

CS 33

Introduction to C Part 7

Implicit Conversions (1)

```
float x, y=2.0;
```

```
int i=1, j=2;
```

```
x = i/j + y;
```

```
/* what's the value of x? */
```

Implicit Conversions (2)

```
float x, y=2.0;
```

```
int i=1, j=2;
```

```
float a, b;
```

```
a = i;
```

```
b = j;
```

```
x = a/b + y;
```

```
/* now what's the value of x? */
```

Explicit Conversions: Casts

```
float x, y=2.0;
```

```
int i=1, j=2;
```

```
x = (float)i / (float)j + y;
```

```
/* and now what's the value of x? */
```

Purposes of Casts

- **Coercion**

```
int i, j;  
float a;  
a = (float) i / (float) j;
```

modify the
value
appropriately

- **Intimidation**

```
float x, y;  
// sizeof(float) == 4  
swap((int *) &x, (int *) &y);
```

it's ok as is
(trust me!)

Quiz 1

- Will this work?

```
double x, y; //sizeof(double) == 8
```

```
...
```

```
swap ( (int *) &x, (int *) &y );
```

- a) yes
- b) no

Caveat Emptor

- **Casts tell the C compiler:**
“Shut up, I know what I’m doing!”

- **Sometimes true**

```
float x, y;  
swap( (int *) &x, (int *) &y );
```

- **Sometimes false**

```
double x, y;  
swap( (int *) &x, (int *) &y );
```

Nothing, and More ...

- ***void* means, literally, nothing:**

```
void NotMuch(void) {  
    printf("I return nothing\n");  
}
```

- **What does *void ** mean?**
 - it's a pointer to anything you feel like
 - » a generic pointer

Rules

- **Use with other pointers**

```
int *x;  
void *y;  
x = y; /* legal */  
y = x; /* legal */
```

- **Dereferencing**

```
void *z;  
func(*z); /* illegal! */  
func(*(int *)z); /* legal */
```

Swap, Revisited

```
void swap(int *i, int *j) {  
    int tmp;  
    tmp = *j; *j = *i; *i = tmp;  
}  
/* can we make this generic? */
```

An Application: Generic Swap

```
void gswap (void *p1, void *p2,  
           int size) {  
    int i;  
    for (i=0; i < size; i++) {  
        char tmp;  
        tmp = ((char *)p1)[i];  
        ((char *)p1)[i] = ((char *)p2)[i];  
        ((char *)p2)[i] = tmp;  
    }  
}
```

Using Generic Swap

```
short a=1, b=2;  
gswap(&a, &b, sizeof(short));
```

```
int x=6, y=7;  
gswap(&x, &y, sizeof(int));
```

```
int A[] = {1, 2, 3}, B[] = {7, 8, 9};  
gswap(A, B, sizeof(A));
```

Fun with Functions (1)

```
void ArrayDouble(int A[], int len) {  
    int i;  
    for (i=0; i<len; i++)  
        A[i] = 2*A[i];  
}
```

Fun with Functions (2)

```
void ArrayBop(int A[],  
             int len,  
             int (*func) (int)) {  
    int i;  
    for (i=0; i<len; i++)  
        A[i] = (*func) (A[i]);  
}
```

Fun with Functions (3)

```
int triple(int arg) {  
    return 3*arg;  
}
```

```
int main() {  
    int A[20];  
    ... /* initialize A */  
    ArrayBop(A, 20, triple);  
    return 0;  
}
```

Laziness ...

