

REPRESENTING AND MANIPULATING INTEGERS

Jo, Heeseung

Examples

```
int i = 1;  
short j = 2;  
i=(int) j;  
j=(short) i;
```

```
int i = 70000;  
short j = 2;  
j=(short) i;
```

```
int i = -5;  
unsigned k = 6;  
i=(int) k;  
k=(unsigned) i;
```

```
int i = -5;  
unsigned k = -6;  
i=(int) k;  
k=(unsigned) i;
```

Examples

1. `char -> short`
2. `short -> int`
3. `int -> long`
4. `long -> int`
5. `int -> short`
6. `short -> char`
7. `unsigned -> signed`
8. `signed -> unsigned`

Representing Information

Information = Bits + Context

- Computers manipulate representations of things
- Things are represented as binary digits
- What can you represent with N bits?
 - 2^N things
 - Numbers, characters, pixels, positions, source code, executable files, machine instructions, ...
 - Depends on what operations you do on them

	01110011	01100101	01101101	01101001	01110011	01100101	01101101	01101001
(char)	's'	'e'	'm'	'i'	's'	'e'	'm'	'i'
(int)	1768777075				1768777075			
(double)	7.03168990329170808178... x 10 ¹⁹⁹							

Unsigned Integers

Encoding unsigned integers

$$B = [b_{w-1}, b_{w-2}, \dots, b_0] \quad x = 0000 \ 0111 \ 1101 \ 0011_2$$

$$D(B) = \sum_{i=0}^{w-1} b_i \cdot 2^i$$

$$\begin{aligned} D(x) &= 2^{10} + 2^9 + 2^8 + 2^7 \\ &\quad + 2^6 + 2^4 + 2^1 + 2^0 \\ &= 1024 + 512 + 256 + 128 \\ &\quad + 64 + 16 + 2 + 1 \\ &= 2003 \end{aligned}$$

What is the range for unsigned values with w bits?

Signed Integers (1)

Encoding positive numbers

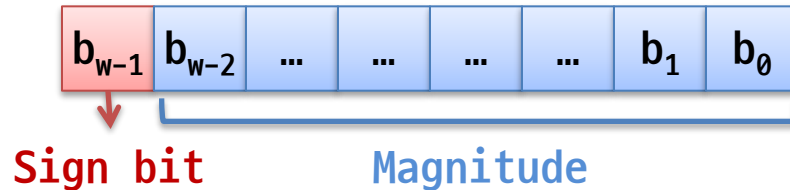
- Same as unsigned numbers

Encoding negative numbers

- Sign-magnitude representation
- Ones' complement representation
- Two's complement representation

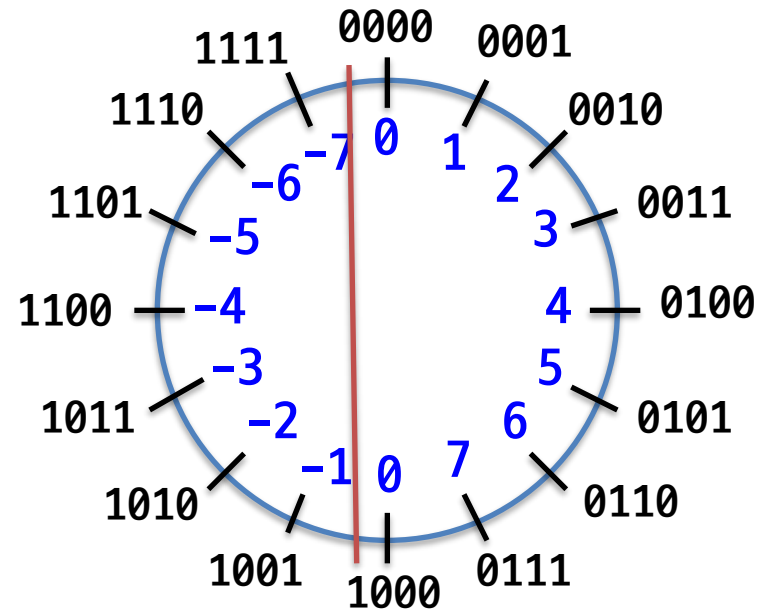
Signed Integers (2)

Sign-magnitude representation



$$S(B) = (-1)^{b_{w-1}} \cdot \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$

- Two zeros
 - $[000\dots00]$, $[100\dots00]$
- Used for floating-point numbers



Signed Integers (3)

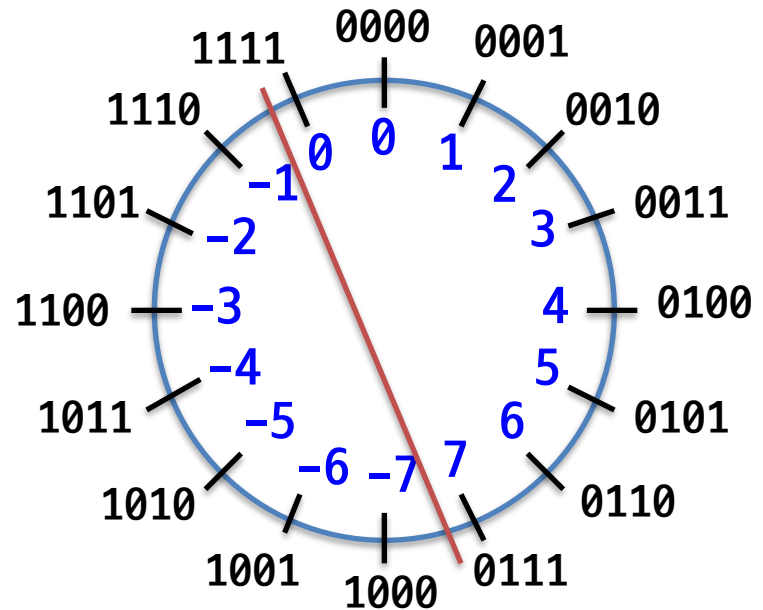
Ones' complement representation



Sign bit

$$O(B) = -b_{w-1}(2^{w-1} - 1) + \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$

