CS / EE 320
Computer Organization and Assembly Language
Spring 2025
Lecture 12

Shahid Masud

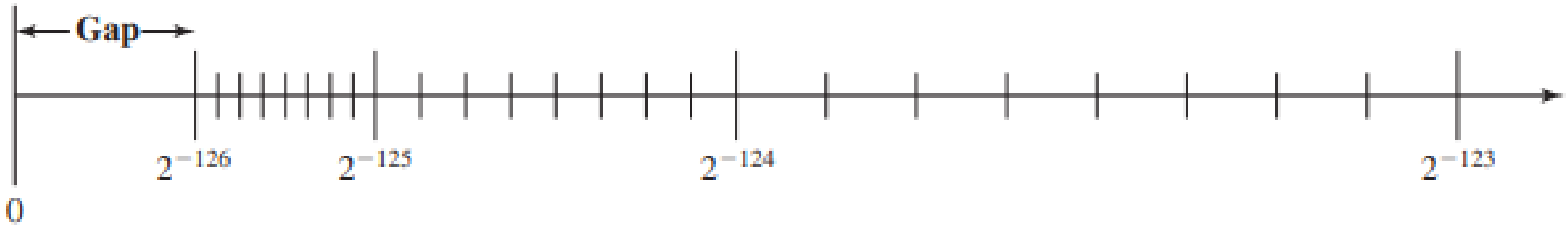
Topics: Floating Point Arithmetic operations, Hardware Design for Floating Point Arithmetic, MIPS and Floating Point, Rounding and Truncation Operations

Topics

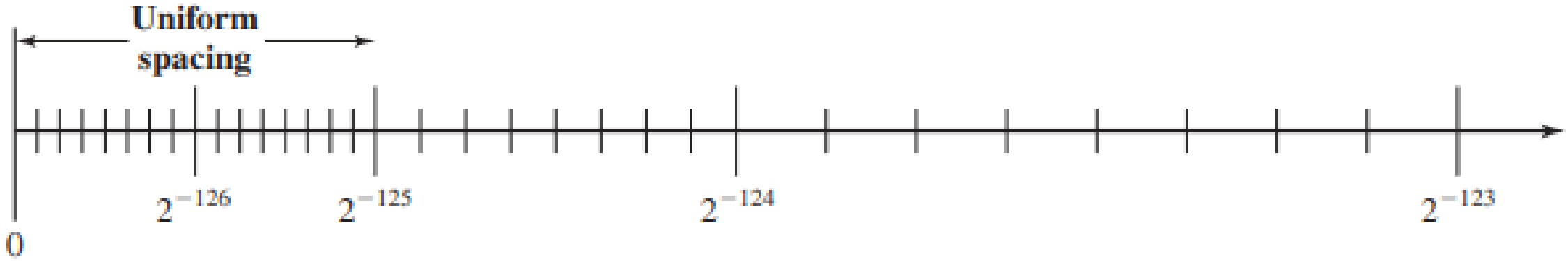
- Maintaining Precision through Guard, Round and Sticky (GRS) bits
- Arithmetic (Add, Sub, Mul, Div) in FP
- Examples of FP Add, Mult process
- Floating Point Arithmetic in Hardware
- MIPS Provisions for Floating Point Operations

QUIZ 3
NEXT WEEK

Subnormal or Denormal Numbers



(a) 32-bit format without subnormal numbers



(b) 32-bit format with subnormal numbers

IEEE 754 Floating-Point Representation

Single Precision IEEE 754

Sign	Exponent	Fraction	G R S
1 bit	8 bits	23 bits	Guard Bits

32 bits

For Accuracy

Double Precision IEEE 754

Sign	Exponent	Fraction	G R S
1 bit	11 bits	52 bits	Guard Bits

64 bits

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (**guard**, **round**, **sticky**)
  - Choice of **rounding modes**
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Rounding Policy

ROUNDING Another detail that affects the precision of the result is the rounding policy. The result of any operation on the significands is generally stored in a longer register. When the result is put back into the floating-point format, the extra bits must be eliminated in such a way as to produce a result that is close to the exact result. This process is called **rounding**.

A number of techniques have been explored for performing rounding. In fact, the IEEE standard lists four alternative approaches:

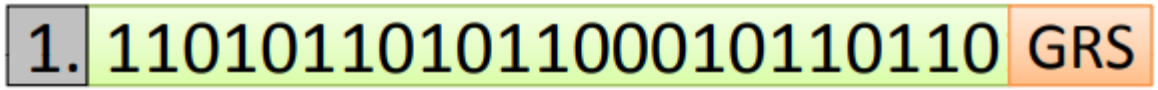
- **Round to nearest:** The result is rounded to the nearest representable number.
- **Round toward $+\infty$:** The result is rounded up toward plus infinity.
- **Round toward $-\infty$:** The result is rounded down toward negative infinity.
- **Round toward 0:** The result is rounded toward zero.

Let us consider each of these policies in turn. **Round to nearest** is the default rounding mode listed in the standard and is defined as follows: The representable value nearest to the infinitely precise result shall be delivered.

