CSE 3430
Overview of Computer Systems For Non-Majors

Part 2

Storing Information (contd.)

- Floating point numbers: A powerful representation used to represent a wide range of values (the way shown below is used in IEEE 754 (single precision):
  - 1 bit for the *sign* of the number
  - 8 bits to represent a *signed* exponent to a base of 2
  - 23 significant bits (the X's) in the form 1.XX…X
  - The value represented is: $\pm$ 1.XX…X * 2^{exp}
  - NOTE CAREFULLY that there is an implied 1 before the binary point!
  - This works because the use of the exponent covers a wide range of values with the number of significant bits being the same at different values
- The description above is for single precision; we will not look at double precision

IEEE 754

- The most common way of encoding floating point numbers is using a standard called IEEE 754 (IEEE is Institute of Electrical and Electronics Engineers)
- IEEE 754 is used for floats on Intel based PCs, Macs, and most Unix and Linux platforms (but other ways of representing floats are used in some systems).
- We will only look at single-precision (32 bit) encoding; although 64-bit encoding works in the same way, just with more bits to represent floating point numbers with a larger range and greater precision, doing 64-bit problems is impractical for our purposes.

IEEE 754 continued

- In this standard, the first bit (msb) encodes the sign of the number, 0 for non-negative, and 1 for negative
- The next 8 bits encode the exponent, but the exponent is encoded with a *bias* (a number added to the true exponent); to get ***the true exponent*** (exp), subtract 127 (the bias).
- It is really important to distinguish the biased exponent (what is encoded as part of the bit string which stores the floating point value) from **the true exponent** (what you need to determine the value of the floating point number encoded)
- For a 32 bit IEEE 754-encoded number, the format is:

X	YYYYYYYY	ZZZZZZZZZZZZZZZZZZZZZZZ
(1-bit sign)	(8-bit biased exponent)	(23-bit mantissa)

The standard form of the encoded number is (NOTICE the implied 1):

$$(-1)^X * 1.ZZZZZZZZZZZZZZZZZZZZZZZ * 2^{exp}$$

REMEMBER: *exp* in the formula above is **the *true* exponent**, **NOT the biased exponent (which is the exponent represented as part of the 32 bit encoding)**.

ALSO NOTE: The 1 before the . (the binary point, since it's base 2) is not part of the encoding; *it is implied*.

Example

- Let’s look at how to get the decimal value of an IEEE 754 encoded floating point number with an example (spaces are used to separate the parts of the encoding – sign/exponent/mantissa):

  1 10000011 01011100000000000000000
- Step 1: Since the sign bit is 1, the number is negative.
- Step 2: Get the exp (true exponent). The encoded exponent is 10000011
  - What is the decimal value of the exponent?
  - Subtract the bias (127) from the biased exponent to get the true exponent.
  - What is the true exponent (exp)?
- Step 3: Write the binary value of the floating point number in standard form.
- Step 4: Convert the floating point number to decimal.
- Analysis on the following slides.

Example (Steps explained on following slides)

- Let's look at how to get the decimal value of an IEEE 754 encoded floating point number with an example (spaces are used to show the parts of the encoding – sign/exponent/mantissa):

  1 10000011 01011100000000000000000
- Step 1: Since the sign bit is 1, the number is negative.
- Step 2: Get the exp (true exponent). The encoded exponent is **10000011**
  - What is the decimal value of the encoded exponent: **128 + 2 + 1 = 131**
  - Subtract the bias (127) to get the true exponent: **131 – 127 = 4**
  - What is the true exponent (exp): **4**
- Step 3: Write the binary value of the floating point number in standard form:

  1.010111 X 2^4 (NOTE: Only bits up to and including the rightmost 1 are written)
- Step 4: Convert the floating point number to decimal (See slides below for details), and include the negative sign, if the sign bit was 1 (details on how to do the conversion are given below on the following slides):

  10101.11 = -21.75

Another example (with details)

- Let’s try this IEEE 754 encoded number, and see what we get:

  0 10000001 0001000000000000000000

Step 1

0 10000001 00010000000000000000000

- Step 1: Since the sign bit is 0, the number is *positive*.

