UNIVERSITY OF CAMBRIDGE

A Floating-Point Arithmetic Logic Unit for Uncertainty Propagation

University of Cambridge
Department of Engineering
Physical Computation Laboratory

Harry Sarson

Department of Engineering
University of Cambridge

IIB Project Final Report

Pembroke College

May 2019

Acknowledgements

I would like to acknowledge my supervisor, Phillip Stanley-Marbell, who has gone above and beyond to support my this year, and Ryan Voo, whose implementation of floating-point support in the Sunflower Simulator aided me enormously. I am grateful to Alexa Belsham, Sophie Burke and Julia Sarson for proof reading this report and giving feedback.

“The Son is the image of the invisible God, the firstborn over all creation. For in him all things were created: things in heaven and on earth, visible and invisible, whether thrones or powers or rulers or authorities; all things have been created through him and for him. He is before all things, and in him all things hold together” (Colossians 1:15–17)

A Floating-Point Arithmetic Logic Unit for Uncertainty Propagation

The numerical data used by computing algorithms often contains errors and digital representations of real numbers are necessarily inexact. However, the standard floating-point types used today cannot store or propagate any information about the uncertainty of the number; they only store and propagate the best guess of the number as calculated by the processor's arithmetic logic unit (ALU). To better describe the approximate nature of computer arithmetic, this report proposes a new numeric type, the approximate float, that stores uncertainty information.

Chapter 2 describes a framework for uncertainty propagation, built upon the linear uncertainty propagation equations [20]. The uncertain framework defines an approximate float as the best guess of the true value of a real number paired with the variance representing the range of possible true values. Uncertainty-aware algorithms that use approximate floats estimate the variance of their output as well as giving their best guess of the output's true value. This best guess is equal to the output of an equivalent algorithm that uses standard floats. For speed and energy efficiency, the computation needed to propagate uncertainty through arithmetic operations should be implemented in hardware.

The linear uncertainty propagation equations estimate the variance of $z = f(x,y)$ using the variance of x and y and the covariance between them. A representation of uncertainty is only useful if it can be propagated through arithmetic operations. These equations provide rules that govern propagation of variance and thus allow the distribution describing the uncertainty of an approximate float to be represented by its mean (the best guess) and variance.

The linear uncertainty propagation equations use the covariance between the inputs to calculate variance requiring the uncertain framework to store the covariance between every pair of approximate floats. Consequently, the complexity of each uncertainty-aware operation scales with the total number of approximate floats in the framework, include those stored in memory. Section 2.6 finds that the uncertain ALU requires far more hardware resources than a traditional floating-point ALU.

Chapter 3 proposes the uncertain non-standard extension to the RISC-V instruction set architecture (ISA). The uncertain extension adds sixteen instructions of which four are completely new and twelve are modified from existing floating-point instructions. The proposal adds eight uncertain registers — paired with the floating-point registers `f8–15` — and special memory for storing uncertainty information. The uncertain extension supports the uncertain framework by allowing programmers to utilise hardware for propagating uncertainty.

The author evaluated the uncertain framework by adding support for the uncertain extension in the Sunflower Simulator [19]. Chapter 4 describes this implementation and how uncertainty-aware software written in the C language can be compiled and run in the simulator using a modified toolchain. Straightforward access to uncertainty propagation tools in C is a requirement for the uncertain framework to be widely used. Using this method to compile C, the report shows that the modifications required to software are minimal — that access to uncertainty propagation tools in C is indeed straight forward.

Chapter 5 uses the Sunflower Simulator to evaluate the uncertain framework. Using taylor series analysis, Section 5.2 bounds the error introduced by the linear uncertainty propagation equations. The analysis shows that the estimate of the best guess is good if the input variance is small and the estimate of the variance is good provided the function used is fairly linear. Section 5.3 discusses an uncertainty-aware C implementation of the Newton-Raphson method which tests the results of the uncertain framework for an iterative algorithm. The chapter finishes by discussing the possibility an uncertain framework that does not store or propagate covariances. Such a framework is desirable as it would increase performance and reduce the hardware costs. However, Section 5.4 finds that for accurate estimation of the variance, the ALU requires access to covariances.

This report outlines the design of an uncertain framework. The framework allows programmers to produce software that is aware of uncertainty, can make robust decisions and fails gracefully when the accuracy of the numerical data it relies on has become unacceptably bad. Such software will be performant due to hardware support for the operations that propagate uncertainty. This project has implemented the uncertain extension in the Sunflower Simulator and so, today, uncertainty-aware software can be run in simulation. The report presents the uncertain framework for uncertainty propagation. The framework uses the uncertain ALU to propagate variances in hardware using the uncertainty-aware instructions defined by the uncertain non-standard extension to RISC-V.

Table of contents

Chapter 1

Introduction

Many computing algorithms operate on data derived from measurements or models of real word phenomena. Due to uncertainty in measurements and inaccuracies of models, the output of such algorithms are approximations of the true value. Algorithms may amplify errors in the inputs and can cause errors to accumulate over time.

The floating-point representation defined by the IEEE 754-2008 arithmetic standard [11] stores only the computer's best guess of the true value without any information about possible errors. This project proposes a new floating-point type that contains uncertainty information. This type is implemented in hardware to minimise the performance cost of storing and propagating uncertainty.

An arithmetic logic unit (ALU) is a digital circuit in a processor that performs arithmetic operations. A hardware implementation of an uncertainty-aware floating-point type requires an ALU capable of propagating uncertain through arithmetic. An Instruction Set Architecture (ISA) defines the interaction between hardware and software and this report proposes an extension to allow software to use hardware support for uncertainty propagation. The RISC-V ISA is open source and designed to "provide a basis for more specialized instruction-set extension" [17]. The uncertain framework designed in this project is built upon the foundations provided by RISC-V.

This project makes the following contributions:

- The uncertain framework for uncertainty propagation in hardware.
- The uncertain non-standard extension to RISC-V ISA.
- The uncertain ALU.
- An implementation of the uncertain extension in the Sunflower Simulator [19].
- A method to generate instructions from the uncertain extension when compiling c code.

Chapter 2

Designing a Framework for Uncertainty

2.1 Design Criteria

The uncertain framework should satisfy the following criteria which are adapted from the RISC-V XBitmanip extension proposal specification [22]:

1. Architectural Consistency: The uncertain extension must be consistent with RISC-V philosophy. ISA changes should deviate as little as possible from existing RISC-V standards (such as instruction encodings), and should not re-implement features that are already found in the base specification or other extensions.
2. No Cost in Accuracy: The accuracy of the numerical results of this method should not be worse than existing methods. Of particular importance is the accuracy of the best guess; its value should be as close as possible to that obtained using standard floating-point instructions. The accuracy of any estimates of the uncertainty are less important as rough estimates of uncertainty still give useful information.
3. Execution speed: Hardware support will provide a significant reduction in execution time compared to a software implementation of an approximate type. As few as possible additional clock cycles should be required by uncertainty-aware instructions compared to standard floating-point instructions.
4. Hardware Simplicity: Although accuracy and execution speed are the primary aims of the uncertain extension, ideas that dramatically increase the hardware complexity and area, or are difficult to implement, should be penalized and given extra scrutiny.

5. User friendly: The uncertain framework should enable programmers to create robust software that can respond appropriately to uncertain input. Moreover, the work to change existing software to use the uncertain framework should be minimal. The instructions added by the uncertain extension should be compatible with standard instructions and the uncertain application binary interface (ABI) should be compatible with the standard RISC-V ABI.

The wording above is purposely modelled on criteria used in the XBitmanip proposal [22] so that the uncertain framework might inherit the precision of the RISC-V ISA.

2.2 Prior Work

The linear uncertainty propagation equations (defined in Section 2.4) have been analysed in detail [20][1]. Literature has traditionally focused on uncertainty propagation as a tool for analysis of experimental results rather than as a method suitable for computer implementation. More recently, software libraries have begun to use these uncertainty propagation tools to provide solutions for robust mathematical computing. Measurements.jl [6] is a Julia library which propagates uncertainty through mathematical calculations using a new Measurement data type that represents an approximate value. Measurements.jl uses the linear uncertainty propagation equations with a tagging method to track correlation between Measurements. An independent Measurement (a Measurement uncorrelated with all other Measurements) is given a unique tag. A derived Measurement (for example the sum of two independent Measurements) stores the tag of each independent unit from which it has been derived and the derived Measurement's partial derivative with respect to each independent Measurement. The library uses these tags and partial derivatives to correctly calculate variances using the linear uncertainty propagation equations. However, storing and using these tags reduces the performance of the library [5].

The Uncertain<T> library [2] and The NIST Uncertainty Machine [15], use a different approach for propagating uncertainty based on a tree representation of arithmetic expressions and Monte Carlo statistical methods. The Uncertain<T> library defines uncertain values that store an independent or a derived approximate value. An independent uncertain value contains a sampling function and its statistics (for example its mean or its variance) are estimated from samples generated by the sampling function. A confidence interval is constructed for the statistic and samples are generated until the desired value of the confidence interval (for example 95%) is achieved. A derived value stores a reference to the uncertain values it depends on and a function that maps those values to it. For example an uncertain value constructed from the sum of two uncertain values contains a reference to those two values as well as the function $f(x,y) = x + y$. Statistics of derived uncertain values are estimated from samples in the same way as independent uncertain values. To generate a sample from a derived uncertain value, the library first generates samples from each independent value the derived value depends on and then combines those samples using the stored function. These approaches are powerful because, by using Monte Carlo methods, they can model arbitrary probability distributions. The performance of these methods depend on fast procedures for generating samples for independent uncertain values.

Both Measurements.jl and Uncertain<T> abstract the calculations required for uncertainty propagation from the programmer. This abstraction simplifies development of uncertainty-aware software and enables a programmer who is not an expert with uncertainty or probability to use these methods. Whilst these libraries approaches are well suited to software, neither can be directly used to implement a framework with hardware support as they both rely on dynamic allocation of resources. Moreover, both approaches can use an unbounded amount memory when representing an approximate value. Uncertain<T> dynamically allocates memory to store the tree representation of an uncertain value. As arithmetic operations are applied to create new uncertain values, larger trees (requiring more memory) are created. Measurements.jl uses dynamic memory to store information about correlation between Measurements; the amount of memory depends on the number of other values a Measurement is correlated with. In software, the operating system will dynamically allocate memory to libraries on demand but in hardware space is limited and must be specified during design. It is therefore infeasible to use these methods directly in a hardware implementation of uncertainty propagation.

2.3 Propagation of Uncertainty in Hardware

Approximate values are represented by a random variable with a known probability distribution. The laws of probability allow uncertainty information to be propagated through arithmetic operations. However, exactly representing an arbitrary continuous distribution is impossible so this framework must use an approximation of the distribution instead. The uncertain ALU must be able to construct this approximation of the distribution for the result of an arithmetic operation. Two approximations are considered in detail: Monte Carlo samples and moments of a distribution.

