

DATA REPRESENTATION

POSITIONAL NUMBER SYSTEMS

- Positional numbers can use any number as a base
- Given a base, b , distinct symbols must be used to represent the numbers $0, 1, \dots, b-1$

For base 10 we use the 10 symbols
 $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$

For binary (base 2) we use the 2 symbols 0 and 1

For octal (base 8) we use the 8 symbols $0, 1, 2, 3, 4, 5, 6, 7$

For hex (base 16) we use the 16 symbols $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$

No reason that we can't also use base 7 or 19 but they're obviously not very useful

- Numbers are deciphered from right to left

Number positions by assigning 0 to the rightmost position in the number and increasing the number of the position by 1 as you number the positions from right to left

The character at position k determines how many b^k (b raised to the k -th power) units are present

Recall that b^0 is 1 for all b

The restriction is that only the symbols for $0, \dots, b-1$ may appear in each position of the number

- For example, in base 16, $347_{16} = 768 + 48 + 7 = 839_{10}$

3	4	7
$3 * 16^2 = 3 * 256$	$4 * 16^1 = 4 * 16$	$7 * 16^0$
768	48	7

- In base 8, $347_8 = 192 + 32 + 7 = 231_{10}$

3	4	7
$3 * 8^2 = 3 * 64$	$4 * 8^1 = 4 * 8$	$7 * 8^0$
192	32	7

- In base 2, 347 is illegal. Why?
- In base 10, 347 is of course

$$= 3 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0$$

$$= 300 + 40 + 7$$

$$= 347$$

CONVERTING FROM ONE BASE TO ANOTHER

- Any non-negative number can be written in any base

Since most humans are used to the decimal system and most computers use the binary system it is important for people who work with computers to understand how to convert between binary and decimal

- The following ideas work for conversions from any base to any other

Let's assume that there is one base that you are most familiar with and in which you know how to do the standard arithmetical operations +, -, * and /

Let's call this base the home base (b_h).

For most of use humans the home base will be 10

- In general, the easiest procedure for converting from base b_1 to base b_2 is to convert from b_1 to b_h , and then from b_h to b_2

In special cases, as we shall see below, you can convert directly from b_1 to b_2

CONVERTING FROM BASE b_1 TO BASE b_h

- First find what b_1 is equal to when written as a number in base b_h
- If the number to be converted looks like

$$D_k D_{(k-1)} D_{(k-2)} \dots D_3 D_2 D_1 D_0$$

- Form the sum (in base b_h)

$$D_k \cdot b_h^k + D_{(k-1)} \cdot b_h^{(k-1)} + \dots + D_1 \cdot b_h + D_0$$

- This is the process we used above to convert 347 from various bases into base 10
- The above process can be made more efficient by intermingling addition and multiplication as follows:

Start off with SUM = 0

At step j , multiply SUM by b_1 and add $D_{(k+1-j)}$

Continue this process until D_0 is added

- For example, to express 347 in hex proceed as follows
 - SUM = 0
 - SUM = 0*16 + 3 = 3
 - SUM = 3*16 + 4 = 52
 - SUM = 52*16 + 7 = 839
- This algorithm is the standard way that numeric conversions are performed on a computer.

However humans don't seem to find it so easy--maybe we're all brainwashed in elementary school

CONVERTING FROM BASE b_h TO BASE b_2

- This algorithm starts by finding the units digit of the number. It finds digits from right to left, so we can use a stack to put them back in left-to-right order
- Algorithm
 - Let num = Value(b_h)
 - do until num = 0
 - d = num % b_2 ; //Modulus
 - push d onto stack;
 - num = num - d; //Subtract
 - num = num \ b_2 ; //integer division
 - next;
 - Pop numbers off stack to get correct order

- For example, to convert 1000 to hex proceed as follows

			(Stack)
1000 mod 16	= 8	8	
1000 - 8	= 992		
992/16	= 62		
62 mod 16	= 14	8E	
(62 - 14)/16	= 3		
3 mod 16	= 3	8E3	
(3 - 3) / 16	= 0 STOP		

Pop 3, Pop E, Pop 8

$$1000 \text{ (base 10)} = 3E8 \text{ (base 16)}$$

- You can check your work by computing
 - $3*16^2 + E*16^1 + 8*16^0$
 - = $3*256 + 14*16 + 8$
 - = $768 + 224 + 8 = 1000$ (base 10) or
 - $3*16 + 14 = 62$
 - $62*16 + 8 = 992 + 8 = 1000$

CONVERTING BETWEEN DECIMAL AND BINARY

- The conversion between decimal and binary is particularly easy since binary has only two digits 1 and 0, and since multiplication by 2 is particularly simple
- Converting 1000 to binary proceeds as follows

