Bits, Bytes, and Integers

CSCI3240: Lecture 2 and 3

Dr. Arpan Man Sainju

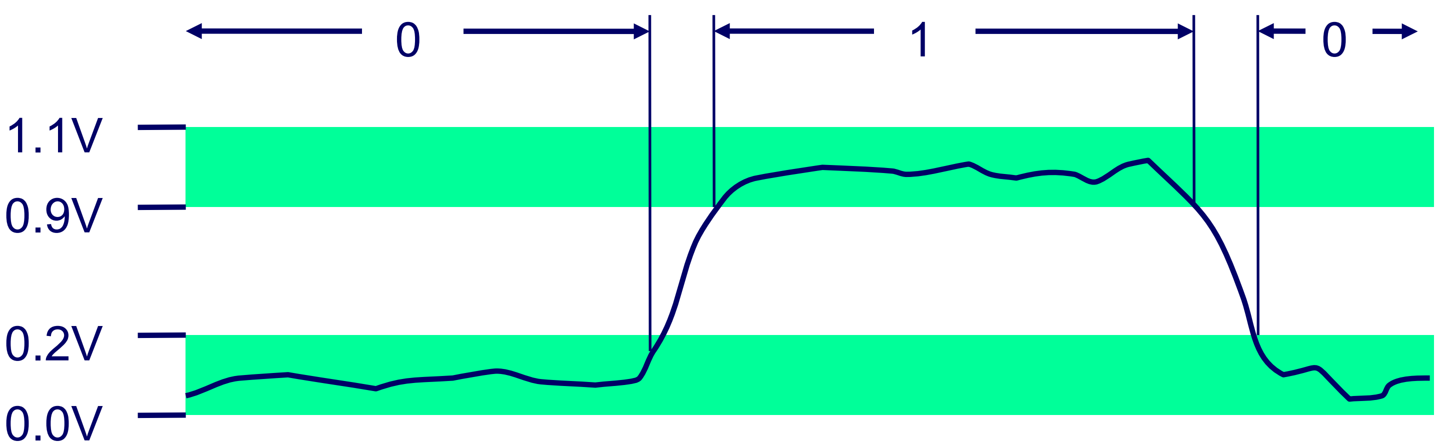
Middle Tennessee State University

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

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

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

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

Number of bytes for each data type in C

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 Operation

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

B

&	0	1
0	0	0
1	0	1

A

Or Operation

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

B

|	0	1
0	0	1
1	1	1

A

Boolean Algebra

- Not and Xor Operations

Not Operation

- ~A = 1 when A=0

~	
0	1
1	0

A

Exclusive-Or (Xor) Operation

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

B

^	0	1
0	0	1
1	1	0

A

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

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

AND

&	0	1
0	0	0
1	0	1

OR

|	0	1
0	0	1
1	1	1

NOT

~	
0	1
1	0

XOR

^	0	1
0	0	1
1	1	0

- All of the Properties of Boolean Algebra Apply

Boolean Properties

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$xx' = 0$	$x + x' = 1$
Commutative Law	$xy = yx$	$x + y = y + x$
Associative Law	$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$
Distributive Law	$x + yz = (x + y)(x + z)$	$x(y + z) = xy + xz$
Absorption Law	$x(x + y) = x$	$x + xy = x$ $x + x'y = x + y$ $x' + xy = x' + y$
DeMorgan's Law	$(xy)' = x' + y'$	$(x + y)' = x'y'$
Double Complement Law	$(x)'' = x$	

Example: Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - N = 01101001 { 0, 3, 5, 6 }
    - *76543210*
    - M= 01010101 { 0, 2, 4, 6 }
    - *76543210*
- Operations

&	Intersection	N&M=	01000001	{ 0, 6 }
|	Union	N|M=	01111101	{ 0, 2, 3, 4, 5, 6 }
^	Symmetric difference	N^M=	00111100	{ 2, 3, 4, 5 }
~	Complement	~M=	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
    - $\sim 01000001_2 \Rightarrow 10111110_2$
  - ~0x00 ⇒ 0xFF
    - $\sim 00000000_2 \Rightarrow 11111111_2$
  - 0x69 & 0x55 ⇒ 0x41
    - $01101001_2 \mathbin{\&} 01010101_2 \Rightarrow 01000001_2$
  - 0x69 | 0x55 ⇒ 0x7D
    - $01101001_2 \mid 01010101_2 \Rightarrow 01111101_2$

```c
char a = 0x69;
char b = 0x55;
char c = a & b;
printf("%x & %x = %x\n",a,b,c);
```

Console Output

```
69 & 55 = 41
```

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)

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
    - View 0 as “False
    - Anything
    - Always
    - Early t
- Examples
  - !0x41 → 0
  - !0x00 → 0
  - !!0x41 →
  - 0x69 && 0x
  - 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 left
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	01100010
<< 3	00010*000*
Log. << 2	*00*011000
Arith. << 2	*00*011000

Argument **x**	10100010
>> 3	11110100
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

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;
```

- 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

- 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
  - $|TMin| \quad = \quad TMax + 1$
    - Asymmetric range
  - $UMax \quad = \quad 2 * TMax + 1$

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

Printf function: format specifier

Format Specifier	Description
d or i	Signed decimal integer
u	Unsigned decimal integer
o	Unsigned octal number
x	Unsinged hexadecimal number
X	Unsinged hexadecimal in uppercase
f	Decimal floating point
c	Character
s	String
a	Hexadecimal floating-point in lowercase
A	Hexadecimal floating point in uppercase
p	Pointer address

Refer: https://www.delftstack.com/howto/c/format-specifier-lu-in-c/

Working with limits in C

MACRO	VALUE	DESCRIPTION
CHAR_BIT	8	Maximum number of bits in a byte
SCHAR_MIN	-128	Minimum value for signed char
SCHAR_MAX	+127	Maximum value for signed char
UCHAR_MAX	225	Maximum value for unsigned char
CHAR_MIN	-128	Minimum value for *char* type
CHAR_MAX	+127	Maximum value for *char* type
MB_LEN_MAX	16	Maximum bytes in multi-byte array
SHRT_MIN	-32768	Minimum value of short int
SHRT_MAX	+32767	Maximum value of short int
USHRT_MAX	65535	Maximum value of unsigned short int
INT_MIN	-2147483648	Minimum value of int
INT_MAX	+2147483647	Maximum value of int
UINT_MAX	4294967295	Maximum value of unsigned int
LONG_MIN	-9223372036854775808	Minimum value of long int
LONG_MAX	+9223372036854775807	Maximum value of long int
ULONG_MAX	18446744073709551615	Maximum value of unsigned long int

Working with limits in C

