Computer Science 1510

Lecture 2

Lecture Outline

- Computer number systems and representation

Computer Number Systems

- We are accustomed to using the *decimal* number system, which uses 10 digits: $0, 1, 2, 3, 4, 5, 6, 7, 8,$ and 9. To count in decimal we simply add one to get the next number ($1 + 1 = 2$, $2 + 1 = 3$, etc.).

- When we reach the last digit, 9, to add one we concatenate two digits to give $9 + 1 = 10$.

- If we consider a larger number, say 3025, we can decompose that number into its concatenated parts:

$$\begin{aligned} 3025 &= (3 \times 1000) + (0 \times 100) + (2 \times 10) + (5 \times 1) \\ &= (3 \times 10^3) + (0 \times 10^2) + (2 \times 10^1) + (5 \times 10^0). \end{aligned}$$

- The decimal number system uses *base* 10.

Binary

- We have already seen that computers use the *binary* number system to store information.

- The binary number system uses base 2, with digits 0 and 1. Counting in binary we have,

$$\begin{aligned} 0+1 &= 1, \\ 1+1 &= 10, \\ 10+1 &= 11, \\ 11+1 &= 100, \end{aligned}$$

  and so on.

- Although computers use binary to store information, arithmetic may be performed in number systems such as *hexadecimal*, or *octal*.

Hexadecimal and Octal

- The hexadecimal system uses base 16, with digits $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$.

- Counting in hexadecimal:

$$\begin{aligned} 0 + 1 &= 1, \\ 9 + 1 &= A, \\ F + 1 &= 10, \\ 19 + 1 &= 1A, \end{aligned}$$

  and so on.

- Note that "10" in hexadecimal is 16 in decimal. To avoid any confusion we write the base as a subscript, 10_{16}.

- The octal system uses base 8, with digits $0, 1, 2, 3, 4, 5, 6, 7$.

Conversion to decimal

- To convert a number $d_i d_{i-1} \ldots d_1 d_0$ from base b to decimal we decompose the number into its constituent parts,

$$\begin{aligned}(d_i d_{i-1} \ldots d_1 d_0)_b &= (d_i \times b^i) + (d_{i-1} \times b^{i-1}) + \cdots \\ &\quad + (d_1 \times b^1) + (d_0 \times b^0).\end{aligned}$$

- Examples:
  - Hexadecimal to decimal conversion (keeping in mind that $A_{16} = 10_{10}$, $B_{16} = 11_{10}$, etc.):

$$\begin{aligned}1A8_{16} &= (1 \times 16^2) + (10 \times 16^1) + (8 \times 16^0) \\ &= 256 + 160 + 8 = 424_{10}.\end{aligned}$$

  - Binary to decimal conversion:

$$1011_2 = (1\times 2^3)+(1\times 2^1)+(1\times 2^0) = 8+2+1 = 11_{10}$$

Conversion from decimal

- To convert a decimal number to base b we divide out successive powers of b.

- Example: Decimal to hexadecimal conversion: Consider 586_{10}.

  - We can divide out 16^2 since $586/16^2$=2.289. Thus there are 2, 16^2's in 586 plus some leftover (0.289), that is, $586_{10} = (2 \times 16^2) + \ldots$.
  - We are now left with $586 - (2 \times 16^2) = 586 - 512 = 74$. From 74 we can divide out 16^1 since $74/16^1 = 4.625$, $586_{10} = (2 \times 16^2) + (4 \times 16^1) + \ldots$.
  - Finally, we have $74 - (4 \times 16^1) = 10$ remaining, which can be expressed in hexadecimal as $(A \times 16^0)$, giving us the final hexadecimal representation,

    $$586_{10} = (2 \times 16^2) + (4 \times 16^1) + (A \times 16^0) = 24A_{16}.$$

Conversion from decimal

- Conversion from decimal to binary can be done very quickly by using the remainders.

- Example:
  Consider 157.

$$\begin{array}{rcll} 157 \,/\, 2 & = & 78 & \mathrm{R}\ \ 1 \\ 78 \,/\, 2 & = & 39 & \mathrm{R}\ \ 0 \\ 39 \,/\, 2 & = & 19 & \mathrm{R}\ \ 1 \\ 19 \,/\, 2 & = & 9 & \mathrm{R}\ \ 1 \\ 9 \,/\, 2 & = & 4 & \mathrm{R}\ \ 1 \\ 4 \,/\, 2 & = & 2 & \mathrm{R}\ \ 0 \\ 2 \,/\, 2 & = & 1 & \mathrm{R}\ \ 0 \\ 1 \,/\, 2 & = & 0 & \mathrm{R}\ \ 1 \end{array}$$

  Reading the remainders from the bottom we have,

$$157_{10} = 10011101_2$$

Binary to hexadecimal

- Binary numbers are difficult to read since even small numbers require many bits (binary digits).

