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

Diplomarbeit Nr. 3084

Development of an Automatic Numerical Stability Analyser Based on a Hardware Implementation of Discrete Stochastic Arithmetic

Sylvain Burlet

Course of Study:	Elektrotechnik und Informationstechnik
Examiner:	Prof. Dr.-Ing. Sven Simon
Supervisor:	M.Sc. Wenbin Li
Commenced:	September 1, 2010
Completed:	March 3, 2011
CR-Classification:	C.3, D.2.5, G.1.0

Abstract

Computers usually represent numbers with finite precision arithmetic. Most real numbers can not be exactly represented. Results of assignment and computations have then to be approximated, and rounding errors are induced. Several approaches have been developed to study the rounding errors. Among these methods, the numerical accuracy analysis based on the discrete stochastic arithmetic is able to provide a tight and accurate estimation of the rounding errors. It have been proven to be effective and reliable in many publications. A hardware platform has been developed in a previous project to use the principles of discrete stochastic arithmetic. On this hardware platform, multiple processing blocks can execute the same program in parallel with random rounding. Adapted floating point units provide the necessary values to compute an estimation of the accuracy.

Based on the dedicated hardware platform, a numerical stability analyser is investigated in this work. The proposed numerical stability analyser is able to gather information from the dedicated hardware platform and provide the numerical accuracy information to the user. The interface to the user is kept the same as a state-of-the-art debugger, while providing additional commands that allows the accuracy estimation of intermediate results. The mechanisms involved in the numerical stability analyser is exposed and its usage is explained.

Contents

List of Figures

1 Introduction

1.1 Motivation

Applications on computers often involves real numbers that are represented according to the IEEE-754 floating point description (see 2.1.1). This representation uses a finite number of binary digits and thus can only represent a finite number of values. The real numbers are approximated and rounded to fit this representation. This causes rounding errors not only in the assignment, but also in each arithmetic operation. These rounding errors propagate and degrade thus the accuracy of the results. The effect of rounding error have to be estimated.

Along the years, several methods have been developed to study the influence of the round-off errors on the results. In theory, this methods could be used to overestimate the error. However, these methods are algorithm-specific and imply a lot of computations. The discrete stochastic arithmetic aims to estimate the loss of accuracy in a computation. The result of any operation is randomly rounded up or down. With several results obtained while running the same program with the same data, it is then possible to estimate the accuracy of a computed result.

The influence of the rounding errors depends on the data but also on the algorithm used and even on the order of the operations. It becomes interesting to be able to stop the execution of a program at a certain steps to check the accuracy loss on several variables. Tools enabling such verifications present then an interest. The discrete stochastic arithmetic provides accuracy information for any variable at any step of a program.

In order to be able to estimate the accuracy of a result by using the discrete stochastic arithmetic, some verifications have to be done with the different intermediate results at each step of the computations. On the hardware platform developed for this purpose, multiple processing blocks can execute the same program in parallel with random rounding. Two adapted floating point units provide the necessary values to compute an estimation of the accuracy at any step. Then a special tool has to be developed to use this hardware and provide the user with the informations he needs.

A normal debugger enables the user to stop the execution of a program at given points of the execution and then have a look at the value of the intermediates results. An existent debugger is modified to be able to run a program on the hardware implementation of the discrete stochastic arithmetic and then provide additionally to the previous information the accuracy of the displayed result.

1.2 Overview

In this work, it will first be explained how real numbers are represented on by computers using the floating point representation. The round-off error for each floating point operation will be analysed. The effect of these errors to the final results will be explained. Different methods developed over the years to overestimate the error of the result will be reviewed. Only a probabilistic approach allows to answer the question : what is the rounding error of the computed result? Discrete stochastic arithmetic provides a method to estimate the accuracy of a computed result by using several runs of the program with random rounding.

In order to apply the principles of discrete stochastic arithmetic, an hardware platform has been developed consisting of two processing blocks. Each processing block is composed of a PowerPC processor and a special Floating Point Unit. It will be explained how this platform is used to compute twice the results of a program with random rounding. The multiple results are used to compute an estimation of the accuracy of the result.

The tool used to analyse the program on the special structure acts as a debugger. It has been modified to use a single program on two processing blocks at the same time. The corresponding modifications of the normal functionalities of the debugger will be reviewed. It will also be explained how the accuracy of the result can then be computed.

The use of the automatic numerical stability analyser based on the hardware implementation of discrete stochastic arithmetic will be explained. The results it achieved and the problems that have been encountered will be reviewed. Finally, some possible extensions of the tool will also be described.

2 Numerical Accuracy analysis

2.1 Rounding Errors in the Computed Results

2.1.1 Finite representation of numbers

Computers use binary coded numbers for every computations. As only a finite number of binary digits is used, not every real number can be exactly represented.

Normalised representation of a real in base b

In mathematics, every real number except zero can be uniquely written with a sign ε, a mantissa m and an exponent e. This representation is used in [3] ($\mathbf{R}^*$ is the set of all real numbers except zero):

$$\forall x \in \mathbf{R}^*,\ x = \varepsilon.b^e.m$$

$$\text{with } \varepsilon \in \{-1, 1\},\ e \in \mathbf{Z},\ m \in [1, b[$$

$$m = \sum_{i=1}^{+\infty} a_i.b^{-i} = a_0, a_1 a_2 a_3 ... a_i ...$$

$$\text{with } a_i \in \{0, 1, 2, ..., b-1\} \text{ and } a_0 \neq 0$$

To represent real numbers, computers use a representation derived from this writing.

IEEE-754 Floating Point Standard.

Representing a real number on computer means coding the triplet $\{\varepsilon, e, m\}$. Since 1985, the IEEE-754 standard has been defined and used by most of the computer constructors. Computers use the base 2, meaning that e and m are calculated in the following way [3] :

$$e = \sum_{i=0}^{p} b_i.2^i \text{ and } m = \sum_{i=0}^{+\infty} a_i.2^{-i-1} \text{ with } (a_i, b_i) \in \{0, 1\}$$

The sign ε is coded with s, one single bit that is 0 when the number is positive, and 1 otherwise. The exponent is coded with 8 bits for single precision, and with 11 bits for double precision. In order to avoid negative exponent, the representation is biased with $2^7 - 1$ for single precision, $2^{10} - 1$ for double precision. That lets 23 bits for the mantissa for single precision, 52 for double precision. In normalized writing, a_0 is not represented as it always is 1.

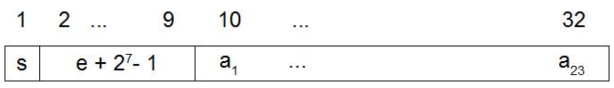


Figure 2.1: IEEE 754, Floating point representation for single precision

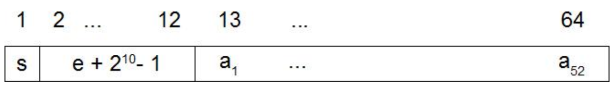


Figure 2.2: IEEE 754, Floating point representation for double precision

Only a finite ensemble of numbers can be represented so is limited. This ensemble is narrow-minded and discrete, which leads to some limitations. This ensemble is written as (F) in the following

Limitations of the representation

As the number of available bits to represent a number is limited, only numbers with defined properties can be exactly represented : the exponent have to be between -127 and +128 (between -1023 and +1024 for double precision), the mantissa in base 2 should contain only zeros after the 23th bit (after the 52th for double precision). Not every real number has such properties, which explains why they have to be approximated and rounded to be represented in this ensemble.

2.1.2 Rounding errors

Due to the finite representations of real numbers by computers, numbers often have to be represented by an approximation as explained in the previous section. Different rounding modes can be used.

Different rounding modes

When the real values can not be represented precisely using Floating Point numbers, rounding is used. The IEEE 754 standard defines four rounding modes. The exact representation of the real number x is $\varepsilon.b^e.m$ with $m = \sum_{i=0}^{+\infty} a_i.2^{-i-1}$. x' is the approximation of x, written with $x' = s.b^e.m'$ and $m' = \sum_{i=0}^{+23} a'_i.2^{-i-1}$. s is the representation of ε [3] [6].

- Rounding towards zero :

  $a'_i = a_i \; 0 \leq i \leq 23$

  $a'_i = 0 \; i > 23$

- Rounding towards nearest :

  $a'_{23} = a_{23} + a_{24}$

  with carry propagation (the exponent may be increased by one)

- Rounding towards +infinity :

  $a'_{23} = a_{23} + \bar{s}$

  with carry propagation ($\bar{s}$ is the logic negation of s)

- Rounding towards -infinity :

  $a'_{23} = a_{23} + s$

  with carry propagation

This rounding is used for each value assignment, and after each elementary operation.

Rounding errors in computation

The operands and the calculated results of each operation might be rounded to be represented in floating point format. This leads to cascading rounding errors that influence the result. This result X might differ from the exact result x. The resulting error can be defined in two ways [6] :

- the absolute error :

  $X = x + e_a$ with $e_a \in \mathbf{R}$

- the relative error :

  $X = x(1 + e_r)$ with $e_r \in \mathbf{R}$

The second representation can be used to calculate the number of significant digit, which is defined as follow [3] :

$$C_{X,x} = log_{10} \left| \frac{x + X}{2.(X - x)} \right| \; = \; log_{10} \left| \frac{x.(2 + e_r)}{2.e_r} \right|$$

This definition corresponds to the intuitive idea of the number of exact decimal digits between two numbers.

Floating point numbers and relative error

$x \in \mathbf{R}^*$ can be represented as following :

$x = \varepsilon.m.b^e$ with $1 <= m < b$

Representing x with a floating point number means choosing a mantissa M with p digits such that :

$X = \varepsilon.M.b^e$ .Then we have : $X = x.(1 + b^{-p}.\alpha)$

$b^{-p}\alpha$ is the relative error. α is called the normalized relative error. While rounding numbers, the error might differ with the used modus :

- towards to nearest : $\alpha \in [-0.5, 0.5[$
- towards zero : $\alpha \in [0, 1[$
- towards plus infinity : $\alpha \in [-1, +1[$
- towards minus infinity : $\alpha \in [-1, +1[$

2.1.3 Propagation of rounding errors

As rounding errors might occur at every step of computer calculations, the final result is effected by rounding error propagation due to approximations.

Overflow and Underflow

As previously exposed, a number is represented with a mantissa, an exponent and a sign (see 2.1.1). For single precision, the exponent is represented with 8 bits. It means that numbers with an exponent greater than 128 can not be represented. Such cases are called "overflow". Similarly, exponents under -127 cause "Underflow". For double precision, these phenomenons occur with exponents greater than 1024 or smaller than -1023. In order to denote such problems, the maximal value of the exponent is used to represent these exceptions. Thus, for single precision, the exponent 128 (binary : 11111111 considering the bias) in correlation with a mantissa consisting only of zeros represents plus or minus infinity (the sign still have to be considered). When this exponent is used with a non-zero mantissa, it is used to signal the exception "NaN" (Not a Number). Such exception will naturally propagate into calculations because there is no way to go back to the exact mathematical value. This leads to indubitably marked loss of accuracy in the result.

Cancellation

Due to the chosen representation of numbers, a great loss of accuracy might happen with single operations. The cancellation is one of those situations. It happens while adding x to y with $x \approx -y$.

$$X + Y = x(1 + e_r^x) + y(1 + e_r^y) = (x + y)(1 + \frac{x}{x + y} e_r^x + \frac{y}{x + y} e_r^y)$$

As the term $x + y$ is close to zero, the relative error of the result increased significantly in this operation.

