
WELCOME
TO
15-213
14-513
18-213
18-613
15-513

# Please read pinned Ed posts!

# From Bits through Integers

15-213/14-513/15-513:
Introduction to Computer Systems

2nd Lecture, Spring 2026

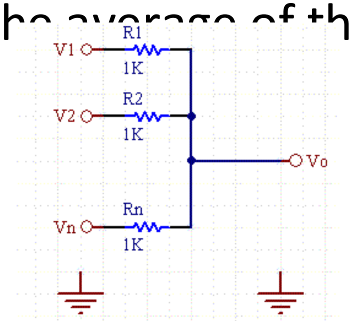
# Bits, Bytes, and Integers

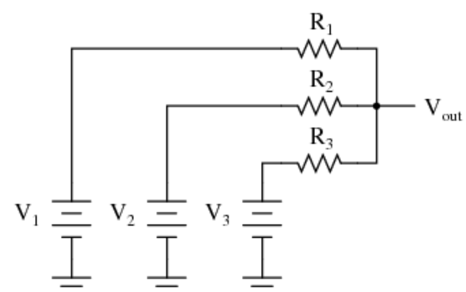
- **Representing information as bits** CSAPP 2.1
- **Bit-level manipulations**
- **Integers** CSAPP 2.2
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting CSAPP 2.3
- **Byte Ordering** CSAPP 2.1.3

# Analog Computers

- **Before digital computers there were analog computers.**

- **Consider a couple of simple analog computers:**
  - A simple circuit can allow one to adjust voltages using variable resistors and measure the output using a volt meter:
  - A simple network of adjustable parallel resistors can allow one to find the average of the inputs.




https://www.daycounter.com/Calculators/Voltage-Summer/Voltage-Summer-Calculator.phtml

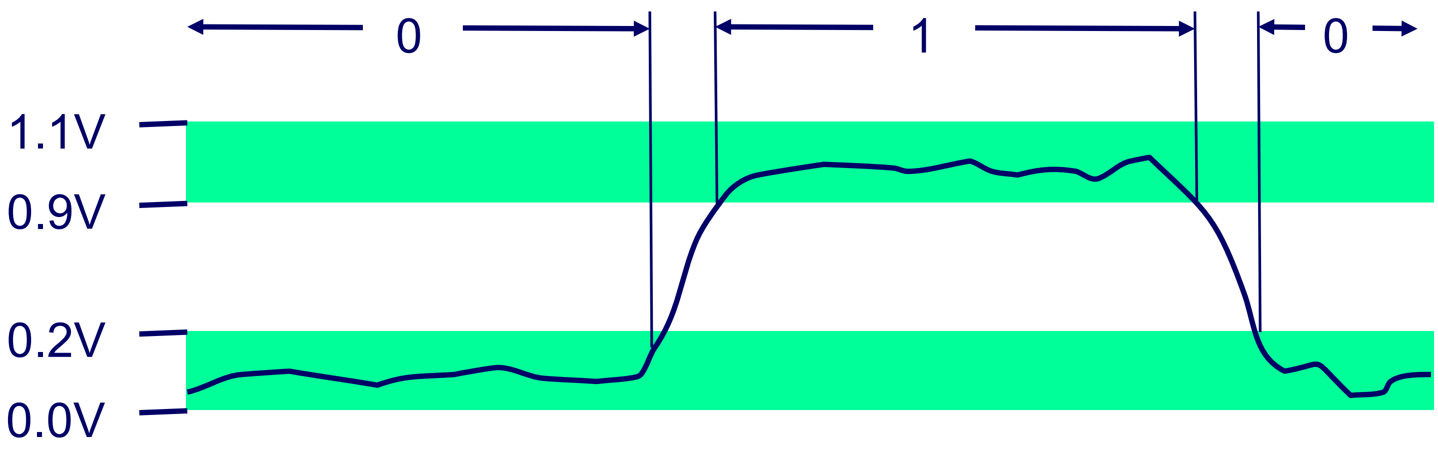
https://www.quora.com/What-is-the-most-basic-voltage-adder-circuit-without-a-transistor-op-amp-and-any-external-supply

# Needing Less Accuracy+Precision is Easier

- **We don't try to measure exactly**
  - We just ask, is it high enough to be "On", or
  - Is it low enough to be "Off".

- **We have two states, so we have a binary, or 2-ary, system.**
  - We represent these states as 0 and 1

- **Now we can easily interpret, communicate, and duplicate signals well enough to know what they mean.**



# Binary Representation

- **Binary representation -> binary, base-2, numbering system**
  - 0 represents 0
  - 1 represents 1
  - Each “place” represents a power of two, exactly as each place in our usual “base 10”, 10-ary numbering system represents a power of ten
- **Encoding/interpreting sets of bits can represent…**
  - Operations to be executed by the processor
  - numbers
  - enumerable things, such as text characters
  - more!
- **If can assign it to a discrete number, we can represent it in binary**

# Binary Representation: Simple Numbers

- **Example: we can count in binary, a base-2 numbering system**
  - 000, 001, 010, 011, 100, 101, 110, 111, …
    - $000 = 0*2^2 + 0*2^1 + 0*2^0 = 0$ (in decimal)
    - $001 = 0*2^2 + 0*2^1 + 1*2^0 = 1$ (in decimal)
    - $010 = 0*2^2 + 1*2^1 + 0*2^0 = 2$ (in decimal)
    - $011 = 0*2^2 + 1*2^1 + 1*2^0 = 3$ (in decimal)
    - Etc.
- **For reference, consider some base-10 examples:**
    - $000 = 0*10^2 + 0*10^1 + 0*10^0$
    - $001 = 0*10^2 + 0*10^1 + 1*10^0$
    - $357 = 3*10^2 + 5*10^1 + 7*10^0$

# Hexadecimal and Octal

- **Writing numbers in binary takes too many digits**

- **Want a way to represent numbers more densely such that fewer digits are required**
  - But also so it is easy to get at the bits that we want

- **Any power-of-two base provides this property**
  - Octal (base 8) and hexadecimal (base 16) are closest to our familiar base-10.
  - Each has been used by “computer people” over time
  - Hexadecimal is often preferred because it is denser.

