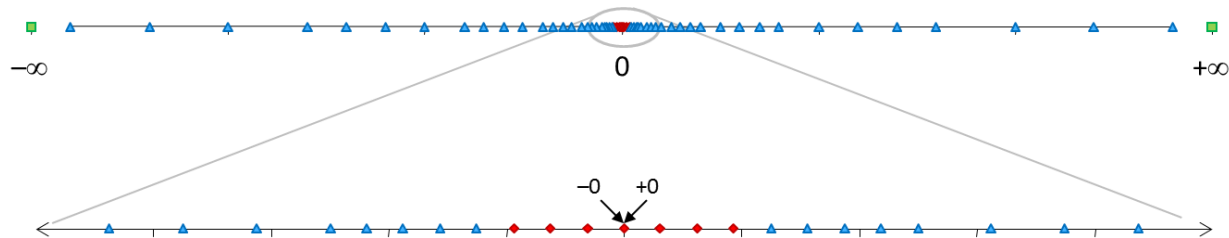


Data Representations

Floating Point Numbers



Module Outline

- Floating Point Representations
- Fixed-Point Encoding
- Floating-Point Encoding
- Module Summary

$$B2F(X) = \sum_{i=-n}^m 2^i * b_i$$

Floating Point Representations

Fractional Binary Numbers

■ Digital fractional numbers

d_6	d_5	d_4	d_3	.	d_2	d_1	d_0
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
1000	100	10	1		0.1	0.01	0.001

digits (0-9)

weight

■ Binary fractional numbers

b_6	b_5	b_4	b_3	.	b_2	b_1	b_0
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
8	4	2	1		1/2	1/4	1/8

bits (0-1)

weight

Encoding Fractional Binary Numbers

3	2	1	0		-1	-2	-3
b_6	b_5	b_4	b_3	.	b_2	b_1	b_0
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
8	4	2	1		1/2	1/4	1/8

- $m+1$ bits to the left of binary point, n bits to the right, total $m+n+1$ bits

$$B2F(X) = \sum_{i=-n}^m 2^i * b_i$$

Encoding Example

■ $f = 7.75$

3	2	1	0		-1	-2	-3
0	1	1	1	.	1	1	0
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
	4	2	1		1/2	1/4	

$= 7.75$

■ $f = 1.4$

3	2	1	0		-1	-2	-3
0	0	0	1	.	1	0	0
			1		1/2		

$= 1.5$

0	0	0	1	.	0	1	1
			1		1/4	1/8	

$= 1.375$

■ Issues: position of binary dot, rounding

IEEE 754: Floating Point Format

■ 32-bit floating point



■ 64-bit floating point



■ Encoding

$$V = (-1)^s \times M \times 2^E$$

- Sign encoded directly by s: 0: positive, 1: negative
- Exponent E encoded by exp: 8/11-bit biased unsigned integer
- Significand M encoded by frac: 23/52-bit fractional binary number

IEEE 754: Floating Point Encodings

■ Normalized Values

- when $\text{exp} \notin \{00\dots0, 11\dots1\}$
- $E = \text{exp} - \text{Bias}$ 32 bit: $\text{Bias} = 127$, 64 bit: $\text{Bias} = 1023$
range of E : $-126\dots+127$ (32 bit) and $-1022\dots+1023$ (64-bit)
- $M = 1.\text{frac}$
- Example: $v = -4.875$

1		10000001		001110000000000000000000
-		129		1.00111

$$\begin{aligned} v &= -1 \times 1.00111 \times 2^{129-127} = -1 \times 1.00111 \times 2^2 \\ &= -1 \times 100.111 = -1 \times 4.875 = -4.875 \end{aligned}$$

IEEE 754: Floating Point Encodings

■ Denormalized Values: represent 0 and values very close to 0

- when $\text{exp} = 00\dots 0$

- $E = 1 - \text{Bias}$ 32 bit: $\text{Bias} = 127$, 64 bit: $\text{Bias} = 1023$

$$E = -126 \text{ (32 bit)} / -1022 \text{ (64 bit)}$$

- $M = 0.\text{frac}$

- Example: smallest positive normalized value vs biggest denormalized value:

$$0 \mid 00000001 \mid 000000000000000000000000 = 1 \times 2^{-126}$$

$$0 \mid 00000000 \mid 111111111111111111111111 = 0.11\dots 1 \times 2^{-126}$$

IEEE 754: Floating Point Encodings

■ Special Values: represent $\pm\infty$ and $\pm\text{NaN}$ (Not A Number)

