_{(7)} + 2b_{(8)} + b_{(9)}$
1 0 0 x x	$4b_{(0)} + 2b_{(1)} + b_{(2)}$	$4b_{(3)} + 2b_{(4)} + b_{(5)}$	$8 + b_{(9)}$
1 0 1 x x	$4b_{(0)} + 2b_{(1)} + b_{(2)}$	$8 + b_{(5)}$	$4b_{(3)} + 2b_{(4)} + b_{(9)}$
1 1 0 x x	$8 + b_{(2)}$	$4b_{(3)} + 2b_{(4)} + b_{(5)}$	$4b_{(0)} + 2b_{(1)} + b_{(9)}$
1 1 1 0 0	$8 + b_{(2)}$	$8 + b_{(5)}$	$4b_{(0)} + 2b_{(1)} + b_{(9)}$
1 1 1 0 1	$8 + b_{(2)}$	$4b_{(0)} + 2b_{(1)} + b_{(5)}$	$8 + b_{(9)}$
1 1 1 1 0	$4b_{(0)} + 2b_{(1)} + b_{(2)}$	$8 + b_{(5)}$	$8 + b_{(9)}$
1 1 11 1	$8 + b_{(2)}$	$8 + b_{(5)}$	$8 + b_{(9)}$

Table 2.4: Decoding 10-bit Densely Packed Decimal to 3 Decimal Digits

Decoding Densely Packed Decimal

Table 2.3 decodes a declet, with 10 bits b(0) to b(9), into 3 decimal digits d(1), d(2), d(3). The first column is in binary and an “x” denotes “don’t care”. Thus all 1024 possible 10-bit patterns shall be accepted and mapped into 1000 possible 3-digit combinations with some redundancy.

$d_{(1,0)}$, $d_{(2,0,)}$, $d_{(3,0)}$	$b_{(0)}$, $b_{(1)}$, $b_{(2)}$	$b_{(3)}$, $b_{(4)}$, $b_{(5)}$	$b_{(6)}$	$b_{(7)}$, $b_{(8)}$, $b_{(9)}$
0 0 0	$d_{(1,1:3)}$	$d_{(2,1:3)}$	0	$d_{(3,1:3)}$
0 0 1	$d_{(1,1:3)}$	$d_{(2,1:3)}$	1	0, 0, $d_{(3,3)}$
0 1 0	$d_{(1,1:3)}$	$d_{(3,1:2),}$ $d_{(2,3)}$	1	0, 1, $d_{(3,3)}$
0 1 1	$d_{(1,1:3)}$	1, 0, $d_{(2,3)}$	1	1, 1, $d_{(3,3)}$
1 0 0	$d_{(3,1:2),}$ $d_{(1,3)}$	$d_{(2,1:3)}$	1	1, 0, $d_{(3,3)}$
1 0 1	$d_{(2,1:2),}$ $d_{(1,3)}$	0, 1, $d_{(2,3)}$	1	1, 1, $d_{(3,3)}$
1 1 0	$d_{(3,1:2),}$ $d_{(1,3)}$	0, 0, $d_{(2,3)}$	1	1, 1, $d_{(3,3)}$

1 1 1	0,0, $d_{(1,3)}$	1, 1, $d_{(2,3)}$	1	1, 1, $d_{(3,1,3)}$

Table 2.5: Encoding 3 Decimal Digits to 10-bit Densely Packed Decimal

Encoding Densely Packed Decimal

Table 2.4 encodes 3 decimal digits d(1), d(2), and d(3), each having 4 bits which can be expressed by a second subscript d(1,0:3), d(2,0:3), and d(3,0:3), where bit 0 is the most significant and bit 3 the least significant, into a declet, with 10 bits b(0) to b(9). Computational operations generate only the 1000 canonical 10-bit patterns defined by table 2.2c.

The 24 noncanonical patterns of the form 01x11x111x, 10x11x111x, or 11x11x111x (where an “x” denotes “don’t care”) are not generated in the result of a computational operation. However, as listed in table 2-3, these 24 bit patterns do map to valid numbers. The bit pattern in a NaN significand can affect how the NaN is propagated.

Rounding

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format while signaling the inexact exception. Every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the modes in this section.

The rounding modes affect all computational operations that might be inexact. The rounding modes may affect the signs of zero sums, and do affect the thresholds beyond which overflow and underflow are signaled.

Rounding Modes to Nearest

In these modes However an infinitely precise result with magnitude at least b^{emax} ($b - ½\ b^{1-p}$) shall round to ∞ with no change in sign; here *emax* and *p* are determined by the destination format unless overridden by a rounding precision mode

Round to Nearest, Ties to Even

An implementation of this standard shall provide round to nearest, ties to even, as the default rounding mode. In this mode the representable number nearest to the infinitely precise result shall be delivered; if the two nearest representable numbers bracketing an unrepresentable infinitely precise result are equally near, the one with it's an even least significant digit shall be delivered.

Round to Nearest, Ties Away from Zero

A decimal implementation of this standard shall provide round to nearest, ties away from zero, as a user-selectable rounding mode. In this mode the representable number nearest to the infinitely precise result shall be delivered; if the two nearest representable numbers bracketing an unrepresentable infinitely precise result are equally near, the one with larger magnitude shall be delivered.

Directed Rounding Modes

An implementation shall also provide three other user-selectable rounding modes: the directed rounding modes are:

Round toward +∞: When rounding toward +∞ the result shall be the format's representable number (possibly +∞) closest to and no less than the infinitely precise result.

Round toward –∞: When rounding toward –∞ the result shall be the format's representable number (possibly –∞) closest to and no greater than the infinitely precise result.

Round toward 0: When rounding toward 0 the result shall be the format's representable number closest to and no greater in magnitude than the infinitely precise result.

Rounding Precision

Normally, a result is rounded to the precision of its destination. However, some systems deliver arithmetic results only to destinations wider than their operands. On such a system the user, which may be a high-level language compiler, shall be able to specify that a result be rounded instead to any supported narrower precision with only one rounding, though it may be stored in a wider format with its wider exponent range.

Chapter 3

Architecture and Implementation

Background

Before proceeding with the architecture and the implementation, a quick overview about the floating point representation motivation, properties and computation problem is introduced.

Floating Point Number representation

Motivation and Terminology

The problem with fixed point arithmetic is the lack of dynamic range, which can be illustrated by the following example in the decimal number system.
Assuming that there are four decimal digits. Then the dynamic range 9999 to 0 is ≈ 10,000. This rang is independent of the decimal point positions, that is, the dynamic range of 0.9999 to 0.0000 is also ≈ 10,000. Since this is 4-digits number, we may want to represent during the same operation both 9999 and 0.0001; but is impossible to do in fixed point arithmetic without scaling.

The above example illustrates the motivation for floating point representation: dynamic range.

Floating point representation is similar to scientific notation; that is

$$\text{Fraction x (radix)}^{\text{exponent}}$$

For example the number 9999 is expressed as 0.9999 X 104. In a computer with floating point instructions, the radix is implicit, so only the fraction and the exponent need to be represented explicitly.

The floating point format for the above four decimal digits could be like this:

exponent fraction

Properties of floating point Representation

Lack of Unique Representation

Generally, a floating point number is evaluated by the equation $M \times ß^e$
where

M = mantissa

$ß$ = radix

e = exponent

In a 5-digit decimal floating point representation, the number 9 can be written as 0.9 X 101 or as 0.09 X 102 . The lack of unique representation makes comparison of numbers difficult. Consequently, floating point numbers are usually represented in normalized from, where the mantissa is always represented by a nonzero most significant digit. Obviously, this rule could not apply to the case of zero. Therefore, by definition, normalized zero is represented by all zero digits (which simplifies zero detection circuitry). It is interesting to note that a normalized zero is floating point representation is designed to be identical to the fixed point representation of the zero.

Range and Precision

The range is a pair of numbers (smallest, largest) which bounds all representable numbers in a given system. Precision, on the other hand, indicates the smallest difference between the mantissas of any two such representable numbers.

The largest number representable in any normalized floating point system is approximately equal to the radix raised to the power of the most positive exponent, and the absolute value of the smallest nonzero number is approximately equal to the radix raised to the power of the most negative exponent.

Assuming M_{max} and expmax to be the largest mantissa and exponent respectively, we write the largest representable number as:

$$max = M_{max} \times \beta^{exp}{}_{max}$$

Similarly, we get the minimum representable number min from the minimum normalized mantissa M_{min} and the minimum exponent exp_{min}:

$$min = M_{min} \times \beta^{exp}{}_{min}$$

For a given radix, the range is mainly a function of the exponent. By contrast, the precision is a function of the mantissa. Precision is the resolution of the system, and it indicates the minimum difference between two mantissa representations, which is equal to the value of the least significant bit of the mantissa. Precision is defined independently of the exponent; it depends only on the mantissa and is equal to the maximum number of significant digits representable in a specific format. In the IBM short format, there are 24 bits in the mantissa. Therefore, the precision is six hexadecimal digits because $16^{-6} = 2^{-24}$. If we convert this to human understandable numbers $2^{-24} \approx 0.6 \times 10^{-7}$, or approximately seven significant decimal digits.

In the literature, some prefer to express the precision as the difference between two consecutive mantissas so that in the previous example, it would be 16^{-6} and not six.

Floating Point Addition and Subtraction

Addition and subtraction require that exponents of the two operands be equal. This alignment is accomplished by shifting the mantissa of the smaller operand to the right, while proportionally increasing its exponent until it is equal to exponent of the larger number. (In general scientific notation, the alignment could be accomplished by the converse operation, that is, shift the mantissa of the larger number left, and while decreasing it is exponent. However, this is impossible in normalized floating point system, since a left-shifted normalized mantissa has to be larger than1, but 1- $ß\text{-}^{p}$ is the largest representable *p*–digit mantissa). After the alignment, the two mantissa are added (or subtracted), and the resultant number, with the common exponent, is normalized. The latter operation is called postnormalization.

Problems in Floating Point Computations

Loss of Significance

The following example illustrates the loss of significance problem. Assume the two numbers are different by less than 2^{-24}. (The representation is the IBM System 370 short format.)

$$A = 0.100000 \text{ X } 16^1$$
$$B = 0.FFFFFF \text{ X } 16^0$$

When one is subtracted from the other, the smaller must be shifted right to align the radix points. (Note that the least significant digit of B is now lost.)

$$A = 0.100000 \text{ X } 16^1$$
$$\underline{B = 0.FFFFFF \text{ X } 16^1}$$
$$A - B = 0.000001 \text{ X } 16^1 = .1 \text{ X } 16^{-4}$$

Now let us calculate the error generated due to loss of digit in the smaller number. The result is (assuming infinite precision):

$$A = 0.100000 \text{ X } 16^1$$
$$\underline{B = 0.FFFFFF \text{ X } 16^1}$$
$$A - B = 0.000001 \text{ X } 16^1 = .1 \text{ X } 16^{-5}$$

$$\text{ERROR} = 0.1 \text{ x } 16^{-4} - 0.1 \text{ x } 16^{-5} = 0.F \text{ x } 16^{-5} \cdot$$

Thus, the loss of significance (error) is $0.F \text{ x } 16^{-5}$. An obvious solution to this problem is a guard digit, that is, additional bits are used to the right of the mantissa to hold intermediate results. In the IBM format, an additional 4 digit (one hexadecimal digit) are appended to the 24 bits of the mantissa. Thus with a guard digit the above example will produce no error. On first thought, one might think that in order to obtain maximum accuracy it is necessary to equate the number of guard bit to the number of bits in the mantissa. However, it has proven that two guard digits are always sufficient to preserve maximal

accuracy. Regardless of operation (subtraction and multiplication are the operation of concern), only one nonzero bit can be left-postshifted into the result mantissa. Thus, no more than one guard digit will enter the final significant result. However, to insure an unbiased rounding, a third digit (sticky digit) can be added beyond the two guard digits [7].

Design Specification

The target design has to fulfill the following specifications:

- Decimal Adder/Subtractor unit.
- Single path.
- 64bits.
- Support 5 rounding modes.
    - Round to zero.
    - Round to +∞.
    - Round to -∞.
    - Round to nearest
        - Ties to even.
        - Ties away from zero.
- Support exception handling by raising a flag.
    - Invalid.
        - Any operation on a signaling NaN except those operations defined to be quiet.
        - Magnitude subtraction of infinities, such as (+∞) + (-∞).
    - Overflow.
    - Inexact.

Unit Interface

The unit interface as shown in figure 3.1 has 2 input operands with 64-bit wide, one bit (sign_in) indicating the operation to be performed, two inputs for clock and reset signals and finally 3 bits specifying the rounding mode that will be applied on the intermediate final result. As output of this unit we have the result as 64-bit wide in the standard format in addition to 3 flags for exception handling (inexact, overflow and invalid).

