Chapter 4: Data Representations

Data Representation

Goal: to store numbers, characters, etc. in computer

Location: store in a memory location
a BOX or CONTAINER that can hold a value

(Memory is just an array (1-D) of these boxes, address is just the array index)

Concentrate on one box.

---

binary representation- represent all information using only 0s and 1s (low/high voltage). -computers

Many different ways to represent the same information using only 0s and 1s (binary).

Its easiest to build electronic circuits with two states,
logically called 1 and 0, **=> 1 bit**
physically often 3.3 and 0 volts.

Assume our box (memory) consists of one bit

We can use the bit to represent two different values

value	representation	
1	0	but only two numbers not useful
2	1	

Recall number vs. representation in last chapter

value	representation	
false	0	for Pascal's boolean variables
true	1	

What if box has two bits:

one combination has zero ones:	00
two have one one:	01, 10
one has two ones:	11

Since position matters can represent four values (or 2^2)

value	representation
east	00
north	01
west	10
south	11

Three bits can represent 8 (2^3) values: 000, 001, ..., 111

n bits can represent 2^n values:

n	can represent	about
8	256	
16	65,536	65 thousand (64K where K=1024)
32	4,294,967,296	4 billion
64	1.8446... x 10^19	20 billion billion

Most computers today use:

type	bits	name for box size
characters	8 | 16	byte (ASCII) | 16b Unicode (e.g., Java)
integers	32	word (sometimes 16 or 64 bits)
reals	32 | 64	word | double-word

Integer Representations

Why do we have four (popular) different representations for integers?
Each has it's own advantages and disadvantages.

Assume our box has a **fixed number** of bits n (e.g., 32).

We have two problems.

(1) Which 4 billion integers do we want? Remember there are an **infinite** number of integers less than zero and an **infinite** number greater than zero.

Today (1) is answered with either
(a) non-negative integers: zero & first positive integers
(b) positive and negative integers: zero about half negative & half positive

(2) What bit patterns should we select to represent each integer from (1)? Recall representation does not affect the calculated result , but it can affect its ease of use.

Since we'll convert to decimal before showing numbers to humans, select representation for computation ease, not intuition.

Today (2):
unsigned for (a)
signed magnitude for (b)
one's complement for (b)
two's complement for (b)
biased for (b)

Today unsigned and two's complement most common.

Unsigned Binary Integer Representation

== "Base 2 Number System" (Chapter 3)

The range of values indicates which integers can be represented for a given number of bits.
In each example, *n* represents the number of bits available to store the value.

The range of values for an unsigned binary number with *n* bits is: `0` to 2^n-1

4 bit Unsigned Binary example

Number of bits: `4`
Range: `0` to 2^4-1, OR `0` to `+15`

binary	decimal	hexidecimal		binary	decimal	hexidecimal
0000	0	0		1000	8	8
0001	1	1		1001	9	9
0010	2	2		1010	10	a
0011	3	3		1011	11	b
0100	4	4		1100	12	c
0101	5	5		1101	13	d
0110	6	6		1110	14	e
0111	7	7		1111	15	f

Notice: Only **positive** numbers can be represented.

Why can't the '-' character be used for storing negative values?
Computers can only recognize a value of `0` or a `1` in each bit.
An additional symbol like '-' can not be added.

Sign-magnitude Integer Representation

Sign-magnitude reserves one bit to store the sign (0 means positive and 1 means negative).

Good: represent negative numbers using only 0s and 1s

Bad: reduces the range of positive values that can be represented with the same number of bits. *Note: the most significant bit is being used to represent negative values.*

sign bit- bit that stores the sign of the number

convention: store the sign bit as the left-most bit of the number.

- 0 for positive
- 1 for negative

The range of values for a sign-magnitude number with *n* bits is: $^{-}(2^{n-1})+1$ to $^{+}(2^{n-1})-1$

4 bit Sign-Magnitude example

Number of bits: 4
Range: $^{-}(2^{(4-1)})+1$ to $^{+}(2^{(4-1)})-1$, which is $^{-}7$ to $^{+}7$

Sign-Mag	decimal	Sign-Mag	decimal
0000	$^{+}0$	1000	$^{-}0$
0001	$^{+}1$	1001	$^{-}1$
0010	$^{+}2$	1010	$^{-}2$
0011	$^{+}3$	1011	$^{-}3$
0100	$^{+}4$	1100	$^{-}4$
0101	$^{+}5$	1101	$^{-}5$
0110	$^{+}6$	1110	$^{-}6$
0111	$^{+}7$	1111	$^{-}7$

Notice: There are two representations of *zero*, $^{+}0$ and $^{-}0$.

