Binary representation and logic

bit - binary digit - 0/1, off/on

binary devices are relatively easy to build

**Mathematical diversion**: assume an n-state device where the cost of the device is linear in n (= kn); the number of devices required for representing an arbitrary number x is log\_base\_n(x); thus, the total cost to represent x is equal to kn*\log\_base\_n(x); find the number n that minimizes this cost. (turns out to be n = e = 2.718...)

n bits have $2^n$ different patterns (permutations)

unsigned integer representation = 0 to $2^n - 1$
Memory unit terminology

Memory contains series of bits, grouped into addressable units:

- **word** - unit of memory access and/or size of data registers, 16/32/64 bits
- **byte** - unit for character representation, 8 bits
- **nibble** - unit for binary-coded decimal (BCD) digit, 4 bits
Conversions

- **decimal** (base 10, digits 0-9)
- **binary** (base 2, digits 0,1)
- **octal** (base 8, digits 0-7)
- **hexadecimal** (base 16, digits 0-f)

**Binary-to-Decimal Conversion by Positional Representation**

**Decimal-to-Binary Conversion by Using Remainder Bits from Repeated Division by Two**
Conversions

To convert *decimal_number* from decimal to binary using repeated division:

```plaintext
repeat
    divide decimal_number by 2, getting quotient and remainder
    remainder is the next binary digit (right to left)
    decimal_number = quotient
until decimal_number = 0;
```
Conversions

$75_{10} = \underline{}_{2}$

$75 / 2 = 37 \ r \ 1$
$37 / 2 = 18 \ r \ 1$
$18 / 2 = 9 \ r \ 0$
$9 / 2 = 4 \ r \ 1$
$4 / 2 = 2 \ r \ 0$
$2 / 2 = 1 \ r \ 0$
$1 / 2 = 0 \ r \ 1$

Writing the remainder from right to left
$75_{10} = 1001011_{2}$
Conversions

$121_{10} = \underline{\phantom{0}}_{2}$

$121 / 2 = 60 \; r \; 1$
$60 / 2 = 30 \; r \; 0$
$30 / 2 = 15 \; r \; 0$
$15 / 2 = 7 \; r \; 1$
$7 / 2 = 3 \; r \; 1$
$3 / 2 = 1 \; r \; 1$
$1 / 2 = 0 \; r \; 1$
Conversions

To convert \textit{decimal\_number} from decimal to binary using the sum of powers of two:

write \textit{decimal\_number} as the sum of powers of two.

\[
\text{e.g. } 75_{10} = 64 + 8 + 2 + 1 = 2^6 + 2^3 + 2^1 + 2^0
\]

For the binary representation, place a 1 if the power is present in the sum and a 0 if it is not. In the sum above, \(2^5\), \(2^4\), and \(2^2\) are missing; therefore,

\[
75_{10} = 1001011_2
\]
Conversions

**binary-to-decimal** conversion by positional representation

\[10110111_2 = \text{___________}_{10}\]

\[1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\]

\[= 128 + 32 + 16 + 4 + 2 + 1\]

\[= 183\]
Conversions

**binary-to-hexadecimal and back using a table**

organize the stream of binary digits into groups of four.
find the hexadecimal value for each group of 4 bits.

```
10010010111000011010
1001  0010  1110  0001  1010
     9       2         E        1       A
```

**hexadecimal-to-binary**

convert each digit to its 4-bit equivalent

```
9   2   E   1   A
1001  0010  1110  0001  1010
```
Conversions

conversion from binary-to-octal and octal-to-binary in a similar manner: simply organize the stream of binary digits into groups of three.

10 010 010 111 000 011 010
2 2 2 7 0 3 2

similar conversions between decimal and other bases, but typically decimal is first converted to binary then to octal or hex
Conversions

Converting to octal and hexadecimal

\[ 121_{10} = 1111001_2 \]

\[ 1111001_2 = 001 \ 111 \ 001_2 = 171_8 \]

\[ 1111001_2 = 0111 \ 1001_2 = 79_{16} \]
Conversions

**gdb conversions**

- `p/d 0x123` -- print as decimal, passing a hexadecimal constant
- `p/x 123` -- print as hexadecimal, passing a decimal constant
- `p/t 123` -- print as binary, passing a decimal constant
  (note that a leading zero is significant to gdb, 0123 is taken as octal)

**Character encoding - ASCII**
Conversions

**Magic numbers**

numbers that read as English words/phrases when displayed in hexadecimal

- catch uninitialized variables / bad pointers - 0xdeadbeef, 0xbaadf00d
- identify file type - 0xcafebabe in Java class file
- typically chosen to be a large, negative, odd integer (and thus unlikely to be a useful program constant)

(see en.wikipedia.org/wiki/Hexspeak)
Signed binary number representations

Signed binary number
- sign-magnitude
- one's complement
- two's complement
Signed binary number representations

sign-magnitude:
  • This is the simplest method.
  • Write the number in binary, then put a 0 in the most significant bit (msb) if the number is positive and a 1 if it is negative.

\[ d_i \text{ is digit, } 0 \leq d_i < b \]
\[ b \text{ is base (or radix)} \]

e.g. for an 8-bit representation:

\[ +5 \text{ is } 00000101 \]
\[ -5 \text{ is } 10000101 \]
\[ +15 \text{ is } 00001111 \]
\[ -15 \text{ is } 10001111 \]
Signed binary number representations

sign-magnitude (cont’d)

