Floating-point arithmetic

many applications require:

- real numbers (like $\pi, \hbar \ldots$)
- numbers over huge range
  (from femtoseconds to hours, from nanometers to kilometers…)

- a compromise:

  in many practical engineering problems
  accuracy is less important than „close enough”

Fixed-point arithmetic

• the numbers of digits of integer and fraction parts are fixed (for given application, measuring range etc.)

• computations are carried out using common integer arithmetic…

Fixed-point arithmetic

- 4 digits, unsigned decimal

integer	1 decimal	2 decimals	3 decimals
0000	000.0	00.00	0.000
0001	000.1	00.01	0.001
.	.	.	.
.	.	.	.
.	.	.	.
9998	999.8	99.98	9.998
9999	999.9	99.99	9.999
range: $0\ldots10^4-1$	range: $0\ldots10^3-0.1$	range: $0\ldots10^2-0.01$	range: $0\ldots10-0.001$
absolute rounding error: <=1/2	absolute rounding error: <=0.1/2	absolute rounding error: <=0.01/2	absolute rounding error: <=0.001/2

1/2**ulp**s – **u**nits **p**er **l**ast place

Fixed-point arithmetic
Fractional Binary Numbers

0 0 1 0 1 0 1 0 . 0 1 1 0 1 0 0

$$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 \quad 2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5} 2^{-6} 2^{-7}$$

$$32+8+2 \qquad \frac{1}{4}+\frac{1}{8}+\frac{1}{32} \qquad 42\frac{13}{32}$$

$$42\frac{13}{32} = 42.40625$$

In this case the binary representation is accurate…

Fractional Binary Numbers – limitations

- **only the numbers of the form: $\frac{x}{2^n}$ can be represented exactly**

- the rest have repetitive bit patterns…

$$0.1_{10} \approx\sim 0.00011001100110011001100110011(0011)_2$$

$$0.3_{10} \approx\sim 0.010101010101010101010101010101(01)_2$$

Scientific notation

- renders numbers with a single digit to the left of the decimal(binary) point

$0.000000001 = 1 \times 10^{-9}$
$3155760000 = 3.15576 \times 10^{9}$

Normalized number – a number in scientific/floating point notation that has **no leading zeros**

3.141592654 – normalized number
$3141.592654 \times 10^{-3}$ – denormal number
$0.003141593 \times 10^{+3}$ – denormal number

$$real\_value = S \cdot F \cdot B^{E}$$

S – sign: 1 or -1
F – mantissa, significand, **normalized fraction** [1,B)
B – base of the number system
E – exponent (signed integer)

Floating-point numbers

Equivalent representations of 1234.0:

1234000.0×10^{-3}

123400.0×10^{-2}

12340.0×10^{-1}

1234.0×10^{0}

123.4×10^{1}

12.34×10^{2}

1.234×10^{3}

0.1234×10^{4}

normalized number

•The decimal point ”floats” to the left or right
(with the appropriate adjustment of the exponent)

- **Floating point representation is generally non-unique**
- **Normalization makes this representation unique!**

Unfortunately:
Normalization makes impossible to represent the zero!!!

Floating-point numbers

a 4 digit number as in previous „fixed point” example, written in scientific notation:

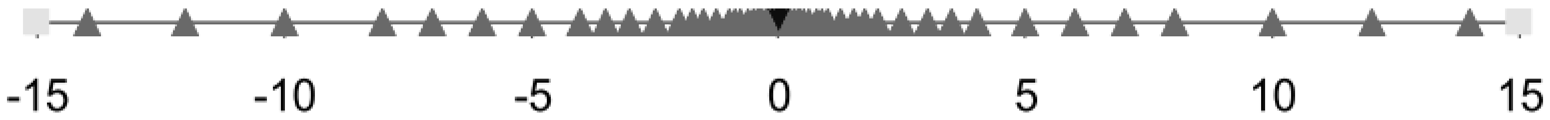
$1.23 * 10^{4}$ (one-digit, signed exponent: -4…,0,…+5)

Minimal normalized value: **1.00*10-4**
Max: **9.99*10+5**

resultant range: **0.0001…999000**

abs. rounding error exponent=5: <=1000/2
abs. rounding error exponent=-4: <=0.000001/2

Floating point representation allows to use **large range** or **low representation error**



e.g. max range in the 4 digit fixed point representation was 0…9999 ($0 \dots 9.999 * 10^{3}$)

