Chapter 3

Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Integer Addition

- Example: 7 + 6

add extra 0s to show this is positive number

	(0)	(0)	(1)	(1)	(0)	(Carries)
. . .	0	0	0	1	1	1
. . .	0	0	0	1	1	0
. . .	(0) 0	(0) 0	(0) 1	(1) 1	(1) 0	(0) 1

- Overflow if result out of range
  - Adding positive (+) and negative (–) operands, no overflow
  - Adding two positive (+ +) operands
    - Overflow if result sign is 1 (negative)
  - Adding two negative (-- –) operands
    - Overflow if result sign is 0 (positive)

6 = 110

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

+7:		0000 0000 … 0000 0111
–6:		1111 1111 … 1111 1010
+1:	~~1~~	0000 0000 … 0000 0001

drop the overflow sin abs(7) > abs(-6).

- Overflow if result out of range
  - Subtracting two positive (+ +) or two negative (– --) operands, no overflow
  - Subtracting positive (+) from negative (–) operand
    - Overflow if result sign is 0 (positive)
  - Subtracting negative (–) from positive (+) operand
    - Overflow if result sign is 1 (negative)

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
- Other languages (e.g., Ada, Fortran) has overflow
- MIPS has overflow, require raising an exception on overflow
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Multiplication

- Start with long-multiplication approach

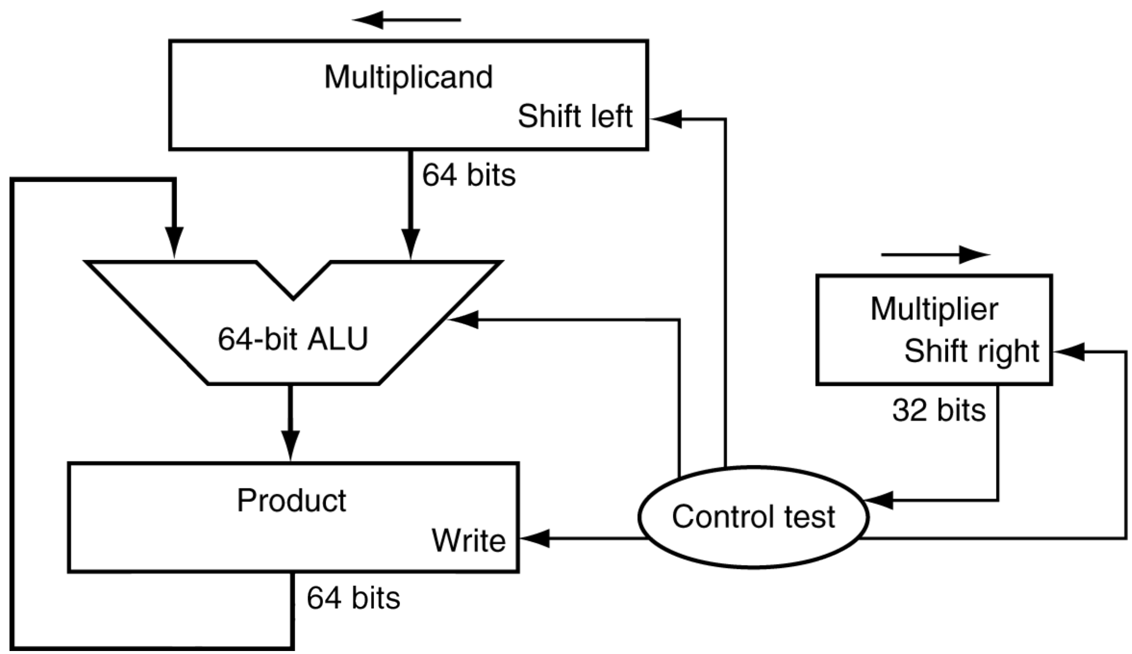
multiplicand

multiplier

product

```
        1000
  ×     1001
  ----------
        1000
       0000
      0000
     1000
  ----------
     1001000
```

Length of product is the sum of operand lengths



Multiplication Hardware

Multiplicand	Multiplier	Product
1000	0011	0000
		+ 1000
		1000
10000	0001.	+ 10000
100000	0000	1. 1000

