Representing and Manipulating Integers Part II

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41` --> `0xBE`
    $\sim 01000001_2$ --> 10111110_2
  - `~0x00` --> `0xFF`
    $\sim 00000000_2$ --> 11111111_2
  - `0x69 & 0x55` --> `0x41`
    01101001_2 & 01010101_2 --> 01000001_2
  - `0x69 | 0x55` --> `0x7D`
    01101001_2 | 01010101_2 --> 01111101_2
  - `0x69 ^ 0x55` --> `0x4C`
    01101001_2 ^ 01010101_2 --> 00111100_2

Logic Operations in C

- **Contrast to logical operators**
  - `&&, ||, !`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- **Examples (char data type)**
  - `!0x41  -->  0x00`
  - `!0x00  -->  0x01`
  - `!!0x41 -->  0x01`

  - `0x69 && 0x55  -->  0x01`
  - `0x69 || 0x55  -->  0x01`
  - `if (p && *p)` (avoids null pointer access)

Shift Operations

- **Left shift: x << y**
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
    - Fill with 0's on right
- **Right shift: x >> y**
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate MSB on right
    - Useful with two's complement integer representation
- **Undefined if y < 0 or y ≥ word size**

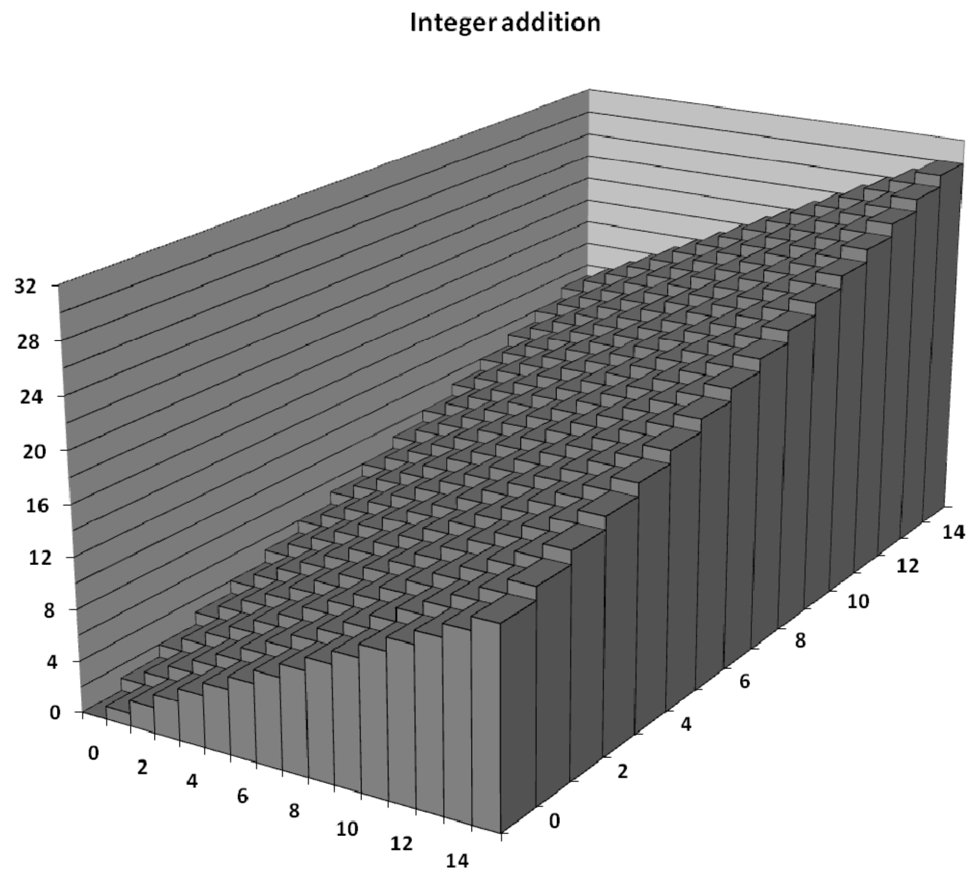
Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Addition (1)

