Data representation: Bits, Bytes, Integers

Yipeng Huang

Rutgers University

October 2, 2025

Table of contents

Failure to free

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

int main () {

    int* pointer0 = malloc(sizeof(int));
    *pointer0 = 100;
    printf("*pointer0 = %d\n", *pointer0);

}
```

Note: calloc() functions like malloc(), but calloc() initializes memory to zero while malloc() offers no such guarantee.

Memory leaks

Have you ever had to restart software or hardware to recover it?

Debug by compilation in GCC, running with Valgrind, Address Sanitizer

Use after free

```
1     int* pointer0 = malloc(sizeof(int));
2
3     printf("pointer0 = %p\n", pointer0);
4     *pointer0 = 100;
5     printf("*pointer0 = %d\n", *pointer0);
6
7     free(pointer0);
8     pointer0 = NULL;
9
10    printf("pointer0 = %p\n", pointer0);
11    *pointer0 = 10;
12    printf("*pointer0 = %d\n", *pointer0);
```

Dangling pointers

- One defensive programming style is to set any freed pointer to NULL.
- Debug by running with Valgrind, Address Sanitizer.

Pointer aliasing

```
1     int* pointer0 = malloc(sizeof(int));
2     int* pointer1 = pointer0;
3
4     *pointer0 = 100;
5     printf("*pointer1 = %d\n", *pointer1);
6
7     *pointer0 = 10;
8     printf("*pointer1 = %d\n", *pointer1);
9
10    free(pointer0);
11    pointer0 = NULL;
12
13    *pointer1 = 1;
14    printf("*pointer1 = %d\n", *pointer1);
```

Debug by running with Valgrind, Address Sanitizer

Pointer aliasing and overhead of garbage collection

- Java garbage collection tracks dangling pointers but costs performance.
- C requires programmer to manage pointers but is more difficult.

Pointer typing

```
1     unsigned char n = 2;
2     unsigned char m = 3;
3
4     unsigned char ** p;
5     p = calloc( n, sizeof(unsigned char) );
6
7     for (int i = 0; i < n; i++)
8         p[i] = calloc( m, sizeof(unsigned char) );
9
10    for (int i = 0; i < n; i++)
11        for (int j = 0; j < m; j++) {
12            p[i][j] = 10*i+j;
13            printf("p[%d][%d] = %d\n", i, j, p[i][j]);
14        }
```

Defend using explicit pointer casting.

Table of contents

Non existent memory

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main () {
5
6    int **x = malloc(sizeof(int*));
7    **x = 8;
8    printf("x = %p\n", x);
9    printf("*x = %p\n", *x);
10   printf("**x = %d\n", **x);
11   fflush(stdout);
12
13 }
```

Debug by running with Valgrind, Address Sanitizer

Returning null pointer

```
1
2  int* returnsNull () {
3    int val = 100;
4    return &val;
5  }
6
7  int main () {
8
9    int* pointer = returnsNull();
10   printf("pointer = %p\n", pointer);
11   printf("*pointer = %d\n", *pointer);
12
13 }
```

Prevent using -Werror compilation flag.

Table of contents

Future computer architectures: A quantum pointer?

For $n = 1$, four possibilities

	f_0	f_1	f_2	f_3
f(0)	0	0	1	1
f(1)	0	1	0	1
	f is constant 0	f is balanced	f is balanced	f is constant 1

A simple physics experiment that classical computing cannot replicate

Algorithm

David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. 1992.

Implementation

- Mach-Zehnder interferometer implementation.
  `https://www.st-andrews.ac.uk/physics/quvis/simulations_html5/sims/SinglePhotonLab/SinglePhotonLab.html`

Mathematical description of the algorithm

$$|0\rangle \xrightarrow{H} |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \begin{cases} \xrightarrow{I} |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} |0\rangle \\ \xrightarrow{Z} |-\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} |1\rangle \\ \xrightarrow{-Z} -|-\rangle = \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} -|1\rangle \\ \xrightarrow{-ZZ=-I} -|+\rangle = \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} -|0\rangle \end{cases}$$

The binary abstraction: classical and quantum

High, low voltage

Adds resilience against noise.

