

Computer Organization and Architecture

COSC 2425

Lecture – 11

Sept 26th, 2022

Acknowledgement: Slides from Edgar Gabriel & Kevin Long

Chapter 3

Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
 - Addition and subtraction
 - Dealing with overflow
 - Multiplication and division

Adding 64-bit numbers

$$\begin{array}{r} 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{\text{two}} = 7_{\text{ten}} \\ +\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{\text{two}} = 6_{\text{ten}} \\ \hline =\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001101_{\text{two}} = 13_{\text{ten}} \end{array}$$

Subtraction

- Subtracting 6_{ten} from 7_{ten} directly

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten}$

[illegible]

- Subtracting 6_{ten} from 7_{ten} using two's complement.

$$7 + (-6)$$

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten}$$

$$+ \quad 11111111 \ 11111111 \ 11111111 \ 11111111 \ 11111111 \ 11111111 \ 11111010_{\text{two}} = -6_{\text{ten}}$$

$$= \underbrace{00000000}_{\text{billions}} \underbrace{00000000}_{\text{millions}} \underbrace{00000000}_{\text{thousands}} \underbrace{00000000}_{\text{hundreds}} \underbrace{00000000}_{\text{tens}} \underbrace{00000000}_{\text{ones}} 1_{\text{two}} = 1_{\text{ten}}$$

Overflow

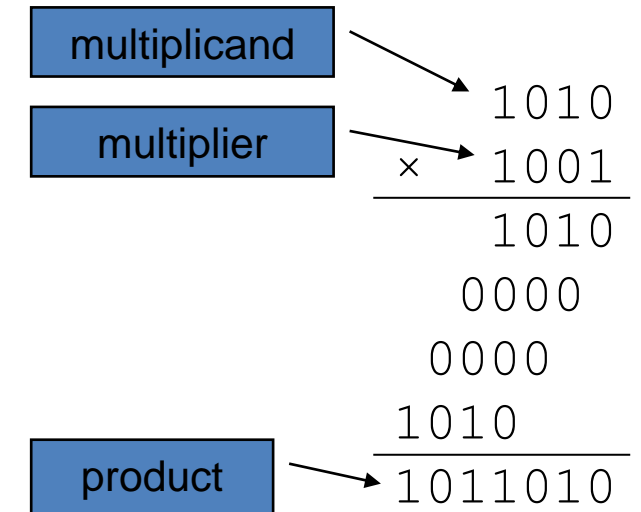
- Signed integers, addition
- When can an overflow occur?

Operand 1	Operand 2	Overflow	Check
+ve	-ve	No	
-ve	+ve	No	
+ve	+ve	Yes	-ve result
-ve	-ve	Yes	+ve result

Multiplication

- Start with long-multiplication approach

Multiplication of binary number is similar to decimal system



Multiplication Hardware

• Iteration 4

- Check if multiplier bit is 0/1
 - If 1 add multiplicand to product register
 - Else do nothing
- Shift multiplicand to left by 1 bit
- Shift multiplier 1 bit to the right
- Repeat **4 times (total)**
- Exit

$$\begin{array}{r}
 1010 \\
 \times 1001 \\
 \hline
 1010 \\
 0000 \\
 0000 \\
 1010 \\
 \hline
 1011010
 \end{array}$$

