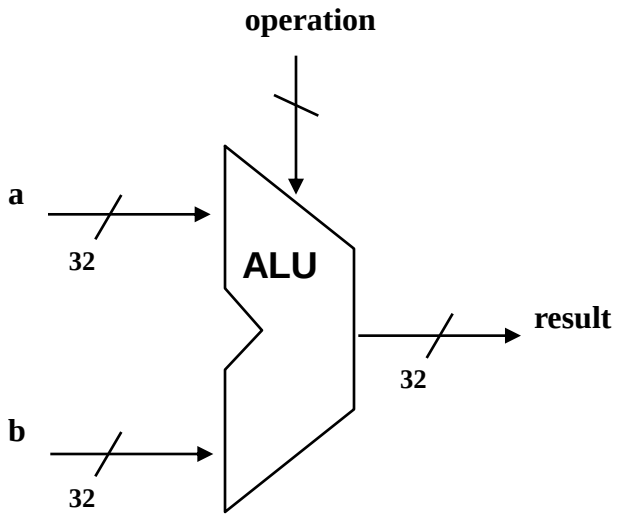
Chapter 3
Arithmetic for Computers

Arithmetic

- **Where we've been:**
  - 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 numbers (base 2)**
  - **0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...**
  - **decimal: $0...2^n-1$**
- **Of course it gets more complicated:**
  - **numbers are finite (overflow)**
  - **fractions and real numbers**
  - **negative numbers**
  - **e.g., no MIPS subi instruction; addi can add a negative number)**
- **How do we represent negative numbers?**
  - **i.e., which bit patterns will represent which numbers?**

Possible Representations

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Issues: balance, number of zeros, ease of operations

MIPS

- **32 bit signed numbers:**

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

Two's Complement Operations

- **Negating a two's complement number: invert all bits and add 1**
  - **remember: “negate” and “invert” are quite different!**
- **Converting n bit numbers into numbers with more than n bits:**
  - **MIPS 16 bit immediate gets converted to 32 bits for arithmetic**
  - **copy the most significant bit (the sign bit) into the other bits**

```
0010   -> 0000 0010
1010   -> 1111 1010
```

  - **"sign extension" (lbu vs. lb)**

2's complement

16-bit binary $0000\ 0000\ 0000\ 0010_{two}$

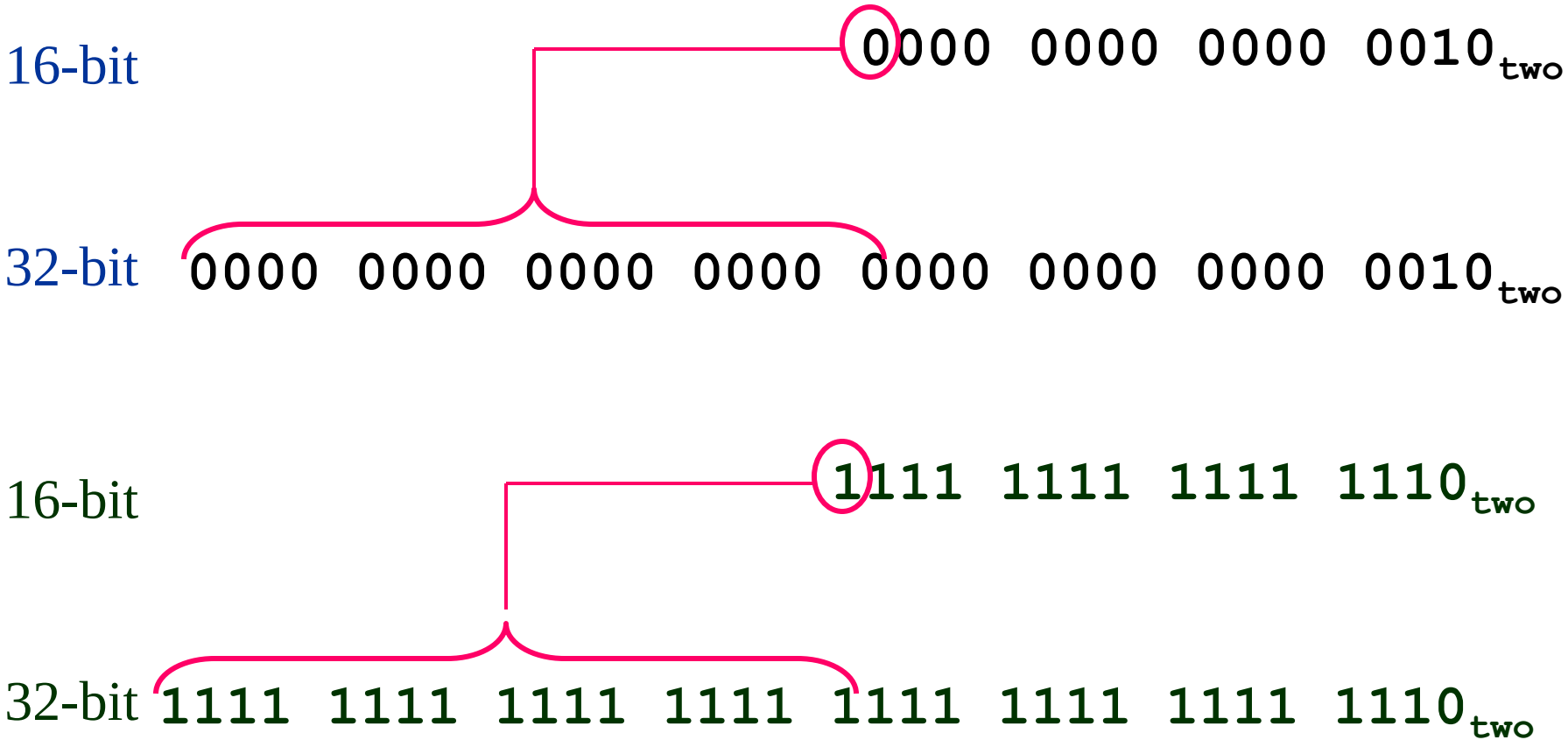
Negate number (invert & add 1)

$$\begin{array}{rr} & 1111\ 1111\ 1111\ 1101_{two} \\ + & \qquad\qquad\qquad\qquad\ 1_{two} \\ \hline & 1111\ 1111\ 1111\ 1110_{two} \end{array}$$

2’s complement

$$\begin{array}{r}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\
1_{two} \\
\hline
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}
\end{array}$$

Sign extension



Two's complement

Number	binary	2's complement
5	0000 0101	1111 1011
12	0000 1100	1111 0100
16	0001 0000	1111 0000
37	0010 0101	1101 1011

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

Addition & Subtraction

- **Just like in grade school (carry/borrow 1s)**

```
   0111          0111          0110
 + 0110        - 0110        - 0101
```

- **Two's complement operations easy**
  - subtraction using addition of negative numbers

```
   0111
 + 1010
```

- **Overflow (result too large for finite computer word):**
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
   0111
 + 0001
   1000
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive

Overflow

Operation	A	B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
  - — new MIPS instructions: `addu, addiu, subu`

  note: `addiu` *still sign-extends!*
  note: `sltu, sltiu` *for unsigned comparisons*

Review: Boolean Algebra & Gates

- **Problem: Consider a logic function with three inputs: A, B, and C.**

  Output D is true if at least one input is true
  Output E is true if exactly two inputs are true
  Output F is true only if all three inputs are true

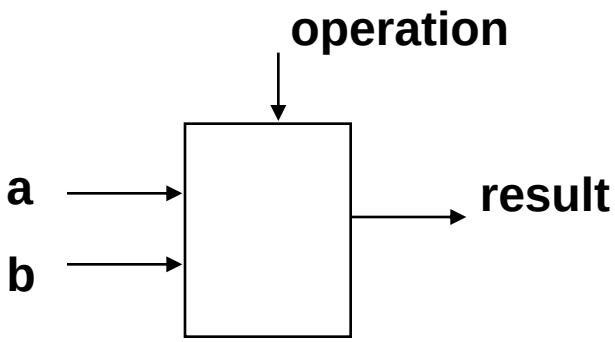
- **Show the truth table for these three functions.**

