Representing Data: Logical Operations on Bits + Beyond Integers

Outline

- Bitwise logical operations
- Representing fractions
  - Base 2
  - General
- Representing general floating-point numbers

Logical Operations

- Operations involving logical TRUE or FALSE
  - two states -- takes one bit to represent: TRUE=1, FALSE=0 Truth tables show the results of logical operations for all possible input combinations (for n source operands, 2^n rows)

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

- View *n*-bit number as a collection of *n* logical values
  - operation applied to each bit independently

Examples of Logical Operations

- AND
  - useful for clearing bits (to 0)
    - AND with zero = 0
    - AND with one = no change

```
      11000101
AND   00001111
      --------
      00000101
```

- OR
  - useful for setting bits (to 1)
    - OR with zero = no change
    - OR with one = 1

```
      11000101
OR    00001111
      --------
      11001111
```

- NOT
  - unary operation -- one argument
  - complements each bit

```
NOT   11000101
      --------
      00111010
```

Logical Operations Examples

- Find the bitwise AND and the bitwise OR of the two inputs 1001 0011 and 0111 1010.
- OR represents an inclusive or and is set to true if either or both of the inputs are 1. Consider the exclusive OR operation XOR, which will not evaluate to true if the inputs are the same. Complete the truth table and then calculate XOR for the same inputs as above.

A	B	A XOR B
0	0	
0	1	
1	0	
1	1	

Logical Operations Examples

- Find the bitwise AND and the bitwise OR of the two inputs 1001 0011 and 0111 1010.

  AND:
  $$\begin{array}{r} 1001\ 0011 \\ \underline{0111\ 1010} \\ 0001\ 0010 \end{array}$$

  OR:
  $$\begin{array}{r} 1001\ 0011 \\ \underline{0111\ 1010} \\ 1111\ 1011 \end{array}$$

- XOR:
  $$\begin{array}{r} 1001\ 0011 \\ \underline{0111\ 1010} \\ 1110\ 1001 \end{array}$$

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

The Problem with Integers…

- What about really large numbers?
  - Integers have limited range
  - 32-bit integers

Signed:	from -2,147,483,648 to 2,147,483,647 (inclusive)
Unsigned:	from 0 to 4,294,967,295 (inclusive)

- What about fractions?

Floating point numbers address both of these concerns

- Floating point numbers increase range, *not* precision.
- Why?
- The number of available bits is the same.

Fractions in Base 2

- Base-2 fractions work just like base-10 fractions.
- To convert **from base-2 to base-10**, add up the relevant powers of 2: because the exponents are negative, we are adding positive powers of ½.
  - Example:

$$(.1011)_2 = \frac{1}{2}^{1} + \frac{1}{2}^{3} + \frac{1}{2}^{4} = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = \frac{8}{16} + \frac{2}{16} + \frac{1}{16} = \frac{11}{16}.$$

  - Note that the numerator is the base-10 representation of the base-2 fraction if it were an integer instead (base-2 1011 represents base-10 11).
  - The denominator is the power of 2 corresponding to the number of binary digits.

Fractions in Base 2

- **Converting a base-10 fraction to base 2:** Let’s convert 0.6875 to base 2.
- Similar to what we did with integers.

$$0.6875 * 2 = 1.375 = 0.375 + 1$$
$$0.375 * 2 = 0.75 + 0$$
$$0.75 * 2 = 0.5 + 1$$
$$0.5 * 2 = 0 + 1$$

- Stop; all remaining digits would be 0.
- Read off the base-2 representation of the fraction as the remainders from the top down (because this time we are moving away from the decimal point to the right). Thus, $(0.6875)_{10} = (0.1011)_2$.
- Verify:

$$(0.1011)_2 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = 0.5 + 0.125 + 0.0625 = 0.625 + 0.0625 = 0.6875.$$

Fraction Examples

- Convert from base 10 to base 2, then verify by converting the result back to base 10.
  - 0.28125
  - 12.5625

Fraction Examples

- 0.28125:

$$0.28125 * 2 = 0.5625 + 0$$
$$0.5625 * 2 = 0.125 + 1$$
$$0.125 * 2 = 0.25 + 0$$
$$0.25 * 2 = 0.5 + 0$$
$$0.5 * 2 = 0 * 0 + 1$$

