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 <u>a single bit pattern can represent two different values.</u>
- 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? */

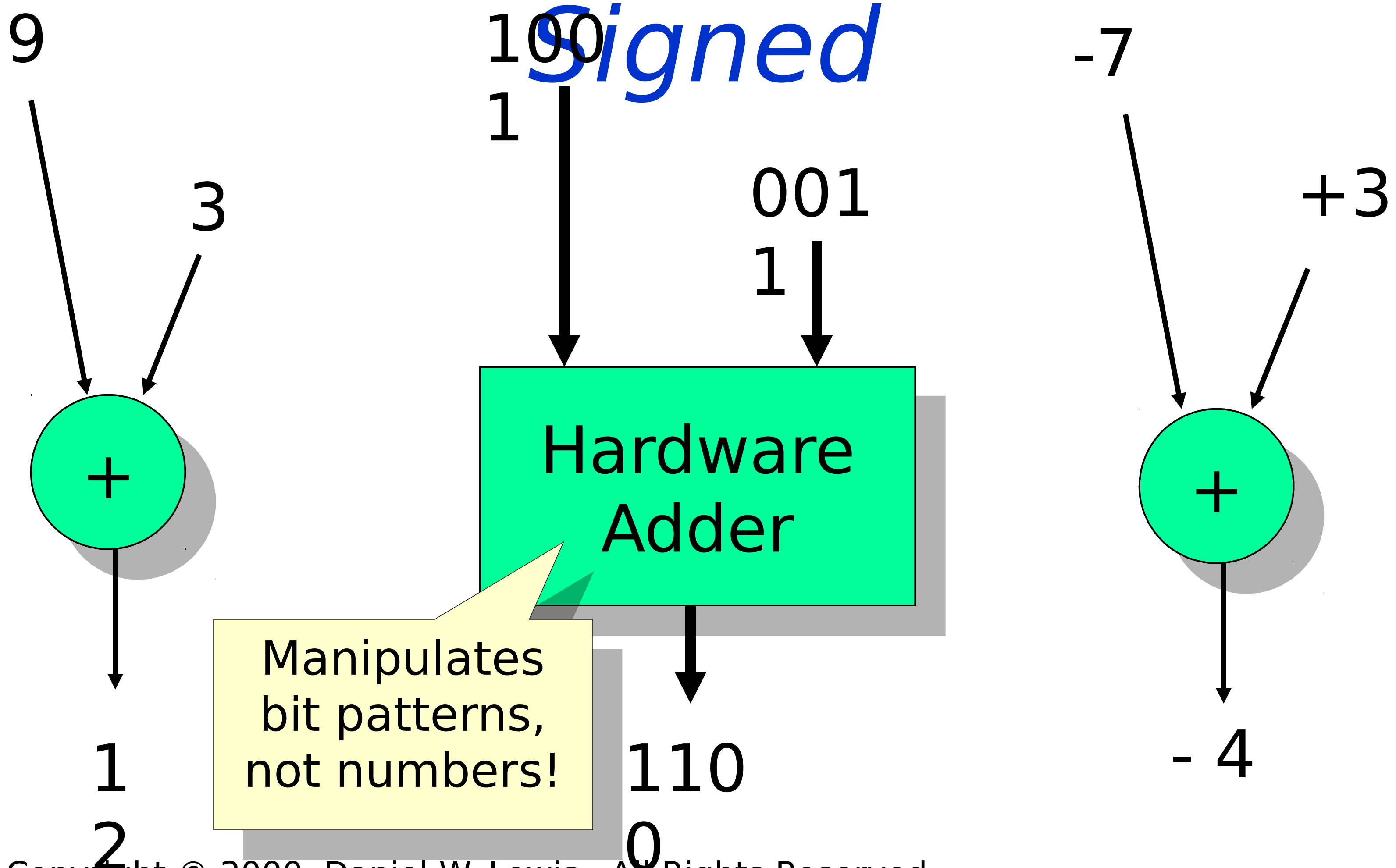
It’s a matter of interpretation, and depends on how x and y were declared: signed? Or unsigned?

Which is Greater: 1001 or 0011?

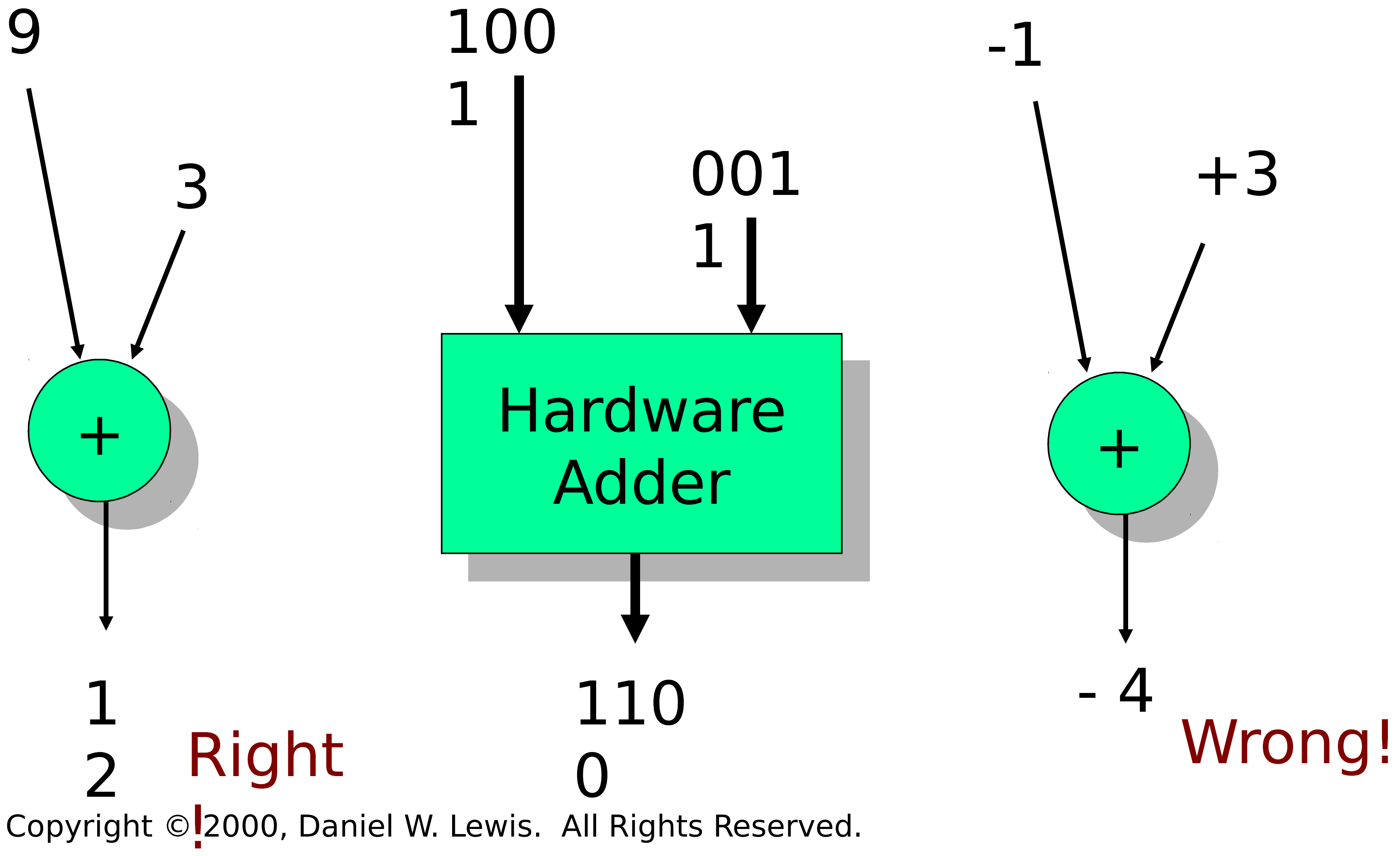
```
signed int x, y ;          MOV EAX,[x]
                    CMP  EAX,[y]
if (x > y) ...          ⇒    JLE
Skip_Then_Clause
```

