Bits and Bytes

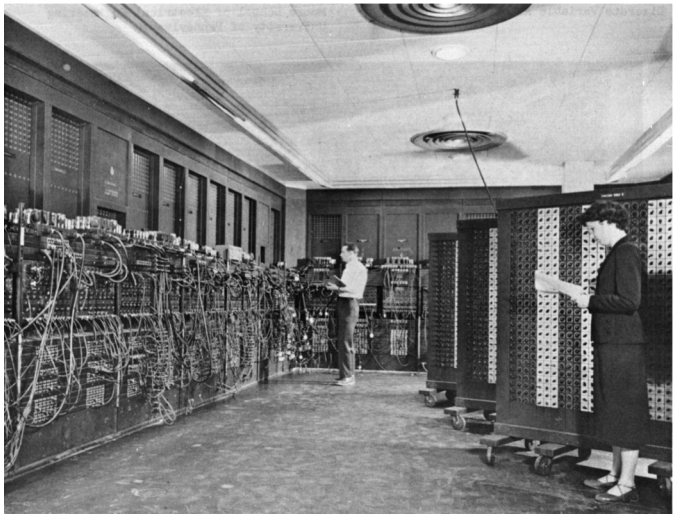
Reading: CS:APP Chapter 2.1

These slides adapted from materials provided by the textbook authors.

Bits, Bytes, and Integers

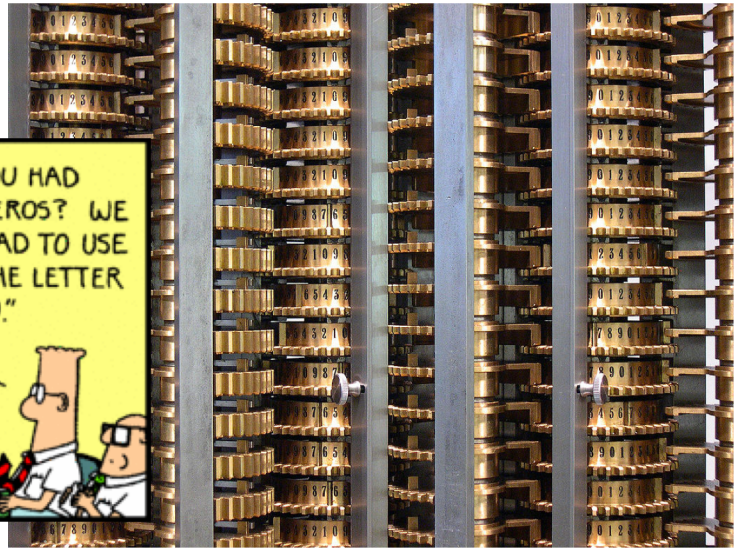
- Early Computers
- Representing information as bits
- Bit-level manipulations

Early Computers Used Many Bases



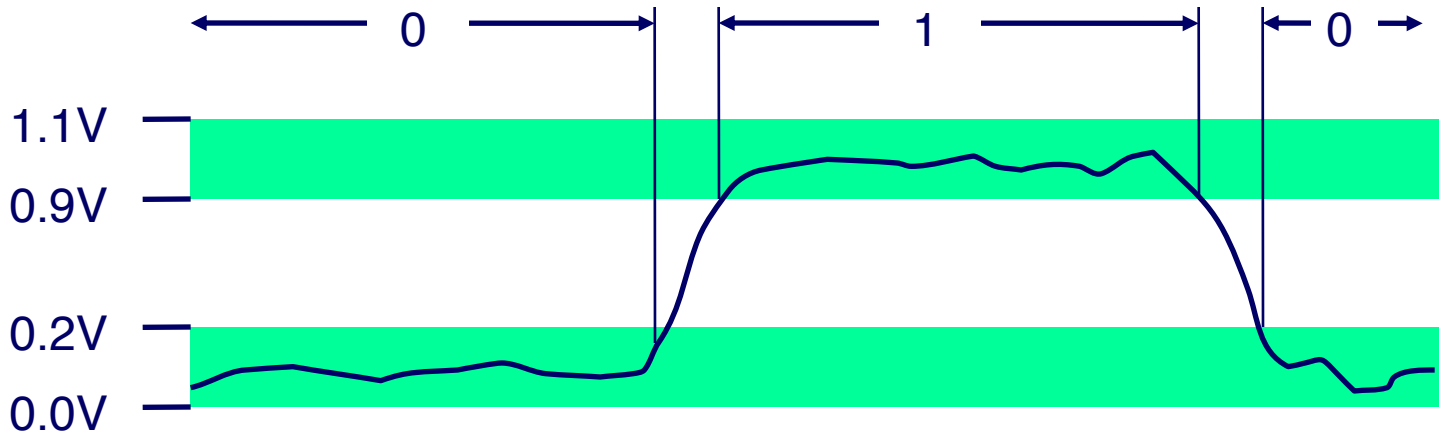
- Babbage 1830's difference engine used decimal
- Fowler's wooden calculating machines in 1840's used Ternary
- ENIAC used 10's complement