Representation as a state vector

- $\begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$ We pronounce this "ket" 0
- $\begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$ We pronounce this "ket" 1

The Hadamard gate

Matrix representation of Hadamard operator: $H = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix}$

- $H|0\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$
- $H|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$

Superposition

Single qubit state

- $\alpha |0\rangle + \beta |1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$
- Amplitudes $\alpha, \beta \in \mathbb{C}$
- $|\alpha|^2 + |\beta|^2 = 1$
- The above constraints require that qubit operators are unitary matrices.

Many physical phenomena can be in superposition and encode qubits

- Polarization of light in different directions
- Electron spins (Intel solid state qubits)
- Atom energy states (UMD, IonQ ion trap qubits)
- Quantized voltage and current (IBM, Google superconducting qubits)

If multiple discrete values are possible (e.g., atom energy states, voltage and current), we pick (bottom) two for the binary abstraction.

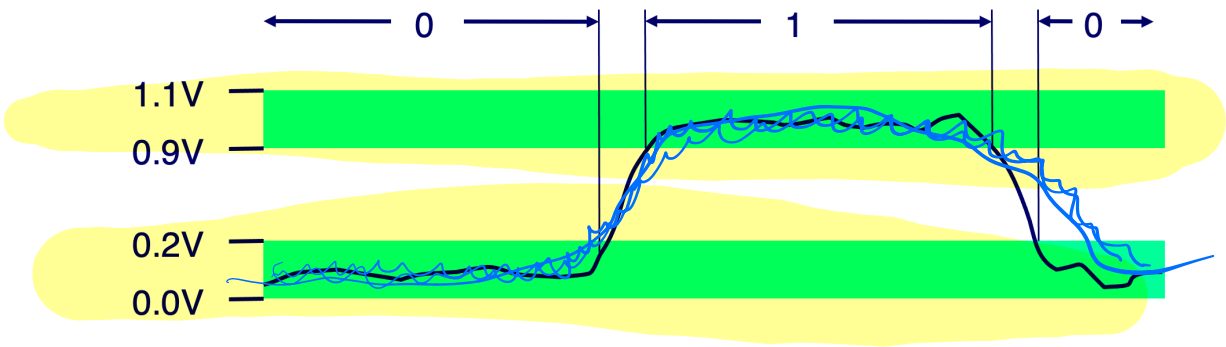
Mathematical description of the algorithm

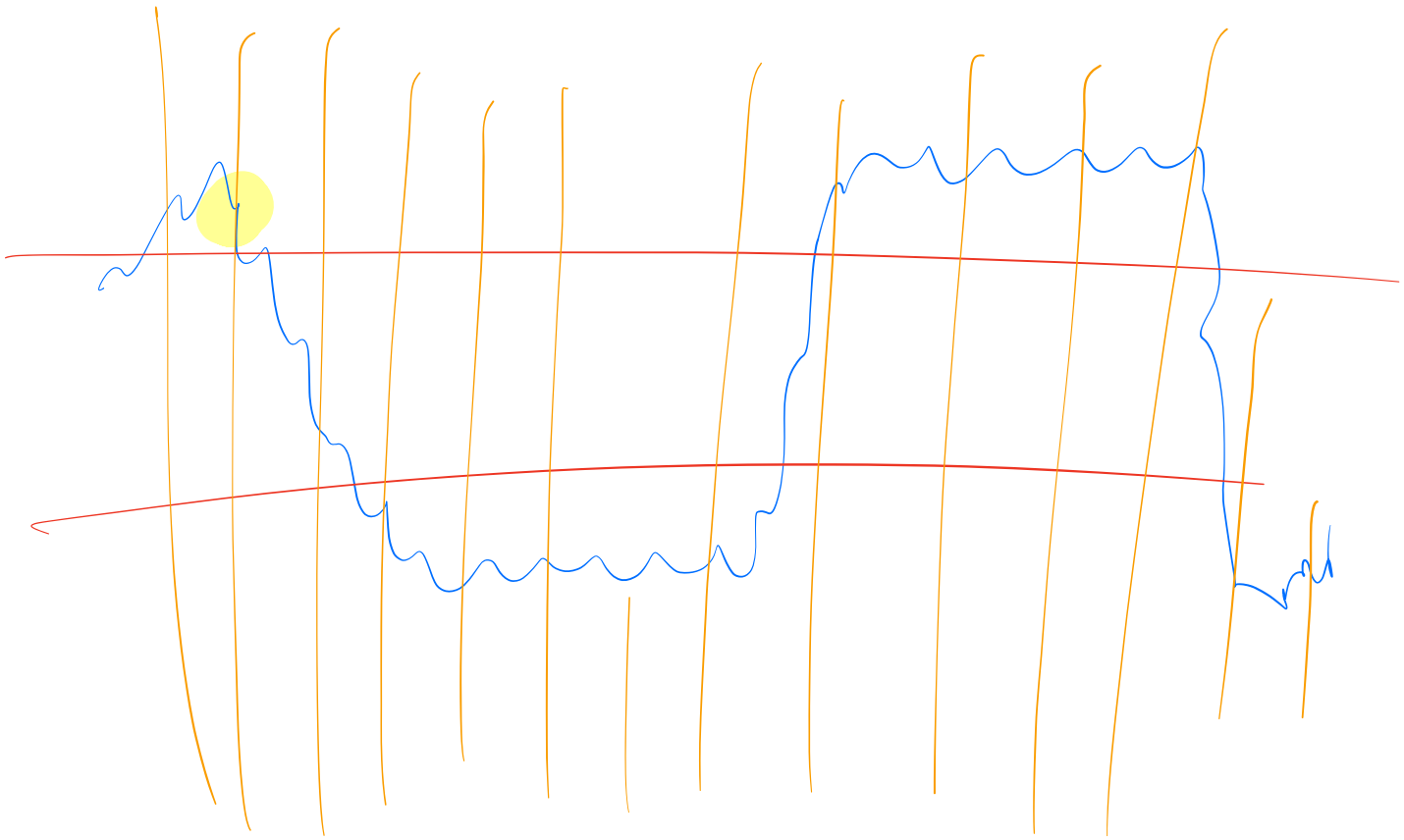
$$|0\rangle \xrightarrow{H} |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \begin{cases} \xrightarrow{I} |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} |0\rangle \\ \xrightarrow{Z} |-\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} |1\rangle \\ \xrightarrow{-Z} -|-\rangle = \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} -|1\rangle \\ \xrightarrow{-ZZ=-I} -|+\rangle = \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} -|0\rangle \end{cases}$$

