CHAPTER 3
Implementing Arithmetic

Two Interpretations

$$\begin{tikzcd} 167_{10} & 10100111_2 \arrow[l, "\text{unsigned}"'] \arrow[r, "\text{signed}"] & -89_{10} \end{tikzcd}$$

- Signed vs. unsigned is a matter of interpretation; thus a single bit pattern can represent two different values.
- Allowing both interpretations is useful:

  Some data (e.g., count, age) can never be negative, and having a greater range is useful.

Which is Greater: 1001 or 0011?

Answer: It depends!

So how does the computer decide:

“if (x > y)..”   /* Is this true or false? */

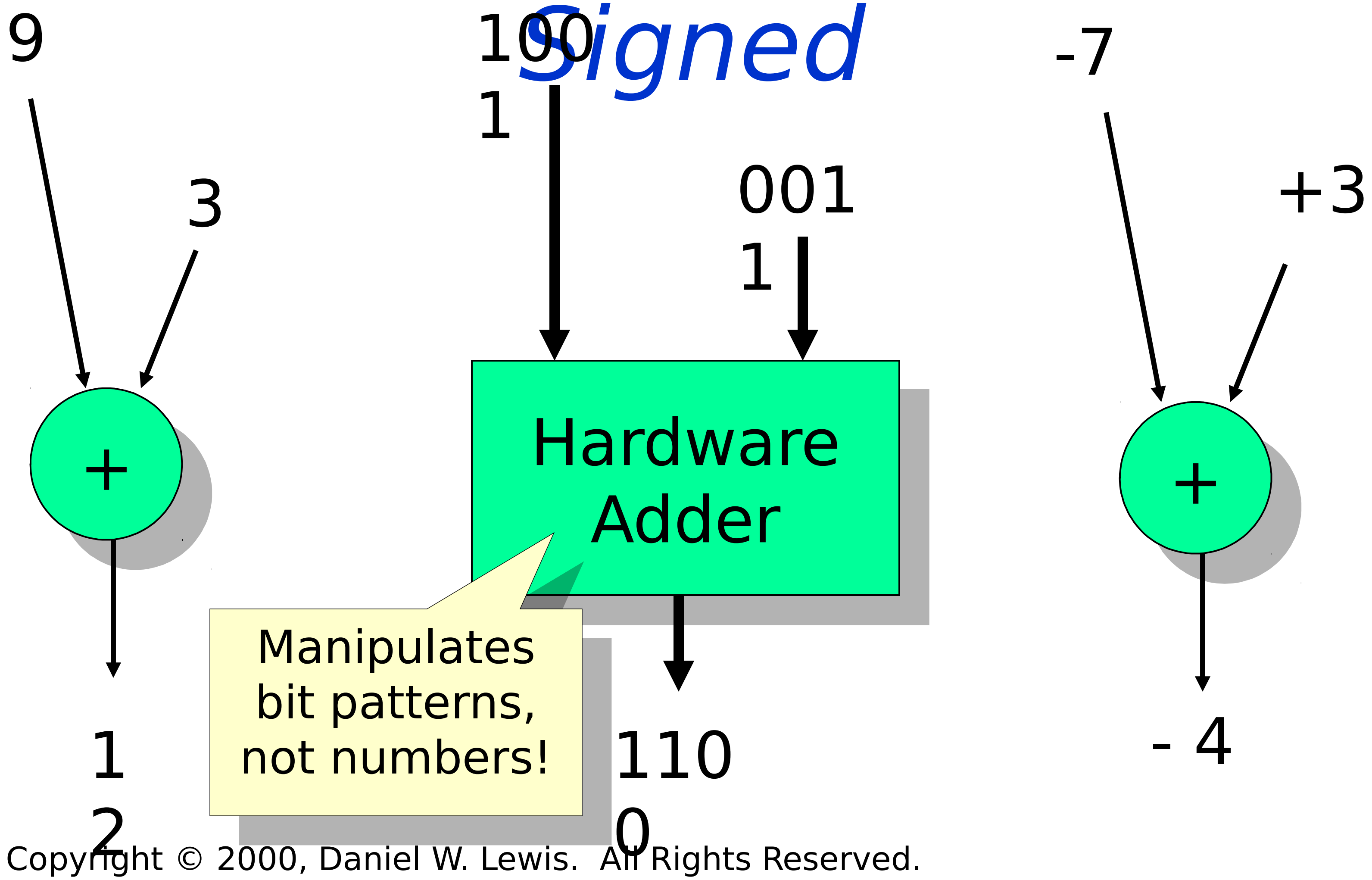
It’s a matter of interpretation, and depends on how x and y were declared: signed? Or unsigned?

Which is Greater: 1001 or 0011?

