Welcome to CS61C: Great Ideas in Computer Architecture (Machine Structure)

Instructors: Rosalie Fang, Charles Hong, Jero Wang

Rosalie

- CS + Econ ’23!
- TAed 61C since FA21
  - Did mostly project dev + OH as a TA
    - I love debugging RISC-V (Project 2)
  - But have also taught both discussion + lab!
- Instructor OH: M/Tu/W 6-7PM (In the OH room unless otherwise explained)
  - Come for conceptual questions, but also chat with me about classes, teaching, or anything in life!

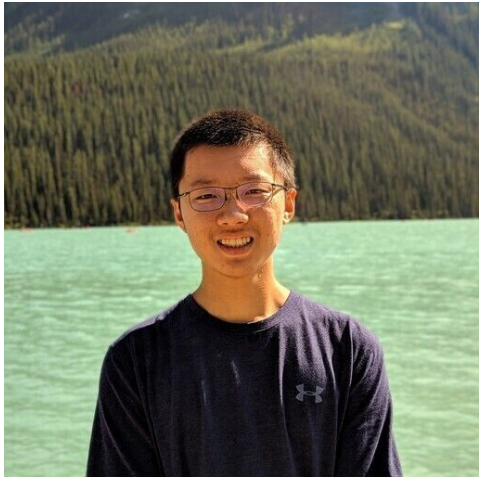
Charles

- PhD student in computer architecture
  - Building machine learning accelerators, applying machine learning and optimization to hardware design
  - Ask me about grad school/hardware research/hardware internships!
- Berkeley EECS B.S.’21
  - 6 years at Berkeley… so far
- TAed 61C twice as an undergrad
  - Also TAed 61A and 151 (digital design and FPGAs)
- Happy to chat about anything (but ideally conceptual questions) at conceptual OH: Tues/Thur 5-6pm (after lecture)

Very current photo!

Jero

- CS '24!
- Also TAed 61C since FA21
  - Did lots of project dev, office hours, and Ed
  - Summer: mostly infra and logistics
- Instructor OH: by appointment
  - Please reach out if you want to chat about anything!
  - I will also be around in the OH room on some days



Very not current photo!

We also have amazing TAs and tutors!!!

They’re going to be the ones helping you in discussions, labs, exam prep sections, and office hours!

Course Logistics

Course Format

- In-person
- A note about why we have limited remote support
    - Historically, remote learning has always been less effective. Take online OH for example, there's a lot of overhead waiting for students to join zoom, screen sharing, writing demos for conceptual questions since we can't use a physical whiteboard. A similar thing with Ed private posts: a lot of overhead goes into going back and forth between student and TA.
    - Since we have limited staff hours, we're going to try to put those hours where it creates most values for you, hence less remote support.

Communication

- Ed: https://edstem.org
  - Discussion forum for Q&A and announcements
  - For private matters, you can make a private post
    - We're going to restrict the use of private posts for debugging purposes this semester
- Course website: https://cs61c.org/su23/
  - Course schedule, lecture slides, assigned readings, and other resources are all posted here
- Email: cs61c@berkeley.edu
  - Ed response times will generally be faster, but email may be useful for private matters

Course Structure: Lectures

- You made it here, congrats!
- Monday through Thursday, 3:30-5:00 PM PT
- Lecture attendance is not taken
- We’re not planning on webcasting the content, but all lectures will be recorded and posted
    - Hopefully many questions to keep you engaged and coming to lecture in person!

