INCLUDING a DECIMAL FLOATING POINT UNIT IN AN OPEN-SOURCE PROCESSOR AND PATCHING ITS COMPILER

By

Mohamed Hosny Amin Hassan

A Thesis Submitted to
Faculty of Engineering at Cairo University

In Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
In
ELECTRONICS AND ELECTRICAL COMMUNICATIONS ENGINEERING

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2012

INCLUDING a DECIMAL FLOATING POINT UNIT IN AN OPEN-SOURCE PROCESSOR AND PATCHING ITS COMPILER

By

Mohamed Hosny Amin Hassan

A Thesis Submitted to
Faculty of Engineering at Cairo University

In Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
In
ELECTRONICS AND ELECTRICAL COMMUNICATIONS ENGINEERING
FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT

Under the Supervision of
Associate Prof. Dr:
Hossam A. H. Fahmy
Principal Adviser

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2012

ABSTRACT

Although the binary representation is convenient to computer arithmetic; it is not natural to humans. So, decimal arithmetic has proved its necessity in some applications such as business. Accuracy is the main reason to include the decimal floating point specifications in IEEE 745- 2008. This can be performed either in software or hardware. However, hardware implementations speed up the operation with more energy savings. So far, only two processor architectures include decimal floating point units (z series and power series from IBM). This research provides the first free and open-source alternative to the above two architectures with a processor containing decimal floating point as well as the corresponding tool chain.

Our open-source processor uses a decimal Fused Multiply-Add unit (FMA) designed by our team as the core of the decimal unit. This processor is based on the UltraSparc T2 architecture from Oracle/Sun. It provides the basic decimal operations (Addition, Subtraction, Multiplication, Fused Multiply-Add and Fused Multiply-Subtract). To implement these changes, we add a new unit called Decimal Floating Point Unit (DFPU) then adapt the Instruction Fetch Unit (IFU), the Pick Unit (PKU), the Decoding Unit (DEC), the Floating Point Control Unit (FPC), and the Floating Point and Graphics Unit (FGU).

The second part of the work is to provide the necessary SW tools to generate programs for the new architecture. The GCC Compiler is patched to include several decimal double-precision floating point instructions. Finally, the op-codes of these instructions are added to the standard SPARC Instruction Set Architecture (SPARC ISA v9).

Contents

LIST OF TABLES

LIST OF FIGURES

Chapter 1 DECIMAL FLOATING POINT ARITHMETIC

The invention of numbers is one of the great and remarkable achievements of the human race. The numeration system used by humans has always been subjected to developments in order to satisfy the needs of a certain society at a certain point in time. The variation of these needs from a culture to another along with the evolution of numbering demands led to many different numeration systems across the ages. The traces of these systems were found and tracked by both linguists and archaeologists [1].

However, the fundamental step in developing all number systems was to develop the number sense itself which is the fact that the number is an abstract idea independent of the counted object. The sense of numbers evolved into three main stages. The first was to assign different sets of numbers to different types of objects. The second stage was matching the counted items against other more available and/or accessible ones. For example, counted items of any type were matched against a group of pebbles, grain of corns, or simply fingers. There were many bases for various numeration systems; however, the most common number systems at this stage were based on ten which is logically justified by the ease of counting on fingers. This fact is also, most probably, the main reason for the stability of our nowadays decimal system. Finally, once the sense of numbers is completely developed, distinct names should be assigned to numbers [1].

The need to record the results of counting led to inventing different ways to express numbers in a symbolic written format. This step in the numerals evolution led to two distinct systems, additive and positional. The additive system assigns distinct symbols to certain numbers. A combination of these symbols with the possibility of repeating any symbol as much as necessary can represent any number. This system was used by old Romans and Egyptians. It is easy for counting and simple for calculations, however, it is very complex with advanced arithmetic operations. On the other hand, in the positional system, the symbols representing numbers are positioned in a string with each position indicating a certain weight for the digit inside it. The Chinese and Babylonians used positional number systems. However, a main drawback with these systems was that, there was no symbol for ’zero’ to indicate an empty position. This led to both complexity and ambiguity in their numbering system [2].

The decimal numbering system was completely represented by Al-Khwarizmi in his book “The Keys of Knowledge” [3]. In the ninth century, while he was working as a scholar in the House of Wisdom in Baghdad, he developed the science of Algebra based on decimal numeration. The most remarkable achievement was introducing the digit ’zero’. In his book, he indicates that he learned this numbering system from Indians. This system, known as the Hindu-Arabic number system, spread gradually in Europe until it almost completely replaced the previously widespread Roman system at the 17th century [2].

The rest of this chapter is organized as follows: section 1.1 gives an overview about the history of the decimal numeration system in computers. Next, section 1.2 explains the increasing importance of decimal floating point arithmetic. The decimal floating point standard format with its arithmetic operations is discussed in section1.3. Section 1.4 surveys the recent published hardware implementations for different decimal floating point operations. Finally, a brief review for processors that support decimal is presented in section 1.5.

1.1 Decimal Arithmetic in Computers

Since the decimal number system was completely the dominant used numbering system at the 17th century, the first trials for mechanical computers adopted this system for calculations. A well-known example for these mechanical computers is the analytical engine by Charles Babbage [4]. However, the decimal numeration system was questionable again when the computer industry entered the electronic era.

The early electronic computers that depended on vacuum tube technology such as the ENIAC maintained the decimal system for both addressing and numbers. The main representation used was BCD (Binary Coded Decimal) [5]. The superiority of binary system over decimal was first discussed by Burks, Goldstine and von Neumann [6]. Despite the longstanding tradition of building digital machines using decimal numbering system, they argued that a pure binary system for both addressing and data processing would be more suitable for machines based on the two-state digital electronic devices such as vacuum tubes. They stated that binary system will be simpler, more reliable and more efficient than decimal. According to their reports, the simplicity stems from the fact that the fundamental unit of memory is naturally adapted to the binary which leads to more efficient representation and hence more precision. Also, they pointed out to the prevalence of binary system in elementary arithmetic and, of course, logical

operations which can be performed much faster than in decimal case. Due to its simplicity, it implies greater reliability due to the reduced number of components. Meanwhile, they underestimated the problem of conversion between binary and decimal, that is more familiar to humans. They argued that this conversion problem can be solved by the computer itself without considerable delay.

On the other hand, other researchers [7] outlined that, the format conversions between decimal and binary can contribute significantly to the delay in many applications that perform few arithmetic operations on huge data workloads. They concluded that the best solution for such case is to build separate arithmetic units. One of them is binary for addressing and the other is decimal for data processing. This debate ended up with two separate lines of computers around the 6th decade of the 20th century, one of them is dedicated to scientific and engineering applications which do complex calculations on small amount of input data and this line uses a fully binary ALU. While the other line is dedicated to the commercial applications which operate on huge data amounts with simple operations so it uses decimal ALU for data processing and binary ALU for addressing [8].

Two main factors led to merging these two lines in a single product between 1960 and 1970. First, the evolution of the solid-state semiconductor technology which contributed to the large scale production of computers with reduced area and cost. Second, the fact that customers are used to run commercial applications on scientific computers as well as business-oriented computers were used for some research purposes. These two reasons provided both the ability and the desire to merge both binary and decimal arithmetic units in one ALU [9].

In the 1970s, huge research efforts were exerted to speed up arithmetic operations in binary with limited equivalent efforts for decimal [10, 11, 12, 13, 14]. This led to more popularity for binary systems. Therefore, the early personal computers integrated only binary ALUs with limited decimal operations on the software layer performed on a binary hardware. A remarkable example is the Intel x86 microprocessor which provides some instructions for BCD such as DAA (Decimal Adjustment after Addition) and DAS (Decimal Adjustment after Subtraction) which adjust the binary result of addition or subtraction as if the operation was conducted on decimal hardware [15]. On the other side, binary floating point, which was first proposed in 1914, was supported in the x86 by specialized chips calledfloating-point accelerators. This was mainly because of its

complexity and hence the difficulty to integrate it within the microprocessor chip [16].

The floating point units gained increased popularity, specifically for scientific applications. This led to many designs with different formats and rounding behaviors for arithmetic operations. Therefore it was necessary to standardize a floating-point system so that the same operation can provide the same result on different designs. Thus, the IEEE 754-1985 standard was issued as a binary floating-point standard.

In 1987, another standard for radix independent floating-point arithmetic (IEEE 854-1987) was released [17]. However, it found no echo in the market. This was, from one hand, due to a shortage in the standard itself which lacked some features such as an efficient binary encoding for numbers of higher radices; especially decimal. On the other hand, there was no sufficient demand in the market for decimal floating point processing, particularly which, a decimal floating point unit was still relatively complex enough not to be integrated into a general-purpose microprocessor with the fabrication technologies available at that time [9].

At the beginning of 2000s, there was growing importance of decimal arithmetic in commercial and financial applications, along with technological improvements that allow integration of more complex units. This resulted in a demand for standard specifications for decimal floating-point arithmetic. Thus, the new revision of the IEEE standard for floating-point arithmetic (IEEE 754-2008) includes specifications for decimal floating point arithmetic [18].

In the next section, the importance of decimal floating point that led to its adoption in the new standard will be explored.

1.2 Importance of Decimal Floating Point Arithmetic

The controversy over binary and decimal numeration systems that was opened in the 1970s led initially to merging both systems in the same ALU and ended up with the complete adoption of binary system and depending only on software to perform decimal calculations. Yet, the same debate was reopened again in the 2000s.

Banking, billing, and other financial applications use decimal extensively. Such applications should produce final results that are expected by humans and required by law. Since conversion of some decimal fractions to their binary

equivalents may result in endless fractions, this implies a loss of accuracy due to limited storage in case of using pure binary arithmetic. For example, simple decimal fractions such as 0.1 that might represent a tax amount or a sales discount yield an infinitely recurring number if converted to a binary representation ($0.0001100110011\cdots$). This conversion error accumulates and may lead to significant losses in the business market. In a large telephone billing application such an error may end up to $5 million per year [19].

In addition to the accuracy problem, the user of a human oriented application expects trailing zeros to be preserved in different operations. Without these trailing zeros the result of operation appears to be vague. For example, if the specification of a resistor states that it should be of 1.200 kW, this implies that this measurement is to the nearest 1W. However, if this specification is altered to 1.2 kW, then the precision of the measurement is understood to be to the nearest 100 W. This example shows that it is not only the numerical value of a number that is significant; however, the full precision of a number should be also taken into consideration. The binary floating point arithmetic does not follow this rule because of its normalized nature.

Such applications may rely on either a low level decimal software library or use dedicated hardware circuits to perform the basic decimal arithmetic operations. However, as stated in [8], some applications use the decimal processing in 50% to 90% of their work and that software libraries are much slower than hardware designs. So, instead of pure software layering on binary floating-point hardware, one solution is to use decimal fixed point (DXP) hardware to perform decimal arithmetic. Yet, there are still several reasons to use direct decimal floating-point (DFP) hardware implementations. First, financial applications often need to deal with both very large numbers and very small numbers. Therefore, it is efficient to store these numbers in floating-point formats. Second, DFP arithmetic provides a straightforward mechanism for performing decimal rounding, which produces the same results as when rounding is done using manual calculations. This feature is often needed to satisfy the rounding requirements of financial applications, such as legal requirements for tax calculations. Third, DFP arithmetic also supports representations of special values, such as notanumber (NaN) and infinity (∞), and status flags, such as inexact result and divide-by-zero. These special values and status flags simplify exception handling and facilitate error monitoring.

A benchmarking study [20] estimates that many financial applications spend over 75% of their execution time in Decimal Floating Point (DFP) functions. For this class of applications, the speedup for a complete application (including non-decimal parts) resulting from the use of a fast hardware implementation versus a pure software implementation ranges from a factor of 5.3 to a factor of 31.2 depending on the specific application running.

Besides the accuracy and the speed up factors, savings in energy are very important. A research paper [21] estimates that energy savings for the whole application due to the use of a dedicated hardware instead of a software layer are of the same order of magnitude as the time savings. It also indicates that the process normalized Energy Delay Product (EDP) metric, suggested in [21], clearly shows that a hardware implementation for DFP units gives from two to three orders of magnitude improvement in EDP as a conservative estimate if compared with software implementations.

The decimal arithmetic seems to take the same road map of binary. After the domination of binary ALUs in processors, a common trend now is to include either separated Decimal (including DFP) ALUs besides their binary equivalents [22, 23]or to use combined binary and decimal ALUs [24]. This leads to a question whether the decimal arithmetic will dominate if the performance gap between the decimal and binary implementations shrinks enough.

1.3 IEEE Decimal Floating-Point Standard

As previously indicated, there was an increasing need to DFP arithmetic. Hence, there were many efforts to find out the most appropriate DFP formats, operations and rounding modes that completely define the DFP arithmetic. These efforts ended up with the IEEE 754-2008 floating-point arithmetic standard. This section gives a brief overview to this standard [18].

1.3.1 Decimal Formats

The IEEE 754-2008 defines DFP number as: $(-1)^s \times (10)^q \times c$, where: S is the sign bit, q is the exponent, $c = (d_{p-1}d_{p-2} \cdots d_0)$ is the significand, where $di \in 0,1,2,3,4,5,6,7,8,9$, and p is the precision.

Figure1.1 shows the basic decimal interchange format specified in the IEEE 754-2008 standard. S is the sign bit which indicates either the DFP number is positive (S = 0) or negative (S = 1) and G is a combination field that contains the exponent, the most significant digit of the significand, and the encoding

classification. The rest of the significand is stored in the Trailing Significand Field, T, using either the Densely Packed Decimal (DPD) encoding or the Binary Integer Decimal (BID) encoding, where the total number of significand digits corresponds to the precision, p. The DPD encoding represents every three consecutive decimal digits in the decimal significand using 10 bits, and the BID encoding represents the entire decimal significand in binary.

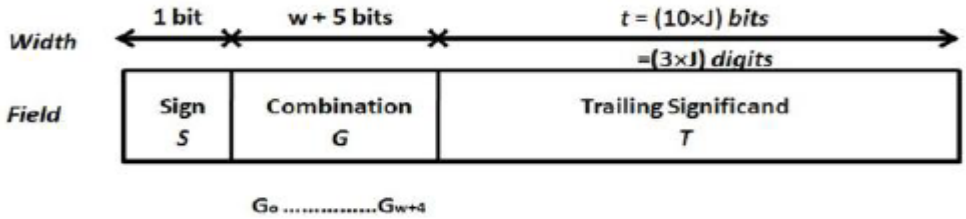


Figure 1.1: DFP interchange format

Format name	decimal64	decimal128
Total storage width	64	128
Combination field width (w + 5)	13	17
Trailing significand field (t)	50	110
Total significand digits (p)	16	34
Exponent bias	398	6176

Table 1.1: Parameters for different decimal interchange formats

Before being encoded in the combination field, the exponent is first encoded as binary excess code and its bias value depends on the precision used. There are also minimum and maximum representable exponents for each precision. The different parameters for different precision values are presented in Table 1.1.

1.3.2 Operations

The standard specifies more than 90 obligatory operations classified into two main groups according to the kinds of results and exceptions they produce:

- **Computational Operations:**

  These operations operate on either floating-point or integer operands and produce floating-point results and/or signal floating-point exceptions. This general category can be also decomposed into three classes of operations.

  General-computational operations: produce floating-point or integer results, round all results and might signal floating-point exceptions. For

example, all arithmetic operations such as addition, subtraction, multiplication and so on.

Quiet-computational operations: produce floating-point results and do not signal floating-point exceptions. It includes operations such as negate, absolute, copy and others.

Signaling-computational operations: produce no floating-point results and might signal floating point exceptions; comparisons are signaling computational operations.

- **Non-Computational Operations:**

  These operations do not produce floating-point results and do not signal floating-point exceptions. It includes, for example, operations that identify whether a DFP number is negative/positive, finite/infinite, Zero/Non-zero and so on.

Operations can be also classified in a different way according to the relationship between the result format and the operand formats:

- **Homogeneous operations**: in which the floating point operands and floating point results have the same format.

- **FormatOf operations**: which indicates that the format of the result, independent of the formats of the operands.

Each of the computational operations that return a numeric result specified by this standard shall be performed as if it first produced an intermediate result correct to an infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format . In some cases, exceptions are raised to indicate that the result is not the same as expected or invalid operations. On the other hand, as indicated before, a floating-point number might have multiple representations in a decimal format. All these operations, if producing DFP numbers, do not only specify the correct numerical value but they also determine the correct member of the cohort.

It should be highlighted that, besides the required operations for a standard compliant implementation, there are other recommended operations for each supported format. These operations mainly include the elementary functions such as sinusoidal and exponential functions and so on.

1.3.3 Rounding

There are five rounding modes defined in the standard, Round ties to even, Round ties to away, Round toward zero, Round toward positive infinity, and Round toward negative infinity. Also, there are two well-known rounding modes supported in the Java BigDecimal class [25]. Table 1.2 summarizes the different rounding modes with their required action.

Rounding Mode	Rounding Behavior
Round Ties To Away RA	Round to nearest number and round ties to nearest away from zero, the result is the one with larger magnitude.
Round Ties to Even RNE	Round to nearest number and round ties to even, the result is the one with the even least significand digit.
Round Toward Zero RZ	Round always towards zero , the result is the closest DFP number with smaller magnitude.
Round Toward Positive RPI	Round always towards positive infinity, the result is the closest DFP number greater than the exact result.
Round Toward Negative RNI	Round always towards negative infinity, the result is the closest DFP number smaller than the exact result.
Round Ties to Zero RZ	Round to the nearest number and round ties to zero.
Round To Away RA	Round always to nearest away from zero, the result is the one with larger magnitude.

Table 1.2: Parameters for different decimal interchange formats

1.3.4 Special numbers and Exceptions

1.3.4.1 Special numbers

Operations on DFP numbers may result in either exact or rounded results. However, the standard also specifies two special DFP numbers, infinity and NaN.

1.3.4.2 Normal and Subnormal numbers

A normal number can be defined as a non-zero number in a floating point representation which is within the balanced range supported by a given floating-point format. The magnitude of the smallest normal number in a format is given by $b^{e_{min}}$, where b is the base (radix) of the format and e_{min} is the minimum representable exponent. On the other hand, subnormal numbers fill the underflow gap around zero in floating point arithmetic. Such that any non-zero number which is smaller than the smallest normal number is a subnormal number.

1.3.4.3 Infinities

Infinity represents numbers of arbitrarily large magnitudes, larger than the maximum represented number by the used precision. That is:

$$-\infty < \{each\ representable\ finite\ number\} < +\infty.$$

In Table 1.3, lists of some arithmetic operations that involve infinities as either operands or results are presented. In this table, the operand x represents any finite normal number.

Operation	Exception	Operation	Exception
$\infty + x = \infty$	None	$\infty / x = \infty$	None
$\infty + \infty = \infty$	None	$x / \infty = 0$	None
$\infty - x = \infty$	None	$\infty / \infty = NaN$	Invalid
$\infty - \infty = NaN$	Invalid	$\sqrt{\infty} = \infty$	None
$\infty \times x = \infty$	None	$\sqrt{-\infty} = NaN$	Invalid
$\infty \times \infty = \infty$	None	$\pm x / 0 = \pm\infty$	Division by Zero
$\infty \times 0 = NaN$	Invalid		

Table 1.3: Examples of some DFP operations that involve infinities

1.3.4.4 NaNs (Not a Number)

Two different kinds of NaN, signaling and quiet, are supported in the standard. Signaling NaNs (sNaNs) represent values for uninitialized variables or missing data samples. Quiet NaNs (qNaNs) result from any invalid operations or operations that involve qNaNs as operands. When encoded, all NaNs have a sign bit and a pattern of bits necessary to identify the encoding as a NaN and which determines its kind (sNaN or qNaN). The remaining bits, which are in the trailing significand field, encode the payload, which might contain diagnostic information that either indicates the reason of the NaN or how to handle it. However, the standard specifies a preferred (canonical) representation of the payload of a NaN.

1.3.5 Exceptions

There are five different exceptions which occur when the result of an operation is not the expected floating-point number. The default nonstop exception handling uses a status flag to signal each exception and continues execution, delivering a default result. The IEEE 754-2008 standard defines these five types of exceptions as shown in Table 1.4.

Exceptions	Description	Output
Invalid Operation (Description shows only common examples)	-Computations with sNaN operands -Multiplication of $0 \times \infty$ -Effective subtraction of infinities -Square-root of negative operands -Division of $0/0$ or ∞/∞ -Quantize in an insufficient format -Remainder of $x/0$ or ∞/x (x: finite non zero number)	Quite NaN
Division by Zero	The divisor of a divide operation is zero and the dividend is a finite non-zero number.	Correctly signed ∞
Overflow	The result of an operation exceeds in magnitude the largest finite number representable.	The largest finite number representable or a signed ∞ according to the rounding direction.
Underflow	The result of a DFP operation in magnitude is below $10^{e_{min}}$ and not zero	zero, a subnormal number or $\pm 10^{e_{min}}$ according to rounding mode.
Inexact	The final rounded result is not numerically the same as the exact result (assuming infinite precision)	The rounded or the overflowed result.

Table 1.4: Exceptions' types

1.4 Standard Compliant Implementations of DFP Operations

As mentioned earlier, support of DFP arithmetic can either be through software libraries such as the Java BigDecimal library [25], IBM's decNumber library [26], and Intel's Decimal Floating-Point Math library [27], or through hardware modules. These software libraries are re-mentioned in Chapter 2 with more details. Many hardware implementations have been introduced in the last decade to perform different operations defined in the standard. This includes adders, multipliers, dividers and some elementary functions and others.

Many DFP adder designs have been proposed for the last few years. Thompson et al. [28] proposed the first published DFP adder compliant with the standard. A faster implementation with architectural improvements is proposed in [29]. An extension and enhancement of this work is proposed again in [30]. Further improvements are proposed by Vazquez and Antelo in [31]. Fahmy et al [21] proposed two other different adder implementations, one for high speed and

the other for low area. Yehia and Fahmy [32] proposed the first published redundant DFP adder to allow for a carry-free addition.

There are also many designs for integer decimal multiplication [33, 34]. Erle et al. [35] published the first serial DFP multiplier compliant with the IEEE 754-2008 standard. While Hickmann et al. [36] published the first parallel DFP multiplier. Raafat et al. [37] presented two proposals to decrease the latency of parallel decimal multiplication. Also Vazquez, in [38], proposed two high performance schemes for DFP multipliers, one optimized for area and the other optimized for delay.

An incomplete decimal FMA floating-point unit is developed and combined with a known binary FMA algorithm in [24]. This incomplete unit supports the decimal64 and binary64 formats and claims conformance to the standard's specification for rounding and exceptions, but not underflow and subnormal numbers. However, the first known conforming hardware implementation for decimal FMA is presented in [39].

Early proposals for DFP dividers are introduced in [40, 41]. However, the first DFP standard compliant designs can be found in IBM POWER6 [23] and Z10 [23] microprocessors. Also, another compliant DFP divider is proposed by Vazquez in [42].

Since the IEEE 754-2008 standard has been approved, many designs and implementations for elementary functions in decimal are introduced. For example, different proposals for modifying the CORDIC method to work on decimal without conversion to binary are represented in [43]. The CORDIC algorithm is also used to implement different transcendental functions [44]. A comprehensive library of transcendental functions for the new IEEE decimal floating-point formats is presented in [45]. There is also different proposal for a DFP logarithmic function in [46] and [47].

1.5 IEEE 754-2008 DFP Support in Microprocessors

As discussed in section 1.2, decimal arithmetic was supported by many processors. Moreover, the first generations of processors, such as ENIAC, support only decimal. However, the zSeries DFP facility was introduced in the IBM System z9 platform. The z9 processor implements the facility with a mixture of low-level software - using vertical microcode, called millicode and hardware assists using the fixed point decimal hardware [48]. Because the DFP was not fully defined

when the z9 processor was developed, there was only basic hardware support for decimal. Yet, more than 50 DFP instructions are supported in millicode. Millicode enables implementing complex instructions where hardware support is not possible, and to add functions after hardware is finalized. This leaves System z9 as the first machine to support the decimal floating point (DFP) instructions in the IEEE Standard P754.

The POWER6 is the first processor that implements standard compliant decimal floating-point architecture in hardware. It supports both the 64-bit and the 128-bit formats. As described in [49, 50], 54 new instructions and a decimal floating-point unit (DFU) are added to perform basic DFP operations, quantum adjustments, conversions, and formatting. The POWER6 implementation uses variable-latency operations to optimize the performance of common cases in DFP addition and multiplication.

The IBM System z10 microprocessor is a CISC (complex instruction set computer) microprocessor. It implements a hardwired decimal floating point arithmetic unit (DFU) which is similar to the DFU of the POWER6 with some differences [22, 23]. The differences are mainly about the DXP unit architecture and its interface with DFP unit. However, many of the DFP operations are implemented in hardware in both POWER6 and System z10, but there are other operations that are not. For example, the FMA operation which is required for a standard compliant DFP unit is not implemented in hardware. More details about these processor architectures and their decimal units are introduced in the next chapter.

In this chapter, an introduction to the decimal floating-point arithmetic is presented. The second chapter surveys in some details the software libraries and the processors that support decimal floating point operations. Chapter 3 discusses the OpenSPARC T2 core architecture. It goes in deep with each unit describes its block diagram and operation. In chapter 4, we introduce the extension of the SPARC instruction set architecture to include decimal floating point instructions; we also introduce in this chapter our decimal floating point unit and its inclusion in the OpenSPARC T2 core. Chapter 5 investigates the work done on the software level by this thesis. It presents the engender of a software tool chain to generate SPARC assembly files as well as binary files from C decimal floating point programs. Finally, Chapter 6 explains the verification environment used for testing the modified OpenSPARC T2 core. It concludes with results and proposed future work.

Chapter 2 DECIMAL PROCESSORS AND LIBRARIES

2.1 Introduction

In this chapter, we survey both the processors that support DFP hardware instructions and the software libraries that emulate the DFP through software routines executed by binary floating point processors. Section 2.2 starts with the IBM processors that support DFP through different algorithms. These processors are the IBM Z9, POWER6 and IBM Z10. After that, section 2.3 refers to the IBM decNumber and the Intel Math libraries. Finally, section 2.4 compares between these different hardware/software implementations.

2.2 Hardware Decimal Floating Point

2.2.1 Decimal Floating point support in Z9

The zSeries DFP facility was introduced in the IBM System z9 platform. System z9 is the first IBM machine to support the decimal floating point (DFP) instructions. The z9 processor implements the facility with a mixture of low-level software, using vertical microcode, called millicode [51], and hardware assists using the fixed point decimal hardware [48].More than 50 DFP instructions were added. They are implemented mainly in millicodes, while performing only the most basic tasks in hardware. The DFP facility shares the floating-point registers (FPRs) with the binary and the hexadecimal floating-point operands.