- Easy to find $-n$
- Two zeros
 - $[000\dots 00]$, $[111\dots 11]$
- No longer used



Signed Integers (4)

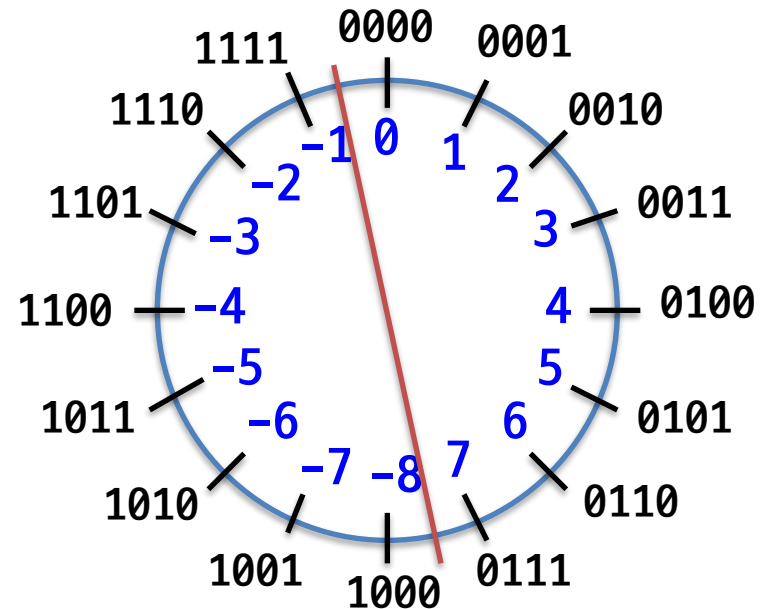
Two's complement representation



Sign bit

$$O(B) = -b_{w-1} \cdot 2^{w-1} + \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$

- Unique zero
- Easy for hardware
 - leading 0 ≥ 0
 - leading 1 < 0
- Used by almost all modern machines



Representation of Negative Numbers

$+N$	Positive Integers (all systems)	$-N$	Negative Integers		
			Sign and Magnitude	2's Complement N^*	1's Complement $\bar{N}$
+0	0000	-0	1000	—	1111
+1	0001	-1	1001	1111	1110
+2	0010	-2	1010	1110	1101
+3	0011	-3	1011	1101	1100
+4	0100	-4	1100	1100	1011
+5	0101	-5	1101	1011	1010
+6	0110	-6	1110	1010	1001
+7	0111	-7	1111	1001	1000
		-8	—	1000	—

Signed Integers (5)

Two's complement representation (cont'd)

- Following holds for two's complement

$$\sim x + 1 == -x$$

- Complement
 - Observation: $\sim x + x == 1111\dots 11_2 == -1$
- Increment
 - $\sim x + x == -1$
 - $\sim x + x + (-x + 1) == -1 + (-x + 1)$
 - $\sim x + 1 == -x$

Numeric Ranges (1)

Unsigned values

- $UMin = 0$ [000...00]
- $UMax = 2^w - 1$ [111...11]

Two's complement values

- $TMin = -2^{w-1}$ [100...00]
- $TMax = 2^{w-1} - 1$ [011...11]

Values for $w = 16$

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Numeric Ranges (2)

Values for different word sizes

	w = 8	w = 16	w = 32	w = 64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

Observations

- $|TMin| = TMax + 1$
(Asymmetric range)
- $UMax = 2 * TMax + 1$

In C programming

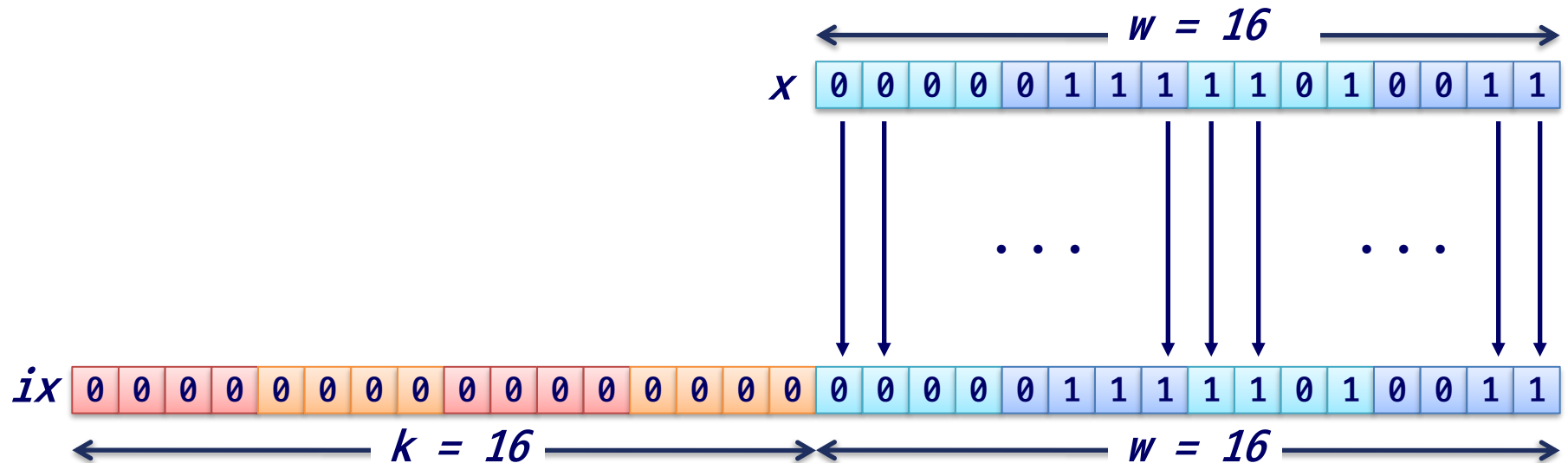
- `#include <limits.h>`
- `INT_MIN`, `INT_MAX`,
`LONG_MIN`, `LONG_MAX`,
`UINT_MAX`, ...
- Values platform-specific