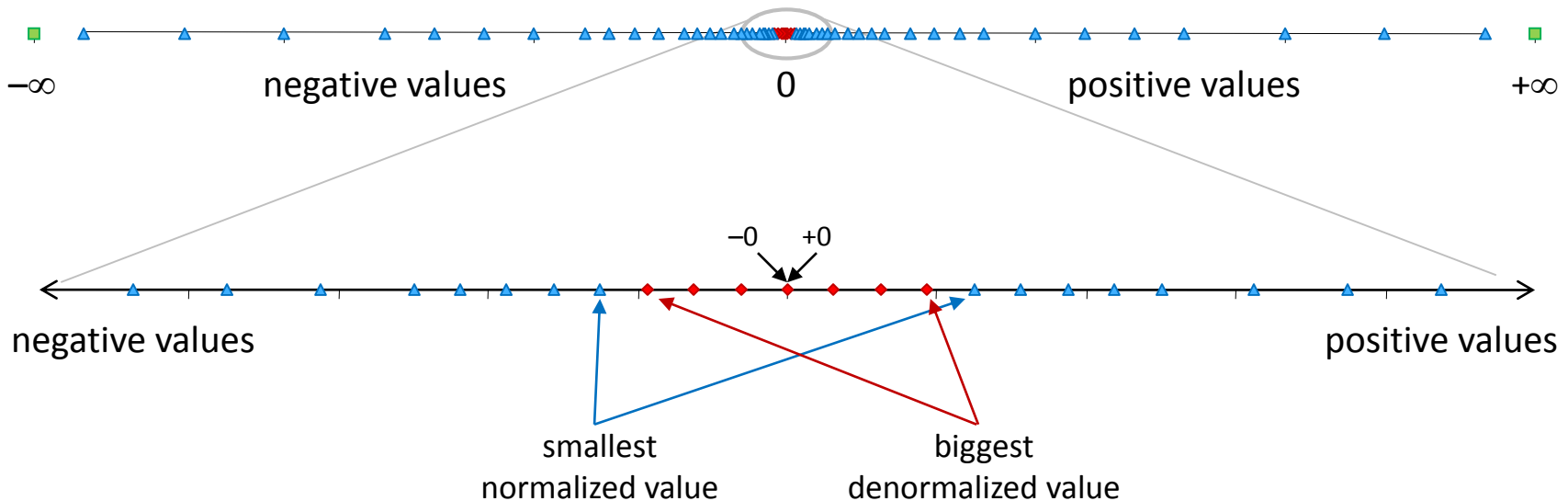
- when $\text{exp} = 11\dots 1$
- infinity when $\text{frac} = 00\dots 0$: $\pm\infty$, depending on the sign s
- not a number when $\text{frac} \neq 00\dots 0$
- Example : $v = +\text{infinity}$:

$$0 \mid 11111111 \mid 00000000000000000000000000000000 = +\infty$$

IEEE 754: Floating Point Distribution

■ Distribution of number range

◆ Denormalized ▲ Normalized ■ Infinity



- *equal distribution* between denormalized values around 0
- *increasing gap* for normalized values the farther away from 0
 - ▶ gap doubles for each increment in the exponent

Tiny FP Example

- 8-bit floating point representation
 - The sign bit is in the most significant bit
 - The next four bits are the **exp** with a bias of 7
 - The last three bits are the **frac**
- Same general form as IEEE format
 - Normalized, denormalized
 - Representation of 0, NaN, infinity



Tiny FP Example

- Values related to the exponent ($Bias = 7$)

Description	exp_2	exp_{10}	$E = exp - Bias$	2^E
Denormalized	0000	0	-6	1/64
Normalized	0001	1	-6	1/64
	0010	2	-5	1/32
	0011	3	-4	1/16
	0100	4	-3	1/8
	0101	5	-2	1/4
	0110	6	-1	1/2
	0111	7	0	1
	1000	8	1	2
	1001	9	2	4
	1010	10	3	8
	1011	11	4	16
	1100	12	5	32
	1101	13	6	64
	1110	14	7	128
inf, NaN	1111	15	-	-