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

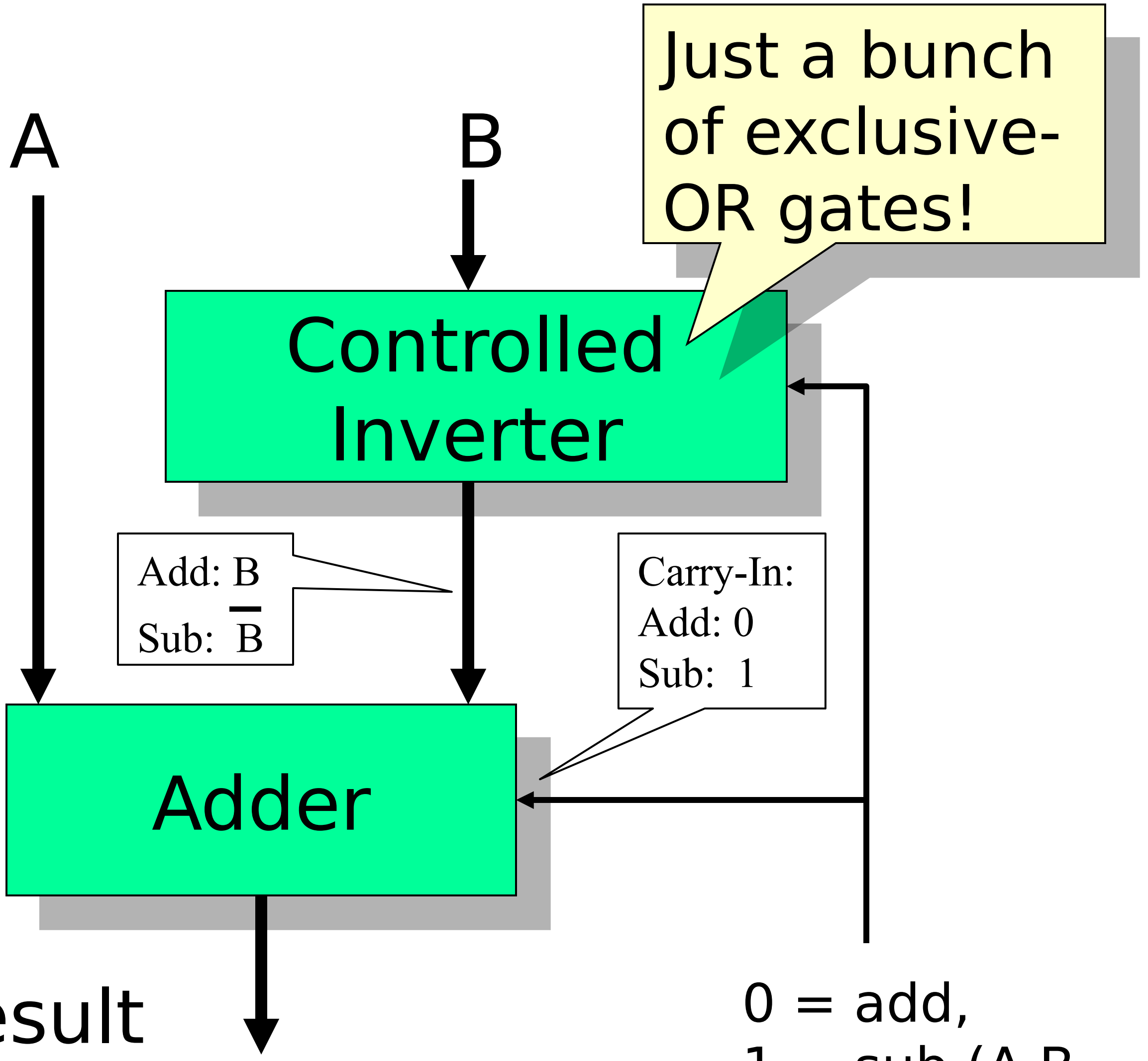
One Hardware Adder Handles Both: *Unsigned and 2's Complement Signed*



Why Not Sign+Magnitude?



Subtraction Is Easy!