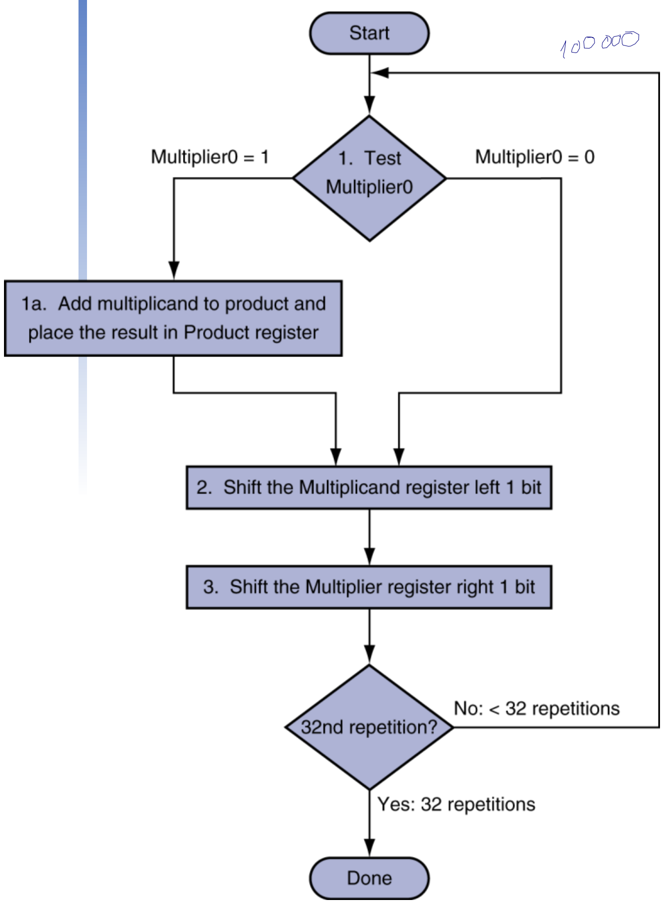
$8 \times 3 = 24$

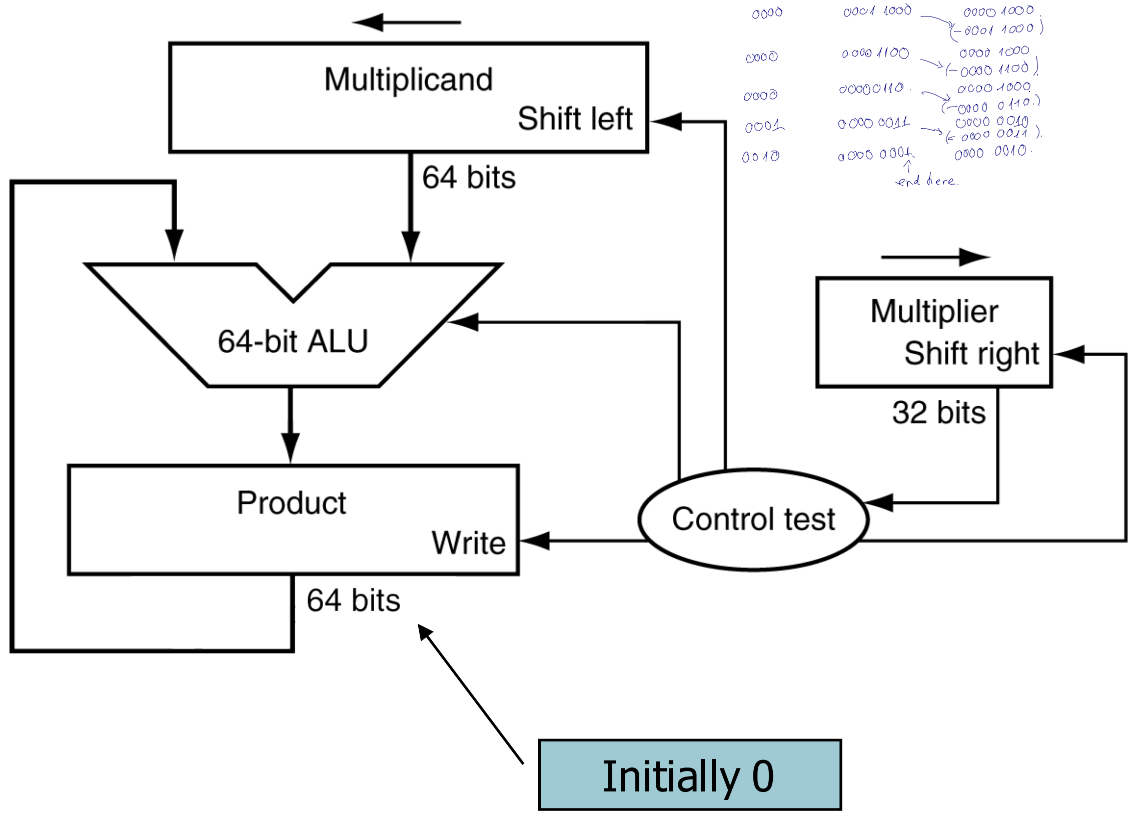
Multiplicand	multiplier	product	
0011	00001000	0000 0000.	
		+ 0000 1000	8
0001.	00010000	00001000	8
		+ 00010000	16
0000	00100000	00011000 ← final product	