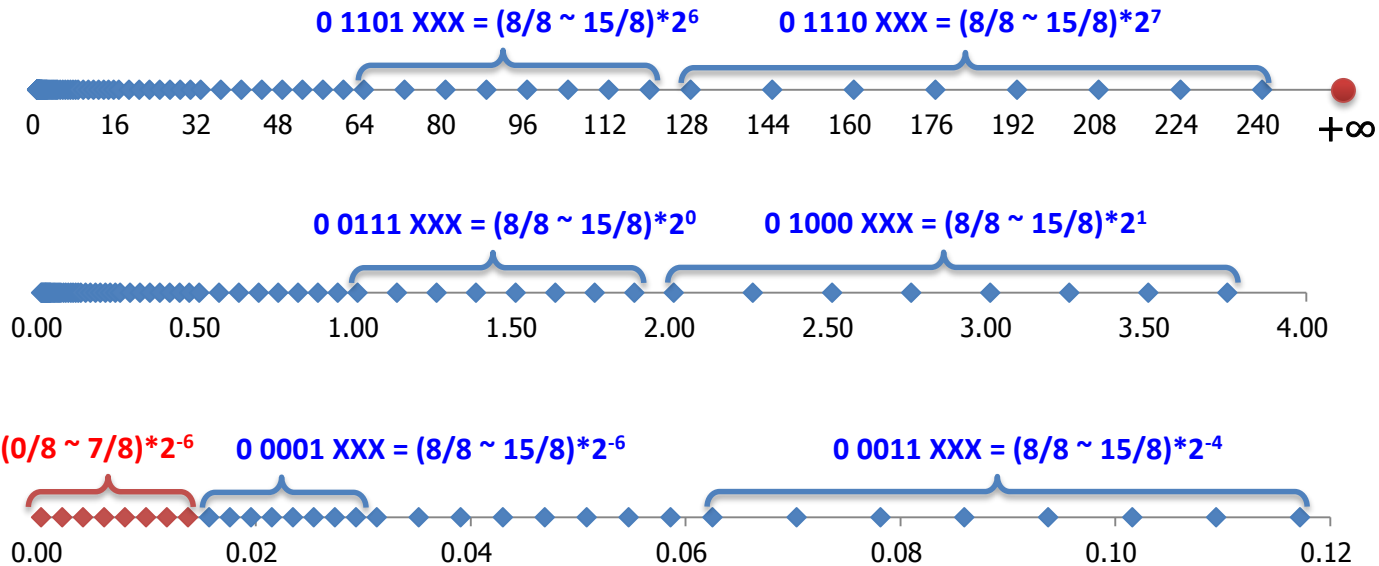
Tiny FP Example

■ Positive dynamic range

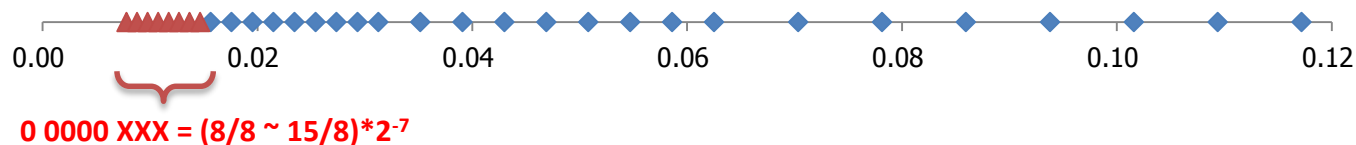
	Description	Bit representation	exp	E	frac	M	value
Denormalized	Zero	0 0000 000	0	-6	0	0	0
	Smallest possible value	0 0000 001	0	-6	1/8	1/8	1/512
		0 0000 010	0	-6	2/8	2/8	2/512
		0 0000 011	0	-6	3/8	3/8	3/512
		0 0000 110	0	-6	6/8	6/8	6/512
	Largest denormalized value	0 0000 111	0	-6	7/8	7/8	7/512
Normalized	Smallest normalized value	0 0001 000	1	-6	0	8/8	8/512
		0 0001 001	1	-6	1/8	9/8	9/512
		0 0110 110	6	-1	6/8	14/8	14/16
		0 0110 111	6	-1	7/8	15/8	15/16
	One	0 0111 000	7	0	0	8/8	1
		0 0111 001	7	0	1/8	9/8	9/8
		0 0111 010	7	0	2/8	10/8	10/8
		0 1110 110	14	7	6/8	14/8	224
	Largest normalized value	0 1110 111	14	7	7/8	15/8	240
Special	Infinity	0 1111 000	-	-	-	-	$+\infty$

Tiny FP Example

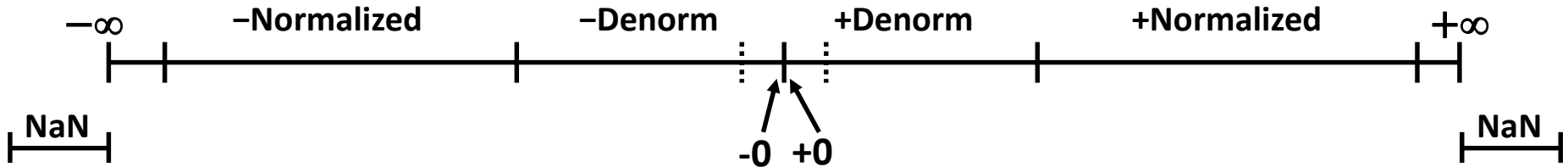
■ Encoded values (nonnegative numbers only)



