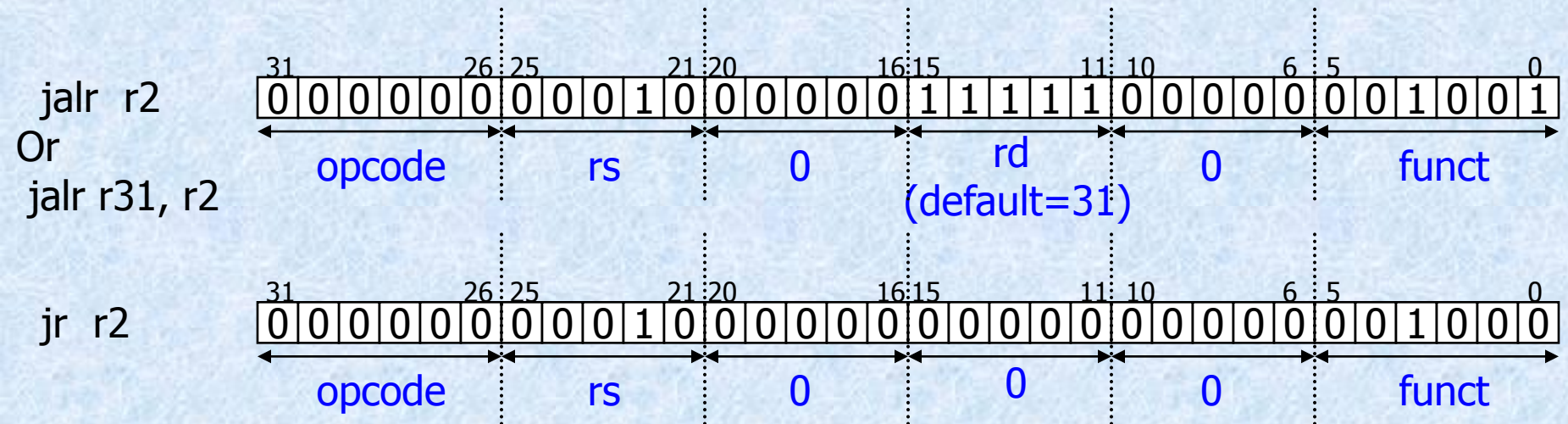


# JALR and JR uses R-Type

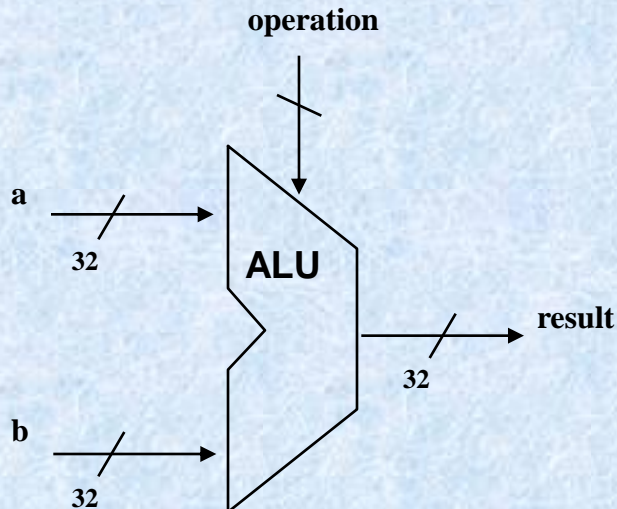
- JALR (Jump And Link Register) and JR (Jump Register)
  - Considered as R-type
  - Unconditional jump
  - JALR used for procedural call



# Arithmetic for Computers

# Arithmetic

- Where we've been:
  - performance
  - abstractions
    - *instruction set architecture*
    - *assembly language and machine language*
- What's up ahead:
  - *implementing the architecture*



# Numbers

- Bits are just bits (no inherent meaning)
  - conventions define relationship between bits and numbers
- Binary integers (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  - decimal: 0, ...,  $2^n - 1$  n bits
- Of course it gets more complicated:
  - bit strings are *finite*, but
    - for some *fractions* and *real* numbers, finitely many bits is not enough, so
    - *overflow* & *approximation* errors: e.g., represent 1/3 as binary!
  - *negative* integers
- How do we represent negative integers?
  - which bit patterns will represent which integers?

# Possible Representations

- Sign Magnitude:

000 = 0	ambiguous zero
001 = +1	
010 = +2	
011 = +3	
100 = 0	
101 = -1	
110 = -2	
111 = -3	
- One's Complement

000 = 0	ambiguous zero
001 = +1	
010 = +2	
011 = +3	
100 = -3	
101 = -2	
110 = -1	
111 = 0	
- Two's Complement

unequal no. of negatives and positives; unique zero	000 = 0
	001 = +1
	010 = +2
	011 = +3
	100 = -4
	101 = -3
	110 = -2
	111 = -1
- Issues:
  - *balance* – equal number of negatives and positives
  - *ambiguous zero* – whether more than one zero representation
  - ease of arithmetic operations



# Representation Formulae

- Two's complement:

$$x_n x_{n-1} \dots x_0 = x_n * -2^n + x_{n-1} * 2^{n-1} + \dots + x_0 * 2^0$$

**or**

$$x_n X' = x_n * -2^n + X' \quad (\text{writing rightmost } n \text{ bits } x_{n-1} \dots x_0 \text{ as } X')$$

$$= \begin{cases} X', & \text{if } x_n = 0 \\ -2^n + X', & \text{if } x_n = 1 \end{cases}$$

- One's complement:

$$x_n X' = \begin{cases} X', & \text{if } x_n = 0 \\ -2^n + 1 + X', & \text{if } x_n = 1 \end{cases}$$

# MIPS – 2's complement

- 32 bit signed numbers:

0000	0000	0000	0000	0000	0000	0000	0000	$_{\text{two}}$	=	$0_{\text{ten}}$	
0000	0000	0000	0000	0000	0000	0000	0001	$_{\text{two}}$	=	$+1_{\text{ten}}$	
0000	0000	0000	0000	0000	0000	0000	0010	$_{\text{two}}$	=	$+2_{\text{ten}}$	
...											
0111	1111	1111	1111	1111	1111	1111	1110	$_{\text{two}}$	=	$+2,147,483,646_{\text{ten}}$	maxint
0111	1111	1111	1111	1111	1111	1111	1111	$_{\text{two}}$	=	$+2,147,483,647_{\text{ten}}$	maxint
1000	0000	0000	0000	0000	0000	0000	0000	$_{\text{two}}$	=	$-2,147,483,648_{\text{ten}}$	minint
1000	0000	0000	0000	0000	0000	0000	0001	$_{\text{two}}$	=	$-2,147,483,647_{\text{ten}}$	minint
1000	0000	0000	0000	0000	0000	0000	0010	$_{\text{two}}$	=	$-2,147,483,646_{\text{ten}}$	minint
...											
1111	1111	1111	1111	1111	1111	1111	1101	$_{\text{two}}$	=	$-3_{\text{ten}}$	
1111	1111	1111	1111	1111	1111	1111	1110	$_{\text{two}}$	=	$-2_{\text{ten}}$	
1111	1111	1111	1111	1111	1111	1111	1111	$_{\text{two}}$	=	$-1_{\text{ten}}$	

Negative integers are exactly those that have leftmost bit 1

# Two's Complement Operations

- Negation Shortcut: To *negate* any two's complement integer (except for minint) *invert* all bits and *add 1*
  - note that *negate* and *invert* are different operations!
- Sign Extension Shortcut: To convert an n-bit integer into an integer with more than n bits – i.e., to make a narrow integer fill a wider word – *replicate the most significant bit (msb)* of the original number to fill the new bits to its left

- *Example:*    4-bit                      8-bit  
                  0010    =    0000 0010  
                  1010    =    1111 1010



# MIPS Notes

- `lb` **vs.** `lbu` (also `lh` **vs.** `lhu`)
  - signed load sign extends to fill 24 left bits
  - unsigned load fills left bits with 0's
- `sb` and `sh`
- `slt` & `slti`
  - compare signed numbers
- `sltu` & `sltiu`
  - compare unsigned numbers, i.e., treat both operands as non-negative

# Two's Complement Addition

- Perform add just as in school (carry/borrow 1s)

- Examples (4-bits):

0101	0110	1011	1001	1111
<u>0001</u>	<u>0101</u>	<u>0111</u>	<u>1010</u>	<u>1110</u>

Remember all registers are 4-bit including result register!

So you have to **throw away** the carry-out from the msb!!

- Have to beware of *overflow* : if the *fixed* number of bits (4, 8, 16, 32, etc.) in a register *cannot represent the result* of the operation
  - *terminology alert*: overflow *does not mean* there was a carry-out from the msb that we lost (though it sounds like that!) – it means simply that the result in the fixed-sized register is incorrect
    - as can be seen from the above examples there are cases when the result is correct even after losing the carry-out from the msb

# Detecting Overflow

- *No overflow* when adding a positive and a negative number
- *No overflow* when subtracting numbers with the same sign
- *Overflow occurs* when the result has “wrong” sign (*verify!*):

Operation	Operand A	Operand B	Result Indicating Overflow
A + B	$\geq 0$	$\geq 0$	$< 0$
A + B	$< 0$	$< 0$	$\geq 0$
A - B	$\geq 0$	$< 0$	$< 0$
A - B	$< 0$	$\geq 0$	$\geq 0$

# Effects of Overflow

- If an *exception* (interrupt) occurs
  - control jumps to predefined address for exception
  - interrupted address is saved for possible resumption
  - Address is stored in Exception Program Counter(EPC) register
- The instruction 'move from system control' mfc0 is used to copy EPC to a general purpose register so that jr can be used subsequently to return back to offending instruction. \$k0 and \$k1 are used for this purpose.
- Don't always want to cause exception on overflow
  - add, addi, sub *cause exceptions* on overflow
  - addu, addiu, subu *do not cause exceptions* on overflow