Accuracy in Floating-Point

- Precision is lost when some bits are shifted to right of the rightmost bit or are thrown
- Three extra bits are used internally -

  G (guard), R (round) and S (sticky)
  - G and R are simply the next two bits after LSB
  - S = 1 iff any bit to right of R is non-zero



Rounding Scheme in Floating-Point

- if G=1 & R=1, add 1 to LSB
- if G=0 & R=0 or 1, no change
- if G=1 & R=0, look at S
  - if S=1, add 1 to LSB
  - if S=0, round to the nearest "even"

    i.e., add 1 to LSB if LSB = 1

Another view of Rounding Scheme

G Guard	R Round	S Sticky	Rounding Applied
0	0	0	Truncate
0	0	1	Truncate
0	1	0	Truncate
0	1	1	Truncate
1	0	0	Round to Even
1	0	1	Round Up
1	1	0	Round Up
1	1	1	Round Up

S = OR (All bits in S and to the right of S)

Round to Even:
If S = 0, do nothing
If S = 1, add +1

The Use of Guard Bits

$$
\begin{aligned}
x &= 1.000.....00 \times 2^{1} \\
-y &= 0.111.....11 \times 2^{1} \\
z &= 0.000.....01 \times 2^{1} \\
&= 1.000.....00 \times 2^{-22}
\end{aligned}
$$

(a) Binary example, without guard bits

$$
\begin{aligned}
x &= 1.000.....00\ 0000 \times 2^{1} \\
-y &= 0.111.....11\ 1000 \times 2^{1} \\
z &= 0.000.....00\ 1000 \times 2^{1} \\
&= 1.000.....00\ 0000 \times 2^{-23}
\end{aligned}
$$

(b) Binary example, with guard bits

$$
\begin{aligned}
x &= .100000 \times 16^{1} \\
-y &= .0FFFFF \times 16^{1} \\
z &= .000001 \times 16^{1} \\
&= .100000 \times 16^{-4}
\end{aligned}
$$

(c) Hexadecimal example, without guard bits

$$
\begin{aligned}
x &= .100000\ 00 \times 16^{1} \\
-y &= .0FFFFF\ F0 \times 16^{1} \\
z &= .000000\ 10 \times 16^{1} \\
&= .100000\ 00 \times 16^{-5}
\end{aligned}
$$

(d) Hexadecimal example, with guard bits

Floating-Point Arithmetic – Basics

$$N=(-1)^S \times (1+F)\times 2^E$$

- A signed-magnitude system for the fractional part and a biased notation for the exponent
- Three subfields
  - Sign S
  - Fraction F (or Significand or Mantissa)
  - Exponent E
- Sign bit is 0 for positive numbers, 1 for negative numbers
- Fractions always start from **1**.xxxx, hence the integer **1** is not written (register has xxxx)
- Exponent is biased by +127 (add 127 to whatever is in register bits)
- Normalize: Express numbers is the standard format by shifting of bits and adding / subtracting from Exponent register

Floating-Point Arithmetic Operations

Floating-Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E}$ $X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E}$ } $X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Floating-Point Addition – Decimal example

- 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
  - 1.002×10^{2}

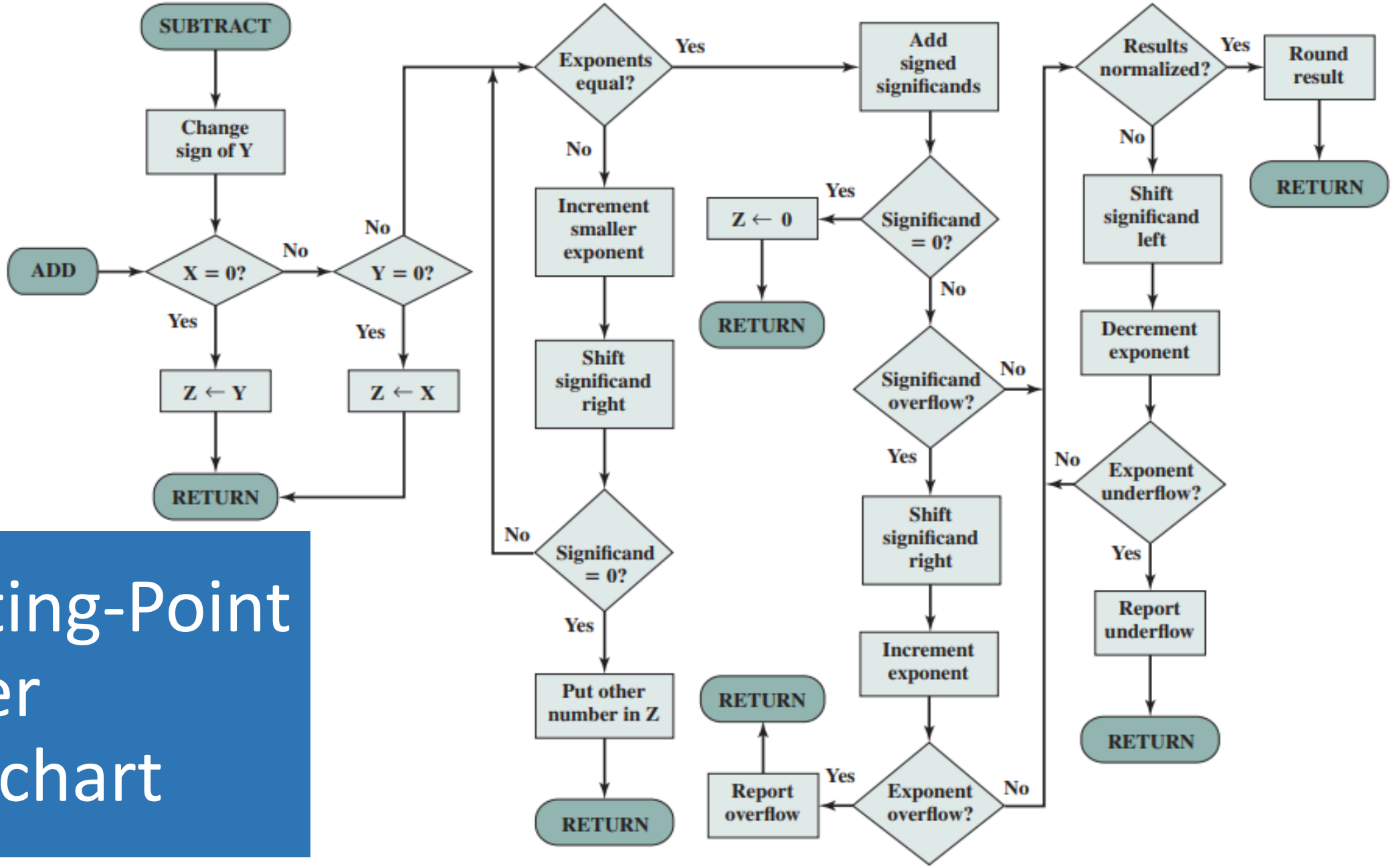
Floating-Point Addition – Binary example

