CMPSC 311 - Introduction to Systems Programming

Bit/Byte Operations

Professor Abutalib Aghayev
(Slides are mostly by Professor Patrick McDaniel)

Number Systems

- All base-X systems have the following characteristic:

Assume a base b and digits $P = \{p_k, p_{k-1}, p_{k-2}, \ldots, p_1, p_0\}$

$$value = \sum_{i=0}^{k} b^i * p_i$$

where $\forall p_i \in P, p_i = [0, b-1]$

Base-10 (decimal)
- Digits: 0,1,2,3,4,5,6,7,8,9
- Place values: 10^0 , 10^1, 10^2, …
- Example: 123 = 3x1+2x10+1x100
  = 123

Base-2(binary)
- Digits: 0,1
- Place values: 2^0 , 2^1, 2^2, …
- Example: 1011 = 1x1+1x2+0x4+1x8
  = 11

Base-16 (hexadecimal)
- Digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Place values: 16^0 , 16^1, 16^2, …
- Example: 0xAFC = Cx1+Fx16+Ax256
  = 2812

Converting Decimal to Binary

PennState

- while n != 0:
  - next binary digit (from right to left) = n % 2
  - n = n / 2

n	remainder	digit #
235	1	0
117	1	1
58	0	2
29	1	3
14	0	4
7	1	5
3	1	6
1	1	7

235 (base-10) = 11101011 (base-2)

Converting Decimal and Binary to Hex

- Converting decimal to hexadecimal
  - while n != 0:
    - next hex digit (from right to left) = n % 16
    - n = n / 16

n	remainder	digit #
235	B (11)	0
14	E (14)	1

235 (base-10) = EB (base-16)

- Converting binary to hexadecimal
  - convert decimal to binary
  - group binary digits to into groups of 4 bits (nibbles) starting from right
  - Convert each nibble to hexadecimal digit

235 (base-10) = 11101011 (base-2) = 1110 1011 = EB

- Be familiar with hex notation: 0xDEADBEEF has 8 nibbles, 4 bytes (or octets)
- Hexadecimal to binary: just convert each nibble to binary

Conversion Summary

- Binary to decimal
  - sum of binary digits times powers of 2
- Hexadecimal to decimal
  - sum of hex digits times power of 16
- Decimal to binary
  - division method
- Decimal to hex
  - division method
  - or, convert to binary and use binary to hex method below
- Binary to hex
  - group binary digits to nibbles, convert nibbles to hex digits
- Hex to binary
  - convert each hex digit to binary

Byte Ordering Example

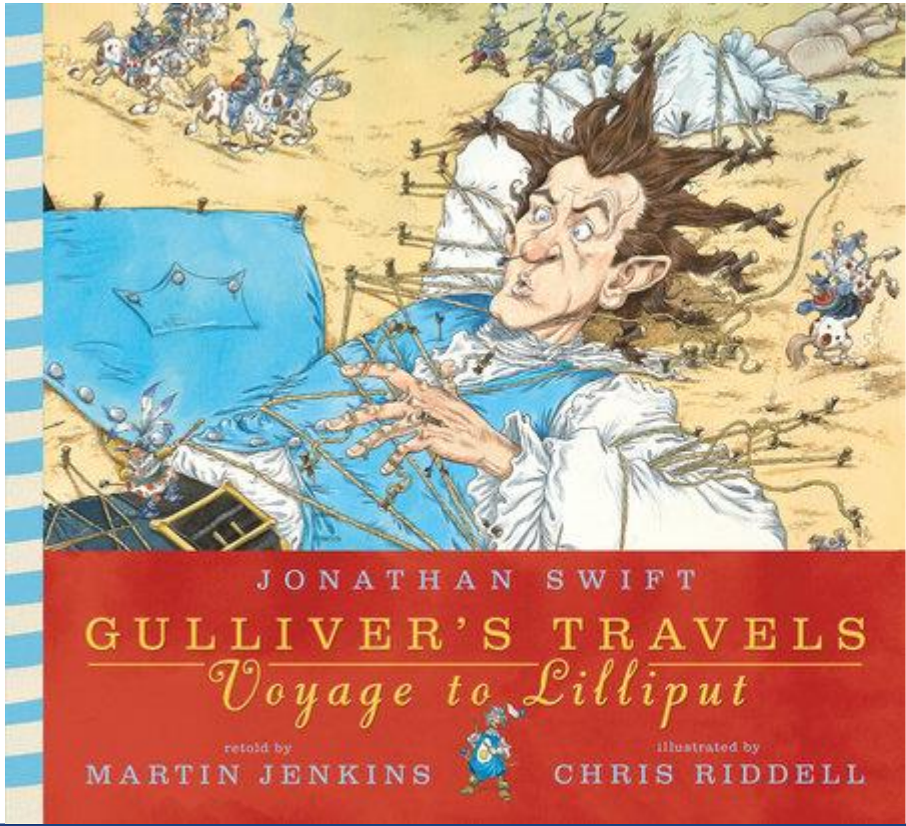
- How should bytes within a multi-byte word be ordered in memory?
  - Big Endian: Least significant byte has *highest* address (SPARC)
  - Little Endian: Least significant byte has *lowest* address (x86)
- Example: int x = 305419896
  - Variable x has 4-byte representation 0x12345678
  - Address given by &x is 0x100

