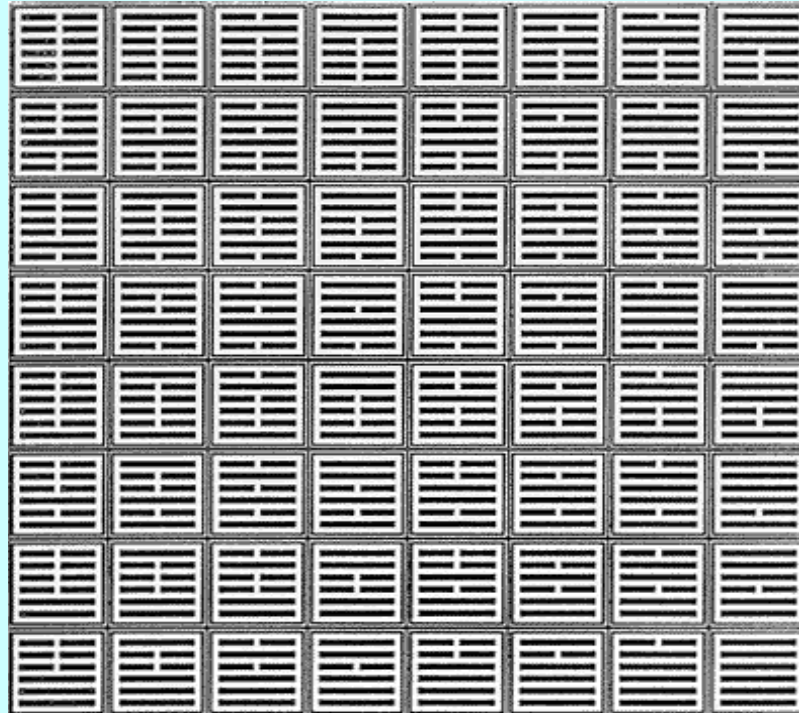
Binary Representation and Strings



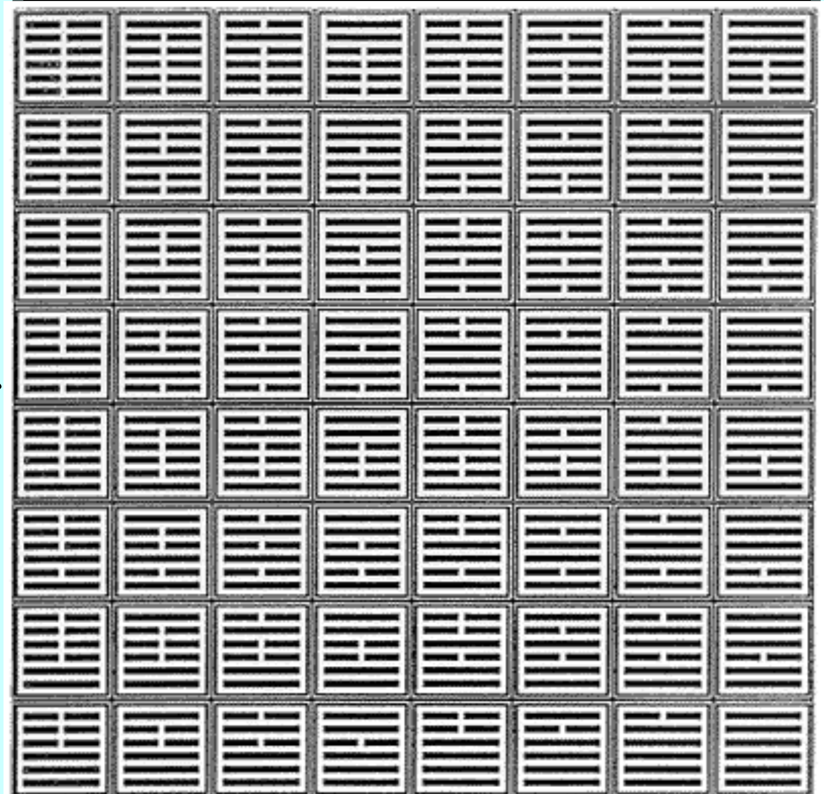
Chris Gregg, based on slides by Eric Roberts
CS 208E
October 9, 2018

The Power of Bits

- The fundamental unit of memory inside a computer is called a ***bit***—a term introduced in a paper by Claude Shannon as a contraction of the words *binary digit*.
- An individual bit exists in one of two states, usually denoted as **0** and **1**.
- More sophisticated data can be represented by combining larger numbers of bits:
  - Two bits can represent four (2×2) values.
  - Three bits can represent eight ($2 \times 2 \times 2$) values.
  - Four bits can represent 16 (2^4) values, and so on.
- This laptop has 16GB of main memory and can therefore exist in $2^{549,755,813,888}$ states. If you were to write that number out, it would contain more than fifty billion digits.