Millicode	Operation	Description
Interface between the MGPRs and the FPRs		
EXFDI	Extract FPR indirect	load an MGR from an FPR
SFDI	Set FPR indirect	load an FPR from an MGR
Decoding and Encoding (DPD ←→BCD)		
	Extract exponent	Decode the exponent from the DPD→ biased binary
EBCDR	Extract Coefficient	Decode the exponent from the DPD→ BCD
IXPDR	Insert exponent	Encode the exponent from the biased binary → DPD
CBCDR	Compress Coefficient	Encode the significand from BCD → DPD
Basic Operations		
APRR	Add decimal register	APRR and SPRR are single-cycle instructions, while MPRR and DPRR are multiple cycles. The four millicodes use the decimal fixed point (FXU) that already exists in hardware.
SPRR	Subtract decimal register	
MPRR	Multiply decimal register	
DPRR	Divide decimal register	

Table 2.1: DFP Millicodes

2.2.1.1 The hardware supprot

Millicode is the lowest-level firmware in the IBM z-series and is used to implement instructions that can't be implemented using hardware. It is written in a subset of the System z assembly language with the millcode-instructions (milliops). The millicodes use a special register file called Millicode General Purpose Registers (MGPRs) and the following millicodes are added to support DFP instructions, Table 2.1.

2.2.1.2 DFP Instruction Execution

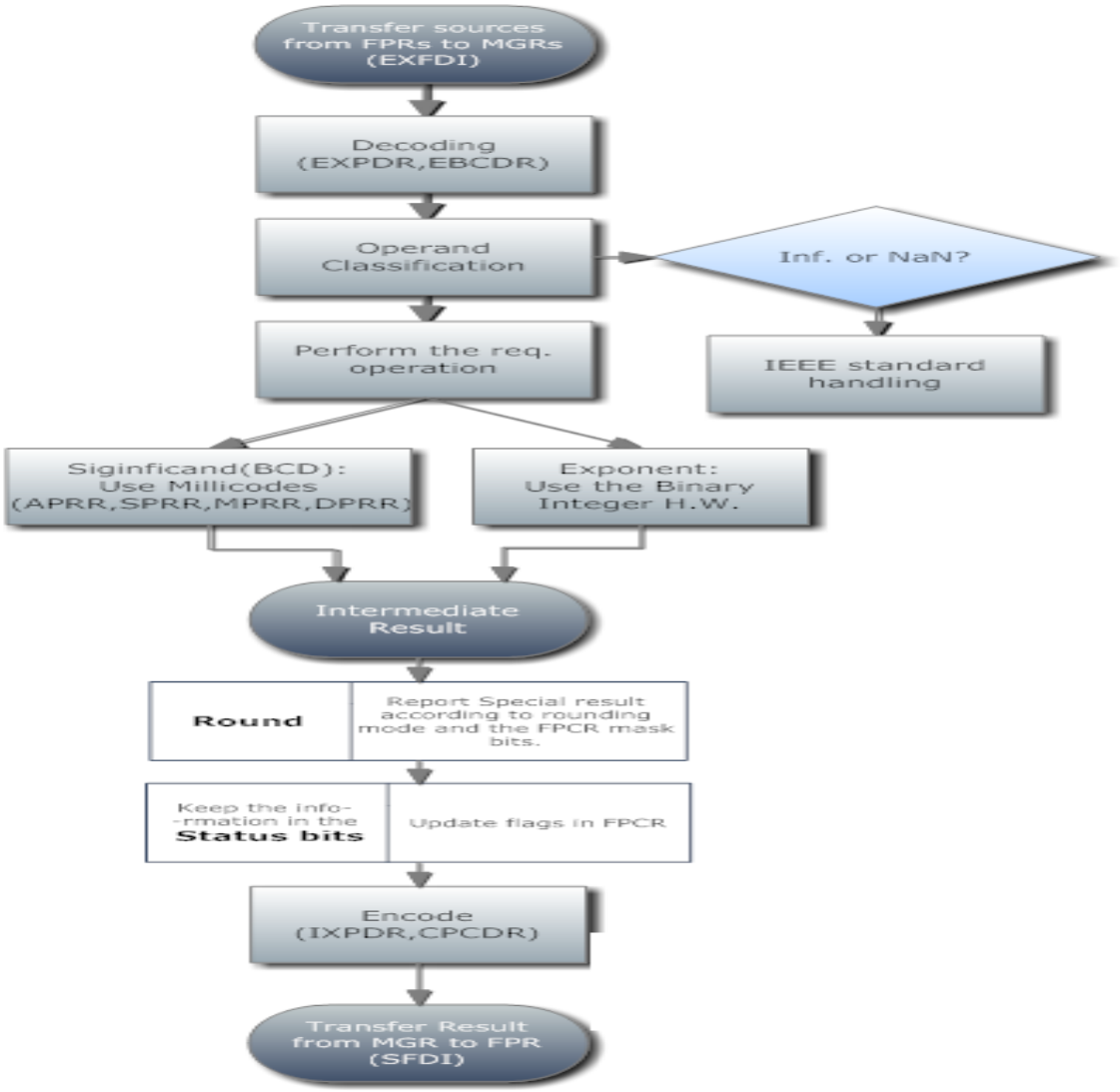


Figure 2.1: DFP Instructions Flow Chart

2.2.1.3 Performance

To increase the performance different algorithms were used according to different inputs' cases sources. For example, in the addition/subtraction case: as the equal exponent operand case is more likely common, it is performed in a separate fast path which leads to a different latency hardware that depends in the class of the sources.

2.2.2 Decimal Floating point unit in POWER6 [52]

The POWER6 is the first processor that implements the decimal floating-point architecture completely in hardware. It supports both the 64-bit and the 128-bit formats.

2.2.2.1 The Floating Point register file

The register file (FPRs) is shared between the FPU and the DFU:

- Because of the fundamental differences in the requirements met between these two radices, a program is unlikely to require both binary and decimal floating-point computations simultaneously.
- This will optimize the area.
- This also will optimize the instruction set as by sharing the FPRs, there is no need to an additional load and store instructions for decimal. They are shared with binary.

FPR contains 32 double-word registers (i.e. 16 quad-word registers).

2.2.2.2 The floating-point status and control register (FPSCR)

The floating-point status and control register (FPSCR) is used by both binary and decimal floating-point units. Only the rounding mode is separated for decimal floating point. The decimal rounding mode field is 3 bits and allows eight different rounding modes.

Rounding modes:

- Round to nearest even
- Truncate
- Round toward positive infinity.
- Round toward negative infinity.
- round to nearest ties away from zero,
- round to nearest ties toward zero,
- Round away from zero.

- Round to prepare for shorter precision.

The FPCR records the class of the result for arithmetic instructions. The decimal classes are (subnormal – normal – zero – infinity – quiet NaN – Signaling NaN).

2.2.2.3 Hardware Implementation

2.2.2.3.1 The adder

The main component of the POWER6 DFU is a wide 36-digit (or 144-bit) adder. The cycle time of POWER6 processor is approximately 13 FO4. As a result, the widest decimal adder which can complete its task in one cycle is a 4-digit width adder. Actually, the implemented 4D adder shown in Figure 2.2 is a group of four separate conditional 1D adders that choose between (sum, sum+1, sum+6, sum+7) depending on the input carry to each digit.

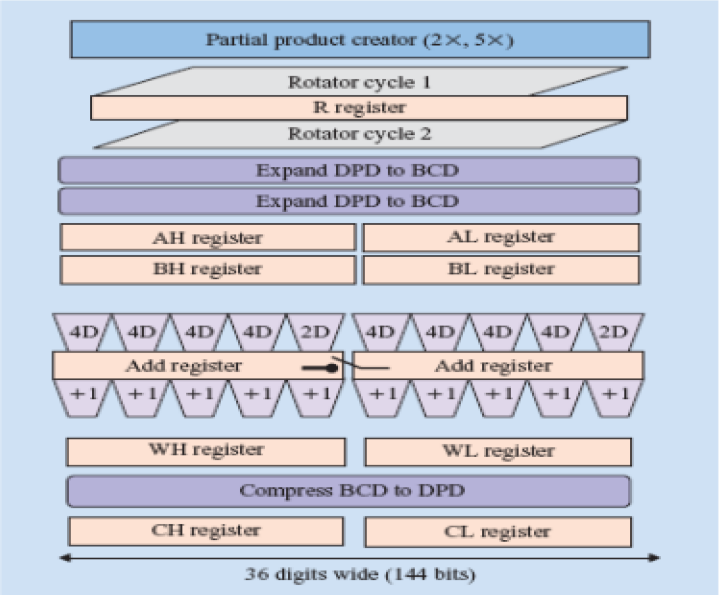


Figure 2.2: DFP unit in POWER6

The wide 36-digit adder is designed by replicating this 4D adder group without carry propagation between the groups, such that the final result will be the sum or sum+1. To calculate the sum+1, 4-digit incrementers are used in the next cycle. Consequently, the final adder result is available after two cycles, but we can start a new instruction execution after only one cycle as it is based on a pipelined architecture.

The adder supports both double precision and quad precision instructions. It can be used as a whole one 36-digits adder to perform quad precision addition or

as two 18-digits adders to perform double precision addition. This selection is done on the fly by a control signal.

2.2.2.3.2 The rotator

A rotator is used to align the operands and shift the result if required. It takes two cycles to rotate to any of 36-digits.

The addition operation has three paths for different three cases:

Exponents Equal

Expand DPD data to BCD.

Add the coefficients.

If there is a carryout from the adder, inc- rement the exponent, shift the coeff- icient right, and round.

Compress the result to DPD format.

Aligning to the smaller exponent

Expand to BCD and in parallel compare the exponents.

Swap the operands, creating two operands called big and small.

Shift the operand with the larger exponent left by the exponent difference.

Add aligned big to small.

Round, if necessary.

Compress to DPD format.

Shifting both operand

Expand to BCD and, in parallel, compare the exponents.

Swap the operands.

Shift the operand with the larger exponent left by the exponent difference (D).

Reshift the operand with the larger exponent left by the number of leading zeros in its coefficient (Z).

Compute D - Z and shift the operand with the smaller exponent right by the result.

Add the now-aligned coefficient.

Round.

Compress the result to DPD format.

These three cases are executed concurrently, and in the event of a conflict, the faster case is given precedence. The subtraction operation also performs A-B and B-A in parallel and finally chooses the right answer. An exception for this parallelism is the double-word subtraction operation as it can't do the $A - B$ and $B - A$ in parallel due to register restrictions, so it does it in series which for sure takes much delay.

2.2.2.3.3 The multiplier

Generating the partial products:

It is implemented in the (dumult) macro. A doubler and quintupler are hardware implemented and then used to create easy multiples (x1, x2, x5, and x10) of the multiplicand. The doubler and quintupler are very fast because each digit is independent of other digits and there is no carry propagation, then all possible multiples of the multiplicand can be formed by a simple addition or subtraction of two of the easy multiples. The adder is specially optimized to speed up multiplication. It can work in two modes:

- **For 16-digit multiplication:**

The half of adder is used to perform 18-digits addition to create a new partial product every cycle by summing or subtracting two easy multiples of the multiplicand. The other half of the adder is split into even and odd cycles, with even cycles used to create the sum of two paired partial products and odd cycles used to accumulate paired products with the running sum in the other half.

- **For 34-digit multiplication**:

The total 36-digit adder is used to create a new partial product every clock cycle and to accumulate it with the running sum in another cycle.

2.2.2.3.4 Division

The non-restoring division with prescaling algorithm is used to generate the quotient digits by the following steps each digit [53]:

- Quotient selection based on a partial remainder $q_{sel(i)}$ (**1 cycle**)

It is made from a redundant set of {-5 to 5} to reduce the number of divisor. The quotient digits are adjusted on the fly after they are selected and before they are put into the final result register. This adjustment is done in parallel

with the next partial remainder computation and so no additional delay is added to the critical path.

- Multiplication of the divisor D by the quotient digit. (**1 cycle**).

Multiplication of the selected quotient by the pre-scaled divisor is done by selecting the appropriate multiple of the divisor. Divisor multiples x1, x3, and x4 are pre-computed and stored in the partial product creator, and x2 and x5 are generated on the fly in the BCD doubler and BCD quintupler logic in the partial product creator block.

- Computation of the next partial remainder P_{i+1}, as shown by $P_{i+1} = P_i - q_{sel(i)}.D.$ (**2 cycles**).

2.2.3 Decimal Floating point support in Z10 [48]

2.2.3.1 Z10 overview

The IBM System z10 microprocessor is a CISC (complex instruction set computer) microprocessor operates at 4.4 GHz. It implements a hardwired decimal floating-point arithmetic unit (DFU) which is similar to the DFU of the POWER6 but has some differences [54].

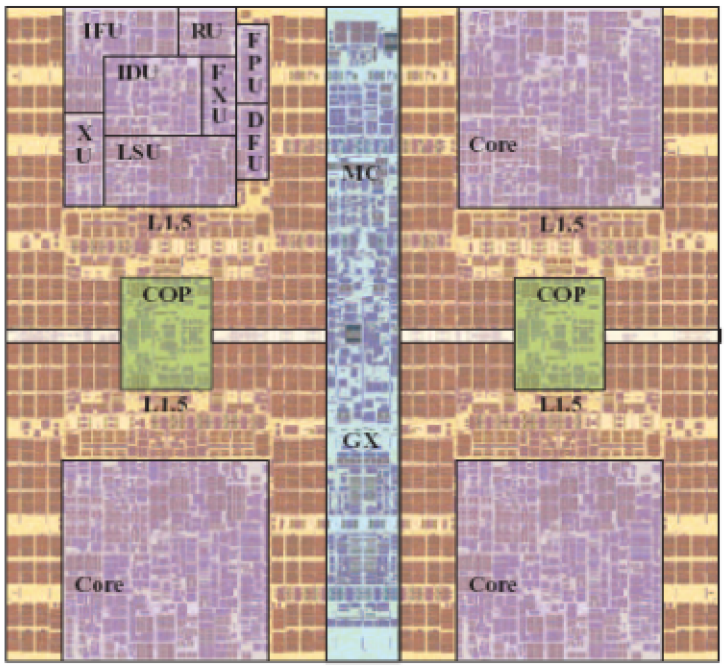


Figure 2.3: z10 Architecture

The block diagram in Figure 2.3 shows that the z10 processor offers a DFU that is separate from the binary and hexadecimal floating-point pipelines (FPU) and also separate from the dual fixed point unit (FXU) pipelines. However, all floating point operations share the same register file 16 X 64-bit floating-point register (FPR).

The z10 DFU supports DFP operations using the same main dataflow of the POWER6 processor in addition to supporting the traditional fixed-point decimal operations.

2.2.3.2 The z10 DFU and the POWER6 DFU

- Z10 has extra interfaces to the fixed-point unit (FXU) and data cache.
- Z10 has completely new set of controls to support the additional instructions in the IBM z/Architecture platform
- The z10 DFU has an additional 13 decimal fixed-point instructions and four hardware-assist instructions. Both the z10 DFU.

Both the z10 DFU and the POWER6 processor DFU have 54 DFP instructions. DFP operands have three formats: short (7-digits significand), long (16-digits significand), and extended (34-digits significand). Arithmetic operations are performed only on the long and extended formats. The operands are loaded from memory into the 16 * 64- bit floating-point register (FPR) files, which are also shared with the binary and hexadecimal floating-point operations [54].

2.2.3.3 *The DFU Hardware Implementation*

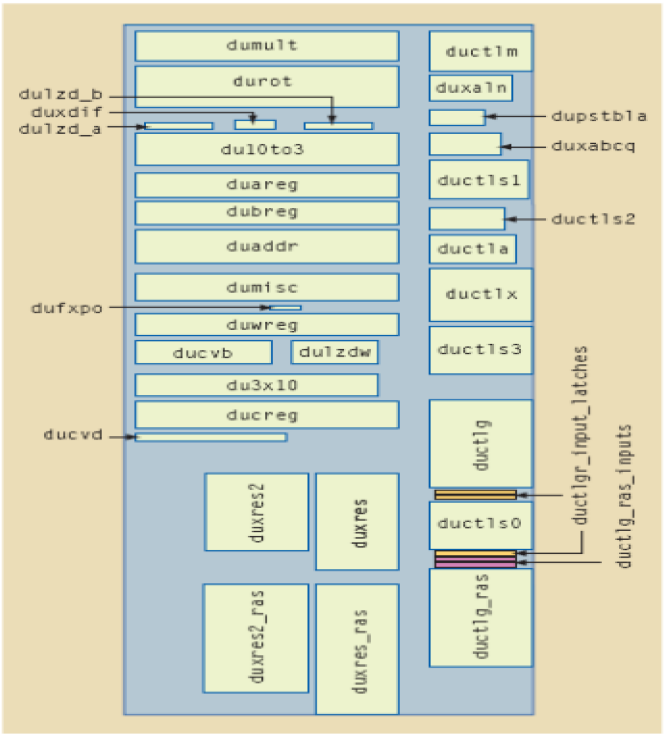


Figure 2.4: DFP modules in z10

2.2.3.4 *Decimal fixed point support*

Decimal fixed-point operations have been in the z/Architecture since its beginning in 1964 [54]. For fixed-point decimal operations, both source operands and the target are in memory in packed BCD format. So, using the result of a prior operation creates an interlock in memory, this is a point needs to be resolved. This problem does not exist in the DFU as the operands are in registers not in the memory which makes dependencies easier and faster to handle.

Decimal fixed-point operations have are faster than the decimal floating-point operations as they have fewer execution sequences: both have the same execution, except that no expansion, compression, and no rounding cycles are required in the fixed point operations.

2.2.3.5 Macros

Macro	Name	Discription
Dumult	Multiple creator macro	multiplicand (x2) and (x5).
Durot	Rotator	Shift the significand right or left.
dulzd_a, dulzd_b	Leading zero detector	For both operands
Duxdif	Exponent difference	
du10to3	Decoding	DPD → BCD
Duaddr	The adder	Can be two 18-digit adders or one 36-digit adder
du3x10	Encoding	BCD → DPD
Ducvb		Decimal → Binary
Ducvd		Binary → Decimal
Dulzdw	leading zero detector	for the result
Ductlm	Control	of the multiplication and division
Ductla	Control	of the addition
Ductlg	instruction decode	also contains global controls
ductls1,2,3	handling special results	also common rounding routine
ductls0	RAS	checking and reporting
Duxabcq	holds input exponents	
Duxaln	significand alignment	
Dupstbla	look-up-table	for the division

Table 2.2: z10 DFP macros explanation

2.3 Decimal Floating Point Libraries

2.3.1 IBM DecNumber [26]

The decNumber library implements the General Decimal Arithmetic Specification in ANSI C. The library supports integer, fixed-point, and floating-point decimal numbers including infinite, NaN (Not a Number), and subnormal values.

The library consists of several modules (corresponding to classes in an object-oriented implementation). Each module has a header file (for example, decNumber.h) which defines its data structure, and a source file of the same name (e.g., decNumber.c) which implements the operations on that data structure.

The core of the library is the decNumber module. Once a variable is defined as a decNumber, no further conversions are necessary to carry out arithmetic. Most functions in the decNumber module take as an argument a decContext structure, which provides the context for operations (precision,

rounding mode, etc.) and also controls the handling of exceptional conditions (corresponding to the flags and trap enablers in a hardware floating-point implementation).

The decNumber representation is machine-dependent (for example, it contains integers which may be big-endian or little-endian), and is optimized for speed rather than storage efficiency.

2.3.1.1 Storage Formats

Four machine-independent (but optionally endian-dependent) compact storage formats are provided for interchange. These are:

- decimal32
  This is a 32-bit decimal floating-point representation, which provides 7 decimal digits of precision in a compressed format.

- decimal64
  This is a 64-bit decimal floating-point representation, which provides 16 decimal digits of precision in a compressed format.

- decimal128
  This is a 128-bit decimal floating-point representation, which provides 34 decimal digits of precision in a compressed format.

- decPacked
  The decPacked format is the classic packed decimal format implemented by IBM S/360 and later machines, where each digit is encoded as a 4-bit binary sequence (BCD) and a number is ended by a 4-bit sign indicator. The decPacked module accepts variable lengths, allowing for very large numbers (up to a billion digits), and also allows the specification of a scale.

The module for each format provides conversions to and from the core decNumber format. The decimal32, decimal64, and decimal128 modules also provide conversions to and from character string format (using the functions in the decNumber module).

2.3.1.2 Standards compliance

It is intended that the decNumber implementation complies with:

- The floating-point decimal arithmetic defined in ANSI X3.274-1996.
- All requirements of IEEE 854-1987, as modified by the current IEEE 754r revision work, except that:
  1. The values returned after overflow and underflow do not change when an exception is trapped.
  2. The IEEE remainder operator (decNumberRemainderNear) is restricted to those values where the intermediate integer can be represented in the

current precision, because the conventional implementation of this operator would be very long-running for the range of numbers supported (up to ±101,000,000,000).

All other requirements of IEEE 854 (such as subnormal numbers and –0) are supported.

2.3.2 Intel Decimal Floating-Point Math Library [27]

The library implements the functions defined for decimal floating-point arithmetic operations in the IEEE Standard 754-2008 for Floating-Point Arithmetic. It supports primarily the binary encoding format (BID) for decimal floating-point values, but the decimal encoding format is supported too in the library, by means of conversion functions between the two encoding formats.

Release 1.0 Update 1 of the library implements all the operations mandated by the IEEE Standard 754-2008. Alternate exception handling is not supported currently in the library. Also it provides several useful functions that are not part of the IEEE 754-2008 standard like the rounding modes. For operations involving integer operands or results, the library supports signed and unsigned 8-, 16-, 32-, and 64-bit integers.

2.4 Performance Analysis and Comparisons

2.4.1 Hardware V.S. Software DFP Instructions

In [55], the effect of using DFP hardware on speedup is investigated. They wrote software routines for addition, subtraction, multiplication and division, then simulated it using the SimpleScalar simulator and finally got the cycle count. Concerning Hardware implementations, some existing designs were studies and new ones were suggested for the same four operations. The total number of cycles was estimated. The results are shown in Table 2.3.

Instruction	Software	Hardware	Speedup
Add	652	3	217.33
Sub	1060	3	353.33
Mul	4285	33	129.85
Div	3617	63	57.41

Table 2.3: Cycle Count and Speedups for S.W. and H.W. DFP Instruction

The DFP instructions in the z9 processor are implemented using millicodes, which is something between software and hardware (back to section). Consequently, Table 2.4 compares between the cycle count of this millicodes [51] and the software routines simulated in [55].

Operation	Software	Millcode
Add/Subtract	652 to 1060	100 to 150
Multiply	4285	150 to 200
Divide	3617	350 to 400

Table 2.4: Comparison between Software and Millcodes Execution Cycles

2.4.2 Intel Decimal Library V.S. IBM DecNumber

In [56], authors provide an expanded suite with five benchmarks that support seven DFP types: three of IBM DecNumber (DecNumber with arbitrary precision, DPD64 and DPD128), two of Intel Decimal library (BID64 and BID 128) and two of built in GCC types. It gives an average number of cycles for common DFP operations, Table 2.5 shows the average number of cycles for the Add, Subtract, Multiply and Divide operations using DPD64, DPD128, BID64 and BID128.

	DPD64	DPD128	BID64	BID128
Add	154	233	109	213
Sub	289	580	126	313
Mul	296	453	117	544
Div	627	940	370	1420

Table 2.5: Comparison between Intel library and DecNumber different types

Chapter 3 OPENSPARC T2 CORE ARCHITECTURE

3.1 Chip Multi-Threading

3.1.1 ILP vs. TLP

In the last few decades, most processor architects were targeting desktop workloads. Their design goal was to run the single-threaded instruction as fast as possible. Semiconductor technology has advanced exponentially. It delivered faster transistors operating at multi-GHz frequencies. The number of the available transistors doubled approximately every two years and moreover the frequency doubled every four years.

Architects benefited from these abnormal advances. They developed many complicated techniques to increase the single-thread instruction's speed and improve the instruction level parallelism (ILP) such as:

- Superscalar microprocessors: Intel Pentium M Processor [57], MIPS R10000 [58], Sun Microsystems UltraSparc-II [59].
- Out-of-order execution: PowerPC 620 [60].
- Deep pipelining: MIPS R4000-series [61].
- Complicated branch prediction techniques: PowerPC 620 [60].

However, there are many challenges that limit further improvements in overall performance using these techniques. These challenges are mainly due to two reasons: the power wall and the memory wall.

- Power wall

  Increasing clock frequency needs more cost-effective cooling methods which put a limit on this increase.

- Memory wall

  ILP designs are targeting to decrease the instruction's execution time, but do nothing with the memory latency. Architects tried to overcome this latency problem using the out-of-order execution. Although out-of-order technique can overlap some memory latency with execution, it is limited to shorter memory latency such as Level1 (L1) cache miss and L2 hit. Larger memory latencies are hard to be overlapped with execution. Deeper complicated pipelining, even

if it decreased the single-threaded instruction's execution time, doesn't translate these decreasing into significant performance improvements.

Due to the aforementioned reasons, architects are searching for new methods instead of the conventional ILP like chip multiprocessors (CMP) [62], simultaneous multithreading (SMT) [63] and chip multithreading (CMT) [64]. These new design methods are much more suitable for commercial workloads which employ a relatively higher degree of threading level parallelism (TLP).

CMP means integrating more than one core processor onto the same chip. For commercial workloads, the total performance of the amalgamated cores can be many times that of a single-core processor. Also, those cores share chip resources such as memory controller and L2 and L3 caches. This increases the resources utilization.

CMT means supporting simultaneous execution of many hardware threads per core. It enables the threads to share the core's resources to overlap the long latencies of the off-chip misses (memory wall) and hence increases the hardware utilization. It also overcomes the power wall by decreasing the frequency. As the power consumption-frequency relation is a cubic relation [56], decreasing the frequency to the half and doubling the number of cores will get the same performance (assuming commercial loads with TLP) and abase the power consumption by a factor of four.

3.1.2 CMT Processors' History

The first CMP processor was Stanford Hydra CMP. It is proposed in 1996 and integrated four MIPS-based processors on a single chip [62], [65].Piranha was a processor presented by a co-operating team from DEC and Compaq. It is composed of eight Alpha cores and an L2 cache on the same chip [66].

3.1.2.1 IBM towards CMT processors

In 2001, POWER4 was introduced as a dual-core processor [67], followed by announcing Power5 as a dual core processor with each core supports 2-way SMT [68], the L2 and L3 caches are shared between cores in both POWER4 and POWER5.

The design of the POWER6 microprocessor, announced in 2007, extends IBM leadership by introducing a high-frequency core design coupled with a cache

hierarchy and memory subsystem specifically tuned for the ultrahigh-frequency multithreaded cores [49]. POWER7 was the IBM breakthrough towards CMT processors. It consists of eight cores. Each core can support 4-way SMT. As in POWER4,5,6 the L2 and L3 caches were shared [69].

