Numeral Systems

Which number is larger?

25 8

We need to distinguish between numbers and the symbols that represent them, called numerals.

The number 25 is larger than 8, but the numeral 8 above is larger than the numeral 25.

The number twenty-five can be represented in many ways:

Decimal system (base 10): 25
Roman numerals: XXV
Binary system (base 2): 11001
Hexadecimal (base 16): 19

On a computer, the binary switch is easy to implement, so numbers are stored in a computer as binary (on and off).

In fact, everything stored in a computer is encoded using the binary number system.

numbers, characters, strings, booleans, objects, instructions

We need to understand the binary system to understand computers.

Positional number representation

Choose a **base (or radix) b**, and a set of **b** distinct symbols.

The radix b representation of a nonegative integer m is a string of digits chosen from the set { 0, 1, 2, 3, … , b-1 } so that if

$$m = (d_k d_{k-1} \ldots d_3 d_2 d_1 d_0)_b,$$

then $m = d_k b^k + d_{k-1} b^{k-1} + \ldots + d_3 b^3 + d_2 b^2 + d_1 b^1 + d_0 b^0$

We can represent any *nonnegative* number in **base b** by forming such a polynomial where for all d_k, $0 \le d_k < b$.

If b>10, new symbols must be used for digits above 9.

Examples

Decimal (b=10): Set of digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

$$493 = 4 \cdot 10^2 + 9 \cdot 10^1 + 3 \cdot 10^0 = 400 + 90 + 3$$

Binary (b=2): Set of digits = { 0, 1 }

$$\begin{aligned} 11010_2 &= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 16 + 8 + 2 = 26_{10} \end{aligned}$$

Octal (b=8): Set of digits = { 0, 1, 2, 3, 4, 5, 6, 7 }

$$\begin{aligned} 2076_8 &= 2 \cdot 8^3 + 0 \cdot 8^2 + 7 \cdot 8^1 + 6 \cdot 8^0 \\ &= 2 \cdot 512 + 56 + 6 \\ &= 1086_{10} \end{aligned}$$

Hexadecimal (b=16):

Set of digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }

$$\begin{aligned} \mathrm{B3F}_{16} &= 11 \bullet 16^2 + 3 \bullet 16^1 + 15 \bullet 16^0 \\ &= 11 \bullet 256 + 48 + 15 \\ &= 2879_{10} \end{aligned}$$

Note that the arithmetic is done in the decimal number system.

Number Conversions

1. **Converting a radix b numeral to decimal**

   Evaluate the corresponding polynomial.

   See examples above or use Horner's Method.

Horner's Method for Evaluating Polynomials:

Start with zero and then alternate adding a digit with multiplying by the base.

Binary Example

$$\begin{aligned} 1011011_2 &= 1 \bullet 2^6 + 0 \bullet 2^5 + 1 \bullet 2^4 + 1 \bullet 2^3 + 0 \bullet 2^2 + 1 \bullet 2^1 + 1 \bullet 2^0 \\ &= ((((((0+1) \bullet 2+0) \bullet 2+1) \bullet 2+1) \bullet 2+0) \bullet 2+1) \bullet 2+1 \end{aligned}$$

Multiply by 2 moving down; add a bit moving right.

$0 + 1 = 1$
$2 + 0 = 2$
$4 + 1 = 5$
$10 + 1 = 11$
$22 + 0 = 22$
$44 + 1 = 45$
$90 + 1 = 91_{10}$

Octal Example

$456_8 = 4 \bullet 64 + 5 \bullet 8 + 6 = 302_{10}$

$0 + 4 = 4$

$32 + 5 = 37$

$296 + 6 = 302_{10}$

Hexadecimal Example $37D2_{16}$

$0 + 3 = 3$

$\underline{x\ 16}$

$48 + 7 = 55$

$\underline{x16}$

$880 + 13 = 893$

$\underline{x16}$

$14288 + 2 = 14290_{10}$

2. Converting a decimal numeral to radix b

Suppose a decimal numeral m has the following representation in radix *b*:

$$m = d_k b^k + d_{k-1} b^{k-1} + \ldots + d_3 b^3 + d_2 b^2 + d_1 b + d_0$$

Divide *m* by *b* using decimal arithmetic.

$$\text{quotient} = d_k b^{k-1} + d_{k-1} b^{k-2} + \ldots + d_3 b^2 + d_2 b + d_1$$

$$\text{remainder} = d_0$$

Continue dividing quotients by *b*, saving the remainders, until the quotient is zero.