# Multiply

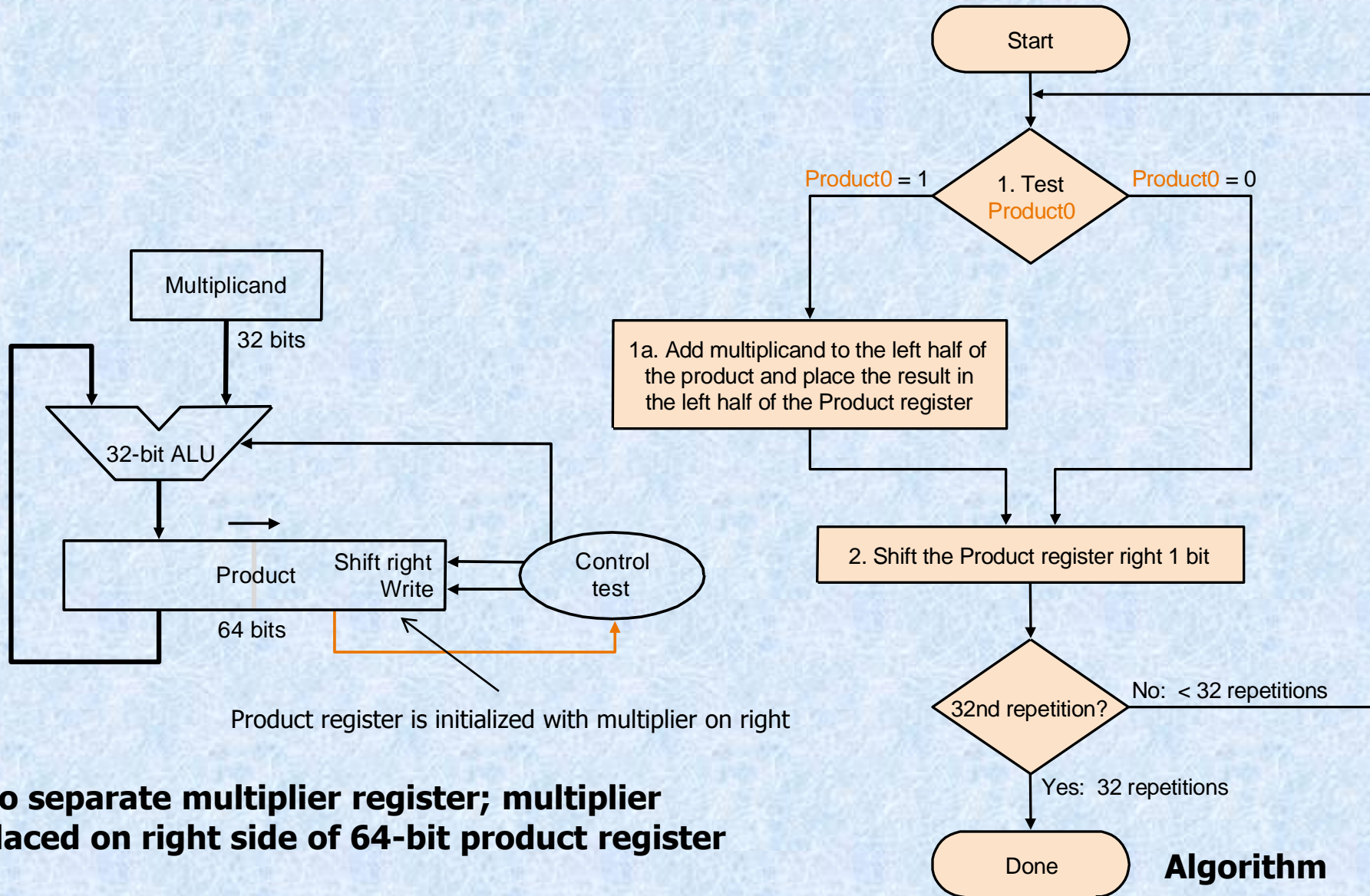
- School shift-add method:

$$\begin{array}{r} \textbf{Multiplicand} \quad 1000 \\ \textbf{Multiplier} \quad \mathbf{x} \quad 1001 \\ \hline \quad \quad \quad 1000 \\ \quad \quad 0000 \\ \quad 0000 \\ 1000 \\ \hline \textbf{Product} \quad \mathbf{01001000} \end{array}$$

- $m$  bits  $\times$   $n$  bits =  $m+n$  bit product
- Binary makes it easy:
  - multiplier bit 1  $\Rightarrow$  copy multiplicand (1  $\times$  multiplicand)
  - multiplier bit 0  $\Rightarrow$  place 0 (0  $\times$  multiplicand)



# Shift-add Multiplier



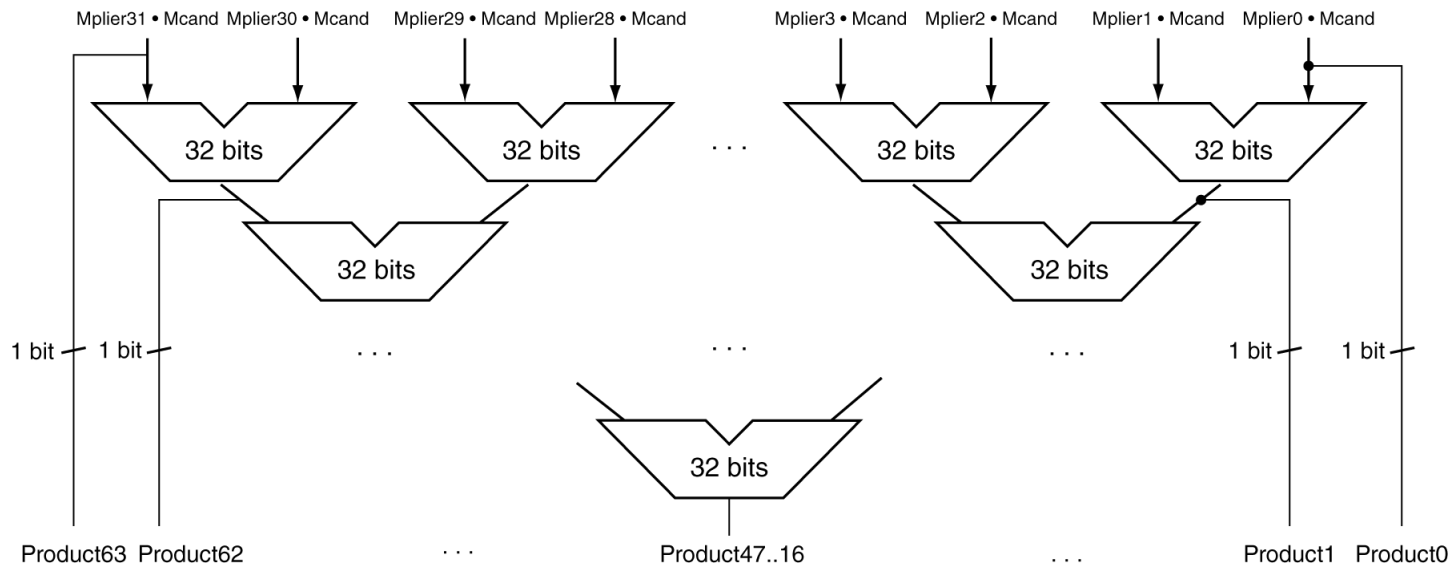
**No separate multiplier register; multiplier placed on right side of 64-bit product register**

# Observations on Multiply

- 2 steps per bit because multiplier & product combined
- What about *signed* multiplication?
  - easiest solution is to make both positive and remember whether to negate product when done, i.e., leave out the sign bit, run for 31 steps, then negate if multiplier and multiplicand have opposite signs
- Booth's Algorithm is an elegant way to multiply signed numbers using same hardware – it also often quicker...

# Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplication performed in parallel

# MIPS Notes

- MIPS provides two 32-bit registers  $H_i$  and  $L_o$  to hold a 64-bit product
- `mult`, `multu` (unsigned) put the product of two 32-bit register operands into  $H_i$  and  $L_o$ : overflow is ignored by MIPS but can be detected by programmer by examining contents of  $H_i$
- `mflo`, `mfhi` moves content of  $H_i$  or  $L_o$  to a general-purpose register

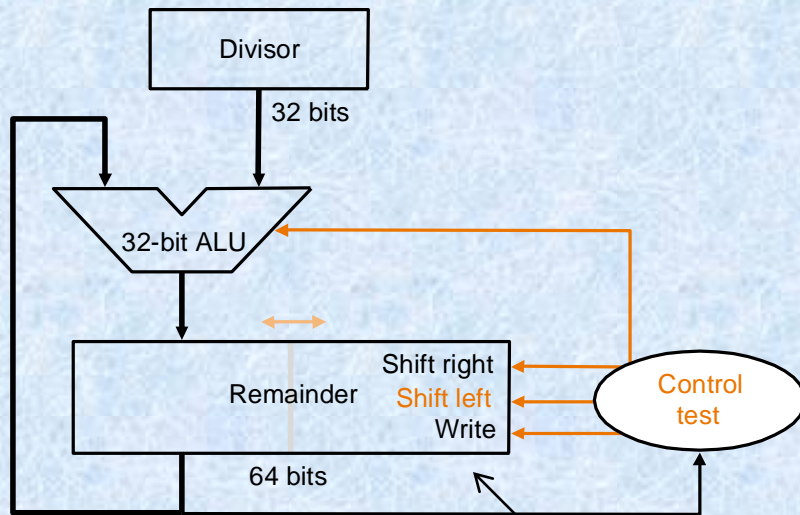
# Divide

$$\begin{array}{r} \text{Divisor } 1000 \quad \overline{) 1001010} \quad \begin{array}{l} \text{Quotient } 1001 \\ \text{Dividend } 1001010 \end{array} \\ \underline{-1000} \phantom{0} \\ 10 \\ 101 \\ 1010 \\ \underline{-1000} \\ 10 \quad \text{Remainder} \end{array}$$

- Junior school method: see how big a multiple of the divisor can be subtracted, creating quotient digit at each step
- Binary makes it easy  $\Rightarrow$  *first*, try  $1 * \text{divisor}$ ; *if too big*,  $0 * \text{divisor}$
- $\text{Dividend} = (\text{Quotient} * \text{Divisor}) + \text{Remainder}$