Floating-point IEEE 754

- in the beginning each designer/manufacturer of the software and hardware had a different representation

- now we have **IEEE 754** (1985-2019): uniform standard for floating point arithmetic
  - data types
  - rounding rules
  - **operations**
    - **required** (+,-,*,/, type conversions, comparisons etc)
    - **recommended**, like sin(x), e^x, x^n, square root…
  - exception handling:
    divide by zero, underflow, overflow, square root of negative…

Floating-point IEEE 754

implied, not stored

$$floating\_point\_number = (-1)^S \cdot (1 + F) \cdot 2^{E-Bias}$$

S – sign bit (1 – negative, 0 – positive)
F – normalized mantissa (without integer part)
Bias – offset: 127 – single, 1023 double precision, respectively
E – biased exponent

S E F

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	single precision (float)
	11 bits	52 bits	double precision (double)
	15 bits	64 bits	extended double precision „long double” 80bit (x86/x87 internal data type)

- IEEE 754 also defines 128 and 256 bit types (quadruple and octuple precision…)

Floating-point IEEE 754

BIASED EXPONENT

- exponent in fp/scientific notation is a **signed** integer number,

- however, value in „exponent field” is stored as an unsigned binary number…

To provide negative exponents, the **bias** is subtracted from the value
in the exponent field to determine its true value.

- **Bias** is a number that is approximately in the middle of the range of values expressible by the exponent.

- The minimal and maximal values are reserved numbers for „special cases”.

Example – **Single Precision** – 8 bit exponent

- **0 i 255 – special/reserved values**
- **useful range 1 - 254**

- **Maximal exponent Emax = 127**, coded as 254 (127+127)
- **Minimal exponent Emin = -126**, coded as 1 (-126+127)

Floating Point – IEEE 754

Special symbols

Signed zeros!
+0 = -0 mul/div keep the sign:
5*(+0) = +0 5*(-0) = -0

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

Single precision (float):

E_{max} = 254 – 127 = 127
F_{max} = 1.1…1 (almost 2)
Max ≈ 2*2^127 ≈ $3.4028*10^{38}$
(range: $-3.4028*10^{38}$ $+3.4028*10^{38}$)

E_{min} = 1 – 127 = –126
F_{max} = 1.0…0
min_normalized_value = $1*2^{-126} = 1.17549*10^{-38}$

$$\frac{+1}{+0} = +\infty \qquad \frac{+1}{-0} = -\infty$$

$$x + (-\infty) = -\infty$$

result of:
- illegal computation:

$$\infty - \infty$$

$$0 / 0$$

$$\infty / \infty$$

- or operation involving a NaN

Floating Point – IEEE 754 - conversion

1. Normalize

67.5 /2
33.75 /2
16.875 /2
8.4375 /2
4.21875 /2
2.109375 /2
1.0546875

6 times

(1.0546875 * 2^**6** = 67.5)

2. Convert mantissa (without leading 1) **to binary**:

0.0546875 *2 = **0**.1099375
0.1099375 *2 = **0**.21875
0.21875 *2 = **0**.4375
0.4375 *2 = **0**.875
0.875 *2 = **1**.75
0.75 *2 = **1**.5
0.5 *2 = **1**.**0**

bias
133=127+**6**

0 10000101 **000011**00000000000000000 -> 0100 0010 1000 0111 0000 0000 0000 0000
42870000hex

S E F

Floating Point – IEEE 754 - conversion

-0.4375

Normalize

0.4375 *2
0.875 *2
1.75

2 times

(1.75 * 2^**-2** = 0.4375)

0.75 *2 = 1.5
0.5 *2 = 1.**0**

125=127+(**-2**)

1 01111101 **11**000000000000000000000 -> 1011 1110 1110 0000 0000 0000 0000 0000

BEE00000hex

S E F

Floating Point – IEEE 754 - conversion

What is 0xC0A80000 ?