Answer: 0.01001

Verify: $(0.01001)_2 = \frac{1}{4} + \frac{1}{32} = 0.25 + 0.03125 = 0.28125$

- 12.5625:

$$0.5625 * 2 = 0.125 + 1$$
$$0.125 * 2 = 0.25 + 0$$
$$0.25 * 2 = 0.5 + 0$$
$$0.5 * 2 = 0 * 0 + 1$$

Also 12 in base 2 is 1100

Answer: 1100.1001

Verify: $(1100.1001)_2 = 8 + 4 + \frac{1}{2} + \frac{1}{16} = 12.5625$

Operations with Fractions

- Base-2 fractions work just like base-10 fractions
- Two’s complement addition/subtraction work the same as integer operations

$2^{-1} = 0.5$

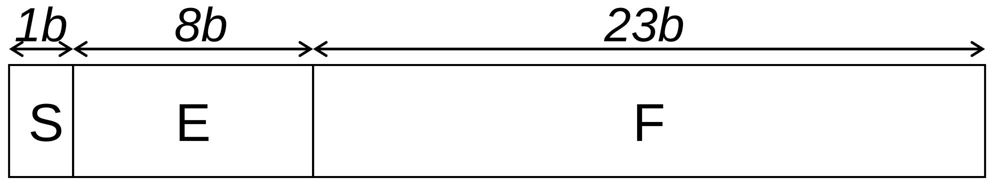
$2^{-2} = 0.25$

$2^{-3} = 0.125$

$$
\begin{array}{rl}
 & 0\,0\,1\,0\,1\,0\,0\,0\,.\,1\,0\,1 = 40.625 \\
+ & \underline{1\,1\,1\,1\,1\,1\,1\,0\,.\,1\,1\,0 =} -1.25 \\
 & 0\,0\,1\,0\,0\,1\,1\,1\,.\,0\,1\,1 = 39.375
\end{array}
$$

Floating Point Numbers

- IEEE 754 Floating Point Standard
- Based on scientific notation
  - Example (base 10): 6.02214 x 10^{23}
  - Normalized: one non-zero digit before decimal point
- Binary scientific notation: F x 2^E
  - Non-zero digit before binary point must be ‘1’ so no need to explicitly represent it
  - Magnitude: fraction (F) + sign (S) in sign-magnitude format
  - Exponent: value is *E-127* (excess 127 representation)



Single-precision (32-bit)
For double precision (64-bit), use 11 bits for exponent, 52 for fraction (mantissa)

$N = (-1)^S \times 1.F \times 2^{E-127}$, $1 \leq E \leq 254$
(E = 0, 255 are special cases)

Floating Point Example

Single-precision IEEE floating point number:

$$\underbrace{1}_{\uparrow\ \textit{sign}}\underbrace{01111110}_{\uparrow\ \textit{exponent}}\underbrace{10000000000000000000000}_{\uparrow\ \textit{fraction}}$$

- Sign is 1: number is negative.
- Exponent field: 01111110 = 126 (base 10).
- Fraction: 0.100000000000… = 0.5 (base 10).

- Value = -1.5 x $2^{(126-127)}$ = -1.5 x 2^{-1} = -0.75
- What is the reason for the E-127 offset (exponent bias)?
- Want to be able to represent both large and small numbers! With 8 bits, 2^8=256 possible exponents; half are small (<0), half large (>0).

Floating Point Numbers

- While we can find a base-10 representation for each floating-point number, not all real numbers can be represented exactly using floating-point: finite precision.
- In converting a real number to floating-point, we usually will have to round.
- Individual roundoff errors are small, but they can have large consequences, especially when accumulated.
- Hexadecimal can be a convenient way to write 32- or 64-bit numbers (8 or 16 hexadecimal digits).

Floating Point Examples

- Write -6.625 as a single-precision floating point number.

