Bits and Bytes

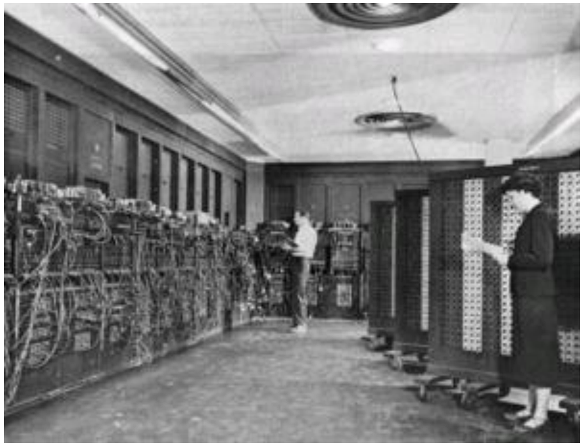
NORTHWESTERN UNIVERSITY
QUAECUMQUE SUNT VERA
1851

Today

- Why bits?
- Binary/hexadecimal
- Byte representations
- Boolean algebra
- Expressing in C

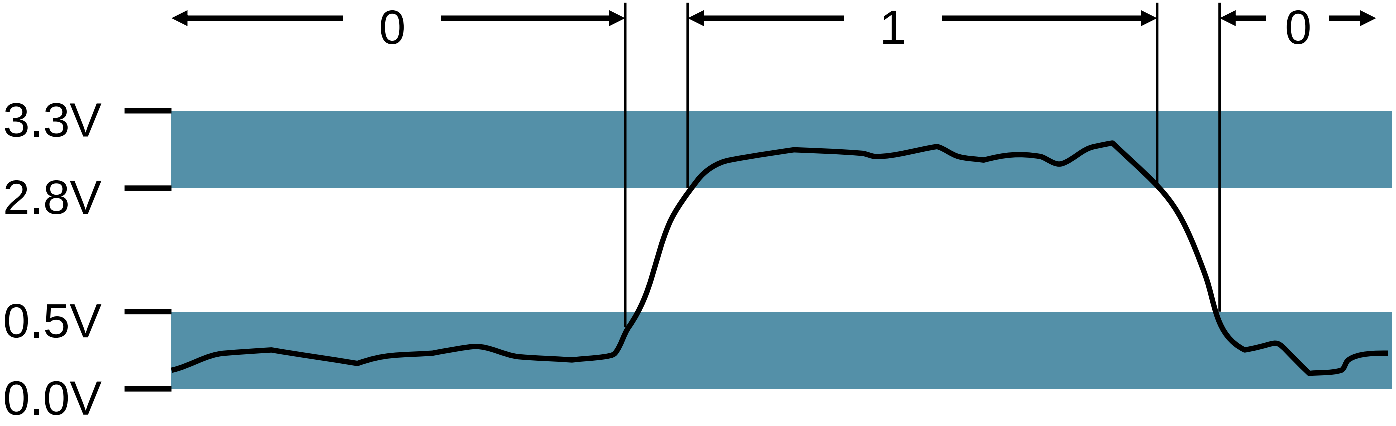
Why don’t computers use Base 10?

- Base 10 number representation
  - “Digit” in many languages also refers to fingers (and toes)
    - Of course, decimal (from Latin decimus) , means tenth
  - A position numeral system (unlike, say Roman numerals)
  - Natural representation for financial transactions
  - Even carries through in scientific notation
- Implementing electronically
  - Hard to store
    - ENIAC (First electronic computer) used 10 vacuum tubes / digit
  - Hard to transmit
    - Need high precision to encode 10 signal levels on single wire
  - Messy to implement digital logic functions
    - Addition, multiplication, etc.



Binary representations

- Base 2 number representation
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$
- Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



  - Straightforward implementation of arithmetic functions

Byte-oriented memory organization

- Programs refer to virtual addresses
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
  - In Unix and Windows NT, address space private to particular “process”
    - Program being executed
    - Program can manipulate its own data, but not that of others
- Compiler + run-time system control allocation
  - Where different program objects should be stored
  - Multiple mechanisms: static, stack, and heap
  - In any case, all allocation within single virtual address space

How do we represent the address space?

