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

Course Overview + From Bits to Integers

18-213/18-613

Introduction to Computer Systems

1st Lecture, Jan 18, 2022

Instructor

Greg Kesden

Scope:
(Systems) Knowledge is Power!

- **Scope**
  - Computer organization (instruction-set architecture and assembly programming, etc)
  - Software development tool chain (ABI, compilers, linkers, debuggers, etc)
  - Memory hierarchy (types of memory, locality, caching)
  - Virtual memory
  - Processes and process management
  - Exceptions, signals, and other exceptional control flow
  - Files, File systems, and File I/O
  - Networking and Network programming
  - Concurrency and concurrency control (synchronization)

Course Perspective

- **Most Systems Courses are Builder-Centric**
  - Computer Architecture: Design pipelined processor in Verilog
  - Operating Systems: Implement sample portions of operating system
  - Compilers: Write compiler for simple language
  - Networking: Implement and simulate network protocols

- **Our Course is Programmer-Centric**
  - By knowing more about the underlying system, you can be more effective as a programmer:
    - Write programs that are more reliable and efficient
    - Incorporate features that require hooks into OS
      - E.g., concurrency, signal handlers
  - There is material in this course that you won't see elsewhere
  - **We bring out the hacker in everyone!**

It's Important to Understand How Things Work

- **Why do I need to know this stuff?**
  - Abstraction is good, but don't forget reality
- **Most CS courses emphasize abstraction**
  - **(CE courses less so)**
  - Abstract data types
  - Asymptotic analysis
- **These abstractions have limits**
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
  - Sometimes the abstract interfaces don't provide the level of control or performance you need
- **Important foundation for downstream courses, industry, etc.**

Course Components

- **Lectures**
  - Higher level concepts
  - In-class quizzes could tilt you to a higher grade if borderline
- **Labs (8)**
  - 1-2+ weeks each
  - Provide in-depth understanding of an aspect of systems
  - Programming and measurement
  - Done via Autolab
- **Weekly Assignments (drop lowest two)**
  - Done via Canvas
  - Reinforce concepts
  - Take-home midterm exam counts as a double homework
- **Final Exam**
  - Test your understanding of concepts & mathematical principles
  - Covers content from the whole semester
- **Small student groups (weekly)**

Role within CS/ECE Curriculum

CS 122
Imperative
Programming

Foundation of Computer Systems
Underlying principles for hardware, software, and networking

213/513
/613

CS Systems
- 15-319 Cloud Computing
- 15-330 Computer Security
- 15-410 Operating Systems
- 15-411 Compiler Design
- 15-415 Database Applications
- 15-418 Parallel Computing
- 15-440 Distributed Systems
- 15-441 Computer Networks
- 15-445 Database Systems

ECE Systems
- 18-349 Computer Security
- 18-349 Intro to Embedded Systems
- 18-441 Computer Networks
- 18-447 Computer Architecture
- 18-452 Wireless Networking
- 18-451 Cyberphysical Systems

CS Graphics
- 15-462 Computer Graphics
- 15-463 Comp. Photography

Getting Help

- **Class Web pages:**

  http://www.cs.cmu.edu/~18213 for 18-213/18-613

  - Complete schedule of lectures, exams, and assignments
  - Copies of lectures, assignments, exams, solutions
  - FAQ
- **Piazza**
  - Best place for questions about assignments
  - We will fill the FAQ and Piazza with answers to common questions
  - Be careful about public posts: Remember the AIV policy
- **Canvas**
  - Recorded lectures
  - In-class quizzes
  - Written assignments

Getting Help

- **Email**
  - Send email to individual instructors or TAs only to schedule appointments
    - (Kesden is the exception, and you can feel free to email or call, he is sometimes hard to reach otherwise)

- **Office hours**
  - TAs: Sun-Thurs 6-10 pm
  - Instructor: https://www.andrew.cmu.edu/~gkesden/schedule.html

- **1:1 Appointments**
  - You can schedule 1:1 appointments with any of the teaching staff

Small Student Groups

- **Replaces recitation**
- **Begins next week during scheduled recitation time**
- **Goal: Descale the course, making it more personal and personally supportive**
  - Also taming office hours, which could get crazy at times in the past.
- **Groups of 5 students + 1 TA facilitator**
  - Meet for 1 hour each week (Mandatory)
  - Maintain a group chat (Slack, GroupMe, Hangouts, WeChat, whatever)
  - Try to develop a good social bond, like 5 friends going through class
    - And a TA who really knows you and how to support you
  - TAs have time reserved for helping their group members, hopefully reducing dependency upon global office hours

TA Office Hours

- **6-10pm, Sundays – Thursdays**
    - Plus special office hours for CMU-SV students at the CMU-SV campus
    - 6-8pm, Local or Remote; 8-10pm, Remote (via Zoom) only
    - Starts soon, standby for announcement
- **No queue, sign up for time in 15 minute intervals**
  - Can sign up for as many as you'd like
    - But you can only have one outstanding appointment at a time
    - Once you finish the appointment, you can make another.
  - Often times slots are immediately available
    - Even on busy days, it seems to take less than 45minutes
  - Sign-ups open at a random time in the morning.
    - Don't sign up "just in case".
- **Local: Ansys A050**
  - Need to reserve a slot; don't stalk TAs.
- **Remote: https://office-hours-01.andrew.cmu.edu:4443/**
  - Via Zoom. Link in OH Page.
  - Will be in waitroom until TA is ready

Textbooks

- **Randal E. Bryant and David R. O'Hallaron,**
  - *Computer Systems: A Programmer's Perspective*, **Third Edition** (CS:APP3e), Pearson, 2016
  - http://csapp.cs.cmu.edu
  - This book really matters for the course!
    - How to solve labs
    - Practice problems typical of exam problems
  - Electronic editions available (*Don't get paperback version*!)
  - On reserve in Sorrells Library
- **Brian Kernighan and Dennis Ritchie,**
  - *The C Programming Language*, Second Edition, Prentice Hall, 1988
  - Still the best book about C, from the originators
  - Even though it does not cover more recent extensions of C
  - On reserve in Sorrells Library

Autolab (https://autolab.andrew.cmu.edu)

- **Labs are provided by the CMU Autolab system**
  - Project page: http://autolab.andrew.cmu.edu
  - Developed by CMU faculty and students
  - Key ideas: Autograding and Scoreboards
    - **Autograding:** Providing you with instant feedback.
    - **Scoreboards:** Real-time, rank-ordered, and anonymous summary.
  - Used by over 3,000 students each semester
- **With Autolab you can use your Web browser to:**
  - Download the lab materials
  - Handin your code for autograding by the Autolab server
  - View the class scoreboard
  - View the complete history of your code handins, autograded results, instructor's evaluations, and gradebook.
  - View the TA annotations of your code for Style points.

Facilities

- **Labs will use the Intel Computer Systems Cluster**
  - The “shark machines”
  - `linux> ssh shark.ics.cs.cmu.edu`

  - 21 servers donated by Intel for 213/513/613
    - 10 student machines (for student logins)
    - 1 head node (for instructor logins)
    - 10 grading machines (for autograding)
  - Each server: Intel Core i7: 8 Nehalem cores, 32 GB DRAM, RHEL 6.1
  - Rack-mounted in Gates machine room
  - Login using your Andrew ID and password

Policies: Grading

- Labs (50%): weighted according to effort
- Final Exam (25%)
- Written Assignments (20%): drop lowest 2
- Small group participation (5%)
- Final grades based on a straight scale (90/80/70/60)

Timeliness

- Grace days
  - **5 grace days for the semester**
  - **Limit of 0, 1, or 2 grace days per lab used automatically**
  - Covers scheduling crunch, out-of-town trips, illnesses, minor setbacks
- Lateness penalties
  - Once grace day(s) used up, get penalized **15% per day**
  - No handins later than **3 days after due date (See lab page for details)**
- Catastrophic events
  - Major illness, death in family, …
  - Formulate a plan (with your academic advisor) to get back on track
- Advice
  - Once you start running late, it's really hard to catch up
  - Try to save your grace days until the last few labs

