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

Thi Minh Tuyen Nguyen

présentée le 11 juin 2012 devant le jury composé de :

Sylvie BOLDO
Florent HIVERT
Xavier LEROY
Claude MARCHÉ
David MONNIAUX
Jean-Michel MULLER
César A. MUÑOZ
A case study

KB3D (NASA Langley Research Center)
- An aircraft conflict detection and resolution program
- Formally proved correct using PVS [Dowek & Muñoz, 2005]
- Provided the calculations are exact

Our case study
- Use a small part of KB3D
- Make a decision corresponding to value -1 and 1 to decide if the plane should go to its left or its right
A case study

**KB3D (NASA Langley Research Center)**
- An aircraft conflict detection and resolution program
- Formally proved correct using PVS [Dowek & Muñoz, 2005]
- Provided the calculations are exact

**Our case study**
- Use a small part of KB3D
- Make a decision corresponding to value -1 and 1 to decide if the plane should go to its left or its right
Introductory example

```c
int sign(double x) {
    if (x >= 0) return 1;
    else return -1;
}

int eps_line(double sx, double sy, double vx, double vy) {
    int s1, s2;
    s1 = sign(sx * vx + sy * vy);
    s2 = sign(sx * vy - sy * vx);
    return s1 * s2;
}
```
Introductory example

```c
int sign(double x) {
    if (x >= 0) return 1;
    else return -1;
}

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

![Graph showing sign(x) function and line equation](image)
**Introductory example**

```c
int sign(double x) {
    if (x >= 0) return 1;
    else return -1;
}

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

**Make a direction decision**
Introductory example

```c
int sign(double x) {
    if (x >= 0) return 1;
    else return -1;
}

int eps_line(double sx, double sy, double vx, double vy) {
    int s1, s2;
    s1 = sign(sx * vx + sy * vy);
    s2 = sign(sx * vy - sy * vx);
    return s1 * s2;
}

int main() {
    double sx = -0x1.00000000000001p0;    // sx = -1 - 2^{-52}
    double vx = -1.0;
    double sy = 1.0;
    double vy = 0x1.ffffffffpp -1;         // vy = 1 - 2^{-53}
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = %d\n", result);
}
```
Introductory example

```
int sign(double x) {
    if (x >= 0) return 1;
    else return -1;
}

int eps_line(double sx, double sy, double vx, double vy) {
    int s1, s2;
    s1 = sign(sx * vx + sy * vy);
    s2 = sign(sx * vy - sy * vx);
    return s1 * s2;
}

int main() {
    double sx = -0x1.0000000000001p0; // sx = -1 - 2^-52
    double vx = -1.0;
    double sy = 1.0;
    double vy = 0x1.ffffffffffffffff -1; // vy = 1 - 2^-53
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = %d\n", result);
}
```
# Introductory example

**Function `sign`**

```c
int sign(double x) {
    if (x == 0)
        return 1;
    else
        return -1;
}
```

**Function `eps_line`**

```c
int eps_line(double sx, double sy, double vx, double vy) {
    int s1, s2;
    s1 = sign(sx * vx + sy * vy);
    s2 = sign(sx * vy - sy * vx);
    return s1 * s2;
}
```

**Main function**

```c
int main() {
    double sx = -0x1.00000000000001p0; // sx = -1 - 2^{-52}
    double vx = -1.0;
    double sy = 1.0;
    double vy = 0x1.fffffffffffffffp -1; // vy = 1 - 2^{-53}
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = %d\n", result);
}
```

**Example compilation and output**

- `gcc -mfpmath=387 eps_line.c`

  ```
  gcc -mfpmath=387 eps_line.c
  ```

  Result = -1
Introductory example

```c
int sign(double x) {
    if (x > 0)
        return 1;
    else
        return -1;
}

int eps_line(double sx, double sy, double vx, double vy) {
    int s1, s2;
    s1 = sign(sx * vx + sy * vy);
    s2 = sign(sx * vy - sy * vx);
    return s1 * s2;
}

int main() {
    double sx = -0x1.00000000000001p0; // sx = \( -1 - 2^{-52} \)
    double vx = -1.0;
    double sy = 1.0;
    double vy = 0x1.f8p-1; // vy = \( 1 - 2^{-53} \)
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = %d\n", result);
}
```

```
gcc -mfpmath=387 eps_line.c
```

Result = -1

?!?!
Famous computer arithmetic failures

- Patriot Missile Failure in 1991:
  - inaccurate calculation of the time since boot due to computer arithmetic errors
  - 28 soldiers dead, > 90 injured

- explosion of the Ariane 5 in 1996:
  - conversion from 64-bit floating-point to 16-bit signed integer value
  - $500 millions

- FDIV bug in 1994 in the Intel Pentium processors:
  - A few floating-point divisions produced incorrect results
  - $500 millions
Famous computer arithmetic failures

- Patriot Missile Failure in 1991:
  - inaccurate calculation of the time since boot due to computer arithmetic errors
  - 28 soldiers dead, > 90 injured

- explosion of the Ariane 5 in 1996:
  - conversion from 64-bit floating-point to 16-bit signed integer value
  - $500 millions

- FDIV bug in 1994 in the Intel Pentium processors:
  - A few floating-point divisions produced incorrect results
  - $500 millions
Famous computer arithmetic failures

- Patriot Missile Failure in 1991:
  - inaccurate calculation of the time since boot due to computer arithmetic errors
  - 28 soldiers dead, > 90 injured

- explosion of the Ariane 5 in 1996:
  - conversion from 64-bit floating-point to 16-bit signed integer value
  - $500 millions

- FDIV bug in 1994 in the Intel Pentium processors:
  - A few floating-point divisions produced incorrect results
  - $500 millions
Famous computer arithmetic failures

- Patriot Missile Failure in 1991:
  - inaccurate calculation of the time since boot due to computer arithmetic errors
  - 28 soldiers dead, > 90 injured

- explosion of the Ariane 5 in 1996:
  - conversion from 64-bit floating-point to 16-bit signed integer value
  - $500 millions

- FDIV bug in 1994 in the Intel Pentium processors:
  - A few floating-point divisions produced incorrect results
  - $500 millions

⇒ Safety-critical systems require verification and certification
Architecture and rounding precision issues

- All current processors support the floating-point arithmetic standard IEEE-754
- Some architecture-dependent issues:
  - x87 floating-point unit (x87 FPU) uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - Fused multiply-add (FMA) instruction supported by the PowerPC and the Intel Itanium architecture
    - calculates \((x \times y \pm z)\) with a single rounding
- Some compiler optimization issues such as compiler reparenthesizing
Architecture and rounding precision issues

- All current processors support the floating-point arithmetic standard IEEE-754

- Some architecture-dependent issues:
  - x87 floating-point unit (x87 FPU) uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - Fused multiply-add (FMA) instruction supported by the PowerPC and the Intel Itanium architecture
    - calculates \((x \times y \pm z)\) with a single rounding

- Some compiler optimization issues such as compiler reparenthesizing
Architecture and rounding precision issues