Monte Carlo Methods

The Uncertain<T> approach [2] uses a variable number of samples and is thus unsuitable for hardware. However, a probability distribution could instead be modelled by a fixed number of samples. These samples build an estimation of the approximate value's underlying distribution. To propagate uncertainty, the ALU applies arithmetic operations piecewise to every sample yielding an estimation of the distribution of the resulting approximate value. The mean, variance and other statistical properties can be estimated using Monte Carlo methods. In particular the best guess of an approximate value is given by the empirical average of sample values. An advantage of this approach is that the memory and computational complexity are fixed by the hard-coded number of samples. However, this approach does not satisfy the second design criteria: to construct the samples that this approach requires, random numbers must be added to the measured or estimated value. The best guess of approximate value (which is the mean of these generated samples) will vary around the measured or estimated value due to the random numbers. Accurate best guesses of approximate floats requires use of a large number of samples which is infeasible due to the fourth design criteria.

Finally, it is possible that specialised hardware support for Monte Carlo based approximate values may be made redundant by the RISC-V vector extension [18] currently under design. The main operation requiring support from hardware is the parallel update of samples. These parallel operations will be supported efficiently by the vector extension, support that — according to the first design criteria — should not be reimplemented as part of this framework. It is not clear if other hardware supported operations (for example a shuffle of samples) are needed and, if so, whether the vector extension will support them. As the vector extension will, at least partially, provide the hardware support needed for Monte Carlo methods, the scope for a specialised uncertain ALU is reduced.

Moment Methods

Other designs use moments to approximate a distribution. The first moment, the mean, of a distribution is the best guess of an approximate value and traditional computer arithmetic operations effectively consider only this first moment. The variance is the second central moment of a distribution, the skew is the third and the the kurtosis is the fourth. The description of the distribution becomes more accurate as more moments are used. Using the best guess and the variance as approximation of a distribution allows a compact binary representation whilst encoding information about the distribution. The linear uncertainty propagation equations described in Section 2.4

approximate the variance of the result of an arithmetic function based on the first-order partial derivatives of the function and the variances of the inputs. A uncertain ALU could use the linear uncertainty propagation equations to estimate the best guess and variance of the result of an arithmetic operation. The estimate of the best guess is simply the traditional floating-point result of the operation and thus this design satisfies the second criteria. Difficulties with this approach arise due to the covariance term in the linear uncertainty propagation equations.

Storing the covariance between all possible combinations of approximate values is required for accurate uncertainty propagation but uses a large amount of hardware resources and complicates implementation. To avoid requiring a computer to store these covariances, the uncertain ALU could instead work with an upper bound on the variance (for example the bound given by Equation 2.6). It may also be viable to assume the covariance to always be zero in the hope that errors are small or cancel out.

Keeping track of the correlation between values increases accuracy at the cost of significant computational complexity. If the total number of approximate values in the system is N then $N(N-1)/2$ covariances must be stored to correctly account for correlation between values. Using these correlations, the variance of the result of a binary function can be accurately computed, notwithstanding the approximations of the linear uncertainty propagation equation. The significance of the error introduced by bounding or ignoring covariances will be analysed in Section 5.4. The following sections lay out a framework and describes a possible hardware implementation for a moment method using covariance information.

2.4 Linear Uncertainty Propagation

The uncertain framework uses the linear uncertainty propagation equations [20] to calculate variance. For an approximate value defined $z = f(x,y)$ where $x = \bar{x} \pm \sigma_x$ and $y = \bar{y} \pm \sigma_y$, the linear uncertainty propagation equations estimate the best guess and the variance of z.

$$\bar{z} = f(\bar{x}, \bar{y}) \tag{2.1}$$

$$\sigma_z^2 = \left(\frac{\partial f}{\partial x}\sigma_x\right)^2 + 2\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\sigma_{xy} + \left(\frac{\partial f}{\partial y}\sigma_y\right)^2 \tag{2.2}$$

where all derivatives of f are evaluated at $x = \bar{x}$ and $y = \bar{y}$. The covariance between z and x or y is

$$\sigma_{zx} = \frac{\partial f}{\partial x}\sigma_x^2 + \frac{\partial f}{\partial y}\sigma_{xy} \tag{2.3}$$

$$\sigma_{zy} = \frac{\partial f}{\partial x}\sigma_{xy} + \frac{\partial f}{\partial y}\sigma_y^2. \tag{2.4}$$

The covariance of z with another approximate value $u = \bar{u} \pm \sigma_u$ is

$$\sigma_{zu} = \frac{\partial f}{\partial x}\sigma_{xu} + \frac{\partial f}{\partial y}\sigma_{yu}. \tag{2.5}$$

Because $|\sigma_{xy}| \leq \sigma_x\sigma_y$ [13], the standard deviation of z must obey

$$\sigma_z \leq \left|\frac{\partial f}{\partial x}\right|\sigma_x + \left|\frac{\partial f}{\partial y}\right|\sigma_y. \tag{2.6}$$

This upper bound for the variance of the output of a binary function can be calculated when the covariance between the inputs is unknown.

The approximate float consists of a best guess and a variance estimate stored using a floating-point representation. Equation 2.2 implies that calculation of the variance of the result of an arithmetic operation requires the covariance between all possible inputs to be stored.

An approximate float can have unknown uncertainty, in which case its variance and covariances are NaN. The standard rules for propagation of NaN values through floating-point operations (as described by the section 6.2.3 of the IEEE 754-2008 arithmetic standard [11]) ensure arithmetic applied to values with unknown uncertainty produces results with unknown uncertainty.

2.5 A RISC-V Extension for Approximate Floats

In this project, the uncertain framework will be implemented as a non standard extension to RISC-V, an open source ISA developed at UC Berkeley [17]. To do so, four things are required: a way to store the variance and covariances in hardware; a new ALU capable of applying Equations 2.1–2.5 to the stored uncertainty information; instructions that allow programmers to use the new ALU; and an ABI which allows approximate floats to be passed to and returned from procedures.

Special Memory

In the same way that traditional floats are stored in memory and can be loaded into registers when they are needed in computations, the uncertain extension will allow approximate floats to be stored in memory.

Registers will contain $N_R = 8$ approximate floats and up to N_M approximates floats can be stored in memory. The uncertain memory size N_M specifies the maximum number of unique uncertain memory locations available in memory — it is not equal to, nor does it scale linearly with, the number of bytes needed to physically provide this memory. General memory is not suitable for storing the uncertainty information associated with approximate floats as the correlations between values requires storing all the floats contiguously in memory. Additionally, the nature of Equations 2.1 and 2.2 necessitates a much tighter coupling between the processor and the uncertain memory than between the processor and general memory.

Consider the case where x, y and z represent approximate floats contained in registers and u represents an approximate float stored in memory. The uncertainty information about u consists of the variance of u and any covariance involving u. The uncertain ALU does not use uncertainty information about u to calculate either the best guess or the variance of z Neither is uncertainty information about u used to calculate the covariance of z with x (Equation 2.3), z with y (Equation 2.4), or z with any other approximate float contained in registers. However, the covariance between z and u does depend on uncertainty information about u as shown by Equation 2.5. Should u be loaded into a register, the ALU would require the covariance between z and u. Thus, the uncertain ALU must calculate σ_{zu} despite u being stored in memory. The ALU requires access to all covariances between approximate floats in registers and approximate floats in memory.

Registers are required for N_R variances, one for each approximate float. There are $N_R(N_R - 1)/2$ register-register covariances between pairs approximate floats both contained in registers. There are $N_R N_M$ register-memory covariances between approximate floats contained in registers and approximate floats stored in memory. All register-register and register-memory covariances ($N_R(2N_M + N_R - 1)/2$ in total) must be available to the uncertain ALU; they must be contained in registers. In memory, there is space to store the variance of up to N_M approximate floats. There are $N_M(N_M - 1)/2$ memory-memory covariances between pairs approximate floats in memory. As the uncertain ALU will not change their value, memory-memory covariances are stored in memory. The number of covariances stored in hardware increases linearly with the size of the uncertain memory. Therefore, the amount of uncertain memory available is necessarily small.

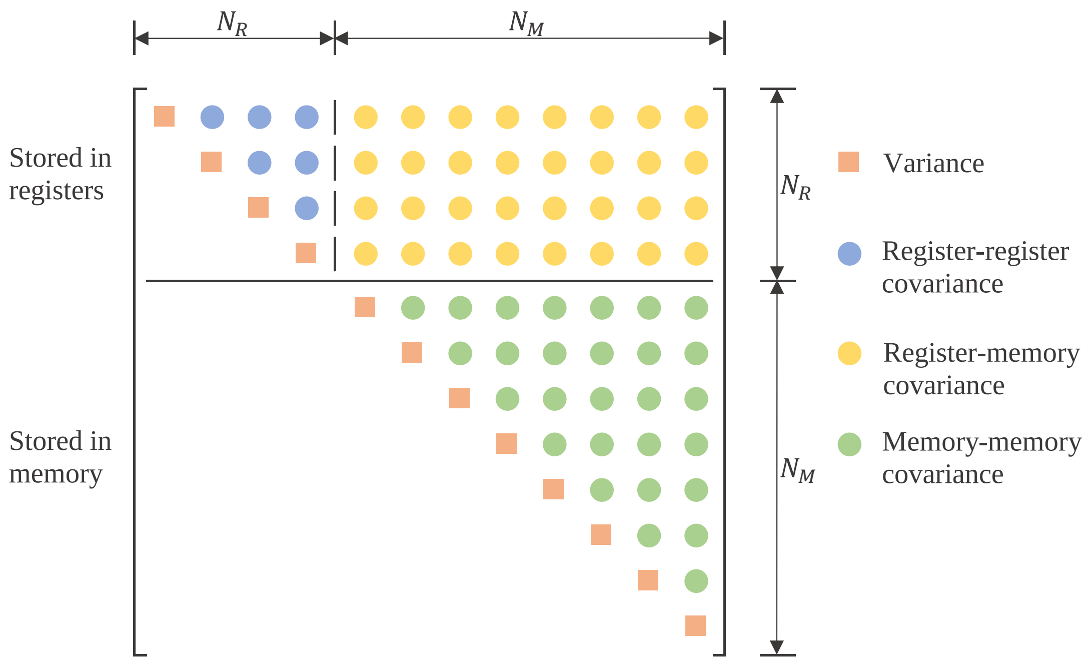


Fig. 2.1 Uncertainty information is stored in registers and in memory. In this example, $N_R = 4$ and $N_M = 8$.

Figure 2.1 shows all the uncertainty information in the system partitioned into registers and memory. Memory-memory covariances make up part of the uncertainty information for two approximate floats. The requirement for contiguous memory stems from sharing this uncertainty information.

Uncertain Arithmetic Logic Unit

A processor requires an uncertain ALU to support this ISA extension. The ALU uses Equation 3.2 to update one variance, $N_R - 1$ register-register covariances and N_M register-memory covariances. In hardware, these updates can be performed in parallel providing a concrete benefit of a hardware implementation over an approximate float software library.

New Uncertainty-Aware Instructions

Software requires instructions to set, retrieve and propagate uncertainty information. The author considered using existing floating-point instructions to manage uncertainty. Uncertainty-aware load and store instructions require extra information compared to the standard load and store instructions as they must manage both the best guess and the variance. Therefore, existing floating-point load

Table 2.1 Uncertain instructions allowing access to uncertain information.

UNGCOV.S	Get covariance between two approximate floats.
UNSVAR.S	Set variance of an approximate floats.
UNCLVAR.S	Clear the variance of an approximate float.
UNUPG.S	Update the uncertainty information of an approximate float using a gradient.

and store instructions are not suitable and, as described in Section 3.5, the uncertain extension adds new instructions for uncertainty-aware load and store operations.

