Hardware-independent proofs of numerical programs

Sylvie Boldo and Thi Minh Tuyen Nguyen

INRIA Saclay, France

April 14th, 2010
A first example

```c
void sign(double x){
    if (x > 0.0) printf(" Positive");
    else if (x < 0.0) printf(" Negative");
    else printf(" Zero");
}

void main(){
    double a = 0x1p-53 + 0x1p-64;  // a = 2^{-53} + 2^{-64}
    double b = 1.0 + a - 1.0;
    sign(b - a);
}
```

A first example

```c
void sign(double x) {
    if (x > 0.0) printf(" Positive");
    else if (x < 0.0) printf(" Negative");
    else printf(" Zero");
}
void main() {
    double a = 0x1p-53 + 0x1p-64; // a = 2^{-53} + 2^{-64}
    double b = 1.0 + a - 1.0;
    sign(b - a);
}
```

```
gcc test.c
```

Positive

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

Negative

Sylvie Boldo and Thi Minh Tuyen Nguyen
A first example

```c
void sign(double x){
    if (x > 0.0) printf(" Positive");
    else if (x < 0.0) printf(" Negative");
    else printf(" Zero");
}
void main(){
    double a = 0x1p-53 + 0x1p-64; // a = 2^{-53} + 2^{-64}
    double b = 1.0 + a - 1.0;
    sign(b - a);
}
```

Positive

gcc test.c

Sylvie Boldo and Thi Minh Tuyen Nguyen
A first example

```c
void sign(double x){
    if (x > 0.0) printf("Positive");
    else if (x < 0.0) printf("Negative");
    else printf("Zero");
}
void main(){
    double a = 0x1p-53 + 0x1p-64; // a = 2^{-53} + 2^{-64}
    double b = 1.0 + a - 1.0;
    sign(b - a);
}
```

gcc -mfpmath=387 test.c
A first example

```c
void sign(double x) {
    if (x > 0.0) printf("Positive");
    else if (x < 0.0) printf("Negative");
    else printf("Zero");
}

void main() {
    double a = 0x1p-53 + 0x1p-64; // a = 2^{-53} + 2^{-64}
    double b = 1.0 + a - 1.0;
    sign(b - a);
}
```

Negative

```bash
gcc -mfpmath=387 test.c
```
Architecture and rounding precision

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

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

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

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

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

- All current processors support IEEE-754
  - A floating-point arithmetic standard
- Some architecture-depend issues:
  - x87 floating-point unit uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - FMA(fused multiply-add) instruction supported by the PowerPC and the Intel Itaniumum architecture
    - calculates \((x \times y \pm z)\) with a single rounding

\[ \implies \text{introduce subtle inconsistencies between program executions} \]
Analyzing numerical program

Tools for analyzing numerical programs

- Abstract interpretation:
  - Fluctuat, Astrée, etc.

- Frama-C:
  - A framework for static analysis of C code
  - Flexible: Easy to add a new plug-in
    - Value analysis: plug-in based on abstract interpretation
    - Jessie: deductive verification

- In Jessie: Easy to change the interpretation of floating-point operation
Frama-C and floating-point arithmetic

Annotated C program

Frama-C/Jessie plug-in

WHY verification condition generator

Verification conditions

Automatic provers (Alt-Ergo, Gappa, CVC3, etc.)

ACSL specification language
Frama-C and floating-point arithmetic

- Formal verification of FP Programs
  Boldo and Filliâtre(2007)

- Frama-C/Jessie plug-in

- WHY verification condition generator

- Behavioral Properties of FP Programs
  Ayad and Marché(2009)

- Automatic provers
  (Alt-Ergo,Gappa,CVC3,etc.)

ACSL specification language
Frama-C and floating-point arithmetic

Formal verification of FP Programs
Boldo and Filliâtre (2007)

only follow the strict IEEE-754

Frama-C/Jessie plug-in

WHY verification condition generator

Behavioral Properties of FP Programs
Ayad and Marché (2009)

Automatic provers
(Alt-Ergo, Gappa, CVC3, etc.)

