Hardware-Dependent Proofs of Numerical Programs

Thi Minh Tuyen Nguyen, Claude Marché

▶ **To cite this version:**

Hardware-Dependent Proofs of Numerical Programs [*]

Thi Minh Tuyen Nguyen[1,2] and Claude Marché[1,2]

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

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 us to 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.

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 a 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 for that several possible reasons, e.g. the x87 floating-point unit (FPU) uses 80-bit internal floating-point registers, FMA instructions compute $xy \pm z$ with a single rounding, or the compiler may optimize the assembly code by changing the order of operations. Such issues have been extensively analyzed by D. Monniaux [27]. 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 follow the IEEE-754 standard, the result should be $1 + 2^{-52}$, but if compiled using the x87 FPU, a *double rounding* happens and the result is 1. The latter compilation does not *strictly* follows the standard[3].

In the context of static verification, FP computations have been considered in part. In analyses based on the abstract interpretation framework, support for FP computations is proposed in tools like Fluctuat [20], Astrée [17] and the Value analysis of Frama-C [18]. Generally speaking, FP arithmetic has been formalized since 1989 to formally prove hardware components or algorithms [14, 24, 32].

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 [12] for Java, ACSL [7] for C, Spec#[4] for C#. The support for floating-point computations in such approaches is poorly studied. In 2006, Leavens [25] enumerates a set of possible traps when one attempts to specify FP programs. In 2007, Boldo and Filliâtre [8] 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 $1 + 2^{-52}$.

In 2010, Boldo and Nguyen [9, 10] 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.

This paper is organized as follows. Section 2 presents the approach from a user's point of view, and is largely illustrated by examples. Section 3 explains the technicalities of our approach, which consists in translating annotations and assembly instructions into the Why intermediate language [23]. Conclusions and future work will be presented in the last section.

2   Overview of the approach

Fig. 1 presents all the steps to prove an annotated C program by analyzing its assembly code. The specification language we consider is ACSL [7], where

---

[3] The term *strict* here refers to the `-fp-model strict` or `/fp:strict` options of C compilers, or the `strictfp` keyword of Java, which explicitly requires the compilation to strictly conform to the standard.
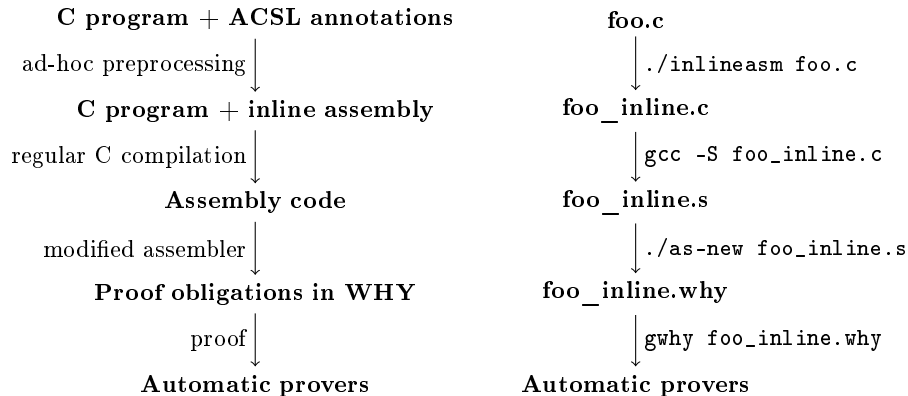
Fig. 1. Step-by-step from C program to WHY proof obligations

annotations are put in comments. To transport these annotations into the generated assembly, we have a preprocessing step in which we rewrite annotations as inline assembly. Assembly code is then generated from this new C source by the regular `gcc` compiler to generate assembly code, with precise architecture-related options, e.g. `-mfpmath=387` to generate x87 assembly or `-On` to optimize at level n. The main original step is then a translation of the assembly code to a Why program. This is implemented by modifying the GNU assembler so as to produce Why source instead of binary object. The Why environment is finally invoked to generate proof obligations, and prove them by automatic provers such as Gappa [26], Alt-Ergo [16], CVC3 [5], Z3 [19] or interactive provers like Coq [33].