3.1.2.2 Sun towards CMT processors

In 1995 MAJC architecture is defined by Sun as the first industrial architecture for general purpose CMT processors [70], then Sun announced the first sun MAJC processor (MAJC-5200). MAJC-5200 was a dual-core processor with a shared L1 cache [71].

Later in 2003, Sun announced two CMP processors Gemini [72] and UltraSparc IV [73]. Both were as basic as dual-core processors with no shared resources at all. After that, UltraSparc IV was developed to include the sharing of on-chip L2 cache and off-chip L3 cache between the two cores [73].

3.1.2.2.1 OpenSparc T1

OpenSPARC T1 is a single-chip multiprocessor. OpenSPARC T1 contains eight SPARC physical processor cores. Each SPARC physical processor core has full hardware support for four virtual processors (or “strands”). These four strands run simultaneously, with the instructions from each of the four strands executed round-robin by the single-issue pipeline. When a strand encounters a long-latency event, such as a cache miss, it is marked unavailable and instructions are not issued from that strand until the long-latency event is resolved. Round-robin execution of the remaining available strands continues while the long-latency event of the first strand is resolved.

Each OpenSPARC T1 physical core has a 16-Kbyte, 4-way associative instruction cache (32-byte lines), 8-Kbyte, 4-way associative data cache (16- byte lines), 64-entry fully associative instruction Translation Lookaside Buffer (TLB), and 64-entry fully associative data TLB that are shared by the four strands. The eight SPARC physical cores are connected through a crossbar to an on-chip unified 3-Mbyte, 12-way associative L2 cache (with 64-byte lines).

The L2 cache is banked four ways to provide sufficient bandwidth for the eight OpenSPARC T1 physical cores. The L2 cache connects to four on-chip DRAM controllers, which directly interface to DDR2-SDRAM. In addition, an on-chip J-Bus controller and several on-chip I/O-mapped control registers are accessible to the SPARC physical cores. Traffic from the J-Bus coherently interacts with the L2 cache [74].

3.1.2.2.2 OpenSparc T2

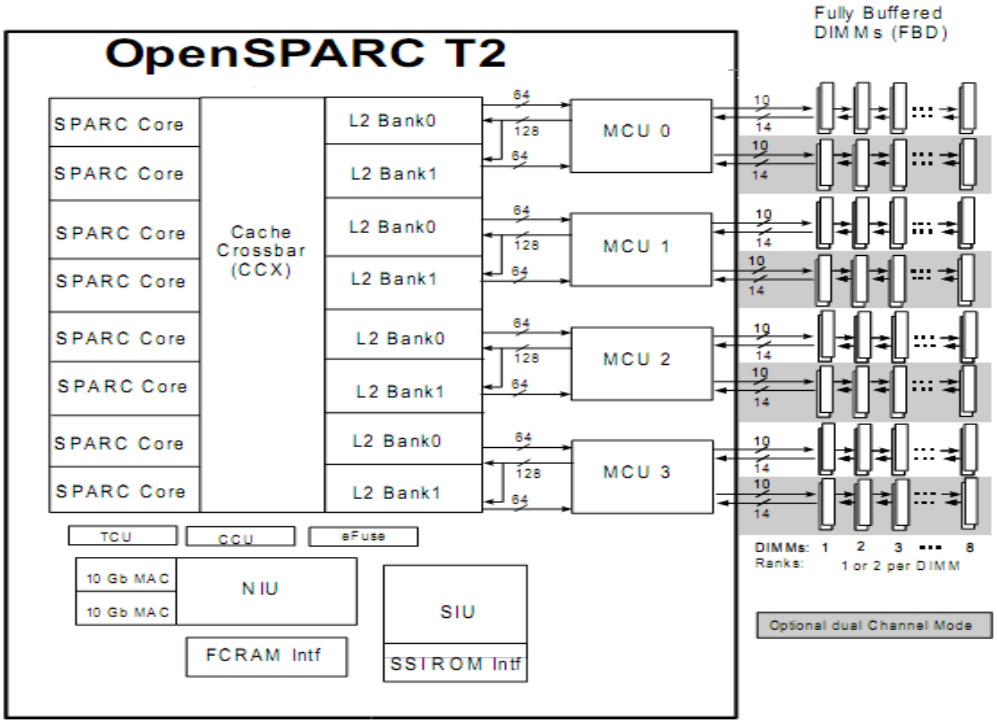


Figure 3.1: OpenSPARC T2 Chip

OpenSPARC T2 shown in Figure 3.1 is a single chip multithreaded (CMT) processor. It contains eight SPARC physical processor cores. Each SPARC physical processor core has full hardware support for eight processors, two integer execution pipelines, one floating-point execution pipeline, and one memory pipeline. The floating-point and memory pipelines are shared by all eight strands.

The eight strands are hard-partitioned into two groups of four, and the four strands within a group share a single integer pipeline. While all eight strands run simultaneously, at any given time at most two strands will be active in the physical core, and those two strands will be issuing either pair of integer pipeline operations, an integer operation and a floating-point operation, an integer operation and a memory operation, or a floating-point operation and a memory operation. Strands are switched on a cycle-by-cycle basis between the available strands within the hard-partitioned group of four, using a least recently issued priority scheme.

When a strand encounters a long-latency event, such as a cache miss, it is marked unavailable and instructions will not be issued from that strand until the

long-latency event is resolved. Execution of the remaining available strands will continue while the long-latency event of the first strand is resolved.

Each OpenSPARC T2 physical core has a 16-Kbyte, 8-way associative instruction cache (32-byte lines), 8-Kbyte, 4-way associative data cache (16-byte lines), 64-entry fully associative instruction TLB, and 128-entry fully associative data TLB that are shared by the eight strands. The eight OpenSPARC T2 physical cores are connected through a crossbar to an on-chip unified 4-Mbyte, 16-way associative L2 cache (64-byte lines).

The L2 cache is banked eight ways to provide sufficient bandwidth for the eight OpenSPARC T2 physical cores. The L2 cache connects to four on-chip DRAM Controllers, which directly interface to a pair of fully buffered DIMM (FBD) channels. In addition, two 1-Gbit/10-Gbit Ethernet MACs and several on-chip I/O-mapped control registers are accessible to the SPARC physical cores. [75].

As our work is based on this processor, its architecture will be explained in details in this chapter.

3.1.2.3 CMT in General-Purpose Processors

This revolution towards CMT processors isn't limited on commercial workloads, it is extended to general-purpose processor designs as well. Intel has now a bunch of multicore processors such as its dual-core processors family, quad-core processor family, core i3 processor family, core i5 processor family and core i7 processor family. AMD also has families of multicore processors such as AMD Athlon™ X2 Dual-Core Processor Product Data Sheet and the triple-core and quad-core options in Family 10h AMD Phenom™ Processor.

3.2 OpenSPARC T2 Core Microarchitecture

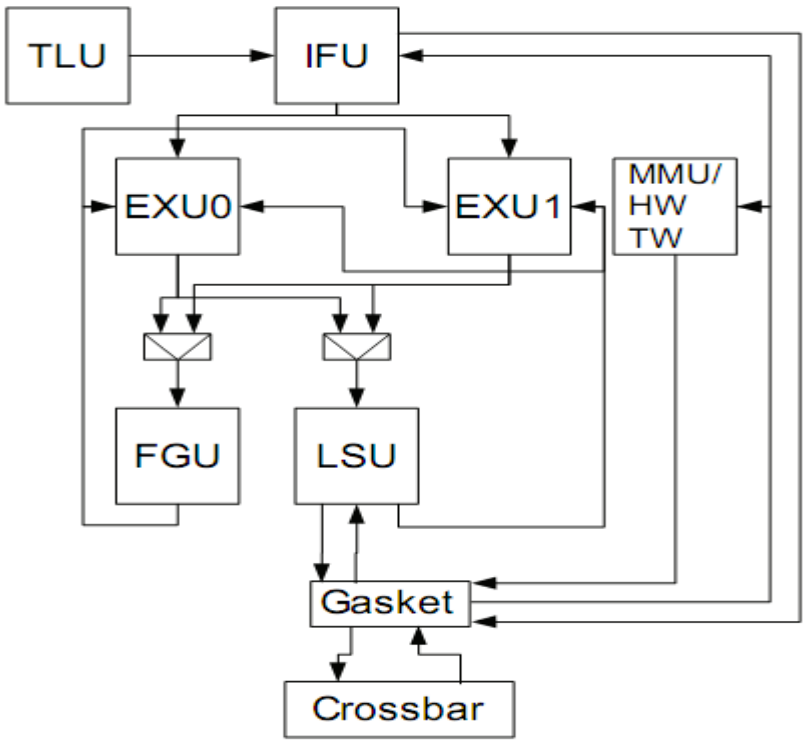


Figure 3.2: OpenSPARC T2 Core Architecture

Figure 3.2 demonstrates the architecture of one physical core of OpenSPARC T2 processor. It consists of a Trap Logic Unit (TLU), an Instruction Fetch Unit (IFU), two EXecution Units (EXU0 and EXU1), a Floating point and Graphical Unit (FGU), a Load Store Unit (LSU), a Gasket, a Cache Crossbar (CCX), and a Memory Management Unit (MMU).

3.2.1 Instruction Fetch Unit (IFU)

The IFU provides instructions to the rest of the core. The IFU generates the Program Counter (PC) and maintains the instruction cache (icache). The IFU contains three subunits: the fetch unit, pick unit, and decode unit, see Figure 3.3.

3.2.1.1 Fetch Unit

OpenSPARC T2 has an 8-way set associative, 16 KB instruction cache (icache) with a 32 byte line. Each cycle the fetch unit fetches up to four instructions for one thread. The fetch unit is shared by all eight threads of OpenSPARC T2 and only

one thread is fetched at a time. The fetched instructions are written into instruction buffers (IBs) which feed the pick logic. Each thread has a dedicated 8 entry IB.

The fetch unit maintains all PC addresses for all threads. It redirects threads due to branch mispredicts, LSU synchronization, and traps. It handles instruction cache misses and maintains the Miss Buffer (MB) for all threads. The MB ensures that the L2 does not receive duplicate icache misses.

3.2.1.2 Pick Unit

The pick unit attempts to find two instructions to execute among eight different threads. The threads are divided into two different thread groups of four threads each: TG0 (threads 0-3) and TG1 (threads 4-7). The Least Recently Picked (LRP) ready thread within each thread group is picked each cycle.

The pick process within a thread group is independent of the pick process within the other thread group. This independence facilitates a high frequency implementation. In some cases, hazards arise because of this independence. For example, each thread group may pick an FGU instruction in the same cycle. Since OpenSPARC T2 has only one FGU, hardware hazard results. The decode unit resolves hardware hazards that result from independent picking.

3.2.1.3 Decode Unit

The decode unit decodes one instruction from each thread group (TG0 and TG1) per cycle. Decode determines the outcome of all instructions that depend on the CC and FCC bits (conditional branches, conditional moves, etc.). The integer source operands rs1 and rs2 are read from the IRF during the decode stage. The integer source for integer stores is also read from the IRF during the decode stage. The decode unit supplies pre-decodes to the execution units.

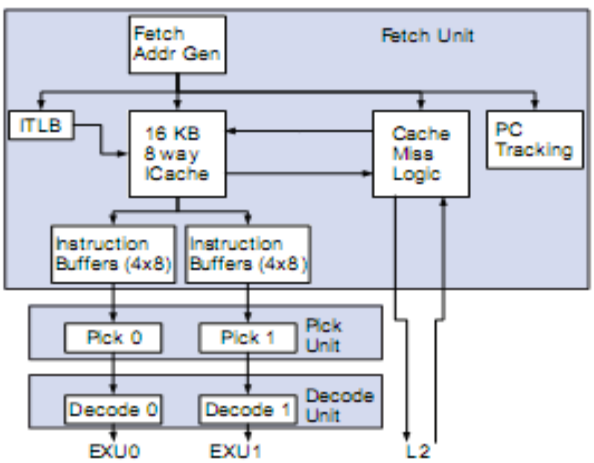


Figure 3.3: Instruction Fetch Unit (IFU)

The decode unit resolves scheduling hazards not detected during the pick stage between the two thread groups. These scheduling hazards include:

- Both TG0 and TG1 instructions require the LSU AND the FGU unit (storeFGU- storeFGU hazard)
- Both TG0 and TG1 instructions require the LSU (load-load hazard, including all loads and integer stores)
- Both TG0 andTG1 instructions require the FGU (FGU-FGU hazard)
- Either TG0 or TG1 is a multiply and a multiply block stall is in effect (multiply block hazard)
- Either TG0 or TG1 require the FGU unit and a PDIST block is in effect (PDIST block hazard)

3.2.2 The Execution Unit (EXU)

OpenSPARC T2 has two execution units. One supports Thread Group1 (TG1) which contains thread0 through 3, and the other supports Thread Group2 (TG2) which contains thread4 through 7. The Execution Unit performs the following tasks:

- Executes all integer arithmetic and logical operations except for integer multiplies and divides.
- Calculates memory and branch addresses.
- Handles all integer source operand bypassing.

Its block diagram is shown in Figure 3.5. It is composed of the following subunits:

- Arithmetic Logic Unit (ALU)
- Shifter (SHFT)
- Operand Bypass (BYP): rs1, rs2, rs3, and rcc bypassing.
- Integer Register File (IRF)
- Register Management Logic (RML)

The integer execution pipeline takes eight stages as shown in Figure 3.4.

Fetch	Cache	Pick	Decode	Execute	Mem	Bypass	W

Figure 3.4: Integer Execution Pipeline

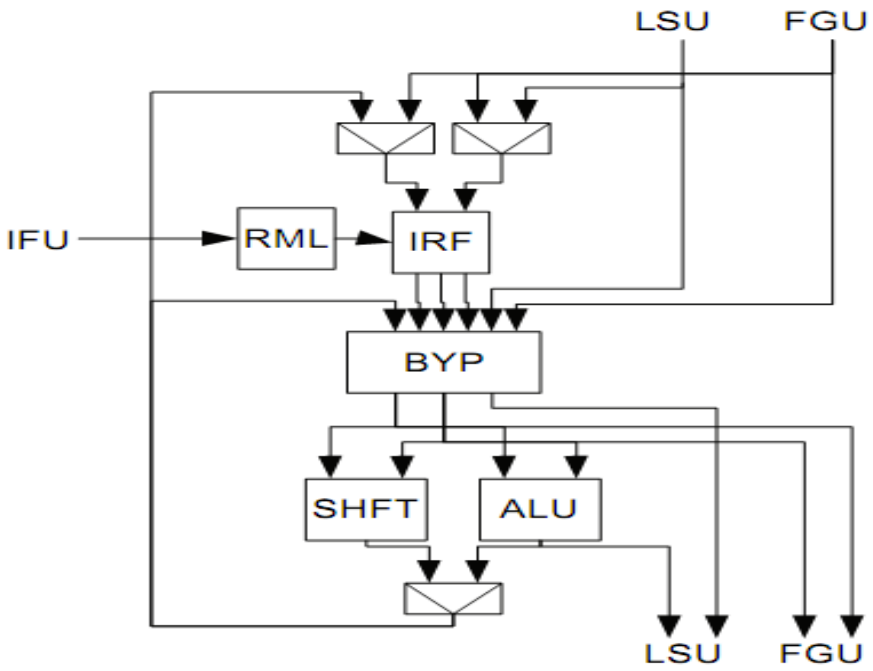


Figure 3.5: Integer Execution Unit (EXU)

3.2.2.1 Differences from T1 execution unit

OpenSPARC T2 adds an additional bypass stage between the memory stage and the write back stage. There are some integer instructions that are not executed in the execution unit. They are executed in the floating point unit instead. These instructions are the integer multiply, integer divide, multiply step (MULSCC), and population count (POPC). The execution unit reads the operand from the Integer File Register (IRF). Multiplexers below the two EXUs provide instruction and integer operand data to the FGU.

Also, to support VIS 2.0, the EXU executes Edge instructions, Array addressing instructions, and the BMASK instruction. Edge instructions handle boundary conditions for parallel pixel scan line loops. Array addressing instructions convert three dimensional (3D) fixed point addresses contained in rs1 to a blocked-byte address and store the result in rd. These instructions specify an element size of 8 (ARRAY8), 16 (ARRAY16), and 32 bits (ARRAY32). The rs2 operand specifies the power-of-two size of the X and Y dimensions of a 3D image array. BMASK adds two-integer registers, rs1 and rs2, and stores the result in rd. The least significant 32 bits of the result are stored in the General Status Register (GSR.mask) field.

3.2.3 Load Store Unit (LSU)

3.2.3.1 Block Diagram

The block diagram of the LSU is shown in Figure 3.6. It consists of the following units:

- DCA and DTAG make up the level 1 data cache.
- Data Translation Lookaside Buffer (DTLB): provides virtual to physical and real to physical address translation for memory operations.
- Load Miss Queue (LMQ): stores the currently pending load miss for each thread (each thread can have at most one load miss at a time).
- Store Buffer (STB): contains all outstanding stores.
- Processor to Cache Crossbar (PCX) interface (PCXIF): controls outbound access to the PCX and ASI controller.
- Cache to Processor Crossbar (CPX) interface (CPXIF): receives CPX packets (load miss data, store updates, ifill data, and invalidates), stores them in a FIFO (the CPQ), and sends them to the data cache.

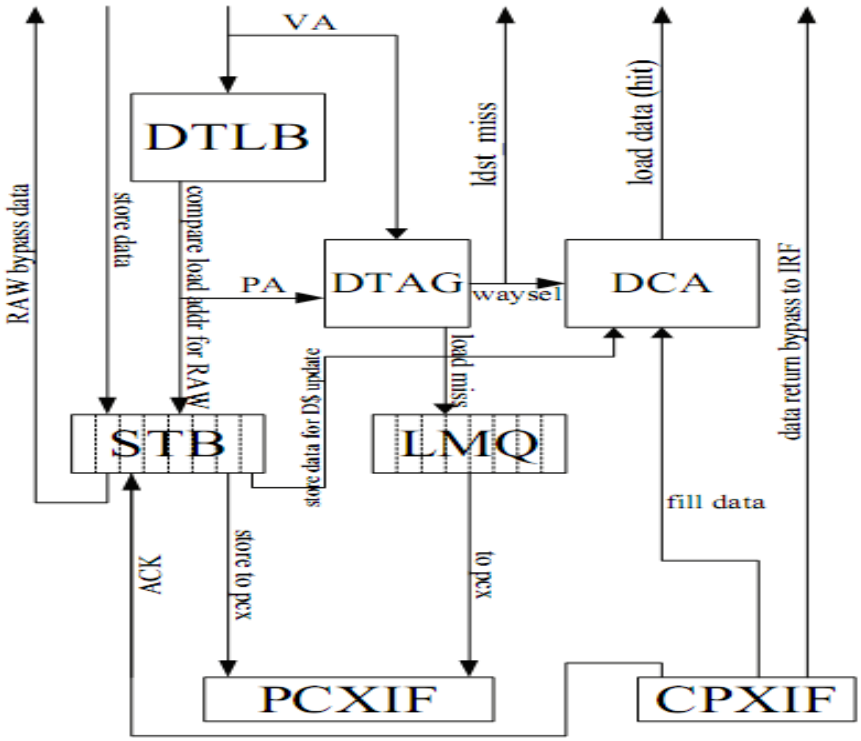


Figure 3.6: Load Store Unit (LSU)

3.2.3.2 LSU Pipeline

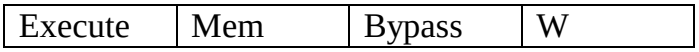


Figure 3.7: LSU Pipeline

E (Execute): The virtual address and store data are received from the EXU. Most control signals from decode arrive in this cycle.

M (Memory): The TLB performs address translation. D$tags and data are accessed here. Tag comparison with the PA is performed at the end of the cycle in the TLB. FP store data comes in this cycle.

B (Bypass): For loads, way select, data alignment, and sign extension done before the load result is transmitted to EXU/FGU. The store buffer is checked this cycle for RAW hazards. For stores, the PA is written into the store buffer and store data is formatted and ECC generated.

W (Write back): Store data is written into the buffer. Instructions which were not flushed prior to this point are now committed.

Load data can bypass from the B stage to an instruction in the D (Decode) stage of the pipeline. This means that a dependent instruction can issue two cycles after a load.

3.2.3.3 Writing in the Dcache

The Dcache uses the write-through mechanism to write data in the cache. If a store hits in the L1 cache, it updates the Dcache. If it misses in the L1 cache, data is stored in the L2 cache directly. To maintain coherency between L1 and L2 caches, a copy of the L1 tags exists in L2 cache, and any updates or invalidations occur only after receiving an acknowledgment from L2 cache.

3.2.3.4 Reading from the Dcache

If a load hits, the Dcache does not make a request to the L2 cache. If a load misses, the Dcache makes a request to the L2 cache through PCX. When data is loaded from L2 cache, it writes in the CPQ FIFO and waits for a hole in the Dcache pipe and a free writing port on the targeted register file.

The load miss path shares the w2 port of the floating point register file (FRF) with the divide pipeline. The divide pipeline has higher priority at the w2 port, so if there is a division operation near completion, the FGU signals the LSU

to stall the data return for one cycle. This delay does not happen when the load data targets the integer register file (IRF) as the load path is the only source for the write port w2 of IRF.

The data cache is an 8 KB, 4-way set associative cache with 16 B lines. The data is stored in the DCA array, the tags are stored the DTAG array, the valid bits are stored the DVA array, and the used bits are stored the LRU array. DCA and DTA are single ported memories. Each line requires a physical tag of 29 bits (40 bit PA minus 11 bit cache index) plus one parity bit. The dcache is write-through as described before and it is parity protected with one parity bit for each byte of data and another parity bit for the entire 29 bit tag.

3.2.4 Cache Crossbar (CCX)

Cache Crossbar (CCX) connects the 8 SPARC cores to the 8 banks of the L2 cache. An additional port connects the SPARC cores to the IO bridge. A maximum of 8 load/store requests from the cores and 8 data returns/acks/invalidations from the L2 can be processed simultaneously. The cache crossbar is divided into two blocks: the processor-to-cache crossbar (PCX) and the cache-to-processor crossbar (CPX). Both has N x M bussed mux structure. The PCX has N=8 (SPARC cores) and M=9 (8 L2 banks + IO). The CPX has N=9 and M=8, see Figure 3.8.

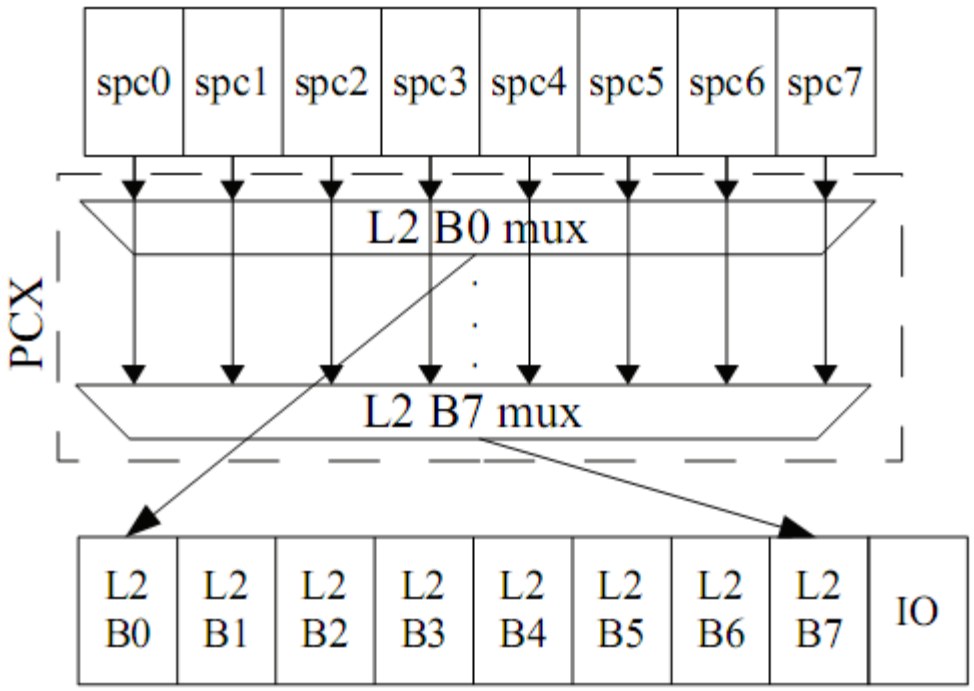


Figure 3.8: Processor-to-Cache Crossbar (PCX)

Sources issue requests to the crossbar. The crossbar queues requests and data to the different targets. Since multiple sources can request access to the same target, arbitration within the crossbar is required. Priority is given to the oldest requestor(s) to maintain fairness and ordering. The arbitration requirements of the PCX and CPX are identical except for the numbers of sources and targets that must be handled. The CPX must also be able to handle multicast transactions. A three-cycle arbitration protocol is used. The protocol consists of three steps: Request, Arbitrate, and Grant. The PCX Timing pipeline is shown in Figure 3.9.

PQ	PA	PX
SPARC cores issue requests	SPARC cores send packets to PCX Queue the packets Arbitration for target Send the grant to the muxes	Transmit grant to SPARC core Perform data muxing

Figure 3.9: PCX Timing pipeline

3.2.5 Floating point and Graphics Unit (FGU)

3.2.5.1 Block Diagram

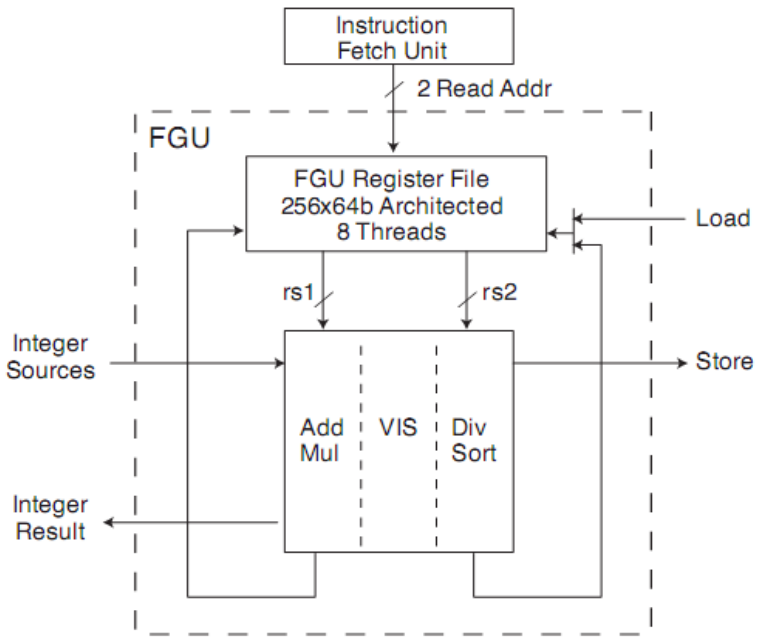


Figure 3.10: Floating Point and Graphics Unit (FGU)

Block diagram of the FGU is shown in Figure 3.10. The OpenSPARC T2 floating-point and graphics unit (FGU) implements the SPARC V9 floating-point instruction set, the SPARC V9 integer multiply, divide, and population count (POPC) instructions, and the VIS 2.0 instruction set. The only exception is that all quad precision floating point instructions are unimplemented.