Cheating/Plagiarism: Description

- **http://www.cs.cmu.edu/~18213/academicintegrity.html**
- **What is NOT cheating?**
  - Explaining how to use systems or tools
  - Helping others with *high-level* design issues
    - High means very high
  - Using code supplied by us
    - Starter code, class examples
  - Using code from the CS:APP web site
- **Attribution Requirements**
  - Starter code: No
  - Other allowed code (course, CS:APP): Yes
  - Indicate source, beginning and end

Some Concrete Examples:

- **This is Cheating:**
  - Searching the internet with the phrase 15-213, 15213, 213, 18213, malloclab, etc.
    - That's right, just entering it in a search engine
  - Looking at someone's code on the computer next to yours
  - Giving your code to someone else, now or in the future
  - Posting your code in a publicly accessible place on the Internet, now or in the future
  - Hacking the course infrastructure
- **This is OK (and encouraged):**
  - Googling a man page for fputs
  - Asking a friend for help with gdb (but not with your code)
  - Asking a TA or course instructor for help, showing them your code, …
  - Using code examples from book (with attribution)
  - Talking about a (high-level) approach to the lab with a classmate

Cheating: Consequences

- **Penalty for cheating:**
  - Best case: -100% for assignment
    - You would be better off to turn in nothing
  - Worst case: Removal from course with failing grade
    - This is the default
  - Permanent mark on your record
  - Loss of respect by you, the instructors and your colleagues
  - If you do cheat – come clean asap!

- **Detection of cheating:**
  - We have sophisticated tools for detecting code plagiarism
  - In Fall 2015, 20 students were caught cheating and failed the course.
    - Some were **expelled** from the University
  - In January 2016, 11 students were penalized for cheating violations that occurred as far back as Spring 2014.
  - In May 2019, we gave an AIV to a student who took the course in Fall 2018 for unauthorized coaching of a Spring 2019 student. His grade was changed retroactively.

- **Don't do it!**
  - Manage your time carefully
  - Ask the staff for help when you get stuck

Why It's a Big Deal

- **This material is best learned by doing**
  - Even though that can, at times, be difficult and frustrating
  - Starting with a copy of a program and then tweaking it is very different from writing from scratch
    - Planning, designing, organizing a program are important skills
- **We are the gateway to other system courses**
  - Want to make sure everyone completing the course has mastered the material
- **Industry appreciates the value of this course**
  - We want to make sure anyone claiming to have taken the course is prepared for the real world
- **Working in teams and collaboration is an important skill**
  - But only if team members have solid foundations
  - This course is about foundations, not teamwork

How to Avoid AIVs

- **Start early**
- **Don’t rely on marathon programming sessions**
  - Your brain works better in small bursts of activity
  - Ideas / solutions will come to mind while you’re doing other things
- **Plan for stumbling blocks**
  - Assignment is harder than you expected
  - Code doesn’t work
  - Bugs hard to track down
  - Life gets in the way
    - Minor health issues
    - Unanticipated events

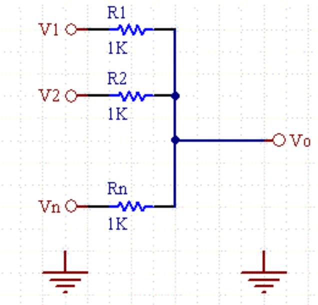
Bits, Bytes, and Integers

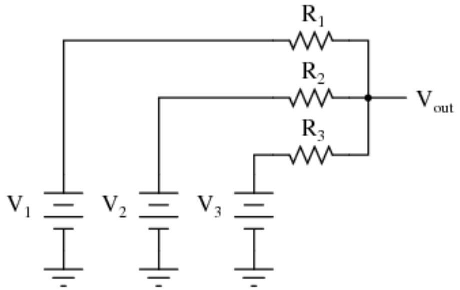
- **Representing information as bits**
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Byte Ordering

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

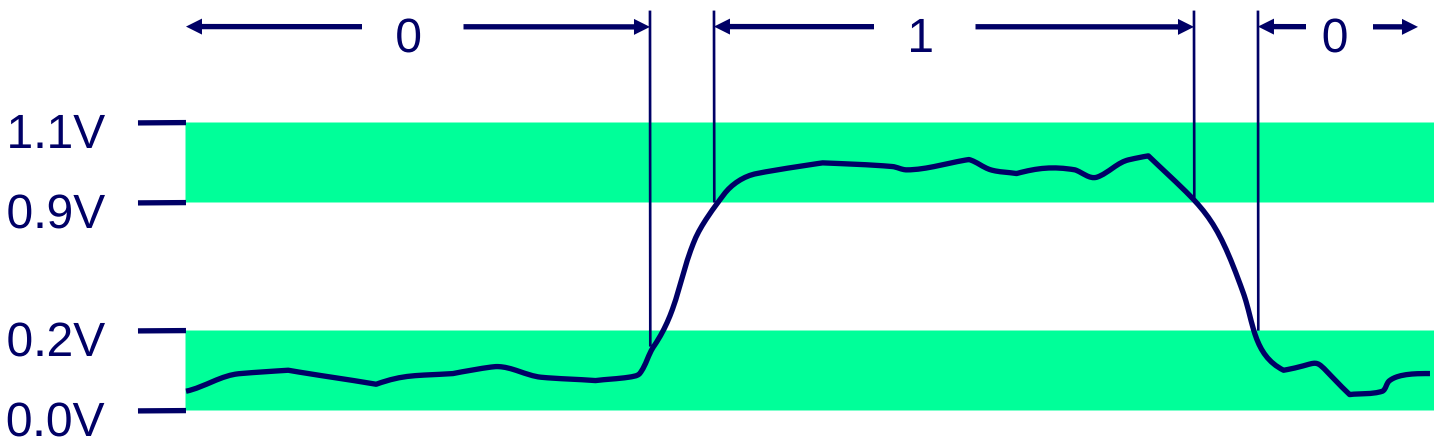
https://www.quora.com/What-is-the-most-basic-voltage-adder-circuit-without-a-transistor-op-amp-and-any-external-supply

Needing Less Accuracy, Precision is Better

- **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 leads to a simple binary, i.e. 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 10
- **By encoding/interpreting sets of bits in various ways, we can represent different things:**
  - Operations to be executed by the processor, numbers, enumerable things, such as text characters
- **As long as we can assign it to a discrete number, we can represent it in binary**

Binary Representation: Simple Numbers

- **For 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:**
    - $000 = 0*10^2 + 0*10^1 + 0*10^0$
    - $001 = 0*10^2 + 0*10^1 + 1*10^0$
    - $357 = 3*10^2 + 5*10^1 + 7*2^0$

Hexadecimal and Octal

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

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

- **Any power-of-two base provides this property**
  - Octal, e.g. base-8, and hexadecimal, e.g. base-16 are the 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

15213: 0011 1011 0110 1

3 B 6 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

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 →
    -
  - ~0x00 →
    -
  - 0x69 & 0x55 →
    -
  - 0x69 | 0x55 →
    -

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

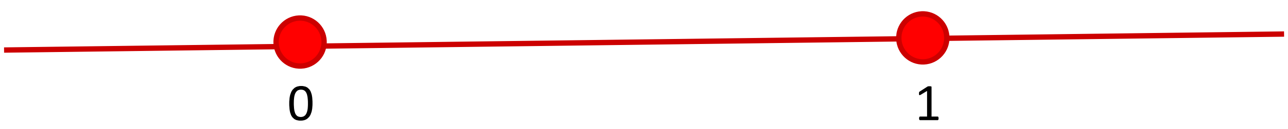
Today: Bits, Bytes, and Integers

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

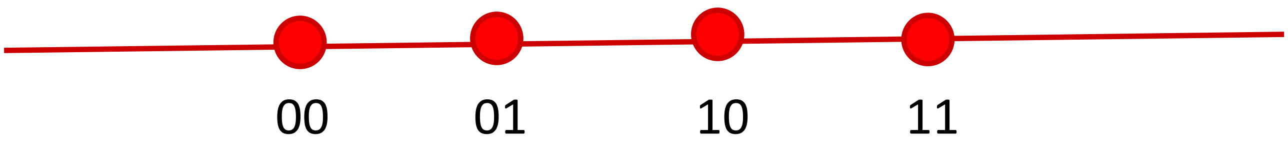
Binary Number Lines

- **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

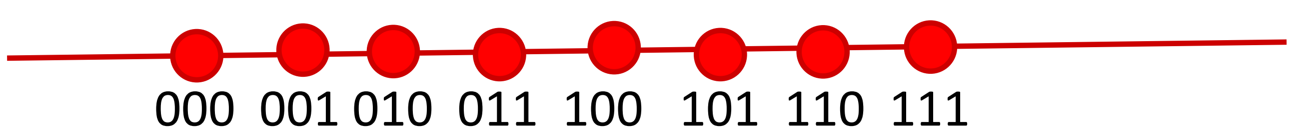
- **Consider the following examples:**
  - 1 bit number line