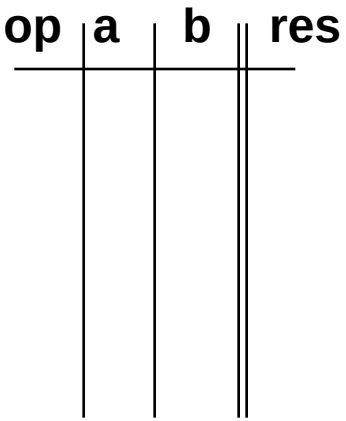
- **Show the Boolean equations for these three functions.**

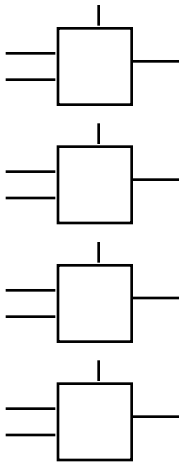
- **Show an implementation consisting of inverters, AND, and OR gates.**

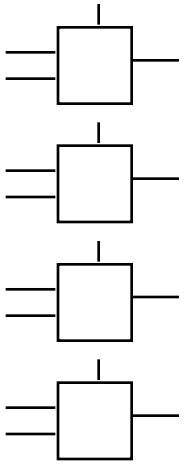
An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
  - we'll just build a 1 bit ALU, and use 32 of them





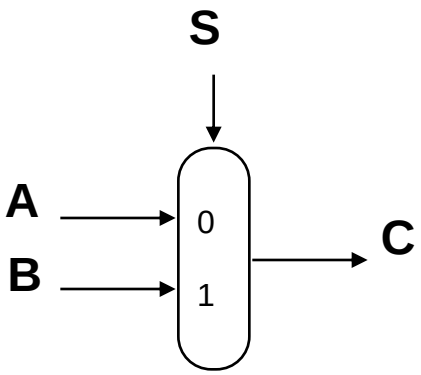




- Possible Implementation (sum-of-products):

Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

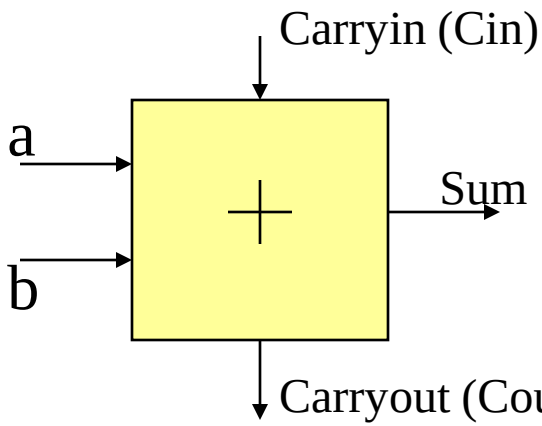


note: we call this a 2-input mux
even though it has 3 inputs!

- Lets build our ALU using a MUX:

Different Implementations

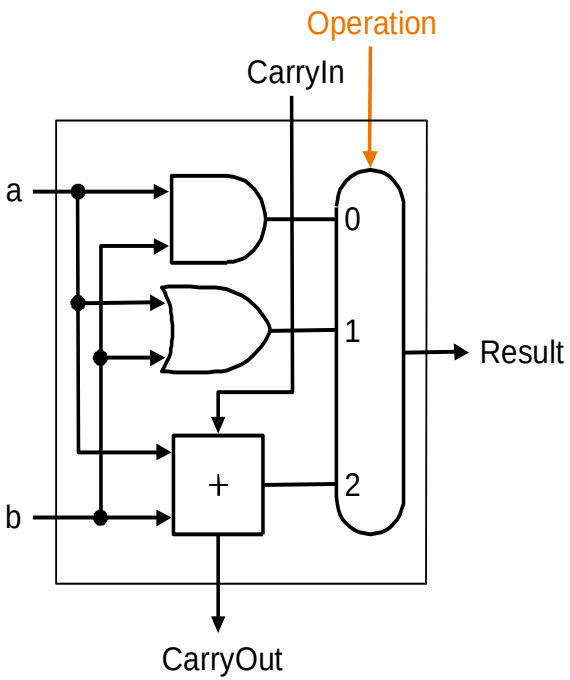
- **Not easy to decide the "best" way to build something**
  - Don't want too many inputs to a single gate
  - Dont want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- **Let's look at a 1-bit ALU for addition:**

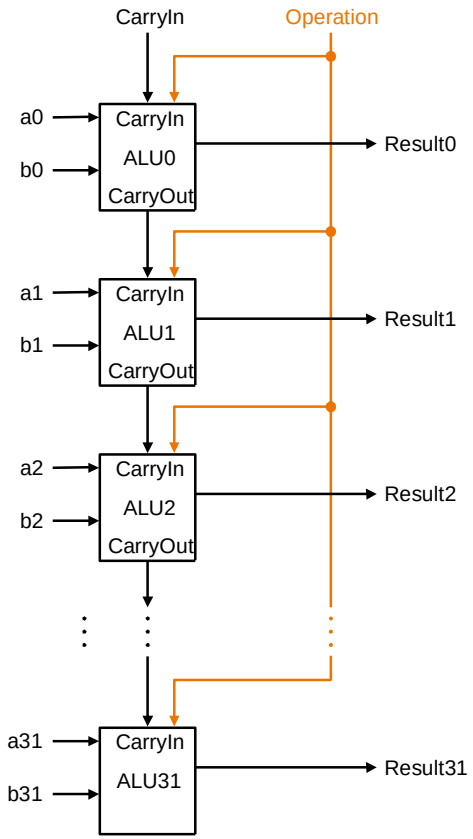


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- **How could we build a 1-bit ALU for add, and, and or?**
- **How could we build a 32-bit ALU?**

Building a 32 bit ALU

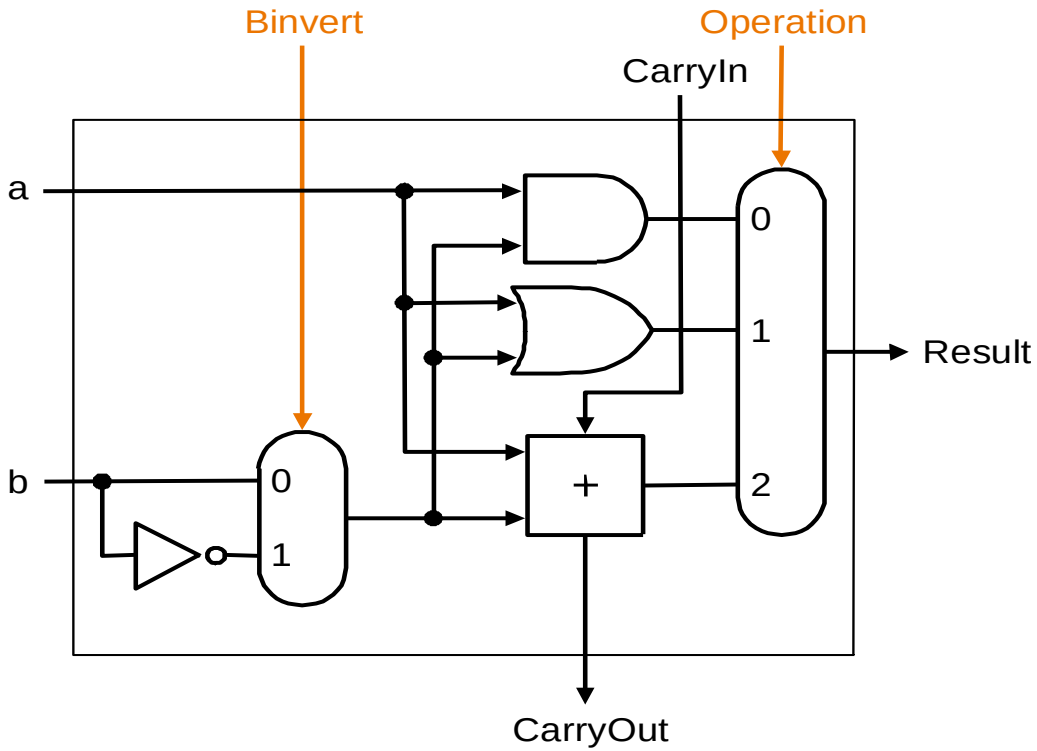




What about subtraction (a – b) ?

- Two's complement approach: just negate b and add.
- How do we negate?