Is it good or bad to have two representations of zero?
It's bad.

Why is it bad?

- **For the hardware to calculate any arithmetic or logical operation on zero must give the correct answer regardless of which value of zero is used.**
- **More complex (costly) than a system that has only one representation of zero.**

example: 4 bits

0101 is 5

1101 is -5

8 bits		s	mag	
00100001	is	0	0100001	is 33
10100001	is	1	0100001	is -33

to get the additive inverse of a sign magnitude integer, just flip
`(not, invert, complement, negate) the sign bit.`

<u>One's Complement Integer Representation</u>

Importance:

- historical reasons (it used to be used, e.g. Cray)
- it is used to produce two's complement representations (what is currently used).

Positive integers = unsigned binary (and sign-magnitude for that matter).

Negative integers = complement of the number's positive representation.

The complement of `0` is `1`

the complement of `1` is `0`.

Complement of an *n*-bit number is produced by flipping each bit of the number. (ADDITIVE INVERSE, NEGATE, FLIP, INVERT or NOT).

4 bit One's Complement example

Number of bits: 4
Range: $^{-}(2^{(4-1)})+1$ to $^{+}(2^{(4-1)})-1$, which is $^{-}7$ to $^{+}7$

1s Comp	decimal	1s Comp	decimal
0000	$^{+}0$	1000	$^{-}7$
0001	$^{+}1$	1001	$^{-}6$
0010	$^{+}2$	1010	$^{-}5$
0011	$^{+}3$	1011	$^{-}4$
0100	$^{+}4$	1100	$^{-}3$
0101	$^{+}5$	1101	$^{-}2$
0110	$^{+}6$	1110	$^{-}1$
0111	$^{+}7$	1111	$^{-}0$

***Bad**:*

- still two representations of *zero*, $^{+}0$ and $^{-}0$, and they are the one's complement of each other (0000 and 1111).
- the range of values is the same for the same number of bits in sign-magnitude.

Translation:

- If the most significant bit is a 1, the number represents a negative value.
- To determine which negative value, take the one's complement to find the positive decimal magnitude (value).
- Example: Given 1100
  - ⇨ 0011 (flip the bits
  - ⇨ 3 (but the MSB is 1)
  - ⇨ -3

```
things to notice:
1. any negative number will have a 1 in the MSB.
2. there are 2 representations for 0, 00000 and 11111.
```

Two's Complement Integer Representation

Two's complement = one's complement

allows one additional negative value instead of having two representations of zero.

- If MSB of a number is a `0` => the number is positive and can be interpreted the same as an unsigned binary representation.
- If MSB is a `1` => the number is negative (use the additive inverse to determine its decimal value in decimal).
- Note: the bit pattern `1111` represents `-1` instead of `-0`.

The value of each ***negative*** representation is ***shifted by one*** in this way.
shifting allows

- one additional negative value
- leaves only one representation of zero.

Two's complement representation of a **positive** decimal number:

- use the unsigned binary representation for the decimal value
- ensure the decimal value is in the valid range for the given number of bits
- Otherwise, overflow will occur and the result will be a negative value.

Two's complement representation of a **negative** decimal number:

- take the additive inverse of the representation of its positive value
- *Example:*
  - two's complement representation of `-3`
  - take the additive inverse (the *two's* complement) of `0011`
  - which is `1101`.

take the positive value	0101 (+5)
take the 1's comp.	1010 (-5 in 1's comp)
add 1	+ 1

	1011 (-5 in 2's comp)

Ok, but how do I take the two's complement?
1. Take the one's complement. (Flip the bits)
2. Add one to the result.