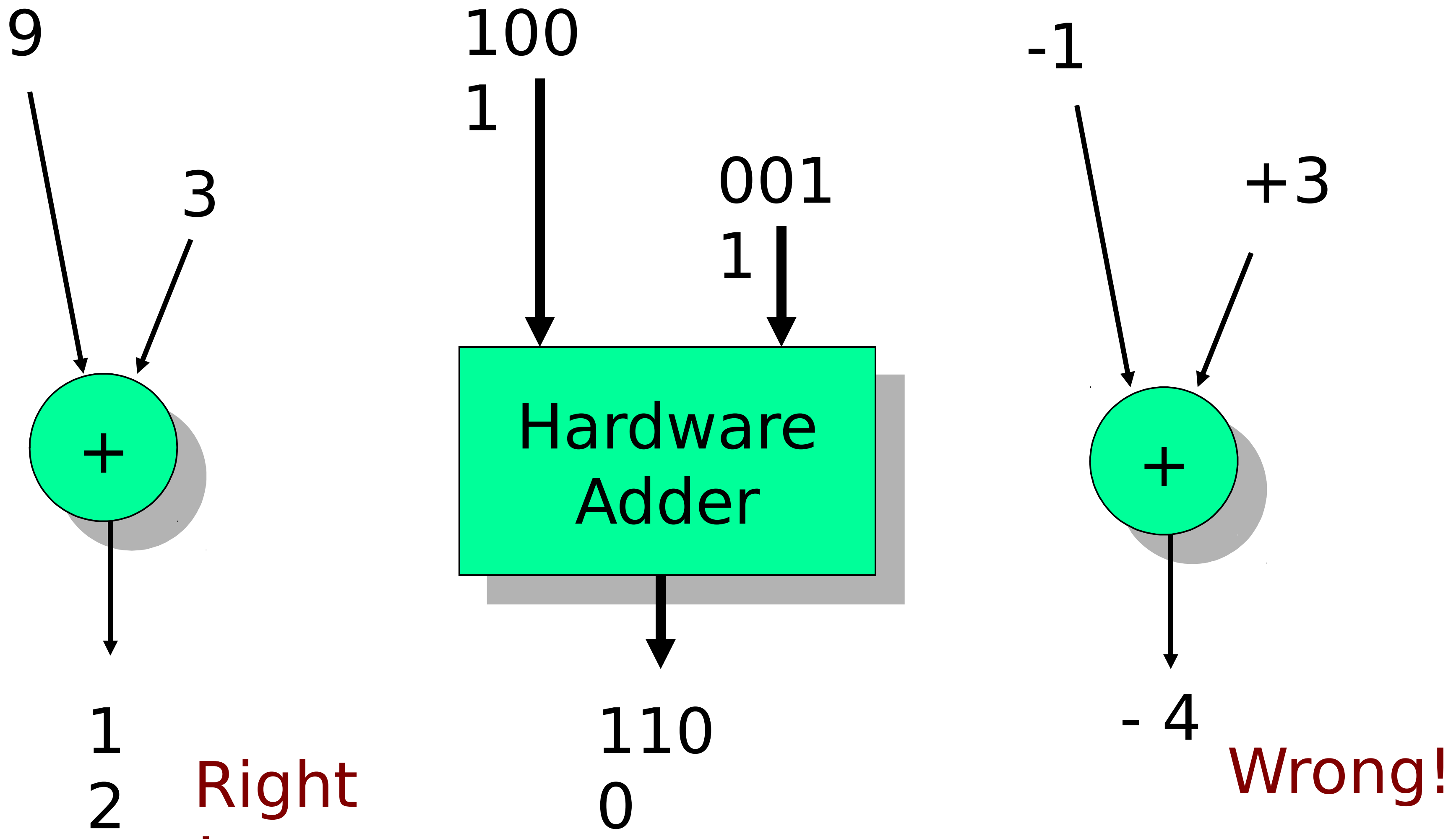
```
signed int x, y ;         MOV EAX,[x]
                    CMP  EAX,[y]
if (x > y) ...          □ JLE   Skip_Then_Clause
```

```
unsigned int x, y ;     MOV EAX,[x]
                    CMP  EAX,[y]
if (x > y) ...          □ JBE   Skip_Then_Clause
```

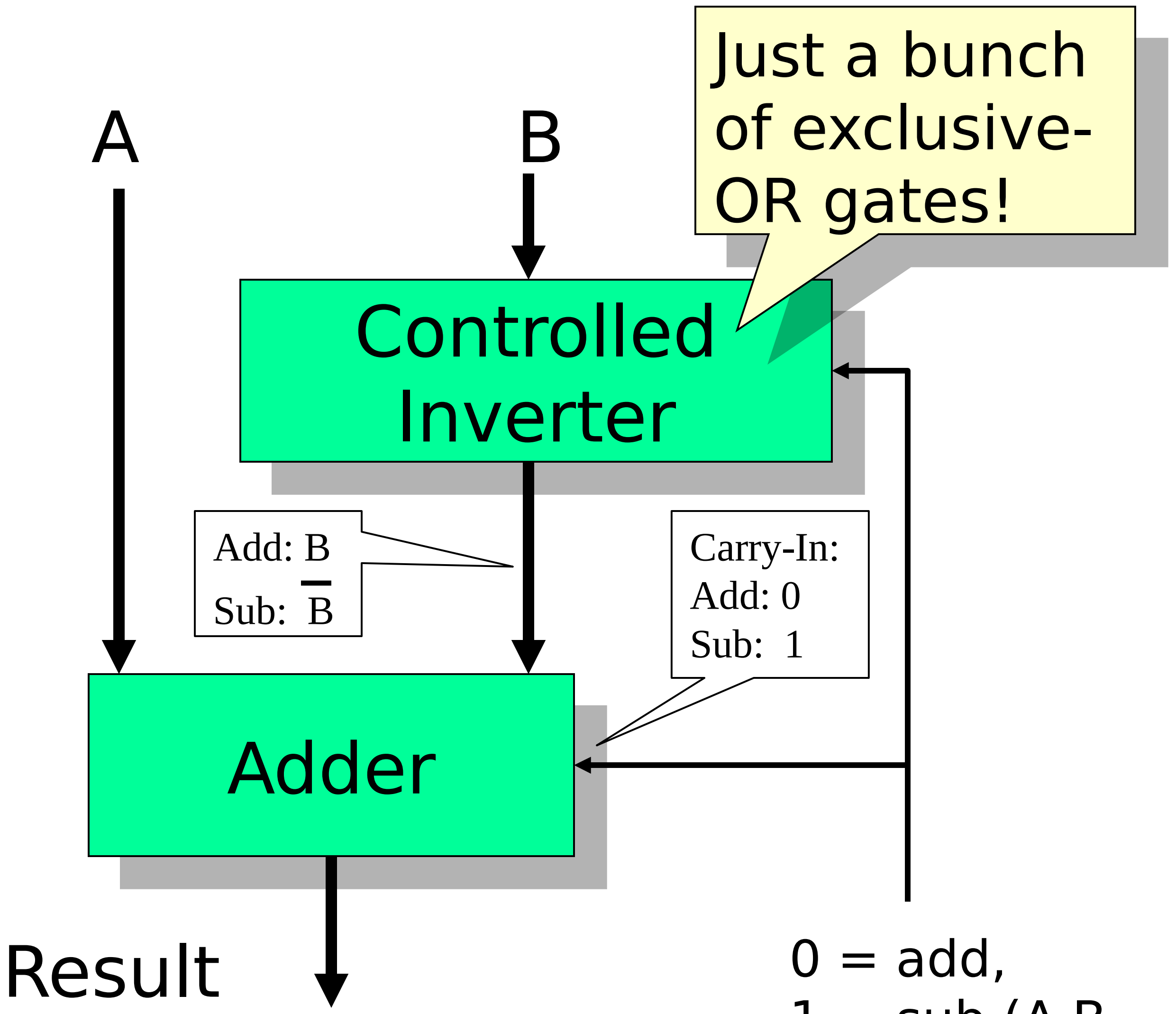
One Hardware Adder Handles Both: *Unsigned and 2's Complement Signed*



Why Not Sign+Magnitude?



Subtraction Is Easy!



0 = add,
1 = sub (A-B = $\overline{A}+B+1$)

Signed vs. Unsigned Multiplication

Unsigned

Decimal	Binary
12	1100
× 4	0100
48	**0011**0000

Signed (2’s complement)

Decimal	Binary
-4	1100
× +4	0100
-16	**1111**0000

Multiplying two n-bit numbers produces 2n bits of product. Least-significant halves of products are always identical, but most-significant halves will sometimes differ.