1000 mod 2	= 0	(D ₀)
(1000 - 0)/2	= 500	
500 mod 2	= 0	(D ₁)
(500-0)/2	= 250	
250 mod 2	= 0	(D ₂)
(250-0)/2	= 125	
125 mod 2	= 1	(D ₃)
(125-1)/2	= 62	
62 mod 2	= 0	(D ₄)
(62-0)/2	= 31	
31 mod 2	= 1	(D ₅)
(31-1) mod 2	= 15	
15 mod 2	= 1	(D ₆)
(15-1)/2	= 7	
7 mod 2	= 1	(D ₇)
(7-1)/2	= 3	
3 mod 2	= 1	(D ₈)
(3-1)/2	= 1	
1 mod 2	= 1	(D ₉)
(1-1)/2	= 0 STOP	

- Thus, 1000 (base 10) = 1111101000 (base 2)
- People prefer hex to binary because (a) numbers are shorter and easier to read and (b) it is easy convert directly from hex to binary

Machines prefer binary because, with current technology, binary circuits are easier to design and build than circuits for other bases

If someone comes up with a more powerful technology in which it is easier to build circuits in some other base, people will probably switch to that base

- We always want to check our work so:

1111101000 =

1*2 + 1	= 3
3*2 + 1	= 7
7*2 + 1	= 15
15*2 + 1	= 31
31*2 + 0	= 62
62*2 + 1	= 125
125*2 + 0	= 250
250*2 + 0	= 500
500*2 + 0	= 1000

CONVERTING BETWEEN DECIMAL AND BINARY USING TABLES

- It is possible to use table to aid in the process of conversion.

Power	Decimal	Power	Decimal
0	1	13	8192
1	2	14	16384
2	4	15	32768
3	8	16	65536
4	16	17	131072
5	32	18	262144
6	64	19	524288
7	128	20	1048576
8	256	21	2097152
9	512	22	4194304
10	1024	23	8388608
11	2048	24	16777216
12	4096	25	33554432

CONVERTING BETWEEN BINARY AND HEX

- Converting between hex and binary is very easy since each group of 4 binary digits gives one hex digit and vice versa

Always form digits starting from the right

If the total number of digits is not a multiple of 4 pad with zeroes on the left

Use the following table:

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

- Thus converting 3E8 from hex to binary produces
11 1110 1000 = 1111101000 as above
 - Converting from binary to hex requires cutting up 1111101000 into 0011 1110 1000 = 3E8
 - To indicate that numbers are in hex, people often add an H or h to the end of the number
- NEGATIVE (SIGNED) NUMBERS**
- In regular human notation for numbers, people just affix a "-" to a number to represent its negation

But computers just have 0s and 1s

- You could do this in a computer by using the first bit of a number to represent + (0) and - (1)
- Computer designers do not use this scheme for three reasons
 - It permits +0 (00..0) and -0 (10..0)
 - This complicates CPU design since this must always be set equal, but are different bit patterns
 - The above might seem trivial to but try to design hardware to deal with it
 - It wastes a representation
- The most common representation that is used, **TWO'S COMPLEMENT**, permits a very elegant and efficient handling of mixed numerical computations

How do you handle $527 + (-289)$?

Note that subtraction becomes trivial when you have an easy way to find the additive inverse of a number. When the additive inverse is known a simple binary adder can also handle subtraction

The additive inverse of n is the number y such that $n + y = 0$

UNSIGNED BINARY NUMBERS

- These are just the unadorned binary numbers that you generally think of
- These numbers are **NON-NEGATIVE** and come in 8-bit, 16-bit and 32-bit versions

8-bit numbers range from 0 to 255 (0..FF)

16-bit numbers range from 0 to 65,535 (0..FFFF)

32-bit numbers range from 0 to 4,294,967,295 (0..FFFFFFFF)

- You add and multiply them just as you would expect

- To convert an 8-bit unsigned number to a 16-bit unsigned number simply add 8 zeroes to the front of it
- To convert a 16-bit unsigned number to a 32-bit unsigned number simply add 16 zeroes to the front of it

OVERFLOW

- What happens when the result of an operation is "too big?"
- Assuming 8-bit integers, what is 208+64?

$$\begin{array}{r}
 1110\ 0000 \\
 +\ 0100\ 0000 \\
 \hline
 1\ 0010\ 0000 = 272
 \end{array}$$

- But this is a 9-bit result
- This condition is called **OVERFLOW**
- All computers set some indicator after every arithmetic operation to determine if overflow has occurred
- In unsigned arithmetic, an overflow occurs if there is a carry or a borrow out of the most significant position
- Caution: In the language we will be studying the Overflow flag signals a different condition

The condition referred to above as "overflow" is signalled by the Carry flag in the 80x86

COMMON REPRESENTATIONS OF SIGNED BINARY NUMBERS

- There are many possible ways to represent negative numbers
- They all seem slightly "unnatural" because we learned arithmetic with signs by rote
- They are no more unnatural than the use of an extra symbol (-) to denote negation
- Note that we have many different representations for any number

Example: 15

15	15.0	3^*5	$\sqrt{225}$
7+8	11.1.363636...	$1*10^1+5*10^0$	$15*10^0$
$2^3+2^2+2^1+2^0$	$0F_{16}$	17_8	1111_2
fifteen	2^4-2^0	XV	0xf
1+2+3+4+5	IIII IIIII IIIII		