Ok, but how do I add one?
It's just like decimal.

add two bits and carry to the next place
if the result is two bits.

$$0_2 + 0_2 = 0_2$$
$$0_2 + 1_2 = 1_2$$
$$1_2 + 0_2 = 1_2$$
$$1_2 + 1_2 = 10_2$$
$$1_2 + 1_2 + 1_2 = 11_2$$

4 bit Two's Complement example

Number of bits: 4
Range: $^{-}(2^{(4-1)})$ to $^{+}(2^{(4-1)})-1$, which is $^{-}8$ to $^{+}7$

2s Comp	decimal	2s Comp	decimal
0000	$^{+}0$	1000	$^{-}8$
0001	$^{+}1$	1001	$^{-}7$
0010	$^{+}2$	1010	$^{-}6$
0011	$^{+}3$	1011	$^{-}5$
0100	$^{+}4$	1100	$^{-}4$
0101	$^{+}5$	1101	$^{-}3$
0110	$^{+}6$	1110	$^{-}2$
0111	$^{+}7$	1111	$^{-}1$

With 32 bits:

[2147483648,..,-1,0,+1,..,2147483647] approx= +/- 2G
[2^31,..,-1,0,+1,..,(2^31 - 1)] = 2^31 + 1 + (2^31 - 1) = 2^32

- There is only one representation of *zero*, `+0`
- two's complement of zero is still zero
- Try taking the two's complement of zero to convince yourself.

What happens to the `1` that gets carried out of the most significant bit when I take the two's complement of zero?
Nothing, it gets carried out and the *n*-bit result is still correct.

> **What decimal value does the two's complement 1110011 represent?**
>
> ```
>    1110011 (in 2's comp)
>
>    0001100 (after taking the 1's complement)
> +        1
> ----------
>    0001101 (this value is 13,
>             therefore its additive inverse
>             1110011 is -13.)
> ```

```
Verify on your own that the 2's complement of
0001101 is 1110011.
```

The most significant bit in a one or two's complement number is <u>not</u> a sign bit. Why not?
A sign bit is one bit that indicates the sign of the number and nothing more.

MSB of a two's complement number is only part of the sign of the number.
Example:

> the sign of the two's complement number `1111101` is indicated by each of the five most significant bits.
>
> The value `1111101` is the same as `101` or `11101`.

a 3-bit example:

bit pattern:	100	101	110	111	000	001	010	011
1's comp:	-3	-2	-1	0	0	1	2	3
2's comp.:	-4	-3	-2	-1	0	1	2	3

A LITTLE BIT ON ADDING

we'll see how to really do this in the next chapter, but here's a brief overview.

its really just like we do for decimal!

0 + 0 = 0
1 + 0 = 1
1 + 1 = 2 which is 10 in binary, sum is 0 and carry the 1.
1 + 1 + 1 = 3 sum is 0, and carry a 1.

```
  a     0011
 +b    +0001
 --     -----
sum    0100
```

see truth table next

carry in	a	b	sum	carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Biased Integer Representation

Similar to unsigned binary

- Difference: it can represent negative numbers
- The range of values represented is skewed.
  - Instead of representing `0 to` 2^n-1
  - bias-representation represents the range of values from `-bias to` 2^n-1-`bias`.

4 bit Bias-7 example

Number of bits: `4`
Range: `-B` to 2^4-1-`B`, which is `-7` to 2^4-1-7
this evaluates to `-7` to `+8`

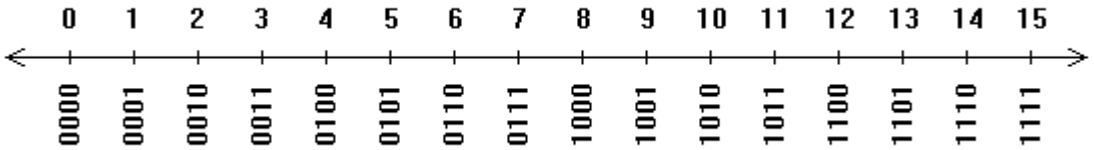
binary	decimal	Bias-7	binary	decimal	Bias-7
0000	0	-7	1000	8	1
0001	1	-6	1001	9	2
0010	2	-5	1010	10	3
0011	3	-4	1011	11	4
0100	4	-3	1100	12	5
0101	5	-2	1101	13	6
0110	6	-1	1110	14	7
0111	7	0	1111	15	8

Any integer can be used as the bias. The bias is chosen based on the range desired.

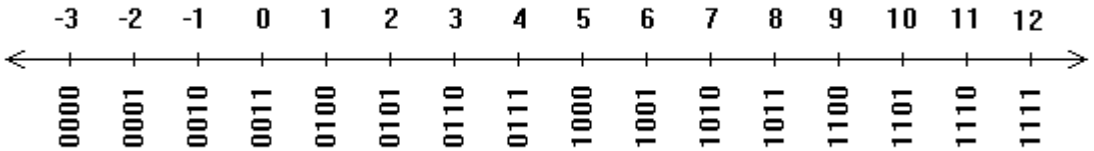
Bias-B
Binary Integer Representation

The range of values represented includes negative numbers
The bias is usually chosen to give an equal number of positive and negative values.

