[Chap.2-2] Representing and Manipulating Information

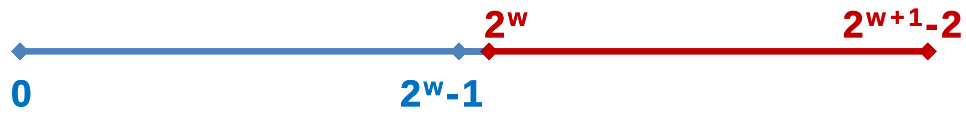
Young Ik Eom (yieom@skku.edu, 031-290-7120)
Distributing Computing Laboratory
Sungkyunkwan University
http://dclab.skku.ac.kr

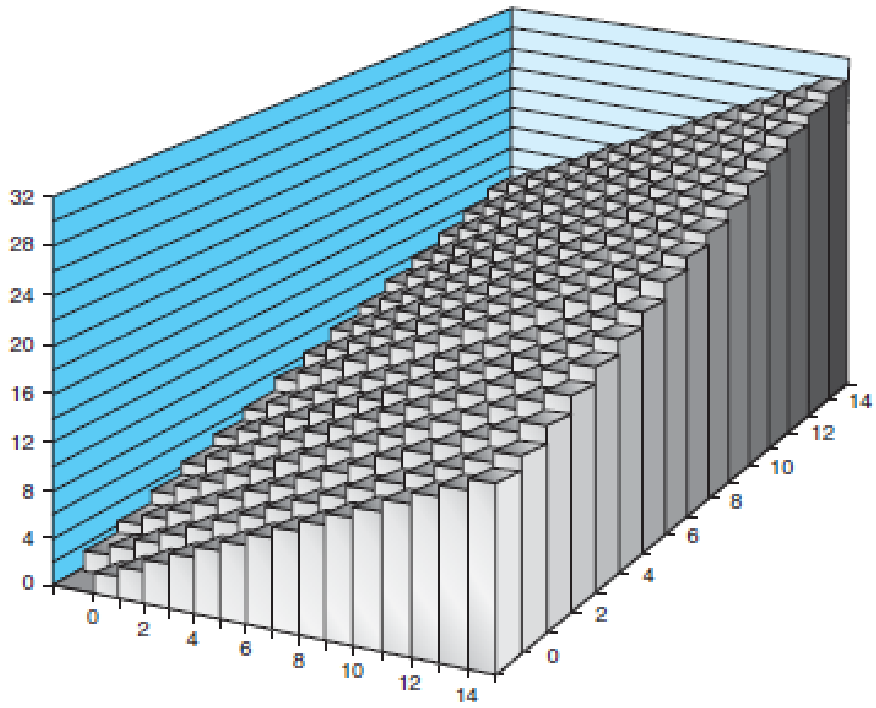
Contents

- Introduction
- Information storage
- Integer representations
- Integer arithmetic
- Floating point
- Summary

Integer Arithmetic

- **Unsigned addition**
  - ***w***-bit integers ***x***, ***y***
  - Compute the true sum
    - $0 \le x + y \le 2^{w+1} - 2$
  - True sum requires one more bit ("carry")
    - Values increase linearly with ***x*** and ***y***
    - Forms planar surface