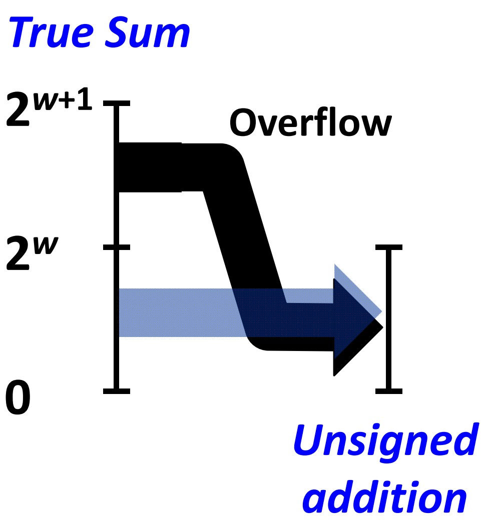
Integer addition example

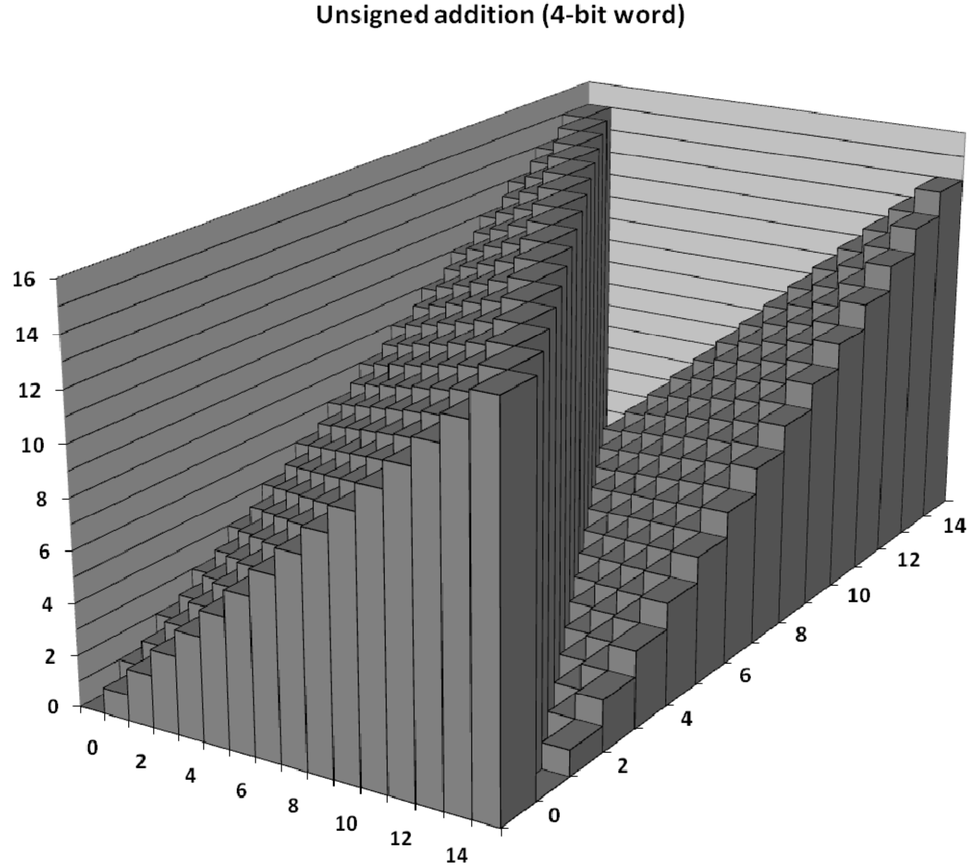
- 4-bit integers ***u***, ***v***
- Compute true sum
- True sum requires one more bit ("carry")
- Values increase linearly with ***u*** and ***v***
- Forms planar surface



Addition (2)