$8 \div 3$

Quotient	Divisor	Remainder
0000	0011 0000	0000 1000 (-0011 0000.)
0000	0001 1000	0000 1000 (-0001 1000)
0000	0000 1100	0000 1000 (-0000 1100)
0000	00000110.	0000 1000 (-0000 0110.)
0001	0000 0011	0000 0010 (-0000 0011).
0010	0000 0001.	0000 0010.

end here.





MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt / multu rs, rt`
    - 64-bit product in HI/LO ← special registers not accesible by name but rather by mfhi or mflo
  - `mfhi rd / mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - similar to mult but only use least-significan 32 bit (i.e LO)
    - Least-significant 32 bits of product –> rd

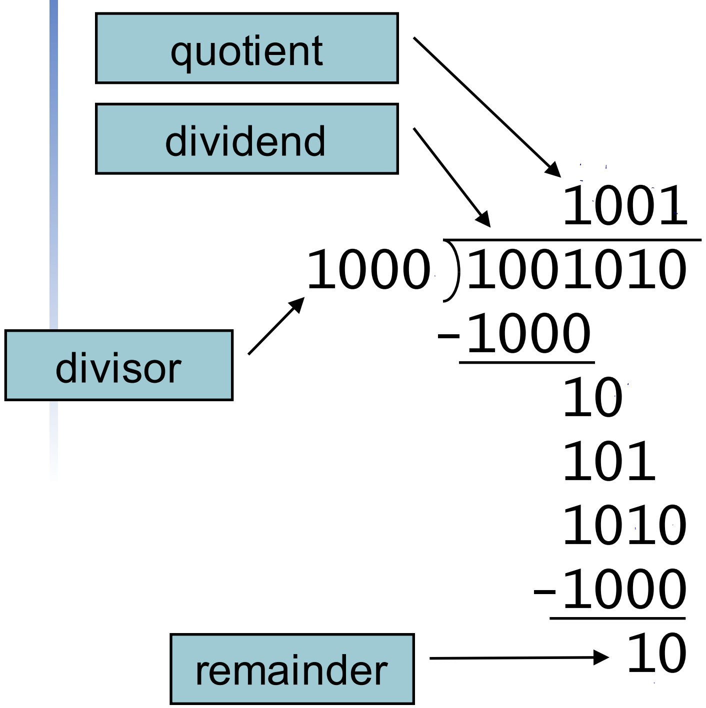
Division

Dividend. Divisor Quotient Remainder.
8 ÷ 3 = ? ?
left.

	Quotient	Divisor.	Remainder	
Step 0:	0000	0011 0000	0000 1000	
			(−0011 0000)	# minus the divisor.
Step 1:			~~1011 1000~~	(negative, reuse perevious value.)
	0000	0001 1000	0000 1000	# shift Divisor 1 bit to the right.
			(−0001 1000)	Quotient 1 ——— left.
	0000	0000 1100	0000 1000	# —— Divisor ——— right
			(−0000 1100)	
	0000	0000 0110	0000 1000	—— Quotient ——— left.
			(−0000 0110)	
	0001	0000 0011	0000 0010	← keep. # add quotient with 1.
	0010	0000 0001	0000 0010	
			(−0000 0001	