```
#include <stdio.h>
#include <limits.h>

int main(){
    printf("CHAR_BIT = %i \n",CHAR_BIT);
    printf("SCHAR_MIN = %i \n",SCHAR_MIN); // signed character type min
    printf("SCHAR_MAX = %i \n",SCHAR_MAX); // singned character type max
    printf("UCHAR_MAX = %i \n",UCHAR_MAX); // unsinged char max
    printf("CHAR_MIN = %i \n",CHAR_MIN);
    printf("CHAR_MAX = %i \n",CHAR_MAX);
    printf("SHRT_MIN = %i \n",SHRT_MIN);  //min for short data type
    printf("SHRT_MAX = %i \n",SHRT_MAX);  //max for short data type
    printf("USHRT_MAX = %i \n",USHRT_MAX); //,ax for unsigned short data type
    printf("INT_MIN = %i \n",INT_MIN); // integer min (default is singed)
    printf("INT_MAX = %i \n",CHAR_BIT);     // integer max
    printf("UINT_MAX = %u \n",UINT_MAX);   //unsinged int max
    printf("LONG_MIN = %li \n",LONG_MIN);  //Minimum value of long int
    printf("LONG_MAX = %li \n",LONG_MAX);  //Maximum value of long int
    printf("ULONG_MAX = %lu \n",ULONG_MAX);
    return 0;
}
```

Hint for Project 1

```c
#include <stdio.h>

int main(){
    int counter= 0;
    for (int counter=0;counter<1024; counter+=16)
    {
        // shows 7 digits output with leading 0s
        printf("%07x:\n", counter);
    }
    return 0;
}
```

Console Output

