On Automatically Proving the Correctness of `math.h` Implementations*

Wonyeol Lee
Stanford (on leave)
KAIST

Rahul Sharma
Microsoft Research

Alex Aiken
Stanford

*Presented at [PLDI'16] and [POPL'18].

FPTalks 2020

Our Goal (Informal)

$$\log x$$

mathematical specification $\boldsymbol{f}$

Our Goal (Informal)

$$\log x$$

mathematical specification $\boldsymbol{f}$

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

Our Goal (Informal)

$$\log x$$

mathematical specification $\boldsymbol{f}$

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

Our Goal (Informal)

infinite # bits

$$\log x$$

mathematical specification $\boldsymbol{f}$

fixed # bits

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

Our Goal (Informal)

infinite # bits

$\log x$

mathematical specification $\boldsymbol{f}$

$\neq$

fixed # bits

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

Our Goal (Informal)

infinite # bits

$\log x$

mathematical specification $\boldsymbol{f}$

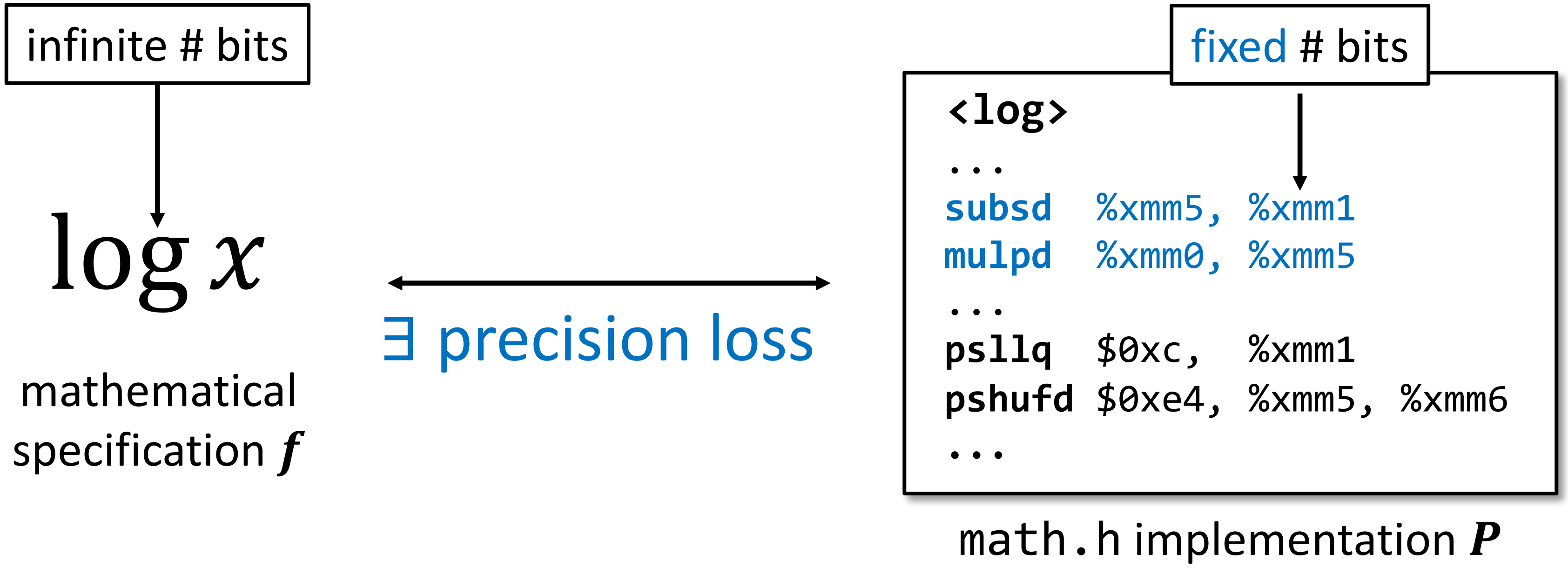
∃ precision loss

fixed # bits

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

Our Goal (Informal)



- Find a tight bound on the precision loss.

Our Goal (Formal)

$$\log x$$

mathematical specification $\boldsymbol{f}$

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

Our Goal (Formal)

$$\log x$$

mathematical specification $\boldsymbol{f}$

input interval
$\boldsymbol{X} = (0, 2^{1024})$

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

Our Goal (Formal)

$$\log x$$

mathematical specification $\boldsymbol{f}$

input interval
$\boldsymbol{X} = (0, 2^{1024})$

↔

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

- Find a small $\Theta > 0$ automatically such that

$$\mathrm{err}(\boldsymbol{f}(x), \boldsymbol{P}(x)) \leq \Theta \quad \text{for all } x \in \boldsymbol{X}.$$

Our Goal (Formal)

$\log x$

mathematical specification $\boldsymbol{f}$

input interval
$\boldsymbol{X} = (0, 2^{1024})$

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

- Find a small $\Theta > 0$ automatically such that

$$\mathrm{err}(\boldsymbol{f}(x), \boldsymbol{P}(x)) \leq \Theta \quad \text{for all } x \in \boldsymbol{X}.$$

- Prove a bound on the maximum precision loss.

