CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 2: Number Representations

Instructors: Dan Garcia, Justin Yokota

Computer Architecture in the News

Note: Slides with a green background are examples, or further explanations. They are not considered in scope for this course.

Agenda

- Binary and Hexadecimal
- Representing Data using Binary
  - ASCII
- Binary Mathematics
- Integer representations
  - Unsigned numbers
  - Sign-Magnitude
  - Bias
  - Two’s Complement

Agenda

- **Binary and Hexadecimal**
- Representing Data using Binary
  - ASCII
- Binary Mathematics
- Integer representations
  - Unsigned numbers
  - Sign-Magnitude
  - Bias
  - Two's Complement

Binary

- A system of storing data using just two digits: 1 and 0.
- Everything in a computer is ultimately stored in binary (high voltage wire = 1, low voltage wire = 0)
- Generally rooted in the mathematical concept of binary (as a base 2 system of representing numbers)

Analogous System: Decimal

- A system of storing numbers using ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.
- Numbers get written using multiple digits, with every integer having a unique representation: If I write “5678”, this means:
- $5 \times 10^3 + 6 \times 10^2 + 7 \times 10^1 + 8 \times 10^0$

10^3	10^2	10^1	10^0
5	6	7	8

Binary

- A system of storing numbers using two digits: 0, 1.
- Numbers get written using multiple digits, with every integer having a unique representation: If I write “0b1010”, this means:
- $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
- Note: Since we only ever multiply by 1 or 0, we can just write the powers of 2 that have 1s in them:
  - 0b1010 = $2^3 + 2^1$
  - As a corollary, every integer can be uniquely expressed as a sum of powers of 2.

Binary

- A system of storing data using just two digits: 1 and 0.
- Everything in a computer is ultimately stored in binary (high voltage wire = 1, low voltage wire = 0)
- Generally rooted in the mathematical concept of binary (as a base 2 system of representing numbers)
- Since computers tend to “think” in binary, it is ultimately useful to work with values in binary. By convention we prepend any binary value with “0b”
    - Ex. 0b11 means 3, while 11 means 0b1011

Challenges with using Binary

- Conversion from binary to decimal and back is generally time consuming
- Ex. To convert 83 to binary:
    - 83 is greater than $2^6 = 64$, but less than $2^7 = 128$
    - $83 = 1 \times 2^6 + 19$
    - $83 = 2^6 + 2^4 + 3$
    - $83 = 2^6 + 2^4 + 2^1 + 1$
    - $83 = 2^6 + 2^4 + 2^1 + 2^0$
    - 83 = 0b1010011
- As such, it is generally a good idea to have memorized a few binary-to-decimal conversions:
    - Numbers 0 through 15
    - Select powers of 2
    - Approximations for various powers of 2

Useful Numbers to Memorize

Powers of 2

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
2^{11}	2048
2^{12}	4096
2^{13}	8192
2^{14}	16384
2^{15}	32768
2^{16}	65536

Decimal	Binary
0	0b0000
1	0b0001
2	0b0010
3	0b0011
4	0b0100
5	0b0101
6	0b0110
7	0b0111

Decimal	Binary
8	0b1000
9	0b1001
10	0b1010
11	0b1011
12	0b1100
13	0b1101
14	0b1110
15	0b1111

Binary SI-prefixes

- Note: $2^{10} = 1024$ is extremely close to 1000. There's absolutely no good reason for this; the numbers just happened to line up nicely
- On the plus side, this means that we can approximate large powers of 2 in decimal
  - Ex. $2^{32} = 2^{30} \times 2^2$
  - $= (2^{10})^3 \times 4$
  - $\approx 4 \times (1000)^3$
  - $\approx$ 4 billion
  - Actual value is 4,294,967,296, ~7% off
- This also gets used in binary SI prefixes
- "Kibi, Mebi, Gibi"... instead of "Kilo, Mega, Giga"
  - Ex. 512 TiB = 512 Tebibytes = 2^{49} bytes.