2.1 Example: double-rounding

To illustrate this process, let's add a specification in the short program given in introduction under the form of an ACSL *assertion*:

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

The assembly code generated with our tools in the x87 mode is shown in Fig. 2. Notice on line 16 how the inline assembly preprocessing allowed to replace the occurrence of `z` in the assertion by its assembly counterpart. Notice also that the first addition is computed at compile-time (line 9), whereas the second one is compiled into x87 instructions (lines 11-14). When this assembly code is fed into our translator to Why, and the result analyzed by Why, three proof obligations are produced. One is naturally for proving the assertion, the two

3

```
1   .globl  doublerounding
2           .type   doublerounding,  @function
3   doublerounding :
4   .LFB0 :
5           .cfi_startproc
6   ....
7           movabsq  $4607182418800017408, %rax
8           movq    %rax,  −16(%rbp)
9           movabsq  $4368493837572636672, %rax
10          movq    %rax,  −8(%rbp)
11          fldl    −8(%rbp)
12          fld1
13          faddp   %st, %st(1)
14          fstpl   −24(%rbp)
15  #APP
16          /∗ assert #double#−24(%rbp)# == 1.0 ;∗/
17  #NO_APP
18          movq    −24(%rbp), %rax
19          movq    %rax,  −40(%rbp)
20          movsd   −40(%rbp), %xmm0
21          leave
22          ret
23          .cfi_endproc
```

Fig. 2. Assembly code of the double-rounding example in x87 mode

others are required to prove the absence of overflow, once at line 13 of Fig. 2, corresponding to the addition x+y in the source code, and once at line 14, which amount to store the 80-bit value of the x87 stack into a 64-bit memory cell. These three obligations are proved valid using the Gappa automatic prover.

If we compile the program in SSE2 mode (Streaming SIMD Extensions, 64-bits precision arithmetic) then the generated proof obligation corresponding to the assertion cannot be proved anymore. The modified assertion z == 1.0 + 0x1p-52 can be proved instead. As seen on this example, our approach produces proof obligations to show that the program satisfies its specification, but also to show the absence of overflow in FP computations.

2.2   Example: Architecture dependent Overflow

Monniaux [27] 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;
}
```

4

No optimization		**Optimized, level 1**	
1	`movabsq  $9214871658872686752, %rax`	1	`fldl    .LC0(%rip)`
2	`movq    %rax, −8(%rbp)`	2	`fld     %st(0)`
3	`fldl    −8(%rbp)`	3	`fmul    %st(1), %st`
4	`fmull   −8(%rbp)`	4	`fdivp   %st, %st(1)`
5	`fstpl   −16(%rbp)`	5	`fstpl   −8(%rsp)`
6	`fldl    −16(%rbp)`	6	`movsd   −8(%rsp), %xmm0`
7	`fdivl   −8(%rbp)`	7	`....`
8	`fstpl   −24(%rbp)`	8	`.LC0:`
9	`movsd   −24(%rbp), %xmm0`	9	`.long  2246822048`
10	`....`	10	`.long  2145504499`

Fig. 3. Optimized versus non-optimized assembly

Excerpts of the generated assembly code are shown on Fig. 3. The left part corresponds to non-optimized x87 code where $v*v$ is rounded in 64 bits whereas the right part is optimized (`-O1`) where $v*v$ is store in 80-bit registers and then the division is also done in 80-bit register. This is the reason why with no-optimization, overflow occurs but with optimization, it does not.

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 SSE2 mode, resulting in 3 obligations: overflows for the multiplication and division and check divisor is not null. As expected, it cannot be proved that multiplication does not overflow.

2.3   Example: KB3D

Our next example illustrates the handling of function calls, and the way we express properties on rounding errors across functions. This example is an excerpt of the KB3D collision detection and resolution system developed by Dowek and Munoz [21] 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 [10] using their architecture-independent approach. The annotated C source is given on Figure 4. 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 $x - \exact(x)$) 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

5

```
#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;
}
```

Fig. 4. Excerpt of KB3D program

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.

Feeding this annotated source code in our assembly analyser in SSE2 mode, each VC is automatically proved valid using either Gappa or one of the SMT solvers Alt-Ergo or CVC3. The bound E is indeed in that case exactly the same as the one found by Boldo and Nguyen [10] 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	2048×2^{-56}	1025×2^{-56}	1025×2^{-56}	1536×2^{-56}

The FMA setting[4] asks to use the *fused-multiply-add* operation, which computes expressions of the form $x * y \pm z$ with only one rounding [1]. As expected, using FMA improves over SSE2 (25% less) since fewer roundings occur. The extended precision of x87 is even better (around 50% less whatever the optimization level).

---

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