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

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

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

$= (N \ll 3) + (N \ll 2) + N$

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

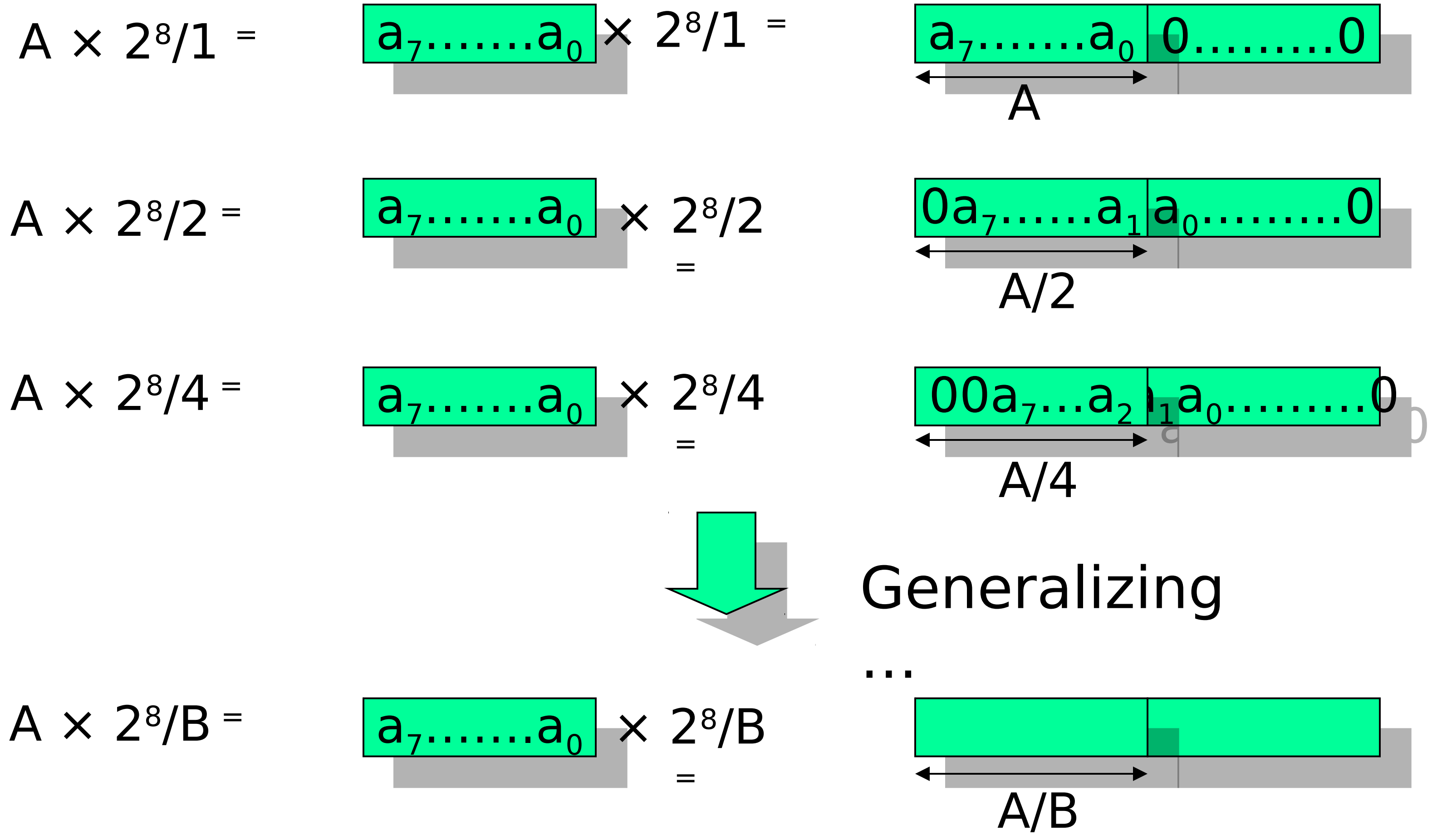
- This requires 4 shifts and 3 additions.

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!



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 16×16 Multiplication

A÷397 ≈ 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

Converting 4-bit Unsigned Product to 2’s Complement Product

$A_uB_u = (2^3A_3 + 2^2A_2 + 2^1A_1 + 2^0A_0)(2^3B_3 + 2^2B_2 + 2^1B_1 + 2^0B_0)$

Let “$A_{2..0}$” represent all of $2^2A_2 + 2^1A_1 + 2^0A_0$

And “$B_{2..0}$” represent all of $2^2B_2 + 2^1B_1 + 2^0B_0$

$= (2^3A_3 + A_{2..0})(2^3B_3 + B_{2..0})$

$= 2^6A_3B_3 + 2^3(A_3B_{2..0} + B_3A_{2..0}) + A_{2..0}B_{2..0}$

Converting 4-bit Unsigned Product to 2's Complement Product

$$A_sB_s = (-2^3A_3 + 2^2A_2 + 2^1A_1 + 2^0A_0)(-2^3B_3 + 2^2B_2 + 2^1B_1 + 2^0B_0)$$

$$= (-2^3A_3 + A_{2..0})(-2^3B_3 + B_{2..0})$$

$$= 2^6A_3B_3 - 2^3(A_3B_{2..0} + B_3A_{2..0}) + A_{2..0}B_{2..0}$$

Converting 4-bit Unsigned Product to 2's Complement Product

$A_uB_u = 2^6A_3B_3$ **+ $2^3(A_3B_{2..0}$ + $B_3A_{2..0})$** $+ A_{2..0}B_{2..0}$

$A_sB_s = 2^6A_3B_3$ **- $2^3(A_3B_{2..0}$ + $B_3A_{2..0})$** $+ A_{2..0}B_{2..0}$

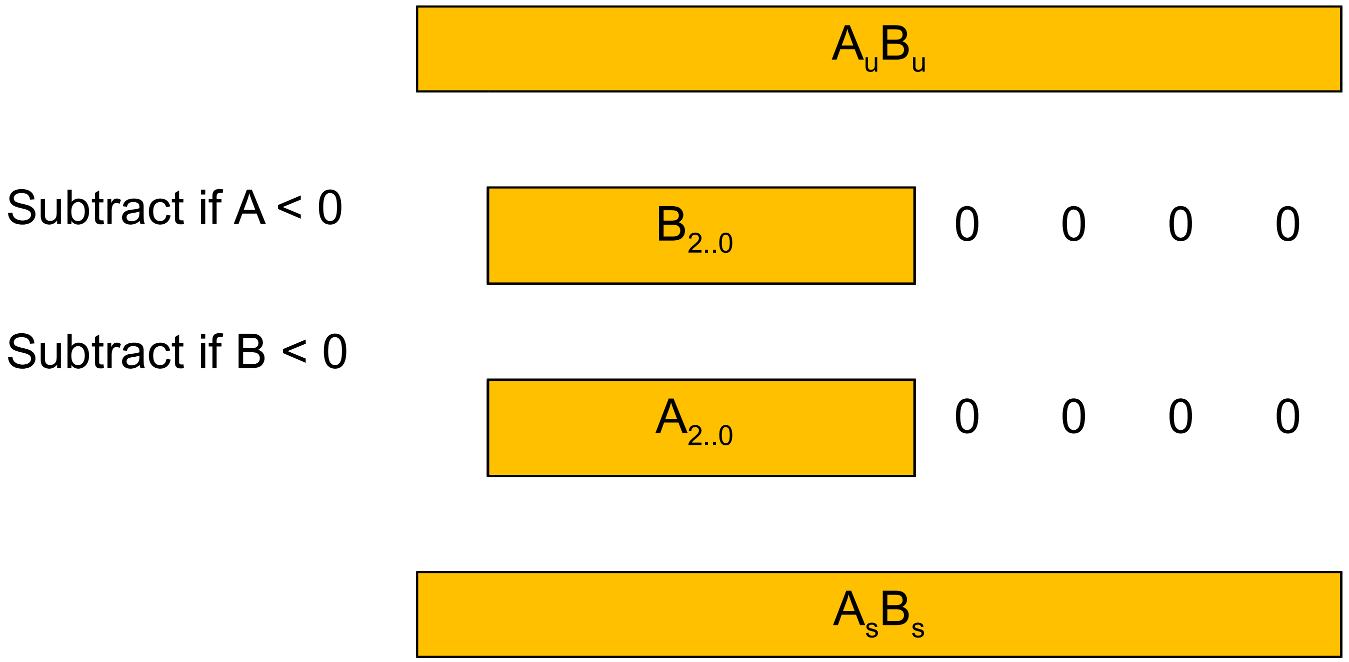
Thus: $A_sB_s = A_uB_u - 2\times$ **$2^3(A_3B_{2..0}$ + $B_3A_{2..0})$**