For floating-point computational instructions, propagating uncertainty may require additional energy or increase the execution time of the instruction. If a program chooses not to use the uncertain extension, it should not pay this cost. Additionally, software may wish to specify a different rounding mode for uncertainty calculations, for example rounding towards infinity for a conservative variance estimate. Separate uncertainty-aware computational instructions would permit a different round mode for the best guess and the variance. Finally, changing the behaviour of existing instructions breaks the stability guarantees given by the RISC-V foundation when they froze the ISA. Therefore, the uncertain extension adds new computational instructions, defined in Section 3.6, that behave in a similar way to standard computational instructions but also propagate uncertainty. Software can use four new instructions listed in Table 2.1 to set and retrieve uncertainty information. These instructions are defined fully in Section 3.7.

Uncertain ABI

The uncertain extension needs an ABI to allow linking of external libraries for uncertainty-aware calculation. The uncertain ABI is based on the RISC-V ELF psABI [16] and is defined in 3.8. To satisfy the fifth design criteria this ABI must be compatible with the standard ABI so that programmers can use standard libraries without needing to recompile.

Interactions with Standard Floating-Point Instructions

Support for the non-standard uncertain RISC-V extension builds on the base single-precision instruction subset F. Therefore, one must consider the behaviour of intermixed standard single-precision floating-point instructions and uncertainty-aware floating-point values.

Defining this relationship would not be necessary if the behaviour of standard floating-point instructions were adapted to propagate uncertainty. However, the downsides of this approach described above outweigh this benefit.

A defensive option is to specify that any non floating-point operations produce output with unknown uncertainty (see Section 2.4). This approach, accidental usage of standard floating-point instructions in place of uncertainty-aware instructions would lead to software reading a NaN variance. It would be straightforward to detect something was awry and hopefully possible to pinpoint the source of the error by tracing back to the first occurrence of a NaN variance. One facet of the ABI compatibility is that procedures that use uncertainty-aware instructions can call procedures that do not. Standard RISC-V procedures must preserve the values in the registers `fs0`–`fs11` across procedure calls. However, procedures as procedures will preserve the values by storing to and loading from the stack using FSW and FLW, they will not preserve any uncertainty information associated with the registers `fs0`–`fs1`. (The registers `fs2`–`fs11` are not paired and so can not have any associated uncertainty information.) Therefore, to allow ABI compatibility the uncertain extension would have to declare that `fs0` and `fs1` are no longer callee saved. This is an undesirable and confusing behaviour and so alternative interactions between uncertainty-ware and standard floating-point instructions were sought.

A simple option is to specify that standard floating-point instructions leave uncertainty information untouched. As a concrete example using instructions from Table 2.1, the sequence of instructions

```
UNSVAR.s  fa2, fa0
FMV.s     fa2, fa1
UNGCOV.s  fa3, fa2
```

would copy the (possibly rounded) value of `fa0` to `fa3` and copy the value of `fa1` to `fa2`. This approach may lead to hard to detect errors where uncertainty information is not updated due to standard instructions being erroneously used in place of uncertainty-aware instructions. The author hopes that static checking tools may be able to help detect these errors and this would not be a problem for compiler generated code which makes up the vast majority of executables. No matter what instructions a RISC-V procedure executes, provided there are no uncertainty-aware instructions, the uncertainty information for all registers will remain unchanged. If a programmer wishes to use uncertain instructions then they can simply preserve the uncertainty information using the UNFSW and UNFLW instructions. Callee saved registers and ABI compatibility can both be achieved at the cost of potentially hard to detect errors. The author believes this is a worthwhile compromise.

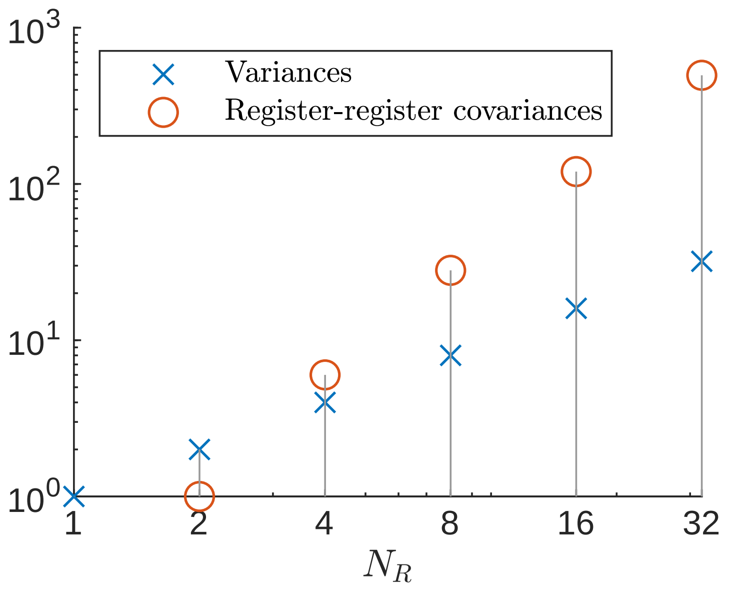


Fig. 2.2 The number of variances and register-register covariances required for a given number of uncertain registers N_R. This figure uses a logarithmic scale.

2.6 Required Hardware Resources

The author believes that the scaling of hardware resources required to implement the uncertain extension will be close to the scaling in the number of values contained in the processor. Therefore, as proxy for the required hardware resources, this section considers the number of values that must be contained within the processor. In future work, more accurate estimates of the required hardware resources may be calculated using a synthesisable RTL design of the uncertain extension.

The uncertain extension requires three categories of value to be contained in the processor: variances, register-register covariances and register-memory covariances. The number of values from each category contained within the processor are defined in 2.5. Figure 2.2 shows how the number of variance and register-register covariances depends on the number of uncertain registers. Figure 2.3 shows how the number of register-memory covariances depends on the number of uncertain registers and memory locations vary. The quadratic scaling of the number of register-register covariances with uncertain registers favours a small number of uncertain registers. The ratio of the maximum total number of approximate floats in the uncertain framework to the number of values contained in registers $(2(N_R + N_M)/(N_R(2N_M + N_R + 1)$ is maximised by storing all approximate floats in memory. For a usable uncertain extension, some uncertain registers are required but efficient hardware usage favours a minimum number of uncertain registers. The "C" Standard Extension for Compressed Instructions [17] provides precedence for using 8 registers `f8`–`fa15`.

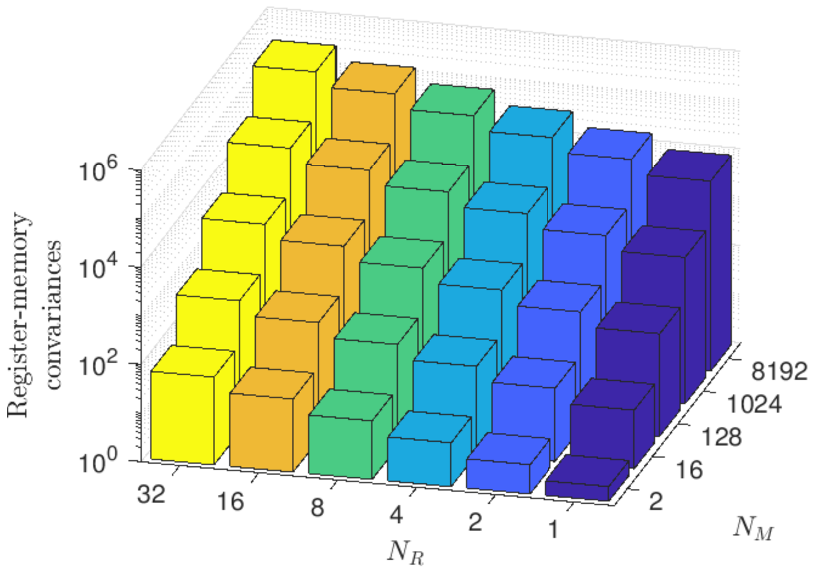


Fig. 2.3 The number register-memory covariances required for a given number of uncertain registers N_R and memory locations N_M. This figure uses a logarithmic scale.

Having chosen to use 8 uncertain registers, the author picked an uncertain memory size of 1024. 1024 uncertain memory locations requires 8192 register-register covariances contained within the processor.

The uncertain ALU also requires additional hardware resources compared to a standard floating-point ALU. To calculate a variance using Equation 2.2, the ALU must compute nine multiplications (three for each term in the equation) and two additions. To calculate each covariance (for example the covariance between z and x using Equation 2.3), the uncertain ALU must compute four multiplications (two for each term in the equation) and one addition. As $N_R - 1$ register-register covariances and N_M register-memory covariances must be computed for every floating point operation, a large number of hardware resources are required to fully parallelise these covariances updates. As described by the second design criteria, highly accurate estimates of variances and covariances are not of first importance. Therefore, there is potential to use lower precision binary representations of covariance that would reduce the hardware resources required for propagation. Additionally, imprecise hardware has been shown to lower the hardware resources required for a floating-point ALU by allowing some loss of accuracy [3]. Such techniques may be useful to further reduce the hardware resources required by an uncertain ALU.

Chapter 3

"Uncertain" Non Standard Extension for Approximate Floating-Point

This chapter proposes the uncertain non-standard extension to version 2.2 of the RISC-V ISA manual [17]. The style of writing in this chapter is intended to match RISC-V ISA manual and inspiration has been taken from the proposed bit manipulation [22] and vector [18] extension specifications. This chapter forms a self contained document although it may be best understood in conjunction with the base RISC-V ISA manual. The uncertain non-standard extension depends on the base single-precision instruction subset F.

As in the RISC-V ISA manual, commentary on design decisions is formatted as in this paragraph, and can be skipped if the reader is only interested in the specification itself.

3.1 Uncertain Register State

The uncertain extension adds 8 uncertain registers `u8`–`u15`. These registers are paired with the floating-point registers `f8`–`f15`. We describe the floating-point registers `f8`–`f15` as paired registers which, together with uncertain registers `u8`–`u15` constitute the uncertain register file. We describe the remaining 24 floating-point registers as unpaired. A paired floating-point register contains the best guess, or nominal value, of an approximate float whilst the corresponding uncertain register contains that float's variance. Figure 3.1 shows this additional state.