- A very clever solution:

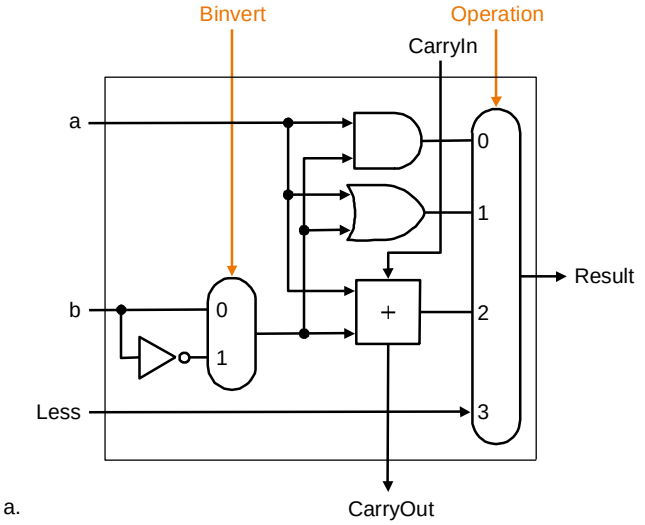


Tailoring the ALU to the MIPS

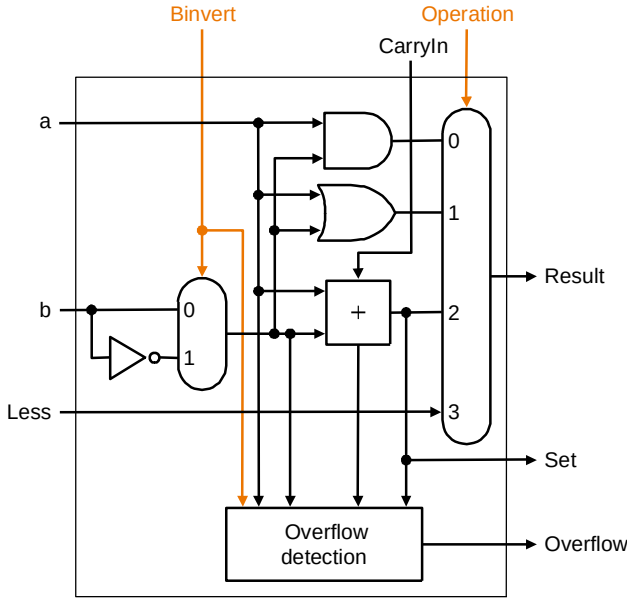
- **Need to support the set-on-less-than instruction (slt)**
  - **remember: slt is an arithmetic instruction**
  - **produces a 1 if rs < rt and 0 otherwise**
  - **use subtraction: (a-b) < 0 implies a < b**
- **Need to support test for equality (beq $t5, $t6, $t7)**
  - **use subtraction: (a-b) = 0 implies a = b**

Supporting slt

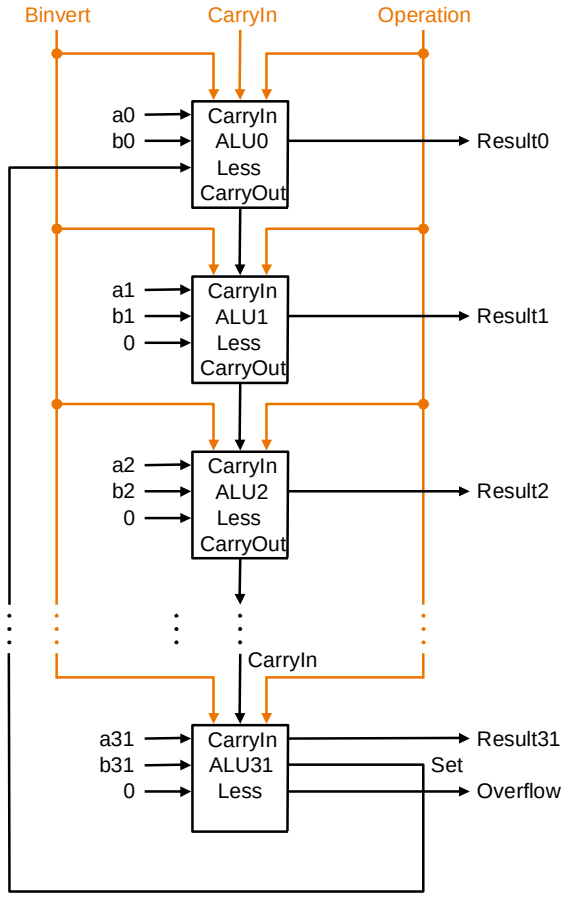
- **Can we figure out the idea?**



a.



b.

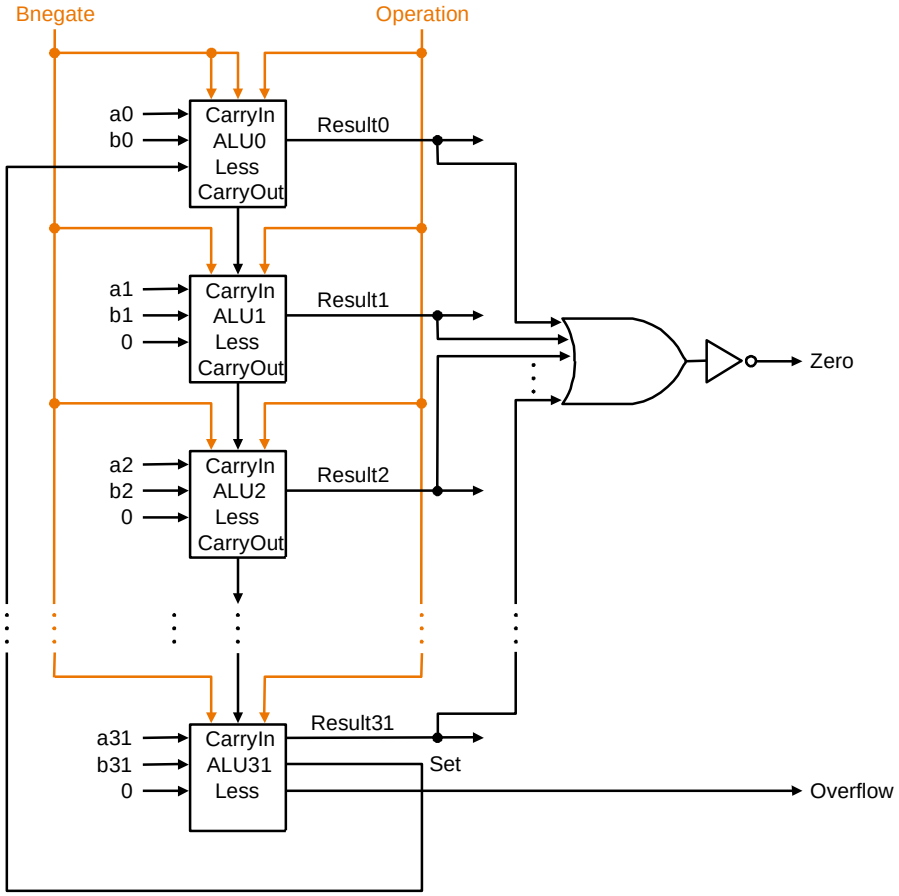

Binvert
CarryIn
Operation
a0
b0
CarryIn
ALU0
Less
CarryOut
Result0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Result1
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Result2
CarryIn
a31
b31
0
CarryIn
ALU31
Less
Result31
Set
Overflow

Test for equality

- **Notice control lines:**

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

- ***Note: zero is a 1 when the result is zero!***

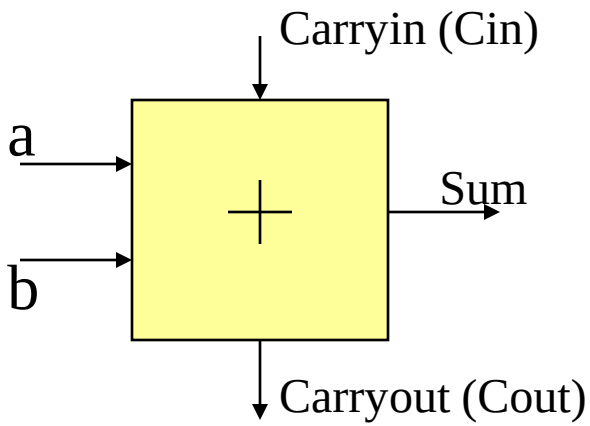


Conclusion

- **We can build an ALU to support the MIPS instruction set**
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- **Important points about hardware**
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "**critical path**" or the "deepest level of logic")
- **Our primary focus: comprehension, however,**
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at two examples for addition and multiplication