$= A_uB_u - 2^4(A_3B_{2..0} + B_3A_{2..0})$

$= A_uB_u - 2^4A_3B_{2..0} - 2^4B_3A_{2..0}$

Converting 4-bit Unsigned Product to 2's Complement Product

$$A_sB_s = A_uB_u - 2^4A_3B_{2..0} - 2^4 B_3A_{2..0}$$



Converting 4-bit Unsigned Product to 2's Complement Product

3-bit operands

$$A_sB_s = A_uB_u - 2^4A_3\boxed{B_{2..0}} - 2^4B_3\boxed{A_{2..0}}$$

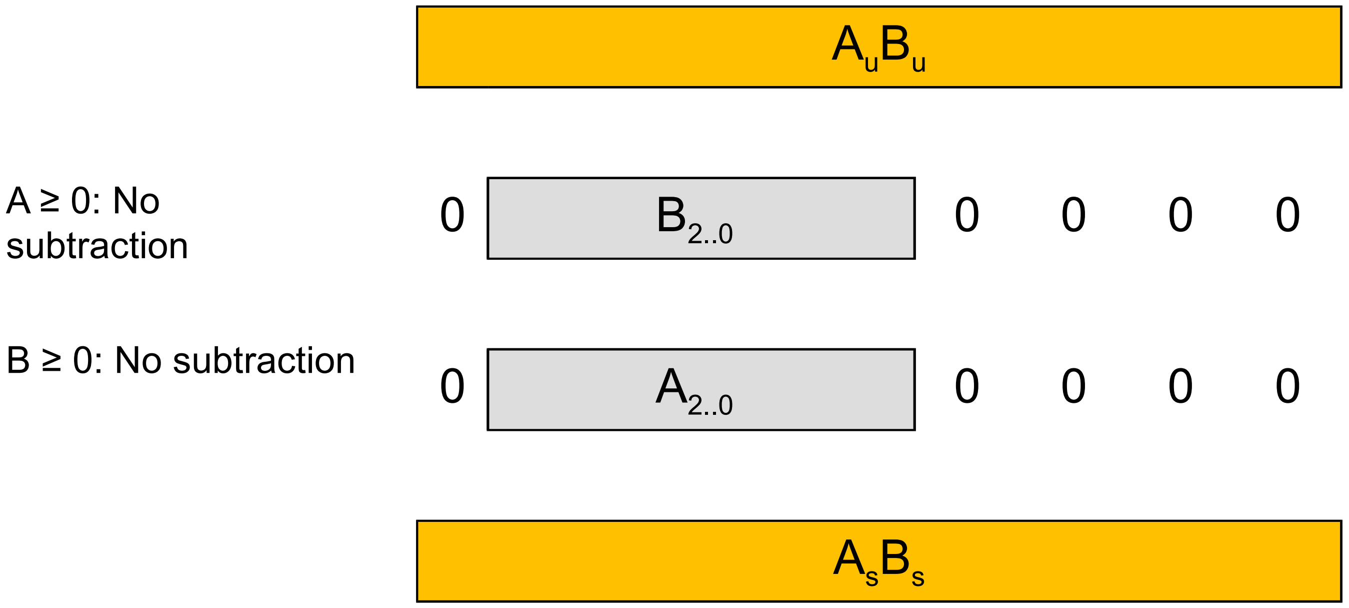
Same result using all 4 bits:

4-bit operands

$$A_sB_s = A_uB_u - 2^4A_3\boxed{B_s} - 2^4B_3\boxed{A_s}$$

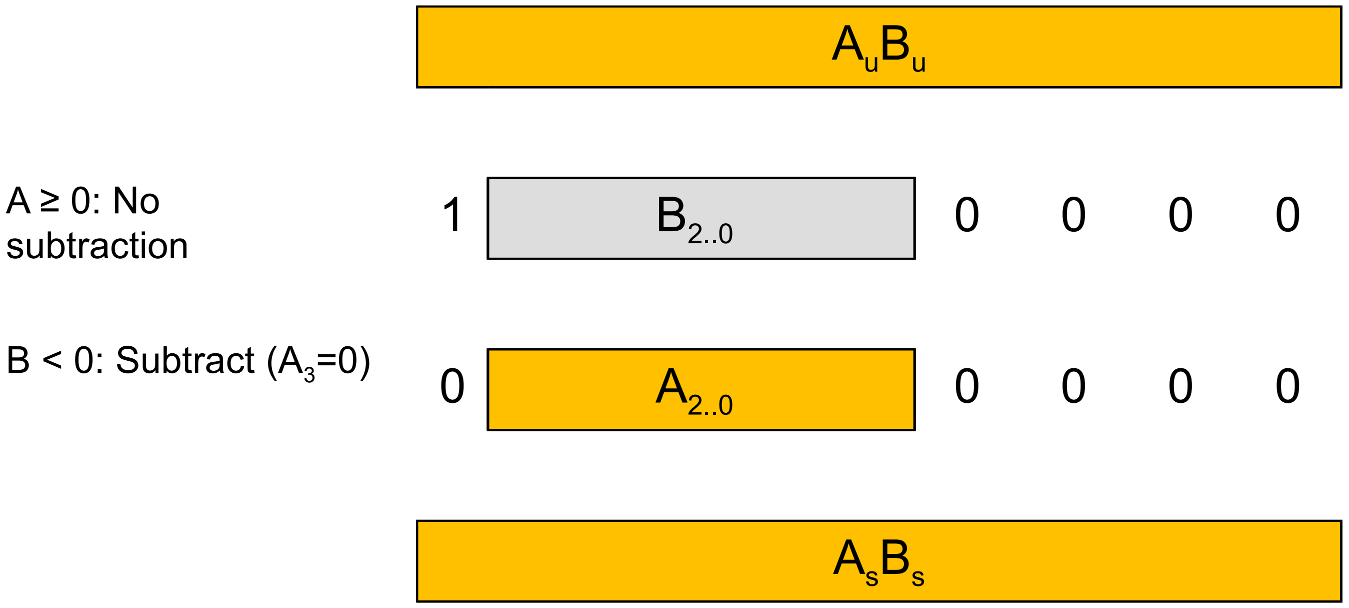
Converting 4-bit Unsigned Product to 2's Complement Product