● without denormalization



Interesting Numbers

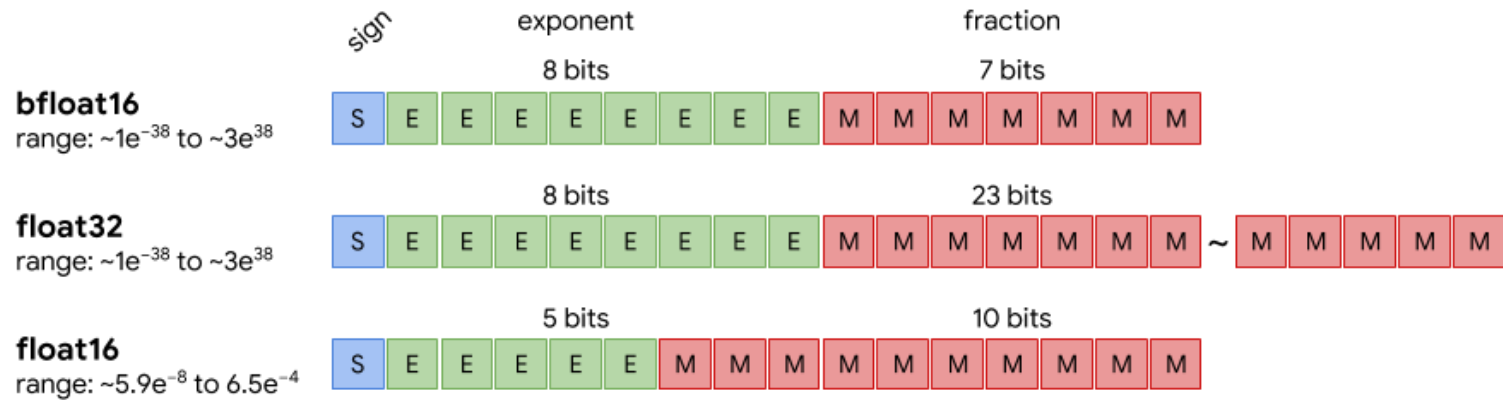


Description	exp	frac	Numeric Value
Zero	000 ... 00	000 ... 00	0.0
Smallest denormalized value	000 ... 00	000 ... 01	Single: $2^{-23} \times 2^{-126} \approx 1.4 \times 10^{-45}$ Double: $2^{-52} \times 2^{-1022} \approx 4.9 \times 10^{-324}$
Largest denormalized value	000 ... 00	111 ... 11	Single: $(1.0 - \epsilon) \times 2^{-126} \approx 1.18 \times 10^{-38}$ Double: $(1.0 - \epsilon) \times 2^{-1022} \approx 2.2 \times 10^{-308}$
Smallest normalized value	000 ... 01	000 ... 00	Single: 1.0×2^{-126} , Double: 1.0×2^{-1022} (Just a bit larger than largest denormalized)
One	011 ... 11	000 ... 00	1.0
Largest normalized value	111 ... 10	111 ... 11	Single: $(2.0 - \epsilon) \times 2^{127} \approx 3.4 \times 10^{38}$ Double: $(2.0 - \epsilon) \times 2^{1023} \approx 1.8 \times 10^{308}$

Special Properties

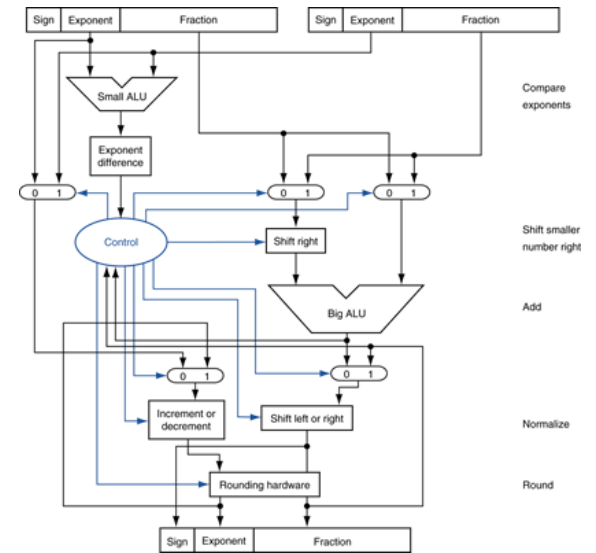
- FP zero same as integer zero
 - All bits = 0
- Can (almost) use unsigned integer comparison
 - Must first compare sign bits
 - Must consider $-0 = 0$
 - NaNs problematic
 - ▶ Will be greater than any other values
 - Otherwise OK
 - ▶ Denormalized vs. normalized
 - ▶ Normalized vs. Infinity

IEEE FP16 vs. Google bfloat16



■ Google bfloat16

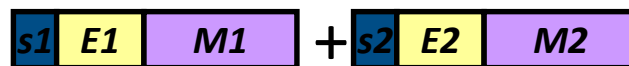
- Introduced by Google in 2018 for TPUs (Supported by Intel NPUs too)
- Same dynamic range as FP32
- Smaller mantissa reduces power and physical silicon area