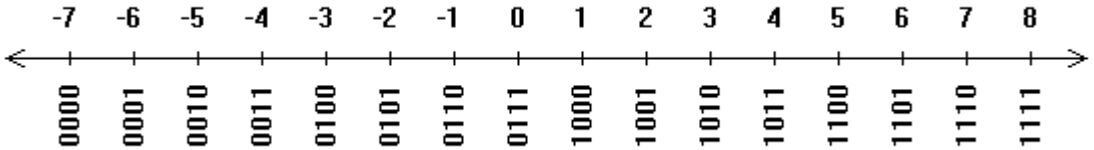
Bias-0



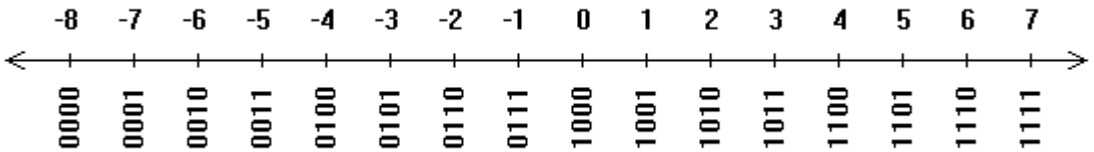
Bias-3



Bias-7



Bias-8



Notice: The bias of 7 was chosen so that approximately half the representable values were negative and the other half were positive.

```
visual example (of the re-mapping):

bit pattern:        000   001   010   011   100   101   110   111

unsigned value:      0     1     2     3     4     5     6     7

biased-2 value:     -2    -1    0     1     2     3     4     5

This is biased-2.  Note the dash character in the name
of this representation.  It is not a negative sign.
```

examples: given 4 bits, we BIAS values by 2^3 (8)

true value to be represented 3
add in the bias +8
---
unsigned value 11

so the bit pattern of 3 in biased-8 representation
will be 1011

---

going the other way, suppose we were given a
biased-8 representation as 0110

unsigned 0110 represents 6
subtract out the bias - 8
---
true value represented -2

ADD BIAS TO TABLE

this representation allows operations on the biased numbers
to be the same as for unsigned integers, but actually represents
both positive and negative values.

choosing a bias:
the bias chosen is most often based on the number of bits
available for representing an integer. To get an approx.
equal distribution of true values above and below 0,
the bias should be $2^{(n-1)}$ or $(2^{(n-1)}) - 1$

Used in floating-point exponents

Comparison of Integer Representations

Representation	Range	Addition	Additive Inverse
Unsigned	0 to 2^n-1	easy same as decimal	N/A, can't represent negatives
Sign-Magnitude	-(2^{n-1})+1 to +(2^{n-1})-1	complex	easy flip the sign bit
One's Complement	-(2^{n-1}) + 1 to +(2^{n-1}) - 1	medium have to consider both zeros	take additive inverse of each bit
Two's Complement	-(2^{n-1}) to +(2^{n-1}) - 1	easy	take one's complement and add one
Biased	-bias to +(2^n) - 1 - bias	<easy convert add convert	hard

How many distinct values can be represented with *n* bits?

2^n, regardless of the representation.

Why is Two's Complement used to represent integers, by most computers?

1. The hardware for the most common operations is faster.

2. The hardware is simpler (cheaper to design & build).

Use n=4 to illstrate,

	values				
in box	unsign	SM	1SC	2SC	Bias-8
0000	0	+0			-8
0001	1	+1			-7
0010	2	+2	non-neg		-6
0011	3		the same		
0100	4				
0101	5				
0110	6				
0111	7	+7			-1
1000	8	-0	-7	-8	0
1001	9	-1	-6	-7	+1
1010	10	-2	-5	-6	+2
1011	11				
1100	12				
1101	13				
1110	14				
1111	15	-7	-0	-1	+7

key values

0	0
2^(n-1) - 1	7
2^(n-1)	8
2^n - 1	15
2^n	16

(and corresponding negative values)

Sign-Extension

How do I change an integer with a smaller number of bits into the same integer (same representation) with a larger number of bits?
There is a different answer for each representation.

