

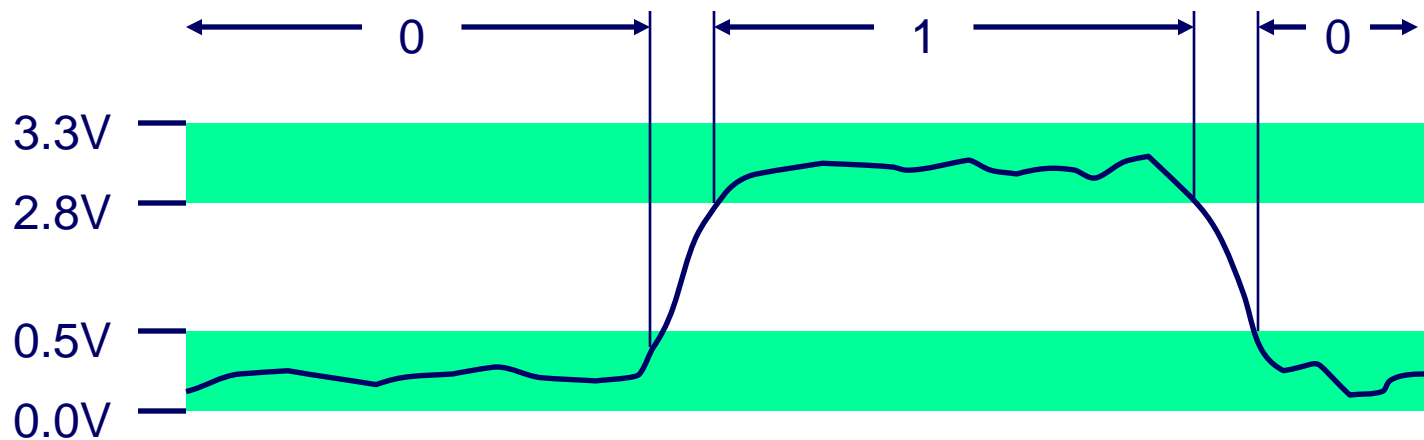
CS 201

Data Representation

Bits

Computers are digital

- Binary (base 2)
 - Easy to store discrete On-Off states (1 and 0)
 - Reliably transmitted on noisy and inaccurate wires



Converting bases

Number systems

- $B = \text{base}$
- $wxyz_B = w \cdot B^3 + x \cdot B^2 + y \cdot B^1 + z \cdot B^0$
 - $2135_{10} = 2 \cdot 10^3 + 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$
- Range of digits = 0 to $B-1$
 - Base 10 (decimal) = 10 digits (0-9)
 - Base 16 (hexadecimal) = 16 digits (0-9, ABCDEF)
 - Base 2 (binary) = 2 digits (0,1)
 - Base 8 (octal) = 8 digits (0-7)
- Examples
 - $B8_{16} = 11 \cdot 16^1 + 8 \cdot 16^0 = 184_{10}$
 - $010011_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19_1$
 - $327_8 = 3 \cdot 8^2 + 2 \cdot 8^1 + 7 \cdot 8^0 = 215_{10}$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Converting bases

From Base 10 to other bases

- Find largest power x of base less than number n
- Find largest base digit b where $b * x < n$
- Recursively repeat using $n - (b * x)$
- Example: $15213_{10} = 1 * 10^4 + 5 * 10^3 + 2 * 10^2 + 1 * 10^1 + 3 * 10^0$

Base 16 (hexadecimal)

- » $x=16$ 65536 4096 256 16 1
- » $15213_{10} =$ 0 3 B 6 D₁₆
- » $3B6D_{16} = 3 * 16^3 + 11 * 16^2 + 6 * 16^1 + 13 * 16^0$
- » Written in C as 0x3b6d

Base 2 (binary)

- » $x=2$ 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1
- » $15213_{10} =$ 0 1 1 1 0 1 1 0 1 1 0 1 1 0 1₂
- » $11101101101101_2 = 1 * 2^{13} + 1 * 2^{12} + 1 * 2^{11} + 0 * 2^{10} + 1 * 2^9 + \text{etc...}$

From Base 2 binary to bases that are powers of two

- Done by grouping bits and assigning digits
- Example of binary to hex

- 11 1011 0110 1101 = 3 B 6 D

Practice

Convert the following

- 10110111_2 to Base 10
- 11011001_2 to Base 16
- $0x2ae$ to Base 2
- $0x13e$ to Base 10
- 150_{10} to Base 2
- 301_{10} to Base 16