```
0000000:
0000010:
0000020:
0000030:
0000040:
0000050:
0000060:
0000070:
0000080:
0000090:
00000a0:
00000b0:
00000c0:
00000d0:
00000e0:
00000f0:
0000100:
0000110:
0000120:
0000130:
```

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 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 ←

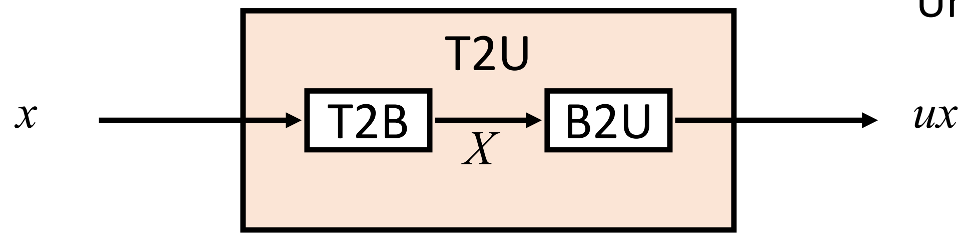
Mapping Signed ↔ Unsigned

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	+/- 16	10
1011	-5	↔	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

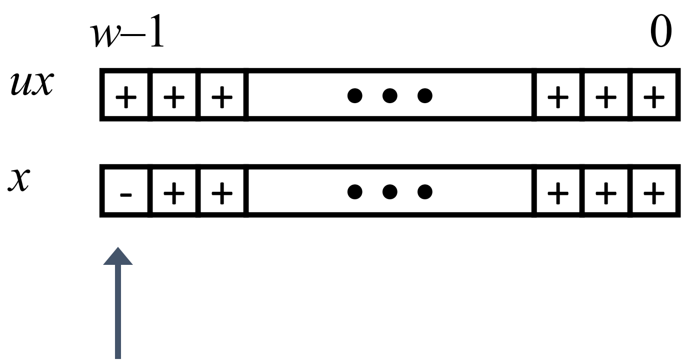
Relation between Signed & Unsigned

Two's Complement

Unsigned



Maintain Same Bit Pattern



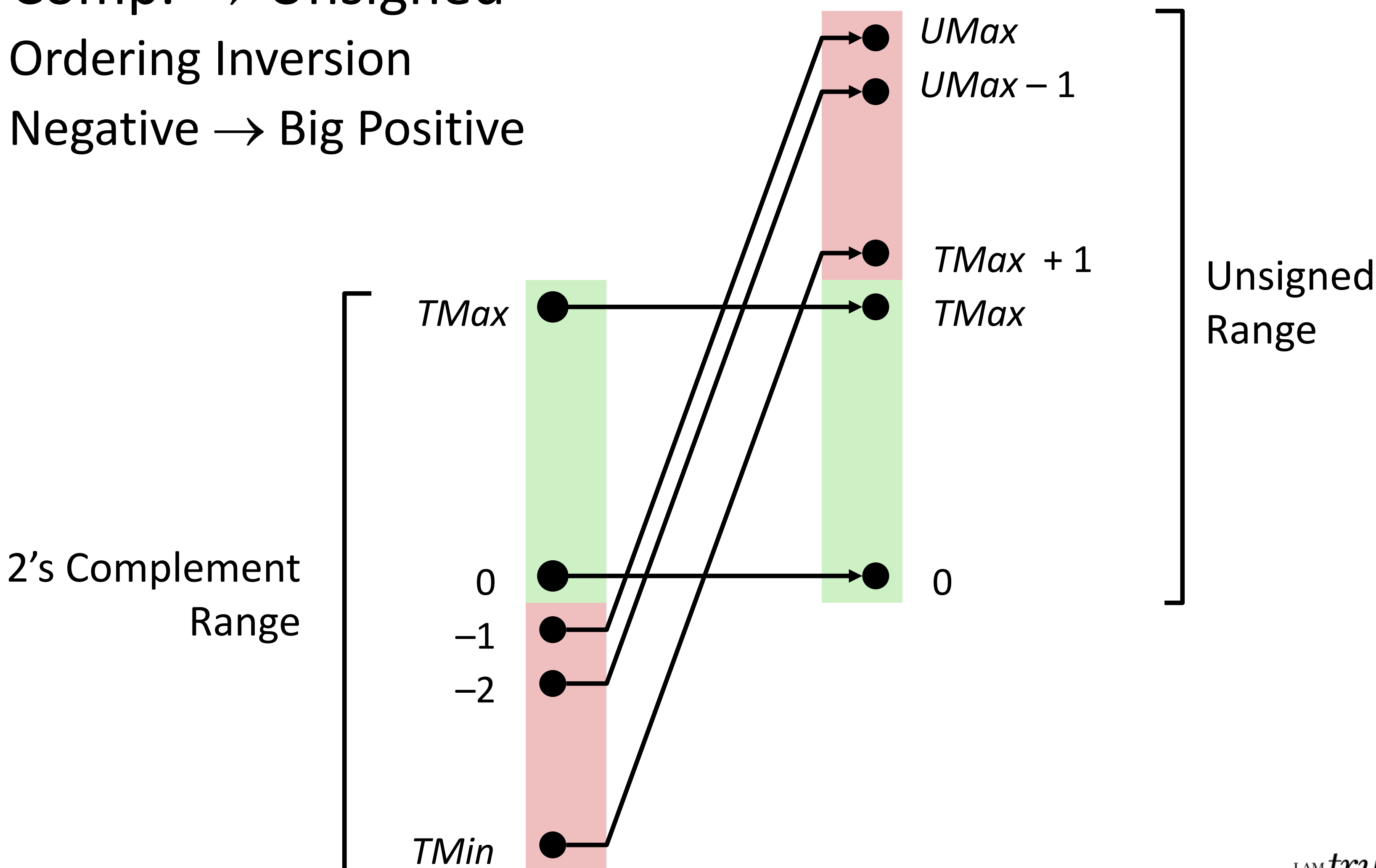
Large negative weight

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	-2147483648	>	Signed	
2147483647U	-2147483648	<	Unsigned	??
-1	-2	>	Signed	
(unsinged)-1	-2	>	Unsigned	
1U	-2	<	Unsigned	??
2147483647	2147483648U	<	Unsigned	
2147483647	(int) 2147483648U	>	Unsigned	??

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`!!

Fun Fact

- Top Programming Languages 2022

- Check the following Link:

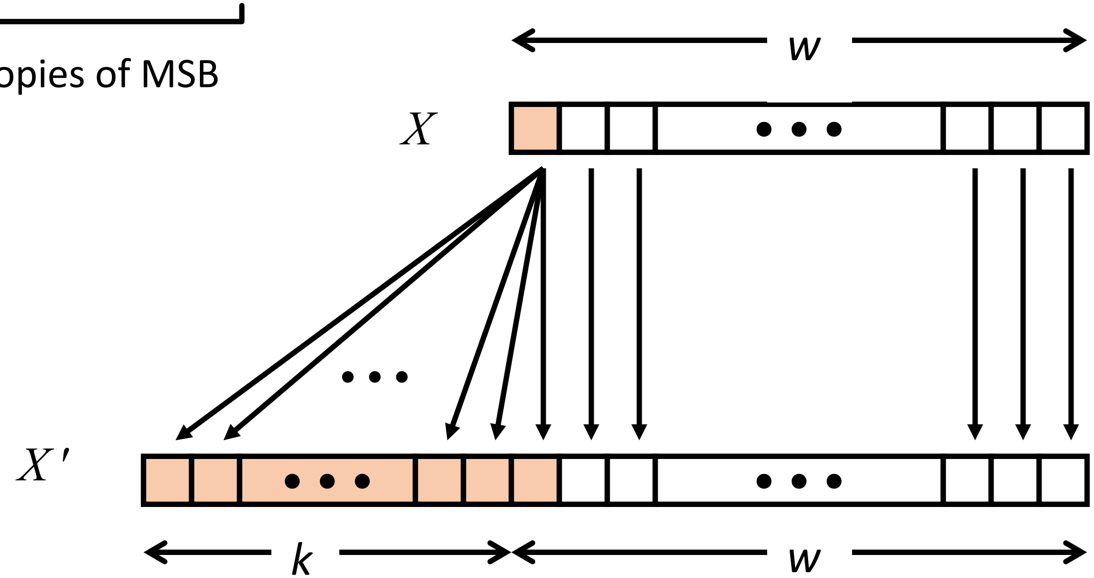
https://spectrum.ieee.org/top-programming-languages-2022

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;
```

Note: The short int is 2 bytes.

	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

Truncating (signed)

- Int to short int.
- From 4 bytes representation to 2 bytes.

```
int ix =      352758;
short int  x = (short int) ix;
int iy = 1437111;
short int  y = (short int) iy;
```

	Decimal	Hex	Binary
ix	352758	00 05 61 F6	00000000 00000101 01100001 11110110
x	25078	61 F6	01100001 11110110
iy	1437111	00 15 ED B7	00000000 00010101 11101101 10110111
y	-4681	ED B7	11101101 10110111

- Be careful with the type conversions.

Truncating (unsigned)

- unsigned Int to unsigned short int.
- From 4 bytes representation to 2 bytes.

```
unsigned int ux = 352758;
  unsigned short int usx = (unsigned short int) ux;

  unsigned int uy = 1437111;
  unsigned short int usy = (unsigned short int) uy;
```

	Decimal	Hex	Binary
ux	352758	00 05 61 F6	00000000 00000101 01100001 11110110
usx	25078	61 F6	01100001 11110110
uy	1437111	00 15 ED B7	00000000 00010101 11101101 10110111
usy	60855	ED B7	11101101 10110111

- Be careful with the type conversions.