C 0 A 8 0 0 0 0

1100 0000 1010 1000 0000 0000 0000 0000

1 10000001 01010000000000000000000

- 129-127= 2 1/4 + 1/16 = 5/16=0.3125

$(-1)*(1+0.3125)*2^{2}= -5.25$

Floating Point – IEEE 754

Addition 4 significant decimal digits

$9.979*10^{1} + 3.52*10^{-1}$ (already normalized)

1. **Shift the smaller number to right to align the exponents:**

   $3.52*10^{-1} = 0.0352*10^{1} \Rightarrow 0.035*10^{1}$ (we may lose accuracy…)

2. Add

```
   9.979 *10^1
   0.035 *10^1
+-----------------
  10.014 *10^1
```

3. Normalize (if necessary): $10.014 *10^{1} = 1.0014 *10^{2}$

4. and **round** the result:

$1.0014 *10^{2} = 1.001 *10^{2}$ (may result in another loss of accuracy…)

5. If necessary, normalize again (repeat 3 and 4)

Floating Point – IEEE 754 - Rounding

- **Round to Nearest, Half to Even** (default)

  Round to the nearest representable number.
  If exactly halfway between, round to nearest representable value with 0 in LSB (the nearest even fraction).

- **Round towards 0** (truncation)

  equivalent to dropping the extra bits.

- **Round up / towards +∞**

  to the closest representable (normalized) value greater then rounded value.

- **Round down / towards -∞**

  to the closest representable (normalized) value less than rounded value.

Floating Point – IEEE 754 - Rounding

How will be 2.5 rounded?

- **Round to Nearest, Half to Even:**

5.5	**6**
2.5	**2**
1.6	2
1.1	1
-1.1	-1
-1.6	-2
-2.5	**-2**
-5.5	**-6**

Floating Point – IEEE 754

Addition of single precision numbers:

1.11100100000000000000010 * 2^4
1.10000000000000010000101 * 2^2

shift right, align the exponents

1.11100100000000000000010 * 2^4
0.01100000000000000100001 **01** * 2^4

add

  1.11100100000000000000010 * 2^4
  0.01100000000000000100001 **01** * 2^4
+--
 10.01000100000000000100011 **01** * 2^4

normalize:

  1.00100010000000000010001 **101** * 2^5

do not discard these bits!

Floating Point – IEEE 754

Round bit R=0

Rounding to nearest:

1.00100010000000000010001 **1 0 1** * 2^5

LSB

Guard bit G=1

Sticky bit S=1
(logical OR of the rest discarded bits)

G R S – three bits – eight combinations:

0 0 0 - no action
0 x x - less than half way - **round down** (discard GRS bits)

1 0 0 - exactly half way - **round to even:** test the LSB

1 x x - more than half way - **round up:** add 1 to LSB

```
  1.00100010000000000010001   * 2^5
+ 0.00000000000000000000001
--------------------------------------------
  1.00100010000000000010010   * 2^5
```