Course Structure: Discussions/Labs

- Discussion
  - Smaller sections led by a TA to practice with material
  - Monday sections will cover one worksheet, and Wednesday sections will cover another worksheet
  - All discussions start this Wednesday, June 21 (tomorrow!)
  - You can attend any discussion section you want
  - In-Person Attendance
    - Required for 6/12 discussions
    - 1% of your grade
    - Each discussion is worth 0.5 course point
- Lab
  - Not required, but can be helpful
  - 2 hours long, with 1-hour conceptual mini-lecture from a TA, and 1-hour office hour style Q&A
- Discussion and lab schedule will be available at https://cs61c.org/su23/calendar/

Course Structure: Office Hours/Ed

- OH
  - Space to ask questions about content, get help with projects, raise concerns with the course, etc. with staff member
  - OH schedule will be available at https://cs61c.org/su23/calendar/
  - OH queue at https://oh.cs61c.org
  - OHs will start tomorrow (Wednesday)
  - Most OHs will be in person, but we will have 2 hours of remote OH every week
- Ed
  - Space to ask questions about content & get help with projects in a **collaborative** setting
  - We will be providing limited private Ed support this summer
    - However, public posts (and discussion) are still highly encouraged!
  - We will only answer Ed posts if you go to OH and a staff member told you to make an Ed post
    - Go to OH!!! It's so much better in terms of collaboration

Course Structure: Exams

- Midterm: tentatively Friday, July 14, 5:00–7:00 PM PT
- Final: Thursday, August 10, 3:00–6:00 PM PT
- All exams will be in person
  - Online final if you have extenuating circumstances on a case-by-case basis
- We will have an alternate exam immediately following the scheduled exam
- Clobber
  - If you score better on the final, we will use the z-score of your final exam score to replace your midterm score.

Grading breakdown

- Discussion participation: 1% for attending 6/12 discussions
- Homework: 9%
  - Try out questions as much as you'd like, and get instant feedback as you work
  - Released weekly and due on Wednesdays. Historically, each homework takes 2–4 hours on average
  - Note: HW1 and HW3 are significantly longer than other homeworks and will be worth more points!
  - Submitted on PrairieLearn
- Labs: 10%
  - Hands-on experience with course material
  - You can attend lab sections (twice a week) to work through the lab, or do them on your own
  - Submitted on Gradescope
- Projects: 40%
  - Apply course concepts on larger codebases, 1 partner allowed
  - Submitted on Gradescope
- Midterm: 14%
- Final: 26%

Class Policies: Extensions

- You can request extensions for **up to 7 days** on any assignment for any reason
  - Extensions form link coming soon
  - Reasoning for a hard deadline: summer moves extremely fast. If you are falling behind by more than a week, it is suggested to move on and catch up on new assignments
- It is okay to request an extension if things come up! We're here to support you!
  - Life happens.
  - You can always request an extension for any reason
  - Stressed out by other classes? Request an extension.
  - Not feeling well? Request an extension.

Class Policies: DSP

- Disabled Students' Program (DSP)
  - There's a variety of accommodations UC Berkeley can help us set up for you in this class
  - https://dsp.berkeley.edu/
- Are you facing barriers in school due to a disability?
  - Apply to DSP!
  - Only instructors, logistics TAs, and student support TAs can access any DSP-related info
- Our goal is to teach you the material in our course. The more accessible we can make it, the better.

Class Policies: Collaboration

- Asking questions and helping others is encouraged
  - Discussing course topics with other is welcome!
- Limits of collaboration
  - Don’t share solutions with each other (except project partners)
  - You should never see or have possession of anyone else’s solutions—including from past semesters
  - This policy always applies! Even after the deadline or after the semester!

Class Policies: Academic Misconduct

- We’re here to help! There are plenty of staff and resources available for you
  - You can always talk to a staff member if you’re feeling stressed or tempted to cheat
- Academic misconduct policies
  - At minimum, the student will receive negative points on the assignment
    - Example: If the midterm is worth 150 points, the student will receive a score of -150 on the midterm.
  - The student may be referred to the Center for Student Conduct
  - If you take the class with integrity, you don’t need to worry about these!
- Choose your partners wisely!
  - If your partner cheats, you will be flagged as well on that assignment!