- It is not the best method or representation because it makes computation awkward
  - Cannot simply add +5 + (-5), since that gives 10001010, which is NOT zero.
  - Addition and subtraction require attention to the sign bit.
  - Require much more complicated hardware to implement basic mathematical functions. i.e., add/subtract/multiply.
  - For addition, if the signs are the same, add the magnitudes as unsigned numbers and watch for overflow. If the signs differ, subtract the smaller magnitude from the larger, and keep the sign of the larger.
Signed binary number representations

sign-magnitude (cont’d):

Range: $+/- 2^{n-1} - 1$
Signed binary number representations

One's complement format of a number
- Change the number to binary, ignoring the sign.
- Add 0s to the left of the binary number to make a total of n bits
- If the sign is positive, no more action is needed.
- If the sign is negative, complement every bit (i.e. change from 0 to 1 or from 1 to 0)
- Example for 8-bit representation

+9 is 00001001
−9 is 11110110

Adding the two gives 11111111, which is −0
00000000 is +0. There are two representations for 0
Signed binary number representations

One's complement format of a number

• Change the number to binary, ignoring the sign.
• Add 0s to the left of the binary number to make a total of $n$ bits
• If the sign is positive, no more action is needed.
• If the sign is negative, complement every bit (i.e. change from 0 to 1 or from 1 to 0)
• Example for 8-bit representation

+9 is 00001001
−9 is 11110110

Adding the two gives 11111111, which is −0
00000000 is +0. There are two representations for 0
Signed binary number representations

Two's complement format of a number
- Most computers today use 2's complement representation for negative numbers.
- The 2's complement of a negative number is obtained by adding 1 to the 1's complement. i.e.

Two's complement format of a number
- Change the number to binary, ignoring the sign.
- Add 0s to the left of the binary number to make a total of n bits
- If the sign is positive, no more action is needed.
- If the sign is negative, complement every bit (i.e. change from 0 to 1 or from 1 to 0) and add 1 (i.e. add 1 to one’s complement)
Signed binary number representations

Examples converting to two’s complement representation:
For -13, 8-bit representation

00001101   base integer
11110010   1's complement
+1
11110011   2's complement

For -227, 12-bit representation

000011100011   base integer
111100011100   1's complement
+1
111100011101   2's complement
Signed binary number representations

Converting from two's complement to decimal: Suppose you already have a number that is in two’s complement representation and you want to find its value in binary

- If the sign bit is 1, the number is a negative number. If it is a 0, the number is positive.
- If the number is a positive number (sign bit is 0), convert the binary number to decimal and you are done.
- If the number is a negative number (sign bit is 1)
  - complement each bit
  - add 1
  - convert the binary number to decimal
  - put a minus sign in front
Signed binary number representations

What makes 2’s complement better?

• Addition, subtraction, and multiplication all work in 2’s complement

• If we add +5 and -5 in 8-bit representation, we get

  00000000

  00000101 + 11111011 = 00000000  (we discard the carry)
## Signed binary number representations

Two's complement encoding - note one zero and asymmetric range. The first (leftmost) bit is the sign bit (1 => -).

<table>
<thead>
<tr>
<th>binary</th>
<th>sign magnitude</th>
<th>one's complement</th>
<th>two's complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+0</td>
<td>+0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>0010</td>
<td>+2</td>
<td>+2</td>
<td>+2</td>
</tr>
<tr>
<td>0011</td>
<td>+3</td>
<td>+3</td>
<td>+3</td>
</tr>
<tr>
<td>0100</td>
<td>+4</td>
<td>+4</td>
<td>+4</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
<td>+5</td>
<td>+5</td>
</tr>
<tr>
<td>0110</td>
<td>+6</td>
<td>+6</td>
<td>+6</td>
</tr>
<tr>
<td>0111</td>
<td>+7</td>
<td>+7</td>
<td>+7</td>
</tr>
<tr>
<td>1000</td>
<td>-0</td>
<td>-7</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-1</td>
<td>-6</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-2</td>
<td>-5</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-3</td>
<td>-4</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-5</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-6</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-7</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
Signed binary number representations

range for n-bit field: unsigned is $[0, 2^n - 1]$
2's compl. signed is $[-2^{n-1}, 2^{n-1} - 1]$
## Binary Logic

<table>
<thead>
<tr>
<th>a</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>or</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>xor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**ARM**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>a = 0 0 1 1</th>
<th>b = 0 1 0 1</th>
<th>Some Common Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>0 0 0 0</td>
<td>false, zero, clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>a and b</td>
<td>0 0 0 1</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>bic</td>
<td>a and (not b)</td>
<td>0 0 1 0</td>
<td>and-not, inhibit, a&gt;b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>0 0 1 1</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0 1 0 0</td>
<td>inhibit, a&lt;b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eor</td>
<td>a xor b</td>
<td>0 1 1 0</td>
<td>xor, exclusive or, a!=b</td>
<td></td>
</tr>
<tr>
<td>orr</td>
<td>a or b</td>
<td>0 1 1 1</td>
<td>or, inclusive or</td>
<td></td>
</tr>
</tbody>
</table>

iff, xnor, exclusive nor
Binary Logic

**Logic operators in C:** && and  || or  ! not
(zero word = false, nonzero word = true)

**Bitwise operators in C:** & and  | or  ~ not  ^ xor
(each bit in word independent)

**Logic operators in ARM assembly language:**
and[cc]  bic[cc]  eor[cc]  orr[cc]