Representing Information

Outline

- Bit and Byte
- Understand machine representations of numbers
- Suggested reading
  - 1.1, 2.1.1

Why Bit?

- Modern computers store and process
  - Information represented as two-valued signals
  - These lowly binary digits are ***bits***
- Bits form the basis of the digital revolution

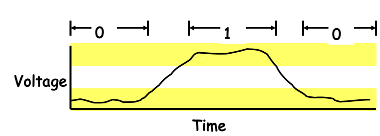
The Decimal Representation

- Base-10
- Has been in use for over 1000 years
- Developed in India
- Improved by Arab mathematicians in the 12th century
- Brought to the West in the 13th century by
  - the Italian mathematician Leonardo Pisano,
    - better known as Fibonacci.

Why Bit?

- Using decimal notation is natural for ten-fingered humans
- But binary values work better when building machines
  - that store and process information

Why Bit?



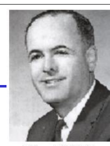
Why Bit?

- The electronic circuitry is very simple and reliable for
  - storing and performing computations on two-valued signals
- This enabling manufacturers to integrate
  - millions of such circuits on a single silicon chip

Group Bits

- In isolation, a single bit is not very useful
- In English, there are 26(or 52) characters in its alphabet. They are not useful either in isolation
- However, there are plenty of words in its vocabulary. How is this achieved?
- Similarly, we are able to represent the elements of any finite set by using bits (instead of bit)

Group Bits


维纳・布赫霍尔兹

- To do this, we
  - first group bits together
  - then apply some *interpretation* to the different possible bit patterns
    - that gives meaning to each patterns
- 8-bit chunks are organized as a byte
  - Dr. Werner Buchholz in July 1956
  - during the early design phase for the IBM Stretch computer

Value of Bits

```
Bits          01010
Value         0*2^4+1*2^3+0*2^2+1*2^1+0*2^0 = 10

Value              102(1100110)
Bits          102 = 51*2 + 0 (0)
              51  = 25*2 + 1 (1)
              25  = 12*2 + 1 (1)
              12  =  6*2 + 0 (0)
               6  =  3*2 + 0 (0)
               3  =  1*2 + 1 (1)
               1  =  0*2 + 1 (1)
```


Group bits as numbers — Three encodings

- Unsigned encoding
  - Representing numbers greater than or equal to 0
  - Using traditional binary representation
- Two's-complement encoding
  - Most common way to represent either positive or negative numbers
- Floating point encoding
  - Base-two version of scientific notation for representing real numbers

Group bits as numbers — Understanding numbers

- Machine representation of numbers are not same as
  - Integers and real numbers
- They are finite approximations to integers and real numbers
  - Sometimes, they can behave in unexpected way

'int' is not integer

- Overflow
  - 200*300*400*500 = -884,901,888
  - Product of a set of positive numbers yielded a negative result
- Commutativity & Associativity remain
  - (500 * 400) * (300 * 200)
  - ((500 * 400) * 300) * 200
  - ((200 * 500) * 300) * 400
  - 400 * (200 * (300 * 500))

'float' is not real number

- Product of a set of positive numbers is positive
- Overflow and Underflow
- Associativity does not hold
  - (3.14+1e20)-1e20 = 0.0
  - 3.14+(1e20-1e20) = 3.14

Hexadecimal

- Base 16 number representation
- Use characters '0' to '9' and 'A' to 'F'
- Write $FA1D37B_{16}$ in C as
  - `0xFA1D37B` or
  - `0xfa1d37b`

Hexadecimal

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}

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

Hexadecimal vs. Binary

```
0x173A4C
Hexadecimal    1    7    3    A    4    C
Binary       0001 0111 0011 1010 0100 1100

1111001010110110110011
Binary          11 1100 1010 1101 1011 0011
Hexadecimal      3    C    A    D    B    3
0x3CADB3
```


Hexadecimal vs. Decimal

```
Hexadecimal    0xA7
Decimal        10*16+7 = 167

Decimal        314156 = 19634*16 + 12 (C)
                19634 =  1227*16 +  2 (2)
                 1227 =    76*16 + 11 (B)
                   76 =     4*16 + 12 (C)
                    4 =     0*16 +  4 (4)
Hexadecimal    0x4CB2C
```

Hexadecimal vs. Binary

- 1100100101111011 -> *C97B*
- 100110111001111010101 -> **2 6 E 7 B 5**

Decimal, Hexadecimal, Binary

Decimal	Binary	Hexadecimal
62	00111110	0x3E
3*16+7=55	0011 0111	0x37
8*16+8=136	01010010	0x52

Hexadecimal

- 0x503c + 0x8 = **0x5044**
- 0x503c – 0x40 = **0x4ffc**
- 0x503c + **64** = **0x507c**
- 0x50ea -0x503c = **0xae**

C Programming Language (1)

Outline

- *Sample codes*
- *Introduction*
- *Compiler drivers*
- *Assembly code and object code*
- *Suggested reading*
  - 1.2