  - 2 bit number line

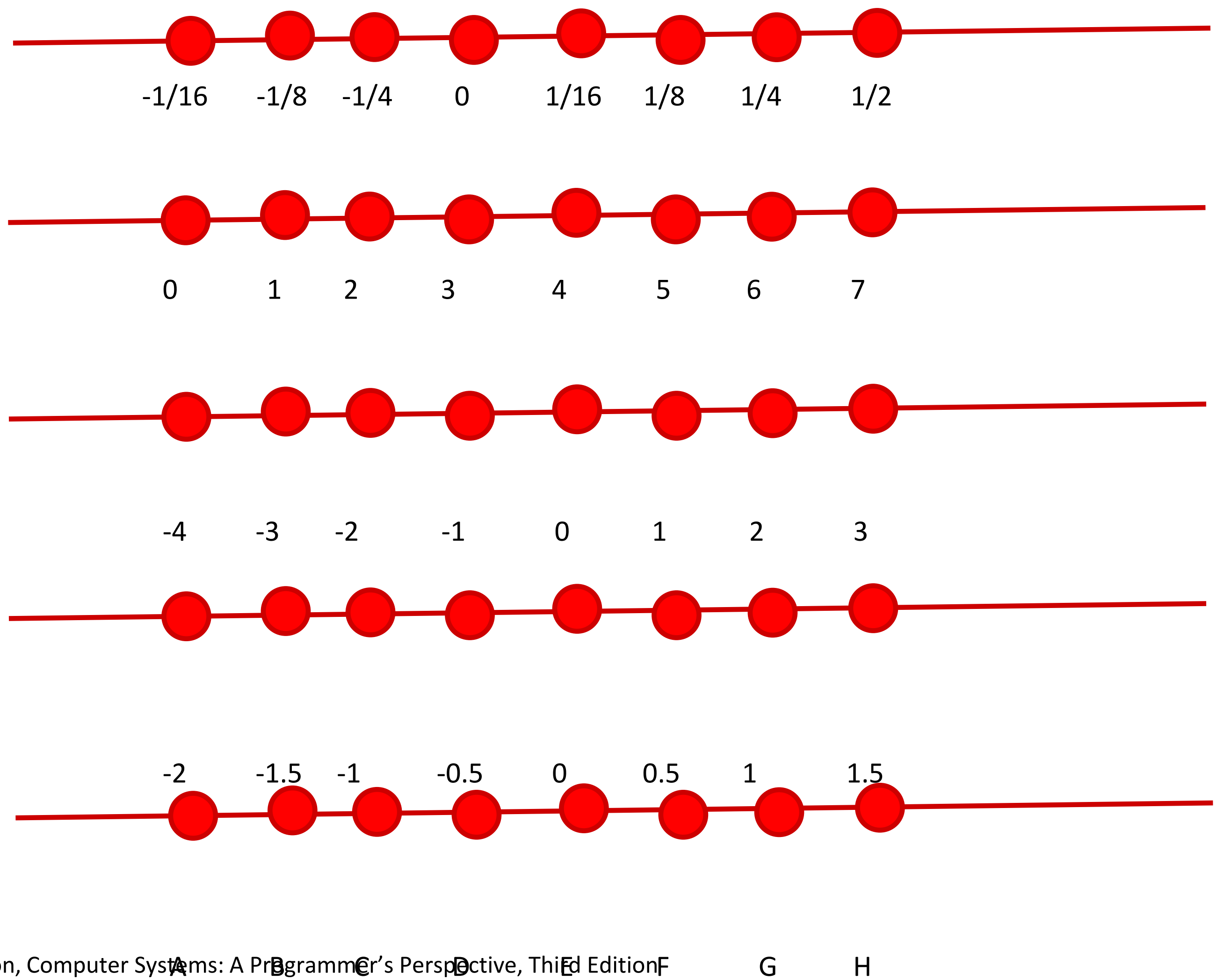


  - 3 bit number line



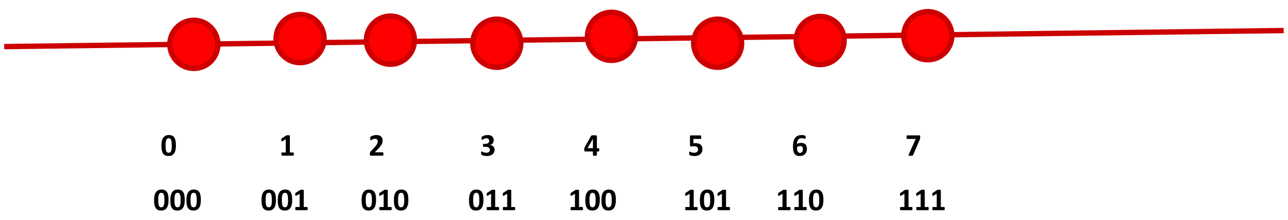
Some Purely Imaginary Examples

- 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"**

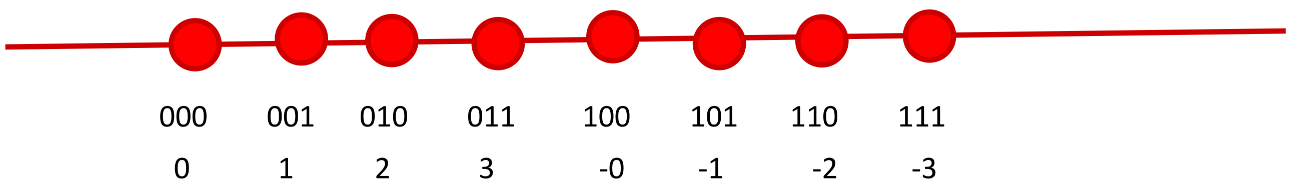
Unsigned and Non-Negative Integers

- **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 represent negative Numbers?

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



- **This has some benefits**
  - It lets us represent negative and non-negative numbers
  - 0 represents 0

- **It also has some 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 Magic Trick!

- **Let's just 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.

- **We want a situation where "-1" + 1 = 0**

- **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
- **We can 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?

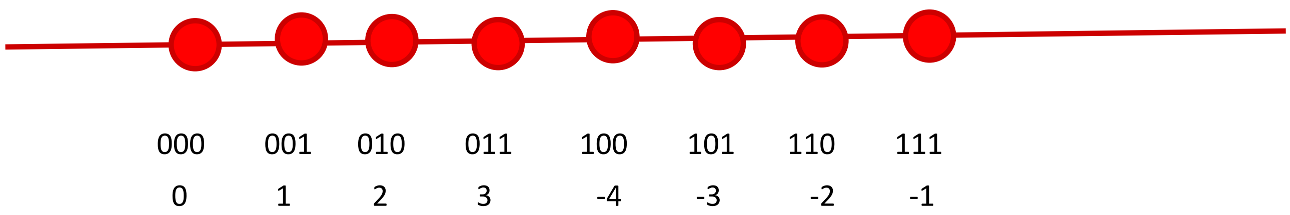
- Consider any number and its (ones) complement:
  - 0101
  - 1010
- They are called complements because complementary bits are set. As a result, if they are added, all bits are necessarily set:
  - 0101 + 1010 = 1111
- Adding 1 to the sum of a number and its complement necessarily results in a 0 due 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**