Floating Point Operations

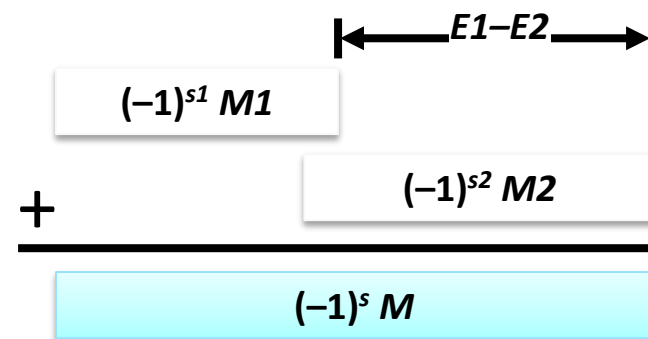
Floating Point Addition

Assume $E1 > E2$



1. Align binary points

- ▶ Shift number with smallest exponent
- ▶ Shift right $M2$ by $E1 - E2$



2. Add significands

- ▶ Result: Sign s , Significand M , Exponent $E (= E1)$

3. Normalize result & check for over/underflow

- ▶ if $(M \geq 2)$, shift M right, increment E
- ▶ if $(M < 1)$, shift M left k positions, decrement E by k

4. Round M and renormalize if necessary

Floating Point Addition Example

■ Compute 2.75 + 1.25

$2.75 = 0 \ 10000000 \ 011000000000000000000000 + 128 \ 1.0110000...0$
 $1.25 = 0 \ 01111111 \ 010000000000000000000000 + 127 \ 1.0100000...0$

1. Align: shift right mantissa of 1.25 by 128-127=1

$$\begin{array}{r} 1 \quad 1.011000000000000000000000 \\ 1 \quad + \quad 0.101000000000000000000000 \end{array}$$

$$\begin{array}{r} 1.011 \times 2^1 \\ 0.101 \times 2^1 \\ \hline = 1.000 \times 2^1 \end{array}$$

2. Add significands (signed two's complement format)

$$\begin{array}{r} 1 \quad 01.011000000000000000000000 \\ 1 \quad + \quad 00.011000000000000000000000 \\ \hline 1 \quad 10.000000000000000000000000 \end{array}$$

$$\begin{array}{r} = 1.0 \times 2^2 \\ \Rightarrow 01000000 \ 1000... \end{array}$$

3. Normalize result

$$2 \quad 1.000000000000000000000000$$

4. Encode result

$4.00 = 0 \ 10000001 \ 000000000000000000000000 + 129 \ 1.0000000...0$

Floating Point Addition Example

■ Compute 2.75 – 1.25

$$\begin{array}{rcl} 2.75 & = & 0 \ 10000000 \ 011000000000000000000000 + 128 \ 1.0110000...0 \\ -1.25 & = & 1 \ 01111111 \ 010000000000000000000000 + 127 \ 1.0100000...0 \end{array}$$

1. Align: shift right mantissa of 1.25 by 128-128=1

$$\begin{array}{r} 1 \quad 1.011000000000000000000000 \\ 1 \quad - \quad 0.101000000000000000000000 \end{array}$$

$$\begin{array}{r} 1.011 \times 2^1 \\ - 0.101 \times 2^1 \\ \hline 0.110 \times 2^1 \end{array}$$

$$= 1.10 \times 2^0$$

2. Add significands (signed two's complement format)

$$\begin{array}{r} 1 \quad 01.011000000000000000000000 \\ 1 \quad + \quad 11.011000000000000000000000 \\ \hline 1 \quad 100.110000000000000000000000 \end{array}$$

