Lecture 02
Data Representations

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2026

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Northwestern

Announcements

- Homework 1 will be posted sometime later today
  - Practice problems on binary-to-hex-to-decimal conversion, integer encodings, memory, and floating point
  - Some questions will rely on future lectures. There will be a title for each section clarifying which lecture(s) it relies on
    - Goal: work on parts of the homework as class goes on
  - Due Tuesday, January 20th
    - You'll submit to Gradescope, which you should all have access to
    - Reminder: no late submissions for homework

Religious Accommodations

- All kinds of holidays overlap with the quarter

- If those are going to affect your participation in CS213, reach out ASAP and let me know (by the end of the second week of the quarter)
  - We can give accommodations for that kind of thing

Today’s Goals

- Review of binary and hexadecimal
- Discuss data representation in memory
- Explore data representations
  - Integers, signed and unsigned
  - Different bit widths
  - Translating between encoding schemes
  - Other encodings besides integers

Outline

- **Binary and Hex**
- Memory
- Encoding
- Integer Encodings
  - Signed Integers
  - Converting Sign
  - Converting Length
- Other encodings

Review: binary

- Base 2 counting system used by computers
  - Possible values are limited to 0 or 1
  - Positional numbering system (like decimal is)
    - Further to the left is bigger, further to the right is smaller

- Example: 0110_2
  - Converting to decimal (base 10):
    - Multiply bits by $2^{position}$
  - $0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$
  - $= 0*8 + 1*4 + 1*2 + 0*1$
  - $= 4 + 2 = 6_{10}$

- Example: 12_{10}
  - Converting to binary (base 2):
    - Break down into powers of two
  - 12 = 8 + 4
  - $= 1*8 + 1*4 + 0*2 + 0*1$
  - $= 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0$
  - = 0b1100

Practice: Base 2

Remember to start counting bit positions from 0!!

- Translate 0b10111 to decimal (0b means binary)
  - 1*16 + 0*8 + 1*4 + 1*2 + 1*1
  - 16 + 4 + 2 + 1
  - 23

Remember to start counting powers of two from 1!!

- Translate decimal 40 to binary
  - 32 + 8
  - 1*32 + 0*16 + 1*8 + 0*4 + 0*2 + 0*1
  - 0b101000

Base 16: Hexadecimal

- Writing long sequences of 0s and 1s is tedious and error-prone
  - And takes up a lot of space on a page!
- So we'll often use base 16 (also called *hexadecimal*)

- Base 2 = 2 symbols (0, 1)
  Base 10 = 10 symbols (0-9)
  Base 16, need 16 symbols
  - Use letters A-F once we run out of decimal digits

- Example: 0x3A (0x means hexadecimal)

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

Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:

  - Convert binary to decimal:

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

Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:
    - 0x4 -> 0b0100   0x2 -> 0b0010
    - 0x42 -> 0b 0100 0010
  - Convert binary to decimal:

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

Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:
    - 0x4 -> 0b0100   0x2 -> 0b0010
    - 0x42 -> 0b 0100 0010
  - Convert binary to decimal:
    - $1*2^6 + 1*2^1 = 64 + 2 = 66$

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

Practice problem

- **Convert 0x42 to decimal**

- Alternate method:
  - 0x42
  - $= 4 \times 16^1 + 2 \times 16^0$
  - $= 64 + 2$
  - $= 66$

- But you're honestly better off converting hex to binary first
  - It's good practice!

Decimal-to-Hex example

- Convert decimal 165 to hex (first convert to binary)
- Decimal to binary
  - 165
  - 128 + 32 + 4 + 1
  - 1*128 + 0*64 + 1*32 + 0*16 + 0*8 + 1*4 + 0*2 + 1*1
  - 0b10100101
- Binary to hex
  - 0b1010 0101
  - 0xA5

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

Specific bit widths

- Since computers are hardware, cannot have arbitrary bit lengths
  - Must be *some* specific number of bits
  - We'll say "this is a 16-bit number" or this is a "9-bit number"

- What if the number of bits isn't divisible by four, can it still be hex?
  - Yes! All non-existent bits MUST be zero

- Example: translate 0x35 to a 6-bit binary number
  - 0b0011 0101 -> 0b11 0101

  - If it had been 0x75 (0b111 0101) writing as a 6-bit binary number would be impossible (converting could happen, but we'd have overflow)

More bit-width examples

- Convert 0b11 to hexadecimal
  - 0x3

- How many bits is 0x3?
  - MUST be at least 2 bits
  - Could be more though: 4-bit, or 64-bit, or as many zeros as you want
    - 0b0011
    - 0b0000000000000000000000000000000000000000000000000000000000000011

Outline

- Binary and Hex
- **Memory**
- Encoding
- Integer Encodings
  - Signed Integers
  - Converting Sign
  - Converting Length
- Other encodings