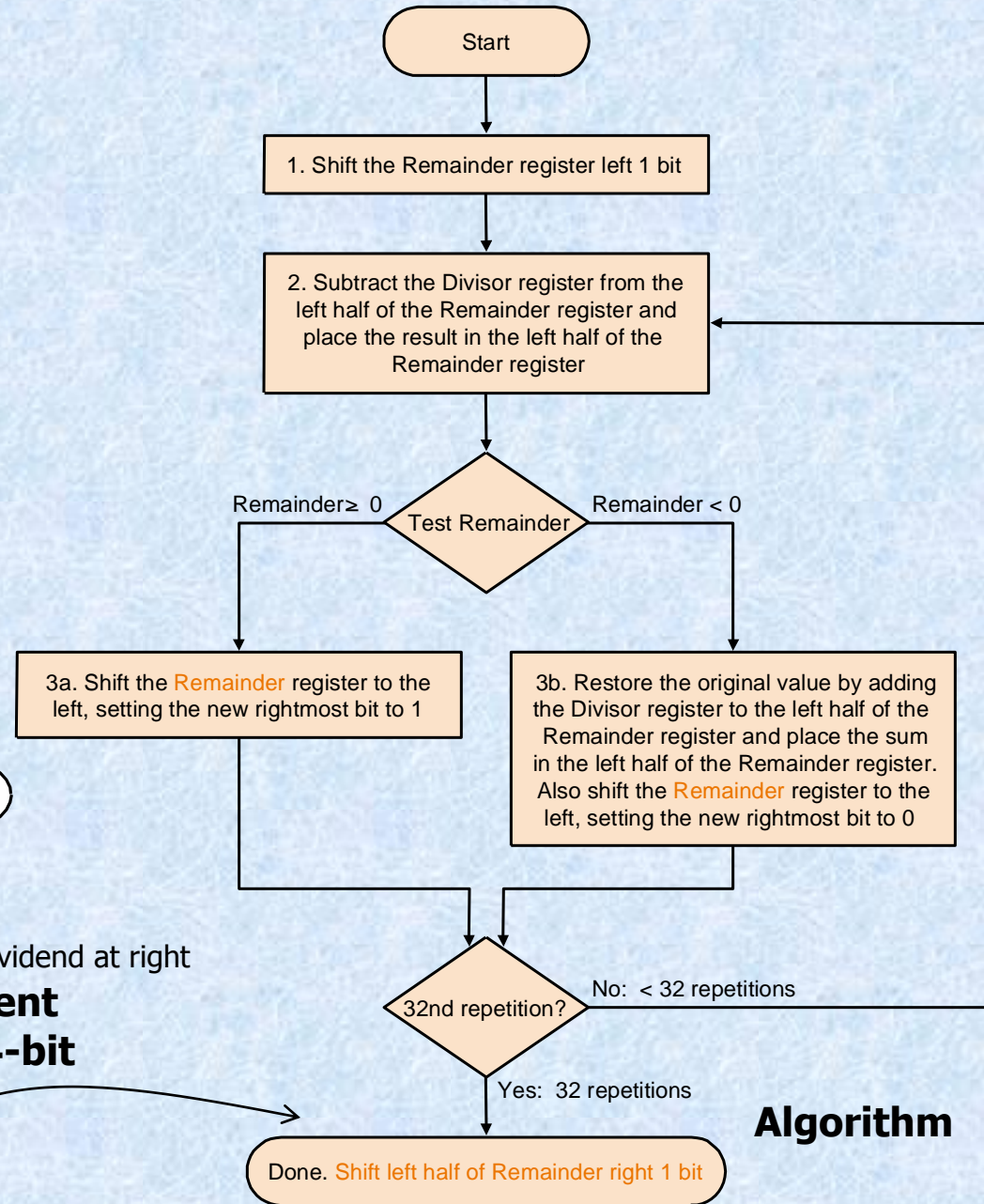
# Divide



Remainder register is initialized with the dividend at right

**No separate quotient register; quotient is entered on the right side of the 64-bit remainder register**

Why this correction step?



**Algorithm**

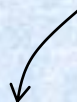
# Observations on Divide

- *Same hardware as Multiply*
- *Signed divide:*
  - make both divisor and dividend positive and perform division
  - negate the quotient if divisor and dividend were of opposite signs
  - make the sign of the remainder match that of the dividend
  - this ensures always
    - $\text{dividend} = (\text{quotient} * \text{divisor}) + \text{remainder}$
    - $-\text{quotient} (x/y) = \text{quotient} (-x/y)$  (e.g.  $7 = 3*2 + 1$  &  $-7 = -3*2 - 1$ )

# MIPS Notes

- `div` (signed), `divu` (unsigned), with two 32-bit register operands, divide the contents of the operands and put remainder in `Hi` register and quotient in `Lo`; overflow is ignored in both cases
- MIPS s/w must check the divisor to discover division by zero as well as overflow.

# Floating Point

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers (in absolute value), e.g., .000000000023
  - very large numbers (in absolute value) , e.g.,  $-3.15576 * 10^{46}$
- Representation:
  - *scientific*: sign, exponent, significand form:  binary point
    - $(-1)^{\text{sign}} * \text{significand} * 2^{\text{exponent}}$ . E.g.,  $-101.001101 * 2^{111001}$
  - more bits for *significand* gives more accuracy
  - more bits for *exponent* increases range
  - if  $1 \leq \text{significand} < 10_{\text{two}} (=2_{\text{ten}})$  then number is **normalized**, **except for** number 0 which is normalized to significand 0
    - E.g.,  $-101.001101 * 2^{111001} = -1.01001101 * 2^{111011}$  (normalized)

# IEEE 754 Floating-point Standard

- IEEE 754 floating point standard:
  - single precision: one word

31	bits 30 to 23	bits 22 to 0
sign	8-bit exponent	23-bit significand

- double precision: two words

31	bits 30 to 20	bits 19 to 0
sign	11-bit exponent	upper 20 bits of 52-bit significand

bits 31 to 0		
lower 32 bits of 52-bit significand		



# IEEE 754 Floating-point Standard

- Sign bit is 0 for positive numbers, 1 for negative numbers
- Number is assumed normalized and leading 1 bit of significand left of binary point (for non-zero numbers) is *assumed* and not shown
  - e.g., significand 1.1001... is represented as 1001...,
  - **exception** is number 0 which is represented as all 0s
  - for other numbers:  
$$\text{value} = (-1)^{\text{sign}} * (1 + \text{significand}) * 2^{\text{exponent value}}$$
- Exponent is *biased* to make sorting easier
  - all 0s is smallest exponent, all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - therefore, for non-0 numbers:  
$$\text{value} = (-1)^{\text{sign}} * (1 + \text{significand}) * 2^{\overbrace{(\text{exponent} - \text{bias})}^{\text{equals exponent value}}}$$

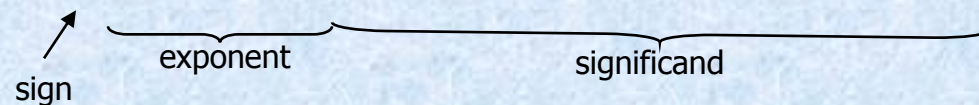
# IEEE 754 Floating-point Standard

- **Special treatment of 0:**

- if exponent is all 0 and significand is all 0, then the value is 0 (sign bit may be 0 or 1)

- *Example* : Represent  $-0.75_{\text{ten}}$  in IEEE 754 single precision

- decimal:  $-0.75 = -3/4 = -3/2^2$
- binary:  $-11/100 = -.11 = -1.1 \times 2^{-1}$
- IEEE single precision floating point exponent = bias + exponent value  
 $= 127 + (-1) = 126_{\text{ten}} = 01111110_{\text{two}}$
- IEEE single precision: 10111111010000000000000000000000



# Floating-Point Example

- What number is represented by the single-precision float

11000000101000...00

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

# IEEE 754 Standard Encoding

Single Precision		Double Precision		Object Represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0 (zero)
0	Non-zero	0	Non-zero	$\pm$ Denormalized number
1-254	Anything	1-2046	Anything	$\pm$ Floating-point number
255	0	2047	0	$\pm$ Infinity
255	Non-zero	2047	Non-zero	NaN (Not a Number)

- **NaN** : (infinity – infinity), or 0/0
- **Denormalized number** =  $(-1)^{\text{sign}} * 0.f * 2^{1-\text{bias}}$

# 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  $\approx 2.0$
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$



# Double-Precision Range

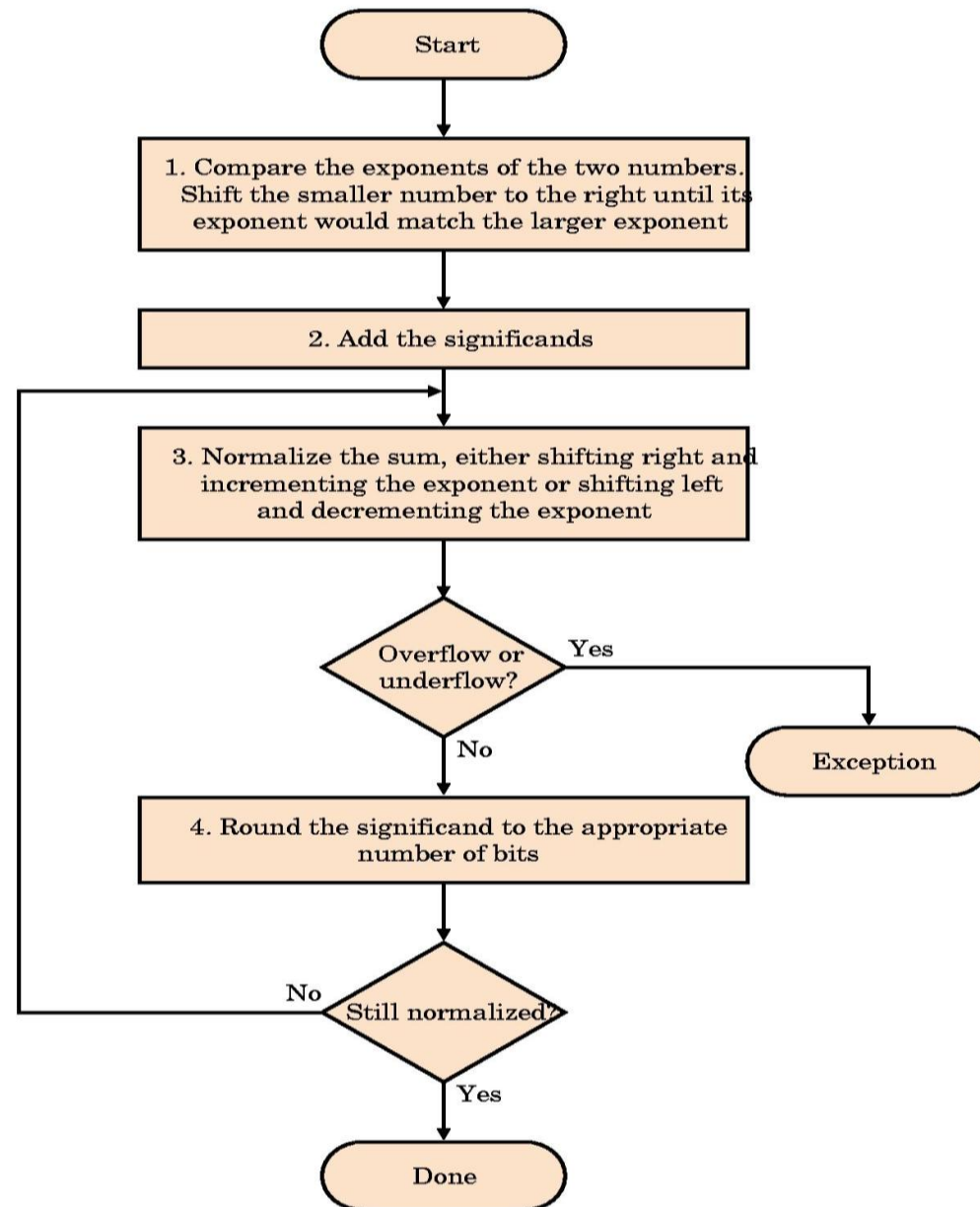
- Exponents 0000...00 and 1111...11 reserved
- Smallest value
  - Exponent: 000000000001  
 $\Rightarrow$  actual exponent =  $1 - 1023 = -1022$
  - Fraction: 000...00  $\Rightarrow$  significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 111111111110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