Challenges

$$\log x$$

mathematical specification ***f***

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation ***P***

Challenges

$$\log x$$

mathematical specification $\boldsymbol{f}$

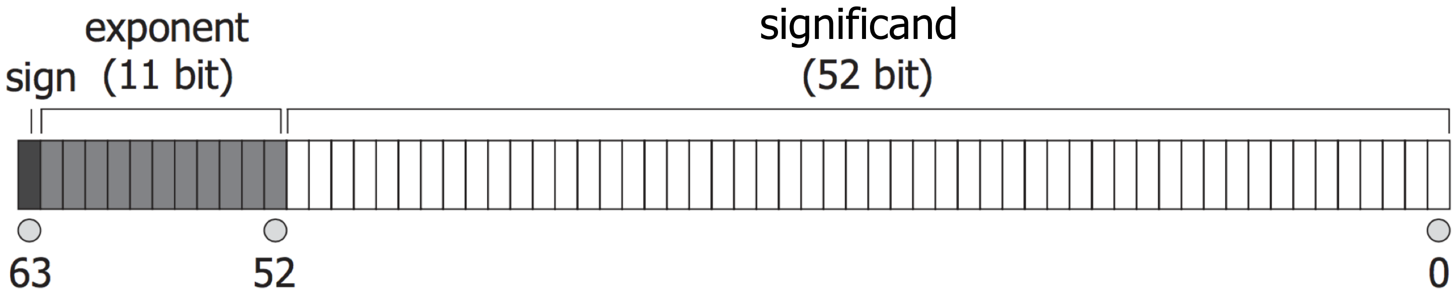
```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

math.h implementation $\boldsymbol{P}$

(1) $\boldsymbol{P}$ mixes floating-point and bit-level operations.

Challenge 1: Bit-Level Operations

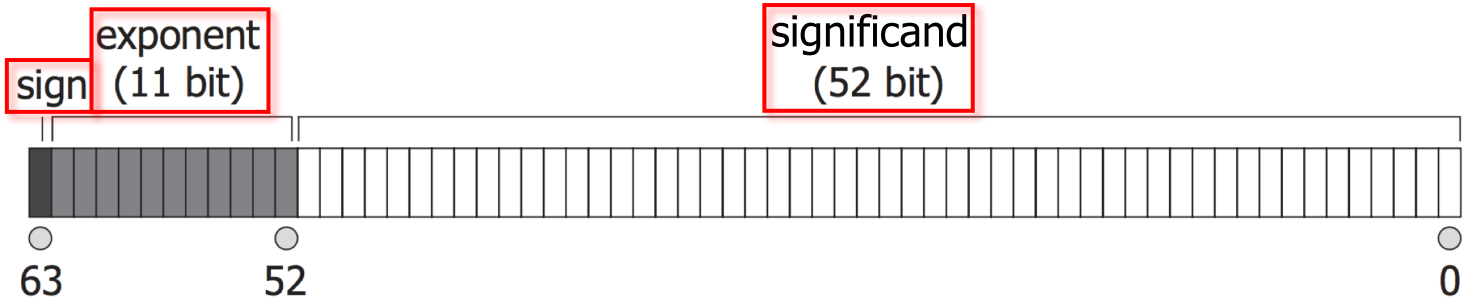
- Floating-point numbers (64-bit double-precision).



Type	Exponent (p)	Significand (f)	Value
Zero	0	0	$(-1)^s \cdot 0$
Subnormal	0	$\neq 0$	$(-1)^s \cdot 2^{-1022} \cdot 0.f$
Normal	$[1, 2046]$	unconstrained	$(-1)^s \cdot 2^{p-1023} \cdot 1.f$
Infinity	2047	0	$(-1)^s \cdot \infty$
NaN	2047	$\neq 0$	$(-1)^s \cdot \perp$

Challenge 1: Bit-Level Operations

- Floating-point numbers (64-bit double-precision).



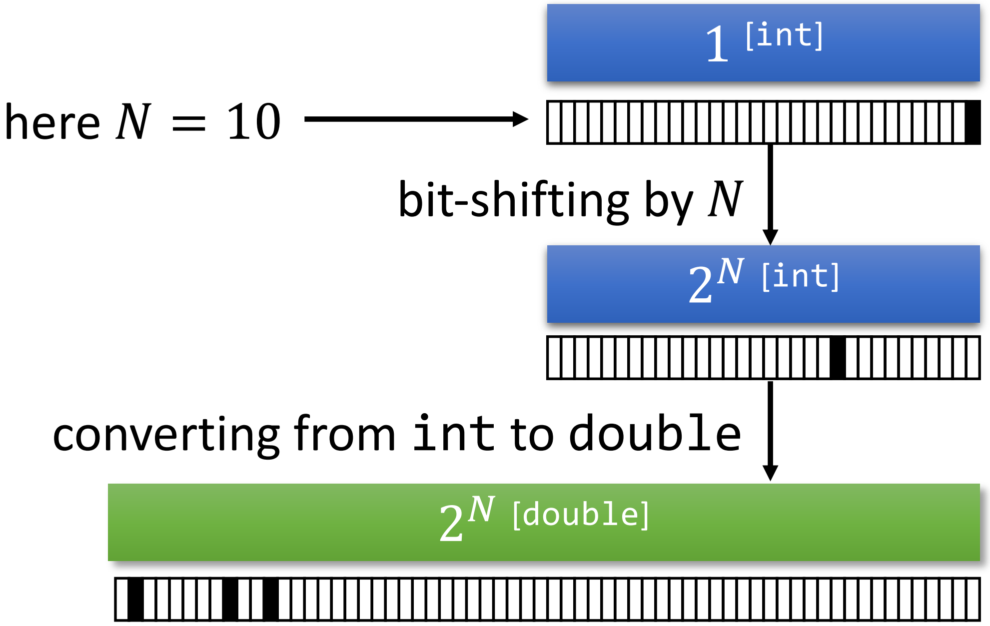
Type	Exponent (p)	Significand (f)	Value
Zero	0	0	$(-1)^s \cdot 0$
Subnormal	0	$\neq 0$	$(-1)^s \cdot 2^{-1022} \cdot 0.f$