Bytes

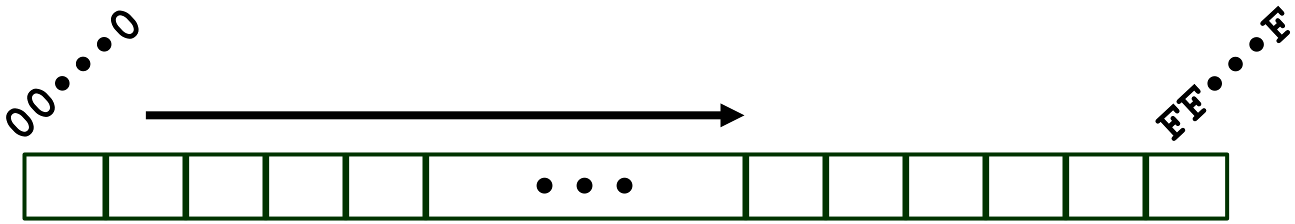
- A single bit doesn't hold much information
  - Only two possible values: 0 and 1
  - So we'll typically work with larger groups of bits
- For convenience, we'll refer to groups of 8 bits as ***bytes***
  - And usually work with multiples of 8 bits at a time
  - Conveniently, 8 bits = 2 hexits
- Some examples
  - 1 byte: 0b01100111 = 0x67
  - 2 bytes: $11000100\ 00101111_2$ = 0xC42F

"0b" prefix = it's in binary
"0x" prefix = it's in hex

Byte-oriented memory organization

- We’ve seen how sequences of bits can express numbers
  - And how we usually work with groups of 8 bits (***bytes***) for convenience
  - Variables are almost always some multiple number of bytes (1 byte, 2 bytes, 8 bytes, etc.)

- In a computer system, bytes can be stored in memory
  - Conceptually, memory is a very large array of bytes
  - Each byte has its own address (≈ pointer)



- Compiler + run-time system control allocation
  - Where different program objects should be stored
  - Multiple mechanisms, each with its own region: static, stack, and heap

Most/least significant bits/bytes

- When working with sequences of bits (or sequences of bytes), need to be able to talk about specific bits (bytes)
  - Most Significant bit (MSb) and Most Significant Byte (MSB)
    - Have the largest possible contribution to numeric value
    - Leftmost when writing out the binary sequence (always)
  - Least Significant bit (LSb) and Least Significant Byte (LSB)
    - Have the smallest possible contribution to numeric value
    - Rightmost when writing out the binary sequence (always)

$\underbrace{10110110}_{\text{Most significant byte}}$ 01101010 10101010 $\underbrace{01111100}_{\text{Least significant byte}}$

Most significant bit

Least significant bit

Addressing and byte ordering

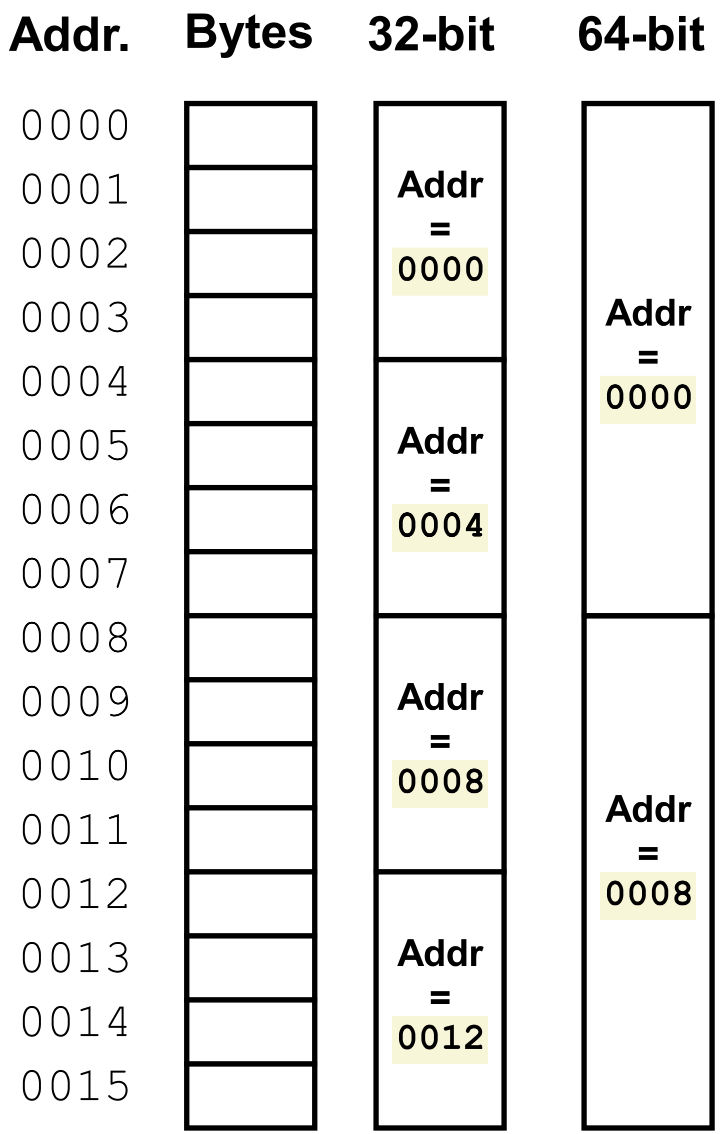
- For data that spans multiple bytes, need to agree on two things
  - **1. What should be the address of the object?**(each byte has its own!)
    - And by extension, given an address, how do we find the relevant bytes (same question!)
  - **2. How should we order the bytes in memory?**
    - Do we put the *most* or *least* significant byte at the first address?

