Taking architecture and compiler into account in formal proofs of numerical programs

Thi Minh Tuyen Nguyen

▶ To cite this version:

Thi Minh Tuyen Nguyen. Taking architecture and compiler into account in formal proofs of numerical programs. Other [cs.OH]. Université Paris Sud - Paris XI, 2012. English. NNT : 2012PA112090 . tel-00710193

HAL Id: tel-00710193
https://theses.hal.science/tel-00710193v1

Submitted on 20 Jun 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° NNT: 2012PA112090

UNIVERSITÉ DE PARIS-SUD

ÉCOLE DOCTORALE: INFORMATIQUE PARIS-SUD

DISCIPLINE : INFORMATIQUE

THÈSE DE DOCTORAT

soutenue le 11 juin 2012

par

Thi Minh Tuyen NGUYEN

Preuves formelles de programmes numériques en prenant en compte l'architecture et le compilateur

Composition du jury:

Président :	Jean-Michel MULLER
Rapporteurs:	David MONNIAUX César A. MUÑOZ
Examinateurs	Xavier LEROY Florent HIVERT Claude MARCHÉ, co-directeur Sylvie BOLDO, co-directrice

Acknowledgements

This thesis is the result of more than three years of work under the supervision of Sylvie Boldo and Claude Marché, to who I would like to express my sincere gratitude. They gave me a chance to work in a big team. I would like to thank to Sylvie for her helps from the beginning of my PhD. She was always available to discuss with me each time I needed her. I would like to thank to Claude for his guidance. Spending time working with him gave me more ideas to finish my thesis.

I also thank my reviewers David Monniaux and Cesar A. Muñoz who spent much of their time to understand my memoir and gave me constructive comments. I especially thank Cesar Muñoz who came to Paris to attend my defence.

Not forget, great appreciations go to ProVal team. This is the best team that I've ever worked with. I also thank Régine for her helps in setting up my life. I thank Guillaumne for his helps in using Gappa tool. Thanks Romain, François, Asma, Kalyan, etc.

I would like to thank the people in Hisseo project who gave me helpful ideas for improving my work.

The special thank goes to my father, mother, brothers and sisters who shared with me all my difficulties. They gave me the motivation to finish my PhD.

Last but not least, thanks Luyen, my husband and my son, Minh Thien.

Contents

1 Introduction **9**
- 1.1 Floating-point arithmetic . . . 9
- 1.2 Formal verification . . . 10
- 1.3 Contributions . . . 11

2 Background **13**
- 2.1 Floating-point arithmetic . . . 13
  - 2.1.1 The IEEE-754 floating-point standard . . . 13
  - 2.1.2 Architecture-dependent issues . . . 16
- 2.2 Why- An Intermediate Verification Language . . . 18
  - 2.2.1 Logical declarations in Why . . . 20
  - 2.2.2 Programs in Why . . . 20
  - 2.2.3 Example . . . 21
  - 2.2.4 Use of Why in this thesis . . . 23
- 2.3 ACSL and Frama-C . . . 25
  - 2.3.1 ACSL . . . 25
  - 2.3.2 Frama-C . . . 25

I Hardware-independent proofs **27**

3 Hardware-independent bounds for floating-point operations **29**
- 3.1 Bounds for one floating-point operation . . . 29
  - 3.1.1 Case 1: Rounding error in 64-bit rounding . . . 30
  - 3.1.2 Case 2: Rounding error in 80-bit rounding . . . 30
  - 3.1.3 Case 3: Rounding error in double rounding . . . 31
  - 3.1.4 Proof in Coq . . . 31
- 3.2 Proofs of numerical programs . . . 32
  - 3.2.1 When FMA occurs . . . 32
  - 3.2.2 Bounds of a sequence of operations . . . 33
- 3.3 Implementation . . . 33

4 When the compiler re-organizes a computation **35**
- 4.1 Associativity for the addition . . . 35
- 4.2 Implementation . . . 41

5 Experimentations **43**
- 5.1 Double rounding example . . . 43
- 5.2 KB3D example . . . 44

5.3 Summation example 46
5.4 Clock drift example 47
5.5 Scalar product example 49

II Hardware-dependent proofs **51**

6 Principle of proofs on assembly code with Why **53**
6.1 Steps of proofs 53
6.2 Essential elements of assembly language 54
6.2.1 Operands and Instruction Naming 54
6.2.2 EFLAGS register 55
6.2.3 General-purpose instructions 56
6.2.4 Calling procedures using `call` and `ret` 56
6.2.5 Some assembler directives 57
6.2.6 Inline assembly 58
6.3 Preparation of source code 59
6.4 Examples 61

7 Case of Simple programs **65**
7.1 Definition of the class of "simple" C programs 65
7.2 Translation to Why 65
7.2.1 Translation of 32-bit and 64-bit integers 65
7.2.2 Translation of operands 66
7.2.3 Annotations 68
7.2.4 Translation of an instruction 69
7.2.5 Sequences and functions 70
7.3 Soundness of translation 71
7.3.1 Reminder of the soundness of Why 72
7.3.2 About the condition in function call 73
7.3.3 Definition of the execution of an assembly program 74
7.3.4 Relation between the Why state and the assembly state 75
7.4 Examples 77
7.4.1 Simple example 77
7.4.2 Square example 78

8 Floating-point programs **83**
8.1 Assembly with floating-point arithmetic 83
8.1.1 SSE/SSE2 83
8.1.2 x87 Floating-point Unit 84
8.1.3 FMA 86
8.2 Definition of programs supported 87
8.3 Translation to Why 89
8.3.1 Abstract functions 89
8.3.2 When constants are referenced by `%rip` 89
8.3.3 Modifying the translation of general-purpose instructions 90
8.3.4 Translation of SSE/SSE2 instructions 91
8.3.5 x87 Floating-point Unit 92
8.3.6 AVX instructions 95
8.4 Translation of annotations to Why 96

8.4.1 Translation of annotations in presence of floating-point arithmetic 96
8.5 Soundness of translation 97
8.5.1 Definition of the execution of a assembly program 97
8.5.2 Relation between Why state and assembly state (case of floating-point programs) 98
8.5.3 About exact value 99
8.6 Illustrations 101
8.6.1 Double rounding example 101
8.6.2 Overflow example 104

9 Handling Conditional and loop statements 109
9.1 Conditional instructions in assembly 109
9.1.1 Jump instructions 109
9.1.2 Conditional move instructions: CMOVcc 110
9.2 Definition of programs supported 110
9.3 Translation of comparison instructions 110
9.3.1 Translation of `cmp` instruction 111
9.3.2 Translation of floating-point comparison instructions 111
9.4 Control Flow Graph construction from assembly code 112
9.4.1 Example with `if` statement 112
9.4.2 Example with `do while` statement 114
9.4.3 Example with `goto`, `do while` and `if` statement 114
9.5 Translation from a CFG to Why 115
9.6 Examples 120
9.6.1 Clock drift 120
9.6.2 KB3D 123
9.7 Discussion 123

10 Handling Arrays and Pointers 127
10.1 Handled programs 127
10.2 New rules of translation for operands and instructions 127
10.2.1 Representation of memory in Why 127
10.2.2 Definition of memory model 134
10.2.3 Translation of operands and instructions to Why 134
10.2.4 Translation of annotations to Why 138
10.3 When local variables are pointed by `%rsp` 139
10.4 Examples 141
10.4.1 Maximum of an array 141
10.4.2 Scalar Product 142

11 Bit-level reasoning 147
11.1 Motivations 147
11.1.1 Examples of the chapter 147
11.1.2 Goals of the chapter 149
11.2 About endianness 149
11.3 Why 3 150
11.3.1 Logic 150
11.3.2 Theories 151
11.4 Bitvector theories in Why 3 153
11.4.1 Theory BitVector 153

11.4.2 Theory BV32 and BV64 . . . 156
11.4.3 Theory BV32_64 . . . 157
11.4.4 Theory BV_double . . . 157
11.5 Examples . . . 159
11.5.1 Negation by xor . . . 159
11.5.2 Conversion of an integer to a double . . . 160
11.6 Discussion . . . 164

12 Conclusion and Future works 165
12.1 Summary . . . 165
12.1.1 Hardware-independent approach . . . 165
12.1.2 Hardware-dependent approach . . . 166
12.1.3 Comparison with related works . . . 166
12.1.4 Difficulties . . . 167
12.2 Future works . . . 167
12.2.1 Hardware-independent approach . . . 167
12.2.2 Hardware-dependent approach . . . 168

Chapter 1

Introduction

The problem of program correctness is always an issue, especially as software applications are more and more complex. Moreover, software correctness plays a crucial role in the overall software quality especially in the automatic control of engines like airplanes or subways. A bug in such systems can not only lost a lot of money but also human life. Thus, the validation of program implementation, which checks whether the code conforms to developer's intentions, is necessary.

A common method for reducing bugs in programs is testing. However, this technique cannot cover all cases of the input space, and thus, although the tests are successful, the implementation may still contain errors. Another possibility is to use formal verification methods. It mathematically proves the correctness of the specification. Although formal verification is used in critical software, there are some aspects which are not studied fully such as computations on floating-point numbers.

Before describing the contributions and the organization of this thesis, we introduce some reasons which cause the discrepancies of results in floating-point programs and give a state of the art of formal verification in general and a state of the art of formal verification for floating-point arithmetic.

1.1 Floating-point arithmetic

Nowadays, floating-point computations are widely used in critical systems from domains such as physics, aerospace system, nuclear simulation, etc. In systems which use floating-point computations, the calculations are not exact. This means that there is an approximation for each operation and thus rounding error occurs. There were some famous disasters caused by calculation bugs. The first one is Patriot Missile Failure [67] on February 25, 1991. The cause was an inaccurate calculation of the time since boot due to computer arithmetic errors. The second one is the explosion of the Ariane 5 on June 4, 1996, just forty seconds after lift-off. *"The internal SRI* [1] *software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer."*, explained the report of the Inquiry Board [52]. The third one is a famous FDIV bug in 1994 in the Intel Pentium processor that costs $500 millions [38]. The effort to reduce errors is thus necessary for the safety and reliability of programs.

As we have presented, inaccurate calculation is an issue in critical systems. The reason is that there exist inconsistencies between program executions on different architectures. The IEEE-

[1] SRI stands for Système de Référence Inertielle or Inertial Reference System.

754 is the IEEE standard for floating-point arithmetic. This standard specifies interchange, arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments. An implementation of a floating-point system conforming to this standard may be realized entirely in software, hardware or in any combination of software and hardware. This standard ensures that we get exactly the same results from the floating-point calculations of a program when it executes in any processor that follows strictly this standard. In Java, the floating-point calculations in a class or a method containing the modifier `strictfp` [2] will follow strictly the standard. In C, the compiler `gcc` for x86-64 architectures uses `-msse` and `-msse2` by default to enable SSE extensions. For example, a *double* number is represented in 64 bits; a calculation on *double* is always rounded in 64 bits. However, the computations do not always follow strictly the IEEE-754 standard because of some architecture-dependent issues. This means that for a `double` type in C for example, by changing options of the compiler, the calculation is not always done directly in 64 bits. The first architecture-dependent issue is the x87 floating-point unit (FPU) featured in processors of IA32 architecture. It uses the 80-bit internal floating-point registers on the Intel platform. The second one is the fused multiply-add (FMA) instruction, supported by the PowerPC and the Intel Itanium architectures, computes $xy \pm z$ with a single rounding. Besides, compiler issues like optimization may also cause discrepancies in results. Because of the issues above, the floating-point computations of a program running on different architectures may be different [58].

1.2 Formal verification

Formal verification is more and more studied and used in both academy and industry research in order to guarantee the safety of programs. There are many approaches for verifying programs formally. Among of them are approaches which verify the source code such as static analysis, others formally verify the machine code implementation.

Static analysis is an approach for checking a program without running it. Deductive verification techniques which perform static analysis of code, rely on the ability of theorem provers to check validity of formulas in first-order logic or even more expressive logics. They usually come with expressive specification languages such as JML [15, 46] for Java, ACSL [9] for C, Spec# [6] for C#, etc. to specify the requirements. For automatic analysis of floating-point codes, a successful approach is abstract interpretation based static analysis, that includes Astrée [22, 59] and Fluctuat [26].

We introduce here a short history about the formalization of floating-point arithmetic for the specification and the verification. Floating-point arithmetic has been firstly formalized in 1989 by G. Barrett [8]. This is only the formalization of the IEEE standard for binary floating-point arithmetic in the specification language Z. However, there are not any proofs using this specification. Then, in 1995, Carreño and Miner [17] presented the specification of the IEEE-875 Floating-point standard in HOL and PVS [63]. Two year later, Harrison [39] presented his works on the specification of floating-point in HOL Light – a tool written by himself – about the specification of floating-point arithmetic and demonstrated how to use this specification for proving. The works of Russinoff [65] are the formal verification of floating-point multiplication, division and square root instructions of the AMD-K7 microprocessor in ACL2 [47]. The formalization of IEEE-754 standard in Coq [20] was first done in 2001 by Théry, Rideau and Daumas [24]. The common point of these works is the formalization of the floating-point standard for a specification language.

[2] http://java.sun.com/docs/books/jls/third_edition/html/expressions.html#249198

There are few attempts in specifying and proving behavioral properties of floating-point programs in deductive verification system. In 2004, Miné used relational abstract domains to dectect floating-point run-time errors [57]. Leavens presented how to handle NaN problems in floating-point operations for JML in Java in 2006 [48]. Another proposal has been made in 2007 by Boldo and Filliâtre [11]. Authors introduced a methodology and proofs in Coq to perform formal verification of floating-point C programs which was implemented in a tool called Caduceus [34]. In 2010, Ayad and Marché [4] extended this to the support of special values and to the use of automatic theorem provers. This work was implemented in the Jessie plug-in of Frama-C platform. However, these works only follow strictly the IEEE-754 standard, with neither FMA, nor extended registers, nor considering optimization aspects.

Formal verification of machine-code programs is an approach for verifying a program at machine-code level. The specification and correctness of machine-code programs has been first studied in 1961 by Goldstein and von Neumann [37]. After that, program verification was applied to machine code by Maurer [54] in 1976. This paper presented the effort to put techniques for proving the correctness of assembly language and machine language programs into practice. More precisely, it is used for programs written for the Litton C400 airborne computer including overflow analysis, round-off and truncation analysis, fixed-point scaling considerations, etc.. A year later, Floyd-Hoare-style verification condition generator was applied to machine code by Clutterbuck and Carré [18]. In 1996, Boyer and Yu [14] formalized the MC68020 Instruction Set in the automated reasoning system Nqthm (also known as "the Boyer-Moore theorem prover" and predecessor of ACL2). In this paper, bit vector operations are defined with non-negative integer arithmetic and not for floating-point arithmetic. Another point is that they were able to verify few programs only specified in Nqthm. In 2006, Matthews et al. [53] published a paper on mechanized Hoare logic reasoning for machine code in the form of verification condition generator (VCG) inside the ACL2 theorem prover. Meanwhile, Leroy [49] used Coq for the correctness of an optimizing compiler which takes a significant subset of C as input and produces PowerPC assembly code as output from a simple imperative intermediate language called Cminor. In a separate work on compilation, Li and Slind et al. [51] showed that one can compile programs with proof, directly from the logic of the HOL4 theorem prover. In 2007, Pavlova [64] proposed a framework which allows the verification of Java bytecode programs. In that thesis, she defined both a Bytecode Modeling Language (BML) and a verification condition generator for Java bytecode which is independent from the source code. A year later, Myreen [61] made contributions both to approaches for verification of programs and methods for automatically constructing correct code. This work was implemented only for the HOL4 theorem prover.

Although there are a lot of former works on formal verification at low-level programs, none of them considers any aspect of floating-point computation behavioral verification at assembly level.

1.3 Contributions

As our goal is to verify floating-point programs in considering architecture and compiler aspect, this thesis consists of two approaches: first, proving floating-point properties by using static analysis for multiple architectures; second, analyzing assembly code and translating it to verification conditions proved by automatic or interactive provers.

The first approach proves numerical programs whatever the environment, or in other words, it proves numerical programs with few restrictions on the compiler and the processor. More precisely, considering rounding-to-nearest mode, double precision numbers and computations, we calculate the rounding error for each operation independent to hardware and the choice of

compiler. This approach is implemented in Frama-C platform [3] associated with Why [35] for static analysis of C code [4].

The second approach is to prove a program on its assembly code. This means that we prove a program depending on the architecture and compiler. Indeed, with the first approach, we assure that the result is correct in an interval and thus we cannot prove an exact value. Once we compile the program into assembly code, all the necessary information is known such as the precision of each operation, the order of operations, etc. By analyzing assembly code and translating its instructions to verification conditions in the intermediate verification language Why, we can prove a numerical program totally dependent to the architecture and the compiler. A modified version of GAS (GNU Assembler) is used to generate Why verification conditions (VCs) and then these VCs are proved by automatic or interactive provers. Our contributions consist of a translator which allows us to translate all annotations in C program under the forme of inline assmemly and another translator which interpret assembly code into Why. This implementation is integrated in a modified version of GAS that we've mentioned above [5].

The structure of this dissertation is as follows:

Chapter 2 presents basic knowledge about the IEEE-754 floating-point standard and some architecture-dependent issues. We also present in this chapter a few words about Why intermediate verification language; Frama-C framework and ANSI/ISO C Specification Language (ACSL).

Chapters 3, 4 and 5 present the approach for proving numerical programs for multiple architectures. Chapter 3 studies how to bound a floating-point operation and how to prove a program independent to hardware. Next, Chapter 4 consider the bound when the additions/subtractions re-organize in case of optimization. Chapter 5 then illustrates this approach by examples.

Chapters 6 to 10 will present step by step how to translate assembly code of a numerical program into Why. Chapter 6 introduces an overview of how to prove a C program by using our approach and presents essential elements of assembly language. Chapter 7 presents how to translate assembly instructions to Why with simple programs. Then this translation will be extended for floating-point programs in Chapter 8. Chapter 9 considers programs containing conditional and loop statements. Chapter 10 presents a memory model for programs which contain arrays and pointers. Chapter 11 is a preliminary attempt to support programs with operations at the bit level.

The last chapter gives the summary of contributions and discusses directions of future works.

[3] http://frama-c.com/

[4] This approach is integrated in the version 2.30 of Why, available at http://why.lri.fr/download/why-2.30.tar.gz

[5] These tools are available at http://www.lri.fr/~nguyen/contrib/

Chapter 2

Background

This chapter includes the background on the floating-point arithmetic, architecture-dependent and compiler optimization issues. We also presents the Why intermediate language, the ACSL specification language and the Frama-C framework as they will be used in the next chapters.

2.1 Floating-point arithmetic

This section consists of two subsections: firstly, we will talk about the IEEE-754 standard; secondly, we will present architecture-independent issues.

2.1.1 The IEEE-754 floating-point standard

The IEEE-754 standard [1] for floating-point arithmetic was developed to define formats and behaviors for floating-point numbers and computations.

There are five basic formats defined in this standard:

- Three binary formats, with encodings in lengths of 32, 64 and 128 bits.
- Two decimal formats, with encodings in lengths of 64 and 128 bits.

In this thesis, we take into account only 32-bit and 64-bit binary formats.

Floating-point data

The set of finite floating-point numbers representable within a particular format is determined by the following integer parameters:

- b = the radix, 2 or 10,
- p = the number of digits in the significand (precision),
- $emax$ = the maximum exponent,
- $emin$ = the minimum exponent where $emin = 1 - emax$ for all formats.

Within each format, the following floating-point data is represented:

- Signed zero or non-zero floating-point numbers of the form

$$(-1)^s \times b^e \times m \tag{2.1}$$

  where

- s is 0 or 1,
- e is any integer such that $emin \le e \le emax$,
- m is a number represented by a digit string of the form $d_0 \cdot d_1 d_2 \ldots d_{p-1}$ where d_i is an integer digit $0 \le d_i < b$ (therefore $0 \le m < b$).

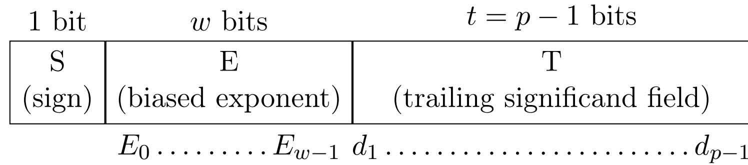
- Two infinities: $+\infty$ and $-\infty$,
- Two NaNs: qNaN (quiet) and sNaN (signaling).

There are two kinds of floating-point number: normal and subnormal. The smallest positive normal floating-point number is b^{min} and the largest one is $b^{max} \times (b - b^{1-p})$. The non-zero floating-point numbers for a format with magnitude less than b^{min} are called subnormal numbers.

Binary format encodings

In this thesis, we use only binary formats. The representation of floating-point data in the binary format is coded in k bits in the three fields ordered as follow:

- 1-bit sign S,
- w-bit biased exponent $E = e + bias$,
- $(t = p-1)$-bit trailing significand field $T = d_1 d_2 \ldots d_{p-1}$. The leading bit of the significand, d_0, is implicitly encoded in the biased exponent E.



The values of k, p, $emax$, t, w and $bias$ are listed as follows:

Parameter	**binary32**	**binary64**
k, storage width in bits	32	64
sign bit	1	1
w, exponent field width in bits	8	11
$bias$, $E - e$	127	1023
t, trailing significand field width in bits	23	52
p, precision in bits	24	53
$emax$, maximum exponent e	127	1023

The range of the encoding biased exponent E include:

- every integer in $[1, 2^w - 2]$ to encode normal numbers.
- the reserved value 0 to encode ± 0 and subnormal numbers.
- the reserved value $2^w - 1$ to encode $\pm\infty$ and NaNs.

The value v of the floating-point datum represented are:

E	T	v
$2^w - 1$	$\neq 0$	NaN
$2^w - 1$	0	$(-1)^S \times (+\infty)$
$[1, 2^w - 2]$		$(-1)^S \times 2^{E-bias} \times (1 + 2^{1-p} \times T)$ (normal numbers have an implicit leading significand bit of 1)
0	$\neq 0$	$(-1)^S \times 2^{emin} \times (0 + 2^{1-p} \times T)$ (subnormal numbers have an implicit leading significand bit of 0)
0	0	$(-1)^S \times (+0)$ (signed zero)

Rounding error

Let x is a real number, x_1 and x_2 are the two floating-point numbers closest to x such that $x_1 \leq x \leq x_2$. Let $\circ(x)$ the rounding value of x to a floating-point number. IEEE-754 standard includes five standard rounding modes:

- Round-to-$+\infty$: $\circ(x) = x_2$.
- Round-to-$-\infty$: $\circ(x) = x_1$.
- Round-to-zero:

$$\circ(x) = \begin{cases} x_1 \text{ if } |x_1| \leq |x_2|, \\ x_2 \text{ if } |x_1| > |x_2|. \end{cases}$$

- Round-to-nearest (ties to even):

$$\circ(x) = \begin{cases} x_1 \text{ if } |x_1 - x| < |x_2 - x|, \\ x_2 \text{ if } |x_1 - x| > |x_2 - x|, \\ \text{the one with the bit } p-1 \text{ of the significand being 0 if } |x_1 - x| = |x_2 - x|. \end{cases}$$

- Round-to-nearest (ties away from zero):

$$\circ(x) = \begin{cases} x_1 \text{ if } & |x_1 - x| < |x_2 - x| \text{ or} \\ & (|x_1 - x| = |x_2 - x| \text{ and } |x_1| > |x_2|), \\ x_2 \text{ if } & |x_1 - x| > |x_2 - x| \text{ or} \\ & (|x_1 - x| = |x_2 - x| \text{ and } |x_1| \leq |x_2|). \end{cases}$$

When approximating a real number x by its rounding $\circ(x)$, a rounding error usually happens. We here consider only round-to-nearest mode, that includes both the default rounding mode (ties to even) and the new round-to-nearest, ties away from zero. In radix 2 and round-to-nearest mode, a well-known bound on the error is given by D. Goldberg [36] as follows:

- If a floating-point $f = \circ(x)$ is such that $|f| \geq 2^{e_{min}}$ then we bound the rounding error by the relative error:

$$\left| \frac{x - f}{x} \right| \leq 2^{-p}$$

- For smaller f, the value of the relative error becomes large (up to 0.5). In that case, f is a subnormal number and we prefer a bound based on the absolute error:

$$|x - f| \leq 2^{e_{min} - p}$$

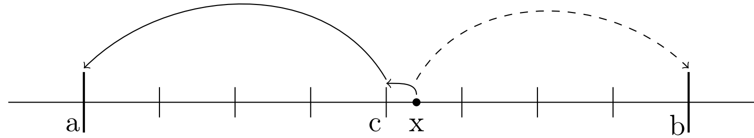


Figure 2.1: Bad case for double rounding

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64; // y = 2^{-53} + 2^{-64}
  double z = x + y;
  printf("z=%a\n",z);
}
```

Figure 2.2: A simple program giving different answers depending on the architecture.

2.1.2 Architecture-dependent issues

With the same program containing floating-point computations, the result may be different depending on the compiler and the processor. We present in this subsection some architecture-dependent issues.

x87 floating-point unit

A well-known cause of discrepancy happens in the IA32 architecture (Intel 386, 486, Pentium etc.) [58]. The IA32 processors feature a floating-point unit called "x87". This unit has 80-bit registers in "double extended" format (64-bit significand and 15-bit exponent), often associated to the *long double* C type. When using the x87 mode, the intermediate calculations are computed and stored in the x87 registers (80 bits). The final result is rounded to the destination format. Extended registers may lead to double rounding, where floating-point results are rounded twice. For instance, the operations are computed in the *long double* type of x87 floating-point registers, then rounded to IEEE double precision type for storage in memory. Double rounding may yield different result from direct rounding to the destination type.

An example is given in Figure 2.1: the long vertical lines are floating-point numbers representable in 64 bits, the short ones are floating-point numbers representable in 80 bits. We assume x is near the midpoint c of two consecutive floating-point numbers a and b in the destination format. Using round-to-nearest, with single rounding, x is rounded to b. However, with double rounding, it may firstly be rounded towards the middle c and then be rounded to a (if a is even). The two obtained results are different.

A C program illustrated for Figure 2.1 is presented in Figure 2.2: the values $x = 1.0$ and $y = 2^{-53} + 2^{-64}$ are exactly representable in double precision. With strict IEEE-754 double precision computations for `double` type, the result obtained is $z = 1 + 2^{-52}$. Otherwise, on IA32, if the computations on `double` is performed in the `long double` type inside x87 unit, then converted to double precision, $z = 1.0$.

Another example which gives inconsistencies between x87 and SSE is presented in Figure 2.3. It is is part of the KB3D [27], an aircraft conflict detection and resolution program. In this example, we have a function `int sign(double x)` which returns a value which is either -1 if

```c
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx, double sy, double vx, double vy){
  return sign(sx*vx + sy*vy) * sign(sx*vy - sy*vx);
}
int main(){
  double sx = -0x1.0000000000001p0;   // sx = -1 - 2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy =  0x1.fffffffffffffp-1; // vy = 1 - 2^-53
  int r = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",r);
}
```

Figure 2.3: A more complex program giving different answers depending on the architecture.

$x < 0$, or 1 if $x \geq 0$. The function `int eps_line(double sx, double sy, double vx, double vy)` then makes a direction decision depending on a sign after few floating-point computations. We execute this program on SSE unit and obtain that `Result = 1`. When it is performed on IA32 inside x87 unit, the result is `Result = -1`.

FMA

Another cause of discrepancy in the result of a floating-point program is the fact that some processors (IBM PowerPC or Intel/HP Itanium) have a *fused multiply-add* (FMA) instruction which computes $x \times y \pm z$ as if with unbounded range and precision, and rounds only once to the destination format. This operation can speed up and improve the accuracy of dot product, matrix multiplication and polynomial evaluation. But how should $a \times b + c \times d$ be computed? When a FMA is available, the compiler may choose either $\circ(a \times b + \circ(c \times d))$, or $\circ(\circ(a \times b) + c \times d)$, or $\circ(\circ(a \times b) + \circ(c \times d))$ which may give different results.

Optimization issues

The last cause for discrepancies occurs when compilers optimize floating-point computations. This includes re-organizing additions or multiplications, use of distributivity, etc. Those mathematically correct identities are usually refuted by floating-point operations.

When we compile a C program with optimization options[1], the assembly code may be changed. Here are some options of `gcc`:

- `-O0`: no optimization. This is the default.
- `-O`, `-O1`: the compiler tries to reduce code size and execution time, without performing any optimizations.
- `-O2`: Optimize even more. As compared to `-O`, this option increases both compilation time and the performance of the generated code. `-O2` turns on `-finline-small-functions` that

[1] http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

```
double f(double x){
  return x/1e308;
}

double square(double x){
  double y = x*x;
  return y;
}

int main(){
  printf("%g\n",f(square(1E308)));
  return 0;
}
```

Figure 2.4: A program giving different answers depending on the optimization.

---

allows to integrate functions into their callers when their body is smaller than expected function call code.

- `-O3`: Optimize yet more. It turns on all optimizations specified by `-O2` and also other options including `-finline-functions`.
- `-funsafe-math-optimizations`: Allow optimizations for floating-point arithmetic, e.g. use associative math, use reciprocal instead of division, disregard floating-point exceptions (division by 0, overflow, underflow, etc).

The options above may change the assembly code and thus affect the result of floating-point calculations except the first and the second one.

Inline function means that the function is integrated into their caller. On x87, inlining may change the result. The program in Figure 2.4 which is presented by D. Monniaux [58] illustrates this case. When compiling it with `-mfpmath=387 -O2`, function inlining occurs, the two functions `f` and `square` are integrated in the function `main` and intermediate values are stored in 80-bit registers, not in 64-bit registers. This means that the result is $1E308$ and not overflow although it is overflow (the result is Inf) with default option of `gcc`.

2.2 Why- An Intermediate Verification Language

Why[2] is a software verification platform [35]. It is also known as an intermediate language for program verification. In this dissertation, we use Why as a tool to express our results in the logic world.

Figure 2.5 shows us the process of how Why receives Java and C programs as inputs and generates verification conditions which are proved by automatic or interactive provers. Why has front-ends for C and Java in which the front-end for C is in fact integrated in the Frama-C environment for static analysis of C programs.

One advantage of Why is that it allows us to declare logical models (types, functions, predicates, axioms, lemmas) that can be used in programs and annotations. Another advantage is that the Why tool supports a set of existing provers:

---

[2] http://why.lri.fr/

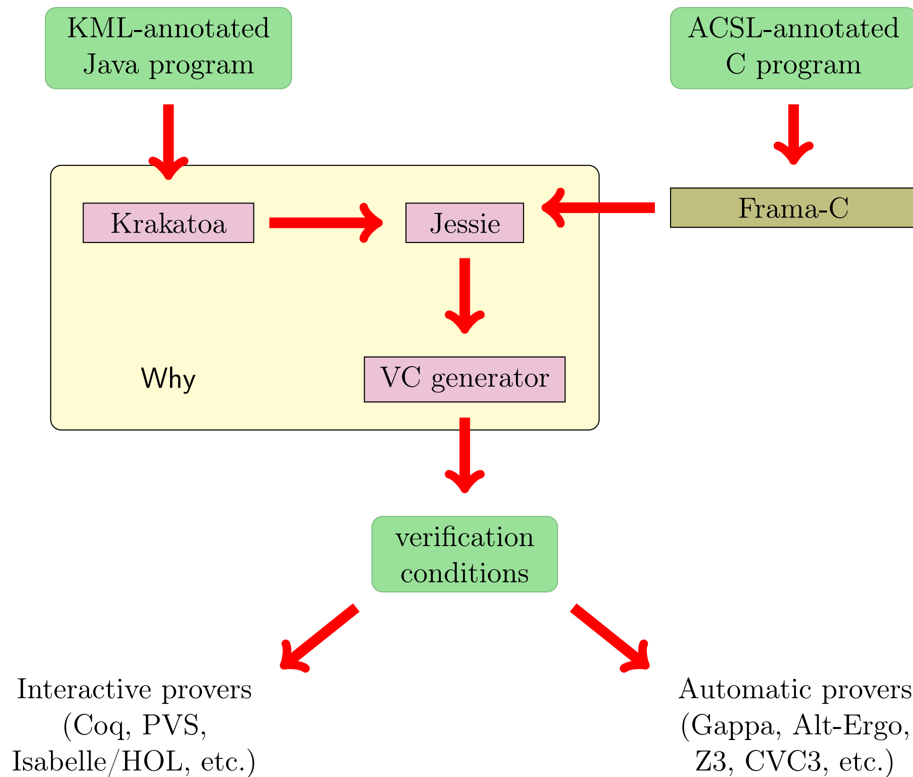


Figure 2.5: Why platform

- Proof assistants: Coq, PVS, etc.

- Automatic theorem provers: Alt-Ergo [19], CVC3 [7], Gappa [25, 55], etc.

In the input language of Why, we can define a pure model in the logic world by declaring abstract sort names, declaring logic symbols operating on these sorts and posing first-order axioms to axiomatize the behavior of these symbols. Equality and both integer and real arithmetic are built-in in the logic. We can then declare a set of references which are mutable variables denoting logic values. Finally, we can define procedures which can modify these references. The body of such a procedure is made of statements in a while-style language.

Why includes also logical annotations (pre- and post-conditions, loop invariants and variants, intermediate assertions). Procedures are equipped with pre- and post-conditions. The Why VC generator then produces the necessary VCs to ensures that the body respects the post-condition. One can alternatively just declare procedures by only giving pre- and post- conditions, but also declaring the set of modified references. This feature allows to declare how the atomic operations on a given data type behave.

The detail of the syntax of Why is found at: http://why.lri.fr/. In this section, we will present in a few words axiomatizations, logic functions and parameters which we will use in the thesis.

2.2.1 Logical declarations in Why

Axiomatization

The axiomatization includes abstract types, declarations of functions and predicates, axioms. For example:

```
predicate is_int32(x:int) = -2147483648 <= x and x <= 2147483647
type int32
logic integer_of_int32: int32 -> int
axiom int32_coerce: forall x:int32. is_int32(integer_of_int32(x))
```

Predicate gives a definition based on propositions. For example:

```
predicate is_int32(x:int) = -2147483648 <= x and x <= 2147483647
```

This predicate says that its argument ranges between `-2147483648` and `2147483647`.

Type is a declaration of an abstract type. For example:

```
type int32
```

The type `int32` is an abstract type which represents a 32-bit integer.

Logic function is a function symbol which declares type of inputs and one of output. The body of the function is not defined. Normally, we need to declare axioms to specify logic functions. For example:

```
logic integer_of_int32: int32 -> int
```

The function logic `integer_of_int32` returns an integer value from an abstract type `int32`.

Axiom is a logical statement that is assumed to be true.

```
axiom int32_coerce: forall x:int32, is_int32(integer_of_int32(x))
```

This axiom says that all value `x` having type `int32` always denotes an integer value in the range of 32-bit word.

2.2.2 Programs in Why

A program in Why can be defined as a function with a body, or declared as a parameter.

Reference declaration with `parameter`

Parameter can be used to declare a reference (mutable value) in Why. For example:

```
parameter value: int ref
```

Function with `let`

A function in Why may contain annotations (preconditions, post-conditions and assertions) and statements.

```
let f (n:int) =
   {  }
   r := min !r n
   { r <= r@ }
```

The function `f` takes an integer `n` as input and assigns the value of `min !r n` to `r`. It has no precondition and a post-condition which specifies that the final value of `r` is not greater than its initial value (denotes by `r@`).

Function declaration with `parameter`

Parameter can be used for the declaration of an "abstract program" with precondition and post-condition. For example:

```
parameter add_int: a:int -> b:int ref ->
   {  }
   unit writes b
   { b = b@ + a }
```

The parameter `add_int` takes an integer `a` and a variable `b` having type `int` with no precondition, returns `unit` (nothing). The post-condition is that the new value of `b` is the sum of `a` and the previous value of `b`. Notice that in parameter, there is a declaration of side-effects: the `reads` and `writes` keywords declare the set of references possibly accessed and modified, respectively.

2.2.3 Example

In order to know how to write a Why program, a small program is presented as follows:

```
parameter x: int32 ref

parameter inc: n:int32 ref ->
 { is_int32(integer_of_int32(n) + 1) }
   unit writes n
 { integer_of_int32(n) = integer_of_int32(n@) + 1 }

let main() =
 { is_int32(integer_of_int32(x) + 2)}
   inc x;
   inc x;
   void
 { integer_of_int32(x) = integer_of_int32(x@) + 2 }
```

All the logical declarations of this program are explained in subsection 2.2.1. We declare a parameter `x` having type `int32`, another parameter `inc` which returns the value of input after adding 1.

In the function `main`, we assign to `x` the value of `inc !x`. This function is used twice. The post-condition of the function `main` assures that the new value of `x` is equal to the sum of old

Proof obligations	Alt-Ergo 0.93	CVC3 2.4.1 (SS)	Statistics
function main Correctness	✓	✓	3/3
1. precondition	✓	✓	
2. precondition	✓	✓	
3. postcondition	✓	✓	

```
x: int32
H1: is_int32(integer_of_int32(x) + 2)
H2: is_int32(integer_of_int32(x) + 1)
x0: int32
H3: integer_of_int32(x0) = integer_of_int32(x) + 1
H4: is_int32(integer_of_int32(x0) + 1)
x1: int32
H5: integer_of_int32(x1) = integer_of_int32(x0) + 1
_______________
integer_of_int32(x1) = integer_of_int32(x) + 2
```

```
 let main() =
  { is_int32(integer_of_int32(x)+2) }
   inc x;
   inc x;

   void
  { integer_of_int32(x) = integer_of_int32(x@) + 2 }
```

Figure 2.6: Result of a Why program

one and 2. A screen-shot in Figure 2.6 illustrates how to prove a Why program using automatic provers.

There are three obligations generated. The two first ones check the precondition of the parameter `inc`. As the parameter `inc` has a precondition, each time it is invoked, the precondition must be proved. In the function `main`, `inc` is called twice, this is the reason why we have two obligations that check the precondition of `inc`. The last one checks the post-condition of the function `main`.

We present below the VCs for these obligations and show how to prove it manually.

VCs for the first obligation

The VCs for proving the precondition of the first `inc x` in the function `main` are:

```
x: int32
H1: is_int32(integer_of_int32(x) + 2)
---------------------------------------------------
is_int32(integer_of_int32(x) + 1)
```

The hypothesis of this obligation is the precondition of `main`. As in H1 we have $int(x)+2$ which is in range of 32-bit integer and as $int(x)$ is also in the range thanks to axiom `int32_coerce`, easily we prove that $int(x)+1$ is also in this range.

VCs for the second obligation

The VCs for proving the precondition of the second `inc x` in the function `main` are:

```
x: int32
H1: is_int32(integer_of_int32(x) + 2)
H2: is_int32(integer_of_int32(x) + 1)
x0: int32
H3: integer_of_int32(x0) = integer_of_int32(x) + 1
---------------------------------------------------
is_int32(integer_of_int32(x0) + 1)
```

They are explained by:

$$
\begin{aligned}
&H1 : int(x) + 2 \ is \ in \ range \ of \ 32 - bit \ integer.\\
&H2 : int(x) + 1 \ is \ in \ range \ of \ 32 - bit \ integer.\\
&H3 : int(x0) = int(x) + 1
\end{aligned}
$$

$$Goal : int(x0) + 1 \ is \ in \ range \ of \ 32 - bit \ integer.$$

We need to prove that: $int(x0)+1$ is in range of 32-bit integer. This means that $(int(x)+1)+1$ or $int(x) + 2$ (from H2, replace $int(x0)$ with $int(x) + 1$) must be in range of 32-bit integer. As $int(x)$ is in the range and from H1, this obligation is proved.

VCs for the post-condition obligation

The VCs generated for post-condition of the function `main` are shown below:

```
x: int32
H1: is_int32(integer_of_int32(x) + 2)
H2: is_int32(integer_of_int32(x) + 1)
x0: int32
H3: integer_of_int32(x0) = integer_of_int32(x) + 1
H4: is_int32(integer_of_int32(x0) + 1)
x1: int32
H5: integer_of_int32(x1) = integer_of_int32(x0) + 1
---------------------------------------------------
integer_of_int32(x1) = integer_of_int32(x) + 2
```

This means that

$$
\begin{aligned}
&H1 : int(x) + 2 \ is \ in \ range \ of \ 32 - bit \ integer.\\
&H2 : int(x) + 1 \ is \ in \ range \ of \ 32 - bit \ integer.\\
&H3 : int(x0) = int(x) + 1\\
&H4 : int(x0) + 1 \ is \ in \ range \ of \ 32 - bit \ integer.\\
&H5 : int(x1) = int(x0) + 1
\end{aligned}
$$

$$Goal : int(x1) = int(x) + 2$$

We need to prove that: $int(x1) = int(x)+2$. From H5, we replace $int(x1)$ with $int(x0)+1$, then from H3, we replace $int(x0)$ with $int(x)+1$. Now what we have is: $((int(x)+1)+1) = int(x)+2$ and the obligation is proved.

2.2.4 Use of Why in this thesis

We all know about the syntax of Why. The question now is how it works in this thesis. In the whole thesis, we use the platform Why to generate verification conditions which are then proved by automatic or interactive provers. The use of Why is illustrated in Figure 2.7. In Part 1, we use Frama-C/Jessie which is a front-end of the Why platform for deductive program verification for proving a numerical program. Here, we define a model written in Why called `multirounding`. The Jessie plug-in uses this model to interpret each operation in a C program. In Part 2, we use

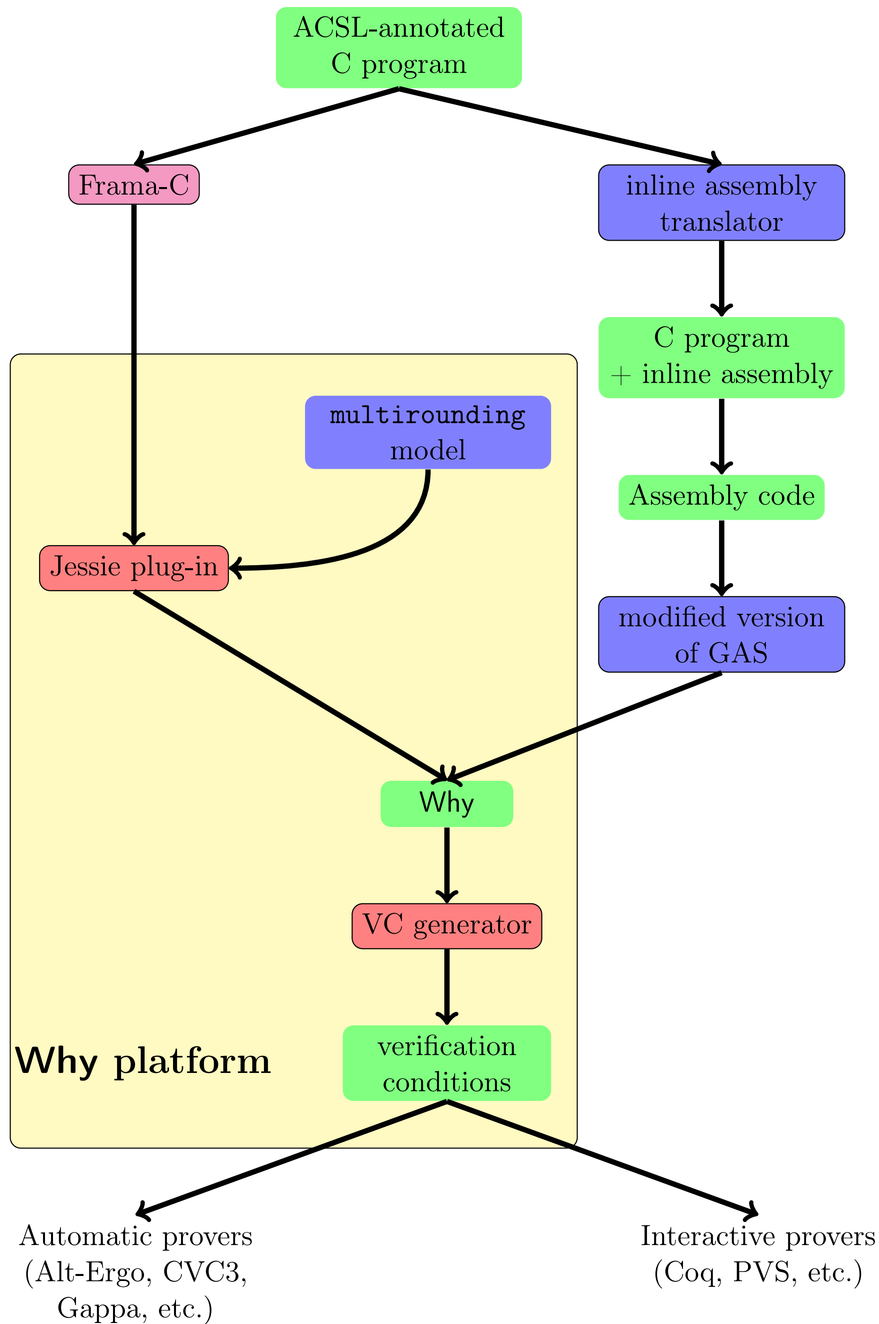


Figure 2.7: Use of Why

Why as the output of our translator. This means that we will translate the assembly code into programs in Why. These Why programs are then the inputs of VC generator to generate VCs. In Chapter 11, Why 3 – the next generation of Why (also called Why 2) – is used. We consider in this chapter that Why 3 is a specification language and we write the program directly in Why 3. The VCs are generated from this Why 3 program by Why 3 platform.

