Proving Floating-Point Numerical Programs by Analysis of their Assembly Code

Nguyen Thi Minh Tuyen, Claude Marché

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Proving Floating-Point Numerical Programs by Analysis of their Assembly Code

Thi Minh Tuyen Nguyen — Claude Marché

N° 7655

June 2011

ISSN 0249-6399 ISRN INRIA/RR--7655--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY – ÎLE-DE-FRANCE

Proving Floating-Point Numerical Programs by Analysis of their Assembly Code

Thi Minh Tuyen Nguyen[*] [†] , Claude Marché[*†]

Thème : Programmation, vérification et preuves
Équipes-Projets PROVAL

Rapport de recherche n° 7655 — June 2011 — 58 pages

Abstract: We present an approach for proving 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 to accurately take into account architecture- or compiler-dependent features such as the possible use of extended precision registers.

The approach is implemented on top of the generic Why platform for deductive verification, which allows us to perform experiments where proofs are discharged by combining several back-end automatic provers.

Key-words: Formal Specification, Proof, assembly code, floating-point computations

[*] INRIA Saclay - Île-de-France, F-91893
[†] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Preuve de programmes avec calculs en virgule flottante par analyse de leur code assembleur

Résumé : Nous décrivons une nouvelle approche pour prouver des propriétés du comportement des programmes numériques en analysant leur code assembleur compilé. Nous mettons l'accent sur les enjeux et les pièges qui peuvent survenir lors des calculs en virgule flottante. L'analyse directe du code assembleur permet de prendre en compte de façon précise l'architecture et le compilateur, par exemple l'utilisation de registres en précision flottante étendue.

Un prototype est implanté au-dessus de la plate-forme générique Why plate-forme pour la vérification déductive. Nous présentons des expérimentations où les preuves sont effectuées par une combinaison de plusieurs prouveurs automatiques.

Mots-clés : Spécification formelle, Preuve, assembleur, calculs en virgule flottante

Contents

List of Figures

Chapter 1

Introduction

The C language is the first choice for embedded systems or critical software from domains such as simulation of physical systems, control-command programs in transportation, etc. For such systems, floating-point (FP for short) computations are involved and precision of calculations is an important issue. The IEEE-754 standard [1] enforces a precise definition on how the basic arithmetic operations (+, -, *, /, and also absolute value, square root, etc.) must be computed on given FP format (32 bits, 64 bits, etc.) and w.r.t a given rounding mode. This standard is currently supported by most of the processor chips. However, this does not imply that a given C program must produce exactly the same results whatever is the compiler and the underlying architecture. There are several possible reasons, e.g. the x87 floating-point unit (FPU) uses 80-bit internal floating-point registers, or the compiler may optimize the assembly code be changing the order of operations. Such issues have been extensively analyzed by D. Monniaux [23]. A small example that illustrates such an issue is as follows.

```
double doublerounding() {
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64;
  double z = x + y;
  return z;
}
```

If computations follows the IEEE-754 standard, the result should be $1.0+2^{-52}$, but if compiled using the x87 FPU, a *double rounding* happens and the result is 1.0. The latter compilation does not *strictly* follows the standard[1].

In the context of static verification, FP computations have been considered in part. In analysis based on the abstract interpretation framework, support for FP computations is proposed in tools like Fluctuat [15] and Astrée [14]. Generally speaking, FP arithmetic has been formalized since 1989 to formally prove hardware components or algorithms [11, 20, 25].

However, there are very few attempts to analyze FP programs in the so-called *extended static checking* techniques, or *deductive verification* techniques, where verification is typically performed by producing proof obligations, which are formulas to be shown valid using theorem provers. In this context, complex behavioral properties are formally specified, using specification languages such as JML [9] for Java, ACSL [5] for C, Spec#[4] for C#. The support for floating-point computations in such approaches is poorly studied. In 2006, Leavens [22] enumerates a set of possible traps when one attempts to specify FP programs. In 2007, Boldo and Filliâtre [6] propose both a specification language to specify FP programs and an approach to generate proof obligations to be proved in the Coq proof assistant. In 2010, Ayad and Marché [2] extended this to the support of special values and to the use of automated theorem provers. However, the former approaches assume that the compiler strictly follows the IEEE-754 standard. In other words, on the example above they can prove that the result is $1 + 2^{-52}$.

[1] The term *strict* here refers to the `-fp-model strict` or `/fp:strict` options on C compilers, or the `strictfp` keyword of Java, which explicitly require the compilation to strictly conform to the standard.

In 2010, Boldo and Nguyen [7, 8] proposed a deductive verification approach which is compiler and architecture *independent*, is the sense that the behavioral properties that can be proved valid on a FP program are true whatever does the compiler (up to some extent). On the same example, the only property that can be proved is that the result is between 1 and $1 + 2^{-52}$. In this paper, we propose an approach which is compiler and architecture *dependent*: the requirements are proved valid with respect to the assembly code generated by the compiler. At the level of the assembly, all architecture-dependent information is known, such as the precision of each operation.

In Chapter 2, we present the necessary background needed on assembly code on one hand, and on the Why platform on the other hand. Chapter 3 first considers a reduced class of "simple" C programs to present the main principles of our approach. Chapter 4 then specifically focuses on floating-point computation and how the specificities of the compiler and architecture are taken into account. Chapter 5 presents how we deal with programs containing conditional statements and loops, that is when the assembly code contains jump statements. Chapter 6 considers the case of programs involving arrays, for which we need to change our so-called memory model. Chapter 7 concludes with comparisons to related works and perspectives.

Chapter 2

Basic background

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.

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 uses GCC to generate assembly code, only AT&T syntax will be used in this document.

2.1 Assembly language

2.1.1 Operands

An operand in assembly language may be a register, a memory reference or a constant.

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.

For example, in AT&T syntax:

- `-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 this 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. Only constant offsets are valid. For example, in AT&T syntax:

- `1234(%rip)` points to the address 1234 bytes past the end of the current instruction.
- `symbol(%rip)` points to the symbol in RIP relative way [17].

Immediate operands are preceded by '$'. For example: $1, $12

2.1.2 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.

2.1.3 General-purpose instructions

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

[label] mnemonic [operands] [comment]

The fields in the square brackets are optional in some statements.

In this section, we only talk about the general-purpose instructions, floating-point instructions does not include.

Also note that the order of source operands and destination operand is reversed in AT&T syntax. This means that source operand is on the left-hand side.

Data transfer instructions

The *mov* instruction uses to transfer data from source operand to destination operand. It requires two operands and has the syntax:

mov source, destination

The data is copied from `source` to `destination` and the `source` 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

INC and DEC Instructions These instructions can be used to either increment or decrement the operands by one. The `inc` (INCrement) instruction adds one to its operand and the `dec` (DECrement) instruction subtracts one from its operand. Both instructions require a single operand. The operand can be either in a register or in memory. It does not make sense to use an immediate operand such as `inc 55` or `dec 109`. The general format of these instructions is

inc destination
dec destination

where `destination` may be an 8-, 16- or 32-bit operand.

ADD/SUB/MUL/DIV Instructions

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

add source, destination	destination = destination + source
sub source, destination	destination = destination - source
mul source, destination	destination = destination * source
div source, destination	destination = destination / source

CMP Instructions

The `cmp` (CoMPare) instruction is used to compare two operands (equal, not equal, and so on). The `cmp` instruction performs the same operation as the sub except that the result of subtraction is not saved. Thus, `cmp` does not disturb the source and destination operands. The `cmp` instruction is typically used in conjunction with a conditional jump instruction for decision making [13, 24].

2.1.4 Assembler directives

Here are some assembler directives we will see in our examples.

.comm *symbol, length*

.comm 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 *ld* does not see a definition for the symbol – just one or more common symbols – then it will allocate length bytes of uninitialized memory. 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.

.globl *symbol*, .global *symbol*

.global makes the symbol visible to *ld*. If you define symbol in your 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_startproc

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

.cfi_endproc

.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* [17].

2.2 Inline assembly

2.2.1 Simple Inline Statement

The form of a basic inline statement is:

`asm("`*assembly code*`");`

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

2.2.2 Extended Inline Statements

In basic inline assembly, we had 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

Inputs are specified in the same why, 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 .... For example:

```
asm(   "movl %0, %1"
       :"=r"(sum)/* output */
       :"r"(number1)/* input */
       );
```

The C variables *sum* and *number1* are both mapped to registers. In assembly code statement, *sum* is identified by %0 and *number1* 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 (...:  ...:  ...:  ...)
```

2.3 Why: a software verification platform

Why is a generic platform for deductive verification [19]. It is generic on the front-end side since it is an intermediate language for higher-level like C or Java. It is generic on the side of output since it can produce proof obligations for different provers.

In the input language of Why, one 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. One can then declare a set of *references* which are mutable variables denoting logic values. Finally, one can define procedures which can modify these references. The body of such a procedure is made of statements in a while-style language. Procedures are also 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. We use this feature extensively in the remaining.

2.4 Frama-C and the ACSL specification language

Frama-C is a framework for static analysis of C source code. Its architecture is modular: a kernel is provided which performs parsing and typing of source code, to which analyzers can plug-in. The parser is able to parse formal annotations given as a special kind of comments. The formal language of these annotations is ACSL [5]. Examples of such annotations will come along this paper.

Chapter 3

Case of Simple programs

In this chapter we describe the main principles of our approach, on a reduced class of C programs called "simple", described below.

3.1 Overview of the approach

Our approach for proving a C source via analyzing its assembly is made in several steps illustrated on Figure 3.1. The figure in the left hand-side is all steps to prove a program with assembly code. The one in the right hand-side instantiates these steps concretely for the proof of some program `foo.c`.

In C program, all annotations are put in comments. When `gcc` generates assembly code, these annotations will be ignored. As we need them to prove the program, a preprocessing step is needed. This step puts all annotations into inline assembly in order to keep them in assembly code. This is detailed in Section 3.4.

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

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

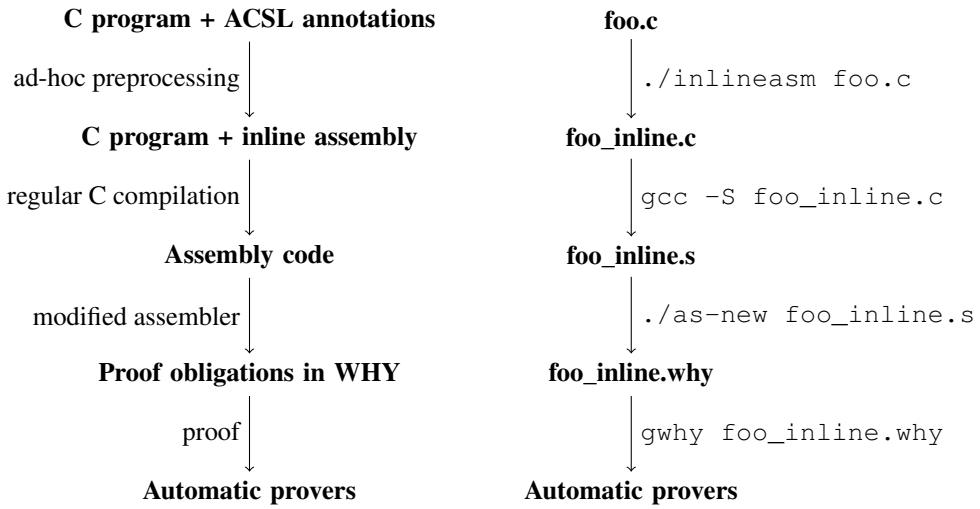


Figure 3.1: Step-by-step from C program to WHY proof obligations

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

  return tmp;
}
```

Figure 3.2: A simple program

3.2 Definition of the class of "simple" C programs

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

- The only data type is the type `int` which is assumed to denote 32-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, no `if` and no `switch` statements and no `goto`.
- The allowed expressions are the arithmetic expressions plus the functions 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.

