UNIT 7A

Representing Numbers

Reminder

- Lab exam is Monday next week
- PA 6 is due Sunday

Digital Data

10010101011110101010110101001110

- What does this binary sequence represent?
- It could be:
  - an integer
  - a floating point number
  - text encoded with ASCII or another standard
  - a pixel of an image
  - several digital samples of a music recording
  - an instruction that the computer is executing
  - ...

New Unit: Representation

- Issue: we use computers to model, i.e., *represent*, things in the real world
  - numbers, pictures, music, climates, markets, …

- Three lectures:
  - representing numbers
  - exploiting redundancy in representations
  - representing images and sound

REMEMBER THE FIRST DAY

At the very basics

This is a series of words that is called a ‘sentence’.

- A ‘word’ is a series of letters.
- What makes a letter different from another?

- Agreed forms are used to build words then sentences then paragraphs ...

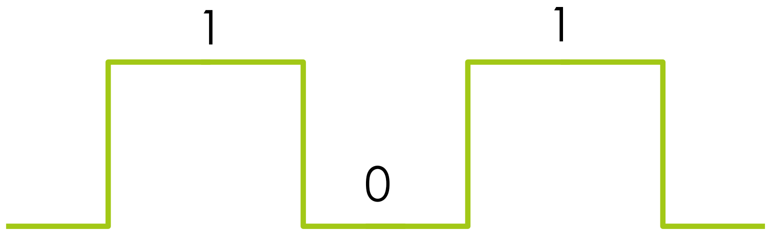
  Simple symbols → combinations → complexity and higher level

At the very basics

- On – Off
- Yes – No
- True – False
- Correct – Wrong

At the very basics

- On – Off
- 1 – 0 ← Electric pulses (High - Low)



At the very basics

- Ligth is on
- Light is off

- First light is on
- First light is off
- Second light is on
- Second light is off

How many options

2 x 2 x 2 x 2 = 16 options

How many options

2x2x2x2 = 16 options

Down (Off) → 0   Up (On) → 1

0 0 0 0	0 1 0 0	1 0 0 0	1 1 0 0
0 0 0 1	0 1 0 1	1 0 0 1	1 1 0 1
0 0 1 0	0 1 1 0	1 0 1 0	1 1 1 0
0 0 1 1	0 1 1 1	1 0 1 1	1 1 1 1