As an example, take $x = 3.1416$, $y = -3.141592654$. The exact result should be $x + y = 7.346 * 10^{-6}$. Now y as been represented in single precision with a relative error of $9 * 10^{-5}$: $Y = -3.1415$. x is exactly represented : $X = 3.1416$. The calculation returns : $X + Y = 1.10^{-4}$. The relative error is now approximately 13. In many applications, such a large relative error makes the computed result meaningless, although the approximation of the operands has not been so bad .

Conditional Branches

We have seen that the accuracy of some computed results is highly dependent on the input data. It should now be considered in correlation with conditional branches in a program. The following code illustrate the problem :

```
IF A==B THEN
...
ELSE
...
END IF
```

Comparing A and B is similar to calculating $A - B$ and using the sign of the result. As we have previously seen, this calculation might result in an inaccurate result. This might then change the behaviour of the program, and the result might be really different from what is expected using exact mathematics.

2.2 Numerical Accuracy analysis

2.2.1 Overview of numerical accuracy analysis methods

The influence of the chosen representation of numbers on computer computations have been known for while and several methods have been developed to estimate the accuracy of the computed result. These methods are reviewed in [3].

Regressive Analysis

Regressive analysis considers the computer result as the result of the same algorithm, but with disturbed data. This approach works with arithmetic of real numbers. It is working particularly well with linear algebra algorithms. The inconvenient is that it completely depends upon the used algorithm that as to be studied to used partial derivative of the result with the data and partial results. It returns an uprate of the global error of the result. This approach have been started by J.H. Wilkinson ([12] resumed in [3]).

Direct Analysis

This kind of analysis consists of uprating the rounding error at each step of the algorithm. As it carefully follows the algorithms, this approach can provide fine uprate of the error. It sometime provides good results and is often easy to implement. F. Stummel showed that the accuracy of the result of the Gaussian elimination depends on the precision of the pivot ([11] used in [3]). It can also help to see where the accuracy loss can happen. The disadvantage of this method is that it does not provide an estimation of the accuracy of the result, but only an uprate of the error.

Interval Arithmetic

This approach consists of calculating the interval in which the result of each operation is contained at each step of the program. The principal interest of this method is that it is based on a solid mathematical algebra. It makes sure that the the exact mathematical result can be fund in the computed interval. The inconvenient of this method is that it usally overestimates the error. Indeed, it does not take into account that rounding error may compensate, and that it is affected by the dispersion effect. The dispersion effect is the following effect :

$$\forall X \in \mathbf{F}^*,\ X - X = 0 \text{ and } X/X = 1$$

But in interval arithmetic :

$X = [1, 2]$ implies $X - X = [-1, 1]$, $X/X = [0.5, 2]$

This phenomenon explains why this method can not be used easily.

Probabilistic approach

The idea of the probabilistic approach is to execute the same program several times while propagating the rounding errors differently. Thus several different results are obtained. The common part of these results give an estimation of the accuracy, the rest is the non significant part. It is based on random arithmetic to generate different propagation of rounding errors. It has been showed that some properties of exact arithmetic lost in computer arithmetic are restored [5].The idea is that every exact result r of an arithmetic operation can be surrounded with two successive floating point numbers R^+ and R^-. The random arithmetic consists of randomly choosing one or the other. Thus, executing several times a same program results in as much computed results. The aim is not to get a better result, but to be able to give an estimation of the number of exact digits using these results.

2.2.2 Discrete Stochastic Arithmetic

One of the methods to estimate the accuracy of a computed result consists in using random errors to provide this information. This is based on properties of the stochastic arithmetic.

Random rounding

If we represent two real numbers x_1 and x_2 as floating point numbers with X_1 and X_2. We have then $X_1 = x_1 - 2^{E_1 - p}.\varepsilon_1.\alpha_1$ and$X_2 = x_2 - 2^{E_2 - p}.\varepsilon_2.\alpha_2$, where p is the number of bits of the mantissa, E the exponent and ε the sign. $2^{-p}.\alpha$ is the absolute error when coding the mantissa ($\alpha \in [0, 1[$). Than with x_3 the result of operation and X_3 the result of the computer addition, we have :

- addition :

$$X_3 = X_1 + X_2 = (x_1 + x_2) - 2^{E_1 - p}.\varepsilon_1.\alpha_1 - 2^{E_2 - p}.\varepsilon_2.\alpha_2 - 2^{E_3 - p}.\varepsilon_3.\alpha_3$$

- subtraction :

$$X_3 = X_1 - X_2 = (x_1 - x_2) - 2^{E_1 - p}.\varepsilon_1.\alpha_1 + 2^{E_2 - p}.\varepsilon_2.\alpha_2 - 2^{E_3 - p}.\varepsilon_3.\alpha_3$$

- multiplication :

$$X_3 = X_1 * X_2 = (x_1 * x_2) - x_2.2^{E_1-p}.\varepsilon_1.\alpha_1 - x_1.2^{E_2-p}.\varepsilon_2.\alpha_2 - 2^{E_3-p}.\varepsilon_3.\alpha_3 - \mathcal{O}(2^{-2.p})$$

- division :

$$X_3 = X_1/X_2 = (x_1/x_2) - 2^{E_1-p}.\varepsilon_1.\alpha_1/x_2 - 2^{E_2-p}.\varepsilon_2.\alpha_2/x_1 - 2^{E_3-p}.\varepsilon_3.\alpha_3 - \mathcal{O}(2^{-2.p})$$

While doing it with the several basic operations, it can be showed that the computer result R of a classical elementary operation can be written :

$$R = r + \sum_{i=1}^{s_1} g_i(d).2^{E_i-p}.\alpha_i.\varepsilon_i + \mathcal{O}(2^{-2.p})$$

where : $E_i, \alpha_i, \varepsilon_i$ are the exponent, rounding and sign of intermediate values. $g_i(d)$ are quantities independent of the arithmetic and of the accuracy. P is the position of the last binary digit. r is is the exact mathematical result. s_1 is the number of operands of the operation. With random rounding, only the last bit can be changed for one operation. Each single operation results of the addition of the following term : $+2^{E-p}.\varepsilon.h$ Thus the result of several operations becomes :

$$R = r + \sum_{i=1}^{n} u_i(d).2^{-p}.(\alpha_i - h_i) + \mathcal{O}(2^{-2.p})$$

where : $u_i(d)$ are quantities depending on data and the algorithm. α_i, h_i are the rounding and the perturbation of intermediate results. n is the number of elementary operations. $u_i(d)$ is independent of α_i and h_i. α_i, h_i are random variables, and depend of anterior h_i These calculation are done in [4] or [2]. By identifying the α_i's to independent identically distributed random variables, we can simplify this writing.

Accuracy estimation

It has be shown by R.W. Hamming and D.E. Knuth (and used in [4]) that it can be assumed that the distribution of the mantissa is a logarithm distribution. This proof is also examined in [4]. A. Feldstein and R. Goodman proved that the distribution of the trailing digits converges to to the uniform distribution when the rank of the rounded digit tends to infinity (the proof is used in [4]). The α_i's can then be modelled by uniformly distributed random variable on $[0, 1]$ while chopping, on $[-1/2, 1/2]$ for rounding. A uniform distribution is then also assumed for the $\alpha_i - h_i$. As the terms in $2^{-2.p}$ can be neglected, a computer result R can be modelled by the random variable Z defined by :

$$Z = r + \sum_{i=1}^{n} u_i(d).2^{-p}.z_i$$

where $u_i(d)$ are constants and z_i independent identically distributed random variables.

With this modelling the exact result r of the operation is the mean value of this random variable as the z_i's are centred due to the choice of the h_i. As the distribution of Z is a Gaussian distribution, the Student test can be used to determine the accuracy of the result, as described in equation (2.1) :

$$(2.1)\quad C_{\overline{Z}} = \log_{10}\left(\frac{\sqrt{N}.|\overline{Z}|}{\sigma.\tau_\beta}\right) \text{ with } \sigma^2 = \frac{1}{N-1}.\sum_{i=1}^{N-1}(Z_i - \overline{Z})^2$$

τ_β is the value of the Student's distribution for $N-1$ degrees of freedom and a probability level $1-\beta$. In practical, we take $\beta = 0.95$ and $N = 2$ *or* 3. $C_{\overline{Z}}$ is the number of significant digit in base 10 of the computed result

Validity of the stochastic arithmetic

The validity of the previous calculations is granted only if some hypothesis are true. These are :

- Signs and exponent of intermediate results are true. This is only a theoretical problem : in practice the central limit theorem and the robustness of the Student's test ensure that it is never a problem.
- Rounding errors α_i are indeed independent centred equally distributed random variables : due to the introduction of the h_i, the α_i are indeed independent random variables. In practice, then are never really centred. J.-M. Chesneaux [4] have showed that in practice, this is never completely true and that there always is a bias of several σ for $\overline{Z}$, but that error never implies more that one digit of error for the accuracy measure.
- The approximation in first order in 2^{-p} is true : while using addition or subtractions, this can never be false. However in multiplications or divisions, the relative error needs to be smaller than one to verify this approximation : $(e_r = \frac{e_a}{x} << 1$. In practice, such numbers should be detected as soon as they appear during calculation and noticed to the user. They are called informatical zeros, as they are value that the computer can not distinguish from zero.
- No Overflow or Underflow happens : when an overflow or underflow appears during calculation, this one can not be continued with a real meaning. This should also be checked during the execution.

As we can see, it is theoretically possible to calculate the accuracy of a calculated result using stochastic arithmetic, but for that some condition have to be checked during the execution.

2.2.3 Stability analysis

Parallel execution

As we have seen previously (2.2.2), there are several verifications that has to be done at each several steps of the execution of the program. For this reason, it is needed to check for Overflow and Underflow at each step at the execution, but also to compare the result of each execution before any multiplication or division to check for informatical zeros. Informatical zeros are introduced to represent the real zero or numbers with no significant digits.

$$\forall i\ R_i = 0\ or\ C_{\overline{R}} \leq 0$$

Every arithmetical operation has to be run N times before going to the next. For each of these operations, the random rounding has to be applied. With the results of these N runs, it is possible to detect informatical zeros and then to detect operations that could implies meaningless results. There are two ways to realise it. One is to run each operation several time with separate operands that has to be stored and restored each time. The second possibility is to have several executing units running simultaneously and exchanging data whenever is is needed to make some check.

Accuracy loss

Due to the representation of numbers, computer arithmetic is more vulnerable to the changing of the order of operations. Here is an example :

$10^{30} - 10^{30} + 1 = 1$ in normal math

$(10^{30} - 10^{30}) + 1 = 1$ executed in this order for single precision

$10^{30} + (-10^{30} + 1) = 0$ executed in this order for single precision

Thus, we can see that a same algorithm written differently might provide a different result. For this reason, it is an interesting problem to check for brutal loss of accuracy during the execution of a program, in order to tell the user where some improvement might be useful in the code to provide more accurate results. The idea of an accuracy analyser based on the stochastic arithmetic is that it can provide the accuracy loss for any floating point variable at any step of the execution. When computing the accuracy of the final result with stochastic arithmetic, the intermediate computations already enable to compute accuracy of intermediate results.

Unstable execution

As we have seen, there are some verifications that have to be done at each step of the execution of the calculations. Overflow, Underflows and informatical zeros can be gathered during the execution and noticed at the end in order to tell the user the accuracy (or inaccuracy) of the calculated result. However, some exceptions generated by such an analyser might bring some problems. Indeed informatical zeros might change the result of a comparison used for conditional branching, and thus change the execution of the program. For some examples, iterative algorithm use tests to stop. The propagation of rounding errors might disturb the computation of the test condition. Using an informatical zero to stop iterations is a problem. Such difference should be recognized and signalled to the user as soon as they appear. Thus, an accuracy analyser based on stochastic arithmetic as to make some verifications during the execution of the program, and might interrupt the normal execution of this one.