- Binary is often converted to hexadecimal since the conversion can be done quickly and hex numbers are much easier to read.

- Quick conversion between binary and hex is due to the hex base being a multiple of the binary base ($2^4 = 16$). Thus, every hex digit can be represented by 4 bits.

- We only need to be able to count to 16 in binary.

- Consider $3A5_{16}$.
  Taking each digit: $3_{16} = 0011_2$, $A_{16} = 1010_2$, $5_{16} = 0101_2$,

$$3A5_{16} = 001110100101_2.$$

Character representation

- Recall that 8 binary bits make up a byte.
- An encoding method such as ASCII or Unicode is used to store characters. The ASCII representation of the letter 'A' is 01000001.
- Each letter requires 1 byte, therefore each location in memory can store one character.
- The ASCII table includes the alphabet in both upper and lower case, digits 0–9, symbols (;, &, @, etc.), white spaces (blank space, tab, new line), unprintable symbols such as CTRL-A, and some graphic shapes.

Ascii Table

(nul)	0	(sub)	26	4	52	N	78	h	104
(soh)	1	(esc)	27	5	53	O	79	i	105
(stx)	2	(fs)	28	6	54	P	80	j	106
(etx)	3	(gs)	29	7	55	Q	81	k	107
(eot)	4	(rs)	30	8	56	R	82	l	108
(enq)	5	(us)	31	9	57	S	83	m	109
(ack)	6	(sp)	32	:	58	T	84	n	110
(bel)	7	!	33	;	59	U	85	o	111
(bs)	8	”	34	¡	60	V	86	p	112
(ht)	9	#	35	=	61	W	87	q	113
(nl)	10	$	36	¿	62	X	88	r	114
(vt)	11	%	37	?	63	Y	89	s	115
(np)	12	&	38	@	64	Z	90	t	116
(cr)	13	’	39	A	65	[	91	u	117
(so)	14	(	40	B	66	\	92	v	118
(si)	15	)	41	C	67	]	93	w	119
(dle)	16	*	42	D	68	∧	94	x	120
(dc1)	17	+	43	E	69	_	95	y	121
(dc2)	18	,	44	F	70	‘	96	z	122
(dc4)	20	.	46	G	71	a	97	{	123
(dc3)	19	-	45	H	72	b	98	—	124
(nak)	21	/	47	I	73	c	99	}	125
(syn)	22	0	48	J	74	d	100	∼	126
(etb)	23	1	49	K	75	e	101	(del)	127
(can)	24	2	50	L	76	f	102		
(em)	25	3	51	M	77	g	103		

Integer representation

- Bits are grouped into bytes for memory organization.
- Bits (or bytes) are grouped together into *words* for number storage and processing.
- The number of bits in a word (the word size) varies between computers.
- Integers are usually stored in 16 or 32 bit words, the most commonly used being 32 bits.
- For example, using a 16-bit word, the value 10 could be stored as:

$$0000000000001010$$

- But what about -10??

Integers: Sign-magnitude notation

- One way to store negative numbers is to use 1 bit as a *sign bit* and the remaining 15 (or 31) bits for the value.

- The sign bit would be 1 for negative numbers and 0 for positive numbers.

- So -10 would be stored as:

$$1000000000001010$$

- The largest positive number that can be stored in a sign-magnitude 16-bit word is,

$$0111111111111111 = 2^{15} - 1 = 32767$$

- The largest negative number that can be stored in a sign-magnitude 16-bit word is,

$$1111111111111111 = -(2^{15} - 1) = -32767$$

Integers: Sign-magnitude notation

- Any number outside of the range [-32767,32767] results in an *overflow*.

- What about zero?

- In sign-magnitude notation, zero has two different representations,

$$1000000000000000 = \text{-}0$$
$$0000000000000000 = 0$$

- Operations involving negative and positive numbers does not work properly in all cases.

- One method that is used to overcome these issues is called *2's complement* notation.

Integers: 2's complement notation

- In 2's complement notation, the leftmost bit still represents the sign.

- Positive numbers are stored the same as in sign-magnitude, allowing representation of values in the range $[1, 2^{15} - 1]$ (in the case of a 16-bit word).

- Negative numbers in the range $-x \in [-2^{15}, -1]$ are stored as the binary representation of $2^{16} - x$.

- Example:

$$\begin{aligned} 10 &= 0000000000001010 \\ -10 &= 1111111111110110 \end{aligned}$$

- To see that this does represent -10 we can add the two binary numbers together and the result should be zero. Try this!

Integers: 2's complement notation

- Given a negative integer, we can find its 2's complement representation by:

  1. Start with the binary representation of the positive value.
  2. Complement (reverse) each of the bits (1's complement).
  3. Add one (2's complement).

- Example: 10000111 in sign-magnitude ($= -7_{10}$) is 11111001 in 2's complement.

Real numbers: IEEE standard

- The sign-magnitude and 2's complement notations above are used to represent integers. What about real numbers?