Step 2

0 10000001 00010000000000000000000

- Step 2: Get the exp (true exponent). The encoded (biased) exponent is 10000001
  - What is the decimal value of the biased (encoded) exponent?

    10000001 = 128 + 1 = 129
  - Subtract the bias (127) to get the true exponent:

    exp = 129 – 127 = 2
  - So exp (the true exponent) is **2**

Step 3

- Step 3: Write the binary value of the floating point number in **standard form**.
- To do this, write the 23 least significant bits of the original IEEE 754 encoded number (the mantissa) after the implied 1. (we can omit any 0’s on the right side, after the last 1 in these last 23 bits) X 2^{exp}
- For our problem, since exp is 2 (from step 2 above) this is:

  1.0001 X 2^2

NOTICE: The 19 0’s on the right of the mantissa have been omitted from 00010000000000000000000 (the 23 least significant bits from the original IEEE 754 encoded number). They can be omitted because they do not change the value (0's on the far right of the mantissa can always be omitted when converting to decimal).

Step 4: Conversion to decimal

- To eliminate the exponent, move the binary point (or radix point) ***to the right for a positive exp*** (true exponent), or ***to the left for a negative exp*** (true exponent): in our problem, since exp is positive 2, move the binary point to the right 2 places:

  100.01

- Now, convert the binary number to decimal.
- The integer part (to the left of the binary point), 100, is 4
- To get the fractional part, treat the bits to the right of the binary point as a whole number, and divide by $2^{(\text{number of bits up to, and including, the rightmost 1})}$

  .01 = $1/(2^2)$ =¼ or .25, so the number is 4.25

- Add sign: negative if sign bit was 1, or positive if sign bit was 0; here it was 0, so the number is +4.25 or just 4.25

Example with negative exp (true exponent)

- Let’s try this example:

  0 01111110 10100000000000000000000
- Step 1: Since the sign bit is 0, the number is positive.
- Step 2: Get the exp (true exponent). The encoded exponent is 01111110
- What is the decimal value of the exponent?
  - Subtract the bias (127).
  - What is the true exponent (exp)?
- Step 3: Write the binary value of the floating point number in standard form.
- Step 4: Convert the floating point number to decimal.

Example with negative exp (true exponent)

- Let’s try this example:

  0 01111110 10100000000000000000000
- Step 1: Since the sign bit is 0, the number is positive.
- Step 2: Get the exp (true exponent). The (biased) encoded exponent is 01111110
- What is the decimal value of the exponent: **64 + 32 + 16+ 8 + 4 + 2 = 126**
  - Subtract the bias (127): **126 – 127 = -1**
  - What is the true exponent (exp): **-1**
- Step 3: Write the binary value of the floating point number in standard form.
- Step 4: Convert the floating point number to decimal.

Step 3

- Step 3: Write the binary value of the floating point number in **standard form**.
- To do this, write the 23 least significant bits of the original IEEE 754 encoded number (the mantissa) after the implied 1. (we can omit any 0's on the right side) X 2^{exp}
- For our problem, since exp is -1 (from step 2 above) this is:

  1.101 X 2^{-1}

NOTICE: The 20 0's on the right have been omitted from 10100000000000000000000 (the 23 bits of the mantissa from the original IEEE 754 encoded number); 0's on the right can always be omitted without changing the value when converting to decimal.

