Bits & Bytes

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka, Daniel Gearhardt, Ash Fujiyama, Emily Shen

TAs:

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

Poll Everywhere

pollev.com/cis2400

- How are you? Any Questions?

Lecture Outline

- How do we count?
  - Bases
- Binary
  - Conversions
  - Hexadecimal
- Unsigned Numbers
- Overflow
- Signed Numbers
  - Two’s Complement
  - Two’s Complement Overflow

Lecture Outline

- How do we count?
  - Bases
- Binary
  - Conversions
  - Hexadecimal
- Unsigned Numbers
- Overflow
- Signed Numbers
  - Two's Complement
  - Two's Complement Overflow

Base 10 (Decimal Numbers)

- ***Humans*** typically process numbers in base 10

5 9 3 4

Digits 0-9 (*0* to *base-1*)

Base 10 (Decimal Numbers)

- ***Humans*** typically process numbers in base 10

5 9 3 4

10^3 10^2 10^1 10^0

Base 10 (Decimal Numbers)

- ***Humans*** typically process numbers in base 10

5 9 3 4

10^X: 3 2 1 0

Base 2

Each of these is a bit!

1 0 1 1

2^X: 3 2 1 0

Digits 0-1 (*0* to *base-1*)

Base 2

1 0 1 1

2^3 2^2 2^1 2^0

Base 2

Most significant bit (MSB) Least significant bit (LSB)

1 0 1 1

eights fours twos ones

$1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11_{\text{(base 10)}}$

$1*8 + 0*4 + 1*2 + 1*1 = 11_{\text{(base 10)}}$

Note: this is only an example with 4 bits!!!

Base 2

1	1	1	1	1	1	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0

- The I'th bit represents 2^i
- We can also use the prefix '0b' to denote base 2. (e.g. ***0b1101***)

Practice: Base 2 to Base 10

- What is 0b10110 in base 10?

Poll Everywhere

pollev.com/cis2400

- What is 0b10110 in base 10?

A. 6

B. 22

C. 16

D. 38

E. Tbh, I'm not sure.

Poll Everywhere

pollev.com/cis2400

- What is 0b10110 in base 10?

A. 6

B. 22

C. 16

D. 38

E. Tbh, I'm not sure.

0b10110

$(1 * 2^4) + (0 * 2^3) + (1 * 2^2) + (1 * 2^1) + (0 * 2^0)$

16 + 0 + 4 + 2 + 0

16 + 4 + 2

22

Lecture Outline

- How do we count?
  - Bases
- Binary
  - Conversions
  - Hexadecimal
- Unsigned Numbers
- Overflow
- Signed Numbers
  - Two's Complement
  - Two's Complement Overflow

From Decimals to Binary

- Algorithm 1:
  - Find the largest power of 2 <= the num
  - Subtract this largest power of 2 from the num
  - Place a '1' in the bit position corresponding to this power of 2
  - Repeat until number is 0

- Example: 104
  - 104 - 64 = **40** — 64 is 2^6, so bit 6 is a '1'
  - **40** - 32 = **8** — 32 is 2^5, so bit 5 is a '1'
  - **8** – 8 = 0 — 8 is 2^3, so bit 3 is a '1'
  - 104 = 0b1101000

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

From Decimals to Binary: Division

- Algorithm 2:
  - Divide by two – remainder will be the next smallest bit
  - Keep dividing until answer is 0
- Example: 104
  - 104 / 2 = 52 r 0 bit 0 is 0
  - 52 / 2 = 26 r 0 bit 1 is 0
  - 26 / 2 = 13 r 0 bit 2 is 0
  - 13 / 2 = 6 r 1 bit 3 is 1
  - 6 / 2 = 3 r 0 bit 4 is 0
  - 3 / 2 = 1 r 1 bit 5 is 1
  - 1 / 2 = 0 r 1 bit 6 is 1
  - 104 = 0b1101000

Note: think about what it means to divide a binary number by two.