Binary

- A system of storing data using just two digits: 1 and 0.
- Everything in a computer is ultimately stored in binary (high voltage wire = 1, low voltage wire = 0)
- Generally rooted in the mathematical concept of binary (as a base 2 system of representing numbers)
- Since computers tend to “think” in binary, it is ultimately useful to work with values in binary. By convention we prepend any binary value with “0b”
- Generally too unwieldy to use directly, so binary data is often written using hexadecimal or octal
  - Binary numbers use almost 3 times as many digits as decimal, and it’s painful to read long binary strings, such as: 0b0101 0000 0110 0001 0110 1001 0110 1110
  - By convention, we often add spacing to long digit strings to help read them

Hexadecimal

- A system of storing numbers using 16 digits: 0123456789ABCDEF
- Numbers get written using multiple digits, with every integer having a unique representation: If I write “0x87”, this means:
- $8 \times 16^1 + 7 \times 16^0$

16^1	16^0
8	7

Converting Hexadecimal to Binary

- Base 16 is specifically chosen because $16 = 2^4$
- This makes conversion to binary very easy:
  - $0x87 = 8 \times 16^1 + 7 \times 16^0$
  - $= 0b1000 \times 2^4 + 0b0111 \times 2^0$
  - = 0b1000 0000 + 0b0111
  - = 0b1000 0111
- Conclusion: To convert from hexadecimal to binary:
  - Convert each digit separately to binary
  - Concatenate the results
- To convert from binary to hexadecimal
  - Split the data into sets of 4 digits (prepending 0s to get a multiple of 4 digits)
  - Convert each set of 4 digits into its corresponding hex digit
- Octal is Base 8, which does the same, but using 3 bits at a time instead.

Hexadecimal

- In this course, hex is ALWAYS used as a proxy for binary; we write a hex value for brevity and readability, but the intent is that of the underlying binary.
- Hexadecimal values are prepended with the “0x” prefix. All other bases not mentioned don’t have a specific prefix, but instead have the base written as a subscript.
    - Ex. 0x14 == 0b10100 == 20
    - Ex. 43_5 (or 43_5) == $4 \times 5^1 + 3 \times 5^0$ == 23
- Hex to decimal or decimal to hex is about as difficult as binary to decimal.

Agenda

- Binary and Hexadecimal
- **Representing Data Using Binary**
  - ASCII
- Binary Mathematics
- Integer representations
  - Unsigned numbers
  - Sign-Magnitude
  - Bias
  - Two's Complement

Representing Data using Binary

- What we’ve discussed so far is the definition of binary from a mathematical perspective
- From the computer’s perspective, all it sees is a cluster of wires, some of which are high voltage, and some of which are low voltage.
    - When we ask a computer to “do math”, all the computer sees is a set of wires whose values get “transformed” by being sent through a bunch of transistors. The computer doesn’t inherently know the meaning assigned to that data, so it doesn’t even know that it’s computing something.
- As such, whenever we deal with data in a computer, it’s up to the human creating the computer to:
    - Assign some meaning to each binary state
    - Put together some transistors so that the transformation induced acts equivalently to meaningful operations, ideally with intuitive edge case handling.

Representing Data using Binary

- Anything that we put in memory gets stored using some number of bits (e.g. a 1-bit boolean, an 8-bit integer)
- Each possible sequence of 1s and 0s (bitstring) can get assigned to at most one meaning
- In general, if we use n bits in our data object, we can represent up to 2^n different bitstrings, and therefore 2^n different meanings
    - 2 choices for the first bit × 2 choices for the second bit …

Example: Boolean Values

- Only two possible states: TRUE and FALSE
- Can be stored in 1 bit
  - Assign “1” to mean TRUE, and “0” to mean FALSE
- Meaningful operators: OR, AND, NOT, XOR