# Floating point addition

- **Make both exponents the same**
  - Find the number with the smaller one
  - Shift its mantissa to the right until the exponents match
    - Must include the implicit 1 (1.M)
- **Add the mantissas**
- **Choose the largest exponent**
- **Put the result in normalized form**
  - Shift mantissa left or right until in form 1.M
  - Adjust exponent accordingly
- **Handle overflow or underflow if necessary**
- **Round**
- **Renormalize if necessary if rounding produced an unnormalized result**

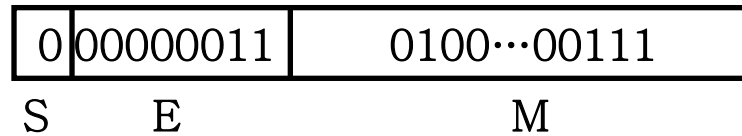
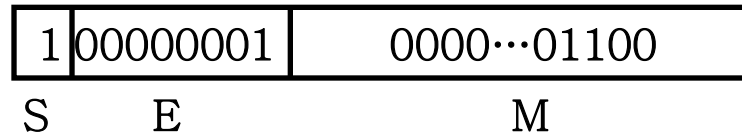
# Floating point addition

## ■ Algorithm



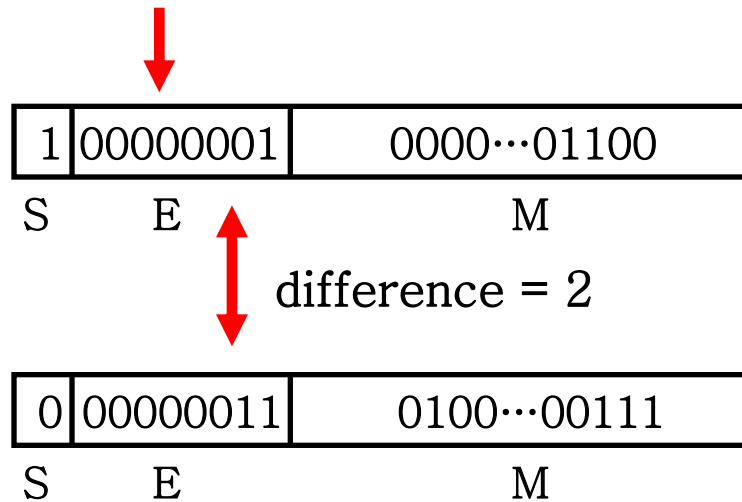
# Floating point addition example

## ■ Initial values



# Floating point addition example

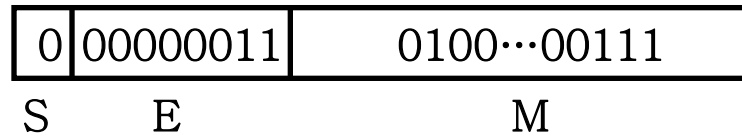
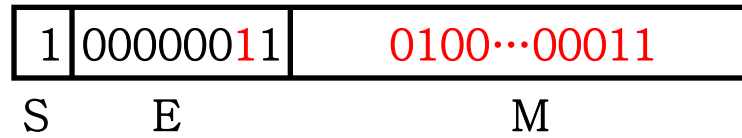
- Identify smaller E and calculate E difference





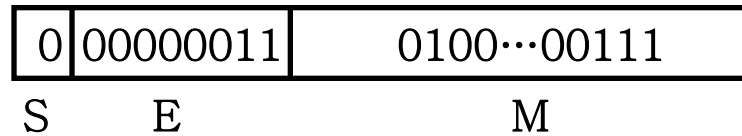
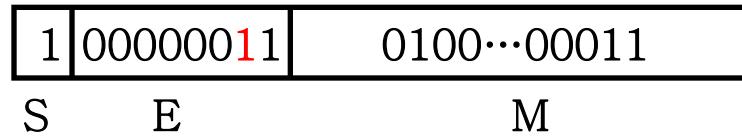
# Floating point addition example

- Shift smaller M right by E difference

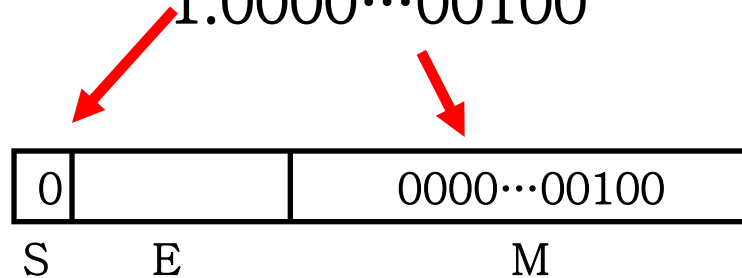


# Floating point addition example

## ■ Add mantissas

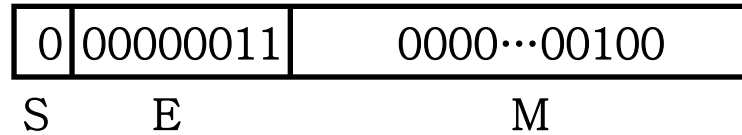


$$-0.0100\dots00011 + 1.0100\dots00111 = 1.0000\dots00100$$



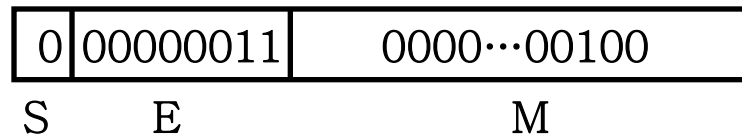
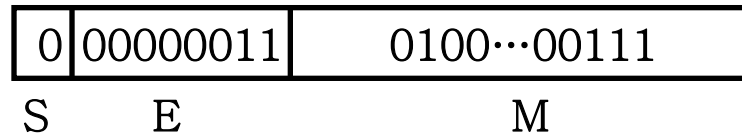
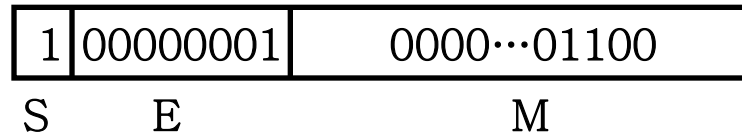
# Floating point addition example

