CS 520
Bits, Bytes and Integers

Number Systems

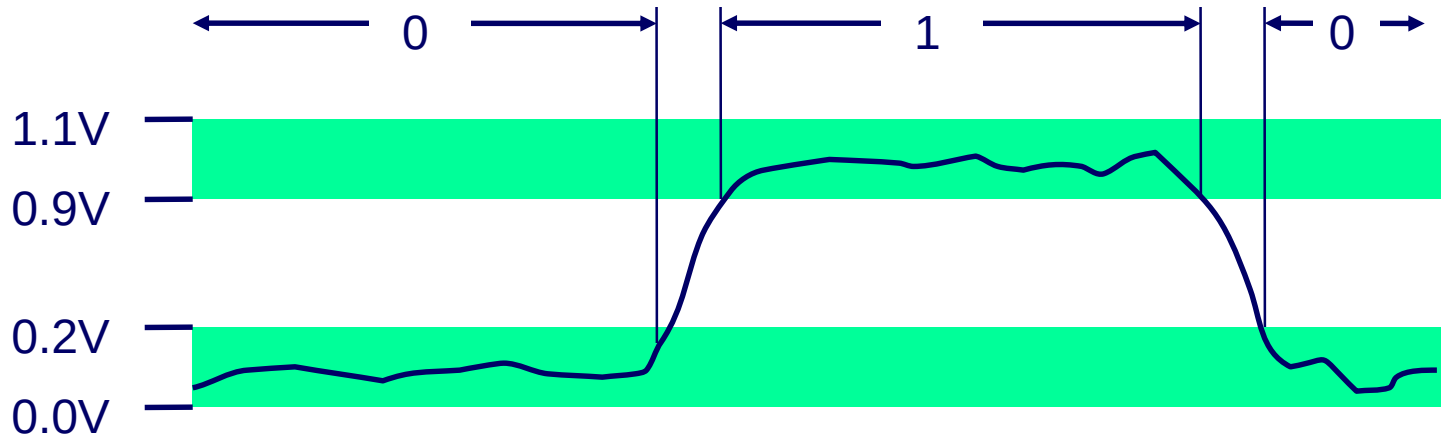
- Base or Radix is the total number of digits used in the number system
  - Decimal Numbers: 0 through 9 digits (1415, 7980)
  - Binary Numbers: 0 and 1 (0110, 101101)
  - Hexadecimal Numbers: 0 through 9, A through F (30AD, 9129)
  - Octal: 0 through 8 (0152, 7216)
- Bit (**B***inary Dig***it**)
- So what exactly is 1385 in decimal?!

Binary number conversion & Byte

- What is 01101 in Binary?
- Binary → Decimal
  - Use radix or base of number system for conversion
  - $01101_2 = 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 0+8+4+0+1 = 11_{10}$
- Byte: A series of 8 bits
- What is 10111101 in Binary?
- So how many numbers can a byte hold?
- What is a kilo byte?

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



Binary numbers in digital electronics

- Greatly simplifies engineering of electronic circuits
- In reality, electrical signals can not change state from 0V to 5V in an instant
- Wires are not perfect conductors. Every wire acts as a small antenna and emits radio waves, and that radio waves are picked by other wires which also act as antennae.
- Within a wire, current bounces whenever it reaches a spot which is different from the rest of the wire.
- From electrical point of view, this gets extremely complicated to manage, and it becomes almost impossible to design a circuit that would operate reliably at high speeds and low power.

Binary numbers in digital electronics

- By choosing only two states, you eliminate much of these problems. If the voltage is sufficiently small (i.e. 0–1V), then you treat that as “0V”, and if it is sufficiently high (i.e. 4–5V), you treat it as “5V”.
- From infinite number of possibilities you now have to manage only two. We still have to think about the timings and other stuff, but it’s much simpler.
- You can start thinking about signals in terms of bits and boolean logic, which is really neat.