Poll Everywhere

pollev.com/cis2400

- What is 99 in binary?

A. 0b111111

B. 0b110111

C. 0b1011111

D. 0b1100011

E. Tbh, I'm not sure

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Poll Everywhere

pollev.com/cis2400

- What is 99 in binary?

A. 0b111111

B. 0b110111

C. 0b1011111

D. 0b1100011

E. Tbh, I'm not sure

99 – 64 = 35, bit 6 is 1

35 – 32 = 3, bit 5 is 1

3 – 2 = 1, bit 1 is 1

1 – 1 = 0, bit 0 is 1

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Byte Values

- What is the minimum and maximum base 10 value a single byte (8 bits) can store?

minimum = 0 **maximum = 255**

11111111

2^x: 7 6 5 4 3 2 1 0

- **Strategy 1:** $1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 255$
- **Strategy 2:** $2^8 – 1 = 255$

Multiplying and Dividing by Bases

1450 x 10 = 1450**<u>0</u>**

0b1100 x 2 = 0b1100**<u>0</u>**

***Key Idea*: inserting 0 at the end multiplies by the bases**

1450 / 10 = 145

0b1100 / 2 = 0b0110

***Key Idea*: removing 0 at the end divides by the base!**

Hexadecimal

- When working with bits, we can have large numbers with up to 64 bits.

1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111

Umm, let's not....

Hexadecimal

- When working with bits, we can have large numbers with up to 64 bits.
- Instead, we'll represent bits in *base-16 instead;* this is called **hexadecimal**.

0110 1010 0011

0-15 0-15 0-15

Every 4 bits is a base-16 digit!

Hexadecimal

- Hexadecimal is *base-16*, so we need digits for 1-15.

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
										10	11	12	13	14	15

Quick Pneumonic:

0x*f*, means the bits are *full* and there are *four*: *0b1111 == 0xf*

Hexadecimal

- We can distinguish hexadecimal numbers by prefixing them with **`0x`**

`1523` Base-10: Human-readable, but cannot easily interpret on/off bits

`0b10111110011` Base-2: Yes, computers use this, but not human-readable

`0x5F3` Base-16: Easy to convert to Base-2, More "portable" as a human-readable format

(fun fact: a half-byte is called a nibble)

Poll Everywhere

pollev.com/cis2400

- What is 0b110101110100 in hex?

A. 0xD74

B. 0x6BA

C. 0x45D

D. 0x2EB

E. Tbh, I'm not sure

Decimal	Binary	Hex
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xA
11	1011	0xB
12	1100	0xC
13	1101	0xD
14	1110	0xE
15	1111	0xF

Poll Everywhere

pollev.com/cis2400

- What is 0b110101110100 in hex?

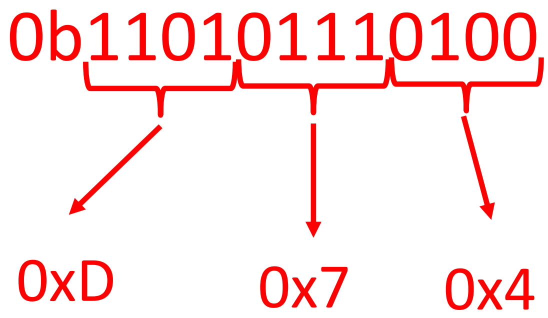
A. 0xD74

B. 0x6BA

C. 0x45D

D. 0x2EB

E. Tbh, I'm not sure

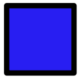
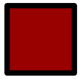


Decimal	Binary	Hex
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xA
11	1011	0xB
12	1100	0xC
13	1101	0xD
14	1110	0xE
15	1111	0xF

Hex Spelling (Hexspeak)

