Data Representation

Module 2
CS 272
Sam Houston State University
Dr. Tim McGuire

Positional Number Systems

- Decimal (base 10) is an example
  - e.g.,
    - 435 means
      - 400 + 30 + 5
      - $4 \times 10^2 + 3 \times 10^1 + 5 \times 10^0$
- Example of a non-positional system: Roman numerals
  - inconvenient for humans
  - unusable for computers
- This concept applies to other bases as well

Binary, Octal, & Hexadecimal Numbers

- Base 2 -- natural for computers
  - 0 represents OFF, 1 represents ON
- Base 10 -- natural for humans
  - decimal system uses 10 symbols, 0 - 9
- binary system uses 2 symbols, 0 & 1
  - **0**, **1**, **10**, **11**, **100**, **101**, **110**, **111**, **1000**, **1001**, etc.
  - 1101_2 may be written as
    - $1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 8 + 4 + 0 + 1 = 13_{10}$

Problems with Binary

- Binary numbers are long and cumbersome for people:
  - $\underbrace{10001011000111110_2}_{\text{17 digits}} = \underbrace{71{,}230_{10}}_{\text{5 digits}}$
  - Easy to drop a digit by hand
  - Therefore, we often use a binary-compatible base (decimal is not binary-compatible)

Octal and Hexadecimal

- Two binary compatible bases are (a) octal -- base 8, and (b) hexadecimal -- base 16
- For base 8, we need eight symbols, 0 - 7

decimal	0	1	2	3	4	5	6	7	8	9	10
binary	0	1	10	11	100	101	110	111	1000	1001	1010
octal	0	1	2	3	4	5	6	7	10	11	12

- When binary changes from 3 symbols to 4, octal changes from 1 symbol to 2
  - Reason: 3 binary digits form one octal digit
- Octal was used on several machines, most notably the PDP-11
- $31_8 = 3x8^1 + 1x8^0 = 24 + 1 = 25_{10}$

Hexadecimal

- Hexadecimal (hex for short) is performed similarly, only 16 symbols are needed (0 - 9 and A - F)
- Four bits for one hexadecimal digit

decimal	0	1	2	3	4	5	6	7	8	9	10	11	12
binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100
hex	0	1	2	3	4	5	6	7	8	9	A	B	C

- **0001 0110 0011 1110 = $163E_{16}$**
- We often use a suffix of h for hex numbers, e.g., 163Eh

Conversions

- Hexadecimal to decimal

```
2BD4₁₆ = 2x16³ + Bx16² + Dx16¹ + 4x16⁰
= 2x4096 + 11x256 + 13x16 + 4x1
= 8192 + 2816 + 208 + 4 = 11,220₁₀
```

$2BD4_{16} = 2 \times 16^3 + B \times 16^2 + D \times 16^1 + 4 \times 16^0 = 2 \times 4096 + 11 \times 256 + 13 \times 16 + 4 \times 1 = 8192 + 2816 + 208 + 4 = 11{,}220_{10}$

- Decimal to hexadecimal

```
11172 ÷ 16 = 698 r 4
  698 ÷ 16 =  43 r 10 (Ah)
   43 ÷ 16 =   2 r 11 (Bh)
    2 ÷ 16 =   0 r 2
```

Converting the remainders to hex and putting them together in reverse order, we get

$11172_{10} = 2BA4_{16}$

Conversions Between Hex and Binary

- To convert a hex number to binary, we need only express each hex digit in binary
  - Convert 2B3Ch to binary

```
   2    B    3    C
0010 1011 0011 1100 = 0010101100111100
```

- To go from binary to hex, just reverse this process, starting from the right; pad extra zeroes, if necessary, on the left
  - Convert 1110101010 to hex

```
0011 1010 1010 = 3AAh
```


Conversions Between Hex and Octal

- When converting from one binary compatible base to another, it is easiest to go through binary as an intermediate step
- Hexadecimal to octal (go through binary)

$3F74_{16} = 0011\ 1111\ 0111\ 0100 = 0\ 011\ 111\ 101\ 110\ 100 = 037564_8$

Exercise:

- Convert the following ***octal*** numbers to ***hexadecimal***:
  - 12, 5655, 2550276, 76545336, 3726755

Addition and Subtraction

- base 10

```
 2546
+1872
 4418
```

- base 16

```
 5B39
+7AF4
 D62D
```

- base 2
  - easier than hex because the addition table is so small

```
 100101111
+000110110
 101100101
```


A Note on Notation

- We will represent base 2 numbers with a **b** suffix, e.g., **0100111100001010b**
- Base 16 numbers are represented with an **h** suffix, e.g., **4F0Ah**
  - if the hex number starts with A-F, prefix a **0** at the beginning so it doesn't look like a variable name, e.g., **0BACh** , not **BACH**
- Base 10 can be suffixed with a **d**, but no suffix indicates base 10 by default
- We don't worry about octal on Intel processors

Two's Complement Arithmetic

- So far, the numbers we have discussed have had no size restriction on them
- However, since computer arithmetic is performed in registers, this will restrict the size of the numbers
  - On some machines, this is 8 bits, others 16 bits, and others it is 32 or even 64 bits
  - To keep it simple, we'll use 4 bits at first