Big Endian

		0x100	0x101	0x102	0x103		
		12	34	56	78		

Little Endian

		0x100	0x101	0x102	0x103		
		78	56	34	12		



Examining Data Representations

PennState

- Code to find endianness of the architecture
  - Casting pointer to uint8_t * creates byte array

```
#include <stdio.h>
#include <stdint.h>

void show_bytes(uint8_t *start, int len) {
  for (int i = 0; i < len; ++i)
    printf("%p\t0x%.2x\n", start+i, start[i]);
  printf("\n");
}

int main(void) {
  int a = 305419896;
  printf("a lives at address %p\n\n", &a);
  show_bytes((uint8_t *)&a, sizeof(int));
}
```

Result (Linux on x86):

```
a lives at address 0x7ffebf803174

0x7ffebf803174    0x78
0x7ffebf803175    0x56
0x7ffebf803176    0x34
0x7ffebf803177    0x12
```

printf directives

%p	Print pointer
%x	Print hexadecimal

Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic that based on “True” (as 1) and “False” (as 0)

And

A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

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`
    - `~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`$_2$

Contrast: Logic Operations in C

- Contrast to Logical Operators (`&&`, `||`, `!`)
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - Early termination

Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of {0, …, w–1} for set "A"
  - $a_j = 1$ if $j \in A$

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

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

```
Operations On Sets:

& 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 }
```

Representing Signed Numbers

b3	b2	b1	b0		Unsigned	One's complement	Two's complement
0	0	0	0		0	0	0
0	0	0	1		1	1	1
0	0	1	0		2	2	2
0	0	1	1		3	3	3
0	1	0	0		4	4	4
0	1	0	1		5	5	5
0	1	1	0		6	6	6
0	1	1	1		7	7	$7 = 2^{n-1} - 1$
1	0	0	0		8	-7	$-8 = -2^{n-1}$
1	0	0	1		9	-6	-7
1	0	1	0		10	-5	-6
1	0	1	1		11	-4	-5
1	1	0	0		12	-3	-4
1	1	0	1		13	-2	-3
1	1	1	0		14	-1	-2
1	1	1	1		15	-0	-1

- Computing one's complement negative representation:
  - Complement the positive number
- Computing two's complement negative representation:
  - Complement the positive number and add 1
- Most architectures use two's complement
- Given n-bit signed integer:
  - Positive range: $0 – 2^{n-1}-1$
  - Negative range: -2^{n-1}

type	bytes (32-bit)	bytes (64-bit)	32-bit range	printf
char	1	1	[0, 255]	%c
short int	2	2	[-32768,32767]	%hd
unsigned short int	2	2	[0, 65535]	%hu
int	4	4	[-214748648, 2147483647]	%d
unsigned int	4	4	[0, 4294967295]	%u
long int	4	8	[-2147483648,	%ld

Beware of integer overflows

- Spot the bug in the following:

```
for (uint32_t n = 10; n >=0; --n) {
  printf("do I even terminate?\n");
}
```