Example 3.1 (A simple example) *This small example (Figure 3.2) has a function int f(int n) that returns the value of* $100 - n$*. It is found in the documentation of ACSL [5]. 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* $tmp \geq 100$*.*

The assembly code of this program (See Figure 3.3) is generated by the default option of `gcc`*. There are only three basic instructions to use: transfer data with mov instruction, subtract instruction and instructions for returning from a function.*

The function **f** *in assembly code is defined as a global symbol with its type is* **@function***(line 3–4). This means that this function is visible in other files. A label* **f** *begins this function. The body of this function is between two directives* **.cfi_startproc** *and* **.cfi_endproc***.*

As one can see, ACSL annotations appear between **#APP** *and* **#NO_APP** *in assembly code. All the lines between* **#APP** *and* **#NO_APP** *are indeed ignored by the GNU assembler. We use this feature for putting annotations, this is described in Section 3.4.*

3.3 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 instructions to Why. Finally, it is the translation of annotations.

3.3.1 Translation of 32-bit integers

Why only have unbounded mathematical integers built-in. Thus, 32-bits 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.

```
1           .file   "simple.c"
2           .text
3   .globl f
4           .type   f, @function
5   f:
6   .LFB0:
7           .cfi_startproc
8   ....
9           movl    %edi, -20(%rbp)
10  #APP
11  # 2 "simple.c" 1
12          /*requires #int#-20(%rbp)# >= 0 && #int#-20(%rbp)# < 100;*/
13  # 0 "" 2
14  #NO_APP
15          movl    -20(%rbp), %eax
16          movl    $100, %edx
17          movl    %edx, %ecx
18          subl    %eax, %ecx
19          movl    %ecx, %eax
20          movl    %eax, -4(%rbp)
21  #APP
22  # 4 "simple.c" 1
23          /* assert #int#-4(%rbp)# > 0;*/
24  # 0 "" 2
25  # 5 "simple.c" 1
26          /* assert #int#-4(%rbp)# <= 100;*/
27  # 0 "" 2
28  #NO_APP
29          movl    -4(%rbp), %eax
30          leave
31          .cfi_def_cfa 7, 8
32          ret
33          .cfi_endproc
34  ....
```

Figure 3.3: Assembly code of the example of Figure 3.2

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

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
```

We have an axiom

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

Although the 8- and 16- integers are considered here for simplicity, they could be handled similarly, as it is in Jessie.

3.3.2 Translation of operands

We want to translate operands of kind register or memory reference into Why variables. To do so, we make the following hypothesis:

Assumption 3.2 (Separate 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 not 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 6, 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.

For the simplicity, we ignore the size of the register. This means that `%ax`, `%eax` and `%rax` has the same name `_rax`. Once we have this, we do not need to cast from `%ax` to `%eax` `%rax` and otherwise.

The following abstract type and logic function will be used in this section:

```
type register
```

Each register or memory reference used as an operand will be declared as a variable with type *register*.

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

The logic function *sel_int32* returns an 32-bit integer from a *register*.

We distinguish two types of operands as follows:

- Immediate operand: begins by '$'. There is not any declaration here because this operand is a constant. What we do is to delete the prefix '$'.
- 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: $\overline{\textit{-4(\%rbp)}} = \textit{\_4\_\_rbp\_}$. The variable in Why to declare for this operand is *parameter _4__rbp_: register ref*

We denote

- $[\![imm]\!]_{int32} =$ `imm`
- $[\![reg]\!]_{int32} =$ `(integer_of_int32 (sel_int32 !reg))`
- $[\![mem]\!]_{int32} =$ `(integer_of_int32 (sel_int32 !mem))`

3.3.3 Translation of instructions

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

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

This abstract function will set an 32-bit integer to a register without verifying if this value is overflow 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`.

We denote by

- $[\![$ ins $]\!]_i$ the Why translation of an instruction *ins*,
- *op* the Why interpretation of *op* as a left-value
- $[\![op]\!]_r$ the Why interpretation of *op* as a right value

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*
$[\![$ mull 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()`

Note that `leave` and `ret` are instructions in assembly language but they do not have any translation here.

As specified in the documentation, *src* of the instruction *movl* is either a constant, a register or a memory reference in 32 bits. Therefore, we do not need to verify if it is overflow 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 as other operations since the divisor must be check non-null. We thus use

$[\![$ divl src, dest $]\!]_i$ = `div_int32` $[\![src]\!]_{int32}$ *dest*
$[\![$ idivl src, dest $]\!]_i$ = `div_int32` $[\![src]\!]_{int32}$ *dest*

with the special Why parameters:

```
parameter div_int32: imm: int32 -> 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)
}
```

The function `computer_div` is defined in 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.

3.3.4 Translation of a sequence of instructions

A function in assembly code is a sequence of instructions. All we have until now is the interpretation of each separate instruction. What we need is how to assure that the state at a instruction corresponds to a state in Why. This will be presented in Theorem 3.3.

Theorem 3.3 *Let S be one state in assembly program, $\overline{S}$ be the state in Why corresponding to S, $R(S, \overline{S})$ be the relation between S and $\overline{S}$.*
$\forall S, \overline{S}, i : R(S, \overline{S}) \bigwedge$ *(S,i* $\Rightarrow$ *S')* $\bigwedge$ *(* $\overline{S}$*,* $[\![i]\!] \Rightarrow \overline{S'}$ *)* $\longrightarrow$ *R(S', $\overline{S'}$)*

3.4 Annotations

3.4.1 Types of annotations interpreted

The annotations mentioned in this section are:

- preconditions
- post-conditions
- assertions

Normally, when assembly code is generated, all the comments in C after // or between /* and */ will be eliminated. The annotations are thus disappeared in assembly code. In order to keep these annotations in assembly code, we need a pre-processing step which will be presented in 3.4.2.

Note that in this document, both ACSL annotations and Why annotations are authorize. This is the reason why we don't use Frama-C plugin.

3.4.2 Preprocessing: keeps annotations in assembly code

The goal of this step is to create a new C file which contains inline assembly in order to have annotations in the assembly file generated. The idea is that we put all annotations in inline assembly statements.

This step follows several steps:

- Firstly we detect all the variables in the program and their types. We use an array to store these variables.
  - With assertion: if the assertion is in n^{th} line, we only need to find global variables, function parameters and local variables (in the function containing this assertion) declared before line n.
  - With precondition and post-condition: function parameters may be needed in this kind of annotation. Therefore, we have to get all function parameters and their type. Then put the annotations after the function declaration (more precisely, after '{').
- Replace variables in annotation: Each time we meet a variable in annotation, we will replace it by "#type#argument#" where *type* is the type of variable, *argument* has format '%' + *order number*.The syntax of inline assembly was mentioned in Chapter 2.
- Put the annotation replaced in inline assembly statement.

By using this syntax of inline assembly, we are able to have directly the memory reference/register corresponding to the variable when compiling. For example, an annotation in ACSL :

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

is put in inline assembly as the following format:

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

Figure 3.4: Translation of a function in assembly to Why

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

"%0", "%1" are replaced by the memory reference of the variable n.

This inline assembly will then be translated in assembly code as follows:

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

It is not easy to know exactly type of a memory reference/register in assembly code. Here, thank to inline assembly, we have both memory reference/register and type of variable in C.

This preprocessing step will deplace precondition and put it after the declaration of the function, other words, put it after ''. The post-condition will move to the line before `return value`. Once this is done, we have memory reference/register of the variables in both pre- and post-condition and even the one of the abstract variable `\result`.

In a program, we have many `return` and it will be a bad idea if we put each post-condition before each `return`. We add a new variable `res` with the type is the type of the returned value. We analyze the code source C and replace each `return v;` by `{res = v; goto Lres;}`. At the end of the program, before '', we put `Lres: return res;`. If there is a post-condition, put it between the `Lres:` and `return res;`

In Why, each function will have a Why interface with the input and output are registers (as they are input and output of a function in assembly language). However, maybe the variables of the annotation are memory reference. This is the reason why we need to match memory references with input/output registers . To do this, we create an array and find only in the **mov** instructions all the operands and match the input register and the memory reference. When creating the function interface, we simply replace the memory reference in precondition and post-condition by the corresponding register.

3.4.3 Translation of annotations in assembly to Why

Now we have annotations in assembly file. The question is how to translate these annotation to Why? We already know that all inline assembly are put between *#APP* and *#NO_APP*, it is thus easy to identify where annotations are in assembly code.

We denote by $[\![A]\!]_{annot}$ the translation of an annotation to Why and

$[\![v]\!]_{int32@} =$ `integer_of_int32(sel_int32(v))`

Assume that we have a function with preconditions, post-conditions and assertions. The translation of this function in assembly to Why is illustrated in Figure 3.4. As we see in this figure, the post-condition becomes an assertion in Why.

For each function with pre- and post-condition, we define an interface of function in Why (See Figure 3.4) where w is is a set of variables modified in the function.

The translation of annotations to Why is described in the table below:

$$
\begin{array}{lcl}
[\![A \texttt{ ==> } B]\!]_{annot} & = & [\![A]\!]_{annot} \texttt{ -> } [\![B]\!]_{annot} \\
[\![A \texttt{ == } B]\!]_{annot} & = & [\![A]\!]_{annot} \texttt{ = } [\![B]\!]_{annot} \\
[\![A \texttt{ \&\& } B]\!]_{annot} & = & [\![A]\!]_{annot} \texttt{ and } [\![B]\!]_{annot} \\
[\![A \texttt{ || } B]\!]_{annot} & = & [\![A]\!]_{annot} \texttt{ or } [\![B]\!]_{annot} \\
[\![\texttt{!}A]\!]_{annot} & = & \texttt{not(} [\![A]\!]_{annot} \texttt{)} \\
[\![\texttt{\#int\#v\#}]\!]_{annot} & = & [\![v]\!]_{int32@} \\
[\![\texttt{\textbackslash abs(\#int\#v\#)}]\!]_{annot} & = & \texttt{abs\_int(}[\![\texttt{\#int\#v\#}]\!]_{annot}\texttt{)} \\
[\![e_1 \texttt{ } op \texttt{ } e_2]\!]_{annot} & = & [\![e_1]\!]_{annot} \texttt{ } op \texttt{ } [\![e_2]\!]_{annot} \text{ where } op \in \{+,-,*,/\} \\
[\![e_1 \texttt{ } op \texttt{ } e_2]\!]_{annot} & = & [\![e_1]\!]_{annot} \texttt{ } op \texttt{ } [\![e_2]\!]_{annot} \text{ where } op \in \{>,<,>=,<=,!=\} \\
[\![\texttt{\textbackslash forall } \tau \texttt{ } i\texttt{; } P]\!]_{annot} & = & \texttt{forall } i\texttt{:}[\![\tau]\!]_{annot}\texttt{. } [\![P]\!]_{annot} \\
[\![\texttt{\textbackslash exists } \tau \texttt{ } i\texttt{; } P]\!]_{annot} & = & \texttt{exists } i\texttt{:}[\![\tau]\!]_{annot}\texttt{. } [\![P]\!]_{annot} \\
[\![\texttt{integer}]\!]_{annot} & = & \texttt{int}
\end{array}
$$

3.5 Examples

Figure 3.5 is the Why program of the assembly code in Example 3.1. The result of this Why program is in Figure 3.6.