3.2.5.2 FGU features

Unlike OpenSPARC T2 which has one FGU shared between all cores, OpenSPARC T2 Contains one dedicated FGU per core. Each FGU complies with the IEEE 754 standard. It supports IEEE 754 single-precision (SP) and double-precision (DP) data formats, but quad precision floating-point operations are unimplemented. It also supports all IEEE 754 floating-point data types (normalized, denormalized, NaN, zero, infinity) with the exception that certain denormalized operands or expected results may generate an unfinished_FPop trap to software, indicating that the FGU was unable to generate the correct results.

3.2.5.3 Architecture

FGU Includes three execution pipelines (Figure 3.11):

- Floating-point execution pipeline (FPX)
- Graphics execution pipeline (FGX)
- Floating-point divide and square root pipeline (FPD)

Up to one instruction per cycle can be issued to the FGU. Instructions for a given thread are executed in order. FGU operations are pipelined across threads. A maximum of two FGU instructions (from different threads) may write back into the FRF in a given cycle (one FPX/FGX result and one FPD result). FPX, FGX, and FPD pipelines never stall.

All FGU-executed instructions, except floating-point and integer divides and floating-point square root are fully pipelined, single pass instructions. It has a single-cycle throughput and a fixed six-cycle execution latency, independent of operand values. Divide and square root are not pipelined but execute in a dedicated datapath. Floating-point divide and square root have a fixed latency. Integer divide has a variable latency, dependent on operand values.

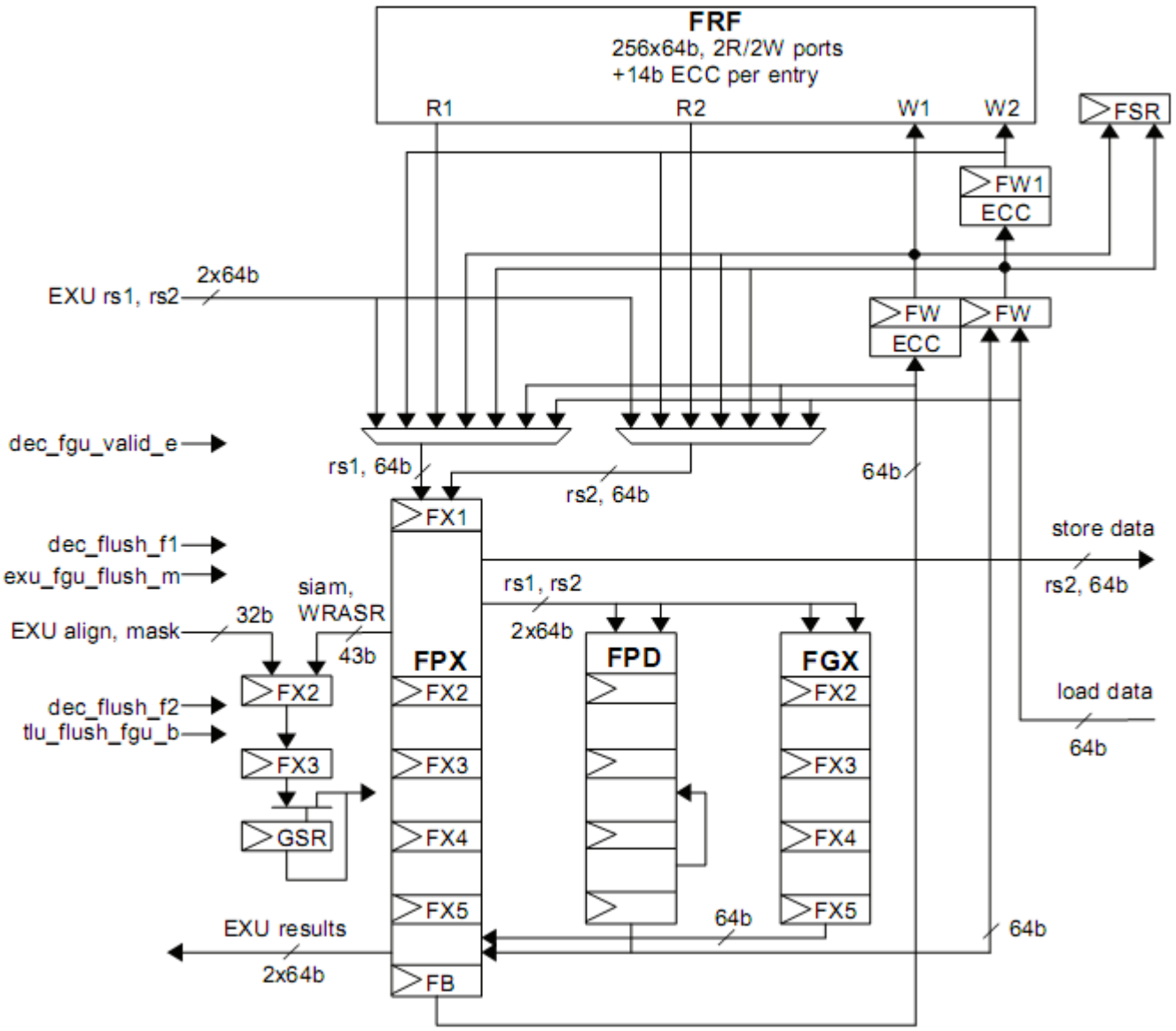


Figure 3.11: FGU Pipelines

FGU has four related registers:

- The Floating Point Register File (FRF).
- The Floating Point State Register (FSR).
- The General Status Register (GSR).
- The Floating-point Registers State (FPRS)

3.2.5.3.1 Floating Point Register File (FRF)

The floating point register file (FRF) is a 256-entry × 64-bit with two read and two write ports. Floating-point store instructions share an FRF read port with the execution pipelines. Write port (W1) is dedicated to FPX and FGX results. Arbitration is not necessary for the FPX/FGX write port (w1) because of single instruction issue and fixed execution latency constraints. The other port (W2) is dedicated to floating-point loads and FPD floating-point results. FPD results always have highest priority for W2. The FRF supports eight-way multithreading (eight threads) by dedicating 32 entries for each thread. Each register file entry also includes 14 bits of ECC for a total of 78 bits per entry. Correctable ECC errors (CEs) and uncorrectable ECC errors (UEs) result in a trap if the corresponding enables are set.

3.2.5.3.2 Floating Point State Register (FSR)

The Floating-Point State register (FSR) fields, illustrated in Figure 3.12, contain FPU mode and status information. Bits 63–38, 29–28, 21–20 and 12 of FSR are reserved.

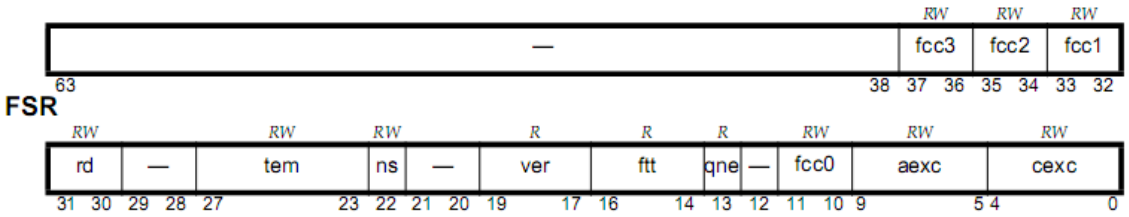


Figure 3.12: FSR bits

Floating Point Condition Codes

Execution of a floating-point compare instruction (FCMP or FCMPE) updates one of the fccn fields in the FSR, as selected by the compare instruction, see Table 3.1.

Content of fccn	Relation
0	F[rs1] = F[rs2]
1	F[rs1] < F[rs2]
2	F[rs1] > F[rs2]
3	F[rs1] ? F[rs2] (unordered)

Table 3.1: FP Condition Codes

3.2.5.3.2.1 Rounding Direction (rd)

Bits 31 and 30 select the rounding direction for floating-point results according to IEEE Standard 754-1985. Table 3.2 shows the encodings. If the

interval mode bit of the General Status register has a value of 1 (GSR.im =1), then the value of FSR.rd is ignored and floating-point results are instead rounded according to GSR.irnd.

Rd	Round Toward
0	Nearest (even, if tie)
1	0
2	+ inf
3	- inf

Table 3.2: Rounding Modes

3.2.5.3.2.2 Non-Standard Floating Point (ns)

When FSR.ns = 1, it causes a SPARC V9 virtual processor to produce implementation-defined results that may or may not correspond to IEEE Std 754-1985. For an implementation in which no nonstandard floating-point mode exists, the ns bit of FSR should always read as 0 and writes to it should be ignored.

3.2.5.3.2.3 FPU Version (ver)

Bits 19 through 17 identify one or more particular implementations of the FPU architecture. FSR.ver = 7 is reserved to indicate that no hardware floating-point controller is present. The ver field of FSR is read-only; it cannot be modified by the LDFSR or LDXFSR instructions.

3.2.5.3.2.4 Floating Point Trap Type (ftt)

FSR.ftt encodes the floating-point trap type that caused the generation of an fp_exception_other or fp_exception_ieee_754 exception. It is possible for more than one such condition to occur simultaneously; in such a case, only the highest-priority condition will be encoded in FSR.ftt, see Table 3.3 for details.

Condition Detected During FPop	Relative Priority (1 = highest)	Result	
		FSR.ftt Set to Value	Exception Generated
Invalid_fp_register	20	6	Fp_exception_other
Unfinished_754_exception	30	2	Fp_exception_other
IEEE_754_exception	40	1	Fp_exception_ieee_754
Reserved	—	3, 4, 5, 7	—
(none detected)	—	0	—

Table 3.3: FP Trap Types

3.2.5.3.2.5 FQ not Empty (qne)

Since OpenSPARC T2 does not implement a floating-point queue, FSR.qne always reads as zero and writes to FSR.qne are ignored.

3.2.5.3.2.6 Trap Enable Mask (tem)

Bits 27 through 23 are enable bits for each of the five IEEE-754 floating-point exceptions that can be indicated in the current_exception field (cexc). See Table 3.4 (a), where “nv” fields are related to invalid exception, “of” fields are related to overflow exception, “uf” fields are related to underflow exception, “dz” fields are related to division by zero exception, and finally “nx” fields are related to inexact exception.

Bit	27	26	25	24	23	9	8	7	6	5	4	3	2	1	0
	nvm	Ofm	ufm	Dzm	Nxm	Nva	Ofa	Ufa	dza	nxa	Nvc	ofc	ufc	dzc	nxc
field	Tem					Aexc					cexc				
	(a)					(b)					(c)				

Table 3.4: (a) tem (b) aexc (c) cexc

If a floating-point instruction generates one or more exceptions and the tem bit corresponding to any of the exceptions is 1, then this condition causes an fp_exception_ieee_754 trap. A tem bit value of 0 prevents the corresponding IEEE 754 exception type from generating a trap.

3.2.5.3.2.7 Current Exception (cexc)

FSR.cexc (FSR {4:0}) indicates whether one or more IEEE 754 floating-point exceptions were generated by the most recently executed FPop instruction. The absence of an exception causes the corresponding bit to be cleared (set to 0). See Table 3.4 (c).

3.2.5.3.2.8 Accrued Exceptions (aexc)

Bits 9 through 5 accumulate IEEE_754 floating-point exceptions as long as floating-point exception traps are disabled through the tem field. See Table 3.4 (b).

3.2.5.3.2.9 General Status Register (GSR)

The General Status Register (GSR) is the nineteenth register in the Ancillary State Registers (ASR). It is implicitly referenced by many Visual Instruction Set (VIS) instructions. The GSR is illustrated in Figure 3.13 and described in Table 3.5.

	Mask		__	im	irnd	__	scale	align
bits	63	32	31 28	27	26 25	24 8	7 3	2 0

Figure 3.13: GSR bits

Bit	Field	Description
63:32	Mask	This 32-bit field specifies the mask used by the BSHUFFLE instruction. The field contents are set by the BMASK instruction.
31:28	__	*Reserved*
27	Im	Interval Mode: If GSR.im = 0, rounding is performed according to FSR.rd;if GSR.im = 1, rounding is performed according to GSR.irnd.
26:25	Irnd	IEEE Std 754-1985 rounding direction to use in Interval Mode (GSR.im = 1) as follows: 0 – Nearest(even; if tie) 1 – 0 2 – + inf 3 – -inf
24:8	__	*Reserved*
7:3	Scale	5-bit shift count in the range 0–31, used by the FPACK instructions for formatting.
2:0	Align	Least three significant bits of the address computed by the last-executed ALIGNADDRESS or ALIGNADDRESS_LITTLE instruction.

Table 3.5: GSR Description

3.2.5.3.3 Floating-Point Registers State (FPRS) Register

The Floating-Point Registers State (FPRS) register, shown in Figure 3.14, contains control information for the floating-point register file.

	Fef	Du	dl
Bits	2	1	0

Figure 3.14: FPRS fields

3.2.5.3.3.1 Enable FPU (fef)

Bit 2, fef, determines whether the FPU is enabled or disables. If it is disabled, executing a floating-point instruction causes an fp_disabled trap. If this bit is set (FPRS.fef = 1) but the PSTATE.pef bit is not set (PSTATE.pef = 0), then executing a floating-point instruction causes an fp_disabled exception; that is, both FPRS.fef and PSTATE.pef must be set to 1 to enable floating-point operations.

3.2.5.3.3.2 Dirty Upper Registers (du)

Bit 1 is the "dirty" bit for the upper half of the floating-point registers; that is, F[32]–F[62]. It is set to 1 whenever any of the upper floating-point registers is modified.

3.2.5.3.3.3 Dirty Lower Registers (dl)

Bit 0 is the "dirty" bit for the lower 32 floating-point registers; that is, F[0]–F[31]. It is set to 1 whenever any of the lower floating-point registers is modified.

Both dl and du bits are cleared only by software. If the FPU is disabled, neither dl nor du is modified.

3.2.5.4 Interfacing with other units

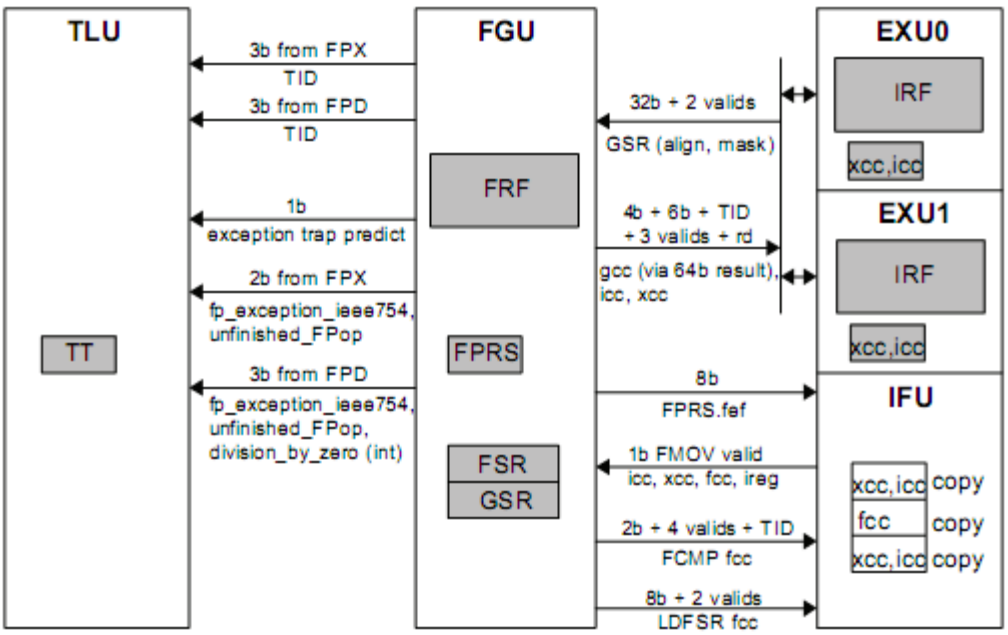


Figure 3.15: FGU Interfaces

3.2.5.4.1 Instruction Fetch Unit (IFU)

The IFU provides instruction control information as well as rs1, rs2, and rd register address information. It can issue up to one instruction per cycle to the FGU.

The IFU does the following:

- Sends the following flush signals to the FGU:
  - Flush execution pipeline stage FX2 (transmitted during FX1/M stage)
  - Flush execution pipeline stage FX3 (transmitted during FX2/B stage)
- Maintains copies of fcc for each thread.
- Provides a single FMOV valid bit to the FGU indicating whether the appropriate icc, xcc, fcc, or ireg condition is true or false.

The FGU does the following:

- Flushes the FPD based on the IFU- and trap logic unit (TLU)-initiated flush signals. Once an FPD instruction has executed beyond FX3, it cannot be flushed by an IFU- or TLU-initiated flush.
- Provides appropriate FSR.fcc information to the IFU during FX2 and FX3 (including load FSR). The information includes a valid bit, the fcc data, and thread ID (TID) and is non-speculative.
- Provides the FPRS.fef bit to the IFU for each TID (used by the IFU to determine fp_disable).

3.2.5.4.2 Trap logic unit (TLU)

The FGU provides the following trap information to the TLU:

- unfinished_FPop
- fp_exception_ieee_754
- fp_cecc (FRF correctable ECC error)
- fp_uecc (FRF uncorrectable ECC error)
- division_by_zero (integer).
- Exception trap prediction

The FGU receives the following flush signal from the TLU:

- Flush execution pipeline stage FX3 (transmitted during FX2/B stage)

3.2.5.4.3 Load-store unit (LSU)

- Floating-point load instructions share an FRF write port with FPD floating-point results, which always have priority for the shared write port. FPD notifies the IFU and LSU when a divide or square root is near completion to guarantee that load data does not collide with the FPD result.
- Loads instruction may update the FRF or FSR registers. Loads update them directly, without accessing any pipeline. The LSU always delivers 32-bit load data replicated on both the upper (even) and lower (odd) 32-bit halves of the 64-bit load data bus.

3.2.5.4.4 Execution Units

The EXU does the following:

- Each EXU can generate the two 64-bit source operands needed by the integer multiply, divide, POPC, SAVE, and RESTORE instructions.
- The EXUs provide the appropriate sign-extended immediate data for rs2; provide rs1 and rs2 sign extension; and provide zero fill formatting as required. The IFU provides a destination address (rd), which the FGU provides to the EXUs upon instruction completion.
- Each EXU provides GSR.mask and GSR.align fields, individual valid bits for those fields, and the thread ID (TID).

The EXU does the following:

- The FGU provides a single 64-bit result, along with appropriate integer condition codes (icc and xcc).
- The same result bus provides appropriate 64-bit formatted “gcc” information to the EXUs upon completion of the VIS FCMP (pixel compare) instructions. The result information includes a valid bit, TID, and destination address (rd). FGU clears the valid bit under the following conditions:
    - division_by_zero trap (IDIV only)
    - Enabled FRF ECC UE/CE (VIS FCMP only)
    - EXU-, IFU-, or TLU-initiated flush

3.2.6 Trap Logic Unit

3.2.6.1 Functional Description

The Trap Logic Unit (TLU) manages exceptions and trap requests which are conditions that may cause a thread to take a trap. It also manages traps which are vectored transfers of control to supervisor software through a trap table [76]. The TLU maintains processor state related to traps as well as the Program Counter (PC) and Next Program Counter (NPC). If an exception or trap request happens, the TLU prevents the update of architectural state for the instruction or instructions after an exception.

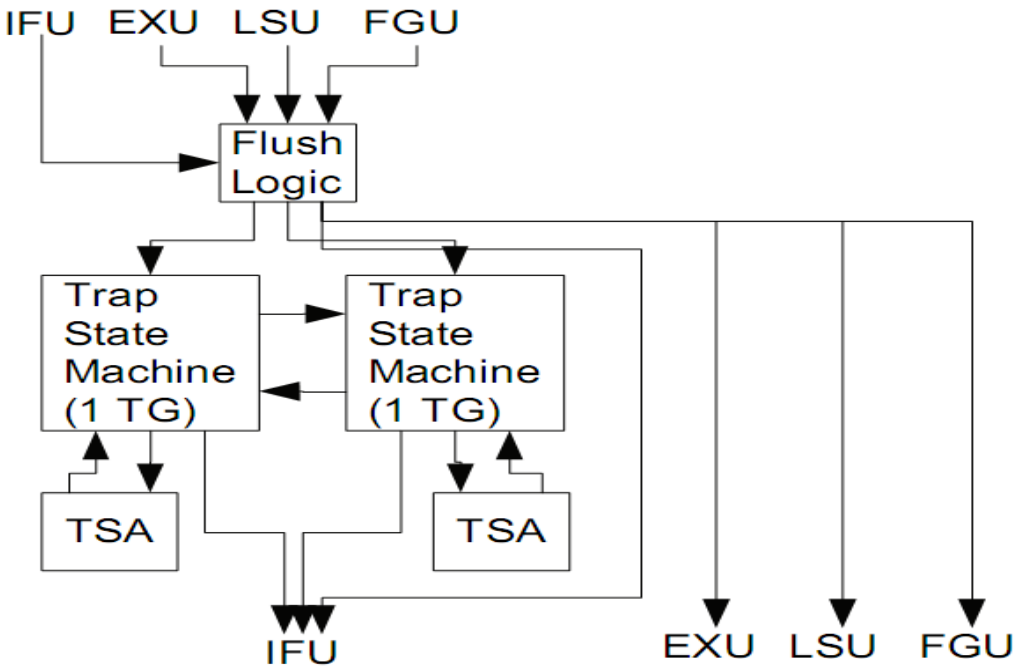


Figure 3.16: TLU Block Diagram

3.2.6.2 Block Diagram

The TLU block diagram shown in Figure 3.16 consists of the following units:

- The Flush Logic generates flushes in response to exceptions to create precise interrupt points (when possible).
- The Trap Stack Array (TSA) maintains trap state for the eight threads for up to six trap levels per thread.
- The Trap State Machine holds and prioritizes trap requests for the eight threads in two thread groups.

3.2.7 Memory Management Unit (MMU)

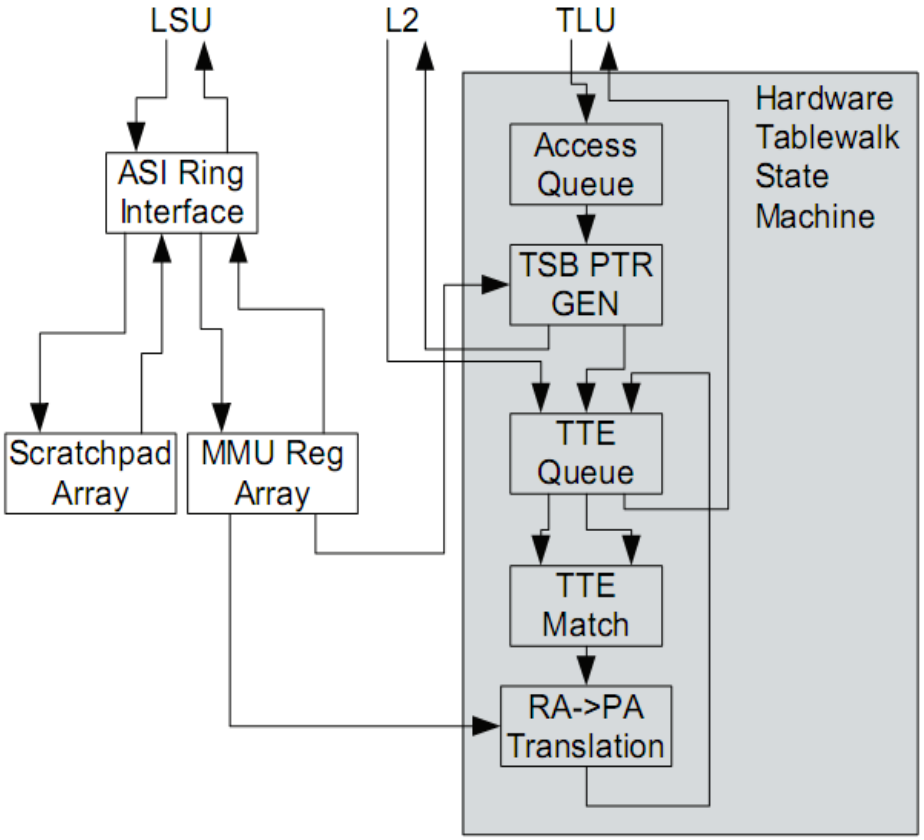


Figure 3.17: MMU Block Diagram

The Memory Management Unit (MMU) shown in Figure 3.17 reads Translation Storage Buffers (TSBs) for the Translation Lookaside Buffers (TLBs) for the instruction and data caches. The MMU receives reload requests for the TLBs and uses its hardware tablewalk state machine to find valid Translation Table Entries (TTEs) for the requested access. The TLBs use the TTEs to translate Virtual Addresses (VAs) and Real Addresses (RAs) into Physical Addresses (PAs). The TLBs also use the TTEs to validate that a request has the permission to access the requested address.

The MMU maintains several sets of Alternate Space Identifier (ASI) registers associated with memory management. Software uses the scratchpad registers in handling translation misses that the hardware tablewalk cannot satisfy; the MMU maintains these registers. The MMU maintains translation error

registers that provide software with the reasons why translation misses occur. Hardware tablewalk configuration registers control how the hardware tablewalk state machine accesses the TSBs. Software reads and writes the TLBs through another set of ASI registers.

Chapter 4 INCLUDING DFPU IN THE ULTRASPARC T2

4.1 Introduction

So far, only two processor architectures include decimal floating point units (z series and power series from IBM). This research provides the first free and open-source alternative to the above two architectures with a processor containing decimal floating point as well as the corresponding tool chain.

This chapter explains how we extended the SPARC instruction set architecture and adapted the UltraSparc T2 architecture from Oracle/Sun to perform DFP operations. Our OpenSPARC T2 version uses a decimal Fused Multiply-Add unit (FMA) designed by our team as the core of the decimal unit. It provides the basic decimal operations (Addition, Subtraction, Multiplication, Fused Multiply-Add (FMA) and Fused Multiply-Subtract (FMS)).