- **Normalize the result by shifting (already normalized)**



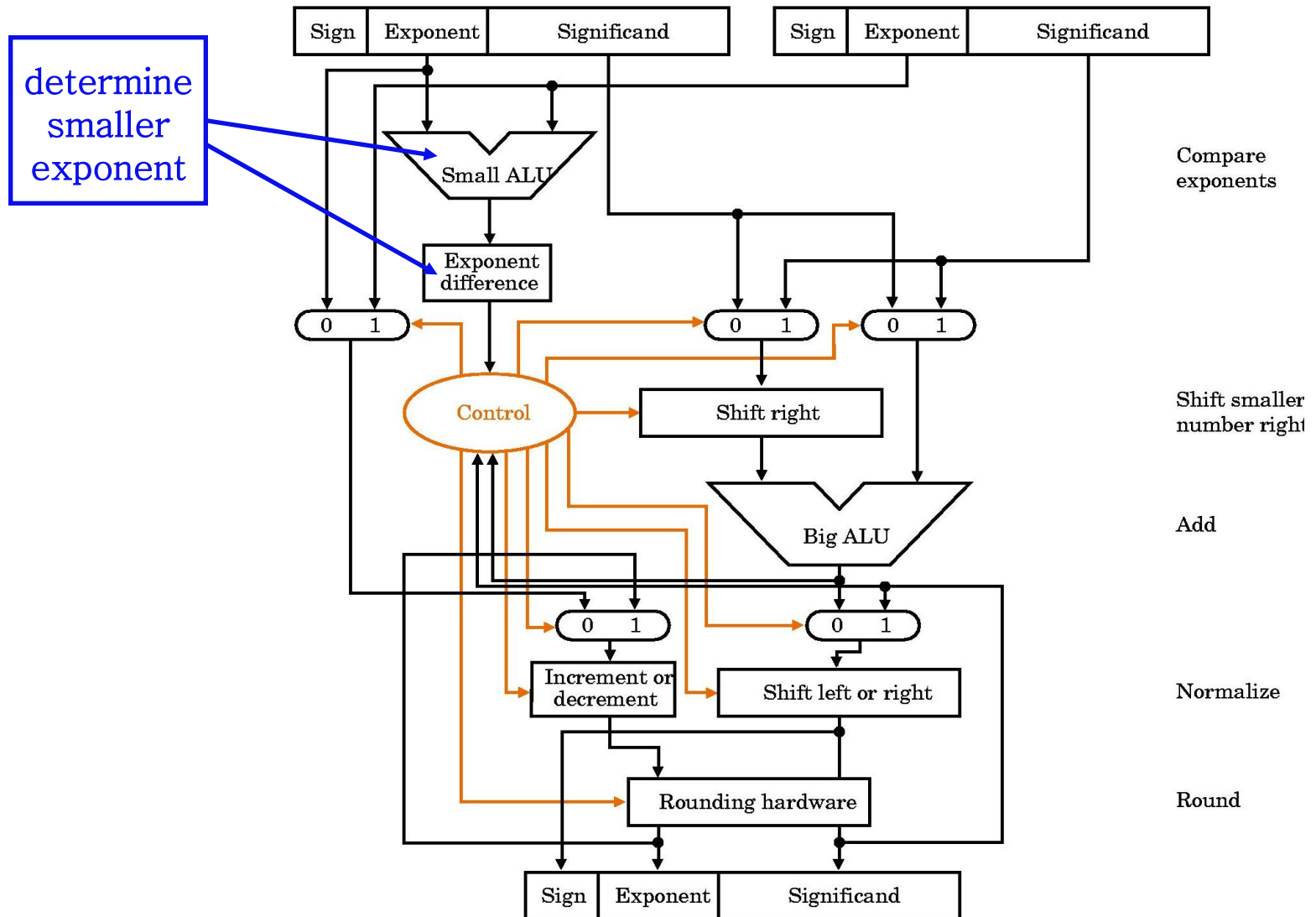
# Floating point addition example

## ■ Final answer



# Floating point addition

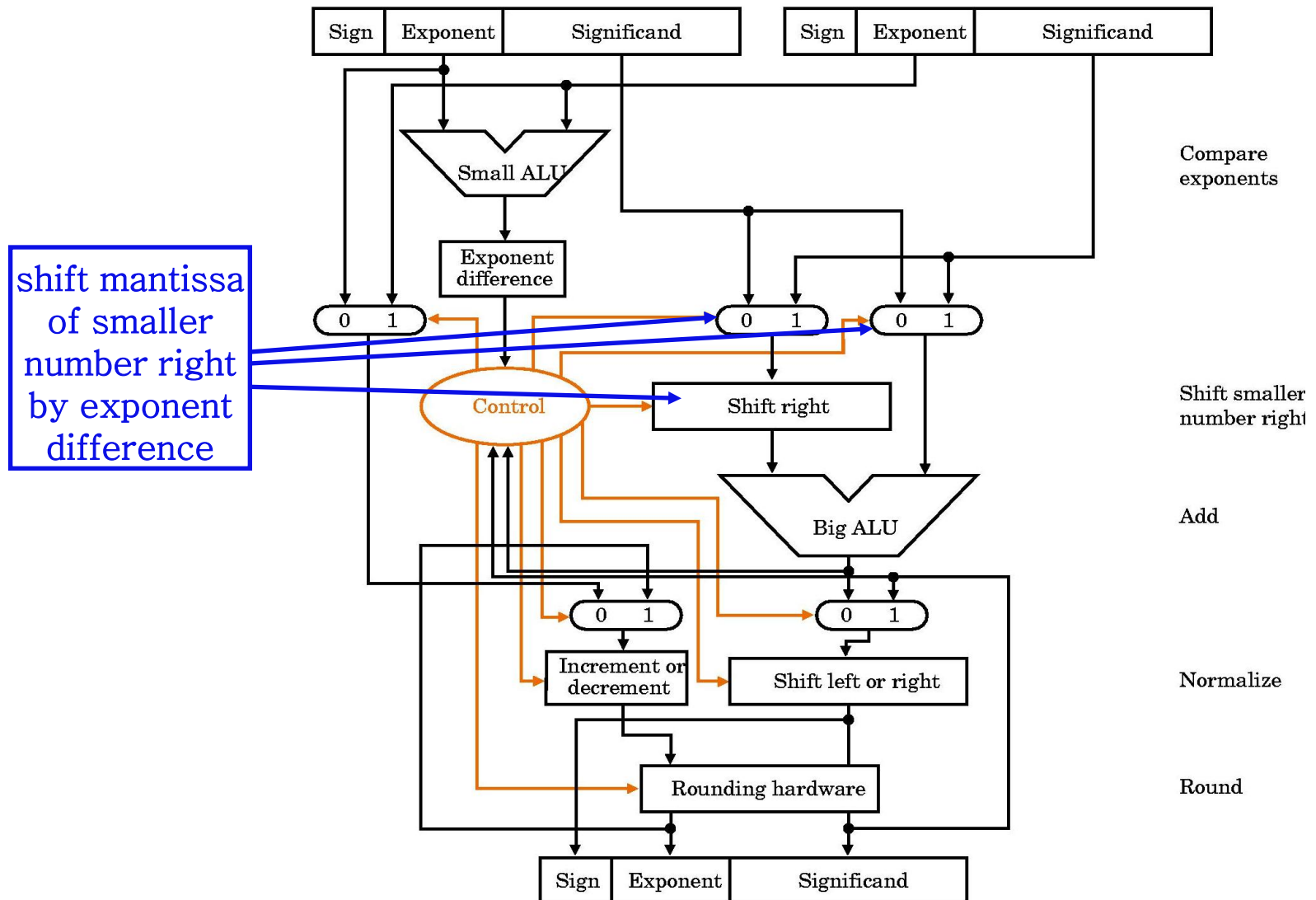
## ■ Hardware design





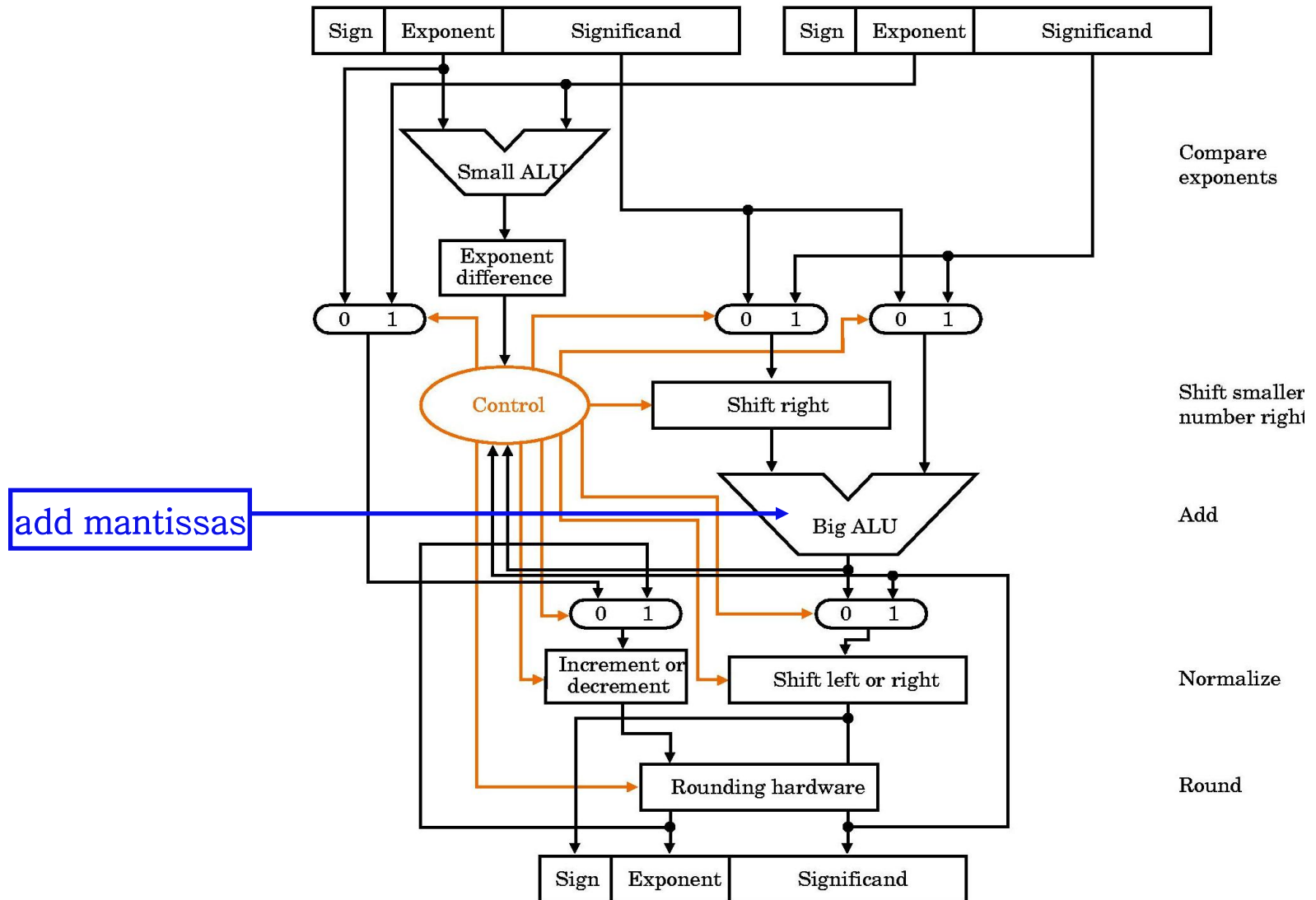
# Floating point addition

## ■ Hardware design



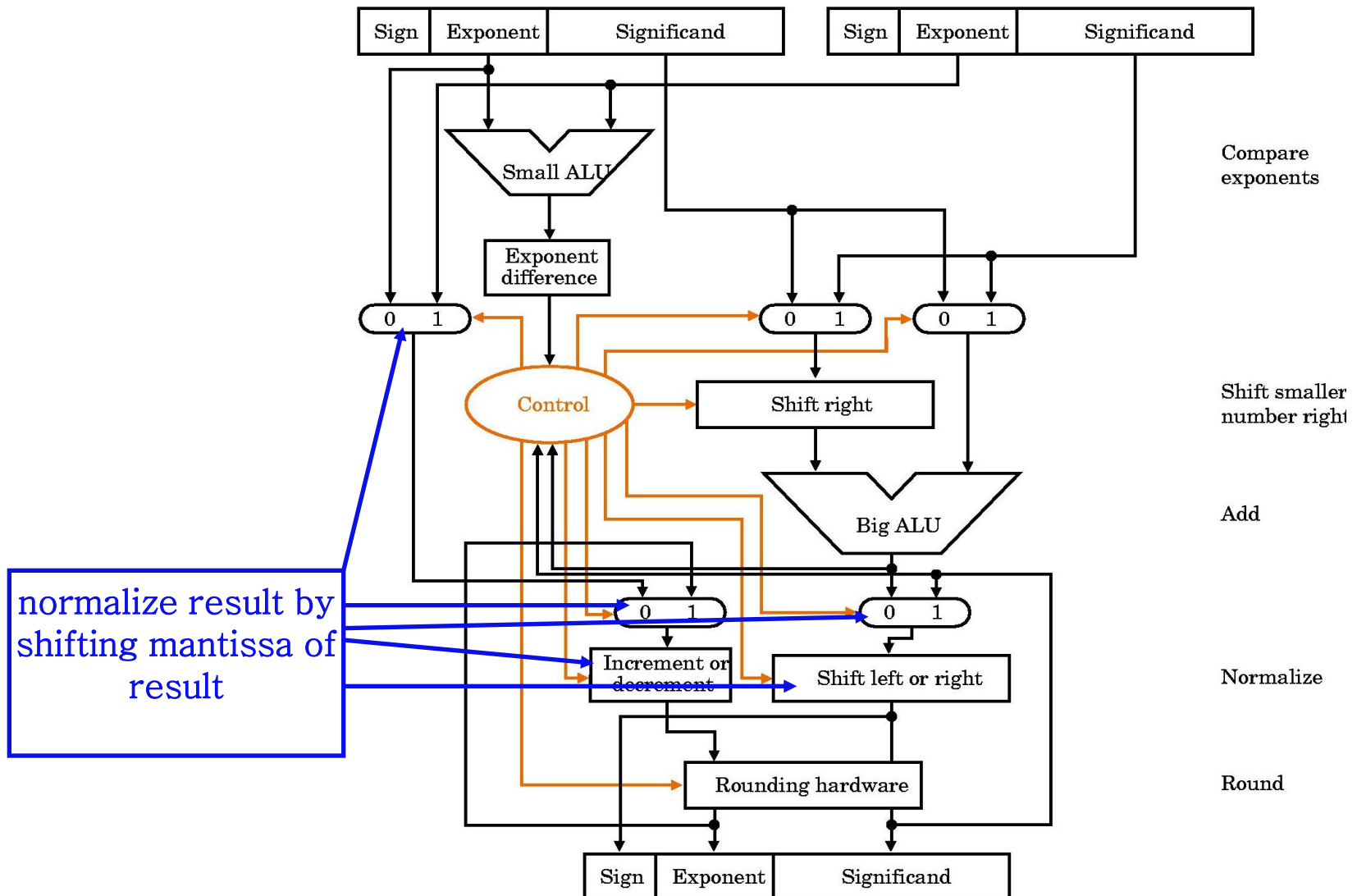
# Floating point addition

## ■ Hardware design



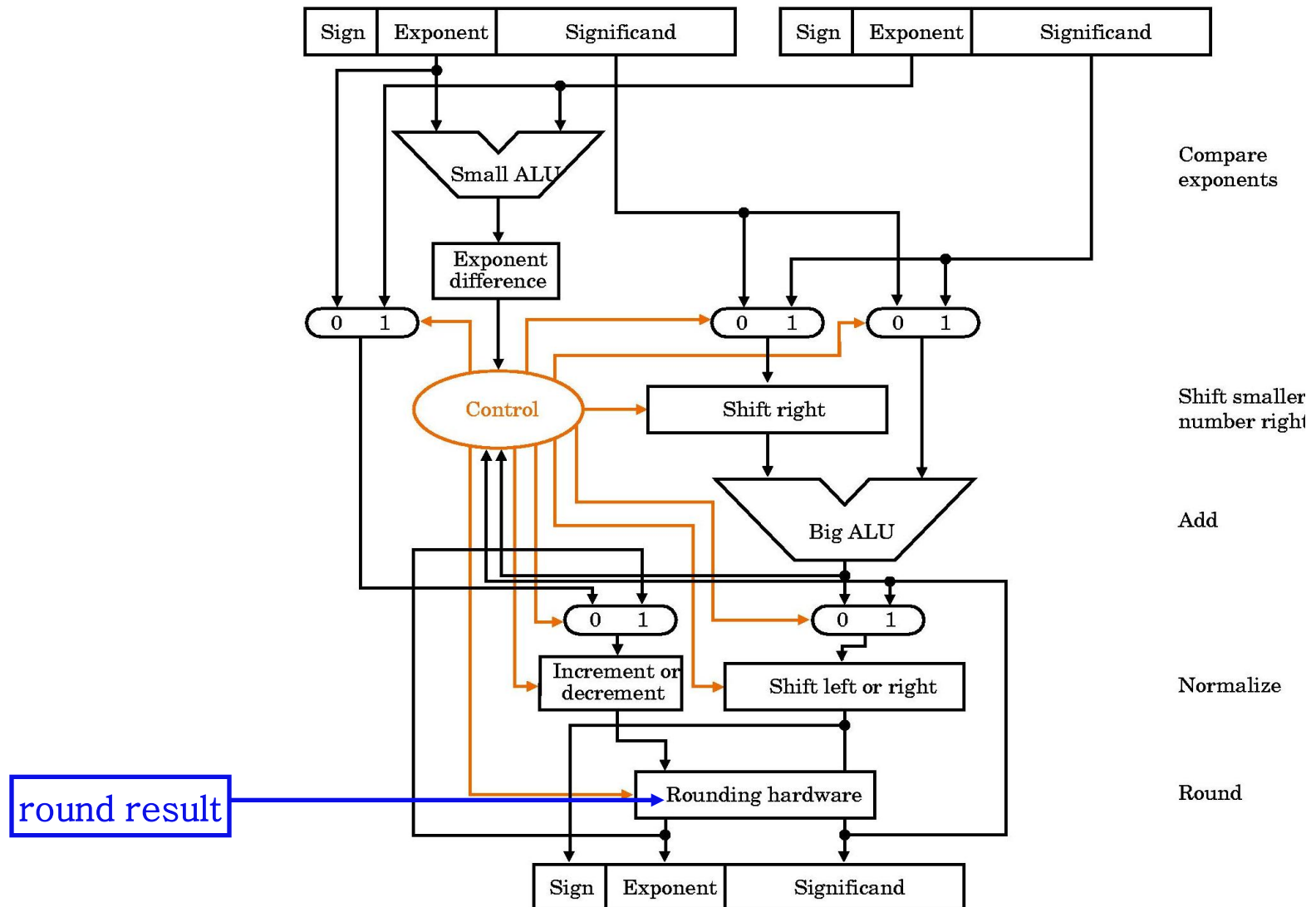
# Floating point addition

## ■ Hardware design



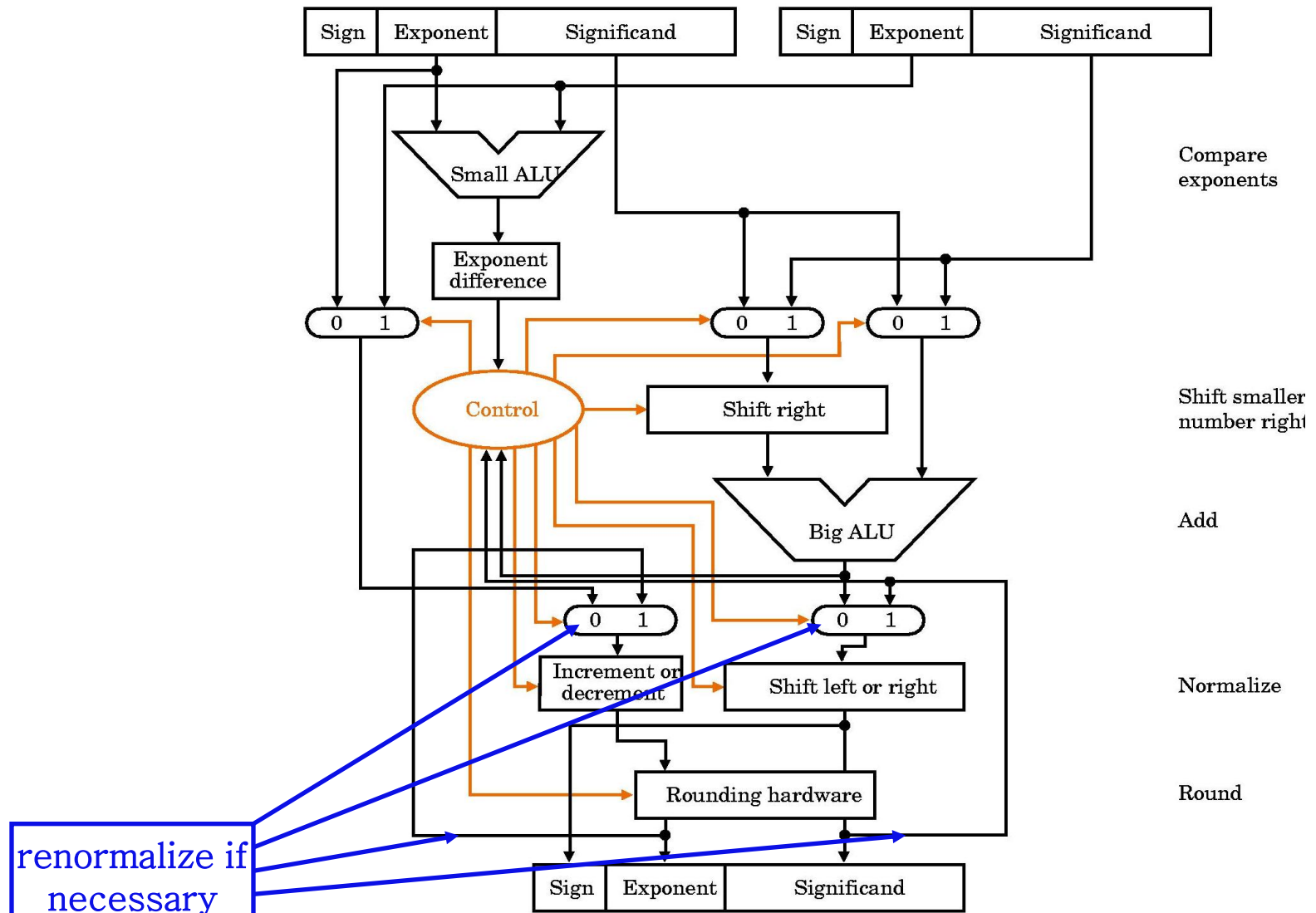
# Floating point addition

## ■ Hardware design



# Floating point addition

## ■ Hardware design





# FP Adder Hardware

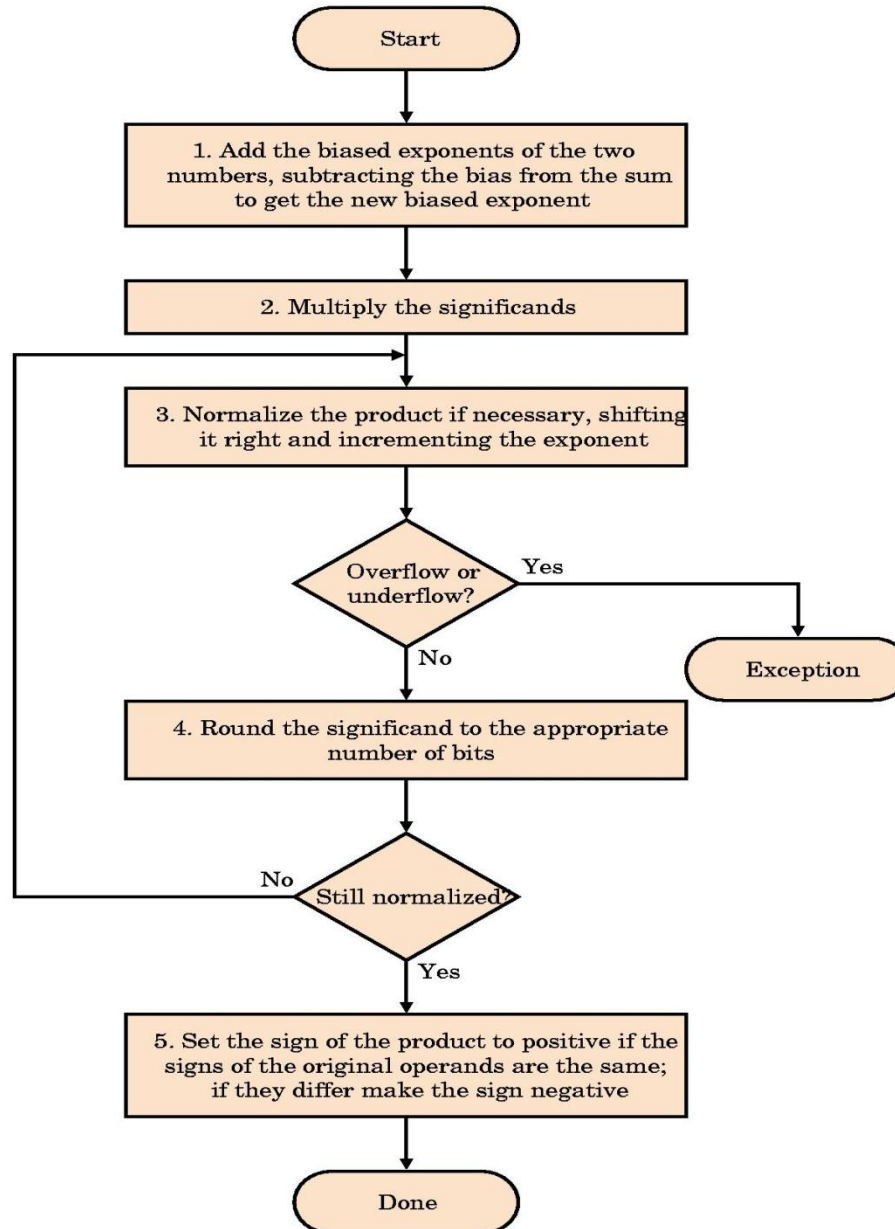
- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

# Floating point multiply

- **Add the exponents and subtract the bias from the sum**
  - Example:  $(5+127) + (2+127) - 127 = 7+127$
