Institute of Parallel and Distributed Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3260

# A Hardware Architecture for Numerical Instability Detection Based on Discrete Stochastic Arithmetic

Yousef Baroud

| | |
|---|---|
| **Course of Study:** | Information Technology |
| **Examiner:** | Prof. Dr. Sven Simon |
| **Supervisor:** | MSc. Wenbin Li |
| **Commenced:** | 9. May 2011 |
| **Completed:** | 8. November 2011 |
| **CR-Classification:** | G.1.0, D.2.5, B.2.1, B.6.0, B.8 |

# Abstract

Numerical validation of computed results is of great importance especially in sceintific computing. Due to the use of finite representation of real numbers, round-off errors are introduced and accumulated in arithmetic operations. Nowadays, softwares tend to run longer. This exaggerate the problem as the computed results would get severely contaminated by the propagation of the round-off errors which might, at some point, lead to obtaining unreliable results.

The Discrete Stochastic Arithmetic (DSA) provides an effective and reliable approach to validate the numerical accuracy of the computed results. In DSA, a code is run $N$ times with random rounding at every floating point operation, and the numerical accuracy information can be obtained by calculating the confidence interval of the randomly rounded results.

In this work, we present a novel hardware architecture which efficiently implements the DSA. A Numerical Analysis Unit (NAU) that estimates the numerical accuracy of any intermediate result and detects numerical instabilities has been implemented based on a hardware-reduced approach. The NAU has been integrated into a high-performance FPGA system that consists of two PowerPC processors which use stochastic floating point units. Upon catching numerical instabilities, the NAU raises exceptions to the PowerPCs stopping them at the instruction that caused the exception.

In contrary to the existent implementations, the proposed implementation has been developed to meet three constrains; minimal original source code modifications, minimizing hardware resource cost and exhibiting a good performance.

An extension to a state of the art debugger has been developed for the debugging of numerical instabilities in a code. This extension adds functionalities which facilitate communicating with the FPGA system. Moreover, functionality specific to numerical accuracy, such as getting more details about a NAU exception or resuming execution after catching one, is provided through this extension.

# Acknowledgments

I would like to thank my advisor M.Sc. Wenbin Li for his continuous support and advice throughout working on this thesis. I would like also to thank Prof. Dr. Sven Simon for giving me the chance to work on such an interesting problem.

Thanks also go to the department of Parallel Systems for making it easy for me to conveniently work on the thesis by providing the necessary software tools and the evaluation board.

Special thanks to Dipl.-Phys. Alen-Pilip Prskalo and Dr. Andrew Aird for their support and advice.

Finally, I would like to thank my family for their patience and support; especially my brother Hashem for proofreading the thesis. I would also like to thank Mahmoud, Liana, Mohammed, A. Sleem, L. Mustafa and A. Bannoura.

# Contents

# List of Figures

# List of Tables

# List of Listings

# 1 Introduction

## 1.1 Motivation

Representing infinitely many real numbers in a finite number of bits (usually 32 or 64) requires approximate representation. Thus, if a result of a floating point calculation cannot be exactly representable on the target machine, it needs to be rounded to fit into the finite representation. This approximation introduces the *round-off error*. As the program runs longer, the propagation of this error from one floating point operation to another might, at some point, lead to badly affecting the computed result making it inaccurate or even wrong. Validation of the numerical results in scientific computing is possible by applying *Discrete Stochastic Arithmetic* (DSA).

The basic idea of DSA can be summarized as follows:

1. Running the same code synchronously *N* times while randomly rounding the results (up or down) at every floating point operation.
2. Based on the results obtained after every floating point operation from the *N* runs, it is possible to estimate the accuracy of the calculation.

An efficient implementation of DSA should satisfy these three points:

1. Requires minimal changes to be incorporated in the original source code.
2. Does not notably slow down the execution of a program.
3. Requires minimal additional hardware.

There are at least two existent implementations supporting DSA. The first one is the CADNA library which is a software implementation of DSA. The execution time of a program, when using CADNA, will be orders of magnitude the execution time of the original program. More important, many modifications to the source code are required to migrate the code to make use of the CADNA library. This is tedious and error prone especially in softwares with multiple thousands SLOC[1]. The second implementation of DSA is a hardware implementation presented in [ACM04]. This implementation requires modifications to the original source code and does not support all floating point operations.

In this thesis we present a new hardware implementation of DSA. Tendency to meet the three points stated earlier has been a core target. The proposed implementation is centered

[1]Source Lines of Code

around a hardware-reduced approach for estimating the accuracy of results. While saving hardware resources and giving a good estimation of the accuracy of results, this approach made it possible to reach lower latencies and higher operating frequencies.

On the software level, a numerical accuracy debugging tool to help localizing numerical instabilities has been developed. Getting more details about the type of a numerical instability has become also possible using this tool.

## 1.2 The Scope of Work

The work in this thesis can be categorized into two parts:

**Hardware:** A Numerical Analysis Unit (NAU), which is able to detect numerical instabilities and raise exceptions upon catching one, has been implemented.

**Software:** A means has been developed means to catch the exceptions raised to the processors and check their cause.

The original floating point units used in the architecture were implemented in a previous project and not in the scope of this thesis. Though, changes have been carried out on these units in order to properly integrate the NAU and communicate the numerical instability exceptions, as applicable, to the processors.

## 1.3 Thesis Structure

This thesis is written in seven chapters (including the introduction) and an appendix. A brief description of these chapters is provided as follows:

1. **Chapter 2:** The theoretical basis of the CESTAC method and DSA is presented along with the discussion of two existent implementations of DSA.
2. **Chapter 3:** Basic components of the systems are presented.
3. **Chapter 4:** A detailed description of the different components of the numerical analysis unit is provided. Performance metrics and resource utilization is also provided. In addition, The necessary changes that are required for the stochastic floating point unit are presented.
4. **Chapter 5:** Managing exceptions is discussed. The mechanism of catching and processing numerical instability exceptions is presented as well.
5. **Chapter 6:** Performance evaluation of different DSA implementations is presented based on benchmarking.
6. **Chapter 7:** The features of the proposed implementation are summarized along with possible future enhancements.

7. **Appendix A:** Look-up tables used in the implementation and benchmarks used for performance evaluation are provided. In addition, a C program that is used for validation purposes and a TCL script that works as a numerical accuracy debugging tool are provided.

# 2 Review of Stochastic Discrete Arithmetic

In this chapter, a brief introduction of floating point (FP) numbers and round-off error is presented in Section 2.1. The Discrete Stochastic Arithmetic (DSA) is introduced in Section 2.2. Two implementations of DSA, one in software (Section 2.3) and the other in hardware (Section 2.4), are presented and evaluated.

## 2.1 Floating Point Representation

Real numbers can be represented as $(s \cdot b^e \cdot m)$, where: $s$ is the sign, $m$ is the mantissa (significant) that satisfies the inequality $1 \leq m < b$ and $b$ is the basis of exponent $e$. The same convention is followed when representing a floating point number on a machine where $b$ is chosen as 2, and $m$ is approximated to $m'$ due to the limited number of bits used in the representation.

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) defines five basic formats [iee08]. The most used representations are the single precision and double precision. As depicted in Figure 2.1, the single precision representation is encoded in 32 bits; 1 bit is used to encode the sign (0 for $+$ and 1 for $-$), the exponent is encoded in 8 bits and 23 bits are used to represent the mantissa. As the exponent might be of a negative value, a bias is added to the actual exponent. In single precision, the bias is 127. Similarly, the double precision representation, depicted in Figure 2.2, is encoded the same way but with different number of bits for the mantissa and the exponent. The bias added to the actual exponent in the double precision format is 1023.

As limited bits are used to represent floating point numbers, not every real number is exactly representable. Let $F$ be a floating point number which cannot exactly be represented in the target format, $F^+$ be the next larger representable number and $F^-$ be the next smaller representable number. Then, $F$ is either approximated to $F^+$ or $F^-$ depending on the rounding mode being used. These rounding modes are:

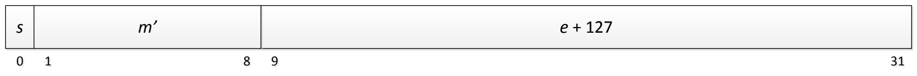


**Figure 2.1:** Single precision floating point number representation.

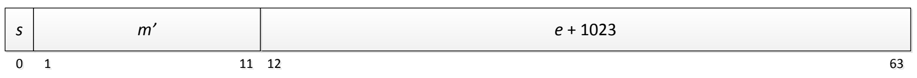


**Figure 2.2:** Double precision floating point number representation.

- **Round to nearest, ties to even:** The value which is closer to $F$ is used. In case of a tie ($F$ has equal distance to $F^+$ and $F^-$), the one that is even (least significant bit in the mantissa is 0) is used.
- **Round toward zero:** Also known as truncation. The smaller one in magnitude is used.
- **Round toward $+\infty$:** Also known as rounding up. $F^+$ is used.
- **Round toward $-\infty$:** Also known as rounding down. $F^-$ is used.

It is essential here to introduce the concept of *random rounding*. If a number that results from an FP operation is not exactly representable, then random rounding is the process of choosing either $F^-$ or $F^+$ with equal probability to represent that number.

This rounding solution introduces *round-off error*. A real number $x$ that is represented exactly as: $x = s \cdot m \cdot b^e$, can be represented on the machine as: $X = x \cdot (1 + b^{-p}\alpha) \cdot b^e$, where: $p$ is the number of bits in the mantissa, $\alpha$ is the normalized relative error and $b^{-p}\alpha$ is the relative error [Che]. Depending on the rounding mode used, $\alpha$ can take these values:

- **Round to nearest:** $\alpha \in [-0.5, 0.5[$.
- **Round toward zero:** $\alpha \in [0, 1[$.
- **Round toward $+\infty$:** $\alpha \in [-1, 1[$.
- **Round toward $-\infty$:** $\alpha \in [-1, 1[$.

In computational intensive programs that run for longer time (days or even weeks) the study of propagation of round-off error in consecutive FP operations is of crucial importance. The accuracy of final results might be badly affected by these errors to the point that a final result is considered as of unacceptable accuracy or even wrong.

## 2.2 Discrete Stochastic Arithmetic

Discrete Stochastic Arithmetic (DSA) is the association of the synchronous implementation of the CESTAC method (Section 2.2.1), the informational zero concept (Section 2.2.2) and the stochastic relations (Section 2.2.3)[Vig04].

### 2.2.1 The CESTAC Method

**C**ontrole et **E**stimation **S**tochastic des **A**rrondis de **C**alculs (CESTAC) method was first developed by J. Vignes and M.L. Porte in [VP74] in order to estimate the effect of round-off error propagation on results and to detect numerical instabilities.

For better understanding of the method, let us first consider this scenario. Assume we have a code of $n$ floating point operations. This code is run several times, with the intermediate results at every FP operation rounded up or down, so as to consider all the possible propagations of the round-off error. The number of common digits, $k$, of the results obtained, $r_i \in R$, is noted. Logically, this number also gives the first $k$ exact significant digits of the exact result $r$.

What makes this approach unusable is the high cost imposed by running the code $2^n$ times in order to cover all the possible propagations of the round-off error. The CESTAC method, which is based on a probabilistic approach, is developed to overcome this problem.

In this probabilistic (stochastic) approach [Vig93], only a subset of the computed results, $\hat{r}_i \in \hat{R} \mid \hat{R} \subset R$, is further considered. The elements of the subset $\hat{R}$ are obtained by randomly rounding the calculated results up or down at every FP operation with equal probability. Every sample, $\hat{r}_i$, of these $N$ samples can be modeled as:

$$\hat{r}_i = r + \sum_{k=1}^{n} g_k(d) 2^{-p} \alpha_k + O(2^{-2p}),$$

where $r$ is the exact result, $n$ is the number of floating point operations, $g_k(d)$'s are quantities depending exclusively on data and algorithm, $p$ is the wordlength of the mantissa, $\alpha_k$'s are normalized rounding errors which are independent and identically distributed random variables. The computed result is taken as the average of all the samples $\hat{r}_i$:

$$\bar{r} = \frac{1}{N} \sum_{i=1}^{N} \hat{r}_i.$$

The exact significant digits of $\bar{r}$ is given by the formula [Vig93]:

$$C_{\bar{r}} = \log_{10} \left( \frac{\sqrt{N} \cdot |\bar{r}|}{\tau_\beta \cdot \sigma} \right),$$

where $\sigma^2 = \frac{1}{N-1} \cdot \sum_{i=1}^{N} (\hat{r}_i - \bar{r})^2$, and $\tau_\beta$ is the value of the Student distribution for $N-1$ degrees of freedom and a probability level $1-\beta$.

In the CESTAC method, two hypotheses must hold true for the model to be reliable [Vig04]:

1. The elementary round-off errors $\alpha_i$ of the FP arithmetic operations are independent, centered and uniformly distributed variables.
2. The negligibility of the items in the order of $2^{-2p}$ is legitimate.

The first hypothesis holds true due to the robustness of the Student's test. However, for the second hypothesis to hold true, two more checks must be taken into account [Vig04]:

1. The operands of any multiplication are both significant (i.e. not an *informational zero*).
2. The divisor of any division is significant.

### 2.2.2 Informational Zero

A result is considered as an informational zero (denoted as @.0), when one of these conditions is true [Vig93]:

1. $\forall i,\ i = 1, \ldots, N, \quad \hat{r}_i = 0$.
2. $C_{\bar{r}} \leq 0$.

### 2.2.3 Stochastic Relations

Relational operators need to be redefined in order to only consider the significant part of the operands. Let $X$ and $Y$ be $N$-samples provided by CESTAC method, then [Vig93]:

- $X =^s Y \quad$ if $\quad X - Y = @.0$, where $=^s$ is the stochastic equality.
- $X >^s Y \quad$ if $\quad \bar{X} > \bar{Y}$ and $X - Y \neq @.0$, where $>^s$ is the stochastic greater than inequality.
- $X \geq^s Y \quad$ if $\quad \bar{X} \geq \bar{Y}$ or $X - Y = @.0$, where $\geq^s$ is the stochastic greater than or equal inequality.

The way we run the code $N$ times is crucial. If the code is run sequentially, there will be no means to check the validity of the second hypothesis. Thereby, asynchronous implementation is considered unreliable.

## 2.3 CADNA Library

**C**ontrol of **A**ccuracy and **D**ebugging for **N**umerical **A**ccuracy (CADNA) software library implements DSA. CADNA supports programs written in ADA, C or FORTRAN, and during the run of the code it monitors [cad, p. 5 - p. 6]:

- The numerical error due to round-off errors.
- Branching instabilities (when an informational zero (Subsection 2.2.2) is detected in branching).
- The numerical accuracy of all the intermediate results.

The implementation is done by performing $N$ times $(N = 3)$ each FP operation before proceeding to the next operation. Upon the detection of an instability or accuracy loss of the intermediate results, information is be written to the standard output.

Using the CADNA library is costly in terms of memory and running time. When using CADNA, the program runs orders of magnitude times slower, and uses more than three times the memory footprint it needs when run normally [JC08]. We have tested the performance using a benchmark on a T6600 intel processor, and CADNA increased the running time of the program more than 170 times.

### 2.3.1 Technical Deception

CADNA introduces new (stochastic) data types to be used in the code which are `float_st` and `double_st` which are correspondent to the normal `float` and `double`, respectively. A stochastic data type is merely a triplet of floating point values, where the randomly rounded results after each FP operations are saved. FP operations `(+, -, *, /)` are overloaded to support the new data types and to perform the random rounding for the values after each computation. Order relations $(<, \leq, ==, \geq, >)$ are overloaded as well and redefined as in Subsection 2.2.3. Functions in `math.h` (such as `sin, cos, log...`) are also overloaded to make use of stochastic data types [cad, p. 8].

Furthermore, functions such as `printf()` and `scanf()` cannot be used directly to display or read-in stochastic data types. In the `printf()` case, it should be used in joint with `strp()` library function which outputs a `string` instead of a floating point number. This string gives the mean value of the $N$ triplet with only its significant digits. In the `scanf()` case, it is used at first to save the value to a normal floating point data type. Then, assignment is used to assign this value to the stochastic floating point data type variable.

The number of exact significant decimal digits can be retrieved by a call to the library function `x.nb_significant_digit()`. Upon the loss of significance of `x`, a call to the function will return 0. The $N$ triplet is displayed using the `x.display()` function call.

### 2.3.2 Numeric Example Using CADNA

For comparison, a small program with and without the use of CADNA Library is given in Listing 2.1 and Listing 2.2 respectively. In Listing 2.2 changes to the source code are highlighted in red. Apparently, many changes are needed to be incorporated in the original source code to make use of CADNA and this is not trivial in the case of complex existent programs to use CADNA.

```c
#include <stdio.h>

int main()
{
  double x = 1.1;
  double y = 1.3;
  double res;

  res = x + y;
  printf("res=%f\n",res);
}
```

**Listing 2.1:** Original source code without the use of CADNA library.

```c
#include <cadna.h>
#include <stdio.h>

int main()
{
  // This function is used to define the type of stability to be detected
   cadna_init(-1);

  // The use of stochastic data types
   double_st x = 1.1;
   double_st y = 1.3;
   double_st res;

  res = x + y;

  // The triplet res.x, res.y, res.z is displayed
  res. display();

  printf ("Number of significant digits = %d\n",res.nb_significant_digit());

  // Note how printf is used here with %s NOT %f
  printf("res=%s\n", strp(res));

  // Release the resources reserved by CADNA and exit
   cadna_end();
}
```

**Listing 2.2:** Modified source code with the use of CADNA library

The output after running the code is:

```
----------------------------------------------------------------
CADNA_C 1.1.1 software --- University P. et M. Curie --- LIP6
Self-validation detection: OFF
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
----------------------------------------------------------------
 +2.3999999999999999e+00 -- +2.3999999999999999e+00 -- +2.4000000000000004e+00
Number of significant digits = 15
res= 0.239999999999999E+001
----------------------------------------------------------------
CADNA_C 1.1.1 software --- University P. et M. Curie --- LIP6
No instability detected
----------------------------------------------------------------
```

CADNA library with the help of a debugger (`gdb` or `idb`) can localize the statement which caused the instability. This is possible by running the program from the debugger after setting a break point on the `instability` internal library function, which is called upon detecting any type of stability, then using the command `where` or `backtrace` to localize the instability source.

## 2.4 Hardware Implementation by Avot-Chotin and Mehrez

The Authors of [CM, ACM04] implemented DSA in hardware. Their implementation consists of two parts:

1. A floating point unit which supports the floating point operations (+, -, *), conversion between integer and floating point and comparison.
2. A specific hardware to implement the CESTAC method.

The stochastic floating point is shown in Figure 2.3. The hardware calculates the number of significant bits, detects informational zeros and performs stochastic floating point order relations as defined in Subsection 2.2.3.

The most notable part of the implementation is that they have used a simplified approach than the one presented in Section 2.2. This approach is summarized in three steps:

1. Computation of the distances between the samples; $d_1 = |\hat{r}_1 - \hat{r}_2|$, $d_2 = |\hat{r}_1 - \hat{r}_3|$, $d_3 = |\hat{r}_2 - \hat{r}_3|$.
2. For each distance $d_i$, the position of the first bit equal to 1, $p_i$, is noted.
3. The number of significant bits is $\min\{p_i\}$.

The number of subtractors, $N_{sub}$, is a function of number of samples, $N$, and can be calculated using the equation $N_{sub} = \frac{N(N-1)}{2}$. Thus, this approach becomes costly in terms of hardware when number of samples grows.