ACSL specification language
Formal verification of FP Programs
Boldo and Filliâtre(2007)

only follow the strict IEEE-754

Frama-C/Jessie plug-in

WHY verification condition generator

Behavioral Properties of FP Programs
Ayad and Marché(2009)

Automatic provers
(Alt-Ergo,Gappa,CVC3,etc.)
Behavioral Properties of FP Programs
Ayad and Marché (2009)

Automatic proof of FP Programs
(Alt-Ergo, Gappa, CVC3, etc.)

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

Floating-point arithmetic

Formal verification of FP Programs
Boldo and Filliâtre (2007)

Frama-C and floating-point arithmetic

Verification conditions
WHY verification conditions

Automatic provers
(Alt-Ergo, Gappa, CVC3, etc.)

FMA
double rounding
80-bit rounding
64-bit rounding

Behavioral Properties of FP Programs
Ayad and Marché (2009)

Floating-point arithmetic

Behavioral Properties of FP Programs
Ayad and Marché (2009)

Formal verification of FP Programs
Boldo and Filliâtre (2007)

Frama-C and floating-point arithmetic
Outline

1. Floating-point arithmetic
2. Floating-point computations independent to hardwares and compilers
3. A case study
4. Conclusion and future work
Outline

1. Floating-point arithmetic

2. Floating-point computations independent to hardwares and compilers

3. A case study

4. Conclusion and future work
Floating-point number

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

\[
0 \quad 2^{e_{\text{min}}} \quad (2 - 2^{1-p}) \times 2^{e_{\text{max}}} \quad +\infty
\]

- \( |x| \geq 2^{e_{\text{min}}} \)
- \( x \) is normal number
Floating-point number

### 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

- $2^{e_{\text{min}}}$
- $(2 - 2^{1-p}) \times 2^{e_{\text{max}}}$
- $0 \leq |x| < 2^{e_{\text{min}}}$
- $x$ is subnormal number
Rounding error

Absolute error vs. relative error

- **Absolute error**
  \[ \epsilon(x) = |x - \circ(x)| \]

- **Relative error**
  \[ \epsilon(x) = \left| \frac{x - \circ(x)}{x} \right| \]

\( \circ(x) \) is the rounding value of \( x \)
Rounding error

\[ a \quad x \quad o(x) \quad b \]

Absolute error vs. relative error

- Absolute error
  \[ \epsilon(x) = |x - o(x)| \]

- Relative error
  \[ \epsilon(x) = \left| \frac{x - o(x)}{x} \right| \]

\( o(x) \) is the rounding value of \( x \)
Rounding error

Absolute error vs. relative error

- **Absolute error**
  \[ \epsilon(x) = |x - \omega(x)| \]

- **Relative error**
  \[ \epsilon(x) = \left| \frac{x - \omega(x)}{x} \right| \]

\(\omega(x)\) is the rounding value of \(x\)
Rounding error

Rounding error in normal range
Use relative error

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

Rounding error in subnormal range
Use absolute error

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

**Using round-to-nearest mode**
Rounding error

IEEE-754 double precision (64-bit rounding)

- precision \( p = 53 \)
- \( e_{\text{min}} = -1022 \) and \( e_{\text{max}} = 1023 \)

**Rounding error in normal range**

Use relative error

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

**Rounding error in subnormal range**

Use absolute error

\[
|x - \circ(x)| \leq \alpha \quad (\text{with } \alpha = 2^{-1075})
\]

**Using round-to-nearest mode**
Rounding error

\( x87 \) (80-bit rounding)
- precision \( p = 64 \)
- \( e_{\text{min}} = -16382 \) and \( e_{\text{max}} = 16383 \)

Rounding error in normal range

Use relative error
\[
\left| \frac{x - \circ(x)}{x} \right| \leq 2^{-64}
\]