Truncation unsigned: mod operation effect

- Assume 5-bit unsigned number $(10101)_2$ = 21
- Now, lets truncate it to 4-bit (removing most significant bit)
- You get: $(0101)_2$ = 5
- 5 = 21 MOD 2^4

- Assume 4-bit unsigned number $(1110)_2$ = 14
- Now, lets truncate it to 3-bit (removing most significant bit)
- You get: $(110)_2$ = 6
- 6 = 14 MOD 2^3

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

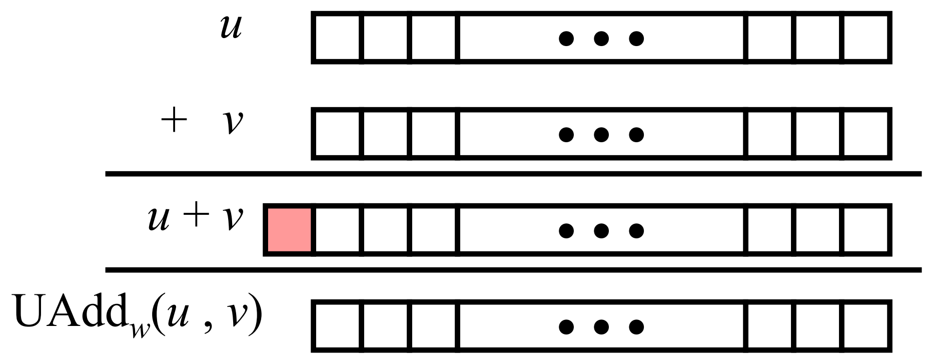
- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: *w* bits

True Sum can be: *w*+1 bits

Discard Carry: *w* bits



- Standard Addition Function
  - Ignores carry output

- Implements Modular Arithmetic

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

$$s \;=\; 10 + 14 \mod 2^4 \;=\; 24 \mod 16 = 8$$

W = 4

	1010	= 10
+	1110	= 14
	11000	= 8

discarded

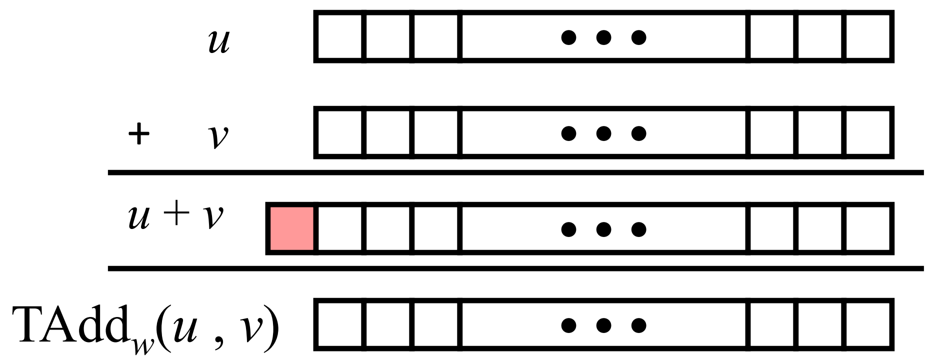
Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Two’s Complement Addition

Operands: *w* bits

True Sum can be: *w*+1 bits

Discard Carry: *w* bits



W = 4

```
   1101    = -3
+  0101    =  5
  10010    =  2
```

discarded

W = 4

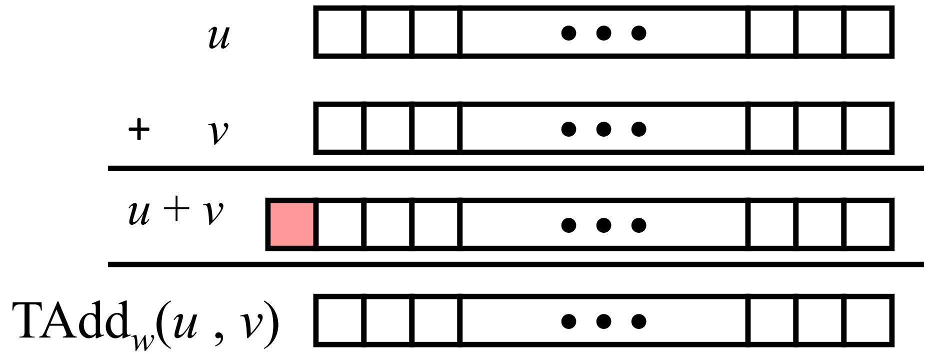
```
   1011    = -5
+  0011    =  3
   1110    = -2
```

Two's Complement Addition: overflow cases

Operands: *w* bits

True Sum can be: *w*+1 bits

Discard Carry: *w* bits



W = 4

$$
\begin{array}{rl}
1101 & = -3 \\
+\ 1010 & = -6 \\
\hline
\color{red}{1}0111 & = 7
\end{array}
$$

discarded

Negative overflow

W = 4

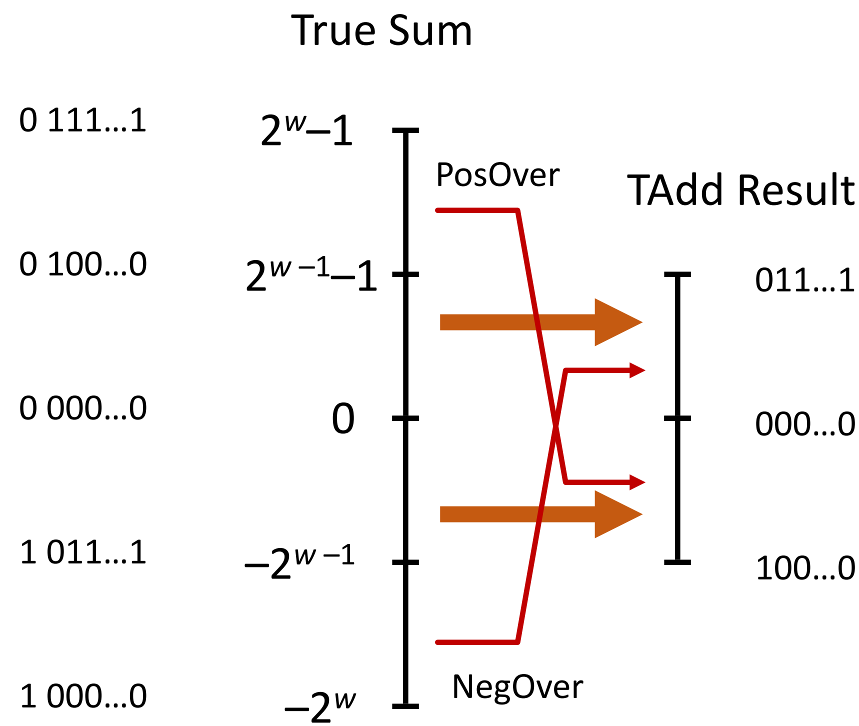
$$
\begin{array}{rl}
0111 & = 7 \\
+\ 0101 & = 5 \\
\hline
1100 & = -4
\end{array}
$$