Arithmetic units must sometimes convert values held in one number of bits into the same value in a larger number of bits.

The process is different for each representation.

Unsigned

The original value goes in the least significant bits and 0's are placed in all more significant bits.

```
                          place 0s
                          ||||||||
          01001101   -->   0000000001001101
          11101001   -->   0000000011101001
                                   ^^^^^^^^
                                   original
                                   value
```

- only positive values and zero

```
             xxxxx   -->   yyyyyyyy
                           000xxxxx
copy the original integer into the LSBs, and put 0's
elsewhere
```

range: 0 to 2^n - 1, for n bits

ADD UNSIGN TO TABLE

example: 4 bits, values 0 to 15

```
n=4, 2^4 -1 is 15
[0, 15] = 16 = 2^4 different numbers

7 is 0111
17 not represenable
-3 not represenable
```

example: 32 bits = [0, 4,294,967,295]

```
4,294,967,296 = 2^32 different numbers
```

Sign-Magnitude

Copy the original integer's magnitude into the least significant bits, and put the original sign into the most significant bit. Place 0's everywhere else in the larger number.

```
                          place 0s
                          ||||||||
  01001101    -->    0000000001001101
  11101001    -->    1000000001101001
                     ^        ^^^^^^^
                  sign       original
                   bit       magnitude
```

```
sxxxx   -->   yyyyyyyy
              s00xxxxx
copy the original integer's magnitude into the LSBs,
put the original sign into the MSB, and put 0's elsewhere
```

IDEA: use 1 bit of integer to represent the sign of the integer

let sign bit be msb where
0 is +
1 is -

the rest of the integer is a magnitude, uses same encoding as unsigned integers

To get the additive inverse of a number, just flip (invert, complement) the sign bit.
range: $-(2^{(n-1)}) + 1$ to $2^{(n-1)} - 1$

ADD SM TO TABLE

4 bits, -7 to +7
n=4, $- 2^3 + 1$ to $2^3 - 1$
$-8 + 1$ to $8 - 1$

example: 4 bits
0101 is 5

-5 is represented as 1101
+12 not representable

[-7,..,-1,0,+1,..,+7] = 7 + 1 + 7 = 15 < 16 = 2^4 Why?

because of the sign bit, there are 2 representations for 0.
This is a problem for hardware. . .

0000 is +0, 1000 is -0

Since +0 equals -0, comparison logic can't just test for the same representation -- sounds trivial, but it's a big deal!

One's and Two's Complement

Copy the original integer into the least significant bits and copy the most significant bit of the original integer and copy it elsewhere. The most significant bit of the original value is shown in **bold** type-face.

1's and 2's complement: called **SIGN EXTENSION**

```
                        sign-extension
                           ||||||||
          01001101   -->   00000000**0**1001101
          11101001   -->   11111111**1**1101001
                                    ^^^^^^^^
                                    original
                                     value
```

.

copy the original integer into the LSBs,
take the MSB of original integer and copy it elsewhere.

```
      example:        0010101
                  000 0010101

                     11110000
              11111111 11110000
```

Bias

In most cases, the bias would change and other calculations would be required to make the change to a larger number of bits.

Overflow

Each representation in a given number of bits has a range of integers that can be represented. However, there are many integers that are greater (or smaller) than that range. If one of these integers is the result of a calculation, then *overflow* is said to have occurred.

Example: The result of adding the integers 0111 + 1101 causes overflow if the result (11100) must fit in 4-bits.

Overflow occurs when performing arithmetic operations.

sometimes a value cannot be represented in the limited number of bits allowed. Examples:

unsigned, 3 bits: 8 would require at least 4 bits (1000)
sign mag., 4 bits: 8 would require at least 5 bits (01000)

when a value cannot be represented in the number of bits allowed, we say that overflow has occurred. Overflow occurs when doing arithmetic operations.

example: 3 bit unsigned representation

```
  011 (3)
+ 110 (6)
---------
  ?  (9)     it would require 4 bits (1001) to represent
                    the value 9 in unsigned rep.
```

What happens on overflow?

- ignored
- tested
- trap

Character Representation

Characters are represented by 0s and 1s (***binary sequence***) in a computer.

ASCII Character Set- There is a standard 8-bit encoding for representing characters in binary. ASCII stands for American Standard for Character Information Interchange.