Normal	$[1, 2046]$	unconstrained	$(-1)^s \cdot 2^{p-1023} \cdot 1.f$
Infinity	2047	0	$(-1)^s \cdot \infty$
NaN	2047	$\neq 0$	$(-1)^s \cdot \perp$

Challenge 1: Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`).

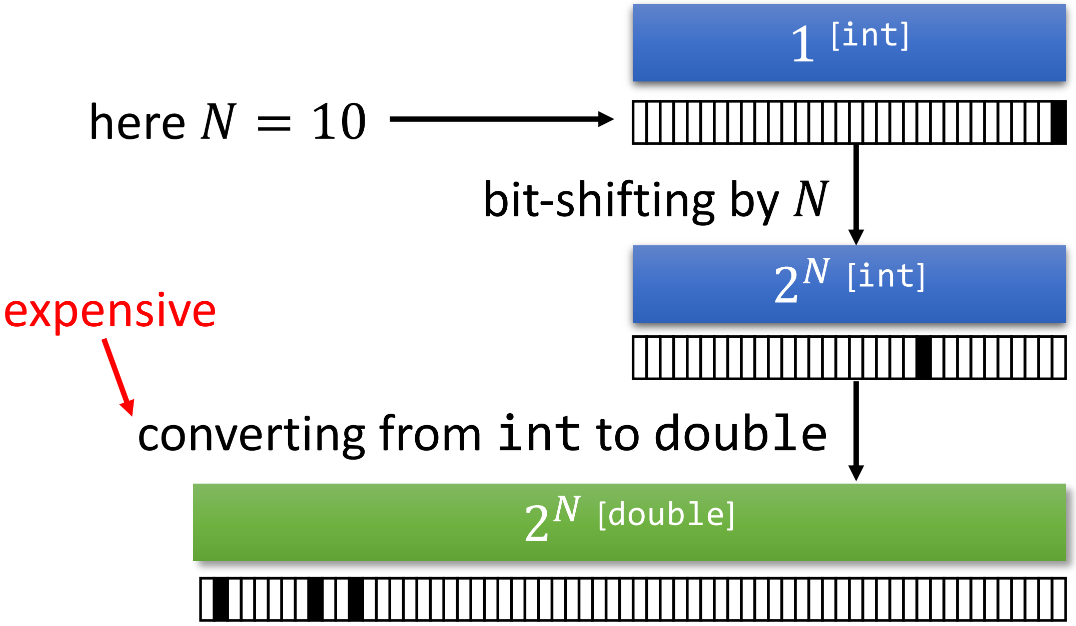
Challenge 1: Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`).



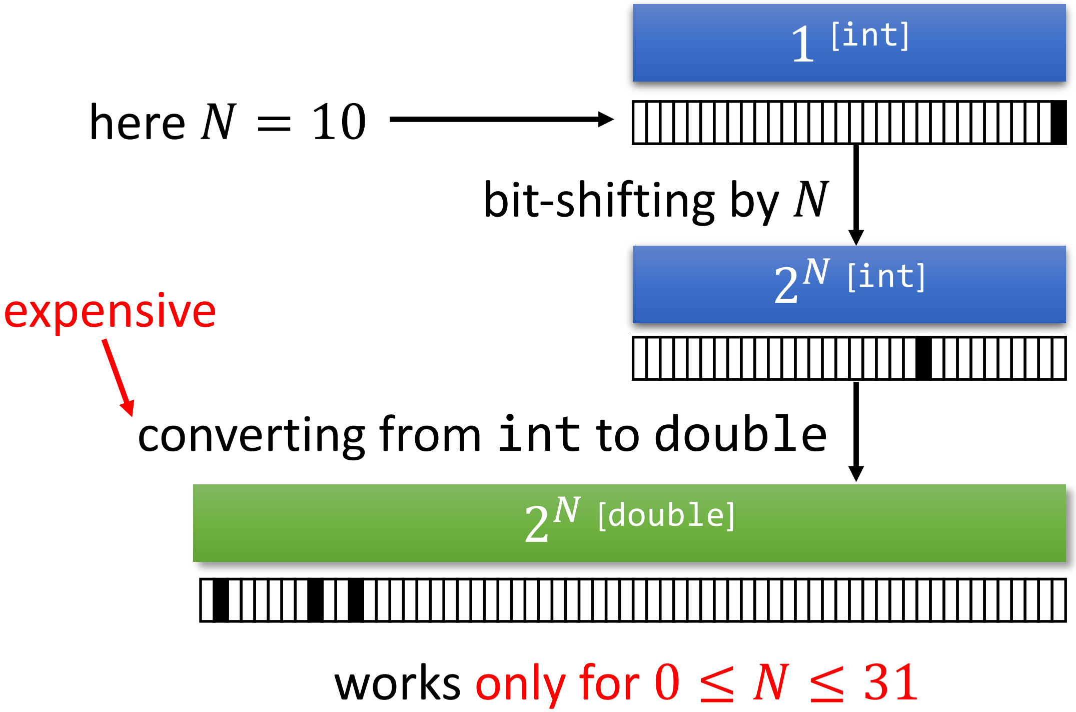
Challenge 1: Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`).



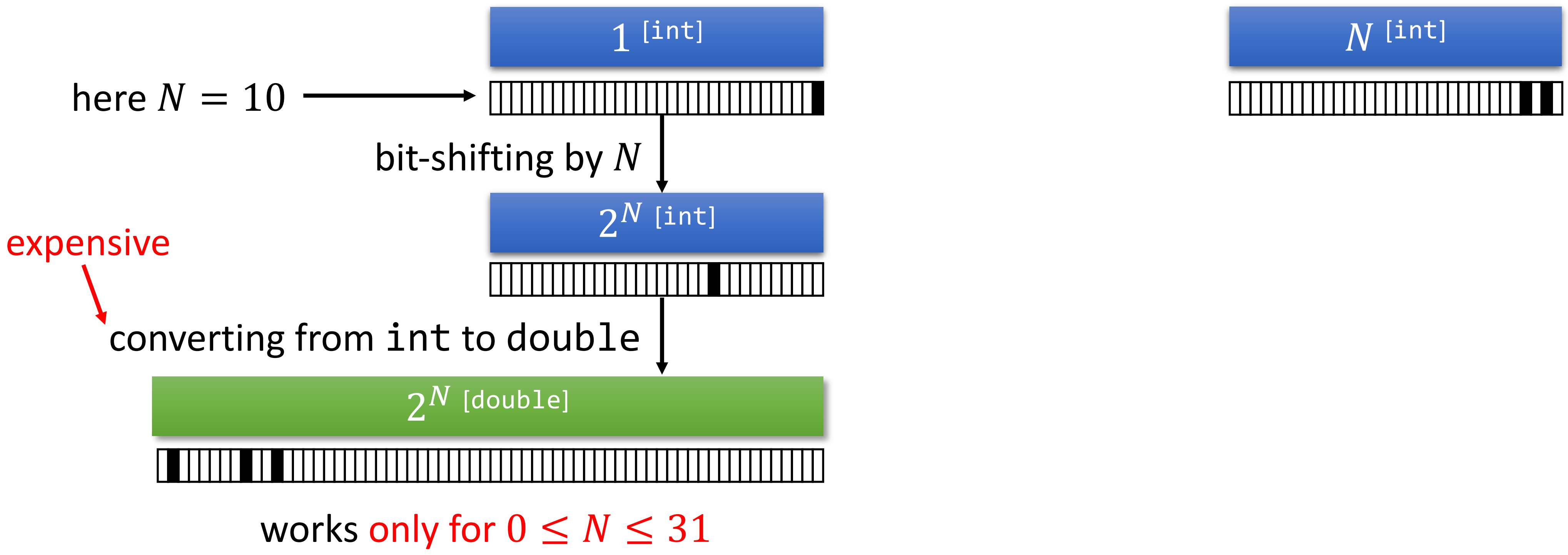
Challenge 1: Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`).



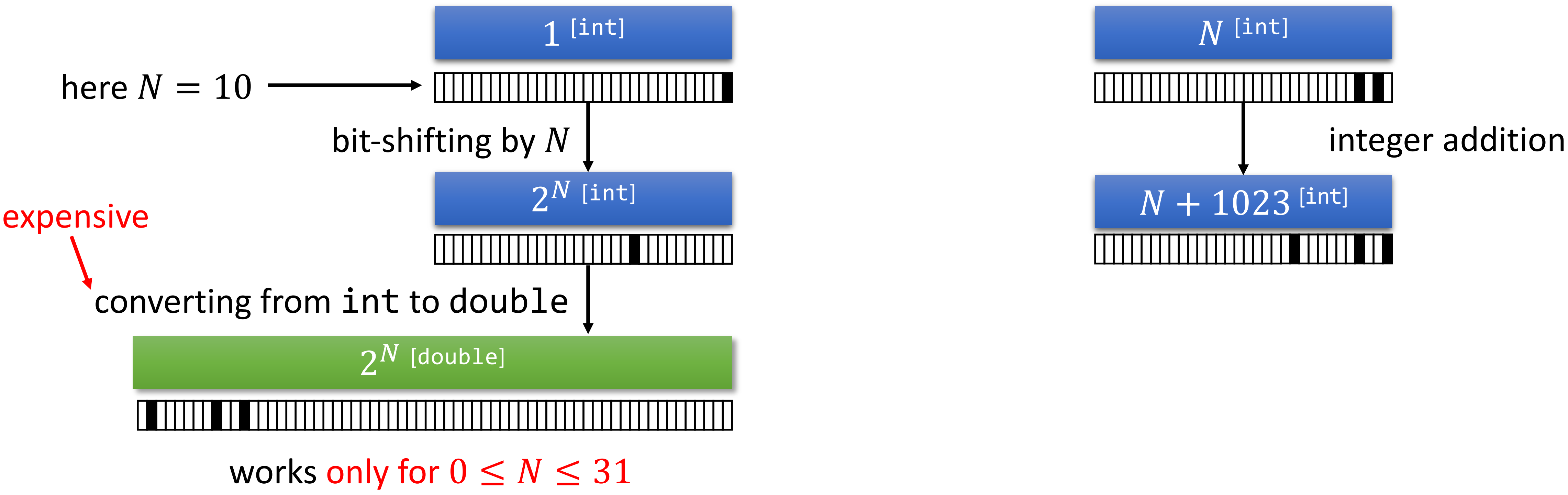
Challenge 1: Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`).



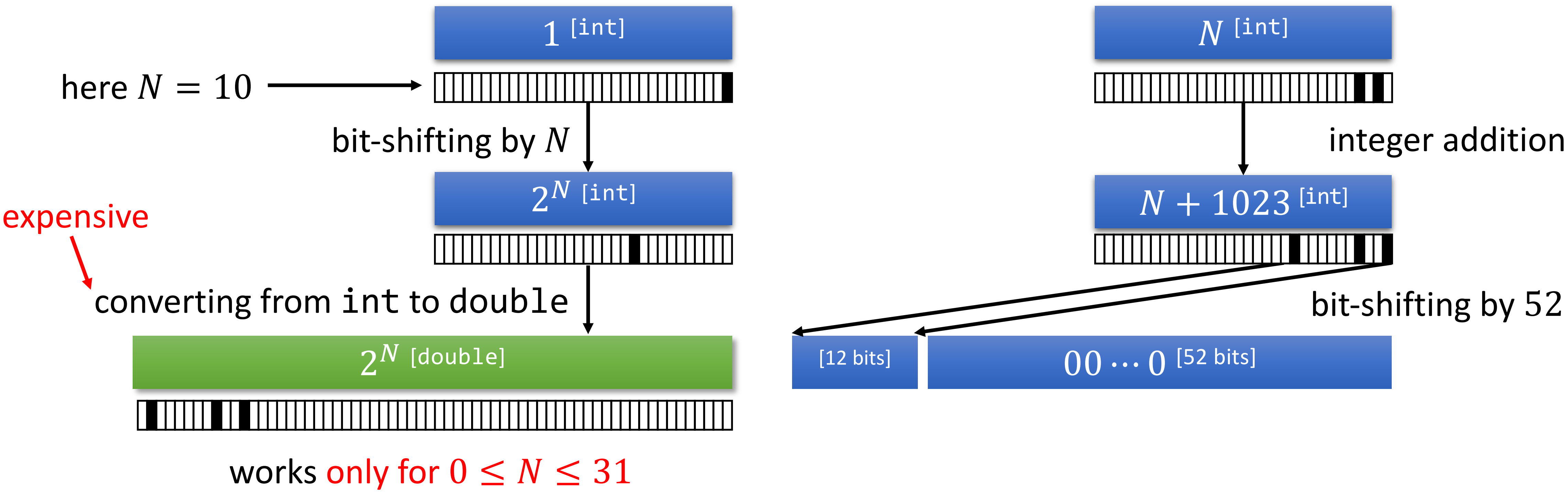
Challenge 1: Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`).



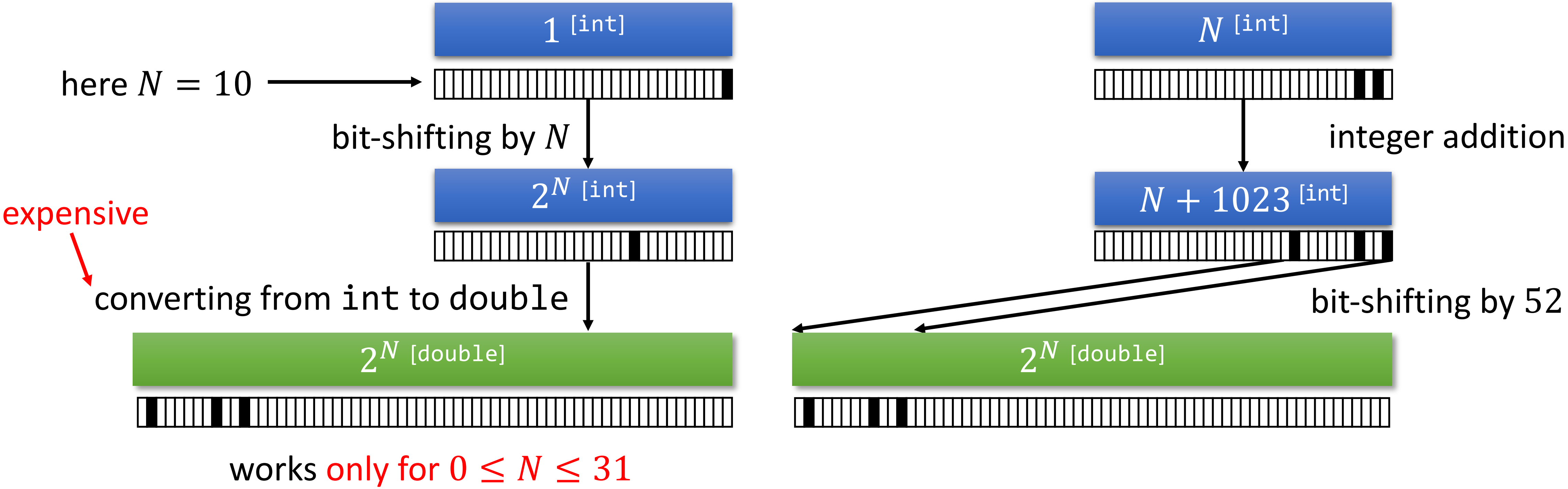
Challenge 1: Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`).



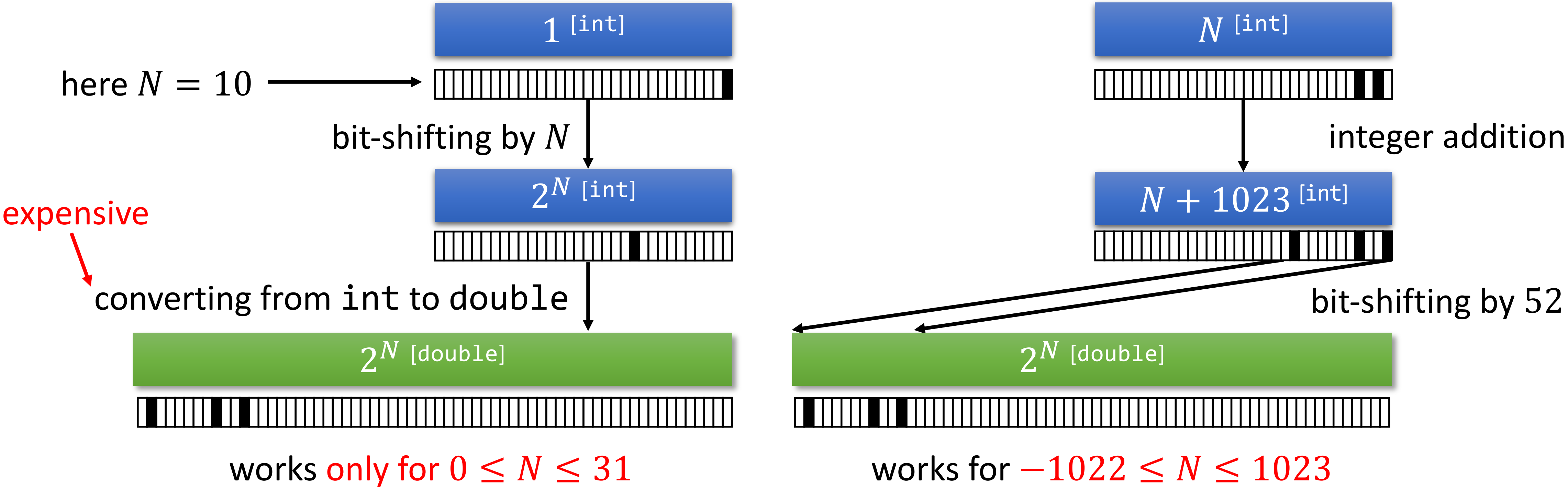
Challenge 1: Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`).



Challenge 1: Bit-Level Operations