There isn’t always one correct answer

- Different systems can pick different answers! (mostly for 2nd question)
  - Very nice illustration of two overarching principles in systems:
    You need to know the specifics of the system you’re using!
    - Many questions don’t really have right or wrong answers!
    - Instead, they have tradeoffs. What the “right” answer is depends on context!
  - Different answers across systems is perfectly fine
    - But all the parts of a given system must agree with each other!

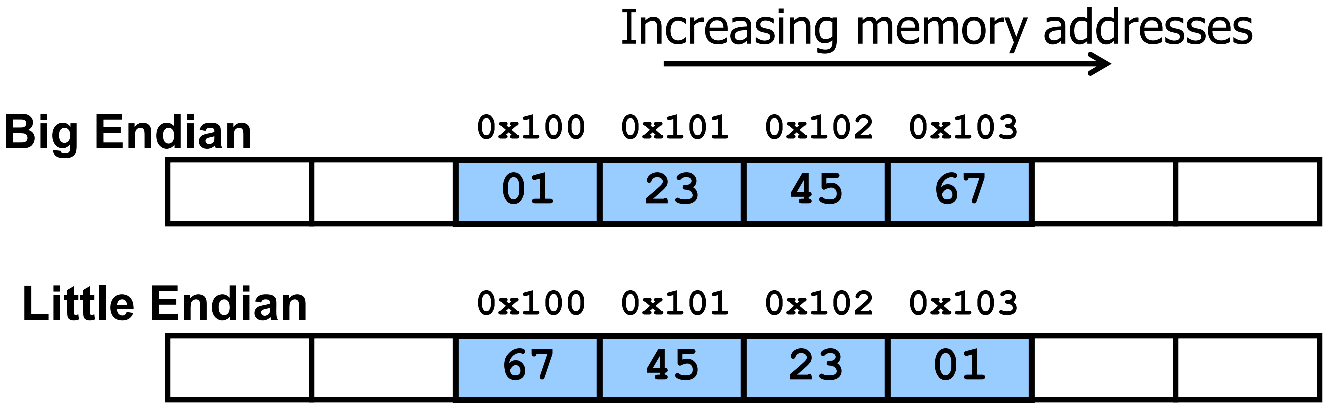
1. Addressing data in memory

- All addresses refer to one or more bytes
  - Never bits
- For multi-byte objects, the lowest address refers to the entire object
  - Addresses of successive objects differ by 4 (32-bit) or 8 (64-bit)
- Systems pretty much universally use the address of the first byte as the address for the whole object
  - I'm not aware of any system that does otherwise, but there could be some weirdo systems out there (or historically)



2. Byte ordering

- How to order bytes within a multi-byte object in memory
  - Only relevant when working with data larger than a byte!
- Conventions
  - Big Endian: Computer Networks, various historical computer systems
    - Most significant byte has lowest address (comes first)
  - Little Endian: Intel (x86, x86-64), ARM, other modern computer systems
    - Least significant byte has lowest address (comes first)
- Example
  - 4-byte piece of data: **`0x01234567`**
  - Address of that data is **`0x100`**



Practice: reading memory

- Assume memory is **Little Endian**
  - So the Least Significant Byte comes first

1. What is the four-byte value at 0x2010?

2. What is the two-byte value at 0x2014?

3. What is the one-byte value at 0x2016?

Address	**Value**
0x2010	0x37
0x2011	0x1A
0x2012	0xBE
0x2013	0x98
0x2014	0x0C
0x2015	0x80
0x2016	0x42

Practice: reading memory

- Assume memory is **Little Endian**
  - So the Least Significant Byte comes first

1. What is the four-byte value at 0x2010?
   0x98BE1A37
2. What is the two-byte value at 0x2014?
   0x800C
3. What is the one-byte value at 0x2016?

Address	**Value**
0x2010	0x37
0x2011	0x1A
0x2012	0xBE
0x2013	0x98
0x2014	0x0C
0x2015	0x80
0x2016	0x42

Practice: reading memory

- Assume memory is **Little Endian**
  - So the Least Significant Byte comes first

1. What is the four-byte value at 0x2010?
   0x98BE1A37
2. What is the two-byte value at 0x2014?
   0x800C
3. What is the one-byte value at 0x2016?
   0x42

Address	**Value**
0x2010	0x37
0x2011	0x1A
0x2012	0xBE
0x2013	0x98
0x2014	0x0C
0x2015	0x80
0x2016	0x42

Practice: reading memory

- Change: assume memory is **Big Endian**
  - So the Most Significant Byte comes first

1. What is the four-byte value at 0x2010?
   0x371ABE98

2. What is the two-byte value at 0x2014?
   0x0C80

3. What is the one-byte value at 0x2016?
   0x42

Note: endianness doesn't affect one-byte values!