(See table on p.102 for the printable characters of the ASCII Character Set)

past: most I/O devices operated on 8-bit quantities

> Box (memory location) for a character usually contains 8 bits:
> 00000000 to 1111111 or in hex 0x00 to 0xff.

As with integers we ask

(1) Which characters?

(2) Which bit patterns for which characters?

For (1): A, B, C, ..., Z, a, b, c, ..., Z, 0, 1, 2, ..., 9
punctuation (,:{ ...) and special (\n \0 ...)

For (2): (a) Want STANDARD! and (b) want to help sorting
(i.e., representation(B) is between rep(A) and rep(C)).

I/O devices work with 8 bit (really only 7 bit) quantities.
A standard code ASCII (American Standard for Computer Information Interchange) defines what character is represented by each sequence.

now: 16-bit is more common. We will use 8-bit sequences in this course.

The order of bit patterns makes it possible to sort letters based on the integer values.

Uppercase letters have different bit patterns (values) than the same lowercase letter.

Also, notice that the bit pattern for the characters that represent digits are different than the encoding for decimal values.

For example, the character '3' is different than the integer value 3.

examples:
0100 0001 is 41 (hex) or 65 (decimal). It represents 'A'
0100 0010 is 42 (hex) or 66 (decimal). It represents 'B'

Different bit patterns are used for each different character that needs to be represented.

The code has some nice properties. If the bit patterns are compared, (pretending they represent integers), then

'A' < 'B'
65 < 66

This is good, because it helps with sorting things into alphabetical order.

Notes: 'a' (61 hex) is different than 'A' (41 hex)
'8' (38 hex) is different than the integer 8

the digits:
'0' is 48 (decimal) or 30 (hex)
'9' is 57 (decimal) or 39 (hex)

Quiz question: Why are there no character codes to represent: 10, 12 or 354?
Answer: Use 2 or 3 chars

Because of this, you have to be careful. Consider the following example:

Incorrect Program Example

```
in1:      .byte
result:   .byte

          get  in1
          add  result, in1, in1
          put  result
```

If the user had entered '3', result would be `51 + 51 = 102` (decimal)
The output would be 'f', since the ASCII code for 102 is 'f'

Correct Program Example

```
in1:      .byte
number:   .word
result:   .byte
out1:     .byte

          get  in1
          sub  number, in1, 0x30 # subtract ASCII
                                 # value for '0'
          add  result, number, number
          add  out1, result, 0x30 # covert decimal
                                 # to character
          put  out1
```

The subtraction takes the *bias* out of the character representation.
The addition puts the *bias* back into the ASCII encoding.

This will only work right if the result is a single digit.
(What would happen if it wasn't?)

What we need is an algorithm for translating character strings
to the integers the represent, and visa versa.

How do I convert a string of characters into an integer value?

We need an algorithm for the conversion.

```
Algorithm to convert a string of characters into an
integer:

    initialize integer to 0
    while (there are more characters) {
         get a character
         set digit to the character - 0x30
                                    (the value of '0')
         set integer to the value of
                              integer * base + digit
    }

ALGORITHM:   character string --> integer
   the steps:

      for '3' '5' '4'

      read '3'
      translate '3' to 3

      read '5'
      translate '5' to 5
      integer =  3 * 10  + 5 = 35

      read '4'
      translate '4' to 4
      integer =  35 * 10  + 4 = 354

  the algorithm:

     asciibias = 48
     integer = 0
     while there are more characters
       get character
       digit <-  character - asciibias
       integer <- integer * 10  + digit
```

How do I convert an integer into a string of characters?

We need an algorithm for the conversion.

Algorithm to convert an integer into a string of characters:

calculate the number of characters in the result value of the base
calculate the power = **base**$^{(\text{number of characters - 1})}$ e.g. $(10^2)=100$

```
while (integer > 0 ) {
     set digit to the integer / power
     set integer to integer mod power
     convert digit to it's ASCII code
     output the character
     set power to power / base
}
```

ALGORITHM: integer --> character string
the steps:
for 354, figure out how many characters there are (3)

354 div 100 gives 3
translate 3 to '3' and print it out
354 mod 100 gives 54
(100/10 = 10 for the power)

54 div 10 gives 5
translate 5 to '5' and print it out
54 mod 10 gives 4
(10/10 = 1 for the power)