Multiplicand register
0101 0000

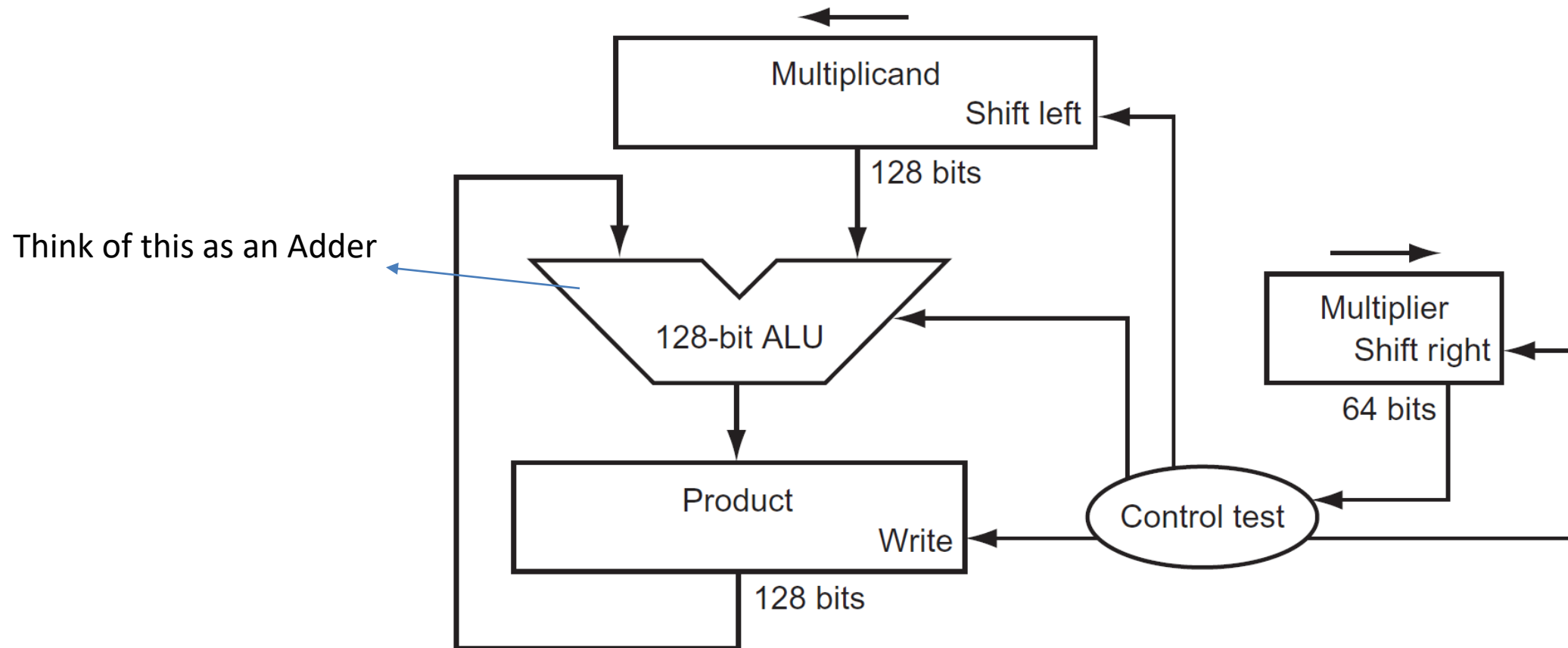
Shift
left

Multiplier register
0001

Shift
right

Product register
0101 1010


Multiplication Hardware



Signed Multiplication

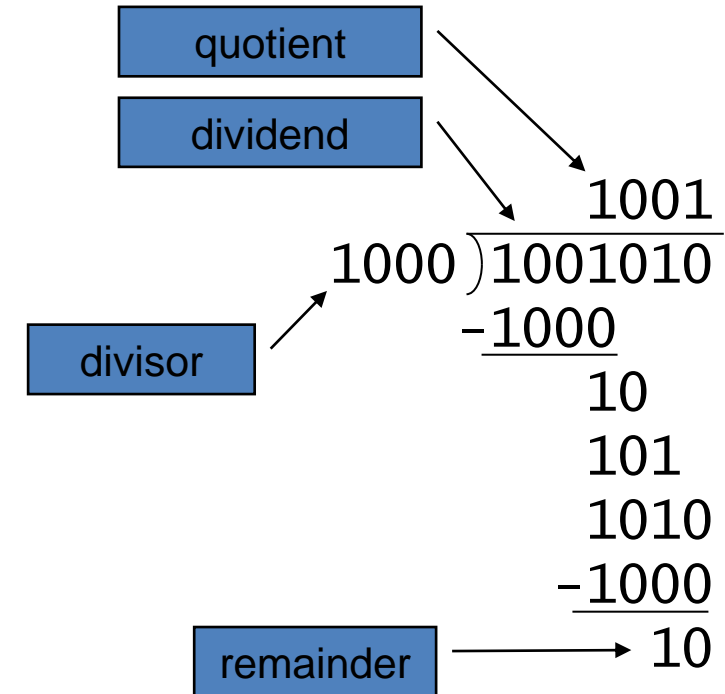
- Convert to Multiplicand and Multiplier to positive and remember the sign

Multiplicand	Multiplier	Result
-ve	+ve	-ve
+ve	-ve	-ve
+ve	+ve	+ve
-ve	-ve	+ve



If multiplicand and multiplier signs disagree, then the result is negative.

Division



n-bit operands yield *n*-bit
quotient and remainder

Division Hardware

1. Remainder = Remainder - Divisor

1. If remainder < 0,
 1. Shift quotient to left, and add 0 to end
 2. Add the remainder back to divisor, and restore value

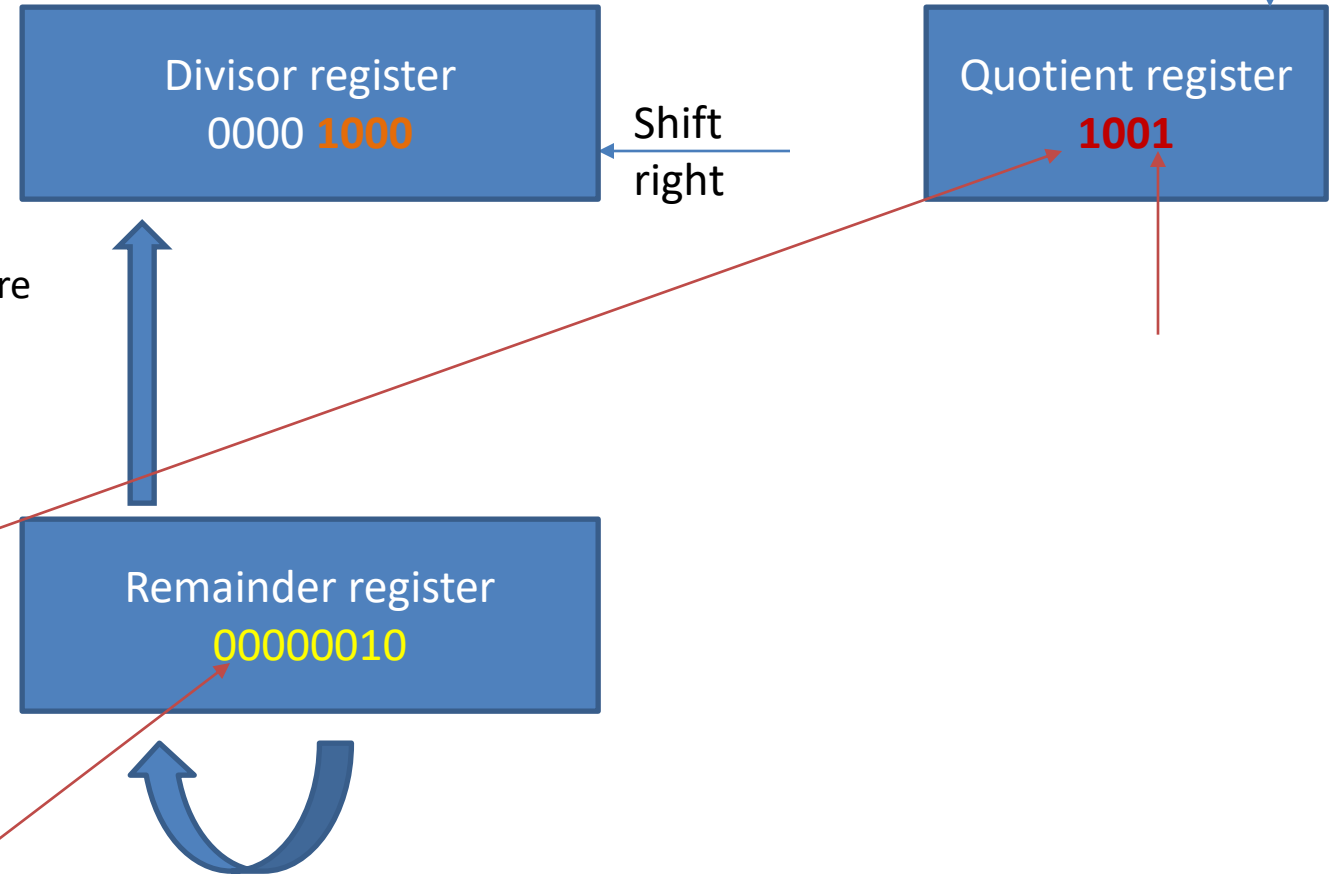
2. If remainder > 0,

1. Shift quotient to left, and add 1 to end

2. Shift Divisor to the right by 1 bit

3. Repeat 5 times

1001
 1000 $\overline{)1001010}$
 -1000
 10
 101
 1010
 -1000
 10



Instructions

Type	Name
Arithmetic	ADD, SUB, MUL
Data transfer	LDUR, STUR
Arithmetic Immediate	ADDI, SUBI, ORRI, ANDI, EORI, MUL, SMULH, UMULH, SDIV, UDIV
Logical Operations	LSL, LSR, AND, ORR, EOR
Branches	B, CBZ, CBNZ, B.Cond
Set Condition Flag	ADDS, ADDIS, SUBS, SUBIS, ANDS, ANDIS