Address	**Value**
0x2010	0x37
0x2011	0x1A
0x2012	0xBE
0x2013	0x98
0x2014	0x0C
0x2015	0x80
0x2016	0x42

Tables of memory

Address	0	1	2	3	4	5	6	7
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

- Method of displaying large chunks of memory
  - 8 bytes per row (in this example)
  - Data values in hexadecimal

- Memory addresses labeled on the left and byte offset on the top
  - Also in hexadecimal

Tables of memory

Address	0	1	2	3	4	5	6	7
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

- To find an address: deconstruct into:
  - base address (left) + offset (top)
- Remember: addresses are in hexadecimal!
  - 0x1048+2 = 0x104A

Practice: Tables of memory

Address	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

- What value is at address 0x1050?
  - 0x54 (0x1050 + 0)
- What value is at address 0x105F?
  - 0xCE (0x1058 + 7)

Outline

- Binary and Hex
- Memory
- **Encoding**
- Integer Encodings
  - Signed Integers
  - Converting Sign
  - Converting Length
- Other encodings

Big Idea: What do bits and bytes ***mean*** in a system?

- The answer is: it depends!

- Depending on the context, the bits **`11000011`** could mean
  - The number 195
  - The number -61
  - The number -19/16
  - The character '├'
  - The **`ret`** x86 instruction

- You have to know the context to make sense of any bits you have!
  - Looking at the same bits in different contexts can lead to interesting results
  - Information = bits + context!

- An ***encoding*** is a set of rules that gives meaning to bits

An example encoding: ASCII characters

- ASCII = American Standard Code for Information Interchange
  - Standard dating from the 60s
- Maps 8-bit* bit patterns to characters
  - (* the standard is actually 7-bit, leaving the 8th bit unused)
  - Each bit pattern maps to a single character
- Examples
  - $0100\ 0001_2$ = 0x41 = 65_{10} = **'A'**
  - $0100\ 0010_2$ = 0x42 = 66_{10} = **'B'**
  - $0011\ 0000_2$ = 0x30 = 48_{10} = **'0'**
  - $0011\ 0001_2$ = 0x31 = 49_{10} = **'1'**
- Reference: https://www.asciitable.com/

Full ASCII table

Values listed in:

Decimal

Hexadecimal

Octal

HTML

Character

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Encodings are just determined by people

- There's no inherent **truth** in the design of an encoding
  - Although some encodings are nice or annoying for various reasons
  - Example: it's nice in ASCII that letters are in alphabetical order

- You could come up with an entirely new way of encoding characters
  - The hard part would be getting everyone else to agree to use it

Open Question + Break

- **What things might we want to encode?**

Open Question + Break

- **What things might we want to encode?**

  - Numbers
    - Signed and unsigned integers
    - Real numbers
    - Mathematical symbols: ∞ π

  - Language
    - Characters in various different languages ΩИسل서北
    - Emoji 😱😡😁😭🤷‍♀️🦃🎶🎂✨🐱‍👤🍢

  - Colors, Playing Cards, User Actions, anything!

Outline

- Binary and Hex
- Memory
- Encoding
- **Integer Encodings**
  - Signed Integers
  - Converting Sign
  - Converting Length
- Other encodings

Integer types in C

- C types describe both encoding and size

- Encoding
  - `signed` or `unsigned`
  - Defaults to signed if no label included

- Sizes
  - 1 byte   (8-bit): `char`
  - 2 bytes (16-bit): `short`
  - 4 bytes (32-bit): `int`
  - 8 bytes (64-bit): `long`

Examples:

- `unsigned int`
  - 32-bit unsigned integer

- `signed short`
  - 16-bit signed integer

- `char`
  - 8-bit signed integer

Sizes of C types are technically system dependent

- C only defines types as a "minimum" size
  - So they could be different on different computers
  - This is not a good choice
  - You should assume the x86-64 rules
    - Memorize these! Matches prior slide

- How I program
  - Use fixed width integer types <stdint.h>
  - int8_t, int16_t, int32_t
  - uint8_t, uint16_t, uint32_t

C Data Type	x86-64	C Standard* (C99)
char	1	≥1
short	2	≥2
int	4	≥2
long	8	≥4
long long	8	≥8
float	4	
double	8	
pointer	8	Widths for data, code pointers may differ!

↑ CS213 uses this

Expressing C types in bits

- Two families of encodings to express integers using bits
  - ***Unsigned*** encoding for unsigned integers
  - ***Two's complement*** encoding for signed integers

- Each encoding will use a fixed size (# of bits)
  - For a given machine
  - Size + encoding family determine which C type we're representing
    - Reverse is also true: a C type specifies both size and encoding
  - Fixed size is because computers are finite!