Pseudo-code Version of Algorithm

```
num = decimal numeral to be converted;
k = 0;
do
{
    $d_k$ = num % b;
    write symbol for $d_k$;     // k denotes positions of
    num = num / b;              // symbols from right to left.
    k++;
}
while (num > 0);
```

Resulting numeral, base *b*

$d_9\ d_8\ d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0$

for as many digits as are produced until *num* becomes zero.

Examples

- Decimal to Octal: 1492_{10}

```
    186          23           2         0
 8)1492       8)186       8)23      8) 2
    8           16          16         0
   69            26          7         2
   64            24
    52            2
    48
     4
```

`Result =` 2724_8

- Decimal to Hexadecimal: 438_{10}

```
      27             1             0
  16)438         16)27         16)1
     32             16            0
      118           11            1
      112
        6
```

Result = $1B6_{16}$

- Decimal to Binary: 777_{10}

Quotients	Remainders
777	
388	1
194	0
97	0
48	1
24	0
12	0
6	0
3	0
1	1
0	1

Result = 1100001001_2

Some Useful Values

2•16 = 32	6•16 = 96
3•16 = 48	7•16 = 112
4•16 = 64	8•16 = 128
5•16 = 80	9•16 = 144
$16^2 = 256 = 2^8$	$16^4 = 65{,}536 = 2^{16}$
$16^3 = 4{,}096 = 2^{12}$	$16^5 = 1{,}048{,}576 = 2^{20}$

3. Converting Binary to Hexadecimal:

   Group binary digits, 4 bits at a time, proceeding from right to left, and transform the groups into hexadecimal digits.

Binary to Hexadecimal

$11010011101101_2 = 11\ 0100\ 1110\ 1101 = 34ED_{16} = 34ED_H$

4. Converting Hexadecimal to Binary:

   Substitute correct 4 bit patterns for hexadecimal digits.

Hexadecimal to Binary

$7A5F_{16} = 0111\ 1010\ 0101\ 1111_2$

Octal and Hexadecimal

These number systems provide us with a convenient form of shorthand for working with binary quantities.

```
int m = 25979882;
```

m is stored as 32 bits: 0000 0001 1000 1100 0110 1011 1110 1010

Easier to represent as 8 hex digits: 018c6bea

Learn the four-bit binary patterns for the 16 hex digits.

Note: We are only dealing with nonnegative integers at this time.

We call them *unsigned* integers.

Arithmetic

Algorithms for addition and multiplication are essentially the same for any radix system.

The difference lies in the tables that define the sum and product of two digits.

Binary Addition:

+	0	1
0	0	1
1	1	10

Binary Multiplication:

•	0	1
0	0	0
1	0	1

Example

```
     1101101
     x 11011
    --------
     1101101
    1101101
   0000000        Note: 1+1+1+1+1 = 101
  1101101
+1101101
------------
101101111111
```

Tables for hexadecimal are trickier.

A + A = 14

A • A = 64

Practice: Construct an addition table and a multiplication table in hexadecimal.

Finite Representation

Numbers inside a computer are constrained to a fixed finite number of bits, which affects range of numbers we can represent.

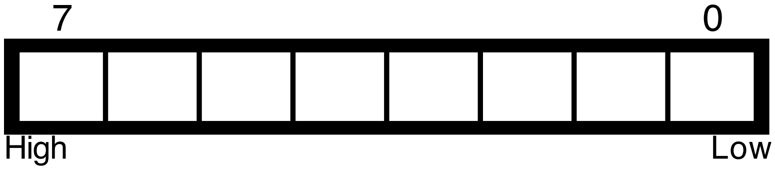
With n bits:

Number of possibilities = $2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 = 2^n$

Unsigned numbers may range from 0 to $2^n - 1$.

Example

Consider n = 8



Range: 0 to 2^8-1 (0 to 255 decimal)

Hexadecimal representations: 00 to FF

Representing Negative Integers

1. Sign / magnitude

Leftmost bit represents a sign, 0 for + and 1 for –.
The rest of the bits represent the *absolute value* of number.