Base 2

128 64 32 16 8 4 2 1

Base 16

256 16 1

Hex digits

$a=10=1010_2$

$b=11=1011_2$

$c=12=1100_2$

$d=13=1101_2$

$e=14=1110_2$

$f=15=1111_2$

Practice

Convert the following

- 10110111_2 to Base 10 = $128+32+16+4+2+1 = 183_{10}$
- 11011001_2 to Base 16 = $0xd9$
- $0x2ae$ to Base 2 = $0010\ 1010\ 1110_2$
- $0x13e$ to Base 10 = 318_{10}
- 150_{10} to Base 2 = 010010110_2
- 301_{10} to Base 16 = $0x12d$

Base 2

128 64 32 16 8 4 2 1

Base 16

256 16 1

Hex digits

$a=10=1010_2$

$b=11=1011_2$

$c=12=1100_2$

$d=13=1101_2$

$e=14=1110_2$

$f=15=1111_2$

Byte-Oriented Memory Organization

Memory organized as an array of bytes

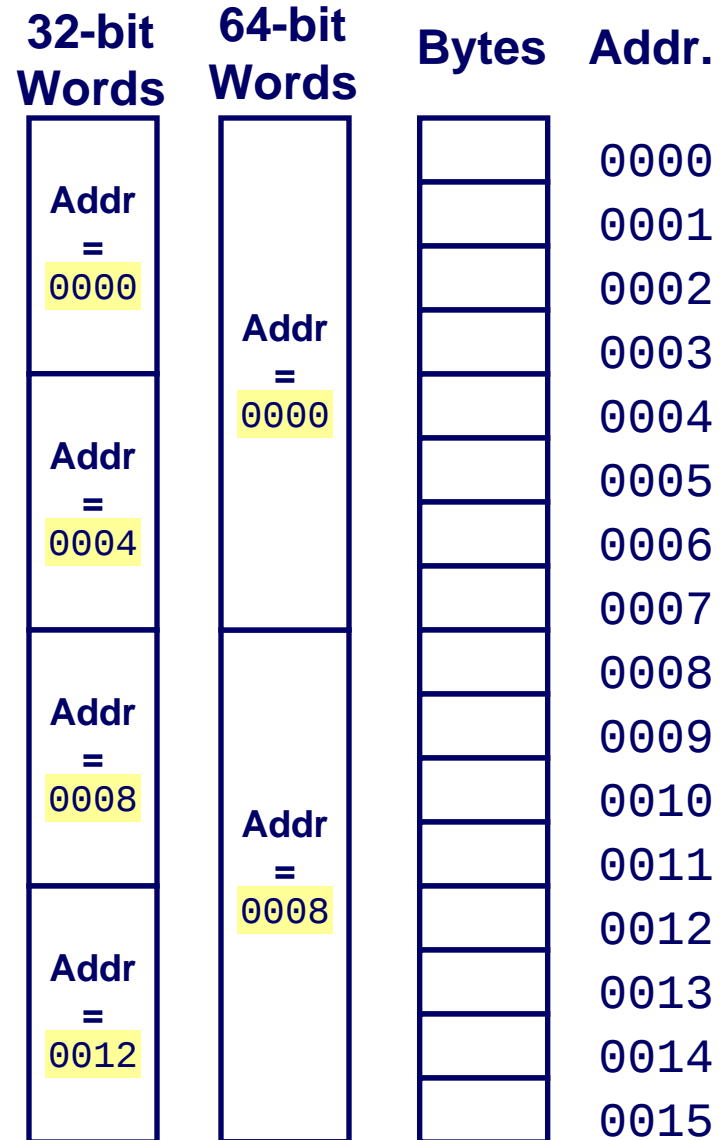
- Addressable unit of memory is a byte
- Byte=8 bits

- Binary 00000000_2 to 11111111_2
- Decimal: 0_{10} to 255_{10}
- Hexadecimal 00_{16} to FF_{16}

Word size

Nominal size of integers and pointers on a system

- Mostly 32 bits (4 bytes)
- Now 64 bits (8 bytes)
- Addresses of successive words differ by
 - 4 (32-bit)
 - 8 (64-bit)



Data Representations

Sizes of C Objects (in Bytes)

■ C Data Type	Compaq Alpha	Typical 32-bit	Intel IA32
● int	4	4	4
● long int	8	4	4
● char	1	1	1
● short	2	2	2
● float	4	4	4
● double	8	8	8
● long double	8	8	10/12
● char *	8	4	4

» *Or any other pointer

Byte Ordering

How should bytes within multi-byte word be ordered in memory?