Unsigned integer encoding

- Just write out the number in binary
  - Works for 0 and all positive integers

- Example: encode 104_{10} as an **unsigned** 8-bit integer
  - $104_{10} = 0\times2^7 + 1\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0$

$\Rightarrow$ **`0b01101000`**

$\Rightarrow$ **`0x68`**

Formal Definition

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

(Binary To Unsigned)

Bounds of unsigned integers

- For a fixed width ***w***, a limited range of integers can be expressed
  - Smallest value (we will call ***UMin***):
    - all 0s bit pattern: 000…0, value of 0
  - Largest value (we will call ***UMax***):
    - all 1s bit pattern: 111...1, value of $2^w - 1$
    - $2^w - 1 = 1 \times 2^{w-1} + 1 \times 2^{w-2} + ... + 1 \times 2^1 + 1 \times 2^0 = 11111...$
- Maximum 8-bit number = 2^8-1 = 256-1 = 255

Outline

- Binary and Hex
- Memory
- Encoding
- **Integer Encodings**
  - **Signed Integers**
  - Converting Sign
  - Converting Length
- Other encodings

Encoding signed integers

- What’s different about representing a signed number?
  - It can be negative!

- So, we’re going to have to somehow represent values that are negative and positive

- There are actually many different encodings capable of doing this
  - This is when that “nice encoding” versus “annoying encoding” matters
  - We’ll focus on the most-commonly used method: two’s complement

Two's complement encoding

- Bad news: need to make the encoding more complicated to handle negatives

- Plan:
  - Start with unsigned encoding, but then make ONLY the largest power negative
  - Example: for 8 bits, most significant bit is worth -2^7 not $+2^7$ (other bits are still positive)

- To encode a negative integer
  - First, set the most significant bit to 1 to start with a big negative number
  - Then, add positive powers of 2 (the other bits) to "get back" to number we want

- Example: encode -6 as a 4-bit two's complement integer
  - -6_{10} =

$1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^1$ ⇒ `0b1010` ⇒ **`0xa`**

Two's complement examples

- Encode -100 as an 8-bit two's complement number

  - $-100_{10} =$ $1 \times -2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

    $-128 \quad + 0 \quad + 0 \quad + 16 \quad + 8 \quad +4 \quad +0 \quad +0$

    Problem becomes:
    encode +28 as a 7-bit unsigned number

  - -100_{10} = 0b10011100 = 0x9C

  - **Shortcut:** determine positive version of number, flip all the bits, and add one
    - 100_{10} = 0b01100100
    - Flipped = 0b10011011
    - Plus 1 = 0b10011100 = 0x9C We'll talk about binary addition next lecture

Interpreting binary signed values

Formal Definition

- Converting binary to signed:

$$B2T(X) \quad = \quad -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign bit

- Note: most significant bit tells us sign!! 1-> negative
  - Checking if a number is negative is just checking that top bit
- Only one encoding exists for zero
  - 0b00000000 = 0 0b10000000 = -128
- -1: 0b111…1 = -1 (regardless of number of bits!)

Bounds of two's complement integers

- For a fixed width ***w***, a limited range of integers can be expressed
  - Smallest value, most negative (we will call ***TMin***):
    - 1 followed by all 0s bit pattern: 100…0 = -2^{w-1}
  - Largest value, most positive (we will call ***TMax***):
    - 0 followed by all 1s bit pattern: 01...1, value of $2^{w-1} - 1$
- Beware the asymmetry! Bigger negative number than positive

Ranges for different bit amounts

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
Umin	0	0	0	0
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- Observations
  - $|TMin| \quad = \; TMax + 1$
    - Asymmetric range
  - $UMax \;\; = \;\; 2 * \; TMax + 1$
- C Programming
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values are platform specific

Unsigned & Signed Numeric Values

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Equivalence**
  - Same encodings for non-negative values
- **Uniqueness**
  - Every bit pattern represents a single, unique integer value
  - Each representable integer has a single, unique bit encoding
- $\Rightarrow$ **Can Invert Mappings**
  - Can go from bits to number and back, and vice versa

Practice + Break

- What range of integers can be represented with 5-bit two's complement?

  - A -31 to +31
  - B -15 to +15
  - C 0 to +31
  - D -16 to +15
  - E -32 to +31

Practice + Break

- What range of integers can be represented with 5-bit two's complement?

  - A -31 to +31 No asymmetry and 6-bits
  - B -15 to +15 No asymmetry
  - C 0 to +31 Unsigned
  - D -16 to +15 Correct
  - E -32 to +31 6-bits

Outline

- Binary and Hex
- Memory
- Encoding
- **Integer Encodings**
  - Signed Integers
  - **Converting Sign**
  - Converting Length
- Other encodings

Casting signed to unsigned

- C allows conversions from signed to unsigned (and vice versa)

```
short int          x  =  15213;
unsigned short int ux = (unsigned short) x;
short int          y  = -15213;
unsigned short int uy = y; /* implicit cast! */
```

- Resulting value
  - Not based on current decimal value: instead keep the bits and ***reinterpret*** them!
  - Non-negative values unchanged
    - *ux* = 15213
  - Negative values change into (large) positive values (and vice versa)
    - *uy* = 50323
- Warning: Casts can be implicit in assignments or function calls!