- **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

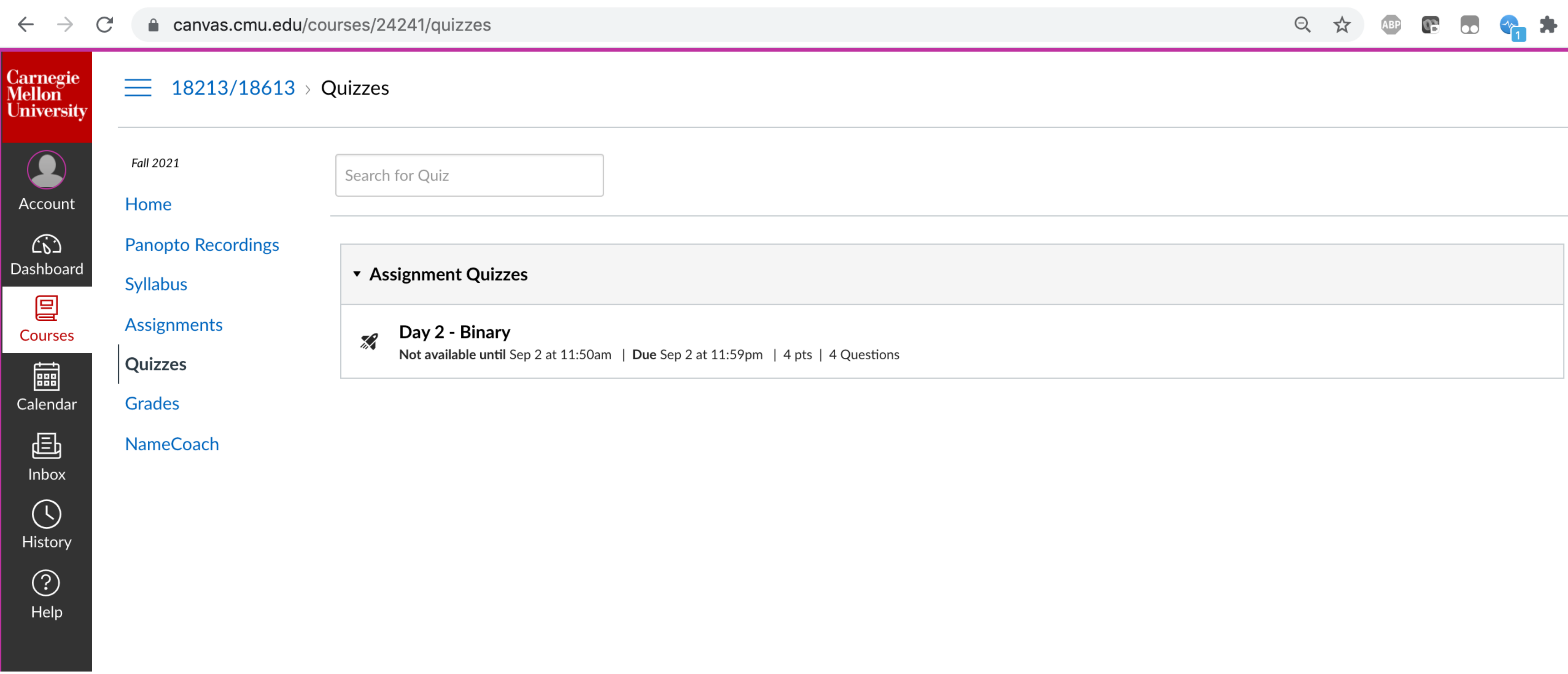
- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- **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

Quiz Time!


canvas.cmu.edu/courses/24241/quizzes
Carnegie Mellon University
Account
Dashboard
Courses
Calendar
Inbox
History
Help
18213/18613 › Quizzes
Fall 2021
Home
Panopto Recordings
Syllabus
Assignments
Quizzes
Grades
NameCoach
Search for Quiz
Assignment Quizzes
Day 2 - Binary
Not available until Sep 2 at 11:50am | Due Sep 2 at 11:59pm | 4 pts | 4 Questions