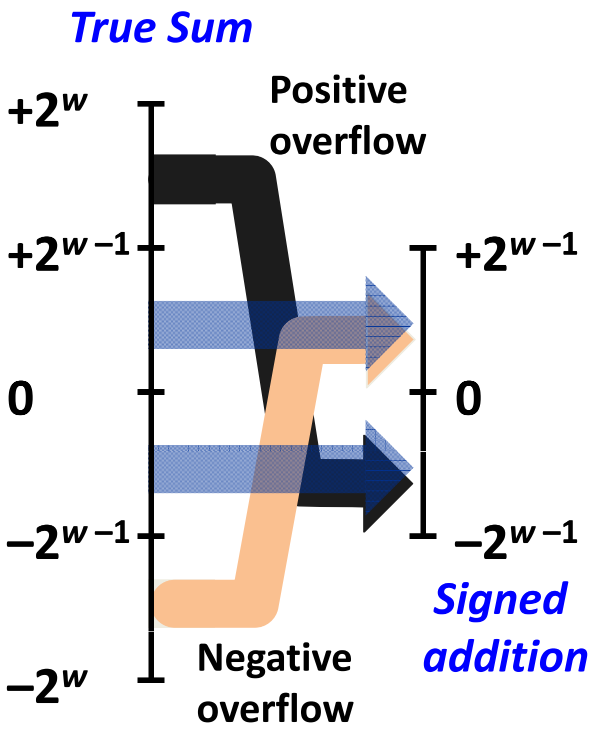
- **Unsigned addition**
  - Ignores carry output
  - Wraps around
    - If true sum ≥ 2^W
    - At most once

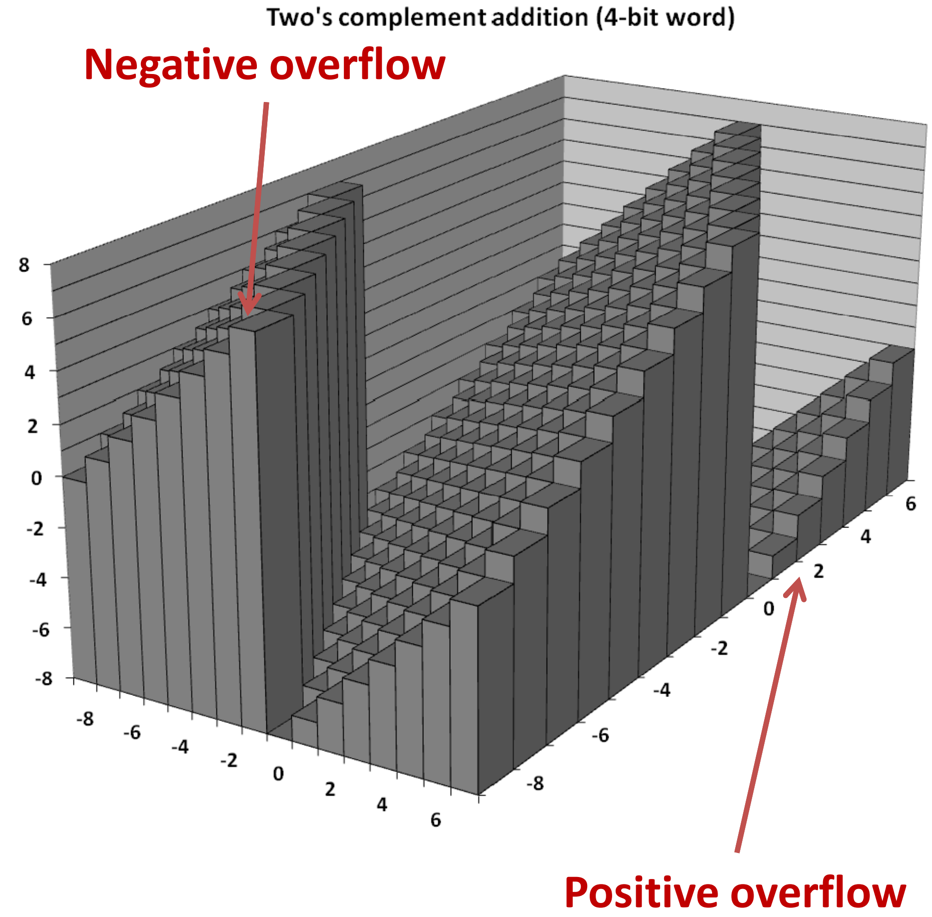




Addition (3)

- **Signed addition**
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer





Addition (4)

- **Signed addition in C**
  - Ignores carry output
  - The low-order **w** bits are identical to unsigned addition

Examples for w = 3

Mode	x	y	x + y	Truncated x + y
Unsigned	4 [100]	3 [011]	7 [0111]	7 [111]
Two's comp.	-4 [100]	3 [011]	-1 [1111]	-1 [111]
Unsigned	4 [100]	7 [111]	11 [1011]	3 [011]
Two's comp.	-4 [100]	-1 [111]	-5 [1011]	3 [011]
Unsigned	3 [011]	3 [011]	6 [0110]	6 [110]
Two's comp.	3 [011]	3 [011]	6 [0110]	-2 [110]