Let's say 'HI' to digital world

237 =

2	3	7
100s	10s	1s

2 hundreds + 3 tens + 7 ones

1	0	1	1
8s	4s	2s	1s

1 Eight + 0 Four + 1 Two + 1 One = **11**

Say 'HI' to digital world

0	1	0	0	1	0	0	0
128	64	32	16	8	4	2	1

72

0	1	0	0	1	0	0	1
128	64	32	16	8	4	2	1

73

↓

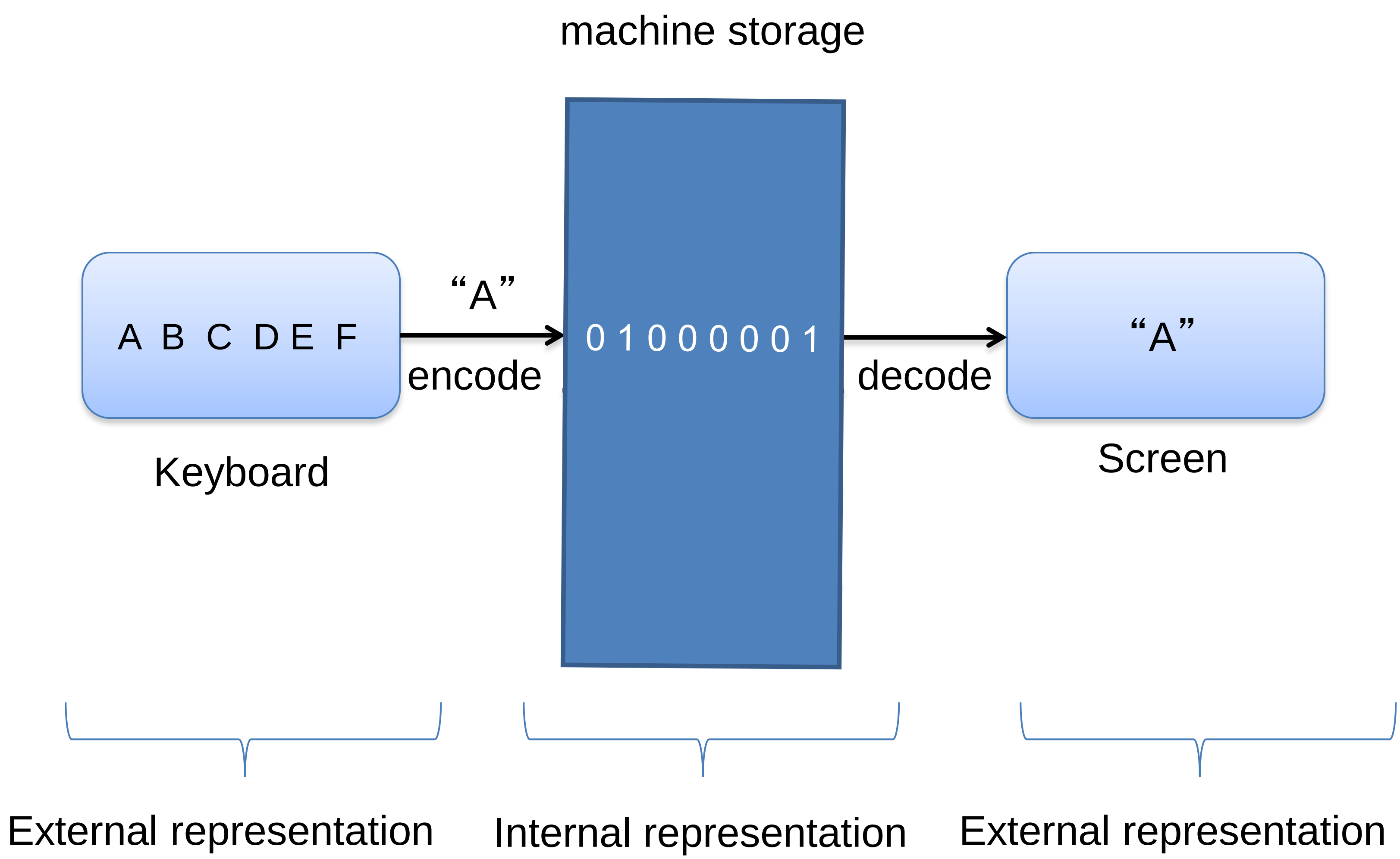
H I

01000000	64	@
01000001	65	A
01000010	66	B
01000011	67	C
01000100	68	D
01000101	69	E
01000110	70	F
01000111	71	G
01001000	72	H
01001001	73	I
01001010	74	J

first, what do we mean by

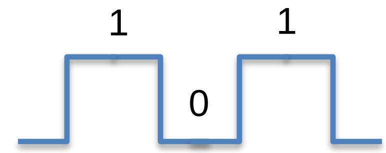
REPRESENTATION?

Representing Data



Digital Data

- Inside the digital machine it's all just
  - **binary** physical states (high or low voltages, etc.)
  - which we **interpret** as bits (1s and 0s)



- In turn we interpret these bits as representing data such as integers, real numbers, text, …

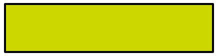
- Machine storage is finite and divided into fixed-size chunks of bits
  - **bytes**, usually **8 bits**
  - **words**, usually **64 or 32 bits**
  - machine storage capacity usually expressed as number of bytes or words
  - loosely speaking: "**memory size**"

Types interpret bits

- a 32-bit "word" might be

  1100 1100 1011 0111 0000 0000 0000 0000

- what this means depends on the machinery to interpret it, could be (**explore with 0xED**)

Type	Interpretation
“Raw” bits	1100 1100 1011 0111 0000 0000 0000 0000
Floating point number	6.59339×10^{-41}
String (Unicode UTF-16)	첷
RGB pixel color	
Little-endian integer	47052

Fundamental Issue: Information Capacity

# bits	Possible values	# possible values
1	0 1	2
2	00 01 10 11	4
3	000 001 010 011 100 101 110 111	8
4	0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111	16

$2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, ..., ..., ... 2^k

k bits can represent $\mathbf{2^k}$ different values.

don’t be drawn like moths to the flame of meaning*:

NUMERALS ARE NOT NUMBERS!