Table of contents

Everything is bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- **Why bits? Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

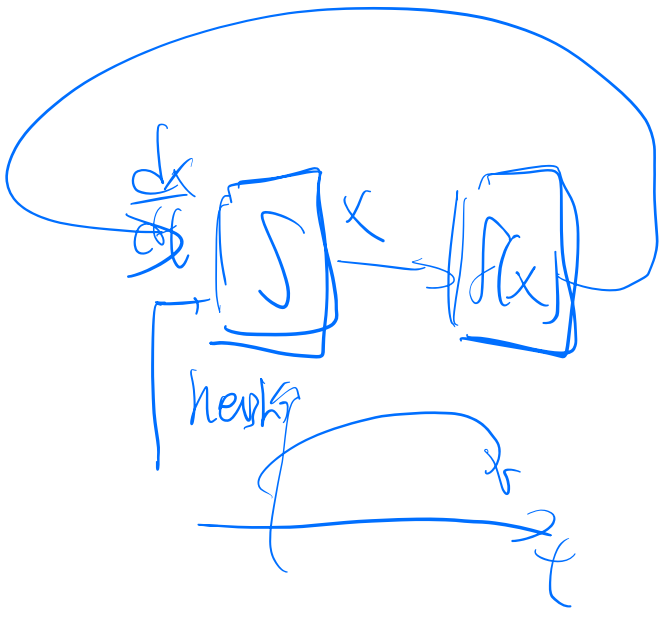


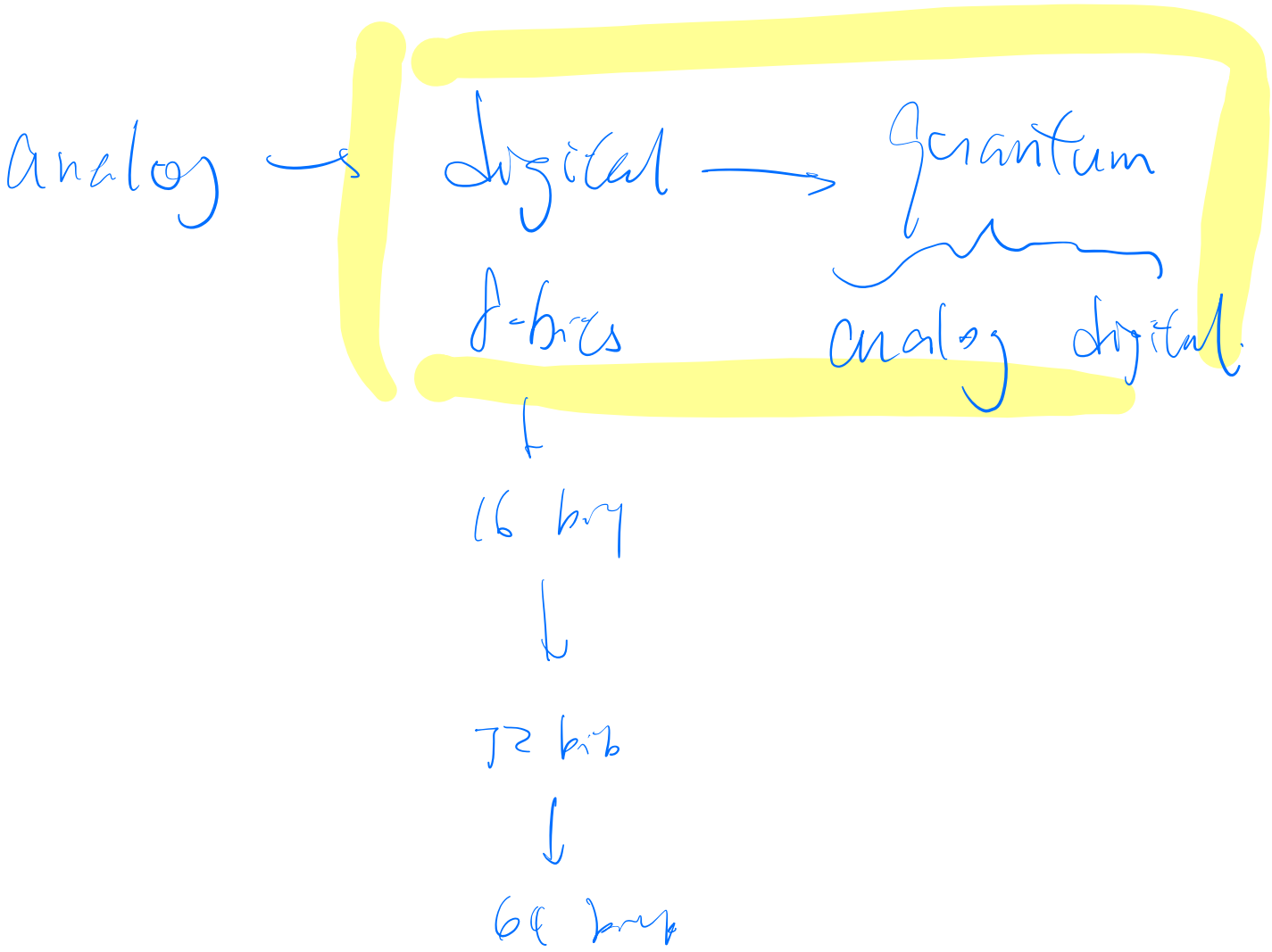


$4\,GHz = 4.0 \times 10^9\ Hz$

$\frac{1}{4\,GHz} = 0.25\ ns$

$$\frac{dx}{dt} = f(x)$$




Analog
Digital
Quantum
8-bits
Analog
digital
16 bit
32 bit
64 bit

Decimal, binary, octal, and hexadecimal

Decimal	Binary	Octal	Hexadecimal	Decimal	Binary	Octal	Hexadecimal
0	0b0000	0o0	0x0	8	0b1000	0o10	0x8
1	0b0001	0o1	0x1	9	0b1001	0o11	0x9
2	0b0010	0o2	0x2	10	0b1010	0o12	0xA
3	0b0011	0o3	0x3	11	0b1011	0o13	0xB
4	0b0100	0o4	0x4	12	0b1100	0o14	0xC
5	0b0101	0o5	0x5	13	0b1101	0o15	0xD
6	0b0110	0o6	0x6	14	0b1110	0o16	0xE
7	0b0111	0o7	0x7	15	0b1111	0o17	0xF

In C, format specifiers for printf() and fscanf():

1. decimal: '%d'
2. binary: none
3. octal: '%o'
4. hexadecimal: '%x'

decimal 65

↓

$6 \times 10 + 5 \times 1$

binary

$\cdots 0 \times 128 + 1 \times 64 + 0 \times 32 + 0 \times 16 + 0 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1$

$= 0b01000001_2$

hex

$0 \times 256 + 4 \times 16 + 1 \times 1$

$0x41_{16}$

hex $0 \times 256 + 4 \times 16 + 11 \times 1$

$= 64 + 11 = 75$

$0x4B$

Octal

64

$1 \times 64 + 0 \times 8 + 1 \times 1$

$\Rightarrow$ 0101_8

0b1111_1111

1×128 + 1×64 + 1×32 + 1×16 + 1×8 + 1×4 + 1×2 + 1×1 = 255

$2^8 - 1 = 255$

Decimal, binary, octal, and hexadecimal

How to represent the range of unsigned char in each?