Basic adder

- **1-bit Adder**



$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$

$sum = a \text{ xor } b \text{ xor } c_{in}$

- **How could we build a 32-bit adder?**

- **Assume following delays:**
  NAND, NOR, INV = 1 Δ
- **XOR = 2 Δ (two input XOR)**

- **Implement ripple carry for 32-bit adder (what is the time?)**

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$
$$c_2 = b_1c_1 + a_1c_1 + a_1b_1$$
$$c_3 = b_2c_2 + a_2c_2 + a_2b_2$$
$$c_4 = b_3c_3 + a_3c_3 + a_3b_3$$

$c_2 =$

$c_3 =$

$c_4 =$

Not feasible! Why?

$$c_2 = b_1(b_0c_0+a_0c_0+a_0b_0) + a_1(b_0c_0+a_0c_0+a_0b_0) + a_1b_1$$

Carry-lookahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i b_i$
  - When would we propagate the carry? $p_i = a_i \oplus b_i$
- Did we get rid of the ripple?

$$c_1 = g_0 + p_0 c_0$$

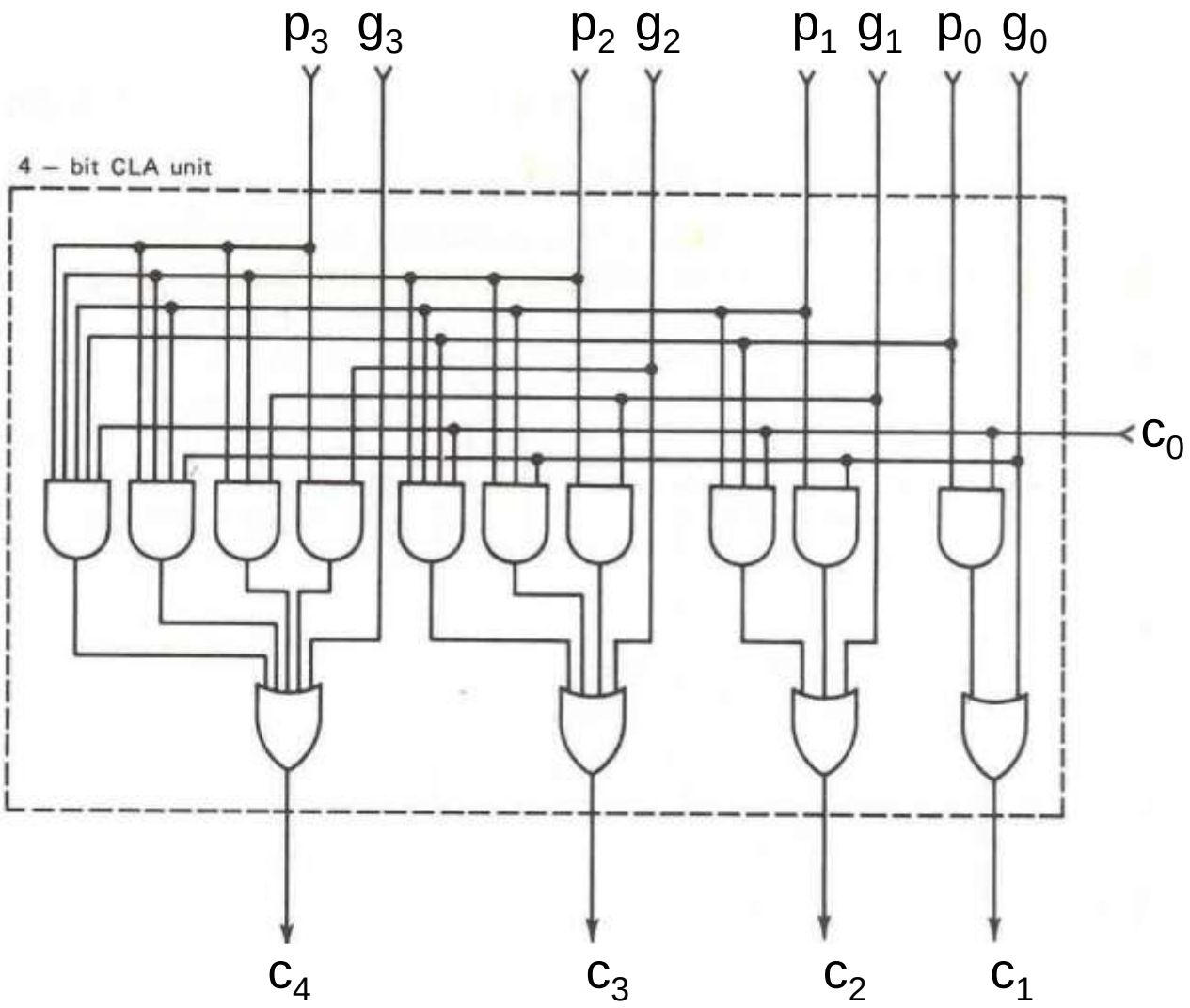
$$c_2 = g_1 + p_1 c_1 \qquad c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 c_2 \qquad c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

$$c_4 = g_3 + p_3 c_3 \qquad c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

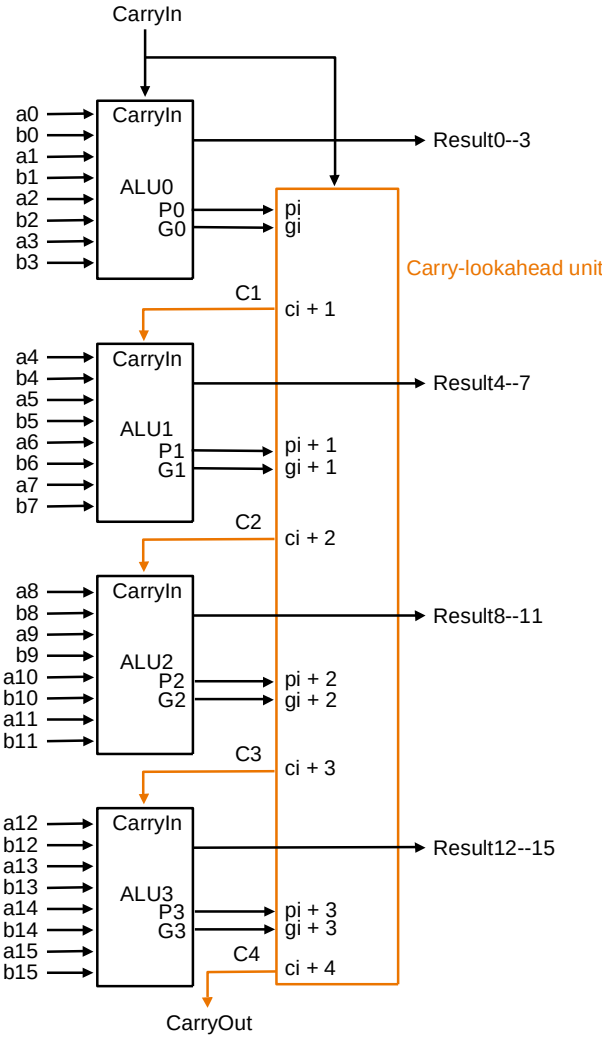
Feasible! Why?