Type Conversion (1)

Unsigned: w bits $\rightarrow w+k$ bits

- Zero extension: just fill k bits with 0's

```
unsigned short x = 2003;  
unsigned      ix = (unsigned) x;
```



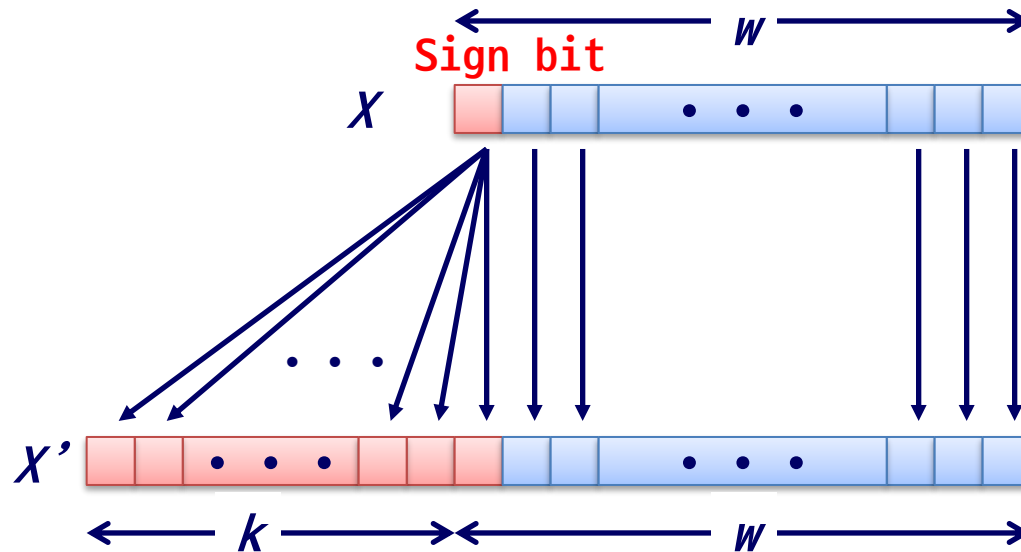
Type Conversion (2)

Signed: w bits $\rightarrow w+k$ bits

- Given w -bit signed integer x
- Convert it to $w+k$ bit integer with same value

Sign extension

- Make k copies of sign bit



Type Conversion (3)

Sign extension example

- Converting from smaller to larger integer type
- C automatically performs sign extension

```
short  int  x    = 2003;
        int  ix   = (int) x;
short  int  y    = -2003;
        int  iy   = (int) y;
```

	Decimal	Hex	Binary
x	2003	07 D3	00000111 11010011
ix	2003	00 00 07 D3	00000000 00000000 00000111 11010011
y	-2003	F8 2D	11111000 00101101
iy	-2003	FF FF F8 2D	11111111 11111111 11111000 00101101

Type Conversion (4)

Unsigned & Signed: $w+k$ bits $\rightarrow$ w bits

- Just **truncate** it to lower w bits
- Equivalent to computing $x \bmod 2^w$

```
unsigned int x = 0xcafebabe;
unsigned short ix = (unsigned short) x;
int y = 0x2003beef;
short iy = (short) y;
```

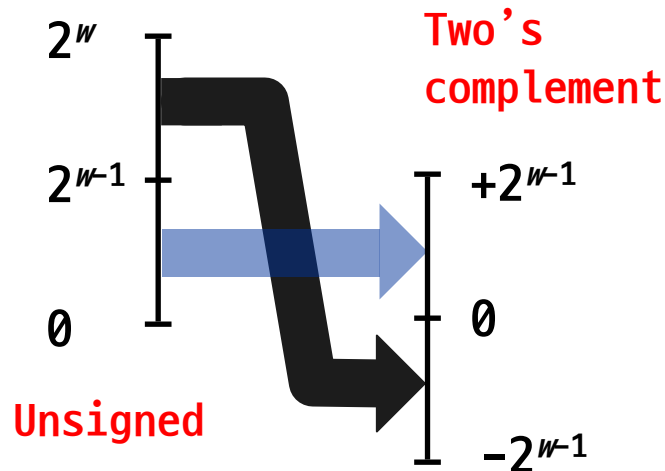
	Decimal	Hex	Binary
x	3405691582	CA FE BA BE	11001010 11111110 10111010 10111110
ix	47806	BA BE	10111010 10111110
y	537116399	20 03 BE EF	00100000 00000011 10111110 11101111
iy	-16657	BE EF	10111110 11101111

Type Conversion (5)

Unsigned -> Signed

- The same bit pattern is interpreted as a signed number

$$U2T_w(x) = \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases}$$



```
unsigned short x = 2003;
short ix = (short) x;
unsigned short y = 0xbabe;
short iy = (short) y;
```

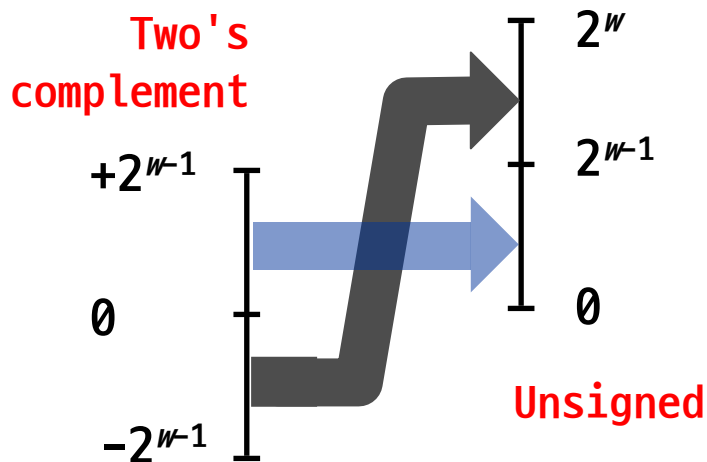
	Decimal	Hex	Binary
x	2003	07 D3	00000111 11010011
ix	2003	07 D3	00000111 11010011
y	47806	BA BE	10111010 10111110
iy	-17730	BA BE	10111010 10111110