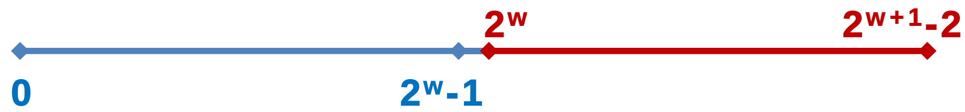




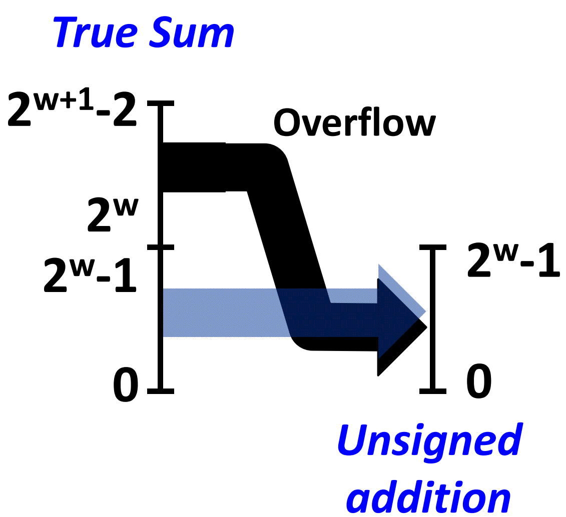
Integer Arithmetic

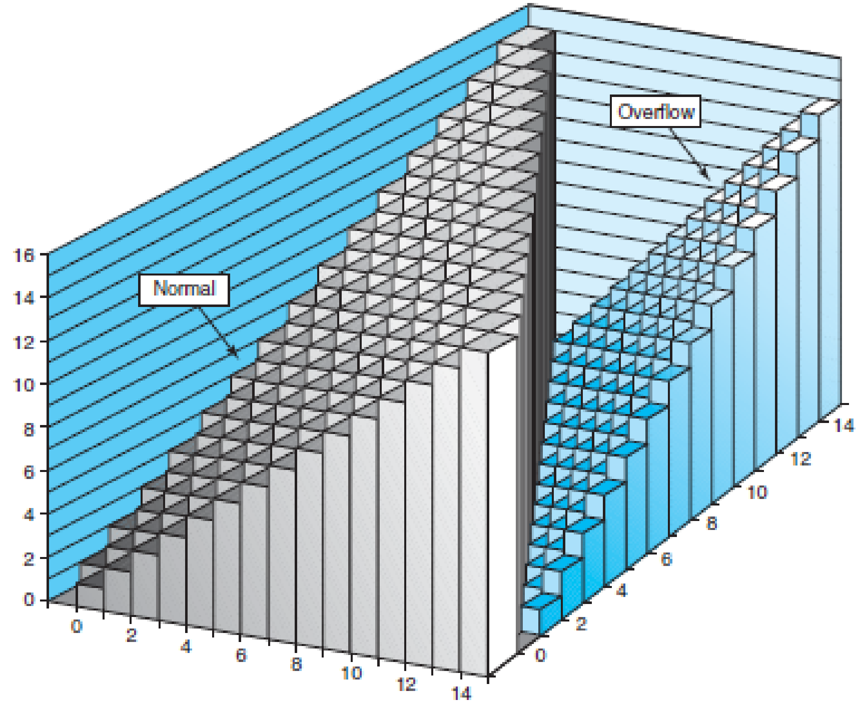
Unsigned addition

- Ignores carry output
- Wraps around when true sum $\geq 2^w$



$$x +_w^u y = \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \leq x + y \end{cases}$$





Integer Arithmetic

Unsigned addition

- Example-1) 4-bit unsigned integers
  - (Decimal) 3 + 5 → 8
  - (Binary) 0011 + 0101 → 1000
- Example-2) 4-bit unsigned integers (overflow)
  - (Decimal) 9 + 12 → 21 – 16 → 5
  - (Binary) 1001 + 1100 → 10101 → 0101

Integer Arithmetic

Unsigned addition in C

- When executing C programs,
  overflows are not signaled as errors

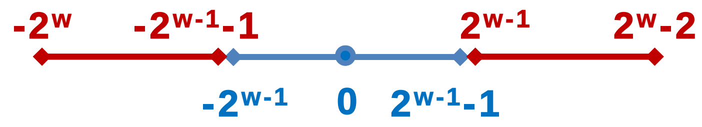
  $$s \doteq x +^{u}_{w} y$$

- To determine whether overflow has occurred
  - Check ***s*** < ***x*** (or ***s*** < ***y***)

Integer Arithmetic

Signed (2's-complement) addition

- ***w***-bit integers ***x***, ***y***
  - $-2^{w-1} \le x, y \le 2^{w-1} - 1$

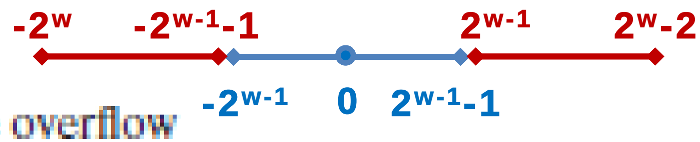


- Compute the true sum
  - $-2^{w} \le x + y \le 2^{w} - 2$
- True sum requires one more bit ("carry")