Conventions

- Sun's, PowerPC Macs are “Big Endian” machines
 - Least significant byte has highest address
- Alphas, PCs, x86 Macs are “Little Endian” machines
 - Least significant byte has lowest address

Byte Ordering Example

Big Endian

- Least significant byte has highest address

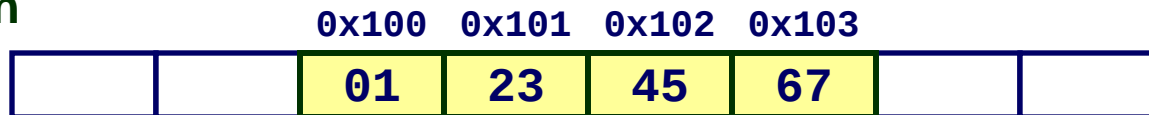
Little Endian

- Least significant byte has lowest address (LLL)

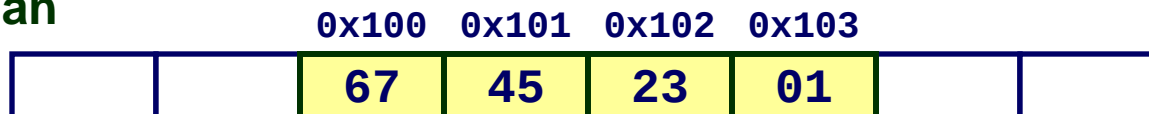
Example

- Variable x has 4-byte representation 0x01234567
- Address given by &x is 0x100

Big Endian



Little Endian



Endian

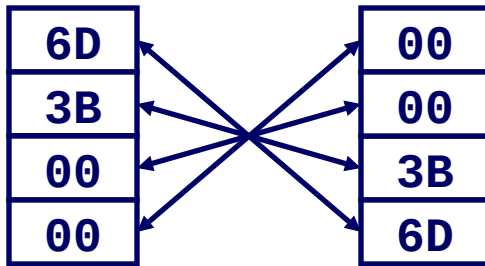
```
int A = 15213;  
int B = -15213;  
long int C = 15213;
```

Decimal: 15213

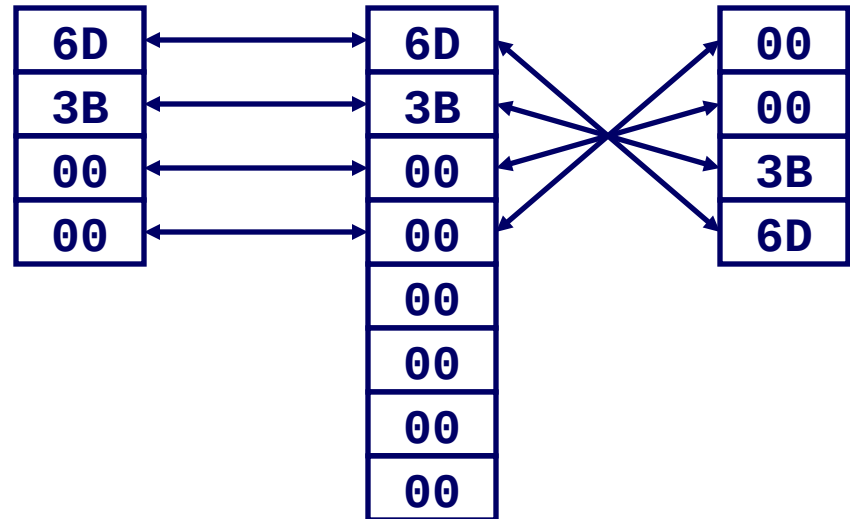
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

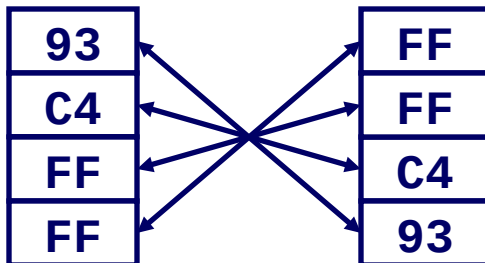
Linux/Alpha A Sun A



Linux C Alpha C Sun C



Linux/Alpha B Sun B



Two's complement representation
(Covered later)

Endian

How do you test endian-ness?