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

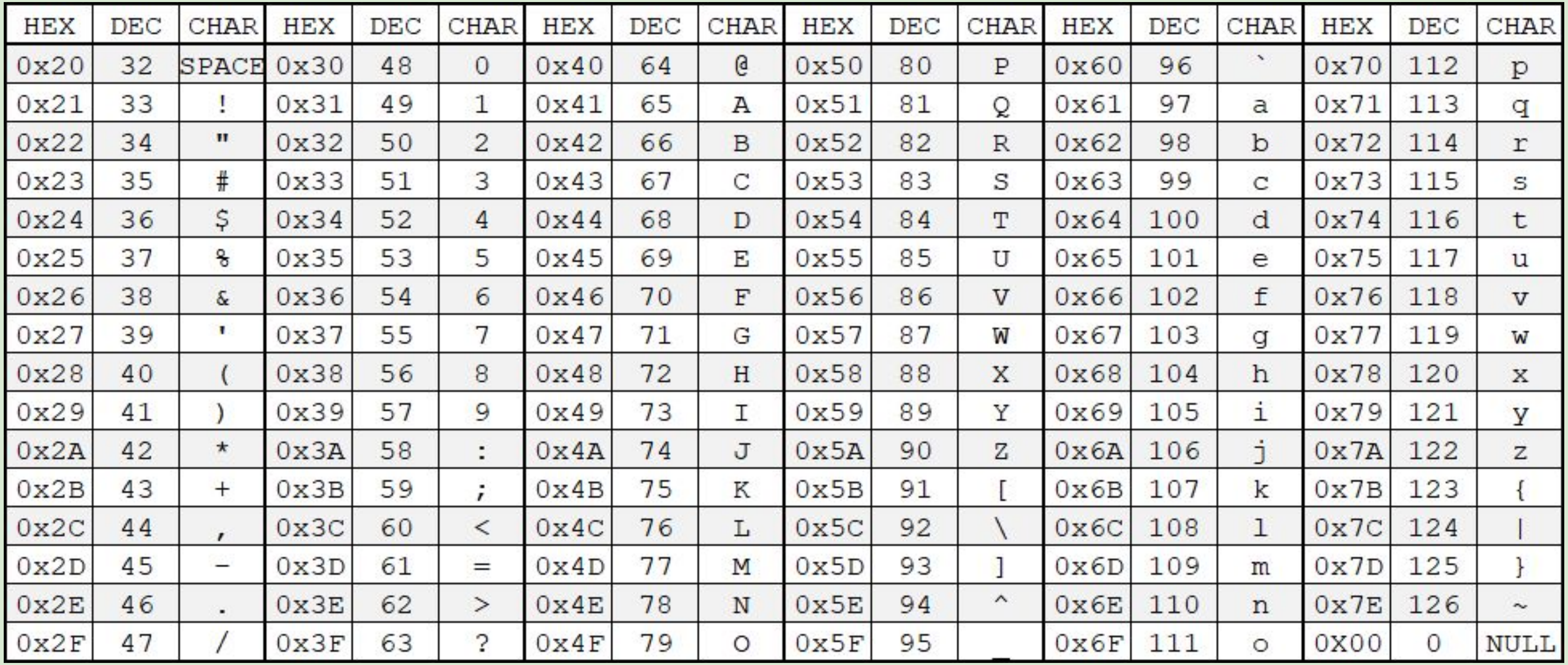
x	NOT x
0	1
1	0

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Example: ASCII

- There’s no “natural” way to assign bitstrings to characters
- Several options exist to map characters to bits
  - ASCII, UTF-8, UTF-32, Morse Code, Braille
  - For this class, we'll focus on ASCII
- In ASCII standard, each character uses 7 bits of data
- The 2^7=128 different bitstrings correspond to 128 common meanings; lowercase letters, uppercase letters, numbers, punctuation, spacing, and control characters
- The next slide will have a shortened version of the ASCII table; this will be provided whenever ASCII translations are needed.