```
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:
(*#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 3.5: Why program of Figure 3.3

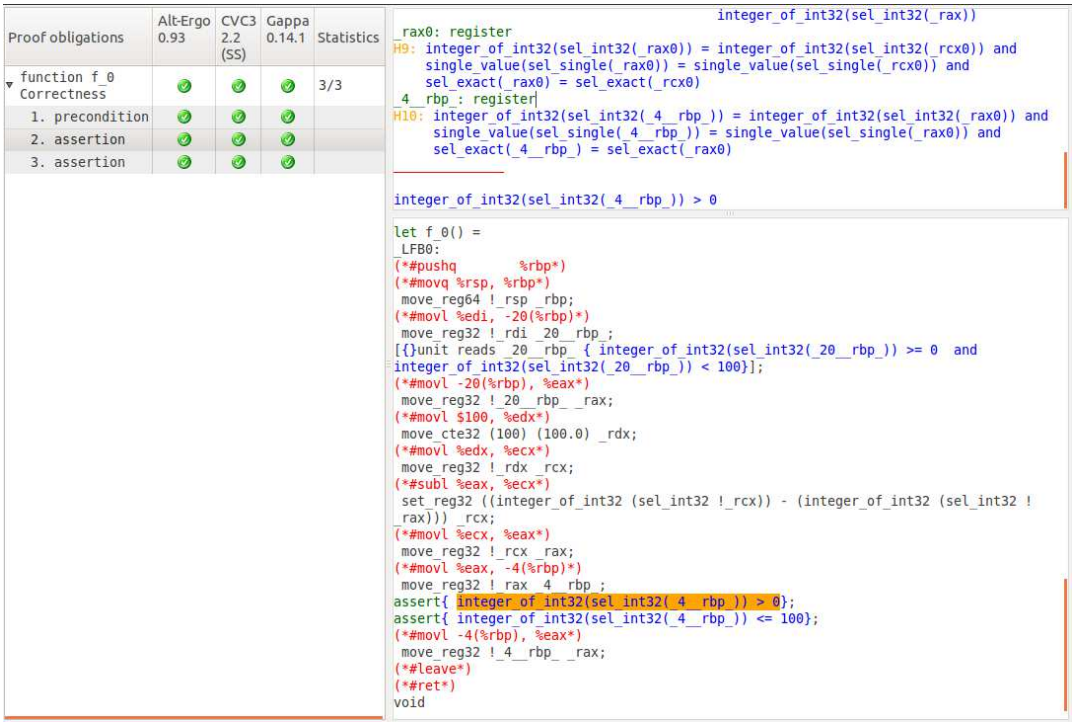


Figure 3.6: Result of Figure 3.5 program

Chapter 4

Floating-point programs

Chapter 3 talks about the translation of the program containing only *32-bit integer* type. In this chapter, we will extend it with the computation in both integer and floating-point. There are some points we need to rewrite in order to make it true in both integer and floating-point computation.

4.1 Definition of programs

This chapter is an extension of the previous one. The programs concerned are the one support for both `int`, `long`, `float` and `double`. Certainly, the annotations can contain variables with those type. An interesting point of this chapter is that we will show the different results obtained by compiling a floating-point program with different mode of compiler and architecture.

4.2 Assembly with floating-point arithmetic

Before entering to the translation, we give some basic knowledge about the different modes: SSE/SSE2, x87 and FMA and their instructions.

4.2.1 SSE/SSE2

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.

SSE2 is a major enhancement to SSE. It adds new maths 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/m632`
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

`comisd` and `comiss` 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 OF, SF and AF flags in the EFLAGS register are set to 0. In our model, we consider that there is no difference between `ucomiss`/`ucomisd` and `comiss`/`comisd`.

Conversion Instructions

4.2.2 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.

FPU registers

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

The 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. In addition, we can access these registers individually using `st0`, `st1`, and so on.

Since these registers are organized as a register stack, these names are not statically assigned. That is, `st0` 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 `st1`, and so on; the last register as `st7`. There is a 3-bit top-of-stack pointer in the status register to identify the TOS register.

Each data register can hold an extended-precision floating-point number. This format uses 80 bits as opposed to single-precision (32 bits) or double-precision (64 bits) formats. The rationale is that these registers typically hold intermediate results and using the extended format improves the accuracy of the final result.

	sign	exponent	mantissa
ST7			
ST6			
ST5			
ST4			
ST3			
ST2			
ST1			
ST0			

Figure 4.1: FPU data registers

x87 FPU instructions

Most floating-point instructions requires 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 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	Load +0.0
fld1	Load +1.0
fldpi	Load π
fldl2t	Load $log_2 10$
fldl2e	Load $log_2 e$
fldlg2	Load $log_{10} 2$
fldln2	Load $log_e 2$

Basic Arithmetic Instructions These are the floating-point instructions perform basic arithmetic operations on floating-point numbers:

fiadd	Add integer to floating point
fsub/fsubp	Subtract floating point
fisub	Subtract integer from floating point
fsubr/fsubrp	Reverse subtract floating point
fisubr	Reverse subtract floating point from integer
fmul/fmulp	Multiply floating point
fimul	Multiply integer by floating point
fdiv/fdivp	Divide floating point
fidiv	Divide floating point by integer
fdivr/fdivrp	Reverse divide
fidivr	Reverse divide integer by floating point
fabs	Absolute value
fchs	Change sign

The add, subtract, multiply and divide 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

Reverse versions of the subtract (`fsubr`) and divide (`fdivr`) instructions enable efficient coding. For example, the following options are available with the `fsub` and `fsubr` instructions for operating on values in a specified x87 FPU data register `st(i)` and the `st(0)` register:

`fsub`:
st(0) ← st(0) - st(i)
st(i) ← st(i) - st(0)

`fsubr`:
st(0) ← st(i) - st(0)
st(i) ← st(0) - st(i)

The pop versions of the add, subtract, multiply, and divide instructions offer the option of popping the x87 FPU register stack following the arithmetic operation. These instructions operate on values in the `st(i)` and `st(0)` registers, store the result in the `st(i)` register, and pop the `st(0)` register.

4.2.3 FMA

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

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

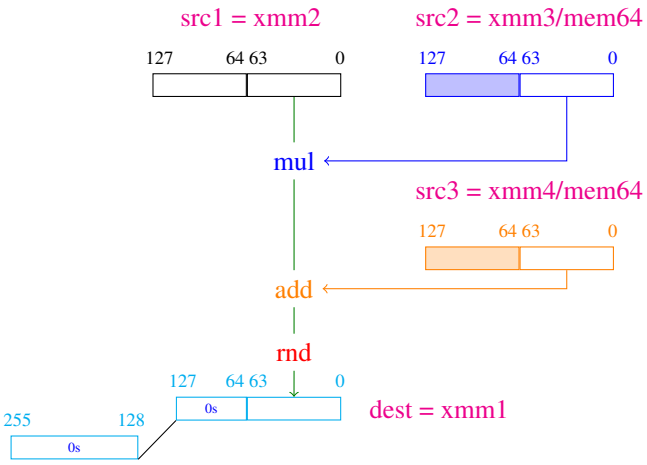


Figure 4.2: Illustration of `vfmaddsd` instruction

FMA instructions

Now both AMD and Intel have specifications for FMA. In this document, FMA instructions are generated (thanks to `gcc`) by FMA4, specified by AMD.

vfmaddss src3, src2, src1, dest	dest = src1* src2 + src3	Multiply and Add Scalar Single-Precision Floating-Point
vfmaddsd src3, src2, src1, dest	dest = src1* src2 + src3	Multiply and Add Scalar Double-Precision Floating-Point
vfmsubss src3, src2, src1, dest	dest = src1* src2 - src3	Multiply and Subtract Scalar Single-Precision Floating-Point
vfmsubsd src3, src2, src1, dest	dest = src1* src2 - src3	Multiply and Subtract Scalar Double-Precision Floating-Point
vfnmaddss src3, src2, src1, dest	dest = -(src1* src2) + src3	Negative Multiply and Add Scalar Single-Precision Floating-Point
vfnmaddsd src3, src2, src1, dest	dest = -(src1* src2) + src3	Negative Multiply and Add Scalar Double-Precision Floating-Point
vfnmsubss src2, src2, src1, dest	dest = -(src1* src2) - src3	Negative Multiply and Subtract Scalar Single-Precision Floating-Point
vfnmsubsd src3, src2, src1, dest	dest = -(src1* src2) - src3	Negative Multiply and Subtract Scalar Double-Precision Floating-Point

The implement of `vfmaddsd` is presented in Figure 4.2. The destination is an `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 4.3. The destination is always an `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.

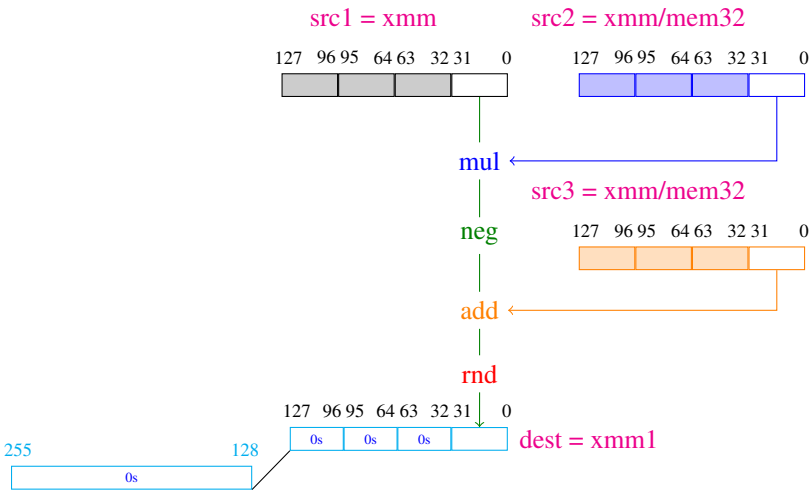


Figure 4.3: Illustration of VFNMADDSS instruction

```
double doublerounding(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64;
  double z = x + y;

  //@ assert z == x;
  return z;
}
```

Figure 4.4: A simple floating-point program

4.3 Examples of the chapter

Two following examples are used for illustrating the translation of floating-point instruction to Why. Their proofs will be presented at the end of this chapter.

4.3.1 Double rounding example

This example do the calculation of two *double* values. The goal of this example is to show that the result depends on the architecture and the choice of compiler options. More precisely, with the strict IEEE-754 standard (default option of *gcc*), the returned value of the function *doublerounding()* is $1 + 2^{-52}$. When it is compiled on IA32 architecture – the computation on *double* are performed in the long double type inside x87 unit – the result is 1. We insert an assertion `//@ assert z == x` which will be true in SSE case and won't be proved in x87 case.

We show here a piece of assembly code which shows the difference in these two modes (See Figure 4.5).

On the left hand-side of Figure 4.5, instructions on 64 bits (`movabs`, `movq`, `movsd`, `addsd`) are used. Thus, all the calculations are in 64 bits (SSE mode). On the right hand-side, the intermediate value are stored in x87 register (80-bit registers) and then it is rounded to 64-bit value (line 5–7). This is called *double rounding* and its value is different from the direct one (SSE mode).

SSE

```
1 movabsq $4607182418800017408, %rax
2 movq    %rax, -16(%rbp)
3 movabsq $4368493837572636672, %rax
4 movq    %rax, -8(%rbp)
5 movsd   -16(%rbp), %xmm0
6 addsd   -8(%rbp), %xmm0
7 movsd   %xmm0, -24(%rbp)
```

x87

```
1 movabsq $4607182418800017408, %rax
2 movq    %rax, -16(%rbp)
3 movabsq $4368493837572636672, %rax
4 movq    %rax, -8(%rbp)
5 fldl    -16(%rbp)
6 faddl   -8(%rbp)
7 fstpl   -24(%rbp)
```

Figure 4.5: Assembly code in SSE mode and x87 mode of Figure 4.4 example

4.3.2 Example: Architecture dependent Overflow

Monniaux [23] considers the following program to illustrate differences between architectures with respect to overflows.

```
double foo() {
  double v = 1e308;
  double y = v * v;
  return y/v;
}
```

Excerpts of the generated assembly code are shown on Fig. 4.6. The left part corresponds to non-optimized x87 code (precisely, `gcc -mfpmath=387 -O0`) whereas the right part is optimized (`-O1`).

No optimization

```
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 ....
```

Optimized, level 1

```
1  fldl   .LC0(%rip)
2  fld    %st(0)
3  fmul   %st(1), %st
4  fdivp  %st, %st(1)
5  fstpl  -8(%rsp)
6  movsd  -8(%rsp), %xmm0
7  ....
8  .LC0:
9    .long 2246822048
10   .long 2145504499
```

Figure 4.6: Optimized versus non-optimized assembly of overflow example

4.4 Translation to Why

4.4.1 Abstract functions

An integer in 64 bits has type `int64`. Like `int32`, we define two following logic functions for it:

```
type int64
logic sel_int64: register -> 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))
```

In order to get a floating-point value from a register, we need some following logic functions:

```
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` value and exact value, respectively, from a register.

We also need to set a floating-point value to a `register`. To do that, we need 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 don't 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 FP 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` and 80-bit FP 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}

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}
```

Division is a special case. We need to assure that the denominator is not equal to 0. Different from `div`, with `divr`, the numerator and the denominator are reversed.

```
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)}

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
 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@)}
```

where `round_single(mode, value)`, `round_double(mode, value)`, `round_binary80(mode, value)` have already defined in Why library `floats_common.why`.

