Bits, Bytes, and Integers

CENG331 - Computer Organization

Adapted from slides of the textbook: http://csapp.cs.cmu.edu/

Hello World!

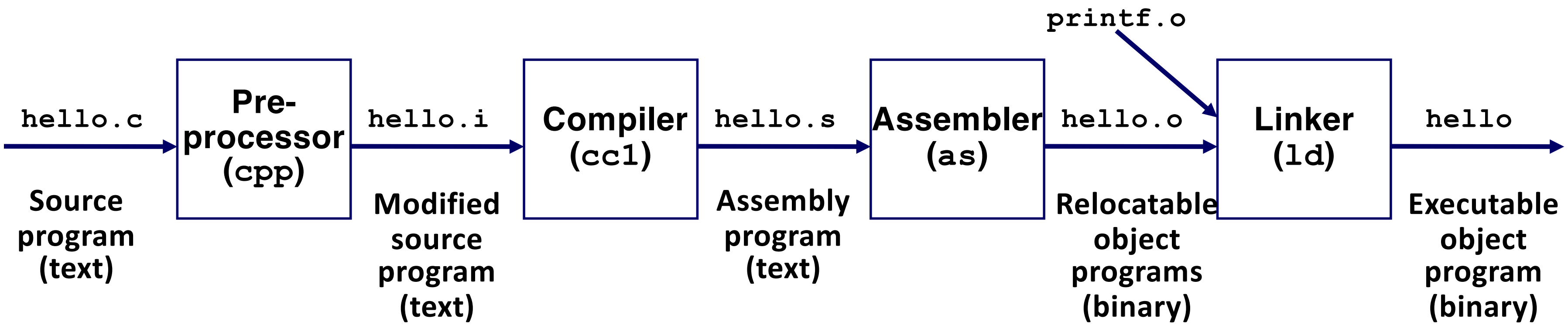
- **What happens under the hood?**

hello.c

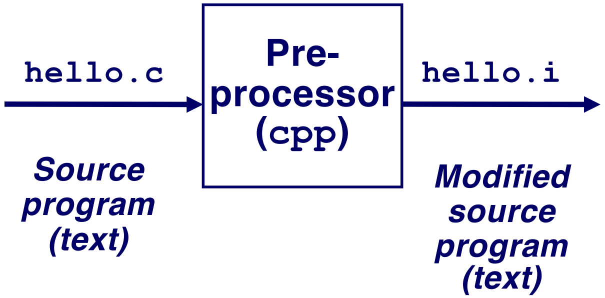
```
#include <stdio.h>


int main() {
  printf("Hello World");
}
```

Compilation of hello.c



Preprocessing




```
#include <stdio.h>


int main() {
   printf("Hello
   World");
}
```

cpp hello.c > hello.i

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
……………………………………..
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
………………………………………..
int main() {
   printf("Hello World");

}
```

Compiler

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
……………………………..
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
………………………………..
int main() {
   printf("Hello World");
}
```

hello.i → **Compiler (cc1)** → hello.s

gcc -Wall -S hello.i > hello.s

```
.file    "hello.c"
         .section
.rodata
.LC0:
         .string "Hello
World"
         .text
.globl main
         .type   main,
@function
main:
         pushl   %ebp
         movl    %esp, %ebp
         subl    $8, %esp
         andl    $-16, %esp
         movl    $0, %eax
         addl    $15, %eax
         addl    $15, %eax
         shrl    $4, %eax
         sall    $4, %eax
         subl    %eax, %esp
         subl    $12, %esp
         pushl   $.LC0
         call    printf
         addl    $16, %esp
         leave
         ret
         .size   main, .-
main
         .section
.note.GNU-
stack,"",@progbits
         .ident  "GCC:
(GNU) 3.4.1"
```

Assembler

```
.file    "hello.c"
         .section
.rodata
.LC0:
         .string "Hello
World"
         .text
.globl main
         .type   main,
@function
main:
         pushl   %ebp
         movl    %esp, %ebp
         subl    $8, %esp
         andl    $-16, %esp
         movl    $0, %eax
         addl    $15, %eax
         addl    $15, %eax
         shrl    $4, %eax
         sall    $4, %eax
         subl    %eax, %esp
         subl    $12, %esp
         pushl   $.LC0
         call    printf
         addl    $16, %esp
         leave
         ret
         .size   main, .-
main
         .section
.note.GNU-
stack,"",@progbits
         .ident  "GCC:
(GNU) 3.4.1"
```

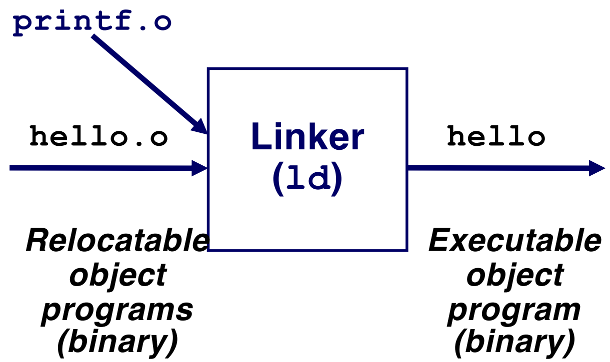
hello.s → **Assembler (as)** → hello.o

as hello.s -o hello.o

```
0000500 nul nul nul nul esc nul nul nul  ht nul nul nul nul nul nul nul
0000520 nul nul nul nul   d etx nul nul dle nul nul nul  ht nul nul nul
0000540 soh nul nul nul eot nul nul nul  bs nul nul nul   % nul nul nul
0000560 soh nul nul nul etx nul nul nul nul nul nul nul   d nul nul nul
0000600 nul nul nul nul nul nul nul nul nul nul nul nul eot nul nul nul
0000620 nul nul nul nul   + nul nul nul  bs nul nul nul etx nul nul nul
0000640 nul nul nul nul   d nul nul nul nul nul nul nul nul nul nul nul
0000660 nul nul nul nul eot nul nul nul nul nul nul nul   0 nul nul nul
0000700 soh nul nul nul stx nul nul nul nul nul nul nul   d nul nul nul
0000720  ff nul nul nul nul nul nul nul nul nul nul nul soh nul nul nul
0000740 nul nul nul nul   8 nul nul nul soh nul nul nul nul nul nul nul
0000760 nul nul nul nul   p nul nul nul nul nul nul nul nul nul nul nul
0001000 nul nul nul nul soh nul nul nul nul nul nul nul   H nul nul nul
0001020 soh nul nul nul nul nul nul nul nul nul nul nul   p nul nul nul
0001040   2 nul nul nul nul nul nul nul nul nul nul nul soh nul nul nul
0001060 nul nul nul nul dc1 nul nul nul etx nul nul nul nul nul nul nul
0001100 nul nul nul nul   " nul nul nul   Q nul nul nul nul nul nul nul
0001120 nul nul nul nul soh nul nul nul nul nul nul nul soh nul nul nul
0001140 stx nul nul nul nul nul nul nul nul nul nul nul   , stx nul nul
0001160  sp nul nul nul  nl nul nul nul  bs nul nul nul eot nul nul nul
0001200 dle nul nul nul  ht nul nul nul etx nul nul nul nul nul nul nul
0001220 nul nul nul nul   L etx nul nul nak nul nul nul nul nul nul nul
0001240 nul nul nul nul soh nul nul nul nul nul nul nul nul nul nul nul
0001260 nul nul nul nul nul nul nul nul nul nul nul nul soh nul nul nul
0001300 nul nul nul nul nul nul nul nul eot nul   q del nul nul nul nul
0001320 nul nul nul nul nul nul nul nul etx nul soh nul nul nul nul nul
0001340 nul nul nul nul nul nul nul nul etx nul etx nul nul nul nul nul
0001360 nul nul nul nul nul nul nul nul etx nul eot nul nul nul nul nul
0001400 nul nul nul nul nul nul nul nul etx nul enq nul nul nul nul nul
0001420 nul nul nul nul nul nul nul nul etx nul ack nul nul nul nul nul
0001440 nul nul nul nul nul nul nul nul etx nul bel nul  ht nul nul nul
0001460 nul nul nul nul   . nul nul nul dc2 nul soh nul  so nul nul nul
0001500 nul nul nul nul nul nul nul nul dle nul nul nul nul   h   e   l
0001520   l   o   .   c nul   m   a   i   n nul   p   r   i   n   t   f
0001540 nul nul nul nul  sp nul nul nul soh enq nul nul   % nul nul nul
0001560 stx  ht nul nul
0001564
```

od –a hello.o

Linker



gcc hello.o –o hello

```
0000500 nul nul nul nul esc nul nul nul  ht nul nul nul nul nul nul nul
0000520 nul nul nul nul   d etx nul nul dle nul nul nul  ht nul nul nul
0000540 soh nul nul nul eot nul nul nul  bs nul nul nul   % nul nul nul
0000560 soh nul nul nul etx nul nul nul nul nul nul nul   d nul nul nul
0000600 nul nul nul nul nul nul nul nul nul nul nul nul eot nul nul nul
0000620 nul nul nul nul   + nul nul nul  bs nul nul nul etx nul nul nul
0000640 nul nul nul nul   d nul nul nul nul nul nul nul nul nul nul nul
0000660 nul nul nul nul eot nul nul nul nul nul nul nul   0 nul nul nul
0000700 soh nul nul nul stx nul nul nul nul nul nul nul   d nul nul nul
0000720  ff nul nul nul nul nul nul nul nul nul nul nul soh nul nul nul
0000740 nul nul nul nul   8 nul nul nul soh nul nul nul nul nul nul nul
0000760 nul nul nul nul   p nul nul nul nul nul nul nul nul nul nul nul
0001000 nul nul nul nul soh nul nul nul nul nul nul nul   H nul nul nul
0001020 soh nul nul nul nul nul nul nul nul nul nul nul   p nul nul nul
0001040   2 nul nul nul nul nul nul nul nul nul nul nul soh nul nul nul
0001060 nul nul nul nul dc1 nul nul nul etx nul nul nul nul nul nul nul
0001100 nul nul nul nul   " nul nul nul   Q nul nul nul nul nul nul nul
0001120 nul nul nul nul soh nul nul nul nul nul nul nul soh nul nul nul
0001140 stx nul nul nul nul nul nul nul nul nul nul nul   , stx nul nul
0001160  sp nul nul nul  nl nul nul nul  bs nul nul nul eot nul nul nul
0001200 dle nul nul nul  ht nul nul nul etx nul nul nul nul nul nul nul
0001220 nul nul nul nul   L etx nul nul nak nul nul nul nul nul nul nul
0001240 nul nul nul nul soh nul nul nul nul nul nul nul nul nul nul nul
0001260 nul nul nul nul nul nul nul nul nul nul nul nul soh nul nul nul
0001300 nul nul nul nul nul nul nul nul eot nul   q del nul nul nul nul
0001320 nul nul nul nul nul nul nul nul etx nul soh nul nul nul nul nul
0001340 nul nul nul nul nul nul nul nul etx nul etx nul nul nul nul nul
0001360 nul nul nul nul nul nul nul nul etx nul eot nul nul nul nul nul
0001400 nul nul nul nul nul nul nul nul etx nul enq nul nul nul nul nul
0001420 nul nul nul nul nul nul nul nul etx nul ack nul nul nul nul nul
0001440 nul nul nul nul nul nul nul nul etx nul bel nul  ht nul nul nul
0001460 nul nul nul nul   . nul nul nul dc2 nul soh nul  so nul nul nul
0001500 nul nul nul nul nul nul nul nul dle nul nul nul nul   h   e   l
0001520   l   o   .   c nul   m   a   i   n nul   p   r   i   n   t   f
0001540 nul nul nul nul  sp nul nul nul soh enq nul nul   % nul nul nul
0001560 stx  ht nul nul
0001564
```