- **Multiply the mantissas**
- **Put the result in normalized form**
  - Shift mantissa left or right until in form 1.M
  - Adjust exponent accordingly
- **Handle overflow or underflow if necessary**
- **Round**
- **Renormalize if necessary if rounding produced an unnormalized result**
- **Set  $S=0$  if signs of both operands the same,  $S=1$  otherwise**

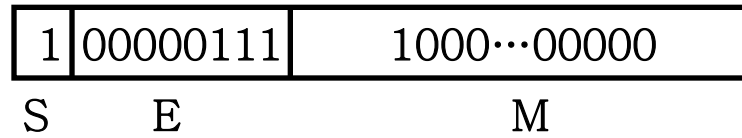
# Floating point multiply

## ■ Algorithm

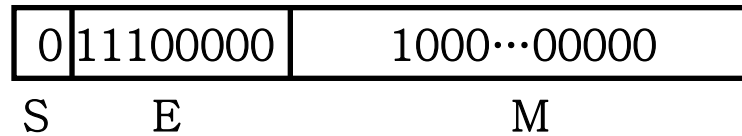


# Floating point multiply example

## ■ Initial values



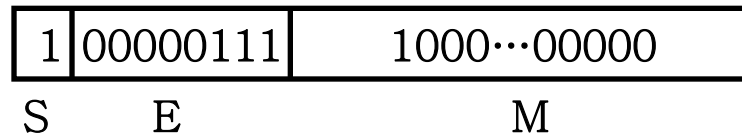
$$-1.5 \times 2^{7-127}$$



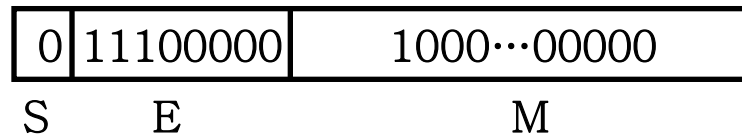
$$1.5 \times 2^{224-127}$$

# Floating point multiply example

## ■ Add exponents



**$-1.5 \times 2^{7-127}$**



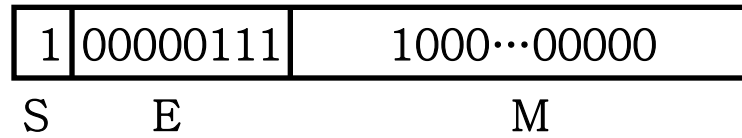
**$1.5 \times 2^{224-127}$**

$$00000111 + 11100000 = 11100111 \text{ (231)}$$

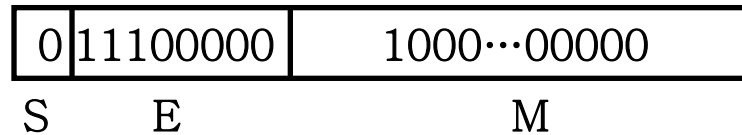