https://en.wikipedia.org/wiki/Method_of_complements

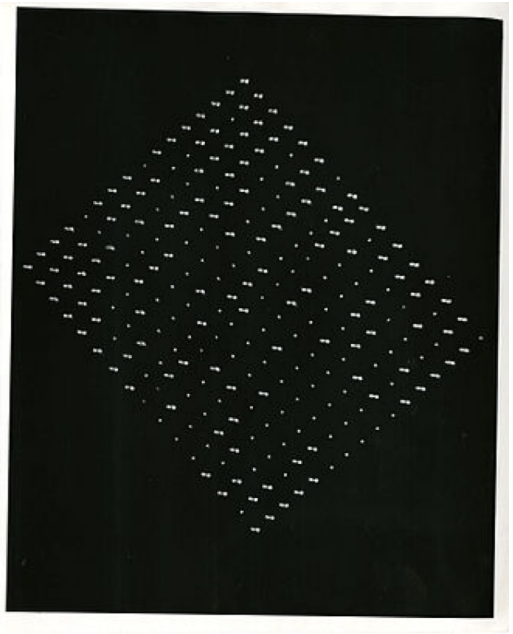


Everything is bits

- **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 bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires

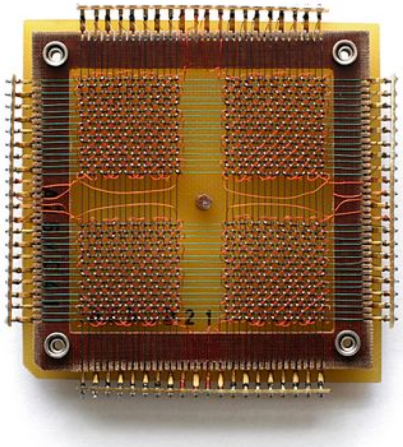


Bits in Real Life



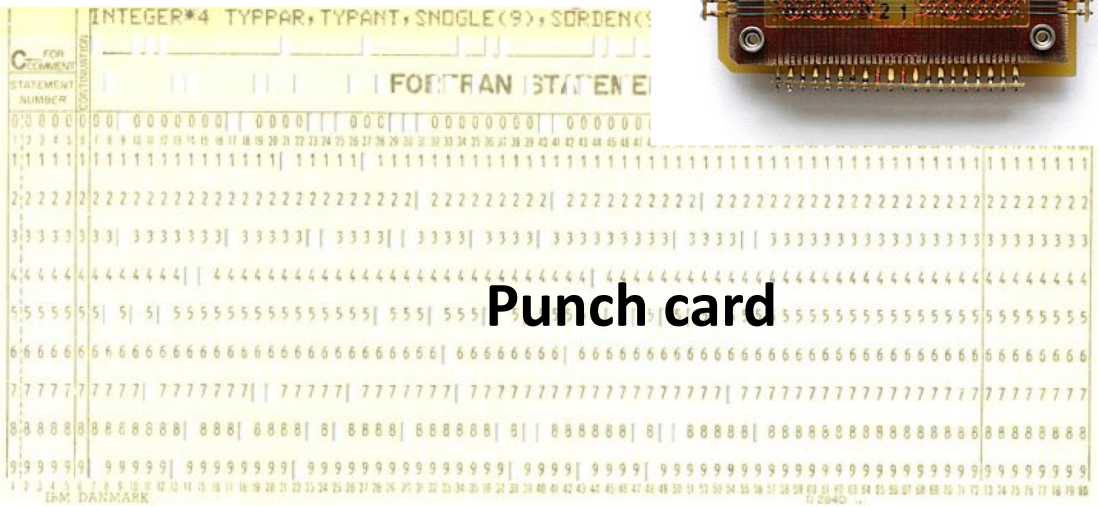
Williams Tube

Core Memory



Paper Tape & TTY

Jaquard Loom



Punch card

(All images from Wikimedia)

For example, can count in binary

- **Base 2 Number Representation**
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$
  - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

Octal	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

- **Octal**
  - Octal (base-8) uses digits 0, 1, 2, 3, 4, 5, 6, 7
  - Represent 11101101101101_2
    $\rightarrow 11\_101\_101\_101\_101_2$
    $\rightarrow 035555_8$
  - In C/C++ leading zero (0) indicates octal rather than decimal
  - Octal still used when manipulating file access (r/w/x)

Encoding Byte Values

- **Hexadecimal 00_{16} to FF_{16}**
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Convert 11101101101101_2
    $11\_1011\_0110\_1101_2$
    $3\ B\ \ 6\ D_{16}$
  - Write $FA1D37B_{16}$ in C as
    - 0xFA1D37B
    - 0xfa1d37b

- **Byte = 8 bits = 2 hex digits**
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}

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

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
`long double`	–	–	10/16
pointer	4	8	8

Today: Bits, Bytes, and Integers

- Representing information as bits
- **Bit-level manipulations**
  - Boolean operations
  - Bit vectors
  - Bit vectors as sets