```
/*@ requires \abs(x) <= 0x1p-5;
  @ ensures \abs(\result - \cos(x)) <= 0x1p-23;
  @*/
float my_cos(float x) {
  //@ assert \abs(1.0 - x*x*0.5 - \cos(x)) <= 0x1p-24;
  return 1.0f - x * x * 0.5f;
}
```

Figure 2.8: A C program annotated in ACSL

2.3 ACSL and Frama-C

2.3.1 ACSL

The ANSI/ISO C Specification Language (ACSL) [3] is a behavioral specification language for C programs. It is inspired by Caduceus [34, 33], itself inspired by JML [15, 46]. Its goal is to allow us to formally verify that the implementation of a C function respects its specifications.

The most important ACSL concept is the function contract. A function contract for a C function `f` is a set of requirements over the arguments of `f` and/or a set of properties that are ensured at the end of the function.

ACSL annotations are written in special C comments:

- `/*@ */` for one or multiple lines,
- `//@` for a single-line annotation.

There are three important annotations in ACSL:

Precondition Precondition begins by the keyword `requires`. It contains the conditions which must hold before calling a function.

Post-condition Post-condition begins with the keyword `ensures`. It is the conditions which must hold after calling a function.

Assertion Properties which must hold at a program point.

The example in Figure 2.8 contains a C function `float my_cos(float x)` with ACSL annotations. It provides an approximation of the cosine function on a small interval around 0. The precondition assumes that the absolute value of `x` is not greater than 2^{-5} $(= \frac{1}{32})$. The post-condition says that the absolute value of total error (which includes method error and rounding error) is not greater than 2^{-23}. The method error is stated in an assertion which says that the bound of method error is 2^{-24}.

Notice that all the operations in the annotations are done in real, they are not floating-point operations.

2.3.2 Frama-C

Frama-C is a framework for static analysis of C code. This framework is flexible as it is easy to add a new plug-in. There are some ready-to-use plug-ins:

[3] http://frama-c.com/acsl.html

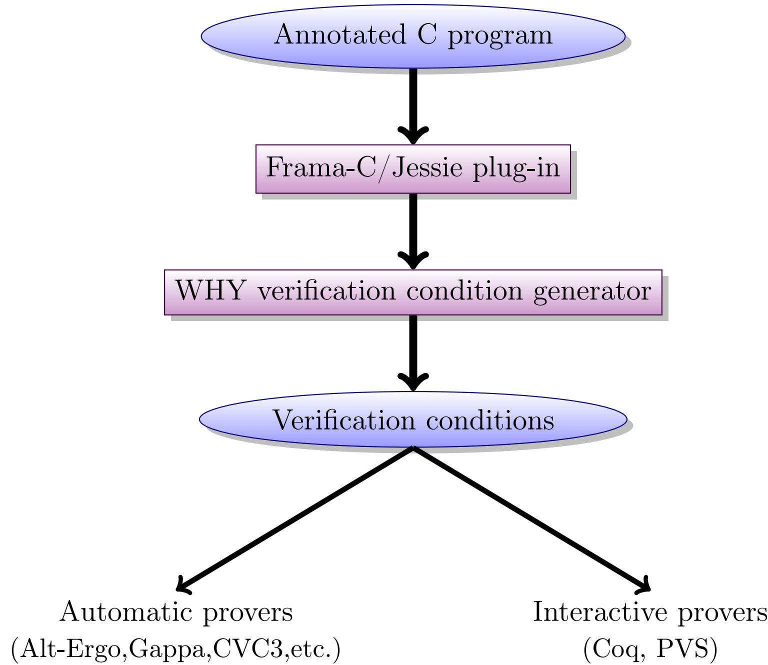


Figure 2.9: Proof of numerical programs in Frama-C/Jessie

Value analysis This plug-in uses abstract interpretation techniques. It computes variation domains for variables.

Jessie This is deductive verification plug-in. It is based on weakest precondition computation techniques. It aims at deductive verification of behavioral properties of the code, specified using the ACSL language.

In Jessie, there are some ready-to-use models for floating-point arithmetic called strict model and full model. The strict model includes rounding modes; single and double types; basic operations, etc. However, this model forbids exceptional values. The full model is an extension of the strict one. It allows to verify the programs in considering special values such as infinities, NaN, etc..

Figure 2.9 shows us how to prove a numerical program with Frama-C/Jessie. The input of this plug-in is an annotated C program. Its annotations are specified in ACSL. The output of Frama-C/Jessie plug-in is then the input of Why verification condition generator. This generator creates VCs which can be proved by automatic provers (Alt-Ergo, Gappa, CVC3, etc.) or interactive provers (Coq, PVS). The Gappa tool is an automatic prover, which handles formulas made of equalities and inequalities over expressions involving real constants and arithmetic operations. While Gappa does not handle quantifiers, SMT solvers which are a decision procedure that can handle quantifiers, various types of arithmetic and other decidable theories. This is the reason why in our examples, we often combine several automatic provers to prove a program.

Now let's back to example in Figure 2.8. All the needed properties about floating-point arithmetic are specified in ACSL. The analysis of this program is done by using Frama-C/Jessie with the strict model which follows strictly the IEEE-754 standard. The verification conditions generated are proved by Gappa except the assertion which is proved using Coq and interval tactic [56].

Part I

Hardware-independent proofs of numerical programs

Chapter 3

Hardware-independent bounds for floating-point operations

This chapter presents how to bound the rounding error of floating-point operations whatever the architecture and the compiler.

3.1 Bounds for one floating-point operation

As we want both correct and interesting properties on a floating-point computation without knowing which rounding will be in fact executed, the chosen approach is to consider only the rounding error. This will be insufficient in some cases, but we believe this can give useful and sufficient results in most cases.

As explained, the choice between 64-bit, 80-bit and double rounding is the main reason that causes the discrepancies of result. We prove a rounding error bound that is valid whatever the hardware, and the chosen rounding (corresponds to interval arithmetic [41, 60] and standard model [42]).

We denote by $\circ_{64}$ the round-to-nearest in the `double` 64-bit type, by $\circ_{80}$ the round-to-nearest to the extended 80-bit registers.

Theorem 3.1 *For a real number x, let $\Box(x)$ be either $\circ_{64}(x)$, or $\circ_{80}(x)$, or the double rounding $\circ_{64}(\circ_{80}(x))$. We have either*

$$\left(|x| \geq 2^{-1022} \text{ and } \left|\frac{x - \Box(x)}{x}\right| \leq 2050 \times 2^{-64}\right)$$

or

$$\left(|x| \leq 2^{-1022} \text{ and } |x - \Box(x)| \leq 2049 \times 2^{-1086}\right).$$

The proof is done by case analysis in the following subsections. This theorem is the basis of our approach to correctly prove numerical programs whatever the hardware. These bounds are tight as they are reached in all cases where $\Box$ is the double rounding. They are a little bigger than the ones for 64-bit rounding (2050 and 2049 instead of 2048) for both cases. These bounds are therefore both correct, very tight, and just above the 64-bit's.

As 2^{-1022} is a floating-point number, we have $\Box(2^{-1022}) = 2^{-1022}$. As all rounding functions are monotone[1], $\Box(x)$ is also monotone. Then $|x| \geq 2^{-1022}$ implies $|\Box(x)| \geq 2^{-1022}$ and vice versa.

Now let us prove the bounds of Theorem 3.1 on the rounding error for all possible values of $\Box$.

[1] A monotonic function f is a function such that, for all x and y, $x \leq y$ implies $f(x) \leq f(y)$.

3.1.1 Case 1: Rounding error in 64-bit rounding

From the formulas given by D. Goldberg [36] (explained in subsection 2.1.1), with 64-bit rounding, we have $p = 53$ and $e_{min} = -1022$. Therefore:

- With $|\circ_{64}(x)| \geq 2^{-1022}$: $\left|\frac{x - \circ_{64}(x)}{x}\right| \leq 2^{-53}$
- With $|\circ_{64}(x)| \leq 2^{-1022}$: $|x - \circ_{64}(x)| \leq 2^{-1075}$

Let us see Figure 3.1, the rounding error bounds of 64-bit rounding are above the line and the ones of Theorem 3.1 (T3.1) are below the line. The rounding error bounds of D. Goldberg are smaller than the desired ones, therefore Theorem 3.1 holds in 64-bit rounding.

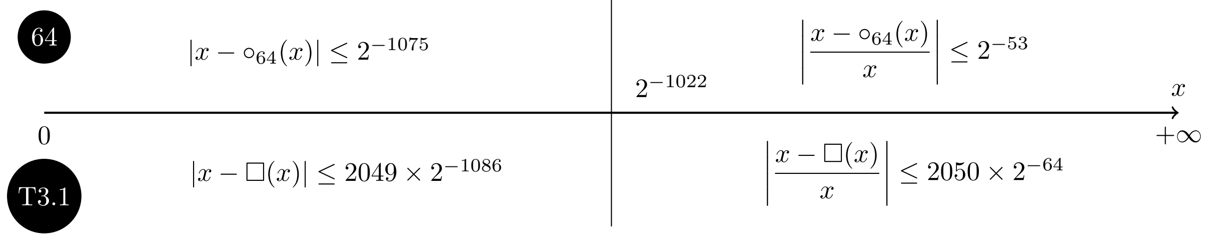


Figure 3.1: Rounding error in 64-bit rounding vs. Theorem 3.1

3.1.2 Case 2: Rounding error in 80-bit rounding

The 80-bit registers used in x87 have a 64-bit significand and a 15-bit exponent. Thus, $p = 64$ and $e_{min} = -16382$. By applying the formulas of D. Goldberg:

- With $|\circ_{80}(x)| \geq 2^{-16382}$: $\left|\frac{x - \circ_{80}(x)}{x}\right| \leq 2^{-64}$
- With $|\circ_{80}(x)| \leq 2^{-16382}$: $|x - \circ_{80}(x)| \leq 2^{-16446}$

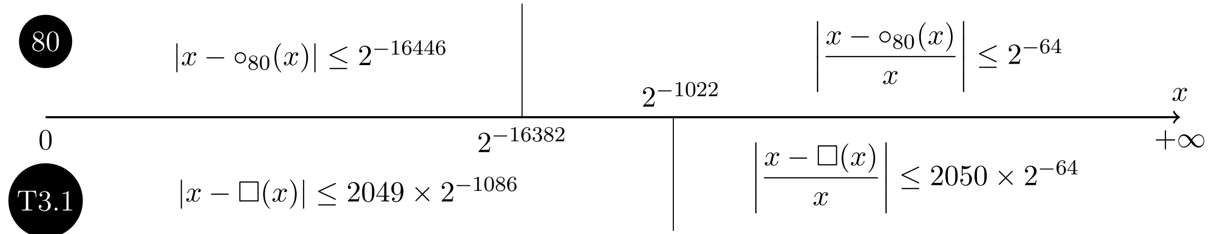


Figure 3.2: Rounding error in 80-bit rounding vs. Theorem 3.1

In Figure 3.2, the rounding error bounds in 80-bit rounding are above the line and the ones of Theorem 3.1 are below the line. Intuitively, the rounding error bounds in case of 80-bit rounding are much smaller in this case than Theorem 3.1's except in the case where $|x|$ is between 2^{-16382} and 2^{-1022}. There, we have

$$\begin{aligned} |x - \circ_{80}(x)| &\leq 2^{-64} \times |x| \\ &\leq 2^{-64} \times 2^{-1022} = 2^{-1086}. \end{aligned}$$

All bounds are smaller than that of Theorem 3.1, so Theorem 3.1 holds in 80-bit rounding.

3.1.3 Case 3: Rounding error in double rounding

The bounds here will be the one of Theorem 3.1. We split in two cases depending on the value of $|x|$.

Normal range We first assume that $|x| \geq 2^{-1022}$. We bound the relative error by some computations and the previous formulas:

$$\begin{aligned}
\left|\frac{x - \circ_{64}(\circ_{80}(x))}{x}\right| &\leq \left|\frac{x - \circ_{80}(x)}{x}\right| && + \left|\frac{\circ_{80}(x) - \circ_{64}(\circ_{80}(x))}{x}\right| \\
&\leq 2^{-64} && + \left|\frac{\circ_{80}(x) - \circ_{64}(\circ_{80}(x))}{\circ_{80}(x)} \times \frac{\circ_{80}(x)}{x}\right| \\
&\leq 2^{-64} && + \left|\frac{\circ_{80}(x) - \circ_{64}(\circ_{80}(x))}{\circ_{80}(x)}\right| \times \left(\left|\frac{\circ_{80}(x) - x}{x}\right| + 1\right) \\
&\leq 2^{-64} && + 2^{-53} \times \left(\left|\frac{\circ_{80}(x) - x}{x}\right| + 1\right) \\
&\leq 2^{-64} && + 2^{-53} \times (2^{-64} + 1) = 2050 \times 2^{-64}
\end{aligned}$$

Subnormal range We now assume that $|x| \leq 2^{-1022}$. The absolute error to bound is $|x - \circ_{64}(\circ_{80}(x))|$. We have two cases depending on whether x is in the 80-bit normal or subnormal range.

If x is in the 80-bit subnormal range, then $|x| < 2^{-16382}$ and

$$\begin{aligned}
|x - \circ_{64}(\circ_{80}(x))| &\leq |x - \circ_{80}(x)| + |\circ_{80}(x) - \circ_{64}(\circ_{80}(x))| \\
&\leq 2^{-16446} + 2^{-1075} < 2^{-1086} + 2^{-1075} = 2049 \times 2^{-1086}.
\end{aligned}$$

If x is in the 80-bit normal range, then $2^{-16382} \leq |x| < 2^{-1022}$ and

$$\begin{aligned}
|x - \circ_{64}(\circ_{80}(x))| &\leq |x - \circ_{80}(x)| + |\circ_{80}(x) - \circ_{64}(\circ_{80}(x))| \\
&\leq 2^{-64} \times |x| + 2^{-1075} \\
&\leq 2^{-64} \times 2^{-1022} + 2^{-1075} = 2^{-1086} + 2^{-1075} = 2049 \times 2^{-1086}.
\end{aligned}$$

Then Theorem 3.1 holds in double rounding.

In conclusion, Theorem 3.1 is proved for all 3 roundings.

In practice, instead of dividing into two cases: normal range and subnormal range, we can use the following formula:

$$|x - \square(x)| \leq \varepsilon \times |x| + \eta \tag{3.1}$$

with $\varepsilon = 2050 \times 2^{-64}$ and $\eta = 2049 \times 2^{-1086}$.

In strict IEEE-754, where inputs and outputs are on 64 bits, we can set $\eta = 0$ for addition and subtraction. Unfortunately here, inputs may be 80-bit numbers so η cannot be set to 0. Note also that absolute value and negation may produce a rounding if we put a 80-bit number into a 64-bit number.

3.1.4 Proof in Coq

We used the Gappa Coq library [2] (version 0.13) for Coq (version 8.2) [12] to prove the correctness of Theorem 3.1.

[2] The Gappa Coq Library adds a Gappa tactic to the Coq Proof Assistant. This tactic invokes the Gappa tool to solve properties about floating-point or fixed-point arithmetic. It can also solve simple inequalities over real numbers.

```
Theorem post_conditions_correctness:
 forall x f, Rabs x <= powerRZ 2 (35000) ->
  ( f = gappa_rounding (rounding_float roundNE 53 (1074)) x
    \/ f = gappa_rounding (rounding_float roundNE 64 (16445)) x
    \/ f = gappa_rounding (rounding_float roundNE 53 (1074))
            (gappa_rounding (rounding_float roundNE 64 (16445)) x)
    \/ f = x)  ->
  (powerRZ 2 (-1022) <= Rabs x ->
        (Rabs ((f-x)/x) <= 2050 * powerRZ 2 (-64)
         /\ powerRZ 2 (-1022) <= Rabs f))
   /\
  (Rabs x  <= powerRZ 2 (-1022) ->
        (Rabs (f-x) <= 2049 * powerRZ 2 (-1086)
         /\ Rabs f <= powerRZ 2 (-1022))).
```

Figure 3.3: Coq theorem certifying the correctness of Theorem 3.1

The corresponding theorem (see Figure 3.3) and proof (228 lines) are in Coq [3].

As we use the Gappa tactic in Coq to prove the theorem, a bound on x is necessary. This is the reason why we add the requirement that $|x| \leq 2^{35000}$. This value is large enough to satisfy all operations (addition, subtraction, multiplication, division, square root, negation and absolute value) in all types (64 or 80-bit rounding).

The Coq proof exactly corresponds to the one described. It is not very difficult, but needs many computations and a very large number of subcases. The formal proof gives a very strong guarantee on this result, allowing its use without doubt in the Frama-C platform.

3.2 Proofs of numerical programs

In the previous section, we gave a bound for one operation. If we have a program containing a sequence of operations, how can we prove it? In this section, we will use Theorem 3.1 to prove such a program.

3.2.1 When FMA occurs

Theorem 3.1 gives rounding error formulas for various roundings denoted by $\square$ (64-bit, 80-bit and double rounding). Now, let us consider the FMA that computes $x \times y + z$ with one single rounding. The question is whether a FMA was used in a computation. We therefore need an error bound that covers all the possible cases.

The idea is very simple: we consider a FMA as a *rounded* multiplication followed by a rounded addition. And we only have to consider another possible "rounding" that is the identity: $\square(x) = x$.

This specific "rounding" magically solves the FMA problem: the result of a FMA is $\square(x \times y + z)$, that may be considered as $\square_1(\square_2(x \times y) + z)$ with $\square_2$ being the identity. So we handle in the same way all operations even in presence of FMA or not, by considering one rounding for each basic operation (addition, multiplication, etc.). Of course, this "rounding" easily verifies the formulas of Theorem 3.1.

[3] The proof is available at http://www.lri.fr/~nguyen/research/rnd_64_80_post.html

What is the use of this odd rounding? The idea is that each basic operation will be considered as rounded with a $\square$ that may be one of the four possible roundings. Let us go back to the computation of `a*b+c*d`: it becomes $\square(\square(a \times b) + \square(c \times d))$ with each $\square$ being one of the 4 roundings. It gives us 64 possibilities. In fact, only 45 possibilities are allowed (for example, the addition cannot be exact). But *all* the real possibilities are *included* in all the considered possibilities. And all considered possibilities have a rounding error bounded by Theorem 3.1.

So, by considering the identity as a rounding like the others, we handle all the possible uses of the FMA in the same way as we handle multiple roundings.

3.2.2 Bounds of a sequence of operations

Theorem 3.2 *Let $\odot$ be an operation among addition, subtraction, multiplication, division, square root, negation and absolute value. Let $x = \odot(y, z)$ be the exact result of this operation (without rounding). Then, whatever the architecture and the compiler, the computed result $\tilde{x}$ is such that*

If $|x| \geq 2^{-1022}$, then
$$\tilde{x} \in \left[x - 2050 \times 2^{-64} \times |x| \ ,\ x + 2050 \times 2^{-64} \times |x|\right] \ \setminus \ \left]-2^{-1022} \ ,\ 2^{-1022}\right[.$$
If $|x| \leq 2^{-1022}$, then
$$\tilde{x} \in \left[x - 2049 \times 2^{-1086} \ ,\ x + 2049 \times 2^{-1086}\right] \cap \left[-2^{-1022} \ ,\ 2^{-1022}\right].$$

This is straightforward as the formulas of Theorem 3.1 subsume all possible roundings (64 or 80-bit) and operations (using FMA or not), whatever the architecture and the compiler choices.

Theorem 3.3 *If we define each result of an operation by the formulas of Theorem 3.2, and if we are able to deduce from these intervals an interval $\mathcal{I}$ for the final result, then the really computed final result is in $\mathcal{I}$ whatever the architecture and the compiler that preserves the order of operations.*

This is proved by using Theorem 3.2.

3.3 Implementation

What we have until now is the bound of the rounding error for a sequence of operations. The question is how to use Theorem 3.1 to prove a program in Frama-C/Jessie?

An interesting point of the Jessie plug-in is that we can change the interpretation of floating-point operations easily. We define a new "pragma" called `multirounding` in the same way as the strict model of Jessie plug-in and change the interpretation of each basic floating-point operation (addition, subtraction, multiplication, division, absolute value, square root and negation). In order to do this, we put as post-conditions the formulas of Theorem 3.1 for operations in the Frama-C platform to look into the rounding error of the whole program. Ordinarily, the pragma directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. In our pragma as in Jessie, each floating-point number is represented by two values, an exact one (a real value, as if no rounding occurred) and a rounded one (the true floating-point value). At each computation, we are only able to bound the difference between these two values, without knowing the true rounded value.

For example, the post-condition of the addition is defined by the following predicate:

```
predicate add_double_post(m:mode,x:double,y:double,result:double) =
(*CASE 1:*)
((abs_real(double_value(x) + double_value(y)) >= 0x1p-1022
  and
  abs_real(double_value(result) - (double_value(x) + double_value(y)))
                  <= 0x1.004p-53 * abs_real(double_value(x) + double_value(y))
 )
 or
(*CASE 2:*)
 (abs_real(double_value(x) + double_value(y)) <= 0x1p-1022
  and
  abs_real(double_value(result)-double_value(x)+double_value(y))<=0x1.002p-1075
  )
)
and
double_exact(result) = double_exact(x) + double_exact(y)
```

`double` is an abstract type, `double_value` and `double_exact` are logic functions which return a rounded and an exact value from a `double`. This abstract type and the two logic functions have already defined in the strict and full model and we reuse them in our approach. In this predicate, we also define the exact result which is the addition of two exact values. Notice that the value `0x1.004p-53` $= 2050 \times 2^{-64}$ and `0x1.002p-1075` $= 2049 \times 2^{-1086}$.

The post-conditions for other operations (subtraction, multiplication division, square root, negation, absolute value) are defined in the same way as the addition.

Each operation is declared by a parameter and the post-condition will use the predicate above. The parameter `add_double` for addition in double, for example, is declared as follows:

```
parameter add_double :
  m:mode -> x:double -> y:double ->
  { no_overflow_double(m,double_value(x) + double_value(y)) }
  double
  { add_double_post(m,x,y,result) }
```

This parameter returns a `double` value. The precondition assures that there is no overflow in the addition. The post-condition is the predicate `add_double_post`.

Notice that the rounding modes that we handle in this approach are rounding-to-nearest modes (ties to even and ties away from zero). Thus, in this Why parameter, either `m = nearest_even` or `m = nearest_away` are allowed.

Chapter 4

When the compiler re-organizes a computation

In the previous chapter, we bound the rounding error of each operation provided that the order of the operations is preserved. We also showed that in case of FMA, our method is still valid. Our problem here is that in case of optimizations, the re-organization of operations in a computation may happen. How do we deal with it? We will explain in this chapter how to bound the error of a computation when a reorganization of additions/subtractions occurs.

4.1 Associativity for the addition

For the sake of simplicity we will denote floating-point addition by $\oplus$ and floating-point subtraction by $\ominus$, when the precision is unknown (it may be 64- or 80-bit or double rounding).

Of course, floating-point addition is not associative even if compilers may re-associate additions. For example, if $|e| \ll |x|$, then $(e \oplus x) \ominus x$ gives zero while $e \oplus (x \ominus x)$ gives e. This catastrophic cancellation is the main problem for the reorganization of additions. The idea here is that we will change the rounding error formula for the addition in order to guarantee that, even if $(a+b)+c$ is transformed into $a+(b+c)$ by the compiler, the bound on the rounding error will still hold. For that, instead of using the formula

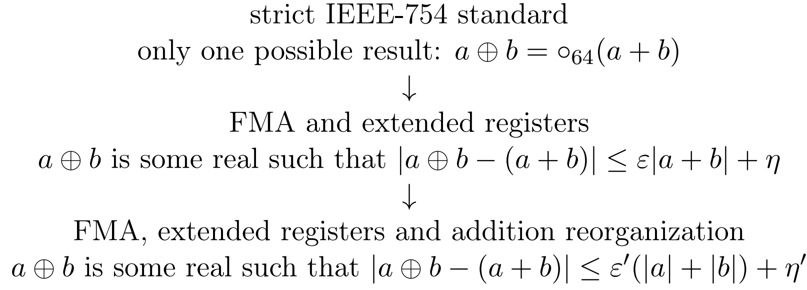
$$|a \oplus b - (a+b)| \leq \varepsilon \times |a+b| + \eta$$

with $\varepsilon = 2050 \times 2^{-64}$ and $\eta = 2049 \times 2^{-1086}$ (apply Formula (3.1) on page 31 for the addition), we will use a formula (with a given ε' and η'):

$$|a \oplus b - (a+b)| \leq \varepsilon' \cdot (|a| + |b|) + \eta' \tag{4.1}$$

The new error proportional to $|a| + |b|$ may increase the bound on the rounding error but it handles the cancellations.

To prove a program in multiple environments, we will change the definition of the result of an addition. As we have said in Section 3.3, the post-condition describes how to calculate an operation result. Here, we modify the post-conditions to cover all cases, including the fact that there are several possible results:



Note that the strict IEEE-754 definition implies a rounding error formula of the same type but is moreover deterministic. The advantage of modifying the operation post-condition is that it also handles reordering when intermediate values are handled. For example, `x=a+b; y=x+c;` can be reordered into `y=a+(b+c)` if `x` is unused and `b+c` already computed.

To reason about any ordering of the additions, let us consider a generic algorithm for adding a sequence of numbers [42].

Algorithm 1 *Let $\mathcal{S} = \{a_0, \dots, a_n\}$.*
Repeat while $\mathcal{S}$ contains more than one element
Remove two numbers x and y from $\mathcal{S}$
and add their sum $x \oplus y$ to $\mathcal{S}$.
Return the remaining element.

This generic algorithm is instantiated by the choice of the two numbers that are removed from $\mathcal{S}$. We will denote by o an ordering and by S_n^o the result of Algorithm 1 for the ordering o. For example, if you choose the preceding computed value and the a_i of smaller index, you get the left-associated summation $(((a_0 + a_1) + a_2) + \cdots) + a_n$.

To ensure the correctness of the approach of taking Formula 4.1 as post-condition, we proved the following theorem for a positive ε. We pose $\varepsilon_n = (1+\varepsilon)^n - 1$.

Theorem 4.1 *Let n be an integer such that $n \leq \frac{1}{\varepsilon}$, $(a_i)_{0 \leq i \leq n}$ be a sequence of real numbers and I be a real. Assume that we set the addition post-condition as: $x \oplus y$ is any real number r such that*

$$|r - (x+y)| \leq \varepsilon_n \cdot (|x| + |y|) + n \cdot \eta.$$

and for an ordering o_1 of the additions, we are able to deduce that $|S_n^{o_1} - \sum_0^n a_i| \leq I$.

Now if we set the addition post-condition as: $x \oplus y$ is any real number r such that

$$|r - (x+y)| \leq \varepsilon \cdot |x+y| + \eta.$$

Then, whatever the ordering o_2 of the additions, we have $|S_n^{o_2} - \sum_0^n a_i| \leq I$.

This means that, if we are able to prove a bound on the rounding error for a sum in a program using our loose post-conditions (Formula (4.1)), then this bound is still correct whatever the compiler reorganization. What is proved using Frama-C and the loose post-conditions (Formula (4.1)) still holds with another ordering (in that case, we use the correct tight post-condition (Formula (3.1) on page 31) proved in the preceding Chapter).

Proof. First, we prove an overestimation of $|S_n^{o_2} - \sum_0^n a_i|$ with $\oplus$ having the $\varepsilon \cdot |x+y|$ property. We prove by induction on n, that whatever the ordering,

$$|S_n^{o_2} - \sum_0^n a_i| \leq \varepsilon_n \sum_0^n |a_i| + n^2 \eta.$$

If $n = 0$, then

$$\begin{aligned}
|S_0^{o_2} - \sum_0^0 a_i| &= |a_0 - a_0| \\
&= 0 \\
&= \varepsilon_0 |a_0| + 0^2 \eta
\end{aligned}$$

so the property holds.

If $n = 1$, then

$$\begin{aligned}
|S_1^{o_2} - \sum_0^1 a_i| &= |a_0 \oplus a_1 - (a_0 + a_1)| \\
&\leq \varepsilon |a_0 + a_1| + \eta \\
&\leq \varepsilon \cdot (|a_0| + |a_1|) + \eta \\
&\leq \varepsilon_1 \sum_0^1 |a_i| + 1^2 \eta.
\end{aligned}$$

If $n = 2$, then

$$\begin{aligned}
|S_2^{o_2} - \sum_0^2 a_i| &= |(a_0 \oplus a_1) \oplus a_2 - (a_0 + a_1 + a_2)| \\
&\leq \varepsilon |a_0 \oplus a_1| + \varepsilon |a_2| + \eta \\
&\leq \varepsilon \cdot (|a_0| + |a_1| + |a_2|) + \varepsilon^2 (|a_0| + |a_1|) + \eta + \varepsilon \eta \\
&\leq \varepsilon_2 \cdot (|a_0| + |a_1| + |a_2|) + 2\eta \\
&\leq \varepsilon_2 \sum_0^2 |a_i| + 2^2 \eta.
\end{aligned}$$

The other orderings of course give the same property so the overestimation of $|S_n^{o_2} - \sum_0^n a_i|$ holds for $n = 2$.

Let us assume that the property holds for any value $k < n$ and that $n \geq 2$ and let us consider a sequence $(a_i)_{0 \leq i \leq n+1}$ and an ordering o_2. As $n+1 > 1$, the value $S_{n+1}^{o_2}$ is computed as the sum of two preceding computed values x and y. And x is a computed sum with a known ordering deduced from o_2 of a part of the $\{a_0, \ldots, a_{n+1}\}$. Let I_1 be such that $x \approx \sum_{i \in I_1} a_i$. If $k = |I_1|$,

then $1 \leq k \leq n+1$. Let us denote $I_2 = \{0, \ldots, n+1\} \setminus I_1$, then $|I_2| = n+1-k$.

$$\begin{aligned}
|S_{n+1}^{o_2} - \sum_0^{n+1} a_i| &= |x \oplus y - \sum_0^{n+1} a_i| \\
&\leq |x \oplus y - (x+y)| + |x - \sum_{i \in I_1} a_i| + |y - \sum_{i \in I_2} a_i| \\
&\leq \varepsilon|x+y| + \eta + |x - \sum_{i \in I_1} a_i| + |y - \sum_{i \in I_2} a_i| \\
&\leq \varepsilon|x - \sum_{i \in I_1} a_i + y - \sum_{i \in I_2} a_i + (\sum_{i \in I_1} a_i + \sum_{i \in I_2} a_i)| + \eta + |x - \sum_{i \in I_1} a_i| + |y - \sum_{i \in I_2} a_i| \\
&\leq \varepsilon(|x - \sum_{i \in I_1} a_i| + |y - \sum_{i \in I_2} a_i| + |\sum_0^{n+1} a_i|) + \eta + |x - \sum_{i \in I_1} a_i| + |y - \sum_{i \in I_2} a_i| \\
&\leq \varepsilon|\sum_0^{n+1} a_i| + \eta + (1+\varepsilon)\left(|x - \sum_{i \in I_1} a_i| + |y - \sum_{i \in I_2} a_i|\right) \\
&\leq \varepsilon\sum_0^{n+1} |a_i| + \eta + (1+\varepsilon)\left(|x - \sum_{i \in I_1} a_i| + |y - \sum_{i \in I_2} a_i|\right)
\end{aligned}$$

And x is the sum of $(a_i)_{i \in I_1}$ with $k-1$ numbers that is less or equal to n so the induction hypothesis can be used. In a similar way, y is the sum of $(a_i)_{i \in I_2}$ with $n+1-k$ numbers that is less or equal to n. Both are using an ordering that can be deduced from o_2.

$$\begin{aligned}
|S_{n+1}^{o_2} - \sum_0^{n+1} a_i| &\leq \varepsilon\sum_0^{n+1} |a_i| + \eta + (1+\varepsilon) \cdot \\
&\quad \left(\varepsilon_k \sum_{i \in I_1} |a_i| + k^2\eta + \varepsilon_{n+1-k} \sum_{i \in I_2} |a_i| + (n+1-k)^2\eta\right) \\
&\leq (\varepsilon + (1+\varepsilon) \cdot \max(\varepsilon_k, \varepsilon_{n+1-k}) \cdot \sum_0^{n+1} |a_i| \\
&\quad + \eta \cdot \left(1 + (1+\varepsilon) \cdot \left(k^2 + (n+1-k)^2\right)\right)
\end{aligned}$$

As (ε_i) is an increasing sequence, and as $k^2 + (n+1-k)^2$ has its maximum value for $k=1$ or $k=n$, we have:

$$\begin{aligned}
|S_{n+1}^{o_2} - \sum_0^{n+1} a_i| &\leq (\varepsilon + (1+\varepsilon)\varepsilon_n) \cdot \sum_0^{n+1} |a_i| + \eta \cdot (1 + (1+\varepsilon)(n^2+1)) \\
&= \varepsilon_{n+1} \cdot \sum_0^{n+1} |a_i| + \eta \cdot (1 + (1+\varepsilon)(n^2+1))
\end{aligned}$$

The last equality is due to this fact:

$$\begin{aligned}
\varepsilon + (1+\varepsilon)\varepsilon_n &= \varepsilon + (1+\varepsilon)((1+\varepsilon)^n - 1) \\
&= \varepsilon + (1+\varepsilon)^{n+1} - (1+\varepsilon) \\
&= (1+\varepsilon)^{n+1} - 1 \\
&= \varepsilon_{n+1}.
\end{aligned}$$

Now we bound the η term:

$$1 + (1+\varepsilon)(n^2+1) = n^2 + 1 + (1+\varepsilon) + \varepsilon n^2$$

As $n \geq 2$, we have

$$1 + (1+\varepsilon)(n^2+1) \leq n^2 + 1 + n + n \cdot (n\varepsilon).$$

And as $n \leq \frac{1}{\varepsilon}$, we deduce

$$1 + (1+\varepsilon)(n^2+1) \leq n^2 + 1 + n + n = (n+1)^2.$$

Therefore,

$$|S^{o_2}_{n+1} - \sum_0^{n+1} a_i| \leq \varepsilon_{n+1} \cdot \sum_0^{n+1} |a_i| + (n+1)^2 \eta,$$

so this overestimation property holds.

Next, we prove that

$$\varepsilon_n \sum_0^n |a_i| + n^2 \eta \leq I$$

For that, we use the first hypothesis. The idea is that, if we were able to prove I with the given post-condition, then we may choose each result of an operation (fulfilling this post-condition) and see which error it creates.

For that, we will pose each operation result. More precisely,

- if neither x, nor y is an a_i, then we choose for $x \oplus y$ the value $x + y + n\eta$;
- if $x = a_i$ and y is not an a_i, then we choose for $x \oplus y$ the value $a_i + y + \varepsilon_n |a_i| + n\eta$;
- if $y = a_i$ and x is not an a_i, then we choose for $x \oplus y$ the value $x + a_i + \varepsilon_n |a_i| + n\eta$;
- if $x = a_i$ and $y = a_j$, then we choose for $x \oplus y$ the value $a_i + a_j + \varepsilon_n |a_i| + \varepsilon_n |a_j| + n\eta$.

All those results fulfill the post-condition requirements. Note also that there will be exactly n additions (whatever the ordering). Therefore,

$$\begin{aligned} I &\geq |S^{o_1}_n - \sum_0^n a_i| \\ &= \left| \varepsilon_n \sum_0^n |a_i| + n \cdot n\eta \right| \\ &= \varepsilon_n \sum_0^n |a_i| + n^2 \eta \\ &\geq |S^{o_2}_n - \sum_0^n a_i| \end{aligned}$$

that ends the proof.

□

We will use the preceding value $\varepsilon = 2050 \times 2^{-64}$ and $\eta = 2049 \times 2^{-1086}$ to handle any rounding of one operation. What is proved is that, if we put Formula (4.1) as post-condition of the addition and subtraction with $\varepsilon' = \varepsilon_n$ and $\eta' = n \cdot \eta$ for a given n, then the produced properties will be correct, even if the compiler re-associate the additions. Note that Formula (4.1) subsumes Formula (3.1) for $n \geq 1$. Note also that $\varepsilon_n = n\varepsilon + O(\varepsilon^2)$ and that a similar value for bounding the rounding error of a sum can be found by Higham [42].

The result is reasonable as the rounding error $\varepsilon' \approx n\varepsilon$. This is an intuitive demonstration of the optimality where we discard the ε^2 terms (which is reasonable as $\varepsilon \approx 2^{-53}$) and we discard underflows. We consider we are only allowed to modify the addition post-condition. In that case, if we consider the post-condition of Formula (3.1) and if we study $(((a_0 \oplus a_1) \oplus a_2) \oplus a_3) \cdots$, then the final error is at least $n \cdot \varepsilon \cdot |a_0| + (n-1) \cdot \varepsilon \cdot |a_1| + \cdots + \varepsilon \cdot |a_n|$. To justify this, we consider an example using 64-bit computations: let $a_0 = 1$ and $a_i = 2^{-53}$, then $(((a_0 \oplus a_1) \oplus a_2) \oplus a_3) \cdots = a_0$ and the error is $n \cdot 2^{-53}$. We are only interested in the first term $(n \cdot \varepsilon \cdot |a_0|)$. Now let us assume we use Formula (4.1) as post-condition and that the program was written $a_0 \oplus (a_1 \oplus (a_2 \oplus (a_3 \cdots)))$ (but the compiler rewrote it in the inverse order). Then the error will be about $\varepsilon' \cdot |a_0| + 2 \cdot \varepsilon' \cdot |a_1| + \cdots + n \cdot \varepsilon' \cdot |a_n|$. As we want this last error to subsume the previous one, we need $\varepsilon' \gtrapprox n \cdot \varepsilon$ to make this approach work.

We need that the ε' of Formula (4.1) be ε_n and η' will be $n\eta$ but we do not know n beforehand. The question left is the choice of n. A solution is to look into the program before to have an overestimation of n. We did not put this idea in practice and decided that 16 will be enough. Of course, for linear algebra, it will be insufficient, but for our examples, it will be correct. Moreover, this value can be changed if a bigger value is needed. We will therefore put $\varepsilon' = 2051 \cdot 2^{-60}$ so that $\varepsilon' \geq \varepsilon_{16} = 16\varepsilon + 256\varepsilon^2(1+\varepsilon)^{16}$ and we will put $\eta' = 16\eta = 2049 \times 2^{-1082}$.

If we constructed a post-condition for a n-term floating-point addition, the results would be better in some cases, but it would not handle the reordering that goes into intermediate values. This is why we chose to only modify the basic block of the numerical program and did so in the best possible way.

Theorem 4.2 is the combination of Theorem 3.1 and Theorem 4.1. It is proved by the previous results.

Theorem 4.2 *Let* $\varepsilon' = 2051 \cdot 2^{-60}$ *and* $\varepsilon = 2050 \times 2^{-64}$. *Let* $\eta' = 2049 \times 2^{-1082}$ *and* $\eta = 2049 \times 2^{-1086}$.
If we define each operation result as any real such that

$$
\begin{aligned}
|x \oplus y - (x+y)| &\leq \varepsilon' \cdot (|x| + |y|) + \eta' \\
|x \ominus y - (x-y)| &\leq \varepsilon' \cdot (|x| + |y|) + \eta' \\
|x \otimes y - (x * y)| &\leq \varepsilon \cdot |x * y| + \eta \\
|x \oslash y - (x/y)| &\leq \varepsilon \cdot |x/y| + \eta \\
\left|\circ(\sqrt{x}) - \sqrt{x}\right| &\leq \varepsilon \cdot \left|\sqrt{x}\right| + \eta
\end{aligned}
$$

and if we are able to deduce a property (such as a rounding error), then this property holds whatever the architecture and the compiler optimizations among commutativity, addition/subtraction associativity (for less than 16 additions/subtractions), use of FMA, use of extended registers, expression factorization and unfactorization.

Without any further work, our method handles other optimizations:

- commutativity: As we have only symmetric formulas for defining the result of an operation, an optimization such as $a+b \longrightarrow b+a$ does not endanger our analysis. And commutativity inside associativity is handled by Theorem 4.1.

- expression factorization: As we only consider rounding errors for one operation, the fact that the compiler factorizes or un-factorizes expressions is not a problem. This includes reordering inside intermediate results.

4.2 Implementation

Similarly to Chapter 3, we create a new pragma `multiroundingR` and define the post-conditions by using the formulas of Theorem 4.2 for basic operations in the Frama-C platform to look into the rounding error of the whole program. With this pragma, we let Frama-C know that floating-point computations may be done with extended registers and/or FMA and/or compiler optimizations.

The parameter declared for each operation is the same as the one in the previous chapter. What is changed in this pragma is the predicate in the post-condition of each operation. Here is the predicate for the post-condition of addition. With subtraction, we do the same way as the one of addition.

```
predicate add_double_post(m:mode,x:double,y:double,result:double)=
abs_real(double_value(result)-(double_value(x)+double_value(y)))
      <=0x1p3000
and
abs_real(double_value(result)-(double_value(x)+double_value(y)))
      <= 0x1.006p-49*(abs_real(double_value(x))
                      +abs_real(double_value(y)))+0x1.002p-1071
and
double_exact(result) = double_exact(x) + double_exact(y)
```

We use Gappa tool to prove floating-point properties, and Gappa need to bound correctly $|x \oplus y - (x + y)|$ as a hint. For this reason, we assume that $|x \oplus y - (x + y)| \leq 2^{3000}$ and this bound is large enough for all cases.

The post-condition of the multiplication is defined as follows:

```
predicate mul_double_post(m:mode,x:double,y:double,result:double)=
abs_real(double_value(result)-double_value(x)*double_value(y))
        <=0x1p3000
and
abs_real(double_value(result)-double_value(x)*double_value(y))
        <=0x1.004p-53*abs_real(double_value(x)*double_value(y))+0x1.002p-1075
and
double_exact(result) = double_exact(x) * double_exact(y)
```

The predicates and parameters of division, negation, square root and absolute value are defined similarly.

Chapter 5

Experimentations

5.1 Double rounding example

This first example is the same as the one in Figure 2.2 and is presented in subsection 2.1.2. We add to the original program an assertion written in ACSL (See Figure 5.1). In the modified program, we define two bounds `A` and `B`. The assertion we need to prove is that z must be between `A` and `B`. Thanks to Gappa, we can find the value of `A` and `B` depending on the pragma we use.

```
#pragma JessieFloatModel(multirounding)

#define A 1.0
#define B 1.0 + 0x1p-52

int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64;
  double z = x + y;
  //@ assert A <= z <= B;
}
```

Figure 5.1: Double rounding example with ACSL annotation.

	A	B
Strict model	$1.0 + 2^{-52}$	$1.0 + 2^{-52}$
Multi-rounding	1.0	$1.0 + 2^{-52}$
Multi-rounding + reorganization	$1.0 - (2^{-49} - 2^{-53})$	$1.0 + 2^{-49}$

With the multirounding model, we automatically prove that in every architecture or compiler, the value of z in this program is always in $[1, 1 + 2^{-52}]$. If we use the multirounding model with addition reorganization, z will be in $[1.0 - (2^{-49} - 2^{-53}), 1.0 + 2^{-49}]$. This bound is higher than the previous one but it ensures that with double rounding or not, with reordering of addition or not, the bound is still hold. With the strict model, the value z is proved to be $1 + 2^{-52}$ where only 64-bit rounding is used.

```
#pragma JessieFloatModel(multirounding)

#define E 0x1.aap-42

/*@ logic integer l_sign(real x) = (x >= 0.0) ? 1 :-1;*/

/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures \abs(\result) <= 1 &&
  @     (\result != 0  ==> \result == l_sign(\exact(x)));
  @*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}

/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures \result != 0
  @   ==>\result==l_sign(\exact(sx) * \exact(vx) + \exact(sy) * \exact(vy))
  @             *l_sign(\exact(sx) * \exact(vy) - \exact(sy) * \exact(vx));
  @*/
int eps_line(double sx,double sy, double vx,double vy){
  int s1=sign(sx*vx+sy*vy,-E,E);
  int s2=sign(sx*vy-sy*vx,-E,E);
  return s1*s2;
}
```

Figure 5.2: Avionics program

5.2 KB3D example

This example includes possible FMA and/or extended registers use but not addition reordering (only one addition). It is part of KB3D [28][1], an aircraft conflict detection and resolution program. The aim is to make a decision corresponding to value -1 and 1 to decide if the plane will go to its left or its right. The inputs are the relative position and speed of the other aircraft. Note that KB3D has been formally proved correct using PVS and under the assumption that the calculations are exact [28]. However, in practice, when the computed value is small, the result may be inconsistent or incorrect. The original code is in Figure 2.3 and may give various answers depending on the architecture/compilation. To prove the correctness of this program which is independent to the architecture/compiler, we need to modify this program to detect when the answer may be incorrect.

The modified program (See Figure 5.2) provides an answer that may be 1, -1 or 0. The idea is that, if the result is nonzero, then it is correct. If the result is 0, it means that the result may be under the influence of the rounding errors and the program is unable to give a certified answer. The correctness of the modified program is proved with respect to the following specification: if the result is nonzero, it is the same as if the computations were done on real

[1] See also http://research.nianet.org/fm-at-nia/KB3D/.

numbers.

In the original program, the discrepancy of the result is derived from the function

```
int sign(double x)
```

To use this function only at the specification level, we define a logic function

```
logic integer l_sign (real x)
```

with the same meaning. Then we define another function

```
int sign (double x, double e1, double e2)
```

that gives the sign of x provided we know its rounding error is between e_1 and e_2. In the other cases, the result is zero.

The function

```
int eps_line (double sx, double sy, double vx, double vy)
```

of Figure 5.2 then does the same computations as the one of Figure 2.3, but the result may be different. More precisely, if the modified function gives a nonzero answer, it is the correct one (it gives the correct sign). But it may answer zero (contrary to the original program) when it is unable to give a certified answer. As in interval arithmetic, the program does not lie, but it may not answer.

About the other assertions, the given values of sx, vx, sy, vy are reasonable for the position and the speed of the plane. The assertions about $s1$ and $s2$ are here to help the automatic provers.

The most interesting part is the value chosen for e_1 and e_2: they need to bound the rounding error of the computation $sx * vx + sy * vy$ (and its counterpart). Thanks to Gappa, we can find the bound of the rounding error e_1 and e_2 and we can also prove that no overflow occurs.

The following table shows us the value of e_1 and e_2 depending on each model. In practice, $e_1 = e_2 = E$. Thus, we will show only E in this table:

	E
Strict model	`0x1p-45`
Multi-rounding	`0x1.90641p-45`
Multi-rounding + re-organization	`0x1.aap-42`

In the usual formalization where all computations directly round to 64 bits, the value $E =$ `0x1p-45` is correct. With multi-rounding approach, we have proved that the value $E =$ `0x1.90641p-45` is correct whatever the architecture. We can also have the value $E =$ `0x1.aap-42` when applying Theorem 4.2 (although it is useless because there is only one addition). This means that the rounding error of $sx*vx+sy*vy$ will always be smaller than this value whatever the architecture or the compiler choices. This means that, even if a FMA is used or if extended registers are used somewhere, this function *does not lie.*

The analysis of this program (obtained from the verification condition viewer gWhy [35]) is given in Figure 5.3. By combining different automatic theorem prover: Alt-Ergo, CVC3, Gappa, we successfully prove all proof obligations in this program.

Proof obligations	Alt-Ergo 0.91	CVC3 2.2 (SS)	Gappa 0.13.0	Statisti
Function eps_line Default behavior				1/1
1. postcondition				
Function eps_line Safety				13/13
1. check FP overflow				
2. check FP overflow				
3. check FP overflow				
4. check FP overflow				
5. check FP overflow				
6. precondition for user				
7. precondition for user				
8. check FP overflow				
9. check FP overflow				
10. check FP overflow				
11. check FP overflow				
12. precondition for use				
13. precondition for use				
Function sign Default behavior				6/6
1. postcondition				
2. postcondition				
3. postcondition				
4. postcondition				
5. postcondition				
6. postcondition				

```
result2: double
H13: double_of_real_post(nearest_even,
0x1.aap-42, result2)
H14: no_overflow_double(nearest_even, -
double_value(result2))
result3: double
H15: neg_double_post(nearest_even, result2,
result3)
H16: no_overflow_double(nearest_even,
0x1.aap-42)
result4: double
H17: double_of_real_post(nearest_even,
0x1.aap-42, result4)

double_value(result3) <= double_value(result1)
- double_exact(result1)
```

```
/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @    \result != 0
  @      ==> \result == l_sign(\exact(sx)*\exact
(vx)+\exact(sy)*\exact(vy))
  @            * l_sign(\exact(sx)*\exact(vy)-
\exact(sy)*\exact(vx));
  @*/

int eps_line(double sx, double sy,double vx,
double vy){
  int s1,s2;
  s1=sign(sx*vx + sy*vy, -0x1.aap-42,
0x1.aap-42);
  s2=sign(sx*vy-sy*vx, -0x1.aap-42, 0x1.aap-42);
  return s1*s2;
}
```

Figure 5.3: Result of Figure 5.2 program

5.3 Summation example

To demonstrate our choices about summation reordering, we use an example by Ogita, Rump and Oishi [62]. Take $\delta = 2^{-54}$. Then we add 1, δ, -1, δ^2 and $-\delta$. We denote by $\oplus$ the 64-bit addition:

- exact computation: $1 + \delta + (-1) + \delta^2 + (-\delta) = \delta^2$
- left-associated floating-point additions:
  $(((1 \oplus \delta) \oplus (-1)) \oplus \delta^2) \oplus (-\delta) = -\delta$
- right-associated floating-point additions:
  $1 \oplus (\delta \oplus ((-1) \oplus (\delta^2 \oplus (-\delta)))) = 0$

From this example, we make a small program (see Figure 5.4). Here, the reordering is critical. With the strict model, we are able to prove that $a = -\delta$, that $b = 0$ and that the exact values of a and b are equal to δ^2, and that no overflow occur. But if the compiler reorders these additions (if we had not put parentheses for example), then these proved properties are fallacious. In the `multiroundingR` pragma, we are only able to prove that the rounding error of a and b is smaller than $0x1.0041p-47$ and that no overflow occur. The rounding error is the same for a and b as this rounding error is big enough to cover all possible orderings, including the left- and