Stress Management and Mental Health

- We want to reduce your stress where we can

Your health is more important than this course

- If you feel overwhelmed, there are options
  - Academically: Ask on Ed, talk to staff in office hours, set up a meeting with staff to make a plan for your success this semester
  - Non-academic:
    - Counselling and Psychological Services (CAPS) has multiple free, confidential services
      - Casual consultations: https://uhs.berkeley.edu/counseling/lets-talk
      - Crisis management: https://uhs.berkeley.edu/counseling/urgent
    - Check out UHS's resources: https://uhs.berkeley.edu/health-topics/mental-health

Stress Management and Mental Health

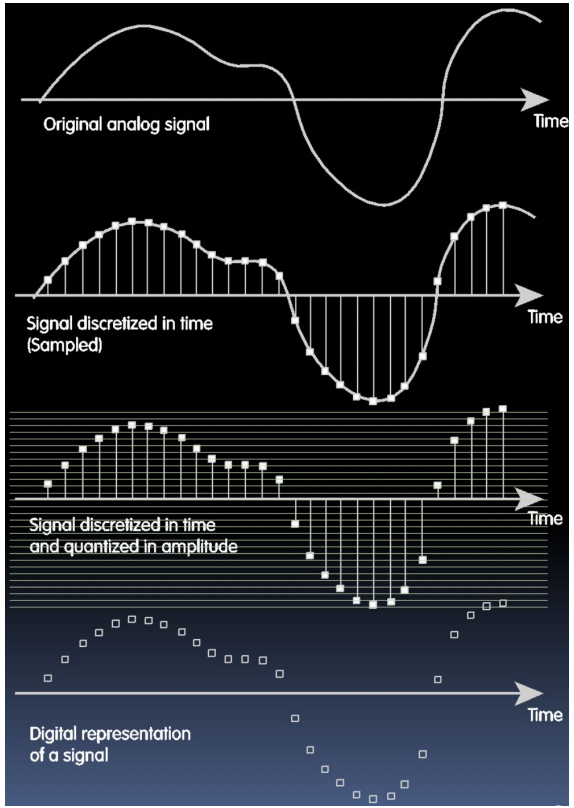
- Life happens.
  - And sometimes life is really sucky. And that's okay.
  - If you find yourself in a tough situation at any point in the semester, please don't hesitate to reach out! We will *not* judge you in any way; our top priority is just making sure you're supported.
  - We're often our own harshest critics; if you're struggling or are faced with a situation that's affecting your well-being in any way, and can't decide whether to reach out or not, *please do* if you feel comfortable doing so.
  - There are many ways to handle tough situations from a course-side perspective, ones that don't involve you having to fail a course or perform less than you believe you can!
    - Withdrawals
    - Incompletes: please keep us updated about your situation throughout the semester if this is something that you might be interested in!

Questions?

Lecture 1 - Number Representation

Data Input: Analog → Digital

- The real world is analog
- To import analog information, there are 2 things we must do
  - Sample
    - E.g., for a CD, every 44,100ths of a second, we ask a music signal how loud it is.
  - Quantize
    - For every one of these samples, we figure out where, on a 16-bit (65,536 tic-mark) “yardstick”, it lies.
  - What do we do with the quantized data?
    - Binary encoding!



BIG IDEA: Bits can represent everything

- Binary: a system of storing data using just 0 and 1
  - Everything in a computer is just bits (**bi**nary digi**ts**)
  - High voltage wire = 1, low voltage wire = 0
- Can represent just about everything
  - Numbers
  - Characters
    - 26 letters → 5 bits (32 values)
    - All the English letters & punctuations are represented by ASCII
    - Unicode: 8, 16, 32 bits
  - Logical values
    - True → 1; False → 0
  - Computer Instructions!
    - 1 + 3
    - You'll learn about them next week!
  - N bits → 2^N unique values

Decimal
Binary
Hex

Decimal System (Base 10)