Floating Point Numbers

- Support number for fractions, “*reals*” in mathematics.
- Example

$3.14159265\dots_{\text{ten}}$ (pi)

$2.71828\dots_{\text{ten}}$ (*e*)

0.000000001_{ten} or $1.0_{\text{ten}} \times 10^{-9}$ (seconds in a nanosecond)

$3,155,760,000_{\text{ten}}$ or $3.15576_{\text{ten}} \times 10^9$ (seconds in a typical century)

Floating Point Numbers

- Support number for fractions, “*reals*” in mathematics.
- Example

$3.14159265\dots_{\text{ten}}$ (pi)

$2.71828\dots_{\text{ten}}$ (*e*)

0.000000001_{ten} or $\boxed{1.0_{\text{ten}} \times 10^{-9}}$ (seconds in a nanosecond)

$3,155,760,000_{\text{ten}}$ or $\boxed{3.15576_{\text{ten}} \times 10^9}$ (seconds in a typical century)

Normalized Scientific notation: Single digit before decimal, and no leading zeros.

Binary Floating Point Numbers

$$\begin{aligned} 23_{ten} &= 10111_{two} \\ &\Rightarrow 1011.1_{two} \times 2^1 \end{aligned}$$

Binary Floating Point Numbers

$$\begin{aligned}23_{ten} &= 10111_{two} \\ \Rightarrow 1011.1_{two} \times 2^1 &= 11.5 \times 2 = 23 \\ \Rightarrow 101.11_{two} \times 2^2 &= 5.75 \times 4 = 23 \\ \Rightarrow 1.0111_{two} \times 2^4 &= 1.4375 \times 16 = 23\end{aligned}$$

Normalized Scientific notation

Binary Floating-Point Numbers

$$\begin{aligned}23_{ten} &= 10111_{two} \\ \Rightarrow 1011.1_{two} \times 2^1 &= 11.5 \times 2 = 23 \\ \Rightarrow 101.11_{two} \times 2^2 &= 5.75 \times 4 = 23 \\ \Rightarrow 1.0111_{two} \times 2^4 &= 1.4375 \times 16 = 23\end{aligned}$$

Normalized Scientific notation

In binary, general form

$$\pm 1.xxxxxxx_{two} \times 2^{yyyy}$$

Binary Floating-Point Numbers

$$\begin{aligned}
 23_{ten} &= 10111_{two} \\
 \Rightarrow 1011.1_{two} \times 2^1 &= 11.5 \times 2 = 23 \\
 \Rightarrow 101.11_{two} \times 2^2 &= 5.75 \times 4 = 23 \\
 \Rightarrow 1.0111_{two} \times 2^4 &= 1.4375 \times 16 = 23
 \end{aligned}$$

Normalized Scientific notation

In binary, general form

$$\pm 1.\underbrace{\text{xxxxxxxx}}_{\text{Fraction}}_{two} \times 2^{\underbrace{\text{yyyy}}_{\text{Exponent}}}$$

Floating-Point Representation

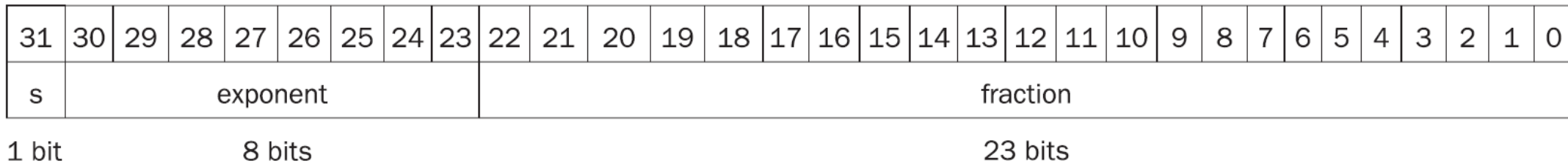
- If n number of bits are available to represent floating point numbers,
 - Allocate sign bit
 - Allocate x bits to represent exponents
 - Allocate $(n - 1) - x$ to represent fractions

Floating-Point Representation

- If n number of bits are available to represent floating point numbers,
 - Allocate sign bit (1-bit)
 - Represents $\pm ve$ numbers
 - Allocate x bits to represent exponents
 - Increasing x improves range
 - Much larger or smaller numbers can be represented
 - Allocate $(n - 1) - x$ to represent fractions
 - Decreasing x increases $(n - 1) - x$, improves precisions
 - Numbers can be represented more precisely

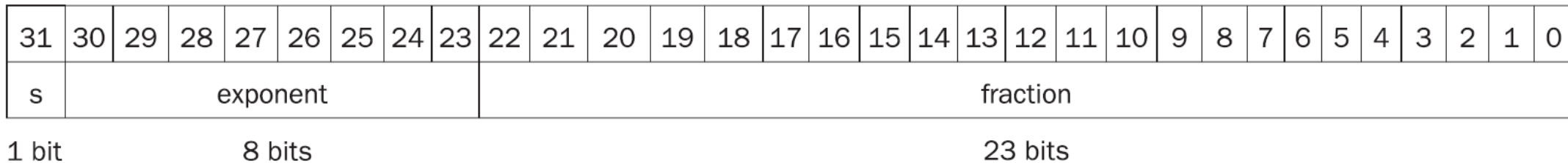
Floating-Point Numbers in LEGv8

- Two representations
 - Using 32-bits, single precision
 - 1-bit for sign
 - 8-bits for exponent
 - 23 bits for fraction