In assembly language, there are instructions which transfer data from `src` to `dest` and we know only their size but we do not know the type of the data. For example, `movl` is a data transfer instruction but we do not know if it transfer a 32-bit integer or a 32-bit FP number(`float` type in C).

Our idea is that we define a parameter `move_cte32` which copies at the same time an integer value , a single value and its exact value to `dest` and a parameter `move_cte64` which copies an integer value, a double value and its exact value to `dest`.

```
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
}
```

We denote by $[\![opr]\!]_{int32}$,$[\![opr]\!]_{single}$, $[\![opr]\!]_{double}$ and $[\![opr]\!]_{binary80}$ the interpretation of an operand that return an integer value, a real in 32 bits, 64 bits and 80 bits respectively from a *register*.

We denote by *opr* and operand being register or memory reference. The translation of operands is specified as follows:

$[\![opr]\!]_{int32}$ = `(integer_of_int32 (sel_int32 !opr))`
$[\![opr]\!]_{int64}$ = `(integer_of_int64 (sel_int64 !opr))`
$[\![opr]\!]_{single}$ = `(single_value (sel_single !opr))`
$[\![opr]\!]_{double}$ = `(double_value (sel_double !opr))`
$[\![opr]\!]_{binary80}$ = `(binary80_value (sel_80 !opr))`
$[\![opr]\!]_{exact}$ = `((sel_exact !opr)`

$[\![symbol]\!]_{int32}$ = `(integer_of_int32 (sel_int32 symbol))`
$[\![symbol]\!]_{int64}$ = `(integer_of_int64 (sel_int64 symbol))`
$[\![symbol]\!]_{single}$ = `(single_value (sel_single symbol))`
$[\![symbol]\!]_{double}$ = `(double_value (sel_double symbol))`
$[\![symbol]\!]_{binary80}$ = `(binary80_value (sel_80 symbol))`
$[\![symbol]\!]_{exact}$ = `((sel_exact symbol)`

4.4.2 When constants is referenced by `%rip`

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

```
    .align 8
.LC1:
    .long 0
    .long 1025507328
```

Thank to the directive `.align` we know that the constant at `.LC1` has 8 bytes, corresponding to a `double` or 64-bit integer. Each `.long` has 4 bytes. Therefore, the constant at `.LC1` has two parts,

each part has 4 bytes. The 32-bit lower part is 0 and the 32-bit higher part is represented in integer (1025507328).

When a value is stored in memory, we don't know its type. Because of this, in our axiom, we consider this value has both form: 64-bit integer and `double`. From this information, we translate it to Why as follows:

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

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}. One important point is that in memory, these value is stored at the same position and is represented by 0s and 1s. The two values we have are just two different representations of the same number.

By using these logic and parameter functions above, the interpretation of floating-point instructions is presented as follows:

4.4.3 Rewrite 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. This is the reason why we have three value at the same time. It is true that the information is redundant but it make sure that we have enough information to prove.

$[\![$ movl src, dest $]\!]_i$ = `move_cte32` $[\![src]\!]_{int32}$ $[\![src]\!]_{single}$ $[\![src]\!]_{exact}$ `dest`
$[\![$ movq src, dest $]\!]_i$ = `move_cte64` $[\![src]\!]_{int64}$ $[\![src]\!]_{double}$ $[\![src]\!]_{exact}$ `dest`

4.4.4 Translation of SSE/SSE2 instructions

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

4.4.5 x87 Floating-point Unit

Representation of stack in Why

As we mentioned in 4.2.2, the stack has eight floating-point registers (`st0` – `st8`) 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 an array of type `register`. We have tried it but the problem is that Gappa does not understand the array defined in Why. Thus, we cannot prove the program with Gappa.

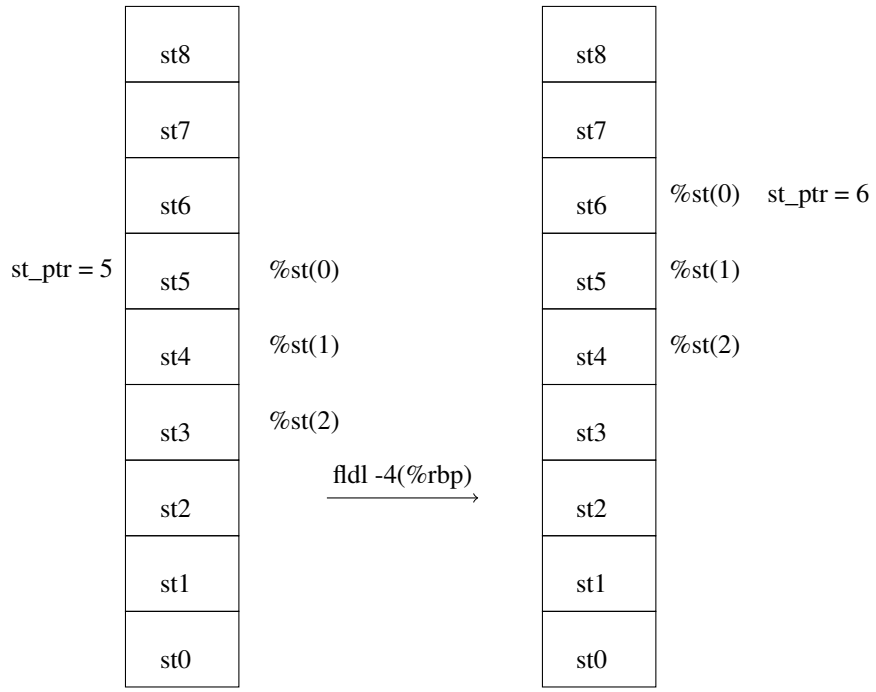


Figure 4.7: Illustration of the stack with instruction *fldl*

We propose another solution which help Gappa to prove the program in x87: instead of using an array, we define eight variable of type `registers`: `st0`, ..., `st8` to represent for `st0`–`st8`. A variable `st_ptr` stores the position of the current register. The initial value of `st_ptr` is -1. Before each load instruction is executed, `st_ptr` increases by 1. Otherwise, if the stack pops a value (for example `fstp`, `faddp`, ect.), `st_ptr` decreases by 1 after this instruction is executed. The variable `st_ptr` is indeed not a variable in Why program, it is simply a variable for translating a "relative" stack register to its physical one. In Why program, there will be only physical stack registers.

We also note that $-1 \leq st\_ptr \leq 7$ and we assume that the stack is empty at functions entrance and exit. Our translator statically computes the value of the top-of-stack pointer at each instruction. This value must be unique whatever is the path of the control-flow graph to reach the instruction.

Figure 4.7 shows us the changes of the stack after executing the instruction `fldl`. Assume that the current value of `st_ptr` is 5. This means that in Why, `st5` is on the top of the stack. If we access `%st(1)`, it will be `st4` in Why. After the instruction `fldl`, `st_ptr = 6` and now `st6` is on the top of the stack. If we want to access `%st(i)` then in Why it 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:

$[\![st]\!]_{st} \quad = \quad \texttt{st}\bar{i}$ where $\bar{i} = st\_ptr$

$[\![st(i)]\!]_{st} \quad = \quad \texttt{st}\bar{i}$ where $\bar{i} = st\_ptr - i$ and $0 \leq i \leq 7$

Translation of instructions to Why

The interpretation of instructions is presented as follows:

Data transfer instructions

$$
\begin{array}{lcl}
[\![\text{ flds src }]\!]_i & = & \texttt{set\_80 } [\![src]\!]_{single} \ [\![src]\!]_{exact} \ \texttt{st}(\overline{0}) \\
[\![\text{ fldl src }]\!]_i & = & \texttt{set\_80 } [\![src]\!]_{double} \ [\![src]\!]_{exact} \ \texttt{st}(\overline{0})
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{ fldlz }]\!]_i & = & \texttt{set\_80 (0.0) (0.0) st}(\overline{0}) \\
[\![\text{ fldl1 }]\!]_i & = & \texttt{set\_80 (1.0) (1.0) st}(\overline{0})
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{ fsts dest }]\!]_i & = & \texttt{set\_single } [\![st(\overline{0})]\!]_{binary80} \ [\![st(\overline{0})]\!]_{exact} \ \texttt{dest} \\
[\![\text{ fstl dest }]\!]_i & = & \texttt{set\_double } [\![st(\overline{0})]\!]_{binary80} \ [\![st(\overline{0})]\!]_{exact} \ \texttt{dest}
\end{array}
$$

Arithmetic instructions
The translation of arithmetic instructions is specified as follows:

$$
\begin{array}{lcl}
[\![\text{ fadds src }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{0}]\!]_{binary80}+[\![src]\!]_{single}) \ ([\![st\overline{0}]\!]_{exact}+[\![src]\!]_{exact}) \ st\overline{0} \\
[\![\text{ faddl src }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{0}]\!]_{binary80}+[\![src]\!]_{double}) \ ([\![st\overline{0}]\!]_{exact}+[\![src]\!]_{exact}) \ st\overline{0}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{ fsubs src }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{0}]\!]_{binary80}-[\![src]\!]_{single}) \ ([\![st\overline{0}]\!]_{exact}-[\![src]\!]_{exact}) \ st\overline{0} \\
[\![\text{ fsubl src }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{0}]\!]_{binary80}-[\![src]\!]_{double}) \ ([\![st\overline{0}]\!]_{exact}-[\![src]\!]_{exact}) \ st\overline{0}) \\
[\![\text{ fsubrs src }]\!]_i & = & \texttt{set\_80 } ([\![src]\!]_{single}-[\![st\overline{0}]\!]_{binary80}) \ ([\![src]\!]_{exact}-[\![st\overline{0}]\!]_{exact}) \ st\overline{0} \\
[\![\text{ fsubrl src }]\!]_i & = & \texttt{set\_80 } ([\![src]\!]_{double}-[\![st\overline{0}]\!]_{binary80}) \ ([\![src]\!]_{exact}-[\![st\overline{0}]\!]_{exact}) \ st\overline{0}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{ fmuls src }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{0}]\!]_{binary80}*[\![src]\!]_{single}) \ ([\![st\overline{0}]\!]_{exact}*[\![src]\!]_{exact}) \ st\overline{0} \\
[\![\text{ fmull src }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{0}]\!]_{binary80}*[\![src]\!]_{double}) \ ([\![st\overline{0}]\!]_{exact}*[\![src]\!]_{exact}) \ st\overline{0}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{ fdivs src }]\!]_i & = & \texttt{div\_80 } [\![src]\!]_{single} \ [\![src]\!]_{exact} \ st\overline{0} \\
[\![\text{ fdivl src }]\!]_i & = & \texttt{div\_80 } [\![src]\!]_{double} \ [\![src]\!]_{exact} \ st\overline{0} \\
[\![\text{ fdivrs src }]\!]_i & = & \texttt{divr\_80 } [\![src]\!]_{single} \ [\![src]\!]_{exact} \ st\overline{0} \\
[\![\text{ fdivrl src }]\!]_i & = & \texttt{divr\_80 } [\![src]\!]_{double} \ [\![src]\!]_{exact} \ st\overline{0}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{ fadd \%st(i), \%st(j) }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{j}]\!]_{binary80}+[\![st\overline{i}]\!]_{binary80}) \ ([\![st\overline{j}]\!]_{exact}+[\![st\overline{i}]\!]_{exact}) \ st\overline{j} \\
[\![\text{ faddp \%st(i), \%st(j) }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{j}]\!]_{binary80}+[\![st\overline{i}]\!]_{binary80}) \ ([\![st\overline{j}]\!]_{exact}+[\![st\overline{i}]\!]_{exact}) \ st\overline{j}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{ fsub \%st(i), \%st(j) }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{j}]\!]_{binary80}-[\![st\overline{i}]\!]_{binary80}) \ ([\![st\overline{j}]\!]_{exact}-[\![st\overline{i}]\!]_{exact}) \ st\overline{j} \\
[\![\text{ fsubp \%st(i), \%st(j) }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{j}]\!]_{binary80}-[\![st\overline{i}]\!]_{binary80}) \ ([\![st\overline{j}]\!]_{exact}-[\![st\overline{i}]\!]_{exact}) \ st\overline{j} \\
[\![\text{ fsubr \%st(i), \%st(j) }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{i}]\!]_{binary80}-[\![st\overline{j}]\!]_{binary80}) \ ([\![st\overline{i}]\!]_{exact}-[\![st\overline{j}]\!]_{exact}) \ st\overline{j} \\
[\![\text{ fsubrp \%st(i), \%st(j) }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{i}]\!]_{binary80}-[\![st\overline{j}]\!]_{binary80}) \ ([\![st\overline{i}]\!]_{exact}-[\![st\overline{j}]\!]_{exact}) \ st\overline{j}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{ fmul \%st(i), \%st(j) }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{j}]\!]_{binary80}*[\![st\overline{i}]\!]_{binary80}) \ ([\![st\overline{j}]\!]_{exact}*[\![st\overline{i}]\!]_{exact}) \ st\overline{j} \\
[\![\text{ fmulp \%st(i), \%st(j) }]\!]_i & = & \texttt{set\_80 } ([\![st\overline{j}]\!]_{binary80}*[\![st\overline{i}]\!]_{binary80}) \ ([\![st\overline{j}]\!]_{exact}*[\![st\overline{i}]\!]_{exact}) \ st\overline{j}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\text{ fdiv \%st(i), \%st(j) }]\!]_i & = & \texttt{div\_80 } [\![st\overline{i}]\!]_{binary80} \ [\![st\overline{i}]\!]_{exact} \ st\overline{j} \\
[\![\text{ fdivp \%st(i), \%st(j) }]\!]_i & = & \texttt{div\_80 } [\![st\overline{i}]\!]_{binary80} \ [\![st\overline{i}]\!]_{exact} \ st\overline{j} \\
[\![\text{ fdivr \%st(i), \%st(j) }]\!]_i & = & \texttt{divr\_80 } [\![st\overline{i}]\!]_{binary80} \ [\![st\overline{i}]\!]_{exact} \ st\overline{j} \\
[\![\text{ fdivrp \%st(i), \%st(j) }]\!]_i & = & \texttt{divr\_80 } [\![st\overline{i}]\!]_{binary80} \ [\![st\overline{i}]\!]_{exact} \ st\overline{j}
\end{array}
$$