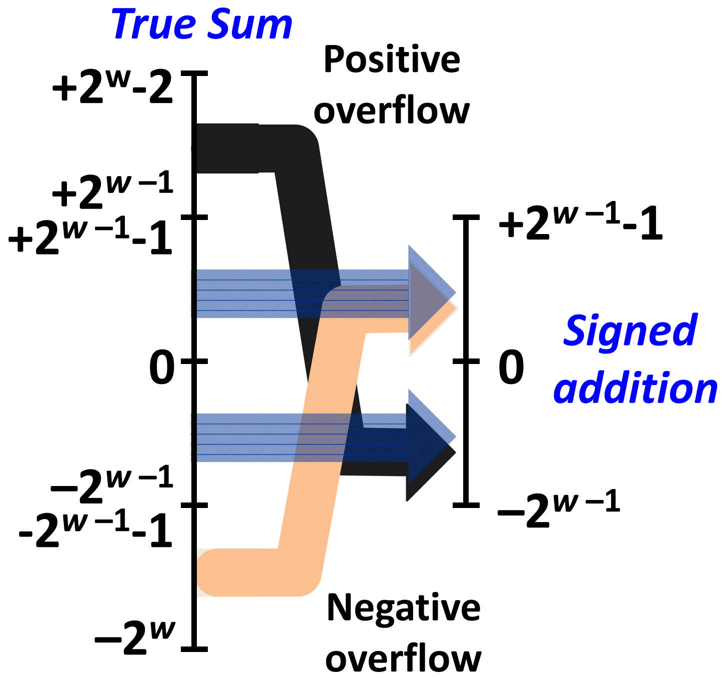
- But, in signed addition (in C),
  - The leading (carry) bit is truncated and
  - Treat the remaining bits as 2's-complement integer

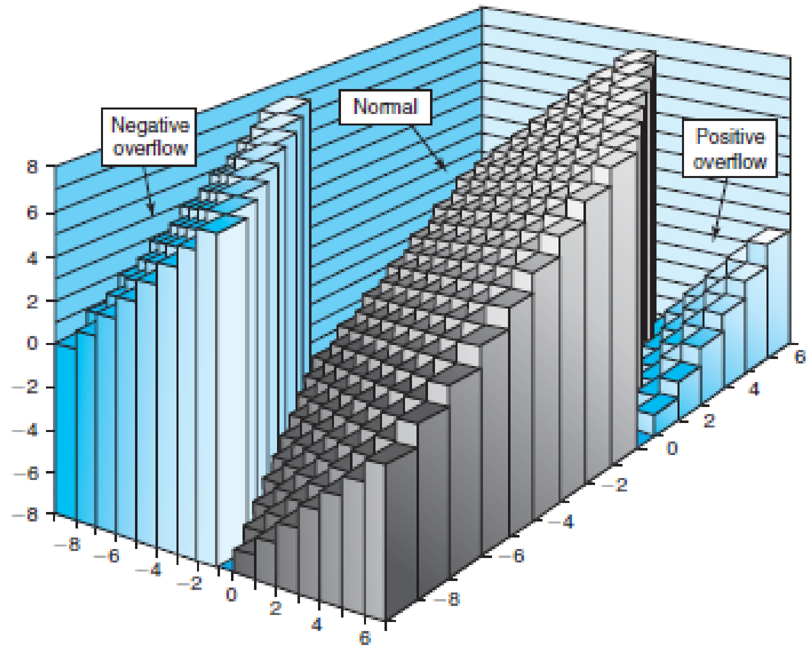
Integer Arithmetic

- Signed (2's-complement) addition



$$x +^t_w y = \begin{cases} x+y-2^w, & 2^{w-1} \le x+y \quad \text{Positive overflow} \\ x+y, & -2^{w-1} \le x+y < 2^{w-1} \quad \text{Normal} \\ x+y+2^w, & x+y < -2^{w-1} \quad \text{Negative overflow} \end{cases}$$





Integer Arithmetic

Signed (2's-complement) addition

- Example) 4-bit integers (-8 ~ 7)

x	y	$x+y$	$x +^{t}_{4} y$	
-8 [1000]	-5 [1011]	-13 [10011]	3 [0011]	Negative overflow
-8 [1000]	-8 [1000]	-16 [10000]	0 [0000]	Negative overflow
-8 [1000]	5 [0101]	-3 [11101]	-3 [1101]	Normal
2 [0010]	5 [0101]	7 [00111]	7 [0111]	Normal
5 [0101]	5 [0101]	10 [01010]	-6 [1010]	Positive overflow