The uncertain extension uses the linear uncertainty propagation equations to calculate uncertainty information [20]. Consider an uncertainty-aware instruction that writes the result of the function

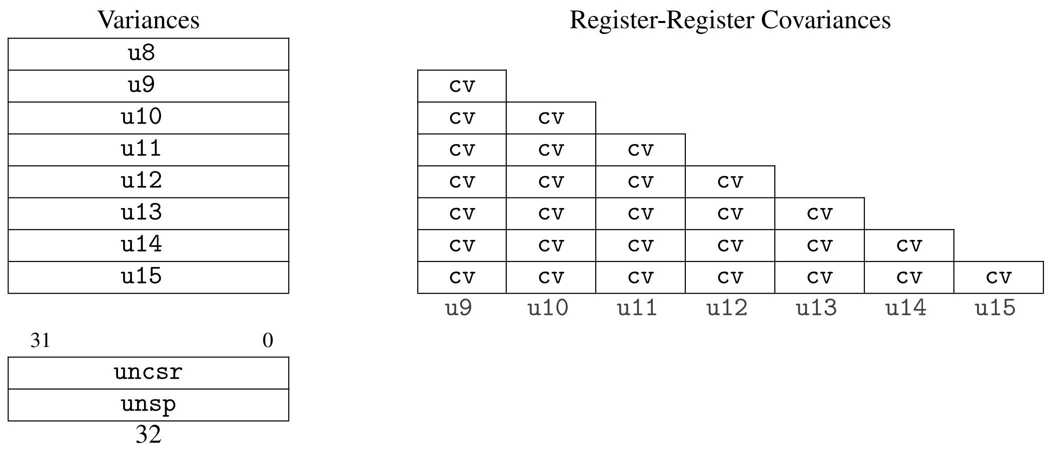


Fig. 3.1 Uncertain extension additional register state. There is one covariance between each unique pair of uncertain registers.

$f(x,y)$ to *rd* where $x = \mathit{rs1}$ and $y = \mathit{rs2}$. *rs1*, *rs2* and *rd* are all paired floating-point registers. The best guess $\overline{\mathit{rd}}$ and variance $\sigma[\mathit{rd},\, \mathit{rd}]$ of the approximate float contained by *rd* will have the new values

$$\overline{\mathit{rd}} \leftarrow f\left(\overline{\mathit{rs1}}, \overline{\mathit{rs2}}\right) \tag{3.1}$$

$$\sigma[\mathit{rd},\, \mathit{rd}] \leftarrow \left(\frac{\partial f}{\partial x}\right)^2 \sigma[\mathit{rs1},\, \mathit{rs1}] + 2\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\sigma[\mathit{rs1},\, \mathit{rs2}] + \left(\frac{\partial f}{\partial y}\right)^2 \sigma[\mathit{rs2},\, \mathit{rs2}] \tag{3.2}$$

where $\sigma[\mathit{rs1},\, \mathit{rs2}]$ is the covariance between *rs1* and *rs2* and the partial derivatives of f are evaluated at $x = \overline{\mathit{rs1}}$ and $y = \overline{\mathit{rs2}}$.

To correctly propagate uncertainty, Equation 3.2 requires the covariance between *rs1* and *rs2*. Therefore, the uncertain extension adds 28 additional registers to contain the covariance between each of the 8 approximate floats contained within paired registers. We call the covariance between two approximate floats contained within the uncertain register file the register-register covariance. This is the first of three classes of covariances used by the uncertain extension which are outlined in Table 3.1. When updating the best guess and variance of *rd*, the register-register covariance $\sigma[\mathit{rd}, \mathit{fx}]$ must also be updated. We have

$$\sigma[\mathit{rd}, \mathit{fx}] \leftarrow \frac{\partial f}{\partial x}\sigma[\mathit{rs1}, \mathit{fx}] + \frac{\partial f}{\partial y}\sigma[\mathit{rs2}, \mathit{fx}] \tag{3.3}$$

for all paired registers $\mathit{fx} \neq \mathit{rd}$. (The partial derivatives of f are evaluated at $x = \overline{\mathit{rs1}}$ and $y = \overline{\mathit{rs2}}$.)

The number of register-register covariances grows quadratically with the number of approximate

Table 3.1 Covariances that make up the uncertain extension can be classified as one of three types.

Covariance type	Written by computational instructions?	Usage
Register-register	Yes	Used to calculate the variance of arithmetic operations.
Register-memory	Yes	Will become register-registers covariances if the memory location they refer to is loaded by UNFLW.
Memory-memory	No	Will be copied into register-covariances if one of the memory locations they refer to is loaded by UNFLW.

values stored in registers. To limit the cost of implementation we limit the number of uncertain registers to 8 which we believe is sufficient for most use cases.

A floating-point register that is not paired with an uncertain register stores no information about the uncertainty and therefore operations that retrieve uncertainty will return NaN if applied to these unpaired registers.

3.2 Special memory

The uncertain non-standard extension adds uncertain memory used to store uncertainty information. This memory can store uncertainty information about up to UNMSIZE=1024 approximate floats.

When loading an approximate float from uncertain memory with UNFLW, the best guess, the variance, and the covariance between the loaded value and each of the 7 other approximate floats currently contained within registers must be copied from memory. In the uncertain RISC-V architecture, we call the covariance between any approximate float in a register and any approximate float stored in memory a register-memory covariance. A register-memory covariance is the second of the classes of covariance listed in Table 3.1. When the ALU updates a register it must also update the register-memory covariances. As a register-memory covariance can be updated by uncertain arithmetic logic unit (ALU) it must be stored within the processor.

All register-memory covariances must be contained in registers. There are $8 \times$ *UNMSIZE register-memory covariances and they are not exposed to software.*

The final class of covariance is memory-memory covariance – covariances between two approximate floats that are both stored in uncertain memory. The uncertain ALU will not change the value of these covariances and so they can be stored in memory.

As register-memory covariances are constrained to be stored in registers, the maximum size of the uncertain memory, UNMSIZE, must be small and specified by this ISA. We tentatively propose a value of 1024 requiring 8192 register-memory covariances.

Updating every register-memory covariance involving rd as part of each uncertainty-aware floating-point operation is a potential bottleneck for the implementation of an uncertain ALU. However, these register-memory covariances only need to be read in two cases:

- *When loading uncertainty information from uncertain memory.*
- *When updating other register-memory covariances.*

There is potential, therefore, for micro-architectural optimisations to exploit the restricted set of situations where a register-memory covariances is read.

3.3 Opcode Encoding

This extension proposes a mixture of 32 bit and 64 bit instructions. The 32 bit instructions use *brownfield* encodings and fit around the standard instructions with the OP-FP major opcode. The 64-bit instructions use *greenfield* encodings and consist of 32 bits prepended to a standard instruction RISC-V floating-point instruction.

Green and brownfield encodings are defined in Chapter 21 of the RISC-V ISA Manual [17].

The hope is that this format will be simple to decode; processors designed to process 32 bit instructions can treat a 64 bit uncertainty-aware instruction as two 32 bit instructions and introduce some extra micro-architectural state. The processor will update the uncertainty information when evaluating a floating-point instruction if and only if the previous instruction evaluated by the processor was the first part of an uncertainty-aware instruction. Figure 3.2 shows the proposed encodings for the instructions defined in this document.

The implementation of this extension in the Sunflower Simulator [19] used this approach to allow a system designed for 32 bit instructions to decode and evaluate the 64 bit uncertainty-aware instructions. We chose to not attempt the optimise these opcodes for code size, instead aiming for simplicity. Later iterations of this proposal may choose to redesign the opcode encoding to achieve higher code density.

3.4 Uncertain Control and Status Registers

The Uncertain extension adds two control and status registers (CSRs). The first, `uncsr`, is a 32-bit read/write register that holds the accrued exception flags. The second, `unsp`, is a XLEN bit read/write register that stores the current uncertain stack pointer described in Section 3.8.

Time constraints did not permit either an implementation in the Sunflower Simulator [19] or a full evaluation of these CSRs. Consequently this section is little more than a stub to be expanded upon in future work.

3.5 Uncertain Load and Store Instructions

UNFLW and UNFSW load and store approximates float to and from memory. These instructions use a similar addressing mode as the standard ISA, the best guess loaded from or stored to the address in register *rs1'* with a 12-bit signed offset. The uncertainty is loaded from and stored to uncertain memory at the index in register *rs1* with another 12-bit signed offset. Each address in uncertain memory contains uncertainty information about a different approximate float.

UNFLW and UNFSW are both 64 bits in length. The lower 32 bits (those fetched first from little endian memory) of each instruction have an I-type encoding and contain the register specifier *rs1* and *offset* which together address uncertain memory. The upper 32 bits of UNFLW are exactly those of FLW with the register *rs1'* specifier and *offset'* encoded as in the I format if one starts counting from the 32 bit. The *rs1'* specifier and *offset'* together specify the address of a single precision float to load as the best guess of *rd*. The upper 32 bits of UNFLW are exactly those of FSW with the register *rs1'* specifier and *offset'* encoded as in the S format if one starts counting from the 32 bit. The *rs1'* specifier and *offset'* together specify an address to which the best guess of *rs2* should be stored as single precision float. The binary format of UNFLW and UNFSW is shown in Figure 3.2.

These new instructions have the following assembly mnemonics:

```
unflw rd,  offset'(rs1'), offset(rs1)
unfsw rs2, offset'(rs1'), offset(rs1)
```

We designed the uncertain instruction formats to be as similar as possible to the standard *instructions. In particular, all register specifiers are either in the same place as standard instructions or are shifted 32 bits left.*

3.6 Uncertainty-Aware Computational Instructions

There are 10 uncertainty-aware floating-point computational instructions: UNFADD.S, UNFSUB.S, UNFMUL.S, UNFDIV.S, UNFSGNJ.S, UNFSQRT.S, UNFSGNJN.S, UNFSGNJX.S, UNFMIN.S and UNFMAX.S. These instructions are 64 bits long and, with the exception of the funct3 field which is used to set the rounding mode, have identical lower 32 bits. The upper 32 bits of each instruction are exactly the bits of the corresponding uncertainty unaware floating-point computational instruction.

Adding uncertainty-aware fused multiply add instructions is out of scope for this work but a desirable addition to the ISA extension.

The rounding mode defined in bits 12 to 14 controls the rounding used when propagating uncertainty. This rounding mode must be statically set. Bits 44 to 46 define the rounding mode used to compute the best guess of the result and can be set to use the dynamic rounding mode.

We refer the reader to section 8.2 of the RISC-V ISA Manual [17] for a discussion on rounding modes for floating-point operations.

If the destination register is not a paired register, an uncertainty-aware computational instruction will behave identically to the corresponding standard floating-point instruction. If one of the source registers is not a paired register, uncertainty-aware computational instructions will write NaN to the variance of the destination register (if it is a paired register).

3.7 Uncertain Instructions

The uncertain extension adds four new 32 bit long instructions. UNGCOV.S calculates the covariance between *rs1* and *rs2* and writes the result to *rd*. If the uncertainty of either *rs1* or *rs2* is unknown then NaN will be written to *rd*. Note, UNGCOV.s *rd*, *rs1*, *rs1* writes the variance of *rs1* to *rd*. UNSVAR.S sets the variance of *rd* to the value contained by the floating-point register *rs1* and the covariance between *rd* and every other approximate float is set to zero. This ISA extension only allows software to initialise independent approximate floats. UNCLVAR.S sets the variance of *rd* to zero (indicating the value contained in *rd* is exact). Both UNSVAR.S and UNCLVAR.S are no-ops if *rd* is not a paired register.

The covariance matrix of a vector of random variables must be positive semi-definite. We do not allow software to set covariances directly as doing so could violate this constraint leading to mathematically inconsistent results. No elegant method for specifying arbitrary covariances whilst ensuring validity of the matrix has yet been devised and so such functionality has been left out of this ISA extension.