- Why type the declaration

```
void * (*f) (void *, void *);
```

- You could, instead, type

```
MyType f;
```

- (If, of course, you can somehow define *MyType* to mean the right thing)

typedef

- **Allows one to create new names for existing types**

```
typedef int *IntP_t;
```

```
IntP_t x;
```

– means the same as

```
int *x;
```

More typedefs

```
typedef struct complex {  
    float real;  
    float imag;  
} complex_t;
```

```
complex_t i, *ip;
```

And ...

```
typedef void * (MyFunc_t) (void *, void *);
```

```
MyFunc_t f;
```

```
// you must do its definition the long way
```

```
void *f(void *a1, void *a2) {  
    ...  
}
```

Not a Quiz

- What's A?

```
typedef double X_t[N];  
X_t A[M];
```

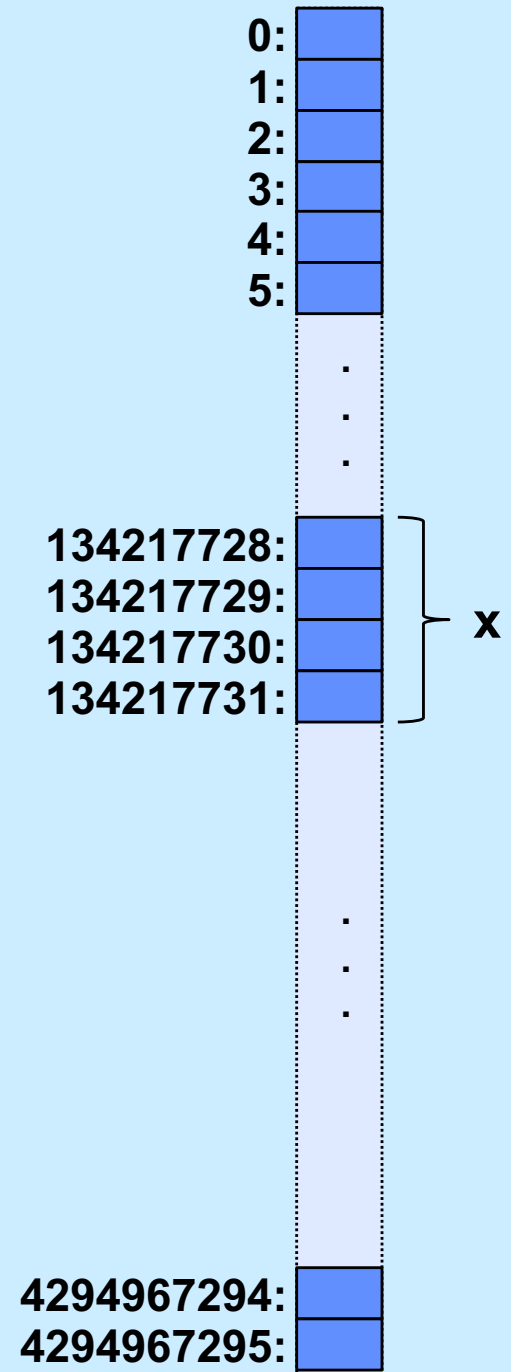
- a) an array of M doubles
- b) an MxN array of doubles
- c) an NxM array of doubles
- d) a syntax error

CS 33

Data Representation, Part 1

Representing Data in Memory

- **x** is a 4-byte integer
 - how do the 32 bits represent its value?



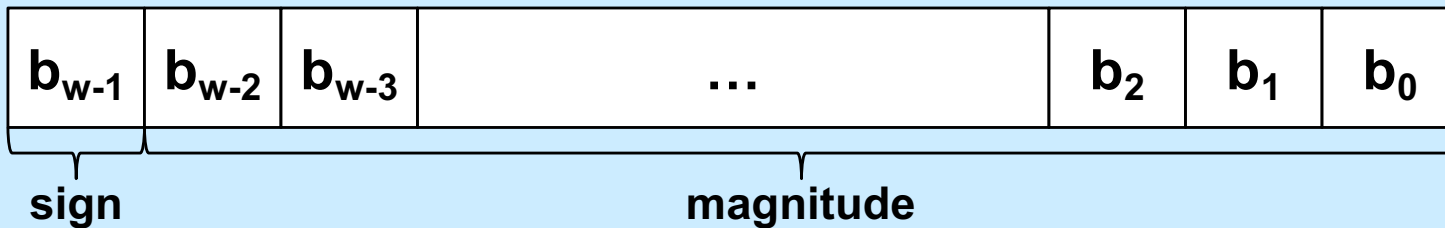
Unsigned Integers



$$\text{value} = \sum_{i=0}^{w-1} b_i \cdot 2^i$$

Signed Integers

- **Sign-magnitude**



$$\text{value} = (-1)^{b_{w-1}} \cdot \sum_{i=0}^{w-2} b_i \cdot 2^i$$

- **two representations of zero!**
 - **computer must have two sets of instructions**
 - **one for signed arithmetic, one for unsigned**

