CPTR 215 Assembly Language Programming I
Number Systems Lecture

HOW CAN WE REPRESENT NUMBERS?

Why do two lines drawn to look like this, "7", mean the number seven? Obviously they mean that because people have agreed that "7" means the number seven. We could just have well as used any other symbol (for example "3" or 'Q') to represent the number seven. In fact, wouldn't it make even more sense to represent numbers by drawing the appropriate number of dots. Thus, seven would be ".......". This representation has the advantage that the number eight can be represented by "........" instead of a completely different symbol (such as "8"). In dot notation addition is easy.

"...." + "..." = "......."

There are no addition tables to memorize, we just put the dots together.

The Romans invented a number representation similar to our dot notation. In the Roman Numeral System the first three numbers are

```
.    -->  I
..   -->  II
...  -->  III.
```

But the Romans must have realized that large numbers were going to be rather inconvenient to write down so they came up with some more symbols. For example "V" represents "IIIII". And "X" represents "VV". They added one other twist too, whenever a lesser symbol precedes a greater one, the value of the lesser one is subtracted from the greater. So

IV --> IIII.

It is possible to represent even large numbers fairly easily: Man landed on the moon in the year MCMLXIX. One of the problems with the Roman system is that there are multiple representations for the same number. It is just as correct to say that "Man landed on the moon in year MCMLXVIV." But the biggest problem with this system is that multiplication and division are really hard. Think about it. Without resorting to our traditional number system, how would you solve this problem?

```
LXXIX
-------  .
 XIV
```

Perhaps you can guess why the Romans didn't make too much progress in mathematics. Even though one can represent all the integers with Roman numerals, it is not an appropriate number representation system for computations.

Sometime around 500 AD, an Arabian astronomer had a better idea. Not only was the new Arabic notation better for representing long numbers than the Roman method, it greatly facilitated calculations. The breakthrough was to let each symbol have a different value depending on its position in the number. First you decide on a base (say ten), then you invent ten symbols (for instance, "0","1","2","3","4","5","6","7","8","9"). Thus was born the Positional Number System.

This is the number system that we have been using all our lives but probably didn't realize how it worked. Have you ever noticed that the symbol "6" in the number 436 means something entirely different than "6" in the number 4683? In the first case it means just six, in the second it means six hundred. To be precise, the value of a symbol is equal to its value as a digit times the base raised to the power of the digit's position in the number. (Positions are always counted from right to left starting with 0.) For example, in base ten the number 6825 is really

$$6*10^3 + 8*10^2 + 2*10^1 + 5*10^0 .$$

(Recall that any number raised to the zero power is simply 1.)

The positional number system works with bases other than ten, of course. If we use base eight then we need only the symbols "0","1","2","3","4","5","6", and "7". The number 3512 is really

$$3*8^3 + 5*8^2 + 1*8^1 + 2*8^0 ,$$

which in base ten would be written as 1283. When we are dealing with numbers with different bases, in order to avoid confusion we often use a subscript on the number to indicate the base. For example, 1234_{10} or 1234_8. (Incidently, everyone has agreed that the subscript is always written in base ten.) Base ten is called decimal (from the Latin decem, meaning ten).

There is nothing magical about base ten. We are used to it and it seems natural, but any other base like eight, twelve, or fifty would serve just as well. One disadvantage of low bases is that numbers become very long. The number 256_{10} equals 100000000_8. The primary disadvantage with large bases is that there are lots of symbols to memorize and keep track of. In base fifty we would have to think up fifty unique symbols.

As familiar as decimal is to us, it just doesn't work out very well for computers. (You will see better why after the lecture on digital logic.) It turns out that computers love to think in terms of ON versus OFF, or 5 VOLTS versus 0 VOLTS, and this lends itself naturally to base two. Since base two, or binary, requires two symbols, the computer's OFF and ON suffice nicely. On paper, we naturally use the symbols "0" and "1", but the computer doesn't know anything about our symbols "0" and "1", just its two states ON and OFF. Counting to ten in binary is

quite easy once you get used to it.

Counting in binary:

zero	0	(Computers and computer scientists always
one	1	start counting from zero.)
two	10	
three	11	
four	100	
five	101	
six	110	
seven	111	
eight	1000	
nine	1001	
ten	1010	

You might as well study this until it becomes second nature to you. You will not really understand anything in the rest of this course until you have mastered binary notation.

Notice that counting in binary is really just the same as counting in decimal except that we run out of symbols faster. In decimal, each symbol in a number is called a decimal digit. In binary, each symbol is called a binary digit, or bit. So it takes four bits to represent the number ten.

Addition in binary works just the same as in decimal.

```
  1001110
+ 1100101
---------
 10110011
```

In grade school we learned our addition and multiplication tables (for decimal), now we should learn them for binary.

+	0	1
0	0	1
1	1	10

*	0	1
0	0	0
1	0	1

As you can see, if we had all learned binary instead of decimal, second grade math would have been a lot less traumatic.

Here's a multiplication problem.

```
    1011
  * 1101
 -------
    1011
   0000
  1011
 1011
--------
10001111
```

Since you will be dealing with binary numbers so frequently in assembly language, it pays to memorize a few common ones.

Useful Powers of Two:

Decimal	Power of 2	Binary
1	0	1
2	1	10
4	2	100
8	3	1000
16	4	10000
32	5	100000
64	6	1000000
128	7	10000000
256	8	100000000
1024	10	10000000000
32768	15	1000000000000000
65536	16	10000000000000000

Tables like this can be used to convert numbers between binary and decimal. For example, 100110_2 is equal to 32 + 4 + 2 or 38. Likewise, 97 is equal to 64 + 32 + 1 or 1100001.