Multiplication (1)

- **Ranges of (x * y)**
  - Unsigned: up to **2w** bits

    $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$

  - Two's complement min: up to **2w-1** bits

    $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$

  - Two's complement max: up to **2w** bits (only for TMin^2)

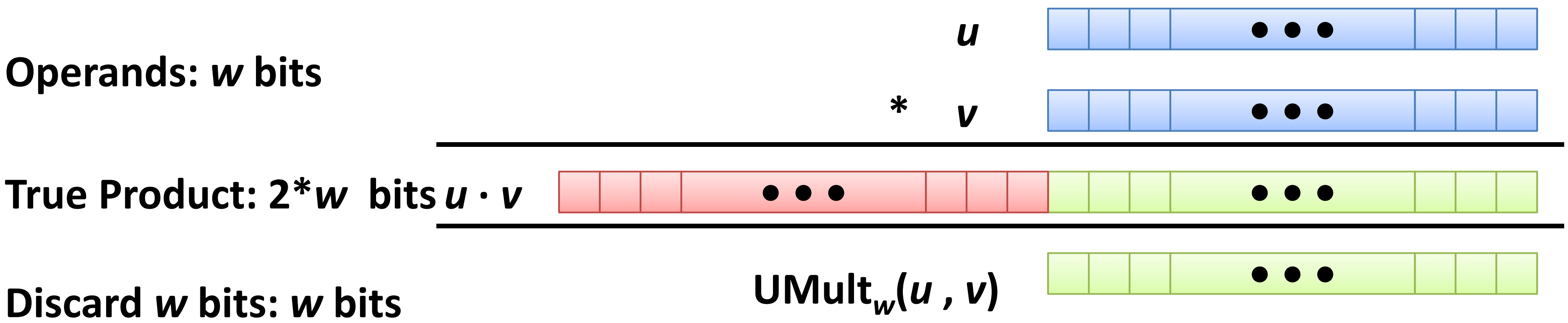
    $x * y \le (-2^{w-1})^2 = 2^{2w-2}$

- **Maintaining exact results**
  - Would need to keep expanding word size with each product computed
  - Done in software by "arbitrary precision" arithmetic packages

Multiplication (2)

- **Unsigned multiplication in C**
  - Ignores high order **w** bits
  - Implements modular arithmetic

$$UMult_w(u,v) = u \cdot v \mod 2^w$$



Multiplication (3)

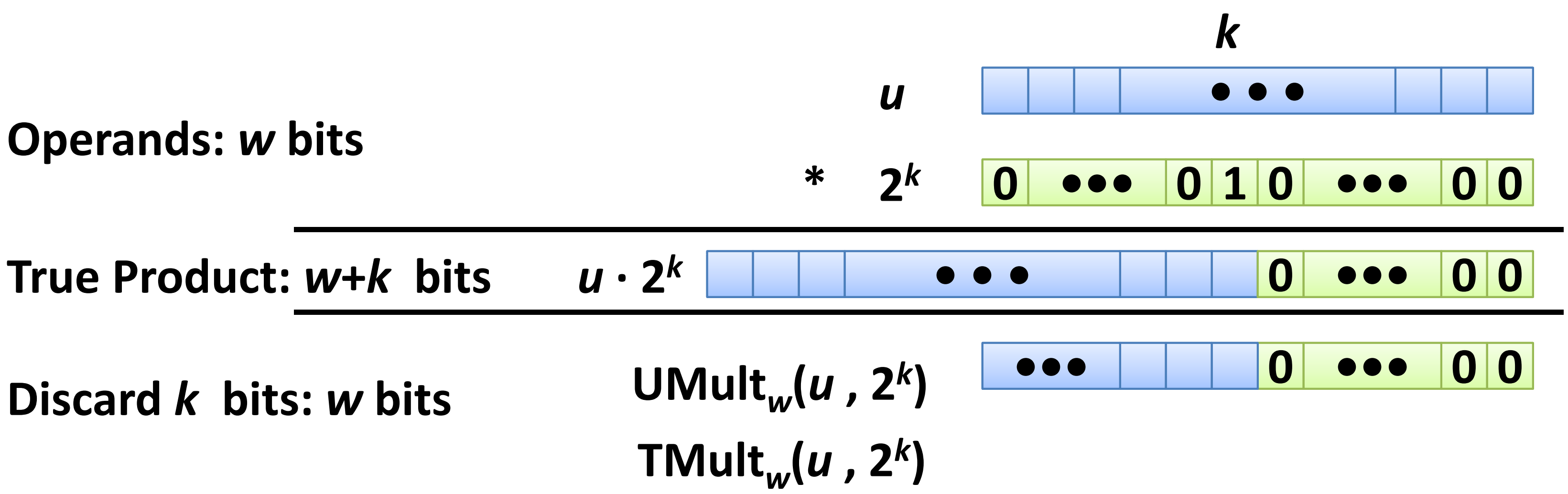
- **Signed multiplication in C**
  - Ignores high order **w** bits
  - The low-order **w** bits are identical to unsigned multiplication