- Example: Given N (in `int`), compute 2^N (in `double`).



Challenge 1: Bit-Level Operations

- Such bit-level operations are ubiquitous in industry standard implementations of `math.h` (e.g., Intel's implementation).

Challenge 1: Bit-Level Operations

- Such bit-level operations are ubiquitous in industry standard implementations of `math.h` (e.g., Intel's implementation).

- But reasoning about such mixed codes is difficult.
  - Floating-point operations: continuous.
  - Bit-level operations: discrete.

Challenges

$$\log x$$

mathematical specification $\boldsymbol{f}$

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

(1) $\boldsymbol{P}$ mixes floating-point and bit-level operations.

Challenges

$$\log x$$

mathematical specification $\boldsymbol{f}$

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

(1) $\boldsymbol{P}$ mixes floating-point and bit-level operations. [PLDI'16]

Challenges

$\log x$

mathematical specification $\boldsymbol{f}$

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

(1) $\boldsymbol{P}$ mixes floating-point and bit-level operations. [PLDI'16]

(2) $\boldsymbol{P}$ is claimed to have precision loss of < 1 ulp.

Challenge 2: Highly Accurate Implementations

- Example: `log` has precision loss of < 1 ulp iff

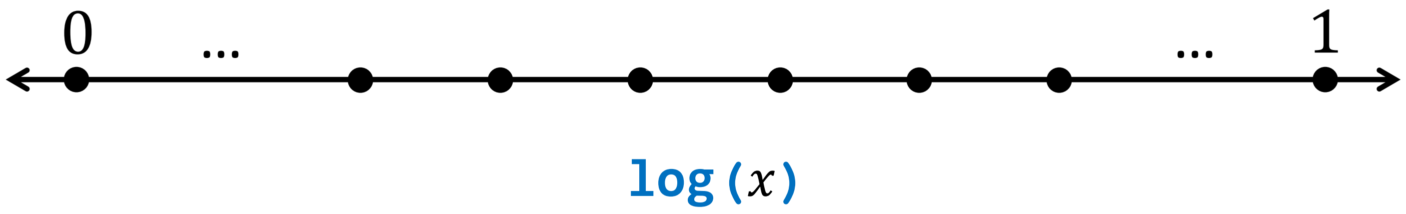
Challenge 2: Highly Accurate Implementations

- Example: `log` has precision loss of < 1 ulp iff
  for any $x \in \boldsymbol{X}$,

Challenge 2: Highly Accurate Implementations

- Example: `log` has precision loss of < 1 ulp iff

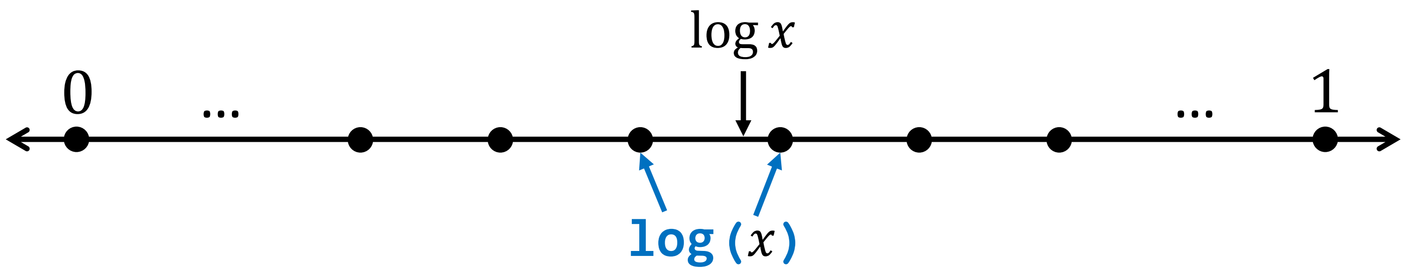
  for any $x \in \boldsymbol{X}$,



Challenge 2: Highly Accurate Implementations

- Example: `log` has precision loss of < 1 ulp iff

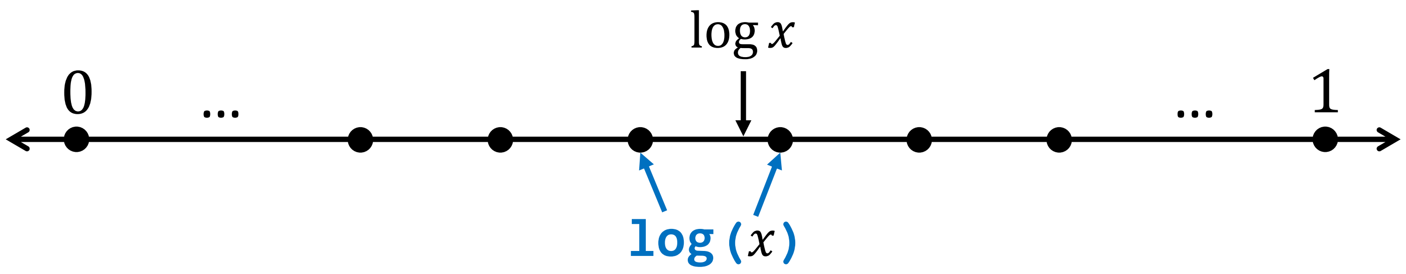
  for any $x \in \boldsymbol{X}$,



Challenge 2: Highly Accurate Implementations

- Example: **`log`** has precision loss of < 1 ulp iff

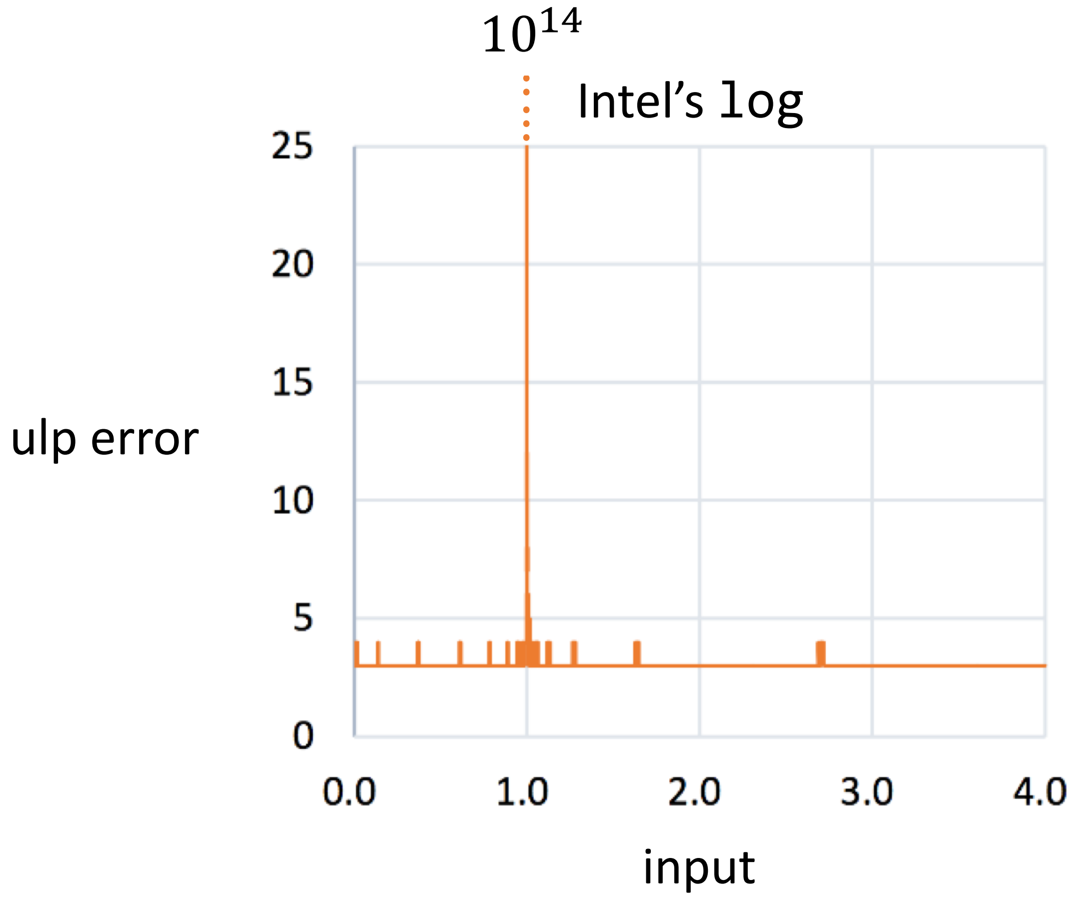
  for any $x \in \boldsymbol{X}$,



Why difficult to prove the 1 ulp error bound?

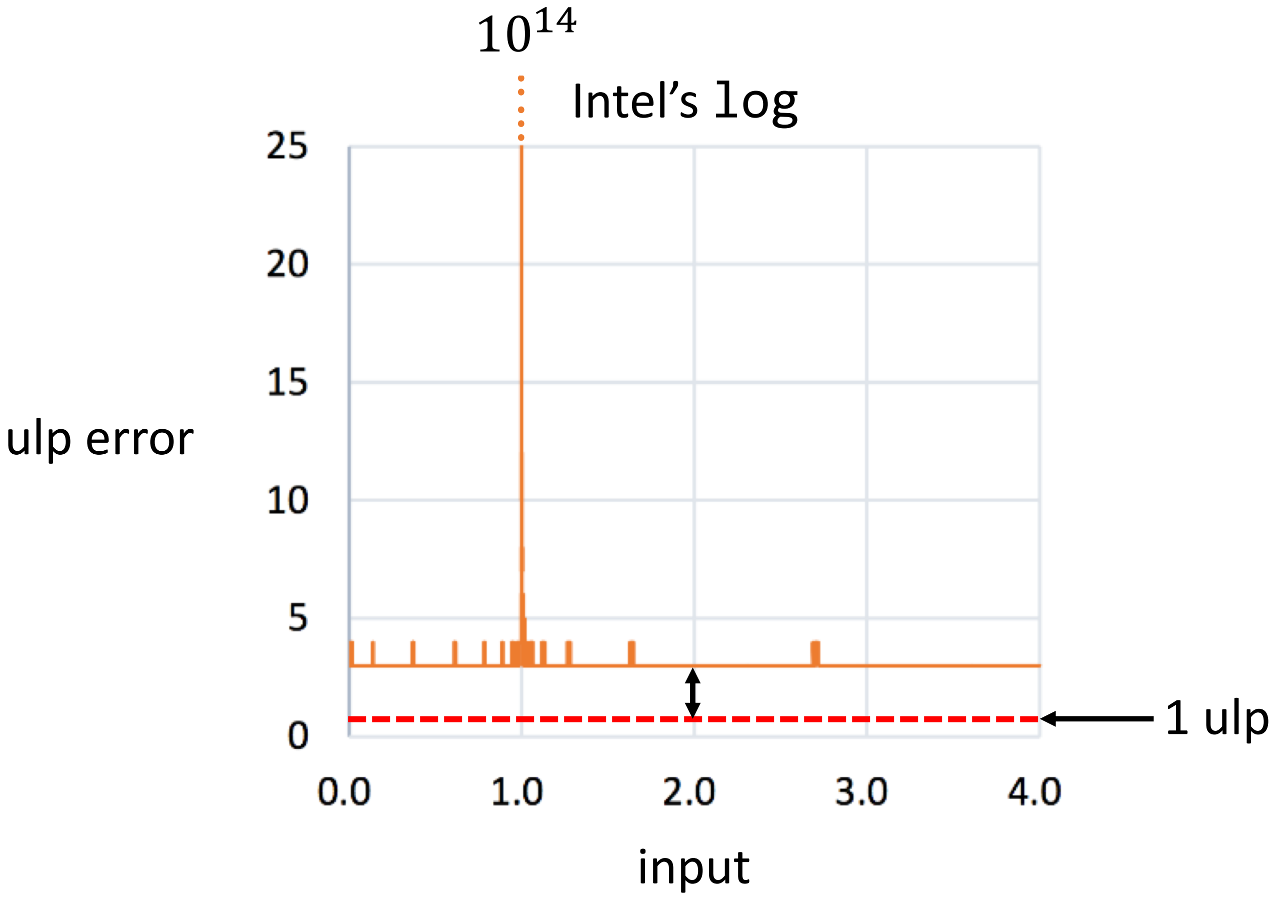
Challenge 2: Highly Accurate Implementations