ASCII table



HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR
0x20	32	SPACE	0x30	48	0	0x40	64	@	0x50	80	P	0x60	96	`	0x70	112	p
0x21	33	!	0x31	49	1	0x41	65	A	0x51	81	Q	0x61	97	a	0x71	113	q
0x22	34	"	0x32	50	2	0x42	66	B	0x52	82	R	0x62	98	b	0x72	114	r
0x23	35	#	0x33	51	3	0x43	67	C	0x53	83	S	0x63	99	c	0x73	115	s
0x24	36	$	0x34	52	4	0x44	68	D	0x54	84	T	0x64	100	d	0x74	116	t
0x25	37	%	0x35	53	5	0x45	69	E	0x55	85	U	0x65	101	e	0x75	117	u
0x26	38	&	0x36	54	6	0x46	70	F	0x56	86	V	0x66	102	f	0x76	118	v
0x27	39	'	0x37	55	7	0x47	71	G	0x57	87	W	0x67	103	g	0x77	119	w
0x28	40	(	0x38	56	8	0x48	72	H	0x58	88	X	0x68	104	h	0x78	120	x
0x29	41	)	0x39	57	9	0x49	73	I	0x59	89	Y	0x69	105	i	0x79	121	y
0x2A	42	*	0x3A	58	:	0x4A	74	J	0x5A	90	Z	0x6A	106	j	0x7A	122	z
0x2B	43	+	0x3B	59	;	0x4B	75	K	0x5B	91	[	0x6B	107	k	0x7B	123	{
0x2C	44	,	0x3C	60	<	0x4C	76	L	0x5C	92	\	0x6C	108	l	0x7C	124	|
0x2D	45	-	0x3D	61	=	0x4D	77	M	0x5D	93	]	0x6D	109	m	0x7D	125	}
0x2E	46	.	0x3E	62	>	0x4E	78	N	0x5E	94	^	0x6E	110	n	0x7E	126	~
0x2F	47	/	0x3F	63	?	0x4F	79	O	0x5F	95	_	0x6F	111	o	0X00	0	NULL

Example: DNA Segments

- Each DNA base is one of A, C, T, or G
- Can represent a single DNA base with 2 bits
  - A = 00, C = 01, T = 10, G = 11
- How to represent whole genomes (say, of 100,000 DNA bases)?
  - Idea: Express it as an array of bases
  - In total, we can represent 100,000 DNA bases using 200,000 bits of data
- Example operator: Convert to the opposite DNA base
  - Using the above encoding scheme, can be implemented by taking the NOT of the top bit

Input	Output
A	T
C	G
T	A
G	C

Representing Data using Binary

- Anything that we put in memory gets stored using some number of bits (e.g. a 1-bit boolean, an 8-bit integer)
- Each possible bitstring can get assigned to at most one meaning
- In general, if we use n bits in our data object, we can represent up to 2^n different bitstrings, and therefore 2^n different meanings
- In most systems, it’s generally inefficient to work at the individual bit level. It’s often easier to add additional “0” bits to any data value up to the nearest multiple of 8 (so that each object is a whole number of bytes long), and sometimes even up to the word size (for this class, 4 bytes or 32 bits).
- Ex. While bools can theoretically be saved as one bit, in practice, we end up assigning an 8-bit char or even a 32-bit integer as a bool. Similarly, ASCII chars are normally given 8 bits (hence why 8-bit values are often called chars).

Example: File Permissions

- In a computer file system, keeps track of what a user is allowed to do with a file
- Generally composed of 9 True/False values
  - 3 levels of security with different permissions
  - Read/Write/Execute permissions
- Can theoretically be saved as 9 different Booleans
- However, if we use 4 bytes per Bool, that would take 36 bytes
- Solution: Treat a 32-bit integer as an array of 32 bits, and use 9 of them to store the relevant permissions.
  - Use bitwise operations to modify individual bits within this integer
- Saves the file permissions as a single 32-bit integer.