- ***0x8BADF00D***
  - ***"ate bad food"***
    - Used by Apple in iOS crash reports, when an application takes too long to launch, terminate, or respond to system event
- ***0xDEADBEEf***
    - Originally used to mark areas of memory that had not yet been initialized
- ***0xDEADFA11***
  - ***"dead fall"***
    - *Used by Apple in iOS crash reports, when the user force quits an application*
- ***0x0000CACA***
  - ***"Caca"***
    - ***Just for fun***

Encoding

- We can represent more than just numbers with bits
  - We just need an agreed upon ***encoding***
- Decimal Numbers
  - 0 → **`0x00`**, 1 → **`0x01`**, … , 240 → **`0xF0`** …
- Characters
  - A → **`0x41`**, B → **`0x42`**, C → **`0x43`**, …
- Colors
  -  → 0x281EF2,  → 0x990000

The Meaning of Bits

- *A sequence of bits can have many meanings!*

- Consider the hex sequence 0x4E6F21
  - Common interpretations include:
  - The decimal number 5140257
  - The characters “No!”
  - The background color of this slide
  - The real number 7.203034 $\times 10^{-39}$

- A series of bits can also be code!
- Eg. 0x94000005 means bl 0x100003f90 <_printf.....>
- It is up to the program/programmer to decide how to ***interpret*** the sequence of bits

ASCII

- We can encode binary values to represent characters

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

ASCII Design

- ASCII:
  American Standard Code for Information Interchange

- Designed to communicate American letters, numbers, and some control signals efficiently
  - Used only 7 bits to minimize number of bits that need to be communicated
  - Other languages not considered

Unicode

- Unicode Standard UTF-8 is an alternate text encoding
  - Uses between 8 and 32 bits for each “character”
  - Characters include more than just English
  - Characters include emojis 🍿📺❄️
- Unicode table is a lot longer: https://unicode-table.com/en/

Lecture Outline

- How do we count?
  - Bases
- Binary
  - Conversions
  - Hexadecimal
- Unsigned Numbers
- Overflow
- Signed Numbers
  - Two's Complement
  - Two's Complement Overflow

Unsigned Integers

- An **unsigned** integer is 0 or a positive integer (no negatives).
- Converting between decimal and binary, no difference!
- Examples:

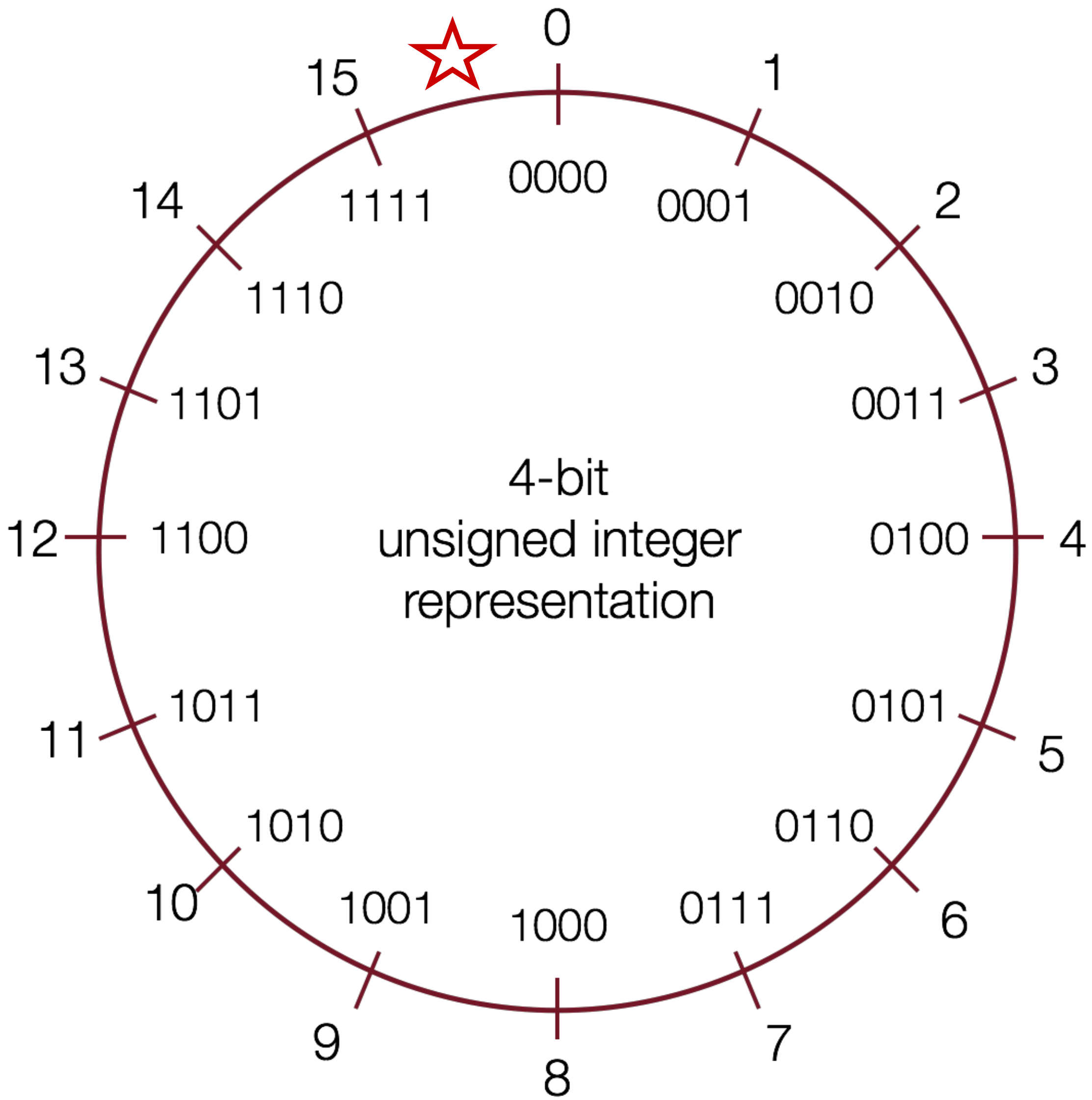
```
0b0001 = 1
0b0101 = 5
0b1011 = 11
0b1111 = 15
```

- The range of an unsigned number is 0 → $2^{w} - 1$
  - where w is the number of bits.
  - E.g. a ***32-bit integer*** can represent 0 to $2^{32} - 1$ (4,294,967,295).

Lecture Outline

- How do we count?
  - Bases
- Binary
  - Conversions
  - Hexadecimal
- Unsigned Numbers
- Overflow
- Signed Numbers
  - Two's Complement
  - Two's Complement Overflow

Unsigned Integers



Overflow

If you exceed the maximum value of your bit representation, you wrap around or ***overflow*** back to the smallest bit representation.

- 0b1111 + 0b1 = 0b0000

If you go below the minimum value of your bit representation, you wrap around or ***overflow*** back to the largest bit representation.

- 0b0000 - 0b1 = 0b1111

Here we're assuming we ***only*** have 4 bits to work with!

Lecture Outline

- How do we count?
  - Bases
- Binary
  - Conversions
  - Hexadecimal
- Unsigned Numbers
- Overflow
- Signed Numbers
  - Two's Complement
  - Two's Complement Overflow

Signed Numbers: Where Are the Negatives?

- ***Problem:*** How can we represent negative *and* positive numbers in binary?

Signed Numbers: Where Are the Negatives?

- ***Problem:*** How can we represent negative *and* positive numbers in binary?

- Ideally, addition would work just like it usually does.

10 + -10 = 0…

Signed Numbers

0b 0 1 0 1 (5 in decimal)

\+ 0b ? ? ? ? (should be -5 in decimal)

0b 0 0 0 0

Signed Numbers

0b 0 1 0 1 (5 in decimal)

+ 0b 1 0 1 0 Here we inverted the bits!