Step 4: Conversion to decimal

- To eliminate the exponent, move the binary point (or radix point) to the right for a positive exp (true exponent), or to the left for a negative exp (true exponent): in our problem, since exp is -1, move the binary point to the left 1 place:

  0.1101
- Now, convert the binary number to decimal.
- The integer part (to the left of the binary point), 0, is 0
- To get the fractional part, use the same method as before: Treat the bits to the right of the binary point as a whole number, and divide by $2^{(\text{number of bits up to and including the rightmost 1})}$

  .1101 = $13/(2^4)$ = 13/16 or 0.8125, so the number is 13/16 (It is fine to write it this way; you do not need to convert to 0.8125 with a calculator).
- Add sign: negative if sign bit was 1, or positive if sign bit was 0; here, it was 0, so the number is: 13/16 or 0.8125 (13/16 is fine)

Values way below 1 and close to 0

- As we saw in the example above, if the exponent (exp) is negative, floating point numbers can be used to represent values which are less than 1.
- If the exponent is negative and very far below zero (say -120 or less), positive numbers can be represented which are ***extremely small***; that is very much less than 1, i.e., ***extremely*** close to 0 (or negative numbers which are much greater than –1, i.e, extremely close to 0, but negative).
- The range of exponents in IEEE 754 single precision is:
  - Biased exponent range: 0 – 255
  - True exponent range (explanation below): -126 to 127

Range of exp

- What is discussed above is only for values of exp from -126 to 127 (biased, or encoded, exponents from 1 to 254). Such values are called ***normalized*** numbers in IEEE 754.
- With this range of true exponents, we can encode floating point numbers in the range: +/- 1.0 X 2^{-126} to +/- 1.0 X 2^{127}
- If the encoded exponent is 0 (all 8 bits of the exponent are 0's), the number is a ***denormalized*** number. This can be used to represent 0, and values with absolute value much less than 1, that is, **values extremely close to 0 (just below or just above 0)**.
  - We will not consider these values.
- Numbers that have a biased exponent of 255 (all 8 exponent bits are 1) are used to encode **+/- infinity (overflow), and errors** (values that are not real numbers, for example, the square root of a negative number).

Comparison of float range and integer range

- If we compare the range of floats using IEEE 754 single precision to 32-bit signed integers, we can see floats have ***a tremendously larger range:***
- Floats: +/- 1.0 X 2^{-126} to +/- 1.0 X 2^{127}
- Signed ints (B2T): -2^{31} to $2^{31} – 1$

Floating Point Range

- Therefore, floating point numbers represented in this way have a tremendous range compared with int representation. For non-negative floats, we can represent:
  - 0
  - From values which are extremely small, but close to 0, to values which are extremely large, much larger than the largest integer which can be represented, even with a 64-bit encoding.
  - Thus, if we want tremendous range, but are willing to trade some precision, we can use floating point data types.
  - If we want **perfect precision**, but much more limited range, we need integer types.
  - *SOME* **encoded floats** represent real number values precisely, but only if the number can be written in the form (+/-) x/y where y is an integer of the form 2^n, for integer n $\geq$ 0.
  - So, fractional values such as 1/5 cannot be stored precisely in binary; they can only be ***approximated***. Likewise, 1/3, 1/6, 1/7, 1/9, etc. can only be approximated in binary representation with a finite number of bits. The approximation is very good generally, but it is important to understand that it is *only an approximation in these cases (a large percentage of cases)!*

Character representation

- Characters
- There are various encoding schemes which are used
- For the Latin alphabet and related characters, and decimal digits (0 to 9) one early system (still being used) is ASCII (American Standard Code for Information Exchange):
  Corresponding to each character, there is a unique 8 bit code
  Example: ‘A’ has the code 0100 0001; ‘B’ has code 0100 0010
    ‘7’ has code 0011 0111; ‘8’ has code 0011 1000;
    Why? … because! (it is an accepted standard)
- Please see the ASCII table posted on Carmen

ASCII – Important Point

- Please note that the most significant bit (the msb, that is, leftmost bit) in an ASCII character encoding is *always 0*.
- This means that the msb can be used to detect errors in the transmission of character data, for example, across a network connection (see discussion of how this can be done below).

Other methods of character encoding – UTF-8

- Because ASCII uses only 7 bits (remember that the msb is always 0), it can encode a very limited number of characters.
- How many?
- If we want an encoding for a larger character set, what can we do?