4-bit CLA



- $P_0 = p_3 p_2 p_1 p_0$
- $G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$

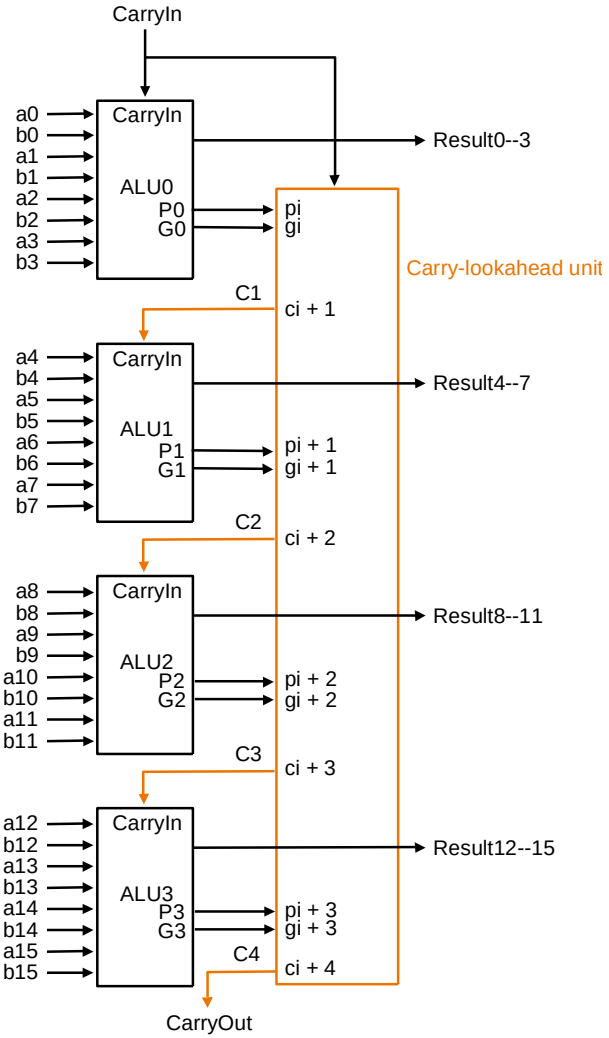
Use principle to build bigger adders



- **Can't build a 16 bit adder this way... (too big)**
- **Could use ripple carry of 4-bit CLA adders**
- **Better: use the CLA principle again!**

- $g_i = a_i \cdot b_i$
- $p_i = a_i \oplus b_i$
- $S_i = a_i \oplus b_i \oplus c_i$
- $S_i = p_i \oplus c_i$

Use principle to build bigger adders



- **Can't build a 16 bit adder this way... (too big)**
- **Could use ripple carry of 4-bit CLA adders**
- **Better: use the CLA principle again!**

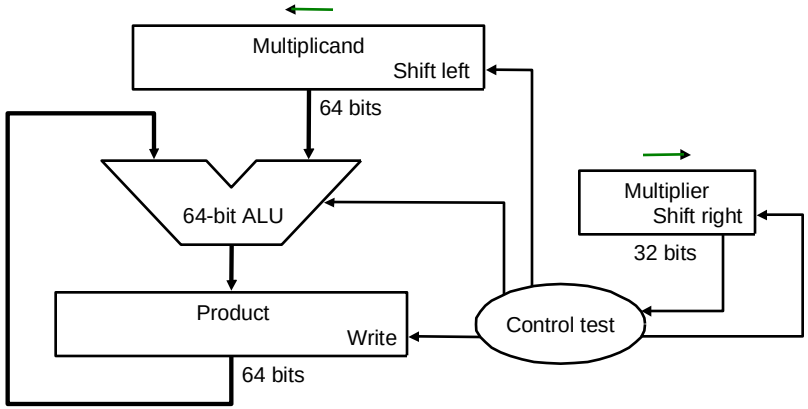
Multiplication

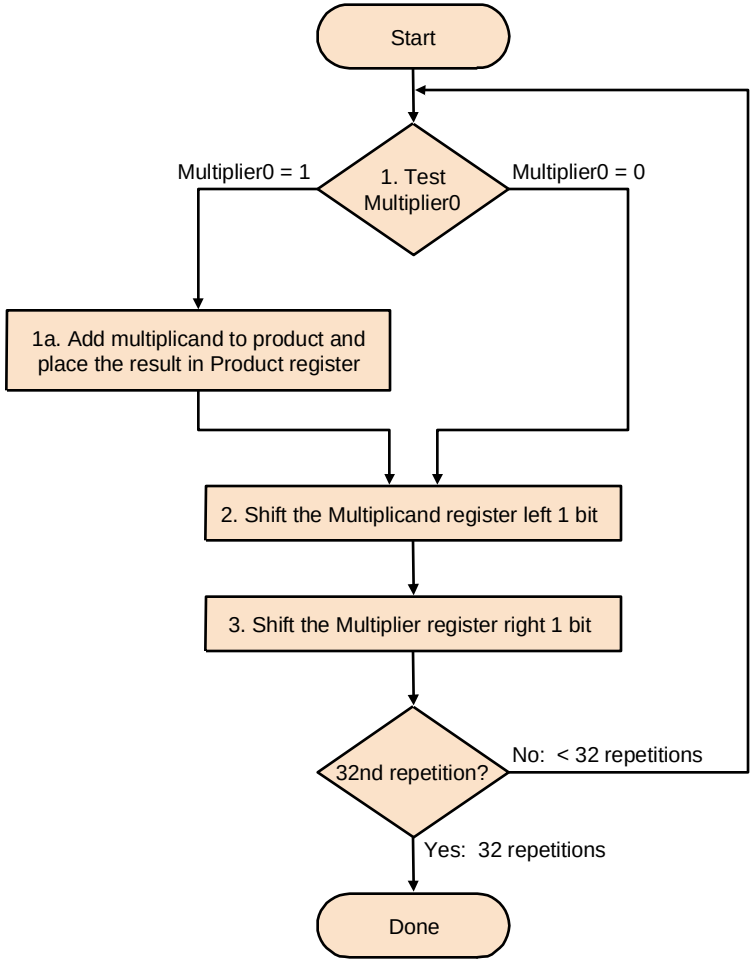
- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

```
    0010   (multiplicand)
  x 1011   (multiplier)
  ------
```

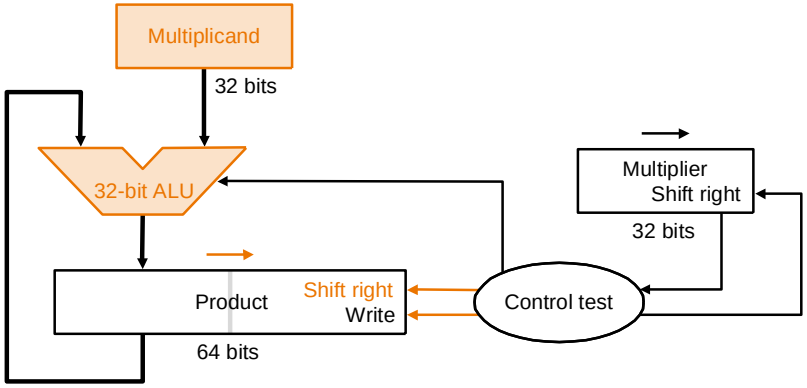
- Negative numbers: convert and multiply
  - there are better techniques, we won't look at them