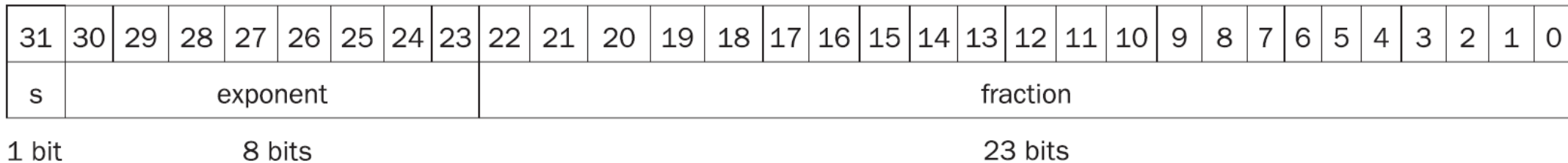
Floating-Point Numbers in LEGv8

- Two representations
 - Using 32-bits, single precision
 - 1-bit for sign
 - 8-bits for exponent
 - Can be both +ve and –ve (2's complement)
 - 23 bits for fraction



Floating-Point Numbers in LEGv8

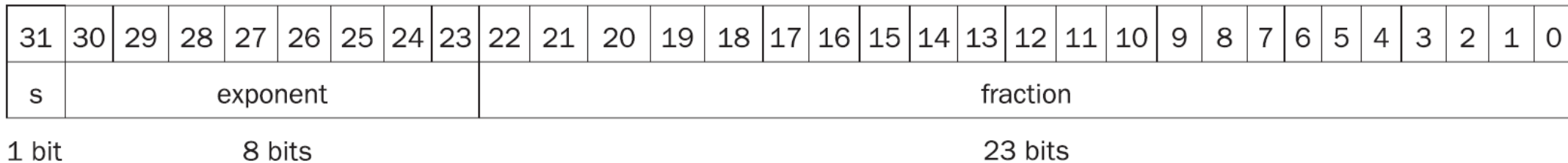
- Two representations
 - Using 32-bits
 - 1-bit for sign
 - 8-bits for exponent
 - Can be both +ve and –ve (2's compliment)
 - 23 bits for fraction



Floating-Point Numbers in LEGv8

- Two representations
 - Using 32-bits
 - 1-bit for sign
 - 8-bits for exponent
 - Can be both +ve and –ve (2's compliment)
 - 23 bits for fraction

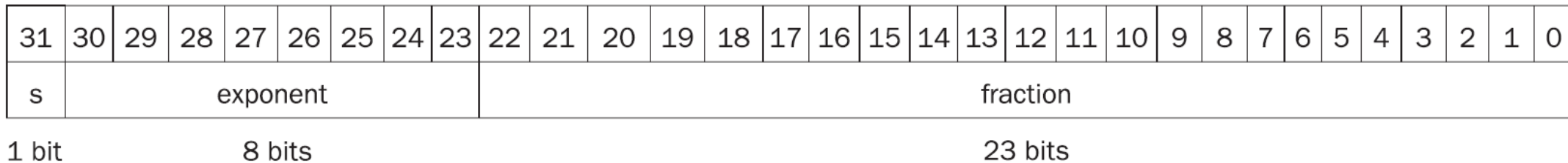
What is the smallest and largest positive number we can represent?



Floating-Point Numbers in LEGv8

- Two representations
 - Using 32-bits
 - 1-bit for sign
 - 8-bits for exponent
 - Can be both +ve and –ve (2's compliment)
 - 23 bits for fraction

What is the smallest and largest positive number we can represent?
 8 bits using two's compliment,
 We can represent from -128 to 127 (**excluding special reservations, more on this later!!!**)
 $\sim 1.0 \times 2^{-128}$ to $\sim 1.0 \times 2^{127}$



Floating Point-Numbers in LEGv8

- Two representations
 - Using 32-bits
 - 1-bit for sign
 - 8-bits for exponent
 - Can be both +ve and -ve (2's complement)
 - 23 bits for fraction

What is the smallest and largest positive number we can represent?

8 bits using two's compliment,

We can represent from -128 to 127 (**excluding special reservations, more on this later!!!**)

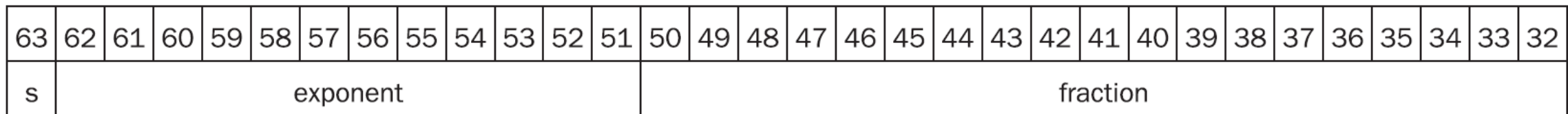
$$\sim 1.0 \times 2^{-128} \text{ to } \sim 1.0 \times 2^{127}$$

A horizontal number line with arrows at both ends. The left arrow points to the left and is labeled "underflow". The right arrow points to the right and is labeled "Overflow".

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

Floating Point-Numbers in LEGv8

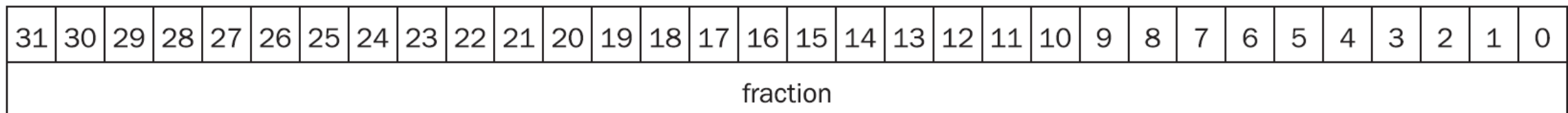
- Two representations
 - Using 32-bits, **single precision**
 - Using 64-bits, **double precision**
 - 1-bit for sign
 - 11-bits for exponent (larger range)
 - 52-bits for fraction (higher precision)



1 bit

11 bits

20 bits



32 bits

IEEE 754 Floating-Point Standard

- Not just on LEGv8
- Beginning from 1980's, most computers follow this standard.

$$1.0111_{two} \times 2^4$$



Can only be a zero or a one.