Mapping Signed $\leftrightarrow$ Unsigned (4 bits)

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3	=	3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6	+ 16 (i.e., 2^4)	10
1011	-5		11
1100	-4		12
1101	-3	- 16 (i.e., 2^4)	13
1110	-2		14
1111	-1		15

Large negative value becomes large positive!

Example

- Convert signed 8-bit number -120 into an unsigned 8-bit number

1. Convert -120 into binary

2. Convert binary back into unsigned decimal

Example

- Convert signed 8-bit number -120 into an unsigned number

1. Convert -120 into binary
   -120 = -128 + 8 =

2. Convert binary back into unsigned decimal

Example

- Convert signed 8-bit number -120 into an unsigned number

1. Convert -120 into binary
   -120 = -128 + 8 =
   1x(-128) + 0x64 + 0x32 + 0x16 + 1x8 + 0x4 + 0x2 + 0x1
   0b 1000 1000

2. Convert binary back into unsigned decimal

Example

- Convert signed 8-bit number -120 into an unsigned number

1. Convert -120 into binary
   -120 = -128 + 8 =
   1x(-128) + 0x64 + 0x32 + 0x16 + 1x8 + 0x4 + 0x2 + 0x1
   0b 1000 1000

2. Convert binary back into unsigned decimal
   1x128 + 0x64 + 0x32 + 0x16 + 1x8 + 0x4 + 0x2 + 0x1
   128 + 8 = 136

Outline

- Binary and Hex
- Memory
- Encoding
- **Integer Encodings**
    - Signed Integers
    - Converting Sign
    - **Converting Length**
- Other encodings

Why convert bit length?

- Hardware has physical limits
  - We cannot hold an infinite number of bits

- So sometimes we have to change the sizes of values
  - Reducing to use fewer bits: Truncation
  - Expanding to use more bits: Extension
  - Computer hardware automatically follows these rules
    - So programming languages automatically do them as well

- These aren’t mathematical operations, they’re hardware limitations
  - So they might not work as you expected

Truncation (reducing number of bits)

- Truncation rule:
  - Just delete most-significant bits until the value is the correct size
  - Same rule no matter the encoding

- If the value fits in both bit sizes, it remains the same
- If not, it will be totally different
  - Result becomes identical to modular arithmetic
  - But we can always just do the truncation and check result

- 6-bit to 2-bit truncation
  - 0b000010 -> 0b~~0000~~10 = 0b10 (decimal 2 becomes decimal 2)
  - 0b110010 -> 0b~~1100~~10 = 0b10 (decimal 50 becomes decimal 2)

    Identical to 50 modulo 4 = 2

Sidebar: why do truncation?

- It can totally mess up the value!! So why do it?

- Because you sometimes have to
  - You might have a 32-bit result, but only be able to store an 8-bit value

  - Good code checks whether the result is in-bounds, and takes some action if it isn't
    - Maybe just set to maximum
    - Maybe throw an error

Extension (increasing number of bits)

- Extension rule:
  - Just add most-significant bits until the value is the correct size

- But should we add zeros or ones??
  - Unsigned extension: add zeros
  - Signed extension: add copies of most-significant bit (0 or 1)
  - Both results preserve the original value!

- 4-bit to 7-bit extension
  - Unsigned 0b0110 -> 0b0000110 (decimal 6 stays decimal 6)
  - Unsigned 0b1000 -> 0b0001000 (decimal 8 stays decimal 8)

  - Signed 0b0110 -> 0b0000110 (decimal 6 stays decimal 6)
  - Signed 0b1000 -> 0b1111000 (decimal -8 stays decimal -8 = -64+32+16+8)

Break + Practice

- Convert 16-bit 0x3427 to an 8-bit signed integer (hex)

- Convert 8-bit 0xF0 to a 16-bit signed integer (hex)

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

Break + Practice

- Convert 16-bit 0x3427 to an 8-bit signed integer (hex)
  - Process: truncate extra bits
  - Answer is **0x27**
- Convert 8-bit 0xF0 to a 16-bit signed integer (hex)

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

Break + Practice

- Convert 16-bit 0x3427 to an 8-bit signed integer (hex)
  - Process: truncate extra bits
  - Answer is **0x27**
- Convert 8-bit 0xF0 to a 16-bit signed integer (hex)
  - Process: sign extend. Is the most-significant bit one? Yes!
  - Answer is **0xFFF0**