Signed Integers

- **Ones' complement**
 - negate a number by forming its bit-wise complement
 - » e.g., $(-1) \cdot 01101011 = 10010100$

$b_{w-1} = 0 \Rightarrow$ non-negative number

$$\text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$b_{w-1} = 1 \Rightarrow$ negative number

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

two zeros!

Signed Integers

- **Two's complement**

$b_{w-1} = 0 \Rightarrow$ non-negative number

$$\text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$b_{w-1} = 1 \Rightarrow$ negative number

$$\text{value} = (-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

one zero!

Example

- **w = 4**

0000: 0

0001: 1

0010: 2

0011: 3

0100: 4

0101: 5

0110: 6

0111: 7

1000: -8

1001: -7

1010: -6

1011: -5

1100: -4

1101: -3

1110: -2

1111: -1

Signed Integers

- **Negating two's complement**

$$value = -b_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

- **how to compute $-value$?**
 $(\sim value) + 1$

Signed Integers

- Negating two's complement (continued)

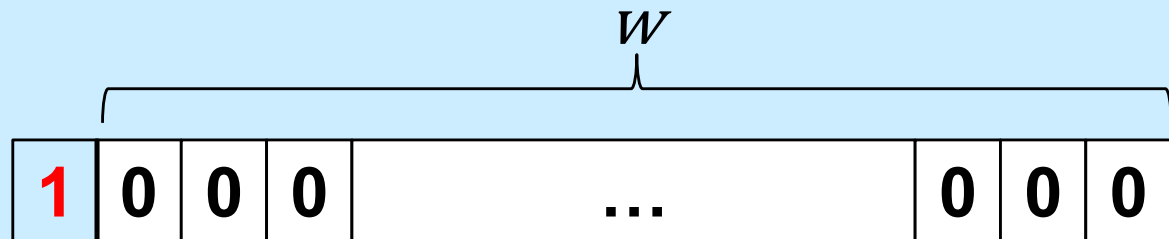
$$value + (\sim value + 1)$$

$$= (value + \sim value) + 1$$

$$= (2^w - 1) + 1$$

$$= 2^w$$

=



Quiz 2

- **We have a computer with 4-bit words that uses two's complement to represent signed integers. What is the result of subtracting 0010 (2) from 0001 (1)?**
 - a) 1110
 - b) 1001
 - c) 0111
 - d) 1111

Signed vs. Unsigned in C

- **char, short, int, and long**
 - signed integer types
 - right shift (>>) is arithmetic
- **unsigned char, unsigned short, unsigned int, unsigned long**
 - unsigned integer types
 - right shift (>>) is logical

