

# CS 33

## Introduction to C Part 7

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective.” 2<sup>nd</sup> Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

## Implicit Conversions (1)

```
float x, y=2.0;
int i=1, j=2;

x = i/j + y;
/* what's the value of x? */
```

**x**'s value will be 2, since the result of the (integer) division of **i** by **j** will be 0.

## 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? */
```

Here the values of **i** and **j** are converted to float before being assigned to **a** and **b**, thus the value assigned to **x** is 2.5.

## 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? */
```

Here we do the int-to-float conversion explicitly; **x**'s value will be 2.5.

## 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!)

“Coercion” is a commonly accepted term for one use of casts. “Intimidation” is not. The concept is more commonly known as a “sidecast”. Coercion means to convert something of one datatype to another. Intimidation (or sidecasting) means to treat an instance one datatype as being another datatype without doing any conversion of the actual data. Intimidation works only for pointer datatypes.

## 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);
```

The call to **swap** makes sense as long as what **x** and **y** point to are the same size as **int**'s.

The moral is to be careful with casting, particularly intimidation casts, since they effectively turn off type checking.

## 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

The ***void \**** type is an exception to the rule that the type of the target of a pointer must be known.



## 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 */
```

Dereferencing a pointer must result in a value with a useful type. “void” is not a useful type.

## Swap, Revisited

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

Can we write a version of **swap** that handles a variety of data types?

## 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;
    }
}
```

Note that there is a function in the C library that one may use to copy arbitrary amounts of data — it's called **memmove**. To see its documentation, use the Linux command “man memmove”.

## 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]);
}
```

Here **func** is declared to be a pointer to a function that takes an **int** as an argument and returns an **int**.

What's the difference between a pointer to a function and a function? A pointer to a function is, of course, the address of the function. The function itself is the code comprising the function. Thus, strictly speaking, if **func** is the name assigned to a function, **func** really represents the address of the function. You might think that we should invoke the function by saying “**\*func**”, but it's understood that this is what we mean when we say “**func**”. Thus, when one calls **ArrayBop**, one supplies the name of the desired function as the third argument, without prepending “&”.

## Fun with Functions (3)

```
int triple(int arg) {  
    return 3*arg;  
}  
  
int main() {  
    int A[20];  
    ... /* initialize A */  
    ArrayBop(A, 20, triple);  
    return 0;  
}
```

Here we define another function that takes a single **int** and returns an **int**, and pass it to **ArrayBop**.

## 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;
```

A standard convention for C is that names of datatypes end with “\_t”. Note that it’s not necessary to give the struct a name in this example (we could have omitted the “complex” following “struct”). It’s also not necessary for the name of the type to be different from the name of the struct. Though it’s a bit confusing, we could have coded the above as:

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