Case 1: $A \geq 0$ and $B \geq 0$: No Impact on A_sB_s



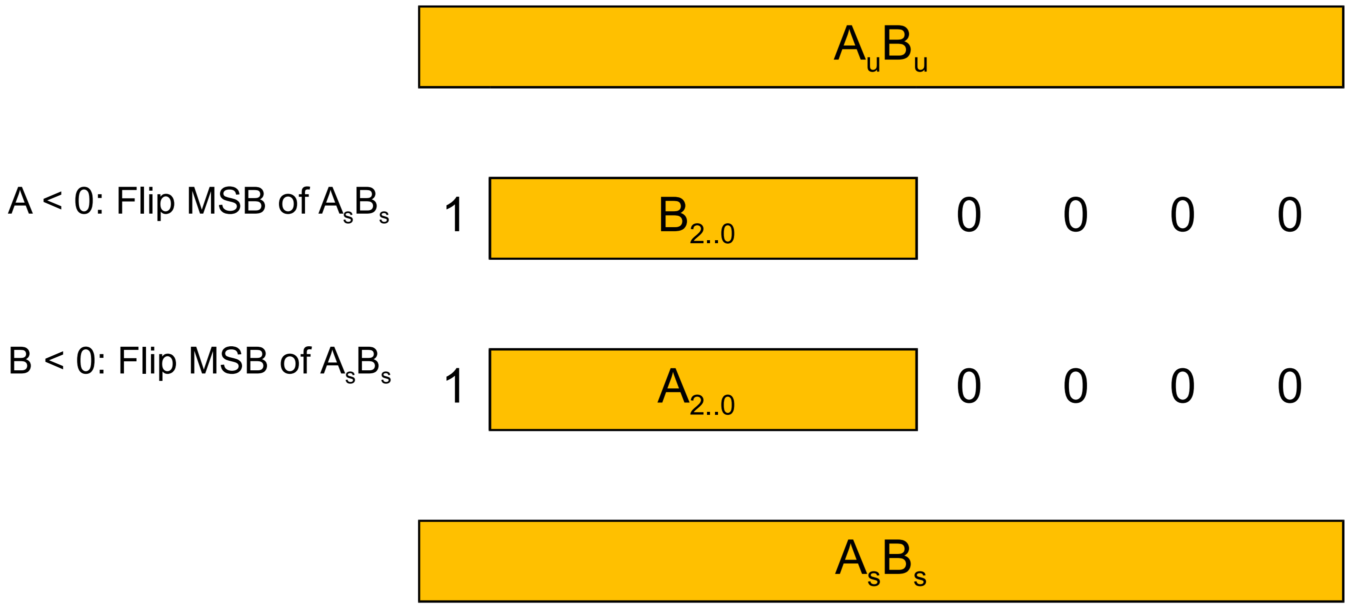
Converting 4-bit Unsigned Product to 2’s Complement Product

Case 2: A & B have different signs: No Impact on A_sB_s



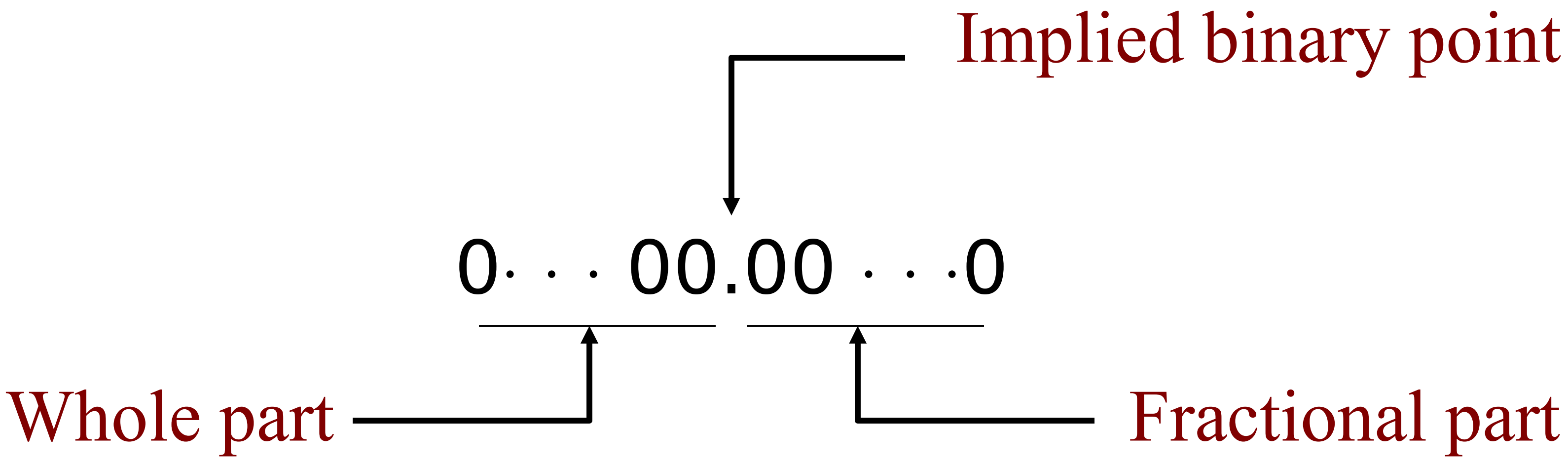
Converting 4-bit Unsigned Product to 2's Complement Product

Case 3: A < 0 and B < 0: Inverts MSB twice



Fixed-Point Reals

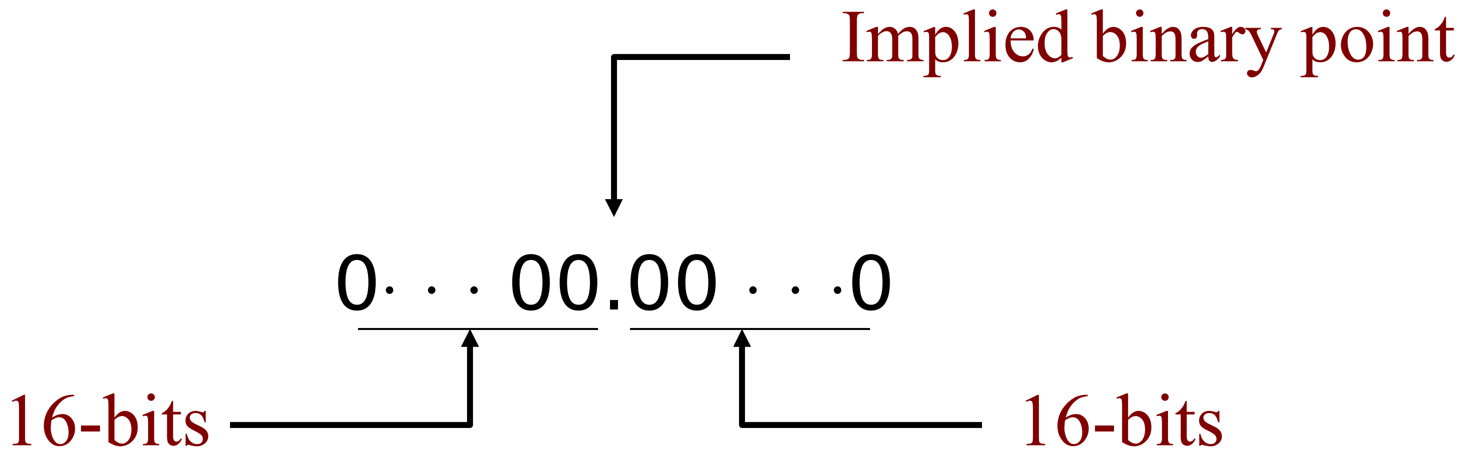
Three components:



Fixed vs. Floating

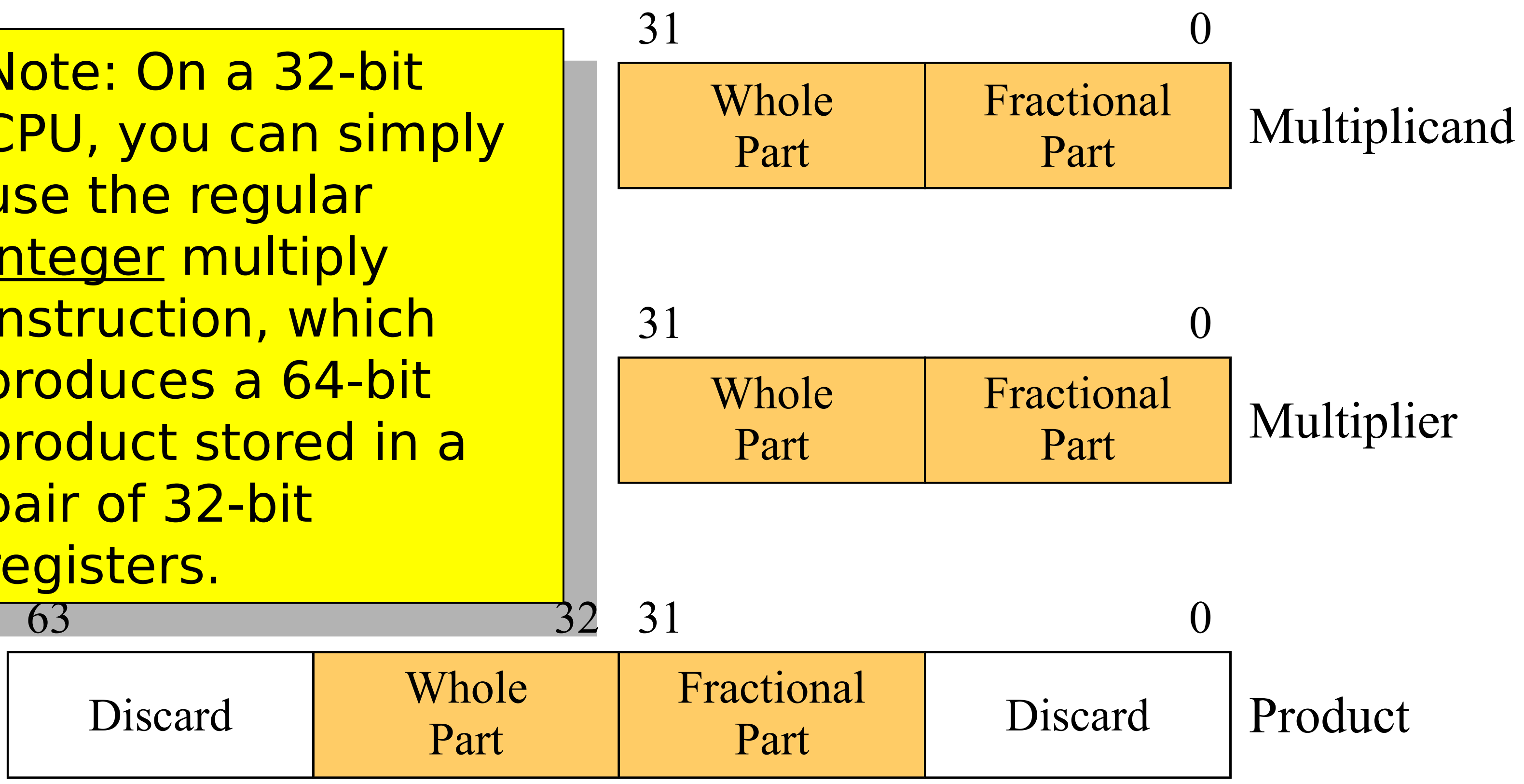
- Floating-Point:
  - Pro: Large dynamic range determined by exponent; resolution determined by significand.
  - Con: Implementation of arithmetic in hardware is complex (slow).
- Fixed-Point:
  - Pro: Arithmetic is implemented using regular integer operations of processor (fast).
  - Con: Limited range and resolution.