- All current processors support the floating-point arithmetic standard IEEE-754

- Some architecture-dependent issues:
  - x87 floating-point unit (x87 FPU) uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - Fused multiply-add (FMA) instruction supported by the PowerPC and the Intel Itanium architecture
    - calculates \((x \times y \pm z)\) with a single rounding

- Some compiler optimization issues such as compiler reparenthesizing
Architecture and rounding precision issues

- All current processors support the floating-point arithmetic standard IEEE-754
- Some architecture-dependent issues:
  - x87 floating-point unit (x87 FPU) uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - Fused multiply-add (FMA) instruction supported by the PowerPC and the Intel Itanium architecture
    - calculates \((x \times y \pm z)\) with a single rounding
- Some compiler optimization issues such as compiler reparenthesizing
  \(\Rightarrow\) introduce subtle inconsistencies between program executions
Frama-C

- A framework for static analysis of C programs
- Flexible: easy to add new plug-ins

**Value analysis:**
uses abstract interpretation techniques
computes variation domains for variables

**Jessie:**
is a deductive verification plug-in

etc.
Previous works

Annotated C program

Frama-C/Jessie

Why VC generator

Automatic/interactive provers

Why VC generator

Frama-C/Jessie

Annotated C program

Previous works

Automatic/interactive provers

Formal verification of FP Programs

[Boldo & Filliâtre, 2007]

Behavioral Properties of FP Programs

[Ayad & Marché, 2010]
Previous works

Annotated C program

- Formal verification of FP Programs [Boldo & Filliâtre, 2007]
- Behavioral Properties of FP Programs [Ayad & Marché, 2010]

Frana-C/Jessie

Why VC generator

Automatic/interactive provers
Previous works

Annotated C program

Formal verification of FP Programs [Boldo & Filliâtre, 2007]

only follow the strict IEEE-754

Behavioral Properties of FP Programs [Ayad & Marché, 2010]

Frama-C/Jessie

Why VC generator

Automatic/interactive provers
Contributions of the thesis

Annotated C program

Why VC generator

Automatic/interactive provers

Take the architecture and compiler into account
Contributions of the thesis

- Annotated C program
- multirounding model
- Frama-C/Jessie
- Why VC generator
- Static analysis of C code
- Automatic/interactive provers
- Hardware-independent approach
- Hardware-dependent approach
Contributions of the thesis

- Annotated C program
- multirounding model
- Why VC generator
- Automatic/interactive provers

Hardware-independent approach:
- Static analysis of C code
- Frama-C/Jessie

Hardware-dependent approach:
- analysis of assembly code
- GNU Assembler

Why VC generator

Contributions of the thesis:

- Automatic/interactive provers
- Frama-C/Jessie
- Static analysis of C code
- GNU Assembler
- Analysis of assembly code
- Hardware-independent approach
- multirounding model
- Hardware-dependent approach
Outline

1. Introduction
2. Preliminaries
3. Hardware-independent proofs
4. Hardware-dependent proofs
5. Conclusions and Future works
Outline

1. Introduction
2. Preliminaries
3. Hardware-independent proofs
4. Hardware-dependent proofs
5. Conclusions and Future works
ANSI/ISO C Specification Language (ACSL)

- ACSL is a specification language
- ACSL allows to formally specify the properties of a C program
- Frama-C uses ACSL annotations to formally verify that the implementation respects these annotations.

```c
double square(double x){
  return x*x;
}
```
ANSI/ISO C Specification Language (ACSL)

- ACSL is a specification language
- ACSL allows to formally specify the properties of a C program
- Frama-C uses ACSL annotations to formally verify that the implementation respects these annotations.

```c
/*@ requires \abs(x) <= 1.0;
@ ensures \abs(result - x*x) <= 0x1p-53; //2^{-53}
@*/
double square(double x){
    return x*x;
}
```
ANSI/ISO C Specification Language (ACSL)

- ACSL is a specification language
- ACSL allows to formally specify the properties of a C program
- Frama-C uses ACSL annotations to formally verify that the implementation respects these annotations.

Precondition

```c
/*@ requires \abs(x) <= 1.0;
@ ensures \abs(result - x*x) <= 0x1p-53; //2^{-53}
@*/
double square(double x){
    return x*x;
}
```
ANSI/ISO C Specification Language (ACSL)

- ACSL is a specification language
- ACSL allows to formally specify the properties of a C program
- Frama-C uses ACSL annotations to formally verify that the implementation respects these annotations.

```c
/*@ requires \abs(x) <= 1.0;
@ ensures \abs(result - x*x) <= 0x1p-53; */

double square(double x){
    return x*x;
}
```
ANSI/ISO C Specification Language (ACSL)

- ACSL is a specification language
- ACSL allows to formally specify the properties of a C program
- Frama-C uses ACSL annotations to formally verify that the implementation respects these annotations.

```c
/*@ requires \abs(x) \leq 1.0;
@ ensures \abs(result - x*x) \leq 0x1p-53; //2^{-53}
@*/
double square(double x){
    return x*x;
}
```
Translation into Why

General principle:

Use Why to program an axiomatic semantics of instructions

Why logic language:

- first-order logic with equality, integer and real arithmetic
- declaration of abstract data types
- axiomatized functions and predicates

Why programming language:

- set of procedures/functions in a basic imperative language
- abstract Why subprograms, specified by pre-conditions, effects, and post-conditions
Floating-point numbers

Definition

A floating-point number $x$ in a format $(p, e_{\text{min}}, e_{\text{max}})$ is represented by the triple $(s, m, e)$ so that

$$x = (-1)^s \times 2^e \times m$$

- $s \in \{0, 1\}$
- $e_{\text{min}} \leq e \leq e_{\text{max}}$
- $0 \leq m < 2$, represented by $p$ bits

Normal number vs. Subnormal number

$$|x| \geq 2^{e_{\text{min}}}$$

$x$ is normal number

$$\frac{2^{e_{\text{min}}}}{(2 - 2^{1-p}) \times 2^{e_{\text{max}}}}$$

$0 \quad 2^{e_{\text{min}}} \quad (2 - 2^{1-p}) \times 2^{e_{\text{max}}} \quad +\infty$
Floating-point numbers

Definition

A floating-point number $x$ in a format $(p, e_{\text{min}}, e_{\text{max}})$ is represented by the triple $(s, m, e)$ so that

$$x = (-1)^s \times 2^e \times m$$

- $s \in \{0, 1\}$
- $e_{\text{min}} \leq e \leq e_{\text{max}}$
- $0 \leq m < 2$, represented by $p$ bits

Normal number vs. Subnormal number

$$2^{e_{\text{min}}} \quad \leftarrow \quad (2 - 2^{1-p}) \times 2^{e_{\text{max}}} \quad \rightarrow \quad +\infty$$

$$|x| < 2^{e_{\text{min}}}$$

$x$ is subnormal number
Floating-point numbers in Why

Abstract the IEEE-754 bit-level representation [Ayad & Marché, 2010]

```
type double
function double_value: double -> real
constant max_double: 0x1F.FFFFFFFFp1023
axiom double_range:
    forall x:double. |double_value(x)| <= max_double
function o64: real -> real
```