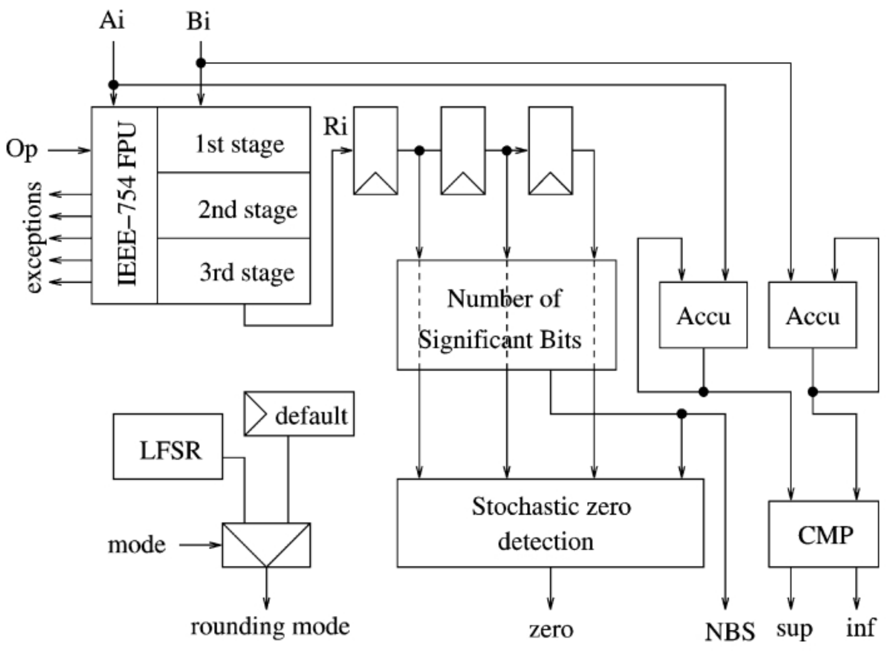


**Figure 2.3:** Stochastic floating point unit [ACM04]

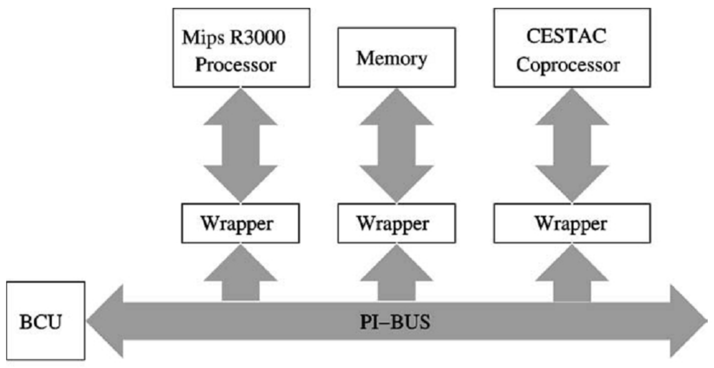


**Figure 2.4:** System architecture of the stochastic FPU [ACM04]

The complete architecture of a system which uses the stochastic floating point unit is shown in Figure 2.4. The system consists of:

- The FPU is packaged as a coprocessor and connected via a Virtual Component Interface (VCI) wrapper to the PI-bus.
- Mips R3000 processor.
- Memory to save binaries and data.
- Bus Control Unit (BCU) which manages the communication between the different components on the bus.

In this architecture, there is no support for the instruction set of the processor. Consequently, use of function calls (or assembly instructions) to communicate the data to / from the processor is required and that implies many changes to the source code of the original program. In addition, FP division and FP square root operations are not supported.

# 3 System Model

In this chapter, the components used in the design are briefly described. In Section 3.1, the Stochastic Floating Point Unit which was developed in a previous project is presented. Section 3.2 provides functional description of the Numerical Analysis Unit used to detect numerical instabilities. Section 3.3 gives the big picture of how different components of the system are utilized on the evaluation board.

## 3.1 Stochastic Floating Point Unit

### 3.1.1 Description and Features

In a previous project a **ST**ochastic **F**loating **P**oint **U**nit (STFPU) has been developed [Li10]. The STFPU is packaged as a coprocessor and attached to the PowerPC through the **A**uxiliary **P**rocessor **U**nit (APU).

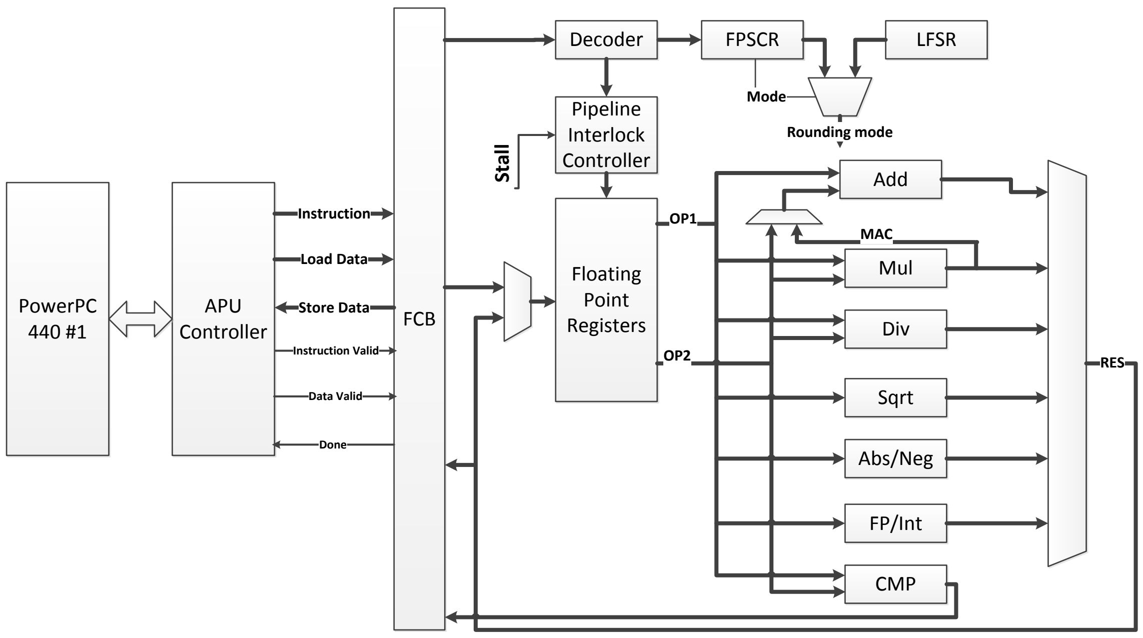


**Figure 3.1:** Hardware architecture of the STFPU.

The architecture of the STFPU is depicted in Figure 3.1. The STFPU supports random rounding by using a random number generator based on a **L**inear **F**eedback **S**hift **R**egister (LFSR). When the random rounding mode is used, rounding up or down is dependent on the output of the LFSR. The features of the STFPU are:

- The ablility to decode and execute, in a fully pipelined manner, standard PowerPC processor floating point instructions defined in [b0002, p. 98 - p. 106].
- Supporting single precision and double precision formats.
- Detecting and raising floating point exceptions (Overflaw, Underflaw,...).
- Supporting IEEE-754 rounding modes as well as random rounding.
- Supporting two running modes:
  1. **Autonomous mode:** In this mode the PowerPC can continue executing integer operations while the floating point instruction is being executed by the FPU. Moreover, more than one floating point instruction can be executed at the same time. No exact exceptions [1] can be caught by the PowerPC in this mode.
  2. **Non-Autonomous mode:** Floating point instructions which returns result to the PowerPC stalls the processor till it finishes the execution. Only one instruction can be executed in the FPU at a time in this mode. Exact exceptions can be caught by the PowerPC.

### 3.1.2 Technical Information

Table 3.1 summarizes the hardware utilization of the STFPU and the maximum operating frequency when synthesized on XC5VFX130T Virtex-5 FPGA with speed grade $-2$.

| Description | Value |
|---|---|
| Number of Slice Registers | 6385 |
| Number of Slice LUTs | 8730 |
| Maximum operating frequency | 103 MHz |

**Table 3.1:** Hardware utilization and maximum operating frequency.

The latencies of decoding and executing the elementary floating point operations, when the STFPU runs non-autonomously, are summarized in Table 3.2. The information in Table 3.2 will be used for evaluation purposes later in the thesis.

[1] By Exact exceptions we mean that the very instruction which caused the exception can be identified.

| Operation | Latency (Clock cycles) |
|---|---|
| Add/Subtract | 23 |
| Multiply | 24 |
| Divide | 50 |
| Multiply-Accumulate | 29 |

**Table 3.2:** Latencies of the floating point operators in STFPU.

## 3.2 Numerical Analysis Unit

The Numerical Analysis Unit (NAU) is centered around a resource-reduced architecture and it will be presented in great details in the next chapters. In this section, only a functional description is given. The NAU is composed of:

1. **Synchronization Unit:**

   This unit is responsible for synchronization of the FPUs in the design in order to ensure synchronous implementation of DSA. Section 2.2.3 explains the importance of such implementation.

2. **Significant Bits Estimation Unit (SBEU):**

   This unit represents the core of the NAU. It is responsible for:

   a) Validating the significance of the multiplication operands and the divisor.

   b) Checking if the results are of acceptable accuracy.

   c) Check if a cancellation error occurs in an addition / subtraction operation.

   d) Checking whether the differences in a comparison operation are significant or not.

   e) Raising exceptions to the STFPU upon detecting any instability.

3. **Comparison Support Unit:**

   The task of this unit is to support stochastic relations according to the definitions provided in Section 2.2.3.

## 3.3 System Model

Figure 3.2 shows the system model of our design (some details are omitted for clarity). As a hardware platform for our design, we have chosen ML510 evaluation board. This board provides a high performance Xilinx XC5VFX130T Virtex-5 FPGA, two hardwired PowerPC processors which can operate on a frequency up to 400MHz, two 512MB DDR2 SDRAM, two serial ports.

**X**ilinx **M**icroprocessor **D**ebugger (XMD) connects to the board through a JTAG connection. XMD can connect to the two processors on the board in one debugging session. XMD is the means to download executables to the processor. It supports many debugging features such as reading the value of a register, stopping and continuing execution from a preset breakpoint or watchpoint. One of the most important features of XMD is the possibility to extend its original functionality by means of custom TCL scripts.

GDB (The GNU Debugger) might be used to connect through a TCP connection (either remotely or locally) to the XMD server. This provides an easier way of debugging on the source code level with the help of a Graphical User interface (GUI).

The main target of using XMD and GDB here is to catch exceptions raised from NAU when detecting a numerical instability. Furthermore, they are used to localize the source of an exception (on the assembly instruction level or on the C-statement level), and to identify the type of instability.

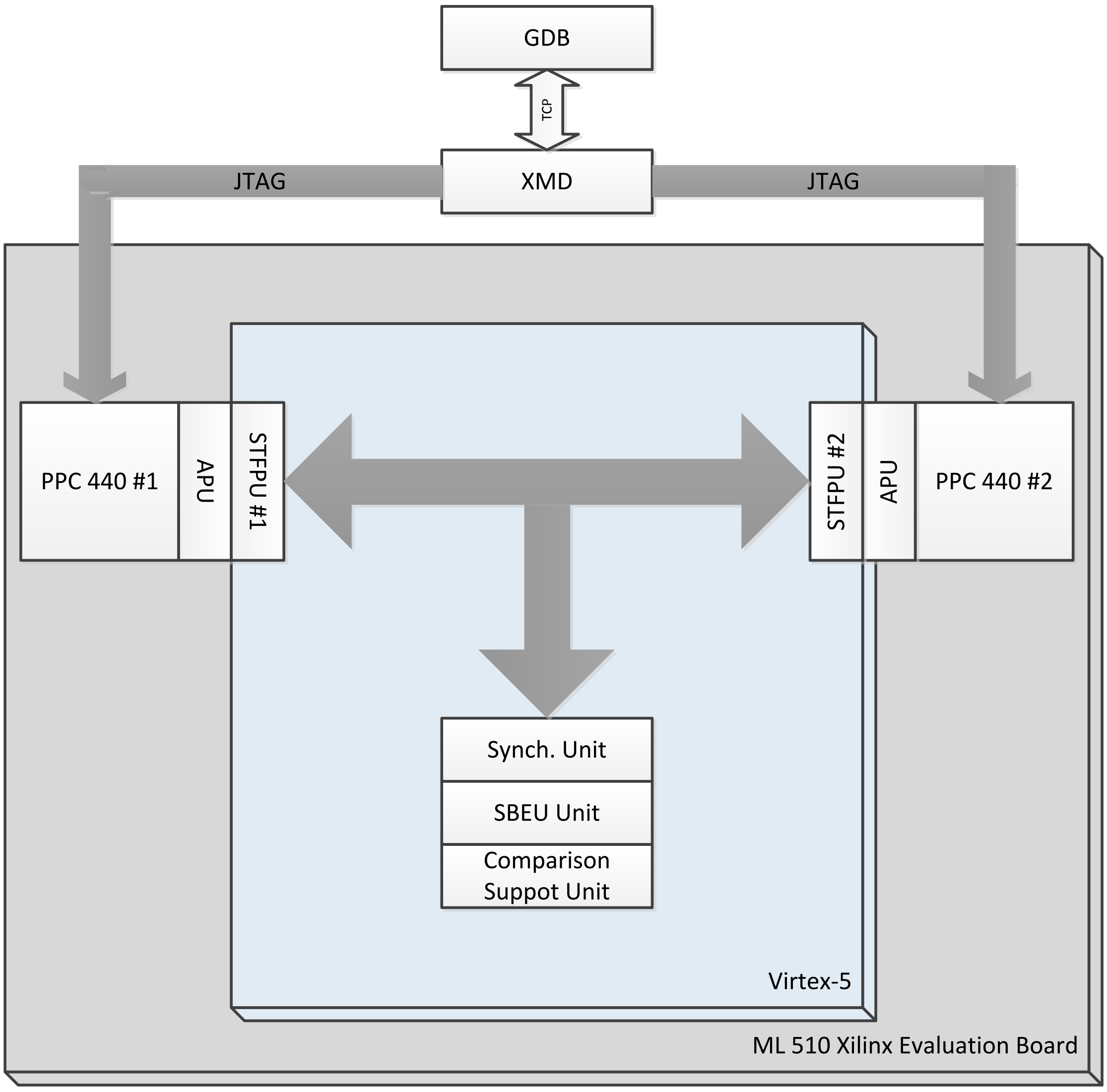


**Figure 3.2:** System model

# 4 Hardware Implementation

In this chapter we introduce the three different components used to implement DSA – namely **S**ignificant **B**its **E**stimation **U**nit (SBEU) in Section 4.1, Comparison Support Unit in Section 4.2 and the Synchronization Unit in Section 4.3. For every unit, we provide the performance metrics. Then, we introduce the mechanism to generate different NAU exceptions in Section 4.4. The necessary changes made to the STFPUs are provided in Section 4.5. Finally, we present the complete architecture to illustrate how different components were integrated on the evaluation board to implement DSA correctly in Section 4.6.

## 4.1 Significant Bits Estimation Unit

Two data paths for this unit are presented – the first is generic and can be readily extended (Subsection 4.1.1), and the other is optimized for two samples only (Subsection 4.1.2).

### 4.1.1 Generic Data Path

The equation used to calculate the number of significant digits in Section 2.2 is repeated here for convenience:

$$C_{\bar{r}} = \log_{10}\left(\frac{\sqrt{N}\cdot|\bar{r}|}{\tau_\beta \cdot \sigma}\right),$$

where $\sigma^2 = \frac{1}{N-1}\cdot\sum_{i=1}^{N}(\hat{r}_i - \bar{r})^2$, and $\tau_\beta$ is the value of the Student distribution for $N-1$ degrees of freedom and a probability level $1-\beta$.

The hardware cost of using floating point operators to calculate the square root, logarithm and division is high. Also, the high latency imposed upon using them degrades the overall performance of a system. Table 4.1 shows the hardware resource utilization and maximum operating frequencies for some FP operators at different latencies when synthesized on XC5VFX130T Virtex-5 FPGA with speed grade −2. From this table, we can observe that FP operators with shorter data format not only utilize less resources but also can operate at higher frequencies.

| Operator | Latency | Data format | | Resource utilization | | Maximum |
|---|---|---|---|---|---|---|
| | (clock cycles) | Exp. | Mant. | Slice Registers | Slice LUTs | frequency (MHz) |
| FP Add | 4 | 11 | 52 | 560 | 1005 | 245.78 |
| | 4 | 11 | 4 | 109 | 184 | 292.91 |
| FP Mul | 2 | 11 | 52 | 293 | 3215 | 134.43 |
| | 2 | 11 | 4 | 50 | 126 | 363.80 |
| FP Div | 57 | 11 | 52 | 6009 | 3225 | 345.74 |
| FP Sqrt | 57 | 11 | 52 | 3298 | 1856 | 345.74 |

**Table 4.1:** Latencies of different double precision FP operators.

A simplified approach is presented in [Li10]. This approach is based on simplification of the calculation as follows:

$$C_{\bar{r}} = \log_{10}\left(\frac{\sqrt{N}\cdot|\bar{r}|}{\tau_\beta\cdot\sigma}\right) = \log_{10}\frac{\sqrt{N}}{\tau_\beta} + \log_{10}\frac{|\bar{r}|}{\sigma} = \log_{10}\frac{\sqrt{N}}{\tau_\beta} + \log_{10}\frac{|\bar{r}|}{\sqrt{\frac{1}{N-1}\cdot\sum\limits_{i=1}^{N}(\hat{r}_i-\bar{r})^2}}$$

$$= \log_{10}\frac{\sqrt{N\cdot(N-1)}}{\tau_\beta} + \log_{10}\frac{|\bar{r}|}{\sqrt{\sum\limits_{i=1}^{N}(\hat{r}_i-\bar{r})^2}}$$

By defining $v := \log_{10}\frac{\sqrt{N\cdot(N-1)}}{\tau_\beta}$ and $\rho_i = \hat{r}_i - \bar{r}$, the equation can be rewritten as:

$$C_{\bar{r}} = v + \log_{10}\frac{|\bar{r}|}{\sqrt{\sum_{i=1}^{N}(\rho_i)^2}}$$

As lower precision floating point operators require less hardware resources and can operate at higher frequencies, it is logical to decrease the precision of the floating point operators whenever possible, provided that the final result is not seriously affected.

Let $\hat{Z} := \sum\limits_{i=1}^{N}(\rho_i)^2 \approx \sum\limits_{i=1}^{N}(\hat{\rho}_i \hat{*} \hat{\rho}_i)\Bigg|_{LP}$, where: $\hat{\rho}_i$ is the truncated value of $\rho_i$, $\hat{*}$ is the floating point multiplication and $\sum\Big|_{LP}$ is the pairwise summation – all in lower precision format. Moreover, recalling from Section 2.1 that any floating point number $F$ can be represented as $F = s\cdot M\cdot 2^E$, where $s$ is the sign, $M$ is the mantissa and $E$ is the exponent. The equation can be further simplified to:

$$\begin{aligned} C_{\bar{r}} &\approx v + \log_{10}\frac{|\bar{r}|}{\sqrt{\hat{Z}}} = v + \log_{10}\frac{M_{|\bar{r}|}\cdot 2^{E_{|\bar{r}|}}}{\sqrt{M_{\hat{Z}}\cdot 2^{E_{\hat{Z}}}}} \\ &= v + \left(E_{|\bar{r}|} - \frac{E_{\hat{Z}}}{2}\right)\cdot\log_{10}2 + \log_{10}\frac{M_{|\bar{r}|}}{\sqrt{M_{\hat{Z}}}} \end{aligned}$$

In order to avoid the costly division and logarithm calculations, the term $\log_{10}\frac{M_{|\bar{r}|}}{\sqrt{M_{\hat{Z}}}}$ is approximated to $\log_{10}\frac{\hat{M}_{|\bar{r}|}}{\sqrt{\hat{M}_{\hat{Z}}}}$, where: $\hat{M}_{|\bar{r}|}$ is assigned the first *three* most significant bits (MSBs) of $M_{|\bar{r}|}$ and $\hat{M}_{\hat{Z}}$ is assigned the first *two* MSBs of $M_{\hat{Z}}$.

$$\hat{C}_{\bar{r}} := v + \left(E_{|\bar{r}|} - \frac{E_{\hat{Z}}}{2}\right)\cdot\log_{10}2 + \log_{10}\frac{\hat{M}_{|\bar{r}|}}{\sqrt{\hat{M}_{\hat{Z}}}}$$

This equation gives the significant number of digits in base-10. As a final simplification, we divide the two sides of the equation by $\log_{10}2$ to avoid the floating point multiplication ($\times\log_{10}2$). This gives the value of $\hat{C}_{\bar{r}}$ in base-2. With $(\frac{\hat{C}_{\bar{r}}}{\log_{10}2})$ denoted as $[\hat{C}_{\bar{r}}]_2$ and $\frac{v}{\log_{10}2}$ denoted as $v_2$, the final equation turns to be:

$$\begin{aligned} [\hat{C}_{\bar{r}}]_2 &= v_2 + \left(E_{|\bar{r}|} - \frac{E_{\hat{Z}}}{2}\right) + \log_2\frac{\hat{M}_{|\bar{r}|}}{\sqrt{\hat{M}_{\hat{Z}}}} \\ &= \left(E_{|\bar{r}|} - \frac{E_{\hat{Z}}}{2}\right) + f(\hat{M}_{|\bar{r}|}, \hat{M}_{\hat{Z}}) \end{aligned}$$

Let the number of bits in the mantissa and exponent of the reduced precision format be denoted as $n_M$ and $n_E$ respectively. The design parameters used in this thesis and the constants, which are calculated accordingly, are listed in Table 4.2.

| Design parameters | | | | Constants | | |
|---|---|---|---|---|---|---|
| $N$ | $\beta$ | $n_M$ | $n_E$ | $\tau_\beta$ | $v$ | $v_2$ |
| 2 | 95% | 4 | 11 | 12.71 | −0.9532 | −3.1666 |

**Table 4.2:** Parameters and pre-calculated constants used in the simplified approach

$f(\hat{M}_{|\bar{r}|}, \hat{M}_{\hat{Z}})$ can be implemented in hardware as a small look-up table with $2^{(3+2)}$ (32) entry. The contents of this look-up table are provided in Appendix A.1.