Rounding error in subnormal range

Use absolute error
\[
|x - \circ(x)| \leq \beta \quad (\text{with } \beta = 2^{-16446})
\]

**Using round-to-nearest mode**
Double rounding

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

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

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

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

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

\[ \text{z} = 1.0 + 2^{-52} \]

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

\[ 0_{64}(x + y) \]

\[ 1 + 2^{-52} \]

**gcc double_rounding.c**
Double rounding

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

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

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

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


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

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

```bash
gcc double_rounding.c
```

Double rounding

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

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

Stored in 80-bit register

Sylvie Boldo and Thi Minh Tuyen Nguyen
Double rounding

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

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

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

```latex
y = 2^{-53} + 2^{-64}
```

Sylvie Boldo and Thi Minh Tuyen Nguyen

Floating-point arithmetic Hardware-independent proofs A case study Conclusion and future work
Rounding error in double rounding

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

\[ |x| \geq \alpha \Rightarrow \left| \frac{x - o_{64}(o_{80}(x))}{x} \right| \leq \beta \]
\[ |x| \leq \alpha \Rightarrow |x - o_{64}(o_{80}(x))| \leq \gamma \]

with $\beta = 2050 \times 2^{-64}$
\[ \gamma = 2049 \times 2^{-1086} \]
Outline

1. Floating-point arithmetic

2. Floating-point computations independent to hardwares and compilers

3. A case study

4. Conclusion and future work
C expression

\[ a = x \times y + z \]
C expression

\[ a = x \times y + z \]

is interpreted as

\[ \square(x \times y) \]

where \( \square \) is one of the following rounding:

- 64-bit rounding
- 80-bit rounding
- double rounding
C expression

\[ a = \overline{x \times y} + z \]

is interpreted as

\[ a = \square(\square(x \times y) + z) \]

where \( \square \) is one of the following rounding:

- 64-bit rounding
- 80-bit rounding
- double rounding
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 $\beta = 2050 \times 2^{-64}$

$\gamma = 2049 \times 2^{-1086}$
Theorem 1

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

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

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

or

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

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

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

proved in Coq

(Available at http://www.lri.fr/~nguyen/research/rnd_64_80_post.html)
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

\[ \square(x) = x \]

FMA (Fused Multiply-Add)

Sylvie Boldo and Thi Minh Tuyen Nguyen
Floating-point arithmetic

Hardware-independent proofs

A case study

Conclusion and future work

Rounding error in presence of FMA

64-bit rounding

80-bit rounding

double rounding

"identity" rounding

FMA (Fused Multiply-Add)

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

\(\square(x) = x\)

Sylvie Boldo and Thi Minh Tuyen Nguyen

Hardware-independent proofs

NFM 2010
Rounding error in presence of FMA

64-bit rounding

80-bit rounding

double rounding

"identity" rounding

\( \square(x) = x \)

FMA (Fused Multiply-Add)

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

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

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

\[ \Box(x) = x \]

FMA (Fused Multiply-Add)

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

"Identity" rounding
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.
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.

\[
\begin{align*}
x_1 &= a \times b \\
x_2 &= x_1 + c \\
\vdots \\
x_n &= x_{n-1} \times d
\end{align*}
\]
Theorem 2

Theorem

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.

$x_1 = a \times b$
$x_2 = x_1 + c$
$\vdots$
$x_n = x_{n-1} \times d$

Frama-C

$\overrightarrow{\text{Theorem 1}}$ $x_1 \in I_1(a, b)$
$\overrightarrow{\text{Theorem 1}}$ $x_2 \in I_2(x_1, c)$
$\vdots$
$\overrightarrow{\text{Theorem 1}}$ $x_n \in I_n(x_{n-1}, d)$
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.

$x_1 = a \times b$
$x_2 = x_1 + c$
$\vdots$
$x_n = x_{n-1} \times d$

Theorem 1 $\xrightarrow{}$ $x_1 \in I_1(a, b)$
Theorem 1 $\xrightarrow{}$ $x_2 \in I_2(x_1, c)$
$\vdots$
Theorem 1 $\xrightarrow{}$ $x_n \in I_n(x_{n-1}, d)$

$\rightarrow x_n \in I$

Frama-C

Gappa
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.