Numbers written in binary tend to be rather lengthy. For instance, 65536 in decimal takes only 5 digits, where in binary it takes 17. However, lest you conclude that binary notation is hopelessly unwieldy in handling any reasonably large numbers, note that with only 30 binary digits we can represent numbers as large as about 10^9.

Every operation on a computer is carried out in binary. However, binary notation can become very tiresome for human programmers to use. So, as a compromise between the binary that the computer loves and the decimal that we are used to, hexadecimal notation is used in assembly languages to communicate with the computer.

If we take a binary number, such as 1001101110101101 and clump its bits into groups of four bits each, we could write this number like this

1001 1011 1010 1101.

This is much like writing large decimal numbers with a comma between groups of three digits. How many unique combinations of four bits are there? Right, sixteen. If we let each unique combination of four bits correspond to a symbol there will be 16 different symbols. This leads us naturally to base 16, otherwise known as hexadecimal (hex for short). Since we only have ten symbols for our numbers in decimal, we will raid the alphabet of its first six symbols, "A" through "F" so we have a total of 16 symbols. The hex symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

The arithmetic operations work just as well in hex as they do in decimal or binary. However, the addition and multiplication tables are considerably larger than with binary. I don't suppose there is much chance of having everyone learn the hex multiplication table, is there? Fortunately, we won't have to since the assembler does most of the hex calculations for us. And when there is the occasional need, there are hex calculators widely available.

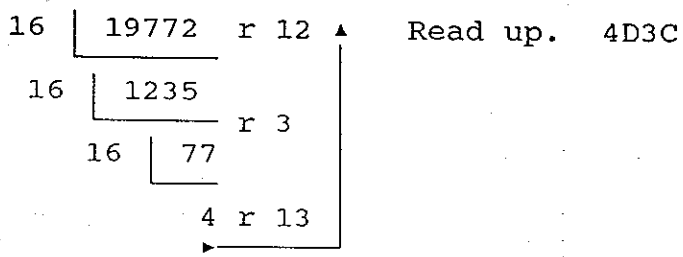
4-Bit Binary Combination	Hexadecimal Digit
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Incidently, when you see a number like 4C00 its a safe bet that the number is in hex. However, 1000 could be 1000_{16} or 1000_{10} which are completely different numbers. If there is any chance for confusion, we will always specify the base using the subscript or verbally (by saying "hex" or "decimal").

It is worth seeing how to convert between hex and decimal. First, convert a hex number to decimal

$$4D3C = 4*16^3 + 13*16^2 + 3*16 + 12 = 19772$$

Going the other way is a little harder.



A group of four bits is often called a nibble. Two nibbles or 8 bits is called a byte. And in the context of the 8088, two bytes or 16 bits is called a word.

REPRESENTING NEGATIVE NUMBERS

Binary notation so far has only shown us how to represent positive integers. Rest assured that it is also capable of representing negative numbers (not just integers either). For the sake of simplicity let's say that our computer uses only 4-bit registers. In these registers we can store sixteen different combinations of bits--enough to hold the numbers 0 through 15. But what if we also wanted to deal with negative numbers? Well we have to give up some of the positive integers and hand over those bit combinations to be used by the negative numbers. There are at least three ways of deciding which negative numbers get which bit combinations. They are called. Signed-magnitude, two's-complement, and ones'-complement (pay attention to the placement of the apostrophes in those last two). The following chart shows how the arrangements are made on the three representations.

Binary	Signed-Magnitude	Two's-Complement	Ones'-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	-0	-8	-7
1001	-1	-7	-6
1010	-2	-6	-5
1011	-3	-5	-4
1100	-4	-4	-3
1101	-5	-3	-2
1110	-6	-2	-1
1111	-7	-1	-0

Signed-magnitude is the easiest to explain. The highest-order bit is used to represent the sign. If the sign bit is 0, then the rest of the bits of the number are just treated as a regular 3-bit positive integer. If the sign bit is 1, then the number of the negative of the positive number represented in the other three bits.

Ones'-complement is the same as signed-magnitude up to 0111. At that point (i.e., the left-most bit is set), the number is the negative of the bits after they are all inverted. For example, the bit pattern 1100 has its left-most bit set, so after inversion it is 0011 which is 3. Therefore the number of -3.

Two's-complement is similar to ones'-complement except that after inverting the bit pattern and interpreting it as a negative number, subtract one from it. For example, the bit pattern 1100

after inversion becomes 0011 which would be interpreted as -3, but after subtracting one it becomes -4.

As strange as two's-complement seems it is actually the most useful of the three schemes. Among other reasons, it does not suffer from the problem of two different representations for 0 that the other two do. In the 8086 family, two's-complement notation is used whenever signed numbers are involved.

Please note that even though the example above used four bits, the representations will work for any number of bits. Keep in mind though that in order to determine the two's-complement (or ones'-complement) representation of a number, one must know how many bits are to be used.

The following examples show how to convert from one representation to another. We will use 8 bits.

The decimal number 27 can be expressed in binary as 11011. In an 8-bit register this will be 00011011. This is the same in all three representations.

To do the number -27, first look at the number 27. As shown above, its 8-bit representation is 00011011. So in signed-magnitude form: -27 --> 10011011.

The ones'-complement form can be found by inverting all the bits: -27 --> 11100100.

The two's-complement form can be found by adding 1 to the ones'-complement form: -27 --> 11100101.