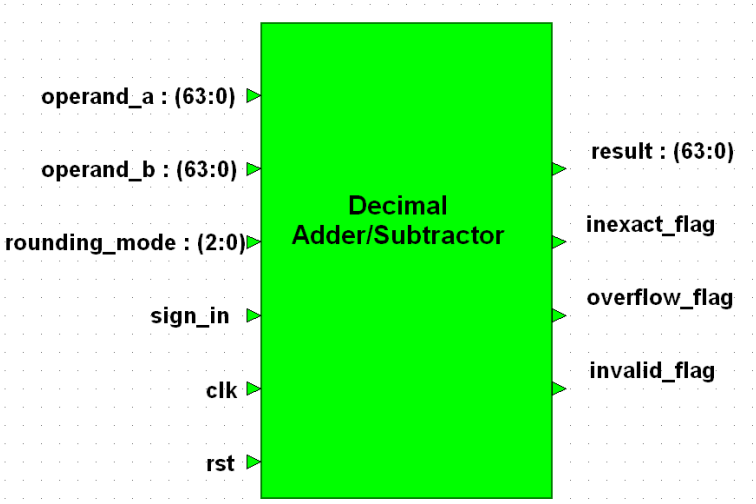

Figure 3.1: Unit Interface

Internal representation

In our design we used the single path technique and we tried different internal design for some blocks that will be explained in details. A block diagram describing our 64-bit decimal floating-point adder/subtractor design is shown in Figure 3.2. In which, the two input operands are decomposed from the Densely Packed format to extract the sign, the significand and the exponent fields of each operand. The two significands are then transformed to BCD.

With the operation specifier and the sign of each operand the effective operation is then deduced.

The two significands are then aligned to have the same exponent to be added or subtracted according to the effective operation.

The result of the BCD adder is shifted and adjusted according to the rounding mode specifier.

After the calculation of the exponent field it is adjusted in accordance with the shifting done on the BCD result.

The sign of the result is calculated in parallel with the BCD result and the exponent of the result, and then all the fields are repacked into Densely Packed format with all the exception handling and raising the appropriate flags.

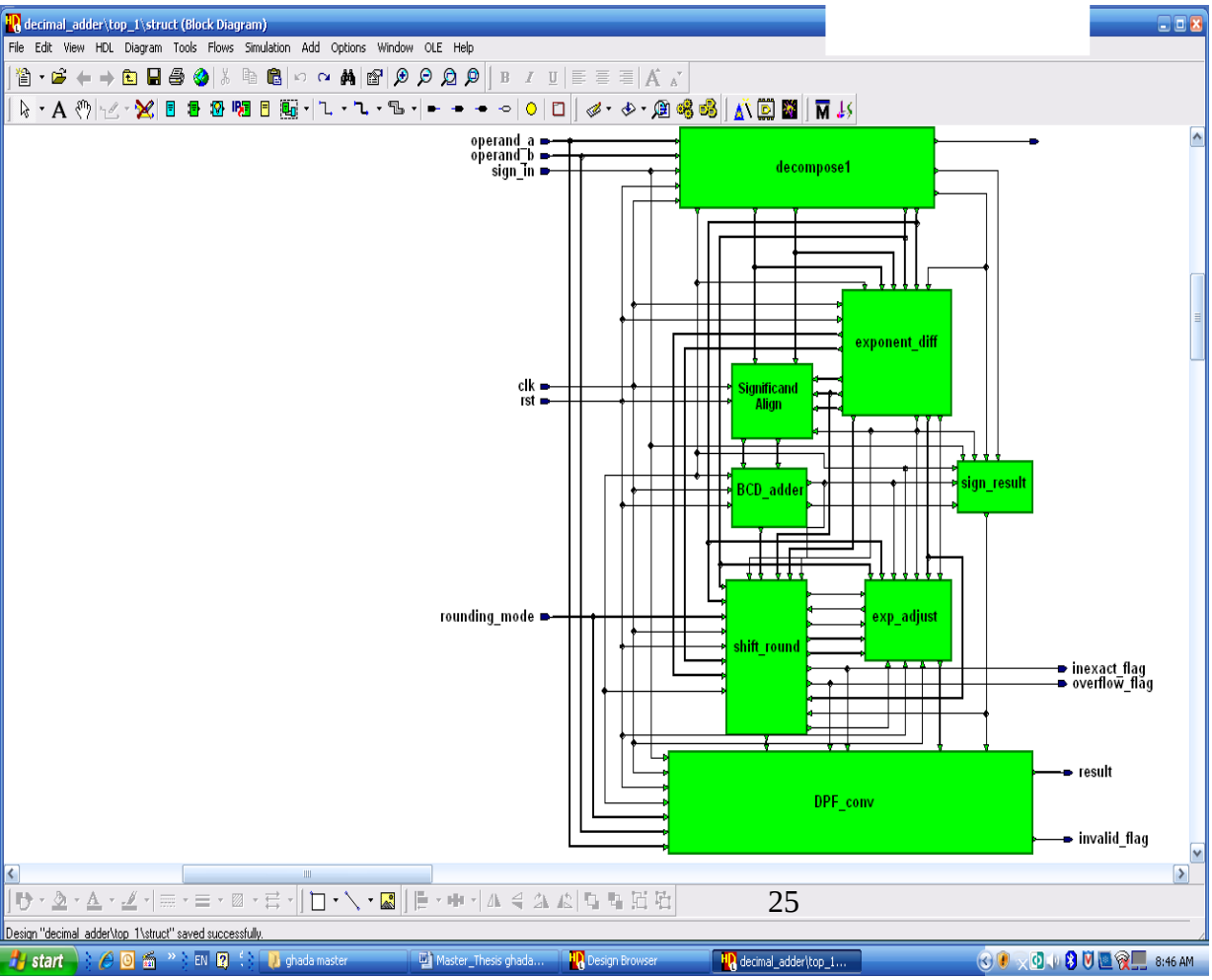

Figure 3.2: Block Diagram

Decompose

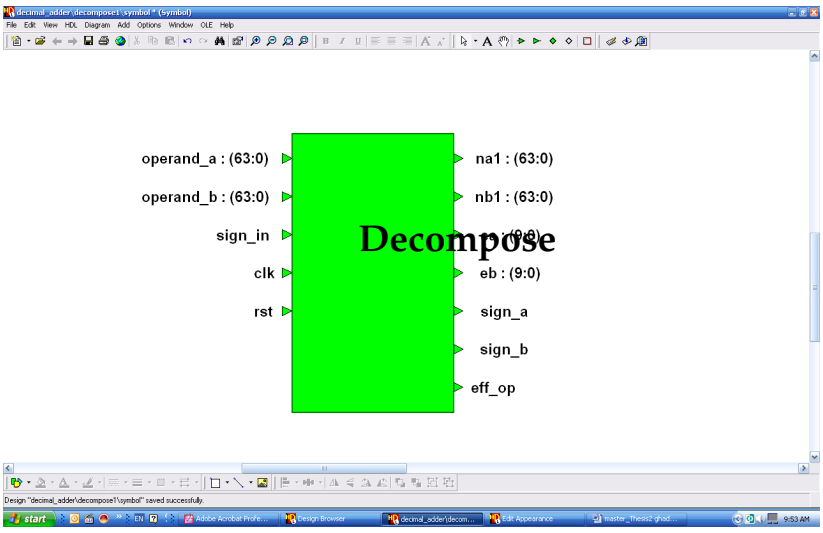

Figure 3.3: Decompose Interface

The two IEEE-754 decimal encoded numbers (operand_a and operand_b) are unpacked into their corresponding sign-bits (sign_a and sign_b), 10-bit biased binary exponents (ea and eb), and 16-digit significands.

Each 64-bit operand has the format shown in table 3.1, which consists of a 1-bit sign field, an 8-bit exponent continuation field, a 50-bit coefficient continuation field, and a 5–bit combination field. The combination field is decoded and combined with the exponent and coefficient continuation fields to determine the operand's exponent and coefficient, respectively.

Length (bits)	1	5	8	50
Contents	Sign	Combination Field	Exponent continuation	Coefficient continuation

Table 3.6: Densely Packed Decimal-64 Operand Format.

With the sign_in signal and the deduced sign of each operand the effective operation is then deduced according to the following equation:

$$Eff\_op = sign\_in \oplus sign\_a \oplus sign\_b.$$

The two unpacked operands are decoded from Densely Packed Format (DPF) (54 bits) to their corresponding 64 bits (16 digits) in BCD format (na1 and nb1) table 3.2.

Length (bits)	1	10	64
Contents	Sign	Exponent	Coefficient

Table 3.7: BCD Operand Format

Exponent Difference

Figure 3.4 shows the interface of this block in which the two input BCD operands (na1 and nb1) are checked to calculate the number of leading zeros in each (na_zero, nb_zero). At this step we will internally calculate a signal called "effective exponent" which represents the difference between the exponent and the number of leading zeros for each operand.

With the number of leading zeros and the effective exponent we deduce the larger operand which will be placed on (NA2) and if we have equal effective

exponent we assume that operand_a is the larger.

In order to align the 2 operands, calculation for the amount the larger operand has to be shifted left (left_amount) as well as the amount the small operand has to be shifted right should (right_amount) also be calculated.
In case the small operand has a larger exponent then it has to be shifted left (left_small_amount) in order to align the 2 operands.

Internal initial calculation for the result exponent (er_int_out) is done inside this block based on the calculated large operand, shift amounts and the initial exponents for both operands.

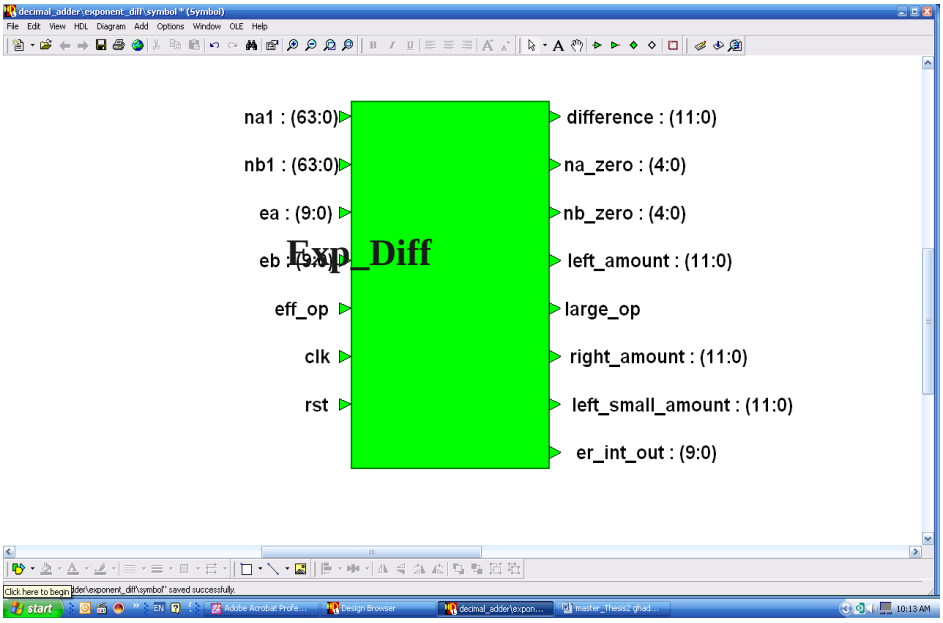

Figure 3.4: Exponent Difference Interface

The exponent difference block calculates the amount of shift for each of the two BCD significand values so that their corresponding exponents are equal. It determines the largest value by which NA1 can be shifted to the left, thus decreasing its exponent towards the value of the lesser exponent without encountering a loss of information. This is done in accordance with the following formula:

$$Left\_amount = min\ \{EA - EB,\ X - M\} \qquad (1)$$

Where(EA – EB) is the exponent difference of the 2 operands, M is the index of the most significant non-zero digit of NA1, and X is the index of the most significant digit available for the operand (for our 16-digit implementation, X = 16) on the condition of being positive number.

In parallel with this, it is also determined if and by how much NB must be shifted to the right or left in order to complete the alignment process. This is done in accordance with the following formulas:

$$Right\_amount = max\ \{EA - EB + M - X,\ 0\} \qquad (2)$$

$$left\_small\_amount = max\ \{EB - EA + X - M,\ 0\} \qquad (3)$$

Once the left and right shift amounts are computed, the significand that is associated with the larger exponent (NA1) is shifted to the left up to the edge of its available register space to guarantee no loss in the accuracy of the result. At the same time, the operand with the smaller exponent (NB1) is shifted to the right until the two significands have associated exponents that are equal. This shift does not affect the result unless non-zero digits are shifted out of the 64-bit (16-digit) significand field. In this case these digits are shifted through the round digit, guard digit and sticky bit, which are later used for rounding. Fig. 3.5 shows the adder operation and result format.

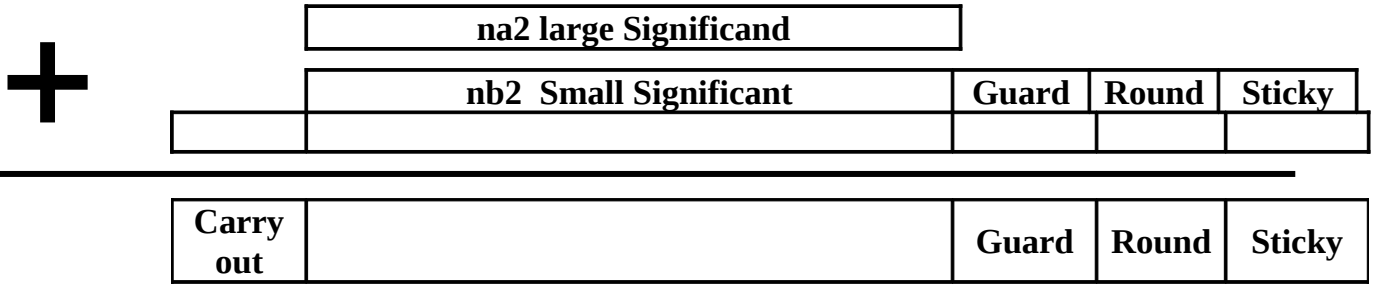

Figure 3.5: Adder operation and result format.

