Custom floating-point arithmetic for integer processors: algorithms, implementation, and selection

Jingyan Jourdan-Lu

Computer Arithmetic group *ARIC*
ENS de Lyon

Compilation Expertise Center
STMicroelectronics Grenoble

Advisor: Jean-Michel Muller
Co-advisors: Claude-Pierre Jeannerod, Christophe Monat

Context

Efficient support of **IEEE floating-point arithmetic** on integer processors requires

- **software libraries** emulating standard-compliant operations
  - usually written in C for portability
  - better performance if processor features considered
- **compiler optimizations** supporting the target processor
  - efficient code selection when compiling the libraries
  - efficient selection of library operators for applications

Our current target is the **ST231**, a very long instruction word (VLIW) integer processor from the ST200 family.

Context (cont'd)

So far, two software libraries have been used on the ST231:

- SoftFloat [Hauser, 2000]
- FLIP [Arénaire team, 2009]

FLIP achieves higher **instruction level parallelism (ILP)** for each arithmetic operator [Revy, 2009]:

	SoftFloat cycles	FLIP cycles	Speedup
$+$	48	26	1.9x
$-$	49	26	1.9x
$\times$	31	21	1.5x
$/$	177	34	5.2x
$\sqrt{}$	95	23	4.1x

Motivation

In application codes, basic operators are often **too generic**.

For example,

- products by special constants like 2.0f or 0.5f
- squares
- 2-norm computations:

```
float two_norm(float a[], int n){
  int i; float s = 0.0f;
  for (i=0; i<n; i++)
    s = s + a[i]*a[i];  ⇝ fused square-add
  return sqrtf(s); }
```

To fully exploit such codes, **operator customization** is needed.

Motivation (cont'd)

Another example: FFT computation

```
for (k=j; k<n; k=k+n2 ){  // float t1, t2, x[], y[], s, c
    t1 = c*x[k+n1] - s*y[k+n1];
    t2 = s*x[k+n1] + c*y[k+n1];
    x[k+n1] = x[k] - t1; x[k] = x[k] + t1;
    y[k+n1] = y[k] - t2; y[k] = y[k] + t2;
}
```

- each of `t1` and `t2` is a two-dimensional dot product: $xy + zt$
- each of (`x[k+n1]`,`x[k]`) and (`y[k+n1]`,`y[k]`) corresponds to simultaneous addition and subtraction: $(x + y, x - y)$

Custom operators studied during this thesis

- **Specialized** operators:

multiplication by two (mul2)	$2x$
division by two (div2)	$x/2$
scaling (scaleB)	$x \cdot 2^n$ with n an integer
squaring (square)	x^2
addition of nonnegative terms (addnn)	$x + y$ with $x \geqslant 0$ and $y \geqslant 0$

- **Fused** operators:

fused multiply-add (FMA)	$xy + z$
fused square-add (FSA)	$x^2 + z$ with $z \geqslant 0$
two-dimensional dot product (DP2)	$xy + zt$
sum of two squares (SOS)	$x^2 + y^2$

- **Paired** operators:

simultaneous addition and subtraction (addsub)	$(x + y, x - y)$
simultaneous sine and cosine (sincos)	$(\sin x, \cos x)$

Contributions: computer arithmetic aspects

Algorithms and **implementations** for all these custom operators

- **accurate**: IEEE compliant
- **fast**: low latencies via high ILP exposure
- **scalable**: designs parametrized by the floating-point format
  - only one correctness proof
  - C code generation for single and double precision

Contributions: compilation aspects

Code-selection optimizations in the ST200 C/C++ compiler

- **better selection when compiling custom operators**
  - enhanced 64-bit integer support
  - integer range analysis for shift operators
- **selection of custom operators from applications**
  - optimizations at different intermediate representation levels
  - extension of range analysis framework from integers to floating-point numbers

Outline

IEEE 754 standard

Floating-point arithmetic is specified by the **IEEE 754 standard**.

This standard (1985-2008) aims at increased robustness, efficiency and portability of numerical programs.