Type Conversion (6)

Signed → Unsigned

- Ordering inversion
- Negative → Big positive

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases}$$



```
short x = 2003;
unsigned short ix = (unsigned short) x;
short y = -2003;
unsigned short iy = (unsigned short) y;
```

	Decimal	Hex	Binary
x	2003	07 D3	00000111 11010011
ix	2003	07 D3	00000111 11010011
y	-2003	F8 2D	11111000 00101101
iy	63533	F8 2D	11111000 00101101

Type Casting in C (1)

Constants

- By **default**, considered to be **signed integers**
- **Unsigned** if have "U" or "u" as suffix
 - 0U, 12345U, 0x1A2Bu

Type casting

- Explicit casting
- Implicit casting via
 - Assignments
 - Procedure calls

```
int tx, ty;  
unsigned ux, uy;  
tx = (int) ux;  
uy = (unsigned) ty;
```

```
int f(unsigned);  
f(ty);  
tx = ux;
```

Type Casting in C (2)

Expression evaluation

- If mix unsigned and signed in single expression, signed values **implicitly cast to unsigned**
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`

Expression	Type	Evaluation
<code>0 == 0U</code>		
<code>-1 < 0</code>		
<code>-1 < 0U</code>		
<code>-1 > -2</code>		
<code>(unsigned) -1 > -2</code>		
<code>2147483647 > -2147483647-1</code>		
<code>2147483647U > -2147483647-1</code>		
<code>2147483647 > (int) 2147483648U</code>		

Type Casting in C (3)

Example 1-1

```
int main ()
{
    unsigned i;
    for (i = 10; i >= 0; i--)
        printf ("%u\n", i);
}
```

Example 1-2

```
int main ()
{
    unsigned i;
    for (i = 10; i > 0; i--)
        printf ("%u\n", i);
}
```

Type Casting in C (4)

Example 2

```
int sum_array (int a[], unsigned len)
{
    int i;
    int result = 0;

    for (i = 0; i <= len - 1; i++)
        result += a[i];

    return result;
}
```

Type Casting in C (5)

Example 3-1

```
void copy_mem1 (char *src, char *dest, unsigned len)
{
    unsigned i;
    for (i = 0; i < len; i++)
        *dest++ = *src++;
}
```

Example 3-2

```
void copy_mem2 (char *src, char *dest, unsigned len)
{
    int i;
    for (i = 0; i < len; i++)
        *dest++ = *src++;
}
```


Type Casting in C (6)

Example 3-3

```
void copy_mem3 (char *src, char *dest, unsigned len)
{
    for (; len > 0; len--)
        *dest++ = *src++;
}
```

Example 3-4

```
void copy_mem4 (char *src, char *dest, unsigned len)
{
    for (; (int) len > 0; len--)
        *dest++ = *src++;
}
```

Type Casting in C (7)

Example 4

```
#include <stdio.h>

int main ()
{
    unsigned char c;

    while ((c = getchar()) != EOF)
        putchar (c);
}
```

Type Casting in C (8)

Lessons

- There are many tricky situations when you use unsigned integers - hard to debug
- Do not use just because numbers are nonnegative
- Use "unsigned" for large positive numbers within limited bits
- Use "unsigned" when you need collections of bits with no numeric interpretation ("flags")
- Few languages other than C support unsigned integers

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` $\rightarrow$ `0xBE`
- `~010000012` $\rightarrow$ `101111102`
- `~0x00` $\rightarrow$ `0xFF`
- `~000000002` $\rightarrow$ `111111112`
- `0x69 & 0x55` $\rightarrow$ `0x41`
- `011010012 & 010101012` $\rightarrow$ `010000012`
- `0x69 | 0x55` $\rightarrow$ `0x7D`
- `011010012 | 010101012` $\rightarrow$ `011111012`
- `0x69 ^ 0x55` $\rightarrow$ `0x3C`
- `011010012 ^ 010101012` $\rightarrow$ `001111002`

Logic Operations in C

Logical operators

- `&&`, `||`, `!`
- View 0 as "False"
- Anything nonzero as "True"
- Always return 0 or 1

Early termination

```
int i=1;
int a=4, b=4, c=5;

if ( a == b && i > c )
{
    do_something();
}
```

```
int i=1;
int a=4, b=4, c=5;

if ( i > c && a == b )
{
    do_something();
}
```

Logic Operations in C

Examples (char data type)

- `!0x41 --> 0x00`
- `!0x00 --> 0x01`
- `!!0x41 --> 0x01`

- `0x69 && 0x55 -->`
- `0x69 || 0x55 -->`

```
if (p && *p) // avoids null pointer access
{
    // p is not null pointer && *p is not 0
}
```

Shift Operations

Left shift: $x \ll y$

- Shift bit-vector x left y positions
 - Throw away extra bits on left
 - Fill with 0's on right

Right shift: $x \gg y$

- 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 $y \geq \text{word size}$

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Shift Operations

Right shift example

```
void main()
{
    char i = -2;

    printf("%d\n", i);
    i = i >> 2;
    printf("%d\n", i);
}
```

