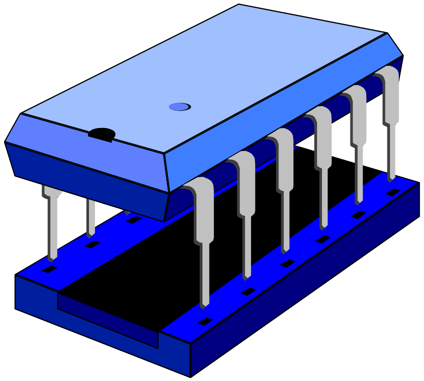
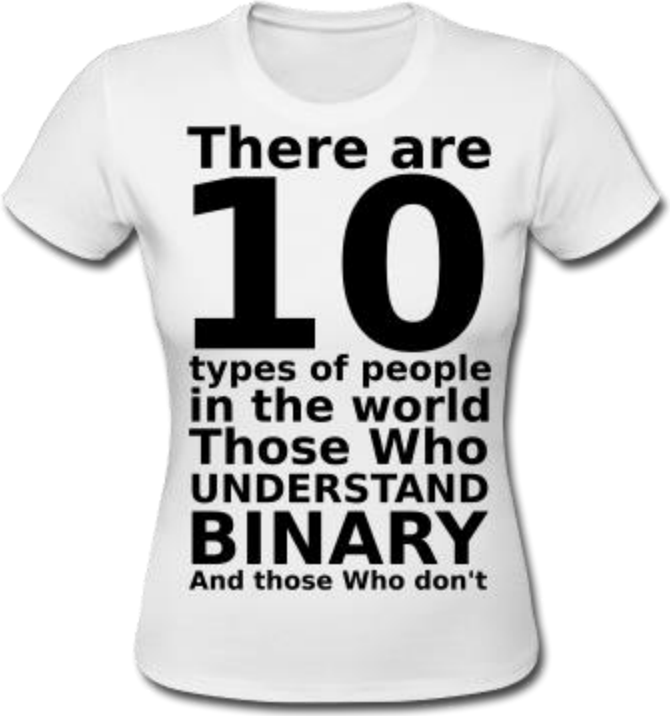


DATA REPRESENTATION

Anandha Gopalan (with thanks to **N. Dulay** and **E. Edwards**)
axgopala@imperial.ac.uk

Common statement



Why Binary Numbers?

- Computers process binary patterns
  - Patterns of *0s* and *1s*

  - To represent data within a computer, we need to code it as a binary pattern

  - Most important to consider representing numbers and characters

    - Convert into binary

Decimal to Binary

- Steps:
  - Divide the number by 2 giving the quotient and the remainder
  - Repeat previous step with the new quotient until a zero quotient is obtained
  - Answer is obtained by reading the remainder column **bottom to the top**

Decimal to Binary (Example)

- What is 98_{10} in binary?

	Quotient	Remainder
98 ÷ 2	49	0
49 ÷ 2	24	1
24 ÷ 2	12	0
12 ÷ 2	6	0
6 ÷ 2	3	0
3 ÷ 2	1	1
1 ÷ 2	0	1

1100010_2

$$1100010_2 = 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$
$$= 64 + 32 + 0 + 0 + 0 + 2 + 0 = 98_{10}$$

Octal (Base 8)

- Used in the past as a more convenient base for representing long binary values
- Converting to binary
  - Starting from the rightmost (least significant) end, each group of 3 bits (why? $8 = 2^3$) represents 1 octal digit (called octet)
- Example: What is 10101_2 in Octal?

10	101
↓	↓
2	5

$= 25_8$

- Example: What is 357_8 in Binary?

3	5	7
↓	↓	↓
011	101	111

$= 11101111_2$

Hexadecimal (Base 16)

- Used by programmers to represent long binary values
  - Preferred over Octal
- $16 = 2^4$ ➔ 4 Binary digits represent one hexadecimal digit (bits) - starting from the rightmost end, each group of 4 bits represents 1 hexadecimal digit
- Example: What is 10010100_2 in hexadecimal?