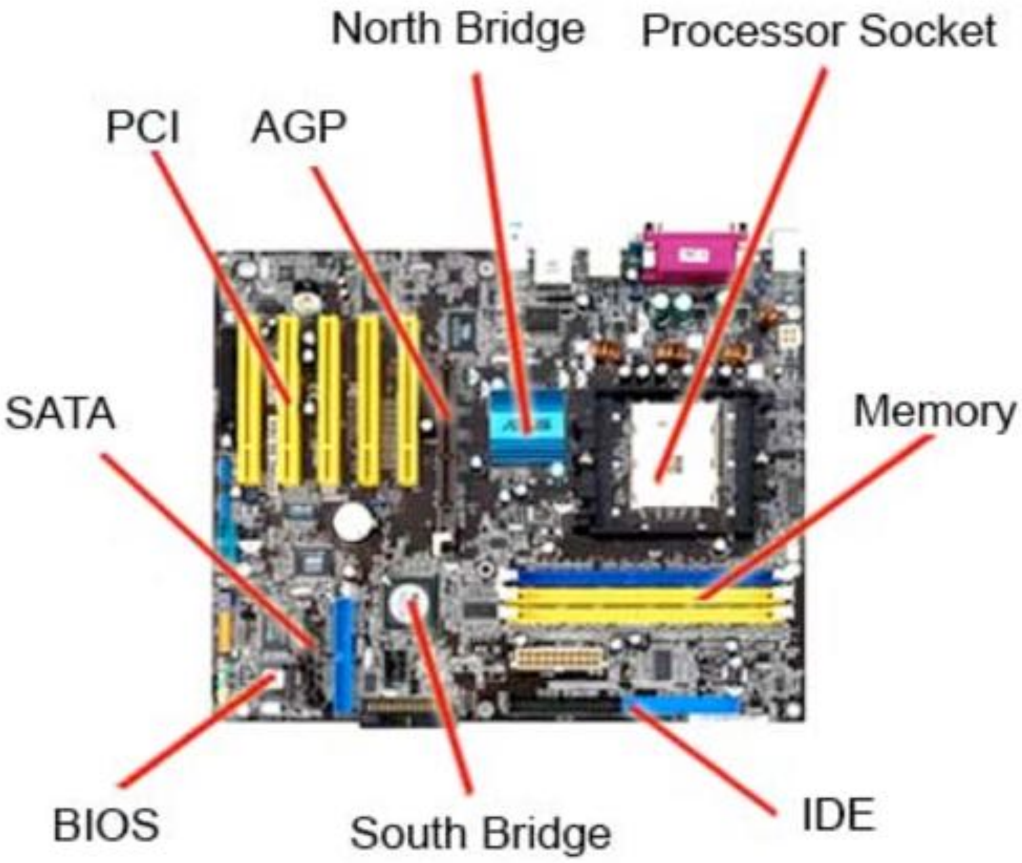
```
0000000 del   E   L   F soh soh soh nul nul nul nul nul nul nul nul nul
0000020 stx nul etx nul soh nul nul nul   @ stx eot  bs   4 nul nul nul
0000040   X  cr nul nul nul nul nul nul   4 nul  sp nul bel nul   ( nul
0000060   ! nul  rs nul ack nul nul nul   4 nul nul nul   4 nul eot  bs
0000100   4 nul eot  bs   ` nul nul nul   ` nul nul nul enq nul nul nul
0000120 eot nul nul nul etx nul nul nul dc4 soh nul nul dc4 soh eot  bs
0000140 dc4 soh eot  bs dc3 nul nul nul dc3 nul nul nul eot nul nul nul
0000160 soh nul nul nul soh nul nul nul nul nul nul nul nul nul eot  bs
0000200 nul nul eot  bs  bs eot nul nul  bs eot nul nul enq nul nul nul
0000220 nul dle nul nul soh nul nul nul  bs eot nul nul  bs dc4 eot  bs
0000240  bs dc4 eot  bs nul soh nul nul eot soh nul nul ack nul nul nul
0000260 nul dle nul nul stx nul nul nul dc4 eot nul nul dc4 dc4 eot  bs
0000300 dc4 dc4 eot  bs   H nul nul nul   H nul nul nul ack nul nul nul
0000320 eot nul nul nul eot nul nul nul   ( soh nul nul   ( soh eot  bs
0000340   ( soh eot  bs  sp nul nul nul  sp nul nul nul eot nul nul nul
0000360 eot nul nul nul   Q   e   t   d nul nul nul nul nul nul nul nul
0000400 nul nul nul nul nul nul nul nul nul nul nul nul ack nul nul nul
0000420 eot nul nul nul   /   l   i   b   /   l   d   -   l   i   n   u
0000440   x   .   s   o   .   2 nul nul eot nul nul nul dle nul nul nul
0000460 soh nul nul nul   G   N   U nul nul nul nul nul stx nul nul nul
0000500 stx nul nul nul enq nul nul nul etx nul nul nul ack nul nul nul
0000520 enq nul nul nul soh nul nul nul etx nul nul nul nul nul nul nul
0000540 nul nul nul nul nul nul nul nul stx nul nul nul nul nul nul nul
...................................................................
```

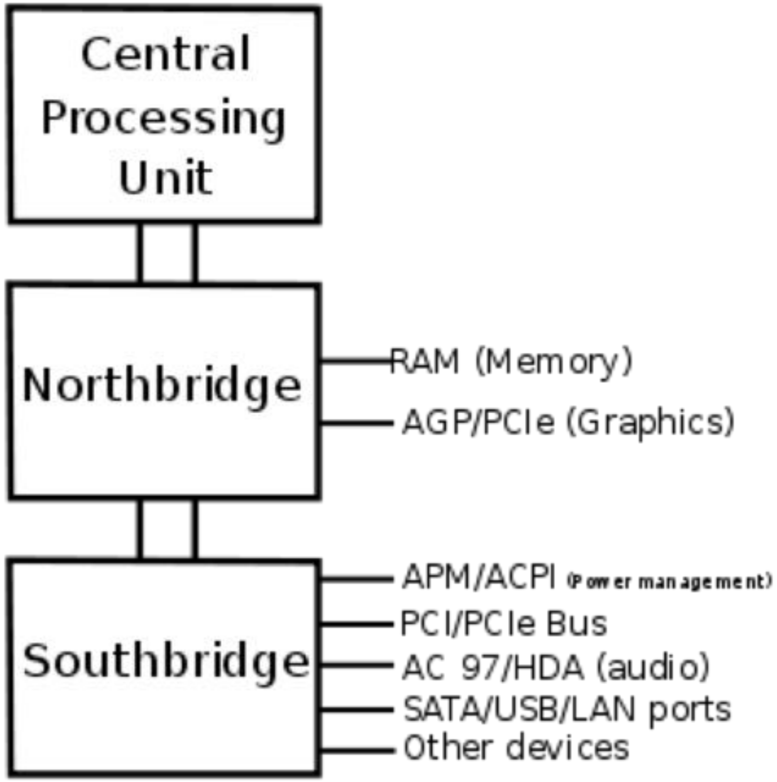
od –a hello

Finally…

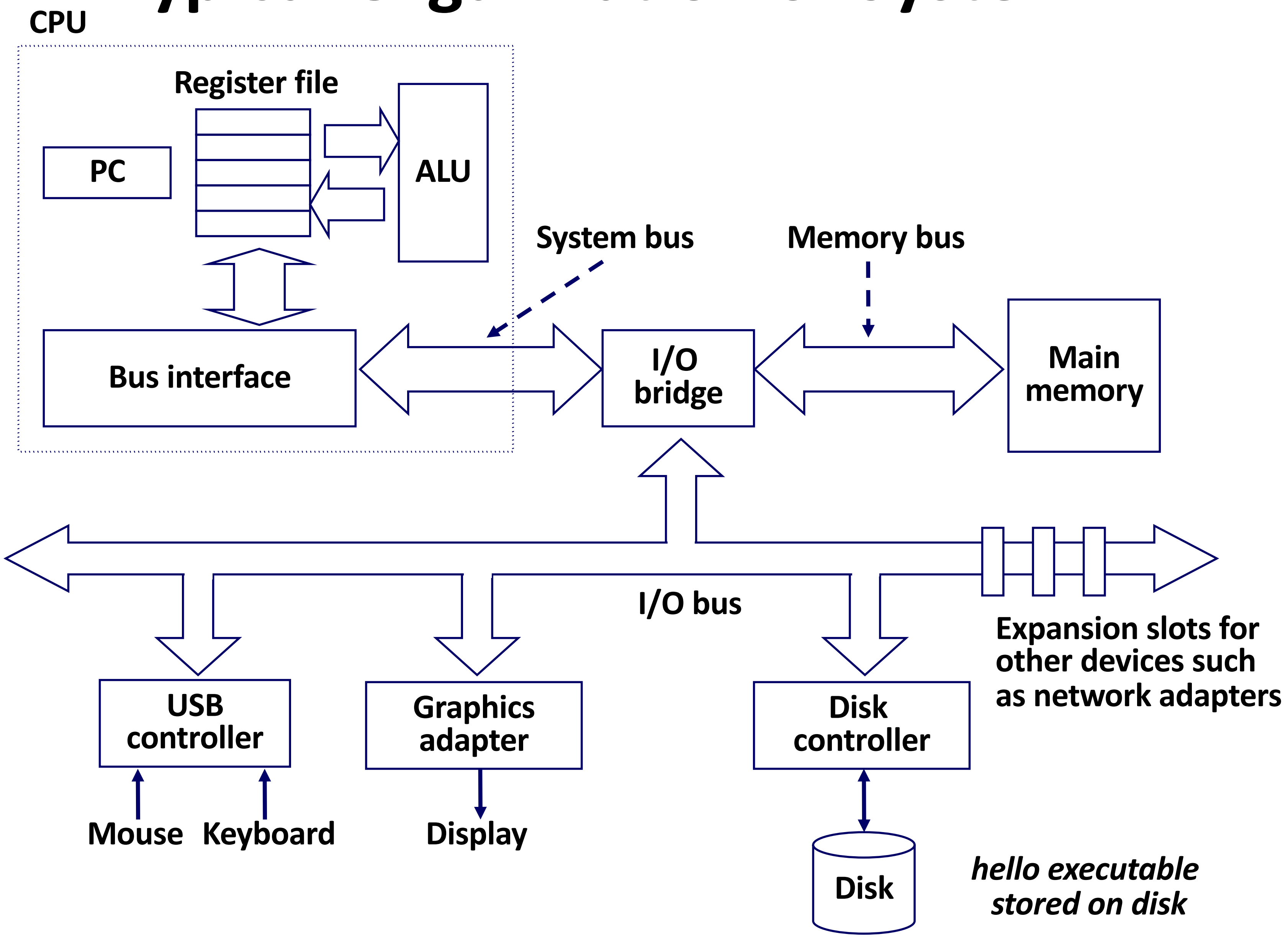
```
$ gcc hello.o -o hello
$ ./hello
Hello World$
```

How do you say "Hello World"?

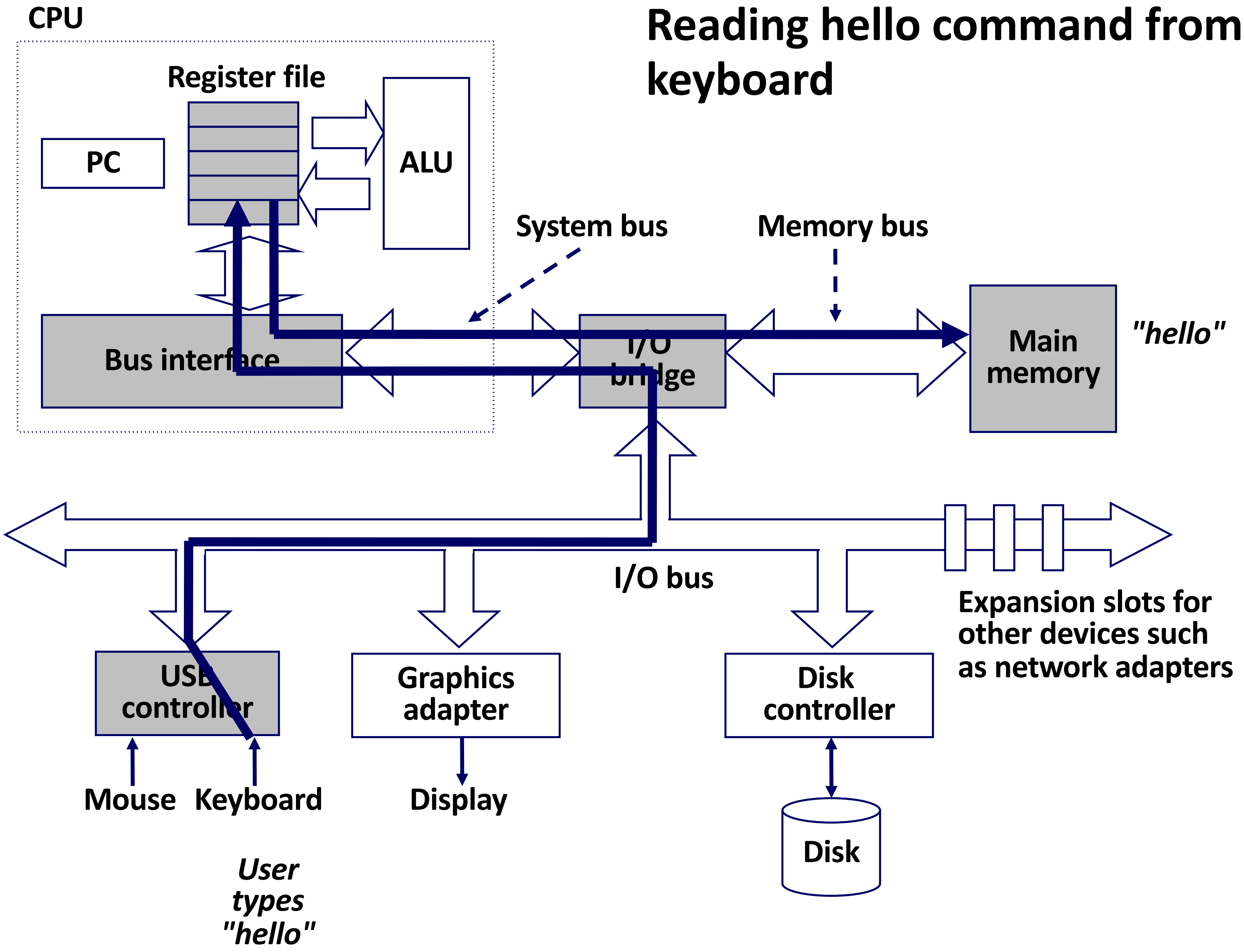




Typical Organization of System



Reading hello command from keyboard



Today: Bits, Bytes, and Integers

- **Representing information as bits**
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

For example, can count in binary

- **Base 2 Number Representation**
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
  - Represent $1.5213 \text{ X } 10^4$ as $1.1101101101101_2 \text{ X } 2^{13}$

Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b
        - » Convert in 4-bit groups

