CS429: Computer Organization and Architecture

Bits and Bytes

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: September 3, 2014 at 08:40

Topics of this Slideset

There are 10 kinds of people in the world: those who understand binary, and those who don't!

- Why bits?
- Representing information as bits
  - Binary and hexadecimal
  - Byte representations : numbers, characters, strings, instructions
- Bit level manipulations
  - Boolean algebra
  - C constructs

Why Not Base 10?

Base 10 Number Representation.

- Thats 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

$$1.5213 \times 10^4$$

Implementing Electronically

- 10 different values are hard to store. ENIAC (First electronic computer) used 10 vacuum tubes / digit
- They’re hard to transmit. Need high precision to encode 10 signal levels on single wire.
- Messy to implement digital logic functions: addition, multiplication, etc.

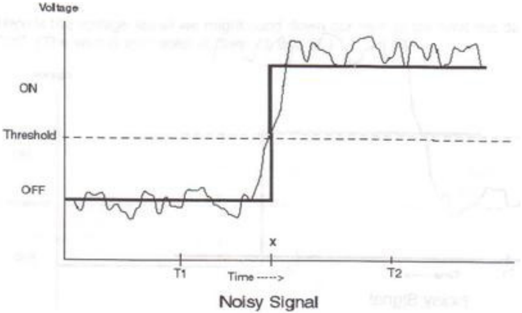
Binary Representations

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.



Representing Data

Fact: Whatever you plan to store on a computer ultimately has to be represented as a collection of bits.

That's true whether it's integers, reals, characters, strings, data structures, instructions, pictures, videos, etc.

In a sense the representation is *arbitrary*. The representation is just a *mapping from the domain onto a finite set of bit strings*.

But some representations are better than others. Why would that be? Hint: what operations do you want to support?

Representing Data

Fact: If you are going to represent any type in k bits, you can only represent 2^k different values. *There are exactly as many integers as floats on IA32.*

Fact: The same bit string can represent an integer (signed or unsigned), float, character string, list of instructions, etc. depending on the context.

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 storage for regions actually used by program.
- In Unix and Windows NT, address space private to particular "process."
  - Encapsulates the program being executed.
  - Program can clobber its own data, but not that of others.

Compiler and Run-Time System Control Allocation

- Where different program objects should be stored.
- Multiple storage mechanisms: static, stack, and heap.
- In any case, all allocation within single virtual address space.

Encoding Byte Values

Byte = 8 bits

Which can be represented in various forms:

- 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	Dec	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

Machines generally have a specific “word size.”

- It’s the nominal size of integer-valued data, including addresses.
- Most current machines run 64-bit software (8 bytes).
  - 32-bit software limits addresses to 4GB.
  - Becoming too small for memory-intensive applications.
- All x86 current hardware systems are 64 bits (8 bytes). Potentially address around $1.8X10^{19}$ bytes.
- Machines support multiple data formats.
  - Fractions or multiples of word size.
  - Always integral number of bytes.
- X86-hardware systems operate in 16, 32, and 64 bits modes.
  - Initially starts in 286 mode, which is 16-bit.
  - Under programmer control, 32- and 64-bit modes are enabled.

Word-Oriented Memory Organization

Addresses Specify Byte Locations

- Which is the address of the first byte in word.
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit).

32-bit words	64-bit words	bytes	addr.
Addr: 0000	Addr: 0000		0000
			0001
			0002
			0003
Addr: 0004			0004
			0005
			0006
			0007
Addr: 0008	Addr: 0008		0008
			0009
			0010
			0011
Addr: 0012			0012
			0013
			0014
			0015

Data Representations

Sizes of C Objects (in Bytes)

C Data Type	Alpha	Intel IA32	AMD 64
int	4	4	4
long int	8	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	8	10/12
char *	8	4	8
other pointer	8	4	8

Byte Ordering

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

Conventions

- Sun, PowerPC MacIntosh computers are “big endian” machines: least significant byte has highest address.
- Alpha, Intel MacIntosh, PC’s are “little endian” machines: least significant byte has lowest address.
- ARM processor offer support for big endian, but mainly they are used in their default, little endian configuration.
- There are many (hundreds) of microcontrollers so check before you start programming!