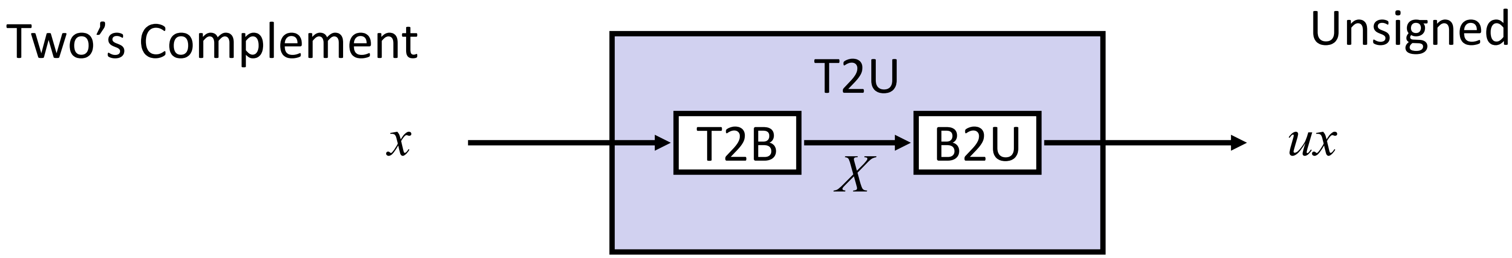
Today: Bits, Bytes, and Integers

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

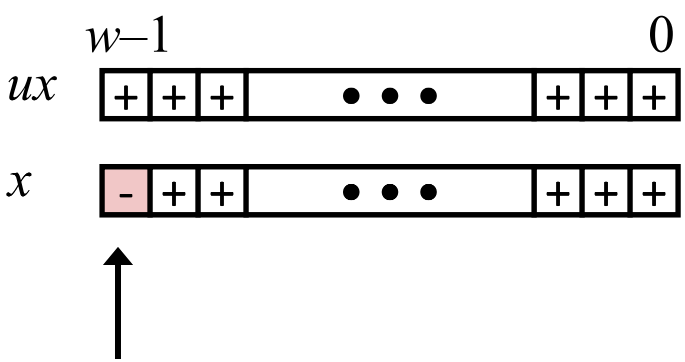
Mapping Signed ↔ Unsigned

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

Relation between Signed & Unsigned



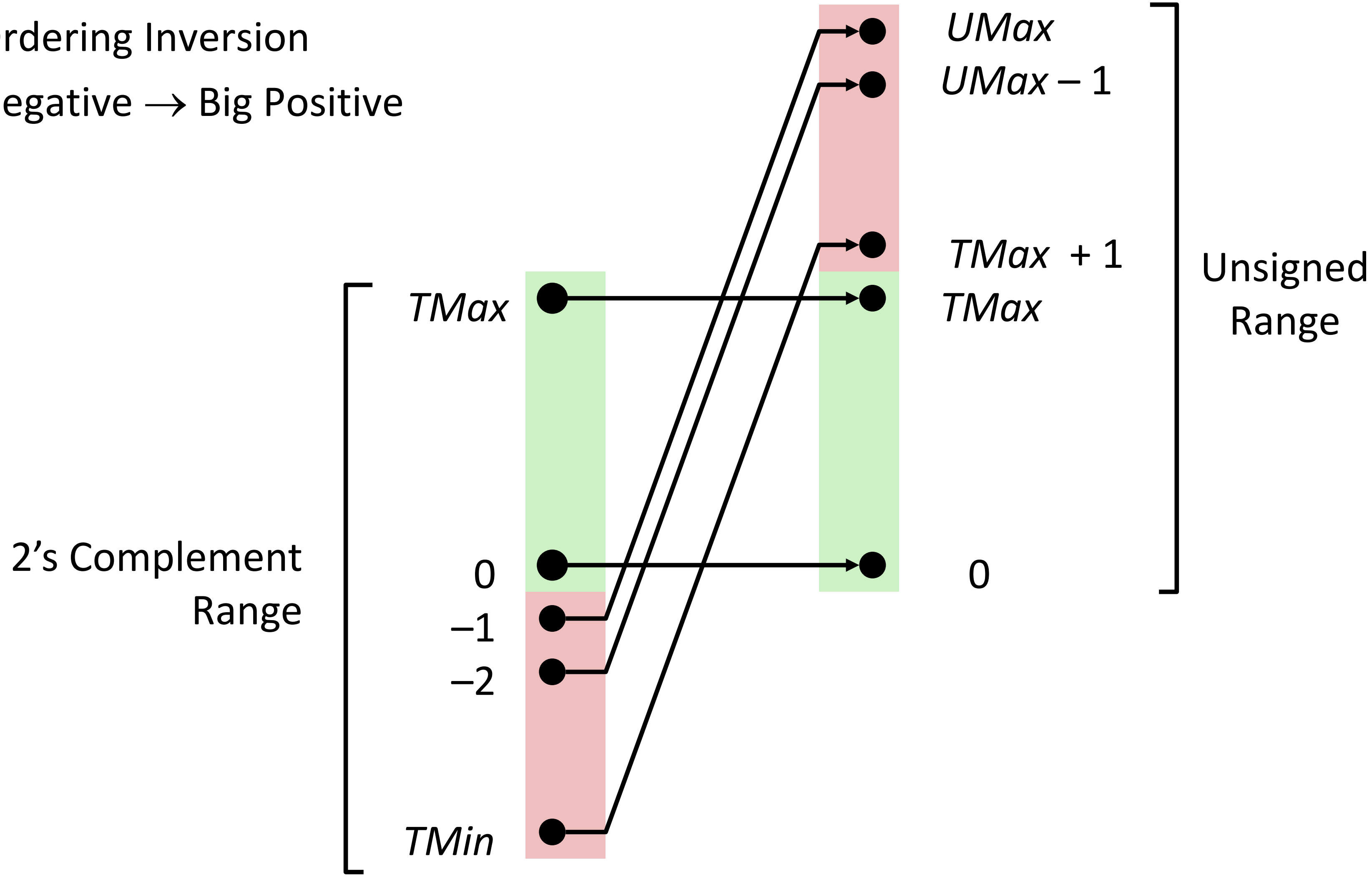
Maintain Same Bit Pattern



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;
```

```
int fun(unsigned u);
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 Ч Unsigned: Basic Rules

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

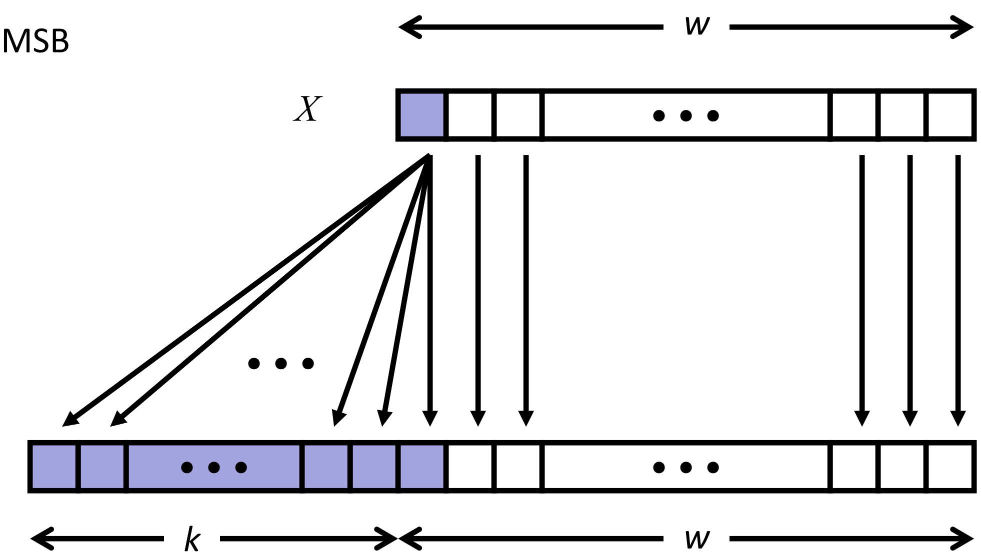
- **Expression containing signed and unsigned int**
  - `int` is cast to `unsigned`!!

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

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: Simple Example

Positive number

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

	-32	16	8	4	2	1
10 =	0	0	1	0	1	0

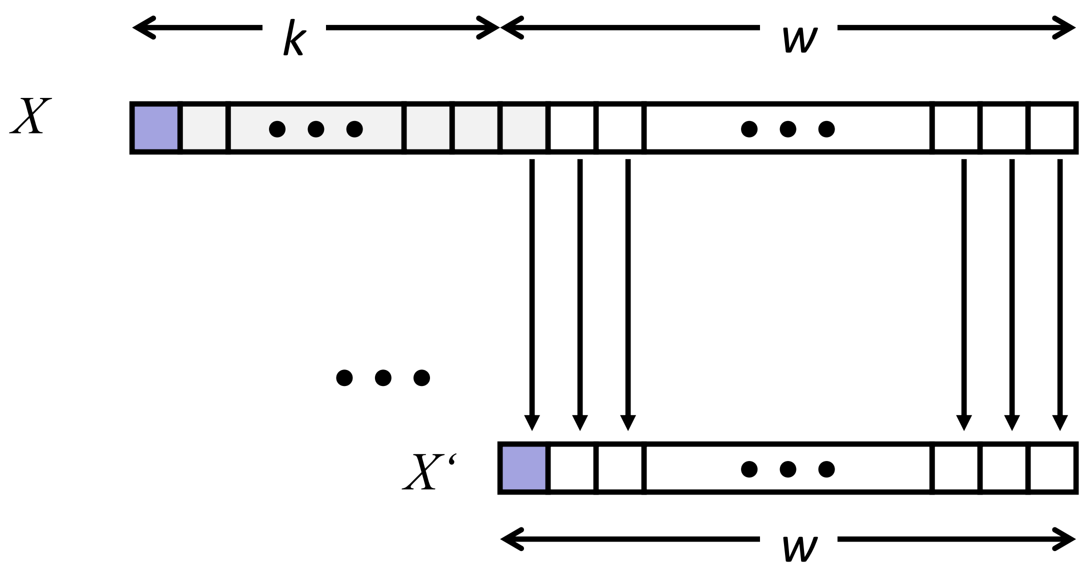
Negative number

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

	-32	16	8	4	2	1
-10 =	1	1	0	1	1	0

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

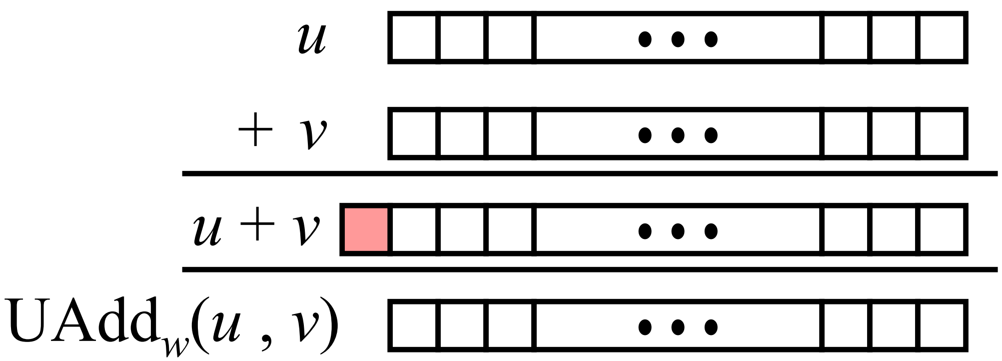
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Byte Ordering

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
```

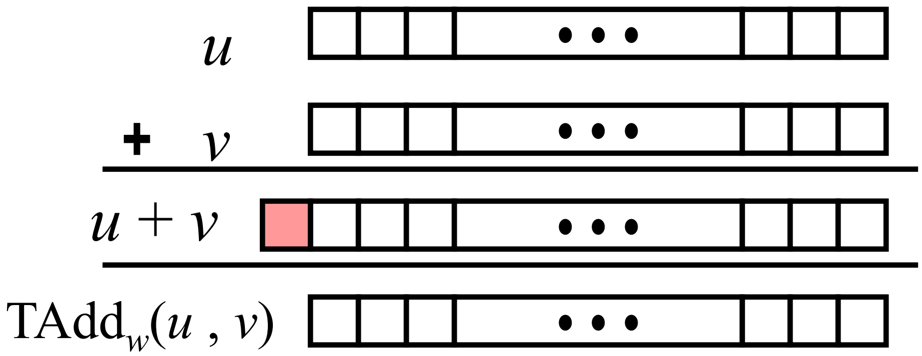
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

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
```