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

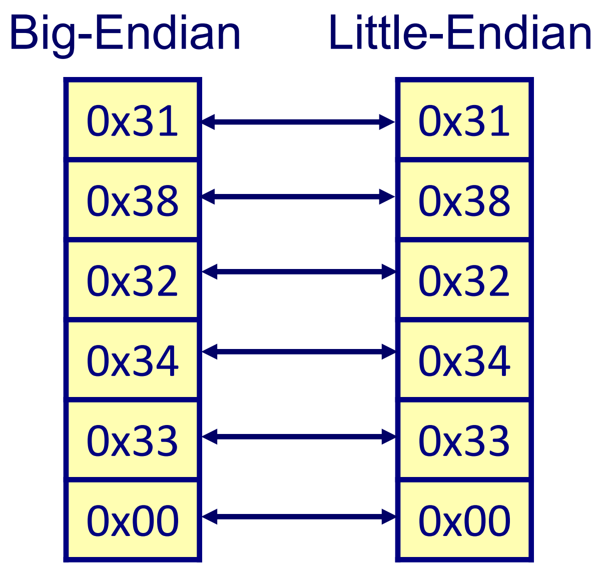
Outline

- Binary and Hex
- Memory
- Encoding
- Integer Encodings
    - Signed Integers
    - Converting Sign
    - Converting Length
- **Other encodings**

Encoding strings (The C way)

- Represented by array of characters
  - Each character encoded in ASCII format
  - NULL character (code 0) to mark the end
- Compatibility
  - Byte ordering not an issue (data all single-byte!)
  - ASCII text files generally platform independent
    - Except for different conventions of line termination character(s)!

```
char S[6] = "18243";
```



Encoding color

- RGB colors
  - 3-byte values
  - First byte is Red, then Green, then Blue

- Usually specified in hexadecimal
  - #FF0000 -> maximum red, zero green or blue
  - #4E2A84 -> 1/4 red, 1/8 blue, 1/2 green (Northwestern Purple)

- 2^{24} possible colors = 16777216 colors

Interpreting file contents

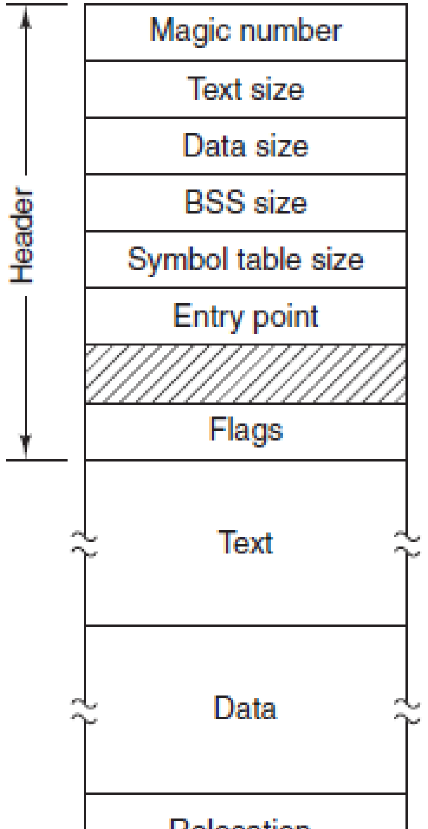
- Collections of data
    - Usually in permanent storage on your computer

- Regular files
    - Arbitrary data
    - Think of as a big array of bytes

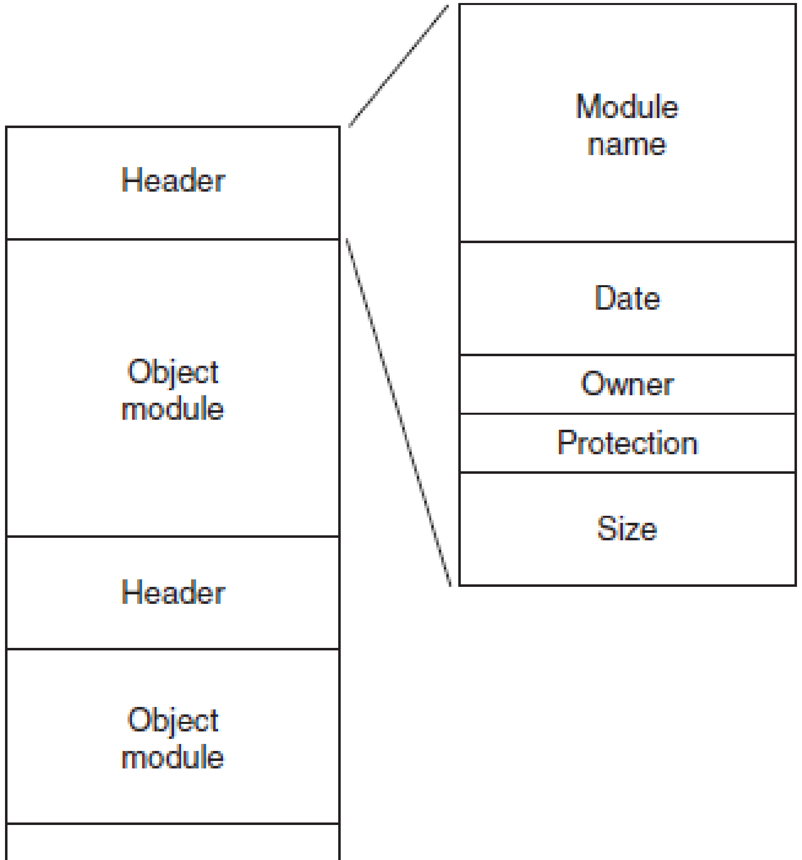
- Non-regular files would be directories, symbolic links, or other less used things

What about different types of regular files?