- Hexadecimal notation
- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as `0xFA1D37B`
      - Or `0xfa1d37b`

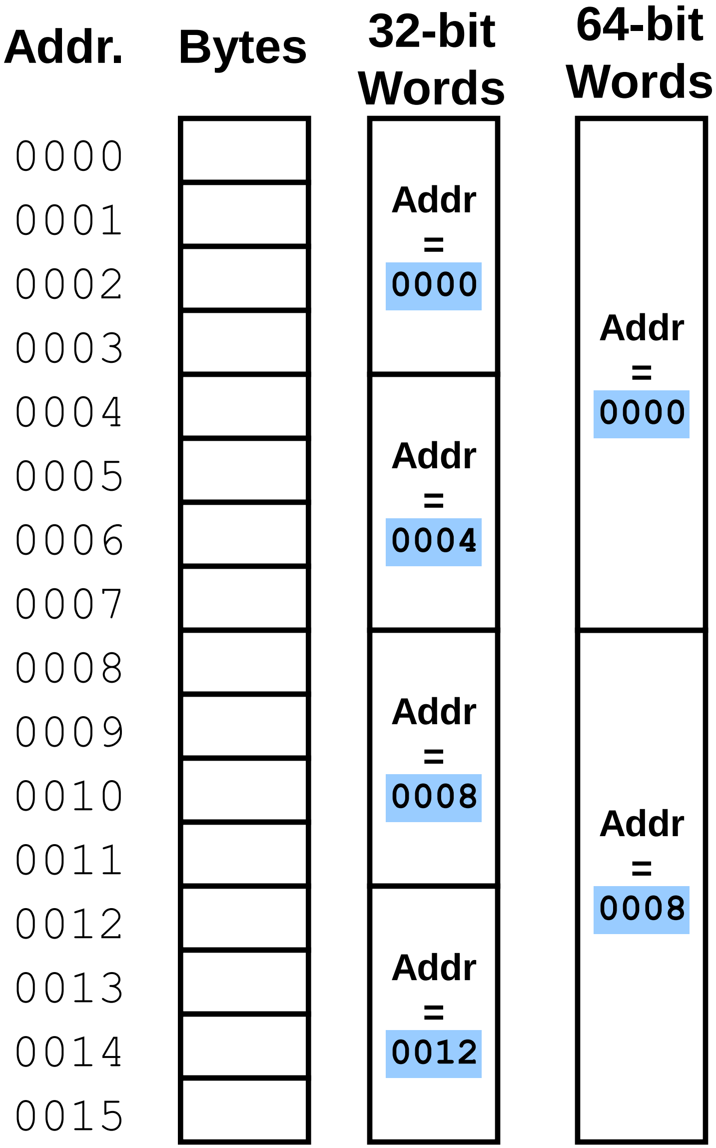
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Machine words

- Machine has “word size”
  - Nominal size of integer-valued data
    - Including addresses
    - A virtual address is encoded by such a word
  - Most current machines are 32 bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Newer systems are 64 bits (8 bytes)
    - Potentially address $\approx 1.8 \text{ X } 10^{19}$ bytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-oriented memory organization

- Addresses specify byte locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Addr. Bytes 32-bit Words 64-bit Words

0000
0001
0002
0003
0004
0005
0006
0007
0008
0009
0010
0011
0012
0013
0014
0015

Addr = 0000
Addr = 0004
Addr = 0008
Addr = 0012

Addr = 0000
Addr = 0008

Data representations

- Sizes of C Objects (in Bytes)

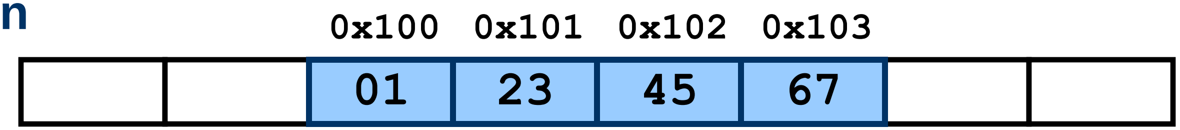
C Data type	**32 bit**	**64-bit**
char	**1**	**1**
short int	**2**	**2**
int	**4**	**4**
long int	**4**	**8**
long long int	**8**	**8**
char*	**4**	**8**
float	**4**	**4**
double	**8**	**8**