```
complex i, *ip;
```

After doing this, “struct complex” and “complex” would mean exactly the same thing.

## And ...

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

MyFunc_t f;

// you must do its definition the long way

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

**MyFunc\_t** is the type of a function that takes two **void \*** arguments and returns a **void \***. Note that **f**, declared as a **MyFunc\_t**, is function, not a pointer to a function.

## 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

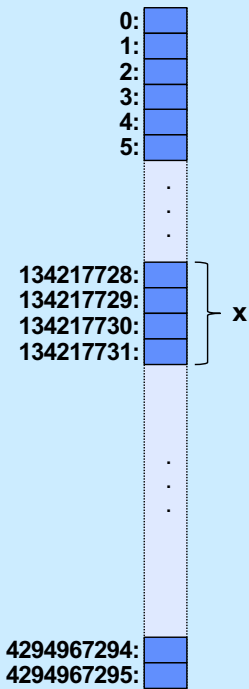
# CS 33

## Data Representation, Part 1

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective.” 2<sup>nd</sup> Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

## Representing Data in Memory

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



In the diagram, **x** is an `int` occupying bytes 134217728, 134217729, 134217730, and 134217731. Its address is 134217728; its size is 4 (bytes).

# Unsigned Integers

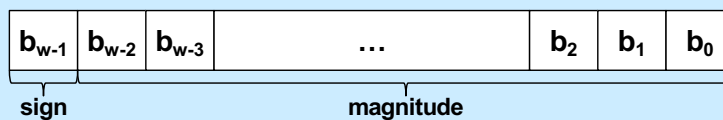
$b_{w-1}$	$b_{w-2}$	$b_{w-3}$	...	$b_2$	$b_1$	$b_0$
-----------	-----------	-----------	-----	-------	-------	-------

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

If a computer word is to be interpreted as an unsigned integer, we can do so as shown in the slide, where  $w$  is the number of bits in the word.

## 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

We might also want to interpret the contents of a computer word as a signed integer. There are a few options for how to do this. One straightforward approach is shown in the slide, where we use the high-order (leftmost) bit as the “sign bit”: 0 means positive and 1 means negative. However, this has the somewhat weird result that there are two representations of zero. This further means that the computer would have to have two implementations of arithmetic instructions: one for signed arithmetic, the other for unsigned arithmetic.



# 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!

In ones' complement, a number is positive if its leftmost bit is zero negative otherwise. We negate a number by complementing **all** its bits. Thus, if the leftmost bit is zero, a one in position **i** of the remaining bits contributes a value of **2<sup>i</sup>** and a zero contributes nothing. But if the leftmost bit is one, a zero in position **i** contributes a value of **-2<sup>i</sup>** and a one contributes nothing.

Note that the most-significant bit serves as the sign bit. But, as with sign-magnitude, the computer would need two sets of instructions: one for signed arithmetic and one for unsigned.

# 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!

There's only one zero!

Two's complement is used on pretty much all of today's computers to represent signed integers.

Note that the high-order (most-significant) bit represents  $-2^{w-1}$ . All the other bits represent positive numbers.

## 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  $-value$ ?

$(\sim value) + 1$

To negate a two's-complement number, simply complement each of its bits, then add one to the result. We show why this works in the next slide.

## Signed Integers

- Negating two's complement (continued)

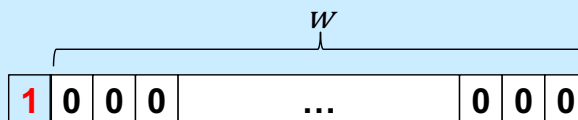
$$value + (\sim value + 1)$$

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

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

$$= 2^w$$

=



If we add to the two's complement representation of a w-bit number the result of adding one to its bitwise complement, we get a w+1-bit number whose low-order w bits are zeroes and whose high-order bit is one. However, since we're constrained to only w bits, the result is a w-bit value of all zeroes, plus an overflow. If we ignore the overflow, the result is zero.

## 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

Why the signed integer types use the arithmetic right shift will be clear by the end of this lecture.

## Numeric Ranges

- **Unsigned Values**

- $UMin = 0$

- 000...0**

- $UMax = 2^w - 1$

- 111...1**

- **Two's Complement Values**

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

- 100...0**

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

- 011...1**

- **Other Values**

- Minus 1

- 111...1**

Values for  $W = 16$

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



## 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

$$UMax = 2 * TMax + 1$$

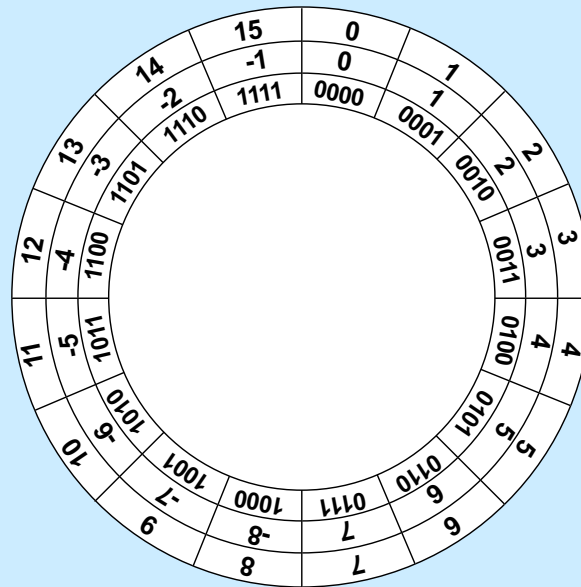
- **C Programming**

- **#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



Unsigned computer arithmetic is performed modulo 2 to the power of the computer's word size. The outer ring of the figure demonstrates arithmetic modulo  $2^4$ . To see the result, for example, of adding 3 to 2, start at 2 and go around the ring three units in the clockwise direction. If we add 5 to 14, we start at 14 and move 5 units clockwise, to 3. Similarly, to subtract 3 from 1, we start at one and move three units counterclockwise to 14.

What about two's-complement computer arithmetic? We know that the values encoded in a 4-bit computer word range from -8 to 7. How do we arrange them in the ring? As shown in the second ring, it makes sense for the non-negative numbers to be in the same positions as the corresponding unsigned values. It clearly makes sense for the integer coming just before 0 to be -1, the integer just before -1 to be -2, etc. Thus, since we have a ring, the integer following 7 is -8. Now we can see how arithmetic works for two's-complement numbers. Adding 3 to 2 works just as it does for unsigned numbers. Subtracting 3 from 1 results in -2. But adding 3 to 6 results in -7; and adding 5 to -2 results in 3.

The innermost ring shows the bit encodings for the unsigned and two's-complement values. The point of all this is that, with only one implementation of arithmetic, we can handle both unsigned and two's-complement values. Thus, adding unsigned 5 and 9 is equivalent to adding two's-complement 5 and -7. The result will 1110, which, if interpreted as an unsigned value is 14, but if interpreted as a two's-complement value is -2.

## 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;
```

Supplied by CMU.