Floating Point Examples

- Write -6.625 as a single-precision floating point number.
  6: 4+2 = 110
  0.625: 0.5 + 0.125 = 0.101
  -6.625: -110.101

Floating Point Examples

- Write -6.625 as a single-precision floating point number.
  6: 4+2 = 110
  0.625: 0.5 + 0.125 = 0.101
  -6.625: -110.101
  Normalize: -1.10101 x 2^2
  Sign bit: 1 (because negative)
  Exponent: 2 = E-127 -> E = 129, which is 1000 0001
  Fraction: delete leading one to get 10101 (followed by 0's)

Floating Point Examples

- Write -6.625 as a single-precision floating point number.
  6: 4+2 = 110
  0.625: 0.5 + 0.125 = 0.101
  -6.625: -110.101
  Normalize: -1.10101 x 2^2
  Sign bit: 1 (because negative)
  Exponent: 2 = E-127 -> E = 129, which is 1000 0001
  Fraction: delete leading one to get 10101 (followed by 0's)
  Answer:
  1 1000 0001 10101 00 0000 0000 0000 0000 (18 zeros)

Floating Point Examples

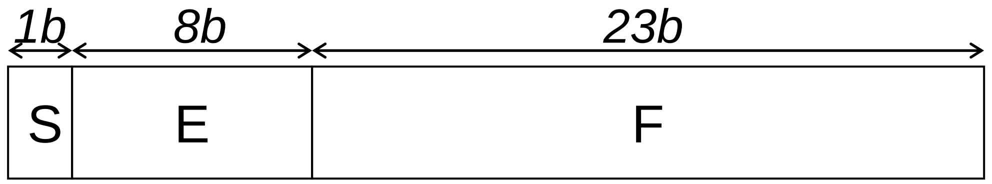
- What number does the single-precision value 0011 1101 1000 0000 0000 0000 0000 0000 represent?
- What number does the single-precision value 3FF0 0000 represent?

Floating Point Examples

- What number does the single-precision value
  0$\boxed{011\ 1101\ 1}$000 0000 0000 0000 0000 0000 represent?
  Sign bit: 0, positive
  Exponent: 0111 1011, which is $2^6+2^5+2^4+2^3+2^1+2^0$= 64+32+16+8+2+1 = 123. exponent is 123-127 = -4
  Fraction: all 0's
  Normalize: 1.0… (all zeros)
  Together: 1 x 2^{-4} = 1/16
- What number does the single-precision value 3F80 0000 represent?
  Expand to 0$\boxed{011\ 1111\ 1}$000 (followed by 20 zeros)
  Sign bit: 0, positive
  Exponent: 0111 1111, which is 2^7-1 = 127. exponent is 127-127=0.
  Fraction: all 0's
  Normalize: 1.0… (all zeros)
  Together: 1 x 2^0 = 1

Special Cases

- Exponent value of 0 (subnormal numbers)
  - Used to represent very small numbers by relaxing the normalization requirement



Usual (non-zero exponent): $N = (-1)^S \times 1.F \times 2^{E-127}$, $1 \leq E \leq 254$

Special case: $N = (-1)^S \times 0.F \times 2^{-126}$, $E = 0$

Example: 0 00000000 00001000000000000000000

Value = $+\ 0.00001 \times 2^{-126} = 2^{-131}$

Other special cases (exponent=255):

+infinity, -infinity, NaN (not a number, e.g., 0/0)

Other Data Types: Text Using ASCII Characters

ASCII: Maps 128 characters to 7-bit code

- both printable and non-printable (ESC, DEL, …) characters
- Successive numbers and letters have sequential codes

00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(	38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29	)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[	6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	|
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d	]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

What do these statements yield?

‘a’ + 2;

‘A’ + 5;

c – ‘0’
(c holds a digit character)

d + (‘a’ – ‘A’)
(d holds an upper case character)

Other Data Types

- Text strings
  - sequence of characters, terminated with NULL (0)
  - typically, no hardware support
- Image
  - array of pixels
    - monochrome: one bit (1/0 = black/white)
    - color: red, green, blue (RGB) components (e.g., 8 bits each)
    - other properties: transparency
  - hardware support:
    - typically none, in general-purpose processors
    - MMX instruction set (Intel): multiple 8-bit operations on 32-bit word
- Sound
  - sequence of fixed-point numbers

Final Comments on Representing Data

- Most computers use IEEE floating point representation
- Many real numbers cannot be represented exactly on computers
- There is a whole field (numerical analysis) to study the impacts of finite-precision arithmetic on accuracy of calculations