  - Portability:
    - Many programmers assume that object declared as *int* can be used to store a pointer
      - *OK for a typical 32-bit machine*
      - *Problems on a 64-bit machine*

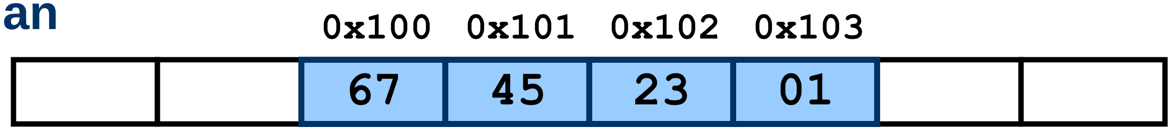
Byte ordering

- How to order bytes within multi-byte word in memory
- Conventions
  - (most) Sun’s, IBMs are “Big Endian” machines
    - Least significant byte has highest address (comes last)
  - (most) Intel’s are “Little Endian” machines
    - Least significant byte has lowest address (comes first)
- Example
  - Variable `x` has 4-byte representation `0x01234567`
  - Address given by `&x` is `0x100`

Big Endian



Little Endian



Reading byte-reversed Listings

- For most programmers, these issues are invisible
- Except with networking or disassembly
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- Example fragment

```
Address      Instruction Code          Assembly Rendition
8048365:     5b                        pop     %ebx
8048366:     81 c3 ab 12 00 00         add     $0x12ab,%ebx
804836c:     83 bb 28 00 00 00 00      cmpl    $0x0,0x28(%ebx)
```

- Deciphering Numbers
  - Value: `0x12ab`
  - Pad to 4 bytes: `0x000012ab`
  - Split into bytes: `00 00 12 ab`
  - Reverse: `ab 12 00 00`

Examining data representations

- Code to print byte representation of data
  - Casting pointer to `unsigned char *` creates byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
  int i;
  for (i = 0; i < len; i++)
    printf("0x%p\t0x%.2x\n",
           start+i, start[i]);
  printf("\n");
}
```

Printf directives:

`%p`: Print pointer

`%x`: Print Hexadecimal

`show_bytes` execution example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

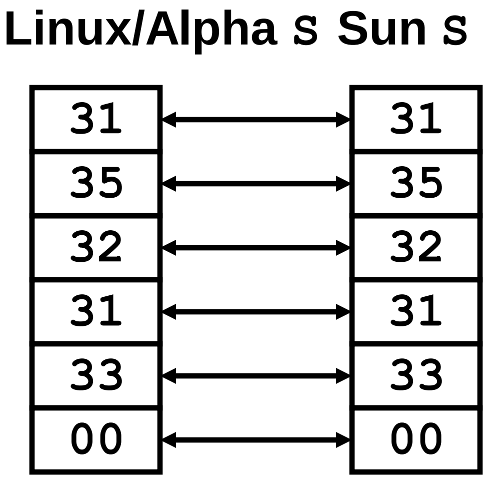
Result (Linux):

```
int a = 15213;
0x11ffffcb8   0x6d
0x11ffffcb9   0x3b
0x11ffffcba   0x00
0x11ffffcbb   0x00
```

Representing strings

- Strings in C
  - – Represented by array of characters
  - – Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Other encodings exist, but uncommon
    - Character “0” has code 0x30
      - – Digit i has code 0x30+i
  - – String should be null-terminated
    - Final character = 0
- Compatibility
  - – Byte ordering not an issue
    - Data are single byte quantities
  - – Text files generally platform independent
    - Except for different conventions of line termination character(s)!