- Direct inspection of memory via gdb

- endian

- » gdb endian

- » break 5

- » run

- » p /x &i

- » x/b &i

- » x/b ...

```
#include <stdio.h>
main()
{
    int i=0x01020304;
    printf("%d\n",i);
}
```

<http://thefengs.com/wuchang/work/courses/cs201/class/04/endian.c>

Endian

How do you test endian-ness?

■ Simple program from book (show_bytes)

```
#include <stdio.h>
#include <string.h>
typedef unsigned char *byte_pointer;
void show_bytes(byte_pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf(" %.2x", start[i]);
    printf("\n");
}

void show_int(int x)
{
    show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x)
{
    show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x)
{
    show_bytes((byte_pointer) &x, sizeof(void*));
}
```

```
int main()
{
    int i=0x01020304;
    float f=2345.6;
    int *ip=&i;
    char *s = "ABCDEF";

    show_int(i);
    show_float(f);
    show_pointer(ip);
    show_bytes(s,strlen(s));
}
```

Output:
04 03 02 01
9a 99 12 45
d0 a7 af ff
41 42 43 44 45 46

Representing pointers

A *pointer* is a variable containing a memory address of an object of a particular data type

- “&” operator gives address of object
- Contains a “reference” address for data

Use the “*” to declare a pointer type

```
char* cp;    // Declares cp to be a pointer to a character
int* ip;    // Declares ip to be a pointer to an integer
```

- Both store address locations

Why does the declaration of a pointer have to tell the data type it points to?

- Pointer arithmetic is done based on data type
- Type determines what is returned when “dereferenced”
 - Incrementing an (int *) adds 4 to pointer
 - Incrementing a (char *) adds 1 to pointer

Representing pointers

Example:

```
int x; // x is an integer
```

```
// &x is the address of x
```

```
int *p; // p is a pointer to an integer
```

```
int *p = &x; // p is initialized to &x
```

Example

```
int x;
```

```
int *p = &x;
```

What is the data type of x?

What is the data type of p?

What is the data type of *p?

What about &x?

Pointer Assignment / Dereference

Dereferencing pointers

- Returns the data that is stored in the memory location specified by the pointer

The unary operator `*` is used to dereference a pointer variable

```
int x = 1, y = 2, z[10];
int *ip = &x;
■ y = *ip;           // y is now 1
■ *ip = 0;           // x is now 0
```

Dereferencing uninitialized pointers:

- What happens?
- Try it:

```
int *ip;
*ip = 3;
```

Segmentation fault

Representing pointers

```
#include <stdio.h>

int main(int argc, char** argv)
{
    int *ip;
    int i;
    i = 3;
    ip = &i;
    printf("%d\n", *ip);
    return(0);
}
```

Output is 3

Recall: Call by value vs. reference

Call by reference implemented via pointer passing

```
void swap(int *px, int *py) {  
    int tmp;  
    tmp = *px;  
    *px = *py;  
    *py = tmp;  
}
```

- Swaps the values of the variables x and y if px is &x and py is &y
- Uses integer pointers instead of integers

Otherwise, call by value...

```
void swap(int x, int y) {  
    int tmp;  
    tmp = x;  
    x = y;  
    y = tmp;  
}
```

Pointers and arrays

Arrays

- Stored contiguously in one block of memory
- Indexed based on element size
- Index specifies offset from start of array in memory
 - `int a[20];`
 - “a” used alone is a pointer containing address of the start of the integer array
- Elements can be accessed using index or via pointer increment and decrement
 - Pointer increments and decrements based on type of array