Examples for w = 3

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

Multiplication (4)

- **Power-of-2 multiply with shift**
  - ***u << k*** gives ***u * 2^k***
    - e.g., ***u << 3 == u * 8***
  - Both signed and unsigned
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically



Multiplication (5)

- **Compiled multiplication code**
  - C compiler automatically generates shift/add code when multiplying by constant

C Function

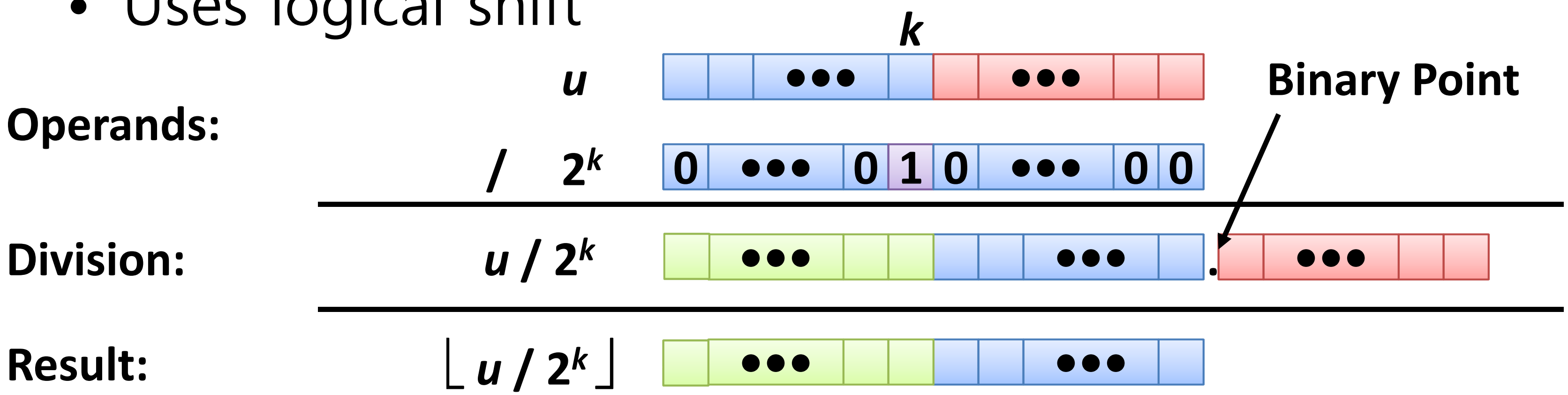
```
int  mul12 (int  x)
{
     return x * 12;
}
```

Compiled Arithmetic Operations

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

Division (1)

- **Unsigned power-of-2 divide with shift**
  - ***u* >> *k*** gives ⌊ ***u* / 2*^k^*** ⌋
  - 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

Division (2)