Leibniz and Binary Notation

- Binary notation is an old idea. It was described back in 1703 by the German mathematician Gottfried Wilhelm von Leibniz.
- Writing in the proceedings of the French Royal Academy of Science, Leibniz describes his use of binary notation in a simple, easy-to-follow style.
- Leibniz's paper further suggests that the Chinese were clearly familiar with binary arithmetic 2000 years earlier, as evidenced by the patterns of lines found in the *I Ching*.



qu'elle sert à la perfection de la science des Nombres. Ainsi je n'y employe point d'autres caracteres que 0 & 1, & puis allant à deux, je recommence. C'est pourquoi *deux* s'écrit ici par 10, & deux fois deux ou *quatre* par 100; & deux fois quatre ou *huit* par 1000; & deux fois huit ou *seize* par 10000, & ainsi de suite. Voici *la Table des Nombres* de cette façon, qu'on peut continuer tant que l'on voudra.

Using Bits to Represent Integers

- Binary notation is similar to decimal notation but uses a different ***base***. Decimal numbers use 10 as their base, which means that each digit counts for ten times as much as the digit to its right. Binary notation uses base 2, which means that each position counts for twice as much, as follows:

0	0	1	0	1	0	1	0

Bit	x	Value	=	Result
0	x	1	=	0
1	x	2	=	2
0	x	4	=	0
1	x	8	=	8
0	x	16	=	0
1	x	32	=	32
0	x	64	=	0
0	x	128	=	0
				42

Numbers and Bases

- The calculation at the end of the preceding slide makes it clear that the binary representation 00101010 is equivalent to the number 42. When it is important to distinguish the base, the text uses a small subscript, like this:

$$00101010_2 = 42_{10}$$

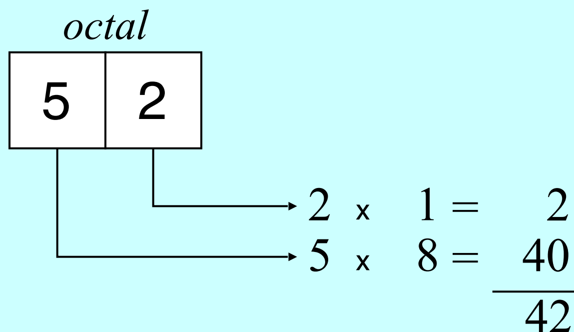
- Although it is useful to be able to convert a number from one base to another, it is important to remember that the number remains the same. What changes is how you write it down.
- The number 42 is what you get if you count how many stars are in the pattern at the right. The number is the same whether you write it in English as *forty-two,* in decimal as 42, or in binary as 00101010.

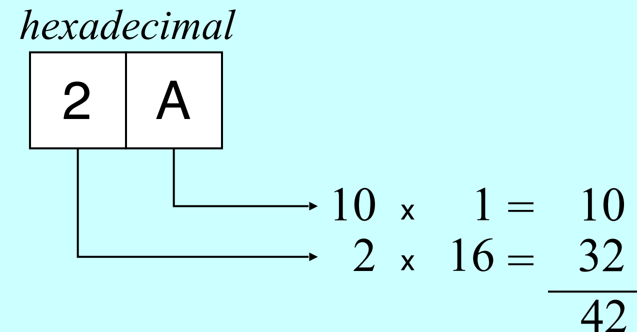
- Numbers do not have bases; representations do.

Octal and Hexadecimal Notation

- Because binary notation tends to get rather long, computer scientists often prefer ***octal*** (base 8) or ***hexadecimal*** (base 16) notation instead. Octal notation uses eight digits: 0 to 7. Hexadecimal notation uses sixteen digits: 0 to 9, followed by the letters A through F to indicate the values 10 to 15.

- The following diagrams show how the number forty-two appears in both octal and hexadecimal notation:





- The advantage of using either octal or hexadecimal notation is that doing so makes it easy to translate the number back to individual bits because you can convert each digit separately.

Exercises: Number Bases

- What is the decimal value for each of the following numbers?

10001_2 → 17 177_8 → 127 AD_{16} → 173

- As part of a code to identify the file type, every Java class file begins with the following sixteen bits:

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

How would you express that number in hexadecimal notation?