Digits: 0 1 2 3 4 5 6 7 8 9

$$\mathbf{5678}_{10} = \mathbf{5} \times 10^3 + \mathbf{6} \times 10^2 + \mathbf{7} \times 10^1 + \mathbf{8} \times 10^0$$

Binary System (Base 2) → Decimal

- Digits: 0 1
- What is 1101?
- We prepend with “0b” to indicate that we’re representing a number in binary
- 0b01101 = $\mathbf{01101}_2$

  $= \mathbf{0} \times 2^4 + \mathbf{1} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0$

  $= 8 + 4 + 1$

  $= 13$

Hexadecimal System (Base 16) → Decimal

- We prepend with “0x” to indicate that we’re representing a number in hex

Digits: 0 1 2 3 4 5 6 7 8 9 $A_{(10)}$ $B_{(11)}$ $C_{(12)}$ $D_{(13)}$ $E_{(14)}$ $F_{(15)}$

0x A5 = $\mathbf{A5}_{16}$

= $\mathbf{10} \times 16^1 + \mathbf{5} \times 16^0$

= 160 + 5

= 165_{10}

Decimal → Binary

Dan's analogy: Amazon egg boxes

Start with the left

- Put **n // [column]**, and keep going with **n % [column]**
- For binary, we can just compare whether [column] < n

Convert **83** to 8-bits binary **= 0b01010011**

$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
0	1	0	1	0	0	1	1

- 83 - 64 = **19**
- 19 - 16 = **3**
- 3 - 2 = **1**
- 1 - 1 = 0

Decimal → Hexadecimal

Start with the left

- Put **n // [column]**, and keep going with **n % [column]**

Convert **83** to 4-digits hex

= 0× 53

$16^3 = 4096$	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
0	0	5	3

- 83 - 5×16 = **3**
- 3 - 3×1 = **0**

Binary ↔ Hexadecimal

- Just use the chart!
- Remember: if a number gets left padded with 0's, its value doesn't change (0b101 == 0b0101)
- Binary → Hex
  - Start from the right!
  - Treat every 4 binary digits as a group, and look up the corresponding hex digit
  - E.g. 0b1010011 = 0b 0101 0011 = 0× 53
- Hex → Binary
  - Every hex digit translates to 4 bits
  - Drop leading 0's if necessary
  - E.g. 0× 1E = 0b 0001 1110 = 0b11110

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

Useful Tables

2's power	decimal
2^0	0
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024

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

Decimal vs. Binary vs. Hexadecimal

- 8 bits
  - 1 “Byte”
  - 2 hex digits (= 256 values)
- 4 bits
  - 1 “Nibble” XD
  - 1 hex digit
- Conversions are usually time consuming, so it’s generally good to familiarize yourself with some powers of 2 and decimal→hex→binary conversion charts

Binary Mathematics

There are infinitely many integers...

- We would need infinitely many bits to represent all the integers!
- Solution #1: Treat an integer as an array of digits, extending the array as needed to save the entire integer.
  - Ex. Decimal numbers in math
  - Ex. Python's large integer primitive
  - Would require variable amounts of time to compute operations
- Solution #2: Only allow for integers within a certain range with a fixed a number of bits
  - Ex. C's integer primitive
  - Requires edge case handling when math results exceed the maximum value
  - Often referred to as an "n-bit signed/unsigned integer", where we allow numbers from 0 to 2^n-1 (ex. 8-bit unsigned int goes from 0 to 255)

Bit Operations

- Common Bitwise Operations (C syntax): &, |, ~, ^, <<, >>
- The first four correspond to the logical operators AND, OR, NOT, and XOR.
  - There's no real decimal meaning to these operations, but they tend to be useful when working with raw binary data.
  - Note that C defines ^ as XOR, not exponentiation!
- To run these operations:
  - Convert the numbers to binary
  - Run the corresponding logical operator on each pair of bits (1=TRUE, 0=FALSE)
- << and >> are left shift and right shift, respectively
  - Left shift: Convert to binary, then move all bits left, appending 0s as needed
  - Right shift (logical): Convert to binary, then move all bits right, prepending 0s as needed