With k=6 bits,

$$9 = 0\ 01001$$
$$-9 = 1\ 01001$$

Range of values: $-(2^{k-1} - 1)$ to $(2^{k-1} - 1)$.

For k=6, the values range from -31 to 31.

Only 5 bits available to represent the magnitude.

The maximum and minimum numbers, 011111 and 111111, have equal magnitude

Problems

- We have two forms of zero, 000000 and 100000.
- Arithmetic is expensive because we need an adder and a subtractor to do addition.

If the signs are the same, add the magnitudes.

If the signs are different, subtract the smaller magnitude from the larger and keep the sign from the larger magnitude.

2. Two's complement

Nonnegative integers 0 to (2^{k-1} - 1) are represented in binary the same way as with sign-magnitude, 0 followed by a (k-1)-bit absolute value.

Negative integers -(2^{k-1}) to -1 are represented by adding 2^k and expressing the result in binary.

Example k=6

Note that $2^k = 64$.

$2 = 000010_2$

$13 = 001101_2$

$30 = 011110_2$

$31 = 011111_2$

$-2 \Rightarrow 64 - 2 = 62 = 111110_2$

$-13 \Rightarrow 64 - 13 = 51 = 110011_2$

$-30 \Rightarrow 64 - 30 = 34 = 100010_2$

$-32 \Rightarrow 64 - 32 = 32 = 100000_2$

Arithmetic: 2 + (-2) = 000010 + 111110 = 000000

Ways of Interpreting Bit Patterns (k = 4)

BIT PATTERN	UNSIGNED	SIGN & MAGNITUDE	TWO'S COMPLEMENT
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	-0	-8
1001	9	-1	-7
1010	10	-2	-6
1011	11	-3	-5
1100	12	-4	-4
1101	13	-5	-3
1110	14	-6	-2
1111	15	-7	-1

Algorithm for Negating a Number in two's complement

a) Complement (flip) each bit.
b) Add 1

Complementing a k-bit number is same as subtracting it from $2^k - 1$.

Examples

k=6 Note $2^k - 1 = 111111_2$

$13 = 001101_2$

a) Complement bits: 110010

b) Add one: 110011

$-13 = 110011_2$

a) Complement bits: 001100

b) Add one: 001101

$0 = 000000_2$

a) Complement bits: 111111

b) Add one: 000000 (discard left carry)

The high order bit still represents sign of the number (0 for nonnegative and 1 for negative).

Now we have only one form of zero.

The magnitudes of the maximum and minimum values differ:

$$| \text{minimum} | = \text{maximum} + 1$$

Hence, one number cannot have the negation operation performed on it. Which one?