*Geoffrey Pullum

Numbers: semantics (quantities) versus syntax (numerals)

	Semantics	Syntax
What is it?	Our idea of quantity	How we write our idea of quantity
What is it good for?	Insight	Calculation, communication, computation
Example	 	**II (Roman numeral)** **2 (decimal Arabic numeral)** **10 (binary numeral)** – all with the same semantics!

machines don't have ideas!

only syntax!

Place-value numerals (base 10)

- The *numeral* we write*:* 15627
- What it means:

  $7 \times 10^0 + 2 \times 10^1 + 6 \times 10^2 + 5 \times 10^3 + 1 \times 10^4$

Problem:
Electronic circuitry for base-10 arithmetic is slow.

Solution:
Use place-value numerals,
but in base 2–*binary notation*

Place-value numerals in general

- Choose a number *b* for the **base** or **radix**
- Choose list of **digits**, there must be *b* of them
  - **base 10 example: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
  - **base 2 example: 0, 1**
  - **base 16 example: 0, 1, …, 9, A, B, C, D, E, F**

- To represent a quantity *n* in base *b*
  - integer divide *n* by *b* with remainder *r* (a **digit**)
  - repeat until the quotient is zero
  - the remainders are the digits in reverse order

Binary place-value example

- Base two, digits 0 and 1

- To represent “six”:
  - 6 // 2 = 3 remainder 0

remainder when dividing by 2 can only be 0 or 1

Binary place-value example

- Base two, digits 0 and 1

- To represent "six":
  - 6 // 2 = 3 remainder 0
  - 3 // 2 = 1 remainder 1

Binary place-value example

- Base two, digits 0 and 1

- To represent "six":
  - 6 // 2 = 3 remainder **0**
  - 3 // 2 = 1 remainder **1**
  - 1 // 2 = 0 remainder **1**