- Four Common Techniques:
 - a) Signed Magnitude
 - b) Bias
 - c) 1's complement
 - d) 2's complement
- Each technique has advantages and disadvantages

Criteria for choosing a good representation

- compute additive inverse easily
- minimize hardware complexity of:
 - addition
 - subtraction
 - comparison
- would like to use same circuitry for signed and unsigned arithmetic
- get maximum number of representations from a fixed number of bits

SIGNED MAGNITUDE

- Use first bit as sign, 0 is + 1 is -
- Remaining bits are treated as the unsigned magnitude

Example: 65 = 0100 0001
 -65 = 1100 0001

- Advantages:
 - Additive inverse is easy to compute

Disadvantages:

There are two distinct representations of 0

1000 0000 -0
 0000 0000 +0

Circuitry has to treat these as equal

Addition and subtraction are not easily implemented in circuitry. Different algorithm from unsigned operations.

Biased Representations

- Subtract a bias B from the value of the unsigned representation
- Example: Bias-128

1101 0110 = 214-128 = 86
 1111 1111 = 255-128 = 127
 0101 0001 = 81-128 = -47
 0000 0000 = 0-128 = 128
 1000 0000 = 128-128 = 0

- Note that bias can be chosen to allow skewed distribution of positive and negative numbers
 Bias-50 has range of -50 to 205

- **Advantages**
only one representation of 0
- **Disadvantages**
additive inverse not easy to compute
basic operations not easy to implement

One's Complement

- Simply invert each bit to obtain additive inverse
- **Examples**

0100 0001	=	65
1011 1110	=	-65
0111 1111	=	127
1000 0000	=	-127
0000 0000	=	0
1111 1111	=	0

- **Advantages**
Additive inverse is trivial
- **Disadvantages**
Two representations of 0
Different arithmetic algorithms for signed and unsigned operations

TWO'S COMPLEMENT

- Similar to one's complement
- Invert each bit in unsigned representation to obtain one's complement
Then add 1 to the result
- **Example: -57**

57 =	0011 1001	(unsigned)
	=	1100 0110 (one's complement)
	=	1100 0111 (two's complement)
- This appears to be a bizarre and arbitrary way to represent negative numbers
But two's complement has many useful properties
Is the most common way to represent signed integers
- In two's complement numbers have the following ranges
 - 8-bit numbers range from -128 to 127 (80h..7Fh)
 - 16-bit numbers range from -32768 to 32767 (8000h..7FFFh)
 - 32-bit numbers range from -2147483648 to 2147483647 (80000000..7FFFFFFFh)
- Note that the number of negative numbers is 1 greater than the number of positive numbers

TWO'S COMPLEMENT (Cont'd)

- To illustrate the concept let's restrict our attention to 8-bit numbers
- The non-negative numbers 0..127 are represented by their usual binary representation using 8-bits, which means that the leading bit is 0

The range 0..127 is represented by the numbers 0000 0000b to 0111 1111b which is equal to 0H to 07FH

- To represent a negative number, follow the following steps

Convert its negation to an 8-bit binary number in the usual manner

Take the representation computed in the previous step and complement every digit: this means replaces 0's by 1's and 1's by 0's

Add 1 to the result obtained in the previous step

- For example, to convert -128 to two's complement proceed as follows

Represent 128 in binary to get 1000 0000b

Complement the digits to get 0111 1111b

Add 1 to get 1000 0000b which is 080H

- For example, to convert -127 to two's complement proceed as follows

- Represent 127 in binary to get 0111 1111b

- Complement the digits to get 1000 0000b

- Add 1 to get 1000 0001b which is 081H

- If you convert the numbers -126, -125,..., -1, you will get the numbers

082H, 083H,...,0FFH

- Thus going from 0000 0000b to 1111 1111b is counting from 1 to 127, then jumping to -128 and counting by 1's until -1 is reached

- Note that the negative numbers all have a leading bit of 1

CONVERTING FROM TWO'S COMPLEMENT TO DECIMAL

- Suppose we are given a two's complement number N
- If the leading bit of N is 0, the number is non-negative and may be converted to decimal in the usual way
- If the leading bit of N is 1, convert it to decimal as follows:
 - Subtract 1 from the representation
 - Complement the digits
 - Convert to decimal in the usual way
 - Add a - to the front of the result

OR

For an n-bit number, compute the decimal value of the unsigned binary number composed of the least significant n-1 bits

Subtract 2^{n-1} from the result

- Examples:
 - Represent 0101 0110b in decimal
 - This may be done in the usual way

$$64 + 16 + 4 + 2 = 86$$

Represent 1010 1011b in decimal

Since the leading bit is 1, this is a negative number which must be converted as follows

Subtract 1 to get 1010 1010b

Complement the digits to get 0101 0101b

Converting as in the previous example give 85

Adding the minus sign gives -85

OR

$$010\ 1011b = 43$$

$$43 - 128 = -85$$

WHAT IS TWO'S COMPLEMENT REALLY?

