CS 241
Data Organization
Binary

Brooke Chenoweth

University of New Mexico

Fall 2014

Combinations and Permutations

In English we use the word "combination" loosely, without thinking if the order of things is important. In other words:

- *"My fruit salad is a* **combination** *of apples, grapes and bananas."* In this statement, order does not matter: "bananas, grapes and apples" or "grapes, apples and bananas" make the same salad.
- *"The* **combination** *to the safe is 472."* Here the order is important: "724" would not work, nor would "247".

Combinations and Permutations

In Computer Science and Math, we use more precise language:

- If the order doesn't matter, it is a **Combination**.
- If the order does matter it is a **Permutation**.
    - *Repetition is Allowed*: such as the lock above. It could be "333".
    - *No Repetition*: for example the first three people in a running race. Order does matter, but you can't be first *and* second.

Information in a Binary Signal

1 Bit
2 Permutations

0
1

2 Bits
4 Permutations

0	0
0	1
1	0
1	1

3 Bits
8 Permutations

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

4 Bits
16 Permutations

0000	1000
0001	1001
0010	1010
0011	1011
0100	1100
0101	1101
0110	1110
0111	1111

Numbers in Base Ten and Base Two

Base 10

$$\begin{aligned} 5307 &= 5 \times 10^3 &&+ 3 \times 10^2 &&+ 0 \times 10^1 &&+ 7 \times 10^0 \\ &= 5000 &&+ 300 &&+ 0 &&+ 7 \end{aligned}$$

Base 2

$$\begin{aligned} 1011 &= 1 \times 2^3 &&+ 0 \times 2^2 &&+ 1 \times 2^1 &&+ 1 \times 2^0 \\ &= 8 &&+ 0 &&+ 2 &&+ 1 \end{aligned}$$

Examples of Binary Numbers

0	0	0	0	**1**	0	0	0	**1**	**1**	= 35
512	256	128	64	**32**	16	8	4	**2**	**1**	

0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	= 63
512	256	128	64	**32**	**16**	**8**	**4**	**2**	**1**	

0	0	0	**1**	0	0	0	0	0	0	= 64
512	256	128	**64**	32	16	8	4	2	1	

1	**1**	0	**1**	**1**	0	0	0	**1**	**1**	= 867
512	**256**	128	**64**	**32**	16	8	4	**2**	**1**	

Hexadecimal: Base-16

Hexadecimal (or hex) is a base-16 system that uses sixteen distinct symbols, most often the symbols 09 to represent values zero to nine, and A, B, C, D, E, F to represent values ten to fifteen.

Base 16

$$
\begin{array}{llllll}
0x53AC & = 5\times16^3 & +3\times16^2 & +10\times16^1 & +12\times16^0 \\
 & = 5\times4096 & +3\times256 & +10\times16 & +12\times1 \\
 & = 20{,}480 & +768 & +160 & +12 \\
 & = 21{,}420 & & & \\
\end{array}
$$

Why Hexadecimal?

- Hexadecimal is more compact than base-10
- Hexadecimal is way more compact that base-2
- Since 16 is a power of 2, it is very easy to convert between Binary and Hexadecimal

Base 16							
Four bytes:			0x01239ACF				
01		23		9A		BF	
0000	0001	0010	0011	1001	1010	1011	1111

Hexadecimal Literals

```
#include <stdio.h>
void main(void)
{
   printf("%d\n", 0x1);
   printf("%d\n", 0x2);
   printf("%d\n", 0x3);
   printf("%d\n", 0x8);
   printf("%d\n", 0x9);
   printf("%d\n", 0xA);
   printf("%d\n", 0xB);
   printf("%d\n", 0xC);
   printf("%d\n", 0xD);
   printf("%d\n", 0xE);
   printf("%d\n", 0xF);
   printf("%d\n", 0x10);
   printf("%d\n", 0x11);
   printf("%d\n", 0x12);
}
```

```
1
2
3
8
9
10
11
12
13
14
15
16
17
18
```

Hexadecimal Literals (using %x)

```c
#include <stdio.h>
void main(void)
{
   printf("%x\n", 0x1);
   printf("%x\n", 0x2);
   printf("%x\n", 0x3);
   printf("%x\n", 0x8);
   printf("%x\n", 0x9);
   printf("%x\n", 0xA);
   printf("%x\n", 0xB);
   printf("%x\n", 0xC);
   printf("%x\n", 0xD);
   printf("%x\n", 0xE);
   printf("%x\n", 0xF);
   printf("%x\n", 0x10);
   printf("%x\n", 0x11);
   printf("%x\n", 0x12);
}
```

```
1
2
3
8
9
a
b
c
d
e
f
10
11
12
```