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

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
pointer	4	8	8

Today: Bits, Bytes, and Integers

- Representing information as bits
- **Bit-level manipulations**
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  --------     --------     --------     --------
  01000001     01111101     00111100     10101010
```

- **All of the Properties of Boolean Algebra Apply**

Example: Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - 01101001 { 0, 3, 5, 6 }
    - *76543210*
    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- **Operations**
  - & Intersection 01000001 { 0, 6 }
  - | Union 01111101 { 0, 2, 3, 4, 5, 6 }
  - ^ Symmetric difference 00111100 { 2, 3, 4, 5 }
  - ~ Complement 10101010 { 1, 3, 5, 7 }

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`
    - `~01000001`$_2$ `= 10111110`$_2$
  - `~0x00 = 0xFF`
    - `~00000000`$_2$ `= 11111111`$_2$
  - `0x69 & 0x55 = 0x41`
    - `01101001`$_2$ `& 01010101`$_2$ `= 01000001`$_2$
  - `0x69 | 0x55 = 0x7D`
    - `01101001`$_2$ `| 01010101`$_2$ `= 01111101`$_2$

Contrast: 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`
  - `p && *p` (avoids null pointer access due to lazy evaluation)

Contrast: 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`
  - `p && *p` (avoids null pointer access)

Watch out for && vs. & (and || vs. |)…
one of the more common oopsies in
C programming

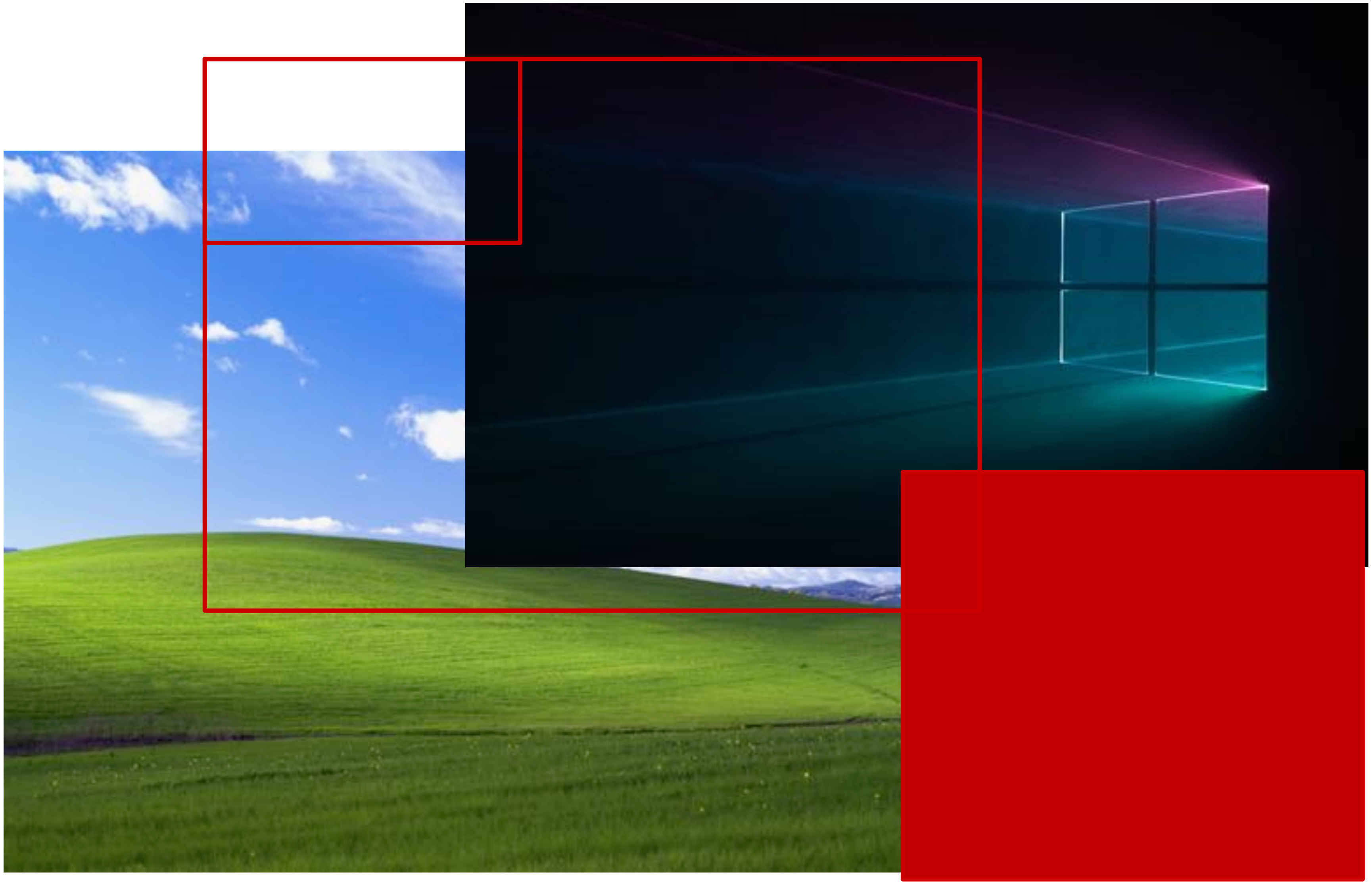
Shift Operations

- **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 most significant bit on right
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

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

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

C, however, has only one right shift operator, >>. Many C compilers choose which right shift to perform depending on what type of integer is being shifted; often signed integers are shifted using the arithmetic shift, and unsigned integers are shifted using the logical shift.



Swap

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

Challenge: Can you code swap function without using a temporary variable?

Hint: Use bitwise XOR (^)

Swapping with Xor

- Bitwise Xor is a form of addition
- With extra property that every value is its own additive inverse

  A ^ A = 0