From this equation, the data-path shown in Figure 4.1 can be derived for $N = 2^m$ samples. In order to calculate the average, $m$ is subtracted from the exponent of the sum of the samples. 1023 is subtracted from the exponents in order to get rid of the bias before using the exponents in the fixed point operations. The "shift right" block is used to evaluate $\frac{E_{\hat{Z}}}{2}$. Shifting maintains the MSB unchanged in order to keep the sign of $E_{\hat{Z}}$ which might be negative after removing the bias. The blue dashed lines and operators show how the data path can be extended when $m > 1$. If $m \notin \mathbb{Z}$, then an FP division operator, to calculate $\frac{1}{N}\sum_{i=1}^{N}\hat{r}_i$, is required to replace the block "exponent - $m$" in the figure. Further in this thesis, we refer the data path provided in Figure 4.1 as "the generic data path".

SBEU decides if the average of the samples is of sufficient accuracy ($[\hat{C}_{\bar{r}}]_2 > C_{TH}$) and if it is an informational zero ($[\hat{C}_{\bar{r}}]_2 < 0$) according to the conditions stated in Subsection 2.2.2.

The effect of using the lower precision and the look-up table is proved to guarantee that [Li10]: $|\hat{C}_{\bar{r}} - C_{\bar{r}}| < 0.1$. This is indeed a good approximation bearing in mind the high reduction in terms of hardware resources and latencies.

### 4.1.2 Simplified Data Path

For $N = 2$, a further simplification for the data-path provided in Figure 4.1 can be done. Recalling that $\sigma = \sqrt{\sum_{i=1}^{2}(\hat{r}_i - \bar{r})^2}$ and $[C_{\bar{r}}]_2 = \log_2\left(\frac{\sqrt{2}\cdot|\bar{r}|}{\tau_\beta\cdot\sigma}\right)$, and that $(\hat{r}_1 - \bar{r})^2 = (\hat{r}_2 - \bar{r})^2$, we can rewrite $[C_{\bar{r}}]_2$ in this form:

$$[C_{\bar{r}}]_2 = \log_2\left(\frac{|\bar{r}|}{\tau_\beta \cdot |(\hat{r}_1 - \bar{r})|}\right) = \log_2\left(\frac{|(\hat{r}_1 + \hat{r}_2)|}{\tau_\beta \cdot |(\hat{r}_1 - \hat{r}_2)|}\right) \quad .$$

By defining $|d| = |(\hat{r}_1 - \hat{r}_2)|$ and $|s| = |(\hat{r}_1 + \hat{r}_2)|$, we have:

$$[C_{\bar{r}}]_2 = \log_2\left(\frac{1}{\tau_\beta}\right) + \log_2\left(\frac{M_{|s|}}{M_{|d|}}\right) + (E_{|s|} - E_{|d|}) \quad .$$

Let $\hat{M}_{|s|}$ be the first *three* MSBs of $M_{|s|}$ and $\hat{M}_{|d|}$ be the first *two* MSBs of $M_{|d|}$, we have:

$$[\hat{C}_{\bar{r}}]_2 = (E_{|s|} - E_{|d|}) + w(\hat{M}_{|s|}, \hat{M}_{|d|}) \quad ,$$

where $w(\hat{M}_{|s|}, \hat{M}_{|d|}) = \log_2\left(\frac{1}{\tau_\beta}\right) + \log_2\left(\frac{M_{|s|}}{M_{|d|}}\right)$. $w$ is calculated based on the different $2^{(2+3)} = 32$ combinations of $\hat{M}_{|s|}$ and $\hat{M}_{|d|}$, and the parameters provided in Table 4.2. The contents of this look-up table is provided in Appendix A.1

The data path for 2 samples is shown in Figure 4.2. Further in this thesis, we refer to this data path as the "simplified data path".

The simplified data path exhibits better approximation than the generic data path in many cases due to the fact that there is no truncation error in the former.

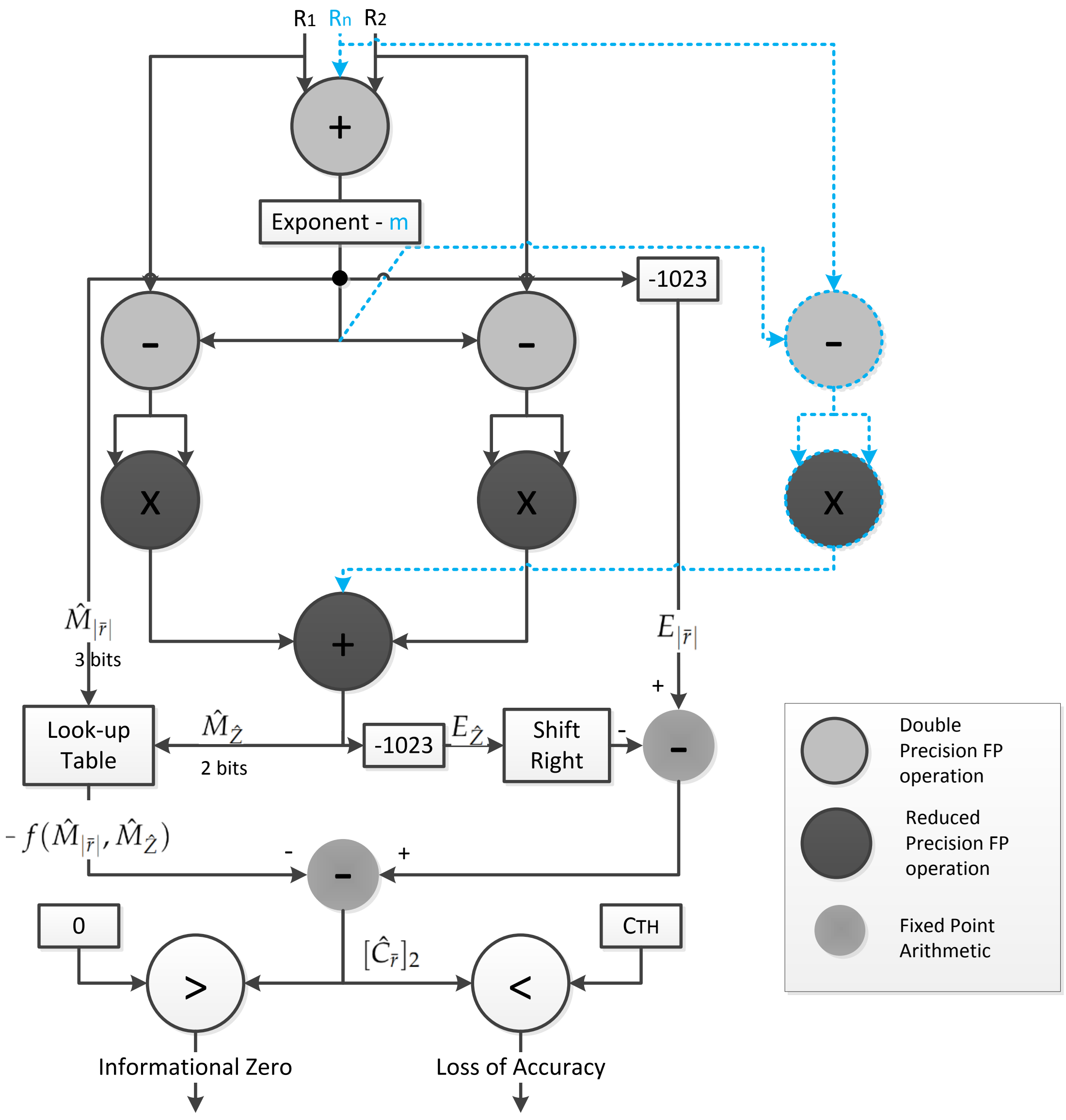


**Figure 4.1:** SBEU data-path (with $N = 2^m$ samples).

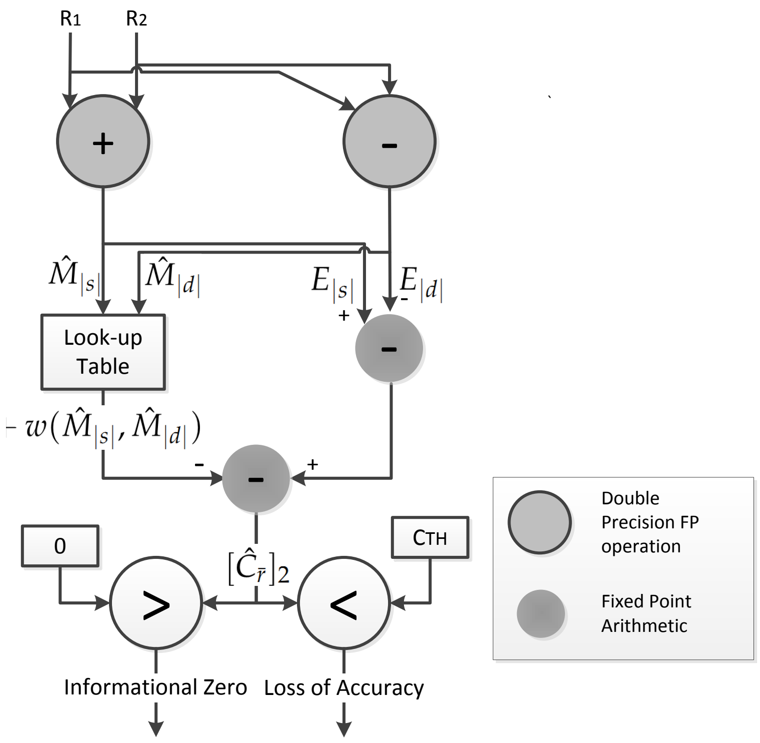


**Figure 4.2:** SBEU data-path (optimized for two samples).

### 4.1.3 Synthesis and Performance Metrics

The two data-paths were implemented in a fully pipelined manner. The target of the implementation in terms of efficiency can be described by these criteria:

$$\min\left(\frac{L}{\lfloor\frac{f_{1m}}{f_{2o}}\rfloor}\right)$$

$$f_{1o} = n \cdot f_{2o}, \quad n \in \mathbb{Z}^+,$$

where: $L$ is the overall latency in (clock cycles), $f_{1m}$ is the maximum operating frequency of SBEU obtained in synthesis, $f_{2o}$ is the actual operating frequency of STFPU and $f_{1o}$ is the actual operating frequency of SBEU. All frequencies are in MHz.

The first criterion is set such that the minimum latency, in terms of elapsed time, of the data path is obtained. Having two clock domains in the design, the metastability problem might arise. This problem is easier to overcome when one clock domain frequency is integer multiples of the other. Hence, the second criterion is introduced. This also explains the floor function in the denominator of the first criterion.

Different latencies for the FP operators were tried out and the optimum values according to our criteria for the two approaches are summarized in Table 4.3.

| Operator Type | The generic data path (clock cycles) | The simplified data path (clock cycles) |
|---|---|---|
| Double precision FP Add/Sub | 4 | 4 |
| Reduced precision Mul | 2 | – |
| Reduced Precision Add | 4 | – |
| Fixed point part | 2 | 1 |
| **Overall latency** | 16 | 5 |

**Table 4.3:** Latencies of the different operators used in SBEU.

When the designs were synthesized on XC5VFX130T Virtex-5 FPGA with speed grade $-2$, the maximum frequencies summarized in Table 4.4 were obtained.

| The generic data path (Mhz) | The simplified data path (MHz) |
|---|---|
| 245 | 245 |

**Table 4.4:** Maximum operating frequency of SBEU.

### 4.1.4 Extendability

Assuming that the number of samples is $N = 2^m, m \in \mathbb{Z}^+$. Referring to Figure 4.1, we can see that the data path is readily extendable. In order to calculate the average of the samples, double precision adders are to be arranged in a tree-like structure. Then, the average is calculated by subtracting $m$ from the exponent of the result. This structure requires $N - 1$ adder arranged in $m$ levels. The same holds for the reduced precision addition. Table 4.5 generalizes the number of different resources utilized when the number of samples is $N$. 64 bit registers are used to buffer the samples and maintain pipeline, and 14 bit registers are used to buffer the first 3 bits of the mantissa and the 11 bits of the exponent of the mean of the samples and to maintain the pipeline.

Applying the same assumption and referring to Table 4.3, we describe the overall latency $L$ as a function of $m$ as follows:

$$L = 8(m + 1)$$

It is important to mention here that in order to improve the estimation by one significant digit we need to multiply $N$ by 100. It is not practical to increase the number of samples in such away, especially that a small number such as $N = 2$ or $N = 3$ gives a good estimation of the number of significant digits[Vig04].

| Operator Type | No. of resources |
|---|---|
| Double precision FP Add | $N-1$ |
| Double precision FP Sub | $N$ |
| Reduced FP Mul | $N$ |
| Reduced FP Add | $N-1$ |
| 64 bit register | $8 \times m$ |
| 14 bit register | $6+4 \times m$ |

**Table 4.5:** Resource utilization for $N$ samples .

### 4.1.5 Verification

A C program which emulates the behavior of the two designs on the bit level was developed in order to facilitate validating the functionality of SBEU. Final and intermediate values of the signals in the HDL design were compared to the corresponding variables obtained from the C program. The source code for this program is provided in Appendix A.2.

Figure 4.3 shows how SBEU responds when loaded with different samples. At 95 ns, SBEU is loaded with two samples of sufficient accuracy (i.e. $[\hat{C}_{\bar{r}}]_2 > C_{TH}$). Then, two samples that are of insufficient accuracy (i.e. $0 < [\hat{C}_{\bar{r}}]_2 \leq C_{TH}$) are loaded at 105 ns. At last, two samples with no accurate digits at all (i.e. $[\hat{C}_{\bar{r}}]_2 < 0$) are loaded at 115 ns. The number of significant bits in fixed point notation, $[\hat{C}_{\bar{r}}]_2$, for the respective samples is calculated at 145 ns, 155 ns and 165 ns. The fixed point is between the sixth and seventh bit from right. For instance "$[00528]_{16} = [10100.101000]_2$", which means 20.625 significant bits or equivalently $20.625 \times \log_{10} 2 = 6.209$ significant digits. Similarly, "$[0037C]_{16}$" means 13.928 significant bits (4.193 significant digits) and so on. In this simulation, $C_{TH} = 5$ significant digits.

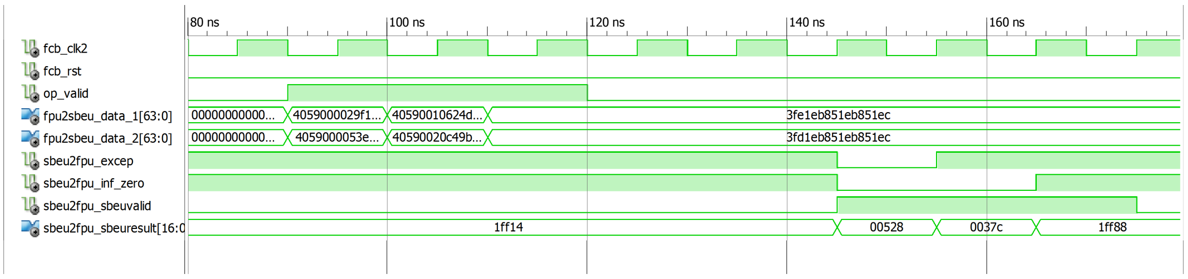


**Figure 4.3:** SBEU simulation.

## 4.2 Comparison Support Unit

This unit is designed to support stochastic relations according to the definitions provided in Subsection 2.2.3. The function of this unit can be described as follows:

- Calculating the differences between the samples. These differences are then loaded to the SBEU to be checked against informational zeros.
- Comparing the averages (or sum) of the samples and sending the result to the STFPUs.

The data path for the unit is shown in Figure 4.4. In this figure, $a_n$ and $b_n$ are the comparison operands sent from STFPU#$n$. The blue dashed lines and operators shows how the data path can be extended when $N > 2$.

The design is fully pipelined. The latency and maximum operating frequency are listed in Table 4.6.

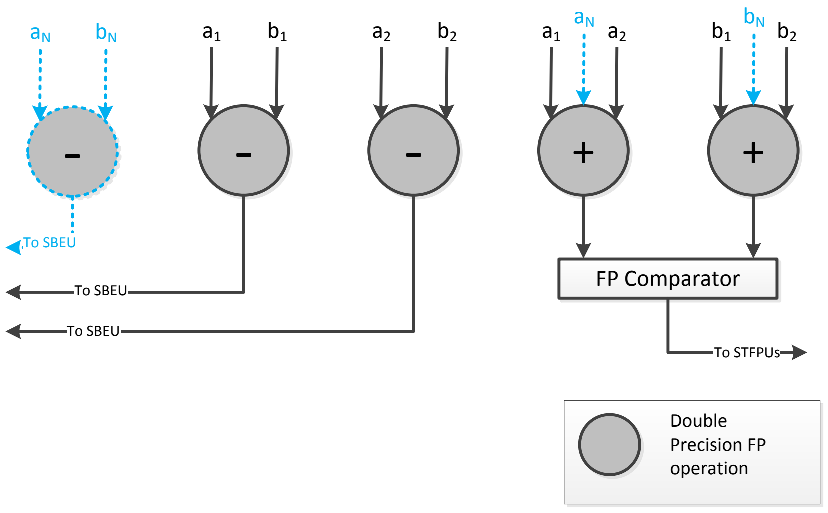


**Figure 4.4:** Comparison Support Unit data path.

Applying the same assumption made in Subsection 4.1.4, the comparison unit can be readily extended. Referring to Figure 4.4 and recalling that the latency of the utilized Add / Sub operators is 4 clock cycles, the number of resources needed can be generalized as in Table 4.7. Also the total latency of the unit can be expressed as:

$$L = 4 \times m + 1$$

| Latency (Clock cycles) | Frequency (MHz) |
|---|---|
| 5 | 245 |

**Table 4.6:** Performance of the comparison unit.

| Operator Type | No. of instances |
|---|---|
| FP Add | $2(N-1)$ |
| FP Sub | $N$ |
| FP comparator | 1 |

**Table 4.7:** Resource utilization for $N = 2^m$ STFPUs.

## 4.3 Synchronization Unit

When a floating point operation is started on one STFPU and not on the other, the synchronization unit issues a *stall* signal to the STFPU which has started the FP operation. This signal is asserted to maintain the targeted STFPU in the same state till the other STFPU catches up with the stalled one. Only then, the Synchronization Unit deasserts the stall signal. The basic building block of the Synchronization Unit is depicted in Figure 4.5.