Agenda

- Binary and Hexadecimal
- Representing Data using Binary
    - ASCII
- **Binary Mathematics**
- Integer representations
    - Unsigned numbers
    - Sign-Magnitude
    - Bias
    - Two’s Complement
- ASCII
- Summary

Representing Nonnegative Integers

- Problem: There's infinitely many integers. Making a representation that could represent any integer would require infinitely many bits.
- 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 numbers up to a certain max value, using a fixed 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 unsigned integer, where we allow numbers from 0 to 2^{n-1} (ex. 8-bit unsigned int goes from 0 to 255)
    - To write an n-bit number, we always write out the full n bits, including leading zeros

Bitwise 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)

Examples: Bitwise Operations

- Let’s assume we’re working with 4-bit integers. Compute:
    - 6 & 5
    - 6 | 5
    - 6 ^ 5
    - ~6

6 = 0b	0	1	1	0	
5 = 0b	0	1	0	1	
6 & 5 = 0b	0	1	0	0	=4
6 | 5 = 0b	0	1	1	1	=7
6 ^ 5 = 0b	0	0	1	1	=3
~6 = 0b	1	0	0	1	=9

Bitwise Operations

- C gives access to the following bitwise operations: &, |, ~, ^, <<, >>
- 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

Shifts

- Ex. 0b1011 0100 << 3
- Ex. 0b1011 0100 >> 2

0	0	0	1	0	1	1	0	1	0	0	0	0	0b1011 0100
~~1~~	~~0~~	~~1~~	1	0	1	0	0	0	0	0	0	0	0b1010 0000
0	0	0	0	0	1	0	1	1	0	1	~~0~~	~~0~~	0b0010 1101

Decimal Analogue: Shifts

- For this slide, we’ll use $<<_{10}$ and $>>_{10}$ to refer to decimal shifts instead of our normal binary shifts.
- 1234 $<<_{10}$ 1 means:
    - Move all the digits one spot left, and add a zero to the end
    - 1234 -> 12340
    - Equivalent to multiplying by 10
- 1234 $>>_{10}$ 1 means:
    - Move all the digits one spot right, cutting off the last digit
    - 1234 -> 123
    - Equivalent to dividing by 10 (rounding down)
- In general:
    - Decimal left shifting by n multiplies a number by 10^n
    - Decimal right shifting by n divides a number by 10^n, rounding down.

Bitwise Operations

- C gives access to the following bitwise operations: &, |, ~, ^, <<, >>
- 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
  - Left shift: Equivalent to multiplying by a power of 2
  - Right shift (logical): Equivalent to dividing by a power of 2

Mathematical Operations

- 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.
  - The general approach for these is to follow a “grade-school algorithm”, but in binary instead of decimal.

Example: Addition

Decimal

```
    1
 31415
+27128
------
 58543
```

Binary

```
    11
 0b1001
+0b0011
-------
 0b1100
```

Example: Multiplication

Decimal

```
   31415
x    123
--------
   94245
  62830
 31415
--------
 3864045
```

Binary

```
 0b001001
x0b000011
---------
 0b  1001
 0b 1001
---------
 0b011011
```

Mathematical Operations

- C also allows for mathematical operators: +, -, *, /, %, <, >, ==, etc.
- 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)
        - 200 + 200
        - 100 * 4
        - 100 << 2
    - Going too low (also called overflow)
        - 100 - 200
    - Fractional result
        - 10 / 3

Mathematical Operations

- For fractional results: Treat division as floor division (similar to the // operator in Python)
  - 10 / 3 == 3
- For numbers too big or too small: “wrap around” so that the largest number representable + 1 becomes the smallest number
  - Ex. In an 8-bit unsigned integer, 255+1 == 0
- Equivalent definition: Take the result mod 2^n, or take only the lowest n bits of the number.
  - 200 + 200 is 400, which is (400-256 = 144) mod 256. Thus, 200 + 200 == 144
  - 100 * 4 == 144
  - 100 << 2 == 144. Note that 100 == 0b0110 0100, and 144 == 0b1001 0000, matching the left shift behavior described earlier
  - 100 - 200 == -100, which is (-100+256 = 156) mod 256. Thus, 100 - 200 == 156
  - Note that all of the above values are correct only for math with 8-bit unsigned integers