3 Structure of the Analyser

3.1 Introduction to the dedicated Hardware for numerical accuracy analysis

Advantages of dedicated hardware

Estimating the accuracy of a computed result with stochastic arithmetic means computing each intermediate result of a floating point operation several times using random rounding. It is possible to write libraries applying these principles. The steps for each floating point operations become then [1] : randomly choose the rounding mode, execute the operation N times, compute the average result, compute the number of significant digits, detect informatical zeros and control order relations.

Such a software implementation of discrete stochastic arithmetic implies a significant increase of the computational time. Realising the implementation on hardware simplifies the utilisation because the program does not have to be compiled with modified libraries, and reduces the computation time. Indeed, it is then possible to execute some operations in parallel.

3.1.1 Discrete Stochastic Floating Point Unit

In order to calculate the accuracy of a computer result using the stochastic arithmetic, modified floating point unit is used.

Power PC and APU

In order to provide efficient means to calculate with floating point numbers, current processors are often using so called "Floating Point Unit" (FPU). In such an architecture, the processor uses a special interface to communicate with this entity. It sends command to this external unit and and then wait result to be available. The Floating Point Unit is then in charge of all operations with floating point numbers : addition, subtraction, multiplication, division, square root, conversion and comparison. A FPU designed with the principles of stochastic arithmetic is not the common case, it has to be integrated separately. In our case, it is realized on a FPGA (field-programmable gate array) containing PowerPC 440 processors that can use a custom FPU. Then the communication between them have to use a bus called the Fabric Co-Processor Bus (FCB). In order to use this bus, the processor uses the so called Auxiliary Processor Unit (APU). This structure allows the user to use a provided FPU or to integrate its own using a FPGA board. The corresponding structure is depicted in figure : 3.1

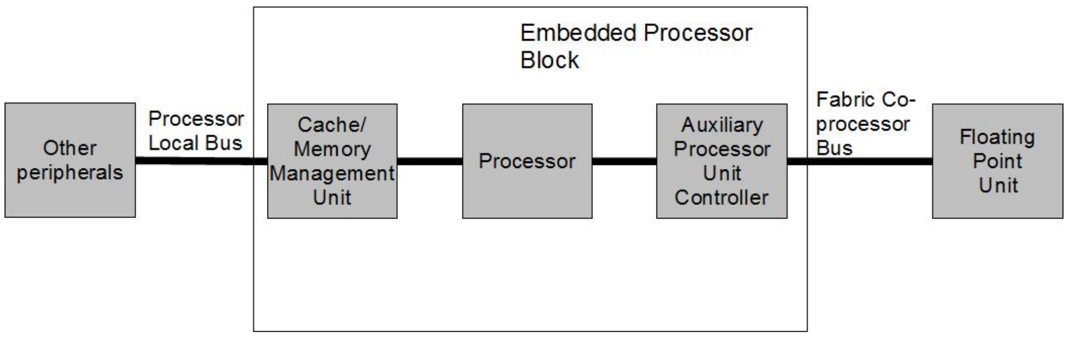


Figure 3.1: The processor communicates with an external Floating Point Unit usinf a Fabric co-processor Bus

Floating Point Unit

The Floating Point Unit is separated from the processor and communicates with it using the signals provide by the Fabric Co-processor Bus. The FPU has 32 floating point registers. The operations that are supported by the FPU are addition, multiplication, division, square root, absolute value and conversion to integer or from integer to floating point. These operations are pipelined in order to provide a maximum throughput. Comparisons between floating points is also possible. The calculation units have to provide some extra features in order to support discrete stochastic arithmetic. A view of the developed FPU can be seen on Figure 3.2. The structure of the DSFPU is described in details in the article [8].

3.1.2 Random Rounding

Discrete stochastic arithmetic is based on the introduction of random rounding for each floating point operation. This has to be implemented inside the Floating Point Unit.

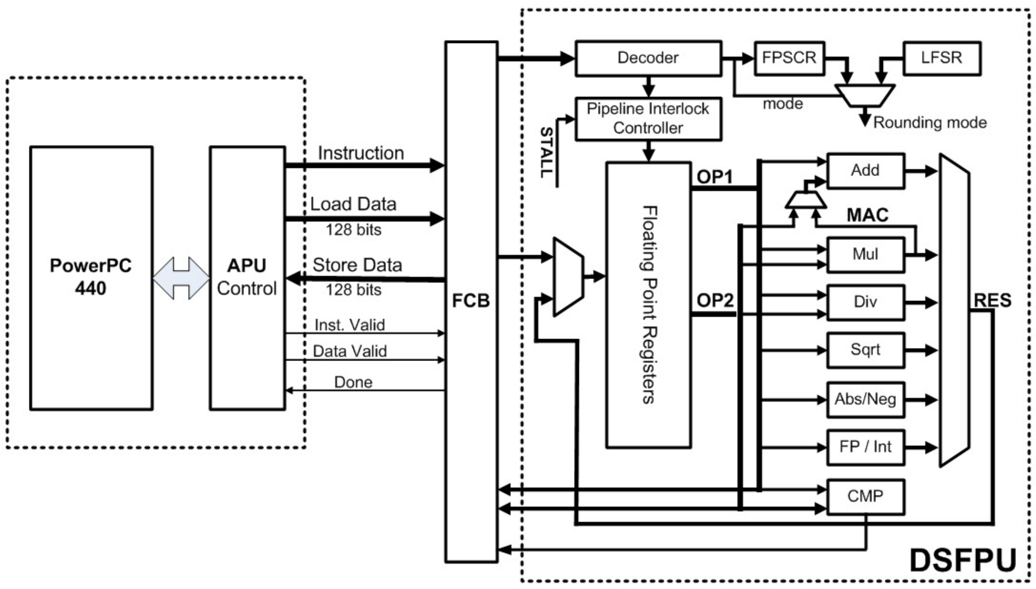


Figure 3.2: Discrete Stochastic Floating Point Unit and its processor

Rounding methods

The execution units of the Floating Point Unit are able to provide results using one of the 5 following rounding modes :

- rounding to nearest (IEEE-754)
- rounding towards zero (IEEE-754)
- rounding towards plus infinity (IEEE-754)
- rounding towards minus infinity (IEEE-754)
- random rounding : either towards plus infinity or towards minus infinity. Both possibility are equally probable.

On traditional CPUs, switching of the rounding mode might have influence on several stages of the pipeline. As a consequence, the pipeline has to be flushed. However, in the architecture of the STochastic Floating Point Unit(STFPU), the random rounding mode is a switch between two modes without flushing the pipeline, in order to be used for stochastic arithmetic.

Random Rounding

The random rounding mode switches between rounding-up and rounding-down randomly. To do so, it has to use a bit that is randomly generated. A Fibonacci-style Linear Feedback Shift Register (LFSR) is used as Pseudo Random Number Generator. It consists of a 32 bits register that is shifted and filled with a logical combination of some of the bits. The 32th, 31th, 30th and 10th bits are used to compute the output that is also used to fill the shifted register. The repeating cycle have a length of $2^{32} - 1$. The output is "0" with a probability of 50,572%, "1" with a probability of 49,428%.

The integration of the random rounding modus can be seen on the Figure 3.2 (top-right).

3.1.3 Parallel Processing Blocks

The second special feature of a discrete stochastic floating point unit is the stability analysis while the program is running. To do so, it has to use the N computed results of the same program with different rounding error propagations, what is realised with parallel processing blocks.

Distinct Memories

As we previously saw, the different processing blocks have to run independent random rounding. In order to do so, every floating point variable that is used has to be stored separately in each processing block. However, it might also happen that such a variable is converted to an integer, used as an integer and the result then used again as a floating point. Thus integer variable should also be handled separately. This example shows that every single piece of memory should be duplicated in order to be sure that the several processing blocks running the program are effectively generating independent propagation of rounding errors. So the processing blocks should not only consists of parallel running FPU, but of real blocks consisting of processors, FPU and memory. However, these blocks needs some communications in-between.

Synchronization and Communications

In order to compute the accuracy of a computed result, several processing blocks have to run the same program going through the same data-path. During the execution, random rounding is applied independently in each block. However, if the block are completely independent, there is no easy way to ensure that the same branch has been taken for each conditional branch. This phenomenon has to be watched in order to avoid meaningless results. Thus each time a comparison between two numbers occurs, every block has to be stalled after computing the result until all of them finish. Then, the average has to be compared, and a unique decision is taken. It informatical zero appear at this point, no decision can be taken. In such a case, the user should be warned.

Resulting Architecture

The processing blocks consisting of processors, memory and Floating Point Unit running the program are completely independent except for the synchronization part that have to use the different results to avoid different decisions to be taken for conditional branches. The resulting architecture can be seen in the Figure 3.3 :

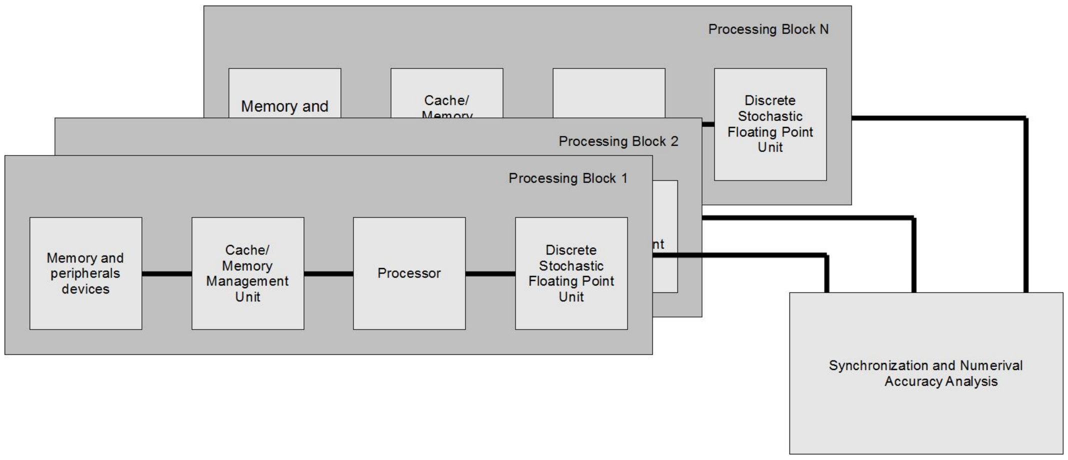

Processing Block N
Memory and
Cache/ Memory
Discrete Stochastic Floating Point Unit
Processing Block 2
Processing Block 1
Memory and peripherals devices
Cache/ Memory Management Unit
Processor
Discrete Stochastic Floating Point Unit
Synchronization and Numerival Accuracy Analysis

Figure 3.3: Architecture of a system including several processing blocks used for accuracy estimation based on discrete stochastic arithmetic

In order to calculate the computed accuracy, a modified debugger is used to run the program on the different processing blocks and the collect the necessary information to display the accuracy.

3.2 Utilisation of the dedicated hardware

3.2.1 Running One program on several Processing Blocks

Shared Data

To calculate accuracy, the processing blocks have to run through the same calculations with the same initial data, that get then perturbed by the propagation of errors due to random rounding. The variables used during the calculations might be really different between several blocks during the calculations, but it has to start with exactly the same code. To run the program, a binary file has to be loaded into the memory and then the processor started. Here the same binary file containing the same program with the same values have to be loaded for all the processing blocks.

Simultaneous start

The different processing blocks running in parallel are stalled at every operation that needs synchronisation until the others reach the same elementary operation. In order for the stability analyser not to stall the hardware, all the processing blocks have to run simultaneously. Usually when executing a program on a processing block, it is started and it is then expected to reach a normal stop by itself. The processing blocks used here can not run independently for each other, that is why they have to be all started before expecting something to happen.