4 div 1 gives 4
translate 4 to '4' and print it out
4 mod 1 gives 0
(1/10 = 0, so you're done)

written in a form using two loops:

```
    # figure out base^(number of characters - 1)
    power_of_base = base
    while power_of_base is not large enough
       power_of_base = power_of_base * base

    while power_of_base != 0
       digit = integer / power_of_base
           char_to_print = digit + 48
           print char_to_print
           integer = integer % power_of_base
           power_of_base = power_of_base / base
```

Compare:

```
mystring:        .asciiz   "123"
mynumber:        .word     123

"123" is '1'     0x31      0011 0001
         '2'     0x32      0011 0010
         '3'     0x33      0011 0011
         '\0'    0x0       0000 0000

         ==> 0011 0001 0011 0010 0011 0011 0000 0000
         Series of four ASCII characters

123 = 0x7b = 0x0000007b = 00 00 00 7b

         ==> 0000 0000 0000 0000 0000 0000 0111 1011
         a 32-bit 2SC integer
```

P.S. if you read "123" as .word it would be 825,373,440

(OPTIONAL) GO OVER FIG 4.7 (p. 103) (SAL codes for char/int conversion.)

Floating Point Representation

Box (memory location) for a real number usually contains 32 or 64 bits,
allowing 2^23 or 2^64 numbers.

As with integers and chars, we ask

(1) Which reals? There are an infinite number between two adjacent integers.
In fact, there are an infinite number between any two reals!!!!!!!

(2) Which bit patterns for reals selected for (1)?

Answer for both strongly related to scientific notation.

Computers represent real values in a form similar to that of scientific notation.

1.23×10^4

The number has a sign (positive in this case).
The significand (1.23) is written with one non-zero digit to the left of the decimal point.
The base (radix) is 10.
The exponent (an integer value) is 4. It also has a sign, positive.

There are standards which define what the representation means so that across computers there will be consistency. Note that this is not the only way to represent floating point numbers, it is just the IEEE standard way of doing it.

Consider: a x 10^b and show on number line, where
"a" has only one digit of precision.

a	b	a x 10^b
0	any	0
1 .. 9	0	1 .. 9
1 .. 9	1	10 .. 90
1 .. 9	2	100 .. 900
1 .. 9	-1	0.1 .. 0.9
1 .. 9	-2	0.01 .. 0.09

Many representable numbers close to zero where a small error is a big deal

Representable numbers spread out far from zero where a larger absolute error is still a small relative error

Let r be some real number and let fp(r) be the representable number closest to r, want

$$\left| \frac{fp(r) - r}{r} \right| < \text{small for all r (but zero)}$$

For above error maximum at r=1.5 | (1-1.5)/1.5 | = 1/3

If a can have five digits, worst relative error at 1.00005

| (1-1.00005)/1.00005 | approx= 0.00005

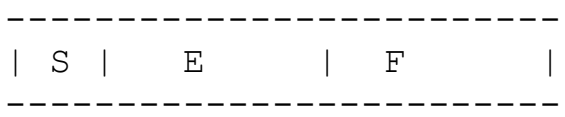
For (1): Minimize max relative error due to representation

For (2): (a) Want STANDARD!

Answer: Floating-point, especially IEEE FP

Here's what we do:

The representation has three fields:



`S` is one bit representing the sign of the number
`E` is an 8-bit biased integer representing the exponent
`F` is an unsigned integer

the value represented is:

$$(-1)^S \times f \times 2^e$$

where

$$e = E - bias$$
$$f = F/2^n + 1$$

for single precision representation (the emphasis in this class)

$$n = 23$$
$$bias = 127$$

What does all this mean?

- `S, E, F` all represent fields within a representation.
  Each is just a bunch of bits.

- `S` is just a sign bit. `0` for positive, `1` for negative.
  This is the sign of the number.

- `E` is an exponent field. The `E` field is a biased-127 representation.
  The true exponent represented is `(E - bias)`.
  The radix for the number is ALWAYS 2.

  Note: Computers that did not use this representation, like those built before the standard, did not always use a radix of 2.
  Example: some IBM machines had radix of 16.

- `F` is the mantissa (significand). It is in a somewhat modified form.
  There are 23 bits available for the mantissa.

- If floating point numbers are always stored in their normalized form, then the leading bit (the one on the left, or MSB) is ALWAYS a 1.
  So, why store it at all? It gets put back into the number (giving 24 bits of precision for the mantissa) for any calculation, but we only have to store 23 bits.