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $3FA1D37B_{16}$ in C as
      - 0x3FA1D37B
      - 0x3fa1d37b
- Hex numbers can be viewed as ‘compressed’ binary numbers
- 3F A1 D3 7B --> 0011 1111 1010 0001 1101 0011 0111 1011

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Hex to Decimal conversion

- Radix of Hex numbers is 16
- Hex → Decimal
  - Use radix or base of number system for conversion
  - $3F9B_{16} = 3x16^3 + 15x16^2 + 9x16^1 + 11x16^0$

    $= 12288 + 3840 + 144 + 11 = 16283_{10}$

Decimal to Binary conversion

- Divide decimal number repeatedly by 2 (radix of binary number) until you cannot divide anymore
- Take the remainders from bottom to top!
- 87_{10} → Binary

Decimal to Hex conversion

- Same scheme as decimal to binary conversion – repeated division by 16
- 2113_{10} → Hex
- Simplified technique:
  - 2113/16 = 132 R 1
  - 132/16 = 8 R 4
  - 8/16 = 0 R 8
  - Take all the remainders from last to first → 841_{16}

Integer sizes in Intel Architecture

Number of bits	Name	Hex Digits
8 bits	Byte	2
16 bits	Word	4
32 bits	Longword	8
64 bits	Quadword	16

- Intel IA-32
  - 32 bit registers
  - 32 bit addresses
- Intel IA-64
  - 64 bit registers
  - 64 bit addresses

- Hex representation is very convenient for Intel!

Historically speaking .. CDC 6600 (1960s, 70s)

- 6 bit character
- 60 bit word
- 18 bit addresses
- 18 & 60 bit registers
- Octal was convenient for representation
  - Character needs 2 Octal digits
  - Address needs 6 Octal digits
- Reason for such a large word size - mostly interested in floating point values
- Modern machines are byte-oriented since we are mostly interested in text processing
- Text processing in CDC was mostly a nightmare since we either wasted space or spent time packing and unpacking characters to minimize wastage

Data in Memory

- Example: Byte values in memory

Address	Data
100	
101	
102	
103	
104	
105	00010001
106	
107	

$17_{10} = 10001_2$

It's actually padded up with leading zeroes

$17_{10} = 00010001_2$

Data in Memory

- If the data is in a 32-bit quantity (longword) it's actually stored with all the leading zeroes:

31 24 23 16 15 8 7 0

17_{10} = 0000 0000 0000 0000 0000 0000 0001 0001_2

higher order bytes ---------- > lower order bytes

17_{10} = 00 00 00 11_{16} or 00000011_{16}

- How these bytes actually get stored in memory depends on "endian-ness"

Data in Memory – Big Endian & Little Endian

- Big Endian – high order bytes go first (in increasing order of memory addresses) (IBM, ARM)
- Little Endian – low order bytes go first (Intel, AMD)

17_{10} = 0000 0000 0000 0000 0000 0000 0001 0001_2

Big Endian

Address	Data
100	
101	
102	
103	
104	
105	00000000
106	00000000
107	00000000
108	00010001

Little Endian

Address	Data
100	
101	
102	
103	
104	
105	00010001
106	00000000
107	00000000
108	00000000

Data in Memory – Big Endian & Little Endian

$17_{10} = 00\ 00\ 00\ 11_{16}$(in Hex)

Big Endian

Address	Data
100	
101	
102	
103	
104	
105	00
106	00
107	00
108	11

Little Endian

Address	Data
100	
101	
102	
103	
104	
105	11
106	00
107	00
108	00

Big Endian & Little Endian

- In “Little Endian” form, assembly language instructions for picking up a 1, 2, 4, or longer byte number proceed in exactly the same way for all formats: first pick up the lowest order byte at offset 0.
- Also, because of the 1:1 relationship between address offset and byte number (offset 0 is byte 0), multiple precision math routines are correspondingly easy to write.
- In “Big Endian” form, by having the high-order byte come first, you can always test whether the number is positive or negative by looking at the byte at offset zero. You don’t have to know how long the number is, nor do you have to skip over any bytes to find the byte containing the sign information.
- The numbers are also stored in the order in which they are printed out, so binary to decimal routines are particularly efficient.