It specifies

- **data** and their **encoding into integers** for various formats
- **results** of operations
- **rounding** modes
- **exceptions** and their handling
- conversions between formats

Floating-point data

Finite nonzero floating-point numbers: $x = (-1)^s \cdot m \cdot 2^e$ with

- sign bit s
- integer exponent e such that $e_{\mathsf{min}} \leqslant e \leqslant e_{\mathsf{max}}$
- p-bit significand $m = (m_0.m_1 \cdots m_{p-1})_2$

$\hookrightarrow$ x is **normal** $(m_0 = 1)$ or **subnormal** $(m_0 = 0$ and $e = e_{\mathsf{min}})$

Special data: signed **zeros** and **infinities**, not-a-numbers (**NaN**)

Formats

Floating-point data are specified for some values of the precision p and the exponent range $[e_{\mathrm{min}}, e_{\mathrm{max}}]$

Basic standard formats:

	p	e_{min}	e_{max}	w	k
binary32 ("single")	24	-126	127	8	32
binary64 ("double")	53	-1022	1023	11	64
binary128 ("quad")	113	-16382	16383	15	128

They are special cases of the **binary$k$ format**, for which:

$$k = p + w \qquad \text{and} \qquad \begin{aligned} e_{\mathrm{max}} &= 1 - e_{\mathrm{min}} \\ &= 2^{w-1} - 1 \end{aligned}$$

Encoding into integers

Binary$k$ fl-point data have encodings into k-bit unsigned integers:

Finite nonzero number $x = (-1)^s \cdot m \cdot 2^e$ encoded uniquely into $X \in \mathbb{N}$, whose bitstring is

s	$E_{w-1} \cdots E_0$	$m_1 \cdots m_{p-1}$

and where $\sum_{i=0}^{w-1} E_i 2^i$ is the biased exponent $e - e_{\mathsf{min}} + m_0 \geqslant 0$.

Zeros, **infinities**, **NaNs** encoded by special values of X.

Correct rounding and exceptions

Correct rounding (CR): operation performed "as if to infinite precision" and then rounded.

Rounding modes: RN (default), RD, RU, RZ.

$\hookrightarrow$ the 2008 revision of IEEE 754

- specifies the FMA operation (CR for $xy + z$),
- *requires* CR for basic arithmetic,
- *recommends* CR for functions like sine, exponential...

Some exceptions:

- If NaN in input or $0/0$ or $\sqrt{-1}$ then invalid: return NaN
- If exact result outside the normal floating-point range then overflow or gradual underflow: return $\pm\infty$ or a subnormal.

Outline

ST231: overview

A four-way VLIW integer processor from the ST200 family:

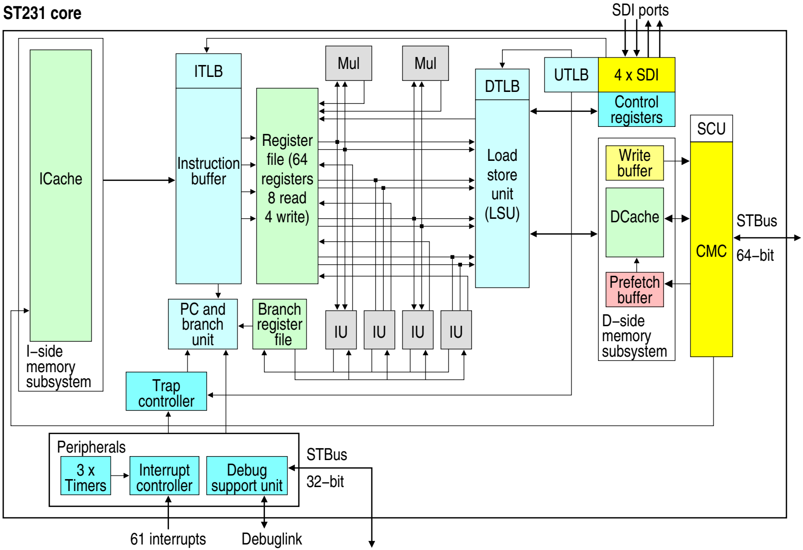
- up to 4 instruction words grouped into 1 bundle
- up to 4 instructions executed in 1 cycle

↪ key to realize **instruction level parallelism**

Typical applications:

- a media processor with an embedded OS
- a host processor running Linux and applications