Integer Arithmetic

- **Signed (2's-complement) addition**
  - Notes)
    - When both x and y are negative, but the sum ≥ 0
      - ✓ Negative overflow
    - When both x and y are positive, but the sum < 0
      - ✓ Positive overflow

Integer Arithmetic

2's-complement negation

- ***w***-bit integers ***x***
  - $-2^{w-1} \le x \le 2^{w-1} - 1$
- Compute the true negation
  - $-(2^{w-1} - 1) \le -x \le 2^{w-1}$

- No problem when $-2^{w-1} < x \le 2^{w-1} - 1$
- But, when $x = -2^{w-1}$, $-x$ can not be represented as a ***w***-bit number
  - In this case, 2's-complement negation of ***x*** becomes -2^{w-1}

$$-^t_w x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases}$$

Integer Arithmetic

■ 2's-complement negation

- Example)

x		$-x$	
[1100]	–4	[0100]	4
[1000]	–8	[1000]	–8
[0101]	5	[1011]	–5
[0111]	7	[1001]	–7

Integer Arithmetic

Unsigned multiplication

- ***w***-bit unsigned integers ***x***, ***y***
  - $0 \le x, y \le 2^w - 1$
- Maximum value of the product $x \cdot y$
  - $0 \le x \cdot y \le 2^{2w} - 2^{w+1} + 1$
- Requires $2w$ bits to represent $x \cdot y$

- But, in unsigned multiplication (in C),
  - The multiplication yields to the ***w***-bit value given by the low-order ***w*** bits of the $2w$-bit integer product

Integer Arithmetic

Unsigned multiplication in C

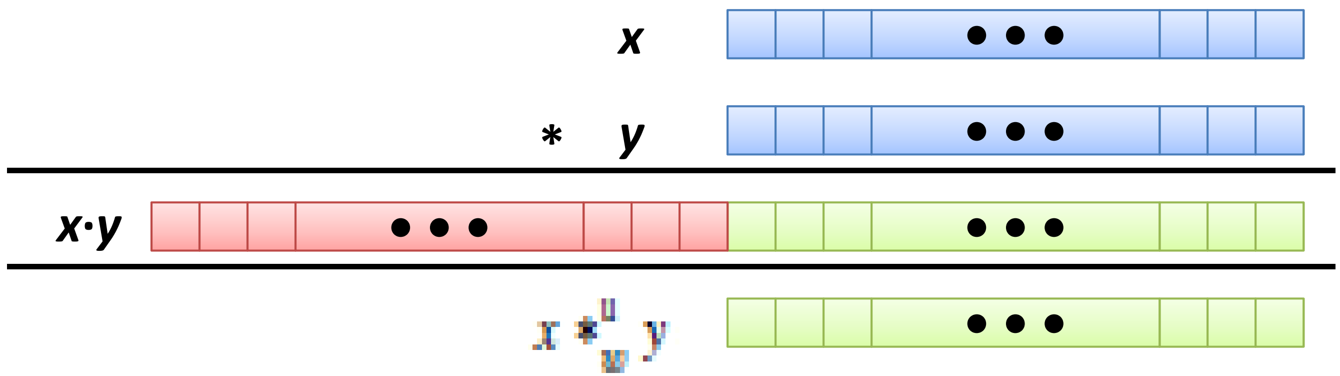
- Ignores high order **w** bits
- Implements modular arithmetic

$$x *_w^u y = (x \cdot y) \bmod 2^w$$

Operands (*w* bits)

True product (2*w* bits)

Discard *w* bits (*w* bits)



Integer Arithmetic

2's-complement multiplication

- ***w***-bit signed integers ***x***, ***y***
  - $-2^{w-1} \le x, y \le 2^{w-1} - 1$
- The product $x \cdot y$ can range between
  - $-2^{w-1} \cdot (2^{w-1} - 1) \le x \cdot y \le (-2^{w-1}) \cdot (-2^{w-1})$
  - $-2^{2w-2} + 2^{w-1} \le x \cdot y \le 2^{2w-2}$
  - Requires $2w$ bits to represent $x \cdot y$