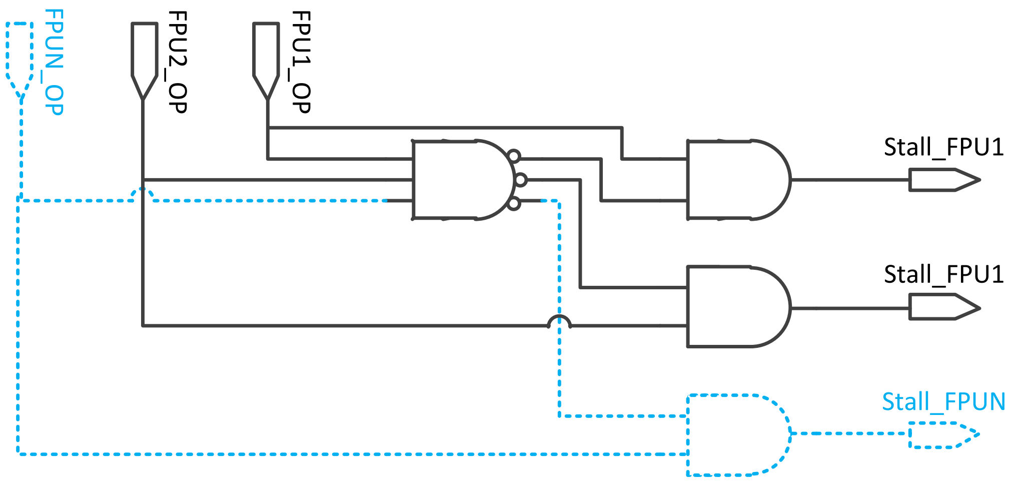


**Figure 4.5:** The building block of the Synchronization Unit.

The Synchronization Unit is composed of five of these building block to synchronize addition/subtraction, multiplication, division, comparison and square root operation on the two STFPUs. Then, the stall signals, from every building block, targeting an STFPU are logically ored.

## 4.4 NAU Controller and Syndrome Register

NAU is highly configurable. In order to support this, a state machine controller has been designed. The controller also writes information about the type of an exception to a special register (we call Syndrome Register) which can be read by the help of XMD. The Syndrome Register is connected to the PLB of one of the processors.

Different types of signals are communicated from the two STFPUs to the NAU. These signals are:

- Results of FP operations (Add / Subtract, Multiply, Divide, Square root, Comparison). These are being checked if they are of acceptable accuracy.
- Operands of multiplication and divisor are loaded to be checked against informational zeros.
- Operands of (addition / subtraction) are loaded to check if a *cancellation* is pointed out.
- Operands of Comparison.

Cancellation is the case when the minimum of the number of exact significant digits of the two operands is 3 digits [1] greater than the number of exact significant digits in the result, provided that both the operands and the result must be of acceptable accuracy ($[\hat{C}_{\bar{r}}]_2 > C_{TH}$)[2].

The state machine of the controller is shown in Figure 4.6. It is connected to the same clock used in the STFPUs. The transition from the "add" state to the "cancel" state occurs when $[\hat{C}_{\bar{r}}]_2$s for the operands are calculated. In the "cancel" state, $[\hat{C}_{\bar{r}}]_2$ for the result is subtracted from the minimum of the two $[\hat{C}_{\bar{r}}]_2$s. If the difference is greater than 3 digits, a cancellation exception is generated and relevant data is written to the Syndrome Register. In the figure, the transitions that generate exceptions (and writes data to the Syndrome Register) are colored in red for clarity.

Every group of the signals communicated from the STFPUs to the NAU has its own "VALID" signal. That makes it possible for the controller to distinguish between the types of data loaded to SBEU. Based on the context of the data, the controller may either generate exceptions depending on the detection of an informational zero or loss of accuracy. For instance, if the results of multiplication are loaded to SBEU, then the controller raises an exception to the STFPUs when it detects that the loss of accuracy signal is asserted. On the other hand, if the divisors are loaded to SBEU, a check for informational zero is carried out and an exception might be raised accordingly. In both cases, information about the type of exception is written to the Syndrome Register.

With the use of The Syndrome Register, five types of exceptions can be reported. The type of an exception might be:

[1]This value is passed as a generic to the NAU.

[2]If $[\hat{C}_{\bar{r}}]_2 < C_{TH}$ for the results, then a loss of accuracy exception will be generated.

1. Loss of accuracy: When the accuracy is less than a predefined number of digits.
2. Cancellation: When a sudden loss of accuracy in an Add / Sub FP operation is detected due to cancellation.
3. Insignificant divisor: When the divisor is informational zero.
4. Insignificant multiplication operand: When at least one of the multiplication operands is informational zero.
5. Branching instability: When informational zero is detected in the differences of operands of a comparison.

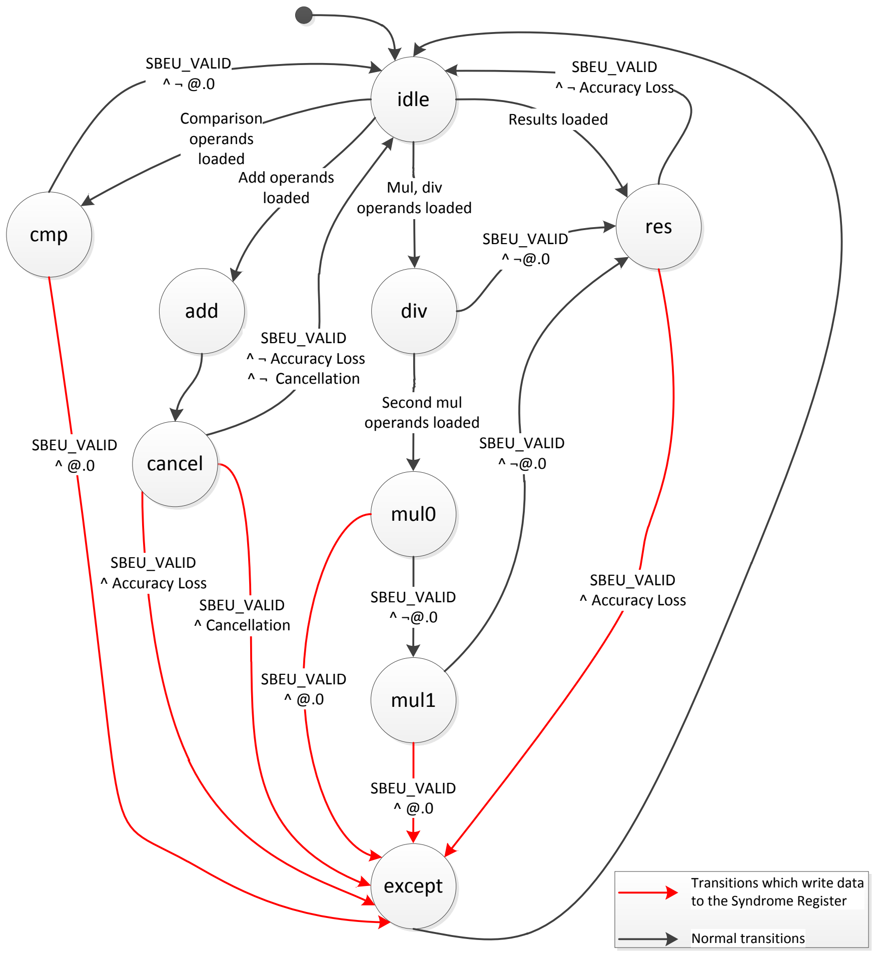

idle
cmp
add
div
res
cancel
mul0
mul1
except
SBEU_VALID ^ ¬ @.0
Comparison operands loaded
Add operands loaded
Mul, div operands loaded
SBEU_VALID ^ ¬ Accuracy Loss
Results loaded
SBEU_VALID ^ ¬@.0
SBEU_VALID ^ ¬ Accuracy Loss ^ ¬ Cancellation
Second mul operands loaded
SBEU_VALID ^ ¬@.0
SBEU_VALID ^ @.0
SBEU_VALID ^ Accuracy Loss
SBEU_VALID ^ Cancellation
SBEU_VALID ^ @.0
SBEU_VALID ^ ¬@.0
SBEU_VALID ^ @.0
SBEU_VALID ^ Accuracy Loss
Transitions which write data to the Syndrome Register
Normal transitions

**Figure 4.6:** NAU controller state machine.

## 4.5 Changes to The STFPUs

Some necessary changes have been carried out on the STFPUs. These changes can be put into three categories:

1. A data loading mechanism to communicate data to the NAU.
2. Necessary changes in order to support Stochastic Relations.
3. Catching and forwarding exceptions from NAU to the PowerPC.

In the following three subsections these changes are presented in more details.

### 4.5.1 Loading data to NAU

A multiplexer is added to the STFPU in order to load applicable data to NAU according to the instruction being executed. For instance, in a multiplication operation the operands of multiplication as well as the result is communicated to NAU one after the other. On the other hand, in a division operation only the divisor and the result are loaded to NAU. The data communicated from an STFPU to NAU is summarized in Table 4.8.

| Instruction | Add / Subtract, Multiply | Divide | Square root | Compare |
|---|---|---|---|---|
| Data loaded to NAU | Operands and result | Divisor and result | Result | Operands |

**Table 4.8:** The data loaded to NAU when different instructions are executed.

### 4.5.2 Support for Stochastic Relations

Instead of making decision for comparison operation in every STFPU, the operands are sent to NAU and a unified decision is made for all the STFPUs. As shown in Figure 4.4, the result of comparison based on the sums is saved in a register in the STFPU. The differences are calculated and loaded to SBEU so as to be checked against informational zero. Three bits of the signal `FCMAPUCR` are sent to the PowerPC as a result for comparison. They correspond to equal (==), greater than (>) and smaller than (<) respectively. The processor can check for "greater than or equal" (>=), or "smaller than or equal" (<=) by logically oring the two corresponding bits. Table 4.9 shows how these bits are set in order to comply to the definitions provided in Subsection 2.2.3.

| $X - Y$ | $\bar{X}?\bar{Y}$ | | | FCMAPUCR | | |
|---|---|---|---|---|---|---|
| | == | < | > | == | < | > |
| @.0 | X | X | X | 1 | 0 | 0 |
| ¬@.0 | 0 | 1 | 0 | 0 | 1 | 0 |
| ¬@.0 | 0 | 0 | 1 | 0 | 0 | 1 |

**Table 4.9:** Changes to STFPU to support stochastic relations.

### 4.5.3 Raising Exceptions to The PowerPC

#### Floating Point Exception Modes in The PowerPC

There are two bits in the **M**achine **S**tate **R**egister (MSR) which are used to set the floating point exception mode in the PowerPC [b0002, p. 39 - p. 41], namely the `FE0` and `FE1`. These two bits are sent to the STFPUs through the APU interface by the signals `APUFCMMSRFE0` and `APUFCMMSRFE1` respectively. Based on these two signals the STFPU will either run autonomously or non-autonomously (see Section 3.1 for more details). These modes are summarized in Table 4.10.

| MSR | | Mode | Description |
|---|---|---|---|
| `FE0` | `FE1` | | |
| 0 | 0 | Ignore Exceptions | In this mode the PowerPC does not check for floating point exceptions and the STFPU runs autonomously. |
| 1 | 0 | Imprecise Recoverable | In this mode an exception raised might be caught by the PowerPC at some point at or beyond the instruction which caused the exception. There is no means to identify the instruction which caused the exception in this mode. |
| 0 | 1 | Imprecise Recoverable | In our STFPU, this mode is equivalent to the imprecise recoverable mode. |
| 1 | 1 | Precise | This mode guarantees that any floating point exception will be taken exactly. The STFPU in this mode runs non-autonomously. |

**Table 4.10:** Floating point exception modes in the PowerPC.

We are more interested in the precise mode as it is the only means to raise exceptions which can be timely caught by the PowerPC. In order to raise exceptions correctly over the APU interface to the PowerPC, a strict sequence must be followed. After the instruction is sent over the APU interface and the STFPU is willing to generate an exception, then this sequence must occur:

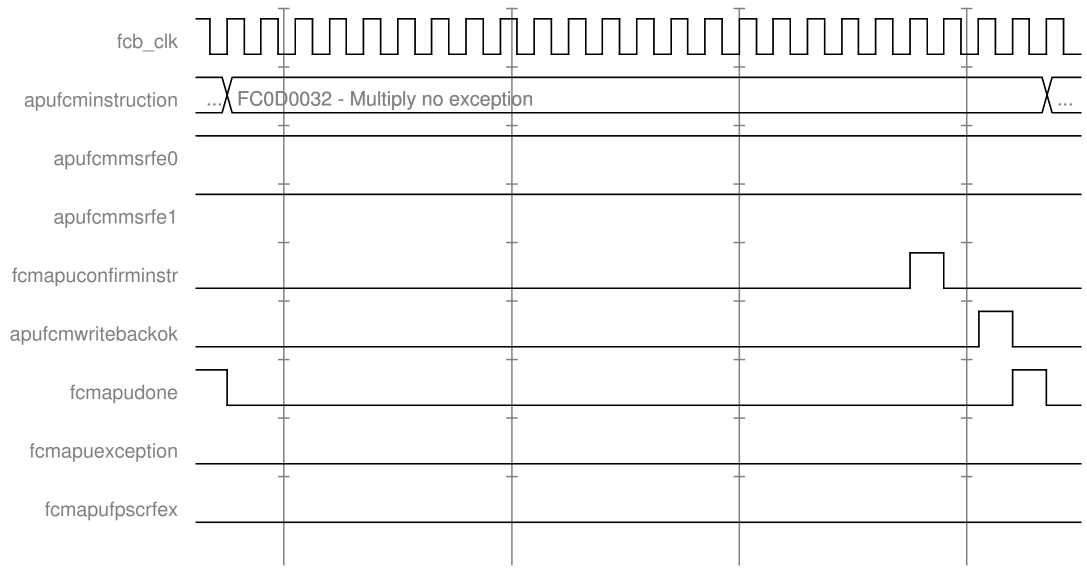


**Figure 4.7:** Normal non-autonomous multiplication operation with no exception.

1. Do not assert `FCMAPUCONFIRMINSTR` after receiving the instructions. i.e. do not confirm that the instruction will not generate exceptions.
2. `FCMAPUEXCEPTION` must be asserted and held.
3. `FCMAPUFPSCRFEX` must be asserted and held.

Then, the APU responds as follows:

1. Sets `APUFCMWRITEBACKOK = 1` if the exception was accepted or `APUFCMWRITEBACKOK = 0` if not.
2. Sets `APUFCMFLUSH = 0` if the exception was received, `APUFCMFLUSH = 1` if not.

To complete the transaction, the STFPUs responds with:

1. Asserts `FCMAPUDONE` and deasserts `FCMAPUEXCEPTION` and `FCMAPUFPSCRFEX` if `APUFCMWRITEBACKOK = 1` was received.
2. Deasserts `FCMAPUEXCEPTION` and `FCMAPUFPSCRFEX` if `APUFCMFLUSH = 1` was received. STFPU should not assert `FCMAPUDONE`.

To illustrate this, Figure 4.7 and Figure 4.8 are provided. Figure 4.7 shows a normal multiplication which does not generate an exception. Figure 4.8 shows a multiplication operation which generates an overflaw exception.

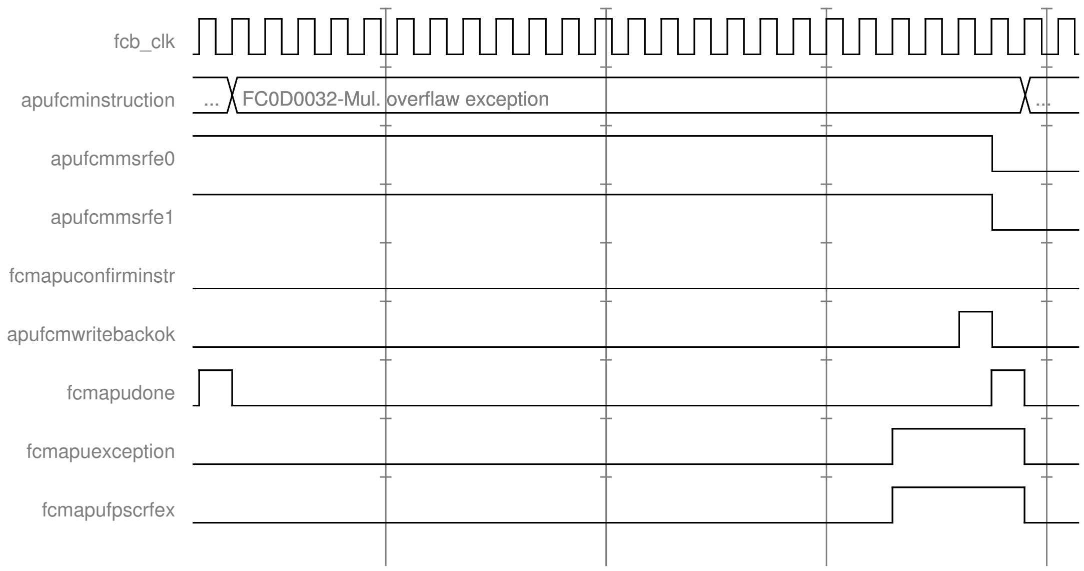


**Figure 4.8:** Non-autonomous multiplication operation which generates an overflaw exception.

Raising NAU Exceptions to The PowerPC

We have adopted a similar methodology to the one presented in Subsection 4.5.3 which is originally used to raise normal floating point exceptions (division over zero, overflaw, underflaw . . . ) to the PowerPC. This mechanism is described as follows:

1. When data is sent to the NAU, a flag is set in the STFPUs indicating that NAU is busy (`NAU_BUSY`).
2. While `NAU_BUSY` is set, the STFPUs cannot assert the `FCMAPUCONFIRMINSTR`.
3. When NAU is done with calculations it asserts the signal `NAU2FPU_NAUvalid` along with `NAU2FPU_EXCEP = 0` or `NAU2FPU_EXCEP = 1`. This will clear the `NAU_BUSY` flag.
    a) If `NAU2FPU_EXCEP = 0`, then `FCMAPUCONFIRMINSTR` is asserted and the STFPU continues normally.
    b) If `NAU2FPU_EXCEP = 1`, then the STFPUs assert and hold the `FCMAPUEXCEPTION` and `FCMAPUFPSCRFEX` signals while keeping `FCMAPUCONFIRMINSTR` deasserted. Then the APUs responds with `APUFCMWRITEBACKOK = 1`. Consequently, The STFPUs assert `FCMAPUDONE` and clear the signals `FCMAPUEXCEPTION` and `FCMAPUFPSCRFEX`.

Design Generics and Configurations

Here we summarize the different modes supported in our architecture. At first, we start with the automatic support (no need to rebuild the hardware) for the autonomous and non-autonomous mode of an STFPU. As we stated in Subsection 4.5.3, there are two signals responsible for setting the running mode of an STFPU, namely `APUFCMMSRFE0` and `APUFCMMSRFE1`. In our design, unless these two signals are both 1's, the STFPUs run as if there is no NAU connected to them. The importance of this mode is giving the possibility for users, who are not interested in localizing the sources of numerical instabilities in their codes, to gain the speed up from running the code autonomously while still being informed concerning the numerical accuracy of the final results.

There are several configurations supported by our architecture. Table 4.11 presents these configurations. For instance, we can easily disable the check for multiplication operands and disable the exceptions upon the detection of an informational zero in comparison by changing the applicable generics.

## 4.6 System Integration

### 4.6.1 Detailed System Architecture and Performance Metrics

We have integrated the different components of the system presented in the previous sections of this chapter. The underlying platform is the ML510 evaluation board from Xilinx. Table 4.12 shows the resource utilization of the NAU when considering the two designs for SBEU (both the generic and simplified ones). Comparing this table to Table 3.1, we can see that NAU, considering the simplified data path, consumes about 55% as much as an STFPU in terms of slice registers and about 73% as much as an STFPU in terms of slice LUTs.

All the timing constraints set by Xilinx tools have been met successfully. The latencies for different FP operations after integrating NAU in the system, considering the two data paths of SBEU, are provided in Table 4.13. Comparing Table 3.2 to Table 4.13, we can see that the latency of the FP operation is not affected at all after integrating NAU in the system.

Figure 4.9 shows the detailed and final system architecture. It shows how different components interact with each others, with the main changes done to the STFPU highlighted for clarity.