Agenda

- Binary and Hexadecimal
- Representing Data using Binary
  - ASCII
- Binary Mathematics
- **Integer representations**
  - Unsigned numbers
  - Sign-Magnitude
  - Bias
  - Two’s Complement

Goals for integer systems

- Any system/convention to store data ideally has the following properties:
- 1: The values that can be represented are relevant to the subject at hand
  - Different subject matters need different value sets, so we often have several different options with various trade-offs
- 2: The system is efficient, 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 Numbers

- The unsigned number system is the system that we've been describing up to now:
  - For n bits of data, we represent 0~(2^n-1), translating our value into its (mathematical) binary value.
  - Arithmetic operations use modular arithmetic to account for overflows
  - Called "unsigned" because we don't handle negative numbers, so we don't account for positive/negative sign

Analysis: Unsigned Numbers

- 1: The values that can be represented are relevant to the subject at hand
  - Yes; you often need to store positive/nonnegative integers in your code
- 2a: The system is memory efficient
  - Yes; every possible bitstring yields a different, valid integer
- 2b: The system has efficient operators
  - Somewhat: Bitwise operations, comparators, and addition/subtraction only require O(n) transistors to construct. Multiplication, division, and modulo arithmetic require O(n^2) transistors, so they're generally slower
    - For this reason, some languages like RISC-V don't support multiplication/division operators in their base instruction set.
- 3: The system is intuitive for a human to understand
  - Yes; most aspects of this system are directly analogous to decimal math

Example: Unsigned Numbers

- The unsigned number system is the system that we've been describing up to now:
  - For n bits of data, we represent 0-(2^n-1), translating our value into its (mathematical) binary value.
  - Arithmetic operations use modular arithmetic to account for overflows
  - Called "unsigned" because we don't handle negative numbers, so we don't account for positive/negative sign
- Often used in programs when you don't need negative numbers, or when dealing with bitwise operations
- 8-bit unsigned number: values from 0-255, arithmetic done mod 256
- 16-bit unsigned number: values from 0-65535, arithmetic done mod 65536

Problem: What if we want to handle negative numbers?

- Need to specify some encoding that includes negative values as well as positive values
- Immediately, there’s one issue: we won’t be able to represent as many positive numbers as before
    - With an unsigned number, every possible bitstring was assigned a nonnegative number. If we want to represent negative numbers, we’ll have to replace some positive numbers, so our max value will be smaller
- For most signed systems, we want to have about as many positive numbers as negative numbers

Sign-Magnitude Numbers

- Main idea: In decimal, we write negative numbers by first writing a “-” sign, then the absolute value of that number.
    - Ex. “-1234”
    - For completeness, let’s also imagine that we use “+” for positive numbers, as in “+1234”
- Binary equivalent:
    - Use the first bit of the data to store the sign (1 is negative, 0 is positive), then use the remaining bits to store the absolute value of the number

Sign	Magnitude						
-	23						
0b1	0	0	1	0	1	1	1

Analysis: Sign-Magnitude Numbers

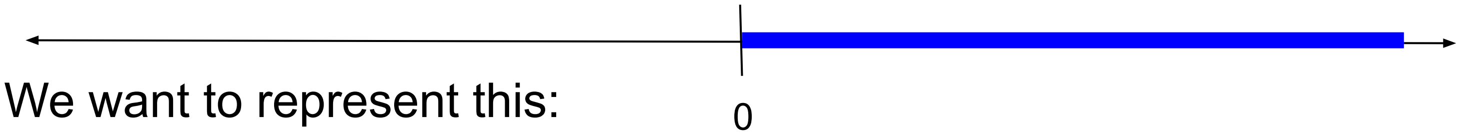
- 1: The values that can be represented are relevant to the subject at hand
  - Yes; negative numbers get stored, with about the same number of negative values as positive
- 2a: The system is memory efficient
  - Somewhat; the number 0 gets two different representations (0b0000 0000 and 0b1000 0000), but otherwise, every value is unique
- 2b: The system has efficient operators
  - No: comparators and addition/subtraction require additional handling for the sign bit
- 3: The system is intuitive for a human to understand
  - Yes; this is how we write numbers in decimal