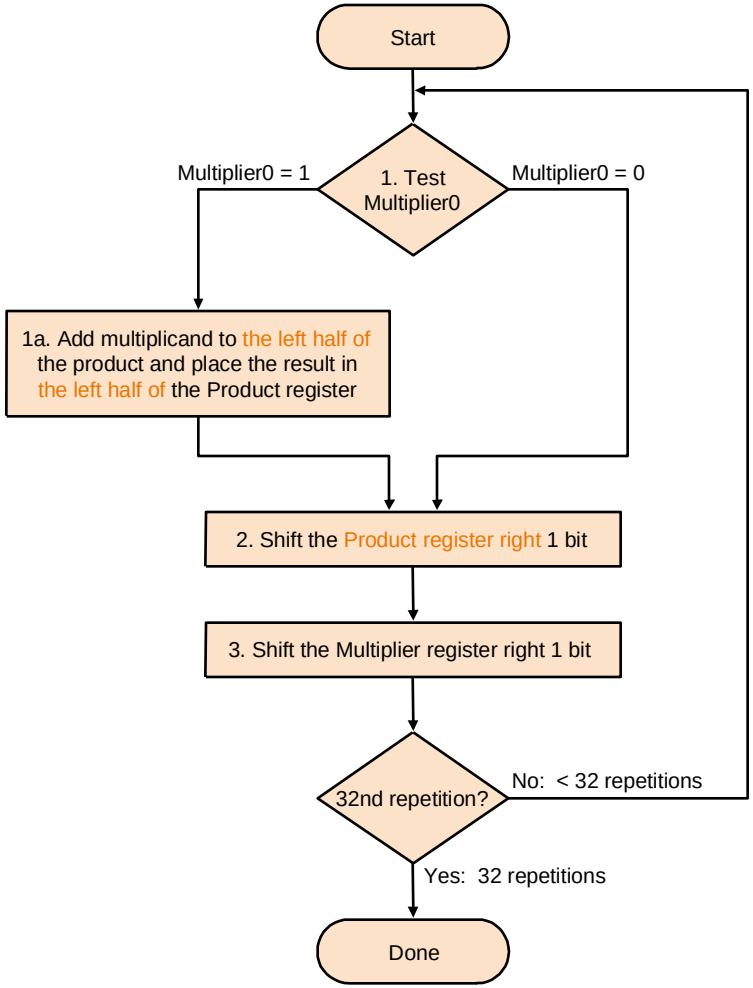
Multiplication: Implementation



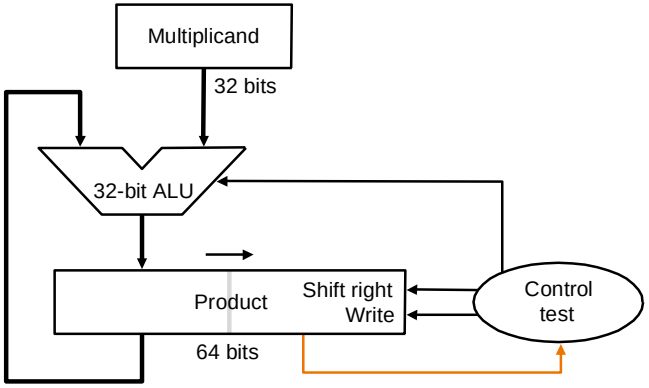


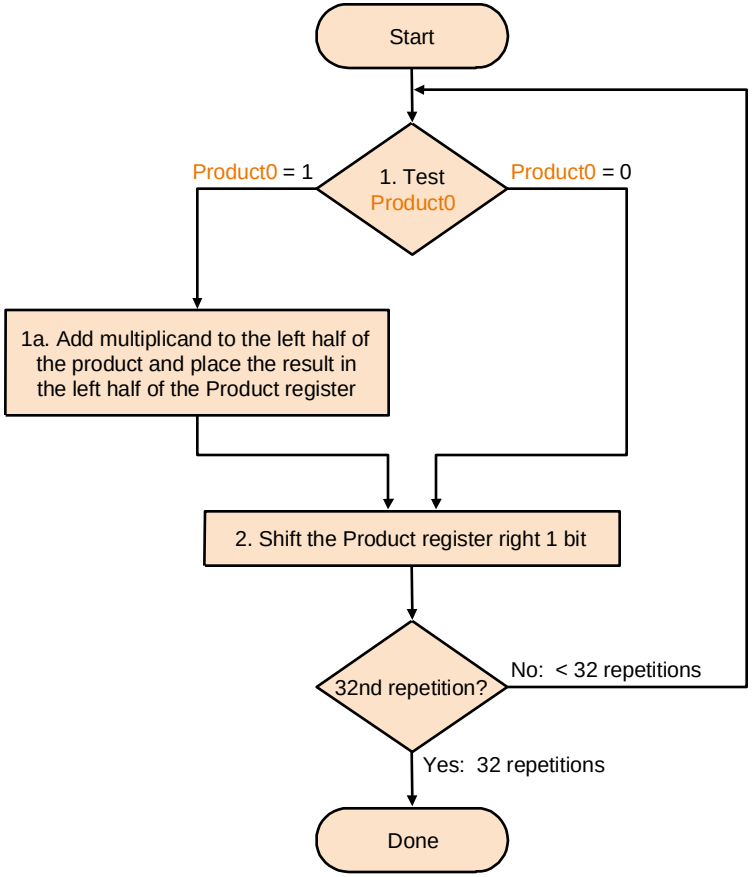
Second Version





Final Version





Floating Point Numbers

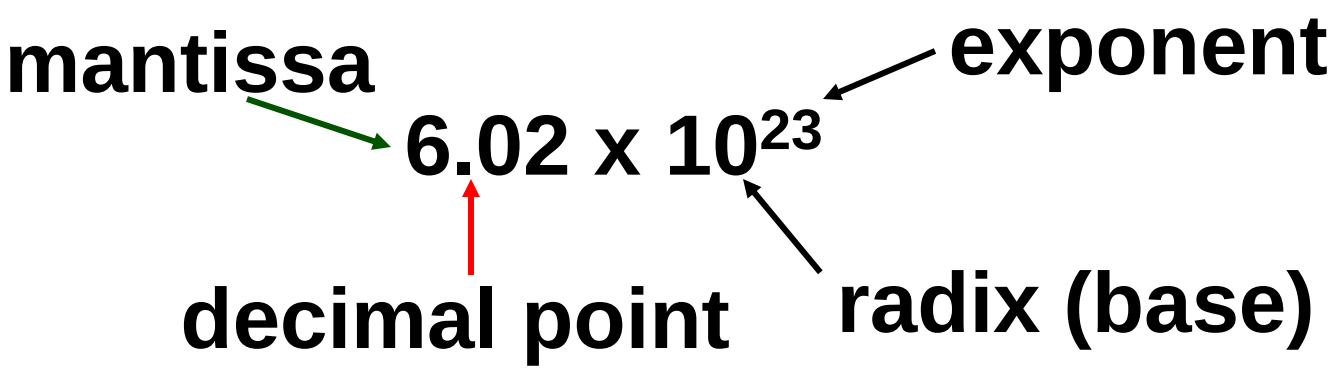
Review of Numbers

- **Computers are made to deal with numbers**
- **What can we represent in N bits?**
  - **Unsigned integers:**

    0 to $2^N - 1$
  - **Signed Integers (Two's Complement)**

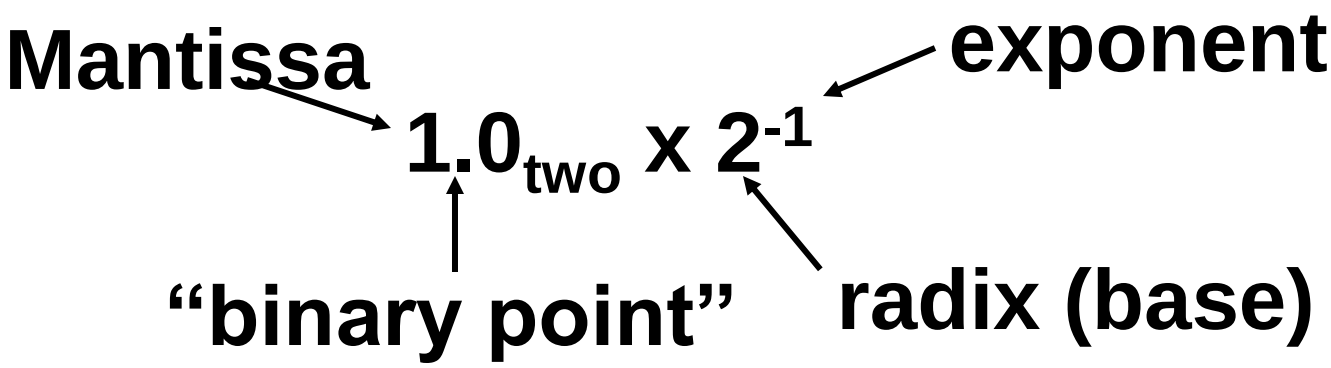
    $-2^{(N-1)}$ to $2^{(N-1)} - 1$
- **What about other numbers?**
  - **Very large numbers? (seconds/century)**
    $3{,}155{,}760{,}000_{10}$ ($3.15576_{10} \times 10^9$)
  - **Very small numbers? (atomic diameter)**
    0.00000001_{10} ($1.0_{10} \times 10^{-8}$)
  - **Rational (repeating pattern) 2/3 (0.666666666. . .)**
  - **Irrational $2^{1/2}$ (1.414213562373. . .)**
  - **Transcendental *e* (2.718...), π (3.141...)**
- **All represented in scientific notation**

Scientific Notation: Review



- Normalized form: no leadings 0s (exactly one digit to left of decimal point)
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