- Text files versus Executables versus Tar files
  - All just differing patterns of bytes!
  - It really is just all data. The meaning is in how you interpret it.



Executable File



Archive (tar)

Identifying regular files

- **`file`** in Linux command line can help determine the type of a file
  - https://github.com/file/file

```
arguments  arguments.c
[brghena@ubuntu code] $ file arguments.c
arguments.c: C source, ASCII text
[brghena@ubuntu code] $ file arguments
arguments: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64
/ld-linux-x86-64.so.2, BuildID[sha1]=8731c4961d371f4989cd1b056f796ad54b711e6f, for GNU/Linux 3.2.0, not s
tripped
[brghena@ubuntu code] $ file ./
./: directory
[brghena@ubuntu code] $ file ~/scratch/GlobalProtect_UI_deb-5.1.0.0-101.deb
/home/brghena/scratch/GlobalProtect_UI_deb-5.1.0.0-101.deb: Debian binary package (format 2.0), with cont
rol.tar.gz, data compression xz
```

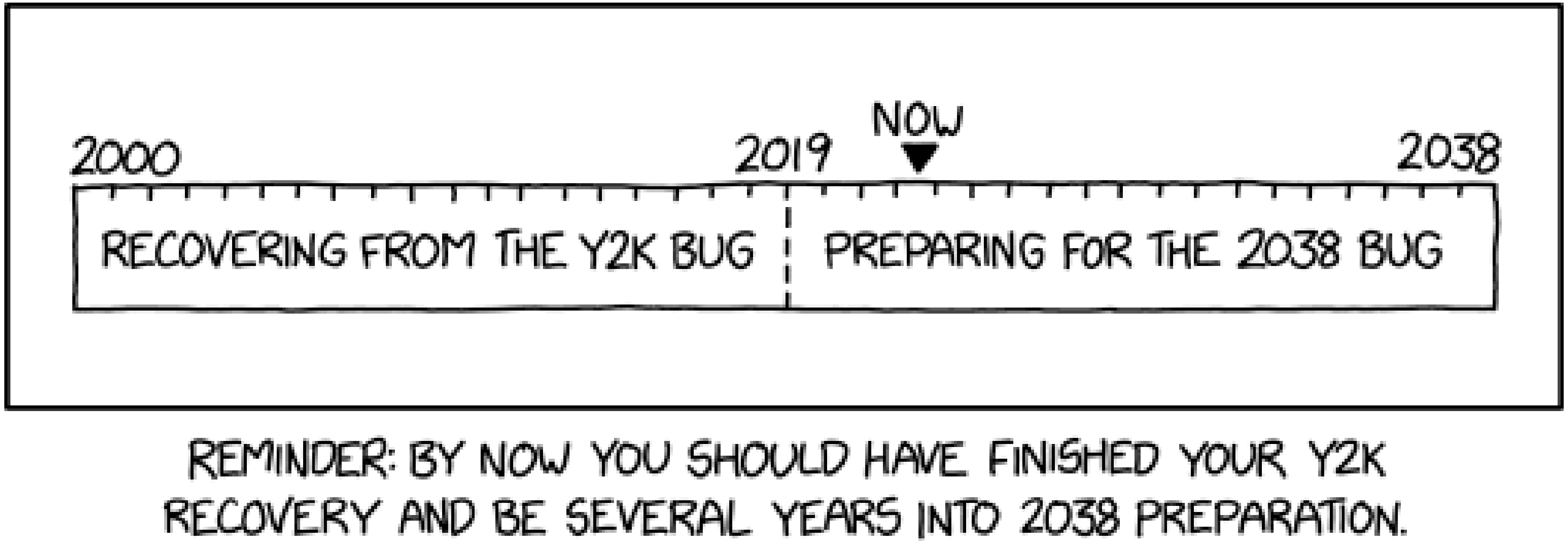
Encoding time

- Unix time:
  - 32-bit signed integer counting seconds elapsed since initial time
  - Initial time was January 1st at midnight UTC, 1970

- Current Unix time (as of last editing this slide): 1767890342
  - Negative numbers would mean times before 1970

- Problem: when does Unix time hit the maximum value?
  - 2147483647 seconds from January 1st 1970
  - Result: January 19th, 2038
  - This is the "Year 2038 Problem"

Bonus xkcd comic



https://xkcd.com/2697/

Outline

- Binary and Hex
- Memory
- Encoding
- Integer Encodings
  - Signed Integers
  - Converting Sign
  - Converting Length
- Other encodings