- What does the following print?

```
int x = 0xffffffff;
printf("%d\n", x);
```

Google AI Blog

The latest news from Google AI

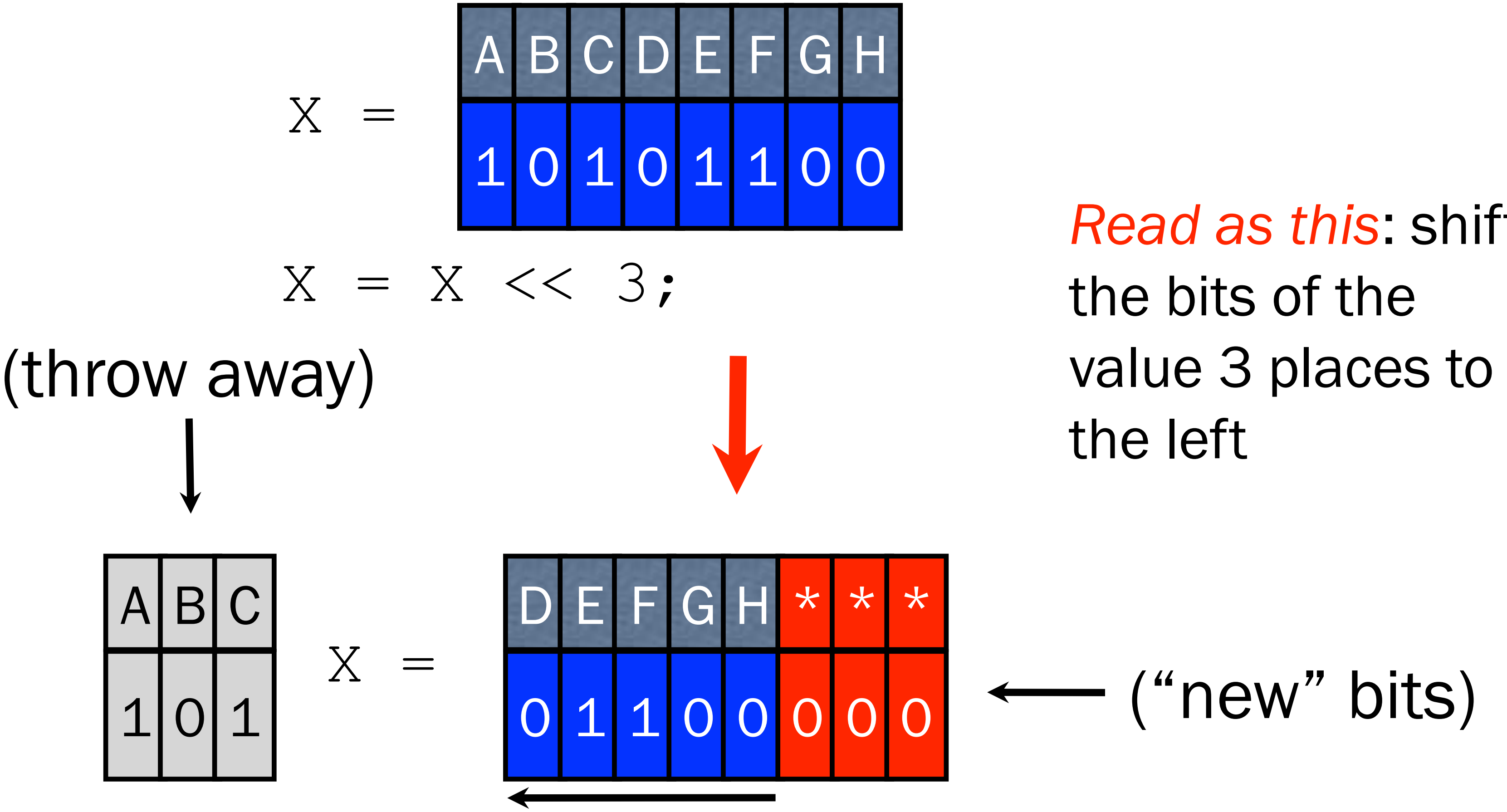
Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken

Friday, June 2, 2006

Posted by Joshua Bloch, Software Engineer

Shift Operations

- A shift operator (<< or >>) moves bits to the right or left, throwing away bits and adding bits as necessary



Read as this: shift the bits of the value 3 places to the left

Putting it all together

- Suppose you want to place multiple values in the same 32-bit integer
  - Value a in least significant byte
  - Value b in 2nd byte
  - Value c in 3rd byte
  - Value d in 4th byte

Bits	31-24	23-16	8-15	0-7
Values	d	c	b	a

Using bit operations …

```c
uint32_t pack_bytes(uint32_t a, uint32_t b, uint32_t c, uint32_t d) {

        // Setup some local values
        uint32_t retval = 0x0, tempa, tempb, tempc, tempd;

        tempa = a&0xff; // Make sure you are only getting the bottom 8 bits
        tempb = (b&0xff) << 8; // Shift value to the second byte
        tempc = (c&0xff) << 16; // Shift value to the third byte
        tempd = (d&0xff) << 24; // Shift value to the top byte
        retval = tempa|tempb|tempc|tempd; // Now combine all of the values

        // Print out all of the values
        printf("A: 0x%08x\n", tempa);
        printf("B: 0x%08x\n", tempb);
        printf("C: 0x%08x\n", tempc);
        printf("D: 0x%08x\n", tempd);

        // Return the computed value
        return retval;
}

...

printf("Packed bytes : 0x%08x\n", pack_bytes(0x111, 0x222, 0x333, 0x444));
```

```
A: 0x00000011
B: 0x00002200
C: 0x00330000
D: 0x44000000
Packed bytes : 0x44332211
```