- But, in 2's-complement multiplication (in C),
  - The multiplication yields to the ***w***-bit value
    given by the low-order ***w*** bits of the 2***w***-bit integer product

$$x *^{t}_{w} y = U2T_{w}((x \cdot y) \bmod 2^{w})$$

Integer Arithmetic

2's-complement multiplication

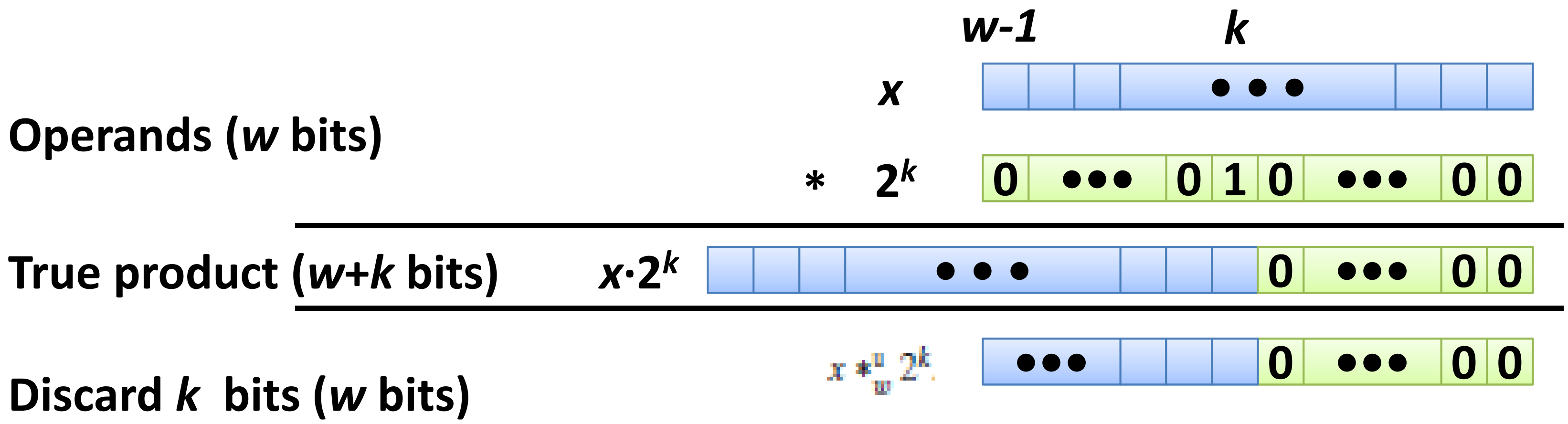
- Example) 3-bit integers (unsigned: 0~+7, signed: -4~+3)

Mode	x	y	x · y	Truncated x · y
Unsigned	5 [101]	3 [011]	15 [001111]	7 [111]
2's-complement	-3 [101]	3 [011]	-9 [110111]	-1 [111]
Unsigned	4 [100]	7 [111]	28 [011100]	4 [100]
2's-complement	-4 [100]	-1 [111]	4 [000100]	-4 [100]
Unsigned	3 [011]	3 [011]	9 [001001]	1 [001]
2's-complement	3 [011]	3 [011]	9 [001001]	1 [001]

Integer Arithmetic

Multiplying by constants

- Multiplication by 2^k ($\boldsymbol{x} \cdot 2^k$)
  - Left shift k-times ($\boldsymbol{x}$ << k)
  - Applicable to both unsigned and signed



Integer Arithmetic

Multiplying by constants

- Note)
  - In most machines, integer multiplication is much more costly than shifting and adding
    - ✓ 10 or more clock cycles for integer multiplication
    - ✓ 1 clock cycle for addition, subtraction, shift, and bit-level operations
  - Many C compilers try to substitute multiplication by constant with combinations of shifting, adding, and subtracting (automatically during translation and code generation)
  - Example)
    - ✓ x * 14 (14 = 8 + 4 + 2)
    - ✓ (x << 3) + (x << 2) + (x << 1)   // 14 = 8 + 4 + 2
    - ✓ (x << 4) – (x << 1)   // 14 = 16 -2

Integer Arithmetic

Multiplying by constants

- Example) Shift/add code generated automatically by the C compiler (for the multiply-by-constant statement)

C function

```
int mul12 (int x)
{
  return x * 12;
}
```

Compiled code

```
leal    (%eax, %eax, 2),  %eax        ;  t ← x + x * 2
sall    $2,  %eax                     ;  return  t << 2
```