# Floating point multiply example

## ■ Subtract bias

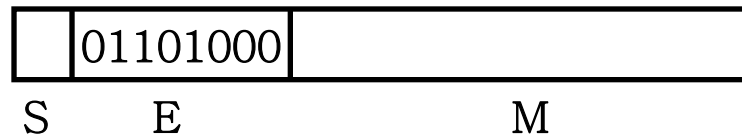


**$-1.5 \times 2^{7-127}$**



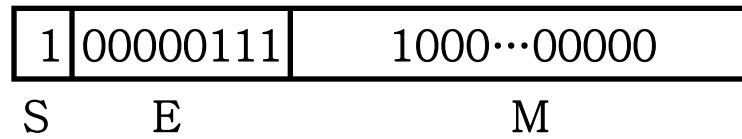
**$1.5 \times 2^{224-127}$**

$$11100111 \text{ (231)} - 01111111 \text{ (127)} = 01101000 \text{ (104)}$$

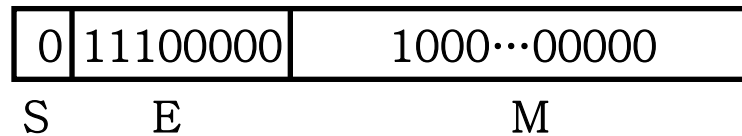


# Floating point multiply example

## ■ Multiply the mantissas

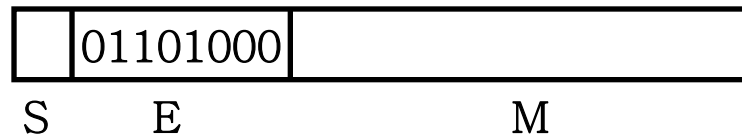


**$-1.5 \times 2^{7-127}$**



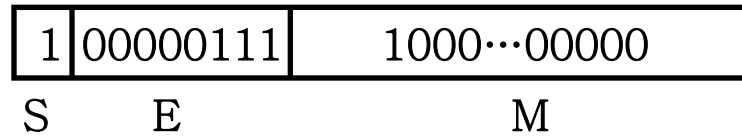
**$1.5 \times 2^{224-127}$**

$$1.1000\cdots \times 1.1000\cdots = 10.01000\cdots$$

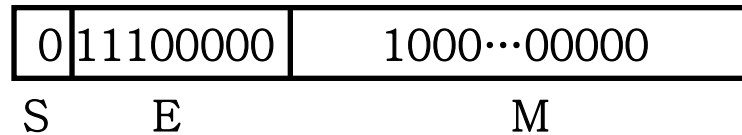


# Floating point multiply example

- Normalize by shifting 1.M right one position and adding one to E

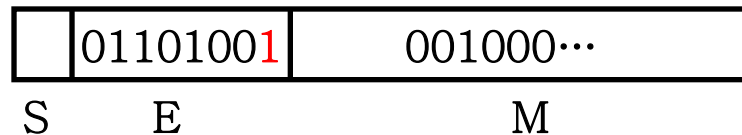


$$-1.5 \times 2^{7-127}$$



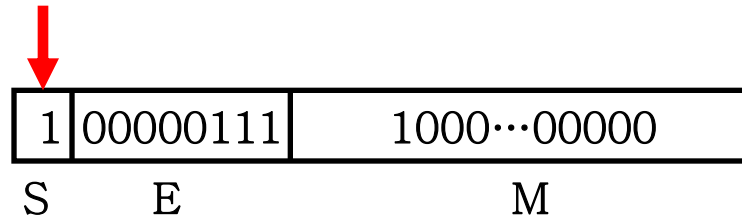
$$1.5 \times 2^{224-127}$$

10.01000...  $\Rightarrow$  1.001000...