UNUPG.S updates the variance and every covariances involving *rd* as if the approximate float had been updated by a function with the gradient of the value contained by the floating-point register *rs2* applied to *rs1*. This is a no-op if *rd* is not a paired register and, if *rs1* is not a paired register, UNUPG.S will write NaN to the variance of *rd*. This instruction allows efficient implementation of mathematical functions with known gradient. For example, the following assembly code stores the arcsine of `fa0` in `fa0`.

```
# Store fa0 in fs0 and asin(fa0) in fa0
unfmv.s   fs0, fa0.
call      asinf

# Calculate d(asin(x))/dx = 1/sqrt(1 - x^2) and store in fa4.
li        a0, 1
fcvt.s.w  fa3, a0
fmul.s    fa4, fs0, fs0
fsub.s    fa4, fa3, fa4
fsqrt.s   fa4, fa4
fdiv.s    fa4, fa3, fa4

# Set the uncertainty information by applying the linear
# uncertainty propagation equations to x.
unupg.s   fa0, fs0, fa4
```

Table 3.2 Uncertain Register Convention.

Register	ABI Name	Paired floating-point register	Description	Preserved across calls
`u8–9`	`us0–1`	`fs0–1`	Saved registers	Yes
`u10–11`	`ua0–1`	`fa0–1`	Arguments/return values	No
`u12–15`	`ua2–5`	`fa2–5`	Arguments	No

3.8 Uncertain ABI

The uncertain ABI defines a calling convention for procedure that use approximate floats. Table 3.2 shows the uncertain register convention which is based on the eight registers used by RISC-V compressed instructions. The uncertain procedure calling convention adds six uncertainty argument registers, the first two of which are also used to return values. The best guess of an approximate float argument is passed as if it were a scalar float — either in a floating-point register or on the stack. The corresponding uncertainty information is passed in an uncertain argument register if available, otherwise it is passed on the uncertain stack instead. Approximate floats are returned in the same manner as a first named argument of the same type would be passed. The uncertainty information associated with registers `fs0–fs11` shall be preserved across procedure calls.

This ABI is adapted from and we hope captures the intent of the RISC-V ELF psABI specification [16].

As the uncertain memory is organised separately from general purpose memory, the uncertain ABI introduces a separate uncertain stack. For ABI compatibility, the stack pointer for the uncertain stack is not stored in the integer registers which all have specific purposes in the standard ABI. Therefore, the uncertainty row stack pointer will be stored in an available control and status register (CSR).

Existing procedures adhering to the standard RISC-V hardware floating-point calling convention [16] are compatible with the uncertain ABI. Standard RISC-V procedures automatically preserve the uncertainty information of saved registers as they do not use uncertainty-aware instructions.

```
|---------------------------------------------------------------:---------------------------------------------------------------|
|      6                   5                   4                :  3                   2                   1                    |
|3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2:1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
|---------------------------------------------------------------:---------------------------------------------------------------|
|  3                   2                   1                    :  3                   2                   1                    |
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0:1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
|---------------------------------------------------------------:---------------------------------------------------------------|
|                                                               :    funct7   |   rs2   |   rs1   |  f3 |    rd   |    opcode   | R-type
|       imm[11:0]       |   rs1'  | f3' |    rd   |    opcode'  :       imm[11:0]       |    rs1  |  f3 |  opcode |    64 bit   | UI-type
|  imm[11:5]  |   rs2   |   rs1'  | f3' |    rd   |    opcode'  :       imm[11:0]       |    rs1  |  f3 |  opcode |    64 bit   | US-type
|    funct7   |   rs2   |   rs1   | f3' |    rd   |    opcode'  :     zero    |  zero   |  zero   |  f3 |  opcode |    64 bit   | UR-type
|===============================================================:===============================================================|
|                                                               :  0011100    |   rs2   |   rs1   |  rm |    rd   |   1010011   | UNUPG.S
|                                                               :  0111100    |   rs2   |   rs1   |  rm |    rd   |   1010011   | UNGCOV.S
|                                                               :  1011100    |  00000  |   rs1   |  rm |    rd   |   1010011   | UNSVAR.S
|                                                               :  1011100    |  00010  |  00000  | 000 |    rd   |   1010011   | UNCLVAR.S
|                                                               :                                                               |
|       imm[11:0]'      |   rs1'  | 010 |    rd   |   0000111   :       imm[11:0]       |    rs1  | 111 |  00001  |   0111111   | UNFLW
|  imm[11:5]' |   rs2   |   rs1'  | 010 |imm[4:0]'|   0100111   :       imm[11:0]       |    rs1  | 111 |  00001  |   0110011   | UNFSW
|                                                               :                                                               |
|    0000000  |   rs2   |   rs1   | rm' |    rd   |   1010011   :  0000000    |  00000  |  00000  |  rm |  00001  |   0110011   | UNFADD.S
|    0000100  |   rs2   |   rs1   | rm' |    rd   |   1010011   :  0000000    |  00000  |  00000  |  rm |  00001  |   0110011   | UNFSUB.S
|    0001000  |   rs2   |   rs1   | rm' |    rd   |   1010011   :  0000000    |  00000  |  00000  |  rm |  00001  |   0110011   | UNFMUL.S
|    0001100  |   rs2   |   rs1   | rm' |    rd   |   1010011   :  0000000    |  00000  |  00000  |  rm |  00001  |   0110011   | UNFDIV.S
|    0010000  |   rs2   |   rs1   | 000 |    rd   |   1010011   :  0000000    |  00000  |  00000  | 000 |  00001  |   0110011   | UNFSGNJ.S
|    0101100  |  00000  |   rs1   | rm' |    rd   |   1010011   :  0000000    |  00000  |  00000  |  rm |  00001  |   0110011   | UNFSQRT.S
|    0010000  |   rs2   |   rs1   | 001 |    rd   |   1010011   :  0000000    |  00000  |  00000  | 000 |  00001  |   0110011   | UNFSGNJN.S
|    0010000  |   rs2   |   rs1   | 010 |    rd   |   1010011   :  0000000    |  00000  |  00000  | 000 |  00001  |   0110011   | UNFSGNJX.S
|    0010100  |   rs2   |   rs1   | 000 |    rd   |   1010011   :  0000000    |  00000  |  00000  |  rm |  00001  |   0110011   | UNFMIN.S
|    0010100  |   rs2   |   rs1   | 001 |    rd   |   1010011   :  0000000    |  00000  |  00000  |  rm |  00001  |   0110011   | UNFMAX.S
|---------------------------------------------------------------:---------------------------------------------------------------|
```

Fig. 3.2 Uncertain instruction opcode listing. This table's formatting is based on the opcode listings in the XBitmanip extension specification [22].

Chapter 4

Implementing the Uncertain Framework

4.1 Uncertainty in the Sunflower Simulator

To evaluate and test the uncertain framework, the author extended the Sunflower Simulator [19], a "Full-System Hardware Emulator and Physical System Simulator for Sensor-Driven Systems". This modification to the Sunflower Simulator includes a software implementation of an uncertain ALU. The Sunflower Simulator supports the RISC-V ISA due to work by Zhengyang Gu [9] and supports the floating-point standard extension due to work by Ryan Voo [21], who both contributed to sunflower as part of coursework projects. The author built upon the existing RISC-V support in the Sunflower Simulator, adding uncertain memory and the ability to decode and evaluate the instructions defined by the uncertain extension.

The work for this project, as it was proposed, included the design of a hardware implementation of the uncertain framework to evaluate feasibility and to measure the required hardware resources. However, the author instead focused the project on the uncertain framework and the design of the uncertain extension. The author believes that the quality of any hardware implementation depends on the framework underlying it and so focused on the design of the uncertain framework and the corresponding RISC-V extension. They hope that this project will provide a foundation for future work that will include hardware designs.

4.2 Implementing an Uncertain ALU

At the core of an implementation of the uncertain extension is the uncertain ALU. For the Sunflower Simulator, the uncertain ALU was written using the C programming language and single precision floating-point arithmetic. Whereas a hardware implementation of the uncertain ALU could parallelise the covariance updates required by linear uncertainty propagation equations, the C implementation loops over each approximate register and memory location and calculates the new covariance between each pair of approximate floats. This uncertain ALU supports all the uncertainty-aware instructions defined by the uncertain RISC-V extension.

4.3 Compiling Uncertainty-Aware C

Writing assembly code is a slow, laborious and error prone process. To demonstrate the potential of the uncertain framework, this project sought to provide access to the framework from the higher level programming language C. The author considered adapting the GNU compiler, called gcc, to support generation of uncertain instructions. Due to the size and complexity of gcc (over 10 million lines of code [14]), making modification to the source code would have been a large undertaking. New uncertainty-aware instructions are designed to be similar to standard floating-point instructions, primarily to simplify decoding. By exploiting these similarities, it proved possible to build a proof of concept method to compile uncertain-aware C without modifying the C compiler. A simpler — though still complex — program, the GNU assembler gas [7], converts assembly into machine code. The author was able to modify gas to assemble the new uncertainty-aware instructions. Uncertainty-aware C is compiled to uncertain instructions in the following stages:

1. C is compiled into standard RISC-V assembly using the command line flag `-fverbose-asm`.
2. A python script converts floating-point instructions in the generated assembly into uncertainty-aware instructions.
3. The uncertain assembly is assembled by the modified version of gas.

The generated machine code contains uncertain instructions can be run by the Sunflower Simulator.

Floating-point computational assembly instructions are straightforward to convert; the script prepends the prefix ‘un’ to the mnemonic and the rest of the instruction is left unchanged. For example, the script converts the assembly `fadd.s fa0, fa0, fa1` into `unfadd.s fa0, fa0, fa1`.

Conversion of uncertain load and store instructions is more complicated as the script must infer the correct uncertain memory address to store to or load from. An uncertainty-aware compiler would generate UNFSW and UNFLW instructions with uncertain memory addresses relative to the uncertain stack pointer in the same way that a standard C RISC-V compiler generates FSW and FLW instructions with memory addresses relative to the standard stack pointer. When post-processing generated instructions, there is not enough information available to manage an uncertain stack. Instead, when the script encounters a standard load or store instruction, it reads the memory address used and replaces the standard instruction by an uncertain instruction which uses the same memory address for both the best guess and the variance of the approximate float. For example, the script converts the assembly `fsw fa0, 4(sp)` into `unfsw fa0, 4(sp), 4(sp)`. This method is made possible by the uncertain implementation in the Sunflower Simulator providing one uncertain memory location for every four bytes of available standard memory. Thus for every unique memory address of an aligned floating-point store there is a unique index in uncertain memory. The program can be guaranteed to use only aligned memory addresses by invoking gcc with the command line flag `-mstrict-align`. Uncertain memory is incredibly costly to provide as described in Section 2.6. In an uncertain framework that is realisable in hardware, the number of uncertain memory location must be far smaller than the number of normal memory addresses. For a simulator implementation demonstrating a proof of concept, the drawbacks of this wasteful use of uncertain memory are outweighed by the simplification it provides.

4.4 Library Abstractions for the Uncertain Extension

To provide access to uncertainty information in C, the uncertain library defines a high level interface to uncertain instructions. This library contains the functionality needed to build the test cases used in Section 5.3. An uncertainty-aware compiler could remove the function call overhead associated with these abstractions by defining the functions exposed by this library as compiler builtins [8].