**We consider only round-to-nearest rounding mode**
Floating-point numbers in Why

\texttt{parameter add\_double(x:double, y:double) :}
\{\texttt{|0_{64}(double\_value(x) + double\_value(y))| \leq max\_double}\}
\texttt{double}
\{\texttt{double\_value(result) =}
\texttt{0_{64}(double\_value(x) + double\_value(y))}\}\n
[Ayad & Marché, 2010]
Floating-point numbers in Why

precondition: does not overflow

```plaintext
parameter add_double(x:double, y:double) :

{ |`64(double_value(x) + double_value(y))| ≤ max_double }
double
{ double_value(result) =
`64(double_value(x) + double_value(y)) }
```

[Ayad & Marché, 2010]
Floating-point numbers in Why

follows the IEEE-754 standard

```plaintext
parameter add_double(x: double, y: double) :
{\[|^{64}(\text{double_value}(x) + \text{double_value}(y))| \leq \text{max_double}\} 

double
{\text{double_value(result)} = \[^{64}(\text{double_value}(x) + \text{double_value}(y))\}}
```

[Ayad & Marché, 2010]
Floating-point numbers in Why

follows the IEEE-754 standard

```
parameter add_double (x: double, y: double) :
  { |\circ 64 (double_value(x) + double_value(y)) | \leq \text{max_double} }
  double
  { double_value(result) =
    \circ 64 (double_value(x) + double_value(y))}
```

Gappa supports $\circ 64$

Gappa is a tool for proving properties on numerical programs [Melquiond, 2006]

[Ayad & Marché, 2010]
Floating-point numbers in Why

follows the IEEE-754 standard

\[
\text{parameter add_double} \ (x: \text{double}, \ y: \text{double}) : \begin{cases} \ |\circ_{64}(\text{double_value}(x) + \text{double_value}(y))| \leq \text{max_double} \end{cases}
\]

double
\[
\text{double_value} (\text{result}) = \circ_{64}(\text{double_value}(x) + \text{double_value}(y))
\]

\[|\text{double_value(result)} - (\text{double_value}(x) + \text{double_value}(y))| \leq ?\]
Outline

1. Introduction
2. Preliminaries
3. Hardware-independent proofs
4. Hardware-dependent proofs
5. Conclusions and Future works
Double rounding example

```c
int main(){
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64; // 2^{-53} + 2^{-64}
    double z = x + y;

    printf("z=%a\n", z);
}
```

\[ y = 2^{-53} + 2^{-64} \]
Double rounding example

```c
int main() {
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64; // 2^{-53} + 2^{-64}
    double z = x + y;

    printf("z=%a\n", z);
}
```

```
gcc double_rounding.c
```

64-bit rounding

\[
\text{\texttt{\textcircled{64}(x + y)}}
\]

\[
y = 2^{-53} + 2^{-64}
\]

\[
z = 1.0 + 2^{-52}
\]
Double rounding example

```c
int main()
{
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64; // 2^{-53} + 2^{-64}
    double z = x + y;

    printf("z=%a\n", z);
}
```

gcc -mfpmath=387 double_rounding.c
Double rounding example

```c
int main()
{
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64;  // 2^{-53} + 2^{-64}
    double z = x + y;

    printf("z=\%a\n", z);
}
```

```
gcc double_rounding.c
```
Double rounding example

```c
int main()
{
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64; // 2^{-53} + 2^{-64}
    double z = x + y;

    printf("z=%a\n", z);
}
```

 GCC `gcc double_rounding.c`  

```bash
gcc -mfpmath=387 double_rounding.c
```

Double rounding

\[ y = 2^{-53} + 2^{-64} \]

1 + 2^{-52}

\[ z = 1.0 \]
Idea of hardware-independent approach

State the rounding error of each floating-point computation whatever the environment

- 64-bit rounding
- 80-bit rounding
- double rounding
- FMA
Rounding error in round-to-nearest mode

**Rounding error in normal range**

Use relative error

\[
\left| \frac{x - \circ(x)}{x} \right| \leq 2^{-p} = \varepsilon
\]

**Rounding error in subnormal range**

Use absolute error

\[
|x - \circ(x)| \leq 2^{e_{\text{min}}-p} = \eta
\]

Well-known consequence of the standard
Rounding error in round-to-nearest mode

IEEE-754 double precision (64-bit rounding)

Normal range:

\[ \left| \frac{x - o_{64}(x)}{x} \right| \leq 2^{-53} \]

Subnormal range:

\[ |x - o_{64}(x)| \leq 2^{-1075} \]

x87 FPU (80-bit rounding)

Normal range:

\[ \left| \frac{x - o_{80}(x)}{x} \right| \leq 2^{-64} \]

Subnormal range:

\[ |x - o_{80}(x)| \leq 2^{-16446} \]
Rounding error in double rounding

Based on 64-bit and 80-bit rounding, with \( \alpha = 2^{-1022} \)

- \(|x| \geq \alpha \Rightarrow \left| \frac{x - \circ_64(\circ_{80}(x))}{x} \right| \leq 2050 \times 2^{-64} \)
- \(|x| \leq \alpha \Rightarrow |x - \circ_64(\circ_{80}(x))| \leq 2049 \times 2^{-1086} \)
### Theorem 1

For a real number $x$, let $\boxplus(x)$ be either $\circ_{64}(x)$, or $\circ_{80}(x)$, or the double rounding $\circ_{64}(\circ_{80}(x))$.

With $\alpha = 2^{-1022}$, we have either

$$|x| \geq \alpha \text{ and } \left| \frac{x - \boxplus(x)}{x} \right| \leq \beta \text{ and } |\boxplus(x)| \geq \alpha$$

or

$$|x| \leq \alpha \text{ and } |x - \boxplus(x)| \leq \gamma \text{ and } |\boxplus(x)| \leq \alpha.$$

with $\beta = 2050 \times 2^{-64} \ (\gtrsim 2^{-53})$

$\gamma = 2049 \times 2^{-1086} \ (\gtrsim 2^{-1075})$
Rounding error of one operation

Theorem 1

For a real number $x$, let $\square(x)$ be either $\circ_{64}(x)$, or $\circ_{80}(x)$, or the double rounding $\circ_{64}(\circ_{80}(x))$.