- In general, a real number x can be written in the form

$$x = \pm d_0.d_1 d_2 d_3 \ldots d_t \times \beta^e$$

where

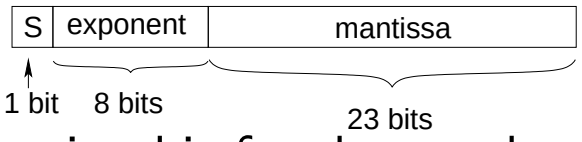
β is the base,
e is the exponent (or characteristic),
$d_i \in 0, 1, 2, \ldots, \beta - 1$, and
$m = d_0.d_1 d_2 \ldots d_t$ is called the *mantissa*.

- If $d_0 \neq 0$, then x is said to be in *normalized* form. From this point on we will consider only numbers in normalized form.

- In the case of binary, d_0 can only be 1 for normalized numbers.

Real numbers: IEEE standard

- A 32-bit IEEE standard floating point number has the following layout:



  where S is the sign bit for the number.

- Note that the exponent does not have a sign bit. How do we store negative exponents?

- To allow for negative exponents we use a biased exponent, where an offset is added to all exponents to ensure that only positive numbers are stored.

- If we have 8 bits for the exponent and e is the (signed) value of the exponent, we would store the value $n = e + 127$

- With a biased exponent these stored values would represent exponents in the range $[-126, 127]$.

- Exponents of -127 ($n = 0$) and 128 ($n = 255$) are reserved for special cases.

Real numbers: IEEE standard

- A floating point number x can be represented as:
  - If $n \in (0, 255)$,

$$x = (-1)^s (1.m)_2 \times 2^{n-127}.$$

  - If $n = 255$ and
    - $m = 0$ then $x = \pm\infty$ depending on s.
    - $m \neq 0$ then x=NaN ("not a number").
  - If $n = 0$ and
    - $m = 0$ then $x = \pm 0$ depending on s.
    - $m \neq 0$ then

$$x = (-1)^s (0.m)_2 \times 2^{-126}$$

    These are "unnormalized" values.

- Given a finite number of bits that can be used to represent a real number, there is a finite set of real numbers that a computer can represent. This set of real numbers is referred to as *floating point numbers*.

Example: Storing 19.25

- To determine how the real number 19.25 is stored in memory we first convert the number to binary.

$$\begin{aligned} 19 &= 10011 \\ 0.25 &= 0.01 \end{aligned}$$

  Therefore, $19.25 = 10011.01$.

- In normalized form: $10011.01 = 1.001101 \times 2^4$.

- Since we store a biased characteristic we note that $4 = 131 - 127$. Thus the value 131 is stored as the exponent. In binary $131 = 10000011$

- Putting the pieces together we have,

$$19.25 = 0 \;\; 10000011 \;\; 00110100000000000000000$$

Example: Storing -118.625

- Since the number to be stored is negative we set the sign bit to 1.
- In binary we have $118.625 = 1110110.101$.
- Normalizing gives $118.625 = 1.110110101 \times 2^6$.
- Since we store a biased characteristic we note that $6 = 133 - 127$. Thus the value 133 is stored as the exponent. In binary $133 = 10000101$
- Putting the pieces together we have,

$$-118.625 = 1 \;\; 10000101 \;\; 11011010100000000000000$$

Example: Storing 0.3

- Unlike the previous examples, 0.3 does not have an obvious representation using negative powers of 2.

- One easy way to convert fractional values to binary is to successively multiply by 2 as follows:

$$\begin{array}{lcll} 0.3 \times 2 & = & 0.6 & \rightarrow \quad 0 \\ 0.6 \times 2 & = & 1.2 & \rightarrow \quad 1 \\ 0.2 \times 2 & = & 0.4 & \rightarrow \quad 0 \\ 0.4 \times 2 & = & 0.8 & \rightarrow \quad 0 \\ 0.8 \times 2 & = & 1.6 & \rightarrow \quad 1 \\ 0.6 \times 2 & = & 1.2 & \rightarrow \quad 1 \text{ (repeat from above)} \end{array}$$

- Thus we have $0.3 = 0.0100110011\ldots$.

- Normalizing gives $0.3 = 1.00110011\cdots \times 2^{-2}$.

- Putting the pieces together we have,

$$0.3 = 0 \;\; 01111101 \;\; 00110011001100110011001$$

Single vs. double precision

- A *single precision* (binary) floating point number is stored in a 32-bit word.

- A *double precision* (binary) floating point number is stored in a 64-bit word.

  In IEEE standard a double precision floating point number uses 1 bit for the sign, 11 bits for the biased exponent, and 52 bits for the mantissa. The bias is 1023.

- A *quadruple precision* (binary) floating point number is stored in a 128-bit word.

  It has a mantissa of 112 bits, and a 15-bit exponent.