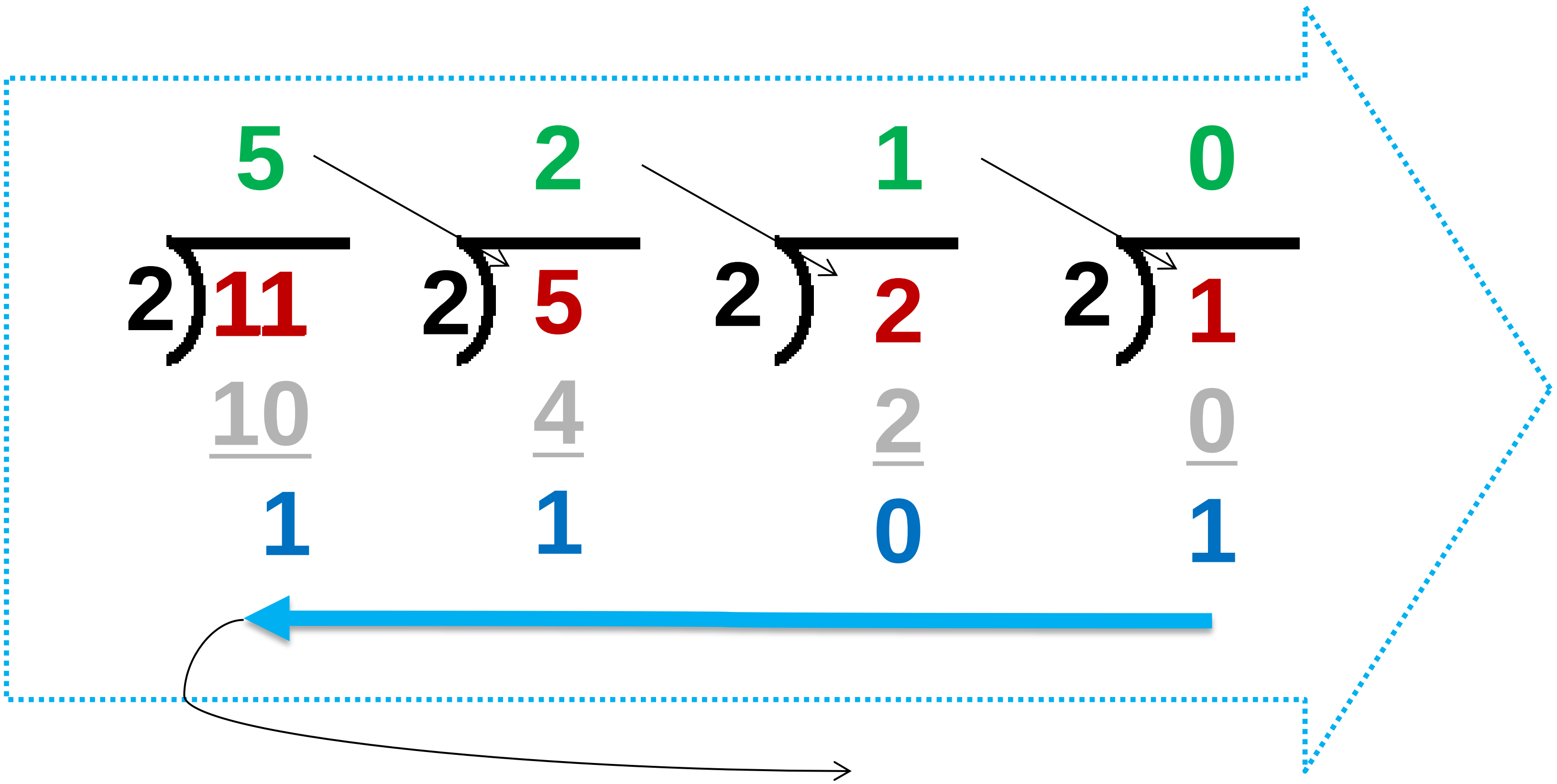
Read the remainders from bottom to top to get bits from left to right

Binary numeral: 110

- What it means:

$$0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 = \text{"six"}$$

Representing 11 as binary numeral



11(base 10) → **1011** (base 2)

Information Capacity and Range

- Remember:
  k bits can represent 2^k different things

- So *k*-bit binary numerals represent $0 \ldots 2^k-1$
  - For $k = 3$,

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7

Ranges for typical computer "word" sizes

bits	minimum	maximum	
8	0	$2^8 - 1$	(255)
16	0	$2^{16} - 1$	(65,535)
32	0	$2^{32} - 1$	(4,294,967,295)
64	0	$2^{64} - 1$	(18,446,744,073,709,551,615)

some familiar operations

BINARY ARITHMETIC

Counting in binary

Binary Numerals		Decimal Equivalents
0	0	
1	1	
10	2	
11	3	
100	4	
101	5	
110	6	
111	7	
1000	8	
1001	9	
1010	10	
1011	11	

Addition and Multiplication Tables

+	0	1
0	0	1
1	1	10

×	0	1
0	0	0
1	0	1

Binary Arithmetic

- All the familiar methods work, but with only 1 and 0 for digits
- 1 + 1 = 10, 10 - 1 = 1, 10 + 1 = 11, ...
- Example:

```
 1  1
  1010
 +1010
 -----
 10100
```

Notice: We need more bits for the answer than we did for the operands.

Overflow: the first difficulty

- Machine word only has *k* bits for some **fixed** *k!*
- If *k* is 4, then we have **overflow** in the following:

```
1  1
 1010
+1010
-----
10100
```

- The machine retains only 0100 (the “least significant” bits),
  so (n+n) – n **not** always equal to n + (n – n)

Modular Arithmetic

- Dropping the overflow bit is **modular arithmetic**
- We can carry out any arithmetic operation modulo 2^k for the precision *k.* The example again for precision 4:

binary	**decimal**
1 0 1 0	= 10
+ 1 0 1 0	= 10
(1) 0 1 0 0	= (20)
0 1 0 0	= 4 (= 20 mod 2^4 = 20 mod 16)

overflow can be ignored or signaled as an error representing all the integers, including

NEGATIVE INTEGERS

Representing a sign +/-

- A natural idea: reserve one of the bits to stand for a sign.