Note that the kind of casting done here is what we called "intimidation" in the previous lecture: no actual conversion takes place, but the value is reinterpreted according to the cast.

## 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 <sub>1</sub>	Constant <sub>2</sub>	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

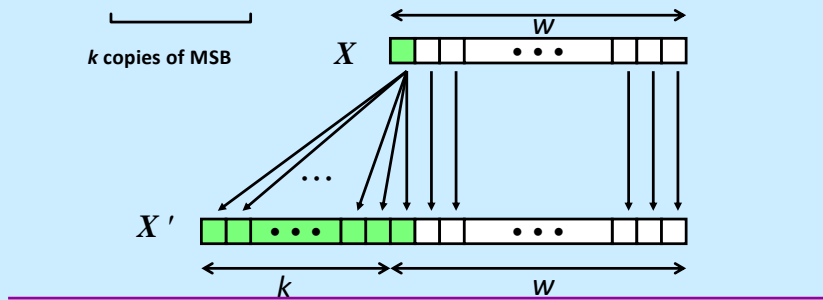
# Sign Extension

- **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
<b>x</b>	15213	3B 6D	00111011 01101101
<b>ix</b>	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
<b>y</b>	-15213	C4 93	11000100 10010011
<b>iy</b>	-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?

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

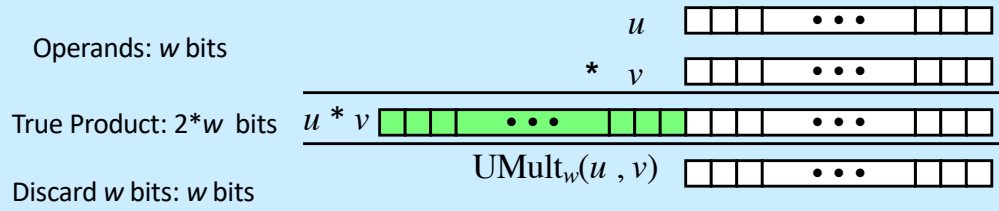
$$\begin{aligned} val_{w+1} &= -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \end{aligned}$$

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

Sign extension clearly works for positive and zero values (where the sign bit is zero). But does it work for negative values? The first line of the slide shows the computation of the value of a  $w$ -bit item with a sign bit of one (i.e., it's negative). The next two lines show what happens if we extend this to a  $w+1$ -bit item, extending the sign bit. What had been the sign bit becomes one of the value bits, and its contribution to the value is now positive rather than negative. But this is compensated by the new sign bit, whose contribution is a negative value, twice as large as the original sign bit. Thus, the net effect is for there to be no change in the value.

We do this again, extending to a  $w+2$ -bit item, and again, the resulting value is the same as what we started with.

# Unsigned Multiplication

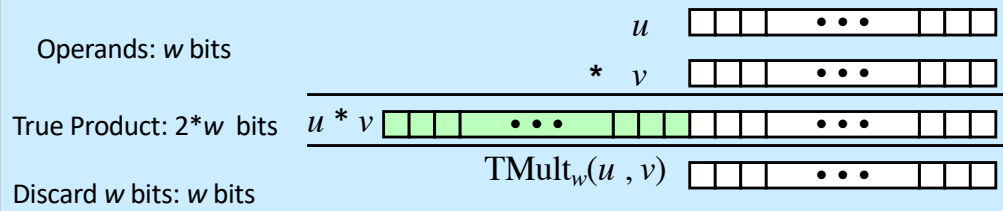


- **Standard multiplication function**
  - ignores high order  $w$  bits
- **Implements modular arithmetic**

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

Supplied by CMU.

## Signed Multiplication



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

Supplied by CMU.

Why is it that the "true product" is different from that of unsigned multiplication? Consider what the true product should be if the multiplier is  $-1$  and the multiplicand is  $1$ . The multiplier is a  $w$ -bit word of all ones; the multiplicand is a  $w$ -bit word of all zeroes except for the least-significant bit, which is  $1$ . The high-order  $w$  bits of the true product should be all ones (since it's negative), but with unsigned multiplication they'd be all zeroes. However, since we're ignoring the high-order  $w$  bits, this doesn't matter.