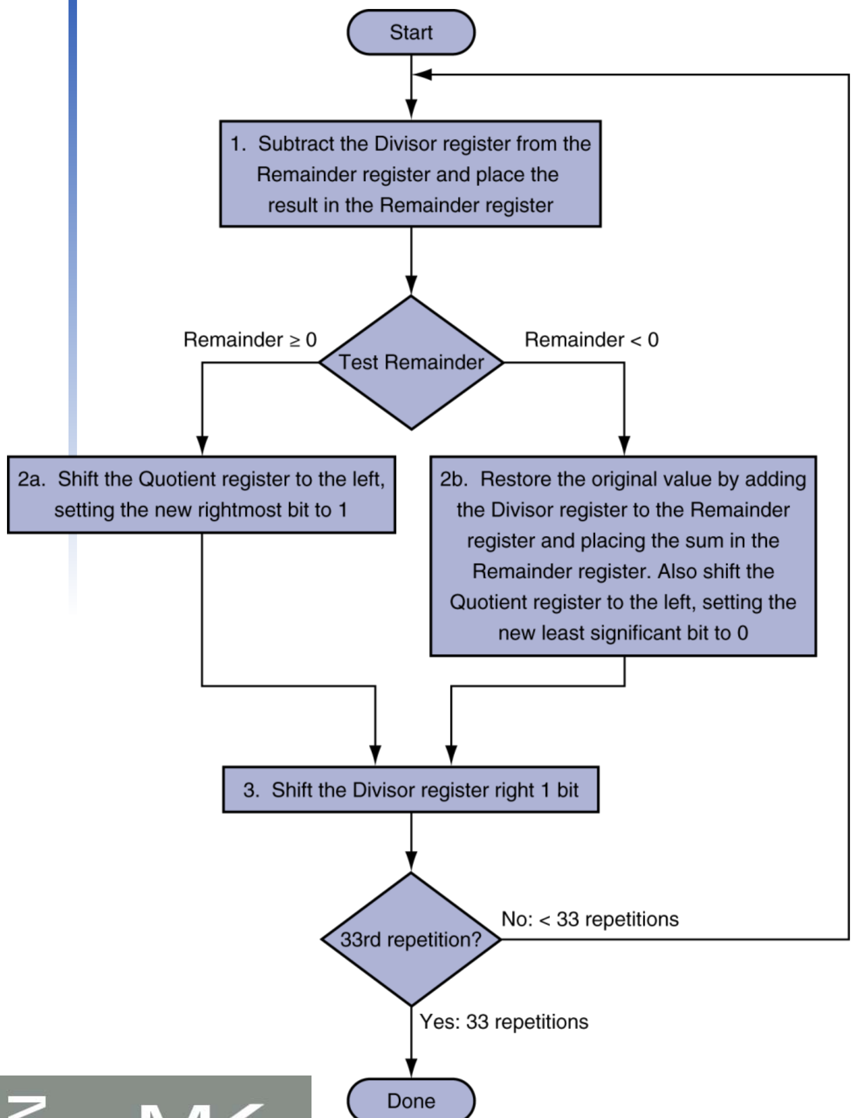
Stop when divisor ends with 1.

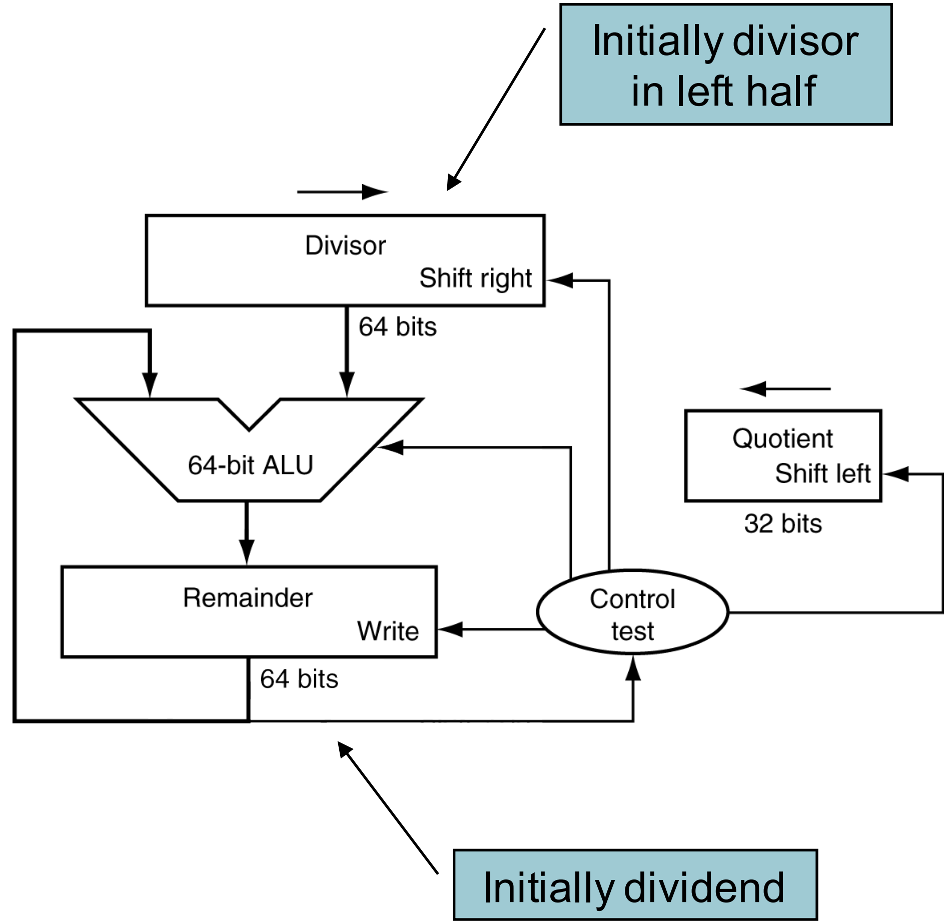
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required