Positive overflow

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

TAdd Overflow

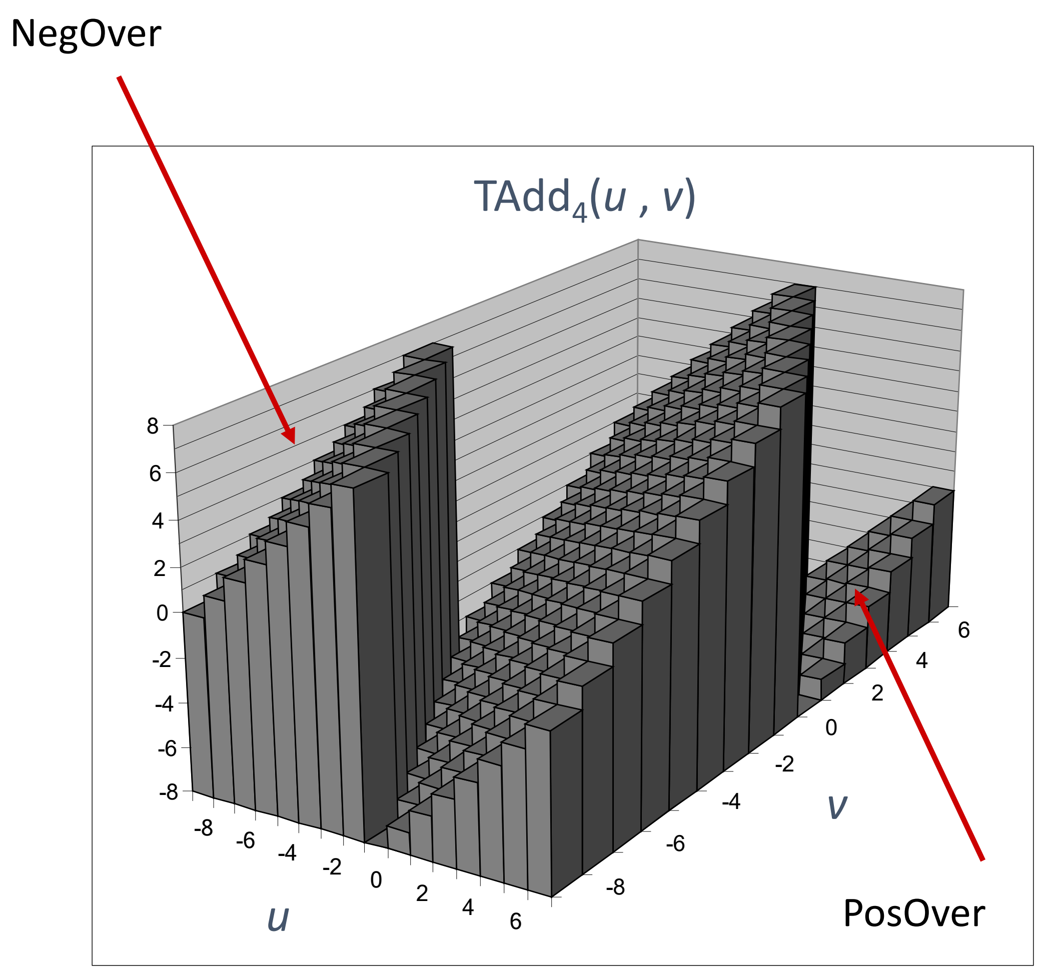
- Functionality
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



- Positive Overflow: true sum of positive number >Tmax becomes negative number
- Negative Overflow: true sum of negative numbers < Tmin becomes positive number

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 $2w$ bits
    - Result range: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to $2w$-1 bits
    - Result range: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$
    - Result range: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- **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

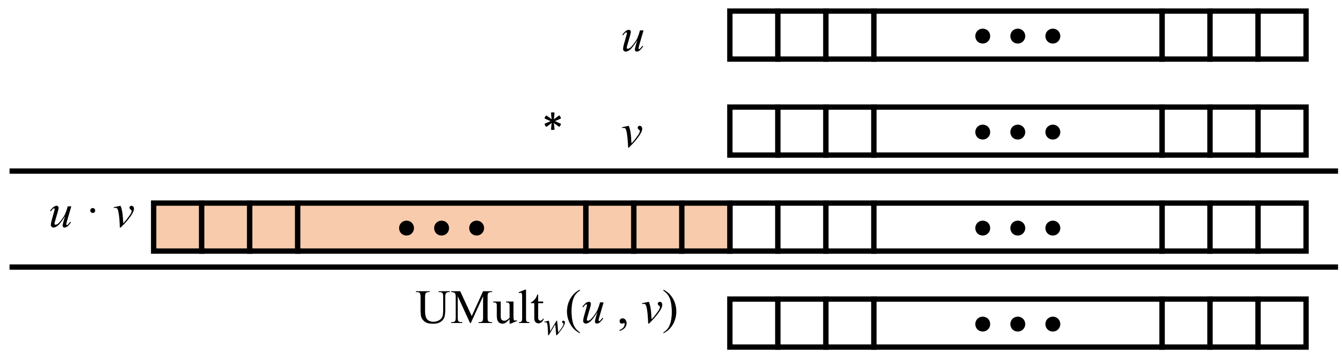
https://www.cs.mtsu.edu/~asainju/Courses/CSCI3240/private/Materials/gmpTutorial.html