```
#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;
}
```

Fig. 5. Scalar product: annotated code

Of course, all these bounds are smaller than the one found by Boldo and Nguyen for *any* architecture, which was $0x1.90641p-45 \approx 3203 \times 2^{-56}$ [10], that is more than 50% higher than the SSE2 one.

2.4   Example: 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. 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 $\sum_{0 \le i < n} x_i y_i$ computed in real numbers. The post-condition expresses 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.

Architecture NMAX	SSE2	x87 -O0	x87 -O2	FMA
10	$2^{-50} + 2^{-54}$	$2^{-50} + 17 \times 2^{-65}$	17×2^{-65}	2^{-50}
100	$2^{-47} + 2^{-54}$	$2^{-47} + 33 \times 2^{-63}$	129×2^{-65}	2^{-47}
1000	$2^{-44} + 2^{-54}$	$2^{-44} + 513 \times 2^{-64}$	1025×2^{-65}	2^{-44}

The SSE2 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-bit precision for the complete execution of the loop: no intermediate rounding to 64-bit is done.

3 Underlying Technique

The core of our technique is to interpret the assembly code into the input language of Why. First we describe the general principles to follow, then we show how we interpret the various assembly statements. Due to the lack of space and the large number of different assembly instructions to handle, we only present a few of them. We focus on the support of FP arithmetic which is the point of interest in this paper. We refer to our technical report [30] for more details.

3.1 Principles of interpretation in Why

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 us to declare how the atomic operations on a given data type behave. It is exemplified below.

3.2 Model of data

Machine integers and Floating-Point Numbers The Why logic has unbounded mathematical integers and reals only. We reuse the modeling of machine integers provided by the Jessie plug-in of Frama-C [28]. This is done as follows for 32-bit integers; the type int64 is modelled similarly.

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

An abstract type `int32` for 32-bit integers is declared, together with a function `integer_of_int32` returning the value such a machine integer denotes. The predicate `is_int32` checks whether an integer is in the range of a 32-bit word or not, and we pose an axiom to specify that the value denoted by an `int32` is always in this range.

To model FP numbers, we reuse the modeling of 32- and 64-bit floats defined by Ayad and Marché [2], which introduces the corresponding abstract types `single` and `double`. We also complete this modeling by the type `binary80` for handling 80-bit floats. Here are the main parts of this model for 64-bit floats

```
type mode = nearest_even | to_zero | up | down | nearest_away
type double
logic double_value : double -> real
logic round_double : mode, real -> real
predicate no_overflow_double(m:mode,x:real) =
  abs(round_double(m,x)) <= 0x1.FFFFFFFFFFFFFp1023
```

An enumerated type `mode` is defined for the 5 possible rounding modes. The abstract type `double` for 64-bit floats is declared, together with the function `double_value` returning the real value it denotes. $\texttt{round\_double}(m, x)$ returns the closest to x representable number with *unbounded exponent* [2], w.r.t mode m. It is declared and partially axiomatized [2]. `Gappa` has this function built-in, that's why it is able to solve the VCs about rounding. The predicate `no_overflow_double` checks if overflow occurred when computing x.

Registers A central feature of our approach is how we model the CPU registers on which the assembly instructions operate. The issue is that a given register only stores a sequence of bits, that can be interpreted either as an integer, a FP number or a memory address. Moreover, for a given register one can either consider it as a 64-bit value or as the 32-bit value stored in its lower part, e.g. the `rax` versus `eax` register of the x86 chip.

To model that behavior, we introduce an abstract type `register` equipped with several access symbols, each of them denotes a different "view" of the value stored in the register. The symbol `sel_exact` is the view for calculations in infinite precision, to model the \exact construct of Fig. 4.