An example that illustrates the workings of the significand alignment procedure is provided in the following example:

Example1. Illustrating significand alignment:

Definitions:

na1 → input significand A (associated with ea1)

nb1 → input significand B (associated with eb1)

ea1 → input exponent A (ea1 >= eb1)

eb1 → input exponent B (eb1 < ea1)

Input values:

na1 = 0786 0000 0000 0000

nb1 = 0000 0000 0004 3720

ea1 = 6

eb1 = 0

Taking into account the available significand round and guard digits and the sticky bit, the two input significands are shown below (also refer to Figure 3.7):

na1 = 0786 0000 0000 0000

nb1 = 0000 0000 0004 3720 00

Using equation (1), it can be found that na2 must be left-shifted one digit:

Left_amount = min {6 – 0, 16 – 15} = 1

In parallel, equation (2) can be used to determine the right-shift amount for nb2:

Right_amount = max {6 – 0 + 15 – 16, 0} = 5

In parallel, equation (3) can be used to determine the left_small_amount shift for nb2:

Left_small_amount = max {0 - 6 +15– 16, 0} = 0

Given these shift amounts, the two significands and their associated exponents are adjusted to become the following:

NA2 = 7860 0000 0000 0000
NB2 = 0000 0000 0000 0000 4372 0000 0
Er_int_out = 5 (common exponent)

Example2: As example for the shift left small amount, note the following case in which the input values are:

na1 = 0000 0023 0786 0000
nb1 = 0000 0000 0000 0004
ea1 = 7
eb1 = 10

Here we should start first by calculating the effective exponent (eff_exp) for both operands.

eff_exp_a = ea1 – na_zero = 7 – 5 =2

eff_exp_b = eb1 – nb_zero = 10 – 15 = -5

So it's clear that operand_a is the larger and we should align both operands to have a common exponent of 2 so using equation (1), it can be found that NA1 must be left-shifted six digits:

Left_ amount = min {7 – 10, 16 – 10} = 6

Right_amount = max {7 -10 + 10-16, 0} = 0

Left_small_amount = max {10 – 7 +16 – 10, 0} = 9

Given these shift amounts, the two significands and their associated exponents are adjusted to become the following:

na2 = 2307 8600 0000 0000
nb2 = 0000 0040 0000 0000 0000 0000 0
Er_int_out = 1 (common exponent)

Significand Alignment

This block is responsible for shifting the two input operands (na1, nb1) with the amount calculated by the previous block (Exponent difference). So that, depending on the large operand value it places the larger operand on na2 and the smaller operand on nb2. It adds to the smaller a guard digit, a round digit and a sticky bit to keep some of the digits whenever a shift to the right is done. The guard digit, round digit and the sticky bit are used later for rounding purposes.

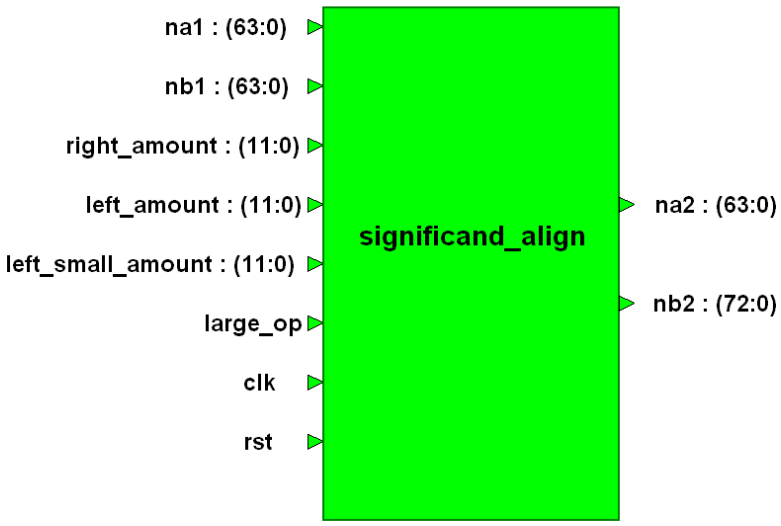

Figure 3.6: Significand Alignment Interface

BCD Adder

The adder block is the most critical block for the overall delay of the design. So, we tried 2 internal design for the adder itself and 2 different designs for the subtractor.

We will represent first the ripple carry adder in which we are using the nine's-complement for subtraction as shown in fig.3.7. In which the two standard BCD operands are added/subtracted after being aligned in the previous step.

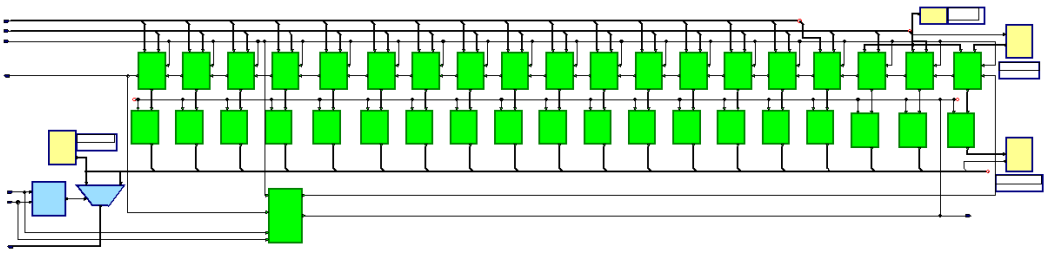

Figure 3.7: BCD Adder/Subtractor.

The 2 input operands na2 and nb2 are 64-bit and 73-bit wide respectively. First, the sticky bit in nb2 is extended to be one digit. Second, in order to be able to add or subtract the 2 operands, the na2 is also extended by 3 digits which are all zeros in order to have same length for both operands. So that now we have to add/subtract 19 digits using our BCD adder.

Also, after getting the intermediate result including the end around carry, we need to check whether the intermediate result has to be complemented or not depending on effective operation and the end around carry.

Each subblock of the 19 identical blocks in first row of fig 3.7 is a BCD adder/subtractor cell which will be illustrated in details as follows.

Adder cell

Each subblock has two 4-bit input operands (inp_a) and (inp_b), an input carry (cin) from the previous stage and an operation specifier (operation). It generates a sum vector of 4 bits (sout) and a carry out signal (cout) as shown in fig.3.8.

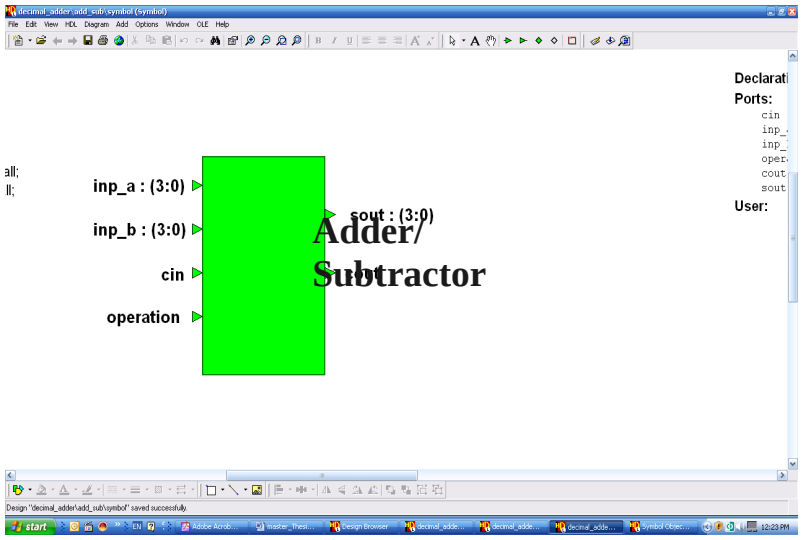

Figure 3.8: BCD Adder/Subtractor cell

We have implemented the subtractor by 2 different designs, the nine's-complements and the ten's-complements. We are using here the nine's-complements which will be explained in details. Operand B is fed into a nine's complement block to prepare it, then according to the operation specifier signal this operand is kept as it is or we get its nine's complement to be fed to the BCD adder with input_a which generates the output (sout) according to equation (3) and the carry (cout) according to equation (4).

Sout = inp_a XOR inp_b XOR cin……(3)

Cout = (cin AND (inp_a OR inp_b)) OR (inp_a AND inp_b)……(4)

Carry effect

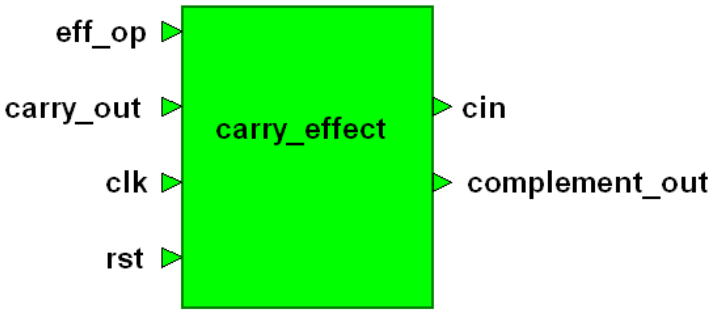

Figure 3.9: carry effect block interface

This block is responsible of generating the input carry to the LSD cell in the adder block as well as detecting whether the generated output should be complemented or not.

If the effective operation is addition, then the cin and complement_out signals are equal to zeros. While, in case of effective subtraction if the end

round carry (carry_out) is generated this means that the result is positive and we should generate cin to be equal to '1' which is fed to the LSD and no complementation is needed for the output.

In case of effective subtraction and no carry is generated this will only occur in case we had both operand having the same effective exponent and we assumed that operand-a is the larger which was not correct. So, we got a negative result (carry_out='0') in which case we should complement the output by raising the complement_out signal.

Nine's complement

The 9's complement of a decimal number as shown in fig.3.10 can be found by subtracting each digit in the number from 9 as shown in table 3.3.

Figure 3.10: nine's complement block interface

DECIMAL DIGIT	9's COMPLEMENT
0	9
1	8
2	7

: :	: :
9	0

Table 3.8: 9's Complenment

Example: 9's COMPLEMENT of 28 = 99 –28 = 71

9's COMPLEMENT of 562 = 999 –562= 437

Subtraction of a smaller decimal number from a larger one can be done by adding the 9's complement of the smaller number to the larger number and then adding the carry to the result (end round carry)[6].

When subtracting a larger number from a smaller one, there is no carry and the result is in 9's complement form and negative.

Examples:

(a)

```
 +8          +8
 -3          +6  ← 9's COMP. OF 3
  5      (1) 4
         ┌→ +1   END AROUND CARRY
         └   5
```

(b)

```
 54          54
-21          78  ← 9's COMP. OF 3
 33      (1) 32
         ┌→ +1   END AROUND CARRY
         └  33
```

(c)

```
 15          15
-28         +71  ← 9's COMP. OF 3
-13          86  → -13
```

NO CARRY >>> NEGATIVE RESULT

86 – **99** = – **13**

Figure 3.11: shows the uncorrected and the corrected BCD sums.

So, from figure 3.11 we can deduce some general rules to follow in case of subtraction:

1- Add 9's complement of b to a
2- If the result >9 correct by adding 0110.
3- If most significant carry is produced [i.e.=1]then the result is positive and the end around carry must be added.
4- If most significant carry is not produced [i.e.=0]then the result is negative and we get the 9's complement of the result.

Correction unit

A correction unit is embedded with each cell of the adder. This unit is responsible of correcting the calculated result. When adding two BCD digits the obtained result may be ranged between (0 -18). It is not allowed to have a calculated decimal numbers greater than 9. Only numbers between 0 and 9 are allowed in order to have the correct BCD code.

DECIMAL DIGIT	UNCORECTED BCD SUM $C'_3 S'_3 S'_2 S'_1 S'_0$	9, COMPLEMENT BCD SUM $C_3 S_3 S_2 S_1 S_0$
0	0 0 0 0	0 0 0 0
:	:	:
:	:	:
9	1 0 0 1	1 0 0 1
10	1 0 10	1 0 0 0 0
11	1 0 1 1	1 0 0 0 1
12	1 1 0 0	1 0 0 1 0
13	1 1 0 1	1 0 0 1 1
14	1 1 1 0	1 0 1 0 0
15	1 1 1 1	1 0 10 1
16	1 0 0 0 0	1 0 1 1 0

17	1 0 0 0 1	1 0 1 1 1
18	1 0 0 1 0	1 1 0 0 0
19	1 0 0 1 1	1 1 0 0 1

Table 3.9: BCD sum correction

Thus, for sums between 10 and 18 we must subtract 10 and produce a carry, Subtracting 10 means by other words adding its 2's complement. So, by adding 0110 the result will be correct.

Also, for answers between 0 and 3 we should check if a carry is produced or not. If a carry is produced this means that the answer is between 16 and 19, and then we must correct the output in the same manner as previous.