"Arithmetic shift right" is the default operation in C

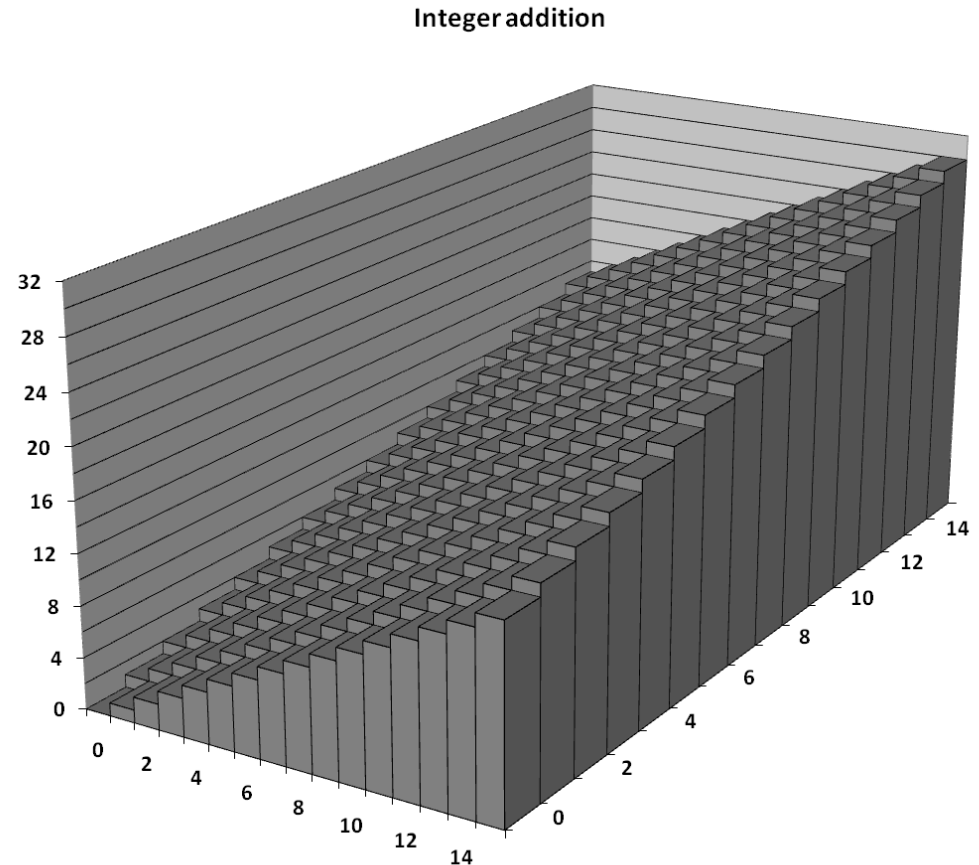
```
void main()
{
    char i = -2;

    printf("%d\n", i);
    i = i >> -2;
    printf("%d\n", i);
}
```


Addition (1)

Integer addition example

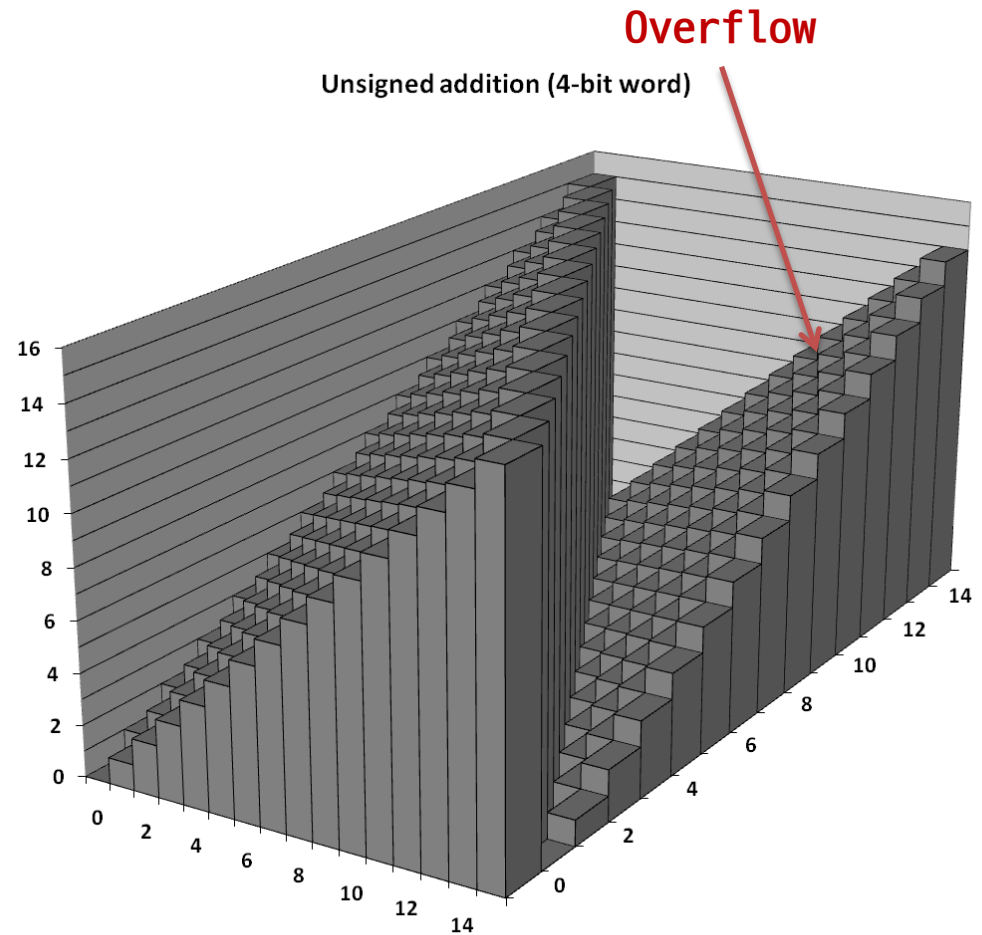
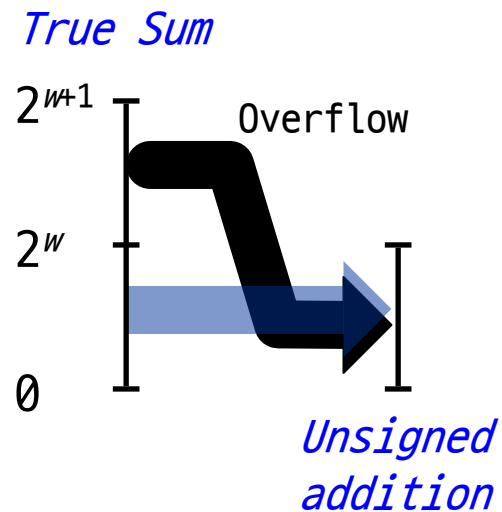
- 4-bit integers u , v
- Compute true sum
- True sum requires one more bit ("carry")
- Values increase linearly with u and v
- Forms planar surface



