ECE260: Fundamentals of Computer Engineering

Data Representation & 2's Complement

James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania

Data Representation

- Internally, computers represent all data as binary
  - Provides a simple method to design and build hardware
    - A switch (i.e. a transistor) is either “on” or “off”
    - A wire is either charged or not charged
- All information is encoded as 1’s and 0’s
  - Characters
  - Integers (positive and negative numbers)
  - Non-integers (fixed-point and floating point numbers)
- Standards ensure interoperability between computers
  - 2’s Complement, ASCII, Unicode, IEEE Floating Point

American Standard Code for Information Interchange (ASCII)

- Common character encoding standard
- Available in 7-bit and extended 8-bit
  - 7-bit version encodes 2^7 (128) characters
  - 8-bit version encodes 2^8 (256) characters
- Not so great for languages based on non-English alphabets
- Unicode has replaced ASCII in many contexts
  - Backwards compatible with ASCII
  - Supports a much wider range of characters

智 ボ 🥓 🍩 🖖 🤮

Binary	Character	Binary	Character	Binary	Character	Binary	Character
00000000	NUL	00100000	SP	01000000	@	01100000	`
00000001	SOH	00100001	!	01000001	A	01100001	a
00000010	STX	00100010	"	01000010	B	01100010	b
00000011	ETX	00100011	#	01000011	C	01100011	c
00000100	EOT	00100100	$	01000100	D	01100100	d
00000101	ENQ	00100101	%	01000101	E	01100101	e
00000110	ACK	00100110	&	01000110	F	01100110	f
00000111	BEL	00100111	'	01000111	G	01100111	g
00001000	BS	00101000	(	01001000	H	01101000	h
00001001	HT	00101001	)	01001001	I	01101001	i
00001010	LF	00101010	*	01001010	J	01101010	j
00001011	VT	00101011	+	01001011	K	01101011	k
00001100	FF	00101100	,	01001100	L	01101100	l
00001101	CR	00101101	-	01001101	M	01101101	m
00001110	SO	00101110	.	01001110	N	01101110	n
00001111	SI	00101111	/	01001111	O	01101111	o
00010000	DLE	00110000	0	01010000	P	01110000	p
00010001	DC1	00110001	1	01010001	Q	01110001	q
00010010	DC2	00110010	2	01010010	R	01110010	r
00010011	DC3	00110011	3	01010011	S	01110011	s
00010100	DC4	00110100	4	01010100	T	01110100	t
00010101	NAK	00110101	5	01010101	U	01110101	u
00010110	SYN	00110110	6	01010110	V	01110110	v
00010111	ETB	00110111	7	01010111	W	01110111	w
00011000	CAN	00111000	8	01011000	X	01111000	x
00011001	EM	00111001	9	01011001	Y	01111001	y
00011010	SUB	00111010	:	01011010	Z	01111010	z
00011011	ESC	00111011	;	01011011	[	01111011	{
00011100	FS	00111100	<	01011100	\	01111100	|
00011101	GS	00111101	=	01011101	]	01111101	}
00011110	RS	00111110	>	01011110	^	01111110	~
00011111	US	00111111	?	01011111	_	01111111	DEL

Encoding Numbers

- Numbers can be represented in any base
  - Humans typically use base 10 (***decimal***) so we can count on our fingers
  - Programmers often represent data in base 16 (***hexadecimal***)
  - Computers represent all data using base 2 (***binary***)
- In any base, the decimal value of the i^{th} digit d can be calculated as: $d \times Base^i$
  - Sum the decimal values of each digit to determine the total value: $\sum_{i=0}^{i=\#digits} (d \times Base^i)$

base 10

5 3 8 = $\mathbf{538_{ten}}$

2 1 0 digit index

(5×10^2)	+	(3×10^1)	+	(8×10^0)
(5×100)	+	(3×10)	+	(8×1)
500	+	30	+	8

base 2

1 0 1 1 = $\mathbf{11_{ten}}$

3 2 1 0 digit index

(1×2^3)	+	(0×2^2)	+	(1×2^1)	+	(1×2^0)
(1×8)	+	(0×4)	+	(1×2)	+	(1×1)
8	+	0	+	2	+	1