Powers of 2: char, int

```c
#include <stdio.h>
void main(void)
{
   char i=0;
   char a=1;
   unsigned char b=1;
   int c = 1;
   for (i=1; i<22; i++)
   {
      a = a * 2;
      b = b * 2;
      c = c * 2;
      printf("%2d) %4d %3d %7d\n",
                i, a, b, c);
   }
}
```

```
 1)    2   2       2
 2)    4   4       4
 3)    8   8       8
 4)   16  16      16
 5)   32  32      32
 6)   64  64      64
 7) -128 128     128
 8)    0   0     256
 9)    0   0     512
10)    0   0    1024
11)    0   0    2048
12)    0   0    4096
13)    0   0    8192
14)    0   0   16384
15)    0   0   32768
16)    0   0   65536
17)    0   0  131072
18)    0   0  262144
19)    0   0  524288
20)    0   0 1048576
21)    0   0 2097152
```

Powers of 2: int, long

```
#include <stdio.h>
void main(void)
{
  char i=0;
  int c=1;
  long d = 1;
  for (i=1; i<65; i++)
  {
    c = c * 2;
    d = d * 2;
    printf("%2d) %11d %20ld\n",
              i, c, d);
  }
}
```

```
...
29)   536870912            536870912
30)  1073741824           1073741824
31) -2147483648           2147483648
32)           0           4294967296
33)           0           8589934592
...
61)           0  2305843009213693952
62)           0  4611686018427387904
63)           0 -9223372036854775808
64)           0                    0
```

Format code: `ld` for long decimal

Bit Operations

C provides several operators for manipulating the individual bits of a value:

`&`	bitwise AND	`1010 & 0011 = 0010`
`|`	bitwise OR	`1010 | 0011 = 1011`
`^`	bitwise XOR	`1010 ^ 0011 = 1001`
`~`	one's complement	`~1010 = 0101`
`<<`	left-shift	`00000100 << 3 = 00100000`
`>>`	right-shift	`00000100 >> 2 = 00000001`

Shift Operator Example

```c
void main(void)
{
  int i;
  for (i=0; i<8; i++)
  {
    unsigned char n = 1 << i;
    printf("n=%d\n", n);
  }
}
```

Output:

n=1
n=2
n=4
n=8
n=16
n=32
n=64
n=128

Convert 77 to an 8-bit Binary String

$2^7 = 128$ is > 77, put a '0' in the 128s place

0							

$2^6 = 64$ is $<= 77$, put a '1' in the 64s place
and subtract 64: $77 - 64 = 13$

0	1						

$2^5 = 32$ is > 13, put a '0' in the 32s place

0	1	0					

$2^4 = 16$ is > 13, put a '0' in the 16s place

0	1	0	0				

$2^3 = 8$ is $<= 13$, put a '1' in the 8s place
and subtract 8: $13 - 8 = 5$

0	1	0	0	1			

$2^2 = 4$ is $<= 5$, put a '1' in the 4s place
and subtract 4: $5 - 4 = 1$

0	1	0	0	1	1		

$2^1 = 2$ is > 1, put a '0' in the 2s place

0	1	0	0	1	1	0	

$2^0 = 1$ is $<= 1$, put a '1' in the 1s place
and subtract 1: $1 - 1 = 0$

0	1	0	0	1	1	0	1

Convert unsigned char to Binary Array

```c
#include <stdio.h>
void main(void)
{
  char bits[9];
  bits[8] = '\0';
  unsigned char n=83;
  unsigned char powerOf2 = 128;
  int i;
  for (i=0; i<=7; i++)
  { if (n >= powerOf2)
    { bits[i] = '1';
      n = n-powerOf2;
    }
    else bits[i] = '0';
    powerOf2 /= 2;
  }
  printf("%s\n", bits);
}
```

Output:

01010011

The Mask

```c
void main(void)
{
  long mask = 1<<23;
  long x = 25214903917;

  /* Not zero if bit 23 is ON in x. */
  printf("%ld\n", x & mask); /* prints: 8388608 */

  /* Turn ON bit-23. If already ON, x is unchanged. *
  x = x | mask;
  printf("%ld\n", x); /* prints: 25214903917 */

  /* Turn OFF bit 23. If already OFF, x is unchanged.
  x = x & (~mask);
  printf("%ld\n", x); /* prints: 25206515309 */
}
```