Other methods of character encoding (cont)

- Right – the same answer as always - use more bits (longer bit strings)!
- UTF-8
  - Unicode
  - Each character is encoded by one to four 8 bit bytes (it is a variable byte-length character encoding method).
  - NOTE: The UTF-8 encoding is the same as the ASCII encoding for characters encoded by ASCII

UTF-8

- UTF-8 can encode Unicode characters. Unicode is an extremely large character set, and currently includes 1,112,064 characters!

UTF-8 (cont)

- This is a large enough number to include encodings for Latin-script alphabets, Greek, Cyrillic (used for Russian and other Slavic languages), Coptic, Armenian, Hebrew, Arabic, Syriac, Thaana, and the N’Ko alphabets, as well as diacritical marks (accents, etc.), and also including Chinese, Japanese, and Korean characters.
- Actually, it includes an encoding for every character in every known language that has a writing system!

Why is it called UTF-8?

- The name UTF-8 comes from *Unicode* (or *Universal Coded Character Set*) *Transformation **F**ormat – **8**-bit* (You do not need to remember this!)

How does it work?

- UTF-8 appears somewhat complicated, but the basic idea is quite simple.
- First, UTF-8 uses a variable length encoding of 1 to 4 bytes for every character.
  - That is, the number of bytes used to encode various characters is *variable* (some characters use 1 byte, some use 2, some use 3, and some use 4).
  - Although the encoding of *any particular character* always uses the same number of bytes, encoding of two different characters may use a different number of bytes.

UTF-8 – How does it work (cont)

- If each character has a variable byte-length encoding (1, 2, 3, or 4 bytes for UTF-8), how does *the decoder* (software, or perhaps a human being, such as you or me ☺) know the length of the character encoding?
- A clever scheme is used to handle this, explained on the next slide.

UTF-8 – Variable byte length encoding

- The scheme is shown on the next slide, with the bits marked x actually used to encode the character (each of these bits, of course, can be 0 or 1, depending on the specific character being encoded).
- The bits that are not represented by x can be thought of as formatting bits; they convey information about the encoding, and not about the specific character represented. These bits always come at the beginning of the byte (they are the msbs).

UTF-8 variable byte length encoding

Number of bytes	Range of codes (decimal)	Number of significant bits	1st byte	2nd byte	3rd byte	4th byte
1	0-127	7	0xxx xxxx	------------	------------	------------
2	128-2047	11	110x xxxx	10xx xxxx	------------	------------
3	2048-65535	16	1110 xxxx	10xx xxxx	10xx xxxx	------------
4	65536-2097151	21	1111 0xxx	10xx xxxx	10xx xxxx	10xx xxxx

UTF-8 conclusion

- Since 2009, UTF-8 has been the dominant character encoding scheme for the World Wide Web
- It now accounts for approximately 93% of all web pages (some of these are pages use just ASCII, but ASCII and UTF-8 are equivalent for one-byte encoded UTF-8 characters) [Please don't worry about remembering the percentage!]

Aside on Hexadecimal

- Hexadecimal (base 16) is used fairly often in computer science, because it is easy to convert to or from binary to hex.
- Every sequence of 4 bits (every half-byte, or "nibble") can be converted to or from hex directly, as follows:
-

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

UTF-8 example – Greek characters
(Please don’t memorize these!)

- Here are some UTF-8 codes for some Greek characters many of us may be familiar with:

Greek character	UTF-8 code (hex)
Δ (uppercase delta)	0xCE94
π (lowercase pi)	0xCF80
Π (uppercase pi)	0xCEA0
Σ (uppercase sigma)	0xCEA3
θ (lowercase theta)	0xCEB8

- Notice that all of the UTF-8 codes for Greek characters are two-byte codes (4 hex digits)

Character strings

- Character strings: Stored in a *sequence* of bytes. Thus "B7A" (encoded in ASCII) is:

B	7	A
0100 0010	0011 0111	0100 0001

- Or, more readably, in *hex* (hexadecimal, or base 16): 0x423741 (the prefix 0x is usually used to indicate that a number is hex; see explanation below of how to convert binary to hexadecimal).