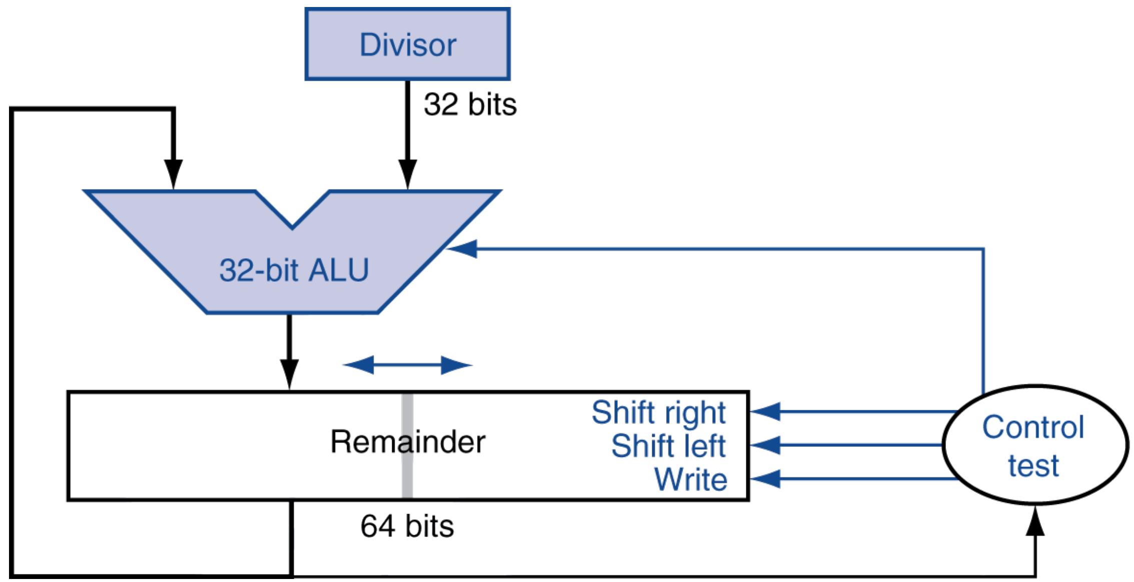
n-bit operands yield *n*-bit quotient and remainder

Division Hardware





Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt  /  divu rs, rt`

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -1.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

normalized

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading 1 bit
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

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
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
  - Double: approx 2^{-52}

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129-127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

HOME WORK

~~Floating-Point Example~~

1. Use the MIPS algorithm in this chapter to get the binary sum for these two decimal numbers, verify your result is correct. $5_{10}+9_{10} = ?_2$
2. Use the MIPS algorithm in this chapter to get the binary difference for these two decimal numbers, verify your result is correct. $5_{10}-9_{10} = ?_2$
3. Use the MIPS algorithm in this chapter to get the binary product for these two decimal numbers, verify your result is correct. $5_{10} \times 9_{10} = ?_2$
4. Use the MIPS algorithm in this chapter to get the binary quotient and binary remainder for these two decimal numbers, verify your result is correct. $9_{10}/2_{10} = ?_2$