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

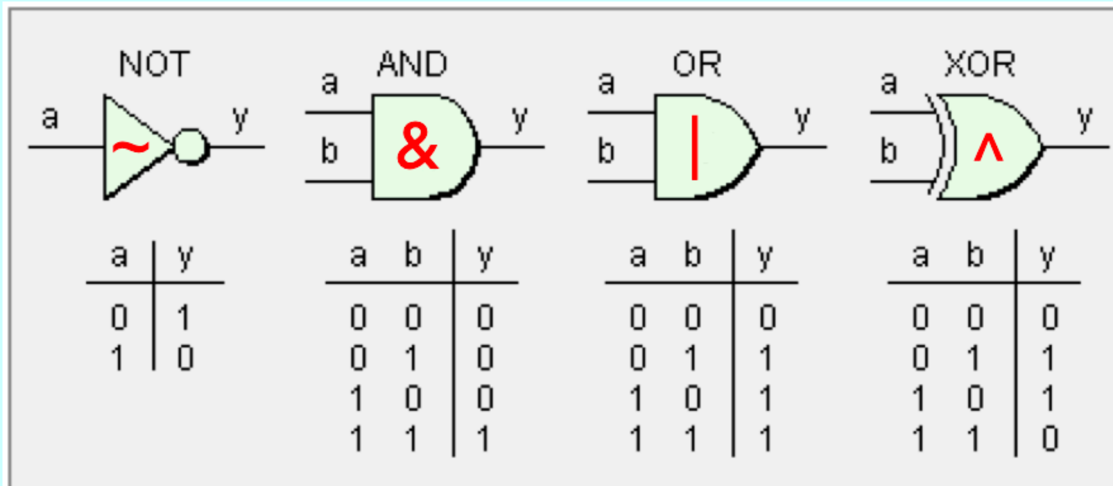
$CAFE_{16}$

Bits and Representation

- Sequences of bits have no intrinsic meaning except for the representation that we assign to them, both by convention and by building particular operations into the hardware.
- As an example, a 32-bit word represents an integer only because we have designed hardware that can manipulate those words arithmetically, applying operations such as addition, subtraction, and comparison.
- By choosing an appropriate representation, you can use bits to represent any value you can imagine:
  - Characters are represented using numeric character codes.
  - Floating-point representation supports real numbers.
  - Two-dimensional arrays of bits represent images.
  - Sequences of images represent video.
  - And so on . . .

Adding Numbers

- How does a computer actually add numbers together?
- We have discussed how computers deal with 0s and 1s at the lowest level. Computers use transistors to manipulate the digital 0s and 1s, and certain transistor configurations can be used to build logic gates to perform boolean functions: AND, OR, XOR, NOT, etc.
- We are going to "build an adder" using logic and a set of logic gates. The symbols for the logic gates we will use look like this:



a	y
0	1
1	0

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

Determining a Circuit from a Truth Table

- We often write A AND B as AB, and A OR B as A+B. We also write A XOR B as A⊕B
- All circuits can be made from a combination of AND and OR gates, in the following way:
  - For each set of inputs that produces a "1" output, AND together all of the inputs such that the result is "1". For inputs that are 0, use the NOT operator. For example:

A	B	C	**Result**
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**0**
1	1	1	**0**

- There are three 1 outputs on the right. The first output is when all three inputs are 0, so it is true for $\overline{A}\,\overline{B}\,\overline{C}$. The next 1 result would be from $\overline{A}\,B\,\overline{C}$, etc.
- Altogether, we could get the result as follows:
- $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,B\,\overline{C} + \overline{A}\,B\,C + A\,\overline{B}\,C$

Distribution and Substitutions

- Once we have the equivalent AND and OR circuit, we can use some mathematical properties to reduce the number of components we will need for our circuit.
- For the result on the previous slide:

$\bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,B\,\bar{C} + \bar{A}\,B\,C + A\,\bar{B}\,C$

- We can use the distributive property to find the common C and $\bar{C}$ terms, as follows:

$\bar{C}\,(\bar{A}\,\bar{B} + \bar{A}\,B) + C\,(\bar{A}\,B + A\,\bar{B})$

- There are a couple of nice substitutions, as well:

$(\bar{A}\,B + A\,\bar{B}) = A \oplus B$

$(\bar{A}\,\bar{B} + AB) = \overline{A \oplus B}$

- So, above, we can make the following substitution:

$\bar{C}\,(\bar{A}\,\bar{B} + \bar{A}\,B) + C\,(\bar{A}\,B + A\,\bar{B}) = \bar{C}\,(\bar{A}\,\bar{B} + \bar{A}\,B) + C\,(A \oplus B)$

Distribution and Substitutions

- Now, we are left with, which we can reduce even further:

  $\overline{C}\,(\overline{A}\,\overline{B} + \overline{A}\,B) + C\,(A \oplus B)$

- For the first term, we can pull out the $\overline{A}$:

  $\overline{C}\overline{A}\,(\overline{B} + B)$

- But, $(\overline{B} + B)$ is *always* true, so this further reduces to:

  $\overline{C}\overline{A}$

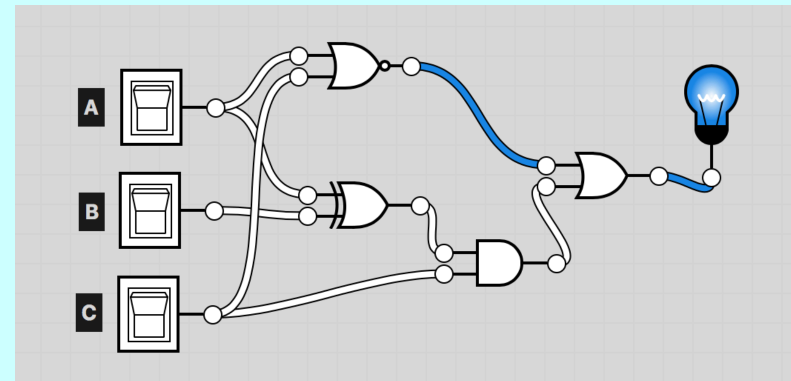
- We now have:

  $\overline{C}\overline{A} + C\,(A \oplus B)$

- We can simplify the first term by using a NAND gate: $\overline{C}\overline{A} = \overline{A+C}$

- So our final result is: $\overline{A+C} + C(A \oplus B)$

- Which looks like this in a logic diagram:



Adding Two Bits

- When adding two one-bit numbers together, we have two outputs: a *sum* and a *carry*.
- Let's fill in the table below to determine these values based on adding two bits:

A	**B**	**Sum**	**Carry**
0	**0**		
0	**1**		
1	**0**		
1	**1**		

Adding Two Bits

- When adding two one-bit numbers together, we have two outputs: a *sum* and a *carry*.
- Let's fill in the table below to determine these values based on adding two bits:

A	**B**	**Sum**	**Carry**
0	**0**	0	0
0	**1**	1	0
1	**0**	1	0
1	**1**	0	1

Adding Two Bits

- When adding two one-bit numbers together, we have two outputs: a *sum* and a *carry*.
- Let's fill in the table below to determine these values based on adding two bits:

A	**B**	**Sum**	**Carry**
0	**0**	0	0
0	**1**	1	0
1	**0**	1	0
1	**1**	0	1

- Can we determine what logic gates we can use to produce the sum and the carry from A and B?

Adding Two Bits

- When adding two one-bit numbers together, we have two outputs: a *sum* and a *carry*.
- Let's fill in the table below to determine these values based on adding two bits:

A	**B**	**Sum**	**Carry**
0	**0**	0	0
0	**1**	1	0
1	**0**	1	0
1	**1**	0	1

- Can we determine what logic gates we can use to produce the sum and the carry from A and B?

Let's Build a Half Adder!

https://logic.ly/demo

The Full Adder

- A half adder takes two one-bit numbers and adds them. But, if we want to do this for more than one bit, we have to include a carry-in as well. Let's make a new table:

A	B	Carry In	Sum	Carry Out
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

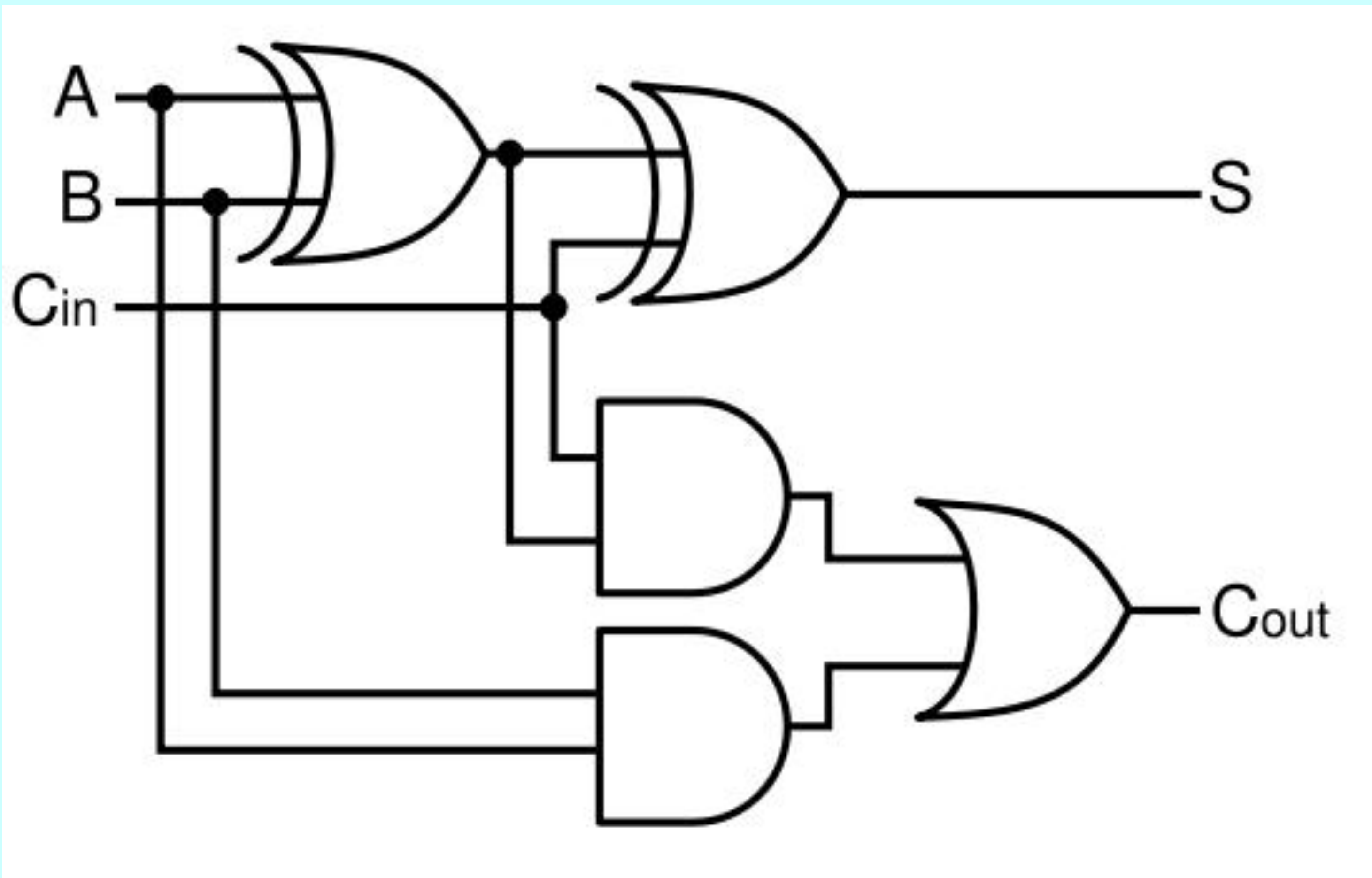
The Full Adder

- A half adder takes two one-bit numbers and adds them. But, if we want to do this for more than one bit, we have to include a carry-in as well. Let's make a new table:

A	B	Carry In	Sum	Carry Out
0	**0**	**0**	0	0
0	**0**	**1**	1	0
0	**1**	**0**	1	0
0	**1**	**1**	0	1
1	**0**	**0**	1	0
1	**0**	**1**	0	1
1	**1**	**0**	0	1
1	**1**	**1**	1	1

