Lecture 2
Bits and Bytes

Topics

- Why bits?
- Representing information as bits
  - Binary/Hexadecimal
  - Byte representations
    - » numbers
    - » characters and strings
    - » Instructions
- Bit-level manipulations
  - Boolean algebra
  - Expressing in C

Why Don't Computers Use Base 10?

Base 10 Number Representation

- That's why fingers are known as "digits"
- Natural representation for financial transactions
  - Floating point number cannot exactly represent $1.20
- Even carries through in scientific notation
  - 1.5213×10^4

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.

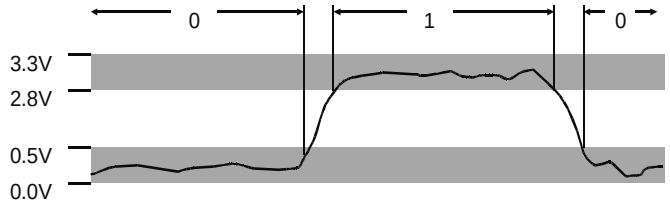
Binary Representations

Base 2 Number Representation

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

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
  - SRAM, DRAM, disk
  - Only allocate for regions actually used by program
- In Unix and Windows NT, address space private to particular "process"
  - Program being executed
  - Program can clobber 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

Encoding Byte Values

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

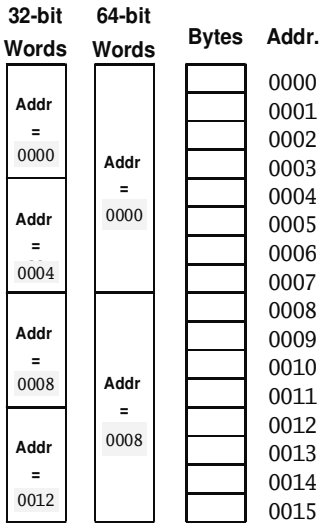
Machine Has "Word Size"

- Nominal size of integer-valued data
  - Including addresses
- Most current machines are 32 bits (4 bytes)
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems are 64 bits (8 bytes)
  - Potentially address ≈ 1.8 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)



Data Representations

Sizes of C Objects (in Bytes)

C Data Type	Compaq Alpha	Typical 32-bit	Intel IA32	Sun SPARC
int	4	4	4	4
long int	8	4	4	4
long long int				8
char	1	1	1	1
short int	2	2	2	2
float	4	4	4	4
double	8	8	8	8
long double	8	8	10/12	16
char *	8	4	4	4

- » Or any other pointer

Byte Ordering

How should bytes within multi-byte word be ordered in memory?

Conventions

- Sun's, Mac's are "Big Endian" machines
  - Least significant byte has highest address
- Alphas, PC's are "Little Endian" machines
  - Least significant byte has lowest address

Byte Ordering Example

Big Endian

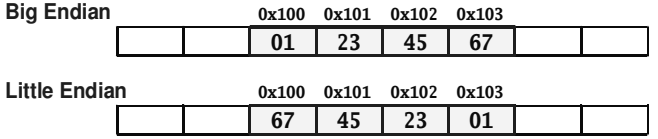
- Least significant byte has highest address

Little Endian

- Least significant byte has lowest address

Example

- Variable x has 4-byte representation 0x01234567
- Address given by &x is 0x100



Reading Byte-Reversed Listings

Disassembly

- Text representation of binary machine code
- Generated by program that reads the machine code

Example Fragment on IA32

```
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)
```

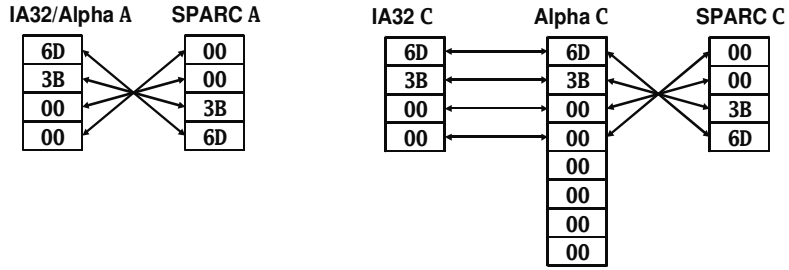
Deciphering Numbers

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

Representing Integers

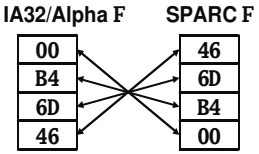
```
int A = 15213;
long int C = 15213;
```

```
Decimal: 15213
Binary:  0011 1011 0110 1101
Hex:        3    B    6    D
```