- E.g., **0** could stand for **+** and **1** could stand for **–**
  - unsigned “ten” is 1010
  - so “negative ten” would be 11010
- But someone had a cleverer idea…
  - first, we’d like to avoid “two zeroes”: +0 and -0
  - second, we’d like the same machinery to work for addition and subtraction

Two's Complement Negative Numbers

- A clever approach based on modular arithmetic
- Remember, with k bits, we do arithmetic mod 2^k
- We define negative numbers as *additive inverse*: $-x$ is the number y such that $x + y = 0 \bmod 2^k$ – this is the **two's complement of *x***
- *Example with 4 bits:* if 1 is 0001, what is -1?

```
carry bits     1      11    111   1111
    0001     0001   0001  0001  0001   0001
  + ????    +???1  +??11 +?111 +1111  +1111
    ----     ----   ----  ----  ----   ----
    0000     ???0   ??00  ?000  0000  10000
```

representation for -1

modular arithmetic discards overflow

Two's complement property

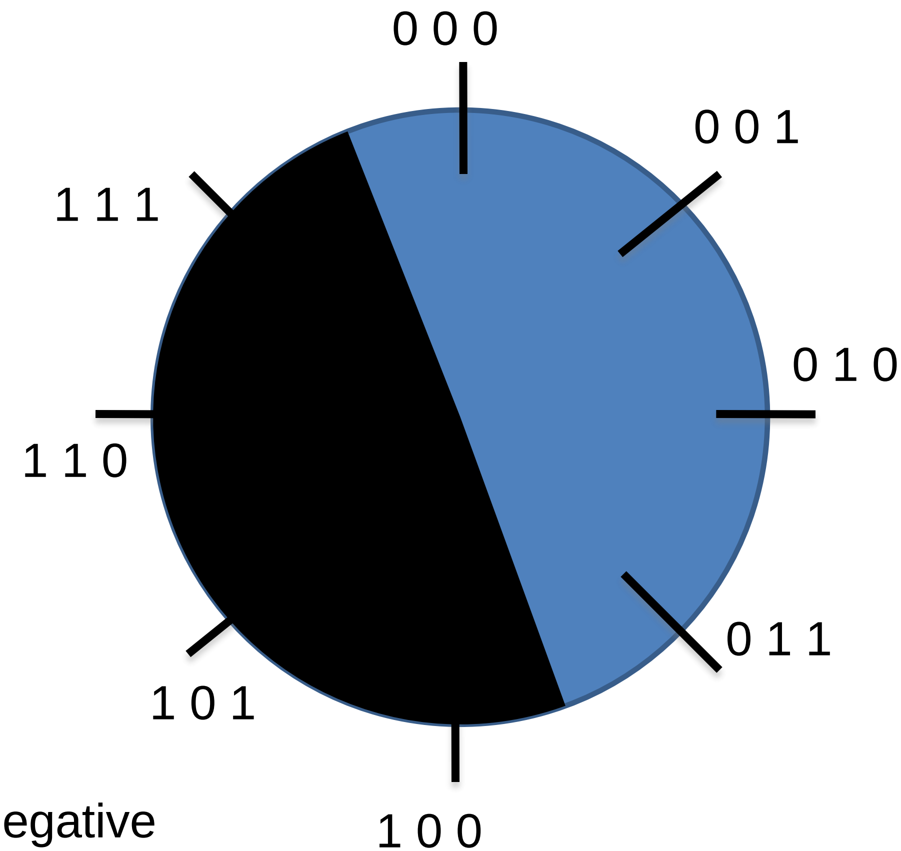
- When you add a number to its two's complement (modulo 2^k), you always get 0.
  - That's why we use it to represent negative numbers!
  - Remember, you're using base 2 arithmetic.