Using the Mask: Binary Array

```
#include <stdio.h>
void main(void)
{
  char bits[9];
  bits[8] = '\0';
  unsigned char n=83;
  unsigned char powerOf2 = 128;
  int i;
  for (i=0; i<=7; i++)
  { if(n & powerOf2)
    { bits[i] = '1';
    }
    else bits[i] = '0';
    powerOf2 = powerOf2 >> 1;
  }
  printf("%s\n", bits);
}
```

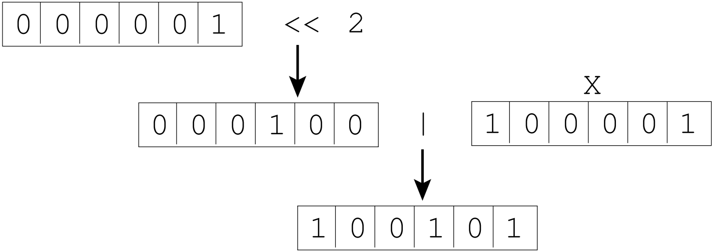
Output:

01010011

In the earlier slide, whenever a power of 2 is found, it is subtracted from n. This method never changes n.

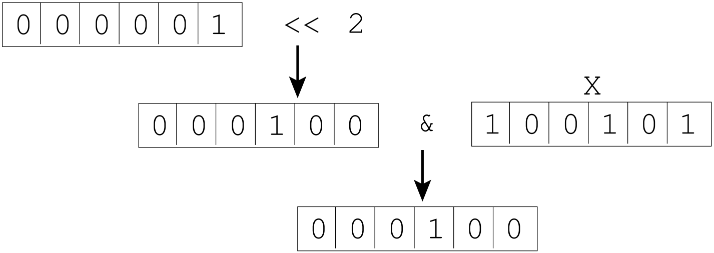
Bit-operations: `x |= (1<<n)`

- Set bit `n` in variable `x`. `(1<<n)` shifts 1 left by `n` bits. The result is OR'ed into `x`.



Bit-operations: `x & (1<<n)`

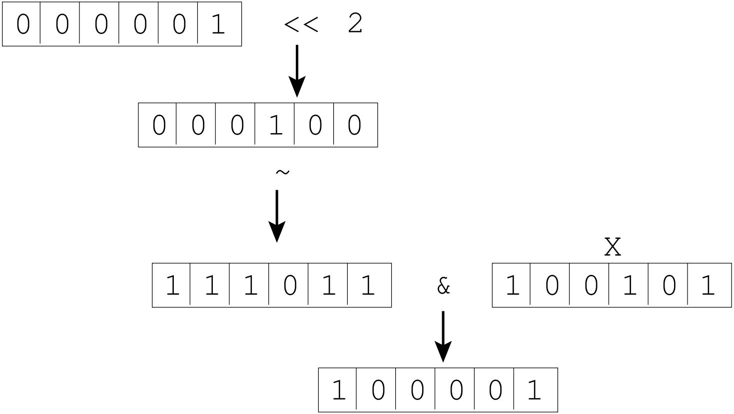
- Test bit `n` in variable `x`. `(1<<n)` creates a value with the appropriate bit set by shifting 1 left by `n` bits. It is AND'ed with `x` to see if that bit is set in `x`.



Bit-operations: `x & ~(1<<n)`

- Clear bit `n` in variable `x`.
- `(1<<n)` creates a value with the appropriate bit set by shifting 1 left by `n` bits.
- The one's complement operation '`~`' flips all the bits in the value, resulting in a value with every bit but the `n`'th set.
- It is AND'ed with `x` to clear the `n`'th bit but leave the rest unchanged.