- In order to understand some of the interesting properties of two's complement it is helpful to have an algebraic representation of what a two's complement number really is
- The two's complement of a non-negative number is just the number itself, so N is represented by N if $N \geq 0$

- If $N < 0$ we note the following facts for numbers of K bits

Complementing the bits of N is equivalent to computing $X + N$ where X is the K -bit number consisting of all 1's

The K -bit number consisting of all 1's is equal to $2^K - 1$

So, the K -bit binary number that results from complementing the bits of N is equal to $(2^K - 1) + N$ (recall that N is negative)

Adding 1 to the complemented bit pattern yields the number $2^K + N$

- Another algebraic representation is the following:

$$N = \sum_{i=0}^{k-2} b_i 2^i - b_{k-1} 2^{k-1}$$

- The following summarizes our discoveries with respect to two's complement numbers

With K bits, two's complement can represent the numbers from $-2^{(K-1)}$ to $2^{(K-1)} - 1$

Any number in the range 0 to $2^{(K-1)} - 1$ is represented using its standard binary K -bit pattern

Any number, N , in the range $-2^{(K-1)}$ to -1 is represented using the standard binary K -bit pattern of $2^K + N$

- The above observations can be summarized in the following sentence:

the K -bit 2's complement representation of N , where N is in the range $-2^{(K-1)}$ to $2^{(K-1)} - 1$, is the K -bit binary representation of N if $N \geq 0$

the K -bit binary representation of $2^K + N$ if $N < 0$

WHAT'S SO SPECIAL ABOUT TWO'S COMPLEMENT?

- Two's complement has many interesting and useful properties
- Two's complement has only one 0 and avoids the ± 0 problem
- Adding two's complement numbers can be done using **EXACTLY THE SAME ALGORITHM AS ADDING** unsigned numbers

there is no need to use special cases based on whether numbers are positive or negative

TWO'S COMPLEMENT OVERFLOW

- When is an operation on 2's complement numbers invalid?

Consider the following, assuming 8 bit numbers:

$$-1 + (-1) = ?$$

$$-128 + (-1) = ?$$

$$127 + 1 = ?$$

$$1 + 1 = ?$$

- Consider these simple rules for determining if there has been an overflow in two's complement arithmetic:
 - A) Overflow cannot occur when adding two numbers with different signs (Why?)
 - B) If the two numbers have the same sign, overflow occurs when there is a "sign change" in the result

EXTENDING TWO'S COMPLEMENT NUMBERS

- Non-negative K-bit, 2's complement values can be turned into non-negative (K+D)-bit 2's complement values just by prefixing D zeroes to the number
- Negative K-bit, 2's complement values can be turned into negative (K+D)-bit 2's complement values just by prefixing D ones to the number
- For example, 0100 is the 4-bit, 2's complement representation of 4 and 1100 is the 4-bit, 2's complement representation of -4
- 000 0100 is the 7-bit, 2's complement representation of 4 and 111 1100 is the 7-bit, 2's complement representation of -4
- The process of creating a larger 2's complement representation from a smaller one is called SIGN EXTENSION

This sign bit is simply replicated to the left

Character Representations

- Characters are represented as arbitrary bit strings.
 - ASCII (American Standard Code for Information Interchange)
 - Developed for teletype devices before computers
 - A 7 bit code
 - Chars 0 -31 are "control characters" intended to control devices

EBCDIC (Extended Binary Coded Decimal Interchange Code)

8-bit character set used only on IBM mainframes

- ASCII allows 128 characters and EBCDIC 256
These are not sufficient for even European languages without even considering Arabic, Hebrew, Korean, Japanese, Mandarin, ...
- ASCII was extended to an 8-bit character set called ANSI to accommodate some European Vowels

A kludge under the name of "code pages" allowed extended character sets to implemented in 8-bit characters

Unicode

- Unicode is a system designed to transcend character encodings - it is not simply an expansion of ASCII code
- To understand Unicode you must also understand UCS (Universal Character Set) or ISO 10646
- UCS is like a giant alphabet (32 bits) designed to encode any human character known
And some that aren't human – it includes a "private use area" that has been used for Klingon characters among other things
- Unicode provides algorithms for encoding UCS characters

See <http://www.unicode.org/standard/WhatIsUnicode.html> for the basics:

"Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. The Unicode Standard has been adopted by such industry leaders as Apple, HP, IBM, JustSystems, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many others. Unicode is required by modern standards such as XML, Java, ECMAScript (JavaScript), LDAP, CORBA 3.0, WML, etc., and is the official way to implement ISO/IEC 10646. It is supported in many operating systems, all modern browsers, and many other products. The emergence of the Unicode Standard, and the availability of tools supporting it, are among the most significant recent global software technology trends."

- Recommended Reading for Unicode
An excellent and concise introduction:

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

At

<http://www.joelonsoftware.com/articles/Unicode.html>

- Unicode can be implemented with different character encodings
Most common are UTF-7, UTF-8, UTF-16 (UCS-2), and UTF 32
These are variable length encodings
- Unicode has been adopted by many modern languages and nearly all popular operating systems

Java, XML, .NET framework, Python, Ruby etc.