Example

```
#include <stdio.h>
main()
{
    char *str="abcdefg\n";
    char *x;
    x = str;
    printf("str[0]: %c str[1]: %c str[2]: %c str[3]: %c\n",
           str[0],str[1],str[2],str[3]);
```

```
    printf("x: %x *x: %c\n",x,*x); x++;
    printf("x: %x *x: %c\n",x,*x); x++;
    printf("x: %x *x: %c\n",x,*x); x++;
    printf("x: %x *x: %c\n",x,*x);
```

```
int numbers[10], *num, i;
for (i=0; i < 10; i++) numbers[i]=i;
num=(int *) numbers;
```

```
printf("num: %x *num: %d\n",num,*num); num++;
printf("num: %x *num: %d\n",num,*num); num++;
printf("num: %x *num: %d\n",num,*num); num++;
printf("num: %x *num: %d\n",num,*num);
```

```
num=(int *) numbers;
printf("numbers: %x num: %x &numbers[4]: %x num+4: %x\n",
       numbers, num, &numbers[4],num+4);
printf("%d %d\n",numbers[4],*(num+4));
```

Output:

str[0]: a str[1]: b str[2]: c str[3]: d

x: 8048690 *x: a

x: 8048691 *x: b

x: 8048692 *x: c

x: 8048693 *x: d

num: fffe0498 *num: 0

num: fffe049c *num: 1

num: fffe04a0 *num: 2

num: fffe04a4 *num: 3

numbers: fffe0498 num: fffe0498

&numbers[4]: fffe04a8 num+4: fffe04a8

4 4

Representing pointers

Given the following code...

```
main()
{
    int B = -15213;
    int *P = &B;
}
```

Suppose the address of B is 0xbffff8d4 and the address of P is 0xbffff8d0

What is the value of P?

What is the size of P?

Write the value of each byte of P in order as they appear in memory.

GDB exercise

<code>p /x B</code>	<code>p /x &B</code>
<code>p /x P</code>	<code>p /x &P</code>
<code>x/b &P</code>	<code>x/b &B</code>
<code>x/w &P</code>	<code>x/w &B</code>

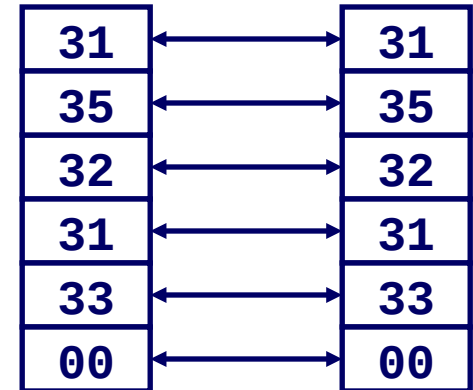
Representing strings

Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
 - Standard 7-bit encoding of character set
 - Other encodings exist
 - Character “0” has code 0x30
 - » Digit i has code $0x30+i$
- String should be null-terminated
 - Final character = 0

```
char S[6] = "15213";
```

Linux/Alpha S Sun S



Compatibility

- Byte ordering not an issue
 - Data are single byte quantities
- Text files generally platform independent
 - Except for different conventions of line termination character(s)!

Representing strings

Unicode

- **ASCII only suitable for English text**
 - 7 bit encoding can not cover characters in all languages
- **16-bit unicode character set**
 - Supports Greek, Russian, Chinese, etc.
- **Default encoding for strings in Java**
- **Support in C libraries for character set**

Encoding integers

Support for two types

- unsigned and signed
- Both are the same size (4 bytes or 32-bits on IA32)
- Differ based on how bits are interpreted

Signed integers (default)

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

- Encodes -2^{31} to $(2^{31}-1)$
- -2,147,483,648 to 2,147,483,647

Unsigned integers

```
unsigned int i;  
printf("%u\n", i)
```

- Encodes 0 to $(2^{32} - 1)$

Encoding Integers

Unsigned

$$\begin{aligned} B2U(X) &= \sum_{i=0}^{w-1} x_i \cdot 2^i \\ &= x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \end{aligned}$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