ST231: architecture



ST231: key features for floating-point support

- select instruction, $\texttt{slct}\ \mathtt{R_{DEST}} = \mathtt{B}, \mathtt{Opnd1}, \mathtt{Opnd2}$, computing $\mathtt{R_{DEST}} = \mathtt{B}\ ?\ \mathtt{Opnd1} : \mathtt{Opnd2}$
  $\rightarrow$ transform branches to straight line code
- 32×32-bit multiplication: latency of 3 cycles, a maximum of 2 multiplications issued at each cycle $\rightarrow$ polynomial evaluation
- leading-zero count instruction $\rightarrow$ subnormal numbers support
- encoding immediate operands up to 32 bits in instruction word
  $\rightarrow$ masking, encoding of polynomial coefficients and of special floating-point data (NaN...)
- min, max, shift-and-add $(a \ll b) + c$ with $b \in \{1, 2, 3, 4\}$

Exposing ILP by code speculation

- We distinguish between **special input** and **generic input**.
- The implementation of each operator essentially reduces to **three independent tasks** T_1, T_2, and T_3:

evaluate the condition C = "x is special"	$[T_1]$
if C is true then	
handle special input	$[T_2]$
else	
handle generic input	$[T_3]$

T_1, T_2, and T_3 are computed in parallel by code speculation.

Illustration: bundle occupancy for our square operator

[Jeannerod, Jourdan-Lu, Monat, Revy (ARITH 2011)]

```
uint square(uint X) {
  Cspec = ...  ; // T1
  if (Cspec)
    { ... } // T2
   else
    { ... } // T3
}
```

$\longrightarrow$

Cycle	issue 1	issue 2	issue 3	issue 4
0	shared	T3	T2	T2
1	shared	T3	T3	T3
2	T3	T3	T3	T3
3	T3	T2	T2	
4	T3	T3	T3	T3
5	T3	T3	T3	T3
6	T3	T3	shared	T1
7	T3	T3	T2	T2
8	T3	T3	T2	T2
9	T3	T2	T1	
10	slct	return		

- Optimize the *a priori* most expensive task
- Try to reuse its intermediate results for the other two tasks

Outline

Simultaneous sine and cosine [Jeannerod and Jourdan-Lu (ASAP 2012)]

Sine and cosine often evaluated at a same floating-point input x, and their routines have much in common [Markstein (2003)].

Classically, evaluation in 3 steps: [Muller (1997), Ercegovac, Lang (2004)]

1. Range reduction: compute x^* such that

$$x^* \in [-\tfrac{\pi}{4}, \tfrac{\pi}{4}] \quad \text{and} \quad x^* = x - k\tfrac{\pi}{2}, \quad k \in \mathbb{Z}.$$

2. Evaluation of sin and cos at reduced argument $\rho = |x^*|$.
3. Reconstruction: $(\sin x, \cos x) = \begin{cases} (\pm \sin \rho, \cos \rho) & \text{if } k \text{ even}, \\ (\pm \cos \rho, \pm \sin \rho) & \text{if } k \text{ odd}. \end{cases}$

Problem: how to implement step 2 accurately and fast on our target?

Our design and implementation of step 2

- **New algorithms for sine and cosine** over $[0, \frac{\pi}{4}]$:
  - **accurate**: error proven to be at most of 1 ulp (*unit in the last place*)
  - **fast**: 19 and 18 cycles on ST231
- **C code for sincos**
  - **as fast as sine alone**
  - **faster than** the correctly-rounded **multiplication** of FLIP

$\hookrightarrow$ all this for single precision, including subnormals.

ulp function and 1-ulp accuracy [Muller et al. (2010)]

For any real number x, the **ulp function** is defined as

$$\boxed{\mathsf{ulp}(x) = \begin{cases} 0 & \text{if } x = 0, \\ 2^{\max\{e_{\mathsf{min}}, e\} - p + 1} & \text{if } |x| \in [2^e, 2^{e+1}). \end{cases}}$$

Let $\mathbb{F}$ be the set of binary32 finite floating-point numbers. Given $f \in \{\sin, \cos\}$ and $x \in \mathbb{F} \cap [0, \frac{\pi}{4}]$, we want $r \in \mathbb{F}$ such that

$$\big| r - f(x) \big| \leqslant \mathsf{ulp}\big(f(x)\big).$$

- This is **"1-ulp accuracy"** ($\approx$ all bits correct but possibly the last one).
- Such a precise **specification** is indispensable for establishing the accuracy of our C codes.