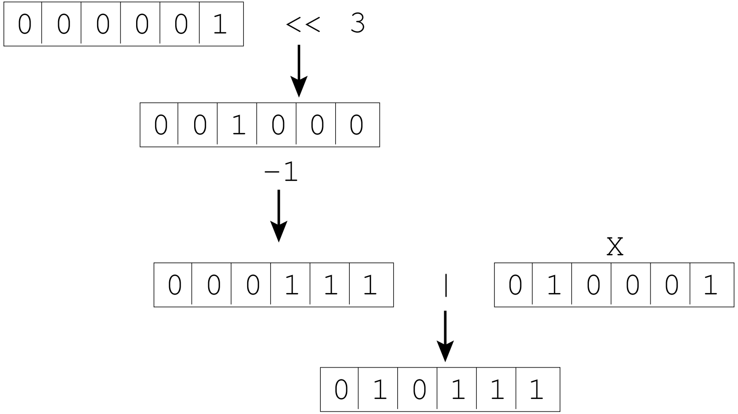
Bit-operations: `x & ~(1<<n)`



Bit-operations: $\boxed{\texttt{x |= (1<<n)-1}}$

- Set n lowest bits in variable `x`.
- We create a mask with all ones in the lower n bits by shifting 1 left by n bits and subtracting 1.
- It is OR'ed with `x` to set the n lowest bits but leave the rest unchanged.

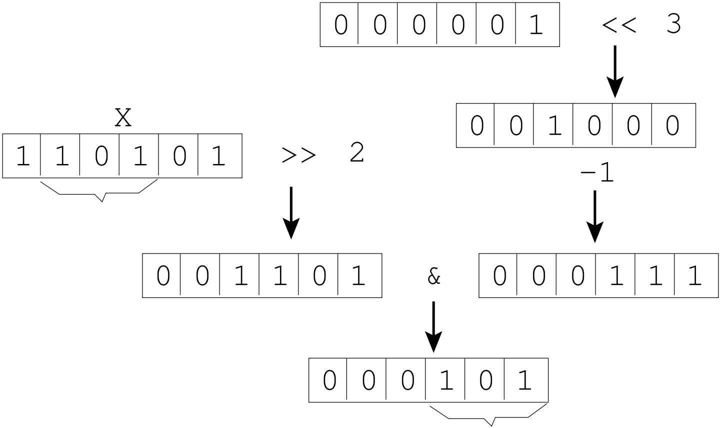
Bit-operations: `x |= (1<<n)-1`



Bit-operations: $\boxed{\texttt{(x>>p)\&((1<<n)-1)}}$

- Suppose we want to extract n bits from `x`, starting at position p.
- We create a mask with all ones in the lower n bits the same way as before: shift 1 left by n bits and subtract 1.
- Next, we shift `x` right by p bits.
- Finally, we AND the mask and `(x>>p)` to strip out any extra high-order bits.

Bit-operations: `(x>>2)&((1<<3)-1)`



Addition: Base 10 and Binary

Base 10

$$
\begin{array}{cccc}
 & & {\scriptstyle 1} & \\
 & & 2 & 9 \\
+ & & 5 & 6 \\
\hline
 & & 8 & 5
\end{array}
$$

Binary

$$
\begin{array}{ccccccccc}
 & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & & \\
 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
+ & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
\hline
 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}
$$

Overflow Addition

```
#include <stdio.h>
void main (void)
{
   char i=0;
   char a = 123, b = 252;
   unsigned char x = 123, y = 252;
   for (i=1; i<=7; i++)
   {
      a++; b++; x++; y++;
      printf("%4d %4d %4d %4d\n", a, x, b, y);
   }
}
```

Output:

124	124	-3	253
125	125	-2	254
126	126	-1	255
127	127	0	0
-128	128	1	1
-127	129	2	2
-126	130	3	3

Two's Complement

From ordinary binary:
Flip the bits and Add 1.

5	0	0	0	0	1	0	1
Flip Bits	1	1	1	1	0	1	0
Add 1	0	0	0	0	0	0	1
-5	1	1	1	1	0	1	1

Ordinary Binary	Decimal
0000 0001	1
0000 0010	2
0000 0011	3
0000 0100	4
0000 0101	5
0000 0010	6
0000 0111	7

Two's Complement	Decimal
1111 1111	-1
1111 1110	-2
1111 1101	-3
1111 1100	-4
1111 1011	-5
1111 1010	-6
1111 1001	-7

Two's Complement Addition

$$
\begin{array}{ccc}
 & & \\
 & 2 & 9 \\
+ & -2 & 9 \\
\hline
 & & 0
\end{array}
\qquad
\begin{array}{cccccccccc}
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\
 & & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
+ & & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
\hline
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

$$
\begin{array}{cc}
 & \\
 & 7 \\
+ & -4 \\
\hline
 & 3
\end{array}
\qquad
\begin{array}{cccccccccc}
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
 & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
+ & & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\hline
 & & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
$$