Byte Ordering Examples

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`

Big Endian:

Address:			0x100	0x101	0x102	0x103		
Value:			01	23	45	67		

Little Endian:

Address:			0x100	0x101	0x102	0x103		
Value:			67	45	23	01		

Reading Byte-Reversed Listings

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: Consider the value `0x12ab` in the second line of code:

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

Examining Data Representations

Code to Print Byte Representations of Data

Casting a pointer to unsigned `char *` creates a 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):

```
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

Representing Integers

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

$$15213_{10} = 0011101101101101_2 = 3B6D_{16}$$

	Linux	**Alpha**	**Sun**
A	6D 3B 00 00	6D 3B 00 00	00 00 3B 6D
B	93 C4 FF FF	93 C4 FF FF	FF FF C4 93
C	6D 3B 00 00	6D 3B 00 00 00 00 00 00	00 00 3B 6D

We'll cover the representation of negatives shortly.

Representing Pointers

```
int B = −15213;
int ∗P = &B;
```

Linux Address:

Hex: BFFFF8D4

Binary: 10111111111111111111100011010100

In memory: D4 F8 FF BF

Sun Address:

Hex: EFFFFFB2C

Binary: 11101111111111111111101100101100

In Memory: EF FF FB 2C

Alpha Address:

Hex: 1FFFFFCA0

Binary: 000111111111111111111111110010100000

In Memory: A0 FC FF FF 01 00 00 00

Different compilers and machines assign different locations.

Representing Floats

All modern machines implement the IEEE Floating Point standard. This means that it is consistent across all machines.