Floating Point – IEEE 754

normalization – e.g. after subtraction

0.111101111111110101011 1 01 * 2^{-2}

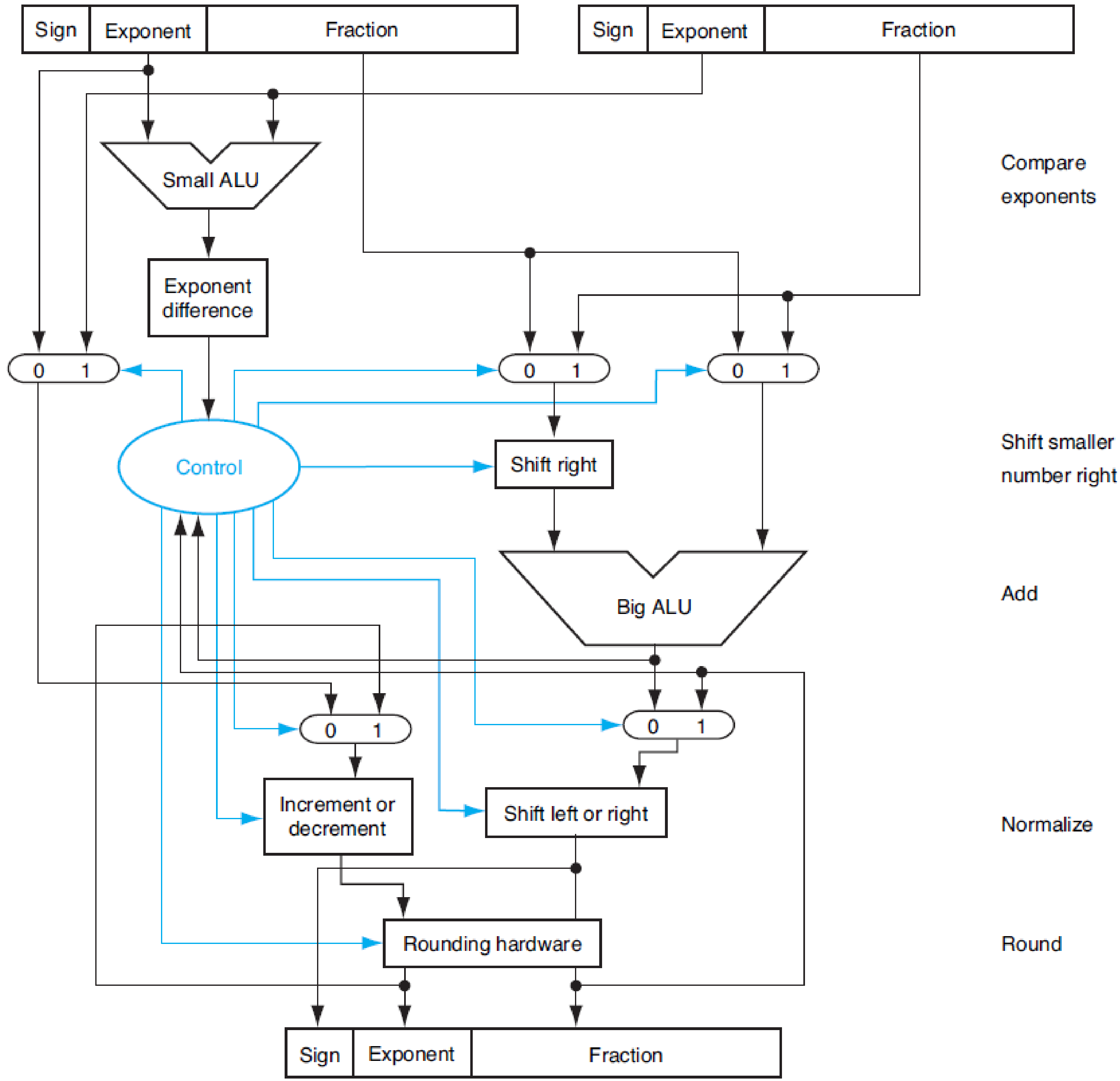
shift left to
eliminate leading zero(s)

Guard bit

1.11101111111110101011 01 * 2^{-3}

Floating Point – IEEE 754

Hardware adder



Floating Point – IEEE 754

multiplication: $2.34*10^{12} * 8.7*10^{-5}$ (normalized, four significant digits)

1. Add exponents: 12 + (-5) = 7

2. Multiply significands:

```
      2.340
      8.700
    x -------
      0000
     0000
   16380
  18720
 +-----------
  20358000
```

i.e. $20.358000*10^7$

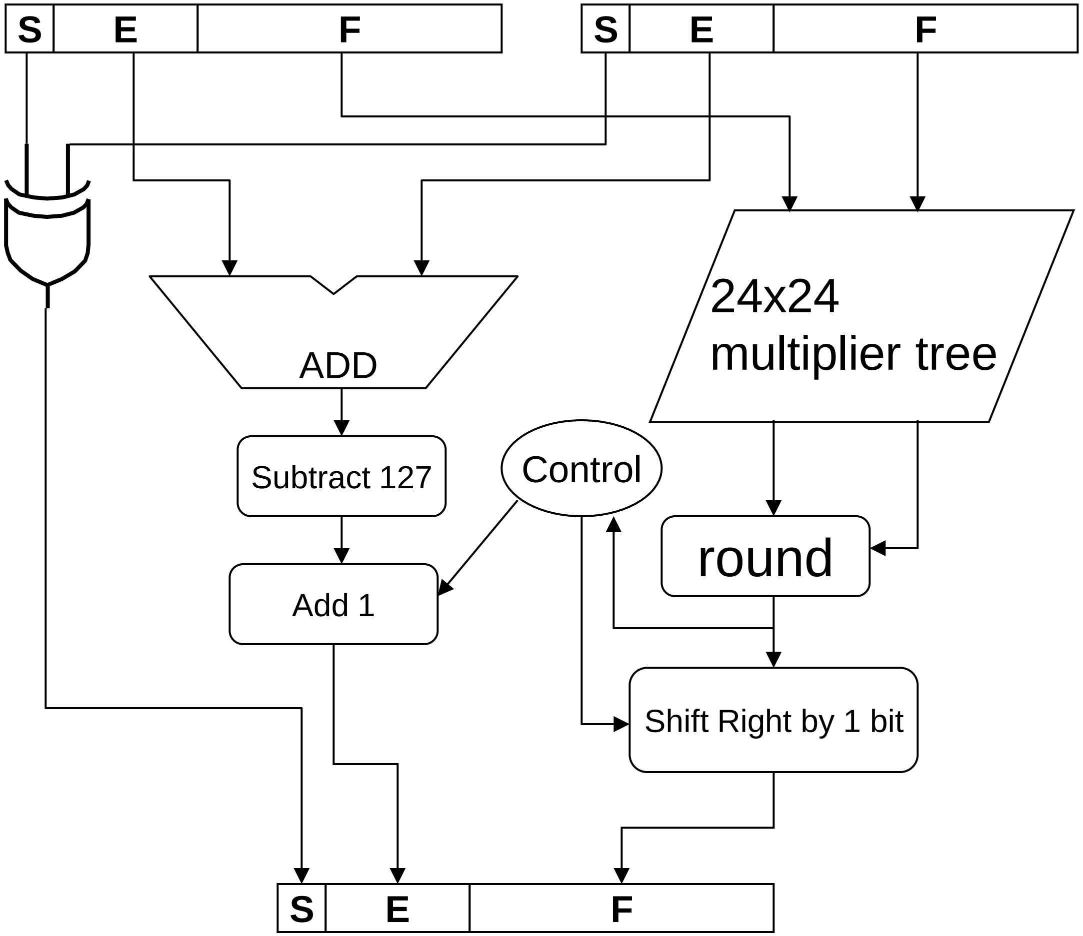
3. **Normalize the result** (if necessary) $20.358*10^7 = 2.0358 *10^8$
   (check for the overflow!)