```
void funny(int *x, int *y)
{
    *x = *x ^ *y;     /* #1 */
    *y = *x ^ *y;     /* #2 */
    *x = *x ^ *y;     /* #3 */
}
```

	`*x`	`*y`
Begin	`A`	`B`
1	`A^B`	`B`
2	`A^B`	`(A^B)^B = A`
3	`(A^B)^A = B`	`A`
End	`B`	`A`

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - **Representation: unsigned and signed**
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
- Summary

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

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

Sign Bit

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

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

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

Sign Bit

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

- **C `short` 2 bytes long**

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

- **Sign Bit**
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Two-complement 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**

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0

    000…0
  - *UMax* = $2^w - 1$

    111…1

Numeric Ranges

- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1

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
`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

Unsigned & Signed Numeric Values

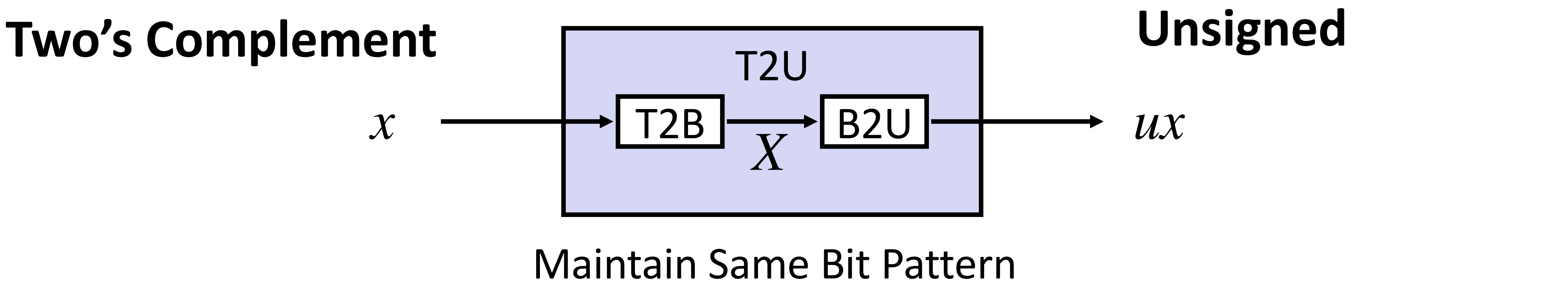
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

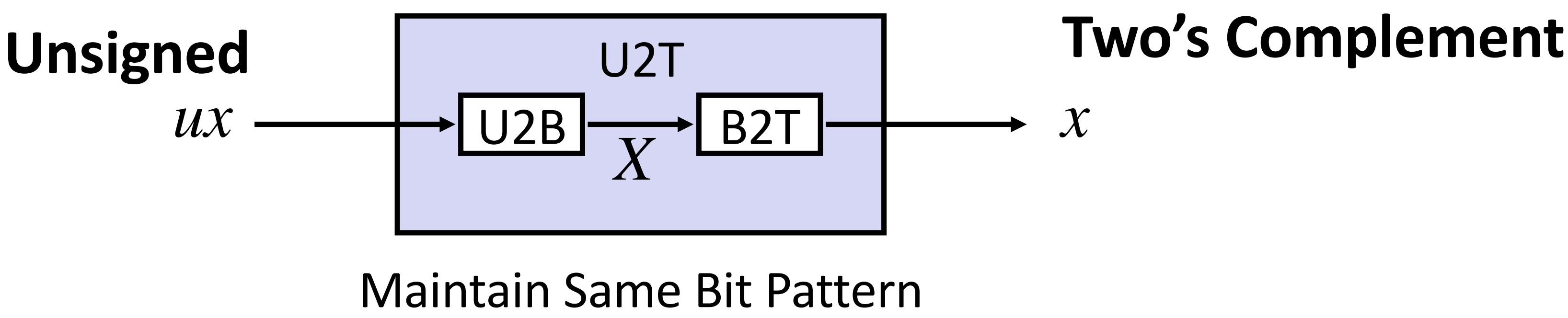
- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **$\Rightarrow$ Can Invert Mappings**
  - U2B(x) = B2U^{-1}(x)
    - Bit pattern for unsigned integer
  - T2B(x) = B2T^{-1}(x)
    - Bit pattern for two's comp integer

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - **Conversion, casting**
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Mapping Between Signed & Unsigned





- **Mappings between unsigned and two's complement numbers: Keep bit representations and reinterpret**

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

→ T2U →

← U2T ←

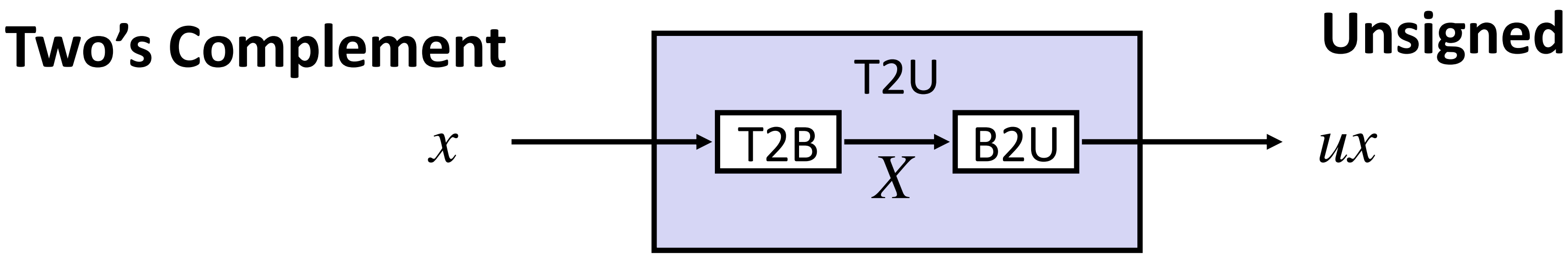
Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

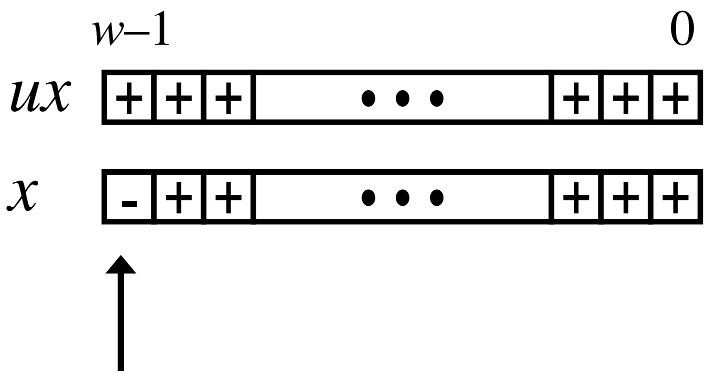
=

+/- 16

Relation between Signed & Unsigned



Maintain Same Bit Pattern



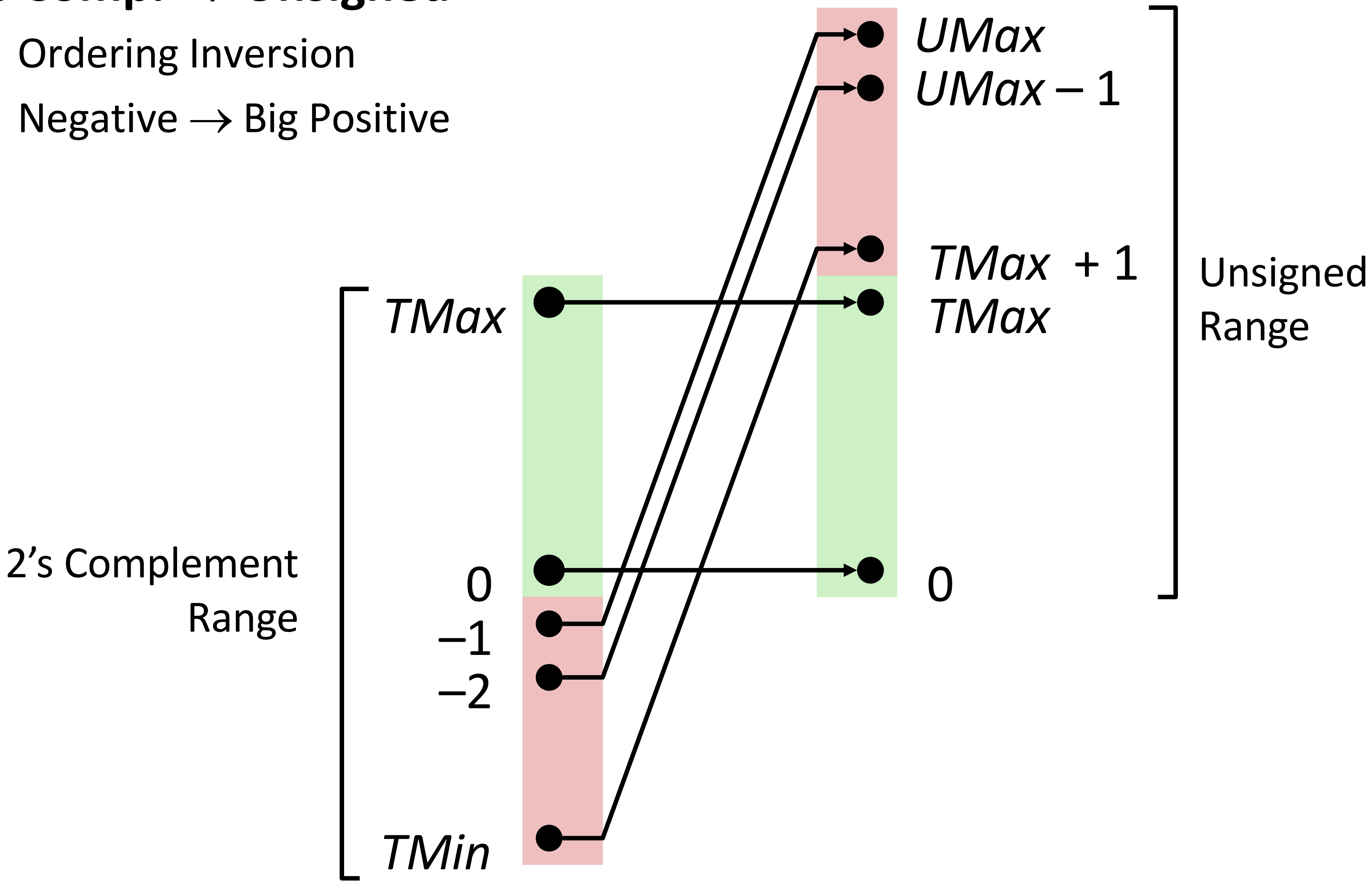
Large negative weight

becomes

Large positive weight

Conversion Visualized

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



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 same as U2T and T2U

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

  - Implicit casting also occurs via assignments and procedure 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**

Summary
Casting Signed ↔ Unsigned: Basic Rules

- **Bit pattern is maintained**
- **But reinterpreted**
- **Can have unexpected effects: adding or subtracting 2^w**

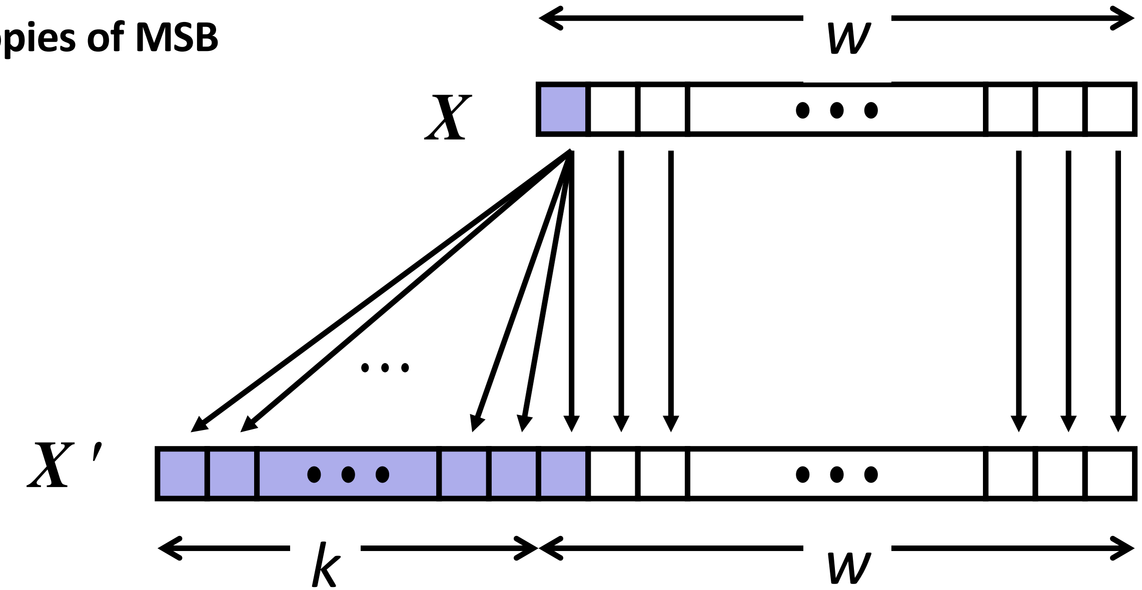
- **Expression containing signed and unsigned int**
  - `int` is cast to `unsigned`!!

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - **Expanding, truncating**
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

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:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



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

Summary: Expanding, Truncating: Basic Rules

- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

Today: Bits, Bytes, and Integers

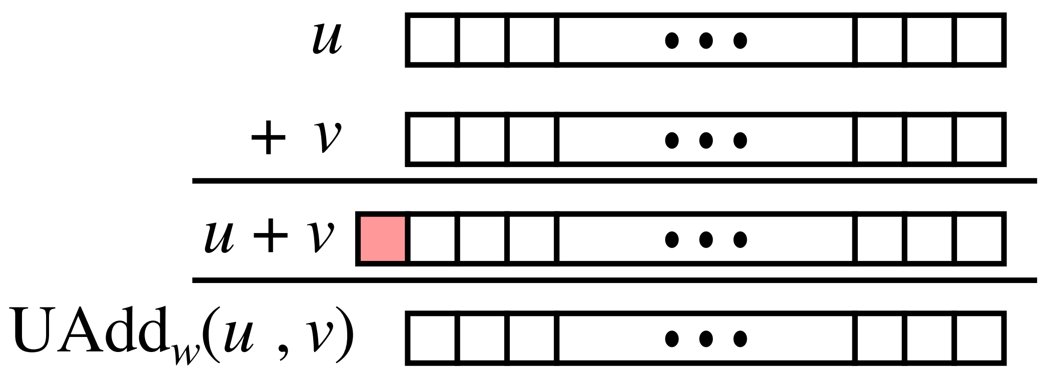
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**

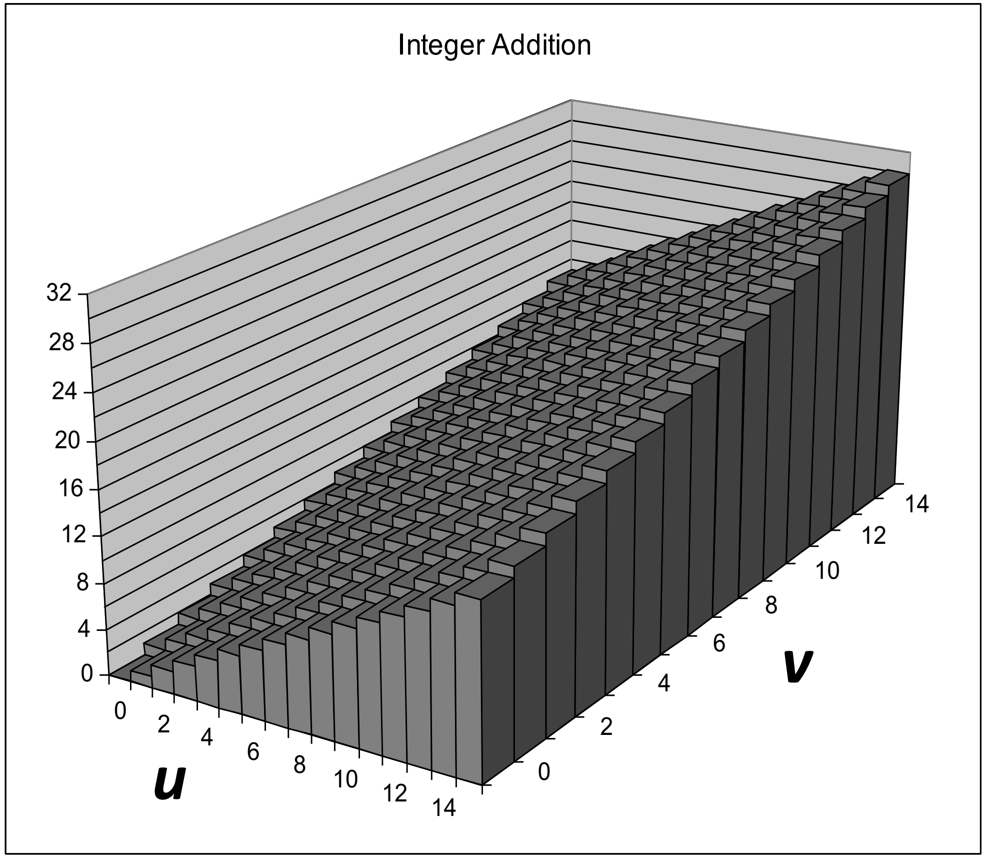
$s \quad = \quad \text{UAdd}_w(u\ ,\ v) \quad = \quad u + v \mod 2^w$

$$UAdd_w(u,v) \ = \ \begin{cases} u+v & u+v < 2^w \\ u+v-2^w & u+v \geq 2^w \end{cases}$$

Visualizing (Mathematical) Integer Addition

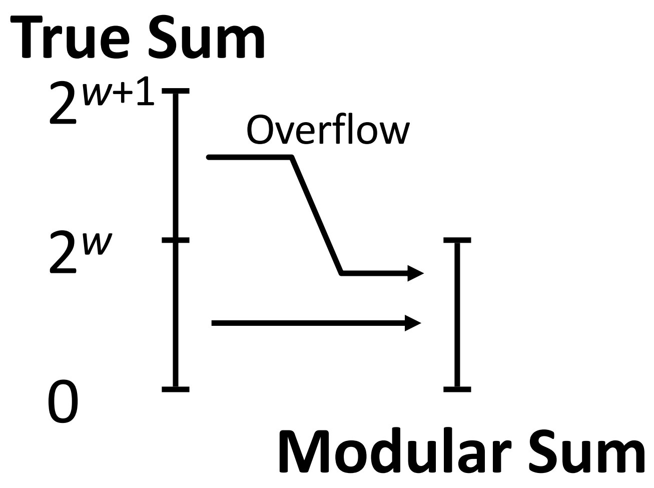
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

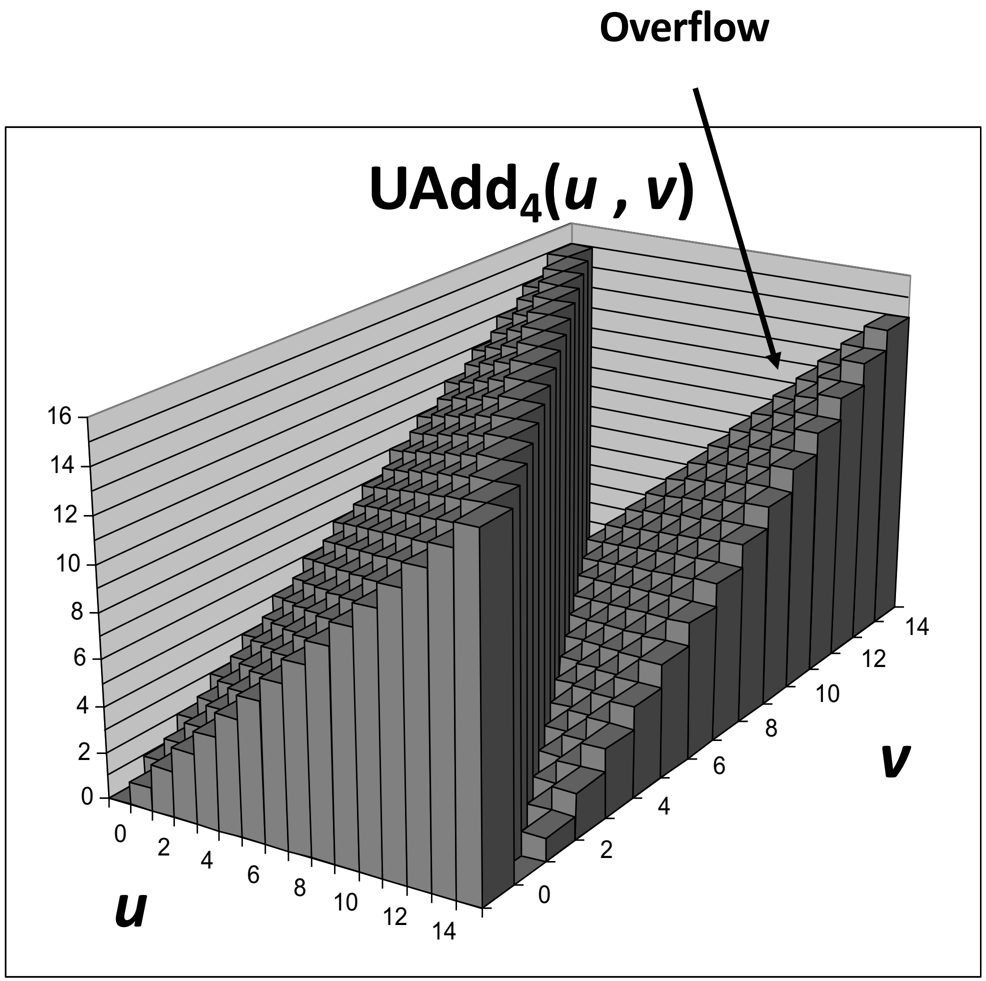
$\text{Add}_4(u, v)$



Visualizing Unsigned Addition

- **Wraps Around**
  - If true sum ≥ 2^w
  - At most once



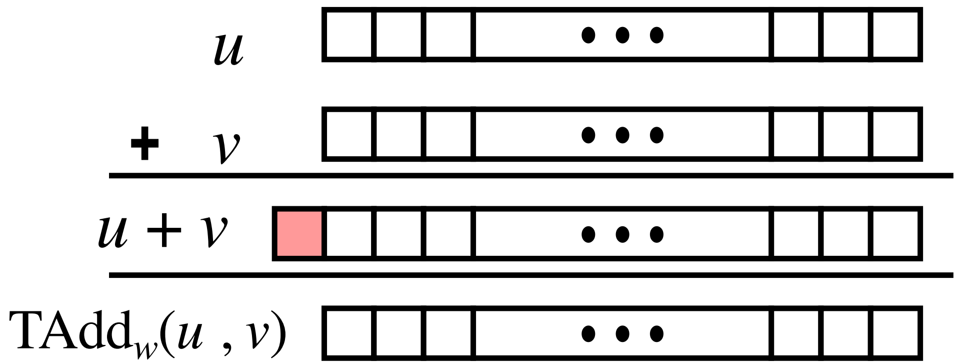


Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



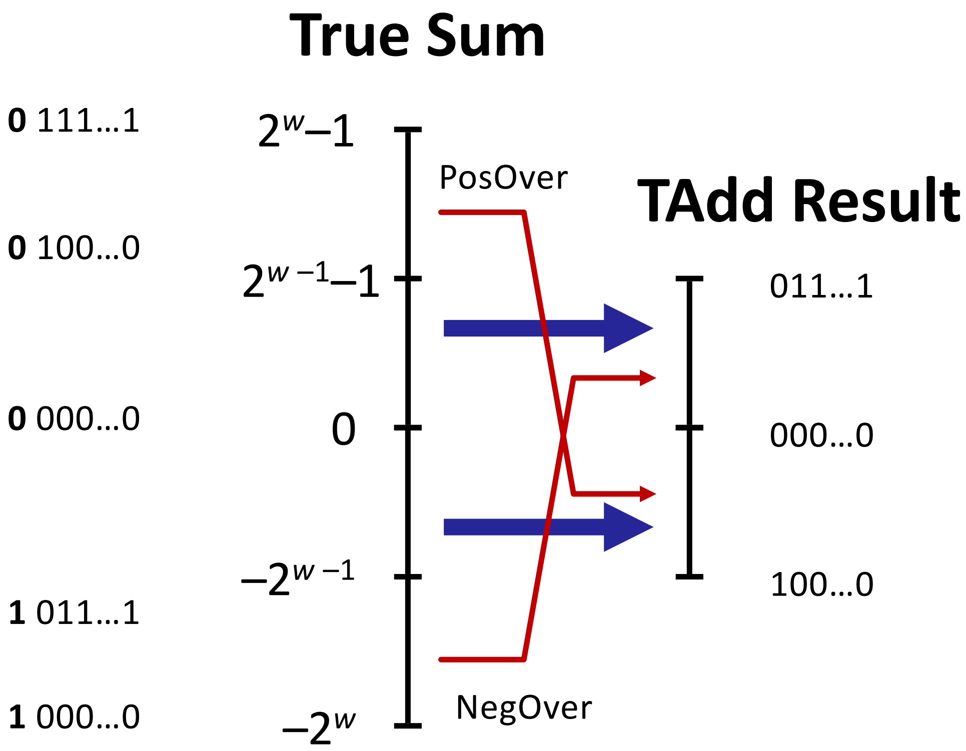
- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

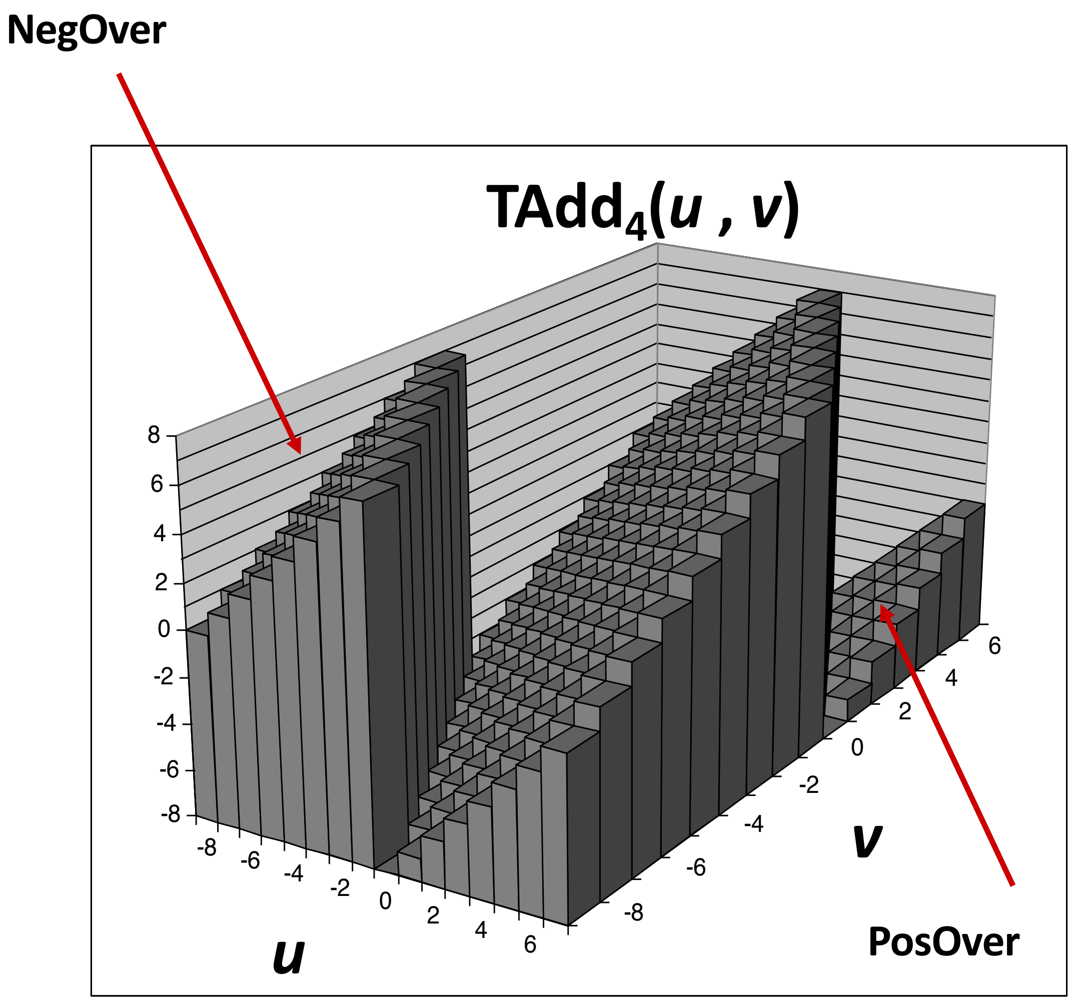
TAdd Overflow

- **Functionality**
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



Visualizing 2's Complement Addition

- **Values**
  - 4-bit two's comp.
  - Range from -8 to +7
- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once



Multiplication

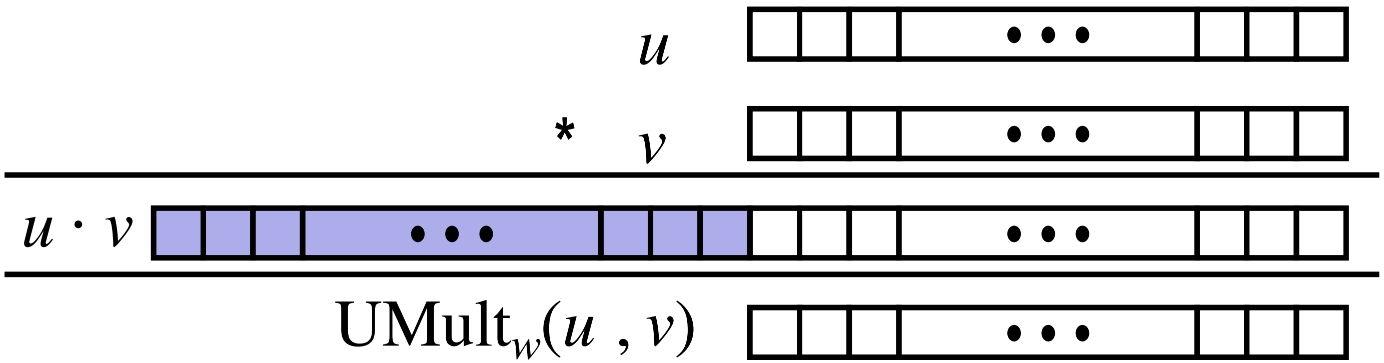
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - Unsigned: up to 2*w* bits
    - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
- **So, maintaining exact results…**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- **Standard Multiplication Function**
  - Ignores high order *w* bits
- **Implements Modular Arithmetic**

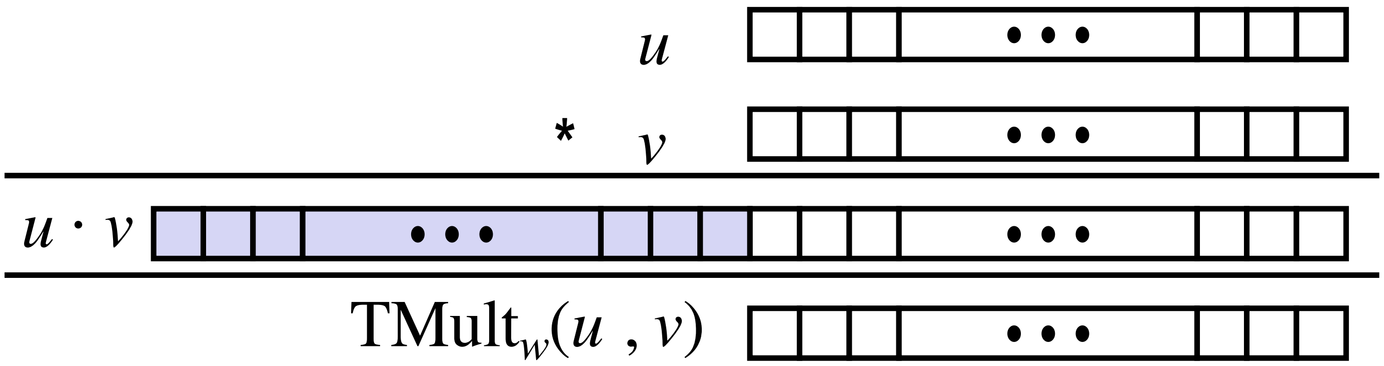
  $\mathrm{UMult}_w(u\ ,\ v) = u \cdot v \bmod 2^w$

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- **Standard Multiplication Function**
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

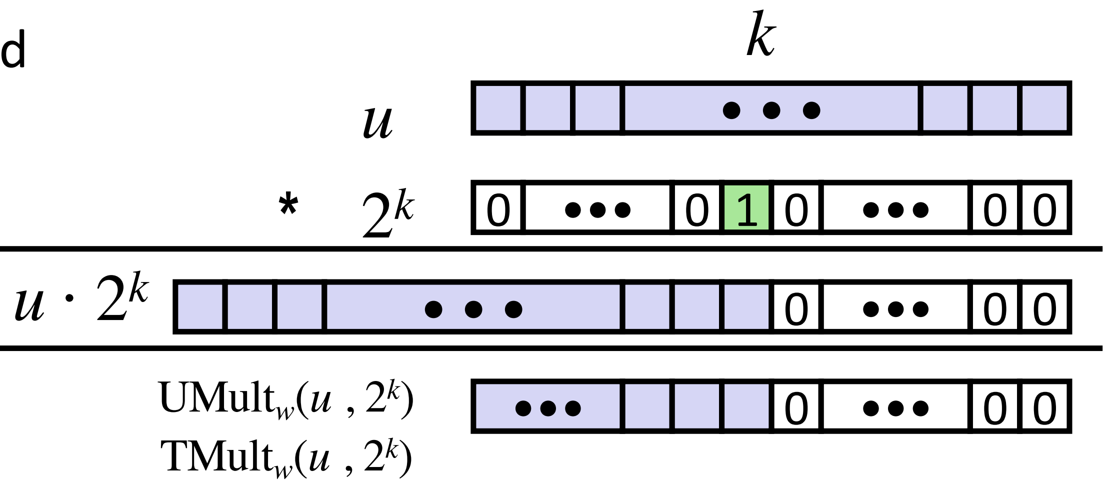
Power-of-2 Multiply with Shift

- **Operation**
  - **`u << k`** gives **`u * 2`^{k}**
  - Both signed and unsigned

Operands: w bits

True Product: $w+k$ bits

Discard k bits: w bits



- **Examples**
  - **`u << 3 == u * 8`**
  - **`(u << 5) - (u << 3) == u * 24`**
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