Error Detection/correction

- Information stored in the computer usually remains unchanged … but
- During transmission (e.g., over a network), errors can creep in.
- Simple error detection method: *parity;* for ASCII, use *eight bits* instead of 7 (the value of 8th bit being chosen to get even number of 1's)
- An error detection/correction scheme:
  (VPB: Vertical parity bit; HPB: horizontal parity bit; top right: "corner bit"

	H	e	l	l	o	,		w	o	r	l	d	HPB
VPB	0	0	0	0	0	1	1	0	0	0	0	1	1
bit 6	1	1	1	1	1	0	0	1	1	1	1	1	0
bit 5	0	1	1	1	1	1	1	1	1	1	1	1	1
bit 4	0	0	0	0	0	0	0	1	0	1	0	0	0
bit 3	1	0	1	1	1	1	0	0	1	0	1	0	1
bit 2	0	1	1	1	1	1	0	1	1	0	1	1	1
bit 1	0	0	0	0	1	0	0	1	1	1	0	0	0
bit 0	0	1	0	0	1	0	0	1	1	0	0	0	0

Storing Information (contd.)

- What about *other* values?
- Say, I want to store the grades that a given student obtains in all the homeworks, exams, etc.?
  Just use the appropriate number of words or bytes:
  - Let us say there are ten graded activities each worth a max of 100 pts
  - We could store each of these in a single byte (why?)
    - but it is more common to use a whole word;
  - And use two words to store the name of the student (why?)
  - for a total of 12 words
- What if I want to store the grades of *all* the students in the class?
  Just use N * 12 words (if there are N students in the class)
- *But*: If I look at the actual contents of these words in memory, it will just be a sequence of 0's and 1's! I have to know what each byte represents
- For example, suppose I thought there were 12 (instead of 10) graded activities, what will happen?

Storing Information (contd.)