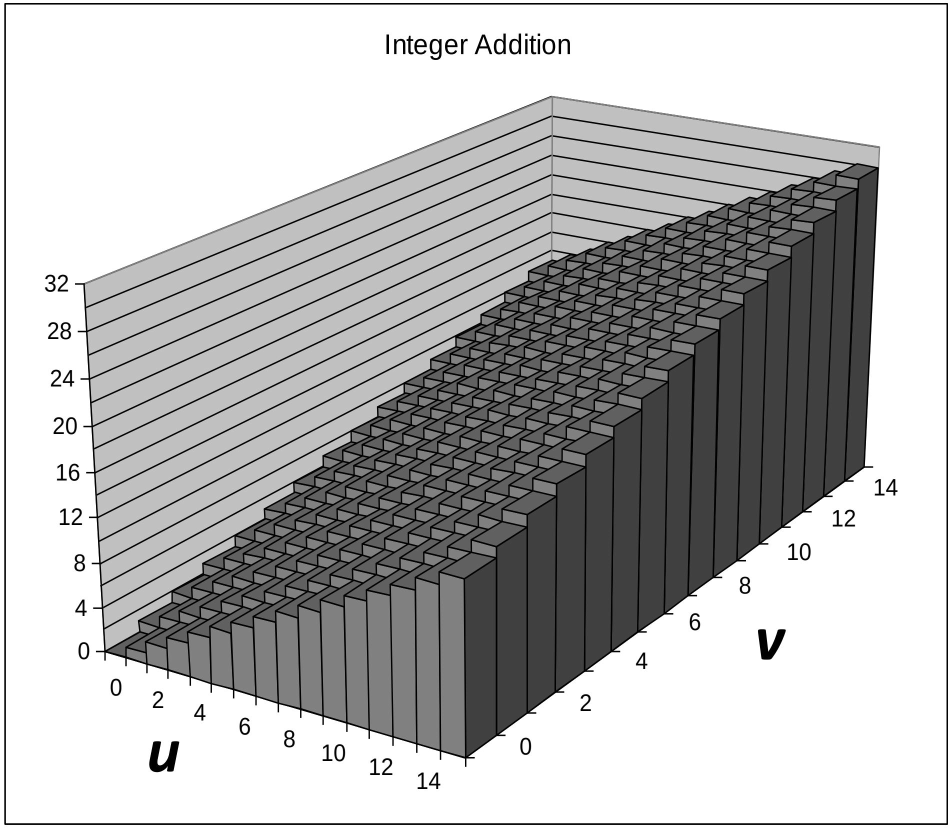
  - Will give `s == t`

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

Visualizing "True Sum" Integer Addition

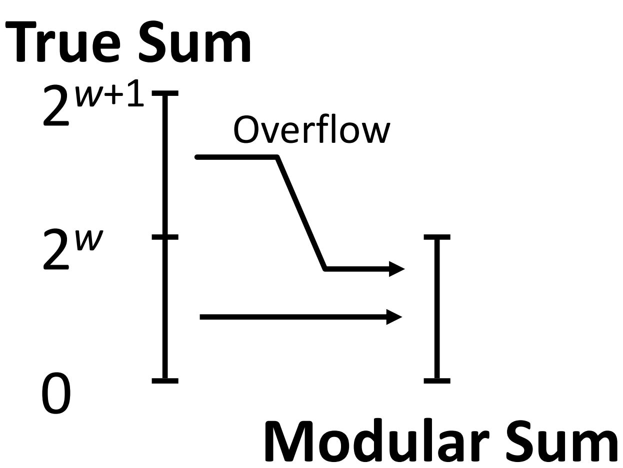
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $Add_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