```
leaq (%rax,%rax,2), %rax
salq $2, %rax
```

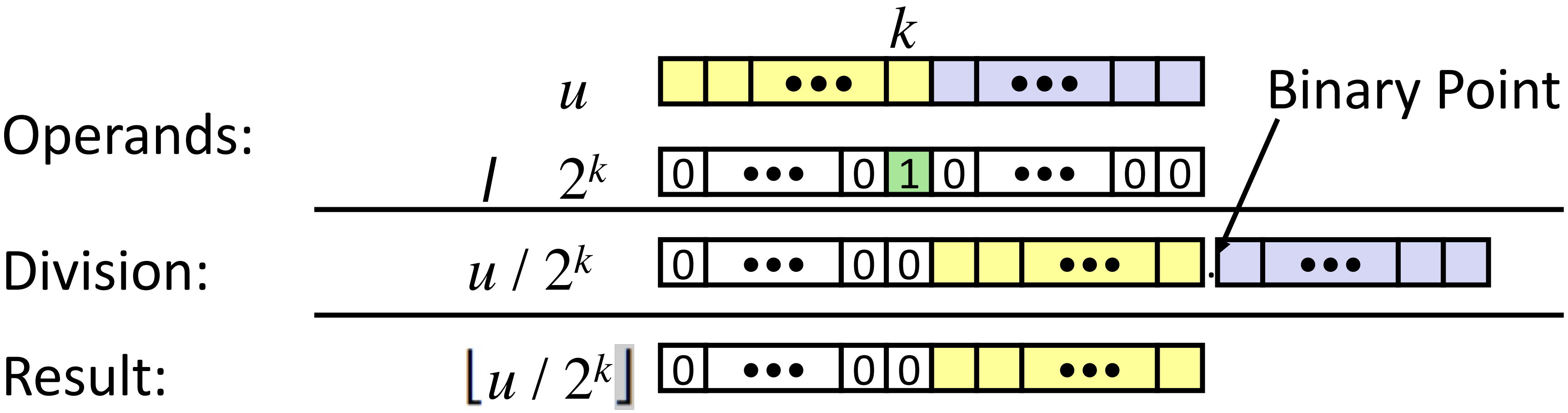
Explanation