- For a Unicode tutorial, see <http://www.unicode.org/notes/tn23/tn23-1.html>
- UTF-7 and UTF-8 are ways of encoding unicode characters in variable-length bit strings

The great advantage of these encodings is that ASCII chars are encoded in single bytes and look just like ASCII

In UCS-2 every character is 16-bits wide and in UCS-4 they are 32 bits wide

ASCII IN HEX

- Note patterns not visible in decimal version of table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL							BE L	BS	TA B	LF		FF	CR		
1								ES C								
2	(space)	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	(del)

- Constant difference between upper and lower case alphabets = 20 h

A = 41h = 0100 0001

a = 61h = 0110 0001

Z = 5Ah = 0101 1010

z = 7Ah = 0111 1010

This difference is one bit

- Constant difference between ASCII digits and binary values = 30 h

"0" = 30h = 0110 0000

"1" = 31h = 0110 0001

...

"9" = 39h = 0110 1001

- Low-order 4 bits has binary value of digit

Converting Case

- Constant difference between upper and lower case alphabets = 20 h

A = 41h = 0100 0001

a = 61h = 0110 0001

- We could convert lower case to upper case with subtraction:

```
mov al, char
```

```
sub al, 20h
```

- We can also mask with a Boolean operator:

```
mov al, char
```

```
and al, 0DFh ; DF = 1101 1111
```

A bitwise AND results in 1 only for inputs 1 and 1
AND masks are used to force bits to 0
Place a 0 in the position you wish to change

```
mov al, char
OR al, 20h ; 20 = 0010 0000
```

A bitwise OR results in 0 only for inputs 0 and 0
This code converts to lowercase by setting bit 5.
OR masks force bits to 1.

- Same principles can be used to convert binary bytes to ASCII digits and vice-versa

What are the appropriate masks

What's the Difference?

- Note that masking with Boolean operators eliminates the need to test for lower or upper case before performing the operation

Using ADD and SUB requires a test before performing the conversion

- The Boolean operations produce branch-free code that can be efficiently executed in pipelined processors

Control Characters

- ASCII was originally designed to control teletype devices

Common Control Characters		
NUL	00	Null character
BEL	07	Bell character
BS	08	Backspace
TAB	09	Tab character
LF	0A	Line Feed
FF	0C	Form Feed
CR	0D	Carriage Return
ESC	1B	Escape

- Not shown are characters such as ACK, etc.
- Note that the newline character \n is Carriage Return + Line Feed in Microsoft operating systems (0D 0A)

A text file containing

ABC

DEF

appears as

41 42 43 0D 0A 44 45 46

in a hex editor

NUMBERS ON THE x86 Processors

- The x86 has instructions that can operate on the following different types of numbers

- Unsigned binary numbers
- Signed binary numbers
- Unpacked binary coded decimal (BCD) numbers
- Packed BCD numbers

- Note that the hardware does not distinguish one type of number from another. It is up to the programmer to apply instructions that are appropriate to the numbers being manipulated.

NUMBER BASE CONVENTIONS

- Since we will have occasion to use binary, decimal and hex numbers mixed in with one another we will adopt the following conventions

- If a number is written without any indication as to its base, it is assumed to be **BASE 10**

- 65535 is in base 10

- If a number has a **b** or **B** immediately next to its last digit, it is in **BINARY**

- 101001b is in binary as is 11101B

- If a number has an **h** or **H** immediately next to its last digit, it is in **HEX**

- 346h is in hex as is 3E8H

- If a number begins with a leading 0, it is in **HEX**

- 0346 and 03E8 are both in hex

- Hex numbers often begin with a leading 0 since numbers like A3 look like variables whereas 0A3 can only be a number
- Assembly languages have slightly different conventions for parsing numbers. The A86 assembly language follows the convention that numbers written with a leading 0 are in hex and otherwise they are in decimal
- It is generally a good idea to avoid ambiguity by using the H suffix whenever you use a hexadecimal number
- A86 also recognizes the following representations for different bases if they follow immediately after the last digit:

- H for hex

- O (oh) or Q for octal

- D or xD for decimal

- x is used in cases where D might be interpreted as a hex digit

B or xB for binary

x is used in cases where B might be interpreted as a hex digit

- **A86 is case insensitive so the preceding suffixes can be either upper or lower case**

BINARY CODED DECIMAL (BCD) NUMBERS

- **The 8086 can operate with the strings formed from the digits '0', ..., '9' representing numbers**
- **The easiest variation of this is using one byte to represent one digit**

Such numbers are called UNPACKED BCD NUMBERS

This is wasteful of space since a byte can represent 256 values instead of just 10

- **The 8086 can also operate with PACKED BCD NUMBERS which use one byte to represent the values 0..99 in decimal**

Unpacked BCD is more completely implemented than packed BCD

- **Programmers tend not to use BCD because it is MUCH SLOWER AND MORE WASTEFUL OF SPACE than the ordinary binary representation**

BCD is generally used only with devices that perform very simple arithmetic only--POS Terminals, etc.