MSB
if 1, negative

16-bit example

■ Unsigned

32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1

■ Signed

-32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1

Encoding Integers

- Allows one to implement “subtraction” via addition

- Example

```
short int x = 15213;  
short int y = -15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Adding x to y
- Treated as normal addition regardless of sign

Encoding Example (Cont.)

x = 15213: 00111011 01101101
y = -15213: 11000100 10010011

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum	15213		-15213	

Two's complement example for $w = 4$

-8 4 2 1

0000 = 0

0001 = 1

0010 = 2

...

0111 = 7



positive values (lower order 3 digits)

1000 = -8

See the book for proof of this

1001 = -7

1010 = -6

...

1111 = -1



-8 + lower order digits

As we will see, the arithmetic works out.

Exercise: Write -3, -4, and -5 in two's complement format for $w=4$

Exercise: Numeric ranges

For 16 bit numbers ($w=16$), write the greatest positive value and the least negative value, in hex and decimal. What does -1 look like?

Do the same for 32 bits.

Exercise: Numeric ranges

For 16 bit numbers ($w=16$), write the greatest positive value and the least negative value, in hex and decimal. What does -1 look like?

- Greatest positive = $0x7FFF = 32767$
- Least negative = $0x8000 = -32768$
- Negative 1 = $0xFFFF$

Do the same for 32 bits.

Negating with Complement & Increment

For 2's complement, negation can be implemented as the bit-wise complement plus 1

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

Complement

- Observation: $\sim x + x == 1111\dots11_2 == -1$

$$\begin{array}{r} x \quad 10011101 \\ + \quad \sim x \quad 01100010 \\ \hline -1 \quad 11111111 \end{array}$$

Increment

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

Comp. & Incr. Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011
y	-15213	C4 93	11000100 10010011

0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Negation with Complementation

More examples

x		$\sim x$		incr($\sim x$)	
0101	5	1010	-6	1011	-5
0111	7	1000	-8	1001	-7
1100	-4	0011	3	0100	4
0000	0	1111	-1	0000	0
1000	-8	0111	7	1000	-8

-8 | 4 | 2 | 1

Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

```
short int          x = 15213;
unsigned short int ux = (unsigned short) x;
short int          y = -15213;
unsigned short int uy = (unsigned short) y;
```

Resulting Value

- No change in bit representation
- Nonnegative values unchanged
 - $ux = 15213$
- Negative values change into (large) positive values
 - $uy = 50323$
 - Why? MSB treated as large positive number rather than large negative one.

Signed vs. unsigned example

x = 11000100 10010011

Weight	x unsigned		x signed	
1	1	1	1	1
2	1	2	1	2
4	0	0	0	0
8	0	0	0	0
16	1	16	1	16
32	0	0	0	0
64	0	0	0	0
128	1	128	1	128
256	0	0	0	0
512	0	0	0	0
1024	1	1024	1	1024
2048	0	0	0	0
4096	0	0	0	0
8192	0	0	0	0
16384	1	16384	1	16384
<u>+32768</u>	1	32768	1	-32768
Sum		50323		-15213

Example

```
#include <stdio.h>
main()
{
    int i = 0xFFFFFFFF;
    unsigned int j = 0xFFFFFFFF;

    printf("Signed: 0x%x is %d , Unsigned: 0x%x is %u\n",i,i,j,j);

    i=0x80000000;
    j=0x80000000;
    printf("Signed: 0x%x is %d , Unsigned: 0x%x is %u\n",i,i,j,j);

    i=0x7FFFFFFF;
    j=0x7FFFFFFF;
    printf("Signed: 0x%x is %d , Unsigned: 0x%x is %u\n",i,i,j,j);

    short int x = 15213;
    short int y = -15213;

    unsigned short int ux = (unsigned short int) x;
    unsigned short int uy = (unsigned short int) y;
    printf("x:%d y:%d ux:%u uy:%u\n", x,y,ux,uy);
}
```

Output:

Signed: 0xffffffff is -1 , Unsigned: 0xffffffff is 4294967295

Signed: 0x80000000 is -2147483648 ,
Unsigned: 0x80000000 is 2147483648

Signed: 0x7fffffff is 2147483647 , Unsigned:
0x7fffffff is 2147483647

x:15213 y:-15213 ux:15213 uy:50323

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;
tx = (int) ux;
uy = (unsigned) ty;
```
- Implicit casting also occurs via assignments and procedure calls