Is always a 1 in normalized scientific notations

The IEEE 754 make this implicit

The extra-bit can be used to represent fractional part (effective adds additional bit)

Reservation in Exponential filed

- Implicit assumption does not work for the number zeros,
 - 000...000 -> reserved to represent zeros.
- The largest number is reserved for special symbols $+\infty$, $-\infty$, *NAN*
- Single Precision
- **0 \Rightarrow 0000 0000**
- **255 \Rightarrow 1111 1111**
- **1 – 254 \Rightarrow 0000 0001 – 1111 1110**

Representing 0

- Implicit assumption does not work for the number zeros,
 - 000...000 -> reserved to represent zeros.
- The largest number is reserved for special symbols
 $+\infty, -\infty, \text{NaN}$

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Representing 0

- Implicit assumption does not work for the number zeros,
 - 000...000 -> reserved to represent zeros.
- The largest number is reserved for special symbols
 $+\infty, -\infty, \text{NaN}$

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Representing 0

- Implicit assumption does not work for the number zeros,
 - 000...000 -> reserved to represent zeros.
- The largest number is reserved for special symbols
 $+\infty, -\infty, \text{NaN}$

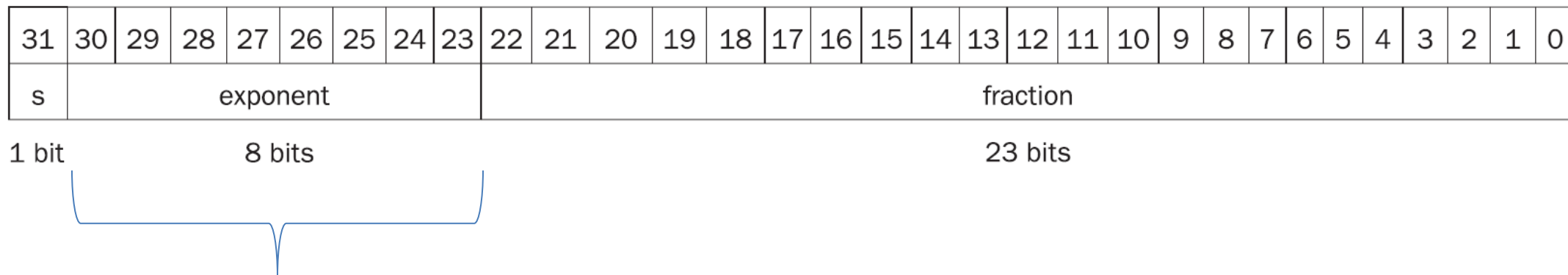
Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

11111111

11111111

Integer comparison on Floating Points.

- Let us consider only unsigned exponents
- Using integer comparison for ($<$, \leq , $>$, \geq) for floating point numbers



Exponent before fraction, makes sorting easier, as number with larger exponents are larger than those with smaller exponents.

Sorting, and integer comparisons can be directly used for sorting.

Integer comparison on Floating Points.

- Signed exponents are problematic
- Example: $1.0_{two} X 2^{-1}$.
- How should this number be represented?

Integer comparison on Floating Points.

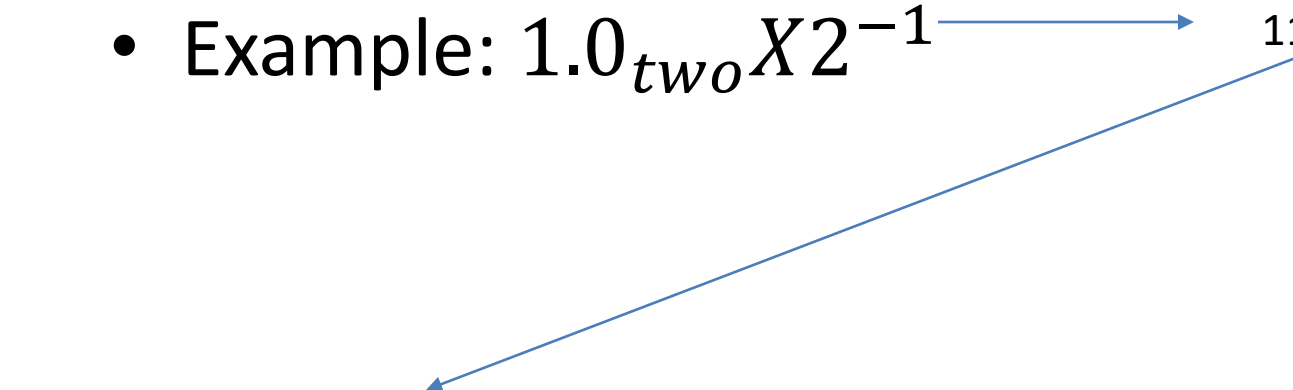
- Signed exponents are problematic
- Example: $1.0_{two} \times 2^{-1}$

Sign bit



Integer comparison on Floating Points.

- Signed exponents are problematic
- Example: $1.0_{two} \times 2^{-1} \longrightarrow 1111111$ (In 2's complement)



31	30	29	28	27	26	25	24	23
●	1	1	1	1	1	1	1	1

Integer comparison on Floating Points.

- Signed exponents are problematic
- Example: $1.0_{two} \times 2^{-1}$

Only need to represent fraction
 (000..00) fraction
 implicit

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
●	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Integer comparison on Floating Points.

- Signed exponents are problematic
- Example: $1.0_{two} \times 2^{-1}$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
●	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Consider $1.0_{two} \times 2^{+1}$

Integer comparison on Floating Points.

- Signed exponents are problematic
- Example: $1.0_{two} \times 2^{-1}$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
●	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Consider $1.0_{two} \times 2^{+1}$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Integer comparison on Floating Points.

- Signed exponents are problematic
- Example: $1.0_{two} \times 2^{-1}$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
●	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Integer comparison produce the result $1.0_{two} \times 2^{-1} > 1.0_{two} \times 2^{+1}$

- Consider $1.0_{two} \times 2^{+1}$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Integer comparison on Floating Points.

- Signed exponents are problematic
- Solution: Used biased representation
 - **Add a bias term** from exponent and use that instead of the exponent for representation.
 - IEEE 754 uses a bias of 127 for single precision representation

Integer comparison on Floating Points.