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

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

W = 4

$$\begin{array}{rl} 1111 & = 15 \\ *\ 0011 & = 3 \\ \hline 101101 & = 13 \end{array}$$

discarded

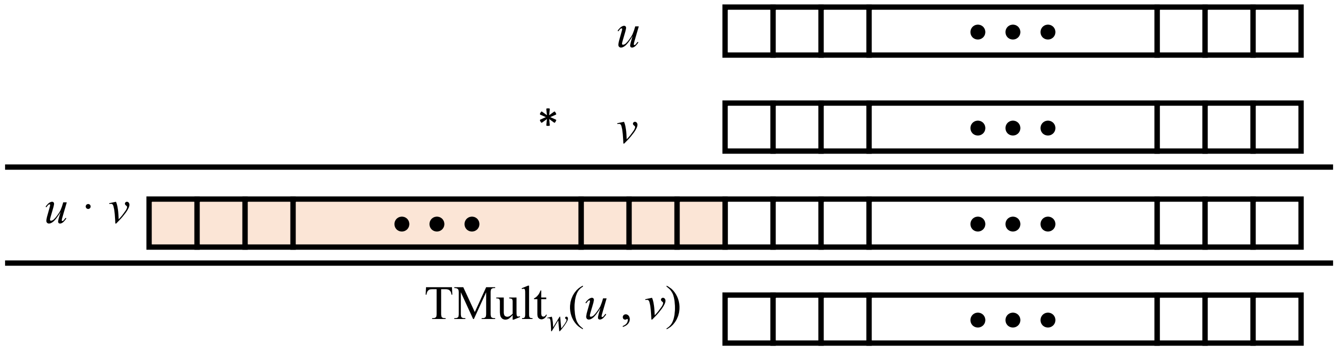
45 mod 2^4 effect

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

W = 4

```
    0101   = 5
 *  0100   = 4
 ---------------
   10100   = 4
```

discarded

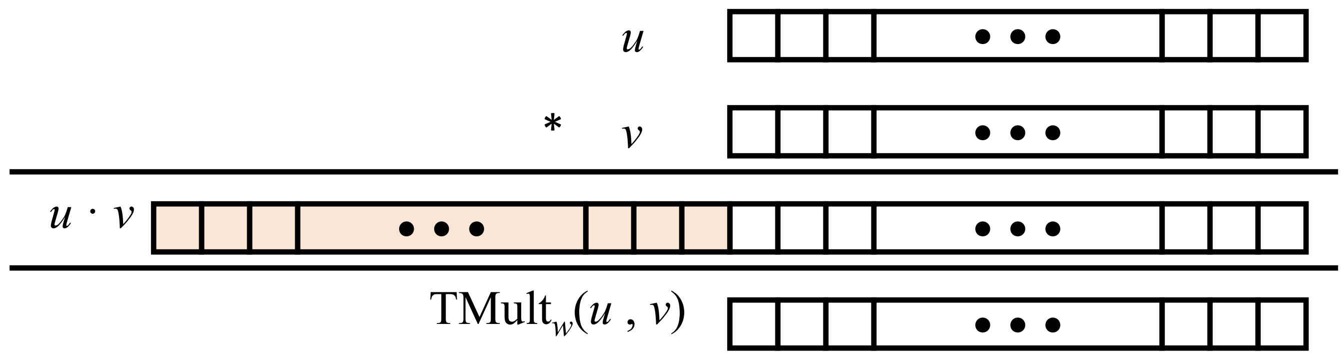
Wrong positive result

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

W = 4

```
  0101    = 5
* 0101    = 5
-----------------
 11001    = -7
```

discarded

Wrong negative result

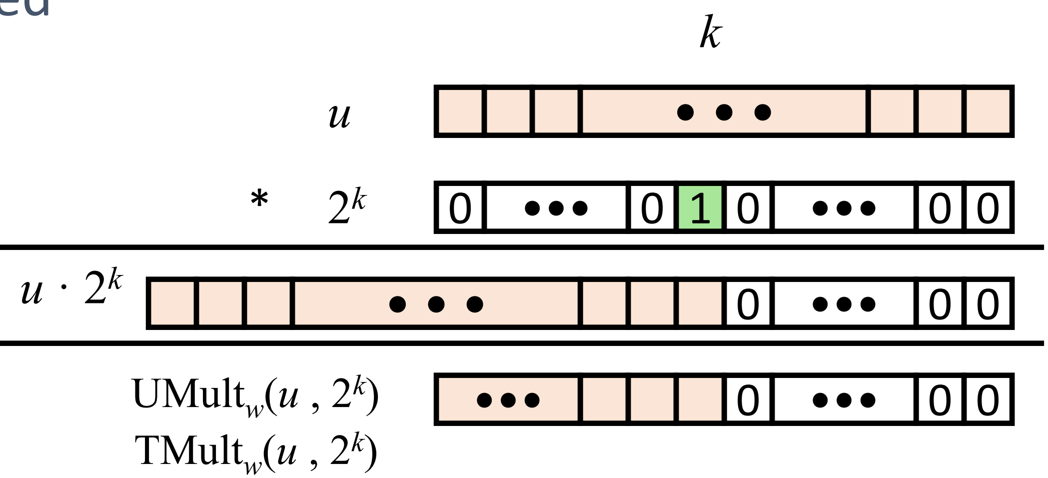
Power-of-2 Multiply with Left Shift

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

Operands: *w* bits

True Product: *w+k* bits

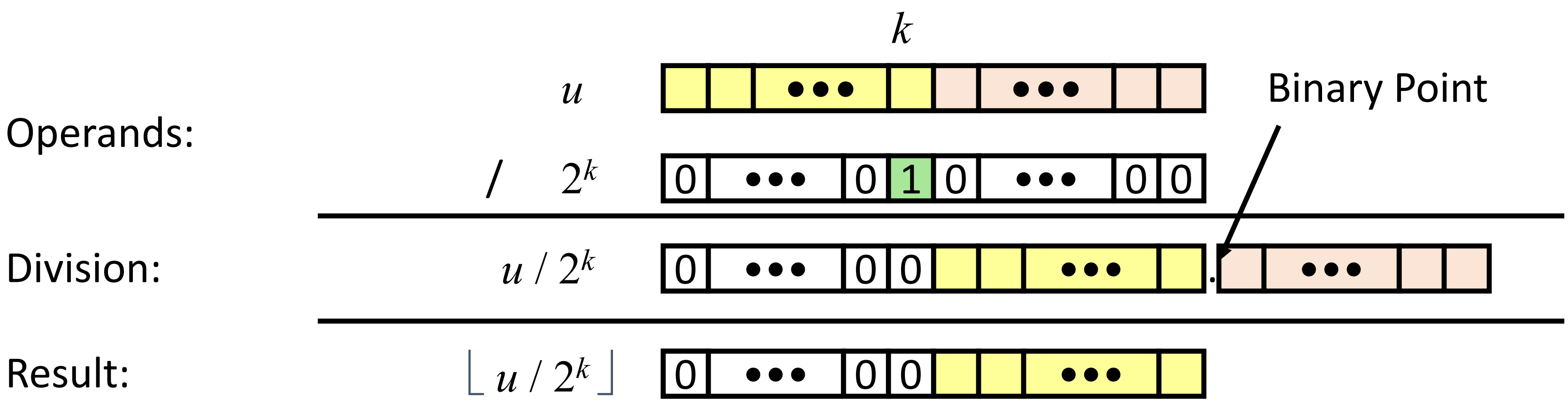
Discard *k* bits: *w* bits



- Examples
  - **`u << 3          ==   u * 8`**
  - **`(u << 5) - (u << 3)     ==    u * 24`**
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically
    - Multiplication may take over 10 clock cycles (slower)
    - Shift takes 1 clock cycle