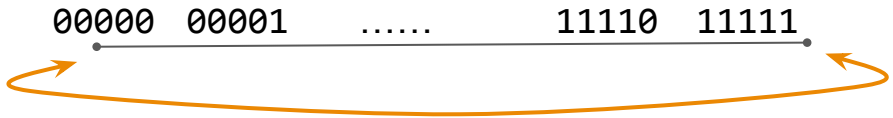
Mathematical Operators

- C also allows for mathematical operators: +, -, *, /, %, <, >, ==, etc.
  - These operators are designed to match the behavior of decimal math as much as possible, so it's always possible to do math with these operators by first converting to decimal, doing the operation, then converting back.
  - Since it takes time to convert to decimal, it's often easier to do the math entirely in binary.
- Example: 255 + 1 = ?

```
    0b      1111 1111
+   0b      0000 0001
-------------------------
    0b     10000 0000
```

What if the number is too large? Overflow!

- Limitations stem from the fact that we only can represent a finite number of integers: need to handle cases where the result of “correct” calculation yields a number outside range, ideally in a way that feels “natural”.
- Three main cases (assuming an 8-bit unsigned integer):
  - Going too high (overflow)
    - `0b1111 1111 + 0b0000 0001`
    - `0b0100 0100 * 4`
    - `0b0100 0100 << 2`
  - Going too low
    - `0b1000 0010 - 0b1000 1100`
    - (also called overflow)
  - Fractions
    - 10 / 3



Number Representation

What do we look for in number representation systems?

1. Relevance
   - Different subject matters need different value sets, so we often have several different options with various trade-offs
2. Efficiency, in terms of:
   - 2a: Memory use: As many possible bitstrings correspond to valid data values as possible
     - Ideal goal: Make it so that every bitstring corresponds to a different, valid data value
   - 2b: Operator cost: The operators we define are simple to make in circuitry, and require few transistors
     - Ideal goal: If we can reuse an existing circuit, then we don't need to add any additional circuitry!
   - Other aspects depending on the system; for now, we'll focus on these two
3. The system is intuitive for a human to understand
   - Often goes together with small operator cost, as well as adaptability to other systems
   - If a system's too complicated, no one will use it. Though if you manage to abstract away a lot of the complexity, you can get away with a less-intuitive system...

Unsigned Integer

- `0b10110010`

1	0	1	1	0	0	1	1
$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

- `0b10110010` = $128 + 32 + 16 + 2 + 1 = 179_{10}$ for an **unsigned number**
- We can't represent negative numbers using unsigned representation!
- An n-bit unsigned integer can represent integers in the range $[0, 2^n-1]$
  - Smallest: `0b00...00`
  - Largest: `0b11...11`
- Often used in programs when you don't need negative numbers
  - Ex. size or length parameters of list-like structures since you can't really have negative length

What if we want to represent negative numbers?

- Decimal numbers: negative sign (e.g. -1234)
- Obvious solution: define the leftmost bit to be the sign bit
  - Leftmost bit = 0 → number is positive
  - Leftmost bit = 1 → number is negative
- The rest of the number is the absolute value

- Consequence: We won't be able to represent as many positive numbers!
- For most signed systems, we want to be able to represent about as many negative numbers as positive numbers

Sign and Magnitude

- `0b10110010`

Sign	Magnitude						
1	0	1	1	0	0	1	1
negative	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

- `0b10110010` = -1 x (32 + 16 + 2 + 1) = -51_{10}
- An n-bit sign and magnitude integer can represent integers in the range $[-(2^{n-1}-1), 2^{n-1}-1]$
  - Smallest: `0b11...11`
  - Largest: `0b01...11`

Pros vs. Cons sign-and-magnitude

- Arithmetic circuit is complicated!
  - Ex. -1 + 2 = …-3?