```
type register
logic sel_int32  : register -> int32
logic sel_int64  : register -> int64
logic sel_single : register -> single
logic sel_double : register -> double
logic sel_80     : register -> binary80
logic sel_exact  : register -> real
```

Then, for each register, we introduce a Why variable of type `register`, e.g. for the code of Fig. 2 two variables `xmm0` and `rax` are declared with type `register ref` (but not for `%rip` and `%st` which have a special meaning).

Model of the memory Interpreting memory access and update at the level of assembly is a major issue, since unlike high-level languages, we have no type information to help interpretation of raw data. In particular a given 64-bit word can be indifferently interpreted as an integer or a memory address. The memory is thus interpreted as a large array of data indexed by integers. However, without type information we would need to know how to encode and decode structured data, like FP numbers, into sequences of bits. Encoding and decoding are defined by complex computations that cannot be handle easily in a purely logical context: the generated VCs would be largely polluted with decoding and encoding hence would unlikely be proved by automatic provers.

We thus decide to keep a typed model of memory instead. This implies that we cannot handle C sources which non type-safe operations: pointer casts and union types. Our ad-hoc preprocessor keeps track of the C type of variables. The memory is then represented not only by one but by several arrays which contains different types of data. Each of these arrays is represented by a Why variable, e.g. a variable `int32M` holds an array of `int32` indexed by integers, another variable `doubleM` holds an array of `double` indexed by integers, etc. The Why type for such an array is declared as a polymorphic type:

```
type 'v memory
logic select: 'v memory -> int -> 'v
```

where `select` is the function to access the element at the given index. In assembly, a *memory reference* is an operand of the general form $disp(base, index, scale)$ where *base* and *index* are registers, and *disp* and *scale* are integer constants. *index* defaults to 0 and *scale* defaults to 1. A memory reference $mem = d(b, i, s)$ is thus interpreted as the integer address $b + d + i \times s$.

3.3 Interpretation of Assembly Instructions

Operands An operand is either an 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 are 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 by $[\![opr]\!]_{exact}$ its abstract `\exact` value.

$$[\![imm]\!]_{int32} = imm$$
$$[\![imm]\!]_{int64} = imm$$
$$[\![imm]\!]_{single} = decode\_float32(imm)$$
$$[\![imm]\!]_{double} = decode\_float64(imm)$$

10

$$\llbracket reg\rrbracket_{int32} = \texttt{integer\_of\_int32(sel\_int32(}!reg\texttt{))}$$
$$\llbracket reg\rrbracket_{int64} = \texttt{integer\_of\_int64(sel\_int64(}!reg\texttt{))}$$
$$\llbracket reg\rrbracket_{single} = \texttt{single\_value(sel\_single(}!reg\texttt{))}$$
$$\llbracket reg\rrbracket_{double} = \texttt{double\_value(sel\_double(}!reg\texttt{))}$$
$$\llbracket reg\rrbracket_{binary80} = \texttt{binary80\_value(sel\_80(}!reg\texttt{))}$$
$$\llbracket reg\rrbracket_{exact} = \texttt{sel\_exact(}!reg\texttt{)}$$
$$\llbracket d(b,i,s)\rrbracket_{addr} = \llbracket b\rrbracket_{int64}\texttt{+d+s*}\llbracket i\rrbracket_{int64}$$
$$\llbracket mem\rrbracket_{int32} = \texttt{integer\_of\_int32(select(int32M,}\llbracket mem\rrbracket_{addr}\texttt{))}$$
$$\llbracket mem\rrbracket_{int64} = \texttt{integer\_of\_int64(select(int64M,}\llbracket mem\rrbracket_{addr}\texttt{))}$$
$$\llbracket mem\rrbracket_{single} = \texttt{single\_value(select(singleM,}\llbracket mem\rrbracket_{addr}\texttt{))}$$
$$\llbracket mem\rrbracket_{double} = \texttt{double\_value(select(doubleM,}\llbracket mem\rrbracket_{addr}\texttt{))}$$
$$\llbracket mem\rrbracket_{exact} = \texttt{select(exactM,}\llbracket mem\rrbracket_{addr}\texttt{)}$$

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.

Move Instructions The move instructions have a mnemonic prefixed with `mov` and their suffix details the size of source and destination[5]. Their interpretation depends on whether the destination is a register or a memory reference.

Moving to a 64-bit register and to memory respectively are interpreted as procedure calls:

$$\llbracket \text{ movq imm, reg }\rrbracket_i = \texttt{move\_cte64 } \llbracket imm\rrbracket_{int64} \; \llbracket imm\rrbracket_{double} \; \llbracket imm\rrbracket_{double} \text{ reg}$$
$$\llbracket \text{ movq src, reg }\rrbracket_i = \texttt{move\_cte64 } \llbracket src\rrbracket_{int64} \; \llbracket src\rrbracket_{double} \; \llbracket src\rrbracket_{exact} \text{ reg}$$
$$\llbracket \text{ movq reg, } mem \rrbracket_i = \texttt{move\_reg\_to\_mem64 } !\text{reg } \llbracket mem\rrbracket_{addr}$$

where the Why procedure `move_cte64` is declared as

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

It reads as follows: calling this procedure modifies the register `r` (and nothing else) and after the call, the new content of `r` denotes both a 64-bit integer representing `a`, the FP double representing `b` and an exact real value `exact`. Notice how this interpretation abstracts away from bitwise representation details. Similarly, the Why procedure `move_reg_to_mem64` is declared as