4.4.6 AVX instructions

There are AVX instructions that are not FMA ones but they are used in the program compiled with FMA. The translations of them is described in the following table. Indeed, the specification of these instructions are not difference from the instructions in SSE/SSE2. The difference is that they have three operands and have the prefix 'v'.

$[\![$ vaddss src2, src1, dest $]\!]_i$ = `set_single` $([\![src1]\!]_{single}+[\![src2]\!]_{single})$ $([\![src1]\!]_{exact}+[\![src2]\!]_{exact})$ `dest`
$[\![$ vaddsd src2, src1, dest $]\!]_i$ = `set_double` $([\![src1]\!]_{double}+[\![src2]\!]_{double})$ $([\![src1]\!]_{exact}+[\![src2]\!]_{exact})$ `dest`

$[\![$ vsubss src2, src1, dest $]\!]_i$ = `set_single` $([\![src1]\!]_{single}-[\![src2]\!]_{single})$ $([\![src1]\!]_{exact}-[\![src2]\!]_{exact})$ `dest`
$[\![$ vsubsd src2, src1, dest $]\!]_i$ = `set_double` $([\![src1]\!]_{double}-[\![src2]\!]_{double})$ $([\![src1]\!]_{exact}-[\![src2]\!]_{exact})$ `dest`

$[\![$ vmulss src2, src1, dest $]\!]_i$ = `set_single` $([\![src1]\!]_{single}*[\![src2]\!]_{single})$ $([\![src1]\!]_{exact}*[\![src2]\!]_{exact})$ `dest`
$[\![$ vmulsd src2, src1, dest $]\!]_i$ = `set_double` $([\![src1]\!]_{double}*[\![src2]\!]_{double})$ $([\![src1]\!]_{exact}*[\![src2]\!]_{exact})$ `dest`

$[\![$ vdivss src2, src1, dest $]\!]_i$ = `div_single !src1 !src2 dest`
$[\![$ vdivsd src2, src1, dest $]\!]_i$ = `div_double !src1 !src2 dest`

The translation of FMA instructions is specified as follows:

$[\![$ vfmaddss src3, src2, src1, dest $]\!]_i$ = `set_single` $([\![src1]\!]_{single}*[\![src2]\!]_{single}+[\![src3]\!]_{single})$ $([\![src1]\!]_{exact}*[\![src2]\!]_{exact}+[\![src3]\!]_{exact})$ `dest`
$[\![$ vfmaddsd src3, src2, src1, dest $]\!]_i$ = `set_double` $([\![src1]\!]_{double}*[\![src2]\!]_{double}+[\![src3]\!]_{double})$ $([\![src1]\!]_{exact}*[\![src2]\!]_{exact}+[\![src3]\!]_{exact})$ `dest`
$[\![$ vfmsubss src3, src2, src1, dest $]\!]_i$ = `set_single` $([\![src1]\!]_{single}*[\![src2]\!]_{single}-[\![src3]\!]_{single})$ $([\![src1]\!]_{exact}*[\![src2]\!]_{exact}-[\![src3]\!]_{exact})$ `dest`
$[\![$ vfmsubsd src3, src2, src1, dest $]\!]_i$ = `set_double` $([\![src1]\!]_{double}*[\![src2]\!]_{double}-[\![src3]\!]_{double})$ $([\![src1]\!]_{exact}*[\![src2]\!]_{exact}-[\![src3]\!]_{exact})$ `dest`
$[\![$ vfnmaddss src3, src2, src1, dest $]\!]_i$ = `set_single` $(-([\![src1]\!]_{single}*[\![src2]\!]_{single})+[\![src3]\!]_{single})$ $(-([\![src1]\!]_{exact}*[\![src2]\!]_{exact})+[\![src3]\!]_{exact})$ `dest`
$[\![$ vfnmaddsd src3, src2, src1, dest $]\!]_i$ = `set_double` $(-([\![src1]\!]_{double}*[\![src2]\!]_{double})+[\![src3]\!]_{double})$ $(-([\![src1]\!]_{exact}*[\![src2]\!]_{exact})+[\![src3]\!]_{exact})$ `dest`
$[\![$ vfnmsubss src3, src2, src1, dest $]\!]_i$ = `set_single` $(-([\![src1]\!]_{single}*[\![src2]\!]_{single})-[\![src3]\!]_{single})$ $(-([\![src1]\!]_{exact}*[\![src2]\!]_{exact})-[\![src3]\!]_{exact})$ `dest`
$[\![$ vfnmsubsd src3, src2, src1, dest $]\!]_i$ = `set_double` $(-([\![src1]\!]_{double}*[\![src2]\!]_{double})-[\![src3]\!]_{double})$ $(-([\![src1]\!]_{exact}*[\![src2]\!]_{exact})-[\![src3]\!]_{exact})$ `dest`

4.4.7 Translation of annotations to Why in presence of floating-point arithmetic

The translation of annotations has already presented in Chapter 3 but that translation is only for 32-bit integer. In this part, we add some rules of the translation of annotations for floating-point types: *float* and *double*.

$[\![$*F(#τ#v#)*$]\!]_{annot}$	=	$\overline{F}$ `(`$[\![$*#τ#v#*$]\!]_{annot}$`)` where $\tau \in$ {`float`, `double`}, $\overline{F}$ is the Why function corresponding to F and type τ
$[\![$*#float#v#*$]\!]_{annot}$	=	`single_value(sel_single(`$\overline{v}$`))`
$[\![$*#double#v#*$]\!]_{annot}$	=	`double_value(sel_doule(`$\overline{v}$`))`
$[\![\backslash$*exact(#τ#v#)*$]\!]_{annot}$	=	`sel_exact(`$\overline{v}$`)` where $\tau \in$ {`float`, `double`}
$[\![\backslash$*model(#τ#v#)*$]\!]_{annot}$	=	`sel_model(`$\overline{v}$`)` where $\tau \in$ {`float`, `double`}
$[\![$*double*$]\!]_{annot}$	=	`real`
$[\![$*float*$]\!]_{annot}$	=	`real`

4.5 Results of examples of the chapter

4.5.1 Double rounding example

This example is presented in Figure 4.4. A screenshot in Figure 4.8 is the obligations proved by Gappa. This program is also automatically proved by Gappa in SSE mode if we use `//@ assert z == 1.0 + 0x1p-52` instead of `//@ assert z ==x`.

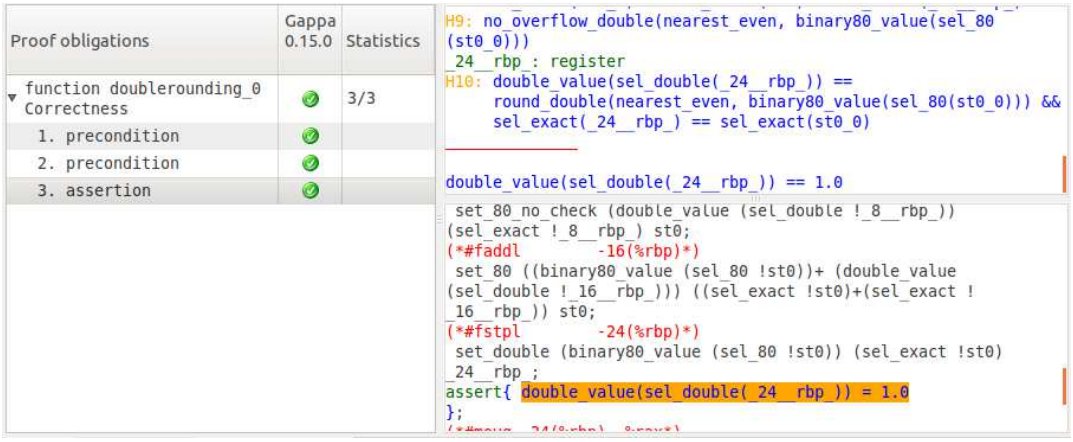


Figure 4.8: Result of Figure 4.4 program

4.5.2 Overflow example

For the non-optimized version, 5 obligations are generated to check absence of overflow at lines 4, 5, 7 and 8, and to check that divisor is not null 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. On the other hand, 4 obligations are generated on the optimized code at lines 3, 4 and 5 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. Finally, notice that we can also analyze the code compiled in the SSE mode, resulting in 3 obligations: overflows for the multiplication and division and check divisor is not null. As expected, it cannot be proved that multiplication does not overflow.

Chapter 5

Handling Conditional and loops

5.1 Definition of programs treated

In this chapter, we continue to extend the model from the previous chapters. This means that the programs may contain condition instructions, corresponding to the complex statements in C: `if then else`, `switch`, `for`, `do while`, etc.

5.2 Control Flow Graph construction

Definition 5.1 *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 denote 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 labelled 0. The exit nodes are those which contain a post-condition, and they never have a normal successor node.

Example 5.2 *A simple example with* `if` *statement is presented in Figure 5.1. This program returns the sign of a double value x provided that we know its rounding error is between e_1 and e_2.*

Assembly code of this example is in Figure 5.2. From this code, we construct a CFG by using Definition 5.1 as follows:

This CFG has 13 nodes, begins by node 0 and ends by node 12. Node 12 points to nothing to show that it is the last node. The nodes that are not the final node will point to one or two other nodes in the list.