$$\Rightarrow 00111111 \ 11000000 \dots$$

3. Normalize result

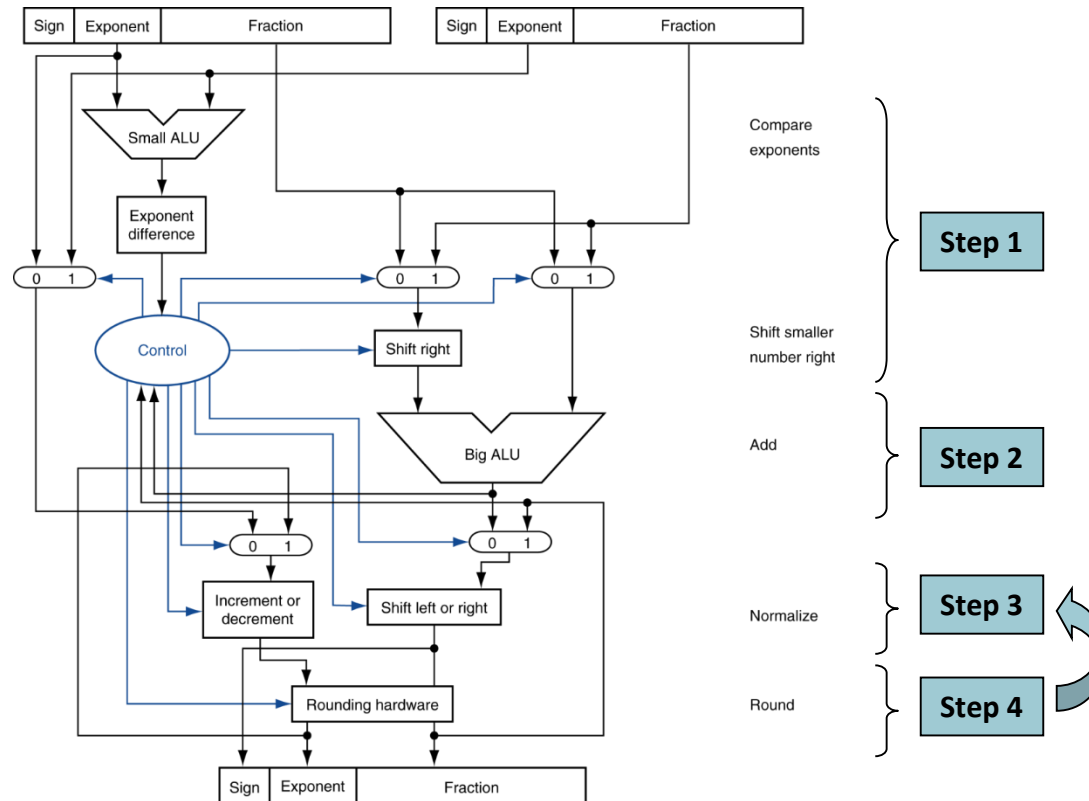
$$0 \quad 1.100000000000000000000000$$

4. Encode result

$$1.50 = 0 \ 01111111 \ 100000000000000000000000 + 127 \ 1.1000000...0$$

Floating Point Adder Hardware

- Significantly more complex than integer adder



Floating Point Multiplication

1. Add exponents

- ▶ $E = E1 + E2$



2. Multiply significands

- ▶ $M = M1 \times M2$

3. Normalize result & check for over/underflow

- ▶ if ($M \geq 2$), shift M right, increment E
- ▶ if ($M < 1$), shift M left k positions, decrement E by k

4. Round M and renormalize if necessary

5. Determine sign

- ▶ $s = s1 \wedge s2$

Floating Point Multiplication Example

■ Compute $2.75 * -1.25$

$2.75 = 0 \ 10000000 \ 011000000000000000000000$ + 128 1.0110000...0
 $-1.25 = 1 \ 01111111 \ 010000000000000000000000$ - 127 1.0100000...0

1. Add exponents

$$\begin{array}{r} 1 \\ + 0 \\ \hline 1 \end{array}$$

2. Multiply mantissas

$$\begin{array}{r} 1.011000000000000000000000 \\ \times 1.010000000000000000000000 \\ \hline 1.101110000000000000000000000 \end{array}$$

3. Normalize the product (nothing to do)

1 1.101110000000000000000000

4. Determine sign

$$\begin{array}{r} 0 \\ \wedge 1 \\ \hline 1 \end{array}$$

5. Encode result

$-3.4375 = 1 \ 10000000 \ 101110000000000000000000$ + 128 1.1011100...0

Sign = $1 \wedge 0 = 1$

1.011×2^1
 1.01×2^0

$\therefore 1 \wedge 0 = 1$
 $\therefore 1.101110 \times 2^1$
 $\Rightarrow 01000000 \ 01011100$

$$\begin{array}{r} 1011 \\ 1010 \\ \hline 0000 \\ 1011 \\ \hline 0000 \\ 1011 \\ \hline 1.101110 \end{array}$$

IEEE 754: Rounding

- **Problem:** for a given value x , finding the “closest” matching value x' that can be represented in floating point format
- IEEE 754 rounding modes

Mode	1.40	1.50	1.60	2.50	-1.50
Round-to-even (default)	1	2	2	2	-2
Round-toward-zero	1	1	1	2	-1
Round-down	1	1	1	2	-2
Round-up	2	2	2	3	-1

- Round-to-even (aka round-to-nearest) gives closest result: $\hat{x} \cong x$
 - avoids statistical bias by rounding upward or downward so that the least significant digit is even
- Round-toward-zero: bound on absolute value: $|\hat{x}| \cong |x|$
- Round-down/up can be used to establish a lower/upper-bound of the error $x^- \leq x \leq x^+$

Rounding

- Hardware maintains additional bits to the right of the LSB of the significand
 - Uses these extra bits to round
 - To round, in principle, we need two extra bits: **guard** and **round**

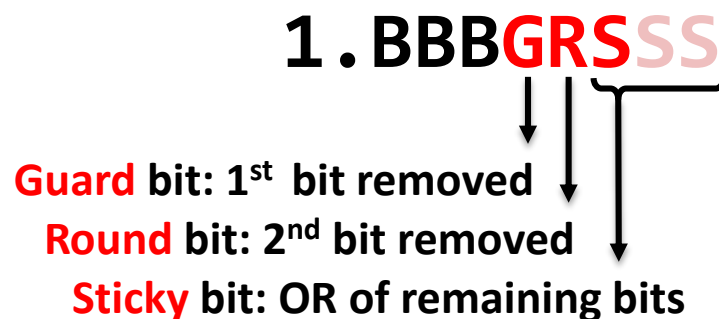


Internal value	GR	Round up	Round down	Round to even
1.010 00	00	1.010	1.010	1.010
1.010 01	01	1.010	1.010	1.010
1.010 10	10	1.011	1.010	1.010
1.001 10	10	1.010	1.001	1.010
1.001 11	11	1.010	1.010	1.010

→ The guard bit has weight $\frac{1}{2}\text{ulp}$, the round bit is used to break ties when $G=1$.