Unsigned char is one byte, 8 bits.

1. decimal: 0 to 255
2. binary: 0b0 to 0b11111111
3. octal: 0 to 0o377 (group by 3 bits)
4. hexadecimal: 0x00 to 0xFF (group by 4 bits)

Often encountered use of hexadecimal: RGB colors

Red, green, blue values ranging from 0-255

#000000	#FFFFFF	#6A757C	??? #CC0033

Often encountered use of hexadecimal: RGB colors

Red, green, blue values ranging from 0-255

#000000	#FFFFFF	#6A757C	#CC0033

Representing characters

- char is a 1-byte, 8-bit data type.
- ASCII is a 7-bit encoding standard.
- "man ascii" to see Linux manual.
- Compile and run ascii.c to see it in action.
- Some interesting characters: 7 (bell), 10 (new line), 27 (escape).

USASCII code chart

$b_7 b_6 b_5$ → / Bits ↓	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 Row ↓ / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

Figure: ASCII character set. Image credit Wikimedia

Bitwise operations

Why are bitwise operations important?

- Network and UNIX settings using bit masks (e.g., umask)
- Hardware and microcontroller programming (e.g., Arduinos)
- Instruction set architecture encodings (e.g., ARM, x86)

Bitwise operations

~: bitwise NOT

unsigned char a = 128

$$
\begin{aligned}
a &= 0b1000\_0000 \\
\sim a &= \sim 0b1000\_0000 \\
&= 0b0111\_1111 \\
&= 127
\end{aligned}
$$

b	~ b
0	1
1	0

Bitwise operations

&: bitwise AND

$$\begin{aligned} 3\&1 &= 0b11\&0b01 \\ &= 0b01 \\ &= 1 \end{aligned}$$

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise operations

| : bitwise OR

$$
\begin{aligned}
3|1 &= 0b11|0b01 \\
&= 0b11 \\
&= 3
\end{aligned}
$$

$$
\begin{aligned}
2|1 &= 0b10|0b01 \\
&= 0b11 \\
&= 3
\end{aligned}
$$

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise operations

^: bitwise XOR

$$\begin{aligned} 3 \wedge 1 &= 0b11 \wedge 0b01 \\ &= 0b10 \\ &= 2 \end{aligned}$$

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

`inplaceSwap.c`: Swapping variables without temp variables.

How does it work?

Don’t confuse bitwise operators with logical operators

Bitwise operators

- ~
- &
- |
- ^

Logical operators

- !
- &&
- ||
- != (for bool type)

Table of contents

Representing negative and signed integers

Ways to represent negative numbers

1. Sign magnitude
2. 1s' complement
3. 2's complement

Representing negative and signed integers

Sign magnitude

Flip leading bit.

Representing negative and signed integers

1s' complement

- Flip all bits
- Addition in 1s' complement is sound
- In this encoding there are 2 encodings for 0
- -0: 0b1111
- +0: 0b0000

Representing negative and signed integers

2's complement

signed char	weight in decimal
00000001	1
00000010	2
00000100	4
00001000	8
00010000	16
00100000	32
01000000	64
10000000	-128

Table: Weight of each bit in a signed char type

- what is the most positive value you can represent? 127
- what is the most negative value you can represent? -128
- how to represent -1? 11111111
- how to represent -2? 11111110

Representing negative and signed integers

2's complement

signed char	weight in decimal
00000001	1
00000010	2
00000100	4
00001000	8
00010000	16
00100000	32
01000000	64
10000000	-128

Table: Weight of each bit in a signed char type

- MSB: 1 for negative
- To make a number negative: flip all bits and add 1.
- Addition in 2's complement is sound