```
#pragma JessieFloatModel(multiroundingR)

void main(){
  double delta = 0x1p-54;
  double a = (((0x1p0 + delta) + (-0x1p0)) + delta*delta) + (-delta);
  /*@ assert \abs(a-(0x1p0+delta+(-0x1p0)+delta*delta +(-delta)))
                                                    <= 0x1.0041p-47;*/
  //@ assert a == -delta;
  double b = 0x1p0 + (delta + ((-0x1p0) + (delta*delta + (-delta))));
  /*@ assert \abs(b-((((0x1p0+delta)+(-0x1p0))+delta*delta)+(-delta)))
                                                    <= 0x1.0041p-47;*/
  //@ assert b == 0;
}
```

Figure 5.4: Summation program

the right-associated ones. Of course, the obtained error is bigger than what may really happen as there are cancellations, but this is *correct* whatever the order of operations. We noticed that Formula (4.1) is especially loose when cancellations happen as the error is proportional to $|x| + |y|$ instead of to $|x + y|$.

This program is fully proved by Gappa using Why/Frama-C with the `multiroundingR` pragma.

5.4 Clock drift example

Clock drift is a phenomena where two clocks do not run at the same speed. This causes a drift of normal clock compared to the actual time. It seems not very important if our clock drifts only some seconds. However, this becomes dramatic in some cases such as the error in a Patriot missile launcher, and caused several deaths [2]. The interesting point of this example is that we try to verify a property on the bound between the computed clock and the exact one.

The example of clock drift [13] [3] is presented in Figure 5.5. We assume that the initial value of `t` is 0.0. At each step, `t` increases by `0.1`. This is a classical example to illustrate rounding errors, that is `0.1` is not a representable floating-point number. Here, the rounding error of `0.1` in double is `A = 5.55114e-18`. The bound `B` is the rounding error of `t - (t'+(double)0.1)` where `t'` is the previous value of `t`. At each step `i`, we assure that `|t-i×0.1|` is bound by `i*(A+B)`. The post-condition says that after `n` steps:

$$|result - n \times 0.1| \leq n \times (A + B)$$

There are several lemmas which are added to help the automatic provers to solve the VCs. This example is proved completely and automatically by the combination of Gappa, Alt-Ergo and CVC3. The value of `B` changed in function of `NMAX`, is determined using Gappa, shown in the following table:

[2] http://autarkaw.wordpress.com/2008/06/02/round-off-errors-and-the-patriot-missile/
[3] This example can also be found at http://hisseo.saclay.inria.fr/drift.html

```
#pragma JessieFloatModel(multirounding)

#define NMAX 100
#define NMAXR 100.0

//@ lemma real_of_int_inf_NMAX: \forall integer i; i <= NMAX ==> i <= NMAXR;

//@ lemma real_of_int_succ: \forall integer n; n+1 == n + 1.0;

//@ lemma inf_mult : \forall real x,y,z; x<=y && 0<=z ==> x*z <= y*z;

// A is a bound of (double)0.1 - 0.1
#define A 5.55114e-18

//@ lemma round01: \abs((double)0.1 - 0.1) <= A;

#define B 1.122421e-15
// B is a bound of round_error(t+(double)0.1) for 0 <= t <=NMAXR+ 0.01

#define C (B + A)

/*@ requires 0 <= n <= NMAX;
  @ ensures \abs(\result - n*0.1) <= n * C;
  @*/
double f_single(int n)
{
  double t = 0.0;
  int i;

  /*@ loop invariant 0 <= i <= n;
    @ loop invariant \abs(t - i * 0.1) <= i * C ;
    @ loop variant n-i;
    @*/
  for(i=0; i < n; i++) {
  L:
    //@ assert 0.0 <= t <= NMAXR*(0.1+C) ;
    t = t + 0.1;
    //@ assert \abs(t - (\at(t,L) + (double)0.1)) <= B;
  }
  return t;
}
```

Figure 5.5: Clock drift program

	NMAX		
	10	**100**	**1000**
strict model/default	1.110224e-16	8.881785e-16	7.10543e-15
multirounding	1.22244e-16	1.122421e-15	1.11242e-14
multirounding+re-organization	1.956855e-15	1.79675e-14	1.78074e-13

5.5 Scalar product example

The annotated C program in Figure 5.6 computes the scalar product (also known as dot product) of two vectors represented as arrays of doubles:

$$\sum_{0 \le i < n} x_i y_i$$

Each vector has n elements. We assume that the values of all the elements of two vectors are not greater than `1.0`. Because of the rounding error, the value of $p - exact(p)$ increases after each step of $p = p' + x_i \times y_i$. This value may change depending on the options of compiler/architecture: it might either follow strictly the standard IEEE-754, or use x87 with 80-bit internal registers, or x87 with optimization, or FMA. In this example, we use a bound `B` for the rounding error of $p-(p'+x_i \times y_i)$ where p' is the previous value of p. The post-condition expresses a bound `B` on the accumulated rounding error in function of a bound `NMAX` on the size of the vectors. Several extra assertions are added in the body of the loop: these are needed to help the automatic provers to solve the generated VCs. In particular, to make Gappa solve the VCs on the accumulated rounding error, it is necessary to guarantee that `p` remains bounded: it is bounded by `NMAX`(1+`B`).

The value of `B` in function of `NMAX` when proving the program with the strict model and our multirounding and multirounding with the addition re-organization are shown in the table below:

	NMAX		
	10	**100**	**1000**
strict model/default	`0x1.1p-50`	`0x1.02p-47`	`0x1.004p-44`
multirounding	`0x1.629p-46`	`0x1.94ep-43`	`0x1.f55p-40`
multirounding + re-org.	`0x1.62ap-46`	`0x1.94ep-43`	`0x1.f55p-40`

```
#pragma JessieFloatModel(multirounding)

#define NMAX 1000
#define NMAXR 1000.0
#define B 0x1.f57d5p-44

/*@ lemma bound_int_to_real:
  @   \forall integer i; i <= NMAX ==> i <= NMAXR; */

/*@ lemma triangular_inequality:
  @   \forall real x,y,z; \abs(x-z) <= \abs(x-y) + \abs(y-z); */


/*@ requires 0 <= n <= NMAX;
  @ requires \valid_range(x,0,n-1) && \valid_range(y,0,n-1) ;
  @ requires \forall integer i; 0 <= i < n ==>
  @            \abs(x[i]) <= 1.0 && \abs(y[i]) <= 1.0 &&
  @            x[i] == \exact(x[i]) && y[i] ==\exact(y[i]) ;
  @ ensures \abs(\result - \exact(\result)) <= n * B; */
double scalar_product(double x[], double y[], int n) {
  double p = 0.0;
  /*@ loop invariant 0 <= i <= n ;
    @ loop invariant \abs(\exact(p)) <= i;
    @ loop invariant \abs(p - \exact(p)) <= i * B;
    @ loop variant n-i; */
  for (int i=0; i < n; i++) {
    // bounds, needed by Gappa
    //@ assert \abs(x[i]) <= 1.0;
    //@ assert \abs(y[i]) <= 1.0;
    //@ assert \abs(p) <= NMAXR*(1+B) ;

  L:
    p = p + x[i]*y[i];

    // bound on the rounding errors in the statement above, proved by gappa
    /*@ assert \abs(p - (\at(p,L) + x[i]*y[i])) <= B; */

    // the proper instance of triangular inequality to show the main invariant
    /*@ assert \abs(p - \exact(p)) <=
              \abs(p - (\at(p,L) + x[i]*y[i])) +
              \abs((\at(p,L) + x[i]*y[i]) - (\exact(\at(p,L)) + x[i]*y[i]));*/

    // a lemma to show the invariant \abs(\exact(p)) <= i
    /*@ assert \abs(\exact(p)) <=
                     \abs(\exact(\at(p,L))) + \abs(x[i]) * \abs(y[i]);*/

    // a necessary lemma, proved only by gappa
    //@ assert \abs(x[i]) * \abs(y[i]) <= 1.0;
  }
  return p;
}
```

Figure 5.6: Scalar product program

Part II

Hardware-dependent proofs of numerical programs

Chapter 6

Principle of proofs on assembly code with Why

This chapter begins a new part of the thesis. Remind that the goal of this part is to show us how to prove a C program on its assembly code. In order to do it, we present firstly the principle of the proofs on assembly code with Why, then we present in each chapter how to translate an assembly program into Why from the simplest cases to the complicated ones.

Before talking about the translation of assembly code into Why, this chapter will give a general view of our approach and some basic knowledge about assembly code.

6.1 Steps of proofs

Our approach for proving a C source via analyzing its assembly follows several steps illustrated on Figure 6.1. All the necessary steps to prove a program with their assembly code are presented in the left hand-side of the figure. The one in the right hand-side instantiates these steps concretely for the proof of some program `foo.c`.

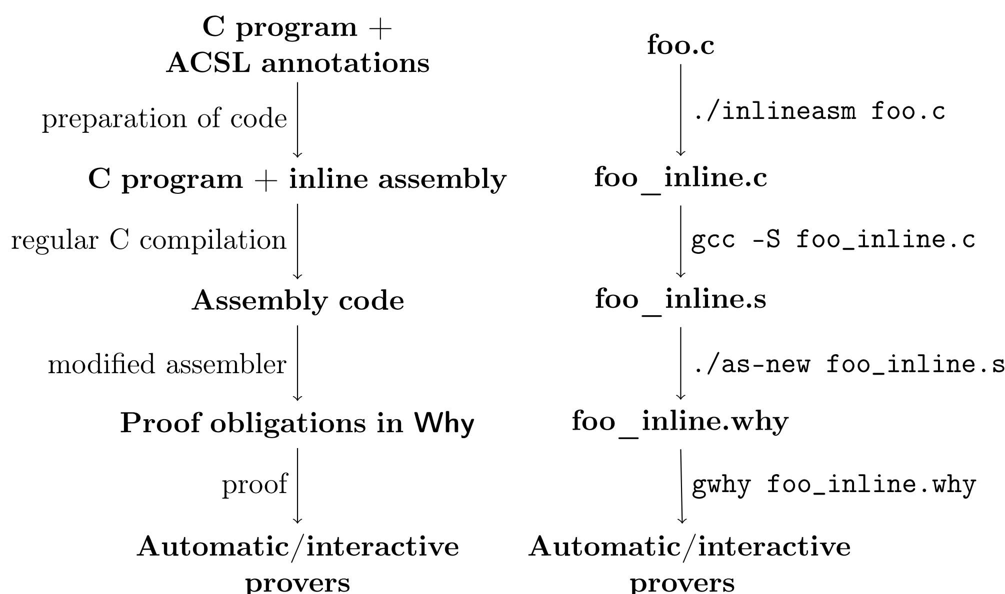


Figure 6.1: Step-by-step from C program to Why proof obligations

In a C program with annotations written by ACSL, all annotations are put in comments. When `gcc` generates assembly code, these annotations will be ignored. As annotations are important to prove the program, a preparation step is needed. This step puts all annotations into inline assembly in order to keep them in assembly code. This will be detailed in Section 6.3.

Once the preparation step is done, another C file containing inline assembly is generated. The regular GNU compiler `gcc` is called with option `-S` to generate assembly code from this C file.

The translation from assembly code to Why is implemented in our own modified version of the GNU assembler GAS. This step generates a file containing proof obligations in Why. These obligations are then attempted to be proved by automatic or interactive provers.

6.2 Essential elements of assembly language

An assembly language is a low-level programming language. It is directly influenced by the instruction set and architecture of the processor. A program written in assembly language consists of a series of statements. They are translated by an assembler to a stream of executable instructions that can be loaded into memory and executed.

There exist many different assemblers which translate assembly source code into binary programs such as NASM [1], MASM [2], GAS [3], etc. The GNU Assembler, commonly known as GAS, is the default back-end of `gcc` and it is a part of the GNU Binutils package. By default, on the x86 and x86-64 architecture, it uses the AT&T assembler syntax. We use `gcc` to generate assembly code. This chapter will give a background on assembly language using the AT&T syntax.

Assembly statements are entered one per line in the source file. They use the same format:

[label] mnemonic [operands] [comment]

The fields in the square brackets are optional.

6.2.1 Operands and Instruction Naming

Operands

An operand in assembly language may be a register, a memory reference or a constant. Note that with AT&T syntax, the first operands are the sources and the last one is the destination (if exists).

Registers are preceded by '%'. For example: the EAX register is specified as %eax

Memory references Memory references in AT&T syntax has the following form:
section:disp(base, index, scale)

where `base` and `index` are the optional 32-bit base and index registers, `disp` is the optional displacement, and `scale`, taking the values 1, 2, 4, 8 and multiplies `index` to calculate the address of the operand. If there is no `scale` specified, it takes 1. `section` specifies the optional section register for memory operand.

The address of a memory reference is calculated by

$$addr(m) = base + scale \times index + disp$$

For example:

[1] Netwide Assembler (NASM): http://www.nasm.us/
[2] Microsoft Macro Assembler (MASM): http://www.masm32.com/
[3] GNU Assembler (GAS): http://sourceware.org/binutils/docs-2.21/as/index.html

- `-4(%rbp)`: `base` is '%rbp'; `disp` is '-4'. `index`, `scale` are both missing.
- `foo(,%eax,4)`: `index` is '%eax'; `scale` is '4'; `disp` is 'foo'. All others fields are missing.

In our model, we suppose that there is no `section`, this means that there is only one section in the memory.

The x86-64 architectures add an RIP (instruction pointer relative) addressing. This addressing mode is specified by using '%rip' as a base register. For example:

- `1234(%rip)` points to the address 1234 bytes past the end of the current instruction.
- `.LC0(%rip)` points to the symbol `.LC0` in RIP relative way [29].

Immediate operands are preceded by '$' and written in decimal or in hexadecimal.
For example: $1, $0x3f800000

Instruction Naming

In AT&T syntax, instruction mnemonics are suffixed with one character modifiers which specify the size of operands. The letter 'b', 'w', 'l', and 'q' specify byte, word, long and quadruple words operands. If no suffix is specified, GAS will try to fill in the missing suffix based on the destination register operand.

For example, in the instruction

```
movl -4(%rbp), %eax
```

`movl` is an instruction mnemonic with the suffix 'l'. This means that the instruction copies data from 32-bit source (`-4%rbp)`) to 32-bit destination (`%eax`).

```
movq %rdi, %rsi
```

`movq` is an instruction mnemonic with the suffix 'q'. This instruction moves data from 64-bit source (`%rdi`) to 64-bit destination (`%rsi`).

6.2.2 EFLAGS register

The 32-bit EFLAGS register holds the state of the processor. It is modified by many instructions and is used for comparing some parameters, conditional loops and conditional jumps. This register contains a group of status flags, a control flag, and a group of system flags. The status flags are presented as follows:

Carry Flag(CF) Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise.

Parity Flag(PF) Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

Adjust Flag(AF) Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic

Zero Flag(ZF) Set if the result is zero; cleared otherwise.

Sign Flag(SF) 0 indicates a positive value and 1 indicates a negative value.

Overflow Flag(OF) Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise.

6.2.3 General-purpose instructions

In this section, we only talk about the general-purpose instructions, floating-point instructions will be presented in Chapter 8.

General-purpose instructions are divided into several groups as follows:

Data transfer instructions

The `mov` instruction transfers data from source operand to destination operand. It requires two operands and has the syntax:

`mov` src, dest

The data is copied from *src* to *dest* and the *src* operand remains unchanged. Both operands should be of the same size. The `mov` instruction can take one of the following five forms:

- `mov` register, register
- `mov` immediate, register
- `mov` immediate, memory
- `mov` register, memory
- `mov` memory, register

Binary Arithmetic Instructions

The following instructions make a binary calculation. They can be used to add/sub/mul/div two 8-, 16-, 32- or 64-bit operands.

`add` src, dest	dest = dest + src
`sub` src, dest	dest = dest - src
`mul` src, dest	dest = dest * src
`div` src, dest	dest = dest / src

Notice that with multiplication and division operations, there exist the signed one and the unsigned one. The unsigned instructions are `mul` and `div`. The signed ones use the prefix 'i': `imul`, `idiv`.

6.2.4 Calling procedures using `call` and `ret`

Stack frame

For function handling in assembly code, the following elements are needed:

- Stack has one stack frame per active function invocation
- Stack pointer register (ESP) points to top (low memory) of current stack frame
- Base pointer register (EBP) points to bottom (high memory) of current stack frame

Each stack frame contains

- Return address (Old EIP [4] (instruction pointer register, points to next instruction to be executed))

[4] In Intel 64 architecture, `ESP`, `EBP` and `EIP` are replaced by `RSP`, `RBP` and `RIP`, respectively

- Old EBP
- Saved register values
- Local variables
- Parameters to be passed to callee function

`call` and `ret` instructions

The `call` instruction allows control transfers to procedures within the current code segment (near call) and in a different code segment (far call). Near calls usually provide access to local procedures within the currently running program or task. Far calls are usually used to access operating system procedures or procedures in a different task.

The `ret` instruction also allows near and far returns to match the near and far versions of the `call` instruction.

In this thesis, we only talk about near `call` and `ret` operation.

When executing a near call, the processor does the following steps:

1. Pushes the current value of the EIP register on the stack.
2. Loads the offset of the called procedure in the EIP register and begins execution of the called procedure.

When executing a near return, the processor performs these actions:

1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. Resumes execution of the calling procedure.

6.2.5 Some assembler directives

All directives are specified by D. Elsner et al. [29]. In this section, we present some of them which often appear in our examples.

Comm directive

The syntax of Comm directive is:

`.comm` *symbol*, *length*

It declares a common symbol named *symbol*. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If the GNU linker `ld` does not see a definition for the symbol – just one or more common symbols – then it will allocate *length* bytes of uninitialized memory. The argument *length* must be an absolute expression. If `ld` sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

Global directive

Its syntax is:

`.globl` *symbol*
`.global` *symbol*

The directive `.global` makes the symbol visible to `ld`. If we define symbol in our partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program. Both spellings `.globl` and `.global` are accepted, for compatibility with other assemblers.

CFI directives

The directive `.cfi_startproc` is used at the beginning of each function that should have an entry in `.eh_frame`. It initializes some internal data structures.

The directive `.cfi_endproc` is used at the end of a function where it closes its unwind entry previously opened by `.cfi_startproc`, and emits it to `.eh_frame`.

6.2.6 Inline assembly

Inline assembly is a way to put directly assembly in high-level source code. We present here the basic syntax of inline assembly which will be used in the next chapters.

Simple Inline Statement

The form of a basic inline statement is:

`asm(“`*assembly code*`”);`

For example: `asm(“move %eax, %ebx”);`

Extended Inline Statements

In basic inline assembly, we have only instructions. In extended assembly, we can also specify the operands. The format of the `asm` statement consists of four components below:

```
asm(  assembly template
      : outputs /* optional */
      : inputs /* optional */
      : clobber list /* optional */
   );
```

where each component is separated by a colon (:). The last three components are optional.

Assembly template consists of the assembly language statements to be inserted into the C code. This may be a single instruction or a sequence of instructions.

Outputs specify the output operands for the assembly code. The format specifying each operand is `“=option-constraint”` where `option-constraint` may be:

- `r` : register operand constraint
- `m` : memory operand constraint
- `rm` : register or memory
- `ri` : register or immediate
- `g` : general
- `X` : any operand whatsoever is allowed[5].

[5] See http://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html

Inputs are specified in the same way, except for the '=' sign.

Clobber list is the list of registers modified by the assembly instructions.

The operands specified in the output and input parts are assigned sequence numbers 0, 1, 2, etc. For example:

```
asm(  ‘‘movl %0, %1’’
      :’’=r’’(sum)/* output */
      :’’r’’(number)/* input */
  );
```

The C variables `sum` and `number` are both mapped to registers. In assembly statement, `sum` is identified by %0 and `number` by %1.

We can put the keyword `volatile` after `asm` if our assembly statement must execute where it is put. Its form is:

```
asm volatile (...:  ...:  ...:  ...)
```

6.3 Preparation of source code

As said in Section 6.1, the goal of this step is to keep annotations when compiling to assembly code. The idea of this step is that we create a new C file in which all annotations are put in inline assembly statements. The assembly code will be generated from this new C file, not the original one.

By using inline assembly in C program, all the variables in annotations will be replaced by a memory reference or a register in assembly code.

For example, an annotation in ACSL :

```
/*@ requires n >= 0 && n < 100;*/
```

is put in inline assembly under the following format:

```
asm volatile("/*requires #int#%0# >= 0 && #int#%1# < 100;*/"::"X"(n),"X"(n));
```

where the first occurrence and the second one of n in this annotation are represented by "%0", "%1".

Notice that in assembly code, we do not know the type of variables. This is the reason why in this step, we add type of variables into annotations. In the example above, each variable is replaced by `#type_of_variable#index#`. We indicate that this variable is input option of inline assembly statement with the constraint "X". The inline assembly above will then be translated in assembly code as follows:

```
/*requires #int#-20(%rbp)# >= 0 && #int#-20(%rbp)# < 100;*/
```

In the programs containing labels like in the following piece of code:

```
for(i=0; i < n; i++) {
L:
  t = t + 0.1f;
  //@ assert \abs(t-(t@L+\round_single(\nearest_even,0.1)))<=B;
}
```

There is a label `L` and in the assertion, we use `t@L` which means that we get the value of `t` at label `L`. When compiling this code into assembly, the label `L` will disappear. In order to keep this label in assembly code, we also put it in inline assembly statement. In this case, the inline assembly is

```
asm volatile("L:");
```

An another point we process in this step is to have only one returned value. This is not a new work, it have been described in CIL (C Intermediate Language) [6]. This ensures that each function has one return statement. For example, the function `sign` below has three returns statements:

```
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;

  return 0;
}
```

The output after the preparation step is

```
int sign(double x, double e1, double e2) {
  int res;

  if (x > e2) {res =  1; goto resL;}
  if (x < e1) {res =  -1; goto resL;}

  {res =  0; goto resL;}
resL:
  return res;
}
```

In the output of this step, we define a local variable `res` (if this variable does not exist in the program) with its type is the returned type of the function. In this case, the returned type is `int`. Then, we replace each return statement `return A;` by `{res = A; goto resL;}`. Finally, at the end of the function, we add `resL: return res;`.

In brief, what we do in this step are

- We replace all the macros in annotations
- We translate annotations under the form of inline assembly statements. We also keep the type of variables in annotations. This is very important because at assembly level, we do not know the type of the registers/memory references.
- We put the labels in C program into inline assembly so that it will appear in assembly code with the same name.
- We have only one return statement in each function.
- Precondition and post-condition are put at the right place. This is important when the function is integrated into the caller. The precondition and post-condition are also integrated with inline function in the right position in assembly code.

[6] http://www.cs.berkeley.edu/~necula/cil/

```
/*@ requires n >= 0 && n < 100;*/
int f(int n){
  int tmp = 100 - n;
  //@ assert tmp > 0;
  //@ assert tmp <= 100;

  return tmp;
}
```

Figure 6.2: A simple program

```
int f(int n){
  asm volatile("/* requires #int#%0#>=0 && #int#%1#<100;*/"::"X"(n),"X"(n));
  int res;
  int tmp = 100 - n;
  asm volatile("/* assert #int#%0# > 0;*/"::"X"(tmp));
  asm volatile("/* assert #int#%0# <= 100;*/"::"X"(tmp));
  {res =  tmp; goto resL;}

resL:
  return res;
}
```

Figure 6.3: The program of Figure 6.2 after passing the preparation step

---

The advantage of using inline assembly statements is that when the program is compiled with `gcc -S`, the compiler will replace all the variables in the annotations automatically by registers/memory refenrences, even with optimization options. However, the assembly code generated may be modified if we use the complex statement of inline assembly. Adding more move instructions is one of the modifications.

6.4 Examples

Now let us see a first simple example (Figure 6.2) to know about a program in assembly code and what we obtain in assembly code after passing the preparation step.

The example in Figure 6.2 has a function `int f(int n)` that returns the value of $100 - n$. The precondition of this function is $0 \leq n < 100$. We have two assertions in the body of the function. These are $tmp > 0$ and $tmp \leq 100$.

After the preparation step, a new C file containing the program as in Figure 6.3 is created. Precondition and two assertions are put into inline assembly statement. A new variable `int res` is added for returning.

The assembly code of this program (See Figure 6.4) is generated from program in Figure 6.3 by the default option of `gcc`. There are only three basic instructions used: transfer data with `mov` instruction, `sub` instruction and instructions for returning a function.

The function `f` in assembly code is defined as a global symbol with type `@function`(line 1–2). This means that this function is visible in other files. A label `f` begins this function. The body is between two directives `.cfi_startproc` and `.cfi_endproc`.

As one can see, ACSL annotations appear between `#APP` and `#NO_APP` in assembly code.

```
1  .globl f
2          .type   f, @function
3  f:
4  .LFB0:
5          .cfi_startproc
6  ....
7          movl    %edi, -20(%rbp)
8  #APP
9  /*requires #int#-20(%rbp)# >= 0 && #int#-20(%rbp)# < 100;*/
10 #NO_APP
11         movl    -20(%rbp), %eax
12         movl    $100, %edx
13         movl    %edx, %ecx
14         subl    %eax, %ecx
15         movl    %ecx, %eax
16         movl    %eax, -4(%rbp)
17 #APP
18 /* assert #int#-4(%rbp)# > 0;*/
19 /* assert #int#-4(%rbp)# <= 100;*/
20 #NO_APP
21         movl    -4(%rbp), %eax
22         leave
23         .cfi_def_cfa 7, 8
24         ret
25         .cfi_endproc
```

Figure 6.4: Assembly code of the example of Figure 6.3 (compiled by `gcc -S`)

```
/*@ requires 0 <= x <= 1000 ;
  @ ensures \result == x * x;*/
int square(int x){
  int tmp = x * x;
  return tmp;
}

int main(){
  int a = 5;
  int b = square(a);
  //@ assert b== 25;

  return 0;
}
```

Figure 6.5: Square program

These lines are generated by `gcc -S`.

The second example we illustrate here is a program containing a function which is called by another (See Figure 6.5). In this example, there are two functions:

- The function `int square(int x)`: calculates the square value of an integer;

```
int square(int x){
asm volatile("/* requires 0 <= #int#%0# <= 1000 ; */"::"X"(x));
int res;
  int tmp = x * x;
  {res =  tmp; goto resL;}

resL:
asm volatile("/* ensures #int#%0# == #int#%1# * #int#%2#;*/"
                                                    ::"X"(res),"X"(x),"X"(x));
return res;
}
int main(){
int res;
  int a = 5;
  int b = square(a);
asm volatile("/* assert #int#%0#== 25;*/"::"X"(b));
  {res =  0; goto resL;}

resL:
return res;
}
```

Figure 6.6: Example in Figure 6.5 after preparation step

- The function `int main()` then calls the function `int square(int x)` in its body.

The new C program generated after the preparation step is illustrated in Figure 6.6. Like previous example, all annotations are put in inline assembly statement. One point we want to emphasize in this example is that the keyword `\result` is replaced by the variable `res` in post-condition. We insist that this translation is done by our translator called "preparation of source code".

The assembly code of this program is in Figure 6.7. The function `square` is called in assembly code by the instruction `call square` (line 41). Thank to the new variable `res` in the new program in Figure 6.6, the returned value is replaced by the reference memory `-4(%rbp)` without any further step (line 24). If we do not replace `\result` in annotation, it is difficult to determine which register/memory reference is used for returning value.

```
1          .file    "square_inline.c"
2          .text
3          .globl   square
4          .type    square, @function
5   square:
6   .LFB0:
7          .cfi_startproc
8   ....
9          movl     %edi, -20(%rbp)
10  #APP
11  /* requires 0 <= #int#-20(%rbp)# <= 1000 ; */
12  #NO_APP
13         movl     -20(%rbp), %eax
14         imull    -20(%rbp), %eax
15         movl     %eax, -8(%rbp)
16         movl     -8(%rbp), %eax
17         movl     %eax, -4(%rbp)
18         nop
19  .L2:
20  #APP
21  /*ensures #int#-4(%rbp)# == #int#-20(%rbp)# * #int#-20(%rbp)#;*/
22  #NO_APP
23         movl     -4(%rbp), %eax
24  ....
25         ret
26         .cfi_endproc
27  ....
28  main:
29  .LFB1:
30         .cfi_startproc
31  ....
32         movl     $5, -12(%rbp)
33         movl     -12(%rbp), %eax
34         movl     %eax, %edi
35         call     square
36         movl     %eax, -8(%rbp)
37  #APP
38  /* assert #int#-8(%rbp)#== 25;*/
39  #NO_APP
40         movl     $0, -4(%rbp)
41         nop
42  .L4:
43         movl     -4(%rbp), %eax
44         leave
45         ret
46         .cfi_endproc
```

Figure 6.7: Assembly code of the example of Figure 6.6 (compiled by `gcc -S`)

Chapter 7

Case of Simple programs

In the previous chapter, we have presented basic knowledge about assembly code and a general view about how to prove a C program from their assembly code with our approach. We also talked about the preparation step for obtaining the registers/memory references corresponding to the variables in annotations.

In this chapter, we describe how to translate a very simple program C into Why including the translation of instructions and annotations.

7.1 Definition of the class of "simple" C programs

Simple C programs considered in this chapter are made of a set of function definitions, specified with ACSL-style annotations, which satisfy these restrictions:

- The only data types `int` and `long int` are assumed to denote 32-bit and 64-bit 2-complement integers. In particular there are no float types, no arrays and no pointers.
- There are no global variables but only local variables and arguments of the functions.
- The body of any function is restricted to a sequence of assignments, i.e. there is no compound instructions: no loop statements of any kind, neither `if`, nor `switch` and nor `goto` statements.
- The allowed expressions are the arithmetic expressions and the function calls.

This class of programs is simple for us because the corresponding assembly codes contain only general-purpose instructions, neither `jump` instructions nor any floating-point instructions.

7.2 Translation to Why

Now we will detail the translation of assembly code to Why. We present firstly how to translate operands to Why. Secondly, we will talk about how to translate annotations to Why. Finally, we present the translation of instructions.

7.2.1 Translation of 32-bit and 64-bit integers

Why has only unbounded mathematical integers built-in. Thus, 32-bit integers must be defined in Why. We follow here the same technique as what is done in the Jessie plug-in of Frama-C.

The type `int32` is an abstract type for an 32-bit integer.

```
type int32
logic integer_of_int32: int32 -> int
```

The logic function `integer_of_int32` returns an integer value from an `int32`.

We need a predicate `is_int32` which verifies whether an integer is in the range of 32-bit word or not.

```
predicate is_int32(x: int) = -2147483648 <= x and x <= 2147483647
```

The axiom

```
axiom int32_coerce: forall x:int32, is_int32(integer_of_int32(x))
```

assures that all value x having type `int32` always denotes an integer value in the range of 32-bit word.

A 64-bit integer has type `int64`. Like `int32`, we define a following logic function and a predicate for it:

```
type int64
logic integer_of_int64: int64 -> int
predicate is_int64(x: int) =
  -9223372036854775808 <= x and x <= 9223372036854775807
```

We also have an axiom

```
axiom int64_coerce: forall x:int64, is_int64(integer_of_int64(x))
```

Although the 8-, 16- integers are not considered here for simplicity, they are handled similarly, as in Jessie. We also consider signed integers and not unsigned integers, they are handled similarly as well.

7.2.2 Translation of operands

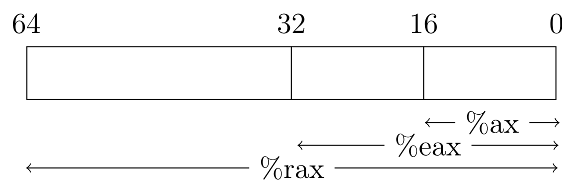
We want to translate operands being registers or memory references into Why variables. To do so, we make the following hypothesis:

Assumption 7.1 (Separation Assumption) *On a simple C program, the compiler generates an assembly code where syntactically distinct memory references denote disjoint memory locations.*

For example, we assume that in any assembly code, the memory references `-16(%rbp)` and `-8(%rax)` are disjoint. Of course there is no reason that this is true in general, but we claim that for the "simple" C programs considered here, and our GNU compiler, this is true. Note that in Chapter 10, this assumption will not be made anymore.

The Separation Assumption allows us to translate each memory reference into a Why variable whose name is syntactically derived from it.

In assembly programs, `%ax`, `%eax`, `%rax` indicate the same register with different size:



As illustrated in the figure above, the size of `%ax`, `%eax` and `%rax` are 16, 32, 64 bits, respectively. When translated into Why, they have the same name: `_rax`. Once we have this, we do not need to cast from `%ax` and `%eax` to `%rax` and otherwise. With other general-purpose registers such as `%bx`, `%cx`, `%di`, `%si`, etc. we do similarly.

The following abstract type will be used in this chapter:

```
type register
```

Each register or memory reference used as an operand will be declared in Why as a variable with type `register`. In order to get a 32-bit and a 64-bit integer from a `register`, we need the two following logic functions:

```
logic sel_int32: register -> int32
logic sel_int64: register -> int64
```

The logic function `sel_int32` and `sel_int64` returns a 32-bit and a 64-bit integer from a `register`.

We distinguish two types of operands as follows:

- Immediate operand: begins by '$'. There is not any declaration here because this operand is a constant. We just delete the prefix '$'.
- Registers and memory references: We denote by $\overline{op}$ the variable in Why corresponding to the operand. Each register or memory reference will have a unique name in Why. In this document, we name the register or memory reference by replacing all special character (()+-%.) by '_'.

  Example: $\overline{\text{-4(\%rbp)}}$ = _4__rbp_. The Why variable to declare for this operand is `parameter _4__rbp_: register ref`

We denote by $[\![op]\!]_{int32}$ and by $[\![op]\!]_{int64}$ Why expressions (of Why type `int`) which denote the integer that *op* represents as a signed 32-bit or 64-bit integer. They are defined by:

$[\![\$imm]\!]_{int32}$ = imm
$[\![reg]\!]_{int32}$ = (integer_of_int32 (sel_int32 !$\overline{reg}$))
$[\![mem]\!]_{int32}$ = (integer_of_int32 (sel_int32 !$\overline{mem}$))
$[\![\$imm]\!]_{int64}$ = imm
$[\![reg]\!]_{int64}$ = (integer_of_int64 (sel_int64 !$\overline{reg}$))
$[\![mem]\!]_{int64}$ = (integer_of_int64 (sel_int64 !$\overline{mem}$))

For example, with the following instruction:

```
movl $5, -4(%rbp)
```

we interpret its operands into Why by

$[\![\$5]\!]_{int32}$ = 5
$[\![-4(\%rbp)]\!]_{int32}$ = (integer_of_int32 (sel_int32 !_4__rbp_))

The instruction

```
movq %rax, -8(%rbp)
```

has two operands `%rax` and `-8(%rbp)`. These operands are interpreted into Why as

$[\![\%rax]\!]_{int64}$ = (integer_of_int64 (sel_int64 !_rax))
$[\![-8(\%rbp)]\!]_{int64}$ = (integer_of_int64 (sel_int64 !_8__rbp_))

7.2.3 Annotations

In order to simplify the translation of annotations in this chapter, the annotations handled are only:

- preconditions
- post-conditions
- assertions

Thanks to the preparation step, all annotations in the original C program are kept in assembly code and all variables in annotations are replaced by registers/memory references.

We denote by $[\![A]\!]_{term}$ the translation of a term (logic expression) into Why. The translation of annotations into Why is described as below:

$$
\begin{array}{lcl}
[\![A \text{ ==> } B]\!]_{term} & = & [\![A]\!]_{term} \texttt{ -> } [\![B]\!]_{term} \\
[\![A \text{ == } B]\!]_{term} & = & [\![A]\!]_{term} \texttt{ = } [\![B]\!]_{term} \\
[\![A \text{ \&\& } B]\!]_{term} & = & [\![A]\!]_{term} \texttt{ and } [\![B]\!]_{term} \\
[\![A \text{ || } B]\!]_{term} & = & [\![A]\!]_{term} \texttt{ or } [\![B]\!]_{term} \\
[\![!A]\!]_{term} & = & \texttt{not(} [\![A]\!]_{term}\texttt{)} \\
[\![\#int\#v\#]\!]_{term} & = & [\![v]\!]_{int32} \\
[\![\#long\#v\#]\!]_{term} & = & [\![v]\!]_{int64} \\
[\![e_1 \ op \ e_2]\!]_{term} & = & [\![e_1]\!]_{term} \ op \ [\![e_2]\!]_{term} \text{ where } op \in \{+,-,*\} \\
[\![e_1 \ / \ e_2]\!]_{term} & = & \texttt{computer_div}([\![e_1]\!]_{term}, [\![e_2]\!]_{term}) \\
[\![e_1 \ \% \ e_2]\!]_{term} & = & \texttt{computer_mod}([\![e_1]\!]_{term}, [\![e_2]\!]_{term}) \\
[\![e_1 \ op \ e_2]\!]_{term} & = & [\![e_1]\!]_{term} \ op \ [\![e_2]\!]_{term} \text{ where } op \in \{>,<,>=,<=\} \\
[\![e_1 \ != \ e_2]\!]_{term} & = & [\![e_1]\!]_{term} \texttt{ <> } [\![e_2]\!]_{term} \\
[\![\backslash forall \ \tau \ i; \ P]\!]_{term} & = & \texttt{forall } i : [\![\tau]\!]_{type} \texttt{. } [\![P]\!]_{term} \\
[\![\backslash exists \ \tau \ i; \ P]\!]_{term} & = & \texttt{exists } i : [\![\tau]\!]_{type} \texttt{. } [\![P]\!]_{term} \\
 & & \\
[\![integer]\!]_{type} & = & \texttt{int} \text{ (unbounded integer, not machine integer)} \\
 & & \\
[\![\backslash abs_int(e)]\!]_{term} & = & \texttt{abs_int(}[\![e]\!]_{term}\texttt{)}
\end{array}
$$

There are built-in constructors in ACSL such as $\backslash abs$, $\backslash max$, $\backslash min$, etc. which are overloaded. This means that we can use these constructors for both integer and real type. Here, we decide to simplify it by using ACSL-style syntax. For example: instead of using $\backslash abs(e)$ which returns the absolute value of an integer or a real expression, we use $\backslash abs_int(e)$ and $\backslash abs_real(e)$.

For example, the following annotation

```
(#int#-4(%rbp)# != 0 ==> #int#-4(%rbp)# == l_sign(\exact(#double#-24(%rbp)#)))
 && \abs_int(#int#-4(%rbp)#) <= 1
```

is interpreted into Why as

```
(integer_of_int32(sel_int32(_4__rbp_)) <> 0 ->
      integer_of_int32(sel_int32(_4__rbp_)) = l_sign((sel_exact(_24__rbp_))))
and
 abs_int(integer_of_int32(sel_int32(_4__rbp_))) <= 1
```

7.2.4 Translation of an instruction

The move instructions and addition/subtraction/multiplication/division instructions are translated thanks to the following abstract functions in Why program:

```
parameter set_int32_no_check: imm:int -> dest: register ref ->
{ }
   unit writes dest
{ integer_of_int32(sel_int32(dest)) = imm }
```

The previous abstract function will set a 32-bit integer to a register without verifying if this value overflows or not.

```
parameter set_int32: imm:int -> dest: register ref ->
{ is_int32(imm) }
   unit writes dest
{ integer_of_int32(sel_int32(dest)) = imm }
```

The post-condition of `set_int32` is the same as `set_int32_no_check`. Its pre-condition verifies whether `imm` is a 32-bit integer.

We denote by $[\![$ ins $]\!]_i$ the Why translation of an instruction `ins`. Instructions in assembly code are interpreted to Why as follows:

$[\![$ movl src, dest $]\!]_i$ = `set_int32_no_check` $[\![src]\!]_{int32}$ $dest$
$[\![$ addl src, dest $]\!]_i$ = `set_int32` $([\![dest]\!]_{int32} + [\![src]\!]_{int32})$ $dest$
$[\![$ subl src, dest $]\!]_i$ = `set_int32` $([\![dest]\!]_{int32}$ - $[\![src]\!]_{int32})$ $dest$
$[\![$ imull src, dest $]\!]_i$ = `set_int32` $([\![dest]\!]_{int32}$ * $[\![src]\!]_{int32})$ $dest$
$[\![$ call label $]\!]_i$ = *label*`_parameter()`

With 64-bit instructions, we declare the following parameters similarly:

```
parameter set_int64_no_check: imm:int -> dest: register ref ->
{ }
   unit writes dest
{ integer_of_int64(sel_int64(dest)) = imm }

parameter set_int64: imm:int -> dest: register ref ->
{ is_int64(imm) }
   unit writes dest
{ integer_of_int64(sel_int64(dest)) = imm }
```

The translation of 64-bit instructions are as follows:

$[\![$ movq src, dest $]\!]_i$ = `set_int64_no_check` $[\![src]\!]_{int64}$ $dest$
$[\![$ addq src, dest $]\!]_i$ = `set_int64` $([\![dest]\!]_{int64} + [\![src]\!]_{int64})$ $dest$
$[\![$ subq src, dest $]\!]_i$ = `set_int64` $([\![dest]\!]_{int64}$ - $[\![src]\!]_{int64})$ $dest$
$[\![$ imulq src, dest $]\!]_i$ = `set_int64` $([\![dest]\!]_{int64}$ * $[\![src]\!]_{int64})$ $dest$

A special case for the translation is that each assertion is considered as an instruction. If we have an assertion `A` then its translation is

$[\![$ /*@ assert A;*/ $]\!]_i$ = `assert` $[\![A]\!]_{term}$;

Notice that `leave` and `ret` are instructions in assembly language but they do not have any translation here. Notice also that when a function is called by another one, there are two instructions: `pushq %rbp` at the beginning of the function and `popq %rbp` at the end of the function before `ret` which push and restore the value of `%rbp`. The memory reference pointed

by `%rbp` is used locally in each function. This means that the addresses of `-4(%rbp)` in two different functions are different. We do not translate `pushq` and `popq` to Why but we can assure that the value of `-4(%rbp)` in two functions are different.

The source `src` of the instructions `movl(q)` is either a constant, a register or a memory reference in 32(64) bits. Therefore, we do not need to verify if it overflows or not. However, for addition/subtraction/multiplication instructions, we need to assure that this computation does not overflow.

The case of division cannot be handled the same way as other operations since the divisor must be check non-null. We thus use

$$[\![\text{ idivl src, dest }]\!]_i = \texttt{div_int32}\ [\![src]\!]_{int32}\ dest$$
$$[\![\text{ idivq src, dest }]\!]_i = \texttt{div_int64}\ [\![src]\!]_{int64}\ dest$$

with the special Why parameters:

```
parameter div_int32: imm: int -> dest: register ref ->
{   imm <> 0
    and
    is_int32(computer_div(integer_of_int32(sel_int32(dest),imm)))
}
    unit writes dest
{
    integer_of_int32(sel_int32(dest)) =
           computer_div(integer_of_int32(sel_int32(dest@)),imm)
}

parameter div_int64: imm: int -> dest: register ref ->
{   imm <> 0
    and
    is_int64(computer_div(integer_of_int64(sel_int64(dest),imm)))
}
    unit writes dest
{
    integer_of_int64(sel_int64(dest)) =
           computer_div(integer_of_int64(sel_int64(dest@)),imm)
}
```

The function `computer_div` is defined in the Why standard library and denotes the integer division which rounds the result towards 0, which corresponds to the usual convention for division in C and other programming languages.

7.2.5 Sequences and functions

Until now, we have described the translation of annotations and the one of each instruction. How do we translate a sequence of instructions or a function containing annotations into Why?

Assume that we have a function with preconditions, post-conditions and assertions. The translation of this function in assembly code into Why is illustrated in Figure 7.1. As we see in this figure, the post-condition becomes an assertion in Why.

For each function having pre- and post-condition, we define an interface of function in Why (at the bottom right hand side of Figure 7.1) where w is is a set of variables modified in the function.

Notice that the real semantic of

`assume` P

```
f:                                     ⟶   let f() =
.cfi_startproc
    /*@ requires P; */                 ⟶     assumes {[[P]]_term};
    (body of the function f)           ⟶     [[ (body of the function f) ]]_i
    /*@ ensures Q; */                  ⟶     assert {[[Q]]_term};
    leave                                    void
    ret
.cfi_endproc                               parameter f: unit ->
                                              { [[P]]_term } unit writes w
                                              { [[Q]]_term }
```

Figure 7.1: Translation of a function in assembly to Why

---

in Why is written as

```
[ {} unit reads w { P } ];
```

where w is a set of variables in P

Translation of annotations in presence of inline function

When the program is compiled with `-O2`, functions may be inlined. This means that the function called is integrated into the caller. Thanks to the inline assembly statement `asm volatile`, we can keep the annotations at the place they must be when integrating.

The question now is how to translate the annotations of inline function? Normally, when a function is called, in Why, we will use its interface and the precondition is demanded to prove. In case of inline function, the translation is specified below

- $[\![precondition_inline]\!]_{term}$ = `assert` $[\![\mathtt{precondition_inline}]\!]_{term}$
- $[\![post\text{-}condition_inline]\!]_{term}$ = `assumes` $[\![\mathtt{post\text{-}condition_inline}]\!]_{term}$

The translation of assertions are not changed in this case.

7.3 Soundness of translation

The goal of this section is to demonstrate that if the verification conditions hold then the assembly program respects its annotations.

To express that a program respects its annotations, we use a *blocking* semantics [40]: the execution of a program will block whenever an invalid annotation is met. Thanks to this definition, to prove that a program respects its annotations, we need to prove that there is a backward simulation [50] of assembly code by the Why code.

Figure 7.2 shows us that the Why program is translated from assembly program. Then, by using Why verification condition generator, verifications condition are generated. Now what we want to prove in this section is that if the verification conditions hold then the Why program does not block (Theorem 7.4 on page 73). If the Why program does not block then the assembly program does not block (Theorem 7.8 on page 75).

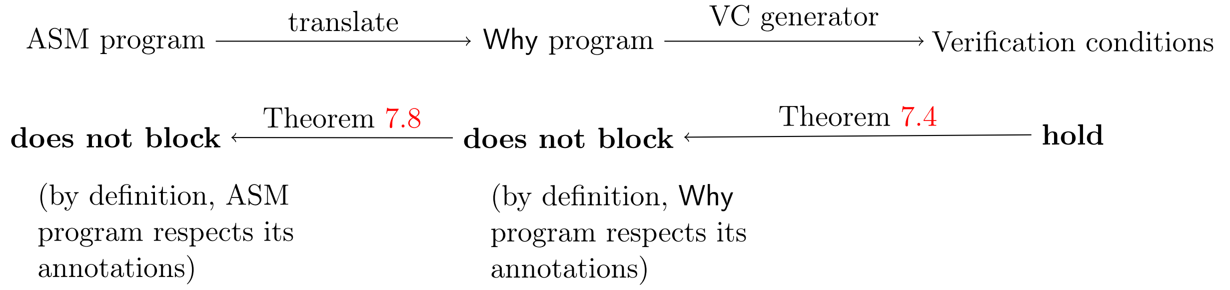


Figure 7.2: Soundness of the translation

7.3.1 Reminder of the soundness of Why

Definition 7.2 (Why state) *A Why state S is a map which gives the value of reference, that is*

- *if r is a Why reference then $S(r)$ denotes the value of r*
- *The notation $S[r \leftarrow v]$ denotes the update of a map.*

We denote (abuse of notation) $S(t)$ the evaluation of a term t in S: it is defined by a natural induction:

$$\begin{array}{rcl} S(t_1 + t_2) & = & S(t_1) + S(t_2) \\ S(t_1 - t_2) & = & S(t_1) - S(t_2) \\ S(t_1 * t_2) & = & S(t_1) * S(t_2) \\ S(t_1 / t_2) & = & S(t_1) \ / \ S(t_2) \\ S(!r) & = & S(r) \end{array}$$

The execution of a program in Why is defined by an operational semantics. We use a big-step-style semantics, that is we define the relation $S,\ P \Rightarrow^m S'$ which means " in state S, the program P executes and terminates after m steps in the state S' ".

It is defined by the following rules:

$$\frac{}{S, r := t \Rightarrow^1 S[r \leftarrow t]}$$

$$\frac{S, p_1 \Rightarrow^m S'' \qquad S'', p_2 \Rightarrow^n S'}{S, p_1; p_2 \Rightarrow^{(m+n+1)} S'}$$

$$\frac{A \ holds \ in \ S}{S, assert \ A \Rightarrow^1 S}$$

Notice that if S(A) does not hold, then the program blocks.

There are two forms of Why functions that we use in this part:

- Case 1: Function (with `let`)
  - with body,
  - without parameters,

- the result is `unit`.

$$\frac{[\![\ Pre\ ]\!]_S\ holds \quad S, body \Rightarrow^n S' \quad [\![\ Post\ ]\!]_{S'}\ holds}{S, f \Rightarrow^{n+1} S'}$$

- Case 2: Function (with `parameter`)
  - without body,
  - with parameters: the pure ones $x_1, \ldots, x_m$ and the reference ones $r_1, \ldots, r_l$,
  - the result is `unit`
  - reads $u_1, \ldots, u_m$
  - `writes` $w_1, \ldots, w_k, r_i, \ldots, r_l$ (where $w_1, \ldots, w_k$ are global references).

$$\frac{\begin{array}{c}[\![\ Pre\ ]\!]_{S \bigcup \{r_i = S(s_i)\} \bigcup \{x_j = S(e_i)\}}\ holds \\ \forall i,j.i \neq j \rightarrow s_i \neq s_j \qquad \forall i,j.\ s_i \neq u_j \qquad \forall i,j.\ s_i \neq w_j \qquad S \underset{w_1,\ldots,w_l,s_1,\ldots,s_k}{\sim} S' \\ [\![\ Post\ ]\!]_{S' \bigcup \{r_i = S'(s_i)\} \bigcup \{x_j = S(e_i)\} \bigcup \{r_i @ = S(s_i)\}}\ holds\end{array}}{S, f(e_1, \ldots, e_n, s_1, \ldots, s_k) \Rightarrow^{n+1} S'}$$

where $S \underset{w_1,\ldots,w_l}{\sim} S'$:= for all references r such that $r \notin \{w_1, \ldots, w_l\}$ then S(r) = S'(r).

The reason why we need to assure that $\forall i,j.i \neq j \rightarrow s_i \neq s_j$ and $\forall i,j.s_i \neq w_j$ is explained in subsection 7.3.2. Notice again that the execution blocks if a precondition or a post-condition is invalid.

Definition 7.3 *A program respects its annotations if it executes without blocking.*

In Why, there is a verification condition generator in which a set of Why functions produces a set of formulas (the verification condition). The soundness of this verification condition generator is expressed by the following Theorem:

Theorem 7.4 *For all Why program P, if the generated proof obligations hold then P executes without blocking.*

In P, the assertions, preconditions and post-conditions are verified when P is executed. Indeed, this theorem was proved by Filliâtre [30, 31]. Recently, Herms certified it in Coq [40].

7.3.2 About the condition in function call