BCD has certain advantages with respect to precision in calculating monetary amounts--example \$0.20 has no exact binary representation

- **Note that the digits '0'..'9' have ASCII codes 48..57, which in hex gives the numbers 030..039**

FRACTIONS AND DECIMALS

- **So far we have only dealt with integers and have not talked about representing fractions or decimals.**

In some cases, even though quantities are written as fractions or decimals, you would tend to use integers when working with them

For example, many financial calculations are most likely done using integers

and carried out in pennies.

Alternatively, because of the decline of the value of the dollar, some calculations (your IRS form) may be done in dollars.

FIXED POINT ARITHMETIC

- You can treat decimals as integers by changing the units that you work in.

For example, if you are working with lengths such as 2.354 meters or 2 feet 6 inches, you can adopt millimeters or inches as your basic units.

In the above examples, you would be working with 2354 mm or 30 inches.

In general, you can treat integers as having a decimal point in some location.

For example, if you are working with monetary amounts you might assume that all amounts have a decimal point to the right of the last two digits. Thus, 1234 would actually be \$12.34.

- This technique is called **FIXED POINT** arithmetic and is now rarely used.

It requires the programmer to keep track of numerical quantities and to perform the correct adjustments after each fixed point operation.

Consider 10 items (10.00) sold at \$20 each (20.00)
 $1000 * 2000 = 2,000,000 = \$20,000!$

- **FLOATING POINT** fixes many of the problems of fixed point arithmetic, but introduces some difficulties of its own.

FLOATING POINT NUMBERS

- A floating point number consists of two parts:

An **EXPONENT** (also called a **CHARACTERISTIC**)

A **FRACTION** (also called a **SIGNIFICAND** or **MANTISSA**)

Some example numbers are:

0.576×10^5

0.11234×10^{-5}

0.23E-4

0.2345E12

- Note the use of both powers of 10 and E, which means powers of 10.
- Note also the use of the 0 before the decimal point.

There exist different conventions for floating point numbers.

The above numbers could also be written as

5.76 x 10⁴

1.1234 x 10⁻⁶

2.3E-5

0.02345E13

- The non-unique representation of floating point numbers is important for addition.
- What is the rule for converting among the different forms of a floating point number?

For example, how would you add 0.576E4 and 0.23E2?

- Answer: convert them to the same exponent and add. Thus, we get

$$\begin{aligned} & 0.576E4 + 0.23E2 \\ &= 57.6E2 + 0.23E2 \\ &= 57.83E2 \\ &= .5783E4. \end{aligned}$$

- A point to note is that the fraction of one of the original numbers consisted of 3 digits following the decimal point, while the fraction of the other number consisted of 2 digits, yet the answer has 4 digits. Is this reasonable?

NORMALIZING FLOATING POINT NUMBERS

- To make floating point useful on a computer, we must adopt some conventions.

It seems wasteful to actually represent a decimal point in a number.

Consequently, people always assume a fixed location of the decimal point.

- In some floating point conventions used by humans, commonly called **SCIENTIFIC NOTATION**, the decimal point comes immediately after the first non-zero digit.

In scientific notation, 1.2E4 would tend to be the preferred form rather than .12E5.

When forms such as 1.2E4 are chosen, it makes more sense to call the two parts of the number the exponent and the significand.

- Computers are happy with whatever form is chosen. The standard form chosen is called the **NORMALIZED FORM**.

As you can see from the simple addition example above, it is necessary to convert floating point numbers to different forms when performing certain operations.

Converting to the normalized form is called **NORMALIZING**.

- Typically, floating point numbers have limits on the number of digits that can be used in both the exponent and the fraction.

ROUNDING OFF

- As noted above, arithmetical operations tend to increase the number of digits present in the fraction and can possibly increase the number of digits in the exponent.
- When a computer performs floating point operations, it might temporarily permit a larger number of digits for the intermediate result, but it eventually must normalize the number so it will often have to reduce the number of digits.
- The process of reducing the number of digits is called **ROUNDING**.
- If you round a number, you typically get a different number so the rounded number is only an approximation to the original number.

The difference between the rounded form and the original number is called the **ROUND OFF ERROR**.

- As you perform floating point calculations, roundoff errors can combine and make the final result meaningless.

Programmers and mathematicians run into trouble when they ignore roundoff error and its future development.

TECHNIQUES FOR ROUNDING

- There are two common techniques that are taught in school for rounding off.

One technique is to simply ignore all digits past a certain point.

This technique is generally referred to as **TRUNCATION**, yet in our more

general sense it is a rounding off operation.

TRUNCATION is not generally used for the normalization of floating point numbers since it has worse error properties than the next technique.

- You probably learned the following rule for rounding off in a particular position of a decimal number.

If the following digit is 4 or less, leave the digit to be rounded unchanged.

If the following digit is 5 or more, add 1 to the digit to be rounded.

For example, if we round to the nearest hundredth:

3.4567 rounds to 3.46

3.4545 rounds to 3.45

3.4550 rounds to 3.46

There is one problem with the above rule: its slight BIAS toward the larger number.