Sign-Magnitude Numbers

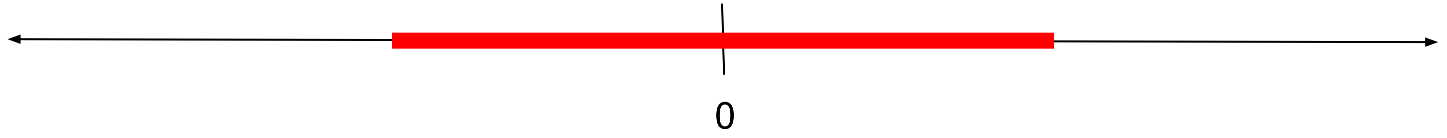
- Main idea: In decimal, we write negative numbers by first writing a “-” sign, then the absolute value of that number.
  - Ex. “-1234”
  - For completeness, let’s also imagine that we use “+” for positive numbers, as in “+1234”
- Binary equivalent:
  - Use the first bit of the data to store the sign (1 is negative, 0 is positive), then use the remaining bits to store the absolute value of the number
- Relatively uncommon as-is, but can be used as a component/starting point for more complicated systems, since as a base, it’s fairly versatile.

Bias Numbers

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

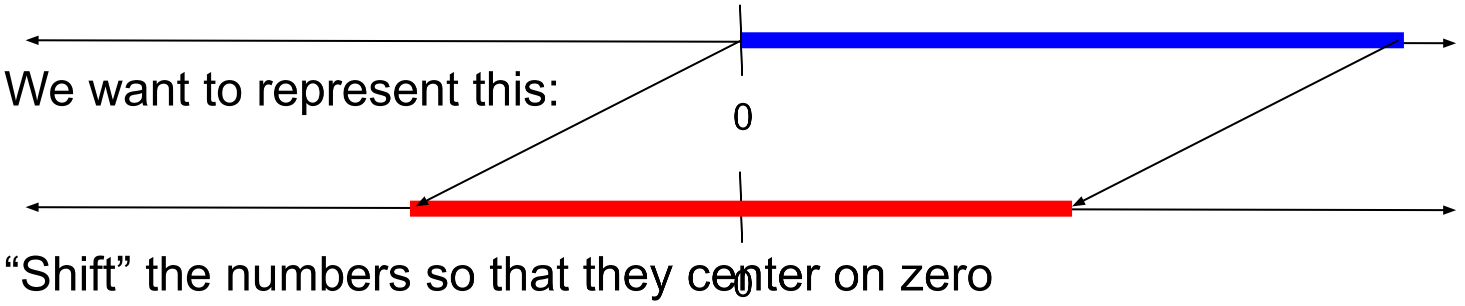


- We want to represent this:



Bias Numbers

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



- “Shift” the numbers so that they center on zero
- Formally:
  - Define a “bias”
  - To interpret stored binary: Read the data as an unsigned number, then add the bias
  - To store a data value: Subtract the bias, then store the resulting number as an unsigned number

Examples: Bias Numbers

- For the following example, let’s assume that we have a bias of -127 with an 8 bit number
- If we store the binary 0b0000 1001, we mean:
  - 0b0000 1001 as an unsigned integer is 9. We then add the bias (-127) to get our value -118.
- If we want to store the number -27:
  - We subtract the bias to get a number in the range of an unsigned integer:
  - -27 - (-127) = 100
  - We then save 100 = 0b0110 0100 as an unsigned number
- Smallest number: 0b0000 0000 = -127
- Largest number: 0b1111 1111 = (255 - 127) = 128