To implement these changes, we add a new unit called Decimal Floating Point Unit (DFPU) then adapt the following already exist units:

- Gasket Unit (GKT).
- The Pick Unit (PKU).
- The Decoding Unit (DEC).
- The Floating Point Control Units (FPC, FAC).
- The Floating Point and Graphics Unit (FGU).

The chapter is organized as follows. Section 4.2 introduces the new instructions we added to the SPARC instruction set architecture. Section 4.3 surveys different design alternatives. Section 4.4 explains the architecture, the operation and the interfaces of the DFPU. Section 4.5 states the modifications done in the core units.

4.2 Extending SPARC ISA to include DFP Instructions

The SPARC instruction set [76] does not have decimal floating point instructions. We need to extend the instruction set to include the decimal floating point support. This is done using the implementation dependent instructions defined in the SPARC v9 architecture (IMPDEP1, IMPDEP2).

10	*Impl-dep*	Op3	*Impl-dep*
31 30	29 25	24 19	18 0

Table 4.1: IMPDEP Opcode format

Opcode	Op3	Operation
IMPDEP1	110110	Implementation-Dependent Instruction 1
IMPDEP2	110111	Implementation-Dependent Instruction 2

Table 4.2: IMPDEP Op3 code

The SPARC V9 architecture provided two instruction spaces that are entirely implementation dependent: IMPDEP1 and IMPDEP2 (Table 4.1 and Table 4.2). In the UltraSPARC Architecture, the IMPDEP1 opcode space is used by the Visual Instruction Set (VIS) instructions. IMPDEP2 is subdivided into IMPDEP2A and IMPDEP2B. IMPDEP2A remains implementation dependent. The IMPDEP2B opcode space is reserved for implementation of the binary floating point multiply-add/multiply- subtract instructions [77].

Although we implemented and tested only the DFP add, sub and multiply, we defined eight new decimal floating point instructions for future extension of our project; four instructions using IMPDEP1 as shown in Table (4.3) and Table (4.4):

- Decimal Floating point Add double (DFADDd)
- Decimal Floating point Subtract double (DFSUBd)
- Decimal Floating point Multiply double (DFMULd)
- Decimal Floating point Division double (DFDIVd)

10	Rd	110110	rs1	opf	rs2
31 30	29 25	24 19	18 14	13 5	4 0

Table 4.3: IMPDEP1-based DFP Instructions format

Opf (3:0)								
Opf(8:4)= '0x09'	0	1	2	3	4	5	6	7
			DFADDd				DFSUBd	
	Opf (3:0)							
	8	9	10	11	12	13	14	15
			DFMULd				DFDIVd	

Table 4.4: Opf field of IMPDEP1-based DFP Instructions

, and four instructions using IMPDEP2 as shown in Table (4.5) and Table (4.6):

- Decimal Fused Multiply-Add double (DFMADDd)
- Decimal Fused Multiply-Subtract double (DFSUBd)
- Decimal Fused Negative Multiply-Add double (DFNMADDd)
- Decimal Fused Negative Multiply-Subtract double (DFNMSUBd)

The Table 4.6 fields that have op5 equal one or two are the binary floating point FMA/FMS instructions defined by the UltraSPARC T2 architecture. They are not implemented in hardware; instead, they are executed by the software layer.

We define and implement only the basic DFP operations. In order to implement all the operations defined by the standard, we will use the same hardware core unit with some extensions. This is out the scope of this thesis and may be considered as a future work.

10	Rd	110111	rs1	rs3	Op5	rs2
31 30	29 25	24 19	18 14	13 9	8 5	4 0

Table 4.5: IMPDEP2-based DFP Instructions format

OP5(1:0)					
OP5(3:2)		0	1	2	3
	0		FMADDs	FMADDd	DFMADDd
	1		FMSUBs	FMSUBd	DFMSUBd
	2		FNMSUBs	FNMSUBd	DFNMSUBd
	3		FNMADDs	FNMADDd	DFNMADDd

Table 4.6: Op5 field of IMPDEP2-based DFP Instructions

4.2.1 Fused Multiply-Add operation (FMA)

As the standard states [18], the Fused Multiply-Add operation for the three operands (A, B, C) ‘FMA(A,B,C)’ computes (A × B) + C as if they were with unbounded range and precision, with rounding only once to the destination format. Moreover, no underflow, overflow, or inexact exception can arise due to multiplication, but only due to addition; and so Fused Multiply-Add differs from a multiplication operation followed by an addition operation. The preferred

exponent is $min(Q(A) + Q(B), Q(C))$ where Q(x) means the exponent of operand x.

This definition of the FMA operation highlights two important restrictions: the intermediate unbounded result of the multiplication and the single rounding step after addition. This clearly shows that this operation produces more accurate result than a multiplication with a result rounded to the required precision then followed by addition with the final result rounded again. The standard also stresses that exception decisions are taken based on the final result and not due to the multiplication step.

4.3 Design alternatives

To include the decimal floating point unit into UltraSPARC T2 core we have two options. We can either include it as a completely separate block like the binary floating point and graphical unit (FGU), EXU0 and EXU1 or merge it inside the FGU.

4.3.1 DFPU as a separate unit

Implementing the decimal floating point unit (DFPU) as a completely new unit has the advantage of a separate DFP datapath and the pipeline in turns is now able to issue instructions to both FGU and DFPU. However, this design will be area consuming and complicate the design due to many reasons:

- Currently the FGU contains the floating-point register file (FRF). The FRF contains the floating point registers for all eight threads. According to the standard specifications [18]; the DFPU and FPU should share the floating point registers. So, a completely separate DFPU will need additional ports to access FRF or arbitration between FGU and DFPU is required.

- A separate interfacing with the Decode Unit is required. The DEC unit, described in chapter 3 section 3.2.1.3, completes the instruction decoding and issues instructions to the floating-point unit. It decodes one instruction from each thread group (TG0 and TG1) per cycle. It needs to select one of them to be issued and stall the other for one cycle if issuing both of them will cause a hazard. This is performed by an arbiter that alternately picks one thread or the other. With the new DFPU, new alternatives exist and need to be identified by the arbiter. All combinations between (Integer, floating, and decimal) are possible except: the two instructions are floating-

point or the two instructions are decimal floating point. In these cases the decode block must select one of them to be issued and stall the other.

- The pipeline is now able to issue instructions to both FGU and DFPU, and the standard states that decimal and binary implementations should have the same register for flags. Consequently, the trap logic unit (TLU) must be modified to handle exceptions from both units. Currently, the FGU sends floating-point exception signals along with a 3-bit Trap ID (TID) to the TLU if an exception occurred. In addition, a floating-point trap prediction signal is sent four cycles earlier if a trap is likely to be taken. These signals are also needed if DFPU is defined as a completely separate block.

Finally, no separate interface with the load-store unit (LSU) is required. The FRF interacts with the LSU for floating-point loads and stores. The FRF is common between FGU and DFPU. As a result, no separate interface is required.

4.3.2 DFPU as a merged pipeline inside FGU

The second option was modifying the FGU itself to handle the decimal floating point operations; by another words, including our DFPU inside the FGU. This will get rid of all the aforementioned complex interfaces making the design much simpler and the area much smaller.

On the other hand, it will complicate the FGU design itself. It also has a limitation on the two issued instructions running simultaneously. A DFP instruction and BFP instruction are both considered floating point instructions; hence, they cannot run on the same cycle. Because program is unlikely to require both binary and decimal floating-point computations simultaneously [49], this limitation is not a problem in major cases. Hence; targeting simpler and less-area design, merging the DFPU inside the FGU as shown in Figure 4.4 is preferred. The design details are explained throughout the next sections.

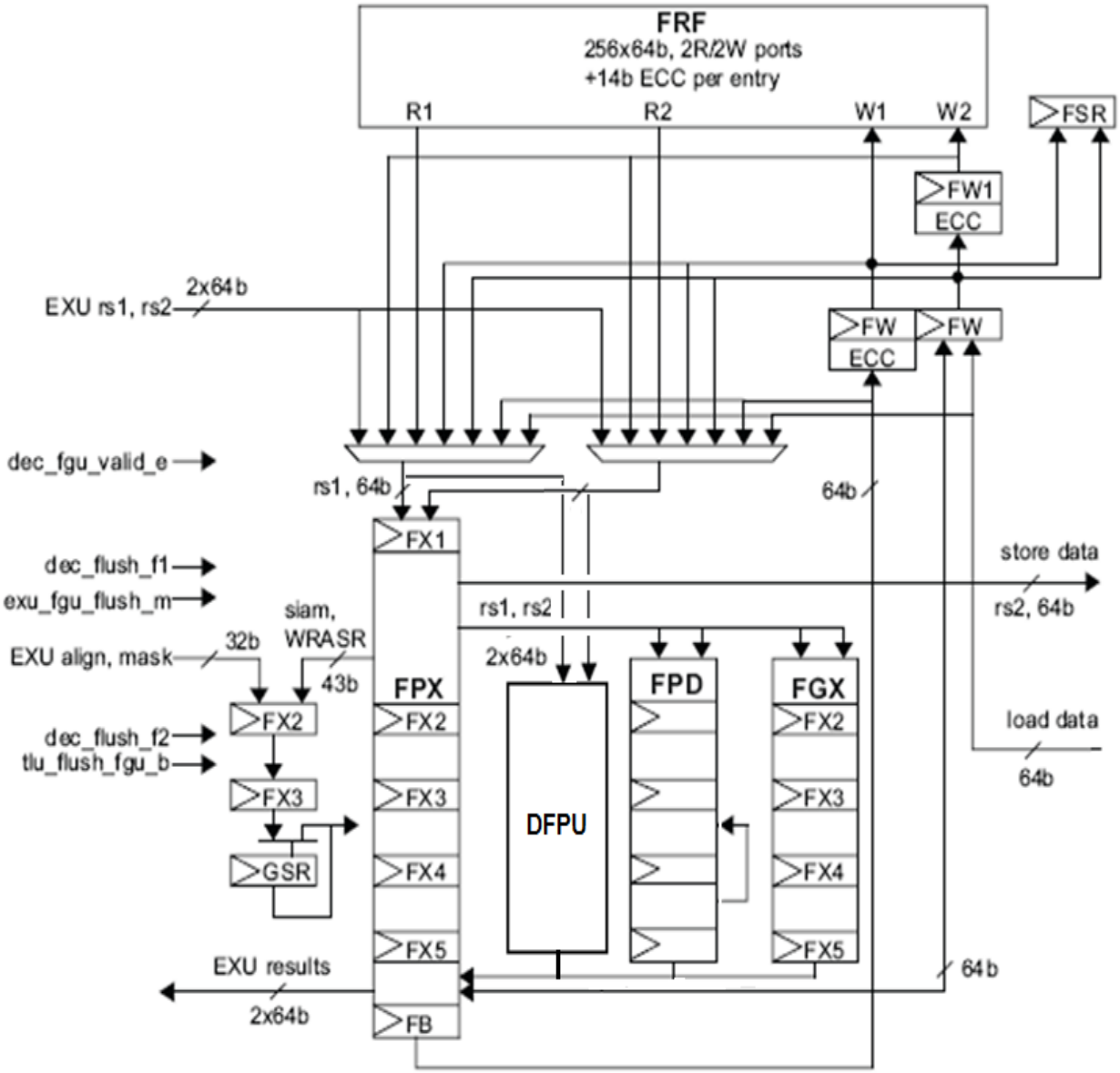


Figure 4.1: FGU including the DFPU pipeline

4.4 The Decimal Floating Point Unit (DFPU)

4.4.1 DFPU Architecture

The block diagram of the DFPU is shown in Figure 4.2. It is composed of three main blocks: the DFP FMA, the Decimal Floating-point Status Register (DFSR) and a set of buffers.

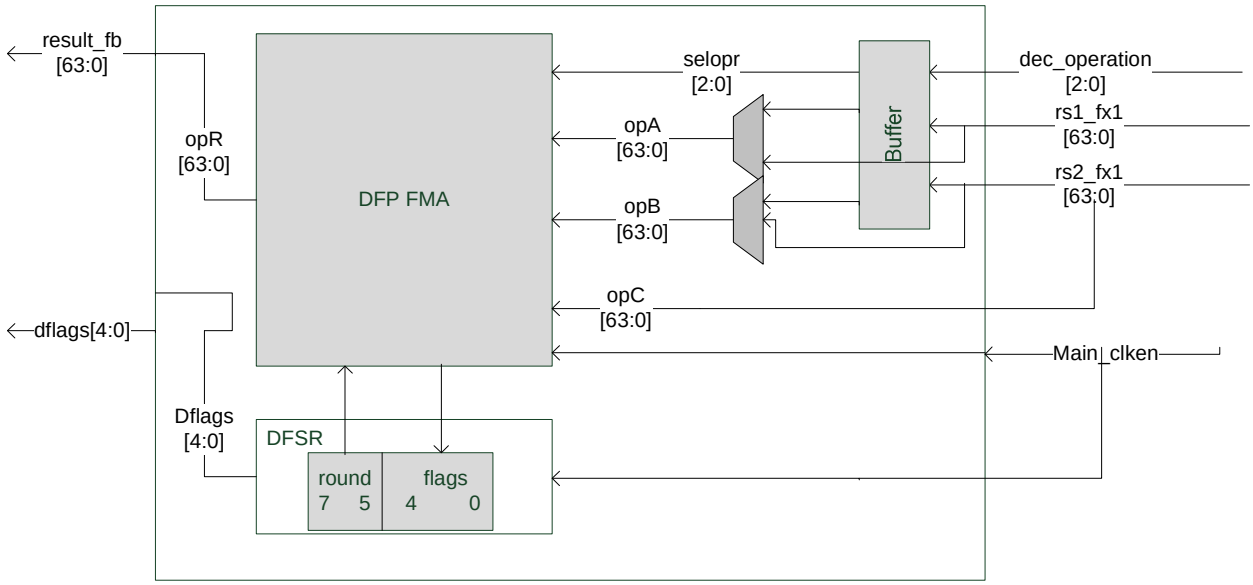


Figure 4.2: DFPU Architecture

4.4.1.1 The DFP FMA

The DFP FMA is the nuclei of the DFPU. The top level of the FMA architecture is shown in Figure 4.3. The architecture is composed of three main stages.

- The first stage is ***the multiplier tree*** which performs the multiplication operation. Also in parallel to the multiplier tree, the addend is prepared for addition. This eliminates the need for further processing on the multiplication result and hence reduces the critical path delay.
- The second stage is ***the leading zero anticipator*** which is important to align the operands for the third stage.
- The third stage is ***the combined add/round unit***.

The default alignment of the addend is such that the fractional point is to the right of the multiplication result. In parallel with the multiplier, the addend is aligned by shifting it to the right (case-1) or to the left (cases-2, 3, 4). From a 4p width, where p is the number of significand digits (16 digits in case of 64-bits format), it is required to anticipate the leading zeros in only 2p-digits. According to the exponent difference and the leading zeros of the addend, the appropriate 2p width is selected. This operation is illustrated in Figure 4.4

A leading zero anticipator anticipates the leading zeros in the addition/ subtraction. Based on the anticipated leading zero count and taking into consideration the preferred exponent, a final shift amount is determined for the two operands. The rounding position in the aligned operands is approximately known (with an error of one digit). Hence, the final result can be calculated using a combined add/round module instead of successive addition and rounding steps.

It is implemented as either a three-stage pipeline or a six-stage. For the three stage pipeline shown in Figure 4.5 (a), the design is pipelined in three stages. The first stage is the multiplier tree. The second stage contains the decimal carry save adder, the leading zero anticipator and the R/L Shifter. Finally, the combined add/round and the rounding set-up modules are in the third stage. For the six stage pipeline shown in Figure 4.5 (b), the partial product generation is placed at the first stage. Then, the carry save adder tree is placed at the second stage with the addend preparation. The remaining stages are presented in the figure.

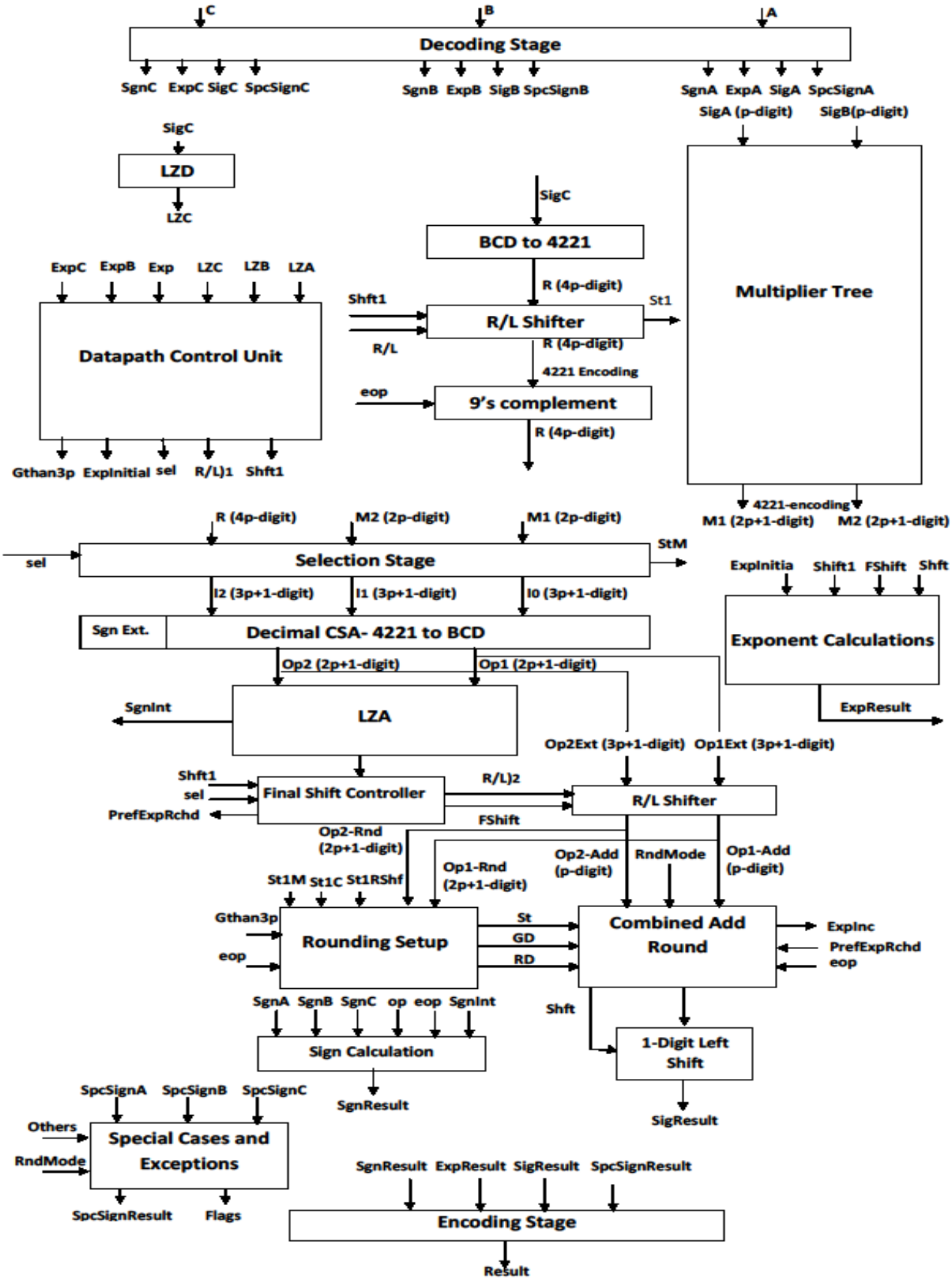


Figure 4.3: DFP FMA Architecture

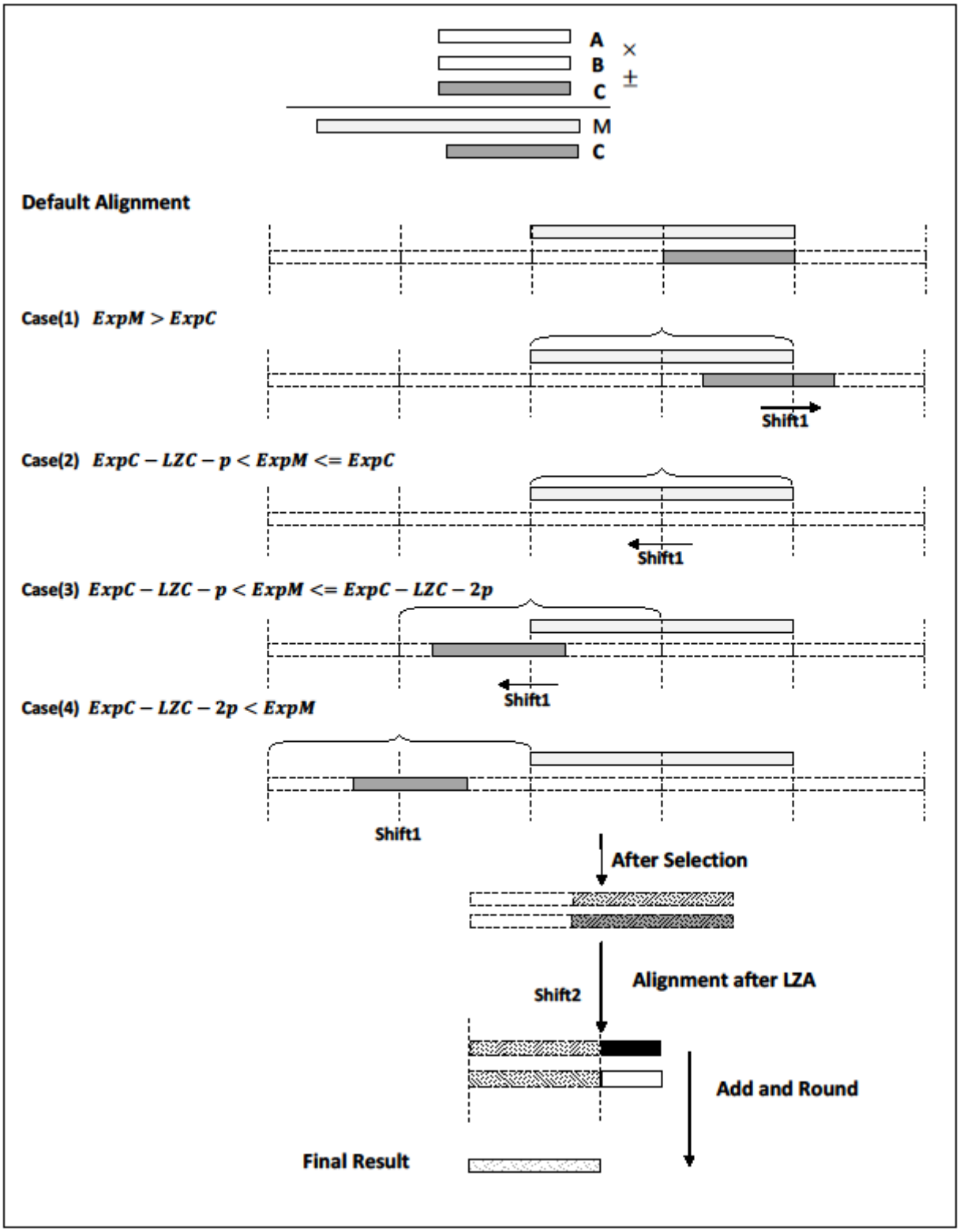


Figure 4.4: Operands alignment operation

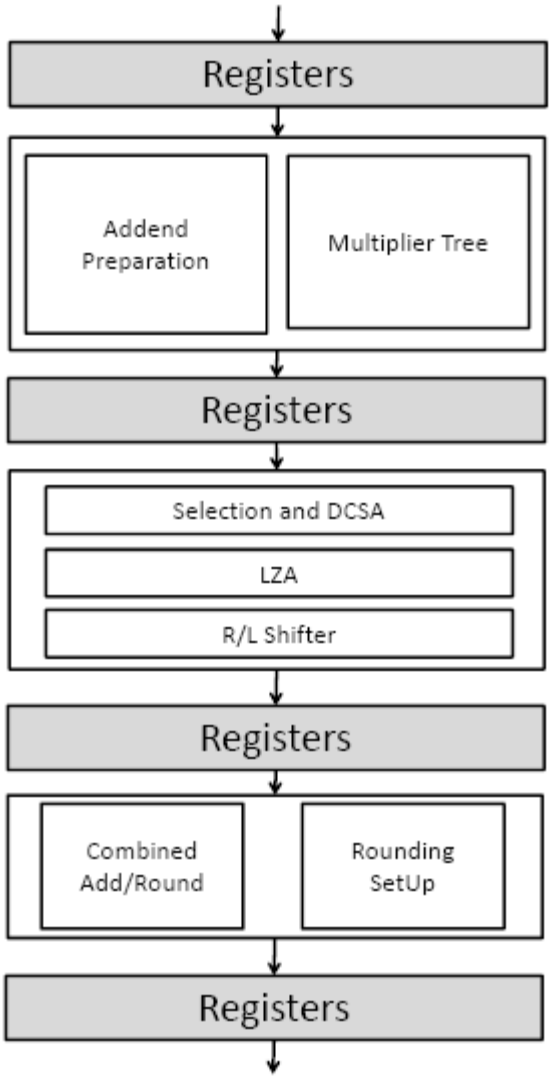
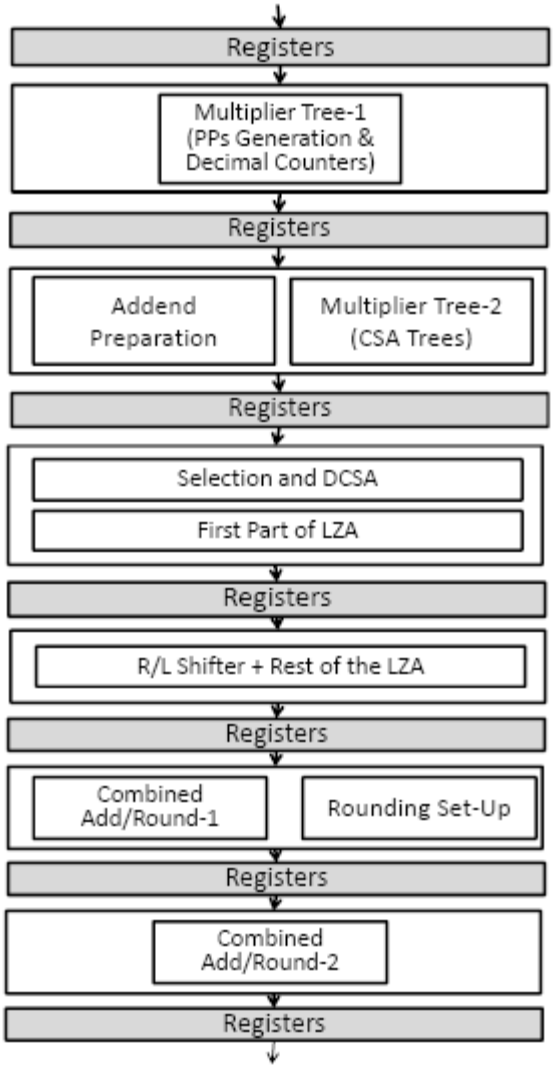


Figure 4.5: (a) Three-stages FMA pipeline



(b) Six-stages FMA pipeline

4.4.1.2 Decimal Floating Point Status Register (DFSR)

IEEE standard requires separate rounding directions for binary and decimal [18]. In addition, it states that the flags are shared between them. DFSR is inside the DFPU to simplify connections and design. DFSR.round contains the rounding direction required by the DFP FMA. Seven rounding modes are available. In addition to the five IEEE 754-2008 decimal rounding modes [18], we implement two additional rounding modes [25]: round to nearest down and round away from zero.