0b 1 1 1 1 Um this isn't 0?

Signed Numbers

0b 1 1 1 1

\+ 0b ? ? ? ?

What do we need to add to make it 0?

Signed Numbers

```
  0b 1 1 1 1
+ 0b 0 0 0 1
-------------
  0b 0 0 0 0
```

Remember: this happens because of overflow!

Signed Numbers

0b 0 1 0 1 (5 in decimal)

\+ 0b 1 0 1 1 So let's add 1 to what we inverted before!

0b 0 0 0 0

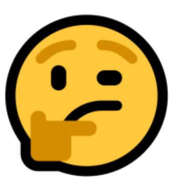
Signed Numbers

And we're done!

0b 0 1 0 1 = 5

0b 1 0 1 1 = -5

Wait...isn't this also 11?



Signed Numbers

The negative number is the positive number **inverted, plus one!**

A binary number plus its inverse is all 1s.

```
  0b 0 1 0 1
+ 0b 1 0 1 0
------------
  0b 1 1 1 1
```

Add 1 to this to carry over all 1s and get 0!

```
  0b 1 1 1 1
+ 0b 0 0 0 1
------------
  0b 0 0 0 0
```

Signed Numbers

The negative number is the positive number **inverted, plus one!**

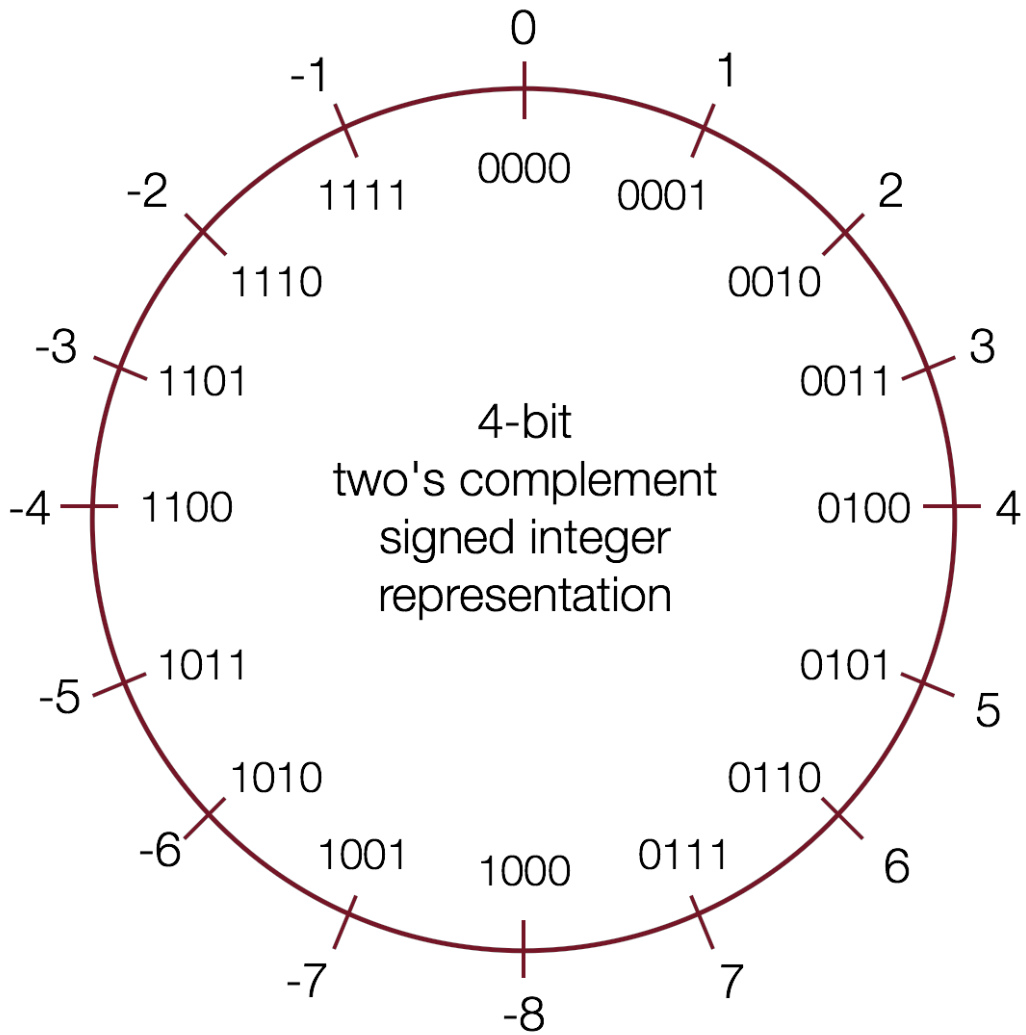
A binary number plus its inverse is all 1s.

```
  0b 0 1 0 1
+ 0b 1 0 1 0
  ----------
  0b 1 1 1 1
```