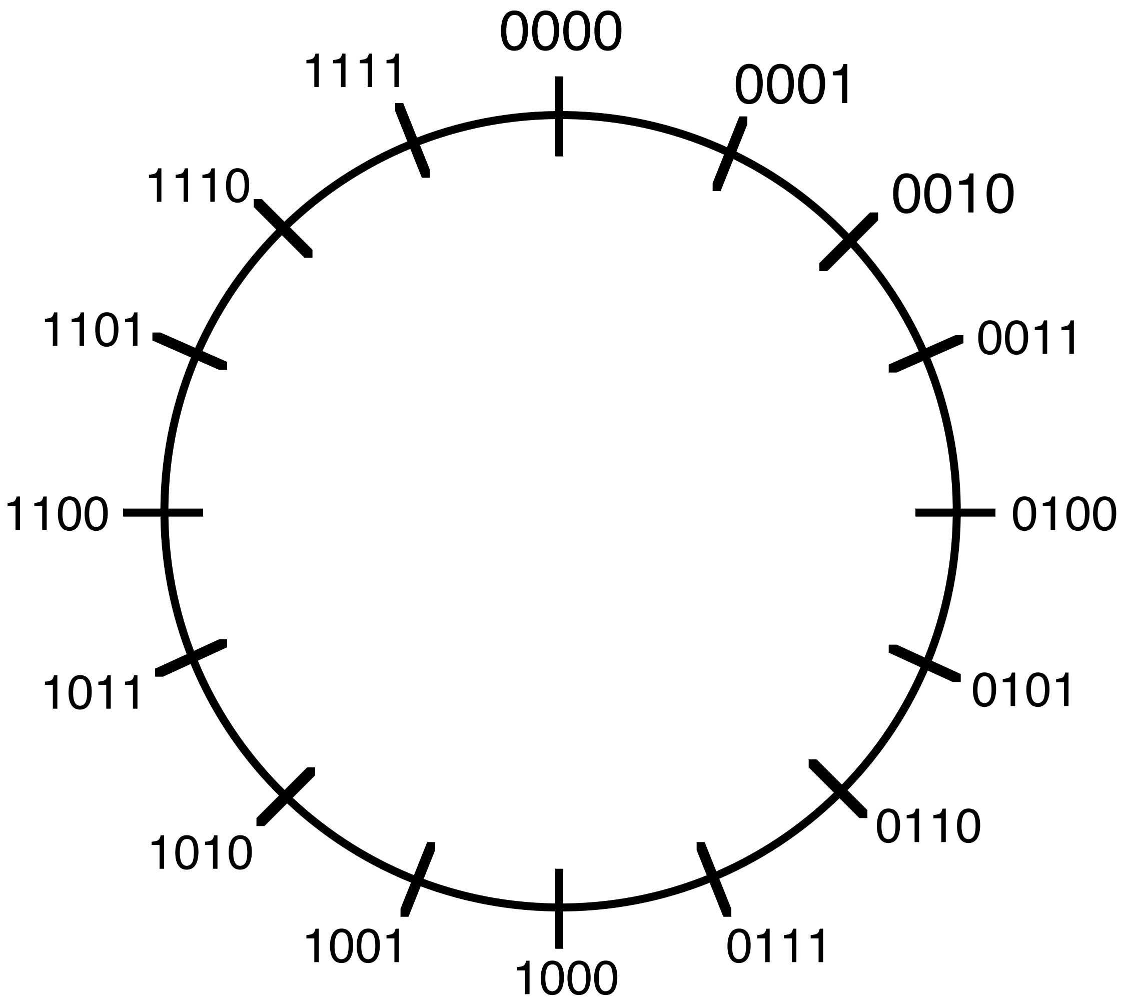
int num = -2147483648;

-num is an erroneous operation now.

long big = -9223372036854775808;

-big is an erroneous operation now.

Odometer principle for twos-complement.



Evaluating Twos Complement

The problem is converting negative twos complement to decimal.

Procedure

1. Find twos complement negative of number (a positive value).
2. Convert that number to decimal
3. Append a minus sign.

Examples

k = 8

10010111

Negate to get 01101001

Convert to decimal: 64 + 32 + 8 + 1 = 105

Append minus: -105

11111111

Negate to get 00000001

Convert to decimal: 1

Append minus: -1

11000000

Negate to get 01000000

Convert to decimal: 64

Append minus: -64

Exercise: Assuming 6 bit numbers, what values are represented by a) 011101 and b) 111010?

Addition of Signed Numbers

Sign / magnitude: Follow the same rules you use when working with the decimal arithmetic learned in grade school.

Add magnitudes when signs are the same.

Subtract magnitudes when signs are different.

Example 1

```
 101101      Since the signs are the same, we add the
+100111      magnitudes of the two numbers.
-------      Note that we do not add the signs.

 01101
+00111
-------
 10100       Now append the sign to get 110100
```

Example 2

```
 110101      Since the signs are different, we subtract
+001101      the magnitudes of the two numbers.
-------      Note that we do not subtract the signs.

 10101       Larger magnitude
-01101       Smaller magnitude
-------
 01000       Now append sign of the larger to get 101000
```

Two's complement Arithmetic

Add right to left, bit by bit, *including* the sign bits.
Ignore any carry out of the sign position.

```
Binary         Decimal      Explanation (optional)

 01010           10
+00011         +  3
------         ---
 01101           13

 11001           -7          25-32
+00101         +  5         + 5
------         ---          ------
 11110           -2          30-32

 11001           -7          25-32
+11001         +-7          +25-32
------         ---          ------
110010*        -14           50-32-32
------                            = 32+18-32-32
 10010                            = 18-32
```

* Carries are discarded

Overflow

- Add two nonnegative numbers and get a negative answer.
- Add two negative numbers and get a nonnegative answer.