```
parameter move_reg_to_mem64: r:register -> addr:int ->
{ } unit writes int64M, doubleM, exactM
{ integer_of_int64(select(int64M, addr)) = integer_of_int64(sel_int64(r))
  and double_value(select(doubleM, addr)) = double_value(sel_double(r))
  and select(exactM, addr) = sel_exact(r) and
```

---

[5] All the instructions we use in this paper are written in AT&T assembly syntax.

```
    forall a:int. not (addr <= a <= addr+7) ->
      integer_of_int64(select(int64M,a))=integer_of_int64(select(int64M@,a))
      and double_value(select(doubleM,a))=double_value(select(doubleM@,a))
      and select(exactM,a) =select(exactM@,a)) }
```

The @ sign in a Why post-condition denotes the value of a variable before the
call. Thus, the quantified part of the post-condition above amounts to specifying
that the rest of the memory is unmodified.

SSE2 Scalar Arithmetic Instructions Instructions for arithmetic operations
of the SSE family operate on 32- or 64-bit integers or floats, depending on the
suffix. The destination is always a register. Here is the interpretation of the
multiplication, other operations being similar (with an additional precondition
for division to check the divisor is not zero).

$$[\![\text{ mull src, reg }]\!]_i \;\; = \texttt{set\_int32}\ ([\![dest]\!]_{int32} * [\![src]\!]_{int32})\ reg$$
$$[\![\text{ mulsd src, reg }]\!]_i = \texttt{set\_double}\ ([\![dest]\!]_{double} * [\![src]\!]_{double})$$
$$([\![dest]\!]_{exact} * [\![src]\!]_{exact})\ reg$$

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

Notice that these procedures have pre-conditions to check for overflow. More-
over, the post-condition of set_double applies FP rounding: this is the way
we interpret the IEEE-754 standard: "the result of multiplication should be the
same as it was first computed in infinite precision then rounded to the result
format". (We hardwire the "nearest even" rounding mode here for simplicity.)

x87 Arithmetic Instructions The x87 FPU has 8 FP registers to hold FP
numbers in extended 80-bit precision. These registers are organized as a stack
ST0–ST7 and the current top of stack is identified internally with a special
register. In assembly code, %st or %st(0) denote the top of stack, whereas %st(i)
denote the i-th register below the top.
    We could have represented this stack by an array in Why with an additional
integer variable. However we can indeed identify statically the current top of
stack while translating into Why, so we just represent the stack by 8 variables
st0,...,st7 of type register ref. The only assumption we make in this opti-
mization is 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. We thus translate x87 registers %st(i) into Why variable st$\bar{i}$ where
$\bar{i} = top\_of\_stack - i$.

12