```
parameter x: int ref

parameter inc: y:int ref ->
 {...}
   unit writes x, y
 { y = 1 and x = 2 }
```

We define a global parameter `x`. In the parameter `inc` we modify both `y` and global reference `x`. In this example, $s_1 = $ `y` and $w_1 = $ x. If we call this function as follows:

```
let main() =
 {...}
  inc(x)
 { x = 1 and x = 2}
```

The Why VCGen rejects it with an error: "Application to x creates an alias". If it was accepted, we would be able to prove the post-condition above which is inconsistent. The reason why we also need to have $s_i \neq s_j$ is illustrated by a similar example:

```
parameter inc: x:int ref -> y:int ref ->
 {...}
   unit writes x, y
 { y = 1 and x = 2 }

parameter z: int ref

let main() =
 {...}
  inc(z,z);
 { z = 1 and z = 2}
```

7.3.3 Definition of the execution of an assembly program

Definition 7.5 (Memory state in assembly program) *A memory state S is a map which returns*

- *a value (a bit vector bv) from the name of a register, denoted by $S(r)$*
- *a value stored in memory at any address x, denoted by $S(x)$.*

We denote by $bv_to_int32(bv)$ the value in two's complement integer of a bv, by $int32_to_bv(const)$ the bitvector converted from a two's complement integer in 32 bits $const$. For example:

$bv_to_int32(\texttt{0x00000001}) = 1$
$bv_to_int32(\texttt{0xffffffff}) = -1$

The execution of an instruction i is denoted as: $S,\ i \Rightarrow S'$. It is defined as follows:

$$\frac{}{S,\ movl\ src, dest \Rightarrow S[dest \leftarrow S(src)]}$$

$$\frac{bv_to_int32(S(dest)) + bv_to_int32(S(src))\ does\ not\ overflow}{S,\ addl\ src, dest \Rightarrow S[dest \leftarrow int32_to_bv(bv_to_int32(S(dest)) + bv_to_int32(S(src)))]}$$

$$\frac{bv_to_int32(S(dest)) - bv_to_int32(S(src))\ does\ not\ overflow}{S,\ subl\ src, dest \Rightarrow S[dest \leftarrow int32_to_bv(bv_to_int32(S(dest)) - bv_to_int32(S(src)))]}$$

$$\frac{bv_to_int32(S(dest)) * bv_to_int32(S(src)) \ does\ not\ overflow}{S,\ imull\ src, dest \Rightarrow S[dest \leftarrow int32_to_bv(bv_to_int32(S(dest)) * bv_to_int32(S(src)))]}$$

$$\frac{bv_to_int32(S(dest)) <> 0 \qquad bv_to_int32(S(dest))/bv_to_int32(S(src)) \ does\ not\ overflow}{S,\ idivl\ src, dest \Rightarrow S[dest \leftarrow int32_to_bv(bv_to_int32(S(dest))/bv_to_int32(S(src)))]}$$

$$\frac{}{S,\ leave \Rightarrow S}$$

$$\frac{}{S,\ ret \Rightarrow S}$$

$$\frac{S,\ i_1 \rightarrow S_1 \qquad S_1,\ i_2; \ldots ; i_m \rightarrow S'}{S,\ i_1; \ldots ; i_m \Rightarrow S'}$$

$$\frac{[\![\ Pre\ ]\!]_S\ holds \quad S, instr(P) \Rightarrow S' \quad [\![\ Post\ ]\!]_{S'}\ holds}{S,\ call\ P \Rightarrow S'}$$

where *instr(P)* is a sequence of instructions from the address P until the instruction `ret`. For 64-bit arithmetic instructions, their execution is defined similarly.

7.3.4 Relation between the Why state and the assembly state (case of "simple" programs)

Definition 7.6 *Let bv be a bitvector and reg be a register. bv is congruent to reg (denotes by $bv \cong reg$) if*

- $bv_to_int32(bv) = [\![reg]\!]_{int32}$
- $bv_to_int64(bv) = [\![reg]\!]_{int64}$

Definition 7.7 *Let $\overline{S}$ be a Why state and S be an assembly state. $\overline{S}$ simulates S (denotes by $\overline{S} \sim S$) iff*

1. *For all register r, $S(r) \cong \overline{S}(\overline{r})$ where $bv \cong reg$ with bv is bitvector and reg is register.*
2. *For all memory reference m = `off(reg,...)`, either $\overline{m}$ is defined and then $S(m) \cong \overline{S}(\overline{m})$ or $\overline{m}$ is not defined.*

It is important to notice that in the "simple" case must follow Assumption 7.1, that is two distinguished memory references (syntactically) imply two different addresses. This means that if $address(m_1) = address(m_2)$ then $\overline{m_1} = \overline{m_2}$.

Lemma 1 *For all assembly state S, Why state $\overline{S}$, if $S \sim \overline{S}$ and the translation of A into Why $[\![A]\!]_{annot}$ is true in $\overline{S}$ then A is true in S.*

Proof. This is a straightforward induction on the structure of A. □

Theorem 7.8 *For all assembly state S, Why state $\overline{S}$ such that $\overline{S} \sim S$. For all sequence of assembly instructions i_1, ..., i_n:*
If $\overline{S}$, $[\![\ i_1; \ldots ; i_n\ ]\!]_i \Rightarrow^m \overline{S'}$
then $\exists S'$ such that $S,\ i_1; \ldots ; i_n \Rightarrow S'$ where $\overline{S'} \sim S'$.

This says that if $[\![i_1, \ldots, i_n]\!]$ executes (without blocking) and we obtain the Why state $\overline{S'}$ then $i_1, \ldots, i_n$ executes (without blocking) and we have the final memory state S' of the execution of $i_1, \ldots, i_n$ so that $\overline{S'} \sim S'$.
Proof. By recurrence on m:

1. With $m = 0$: We have $n = 0$.
2. With $m \geq 1$: We have $n \geq 1$. From $\overline{S}, [\![\ i_1, \ldots, i_n\ ]\!]_i \Rightarrow^m \overline{S'}$

   we have $\overline{S}, [\![\ i_1\ ]\!]_i \Rightarrow^p \overline{S_1}$ and $\overline{S_1}, [\![\ i_2, \ldots, i_n\ ]\!]_i \Rightarrow^q \overline{S'}$

   with $p + q + 1 = m$.

   We prove below by case analysis on i_1 that there is a S_1 such that $S,\ i_1 \Rightarrow S_1$ where $\overline{S_1} \sim S_1$.

   By induction, because $q < m$, there exists S' such that

   - $S_1,\ i_2, \ldots, i_n \Rightarrow S'$ and
   - $\overline{S'} \sim S'$

   then $S,\ i_1, \ldots, i_n \Rightarrow S'$ and $\overline{S'} \sim S'$ (proved).

   We now proceed our case analysis. We illustrate the proof by the three following cases: `movl`, `addl` and `call` instruction. With others instructions, we do similarly.

   (a) `movl` instruction: `movl src, dest`
   In assembly program, what we have is:

   $$bv_to_int32(S'(dest)) = bv_to_int32(S(src)).$$

   The translation into Why is:

   $[\![$ movl src, dest $]\!]_i$ = `set_int32_no_check` $[\![src]\!]_{int32}$ $dest$

   From the post-condition of `set_int32_no_check` we have:
   `integer_of_int32(sel_int32(`$\overline{S'}(\overline{dest})$`)) = integer_of_int32(sel_int32(`$\overline{S'}(\overline{src})$`))`
   As $bv_to_int32(bv)$ = `integer_of_int32(sel_int32(`reg`))` and $\overline{S} \sim S$, we have:

   $$bv_to_int32(\overline{S'}(\overline{dest})) = bv_to_int32(S(src)) = bv_to_int32(S'(dest)) \text{ (proved).}$$

   (b) `addl` instruction: `add src, dest`
   In assembly program:
   $bv_to_int32(S'(dest)) = (bv_to_int32(S(src)) + bv_to_int32(S(dest))) \bmod 2^{32}$
   The translation into Why:

   $[\![$ addl src, dest $]\!]_i$ = `set_int32` $([\![dest]\!]_{int32} + [\![src]\!]_{int32})$ $dest$

   The post-condition of `sel_int32` says that
   `integer_of_int32(sel_int32(`$\overline{S'}(\overline{dest})$`))` $=$
   `integer_of_int32(sel_int32(`$\overline{S}(\overline{dest})$`))`$+$`integer_of_int32(sel_int32(`$\overline{S}(\overline{src})$`))`
   As $S \sim \overline{S}$, we have:
   `integer_of_int32(sel_int32(`$\overline{S'}(\overline{dest})$`))`
   $= bv_to_int32(S(dest)) + bv_to_int32(S(src))$
   $= (bv_to_int32(S(dest)) + bv_to_int32(S(src))) \bmod 2^{32}$ (because there is no overflow)
   $= bv_to_int32(S'(dest))$ (proved).

```
/*@ requires n >= 0 && n < 100;*/
int f(int n){
  int tmp = 100 - n;
  //@ assert tmp > 0;
  //@ assert tmp <= 100;

  return tmp;
}
```

Figure 7.3: A simple program

It is important to notice that in this translation, the precondition is necessary as the result of addition may overflow and the destination is in 32 bits. We need to assure that the addition does not overflow. If it overflows then the Why program blocks.

(c) `call` instruction: `call proc`

We have $[\![\ i_1\ ]\!]_i =$ `f()`: invokes a function in Why.

Because $\overline{S}$, $[\![\ f()\ ]\!]_i \Rightarrow^p \overline{S_1}$, the rule of the function call implies

- $[\![\ \overline{Pre}\ ]\!]_{\overline{S}}$ holds, where $\overline{Pre} = [\![\ Pre\ ]\!]_{annot}$
- $\overline{S}, body \Rightarrow^{p-1} \overline{S_1}$, where $body =[\![$ instr(P) $]\!]_i$ (1)
- $[\![\ \overline{Post}\ ]\!]_{\overline{S_1}}$ holds

Lemma 1 says that $[\![\ Pre\ ]\!]_S$ holds.

By induction, because $p - 1 < m$ and there exists S_1 such that

- $\overline{S}, instr(f) \Rightarrow \overline{S_1}$ and
- $S_1 \sim \overline{S_1}$ (2)

From (1), (2) and with Lemma 1, we have $[\![\ Post\ ]\!]_{S_1}$ holds.

Consequently, the rule of call instruction in assembly applies: S, *call* $f \Rightarrow S_1$.

□

7.4 Examples

7.4.1 Simple example

This example is presented in Section 6.4 and its assembly code is in Figure 6.4. The C source code is also shown in Figure 7.3.

The Why program corresponding to the assembly code is presented in Figure 7.4. The assembly instruction is put in the Why comments under the form `(*....*)` and the translation of this instruction follows each instruction.

The function `f` has a precondition and no post-condition. Thus, in the parameter `f_parameter` the post-condition is `true`.

The translation of this example is very simple. There are only two instructions we need to translate: `movl` and `subl` and two assertions. What we prove are these two assertions. They are proved by both Gappa, Alt-Ergo and CVC3 (See Figure 7.5).

```
parameter f_parameter: _:unit->
  { integer_of_int32(sel_int32(_rdi)) >= 0
    and
    integer_of_int32(sel_int32(_rdi)) < 100 }
        unit reads _rdi
  { true }
let f_0() =
_LFB0:
(*#movl %edi, -20(%rbp)*)
 move_reg32 !_rdi _20__rbp_;
[{ }
    unit reads _20__rbp_
 { integer_of_int32(sel_int32(_20__rbp_)) >= 0
    and
    integer_of_int32(sel_int32(_20__rbp_)) < 100}];
(*#movl -20(%rbp), %eax*)
 move_reg32 !_20__rbp_ _rax;
(*#movl $100, %edx*)
 move_cte32 (100) (100.0) _rdx;
(*#movl %edx, %ecx*)
 move_reg32 !_rdx _rcx;
(*#subl %eax, %ecx*)
 set_reg32 ((integer_of_int32 (sel_int32 !_rcx)) -
               (integer_of_int32 (sel_int32 !_rax))) _rcx;
(*#movl %ecx, %eax*)
 move_reg32 !_rcx _rax;
(*#movl %eax, -4(%rbp)*)
 move_reg32 !_rax _4__rbp_;
assert{ integer_of_int32(sel_int32(_4__rbp_)) > 0};
assert{ integer_of_int32(sel_int32(_4__rbp_)) <= 100};
(*#movl -4(%rbp), %eax*)
 move_reg32 !_4__rbp_ _rax;
(*#leave*)
(*#ret*)
void
```

Figure 7.4: Why program of Figure 6.4

7.4.2 Square example

This example is presented in Section 6.4 and shown in Figure 7.6. Its goal is to show us how to translate a function call to Why. For each function in C program, we define a corresponding parameter in Why. Corresponding to the function `square` in assembly code, we define parameter `square_parameter` with no inputs and no outputs. This parameter contains the precondition and post-condition of the function `square` and a set of variables assigned. Once the function `square` is called in the function `main`, the parameter `square_parameter` is invoked (See Figure 7.7) and the precondition of the parameter `square_parameter` is proved. All the obligation proofs are proved (See Figure 7.8).

Proof obligations	Alt-Ergo 0.93	CVC3 2.2 (SS)	Gappa 0.14.1	Statistics
function f_0 Correctness	✓	✓	✓	3/3
1. precondition	✓	✓	✓	
2. assertion	✓	✓	✓	
3. assertion	✓	✓	✓	

```
                                     integer_of_int32(sel_int32(_rax))
_rax0: register
H9: integer_of_int32(sel_int32(_rax0)) = integer_of_int32(sel_int32(_rcx0)) and
    single_value(sel_single(_rax0)) = single_value(sel_single(_rcx0)) and
    sel_exact(_rax0) = sel_exact(_rcx0)
_4__rbp_: register
H10: integer_of_int32(sel_int32(_4__rbp_)) = integer_of_int32(sel_int32(_rax0)) and
     single_value(sel_single(_4__rbp_)) = single_value(sel_single(_rax0)) and
     sel_exact(_4__rbp_) = sel_exact(_rax0)
______________

integer_of_int32(sel_int32(_4__rbp_)) > 0
```

```
let f_0() =
_LFB0:
(*#pushq        %rbp*)
(*#movq %rsp, %rbp*)
 move_reg64 !_rsp _rbp;
(*#movl %edi, -20(%rbp)*)
 move_reg32 !_rdi _20__rbp_;
[{}unit reads _20__rbp_ { integer_of_int32(sel_int32(_20__rbp_)) >= 0  and
integer_of_int32(sel_int32(_20__rbp_)) < 100}];
(*#movl -20(%rbp), %eax*)
 move_reg32 !_20__rbp_ _rax;
(*#movl $100, %edx*)
 move_cte32 (100) (100.0) _rdx;
(*#movl %edx, %ecx*)
 move_reg32 !_rdx _rcx;
(*#subl %eax, %ecx*)
 set_reg32 ((integer_of_int32 (sel_int32 !_rcx)) - (integer_of_int32 (sel_int32 !
_rax))) _rcx;
(*#movl %ecx, %eax*)
 move_reg32 !_rcx _rax;
(*#movl %eax, -4(%rbp)*)
 move_reg32 !_rax _4__rbp_;
assert{ integer_of_int32(sel_int32(_4__rbp_)) > 0};
assert{ integer_of_int32(sel_int32(_4__rbp_)) <= 100};
(*#movl -4(%rbp), %eax*)
 move_reg32 !_4__rbp_ _rax;
(*#leave*)
(*#ret*)
void
```

Figure 7.5: Result of Figure 7.4 program

```
/*@ requires 0 <= x <= 1000 ;
  @ ensures \result == x * x;*/
int square(int x){
  int tmp = x * x;
  return tmp;
}

int main(){
  int a = 5;
  int b = square(a);
  //@ assert b== 25;

  return 0;
}
```

Figure 7.6: Square program

```
parameter square_parameter: _:unit->
{ 0 <= integer_of_int32(sel_int32(_rdi)) <= 1000 }
 unit reads _rdi
{ integer_of_int32(sel_int32(_rax)) =
        integer_of_int32(sel_int32(_rdi))*integer_of_int32(sel_int32(_rdi)) }

let square_0() =
_LFB0:
(*#movl %edi, -20(%rbp)*)
 move_reg32 !_rdi _20__rbp_;
[{  }
   unit reads _20__rbp_
 { 0 <= integer_of_int32(sel_int32(_20__rbp_)) <= 1000 }];
(*#movl -20(%rbp), %eax*)
 move_reg32 !_20__rbp_ _rax;
(*#imull        -20(%rbp), %eax*)
 set_reg32 ((integer_of_int32 (sel_int32 !_rax))*
             (integer_of_int32 (sel_int32 !_20__rbp_))) _rax;
(*#movl %eax, -8(%rbp)*)
 move_reg32 !_rax _8__rbp_;
(*#movl -8(%rbp), %eax*)
 move_reg32 !_8__rbp_ _rax;
(*#movl %eax, -4(%rbp)*)
 move_reg32 !_rax _4__rbp_;
_L2:
assert{ integer_of_int32(sel_int32(_4__rbp_)) =
            integer_of_int32(sel_int32(_20__rbp_)) *
            integer_of_int32(sel_int32(_20__rbp_)) };
(*#movl -4(%rbp), %eax*)
 move_reg32 !_4__rbp_ _rax;
(*#ret*)
 void


let main_0() =
_LFB1:
(*#movl $5, -12(%rbp)*)
 move_cte32 (5) (5.0) _12__rbp_;
(*#movl -12(%rbp), %eax*)
 move_reg32 !_12__rbp_ _rax;
(*#movl %eax, %edi*)
 move_reg32 !_rax _rdi;
(*#call square*)
 square_parameter(_);
(*#movl %eax, -8(%rbp)*)
 move_reg32 !_rax _8__rbp_;
assert{ integer_of_int32(sel_int32(_8__rbp_)) = 25 };
(*#movl $0, -4(%rbp)*)
 move_cte32 (0) (0.0) _4__rbp_;
_L4:
(*#movl -4(%rbp), %eax*)
 move_reg32 !_4__rbp_ _rax;
 void
```

Figure 7.7: Why program of Figure 6.7

Proof obligations	Alt-Ergo 0.92.2	CVC3 2.2 (SS)	Gappa 0.12.1	Statistics
▸ function square_0 Correctness	✓	✓	✓	2/2
▾ function main_0 Correctness	✓	✓	✓	3/3
1. precondition	✓	✓	✓	
2. precondition	✓	✓	✓	
3. assertion	✓	✓	✓	

```
H5: 0 <= integer_of_int32(sel_int32(_rdi)) &&
    integer_of_int32(sel_int32(_rdi)) <= 1000
H6: integer_of_int32(sel_int32(_rax)) == integer_of_int32
(sel_int32(_rdi)) *
    integer_of_int32(sel_int32(_rdi))
_8__rbp_: register
H7: integer_of_int32(sel_int32(_8__rbp_)) ==
    integer_of_int32(sel_int32(_rax)) &&
    single_value(sel_single(_8__rbp_)) == single_value
(sel_single(_rax)) &&
    sel_exact(_8__rbp_) == sel_exact(_rax)
_______________

integer_of_int32(sel_int32(_8__rbp_)) == 25
```

```
(*#movl $5, -12(%rbp)*)
 move_cte32 (5) (5.0) _12__rbp_;
(*#movl -12(%rbp), %eax*)
 move_reg32 !_12__rbp_ _rax;
(*#movl %eax, %edi*)
 move_reg32 !_rax _rdi;
(*#call square*)
 square_parameter(_);
(*#movl %eax, -8(%rbp)*)
 move_reg32 !_rax _8__rbp_;
assert{ integer_of_int32(sel_int32(_8__rbp_))= 25
};
```

Figure 7.8: Result of Figure 7.7 program

Chapter 8

Floating-point programs

Chapter 7 talks about the translation of the program containing only 32-bit and 64-bit integer type. In this chapter, we will extend it with the floating-point computations.

8.1 Assembly with floating-point arithmetic

Before entering into the translation, we give some basic knowledge about the different modes: SSE/SSE2, x87 and FMA and their instructions [2, 43, 44, 45].

8.1.1 SSE/SSE2

The Intel MMX (MultiMedia eXtensions) technology introduced single-instruction multiple-data (SIMD) capacity into the IA-32 architecture, with the 64-bit `mmx` registers, 64-bit packed integer data types, and instructions that allowed SIMD operations to be performed on packed integers. SSE extensions expand the SIMD execution model by adding facilities for handling packed and scalar single-precision floating-point value contained in 128-bit registers.

The extension SSE2 is a major enhancement to SSE. It adds new math instructions for double-precision (64-bit) floating-point and also extends `mmx` instructions to operate on 128-bit `xmm` registers.

Data Transfer Instruction

movsd xmm1 xmm2/m64	Move scalar double-precision floating-point value from `xmm1` register to `xmm2/m64`
movsd xmm2/m64 xmm1	Move scalar double-precision floating-point value from `xmm2/m64` to `xmm1` register
movss xmm1 xmm2/m32	Move scalar single-precision floating-point value from `xmm1` register to `xmm2/m32`
movss xmm2/m32 xmm1	Move scalar single-precision floating-point value from `xmm2/m32` to `xmm1` register

These instructions move a scalar double-precision (single-precision) floating-point value from the source operand (first operand) to the destination operand (second operand). The source and destination operands can be `xmm` registers or 64-bit (32-bit) memory locations.

Packed Arithmetic Instructions

addsd xmm2/m64, xmm1	Add the low double-precision floating-point value from `xmm2/m64` to `xmm1`
addss xmm2/m32, xmm1	Add the low single-precision floating-point value from `xmm2/m32` to `xmm1`
subsd xmm2/m64, xmm1	Subtracts the low double-precision floating-point values in `xmm2/mem64` from `xmm1`
subss xmm2/m32, xmm1	Subtracts the low single-precision floating-point values in `xmm2/mem32` from `xmm1`
mulsd xmm2/m64, xmm1	Multiply the low double-precision floating-point value in `xmm2/mem64` by low double-precision floating-point value in `xmm1`
mulss xmm2/m32, xmm1	Multiply the low single-precision floating-point value in `xmm2/mem32` by low single-precision floating-point value in `xmm1`
divsd xmm2/m64, xmm1	Divide low double-precision floating-point value in `xmm1` by low double-precision floating-point value in `xmm2/mem64`
divss xmm2/m32, xmm1	Divide low single-precision floating-point value in `xmm1` by low single-precision floating-point value in `xmm2/mem32`

Comparison Instructions

comisd xmm2/m64, xmm1
comiss xmm2/m32, xmm1
ucomisd xmm2/m64, xmm1
ucomiss xmm2/m32, xmm1

The `comisd` and `comiss` instructions compare the double-precision (single-precision) floating-point values in the low quadwords (doublewords) of first operand and second operand, and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The unordered result is returned if either source operand is a NaN. The OF, SF and AF flags in the EFLAGS register are set to 0.

The `comisd` instruction differs from the `ucomisd` instruction in that it signals a SIMD floating-point invalid operation exception when a source operand is either a qNaN or sNaN. The `ucomisd` instruction signals an invalid numeric exception only if a source operand is an sNaN.

8.1.2 x87 Floating-point Unit

The x87 floating-point unit (FPU) instructions are executed by the processor's x87 FPU. These instructions operate on floating-point, integer and binary-coded decimal(BCD) operands.

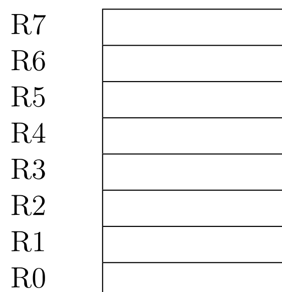
x87 FPU registers

This FPU provides several registers. These registers are divided into three groups: data registers, control and status registers, and pointer registers.

The x87 FPU status register has 16 bits. It indicates the current state of the x87 FPU. The flags in the x87 FPU status register include the FPU busy flag, top-of-stack (TOS) pointer, condition code flags, error summary status flag, stack fault flag, and exception flags.

The x87 FPU has 8 floating-point registers to hold the floating-point operands. These registers supply the necessary operands to the floating-point instructions. Unlike the processor's

general-purpose registers such as the `eax` and `ebx` registers, these registers are organized as a register stack.



Since these registers are organized as a register stack, these names are not statically assigned. That is, `st(0)` does not refer to a specific register. It refers to whichever register is acting as the top-of-stack (TOS) register. The next register is referred to as `st(1)`, and so on; the last register as `st(7)`. There is a 3-bit top-of-stack pointer in the status register to identify the TOS register. For example, if TOS points to `R5` then `st(0)` indicates `R5`, `st(1)` indicates `R6`, etc. Load operations decrement TOS by one and load a value into the new top-of-stack register, and store operations store the value from the current TOS register in memory and then increment TOS by one.

Each data register can hold an extended-precision floating-point number. It uses the 80 bits format that we mentioned in Chapter 2.

x87 FPU instructions

Most floating-point instructions require one or two operands, located on the x87 FPU data-register stack or in memory. When an operand is located in a data register, is referenced relative to the `st(0)` register, rather than by a physical register name. Often the `st(0)` is an implied operand.

These instructions are divided into the following groups: data transfer, load constants, and FPU control instructions.

Data Transfer Instructions The data transfer instructions perform the following operations:

- Load a floating-point, integer, or packed BCD operand from memory into the `st(0)` register.
- Store the value in an `st(0)` register to memory in floating-point, integer, or packed BCD format.
- Move values between registers in the x87 FPU register stack.

Load Constant Instructions The following instructions push commonly used constants onto the top `st(0)` of the x87 FPU register stack:

fldz	st(0) $\leftarrow +0.0$
fld1	st(0) $\leftarrow +1.0$
fldpi	st(0) $\leftarrow \circ_{80}(\pi)$
fldl2t	st(0) $\leftarrow \circ_{80}(log_2 10)$
fldl2e	st(0) $\leftarrow \circ_{80}(log_2 e)$
fldlg2	st(0) $\leftarrow \circ_{80}(log_{10} 2)$
fldln2	st(0) $\leftarrow \circ_{80}(log_e 2)$

Each load instruction in the table above follows TOS ← TOS - 1.

Basic Arithmetic Instructions The addition, subtraction, multiplication and division instructions operate on the following types of operands:

- Two x87 FPU data registers
- An x87 FPU data register and a floating-point or integer value in memory

These are the floating-point instructions that perform basic arithmetic operations on floating-point numbers:

fadd src	st(0) ← st(0) + src
fadd src, dest	dest ← dest + src
faddp src, dest	dest ← dest + src TOS ← TOS + 1
fsub src	st(0) ← st(0) - src
fsub src, dest	dest ← dest - src
fsubp src, dest	dest ← dest - src TOS ← TOS + 1
fsubr src	st(0) ← src - st(0)
fsubr src, dest	dest ← src - dest
fsubrp src, dest	dest ← src - dest TOS ← TOS + 1
fmul src	st(0) ← st(0) * src
fmul src, dest	dest ← dest * src
fmulp src, dest	dest ← dest * src TOS ← TOS + 1
fdiv src	st(0) ← st(0) / src
fdiv src, dest	dest ← dest / src
fdivp src, dest	dest ← dest / src TOS ← TOS + 1
fdivr src	st(0) ← src / st(0)
fdivr src, dest	dest ← src / dest
fdivrp src, dest	dest ← src / dest TOS ← TOS + 1

The `src` operand of instruction with one operand can be either 32- or 64-bit floating-point number in memory. With instructions with two operands `src` and `dest`, both `src` and `dest` must be FPU registers.

8.1.3 FMA

`gcc` uses AVX [1] instructions when generating assembly code with option `-mfma4`. Before talking about FMA instructions we will present some AVX instructions [3].

[1] Advanced Vector Extensions (AVX) is an extension to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel in March 2008.

AVX arithmetic instructions

vaddss xmm3/mem32, xmm2, xmm1	Add Scalar Single-Precision floating-point
vaddsd xmm3/mem64, xmm2, xmm1	Add Scalar Double-Precision floating-point
vsubss xmm3/mem32, xmm2, xmm1	Subtract Scalar Single-Precision floating-point
vsubsd xmm3/mem64, xmm2, xmm1	Subtract Scalar Double-Precision floating-point
vmulss xmm3/mem32, xmm2, xmm1	Multiply Scalar Single-Precision floating-point
vmulsd xmm3/mem64, xmm2, xmm1	Multiply Scalar Double-Precision floating-point
vdivss xmm3/mem32, xmm2, xmm1	Divide Scalar Single-Precision floating-point
vdivsd xmm3/mem64, xmm2, xmm1	Divide Scalar Double-Precision floating-point

The first source operand is either an `xmm` register or a 64-bit memory location and the second source operand is a `xmm` register. The destination is a third `xmm` register. Bits [127:64] of the second source operand are copied to bits [127:64] of the destination. Bits [255:128] of the `ymm` register that correspond to the destination are cleared.

FMA instructions

Now both AMD and Intel have specifications for FMA. In this document, FMA instructions generated thanks to `gcc -mfma4` are specified by AMD [2].

vfmaddss src3, src2, src1, dest	dest = $\circ_{32}$(src1 * src2 + src3)
vfmaddsd src3, src2, src1, dest	dest = $\circ_{64}$(src1 * src2 + src3)
vfmsubss src3, src2, src1, dest	dest = $\circ_{32}$(src1 * src2 - src3)
vfmsubsd src3, src2, src1, dest	dest = $\circ_{64}$(src1 * src2 - src3)
vfnmaddss src3, src2, src1, dest	dest = $\circ_{32}$(-(src1 * src2) + src3)
vfnmaddsd src3, src2, src1, dest	dest = $\circ_{64}$(-(src1 * src2) + src3)
vfnmsubss src3, src2, src1, dest	dest = $\circ_{32}$(-(src1 * src2) - src3)
vfnmsubsd src3, src2, src1, dest	dest = $\circ_{64}$(-(src1 * src2) - src3)

The implementation of `vfmaddsd` is presented in Figure 8.1. The destination is a `xmm` register. When the result is written to the destination `xmm` register, the upper quadword of the destination register (bits 64 – 127) and the upper 128-bits of the corresponding `ymm` register are cleared to zeros. The intermediate product is not rounded; the infinitely precise product is used in the addition. The result of the addition is rounded.

The implementation of `vfnmaddss` is in Figure 8.2. The destination is always a `xmm` register. When the result is written to the destination `xmm` register, the upper three doublewords of the destination register (bits 32 – 127) and the upper 128-bits of the corresponding `ymm` register are cleared to zeros. The intermediate products are not rounded; the infinitely precise products are used in the addition. The results of the addition are rounded.

8.2 Definition of programs supported

This chapter is an extension of the previous one. The programs concerned are the ones in which the types `int`, `long int`, `float double` and `long double` are supported. An interesting point of this chapter is that we will show the different results obtained by compiling a floating-point program with different options of compiler and different architectures.

The hypothesis of the assembly programs handled in this chapter are

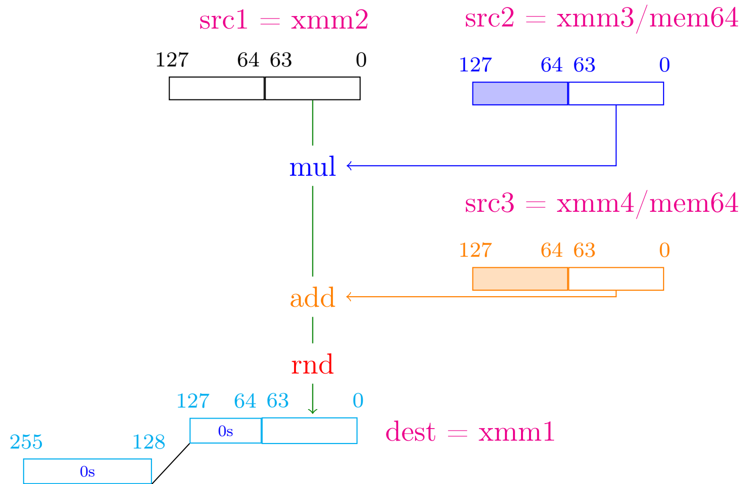


Figure 8.1: Illustration of `vfmaddsd` instruction

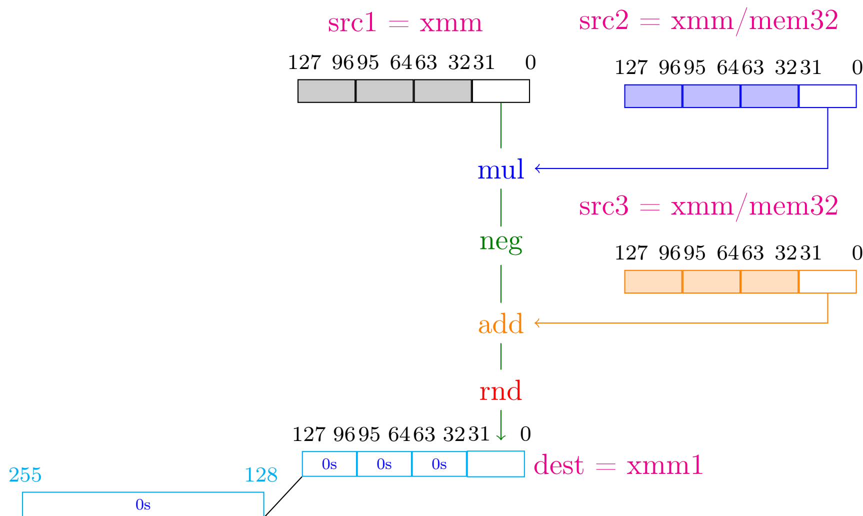


Figure 8.2: Illustration of VFNMADDSS instruction

- We do not consider special values. We only verify the absence of overflow and use only round-to-nearest rounding mode.
- For SSE/SSE2 mode, there is no difference between `ucomiss`/`comiss` and `ucomisd`/`comisd`.
- For x87 mode:
  - We translate only floating-point operations, the integer instructions (fimul, fiadd, etc.) will not be considered here.
  - We consider the case where the stack is empty at the entrance and the exit of a function (the top-of-stack pointer will point to the top register of the stack).

8.3 Translation to Why

8.3.1 Abstract functions

The abstract types `single`, `double` had already defined in the strict model and full mode and we also used them in Part 1. In the same way, we define abstract type `binary80` for x87 mode.

Similar to the functions `sel_int32` and `sel_int64` of Chapter 7, we declare the function for getting the content of a register:

```
logic sel_single : register -> single
logic sel_double : register -> double
logic sel_80     : register -> binary80
logic sel_exact  : register -> real
```

The logic functions `sel_single`, `sel_double`, `sel_80` and `sel_exact` return a `single`, `double`, `binary80` and an exact value, respectively, from a register.

We denote by *opr* an operand being register or memory reference; by $[\![opr]\!]_{int32}$,$[\![opr]\!]_{single}$, $[\![opr]\!]_{double}$ and $[\![opr]\!]_{binary80}$ the interpretation of an operand that returns a 32-bit integer value, a real in 32 bits, 64 bits and 80 bits respectively from a *register*. The translation of operands is specified as follows:

$[\![opr]\!]_{single}$ = `(single_value (sel_single !opr))`
$[\![opr]\!]_{double}$ = `(double_value (sel_double !opr))`
$[\![opr]\!]_{binary80}$ = `(binary80_value (sel_80 !opr))`
$[\![opr]\!]_{exact}$ = `(sel_exact !opr)`

$[\![symbol]\!]_{single}$ = `(single_value (sel_single symbol))`
$[\![symbol]\!]_{double}$ = `(double_value (sel_double symbol))`
$[\![symbol]\!]_{binary80}$ = `(binary80_value (sel_80 symbol))`
$[\![symbol]\!]_{exact}$ = `(sel_exact symbol)`

8.3.2 When constants are referenced by `%rip`

As specified in Intel document [43], bytes, words and doublewords in the packed data types are stored in consecutive addresses. The least significant byte, word, or doubleword is stored at the lowest address and the most significant byte, word, or doubleword is stored at the high address. The ordering of bytes, words, or doublewords in memory is always little endian. That is, the bytes with the low addresses are less significant than the bytes with high addresses.

In assembly language, a floating-point constant is often declared in data section and it is referenced by the special register `%rip`. For example:

```
.LC1:
    .long 0
    .long 1025507328
```

When a value is stored in memory, we do not know its type. In order to point out the values in Why, we define an axiom in which we presuppose all the types that we can access. Moreover, depending on the number of part stored in the memory, we will determine the size of the number pointed by the symbol.

With the example above, there are two 32-bit number following the symbol `.LC1`. Assume that the little endian architecture is used, we will consider that from `.LC1(%rip)`, we can get both 32-bit and 64-bit numbers. Thus, in our axiom, we consider both forms: 32-bit integer and `single`; 64-bit integer and `double` as follow:

```
logic _LC1__rip_: register
axiom _LC1__rip__axiom:
    integer_of_int64(sel_int64(_LC1__rip_)) = 4404520435568345088
    and
    integer_of_int32(sel_int32(_LC1__rip_)) = 0
    and
    double_value(sel_double(_LC1__rip_))= 0x1p-45
    and
    single_value(sel_single(_LC1__rip_))= 0.0
```

Note that the integer value 4404520435568345088 is obtained by 1025507328 $\ll$ 32 (lower part is 0) and this value is converted to `double`, that is 2^{-45}.

8.3.3 Modifying the translation of general-purpose instructions

As we have mentioned before, data transfer instructions copy data from one operand to another without knowing its type. For example, `movl` is a data transfer instruction but we do not know whether it transfers a 32-bit integer or a 32-bit floating-point number (`float` type in C).

In Chapter 7, with the simple program containing only integer value, the instructions `movl` and `movq` are considered to copy 32-bit and 64-bit integer value from source to destination. As in programs containing floating-point computations, the data of the `mov` instructions here may be a floating-point value and the translation of `mov` in Chapter 7 is not enough anymore.

Our idea is that we define a parameter `move_cte32` which copies at the same time a 32-bit integer value, a single value and its exact value to `dest` and a parameter `move_cte64` which copies a 64-bit integer value, a double value and its exact value to `dest`. We need to assure here that the integer value and the floating-point value are interpreted from the same bitvector. We cannot express this in Why but we did it directly in our translator when generating Why program.

```
parameter move_cte32:a:int-> b:real->bexact:real->c:register ref->
{ }
  unit writes c
{ integer_of_int32(sel_int32(c)) = a
  and
  single_value(sel_single(c)) = b
  and
  sel_exact(c) = bexact }

parameter move_cte64:a:int->b:real->bexact:real->c:register ref->
{ }
  unit writes c
{ integer_of_int64(sel_int64(c)) = a
  and
  double_value(sel_doule(c)) = b
  and
  sel_exact(c) = bexact }
```

The translation of `movl` and `movq` is shown as below:

$$\begin{array}{lcl} [\![\text{ movl src, dest }]\!]_i & = & \texttt{move_cte32}\ [\![src]\!]_{int32}\ [\![src]\!]_{single}\ [\![src]\!]_{exact}\ \textit{dest} \\ [\![\text{ movq src, dest }]\!]_i & = & \texttt{move_cte64}\ [\![src]\!]_{int64}\ [\![src]\!]_{double}\ [\![src]\!]_{exact}\ \textit{dest} \end{array}$$

8.3.4 Translation of SSE/SSE2 instructions

We need to set a floating-point value to a `register`. To do that, we define the following parameter functions:

```
parameter set_single_no_check: a:real -> aexact:real -> b:register ref ->
{ }
  unit writes b
{ single_value(sel_single(b)) = a
  and
  sel_exact(b) = aexact }

parameter set_single: a:real -> aexact:real -> b:register ref ->
{ no_overflow_single(nearest_even,a) }
  unit writes b
{ single_value(sel_single(b)) = round_single(nearest_even,a)
  and
  sel_exact(b) = aexact }
```

Each parameter has three arguments: the real value, the exact value and the register to store. Setting a `single` has two cases:

- Case 1: We do not need to check if the input value is overflow or not. We use it when transferring data from `src` to `dest` in the `movss` instructions. Pay attention that in this case, the value `a` is not rounded because we already know that it is a 32-bit floating-point number.
- Case 2: We have to check the input value. This parameter is used when we set a value of a computation (addition, subtraction, etc.) to a `register`.

We do similarly with `double` floating-point value.

```
parameter set_double_no_check: a:real -> aexact:real -> b:register ref ->
{ }
  unit writes b
{ double_value(sel_double(b)) = a
  and
  sel_exact(b) = aexact }

parameter set_double: a:real -> aexact:real -> b:register ref ->
{ no_overflow_double(nearest_even,a) }
unit writes b
{ double_value(sel_double(b)) = round_double(nearest_even,a)
  and
  sel_exact(b) = aexact }
```

Division is a special case. We ensure that the divisor is not equal to 0.

```
parameter div_single: a:register -> b:register -> c:register ref ->
{ single_value(sel_single(a))<>0
  and
  no_overflow_single(nearest_even,
          single_value(sel_single(b))/single_value(sel_single(a))) }
```

```
  unit writes c
{ single_value(sel_single(c)) = round_single(nearest_even,
           single_value(sel_single(b))/single_value(sel_single(a)))
  and
  sel_exact(c) = sel_exact(b)/sel_exact(a) }

parameter div_double: a:register -> b:register -> c:register ref ->
{ double_value(sel_double(a))<>0
  and
  no_overflow_double(nearest_even,
           double_value(sel_double(b))/double_value(sel_double(a))) }
  unit writes c
{ double_value(sel_double(c)) = round_double(nearest_even,
           double_value(sel_double(b))/double_value(sel_double(a)))
  and
  sel_exact(c) = sel_exact(b)/sel_exact(a) }
```

The translation of data instructions and arithmetic ones are presented as follows:

Data Transfer Instructions

$[\![$ movsd src, dest $]\!]_i$ = `set_double_no_check` $[\![src]\!]_{double}$ $[\![src]\!]_{exact}$ *dest*
$[\![$ movss src, dest $]\!]_i$ = `set_single_no_check` $[\![src]\!]_{single}$ $[\![src]\!]_{exact}$ *dest*

Arithmetic Instructions

$[\![$ addsd src, dest $]\!]_i$ = `set_double` ($[\![src]\!]_{double}$+$[\![dest]\!]_{double}$) ($[\![src]\!]_{exact}$+$[\![dest]\!]_{exact}$) *dest*
$[\![$ addss src, dest $]\!]_i$ = `set_single` ($[\![src]\!]_{single}$+$[\![dest]\!]_{single}$) ($[\![src]\!]_{exact}$+$[\![dest]\!]_{exact}$) *dest*

$[\![$ subsd src, dest $]\!]_i$ = `set_double` ($[\![dest]\!]_{double}$-$[\![src]\!]_{double}$) ($[\![dest]\!]_{exact}$-$[\![src]\!]_{exact}$) *dest*
$[\![$ subss src, dest $]\!]_i$ = `set_single` ($[\![dest]\!]_{single}$-$[\![src]\!]_{single}$) ($[\![dest]\!]_{exact}$-$[\![src]\!]_{exact}$) *dest*

$[\![$ mulsd src, dest $]\!]_i$ = `set_double` ($[\![dest]\!]_{double}$*$[\![src]\!]_{double}$) ($[\![dest]\!]_{exact}$*$[\![src]\!]_{exact}$) *dest*
$[\![$ mulss src, dest $]\!]_i$ = `set_single` ($[\![dest]\!]_{single}$*$[\![src]\!]_{single}$) ($[\![dest]\!]_{exact}$*$[\![src]\!]_{exact}$) *dest*

$[\![$ divsd src, dest $]\!]_i$ = `div_double !src !dest dest`
$[\![$ divss src, dest $]\!]_i$ = `div_single !src !dest dest`

It is needed to say that if an instruction `movq a b` (`movl a b`) is followed by an instruction `movsd b c` (`movss b c`) then in the Why program, the condition `integer_of_int64(sel_int(b))` (`integer_of_int32(sel_int32(b))`) is not used. It is just redundant information. This value is only used when the following instruction is a general-purpose one.

8.3.5 x87 Floating-point Unit

Representation of the stack in Why

As we mentioned in 8.1.2, the stack has eight floating-point registers (R0 – R7) to hold the floating-point operands. There is a top-of-stack (TOS) pointer which identifies the TOS register.

To represent the stack, one solution is to use a Why array of type `register`. In order to prove floating-point programs, we use mostly Gappa tool. As Gappa can not use the axioms declared for Why array, we cannot prove Why programs by Gappa in x87 mode.

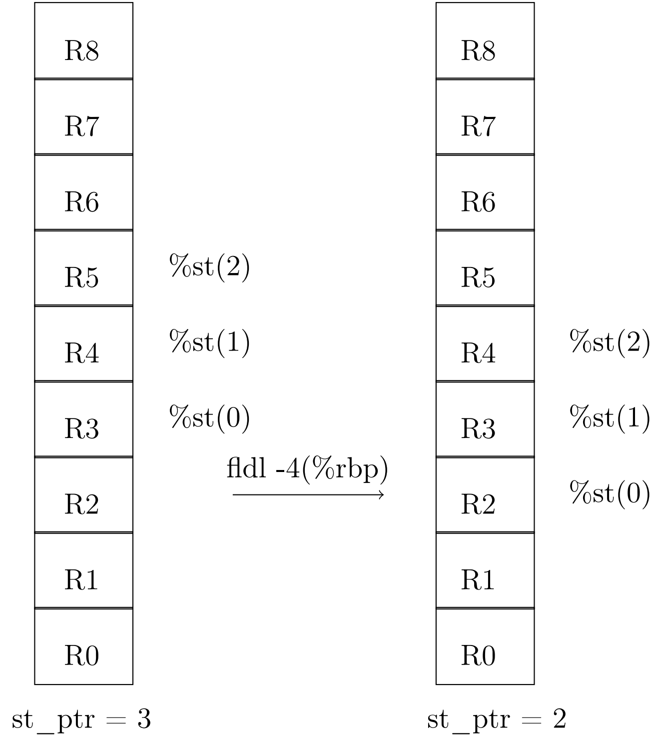


Figure 8.3: Illustration of the stack with instruction *fldl*

We propose another solution which helps Gappa to prove Why programs in x87 mode: instead of using Why array, we define eight variables of type `registers`: `st0`, ..., `st7` to represent for the registers R0–R7 of the stack. When analyzing assembly program, we use a temporary variable *st_ptr* which simulates the top-of-stack TOS and translate "relative" stack registers to their physical ones. We insist that this temporary variable is not a variable in Why program, it is in our translator. In the generated Why program, there will be only physical stack registers.

For translating from "relative" stack registers into physical ones, we set initial value of *st_ptr* as 8. Before each load instruction is executed, *st_ptr* decreases by 1. Otherwise, if the stack pops a value (for example `fstp`, `faddp`, ect.), *st_ptr* increases by 1 after this instruction is executed. The value of `st_ptr` is $0 \leq st_ptr \leq 8$.

For illustration, Figure 8.3 shows us the changes of the stack after executing the instruction `fldl`. Assume that the current value of `st_ptr` is `st_ptr = 3`. This means that in Why, `st3` is on the top of the stack. If we access `%st(1)`, it will be `st4` in Why. After executing the instruction `fldl`, the value of `st_ptr` decreases by 1 and its value is `st_ptr = 2` and now `st2` is on the top of the stack. If we want to access `%st(i)` then the physical register will be `st(st_ptr + i)`.

We denote by $[\![st]\!]_{st}$ the translation of *st* to Why. The translation of the operand – when the stack is used – is specified as follows:

$$
\begin{array}{lcl}
[\![st]\!]_{st} & = & \texttt{st}\bar{i} \text{ where } \bar{i} = st_ptr \\
[\![st(i)]\!]_{st} & = & \texttt{st}\bar{i} \text{ where } \bar{i} = st_ptr + i \text{ and } 0 \leq st_ptr + i \leq 7
\end{array}
$$

Translation of instructions to Why

Similarly to the parameters for `single` and `double` types that we defined in Section 8.3.4, here are the parameters for `binary80` type.