For example, node 4 points to 8 and 10 because of the conditional jump instruction `je .L4`*. Node 8 contains the* `true` *value of* `je .L4` *and node 10 contains its* `false` *value.*

Example 5.3 *This example (See Figure 5.4) contains a loop statement* `do while`*. Its CFG is in Figure 5.5.*

Example 5.4 *This example (See Figure 5.6) finds the maximum element in an array of type* `double`*. As it contains both* `goto`*,* `if` *and* `do while`*, it is the general case of CFG. The CFG of this example is presented in Figure 5.7.*

```
//@ 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 5.1: Example with *if*

```
1  .globl sign
2          .type   sign, @function
3  sign:
4  .LFB0:
5          .cfi_startproc
6  ....
7          movsd   %xmm0, -8(%rbp)
8          movsd   %xmm1, -16(%rbp)
9          movsd   %xmm2, -24(%rbp)
10 #APP
11         /*requires #double#-16(%rbp)#<= #double#-8(%rbp)#-
12                           \exact(#double#-8(%rbp)#) <= #double#-24(%rbp)#;
13           ensures  (#int#\result# != 0
14                 ==> #int#\result# == l_sign(\exact(#double#-8(%rbp)#)))
15                     && \abs(#int#\result#) <= 1 ;*/
16 #NO_APP
17         movsd   -8(%rbp), %xmm0
18         movsd   -24(%rbp), %xmm1
19         ucomisd %xmm1, %xmm0
20         seta    %al
21         testb   %al, %al
22         je      .L2
23         movl    $1, %eax
24         jmp     .L3
25 .L2:
26         movsd   -8(%rbp), %xmm1
27         movsd   -16(%rbp), %xmm0
28         ucomisd %xmm1, %xmm0
29         seta    %al
30         testb   %al, %al
31         je      .L4
32         movl    $-1, %eax
33         jmp     .L3
34 .L4:
35         movl    $0, %eax
36 .L3:
37         leave
38         ret
39         .cfi_endproc
```

Figure 5.2: Assembly code of program in Figure 5.1

Node	Content	Goto
0	.LB0: (Line 10 – 14)	1
1	precondition	2
2	movsd -8(%rbp) , %xmm0 (Line 25 – 27) testb %al, %al	3,5
3	je .L2	4
4	.L2: (Line 33 – 36) testb %al, %al	8,10
5	NOT(je .L2)	6
6	movl $1, %eax	7
7	.L3:	12
8	je .L4	9
9	.L4:	7
10	NOT(je .L4)	11
11	movl $-1, %eax	7
12	post-condition	

Figure 5.3: CFG of Program in Figure 5.1

```
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 5.4: Program with loop statement

Node	Content	Goto
0	.LB0:	1
1	precondition	2
2	movl $0, x(%rip) movl $10, i(%rip)	3
3	.L2:	4
4	invariant	5

5	movl x(%rip), %eax
	addl $1, %eax
	....
		6, 7

6	jg .L2	3
7	NOT(jg .L2)	8
8	movl x(%rip), %eax	9
9	post-condition	

Figure 5.5: CFG of Program in Figure 5.4

```
/*@ requires n > 0;
  @ 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 5.6: Program with loop and `goto` statement

Node	Content	Goto
0	.LB0:	1
1	precondition	2
2	movl $0, -4(%rbp)	3
3	.L2:	8
4	.L4:	5, 7
5	je .L3	6
6	.L3:	9
7	NOT(j3 .L3)	3
8	L:	6
9	assertion	10
10	addl $1, -4(%rbp)	11, 12
11	jl .L4	4
12	NOT(jl .L4)	13
13	movq -16(%rbp), %rax	14
14 15	.L5: post-condition	15

Figure 5.7: CFG of Program in Figure 5.6

5.3 Translation from a CFG to Why

Our goal is now to build a Why program from a given CFG, so that the VCs generated from that Why program guarantees 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 [18, 3]

Assumption: on any cycle of the CFG, there is at least one invariant node.

Algorithm 5.5 *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.* [1]

```
produce_why_fun (post:Why predicate, instr: Why expressions)
  output (reverse (instr))
  output "{" post "}"

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;
```

[1] assume p := [unit p]

```
    (* 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, why_instr(i) :: prefix)
       case assert(p) :
           explore(n.succ, assert (why_pred(p)) :: prefix)
       case pre(p) :
           explore(n.succ, assume (why_pred(p)) :: prefix)
       case post(p) :
           produce_why_fun (why_pred(p),prefix)

       case inv(p) :
           produce_why_fun (why_pred(p), prefix) ;
           traverse_from(n)

       case jump(c,l) :
               explore_rec (l,
                            assume(why_cond(c)) :: prefix) ;
               explore_rec (n.succ
                            assume( not why_cond(c)) :: 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)
```

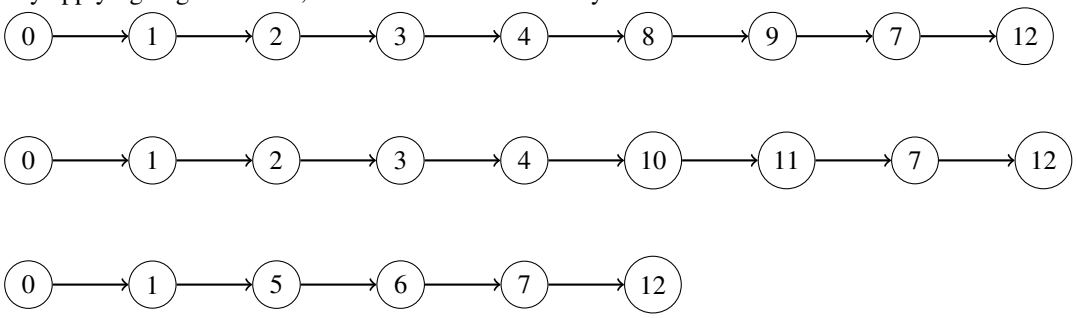
Theorem 5.6 (Soundness) *Let p be the assembly code of the function, g its CFG, and E the set of Why programs generated by generateWhy(g). If the VCs for the functions in E are valid, then p satisfies its annotations.*

5.4 Examples

5.4.1 KB3D

This example illustrates the handling of conditional statements, the handling of function calls, and the way we express properties on rounding errors across functions. It is an excerpt of the KB3D collision detection and resolution system developed by Dowek and Munoz [16] and formally proved in PVS, but using exact calculations on real numbers. An analysis of the same code but with floating-point calculations was done by Boldo and Nguyen [8] using their architecture-independent approach. The annotated C source is given on Figure 5.8. The logical symbol `l_sign` returns the sign of a real number: 1 for positive and -1 for negative (sign of zero is not pertinent). The C function `sign` returns the sign of a FP number x. To make sure that the result is correct, a precondition requires that the rounding error on previous computation on x (written as $x - \texttt{\textbackslash exact}(x)$) is between bounds e_1 and e_2 given as arguments. The C function `eps_line` then attempts to decide whether a aircraft at position sx, sy with velocity vx, vy should avoid the point (0,0) on the left or on the right. The decision is taken from the sign of some quantities, for which rounding errors must be taken into account, here in function of a constant E declared at the beginning. Our goal is to analyze what should be the value of E depending on the architecture.

The function `sign` of this example has been presented in Example 5.1 with its CFG in Figure 5.3. By applying Algorithm 5.5, it is divided into three Why functions:



This function is proved automatically and completely by using Alt-Ergo.

Feeding this annotated source code in our assembly analyser 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 by Boldo and Nguyen [8] 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 Optim. level	SSE2	x87 `-O0`	x87 `-O2`	FMA `-O2`
`E`	`0x1p-45`	`0x1.004p-46`	`0x1.004p-46`	`0x1.8p-46`

The FMA setting[2] asks to use *fused-multiply-add* operation, which computes expressions of the form $x * y + z$ with only one rounding [1]. As expected, using FMA improves over SSE2 (25% less) since less rounding occur. The extended precision of x87 is even better (around 50% less whatever the optimization level). Of course, all these bounds are smaller than the one found by Boldo and Nguyen for *any* architecture, which was `0x1.90641p-45` [8], that is more than 50% higher than the SSE2 one.

[2]obtained by options `-mfma4` of `gcc-4.5`, requires `-O2`

```
#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 5.8: Avionics program

Chapter 6

Handling Arrays and Pointers

6.1 Definition of this case

This chapter will present a new memory model with the programs having the following conditions:

- Support for arrays and pointers with C types: `int`, `long`, `double`, `float`
- There are no structure type, no pointer of pointer and no cast in the program

In order to use this model, the option **-stack** need to be presented in the command line.

The difference of this model from the previous one is that all memory references are represented in a memory model. It is similar to the way the program access and store value in the memory.

6.2 New translation to Why and new rules for instructions and operands

The form of memory reference in assembly language has presented in chapter 2. Basing on it, we present here the representation of memory in Why.

6.2.1 Representation of memory in Why

The memory of variables in assembly code is is discrete. In order to simplify the transformation to Why, we assume that this memory used in a program is continuous. We suppose that the memory has three parts:

- Memory references for local variables and function arguments are normally pointed by register `%rbp` (`%rsp` in case of optimization). This `%rbp` points to the last data item placed on the stack. The memory references pointed by `%rbp` are normally `-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 special register `%rip`. And *disp* is normally a symbol or non-negative value.
- Memory part which is reserved for allocated variables.

First of all, assume that we have two following pointers:

```
logic _rbp: int
logic _rip: int
```

Two axioms associated with them are:

```
axiom rbp_axiom: _rbp <= 0
axiom _rip_axiom: _rip = 4
```

We set the value of $\_rbp <= 0$. As `_rbp` and `_rip` do not point at the same position in the memory, we set $\_rip = 4$. All the memory references are based on these two pointers.

The memory references for variables are presented as follows:

Translation of array defined as global variables

Remind that `.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 * size of each element).

The symbols in assembly are interpreted like the way we did with registers `%rbp` and `%rip`.

To illustrate the translation of an array being a global variable, we have a small example in C:

```
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 6.1: An example containing array as global value.

In the example in Figure 6.1, we have three arrays: `ga1` having one element, `ga2` having two elements and `ga3` having three elements. The type of these arrays is `int` so the size each element is 4.

In assembly code, these arrays are declared as:

```
.comm ga1,4,4
.comm ga2,8,4
.comm ga3,12,4
```

We consider only two arguments after `.comm`. 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 bytes allocated for each symbol. They are defined in Why as follows:

```
logic ga1: int
logic ga2: int
logic ga3: int
```

Each global variable are defined as an variable where its type is `int` indicating the position of the first element in the memory.

An important question is: when we have many arrays in the same program, how can we assure 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 [21]. This model works well in C code source. We tried to apply this model in our assembly code but it doesn't work well. Because of this, we use only one memory space for all.

As we have explained, memory references accessed by `%rbp` and `%rip` has two different directions. Memory references pointed by these two pointers are always disjoint.

Now let us back to the previous example. Three arrays represented in this model are illustrated in Figure 6.2.

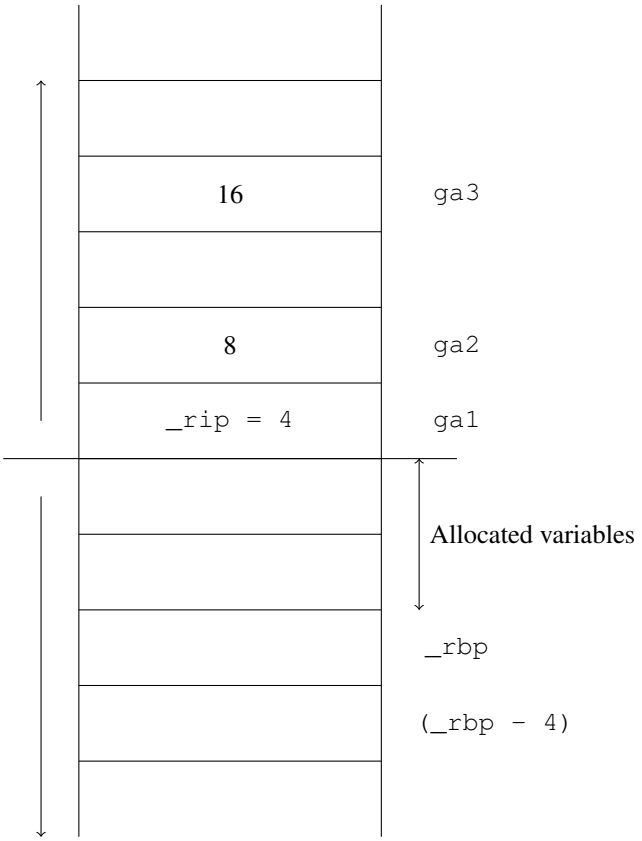


Figure 6.2: Memory model

We know that the size of `int` type is 4 so the total size of `ga1` is 1*4 = 4, the total size of `ga2` is 2*4 = 8, the total size of `ga3` is 3*4 = 12. We can also get the total size in the second argument of `.comm`.

In order to assure that the memory references for these arrays do not overlap, we need some constraints:

- 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.

The advantage of this model is that we need only five variables: `int32M`, `int64M`, `doubleM`, `singleM` and `exactM` which return an 32-bit integer value, 64-bit integer value, double value, single value, exact value, respectively, at a position in the memory.

In Why, we use axioms to specify these constraints:

```
axiom ga1_axiom: ga1 = 4
axiom ga2_axiom: ga2 = 8
axiom ga3_axiom: ga3 = 16
```

When these axioms are defined, the pointer `_rip` will be disappeared in operands of instructions. To access `ga2[1]` for example, it is simply written `(ga2 + 1*4)`.

Translation of array defined as local variables

An example containing arrays being local variable is in Figure 6.3. Its assembly code is in Figure 6.4.