The flags are set by the result of the DFP FMA then they are sent to the FSR to update the current exception field (FSR.cexc) and the accrued exception field (FSR.aexc) as will be illustrated later in this chapter.

4.4.1.2.1 DFSR.flags

The IEEE-754 standard specifies the following about the decimal floating point related flags:

- ***Inexact Flag***
  The inexact flag is raised if the result is rounded. It is detected from the sticky, guard and round digits.

- ***Invalid Flag***
  The invalid flag is generated in either of these cases:
  - One of the operands is sNaN.
  - In case of $(0, \pm\infty, c)$ $FMA(\pm\infty, 0, c)$; where c is any DFP number including special numbers (NaNs, infinities). The standard in this case states that it is optional to raise the invalid flag if the third operand in qNaN. In our implementation we activate the invalid flag even if the third operand in qNaN.
  - In case of $FMA(|c|, +\infty, -\infty)$ or $FMA(|c|, -\infty, +\infty)$; where c is a DFP number that is not a NaN.

- ***Overflow Flag***
  The overflow is detected after rounding. It is signaled if the final exponent exceeds the maximum exponent in the standard. If an overflow is detected, the result is rounded either to infinity or to the largest possible number according to the rounding mode and the final sign.

- ***Underflow Flag***
  If the intermediate result is a non-zero floating point number with magnitude less than the magnitude of that format's smallest normal number (1×10^{-383}, in case of 64-bit format), an underflow is detected. However, the underflow flag is not raised unless the result is inexact.

As shown in Table 4.7 the field of bits [0:4] are the DFSR.flags.. Where *uf* is the underflow flag, *of* is the overflow flag, *nv* is the invalid flag, *nx* is the inexact flag and *dz* is the division by zero flag. As the DFP division is not implemented yet, the Division by Zero flag is always set to zero.

Dz	Nx	Nv	of	uf
4	3	2	1	0

Table 4.7: DFSR.flags

4.4.1.2.2 DFPR.round

The bits [5:7] implement the IEEE decimal rounding modes. Table 4.8 shows these different modes.

	Rounding mode
000	RNE = Round to Nearest ties to Even
001	RA = Round Away from zero
010	RP = Round toward Positive infinity
011	RM = Round toward Minus infinity
100	RZ = Round toward Zero
101	RNA = Round to Nearest ties Away from zero (round-half-up)
110	RNZ = Round to Nearest ties to Zero (round-half-down)

Table 4.8: Implemented rounding modes

4.4.1.3 The Buffers

FMA and FMS are three source instructions; hence, they require two cycles to read the source from the FRF which has only two read ports. No FGU executed instruction may be issued the cycle after FMA or FMS is issued. They are similar to the instruction Pixel Destination (PDIST) which is already implemented in the UltraSPARC T2 VIS.

The buffers store the two early sources and the decode operation type till the third source is ready. The addition, subtraction and multiplication operations do need only two sources; therefore, buffering their sources will waste a cycle. A group of multiplexers are used to select either buffered or non-buffered sources depending on the operation type.

4.4.2 The DFPU operation and Interfaces

The interface between DFPU and other OpenSPARC T2 units is shown in Figure 4.6. The FAD unit, which is part of the FGU, reads the sources and destination registers addresses, accessing the floating point register file (FRF), gets the sources data and send it to the DFPU. Simultaneously, the Floating Point Control unit (FAC) supplies the FPDU by the decimal operation type. So far, we implement the basic decimal instructions: addition, subtraction, multiplication, FMA and FMS. Global signals are the scan in and scan out signals used for testing and the memory built-in self test (MBIST) pins.

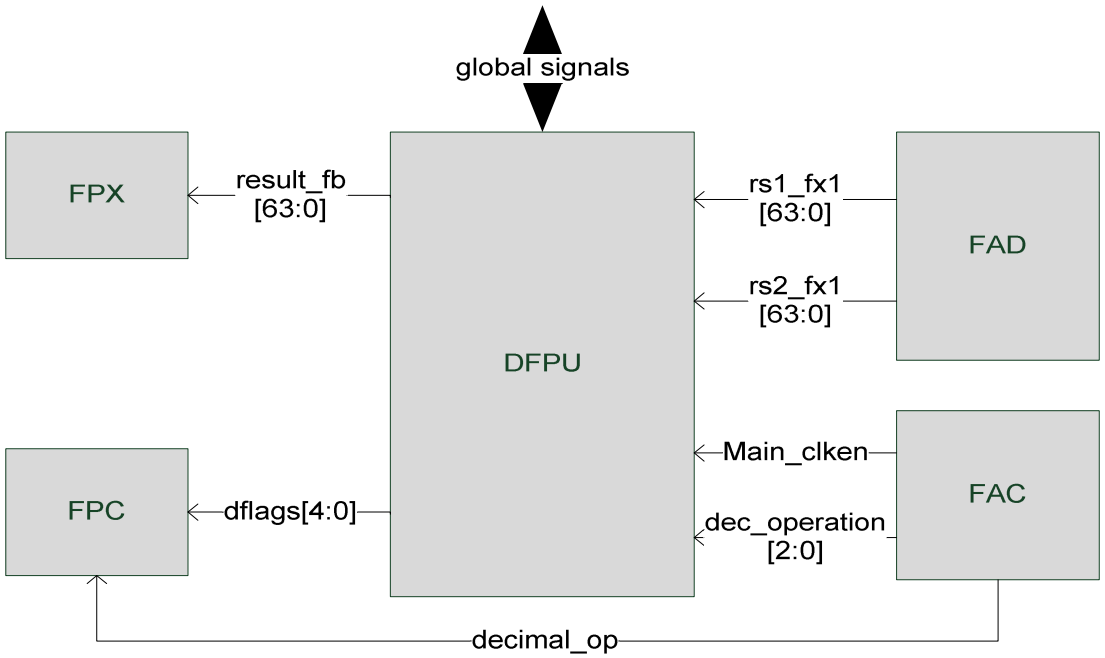


Figure 4.6: DFPU Interfaces

The DFP pipeline works in parallel with the floating point division (FPD), the floating point add/multiply (FPX) and the graphics (FGX) pipelines (as illustrated in Figure).

Depending on the operation type, the DFP FMA either waits another cycle to get the third source data (if the operation is FMA/FMS) or uses the ready data to perform the operation without waiting the second cycle (if the operation is ADD, SUB or MUL). The round type is given by the DFSR.rnd field.

Now, the core unit “DFP FMA” is ready to perform the required decimal operation. The three-stage FMA pipeline requires the three sources to be available at the first stage of the pipeline. Consequently, the buffering stage explained in section 4.3.1.3 is needed. On the other hand, the six-stage pipeline has the advantage of not needing the third source in the first stage. The addend is not processed at the first stage. Hence, it can read the addend after one cycle of reading the multiplier and the multiplicand without needing to wait for the addend in order to start the operation. In other words, the latency of reading the third operand is hidden by the first stage of the multiplier at the first cycle.

Finally when the result is ready, it is sent to the final stage (FB) in the main floating point pipeline (FPX), Figure 4.7. The output-format multiplexers choose between the results of the different pipelines. These multiplexers are modified to include the DFPU result with FPX, FPD and FGX results.

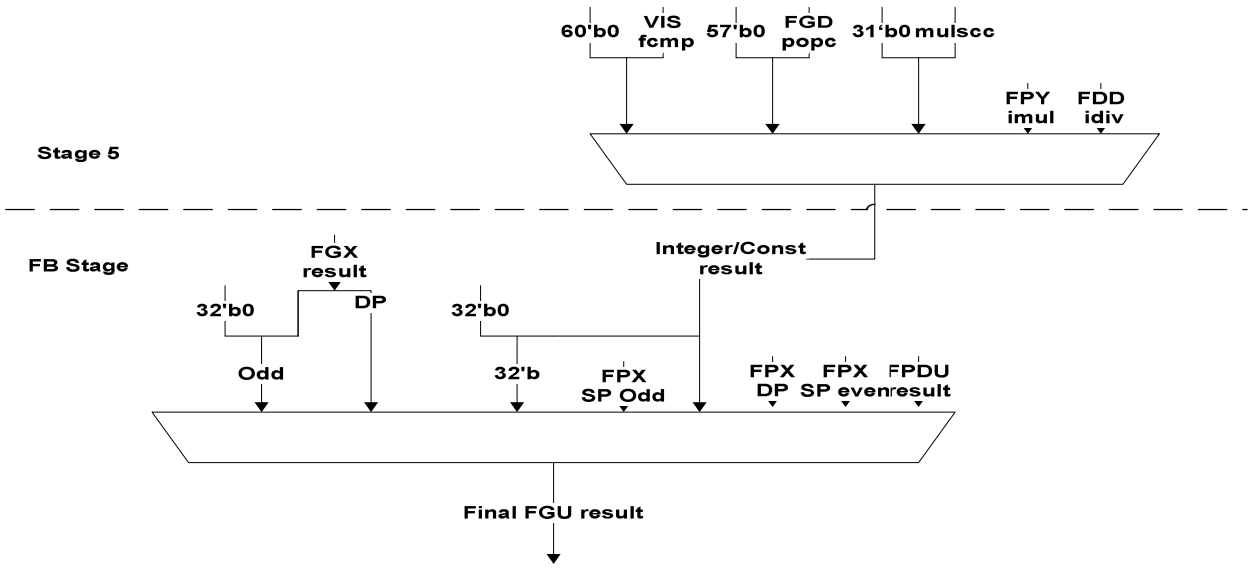


Figure 4.7: Output stage multiplexers

Although the pipelines share the write port (w1) of FRF, no arbitration is required between them. This is because of single instruction issue and fixed latency constraints which mean that we cannot have a DFP instruction and a BFP instruction access the write port at the same cycle.

FPDU also updates the flags of the DFSR which in turns are sent to the flag bits of the FSR register in the next cycle through FPC unit, back to Figure.

4.5 Architecture modifications in UltraSPARC T2 Core

4.5.1 FP Control Units Edits

4.5.1.1 FP Control Unit (FAC)

The Floating Point Control Unit (FAC) receives the opf and op3 fields of the instruction opcode (which are explained in section 4.2) from the decode unit. By decoding them, the operation is determined and the selection signals choose the corresponding operation code to be sent to the DFP unit. The selection logic is shown in Table 4.9 and the decoder design is shown in Figure 4.7.

Selection	Operation
0X0	FMA
0X1	FMS
10X	MUL
110	ADD
111	SUB

Table 4.9: The decimal operation selection bits

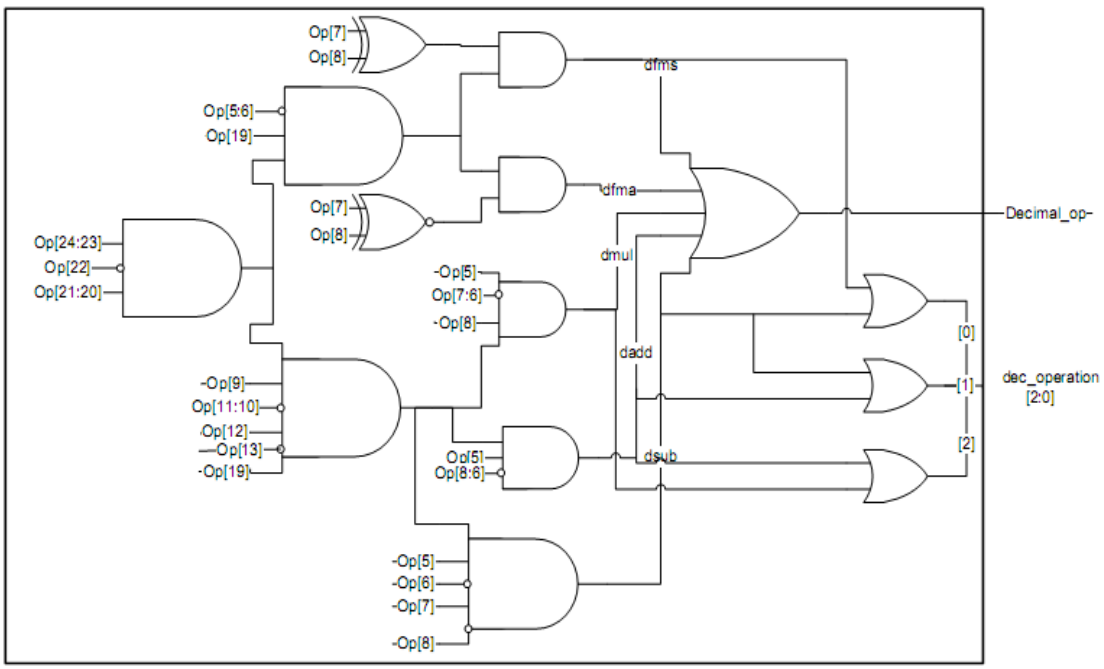


Figure 4.8: The Decimal opcode decoder

4.5.1.2 Floating point Control Unit (FPC)

FPC unit receives the DFSR.flags field to update the FSR corresponding fields if the operation contains a DFP instruction. FPDU causes an IEEE-754 exception if a flag is detected and its corresponding trap enable mask (FSR.TEM) is enabled. In this case the floating point trap type (ftt) is an IEEE exception and the (FSR.ftt) field is set to one. If an exception decimal flag raised and the corresponding trap enable mask (tem) bit is zero, it will prevent the corresponding IEEE 754 exception. Accrued Exceptions field (aexc) in this case will accumulate it till the tem bit is set to one. The decimal flags are ORed with the corresponding binary flags and the result is written into the FSR. All this logic is shown in Figure 4.8.

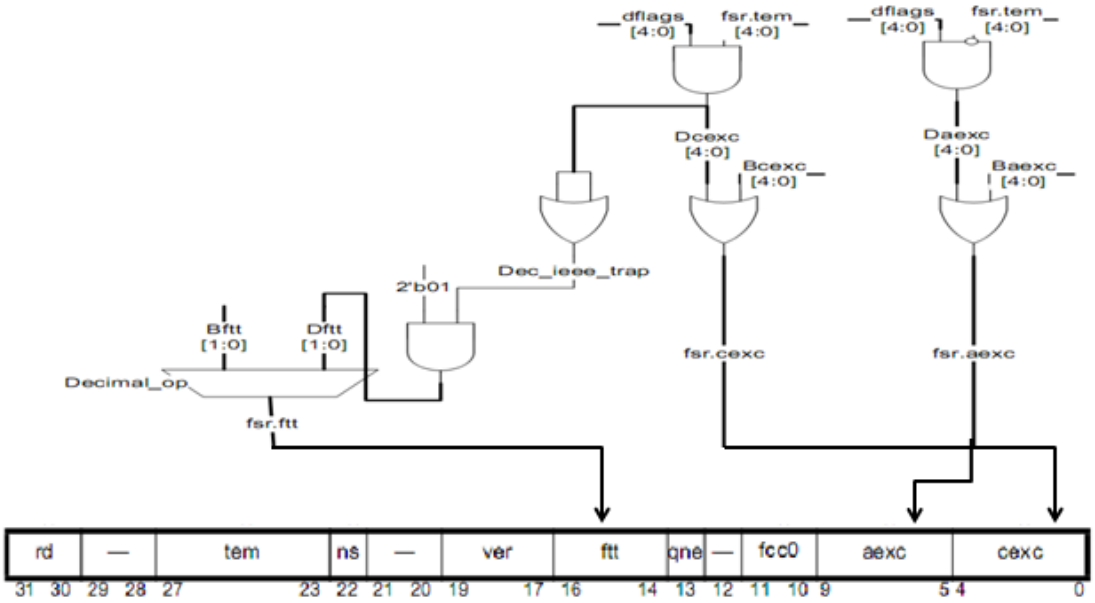


Figure 4.9: Updating FSR logic

4.5.2 Pick Unit

PKU pre-decodes the issued instructions and picks two of them, one from each thread group. Our design considers the DFP instructions as a part of the FGU instructions as illustrated in section 4.3. Decimal operation Detection has to be done here. In addition, all DFP opcodes have to be added to the FGU operations.

The instructions that need two cycles to fetch its data sources are also detected in the pick stage. These instructions include the Load Floating-Point Register from Alternate Space in memory *LDFA*, Store Floating-Point Register to Alternate Space *STFA*, Compare and Swap Word from Alternate Space in memory, *CASA*. They need two cycles to fetch the indirect-addressed source. The two-cycle instructions also include the Pixel Component Distance with Accumulation *PDIST* instruction which needs three sources instead of two and the FRF has only two read ports. We add the DFMA, DFMS instructions to these two-cycle ones as they have (like PDIST) three sources.

Moreover, PKU prevents any two-cycle instruction from going down into the pipeline if an integer load instruction exists in the decode stage. This is done because UltraSPARC T2 core does no dependency checking on the second cycle of a two-cycle instruction.

4.5.3 Decode Unit

DFPU contributes in the following hazards:

- FGU-FGU Hazard:
  A FGU favor bit decides which FGU decodes and which FGU stalls. DFP Instructions are simply FGU instructions; No additional action is needed here!

- DFMA/DFMS Block Hazard
  We define this new block hazard to prevent decoding any FGU instruction from either thread group the cycle after a DFMA decodes. This prevents a hardware hazard on the read ports of the FRF.

Also decode unit keeps the address of the DFMA/DFMS third source (rs3) if a DFMA/DFMS block is detected. It passes the rs3 address to the second read port of the FRF in the next cycle. Moreover, the decode unit decodes the instructions' opcodes and determines the sources and destination types. We add the DFP opcodes to the instruction set that uses floating point sources and destination. Namely, add them to the double precision FP sources and destinations as we implement the Decimal64 standard specifications.

4.5.4 Gasket

All communications with the L2 cache is through the crossbars (PCX, CPX) via the gasket. The gasket has a control logic that partially decodes the coming packet and determines if it has a valid instruction opcode. We define the new decimal floating point opcodes to be considered valid in order to enable the gasket to transfer them from memory to the core pipeline.

Chapter 5 SOFTWARE TOOL CHAIN

5.1 Introduction

The second part of the work is to provide the necessary SW tools to generate programs for the new architecture. The GNU Compiler Collection (GCC) is patched to include several decimal double-precision floating point instructions. GCC development is a part of the GNU Project, aiming to improve the compiler used in the GNU system including the GNU/Linux variant. The GCC development effort uses an open development environment and supports many other platforms in order to foster a world-class optimizing compiler [78]. The op-codes of these instructions are added to the standard SPARC Instruction Set Architecture (SPARC ISA v9) [79].

Section 5.2 provides a brief introduction to the GCC structure. As we work on an Intel machine and targeting a SPARC machine, the Cross Compiling operation is described and compared with different compiling types in section 5.3. Section 5.4 explains the installation of the tool chain (GCC, Binutils and other required libraries). Section 5.5 shows our new version of the GCC which includes the decimal FP instructions. Finally, section 5.6 shows the results of testing the new GCC version.

5.2 GCC Structure

Compilers in general have 3 parts, Front End, Intermediate Part and Back End. The front end is the interface with variety of programming languages (e.g. C++, Java, FORTRAN ...). Each of those languages has a separate front end to deal with the features of this specified language. At the end of the front end a generic representation of the code is generated to be easier to deal with it. The intermediate part function is to make some optimizations on the generic code. The back end transforms that generic code into the target assembly language [80]. Most of our work is in the back end as will be explained in section 5.6.

5.3 Cross Compiler

There are three system names that the build operation knows about: the machine you are building on (*build*), the machine that you are building for (*host*), and the machine that GCC will produce code for (*target*). When we configure GCC, we specify these with `--build=', `--host=', and `--target='. If build, host, and target are all the same, this is called a *native*. If build and host are the same but target is different, this is called a *cross*. If build, host, and target are all different this is called a *canadian*. If host and target are the same, but build is different, you are using a **cross-compiler** to build a native for a different system (This is our case). Some people call this a *host-x-host, crossed native*, or *cross-built native*. If build and target are the same, but host is different, you are using a cross compiler to build a cross compiler that produces code for the machine you're building on. This is rare, so there is no common way of describing it. There is a proposal to call this a *crossback* [81].

5.4 Building and Installing GCC Cross-Compiler

5.4.1 Installing Prerequisites

GCC requires that various tools and packages be available for use in the build procedure. We needed to install these tools:

- GNU Multiple Precision library (GMP) [82]. We used GMP-5.0.2.
- Multiple-Precision Floating-point computations with *correct Rounding library (MPFR) [6]*. *We used MPFR*-3.0.1.
- Multi-Precision C library (MPC) [83]. We used *MPC-0.8.2.*

5.4.2 Building and installing Binutils

These utilities have to be built and installed before the compiler can be built and installed. These are some utilities that are responsible for assembly, linking and other binary utilities used to help manipulate the object files produced by the compiler.

The first step in building and installing the binutils is to download its complete distribution which includes the source files, configuration files and some documentation. The latest release of the Gnu binutils can be downloaded from [84]. In this work we used the binutils version 2.21

Unpack the downloaded package which will be unpacked to a directory called binutils-2.21/. Make a new directory called binutils-2.21-build/ and go to that directory and run the configure script:

```
../binutils-2.21/configure --target=sparc-elf --prefix=/opt/UltraSparc/ --verbose --with-cpu=v9 --enable-decimal-float=yes | tee configure.out;
```

The **--target** means that we want the binutils to be configured with the target machine is the **sparc-elf**, the **--prefix** option means that when this binutils be installed to be installed in the directory **/opt/UltraSparc**, **--verbos** means not to suppress warning messages, **--with-cpu** means that we want to use the sparc **version 9** processor, and at last the **2>&1 | tee configure.out** means that I want in addition to the messages that appear on the screen to redirect the messages also to a file called configure.out.

This command checks various system and compiler functions and builds an appropriate Makefile. This script will print out quite a few status messages and ends with the message "creating Makefile". This indicates that the script has run successfully and produced a valid Makefile for this particular combination of host and target.

At this point, a Makefile has been created and it can be run. The binutils for the Sparc architecture can now be built with the command:

```
Make
```

This process can take several minutes and will produce numerous status messages. When the build is complete, the newly compiled executables can be installed. This must be done using the super user or root account, since the files will be installed in a shared directory of the file system where they can be accessed by other users. The commands to install the binutils are:

```
make install
```

This install command will copy some files into the /opt/UltraSPARC/bin/ directory, these executables are various utilities used by the SPARC GCC compiler to build and manipulate libraries and object files.

5.4.2.1 Building and Installing GCC

The process for downloading, building and installing the GNU GCC is a very similar process to the one used to build the Gnu binutils, except of course, that a different set of distribution files are used.

The first step in building and installing the GNU GCC Compiler is to download the complete distribution. This is a set of files including source code and various configuration and documentation files. This can be downloaded from [85], in this work we used the GCC version 4.6.0

Unpack the downloaded package which will be unpacked to a directory called gcc-4.6.0/. Make a new directory called gcc-4.6.0-build/ and go to that directory and run the configure script:

```
gcc-4.6.0/configure --target=sparc-elf --prefix=/opt/UltraSparc/ --with-gnu-as --with-gnu-ld --verbose --enable-languages=c,c++ --disable-shared --disable-nls --enable-decimal-float=yes --with-newlib --with-cpu=v9 2>&1 | tee configure.out
```

The **--with-gnu-as** and **--with-gnu-ld** means to use the GNU assembler and linker with the GCC. **--enable-decimal-float=yes** to enable the decimal floating point feature.

Like the configure command for the binutils, this configure command checks various system and compiler functions and builds an appropriate Makefile. This script will print out quite a few status messages and ends with the message "creating Makefile". This indicates that the script has run successfully and produced a valid Makefile for this particular combination of host and target.

At this point, a Makefile has been created and it can be run. The GCC for the Sparc architecture can now be built with the command:

```
make all-gcc
```

This process can take several minutes or even hours depending on the host machine and the configuration and will produce numerous status messages. When the build is complete, the newly compiled gcc executable can be installed. Again, as with the binutils, this must be done using the super user or root account, since the files will be installed in a shared directory of the file system where they can be accessed by other users. The command to install the GCC is:

```
make install-gcc
```

Note that this install command copies a single executable file, gcc, into the /opt/UltraSPARC/bin/ directory. The previously installed binutils have already been installed in this directory. As with the binutils, manual pages and other supplementary material may have been installed by this command.

5.4.2.2 Building and Installing Newlib

Newlib is a collection of C libraries that are important to the GCC compiler. The process of building and installing the newlib is very similar to that of the binutils and the GCC. The following are the instructions for building and installing it.

```
../newlib-1.19.0/configure --target=sparc-elf --prefix=/opt/UltraSparc/ --enable-decimal-float=yes --with-cpu=v9 2>&1 | tee configure.out;
```

```
make -j 3 2>&1 | tee compile.out;
```

```
make install 2>&1 | tee compile.out;
```

5.4.2.3 Rebuilding and Installing GCC with Newlib

Finally after installing the newlib, we have to rebuild the GCC compiler to include the new libraries added. the steps for building it is as follows.

```
cd gcc-build;\
/..gcc-4.6.0/configure --target=sparc-elf --prefix=/opt/UltraSparc/ --with-gnu-as --with-gnu-ld --verbose --enable-languages=c,c++ --enable-decimal-float=yes --disable-shared --disable-nls --with-cpu=sparc --with-newlib 2>&1 | tee configure_gcc.out
```

5.5 Editing the GCC Compiler Source Code

Our target is to modify the GCC Compiler to be compliant with our new decimal floating point ULTRASPARC T2 processor and generate decimal floating point instructions in the assembly-code when compiling a decimal segment in a C-code.

The main modifications are in the back end of GCC which contains the target files. The SPARC files on the GCC source code lies in /gcc/configure/sparc except the opcodes files which lie in the binutils source code. We modified seven target files. These files are:

- **The machine description file (sparc.md)**

The '.md' file for a target machine contains a pattern for each instruction that the target machine supports (or at least each instruction that is worth telling the compiler about).

- **The C header file (sparc.h)**
  The header file defines numerous macros that convey the information about the target machine that does not fit into the scheme of the '.md' file.