The MSB is called the HIDDEN BIT.

An example: Put the decimal number 64.2 into the IEEE standard single precision representation.

first step:

get a binary representation for 64.2
to do this, get binary reps. for the stuff to the left
and right of the decimal point separately.

64 is 1000000

.2 can be gotten using the algorithm:

.2 x 2 = 0.4 0
.4 x 2 = 0.8 0
.8 x 2 = 1.6 1
.6 x 2 = 1.2 1

.2 x 2 = 0.4 0 now this whole pattern (0011) repeats.
.4 x 2 = 0.8 0
.8 x 2 = 1.6 1
.6 x 2 = 1.2 1

so a binary representation for .2 is .001100110011. . .
---
or .0011 (The bar over the top shows which bits repeat.)

Putting the halves back together again:
64.2 is 1000000.0011001100110011. . .

second step:

Normalize the binary representation. (make it look like scientific notation)

$$1.000000\ 00110011\ldots \times 2^{6}$$

third step:

6 is the true exponent. For the standard form, it needs to be in biased-127 form.

```
   6
 + 127
 -----
  133
```

133 in 8-bit, unsigned representation is 1000 0101

This is the bit pattern used for E in the standard form.

fourth step:

the mantissa stored (F) is the stuff to the right of the radix point in the normalized form. We need 23 bits of it.

000000 00110011001100110

put it all together (and include the correct sign bit):

```
S    E          F
0  10000101  00000000110011001100110
```

the values are often given in hex, so here it is

0100 0010 1000 0000 0110 0110 0110 0110

0x	4	2	8	0	6	6	6	6

<u>Some extra details:</u>

--> Since floating point numbers are always stored in a normalized form, how do we represent 0?

We take the bit patterns 0x0000 0000 and 0x8000 0000 to represent the value 0.

(What fl. pt. numbers cannot be represented because of this?)

Note that the hardware that does arithmetic on floating point numbers must be constantly checking to see if it needs to use a hidden bit of a 1 or a hidden bit of 0 (for 0.0).

Values that are very close to 0.0, and would require the hidden bit to be a zero are called denormalized or subnormal numbers. These are specified

	S	E	F
0.0	0 or 1	00000000	00000000000000000000000
	(hidden bit is a 0)		
subnormal	0 or 1	00000000	not all zeros
	(hidden bit is a 0)		
normalized	0 or 1	> 0	any bit pattern
	(hidden bit is a 1)		

--> Other special values:

	S	E	F
+infinity	0	11111111	00000... (0x7f80 0000)
-infinity	1	11111111	00000... (0xff80 0000)
NaN (Not a Number)	?	11111111	?????...

(S is either 0 or 1, E=0xff, and F is anything but all zeros)

--> Single precision representation is 32 bits.
Double precision representation is 64 bits.
For double precision:
S is the sign bit (same as for single precision).

E is an 11-bit, biased-1023 integer for the exponent.

F is a 52-bit mantissa, using same method as single precision (hidden bit is not expicit in the representation).

One last example:

0x4228 0000 is stored

0100 0010 0010 1000 0 ...

0 | 1000 0100 | 0101 0000 ...

positive

e = E - 127 = E - 128 + 1 = E - 10000000 + 1 =5

f = F/2^23 + 1 = 0.01010000 + 1 = 1.01010000

+1.01010000 x 2^(+5) = 101010.000 = 32 + 8 + 2 = 42

Important Ideas

n-bit box can represent 2^n things
choose represention that eases computation
integers: UNSIGNED and 2SC most common
character: ASCII
real numbers: IEEE FP

IEEE Instruction Page

Convert floating point decimal number to IEEE representation,

SEF single point precision.

1. Calculate S (or set sign bit).

S = 0, if number is positive.

S = 1, if number is negative

2. Convert unsigned decimal number to unsigned binary notation

a. Convert integer part to unsigned binary

b. Convert fractional part to binary

c. Put both parts together

3. Convert the unsigned binary into Scientific Notation and normalize.

4. Calculate E (8 bits). It's the Bias-127 value of the exponent.

E = e + 127.

5. Convert E to unsigned binary

6. Calculate (or set) F (23 bits). It is the 23 most significant bits of the fractional part calculated in Step 3.

7. Place S, E and F together into one 32-bit single precision IEEE representation.