"Hello world" example

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

Header file

Standard Library

File Inclusion and Macro Substitution

a.c	int j ;

#include "b.h"
 	
main()
{
    return i+j ;
}
	main()
	{
	 return 100 + j ;
	}

b.h #define i 100 int j ;	Macro Substitution

25

File Inclusion and Macro Substitution

- #include "filename"
- #include <filename>

- #define forever for(;;) /* infinite loop */
- #define square(x) (x)*(x)

26

Formatted Output - Printf

- int printf(char* format, arg1, arg2, …)
  int a=1, b=2;
  printf("a=%d, b=%d\n), a, b);
- %d, %i decimal number
- %o octal number(without a leading zero)
- %x, %X hexadecimal number
- %c single charater

27

The Hello Program

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

28

The Hello Program

- Source program
  - Created by editor and saved as a text file (Consists exclusively of ASCII characters)
  - Binary file
    - Not text file
- Begins life as a high-level C program
  - Can be read and understand by human beings
- The individual C statements must be translated by *compiler drivers*
  - So that the hello program can run on a computer system

29

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Control Characters

07	\a	Bell
08	\b	Backspace
09	\t	Horizontal Tab
0A	\n	Line feed
0B	\v	Vertical Tab
0C	\f	Form feed
0D	\r	Carriage return

The Hello Program

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

The Hello Program

- The *C* programs are translated into
  - A sequence of low-level *machine-language* instructions
- These instructions are then packaged in a form
  - called an *object program*
- *Object program* are stored as a binary disk file
  - Also referred to as *executable object files*

The Context of a Compiler (gcc)

hello.c | Source program (text)

Preprocessor (cpp)

hello.i | Modified source program (text)

Compiler (cc1)

hello.s | Assembly program (text)

Assembler (as)

hello.o | Relocatable object program (binary)

Linker (ld)

hello | Executable object program (binary)

Preprocessor

Macro Substitution

a.c
```
#include "b.h"

main()
{
    return i+j ;
}
```

Obtain with command

`gcc -E a.c`

Source file `a.i`

b.h
```
#define i 100
int j ;
```

```
# 1 "a.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "a.c"
# 1 "b.h" 1

int j;
# 2 "a.c" 2

main()
{
 return 100 +j ;
}
```


Source Code and Assembly Code

```
//C code
long mult2(long, long);
void multstore(long x, long y, long *dest)
{
  long t = mult2(x, y);
  *dest = t;
}
```

Obtain with command

`gcc -Og -S mstore.c`

```
Assembly file mstore.s
multstore:
      pushq  %rbx
      movq   %rdx,%rbx
      call   mult2
      movq   %rax,  (%rbx)
      popq   %rbx
      ret
```


Relocatable Object Code

```
53 48 89 d3 e8 00
00 00 00 48 89 03
5b c3
```

Obtain with command

`gcc -Og -c mstore.c`

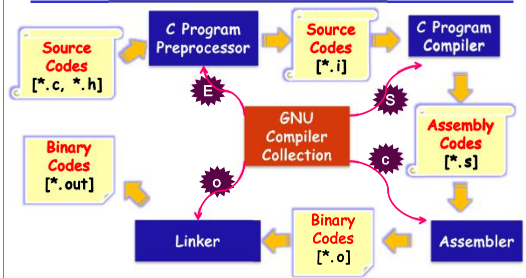
Relocatable object file `mstore.o`

Executable Object Code

`#include <stdio.h>`
`void mulstore(long, long, long*)`
`int main() {`
`    long d;`
`    multstore(2, 3, &d);`
`    printf( "2 * 3 --> %d\n", d);`
`    return 0;`
`}`
 	
`long mult2(long a, long b) {`
`    long s = a * b ;`
`    return s;`
`}`
	53 48 89 d3 e8 42 00
	00 00 48 89 03 5b c3
	
	Obtain with command
	`gcc -Og -o prog main.c mstore.c`

37

COMPILING



38

Manipulating Information

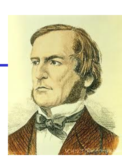
39

Outline

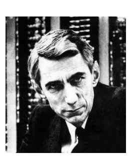
- Bit-level operations
- Suggested reading
  - 2.1.6~2.1.9

40

Boolean Algebra



- Developed by George Boole(1815-1864)
  - Algebraic representation of logic
    - Encode "True" as 1
    - Encode "False" as 0
- Claude Shannon(1916–2001)founded the information theory
  - made the connection between Boolean algebra and digital logic
- Plays a central role in the design and analysis of digital systems



41

Boolean Algebra

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

42

General Boolean Algebras

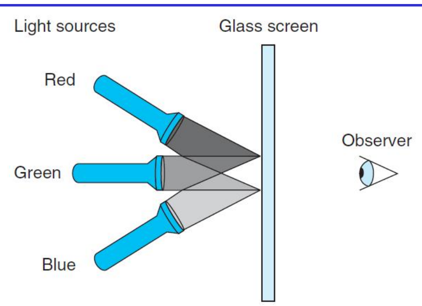
- Operate on Bit Vectors
  - Operations applied bitwise