Radix Conversions in Java

Converting decimal (int) to other bases

```
int m = 123456;
Integer.toBinaryString(m)   returns "11110001001000000"
Integer.toOctalString(m)    returns "361100"
Integer.toHexString(m)      returns "1e240"
Integer.toString(m)         returns "123456"
```

Converting other bases to decimal

```
Integer.parseInt("11110001001000000", 2)
Integer.parseInt("361100", 8)
Integer.parseInt("1e240", 16)
Integer.parseInt("123456")
                            each returns the int 123456
```

Octal and Hex Literals and Unicode

```
int k = 0123;           // leading 0 means octal digits
int m = 0xabc;          // leading 0x means hex digits
char c = '\u0041';      // char 'A' = 41 hex
```

Logical Instructions

Java performs logical operations on integers of type **int** and **long**.

int m1, m2, n.
m1 = *some value*;
m2 = *some other value*;

OR

n = m1 | m2;

m1 and m2 are bit-wise OR-ed to produce the result.

1 | 1 = 1
1 | 0 = 1　　1 means true
0 | 1 = 1　　0 means false
0 | 0 = 0

AND

n = m1 & m2;

m1 and m2 are bit-wise AND-ed to produce the result.

1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0

Exclusive OR

n = m1 ^ m2;

m1 and m2 are bit-wise Exclusive OR-ed to produce the result.

1 ^ 1 = 0　　Equivalent to ≠
1 ^ 0 = 1
0 ^ 1 = 1
0 ^ 0 = 0

NOT

```
n = ~m1;
```

All the bits in m1 are complemented to produce the result.

```
~1 = 0
~0 = 1
```

Examples

Suppose we have declarations

```
int m = 0x6ca64;
int n = 0xB93DE;
int ans;
```

OR

```
ans = m | n;
```

```
  m   = 0000 0000 0000 0110 1100 1010 0110 0100
  n   = 0000 0000 0000 1011 1001 0011 1101 1110
m | n = 0000 0000 0000 1111 1101 1011 1111 1110
```

AND

```
ans = m & n;
```

```
  m   = 0000 0000 0000 0110 1100 1010 0110 0100
  n   = 0000 0000 0000 1011 1001 0011 1101 1110
m & n = 0000 0000 0000 0010 1000 0010 0100 0100
```

Exclusive OR (acts like not-equal)

ans = m ^ n;

```
  m   = 0000 0000 0000 0110 1100 1010 0110 0100
  n   = 0000 0000 0000 1011 1001 0011 1101 1110
m ^ n = 0000 0000 0000 1101 0101 1001 1011 1010
```

NOT

ans = ~m;

```
  m   = 0000 0000 0000 0110 1100 1010 0110 0100
 ~m   = 1111 1111 1111 1001 0011 0101 1001 1011
```

Bit Masks

Terminology
Clear a bit: Make it 0
Set a bit: Make it 1

Problem: Clear all the bits in n.

n = 0;

Problem: Clear bits in the first and third bytes in n.

n = n & 0x00FF00FF   // lead zeros redundant

Problem: Set all the bits in n.

n = 0xFFFFFFFF;

or   n = -1;

Problem: Set the bits in the first and fourth bytes in n.

n = n | 0xFF0000FF;

Problem: Flip all the bits in n.

```
n = ~n;
```

Problem: Flip the bits in the fourth byte in n.

```
n = n ^ 0x000000FF;        //  lead zeros redundant
```

Shift Operations

Move the bits in a word to the left or right.

Two Kinds

Logical Shifts:	Word filled with zeros as bits are moved.
Arithmetic Shifts:	Left shift is same as logical shift. Right shift replicates the sign bit

Shift operations take two operands.

```
int word;          // Word to be shifted
int numb;          // Number of bits to shift
```

Only the last six bits of the shift number are considered, giving shift distance $0 \le numb \le 63$.