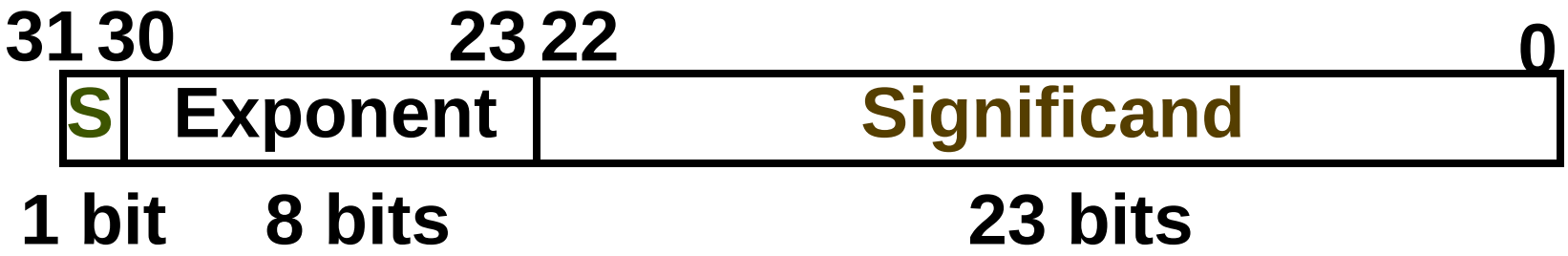
Scientific Notation: Binary Numbers



- Computer arithmetic that supports it called **floating point**, because it represents numbers where binary point is not fixed, as it is for integers
  - Declare such variable in C as `float`

Floating Point (FP) Representation (1/2)

- Normal format: +1.xxxxxxxxxx$_{two}$*2^{yyyy}two
- Multiple of Word Size (32 bits)



- S represents Sign

  Exponent represents y's

  Significand represents x's

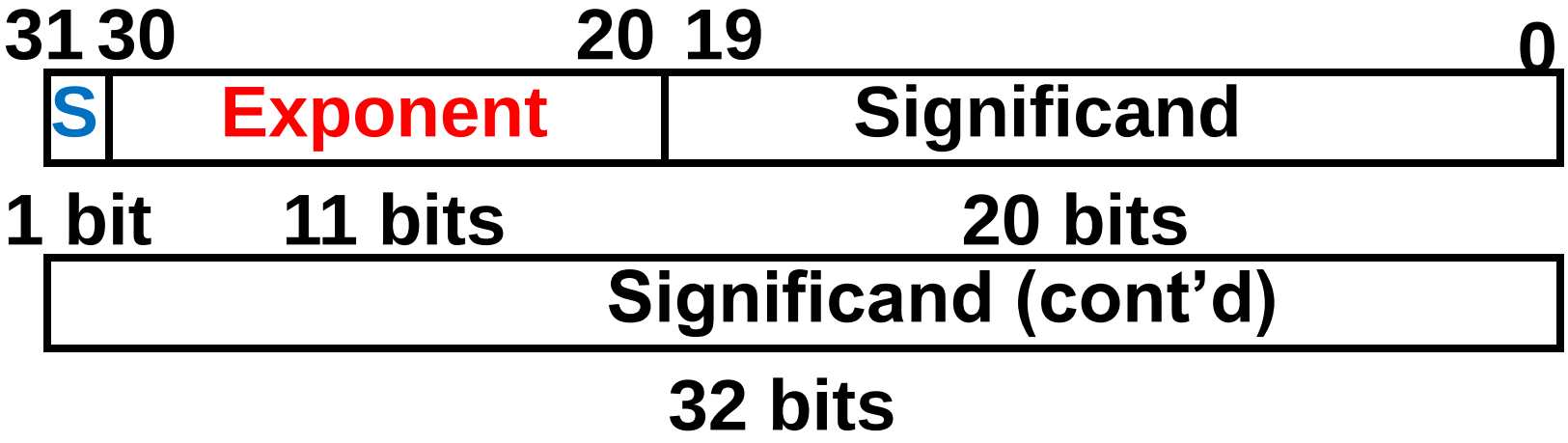
- Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}

FP Representation (2/2)

- What if result too large? (> 2.0×10^{38})
  - **Overflow!**
  - Overflow => Exponent larger than represented in 8-bit Exponent field
- What if result too small? (>0, < 2.0×10^{-38})
  - **Underflow!**
  - Underflow => Negative exponent larger than represented in 8-bit Exponent field
- How to reduce chances of overflow or underflow?

Double Precision FP Representation

- Next Multiple of Word Size (64 bits)



- **Double Precision (vs. Single Precision)**
  - C variable declared as `double`
  - Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}
  - But primary advantage is greater accuracy due to larger significand

IEEE 754 FP Standard (1/4)

- **Single Precision, DP similar**
- **Sign bit: 1 means negative**
  0 means positive
- **Significand:**
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true: Significand < 1 (for normalized numbers)
- **Note: 0 has no leading 1, so reserve exponent value 0 just for number 0**

IEEE 754 FP Standard (2/4)

- Kahan wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
- Could break FP number into 3 parts: compare signs, then compare exponents, then compare significands
- Wanted it to be faster, single compare if possible, especially if positive numbers
- Then want order:
  - Highest order bit is sign (negative < positive)
  - Exponent next, so big exponent => bigger #
  - Significand last: exponents same => bigger #

IEEE 754 FP Standard (3/4)

- Negative Exponent?
  - 2's comp? 1.0 x 2^{-1} v. 1.0 x2^{+1} (1/2 v. 2)

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

- This notation using integer compare of 1/2 v. 2 makes 1/2 > 2!

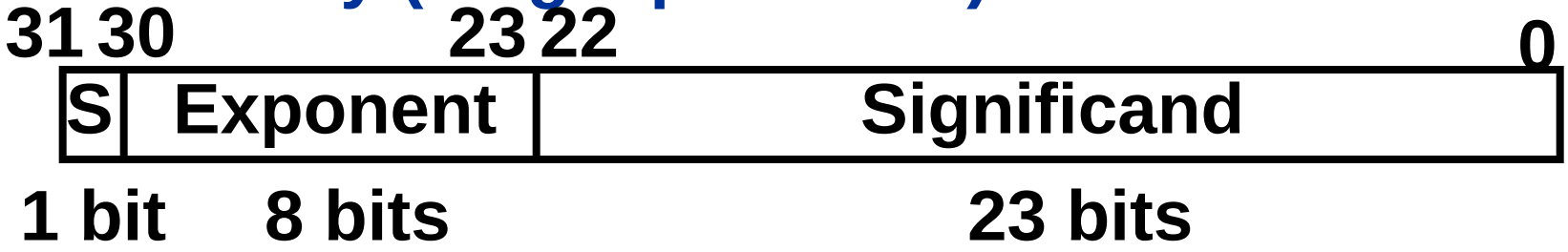
- Instead, pick notation 0000 0001 is most negative, and 1111 1111 is most positive
  - 1.0 x 2^{-1} v. 1.0 x2^{+1} (1/2 v. 2)

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

IEEE 754 FP Standard (4/4)