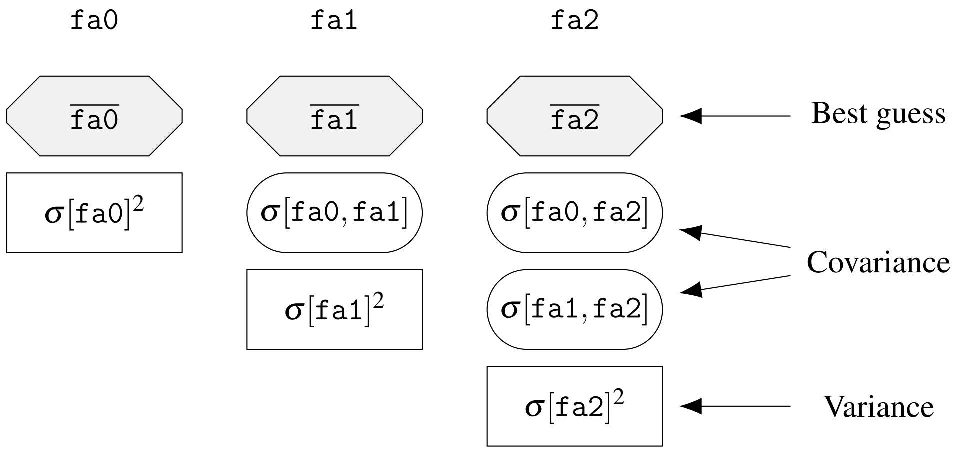
The uncertain library defines a new arithmetic type `approximate_float`. The library should define this type in such a way that it cannot implicitly be converted into a float. However, defining `approximate_float` in such a way would prevent the C arithmetic operators from working with approximate floats. Instead, the library defines `approximate_float` through a typedef, allowing users to interchange `float` and `approximate_float`. This is not desired behaviour, instead programmers should use the function `unf_best_guess()`, defined as part of the uncertain library, to retrieve the best guess of an approximate float. Compiler support for the uncertain extension is needed to define an `approximate_float` type that are distinct from the standard C `float` type but compatible with the arithmetic operators. For the test cases, the author was careful to avoid any implicit conversions.

Chapter 5

Results and Discussion

5.1 Evaluating Uncertain Assembly

This section demonstrates how an uncertain RISC-V assembly snippet that adds and multiplies paired floating-point registers (using the UNFADD.S and UNFMUL.S instructions) is evaluated by a processor implementing the uncertain RISC-V extension. The assembly is divided into five chunks and to the right of each chunk is the register state after the processor has evaluated that chunk. Figure 5.1 shows how the state is represented. Registers shown with a bold outline have been updated whilst evaluating a snippet.

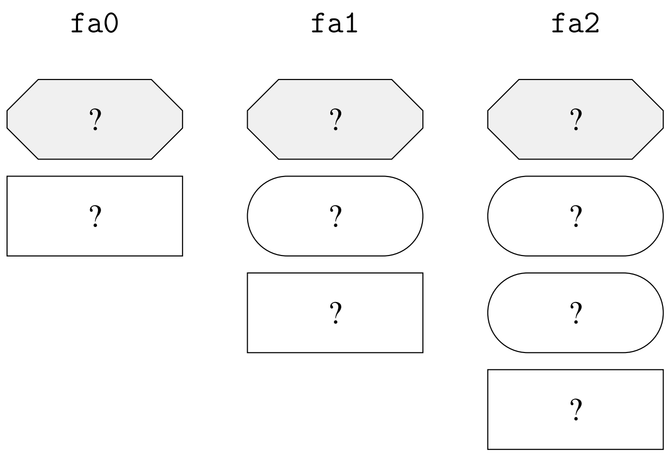


fa0
fa1
fa2
$\overline{\texttt{fa0}}$
$\overline{\texttt{fa1}}$
$\overline{\texttt{fa2}}$
Best guess
$\sigma[\texttt{fa0}]^2$
$\sigma[\texttt{fa0},\texttt{fa1}]$
$\sigma[\texttt{fa0},\texttt{fa2}]$
Covariance
$\sigma[\texttt{fa1}]^2$
$\sigma[\texttt{fa1},\texttt{fa2}]$
$\sigma[\texttt{fa2}]^2$
Variance

Fig. 5.1 Diagrammatic representation of the register state.

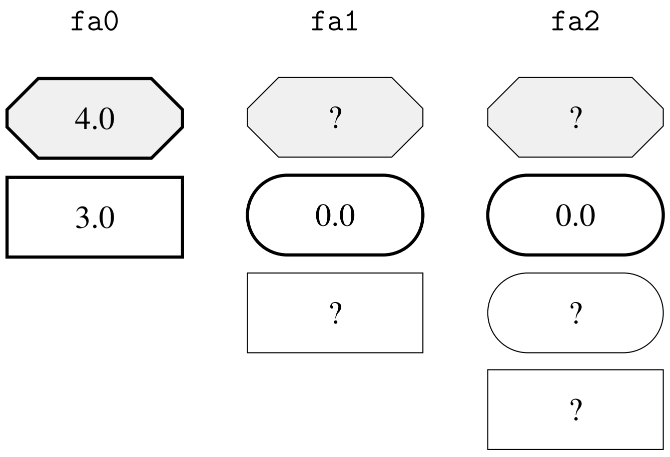
The assembly begins with four standard RISC-V instructions that load the floating-point value 3.0 into `ft0` and the floating-point value of 4.0 into `ft1`. As the registers `ft0` and `ft1` do not have any associated uncertainty information, nor will the floating-point values in these registers change, they are not shown in the diagram. At the start of this snippet, the registers `fa0`, `fa1` and `fa2` contain an undefined initial value which is represented using a question mark.

```
.cthree:
    .word 0x40400000
.cfour:
    .word 0x40800000
example:
    lui a5, %hi(.cthree)
    flw ft0, %lo(.cthree)(a5)
    lui a5, %hi(.cfour)
    flw ft1, %lo(.cfour)(a5)
```



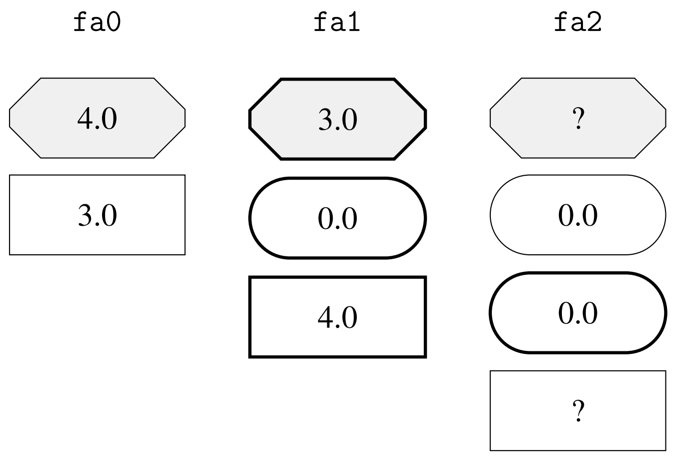
The programmer uses `ft0` and `ft1` to define the uncertainty information of `fa0`. They use the standard RISC-V FMV.S pseudo instruction to set the best guess of `fa0` and the uncertain UNSVAR.S instruction to set the variance. When executing the UNSVAR.S instruction, the uncertain ALU sets the covariance between `fa1` and all other paired registers to zero.

```
    fmv.s fa0, ft1
    unsvar.s fa0, ft0
```



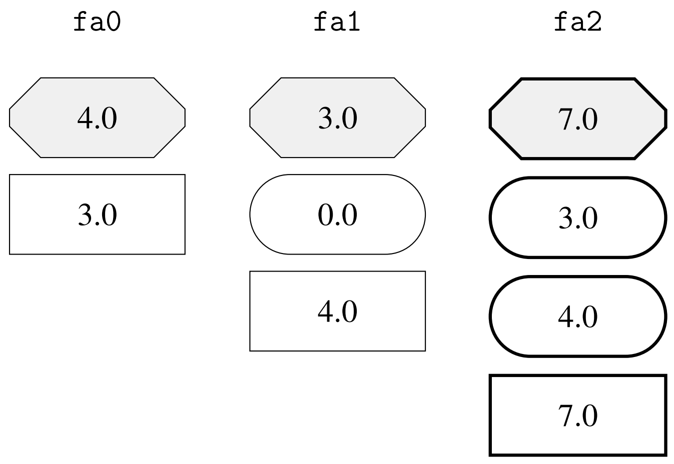
Next, the programmer uses the UNSVAR.S instruction again to set the variance of `fa1`.

```
fmv.s fa1, ft0
unsvar.s fa1, ft1
```



The programmer then adds `fa0` and `fa1` and writes the result to `fa2`. The ALU uses linear uncertainty propagation equations to update the uncertainty information with function $f(x,y) = x+y$. The partial derivative of f with respect to both x and y is one.

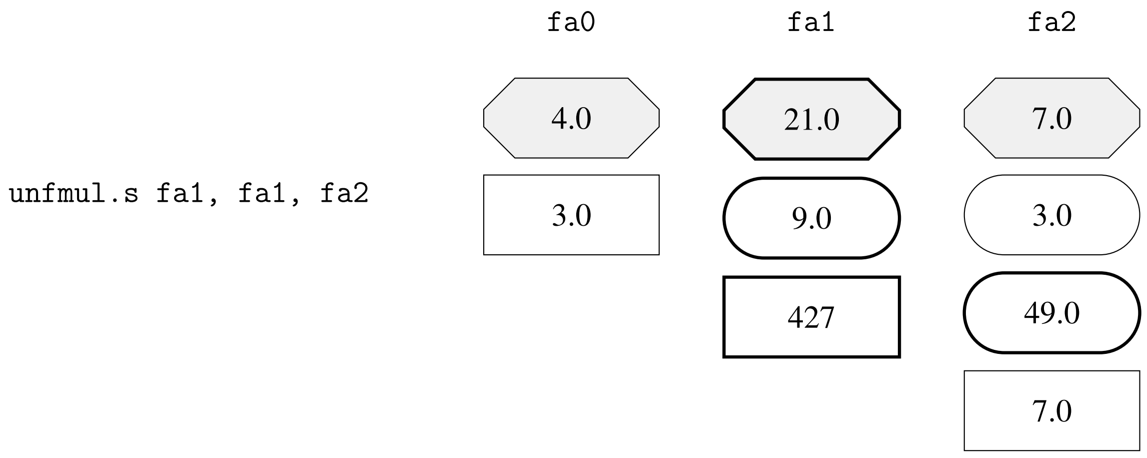
```
unfadd.s fa2, fa0, fa1
```



Finally, the programmer uses the uncertain ALU to calculate the product of `fa1` and `fa2` and write the result to `fa1`. In this case,

$$f(x,y) = xy, \ \frac{\partial f(\bar{x},\bar{y})}{\partial x} = \bar{y} \text{ and } \frac{\partial f(\bar{x},\bar{y})}{\partial y} = \bar{x}.$$

The uncertain ALU both reads from and writes to the floating-point paired register `fa1`.



The final variance of `fa1` is equal to that calculated using the linear uncertainty propagation equations with $f(x,y) = y/(x+y)$. The programmer has correctly propagated uncertainty through this assembly snippet using uncertainty-aware instructions.