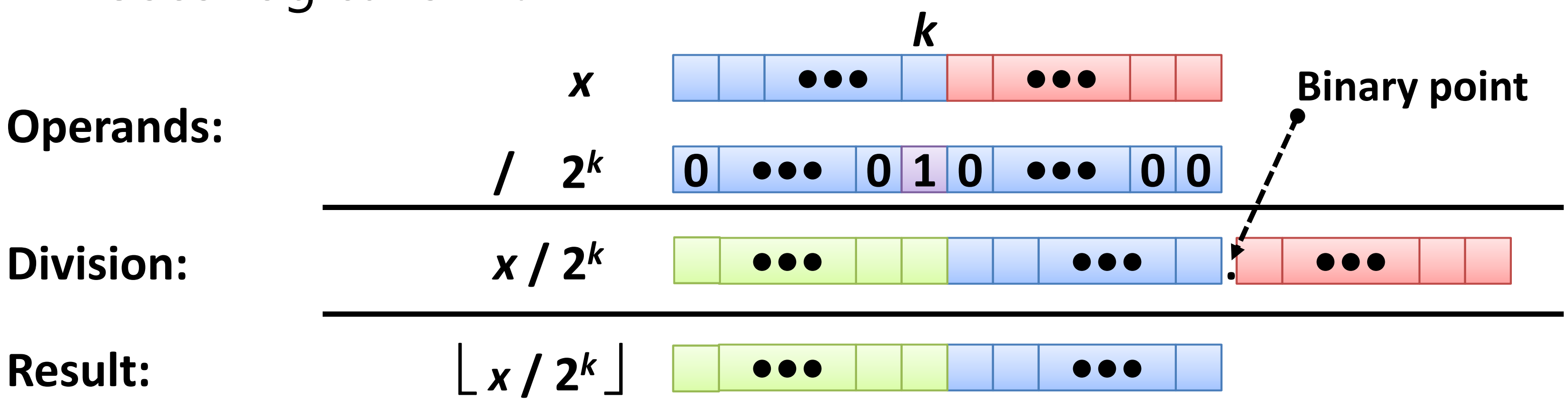
Integer Arithmetic

Division

- Even slower than integer multiplication
- 30 or more clock cycles
- Division by 2^{k} can be transformed to right shift operation
  - Division by arbitrary constant **k** cannot be expressed in terms of division by powers of 2

Integer Arithmetic

- **Division: Unsigned power-of-2 division and shift**
  - ***x >> k*** gives $\lfloor x / 2^k \rfloor$
  - Uses logical shift



Expression	Division	Result	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

Integer Arithmetic

Division: Unsigned power-of-2 division and shift

- Compiled unsigned division code
  - Uses logical shift for unsigned
  - Logical shift written as **>>>** in Java

C function

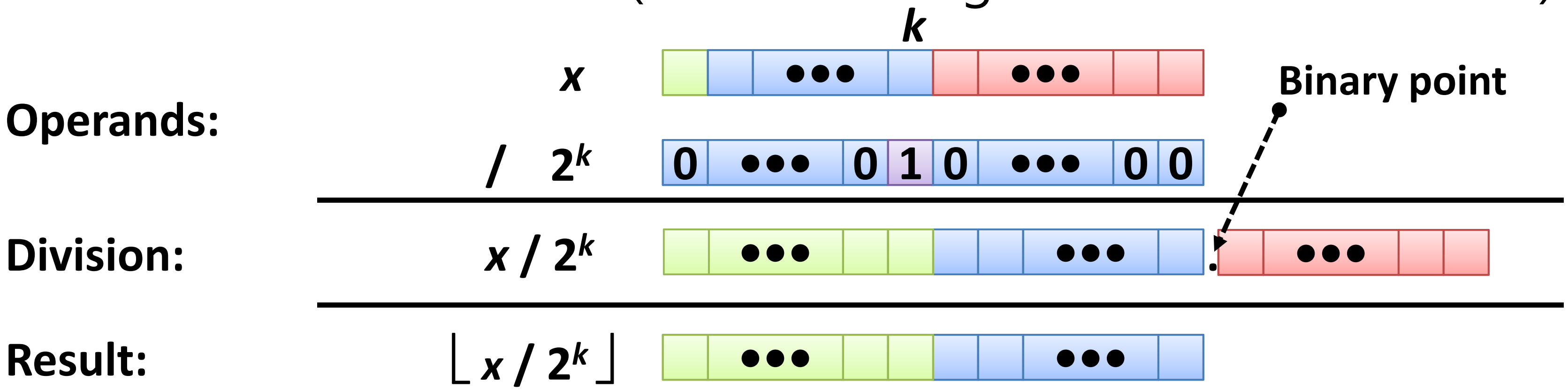
```
unsigned udiv8 (unsigned x)
{
  return x / 8;
}
```