- **The C source file (sparc.c)**
  The source file defines a variable targetm, which is a structure containing pointers to functions and data relating to the target machine. 'machine.c' should also contain their definitions, if they are not defined elsewhere in GCC, and other functions called through the macros defined in the header file.

- **The option specification file (sparc.opt)**
  It is an optional file in the 'machine' directory, containing a list of target-specific options.

- **The opcodes file (/binutils/opcodes/sparc-opc.c).**
  It is the file containing the opcodes of the instructions defined in the machine description file.

- **/binutils/include/opcode/sparc.h**
  It is the file that contains the definitions for opcode table for the SPARC target.

- **/binutils/gas/config/tc-sparc.c**
  It is the file that includes the source code for the GCC assembler for the SPARC target machine.

Following are description for the edits we made in each file to add the decimal floating point capability for the GCC compiler targeting SPARC processor.

5.5.1 Edits in the machine description (sparc.md)

Three instruction patterns were added for the floating point decimal instructions (double add, double subtract and double multiply).

Each instruction pattern contains an incomplete RTL expression, with pieces to be filled in later, operand constraints that restrict how the pieces can be filled in, and an output pattern or C code to generate the assembler output, all wrapped up in a define_insn expression. A define_insn is an RTL expression containing four or five operands [4].

1. **An optional name**. The presence of a name indicates that this instruction pattern can perform a certain standard job for the RTL-generation pass of the compiler. This pass knows certain names and will use the instruction patterns with those names, if the names are defined in the machine description. The defined name for the DFP double-precision addition is "**adddd3"**, for the subtraction is "**subdd3"**, for the multiplication is "**muldd3"**, for the fused multiply-add is "**fmadd4**" and for the fused multiply-sub is "**fmsdd4**".
2. **The RTL template** is a vector of incomplete RTL expressions which show what the instruction should look like. It is incomplete because it may contain match_operand, match_operator, and match_dup expressions that stand for operands of the instruction.
3. **A condition**. This is a string which contains a C expression that is the final test to decide whether an instruction body matches this pattern. For a

named pattern, the condition may not depend on the data in the instruction being matched, but only the target-machine-type flags. The compiler needs to test these conditions during initialization in order to learn exactly which named instructions are available in a particular run. Our defined DFP instructions need to check two flags **"TARGET_FPU && TARGET_DFP".** And the FMA/FMS operations needs to check an additional flag **"TARGET_DFMA".**

4. **The output template**: a string that says how to output matching instruction as assembler code. '%' in this string specifies where to substitute the value of an operand. For example the output template of the DFADDd instruction is **"dfaddd\t%1, %2, %0"** where %1, %2 are the two sources and %0 is the destination.

5. Optionally, a vector containing the values of **attributes** for instructions matching this pattern. For the decimal instructions, the attributes defined indicate the type of the sources to be "**fp**" and "**double**" which mean a double floating point registers. For the FMA/FMS operations, the attribute defined is "**fpmul**" which means that the instruction includes a FP multiplication.

```
(define_insn "adddd3"
  [(set:DD (match_operand:DD 0 "register_operand" "=e")
        (plus:DD (match_operand:DD 1 "register_operand" "e")
                 (match_operand:DD 2 "register_operand" "e")))]
  "TARGET_FPU && TARGET_DFP"
  "dfaddd\t%1, %2, %0"
  [(set_attr "type" "fp")
   (set_attr "fptype" "double")])
```

```
(define_insn "subdd3"
  [(set (match_operand:DD 0 "register_operand" "=e")
        (minus:DD (match_operand:DD 1 "register_operand" "e")
                  (match_operand:DD 2 "register_operand" "e")))]
  "TARGET_FPU && TARGET_DFP"
  "dfsubd\t%1, %2, %0"
  [(set_attr "type" "fp")
   (set_attr "fptype" "double")])
```

```
(define_insn "muldd3"
  [(set (match_operand:DD 0 "register_operand" "=e")
        (mult:DD (match_operand:DD 1 "register_operand" "e")
                 (match_operand:DD 2 "register_operand" "e")))]
  "TARGET_FPU && TARGET_DFP"
  "dfmuld\t%1, %2, %0"
  [(set_attr "type" "fpmul")
   (set_attr "fptype" "double")])
```

```
(define_insn "fmadd4"
  [(set (match_operand:DD 0 "register_operand" "=e")
        (fma:DD (match_operand:DD 1 "register_operand" "e")
                (match_operand:DD 2 "register_operand" "e")
                (match_operand:DD 3 "register_operand" "e")))]
  "TARGET_DFMA"
 "dfmaddd\t%1, %2, %3, %0"
  [(set_attr "type" "fpmul")])
```

```
(define_insn "fmsdd4"
  [(set (match_operand:DD 0 "register_operand" "=e")
        (fma:DD (match_operand:DD 1 "register_operand" "e")
                (match_operand:DD 2 "register_operand" "e")
                (neg:DD (match_operand:DD 3 "register_operand" "e"))))]
  "TARGET_DFMA"
 "dfmsubd\t%1, %2, %3, %0"
  [(set_attr "type" "fpmul")])
```

5.5.2 Edits in the header file (sparc.h)

This file contains C macros that define general attributes of the machine. It defines the default options of the target processor as follows:

```
#define TARGET_DEFAULT (MASK_APP_REGS + MASK_FPU)
```

We added to this definition the hard decimal floating point option such that any C code that contains decimal operations will be compiled by default and without need for additional options:

```
#define TARGET_DEFAULT (MASK_APP_REGS + MASK_FPU +
MASK_DFP)
```

5.5.3 Edits in the options file (sparc.opt)

This file defines the compilation options that the SPARC target knows about. These options enable/disable its related masks.

We defined new options for either using our hardware DFP "**mhard-dfp**" option or using DFP in software level "**msoft-dfp**". The default option is "msoft-dfp" which calls the DecNumber library to execute decimal instructions in software layer using the already exists binary hardware.

```
;Decimal FP
mdfp
Target Report Mask(DFP)
Use hardware DFP
;-------------------------------------------------------------
mhard-dfp
Target RejectNegative Mask(DFP) MaskExists
Enable decimal floating point hardware support
;-------------------------------------------------------------
msoft-dfp
Target RejectNegative InverseMask(DFP)
Disable decimal floating point hardware support
;-------------------------------------------------------------
;-------------------------------------------------------------
;fma
mdfma
Target Report Mask(DFMA)
Generate FMA instructions
;-------------------------------------------------------------------
mhard-dfma
Target RejectNegative Mask(DFMA) MaskExists
Enable decimal floating point FMA hardware support
;-------------------------------------------------------------------
msoft-dfma
Target RejectNegative InverseMask(DFMA)
Disable decimal floating point FMA hardware support
```

Lines which are preceded by the semicolon are comments. For the "**mhard-dfp**" option: the first line is the option name (mhard-dfp). The third line is the related mask which is affected by the defined option. And the final line states the effect of stating this option in the compilation, in our option it will enable the DFP

hardware support. The same explanation applies for the "**msoft-dfp**", "**mhard-dfma**" and "**msoft-dfma**" options.

5.5.4 Edits in the C source file (sparc.c)

1. Define a variable that defines whether an FPU option was specified or not. We does not enable the DFP hardware option by default. Consequently, this variable is set initially to false. We set it to true when the **"mhard-dfp"** option is chosen. The same clarification applies to the dfma option.

```
static bool dfp_option_set = false;
static bool dfma_option_set = false;
```

2. We need to define a new class that represents the DFP double-precision. This is done in the enumeration **sparc_mode_class**.

```
enum sparc_mode_class {
  S_MODE, D_MODE, T_MODE, O_MODE,
  SF_MODE, DF_MODE, TF_MODE, OF_MODE,DD_MODE,
  CC_MODE, CCFP_MODE };
```

, add the DD mode to the definition of modes for double-word and smaller quantities.

```
#define D_MODES (S_MODES | (1 << (int) D_MODE) | (1 << DF_MODE)|
(1<< DD_MODE))
```

and also to the definition of modes for double-float only quantities.

```
#define DF_MODES_NO_S ((1 << (int) D_MODE) | (1 << (int) DF_MODE)
|(1<<(int) DD_MODE))
```

3. In the function sparc_init_modes which does various machine dependent initializations.

```
static void
sparc_init_modes (void)
{
  int i;

for (i = 0; i < NUM_MACHINE_MODES; i++)
   {
     switch (GET_MODE_CLASS (i)) {
/**************************Decimal************************
       case MODE_DECIMAL_FLOAT:
        if (GET_MODE_SIZE (i) == 8)
            sparc_mode_class[i] = 1 << (int) DD_MODE;
         /*  else if (GET_MODE_SIZE (i) == 16)
            sparc_mode_class[i] = 1 << (int) TD_MODE;
          else if (GET_MODE_SIZE (i) == 32)
            sparc_mode_class[i] = 1 << (int) OD_MODE;*/
          else
            sparc_mode_class[i] = 0;
          break;
```

4. The user in compilation can disable/enable the hardware binary floating point unit (FPU), the hardware decimal floating point (DFP) or the hardware decimal FMA (DFMA). So, we need to look up the case of disabling FPU and enabling DFP at the same time. An error message should be shown and the hard DFP is disabled. Furthermore, we need to look up the case where the DFMA is enabled while DFP is disabled. Also, an error message should be shown and the hard DFMA is disabled.

```
if (! TARGET_FPU)
   { if ((target_flags_explicit & MASK_HARD_DFP) &&
TARGET_HARD_DFP)
         error ("-mhard-dfp can%'t be used in conjunction with -msoft-
float");
          target_flags &= ~MASK_HARD_DFP; }
```

```
if (! TARGET_DFP)
   {
    if ((target_flags_explicit & MASK_DFMA) && TARGET_DFMA)
        error ("-mhard-fma can%'t be used in conjunction with -msoft-dfp");
    target_flags &= ~MASK_DFMA; }
```

5. Implement TARGET_HANDLE_OPTION

```
static bool
sparc_handle_option (size_t code, const char *arg, int value
ATTRIBUTE_UNUSED)
{
  switch (code)
    {
//Decimal-----------------------------------------------
    case OPT_mdfp:
    case OPT_mhard_dfp:
    case OPT_msoft_dfp:
      dfp_option_set = true;
      break;
//FMA---------------------------------------------------
    case OPT_mdfma:
    case OPT_mhard_dfma:
    case OPT_msoft_dfma:
      dfma_option_set = true;
      break;
//-----------------------------------------------------
}
 return true;}
```

6. If -mdfp or -mno-dfp (or -mdfma or –mno-dfma) was explicitly used, don't override with the processor default.

```
if (dfp_option_set)
   target_flags = (target_flags & ~MASK_DFP) | (target_flags &
MASK_DFP);
if (dfma_option_set)
   target_flags = (target_flags & ~MASK_DFMA) | (target_flags &
MASK_DFMA);
```

```
switch (mclass)
   {
   case MODE_FLOAT:
   case MODE_DECIMAL_FLOAT:
```

5.5.5 Edits in the opcodes file (sparc-opc.c)

The opcodes is defined in a template structure which has the following fields:

```
typedef struct sparc_opcode
{  const char *name;
  unsigned long match
  unsigned long lose;
  const char *args;
  /* This was called "delayed" in versions before the flags.  */
  char flags;
  short architecture;
} sparc_opcode;
```

Where ***name*** is the instruction name, ***match*** has the bits that must be set or by other words the match component is a mask saying which bits must match a particular opcode in order for an instruction to be an instance of that opcode, ***lose*** has the bits that must not be set, the ***args*** component is a string containing one character for each operand of the instruction, architecture is the bitmask of sparc opcode architecture values. We add the opcodes for five DFP instructions. Namely: DFADDd, DFSUBd, DFMULd, DFMADDd and DFMSUBd.

```
{"dfaddd",   F3F(2, 0x36, 0x092), F3F(~2, ~0x36, ~0x092), "v,B,H",
F_FLOAT,v6}
/*************************************************************/
{"dfsubd",   F3F(2, 0x36, 0x096), F3F(~2, ~0x36, ~0x096), "v,B,H",
F_FLOAT,v6}
/*************************************************************/
{ "dfmuld",  F3F(2, 0x36, 0x09a), F3F(~2, ~0x36, ~0x09a), "v,B,H",
F_FLOAT,v6}
/*************************************************************/
{ "dfmaddd",F4F(2, 0x37, 0x3), F4F(~2, ~0x37, ~0x3), "v,B,4,H", F_FLOAT,
v6},/*Decimal FMA */
/*************************************************************/
{"dfmsubd",F4F(2, 0x37, 0x7), F4F(~2, ~0x37, ~0x7), "v,B,4,H", F_FLOAT,
v6},/*Decimal FMS */
```

```
#define F3F(x, y, z)(OP (x) | OP3 (y) | OPF (z))
#define F4F(x, y, z)(OP (x) | OP3 (y) | OP5 (z)) /* Format3 float insns.  */
```

OP contains bits [31,30] of the opcode and it is “10” (or “0x2” as written in the F3F) for arithmetic instructions. OP3 is the field of bits [24:19] of the opcode and it is “0x36” for DFADDd, DFSUBd and DFMULd and “0x37” for DFMADDd and DFMSUBd. OPF has the bits [13:5] of IMPDEP1 instructions (back to Chapter 4 section 4.2 for details). It is “0x092” for DFADDd, “0x096” for DFMULd and “0x09A” for DFMULd. OP5 is the field of bits [8:5] of FMA/FMS. It is “0x3” for DFMADDd and “0x7” for DFMSUBd. We have added the definition of this field in the file **binutils/include/opcode/sparc.h.**

```
#define OP3(x)          (((x) & 0x3f) << 19)
#define OP5(x)          (((x) & 0xf) << 5)
#define OP(x)           ((unsigned) ((x) & 0x3) << 30)
#define OPF(x)          (((x) & 0x1ff) << 5)
```

The meaning of the operands’ arguments used is shown in Table 5.1.

Argument	Meaning
V	frs1 is a floating point register (double/even).
B	frs2 is a floating point register (double/even).
H	frsd is a floating point register (double/even).
4	frs3 is a floating point register (double/even).

Table 5.1: Operands' arguments

We edit the file **/binutils/gas/config/tc-sparc.c** to define the check on the added argument “4” for the third source of FMA/FMS operations.

```
switch (*args)
                {……………………
                //added check
                case '4':
                opcode |= RS3 (mask);
                continue;
                  …….
               }
```

5.6 Testing the modified Cross Compiler

A test code is presented here to demonstrate the final result of the compiler. We had written a C code that defines variables of the GCC "_Decimal64" built-in type and performs three different operations on them: addition, subtraction and multiplication.

```
#include <stdio.h>
  main ()
  {
 _Decimal64 calculateTotal1,calculateTotal2,calculateTotal3;
 _Decimal64 price1,price2,price3;
 _Decimal64 taxRate1,taxRate2,taxRate3;

price1=50.5dd;
price2=150.5dd;
price3=250.5dd;
taxRate1=0.45dd;
taxRate1=0.55dd;
taxRate1=0.65dd;

 calculateTotal1= price1 + taxRate1;
 calculateTotal2= price2 - taxRate2;
 calculateTotal3= price3 * taxRate3;
  }
```

We then compiled the above C code without the new capability of generating hardware DFP instructions. The compiler calls the decimal software functions: **__dpd_adddd3** for addition, **__dpd_subdd3** for subtraction and **__dpd_muldd3** for multiplication as illustrated in the following SPARC assembly file.

```
        .file   "decimal_test.c"
        .global __dpd_adddd3
        .global __dpd_subdd3
        .global __dpd_muldd3
        .section        ".rodata"
        .align 8
.LLC0:
        .long   573833216
        .long   645
        .align 8
.LLC1:
        .long   573833216
        .long   1669
        .align 8
.LLC2:
        .long   573833216
        .long   2693
        .align 8
.LLC3:
        .long   573571072
        .long   69
        .align 8
.LLC4:
        .long   573571072
        .long   85
        .align 8
.LLC5:
        .long   573571072
        .long   101
        .section        ".text"
        .align 4
        .global main
        .type   main, #function
        .proc   04
main:
        save    %sp, -256, %sp
        sethi   %hi(.LLC0), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC0), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2039]
        sethi   %hi(.LLC1), %g1
```

```
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC1), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2031]
        sethi   %hi(.LLC2), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC2), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2023]
        sethi   %hi(.LLC3), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC3), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2015]
        sethi   %hi(.LLC4), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC4), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2015]
        sethi   %hi(.LLC5), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC5), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2015]
        ldd     [%fp+2039], %f0
        ldd     [%fp+2015], %f2
        call    __dpd_adddd3, 0
         nop
        mov     %o0, %g1
        stx     %g1, [%fp+2007]
        ldd     [%fp+2031], %f0
        ldd     [%fp+1999], %f2
        call    __dpd_subdd3, 0
         nop
        mov     %o0, %g1
        stx     %g1, [%fp+1991]
        ldd     [%fp+2023], %f0
        ldd     [%fp+1983], %f2
        call    __dpd_muldd3, 0
         nop
        mov     %o0, %g1
        stx     %g1, [%fp+1975]
        mov     %g1, %i0
        return  %i7+8
```

```
        nop
        .size   main, .-main
        .ident  "GCC: (GNU) 4.6.0"
```

Next, we compiled the same program using the new added option ***"-mhard-dfp***". In this case, the GCC chain replaced the software routines with the hardware instructions: DFADDd for addition, DFSUBd for subtraction and DFMULd for multiplication.

```
        .file   "decimal_test.c"
        .section        ".rodata"
        .align 8
.LLC0:
        .long   573833216
        .long   645
        .align 8
.LLC1:
        .long   573833216
        .long   1669
        .align 8
.LLC2:
        .long   573833216
        .long   2693
        .align 8
.LLC3:
        .long   573571072
        .long   69
        .align 8
.LLC4:
        .long   573571072
        .long   85
        .align 8
.LLC5:
        .long   573571072
        .long   101
        .section        ".text"
        .align 4
        .global main
        .type   main, #function
        .proc   04
```

```
main:
        save    %sp, -256, %sp
        sethi   %hi(.LLC0), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC0), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2039]
        sethi   %hi(.LLC1), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC1), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2031]
        sethi   %hi(.LLC2), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC2), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2023]
        sethi   %hi(.LLC3), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC3), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2015]
        sethi   %hi(.LLC4), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC4), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2015]
        sethi   %hi(.LLC5), %g1
        add     %g1, %g4, %g1
        add     %g1, %lo(.LLC5), %g1
        ldx     [%g1], %g1
        stx     %g1, [%fp+2015]
        ldd     [%fp+2039], %f10
        ldd     [%fp+2015], %f8
        dfaddd          %f10, %f8, %f8
        std     %f8, [%fp+2007]
        ldd     [%fp+2031], %f10
        ldd     [%fp+1999], %f8
        dfsubd          %f10, %f8, %f8
        std     %f8, [%fp+1991]
        ldd     [%fp+2023], %f10
        ldd     [%fp+1983], %f8
```

```
dfmuld      %f10, %f8, %f8
std     %f8, [%fp+1975]
mov     %g1, %i0
return  %i7+8
 nop
.size   main, .-main
.ident  "GCC: (GNU) 4.6.0"
```

5.7 Conclusion

We now have a complete GCC tool chain that can generate hardware decimal floating point Add, Sub, Multiply instructions. Although we have added the DFP Fused Multiply-Add and Fused Multiply-Sub instruction to the SPARC v9 ISA as stated in Chapter 4, we could not manage to add this support for the tool chain. We have made the required changes in the GCC source code. The problem is that the C standard till now does not support the FMA operation for the decimal floating point types [81]. It may be continued as a future work when the standard supports such operation.

Chapter 6 RESULTS AND FUTURE WORK

6.1 Introduction

OpenSPARC T2 comes with an automated verification environment. We adapted this environment to test our new core with the decimal floating point hardware capability. Section 6.2 introduces how to prepare the verification environment. Section 6.3 explains the simulation procedure. Section 6.4 shows the performance results. Section 6.5 lists the suggested future work. Finally, we conclude the thesis in section 6.6.

6.2 Preparing the Environment

The script used for the preparation is shown below. It sets some environment variables used during the simulation, the included path for executable files, the included directories for libraries and the license path.

The environment variables' meaning is shown in Table 6.1 and the used softwares and simulators are shown in Table 6.2.

Variable	Meaning
DV_ROOT	The top directory of the OpenSPARC T2 environment
MODEL_DIR	Directory where we run the simulations
VERA_HOME	Directory where Vera is installed
NOVAS_HOME	Directory where Debussy is installed
VCS_HOME	Directory where VCS Simulator is installed
NCV_HOME	Directory where NCV Simulator is installed
SYN_HOME	Directory where SYNOPSYS Simulator is installed
CC_BIN	Directory where C++ Compiler binaries are installed
LM_LICENSE_FILE	EDA tool license files

Table 6.1: Environemnt Variables

Software	Usage
VCS simulator	simulating the verilog processor files
Design Compiler	Synthesis tool
Vera	Testbench drivers, monitors, and coverage objects
Perl	Scripts for running simulations and regressions
GCC	Running C/C++ files and the SPARC assembly tests

Table 6.2: Used Programs and Simulators

```
                ################################################
                ###     Setting up the OpenSparc_T2 environment          ###
                ################################################
# For LINUX only
# User needs to define these new variables
export PROJECT PROJECT=OpenSparc_N2;
export DV_ROOT DV_ROOT=/home/Original/OpenSPARCT2;
export MODEL_DIR MODEL_DIR=/home/Original/OpenSPARCT2/OpenSparc_Simulation;
export TEMPDIR TEMPDIR=/var/tmp/cache.$USER/Cache;
export TRE_ENTRY TRE_ENTRY=/;
export TRE_SEARCH TRE_SEARCH=$DV_ROOT/tools/env/$PROJECT.iver;
# User needs to define following paths depending on the environment
#-----------------------------------------------------------------------------------------------------------
# Please define VCS_HOME if using vcs
export VCS_HOME VCS_HOME=/home/Setup/Synopsys/VCS/linux;
#-----------------------------------------------------------------------------------------------------------
# Please define VERA_HOME if using vera
export VERA_HOME VERA_HOME=/home/Setup/Synopsys/A-2007.12/vera_vA-
2007.12_linux;
#-----------------------------------------------------------------------------------------------------------
# Please define NCV_HOME if using ncverilog
  export NCV_HOME NCV_HOME=/home/Setup/Cadence/IUS;
  export CDS_INST_DIR CDS_INST_DIR=$NCV_HOME;
  export INSTALL_DIR INSTALL_DIR=$NCV_HOME;
  export ARCH ARCH=lnx86;
#-----------------------------------------------------------------------------------------------------------
-
 #Please define NOVAS_HOME only if you have debussy
 export NOVAS_HOME NOVAS_HOME=/home/Sources/Novas/Debussy/Debussy-52v15-
basic.tar.gz_FILES;
#-----------------------------------------------------------------------------------------------------------
-
# Please specify C/C++ compilers
  export CC_HOME CC_HOME=/usr;
  export CC_BIN CC_BIN=$CC_HOME/bin;
#-----------------------------------------------------------------------------------------------------------
# Please define SYN_HOME if you are running synopsys design compiler
 export SYN_HOME SYN_HOME=/home/Setup/Synopsys/DC_2009.06;
# Synopsys variables from $SYN_HOME
export SYN_LIB SYN_LIB=$SYN_HOME/libraries/syn;
export SYN_BIN SYN_BIN=$SYN_HOME/bin;
```

```
#-------Licence File-----------------------------------------------------------------------------------------
#export LM_LICENSE_FILE LM_LICENSE_FILE=
/home/Setup/Synopsys/10.9.3/admin/license/synopsys.dat:/home/Setup/Cadence/license/licens
e.dat
#:$LM_LICENSE_FILE;
#-----------------------------------------------------------------------------------------------------------
# Set Perl related variables
  export PERL_MODULE_BASE PERL_MODULE_BASE=$DV_ROOT/tools/perlmod;
  export PERL_PATH PERL_PATH=/usr;
  export PERL5_PATH PERL5_PATH=$PERL_PATH/lib;
  export PERL_CMD PERL_CMD=$PERL_PATH/bin/perl;
#-----------------------------------------------------------------------------------------------------------
# Set path for binaries and shared objects here...
unset $path
export PATH
PATH=$DV_ROOT/tools/Linux/x86_64:$DV_ROOT/tools/bin:$PERL_PATH/bin:/home/Set
up/Cadence/IUS/tools.lnx86/bin:$ModelSim_HOME/bin:$CC_BIN:$SYN_BIN:$VCS_HOM
E/bin:$VERA_HOME/bin:$PATH;
#-----------------------------------------------------------------------------------------------------------
unset LD_LIBRARY_PATH
export LD_LIBRARY_PATH
LD_LIBRARY_PATH=/lib:/usr/lib:/lib64:/usr/lib64:home/FIN64/lib/gcc/sparc64-
elf/4.6.0/include;
#-----------------------------------------------------------------------------------------------------------
# specifically for NC-Verilog
export LD_LIBRARY_PATH
LD_LIBRARY_PATH=$VERA_HOME/lib:$NCV_HOME/tools.lnx86/lib:$NCV_HOME/to
ols.lnx86/verilog/lib:$NCV_HOME/tools.lnx86/inca/lib:${LD_LIBRARY_PATH};
#-----------------------------------------------------------------------------------------------------------
export LD_LIBRARY_PATH
LD_LIBRARY_PATH=$NOVAS_HOME/share/PLI/nc`configsrch debussy_ncv
/`/LINUX/nc_loadpli1:$DV_ROOT/verif/env/common/pli/monitor/loadpli/linux:$DV_ROOT/
verif/env/common/pli/global_chkr/loadpli/linux:$DV_ROOT/verif/env/common/pli/socket/loa
dpli/linux:$DV_ROOT/verif/env/common/pli/bwutility/loadpli/linux:$DV_ROOT/verif/env/c
ommon/pli/cache/loadpli/linux:$DV_ROOT/verif/model/infineon/loadpli/linux:${LD_LIBRA
RY_PATH};
#-----------------------------------------------------------------------------------------------------------
alias lic='$SYNOPSYS/10.9.3/linux/bin/lmgrd -c
$SYNOPSYS/10.9.3/admin/license/synopsys.dat'
```

6.3 Simulation Procedure

After invoking the OpenSPARCT2.bashrc.linux script, the environment is ready for running simulations. The OpenSPARC T2 Design/Verification package comes with four test bench environments: cmp1, cmp8, fc1 and fc8. The cmp1 environment consists of: one SPARC CPU core, cache, memory and crossbar. The cmp1 environment does not have an I/O subsystem. The cmp8 environment consists of: eight SPARC CPU cores, cache, memory and crossbar. The cmp8 environment does not have an I/O subsystem. The fc1 environment consists of: a full OpenSPARC T2 chip with one SPARC Core, cache, memory, crossbar and I/O subsystem. The fc8 environment consists of: a full OpenSPARC T2 chip including all eight cores, cache, memory, crossbar and I/O subsystem. Each environment can perform either a mini-regression or a full regression.