Algorithms for sine and cosine

Since 32×32-bit multipliers are available, a classical approach is via the **evaluation of polynomial approximants** [Tang (1990), Gal, Bachelis (1991)...]

$\hookrightarrow$ high-level algorithm for $x \in [0, \frac{\pi}{4}] \subset [0, 1)$:

1. If x close enough to zero then
   return x for $\sin x$, and $1 - 2^{-24}$ for $\cos x$
2. Else
   evaluate a pair of polynomials approximating $\sin x$ and $\cos x$

- Software toolchain for step 2: Sollya $\to$ CGPE $\to$ Gappa
  [Chevillard, Joldes, Lauter (2010); Mouilleron, Revy (2011); Melquiond (2009)]
- Steps 1 and 2 are independent $\implies$ obvious source of ILP
- Much more ILP can be exposed at the polynomial evaluation level

Polynomial evaluation for cosine

For our accuracy constraint, a polynomial of degree 6 is enough:

$$a(y) = a_0 + a_1 y + \cdots + a_6 y^6.$$

- Each a_i has $\leqslant 32$ fraction bits and is encoded in a uint32_t.
- y is a fixed-point approximation of x.
- We have chosen a **highly-parallel evaluation scheme**:

$$\Big(\big(a_0 + a_1 y\big) + \big(a_2 + a_3 y\big) z\Big) + \Big(\big(a_4 + a_5 y\big) + a_6 z\Big) z^2$$

with $z = y^2$.

$\hookrightarrow$ **accurate enough** and **2.2x faster than Horner's rule**

$$a_0 + y\big(\cdots + y(a_5 + a_6 y)\big).$$

Polynomial evaluation for sine

Over $[0, \frac{\pi}{4}]$, things are more difficult than for cosine:

- cosine was 'flat', ranging in $[0.707..., 1]$ $\Longrightarrow$ already fixed point
- sine ranges in $[0, 0.707...]$ $\Longrightarrow$ 'exponent' not known in advance

Classical workaround [Tang (1990)]:

1. instead of $\sin x$, approximate the function $\frac{\sin x}{x}$ ranging in $[0.8, 1]$
2. reconstruct using $\sin x = \frac{\sin x}{x} \times x$

$\hookrightarrow$ drawback: **steps 1 and 2 are not independent**.

Polynomial evaluation for sine (cont'd)

Our solution is to interleave steps 1 and 2:

- For $x = m \cdot 2^e$ we have $\frac{\sin x}{x} \times x = m\frac{\sin x}{x} \cdot 2^e$
- View $m\frac{\sin x}{x}$ as a bivariate function and approximate it by

$$b(m, x) = b_0 + m\, c(x), \qquad c(x) = c_0 + c_2x^2 + c_4x^4 + c_6x^6$$

- Evaluate b at (m, y) using a highly-parallel evaluation scheme:

$$b(m, x) = \Big(\big(b_0 + mc_0\big) + \big(mc_2\big)x\Big) + \Big(mc_4 + \big(mc_6\big)z\Big)z^2,$$

with $z = x^2$.

Results on ST231 for sine, cosine, and sincos

	latency L (cycles)	**instr. number N**	**IPC = N/L**
sinf	19	31	1.6
cosf	18	25	1.4
sincosf	19	47	2.4

- **sincos** obtained by inlining **as fast as sine alone**
- **faster than** best-known software implementation of IEEE 754 floating-point **multiplication** (21 cycles for single precision)

Bundle occupancy and shared resources for sincos

Cycle	issue 1	issue 2	issue 3	issue 4
0	shared	shared		
1	shared	shared	shared	sin
2	shared	shared	sin	sin
3	shared	sin	sin	
4	shared	sin	sin	
5	sin	sin	cos	cos
6	cos	cos	cos	cos
7	shared	sin	sin	sin
8	sin	sin	cos	cos
9	cos	cos	cos	cos
10	sin	cos	cos	
11	sin	sin	cos	
12	sin	cos		
13	sin	cos		
14	sin	sin		
15	sin	sin	cos	
16	sin	sin	cos	
17	sin	cos	cos	
18	shared	sin	cos	cos

- In 80% of the bundles, at least 3 slots used
- Shared computations:
  - unpacking of input x
  - $y \approx x$, y^2, y^4

Outline

Intermediate representations for st200cc

The ST200 C/C++ compiler, **st200cc**, is based on the Open64 technology and further developed by STMicroelectronics.