Addition (2)

Unsigned addition

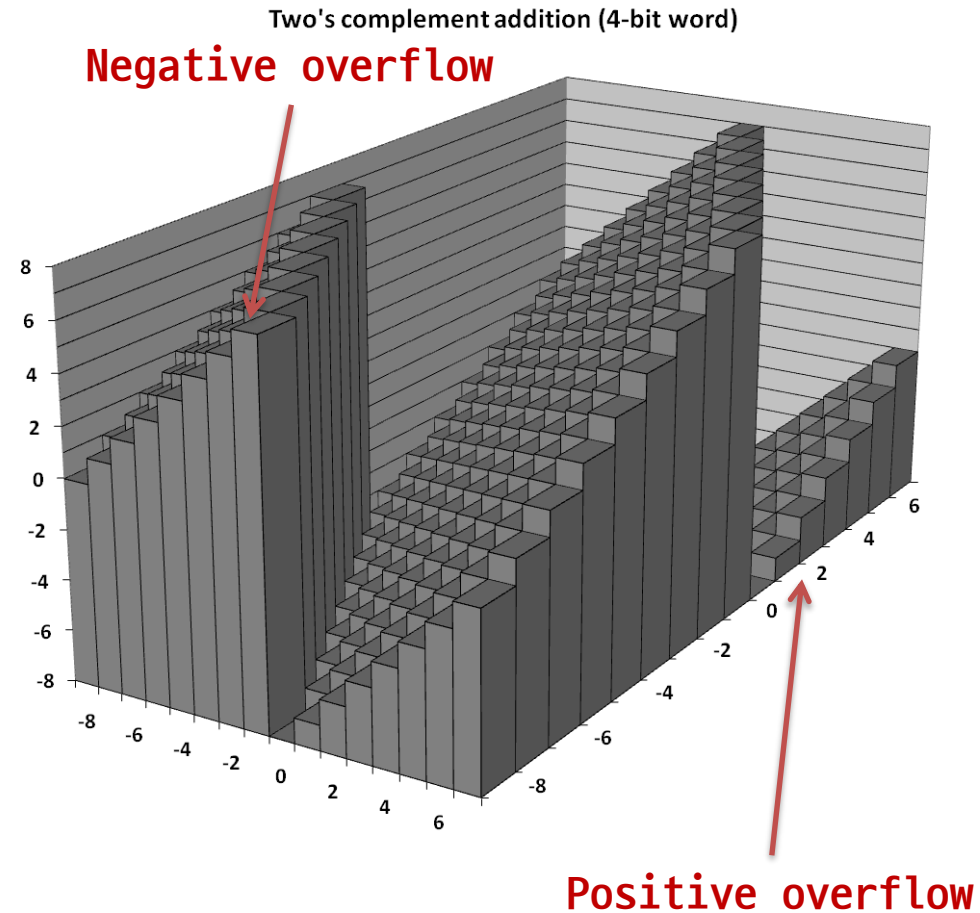
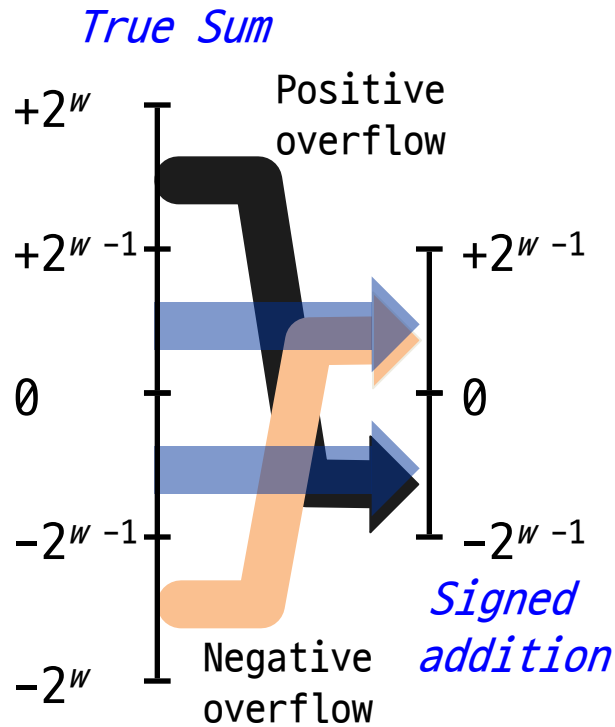
- Ignores carry output
- Wraps around
 - If true sum $\geq 2^w$



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

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

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

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

- Two's complement max: up to $2w$ bits (only for TMin²)

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

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

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

Operands: w bits



True Product:
 $2 \cdot w$ bits



Discard w bits: w
bits



Multiplication (3)

Signed multiplication in C

- Ignores high order w bits
- The low-order w bits are identical to unsigned multiplication

Examples for $w = 3$

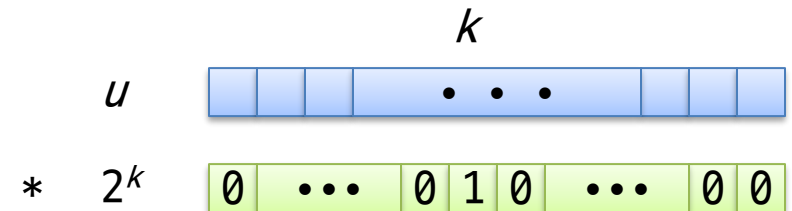
Mode	x	y	$x \cdot y$	Truncated $x \cdot y$
Unsigned	5 [101]	3 [011]	15 [001111]	7 [111]
Two's comp.	-3 [101]	3 [011]	-9 [110111]	-1 [111]
Unsigned	4 [100]	7 [111]	28 [011100]	4 [100]
Two's comp.	-4 [100]	-1 [111]	4 [000100]	-4 [100]
Unsigned	3 [011]	3 [011]	9 [001001]	1 [001]
Two's comp.	3 [011]	3 [011]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