```
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 }
```

Fig. 6. Translation of a function in assembly to Why

For example, in optimized x87 assembly code compiled from Fig. 5, the top-of-stack pointer has value 1 at the loop entrance, because p is stored in the stack. Note that this way of interpreting the x87 stack, instead of considering the stack as an array, greatly improves the verification of VCs by Gappa back-end, since it does not know the theory of arrays as SMT solvers do.

Instructions for loading in the stack and storing from the stack are interpreted as follows.

$$
\begin{aligned}
&[[\ \text{fldl src}\ ]]_i && = \texttt{set\_80}\ [[src]]_{double}\ [[src]]_{exact}\ \texttt{st}\overline{0} \\
&[[\ \text{fld st(\%i)}\ ]]_i && = \texttt{set\_80}\ [[\text{st}\overline{i}]]_{binary80}\ [[\text{st}\overline{i}]]_{exact}\ \texttt{st}\overline{0} \\
&[[\ \text{fldl1}\ ]]_i && = \texttt{set\_80}\ (1.0)\ (1.0)\ \texttt{st}\overline{0} \\
&[[\ \text{fstl reg}\ ]]_i && = \texttt{set\_double}\ [[\text{st}\overline{0}]]_{binary80}\ [[\text{st}\overline{0}]]_{exact}\ !reg \\
&[[\ \text{fstl mem}\ ]]_i && = \texttt{set\_double\_mem}\ [[\text{st}\overline{0}]]_{binary80}\ [[\text{st}\overline{0}]]_{exact}\ [[mem]]_{addr}
\end{aligned}
$$

where set_80 is the analogous of set_double for 80-bit FP numbers.

Arithmetic instructions in the stack are interpreted as follows (for fmul, fadd and fsub being similar).

$$
\begin{aligned}
&[[\ \text{fmull src}\ ]]_i && = \texttt{set\_80}\ ([[\text{st}\overline{0}]]_{binary80}*[[src]]_{double}) \\
&&& \phantom{= \texttt{set\_80}\ } ([[\text{st}\overline{0}]]_{exact}*[[src]]_{exact})\ \texttt{st}\overline{0} \\
&[[\ \text{fmul \%st(i), \%st(j)}\ ]]_i && = \texttt{set\_80}\ ([[\text{st}\overline{j}]]_{binary80}*[[\text{st}\overline{i}]]_{binary80}) \\
&&& \phantom{= \texttt{set\_80}\ } ([[\text{st}\overline{j}]]_{exact}*[[\text{st}\overline{i}]]_{exact})\ \texttt{st}\overline{j}
\end{aligned}
$$

3.4 Translation of Annotated Functions

Assume that we have a function with preconditions, post-conditions and assertions. The translation of this function in assembly language to Why is illustrated in Figure 6. Our preprocessing moved the post-condition to the end of the function. More generally, each annotation is preprocessed into an inline assembly instruction of the form