Arithmetic Shifting

Left Shift = Multiplying by a power of 2:

$13 \times 8 = 1101_2 \times 2^3 = 1101000_2$

“Arithmetic” Right Shift = Dividing by a power of 2?

$+13 \div 4 = 01101_2 \div 2^2 = 00011_2 = +3_{10}$ YES

$-13 \div 4 = 10011_2 \div 2^2 = 11100_2 = -4_{10}$ NO!

Multiplication by a Constant

$13_{10} \times N = 1101_2 \times N = 8N + 4N + 1N$

$= (N << 3) + (N << 2) + N$

2 shifts + 2 additions

On an old CPU, a multiply may take 100 times as long as an add or a shift, and the above will be 25 times faster!

Multiplication by a Constant

Consider $30_{10} \times N = 00011110_2 \times N$

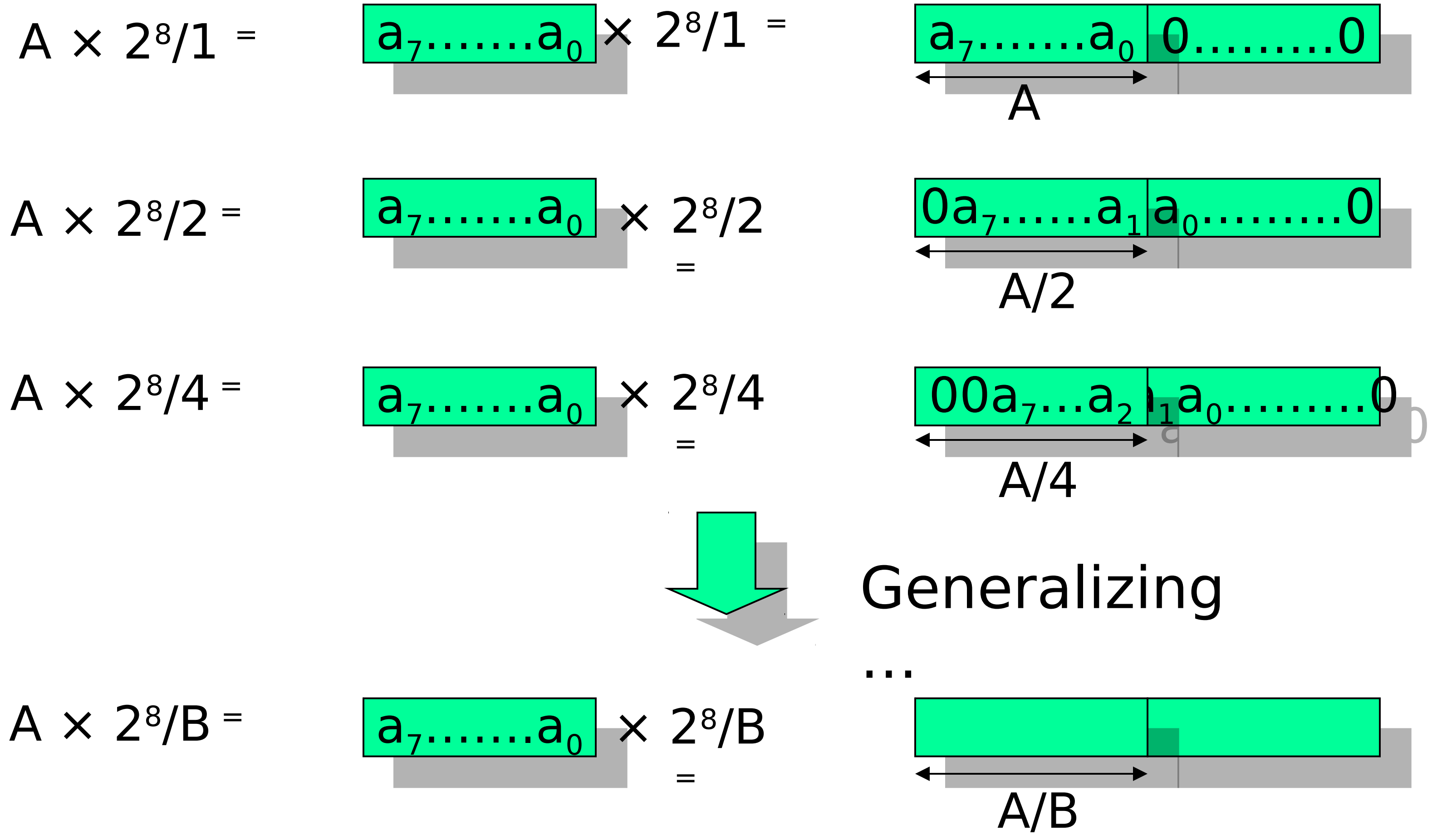
- This requires 4 shifts and 3 additions.

But $30_{10} = 32_{10} - 2_{10} = 2^5 - 2^1$

```
   00100000
-  00000010
   --------
   00011110
```

- Thus: $30_{10} \times N = (2^5 - 2^1) \times N = 2^5N - 2^1N$
- And requires only 2 shifts and 1 subtraction

Division by a constant = Multiplication by a constant!



$A \times 2^8/1 =$ $a_7.......a_0$ $\times\, 2^8/1 =$ $a_7.......a_0$ $0.........0$
A
$A \times 2^8/2 =$ $a_7.......a_0$ $\times\, 2^8/2 =$ $0a_7......a_1$ $a_0.........0$
A/2
$A \times 2^8/4 =$ $a_7.......a_0$ $\times\, 2^8/4 =$ $00a_7...a_2$ $a_1a_0.........0$
A/4
Generalizing …
$A \times 2^8/B =$ $a_7.......a_0$ $\times\, 2^8/B =$
A/B

Reciprocal Multiplication

$$A_{7..0} \div B_{7..0} = A \times (1/B)$$

$$= [\, A \times (2^8/B)\,]_{15..0} \div 2^8$$

$$= [A \times (2^8/B)]_{15..8}$$

Problems: Reciprocal Multiplication

8×8 Multiplication

16×16 Multiplication

A÷3 □□ A×?

A÷15 □□ A×?

A÷197 □□ A×?

A÷1000 □□ A×?