Ten's complement

The ten's-complement block interface is shown in fig 3.12. The ten's-complements of a BCD number is obtained by adding '1' to the nine's complement of the overall result. In other words, in case of effective subtraction we put the cin_compl of the LSD equal to '1'. The "cout_compl" of each cell is fed to the following one.

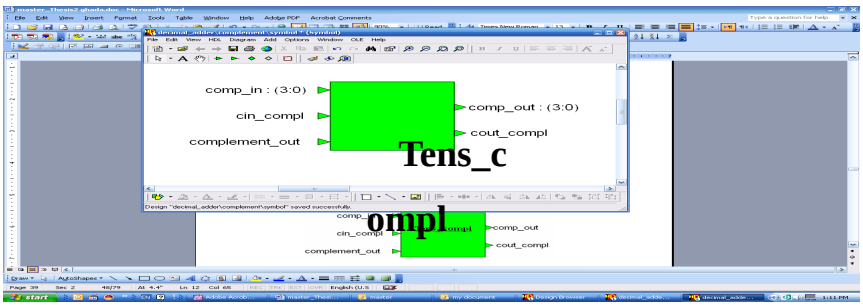

Figure 3.12: nine's complement block interface

The main advantage of the ten's-complement over the nine's-complement is that we don't have to wait for the end round carry to get the correct result. But the main disadvantage is that we have to wait for the carry to ripple to the MSD to get the correct answer.

We shall evaluate the behavior in the synthesis time to decide which one is convenient for the overall system performance.

We introduced another system architecture in order to avoid the waiting for the rippling of the carry. As shown in fig.3.13 we added another BCD-adder block and a selector.

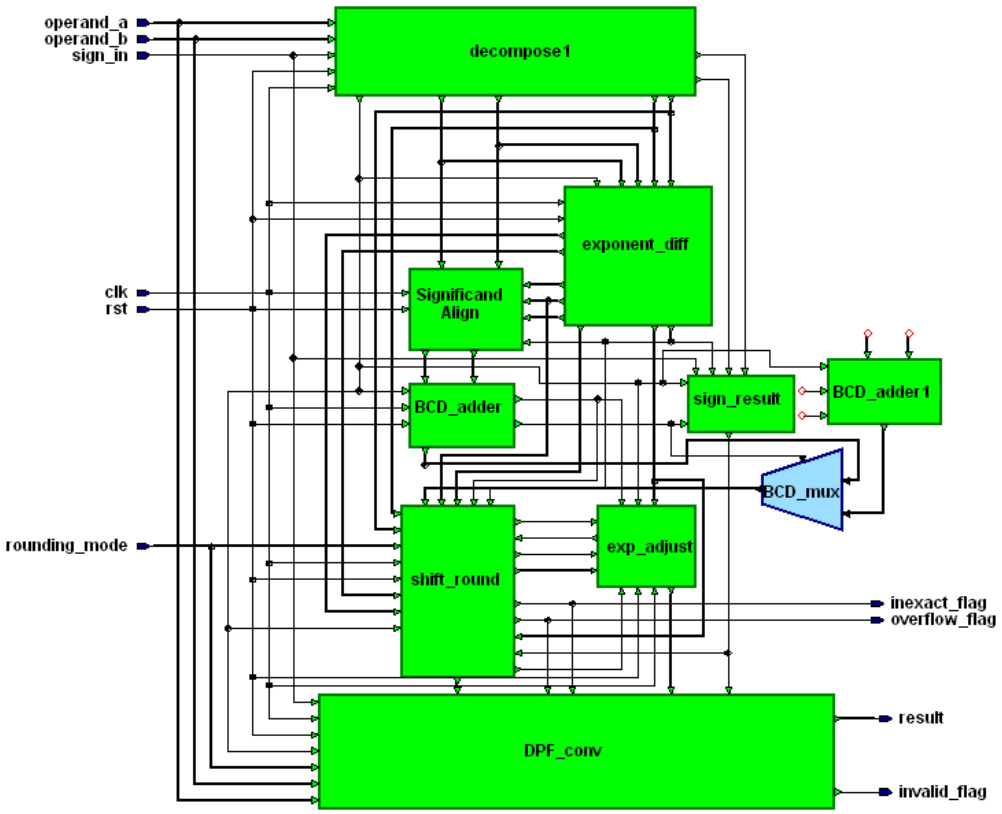

Figure 3.13: Alternative block diagram

In which the second BCD-adder has the 2 input operands interchanged. So that we have at the same time a block is subtracting na1 from nb1 and the other is subtracting nb1 from na1.

According to the complement-out signal the selector will select which output shall be delivered to the next stage.

We introduced another design for the BCD adder which is the carry_look_ahead architecture [7] as shown in figure 3.14.

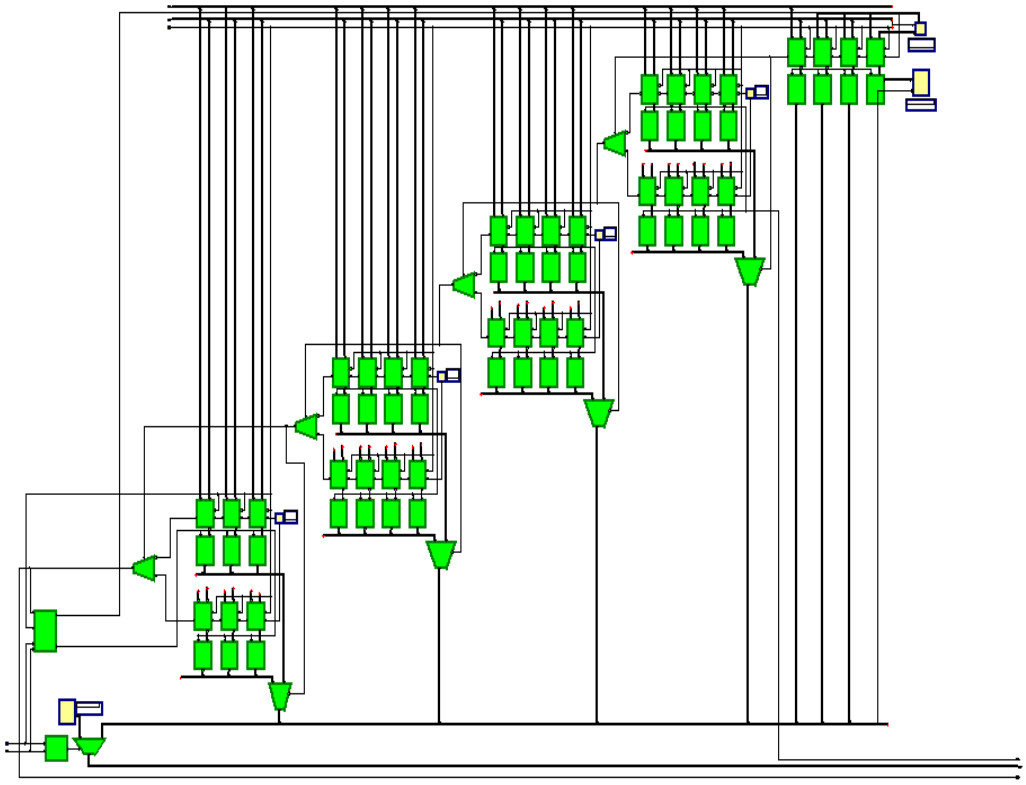

Figure 3.14: Carry_look_ahead adder block diagram

In the last decade, the carry-look-ahead has become the most popular method of addition, due to a simplicity and modularity that make it particularly adaptable to integrated circuit implementation. To see this modularity, we derive the equations for a 4-bit slice[7].

The sum equations for each bit position are:

$$\left.\begin{aligned} S0 &= A0 \oplus B0 \oplus C0 \\ S1 &= A1 \oplus B1 \oplus C1 \\ S2 &= A2 \oplus B2 \oplus C2 \\ S3 &= A3 \oplus B3 \oplus C3 \end{aligned}\right\} \text{ in general: } Si = Ai \oplus Bi \oplus Ci$$

The carry equations are as follows:

$$\left.\begin{aligned} C1 &= A0B0 + C0(A0 + B0) \\ C2 &= A1B1 + C1(A1 + B1) \\ C3 &= A2B2 + C2(A2 + B2) \\ C4 &= A3B3 + C3(A3 + B3) \end{aligned}\right\} \text{ in general: } Ci{+}1 = AiBi + Ci(Ai + Bi)$$

In this adder design, instead of waiting for the end around carry, we grouped each 4 digits together and duplicate it one time assuming the carry from the previous stage is '1' and the other time assuming the carry from the previous stage is '0'. We have two multiplexers, one to select which carry should be passed to the next stage and the other selects the 4-digit output that will be fed to the final output.

At the end, the ripple carry adder and the carry look ahead will be compared with the whole design from both point of view, area and speed.

Sign result

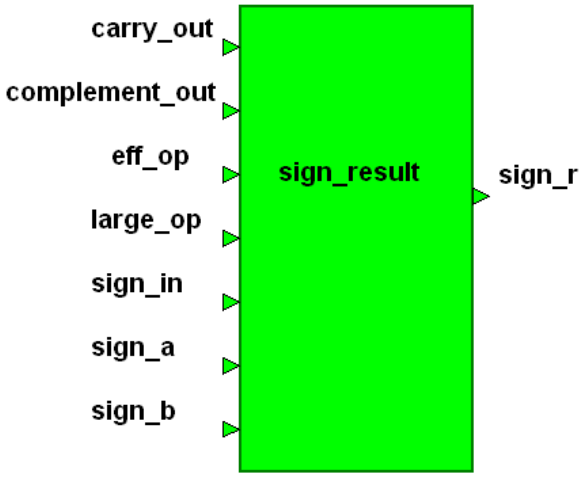

Figure 3.15: result sign block interface

Figure 3.15 shows the interface of the block responsible for generating the sign of the final result. The sign of the result depends mainly on the effective operation. In case of effective addition the sign of the result always follows operand_a sign as shown in table 3.5.

INPUTS			OUTPUTS	
SA	**SB**	**Operation**	**Effective Operation**	**Sign Result**
+	+	Add	Add	+ = Sign A
+	+	Sub	Sub	TBD
+	–	Add	Sub	TBD
+	–	Sub	Add	+ = Sign A
–	+	Add	Sub	TBD
–	+	Sub	Add	- = Sign A
–	–	Add	Add	- = Sign A
–	–	Sub	Sub	TBD

Table 3.10: sign result

TBD: to be deduced

In case of effective subtraction we should check whether there is output complementation in the BCD adder block or not. If operand_b is the larger operand or when "complement_out" signal is generated while the large operand is operand_a, then the sign of the result is according to the following equation:

Sign_r = sign_in XOR sign_b.

Otherwise, the sign of the result is equal to sign_a.

Exp adjust

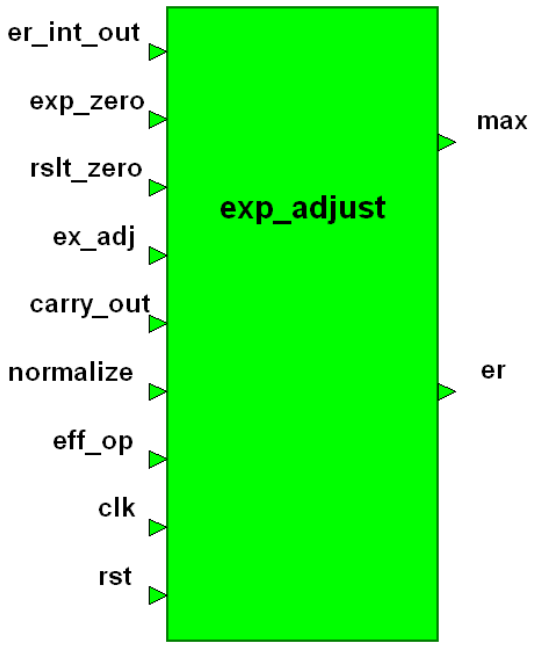

Figure 3.16: exponent adjust block interface

Figure 3.16 shows the interface of the exponent adjust block. The final result exponent in addition to an alert signal for max exponent is calculated within this block.

The previously calculated exponent (er_int_out) within the exponent difference block is adjusted according to the effect of shift and round step. The adjustment may be by increasing or decreasing the previously calculated exponent.

The rising of the input "normalize" signal indicates the generation of a carry out signal after performing the necessary shift during the effective addition operation. This means that a shift to the right to the complete final result has been done in order to keep the generated carry out within the final result. At this condition we should increment the previously calculated exponent by one.

Another adjustment is required, whenever the "exp_zero" signal is raised we should decrease the previously calculated exponent by the amount of the "rslt_zero " signal. Because at this condition, there was leading zeros in the final result and the amount of exponent calculated allows for shift while keeping the final exponent as required by the standard

The preferred exponent is min (Q(x), Q(y)) [3].

Where Q(x) and Q(y) are the exponents of operand_a and operand_b respectively.

Whenever the previously calculated exponent (er_int_out) is equal to the max (767) with effective addition and either a carry_out (from the BCD adder) is generated or an ex_adj (from shift and round) then, "max" signal is raised and the "er" signal is equal to zeros (which is the condition of overflow).