Unsigned Power-of-2 Divide with Right Shift

- Quotient of Unsigned by Power of 2
  - `u >> k` gives $\lfloor \texttt{u} / 2^k \rfloor$
  - 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`

Unsigned Power-of-2 Divide with Right Shift

- Division is very slow operation.
- May take over 30 clock cycles (even in modern computers)

- Compiler based optimization:
  - Compiler will use the shifting technique, when possible, to minimize the computation time.

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)

Be Careful with Unsigned

- *Don't* use without understanding implications
  - Easy to make mistakes

```
unsigned i;
for (i = n-1; i >= 0; i--)
  function(a[i]);
```

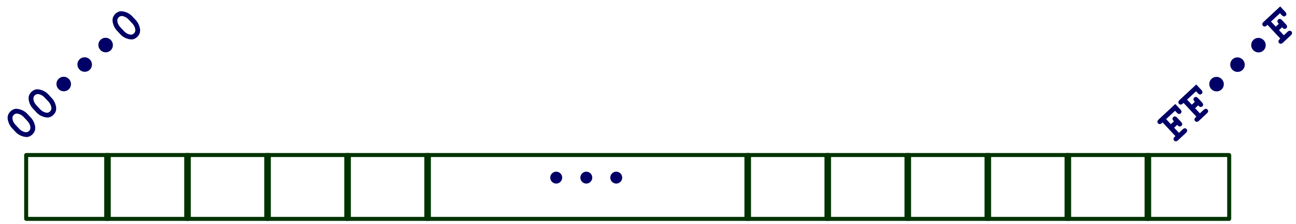
  - Can be very subtle

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

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

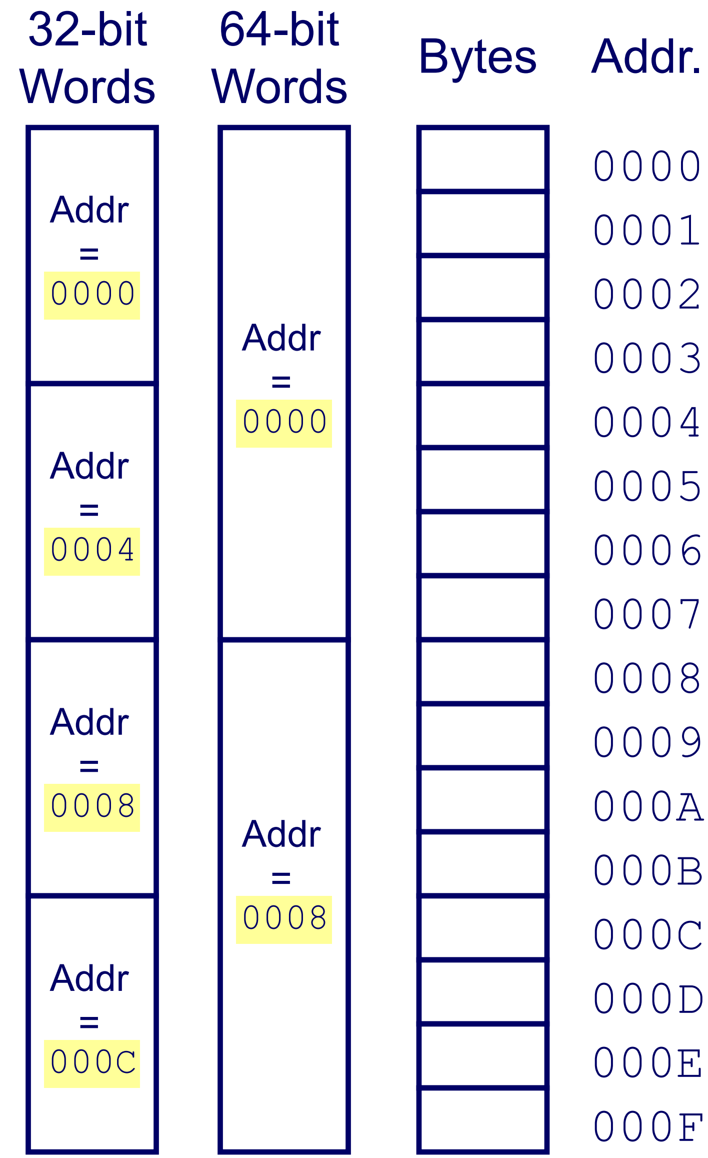
- Any given computer has a "Word Size"
  - Based on computer architecture and compiler.
  - Nominal size of integer-valued data
    - and of addresses

Gcc flags: $\begin{cases} -m32 \\ -m64 \end{cases}$

- 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 EB (exabytes) of addressable memory
  - That's 18.4 X 10^{18}
- 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