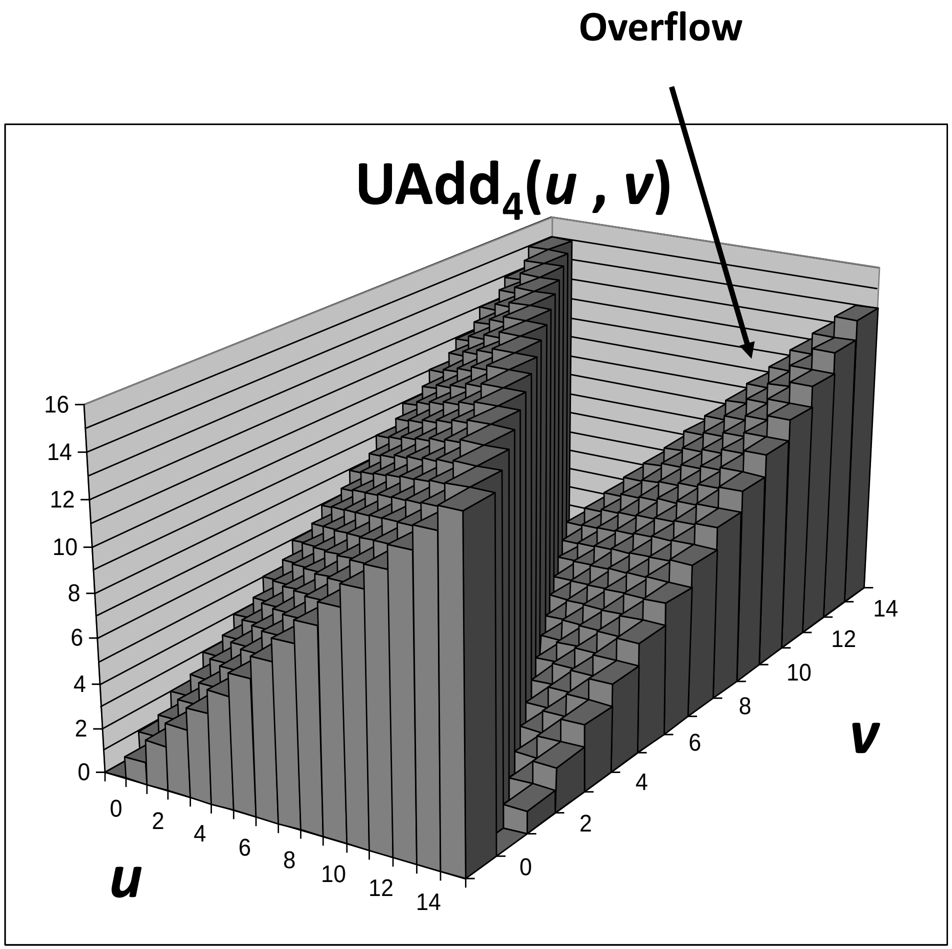
$Add_4(u, v)$



Visualizing Unsigned Addition

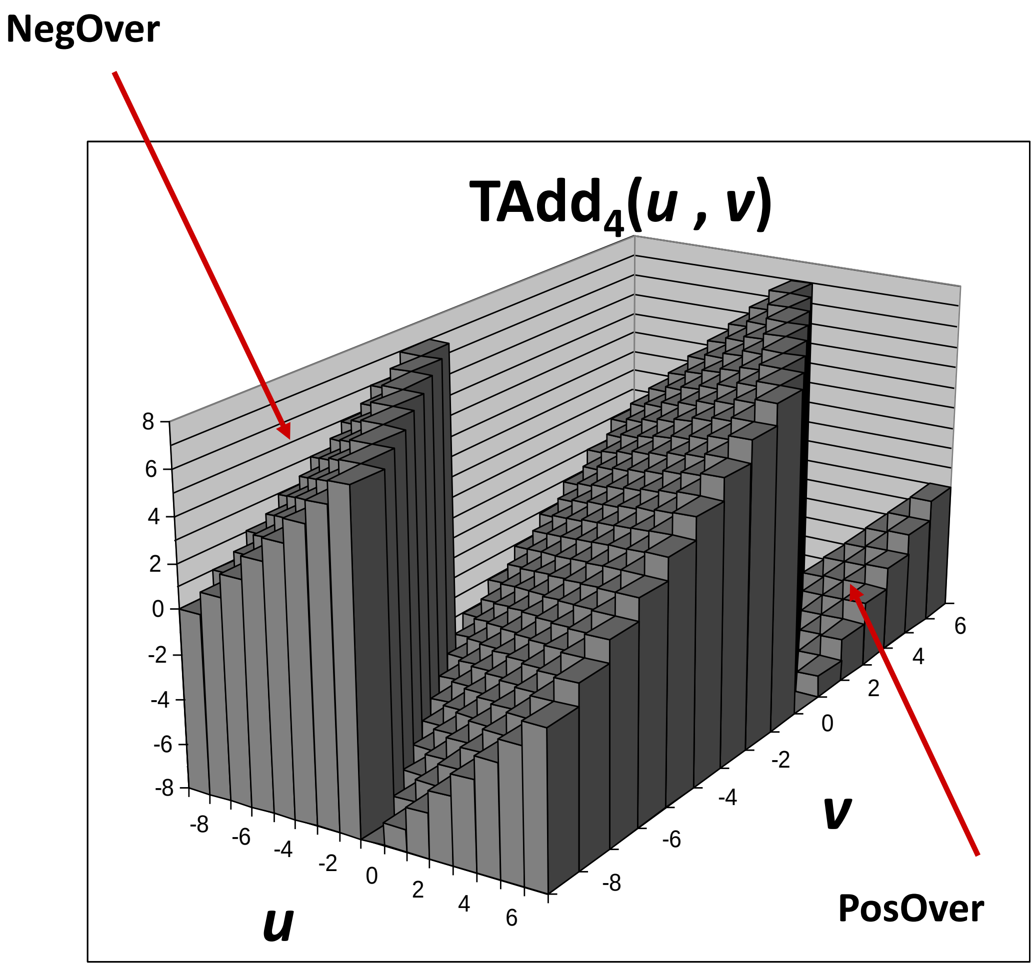
- **Wraps Around**
  - If true sum ≥ 2^w
  - At most once





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

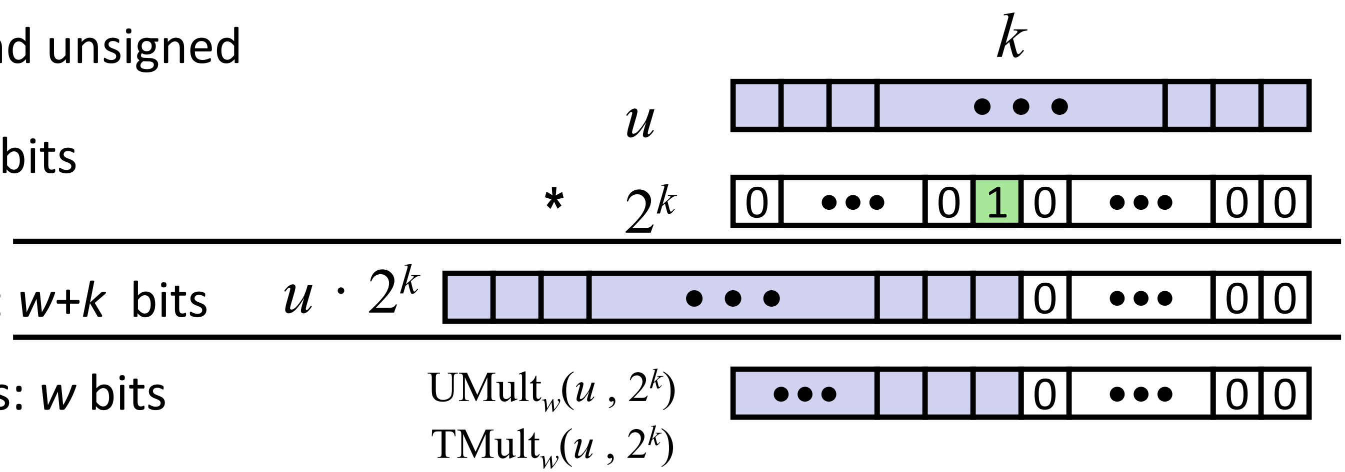
Power-of-2 Multiply with Shift

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

Operands: *w* bits

True Product: *w*+*k* bits

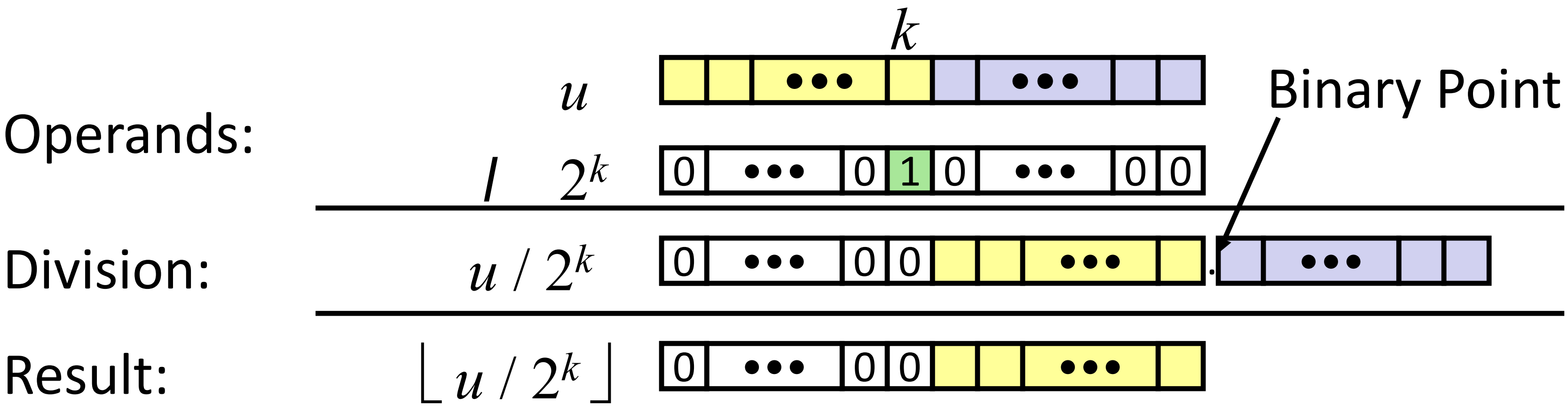
Discard *k* bits: *w* bits



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

Unsigned Power-of-2 Divide with Shift

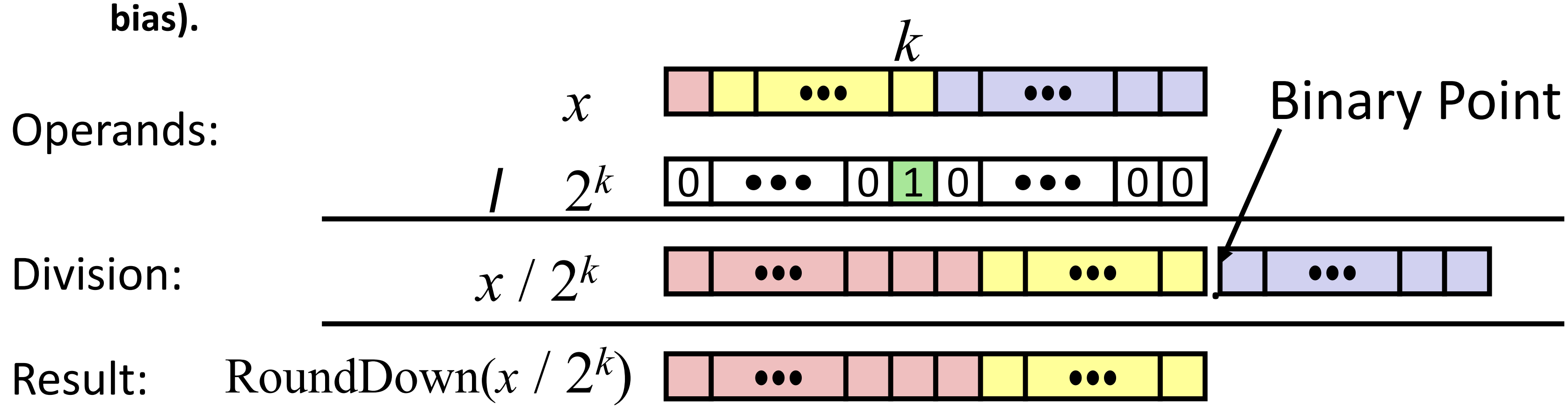
- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$
    - **Uses logical shift**