Shift & Round

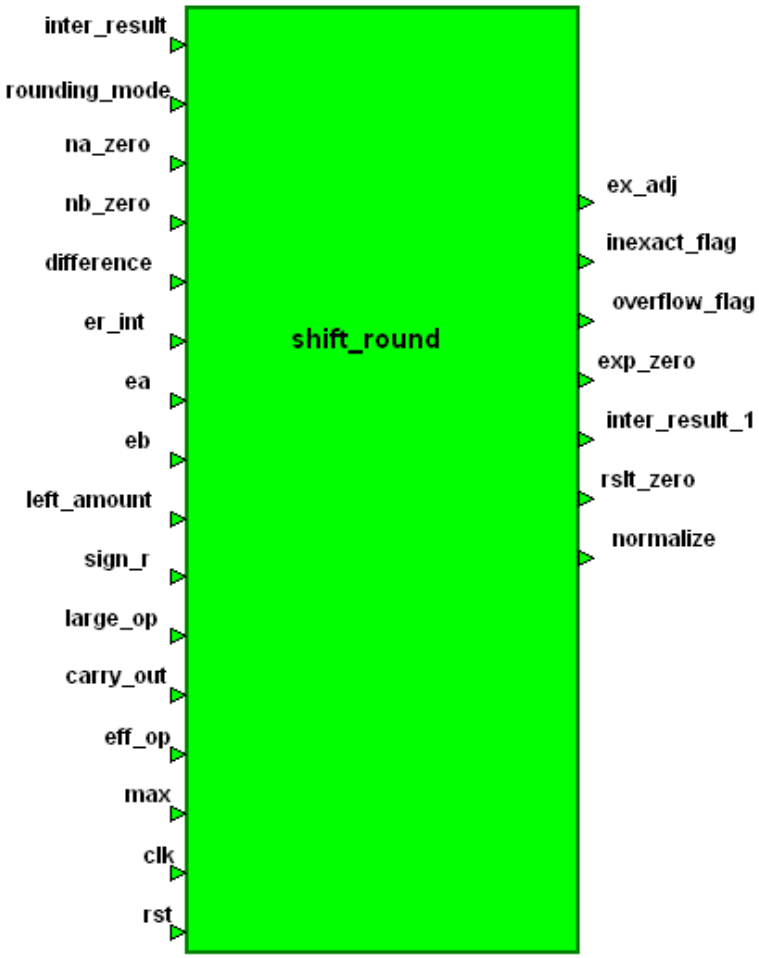

Figure 3.17: shift and round block interface

Figure 3.17 shows the interface of the shift and round block. This block is responsible for the followings:

- Calculate the intermediate result"inter_result_1" after shifting and rounding operation.

- Raise "ex_adj" when there's a need to adjust the exponent.

- Raise the "inexact" and "overflow" flags when their appropriate conditions are available.

- Raise a signal "exp_zero" when a shift to the left has to be performed and send the amount of this shift to the exponent adjust block via "rslt_zero" signal.

- Raise a signal "normalize" when a shift to the right has to be performed. To discuss in details the internal structure, consider the fig.3.18 which represents the internal structure of this block. We are going to elaborate each block separately.

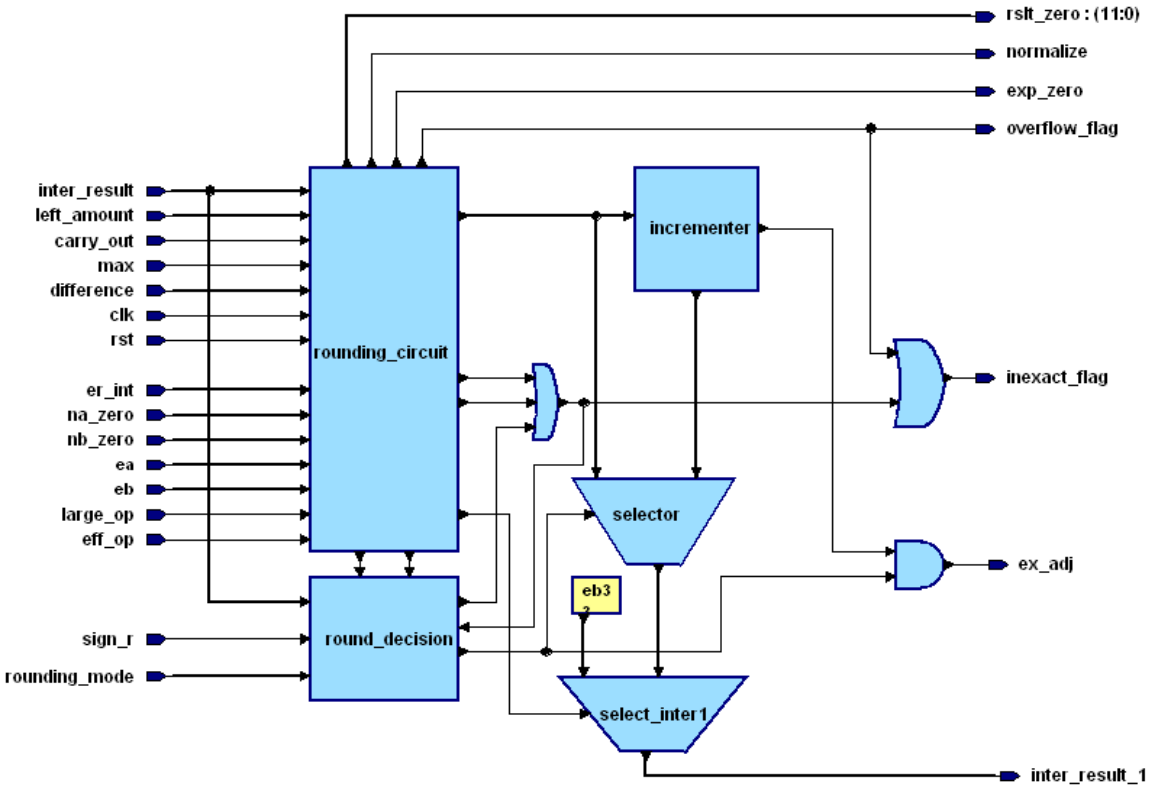

Figure 3.18: shift and round internal structure

Rounding Circuit

The output of the BCD adder "inter_result" is fed to this block in order to check for the number of leading zeros and then check the intermediate exponent (er_int) if it is equal to the is minimum of the exponents of the 2 operands then no shift will be performed. Otherwise shift the whole result to the left and raise the "exp_zero" signal and put the amount to be shifted in the "rslt_zero" variable which is equal to the number of leading zeros.

In case we have the first 2 least digits are non-zero, we raise the "inexact_flag" signal.

The "round_flag" is raised whenever the round digit is greater than "4" or when the round digit is equal to "4" and the guard digit is greater than "4".

If the round digit is equal to 5 and the guard digit and the sticky bit are equal to zero then the tie signal is raised.

The generation of the "carry out" signal from the BCD adder is fed into this block which in case of effective addition will generate the "normalize" signal indicating a shift to the right for the complete intermediate result will be performed by one digit place.

In case we have the "max" signal raised, then accordingly, the "overflow_flag" is raised which will raise the "inexact_flag" as reference to the standard.

Round Decision

In this block we are adopting the following five rounding modes stated by the standard.

Round towards zero

Round towards zero never increments the digit prior to a discarded fraction, that is, truncates. This rounding mode never increases the magnitude of the calculated value. Some references call it round down as shown in figure 3.19.

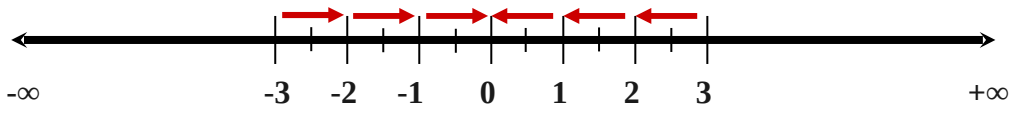

Figure 3.19: round towards zero

Round towards positive infinity

Also is called round ceiling. If the decimal is positive, the output value is incremented (it behaves as for round away from zero); if negative, the output value is not incremented (it behaves as for round towards zero). This rounding mode never decreases the calculated value as shown in figure 3.20.

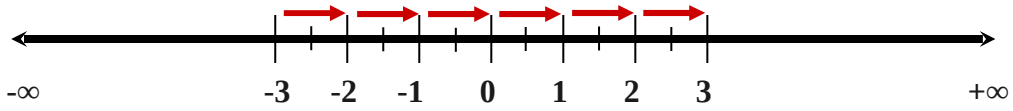

Figure 3.20: round towards positive infinity

`

Round towards negative infinity

Also is called round floor. If the decimal is positive, the output value is not incremented (it behaves as for round towards zero); if negative, the output value is incremented (it behaves as for round away from zero). This rounding mode never increases the calculated value as shown in figure 3.21.

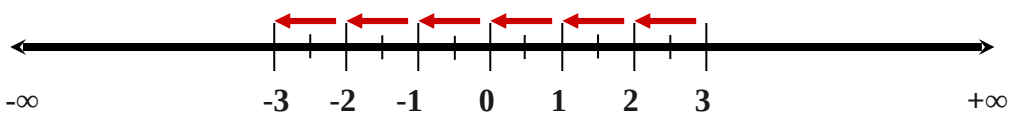

Figure 3.21: round towards negative infinity

Round to nearest, tie to even

Round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor. If the digit to the left of the discarded fraction is odd then, the output value is incremented (it behaves as for round half up); if it is even, the output value is not incremented (it behaves as for round half down). This is the rounding mode that minimizes cumulative error when applied repeatedly over a sequence of calculations, and is sometimes referred to as Banker's rounding as shown in figure 3.22.

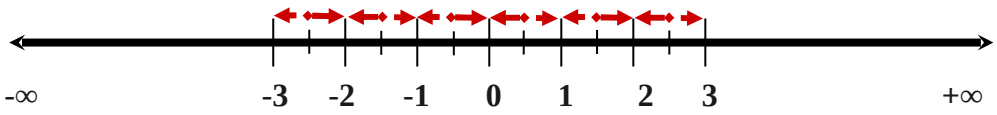

Figure 3.22: round to nearest, tie to even

Round to nearest, away from zero

Round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up. , the output value is incremented (it behaves as for round towards positive infinity) if the discarded fraction is greater than, or equal to, 0.5; otherwise, the output value is not incremented (it behaves as for round

towards negative infinity). This is the rounding mode that is typically taught in schools as shown in figure 3.23.

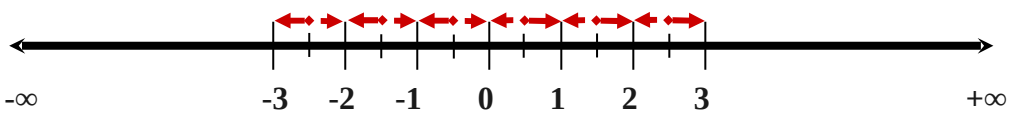

Figure 3.23: round to nearest, away from zero

In addition to the previously mentioned rounding modes, following are two other rounding modes proposed by IBM.

Round away from zero

The output always increments the digit prior to a nonzero discarded fraction. This rounding mode never decreases the magnitude of the calculated value as shown in figure 3.24.

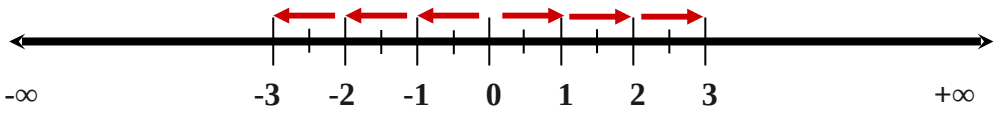

Figure 3.24: round away from zero

Round half down

Round towards "nearest neighbor" unless both neighbors are equidistant, in which case the output value is not incremented (it behaves as for round towards negative infinity). If the discarded fraction is grater than 0.5 the output value is

incremented (it behaves as for round towards positive infinity) as shown in figure 3.25.

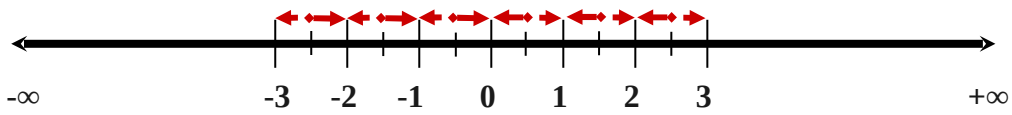

Figure 3.25: round half down

Table 3.6 summarizes the 7 implemented rounding modes

Inputs		**Outputs**								
Round flag	**Sticky bit**	**To 0**	**Toward + ∞**		**Toward - ∞**		**To Even**	**Away from 0**	**Half Up**	**Half Down**	
0	0	0	0		0		0	0	0	0	
0	1	0	+ve	-ve	+ve	-ve	0	+1	0	0	
			+1	0	0	+1					
1	0	0	+1	0	0	+1	Check LSB	+1	+1	T=1	T=0
										0	+1
1	1	0	+1	0	0	+1	+1	+1	+1	0	+1

Table 3.11 rounding table

Table 3.7 shows some examples according to the different rounding modes [10].