```
parameter set_80_no_check: a:real -> aexact:real -> b:register ref ->
{ }
  unit writes b
{ binary80_value(sel_binary80(b)) = a
  and
  sel_exact(b) = aexact }

parameter set_80: a:real -> aexact:real -> b:register ref ->
{ no_overflow_binary80(\nearest_even,a) }
  unit writes b
{ binary80_value(sel_binary80(b)) = round_binary80(nearest_even,a)
  and
  sel_exact(b) = aexact }
```

The interpretation of instructions is presented as follows:

Data transfer instructions

$[\![$ flds src $]\!]_i$ = `set_80` $[\![src]\!]_{single}$ $[\![src]\!]_{exact}$ `st0̄`
$[\![$ fldl src $]\!]_i$ = `set_80` $[\![src]\!]_{double}$ $[\![src]\!]_{exact}$ `st0̄`

$[\![$ fldlz $]\!]_i$ = `set_80 (0.0) (0.0) st0̄`
$[\![$ fldl1 $]\!]_i$ = `set_80 (1.0) (1.0) st0̄`

$[\![$ fsts dest $]\!]_i$ = `set_single` $[\![st\bar{0}]\!]_{binary80}$ $[\![st\bar{0}]\!]_{exact}$ *dest*
$[\![$ fstl dest $]\!]_i$ = `set_double` $[\![st\bar{0}]\!]_{binary80}$ $[\![st\bar{0}]\!]_{exact}$ *dest*

Arithmetic instructions

As before, the division is a special case as we have to verify in the precondition that the divisor is non-zero. The parameters for the division operation is declared below:

```
parameter div_80: a:real -> aexact:real -> b:register ref ->
{ a<>0
  and
  no_overflow_binary80(nearest_even, binary80_value(sel_80(b))/a) }
  unit writes b
{ binary80_value(sel_80(b)) = round_binary80(nearest_even,
              binary80_value(sel_80(b@))/a)
  and
  sel_exact(b) = sel_exact(b@)/aexact }

parameter divr_80: a:real -> aexact:real-> b:register ref ->
{ binary80_value(sel_80(b))<>0.0
  and
  no_overflow_binary80(nearest_even,
              a/binary80_value(sel_80(b))) }
  unit writes b
```

```
{ binary80_value(sel_80(b)) = round_binary80(nearest_even,
             a/binary80_value(sel_80(b@)))
  and
  sel_exact(b) = aexact/sel_exact(b@) }
```

The translation of some arithmetic instructions is specified as follows:

$\llbracket$ fadds src $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{0} \rrbracket_{binary80}$ + $\llbracket src \rrbracket_{single}$) ($\llbracket st\bar{0} \rrbracket_{exact}$ + $\llbracket src \rrbracket_{exact}$) $st\bar{0}$
$\llbracket$ faddl src $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{0} \rrbracket_{binary80}$ + $\llbracket src \rrbracket_{double}$) ($\llbracket st\bar{0} \rrbracket_{exact}$ + $\llbracket src \rrbracket_{exact}$) $st\bar{0}$
$\llbracket$ fsubs src $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{0} \rrbracket_{binary80}$ - $\llbracket src \rrbracket_{single}$) ($\llbracket st\bar{0} \rrbracket_{exact}$ - $\llbracket src \rrbracket_{exact}$) $st\bar{0}$
$\llbracket$ fsubl src $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{0} \rrbracket_{binary80}$ - $\llbracket src \rrbracket_{double}$) ($\llbracket st\bar{0} \rrbracket_{exact}$ - $\llbracket src \rrbracket_{exact}$) $st\bar{0}$
$\llbracket$ fsubrs src $\rrbracket_i$	$=$ `set_80` ($\llbracket src \rrbracket_{single}$ - $\llbracket st\bar{0} \rrbracket_{binary80}$) ($\llbracket src \rrbracket_{exact}$ - $\llbracket st\bar{0} \rrbracket_{exact}$) $st\bar{0}$
$\llbracket$ fsubrl src $\rrbracket_i$	$=$ `set_80` ($\llbracket src \rrbracket_{double}$ - $\llbracket st\bar{0} \rrbracket_{binary80}$) ($\llbracket src \rrbracket_{exact}$ - $\llbracket st\bar{0} \rrbracket_{exact}$) $st\bar{0}$
$\llbracket$ fmuls src $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{0} \rrbracket_{binary80}$ $*$ $\llbracket src \rrbracket_{single}$) ($\llbracket st\bar{0} \rrbracket_{exact}$ $*$ $\llbracket src \rrbracket_{exact}$) $st\bar{0}$
$\llbracket$ fmull src $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{0} \rrbracket_{binary80}$ $*$ $\llbracket src \rrbracket_{double}$) ($\llbracket st\bar{0} \rrbracket_{exact}$ $*$ $\llbracket src \rrbracket_{exact}$) $st\bar{0}$
$\llbracket$ fdivs src $\rrbracket_i$	$=$ `div_80` $\llbracket src \rrbracket_{single}$ $\llbracket src \rrbracket_{exact}$ $st\bar{0}$
$\llbracket$ fdivl src $\rrbracket_i$	$=$ `div_80` $\llbracket src \rrbracket_{double}$ $\llbracket src \rrbracket_{exact}$ $st\bar{0}$
$\llbracket$ fdivrs src $\rrbracket_i$	$=$ `divr_80` $\llbracket src \rrbracket_{single}$ $\llbracket src \rrbracket_{exact}$ $st\bar{0}$
$\llbracket$ fdivrl src $\rrbracket_i$	$=$ `divr_80` $\llbracket src \rrbracket_{double}$ $\llbracket src \rrbracket_{exact}$ $st\bar{0}$
$\llbracket$ fadd %st(i), %st(j) $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{j} \rrbracket_{binary80}$ + $\llbracket st\bar{i} \rrbracket_{binary80}$) ($\llbracket st\bar{j} \rrbracket_{exact}$ + $\llbracket st\bar{i} \rrbracket_{exact}$) $st\bar{j}$
$\llbracket$ faddp %st(i), %st(j) $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{j} \rrbracket_{binary80}$ + $\llbracket st\bar{i} \rrbracket_{binary80}$) ($\llbracket st\bar{j} \rrbracket_{exact}$ + $\llbracket st\bar{i} \rrbracket_{exact}$) $st\bar{j}$
$\llbracket$ fsub %st(i), %st(j) $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{j} \rrbracket_{binary80}$ - $\llbracket st\bar{i} \rrbracket_{binary80}$) ($\llbracket st\bar{j} \rrbracket_{exact}$ - $\llbracket st\bar{i} \rrbracket_{exact}$) $st\bar{j}$
$\llbracket$ fsubp %st(i), %st(j) $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{j} \rrbracket_{binary80}$ - $\llbracket st\bar{i} \rrbracket_{binary80}$) ($\llbracket st\bar{j} \rrbracket_{exact}$ - $\llbracket st\bar{i} \rrbracket_{exact}$) $st\bar{j}$
$\llbracket$ fsubr %st(i), %st(j) $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{i} \rrbracket_{binary80}$ - $\llbracket st\bar{j} \rrbracket_{binary80}$) ($\llbracket st\bar{i} \rrbracket_{exact}$ - $\llbracket st\bar{j} \rrbracket_{exact}$) $st\bar{j}$
$\llbracket$ fsubrp %st(i), %st(j) $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{i} \rrbracket_{binary80}$ - $\llbracket st\bar{j} \rrbracket_{binary80}$) ($\llbracket st\bar{i} \rrbracket_{exact}$ - $\llbracket st\bar{j} \rrbracket_{exact}$) $st\bar{j}$
$\llbracket$ fmul %st(i), %st(j) $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{j} \rrbracket_{binary80}$ $*$ $\llbracket st\bar{i} \rrbracket_{binary80}$) ($\llbracket st\bar{j} \rrbracket_{exact}$ $*$ $\llbracket st\bar{i} \rrbracket_{exact}$) $st\bar{j}$
$\llbracket$ fmulp %st(i), %st(j) $\rrbracket_i$	$=$ `set_80` ($\llbracket st\bar{j} \rrbracket_{binary80}$ $*$ $\llbracket st\bar{i} \rrbracket_{binary80}$) ($\llbracket st\bar{j} \rrbracket_{exact}$ $*$ $\llbracket st\bar{i} \rrbracket_{exact}$) $st\bar{j}$
$\llbracket$ fdiv %st(i), %st(j) $\rrbracket_i$	$=$ `div_80` $\llbracket st\bar{i} \rrbracket_{binary80}$ $\llbracket st\bar{i} \rrbracket_{exact}$ $st\bar{j}$
$\llbracket$ fdivp %st(i), %st(j) $\rrbracket_i$	$=$ `div_80` $\llbracket st\bar{i} \rrbracket_{binary80}$ $\llbracket st\bar{i} \rrbracket_{exact}$ $st\bar{j}$
$\llbracket$ fdivr %st(i), %st(j) $\rrbracket_i$	$=$ `divr_80` $\llbracket st\bar{i} \rrbracket_{binary80}$ $\llbracket st\bar{i} \rrbracket_{exact}$ $st\bar{j}$
$\llbracket$ fdivrp %st(i), %st(j) $\rrbracket_i$	$=$ `divr_80` $\llbracket st\bar{i} \rrbracket_{binary80}$ $\llbracket st\bar{i} \rrbracket_{exact}$ $st\bar{j}$

8.3.6 AVX instructions

There are AVX instructions that are not FMA ones but they are generated when a program is compiled with `gcc -mfma4`. The translation of them is described in the following table. Indeed, the specification of these instructions are not different from the instructions in SSE/SSE2. The difference is that they have three operands and have the prefix 'v'. Here are the translations of some AVX instructions:

$\llbracket$ vaddss src2, src1, dest $\rrbracket_i$	$=$	`set_single`	($\llbracket src1 \rrbracket_{single}$ + $\llbracket src2 \rrbracket_{single}$) ($\llbracket src1 \rrbracket_{exact}$ + $\llbracket src2 \rrbracket_{exact}$) `dest`
$\llbracket$ vaddsd src2, src1, dest $\rrbracket_i$	$=$	`set_double`	($\llbracket src1 \rrbracket_{double}$ + $\llbracket src2 \rrbracket_{double}$) ($\llbracket src1 \rrbracket_{exact}$ + $\llbracket src2 \rrbracket_{exact}$) `dest`
$\llbracket$ vsubss src2, src1, dest $\rrbracket_i$	$=$	`set_single`	($\llbracket src1 \rrbracket_{single}$ - $\llbracket src2 \rrbracket_{single}$) ($\llbracket src1 \rrbracket_{exact}$ - $\llbracket src2 \rrbracket_{exact}$) `dest`
$\llbracket$ vsubsd src2, src1, dest $\rrbracket_i$	$=$	`set_double`	($\llbracket src1 \rrbracket_{double}$ - $\llbracket src2 \rrbracket_{double}$) ($\llbracket src1 \rrbracket_{exact}$ - $\llbracket src2 \rrbracket_{exact}$) `dest`
$\llbracket$ vmulss src2, src1, dest $\rrbracket_i$	$=$	`set_single`	($\llbracket src1 \rrbracket_{single}$ $*$ $\llbracket src2 \rrbracket_{single}$) ($\llbracket src1 \rrbracket_{exact}$ $*$ $\llbracket src2 \rrbracket_{exact}$) `dest`

$$
\begin{array}{lcll}
\llbracket \text{ vmulsd src2, src1, dest } \rrbracket_i & = & \texttt{set_double} & (\llbracket src1 \rrbracket_{double} \;\texttt{*}\; \llbracket src2 \rrbracket_{double}) \\
 & & & (\llbracket src1 \rrbracket_{exact} \;\texttt{*}\; \llbracket src2 \rrbracket_{exact})\ \texttt{dest} \\
\llbracket \text{ vdivss src2, src1, dest } \rrbracket_i & = & \texttt{div_single} & \texttt{!src1 !src2 dest} \\
\llbracket \text{ vdivsd src2, src1, dest } \rrbracket_i & = & \texttt{div_double} & \texttt{!src1 !src2 dest}
\end{array}
$$

The translation of FMA instructions is specified as follows:

$$
\begin{array}{lll}
\llbracket \text{ vfmaddss src3,src2,src1,dest } \rrbracket_i & = \texttt{set_single} & (\llbracket src1 \rrbracket_{single}\texttt{*}\llbracket src2 \rrbracket_{single}\texttt{+}\llbracket src3 \rrbracket_{single}) \\
 & & (\llbracket src1 \rrbracket_{exact}\texttt{*}\llbracket src2 \rrbracket_{exact}\texttt{+}\llbracket src3 \rrbracket_{exact})\ \texttt{dest} \\
\llbracket \text{ vfmaddsd src3,src2,src1,dest } \rrbracket_i & = \texttt{set_double} & (\llbracket src1 \rrbracket_{double}\texttt{*}\llbracket src2 \rrbracket_{double}\texttt{+}\llbracket src3 \rrbracket_{double}) \\
 & & (\llbracket src1 \rrbracket_{exact}\texttt{*}\llbracket src2 \rrbracket_{exact}\texttt{+}\llbracket src3 \rrbracket_{exact})\ \texttt{dest} \\
\llbracket \text{ vfmsubss src3,src2,src1,dest } \rrbracket_i & = \texttt{set_single} & (\llbracket src1 \rrbracket_{single}\texttt{*}\llbracket src2 \rrbracket_{single}\texttt{-}\llbracket src3 \rrbracket_{single}) \\
 & & (\llbracket src1 \rrbracket_{exact}\texttt{*}\llbracket src2 \rrbracket_{exact}\texttt{-}\llbracket src3 \rrbracket_{exact})\ \texttt{dest} \\
\llbracket \text{ vfmsubsd src3,src2,src1,dest } \rrbracket_i & = \texttt{set_double} & (\llbracket src1 \rrbracket_{double}\texttt{*}\llbracket src2 \rrbracket_{double}\texttt{-}\llbracket src3 \rrbracket_{double}) \\
 & & (\llbracket src1 \rrbracket_{exact}\texttt{*}\llbracket src2 \rrbracket_{exact}\texttt{-}\llbracket src3 \rrbracket_{exact})\ \texttt{dest} \\
\llbracket \text{ vfnmaddss src3,src2,src1,dest } \rrbracket_i & = \texttt{set_single} & (\texttt{-}(\llbracket src1 \rrbracket_{single}\texttt{*}\llbracket src2 \rrbracket_{single})\texttt{+}\llbracket src3 \rrbracket_{single}) \\
 & & (\texttt{-}(\llbracket src1 \rrbracket_{exact}\texttt{*}\llbracket src2 \rrbracket_{exact})\texttt{+}\llbracket src3 \rrbracket_{exact})\ \texttt{dest} \\
\llbracket \text{ vfnmaddsd src3,src2,src1,dest } \rrbracket_i & = \texttt{set_double} & (\texttt{-}(\llbracket src1 \rrbracket_{double}\texttt{*}\llbracket src2 \rrbracket_{double})\texttt{+}\llbracket src3 \rrbracket_{double}) \\
 & & (\texttt{-}(\llbracket src1 \rrbracket_{exact}\texttt{*}\llbracket src2 \rrbracket_{exact})\texttt{+}\llbracket src3 \rrbracket_{exact})\ \texttt{dest} \\
\llbracket \text{ vfnmsubss src3,src2,src1,dest } \rrbracket_i & = \texttt{set_single} & (\texttt{-}(\llbracket src1 \rrbracket_{single}\texttt{*}\llbracket src2 \rrbracket_{single})\texttt{-}\llbracket src3 \rrbracket_{single}) \\
 & & (\texttt{-}(\llbracket src1 \rrbracket_{exact}\texttt{*}\llbracket src2 \rrbracket_{exact})\texttt{-}\llbracket src3 \rrbracket_{exact})\ \texttt{dest} \\
\llbracket \text{ vfnmsubsd src3,src2,src1,dest } \rrbracket_i & = \texttt{set_double} & (\texttt{-}(\llbracket src1 \rrbracket_{double}\texttt{*}\llbracket src2 \rrbracket_{double})\texttt{-}\llbracket src3 \rrbracket_{double}) \\
 & & (\texttt{-}(\llbracket src1 \rrbracket_{exact}\texttt{*}\llbracket src2 \rrbracket_{exact})\texttt{-}\llbracket src3 \rrbracket_{exact})\ \texttt{dest}
\end{array}
$$

8.4 Translation of annotations to Why

8.4.1 Translation of annotations in presence of floating-point arithmetic

The translation of annotations was presented in Chapter 7 but it is only for 32-bit and 64-bit integer. In this chapter, we add some rules of the translation of annotations for floating-point types: `float` and `double`.

$$
\begin{array}{lcl}
\llbracket \#float\#v\# \rrbracket_{term} & = & \texttt{single_value(sel_single(}\overline{v}\texttt{))} \\
\llbracket \#double\#v\# \rrbracket_{term} & = & \texttt{double_value(sel_doule(}\overline{v}\texttt{))} \\
\llbracket \backslash exact(\#\tau\#v\#) \rrbracket_{term} & = & \texttt{sel_exact(}\overline{v}\texttt{)} \text{ where } \tau \in \{\texttt{float}, \texttt{double}\}
\end{array}
$$

When st(i) is in annotations

When a C program is compiled with optimization options in x87 mode, the variables in annotations in assembly code may be a x87 stack register. In this case, their type is no longer a `double` or `float`, we impose that its type is extended double.

$$\llbracket \#single\#st(\%i)\# \rrbracket_{term} = \texttt{binary80_value(sel_80(st}\bar{i}\texttt{))}$$
$$\llbracket \#double\#st(\%i)\# \rrbracket_{term} = \texttt{binary80_value(sel_80(st}\bar{i}\texttt{))}$$

For example, the following annotation:

```
\abs_real(#double#%st#) <= 1000.0 * (1.0 + 0x1.004p-44)
```

is translated into Why as

```
abs_real(binary80_value(sel_80(st0))) <= 1000.0*(1.0+0x1.004p-44)
```

8.5 Soundness of translation

The soundness of the translation in simple programs (which contain only general-purpose instructions) is presented in Section 7.3. In this section, we will extend it for floating-point numbers.

8.5.1 Definition of the execution of a assembly program

The definition of memory state in assembly program in Section 7.3 is reused. We denote by $bv2single(bv)$ and $bv2double(bv)$ the representation in single and double of bv, by $single2bv(x)$ and $double2bv(x)$ the bitvector of a single and double value x, by $\circ_{32}(x)$ and $\circ_{64}(x)$ the rounding value in round-to-nearest mode in 32 and 64 bits of x. The execution of an instruction in SSE/SSE2 mode is : $S,\, i \Rightarrow S'$. It is defined as follows:

$$\frac{}{S,\ movss\ src, dest \Rightarrow S[dest \leftarrow single(S(src))]}$$

$$\frac{}{S,\ movsd\ src, dest \Rightarrow S[dest \leftarrow double(S(src))]}$$

$$\frac{bv2single(S(dest)) + bv2single(S(src))\ does\ not\ overflow\ in\ 32\ bits}{S,\ addss\ src, dest \Rightarrow S[dest \leftarrow single2bv(\circ_{32}(bv2single(S(dest)) + bv2single(S(src))))]}$$

$$\frac{bv2double(S(dest)) + bv2double(S(src))\ does\ not\ overflow\ in\ 64\ bits}{S,\ addsd\ src, dest \Rightarrow S[dest \leftarrow double2bv(\circ_{64}(bv2double(S(dest)) + bv2double(S(src))))]}$$

$$\frac{bv2single(S(dest)) - bv2single(S(src))\ does\ not\ overflow\ in\ 32\ bits}{S,\ subss\ src, dest \Rightarrow S[dest \leftarrow single2bv(\circ_{32}(bv2single(S(dest)) - bv2single(S(src))))]}$$

$$\frac{bv2double(S(dest)) - bv2double(S(src))\ does\ not\ overflow\ in\ 64\ bits}{S,\ subsd\ src, dest \Rightarrow S[dest \leftarrow double2bv(\circ_{64}(bv2double(S(dest)) - bv2double(S(src))))]}$$

$$\frac{bv2single(S(dest)) * bv2single(S(src))\ does\ not\ overflow\ in\ 32\ bits}{S,\ mulss\ src, dest \Rightarrow S[dest \leftarrow single2bv(\circ_{32}(bv2single(S(dest)) * bv2single(S(src))))]}$$

$$\frac{bv2double(S(dest)) * bv2double(S(src))\ does\ not\ overflow\ in\ 64\ bits}{S,\ mulsd\ src, dest \Rightarrow S[dest \leftarrow double2bv(\circ_{64}(bv2double(S(dest)) * bv2double(S(src))))]}$$

$$\frac{bv2single(S(src)) <> 0 \quad bv2single(S(dest))/bv2single(S(src))\ does\ not\ overflow\ in\ 32\ bits}{S,\ divss\ src, dest \Rightarrow S[dest \leftarrow single2bv(\circ_{32}(bv2single(S(dest))/bv2single(S(src))))]}$$

$$\frac{bv2double(S(src)) <> 0 \quad bv2double(S(dest))/bv2double(S(src))\ does\ not\ overflow\ in\ 64\ bits}{S,\ divsd\ src, dest \Rightarrow S[dest \leftarrow double2bv(\circ_{64}(bv2double(S(dest))/bv2double(S(src))))]}$$

8.5.2 Relation between Why state and assembly state (case of floating-point programs)

Remind what we have said in Section 7.3 that if $\overline{S}$ is a Why state and S is assembly state then $\overline{S}$ simulates S (denotes by $\overline{S} \sim S$) iff

1. For all register r, $S(r) \cong \overline{S}(\overline{r})$ where $bv \cong reg$ with bv is bitvector and reg is register.
2. For all memory reference $m =$ `off(reg,...)`, either $\overline{m}$ is defined and then $S(m) \cong \overline{S}(\overline{m})$ or $\overline{m}$ is not defined.

It is necessary to make sure that two distinguished memory references (syntactically) imply two different addresses (This follows Assumption 7.1). This means that if $address(m_1) = address(m_2)$ then $\overline{m_1} = \overline{m_2}$.

We denote $double(bv) =$ `double_value(sel_double(reg))`, this means that the translation of $double(bv)$ to Why will be $double_value(sel_double(reg))$. We denote in the same way for single.

Here we will prove that Theorem 8.1 (the stating of this theorem is the same as Theorem 7.8) is still correct with SSE/SSE2 instructions.

Theorem 8.1 *For all assembly state S, Why state $\overline{S}$ such that $\overline{S} \sim S$. For all sequence of assembly instructions i_1, ..., i_n:*
If $\overline{S}$, $[\![i_1; \ldots; i_n]\!]_i \Rightarrow^m \overline{S'}$
then $\exists S'$ such that S, $i_1; \ldots; i_n \Rightarrow S'$ where $\overline{S'} \sim S'$.

As specified in parameters, each floating-point number has two parts: the rounded one and the real one (or the exact one). Here, $\overline{S} \sim S$ does not depend on `sel_exact`. The proof of the rounded part is done below. The exact part will be mentioned in the next subsection. As before, we only detail a few instructions.

Proof.

1. `movsd` instruction: `movsd src, dest`

   In assembly program:

   $bv2double(S'(dest)) = bv2double(S(src))$.

   The translation into Why:

   $[\![$ movsd src, dest $]\!]_i =$ `set_double_no_check` $[\![src]\!]_{double}$ $[\![src]\!]_{exact}$ ***dest***

   From the post-condition of `set_double_no_check` we have:

   `double_value(sel_double(`$\overline{S'}(\overline{dest})$`))=double_value(sel_double(`$\overline{S}(\overline{src})$`))`

   Because $bv2double(bv) =$ `double_value(sel_double(reg))` and $\overline{S} \sim S$, we have:

   $$\begin{aligned} bv2double(\overline{S'}(\overline{dest})) &= bv2double(S(src)) \\ &= bv2double(S'(dest))(\text{proved}). \end{aligned}$$

2. `addsd` instruction: `addsd src, dest`

   In assembly program:

   $bv2double(S'(dest)) = \circ_{64}(bv2double(S(dest)) + bv2double(S(src)))$ if the addition does not overflow.

   The translation into Why of this instruction is:

$[\![$ addsd src,dest $]\!]_i$= `set_double` ($[\![src]\!]_{double}$+$[\![dest]\!]_{double}$) ($[\![src]\!]_{exact}$+$[\![dest]\!]_{exact}$) *dest*

From the post-condition of `set_double` we have:

$\texttt{double_value(sel_double(}\overline{S'}(\overline{dest})\texttt{))} = $
$\quad\texttt{round_double(nearest_even,double_value(sel_double(}\overline{S}(\overline{dest})\texttt{))}+$
$\qquad\qquad\texttt{double_value(sel_double(}\overline{S}(\overline{src})\texttt{)))}$

As $bv2double(bv) = $ `double_value(sel_double(reg))` and $\overline{S} \sim S$, we have:

$$\begin{aligned} bv2double(\overline{S'}(\overline{dest})) &= \circ_{64}(bv2double(S(dest)) + bv2double(S(src))) \\ &= bv2double(S'(dest)) \text{ (proved).} \end{aligned}$$

Notice that in the translation of `addsd` instruction, the precondition must specify that the addition does not overflow. If the addition overflows in 64 bits then the program blocks.

□

For proving other instructions in SSE/SSE2 mode and also in x87 and FMA, we do the same way as the proof of the two instructions above.

8.5.3 About exact value

In assembly, there does not exist the real value of an operand. As the assembly code generated contains annotations, we define here the semantic of the assembly in which each operand *op* (register or memory reference) value is a pair (*bv*, *real*). This semantic is different from the one in Chapter 7 where each operand (register or memory reference) value is a *bv*. The modified semantic helps us to prove the properties about exact value in annotations.

We denote by *S(op)* the pair ($S_b(op)$, $S_e(op)$) where $S_b(op)$ is the notation $S(op)$ we used in subsection 8.5.1, $S_e(op)$ is the exact value of *op*. The execution of an instruction in SSE/SSE2 mode for exact value is defined below:

$$\frac{}{S,\ movss\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(src)}$$

$$\frac{}{S,\ movsd\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(src)}$$

$$\frac{bv2single(S_b(dest)) + bv2single(S_b(src))\ does\ not\ overflow\ in\ 32\ bits}{S,\ addss\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(dest) + S_e(src)}$$

$$\frac{bv2double(S_b(dest)) + bv2double(S_b(src))\ does\ not\ overflow\ in\ 64\ bits}{S,\ addsd\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(dest) + S_e(src)}$$

$$\frac{bv2single(S_b(dest)) - bv2single(S_b(src))\ does\ not\ overflow\ in\ 32\ bits}{S,\ subss\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(dest) - S_e(src)}$$

$$\frac{bv2double(S_b(dest)) - bv2double(S_b(src))\ does\ not\ overflow\ in\ 64\ bits}{S,\ subsd\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(dest) - S_e(src)}$$

$$\frac{bv2single(S_b(dest)) * bv2single(S_b(src)) \ does\ not\ overflow\ in\ 32\ bits}{S,\ mulss\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(dest) * S_e(src)}$$

$$\frac{bv2double(S_b(dest)) * bv2double(S_b(src)) \ does\ not\ overflow\ in\ 64\ bits}{S,\ mulsd\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(dest) * S_e(src)}$$

$$\frac{bv2single(S_b(src)) <> 0 \qquad bv2single(S_b(dest))/bv2single(S_b(src)) \ does\ not\ overflow\ in\ 32\ bits}{S,\ divss\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(dest)/S_e(src)}$$

$$\frac{bv2double(S_b(src)) <> 0 \qquad bv2double(S_b(dest))/bv2double(S_b(src)) \ does\ not\ overflow\ in\ 64\ bits}{S,\ divsd\ src, dest \Rightarrow S_e(dest) \leftarrow S_e(dest)/S_e(src)}$$

The exact value is an additional information and this value may overflow. However, in order to assure that an instruction is executed without overflow in 32 and 64 bits, the precondition about single and double value of the result is needed.

Now we will prove Theorem 8.1 for exact value.

Proof.

1. `movsd` instruction: `movsd src, dest`

   In assembly program, we have:

   $S'_e(dest) = S_e(src)$.

   The translation into Why is:

   $[\![$ movsd src, dest $]\!]_i$ = `set_double_no_check` $[\![src]\!]_{double}$ $[\![src]\!]_{exact}$ *dest*

   From the post-condition of `set_double_no_check`, we have:

   `sel_exact(`$\overline{S'_e}(\overline{dest})$`)` = `sel_exact(`$\overline{S_e}(\overline{src})$`)`

   Because $S_e(reg)$ = `sel_exact(reg)` and $\overline{S} \sim S$, we have:

   $$\begin{aligned}\overline{S'_e}(\overline{dest}) &= S_e(src) \\ &= S'_e(dest)(\text{proved}).\end{aligned}$$

2. `addsd` instruction: `addsd src, dest`

   In assembly program:

   $S'_e(dest) = S_e(dest) + S_e(src)$.

   The translation into Why of this instruction is:

   $[\![$ addsd src,dest $]\!]_i$= `set_double` `(`$[\![src]\!]_{double}$`+`$[\![dest]\!]_{double}$`)` `(`$[\![src]\!]_{exact}$`+`$[\![dest]\!]_{exact}$`)` *dest*

   From the post-condition of `set_double` we have:

   `sel_exact(`$\overline{S'_e}(\overline{dest})$`)` = `sel_exact(`$\overline{S_e}(\overline{dest})$`)` + `sel_exact(`$\overline{S_e}(\overline{src})$`)`

   As $S_e(reg)$ = `sel_exact(reg)` and $\overline{S} \sim S$, we have:

   $$\begin{aligned}\overline{S'_e}(\overline{dest}) &= S_e(dest) + S_e(src) \\ &= S'_e(dest)(\text{proved}).\end{aligned}$$

   Notice that in the translation of `addsd` instruction, the precondition is necessary in order to make sure that the addition does not overflow. If it overflows then the program blocks.

Others instructions are handled similarly.

□

```
double doublerounding(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64;
  double z = x + y;

  //@ assert z == 1.0;
  return z;
}
```

Figure 8.4: A simple floating-point program

```
1 movabsq $4607182418800017408, %rax
2 movq    %rax, -32(%rbp)
3 movabsq $4368493837572636672, %rax
4 movq    %rax, -24(%rbp)
5 fldl    -32(%rbp)
6 faddl   -24(%rbp)
7 fstpl   -16(%rbp)
```

Figure 8.5: Assembly code in x87 mode of Figure 8.4 example

8.6 Illustrations

The two following examples are used for illustrating the translation of floating-point instructions to Why. These examples are proved with different results when compiling with different options of compiler.

8.6.1 Double rounding example

Let us go back to the example about double rounding in Figure 2.2. We modify it by adding an assertion `//@ assert z == 1.0` (See Figure 8.4) which will be true in x87 case and won't be true in SSE2 case.

x87 mode

The assembly code in Figure 8.5 is generated by `gcc -S -mfpmath=387`.

- At line 1, the instruction `movabsq` copies the value `4607182418800017408` to `%rax`. The value `4607182418800017408` corresponds to `1.0` in double. Notice that in assembly code, the floating-point value is represented as an integer value.
- Then at line 2, the content of the register `%rax` is copied to the memory reference `-32(%rbp)` (corresponds to `x`).
- Similarly to line 1, at line 3, `4368493837572636672` (corresponds to $2^{-53} + 2^{-64}$) to `%rax`.
- Next, at line 4, `-24(%rbp)` (corresponds to `y`) receives the data from register `%rax`.
- At line 5, loads the the floating-point value at `-32(%rbp)` to the register `st(0)`(80 bits) of the stack by the instruction `fld`.

Proof obligations	Gappa 0.12.1	Statistics
function doublerounding_0 Correctness	✓	3/3
1. precondition	✓	
2. precondition	✓	
3. assertion	✓	

```
H9: no_overflow_double(nearest_even, binary80_value(sel_80(st7_0)))
_16__rbp_: register
H10: double_value(sel_double(_16__rbp_)) ==
     round_double(nearest_even, binary80_value(sel_80(st7_0))) &&
     sel_exact(_16__rbp_) == sel_exact(st7_0)
_______________

double_value(sel_double(_16__rbp_)) == 1.0
```

```
(*#fldl -32(%rbp)*)
 set_80_no_check (double_value (sel_double !_32__rbp_)) (sel_exact !
_32__rbp_) st7;
(*#faddl        -24(%rbp)*)
 set_80 ((binary80_value (sel_80 !st7))+ (double_value (sel_double !
_24__rbp_))) ((sel_exact !st7)+(sel_exact !_24__rbp_)) st7;
(*#fstpl        -16(%rbp)*)
 set_double (binary80_value (sel_80 !st7)) (sel_exact !st7) _16__rbp_;
assert{ double_value(sel_double(_16__rbp_)) = 1.0
};
```

Figure 8.6: Result of Figure 8.4 program

- At line 6, the instruction `faddl` adds `st(0)` with `-24(%rbp)`:
  `st(0)` $\leftarrow \circ_{80}($`st(0)` $+$ `-24(%rbp)`$)$.
- At line 7, the floating-point value in `st(0)` is rounded to 64 bits and copied to the memory reference `-16(%rbp)`.

When this assembly code is fed into our translator to Why, and the result analyzed by Why, three proof obligations are produced. One is naturally for proving the assertion, the two others are required to prove the absence of overflow: once at line 6 of Figure 8.5, corresponding to the addition `x+y` in the source code, and once at line 7, which amounts to store the 80-bit value of the x87 stack into a 64-bit memory cell. These three obligations are proved valid using the Gappa automatic prover (See the screenshot in Figure 8.6).

Here are the VCs for proving the assertion in x87 mode:

```
_rax: register
H1: double_value(sel_double(_rax)) = 0x1.p0
_32__rbp_: register
H2: double_value(sel_double(_32__rbp_)) = double_value(sel_double(_rax))
_rax0: register
H3: double_value(sel_double(_rax0)) = 0x1.002p-53
_24__rbp_: register
H4: double_value(sel_double(_24__rbp_)) = double_value(sel_double(_rax0))
st7: register
H5: binary80_value(sel_80(st7)) = double_value(sel_double(_32__rbp_))
H6: no_overflow_binary80(nearest_even,
    binary80_value(sel_80(st7)) + double_value(sel_double(_24__rbp_)))
st7_0: register
H7: binary80_value(sel_80(st7_0)) = round_binary80(nearest_even,
                                     (binary80_value(sel_80(st7)) +
                                      double_value(sel_double(_24__rbp_))))
H8: no_overflow_double(nearest_even, binary80_value(sel_80(st7_0)))
_16__rbp_: register
H9: double_value(sel_double(_16__rbp_)) = round_double(nearest_even,
                                          binary80_value(sel_80(st7_0)))
-------------------------------------------------------------------------
double_value(sel_double(_16__rbp_)) = 1.0
```

```
1 movabsq $4607182418800017408, %rax
2 movq    %rax, -32(%rbp)
3 movabsq $4368493837572636672, %rax
4 movq    %rax, -24(%rbp)
5 movsd   -32(%rbp), %xmm0
6 addsd   -24(%rbp), %xmm0
7 movsd   %xmm0, -16(%rbp)
```

Figure 8.7: Assembly code in SSE2 mode of Figure 8.4 example

They are explained as follows:

$$H1 : double(_rax) = 0x1p0$$
$$H2 : double(_32__rbp_) = double(_rax)$$
$$H3 : double(_rax0) = 0x1.002p-53$$
$$H4 : double(_24__rbp_) = double(_rax0)$$
$$H5 : binary80(st7) = double(_32__rbp_)$$
$$H6 : binary80(st7) + double(_24__rbp_) \ does\ not\ overflow\ in\ 80\ bits$$
$$H7 : binary80(st7_0) = \circ_{80}(binary80(st7) + double(_24__rbp_))$$
$$H8 : binary80(st7_0) \ does\ not\ overflow\ in\ 64\ bits$$
$$H9 : double(_16__rbp_) = \circ_{64}(binary80(st7_0))$$

$$Goal : double(_16__rbp_) = 1.0$$

If we do all possible substitutions, the goal reduces to:

$$\circ_{64}(\circ_{80}(0x1p0 + 0x1.002p-53)) = 1.0$$

This is proved automatically by Gappa.

SSE2 mode

The assembly code in Figure 8.7 is generated by `gcc -S` without any other options. In this code, 64-bit instructions (`movabs`, `movq`, `movsd`, `addsd`) are used. Thus, all the calculations are in 64 bits.

- From line 1 to line 4, the instructions generated are the same as the ones in Figure 8.5.
- At line 5, `%xmm0` receives the floating-point value from `-32(%rbp)`.
- At line 6, a floating-point addition in 64-bit is done by the instruction `addsd`, that is `%xmm0` $\leftarrow \circ_{64}($`%xmm0` + `-24(%rbp)`$)$.
- Finally, the floating-point value of `%xmm0` is rounded to 64 bits and then copied to `-16(%rbp)` (corresponds to `z`).

With this assembly code, the generated proof obligation corresponding to the assertion cannot be proved anymore. The modified assertion `z == 1.0 + 0x1p-52` can be proved instead. The VCs generated corresponding to the assembly code in Figure 8.7 are:

```
_rax: register
H1: double_value(sel_double(_rax)) = 0x1.p0
_32__rbp_: register
H2: double_value(sel_double(_32__rbp_)) = double_value(sel_double(_rax))
_rax0: register
H3: double_value(sel_double(_rax0)) = 0x1.002p-53
_24__rbp_: register
H4: double_value(sel_double(_24__rbp_)) = double_value(sel_double(_rax0))
_xmm0: register
H5: double_value(sel_double(_xmm0)) = double_value(sel_double(_32__rbp_))
H6: no_overflow_double(nearest_even,
    double_value(sel_double(_xmm0)) + double_value(sel_double(_24__rbp_)))
_xmm0_0: register
H7: double_value(sel_double(_xmm0_0)) = round_double(nearest_even,
                                         (double_value(sel_double(_xmm0)) +
                                          double_value(sel_double(_24__rbp_))))
_16__rbp_: register
H8: double_value(sel_double(_16__rbp_)) = double_value(sel_double(_xmm0_0))
-------------------------------------------------------------------------
double_value(sel_double(_16__rbp_)) = 1.0 + 0x1.p-52
```

The VCs above say that:

$$
\begin{aligned}
&H1 : double(_rax) = 0x1p0\\
&H2 : double(_32__rbp_) = double(_rax)\\
&H3 : double(_rax0) = 0x1.002p - 53\\
&H4 : double(_24__rbp_) = double(_rax0)\\
&H5 : double(_xmm0) = double(_32__rbp_)\\
&H6 : (double(_xmm0) + double(_24__rbp_))\ does\ not\ overflow\ in\ 64\ bits\\
&H7 : double(_xmm0_0) = \circ_{64}(double(_xmm0) + double(_24__rbp_))\\
&H8 : double(_16__rbp_) = double(_xmm0_0)
\end{aligned}
$$

$$Goal : double(_16__rbp_) = 1.0 + 0x1p - 52$$

If we do all possible substitutions, the goal reduces to:

$$\circ_{64}(0x1p0 + 0x1.002p - 53) = 1.0 + 0x1p - 52$$

This is proved by Gappa.

8.6.2 Overflow example

Monniaux [58] considers the program in Figure 8.8 to illustrate differences between architectures with respect to overflows.

When compiled with non-optimized options

The overflow example is compiled by `gcc -mfpmath=387 -O0`. Excerpt of the generated assembly code are shown on Figure 8.9.

```
double foo() {
  double v = 1e308;
  double y = v * v;
  return y/v;
}
```

Figure 8.8: Overflow example

```
1  movabsq  $9214871658872686752, %rax
2  movq     %rax, -8(%rbp)
3  fldl     -8(%rbp)
4  fmull    -8(%rbp)
5  fstpl    -16(%rbp)
6  fldl     -16(%rbp)
7  fdivl    -8(%rbp)
8  fstpl    -24(%rbp)
9  movsd    -24(%rbp), %xmm0
10 ....
```

Figure 8.9: Non-optimized assembly code of overflow example

- At line 1, `9214871658872686752` (corresponds to $\circ_{64}$`(1e308)`) is copied to `%rax`.
- At line 2, the data in `%rax` is copied to the memory reference `-8(%rbp)` (corresponds to the variable `x` in C program).
- Next, at line 3, `st(0)` of the stack receives the value of `-8(%rbp)`.
- Then, at line 4, a multiplication operation is done by the instruction `fmul`, that is `st(0)` $\leftarrow \circ_{80}$(`st(0) * -8(%rbp)`).
- At line 5, the value in `st(0)` is rounded to 64 bits and copied to `-16(%rbp)` (corresponds to `y`).
- At line 6, `st(0)` receives the value of `-16(%rbp)`.
- At line 7, `st(0)` $\leftarrow \circ_{80}$(`st(0) / -8(%rbp)`).
- At line 8, `-24(%rbp)` $\leftarrow \circ_{64}$(`st(0)`).
- Finally, at line 9, the value of `-24(%rbp)` is copied to the register `%xmm0`.

In brief, with the non-optimized assembly code in x87 mode, $v * v$ is calculated in 80 bits (line 4) and then rounded in 64 bits (line 5) before doing the division operation (line 7). The return value of this version is Infinity.

For this version, 5 obligations are generated to check absence of overflow at lines 4, 5, 7 and 8 of assembly code of Figure 8.9, and to check that divisor is not zero at line 7. All are proved by Gappa except the overflow at line 5, where the content of the 80-bit register holding the result of the multiplication is moved into a 64-bit memory cell, which indeed overflows. The VCs of the obligation at line 5 which check the overflow are shown below:

```
1    fldl    .LC0(%rip)
2    fld     %st(0)
3    fmul    %st(1), %st
4    fdivrp  %st, %st(1)
5    fstpl   -8(%rsp)
6    movsd   -8(%rsp), %xmm0
7  ....
8  .LC0:
9    .long 2246822048
10   .long 2145504499
```

Figure 8.10: Optimized assembly of overflow example

```
_rax: register
H1: double_value(sel_double(_rax)) = 0x1.1ccf385ebc8ap1023
_8__rbp_: register
H2: double_value(sel_double(_8__rbp_)) = double_value(sel_double(_rax))
st7: register
H3: binary80_value(sel_80(st7)) = double_value(sel_double(_8__rbp_))
H4: no_overflow_binary80(nearest_even,
    binary80_value(sel_80(st7)) * double_value(sel_double(_8__rbp_)))
st7_0: register
H5: binary80_value(sel_80(st7_0)) = round_binary80(nearest_even,
                                    (binary80_value(sel_80(st7)) *
                                     double_value(sel_double(_8__rbp_))))
-----------------------------------------------------------------------------
no_overflow_double(nearest_even, binary80_value(sel_80(st7_0)))
```

They are explained as follows:

$$H1 : double(_rax) = 0x1.1ccf385ebc8ap1023 \ (\circ_{64}(1e308))$$
$$H2 : double(_8__rbp_) = double(_rax)$$
$$H3 : binary80(st7) = double(_8__rbp_)$$
$$H4 : binary80(st7) * double(_8__rbp_) \ does\ not\ overflow\ in\ 80\ bits$$
$$H5 : binary80(st7_0) = \circ_{80}(binary80(st7) * double(_8__rbp_))$$

$$Goal : binary80(st7_0) \ does\ not\ overflow\ in\ 64\ bits$$

If we do all possible substitutions, the goal reduces to:

$$\circ_{80}(0x1.1ccf385ebc8ap1023 * 0x1.1ccf385ebc8ap1023) \text{ does not overflow in 64 bits}$$

This obligation is not proved because $\circ_{80}(0x1.1ccf385ebc8ap1023 * 0x1.1ccf385ebc8ap1023)$ overflows in 64 bits.

When compiled with optimized options

The optimized version (Figure 8.10) is compiled with `gcc -mfpmath=387 -O1`. In this assembly code, the multiplication and division are done directly in 80 bits (line 3-4). Its result is v and it does not overflow. More precisely:

- At line 1, `st(0)` receives the value at `.LC0(%rip)` (the value $\circ_{64}$`(1e308)` is put in data section determined by the symbol `.LC0`).
- At line 2, the stack loads the value at `st(0)`, this means that after executing this instruction, `st(1)` and `st(0)` has the same value $\circ_{64}$`(1e308)`.
- At line 3, `st(0)` $\leftarrow \circ_{80}$`(st(0) * st(1))`.
- Then at line 4, `st(1)` $\leftarrow \circ_{80}$`(st(0) / st(1))` and pops `st(0)`.
- At line 5, the result of the division is rounded in 64 bits and put in `-8(%rbp)`.
- The result at line 5 is copied to `%xmm0` (at line 6).

For this version, 4 obligations are generated at lines 3, 4 and 5 of Figure 8.10 and all are proved by Gappa. Indeed there is no overflow in this version because the result of multiplication is not temporarily stored into a 64-bit register. The VCs generated for obligation at line 5 are illustrated below:

```
H1: double_value(sel_double(_LC0__rip_)) = 0x1.1ccf385ebc8ap1023
st7: register
H2: binary80_value(sel_80(st7)) = double_value(sel_double(_LC0__rip_))
st6: register
H3: binary80_value(sel_80(st6)) = binary80_value(sel_80(st7))
H4: no_overflow_binary80(nearest_even,
    binary80_value(sel_80(st7)) * binary80_value(sel_80(st6)))
st6_0: register
H5: binary80_value(sel_80(st6_0)) = round_binary80(nearest_even,
                                    (binary80_value(sel_80(st7)) *
                                     binary80_value(sel_80(st6))))
H6: binary80_value(sel_80(st7)) <> 0.0 and
    no_overflow_binary80(nearest_even,
    binary80_value(sel_80(st6_0)) / binary80_value(sel_80(st7)))
st7_0: register
H7: binary80_value(sel_80(st7_0)) = binary80_value(sel_80(st6_0)) /
                                    binary80_value(sel_80(st7))
----------------------------------------------------------------------
no_overflow_double(nearest_even, binary80_value(sel_80(st7_0)))
```

The hypothesis H1 above comes from the axiom we define for the label `.LC0` in data section (line 8–10 of Figure 8.10). The VCs above are explained as follows:

$$
\begin{aligned}
&H1 : double(_LC0__rip_) = 0x1.1ccf385ebc8ap1023\ (\circ_{64}(1e308))\\
&H2 : binary80(st7) = double(_LC0__rip_)\\
&H3 : binary80(st6) = binary80(st7)\\
&H4 : (binary80(st7) * binary80(st6))\ does\ not\ overflow\ in\ 80\ bits\\
&H5 : binary80(st6_0) = \circ_{80}(binary80(st7) * binary80(st6))\\
&H6 : binary80(st7) <> 0.0\ and\\
&\qquad (binary80(st6_0)/binary80(st7))\ does\ not\ overflow\ in\ 80\ bits\\
&H7 : binary80(st7_0) = \circ_{80}(binary80(st6_0)/binary80(st7))\\
\\
&Goal : binary80(st7_0)\ does\ not\ overflow\ in\ 80\ bits
\end{aligned}
$$

If we do all possible substitutions, the goal reduces to:

$$\circ_{80}(\circ_{80}(0x1.1ccf385ebc8ap1023 * 0x1.1ccf385ebc8ap1023)/0x1.1ccf385ebc8ap1023)$$

does not overflow in 64 bits.

This is proved by Gappa.

With SSE2 mode

We can also analyze the code compiled in the SSE2 mode, resulting in 3 obligations: overflows for the multiplication and division and check divisor is not null. As expected, it cannot be proved that the multiplication does not overflow.

Chapter 9

Handling Conditional and loop statements

The goal of this chapter is to present how to handle conditional and loop statements. In order to do that, we need to construct a control flow graph (CFG) and to translate a CFG to Why.

9.1 Conditional instructions in assembly

The conditional instructions are divided into two groups: Jump instructions and conditional move instructions. Both do their task based on the value of the status flags which are set by instructions such as `test`, `cmp`, `comisd`, `ucomisd`, etc. Jump instructions transfer program control to another point whereas conditional move instructions make a move operation.

9.1.1 Jump instructions

There are two types of jump instructions:

Unconditional jump instruction

The unconditional jump instruction transfers program control to a different point in the instruction stream without recording return information. The destination operand specifies the address of the instruction being jumped to. This operand may be an immediate value, a general-purpose register, or a memory reference. The syntax of unconditional jump instruction is:

`jmp` target

Conditional jump instructions: Jcc

All the Jcc instructions are specified by Intel [44]. In this chapter, we present some of them which are often appeared in our examples.

Mnemonic	Meaning	Condition tested
jz	jump if zero	ZF = 1
je	jump if equal	
jnz	jump if not zero	ZF = 0
jne	jump if not equal	
jg	jump if greater	ZF = 0 and SF = OF
jnle	jump if not less or equal	
jge	jump if greater or equal	SF = OF
jnl	jump if not less	
jl	jump if less	SF <> OF
jnge	jump if not greater or equal	

9.1.2 Conditional move instructions: CMOVcc

These instructions check the state of one or more status flags and perform a move operation if the flags are in a specified state. A condition code (cc) is associated with one instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed, an execution continues with the instruction following the CMOVcc instruction.

Mnemonic	Meaning	Condition tested
cmova	Move if above	CF = 0 and ZF = 0
cmovae	Move if above or equal	CF = 0
cmovb	Move if below	CF = 1
cmovbe	Move if below or equal	CF = 0 or ZF = 1
cmovg	Move if greater	ZF = 0 and SF = OF
cmovge	Move if greater or equal	SF = OF
cmovnz	Move if not zero	ZF = 0
cmovz	Move if zero	ZF = 1

9.2 Definition of programs supported

In this chapter, we continue to extend the model from the previous chapters. More precisely, the assembly programs may contain condition instructions, corresponding to the complex statements in C language: `if then else`, `switch`, `for`, `do while`, `goto`, etc.

We add here some other constraints:

- It is possible that the compiler generates the calculated jump instruction for C statements such as `switch`. In this chapter, we do not handle this kind of jump instruction. We only consider the jump to a near relative address in which its operand is a label.
- Remind that in subsection 8.3.5, we try to translate "relative" stack registers into physical stack register statically. When compiling with x87 mode, one thing we want to make sure is that the value we calculate for top-of-stack register must be unique whatever is the path of the control-flow graph to reach the instruction. In all the programs that we handled in the previous chapter, this hypothesis was satisfied and we think that a compiler never generates the assembly code in which the top-of-stack value is not unique at each point.

9.3 Translation of comparison instructions

As conditional instructions depend on the status flags which are set by comparison instructions such as `cmp`, `comisd`, `ucomisd`, etc., we present here how to translate such instructions into Why.

We declare 6 status flags of EFLAGS register (See subsection 6.2.2) as follows:

```
parameter CF: int ref
parameter PF: int ref
parameter AF: int ref
parameter ZF: int ref
parameter SF: int ref
parameter OF: int ref
```

Notice that the value of these status flags in Why is either equal to or different 0 (instead of equal to 1).

9.3.1 Translation of `cmp` instruction

The `cmp` (CoMPare) instruction compares two operands (equal, not equal, and so on) and sets the status flags (See Section 6.2.2). It performs the same operation as the `sub` except that the result of subtraction is not saved. Thus, `cmp` does not change the source and destination operands. It is typically used in conjunction with a conditional jump instruction for decision making.

To translate the instruction `cmp` into Why, we declare the following parameter:

```
parameter cmp: a:int ->b:int ->
    {  }
      unit writes OF, ZF, SF
    { (b = a -> (ZF <> 0 and OF = 0 and SF = 0))
      and
      (b > a -> ZF = 0 and OF = 0 and SF = 0)
      and
      (b < a -> ZF = 0 and OF = 0 and SF <> 0) }
```

The translation of the instruction `cmp` is as follows:

$$\llbracket \text{ cmpl op1, op2 } \rrbracket_i \quad = \quad \texttt{cmp} \;\; \llbracket op1 \rrbracket_{int32} \;\; \llbracket op2 \rrbracket_{int32}$$
$$\llbracket \text{ cmpq op1, op2 } \rrbracket_i \quad = \quad \texttt{cmp} \;\; \llbracket op1 \rrbracket_{int64} \;\; \llbracket op2 \rrbracket_{int64}$$

9.3.2 Translation of floating-point comparison instructions

The floating-point comparison instructions are presented in subsection 8.1.1. To translate them into Why, we declare only one parameter `comi` as follows:

```
parameter comi: a:real->b:real->
   {  }
     unit writes CF, PF, AF, SF, ZF, OF
   { (b = a -> (ZF <> 0 and CF = 0 and PF = 0 and OF = 0 and SF = 0 and AF = 0))
     and
     (b > a -> (ZF = 0 and CF = 0 and PF = 0 and OF = 0 and SF = 0 and AF = 0))
     and
     (b < a -> (ZF = 0 and CF <> 0 and PF = 0 and OF = 0 and SF = 0 and AF = 0)) }
```

This parameter compares two real values and set status flags. The translation of this kind of instructions are as below:

$$\llbracket \text{ comisd op1, op2 } \rrbracket_i \quad = \quad \texttt{comi} \;\; \llbracket op1 \rrbracket_{double} \;\; \llbracket op2 \rrbracket_{double}$$
$$\llbracket \text{ comiss op1, op2 } \rrbracket_i \quad = \quad \texttt{comi} \;\; \llbracket op1 \rrbracket_{single} \;\; \llbracket op2 \rrbracket_{single}$$
$$\llbracket \text{ ucomisd op1, op2 } \rrbracket_i \quad = \quad \texttt{comi} \;\; \llbracket op1 \rrbracket_{double} \;\; \llbracket op2 \rrbracket_{double}$$
$$\llbracket \text{ ucomiss op1, op2 } \rrbracket_i \quad = \quad \texttt{comi} \;\; \llbracket op1 \rrbracket_{single} \;\; \llbracket op2 \rrbracket_{single}$$

```
//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;

/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  (\result != 0 ==> \result == l_sign(\exact(x))) &&
  @          \abs(\result) <= 1 ;
  @*/
int sign(double x, double e1, double e2) {
  if (x > e2)
    return 1;
  if (x < e1)
    return -1;

  return 0;
}
```

Figure 9.1: Example with *if*

9.4 Control Flow Graph construction from assembly code

CFG is an important part of this chapter. Before talking about how to construct it, the definition of a CFG is given below:

Definition 9.1 *A* Control Flow Graph *(abbreviated as CFG) is a directed graph where each node has:*

- *a label (a unique integer)*
- *a content*
- *an optional other label which denotes its* normal *successor node*

The content is either

- *a set of instructions*
- *an annotation: pre, post, assert or invariant*
- *a jump instruction, either conditional or unconditional, together with the label of the node to jump to.*

By convention, the entry node is labeled 0. The exit nodes are those which are not unconditional jumps or do not have a normal successor node (the unconditional jumps do not have a normal successor node but they have a label of node to jump to).

We construct one CFG for each function in assembly code.

9.4.1 Example with `if` statement

In order to understand how to construct a CFG with assembly code, we begin with a simple example in Figure 9.1 which is presented in Section 5.2. This example contains `if` statements.

The assembly code of function `sign` generated by `gcc -S` is in Figure 9.2. From Definition 9.1, we construct a CFG for this code (See Figure 9.3). This program has only one function `sign`, we have thus only one CFG.

```
1  sign:
2  .LFB0:
3          .cfi_startproc
4  ....
5          movsd    %xmm0, -24(%rbp)
6          movsd    %xmm1, -32(%rbp)
7          movsd    %xmm2, -40(%rbp)
8  #APP
9     /*requires #double#-32(%rbp)# <=
10                #double#-24(%rbp)#-\exact(#double#-24(%rbp)#)
11                                    <= #double#-40(%rbp)#; */
12 #NO_APP
13         movsd    -24(%rbp), %xmm0
14         ucomisd  -40(%rbp), %xmm0
15         seta     %al
16         testb    %al, %al
17         je       .L2
18         movl     $1, -4(%rbp)
19         jmp      .L3
20 .L2:
21         movsd    -32(%rbp), %xmm0
22         ucomisd  -24(%rbp), %xmm0
23         seta     %al
24         testb    %al, %al
25         je       .L4
26         movl     $-1, -4(%rbp)
27         jmp      .L3
28 .L4:
29         movl     $0, -4(%rbp)
30         nop
31 .L3:
32 #APP
33    /*ensures (#int#-4(%rbp)# != 0 ==>
34                       #int#-4(%rbp)# == l_sign(\exact(#double#-24(%rbp)#)))
35              && \abs(#int#-4(%rbp)#) <= 1 ;*/
36 #NO_APP
37         movl     -4(%rbp), %eax
38 ...
39         ret
40         .cfi_endproc
```

Figure 9.2: Assembly code of program in Figure 9.1

The table in Figure 9.3 have 14 nodes, begins by node 0 and ends by node 13. Each node contains a label, its content and an optional successor node. Node 13 does not point to any node in order to show that it is the last one. The nodes that are neither the final ones nor unconditional jump ones will point to a successor node in the list. The unconditional jump nodes such as node 6, node 10 do not have a successor node but they have a label of node to jump to. The conditional jump nodes have both a successor node and a node to jump to. For example, node 3 contains the conditional jump instruction `je .L2`, its successor is node 5 and the node to jump to is node 4. We illustrate this table by a graph (below the table).

Node	Content	Successor node	Jump to
0	(Line 2 – 7)	1	
1	precondition	2	
2	(Line 13 – 16)	3	
3	je .L2	5	4
4	(Line 20 – 24)	8	
5	(Line 18)	6	
6	jmp .L3		7
7	(Line 31)	12	
8	je .L4	9	11
9	(Line 26)	10	
10	jmp .L3		7
11	(Line 28 – 30)	7	
12	post-condition	13	
13	(Line 37 – 39)		

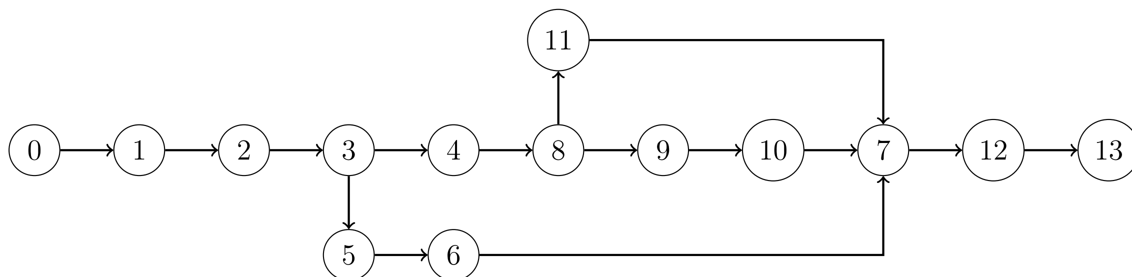


Figure 9.3: CFG for assembly code in Figure 9.2

9.4.2 Example with `do while` statement

The goal of the example in Figure 9.4 is to show how to construct a CFG of a program containing a loop statement `do while`. This example has a function `main` with two variables x and i. Their initial values are $x = 0$ and $i = 10$. In the loop, we increase x by 1 and decrease i by 1. The condition to exit the loop is that $i \leq 0$. The precondition of this function is $x \geq 0$ and the post-condition ensures $x = 10$.

In assembly code, the jump condition instructions generated from the statement `if` in C and the one generated from the statement `do while` are not different. The one different thing is that the jump instruction of `do while` creates a cycle in CFG.

The CFG corresponding to the assembly code generated by `gcc -S` is presented in Figure 9.5. This CFG has 11 nodes from node 0 to node 10. Node 1 contains the preconditions, node 4 contains the invariants and node 9 contains the post-conditions. There is a cycle: $3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3$.

9.4.3 Example with `goto`, `do while` and `if` statement

The example in Figure 9.6 finds the maximum value of a `double` array [9]. It contains many complex statements such as `goto`, `if` and `do while`. This CFG is thus a general one. The assembly instruction corresponding to `goto` statement is just a `jmp` instruction.

The CFG corresponding to the assembly code generated by `gcc -S` is presented in Figure 9.7. This CFG has 18 nodes: from node 0 to node 17 and has two cycles:

```
int x;
int i;

/*@ requires x >= 0;
  @ ensures x == 10;*/
void main(){
  x = 0;
  i = 10;
  do{
   /*@ loop invariant x == 10 - i && 10 >= i > 0;
     @ loop variant i;*/
    x = x + 1;
    i = i - 1;
  }while(i>0);
}
```

Figure 9.4: Program with loop statement

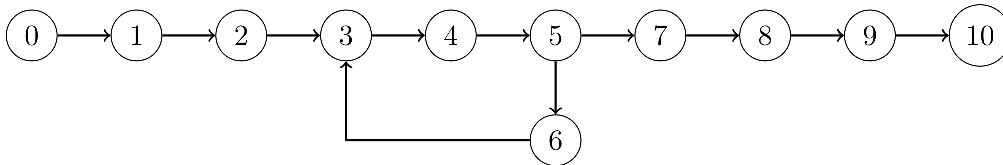


Figure 9.5: CFG of Program in Figure 9.4

- $3 \to 8 \to 6 \to 9 \to 10 \to 11 \to 12 \to 4 \to 7 \to 3$
- $6 \to 9 \to 10 \to 11 \to 12 \to 4 \to 5 \to 6$

The node 10 contains invariants of the program. This node is important when we generate Why functions from this CFG.

Notice that this example is not proved in this chapter because it contains arrays. It will be proved in Chapter 10.

What we have now is a Control Flow Graph for each function in assembly code. It facilitates the translation to Why which will be explained in the next section.

9.5 Translation from a CFG to Why

Our goal now is to build a Why program from a given CFG, so that the VCs generated from that Why program guarantee that the assembly program represented by the CFG satisfies its annotations.

Our construction of that Why program is inspired from other techniques proposed for dealing with unstructured programs in general [5, 32].

We assume that on any cycle of the CFG, there is at least one invariant node.

Algorithm 9.2 *We start from the initial node and traverse the CFG. The traversal stops whenever we meet a final node or an invariant node. From the assumption above, this traversal must terminate.*

```
/*@ requires n > 0 && \valid_range(t,0,n-1);
  @ ensures \forall integer k; 0<=k<n ==> \result >= t[k];
  @ */
double max_array( double t[] , int n ) {
  double m; int i = 0;
  goto L;
  do{
    if(t[i] > m){
    L:
      m = t[i];
    }
    //@ assert m >=t[i];

    /*@ loop invariant 0 <= i < n &&
      @              \forall integer k; 0<=k<=i ==> m >= t[k];
      @ */

    i = i +1;
  }while(i < n);

  return m;
}
```

Figure 9.6: Program with loop and goto statement

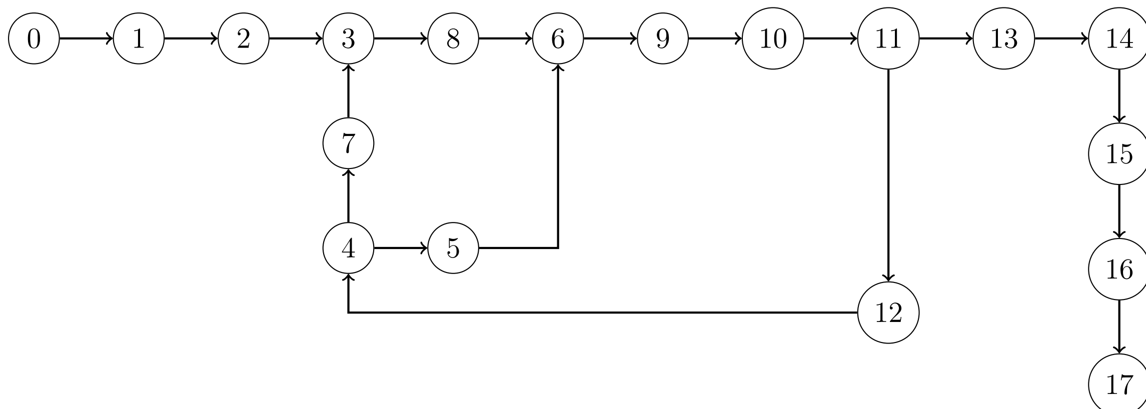


Figure 9.7: CFG for assembly code generated by gcc -S from example in Figure 9.6

```
algo generateWhy(g:CFG) : List of Why functions

var done : array[node] of boolean

(* traverse_from(n) will be called on each node n of the CFG of type pre
   or inv. using array done, we ensure that each of such node is treated
   only once *)
recursive traverse_from(n:node) : Why expression;

   (* explore(n,pre) will generate all the necessary Why functions to
```

```
    encode the subgraph of the CFG starting from node n, with
    precondition pre *)
 procedure explore(n:node, pre:Why predicate)

 var visited : array[node] of boolean

 (* explore_rec(n,prefix) traverses the CFG from node n and produces
    the Why function to encode the subgraph starting from n, assuming
    that prefix is the list of statements which encodes the path which
    arrives to n *)
 recursive explore_rec(n:node,prefix:Why expression)
  if visited[n]: fail (hypothesis not satisfied)
  visited[n] <- true;
  switch n.content_tag :
    case instruction(i) :
        explore(n.succ, prefix :: why_instr(i))
    case assert(p) :
        explore(n.succ, prefix :: assert (why_pred(p)))
    case pre(p) :
        explore(n.succ, prefix :: assume (why_pred(p)))
    case post(p) :
        explore(n.succ, prefix :: assert (why_pred(p))
    case inv(p) :
        produce_why_fun (prefix:: assert why_pred(p));
        traverse_from(n)
    case jump(l):
        explore_rec (l, prefix)
    case jump(c,l) :
        explore_rec (l, prefix :: assume (why_cond(c))) ;
        explore_rec (n.succ, prefix :: assume (not why_cond(c)))
    case conditional_move(i,c,l):
        explore_rec (l, prefix :: assume (why_cond(c)) :: why_instr(i)) ;
        explore_rec (n.succ, prefix :: assume (not why_cond(c)))
   If n is the last node of the subgraph then
        produce_why_fun (prefix)
   end explore_rec

  visited[i] <- false for each node i;
  explore_rec(n,[assume pre]);

end explore

if done[i] return;
done[i] <- true;
switch n.content_tag
  case pre(p): explore(n.succ, why_pred(p));
  case inv(p): explore(n.succ, why_pred(p));
  default : impossible
```

```
end traverse_node

main:
  done[i] <- false for each node i;
  traverse_from(0).
```

The algorithm `generateWhy(g:CFG)` takes a CFG `g` and returns a list of Why functions corresponding to `g`. Assume that we have the following functions:

- `why_instr(i)`: translates a sequence of instructions `i` (except conditional instructions) into Why
- `why_pred(p)`: translates a predicate `p` into Why
- `why_cond(c)`: translates a conditional instruction `c` into Why
- `proc_why_func`: produces a Why function.
- `prefix::str`: concatenates a string `str` to `prefix`.
- `n.succ` : returns the successor node of the node `n`.

This algorithm is described by using the following sub-functions:

- Recursive function `traverse_from(n:node):Why expression` is invoked on each node `n` having type `precondition` or `invariant` of the CFG. We use the array `done` in order to ensure that each of such node is treated once. The following function `explore` will be called in this function.
- Function `explore(n:node, pre: Why predicate)` generates Why functions to encode a subgraph of the CFG from node `n` with precondition `pre`. We use an array `visited` to know whether a node is visited.
- Recursive function `explore_rec(n:node, prefix: Why expression)` traverses the CFG from node `n` and produces the Why function to encode the subgraph starting from `n`, assuming that `prefix` is the list of statements encoding the path which arrives to `n`. The implementation of this function is described as follows:

  Firstly, if this node is visited then fails else we set `visited[n]` by `true`.

  Secondly, we consider the type of node `n`:

  - Instruction `i`: `prefix::why_instr(i)`
  - Assertion `p`: `prefix::assert why_pred(p)`
  - Precondition `p`: `prefix::assume why_pred(p)`
  - Post-condition `p`: `prefix::assert why_pred(p)`
  - Invariant `p`:
    - `produce_why_func(prefix::assert why_pred(p))`
    - `traverse_from(n)`
  - Unconditional jump instruction with label `l`: `explore_rec (l, prefix)`
  - Jump instruction with condition `c` and label `l`:
    - `explore_rec (l, prefix:: assume why_cond(c))`

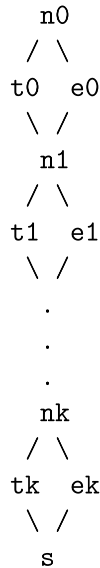
    * `explore_rec (n.succ, prefix:: assume not (why_cond(c)))`
  - Conditional move instruction `i` with condition `c` and label `l`:
    * `explore_rec (l, prefix:: assume why_cond(c)::why_instr(i))`
    * `explore_rec (n.succ, prefix:: assume not why_cond(c))`

Finally, if `n` is the last node of the subgraph then `produce_why_fun (prefix)`

This algorithm fails only when there exists a cycle without any invariant. Its complexity is an exponential number of the size of the CFG. Such complexity is introduced by a following example:

```
if c0 then t0 else e0;
if c1 then t1 else e1;
if c2 then t2 else e2;
...
if ck then tk else ek;
```

Its CFG is as follows



The number of paths from `n0` to `s` of this CFG is 2^k, so our algorithm will generate 2^k Why functions. However, this exponent is not a big problem. If we meet such case, it is possible to solve the problem manually by inserting invariants. With the example above, if we put manually invariants at each node nk then the number of Why functions will be $2k$.

Theorem 9.3 (Soundness) *Let p be the assembly code of a function, g its CFG, and E the set of Why programs generated by `generateWhy(g)`. If the VCs for the Why programs in E are valid, then p satisfies its annotations.*

Proof. From any state satisfying the precondition of p, a given execution corresponds to a path P in the graph g, from its initial node to a final node. If that path contains at least a cycle (with invariant I inside) then it can be split into subpaths $p_1, \ldots, p_n$, such that p_1 goes from the initial node to a node with an invariant I; $p_2, \ldots, p_{n-1}$ goes from a node containing I to a node with an invariant I; and p_n goes from a node containing I to a final node. Each p_i corresponds to an execution of one of the generated Why function, in a state satisfying its precondition (straightforward induction on n) and thus its satisfies the precondition of p.

Each e_i is the corresponding Why program of p_i. This means that if the VCs for the function e_i in E are valid then p_i satisfies its annotations.

□

Now let us go back to the examples that we have presented in the previous section. By applying Algorithm 9.2, the corresponding Why functions for each CFG is as below:

- The CFG in Figure 9.3 does not contains any cycles, so the Why functions generated are:
  - $0 \to 1 \to 2 \to 3 \to 4 \to 8 \to 9 \to 10 \to 7 \to 12 \to 13$
  - $0 \to 1 \to 2 \to 3 \to 4 \to 8 \to 11 \to 7 \to 12 \to 13$
  - $0 \to 1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 12 \to 13$
- For the CFG in Figure 9.5, there is a cycle in which node 4 contains the invariant. The Why functions generated are:
  - $0 \to 1 \to 2 \to 3 \to 4$
  - $4 \to 5 \to 6 \to 3 \to 4$
  - $4 \to 5 \to 7 \to 8 \to 9 \to 10$
- The CFG in Figure 9.7 has two cycles with the node 10 containing the invariant. There are 4 Why functions corresponding to this CFG:
  - $0 \to 1 \to 2 \to 3 \to 8 \to 6 \to 9 \to 10$
  - $10 \to 11 \to 12 \to 4 \to 5 \to 6 \to 9 \to 10$
  - $10 \to 11 \to 12 \to 4 \to 7 \to 3 \to 8 \to 6 \to 9 \to 10$
  - $10 \to 11 \to 13 \to 14 \to 15 \to 16 \to 17$

9.6 Examples

9.6.1 Clock drift

This example (See Figure 9.8) is presented in Section 5.4. We prove it with our translator by analyzing its assembly code. The goal of this example is firstly to construct CFG from assembly code generated from a program containing `for` statement; secondly to apply Algorithm 9.2 to generate Why functions and thirdly to illustrate the value of `st_ptr` on assembly code generated by `gcc -mfpmath=387`. We also demonstrate in this example that when compiling with different options of `gcc`, the bound of the rounding error is changed.

Now let us see the assembly code generated by `gcc -S -mfpmath=387 -O2` in Figure 9.9. With this assembly code, we look at only instructions related to x87 stack: `fldz`, `flds`, etc. which will change the value of top-of-stack. The value of top-of-stack in our model is shown below:

Line	Instruction	st_ptr
		8
6	fldz	7
8	flds .LC1(%rip)	6
17	fstp %st(1)	7
19	fstps -4(%rsp)	8

```
#define A 1.49012e-09
// A is a bound of (float)0.1 - 0.1
#define B 4.76838e-07
// B is a bound of round_error(t+(float)0.1) for 0 <= t <= 10.01
#define C (B + A)

/*@ lemma round01:
        \abs(\round_single(\nearest_even,0.1) - 0.1) <= A;*/
/*@ lemma round_01:
        \round_single(\nearest_even, 0.1) == 0x1.99999ap-4;*/

/*@ requires 0 <= n <= 100;
  @ ensures \abs(\result - n*0.1) <= n * C;*/
float f_single(int n){
  float t = 0.0f;
  int i;
  for(i=0; i < n; i++) {
    /*@ loop invariant 0 <= i < n <=100 &&
      @     \abs(t-i*0.1) <= i*C ;
      @ loop variant n-i;*/
  L:
    //@ assert 0.0 <= t <= 100.0*(0.1+C)   ;
    t = t + 0.1f;
    //@ assert \abs(t-(t@L+\round_single(\nearest_even,0.1)))<=B;
  }

  return t;
}
```

Figure 9.8: Clock drift program

Remind that we use the intermediate variable `st_ptr` to store the value of top-of-stack. At the entry of the function, we set `st_ptr = 8`. For each load instruction, the value of `st_ptr` decreases by 1. For each store and pop instruction, the value of `st_ptr` increases by 1. As expected, at the entry and at the exit of the function `f_single`, `st_ptr = 8` indicates that the stack is empty. There is a cycle (line 11 – 16) in this assembly code but there are not any instructions which change the value of `st_ptr`. Obviously, at each node in the CFG corresponding to this function, the value of `st_ptr` is unique whatever the path of the CFG.

An excerpt of assembly code generated by `gcc -mfpmath=387` is in Figure 9.10. All the instructions that change the value of top-of-stack is in the cycle. The change of `st_ptr` is shown as below:

Line	Instruction	st_ptr
		8
8	flds -12(%rbp)	7
9	flds .LC1(%rip)	6
10	faddp %st, %st(1)	7
11	fstps -12(%rbp)	8

At the entry and at the exit of the cycle, `st_ptr = 8` indicates that the stack is empty. In this case, we can also make sure that the value of `st_ptr` is unique whatever the path.

```
1   f_single:
2   .LFB0:
3           .cfi_startproc
4   ....
5           testl   %edi, %edi
6           fldz
7           jle     .L2
8           flds    .LC1(%rip)
9           fxch    %st(1)
10          xorl    %eax, %eax
11  .L4:
12  .L3:
13          fadd    %st(1), %st
14          addl    $1, %eax
15          cmpl    %edi, %eax
16          jne     .L4
17          fstp    %st(1)
18  .L2:
19          fstps   -4(%rsp)
20          movss   -4(%rsp), %xmm0
21          ret
22          .cfi_endproc
```

Figure 9.9: Assembly code of program in Figure 9.8 (generated by `gcc -S -mfpmath=387 -O2`)

```
1   f_single:
2   .LFB0:
3           .cfi_startproc
4   ....
5   .L4:
6           ;; invariant is here
7   .L3:
8           flds    -12(%rbp)
9           flds    .LC1(%rip)
10          faddp   %st, %st(1)
11          fstps   -12(%rbp)
12          addl    $1, -8(%rbp)
13  .L2:
14          movl    -8(%rbp), %eax
15          cmpl    -20(%rbp), %eax
16          jl      .L4
17  ....
18          ret
19          .cfi_endproc
```

Figure 9.10: Assembly code of program in Figure 9.8 (generated by `gcc -S -mfpmath=387`)

By analyzing assembly code using our translator, we construct a CFG for the function `f_single`. Then, from this CFG, we have four Why programs by using Algorithm 9.2. The obligations of these four Why functions (which are generated from assembly code compiled with `gcc -S` and `gcc - S -mfpmath=387`) are proved completely and automatically by the combination

of Gappa, Alt-Ergo and CVC3.

Thanks to Gappa, we have different values of the bound B when compiling the C program with different options of `gcc`. The following table show us the value of B depending on the mode and the optimization level. When the program is compiled in SSE2 mode or x87 mode without optimization, the value of B is `4.76838e-07`. However, the value of B is much smaller than the previous ones, `B = 4.33681e-19`, when the C program is compiled with `gcc -mfpmath=387 -O2`. There, all the value of `t` at each step is stored in 80 bits stack registers.

Architecture		**B**
SSE2		`4.76838e-07`
x87	`-O0`	`4.76838e-07`
x87	`-O2`	`4.33681e-19`

9.6.2 KB3D

This example (See Figure 9.11) illustrates the handling of conditional statements, the handling of function calls, and the way we express properties on rounding errors across functions. The presentation has been done in Section 5.2. Our goal here is to analyze what should be the value of E depending on the architecture.

Feeding the program in our assembly analyzer in SSE2 mode, the VCs are automatically proved valid using a combination of Gappa and SMT solvers (Alt-Ergo and CVC3). The bound `E` is indeed in that case exactly the same as the one found in Section 5.2 in a strict IEEE-754 mode. At least on this example, this shows that SSE2 assembly conforms strictly to the standard. The table below shows the value of `E` that are proved correct using various architecture-dependent settings.

Architecture		**E**
SSE2		`0x1p-45`
x87	`-O0`	`0x1.004p-46`
x87	`-O2`	`0x1.004p-46`
FMA	`-O2`	`0x1.8p-46`

As expected, using FMA improves over SSE2 (25% less) since less rounding occur. The extended precision of x87 is even better (around 50% less). The result is the best one with x87 and optimization `-O2` where all the values are stored in 80 bits. This new result is 1/1025 the one without optimization. Of course, all these bounds are smaller than the one we found in Section 5.2, which was `0x1.90641p-45` $\approx$ `3203` $\times\ 2^{-56}$, that is more than 50% higher than the SSE2 one. If we prove this program with Frama-C/Jessie, the value E we can prove is exactly the same as the case SSE2 in the table above.

9.7 Discussion

The weakest-precondition computation helps to prove assembly code which is an unstructured programs. However, this approach still have some limits. Compared to the traditional WP, this WP need more information to prove a program, especially when the program has a cycle with a loop invariant. For example, if a constant is stored in a variable and is used in the loop then this

```
#define E 0x1p-45

//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;

/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  (\result != 0 ==> \result == l_sign(\exact(x))) &&
  @          \abs(\result) <= 1 ;
  @*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
  }

/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures \result != 0
  @     ==> \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @            * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
*/
int eps_line(double sx, double sy,double vx, double vy){
  int s1,s2;

  s1=sign(sx*vx+sy*vy, -E, E);
  s2=sign(sx*vy-sy*vx, -E, E);

  return s1*s2;
}
```

Figure 9.11: Avionics program

---

information need to be put in invariant. Or if the precondition is needed to prove annotations in the loop then we also put it in invariant.

To illustrate what we have said above, let us go back to example Clock drift in Figure 9.8. When we generate assembly code with `gcc -S -mfpmath=387 -O2`, the assembly code is in Figure 9.12. Here, the constant $\circ_{32}$`(0.1)` is stored in `st(1)` (line 7 – 8). In the cycle (line 10 – 18), the addition is done by the instruction `fadd %st(1), %st(0)`, that is `%st(0) =` $\circ_{80}$`(%st(0) + %st(1))` and the value of `%st(1)` is set only once at line 8 (outside the cycle). Hence, the Why functions corresponding to the paths: $inv\ I \rightarrow \ldots \rightarrow inv\ I$ and $inv\ I \rightarrow \ldots \rightarrow Postcondition$ do not have any information about `%st(1)`. It is thus necessary to insert the condition `%st(1) =` $\circ_{32}(0.1)$ into the invariant I.

Still in this example, in the precondition, we have $0 \leq n \leq 100$. This condition must be added one more time into the invariant I because the Why functions corresponding to the paths: $inv\ I \rightarrow \ldots \rightarrow inv\ I$ and $inv\ I \rightarrow \ldots \rightarrow Postcondition$ do not know that $0 \leq n \leq 100$.

```
1   f_single:
2   .LFB0:
3           .cfi_startproc
4           testl   %edi, %edi
5           fldz
6           jle     .L2
7           flds    .LC1(%rip)
8           fxch    %st(1)
9           xorl    %eax, %eax
10  .L4:
11  .L3:
12  #APP
13          /* invariant l*/
14  #NO_APP
15          fadd    %st(1), %st
16          addl    $1, %eax
17          cmpl    %edi, %eax
18          jne     .L4
19          fstp    %st(1)
20  .L2:
21          fstps   -4(%rsp)
22          movss   -4(%rsp), %xmm0
23          ret
24          .cfi_endproc
```

Figure 9.12: Assembly code of program in Figure 9.8 (generated by `gcc -S -mfpmath=387 -O2`)

Chapter 10

Handling Arrays and Pointers

With programs containing pointers, the Separation Assumption 7.1 is not correct anymore. In order to prove such programs, we propose a new memory model. New rules for translating instructions and annotations into Why based on the memory model will be presented in the next sections.

10.1 Handled programs

The programs supported in this chapter have the following characteristics:

- Arrays and pointers with C types: `int`, `long`, `float`, `double` and `long double` are supported,
- Dynamic memory allocation is allowed,
- There are no structure type, no pointer of pointer and no cast in the program.
- The assembly code is generated on IA-64 architecture.

In the previous model, we translated registers and memory references into Why variables. However, there were no differences between the translation of a register and the one of a memory reference. In this model, all the memory references are represented and their translation is similar to the way the program access and store value in the memory.

10.2 New rules of translation for operands and instructions

The form of memory references in assembly language has been presented in chapter 6. Based on its form, in this section we will talk about the representation of memory in Why.

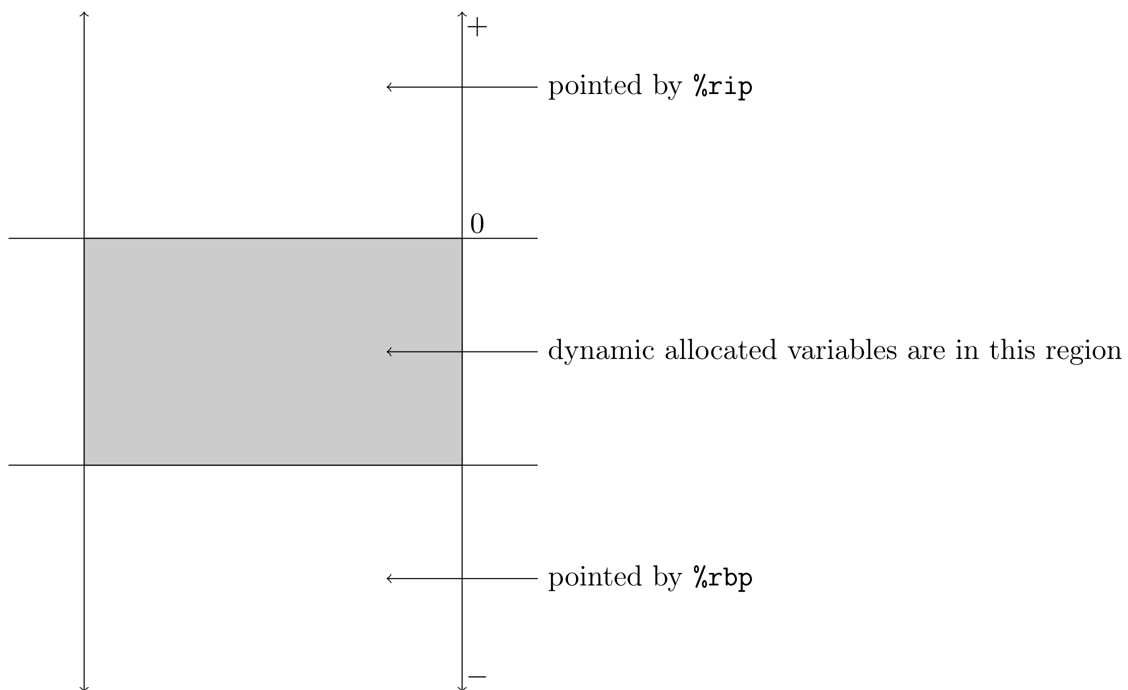
10.2.1 Representation of memory in Why

Memory space is divided into segments. Code, data, and stacks may be all contained in the same or different segments depending on the memory model. Each of them is pointed by different registers. For example, local variables are pointed by the register `%rbp`, a data stored in memory is pointed by `%rip`, etc.

In order to transform the memory references into Why, we model the memory in supposing that the memory is continuous and has three parts:

- Memory references for local variables and function arguments are normally pointed by the register `%rbp`. This `%rbp` points to the last data item placed on the stack. The memory references pointed by `%rbp` are `-4(%rbp)`, `-8(%rbp)`, `-16(%rbp)`, etc. This means that the value of *disp* is negative.
- Memory references for global variables and constants are pointed by the special register `%rip`. The displacement *disp* is usually a symbol or non-negative value.
- Dynamic allocated variables (which are not global variables) occupy a space in memory and their addresses do not change. We reserve a memory space for allocated variables (if exist) and this space is between `%rip` and `%rbp` (see the following figure). In other words, we make an assumption: in our model, the address value of allocated variables is less than `%rip` and greater than `%rbp`.

We insist that what we suppose here is not exactly the same as what the compiler do. What we assure in our memory model is that the three regions do not overlap.



In order to specify the validity of an address, we use the ACSL built-in predicate `\valid` and `\valid_range`. In our translator, they are interpreted by: the address is not less than `%rbp`. The translation of this predicate into Why will be discussed in the subsection 10.2.4.

To model the memory, we declare two following pointers:

```
logic _rbp: int
logic _rip: int
```

We set: `_rbp` $\leq$ `0`. As `_rbp` and `_rip` do not point at the same address in the memory, we will set `_rip` a positive integer. In our model, we thus set `_rip = 4`. All the memory references are based on these two pointers. Note that the address of dynamic allocated variables are always not less than `_rbp`. Two axioms associated with `_rbp` and `_rip` are:

```
axiom rbp_axiom: _rbp <= 0
axiom _rip_axiom: _rip = 4
```

```
int ga1[1], ga2[2], ga3[3];

/*@ ensures ga1[0] == i;
  @ ensures ga2[1] == i+1;
  @ ensures ga3[2] == i+2;
  @*/
void fg(int i) {
  ga1[0] = i;
  ga2[1] = i+1;
  ga3[2] = i+2;
}
```

Figure 10.1: An example containing arrays as global values.

Notice that in assembly code, `%rip` and `%rbp` addressing are *relative*. In our model, `%rip` and `%rbp` are considered as constants. This assumption is correct because the proofs are done function by function. Also notice that as we declare `_rbp` as a constant, all the instructions in which `%rbp` is the destination operand will not be translated into Why. This happens only when

- at the beginning of the function with instruction `movq %rsp, %rbp` to copy the value of `%rsp` to `%rbp`.
- at the end of the function with `popq %rbp` to restore the previous value of `%rbp`.

Without these instructions, the proofs do not change anything because they are done function by function.

We will present the translation of memory references depending on their scope: global variables, local variables and variables being arguments of a function as follows:

Translation of arrays defined as global variables

Remind that the directive `.comm` *symbol*, *length* declares a common symbol. By observing assembly code, we see that the declaration of global variables begins by the assembly directive `.comm` where *symbol* is the name of the variable and *length* is the total size of the variable. If it is an array, *length* is the memory allocated for this array (the number of elements $\times$ size of each element). These symbols are interpreted as constants like `%rbp` and `%rip`.

To illustrate the translation of an array being a global variable, we have a small C program in Figure 10.1. Its assembly code compiled with `gcc -S` is in Figure 10.2.

In this example, we have three integer arrays: `ga1` having one element, `ga2` having two elements and `ga3` having three elements. The size of each element in each array is 4.

In the assembly code, they are declared by using the directive `.comm` (line 1–3 of Figure 10.2). We consider only the first and the second argument after `.comm` (the third one is the alignment of the symbol, it is an optional argument). The first one is the symbol name and the second one is the length (in bytes). In others words, the second argument is the total number of bytes allocated for each symbol. These arrays are defined in Why as follows:

```
logic ga1: int
logic ga2: int
logic ga3: int
```

```
1           .comm    ga1,4,4
2           .comm    ga2,8,4
3           .comm    ga3,12,4
4   ....
5   fg:
6   .LFB0:
7           .cfi_startproc
8   ....
9           movl     %edi, -4(%rbp)
10          movl     -4(%rbp), %eax
11          movl     %eax, ga1(%rip)
12          movl     -4(%rbp), %eax
13          addl     $1, %eax
14          movl     %eax, ga2+4(%rip)
15          movl     -4(%rbp), %eax
16          addl     $2, %eax
17          movl     %eax, ga3+8(%rip)
18          movl     ga1(%rip), %eax
19          movl     ga2(%rip), %edx
20          movl     ga3(%rip), %ecx
21  #APP
22          /*ensures #intpointer#%eax#[0] == #int#-4(%rbp)#;
23            ensures #intpointer#%edx#[1] == #int#-4(%rbp)#+1;
24            ensures #intpointer#%ecx#[2] == #int#-4(%rbp)#+2;*/
25  #NO_APP
26
27          ret
28          .cfi_endproc
29  ....
```

Figure 10.2: Assembly code in SSE2 mode of Figure 10.1 example

Each array is defined as a constant indicating the position of the first element of the array in the memory.

An important question is: when we have many arrays in the same program, how can we make sure that the memory allocated for each array does not overlap the others?

There exists a model that separates the memory for pointers in C program [34]. As defined in this separation model, each pointer is considered to have a memory variable. For the three arrays above, we have three different memory variables. This model works well in C code source. We tried to apply this model in assembly code but it does not work. This is because if we use separation model, we cannot define the relation between these memory variables. We thus use only one memory variable for all. This memory variable will be presented in subsection 10.2.3.

As we have explained in subsection 10.2.1, memory references accessed by `%rbp` and `%rip` have two different directions. Memory references pointed by these two pointers are always disjoint.

Now let us back to the example in Figure 10.1. The three arrays represented in this model are illustrated in Figure 10.3.

We know that the size of `int` is 4 (bytes) so the total size of the array `ga1` is $1 \times 4 = 4$, the total size of `ga2` is $2 \times 4 = 8$, the total size of `ga3` is $3 \times 4 = 12$. We can also get the total size in the second argument of `.comm` and we use this value in our translator.

In order to assure that the memory references for these arrays do not overlap, we have some constraints:

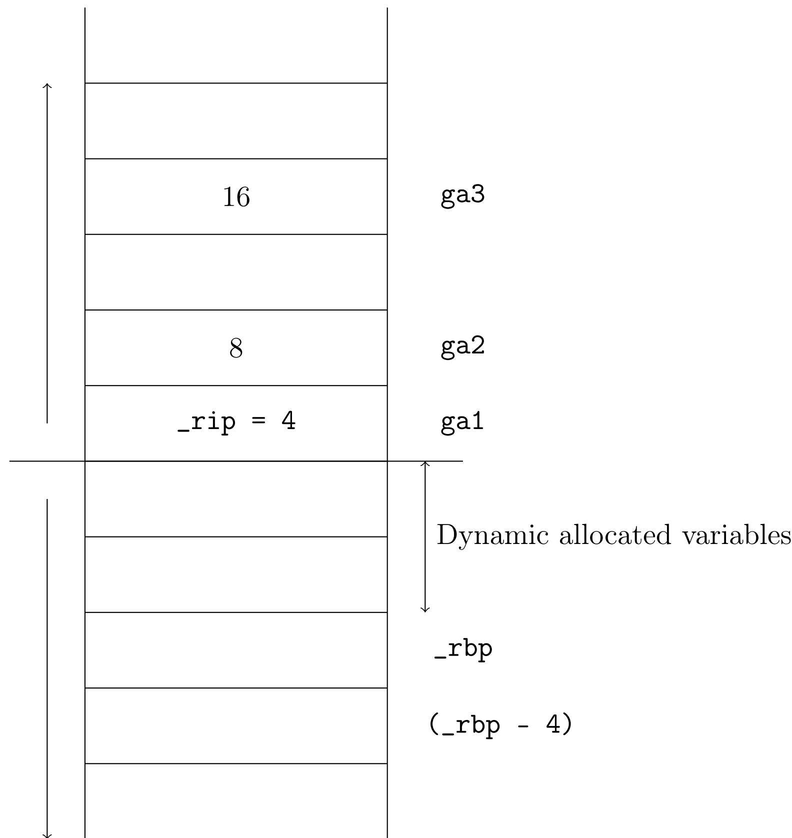


Figure 10.3: Memory model

- ga1 = 4 (ga1 = _rip)
- ga2 = 8 (ga2 = ga1 + 4)
- ga3 = 16 (ga3 = ga2 +8)

The order of the symbols in the memory is not important. The important thing is that the memory space for each symbol does not overlap the others.

In Why, we use the following axioms to specify these constraints:

```
axiom ga1_axiom: ga1 = 4
axiom ga2_axiom: ga2 = 8
axiom ga3_axiom: ga3 = 16
```

Once the axioms above are defined, when we translate these variables into Why, we do not need the pointer `_rip` anymore. To access `ga2[1]` for example, it is simply written `(ga2 + 1*4)`.

Translation of arrays defined as local variables

An example containing arrays as local variables is in Figure 10.4. Its assembly code is in Figure 10.5.

It is simpler to translate local arrays to Why because in assembly code, they are referenced by `_rbp` and the memory reference of a certain element of an array has been indicated by the compiler. For example, with the local variable `int la2[2]`, the first element of `la2` in

```
void lg(int i) {
  int la1[1], la2[2], la3[3];

  la1[0] = i;
  la2[1] = i + 1;
  la3[2] = i + 2;

  //@ assert la1[0] == i;
  //@ assert la2[1] == i + 1;
  //@ assert la3[2] == i + 2;
}
```

Figure 10.4: C code of a program with arrays defined as local variables

```
1  ....
2  lg:
3  .LFB0:
4          .cfi_startproc
5  ....
6          movl    %edi, -52(%rbp)
7          movl    -52(%rbp), %eax
8          movl    %eax, -16(%rbp)
9          movl    -52(%rbp), %eax
10         addl    $1, %eax
11         movl    %eax, -28(%rbp)
12         movl    -52(%rbp), %eax
13         addl    $2, %eax
14         movl    %eax, -40(%rbp)
15         movl    -16(%rbp), %eax
16 #APP
17         /* assert #intpointer#%eax#[0] == #int#-52(%rbp)#;*/
18 #NO_APP
19         movl    -32(%rbp), %eax
20 #APP
21         /* assert #intpointer#%eax#[1] == #int#-52(%rbp)# + 1;*/
22 #NO_APP
23         movl    -48(%rbp), %eax
24 #APP
25         /* assert #intpointer#%eax#[2] == #int#-52(%rbp)# + 2;*/
26 #NO_APP
27         ret
28         .cfi_endproc
```

Figure 10.5: Assembly code in SSE2 mode of Figure 10.4 example

assembly code is at `-32(%rbp)`, so in Why, it will be `(_rbp - 32)`. `la2[1]` then will be `(_rbp + (-32+1*4))` or `(_rbp - 28)` where 4 is the size of `int` (corresponds to the second operand in line 11 of Figure 10.5).