Numeric Ranges

- **Unsigned Values**

- $UMin = 0$

- $000\dots0$

- $UMax = 2^w - 1$

- $111\dots1$

- **Two's Complement Values**

- $TMin = -2^{w-1}$

- $100\dots0$

- $TMax = 2^{w-1} - 1$

- $011\dots1$

- **Other Values**

- Minus 1

- $111\dots1$

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

Values for Different Word Sizes

	W			
	8	16	32	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

$$UMax = 2 * TMax + 1$$

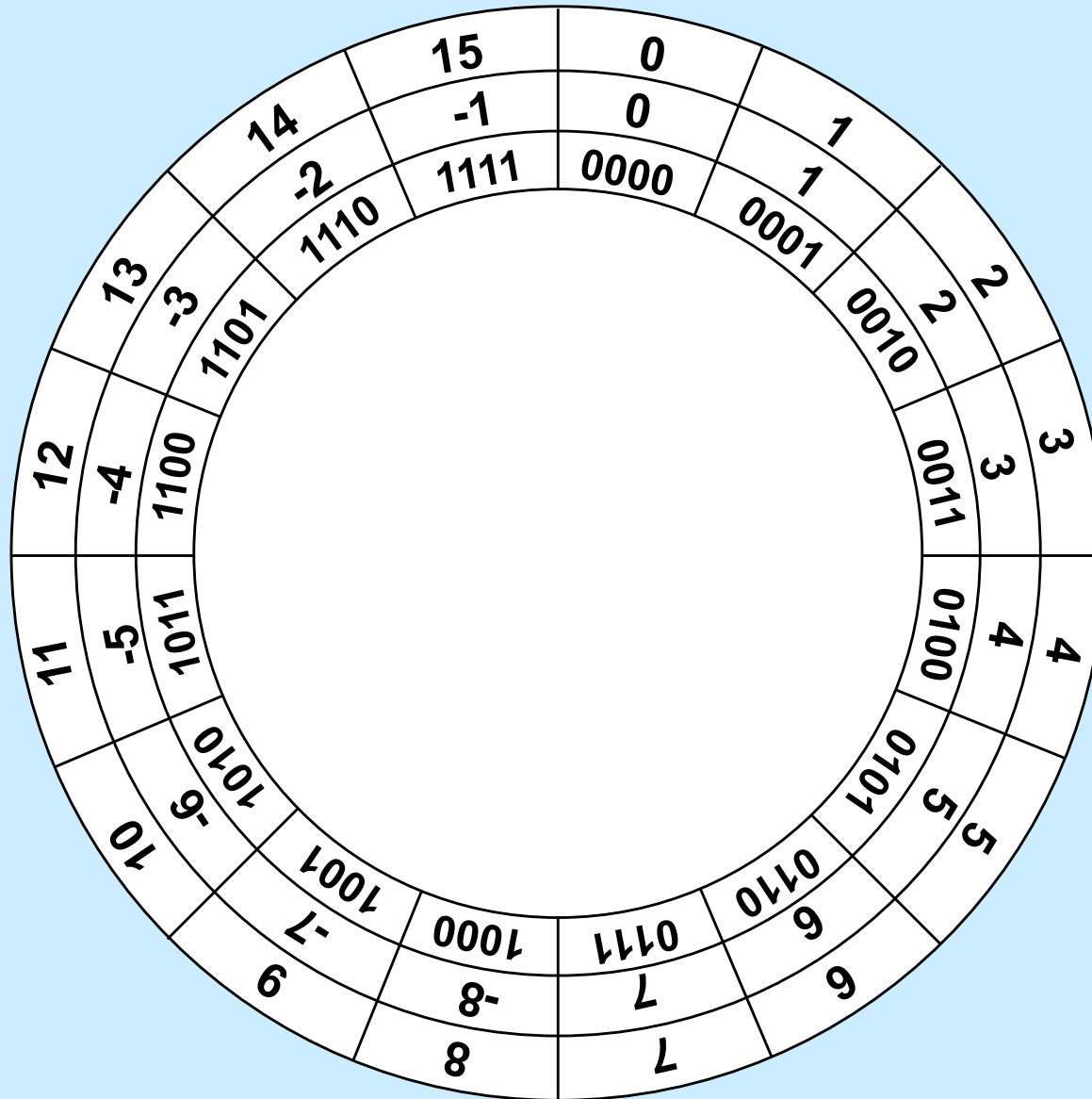
- **C Programming**

- **#include** <limits.h>
- declares constants, e.g.,
 - ULONG_MAX
 - LONG_MAX
 - LONG_MIN
- values platform-specific

Quiz 3

- What is $-TMin$ (assuming two's complement signed integers)?
 - a) $TMin$
 - b) $TMax$
 - c) 0
 - d) 1

4-Bit Computer Arithmetic



Signed vs. Unsigned in C

- **Constants**

- by default are considered to be signed integers
- unsigned if have “U” as suffix