3.2.2 One execution Flow

Common Breakpoints

The accuracy analyser is designed to run a program as normal, but retrieve not only the results of the computations, but also some information about the accuracy of these results. An expected behaviour of an analyser is to look like a debugger : a program is chosen, a target set to run it and some breakpoints chosen to get intermediate results and see the execution flow. Then, the analyser has to execute the chosen program on the different processing block in parallel, and after each of the processing blocks reaches the same breakpoint as defined in the program, it can recover the different values computed for a same variable and thus provide information about the accuracy of the result based on equation (2.1). thus it is necessary to make sure that all processing blocks get stopped at the chosen breakpoint.

Same inputs

The different processing blocks are running an unique program, and are expected to reach the same breakpoint. However, loading the same source code and set the same breakpoints is not sufficient for them to follow the same execution-flow. Indeed, during the execution of the program, peripheral devices might be used to provide input values and might have influence on the computed results. For example, it is the case for variable set after the start of the program using some input in a console. These inputs have to be identical for all processing blocks in order for them to work as expected.

3.2.3 A state-of-the-art debugger

To see what happens inside a program during its execution, a debugger is usually needed. In order to get the information where accuracy of computed values is lost, a modified debugger is meaningful: it allows the user to define breakpoints and then to check the accuracy of any intermediate results.

Multiple connections

In order to preserve resources, the Board is only used to run the program and no Operating System. It is then required that the debugger operates remotely.

As several processing blocks are used, the modified numerical accuracy debugger has to be connected to each of them : the program has to be loaded, messages exchanged to set breakpoints, to start a processing block, to signal that a breakpoint has been reached, and to read the values of variable when the processing block is stopped. Normal debugger are only meant to be connected to one processing block, here it has to be able to use several and communicate independently with them.

Duplicated functionalities

The user should not see any difference in the debugging process by using the proposed numerical accuracy analyser.

As the analyser only have to show the unique execution flow of the program, the user want to set the breakpoints on the code, and the analyser then have to set the real breakpoint for each of the processing blocks in order to execute the same portion of code. The user also wants to start the program once, the analyser has then to start each of the processing block as if there where only one processing block. Thus, all the processing blocks are running the same program, and can be synchronized.

3.3 Numerical accuracy analysis

3.3.1 Several Calculated Results

The accuracy analyser consists of a debugger modified to execute several instances of a program on several processing blocks. After executing the program, each processing block computed a different version of the value with different rounding error propagation.

Different Values

The different processing blocks are loaded with the same program. Thus, the same variable is stored once on each of these blocks. Then the program is run, until it reaches its end or a breakpoint set by the user. As each of these blocks applies random rounding after each floating point operation, the computed results are different in each of the processing blocks. The synchronization part developed on hardware ensures that the same breakpoint in the program is reached in every block, by avoiding different decision to be taken in conditional branches due to the different values of variable in the instances. The debugger have to collect them and calculate the average value and its numerical accuracy (see equation (2.1)).

The analyser is expected to display the computed values and the number of significant digits of this value. These informations is computed by using the different values computed for a single variable.

Different types of computed values

The processing blocks do not only consists of Floating Point Units designed to apply the principles of stochastic arithmetic, but also a normal processor and memory. Then every variable is not always a floating point number represented with the IEEE-754 Standard. Although random rounding is only applied in the custom floating point unit, floating point variables are not the only one affected by the mechanism. Indeed, it is possible to imagine a program that uses a written value as input. This value is at the start a character sequence. It is then translated into a floating point number. Then some calculations are made and a result is computed. This result also is a floating point number that gets converted again into a chain of character. The resulting character sequence would most surely differ between the several processing blocks (the computations done in floating point arithmetic only have to be large enough). This example can not really happen has the synchronisation part prevent conversion form floating point to integer to return different values. However, it shows that the types of variables have to be taken into account.

As the previous example shows, the processing blocks computes different values due to random rounding. The accuracy analyser has to provide the user with only one value for each of the variables. It has then to retrieve all of them and compute the accuracy for each of them. The hardware developed only provide a way to calculate the accuracy of floating point numbers. Thus, the analyser have to know the type of all the variable in order to be able to compute and provide the accuracy only for floating point results.

3.3.2 Retrieving the accuracy

The numerical accuracy analyser is meant to provide the numerical accuracy of results. However numerical accuracy only does not make sense in every case. These case have to be recognized by the analyser.

Retrieving the types

As previously exposed, every single processing block running the program contains an instance of each variable, whatever its type is. When the processing blocks stops at the end of the program or at a breakpoint, the analyser retrieves the addresses of all variable in a symbol file. This symbol file contains informations about the type of a variable and gets the necessary information to read its value. Such a file has to be generated for every processing block in order for the accuracy analyser to utilize them to compute the accuracy of the variable for which it is defined. It is then possible for the analyser to read the different values computed for a single variable and inform the user if they are different and what there values are.

Return a useful value

The aim of the accuracy analyser is to display the accuracy of every floating point variable. A normal debugger already display the value of each of them. The accuracy only have to be added nearby.

The accuracy is simply computed using the formula :

$$C_{\overline{Z}} = \log_{10}\left(\frac{\sqrt{N}.|\overline{Z}|}{\sigma.\tau_\beta}\right) \; with \; \sigma^2 = \frac{1}{N-1}.\sum_{i=1}^{N-1}(Z_i - \overline{Z})^2$$

with Z_i the values of the N different processing blocks.

Conclusion

It has been shown that is is possible to modify a normal debugger in order to run a single program on several processing blocks developed to apply the principle of stochastic arithmetic. Thus, the different values computed for a single variable can be retrieved and used to compute the accuracy of any floating point variable.

4 A numerical accuracy analyser based on GDB

4.1 GDB as initial debugger

The accuracy analyser is a debugger with some additional functionalities. Developing it from scratch would have represent an enormous amount of work. However, it is possible to use the sources of an existent debugger as a starting point, and to modify it to make an accuracy analyser out of it.

4.1.1 Integration with hardware Part

The numerical accuracy analyser has to work with the special hardware introduced in section 3.1. As it is only a modified debugger, it has been decided to use one that already works with the developed hardware.

Integration with existent tools

The Xilinx tolls does not only facilitate the building of a system implementing the principle of stochastic arithmetic, but it also provides the necessary tools to run a program on it. Indeed the Xilinx Platform Studio SDK (Software Development Kit) tool provides a compiler for the developed structure. The code can be written in C and then compiled for the hardware. It also provides a header file defining constants for the addresses of the different peripherals.

After compiling the source code into a .elf file, the Xilinx tools provide a mean to load it on the board and start the execution. It allows the user to open a GNU debugger (GDB) server for each processor at different TCP port number. These GDB servers provide then an interface for debugging that can be used by the debugger GDB. The GDB program does not even have to be on the computer used to program the board and providing the GDB server interface. It only needs the source code and a copy of the executable file. An illustration of the strucure can be seen on Figure 4.1.

4.1.2 Developing possibility

As previously mentioned, the tools used to developed the hardware system already provide a debugger. This debugger had to be adapted in order to realize an accuracy analyser. It has been possible due the interesting properties of the debugger used.

License status

The normal GDB debugger only uses one target at a time to run a program. To be able to use the two processors of the developed system to run in parallel a program, it has been necessary to modify the existing debugger. It is not always possible for a programmer to modify a program, especially when the source code is not available. However, GDB was released under the GNU General Public License (GPL), which means that the source code is available for every user and can be modified for its own use. Naturally, it also means that the derived works can only be distributed under the same license terms.

This last property had its importance. Indeed, the compiled source code targets a custom hardware that differs from the others. It means that the target is not a common computing system. Some of its properties have to be used by the debugger to works properly. The debugger had already been adapted by Xilinx and the resulting software is also under the GPL and available on internet at the address : http://www.xilinx.com/guest_resources/gnu/index.htm. The documentation is also available [7] . This software as been the start of the accuracy analyser.

Programming language

GDB was first written by Richard Stallman in 1986. John Gilmore maintained if from 1990 to 1993 while he worked for Cygnus Solutions. The GDB Steering Committee which is appointed by the Free Software Foundation now maintains it. It is mostly written in C and is composed of several hundreds of files. Most of these files are written in C. The necessary makefiles and scripts used to compile it are provided with the source code. It was then possible to adapt the debugger for accuracy analysis.

Documentation

As the program is now a few years old, and is maintained by a lot of persons, a lot of documentation on it is available. The official web site regroups a lot of general information about the program : http://www.gnu.org/software/gdb/. In particular, the part of the web site dedicated to the internal work of GDB provides explanations of the complex internal organizations : http://sourceware.org/gdb/current/onlinedocs/gdbint/. Two manuals are also available, one for normal use [9], one for developer use [10].

4.2 software debugging using Remote connections

4.2.1 Client/Server Structure

As previously explained, the FPGA board is only used to run the program. The accuracy analyser is running on a computer using a remote connection to a GDB server. This GDB server is provided by Xilinx tools running on the computer used to program the board. The different mechanisms that such a structure implies are explained here.

GDB Remote Protocol

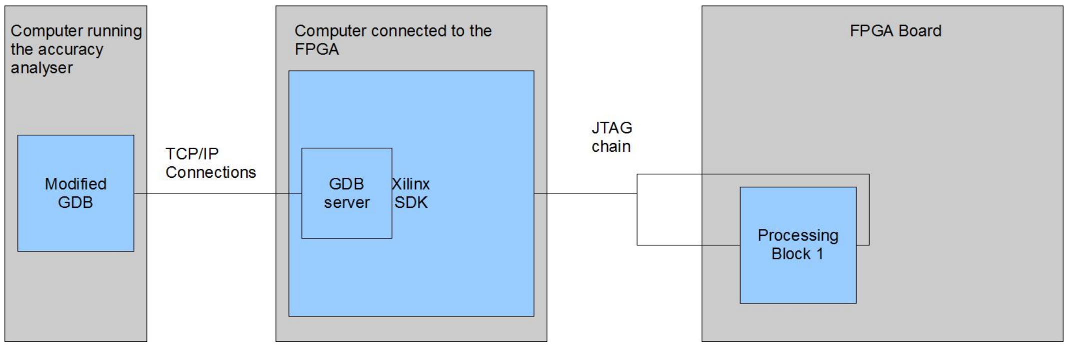


Figure 4.1: Architecture involved to debug a program on a FPGA board

The GDB Remote Protocol is used when a program has to be debugged on a machine that can not run GDB in the usual way. It works over TCP/IP interfaces. It uses a generic serial protocol that is specific to GDB but not to any particular target system. A GDB server has to be started first, and then a GDB can act as a client and connect to it. An usual structure while debugging a program on a single processing block can be seen on 4.1. Here is an example of a connection between GDB and a GDB server running a program called hello_world at the address 192.168.2.3 and port 1234 :

Initial server side :

```
server$ gdbserver localhost:1234 ./hello_world
Process ./hello_world created; pid = 1800
Listening on port 1234
```

On the client side :

```
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
Modified by Sylvain Burlet in 2010.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"
(gdb) target remote 192.168.2.3:1234
Remote debugging using 192.168.2.3:1234
(gdb) run
Starting program: /home/burlet/Desktop/prgm/hello_world
hello_world !
Program exited normally.
(gdb)
```

The server side shows then :

```
server$ gdbserver localhost:1234 ./hello_world
Process ./hello_world created; pid = 1800
Listening on port 1234
Remote debugging from host 192.168.2.2
Killing inferior
server$
```

GDB can be used for different sorts of targets, in particular targets can be a board with microblaze processors or PowerPC processors. In these cases the description of the target has to be previously known and cross-compiler taken into account. For the GDB used as analyser (the client), the targets seems to be the processing block on the board, but it only happens through a GDB server interface running on the computer used to program the board. These implies different versions of the program. It also allows to use ".elf" files as source code. Then the last line of the informations about GDB can become :

- In case of a i686 processor running linux :

  ```
  This GDB was configured as "i686-pc-linux-gnu"
  ```

- In case of a microblaze processor, using the same computer to run the debugger :

  ```
  This GDB was configured as "--host=i686-pc-linux-gnu --target=microblaze-xilinx-elf"
  ```

- In case of a PowerPC processor, using the same computer to run the debugger :

```
    This GDB was configured as "--host=i686-pc-linux-gnu --target=powerpc-eabi"
```

The host is the computer running the client, in this can the analyser. The target has gives the type of processing block that is effectively used to execute the program. The GDB server is just an intermediate interface.