	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`

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 to the left, not towards zero (Unlikely to be what is expected, introduces a bias).**

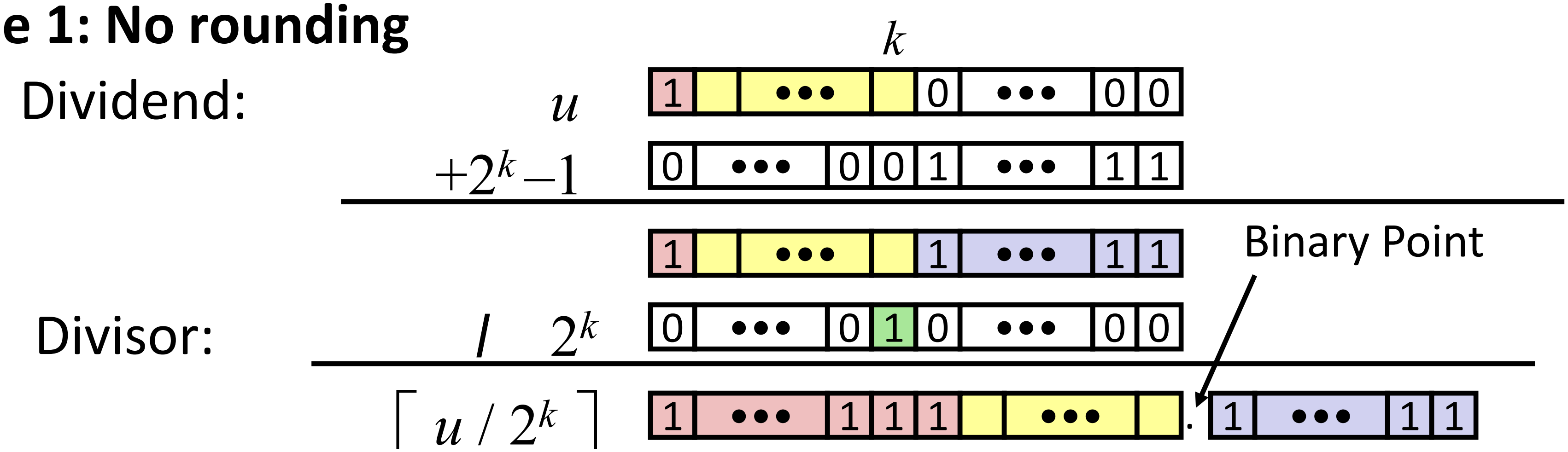


	Division	Computed	Hex	Binary
`x`	-15213	-15213	`C4 93`	`11000100 10010011`
`x >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`x >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`x >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Round-toward-0 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

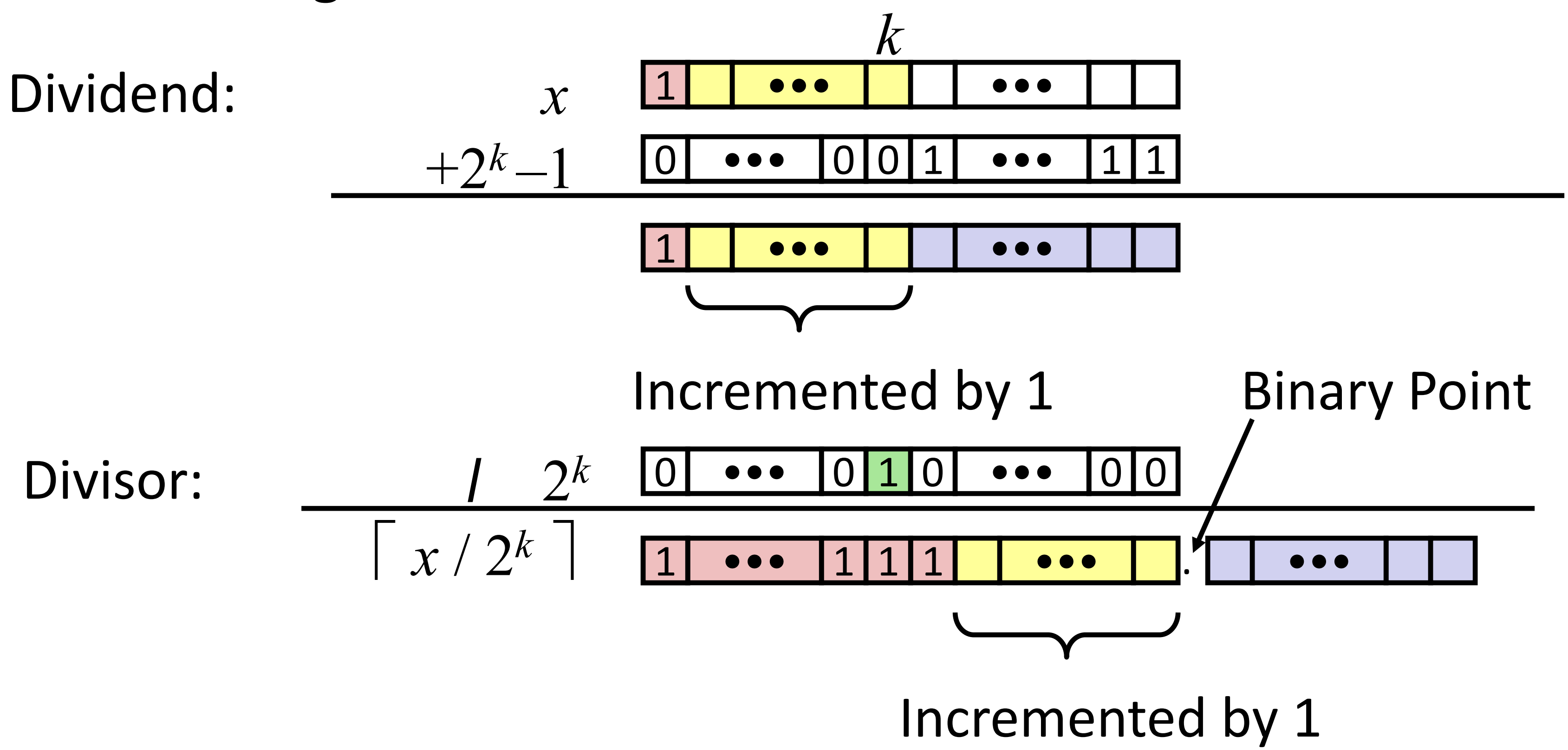
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Today: Bits, Bytes, and Integers

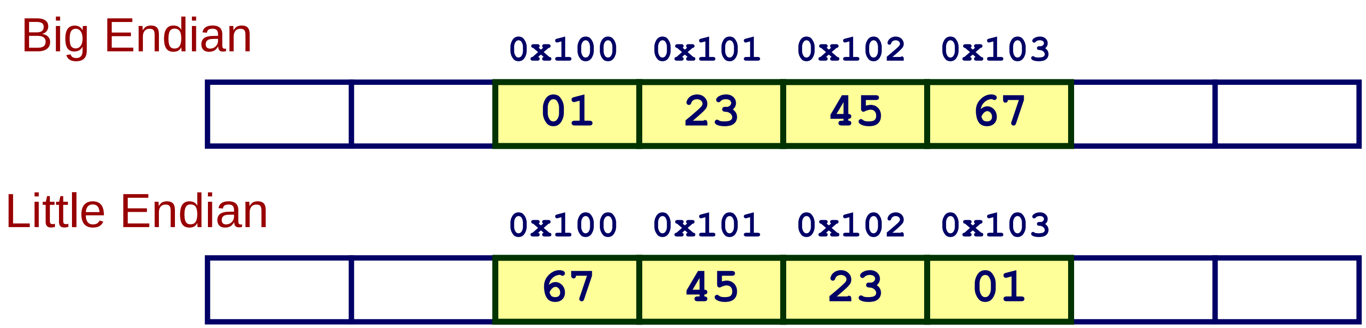
- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- **Byte Ordering**

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`

Thanks!

- **Questions?**
  - See you for office hours!
  - https://www.andrew.cmu.edu/~gkesden/schedule.html