Multiplication & Division by $C=2^k$

Multiplication:

- Logical Left Shift by K bit positions
- Fills vacated bit positions on the right with 0’s

Division:

- Arithmetic Right Shift by K bit positions
- Fills vacated bit positions on the left with copy of sign bit
- Truncates towards negative infinity
- (integer division truncates towards zero)
- Anomaly when dividend is an odd negative number

Multiplication & Division by $C \neq 2^k$

Multiplication:

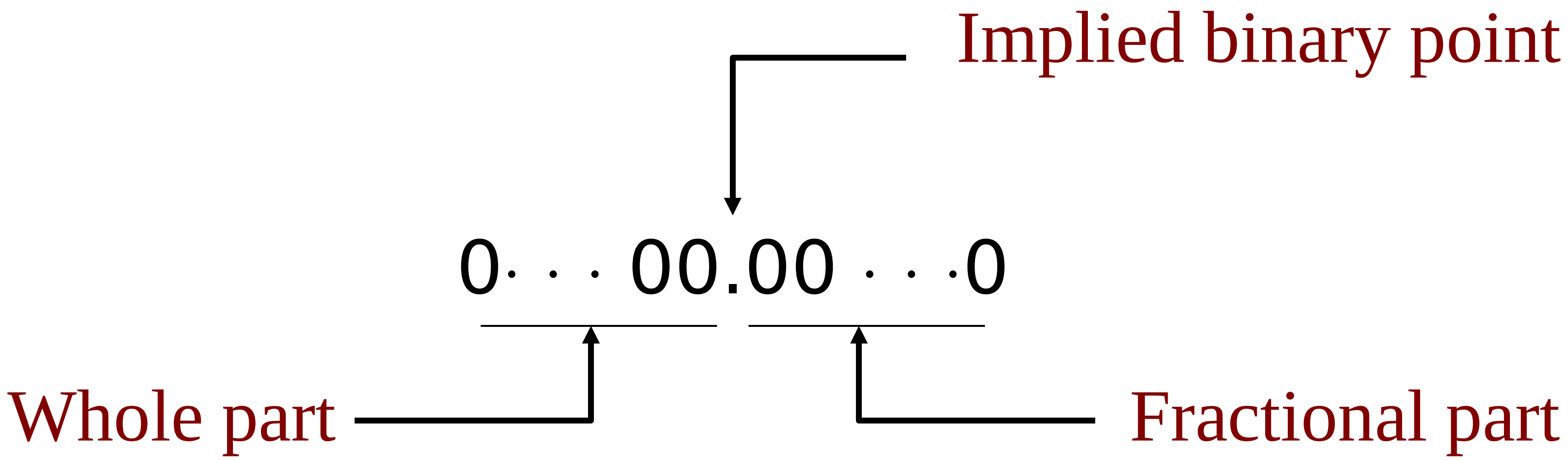
- Combination of left shifts, additions and subtractions
- Determined by binary pattern of constant C

Division:

- Use Reciprocal Multiplication (multiplier is $2^N/C$)
- Quotient left in most-significant half of double-length product
- Least-significant half contains the fractional bits

Fixed-Point Reals

Three components:



Fixed vs. Floating

- Floating-Point:
  - Pro: Large dynamic range determined by exponent; resolution determined by significand.
  - Con: Implementation of arithmetic in hardware is complex (slow).

- Fixed-Point:
  - Pro: Arithmetic is implemented using regular integer operations of processor (fast).
  - Con: Limited range and resolution.

Fixed-Point & Scale Factors

- The position of the binary point is determined by a *scale factor.*
- Different variables can have different scale factors.
- Determine scale factor by expected range and required resolution.
- Programmer must keep track of scale factors! (Tedious)

Fixed-Point Add/Subtract Using Operands w/Same Scale Factors

Operand/ Result	*Bit Pattern*	*Integer*	*×Scale Factor*	*=Value*
A	00011.110	+30	2^{-3} = 1/8	+3.750
B	00110.011	+51	2^{-3} = 1/8	+6.375
A + B	01010.001	+81	2^{-3} = 1/8	+10.125
A - B	11101.011	-21	2^{-3} = 1/8	-2.625

Fixed-Point Add/Subtract Using Operands w/Different Scale Factors

- Must align binary points before adding or subtracting; this makes scale factors the same.
- Two possibilities:
  - <u>If you shift the operand with fewer fractional bits left</u>, be careful that it doesn't cause an overflow.
  - <u>If you shift the operand with more fractional bits right</u>, be careful that it doesn't cause a loss of precision.
- Either approach may be used, but the scale factor of the resulting sum or difference

Fixed-Point Multiplication/Division

- No need to pre-align binary points!
- Number of fractional bits in result (determines the scale factor):
  - Multiplication: The number of fractional bits in the multiplicand plus the number in the multiplier.
  - Division: The number of fractional bits in the dividend less the number in the divisor.

Multiplying Fixed-Point Real Numbers.

Operand/ Result	*Bit Pattern*	*Int*	*×Scale Factor*	*=Value*
A	0000000000011.110	30	2^{-3} = 1/8	+3.7500
B	00000000001100.11	×51	2^{-2} = 1/4	+12.7500
A × B	0000010111.11010	=1530	2^{-3-2} = 1/32	+47.8125

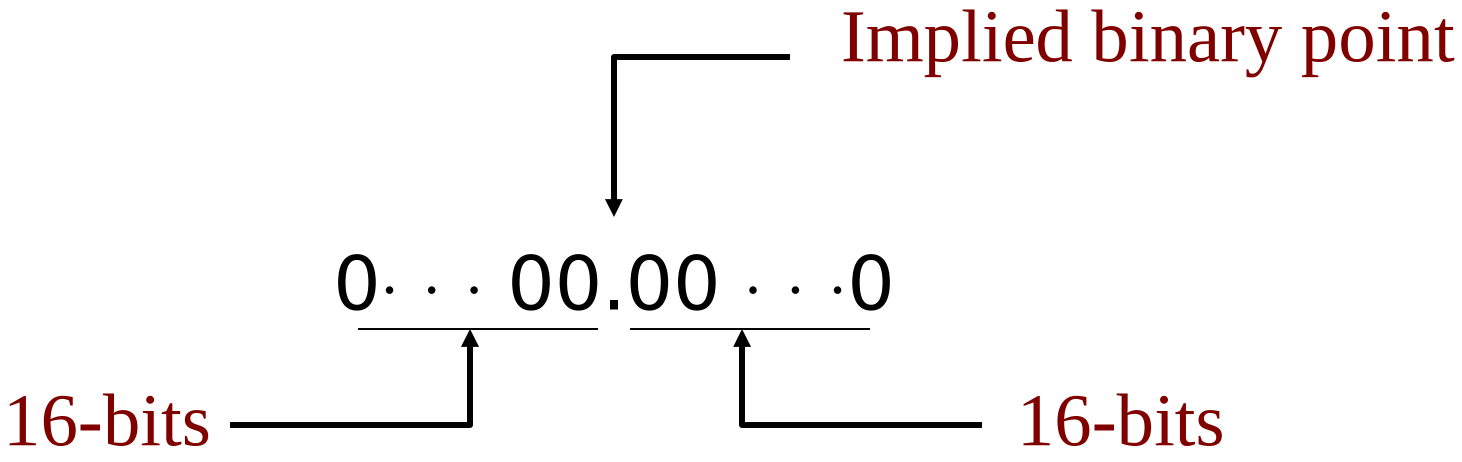
Dividing Fixed-Point Real Numbers.