# Hexadecimal

- **Hexadecimal $00_{16}$ to $FF_{16}$**
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
- **Consider 1A2B in Hexadecimal:**
  - $1*16^3 + A*16^2 + 2*16^1 + B*16^0$
  - $1*16^3 + 10*16^2 + 2*16^1 + 11*16^0$ = 6699 (decimal)
  - The C Language prefixes hexadecimal numbers with “0x” so they aren’t confused with decimal numbers
  - 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 |

$$\texttt{15213:}\ \underbrace{\texttt{0011}}_{\texttt{3}}\ \underbrace{\texttt{1011}}_{\texttt{B}}\ \underbrace{\texttt{0110}}_{\texttt{6}}\ \underbrace{\texttt{1101}}_{\texttt{D}}$$

# 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
- Byte Ordering

# 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 |

# The Boolean Operators Work On ALL bits

- 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
    -
  - ~0x00 → 0xFF
    -
  - 0x69 & 0x55 → 0x41
    -
  - 0x69 | 0x55 → 0x7D
    -

| 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 |

# 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
    - $01101001_2 \ \& \ 01010101_2 \rightarrow 01000001_2$
  - 0x69 | 0x55 → 0x7D
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

| 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 |

# 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 → 1011 1110
  - ~0x00 → 1111 1111
  - `0x69 & 0x55:` / `0x69 | 0x55:`

```
    0110 1001          0110 1001
  & 0101 0101        | 0101 0101
 -----------------  -----------------
    0100 0001          0111 1101
```

| 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 |

# Contrast: Logic Operations in C

- **Contrast to Bit-Level Operators**
  - **Logic Operations: &&, ||, !**
    - 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. |)…
Super common C programming pitfall!

# 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 | 00010000 |
| Log. >> 2 | 00011000 |
| Arith. >> 2 | 00011000 |

| | |
|---|---|
| Argument **x** | 10100010 |
| << 3 | 00010000 |
| Log. >> 2 | 00101000 |
| Arith. >> 2 | 11101000 |

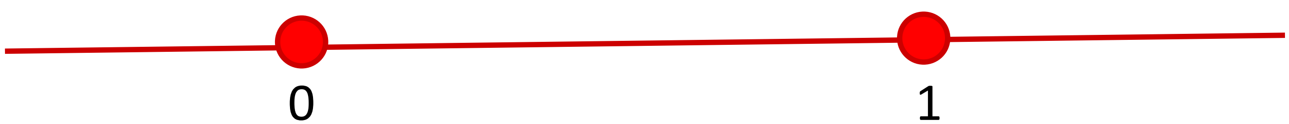
# Today: Bits, Bytes, and Integers

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

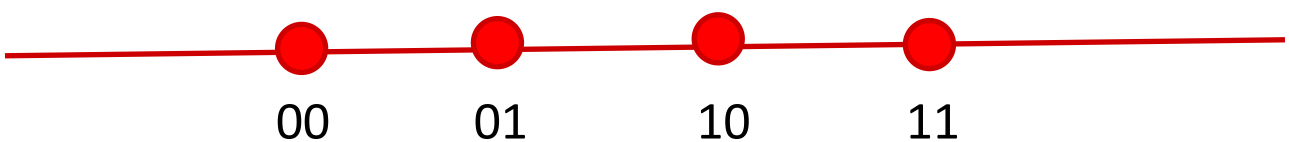
# Binary Number Lines

- **In binary, the number of bits in the data type size determines the number of points on the number line.**
  - We can assign the points any meaning we'd like

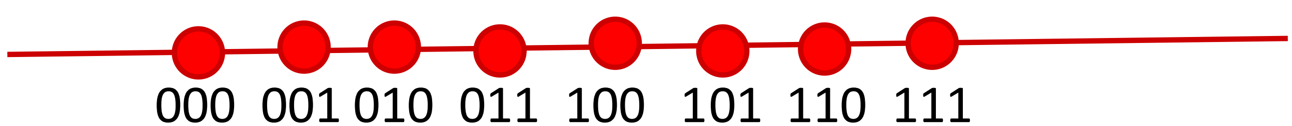
- **Consider the following examples:**
  - 1 bit number line