```
float F = 15213.0;
```

Hex: 466DB400
Binary: 01000110011011011011010000000000
In Memory (Linux/Alpha): 00 B4 6D 46
In Memory (Sun): 46 6D B4 00

Note that it's not the same as the `int` representation, but you can see that the int is in there, if you know where to look.

Representing Strings

Strings in C

- Strings are represented by an array of characters.
- Each character is encoded in ASCII format.
    - Standard 7-bit encoding of character set.
    - Other encodings exist, but are less common.
    - Character `0` has code `0x30`. Digit `i` has code `0x30+i`.
- Strings should be null-terminated. That is, the final character has ASCII code `0`.

Compatibility

- Byte ordering not an issue since the data are single byte quantities.
- Text files are 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 are encoded as sequences of bytes.
  - Alpha, Sun, PowerPC Mac use 4 byte instructions (Reduced Instruction Set Computer" (RISC)).
  - PC's and Intel Mac's use variable length instructions (Complex Instruction Set Computer (CISC)).
- Different instruction types and encodings for different machines.
- Most code is not binary compatible.

Remember: Programs are byte sequences too!

Representing Instructions

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

For this example, Alpha and Sun use two 4-byte instructions. They use differing numbers of instructions in other cases.

PC uses 7 instructions with lengths 1, 2, and 3 bytes. Windows and Linux are not fully compatible.

Different machines typically use different instuctions and encodings.

Instruction sequence for sum program:

Alpha: 00 00 30 42 01 80 FA 68
Sun: 81 C3 E0 08 90 02 00 09
PC: 55 89 E5 8B 45 OC 03 45 08 89 EC 5D C3

Boolean Algebra

Developed by George Boole in the 19th century, Boolean algebra is the algebraic representation of logic. We encode “True” as 1 and “False” as 0.

And: `A & B` = 1 when both A = 1 and B = 1.

0	1	&
0	0	0
0	1	0
1	0	0
1	1	1

Or: `A | B` = 1 when either A = 1 or B = 1.

0	1	|
0	0	0
0	1	1
1	0	1
1	1	1

Not: `~A` = 1 when A = 0.

0	~
0	1
1	0

Xor: `A ^ B` = 1 when either A = 1 or B = 1, but not both.

0	1	^
0	0	0
0	1	1
1	0	1
1	1	0

Application of Boolean Algebra

In a 1937 MIT Master's Thesis, Claude Shannon showed that Boolean algebra would be a great way to model digital networks.

At that time, the networks were relay switches. But today, all combinational circuits can be described in terms of Boolean "gates."

Integer Algebra

Mathematical Rings

- A *ring* is an algebraic structure.
- It includes a finite set of elements and some operators with certain properties.
- A ring has a finite number of elements, a *sum* operation, a *product* operation, additive inverses, and identity elements.
- The addition and product ops must be associative and commutative.
- The product operation should distribute over addition.

Integer Arithmetic

- $\langle Z, +, *, , 0, 1\rangle$ forms a ring.
- Addition is the sum operation.
- Multiplication is the product operation.
- Minus returns the additive inverse
- 0 is the identity for sum.
- 1 is identity for product.

Boolean Algebra

- $\langle \{0,1\}, |, \&, \sim, 0, 1 \rangle$ forms a *Boolean algebra*.
- Or is the sum operation.
- And is the product operation.
- $\sim$ is the "complement" operation (not additive inverse).
- 0 is the identity for sum.
- 1 is the identity for product.

Note that a Boolean algebra is not the same as a ring, though every Boolean algebra gives rise to a ring if you let ˆ be the product operator.

Boolean Algebra like Integer Ring

Commutativity:

$A|B = B|A$ $\qquad$ $A + B = B + A$

$A \,\&\, B = B \,\&\, A$ $\qquad$ $A * B = B * A$

Associativity:

$(A|B)|C = A|(B|C)$ $\qquad$ $(A + B) + C = A + (B + C)$

$(A \,\&\, B)|C = A \,\&\, (B \,\&\, C)$ $\qquad$ $(A * B) * C = A * (B * C)$

Product Distributes over Sum:

$A \,\&\, (B|C) = (A \,\&\, B)|(A \,\&\, C) A * (B + C) = (A * B) + (A * C)$

Sum and Product Identities:

$A|0 = A$ $\qquad$ $A + 0 = A$

$A \,\&\, 1 = A$ $\qquad$ $A * 1 = A$

Zero is product annihilator:

$A \,\&\, 0 = 0$ $\qquad$ $A * 0 = 0$

Cancellation of negation:

$\sim (\sim A)) = A$ $\qquad$ $-(-A)) = A$

Boolean Algebra vs. Integer Ring

Boolean: Sum distributes over product

$A|(B \mathbin{\&} C) = (A|B) \mathbin{\&} (A|C) \qquad A + (B * C) \neq (A + B) * (A + C)$

Boolean: Idempotency

$A|A = A \qquad A + A \neq A$

$A \mathbin{\&} A = A \qquad A * A \neq A$

Boolean: Absorption

$A|(A \mathbin{\&} B) = A \qquad A + (A * B) \neq A$

$A \mathbin{\&} (A|B) = A \qquad A * (A + B) \neq A$

Boolean: Laws of Complements

$A| \sim A = 1 \qquad A + A \neq 1$

Ring: Every element has additive inverse

$A|\ A \neq 0 \qquad A + A = 0$

Properties of & and ˆ

- $\langle \{0, 1\}, \hat{}\,, 0, 1 \rangle$ forms a *Boolean ring*.
- This is isomorphic to the integers mod 2.
- I is the identity operation: $I(A) = A$.

Commutative sum:	$A\,\hat{}\,B = B\,\hat{}\,A$
Commutative product:	$A \mathbin{\&} B = B \mathbin{\&} A$
Associative sum:	$(A\,\hat{}\,B)\,\hat{}\,C = A\,\hat{}\,(B\,\hat{}\,C)$
Associative product:	$(A \mathbin{\&} B) \mathbin{\&} C = A \mathbin{\&} (B \mathbin{\&} C)$
Prod. over sum:	$A \mathbin{\&} (B\,\hat{}\,C) = (A \mathbin{\&} B)\,\hat{}\,(A \mathbin{\&} C)$
0 is sum identity:	$A\,\hat{}\,0 = A$
1 is prod. identity:	$A \mathbin{\&} 1 = A$
0 is product annihilator:	$A \mathbin{\&} 0 = 0$
Additive inverse:	$A\,\hat{}\,A = 0$

Relations Between Operations

DeMorgan's Laws

Express & in terms of |, and vice-versa:

$$A \mathbin{\&} B = \sim(\sim A | \sim B)$$

$$A|B = \sim(\sim A \mathbin{\&} \sim B)$$

Exclusive-Or using Inclusive Or:

$$A\hat{\ }B = (\sim A \mathbin{\&} B)|(A \mathbin{\&} \sim B)$$

$$A\hat{\ }B = (A|B) \mathbin{\&} \sim(A \mathbin{\&} B)$$

General Boolean Algebras

We can also operate on bit vectors (bitwise). All of the properties of Boolean algebra apply:

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101     ~ 01010101

----------   ----------   ----------     ----------
  01000001     01111101     00111100       10101010
```