16.16 Fixed-Point Format

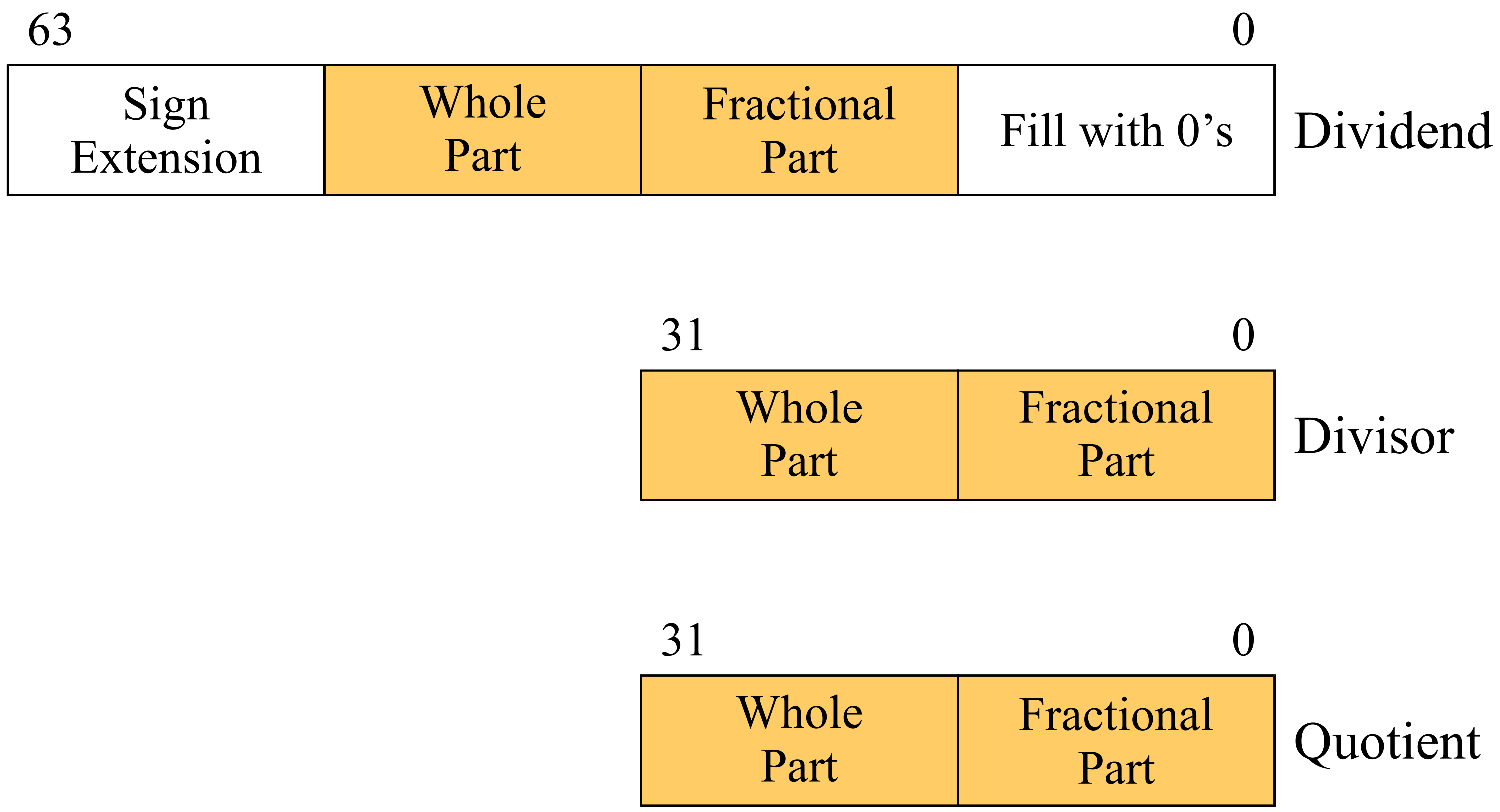


16.16 Fixed-Point Multiplication

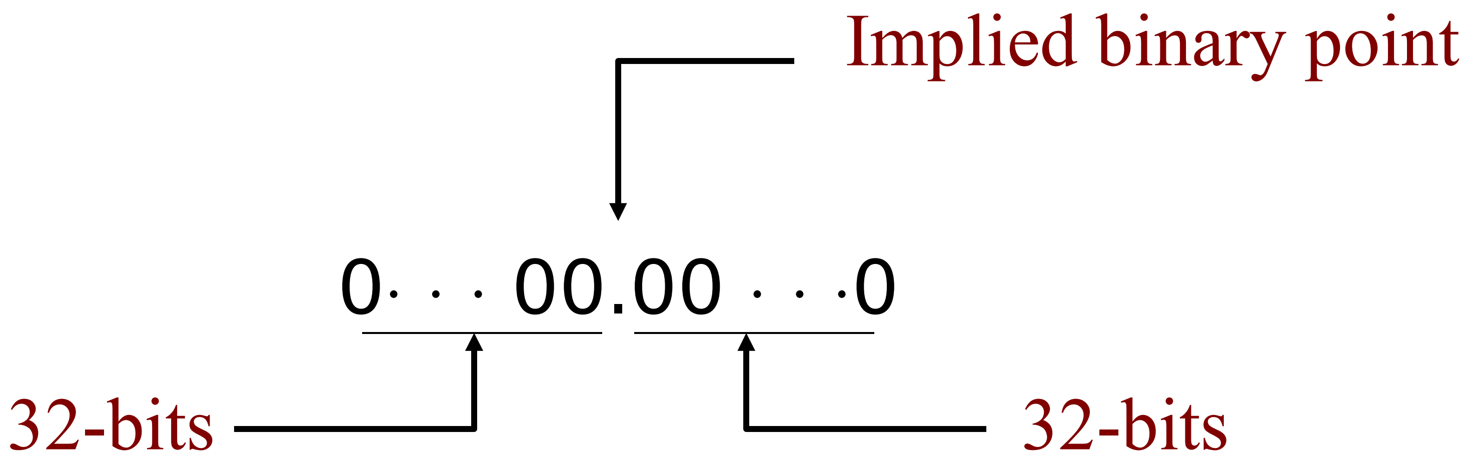
Note: On a 32-bit CPU, you can simply use the regular integer multiply instruction, which produces a 64-bit product stored in a pair of 32-bit registers.