Operand/ Result	*Bit Pattern*	*Int*	*×Scale Factor*	*=Value*
A	00000101111.11010	1530	2^{-5} = 1/32	+47.8125
B	0000000001110.011	÷115	2^{-3} = 1/8	+14.3750
A ÷ B	00000000000011.01	=13	2^{-5+3}= 1/4	+3.2500

Shifting Before Dividing Fixed-Point Real Numbers.

Operand/ Result	*Bit Pattern*	*Int*	*×Scale Factor*	*=Value*
$2^3 \times A$	00101111.11010000	12240	$2^{-8} = 1/256$	+47.8125
B	0000000001110.011	÷115	$2^{-3} = 1/8$	+14.3750
$2^3 \times A \div B$	00000000011.01010	=106	$2^{-8+3} = 1/32$	+3.3125

16.16 Fixed-Point Format



Problems: 8.8 Fixed-Point Representation

$-10.72_{10} \rightarrow ?_2 = -(10.72 \times 2^8)/2^8 = -2744/2^8$

$= -0000101010111000/2^8$

$= 11110101.01001000$

$45.37_{10} \rightarrow ?_2$

Q Number Format

Qm.n:

- m = # integer bits (not counting sign)
- n = # fractional bits
- m+n+1 = total number of bits

Qn:

- m = 0 is assumed (n + 1 = word size)
- 16 bits: Q15 = Q0.15 □ x**.**xxx xxxx xxxx xxxx

Q Number Format

Qm.n Range: -2^m to $+(2^m – 2^{-n})$

When n = 0, the number is an integer:

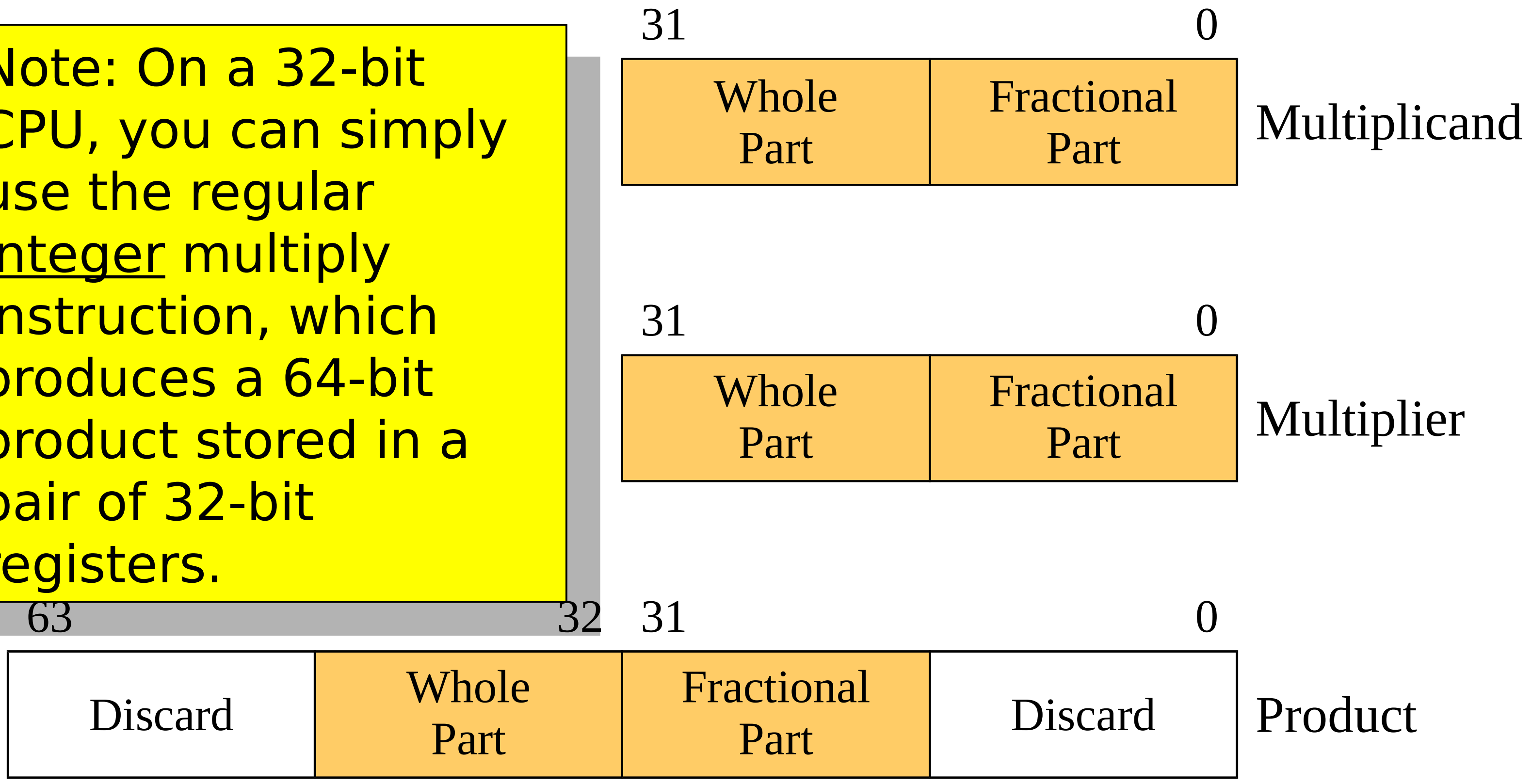
- -2^m to $+(2^m – 1)$
- If word size = 8 bits: 10000000**.** to 01111111**.**

When m = 0, the number is a fraction:

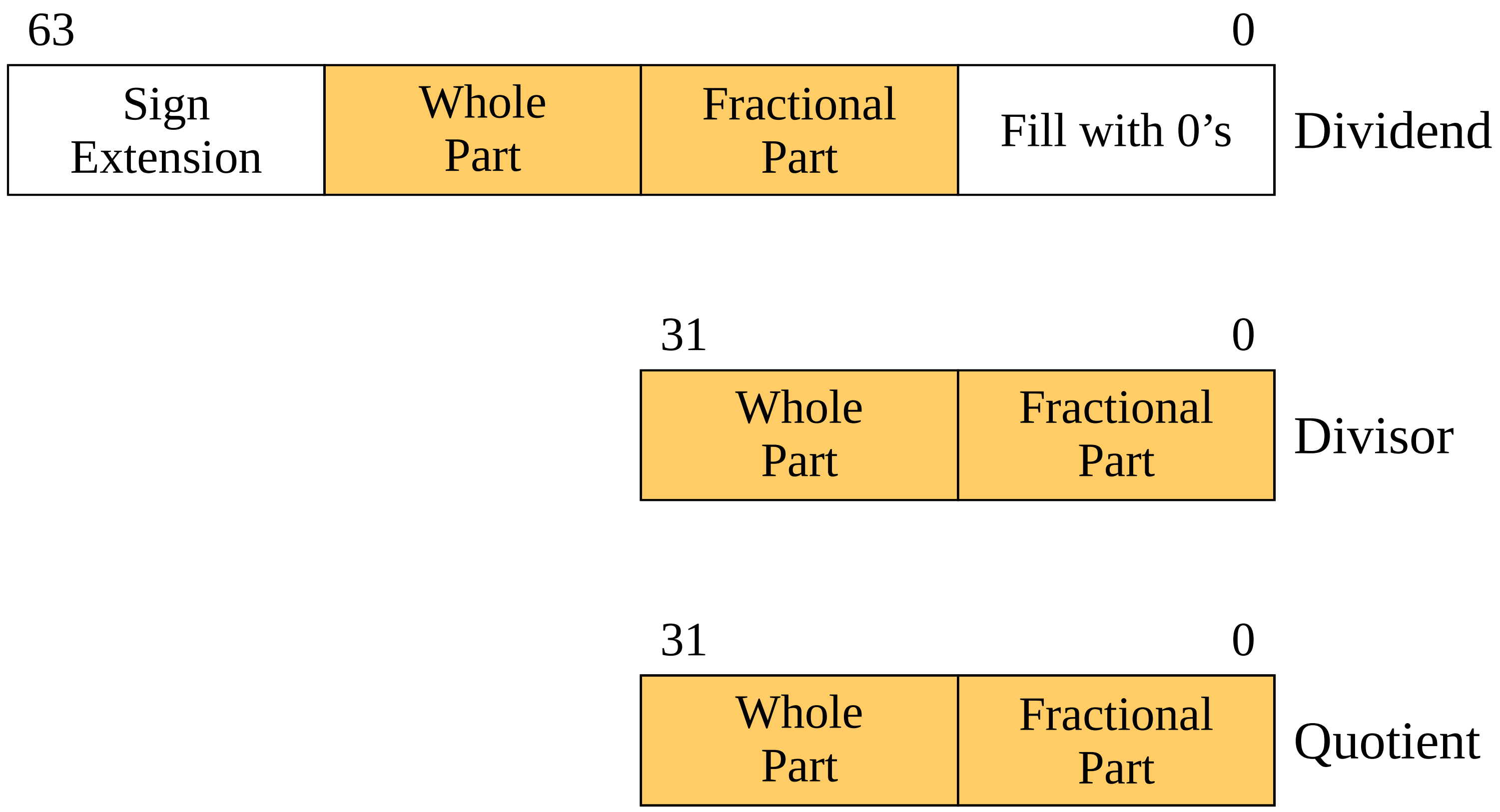
- -1 to $+(1 – 2^{-n})$
- If word size = 8 bits: 1**.**0000000 to

16.16 Fixed-Point Multiplication

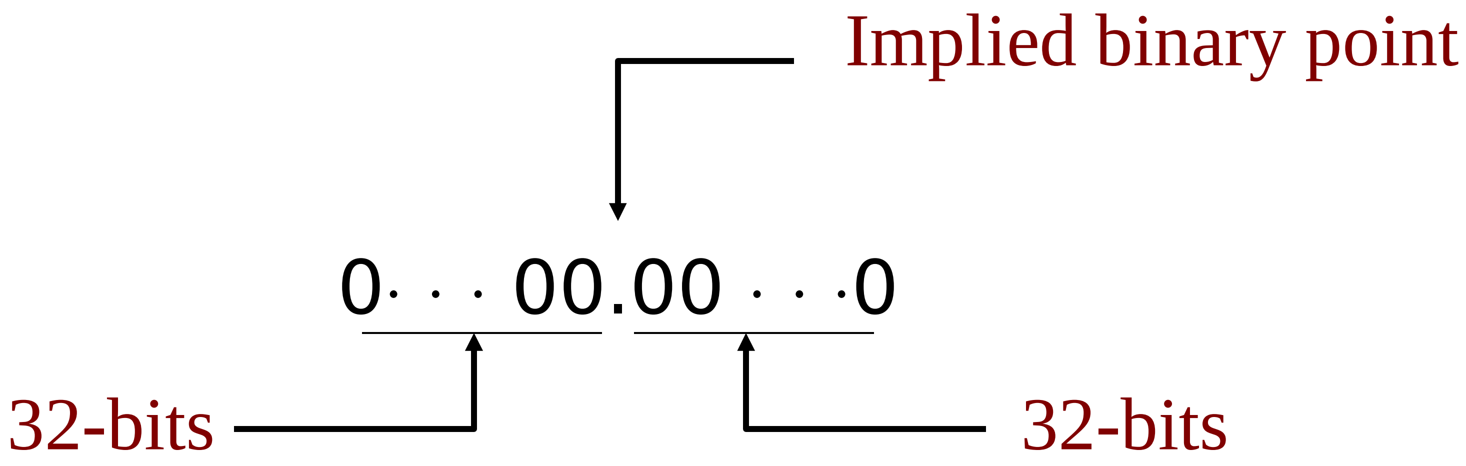
Note: On a 32-bit CPU, you can simply use the regular integer multiply instruction, which produces a 64-bit product stored in a pair of 32-bit registers.