Importance of paying attention to limits of encoding

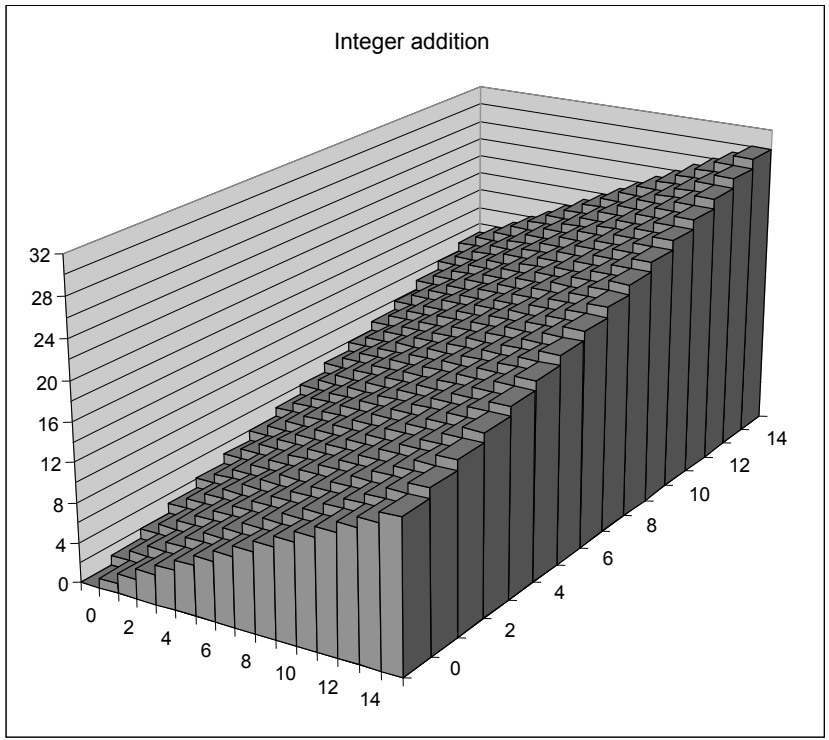


Figure: Image credit: CS:APP

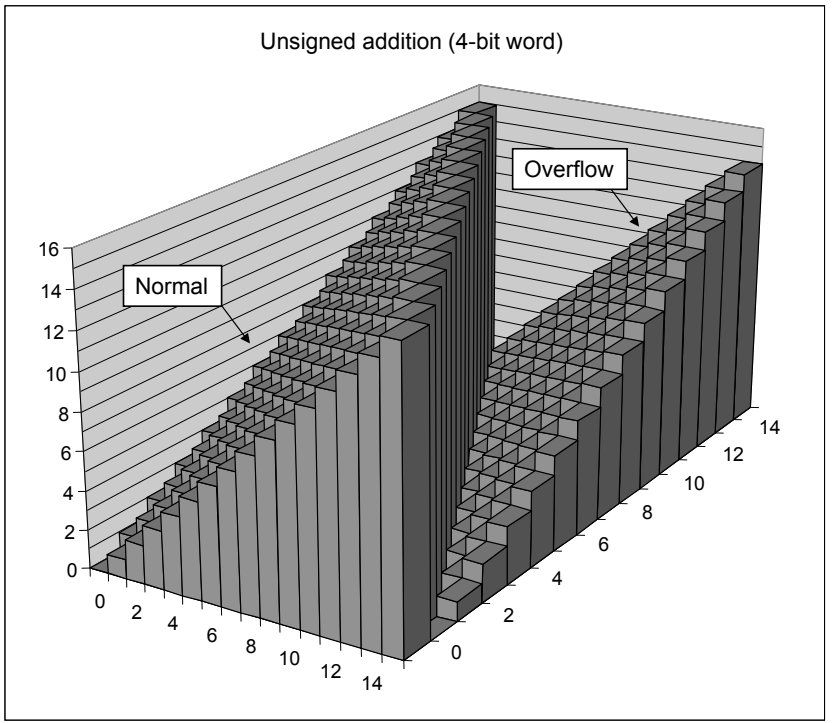


Figure: Image credit: CS:APP

Importance of paying attention to limits of encoding

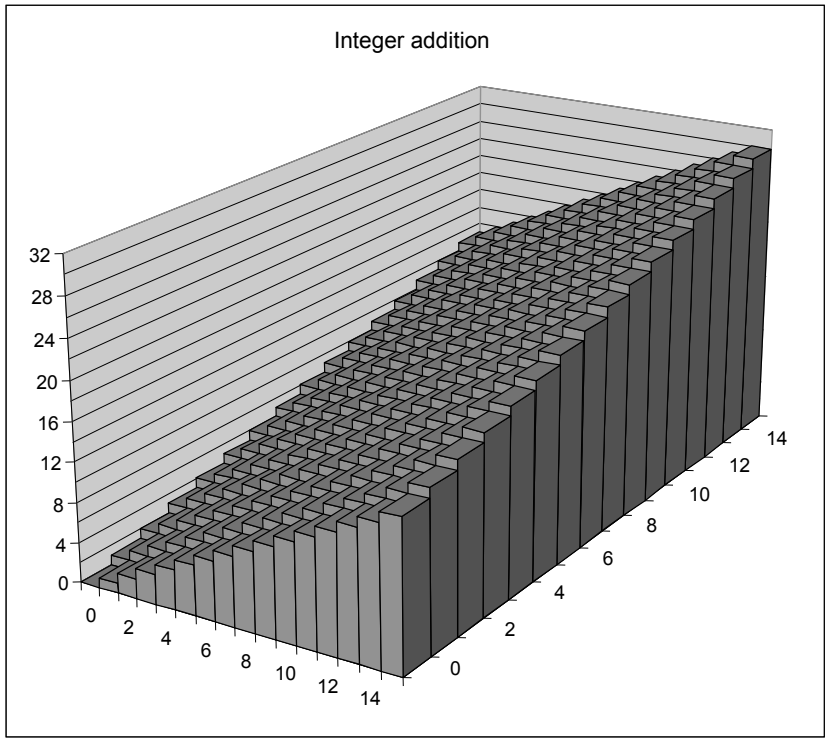


Figure: Image credit: CS:APP

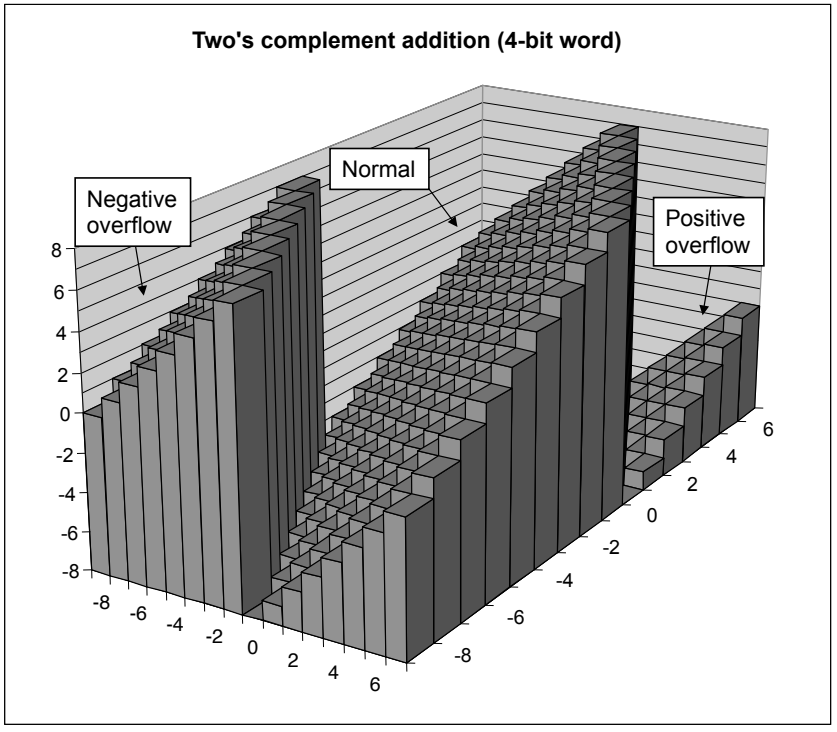


Figure: Image credit: CS:APP

https://www.theatlantic.com/technology/archive/2014/12/how-gangnam-style-broke-youtube/383389/

Table of contents

Programming assignment 2: Graphs, trees, queues, hashes

Programming Assignment 2 parts

1. hashTable: implementing a separate-chaining hash table.
2. edgelist: loading and printing a graph
3. isTree: needs either DFS (stack) or BFS (queue)
4. mst: a greedy algorithm
5. findCycle: needs either DFS (stack) or BFS (queue)
6. matChainMul: a dynamic programming approach to multiplying matrices with minimal operations using recursion