Two main **intermediate representations**:

- **Target-independent**: Winning Hierarchical Intermediate Representation Language (**WHIRL**)
  $\hookrightarrow$ *Most custom operators can be selected at this level.*

- **Target-dependent**: Code Generator Intermediate Representation (**CGIR**)
  $\hookrightarrow$ *Improvements for 64-bit integer support and shifts are done here.*

Selecting fused square-add (FSA) and nonnegative add (addnn) requires work at both the WHIRL and CGIR levels.

Example: selection of FSA ($x^2 + z$ with $z \geqslant 0$)

```
float two_norm(float a[], int n){
  int i; float s = 0.0f;
  for (i=0; i<n; i++)
    s = s + a[i]*a[i];
  return sqrtf(s); }
```

- **WHIRL level**: select the pattern `s + a[i]*a[i]` as a general FSA (gFSA), which
  - computes $x^2 + z$ without assuming $z \geqslant 0$
  - is not as fast as FSA
- **CGIR level**: analyze the range of variable `s`

Integer range analysis framework in st200cc

Implemented at CGIR level in st200cc, the integer range analysis framework consists of two phases:

- **range analysis**, based on **sparse conditional constant propagation algorithm** [Wegman and Zadeck (1991)], calculates the ranges for all variables.
- **range propagation**, based on the information analyzed by the propagation phase, performs various **code improvements**.

Integer range analysis for shift operators

[Bertin, Jeannerod, Jourdan-Lu, Knochel, Monat, Mouilleron, Muller, Revy (PASCO 2010)]

```
S = ...   //uint32_t S, nlz, L, u
nlz = countLeadingZeros(S);
u = max (3-nlz, 0);
L = S >> (5 + u);
```

Better ILP can be achieved thanks to range analysis.

original CGIR	range analysis	improved CGIR after range propagation
`r0 = ...`	$r0 \in [\perp, \perp]$	`r0 = ...`
`r1 = clz r0`	$r1 \in [0, 32]$	`r1 = clz r0`
`r2 = sub 3 r1`	$r2 \in [-28, 3]$	`r2 = sub 3 r1`
`r3 = max r2 0`	$r3 \in [0, 3] \subseteq [0, 31]$	`r3 = max r2 0` || `r7 = shr r0 5`
`r4 = add r3 5`	$r4 \in [5, 8] \subseteq [0, 31]$	`r5 = shr r7 r3`
`r5 = shr r0 r4`		

Analyzing the positivity of general FSA

CGIR instructions for the computation of s:

```
float two_norm(float a[], int n){
  int i;
  float s = 0.0f;          --CGIR-->    mov r1 = Const

  for (i=0; i<n; i++)
    s = s + a[i]*a[i];     --CGIR-->    call_gFSA r = r1, r2;
                                        mov r1 = r;
  return sqrtf(s); }
```

What remains is to check the **nonnegativity** of gFSA.

Bounding rule of general FSA

- From the arithmetic point of view, floating-point numbers $\mathtt{r1}$ and $\mathtt{r2}$ satisfy

$$\mathtt{r1} + \mathtt{r2} \cdot \mathtt{r2} \in [\min, \infty) \text{ when } \mathtt{r1} \in [\min, \max].$$

- Cast the range of $\mathtt{r1}$ to a pair of unsigned integers $[I_1, I_2]$, where I_1 and I_2 are the integer encodings of $\min$ and $\max$. For the binary32 format, we have

$$\mathtt{gFSA}(\mathtt{r1}, \mathtt{r2}) \in [\mathtt{I}_1, \mathtt{0x7fffffff}] \text{ when } [\mathtt{I}_1, \mathtt{I}_2] \subset [\mathtt{0}, \mathtt{0x7fffffff}].$$

When the sign bit of $\mathtt{r1}$ is zero, the sign bit of gFSA is zero.