TCP Connection

As the previous example shows, the GDB client uses a TCP connection and does then not have to run on the same computer. The accuracy analyser needs to be connected to two processing blocks. It means that first two servers have to be started. Then the analyser has to connect to both of them. The two TCP connection are then established. It uses both connection to send the necessary command and retrieve the values of the variables. A simple version of an accuracy analyser would simply be to have two debuggers running, connect each one to a server, and tell both of them to execute the program. The two processing blocks on the FPGA would then execute the program and return the value of the variables. The two debuggers would each be able to display the value for the variable and the user would have to calculate manually the accuracy. The aim of modifying GDB is to integrate the two TCP connections in one debugger, automatically duplicate the commands and retrieval of values and display it.

4.2.2 Connecting the hardware platform

In the previous example, the TCP connection between the server and the client uses IP addresses. However, the FPGA used for the hardware part does not contain an Ethernet interface and therefore can not run directly used as GDB server.

JTAG chain

Indeed the FPGA is connected to a computer through an USB cable and a JTAG (Joint Test Action Group) interface. This cable is used for the connection of a computer to the Board. This computer uses Xilinx tools to interact with the FPGA. This JTAG chain is first used to configure the FPGA and then used to communicate with the two processing blocks : hardware breakpoints can be set, memory read or written. Access to all memories is granted, including registers. It can also be used to start a processing block.

GDB Server interface

The JTAG chain allows to configure and control the two processing blocks. Thus the Xilinx tools are able to use this connection to the FPGA to realise a more useful interface for debugging. Indeed Xilinx tools provide a software called XMD (Xilinx Microprocessor Debug) that runs on the computer connected to the board and offers a GDB server interface. As the GDB server interface is not architecture specific, it simplifies the debugging of the processing blocks. The complex architecture using the JTAG chain disappear behind the GDB server interface available. Two instances of XMD can use the same JTAG chain without interferences and thus the complex hardware system can be seen as two GDB server interfaces by the GDB modified to act as an accuracy analyser.

4.2.3 System Overview

In order to analyse the accuracy of a computed result, a complex system has to be build using an hardware system, a debugger and a complex system to allow them to interact. However, the analyser itself does not need all of the system to be developed.

Test Platform

The hardware part of the system is complex. The configuration of the FPGA have to be compiled, and such a structure has two processing blocks represent a lot of logic blocks . Then this code is used to programme the FPGA. After it two instances of XMD running with one single USB cable have to be started. It automatically provide then two GDB server interfaces for the processing blocks.

The analyser itself only needs to be connected at two GDB servers running the same program. Then it simply sets breakpoints, start the targets and retrieve the values without having to know what kind of hardware really executes the program. The architecture used by the target only has to be known for the interpretation of machine code. Thus all the complex system does not have to be build to develop the analyser. It is possible to run two normal GDB server with the same executable file on a computer and also run a modified version of GDB on this computer that connect to these two servers. Such a test architecture can be seen on the figure 4.2

This architecture has been used to develop the accuracy analyser. The only difference with the real system is that the two instances of the program used in the GDB servers are running completely independently when they would have to synchronize on the real architecture. It has to be taken into account during the development.

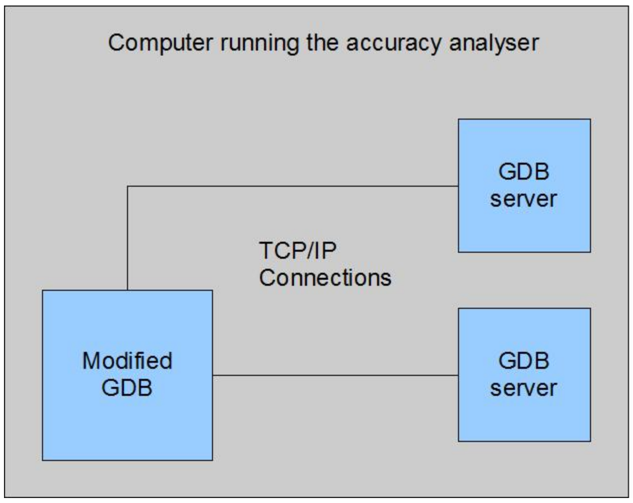


Figure 4.2: System used to develop the accuracy analyser

Real System

An easy system can admittedly be used to develop the accuracy analyser, but the real system still have to be build for the real application. First, the FPGA Board have to be programmed to implement the hardware system including two processing blocks running independently except for the synchronisation part between the two instances of the custom Floating Point Units. These two processing blocks are linked to a computer running XPS (Xilinx Platform Studio) through an USB cable connected two a JTAG interface. This computer then runs two instances of XMD, which opens two GDB server interfaces. The modified debugger used as accuracy analyser has then to be connected to these two interfaces through two TCP/IP connections. This architecture is showed on figure : 4.3

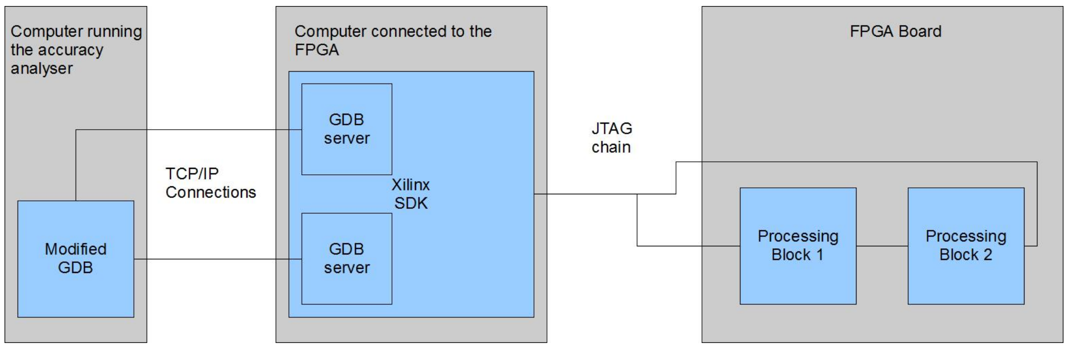


Figure 4.3: Complete overview of the system

Note : Although the modified version of GDB can connect to the two instances of the GDB server through TCP/IP connection, there is not real need for the modified version of GDB to run on a different computer than the one running the server, which result in the same architecture except that the TCP/IP connection is a connection to the local host.

4.3 Modified functionalities

4.3.1 Set a Breakpoint

One of the expected functionalities of the accuracy analyser is to be able to stop the execution when the program reaches a given instruction and check the numerical accuracy of the intermediate result. In order to do it, the processing blocks have to stop before executing this chosen instruction. This is called a breakpoint.

Nature of a breakpoint

The user writes code in a programming language. This code consists of a list of ordered instructions regrouped in functions. Some of these instructions are conditional jump : a function or an other can be used depending on the values of some variables. The value of these variables often is the result of previous instructions. In order for the user to be sure that the program is doing what it is expected to do, it is useful to be able to follow the execution of the program and to know the value of different variables when reaching a given instruction. Similarly, while using an accuracy analyser, the user want to know the accuracy of a variable when reaching an instruction. For example the user might want to know using which function is used to calculate c in the following code section. The * marks were a breakpoint would have to be set.

```
int main(void) {
   int a, b, c ;

   a = calculate_a() ;
   b = read_b() ;

*  if (a==b) {
        c = equal() ;
   } else {
         c = unequal() ;
   }
}
```

The code written by the user has to be compiled into machine code to be executed. Machine code is also a list of instructions regrouped as functions. Some of them allows to jump from a function to an other. The processor knows the current position of the execution-flow through a Program Counter, which is basically the index of the instruction that is executed. During the compilation the instruction written by the user are translated into instructions that the machine understands. It is then possible to match the instruction of the user with those of the machine code. This part also explains why different variants of GDB are used for different architectures. The previous example result on a i386-architecture in :

```
-0x80483e4 <main>: push   %ebp
-0x80483e5 <main+1>: mov    %esp,%ebp
-0x80483e7 <main+3>: and    $0xfffffff0,%esp
-0x80483ea <main+6>: sub    $0x10,%esp
-0x80483ed <main+9>: call   0x8048423 <calculate_a>
-0x80483f2 <main+14>: mov    %eax,0xc(%esp)
-0x80483f6 <main+18>: call   0x804843f <read_b>
-0x80483fb <main+23>: mov    %eax,0x8(%esp)
* 0x80483ff <main+27>: mov    0xc(%esp),%eax
-0x8048403 <main+31>: cmp    0x8(%esp),%eax
-0x8048407 <main+35>: jne    0x8048414 <main+48>
-0x8048409 <main+37>: call   0x804845b <equal>
-0x804840e <main+42>: mov    %eax,0x4(%esp)
-0x8048412 <main+46>: jmp    0x804841d <main+57>
-0x8048414 <main+48>: call   0x8048477 <unequal>
-0x8048419 <main+53>: mov    %eax,0x4(%esp)
-0x804841d <main+57>: mov    0x4(%esp),%eax
-0x8048421 <main+61>: leave
-0x8048422 <main+62>: ret
```

Setting a breakpoint for the user consists of deciding the instruction at witch the execution of the program should halt. It can then be determined with which value of the program counter the processor should stop.

Accuracy Analyser breakpoints

The previous paragraph explained what a breakpoint is in a normal debugger. It is the same in the accuracy analyser, but in this case there are not only one processing block. This means the breakpoint defined by the user is unchanged and it is translated into a line of its code. It is translated into a given instruction of machine code. However, this machine code is written into the memories of two different processing blocks. Thus, the description of the stop for the processor might be translated into different values of the program counter for the two different processing blocks. These values have to be stored by the analyser in order to set them properly for each processing block.

There are two ways to set breakpoints for the embedded PowerPC Processors. First, the breakpoints are set in dedicated registers and when the program register is equal to one of them, the processor stops. However, the number of this these registers is limited and thus software breakpoints are introduced. A software breakpoint is realised by replacing the given instruction by an illegal operation (for example dividing by 0). It causes an exception to be raised while reaching this instruction, and stop the processors. The instruction has to be restored before continuing the execution.

While running the program with breakpoints defined, an exception is raised and reported to the analyser. This one then have to wait until the second processor reaches the same breakpoint.

4.3.2 Run the program

A special Floating Point Unit as been designed to implement the principles of stochastic arithmetic. This FPU is instantiated in two processing blocks that have to work in parallel in order to verify that the two version of the program running with independent random rounding are staying inside the limitations of stochastic arithmetic. In order to work properly, a key feature of the debugger is to be able to run the two instances simultaneously.

Running a program with GDB

While using GDB, the user usually first sets the target, then sets some breakpoints in its code, and finally tells GDB to run the program. Then the program runs and either reaches a breakpoint or finished its execution. The user can then inspect the different variables. Internally, a TCP connection is opened with the GDB server as the user specifies the target. when the user sets breakpoints, they are only listed locally into a chain until a "run" command is issued. Then, these breakpoints are send to the GDB server that has to set them for the target. After this phase, the client send a command to the server to run the target and waits until an error occurs in the connection or a message telling that the targets is stopped arrives. During this phase, the GDB client is stalled waiting for something to happen in the connection.