With $\alpha = 2^{-1022}$, we have either

$$|x| \geq \alpha \text{ and } \left| \frac{x - \square(x)}{x} \right| \leq \beta \text{ and } |\square(x)| \geq \alpha$$

or

$$|x| \leq \alpha \text{ and } |x - \square(x)| \leq \gamma \text{ and } |\square(x)| \leq \alpha.$$ 

with $\beta = 2050 \times 2^{-64} \geq 2^{-53}$

$\gamma = 2049 \times 2^{-1086} \geq 2^{-1075}$

formally proved using the Coq proof assistant
Rounding error in presence of FMA

- 64-bit rounding
- 80-bit rounding
- Double rounding

FMA (Fused Multiply-Add)
Rounding error in presence of FMA

- 64-bit rounding
- 80-bit rounding
- Double rounding
- “Identity” rounding

\[ \Box(x) = x \]
Rounding error in presence of FMA

- 64-bit rounding
- 80-bit rounding
- double rounding
- “identity” rounding

FMA (Fused Multiply-Add)

\[ \circ(x \times y \pm z) \]

\[ \square(x) = x \]
Rounding error in presence of FMA

64-bit rounding

80-bit rounding

double rounding

“identity” rounding

\[ \Box(x) = x \]

FMA (Fused Multiply-Add)

\[ \circ(x \times y \pm z) \]

\[ \Box(\Box(x \times y) \pm z) \]
Rounding error in presence of FMA

- 64-bit rounding
- 80-bit rounding
- Double rounding
- "Identity" rounding

FMA (Fused Multiply-Add)

\[ \diamond (\diamond (x \times y) \pm z) \]

"Identity" rounding

\[ \diamond (x) = x \]
**Rounding of a several operations**

**Theorem 2**

If we define each result of an operation by the formulas of Theorem 1, and if we are able to deduce from these intervals an interval \( \mathcal{I} \) for the final result, then the really computed final result is in \( \mathcal{I} \) whatever the architecture and the compiler that preserves the order of operations.

\[
\begin{align*}
    x_1 &= a \times b \\
    x_2 &= x_1 + c \\
    \vdots \\
    x_n &= x_{n-1} \times d
\end{align*}
\]

\[
\begin{align*}
    &\xRightarrow{\text{Theorem 1}} x_1 \in \mathcal{I}_1(a, b) \\
    &\xRightarrow{\text{Theorem 1}} x_2 \in \mathcal{I}_2(x_1, c) \\
    \vdots \\
    &\xRightarrow{\text{Theorem 1}} x_n \in \mathcal{I}_n(x_{n-1}, d)
\end{align*}
\]

\[\rightarrow x_n \in \mathcal{I}(a, b, c, \ldots, d)\]

**Frama-C/Jessie**

**Gappa**
Rounding of a several operations

Theorem 2

If we define each result of an operation by the formulas of Theorem 1, and if we are able to deduce from these intervals an interval \( I \) for the final result, then the really computed final result is in \( I \) whatever the architecture and the compiler that preserves the order of operations.

\[
\begin{align*}
    x_1 &= a \times b \\
    x_2 &= x_1 + c \\
    \vdots \\
    x_n &= x_{n-1} \times d \\
    \frac{\text{Theorem 1}}{} & \quad x_1 \in I_1(a, b) \\
    \frac{\text{Theorem 1}}{} & \quad x_2 \in I_2(x_1, c) \\
    \vdots \\
    \frac{\text{Theorem 1}}{} & \quad x_n \in I_n(x_{n-1}, d) \\
\end{align*}
\]

\[\rightarrow x_n \in I(a, b, c, \ldots, d)\]

Correct for any compiler and architecture

Frama-C/Jessie

Gappa

Thi Minh Tuyen Nguyen

Formal proofs of numerical programs

23
Reordering

With $|e| \ll |x|$

- $(e + x) - x = 0$
- $e + (x - x) = e$

If the compiler reorders, what can we do?
Addition reordering-independent proofs

Theorem 3

Given a sequence of real numbers \((a_i)_{0 \leq i \leq n} (n \leq \frac{1}{\varepsilon})\) and a real \(l\). We assume that with \(\varepsilon_n = (1 + \varepsilon)^n - 1 \approx n\varepsilon\)

\[|x \oplus y - (x + y)| \leq \varepsilon_n \cdot (|x| + |y|) + n \cdot \eta\]

For an ordering \(o_1\) of the additions, \(S_{n}^{o_1}\) is the summation of \((a_i)\) with the order \(o_1\). If we are able to deduce that \(|S_{n}^{o_1} - \sum_{0}^{n} a_i| \leq l\), then, whatever the ordering \(o_2\) of the additions with the summation \(S_{n}^{o_2}\), we have \(|S_{n}^{o_2} - \sum_{0}^{n} a_i| \leq l\).

- Use \((|x| + |y|)\) instead of \((|x + y|)\)
- \(n\) is unknown
Addition reordering-independent proofs

**Theorem 3**

Given a sequence of real numbers \((a_i)_{0 \leq i \leq n}\) \((n \leq \frac{1}{\varepsilon})\) and a real \(I\). We assume that with \(\varepsilon_n = (1 + \varepsilon)^n - 1 \approx n\varepsilon\)

\[|x \oplus y - (x + y)| \leq \varepsilon_n \cdot (|x| + |y|) + n \cdot \eta\]

For an ordering \(o_1\) of the additions, \(S_{n}^{o_1}\) is the summation of \((a_i)\) with the order \(o_1\). If we are able to deduce that \(|S_{n}^{o_1} - \sum_{0}^{n} a_i| \leq I\), then, whatever the ordering \(o_2\) of the additions with the summation \(S_{n}^{o_2}\), we have \(|S_{n}^{o_2} - \sum_{0}^{n} a_i| \leq I\).

- Use \((|x| + |y|)\) instead of \((|x + y|)\)
- \(n\) is unknown
Addition reordering-independent proofs

- Consider 16 additions/subtractions:
  - $\varepsilon' = 2051 \cdot 2^{-60} \geq \varepsilon_{16}$
  - $\eta' = \eta_{16} = 2049 \times 2^{-1082}$

- Put as postconditions of the addition/subtraction: formulas

\[
| x \oplus y - (x + y) | \leq \varepsilon' \cdot (|x| + |y|) + \eta' \\
| x \ominus y - (x - y) | \leq \varepsilon' \cdot (|x| + |y|) + \eta'
\]
Hardware and addition reordering-independent proofs

**Theorem 1**

Let \( \varepsilon = 2050 \times 2^{-64} \)
Let \( \eta = 2049 \times 2^{-1086} \)

If we define each operation result as any real such that

\[
|x \oplus y - (x + y)| \leq \varepsilon \cdot (|x + y|) + \eta \\
|x \ominus y - (x - y)| \leq \varepsilon \cdot (|x - y|) + \eta \\
|x \otimes y - (x \ast y)| \leq \varepsilon \cdot |x \ast y| + \eta \\
|x \oslash y - (x/y)| \leq \varepsilon \cdot |x/y| + \eta \\
|\circ(\sqrt{x}) - \sqrt{x}| \leq \varepsilon \cdot |\sqrt{x}| + \eta
\]

and if we are able to deduce an interval \( \mathcal{I} \) for the final result then the really computed final result is in \( \mathcal{I} \) whatever

- the architecture and the compiler
Hardware and addition reordering-independent proofs

**Theorem 4**

Let \( \varepsilon = 2050 \times 2^{-64} \) and \( \varepsilon' = 2051 \times 2^{-60} \)

Let \( \eta = 2049 \times 2^{-1086} \) and \( \eta' = 2049 \times 2^{-1082} \)