Representing Sets

Representation

A width w bit vector may represents subsets of $\{0, \dots, w1\}$.
$a_i = 1$ iff $j \in A$

Bit vector A:

01101001 represents $\{0, 3, 5, 6\}$
76543210

Bit vector B:

01010101 represents $\{0, 2, 4, 6\}$
76543210

What bit operations on these set representations correspond to: intersection, union, complement?

Representing Sets

Operations:

Given the two sets above, perform these bitwise ops to obtain:

Set operation	Boolean op	Result	Set
Intersection	`A & B`	01000001	$\{0, 6\}$
Union	`A | B`	01111101	$\{0, 2, 3, 4, 5, 6\}$
Symmetric difference	`A ^ B`	00111100	$\{2, 3, 4, 5\}$
Complement	`~A`	10010110	$\{1, 2, 4, 7\}$

Bit Level Operations in C

The operations &, $|$, $\sim$, ˆ are all available in C.

- Apply to any *integral* data type: long, int, short, char.
- View the arguments as bit vectors.
- Operations are applied bit-wise to the argument(s).

Examples: (char data type)

$\sim 0x41 \rightarrow 0xBE$

$\sim 01000001_2 \rightarrow 10111110_2$

$\sim 0x00 \rightarrow 0xFF$

$\sim 00000000_2 \rightarrow 11111111_2$

$0x69 \ \& \ 0x55 \rightarrow 0x41$

$01101001_2 \ \& \ 01010101_2 \rightarrow 01000001_2$

$0x69 | 0x55 \rightarrow 0x7D$

$01101001_2 | 01010101_2 \rightarrow 01111101_2$

Contrast to Logical Operators in C

Remember the operators: `&&`, `||`, `!`.

- View 0 as "False."
- View anything nonzero as "True."
- Always return 0 or 1.
- Allow for early termination.

Examples:

`!0x41` → `0x00`

`!0x00` → `0x01`

`!!0x41` → `0x01`

`!!0x69 && 0x55` → `0x01`

`!!0x69 || 0x55` → `0x01`

Can use `p && *p` to avoids null pointer access. How and why?

Shift Operations

Left Shift: `x << y`
Shift bit vector x left by y positions

- Throw away extra bits on the left.
- Fill with 0's on the right.

Right Shift: `x >> y`
Shift bit vector x right by y positions.

- Throw away extra bits on the right.
- **Logical shift:** Fill with 0's on the left.
- **Arithmetic shift:** Replicate with most significant bit on the left.

Arithmetic shift is useful with two's complement integer representation.

Shift Operations

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Cool Stuff with XOR

Bitwise XOR is a form of addition, with the extra property that each value is its own additive inverse: `A ^ A = 0`.

```
void funny(int *x, int *y)
{
    *x = *x ^ *y; /* #1 */
    *y = *x ^ *y; /* #2 */
    *x = *x ^ *y; /* #3 */
}
```

	*x	*y
Begin	A	B
1	`A ^ B`	`B`
2	`A ^ B`	`(A ^ B) ^ B = A`
3	`(A ^ B) ^ A = B`	`B`
End	`A`	`B`

Is there ever a case where this code fails?

Main Points

It's all about bits and bytes.

- Numbers
- Programs
- Text

Different machines follow different conventions.

- Word size
- Byte ordering
- Representations

Boolean algebra is the mathematical basis.

- Basic form encodes "False" as 0 and "True" as 1.
- General form is like bit-level operations in C; good for representing and manipulating sets.