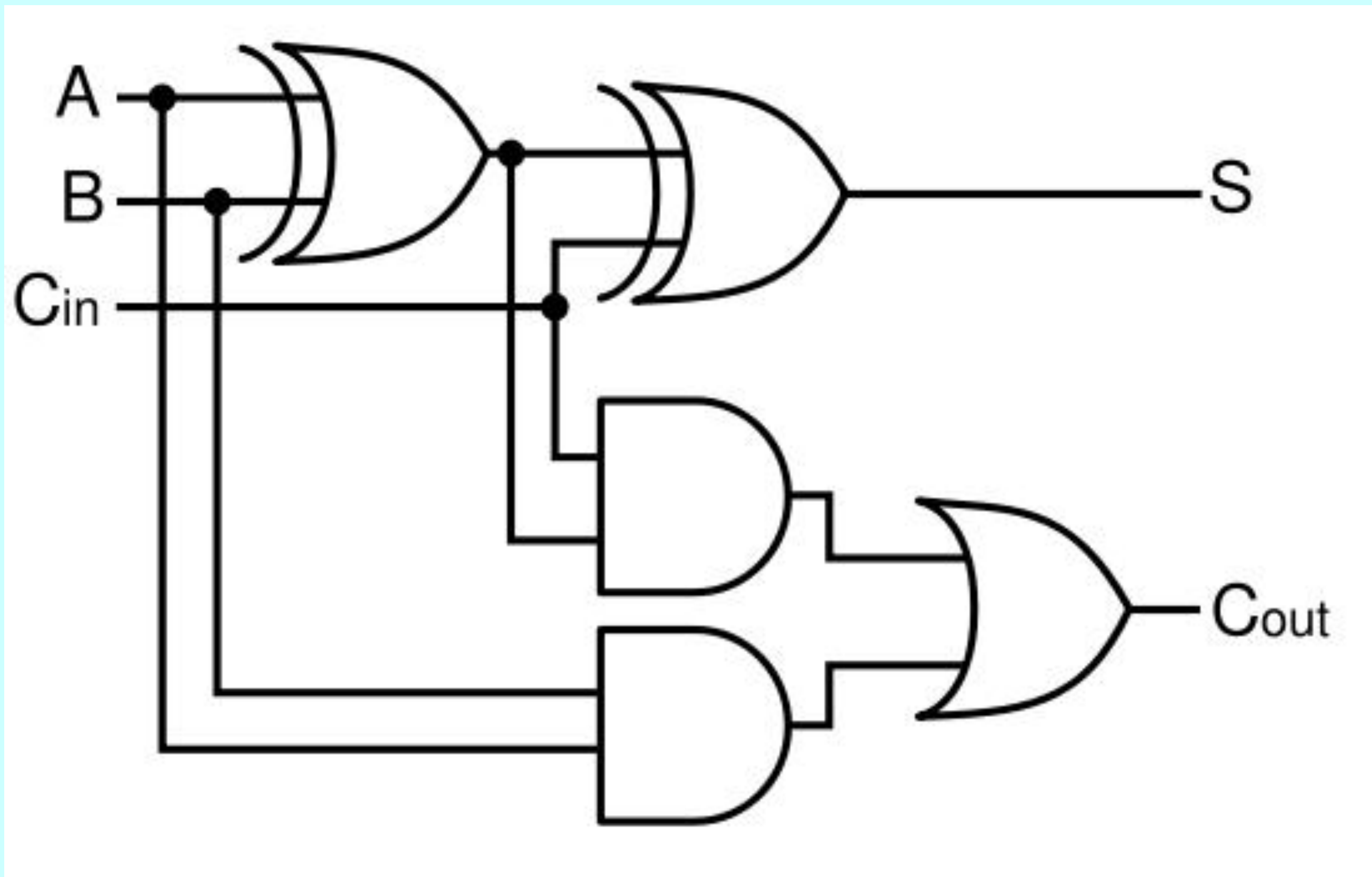
The Full Adder

- The logic for the Full Adder is more difficult, and has multiple solutions. Here is one solution:



The Full Adder

- The logic for the Full Adder is more difficult, and has multiple solutions. Here is one solution:



- Let's build a 4-bit Full Adder!

Representing Characters

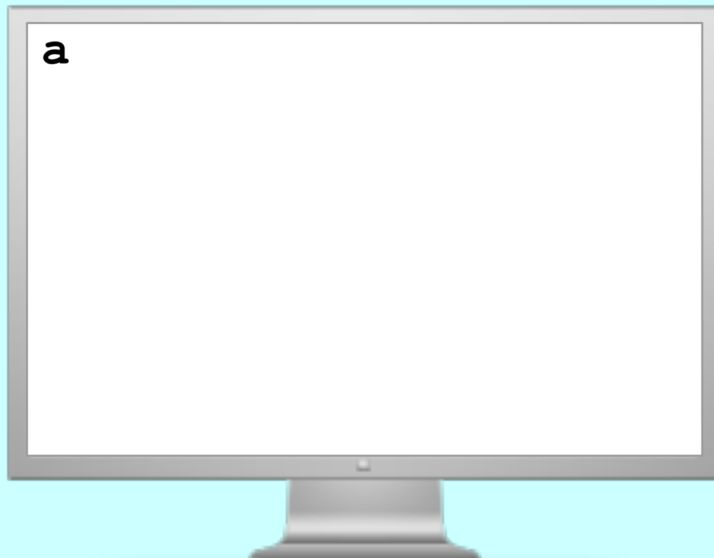
- Computers use numeric encodings to represent character data inside the memory of the machine, in which each character is assigned an integral value.
- Character codes, however, are not very useful unless they are standardized. When different computer manufacturers use different coding sequence (as was indeed the case in the early years), it is harder to share such data across machines.
- The first widely adopted character encoding was ASCII (*American Standard Code for Information Interchange*).
- With only 256 possible characters, the ASCII system proved inadequate to represent the many alphabets in use throughout the world. It has therefore been superseded by Unicode, which allows for a much larger number of characters.