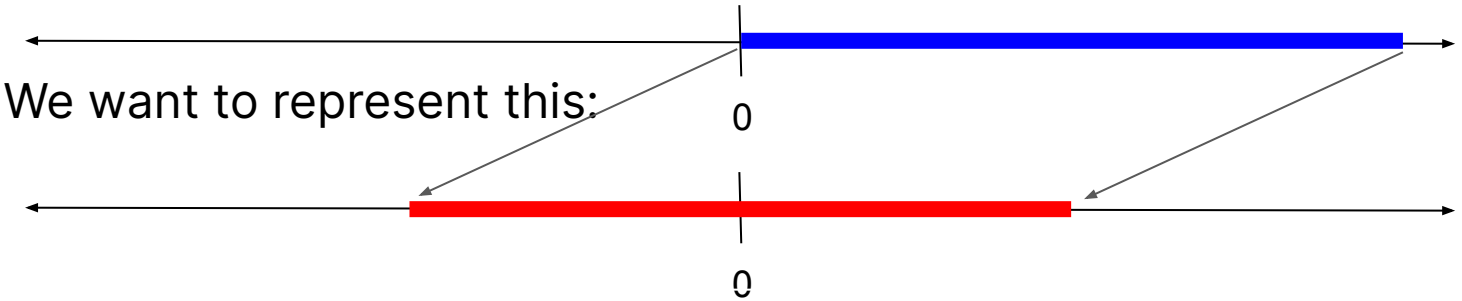
```
    0b 1000 0001
+   0b 0000 0010
-------------------
    0b 1000 0011
```

- Also, <u>two</u> zeroes
  - `0b 1000 0000 = 0b 0000 0000 = 0`
- Adding 1 to the binary number sometimes increases the decimal value by 1, and sometimes decreases it by 1

- Used only in signal processors

Another try...

- Main idea: We have a system that can represent this:



- We want to represent this:
- “Shift” the numbers so they center on 0
- Formally
  - Define a bias (usually negative)
  - To store a value in binary: subtract the bias, then store the resulting number as an unsigned value
  - To interpret a stored binary value: read the data as unsigned value, add the bias

Example: Bias notation

- We still want roughly equal number of positive and negative numbers
- For an 8-bit integer, the bias is -127 unless otherwise specified
- Interpret `0b10110010` in bias notation:
  - `0b10110010` = 179 if unsigned
  - Add -127 → 52
- If we want to store the number -27
  - -27 - (-127) = 100
  - Store 100 as an unsigned integer → 0b0110 0100
- An n-bit biased integer can represent integers in the range [bias, 2^n-1+bias]
  - Smallest: `0b00...00`
  - Largest: `0b11...11`

Pros vs. Cons: bias notation

- Arithmetic is complicated
  - Ex. -127 + 128 = …128?

```
    0b 0000 0000
+   0b 1111 1111
-------------------
    0b 1111 1111
```

- It’s inconvenient to have to keep track of the extra parameter
- Mainly used when data is only intended to be compared, not to be added/subtracted
- Really useful when the range of common values isn’t centered on zero
  - Real world example: The temperatures we see day-to-day often stay in the range of 273 K to 323 K. As such, we often think of temperatures with a bias of +273; this forms the Celsius system. Most of the time, we don’t “add” temperatures together, and just think in terms of “hotter” or “colder”. Only in chemistry/scientific fields do we often add or multiply temperatures; this is why Kelvin tends to be preferred in scientific environments.

Try again...!