```
t <- x+x*2
return t << 2;
```

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

Unsigned Power-of-2 Divide with Shift

- **Quotient of Unsigned by Power of 2**
  - `u >> 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`

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

```
  shrq $3, %rax
```

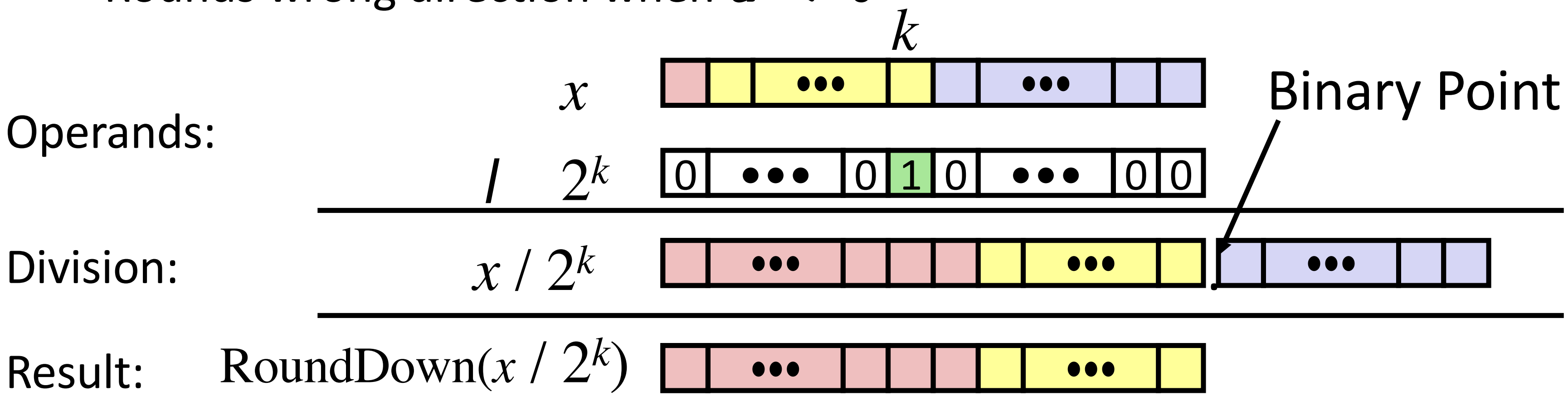
Explanation

```
  # Logical shift
  return x >> 3;
```

- **Uses logical shift for unsigned**
- **For Java Users**
  - Logical shift written as >>>

Signed Power-of-2 Divide with Shift

- **Quotient of Signed by Power of 2**
  - `x >> k` gives $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when `u < 0`

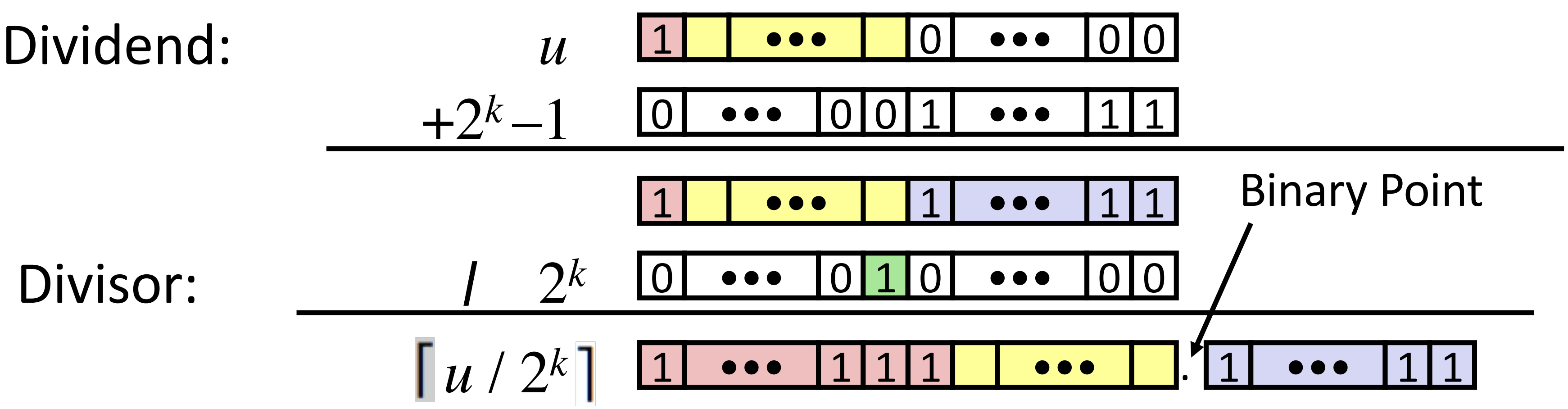


	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

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

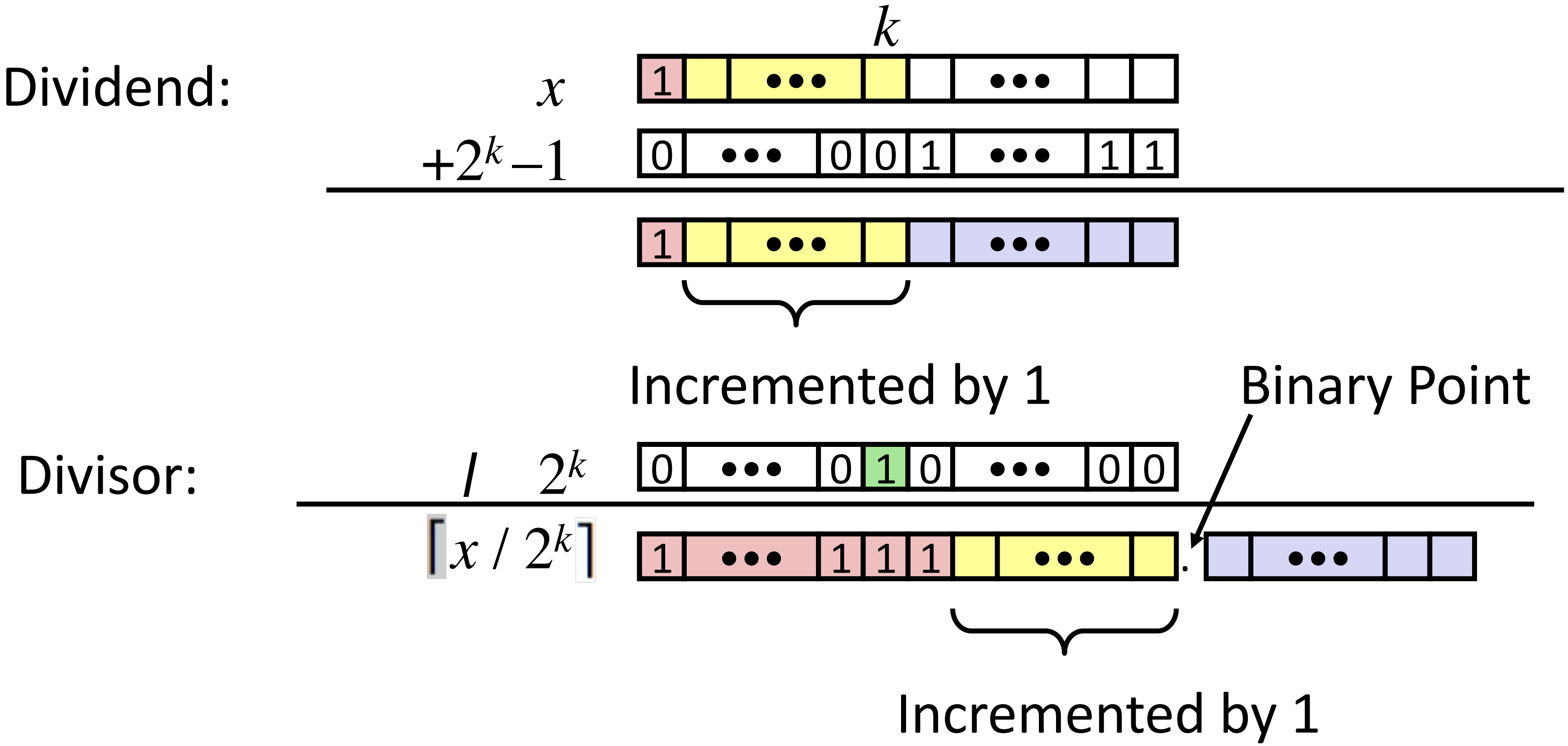
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Compiled Signed Division Code

C Function

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

Compiled Arithmetic Operations

```
  testq %rax, %rax
  js   L4
L3:
  sarq $3, %rax
  ret
L4:
  addq $7, %rax
  jmp  L3
```

Explanation

```
  if x < 0
    x += 7;
  # Arithmetic shift
  return x >> 3;
```

- **Uses arithmetic shift for int**
- **For Java Users**
  - Arith. shift written as >>

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - **Summary**
- Representations in memory, pointers, strings

Arithmetic: Basic Rules

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)

Multiplication

- **Computing Exact Product of w-bit numbers x, y**
  - Either signed or unsigned
- **Ranges**
  - Unsigned: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2w$ bits
  - Two's complement min: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w-1$ bits
  - Two's complement max: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ bits, but only for $(TMin_w)^2$
- **Maintaining Exact Results**
  - Would need to keep expanding word size with each product computed
  - Done in software by "arbitrary precision" arithmetic packages

Buggy code - 1

```
float sum_elements(float a[], unsigned length){
  int i;
  float result=0;

  for(i=0;i<=length-1; i++)
        result+= a[i];
  return result;
}
```

- What's the bug?
- When will it return an error or produce a wrong result?
- How can you fix it?

Buggy code - 2

```
/* Prototype for library function strlen */
size_t strlen(const char *s)

/* whether string s is longer than string t */
int strlonger(char *s, char *t){
  return stlen(s)-strlen(t) >0;
}
```

- What's the bug?
- When will it return an error or produce a wrong result?
- How can you fix it?

Why Should I Use Unsigned?

- ***Don't*** **use without understanding implications**
  - 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)
  . . .
```

Counting Down with Unsigned

- **Proper way to use unsigned as loop index**

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

- **See Robert Seacord, *Secure Coding in C and C++***
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - 0 – 1 → *UMax*

Why Should I Use Unsigned? (cont.)

- ***Do*** **Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic
- ***Do*** **Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension

Mathematical Properties

- **Modular Addition Forms an *Abelian Group***
  - **Closed** under addition

    $0 \leq \mathrm{UAdd}_w(u\,,\,v) \leq 2^w - 1$
  - **Commutative**

    $\mathrm{UAdd}_w(u\,,\,v) \;=\; \mathrm{UAdd}_w(v\,,\,u)$
  - **Associative**

    $\mathrm{UAdd}_w(t,\,\mathrm{UAdd}_w(u\,,\,v)) \;=\; \mathrm{UAdd}_w(\mathrm{UAdd}_w(t,\,u\,),\,v)$
  - **0** is additive identity

    $\mathrm{UAdd}_w(u\,,\,0) \;=\; u$
  - Every element has additive **inverse**
    - Let $\quad \mathrm{UComp}_w(u\,) \;=\; 2^w - u$

      $\mathrm{UAdd}_w(u\,,\,\mathrm{UComp}_w(u\,)) \;=\; 0$

Mathematical Properties of TAdd

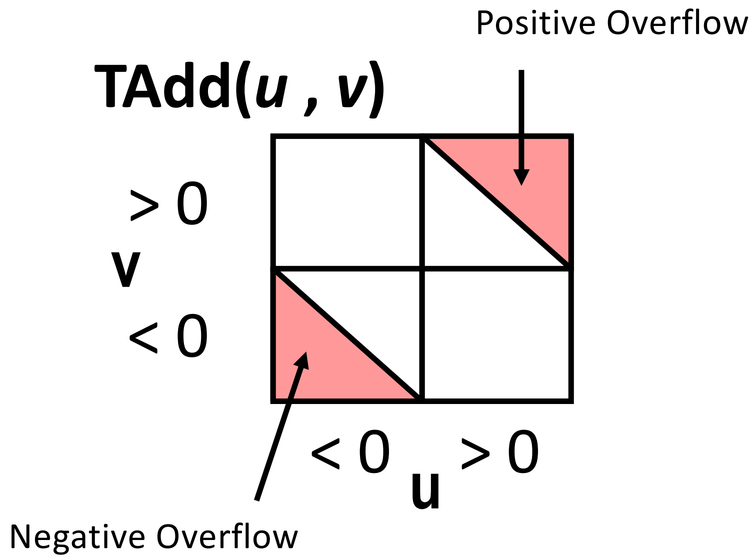
- **Isomorphic Group to unsigneds with UAdd**
  - $TAdd_w(u , v) = U2T(UAdd_w(T2U(u), T2U(v)))$
    - Since both have identical bit patterns