```
/*@ requires \valid_range(t,0,n-1);
  @ ensures \forall integer k; 0<=k<n ==> t[k] = 1.0;
  @*/
void init(double t[],int n){
  int i;
  for (i=0;i<n;i++)
    t[i] = 1.0;
}
```

Figure 10.6: A C program with an array defined as an argument of a function

```
1  init:
2  .LFB0:
3          .cfi_startproc
4  ....
5          movq    %rdi, -24(%rbp)
6          movl    %esi, -28(%rbp)
7  #APP
8  /* requires \valid_range(#doublepointer#-24(%rbp)#,0,#int#-28(%rbp)#-1);*/
9  #NO_APP
10         movl    $0, -4(%rbp)
11         jmp     .L2
12 .L3:
13         movl    -4(%rbp), %eax
14         cltq
15         salq    $3, %rax
16         addq    -24(%rbp), %rax
17         movabsq $4607182418800017408, %rdx
18         movq    %rdx, (%rax)
19         addl    $1, -4(%rbp)
20 .L2:
21         movl    -4(%rbp), %eax
22         cmpl    -28(%rbp), %eax
23         jl      .L3
24 #APP
25 /* ensures \forall integer k; 0<=k<#int#-28(%rbp)#
26                            => #doublepointer#-24(%rbp)#[k] = 1.0;*/
27 #NO_APP
28         ret
29         .cfi_endproc
```

Figure 10.7: Assembly code in SSE2 mode of Figure 10.6 example

Translation of arrays being arguments of a function

This case is illustrated by the example in Figure 10.6. The inputs of the function `init`: an array `t` in double and the number of elements `n`. An excerpt of the generated assembly code in SSE2 mode is shown in Figure 10.7.

Normally, general-purpose registers are used for storing the address of the arguments of a function. We consider their address is a 64-bit integer. In the example above, `%rdi` (line 5 of

Figure 10.7) contains the address of the array `t`. In order to get this address, in Why, we write `(integer_of_int64 (sel_int64 _rdi))`.

In the C program, the variable `t` must be allocated dynamically in the memory before it is used. In the precondition, it is necessary to specify that this pointer is valid by using `\valid_range(t,0,n-1)`. As discussed in Section 10.2.1, there is a memory space reserved for dynamic allocated variables. The translation of `\valid_range(t,0,n-1)` into Why will be: the address of all elements of `t` must be greater than or equal to `_rbp` (See Figure 10.3). We do not specify that `t` must be less than `_rip` because the array `t` may be a global variable.

10.2.2 Definition of memory model

We define here a memory model which will be used in the next section. This model uses the standard theory of arrays [66]. In order to access a value at an address in the memory, we need a variable having the following type:

```
type 'v memory
```

where `'v` is a type which has already been defined such as `int`, `real`, etc. In our approach, `'v` is `register`.

We also need the following abstract function to access the value in the memory:

```
logic select: 'v memory, int -> 'v
```

The logic function `select` returns a `'v` value at an address specified by an integer in the memory `'v memory`.

Notice that in the standard theory of arrays, there exists a function `store` which stores a value to memory. In the previous version, we used both `select` and `store`. However, the proof obligations are proved with a big value of timeout. The reason is that the automatic provers take time to find the relation between `select` and `store`. In addition, Gappa is unable to use directly the axiomatics about the relation. We found another way to improve this: instead of using `store` which make the proofs slower, we use only `select`. The idea here is to indicate in the post-condition of the parameter all the properties we need: express what is changed and what is not changed in the memory after each modification.

10.2.3 Translation of operands and instructions to Why

Type of memory

The address of a pointer or of the first element of an array being argument of a function will be stored in a general-purpose register. As we consider an address to be an integer, there is no difference between the operations (addition, subtraction, multiplication, etc.) on addresses and the ones on integers. The translation of each instruction will be detailed in this section.

The memory is modeled as a map of `register`. We define the following variable:

```
parameter MEM: register memory ref
```

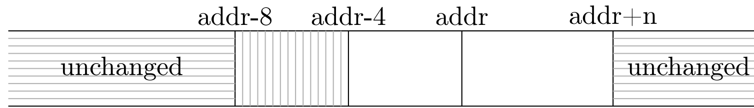
Remind that we have already defined logic functions to get a value from a `register` in Chapter 7 and Chapter 8, we will use it again.

A memory reference $mem = disp(base, index, scale)$ is interpreted as the integer address $[\![mem]\!]_{addr} = base + disp + index \times scale$. The translation of memory reference is as follows:

$$
\begin{aligned}
[\![d(b,i,s)]\!]_{addr} &= [\![b]\!]_{int64}\texttt{+d+s*}[\![i]\!]_{int64}\\
[\![d(\%rbp,i,s)]\!]_{addr} &= \texttt{_rbp + d + s*}[\![i]\!]_{int64}\\
[\![symbol(\%rip)]\!]_{addr} &= \texttt{symbol}\\
[\![mem]\!]_{int32} &= \texttt{integer_of_int32(sel_int32(select(MEM,}[\![mem]\!]_{addr}\texttt{)))}\\
[\![mem]\!]_{int64} &= \texttt{integer_of_int64(sel_int64(select(MEM,}[\![mem]\!]_{addr}\texttt{)))}\\
[\![mem]\!]_{single} &= \texttt{single_value(sel_single(select(MEM,}[\![mem]\!]_{addr}\texttt{)))}\\
[\![mem]\!]_{double} &= \texttt{double_value(sel_double(select(MEM,}[\![mem]\!]_{addr}\texttt{)))}\\
[\![mem]\!]_{exact} &= \texttt{sel_exact(select(MEM,}[\![mem]\!]_{addr}\texttt{))}
\end{aligned}
$$

Remind again that `%rbp` is a special case of the translation as in our model we defined it as a constant (See Section 10.2.1).

In our model, memory is a map of `register` and we assume that the size of this `register` is not fixed (maybe in 32 or in 64 bits depending on the instruction). Therefore, we need some constraints for what is not changed when we update the value at an address.



Let us see the figure above. Assume that we update a `register` with size n bytes at address *addr* in the memory, there are some different cases:

- If the address is less than or equal to *addr-8* (*8* is the number of bytes of a double or 64-bit integer) or is greater than or equal to *addr+n* then the bitvector (not greater than 64 bits) obtained from this address is not changed if we update the value at *addr*.
- If the address is not greater than *addr-4* (*4* is the number of bytes of a single or 32-bit integer) then the single value and 32-bit integer value at this address are not changed if we update the value at *addr*.
- We do not take into account the case where the address is greater than *addr-4* and is less than *addr* because the bitvector (32 or 64 bits) at this address may be changed.

From the idea above, we have the following predicate:

```
predicate unchanged_mem(MEM1: register memory,MEM2:register memory,
                        addr:int,nb:int) =
  (forall i:int. i<= addr-8 or i>=addr+nb -> select(MEM1,i) = select(MEM2,i)
  )
  and
  (forall i:int. i<=addr-4 ->
      integer_of_int32(sel_int32(select(MEM1, i))) =
                integer_of_int32(sel_int32(select(MEM2, i)))
      and
      single_value(sel_single(select(MEM1, i))) =
                single_value(sel_single(select(MEM2, i)))
  )
```

where *MEM1* and *MEM2* are the memory after and before the value at address *addr* is updated.

Here are the translation of some instructions:

Data transfer instructions

We define two parameters *move_mem_to_reg64* and *move_reg_to_mem32* which move the data (in 64 bits and in 32 bits) of a register to memory:

```
parameter move_reg_to_mem64: a:register -> b:int->
{ }
  unit writes MEM
{ integer_of_int64(sel_int64(select(MEM, b)))=integer_of_int64(sel_int64(a))
  and
  double_value(sel_double(select(MEM, b)))=double_value(sel_double(a))
  and
  sel_exact(select(MEM, b))=sel_exact(a)
  and
  unchanged_mem(MEM, MEM@, b, 8)
}

parameter move_reg_to_mem32: a:register -> b:int->
{ }
  unit writes MEM
{ integer_of_int32(sel_int32(select(MEM, b)))=integer_of_int32(sel_int32(a))
  and
  single_value(sel_single(select(MEM, b)))=single_value(sel_single(a))
  and
  sel_exact(select(MEM, b))=sel_exact(a)
  and
  unchanged_mem(MEM, MEM@, b, 4)
}
```

The translation of the `mov` instructions are specified as follows:

$$\begin{array}{lcl}
[\![\text{ movl mem, reg }]\!]_i & = & \texttt{move_cte32} \ [\![mem]\!]_{int32} \ [\![mem]\!]_{single} \ [\![mem]\!]_{exact} \ \texttt{reg} \\
[\![\text{ movl reg, mem }]\!]_i & = & \texttt{move_reg_to_mem32 !reg} \ [\![mem]\!]_{addr} \\
[\![\text{ movq mem, reg }]\!]_i & = & \texttt{move_cte64} \ [\![mem]\!]_{int64} \ [\![mem]\!]_{double} \ [\![mem]\!]_{exact} \ \texttt{reg} \\
[\![\text{ movq reg, mem }]\!]_i & = & \texttt{move_reg_to_mem64 !reg} \ [\![mem]\!]_{addr}
\end{array}$$

The instruction `lea` loads effective address. It computes the effective address of the source operand and stores it in the destination operand. The source operand is a memory address (offset part), the destination operand is a general-purpose register.

$$\begin{array}{lcl}
[\![\text{ leaq mem, reg }]\!]_i & = & \texttt{set_int64} \ [\![mem]\!]_{int64} \ \texttt{reg} \\
[\![\text{ leal mem, reg }]\!]_i & = & \texttt{set_int32} \ [\![mem]\!]_{int32} \ \texttt{reg}
\end{array}$$

The instruction `cltq` extends the 32-bit register `%eax` to the 64-bit register `%rax`. In order to translate this instruction into Why, we declare a parameter:

```
parameter cltq: _:unit ->
{  }
unit writes _rax
{ integer_of_int64(sel_int64(_rax)) = integer_of_int32(sel_int32(_rax@))
  and
  integer_of_int32(sel_int32(_rax)) = integer_of_int32(sel_int32(_rax@)) }
```

Remind that $\overline{\%eax}$ = `_rax`. The translation fo `cltq` is

$[\![$ cltq $]\!]_i$ = `cltq (_)`

Notice that for all instructions, when `dest` is a register, we reuse the Why parameter of Chapter 7 and Chapter 8.

Arithmetic instructions

The instruction `salq imm, dest` multiplies the destination `dest` by 2, `imm` times. In other words, we multiply `dest` by 2^{imm}. We need the following parameter:

```
parameter salq:a:int->b:register ref->
{  }
unit writes b
{ integer_of_int64(sel_int64(b)) = integer_of_int64(sel_int64(b@))*a }
```

The translation of `salq` is as follows:

$[\![$ salq imm, dest $]\!]_i$ = `salq 2pimm dest`

where `2pimm` = 2^{imm}. Notice that this is a bit-level instruction and we can not translate it into Why. Instead, the value `2pimm` is calculated directly by our translator, not in generated Why program.

To store a 32- and 64-bit integer to memory, we need the following parameters:

```
parameter store_imm32: a:int -> b:int->
{ is_int32(a) }
  unit writes MEM
{ integer_of_int32(sel_int32(select(MEM, b))) = a
  and
  unchanged_mem(MEM, MEM@, b, 4)
}

parameter store_imm64: a:int -> b:int->
{ is_int64(a) }
  unit writes MEM
{ integer_of_int64(sel_int64(select(MEM, b))) = a
  and
  unchanged_mem(MEM, MEM@, b, 8)
}
```

We do similarly with `single` and `double`:

```
parameter store_single: a:real -> aexact:real -> b:int->
{ no_overflow_single(nearest_even,a) }
  unit writes MEM
{ single_value(sel_single(select(MEM, b))) = round_single(nearest_even,a)
  and
  sel_exact(select(MEM,b)) = aexact
  and
  unchanged_mem(MEM, MEM@, b, 4)
}
```

```
parameter store_double: a:real -> aexact:real -> b:int->
{ no_overflow_double(nearest_even,a) }
  unit writes MEM
{ double_value(sel_double(select(MEM, b))) = round_double(nearest_even,a)
  and
  sel_exact(select(MEM,b)) = aexact
  and
  unchanged_mem(MEM, MEM@, b, 8)
}
```

The translation of arithmetic instructions are as follows. We present here some general-purpose and SSE/SSE2 instructions:

$\llbracket$ addl reg, mem $\rrbracket_i$ = `store_imm32` ($\llbracket mem \rrbracket_{int32}$ `+` $\llbracket reg \rrbracket_{int32}$) $\llbracket mem \rrbracket_{addr}$
$\llbracket$ addl src, reg $\rrbracket_i$ = `set_int32` ($\llbracket reg \rrbracket_{int32}$ `+` $\llbracket src \rrbracket_{int32}$) `reg`
$\llbracket$ addq src, reg $\rrbracket_i$ = `set_int64` ($\llbracket reg \rrbracket_{int64}$ `+` $\llbracket src \rrbracket_{int64}$) `reg`
$\llbracket$ addq reg, mem $\rrbracket_i$ = `store_imm64` ($\llbracket mem \rrbracket_{int64}$ `+` $\llbracket reg \rrbracket_{int64}$) $\llbracket mem \rrbracket_{addr}$
$\llbracket$ subl reg, mem $\rrbracket_i$ = `store_imm32` ($\llbracket mem \rrbracket_{int32}$ `-` $\llbracket reg \rrbracket_{int32}$) $\llbracket mem \rrbracket_{addr}$
$\llbracket$ subl src, reg $\rrbracket_i$ = `set_int32` ($\llbracket reg \rrbracket_{int32}$ `-` $\llbracket src \rrbracket_{int32}$) `reg`
$\llbracket$ subq src, reg $\rrbracket_i$ = `set_int64` ($\llbracket reg \rrbracket_{int32}$ `-` $\llbracket src \rrbracket_{int64}$) `reg`
$\llbracket$ subq reg, mem $\rrbracket_i$ = `store_imm64` ($\llbracket mem \rrbracket_{int64}$ `-` $\llbracket reg \rrbracket_{int64}$) $\llbracket mem \rrbracket_{addr}$
$\llbracket$ addss reg, mem $\rrbracket_i$ = `store_single` ($\llbracket mem \rrbracket_{single}$ `+` $\llbracket reg \rrbracket_{single}$)
    ($\llbracket mem \rrbracket_{exact}$ `+` $\llbracket reg \rrbracket_{exact}$) $\llbracket mem \rrbracket_{addr}$
$\llbracket$ addss mem, reg $\rrbracket_i$ = `set_single` ($\llbracket reg \rrbracket_{single}$ `+` $\llbracket mem \rrbracket_{single}$)
    ($\llbracket reg \rrbracket_{exact}$ `+` $\llbracket mem \rrbracket_{exact}$) `reg`
$\llbracket$ addsd reg, mem $\rrbracket_i$ = `store_double`($\llbracket mem \rrbracket_{double}$ `+` $\llbracket reg \rrbracket_{double}$)
    ($\llbracket mem \rrbracket_{exact}$ `+` $\llbracket reg \rrbracket_{exact}$) $\llbracket mem \rrbracket_{addr}$
$\llbracket$ addsd mem, reg $\rrbracket_i$ = `set_double` ($\llbracket reg \rrbracket_{double}$ `+` $\llbracket mem \rrbracket_{double}$)
    ($\llbracket reg \rrbracket_{exact}$ `+` $\llbracket mem \rrbracket_{exact}$) `reg`
$\llbracket$ subss reg, mem $\rrbracket_i$ = `store_single`($\llbracket mem \rrbracket_{single}$ `-` $\llbracket reg \rrbracket_{single}$)
    ($\llbracket mem \rrbracket_{exact}$ `-` $\llbracket reg \rrbracket_{exact}$) $\llbracket mem \rrbracket_{addr}$
$\llbracket$ subss mem, reg $\rrbracket_i$ = `set_single` ($\llbracket reg \rrbracket_{single}$ `-` $\llbracket mem \rrbracket_{single}$)
    ($\llbracket reg \rrbracket_{exact}$ `-` $\llbracket mem \rrbracket_{exact}$) `reg`
$\llbracket$ subsd reg, mem $\rrbracket_i$ = `store_double`($\llbracket mem \rrbracket_{double}$ `-` $\llbracket reg \rrbracket_{double}$)
    ($\llbracket mem \rrbracket_{exact}$ `-` $\llbracket reg \rrbracket_{exact}$) $\llbracket mem \rrbracket_{addr}$
$\llbracket$ subsd mem, reg $\rrbracket_i$ = `set_double` ($\llbracket reg \rrbracket_{double}$ `-` $\llbracket mem \rrbracket_{double}$)
    ($\llbracket reg \rrbracket_{exact}$ `-` $\llbracket mem \rrbracket_{exact}$) `reg`

10.2.4 Translation of annotations to Why

The translation of variables in annotations is specified as follows:

$\llbracket \#int\#reg\# \rrbracket_{term} = \llbracket reg \rrbracket_{int32}$
$\llbracket \#single\#reg\# \rrbracket_{term} = \llbracket reg \rrbracket_{single}$
$\llbracket \#double\#reg\# \rrbracket_{term} = \llbracket reg \rrbracket_{double}$

$\llbracket \#intpointer\#mem\# \rrbracket_{term}$ = $\llbracket$`select(MEM,`$\llbracket mem \rrbracket_{addr}$`)`$\rrbracket_{int32}$
$\llbracket \#singlepointer\#mem\# \rrbracket_{term}$ = $\llbracket$`select(MEM,`$\llbracket mem \rrbracket_{addr}$`)`$\rrbracket_{single}$
$\llbracket \#doublepointer\#mem\# \rrbracket_{term}$ = $\llbracket$`select(MEM,`$\llbracket mem \rrbracket_{addr}$`)`$\rrbracket_{double}$

$$
\begin{array}{lcl}
[\![\#intpointer\#reg\#[V]]\!]_{term} & = & [\![\texttt{select(MEM,}[\![reg]\!]_{int64}\texttt{ + }[\![V]\!]_{int32}\texttt{*4)}]\!]_{int32} \\
[\![\#singlepointer\#reg\#[V]]\!]_{term} & = & [\![\texttt{select(MEM,}[\![reg]\!]_{int64}\texttt{ + }[\![V]\!]_{int32}\texttt{*4)}]\!]_{single} \\
[\![\#doublepointer\#reg\#[V]]\!]_{term} & = & [\![\texttt{select(MEM,}[\![reg]\!]_{int64}\texttt{ + }[\![V]\!]_{int32}\texttt{*8)}]\!]_{double}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\#intpointer\#symbol\#[V]]\!]_{term} & = & [\![\texttt{select(MEM,}symbol\texttt{ + }[\![V]\!]_{int32}\texttt{*4)}]\!]_{int32} \\
[\![\#singlepointer\#symbol\#[V]]\!]_{term} & = & [\![\texttt{select(MEM,}symbol\texttt{ + }[\![V]\!]_{int32}\texttt{*4)}]\!]_{single} \\
[\![\#doublepointer\#symbol\#[V]]\!]_{term} & = & [\![\texttt{select(MEM,}symbol\texttt{ + }[\![V]\!]_{int32}\texttt{*8)}]\!]_{double}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\#intpointer\#d(\%rbp)\#[V]]\!]_{term} & = & [\![\texttt{select(MEM,_}rbp\texttt{ + }d\texttt{ + }[\![V]\!]_{int32}\texttt{*4)}]\!]_{int32} \\
[\![\#singlepointer\#d(\%rbp)\#[V]]\!]_{term} & = & [\![\texttt{select(MEM,_}rbp\texttt{ + }d\texttt{ + }[\![V]\!]_{int32}\texttt{*4)}]\!]_{single} \\
[\![\#doublepointer\#d(\%rbp)\#[V]]\!]_{term} & = & [\![\texttt{select(MEM,_}rbp\texttt{ + }d\texttt{ + }[\![V]\!]_{int32}\texttt{*8)}]\!]_{double}
\end{array}
$$

where V is the index of an element in the array, V can be either a register, a symbol or a constant in 32 bits. If V is in 64 bits then we replace $[\![V]\!]_{int32}$ by $[\![V]\!]_{int64}$.

$$
\begin{array}{lcl}
[\![\backslash valid_range(\#intpointer\#p\#,i,j)]\!]_{term} & = & \texttt{forall k:int. i<=k<=j ->} \\
 & & \texttt{integer_of_int64(sel_int64(p)) + k*4 >= _rbp} \\
[\![\backslash valid_range(\#singlepointer\#p\#,i,j)]\!]_{term} & = & \texttt{forall k:int. i<=k<=j ->} \\
 & & \texttt{integer_of_int64(sel_int64(p)) + k*4 >= _rbp} \\
[\![\backslash valid_range(\#doublepointer\#p\#,i,j)]\!]_{term} & = & \texttt{forall k:int. i<=k<=j ->} \\
 & & \texttt{integer_of_int64(sel_int64(p)) + k*8 >= _rbp}
\end{array}
$$

For example, the translation of the annotation `#int#a#[2] == 3` into Why is as below:
$[\![\#int\#a\#[2] == 3]\!]_{term}$ = `integer_of_int32(sel_int32(select (MEM, a + 2*4))) = 3`

10.3 When local variables are pointed by %rsp

For many compilers, `%rbp` is used to point to local variables. However, in some cases, `%rsp` is used instead of `%rbp`. In this section, we will discuss about this case.

A piece of assembly function `f` which uses `%rsp` to point to local variables looks like:

```
f:
   subq  $32, %rsp
....
   movsd %xmm0, (%rsp)
   movsd %xmm1, 8(%rsp)
   movsd %xmm2, 16(%rsp)
   movsd %xmm3, 24(%rsp)
   (preconditions)
....
   addq $32, %rsp
```

At the beginning of `f`, `%rsp` is decreased by `n = 32` bytes where `n` is the total bytes of local variables of the function. After subtracting n from `%rsp`, the displacement of memory reference pointed by `%rsp` is a non-negative integer. Then, the value of parameters of the function is copied to local variables pointed by `%rsp`. The pre-condition generated is put right after these data transfer instructions. At the end of the function `f`, the value of `%rsp` is restored by adding with `n`.

Now let us go back to the memory model, we will declare the Why variable corresponding to `%rsp` as follows:

```
parameter _rsp: int ref
```

and add to each Why function generated from this function a condition `assume _rsp <= 0`.

The translation of the operand containing `%rsp` will be:

$[\![d(\%rsp, i, s)]\!]_{addr}$ = `_rsp + d + s*`$[\![i]\!]_{int64}$

The translation of annotations containing `%rsp` will be:

$[\![\#intpointer\#d(\%rsp)\#[V]]\!]_{term}$ = $[\![$`select(MEM,_rsp + `d` + `$[\![V]\!]_{int32}$`*4)`$]\!]_{int32}$
$[\![\#singlepointer\#d(\%rsp)\#[V]]\!]_{term}$ = $[\![$`select(MEM,_rsp + `d` + `$[\![V]\!]_{int32}$`*4)`$]\!]_{single}$
$[\![\#doublepointer\#d(\%rbp)\#[V]]\!]_{term}$ = $[\![$`select(MEM,_rsp + `d` + `$[\![V]\!]_{int32}$`*8)`$]\!]_{double}$
where V is the index of an element in the array, V can be either a register, a symbol or a constant in 32 bits. If V is in 64 bits then we replace $[\![V]\!]_{int32}$ by $[\![V]\!]_{int64}$.

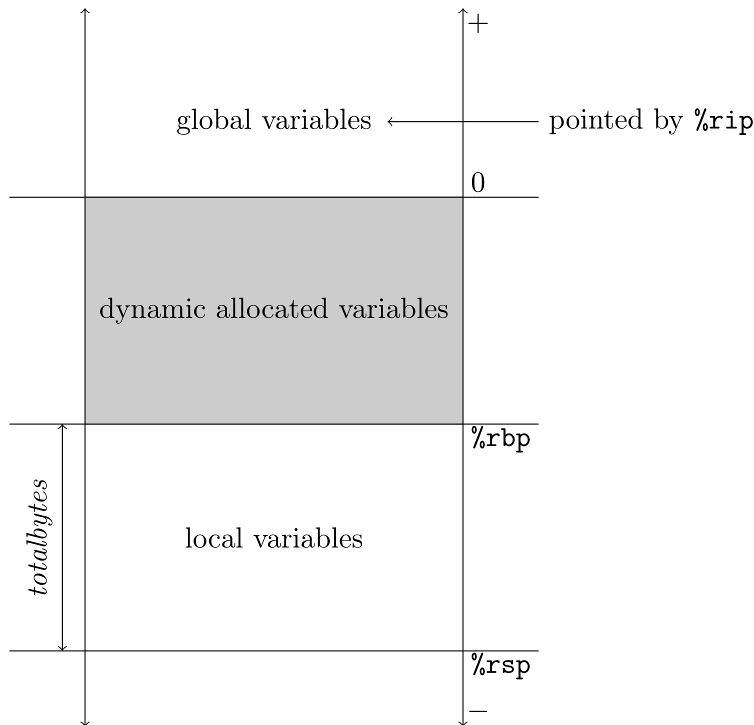
$[\![\backslash valid_range(\#intpointer\#p\#,i,j)]\!]_{term}$ = `forall k:int. i<=k<=j ->`
`integer_of_int64(sel_int64(p)) + k*4 >= _rsp + `*totalbytes*
$[\![\backslash valid_range(\#singlepointer\#p\#,i,j)]\!]_{term}$ = `forall k:int. i<=k<=j ->`
`integer_of_int64(sel_int64(p)) + k*4 >= _rsp + `*totalbytes*
$[\![\backslash valid_range(\#doublepointer\#p\#,i,j)]\!]_{term}$ = `forall k:int. i<=k<=j ->`
`integer_of_int64(sel_int64(p)) + k*8 >= _rsp) + `*totalbytes*

where *totalbytes* is not a Why variable. It is `imm` in the instruction `subq $imm, %rsp` of the current function. If the function does not have this instruction then *totalbytes = 0*. Also notice that in the same program, the value of *totalbytes* is changed from function to function. Our translator will replace *totalbytes* by a constant in the translation `\valid_range` into Why.

The Why function corresponding to the function `f` above is:

```
let f =
   assume _rsp <=0
   [[ subq $32, %rsp ]]_i
   ....
   [[ movsd %xmm0, (%rsp) ]]_i
   [[ movsd %xmm1, 8(%rsp) ]]_i
   [[ movsd %xmm2, 16(%rsp) ]]_i
   [[ movsd %xmm3, 24(%rsp) ]]_i
   assume [[preconditions]]_term
   ....
   [[ addq $32, %rsp ]]_i
void
```

The figure below is the same as the one in subsection 10.2.1. Before executing the instruction
`subq $`*totalbyte*`, %rsp`
the position of `%rsp` and `%rbp` is the same. After this subtraction instruction, `%rbp = %rsp + `*totalbytes*. This is the reason why in the translation of `\valid_range`, instead of using `_rbp`, we use `_rsp + `*totalbytes*.



10.4 Examples

10.4.1 Maximum of an array

The example in Figure 10.8 finds the maximum floating-point value in an array. It was presented in subsection 9.4.3. This first simple example has no floating-point computation but it has an array being input of the function. Therefore, we can illustrate how to apply our memory model to prove it. Moreover, we also show that we can handle many complex statements: `goto`, `do while` and `if` in the same program.

The assembly code of this example is in Figure 10.9. The address of the first element of the array `t` is stored in `-40(%rbp)` (line 5). The statement `m = t[i]` in the source code corresponds to line 27 – 32 in assembly code. There, `-4(%rbp)` (corresponds to `i`) is copied to `%eax` (line 27). In line 28, the instruction `cltq` extends `%eax` in 32 bits to `%rax` in 64 bits. Then in line 29, the instruction `salq` shifts `%rax` to the right by 3 bits, this means that `%rax = %rax *` 2^3 where 2^3 `= 8` is the number of bytes of a double. In line 30, `%rax` receives the address of `t[i]`, that is the address of the first element of `t` plus `i*8`. Then in line 31, the new value of `%rax` is the value obtained at the address stored in `%rax` (old value of `%rax`). Finally, in the line 32, the value of `%rax` is copied to `-24(%rbp)`(corresponds to `m`).

The CFG of the assembly code in Figure 10.9 was presented in Figure 9.7. By using Algorithm 9.2, the four Why functions are generated:

- $0 \to 1 \to 2 \to 3 \to 8 \to 6 \to 9 \to 10$
- $10 \to 11 \to 12 \to 4 \to 5 \to 6 \to 9 \to 10$
- $10 \to 11 \to 12 \to 4 \to 7 \to 3 \to 8 \to 6 \to 9 \to 10$
- $10 \to 11 \to 13 \to 14 \to 15 \to 16 \to 17$

```
/*@ requires n > 0 && \valid_range(t,0,n-1);
  @ ensures \forall integer k; 0<=k<n ==> \result >= t[k];
  @ */
double max_array( double t[] , int n ) {
  double m; int i = 0;
  goto L;
  do{
    if(t[i] > m){
    L:
      m = t[i];
    }
    //@ assert m >=t[i];

    /*@ loop invariant 0 <= i < n &&
      @              \forall integer k; 0<=k<=i ==> m >= t[k];
      @ */

    i = i +1;
  }while(i < n);

  return m;
}
```

Figure 10.8: Maximum of an array program

By analyzing its assembly code (generated by `gcc -S`), this example is translated into four Why functions which are completely and automatically proved by Alt-Ergo and CVC3.

10.4.2 Scalar Product

This example illustrates how we combine floating-point analysis with other features such as loops and arrays. It is presented in Section 5.5 and the same code source is in Figure 10.10.

In this example, the role of the preconditions `\valid_range(x,0,n-1)` and `\valid_range(y,0,n-1)` is important as these preconditions help to separate `x[i]` and `p` and also to separate `y[i]` and `p`. This works because:

- $\mathrm{addr}(x + i \times 8) \geq$ `_rbp` for all i such that $0 \leq i \leq n-1$ (8 is the size (in bytes) of a double)
- $\mathrm{addr}(y + i \times 8) \geq$ `_rbp` for all i such that $0 \leq i \leq n-1$
- $\mathrm{addr}(p) <$ `_rbp`

The table below displays the value of `B` (found by Gappa tool) in function of `NMAX` and the architecture-dependent settings.

```
1  max_array:
2  .LFB0:
3          .cfi_startproc
4  ....
5          movq    %rdi, -40(%rbp)
6          movl    %esi, -44(%rbp)
7  #APP
8  /*requires #int#-44(%rbp)# > 0 &&
9              \valid_range(#doublepointer#-40(%rbp)#,0,#int#-44(%rbp)#-1); */
10 #NO_APP
11         movl    $0, -4(%rbp)
12         jmp     .L2
13 .L4:
14         movl    -4(%rbp), %eax
15         cltq
16         salq    $3, %rax
17         addq    -40(%rbp), %rax
18         movsd   (%rax), %xmm0
19         ucomisd -24(%rbp), %xmm0
20         seta    %al
21         testb   %al, %al
22         je      .L3
23 .L2:
24 #APP
25 /*labelL: */
26 #NO_APP
27         movl    -4(%rbp), %eax
28         cltq
29         salq    $3, %rax
30         addq    -40(%rbp), %rax
31         movq    (%rax), %rax
32         movq    %rax, -24(%rbp)
33 .L3:
34 #APP
35 /*assert #double#-24(%rbp)# >=#doublepointer#-40(%rbp)#[#int#-4(%rbp)#];*/
36 /*loop invariant  ....;*/
37 #NO_APP
38         addl    $1, -4(%rbp)
39         movl    -4(%rbp), %eax
40         cmpl    -44(%rbp), %eax
41         jl      .L4
42         movq    -24(%rbp), %rax
43         movq    %rax, -16(%rbp)
44         nop
45 .L5:
46 #APP
47 /*ensures \forall integer k;0<=k<#int#-44(%rbp)#
48                 ==> #double#-16(%rbp)# >= #doublepointer#-40(%rbp)#[k];*/
49 #NO_APP
50         movq    -16(%rbp), %rax
51         movq    %rax, -56(%rbp)
52         movsd   -56(%rbp), %xmm0
53         ret
54         .cfi_endproc
```

Figure 10.9: Assembly code in SSE2 mode of Figure 10.8 example

Arch.		NMAX		
		10	100	1000
SSE2	**-O0**	`0x1.1p-50`	`0x1.02p-47`	`0x1.004p-44`
x87	**-O0**	`0x1.0022p-50`	`0x1.0021p-47`	`0x1.00201p-44`
x87	**-O2**	`0x1.1p-61`	`0x1.02p-58`	`0x1.004p-55`
FMA	**-O2**	`0x1p-50`	`0x1p-47`	`0x1p-44`

The SSE2 mode, supposed to be strictly compliant with the standard, is worse than FMA and x87, because the roundings in FMA and x87 are slightly more precise.

The x87 without optimization is better than the SSE2 mode. The reason is that the value `p` is calculated in 80 bits and then it is rounded to 64 bits before entering the next step of the loop `for`.

The case of FMA is even better than the two previous modes because `p + x[i]` $\times$ `y[i]` is computed with a single rounding.

The improvement with x87 with optimization is impressive: around $2^{11} \simeq 2000$ times better. The reason is that optimization makes the value of p stored into the x87 stack thus with extended 80-bits precision for the complete execution of the loop: no intermediate rounding to 64-bit is done.

```
#define NMAX 1000
#define NMAXR 1000.0
#define B 0x1.f57d5p-44

/*@ lemma bound_int_to_real:
  @   \forall integer i; i <= NMAX ==> i <= NMAXR; */

/*@ lemma triangular_inequality:
  @   \forall real x,y,z; \abs(x-z) <= \abs(x-y) + \abs(y-z); */


/*@ requires 0 <= n <= NMAX;
  @ requires \valid_range(x,0,n-1) && \valid_range(y,0,n-1) ;
  @ requires \forall integer i; 0 <= i < n ==>
  @          \abs(x[i]) <= 1.0 && \abs(y[i]) <= 1.0 &&
  @          x[i] == \exact(x[i]) && y[i] ==\exact(y[i]) ;
  @ ensures \abs(\result - \exact(\result)) <= n * B; */
double scalar_product(double x[], double y[], int n) {
  double p = 0.0;
  /*@ loop invariant 0 <= i <= n ;
    @ loop invariant \abs(\exact(p)) <= i;
    @ loop invariant \abs(p - \exact(p)) <= i * B;
    @ loop variant n-i; */
  for (int i=0; i < n; i++) {
    // bounds, needed by Gappa
    //@ assert \abs(x[i]) <= 1.0;
    //@ assert \abs(y[i]) <= 1.0;
    //@ assert \abs(p) <= NMAXR*(1+B) ;

  L:
    p = p + x[i]*y[i];

    // bound on the rounding errors in the statement above, proved by gappa
    /*@ assert \abs(p - (\at(p,L) + x[i]*y[i])) <= B; */

    // the proper instance of triangular inequality to show the main invariant
    /*@ assert \abs(p - \exact(p)) <=
              \abs(p - (\at(p,L) + x[i]*y[i])) +
              \abs((\at(p,L) + x[i]*y[i]) - (\exact(\at(p,L)) + x[i]*y[i]));*/

    // a lemma to show the invariant \abs(\exact(p)) <= i
    /*@ assert \abs(\exact(p)) <=
                     \abs(\exact(\at(p,L))) + \abs(x[i]) * \abs(y[i]);*/

    // a necessary lemma, proved only by gappa
    //@ assert \abs(x[i]) * \abs(y[i]) <= 1.0;
  }
  return p;
}
```

Figure 10.10: Scalar product: annotated code

Chapter 11

Bit-level reasoning

There are numerical programs containing bit-level operations such as negation by XOR, conversion of an integer to a double which were not handled in previous chapters. Finding a model which we can integrate into our translator to prove such programs automatically is a difficult task. Hence, this is a future work. What we can do now is to write manually such programs directly in Why 3 [1] [10]. The goal of this step is to create a bit-level model. Firstly, we will explain why we handle these examples. Secondly, we will present the Why3. Then we will detail how to construct theories in Why 3. Finally, there are examples for illustrating our theories.

11.1 Motivations

Before talking about the goal of this chapter, we present firstly some examples which are not proved by our hardware-dependent approach. These examples containing bit-level operations and they are what we want to prove in this chapter.

11.1.1 Examples of the chapter

Negation by XOR

This example is the computation of the negation of a floating-point number using XOR operation which is often found in assembly code. A C program is shown in Figure 11.1 and its assembly code corresponding to the negation is in Figure 11.2.

```
double neg(double x){
  return -x;
}
```

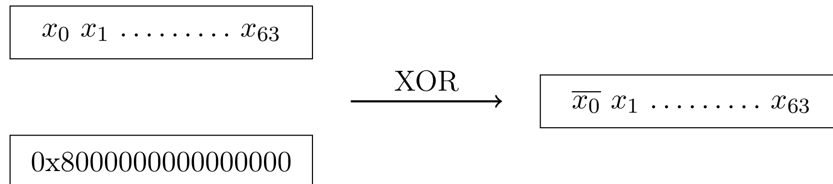
Figure 11.1: Negation of a floating-point number

In this assembly code, `%xmm1` stores the value of `x` (line 1), `%xmm0` contains the value `0x8000000000000000` (line 2) which is represented in 64-bit binary format as `100000...0`. The XOR operation is executed by the instruction `xorpd` (line 3). Return to our hardware-dependent approach, as we do not handle instructions in bit level, this program cannot be proved. In order to compute the negation of a floating-point number `x`:

[1] http://why3.lri.fr/

```
1       movsd       -8(%rbp), %xmm1
2       movsd       .LC0(%rip), %xmm0
3       xorpd       %xmm1, %xmm0
4   ....
5   .LC0:
6       .long       0
7       .long       -2147483648
```

Figure 11.2: Excerpt of assembly code in SSE mode of Figure 11.1 example



where x_0 is the sign bit of the floating-point number x.

Figure 11.3: Negation of a double by XOR

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

union{double d; int i[2];} Var, Const;

int double_of_int(int x){

   int j = 0x43300000;
   Const.i[1] = j; Var.i[1] = j;
   j = 0x80000000;
   Const.i[0] = j; Var.i[0] = j^x;
   double f = Var.d - Const.d;

   assert (f == (double)x);
}
```

Figure 11.4: Example about conversion of an integer to a double

- `x XOR 0x80000000` (if `x` is a single) and
- `x XOR 0x8000000000000000` (if `x` is a double).

It is illustrated in Figure 11.3.

Conversion of an integer to double

This example is presented by a C program in Figure 11.4. Notice that this example is tested in a little endian architecture. The definition of endianness and the difference between little endian and big endian will be talked in the next section.

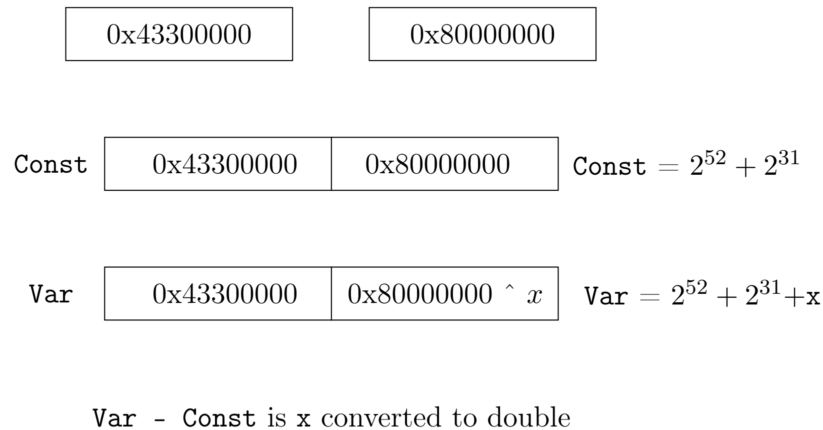


Figure 11.5: Convert an integer to a double

The idea of this example is to convert an integer to a double, shown in Figure 11.5. We have two integers: `j = 0x43300000` and `j' = 0x80000000`. The value of `Const` is obtained by concatenating `j` and `j'`. The value of `Const` is `Const =` $2^{52} + 2^{31}$. `Var` is the concatenation of `j` and `(j' XOR x)`. The value of `Var` is `Var =` $2^{52} + 2^{31}$+x. Then, `Var - Const` is the value of `x` in double.

We cannot prove the examples above with our hardware-dependent approach because this approach did not handle bit-level operations. If we change the translation in our approach in order to prove them, this means that the translation of each instructions must be done in bitvectors, all the examples from Chapter 7 to Chapter 10 will not be proved anymore.

11.1.2 Goals of the chapter

The goal of this chapter is to do an experience in Why 3: We will implement examples directly in Why 3 and see what is possible to prove with bitvectors.

Also notice that in this chapter, we will use Why 3 instead of Why 2 [2] as in Why 3, it is easy to prove a part of obligations in Coq and with automatic provers. Moreover, in Why 3, we can use or clone a theory, this helps to write Why 3 code clearer.

There does not exist the Why 3 standard library for bitvector. Writing a bitvector library in Why 3 is also a goal of this chapter.

11.2 About endianness

Endianness [3] is the format indicating how multi-byte data is stored in computer memory. It describes the location of the most significant byte (MSB) and least significant byte (LSB) of an address in memory.

There are two types of endianness:

- Little endian: stores the LSB at the lowest address. The architectures supporting the little endian are Intel 80x86, DEC Alpha, etc.

[2] The Why syntax that we use from Chapter 2 to Chapter 10 is called Why 2

[3] http://en.wikipedia.org/wiki/Endianness
http://www.intel.com/design/intarch/papers/endian.pdf

- Big endian: stores the MSB at the lowest address. The architectures using this type of endianness are Motorola 68k, IBM Power, etc.

Besides, there are architectures such as ARM, PowerPC, etc. which feature a setting which allow to switch to either big or little endian (Bi-Endian) by setting a processor register.

Here is an example illustrating the difference between little endian and big endian: consider a 32-bit integer (in hex) `0xabcdef12`. It consists of 4 bytes: `ab`, `cd`, `ef` and `12`. Assuming that we store this 32-bit integer at the memory address starting `1000`. Then, on little and big endian system, the memory will be:

Address	**Little endian**	**Big endian**
1000	`12`	`ab`
1001	`ef`	`cd`
1002	`cd`	`ef`
1003	`ab`	`12`

Let us go back to example in Figure 11.4, now we assume that the memory address of `Const` starting at `1000` and with both little and big endian, the memory of `Const` will be:

	i[1]				**i[0]**			
Little endian	43	30	00	00	80	00	00	00
Big endian	00	00	30	43	00	00	00	80
Address	1007	1006	1005	1004	1003	1002	1001	1000

With little endian, the value of `Const` = $2^{52} + 2^{31}$. However, the value of `Const` with big endian does not. In order to get the same value `Const` = $2^{52} + 2^{31}$ with big endian, we just swap the value of `i[0]` and `i[1]`, the memory is as below:

	i[1]				**i[0]**			
Big endian	00	00	00	80	00	00	30	43
Address	1007	1006	1005	1004	1003	1002	1001	1000

In big endian architecture, we also swap the value of `i[0]` and `i[1]` of the variable `Var`.

We do not intend to handle the endianness in this chapter. However, it is necessary to know a little about it because it is related to the the bitvector. We consider in the next sections only little endian. The big endian is handled similarly.

11.3 Why 3

Why 3 is the next generation of Why platform. It clearly separates the purely logical specification part from generation of verification conditions for programs. In this section, we present logic declarations and theories in Why 3 that we will use in the next sections.

11.3.1 Logic

The logic in Why 3 is a first-order logic with polymorphic types with several extensions: recursive definitions, algebraic data types and inductive predicates.

Types

A type can be non-interpreted, an alias for a type expression or an algebraic data type, such as

```
type bv
type bool = True | False
```

Like Why 2, in Why 3 we can declare an abstract type `bv` which represents a bitvector type. The type `bool` is an abstract type with two values `True` and `False`.

Function and predicate symbols

Every function or predicate symbol in Why 3 has a type signature. For example:

```
function nth bv int: bool
```

Both functions and predicates can be given definitions, possibly mutually recursive. For example:

```
predicate eq (v1 v2 : bv) =
    forall n:int. 0 <= n < size -> nth v1 n = nth v2 n
```

The predicate `eq` defines that the value of the nth bit of `v1` is equal to the nth bit of `v2` for all values of `n` such that $0 \leq n < size$.

11.3.2 Theories

A Why 3 input is a file organized as a list of theories. A theory is a list of declarations. Declarations introduce new types, functions and predicates, state axioms, lemmas and goals. These declarations can be directly written in the theory or taken from existing theories.

Excerpt of the theory `BitVector` in Figure 11.6 illustrates how to define a theory in Why 3. The detail of this theory will be talked in the next section.

This theory has a list of declarations:

- the size of a bitvector,
- a type `bv` representing a bitvector,
- the function `nth`: returns the value of the bitvector `bv` at a given position,
- functions `bvzero` and `bvone` and the corresponding axioms `Nth_zero` and `Nth_one`: express that `bvzero` is a bitvector with all values being `False` and `bvone` is a bitvector with all value being `True`,
- a predicate `eq`.

`use` and `clone`

When we need to reuse a theory, we use the keyword `use` or `clone`. If we use a theory without duplicate it, we use the keyword `use`. On the contrary, when we clone a theory with the keyword `clone`, we create a local copy of every cloned declaration, and the newly created symbols, despite having the same names, are different from their originals. More precisely:

If a theory T1 is used in another theory T2 then

- the symbols of T1 are shared

```
theory BitVector

  function size : int

  type bv

  axiom size_positive: size >0

  function nth bv int: bool

  function bvzero : bv
  axiom Nth_zero:
    forall n:int. 0 <= n < size -> nth bvzero n = False

  function bvone : bv
  axiom Nth_one:
    forall n:int. 0 <= n < size -> nth bvone n = True

  predicate eq (v1 v2 : bv) =
    forall n:int. 0 <= n < size -> nth v1 n = nth v2 n

  ....
end
```

Figure 11.6: A Why 3 theory example

- the axioms of T1 remain the axioms
- the lemmas of T1 become the axioms
- the goals of T1 are ignored

If a theory T1 is cloned by another theory T2 then

- the declarations of T1 are copied or replaced
- the axioms of T1 remain the axioms or become the lemmas or goals
- the lemmas of T1 become the axioms
- the goals of T1 are ignored

For example:

```
theory BV32

  function size : int = 32

  clone export BitVector with function size = size

end
```

The theory `BV32` above is a clone of the theory `BitVector` in which `size = 32`.

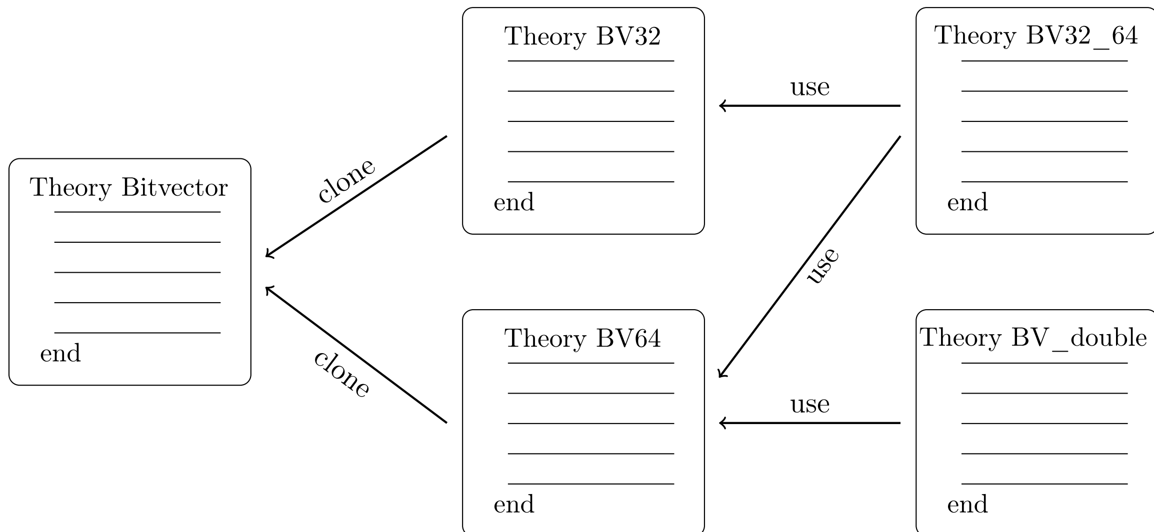


Figure 11.7: Bitvector theories

11.4 Bitvector theories in Why 3

Figure 11.7 contains bitvector theories that we define for this chapter. It illustrates the relations between the theory `BitVector` and the others. Besides, we also define other theories: `Pow2int` `Pow2real` for the exponent of 2 in integer and in real value.