0U, 4294967259U

- **Casting**

- **explicit casting between signed & unsigned**

```
int tx, ty;
```

```
unsigned ux, uy; // “unsigned” means “unsigned int”
```

```
tx = (int) ux;
```

```
uy = (unsigned int) ty;
```

- **implicit casting also occurs via assignments and function calls**

```
tx = ux;
```

```
uy = ty;
```

Casting Surprises

- Expression evaluation
 - if there is a mix of unsigned and signed in single expression,
signed values implicitly cast to unsigned
 - including comparison operations <, >, ==, <=, >=
 - examples for $W = 32$: **TMIN = -2,147,483,648** , **TMAX = 2,147,483,647**

Constant ₁	Constant ₂	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int)2147483648U	>	signed

Quiz 4

What is the value of

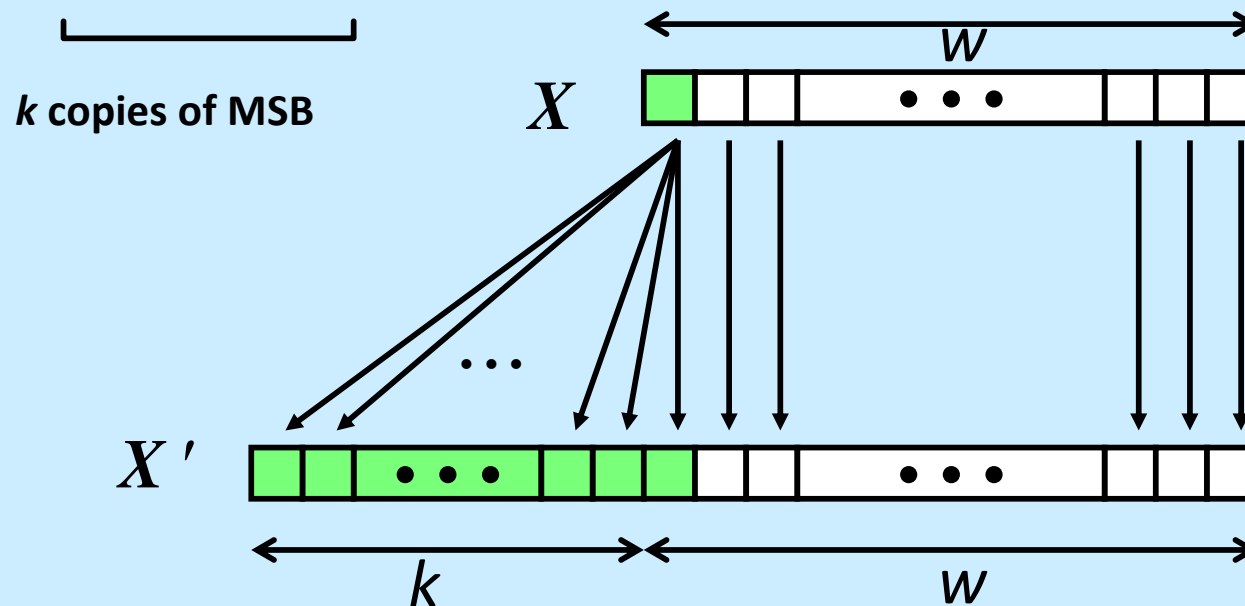
`(unsigned long) -1 - (long) ULONG_MAX`

???

- a) 0
- b) -1
- c) 1
- d) ULONG_MAX

Sign Extension

- **Task:**
 - given w -bit signed integer x
 - convert it to $w+k$ -bit integer with same value
- **Rule:**
 - make k copies of sign bit:
 - $X' = X_{w-1}, \dots, X_{w-1}, X_{w-1}, X_{w-2}, \dots, X_0$