Add 1 to this to carry over all 1s and get 0!

```
  0b 1 1 1 1
+ 0b 0 0 0 1
  ----------
  0b 0 0 0 0
```

Two's Compliment



Two's Compliment

- Here, we represent a positive number as **itself**, and its negative equivalent as the **two's complement of itself.**
- **The two's complement of a number is the binary digits inverted, plus 1.**
- **A nice consequence is all negative numbers have a 1 in the Most Significant Bit.**
- You can use this to go from positive to negative and negative to positive.
  - E.g. 0b1111 -> (invert) 0b0000 -> (plus 1) 0b0001
  - From -1 to 1.

Poll Everywhere

pollev.com/cis2400

- What is the Two's Compliment of 0b10011011?

A. 0b10111010

B. 0b11100101

C. 0b01100101

D. 0b11111110

E. Tbh, I'm not sure

1st Step: Invert Bits
0b01100100

2nd Step: Add one
0b01100101

Size Determines Range

Type	Size (Bytes)	Minimum	Maximum
char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	8	-9223372036854775808	9223372036854775807
unsigned long	8	0	18446744073709551615

We still have overflow issues…

Fiora
@FioraAeterna · **Follow**

248 days == 2^31 100ths of a second.

even in 2015, our airplanes have integer overflow bugs

> **Ben Goldacre** @bengoldacre
> If you leave your Boeing 787 switched on for 248 days the power shuts off and you fall out of the sky. Epic bug.
> theguardian.com/business/2015/...

8:06 AM · May 1, 2015

472 Reply Share

Read 36 replies

Gangnam Style overflows INT_MAX, forces YouTube to go 64-bit

Psy's hit song has been watched an awful lot of times.

ARS STAFF - 12/3/2014, 5:32 PM

Go on, watch it.

Signed vs Unsigned Types

- By default, all standard types are signed.
  - Int, Char, Long, Double
- There are many ways to declare unsigned types.

```
char x = 'a';
unsigned char x = 10;

int x = -2400;
unsigned int x = 2400;

//and you get the idea…
```

Note: the size of the type and it's "signess" determine the range it can represent

Bit Representations

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
  //a is 97 in ascii
  char x = 'a';
  printf("x is 0x%x.\n", x);

  x = -x;
  printf("x is 0x%x.\n", x);
  return EXIT_SUCCESS;
}
```

sign_example.c

Let's see what exactly is printed…

Bit Representations

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
  //a is 97 in ascii
  char x = 'a';
  printf("x is 0x%x.\n", x);

  x = -x;
  printf("x is 0x%x.\n", x);
  return EXIT_SUCCESS;
}
```

sign_example.c

Let's see what exactly is printed…

In general:
"*`x is 0x61.`*"
"*`x is 0x9f.`*"

Bit Operator: & (and)

1 & 1 = 1

1 & 0 = 0

0 & 1 = 0

0 & 0 = 0

Only if both bits
are one, will it stay one!