If we define each operation result as any real such that

\[
|x \oplus y - (x + y)| \leq \varepsilon' \cdot (|x| + |y|) + \eta'
\]

\[
|x \ominus y - (x - y)| \leq \varepsilon' \cdot (|x| + |y|) + \eta'
\]

\[
|x \otimes y - (x \ast y)| \leq \varepsilon \cdot |x \ast y| + \eta
\]

\[
|x \oslash y - (x/y)| \leq \varepsilon \cdot |x/y| + \eta
\]

\[
|\circ(\sqrt{x}) - \sqrt{x}| \leq \varepsilon \cdot |\sqrt{x}| + \eta
\]

and if we are able to deduce an interval \( \mathcal{I} \) for the final result then the really computed final result is in \( \mathcal{I} \) whatever

- the architecture and the compiler
- the compiler reorganization with (maximum 16) additions/subtractions
#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 \abs(sx) <= 100.0 && \abs(sy) <= 100.0 && 
@ \abs(vx) <= 1.0 && \abs(vy) <= 1.0;
@ ensures \result != 0
@ ==> \result == l_sign(sx*vx+sy*vy)*l_sign(sx*vy−sy*vx);*/
int eps_line(double sx, double sy, double vx, double vy) {
    int s1=sign(sx*vx+sy*vy, −E, E);
    int s2=sign(sx*vy−sy*vx, −E, E);
    return s1*s2;
}
KB3D example

#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 || \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 \abs(sx) <= 100.0 &&
     @     \abs(vx) <= 1.0 && \abs(vy) <= 1.0;
@ ensures \result != 0
@    ==> \result == l_sign(sx*vx+sy*vy)*l_sign(sx*vy-sy*vx);*/
int eps_line(double sx, double sy, double vx, double vy) {
   int s1 = sign(sx*vx+sy*vy, -E, E);
   int s2 = sign(sx*vy-sy*vx, -E, E);
   return s1*s2;
}
KB3D example

```c
#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 abs(sx) <= 100.0 &&
@ abs(vx) <= 1.0 &&
@ ensures result != 0
@ ==> result == l_sign(sx*vx+sy*vy) */

int eps_line(double sx, double sy) {
    int s1 = sign(sx*vx+sy*vy, -E, E);
    int s2 = sign(sx*vy-sy*vx, -E, E);
    return s1*s2;
}
```

Thi Minh Tuyen Nguyen
Formal proofs of numerical programs 28
KB3D example

```c
#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 \abs(sx) <= 100.0 &&
@ \abs(vx) <= 1.0 &&
@ ensures \result != 0
@ ==> \result == l_sign(sx*vx+s y*vy)
int eps_line(double sx, double sy)
int s1=sign(sx*vx+sy*vy, -E, E);
int s2=sign(sx*vy-sy*vx, -E, E);
return s1*s2;
```
# KB3D example

```c
#define E 0x1p-45
//@ logic integer l_sign(real x)

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

**Default behavior**

- check FP overflow

**Safety**

- check FP overflow

**Precondition for user call**

- precondition for user call

**Function sign**

**Default behavior**

- postcondition

```c
int eps_line(double sx, double sy, double vx, double vy)
{
    int s1, s2;

    s1 = sign(sx * vx + sy * vy, -0x1.9a0641p-45, 0x1.9a0641p-45);
    s2 = sign(sx * vy - sy * vx, -0x1.9a0641p-45, 0x1.9a0641p-45);

    return s1 * s2;
}
```
Strict IEEE-754: $E = 0x1p^{-45}$

```c
/*@ requires e1 <= x - \exact(x) <= e2; @*/
int sign(double x, double e1, double e2)
```

Arch-independent model: $E = 0x1.90641p^{-45}$

```c
int eps_line(double sx, double sy, double vx, double vy){
    int s1, s2;
    s1 = sign(sx * vx + sy * vy, -0x1.9a0641p-45, 0x1.9a0641p-45);
    s2 = sign(sx * vy - sy * vx, -0x1.9a0641p-45, 0x1.9a0641p-45);
    return s1 * s2;
}
```
Strict IEEE-754: $E = 0x1p-45$

Arch-independent model: $E = 0x1.90641p-45$

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

```c
/*@ requires e1 <= x - exact(x) <= e2; @*/
```
Strict IEEE-754: $E = 0x1p-45$

```c
/*@ requires e1 <= x - \text{exact}(x) <= e2;
@*/
int sign(double x, double e1, double e2)

Arch-independent model: $E = 0x1.90641p-45$

```
Outline

1. Introduction
2. Preliminaries
3. Hardware-independent proofs
4. Hardware-dependent proofs
5. Conclusions and Future works
Back to the double rounding example

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

    //@ assert z == 1.0 + 0x1p-52;
}
```

- Strict IEEE-754 model: proves
  ```c
  //@ assert z == 1.0 + 0x1p-52;
  ```
  not true for some architecture/compiler

- Hardware-independent approach: proves
  ```c
  //@ assert 1.0 <= z <= 1.0 + 0x1p-52;
  ```
  an interval, not an exact value
Goals

Prove floating-point programs

- without assuming strict IEEE-754
- while taking compiler settings and architecture into account
Goals

Prove floating-point programs
- without assuming strict IEEE-754
- while taking compiler settings and architecture into account

How?

By analyzing assembly code, we know
- the precision of each operation
- the use of FMA
- etc.
Assembly code of double rounding example

```c
int main()
{
    double x = 1.0;
    double y = 0x1p−53 + 0x1p−64;
    double z = x + y;

    //@ assert z == ?;
}
```

```
movabsq $4607182418800017408, %rax
movq %rax, −8(%rbp)
movabsq $4368493837572636672, %rax
movq %rax, −16(%rbp)
movsd −8(%rbp), %xmm0
addsd −16(%rbp), %xmm0
movsd %xmm0, −24(%rbp)
```

Assembly code of double rounding example

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

    // @ assert z == ?;
}
```

```
 gcc -S
```

```
movabsq $4607182418800017408, %rax
movq %rax, -8(%rbp)
movabsq $436849383757263672, %rax
movq %rax, -16(%rbp)
movsd -8(%rbp), %xmm0
addsd -16(%rbp), %xmm0
movsd %xmm0, -24(%rbp)
```

Thi Minh Tuyen Nguyen  Formal proofs of numerical programs 33
Assembly code of double rounding example

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

```
movabsq $4607182418800017408, %rax
movq %rax, -8(%rbp)
movabsq $436849383757263672, %rax
movq %rax, -16(%rbp)
movsd -8(%rbp), %xmm0
addsd -16(%rbp), %xmm0
movsd %xmm0, -24(%rbp)
```

```
gcc -S
```

```
decode_float64(4607182418800017408) = 1.0
x
```

Thi Minh Tuyen Nguyen

Formal proofs of numerical programs

33
Assembly code of double rounding example

```c
int main() {
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64;
    double z;
}
```

```assembly
movabsq $4607182418800017408, %rax
movq %rax, -8(%rbp)
movabsq $4368493837572636672, %rax
movq %rax, -16(%rbp)
movsd -8(%rbp), %xmm0
addsd -16(%rbp), %xmm0
movsd %xmm0, -24(%rbp)
```

`gcc -S`
Assembly code of double rounding example

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

```assembly
movabsq $4607182418800017408, %rax
movq %rax, −8(%rbp)
movabsq $4368493837572636672, %rax
movq %rax, −16(%rbp)
movsd −8(%rbp), %xmm0
addsd −16(%rbp), %xmm0
movsd %xmm0, −24(%rbp)
```