- 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 Arithmetic Add / Subtract

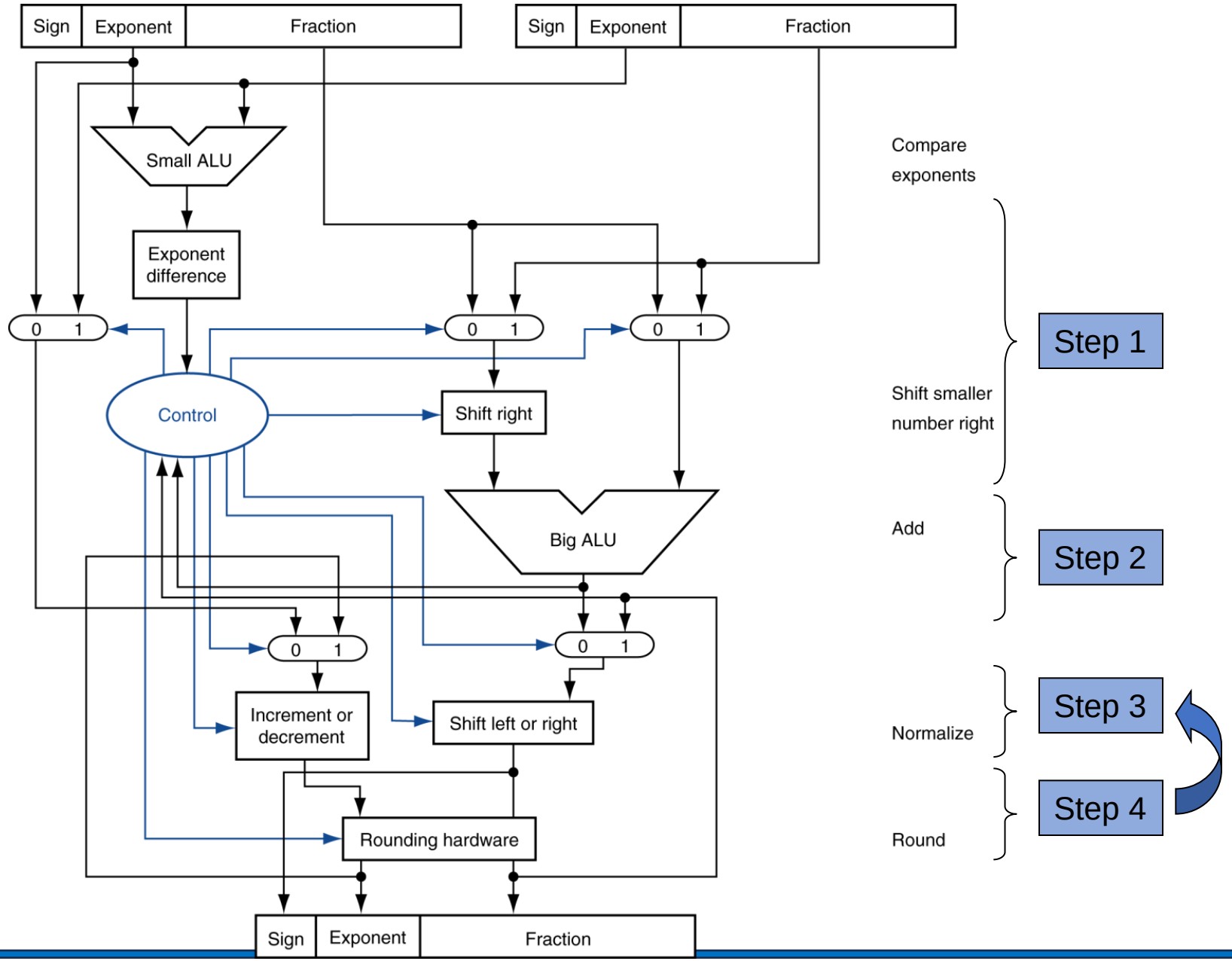
- Check for zeros
- Align significands (adjusting exponents)
- Add or subtract significands
- Normalize result

Floating-Point Adder Flowchart



Floating-Point Addition and Subtraction ($Z \leftarrow X \pm Y$)

FP Adder Hardware



Compare exponents

Shift smaller number right

Step 1

Add

Step 2

Normalize

Step 3

Round

Step 4

Floating-Point Multiplication

Consider two floating point numbers:
$(F_1 \times 2^{E1})$ and $(F_2 \times 2^{E2})$

The product of these two numbers is:

$$= (F_1 \times 2^{E1}) \times (F_2 \times 2^{E2})$$
$$= (F_1 \times F_2) \times 2^{(E1+E2)}$$
$$= F \times 2^{E}$$