- **Two's Complement Under TAdd Forms a Group**
  - Closed, Commutative, Associative, 0 is additive identity
  - Every element has additive inverse

$$TComp_w(u) \quad = \quad \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

Characterizing TAdd

- **Functionality**
  - True sum requires *w*+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \quad \textbf{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^w & TMax_w < u+v \quad \textbf{(PosOver)} \end{cases}$$

Negation: Complement & Increment

- **Claim: Following Holds for 2's Complement**

  `~x + 1 == -x`

- **Complement**
  - Observation: `~x + x == 1111…111 == -1`

	x	1	0	0	1	1	1	0	1
+	~x	0	1	1	0	0	0	1	0
	-1	1	1	1	1	1	1	1	1

Complement & Increment 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

x = 0

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

Arithmetic: Basic Rules

- **Unsigned ints, 2's complement ints are isomorphic rings: isomorphism = casting**

- **Left shift**
  - Unsigned/signed: multiplication by 2^k
  - Always logical shift

- **Right shift**
  - Unsigned: logical shift, div (division + round to zero) by 2^k
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by 2^k
    - Negative numbers: div (division + round away from zero) by 2^k
      Use biasing to fix

Properties of Unsigned Arithmetic

- **Unsigned Multiplication with Addition Forms Commutative Ring**
  - Addition is commutative group
  - Closed under multiplication

    $0 \leq \mathrm{UMult}_w(u\ , v) \leq 2^w - 1$
  - Multiplication Commutative

    $\mathrm{UMult}_w(u\ , v) = \mathrm{UMult}_w(v\ , u)$
  - Multiplication is Associative

    $\mathrm{UMult}_w(t, \mathrm{UMult}_w(u\ , v)) = \mathrm{UMult}_w(\mathrm{UMult}_w(t, u\ ), v)$
  - 1 is multiplicative identity

    $\mathrm{UMult}_w(u\ , 1) = u$
  - Multiplication distributes over addtion

    $\mathrm{UMult}_w(t, \mathrm{UAdd}_w(u\ , v)) = \mathrm{UAdd}_w(\mathrm{UMult}_w(t, u\ ), \mathrm{UMult}_w(t, v))$

Properties of Two's Comp. Arithmetic

- **Isomorphic Algebras**
  - Unsigned multiplication and addition
    - Truncating to *w* bits
  - Two's complement multiplication and addition
    - Truncating to *w* bits
- **Both Form Rings**
  - Isomorphic to ring of integers mod 2^w
- **Comparison to (Mathematical) Integer Arithmetic**
  - Both are rings
  - Integers obey ordering properties, e.g.,

    $u > 0 \quad \Rightarrow \quad u + v > v$

    $u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0$
  - These properties are not obeyed by two's comp. arithmetic

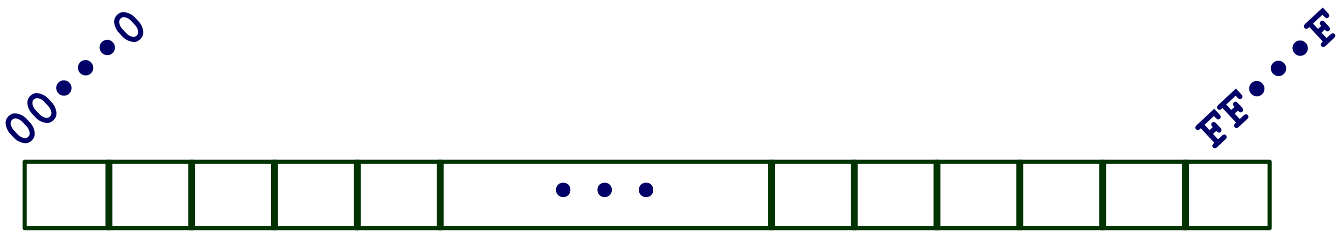
    TMax `+ 1` `==` *TMin*

    `15213 * 30426 == -10030` (16-bit words)

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- **Representations in memory, pointers, strings**

Byte-Oriented Memory Organization



- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

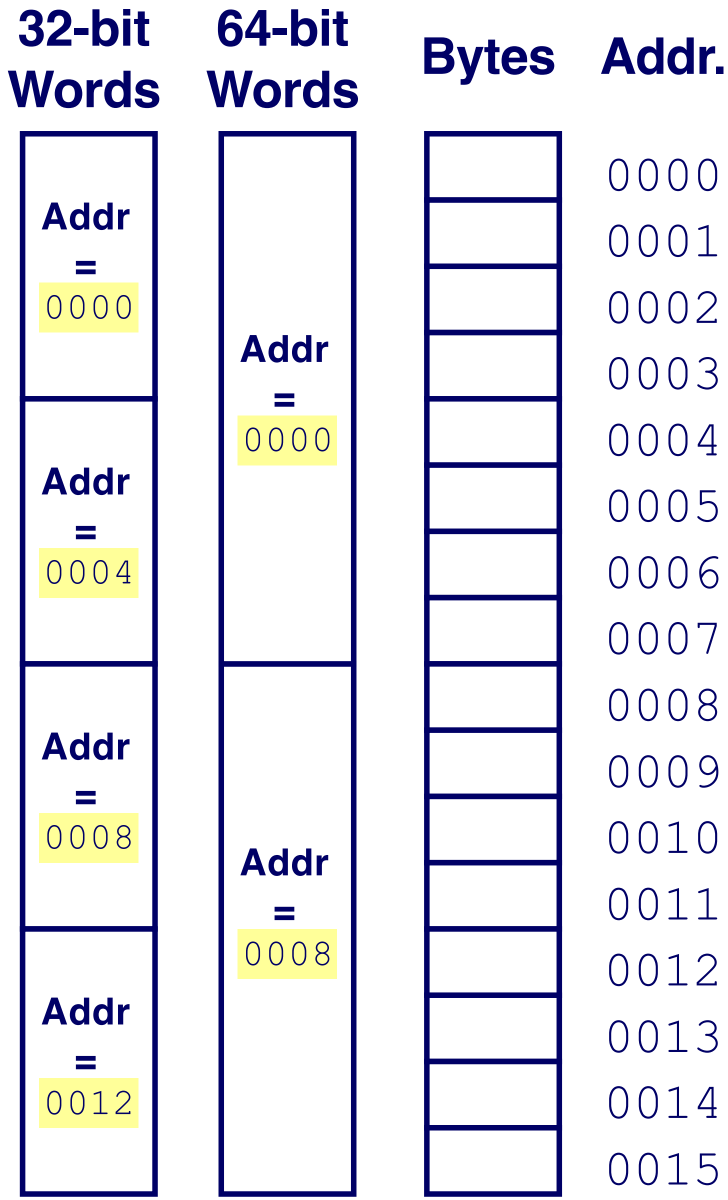
- **Note: system provides private address spaces to each "process"**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine Words

- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 PB (petabytes) of addressable memory
    - That's 18.4 X 10^{15}
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

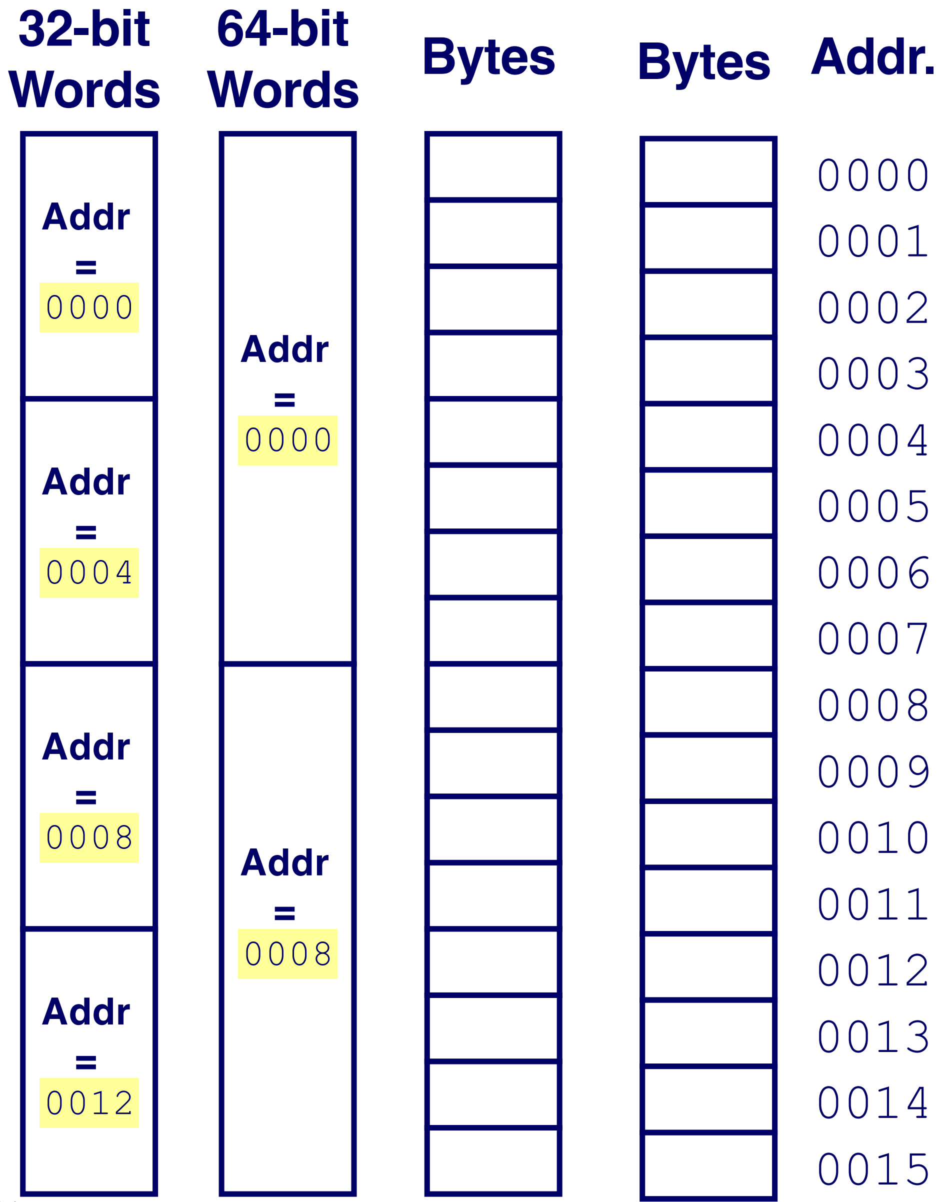


Example Data Representations

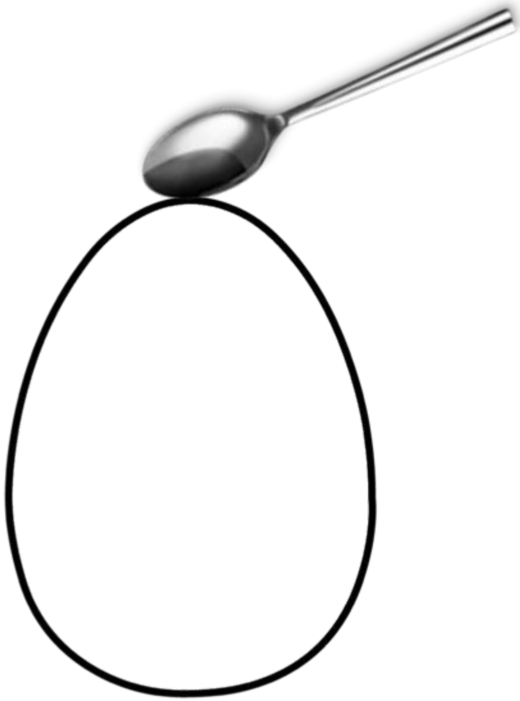
C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
pointer	4	8	8

Byte Ordering

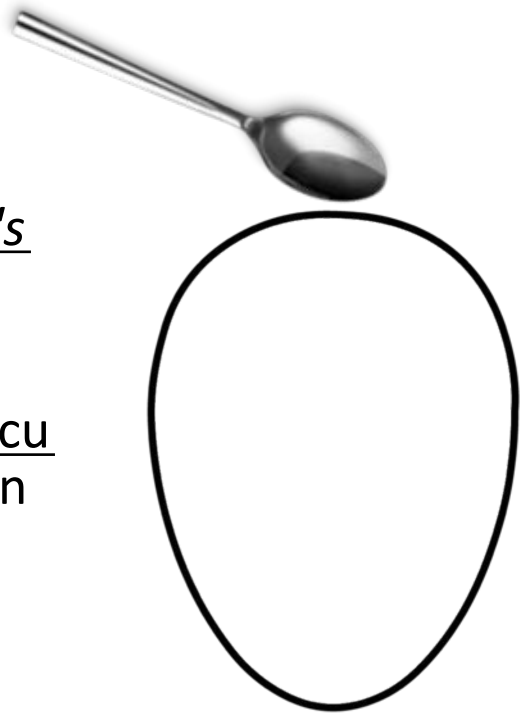
- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address



The origin of the word Endian from Wikipedia



- *Big-endian* and *little-endian* were taken from Jonathan Swift's satiric novel *Gulliver's Travels*.
- Gulliver finds two groups of people in Lilliput and Blefuscu conflict over which end of an egg to crack.