- Error bounds from [PLDI’16].



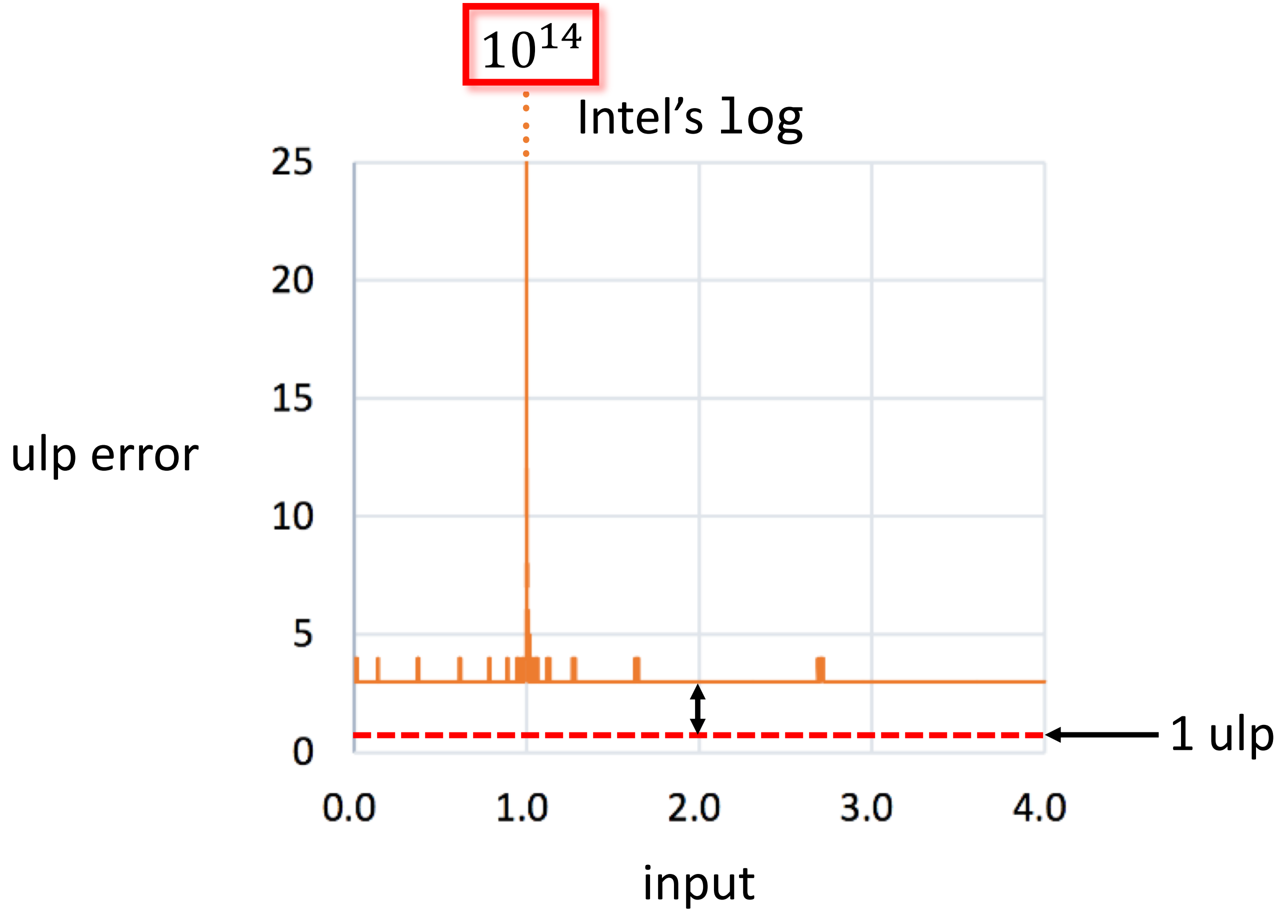
Challenge 2: Highly Accurate Implementations

- Error bounds from [PLDI'16].



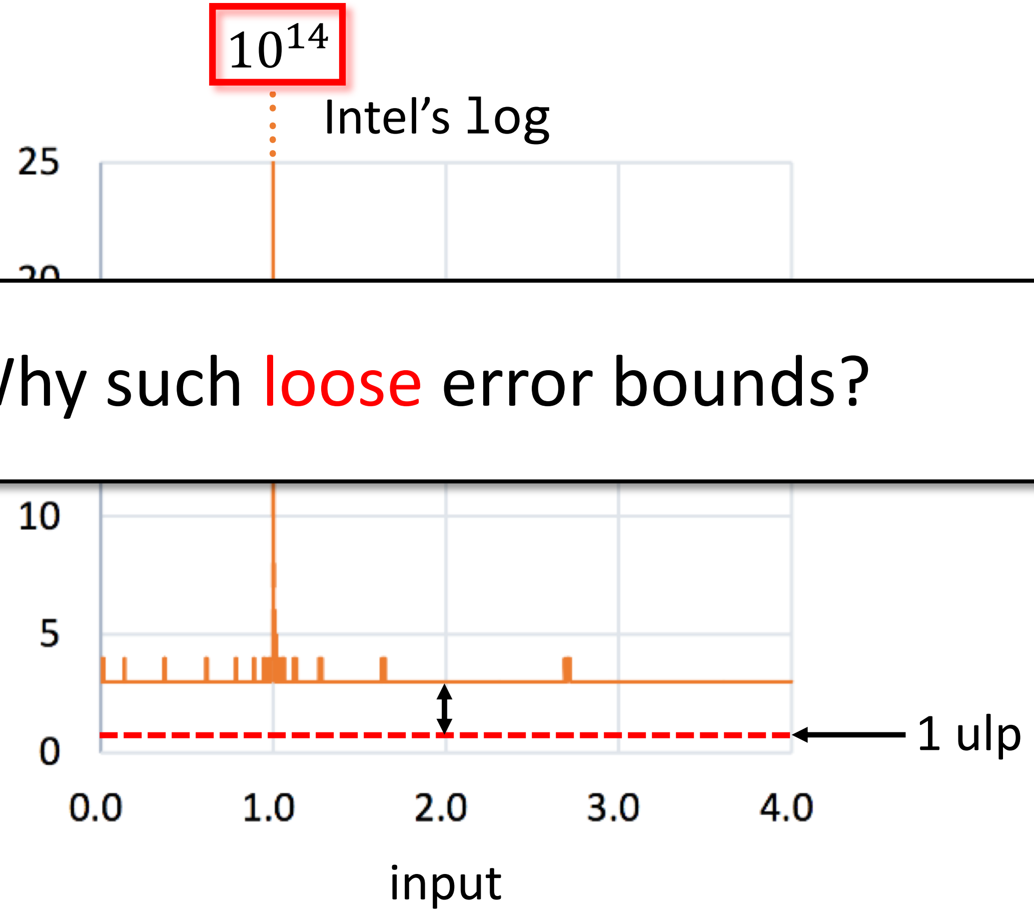
Challenge 2: Highly Accurate Implementations

- Error bounds from [PLDI'16].



Challenge 2: Highly Accurate Implementations

- Error bounds from [PLDI'16].



Why such loose error bounds?

Challenge 2: Highly Accurate Implementations

- Floating-point operations are sometimes exact.

$$a \otimes 2^n = a \times 2^n \quad \text{if } |a \times 2^n| \geq 2^{-1022}.$$

$$a \ominus b = a - b \quad \text{if } b/2 \leq a \leq 2b.$$

Challenge 2: Highly Accurate Implementations

- Floating-point operations are sometimes exact.

$$a \otimes 2^n = a \times 2^n \quad \text{if } |a \times 2^n| \geq 2^{-1022}.$$

$$a \ominus b = a - b \quad \text{if } b/2 \leq a \leq 2b.$$

- Standard error analysis ignores such exactness results, sometimes constructing imprecise abstractions.

Challenges

$\log x$

mathematical specification $\boldsymbol{f}$