- For example, 3.4550 is just as close to 3.45 as it is toward 3.46, so why should it always be 3.46?
- In fact, people have decided that the above rules should be modified so that quantities that look like ..5000... should round up half of the time and down the other half of the time.
- A better rule is that if the digits following the digit to be rounded look like 50000..., then you round up if the preceding digit is odd and you round down if the preceding digit is even.

The above rule can be summarized as ROUND 50000... toward the even number.

- One advantage of this rule is that in a series of additions the errors tend to cancel.

For example, suppose we want to round the following numbers to integers and then add them: 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5

Doing it using the original rule yields the sum 4, 5, 6, 7, 8, 9, 10 which sum to 49. Note that the sum of the original numbers is 45.5, so we get a difference of 3.5.

If we round using the new rule we get 4, 4, 6, 6, 8, 8, 10 which sum to 46, which differs by only .5 from the correct sum.

If we had taken an even number of consecutive original numbers the sum of the rounded numbers would equal the sum of the original numbers.

- The previous example illustrates the PROPAGATION OF ERROR. In other words, errors can accumulate as a result of using rounded numbers.

Notice that if we truncate the sequence, we get 3, 4, 5, 6, 7, 8 and 9 which sum to 42, which differs by 3.5 from the correct sum.

- In general, truncation has worse error problems than the other rounding techniques described above. This is why truncation is not generally thought of as a rounding technique.

INTRODUCTION TO FLOATING POINT ARITHMETIC

- Some arithmetical operations make sense for floating point numbers and others don't.

$+$, $-$, $*$ and $/$ make sense for floating point numbers.

Integer division and Mod do not make sense for floating point numbers.

Floating point arithmetic can be handled in three common ways:

Subroutines (8088, 8086, 80286, 80386)

Special external hardware or coprocessor (8087, 80287, 80387, etc.)

Built-in hardware (80486DX, Pentium)

(The 80486 DX basically has a coprocessor built into the chip; the 80486 SX does not have one.)

- We will discuss $+$, $-$, $*$, and $/$ in more detail.
- The field of NUMERICAL ANALYSIS is primarily concerned with the correct use of floating point numbers for different types of computation

In this course, there is only enough time to show you some of the problems and suggest some of the answers.

If you will do extensive floating point work, you should that you take a numerical analysis course and learn this subject in great detail.

FLOATING POINT ADDITION AND SUBTRACTION

- Note that adding a negative number is like subtracting a positive number.
- While people generally think of addition and subtraction as simpler operations than multiplication and division, in some sense they are more difficult and create more problems than multiplication and division.
- Let's consider the following simple example.
We have a floating point representation that permits us to keep 3 digits in the fraction and 2 digits in the exponent.

We wish to test the associative law of addition:

$$A + (B + C) = (A + B) + C$$

Let $A = 0.1E5$, $B = -0.1E5$ and $C = 0.1E1$.

In ordinary notation, $A = 10,000$, $B = -10,000$ and $C = 1$.

To compute $A + (B + C)$ we must first compute $B + C$.

In ordinary arithmetic, this would give -9999 .

Since numbers are normalized after addition, and since we can keep only 3 digits in the fraction, the correct answer $-0.9999E4$ becomes $-0.1E5$. Now computing $A + -0.1E5$ produces 0!

Computing $(A + B) + C$ first produces $0.0E0$ and then gives $0.1E1$!

One way yields 0 and the other yields 1. Clearly, the associative law does not hold.

FLOATING POINT MULTIPLICATION AND DIVISION

- What problems can happen with floating point multiplication and division?

If our exponents are limited to 2 digits, consider the following.

$0.1E-99 * (0.1E99 * 0.1E2)$ does not return a value since we get an OVERFLOW ERROR

What does $(0.1E-99 * 0.1E99) * 0.1E2$ return?

- What about $A * (B * C) = (A * B) * C$?
- What about division?

BINARY FLOATING POINT NUMBERS

- Many different conventions have been adopted for representing floating points numbers in a computer.
- The IEEE has standardized four formats of floating point numbers (after a LONG debate of 10-15 years)

IEEE STANDARDS FOR FLOATING POINT NUMBERS

- These are standards set up by the professional organization called the IEEE (Institute of Electrical and Electronic Engineers).

These standards are now very widely used.

FOUR IEEE FLOATING POINT FORMATS

- Single-precision 32-bit format.
- Double-precision 64-bit format.
- Single extended 44(or more)-bit format to be used for intermediate results.
- Double extended 80(or more)-bit format to be used for intermediate results.
- The first two formats are more uniform from machine to machine while the last two might vary.

THE BINAMAL SYSTEM

- In the decimal system, digits to the right of the decimal point represent negative powers of 10.
- For example, .645 represents

$$\begin{aligned}
 &= 6 \cdot 10^{-1} + 4 \cdot 10^{-2} + 5 \cdot 10^{-3} \\
 &= 6/10 + 4/100 + 5/1000 \\
 &= 645/1000.
 \end{aligned}$$

- Similarly we can have a BINAMAL POINT and bits to the right of this point represent negative powers of 2.