Multiplication (4)

Power-of-2 multiply with shift

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

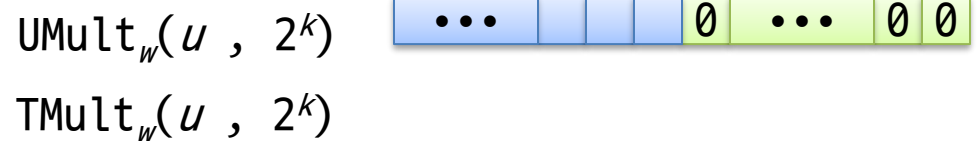
Operands: w bits



True Product: $w+k$ bits



Discard k bits: w bits



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

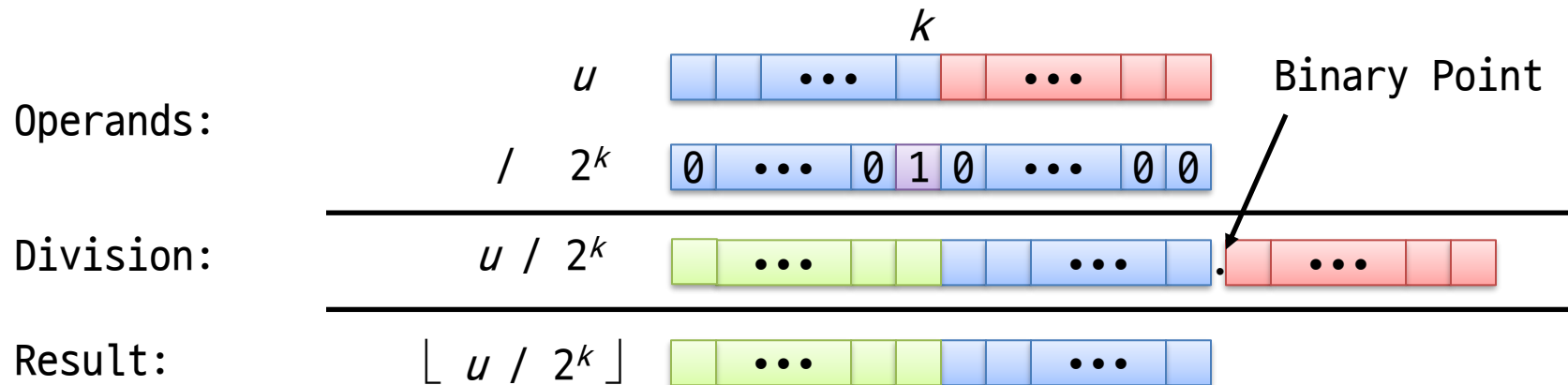
```
leal    (%eax, %eax, 2), %eax      ; t ← x + x * 2
sall    $2, %eax                  ; return t << 2
```

```
imull   $12, %eax                 ; x ← x * 12
```

Division (1)

Unsigned power-of-2 divide with shift (including signed positive)