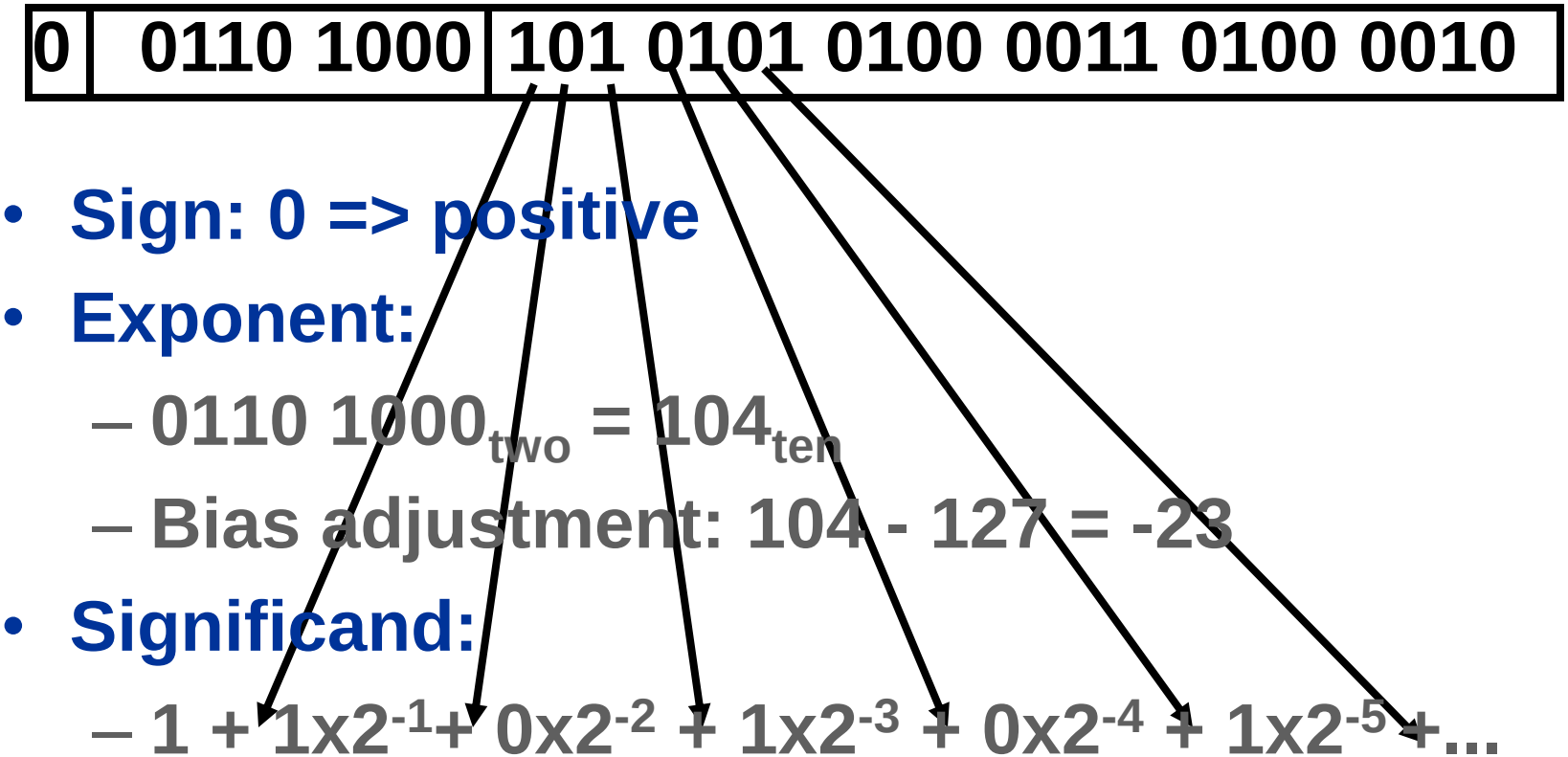
- Called Biased Notation, where bias is number subtract to get real number
  - IEEE 754 uses bias of 127 for single prec.
  - Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision

- Summary (single precision):



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - Double precision identical, except with exponent bias of 1023

Example: Converting FP to Decimal



- **Sign: 0 => positive**
- **Exponent:**
  - $0110\ 1000_{two} = 104_{ten}$
  - Bias adjustment: 104 - 127 = -23
- **Significand:**
  - $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$

    $= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$

    $= 1.0 + 0.666115$

- **Represents: $1.666115_{ten} * 2^{-23} \sim 1.986 * 10^{-7}$**

Converting Decimal to FP

- **Simple Case: If denominator is an exponent of 2 (2, 4, 8, 16, etc.), then it's easy.**
- **Show MIPS representation of -0.75**
  - **-0.75 = -3/4**
  - **$-11_{two}/100_{two} = -0.11_{two}$**
  - **Normalized to $-1.1_{two} \times 2^{-1}$**
  - **$(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$**
  - **$(-1)^{1} \times (1 + .100\ 0000\ \ldots\ 0000) \times 2^{(126-127)}$**

1	0111 1110	100 0000 0000 0000 0000 0000

Another Example

- **1/3**

  $= 0.33333\ldots_{10}$

  $= 0.25 + 0.0625 + 0.015625 + 0.00390625 + 0.0009765625 + \ldots$

  $= 1/4 + 1/16 + 1/64 + 1/256 + 1/1024 + \ldots$

  $= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-10} + \ldots$

  $= 0.0101010101\ldots_2 * 2^0$

  $= 1.0101010101\ldots_2 * 2^{-2}$

0	0111 1101	0101 0101 0101 0101 0101 010

Representation for +/- Infinity

- In FP, divide by zero should produce +/- infinity, not overflow.
- Why?
  - OK to do further computations with infinity e.g., X/0 > Y may be a valid comparison
  - Ask math majors
- IEEE 754 represents +/- infinity
  - Most positive exponent reserved for infinity
  - Significand all zeroes

Representation for 0

- **Represent 0?**
  - exponent all zeroes
  - significand all zeroes too
  - What about sign?
  - `+0: 0 00000000 00000000000000000000000`
  - `-0: 1 00000000 00000000000000000000000`
- **Why two zeroes?**
  - Helps in some limit comparisons

Special Numbers

- **What have we defined so far? (Single Precision)**

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	???

Representation for Not a Number

- **What do I get if I calculate**

  `sqrt(-4.0)` or `0/0`?

  - **If infinity is not an error, these shouldn't be either.**
  - **Called Not a Number (NaN)**
  - **Exponent = 255, Significand nonzero**

- **Why is this useful?**
  - **Hope NaNs help with debugging?**
  - **They contaminate: op(NaN,X) = NaN**

Special Numbers (cont'd)

- **What have we defined so far? (Single Precision)?**

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	NaN

Representation for Denorms (1/2)

- **Problem: There's a gap among representable FP numbers around 0**
  - **Smallest representable pos num:**

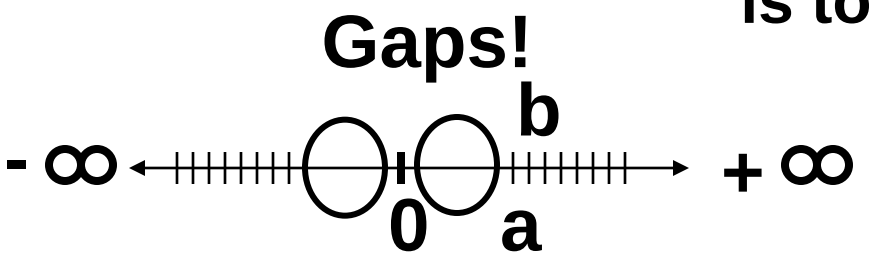
    $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - **Second smallest representable pos num:**

    $b = 1.000\ldots\ldots1_2 * 2^{-126} = 2^{-126} + 2^{-149}$

a - 0 = 2^{-126}

b - a = 2^{-149}

Normalization and implicit 1 is to blame!

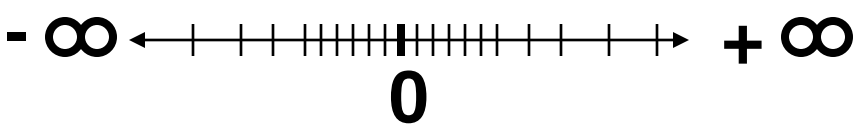


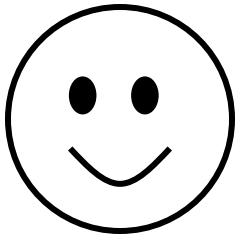
Representation for Denorms (2/2)

- **Solution:**
  - **We still haven't used Exponent = 0, Significand nonzero**
  - **Denormalized number: no leading 1, implicit exponent = -126.**
  - **Smallest representable pos num:**

    $a = 2^{-149}$
  - **Second smallest representable pos num:**

    $b = 2^{-148}$





Question

What is the decimal equivalent of:

1	1000 0001	111 0000 0000 0000 0000 0000

A: A: -3.5

B: B: -3.75

C: C: -7

D: D: -7.5

E: E: -15

F: F: -7 * 2^129

Answer

What is the decimal equivalent of:

1	1000 0001	111 0000 0000 0000 0000 0000
S	Exponent	Significand

$(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

$(-1)^{1} \times (1 + .111) \times 2^{(129-127)}$

$-1 \times (1.111) \times 2^{(2)} = -111.1 = -7.5$

A: A: -3.5

B: B: -3.75

C: C: -7

D: D: -7.5

E: E: -15

F: F: -7 * 2^129