Sign Extension Example

```
short int x = 15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

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

Does it Work?

$$val_w = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$\begin{aligned} val_{w+1} &= -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \end{aligned}$$

$$\begin{aligned} val_{w+2} &= -2^{w+1} + 2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \end{aligned}$$

Unsigned Multiplication

Operands: w bits

u 

$*$ v 

True Product: $2*w$ bits

$u * v$ 

Discard w bits: w bits

$\text{UMult}_w(u, v)$ 

- **Standard multiplication function**
 - ignores high order w bits
- **Implements modular arithmetic**

$$\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$$

Signed Multiplication

Operands: w bits

u

$*$ v

True Product: $2*w$ bits

$u * v$

Discard w bits: w bits

$\text{TMult}_w(u, v)$

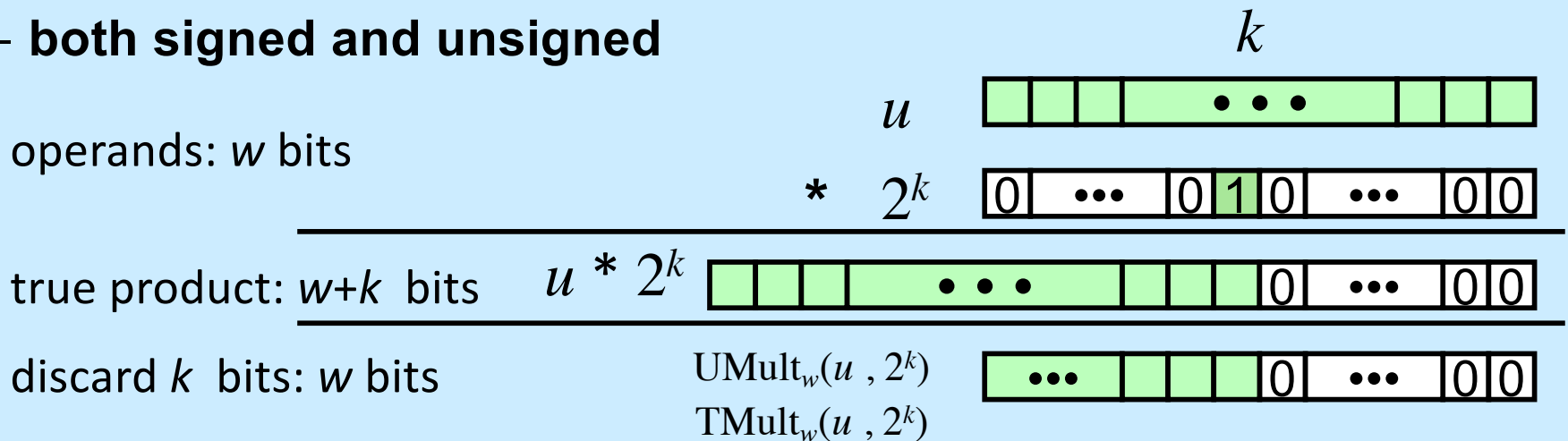
- **Standard multiplication function**
 - ignores high order w bits
 - some of which are different from those of unsigned multiplication
 - lower bits are the same

Power-of-2 Multiply with Shift

- **Operation**

- $u \ll k$ gives $u * 2^k$
- both signed and unsigned

operands: w bits



- **Examples**

$$u \ll 3 == u * 8$$

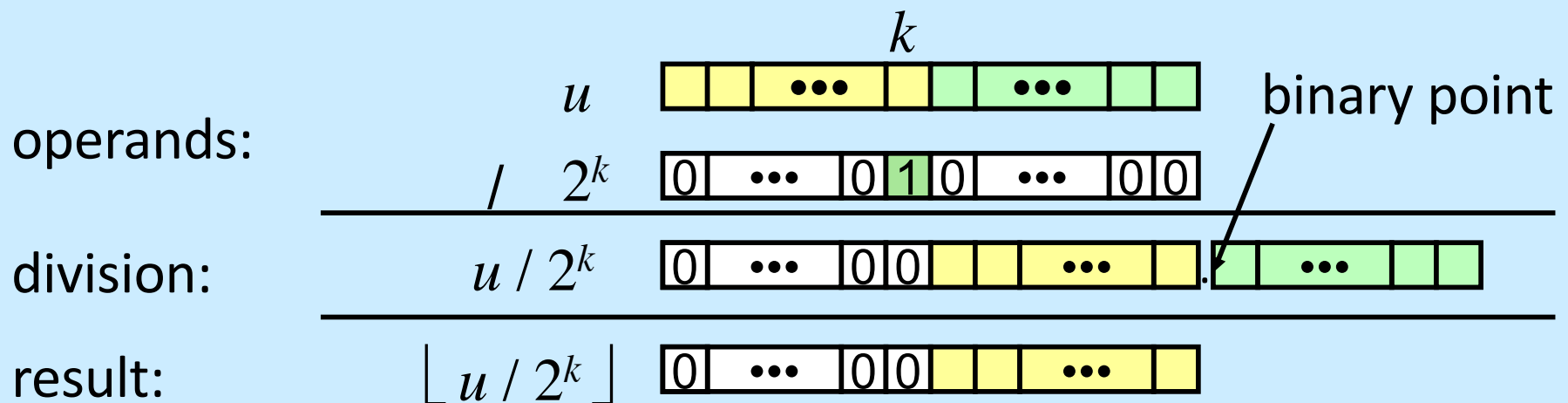
$$u \ll 5 - u \ll 3 == u * 24$$

- most machines shift and add faster than multiply
 - » compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

- Quotient of unsigned and power of 2

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

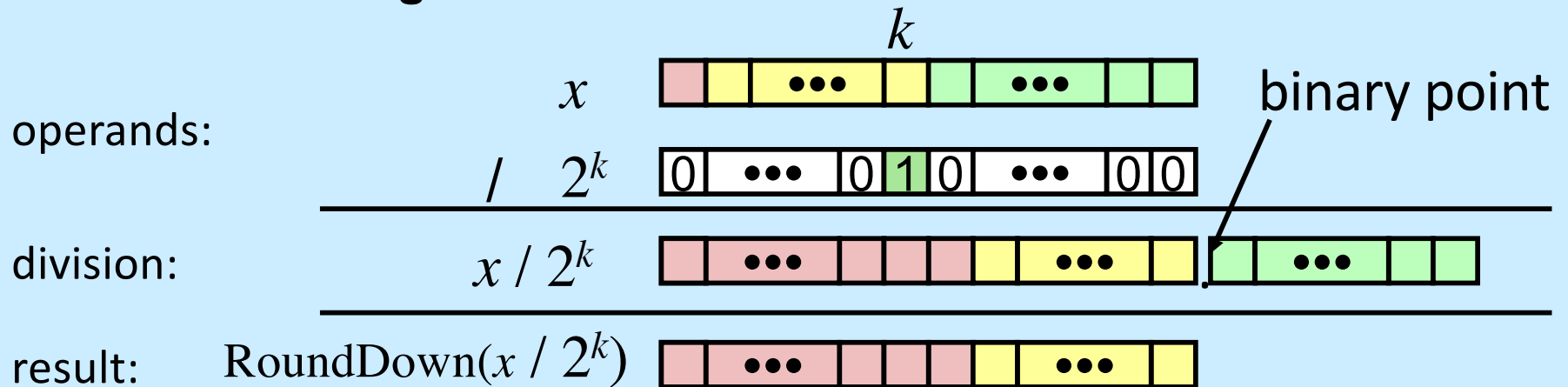


	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

Signed Power-of-2 Divide with Shift

- Quotient of signed and power of 2