Representing Floats

```
Float F = 15213.0;
```



IEEE Single Precision Floating Point Representation

Hex:	4	6	6	D	B	4	0	0
	0100	0110	0110	1101	1011	0100	0000	0000
15213:			1110	1101	1011	01		

Not same as integer representation, but consistent across machines

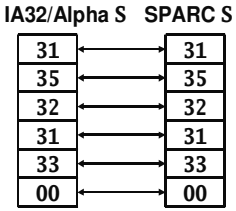
Can see some relation to integer representation, but not obvious

Representing Strings

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

Strings in C

- Represented by array of characters
- Each character encoded in Latin-1 format
  - Standard 8-bit encoding of character set
  - Other encodings exist for other languages
  - 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)!

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, SPARC's, Mac's use 4 byte instructions
    - » Reduced Instruction Set Computer (RISC)
  - PC's use variable length instructions
    - » Complex Instruction Set Computer (CISC)
- Different instruction types and encodings for different machines
  - Most code not binary compatible

Programs are Byte Sequences Too!

Representing Instructions

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

- For this example, Alpha & SPARC use two 4-byte instructions
  - Use differing numbers of instructions in other cases
- IA32 uses 7 instructions with lengths 1, 2, and 3 bytes
  - Same for NT and for Linux
  - NT / Linux not fully binary compatible

Alpha sum	SPARC sum	IA32 sum
00	81	55
00	C3	89
30	E0	E5
42	08	8B
01	90	45
80	02	0C
FA	00	03
6B	09	45
		08
		89
		EC
		5D
		C3

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
- Operations are ~ (not), & (and), | (or), ^ (exclusive or, xor), = (equal)

The operations are defined by the following truth tables:

A	~A
0	1
1	0

A	B	A&B	A|B	A^B	A=B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1

Rules for operations | and &

- *Commutativity*
  - A | B = B | A
  - A & B = B & A
- *Associativity*
  - (A | B) | C = A | (B | C)
  - (A & B) & C = A & (B & C)
- *And distributes over or*
  - A & (B | C) = (A & B) | (A & C)
- *Or distributes over and*
  - A | (B & C) = (A | B) & (A | C)
- *Identities*
  - A | 0 = A
  - A | 1 = 1
  - A & 0 = 0
  - A & 1 = A
- *Cancellation of negation*
  - ~ (~A) = A
- *Laws of complements*
  - A | ~A = 1
  - A & ~A = 0
- *Absorption*
  - A | (A & B) = A
  - A & (A | B) = A
- *Idempotency*
  - A | A = A
  - A & A = A
- *DeMorgan's Laws*
  - ~ (A & B) = ~A | ~B
  - ~ (A | B) = ~A & ~B

Properties of &, ^ (xor) and =

Property	
Commutative	A ^ B = B ^ A
Associative	(A ^ B) ^ C = A ^ (B ^ C)
Distributive	A & (B ^ C) = (A & B) ^ (A & C)
0	A ^ 0 = A (A = 0) = ~A
1	A ^ 1 = ~A (A = 1) = A
Self	A ^ A = 0 (A = A) = 1
Exclusive or using or	A ^ B = (~A & B) | (A & ~B)
	A ^ B = (A | B) & ~(A & B)
^ related to =	A ^ B = ~(A = B)
	A ^ ~B = ~A ^ B

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

Representing and Manipulating Sets

Representation

- Width *w* bit vector represents subsets of {0, …, *w*–1}
- $a_j = 1$ **iff** $j \in A$

  01101001 { 0, 3, 5, 6 } A
  76543210

  01010101 { 0, 2, 4, 6 } B
  76543210

Operations

A&B	Intersection	01000001	{ 0, 6 }
A|B	Union	01111101	{ 0, 2, 3, 4, 5, 6 }
A^B	Symmetric difference	00111100	{ 2, 3, 4, 5 }
A&~B	Difference	00101000	{ 3, 5 }
~B	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`
  $\sim 01000001_2 \rightarrow 10111110_2$
- `~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)

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, see next lecture

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