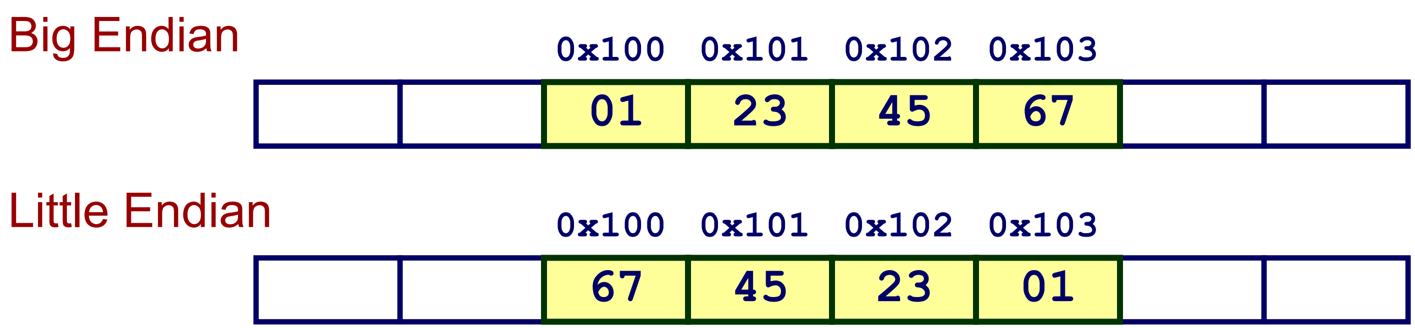
Byte Ordering

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

```
Try this command in your terminal:
    $ lscpu | grep Endian
```

Byte Ordering Example

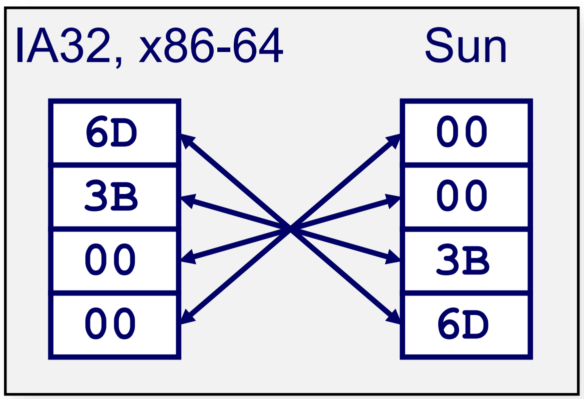
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100



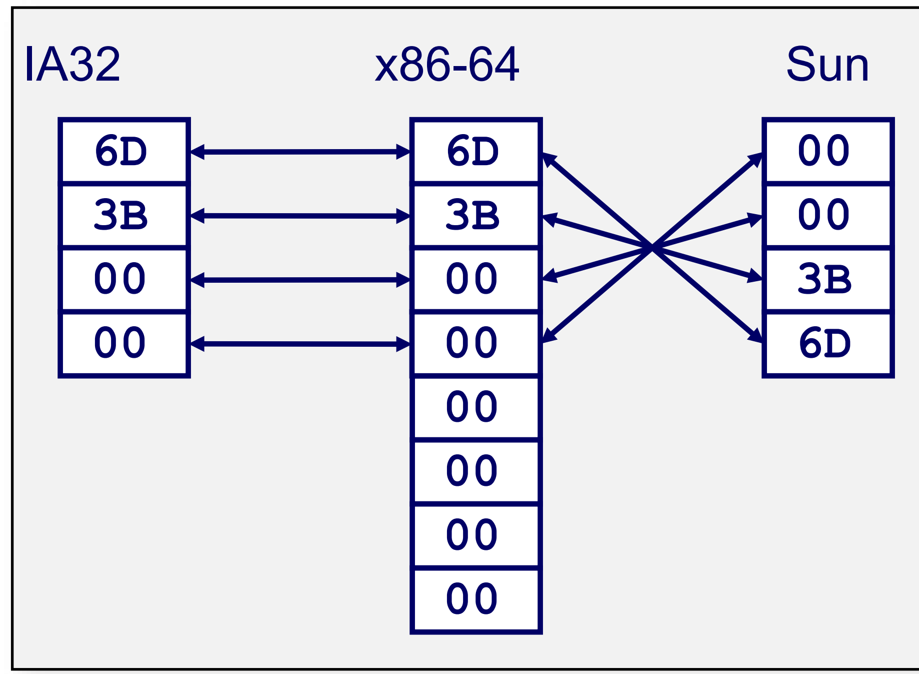
Representing Integers

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

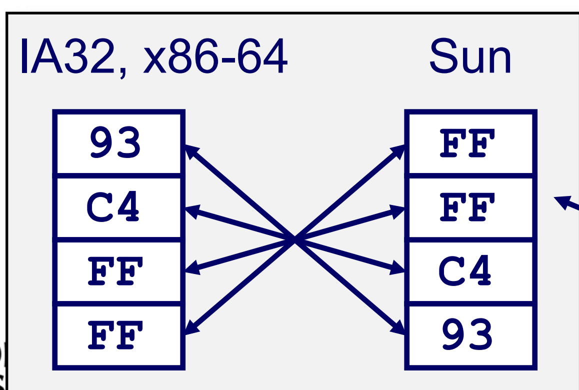
`int A = 15213;`



`long int C = 15213;`



`int B = -15213;`



Two's complement representation

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

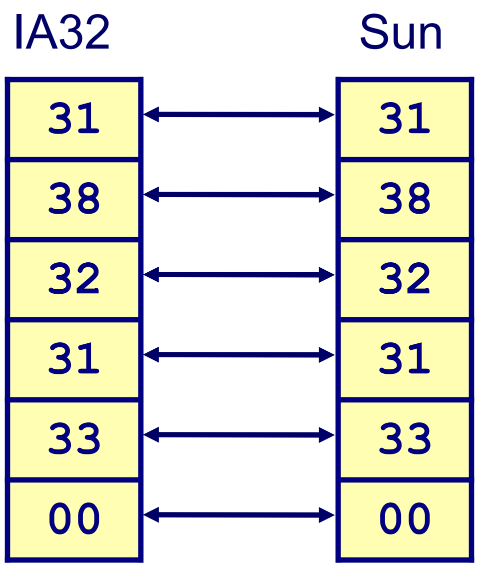
Different compilers & machines assign different locations to objects

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

Check if the statements are always true?

Initialization

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

```
1. x < 0             =>  ((x*2) < 0)
2. ux >= 0
3. x & 7 == 7        =>  (x<<30) < 0
4. ux > -1
5. x > y             =>  -x < -y
6. x * x >= 0
7. x > 0 && y > 0    =>  x + y > 0
8. x >= 0            =>  -x <= 0
9. x <= 0            =>  -x >= 0
10.(x|-x)>>31 == -1
11.ux >> 3 == ux/8
12.x >> 3 == x/8
13.x & (x-1) != 0
```