```
decode_float64(4607182418800017408) = 1.0

decode_float64(4368493837572636672) = 2^{-53} + 2^{-64}
```
Assembly code of double rounding example

```c
int main()
{
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64;
    double z = x + y;
    decode_float64(4607182418800017408) = 1.0
    decode_float64(4368493837572636672) = 2^{-53} + 2^{-64}
}
```

```
    gcc -S decode [-0x-0] float64(4607182418800017408) = 1.0
    gcc -mfpmath=387 -S ◦80(x + y) ◦64(x + y)
```

```
    movabsq $4607182418800017408, %rax
    movq %rax, −8(%rbp)
    movabsq $4368493837572636672, %rax
    movq %rax, −16(%rbp)
    movsd −8(%rbp), %xmm0
    addsd −16(%rbp), %xmm0
    movsd %xmm0, −24(%rbp)
```
Assembly code of double rounding example

```c
int main() {
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64;
    double z = x + y;
}
```

```
/gcc -mfpmath=387 -S
```

```
decode_float64(4607182418800017408) = 1.0
```

```
decode_float64(4368493837572636672) = 2^{-53} + 2^{-64}
```

```
movabsq $4607182418800017408, %rax
movq %rax, -8(%rbp)
movabsq $4368493837572636672, %rax
movq %rax, -16(%rbp)
movsd -8(%rbp), %xmm0
fildl -8(%rbp)
addsd -16(%rbp), %xmm0
faddl -16(%rbp)
movsd %xmm0, -24(%rbp)
fstpl -24(%rbp)
```
Assembly code of double rounding example

```c
int main() {
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64;
    double z = x + y;
}
```

```
 gcc -mfpmath=387 -S

 decode_float64(4607182418800017408) = 1.0
 decode_float64(4368493837572636672) = 2^{-53} + 2^{-64}
```

Instructions:
- `movabsq $4607182418800017408, %rax`
- `movq %rax, -8(%rbp)`
- `movabsq $4368493837572636672, %rax`
- `movq %rax, -16(%rbp)`
- `movsd -8(%rbp), %xmm0`
- `addsd -16(%rbp), %xmm0`
- `movsd %xmm0, -24(%rbp)`
- `fldl -8(%rbp)`
- `faddl -16(%rbp)`
- `fstopl -24(%rbp)`
Assembly code of double rounding example

```c
int main() {
    double x = 1.0;
    double y = 0x1p-53 + 0x1p-64;
    double z = x + y;
}
```

```
gcc -mfpmath=387 -S
```

```assembly
movabsq $4607182418800017408, %rax
movq %rax, -8(%rbp)
movabsq $4368493837572636672, %rax
movq %rax, -16(%rbp)
movsd 8(%rbp), %xmm0
addsd -16(%rbp), %xmm0
movsd %xmm0, 24(%rbp)
```

- `decode_float64(4607182418800017408) = 1.0`
- `decode_float64(4368493837572636672) = 2^{-53} + 2^{-64}`
C program + annotations

preparation

C program + inline assembly

gcc -S

assembly code + annotations

GNU Assembler modified

proof obligations in Why

automatic/interactive provers
Operands

- `movq $4607182418800017408, %rax`
- `movq %rdx, -4(%rbp)`

An operand is either

- an immediate constant (a decimal integer)
- a register: `%eax`, `%rdx`, etc.
- a memory reference
Operands

\begin{verbatim}
movq $4607182418800017408, %rax
movq %rdx, -4(%rbp)
\end{verbatim}

An operand is either

- an immediate constant (a decimal integer)
- a register: \%eax, \%rdx, etc.
- a memory reference

Operands are not typed: may denote an integer, a float, an address, etc.
Translation of operands

**Type register**

CPU registers/memory references are represented in Why by references (aka mutable variables) of type register

```plaintext
logic sel_int32  : register -> int32
logic sel_int64  : register -> int64
logic sel_single : register -> single
logic sel_double : register -> double
```
General-purpose instructions

\[
\text{parameter} \ \text{move\_gte64}: \ a : \text{int} \to b : \text{real} \\
\to r : \text{register refl} \\
\{ \}
\text{unit writes} \ r \\
\{ \ \text{integer\_of\_int64}(\text{sel\_int64}(r)) = a \ \text{and} \\
\text{double\_value}(\text{sel\_double}(r)) = b \ \}
\]

\[
\begin{align*}
[\text{movq imm, reg}]_i & = \text{move\_gte64} \ [\text{imm}]_{\text{int64}} \ [\text{imm}]_{\text{double}} \ \text{reg} \\
[\text{movq src, reg}]_i & = \text{move\_gte64} \ [\text{src}]_{\text{int64}} \ [\text{src}]_{\text{double}} \ \text{reg}
\end{align*}
\]

\[
\begin{align*}
[\text{imm}]_{\text{int64}} & = \text{imm} \\
[\text{imm}]_{\text{double}} & = \text{decode\_float64}(\text{imm}) \\
[\text{reg}]_{\text{int64}} & = \text{of\_int64}(\text{sel\_int64}(!\text{reg})) \\
[\text{reg}]_{\text{double}} & = \text{double\_value}(\text{sel\_double}(!\text{reg}))
\end{align*}
\]
Copy examples

- Immediate to register
  
  ```
  movq  $4607182418800017408, %rax
  ```
  
  Why:
  
  ```
  move_cste64 4607182418800017408 1.0 _rax
  ```
  
  decode_float64(4607182418800017408) = 1.0

- Register to register
  
  ```
  movq  %rdx, %rax
  ```
  
  Why:
  
  ```
  move_cste64 (of_int64(select64(!rdx)))
  (double_value(select_double(!rdx))) _rax
  ```
Translation of SSE2 instructions

\[
\text{parameter } \text{set\_double} : a : \text{real} \rightarrow b : \text{register } \text{ref} \rightarrow \\
\{ | o_{64}(a) | \leq \text{max\_double} \} \\
\text{unit writes } b \\
\{ \text{double\_value}(\text{sel\_double}(b)) = o_{64}(a) \}
\]

\[
[\text{addsd src, dest}]_i = \text{set\_double} ([\text{dest}]_{\text{double}} + [\text{src}]_{\text{double}}) \text{dest}
\]

- similar for \text{sub} and \text{mul}
- For \text{div}: checks division by 0 in the precondition

**SSE2: Streaming SIMD Extensions 2**
Translation of FMA instructions

\[ \odot(x \times y + z) \]

\[[vfmaddsd \ src3, \ src2, \ src1, \ dest]; =
set\_double\ (\[src1\]_{\text{double}} \times \[src2\]_{\text{double}} + \[src3\]_{\text{double}}) \ dest\]
Translation of x87 Floating-point Unit (FPU): stack issue

**Assembly**

- Stack of eight 80-bit registers: `%st0` to `%st7`
- Top-of-stack pointer TOS: the current position in the stack
- Index relative to TOS
- Example:
  \[
  \text{fadd } %st(1), %st(0)
  \]
  Adds to the register at top the value of the register below the top

**Why**

- 8 variables of type register ref
- Value of TOS at each instruction is computed statically by
  \[
  %st(i) \text{ interpreted as } _{\text{st}}i
  \]
  where \( i = TOS + i \)
- \( 0 \leq TOS < 8 \)
## KB3D example

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Optim. level</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE2</td>
<td>-00</td>
<td>$2048 \times 2^{-56}$</td>
</tr>
<tr>
<td>x87</td>
<td>-00</td>
<td>$1025 \times 2^{-56}$</td>
</tr>
<tr>
<td>x87</td>
<td>-02</td>
<td>$1025 \times 2^{-56}$</td>
</tr>
<tr>
<td>FMA</td>
<td>-02</td>
<td>$1536 \times 2^{-56}$</td>
</tr>
<tr>
<td>strict IEEE-754</td>
<td></td>
<td>$2048 \times 2^{-56}$</td>
</tr>
<tr>
<td>any arch.</td>
<td></td>
<td>$3203 \times 2^{-56}$</td>
</tr>
</tbody>
</table>
Other assembly instructions

Also supported

- tests and jumps (if, while, etc.)
- function calls
- indirect memory access (arrays, pointers)
Other assembly instructions

Also supported

- tests and jumps (if, while, etc.)
- function calls
- indirect memory access (arrays, pointers)

How?

- Translation of any control flow graph into Why inspired by [Barnett & Leino, 2005] and [Filliâtre, 2007]
- Use a memory model with a few separation hypothesis
Scalar product: With loop and pointers

#define NMAX 10
#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(result)) <= n * B; */
double scalar_product(double x[], double y[], int n) {
  double p = 0.0;

  for (int i=0; i < n; i++) {
    L:
    p = p + x[i]*y[i];
    /*@ assert abs(p - (at(p,L) + x[i]*y[i])) <= B; */
  }
  return p;
}
Scalar product: With loop and pointers