4. Round $2.0358 *10^8 = 2.036 *10^8$

5. Determine the sign (xor)

Floating Point – IEEE 754

Hardware multiplier



$E_R = E_1 + E_2 - \text{bias}$

(do not need to adjust the bias twice)

Floating Point – IEEE 754

Denormal numbers & gradual underflow

Smallest normalized number = $1*10^{-98}$

base = β = 10

p = 3 digits

e_{min} = -98

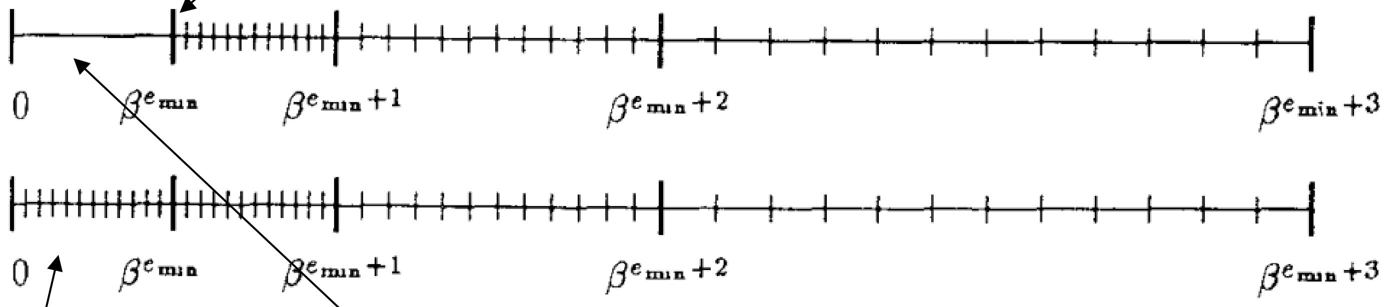


Figure 2. Flush to zero compared with gradual underflow.

Example:

x = $7.69*10^{-97}$

y = $7.62*10^{-97}$

true result: x-y = $7.00*10^{-99}$ computed x-y = 0

so: x-y = 0 but: $x \neq y$

try to execute: **if(x ≠ y) then z=1/(x-y) …**

Solution: use **denormal** number:

x-y = $0.70*10^{-98}$

Floating Point – IEEE 754

in FP arithmetic addition and multiplication are

commutative $a+b=b+a$

$a \times b = b \times a$

but not always associative or distributive $(a+b)+c \stackrel{?}{=} a+(b+c)$

$(a+b) \times c \stackrel{?}{=} a \times b + b \times c$

a=3456.789
b=45.12342
c=000.0003

```
3456.78900                        45.12342
+  45.12342                    +  0.00030
-----------------             ---------------
3501.91242 -> 3501.912            45.12372

3501.9120                       3456.78900
+    0.0003                    +  45.12372
-----------------              ---------------
3501.9123 -> 3501.912           3501.91272 -> 3501.913
```