Input number	Round away from zero	Round toward zero	Round toward + ∞	Round toward - ∞	Round ties to even	Round half up	Round half down
5.5	6	5	6	5	6	6	5
2.5	3	2	3	2	2	3	2
1.6	2	1	2	1	2	2	2
1.1	2	1	2	1	1	1	1
1.0	1	1	1	1	1	1	1
-1.0	-1	-1	-1	-1	-1	-1	-1
-1.1	-2	-1	-1	-2	-1	-1	-1
-1.6	-2	-1	-1	-2	-2	-2	-2
-2.5	-3	-2	-2	-3	-2	-3	-2
-5.5	-6	-5	-5	-6	-6	-6	-5

Table 3.12 rounding table

Table 3.8 shows the internal code corresponding for each rounding mode

Round mode	Code
Round to Nearest ties to Even	000
Round away from zero	001
Round Toward Positive	010
Round Toward Negative	011
Round Toward Zero	100
Round-half-up	101
Round-half-down	110

Table 3.13 Rounding codes

Incrementer

This block increments the output of the rounding_circuit by one and generates (if needed) a carry out flag which will be anded with the round signal in order to generate the "ex_adj" signal.

The round signal selects whether the output of the rounding circuit will be passed as is to the output "inter_result_1"or the incremented value instead.

DPF converter

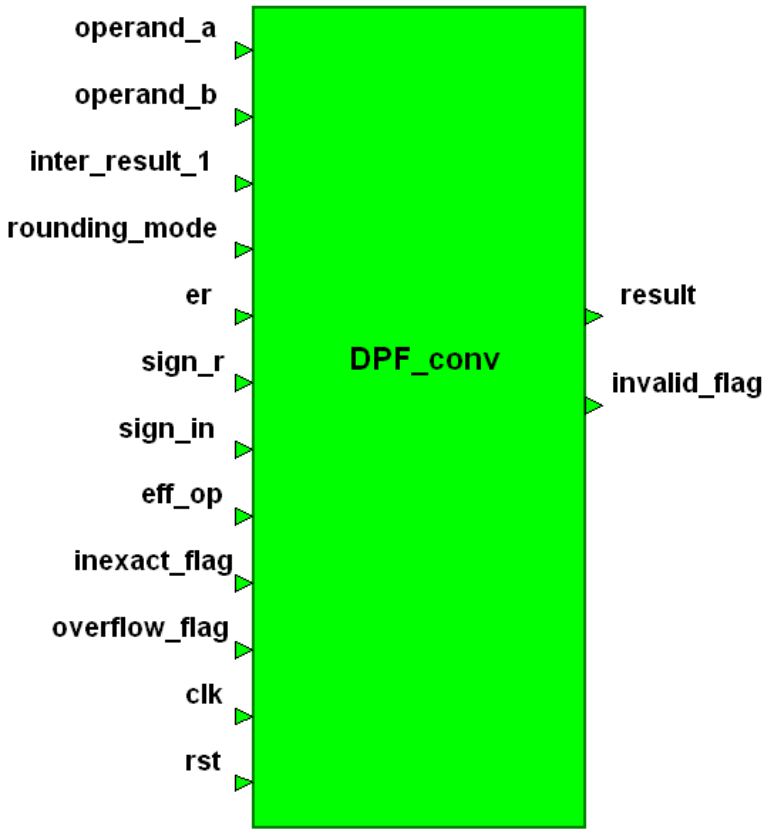

Figure 3.26: Densely Packed Format Converter

The last block in our architecture is shown in figure 3.26. This block is responsible for adjusting the final result and put it in the "Densely Packed Format" as well as raising the invalid flag.
The internal implementation of this block transforms the BCD input "inter_result_1" into its corresponding DPF. There are several checks that should be done before passing this value to the output.

1- The 2 input operands are fed to this block in order to check for if any of the 2 operands is a sNaN at which case the output is as shown in table 3-9. Also, the invalid flag is also raised.

Width	**1 Bit**	**5 Bits**	**8 Bits**	**50 Bits**
Field	Sign S	Combination G	Following Exponent F	Trailing signficand T Containing J declets
Most/least significant bit	0	11111	000…0	0000…0

Table 3.14 output in case of sNaN

2- If any of the 2 operands is a qNaN, we have the same output as the previous case but without raising the invalid flag.

2- In case of having operand_a equals to infinity then I need to be sure that the other operand is neither infinity nor sNaN, in that case the result is operand_a and invalid flag is not raised. But if the other operand is infinity then I need to check if the effective operation is addition then the final result is again operand_a and the invalid flag is not raised. Otherwise (effective subtraction) the result is qNaN and invalid flag is not raised.

The same procedure is followed in case of having operand_b equals to infinity except that the sign of the result is equal to the XOR of the input sign and the operand_b sign.

4- In case of overflow the final result is either zeros or the maximum value depending on the rounding mode and the sign of the result.
In case of Rounding toward zero *OR* rounding toward + infinity with effective subtraction *OR* rounding toward - infinity with effective addition the result is the maximum as shown in table 3.10.

Width	**1 Bit**	**5 Bits**	**8 Bits**	**50 Bits**
Field	Sign S	Combination G	Following Exponent F	Trailing signficand T Containing J declets
Most/least significant bit	Sign_r	11110	111…1	00111111110011111111001111111110011111111100111111

Table 3.15 output in case of infinity

Otherwise:

Width	**1 Bit**	**5 Bits**	**8 Bits**	**50 Bits**
Field	Sign S	Combination G	Following Exponent F	Trailing signficand T Containing J declets
Most/least significant bit	Sign_r	11101	000…0	000…0

Table 3.16 output in case of infinity

5- In case of effective subtraction, exact result and the whole operand is zero, a check to the rounding mode is mandatory. If we have rounding

toward negative then the sign of the result is negative and the rest is as calculated by the combination field, the follow_expo_64 and trailing_sig_64.

Chapter 4

4-Verification &Testing

Test plan

To test the design we are following the IBM test suite [8]. 3063 test cases were applied to the design covering the five standard rounding modes:

- Round to nearest ties to even (000)
- Round away from zero (001)
- Round toward positive (010)
- Round toward negative (011)
- Round toward zero (100)

As well as the two following testing modes:

- Round half up (101): in which if the round digit is greater than 4 then round up, otherwise keep the result as is.
- Round half down (110): in which if the round digit is greater than 5 then round up, otherwise keep the result as is.

First, a test bench to test each case separately was implemented in order to study each problem individually.

Second, a behavioral test bench has been implemented to read the input test vectors from a file with the following format as shown in table 4.1.

- The sign in (one bit)
- The rounding mode (3 bits)
- The 2 operands (each 64 bits in DPF)

Finally, write the final result in another file.
Using special software, we compare the original output file from IBM with the output from our design.

Table 4.17 Input test vector format

Problems:

During testing, we faced some problems which we had solved to fulfill the required standard output. Following are some of these solved problems.

1- In case of effective subtraction, I use to leave the round and guard digits as they are and if a carry in (cin) is generated it was fed to the LSD before the round, guard & sticky bit.

Sol: the sticky bit should be expanded as 4 bits in order to feed the cin to its input so that any change in the intermediate adder result (either by adding a carry at the input or taking the 9's complement of the result) will affect the whole result not just a part of it

2- While specifying the larger operand I used to depend only on the value of the exponent and in case of equal exponent I assumed operand_a is the larger.

Sol: to specify the larger operand first I should calculate the number of leading zeros in each operand then compare it with its exponent to calculate what so called the effective exponent which is so far follow the following equation:

Effective exponent = the original exponent - Leading zero.

3- After specifying the larger exponent, if a shift operation has to be performed then the larger operand shall be shifted left and the small operand shall be shifted right (when needed). A problem appeared when the small operand has originally larger exponent in which case the result exponent should be that of the larger operand.

Sol: in this particular case a shift left to the small operand has to be performed in order to adjust the final result.

Example: If operand_a = 0000 1456 2345 0023 with exponent = 10

operand_b = 0000 0000 2345 0023 with exponent = 12

Here:

The Effective exponent_a = 10 - 4 = 6

The Effective exponent_b = 12 - 8 = 4

So the larger operand (a in this case) should be shifted left 4 digits and the small operand should also be shifted left 6 digits in order to have the same exponent.

4- In case of having leading zero in the final result, I use to shift left the result as much as the exponent allows.

Sol: I should take care of the minimum exponent of two input operands because as stated in the standard section 5.4.1 Arithmetic operations [3]

The preferred exponent is min(Q(x), Q(y)).

So, I should not go beyond the minimum of the 2 exponent even if the exponent and the number of leading zeros of the result allows this.

5- I use to calculate the maximum allowed exponent in an incorrect way as emax+ ebias which is in case of 64 bits(**Table 2-2a**) operands is:

384 + 398 = 782.

Sol: It shows that I should remove the number of digits of the precision (-1) which in our case is 15 so the maximum exponent. After which overflow occurs is emax + ebias – 15 = 782 -15 = 767.

6- The "inexact flag" was raised whenever only round or guard or sticky are not equal to zero, this was an incomplete condition.

Sol: After checking the rounding mode and the rounding condition, in some cases where the rounding condition is fulfilled I have to add "1" to the intermediate which affects the final result and produce an inexact number. So, I should also check the round, guard and sticky after rounding.

7- The overflow condition was depending only on the exponent if it is 767 and there is carryout then raises the overflow flag.

But, it appeared that this is not the only condition, we should tie this condition with the effective operation (when addition) and also whether a carry is generated as a result from the BCD adder or the signal exp_adj is generated from the shif& round block when a carry out is generated as result of rounding.

8- In case of overflow, I used to raise the overflow flag only, but I realized that whenever there is an overflow the inexact flag is raised.

9- In case of overflow the final result is either zeros or the maximum value depending on the rounding mode and the sign of the result.

In case of Rounding toward zero *OR* rounding toward + infinity with effective subtraction *OR* rounding toward - infinity with effective addition the result is the maximum value which is:

Combination ="11101"

follow_expo_64 = "11111111"

trailing_sig_64="0011111111001111111100111111110011111111001111111"

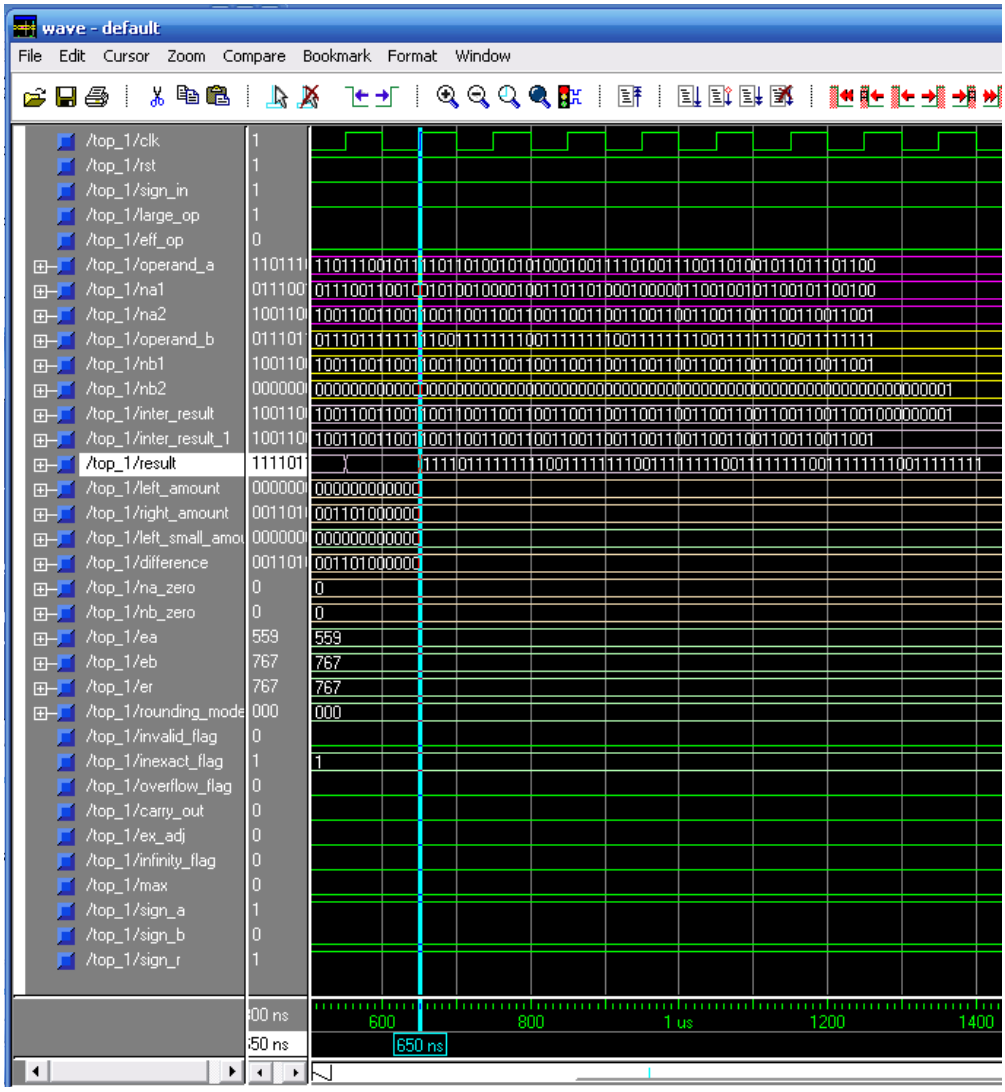

Figure 4.27: Simulation result of overflow case

Otherwise the Combination ="11101", while the rest of the result is all zeros.