- Signed exponents are problematic
- Solution: Used biased representation
 - **Add a bias term** to exponent and use that instead of the exponent for representation.
 - IEEE 754 uses a bias of 127 for single precision representation

Number	Exponent (Without Bias)
$1.0_{two} \times 2^{-1}$	$-1 \Rightarrow \mathbf{0} \ 1111 \ 1111$
$1.0_{two} \times 2^{+1}$	$1 \Rightarrow \mathbf{0} \ 0000 \ 0001$

Sign bit

Integer comparison on Floating Points.

- Signed exponents are problematic
- Solution: Used biased representation
 - **Add a bias term** to exponent and use that instead of the exponent for representation.
 - IEEE 754 uses a bias of 127 for single precision representation

Number	Exponent (Without Bias)
$1.0_{two}X2^{-1}$	$-1 \Rightarrow \mathbf{0} \ 1111 \ 1111$
$1.0_{two}X2^{+1}$	$1 \Rightarrow \mathbf{0} \ 0000 \ 0001$

Sign bit

Number	Exponent (bias = 127)
$1.0_{two}X2^{-1}$	$-1 + \mathbf{127} \Rightarrow \mathbf{0} \ 0111 \ 1110$
$1.0_{two}X2^{+1}$	$1 + \mathbf{127} \Rightarrow \mathbf{0} \ 1000 \ 0000$

Integer comparison on Floating Points.

- Signed exponents are problematic
- Solution: Used biased representation
 - **Add a bias term** to exponent and use that instead of the exponent for representation.
 - IEEE 754 uses a bias of 127 for single precision representation

Number	Exponent (Without Bias)
$1.0_{two}X2^{-1}$	$-1 \Rightarrow \mathbf{0} \ 1111 \ 1111$
$1.0_{two}X2^{+1}$	$1 \Rightarrow \mathbf{0} \ 0000 \ 0001$

Sign bit

Number	Exponent (bias = 127)
$1.0_{two}X2^{-1}$	$-1 + \mathbf{127} \Rightarrow \mathbf{0} \ 0111 \ 1110$
$1.0_{two}X2^{+1}$	$1 + \mathbf{127} \Rightarrow \mathbf{0} \ 1000 \ 0000$

Integer comparison work

IEEE Floating-Point Format

single: 8 bits

single: 23 bits

double: 11 bits

double: 52 bits

S	Exponent	Fraction
---	----------	----------

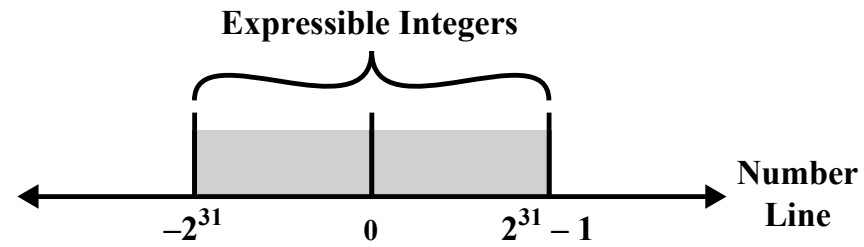
$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

Single-Precision Range

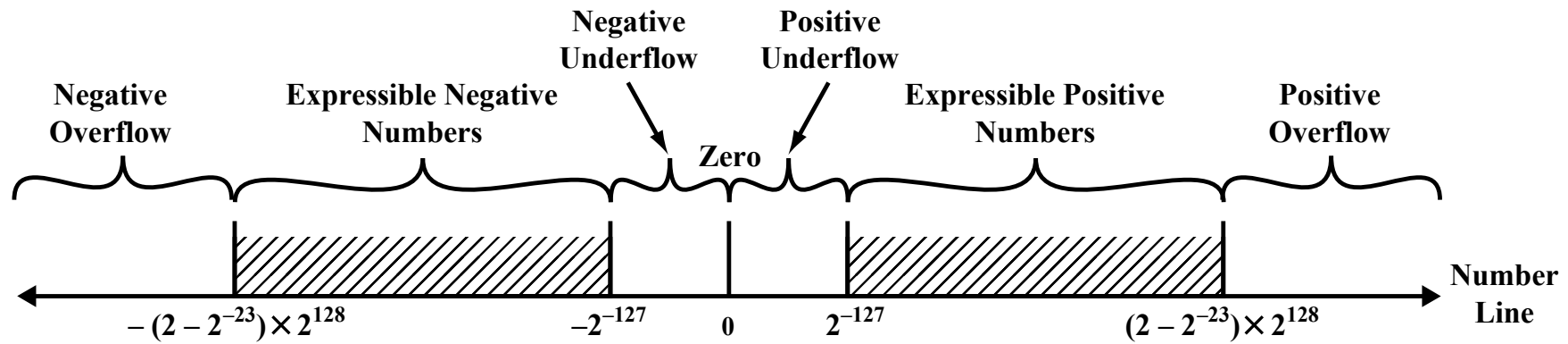
- Exponents 00000000 and 11111111 reserved
- Smallest value
 - Exponent: 00000001
 \Rightarrow actual exponent = $1 - 127 = -126$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
 - exponent: 11111110
 \Rightarrow actual exponent = $254 - 127 = +127$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000...00 and 1111...11 reserved
- Smallest value
 - Exponent: 000000000001
 \Rightarrow actual exponent = $1 - 1023 = -1022$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
 - Exponent: 111111111110
 \Rightarrow actual exponent = $2046 - 1023 = +1023$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$



(a) Twos Complement Integers



Floating-Point Example

- Represent -0.75

Floating-Point Example

- Represent -0.75

0.75_{10} in binary:

$$0.75 * 2 = 1.5 \quad \text{Integer part is 1}$$

$$0.5 * 2 = 1.0 \quad \text{Integer part is 1}$$

$$\Rightarrow 0.75_{10} = 0.11_2$$

Floating-Point Example

- Represent -0.75

- 0.75_{10} in binary:

$$0.75 * 2 = 1.5 \quad \text{Integer part is 1}$$

$$0.5 * 2 = 1.0 \quad \text{Integer part is 1}$$

$$\Rightarrow 0.75_{10} = 0.11_2 = 1.1_2 \times 2^{-1}$$

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

Floating-Point Example

- Represent -0.75

– 0.75_{10} in binary:

$$0.75 * 2 = 1.5 \quad \text{Integer part is 1}$$

$$0.5 * 2 = 1.0 \quad \text{Integer part is 1}$$

$$\Rightarrow 0.75_{10} = 0.11_2 = 1.1_2 \times 2^{-1}$$

$$-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1} \quad x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