To run the simulation, we use the following command:

```
sims -sys=cmp1 -group=cmp1_mini_T2 -diaglist=
/home/Original/my_diaglist.diag
```

-sys is a pointer to a specific test bench configuration to be built and run. It selects one of the four test bench environments: cmp1, cmp8, fc1 and fc8. ***–group*** name identifies a set of diagnostics to run in a regression. We have two groups for each test bench environment. The choices for -group are: cmp1_mini_T2, cmp1_all_T2, cmp8_mini_T2, cmp8_all_T2, fc1_mini_T2, and fc1_all_T2, fc8_mini_T2, and fc8_all_T2**. *–diaglist*** is the full path to diaglist file which identifies the assembly test files for the used group. We defined a new diaglist to include our assembly test files. This diaglist is shown below.

```
#ifndef SYSNAME
#define SYSNAME cmp1
#define sys(x) cmp1_ ## x
#define CMP
#define CMP1
#define ALL_THREADS 8
#endif

<sys(mini_T2) sys=cmp1>
<runargs -sys=cmp1 -tg_seed=1 >
<runargs -sas -vcs_run_args=+show_delta>

/*-----------------------Subtraction testing file ----------------------------------------*/
<dec_sub name=dec_sub.s>
  dec_sub             dec_sub.s
</dec_sub>
/*-----------------------Addition testing file -------------------------------------------*/
<dec_add name=dec_add.s>
  dec_add             dec_add.s
</dec_add>
/*-----------------------Multiplication testing file -------------------------------------*/
<dec_mul name=dec_mul.s>
  dec_mul             dec_mul.s
</dec_mul>
/*---------------------------------------------------------------------------------------*/
</runargs>
</runargs>
</sys(mini_T2)>
```

When running a simulation, the sims command performs the following steps:

1. Compiles the design into the $MODEL_DIR/cmp1 or $MODEL_DIR/fc8 directory, depending on which environment is being used.

2. Creates a directory for regression called $PWD/DATE_ID, where $PWD is your current directory, DATE is in YYYY_MM_DD format, and ID is a serial number starting with 0. For example, for the first regression on August07, 2007, a directory called $PWD/2007_08_07_0 is created. For the second regression run on the same day, the last ID is incremented to become $PWD/2007_08_07_1.

3. Creates a master_diaglist.regression_group file under the above directory. such as master_diaglist.cmp1_mini_T2 for the cmp1_mini_T2 regression group. This file is created based on diaglists under the $DV_ROOT/verif/diag directory.

4. Creates a subdirectory with the test name under the regression directory created in step 2 above.

5. Creates a sim_command file for the test based on the parameters in the diaglist file for the group.

6. Executes sim_command to run a Verilog simulation for the test. If the -sas option is specified for the test, it also runs the SPARC Architecture Simulator (SAS) in parallel with the Verilog simulator. The results of the Verilog simulation are compared with the SAS results after each instruction. The sim_command command creates many files in the test directory. Following are the sample files in the test directory:

diag.ev	diag.s	raw_coverage	seeds.log
status.log	vcs.log.gz	diag.exe.gz	midas.log
sas.log.gz	sims.log	symbol.tbl	vcs.perf.log

The status.log file has a summary of the status, where the first line contains the name of the test and its status (PASS/FAIL). An example is the status.log file for the subtraction instruction test.

```
Rundir: dec_sub:dec_sub.s:cmp1_mini_T2:0          PASS
```

7. Repeats steps 4 to 6 for each test in the regression group.

6.4 Performance Results

To verify the new architecture, three SPARC assembly files were written. One is for the addition, the second is for the subtraction and the third is for the multiplication. Each test file has only one operation on two test vectors from [87]. This is sufficient for measuring the performance and verifying the functionality of the design for two reasons. First, the instruction cycles do not depend on the operands' values. Second, the core unit has been verified using all the test vectors provided by [87] in [88].

```
#include "defines.h"
#include "nmacros.h"
#include "old_boot.s"

.text
.global main

main:           /* test begin */

        !# Initialize registers .
        !# Float Registers
        INIT_TH_FP_REG(%l7, %f2, 0x40BB9167BEA0918C) !Add
        INIT_TH_FP_REG(%l7, %f8,0xC0BA7B65181FA5F8) !Add

        !# Execute some ALU ops ..

        !dfaddd         %f2,    %f8,    %f2
EXIT_GOOD       /* test finish  */
```

```
#include "defines.h"
#include "nmacros.h"
#include "old_boot.s"

.text
.global main

main:           /* test begin */

        !# Initialize registers ..
        !# Float Registers
        INIT_TH_FP_REG(%l7, %f2, 0x0EF7E5EFEC54621F) !sub
         !setx 0x0EF7E5EFEC54621F, %g4, %g5
        !stx %g5, [%l7]
        !ldd    [%l7], %f2
         INIT_TH_FP_REG(%l7, %f8, 0x0EF4D215E2320010) !Sub

        !# Execute The DFP SUB operation
         dfsubd         %f2,    %f8,    %f2
EXIT_GOOD       /* test finish  */
```

```
#include "defines.h"
#include "nmacros.h"
#include "old_boot.s"

.text
.global main

main:           /* test begin */

        !# Initialize registers ..
        !# Float Registers
        INIT_TH_FP_REG(%l7, %f2, 0x00680000318936C4) !Mul
         INIT_TH_FP_REG(%l7, %f8,0x21D40000000F53A9) !Mul

        !# Execute some ALU ops ..
        dfmuld          %f2,    %f8,    %f2
EXIT_GOOD       /* test finish  */
```

The simulation environment boots up the processor and generates a memory image for the program's instructions opcodes. The memory image files shown below verify that the environment has successfully assembled the new DFP instructions using our modified GCC assembler. The opcodes shown in bold are for the DFP instructions: 85b09248 for DFADDd, 85b092c8 for DFSUBd, 85b09348 for DFMULd.

```
@0000000020000000       // Section '.MAIN', segment 'text'

ca75c000c51dc000 09102ee40b2fa824 881121678a11618c 892930208a114004

ca75c000c51dc000 ca75c000cd1dc000 ca75c000d11dc000 09302e9e0b0607e9

881123658a1161f8 892930208a114004 ca75c000d11dc000 ca75c000d91dc000

85b09248033fffff    053c0000821063ff 8410a0e083287020 8410800181c08000

0100000000000000
```

```
@0000000020000000        // Section '.MAIN', segment 'text'
ca75c000c51dc000 09102ee40b2fa824 881121678a11618c 892930208a114004
ca75c000c51dc000 ca75c000cd1dc000 ca75c000d11dc000 09302e9e0b0607e9
881123658a1161f8 892930208a114004 ca75c000d11dc000 ca75c000d91dc000
85b092c8033fffff 053c0000821063ff 8410a0e083287020 8410800181c08000
0100000000000000
```

```
@0000000020000000        // Section '.MAIN', segment 'text'
ca75c000c51dc000 09102ee40b2fa824 881121678a11618c 892930208a114004
ca75c000c51dc000 ca75c000c51dc000 ca75c000d11dc000 09302e9e0b0607e9
881123658a1161f8 892930208a114004 ca75c000d11dc000 ca75c000d11dc000
85b09348033fffff 053c0000821063ff 8410a0e083287020 8410800181c08000
0100000000000000
```

Booting up, generating memory images and other system operations consume large number of cycles. Therefore, to get the exact number of cycles for the DFP instructions we simulated an empty assembly file that only includes the boot up and the initializations. From the regression report shown, the booting up and system initializations take 1762 cycles; hence, the total number of cycles to perform the DFP test program for any of the three instructions is 10 cycles. These 10 cycles includes the floating point registers initializations.

When comparing these results with the results of the software libraries IBM decNumber and Intel Decimal library reported in [56], we find that at least we have a speed-up of more than 10 when perform the DFP operations using hardware instructions instead of software routines. Table 6.3 illustrates the cycle-count comparison for each of the implemented instructions. DPD64 is the

decNumber type that represents the Decimal64 precision; BID64 is the Intel type that represents that same precision.

```
=====================================================
            STATUS OF REGRESSION IN OPENSPARC T2
=====================================================

Summary for
/home/Original/OpenSPARCT2/OpenSparc_Simulation/cmp1_vcs_regression_201
2_05_01__LINUX_0
=============================================================
          Group:Total |   PASS |    FAIL |          Cycles |       Time |        C/S |
-------------------------------------------------------------------------------------------
         empty.s:     1 |        1 |        0 |         1762.50 |      13.20 |   133.52 |
        dec_sub.s:    1 |        1 |        0 |         1772.50 |      12.27 |   144.46 |
        dec_mul.s:    1 |        1 |        0 |         1772.50 |      12.11 |   146.37 |
        dec_add.s:    1 |        1 |        0 |         1772.50 |      12.61 |    140.56 |
-------------------------------------------------------------------------------------------
            ALL:   4 |    4 |    0 |          7080.00 |      50.19 |141.06 |
=============================================================

Total Diags  :              4
Total Passed :              4
Total Unknown:              0
Total Unfini :              0
Total Fail   :              0
Total Cycles :           7080.00
Total Time   :           50.19
Average C/S  :           141.06
=============================================================
```

	DPD64	BID64	Our H.W.
Add	154	109	10
Sub	289	126	10
Mul	296	117	10

Table 6.3: Cycle count for the decNumber library, Intel library and the new H.W. instructions

In order to measure the effect of the added DFPU on the area of the FGU, the area profiling for the original OpenSPARC T2 FGU and for the modified one including the DFPU was obtained using the Synopsys Design Compiler. Table 6.4 illustrates that the area is increased by 17.4% only.

	Area (μm)
OpenSPARC T2 FGU	151342
Modified FGU including DFPU	177662

Table 6.4: Area profile

6.5 Future Work

Future work may focus in the following issues:

- Extending our implementation to 128-bit standard decimal precision. We did not implement the Decimal128 instructions as the OpenSPARC T2 does not support quad precision operations till now.

- Extending the functionality of the floating-point arithmetic unit to include all the decimal floating point operations in the standard.

- Running test programs on a real UltraSPARC T2 machine and get the measurements of using software libraries. We did not manage to run a program linked with the software libraries because of environment's issues.

6.6 Conclusion

In this thesis we proposed the first open-source processor that includes the decimal floating point capability. It is a modified version of the OpenSPARC T2 processor from Sun/Oracle. We implemented the basic instructions (Addition, Subtraction, Multiplication, Fused Multiply-Add and Fused Multiply-Subtract).

We also extended the SPARC ISA to include the new defined DFP instructions. Moreover, we engendered a software tool chain to support the new capability. We edited the GNU GCC compiler to generate the SPARC Assembly programs and binaries from C programs that define DFP variables.

We tested the new design using the verification environment attached to the OpenSPARC T2 package. In comparison with the reported results of the software libraries, we have a speed-up of more than ten times over them.

REFERENCES

[1] T. Dantzig, Number, the Language of Science. The Macmillan Corporation,1930.

[2] P. E. Ceruzzi, A History of Modern Computing. The MIT Press, 2003.

[3] M. ibn Musa Al-Khawarizmi, The Keys of Knowledge. around 830 C.E.

[4] A. G. Bromley, “Charles Babbage’s analytical engine, 1838,” Annals of the History of Computing, vol. 20, no. 4, pp. 29–45, 1982.

[5] H. H. Goldstine and A. Goldstine, “The electronic numerical integrator and computer (ENIAC),” Annals of the History of Computing,IEEE, vol. 18, pp. 10 –16, Mar. 1996.

[6] A. W. Burks, H. H. Goldstine, and J. von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” tech. rep., Institution for Advanced Study, Princeton, 1946

[7] W. Bouchholz, “Fingers or fists ? (the choice of decimal or binary representation),” Communications of the ACM, vol. 2, pp. 3–11, Dec.1959.

[8] M. F. Cowlishaw, “Decimal floating-point: algorism for computers,”in the 16th IEEE Symposium on Computer Arithmetic (ARITH-16), pp. 104 – 111, June 2003.

[9] A. Vazquez, High Performance Decimal Floating Point Units. PhD thesis, University of Santiago de Compostela, 2009.

[10] C. R. Baugh and B. A. Wooley, “A two’s complement parallel array multiplication algorithm,” IEEE Transactions on Computers, vol. C-22, pp. 1045 – 1047, Dec. 1973.

[11] L. Dadda, “Some schemes for parallel multipliers,” Alta Frequenza, vol. 34, pp. 349–356, Mar. 1965.

[12] P. M. Kogge and H. S. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence equations,” IEEE Transactions on Computers, vol. C-22, pp. 786 –793, Aug. 1973.

[13] R. H. Larson, “High-speed multiply using four input carry-save adder,” IBM technical Disclosure Bulletin, vol. 16, pp. 2053–2054, Dec. 1973.

[14] L. P. Rubinfield, “A proof of the modified Booth’s algorithm for multiplication,” IEEE Transactions on Computers, vol. C-24, pp. 1014 – 1015, Oct. 1975.

[15] INTEL, 8080/8085 Floating-Point Arithmetic Library User’s Manual. Intel Corporation, 1979.

[16] A. Heninger, “Zilog’s Z8070 floating point processor,” Mini Micro Systems, pp. 16/2/1–7, 1983.

[17] IEEE, ed., 854-1987 (R1994) IEEE Standard for Radix-Independent Floating-Point Arithmetic. 1987. Revised 1994.

[18] IEEE Task P754, IEEE 754-2008, Standard for Floating-Point Arithmetic. Aug. 2008.

[19] M. F. Cowlishaw, “The ‘telco’ benchmark.” World-Wide Web document., 2002. http://www2.hursley.ibm.com/decimal/telco.html.

[20] L. K. Wang, C. Tsen, M. J. Schulte, and D. Jhalani, “Benchmarks and performance analysis of decimal floating-point applications,” in the 25th IEEE International Conference on Computer Design (ICCD-25), pp. 164 –170, Oct. 2007.

[21] H. A. H. Fahmy, R. Raafat, A. M. Abdel-Majeed, R. Samy, T. ElDeeb, and Y. Farouk, “Energy and delay improvement via decimal

floating point units," in the 19th IEEE Symposium on Computer Arithmetic (ARITH-19), pp. 221 –224, June 2009.

[22] C. F. Webb, "IBM z10: The next-generation mainframe microprocessor," IEEE Micro, vol. 28, no. 2, pp. 19 –29, 2008.

[23] E. M. Schwarz and S. R. Carlough, "Power6 decimal divide," in the 18th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP-18), pp. 128 –133, July 2007.

[24] P. K. Monsson, "Combined binary and decimal floating-point unit," Master's thesis, Technical University of Denemark, 2008.

[25] Sun Microsystems, BigDecimal (Java 2 Platform SE v1.4.0), 2002. http://java.sun/com/products.

[26] M. Cowlishaw, The decNumber C library, Nov. 2006. http://download.icu-project.org/ex/files/decNumber/decNumbericu-337.zip.

[27] Intel Corporation, Intel decimal floating-point math library, 2010. http://software.intel.com/enus/articles/intel-decimal-floatingpoint-math-library/.

[28] J. Thompson, N. Karra, and M. J. Schulte, "A 64-bit decimal floating point adder," in the 3rd IEEE Computer society Annual Symposium on VLSI (ISVLSI-3), pp. 297 –298, Feb. 2004.

[29] L. K.Wang andM. J. Schulte, "Decimal floating-point adder and multifunction unit with injection-based rounding," in the 18th IEEE Symposium on Computer Arithmetic (ARITH-18), pp. 56 –68, June 2007.

[30] L. K. Wang and M. J. Schulte, "A decimal floating-point adder with decoded operands and a decimal leading-zero anticipator," in the 19th IEEE Symposium on Computer Arithmetic (ARITH-19), pp. 125 –134, June 2009.

[31] A. Vazquez and E. Antelo, “Conditional speculative decimal addition,” in the 7th Conference of Real Numbers Computers (RNC-7), pp. 47 –57, July 2006.

[32] K. Yehia, H. A. H. Fahmy, and M. Hassan, “A redundant decimal floating-point adder,” in the 44th Asilomar Conference on Signals, Systems and Computers (Asilomar-44), pp. 1144 –1147, Nov. 2010.

[33] A. Vazquez, E. Antelo, and P. Montuschi, “A new family of high.performance parallel decimal multipliers,” in the 18th IEEE Symposium on Computer Arithmetic (ARITH-18), pp. 195 –204, June2007.

[34] G. Jaberipur and A. Kaivani, “Improving the speed of parallel decimal multiplication,” IEEE Transactions on Computers, vol. 58, pp. 1539–1552, Nov. 2009.

[35] M. A. Erle, M. J. Schulte, and B. J. Hickmann, “Decimal floating-point multiplication via carry-save addition,” in the 18th IEEE Symposium on Computer Arithmetic (ARITH-18), pp. 46 –55, June 2007.

[36] B. Hickmann, A. Krioukov, M. Schulte, and M. Erle, “A parallel IEEE p754 decimal floating-point multiplier,” in the 25th IEEE International Conference on Computer Design (ICCD-25), pp. 296 –303, Oct.2007.

[37] R. Raafat, A. M. Abdel-Majeed, R. Samy, T. ElDeeb, Y. Farouk, M. Elkhouly, and H. A. H. Fahmy, “A decimal fully parallel and pipelined floating point multiplier,” in the 42nd Asilomar Conference on Signals, Systems and Computers (Asilomar-42), pp. 1800 –1804,Oct. 2008.

[38] A. Vazquez, E. Antelo, and P. Montuschi, “Improved design of high-performance parallel decimal multipliers,” IEEE Transactions on Computers, vol. 59, pp. 679 –693, May 2010.

[39] R. Samy, H. A. H. Fahmy, R. Raafat, A. Mohamed, T. ElDeeb, and Y. Farouk, “A decimal floating-point fused-multiply-add unit,” in the 53rd

IEEE Midwest Symposium on Circuits and Systems (MWSCAS-53), pp. 529 –532, Aug. 2010.

[40] L. K. Wang and M. J. Schulte, “Decimal floating-point division using Newton-Raphson iteration,” in the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP-15), pp. 84 –95, Sept. 2004.

[41] H. Nikmehr, B. Phillips, and C. C. Lim, “Fast decimal floating-point division,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 14, pp. 951 –961, Sept. 2006.

[42] A. Vazquez, E. Antelo, and P. Montuschi, “A radix-10 SRT divider based on alternative bcd codings,” in the 25th IEEE International Conference on Computer Design (ICCD-25), pp. 280 –287, Oct. 2007.

[43] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi, “Leading-zero anticipatory logic for high-speed floating point addition,” IEEE Journal of Solid-State Circuits, vol. 31, pp. 1157–1164, Aug. 1996.

[44] A. Vazquez, J. Villalba, and E. Antelo, “Computation of decimal transcendental functions using the CORDIC algorithm,” in the 19th IEEE Symposium on Computer Arithmetic (ARITH-19), pp. 179 –186, June 2009.

[45] J. Harrison, “Decimal transcendentals via binary,” in the 19th IEEE Symposium on Computer Arithmetic (ARITH-19), pp. 187 –194, June 2009.

[46] D. Chen, Y. Zhang, Y. Choi, M. H. Lee, and S. B. Ko, “A 32-bit decimal floating-point logarithmic converter,” in the 19th IEEE Symposium on Computer Arithmetic (ARITH-19), pp. 195 –203, June 2009.

[47] R. Tajallipour, D. Teng, S. B. Ko, and K. Wahid, “On the fast computation of decimal logarithm,” in the 12th International Conference on Computers and Information Technology (ICCIT-12), pp. 32 –36, Dec.2009.

[48] A. Y. Duale, M. H. Decker, H. .-G. Zipperer, M. Aharoni, and T. J.Bohizic, “Decimal floating-point in z9: An implementation and testing perspective,” IBM Journal of Research and Development, vol. 51,pp. 217 – 227, Jan. 2007.

[49] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O’Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, “IBM POWER6 microarchitecture,” IBM Journal of Research and Development,vol. 51, pp. 639 –662, Nov. 2007.

[50] J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti, “Design of the Power6 microprocessor,” in the 54th IEEE International Conference on Solid-State Circuits (ISSCC-54), pp. 96 –97, Feb. 2007.

[51] L. C. Heller and M. S. Farrell, ‘‘Millicode in an IBM zSeries Processor,’’ IBM J. Res. & Dev. 48, No. 3/4, 425–434 (2004)

[52] L. Eisen, J. W. Ward III, H.-W. Tast, N. Ma¨ ding, J. Leenstra, S. M. Mueller, C. Jacobi, J. Preiss, E. M. Schwarz, and S. R. Carlough, ‘‘IBM POWER6 Accelerators: VMX and DFU,’’ IBM J. Res. & Dev. 51, No. 6, 663–683 (2007).

[53] E. M. Schwarz and S. Carlough, ‘‘POWER6 Decimal Divide,’’ Proceedings of the IEEE 18th International Conference on Application-Specific Systems, Architectures and Processors.

[54] C.-L. K. Shum, F. Busaba, S. Dao-Trong, G. Gerwig, C. Jacobi, T. Koehler, E. Pfeffer, B. R. rasky, J. G. Rell, and A. Tsai “Design and microarchitecture of the IBM System z10 microprocessor”

[55] M. A. Erle, J. M. Linebarger, and M. J. Schulte, “Potential Speedup Using Decimal Floating-Point Hardware.” Submitted to the 36th Asilomar Conference on Signals, Systems and Computers, Nov 2002.

[56] M.Anderson, C.Tsen, L.Wang, K.Compton, M.Schulte, “Performance Analysis of Decimal Floating-Point Libraries and Its Impact on Decimal Hardware and Software Solutions,” IEEE Int. Conf on Computer Design, pp.465-471, 2009.

[57] S. Gochman et al., “Intel Pentium M Processor: Microarchitecture and Performance”, in Intel Technology Journal, Vol. 7, No. 2, pp.22–36, 2003.

[58] K. Yeager et. al., “R10000 Superscalar Microprocessor,” presented at Hot Chips VII, Stanford, CA, 1995.

[59] M. Shah, et al, “UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SOC”, IEEE Asian. Solid-State Circuits Conf, Nov. 2007.

[60] D. Levitan, T. Thomas, and P. Tu. “The PowerPC 620 microprocessor: A high performance superscalar RISC processor.” COMPCON 95, 1995.

[61] J. Heinrich. MIPS R4000 Microprocessor User’s Manual. MIPS Technologies Inc., 2nd edition, 1994.

[62] K. Olukotun et al., “The Case for a Single-Chip Multiprocessor,” Proc. Int’l Conf. Architectural Support for Programming Languages and Operating Systems, ACM, 1996, pp. 2-11.

[63] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In 22nd Annual International Symposium on Computer Architecture, pages 392–403, June 1995.

[64] L. Spracklen and S. G. Abraham. Chip Multithreading: Opportunities and Challenges. In Proc. of the IEEE Int'l Symp.on High-Performance Co mputer Architecture, pages 248–252, 2005.

[65] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. IEEE Micro, 20(2):71–84, 2000.

[66] L. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA 00), IEEE CS Press, 2000,pp. 282-293.

[67] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy. "IBM Power4 System microarchitecture". IBM Journal of Research and Development, 46(1):5–26, 2002.

[68] R. Kalla, B. Sinharoy, and J. Tendler , "IBM POWER5 chip: a dual-core multithreaded processor," in IEEE Micro, Vol. 24, No. 2, pp.40–47, 2004.

[69] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. "Power7: IBM's next-generation server processor".IEEE Micro, 30:7–15, March 2010

[70] M. Tremblay et al., "The MAJC Architecture: A Synthesis of Parallelism and Scalability," IEEE Micro, Vol. 20, No. 6, pp. 12–25,2000.

[71] M. Tremblay, "MAJC-5200: A VLIW Convergent MPSOC," in Microprocessor Forum 1999, 1999.

[72] S. Kapil, "UltraSPARC Gemini: Dual CPU Processor," in Hot Chips 15, http://www.hotchips.org/archive/, 2003.

[73] Q. Jacobson, "UltraSPARC IV Processors," in Microprocessor Forum 2003, 2003.

[74] SUN Microsystems, "OpenSPARC T1 Micro Architecture Specification," available at http://www.opensparc.net

[75] SUN Microsystems, “OpenSPARC T2 Micro Architecture Specification,” available at http://www.opensparc.net

[76] David L. Weaver and Tom Germond, editors. The SPARC Architecture Manual. Prentice Hall, 1994. SPARC International, Version 9.

[77] SUN Microsystems, “UltraSPARC Architecture 2007,” available at http://www.opensparc.net

[78] GCC mission statement, available at http://gcc.gnu.org/gccmission.html

[79] Oracle, “SPARC Assembly Language Reference Manual,”, Nov. 2010,

[80] Arthur Griffith, “GCC: The complete reference”, Sep. 2002, McGraw-Hill

[81] Richard. M. Stallman, “GCC Internals”, 2010, available at http://gcc.gnu.org/onlinedocs/gccint 2007.

[82] GMP library, available at http://gmplib.org/

[83] *MPFR library, available at* http://www.mpfr.org/.

[84] MPC library, available at http://www.multiprecision.org/.

[85] GCC Binutils, available at http://ftp.gnu.org/gnu/binutils/.

[86] GCC source code, available at http://mirrors.rcn.net/pub/sourceware/gcc/releases/.

[87] A. A. R. Sayed-Ahmed, H. A. H. Fahmy, and M. Y. Hassan, “Three engines to solve verification constraints of decimal floating-point

operation," in the 44th Asilomar Conference on Signals, Systems and Computers (Asilomar-44), pp. 1153 –1157, Nov. 2010.

[88] A. M. Eltantawy, "Decimal Floating Point Unit based on a Fused Multiply-Add module", MSc. Thesis, Cairo University, 2011.

دمج وحدة حسابية للأرقام العشرية ذات النقطة العائمة في معالج مفتوح المصدر وتعديل المترجم الخاص به

إعداد

محمد حسني أمين حسن

رسالة مقدمة الي

كلية الهندسة , جامعة القاهرة

كجزء من مطلبات الحصول علي

درجة الماجيستير في

هندسة الاتصالات والالكترونيات الكهربية

كلية الهندسة ، جامعة القاهرة

الجيزة ، مصر

2012

دمج وحدة حسابية للأرقام العشرية ذات القلة العائمةفي معالج مفتوح المصدر وتعديل المترجم الخاص به

إعداد
محمد حسني أمين حسن

رسالة مقدمة الي
كلية الهندسة , جامعة القاهرة

كجزء من مطلبات الحصول علي
درجة الماجيستير في
هندسة الاتصالات والالكترونيات الكهربية

تحت إشراف
أم.د حسام علي حسن فهمي

كلية الهندسة ، جامعة القاهرة
الجيزة ، مصر
2012