Sign of result depends on Sign of the two numbers

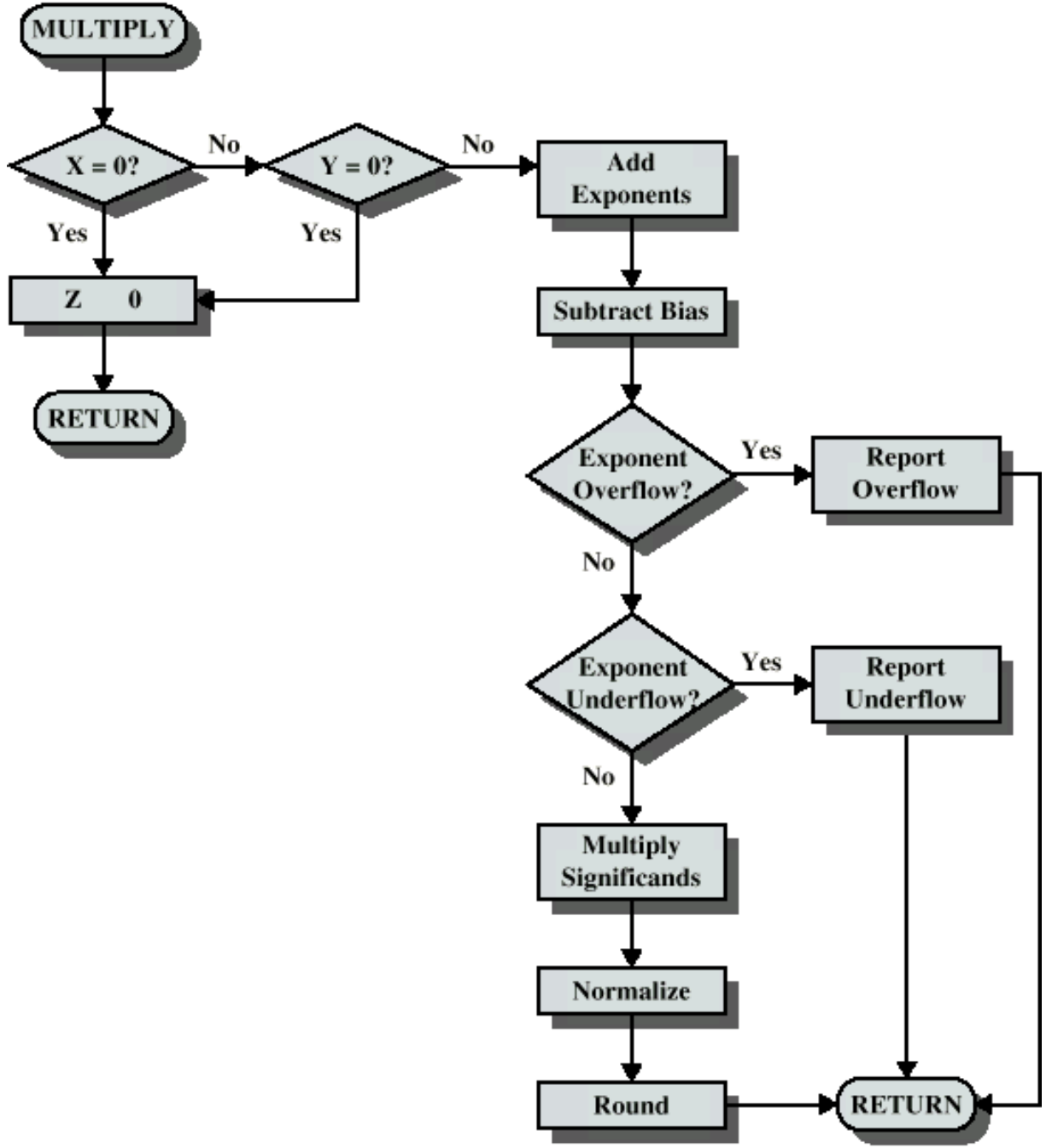
FP Arithmetic Multiply / Divide

- Check for zero
- Add/subtract exponents
- Multiply/divide significands (watch sign)
- Normalize
- Round
- All intermediate results should be in double length storage