1001	0100
↓	↓
9	4

$= 94_{16}$

- Example: What is 86_{16} in Binary?

8	6
↓	↓
1000	0110

$= 10000110_2$

Binary vs. Hexadecimal

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

- Generally:

	1 byte =	8 binary digits =	2 hexadecimal digits
1 word =	2 bytes =	16 binary digits =	4 hexadecimal digits
1 long word =	4 bytes =	32 binary digits =	8 hexadecimal digits

Representing Data

- Data Types of interest
  - Integers (Unsigned/Signed)
  - Reals (Floating Point) → later on in the course
  - Text

Signed and Unsigned Integers

- Natural numbers can be represented by their binary value within the computer
- Representation of signed integers is more important
- Several possibilities:
  - Sign & Magnitude
  - One's Complement
  - Two's Complement
  - Excess-n (Bias-n)
  - Binary-Coded Decimal (BCD)

Signed and Unsigned Integers

- In any representation, desirable properties are:
  - Only one bit-pattern per value
  - Equal number of positive and negative values
  - Maximum range of values
  - No gaps in the range
  - Fast, economic hardware implementation of integer arithmetic
    - Minimal number of transistors AND fast arithmetic, if possible

Sign & Magnitude

- Leftmost ("most significant") bit represents the **sign** of the integer
- Remaining bits to represent its **magnitude**
- For n-bits, $-(2^{n-1}-1) \le$ Sign & Magnitude $\le +(2^{n-1}-1)$
- Simplest for humans to understand
- Two representations for zero → +0 and -0
- Costly to implement (need to compare signs and implement subtractors)

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign & Magnitude	+0	+1	+2	+3	+4	+5	+6	+7	−0	−1	−2	−3	−4	−5	−6	−7

One’s Complement

- Negative numbers are the **complement** of the positive numbers
- $-(2^{n-1}-1) \leq$ One's Complement $\leq + (2^{n-1}-1)$
  - Same as Sign & Magnitude
- Less intuitive (for humans) than Sign & Magnitude
- Less costly to implement
- Bit fiddly

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign & Magnitude	+0	+1	+2	+3	+4	+5	+6	+7	−0	−1	−2	−3	−4	−5	−6	−7
1s Complement	+0	+1	+2	+3	+4	+5	+6	+7	-7	-6	-5	-4	-3	-2	-1	-0

Two's Complement

- Only **one** bit pattern for zero ☺
  - Asymmetric - one *extra* negative value

- Minor disadvantage outweighed by the advantages

- $-2^{n-1} \leq$ Two's complement $\leq 2^{n-1}-1$

- Most useful property: $X - Y = X + (-Y)$
  - No need for a separate subtractor (Sign & Magnitude) or carry-out adjustments (One's Complement)

Two's Complement

- Negative of an integer is achieved by inverting each of the bits and adding 1 to it:
  - Two's complement of -3 (0011) → 1100 (invert) + 1 → 1101

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign & Magnitude	+0	+1	+2	+3	+4	+5	+6	+7	−0	−1	−2	−3	−4	−5	−6	−7
1s Complement	+0	+1	+2	+3	+4	+5	+6	+7	-7	-6	-5	-4	-3	-2	-1	-0
2s Complement	+0	+1	+2	+3	+4	+5	+6	+7	-8	-7	-6	-5	-4	-3	-2	-1

Excess-n (Bias-n) - Motivation

- Sorting in Two’s complement is not easy
  - Assuming you could compare numbers, it would always say negative numbers are greater !!!
- Suppose we wanted to represent negative numbers, but wanted to keep the same ordering where 000 represents the smallest value and 111 represents the largest value in 3-bits?
  - This is the idea behind *excess representation* or *biased representation*
  - bitstring with N 0's maps to the smallest value and the bitstring with N 1's maps to the largest value

Excess-n (Bias-n)