Single Precision

Exponent with bias used for representation ($-1 + 127 = \mathbf{126}$)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit

8 bits

23 bits

Floating-Point Example

- Represent -0.75

- 0.75_{10} in binary:

$$0.75 * 2 = 1.5 \quad \text{Integer part is 1}$$

$$0.5 * 2 = 1.0 \quad \text{Integer part is 1}$$

$$\Rightarrow 0.75_{10} = 0.11_2 = 1.1_2 \times 2^{-1}$$

$$-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1} \quad x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

Double Precision

Exponent with bias used for representation $(-1 + 1023 = \mathbf{1022})$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit

11 bits

20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

32 bits

Convert Binary to decimal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Convert Binary to decimal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

– S = 1

Convert Binary to decimal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

– S = 1

– Exponent = $10000001_2 = 1*2^0 + 1*2^7 = 129$

Convert Binary to decimal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- $S = 1$
- Exponent = $10000001_2 = 1*2^0 + 1*2^7 = 129$
- Fraction = $01000...00_2 = 1*2^{-2}$

Convert Binary to decimal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- $S = 1$
- Exponent = $10000001_2 = 1 \cdot 2^0 + 1 \cdot 2^7 = 129$
- Fraction = $01000...00_2 = 1 \cdot 2^{-2}$
- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)}$
 $= (-1) \times 1.25 \times 2^2$
 $= -5.0$

Floating-Point Addition

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$

Floating-Point Addition

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
 - Shift number with smaller exponent
 - $9.999 \times \underline{10^1} + 0.016 \times \underline{10^1}$

Floating-Point Addition

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
 - Shift number with smaller exponent
 - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
 - 9.999 $\times 10^1 +$ 0.016 $\times 10^1 = 10.015 \times 10^1$

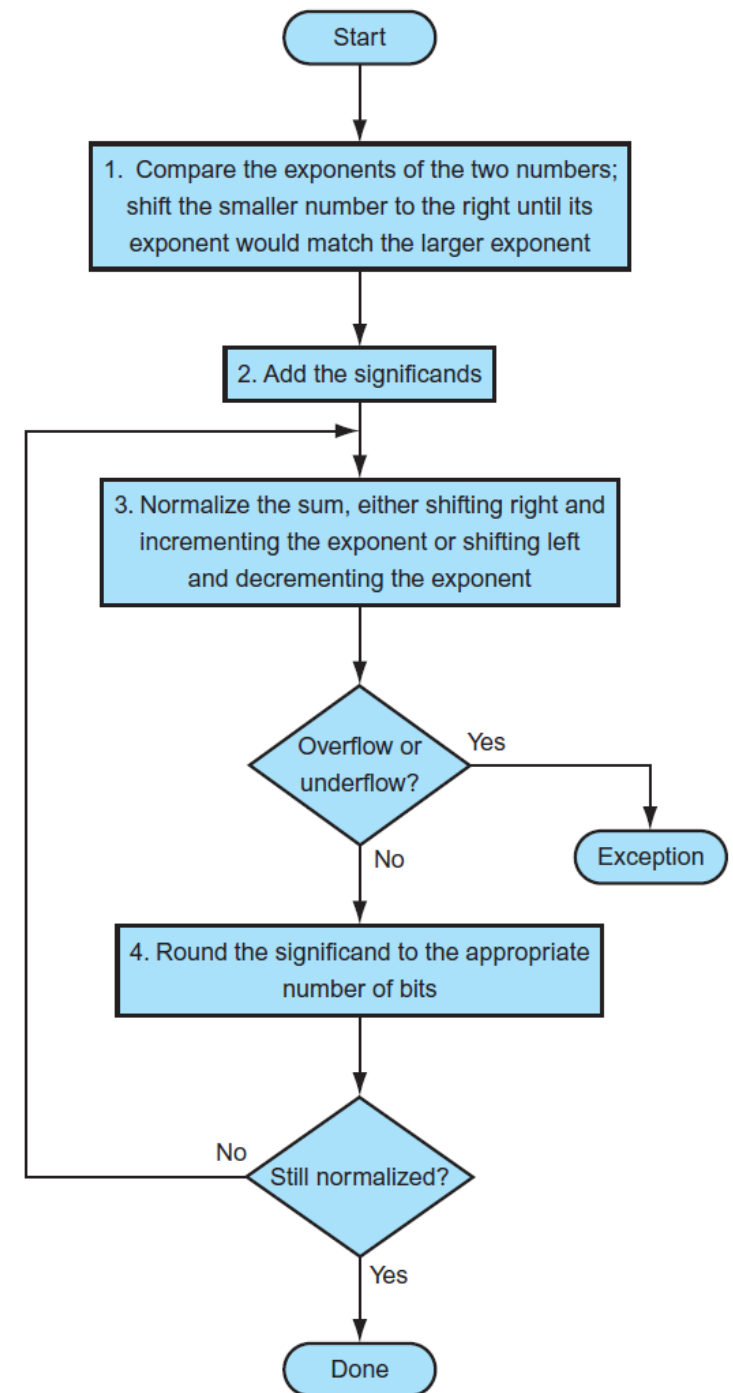
Floating-Point Addition

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
 - Shift number with smaller exponent
 - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
 - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
 - 1.0015×10^2

Floating-Point Addition

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
 - Shift number with smaller exponent
 - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
 - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
 - 1.0015×10^2
- 4. Round and renormalize if necessary
 - 1.002 $\times 10^2$