### 4.6.2 Testing and Simulation

The components have been thoroughly tested separately by means of testbenches. In addition, After the integration of all the components in the system, a complete system simulation has been conducted using Modelsim. We have created many test scenarios to validate the correctness of the implementation. Finally, the hardware was downloaded on the FPGA and tested by executing programs and benchmarks. One of these test cases is shown in

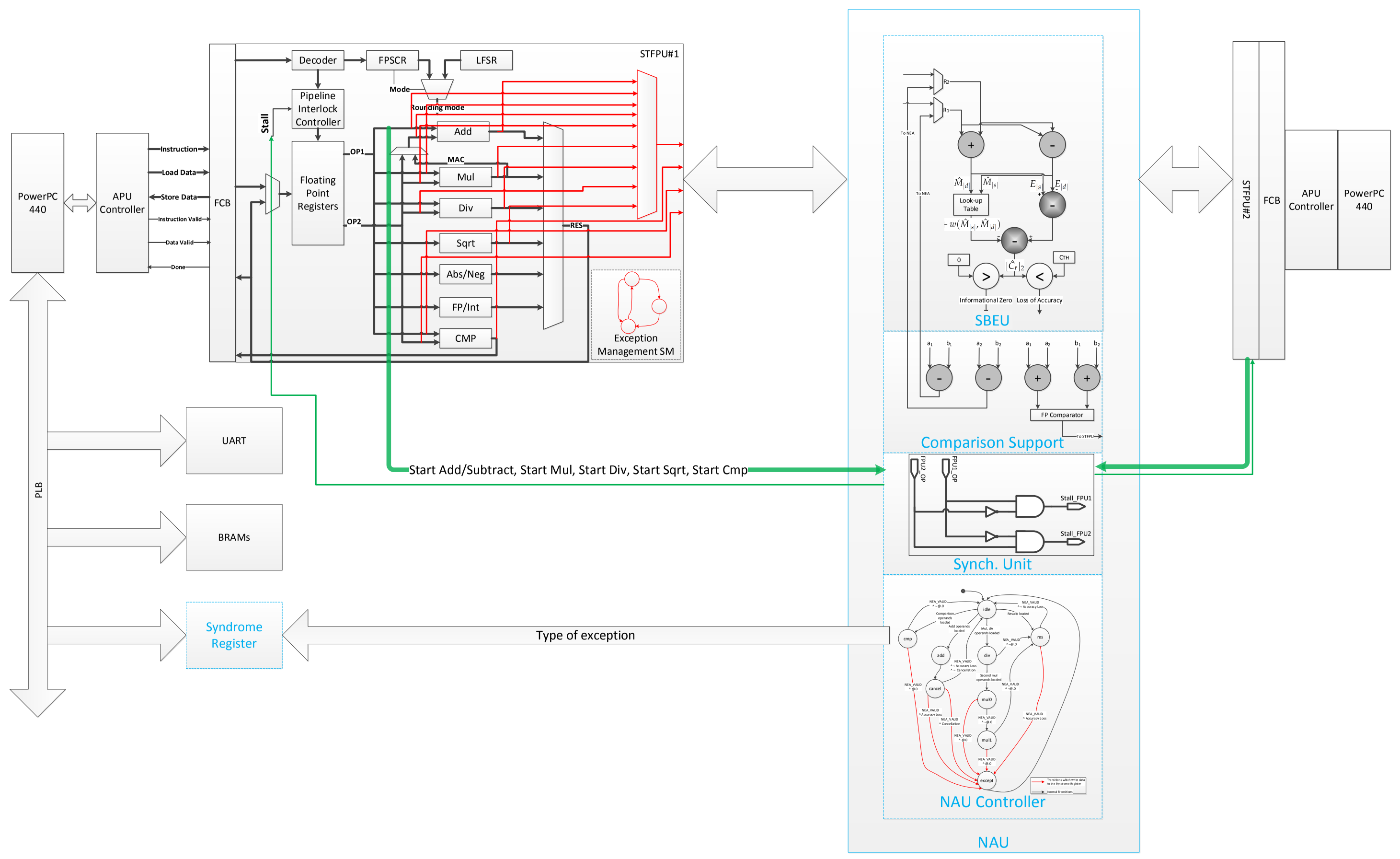


**Figure 4.9:** Detailed system architecture.

Figure 4.10 where an exception is generated due to the loss of accuracy of FP addition results. The figure is annotated for clarity. `EXCEP_CLASS` signal writes the type of exception to The Syndrome Register.

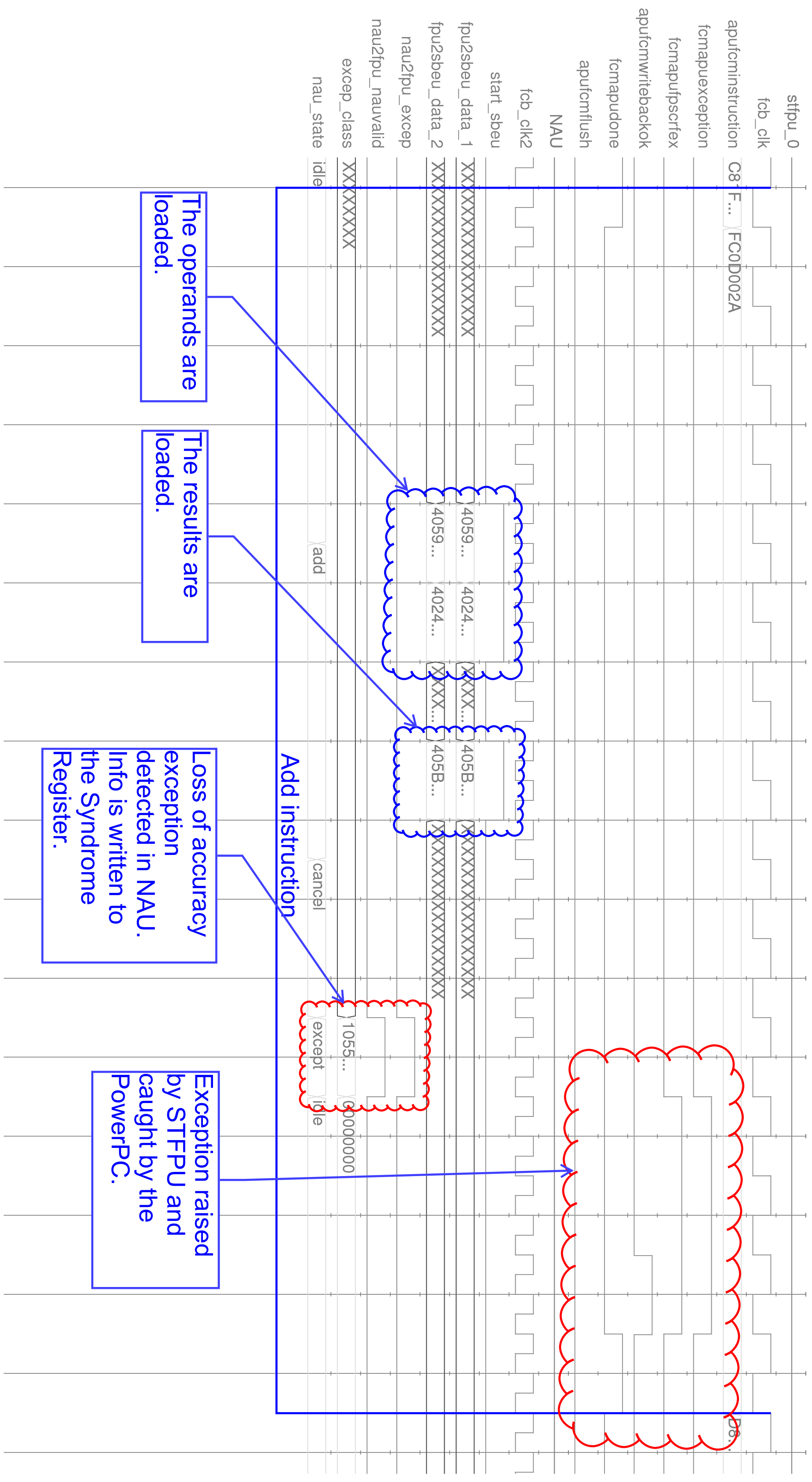


**Figure 4.10:** Loss of accuracy exception generated by NAU and caught by the PowerPC.

| Unit | Generic | Value | Description |
|---|---|---|---|
| STFPU | CMP_MODE | 0 | The results of comparison are sent to NAU. An exception is generated if results sent from the two STFPUs are not the same. |
| | | 1 | The operands of comparison are sent to NAU. This mode gives support for stochastic relations. |
| NAU | CMP_MODE | 0 | Support stochastic relations. No exceptions upon the detection of an informational zero are sent to the STFPUs. |
| | | 1 | Support stochastic relations. Exceptions are sent to STFPUs upon the detection of an informational zero. |
| | OPERANDS_MODE | 0 | No Checks for the significance of the multiplication operands and divisor are done. |
| | | 1 | Exceptions are sent if the multiplication operands or the divisor are informational zeros (The first condition in Subsection 2.2.2 is not applied). |
| | NO_ACCURATE_DIGITS | $00000010000100111_2$ | Specify the minimum acceptable accuracy (in base-2) in fixed point notation ($C_{TH}$ in Figure 4.2). The fixed point is between the sixth and seventh bit from right. |
| | CANCEL_LEVEL | $00000001010000000_2$ | Specify the cancellation level (in base-2) in fixed point notation. The fixed point is between the sixth and seventh bit from right. |
| | CANCEL_CHECK | 0 | Generate exception when cancellation in Add / Sub occurs. |
| | | 1 | No check for cancellation. |

**Table 4.11:** The generics used in the architecture.

| SBEU data path | Description | Value |
|---|---|---|
| The simplified data path | Number of Slice Registers | 3541 |
| | Number of Slice LUTs | 6418 |
| | Maximum operating frequency | 245 MHz |
| The generic data path | Number of Slice Registers | 4450 |
| | Number of Slice LUTs | 8014 |
| | Maximum operating frequency | 220 MHz |

**Table 4.12:** Hardware utilization and maximum operating frequency for NAU considering the two data paths provided in Figure 4.1 and 4.2.

| SBEU | Operation | Latency (Clock cycles) |
|---|---|---|
| The generic data path | Add/Subtract | 23 |
| | Multiply | 28 |
| | Divide | 55 |
| | Multiply-Accumulate | 29 |
| The simplified data path | Add/Subtract | 23 |
| | Multiply | 24 |
| | Divide | 50 |
| | Multiply-Accumulate | 29 |

**Table 4.13:** Latencies of the floating point operators in STFPU (precise mode) after integrating NAU.

# 5 Software and Debugging Support

In this chapter we give a brief introduction about interrupts management and classification in the first two Sections (5.1, 5.2) with focus on the type of exceptions generated by NAU. Then, we introduce the support provided by the Standalone BSP on the software level to manage exceptions in Section 5.3. In Section 5.4, we explain how to localize the source of exception and how to get more info about the type of exception using XMD and gdb.

## 5.1 Interrupt Classification in The PowerPC

An interrupt is the action in which the processor saves its state and changes the normal execution of the program to a predefined interrupt-handler address. Exceptions are the events that cause the processor to take an interrupt.

Interrupts in the PowerPC can be categorized based on the dependency on the instruction execution into two categories [Mil, p. 127 - p. 128]:

1. **Asynchronous Interrupts:** These Interrupts are caused by events that have nothing to do with the instruction being executed (such as external interrupts).
2. **Synchronous Interrupts:** They are caused by the instruction being executed. They can further be classified into two classes.
    a) Precise: These interrupts indicate exactly the instruction which caused the exception.
    b) Imprecise: They might indicate the instruction which caused the exception or some instruction that comes after that one.

Based on their priority, the interrupts can also be classified as [Cli01, p. 8]:

1. **Critical:** Critical interrupts have a higher priority (such as debug events and some external interrupts).
2. **Non-Critical:** They are of lower priority. If a non-critical interrupt happens at the same time with a critical interrupt, the processor will service the critical one first. Interrupts that are caused by instruction execution or timers are considered non-critical.

Based on this description, we classify the interrupts caused by NAU generated exceptions as *Synchronous (Precise) Non-critical Interrupts* or equivalently *Program Interrupts*. A set of registers are used by the processor to manage exception handling upon processing such an interrupt. These register are [boo02, p. 144 - p. 147]:

- Machine State Register (MSR): Defines the state of the processor. It has bits related to enabling and disabling interrupts.
- Save/Restore Register 0 (SRR0): Used to save the instruction which caused the interrupt.
- Save/Restore Register 1 (SRR1): Used to save MSR before entering the interrupt handler and restore it upon returning from the exception handler (executing `rfi`).
- Exception Syndrome Register (ESR): Used to distinguish between the different kinds of exceptions that can generate the same interrupt.
- Interrupt Vector Prefix Register (IPVR): Specifies the high-order 48 bits of the address of the exception handler.
- Interrupt Vector Offset Registers (IVOR0 to IVOR15): Every register of them is used to provide the low-order 12 bits (the remaining 4 bits are affixed to 0s) of the address of the exception handler depending on the interrupt type.

## 5.2 Program Interrupt Processing

Processing NAU interrupts , which are Program Interrupts, is preformed in five consecutive steps as follows:

1. SRR0 is loaded with the instruction that caused the exception.
2. ESR is loaded with more information about the exception type. This information is concerning the built-in exceptions supported by the PowerPC and does not know any thing about NAU and the numerical instability exceptions discussed earlier and that is why we introduced our own Syndrome Register.
3. SRR1 is loaded with a copy of MSR value.
4. Bits FP, FE0, FE1 are set to 0. FP stands for floating point and when this bit is set to zero, the processor cannot execute floating point instructions.[1]
5. Execution of the exception handler starts with the new updated MSR. The address of the exception handler is calculated by means of IVPR and IVOR6, where IVOR6 is the offset for a "Program Interrupt".
6. Upon reaching the end of the exception handler and calling `rfi`, MSR restores its old value from SPR1 and the processor resumes execution at the address saved in SPR0.

The processor executes the instruction which caused the exception twice. At the first time it assert the `APUFCMWRITEBACKOK` to inform the STFPU that the exception is accepted. At the second time it asserts `APUFCMFLUSH` to inform the STFPU that a previous instruction was received.

[1]Some other bits are updated as well, but they are irrelevant to our work.

## 5.3 Exception Management on The Software Level

In our System, there is no operating system. However, A minimal operating system functionality is provided by the Standalone **B**oard **S**upport **P**ackage (BSP). The Standalone BSP library `libxil.a` contains the boot code, memory management and most important for us is the support for managing exceptions [Xil].

Three functions are used to manage exceptions and they are declared in the `xexception_l.h`:

- `void Xexc_Init(void)`: This function sets up the *interrupts vector table* and registers a function which does nothing for every exception. The interrupt vector table can be described as a table which contains the addresses of exception handlers for every exception type.
- `void XExc_RegisterHandler(Xuint8 ExceptionId, XExceptionHandler Handler, void *DataPtr)`: This function registers a handler for a specific exception rather than the initial do-nothing one. `ExceptionId` is set to `XEXC_ID_PROGRAM_INT`[2] which is used to address NAU exceptions, `XExceptionHandler` is a pointer to the exception handler and `DataPtr` is a pointer to some data structure that can be passed to the exception handler.
- `void XExc_mEnableExceptions (EnableMask)`: This function is used to enable critical or non-critical or both exceptions. Since NAU exceptions are non-critical, the `EnableMask` is set to `XEXC_NON_CRITICAL`[2].

## 5.4 Localizing and Getting More Info About The NAU Exceptions

XMD and powerpc-eabi-gdb are used to localize the source of NAU exceptions. Getting more details about an exception is possible by accessing the Syndrome Register.

Figure 5.1 shows the interactions between different components when an exception is generated in the NAU. When a numerical instability is detected in NAU the following actions take place:

1. The exception is sent from NAU to the STFPUs.
2. The STFPUs will assert and deassert applicable signals in order to properly communicate that exception to the processor through the APUs (see Subsection 4.5.3).
3. The exception is caught by XMD, which stops exactly at the instruction that had caused the exception. Reading the Syndrome Register will give more information about the exception.

[2]Predefined Macro

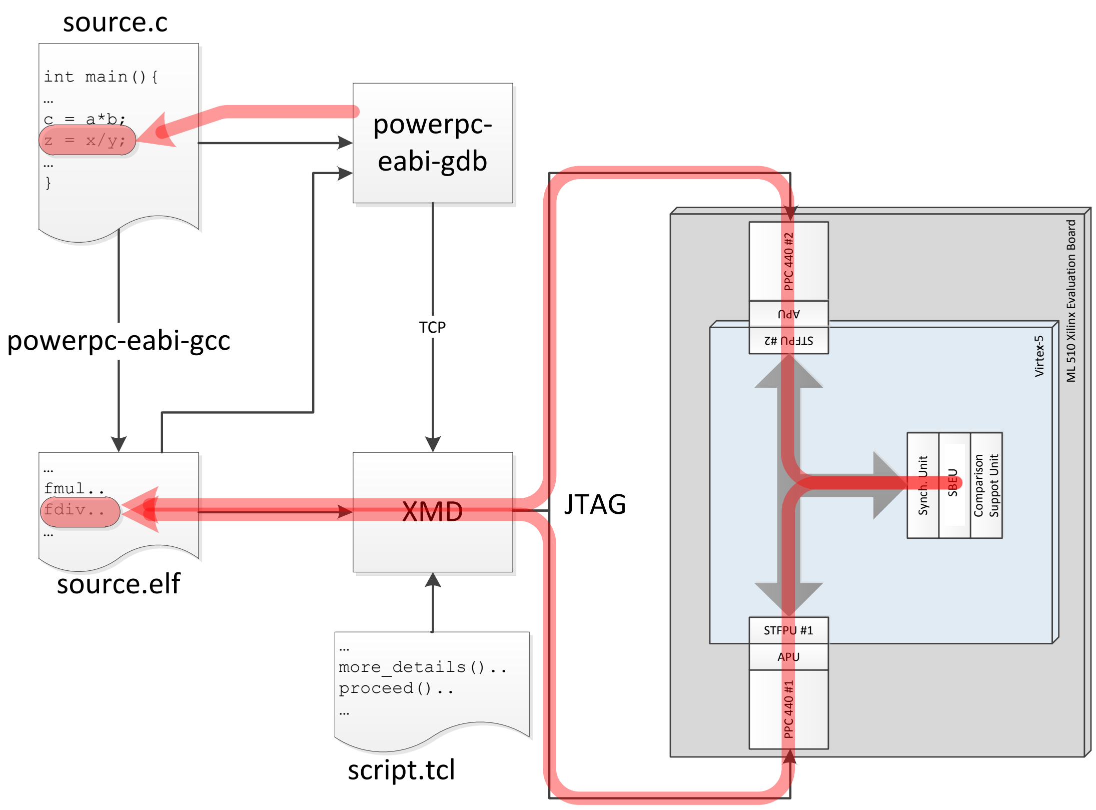


**Figure 5.1:** Catching a NAU exception.

4. By connecting gdb to XMD through a TCP connection, one can also localize the source of exception on the C statement level.

### 5.4.1 XMD

XMD has some built-in features that make it possible to catch the exceptions. It can establish connections with the two processors on the board in one session. Switching between the processors is straightforward. It can be used also as a normal debugger to set breakpoints, watchpoints, read the values of registers and change them, and read memory locations. It can also be configured to catch exceptions upon their occurrence. One of the most important features of XMD is that it accepts **T**ool **C**ommand **L**anguage (TCL) scripts. This allows more functionality to be added.

First Approach: Localize the source of exception and abort

In order to catch the NAU exceptions, a TCL script (provided in Appendix A.4) is loaded to XMD which does the following:

1. Downloads the executables to the BRAMS (or SDRAMS) of the correspondent PowerPCs.
2. Sets the MSR register, in the two processors, value to 0x2900, i.e. setting the bits of FP, FE0 and FE1 to ones, respectively. That means let the STFPUs run non-autonomously.
3. Enables the safemode of XMD. Only with this mode XMD traps (stops at) the exceptions upon their occurrence.
4. Instructs the processors to start the execution of the code.

If a NAU exception is generated, it will be caught and the memory address of source of the exception will be printed on the screen. In order to get more details about the exceptions, a call to the function `more_details` in the TCL script will read the value of the Syndrome Register. Based on that value, the type of exception will be printed on the screen.

Using powerpc-eabi-objdump it is possible to disassemble the executable file. A text editor might be used to search for the address of the assembly instruction that caused the exception.

In this approach the original source is kept as-is. Hence, there is no support for exception handling. XMD is only able to localize the source of exception and after that it will not be possible to resume the execution of the program normally.

Note that if we want to run the program autonomously we have simply to change the value set to the processors in step 1 to 0x2000.