- Main idea: When working with unsigned numbers, we noted that overflows got handled by taking the results mod 2^n.
- Why not handle negative numbers the same way?
- In binary:
  - If the leading bit of the number is 0, interpret the number as an unsigned integer (positive numbers)
  - If the leading bit of the number is 1, interpret the number as an unsigned integer, then subtract 2^n (negative numbers)
- Result: The 2's complement value of a bitstring is always equivalent to the unsigned value (mod 2^n)

Two’s Complement

- `0b10110010`

1	0	1	1	0	0	1	1
$-2^7 = -128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

- `0b10110010` = -128 + 32 + 16 + 2 + 1 = -77_{10}

- Conversion for a positive integer (one that’s in range) is exactly like for unsigned
  - 11, for example, is `0b0000 1011` in both unsigned and two’s complement implementations

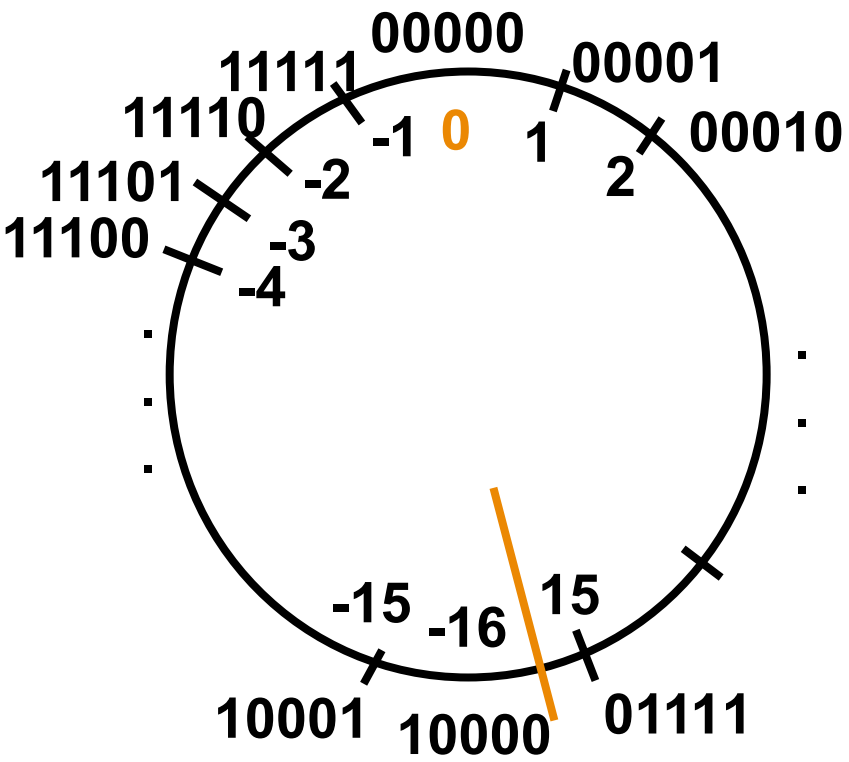
Two’s Complement SHORTCUT

- For **negative numbers only**
- Decimal → Binary
  - Flip the sign
  - Convert to binary as if unsigned, pad the left with 0’s to make n bits
  - Flip all the bits (0→1, 1→0)
  - Add 1 to the binary representation
- Binary → Decimal
  - Flip all the bits
  - Add 1 to the binary representation
  - Convert to decimal
  - Flip the sign

- Ex. -20 to binary
  - Flip the sign: 20
  - 20 = `0b0001 0100`
  - Flip the bits: `0b1110 1011`
  - Add 1: **`0b1110 1100`**

Two's Complement number "line", 5 bit

- An n-bit unsigned integer can represent integers in the range $[-2^{n-1}, 2^{n-1}-1]$
  - Smallest: `0b10…00`
  - Largest: `0b01…11`
- Useful to remember:
  - `0b11…11` = -1
- One more negatives than positives
- Arithmetic is easy!
  - We can just reuse the arithmetic circuit for unsigned numbers
- Generally the preferred signed implementation



In summary...

- We represent “things” in computer science as particular bit patterns: n bits → 2^n things
- The 4 different representations have their own benefits and uses
  - Unsigned
  - Sign and Magnitude: has most problems
  - Bias notation
  - Two’s complement: most widely used implementation for signed integers