Unsigned Integers

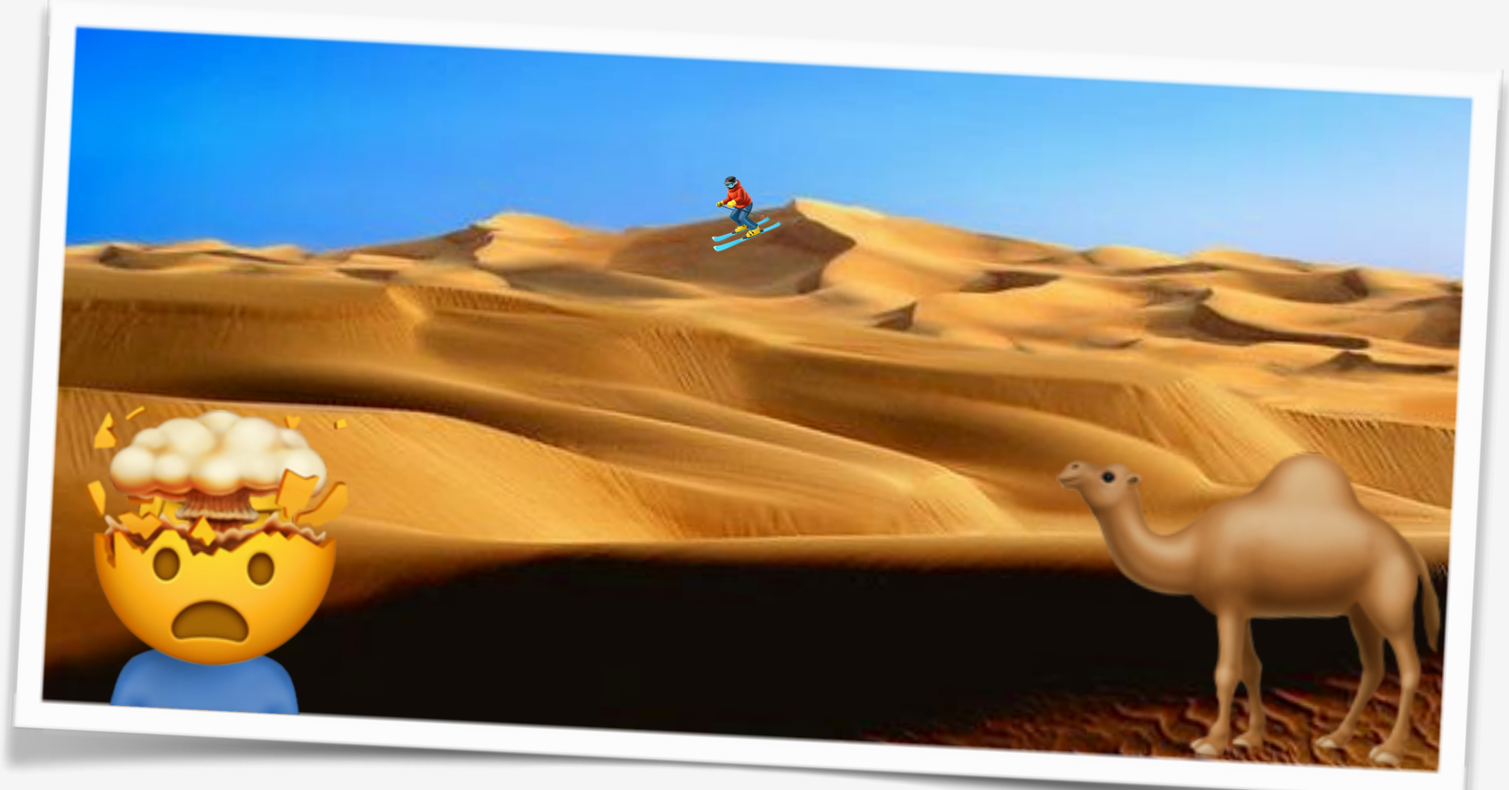
- Represented as binary value (a bit string)
  - Leftmost bit is called the ***most significant bit (MSB)***
  - Rightmost bit is called the ***least significant bit (LSB)***