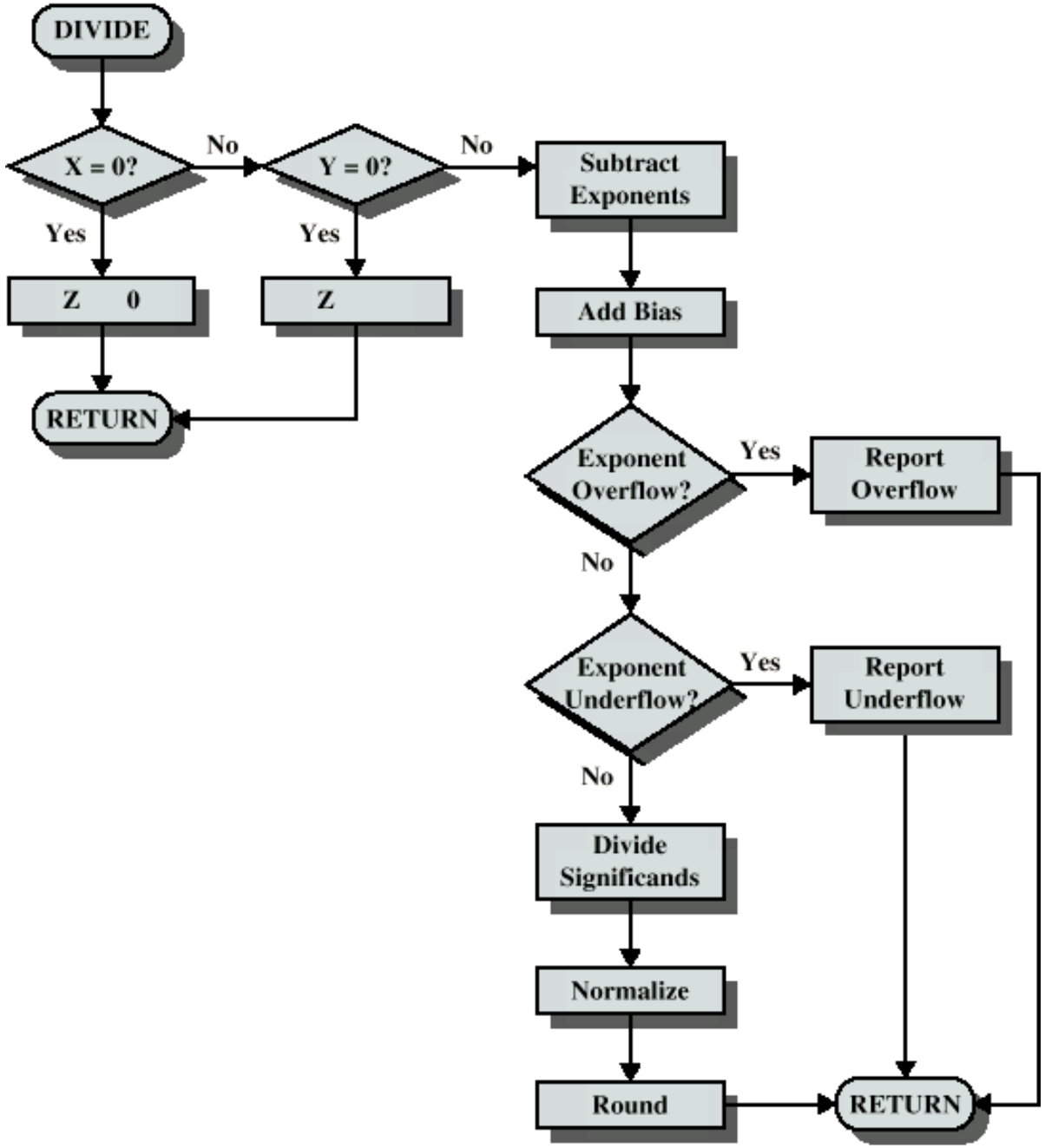
Floating-Point Multiplication Steps

- Normalize the two numbers if not done already
- The exponents of the Multiplier (E1) and the multiplicand (E2) bits are added and the base value is subtracted from the added result. The subtracted result is put in the exponential field of the result block → E1+E2-bias
- Multiply the two fractions (or significands)
- S1, the signed bit of the multiplicand is XOR'd with the multiplier signed bit of S2. The result is put into the resultant sign bit.
- The mantissa of the Multiplier (M1) and multiplicand (M2) are multiplied and the result is placed in the resultant field of the mantissa (truncate/round the result for 24 bits) → M1 * M2
- If the product is 0, adjust the proper representation of answer to 0
- If the product fraction is too big, normalize by shifting it right and incrementing the exponent
- If the product fraction is too small, normalize by shifting left and decrementing the exponent
- Round to appropriate number of bits. If rounding results in loss of normalization, then first normalize and then do the rounding
- If an exponent underflow (below -127) or overflow (above +127) occurs then generate an error condition