```c
#define NMAX 10
#define B 0x1.1p−50

/*@ requires 0 < n <= NMAX; @ requires \forall \text{integer } i; 0 <= i < n \implies \text{range}(y,0,n−1) @ requires \forall i; \abs(x[i]) <= 1.0 && \abs(y[i]) <= 1.0 ; @ ensures \abs(\text{result} - \text{exact(\text{result})}) <= n * B; */

double scalar_product(double x[], double y[], int n) {
    double p = 0.0;

    for (int i=0; i < n; i++) {
        L:
        p = p + x[i] * y[i];
        /*@ assert \abs(p - (\text{at}(p,L) + x[i]*y[i])) <= B; */
    }
    return p;
}
```

Two vectors \( x \) and \( y \)

- have \( n \) elements
- are represented as arrays of doubles

Two vectors \( x \) and \( y \) have \( n \) elements and are represented as arrays of doubles. The scalar product of two vectors is computed by:

\[
\sum_{0 \leq i < n} x_i y_i
\]

The value of \( p \) might be calculated by either:

- following strictly the IEEE-754 standard
- using x87 with 80-bit internal registers
- using x87 with optimization (\( p \) is stored in 80-bit register)
- using the FMA instructions.

The rounding error \( B \) at each step depends on the options of compiler/architecture. The value of \( B \) depends on the value of \( NMAX \).
Scalar product: With loop and pointers

```c
#define NMAX 10
#define B 0x1.1p-50

double scalar_product(double x[], double y[], int n) {
    double p = 0.0;
    for (int i = 0; i < n; i++) {
        p = p + x[i] * y[i];
        /*@ assert \abs(p - (\at(p,L) + x[i]*y[i])) <= B; */
    }
    return p;
}
```

The scalar product of two vectors is computed by:

\[ \sum_{0 \leq i < n} x_i y_i \]

Two vectors \( x \) and \( y \) have \( n \) elements are represented as arrays of doubles. The scalar product of two vectors is computed by:

\[ \sum_{0 \leq i < n} x_i y_i \]

The value of \( p \) might be calculated by either:

- Following strictly the IEEE-754 standard
- Using x87 with 80-bit internal registers
- Using the FMA instructions.

The rounding error \( B \) at each step depends on the options of compiler/architecture. The value of \( B \) depends on the value of \( NMAX \).
Scalar product: With loop and pointers

```c
#include <stdio.h>

int main() {
    double x[10], y[10], p = 0.0;
    for (int i = 0; i < 10; i++) {
        x[i] = i;  // Example values
        y[i] = i;  // Example values
    }
    p = scalar_product(x, y, 10);
    printf("%f\n", p);
    return 0;
}
```

The value of $p$ might be calculated by either
- following strictly the IEEE-754 standard
- or using x87 with 80-bit internal registers
- or using x87 with optimization (p is stored in 80-bit register)
- or using the FMA instructions.
Scalar product: With loop and pointers

```c
#define NMAX 10
#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 double scalar_product(x, y, n) { double p = 0.0;
     for (int i=0; i < n; i++) {
       L:
       p = p + x[i]*y[i];
       /* @ assert abs(p - (at(p,L) + x[i]*y[i])) <= B; */
     }
     return p;
   }
```

Two vectors \( x \) and \( y \) have \( n \) elements are represented as arrays of doubles. The scalar product of two vectors is computed by:

\[
\sum_{0 \leq i < n} x[i] y[i]
\]

The value of \( p \) might be calculated by either:
- Following strictly the IEEE-754 standard
- Using x87 with 80-bit internal registers
- Using x87 with optimization (\( p \) is stored in 80-bit register)
- Using the FMA instructions.

The rounding error \( B \) at each step: depends on the options of compiler/architecture.

The value of \( B \) depends on the value of \( NMAX \).
Scalar product: With loop and pointers

```c
#define NMAX 10
#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(result)) <= n * B; */
double scalar_product(double x[], double y[], int n) {
    double p = 0.0;