Let the processing Blocks run simultaneously

In order to have the two processing blocks running simultaneously, the accuracy analyser can not get stalled after telling one of them to start. To set the two connections, it was possible to duplicate the variable describing its states and call the functions establishing it twice one after the other. The same adaptation have been done to set the breakpoint for the two processing blocks. Two list of breakpoint are update one after the other. However, such a workaround can not be used in this case. As the two targets can be connected independently for each other and have breakpoints set without interacting, the analyser can do it for one after the other. However, while running the program, the two targets are not independent and can not be finished when not running simultaneously.

While a target runs a program, GDB wait for messages from it telling that it is finished or that a breakpoint as been reached. In order to modify GDB at an high level, it was chosen not to do any modifications in the program at the level of the connection but were operations of the target are described. Thus this mode of functioning could not be changed. In order to run the two targets simultaneously, there has to be two processes each one telling one processing block to run and then waiting for it to be finished.

Continue working

To develop an analyser based on a hardware implementation of discrete stochastic arithmetic, the debugger had to be modified to interact with several targets. In order to do so, the variables that describes the state of a target had to be duplicated. The same had to be done with the variables used to set, maintain and use the connection with the target. Each instance could be allocated one after the other, and initialised the same way. Each command of the debugger had then to be modified in order to modify each of these instance in correlation with the modification, for example defining a new breakpoint or removing one.

When the command running the program is used, the process is duplicated and each of the process uses one of the instances of the variables, the one corresponding to the target connected. Thus, the different targets can be started and run simultaneously. To compute the accuracy of the computed results, the different values of the variables have to be collected.

4.3.3 Compute the accuracy

Retrieve the values

As previously explained, after a processing block stops, a symbol table is build by GDB containing information about the different variables of the program. This table contains among others informations about the type of every variable. According to the equation

(2.1) all the computed values of a single floating point variable are needed to compute its accuracy. As these values are stored in the different process running the targets, they have to be collected. The numerical accuracy analyser needs then all of this tables to be able to retrieve the different values.

Compute the accuracy

Once that the different values of each instances of a variable are known, the accuracy of a result can be calculated. First, the type of the variable has to be known. The variables of the type "float" or "double" are the results of computations done in the FPU. Thus, they have to be compared : sign, exponent and mantissa.

As the values are displayed in the debugger as decimal values, the accuracy has to be calculated corresponding to the notation. The formula is :

$$C_{\overline{Z}} = \log_{10}\left(\frac{\sqrt{N}.|\overline{Z}|}{\sigma.\tau_\beta}\right) \; with \; \sigma^2 = \frac{1}{N-1}.\sum_{i=1}^{N-1}(Z_i - \overline{Z})^2$$

Displayed nearby the result, this value tells the user the number of significant digits of this result.

5 Use of the analyser

Useful terms

- client wikipedia : "A client is an application or system that accesses a remote service on another computer system, known as a server, by way of a network". In our case the analyser acts as a client that accesses two server.
- server wikipedia : "a computer program running as a service, to serve the needs or requests of other programs (referred to in this context as "clients") which may or may not be running on the same computer". In this case the GDB server runs an interface allowing the client to interact with a processing block.
- target The target is here the processing block that have to run the program. It is only known from the client as a GDB server interface at a given network address.

5.1 Functionalities

5.1.1 Connecting Processors

In order to develop an automatic numerical stability analyser based on a hardware implementation of discrete stochastic arithmetic, the GNU Project Debugger has been modified to be able to use an hardware platform developed with the principles of stochastic arithmetic. Two processing blocks are implemented on an FPGA and each ot them is able to run the same programme with random rounding. The analyser is used to check the numerical accuracy of any intermediate result without any source-code modification.

Choosing the targets

When the user uses the GDB on his own computer("client") for remote debugging, a TCP connection is set between the debugger and the GDB server. To set this connection, the address of the GDB server is needed. Once this address is given, it is used to connect to the target. The properties of this connection are stored in several variables regrouped in a C struct.

Once the target is connected, the debugger still needs to know some information about the targets, in particular, it requires the information of the architecture of the processing block. This information is sent by the GDB server in form of an XML file.

These mechanisms are triggered by using the "target" command of GDB. An example of the use of this command can be seen at 4.2.1.

Switching between targets

As two targets have to be used, informations about the connection and the target are not unique any more. They each have their own set of variables. It has to be possible to use one set of variables or the other. To do so, two new instances of these variables have been introduced. These two set of variables are backup of the variables used in the initial version of GDB. Thus, after using a target, they are saved and the other set is set to be used.

Here a simplified version of this functioning is showed :

```
struct all_informations_about_a_target *target_description ;//GDB original variable
struct all_informations_about_a_target *target_description_1 ;
struct all_informations_about_a_target *target_description_2 ;

int active_target_number;//new variable used to know which target is currently used

/*
      to save descriptions of a target
*/
void save_target(int number) {
   if (number==1) {
      target_description_1 = target_description ;
   } else if (number==2) {
      target_description_2 = target_description ;
   } else {
      printf("ERROR: unable to save the state of this target\n") ;
   }

   if (debug_target_switch==1) printf("target nr : %d saved\n", number) ;
}

/*
      to set the target description to be used
*/
```

```
void set_target(int number) {
   if (number==1) {
      target_description = target_description_1 ;
      active_target_number = 1;
   } else if (number==2) {
      target_description = target_description_2 ;
      active_target_number = 2;
   } else {
      printf("ERROR: non-existent target\n") ;
      active_target_number = 0;
   }
   if (debug_target_switch==1) printf("target nr : %d set\n", number) ;
}

/*
      changing the target used
*/
extern void switch_target(int number) {
   if (number==1 || number==2) {
      save_target(active_target_number) ;
      if (active_target_number!=number) {
         set_target(number) ;
      } else {
         printf("WARNING : target nr:%d already used\n", number) ;
      }
   } else {
      printf("ERROR : only target 1 and 2 can be used\n") ;
   }
}
```

The two functions are used to store and restore the entire description of a target. Actually, these informations are not in a single C struct, but in several, and some informations are not accessed through pointers but are real variables. All occupied memory pointed to had to be allocated twice before these functions can be used. The variables also had to be properly declared to be visible wherever needed. The original variable used by GDB could not simply be replace by a choice between the two new variable because it as also been used in multiple "define" instructions.

A new command have been introduce in GDB. It is the command "switch". it simply has to be followed by the number of the target to be used. An example of its use follows :

- starting the first server(the last line only appears after connecting to it) :

```
burlet@burlet-laptop:~/test_program$ gdbserver localhost:1234 ./hello_world
Process ./hello_world created; pid = 2316
Listening on port 1234
Remote debugging from host 127.0.0.1
```

- starting the second server(the last line only appears after connecting to it) :

```
burlet@burlet-laptop:~/test_program$ gdbserver localhost:1235 ./hello_world
Process ./hello_world created; pid = 2318
Listening on port 1235
Remote debugging from host 127.0.0.1
```

- starting the modified GDB and connecting to two targets :

```
burlet@burlet-laptop:~$ modified-gdb -nw ../test_program/hello_world
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
Modified by Sylvain Burlet in 2010.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
(gdb) switch 2
target nr : 1 saved
target nr : 2 set
(gdb) target remote localhost:1235
Remote debugging using localhost:1235
(gdb)
```

This command was introduced not only to connect two targets, but also to be able to choose which target is used when using command that have only to use one target during the developing stage. In order to know which target is used. The command "targetsinfo" has been introduced. It prints all the information of the current targets.

```
(gdb) targetsinfo
      target 1 is running the file : /home/bubu/Bureau/prgm/hello_world at localhost:1234
*     target 2 is running the file : /home/bubu/Bureau/prgm/hello_world at localhost:1235
(gdb)
```

5.1.2 Use of the debugger

Once the two targets are connected to the analyser, they are is used to run the two programs simultaneously. It means that they have to be started concurrently and expected to reach the same breakpoint.

Set a breakpoint

Lets take the following program, as an example :

```
1.#include<stdio.h>
2.
3.int main(void) {
4.   while (1==1) {
5.      printf("hello_world !\n") ;
6.      printf("\n") ;
7.   }
8.
9.   return 0 ;
}
```

When the user want the program to stop right after "hello world !", he has to set a breakpoint to the next instruction. Here setting a breakpoint line 6 means that before printing a new line after the "hello world" message, the program would stop. The command in GDB is then used as follow :

```
(gdb) b 6
Breakpoint 1 at 0x80483cd: file ./hello_world.c, line 6.
Now setting the breakpoint for the second target (principal is nr 2)
target nr : 2 saved
target nr : 1 set
Breakpoint 2 at 0x80483cd: file ./hello_world.c, line 6.
Now restoring previous target
target nr : 1 saved
target nr : 2 set
(gdb)
```

As we can see, the breakpoint is stored for both of the target one after the other. Thus when running the two instances of the program, they will stop at the same point.

Run the program

To be able to debug a program, it has to be compiled without optimisations. Debugging informations also have to be introduced to be able to retrieve informations about the symbols (as there are here no variables, it is not really needed) :

```
burlet@burlet-laptop:~/test_program$gcc -g -O0 -o hello_world ./hello_world.c
burlet@burlet-laptop:~$
```

A breakpoint is set line 6 as previously showed. The two target can then be set to run as following :

```
burlet@burlet-laptop:~$ modified-gdb -nw ../test_program/hello_world
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
Modified by Sylvain Burlet in 2010.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
(gdb) switch 2
target nr : 1 saved
target nr : 2 set
(gdb) target remote localhost:1235
Remote debugging using localhost:1235
(gdb) b 6
Breakpoint 1 at 0x80483cd: file ./hello_world.c, line 6.
Now setting the breakpoint for the second target (principal is nr 2)
target nr : 2 saved
target nr : 1 set
Breakpoint 2 at 0x80483cd: file ./hello_world.c, line 6.
Now restoring previous target
target nr : 1 saved
target nr : 2 set
(gdb) continue
target nr : 2 saved
======first process starting======
target nr : 1 set
```

```
Continuing.
======second process starting=====
WARNING : target nr:2 already used
Continuing.

Breakpoint 1, main () at ./hello_world.c:6
6 printf("hello_world !\n") ;
target nr : 2 saved
======second process done   ======

Breakpoint 2, main () at ./hello_world.c:6
6 printf("hello_world !\n") ;
target nr : 1 saved
======first process done    ======
target nr : 2 set
(gdb)
```

The GDB server shows (the two are similar) :

```
burlet@burlet-laptop:~/test_program$ gdbserver localhost:1234 ./hello_world
Process ./hello_world created; pid = 1865
Listening on port 1234
Remote debugging from host 127.0.0.1
hello_world !
```

As we can see, the two targets ran the same program. However the GDB servers used are normal servers. Thus they do not have to be synchronized. It can be proven that they run simultaneously and not one after the other by using input. For example a "scanf" function blocks the two targets before reaching a breakpoint. When the user give an input, only the one processing blocks corresponding can continue. It has been tested thus that the order in which they finish have no influence, and that the analyser waits for both targets to reach the breakpoint.

5.1.3 Accuracy measure

Previously, an example has been showed, where two targets run a simple program simultaneously. This is only half of the aim of the analyser. Indeed, it is meant to compute accuracy of computed values.