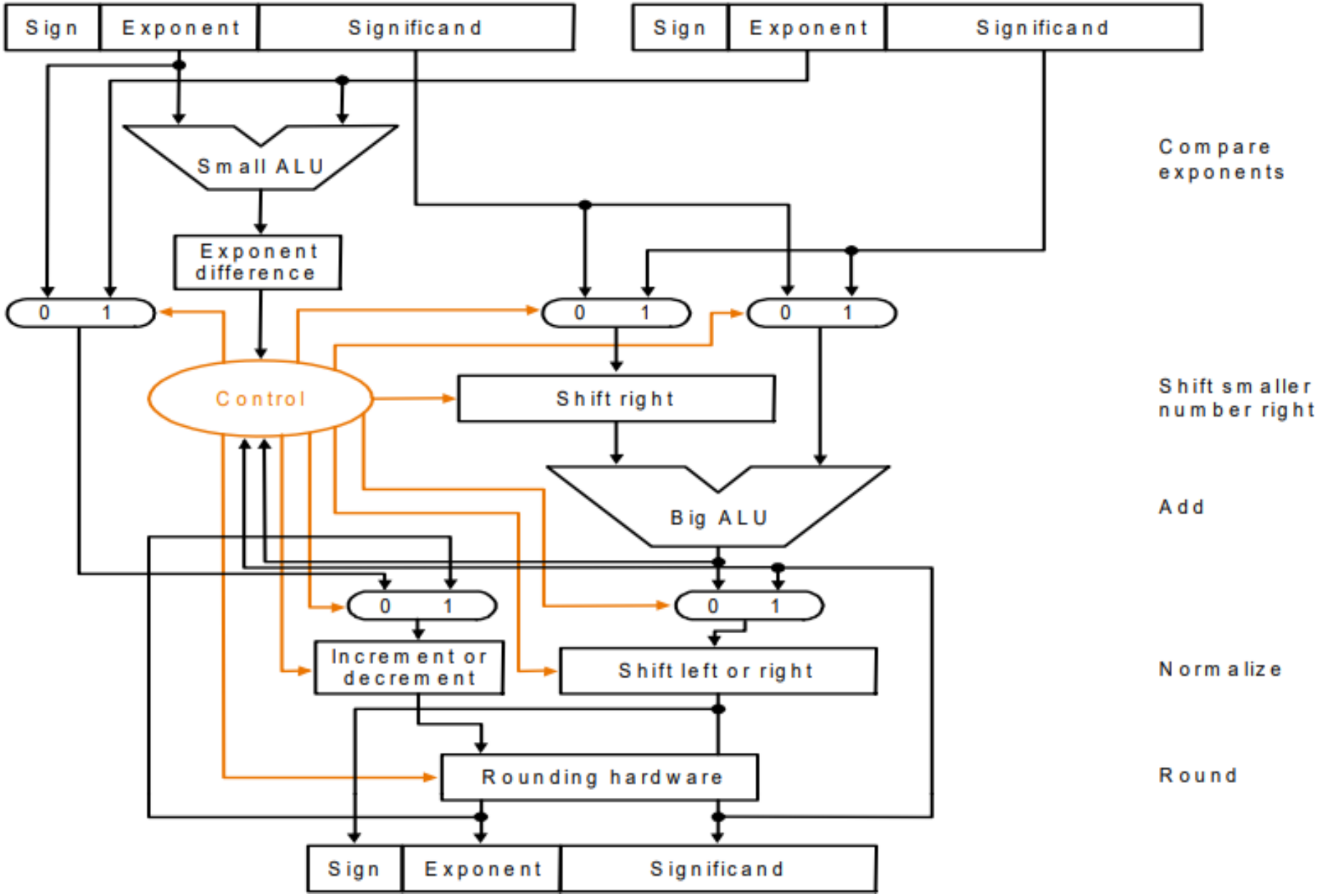
Floating-Point Multiplication



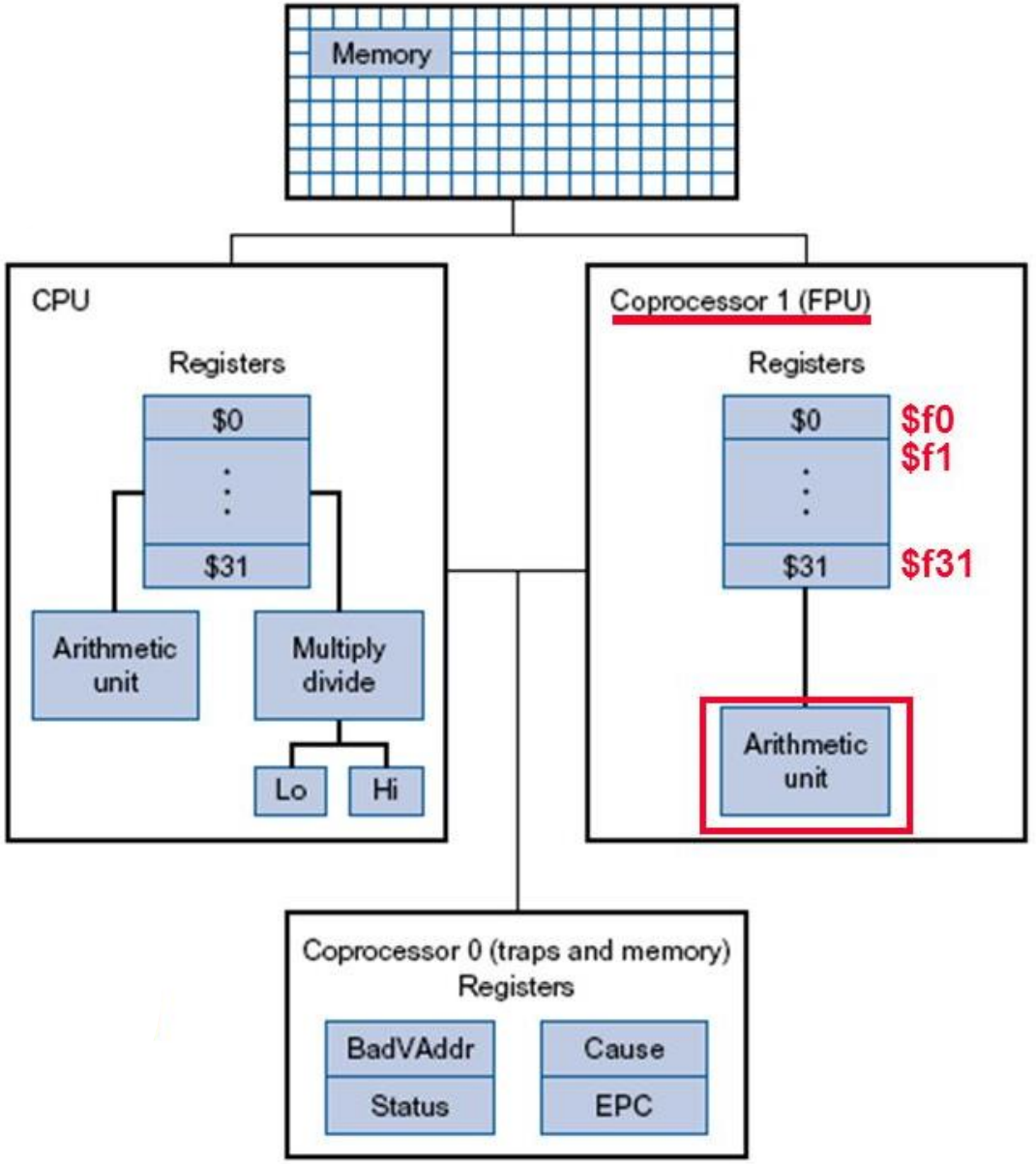
Floating-Point Division



Floating-Point Multiplication Hardware



MIPS CPU Organization viz Floating-Point



FP Support in MIPS Assembly

- **FP hardware is coprocessor 1**
  - Adjunct processor that extends the ISA
- **Separate FP registers**
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP registers
- **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**
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- **Single-precision arithmetic**
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- **Double-precision arithmetic**
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- **Single- and double-precision comparison**
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* is `eq`, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- **Branch on FP condition code true or false**
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