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

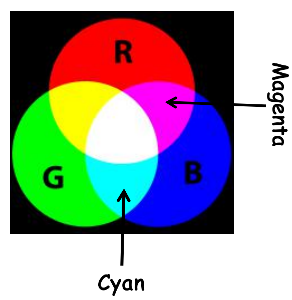

General Boolean Algebras

- Representation of Sets
  - Width w bit vector represents subsets of $\{0, \ldots, w-1\}$
  - $a_j = 1$ if $j \in A$
    - 01101001 { 0, 3, 5, 6 }
    - 01010101 { 0, 2, 4, 6 }
  - & 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 }

RGB Color Model



RGB Color Model



R	*G*	*B*	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

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

Bit-Level Operations in C

`~0x41`	`0xBE`
`~0100 0001`	`1011 1110`
`~0x00`	`0xFF`
`~0000 0000`	`1111 1111`
`0x69 & 0x55`	`0X41`
`0110 1001 & 0101 0101`	`0100 0001`
`0x69 | 0x55`	`0x7D`
`0110 1001 | 0101 0101`	`0111 1101`

Cool Stuff with Xor

- Bitwise Xor is form of addition
- With extra property that every value is its own additive inverse
  - A ^ A = 0

Cool Stuff with Xor

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

Step	*x	*y
Begin	A	B
1	A^B	B
2	A^B	(A^B)^B = A^(B^B) = A^0 = A
3	(A^B)^A = (B^A)^A = B^(A^A) = B^0 = B	A
End	B	A

Cool Stuff with Xor

```
void reverse_array(int a[], int cnt) {
     int first, last;
     for (first = 0, last = cnt-1;
           first <= last;
           first++,last--)
        inplace_swap(&a[first], &a[last]);
}
```


Mask Operations

- Bit pattern
  - 0xFF
    - Having 1s for the least significant eight bits
    - Indicates the lower-order byte of a word
- Mask Operation
  - X = 0x89ABCDEF
  - X & 0xFF =?
- Bit Pattern ~0
  - Why not 0xFFFFFFFF?

Mask Operations

- Write C expressions that work for any word size $w \geq 8$
- For $x = 0x87654321$, with $w = 32$
- The least significant byte of x, with all other bits set to 0
  - [0x00000021]   x & 0xFF

Mask Operations

- All but the least significant byte of complemented, with the least significant byte left unchanged
  - [0x789ABC21]   x ^ ~0xFF
- The least significant byte set to all 1s, and all other bytes of x left unchanged.
  - [0x876543FF].   x | ~0xFF

Logical Operations in C

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

Logical Operations in C

- Examples (char data type)
  - !0x41 --> 0x00
  - !0x00 --> 0x01
  - !!0x41 --> 0x01
  - 0x69 && 0x55 --> 0x01
  - 0x69 || 0x55 --> 0x01

Short Cut in Logical Operations

- a && 5/a
  - If a is zero, the evaluation of 5/a is stopped
  - avoid division by zero
- p && *p
  - Never cause the dereferencing of a null pointer
- Using only bit-level and logical operations
  - Implement x == y
  - it returns 1 when x and y are equal, and 0 otherwise
  - !(x^y)

Shift Operations in C

- Left Shift: x << y
  - Shift bit-vector x left y positions
    - Throw away extra bits on left
    - Fill with 0's on right

Argument x	01100010
<< 3	00010*000*

Argument x	10100010
<< 3	00010*000*

Shift Operations in C

- 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 (especially for the negative number)

Argument x	01100010
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Shift Operations in C

- What happens ?
  - int lval = 0xFEDCBA98 << 32;
  - int aval = 0xFEDCBA98 >> 36;
  - unsigned uval = 0xFEDCBA98u >> 40;
- It may be
  - lval 0xFEDCBA98 (0)
  - aval 0xFFEDCBA9 (4)
  - uval 0x00FEDCBA (8)
- Be careful about
  - 1<<2 + 3<<4 means 1<<(2 + 3)<<4

bitCount

- Returns number of 1's a in word
- Examples: bitCount(5) = 2, bitCount(7) = 3
- Legal ops: ! ~ & ^ | + << >>
- Max ops: 40

61

Sum 8 groups of 4 bits each

```
int bitCount(int x) {
  int m1 = 0x11 | (0x11 << 8);
  int mask = m1 | (m1 << 16);
  int s = x & mask;
  s += x>>1 & mask;
  s += x>>2 & mask;
  s += x>>3 & mask;
```

62

Combine the sums

```
 /* Now combine high and low order sums  */
  s = s + (s >> 16);

  /* Low order 16 bits now consists of 4 sums.
     Split into two groups and sum  */
  mask = 0xF | (0xF << 8);
  s = (s & mask) + ((s >> 4) & mask);
  return (s + (s>>8)) & 0x3F;
}
```

63