The Fabulous Four-bit Machine (FFM)

- The possible numbers it can hold are:
  `0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111,`
  `1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111`
- In order to identify particular bits in a byte or word, we use the following terms:
  - lsb -- least significant bit -- rightmost bit position
    - always numbered as bit 0
  - msb -- most significant bit -- leftmost bit position
    - on FFM it is bit 3

Unsigned Integers

- An unsigned integer is one which is never negative (addresses of memory locations, ASCII codes, counters, etc.)
- On the FFM, then, we can represent numbers from 0 to 15 (0000b to 1111b)
  - On the Intel 8086 the range of integers is 0 to 2^{16}-1 (0 to 65535)
- If the lsb is 0, the number is even; if it is 1, the number is odd

Signed Integers

- How do we represent negative numbers?
- We have three possible methods:
  - Sign and magnitude
  - One's complement
  - Two's complement

Sign and Magnitude

- The **msb** of the number represents the sign of the number
- 0 means positive, 1 means negative
- On FFM
  - **`0000`** to **`0111`** represent 0 - 7
  - **`1001`** to **`1111`** represent -1 to -7
  - **`1000`** is not used (negative **`0`**)
- Easy to understand, but doesn't work well in computer circuits (too many special cases)

One's Complement

- The ***one's complement*** of an integer is obtained by complementing each bit
  - The one's complement of 5 (**`0101b`**) is **`1010b`** = -5
  - The one's complement of 0 (**`0000b`**) is **`1111b`** = -0 (*here we go again*)
- \- 5 + 5 = **`0101`** + **`1010`** = **`1111`** = **`-0`**
- The negative 0 problem can be solved by using ***two's complement***

Two's Complement

- To get the two's complement of an integer, just add 1 to its one's complement
- The two's complement of 5 (**0101**) is **1010+1** = **1011**
- When we add 5 and -5 we get

```
  0101
  1011
 -----
 10000
```

- Because the FFM can only hold 4 bits, the 1 carried out from the msb is lost and the 4-bit result is 0

Complementing the Complements

- It should be obvious that taking the 1's complement of a number twice will give the original number (-(-5) = 5)
- If the 2's complement is to be useful, it must have the same property
- 5 = **0101**, -5 = **1011**, -(-5) = **0100+1** = **0101**

Moving past 4 bits

- Everything true of the FFM is still true for 8, 16, 32, or even 64 bit machines
- Example: Show how the base-10 integer -97 would be represented in (a) 8 bits and (b) in 16 bits, expressing the answer in hex
- (a)
  - **97 = 6x16+1 = 61h = 0110 0001b**
  - **-97 = 1001 1110 + 1 = 1001 1111b = 9Fh**
- (b)
  - **97 = 0000 0000 0110 0001b**
  - **-97 = 1111 1111 1001 1111b = FF9Fh**

Decimal Interpretation

- We have seen how signed and unsigned decimal integers may be represented in the computer
- The reverse problem is how to interpret the contents as a signed or unsigned integer
  - Unsigned is relatively straightforward -- just do a hex to decimal conversion
  - Signed is more difficult -- if the msb is 0, the number is positive, and the conversion is the same as unsigned
  - If the msb is 1, the number is negative -- to find its value, takes the two's complement, convert to decimal, and prefix a minus sign

Signed and Unsigned Interpretations

Hex	Unsigned decimal	Signed decimal
0000	0	0
0001	1	1
0002	2	2
. . .	. . .	. . .
0009	9	9
000A	10	10
. . .	. . .	. . .
7FFE	32766	32766
7FFF	32767	32767
8000	32768	-32768
8001	32769	-32767
. . .	. . .	. . .
FFFE	65534	-2
FFFF	65535	-1

Example

- Suppose AX contains FE0Ch
  - The unsigned decimal interpretation is:
    - 65036
  - To find the signed interpretation:
    - FE0Ch = **1111 1110 0000 1100**
    - 1's cmplment = **0000 0001 1111 0011**

```
                 +1
---------------------
0000 0001 1111 0100
```

= 01F4h = 500

Thus, AX contains -500

Character Representation

- Not all data are treated as numbers
- However they must be coded as binary numbers in order to be processed
- ASCII (American Standard Code for Information Interchange) is the standard encoding scheme used to represent characters in binary format on personal computers

ASCII Code

- 7-bit encoding => 128 characters can be represented
  - codes 0-31 and 127 are control characters (nonprinting characters)
  - control characters used on PC's are: LF, CR, BS, Bell, HT, FF

Input and Output

- ASCII characters codes are used for input and output
  - E.g., when the number 5 is entered on the keyboard, the value read into the processor is the ASCII character code 35h
  - In order to do numeric processing, the ASCII code must be converted to the true numeric value by subtracting 30h, which gives the value 5
  - Then, before the result can be displayed the numeric value must be reconverted to an ASCII code which can be displayed.

Table of ASCII Codes

- A computer may assign special display characters to some of the non-printed ASCII codes
- The screen controller for the PC actually displays an extended set of 256 characters
- A table with the extended ASCII character set is available from the course web page