## Power-of-2 Multiply with Shift

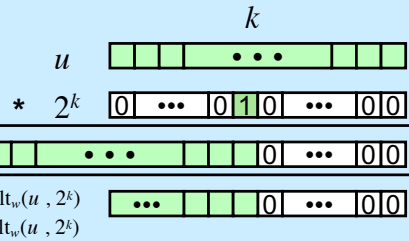
- **Operation**

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

operands:  $w$  bits

true product:  $w+k$  bits

discard  $k$  bits:  $w$  bits



- **Examples**

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

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

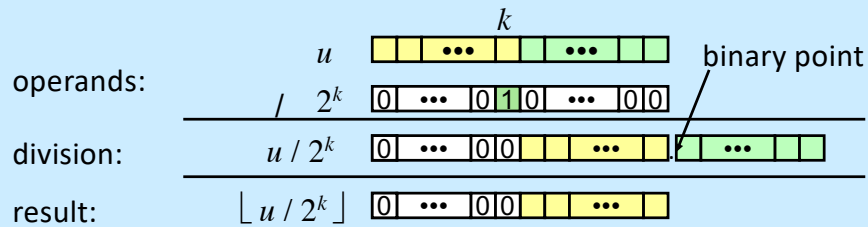
- most machines shift and add faster than multiply
  - » compiler generates this code automatically

## Unsigned Power-of-2 Divide with Shift

- Quotient of unsigned and power of 2

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

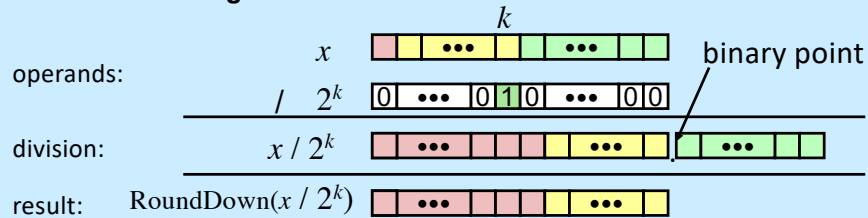
- uses logical shift



## 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 >> 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

Supplied by CMU.

Recall that with two's-complement, all the bits other than the most-significant represent positive values. Thus, we are shifting off (to the right) bits that should be adding a positive value to the number, but now are lost. Thus, if any of these bits are one, after shifting the resulting value will be less than it should be (i.e., more negative).

## Correct Power-of-2 Divide

- Quotient of negative number by power of 2

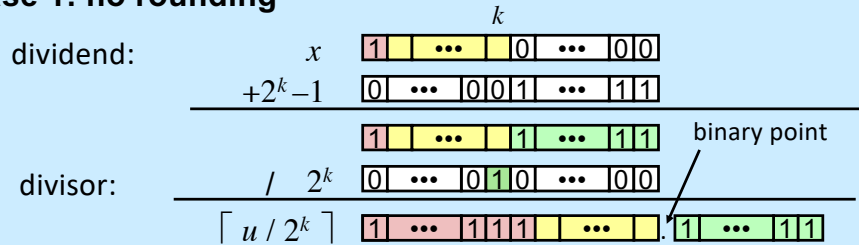
- want  $\lceil x / 2^k \rceil$  (round toward 0)

- compute as  $\lfloor (x+2^k-1) / 2^k \rfloor$

- » in C:  $(x + (1 << k) - 1) >> k$

- » biases dividend toward 0

### Case 1: no rounding



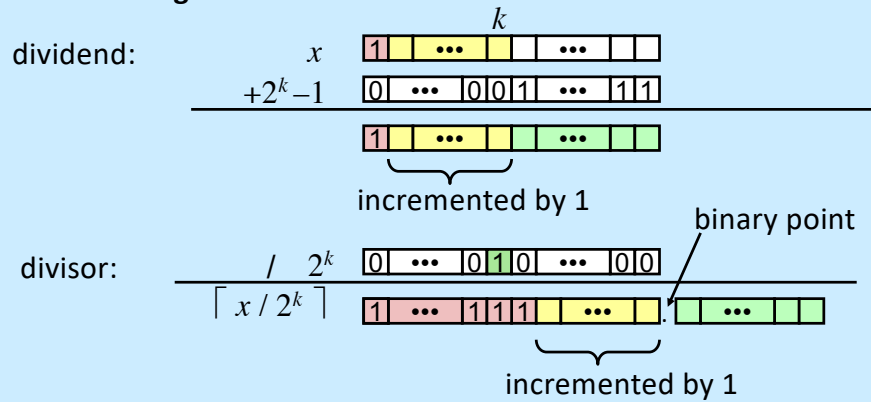
***Biasing has no effect***

Supplied by CMU.

If the least-significant  $k$  bits are all zeroes, then adding in the bias and shifting right by  $k$  bits eliminates any effect of adding the bias.

## Correct Power-of-2 Divide (Cont.)

### Case 2: rounding



***Biasing adds 1 to final result***

Supplied by CMU.

If any of the least-significant  $k$  bits are one, then adding the bias to them causes a carry of one to the bits to their left. Thus, after shifting, the number that's represented by the remaining bits is one greater (less negative) than it would have been if the bias had not been added.



## 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

Supplied by CMU.

Note that “sizeof” returns an unsigned value. (Recall that, when mixing signed and unsigned items in an expression, the result will be unsigned.)