Calculated Values

To be able to run the two targets simultaneously a call to the function "fork" is used to duplicate the needed handling in two process, each one using one GDB server. A simplification of the code used while using the command continue can be seen here :

```
void
continue_command (char *proc_count_exp, int from_tty) {
   int save_target = active_target_number ;

   /*
      create shared memory
   */
   memory *shared = create_shared_memory() ;
   //create shared memory for the symbol tables of the second target

   /*
      The two targets have to run each one in a process.
      Target 2 runs with the child, target 1 with the parent
   */
   int pid = fork() ;//creates the two processes
   if (pid==-1) {
      printf("ERROR: the duplication into two processes did not work\n") ;
   } else if (pid==0) {
      printf("======second process starting=====\n") ;
      switch_target(2) ;

      continue_command_single (proc_count_exp, from_tty) ;
      proceed ((CORE_ADDR) -1, TARGET_SIGNAL_DEFAULT, 0) ;

      save_symbol_tables(shared) ;
      //saves the symbol tables read in the second target into the shared memory

      printf("======second process done   ======\n") ;
      exit(0) ;
   } else {
      printf("======first process starting======\n") ;
      switch_target(1) ;

      continue_command_single (proc_count_exp, from_tty) ;
      proceed ((CORE_ADDR) -1, TARGET_SIGNAL_DEFAULT, 0) ;
      switch_target(save_target) ;
      printf("======first process done    ======\n") ;

      wait() ;//We have to wait for the second process to be finished to continue running

      copy_symbol_table_second_target(shared) ;//store the symbol table at its place
       free_shared_memory(shared) ;
   }
}
```

After running the program, part of the data are stored in the client size. For example the values stored in every register is known by the debugger. Currently, the memory used to store these values is allocated after running the program. As the addresses of the values for the second target only exists in the child process, it get loss while closing. To be able to store these values, shared memory should be allocated before continuing the execution of the two instances. The current version still has some bugs in the implementation. However, the concept is verified with the following code which uses the retrieved symbol table to compute the accuracy. The corresponding part of code is then :

```
void
continue_command (char *proc_count_exp, int from_tty) {
   int saved_target_number = active_target_number ;

   printf("======starting current target=====\n") ;

   continue_command_single (proc_count_exp, from_tty) ;
   proceed ((CORE_ADDR) -1, TARGET_SIGNAL_DEFAULT, 0) ;

   printf("======preparing and running second target=====\n") ;

   if (saved_target_number==1) switch_target(2) ;
   if (saved_target_number==2) switch_target(1) ;

   continue_command_single (proc_count_exp, from_tty) ;
   proceed ((CORE_ADDR) -1, TARGET_SIGNAL_DEFAULT, 0) ;

   switch_target(saved_target_number) ;
}
```

In this code the two targets are not running simultaneously but one after the other. It can not work with the designed hardware platform, because the synchronisation part would prevent the first processing block to get finished, and then second one would never be started. However it still is able to show the different values of a variable and generate the numerical accuracy information. This is a proof of the concept of using several instances of the variables corresponding to the targets.

A table of symbols is stored for each instance of the program containing informations about the variables. Thus, their types are known. It has been possible to modify the part of code printing the values to display additional informations about it. An example of such a result can be seen in the following :

- program used :

```
 #include<stdio.h>

int main(void) {
int i;
float radius ;
float pi = 3.141592654 ;
float circumference ;

for (i=0;i<5;i++) {
printf("enter the radius of the circle:") ;
scanf("%f", &radius) ;

circumference = 2 * radius * pi ;

printf("the circumference is :%f\n", circumference) ;
printf("\n") ;//for breakpoint convenience
}
}
```

- first server :

```
burlet@burlet-laptop:~/demo$ gdbserver localhost:1234 ./demo
Process ./demo created; pid = 19834
Listening on port 1234
Remote debugging from host 127.0.0.1
enter the radius of the circle:0.500
the circumference is :3.141593
```

- second server :

```
burlet@burlet-laptop:~/demo$ gdbserver localhost:1235 ./demo
Process ./demo created; pid = 19836
Listening on port 1235
Remote debugging from host 127.0.0.1
enter the radius of the circle:0.499
the circumference is :3.135310
```

- in the analyser :

```
burlet@burlet-laptop:~/demo$ modified-gdb ./demo -nw
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
Modified by Sylvain Burlet in 2010.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".

(gdb) target remote localhost:1234
Remote debugging using localhost:1234
(gdb) switch 2
(gdb) target remote localhost:1235
Remote debugging using localhost:1235
(gdb) b 16
Breakpoint 1 at 0x80484a5: file ./demo.c, line 16.
(gdb) continue
Continuing.

Breakpoint 1, main () at ./demo.c:16
16 printf("\n") ;//for breakpoint convenience
Continuing.

Breakpoint 1, main () at ./demo.c:16
16 printf("\n") ;//for breakpoint convenience
it seems everything worked fine
(gdb) display circumference
2: circumference = 3.14159274 [_second value : 0 accuracy : -0.456344_]
```

```
    (gdb)
```

As it can be seen, it has been possible to integrate additional informations while displaying the value of a variable. The numerical accuracy information can be calculated based on equation (2.1). Currently, only the sequential version of the analyser is able to retrieve the two values. Here the synchronous version have been used and does not read the real value for the second target.

5.2 Possible extensions

5.2.1 Graphic interface

The version of GDB used inside Xilinx tools is already integrated with a graphical interface called "insight". This interface has been written in Tcl/Tk by people working for Red Hat, Inc. and Cygnus Solutions. This code is freely available and can be modified and used for the purposes of the analyser.

Insight

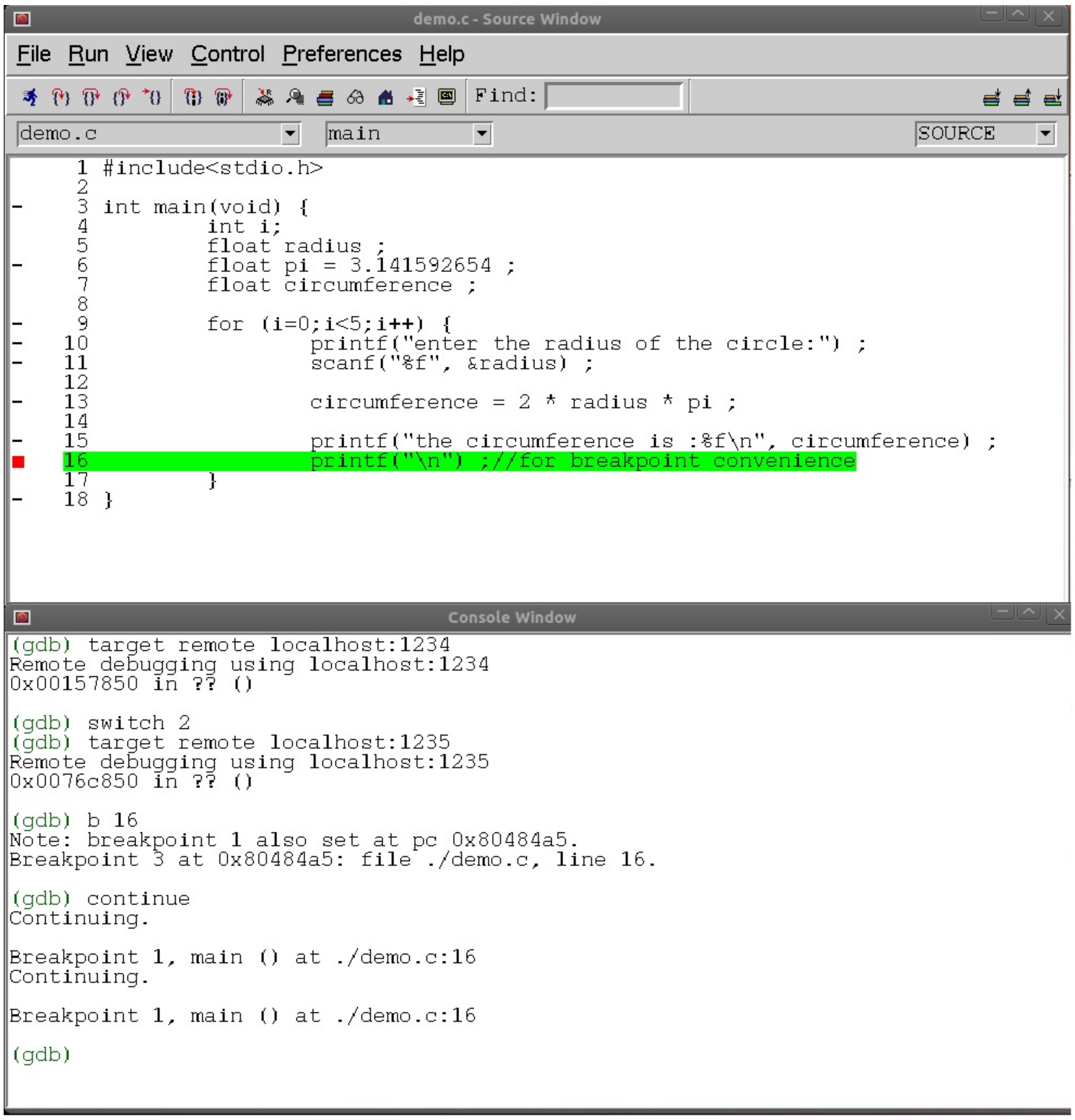
Insight provides the same functionalities as GDB itself, but with graphical interface. The interface can be seen on the figure 5.1

As it can be seen, it is possible using the integrated console to run the modified GDB as needed for its purpose.

Integration with insight

Basically, using the graphical interface in insight provokes the use of command in the underlying GDB, and thus uses the modified ones. However, some adaptations would have to be made to make the use of insight with the modified GDB as user-friendly as currently. The response of GDB to these commands is retrieved and parsed. For example a test have been executed in which the value of the variables are displayed with additional informations. These informations are also displayed inside insight.

The informations about modifying insight can be found at `http://sources.redhat.com/insight/faq.php`