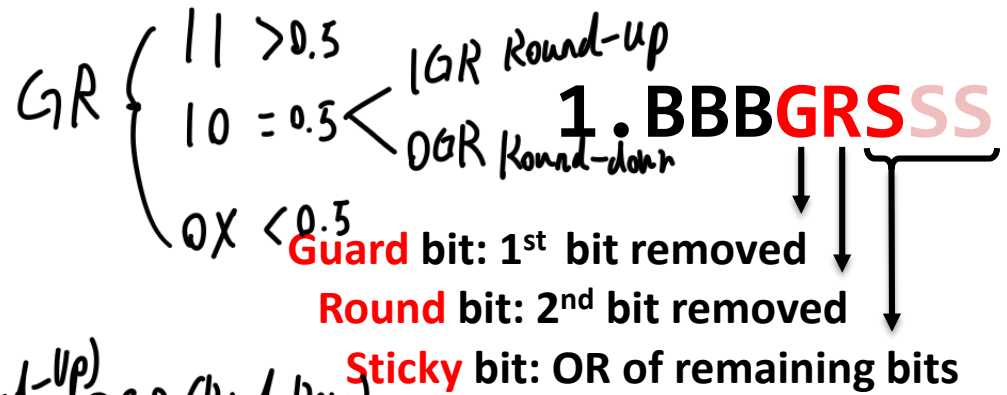
Rounding

- The guard and round bit alone allow us to accurately round up and towards zero, but not down and to even
 - Computations may produce a leading 0 that requires normalization
 - This shift left by 1 operation shifts the guard bit into the LSB and the round bit into the guard bit position
 - We can no longer use the round bit as a tie breaker, i.e., it is impossible to distinguish between $=\frac{1}{2}\text{ulp}$ (10 00...0) and $\geq\frac{1}{2}\text{ulp}$ (10 00...1)!
 - Solution: use a **sticky** bit that remembers whether any of the shifted out bits were 1



Sticky bit: 1 if *any* of the bits shifted through it were 1, 0 otherwise

GRS Rounding



1.010 100

IGR (Round-Up) OGR (Round-Down)

Result	Normalized	GR	Round up	Round down	Round to even
0.101 010	1.010 100	10	1.011	1.010	1.010
0.101 110	1.011 100	10	1.100	1.011	1.100
0.101 111	1.011 110	11	1.100	1.100	1.100
0.101 011	1.010 110	11	1.011	1.011	1.011

GR > 0.5 ∴ Round-up

- Observe what happens if we only had guard and round:

Result	Normalized	GR	Round up	Round down	Round to even
0.101 01	1.010 10	10	1.011	1.010	1.010
0.101 11	1.011 10	10	1.100	1.011	1.100
0.101 11	1.011 10	10	1.100	1.011	1.011
0.101 01	1.010 00	00	1.010	1.010	1.010

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
 - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
 - Addition, subtraction, multiplication, division, reciprocal, square-root, ...
 - FP \leftrightarrow integer conversion
- Completing an operation in one clock cycle would take too long
 - Much longer than integer operations
 - Slower clock would penalize all instructions
- Operations usually take several cycles
 - Can be pipelined

Floating Points in C

- C guarantees two levels
 - **float** (single precision) vs. **double** (double precision)
- Conversions
 - **double or float → int**
 - ▶ Truncates fractional part
 - ▶ Like rounding toward zero
 - ▶ Not defined when out of range or NaN (Generally sets to TMin)
 - **int → double**
 - ▶ Exact conversion, as long as int has ≤ 53 bit word size
 - **int → float**
 - ▶ Will round according to rounding mode

Intel Pentium Floating-Point Bug

- To speed up the (slow) shift-and-subtract division algorithm behind the FDIV instruction, Intel implemented the SRT (Sweeney, Robertson, Tocher) algorithm in the Pentium processor
 - Generates result 2 bits per cycle
 - Relies on a lookup table with 2,048 cells.
- During the manufacturing process of the Pentium, five values were not downloaded correctly into the chip manufacturing machine, resulting in the entries being 0 when they should have contained the value +2.
- The error was discovered on June 13, 1994 by Thomas Nicely, professor of mathematics at Lynchburg College. Reported the problem to Intel on October 24, 1994. Intel decided to keep things quiet to avoid a recall.
- Nicely, not so nicely, send information about the bug to various mailing lists.