Compiled code

```
shrl    $3,  %eax                    ;  return  t >> 3
```

Integer Arithmetic

- **Division: Signed power-of-2 division and shift**
  - ***x >> k*** gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift (rounds wrong direction when x < 0)



Expression	Division	Result	Hex	Binary
x	-12340	-12340	CF CC	11001111 11001100
x >> 1	-6170.0	-6170	E7 E6	11100111 11100110
x >> 4	-771.25	-772	FC FC	11111100 11111100
x >> 8	-48.203125	-49	FF CF	11111111 11001111

Integer Arithmetic

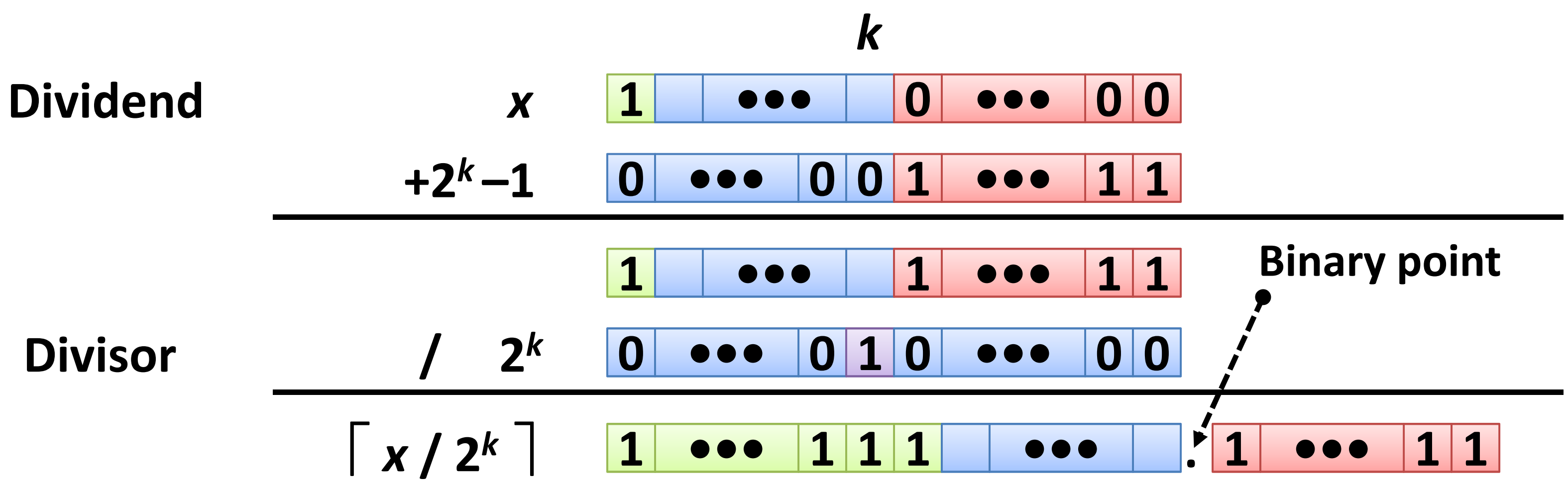
Division: Correct power-of-2 division

- We want $\lceil x / 2^k \rceil$ (Round Toward 0) when $x < 0$
- Compute as $\lfloor (x + 2^k - 1) / 2^k \rfloor$
  - In C, **(x + (1 << k) – 1) >> k**
  - Using the property $\lceil x / y \rceil = \lfloor (x + y - 1) / y \rfloor$
  - Biases dividend toward 0

Expression	Division	Result	Hex	Binary
x	-12340	-12340	CF CC	11001111 11001100
x >> 1	-6170.0	-6170	E7 E6	11100111 11100110
x >> 4	-771.25	-771	FC FC	11111100 11111101
x >> 8	-48.203125	-48	FF CF	11111111 11010000