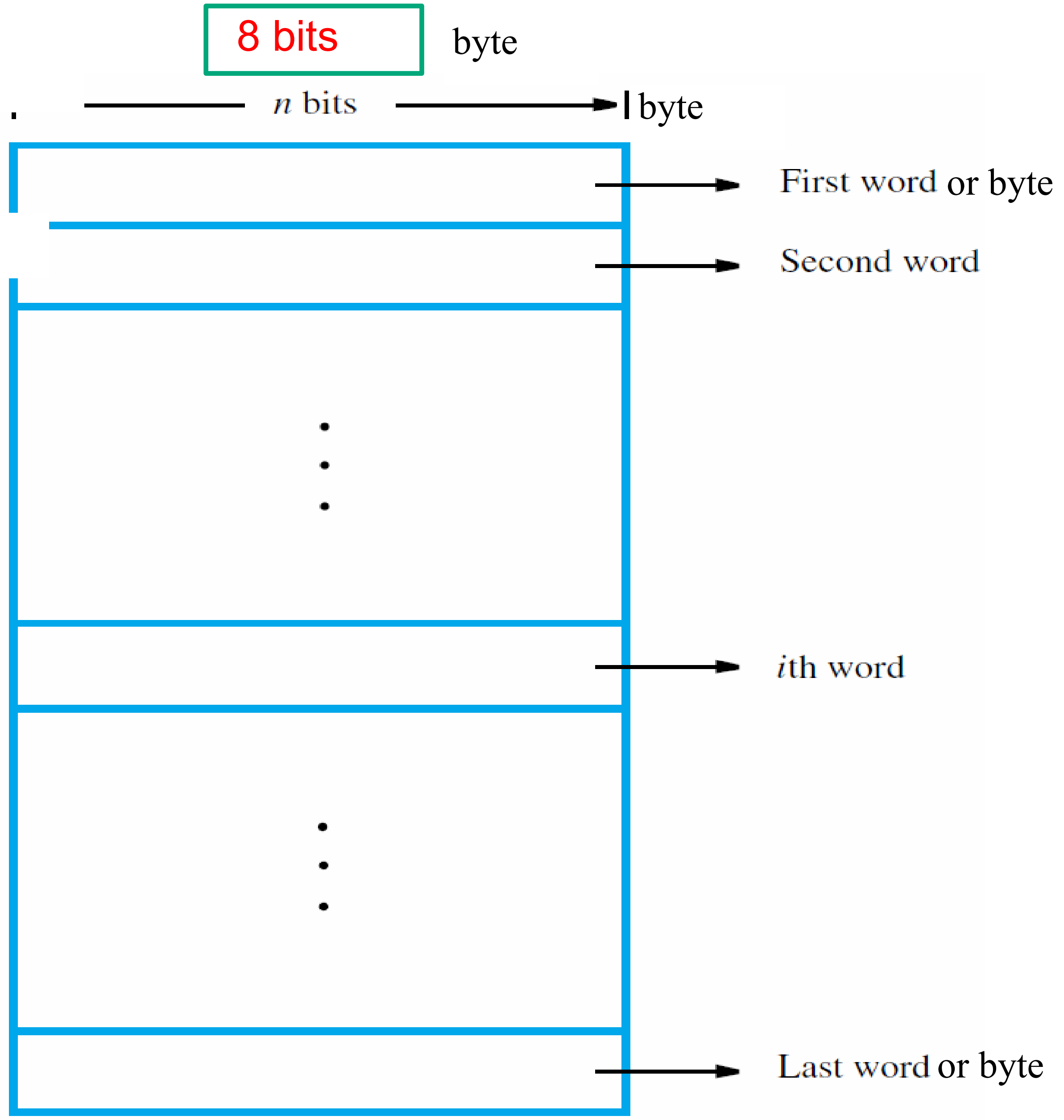
- What about *other* values?
  Say a song that you downloaded (legally:-)?) or a photo? or video?
- For each type of data, standard coding schemes have been defined so that each such "value" can be stored as a (looo…ng!) sequence of bits
- In each case, if you don't know the coding scheme, there is no way for you to know what song or photo or video a given string of bits is …
- In fact, you can't even be sure whether it is a song or a photo or video!
- … or the grade roster of a course!
- **Essential point: *Everything*** in the computer is stored as bit strings. You have to know which value is stored where in memory and what the encoding scheme used to store that value is, in order to be able to interpret (get the meaning of) the encoded data.

Organization of Memory

- Memory is organized as an array of bytes; each byte in the memory array has an index (just as elements of other types of arrays have indexes), and this index of the memory byte is called an address. System addresses always start with index 0.
- In some architectures, memory is byte-addressable (data beginning at *any* byte in memory can be accessed, that is, read or written).
- BUT, in other architectures, addresses must be word-aligned (the only data which is accessible is data which starts at a byte index which is evenly divisible by the size of a word).
- So, in a system which requires aligned access and has an 8-byte word size, only data which starts at an address evenly divisible by 8 could be read or written; 8-byte data starting at address 3, or 7, or 75, for example, could not be accessed.

Memory Structure

- Memory consists of a **huge** number of bytes (of 8 bits each)
- Each byte in memory has a unique *address*
- If the memory has 10 bytes, their addresses will be 0, 1, 2, …, 9
- … represented as bit strings (0000, 0001, 0010, …, 0111, 1001, 1010)
  … because *everything* is represented as a bit string!
- In a typical machine today, the address is represented as a 64-bit string (or 16 hexadecimal digits), and the word size is 8 bytes. In the prior generation of computer systems, addresses were 32 bits, or 8 hexadecimal digits, and word size was 4 bytes.
- Therefore, in the prior generation of machines, we could have a maximum of 4,294,967,296 bytes of memory (why?). In today's machines, we have that number of bytes squared (4,294,967,296 bytes squared) as the maximum amount of memory our machine can have (though real machines do not currently have that much)!
- Numbers (integers and single-precision floats) are usually stored in 4 bytes, though the word size in a 64-bit machine is 8 bytes.


8 bits
byte
n bits
byte
First word or byte
Second word
ith word
Last word or byte