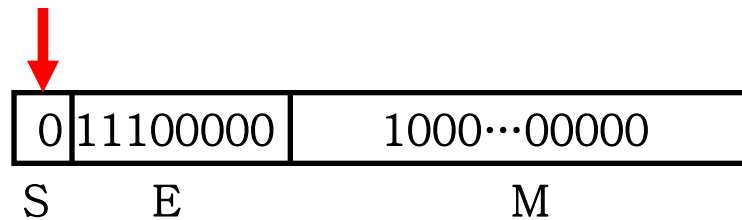


# Floating point multiply example

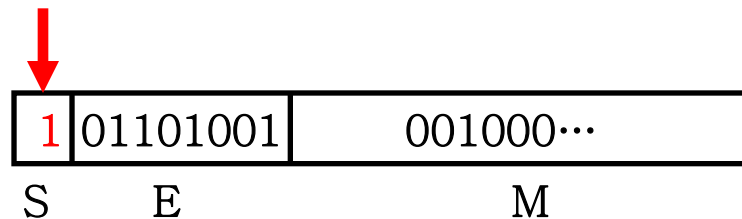
- Set **S=1** since signs are different



$$-1.5 \times 2^{7-127}$$



$$1.5 \times 2^{224-127}$$

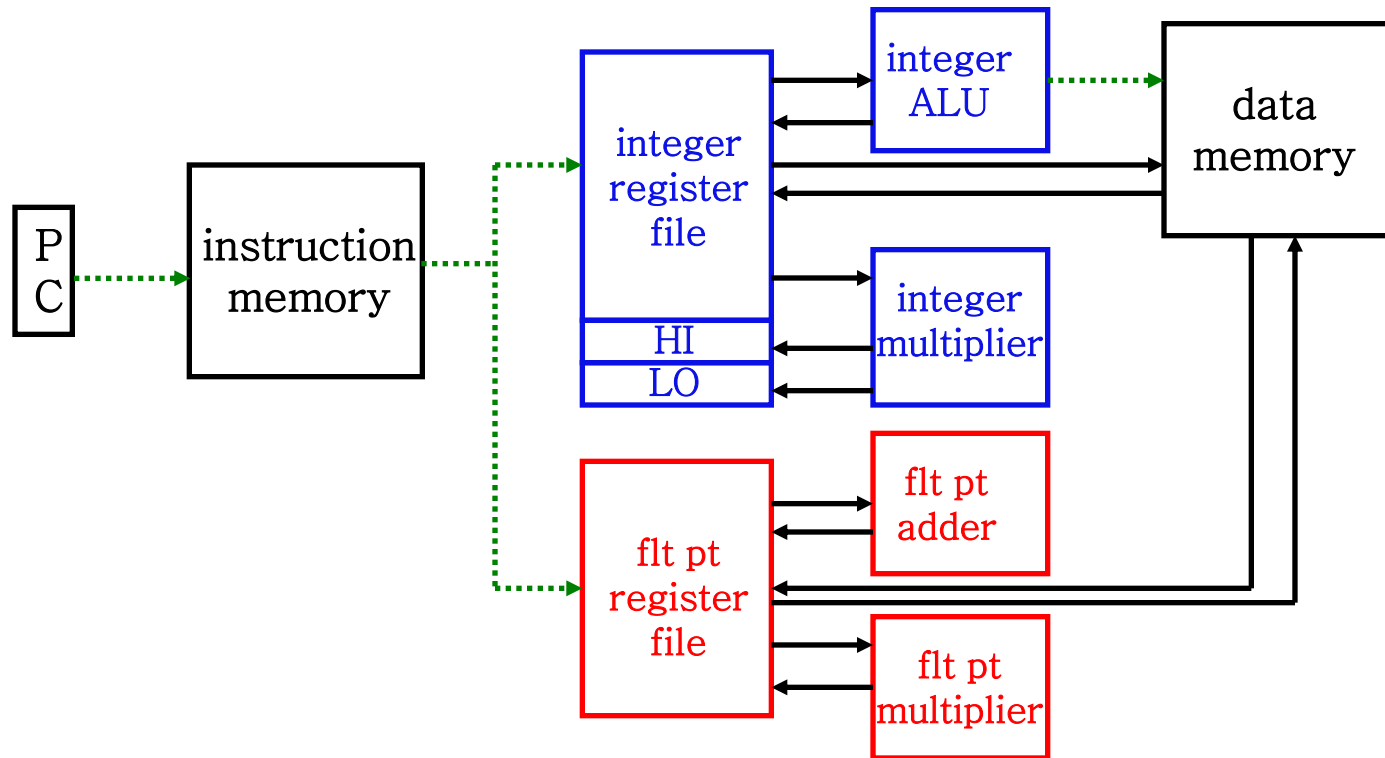


$$-1.125 \times 2^{105-127}$$

# 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
  - $\text{FP} \leftrightarrow \text{integer}$  conversion
- Operations usually takes several cycles
  - Can be pipelined

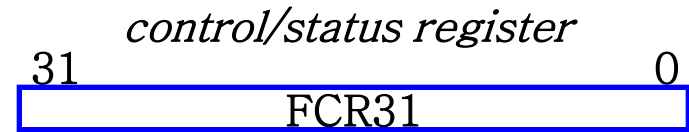
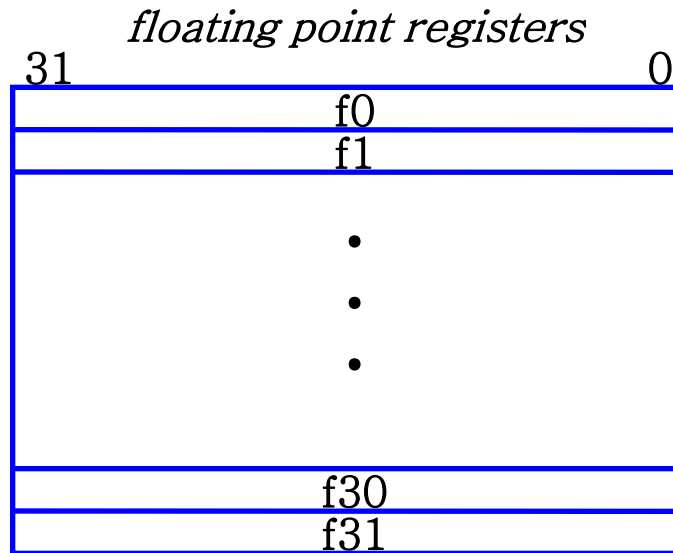
# Integer and floating point revisited



- **Integer ALU handles add, subtract, logical, set less than, equality test, and effective address calculations**
- **Integer multiplier handles multiply and divide**
  - HI and LO registers hold result of integer multiply and divide



# MIPS floating point registers



## ■ 32 32-bit FPRs

- 16 64-bit registers (32-bit register pairs) for dp floating point
- Software conventions for their usage (as with GPRs)

## ■ Control/status register

- Status of compare operations, sets rounding mode, exceptions

## ■ Implementation/revision register

- Identifies type of CPU and its revision number

# FP Instructions in MIPS

- 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, ...
- 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

# MIPS floating point instruction overview

---

- **Operate on single and double precision operands**
- **Computation**
  - Add, sub, multiply, divide, sqrt, absolute value, negate
- **Load and store**
  - Integer register read for EA calculation
  - Data to be loaded or stored in fp register file
- **Move between registers**
- **Convert between different formats**
- **Comparison instructions**
- **Branch instructions**

# MIPS Floating Point

- MIPS has a *floating point coprocessor* (numbered 1) with thirty-two 32-bit registers \$f0 - \$f31. Two of these are required to hold doubles.
- Floating point *arithmetic*: `add.s` (single addition), `add.d` (double addition), `sub.s`, `sub.d`, `mul.s`, `mul.d`, `div.s`, `div.d`
- Floating point *comparison*: `c.x.s` (single), `c.x.d` (double), where `x` may be `eq`, `neq`, `lt`, `le`, `gt`, `ge`
- Floating point comparison sets a bit (`cond`) to true or false
- Floating point branch, true (`bc1t`) and branch, false (`bc1f`)

`lwc1 $f2, 100($s2)`

`ldc1 $f2, 100($s2)`

`swc1 $f2, 100($s2)`

`sdc1 $f2, 100($s2)`

➔ Base registers are the integer registers

➔ A double precision register is an even-odd pair of single precision registers using the even register number as its name

# Floating Point Complexities

- In addition to *overflow* we can have *underflow* (number too small)
- *Accuracy* is the problem with both overflow and underflow because we have only a finite number of bits to represent numbers that may actually require arbitrarily many bits
  - limited precision  $\Rightarrow$  rounding  $\Rightarrow$  rounding error
  - IEEE 754 keeps *two extra bits*, *guard and round*
  - four rounding modes
  - positive divided by zero yields *infinity*
  - zero divide by zero yields *not a number*
  - other complexities
- Implementing the standard can be tricky



# Rounding

---

- Fp arithmetic operations may produce a result with more digits than can be represented in 1.M
- The result must be *rounded* to fit into the available number of M positions
- Tradeoff of hardware cost (keeping extra bits) and speed versus accumulated rounding error

# Rounding

## ■ Guard, Round bits for intermediate addition

- $2.56 \times 10^0 + 2.34 \times 10^2 = 0.0256 \times 10^2 + 2.34 \times 10^2 = 2.3656 \times 10^2$
- 5: guard bit
- 6: round bit
- 00~49: round down, 51~99: round up, 50: tie-break
- Result:  $2.37 \times 10^2$
- Without guard and round bit
  - $0.02 \times 10^2 + 2.34 \times 10^2 = 2.36 \times 10^2$

# Rounding

---

- **In binary, an extra bit of 1 is halfway in between the two possible representations**

1.001 (1.125) is halfway between 1.00 (1) and 1.01 (1.25)

1.101 (1.625) is halfway between 1.10 (1.5) and 1.11 (1.75)

# IEEE 754 rounding modes

## ■ Truncate

- 1.00100 -> 1.00

## ■ Round up to the next value

- 1.00100 -> 1.01

## ■ Round down to the previous value

- 1.00100 -> 1.00

## ■ Round-to-nearest-even

- Rounds to the even value (the one with an LSB of 0)

- 1.**00**100 -> 1.00

- 1.**01**100 -> 1.10

- Produces zero average bias