10- I use to handle the Quite NAN & Signaling NAN in a same manner. For both I use to raise the invalid flag. But the invalid flag is raised when any of the operands is a signaling NAN (SNAN) in which for any operand bits from (62 downto 57) are all "1" So the result should be in the form "011111" & 58_zeros with the invalid flag rose. Which for the QNAN, the result is the same but without raising the invalid flag.

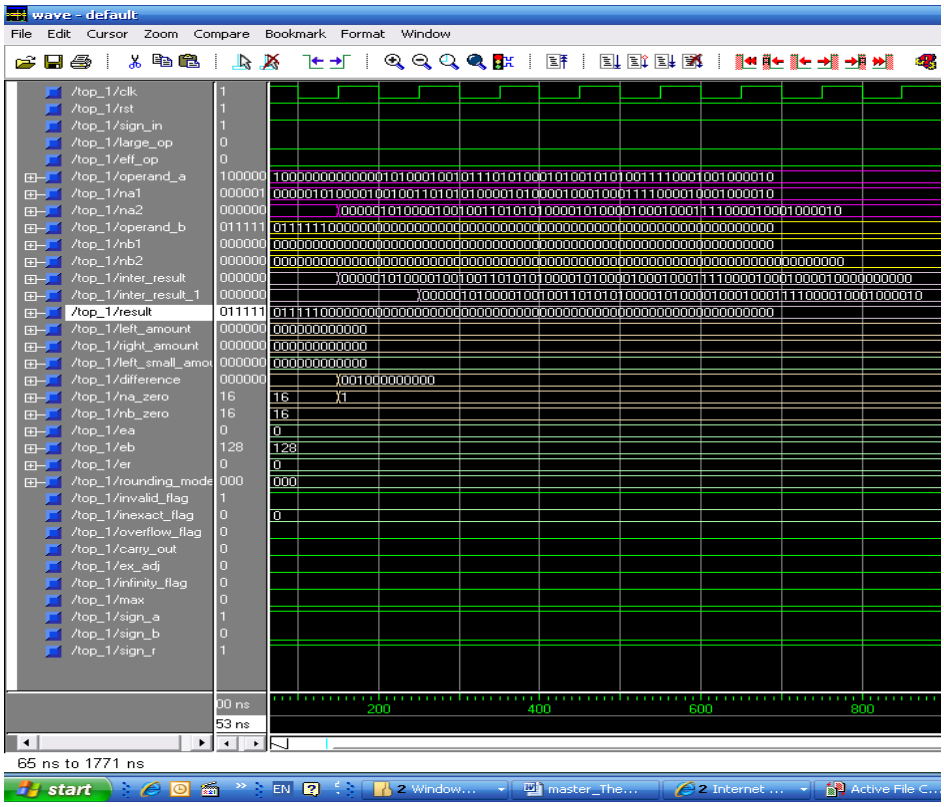

Figure 4.28: Simulation result in case one of the input is SNAN

11- In case of having operand_a equals to infinity then I need to be sure that the other operand is neither infinity nor SNAN, in that case the result is operand_a and invalid flag is not raised. But if the other operand is infinity then I need to check if the effective operation is addition then the final result is again operand_a and invalid flag is not raised. Otherwise (effective subtraction) the result is QNAN and invalid flag is not raised.

The same procedure is followed in case of having operand_b equals to infinity except that the sign of the result is equal to the XOR of the input sign and the operand_b sign.

12- In case of having both inputs equal to zero, I should check their exponent and select the operand with the small exponent to be the final result. In case of effective addition the sign of the result shall be equal to sign operand A irrespective which operand will be delivered to the output. In case of effective subtraction the sign of the result is +ve and the O/P is either operand A or operand B depending on the minimum exponent.

13- In case of one of the exponent is zero and the other is not. I need to check the other operand exponent and if it has the lower exponent then the result is the other operand otherwise I should use the calculated fields with the sign of the result is either the sign of the operand_a (in case of effective addition) or not sign of operand_a (in case of effective subtraction).

14- After calculating the intermediate final result, if there is a carry out produced in case of effective addition so a shift right to the complete final result should be performed with recalculation of the final exponent, round, guard digits as well as the sticky bit.

15- The sign of the final result is always following the sign of operand_a in case of effective addition. In case of effective subtraction the sign of the result depends on the large operand,, the carry out and the complement out signal.

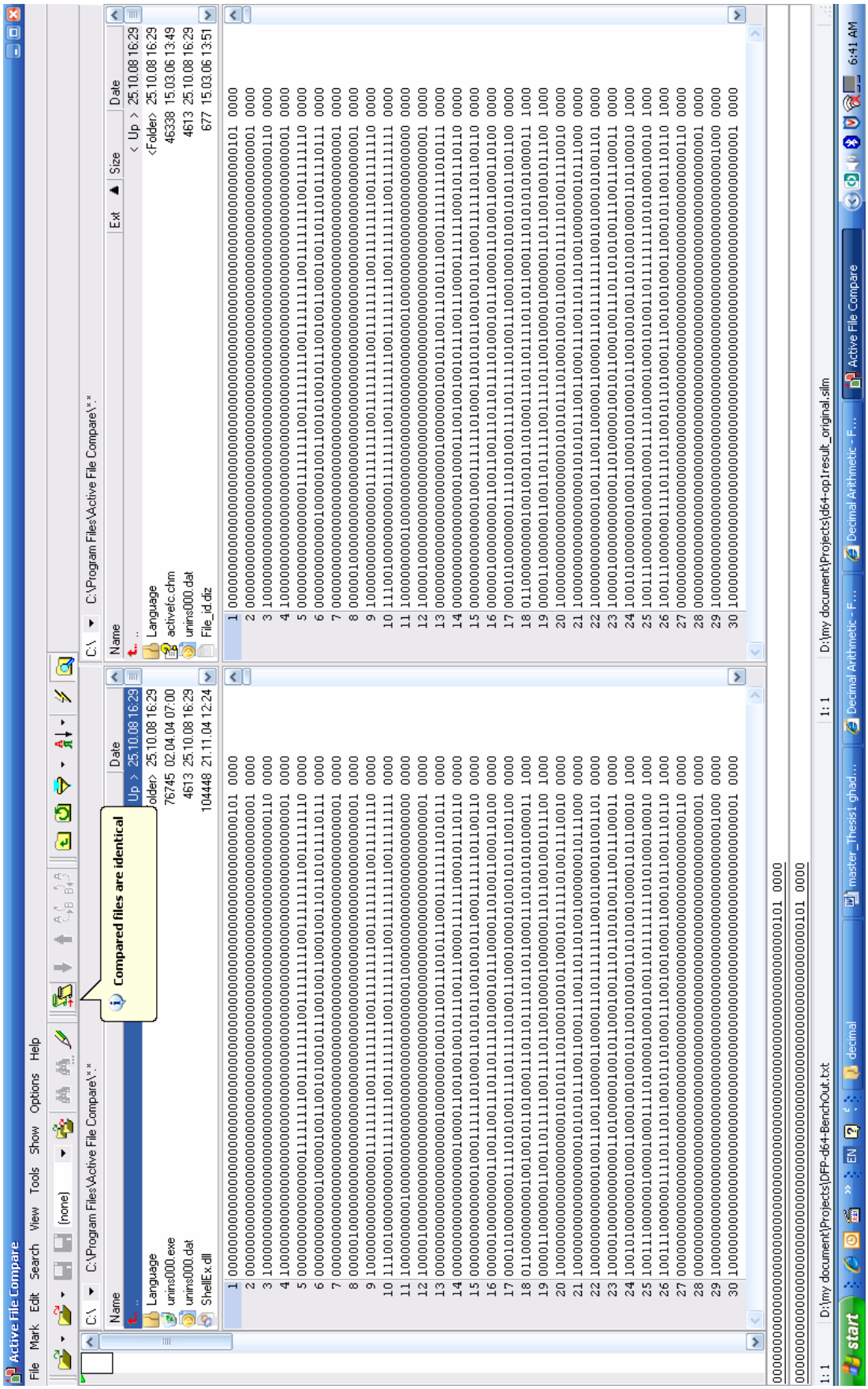

Figure 4.29: Output files comparison

Alternative design for subtraction units:

After solving all the problems of the original design and passing all the test vectors from IBM, another design for the subtractor unit has been implemented. Originally we tried the nines complement design for the BCD subtraction as previously mentioned, now we are introducing the tens complement instead.

The internal design of the adder is as shown in fig.4.4 in which, in case of subtraction the carry_in fed to the full adder and the complement blocks are always '1' since the tens complement is basically the same as the nines complement except that after getting the nines complement we add '1'. Also there's no need to adjust the carry since from the characteristics of the tens complement the carry is automatically adjusted from the forward path and no need to adjust it.

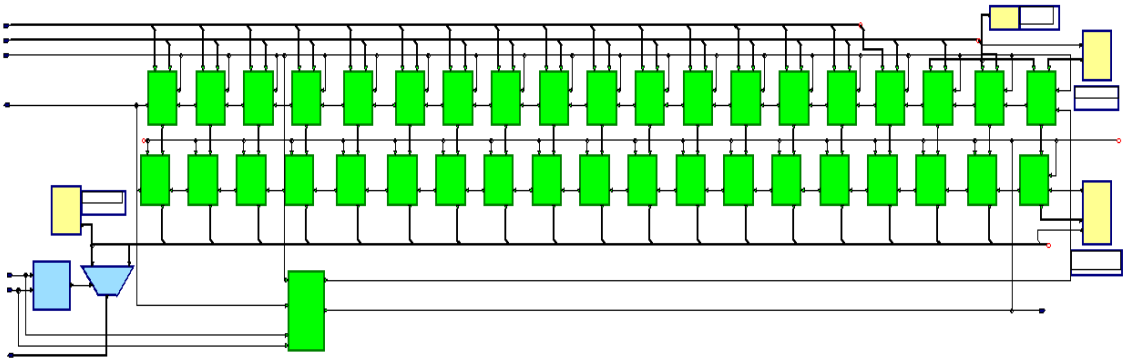

Figure 4.30: BCD adder with tens complement

Synthesis

Synthesizing the decimal adder with Xilinx FPGA for both designs to different families and compare the area and delay reports.

First, we synthesized the design including the ripple carry adder for Spartan II family and we found that the "2s200fg456" chip is the most suitable for the design regarding the number of I/Os and function generators.

Table 4.2 shows the comparison between the nine's and ten's complements from the delay and area point of views. In which it is seen that the nine's complements design runs at higher frequency (almost the double)

	Spartan II 2s200fg456				
	CLK	Area			
		I/O	FG	CLB	DFF
Nines Complement	6.7 MHz	70.14%	75.72%	75.72%	11.08%
Tens complement	3.4 MHz	70.14%	79.83%	79.85%	11.08%

Table 4.18 Delay and area comparison for "2s200fg456"

Second, we synthesized the design for Vertix II family and we found that the "2V500fg456" chip is the most suitable for the design regarding the number of I/Os and function generators.

Table 4.3 shows the comparison between the nines and tens complement from the delay and area point of views. In which it is seen that the nines complement design runs at higher frequency as well as the area is less by a small amount

	Vertix II 2V500fg456				
	CLK	Area			
		I/O	FG	CLB	DFF
Nines Complement	11.2 MHz	76.52%	57.37%	57.39%	8.90%
Tens complement	6.7 MHz	76.52%	60.51%	60.51%	8.90%

Table 4.19 Delay and area comparison for "2V500fg456"

Then, we tried the architecture shown in figure 3.13 in which we added a second adder with interchanged operands and based on the ten's-complements for subtraction. A multiplexer is used to select the output which will be fed to the next block based on the complement_out signal.

Table 4.4 shows the comparison between the nine's and ten's complement in the second architecture from the delay and area point of views for Spartan II family and the "2s200fg456" chip. In which it is seen that the speed of the ten's-complements design has increased by 70 % and its area is also increased by 7.6% by which we conclude that the nines complement design runs at higher frequency as well as using smaller area.

	Spartan II 2s200fg456				
	CLK	Area			
		I/O	FG	CLB	DFF
Nines Complement	6.7 MHz	70.14%	75.72%	75.72%	11.08%
Tens complement	5.8 MHz	70.14%	85.91%	85.93%	11.10%

Table 4.20 Delay and area comparison for "2s200fg456"

Table 4.5 shows the comparison between the nine's and ten's complement in the second architecture from the delay and area point of views for Vertix II family and the "2V500fg456" chip. In which it is seen that the speed of the ten's-complements design has increased by 49.2 % and its area is also increased by 7.6% by which we conclude that the nines complement design runs at higher frequency as well as using smaller area.

	Vertix II 2V500fg456				
	CLK	Area			
		I/O	FG	CLB	DFF
Nines Complement	11.2 MHz	76.52%	57.37%	57.39%	8.90%
Tens complement	10 MHz	76.52%	65.15%	65.17%	8.91%

Table 4.21 Delay and area comparison for "2V500fg456"

Finally, we conclude that using the nines complement for BCD subtraction gives better results as regards the area and the delay.