```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

(1) $\boldsymbol{P}$ mixes floating-point and bit-level operations. [PLDI'16]

(2) $\boldsymbol{P}$ is claimed to have a precision loss of < 1 ulp.

Challenges

$$\log x$$

mathematical specification $\boldsymbol{f}$

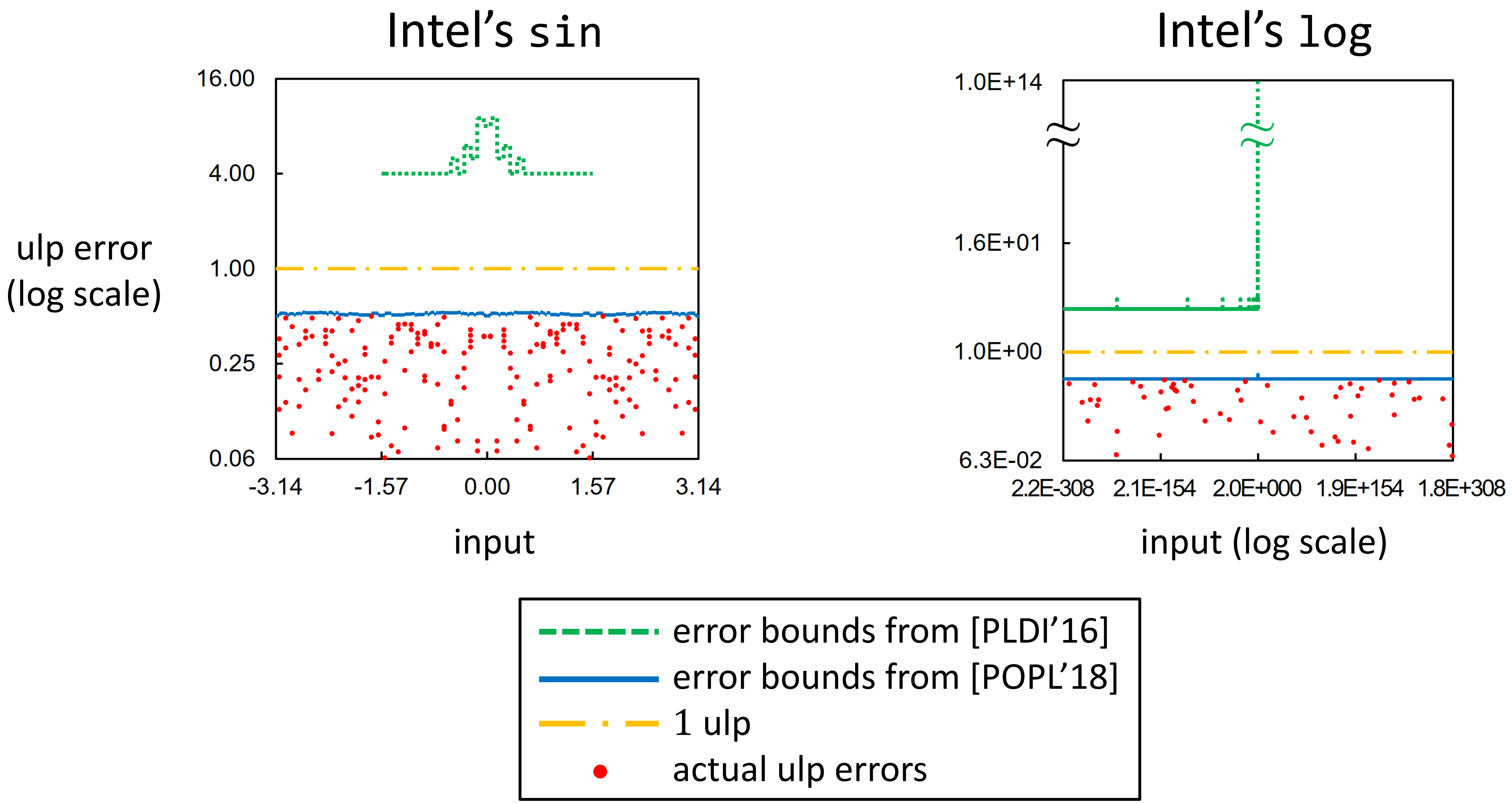
```
<log>
...
subsd  %xmm5, %xmm1
mulpd  %xmm0, %xmm5
...
psllq  $0xc,  %xmm1
pshufd $0xe4, %xmm5, %xmm6
...
```

`math.h` implementation $\boldsymbol{P}$

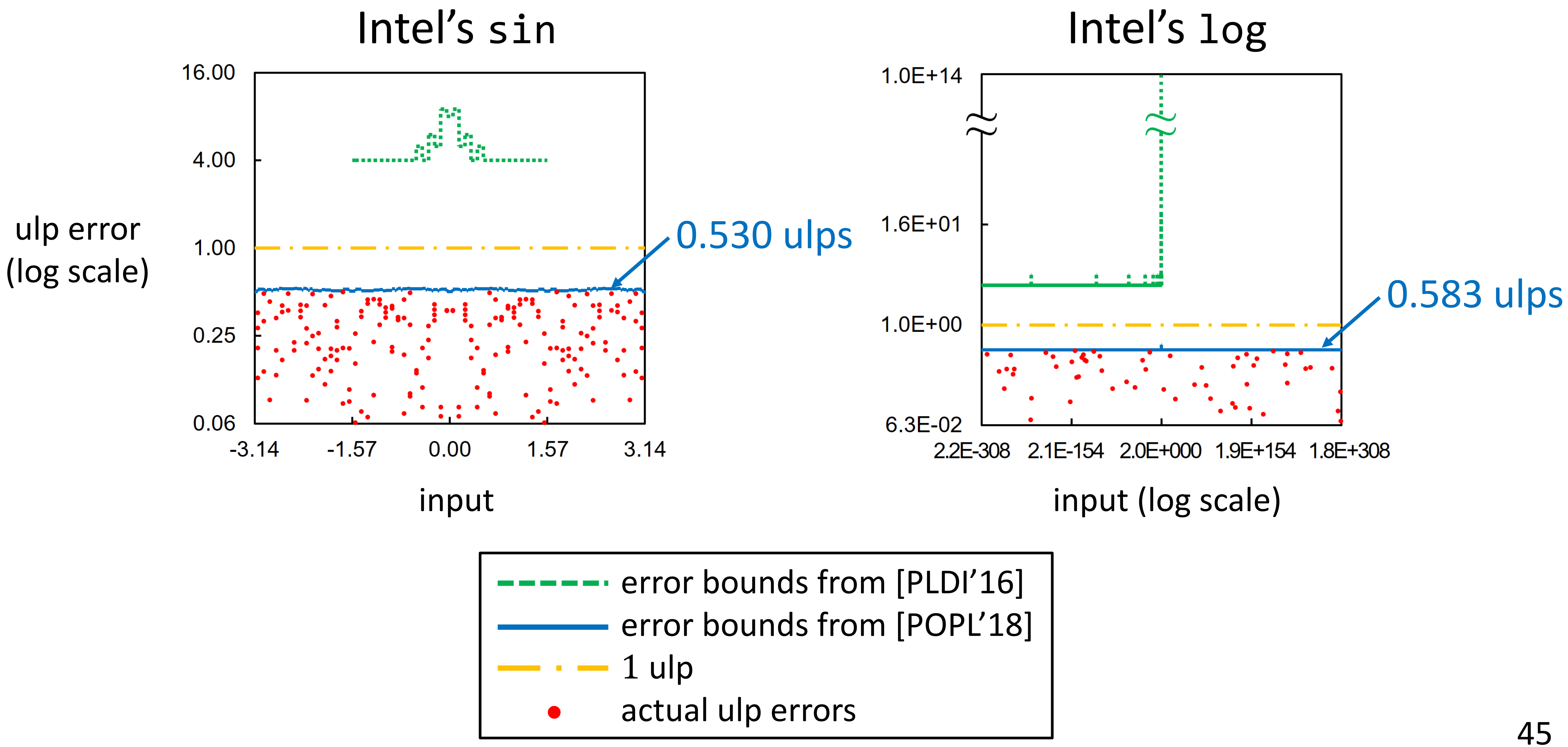
(1) $\boldsymbol{P}$ mixes floating-point and bit-level operations. [PLDI'16]