```
asm("/* <keyword> P */"::"X"(x_0),..,"X"(x_n));
```

where each variable x_i of type τ in proposition P is replaced by #τ#%i#. The compiler thus transforms this inline assembly into lines between #APP and #NO_APP as on Fig. 2, where each %i is replaced by any appropriate memory

reference or register, even the 80-bit ones[6]. The Why interpretation of an annotation A is denoted by $[\![A]\!]_{annot}$. It is defined by a straightforward structural recursion, where the variables are interpreted as follows. in Why.

$$
\begin{aligned}
[\![\#int\#v\#]\!]_{annot} &= [\![v]\!]_{int32} \\
[\![\#float\#v\#]\!]_{annot} &= [\![v]\!]_{single} \\
[\![\#double\#v\#]\!]_{annot} &= [\![v]\!]_{double} \\
[\![\texttt{\textbackslash exact}(\#\tau\#v\#)]\!]_{annot} &= [\![v]\!]_{exact} \text{ where } \tau \in \{\text{float, double}\} \\
[\![\texttt{\textbackslash valid\_range}(\#\tau\#v\#, a, b)]\!]_{annot} &= \texttt{forall i, } a\texttt{<=i<}b \texttt{ ->} \\
&\qquad [\![e]\!]_{int64}\texttt{+i*}sizeof(\tau) \texttt{ >= !rsp}
\end{aligned}
$$

The interpretation of \valid_range allows to provide *separation* information for pointers [30], e.g. in Fig. 5, we need to know that local variable p is separated from x[0..9] and y[0..9]. Finally, the translation of code assertions and function calls are

$$
\begin{aligned}
[\![\texttt{ assert P }]\!]_i &= \texttt{assert } [\![P]\!]_{annot} \\
[\![\texttt{ call } f]\!]_i &= f\_\texttt{parameter ()}
\end{aligned}
$$

About Translation of Compound Statements When the source contains compound statements like conditional and loops, then the assembly code contains conditional jumps to labels. We cannot interpret such arbitrary jumps directly into Why since its programming language only support structured statements. We proceed differently by interpreting the arbitrary control flow graph of the assembly into a finite set of small pieces of codes, where loop invariants play the role of pre- or post-conditions. This technique is not the purpose of this paper and indeed is not original: we refer to our report [30] and earlier work done above Boogie [3] or Why [22] for more details.

4 Conclusion

An early work on verification of machine code is due to Boyer and Yu in 1992 [11]. They formalize the assembly language of a particular micro-processor and its operational semantics in Nqthm, and were able to verify a few programs, specified in Nqthm too. Their approach provides a deep embedding of assembly code, whereas ours is based on a shallow embedding: assembly code is simulated in Why. In our approach, behaviors are specified in the general-purpose ACSL language, and the proofs can be conducted with a large set of automated provers. Former studies on the verification of assembly code are in the context of the so-called *proof-carrying-code* [15], where proof obligations for *safety* (of memory dereferencing, absence of overflow, etc.) are generated on the object code. However these do not consider any behavioral specification language to specify deeper properties than safety. Although, it is worth noting that there is an

---

[6] As specified by the constraint "X", see http://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html

identified need to generate loop invariants in the target code to explicitate compilation choices [15, 31]. In 2006, Burdy and Pavlova [13] consider a specification language on Java bytecode. Barthe et al. [6] showed how proofs of VCs at source level can be reused for proving VCs at bytecode level, but they do not admit compiler optimizations. In 2008, Myreen [29] proposes to compile assembly code into functions in the HOL4 system. Our approach is somewhat close to this, using Why as target language instead of HOL4. Where Myreen must perform the proofs within HOL4, we can use various automatic provers thanks to the multi-prover feature of Why.

As far as we know, nobody ever considered any aspect of FP computations behavioral verification at the level of assembly. 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 demonstrate that handling architecture-dependent aspects is indeed possible. However it is clearly not mature enough for a non-expert user, because there are a lot of open issues. First, some languages features are not supported at the C level (like pointer casts) 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.

References

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 *Fifth IJCAR*, LNAI. Springer, July 2010.
3. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *6th PASTE*, 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 *CASSIS'04*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.
5. C. Barrett and C. Tinelli. CVC3. In *19th CAV*, LNCS. Springer, 2007.
6. G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In *FAST'05*, volume 3866 of *LNCS*, pages 112–126. Springer, 2005.
7. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ANSI/ISO C Specification Language*, 2009. http://frama-c.cea.fr/acsl.html.
8. S. Boldo and J.-C. Filliâtre. Formal Verification of Floating-Point Programs. In *18th ARITH*, pages 187–194, June 2007.
9. S. Boldo and T. M. T. Nguyen. Hardware-independent proofs of numerical programs. In *2nd NASA Formal Methods Symposium*, pages 14–23, Apr. 2010.
10. 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.
11. R. S. Boyer and Y. Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In *CADE*, pages 416–430, 1992.

12. 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.

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

14. V. A. Carreño and P. S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *HOL '95*, Sept. 1995.

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

16. S. Conchon and E. Contejean. Alt-ergo. APP Deposit, Mar. 2010. IDDN.FR.001.110026.000.S.P.2010.000.10000.

17. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, number 3444 in LNCS, pages 21–30, 2005.

18. P. Cuoq and V. Prevosto. *Value Plugin Documentation,* Carbon *version*. CEA-List, 2011. `http://frama-c.com/download/frama-c-value-analysis.pdf`.

19. L. de Moura and N. Bjørner. Z3, an efficient SMT solver. `http://research.microsoft.com/projects/z3/`.

20. 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.

21. G. Dowek and C. Muñoz. Conflict detection and resolution for 1,2,...,N aircraft. In *7th AIAA Aviation, Technology, Integration, and Operations Conference*, 2007.

22. J.-C. Filliâtre. Formal Verification of MIX Programs. In *Journées en l'honneur de Donald E. Knuth*, 2007. `http://knuth07.labri.fr/exposes.php`.

23. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177, 2007.

24. J. Harrison. Formal verification of floating point trigonometric functions. In *3rd FMCAD*, pages 217–233, 2000.

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

26. G. Melquiond. Proving bounds on real-valued functions with computations. In *4th IJCAR*, volume 5195 of *LNAI*, pages 2–17, 2008.

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

28. Y. Moy and C. Marché. *Jessie Plugin Tutorial,* Beryllium *version*. INRIA, 2009. `http://www.frama-c.cea.fr/jessie.html`.

29. M. O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2008.

30. T. M. T. Nguyen and C. Marché. Proving floating-point numerical programs by analysis of their assembly code. Research Report 7655, INRIA, 2011. `http://hal.inria.fr/inria-00602266/en/`.

31. X. Rival. Abstract interpretation-based certification of assembly code. In *Int. Conf. VMCAI*, 2003.

32. 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.

33. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. `http://coq.inria.fr`.