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
- **Similar rules to integer numbers, but not exactly same**
  - a(b+c) = ab + ac
  - a+(bc) = (a+b)(a+c)

And, &, *

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Or, |, +

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Xor, ^ , ⊕

A	B	A^ B
0	0	0
0	1	1
1	0	1
1	1	0

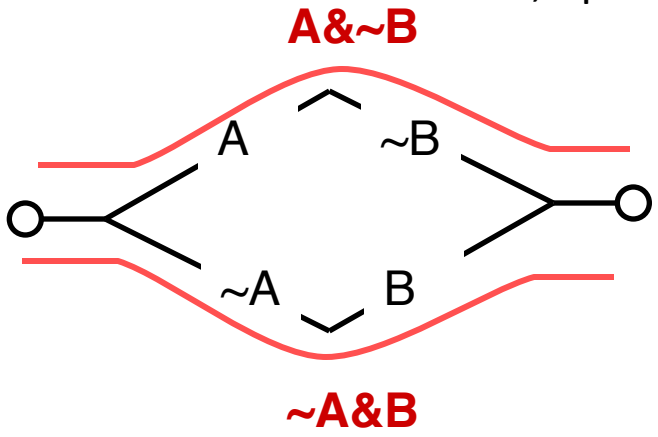
Not, ~

A	~A
0	1
1	0

Application of Boolean Algebra

Applied to Digital Systems by Claude Shannon

- 1937 MIT Master's Thesis
- Reasoned about networks of relay switches
  - Encode closed switch as 1, open switch as 0



Connection when

A&~B | ~A&B

= A^B

- Confusingly, there are multiple notations used:
  - C: P & Q, P | Q, P^Q, ~P
  - ECE-land: PQ, P+Q, $p \oplus q$, $\bar{P}$
  - Symbolic logic-land': $p \oplus q \quad = \quad (p \vee q) \wedge \neg(p \wedge q)$

General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  --------     --------     --------     --------
  01000001     01111101     00111100     10101010
```

- **All of the Properties of Boolean Algebra Apply**

Example: Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - 01101001 { 0, 3, 5, 6 }
    - *76543210*
    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- **Operations**

&	Intersection	01000001	{ 0, 6 }
|	Union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	Symmetric difference	00111100	{ 2, 3, 4, 5 }
~	Complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41 ⇒ 0xBE`
    - $\sim 01000001_2 \Rightarrow 10111110_2$
  - `~0x00 ⇒ 0xFF`
    - $\sim 00000000_2 \Rightarrow 11111111_2$
  - `0x69 & 0x55 ⇒ 0x41`
    - $01101001_2 \;\&\; 01010101_2 \Rightarrow 01000001_2$
  - `0x69 | 0x55 ⇒ 0x7D`
    - $01101001_2 \mid 01010101_2 \Rightarrow 01111101_2$

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
- **Examples (char data type)**
  - `!0x41 ⇒ 0x00`
  - `!0x00 ⇒ 0x01`
  - `!!0x41 ⇒ 0x01`
  - `0x69 && 0x55 ⇒ 0x01`
  - `0x69 || 0x55 ⇒ 0x01`
  - `p && *p` (avoids null pointer access)

Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
- **Examples (char data type)**
  - `!0x41 → 0x00`
  - `!0x00 → 0x01`
  - `!!0x41 → 0x01`
  - `0x69 && 0x55 → 0x01`
  - `0x69 || 0x55 → 0x01`
  - `p && *p` (avoids null pointer access)

Watch out for && vs. &, || vs. |, and = vs. == … one of the more common oopsies in C programming

Masks and Shifting Bit Vectors

- **Bit vectors are commonly used for *masks***

- **Typically involves shifting bit vectors**
  - 10111110_2 << 3 becomes 11110000_2
  - 10111110_2 >> 3 becomes 00010111_2

    or 11110111_2
  - **Logical or arithmetic shift depends on the "integer representation", but masking idiom is very common**

Bit-wise Programming Idioms

Extract Last Byte

- **Task:** Given hex value like `0xb01dface`, extract last byte (‘ce’)
- `0xb01dface & 0xff`

Extract All but Last Digit

- **Task:** Given hex value like `0xb01dface`, extract last byte (‘ce’), e.g. `0xb01dfa00`
- `0xb01dface & ~0xff`

Bit-wise Programming Idioms

Extract Byte w/ Shift & Mask

- **Task:** Given hex value like 0xb01dface, extract 2nd to last byte (0x**fa**)
- (0xb01d**fa**ce >> 8) & 0xff
- After shift, value is 0xb01dfa
- After mask, 0xb01dfa & 0xff -> 0xfa

Change byte w/Shift

- **Task:** Given hex value like 0xb01dface, change 2nd to last byte (0xfa) to 0xbd
- 0xb01d**fa**ce & (~(0xff << 8))
- Result is 0xb01d**00**ce
- Then, put in 0xbd 0xb01d**00**ce | (0xbd << 8) -> 0xb01d**bd**ce