  - 2 bit number line

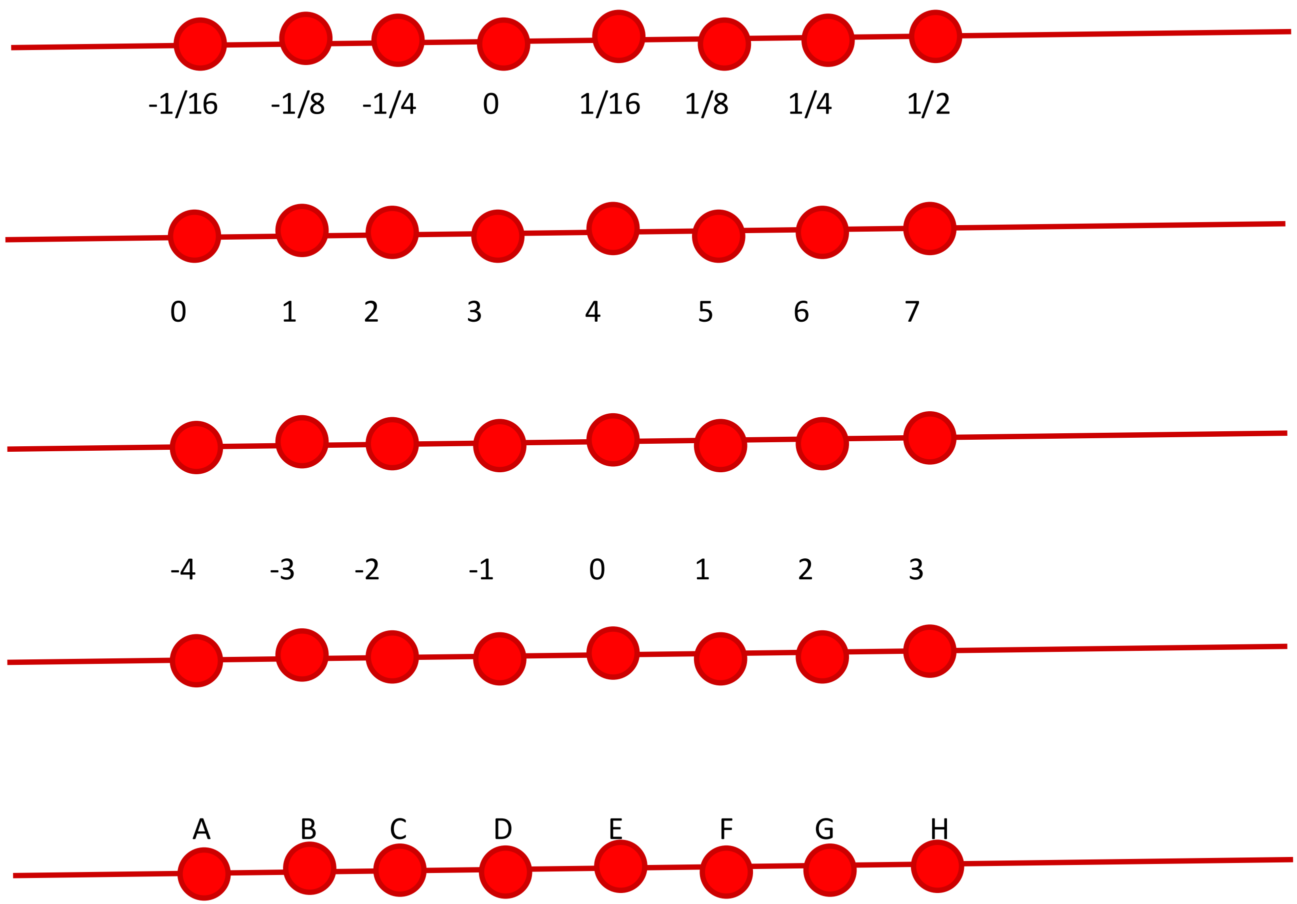



  - 3 bit number line



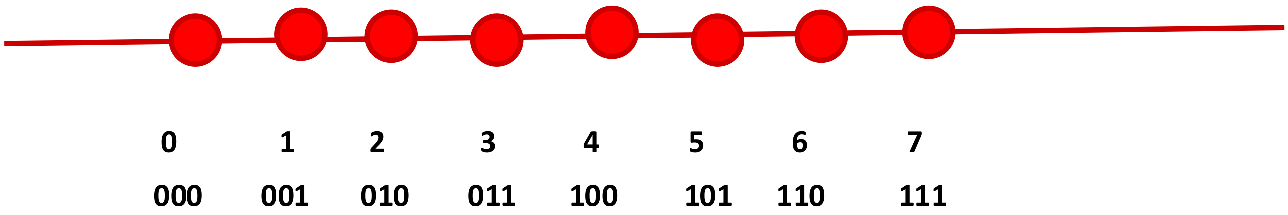
# Some Purely Imaginary Examples

- **3 bit number line**



# Overflow

- **Let's consider a simple 3 digit number line:**




- **What happens if we add 1 to 7?**
  - In other words, what happens if we add 1 to 111?

- **111+ 001 = 1 000**
  - But, we only get 3 bits – so we lose the leading 1.
  - This is called overflow

- **The result is 000**

# Modulus Arithmetic

- **Let's explore this idea of overflow some more**
  - 111 + 001 = 1 000 = 000
  - 111 + 010  = 1 001 = 001
  - 111 + 011 =  1 010  = 010
  - 111 + 100 =  1 011  = 011
  - …
  - 111 + 110  = 1 101 = 101
  - 111 + 111 = 1 110 =  110

- **So, arithmetic "wraps around" when it gets "too positive"**

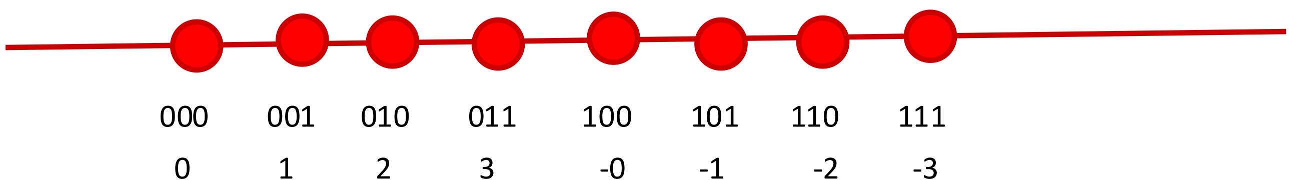
# Unsigned and Non-Negative Integers

- **We'll use the term "ints" to mean the finite set of integer numbers that we can represent on a number line enumerated by some fixed number of bits, i.e. *bit width*.**

- **We normally represent unsigned and non-negative int using simple binary as we have already discussed**
  - An "unsigned" int is any int on a number line, e.g. of a data type, that doesn't contain any negative numbers
  - A non-negative number is a number greater than or equal to (>=) 0 on a number line, e.g. of a data type, that does contain negative numbers

# How to represent negative Numbers?

- **We could use the leading bit as a *sign bit*:**
  - 0 means non-negative
  - 1 means negative




- **Benefits**
  - represents negative and non-negative numbers
  - 0 represents 0

- **Drawbacks**
  - There is a -0, which is the same as 0, except that it is different
  - How to add such numbers 1 + -1 should equal 0
    - But, by simple math, 001 + 101 = 110, which is -2?