```
void lg(int i) {
  int la1[1], la2[2], la3[3];

  la1[0] = i;
  la2[1] = i;
  la3[2] = i;

  //@ assert la1[0] == i;
  //@ assert la2[1] == i;
  //@ assert la3[2] == i;
}
```

Figure 6.3: C code of a program with arrays defined as local variables

Is it more simple to translate local arrays to Why because in assembly code, they are referenced by `_rbp`(See Figure 6.4) and the memory reference of an element of an array has been done by the compiler.

With the local variable `int la2[2]`, the first element of `la2` in 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 10 of Figure 6.4).

```
1  .globl lg
2          .type   lg, @function
3  lg:
4  .LFB0:
5  ....
6          movl    %edi, -52(%rbp)
7          movl    -52(%rbp), %eax
8          movl    %eax, -16(%rbp)
9          movl    -52(%rbp), %eax
10         movl    %eax, -28(%rbp)
11         movl    -52(%rbp), %eax
12         movl    %eax, -40(%rbp)
13 #APP
14         /* assert #int#-16(%rbp)#[0] == #int#-52(%rbp)#;*/
15         /* assert #int#-32(%rbp)#[1] == #int#-52(%rbp)#;*/
16         /* assert #int#-48(%rbp)#[2] == #int#-52(%rbp)#;*/
17 #NO_APP
18         leave
19         ret
20         .cfi_endproc
21 ....
```

Figure 6.4: Assembly code of Figure 6.3

Translation of arrays being arguments of a function

In assembly code, normally general-purpose registers are used for storing the address of arguments of a function. We consider its address is a 64-bit integer, so to get it from `%rax` for example, in Why, we write `(integer_of_int64 (sel_int64 _rax))`.

Assume that the arrays being arguments of a function have already allocated in the memory. Let see Figure 6.2, there is a memory space reserved for allocated variables. For example, if we have an array `arr` having `n` element and type `double`. We will add in the pre-condition that

```
(forall i:int. 0<i<n. integer_of_int64(sel_int64(arr)) + i*8 <= _rbp)
```

6.2.2 Definition of memory model

We define here a memory model that is used in the next section. This model is similar to the one in the built-in library `jessie.why`. In order to access to a value at a position in the memory, we need a variable which has the following type:

```
type 'v memory
```

where `'v` is a type which has already defined. In our approach, `'v` is either `int32`, `int64`, `single`, `double` or `real`.

We also need the following abstract functions to access the value in the memory or store it to memory.

```
logic select: 'v memory, int -> 'v
logic store: 'v memory, int, 'v -> 'v memory
```

The logic function `select` returns a `'v` value at a position specified by an integer in the memory `'v memory`. Otherwise, the logic function `store` set a `'v` value to the memory `'v memory` at the position specified by an integer.

There are axioms expressing the relation of two logic functions above.

```
axiom select_store_eq:
    forall m: 'v memory. forall p1: int. forall p2: int.
    forall a: 'v [store(m,p1,a),p2].
           p1=p2 -> select(store(m,p1,a),p2) = a

axiom select_store_neq:
    forall m: 'v memory. forall p1: int. forall p2: int.
    forall a: 'v [store(m,p1,a),p2] .
           p1 <> p2 -> select(store(m,p1,a),p2) = select(m,p2)
```

6.2.3 Translation of instructions and operands to Why

Type of memory

The address of a pointer or the one of the first element of an array being argument of a function will be store in general-purpose register. As we consider address is an 64-bit integer, there is no difference between the operation on addresses and the others. The translation of each instruction will be detailed in this section.

To return a value at a position in memory model, we need some following variables:

```
parameter int32M: int32 memory ref
parameter int64M: int64 memory ref
parameter singleM:single memory ref
parameter doubleM:double memory ref
parameter exactM: real memory ref
```

A memory reference $mem = d(b, i, s)$ is thus interpreted as the integer address $[\![mem]\!]_{addr} = b + d + i \times s$.

An operand is either a immediate constant, a register or a memory reference. Simple instructions for copying (with name typically starting with `mov`) and arithmetic operations have an output operand called *destination* and one or more input operands called *sources*. There is indeed 6 different interpretations of a source operand depending on the type of the expected value. We denote by $[\![opr]\!]_{int32}$, $[\![opr]\!]_{int64}$, $[\![opr]\!]_{single}$, $[\![opr]\!]_{double}$ and $[\![opr]\!]_{binary80}$ the interpretation of a source operand, respectively as a 32-bit, 64-bit integers and a 32-bit, 64-bit and 80-bit FP number. We also denote $[\![opr]\!]_{exact}$ its abstract `\exact` value.

$$
\begin{array}{rcl}
[\![imm]\!]_{int32} & = & imm \\
[\![imm]\!]_{int64} & = & imm \\
[\![imm]\!]_{single} & = & decode\_float32(imm) \\
[\![imm]\!]_{double} & = & decode\_float64(imm) \\
[\![reg]\!]_{int32} & = & \texttt{integer\_of\_int32(sel\_int32(!}reg\texttt{))} \\
[\![reg]\!]_{int64} & = & \texttt{integer\_of\_int64(sel\_int64(!}reg\texttt{))} \\
[\![reg]\!]_{single} & = & \texttt{single\_value(sel\_single(!}reg\texttt{))} \\
[\![reg]\!]_{double} & = & \texttt{double\_value(sel\_double(!}reg\texttt{))} \\
[\![reg]\!]_{binary80} & = & \texttt{binary80\_value(sel\_80(!}reg\texttt{))} \\
[\![reg]\!]_{exact} & = & \texttt{sel\_exact(!}reg\texttt{)} \\
[\![d(b,i,s)]\!]_{addr} & = & [\![b]\!]_{int64}\texttt{+d+s*}[\![i]\!]_{int64} \\
[\![mem]\!]_{int32} & = & \texttt{integer\_of\_int32(select(int32M,}[\![mem]\!]_{addr}\texttt{))} \\
[\![mem]\!]_{int64} & = & \texttt{integer\_of\_int64(select(int64M,}[\![mem]\!]_{addr}\texttt{))} \\
[\![mem]\!]_{single} & = & \texttt{single\_value(select(singleM,}[\![mem]\!]_{addr}\texttt{))} \\
[\![mem]\!]_{double} & = & \texttt{double\_value(select(doubleM,}[\![mem]\!]_{addr}\texttt{))} \\
[\![mem]\!]_{exact} & = & \texttt{select(exactM,}[\![mem]\!]_{addr}\texttt{)}
\end{array}
$$

Notations *decode_float32* and *decode_float64* are *not* Why logic functions but denote the operations of transforming a decimal literal into the real it represents respectively in single and double format. This decoding is done "at compile-time" in our translator from assembly to Why.

`mov` instruction

We define two parameters *move_mem_to_reg64* and *move_reg_to_mem32* which move the data (in 64 bits and in 32 bits) in register to memory:

```
parameter move_reg_to_mem64: a:register -> b:int->
{}
unit writes int64M,doubleM,exactM
{
   integer_of_int64(select(int64M, b)) = integer_of_int64(sel_int64(a))
   and
   double_value(select(doubleM, b)) = double_value(sel_double(a))
   and
   select(exactM, b) = sel_exact(a)
   and
   (forall i:int. i<>b ->
      integer_of_int64(select(int64M,i))=integer_of_int64(select(int64M@,i))
      and
      double_value(select(doubleM,i))=double_value(select(doubleM@,i))
      and
      select(exactM,i) =select(exactM@,i))
}

parameter move_reg_to_mem32: a:register -> b:int->
{}
unit writes int32M,singleM,exactM
{
```

```
   integer_of_int32(select(int32M, b)) = integer_of_int32(sel_int32(a))
   and
   single_value(select(singleM, b)) = single_value(sel_single(a))
   and
   select(exactM, b) = sel_exact(a)
   and
   (forall i:int. i<>b ->
      integer_of_int32(select(int32M,i))=integer_of_int32(select(int32M@,i))
      and
      single_value(select(singleM,i))=single_value(select(singleM@,i))
      and
      select(exactM,i) =select(exactM@,i))
}
```

The translation of `mov` instructions are specified as follows:

$[\![$ movl mem, reg $]\!]_i$ = `move_cte32` $[\![mem]\!]_{int64}$ $[\![mem]\!]_{single}$ $[\![mem]\!]_{exact}$ `reg`
$[\![$ movl reg, mem $]\!]_i$ = `move_reg_to_mem32 !reg` $[\![mem]\!]_{addr}$
$[\![$ movq mem, reg $]\!]_i$ = $[\![mem]\!]_{int64}$ $[\![mem]\!]_{double}$ $[\![mem]\!]_{exact}$ `reg`
$[\![$ movq reg, mem $]\!]_i$ = `move_reg_to_mem64 !reg` $[\![mem]\!]_{addr}$

Arithmetic instructions

For storing addresses to memory, we need two following parameters:

```
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 }
```

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 int32M
{
   integer_of_int32(select(int32M, b)) = integer_of_int32(sel_int32(a))
   and
   (forall i:int. i<>b ->
      integer_of_int32(select(int32M,i))=integer_of_int32(select(int32M@,i)))
}

parameter store_imm64: a:int -> b:int->
{is_int64(a)}
unit writes int64M
{
   integer_of_int64(select(int64M, b)) = integer_of_int64(sel_int64(a))
   and
   (forall i:int. i<>b ->
      integer_of_int64(select(int64M,i))=integer_of_int64(select(int64M@,i)))
}
```

We do similarly with `single` and `double`:

```
parameter store_single: a:real -> aexact:real -> b:int->
{no_overflow_single(nearest_even,a)}
```

```
unit writes singleM, exactM
{
   single_value(select(singleM, b)) = round_single(nearest_even,a)
   and
   select(exactM,b) = aexact
   and
   (forall i:int. i<>b ->
      single_value(select(singleM, i)) = single_value(select(singleM@, i))
      and
      select(exactM,i) =  select(exactM@,i))
}

parameter store_double: a:real -> aexact:real -> b:int->
{no_overflow_double(nearest_even,a)}
unit writes doubleM, exactM
{
   double_value(select(doubleM, b)) = round_double(nearest_even,a)
   and
   select(exactM,b) = aexact
   and
   (forall i:int. i<>b ->
      double_value(select(doubleM, i)) = double_value(select(doubleM@, i))
      and
      select(exactM,i) =  select(exactM@,i))
}
```

The translation of arithmetic instructions are as follow. We present here some general-purpose and SSE/SSE2 instructions:

$[\![$ addl reg, mem $]\!]_i$ = store_imm32 ($[\![mem]\!]_{int32}$ + $[\![reg]\!]_{int32}$) $[\![mem]\!]_{addr}$
$[\![$ addl src, reg $]\!]_i$ = set_int32 ($[\![src]\!]_{int32}$ + $[\![reg]\!]_{int32}$) reg
$[\![$ addq src, reg $]\!]_i$ = set_int64 ($[\![src]\!]_{int64}$ + $[\![reg]\!]_{int64}$) reg
$[\![$ addq reg, mem $]\!]_i$ = store_imm64 ($[\![mem]\!]_{int64}$ + $[\![reg]\!]_{int64}$) $[\![mem]\!]_{addr}$