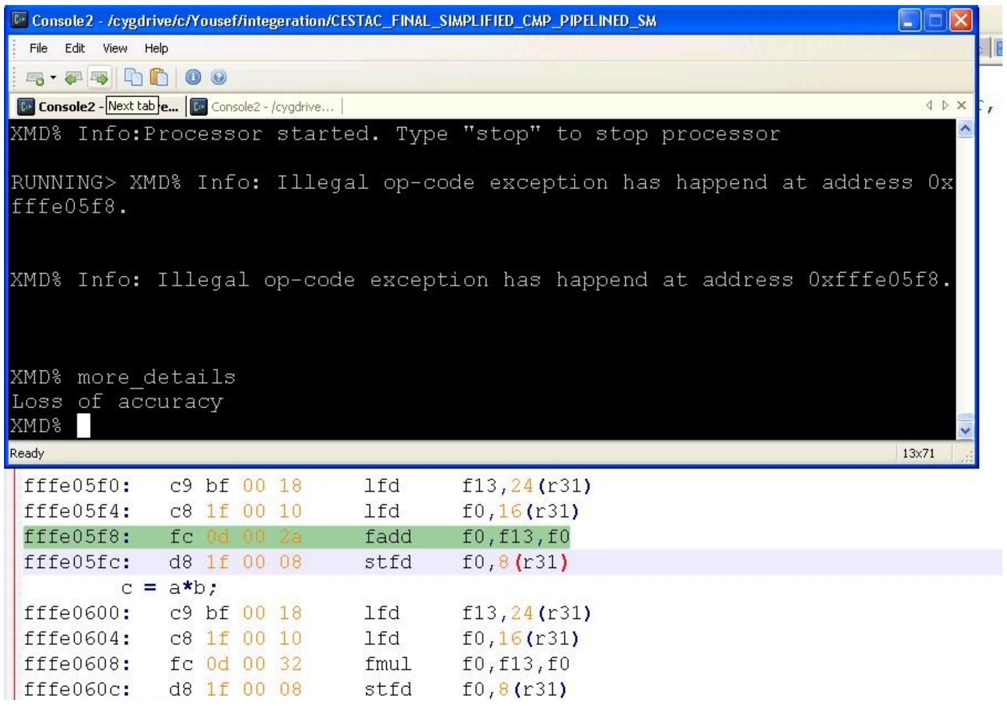
Figure 5.2 shows XMD localizing the source of a NAU exception and giving more details about the type of the exception. The disassembled file in the background is opened in a text editor and the source of exception instruction is highlighted in green.

Second Approach: Localize the source of exception and resume

In order to support recovering from the exception and resuming the execution of the code minor changes to the original source code are required. These changes are:

- Include the `xexception_l.h` header file.
- Upon entering main, make a call to `Xexc_Init` in order to initialize the exception vector table.[3]

[3]It is necessary to add `.vector` section to the linker script.


Console2 - /cygdrive/c/Yousef/integeration/CESTAC_FINAL_SIMPLIFIED_CMP_PIPELINED_SM
File Edit View Help
Console2 - Next tab e...
Console2 - /cygdrive...
Ready
13x71

```
XMD% Info:Processor started. Type "stop" to stop processor

RUNNING> XMD% Info: Illegal op-code exception has happend at address 0x
fffe05f8.


XMD% Info: Illegal op-code exception has happend at address 0xfffe05f8.



XMD% more_details
Loss of accuracy
XMD%
```

```
fffe05f0:   c9 bf 00 18    lfd     f13,24(r31)
fffe05f4:   c8 1f 00 10    lfd     f0,16(r31)
fffe05f8:   fc 0d 00 2a    fadd    f0,f13,f0
fffe05fc:   d8 1f 00 08    stfd    f0,8(r31)
        c = a*b;
fffe0600:   c9 bf 00 18    lfd     f13,24(r31)
fffe0604:   c8 1f 00 10    lfd     f0,16(r31)
fffe0608:   fc 0d 00 32    fmul    f0,f13,f0
fffe060c:   d8 1f 00 08    stfd    f0,8(r31)
```

**Figure 5.2:** XMD localizing a NAU exception.

With these changes, the same TCL script is loaded to XMD. When an exception occurs, XMD localizes the source of the exception. Afterwards, an attempt to resume the execution of the program will result in calling the do-nothing exception handler. After calling `rfi`, the execution resumes at the instruction which caused the exception (see Section 5.2). The processor is supposed to execute the same instruction once again, but this time it should *flush* it. However, this is not the case for some unknown reason. The processor executes the instruction and XMD catches the exception again and then the whole process repeats over and over.

Fortunately, due to the automatic support of our architecture of attaching and detaching NAU, and consequently enabling / disabling its exceptions, by detecting the signals APUFCMMSRFE0 and APUFCMMSRFE1 which reflect the values of the bits FE0 and FE1 in MSR, a workaround the problem is possible.

A new function is added to the TCL script, namely `proceed`. This function is to be called upon the willingness of resuming execution after catching an exception. A call to this function does the following actions:

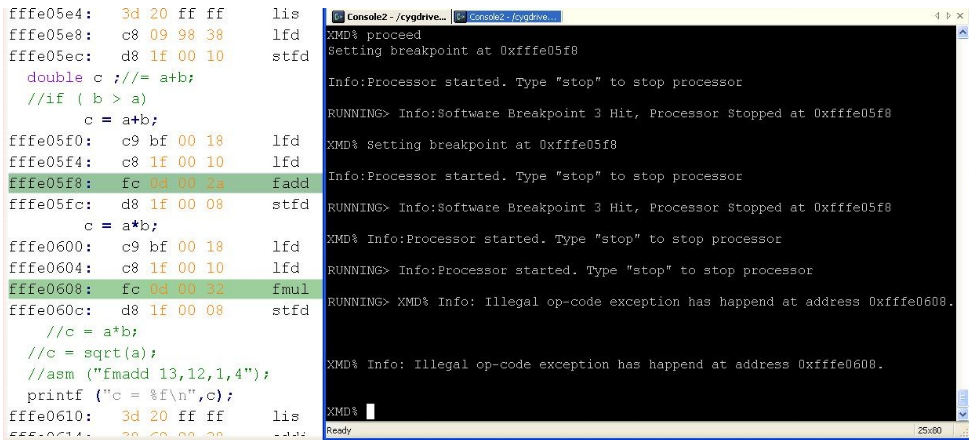
1. Reads the Program Counter Register and sets a breakpoint on the two processors based on that address. This value is actually the address of the instruction which caused the exception.
2. When the breakpoints are reached, sets the two MSRs to 0x2000, i.e. let the STFPUs run autonomously (detach NAU).
3. Instructs the processors to step one instruction.
4. Reloads MSRs again with 0x2900, i.e. let the STFPUs run non-autonomously (reattach NAU).
5. Instructs the processors to continue normally.

It is worthwhile mentioning that it is also possible to write a small exception handler which dereferences the Syndrome Register and accordingly prints on stdout the type of exception being detected. This is possible as shown in listing 5.1

```c
#include <stdio.h>
#include <xexception_l.h>
/* The exception handler code goes in this function */
void Handler_fun(void *DataPtr) {
  /* The address of the Syndrome Register is 0xc9200000 */
  printf ("Syndrome_Register = %x\n",*(long int *)0xc9200000);
}

int main()
{
  /* Declare a pointer to a function */
  void (*Handler_ptr)(void *DataPtr);
  /* Let the pointer point to the handler function */
  Handler_ptr = Handler_fun;
  /* Initialize the Interrupt vector table */
  XExc_Init();
  /* Register Handler_fun to be called upon the detection of a program exception */
  XExc_RegisterHandler(XEXC_ID_PROGRAM_INT, (XExceptionHandler) Handler_ptr, NULL);
  /* Enable NON_CRITICAL exceptions to be trapped */
  XExc_mEnableExceptions(XEXC_NON_CRITICAL);
  /*
       The code of the original program goes here.
  */
}
```

**Listing 5.1:** Exception handler to dereference the Syndrome Register.

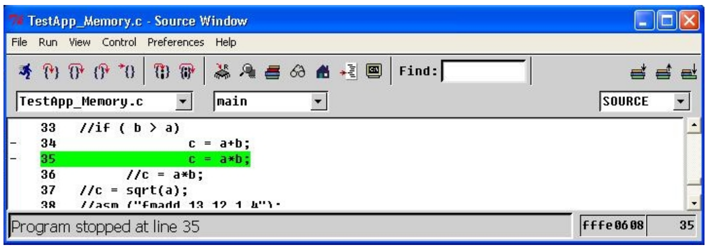
Figure 5.3 shows XMD catching two consequent NAU exceptions. At first, XMD stops at an `fmadd` instruction that has caused a loss of accuracy exception. Then, a call to the function `proceed` resumes execution till XMD stops again at `fmul` which also has caused an exception.



```
fffe05e4:    3d 20 ff ff      lis
fffe05e8:    c8 09 98 38      lfd
fffe05ec:    d8 1f 00 10      stfd
  double c ;//= a+b;
  //if ( b > a)
        c = a+b;
fffe05f0:    c9 bf 00 18      lfd
fffe05f4:    c8 1f 00 10      lfd
fffe05f8:    fc 0d 00 2a      fadd
fffe05fc:    d8 1f 00 08      stfd
        c = a*b;
fffe0600:    c9 bf 00 18      lfd
fffe0604:    c8 1f 00 10      lfd
fffe0608:    fc 0d 00 32      fmul
fffe060c:    d8 1f 00 08      stfd
    //c = a*b;
  //c = sqrt(a);
  //asm ("fmadd 13,12,1,4");
  printf ("c = %f\n",c);
fffe0610:    3d 20 ff ff      lis
```

```
XMD% proceed
Setting breakpoint at 0xfffe05f8

Info:Processor started. Type "stop" to stop processor

RUNNING> Info:Software Breakpoint 3 Hit, Processor Stopped at 0xfffe05f8

XMD% Setting breakpoint at 0xfffe05f8

Info:Processor started. Type "stop" to stop processor

RUNNING> Info:Software Breakpoint 3 Hit, Processor Stopped at 0xfffe05f8

XMD% Info:Processor started. Type "stop" to stop processor

RUNNING> Info:Processor started. Type "stop" to stop processor

RUNNING> XMD% Info: Illegal op-code exception has happend at address 0xfffe0608.



XMD% Info: Illegal op-code exception has happend at address 0xfffe0608.


XMD%
```

**Figure 5.3:** XMD catching two consequent NAU exceptions.



TestApp_Memory.c - Source Window
File Run View Control Preferences Help
Find:
TestApp_Memory.c
main
SOURCE

```
33   //if ( b > a)
34                 c = a+b;
35                 c = a*b;
36           //c = a*b;
37   //c = sqrt(a);
```

```
Program stopped at line 35        fffe0608    35
```

**Figure 5.4:** gdb localizing a NAU exception.

### 5.4.2 GDB

powerpc-eabi-gdb can be used to localize the source of exception on the C-statement level. If an exception takes place upon using one of the approaches presented in 5.4.1. It is possible to use powerpc-eabi-gdb to connect to either one of the processors through a TCP connection to XMD. Upon establishing the connection successfully, the C-statement which caused the exception will be highlighted automatically. Figure 5.4 shows powerpc-eabi-gdb highlighting a C-statement which caused a NAU exception.

# 6 Performance Evaluation

In this chapter, the performance of the proposed architectures are presented. In Section 6.1, the benchmark used to measure the performance is introduced. In the sections that follow, comparison between the performance of different architectures with different optimization levels are presented. Finally, comparison between the performance of the CADNA library vs. the performance of the proposed architecture is presented in Section 6.5.

## 6.1 FP Performance Benchmark

The steps to measure the FP performance can be summarized as follows:

- A bunch of FP instructions are executed for a sufficiently adequate time (approximately 100 seconds).
- The time elapsed for the execution in micro-seconds ($T$) is noted.
- The number of the FP operations executed ($N$) is also noted.
- The performance is measured in terms of **M**ega **F**loating **P**oint **O**peration per **S**econd (MFLOPS) and calculated as follows:

  $$\text{MFLOPS} = \frac{N}{T}$$

  The benchmarks used to evaluate the FP performance are provided in Appendix A.3. The loop that contains the FP instructions is adopted from the Whetstone benchmark[1].

## 6.2 Configuration of The Components

The components used for performance evaluation in the following sections are configured as summarized in Table 6.1. The two PowerPCs were configured to run at their maximum operating frequency. The FPUs were also configured to operate at their maximum operating frequency when optimized for low latency (i.e. `C_LATENCY_CONF` is set to 1). The NAU is configured to run at twice the operating frequency of an STFPU. An Intel T6600 processor is used to evaluate the CADNA performance.

[1]The Whetstone benchmark can be downloaded from www.roylongbottom.org.uk/whets.c

| Component | Operating frequency | Notes |
|---|---|---|
| PowerPC#1 | 400 MHz | |
| PowerPC#2 | 400 MHz | |
| Xilinx FPU | 133 MHz | Optimized for low latency |
| STFPU#1 | 100 MHz | Optimized for low latency |
| STFPU#2 | 100 MHz | Optimized for low latency |
| NAU | 200 MHz | |
| Intel T6600 | 2200 MHz | Used for CADNA evaluation |

**Table 6.1:** Configuration of the components used in the performance evaluation.

## 6.3 Xilinx FPU Vs. STFPU

In order to give a feeling of the performance of the proposed architecture, a comparison is held between an STFPU[2] and Xilnix FPU (v1.01a). Four variants of the benchmark were used. The first two are compiled with no optimization (-O0), and the others are compiled with aggressive optimization (-O3). For each of these optimization levels, the FPUs were configured to run autonomously and non-autonomously.

Figure 6.1 (Figure 6.2) shows the results of comparison when the FPUs run autonomously (non-autonomously) for the two optimization levels. The STFPU is slightly slower than Xilinx FPU as it operates at 75% of the frequency of Xilinx FPU. Another reason is that the latencies for the FP operators in the two FPUs are not exactly the same. Although the autonomous mode exhibits better performance, the PowerPC can not catch exact exceptions in this mode. So the NAU exceptions can only be caught in the non-autonomous mode.

## 6.4 STFPU Vs. STFPU with NAU Attached

The four variants of the benchmark presented in Section 6.3 are used here in order to compare a system that uses the original STFPU to a system that uses two modified STFPUs with the NAU attached.

In the non-autonomous case, Figure 6.3 and Figure 6.4 show the results of comparison when SBEU is implemented based on the generic data path and the simplified data path, respectively. As shown in the figures, the performance is only slightly reduced when the NAU is integrated to the system. The results shown in the two figures collide with the information provided earlier in Table 4.13.

In the autonomous case, there is no difference at all in the performance. Figure 6.5 and Figure 6.6 show the results of the comparison between STFPU and STFPU with NAU

[2]NAU is not attached to the STFPUs.

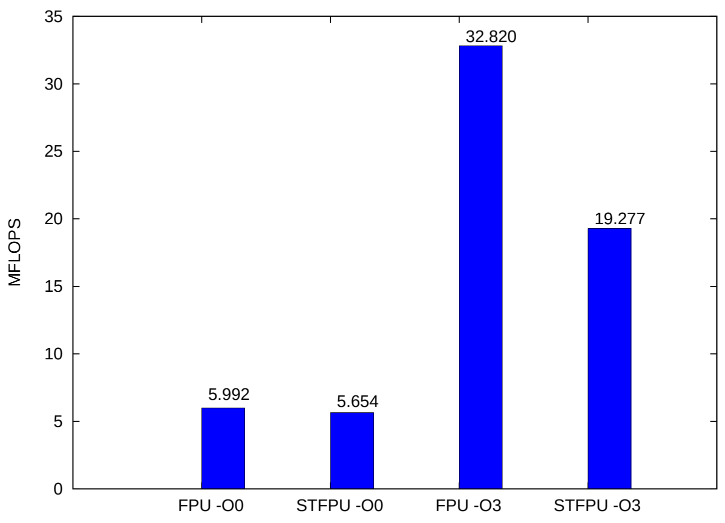


**Figure 6.1:** MFLOPS obtained upon running FPU and Xilinx FPU autonomously.

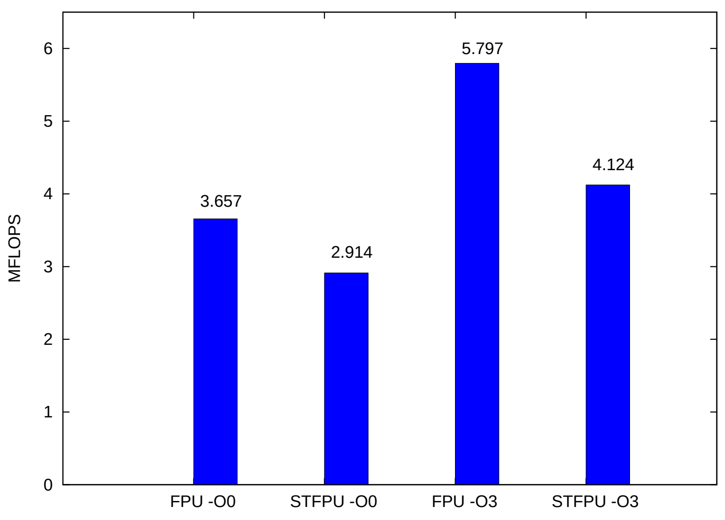


**Figure 6.2:** MFLOPS obtained upon running FPU and Xilinx FPU non-autonomously.

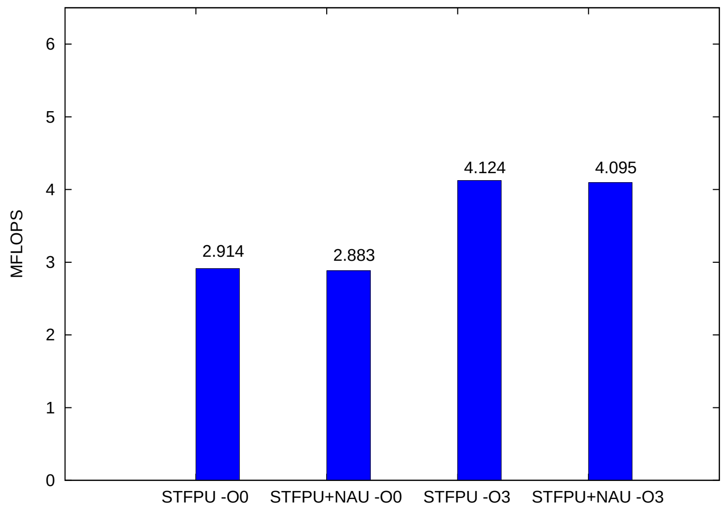


**Figure 6.3:** MFLOPS obtained upon running STFPU with and without the NAU non-autonomously for the generic data path.

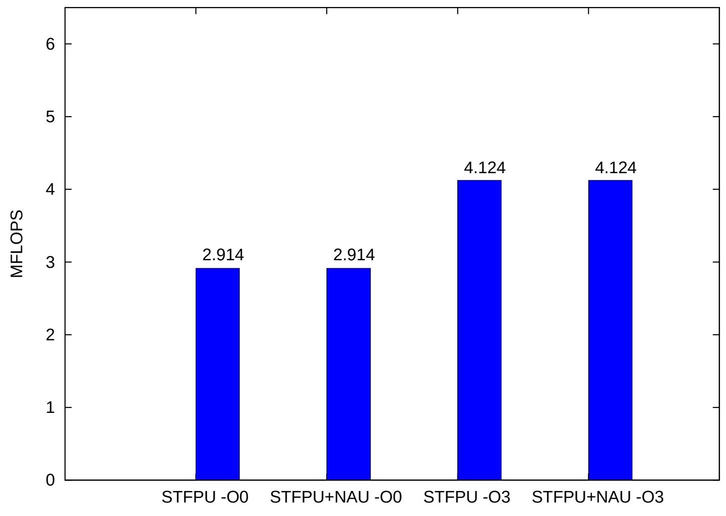


**Figure 6.4:** MFLOPS obtained upon running STFPU with and without the NAU non-autonomously for the simplified data path.