Load / Store in Floating-Point

Load and Store (single precision)

- Load or store from a memory location. Just load the 32 bits into the register.
  - `lwc1 $f0, 0($t0)`
    - `l.s $f0, 0($t0)`
  - `swc1 $f0, 0($t0)`
    - `s.s $f0, 0($t0)`

Single and Double Precision FP Arithmetic

Single Precision	Double Precision
abs.s $f0, $f1	abs.d $f0, $f2
add.s $f0, $f1, $f2	add.d $f0, $f2, $f4
sub.s $f0, $f1, $f2	sub.d $f0, $f2, $f4
mul.s $f0, $f1, $f2	mul.d $f0, $f2, $f4
div.s $f0, $f1, $f2	div.d $f0, $f2, $f4
neg.s $f0, $f1	neg.d $f0, $f2

Convert to / from Floating-Point to Integer

- `cvt.s.w $f0, $f1`
  - convert the 32 bits in `$f1` currently representing an integer to float of the same value and store in `$f0`
- `cvt.w.s $f0, $f1`
  - the reverse

- `cvt.d.w $f0, $f2`
  - convert the 64 bits in `$f2` currently representing an integer to float of the same value and store in `$f0`
- `cvt.w.d $f0, $f2`
  - the reverse

Comparison Instruction in FP

(single precision)

- `c.lt.s $f0,$f1`
  - set a flag in coprocessor 1if $f0 < $f1, else clear it. The flag will stay until set or cleared next time
- `c.le.s $f0,$f1`
  - set flag if $f0 <= $f1, else clear it
- `c.gt.s $f0,$f1`
  - set flag if $f0 > $f1, else clear it
- `c.ge.s $f0,$f1`
  - set flag if $f0 >= $f1, else clear it
- `c.eq.s $f0,$f1`
  - set flag if $f0 == $f1, else clear it
- `c.ne.s $f0,$f1`
  - set flag if $f0 != $f1, else clear it

Data Movement in Floating-Point

- Memory Transfer Instructions
  - l.s $f0, 100($t2) load word into $f0 from address $t2+100
  - s.s $f0, 100($t2) store word from $f0 into address $t2+100
- Data Movement between registers
  - mov.s $f0, $f2 move between FP registers
  - mfc1 $t1, $f2 move from FP registers (no conversion)
  - mtc1 $t1, $f2 move to FP registers (no conversion)
- Data conversion
  - cvt.w.s $f2, $f4 convert from single precision FP to integer
  - cvt.s.w $f2, $f4 convert from integer to single precision FP

Conditional Branch in Floating-Point

- Conditional jumps are performed in two stages
  1. Comparison of FP values sets a code in a special register
  2. Branch instructions jump depending on the value of the code
- Comparison
  - c.eq.s $f2, $f4 if $f2 == $f4 then code = 1 else code = 0
  - c.le.s $f2, $f4 if $f2 <= $f4 then code = 1 else code = 0
  - c.lt.s $f2, $f4 if $f2 < $f4 then code = 1 else code = 0
- Branches
  - bc1f label if code == 0 then jump to label
  - bc1t label if code == 1 then jump to label

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

FP Example: Array Multiplication

- X = X + Y × Z
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double x[][],
         double y[][], double z[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        x[i][j] = x[i][j]
                + y[i][k] * z[k][j];
}
```

  - Addresses of x, y, z in $a0, $a1, $a2, and i, j, k in $s0, $s1, $s2

FP Example: Array Multiplication