# A Trick!

- **Let's start with three ideas:**
  - 1 should be represented as 1
  - -1 + 1 = 0
  - We want addition to work in the familiar way, with simple rules.

- **Consider a 3 bit number:**
  - 001 + "-1" = 0
  - 001 + 111 = 0
    - Remember 001 + 111 = 1 000, and the leading one is lost to overflow.
- **"-1" = 111**
  - Yep!

# Negative Numbers

- **Well, if 111 is -1, what is -2?**
  - -1 - 1
  - 111 – 001 = 110
- **Does that really work?**
  - If it does -2 + 2 = 0
  - 110 + 010 = 1 000 = 000
- **-2 + 5 should be 3, right?**
  - 110 + 101 = 1 011 = 011

# Finding –x the easy way

- **Given a non-negative number in binary, e.g. 5, represented with a fixed bit width, e.g. 4**
  - 0101

- **Find its negative by flipping each bit and adding 1**
  - 0101 This is 5
  - 1010 This is the "ones complement of 5", e.g. 5 with bits flipped
  - 1011 This is the "twos complement of 5", e.g. 5 with the bits flipped and 1 added
  - 0101 + 1011 = 1 0000 = 0000
  - -x = ~x+1

- **Because of the fixed width, the "two's complement" of a number can be used as its negative.**

# Why Does This Work?

- **Consider any number and its (ones) complement:**
  - 0101
  - 1010

- **They are called complements because complementary bits are set. As a result, if they are added, all bits are necessarily set:**
  - 0101 + 1010 = 1111

- **Adding 1 to the sum of a number and its complement necessarily results in a 0 due to overflow**
  - (0101 + 1010) + 1 = 1111 + 1 = 1 0000 = 0000

- **And if x + y = 0, y must equal –x**

# Why Does This Work? *Cont.*

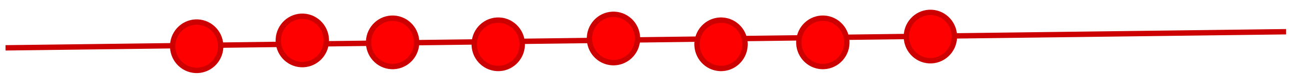
- **If x + y = 0**
  - y must equal –x

- **So if x + (Complement(x) + 1) = 0**
  - Complement(x) + 1 must equal –x

- **Another way of looking at it:**
  - if x + (Complement(x) + 1) = 0
  - x + Complement(x) = -1
  - x = -1 - Complement(x)
  - -x = 1 + Complement(x)

# Visualizing Two's Complement

- **Numbers "wrap around" with -1 at the very end**



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 1 | 2 | 3 | -4 | -3 | -2 | -1 |

- **A few things to note:**
  - All negative numbers start with a "1"
    - E.g. 100 is "-4"
  - You can view the leading "1" as introducing a "-4"
    - E.g. 101 = 1*-4+0*2+1*1= -3
    - But 010 = 0*-4+1*2+0*1 = 2
  - -4 is missing a positive partner

# Complement & Increment Examples

x = 0

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

x = Tmin (The most negative two's complement number)

| | Decimal | Hex | Binary |
|---|---|---|---|
| x | -32768 | 80 00 | 10000000 00000000 |
| ~x | 32767 | 7F FF | 01111111 11111111 |
| ~x+1 | -32768 | 80 00 | 10000000 00000000 |

Canonical counter example

# Encoding Integers: Dense Form

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 does not mandate using two's complement**
  - But, most machines do, and we will assume so
- **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

# 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
  - 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 |

**We're going to skip the quiz today**
**Do it on your own if you want**
**OR BETTER do the exercises at the end of the lecture**

**Check out:**

**https://canvas.cmu.edu/courses/46319/quizzes/139185**

```c
unsigned int u = 1;

int i = (int) u;
```

```
unsigned int u = 0xffffffff;

int i = (int) u;

printf(“%d\n”, i);
```

# 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

# 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;                    int fun(unsigned u);
uy = ty;                    uy = fun(tx);
```

# 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
- Byte Ordering

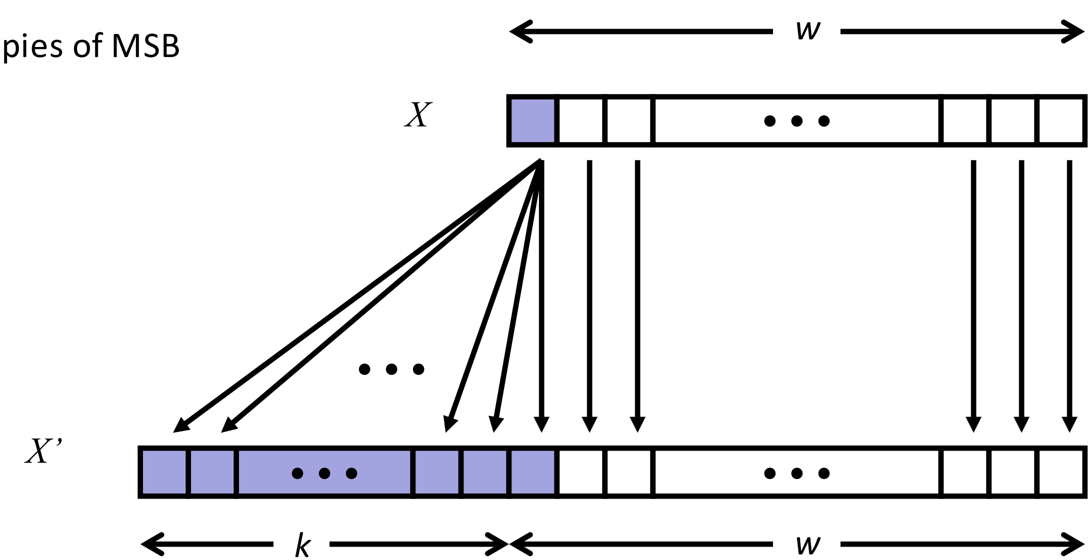
```
uint8_t su = 1;