Selecting FSA at the range propagation stage

Recall the `two_norm` function and its CGIR instructions:

```
float two_norm(float a[], int n){
  int i;
  float s = 0.0f;              --CGIR-->     mov r1 = Const

  for (i=0; i<n; i++)
    s = s + a[i]*a[i];         --CGIR-->     call_gFSA r = r1, r2;
                                             mov r1 = r;
  return sqrtf(s); }
```

Since `mov` copies the range of `gFSA` to `r1`, we can replace the general FSA by **FSA** at the range propagation stage if

$$\mathtt{r1} \in [0, \mathtt{0x7fffffff}].$$

Selecting FSA at the range propagation stage

Recall the `two_norm` function and its CGIR instructions:

```
float two_norm(float a[], int n){
  int i;
  float s = 0.0f;                 --CGIR-->    mov r1 = Const

  for (i=0; i<n; i++)
    s = s + a[i]*a[i];            --CGIR-->    call_FSA r = r1, r2;
                                               mov r1 = r;
  return sqrtf(s); }
```

Since `mov` copies the range of `gFSA` to `r1`, we can replace the general FSA by **FSA** at the range propagation stage if

$$\mathtt{r1} \in [0, \mathtt{0x7fffffff}].$$

Outline

Speedups and code size reductions on the ST231

	Speedup	CRR
mul2	4.2	0.15
div2	3.5	0.22
scalb	1.4	0.70
square	1.75	0.49
addnn	1.73	0.54
FSA	2.14	0.46
FMA	1.02	1.02
SOS	2.62	0.35
DP2	1.33	0.84
addsub	1.86	0.56
sincos	1.95	0.82

Speedup

$$= \frac{\text{latency of direct implementation}}{\text{latency of custom operator}}$$

Code reduction ratio (CRR)

$$= \frac{\text{size of custom operator}}{\text{size of direct implementation}}$$

Example of direct implementation:

DP2 as $\mathrm{RN}(\mathrm{RN}(xy) + \mathrm{RN}(zt))$

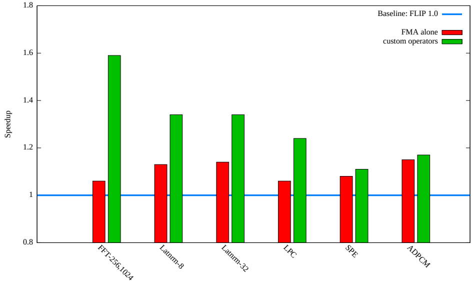
- Speedups up to 4.2 and CRRs as low as 0.15
- FMA's CRR due to bigger alignment logic in addition stage
- On the ST231, a more interesting operator is DP2

Performances on the UTDSP benchmark

UTDSP benchmark [Lee (1992)]

- Assessing C compilers' efficiency on typical DSP codes
- Good predictor of improvements achievable at a larger scale
- Kernels (FFT, Latnrm...) and applications (LPC, SPE, ADPCM...)

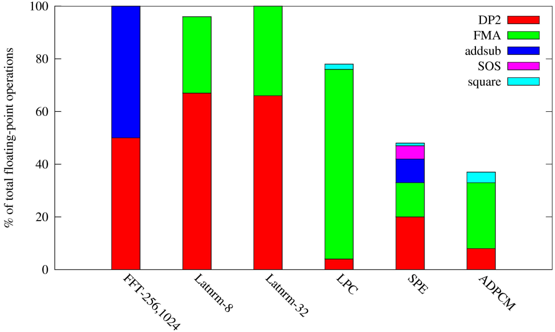
Speedups thanks to our custom operators:



- Using **FMA alone**, beneficial effect of fewer function calls
- Speedup factor up to **1.6x** using the **full set of custom operators**

Performances on the UTDSP benchmark (cont'd)

Usage of custom operators



- (Close to) 100% usage of custom operators in several test suites
- Two key combinations are (**DP2**, **addsub**) and (**DP2**, **FMA**)

Outline

Conclusions

A set of **custom operators** can significantly **improve the performances** of floating-point applications on integer processors.

This has required

- the design of **useful** and **fast** operators
- developments at both the **arithmetic** and **compilation** levels

Specifically,

- highly efficient paired operators like sincos
- practical impact of fast DP2
- floating-point range analysis based on integer framework

Perspectives

Computer arithmetic designs:

- Paired sincos operator:
  - extension to double precision
  - efficient range reduction
- Study of fused operators like $x + y + z$ or $\sqrt{x^2 + y^2}$
- Performance impact of relaxing the 1-ulp accuracy constraint?

Compilation optimizations:

- Development of native 128-bit integer support in the compiler
- Improve range analysis techniques for floating-point operations