Logical shift left

```
word = word << numb;
```

Logical shift right

```
word = word >>> numb;
```

Arithmetic shift left

```
word = word << numb;
```

Arithmetic shift right

word = word >> numb;

Examples

Suppose we have declarations

int p = 500;
int q = -12;
int ans;

ans = p << 2;

```
p     = 0000 0000 0000 0000 0000 0001 1111 0100
p<<2  = 0000 0000 0000 0000 0000 0111 1101 0000
```

ans has the value 2000.

ans = p >>> 2;   // same as p >> 2 in *this* case (p ≥ 0)

```
p     = 0000 0000 0000 0000 0000 0001 1111 0100
p>>> 2 = 0000 0000 0000 0000 0000 0000 0111 1101
```

ans has the value 125.

ans = q << 3;

```
q     = 1111 1111 1111 1111 1111 1111 1111 0100
  q<<3 = 1111 1111 1111 1111 1111 1111 1010 0000
```

ans has the value -96.

ans = q >> 3;

```
q     = 1111 1111 1111 1111 1111 1111 1111 0100
  q>>3 = 1111 1111 1111 1111 1111 1111 1111 1110
```

ans has the value -2.

ans = q >>> 3;

```
q      = 1111 1111 1111 1111 1111 1111 1111 0100
q>>>3 = 0001 1111 1111 1111 1111 1111 1111 1110
```

ans has the value 536870910.

Sign Extension

Observe what happens when integers are enlarged or truncated.

	Binary Storage	**Hexadecimal**
byte b1 = 100;	0110 0100	64
byte b2 = -10;	1111 0110	F6
int n1 = b1;	0000 0000 0000 0000 0000 0000 0110 0100	00000064
int n2 = b2;	1111 1111 1111 1111 1111 1111 1111 0110	FFFFFFF6
int n3 = -200;		FFFFFF38
byte b3 = (**byte**)n3;		38 (= 56_{10})
int n4 = 150;		00000096
byte b4 = (**byte**)n3;		96 (= -106_{10})

Character Data

Computers do not store character data directly.

IO devices provide for a mapping between the character symbols and internal form of the characters.

For portability between machines, standard encodings of characters are used.

Ascii	8 bits	Up to 256 characters
Unicode	16 bits	Up to 65,536 characters

Unicode is an extension of ascii:

Ascii 'A' is 01000001

Unicode 'A' is 00000000 01000001

Typing the letter A on the keyboard

sends the binary string 01000001 to the computer,

which stores it as 00000000 01000001 in Java.

ASCII Character Set

In Hexadecimal

```
| 00 nul| 01 soh| 02 stx| 03 etx| 04 eot| 05 enq| 06 ack| 07 bel|
| 08 bs | 09 ht | 0a nl | 0b vt | 0c np | 0d cr | 0e so | 0f si |
| 10 dle| 11 dc1| 12 dc2| 13 dc3| 14 dc4| 15 nak| 16 syn| 17 etb|
| 18 can| 19 em | 1a sub| 1b esc| 1c fs | 1d gs | 1e rs | 1f us |
| 20 sp | 21  ! | 22  " | 23  # | 24  $ | 25  % | 26  & | 27  ' |
| 28  ( | 29  ) | 2a  * | 2b  + | 2c  , | 2d  - | 2e  . | 2f  / |
| 30  0 | 31  1 | 32  2 | 33  3 | 34  4 | 35  5 | 36  6 | 37  7 |
| 38  8 | 39  9 | 3a  : | 3b  ; | 3c  < | 3d  = | 3e  > | 3f  ? |
| 40  @ | 41  A | 42  B | 43  C | 44  D | 45  E | 46  F | 47  G |
| 48  H | 49  I | 4a  J | 4b  K | 4c  L | 4d  M | 4e  N | 4f  O |
| 50  P | 51  Q | 52  R | 53  S | 54  T | 55  U | 56  V | 57  W |
| 58  X | 59  Y | 5a  Z | 5b  [ | 5c  \ | 5d  ] | 5e  ^ | 5f  _ |
| 60  ` | 61  a | 62  b | 63  c | 64  d | 65  e | 66  f | 67  g |
| 68  h | 69  i | 6a  j | 6b  k | 6c  l | 6d  m | 6e  n | 6f  o |
| 70  p | 71  q | 72  r | 73  s | 74  t | 75  u | 76  v | 77  w |
| 78  x | 79  y | 7a  z | 7b  { | 7c  | | 7d  } | 7e  ~ | 7f del|
```

Converting a Digit Character to Numeric

Suppose ch contains an ascii digit.

char ch = *an ascii digit*;

Conversion:

int m = ch - '0'; // int arithmetic

or

int n = ch & 0xF;

Examples '6' - '0' = 54 - 48 = 6

00000036 & 0000000F = 0000006 // hexadecimal