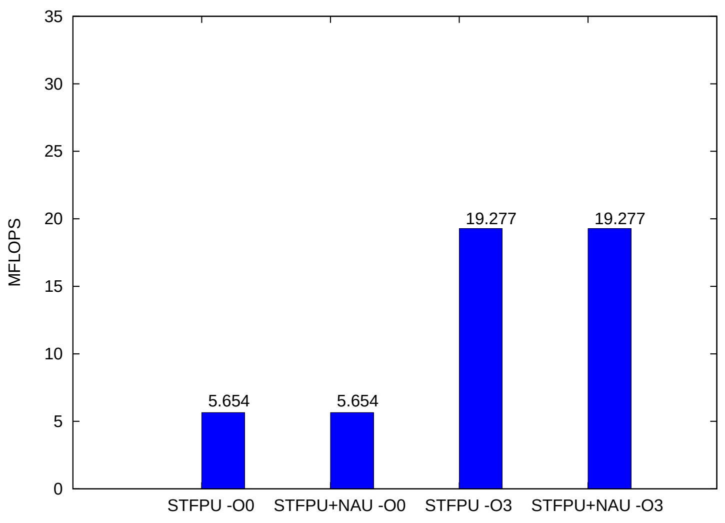


**Figure 6.5:** MFLOPS obtained upon running STFPU with and without the NAU autonomously for the generic data path.

attached considering the generic data path and the simplified data path, respectively. The obtained results are expected because the NAU is logically detached when the STFPUs run autonomously (refer to Section 4.5.3).

## 6.5 The Proposed Architecture Vs. CADNA

The main disadvantage of using the CADNA library is that it requires modifications of the original source code. However, in this section we present the impact of using the CADNA library on the performance.

The benchmark is first run on a T6600 2.2Ghz intel processor and the MFLOPS were noted. Then, appropriate modifications were done to the benchmark in order to use the CADNA library. The benchmark is then run with the MFLOPS noted. Figure 6.7 shows the MFLOPS obtained when running the benchmark with and without CADNA. Apparently, using the CADNA library extremely slows down the perforamce. Every FP operation is evaluated three times with random rounding. Moreover, every addition operation requires evaluation of the number of significant digits of the two operands and the result to decide if a cancellation has occurred. Check for the significance of the two operands and divisor at every multiplication FP and division FP operation, respectively, is required. These reasons explain the dramatic slowdown of the performance when using the CADNA library.

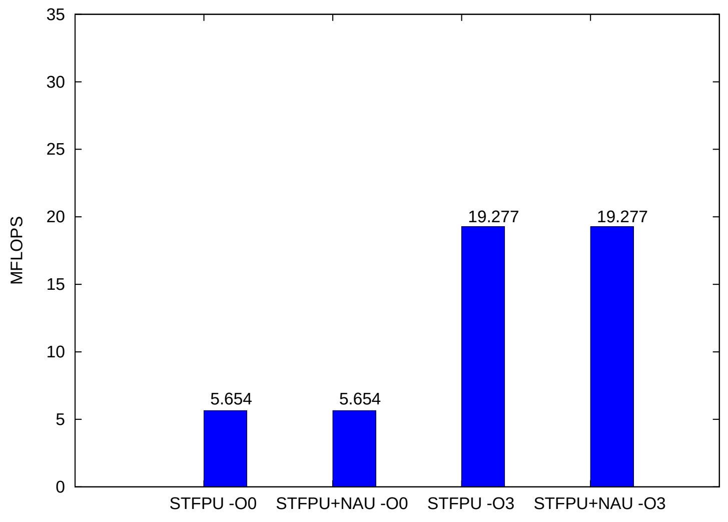


**Figure 6.6:** MFLOPS obtained upon running STFPU with and without the NAU autonomously for the simplified data path.

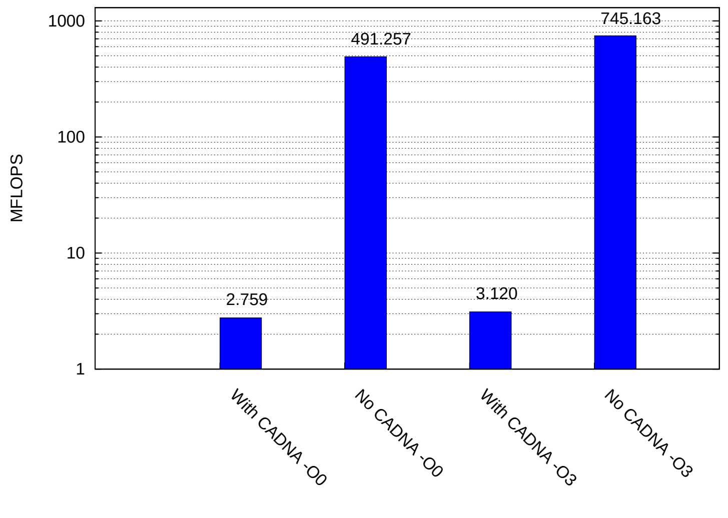


**Figure 6.7:** MFLOPS obtained upon running the benchmark with and without CADNA.

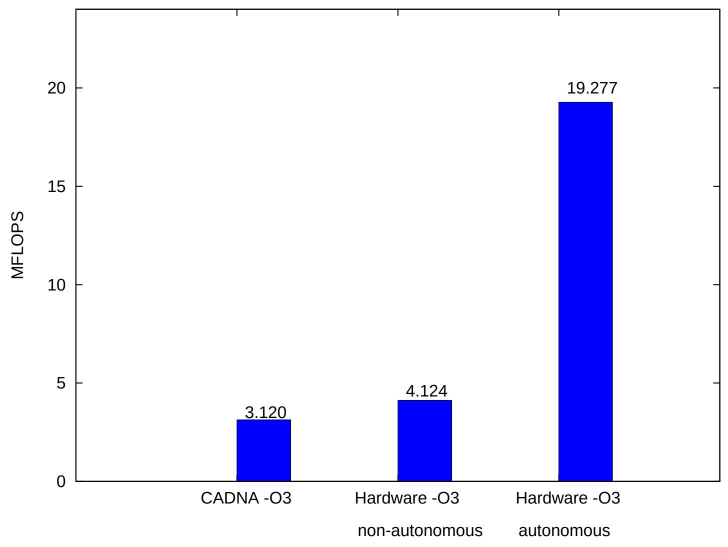


**Figure 6.8:** MFLOPS obtained when using CADNA vs. using our architecture.

Figure 6.8 shows the MFLOPS obtained when using CADNA vs. using the proposed architecture with "-O3" as the optimization level. In the figure, "Hardware" stands for our architecture. Our architecture in the two modes exhibits better performance and particularly in the autonomous mode where it is almost seven times faster than the software solution. It can even exhibit a better performance when implemented on ASIC.

Since CADNA is evaluated on a different architecture, it is logical to compare our architecture to CADNA library by means of *slow down factors*, where:

$$\text{slow down factor} = \frac{\text{MFLOPS obtained originally}}{\text{MFLOPS obtained when implementing DSA}}$$

, holding that the same optimization levels are applied.

Figure 6.9 shows the slow down factors when using our architecture vs. using the CADNA library. Our architecture does not slow down the performance at all in both the autonomous and non-autonomous mode. However, localizing the instruction that is the source of a numerical instability exception is only possible when the STFPUs run non-autonomously. In contrary, the use of CADNA library severely reduces the performance of the system.

For completeness, our architecture does not require any excess memory. On the other hand, the CADNA library requires at least three times the original memory footprint of a program.

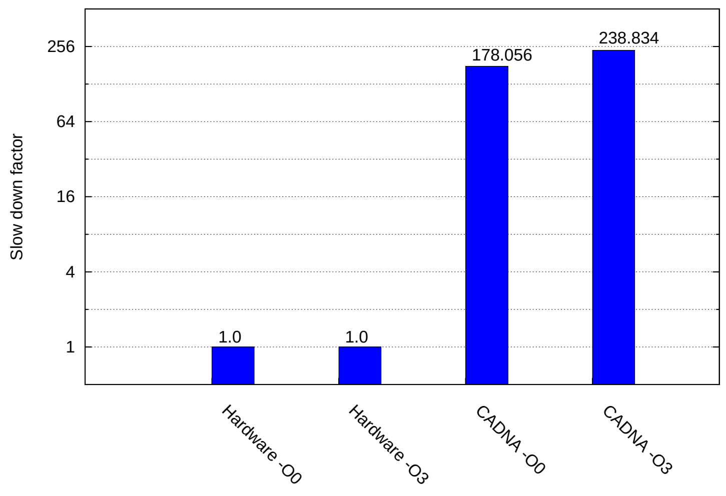


**Figure 6.9:** Slow down factors when using our architecture vs. using the CADNA library.

# 7 Conclusion and Future Work

In this work, a hardware architecture for numerical accuracy analysis has been proposed and implemented. In this architecture, arithmetic and relational operations have been implemented according to the definition of the Discrete Stochastic Arithmetic (DSA), and a Numerical Analysis Unit (NAU) has been implemented such that the information of numerical accuracy can be obtained for any intermediate or final computational result during run time of a program. In the design of the NAU, two simplified approaches for estimating the number of significant digits are used in order to reduce the hardware cost. The first one is generic and can be readily extended, and the other one is optimized for two samples. The latter requires less hardware resources and exhibits better performance.

In order to maintain the reliability of the DSA, operands of multiplication and divisor of division are checked against informational zeros. In addition, exceptions can be generated upon catching a numerical instability (e.g. if the number of significant digits of any intermediate result is less than a predefined value or cancellations are detected in a code) or violation of the hypotheses of the DSA. Information about an exception is written to a special register which can be further accessed by means of a debugger.

There are two existent implementations of DSA – one in software and the other in hardware. Using the software implementation, CADNA library, requires many modifications to be carried out on the original source code. When the source code is not available (e.g. when precompiled library is used), CADNA could not be used. It also imposes a dramatic slow down of the performance. In addition, the memory footprint of a program when using CADNA is at least three times the original one. The existent hardware implementation as a co-processor does neither support all the floating point operations nor the instruction set of the processor. Hence it requires many modifications to be incorporated in the source code, and the additional programing effort is even more than the software solution. For these reasons, it is unlikely to use these solutions in real life. Compared to the existing implementations of the DSA, the architecture proposed in this work exhibits the following advantages and features:

1. Minor modifications to the source code: The STFPUs used in the architecture automatically support the instruction set of the PowerPC. The only required source code modification was the inclusion of the `xexception_l.h` header file and calling `Xexc_Init()` upon entering `main()`. This modification is not even required if the user does not want to resume after catching an exception. This feature makes the proposed solution usable even in the case of precompiled libraries.

2. Reduced hardware cost: This was achieved by using a reduced-hardware approach for calculating the number of significant digits. Using this approach, it was also possible to operate at higher frequencies.
3. Maintaining the original performance of the architecture: This was possible using the reduced-hardware approach and properly implementing the NAU to operate at twice the frequency of the STFPU. By avoiding the use of the costly division, square root and logarithm calculation; the proposed architecture reduces the hardware cost. Benchmarking showed that the performance was not affected at all after integrating the NAU into the system. Although the STFPUs operate at 100MHz, the performance achieved using our implementation supersedes the software implementation. When implemented on ASIC, it can even exhibit a better performance.
4. Possibility to distinguish different types of numerical instabilities: This was possible by utilizing a special register. Upon generating an exception, the NAU writes info about the type of the exception to this register.
5. Localizing the source of an exception and getting more details about them: The possibility to extend the functionality of XMD by loading custom TCL scripts made this possible. A TCL script was developed to facilitate communication with the architecture. Functionality to get more details about exceptions upon their occurrence is also added. Resuming the execution of a program after an exception is caught is provided as well. Helped by the feature, the user may consider to change the code of the statements that caused the exceptions.
6. Wide configuration options: It was made possible to configure the architecture based on the deemed level of reliability through changing a set of generics in the design. For instance, a check for informational zeros in the multiplication operands can be relaxed.

## Future Work

For the PowerPC processor, the only way to raise floating point exceptions is when an STFPU runs non-autonomously. This degrades the performance to some extent. However, there are other interrupts that can be caught by the processor; such as external interrupts. The NAU can be modified to generate such interrupts while the STFPUs run autonomously. The processor will stop, however the challenge is to localize exactly the instruction that caused the exception. This approach will indeed enhance the performance of the architecture.

In the current implementation, the conversion from floating point to integer is done separately on the two STFPUs. This conversion is better done based on the average of the samples instead. In addition, It would be nice to add support for the `printf` function on the hardware level to print the average of the samples instead of printing them separately.

The standalone BSP which provides minimal functionality of an operating system runs on the PowerPCs in our architecture. Still, other operating systems can be also used. Open source operating systems can be modified to support managing the NAU exceptions. This can be doable by incorporating the functionality of the TCL script loaded to XMD in the operating system. This would add a good feature to the proposed solution.

# A Appendix

## A.1 Look-up Tables

### A.1.1 Look-up Tables Entries – The Generic Data Path

The entries for the look-up table utilized in the generic data path are listed in Table A.1. They are calculated based on the formula $f(\hat{M}_{|\bar{r}|}, \hat{M}_{\hat{Z}}) = v_2 + \log_2 \frac{\hat{M}_{|\bar{r}|}}{\sqrt{\hat{M}_{\hat{Z}}}}$. Values of $\hat{M}_{|\bar{r}|}$ ranges from $[000]_2$ ($[1.000]_{10}$) to $[111]_2$ ($[1.875]_{10}$), and values of $\hat{M}_{\hat{Z}}$ ranges from $[00]_2$ ($[1.00]_{10}$) to $[11]_2$ ($[1.75]_{10}$). In the implementation, seven bits are used to represent the value of $-f(\hat{M}_{|\bar{r}|}, \hat{M}_{\hat{Z}})$. For instance, a number such as 3.1664 is saved as $[1100101]_2$ with the implicit fixed point is between the second and third bit from left.

| Key | | $-f(\hat{M}_{\|\bar{r}\|}, \hat{M}_{\hat{Z}})$ | Key | | $-f(\hat{M}_{\|\bar{r}\|}, \hat{M}_{\hat{Z}})$ |
|---|---|---|---|---|---|
| $\hat{M}_{\|\bar{r}\|}$ | $\hat{M}_{\hat{Z}}$ | | $\hat{M}_{\|\bar{r}\|}$ | $\hat{M}_{\hat{Z}}$ | |
| 1.000 | 1.00 | 3.1664 | 1.500 | 1.00 | 2.5814 |
| 1.000 | 1.25 | 3.3274 | 1.500 | 1.25 | 2.7424 |
| 1.000 | 1.50 | 3.4589 | 1.500 | 1.50 | 2.8739 |
| 1.000 | 1.75 | 3.5701 | 1.500 | 1.75 | 2.9851 |
| 1.125 | 1.00 | 2.9965 | 1.625 | 1.00 | 2.4660 |
| 1.125 | 1.25 | 3.1575 | 1.625 | 1.25 | 2.6269 |
| 1.125 | 1.50 | 3.2890 | 1.625 | 1.50 | 2.7585 |
| 1.125 | 1.75 | 3.4002 | 1.625 | 1.75 | 2.8697 |
| 1.250 | 1.00 | 2.8445 | 1.750 | 1.00 | 2.3591 |
| 1.250 | 1.25 | 3.0054 | 1.750 | 1.25 | 2.5200 |
| 1.250 | 1.50 | 3.1370 | 1.750 | 1.50 | 2.6515 |
| 1.250 | 1.75 | 3.2482 | 1.750 | 1.75 | 2.7627 |
| 1.375 | 1.00 | 2.7070 | 1.875 | 1.00 | 2.2595 |
| 1.375 | 1.25 | 2.8679 | 1.875 | 1.25 | 2.4205 |
| 1.375 | 1.50 | 2.9995 | 1.875 | 1.50 | 2.5520 |
| 1.375 | 1.75 | 3.1107 | 1.875 | 1.75 | 2.6632 |

**Table A.1:** Look-up table entries for the generic data path.

### A.1.2 Look-up Tables Entries – The Simplified Data Path

The entries for the look-up table utilized in the simplified data path are listed in Table A.2. They are calculated based on the formula $w(\hat{M}_{|s|}, \hat{M}_{|d|}) = \log_2\left(\frac{1}{\tau_\beta}\right) + \log_2\left(\frac{M_{|s|}}{M_{|d|}}\right)$. Values of $\hat{M}_{|s|}$ ranges from $[000]_2$ ($[1.000]_{10}$) to $[111]_2$ ($[1.875]_{10}$), and values of $\hat{M}_{|d|}$ ranges from $[00]_2$ ($[1.00]_{10}$) to $[11]_2$ ($[1.75]_{10}$). In the implementation, seven bits are used to represent the value of $-w(\hat{M}_{|s|}, \hat{M}_{|d|})$. For instance, a number such as 3.6678 is saved as $[0111001]_2$ with the implicit fixed point is between the third and fourth bit from left.

| Key | | $-w(\hat{M}_{\lvert s\rvert}, \hat{M}_{\lvert d\rvert})$ | Key | | $-w(\hat{M}_{\lvert s\rvert}, \hat{M}_{\lvert d\rvert})$ |
|---|---|---|---|---|---|
| $\hat{M}_{\lvert s\rvert}$ | $\hat{M}_{\lvert d\rvert}$ | | $\hat{M}_{\lvert s\rvert}$ | $\hat{M}_{\lvert d\rvert}$ | |
| 1.000 | 1.00 | 3.6678 | 1.500 | 1.00 | 3.0829 |
| 1.000 | 1.25 | 3.9898 | 1.500 | 1.25 | 3.4048 |
| 1.000 | 1.50 | 4.2528 | 1.500 | 1.50 | 3.6678 |
| 1.000 | 1.75 | 4.4752 | 1.500 | 1.75 | 3.8902 |
| 1.125 | 1.00 | 3.4979 | 1.625 | 1.00 | 2.9674 |
| 1.125 | 1.25 | 3.8198 | 1.625 | 1.25 | 3.2893 |
| 1.125 | 1.50 | 4.0829 | 1.625 | 1.50 | 3.5524 |
| 1.125 | 1.75 | 4.3053 | 1.625 | 1.75 | 3.7748 |
| 1.250 | 1.00 | 3.3459 | 1.750 | 1.00 | 2.8605 |
| 1.250 | 1.25 | 3.6678 | 1.750 | 1.25 | 3.1824 |
| 1.250 | 1.50 | 3.9309 | 1.750 | 1.50 | 3.4455 |
| 1.250 | 1.75 | 4.1533 | 1.750 | 1.75 | 3.6678 |
| 1.375 | 1.00 | 3.2084 | 1.875 | 1.00 | 2.7610 |
| 1.375 | 1.25 | 3.5303 | 1.875 | 1.25 | 3.0829 |
| 1.375 | 1.50 | 3.7934 | 1.875 | 1.50 | 3.3459 |
| 1.375 | 1.75 | 4.0158 | 1.875 | 1.75 | 3.5683 |