- $u \gg k$ gives $\lfloor u / 2^k \rfloor$
- Uses logical shift



Expression	Division	Result	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	0011101 10110110
x >> 4	950.8125	950	03 B6	0000011 10110110
x >> 8	59.4257813	59	00 3B	0000000 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;
}
```

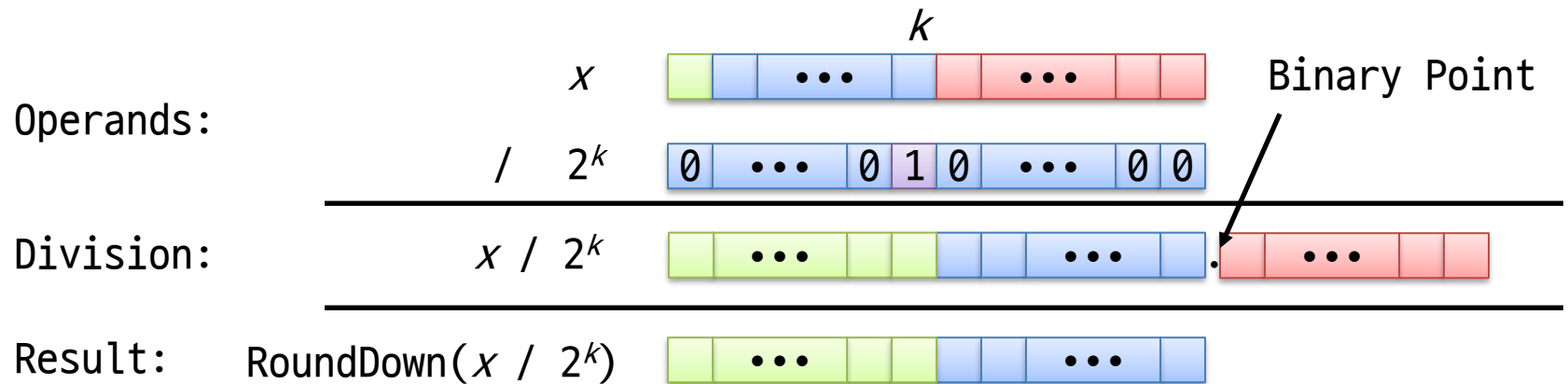
Compiled Arithmetic Operations

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

Division (3)

Signed negative power-of-2 divide with shift

- $x \gg k$ 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 $\lceil x / 2^k \rceil$ (Round Toward 0) when $x < 0$
- Compute as $\lceil (x + 2^k - 1) / 2^k \rceil$
 - In C: $(x + (1 \ll k) - 1) \gg k$
 - Biases dividend toward 0

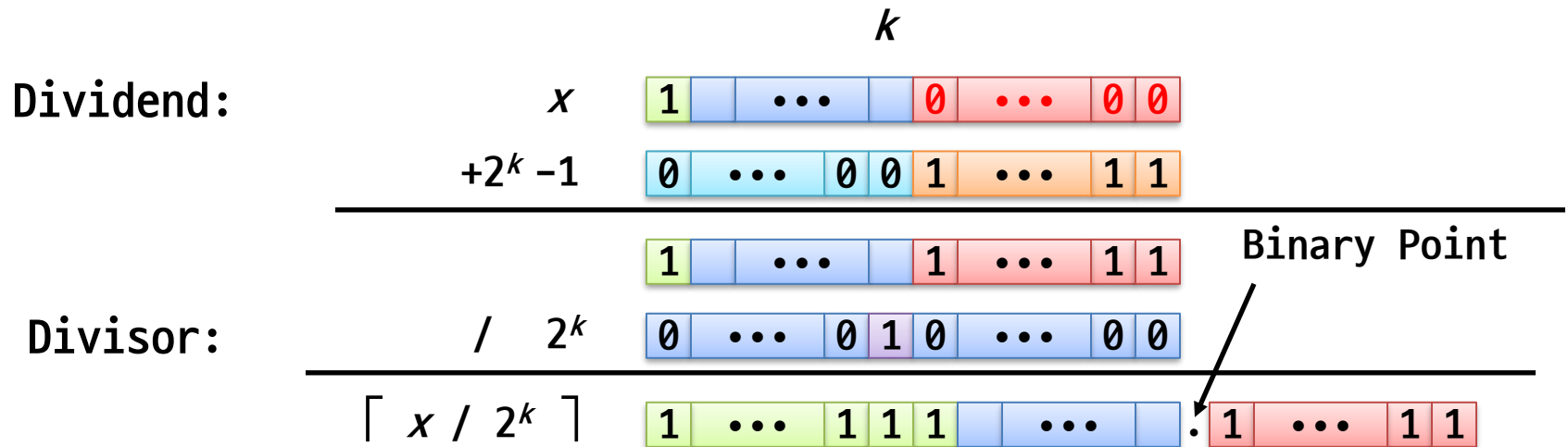
Division (4)

Correct power-of-2 divide

- Want $\lceil x / 2^k \rceil$ (Round Toward 0) when $x < 0$
- Compute as $\lceil (x + 2^k - 1) / 2^k \rceil$
 - In C: $(x + (1 \ll k) - 1) \gg k$
 - Biases dividend toward 0

Case 1: No rounding

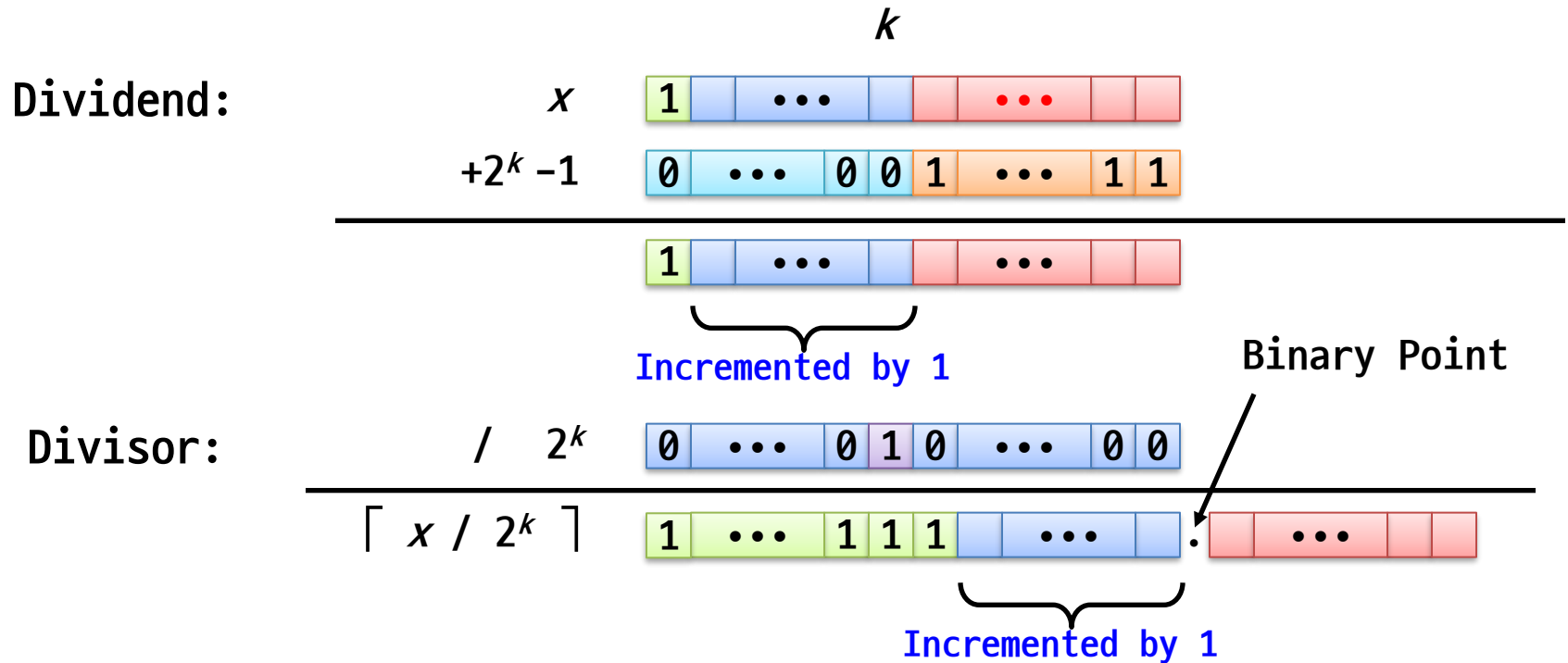
- Biasing has no effect



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

- Compute as $\lceil (x + 2^k - 1) / 2^k \rceil$
 - In C: $(x + (1 \ll k) - 1) \gg k$
 - Biases dividend toward 0

Compiled Arithmetic Operations

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

C Function

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

Explanation

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


Division (7)

Division example

```
void main()
{
    char i = -17;

    printf("%d\n", i);

    i = i / 8;
    printf("%d\n", i);

    i = i >> 3;
    printf("%d\n", i);
}
```