Equality test use:

if abs(x-y)<e
e=very small, e.g 10^{-15}

instead of:

if (x==y)

Read about: machine epsilon, units per last place (ulp)…

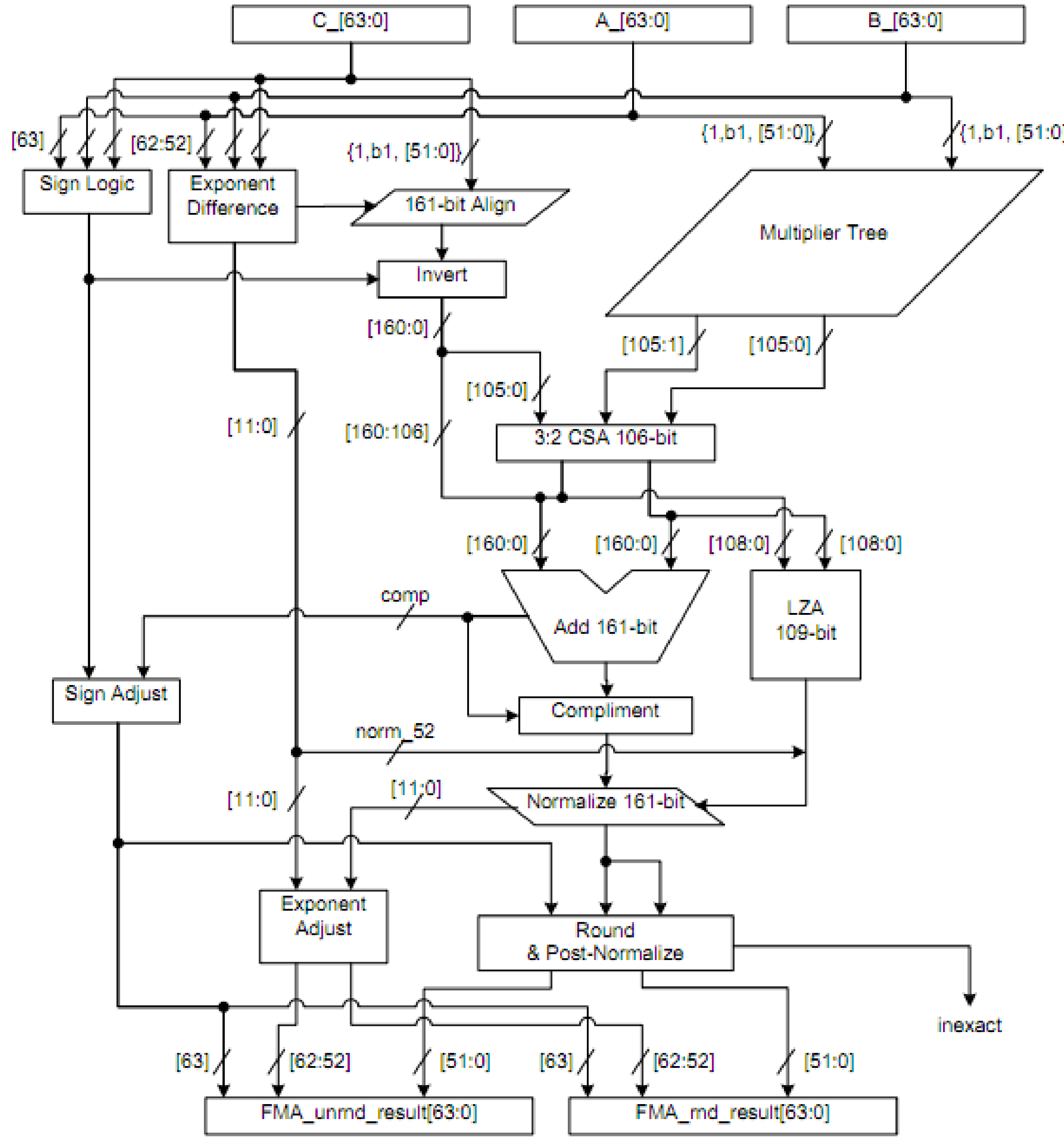
FMA fused multiply - add: d = a * b + c

- just one rounding
  (after addition),

- usually in SIMD
Single Instruction Multiple Data,
vector extensions (AVX).

or Multiply - Accumulate:

c := a * b + c



IBM PowerPC604e (90s)
Intel FMA – 2011