16.16 Fixed-Point Division



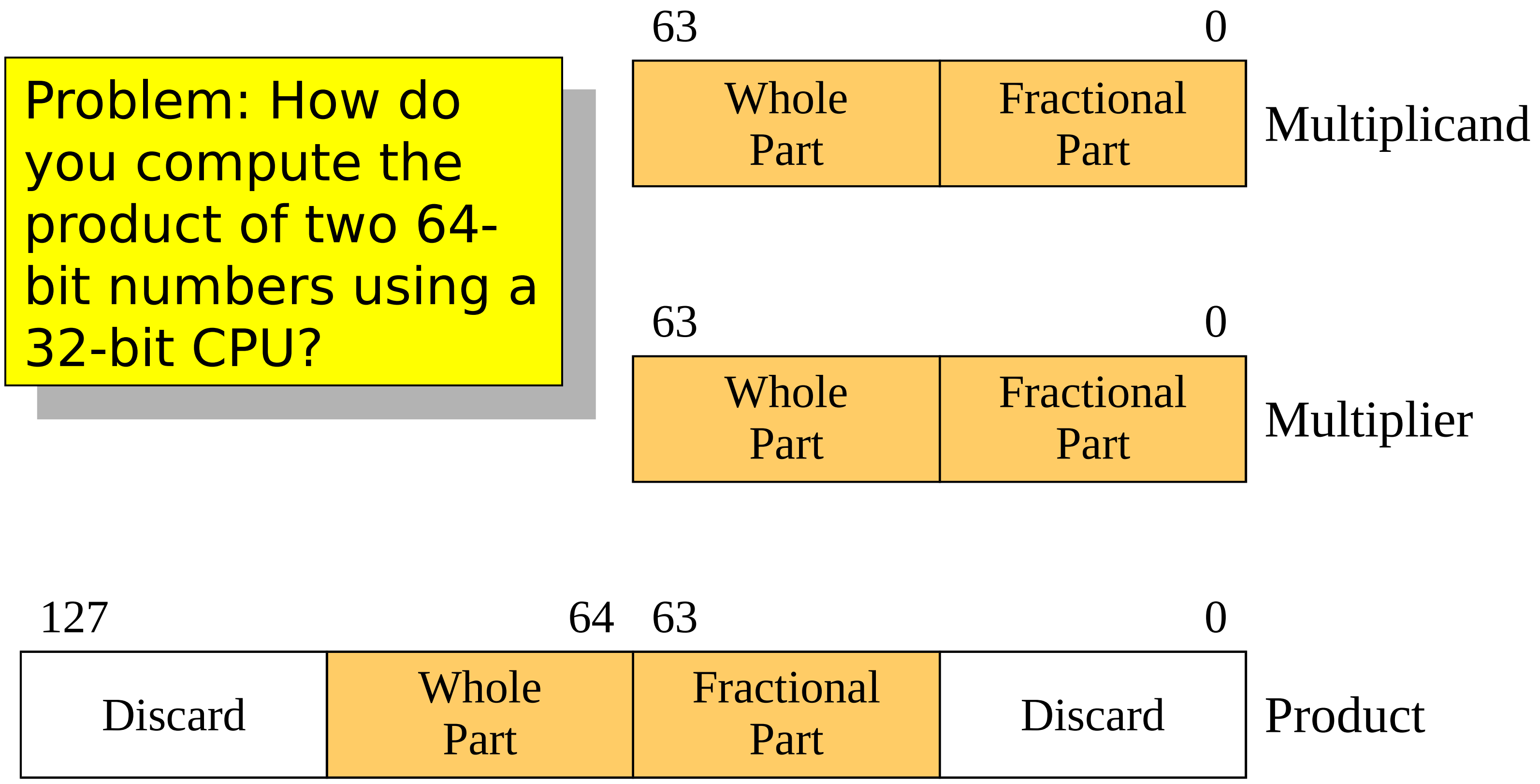
“Brute-Force” 32.32 Format



This format uses lots of bits, but memory is relatively cheap and it supports both very large and very small numbers. If all variables use this same format (i.e., a common scale factor), programming is simplified. This is the strategy used in the Sony PlayStation.

32.32 Fixed-Point Multiplication

Problem: How do you compute the product of two 64-bit numbers using a 32-bit CPU?



32.32 Fixed-Point Multiplication

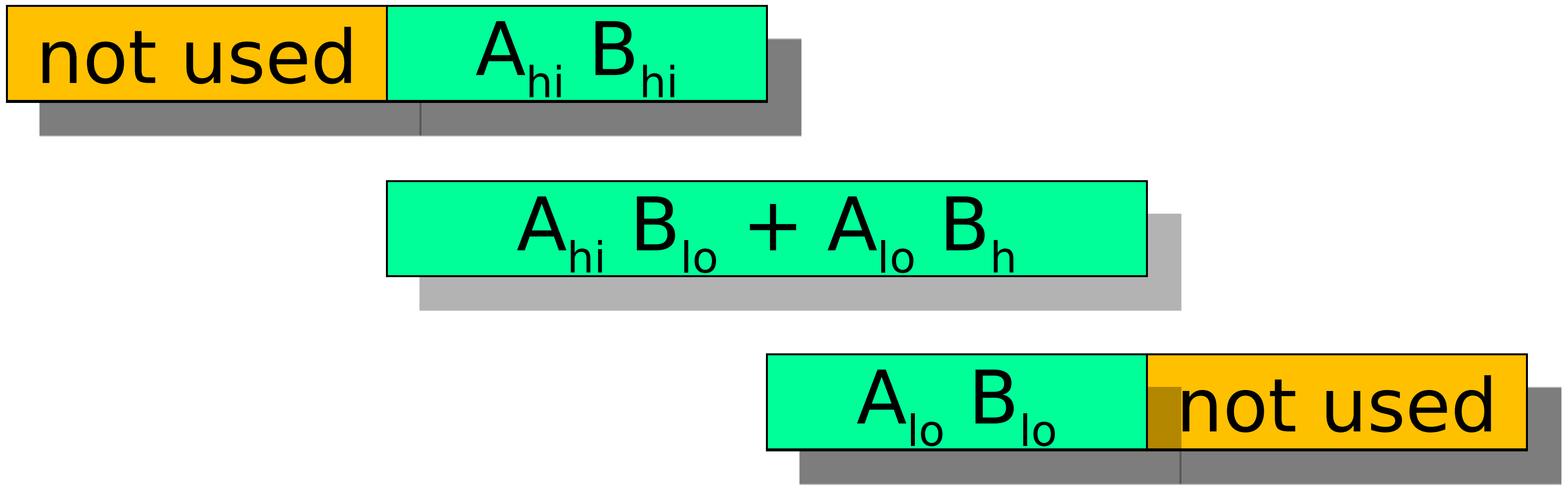
Strategy:

1. Consider how to compute the 128-bit product of two 64-bit *unsigned* integers.
2. Modify that result to handle *signed* integers.
3. Note how discarding the 64 unused bits of the 128-bit product simplifies the computation.