```
char S[6] = "15213";
```



Linux/Alpha S	**Sun S**
31	31
35	35
32	32
31	31
33	33
00	00

Machine-level code representation

- Encode program as sequence of instructions
  - Each simple operation
    - Arithmetic operation
    - Read or write memory
    - Conditional branch
  - Instructions encoded as bytes
    - Alpha's, Sun's, Mac's use 4 byte instructions
      - Reduced Instruction Set Computer (RISC)
    - PC's use variable length instructions
      - Complex Instruction Set Computer (CISC)
  - Different machines → different ISA & encodings
    - Most code not binary compatible
- A fundamental concept:
  Programs are byte sequences too!

Representing instructions

```
int sum(int x, int y)
{
   return x+y;
}
```

- Sun use 2 4-byte instructions
  - Differing numbers in other cases
- PC uses 7 instructs with lengths 1, 2, and 3 bytes
  - Mostly the same for NT and for Linux
  - NT / Linux not fully binary compatible

Linux 32	**55**	**89**	**E5**	**8B**	**45**	**0C**	**03**	**45**	**08**	**C9**	**C3**
Windows	**55**	**89**	**E5**	**8B**	**45**	**0C**	**03**	**45**	**08**	**5D**	**C3**
Sun	**81**	**C3**	**E0**	**08**	**90**	**02**	**00**	**09**			

Different machines use totally different instructions and encodings

Boolean algebra

- Developed by George Boole in 19th Century
  - – Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0

Not ~A

~	
0	1
1	0

And A & B

&	0	1
0	0	0
1	0	1

Or A | B

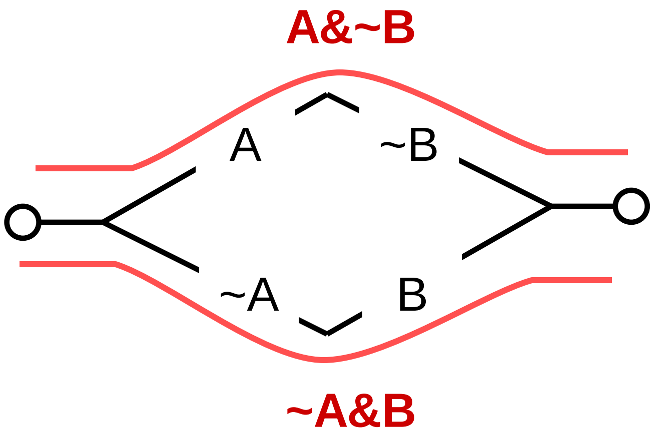
|	0	1
0	0	1
1	1	1

Xor A ^ B

^	0	1
0	0	1
1	1	0

Application of Boolean Algebra

- Applied to Digital Systems by Claude Shannon
  - 1937 MIT Master’s Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0



Connection when

A&~B | ~A&B

= A^B

Integer & Boolean algebra

- Integer Arithmetic
  - ⟨Z, +, *, –, 0, 1⟩ forms a mathematical structure called “ring”
  - Addition is “sum” operation
  - Multiplication is “product” operation
  - – is additive inverse
  - 0 is identity for sum
  - 1 is identity for product
- Boolean Algebra
  - ⟨{0,1}, |, &, ~, 0, 1⟩ forms a mathematical structure called “Boolean algebra”
  - Or is “sum” operation
  - And is “product” operation
  - ~ is “complement” operation (not additive inverse)
  - 0 is identity for sum
  - 1 is identity for product

Boolean Algebra ≈ Integer Ring

Commutative	A | B = B | A A & B = B & A	A + B = B + A A * B = B * A
Associativity	(A | B) | C = A | (B | C) (A & B) & C = A & (B & C)	(A + B) + C = A + (B + C) (A * B) * C = A * (B * C)
Product distributes over sum	A & (B | C) = (A & B) | (A & C)	A * (B + C) = A * B + B * C
Sum and product identities	A | 0 = A A & 1 = A	A + 0 = A A * 1 = A
Zero is product annihilator	A & 0 = 0	A * 0 = 0
Cancellation of negation	~ (~ A) = A	– (– A) = A

Boolean Algebra ≠ Integer Ring

Boolean: Sum distributes over product	$A \mid (B \,\&\, C) = (A \mid B) \,\&\, (A \mid C)$	$A + (B * C) \neq$ $(A + B) * (B + C)$
Boolean: Idempotency	$A \mid A = A$ $A \,\&\, A = A$	$A + A \neq A$ $A * A \neq A$
Boolean: Absorption	$A \mid (A \,\&\, B) = A$ $A \,\&\, (A \mid B) = A$	$A + (A * B) \neq A$ $A * (A + B) \neq A$
Boolean: Laws of Complements	$A \mid {\sim}A = 1$	$A + -A \neq 1$
Ring: Every element has additive inverse	$A \mid {\sim}A \neq 0$	$A + -A = 0$

Properties of & and ^

- Boolean ring