Example (using 3 bits):

```
      011       (+3 in decimal)
+     101       (-3 in decimal)
   ------------------
   (1)000         0
```

modular arithmetic discards

All two's complement integers using 3 bits, arithmetic mod 8



positive integers and zero

negative integers

Bit pattern	Decimal value
0 0 0	0
0 0 1	+ 1
0 1 0	+ 2
0 1 1	+ 3
1 0 0	- 4
1 0 1	- 3
1 1 0	- 2
1 1 1	- 1

Adding + n to – n gives 0
For example: 011 + 101 = 000

Great! but how do we "read" two's complement integers?

Sign: look at leftmost bit

- **1 means negative, 0 means positive**

  e.g. with four bits 1010 represents a negative number

Magnitude: if negative, compute the two's complement

flip each bit (one's complement) *e.g.* flip 1010 to get 0101

then add 1 *e.g.* 0101 + 0001 = 0110, or

$0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 = 6$

- **voilà!** 1010 represents negative six

Another Example

What value is this 8-bit signed integer?

sign bit

	1	1	0	0	1	1	0	0	
	0	0	1	1	0	0	1	1	Flip each bit
+	0	0	0	0	0	0	0	1	Add one
	0	0	1	1	0	1	0	0	

$2^5 \quad 2^4 \quad 2^2$

$32 + 16 + 4 = 52$

two's complement

So 11001100 represents -52

so we can “decode” binary signed integers, now for

ENCODING SIGNED INTEGERS

Signed Integers: encoding negative values

Example: How do you store -52 in 8 bits?
Start by encoding +52:

One way to do it: by repeated integer division

52 // 2 = 26 r 0
26 // 2 = 13 r 0
13 // 2 = 6 r 1
6 // 2 = 3 r 0
3 // 2 = 1 r 1
1 // 2 = 0 r 1

00110100

Another way: find the powers of two that add up to 52:

52 = 32 + 16 + 4

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	1	0	1	0	0

Signed Integers: encoding negative values

Example continued: How do you store -52 in 8 bits?

We've encoded +52 like this:

52 = 32 + 16 + 4

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	1	0	1	0	0

Flip each bit (one's complement):

1	1	0	0	1	0	1	1

Add 00000001, modulo 2^8:

1	1	0	0	1	1	0	0	= -52

The same steps convert positive to negative and vice-versa! (try it and see)

2's complement property

- When you add a number to its 2's complement (in binary), you always get 0.
  - Remember, you're using base 2 arithmetic.

- Example (using 8 bits):

```
     00110100        +52
+    11001100        -52
----------------------------
     00000000          0
```

reminder

ENCODING CHARACTERS AS 7-BIT INTEGERS

ASCII table

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

- 2^7 (128) characters
- 7 bits needed for binary representation
- (Not shown: control characters like tab and newline, values 0…31)

ASCII table

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

- Values above are represented in hexadecimal (base 16).
- ASCII code for “M” is 4D (hex).

ASCII Example

- The ASCII code for “M” is 4D hexadecimal.
- Conversion from base 16 to base 2:

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

- **4D (hex) = <u>0100</u> <u>1101</u> (binary) = 77 (decimal)**

  (leftmost bit can be used for parity)

Python tools for character codes

```
>>> ord('a')
97
>>> chr(97)
'a'
>>>
```

appendix

SOME SKILLS YOU SHOULD HAVE

You should be able to

- Count in unsigned binary
  0, 1, 10, 11, 100, …

- Add in binary and know what overflow is

- Determine the sign and magnitude of an integer represented in two’s complement binary

- Determine the two’s complement binary representation of a positive or negative integer appendix

PYTHON AIDS

Some Helpful Python functions

```
>>> bin(10)
'0b1010'
>>> hex(10)
'0xa'
>>> from decimal import Decimal
>>> Decimal(.2)
Decimal('0.200000000000000011102230246251565404236316680908203125')
```

Next Time

Data Compression
Data Compression
Data Compression
Data Compression
Data Compression
Data Compression