(2) $\boldsymbol{P}$ is claimed to have a precision loss of < 1 ulp. [POPL'18]

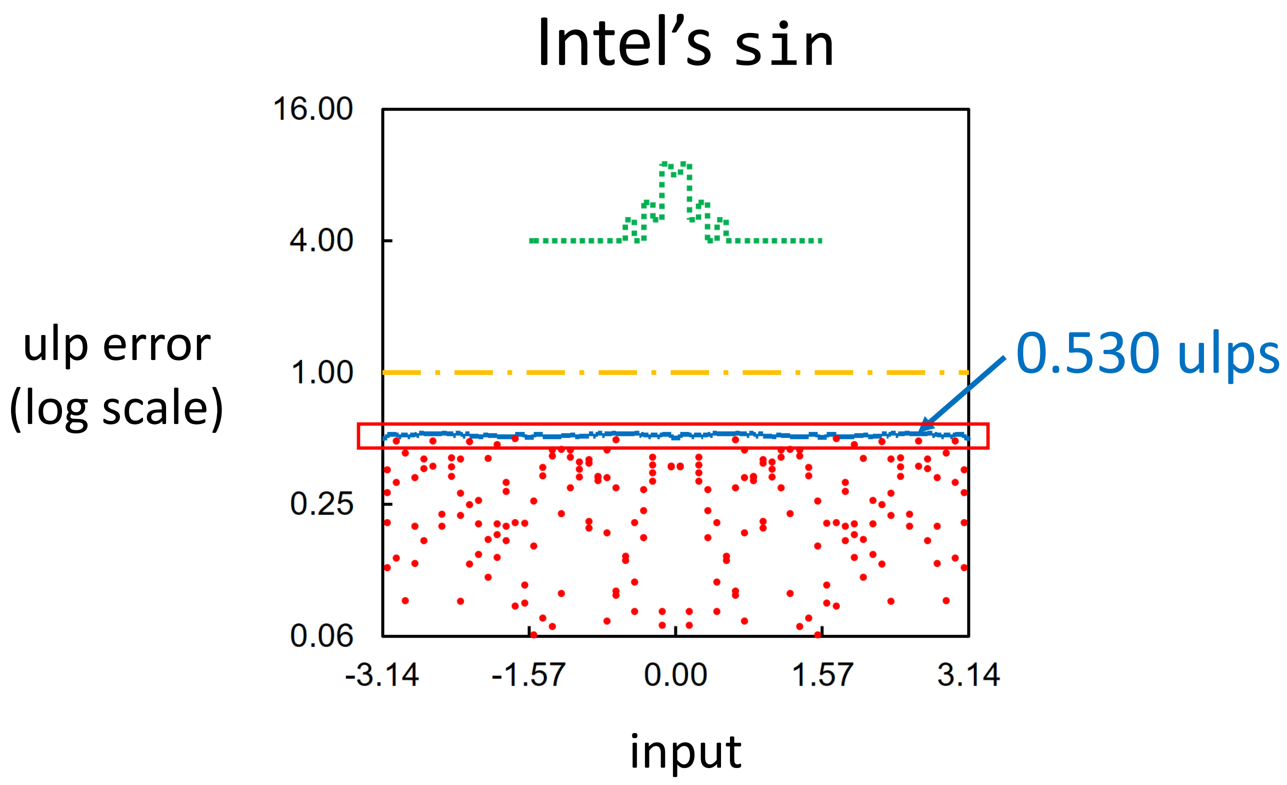
Results from [POPL'18]

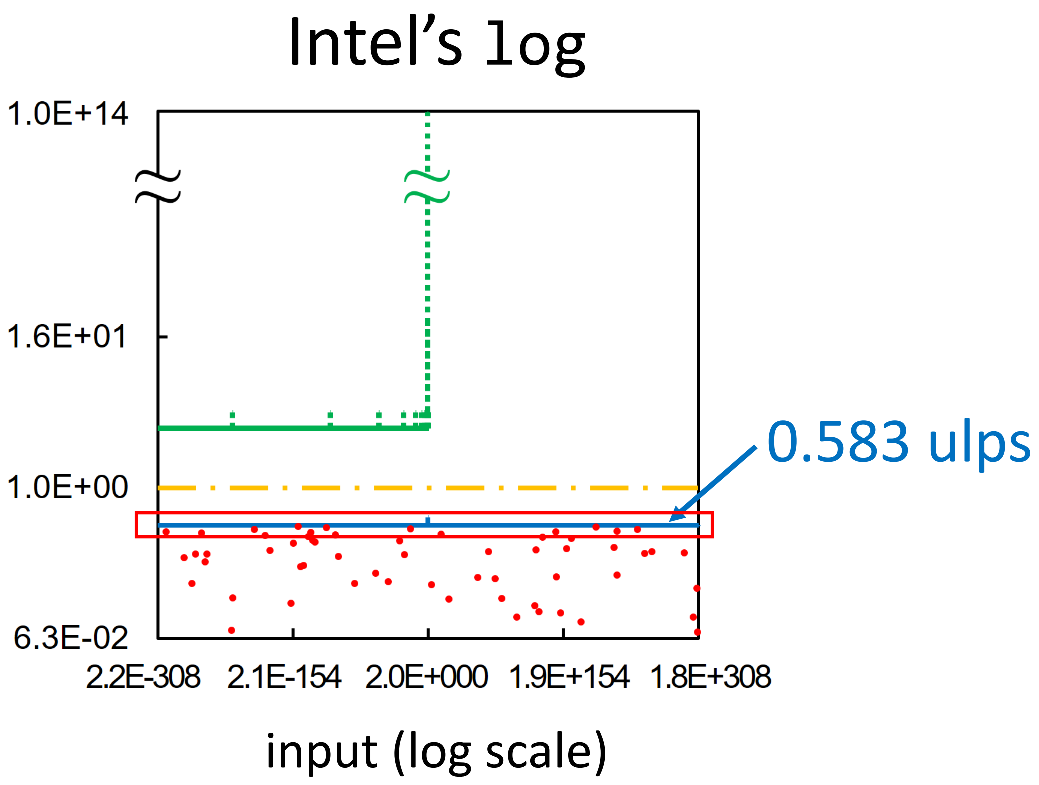


Results from [POPL'18]



Results from [POPL'18]





- error bounds from [PLDI'16]
- error bounds from [POPL'18]
- 1 ulp
- actual ulp errors

Limitations of [POPL'18]

For some cases, our approach

Limitations of [POPL'18]

For some cases, our approach

- may find loose error bounds.
  - `tan`: 13.33 ulps on $[17\pi/64,\ \pi/2)$.

Limitations of [POPL'18]

For some cases, our approach

- may find loose error bounds.
  - `tan`: 13.33 ulps on $[17\pi/64,\ \pi/2)$.
- may require considerable computation (though highly parallelizable).
  - `log`: 461 hours with 16 verifiers run in parallel.

Limitations of [POPL'18]

For some cases, our approach

- may find loose error bounds.
  - `tan`: 13.33 ulps on $[17\pi/64,\ \pi/2)$.
- may require considerable computation (though highly parallelizable).
  - `log`: 461 hours with 16 verifiers run in parallel.
- may not verify for all of intended inputs.
  - `sin`: verified for $x \in (-\pi, \pi)$; intended inputs are $x \in (-90112, 90112)$.

Limitations of [POPL'18]

For some cases, our approach

- may find loose error bounds.
  - `tan`: 13.33 ulps on $[17\pi/64, \ \pi/2)$.
- may require considerable computation (though highly parallelizable).
  - `log`: 461 hours with 16 verifiers run in parallel.
- may not verify for all of intended inputs.
  - `sin`: verified for $x \in (-\pi, \pi)$; intended inputs are $x \in (-90112, 90112)$.
- may become semi-automatic (though fully automatic for our benchmarks).
  - E.g., $\boldsymbol{P}$ computes $\text{round}(g(x))$ and g is non-linear.

Thank you!