  - ⟨{0,1}, ^, &, I, 0, 1⟩
  - Identical to integers mod 2
  - I is identity operation: I (A) = A
    - A ^ A = 0
- Property: Boolean ring

– Commutative sum	A ^ B = B ^ A
– Commutative product	A & B = B & A
– Associative sum	(A ^ B) ^ C = A ^ (B ^ C)
– Associative product	(A & B) & C = A & (B & C)
– Prod. over sum	A & (B ^ C) = (A & B) ^ (B & C)
– 0 is sum identity	A ^ 0 = A
– 1 is prod. identity	A & 1 = A
– 0 is product annihilator	A & 0 = 0
– Additive inverse	A ^ A = 0

Relations between operations

- DeMorgan's Laws
  - Express & in terms of |, and vice-versa
    - A & B = ~(~A | ~B)
      - A and B are true if and only if neither A nor B is false
    - A | B = ~(~A & ~B)
      - A or B are true if and only if A and B are not both false
- Exclusive-Or using Inclusive Or
    - A ^ B = (~A & B) | (A & ~B)
      - Exactly one of A and B is true
    - A ^ B = (A | B) & ~(A & B)
      - Either A is true, or B is true, but not both

General Boolean algebras

- Boolean operations can be extended to work 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
- Now, Boolean |, & and ~ correspond to set union, intersection and complement

Representing & manipulating sets

- Useful application of bit vectors – represent finite sets
- Representation
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - 01101001 represents { 0, 3, 5, 6 }
    - 01010101 represents { 0, 2, 4, 6 }

0	1	1	0	1	0	0	1
7	6	5	4	3	2	1	0

- 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`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - ~0x41 --> 0xBE

    ~01000001_2 --> 10111110_2
  - ~0x00 --> 0xFF

    ~00000000_2 --> 11111111_2
  - 0x69 & 0x55 --> 0x41

    01101001_2 & 01010101_2 --> 01000001_2
  - 0x69 | 0x55 --> 0x7D

    01101001_2 | 01010101_2 --> 01111101_2

Logic operations in C – not quite the same

- Logical operations ||, && and ! (Logic OR, AND and Not)
  - Contrast to logical operators
    - View 0 as “False”
    - But anything nonzero as “True”
    - Always return 0 or 1
    - Early termination (if you can answer by just looking at first argument, you are done)
- Examples (char data type)
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41 → 0x01
  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01

Shift operations

- Left shift: x << y
  - Shift bit-vector x left y positions
    - Throw away extra bits on left
    - Fill with 0’s on right
- Right shift: x >> y
  - Shift bit-vector x right y positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit on right
    - Useful with two’s complement integer representation
  - For unsigned data, >> must be logical; for signed data either could be used
    - Which one? Most follow this but not all

Argument `x`	`01100010`
`<< 3`	`00010`*`000`*
Log. `>> 2`	*`00`*`011000`
Arith. `>> 2`	*`00`*`011000`

Argument `x`	`10100010`
`<< 3`	`00010`*`000`*
Log. `>> 2`	*`00`*`101000`
Arith. `>> 2`	*`11`*`101000`

Main points

- It’s all about bits & bytes
  - Numbers
  - Programs
  - Text
- Different machines follow different conventions
  - Word size
  - Byte ordering
  - Representations
- Boolean algebra is mathematical basis
  - Basic form encodes “false” as 0, “true” as 1
  - General form like bit-level operations in C
    - Good for representing & manipulating sets