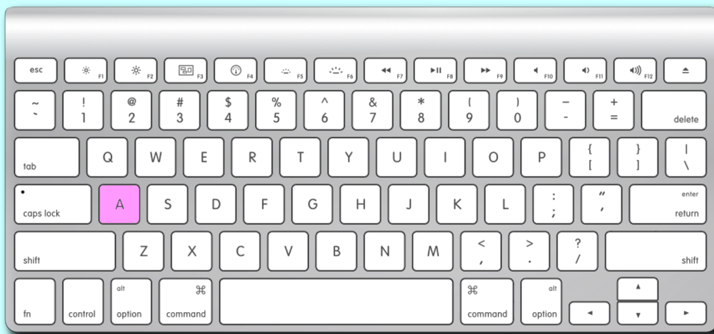
The ASCII Subset of Unicode

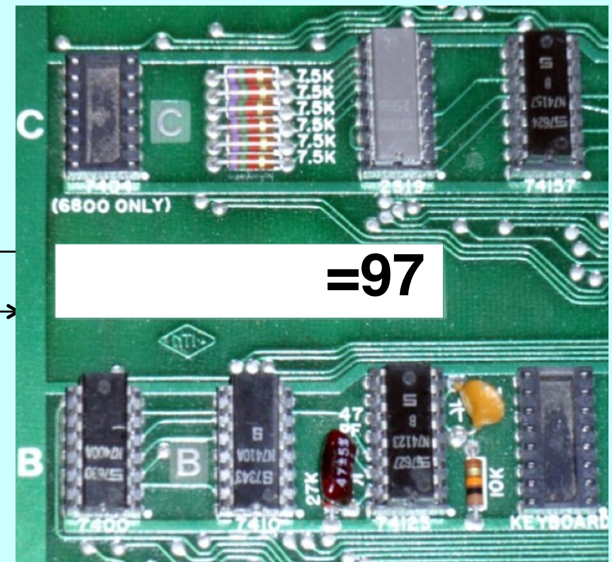
The following table shows the first 128 characters in the Unicode character set, which are the same as in the older ASCII scheme:

	0	1	2	3	4	5	6	7
00*x*	\000	\001	\002	\003	\004	\005	\006	\007
01*x*	\b	\t	\n	\011	\f	\r	\016	\017
02*x*	\020	\021	\022	\023	\024	\025	\026	\027
03*x*	\030	\031	\032	\033	\034	\035	\036	\037
04*x*	*space*	!	"	#	$	%	&	'
05*x*	(	)	*	+	,	-	.	/
06*x*	0	1	2	3	4	5	6	7
07*x*	8	9	:	;	<	=	>	?
10*x*	@	A	B	C	D	E	F	G
11*x*	H	I	J	K	L	M	N	O
12*x*	P	Q	R	S	T	U	V	W
13*x*	X	Y	Z	[	\	]	^	_
14*x*	`	a	b	c	d	e	f	g
15*x*	h	i	j	k	l	m	n	o
16*x*	p	q	r	s	t	u	v	w
17*x*	x	y	z	{	|	}	~	\177

Hardware Support for Characters







Strings as an Abstract Idea

- Characters are most often used in programming when they are combined to form collections of consecutive characters called ***strings***.
- As you will discover when you have a chance to look more closely at the internal structure of memory, strings are stored internally as a sequence of characters in sequential memory addresses.
- The internal representation, however, is really just an implementation detail. For most applications, it is best to think of a string as an abstract conceptual unit rather than as the characters it contains.
- JavaScript emphasizes the abstract view by defining a built-in string type that defines high-level operations on string values.

Using Strings as Values

- You can store a string value in a JavaScript variable, just as if it were any other type of value.
- The following declaration defines a variable named **`str`** and sets it to the ten-character string **`"hello, world"`**:

```
var str = "hello, world";
```

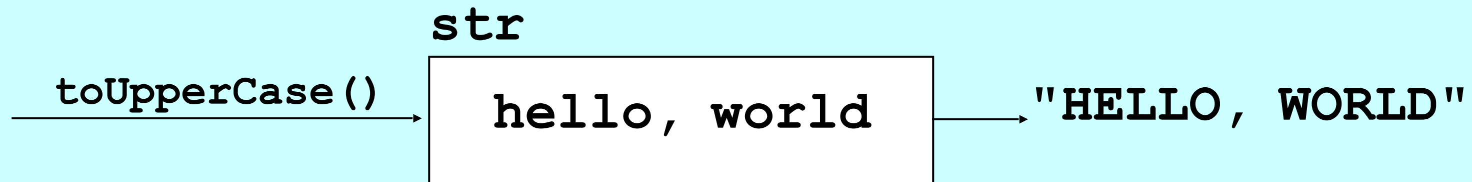
- As with any other value, this declaration is best interpreted as creating a box named **`str`** with the appropriate initial value:

`str`

`hello, world`

Strings as Objects

- In JavaScript, strings are implemented as ***objects***, which are structures that combine data and operations into a single unit.
- In object-oriented languages, operations on objects are implemented using ***methods***, which are functions that belong to a particular object. Calling a method is viewed as ***sending a message*** to that object, as shown in the following example:



- The object to which a message is sent is called the ***receiver***.
- The general pattern for sending a message to an object is

receiver`.`*name*`(`*arguments*`);`

Selecting Characters from a String

- Conceptually, a string is an ordered collection of characters.
- In JavaScript, the character positions in a string are identified by an ***index*** that begins at 0 and extends up to one less than the length of the string, as follows:

h	e	l	l	o	,		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10	11

- The length of a string **`str`** is given by **`str.length`**.
- You can select a character by calling **`charAt(`***k***`)`**, where *k* is the index of the desired character. The expression

```
str.charAt(0);
```

returns the one-character string **`"h"`**.

Concatenation

- One of the most useful operations available for strings is ***concatenation***, which consists of combining two strings end to end with no intervening characters.
- Concatenation is built into JavaScript in the form of the **+** operator. If you use **+** with numeric operands, it signifies addition. If at least one of its operands is a string, JavaScript interprets **+** as concatenation. If either of the operands is not a string, JavaScript converts it to a string before concatenating the strings together.

Extracting Substrings

- The **`substring`** method makes it possible to extract a piece of a larger string by providing index numbers that determine the extent of the substring.
- The general form of the **`substring`** call is

```
str.substring(p1, p2);
```

  where **`p1`** is the first index position in the desired substring and **`p2`** is the index position immediately following the last position in the substring.

- For example, if **`str`** contains **`"hello, world"`**, the following expression extracts the substring **`"ell"`**:

```
str.substring(1, 4);
```

Other Useful String Methods

`indexOf(pattern)` Returns the index of the first match of **`pattern`**, or −1 if none exists.
`lastIndexOf(pattern)` Returns the index of the last match of **`pattern`**, or −1 if none exists.
`toLowerCase()` Returns a copy of this string with all uppercase characters changed to lowercase.
`toUpperCase()` Returns a copy of this string with all lowercase characters changed to uppercase.

Exercise: The "Starts With" Function

- Implement a function **`startsWith(str, prefix)`** that returns true if **`str`** starts with **`prefix`**.

```
function startsWith(str, prefix) {
   if (str.length >= prefix.length) {
      return prefix ===
             str.substring(0, prefix.length);
   } else {
     return result;
   }
}
```

Simple String Idioms

When you work with strings, there are two idiomatic patterns that are particularly important:

1. Iterating through the characters in a string.

```
for (var i = 0; i < str.length; i++) {
   var ch = str.charAt(i);
   . . . code to process each character in turn . . .
}
```

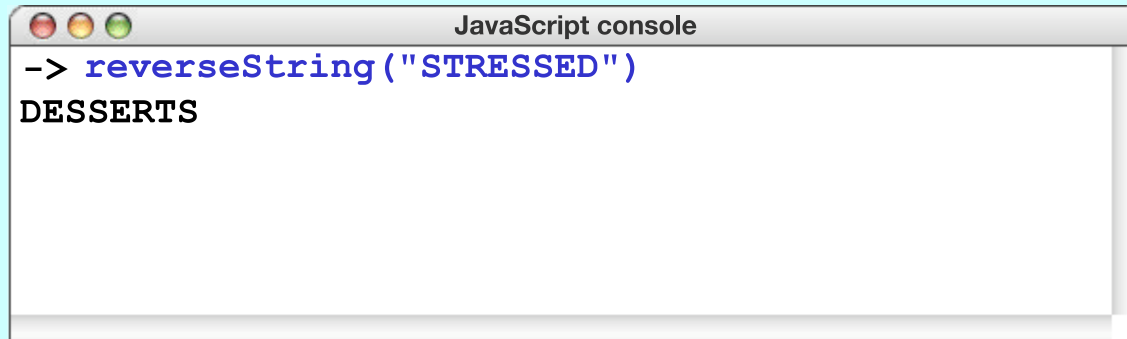
2. Growing a new string character by character.

```
var result = "";
for (whatever limits are appropriate to the application) {
   . . . code to determine the next character to be added . . .
   result += ch;
}
```

The `reverseString` Function

```
function reverseString(str) {
   var result = "";
   for ( var i = 0; i < str.length; i++ ) {
      result = str.charAt(i) + result;
   }
   return result;
}
```

result	str	i
DESSERTS	STRESSED	8



Exercise: Checking Palindromes

A ***palindrome*** is a string that reads the same forward and backward, such as **`"LEVEL"`** or **`"NOON"`**. How would you implement the predicate function **`isPalindrome(str)`**?

The End