uint16_t bu = su;
```

```
int8_t su = -1;

int16_t bu = su;
```

```
int8_t su = -1;

int16_t bu = su;

su is 0xff
bu better be 0xffff

How?
```

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

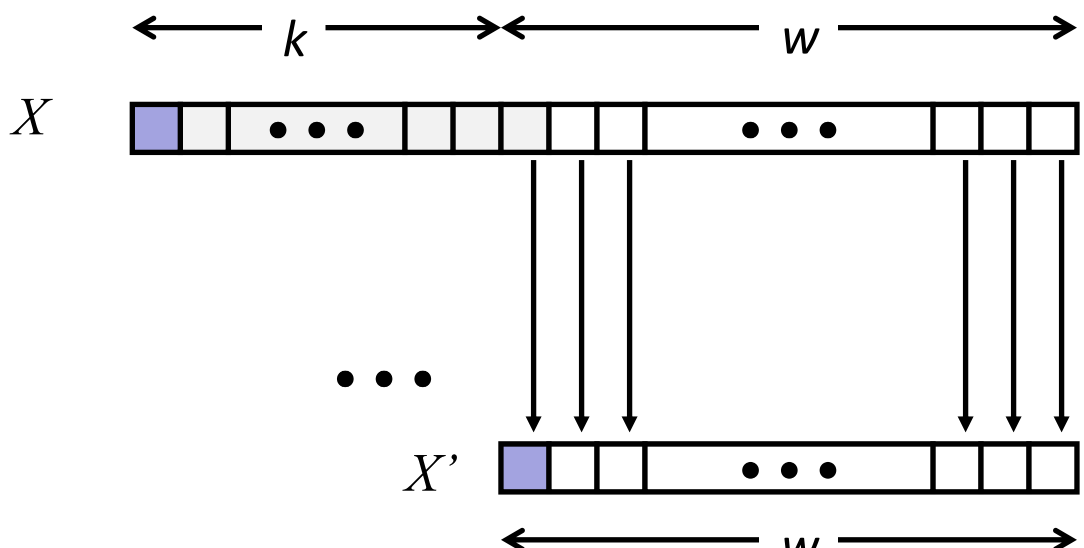


```
uint16_t bu = 1;

uint8_t su = bu;
```

```
uint16_t bu = 1000;

uint8_t su = bu; ???

bu = 0x8000 (0b100..)
   = negative max
   su = bu ???
```

# Truncation

- **Task:**
  - Given k+$w$-bit signed or unsigned integer $X$
  - Convert it to $w$-bit integer X' with same value for "small enough" X
- **Rule:**
  - Drop top $k$ bits:
  - $X' = x_{w-1}, x_{w-2}, \ldots, x_0$



# Truncation: Simple Example

## No sign change

| | -16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|
| 2 = | 0 | 0 | 0 | 1 | 0 |

| | -8 | 4 | 2 | 1 |
|---|---|---|---|---|
| 2 = | 0 | 0 | 1 | 0 |

2 mod 16 = 2

| | -16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|
| -6 = | 1 | 1 | 0 | 1 | 0 |

| | -8 | 4 | 2 | 1 |
|---|---|---|---|---|
| -6 = | 1 | 0 | 1 | 0 |

-6 mod 16 = 26U mod 16 = 10U = -6

## Sign change

| | -16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|
| 10 = | 0 | 1 | 0 | 1 | 0 |

| | -8 | 4 | 2 | 1 |
|---|---|---|---|---|
| -6 = | 1 | 0 | 1 | 0 |

10 mod 16 = 10U mod 16 = 10U = -6

| | -16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|
| -10 = | 1 | 0 | 1 | 1 | 0 |

| | -8 | 4 | 2 | 1 |
|---|---|---|---|---|
| 6 = | 0 | 1 | 1 | 0 |

-10 mod 16 = 22U mod 16 = 6U = 6

# 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 (in magnitude) numbers yields expected behavior

# Today: Bits, Bytes, and Integers

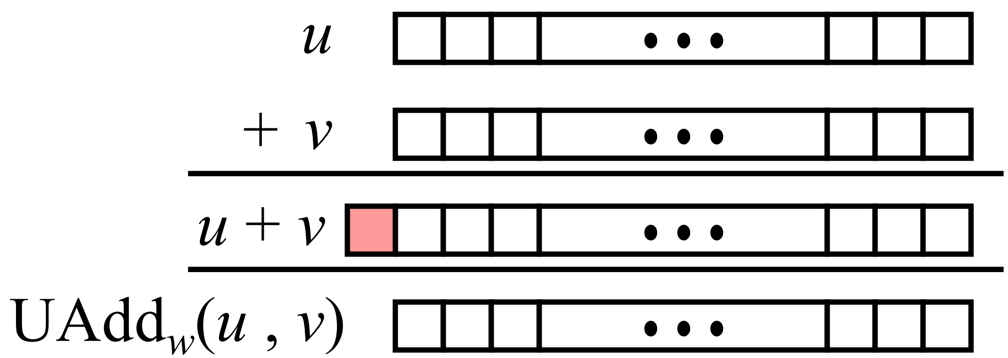
- **Representing information as bits**
- **Bit-level manipulations**
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- **Byte Ordering**

# Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits




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

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