$[\![$ subl reg, mem $]\!]_i$ = store_imm32 ($[\![mem]\!]_{int32}$ - $[\![reg]\!]_{int32}$) $[\![mem]\!]_{addr}$
$[\![$ subl src, reg $]\!]_i$ = set_int32 ($[\![reg]\!]_{int32}$ - $[\![src]\!]_{int32}$) reg
$[\![$ subq src, reg $]\!]_i$ = set_int64 ($[\![reg]\!]_{int32}$ - $[\![src]\!]_{int64}$) reg
$[\![$ subq reg, mem $]\!]_i$ = store_imm64 ($[\![mem]\!]_{int64}$ - $[\![reg]\!]_{int64}$) $[\![mem]\!]_{addr}$

$[\![$ addss reg, mem $]\!]_i$ = store_single($[\![mem]\!]_{single}$+$[\![reg]\!]_{single}$) ($[\![mem]\!]_{exact}$+$[\![reg]\!]_{exact}$) $[\![mem]\!]_{addr}$
$[\![$ addss mem, reg $]\!]_i$ = set_single ($[\![mem]\!]_{single}$+$[\![reg]\!]_{single}$) ($[\![mem]\!]_{exact}$+$[\![reg]\!]_{exact}$) reg
$[\![$ addsd reg, mem $]\!]_i$ = store_double($[\![mem]\!]_{double}$+$[\![reg]\!]_{double}$) ($[\![mem]\!]_{exact}$+$[\![reg]\!]_{exact}$) $[\![mem]\!]_{addr}$
$[\![$ addsd mem, reg $]\!]_i$ = set_double ($[\![mem]\!]_{double}$+$[\![reg]\!]_{double}$) ($[\![mem]\!]_{exact}$+$[\![reg]\!]_{exact}$) reg

$[\![$ subss reg, mem $]\!]_i$ = store_single($[\![mem]\!]_{single}$-$[\![reg]\!]_{single}$) ($[\![mem]\!]_{exact}$-$[\![reg]\!]_{exact}$) $[\![mem]\!]_{addr}$
$[\![$ subss mem, reg $]\!]_i$ = set_single ($[\![reg]\!]_{single}$-$[\![mem]\!]_{single}$) ($[\![reg]\!]_{exact}$-$[\![mem]\!]_{exact}$) reg
$[\![$ subsd reg, mem $]\!]_i$ = store_double($[\![mem]\!]_{double}$-$[\![reg]\!]_{double}$) ($[\![mem]\!]_{exact}$-$[\![reg]\!]_{exact}$) $[\![mem]\!]_{addr}$
$[\![$ subsd mem, reg $]\!]_i$ = set_double ($[\![reg]\!]_{double}$-$[\![mem]\!]_{double}$) ($[\![reg]\!]_{exact}$-$[\![mem]\!]_{exact}$) reg

Lea instruction

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.

$[\![$ leaq mem, reg $]\!]_i$ = set_int64 $[\![mem]\!]_{int64}$ reg
$[\![$ leal mem, reg $]\!]_i$ = set_int32 $[\![mem]\!]_{int32}$ reg

6.2.4 Translation of annotations to Why

The translation of variables in annotations to Why is quite different from the one in chapter 3. We denote by $[\![A]\!]_{int32}$ the 32-bit integer value of A, by $[\![A]\!]_{int64}$ the 64-bit integer value of A.

- $[\![constant]\!]_{int32} =$ `constant`
- $[\![reg]\!]_{int32} =$ `integer_of_int32(sel_int32(reg))`
- $[\![mem]\!]_{int32} =$ `integer_of_int32(select(int32M,` $[\![mem]\!]_{addr}$`))`
- $[\![constant]\!]_{int64} =$ `constant`
- $[\![reg]\!]_{int64} =$ `integer_of_int64(sel_int64(reg))`
- $[\![mem]\!]_{int64} =$ `integer_of_int64(select(int64M,` $[\![mem]\!]_{addr}$`))`

The translation of variables in annotations is specified as follows:

$[\![$*#int#reg#*$]\!]_{annot}$ = $[\![reg]\!]_{int32}$
$[\![$*#single#reg#*$]\!]_{annot}$ = `single_value(sel_single( reg))`
$[\![$*#double#reg#*$]\!]_{annot}$ = `double_value(sel_double(reg))`

$[\![$*#int#mem#*$]\!]_{annot}$ = `integer_of_int32(select(int32M,` $[\![mem]\!]_{addr}$`))`
$[\![$*#single#mem#*$]\!]_{annot}$ = `single_value(select(singleM,` $[\![mem]\!]_{addr}$`))`
$[\![$*#double#mem#*$]\!]_{annot}$ = `double_value(select(doubleM,` $[\![mem]\!]_{addr}$`))`

$[\![$*#int#reg#[V]*$]\!]_{annot}$ = `integer_of_int32(select(int32M,` $[\![reg]\!]_{int64}$`+`$[\![V]\!]_{int32}$`*4))`
$[\![$*#single#reg#[V]*$]\!]_{annot}$ = `single_value(select(singleM,` $[\![reg]\!]_{int64}$`+`$[\![V]\!]_{int32}$`*4))`
$[\![$*#double#reg#[V]*$]\!]_{annot}$ = `double_value(select(doubleM,` $[\![reg]\!]_{int64}$`+`$[\![V]\!]_{int32}$`*8))`

$[\![$*#int#symbol#[V]*$]\!]_{annot}$ = `integer_of_int32(select(int32M,`*symbol*`+`$[\![V]\!]_{int32}$`*4))`
$[\![$*#single#symbol#[V]*$]\!]_{annot}$ = `single_value(select(singleM,`*symbol*`+`$[\![V]\!]_{int32}$`*4))`
$[\![$*#double#symbol#[V]*$]\!]_{annot}$ = `double_value(select(doubleM,`*symbol*`+`$[\![V]\!]_{int32}$`*8))`

$[\![$*#int#disp(%rbp)#[V]*$]\!]_{annot}$ = `integer_of_int32(select(int32M,`*_rbp+disp+*$[\![V]\!]_{int32}$`*4))`
$[\![$*#single#disp(%rbp)#[V]*$]\!]_{annot}$ = `single_value(select(singleM,`*_rbp+disp+*$[\![V]\!]_{int32}$`*4))`
$[\![$*#double#disp(%rbp)#[V]*$]\!]_{annot}$ = `double_value(select(doubleM,`*_rbp+disp+* $[\![V]\!]_{int32}$`*8))`

6.3 Examples

6.3.1 Scalar Product

Our last example illustrates how we combine FP analysis with other features such as loops and arrays. The annotated C program on Fig. 6.5 computes the scalar product of two vectors represented as arrays of doubles. Similarly as the `l_sign` function of previous example, `exact_scalar_product`(x, y, n) is defined to denote the scalar product $\sum_{0 \leq i < n} x_i y_i$ computed in real numbers. The post-condition express a bound `B` on the accumulated rounding error in function of a bound `NMAX` on the size of the vectors. We also assume a bound, here 1.0, on each component 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 appears to be bounded by `NMAX`$(1 +$ `B`$)$.

The table below displays the value of `B` in function of `NMAX` and the architecture-dependent settings.

```
/*@ logic real exact_scalar_product{L}(double *x, double *y, integer n)
  @     reads x[..], y[..];
  @ axiom A1{L}: \forall double *x,*y;
  @      exact_scalar_product(x,y,0) == 0.0;
  @ axiom A2{L}: \forall double *x,*y; \forall integer n ;
  @      n >= 0 ==>
  @        exact_scalar_product(x,y,n+1) ==
  @          exact_scalar_product(x,y,n) + x[n]*y[n]; */

#define NMAX 10
#define NMAXR 10.0
#define B 0x1.1p-50

/*@ 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 ;
  @ ensures \abs(\result - exact_scalar_product(x,y,n)) <= n * B;  */
double scalar_product(double x[], double y[], int n) {
  double p = 0.0;
  /*@ loop invariant 0 <= i <= n ;
    @ loop invariant \abs(exact_scalar_product(x,y,i)) <= i;
    @ loop invariant \abs(p - exact_scalar_product(x,y,i)) <= i * B;
    @ loop variant n-i;
    @*/
  for (int i=0; i < n; i++) {
    //@ assert \abs(x[i]) <= 1.0 && \abs(y[i]) <= 1.0;
    //@ assert \abs(p) <= NMAXR*(1+B) ;
  L:
    p = p + x[i]*y[i];
    //@ assert \abs(p - (\at(p,L) + x[i]*y[i])) <= B;
    /*@ assert \abs(p - exact_scalar_product(x,y,i+1)) <=
               \abs(p - (\at(p,L) + x[i]*y[i])) +
               \abs((\at(p,L) + x[i]*y[i]) -
                    (exact_scalar_product(x,y,i) + x[i]*y[i])) ; */
    /*@ assert \abs(exact_scalar_product(x,y,i+1)) <=
          \abs(exact_scalar_product(x,y,i)) + \abs(x[i]) * \abs(y[i]); */
    //@ assert \abs(x[i]) * \abs(y[i]) <= 1.0;
  }
  return p;
}
```

Figure 6.5: Scalar product: annotated code

Architecture NMAX	SSE	x87 -O0	x87 -O2	FMA
10	0x1.1p-50	0x1.0022p-50	0x1.1p-61	0x1p-50
100	0x1.02p-47	0x1.0021p-47	0x1.02p-58	0x1p-47
1000	0x1.004p-44	0x1.00201p-44	0x1.004p-55	0x1p-44

The SSE mode, supposed to be strictly compliant with the standard, is worse than FMA and x87 without optimization, because the roundings are, as expected, slightly more precise. However 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.

Chapter 7

Conclusion

Former work on the verification of assembly code are mainly in the context of the so-called *proof-carrying-code* [12], where proof obligations for *safety* (of memory dereferencing, absence of overflow, etc.) are generated on the object code. However these does not consider any behavioral specification language to specify deeper properties than safety. The only work we know of that considers a specification language on object code is done in 2006 by Burdy and Pavlova [10] on Java bytecode. But they do not consider any aspect of FP computations. Thus, we believe that what we present in this paper is the first method being able to prove architecture- and compiler-dependent behavioral properties of FP programs.

Our approach and our prototype implementation demonstrates that handling architecture-dependent aspects is indeed possible. However it is clearly not mature enough for an 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 always able to interpret all the compiler optimizations. For example we do not support inlining of functions. We believe that to go further, we should integrate our approach into the compiler itself, following the ideas of proof-carrying-code: the optimizations made by the compiler should also produce annotations of the generated assembly (assertions, loop invariants) to make the optimizations explicit.

Bibliography

[1] IEEE standard for floating-point arithmetic. Technical report, 2008. http://dx.doi.org/10.1109/IEEESTD.2008.4610935.

[2] 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*, Lecture Notes in Artificial Intelligence, Edinburgh, Scotland, July 2010. Springer.

[3] 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.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[5] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. http://frama-c.cea.fr/acsl.html.

[6] 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.

[7] S. Boldo and T. M. T. Nguyen. Hardware-independent proofs of numerical programs. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, NASA Conference Publication, pages 14–23, Washington D.C., USA, Apr. 2010.

[8] S. Boldo and T. M. T. Nguyen. Proofs of numerical programs when the compiler optimizes. *Innovations in Systems and Software Engineering*, 7:151–160, 2011.

[9] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report NIII-R0309, Dept. of Computer Science, University of Nijmegen, 2003.

[10] L. Burdy and M. Pavlova. Java bytecode specification and verification. In *ACM symposium on Applied computing*, 2006.

[11] 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.

[12] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *ACM Conference PLDI*, 2000.

[13] I. Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Manual, Intel, 2011.

[14] 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.

[15] 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.

[16] 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.

[17] D. Elsner, J. Fenlason, and friends. Using as. Manual, 2009.

[18] 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`.

[19] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[20] J. Harrison. Formal verification of floating point trigonometric functions. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 217–233, Austin, Texas, 2000.

[21] T. Hubert. *Analyse Statique et preuve de Programmes Industriels Critiques*. Thèse de doctorat, Université Paris-Sud, June 2008.

[22] G. T. Leavens. Not a number of floating point problems. *Journal of Object Technology*, 5(2):75–83, 2006.

[23] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3):12, May 2008.

[24] S. P.Dandamudi. *Guide to Assembly Language Programming in Linux*. Spinger, 2005.

[25] 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.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399