Analysis: Bias Numbers

- 1: The values that can be represented are relevant to the subject at hand
  - Yes: negative numbers get stored, with about the same number of negative values as positive
  - By choosing different values for the bias, we can represent different ranges, even those not centered at 0.
- 2a: The system is memory efficient
  - Yes: Every bitstring corresponds to a unique value
- 2b: The system has efficient operators
  - Somewhat: Comparators stay the exact same as the unsigned numbers, since we're just adding the same number to all bit patterns
  - Any arithmetic operations become significantly harder to do
- 3: The system is intuitive for a human to understand
  - Somewhat: It makes reasonable sense, but it's harder to keep track of an extra parameter in the system

Bias Numbers

- “Shift” the numbers so that they center on zero
- Formally:
  - Define a “bias”
  - To interpret stored binary: Read the data as an unsigned number, then add the bias
  - To store a data value: Subtract the bias, then store the resulting number as an unsigned number
- 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.

Two’s Complement Numbers

- 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)

Example: Two's Complement Numbers

- 0b0001 0001
  - The leading digit is 0, so we take the unsigned value of this number, which is 17. Thus, this bitstring means 17.
- 0b1110 1111
  - The leading digit is 1, so we take the unsigned value, and subtract 2^8 = 256. The unsigned value of this number is 239, so this bitstring means 239 - 255 = -17
- There's an easy shortcut for negating numbers in two's complement: In order to multiply a number by -1, flip the bits, and add 1.
  - Ex. The number 17 is represented by the binary 0b0001 0001. If we flip the bits of this, we get 0b1110 1110. If we then add 1, we get 0b1110 1111, which is the bitstring that stores -17 in two's complement.
  - This algorithm works both ways. For example, if I start with 0b1110 1111, I can flip the bits to get 0b0001 0000, then add 1 to get 0b0001 0001 = 17, which is the negation of the -17.
- It's often useful to convert values to binary through the above sequence, but the properties of this signed number system stem from the former definition.

Analysis: Two’s Complement Numbers

- 1: The values that can be represented are relevant to the subject at hand
  - Yes: negative numbers get stored, with about the same number of negative values as positive
- 2a: The system is memory efficient
  - Yes: Every bitstring corresponds to a unique value
- 2b: The system has efficient operators
  - Yes! This representation system is equivalent to unsigned numbers mod 2^n, and we’ve defined our operators by taking the result mod 2^n. As a result, addition, subtraction, and multiplication work exactly the same as with unsigned numbers, even using the same circuit. This makes two’s complement extremely efficient to implement if you’ve already implemented unsigned numbers.
- 3: The system is intuitive for a human to understand
  - Somewhat: “Flip the bits and add one” is a fairly useful mnemonic that allows humans to easily translate two’s complement numbers. The underlying math behind this system is fairly complex, though, so it’s somewhat unintuitive why this works.

Two’s Complement Numbers

- 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). Alternatively, flip the bits and add one to get the magnitude of the number, and interpret the number as a negative number.
- Because Two's Complement is equivalent mod 2^n to unsigned numbers, we can reuse the unsigned math circuits to implement 2's complement operations.

Math Aside (e.g. How Math 113 Applies to CS)

- Modular arithmetic forms what's known as a *ring*; in particular the quotient ring Z/nZ
- One property of quotient rings is that the choice of representative is irrelevant to the defined operations.
  - Unsigned numbers: We choose as representative the smallest nonnegative element
  - 2's Complement: We choose as representative the number closest to zero (with one extra negative number)
- Since we don't change the coset-bitstring mapping, we can use the same arithmetic circuits for operations *well-defined on the ring*
  - Addition, Subtraction and Multiplication, but NOT comparisons or division

Two’s Complement Numbers

- 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). Alternatively, flip the bits and add one to get the magnitude of the number, and interpret the number as a negative number.
- Because Two's Complement is equivalent mod 2^n to unsigned numbers, we can reuse the unsigned math circuits to implement 2's complement operations.
- Two’s complement is generally the implementation of choice for signed numbers.
  - From now on, we may sometimes say “signed number” without specifying the exact format. Unless otherwise stated, this refers to a two’s complement representation.