```
…
sll   $t0, $s0, 5       # $t0 = i*32 (size of row of y)
addu  $t0, $t0, $s2     # $t0 = i*size(row) + k
sll   $t0, $t0, 3       # $t0 = byte offset of [i][k]
addu  $t0, $a1, $t0     # $t0 = byte address of y[i][k]
l.d   $f18, 0($t0)      # $f18 = 8 bytes of y[i][k]
mul.d $f16, $f18, $f16  # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16    # f4=x[i][j] + y[i][k]*z[k][j]
addiu $s2, $s2, 1       # $k k + 1
bne   $s2, $t1, L3      # if (k != 32) go to L3
s.d   $f4, 0($t2)       # x[i][j] = $f4
addiu $s1, $s1, 1       # $j = j + 1
bne   $s1, $t1, L2      # if (j != 32) go to L2
addiu $s0, $s0, 1       # $i = i + 1
bne   $s0, $t1, L1      # if (i != 32) go to L1
```

Floating-Point Bug F00F

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia

Pentium FooF bug

7 languages

its [hide]

otion
rounds
so
nces
reading
al links

Article Talk | Read Edit View history

From Wikipedia, the free encyclopedia

The **Pentium F00F bug** is a design flaw in the majority of Intel Pentium, Pentium MMX, and Pentium OverDrive processors (all in the P5 microarchitecture). Discovered in 1997, it can result in the processor ceasing to function until the computer is physically rebooted. The bug has been circumvented through operating system updates.

The name is shorthand for `F0 0F C7 C8`, the hexadecimal encoding of one offending instruction.[1] More formally, the bug is called the *invalid operand with locked CMPXCHG8B instruction bug*.[2]

Description [edit]

In the x86 architecture, the byte sequence `F0 0F C7 C8` represents the instruction `lock cmpxchg8b eax` (locked compare and exchange of 8 bytes in register EAX). The bug also applies to opcodes ending in `C9` through `CF`, which specify register operands other than EAX. The `F0 0F C7 C8` instruction does not require any special privileges.

Big News around Floating-Point Errors

https://www.truenorthfloatingpoint.com/problem

Real Consequences

Since the invention of computers, real number calculations have produced hidden, unreported errors, sometimes catastrophically.

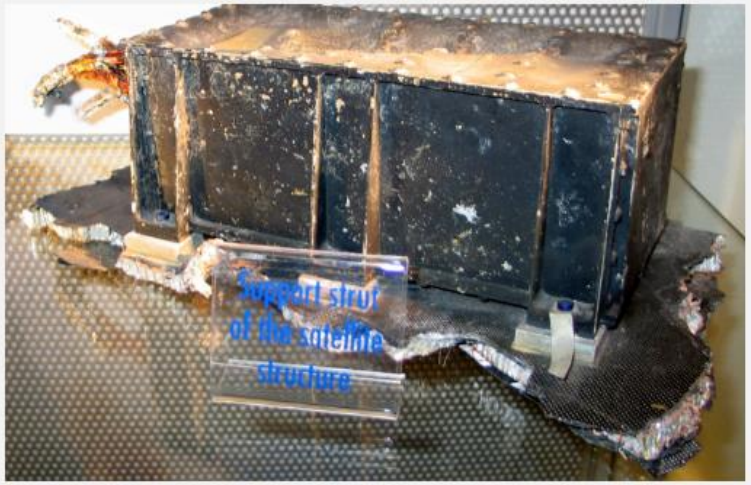
Photo from www.army.mil

PATRIOT MISSILE FAILURE

The most notorious floating point error catastrophe was the Patriot Missile Failure at Dhahran, Saudi Arabia, February 25, 1991, when a Patriot missile failed to destroy a SCUD missile and 128 U.S. military soldiers were killed or wounded as a result. This was the greatest combat loss in an Army unit since Vietnam. The conversion of 100 hours in tenths of a second (3600000) to floating point introduced an undetectable error resulting in the missile guidance software incorrectly locating the SCUD missile.

ARIANE 5 ROCKET, FLIGHT 501

On June 4th, 1996, 40 seconds into flight and at an altitude of 3.7 kilometers, the initial launch of the Ariane 5 rocket, flight 501, ended in RUD (colloquially, Rapid Unplanned Disassembly). Estimates of the loss of the rocket and cargo are as high as $500M. Cause of the failure was an inappropriate floating point conversion. (Photo from Deadpan)



Recovered piece of Ariane 5 after RUD

VANCOUVER STOCK EXCHANGE

In January of 1982 the Vancouver Stock Exchange started a stock index accumulating total stock value for all 1,400 stocks listed on the exchange. but truncating (rounding down) that sum up to 3000 times per day resulting in a loss of index value of about $25 per month for about 23 months indicating an index value of $524.811 when the actual value was $1098.892. (Image by Mafue)

MaverickCrunch Problems due to Floating-Point

https://en.wikipedia.org/wiki/MaverickCrunch

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia

MaverickCrunch

Add languages

Contents [hide]

(Top)
Features
Hardware bugs
Compiler support
References
External links

Article Talk | Read Edit View history

From Wikipedia, the free encyclopedia

The **MaverickCrunch** is a floating point math coprocessor core intended for digital audio. It was first presented by Cirrus Logic in June 2000[1] together with an ARM920T integer processor in their 200 MHz EP9302 EP9307 EP9312 and EP9315 System-on-Chip integrated circuits. Plagued with hardware bugs and poor compiler support, it was seldom used in any of the devices based on those chips and the product line was discontinued on April 1, 2008.

Features [edit]

The coprocessor has 16 64-bit registers which can be used for 32- or 64-bit integer and floating point operations and its floating point format is based on the IEEE-754 standard. It has its own instruction set which performs floating point addition, subtraction, multiplication, negation, absolute value, and comparisons as well as addition, multiplication and bit shifts on integers. It also has four 72-bit registers on which can perform a 32-bit multiply-and-accumulate instruction and a status register, as well as conversions between integer and floating point values and instructions to move data between itself and the ARM registers or memory.

It operates in parallel with the main processor, both processors receiving their instructions from a single 32-bit instruction stream. Thus, to use it efficiently, integer and floating point instructions must be interleaved so as to keep both processors busy.

A Cirrus Logic EP9315 chip containing a MaverickCrunch FPU

Hardware bugs [edit]

Five versions of the EP93xx silicon were issued: "D0" and "D1"/"E0"/"E1" and "E2", with major revisions to the MaverickCrunch core between D0 and D1 to fix its worst bugs. All have a dozen or more hardware bugs which either give imprecise or garbage results or clobber registers or memory when certain sequences of instructions are executed in a certain order.

Extreme Multiplication Errors in FP

Extreme errors

- As we saw, rounding errors in addition can occur if one argument is much smaller than the other, since we need to match the exponents.
- An extreme example with 32-bit IEEE values is the following.

$$(1.5 \times 10^{38}) + (1.0 \times 10^{0}) = 1.5 \times 10^{38}$$

The number 1.0×10^{0} is much smaller than 1.5×10^{38}, and it basically gets rounded out of existence.

- This has some nasty implications. The order in which you do additions can affect the result, so (x + y) + z is not always the same as x + (y + z)!

```
float x = -1.5e38;
float y =  1.5e38;
printf( "%f\n", (x + y) + 1.0 );
printf( "%f\n", x + (y + 1.0) );
```

Readings

- Chapter 3, P&H Textbook