The detail of the theories in Figure 11.7 will be presented below.

11.4.1 Theory BitVector

This theory consists of elements and functions of a bitvector. The detail of this theory is presented in Figure 11.8. It contains:

- the size of a bitvector
- an abstract type `bv`
- the functions about logic operations: bitwise and, bitwise or, bitwise xor, bitwise not, logical shift right/left;
- the function `to_nat_sub b j i`: returns the non-negative integer value of the bitvector `b` from index `i` to `j`;
- the function `from_int n`: returns the bitvector representing an integer `n` on `size` bits;
- the function `from_int2c`: returns the bitvector of an 2-complement integer.

We present below how to define the three functions that we will use in our examples in Section 11.5.

```
theory BitVector

  use export bool.Bool
  use import int.Int
  use import Pow2int
  use import int.EuclideanDivision

  function size : int

  type bv

  function nth bv int: bool

  function bvzero : bv

  function bvone : bv

  function bw_and (v1 v2 : bv) : bv

  function bw_or (v1 v2 : bv) : bv

  function bw_xor (v1 v2 : bv) : bv

  function bw_not (v : bv) : bv

  (* logical shift right *)
  function lsr bv int : bv
  (* arithmetic shift right *)
  function asr bv int : bv
  (* logical shift left *)
  function lsl bv int : bv

  (* conversion to/from integers *)
  (* generalization : (to_nat_sub b j i) returns the non-negative number
     represented by b[j..i] *)

 function to_nat_sub bv int int : int
    (* (to_nat_sub b j i) returns the non-negative integer whose
       binary repr is b[j..i] *)
....
  (* (from_int n) returns the bitvector representing the integer n on
     size bits. *)
  function from_int (n:int) : bv
....
(*from_int2c: int -> bv take an integer as input and returns a bv
                                                 with 2-complement*)
  function from_int2c (n:int) : bv
....
end
```

Figure 11.8: BitVector theory

Function to_nat_sub

The function `to_nat_sub(b:bv, j:int, i:int):int` is defined recursively by:

$$\text{to_nat_sub b j i} = \begin{cases} \text{to_nat_sub b (j - 1) i} & \text{if } \; 0 \leq i \leq j < size \text{ and nth b j} = \text{False} \\ 2^{j-i} + \text{to_nat_sub b (j - 1) i} & \text{if } \; 0 \leq i \leq j < size \text{ and nth b j} = \text{True} \\ 0 & \text{if } \; i > j \end{cases}$$

For example, if we have a 8-bit bitvector `b = 10101010` then

`to_nat_sub b 3 0` $= 2^3 +$ `to_nat_sub b 2 0` (nth b 3 = True)
$= 2^3 +$ `to_nat_sub b 1 0` (nth b 2 = False)
$= 2^3 + 2^1 +$ `to_nat_sub b 0 0` (nth b 1 = True)
$= 2^3 + 2^1 +$ `to_nat_sub b -1 0` (nth b 0 = False)
$= 2^3 + 2^1$ (0 > -1)
$= 10$

The corresponding function [4] in Why 3 is presented as follows:

```
function to_nat_sub bv int int : int
  (* (to_nat_sub b j i) returns the non-negative integer
      whose binary repr is b[j..i] *)

 axiom to_nat_sub_zero :
   forall b:bv, j i:int.
     0 <= i <= j < size ->
       nth b j = False ->
         to_nat_sub b j i = to_nat_sub b (j-1) i

 axiom to_nat_sub_one :
   forall b:bv, j i:int.
     0 <= i <= j < size ->
       nth b j = True ->
         to_nat_sub b j i = (pow2 (j-i)) + to_nat_sub b (j-1) i

 axiom to_nat_sub_high :
   forall b:bv, j i:int.
     i > j ->
       to_nat_sub b j i = 0
```

Function from_int

The function `from_int(n:int):bv` returns the bitvector representing the non-negative integer `n` on `size` bits. It is defined by: For all integers `n`, `i` such as $0 \leq i < size$

- If $(n/2^i)\%2 = 0$ then `nth (from_int n) i = False`
- If $(n/2^i)\%2 <> 0$ then `nth (from_int n) i = True`

The corresponding Why function is shown below:

[4] Notice that Why 3 does not allow defining functions recursively on integers, this is the reason why we use axioms instead of defining recursive functions.

```
function from_int (n:int) : bv

axiom nth_from_int_high_even:
   forall n i:int. size > i >= 0 /\ mod (div n (pow2 i)) 2 = 0 ->
       nth (from_int n) i = False

axiom nth_from_int_high_odd:
   forall n i:int. size > i >= 0 /\ mod (div n (pow2 i)) 2 <> 0 ->
       nth (from_int n) i = True
```

Function from_int2c

The function `from_int2c (n:int):bv` takes a signed integer `n` as input and returns a bitvector `bv` with two's complement representation. It is defined by:

- For sign bit: For all integer n:
  - If $n \geq 0$ then `nth (from_int2c n) (size-1) = False`
  - If $n < 0$ then `nth (from_int2c n) (size-1) = True`
- For all integer n, i such as $0 \leq i < size - 1$
  - If $(n/2^i)\%2 = 0$ then `nth (from_int n) i = False`
  - If $(n/2^i)\%2 <> 0$ then `nth (from_int n) i = True`

The declaration of this function in Why is below:

```
axiom nth_sign_positive:
   forall n :int. n>=0 -> nth (from_int2c n) (size-1) = False

axiom nth_sign_negative:
   forall n:int. n<0 -> nth (from_int2c n) (size-1) = True

axiom nth_from_int2c_high_even:
   forall n i:int. size-1 > i >= 0 /\ mod (div n (pow2 i)) 2 = 0
                        -> nth (from_int2c n) i = False

axiom nth_from_int2c_high_odd:
   forall n i:int. size-1 > i >= 0 /\ mod (div n (pow2 i)) 2 <> 0
                        -> nth (from_int2c n) i = True
```

The logic functions `div` and `mod` are the integer division (which rounds down) and the modulo functions. The function `pow2 i`, which is declared in theory `Pow2int`, return the integer value of 2^i.

11.4.2 Theory BV32 and BV64

The theory `BV64`, presented in Figure 11.9 and the theory `BV32` is the similar one. The theory `BV32(64)` declares a theory for bitvector in 32(64) bits. It is a clone version of the theory `BitVector` with `size = 32(64)`.

```
theory BV64

  function size : int = 64

  clone export BitVector with function size = size

end
```

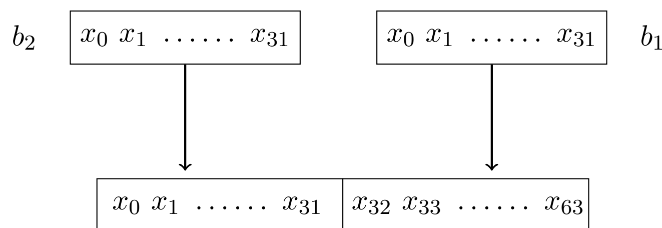
Figure 11.9: BV64 theory

```
theory BV32_64

  use import int.Int
  use BV32
  use BV64

  function concat BV32.bv BV32.bv : BV64.bv

  axiom concat_low: forall b1 b2:BV32.bv.
    forall i:int. 0<=i<32 -> BV64.nth (concat b1 b2) i = BV32.nth b2 i

  axiom concat_high: forall b1 b2:BV32.bv.
    forall i:int. 32<=i<64 -> BV64.nth (concat b1 b2) i = BV32.nth b1 (i-32)

end
```

Figure 11.10: Bv32_64 theory

11.4.3 Theory BV32_64

The theory `BV32_64` (See Figure 11.10) use the theories `BV32` and `BV64`. The function `concat` in this theory concatenates two 32-bit bitvectors to create a 64-bit bitvector.



As shown in the figure above, b_1 and b_2 are two 32-bit bitvectors. After concatenating, b_2 becomes the lower part and b_1 becomes the higher part of a 64-bit bitvector.

11.4.4 Theory BV_double

The theory `BV_double` contains declarations for converting a 64-bit bitvector to a double. It uses the theory `BV64` (see Figure 11.11). In this theory, we do not take into account special

```
theory BV_double

  use import BV64
  use import real.RealInfix
  use import int.Int
  use import Pow2real
  use import real.FromInt

  function double_of_bv64 (b:bv) : real

  function exp (b:bv) : int = BV64.to_nat_sub b 62 52

  function mantissa (b:bv) : int = BV64.to_nat_sub b 51 0

  function sign (b:bv) : bool = BV64.nth b 63

  function sign_value(s:bool):real

  axiom sign_value_false:
        sign_value(False) = 1.0

  axiom sign_value_true:
        sign_value(True) = -.1.0

  axiom zero : forall b:bv.
     exp(b) = 0 /\ mantissa(b) = 0 -> double_of_bv64(b) = 0.0

  axiom sign_of_double_positive:
     forall b:bv. sign b = False -> double_of_bv64(b) >=. 0.0

  axiom sign_of_double_negative:
     forall b:bv. sign b = True -> double_of_bv64(b) <=. 0.0

  axiom double_of_bv64_value:
     forall b:bv. 0 < exp(b) < 2047 ->
       double_of_bv64(b) =  sign_value(sign(b)) *.
                            (pow2 ((exp b) - 1023)) *.
                            (1.0 +. (from_int (mantissa b)) *. (pow2 (-52)))

end
```

Figure 11.11: Bv_double theory

---

values (NaNs, infinities). In order to calculate the double value from a bitvector, we use the following formula (presented in Section 2.1).

$$(-1)^{sign} \times 2^{exp-bias} \times (1 + fraction \times 2^{1-prec}) \tag{11.1}$$

where $1 \leq exp \leq 2046$, $bias = 1023$, $prec = 53$. If `exp = 0` and `fraction = 0` then the double value is `0.0`. The function `double_of_bv64 (b:bv):real` returns the value of a bitvector interpreted as a IEEE double precision number by using Formula (11.1). This function needs the following functions:

```
theory TestNegAsXOR

  use import BV64
  use import BV_double
  use import int.Int
  use import bool.Bool
  use import real.RealInfix

  function j : bv = from_int 0x8000000000000000

  lemma MainResultBits : forall x:bv. forall i:int. 0 <= i < 63 ->
                    nth (bw_xor x j) i = nth x i

  lemma MainResultSign : forall x:bv. nth (bw_xor x j) 63 = notb (nth x 63)

  lemma Sign_of_xor_j : forall x:bv. sign(bw_xor x j) = notb (sign x)

  lemma Exp_of_xor_j : forall x:bv. exp(bw_xor x j) = exp(x)

  lemma Mantissa_of_xor_j : forall x:bv. mantissa(bw_xor x j) = mantissa(x)

  lemma MainResultZero : forall x:bv. 0 = exp(x) /\ mantissa(x) = 0 ->
                     double_of_bv64 (bw_xor x j) = -. double_of_bv64 x

  lemma sign_neg:
          forall x:bv. sign_value(notb(sign(x))) = -.sign_value(sign(x))

  lemma MainResult : forall x:bv. 0 < exp(x) < 2047 ->
                     double_of_bv64 (bw_xor x j) = -. double_of_bv64 x

end
```

Figure 11.12: Negation of a double by XOR

- `exp (b:bv):int` returns the value of bitvector `b` in range [52..62] interpreted as a non-negative integer;
- `mantissa (b:bv):int` returns the value of bitvector `b` in range [0..51] interpreted as a non-negative integer;
- `sign (b:bv):bool` returns the value of the sign bit of `b` (63th bit of `b`);
- `sign_value (s:bool):real` returns the value of sign: either `1.0` or `-1.0`.

11.5 Examples

11.5.1 Negation by xor

This example was presented in subsection 11.1.1. The theory `TestNegAsXOR` is presented in Figure 11.12. A screen-shot of the proofs is in Figure 11.13. There are 8 lemmas to prove:

- 7 lemmas proved automatically by Alt-Ergo and

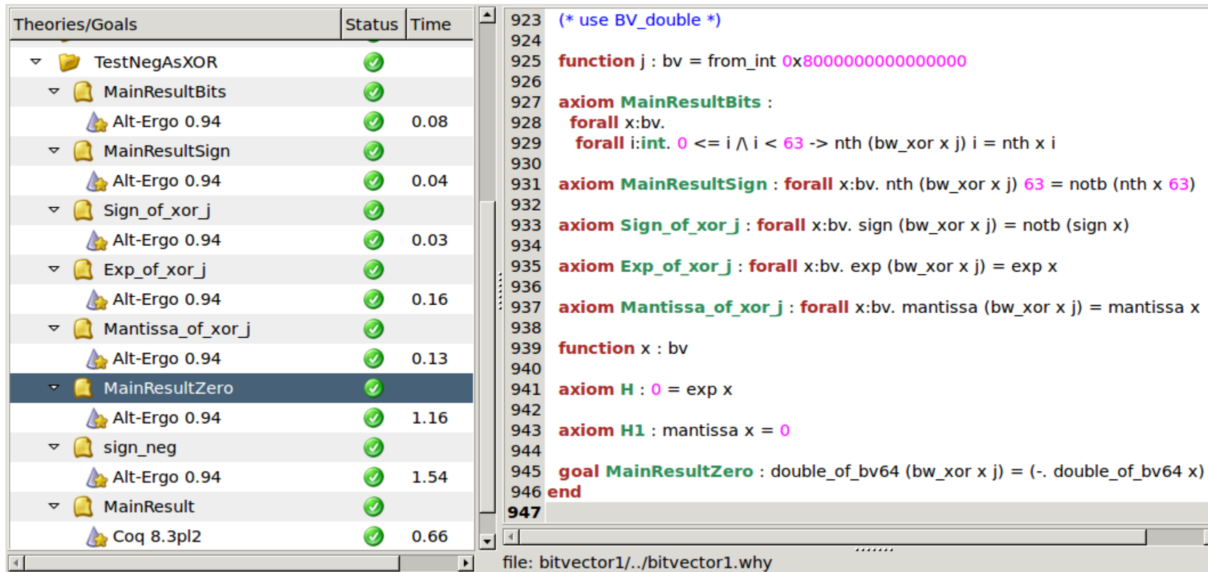


Figure 11.13: Result of Figure 11.12 program

- only one lemma proved using Coq (11 lines).

All these lemmas are described in the following table:

	Lemma	**Description**	**Proved by**
Goals	MainResultZero	The result is `0.0`: when $exp(x) = 0.0$ and $mantissa(x) = 0$	Alt-Ergo
	MainResult	when $0 < exp(x) < 2047$	Coq
Extra lemmas	MainResultBits	The value of each bit in the result bitvector except sign bit	Alt-Ergo
	MainResultSign	The value of sign bit of the result bitvector	
	Sign_of_xor_j	Lemma about the sign of `x XOR j`	
	Exp_of_xor_j	Exponent of the bitvector of `x XOR j`	
	Mantissa_of_xor_j	Mantissa of the bitvector of `x XOR j`	
	sign_neg	A lemma about sign value helps to prove `MainResult`	

11.5.2 Conversion of an integer to a double

This example was presented in subsection 11.1.1. We implement it directly in Why 3. In order to do that, we have three parts:

- Declaration of constants `j` and `j'`
- Proof of `Const` = $2^{52} + 2^{31}$
- Proof of `Var(x)` = $2^{52} + 2^{31} +$ `x`
- Final goal: `Const - Var(x) = (double)x`

The detail of this program is shown below:

```
1   function j : BV32.bv = BV32.from_int 0x43300000
2   function j': BV32.bv = BV32.from_int 0x80000000
3   (*****************************************************************)
4   (* definitions:                                                  *)
5   (*     const : bv64 = concat j j'                                *)
6   (*     const_as_double : real = double_of_bv64(const)            *)
7   (*****************************************************************)
8   function const : BV64.bv = BV32_64.concat j j'
9
10  function const_as_double : real = double_of_bv64 const
11
12  (*****************************************************************)
13  (* next lemma: const_as_double = 2^52 + 2^31                     *)
14  (*****************************************************************)
15  lemma nth_const1: forall i:int. 0 <= i <= 30 -> BV64.nth const i = False
16  lemma nth_const2: BV64.nth const 31 = True
17  lemma nth_const3: forall i:int. 32 <= i <= 51 -> BV64.nth const i = False
18  lemma nth_const4: forall i:int. 52 <= i <= 53 -> BV64.nth const i = True
19  lemma nth_const5: forall i:int. 54 <= i <= 55 -> BV64.nth const i = False
20  lemma nth_const6: forall i:int. 56 <= i <= 57 -> BV64.nth const i = True
21  lemma nth_const7: forall i:int. 58 <= i <= 61 -> BV64.nth const i = False
22  lemma nth_const8: BV64.nth const 62 = True
23  lemma nth_const9: BV64.nth const 63 = False
24
25  lemma sign_const: sign(const) = False
26
27  lemma exp_const: exp(const) = 1075
28
29  lemma to_nat_mantissa: (BV64.to_nat_sub const 30 0) = 0
30
31  lemma mantissa_const_to_nat51:
32    BV64.to_nat_sub const 51 0 = BV64.to_nat_sub const 31 0
33
34  lemma mantissa_const: mantissa(const) = Pow2int.pow2 31
35
36  lemma const_value0: const_as_double = 1.0*.Pow2real.pow2 (1075 - 1023) *.
37                           (1.0 +. Pow2real.pow2 31 *. Pow2real.pow2 (-52))
38
39  lemma const_value: const_as_double = Pow2real.pow2 52 +. Pow2real.pow2 31
```

Figure 11.14: Declaration and proofs of `Const`

Declaration of constants

We declare two bitvectors `j` and `j'` which are the 32-bit binary representation of two values `0x43300000` and `0x80000000`.

```
function j : BV32.bv = BV32.from_int 0x43300000
function j': BV32.bv = BV32.from_int 0x80000000
```

j = 0x43300000
j' = 0x80000000
var(x) = concat j (j' xor x)

↓

`lemma1`: ∀ integer x, to_nat_sub (j xor x) 31 0 = 2^{31} + x
`lemma2`: ∀ integer x, mantissa(var(x)) = 2^{31} + x
`lemma3`: ∀ integer x, exp(var(x)) = 1075
`lemma4`: ∀ integer x, sign(var(x)) = false

↓

`Goal`: ∀ integer x, var_as_double(x) = $2^{52} + 2^{31}$ + x

Figure 11.15: Lemmas for proving `var(x) =` $2^{52} + 2^{31}$+`x`

Proof of `Const =` $2^{52} + 2^{31}$

The declaration of the variable `Const` and the proofs related to `Const` are shown in Figure 11.14. We declare the function `const` that returns a 64-bit bitvector by concatenating `j` and `j'` (line 8). The function `const_as_double` returns the real value of bitvector `const` (line 10). We need to prove that `Const =` $2^{52}+2^{31}$. The corresponding lemma `const_value` is at line 39. All the lemmas from line 15 – 37 help to prove `const_value`. All the lemmas for proving `Const =` $2^{52} + 2^{31}$ are proved automatically by Alt-Ergo, CVC3 and Z3 except the lemma `exp_const` which is proved using Coq (30 lines). There are 5 other lemmas that help to prove the lemmas above by automatic provers.

Proof of `Var(x) =` $2^{52} + 2^{31}$+ `x`

The steps to prove `Var(x) =` $2^{52}+2^{31}$+ `x` are presented in Figure 11.15. We define the following function

```
function jpxor(x:int): BV32.bv = (BV32.bw_xor j' (BV32.from_int2c x))
```

that returns a 32-bit bitvector of `j' XOR x`. The function `var(x)`

```
function var(x:int): BV64.bv = (BV32_64.concat j (jpxor x))
```

returns a 64-bit bitvector by concatenating `j` and `jpxorx(x)`. The double value of an integer converted to double is defined by the function `var_as_double(x)`:

```
function var_as_double(x:int) : real = double_of_bv64 (var x)
```

The lemma `lemma1` helps to prove the lemma `lemma2`. `lemma2` proves that the fraction value of `var(x)` is 2^{31}+ `x`. The lemmas `lemma2`, `lemma3`, `lemma4` helps to find the double value of `var(x)`.

The lemma `lemma1` is the most difficult one to prove. The proofs of `lemma2`, `lemma3`, `lemma4` are similar to the proofs of fraction, exponent and sign in the case of `const`. The Why code of `lemma1` and others lemmas needed to prove `lemma1` is shown in Figure 11.16.

These lemmas are proved by the combination of automatic provers and interactive prover Coq. They are shown in the following table:

```
1    (***************************************************************)
2    (*  lemma 1: for all integer x, to_nat(jpxor(x)) = 2^31 + x      *)
3    (***************************************************************)
4
5    predicate is_int32(x:int) = - Pow2int.pow2 31 <= x < Pow2int.pow2 31
6
7    (* bits of jpxor x *)
8
9   lemma nth_0_30: forall x:int. forall i:int. is_int32(x) /\  0<=i<=30 ->
10         BV32.nth (BV32.bw_xor j' (BV32.from_int2c x)) i =
11                                               BV32.nth (BV32.from_int2c x) i
12   lemma nth_jpxor_0_30:
13         forall x:int. forall i:int. is_int32(x) /\ 0<=i<=30 ->
14                     BV32.nth (jpxor x) i = BV32.nth (BV32.from_int2c x) i
15   lemma nth_var31: forall x:int.
16         BV32.nth (jpxor x) 31 = notb (BV32.nth (BV32.from_int2c x) 31)
17
18   lemma to_nat_sub_0_30: forall x:int. is_int32(x)->
19        BV32.to_nat_sub (BV32.bw_xor j' (BV32.from_int2c x)) 30 0 =
20                                      BV32.to_nat_sub (BV32.from_int2c x) 30 0
21
22   (* case x >= 0 *)
23
24   lemma jpxorx_pos: forall x:int. x>=0 ->
25         BV32.nth (BV32.bw_xor j' (BV32.from_int2c x)) 31 = True
26
27   lemma from_int2c_to_nat_sub_pos:
28         forall i:int. 0 <= i <= 31 ->
29           forall x:int. 0 <= x <  Pow2int.pow2 i ->
30            BV32.to_nat_sub (BV32.from_int2c x) (i-1) 0 = x
31
32   lemma lemma1_pos : forall x:int. is_int32 x /\ x >= 0 ->
33         BV32.to_nat_sub (jpxor x) 31 0 = Pow2int.pow2 31 + x
34
35   (* case x < 0 *)
36
37   lemma jpxorx_neg: forall x:int. x<0 ->
38         BV32.nth (BV32.bw_xor j' (BV32.from_int2c x)) 31 = False
39
40   lemma from_int2c_to_nat_sub_neg:
41         forall i:int. 0 <= i <= 31 ->
42           forall x:int. -Pow2int.pow2 i <= x <  0 ->
43            BV32.to_nat_sub (BV32.from_int2c x) (i-1) 0 = Pow2int.pow2 i + x
44
45   lemma lemma1_neg : forall x:int. is_int32 x /\ x < 0 ->
46         BV32.to_nat_sub (jpxor x) 31 0 = Pow2int.pow2 31 + x
47
48   (* final lemma *)
49
50   lemma lemma1 : forall x:int. is_int32 x ->
51         BV32.to_nat_sub (jpxor x) 31 0 = Pow2int.pow2 31 + x
```

Figure 11.16: Proofs of `lemma1`

Lemma	Proved by
nth_jpxor_0_30	Alt-Ergo
nth_0_30	
nth_var31	
to_nat_sub_0_30	
jpxorx_pos	
from_int2c_to_nat_sub_pos	60 lines of Coq
lemma1_pos	14 lines of Coq
jpxorx_neg	CVC3, Z3
from_int2c_to_nat_sub_neg	82 lines of Coq
lemma1_neg	14 lines of Coq
lemma1	Alt-Ergo

The final goal to prove is

```
function double_of_int32 (x:int) : real = var_as_double(x) -. const_as_double
lemma MainResult: forall x:int. is_int32 x -> double_of_int32 x = from_int x
```

It is proved automatically by Alt-Ergo and Z3.

11.6 Discussion

We had tried to define a bitvector model in Why 2 and tried to prove the two examples above. As Why 3 allows proving a part of obligations in Coq and in addition, it allows reusing existing theories, we choose Why 3 for this experience.

What we obtained in this chapter are that firstly we defined a bitvector library in Why 3. Secondly, we are successful in proving programs containing bit-level operation in Why 3. With this experience, we had to prove many lemmas in Coq. It is thus too difficult to prove such program only with automatic provers. For this reason, in order to integrate the bitvector model into hardware-dependent approach, the improvement of bitvector theory is needed.

Chapter 12

Conclusion and Future works

12.1 Summary

This thesis demonstrates that the formal proofs of numerical programs in considering architecture and compiler aspect is possible. This possibility is shown by two proposed approaches:

12.1.1 Hardware-independent approach

This approach gives correct rounding errors whatever the architecture and allowing many choices to the compiler. This is implemented in the Jessie plugin of the Frama-C framework for all basic operations: addition, subtraction (with possible reordering), multiplication, division, square root, negation, absolute value.

Moreover, it handles both rounding according to 64-bit rounding in IEEE-754 double precision, 80-bit rounding in x87, double rounding in IA-32 architecture, and FMA in Itanium and PowerPC processors and all possible reorganizations of additions and subtractions.

A drawback of this approach is that we may only prove rounding errors. There is no way to prove, for example, that a computation is correct (even if it would be correct in all possible roundings and compilations). This means that some subtle floating-point properties may be lost but bounding the final rounding error is usually what is wanted by engineers and this does not appear to be a big flaw.

Note that we only consider double precision numbers as they are the most used. This is easily applied to single precision computations the same way (with single rounding, 80-bit rounding or double rounding). The idea would be to give similar formulas and to provide the basic operations with those post-conditions.

We only handle rounding-to-nearest (ties to even and ties away from zero). The reason is that directed roundings do not suffer from these problems: double rounding gives the correct answer and if some intermediate computations are done in 80-bit precision, the final result is more accurate, but still correct as it is always rounded in the correct direction. When additions are reordered, we may have different results, but all are smaller than the exact result, so the wanted property still holds whatever the order. Only rounding-to-nearest causes discrepancies.

This work is at the boundary between software and hardware for floating-point programs and this aspect of formal verification is very important. Moreover, this work deals both with normal and subnormal numbers, the latter ones being usually dismissed.

Another interesting point is that our error bounds may be used by other tools. As shown here, considering a slightly bigger error bound for each operation suffices to give a correct final error. This means that if Fluctuat [26] for example would use them, it would also consider all our cases of hardware and of compilation.

12.1.2 Hardware-dependent approach

Former work on the verification of assembly code are mainly in the context of the so-called *proof-carrying-code*, where proof obligations for *safety* (of memory dereferencing, absence of overflow, etc.) are generated on the object code. However these do not consider any behavioral specification language to specify deeper properties than safety. Thus, we believe that what we have presented is the first method being able to prove architecture- and compiler-dependent behavioral properties of floating-point programs. Our approach and our prototype show that proving complex behaviors of floating-point programs is possible. Our approach is implemented by modifying GAS ($\approx$ 10k lines of C code).

The advantage of this method is that we can prove programs based on their assembly code. For this reason, firstly, it is possible to consider the accuracy of the result depending on the architecture and compiler. Secondly, the precision of each operations is also considered in the calculations. Thirdly, the optimizations are taken into account. Finally, there is no reasoning at the bit-level representation. More importantly, we can prove that the same program may give different result when it is compiled by different architecture/options of compiler.

We handled both single, double and extended rounding. In addition, we also handled FMA instructions. Operations on 32-bit and 64-bit integer were also taken into account. We were successful in proving C programs with complex statement such as `for`, `goto`, `do while`, etc. We supported inlining functions which allow us to prove programs compiled with optimization at `-O2` level. By supporting this, we could find better results of programs.

We had two different models:

- the simple one that ignores the access of memory, dedicates to proving programs without arrays and pointers.
- the memory model which supports for programs containing pointers and arrays.

We also considered a special class of programs containing bit operations such as the negation of a floating-point number by using XOR, the bit-level conversion (integer number from/to floating-point number). This work is still in progress as it is only an experience in Why 3 for finding a way to prove such programs automatically.

However, it is clearly not mature enough for a non-expert user, because there is a lot of open issues. First, some languages features are not supported, like pointer casts at the C level, and also at the assembly level. Second, we are not able to interpret all the compiler optimizations. Besides, there are still constraints for each class of programs we handle.

12.1.3 Comparison with related works

Before comparing with previous works, it is important to talk about the role of Why. Why is an excellent tool which supports many automatic and interactive provers. Thanks to Why, we can prove a program by using a combination of many provers. It is very difficult to use only one prover to prove all proof obligations and none of provers can support many features. For example, Gappa is a good prover which can prove obligations about floating point behaviors while others cannot do that. Otherwise, many tools such as Alt-Ergo, CVC3, Z3, etc. helps us to prove non-floating-point properties. For the reasons above, using Why is an advantage of our work compared to previous works which support only one prover.

About the first approach, we compared it firstly with the strict model and full model. We based on these two models to construct our approach. The difference is that our model consider architecture and compiler aspects while these models support strictly the IEEE-754 standard. Although the bound of each rounding error is higher than the one in previous works, it is true

for multiple architecture. In addition, we consider rounding in 64 bits while the strict model and full model support both 32-bit and 64-bit rounding. Moreover, the full model supports also for special values while we prove that they do not happen like in the strict mode.

Fluctuat is a tool aiming at analyzing the numerical precision and stability of complex algorithms. Its aim is to detect automatically a possible catastrophic loss of precision and its source, or else prove its absence. It relies on abstract domains for the estimation of values and errors, based on interval and affine arithmetic. This tool follows only the IEEE-754 standard while we consider the rounding error for multiple architecture and compiler.

About the second approach, as said before, we are the first ones who can prove numerical programs considering architecture and compiler issues. This is thus the first difference compared to the related works.

The work of Burdy and Pavlova [16] defined a specification language for bytecode called BML (Bytecode Modeling Language) and proposed a verification condition generator for Java bytecode which is completely independent from the source code. They also defined a compiler from JML to BML which allows Java to benefit from the JML source specification (requires not to optimize). Our work is a similar one, dedicates for proving C numerical program, and we use ACSL-style specifications and translate the annotations into assembly without defining a compiler for it. We do not need to define another specification language for assembly. More precisely, we simplify what this work did and we focus on floating-point aspect. Moreover, thanks to Why, we can prove assembly program (compiler optimizations are allowed) with automatic provers while the work of Burdy and Pavlova dedicate to the interactive prover Coq.

Another work that we want to compare with is the one of Myreen in 2008. This work proposed to translate assembly code into recursive functions in the HOL4 system and it did not support for floating-point computation. Our work is a similar one but we translate assembly code of floating-point program into Why intermediate language and the proof obligations can be proved with many different provers.

Compared with the default model of Frama-c/Jessie, proving a program on its assembly code is more complicated than proving it at source level. When we prove a C program on its assembly code, in some cases, we need more information in order to prove it with automatic provers. For example, in the case when we use memory model or when we have a loop invariant in the program.

12.1.4 Difficulties

The difficulty I met is my limited knowledge about assembly language. Indeed, there are a lot of assemblers with different syntaxes. It took me a large amount of time to understand different syntaxes and finally I chose AT&T syntax which is supported by `gcc`. I also took a lot of time to try to write a parser in OCaml for assembly and finally I found that it is a difficult task and not the main goal of the thesis. For this reason, I modified GAS which already had a parser for assembly.

12.2 Future works

12.2.1 Hardware-independent approach

Among the many future works are the numerous other possible compiler optimizations. We have looked a little into multiplication reordering, but, due to underflow, the natural formulas are either wrong or unusable. It is very difficult to only modify the one operation formula to handle all possible underflows in a sequence of multiplications. If we had dismissed underflows,

this would have been easy, but we are still trying to find a correct and useful solution. We are also interested in distributivity, meaning $a \otimes (b \oplus c) \longleftrightarrow (a \otimes b) \oplus (a \otimes c)$, and in the replacing of the division by the multiplication by the inverse: $a \oslash b \longleftrightarrow a \otimes (1 \oslash b)$ (this is known to be incorrect, but may speed up a lot of computations, such as Gaussian elimination [21]). The interaction of all those optimizations with one another should be carefully studied.

12.2.2 Hardware-dependent approach

A simple future work is to consider special values such as NaNs, infinities, etc. A similar work was done by Marché and Ayad [4]. We handled only rounding-to-nearest mode which is a default of `gcc`. Another perspective is to consider other rounding modes.

In this thesis, we had two models: simple one (Chapter 7 to 9) and memory one (Chapter 10). The goal is to use only memory model as the main model for this work. However, we need to find a way to prove program automatically more easily. Until now, we need redundant assertions to help automatic provers to prove it.

An ongoing work is bit-level reasoning. As discussed in Chapter 11, we proved some examples containing bit-level operations in Why 3 such as the negation of a floating-point number, the conversion of an integer number from/to a floating-point number. However, to prove such programs automatically, it is still a hard work. A possible future work is to find a way to do it automatically.

One another work that we would like to do is to see how the assembly code is changed when it is compiled with `gcc -funsafe-math-optimizations` where the operations are re-organized and may be rewritten and we would like to study what happens when a processor allows to vectorize [58]. Our approach may work when the program is compiled with this kind of optimization.

An important limitation is that the translation of annotations by using inline assembly may change assembly code such as the compiler may add more move instructions. Thus it is not a safe way. We wonder if the `assert` statement in C program is a better choice.

Our translator is tested only on Intel-64 architecture with `gcc` compiler. One of the future work is try to test it in other architectures. For example, instead of generating FMA instructions by `gcc` on Intel processors, we can generate assembly code directly by a PowerPC or an Intel Itanium and use our translator to prove it. In order to do that, the translator must be modified because the instructions of PowerPC are different from Intel-64. This work is tedious but direct from our work.

A future work that is not related to floating-point arithmetic is to improve the weakest-precondition algorithm for unstructured programs. We took some time to find out another algorithm but we were not successful. We reused proposed techniques [32, 5] and in some cases we have to add more information for loop invariants.

For longterm perspectives, our ambition is to integrate this approach into a certified compiler. A similar work related to certified compiler was presented by Leroy [49]. Author proposed the formal certification of a compiler from Cminor (a Clike imperative language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its correctness. A future work is to see whether we can either integrate this approach into such a work or reuse it, not dedicate to interactive language like Coq but our goal is for automatic provers. We also intend to improve the robustness and reduce the trusted code base.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–58, 2008. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4610933.

[2] Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual. Volume 6: 128-Bit and 256-Bit XOP and FMA4 Instructions. Specification, 2009.

[3] Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions. Specification, 2011.

[4] A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 127–141, Edinburgh, Scotland, July 2010. Springer.

[5] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. ACM.

[6] M. Barnett, K. R. M. Leino, K. Rustan, M. Leino, and W. Schulte. The Spec# Programming System: An Overview. pages 49–69. Springer, 2004.

[7] C. Barrett and C. Tinelli. CVC3. In Damm and Hermanns [23], pages 298–302.

[8] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.

[9] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.5*, 2011. http://frama-c.cea.fr/acsl.html.

[10] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[11] S. Boldo and J.-C. Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[12] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. In *16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning*, volume 5625 of *Lecture Notes in Artificial Intelligence*, pages 59–74, Grand Bend, Canada, July 2009. Springer.

[13] S. Boldo and C. Marché. Formal verification of numerical programs: from C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 2011.

[14] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.

[15] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[16] L. Burdy and M. Pavlova. Java bytecode specification and verification. In *Symposium on Applied Computing*, pages 1835–1839. ACM, 2006.

[17] V. A. Carreño and P. S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *HOL95: 8th International Workshop on Higher-Order Logic Theorem Proving and Its Applications*, Aspen Grove, UT, Sept. 1995.

[18] D. Clutterbuck and B. Carre. The verification of low-level code. *Software Engineering Journal*, 3:97–111, 1988.

[19] S. Conchon, E. Contejean, and J. Kanig. CC(X): Efficiently combining equality and solvable theories without canonizers. In S. Krstic and A. Oliveras, editors, *SMT 2007: 5th International Workshop on Satisfiability Modulo*, 2007.

[20] The Coq Proof Assistant. http://coq.inria.fr/.

[21] M. Cosnard, J.-M. Muller, Y. Robert, and D. Trystram. Computation costs versus communication costs in parallel Gaussian elimination. In M. C. et al., editor, *Parallel Algorithms and architectures*, pages 19–29. North Holland, 1986.

[22] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, number 3444 in Lecture Notes in Computer Science, pages 21–30, 2005.

[23] W. Damm and H. Hermanns, editors. *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, Berlin, Germany, July 2007. Springer.

[24] M. Daumas, L. Rideau, and L. Théry. A generic library for floating-point numbers and its application to exact computing. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '01, pages 169–184, London, UK, 2001. Springer-Verlag.

[25] F. de Dinechin, C. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions using gappa. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1318–1322, New York, NY, USA, 2006. ACM.

[26] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.

[27] G. Dowek and C. Muñoz. Conflict detection and resolution for 1,2,...,N aircraft. In *Proceedings of the 7th AIAA Aviation, Technology, Integration, and Operations Conference, AIAA-2007-7737*, Belfast, Northern Ireland, 2007.

[28] G. Dowek, C. Muñoz, and V. Carreño. Provably safe coordinated strategy for distributed conflict resolution. In *Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005, AIAA-2005-6047*, San Francisco, California, 2005.

[29] D. Elsner, J. Fenlason, and friends. Using as. Manual, 2009.

[30] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, July 1999.

[31] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[32] J.-C. Filliâtre. Formal Verification of MIX Programs. In *Journées en l'honneur de Donald E. Knuth*, Bordeaux, France, October 2007. http://knuth07.labri.fr/exposes.php.

[33] J.-C. Filliâtre, T. Hubert, and C. Marché. The Caduceus tool for the verification of C programs. http://caduceus.lri.fr/.

[34] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In J. Davies, W. Schulte, and M. Barnett, editors, *6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, Nov. 2004. Springer.

[35] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm and Hermanns [23], pages 173–177.

[36] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.

[37] H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. In *John von Neumann, Collected Works, volume V*, pages 34–235. Pergamon Press, Oxford, 1961.

[38] H. P. Sharangpani and M. L. Barton. Statistical analysis of floating point flaw in the Pentium processor. Intel Technical Report, 1994.

[39] J. Harrison. Formal verification of floating point trigonometric functions. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233, Austin, Texas, 2000. Springer.

[40] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In R. Joshi, P. Müller, and A. Podelski, editors, *VSTTE*, Lecture Notes in Computer Science. Springer, 2012.

[41] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48:1038–1068, September 2001.

[42] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[43] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Mannual. Volume 1: Basic Architecture. Specification, 2009.

[44] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Mannual. Volume 2A: Instruction Set Reference, A-M. Specification, 2009.

[45] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Mannual. Volume 2A: Instruction Set Reference, N-Z. Specification, 2009.

[46] JML — Java Modeling Language. www.jmlspecs.org.

[47] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[48] G. T. Leavens. Not a number of floating point problems. *Journal of Object Technology*, 5(2):75–83, 2006.

[49] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 42–54, New York, NY, USA, 2006. ACM.

[50] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[51] G. Li, S. Owens, and K. Slind. Structure of a proof-producing compiler for a subset of higher order logic. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 205–219, Berlin, Heidelberg, 2007. Springer-Verlag.

[52] J.-L. Lions, L. Lübeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. Ariane 5 flight 501 failure. Report, Ariane 501 Inquiry Board, Paris, July 1996.

[53] J. Matthews, J. S. Moore, I. Ray, and D. Vroon. Verification condition generation via theorem proving. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006), Vol. 4246 of LNCS*, pages 362–376, 2006.

[54] W. D. Maurer. Proving the correctness of a flight-director program for an airborne minicomputer. In *Proceedings of the ACM SIGMINI/SIGPLAN interface meeting on Programming systems in the small processor environment*, SIGMINI '76, pages 103–108, New York, NY, USA, 1976. ACM.

[55] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2006.

[56] G. Melquiond. Proving bounds on real-valued functions with computations. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 2–17, Sydney, Australia, 2008.

[57] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. of the European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004. http://www.di.ens.fr/~mine/publi/article-mine-esop04.pdf.

[58] D. Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3):12, May 2008.

[59] D. Monniaux. *Analyse statique : de la théorie à la pratique*. Habilitation to direct research, Université Joseph Fourier, Grenoble, France, June 2009.

[60] R. Moore. *Interval Analysis.* Prentice-Hall, 1966.

[61] M. O. Myreen. *Formal verification of machine-code programs.* PhD thesis, University of Cambridge, 2008.

[62] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26:1955–1988, 2005.

[63] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[64] M. Pavlova. *Vérification de bytecode et ses application.* PhD thesis, École Supérieure en Sciences Informatiques de Sophia Antipolis, 2007.

[65] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[66] A. Stump, C. W. Barrett, D. L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 29–, Washington, DC, USA, 2001. IEEE Computer Society.

[67] United States General Accounting Office. Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia. Report, 1992.

List of Figures

2.1 Bad case for double rounding 16
2.2 A simple program giving different answers depending on the architecture. 16
2.3 A more complex program giving different answers depending on the architecture. 17
2.4 A program giving different answers depending on the optimization. 18
2.5 Why platform 19
2.6 Result of a Why program 22
2.7 Use of Why 24
2.8 A C program annotated in ACSL 25
2.9 Proof of numerical programs in Frama-C/Jessie 26

3.1 Rounding error in 64-bit rounding vs. Theorem 3.1 30
3.2 Rounding error in 80-bit rounding vs. Theorem 3.1 30
3.3 Coq theorem certifying the correctness of Theorem 3.1 32

5.1 Double rounding example with ACSL annotation. 43
5.2 Avionics program 44
5.3 Result of Figure 5.2 program 46
5.4 Summation program 47
5.5 Clock drift program 48
5.6 Scalar product program 50

6.1 Step-by-step from C program to Why proof obligations 53
6.2 A simple program 61
6.3 The program of Figure 6.2 after passing the preparation step 61
6.4 Assembly code of the example of Figure 6.3 (compiled by `gcc -S`) 62
6.5 Square program 62
6.6 Example in Figure 6.5 after preparation step 63
6.7 Assembly code of the example of Figure 6.6 (compiled by `gcc -S`) 64

7.1 Translation of a function in assembly to Why 71
7.2 Soundness of the translation 72
7.3 A simple program 77
7.4 Why program of Figure 6.4 78
7.5 Result of Figure 7.4 program 79
7.6 Square program 79
7.7 Why program of Figure 6.7 80
7.8 Result of Figure 7.7 program 81

8.1 Illustration of `vfmaddsd` instruction 88
8.2 Illustration of VFNMADDSS instruction 88

8.3 Illustration of the stack with instruction *fldl* . . . 93
8.4 A simple floating-point program . . . 101
8.5 Assembly code in x87 mode of Figure 8.4 example . . . 101
8.6 Result of Figure 8.4 program . . . 102
8.7 Assembly code in SSE2 mode of Figure 8.4 example . . . 103
8.8 Overflow example . . . 105
8.9 Non-optimized assembly code of overflow example . . . 105
8.10 Optimized assembly of overflow example . . . 106

9.1 Example with *if* . . . 112
9.2 Assembly code of program in Figure 9.1 . . . 113
9.3 CFG for assembly code in Figure 9.2 . . . 114
9.4 Program with loop statement . . . 115
9.5 CFG of Program in Figure 9.4 . . . 115
9.6 Program with loop and `goto` statement . . . 116
9.7 CFG for assembly code generated by `gcc -S` from example in Figure 9.6 . . . 116
9.8 Clock drift program . . . 121
9.9 Assembly code of program in Figure 9.8 (generated by `gcc -S -mfpmath=387 -O2`) 122
9.10 Assembly code of program in Figure 9.8 (generated by `gcc -S -mfpmath=387`) . 122
9.11 Avionics program . . . 124
9.12 Assembly code of program in Figure 9.8 (generated by `gcc -S -mfpmath=387 -O2`) 125

10.1 An example containing arrays as global values. . . . 129
10.2 Assembly code in SSE2 mode of Figure 10.1 example . . . 130
10.3 Memory model . . . 131
10.4 C code of a program with arrays defined as local variables . . . 132
10.5 Assembly code in SSE2 mode of Figure 10.4 example . . . 132
10.6 A C program with an array defined as an argument of a function . . . 133
10.7 Assembly code in SSE2 mode of Figure 10.6 example . . . 133
10.8 Maximum of an array program . . . 142
10.9 Assembly code in SSE2 mode of Figure 10.8 example . . . 143
10.10 Scalar product: annotated code . . . 145

11.1 Negation of a floating-point number . . . 147
11.2 Excerpt of assembly code in SSE mode of Figure 11.1 example . . . 148
11.3 Negation of a double by XOR . . . 148
11.4 Example about conversion of an integer to a double . . . 148
11.5 Convert an integer to a double . . . 149
11.6 A Why 3 theory example . . . 152
11.7 Bitvector theories . . . 153
11.8 BitVector theory . . . 154
11.9 BV64 theory . . . 157
11.10 Bv32_64 theory . . . 157
11.11 Bv_double theory . . . 158
11.12 Negation of a double by XOR . . . 159
11.13 Result of Figure 11.12 program . . . 160
11.14 Declaration and proofs of `Const` . . . 161
11.15 Lemmas for proving `var(x) =` $2^{52}+2^{31}$+`x` . . . 162
11.16 Proofs of `lemma1` . . . 163

Abstract

On some recently developed architectures, a numerical program may give different answers depending on the execution hardware and the compilation. These discrepancies of the results come from the fact that each floating-point computation is calculated with different precisions. The goal of this thesis is to formally prove properties about numerical programs while taking the architecture and the compiler into account. In order to do that, we propose two different approaches. The first approach is to prove properties of floating-point programs that are true for multiple architectures and compilers. This approach states the rounding error of each floating-point computation whatever the environment and the compiler choices. It is implemented in the Frama-C platform for static analysis of C code. The second approach is to prove behavioral properties of numerical programs by analyzing their compiled assembly code. We focus on the issues and traps that may arise on floating-point computations. Direct analysis of the assembly code allows us to take into account architecture- or compiler-dependent features such as the possible use of extended precision registers. It is implemented above the Why platform for deductive verification.

Keywords: Floating-point arithmetic, Numerical programs, Static analysis, Compile-time optimizations, the Why platform, the Frama-C platform.

Résumé

Sur des architectures récentes, un programme numérique peut donner des réponses différentes en fonction du hardware et du compilateur. Ces incohérences des résultats viennent du fait que chaque calcul en virgule flottante est effectué avec des précisions différentes. Le but de cette thèse est de prouver formellement des propriétés des programmes opérant sur des nombres flottants en prenant en compte l'architecture et le compilateur. Pour le faire, nous avons proposé deux approches différentes. La première approche est de prouver des propriétés des programmes en virgule flottante qui sont vraies sur plusieurs architectures et compilateurs. Cette approche ne considère que les erreurs d'arrondi qui doivent être validées quels que soient l'environnement matériel et le choix du compilateur. Elle est implantée dans la plate-forme Frama-C pour l'analyse statique de code C. La deuxième approche consiste à prouver des propriétés des programmes en analysant leur code assembleur. Nous nous concentrons sur des problèmes et des pièges qui apparaissent sur des calculs en virgule flottante. L'analyse directe du code assembleur nous permet de considérer des caratéristiques dépendant de l'architecture ou du compilateur telle que l'utilisation des registres en précision étendue. Cette approche est implantée comme une sur-couche de la plate-forme Why pour la vérification déductive.

Mots clés: Arithmétique en virgule flottante, Programmes numériques, Analyse statique, Optimisations à la compilation, Plate-forme Why, Plate-forme Frama-C.