- For example, 110.101 in binary =

$$\begin{aligned}
 &= 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\
 &= 4 + 1 + 1/2 + 1/8 \\
 &= 5 + 5/8.
 \end{aligned}$$

- Just as some fractions such as 1/3 have an infinite decimal expansion (.3333...), some fractions have an infinite binamal expansion.

CONVERTING FROM DECIMAL TO BINAMAL

- An easy algorithm
 - 1) Compute the binary representation of the integer part
 - 2) Take the decimal part and proceed as follows:
 - a) Double it
 - b) If result < 1, write down a 0
 - c) If result >= 1, write down a 1 and subtract 1 from the doubled decimal
 - d) Repeat until:
 - Remainder is 0
 - OR - A repeating pattern appears
 - OR - desired precision is reached

- Example: 17.3125 17 = 0001 0001

.3125 * 2	=	.625	0
.6250 * 2	=	1.25	1
.2500 * 2	=	.5	0
.5000 * 2	=	1.0	1

 17.3125 = 0001 0001.0101

- Example: 2.4 2 = 0010

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

 2.4 = 0001.0110 0110 0110

- Example: 12.37

12 = 0110		Remainder .37
.37 * 2	=	.74 0
.74 * 2	=	1.48 1
.48 * 2	=	.96 0
.96 * 2	=	1.92 1
.92 * 2	=	1.84 1
.84 * 2	=	1.68 1
.68 * 2	=	1.36 1
.36 * 2	=	.72 0
.72 * 2	=	1.44 1
.44 * 2	=	.88 0
.88 * 2	=	1.76 1
.76 * 2	=	1.52 1
.52 * 2	=	1.04 1
.04 * 2	=	.08 0
.08 * 2	=	.16 0
.16 * 2	=	.32 0
.32 * 2	=	.64 0

.64 * 2	=	1.28	1
.28 * 2	=	.56	0
.56 * 2	=	1.12	1
.12 * 2	=	.24	0
.24 * 2	=	.48	0

- Note that the lack of an exact binamal representation for many exact decimals is the motivation for the use of BCD or fixed-point in some applications (particularly those involving money)

THE IEEE, 32-BIT, SINGLE-PRECISION FORMAT

- Uses binamal.
- Because there are only two digits in binary, both formats place a binary 1 in front of the binamal point since other than 0, every binamal will start with a 1.
- This provides an extra bit of precision and is sometimes referred to as a HIDDEN BIT.

Sign Bit	Bias-127 Exponent E 8-bits	Unsigned Fraction F 23 bits
----------	----------------------------	-----------------------------

$$\text{Val} = (-1)^{\text{Sign}} * 1.F * 2^{(E-127)} \text{ if } E \neq 0$$

THE IEEE, 64-BIT, DOUBLE-PRECISION FORMAT

Sign Bit	Bias-1023 Exponent E 11-bits	Unsigned Fraction F 52 bits
----------	------------------------------	-----------------------------

$$\text{Val} = (-1)^{\text{Sign}} * 1.F * 2^{(E-1023)} \text{ if } E \neq 0$$

IEEE FORMATS

- Use a BIAS for the exponent to represent both positive and negative exponents.
- Biases are not as large as possible.
- With 8-bits you could have an exponent range of -128 to +127, but the single-precision format uses only the range -127 to +127.
- This is because +128 (corresponding to E = 255) is reserved for special values.
- One such value S = 0 or 1, E = 255 and F = 0 is $\pm\infty$ which is used to represent very large quantities, and typically is the result returned when division of a non-zero quantity by 0 is attempted.

The values S = 0 or 1, E = 255 and F \neq 0 is called NAN, which is short for NOT A

NUMBER.

- NAN arises when an operation is attempted whose outcome is completely undetermined such as dividing 0 by 0.

It is important to understand the difference between ∞ and NAN.

- If you divide a non-zero quantity by 0 or things close to 0, you will get a quantity with large absolute value.

It makes sense to call this ∞ .

- ∞ has some numberlike properties, but does not have all properties that a number would have.

The following are undefined: $\infty - \infty$, ∞/∞ .

The following are defined $\infty+1 = \infty$, $5^{\infty} = \infty$.

- NAN, on the other hand, is completely undefined in the sense that it can be anything. If $b \neq 0$, and $a/b = c$, then $a = b*c$.

However, for any numbers N we have $0 = N*0$ so $0/0 = N$ for any number N.

- Double-precision numbers use E = 2047 to represent ∞ (F = 0) and NAN (F \neq 0).

For both single-precision and double-precision numbers, the case E = 0 is used to represent 0 with F = 0 and what are called DENORMALIZED NUMBERS with F \neq 0.

- The denormalized numbers do not have a leading 1 and permit the representation of smaller numbers.

For more details see other books on numerical analysis and computer arithmetic.

- Summary of Special Values

E	F	Signifies
0	0	0
0	\neq 0	Denorm
255	0	∞
255	\neq 0	NaN