- Using 3-bits as example, 3-bits gives us: $2^3 = 8$ values in total
  - Assuming we start at -4 (1/2 of 8), we get: -4, -3, -2, -1, 0, 1, 2, 3
  - Smallest value = -4, so we *shift* by 4
    - Each value stored is +4 (excess of 4) of actual value ➔ Excess-4 ☺

Stored value	Actual value
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

Excess-n (Bias-n)

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign & Magnitude	+0	+1	+2	+3	+4	+5	+6	+7	−0	−1	−2	−3	−4	−5	−6	−7
1s Complement	+0	+1	+2	+3	+4	+5	+6	+7	-7	-6	-5	-4	-3	-2	-1	-0
2s Complement	+0	+1	+2	+3	+4	+5	+6	+7	-8	-7	-6	-5	-4	-3	-2	-1
Excess-8	-8	-7	-6	-5	-4	-3	-2	1	0	1	2	3	4	5	6	7

Binary Coded Decimal

- Each decimal digit is represented by a fixed number of bits, usually four or eight
- Easy for humans to understand
- Takes up much more space
- Assuming 4-bits, the number 9876510 can be encoded as:

9	8	7	6	5	1	0
1001	1000	0111	0110	0101	0001	0000

- Actual Binary: 100101101011010000011110 (24-bits)

Binary Coded Decimal (BCD)

Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign & Magnitude	+0	+1	+2	+3	+4	+5	+6	+7	−0	−1	−2	−3	−4	−5	−6	−7
1s Complement	+0	+1	+2	+3	+4	+5	+6	+7	-7	-6	-5	-4	-3	-2	-1	-0
2s Complement	+0	+1	+2	+3	+4	+5	+6	+7	-8	-7	-6	-5	-4	-3	-2	-1
Excess-8	-8	-7	-6	-5	-4	-3	-2	1	0	1	2	3	4	5	6	7
BCD	0	1	2	3	4	5	6	7	8	9	-	-	-	-	-	-

Characters

- Characters are mapped to bit patterns
- Common mappings are ASCII and Unicode
- ASCII
  - Uses 7-bits (128 bit-patterns)
  - Most modern computer extend this to 8-bits yielding an extra 128 bit-patterns
  - 26 lowercase and uppercase letters, 10 digits, and 32 punctuation marks. Remaining 34 bit-patterns represent whitespace characters e.g. space (SP), tab (HT), return (CR), and special *control characters*

ASCII Character Set

Bit positions 654								Bit positions 3210
000	001	010	011	100	101	110	111	
NUL	DLE	SP	0	@	P	‘	p	0000
SOH	DC1	!	1	A	Q	a	q	0001
STX	DC2	“	2	B	R	b	r	0010
ETX	DC3	#	3	C	S	c	s	0011
EOT	DC4	$	4	D	T	d	t	0100
ENQ	NAK	%	5	E	U	e	u	0101
ACK	SYN	&	6	F	V	f	v	0110
BEL	ETB	‘	7	G	W	g	w	0111
BS	CAN	(	8	H	X	h	x	1000
HT	EM	)	9	I	Y	i	y	1001
LF	SUB	*	:	J	Z	j	z	1010
VT	ESC	+	;	K	[	k	{	1011
FF	FS	,	<	L	\	l	|	1100
CR	GS	-	=	M	]	m	}	1101
SO	RS	.	>	N	^	n	~	1110
SI	US	/	?	O	_	o	DEL	1111

Strings are represented as sequence of characters. E.g. **Fred** is encoded as follows:

English	F	r	e	d
ASCII (Binary)	0100 0110	0111 0010	0110 0101	0110 0100
ASCII (Hex)	46	72	65	64

Unicode

- Newer, more complex standard
- Attempting to provide a number for every character, no matter the language ☺
- Over 100,000 characters already defined
- First 65,536 (16-bit) characters cover the major alphabets of the world – more and more programming languages support this
- First 127 characters correspond to ASCII characters