MSB LSB

32-bit Unsigned Integer

digit	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
index	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- The number of bits is determined by the ***bitness*** of the computer architecture
  - A 32-bit computer can represent unsigned integers from 0 to $2^{32} - 1$
    - 0 to 4,294,967,295 (4.29 Billion)
  - A 64-bit computer can represent unsigned integers from 0 to $2^{64} - 1$
    - 0 to 18,446,744,073,709,551,615 (18.4 Quintillion)



Arithmetic Limitations & Overflow

- The largest unsigned value a 32-bit computer can represent is ***4,294,967,295***
- What happens when user requests ***4,294,967,295 + 1***?

index		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

↑
OVERFLOW

- CPUs have a special ***status register*** with an ***overflow flag*** to indicate when this happens

2's Complement Signed Integers

- Must be able to represent both positive AND negative numbers
- Represented as binary value where the MSB is now a ***sign bit***
  - When sign bit is 1, the signed integer is a negative number
  - When sign bit is 0, the signed integer is a positive number

MSB / Sign Bit

LSB

32-bit Signed Integer

digit	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
index	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Maximum value less than that of unsigned integers since 1 bit is being used to represent the sign
  - A 32-bit computer can represent signed integers from -2^{31} to $2^{31} - 1$
    - –2,147,483,648 to 2,147,483,647

2's Complement Signed Integer Examples

- **Positive Number Examples**

2,147,483,647	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
247	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- **Negative Number Examples**

–1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
–2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
–3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
–2,147,483,648	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Arithmetic Overflow with Signed Integers

- The largest signed value a 32-bit computer can represent is ***2,147,483,647***
- What happens when user requests ***2,147,483,647 + 1***?

index	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

OVERFLOWS

2,147,483,647 + 1 = ~~–2,147,483,648~~

- Similar badness happens if you subtract 1 from –2,147,483,648

Binary to Decimal Conversion for Signed Integers

- Similar to conversion with unsigned integers, but sign bit is multiplied by -2^{31} instead of 2^{31}
  - If the signed integer is positive => $(0 \times -2^{31}) = 0$
  - If the signed integer is negative => $(1 \times -2^{31}) = -2,147,483,648$

For a 32-bit 2's complement integer, compute decimal value From binary as follows:

$$(d \times -2^{31}) + \sum_{i=0}^{i=30} (d \times 2^i)$$

MSB / Sign Bit

LSB

32-bit Signed Integer

digit	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1
index	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

(1×-2^{31})	+	...	+	(1×2^5)	+	(0×2^4)	+	(1×2^3)	+	(1×2^2)	+	(0×2^1)	+	(1×2^0)
	+	...	+	(1×32)	+	(0×16)	+	(1×8)	+	(1×4)	+	(0×2)	+	(1×1)
–2,147,483,648	+	0	+	32	+	0	+	8	+	4	+	0	+	1

= –2,147,483,603

Negating 2’s Complement Signed Integers

- ***Step #1:*** Toggle every bit, including the sign bit (i.e. all 1’s become 0’s and all 0’s become 1’s)
- ***Step #2:*** Add 1 to result of step #1
- Example: Negate the integer +2

2_{ten}		0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 0
First, toggle every bit		1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 0 1
Then, Add 1	+	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1
		1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 0

$= -2_{ten}$

Sign Extension for 2's Complement Signed Integers

- Must sign extend when increasing the number of bits used to represent a ***signed integer***
  - Must ***extend sign bit*** to preserve the numeric value (i.e. fill all leading zeros with sign bit)
- Example: Loading a signed 8-bit value into a 32-bit register
  - If bits [31:9] of register are left as 0's then sign of 8-bit value may be lost