Casting Surprises

Expression Evaluation

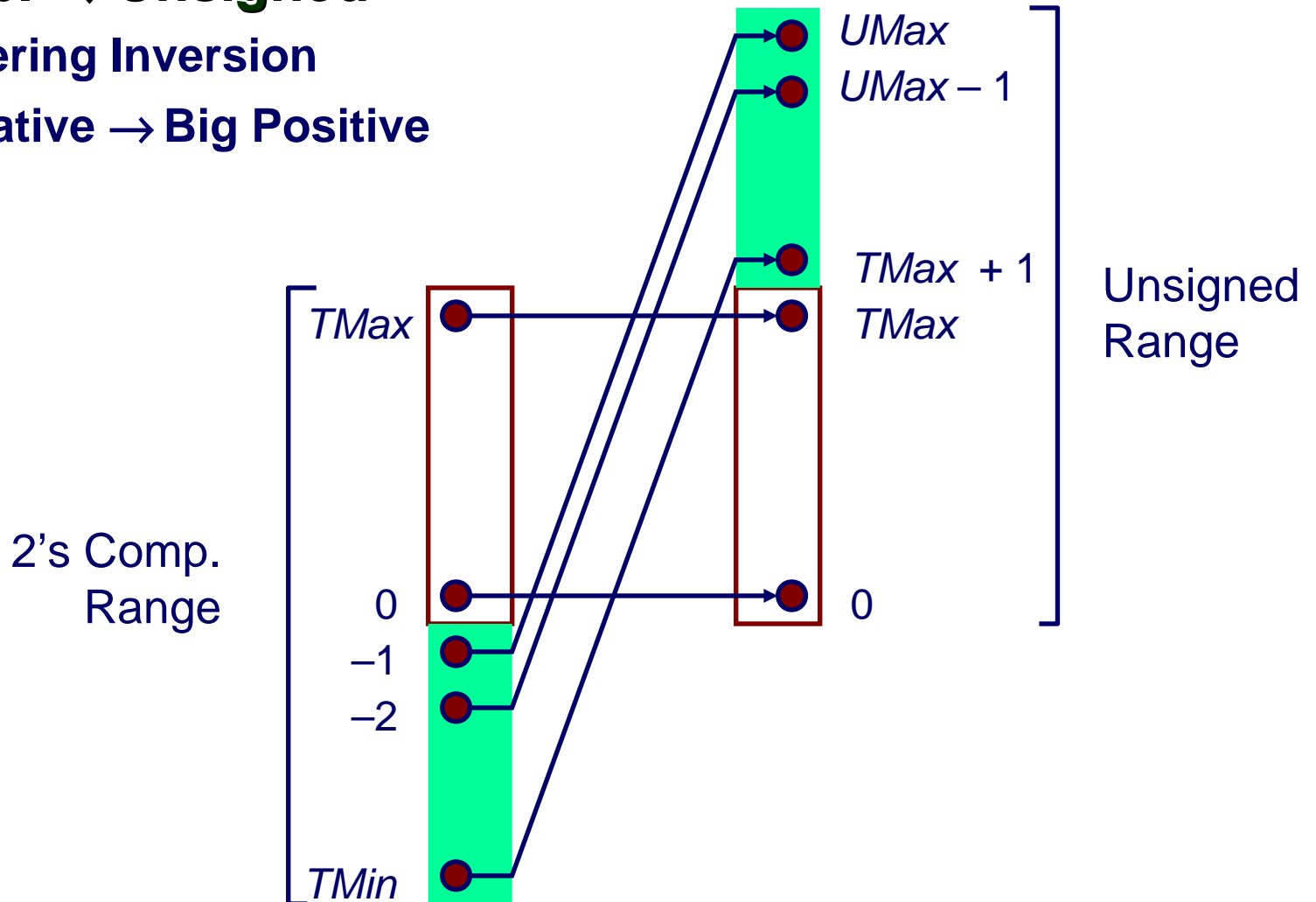
- Mixing unsigned and signed in an expression, signed values implicitly cast to unsigned
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples for $W = 32$