    for (int i=0; i < n; i++) {
        p = p + x[i]*y[i];
        /*@ assert abs(p - (at(p,L) + x[i]*y[i])) <= B; */
    }
    return p;
}
```

Two vectors $x$ and $y$ have $n$ elements are represented as arrays of doubles. The scalar product of two vectors is computed by:

$$\sum_{0 \leq i < n} x[i] y[i]$$

The value of $p$ might be calculated by either following strictly the IEEE-754 standard or using x87 with 80-bit internal registers or using x87 with optimization ($p$ is stored in 80-bit register) or using the FMA instructions. The rounding error $B$ at each step: depends on the options of compiler/architecture. The value of $B$ depends on the value of $NMAX$. 

Thi Minh Tuyen Nguyen
Formal proofs of numerical programs 44
Scalar product example: The value of rounding error $B$

<table>
<thead>
<tr>
<th>Arch.</th>
<th>NMAX</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>SSE2 -O0</td>
<td>0x1.1p-50</td>
<td>0x1.02p-47</td>
<td>0x1.004p-44</td>
<td></td>
</tr>
<tr>
<td>x87 -O0</td>
<td>0x1.0022p-50</td>
<td>0x1.0021p-47</td>
<td>0x1.00201p-44</td>
<td></td>
</tr>
<tr>
<td>x87 -O2</td>
<td>0x1.1p-61</td>
<td>0x1.02p-58</td>
<td>0x1.004p-55</td>
<td></td>
</tr>
<tr>
<td>FMA -O2</td>
<td>0x1p-50</td>
<td>0x1p-47</td>
<td>0x1p-44</td>
<td></td>
</tr>
<tr>
<td>strict IEEE-754</td>
<td>0x1.1p-50</td>
<td>0x1.02p-47</td>
<td>0x1.004p-44</td>
<td></td>
</tr>
<tr>
<td>any arch.</td>
<td>0x1.629p-46</td>
<td>0x1.94ep-43</td>
<td>0x1.f55p-40</td>
<td></td>
</tr>
</tbody>
</table>
Outline

1. Introduction
2. Preliminaries
3. Hardware-independent proofs
4. Hardware-dependent proofs
5. Conclusions and Future works
Conclusions

Two approaches:

- **Hardware-independent proofs**
  - Gives correct rounding errors whatever the architecture and the choices of the compiler
  - **Drawback:**
    - Incomplete: only proves rounding errors
    - Get worse rounding errors than in practice

- **Hardware-dependent proofs**
  - Analyze assembly code
  - **Translator:** $\approx 10K$ lines of C code
  - Show that proving complex behavior of C floating-point programs on their assembly code is possible.
Future works

Hardware-independent proofs:
- Look into multiplication reordering and other optimizations

Hardware-dependent proofs:
- Ongoing work: bit-level reasoning
  - Some experiences in my memoir
  - Complicates proof a lot
- Longer term perspectives:
  - Integrate my approach into a certified compiler [Herms et al., 2012]
    to reduce the trusted code base
Thank you for your attention!
Thank you for your attention!
Translation of a function in assembly into Why

f:
```assembly
.cfi_startproc
    mov reg, mem_ref
    # move input values of the function from register to memory_reference
    /*@ requires P;*/
    (body of the function f)
    //@ assert A;
    mov mem_ref, reg
    # move output value of the function from memory_reference to register
    /*@ ensures Q;*/
    leave
    ret
.cfi_endproc
```

let f() =
```plaintext
[move reg, mem_ref];

[{}] unit reads V {[[P]_annot]};

[(body of the function f)];
assert {[[A]_annot]};

[move reg, mem_ref];

assert {[[Q]_annot]};

void
parameter f: unit ->
{{[P]_annot} unit writes w {[[Q]_annot]}

Thi Minh Tuyen Nguyen
Formal proofs of numerical programs
**x87 instructions**

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

[flsl src]; = set_80 [src]_{double} st0
[fstl dest]; = set_double [st0]_{binary80} dest
[faddl src]; = set_80 ([st0]_{binary80}+[src]_{double}) st0

where set_80 is similar to set_double for 80 bits
```
Memory model in Why

```plaintext
type 'v memory
logic select: 'v memory, int → 'v
```

Memory consists of registers:

```plaintext
parameter MEM: register memory ref
```
Translation of memory reference

\[
\begin{align*}
[\text{mem}]_{\text{int32}} &= \text{of}_{\text{int32}}(\text{select}(\text{MEM}, [\text{mem}]_{\text{addr}})) \\
[\text{mem}]_{\text{int64}} &= \text{of}_{\text{int64}}(\text{select}(\text{MEM}, [\text{mem}]_{\text{addr}})) \\
[\text{mem}]_{\text{single}} &= \text{single}_{\text{value}}(\text{select}(\text{MEM}, [\text{mem}]_{\text{addr}})) \\
[\text{mem}]_{\text{double}} &= \text{double}_{\text{value}}(\text{select}(\text{MEM}, [\text{mem}]_{\text{addr}})) \\
[\text{mem}]_{\text{binary80}} &= \text{binary80}_{\text{value}}(\text{select}(\text{MEM}, [\text{mem}]_{\text{addr}})) \\
[\text{mem}]_{\text{exact}} &= \text{select}(\text{MEM}, [\text{mem}]_{\text{addr}})
\end{align*}
\]

A memory reference \text{mem} having the general form:
\[
\text{disp}(\text{base}, \text{index}, \text{scale})
\]
is interpreted as
\[
[\text{mem}]_{\text{addr}} = [\text{base}]_{\text{int64}} + \text{disp} + \text{scale} \times [\text{index}]_{\text{int64}}
\]
When the memory is updated

\[
\text{predicate unchanged}\_\text{mem}(\text{MEM1: register memory}, \\
\text{MEM2: register memory}, \text{addr: int}, \text{nb: int}) = \\
( \text{for all } i: \text{int. } i <= \text{addr}-8 \text{ or } i >= \text{addr+nb} \rightarrow \\
\text{select(\text{MEM1, i}) = select(\text{MEM2, i})}) \\
\text{and} \\
( \text{for all } i: \text{int. } i <= \text{addr}-4 \rightarrow \\
\text{of\_int32(\text{sel\_int32(select(\text{MEM1, i}))}) =} \\
\text{of\_int32(\text{sel\_int32(select(\text{MEM2, i}))})} \\
\text{and} \\
\text{single\_value(\text{sel\_single(select(\text{MEM1, i}))}) =} \\
\text{single\_value(\text{sel\_single(select(\text{MEM2, i}))})}) 
\]
Translation of instructions

```plaintext
parameter move_reg_to_mem64: a:register -> b:int->
{  }
unit writes MEM
{
  of_int64(select(MEM, b)) =
  of_int64(select(MEM, a))
  and
  double_value(select(select(MEM, b))) =
  double_value(select(a))
  and
  unchanged_mem(MEM, MEM@, b, 8) }

[ movq reg, mem ]; = move_reg_to_mem64 [ reg ]_{src} [ mem ]_{addr}
```
Worst case - Algorithm

if \( c_0 \) then \( t_0 \) else \( e_0 \);
if \( c_1 \) then \( t_1 \) else \( e_1 \);
if \( c_2 \) then \( t_2 \) else \( e_2 \);
...
if \( c_k \) then \( t_k \) else \( e_k \);
The number of paths is $2^k$.

If we insert invariants at each node: the number of paths is $2^k$. 
Select and store in memory model

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

Relation between “store” and “select”:

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)

- Make the proofs slow
- Gappa are not able to use directly the axioms
Only “select” is used

parameter move_reg_to_mem64: a:register -> b:int->
{ }

  unit writes MEM
{ integer_of_int64(select(MEM, b))
    = integer_of_int64(select(MEM, b))

  and
  double_value(select(MEM, b))
    = double_value(select(MEM, b))

  and
  sel_exact(select(MEM, b))=sel_exact(a)
  and
  unchanged_mem(MEM, MEM@, b, 8) }

If we use “store”:

MEM = store(MEM@,b,a)

Use directly “select”:

- Specify in the post-condition of the parameter all the properties that we need: what is changed, what is not changed
Use of Why in this thesis

ACSL-annotated C program

Frama-C

multirounding model

Jessie plug-in

Why platform

Why VC Generator

Automatic/interactive provers

inline assembly translator

Assembly code

modified version of GNU Assembler