Bit Operator: & (and)

```
  0b 0 1 0 1
& 0b 1 1 0 1
------------
  0b 0 1 0 1
```

Bit Operator: | (or)

1 | 1 = 1

1 | 0 = 1

0 | 1 = 1

0 | 0 = 0

If either bits are one, will evaluate to one

Bit Operator: | (or)

```
  0b 0 1 0 1
| 0b 1 1 0 1
  ----------
  0b 1 1 0 1
```

Bit Operator: ^ (xor)

1 ^ 1 = 0

1 ^ 0 = 1

0 ^ 1 = 1

0 ^ 0 = 0

ONLY IF ONE BIT is one, will evaluate to one

Bit Operator: ^ (xor)

```
    0b 0 1 1 1
^   0b 1 1 0 1
--------------
    0b 1 0 1 0
```

Bit Operators

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
  char x = 0xff;
  char y = 0xf0;
  char z = x & y;
  printf("The value of z is %x\n.", z);
  return EXIT_SUCCESS;
}
```

bit_ops.c

What is char z in binary?

In general:

Z will be 0b11110000

```
  0b11111111
& 0b11110000
------------
  0b11110000
```

Bit Operators

Consider the following code:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
  //a is 97 in ascii
  char x = 0xf0;
  char y = 0xf1;
  char z = x ^ y;
  printf("The value of z is %x\n.", z);
  return EXIT_SUCCESS;
}
```

bit_ops.c

What is char z in binary?

In general:
Z will be 0b00000001

```
  0b11110000
^ 0b11110001
------------
  0b00000001
```

C IS NOT JAVA

DO NOT DO THIS

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {

  int ube  = foo();
  int miso = fuh();
  if(ube & miso)
  …
  …
  …
```

& IS NOT A LOGICAL OPERATOR!!

IT IS FOR BITWISE OPERATIONS!!!

It will ***literally*** evaluate to the bit value.

More Bit Operators: ~ (not)

~ 0b 0 1 0 1

0b 1 0 1 0

This operation negates the bits!

More Bit Operators: << (left shift)

0b 0 0 1 0 1 << 1

0b 0 1 0 1 0

This operation shifts the bits ***n*** many times to the left.

More Bit Operators: >> (right shift)

0b 0 0 1 0 1 >> 1

0b 0 0 0 1 0

What happened to the LSB?

This operation shifts the bits ***n*** many times to the right.

What about Logical (Boolean) Operators?

- C doesn't have Booleans! (Technically…)
- Traditionally, just use an ***int*** to represent 1 for true and 0 for false.

&& Logical And || Logical Or ! Logical Not

X	Y	X && Y
T	T	T
T	F	F
F	T	F
F	F	F

X	Y	X || Y
T	T	T
T	F	T
F	T	T
F	F	F

X	! X
T	F
F	T

What about Logical (Boolean) Operators?

&& Logical And `if(X && Y)`

|| Logical Or `if(X || Y)`

! Logical Not `if(!X)`

Poll Everywhere

pollev.com/cis2400

Talk to your neighbor: what will be printed?

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {

  int x = 0xf0;
  int y = 0x0f;

  if( x & y){
    printf("First print!\n");
  }
  if( x && y){
    printf("Second print!\n");
  }
  return EXIT_SUCCESS;
}
```

bitop_v_logic.c

A. `First print!`

B. `Second print!`

C.
```
First print!
Second print!
```

D.
```
Second First
print!
print!
```

Lecture Take-aways

- We can represent anything in binary by using different encodings!
  - Numbers, colors, characters, emojis, code, etc..
- Hexadecimal is more human friendly…
- Our encodings/data is limited due to finite bits
  - Especially, when we are explicit about the types we use.
- Unsigned Numbers are non-negative integers
- Signed numbers use Two’s Compliment to represent negative numbers
- Bitwise operators allow you to manipulate individual bits.