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

Explanation of Casting Surprises

2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



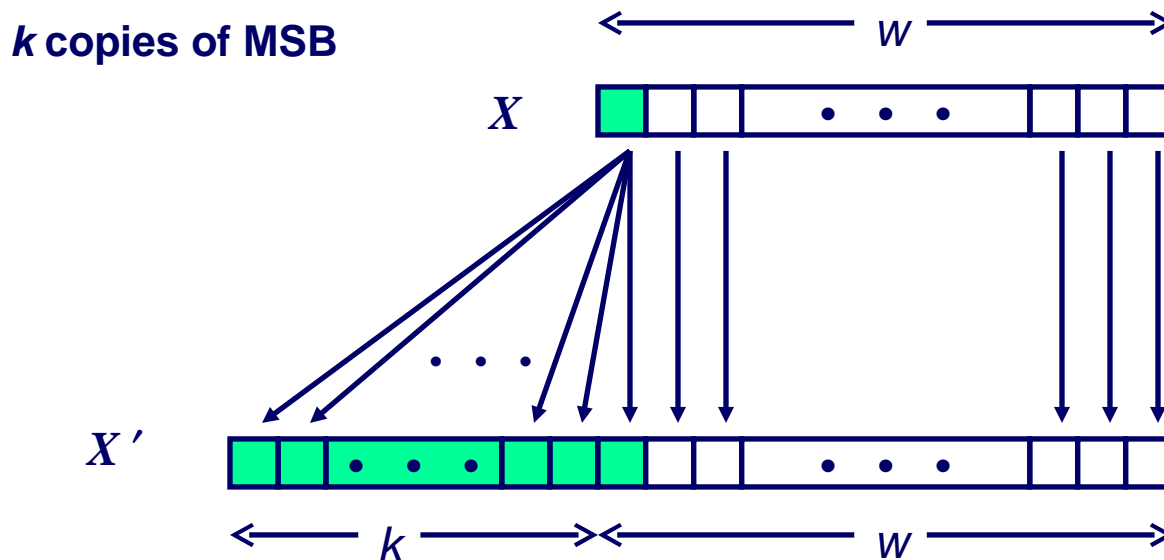
Sign Extension

Task:

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

Rule:

- Make k copies of sign bit:
- $X' = \underbrace{X_{w-1}, \dots, X_{w-1}}_{k \text{ copies of MSB}}, X_{w-1}, X_{w-2}, \dots, X_0$



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
- Exercise: Calculate the hex value of -5 , first for 4 bits, and then for 8 bits.

Representing floats

IEEE Single Precision Floating Point Representation

- Will cover later...

Representing machine code

gcc encodes program as sequence of machine instructions interpreted by CPU

- Each operation encoded as bytes
 - Alphas, SPARCs, Power PCs use 4 byte instructions
 - » Reduced Instruction Set Computer (RISC)
 - PCs use variable length instructions
 - » Complex Instruction Set Computer (CISC)
- Different instruction types and encodings for different machines
 - Most code not binary compatible

Programs are Byte Sequences Too!

Representing machine code

```
int sum(int x, int y)
{
    return x+y;
}
```

- For this example, Alpha & Sun use two 4-byte instructions
- PC (x86) uses 7 instructions with lengths 1, 2, and 3 bytes
 - Same for NT and for Linux
 - NT / Linux not fully binary compatible

Alpha sum

00
00
30
42
01
80
FA
6B

Sun sum

81
C3
E0
08
90
02
00
09

PC sum

55
89
E5
8B
45
0C
03
45
08
89
EC
5D
C3

Different machines use totally different instructions and encodings

Extra slides

Why Should I Use Unsigned?

Don't Use Just Because Number Nonzero

- C compilers on some machines generate less efficient code

```
unsigned i;  
for (i = 1; i < cnt; i++)  
    a[i] += a[i-1];
```

- Easy to make mistakes

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

Do Use When Performing Modular Arithmetic

- Multiprecision arithmetic
- Other esoteric stuff

Do Use When Need Extra Bit's Worth of Range

- Working right up to limit of word size

C Puzzles

- Assume machine with 32 bit word size, two's complement integers
- For each of the following C expressions, either:
 - Argue that is true for all argument values
 - Give example where not true

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- $x < 0 \Rightarrow ((x*2) < 0)$
- $ux \geq 0$
- $x \& 7 == 7 \Rightarrow (x \ll 30) < 0$
- $ux > -1$
- $x > y \Rightarrow -x < -y$
- $x * x \geq 0$
- $x > 0 \&\& y > 0 \Rightarrow x + y > 0$
- $x \geq 0 \Rightarrow -x \leq 0$
- $x \leq 0 \Rightarrow -x \geq 0$

Avoiding Bugs with Signed/Unsigned Numbers

Most languages do not provide for unsigned numbers

- This is often a good thing

```
float sum_elements(float a[ ], unsigned len)
{
    int i;
    float result = 0;
    for (i=0; i <= len - 1; i++)
        result += a[i];
    return result;
}
```

What happens
when len == 0?

**Avoid using unsigned for mathematical operations
unless you are *very* careful**