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

C Function

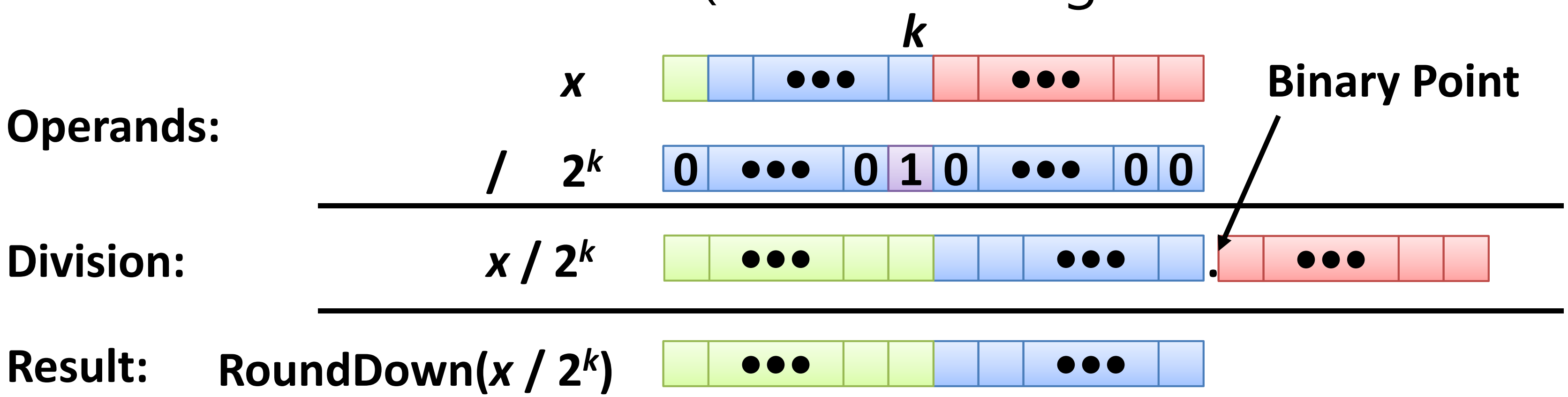
```
unsigned  udiv8 (unsigned  x)
{
    return x / 8;
}
```

Compiled Arithmetic Operations

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

Division (3)

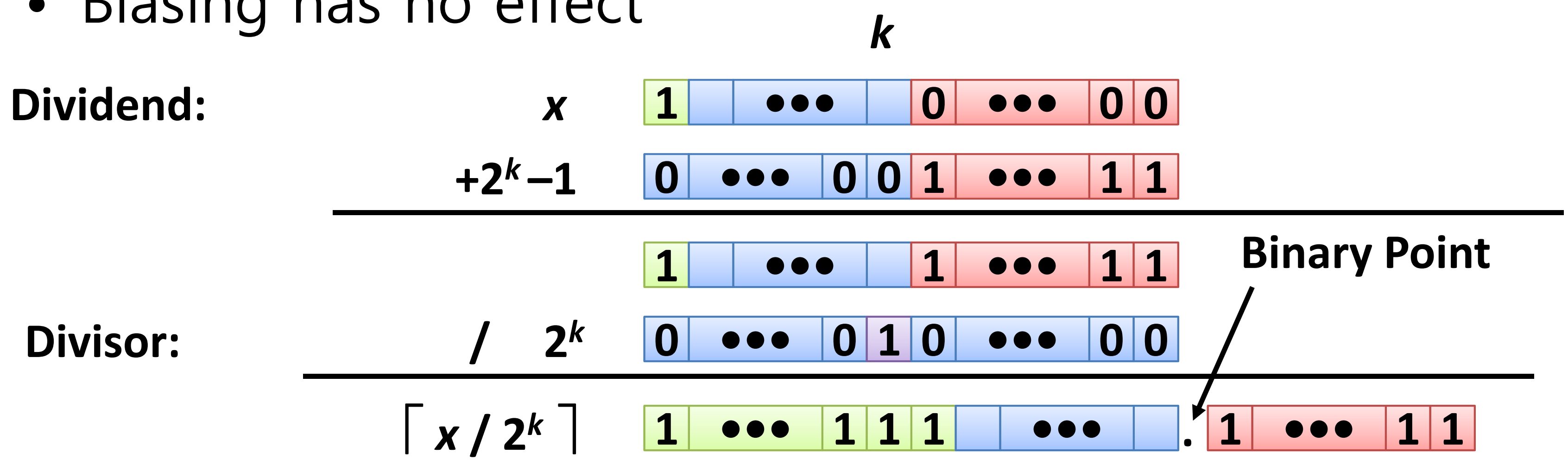
- **Signed power-of-2 divide with shift**
  - ***x* >> *k*** gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift (rounds wrong direction if ***x*** < 0)



Expression	Division	Result	Hex	Binary
y	-15213	15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