Using `graphutils.h`

- The adjacency list representation
- The edgelist representation
- The query

Binary search tree

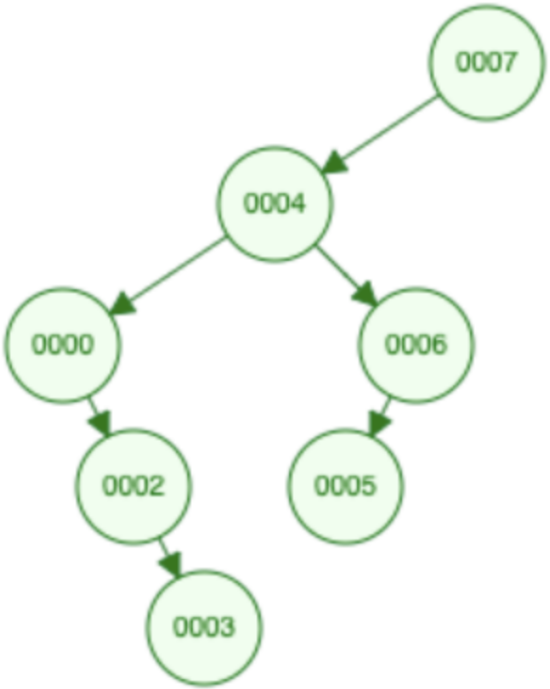


Figure: BST with input sequence 7, 4, 7, 0, 6, 5, 2, 3. Duplicates ignored.

Binary search tree level order traversal

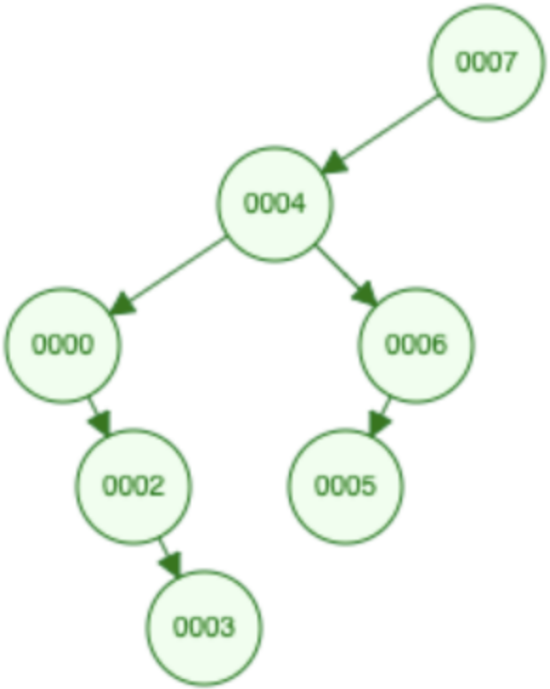


Figure: Level order, left-to-right traversal would return 7, 4, 0, 6, 2, 5, 3.

Binary search tree traversal orders

Breadth-first

- For example: level-order.
- Needs a queue (first in first out).
- Today in class we will build a BST and a Queue.

Depth-first

- For example: in-order traversal, reverse-order traversal.
- Needs a stack (first in last out).
- But in the example code implementation,*where is the stack data structure?*

typedef

Why types are important

- Natural language has nouns, verbs, adjectives, adverbs.
- Type safety.
- Interpretation vs. compilation.

BSTNode

```
typedef struct BSTNode BSTNode;
struct BSTNode {
    int key;
    BSTNode* l_child; // nodes with smaller key will be in left s
    BSTNode* r_child; // nodes with larger key will be in right s
};
```

QueueNode, Queue

```
// queue needed for level order traversal
typedef struct QueueNode QueueNode;
struct QueueNode {
    BSTNode* data;
    QueueNode* next; // pointer to next node in linked list
};
typedef struct Queue {
    QueueNode* front; // front (head) of the queue
    QueueNode* back; // back (tail) of the queue
} Queue;
```

Let's implement `enqueue()`

`https://visualgo.net/en/queue`

- First, consider if queue is empty.
- Then, consider if queue is not empty. Only need to touch back (tail) of the queue.

Let's implement `dequeue()`

`https://visualgo.net/en/queue`

- First, consider if queue will become empty.
- Then, consider if queue will not not empty. Only need to touch front (head) of the queue.

Subtle point: why are the function signatures (return, parameters) of `enqueue()` and `dequeue()` the way they are?