So, now we try the nine's-complement with another adder architecture, which is the carry look ahead one. We synthesized the design for both families and table 4.6 shows the comparison between the ripple carry adder and the carry look ahead adder for the Spartan II family and the "2s200fg456" chip. In which it is seen that the speed of the carry look ahead design has increased by 56.7 % and its area is also increased by 7.9% by which we conclude that the carry look ahead design runs at higher frequency and the increase in area is negligible.

	Spartan II 2s200fg456				
	CLK	Area			
		I/O	FG	CLB	DFF
Ripple carry	6.7 MHz	70.14%	75.72%	75.72%	11.08%
Carry look ahead	10.5 MHz	70.14%	81.68%	81.68%	11..08%

Table 4.22 Delay and area comparison for "2s200fg456" for two different BCD adder architecture

Table 4.7 shows the comparison between the ripple carry adder and the carry look ahead adder for the Vertix II family and the "2V500fg456" chip. In which it is seen that the speed of the carry look ahead design has increased by 42 % and its area is also increased by 7.9% by which we conclude that the carry

look ahead design runs at higher frequency and the increase in area is negligible.

	Spartan II 2s200fg456				
	CLK	Area			
		I/O	FG	CLB	DFF
Ripple carry	11.2 MHz	76.52%	57.37%	57.39%	8.90%
Carry look ahead	15.9 MHz	70.14%	61.88%	61.88%	8.90%

Table 4.23 Delay and area comparison for "2V500fg456" for two different BCD adder architecture

So, finally it is seen that the carry look ahead adder with the nine's-complements for subtraction gives better results on FPGA as regards the speed.

Chapter 5

5- Similar work Comparison

Preview

Since the IEEE 754r standard for binary and decimal floating point was finally issued on August 2008. Few works have been done on its draft version. We are going to compare our work with some of the work done.

A 64-Bit Decimal Floating-Point Adder

University of Wisconsin Madison

The University of Wisconsin Madison has introduced hardware designs for decimal adder/ subtractor compliant with decimal floating point standard. The first implementation of a 64-bit decimal floating-point adder that is compliant with the draft revision of the IEEE-754 Standard was introduced on 2004 [9]. The design performs addition and subtraction on 64-bit operands with the architecture shown in Fig. 5.1.

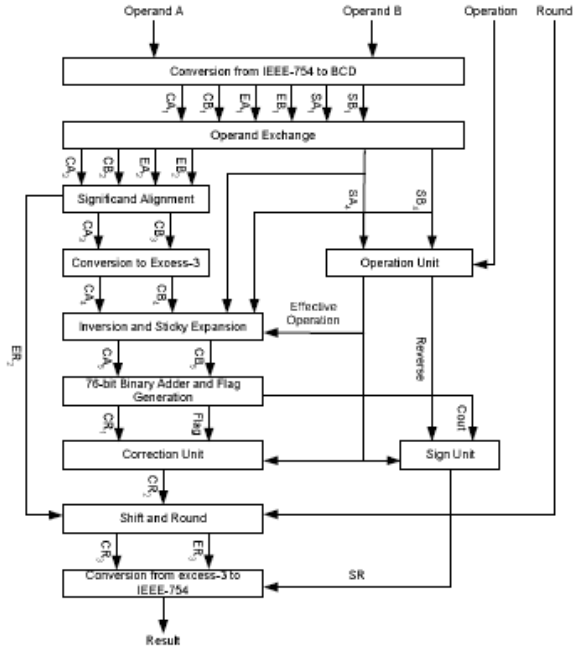

Figure 5.31: university of Wisconsin Madison Architecture

It can be seen from fig.5.1 that from the point of view of the architecture, we are using the same single path technique in the adder implementation with some differences in the internal design. One is that for the adder they are using the excess-3 BCD encoding but we are using the conventional BCD encoding.

One main issue is that for BCD subtraction, nine's complement logic is needed before and after the adder to generate correct results. This approach is used in the IBM S/390 machines. Which is the same as we found after comparing the overall decimal adder as regards the ten's and nine's complement for BCD subtraction.

They introduced some optimization on the design of the decimal adder based on the architecture of fig. 5.1[9] with some modifications [15] as shown in fig.5.2.

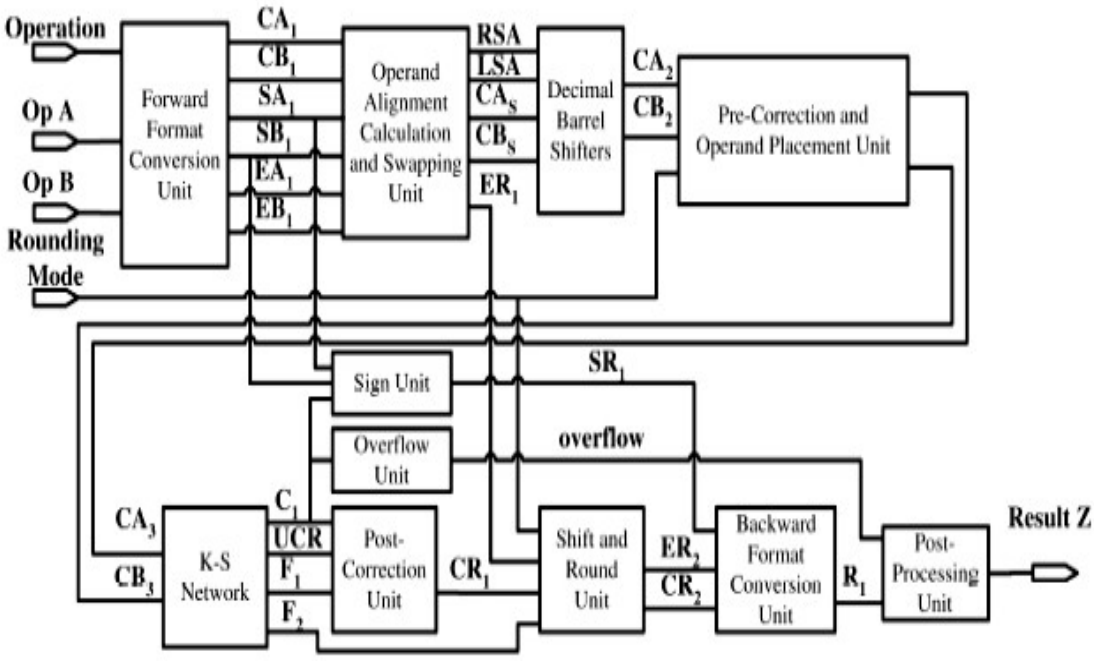

Figure 5.32: university of Wisconsin Madison Architecture

The optimizations include the internal use of the BCD encoding, instead of the excess-3 encoding, which leads to simpler circuitry in the “Precorrection and Operand Placement Unit” and a more efficient placement of the corrected operands for addition and subtraction to simplify the design of the “Shift and Round Unit.”

SilMind Company

Another design was proposed by SilMind Company which has the architecture shown in fig.5.3.

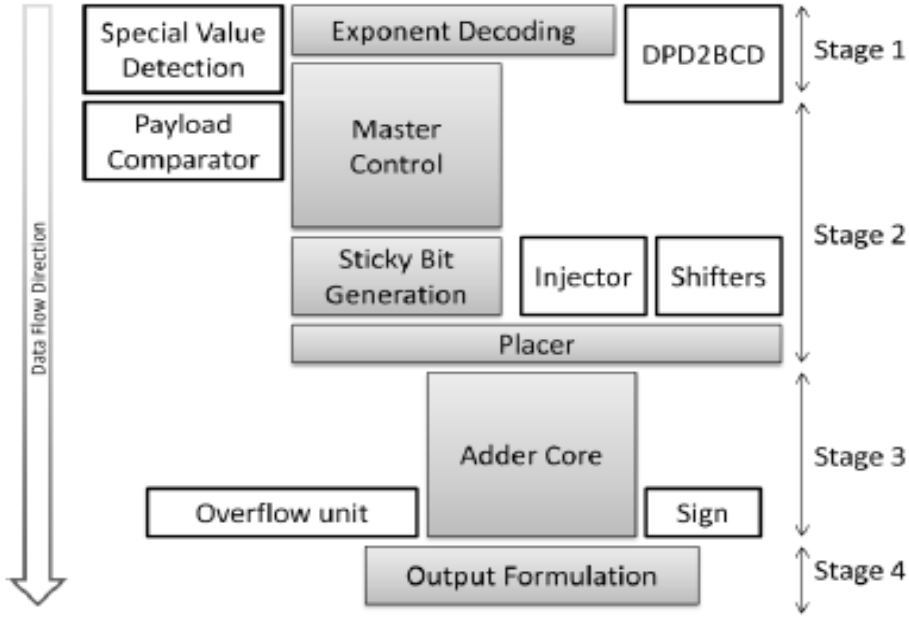

Figure 5.33: SilMind adder design

The proposed design is based on the kogge-Stone parallel prefix network for decimal significand addition and subtraction.

Two hardware implementations were introduced for decimal floating-point adder that is compliant with IEEE 754-2008 standard; one for high speed and the other for low Power/Area.

IBM Company

Chapter 6

6- Conclusions and future work

6.1 Conclusions

In this thesis, a design and implementation of a 64-bit adder/ subtractor compliant to the IEEE-2008 standard for floating point arithmetic has been introduced.

The design performs addition and subtraction on 64-bit operands in a single path adder with exception handling fulfilling the released standard and it can easily be extended to also support operations on 128-bit decimal floating-point numbers.

We introduced 2 different implementations for the BCD-subtractor internal design. The tens complement and the nines complement. We found out that in case we should complement the output the rippling of the carry in case of tens-complement makes it much slower than the nines complement. So, we tried another architecture in which we added another BCD-subtractor block for which we interchanged the 2 operands so that in case we need to complement the output all we have to do is -with the aid of an extra multiplexer- we select either the first or second BCD-subtractor so we won't wait for the carry rippling. This implementation enhanced the speed but on the other hand the area is also increased. Regarding both the area and speed, we found out that the nines complement is more suitable for our design for both area and speed

The internal design of the BCD-adder is the carry-ripple adder which is known by its small area, we introduced another implementation for the BCD-adder which is the carry look-ahead adder and we used the nine's complement for subtraction. We found out that the speed is enhanced by 42% and the area is increased but the design is still fitting in the same FPGA chip.

We compared the overall performance of the decimal adder from the point of view of area and speed for the same FPGA families. We synthesized the design for 2 families of Xilinx, Spartan II and Vertix II. And we got the previously mentioned results.

A behavioral test bench has been implemented to test the design against test vectors supplied by the IBM Corporation. Complete test and verification is performed on all the design versions fulfilling 3063 test vectors and supporting 7 rounding modes (5 stated by the standard and 2 proposed by IBM) with exception handling for overflow, inexact and invalid operations.

After testing the different design and passing all the test vectors, we concluded that the carry look ahead adder with the nine's-complements for subtraction gives better results on FPGA as regards the speed and fitting the same FPGA chip.

6.2 Future work

Based on the work presented in this thesis and the results obtained, we recommend the following items as the future work

- The current design may be easily extended to include the 128 bits wide operands as the second decimal format in the IEEE 754-2008 standard.
- Using Parallel architecture technique instead of the single path one, this will probably increase the speed.
- The main block that introduces the large delay is the BCD adder, trying other designs for it may speed up the design.
- Design and implementation of a decimal ALU.
- Multiplier.

References

[1] General Decimal Arithmetic website, http://speleotrove.com/decimal/

[2] A 64-Bit Decimal Floating-Point Adder John D. Thompson, Nandini Karra, and Michael J. Schulte, Member, IEEE.

[3] IEEE, IEEE 754-2008 Standard for Floating-Point Arithmetic, 2008.

[4] http://www.stanford.edu/class/ee486/doc/hap1

[5] Decimal Arithmetic FAQ Part 1 – General Questions http://speleotrove.com/decimal/decifaq1.html#inexact

[6] Ovidiu Ghita, Digital Electronics, 2003, Page 83

[7] Hossam A.H.Fahmy, Shlomo Waser, Michael J. Flym "Computer Arithmatic" to be published: http://arith.stanford.edu/hfolmy/webpage/arith_class/arith_class/arith.pdf

[8] http://www.haifa.ibm.com/projects/verification/fpgen/ieeets.html

[9] J. Thompson, M.J. Schulte, and N. Karra, “A 64-Bit Decimal Floating-Point Adder,” Proc. IEEE CS Ann. Symp. VLSI (ISVLSI ’04), pp. 297-298, Feb. 2004.

[10]http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r1m0/index.jsp?topic=/com.ibm.etools.mft.doc/ak05380_.htm

[11] Liang-Kai Wang, Charles Tsen, Michael J. Schulte, and Divya Jhalani
Benchmarks and Performance Analysis of Decimal Floating-Point Applications, IEEE,2007

[12] Optimized Decimal 64/128 Floating-Point Fast Adders Conforming to IEEE 754-2008, IEEE, 2009.

[13] E. M. Schwarz, J. S. Kapernick and M. F. Cowlishaw
Decimal floatingpoint support on the IBM System z10 processor, IBM, 2009.

[14] Mark A.Erle, Michael J. Schulte and John M. Linebarger

Potential speedup using Decimal Floating-Point hardware

[15] Liang-Kai Wang, Michael J. Schulte, John D. Thompson, and Nandini Jairam

Hardware Designs for Decimal Floating-Point Addition and Related Operations