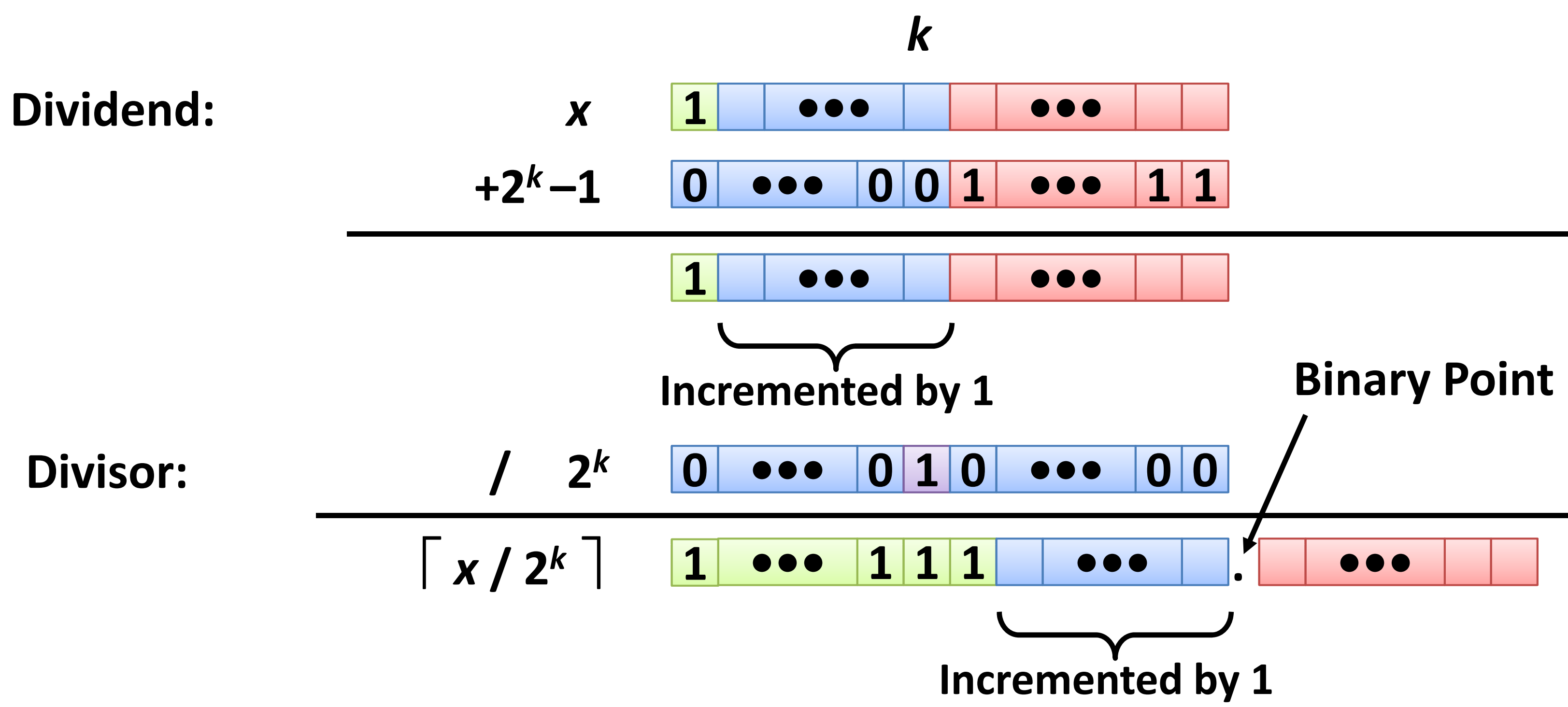
Division (4)

- **Correct power-of-2 divide**
  - Want ⌈ $x / 2^k$ ⌉ (Round Toward 0) when $x < 0$
  - Compute as ⌈ $(x + 2^k - 1) / 2^k$ ⌉
    - In C: **(x + (1 << k) – 1) >> k**
    - Biases dividend toward 0
- **Case 1: No rounding**
  - Biasing has no effect



Division (5)

- **Case 2: Rounding**
  - Biasing adds 1 to final result



Division (6)

- **Compiled signed division code**
  - Uses arithmetic shift for signed
  - Arithmetic shift written as **>>** in Java

C Function

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

Compiled Arithmetic Operations

```
        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;
```