```
unsigned char       1110 1001        E9        223
                  +   1101 0101      + D5      + 213
                   1 1011 1110       1BE        446
                      1011 1110        BE        190
```

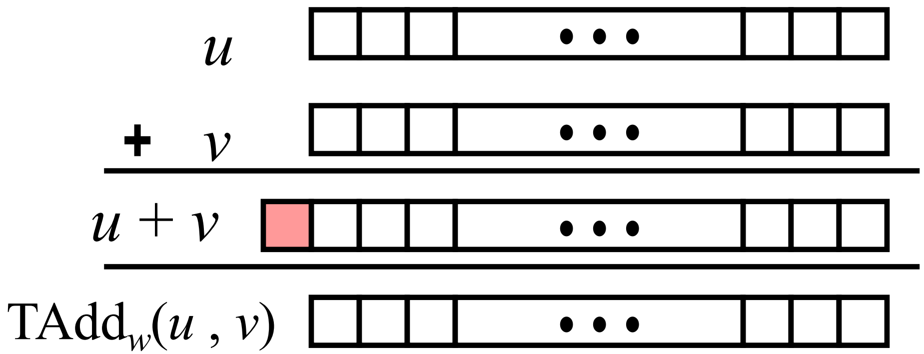
| Hex | Decimal | Binary |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |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

# 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
```

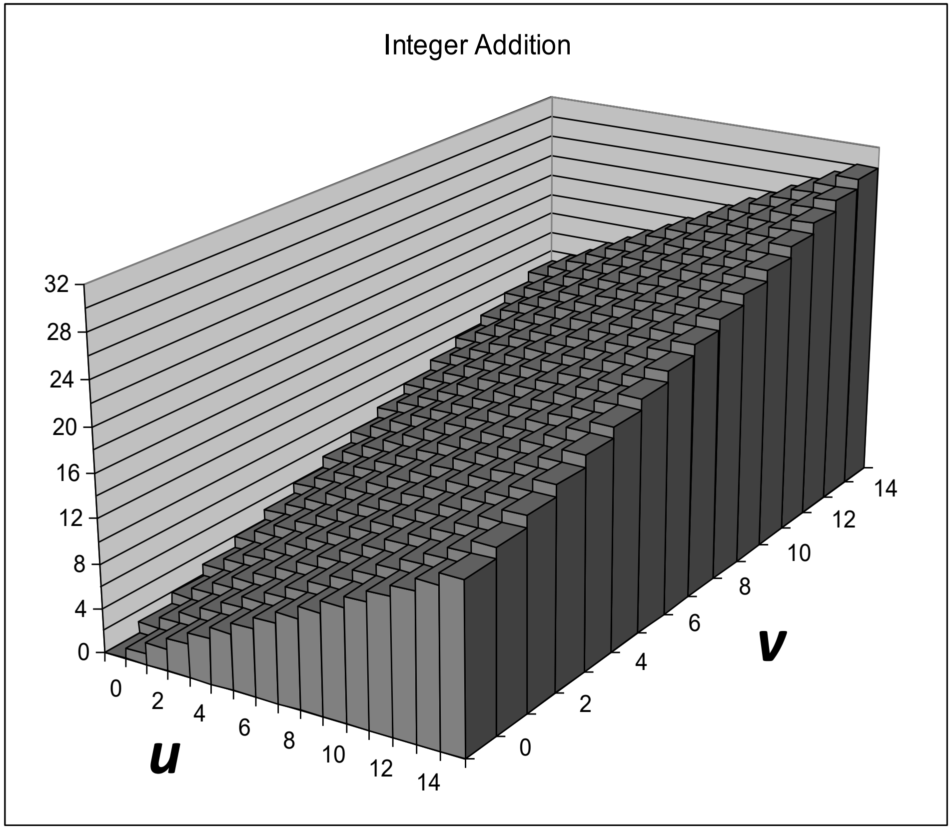
  - Will give `s == t`

| | | |
|---|---|---|
| 1110 1001 | E9 | -23 |
| + 1101 0101 | + D5 | + -43 |
| 1 1011 1110 | 1BE | -66 |
| 1011 1110 | BE | -66 |

# Visualizing "True Sum" Integer Addition

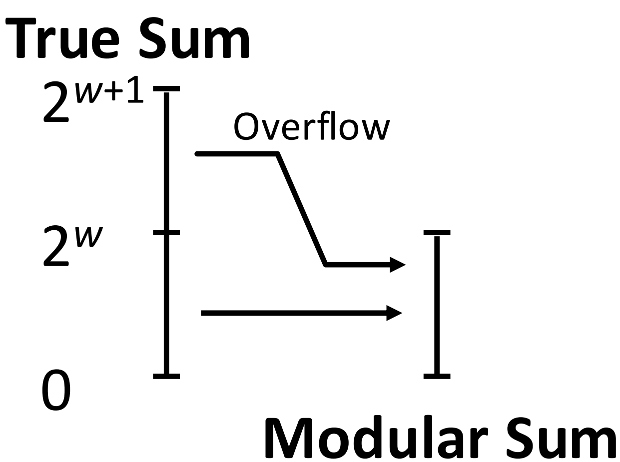
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u\ ,\ v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