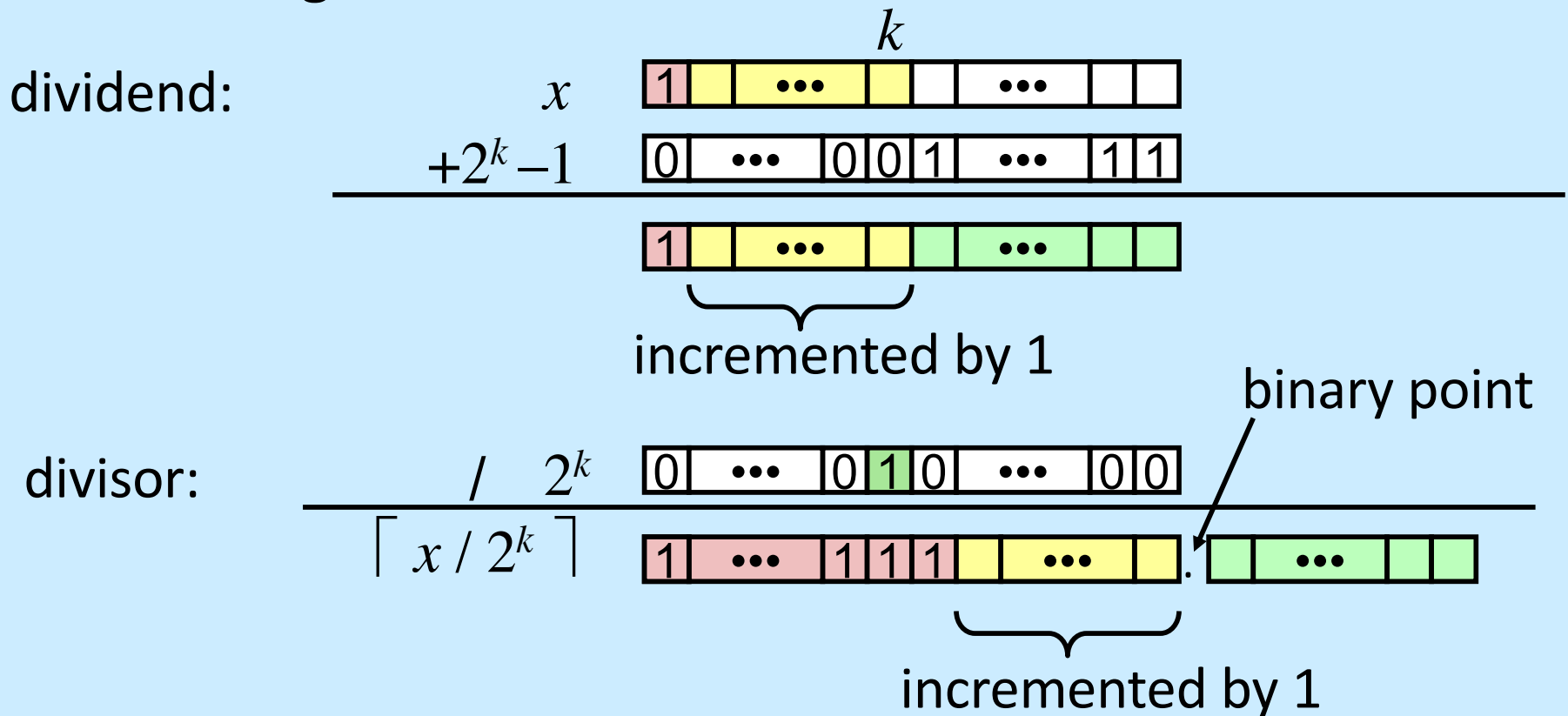
- $x \gg k$ gives $\lfloor x / 2^k \rfloor$
- uses arithmetic shift
- rounds wrong direction when $x < 0$



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
$y \gg 1$	-7606.5	-7607	E2 49	1 1100010 01001001
$y \gg 4$	-950.8125	-951	FC 49	1111 1100 01001001
$y \gg 8$	-59.4257813	-60	FF C4	11111111 11000100

Correct Power-of-2 Divide (Cont.)

Case 2: rounding



Biasing adds 1 to final result

Why Should I Use Unsigned?

- ***Don't*** use just because number nonnegative

- easy to make mistakes

```
unsigned i;  
for (i = cnt-2; i >= 0; i--)  
    a[i] += a[i+1];
```

- can be very subtle

```
#define DELTA sizeof(int)  
int i;  
for (i = CNT; i-DELTA >= 0; i-= DELTA)  
    . . .
```

- ***Do*** use when using bits to represent sets

- logical right shift, no sign extension