16.16 Fixed-Point Division



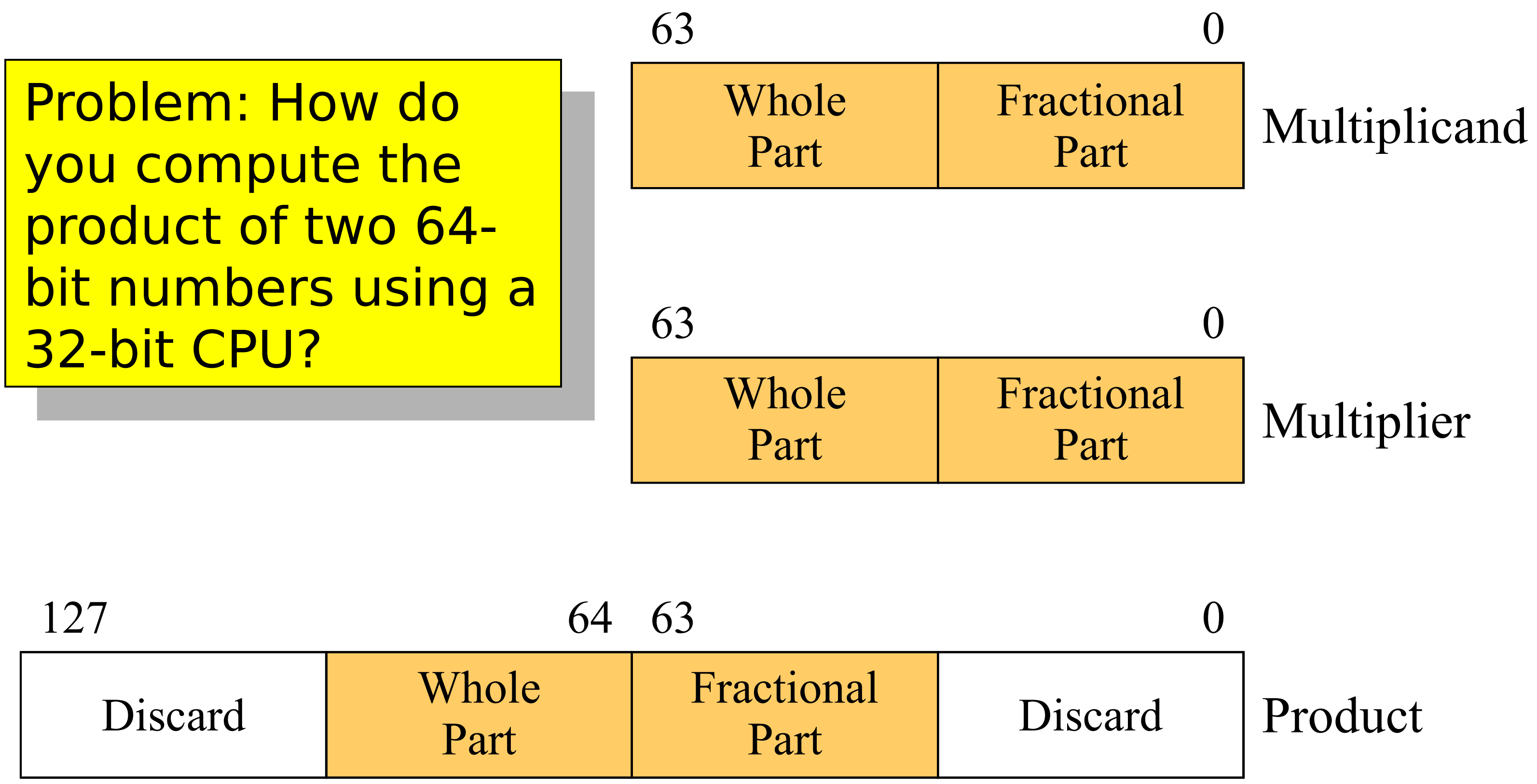
"Brute-Force" 32.32 Format



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

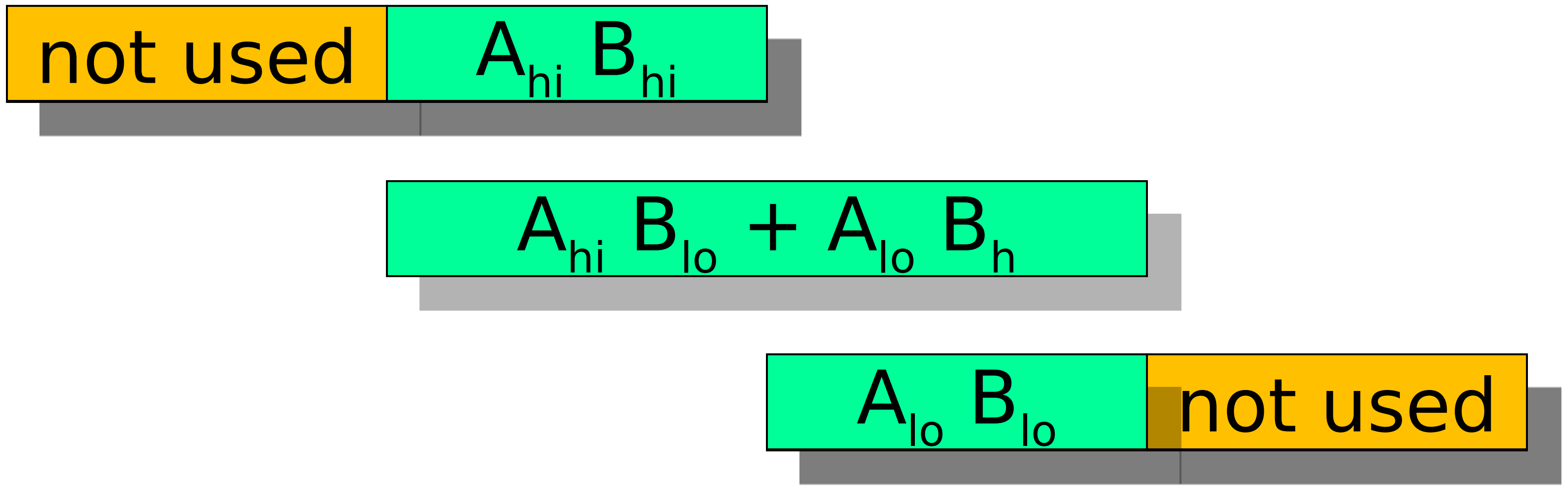
Strategy:

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:

$$\begin{aligned} A_s &= -2^{63}A_{63} + 2^{62}A_{62} + \ldots + 2^0A_0 \\ &= -2^{63}A_{63} + (2^{62}A_{62} + \ldots + 2^0A_0) \\ &= -2^{63}A_{63} + A_{62..0} \end{aligned}$$

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

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

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

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

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

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

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

32.32 Fixed-Point Multiplication

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

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

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

$$= A_uB_u - 2^{64}A_{63}B_u - 2^{64}B_{63}A_u$$

32.32 Fixed-Point Multiplication

What does this result mean?

$$A_sB_s = A_uB_u - 2^{64}A_{63}B_u - 2^{64}B_{63}A_u$$

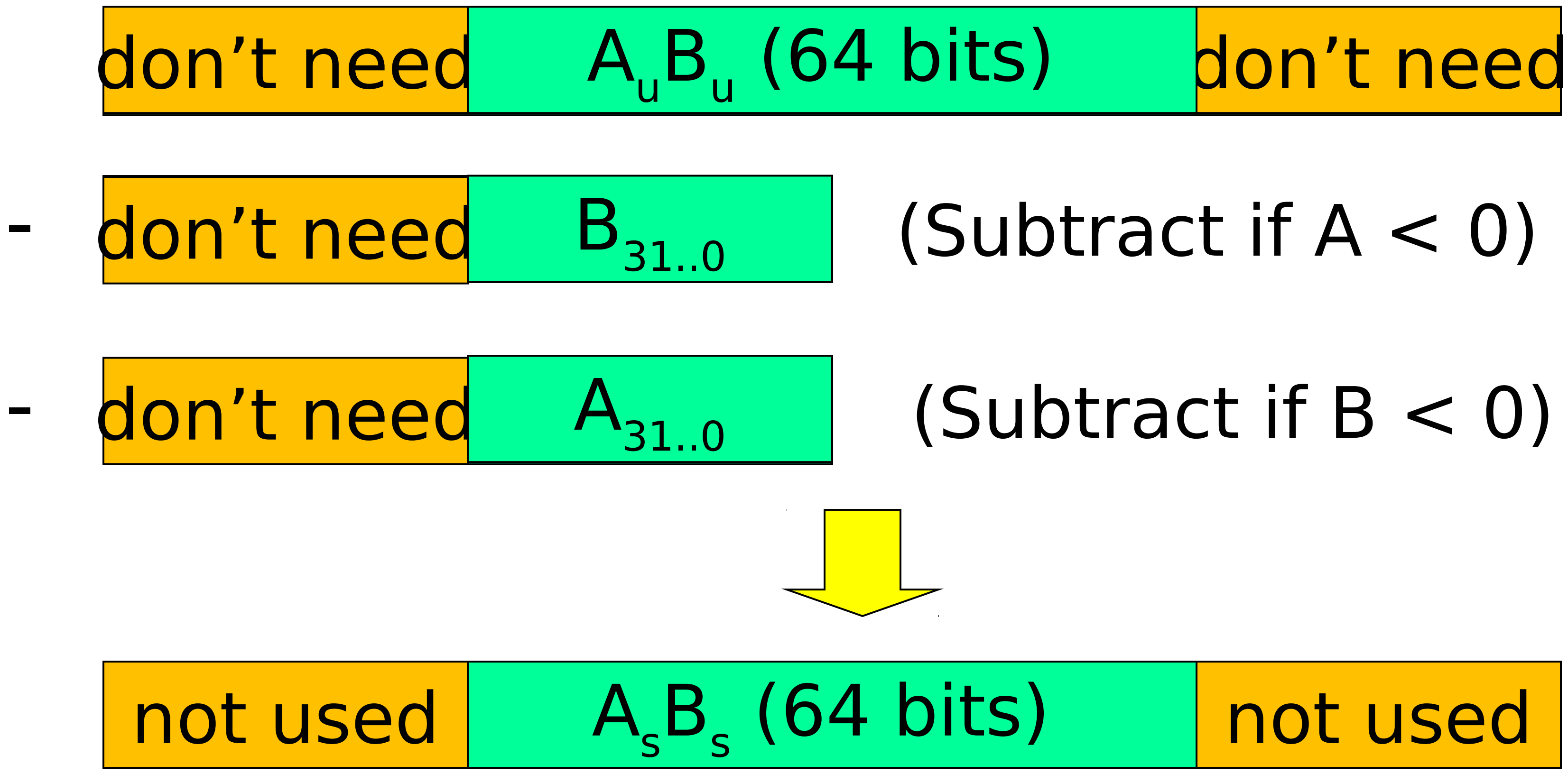
If A is negative, subtract B_u from the most-significant half of A_uB_u

A_uB_u

If B is negative, subtract A_u from the most-significant half of A_uB_u

A_uB_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