These arithmetic operations have introduced correlation between the result and the inputs. If the ALU had ignored the covariance between `fa1` and `fa2` it would have calculated the new variance of `fa1` as 259 instead of 427. This suggests that accurate uncertainty propagation does require the processor to store and update covariances. Section 5.4 discusses the implications of ignoring correlation in more detail.

5.2 Accuracy of the Linear Uncertainty Propagation

Two approximations are made by the linear uncertainty propagation Equations 2.2, 2.3 and 2.4. Firstly, the equations assume that the expected value of a function of random variables is approximated by the function evaluated at the expected value of each random variable. The second assumption is that higher order moments (for example the skew or the kurtosis) of x and y have a negligible impact on the variance of z. These assumptions are known to hold if the f is approximately linear in the region where most of the pass of the joint probability density function of x and y is location [4].

This section derives bounds on the errors introduced in the estimation of the mean and variance of $z = f(x)$ using the linear uncertain propagation equation. TODO TODO It additionally provides conditions that guarantee that the relative errors will be small. The algebra is similar for a bivariate function although there are significantly more terms.

Theorem 5.1. *The error introduced by estimating the mean of $z = f(x)$ using the linear uncertainty propagation equations is*

$$\sum_{n=1}^{\infty} \frac{\partial^n f}{\partial x^n} \frac{E\left[\delta x^n\right]}{n!} \tag{5.1}$$

Proof. Let $\bar{x}$ be the expected value of x and $\delta x = x - \bar{x}$. Using the taylor series expansion, we can write the expected value of z as

$$\begin{aligned} E[z] &= E\left[f(x)\right] \\ &= f(\bar{x}) + E\left[\sum_{n=1}^{\infty} \frac{\partial^n f}{\partial x^n} \frac{\delta x^n}{n!}\right] \\ &= \bar{z} + \sum_{n=1}^{\infty} \frac{\partial^n f}{\partial x^n} \frac{E\left[\delta x^n\right]}{n!} \\ &\approx \bar{z} \end{aligned} \tag{5.2}$$

where $\bar{z}$ is estimated value of $E[z]$ and all derivatives are evaluated at $\bar{x}$. Equation 5.1 is the error $E[z] - \bar{z}$. □

By comparing the magnitude of each term in the sum to the magnitude of $\bar{z}$ we have the following:

Corollary 5.2. *The relative error $|E[z] - \bar{z}|/E[z]$ is guaranteed to be are small if*

$$\left|\frac{\partial^n f(\bar{x})}{\partial x^n} E\left[\delta x^n\right]\right| \ll |f(\bar{x})|\, n! \qquad \textit{for all } n = 1, 2, \ldots. \tag{5.3}$$

Corollary 5.2 provides no bound on the error if the any of the moments of a distribution are infinite. In practice using the linear uncertainty propagation equations with pathological distributions such as the Cauchy distribution will give large errors. Additionally, if $f(\bar{x}) = 0$ then Equation 5.3 cannot be satisfied and the relative error may be very large.

Theorem 5.3. *Let $F(n)$ be a function defined*

$$F(n) = \frac{\sqrt{E\left[\delta x^{2n}\right] - E\left[\delta x^n\right]^2}}{\sigma_x^n \cdot n!} = \frac{\sqrt{E\left[\left(\delta x^n - E\left[\delta x^n\right]\right)^2\right]}}{\sigma_x^n \cdot n!}. \tag{5.4}$$

The error due to estimating the variance of $z = f(x)$ using the linear uncertainty propagation equations is bounded from above by

$$\sum_{k=3}^{\infty}\sum_{n=1}^{k-1}\left|f^{(k-n)}\right|\sigma_x^{k-n}F(k-n)\cdot\left|f^{(n)}\right|\sigma_x^{n}F(n). \tag{5.5}$$

The proof requires the following lemma:

Lemma 5.4. *For a random variable X*

$$|E[X]| \leq E\left[|X|\right] \tag{5.6}$$

$$E\left[|X||Y|\right] \leq \sqrt{E\left[|X|^2\right]E\left[|Y|^2\right]}. \tag{5.7}$$

Equation 5.6 follows from Jensen's inequality [12] where $|\cdot|$ is a convex function. Equation 5.7 is a special case of Hölder's inequality [10] with $p = q = 1$.

Proof. Let $f^{(n)} = \dfrac{\partial^n f(\bar{x})}{\partial x^n}$ be the partial derivative of f with respect to x evaluated at $\bar{x}$. We have

$$\begin{aligned}
\mathrm{var}[z] &= E\left[(z-\bar{z})^2\right] \\
&= E\left[\left(\sum_{n=1}^{\infty} f^{(n)}\frac{(\delta x^n - E\left[\delta x^n\right])}{n!}\right)^2\right] \\
&= \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} f^{(m)}f^{(n)}E\left[\frac{(\delta x^m - E\left[\delta x^m\right])}{m!}\frac{(\delta x^n - E\left[\delta x^n\right])}{n!}\right].
\end{aligned} \tag{5.8}$$

Let $k = n + m$. Then we perform a change of variables to give

$$\begin{aligned}
\mathrm{var}[z] &= \sum_{k=2}^{\infty}\sum_{n=1}^{k-1} f^{(k-n)}f^{(n)}E\left[\frac{\left(\delta x^{k-n} - E\left[\delta x^{k-n}\right]\right)}{(k-n)!}\frac{(\delta x^n - E\left[\delta x^n\right])}{n!}\right] \\
&= \left(\frac{\partial f}{\partial x}\sigma_x\right)^2 + \sum_{k=3}^{\infty}\sum_{n=1}^{k-1} f^{(k-n)}f^{(n)}E\left[\frac{\left(\delta x^{k-n} - E\left[\delta x^{k-n}\right]\right)}{(k-n)!}\frac{(\delta x^n - E\left[\delta x^n\right])}{n!}\right] \\
&\approx \sigma_z^2.
\end{aligned} \tag{5.9}$$

Using Lemma 5.4 we have

$$\begin{aligned}
\left|var[z]-\sigma_z^2\right| &\overset{(a)}{\leq} \sum_{k=3}^{\infty}\sum_{n=1}^{k-1}\left|f^{(k-n)}f^{(n)}E\left[\frac{\left(\delta x^{k-n}-E\left[\delta x^{k-n}\right]\right)}{(k-n)!}\frac{\left(\delta x^{n}-E\left[\delta x^{n}\right]\right)}{n!}\right]\right| \\
&\overset{(b)}{\leq} \sum_{k=3}^{\infty}\sum_{n=1}^{k-1}\left|f^{(k-n)}\right|\left|f^{(n)}\right|E\left[\left|\frac{\left(\delta x^{k-n}-E\left[\delta x^{k-n}\right]\right)}{(k-n)!}\right|\left|\frac{\left(\delta x^{n}-E\left[\delta x^{n}\right]\right)}{n!}\right|\right] \\
&\overset{(c)}{\leq} \sum_{k=3}^{\infty}\sum_{n=1}^{k-1}\left|f^{(k-n)}\right|\left|f^{(n)}\right|\frac{\sqrt{E\left[\left|\delta x^{k-n}-E\left[\delta x^{k-n}\right]\right|^2\right]E\left[\left|\delta x^{n}-E\left[\delta x^{n}\right]\right|^2\right]}}{(k-n)!n!} \\
&= \sum_{k=3}^{\infty}\sum_{n=1}^{k-1}\left|f^{(k-n)}\right|\sigma_x^{k-n}F(k-n)\cdot\left|f^{(n)}\right|\sigma_x^{n}F(n) \qquad (5.10)
\end{aligned}$$

Where step (a) follows from the triangle inequality, step (b) follows from Equation 5.6 and step (c) follows from Equation 5.7. □

Again, we consider the magnitude of each term in the sum, this time comparing it to the magnitude of σ_z^2 to give the following corollary:

Corollary 5.5. *The relative error, $\left|var[z]-\sigma_z^2\right|/var[z]$, is guaranteed to be small if*

$$\left|\frac{\partial^n f(\bar{x})}{\partial x^n}\right|\sigma_x^n F(n) \ll \left|\frac{\partial f(\bar{x})}{\partial x}\right|\sigma_x \qquad (5.11)$$
$$\textit{for all } n = 3,4,\ldots.$$

Numerical data deriving from measurements often has an approximately Gaussian distribution. Measuring physical phenomena typically involves averaging measured values which, by the central limit theorem, produces a result with an approximately Gaussian distribution. We therefore consider the application of Corollaries 5.2 and 5.5 for normally distributed values. If x has a Gaussian distribution, the central moments of x are

$$E\left[\delta x^n\right] = \begin{cases} 0 & \text{if } n \text{ is odd} \\ \sigma_x^n\cdot(n-1)\cdot(n-3)\cdots 3\cdot 1 & \text{if } n \text{ is even.}\end{cases} \qquad (5.12)$$

Corollary 5.2 guarantees the estimate of an approximate float is accurate if

$$\left|\frac{\partial^n f(\bar{x})}{\partial x^n}\frac{\sigma_x^n}{n\cdot(n-2)\cdots 4\cdot 2}\right| \ll |f(\bar{x})| \tag{5.13}$$

$$\left|\frac{\partial^n f(\bar{x})}{\partial x^n}\right|\sigma_x^n \ll |f(\bar{x})|\left(\frac{n}{2}\right)!\cdot 2^{n/2} \tag{5.14}$$

$$n = 2,4,\ldots$$

as Equation 5.3 is automatically satisfied for all odd n.

To apply Corollary 5.5 when x has a Gaussian distribution, we first substitute Equation 5.12 into Equation 5.4 to give

$$F(n) = \frac{\sqrt{\begin{cases}\frac{2n-1}{n-1}\cdot\frac{2n-3}{n-3}\cdots\frac{n+4}{4}\cdot\frac{n+2}{2} & \text{if } n \text{ is odd}\\ \frac{2n-1}{n-1}\cdot\frac{2n-3}{n-3}\cdots\frac{n+3}{3}\cdot\frac{n+1}{1}-1 & \text{if } n \text{ is even.}\end{cases}}}{n(n-2)(n-4)\cdots} \tag{5.15}$$

Therefore, the conditions in Equation 5.11 are satisfied if

$$\frac{\left|\frac{\partial^n f(\bar{x})}{\partial x^n}\right|\sigma_x^{n-1}}{\left|\frac{\partial f(\bar{x})}{\partial x}\right|} \ll \frac{n(n-2)(n-4)\cdots}{\sqrt{\begin{cases}\frac{2n-1}{n-1}\cdot\frac{2n-3}{n-3}\cdots\frac{n+4}{4}\cdot\frac{n+2}{2} & \text{if } n \text{ is odd}\\ \frac{2n-1}{n-1}\cdot\frac{2n-3}{n-3}\cdots\frac{n+3}{3}\cdot\frac{n+1}{1}-1 & \text{if } n \text{ is even}\end{cases}}} \tag{5.16}$$

$$\text{for all } n = 3,4,\ldots.$$

If the distribution describing the uncertainty of an approximate value is approximately Gaussian, the assumptions of the linear uncertainty propagation equations hold if $\sigma_x \ll |\bar{x}|$ and the magnitude of higher order derivatives of f are smaller than the magnitude of the first derivative. Regardless of x's underlying distribution, the estimate of the variance will have a significant error when the derivative of f with respect to x is zero. This will occur if $\bar{x}$ is a stationary point of $f(x)$.