32.32 Fixed-Point Multiplication

$$A_u B_u = (2^{32}A_{hi} + A_{lo})(2^{32}B_{hi} + B_{lo})$$

$$= 2^{64}A_{hi}\,B_{hi} + 2^{32}(A_{hi}\,B_{lo} + A_{lo}\,B_h) + A_{lo}\,B_{lo}$$



32.32 Fixed-Point Multiplication

First consider a 64-bit unsigned number:

$$\begin{aligned} A_u &= 2^{63}A_{63} + 2^{62}A_{62} + \ldots + 2^{0}A_{0} \\ &= 2^{63}A_{63} + (2^{62}A_{62} + \ldots + 2^{0}A_{0}) \\ &= 2^{63}A_{63} + A_{62..0} \end{aligned}$$

where $A_{62..0} = 2^{62}A_{62} + \ldots + 2^{0}A_{0}$

32.32 Fixed-Point Multiplication

Thus the 128-bit product of two 64-bit unsigned operands would be:

$$A_u B_u = (2^{63} A_{63} + A_{62..0})(2^{63} B_{63} + B_{62..0})$$

$$= 2^{126} A_{63} B_{63} + 2^{63} (A_{63} B_{62..0} + B_{63} A_{62..0})$$

$$+ A_{62..0} B_{62..0}$$

32.32 Fixed-Point Multiplication

Now consider a 64-bit signed number:

$$A_s = -2^{63}A_{63} + 2^{62}A_{62} + \ldots + 2^{0}A_{0}$$

$$= -2^{63}A_{63} + (2^{62}A_{62} + \ldots + 2^{0}A_{0})$$

$$= -2^{63}A_{63} + A_{62..0}$$

32.32 Fixed-Point Multiplication

Thus the 128-bit product of two 64-bit signed operands would be:

$$A_s B_s = (-2^{63}A_{63} + A_{62..0})(-2^{63}B_{63} + B_{62..0})$$

$$= 2^{126}A_{63}B_{63} - 2^{63(}A_{63} B_{62..0} + B_{63} A_{62..0})$$

$$+ A_{62..0} B_{62..0}$$

Unsigned vs. Signed Multiplication

$A_u B_u = 2^{126} A_{63} B_{63}$

$+ 2^{63}(A_{63} B_{62..0} + B_{63} A_{62..0})$

$+ A_{62..0} B_{62..0}$

$A_s B_s = 2^{126} A_{63} B_{63}$

$- 2^{63}(A_{63} B_{62..0} + B_{63} A_{62..0})$

$+ A_{62..0} B_{62..0}$

32.32 Fixed-Point Multiplication

Thus the 128-bit product of two 64-bit signed operands would be:

$$A_sB_s = A_uB_u - 2\,(2^{63}A_{63}\,B_{62..0} + 2^{63}B_{63}\,A_{62..0})$$

$$= A_uB_u - 2^{64}A_{63}\,B_{62..0} - 2^{64}B_{63}\,A_{62..0}$$

32.32 Fixed-Point Multiplication

What does this result mean?

$$A_s B_s = A_u B_u - 2^{64} A_{63} B_{62..0} - 2^{64} B_{63} A_{62..0}$$

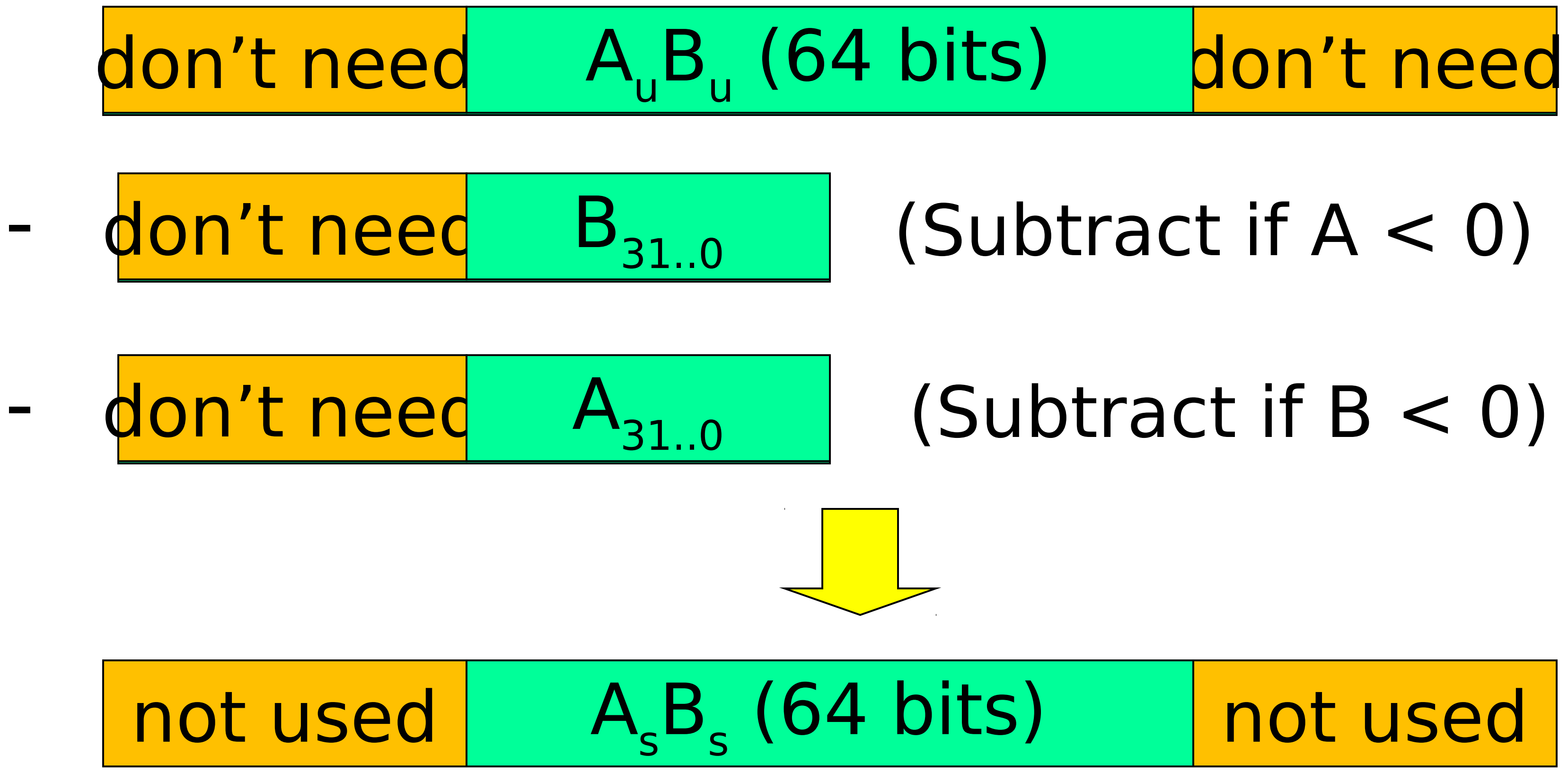
If A is negative, subtract $B_{62..0}$ from the most-significant half of $A_u B_u$

$A_u B_u$

If B is negative, subtract $A_{62..0}$ from the most-significant half of $A_u B_u$

$A_u B_u$

32.32 Fixed-Point Multiplication


don't need
A_uB_u (64 bits)
don't need
-
don't need
$B_{31..0}$
(Subtract if A < 0)
-
don't need
$A_{31..0}$
(Subtract if B < 0)
not used
A_sB_s (64 bits)
not used