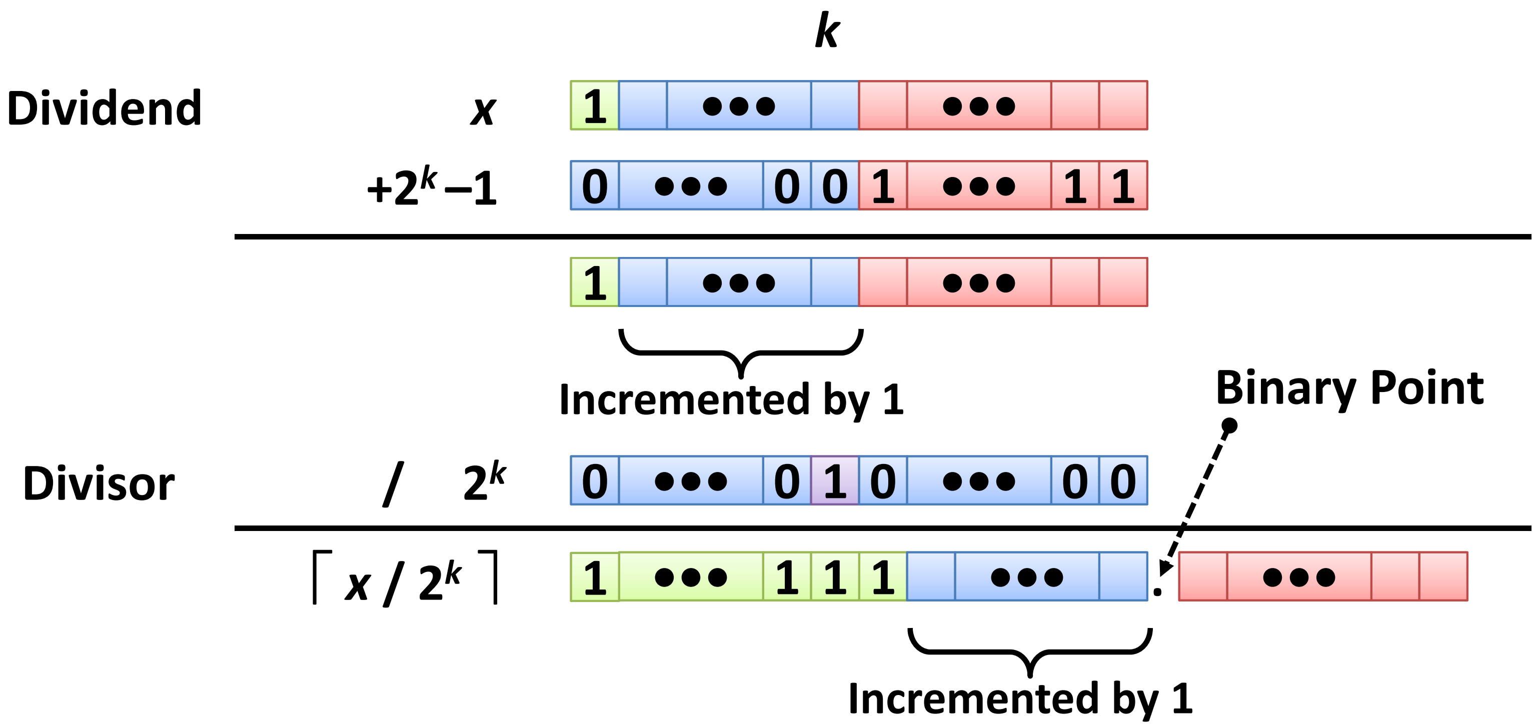
Integer Arithmetic

- **Division: Correct power-of-2 division**
  - Case 1: No rounding
    - Biasing has no effect



Integer Arithmetic

- **Division: Correct power-of-2 division**
  - Case 2: Rounding
    - Biasing adds 1 to the final result



Integer Arithmetic

- **Division: Correct power-of-2 division**
  - Compiled signed division code
    - Uses arithmetic shift for signed

C function

```
int idiv8 (int  x)
{
    return x / 8;
}
```

Compiled code

```
        testl    %eax,  %eax
        js       L4
L3:
        sarl     $3,  %eax
        ret
L4:
        addl     $7,  %eax
        jmp      L3
```

Explanation

```
if  (x  <  0)
    x  += 7;
return  x  >> 3;
```

Summary