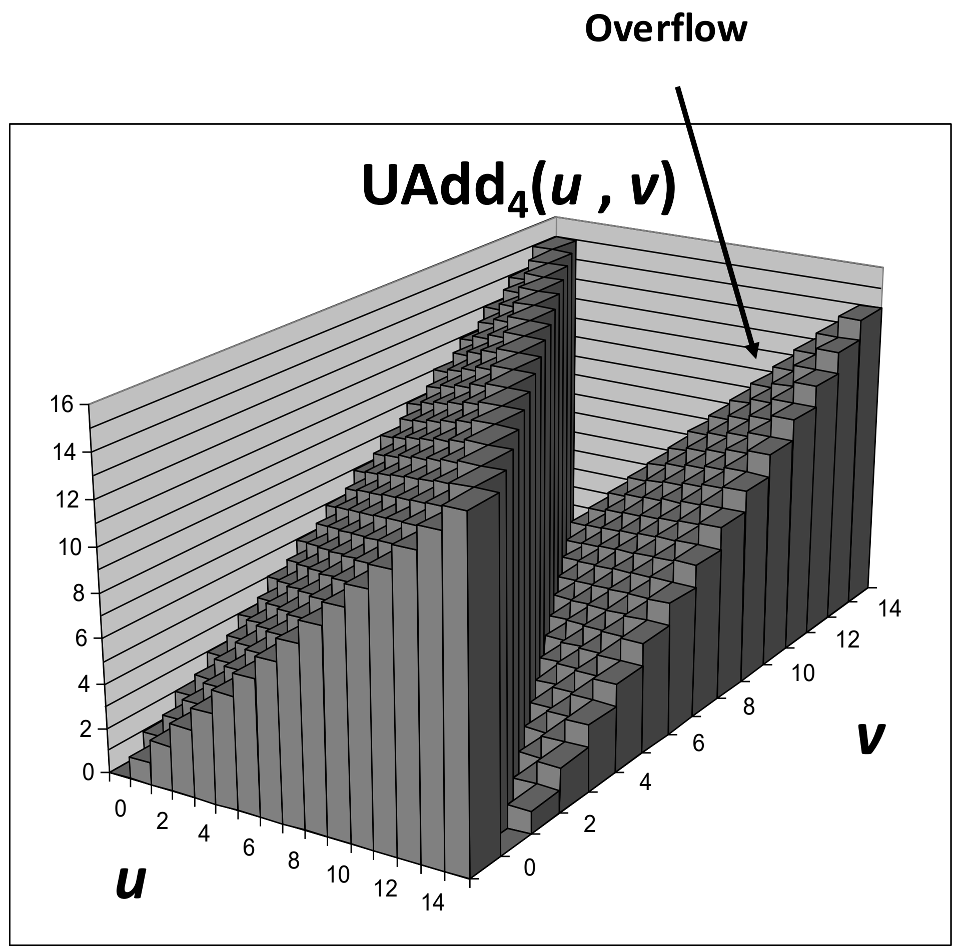
**$\text{Add}_4(u\ ,\ v)$**



# Visualizing Unsigned Addition

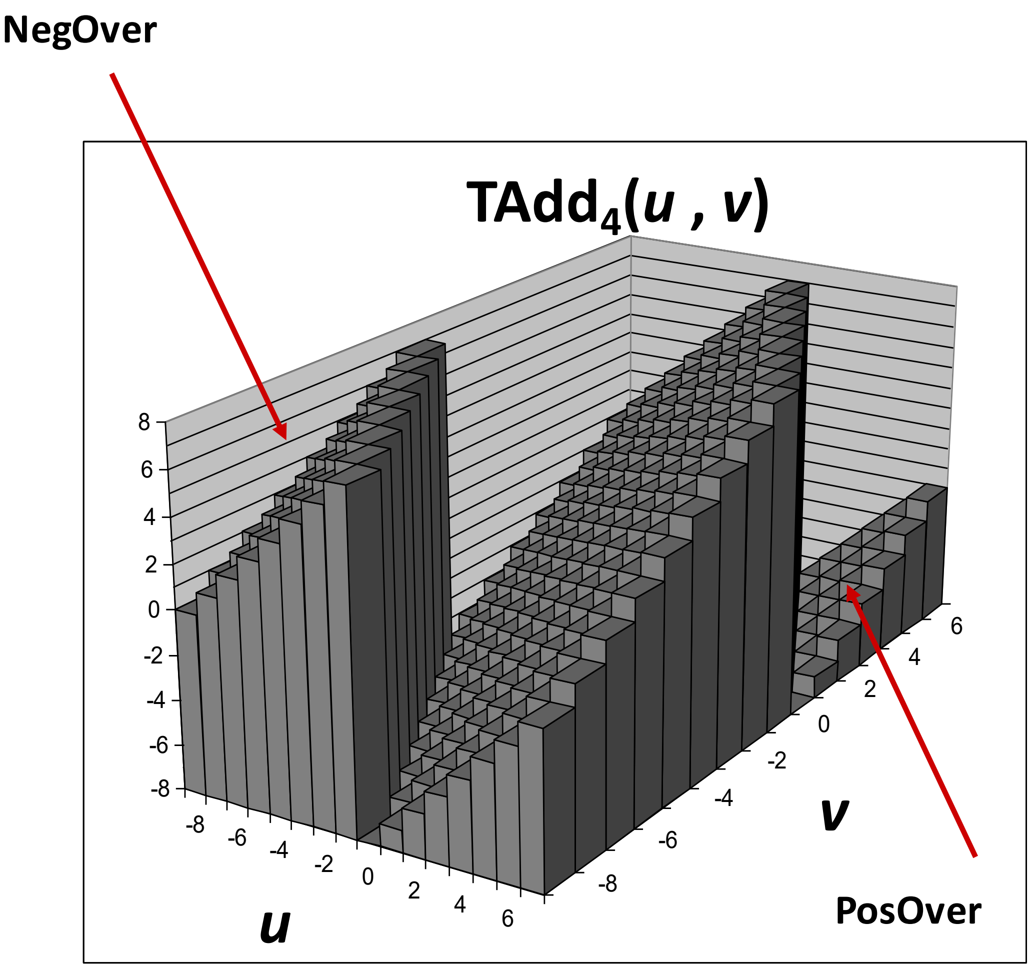
- **Wraps Around**
  - If true sum ≥ $2^w$
  - At most once