Floating-Point Addition



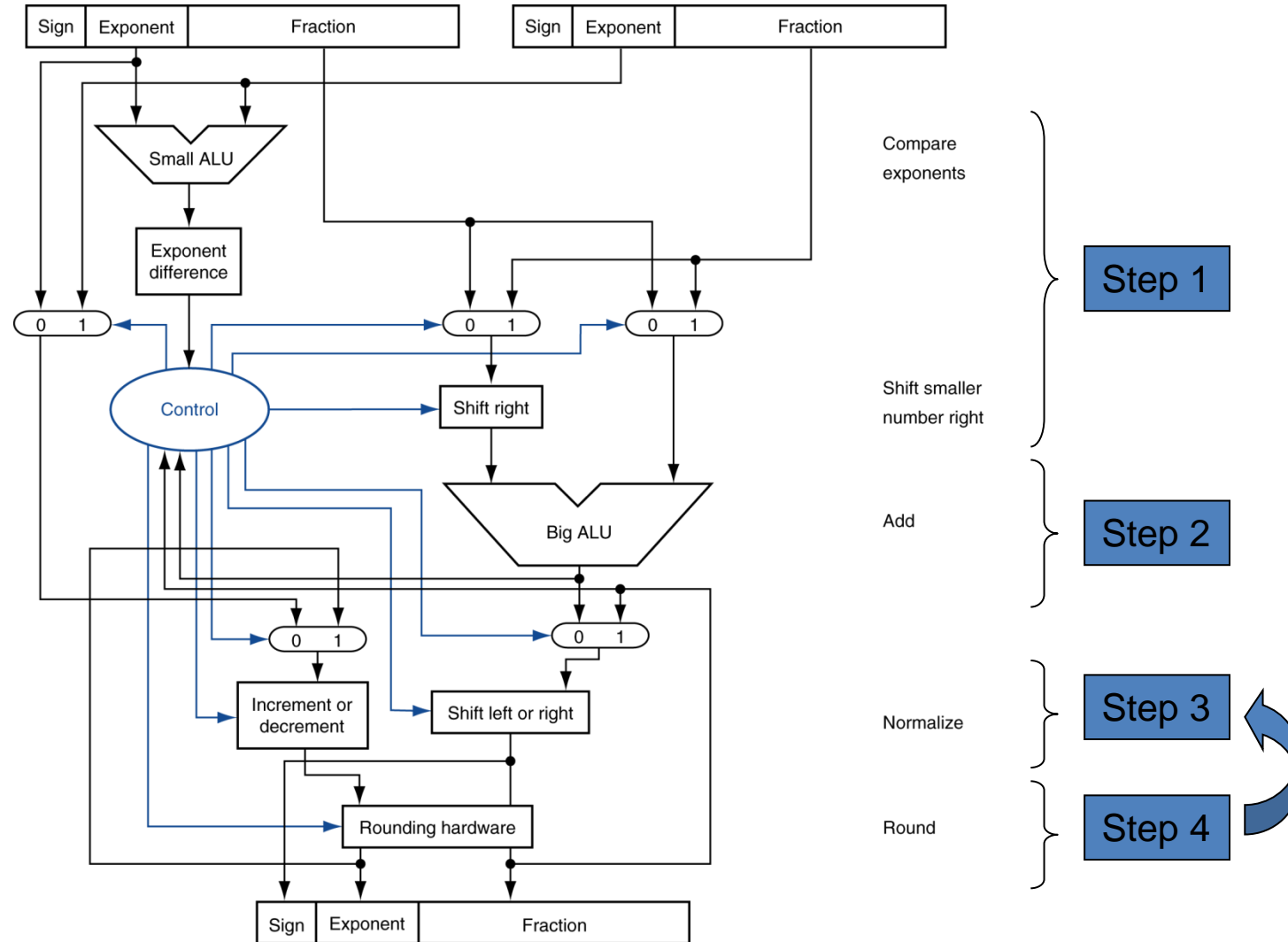
Floating-Point Addition

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ ($0.5 + -0.4375$)
- 1. Align binary points
 - Shift number with smaller exponent
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
 - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
 - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
 - Much longer than integer operations
 - Slower clock would penalize all instructions
- FP adder usually takes several cycles

FP Adder Hardware



Floating-Point Multiplication

- Consider a 4-digit decimal example
 - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
 - For biased exponents, subtract bias from sum
 - New exponent = $10 + -5 = 5$
- 2. Multiply significands
 - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$
- 3. Normalize result & check for over/underflow
 - 1.0212×10^6
- 4. Round and renormalize if necessary
 - 1.021×10^6
- 5. Determine sign of result from signs of operands
 - $+1.021 \times 10^6$

Floating-Point Multiplication

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
 - Unbiased: $-1 + -2 = -3$
 - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
 - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
 - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
 - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: $+ve \times -ve \Rightarrow -ve$
 - $-1.110_2 \times 2^{-3} = -0.21875$

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
- Operations usually takes several cycles

FP Instructions in LEGv8

- Separate FP registers
 - 32 single-precision: S0, ..., S31
 - 32 double-precision: D0, ..., D31
 - S_n stored in the lower 32 bits of D_n
- FP instructions operate only on FP registers
 - Programs generally don't do integer ops on FP data, or vice versa
 - More registers with minimal code-size impact
- FP load and store instructions
 - LDURS, LDURD
 - STURS, STURD

Instructions

Type	Name
Arithmetic	ADD, SUB, MUL
Data transfer	LDUR, STUR, LDURS, STURS, LDURD, STURD
Arithmetic Immediate	ADDI, SUBI, ORRI, ANDI, EORI, MUL, SMULH, UMULH, SDIV, UDIV
Logical Operations	LSL, LSR, AND, ORR, EOR
Branches	B, CBZ, CBNZ, B.Cond
Set Condition Flag	ADDS, ADDIS, SUBS, SUBIS, ANDS, ANDIS

FP Instructions in LEGv8

- Single-precision arithmetic
 - FADDS, FSUBS, FMULS, FDIVS
 - e.g., FADDS S2, S4, S6
- Double-precision arithmetic
 - FADDD, FSUBD, FMULD, FDIVD
 - e.g., FADDD D2, D4, D6
- Branch on FP condition code true or false
 - B.cond

Instructions

Type	Name
Arithmetic	ADD, SUB, MUL
Data transfer	LDUR, STUR, LDURS, STURS, LDURD, STURD
Arithmetic Immediate	ADDI, SUBI, ORRI, ANDI, EORI, MUL, SMULH, UMULH, SDIV, UDIV, FADDS, FSUBS, FMULS, FDIVS, FADDD, FSUBD, FMULD, FDIVD
Logical Operations	LSL, LSR, AND, ORR, EOR
Branches	B, CBZ, CBNZ, B.Cond
Set Condition Flag	ADDS, ADDIS, SUBS, SUBIS, ANDS, ANDIS