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

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

$\rightarrow x_n \in \mathcal{I}$

Correct $\forall$ compiler and architecture
Outline

1. Floating-point arithmetic
2. Floating-point computations independent to hardwares and compilers
3. A case study
4. Conclusion and future work
A case study

KB3D (NASA Langley Research Center)
- An aircraft conflict detection and resolution program
- Formally proved correct using PVS (C. Muñoz, G. Dowek...)
- 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 will 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 (C. Muñoz, G. Dowek...)
- 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 will go to its left or its right
```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.ffffffffffffffffp -1; // vy = 1 - 2^{-53}
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = %d\n", result);
}
```

 GCC eps_line.c

 Result = 1

 GCC -mfpmath=387 eps_line.c

 Result = -1

 Sylvie Boldo and Thi Minh Tuyen Nguyen
```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.fffffffffffffff0p-1;  // vy = 1 - 2^-53
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = %d\n", result);
}
```
```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.fffffffffffffff0p -1; // vy = 1 - 2^{-53}
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = %d\n", result);
}
```
```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 = -0x1.0000000000001p0;  // sx = -1 - 2^{-52}
    double vx = -1.0;
    double sy = 1.0;
    double vy = 0x1.fffffffffffffffip -1;  // vy = 1 - 2^{-53}
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = \%d\n", result);
}
```
```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 = -0x1.0000000000000001p0; // sx = -1 - 2^{-52}
    double vx = -1.0;
    double sy = 1.0;
    double vy = 0x1.fffffffffffffff8p -1; // vy = 1 - 2^{-53}
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = %d\n", result);
}
```

```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.fffffffffffffffpp -1; // vy = 1 - 2^{-53}
    int result = eps_line(sx, sy, vx, vy);
    printf("Result = %d\n", result);
}
```

```
```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 = -0x1.0000000000001p0; // sx = -1 - 2^{-52}
    double vx = -1.0;
    double sy = 1.0;
    double vy = 0x1.fffffffffffffff fp -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
```
/**
 * Logic integer \( l_{\text{sign}}(\text{real } x) = (x \geq 0.0) ? 1 : -1; \)
 */

/*@ requires \( e1 \leq x - \text{exact}(x) \leq e2; @*/

@ ensures \( \text{abs}(\text{result}) \leq 1 \) &&
  \( (\text{result} \neq 0 \implies \text{result} == l_{\text{sign}}(\text{exact}(x))) \);

int \text{sign}(\text{double } x, \text{double } e1, \text{double } e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}

/*@ requires @*/

@ \( sx == \text{exact}(sx) \) \&\& \( sy == \text{exact}(sy) \) \&\&
  \( vx == \text{exact}(vx) \) \&\& \( vy == \text{exact}(vy) \) \&\&
  \( \text{abs}(sx) \leq 100.0 \) \&\& \( \text{abs}(sy) \leq 100.0 \) \&\&
  \( \text{abs}(vx) \leq 1.0 \) \&\& \( \text{abs}(vy) \leq 1.0; \)

@ ensures
  \( \text{result} != 0 \implies \)
  \( \text{result} == l_{\text{sign}}(\text{exact}(sx) \times \text{exact}(vx) + \text{exact}(sy) \times \text{exact}(vy)) \)
  \( \times l_{\text{sign}}(\text{exact}(sx) \times \text{exact}(vy) - \text{exact}(sy) \times \text{exact}(vx)) \);

/*@*/

int \text{eps_line}(\text{double } sx, \text{double } sy, \text{double } vx, \text{double } vy) {
  \text{int } s1=\text{sign}(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  \text{int } s2=\text{sign}(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);
  \text{return } s1*s2;
}
int sign(double x, double e1, double e2) {
    if (x > e2) return 1;
    if (x < e1) return -1;
    return 0;
}
int eps_line(double sx, double sy, double vx, double vy) {
    int s1 = sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
    int s2 = sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);
    return s1*s2;
}
```c
int sign(double x, double e1, double e2) {
    if (x > e2) return 1;
    if (x < e1) return -1;
    return 0;
}
```