# 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
- **Result: Same as computing ideal, exact result *x*y* and keeping *w* lower bits.**
- **Ideal,exact results can be bigger than *w* bits**
  - Worst case is up to 2w bits
    - Unsigned, because all bits are magnitude
    - Signed, but only for Tmin*Tmin, because anything added to Tmin reduces its magnitude and Tmax is less than Tmin.
- **So, maintaining exact results…**
  - would need to keep expanding word size with each product computed
  - Impossible in hardware (at least without limits), as all resources are finite
  - In practice, is done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

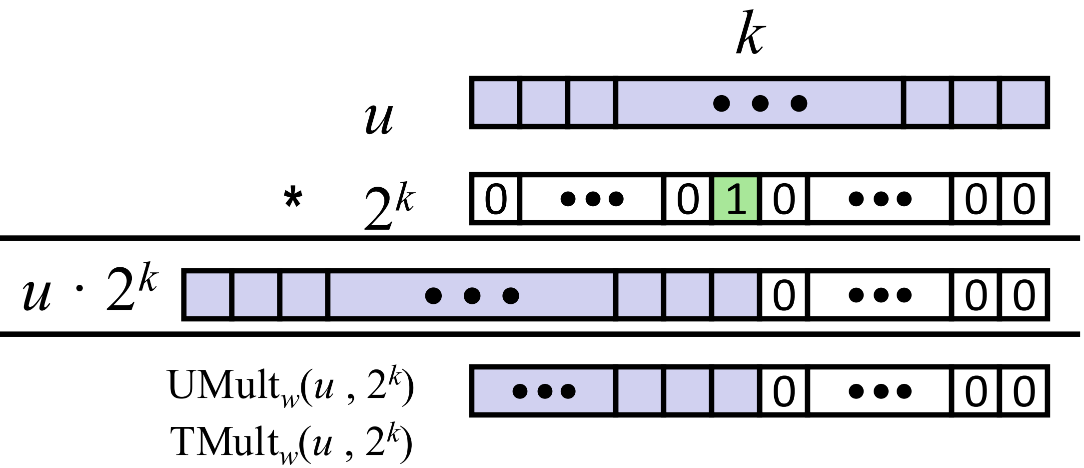
# Power-of-2 Multiply with Shift

- **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

# Dividing by shifting

```
   4 /  2 =  2
0100 >> 1 = 0010

   4 /  4 =  1
0100 >> 2 = 0001

  N  /  2^k
  N >>    k
```

# How does this round?

```
   5 /  2 =  2
0101 >> 1 = 0010


  -5 /  2 = ?
1011 >> 1 = 1101
          = -3
```

It rounds *lower*

# How do you round to zero?

```
Add a bias!

-5     + 1
1011   + 1 = 1100 = -4
1100 >> 1 = 1110 = -2

-4 + 1 % 2
1100 + 1 = 1101
1101 >> 1 = 1110 = -2
```

# Generalized bias for dividing negative #s

$$\left\lfloor \mathbf{u} \;/\; \boldsymbol{2^k} \right\rfloor$$

$$=$$

```
(u + (1  << k) - 1) >> k
```

Rounds to zero

# Today: Bits, Bytes, and Integers

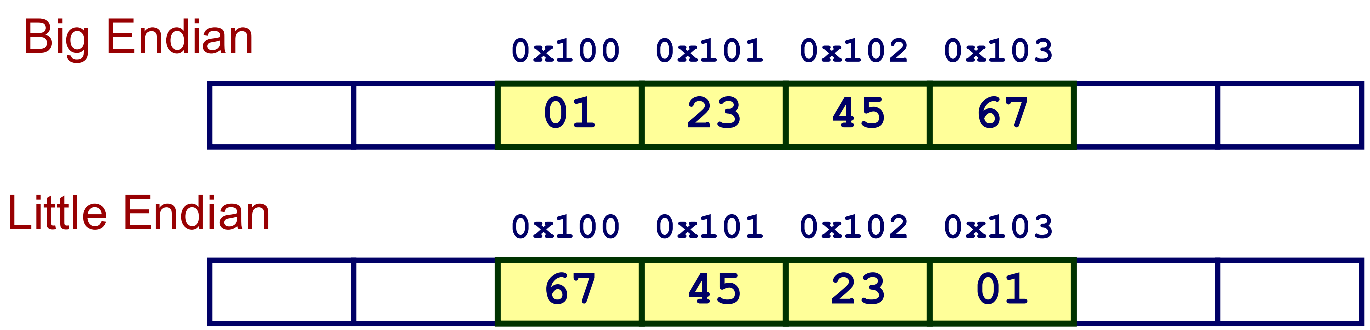
- **Representing information as bits**
- **Bit-level manipulations**
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- **Byte Ordering**

# Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**
- **Conventions**
  - Big Endian: Sun (Oracle SPARC), PPC Mac, *Internet*
    - Least significant byte has highest address
  - Little Endian: *x86*, ARM processors running Android, iOS, and Linux
    - Least significant byte has lowest address
- **Becomes a concern when data is communicated**
  - Over a network, via files, etc.
- **Important notes**
  - Bits are not reversed, as the low order bit is the reference point.
  - Doesn't affect chars, or strings (arrays of chars), as chars are only one byte

# Byte Ordering Example

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



# 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`

# Today: Bits, Bytes, and Integers

- **Representing information as bits** CSAPP 2.1
- **Bit-level manipulations**
- **Integers** CSAPP 2.2
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting CSAPP 2.3
- **Byte Ordering** CSAPP 2.1.3

**Some learning exercises:**

**https://wiredream.com/learning-to-bit-twiddle/**