5.3 Newton-Raphson Test

The quadratic equation

$$x^2 + 2x - 6 = 0 \tag{5.17}$$

has two solutions at $x = -1 \pm \sqrt{6}$. Using the Newton-Raphson method, the equation

$$x_{n+1} = x_n - \frac{x^2 + 2x - 6}{2x + 2} \tag{5.18}$$

can be applied iteratively with $x_0 = 0$ and will converge to the positive solution $-1 + \sqrt{7}$. Let the variance of the coefficient of x be σ_b^2 and the variance of the constant term be σ_c^2. The coefficient of x and the constant term may be correlated with a covariance of σ_{bc}. Applying the linear uncertainty propagation equations (defined Section 2.4) to the quadratic equation, the variance of the positive root is

$$\sigma_x^2 = 0.09673\sigma_b^2 - 0.11755\sigma_{bc} + 0.03571\sigma_c^2, \tag{5.19}$$

and the covariance between the positive root and each coefficient is

$$\sigma_{xb} = -0.31102\sigma_b^2 + 0.18898\sigma_{bc} \tag{5.20}$$

$$\sigma_{xc} = -0.31102\sigma_{bc} + 0.18898\sigma_c^2. \tag{5.21}$$

The uncertain extension is tested by comparing the variance of the result of a numerical solution to Equation 5.17 to the values predicted by the linear uncertainty propagation equations. A small uncertainty-aware C program, running in the adapted Sunflower Simulator, solved the quadratic equation for a various values of σ_b^2, σ_c^2 using the Newton-Raphson method. The following values of the covariance, σ_{bc}, were used: 0, $\sigma_b\sigma_c$, $-\sigma_b\sigma_c$ and, when $\sigma_c > \sigma_b$, σ_b. (The uncertain extension does not allow programmers to directly set the covariance between two approximate values. These values of σ_{bc} can be achieved using uncertainty-aware arithmetic.)

This test was run for 75 combinations of σ_b, σ_c and σ_{bc} and the percentage errors between the values of σ_x^2, σ_{xb}, σ_{xc} resulting from the Newton-Raphson solution and the analytical solution were measured. The percentage errors were less than 0.1% in all but three out of the 75 cases.

The percentage error for σ_x^2 exceeded 0.1% when the correlation between b and c was one. (Twice the error was 0.3% and once it was 3%.) When the correlation between two approximate floats is one there may be subtractive cancellation in the linear uncertainty propagation equations. This cancellation leads to numerical errors in floating-point calculations, causing the large percentage error seen here. Future work on the uncertain extension is required to address these cancellation errors. The author believes that a combination of improved ALU design and a more suitable binary format for uncertainty would solve this issue.

Aside from the relatively small arising from subtractive cancellation in the linear uncertainty propagation equations, this test gives confidence that the implementation of the uncertain extension is correct and that the framework is viable. Moreover, the only changes required to adapt the C code to make use of the uncertain framework were the initialisation of values using `unf_create()`. The program displayed uncertainty information using `printf()`, `unf_var()` and `unf_covar()`.

The fifth design criteria requires that the uncertain extension is simple to use for a programmer with little to no prior experience of uncertainty propagation. The Newton-Raphson test case gives confidence that the uncertain framework satisfies this criteria. Currently compiling and running uncertainty-aware code in the Sunflower Simulator is possible but not straight forward. However, these difficulties are not fundamental to the uncertain framework and future work on the uncertain extension will provide a toolchain that is intuitive and simple to use.

5.4 The Uncorrelated Approximation

Section 2.3 noted that is may be viable to avoid storing the covariance between approximate floats. The covariance can be simply ignored when computing the variance of the result of an arithmetic operation. Ignoring the covariance is equivalent to assuming the inputs to the operation are uncorrelated. Using the Newton-Raphson test case, this approach can be compared with the full covariance method used by the uncertain framework.

The author removed two lines of code from the C implementation of the uncertain ALU so that, when updating the variance, the ALU neglects the covariance term in Equation 2.2. The effect of this change was measured using Newton-Raphson test from 5.3. When the ALU used the full covariance information the variance of x, the calculated root of the quadratic equation, converged to the value predicted by the linear uncertainty propagation equations. When the ALU neglected the covariance term in Equation 2.2, the variance of x increased by about 33% with each iteration after the best guess of x had converged to the correct value. After running the algorithm to convergence (5 iterations in total), the estimated variance of x was consistently greater than 25 times the true value and sometimes over a thousand times the true value. Based on the Newton-Raphson test, neglecting covariance information causes the uncertain framework to overestimate variances which gives a conservative estimate. For some applications, errors of a couple of orders of magnitude are acceptable as long as they cause the variance to increase.

The second method to avoid storing covariances is to use the upper bound of Equation 2.6. This upper bound assumes that the correlation between the inputs to an arithmetic operation is either 1 or -1 depending on the sign of the two gradients. After adapting the ALU to use the upper bound for the variance, the Newton-Raphson test was run. As when neglecting the covariance, the estimated variance of x increased with each iteration of the Newton-Raphson algorithm even after convergence of the best guess. In this case the variance increased by about a factor of four with each iteration. When running Newton-Raphson to convergence (again 5 iterations), the estimated variance was always at least five thousand times larger than the true value.

The author believes that the magnitude of these errors supports the decision to design an uncertain framework that stores covariances. If the variance is over estimated by an order of magnitude when running a simple calculation, the variance will grow too fast to be useful in any practical application.

Chapter 6

Conclusions

This report introduces the uncertain framework and proposes a non-standard extension to the RISC-V ISA providing hardware support for uncertainty propagation. The author, after researching methods for representing and propagating uncertainty, found the linear uncertainty propagation equations to be the best basis for the framework. The approximate float type encodes the best guess of a floating-point number and uncertainty information in the form of a variance. Therefore, an approximate float is suitable for representing measured, estimated or calculated values that may include errors.

This report proposes the uncertain RISC-V extension and defines new uncertainty-aware instructions. Programmers can combine the best guess of a value with a known or estimated variance to create approximate floats. Using uncertainty-aware instructions and the uncertain ALU, algorithms can calculate the variance of their outputs as well as giving the best guess of the output's true value. The uncertainty in calculated values is estimated by propagating variance and covariances through the algorithm as the uncertain ALU evaluates uncertainty-aware instructions. The ALU uses the linear uncertainty propagation equations to calculate the variances and covariances resulting from arithmetic operations.

The author implemented the uncertain extension as part of the Sunflower Simulator [19]. In addition to setting a foundation for a future hardware implementation of the uncertain extension, this implementation demonstrates uncertainty propagation using the uncertain extension to the RISC-V ISA. By adapting the GNU assembler, creating pre- and post-processing scripts and implementing the uncertain library, the author enabled programmers to use the uncertain framework in C.

This project evaluates the uncertain framework (as implemented in the Sunflower Simulator) using the Newton-Raphson solution of a quadratic equation. This evaluation finds that the accuracy of the estimated variance is vulnerable to floating-point rounding errors. Thus, the binary representation of covariances needs be be carefully chosen to avoid vulnerability to rounding and the design of uncertain ALU should avoid rounding intermediate values. The Newton-Raphson test showed that adapting existing algorithms to use the uncertain framework requires no changes to the algorithm and only minimal changes to the surrounding code.

Finally, this report considers the criteria of Section 2.1 used to design the uncertain framework. This report proposes an ISA extension that is consistent with and compliments the existing RISC-V standards. As the best guess calculated by an uncertainty-aware computational instruction is exactly the single-precision result of the corresponding standard floating-point instruction, the uncertain ISA extension has no cost in accuracy. A full evaluation of the execution speed and the hardware requirements of the uncertain extension is left as future work. However, the author believes that, with careful hardware design, a performant and efficient implementation is possible. The hardware requirements of the uncertain extension look to be considerable, the author believes this to be an unavoidable problem for methods supporting uncertainty propagation in hardware. The framework developed in this report is compatible with the RISC-V architecture and is simple to integrate into existing software. The report concludes that the uncertain framework built upon the uncertain extension and supported by the uncertain ALU meets the design criteria.

References

[1] Kai O Arras. 1998. *An Introduction To Error Propagation*. Technical Report.

[2] James Bornholt. 2013. Abstractions and Techniques for Programming with Uncertain Data.

[3] V. Camus, J. Schlachter, C. Enz, M. Gautschi, and F. K. Gurkaynak. 2016. Approximate 32-bit floating-point unit design with 53% power-area product reduction.

[4] A.A. Clifford. 1973. *Multivaraite Error Analysis*. Applied Science Publishers Ltd.

[5] Mosè Giordano. 2016. Issue: Arithmetic operations very slow. https://github.com/JuliaPhysics/Measurements.jl/issues/25

[6] Mosè Giordano. 2016. Uncertainty propagation with functionally correlated quantities.

[7] GNU . 2019. The GNU Assembler, GAS. https://sourceware.org/binutils/docs-2.32/as/

[8] GNU Project. 2019. Other Built-in Functions Provided by GCC. https://gcc.gnu.org/onlinedocs/gcc-9.1.0/gcc/Other-Builtins.html

[9] Zhengyang Gu. 2018. Pull request: Merge in RISCV support. https://github.com/phillipstanleymarbell/sunflower-simulator/pull/28

[10] O. Hölder. 1889. Ueber einen Mittelwertsatz. *Gött. Nachr.* 1889 (1889), 38–47.

[11] IEEE 2008. ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic.

[12] J. L. W. V. Jensen. 1906. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta Math.* 30 (1906), 175–193.

[13] Robert W Keener. 2010. *Theoretical Statistics: Topics for a Core Course*.

[14] Michael Larabel. 2015. GCC Soars Past 14.5 Million Lines Of Code & I'm Real Excited For GCC 5. https://www.phoronix.com/scan.php?page=news_item&px=MTg3OTQ

[15] NIST. 2019. The NIST Uncertainty Machine. https://uncertainty.nist.gov/

[16] RISC-V Foundation. 2018. RISC-V ELF psABI specification. https://github.com/riscv/riscv-elf-psabi-doc

[17] RISC-V Foundation. 2018. *The RISC-V Instruction Set Manual*. https://github.com/riscv/riscv-isa-manual/releases/riscv-user-2.2

[18] RISC-V Foundation. 2019. RISC-V "V" Vector Extension. https://github.com/riscv/riscv-v-spec/tree/9688e01837c512466ddd73ac9f79147a1fe1b2d8

[19] Phillip Stanley-Marbell and Diana Marculescu. 2007. Sunflower: Full-system, Embedded, Microarchitecture Evaluation *(HiPEAC'07)*.

[20] J Taylor. 1997. *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*. University Science Books.

[21] Ryan Voo. 2019. Pull request: rv32FD support. https://github.com/phillipstanleymarbell/sunflower-simulator/pull/41

[22] Clifford Wolf. 2018. RISC-V Bitmanip Extension. https://github.com/riscv/riscv-bitmanip/releases/tag/v0.36

Appendix A

Risk Assessment Retrospective

The “4th Year Project Hazard Assessment Form” identified risks from computer use and lone working. The work on this project consisted of design, programming and discussion. I encountered no hazards during my work other than the two identified at the start of the year. If starting the project again, I would assess the risk in a similar manor.