Intel Pentium Floating-Point Bug

- Intel tried to fix the problem with software patches, but was eventually forced to issue a recall and replace all faulty chips
- The recall cost Intel \$475 million in 1994
 - Net profit in '94: \$2.6 billion
- Led to wider adaptation of formal verification of hardware

From November 7, 1994: *Electronic Engineering Times* (a trade publication)

Intel fixes A Pentium FPU glitch

By Alexander Wolfe

Santa Clara, Calif. - To correct an anomaly that caused inaccurate results on some high-precision calculations, Intel Corp. last week confirmed that it had updated the floating-point unit (FPU) in the Pentium microprocessor

The company said that the glitch was discovered midyear and was fixed with a mask change in recent silicon. "This was a very rare condition that happened once every 9 to 10 billion operand pairs," said Steve Smith, a Pentium engineering manager at Intel.

A spot check last week indicated the problem is present in at least one recently made Pentium-based PC. Intel said it could not quantify how many such systems were in the field.

Said an Intel spokesman: "This doesn't even qualify as an errata. We fixed it in a subsequent stepping."

Erroneous division

The issue came to light last week in a message, on Compuserve's "Canopus" forum, which was a reposting of a private e-mail communication from Lynchburg College (Lynchburg, Va.) mathematics professor Thomas Nicely. "The Pentium floating-point unit is

all digits
which
ant



Cleve's Corner

A Tale of Two Numbers

With the Pentium, there is a very small chance of making a very large error

by Cleve Moler

This is the tale of two numbers, and how they found their way over the Internet to the front pages of the world's newspapers on Thanksgiving Day, embarrassing the world's premier computer chip manufacturer. The first number is the 12-digit integer

$$p = 824633702441$$

Thomas Nicely of Lynchburg College in Virginia is interested in this number because both p and $p+2$ are prime. Two consecutive odd numbers that are both prime are called *twin primes*. Nicely's research involves the distribution of twin primes and, in particular, the sum of their reciprocals,

$$S = 1/5 + 1/7 + \dots + 1/29 + 1/31 + \dots + 1/p + 1/(p+2) + \dots$$

It is known that this infinite sum has a finite value, but nobody

transcendental functions. Within hours of receiving Wolfe's query, Mathisen confirmed Nicely's example, wrote a short, assembly-language test program, and on November 3, initiated a chain of Internet postings in the newsgroup *comp.sys.intel* about the FDIV bug. (FDIV is the floating-point divide instruction on the Pentium.) A day later, Andreas Kaiser of Germany posted a list of two dozen numbers whose reciprocals are computed to only single-precision accuracy on the Pentium.

On November 7, the *EE Times* published the first news article on the bug. The headline on the front page story by Wolfe was "Intel Fixes a Pentium FPU Glitch".

Tim Coe is an engineer at Vitesse Semiconductor in Southern California. He has designed floating-point arithmetic units himself and saw in Kaiser's list of erroneous reciprocals clues to how the Pentium designers had tackled the same task. He surmised, correctly it turns out, that the Pentium's division instruction employs a technique known to chip designers as a



"In June, Nicely noticed

Source: Electronic Engineering Times

Ariane 5 Failure

- On June 4, 1996, the maiden flight of the new Ariane 5 ended with the complete destruction of the rocket
 - Exploded 37 seconds after liftoff
 - On board: satellites worth \$500 million



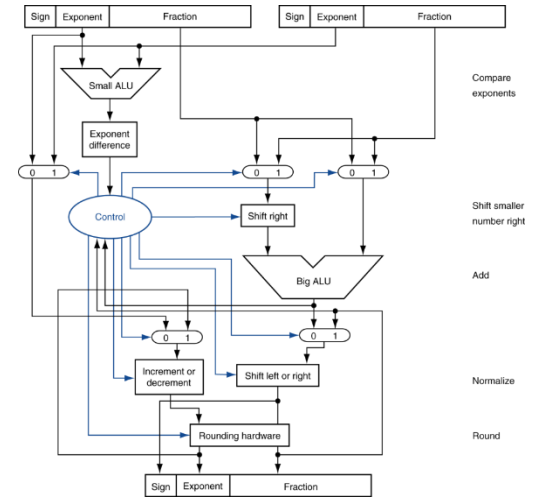
Ariane 5 Failure

- Reason for the failure?
 - Computed horizontal velocity as floating-point number
 - Converted to 16-bit integer
 - ▶ Careful analysis of Ariane 4 trajectory proved 16-bit is enough
 - Reused a module from 10-year-old software
 - ▶ Overflowed for Ariane 5
 - ▶ No precise specification for the software

- https://youtu.be/gp_D8r-2hwk



Module Summary



Module Summary

■ Representations

- representation of floating point numbers

■ Floating point operations

- rounding
- basic operation of
 - ▶ addition, subtraction
 - ▶ multiplication

■ Errors in the FP H/W implementation can be costly!