Byte Ordering Example

- **Example**
  - Variable **`x`** has 4-byte value of **`0x01234567`**
  - Address given by **`&x`** is **`0x100`**

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

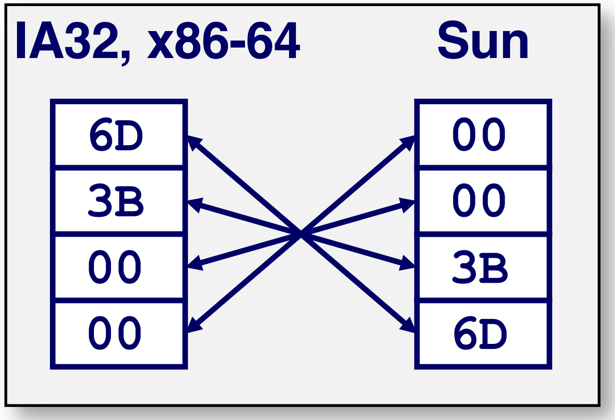
Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

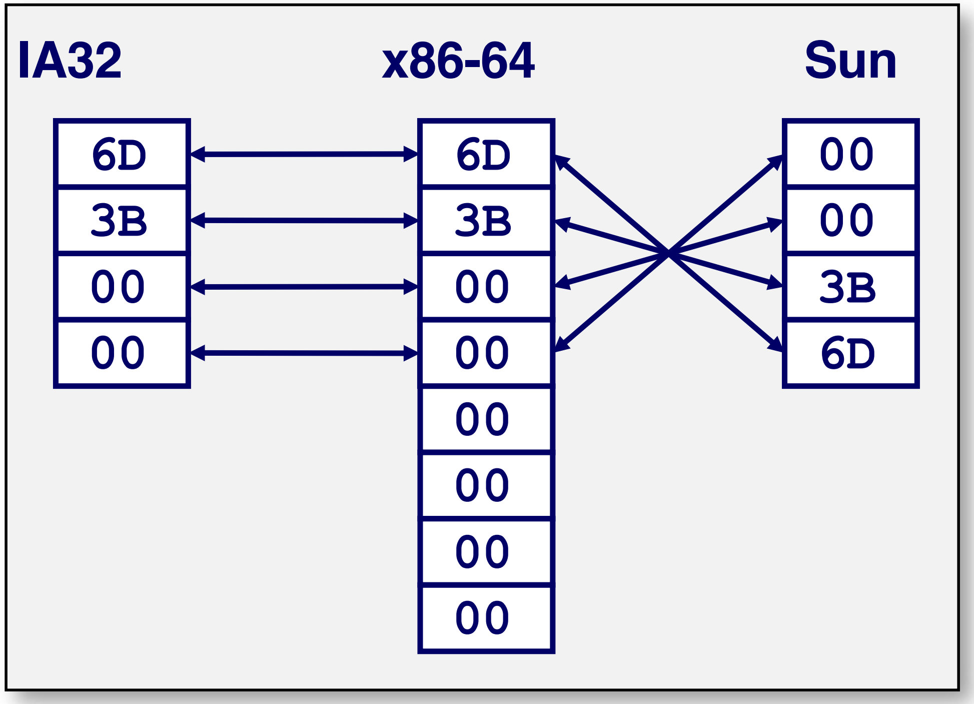
Representing Integers

Decimal:	15213			
Binary:	0011	1011	0110	1101
Hex:	3	B	6	D

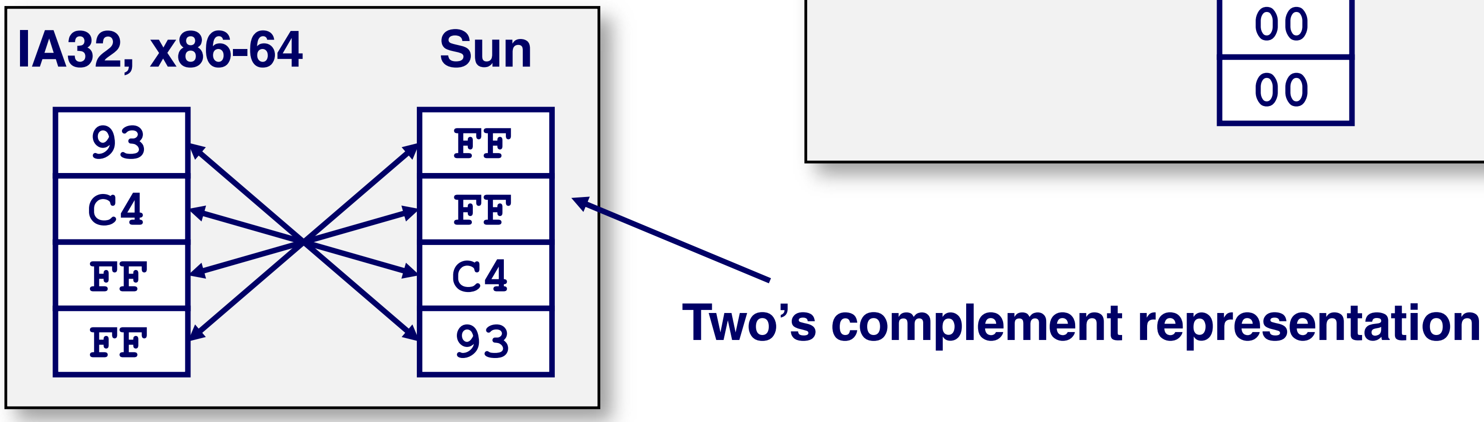
`int A = 15213;`



`long int C = 15213;`



`int B = -15213;`



Examining Data Representations

- **Code to Print Byte Representation of Data**
  - Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n",start+i, start[i]);
  printf("\n");
}
```

Printf directives:

%p: Print pointer
%x: Print Hexadecimal

`show_bytes` Execution Example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 15213;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

Representing Pointers

```
int B = -15213;
int *P = &B;
```

Sun	IA32	x86-64
EF	AC	3C
FF	28	1B
FB	F5	FE
2C	FF	82
		FD
		7F
		00
		00

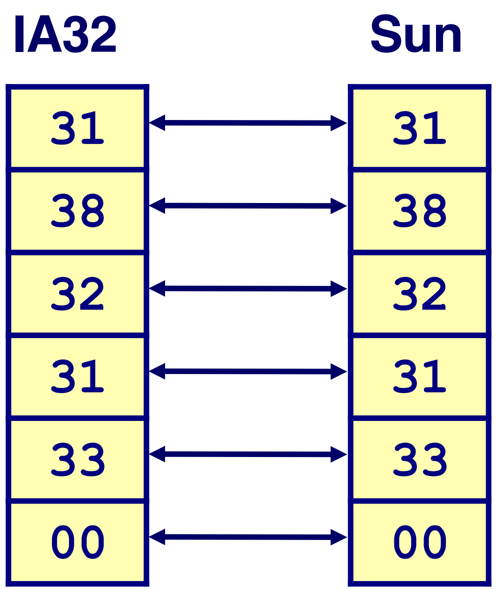
Different compilers & machines assign different locations to objects

Even get different results each time run program

Representing Strings

```
char S[6] = "18213";
```

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - – Digit *i* has code 0x30+*i*
  - String should be null-terminated
    - Final character = 0
- **Compatibility**
  - Byte ordering not an issue



Integer C Puzzles

True or False?

- `x < 0` ⇒ `((x*2) < 0)`
- `ux >= 0`
- `x & 7 == 7` ⇒ `(x<<30) < 0`
- `ux > -1`
- `x > y` ⇒ `-x < -y`
- `x * x >= 0`
- `x > 0 && y > 0` ⇒ `x + y > 0`
- `x >= 0` ⇒ `-x <= 0`
- `x <= 0` ⇒ `-x >= 0`
- `(x|-x)>>31 == -1`
- `ux >> 3 == ux/8`
- `x >> 3 == x/8`
- `x & (x-1) != 0`

Initialization

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

Integer C Puzzles

- `x < 0` ⇒ `((x*2) < 0)`

True or False?

Initialization

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

Integer C Puzzles

- `ux >= 0`

True or False?

Initialization

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

Integer C Puzzles

- `x & 7 == 7` ⇒ `(x<<30) < 0`

True or False?

Initialization

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

Integer C Puzzles

- `ux > -1`

True or False?

Initialization

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

Integer C Puzzles

- `x > y` $\Rightarrow$ `-x < -y`

True or False?

Initialization

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

Integer C Puzzles

- `x * x >= 0`

True or False?

Initialization

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

Integer C Puzzles

- `x > 0 && y > 0` ⇒ `x + y > 0`

True or False?

Initialization

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

Integer C Puzzles

- `x >= 0` ⇒ `-x <= 0`

True or False?

Initialization

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

Integer C Puzzles

- `x <= 0` ⇒ `-x >= 0`

True or False?

Initialization

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

Integer C Puzzles

- `(x|-x)>>31 == -1`

True or False?

Initialization

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

Integer C Puzzles

- `ux >> 3 == ux/8`

True or False?

Initialization

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

Integer C Puzzles

- `x >> 3 == x/8`

True or False?

Initialization

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

Integer C Puzzles

- `x & (x-1) != 0`

True or False?

Initialization

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

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- **Similar to code found in FreeBSD's implementation of getpeername**
- **There are legions of smart people trying to find vulnerabilities in programs**

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Malicious Usage

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

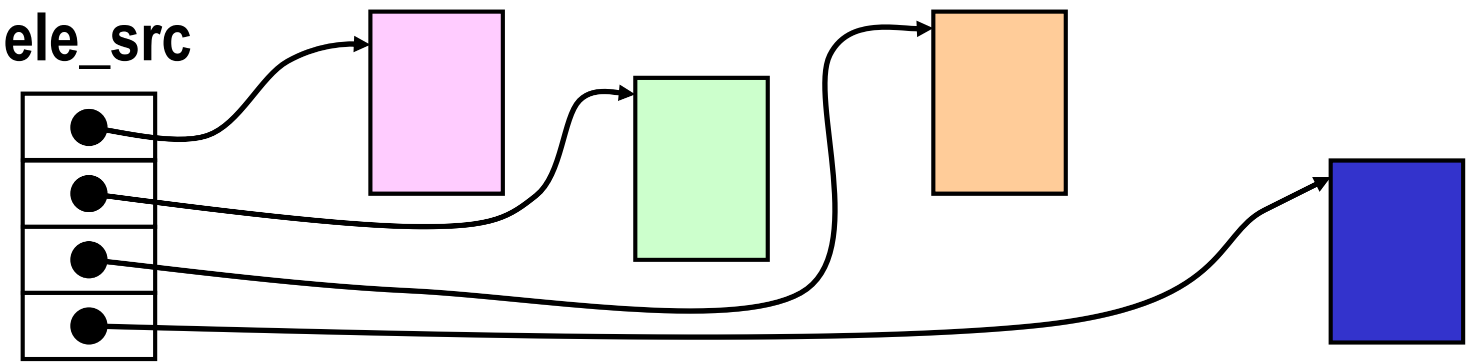
```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

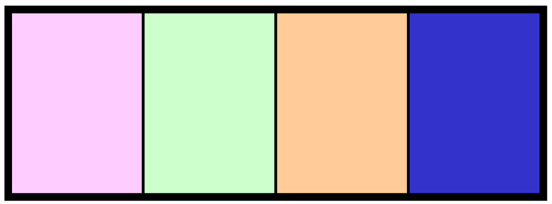
Code Security Example #2

- **SUN XDR library**
  - Widely used library for transferring data between machines

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```



malloc(ele_cnt * ele_size)



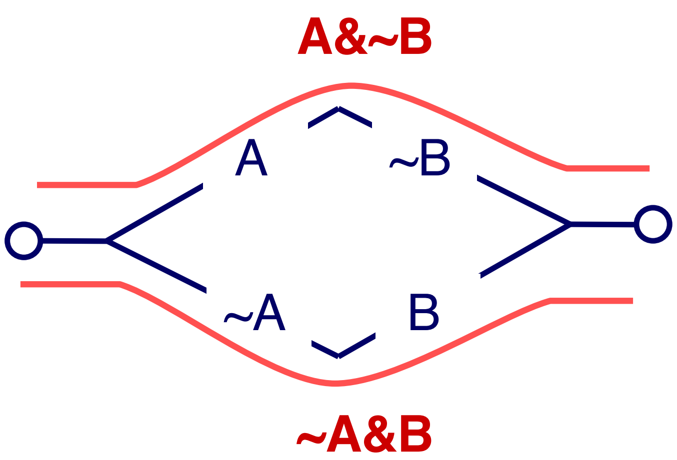
XDR Code

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

Left-over slides

Application of Boolean Algebra

- **Applied to Digital Systems by Claude Shannon**
  - 1937 MIT Master's Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0



Connection when

A&~B | ~A&B

= A^B

Binary Number Property

Claim

$$1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1} = 2^w$$

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

- **w = 0:**
  - $1 = 2^0$
- **Assume true for w-1:**
  - $$\underbrace{1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1}}_{= 2^w} + 2^w = 2^w + 2^w = 2^{w+1}$$

XDR Vulnerability

malloc(ele_cnt * ele_size)

- **What if:**
  - `ele_cnt` = $2^{20} + 1$
  - `ele_size` = 4096 = 2^{12}
  - Allocation = ??

- **How can I make this function secure?**

Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- **Example Fragment**

Address	Instruction Code	Assembly Rendition	
`8048365:`	`5b`	`pop`	`%ebx`
`8048366:`	`81 c3 ab 12 00 00`	`add`	`$0x12ab,%ebx`
`804836c:`	`83 bb 28 00 00 00 00`	`cmpl`	`$0x0,0x28(%ebx)`

- **Deciphering Numbers**
  - Value: `0x12ab`
  - Pad to 32 bits: `0x000012ab`
  - Split into bytes: `00 00 12 ab`
  - Reverse: `ab 12 00 00`