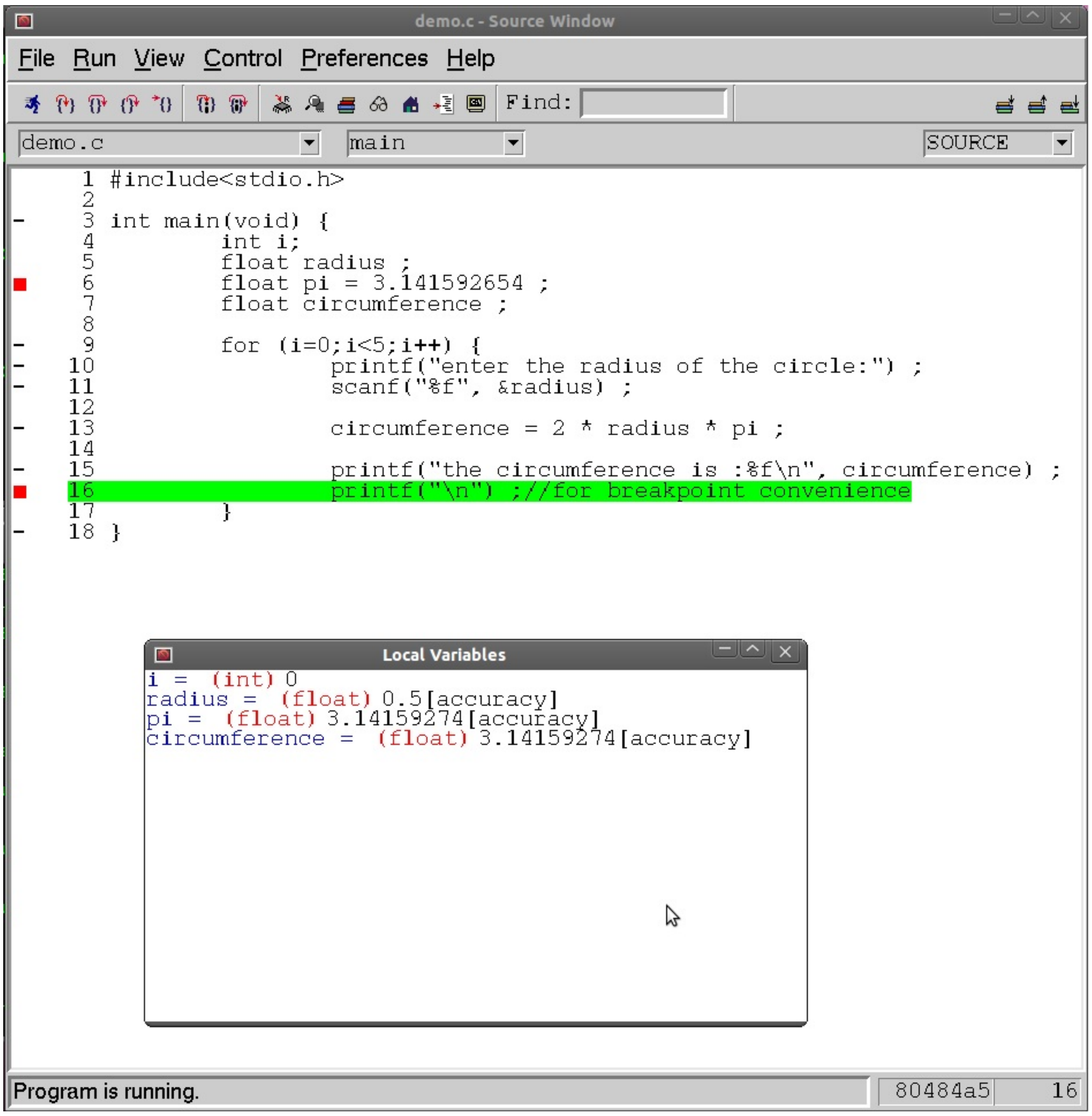
```
#include<stdio.h>

int main(void) {
        int i;
        float radius ;
        float pi = 3.141592654 ;
        float circumference ;

        for (i=0;i<5;i++) {
                printf("enter the radius of the circle:") ;
                scanf("%f", &radius) ;

                circumference = 2 * radius * pi ;

                printf("the circumference is :%f\n", circumference) ;
                printf("\n") ;//for breakpoint convenience
        }
}
```

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x00157850 in ?? ()

(gdb) switch 2
(gdb) target remote localhost:1235
Remote debugging using localhost:1235
0x0076c850 in ?? ()

(gdb) b 16
Note: breakpoint 1 also set at pc 0x80484a5.
Breakpoint 3 at 0x80484a5: file ./demo.c, line 16.

(gdb) continue
Continuing.

Breakpoint 1, main () at ./demo.c:16
Continuing.

Breakpoint 1, main () at ./demo.c:16

(gdb)
```

Figure 5.1: Insight, a graphical interface for GDB, allows to directly use GDB and thus use some modified commands.

5.2.2 More processors

Discrete stochastic arithmetic provides a way to estimate the accuracy of a computed result using the result provided by several execution of a program on several processing blocks. The hardware implementation of these principle that has been considered all along this work only consists of two processing blocks. However, it is possible to use a board with more



```
#include<stdio.h>

int main(void) {
        int i;
        float radius ;
        float pi = 3.141592654 ;
        float circumference ;

        for (i=0;i<5;i++) {
                printf("enter the radius of the circle:") ;
                scanf("%f", &radius) ;

                circumference = 2 * radius * pi ;

                printf("the circumference is :%f\n", circumference) ;
                printf("\n") ;//for breakpoint convenience
        }
}
```

```
i =  (int) 0
radius =  (float) 0.5[accuracy]
pi =  (float) 3.14159274[accuracy]
circumference =  (float) 3.14159274[accuracy]
```

```
Program is running.                    80484a5    16
```

Figure 5.2: Insight directly uses the response from GDB at given commands.

processors. It is straight-forward to adapt the proposed numerical accuracy analyser to work the new system with multiple processors.

Extensible structure

The number of processing blocks in the implementation dictates the number of GDB server that has to run. Indeed each one can only be used for one processing block. It has also to be taken into consideration that the JTAG chain has to support the same number of processing block. Then the analyser has to be able to use all of these processing blocks. When even one is not programmed and started with the others, the entire structure can not run due to the synchronisation part.

It has been previously explained that the debugger normally uses several informations that had to be duplicated to be able to connect to two processing blocks. When more of them would be used, these variables and memory would have to be as many as the number of processing blocks. For example, a list of all the breakpoint would have to be maintained for each of the processing blocks. It would also be the case for the descriptions of the connection to each processing block, and the description of the architecture. Then as many symbol table as the number of processing blocks should be build.

Use loops instead of multiple functions calls

To be able to run the two processing blocks, the function fork() has been used. The child has been meant to write the retrieves informations about symbols in previously allocated shared memory. The parent has to start its corresponding processing blocks and wait for its child to be done. Then it is able to use the symbol tables retrieved to compute the estimated accuracy. With N processing blocks, N-1 child would have to run simultaneously to start each a processing block and write the necessary informations each one in a different area allocated in a shared memory.

Calculate the accuracy

Using two or more processing blocks, calculating the estimation of the accuracy of the computed results uses the same formula given by the stochastic arithmetic :

$$C_{\overline{Z}} = \log_{10}(\frac{\sqrt{N}.|\overline{Z}|}{\sigma.\tau_{\beta}}) \; with \; \sigma^2 = \frac{1}{N-1}.\sum_{i=1}^{N-1}(Z_i - \overline{Z})^2$$

Thus, the analyser would simply have to read the N values of the floating point variables retrieved from the N processing block and use the value of the Student distribution corresponding to N-1 degrees of freedom.

Using more processing block should not change anything to the interface except the settings used to connect to the corresponding additional processing block. Internally, the modifications of the analyser should be minimal. However, the estimation of the numerical accuracy should then be more precise.

5.2.3 Detecting instability

The discrete stochastic arithmetic does not only provide a method to estimate the accuracy of the final results, but it also allows to detect during the execution numerical instabilities.

Accuracy Check at given Point

Checking the accuracy of several intermediate results would be possible as soon as the accuracy can be displayed. Indeed, it means no more than setting a breakpoint at this point and then display the value of the variable of interest. Then the estimation of the accuracy for these variables should be displayed. An example follows :

```
#include <stdio.h>
#include <math.h>

int main(void) {
int i ;

double thefloat ;
printf("please, enter the value to use :") ;
scanf("%lf",&thefloat) ;

for (i=0;i<9999;i++) thefloat += 1.01 ;
for (i=0;i<9999;i++) thefloat -= 1.01 ;

printf("After the addition phase, thefloat is: %2lf\n", thefloat) ;
printf("\n") ;//set a breakpoint here

for (i=0;i<9999;i++) thefloat *= 1.01 ;
for (i=0;i<9999;i++) thefloat /= 1.01 ;

printf("After the multiplication phase, thefloat is : %2lf\n", thefloat) ;
printf("\n") ;//set a breakpoint here

for (i=0;i<10;i++) thefloat *= thefloat ;
for (i=0;i<10;i++) thefloat = sqrt(thefloat) ;

printf("After the three steps, thefloat is: %2lf\n", thefloat) ;
printf("\n") ;//set a breakpoint here
}
```

An example of a run of this program is :

```
burlet@burlet-laptop:~/dummy_programs$ gcc -g -o accuracy_loss ./accuracy_loss.c -lm -O0
burlet@burlet-laptop:~/dummy_programs$ ./accuracy_loss
please, enter the value to use :3.141592654
After the addition phase, thefloat is: 3.141593

After the multiplication phase, thefloat is : 3.141593

After the three steps, thefloat is: inf

burlet@burlet-laptop:~/dummy_programs$
```

When the user wants to know in which part of the program the infinite value appear without adding all the printfs that are used here, he would simply have to had breakpoints between them and then watch the accuracy of the results.

Interrupt Handling

The hardware platform introduced in section 3.1 is able to detect numerical instabilities, and to generate interrupt in this case. This interrupt is visible by the GDB server, which is sending it back to the GDB client connected.

The actual developed modified versions of GDB would simply signal the user that an interrupt was raised and then shut down. However it should be possible to match the program counter with the positions in the code and acting like if a breakpoint had been reached. Thus, it would be possible to watch the different variables at this point to see which one causes the interrupt. Thus it would be much easier to detect were the algorithm should be modified to better use the processor in order to generate more accurate results.

6 Conclusion

In the work, the development of an automatic numerical stability analyser based on a hardware implementation of discrete stochastic arithmetic has been investigated. Discrete stochastic arithmetic provides estimation of the propagation of rounding errors on computed results. In a previous project, a floating point unit has been developed which supports the discrete stochastic arithmetic. This platform is able to provide necessary information to estimate the numerical accuracy information of any computed value. In this work, a numerical accuracy analyser has been developed to gather information from the dedicated hardware system and provide the numerical accuracy information based on the discrete stochastic arithmetic.

In order to do so, the GNU debugger has been modified to be able to connect to the two processing blocks on the dedicated hardware and then execute the same program on both of them. Breakpoints can be set for both instances simultaneously. The different values of variables can be collected and used to compute the accuracy of these values in an asynchronous version of the analyser. However, the asynchronous version does not support self validation of the discrete stochastic arithmetic, and is unable to make an unitive decision for conditional branches. To cover these issues, an synchronous version of the analyser has been investigated. The concept of the synchronous implementation has been verified. However, the current release still has some bugs in retrieving variables from all the processing blocks and additional work is required to complete the necessary functionalities. From the user's point of view, there is no difference from using a state-of-the-art debugger, while it can provide numerical accuracy information without any source code modification for the package under test.

Bibliography

[1] Roselyne Avot-Chotin and habib Mehrez. Hardware implementation of discrete stochastic arithmetic. *Numerical Algorithms*, 2004.

[2] Jean-Marie Chesneaux. Study of the computing accuracy by using probabilistic approach. In C.Ullrich, editor, *Contributions to computer arithmetic and Self-Validating numerical methods*. J.C. Baltzer AG, Scientific Publishing CO.

[3] Jean-Marie Chesneaux. Validité du logiciel numérique.

[4] Jean-Marie Chesneaux. *Etude théorique et implementation en ADA de la méthode CESTAC*. PhD thesis, Université Paris 6, 1988.

[5] Jean-Marie Chesneaux. Les fondements de l'arithmétique stochastique. *C.R. Acad. Sci.Paris*, 315:1435–1440, 1992.

[6] Jean-Marie Chesneaux. Arithmétique des ordinateurs. Polytech'Paris-UPMC, 2003.

[7] Brian Hill. *Software Debugging Techniques for PowerPC 440 Processor Embedded Platforms*. Xilinx, 2008. This application note discusses the use of the Xilinx Microprocessor Debugger (XMD) and the GNU software debugger (GDB) to debug software defects.

[8] Wenbin Li and Sven Simon. An fpu hardware architecture for automatic numerical accuracy analysis. *submitted to IEEE Transactions on computer*, 2011.

[9] Stan Shebs et al.s Richard Stallman, Roland Pesch. *Debugging with GDB*, 2003. The gnu Source-Level Debugger.

[10] John Gilmore Cygnus Solutions Second Edition: Stan Shebs Cygnus Solutions. *GDB Internals*, 2003. A guide to the internals of the GNU debugger.

[11] F. Stummel. Forward analysis of gaussian elimination- part i and ii. *Numerische Mathematik*, 46:365–415, 1985.

[12] James H. Wilknson. *Rounding Erros in algebraic processes*. Englewood Cliffs, N.J., Prentice-Hall, 1963.

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.

(Sylvain Burlet)