**Table A.2:** Look-up table entries for the simplified data path.

## A.2 C Code emulating SBEU

In order to help testing SBEU, the C code provided in Listing A.1 is used to emulate the generic data path and the simplified data path on the bit level. Exact significant bits as well as intermediate variables are printed out in order to be compared to the signal in the HDL design.

```
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
```

```
/* This program is used to generate test cases for the generic data path
   and the simplified data path.
 */
int main()
{
/**********************Samples are loaded here*************************/
    /* The first sample */
    double a =100.0001;
    /* The second sample */
    double b =100.0003;
/*********************************************************************/
/* Local variables */
    long long a_hex= *(long long*)(&a);
    long long b_hex = *(long long*)(&b);
    double man_R;
    long long man_R_hex;
    double man_Z;
    long long man_Z_hex;
    double R, a_R,a_R_2,b_R_2, b_R,sum,dif, Z, sigma;
    long long R_hex, sum_hex, a_R_hex,a_R_2_hex,dif_hex, b_R_2_hex, b_R_hex, Z_hex;
    int man_R_3;
    int man_Z_2;
    int man_sum_3;
    int man_dif_2;
    int exp_R;
    int exp_Z;
    double v = -0.9532;
    double T = 12.71; /* tau */
    double Cr;
    int Cr_win;
    double Cr_app;
    double Cr_app_bits;

    /* The look-up table for the generic data path */
    int lookup_key[32] = {
        0x65, 0x69, 0x6f, 0x72, 0x60, 0x65, 0x69, 0x6d,
        0x56, 0x60, 0x64, 0x68, 0x57, 0x5c, 0x60, 0x63,
        0x53, 0x58, 0x5c, 0x60, 0x4f, 0x54, 0x58, 0x5c,
        0x4b, 0x51, 0x54, 0x58, 0x48, 0x4e, 0x52, 0x55
    };

    /* The look-up table for the simplified data path */
    int look_up_2[32] = {
        0x3B, 0x3F, 0x44, 0x48, 0x38, 0x3D, 0x41, 0x45,
        0x36, 0x3B, 0x3E, 0x42, 0x33, 0x38, 0x3C, 0x40,
        0x31, 0x36, 0x3B, 0x3D, 0x2E, 0x35, 0x39, 0x3C,
        0x2D, 0x33, 0x37, 0x3B, 0x2C, 0x31, 0x36, 0x39
    };
    int key;
    int exp_Z_app, exp_R_app;

    /**********************Generic data path*************************/
    R = (a+b)/2.0;
    R_hex = *(long long*)(&R);
    /* print the samples in hex */
    printf ("a_hex = %llx\tb_hex = %llx\n",a_hex,b_hex);
    /* print the average of the samples in hex */
    printf ("R_hex = %llx\n", R_hex);
    man_R_hex = R_hex & 0x000FFFFFFFFFFFFF;
    man_R_hex = man_R_hex | 0x3FF0000000000000;
    /* print the mantissa of the average in hex */
    printf ("man_R_hex = %llx\n", man_R_hex);
    /* extract the first three MSBs of the mantissa of the average */
    man_R = *(double*)(&man_R_hex);
    man_R_hex = man_R_hex & 0x000FFFFFFFFFFFFF;
    man_R_3 = man_R_hex >> 49;
    man_R_3 = man_R_3 << 2;
    /* print the first three MSBs of the mantissa of the average */
    printf ("man_R = %f\n", man_R);
    /* extract the exponent of the average */
    exp_R = (int) ((R_hex >> 52) & 0x00000000000007FF );
    exp_R -= 1023; /* remove the bias */
    exp_R_app = exp_R << 5;
    /* print the unbiased exponent of the average */
    printf ("exp_R = %d\n", exp_R);
    /* calculate the difference between the average and the samples */
    a_R = a-R;
    a_R_hex = *(long long*)(&a_R);
    a_R_hex = a_R_hex & 0xFFFF000000000000; /* truncate to lower precision */
    a_R = *(double*)(&a_R_hex);
    printf ("a_R_hex = %llx\n",a_R_hex);
    b_R = b-R;
    b_R_hex = *(long long*)(&b_R);
    b_R_hex = b_R_hex & 0xFFFF000000000000; /* truncate to lower precision */
    b_R = *(double*)(&b_R_hex);
    printf ("b_R_hex = %llx\n",b_R_hex);
    /* calculate the square of the differences */
    a_R_2 = pow(a_R,2);
    a_R_2_hex = *(long long*)(&a_R_2);
    a_R_2_hex = a_R_2_hex & 0xFFFF000000000000; /* truncate to lower precision */
    a_R_2 = *(double*)(&a_R_2_hex);
    b_R_2 = pow(b_R,2);
    b_R_2_hex = *(long long*)(&b_R_2);
    b_R_2_hex = b_R_2_hex & 0xFFFF000000000000; /* truncate to lower precision */
    b_R_2 = *(double*)(&b_R_2_hex);
    /* calculate the sum of the squares */
    Z = (a_R_2 + b_R_2);
    printf ("sigma = %f\n",Z);
    Z_hex = *(long long*)(&Z);
    Z_hex = Z_hex & 0xFFFF000000000000;
    printf ("Z_hex = %llx\n", Z_hex);
    man_Z_hex = Z_hex & 0x000FFFFFFFFFFFFF; /* truncate to lower precision */
    man_Z_hex = man_Z_hex | 0x3FF0000000000000;
    printf ("man_Z_hex = %llx\n", man_Z_hex);
    /* extract the two MSBs of the mantissa of sigma */
    man_Z = *(double*)(&man_Z_hex);
    printf ("man_Z = %f\n", man_Z);
    man_Z_hex = man_Z_hex & 0x000FFFFFFFFFFFFF;
    man_Z_2 = man_Z_hex >> 50;
    /* extract the mantissa of sigma */
    exp_Z = (int) ((Z_hex >> 52) & 0x00000000000007FF);
    exp_Z -= 1023; /* remove the bias */
    exp_Z_app = exp_Z << 5;
    printf ("exp_Z = %d\n", exp_Z);
    /* Calculate number of exact significant digits Cr */
    sigma = sqrt(pow(a-R,2)+pow(b-R,2));
    Cr = log10(sqrt(2.0)*fabs(R)/(sigma*T));
    printf ("exact Cr = %f\n",Cr);
    /* Calculate the number of exact significant bits based on the generic data path Cr_win */
    printf("man_R_3 = %d\tman_Z_2 = %d\n", man_R_3, man_Z_2);
    key = (int) (man_R_3 | man_Z_2);
    printf ("key = %d\n",key);
    Cr_win = (exp_R_app - (exp_Z_app >> 1) - lookup_key[key]) << 1;
    printf("Generic Cr =%x\n", Cr_win);

    /**********************simplified data path************************/
    sum = (a+b);
    dif = (a-b);
    sum_hex = *(long long*)(&sum);
    printf ("sum_hex =%llx\n", sum_hex);
    dif_hex = *(long long*)(&dif);
    printf ("dif_hex =%llx\n", dif_hex);
    /* extract the first three MSBs of sum */
    man_R_hex = sum_hex & 0x000FFFFFFFFFFFFF;
    man_R_hex = man_R_hex | 0x3FF0000000000000;
    man_R = *(double*)(&man_R_hex);
    man_R_hex = man_R_hex & 0x000FFFFFFFFFFFFF;
    man_sum_3 = man_R_hex >> 49;
    man_sum_3 = man_sum_3 << 2;
    printf ("man_sum = %f\n", man_R);
    /* extract the first two MSBs of dif */
    dif_hex = *(long long*)(&dif);
    Z_hex = dif_hex & 0x000FFFFFFFFFFFFF;
    man_Z_hex = Z_hex & 0x000FFFFFFFFFFFFF;
    man_Z_hex = man_Z_hex | 0x3FF0000000000000;
    printf ("man_diff_hex = %llx\n", man_Z_hex);
    man_Z = *(double*)(&man_Z_hex);
    man_Z_hex = man_Z_hex & 0x000FFFFFFFFFFFFF;
    man_dif_2 = man_Z_hex >> 50;
    printf ("man_dif = %f\n", man_Z);
    printf("man_R_3 = %d\tman_Z_2 = %d\n", man_R_3, man_Z_2);
    key = (int) (man_sum_3 | man_dif_2);
    printf ("key = %d\n",key);
    /* extract the exponent of the sum and dif */
    exp_R = (int) ((sum_hex >> 52) & 0x00000000000007FF );
    exp_R_app = exp_R << 6;
    exp_Z = (int) ((dif_hex >> 52) & 0x00000000000007FF );
    exp_Z_app = exp_Z << 6;
    /* print the number of exact significant bits for the simplified data path */
    Cr_win = ((exp_R_app - exp_Z_app) - ((look_up_2[key]) << 2));
    printf("Simplified Cr =%x\n", Cr_win);
}
```

**Listing A.1:** A program to emulate SBEU on the bit level.

## A.3 Benchmarks

The benchmarks used for performance evaluation of the software implementation (CADNA) and the proposed hardware implementation are listed in Subsection A.3.1 and Subsection A.3.2, respectively.

### A.3.1 Evaluating The Hardware Architecture

```
/* This is a benchmark to check the performance of the FPUs based on Whetstone benchmark */

#include "xparameters.h"
#include "xcache_l.h"
#include "stdio.h"
#include "xutil.h"
#include "xtime_l.h"
#include <time.h>

int main()
{
  XCache_EnableICache(0x90000000); /* Enable instruction cache */
  XCache_EnableDCache(0x90000000); /* Enable data cache */

  /* Local variables */
  int ix;
  int xtra = 200;
  int n1 = 1200;
  int i;
  int n1mult = 20;
  double e1[4];
  double t = 0.49999975;
  double t2 = 2.0;
  double t0 = t;
  double no_of_fp_op;
  double time_elapsed;
  XTime start,end;
  XTime clock_cycles;
  e1[0] = 1.0;
  /* Here add a redundent operation to synchronize the two FPUs */
  e1[3] = e1[0] * 3.3242;
  e1[1] = -1.0;
  e1[2] = -1.0;
  e1[3] = -1.0;

  XTime_GetTime(&start); /* Read the Time Stamp Counter */
  /* The variables xtra, n1 and n1mult were tuned to let the execution
  loop run more than 30 sec */
  for (ix=0; ix<xtra; ix++)
    {
      for(i=0; i<n1*n1mult; i++)
               { /* 16 floating point operations in the loop */
                 e1[0] = (e1[0] + e1[1] + e1[2] - e1[3]) * t;
```

```
                  e1[1] = (e1[0] + e1[1] - e1[2] + e1[3]) * t;
                  e1[2] = (e1[0] - e1[1] + e1[2] + e1[3]) * t;
                  e1[3] = (-e1[0] + e1[1] + e1[2] + e1[3]) * t;
                }
      t = 1.0 - t;
    }
  XTime_GetTime(&end); /* Read the Time Stamp Counter */
  clock_cycles = end - start; /* No. of elapsed clock cycles*/
  /* The results are printed to check correct functionality of the FPUs */
  print("e1[0]          e1[1]          e1[2]         e1[3]\n\r");
  printf ("%f\t%f\t%f\t%f\n\n", e1[0], e1[1], e1[2], e1[3]);
  /* MFLOPs = xtra * n1 * n1mult * 16 operations / elapsed time(in micro sec) */
  printf ("MFLOPs = %d * %d * %d * 16 * 400 / %lld \n\n", xtra, n1, n1mult, clock_cycles);
  XCache_DisableDCache();
  XCache_DisableICache();
}
```

**Listing A.2:** The benchmark used to evaluate the hardware architecture.

### A.3.2 Evaluating The Software Solution

#### Without CADNA

```
/* This is a benchmark to check the performance of the CPU without CADNA based on Whetstone benchmark */
#include "stdio.h"
#include "stdint.h"
#include "stdlib.h"
#include "unistd.h"

/* This function is used to read the Time Stamp Register*/
static inline unsigned long long getrdtsc(void)
{
  unsigned long long x;

#if defined (__i386__)
  __asm__ __volatile__ ("rdtsc" : "=A" (x));
#elif defined (__x86_64__)
  unsigned int tickl, tickh;
  __asm__ __volatile__ ("rdtsc" : "=a" (tickl), "=d" (tickh));
  x = ((unsigned long long)tickh << 32) | tickl;
#else
#warning "Architecture not yet supported in ASM"
#endif
  return x;
}

int main()
{ /*local variables*/
  double e1[4];
  long long unsigned ix;
  long long unsigned xtra = 20;
  long long unsigned i;
  long long unsigned n1mult = 30000;
  long long unsigned n1 = 1200;
  double t = 0.49999975;
  double t0 = t;
  double t1 = 0.50000025;
  double t2 = 2.0;
  e1[0] = 1.0;
  e1[1] = -1.0;
  e1[2] = -1.0;
  e1[3] = -1.0;
  unsigned long long int start_tsc, stop_tsc;

  start_tsc = getrdtsc(); /* Read the Time Stamp Counter */
  /* The variables xtra, n1 and n1mult were tuned to let the execution
  loop run more than 30 sec */
    for (ix=0; ix<xtra; ix++)
      {
              for(i=0; i<n1*n1mult; i++)
                {/* 16 floating point operations in the loop */
                     e1[0] = (e1[0] + e1[1] + e1[2] - e1[3]) * t;
                     e1[1] = (e1[0] + e1[1] - e1[2] + e1[3]) * t;
                     e1[2] = (e1[0] - e1[1] + e1[2] + e1[3]) * t;
                     e1[3] = (-e1[0] + e1[1] + e1[2] + e1[3]) * t;
                }
              t = 1.0 - t;
      }
    t = t0;

  stop_tsc = getrdtsc(); /* Read the Time Stamp Counter */
  printf("e[3]=%f\n",e1[3]);
  printf("operations = %llu\n", xtra * n1 * n1mult * 16);
  /* MFLOPs = xtra * n1 * n1mult * 16 operations / elapsed time(in micro sec) */
  printf ("%f\n", ((float)(n1mult) * xtra * n1*16)/((stop_tsc - start_tsc)/2200.0));
}
```

**Listing A.3:** The benchmark used to evaluate the original performance of the processor (without CADNA).

With CADNA

```
/* This is a benchmark to check the performance of the CPU with CADNA based on Whetstone
    benchmark */
#include "stdio.h"
#include "stdlib.h"
#include "unistd.h"
#include "string.h"
#include <cadna.h>

/* This function is used to read the Time Stamp Register*/
static inline unsigned long long getrdtsc(void)
{
  unsigned long long x;

#if defined (__i386__)
  __asm__ __volatile__ ("rdtsc" : "=A" (x));
#elif defined (__x86_64__)
  unsigned int tickl, tickh;
  __asm__ __volatile__ ("rdtsc" : "=a" (tickl), "=d" (tickh));
  x = ((unsigned long long)tickh << 32) | tickl;
#else
#warning "Architecture not yet supported in ASM"
#endif
  return x;
}

int main()
{ /*local variables*/
   double_st e1[4];
  long long unsigned ix;
  long long unsigned xtra = 40;
  long long unsigned i;
  long long unsigned n1mult = 100;
  long long unsigned n1 = 1200;
   double_st t = 0.49999975; /* Stochastic double instead of double */
   double_st t0 = t; /* Stochastic double instead of double */
   double_st t1 = 0.50000025; /* Stochastic double instead of double */
   double_st t2 = 2.0; /* Stochastic double instead of double */
  e1[0] = 1.0;
  e1[1] = -1.0;
  e1[2] = -1.0;
  e1[3] = -1.0;
  unsigned long long int start_tsc, stop_tsc;

  /* Inititialize CADNA enabling all numerical instability checks */
   cadna_init(-1);

  start_tsc = getrdtsc(); /* Read the Time Stamp Counter */
  /* The variables xtra, n1 and n1mult were tuned to let the execution
  loop run more than 30 sec */
    for (ix=0; ix<xtra; ix++)
      {
              for(i=0; i<n1*n1mult; i++)
                {
                  e1[0] = (e1[0] + e1[1] + e1[2] - e1[3]) * t;
              e1[1] = (e1[0] + e1[1] - e1[2] + e1[3]) * t;
              e1[2] = (e1[0] - e1[1] + e1[2] + e1[3]) * t;
              e1[3] = (-e1[0] + e1[1] + e1[2] + e1[3]) * t;
                }
              t = 1.0 - t;
      }
    t = t0;

  stop_tsc = getrdtsc(); /* Read the Time Stamp Counter */
  printf("e[3]=%s\n", strp(e1[3]));
  /* MFLOPs = xtra * n1 * n1mult * 16 operations / elapsed time(in micro sec) */
  printf ("MFLOPs = %g\n", (float)(xtra * n1 * n1mult * 16) / ((stop_tsc - start_tsc)/2200));
```

```
    cadna_end();
}
```

**Listing A.4:** The benchmark used to evaluate the performance of the processor when using CADNA

## A.4 Numerical Analysis Debugging Script

The TCL script in Listing A.5 is used to facilitate communicating to the board. Upon loading XMD with the script (`source script.tcl`), the two PowerPCs start the execution of the executables. When an exception is caught, a call to `more_details` prints out more info about the exception. Continuing from an exception is possible using `proceed` function.

```
#Connect to the first processor
connect ppc hw -debugdevice cpunr 1 fputype dp
#Disable resetting the processors upon downloading the executables\
#otherwise elf_verify would fail!
debugconfig -reset_on_run disable
#Download the first executable
dow ./cestac_check1/executable.elf
#Set the safemode in the XMD to on, i.e. stop at trapped exceptions
safemode on

#Connect to the second processor
connect ppc hw -debugdevice cpunr 2 fputype dp
#Disable resetting the processors upon downloading the executables\
otherwise elf_verify would fail!
debugconfig -reset_on_run disable
#Download the second executable
dow ./cestac_check2/executable.elf
#Set the safemode in the XMD to on, i.e. stop at trapped exceptions
safemode on

#Switch to the first PowerPC
targets 0
#Set a breakpoint at main
bps main
#Continue till the breakpoint is reached
con
#Update is used to force finishing "con" before proceding to the next line
update
#Set the mode to non-autonomous and enable exceptions, i.e. set the\
#bits FE0, FE1, FP in the msr to '1'
rwr msr 0x2900
#rwr msr 0x2000; #would be used instead in the autonomous mode
#Continue execution
con
#Do the same for the other processor
targets 1
```

```
bps main
con
update
rwr msr 0x2900
#rwr msr 0x2000; #would be used instead in the autonomous mode
con

#This function is used to read the syndrome register and state the type of the\
#caught exception.
proc more_details args {
        targets 0
        set reg [mrd 0xc9200000]
        if {$reg=="C9200000: BBBBBBBB\n"} {
                        puts "Branching instability"
                }
        if {$reg=="C9200000: DDDDDDDD\n"} {
                        puts "Insignificant divisor"
                        }
        if {$reg=="C9200000: 1F1F1F1F\n"} {
                        puts "Insiginificant multiplication operand"
                        }
        if {$reg=="C9200000: 10551055\n"} {
                        puts "Loss of accuracy"
                        }
        if {$reg=="C9200000: CACECACE\n"} {
                        puts "Cancellation"
                        }
        }

#This function is to be called upon the willingness of resuming the execution\
#after catching an exception
proc proceed args {
        targets 0
        set temp 0x[string replace [rrd pc] 0 7];
        set pc [string trimright $temp ];
        bps $pc;
        con;
        update
        rwr msr 0x2000;
        stp;
        rwr msr 0x2900;
        targets 1
        bps $pc;
        con;
        update
        rwr msr 0x2000;
        stp;
        rwr msr 0x2900;
        con
        targets 0
        con
        }
```

**Listing A.5:** The Numerical Analysis Debugging Tool

# Bibliography

[ACM04] R. Avot-Chotin, H. Mehrez. Hardware implementation of discrete stochastic arithmetic. *Numerical Algorithms*, pp. 21–33, 2004. (Cited on pages 9, 13, 23 and 24)

[boo02] *Book E: Enhanced PowerPC Architecture*, 2002. (Cited on pages 28, 46 and 55)

[cad] *CADNA for C/C++ source codes*. Laboratoire d'Informatique de Paris 6 Université Pierre et Marie Curie - Paris 6 Paris, France. URL http://www.lip6.fr/cadna. (Cited on page 21)

[Che] J.-M. Chesneaux. Validité du logiciel numérique. URL http://www-pequan.lip6.fr/~jmc/polycopies/poly_vln.pdf. (Cited on page 18)

[Cli01] H. E. Cline. IBM PowerPC 440 Microprocessor Core Programming Model Overview. Technical report, IBM Microelectronics, 2001. (Cited on page 55)

[CM] R. Chotin, H. Mehrez. Hardware implementation of a method to control round-off errors. (Cited on page 23)

[iee08] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, 2008. doi:10.1109/IEEESTD.2008.4610935. URL http://dx.doi.org/10.1109/IEEESTD.2008.4610935. (Cited on page 17)

[JC08] F. Jézéquel, J. M. Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, 2008. (Cited on page 21)

[Li10] W. Li. Numerical Accuracy Analysis in Simulations on Hybrid High-Performance Computing Systems. Technical report, Institue of Parallel and Distributed Systems, University of Stuttgart, 2010. (Cited on pages 27, 34 and 36)

[Mil] T. Miller. PPC440 Processor User's Manual. (Cited on page 55)

[Vig93] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Math. Comput. Simul.*, 35:233–261, 1993. doi:10.1016/0378-4754(93)90003-D. URL http://dl.acm.org/citation.cfm?id=165789.165792. (Cited on pages 19 and 20)

[Vig04] J. Vignes. Discrete Stochastic Arithmetic for Validating Results of Numerical Software. 37(1–4):377–390, 2004. URL http://ipsapp009.kluweronline.com/IPS/content/ext/x/J/5058/I/58/A/31/abstract.htm. (Cited on pages 18, 20 and 39)

[VP74] J. Vignes, M. L. Porte. Error Analysis in Computing. In *IFIP Congress'74*, pp. 610–614. 1974. (Cited on page 19)

[Xil] Xilinx. Standalone Board Support Package. (Cited on page 57)

All URLs were last followed on Nov. 1, 2011.

## Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Yousef Baroud)