```c
int eps_line(double sx, double sy, double vx, double vy) {
    int s1 = sign(sx * vx + sy * vy, -0x1.90641p-45, 0x1.90641p-45);
    int s2 = sign(sx * vy - sy * vx, -0x1.90641p-45, 0x1.90641p-45);
    return s1 * s2;
}
```
```c
int sign(double x, double e1, double e2) {
    if (x > e2) return 1;
    if (x < e1) return -1;
    return 0;
}
```

```c
int eps_line(double sx, double sy, double vx, double vy) {
    int s1 = sign(sx * vx + sy * vy, -0x1.90641p-45, 0x1.90641p-45);
    int s2 = sign(sx * vy - sy * vx, -0x1.90641p-45, 0x1.90641p-45);
    return s1 * s2;
}
```
```c
//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;

/*@ requires e1 <= x - \exact(x) <= e2;
@ ensures |\result| <= 1 && 
@ \result != 0 ==> \result == l_sign(\exact(x)); */

int sign(double x, double e1, double e2) {
    if (x > e2) return 1;
    if (x < e1) return -1;
    return 0;
}

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

int eps_line(double sx, double sy, double vx, double vy) {
    int s1 = sign(sx * vx + sy * vy, -0x1.90641p-45, 0x1.90641p-45);
    int s2 = sign(sx * vy - sy * vx, -0x1.90641p-45, 0x1.90641p-45);
    return s1 * s2;
}
```
Strict IEEE-754: $e_2 = -e_1 = 0x1p-45$ / requires $e_1 < x - \text{exact}(x) < e_2$; ensures $|\text{result}| < 1$ && (result != 0 = = sign(\text{exact}(x)) ) ;

```c
int sign(double x, double e1, double e2)
{
    if (x > e2) return 1;
    if (x < e1) return -1;
    return 0;
}
```

Arch-independent model: $e_2 = -e_1 = 0x1.90641p-45$
*/@ requires e1 <= x - exact(x) <= e2;
@ ensures abs(result) <= 1 &&
@ (result != 0 ==> result == l_sign(exact(x)));@*/

int sign(double x, double e1, double e2) {
    if (x > e2) return 1;
    if (x < e1) return -1;
    return 0;
}

 Arch-independent model: e2 = -e1 = 0x1.90641p-45
strict IEEE-754: $e_2 = -e_1 = 0x1p^{-45}$

```c
int sign(double x, double e1, double e2) {
    if (x > e2) return 1;
    if (x < e1) return -1;
    return 0;
}
```
strict IEEE-754: e2 = -e1 = 0x1p-45

/\ requires e1 <= x - \exact(x) <= e2;
@ ensures \abs(\result) <= 1 &&
@ (\result != 0 ==> \result == l_sign(\exact(x)));
@*/

int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}

Arch-independent model: e2 = -e1 = 0x1.90641p-45
Outline

1. Floating-point arithmetic
2. Floating-point computations independent to hardwares and compilers
3. A case study
4. Conclusion and future work
Conclusion

An approach

- gives correct rounding errors whatever the architecture and the choices of the compiler
- is implemented in the Frama-C for all basic operations with the same conditions
- can be applied to single precision computation
- is proved correct in Coq

Drawback

- time to run a program verification (10s for a proof obligation)
- Incomplete: only proves rounding errors
Future work

- reduce time to run
- allow the compiler to do anything, including re-organizing the operations
  Example: if $|e| \ll |x|$
    - $(e + x) - x = 0$
    - $e + (x - x) = e$
- look into the assembly
  - know the order of the operations
  - know the precision used for each operation
  - know if the architecture supports FMA or not
Thank you for your attention!