

# CSCU9V4 Systems

## Systems Lecture 2 Data Representation

Graphical User Interfaces	Higher-level Programming
Operating Systems	
Low-level Programming	
Basic Machine Architecture	
Silicon	

# Data Representation

- Hexadecimal
- Conversion to Binary and Decimal
- Character encoding
- Strings

# Inconvenience of Binary

- Binary numbers are great for digital computers, but inconvenient for humans
- So why not just use decimal to think about the contents of the machine?

- Consider (say) 39 and 55 decimal, each stored in one byte

39        0010 0111

55        0011 0111

- These numbers differ by only one bit, and their right-hand nibbles are the same - this can sometimes be important for digital machines
- Electronics (all technologies!) store data in one of two states
  - Aside: using any finite number of states is possible, but 2 states are virtually always used
- These facts are concealed by decimal representation

# Hexadecimal

- But (for machines with 8-bit bytes) *hexadecimal* representation is excellent (well, a good compromise between machine and human readability)
  - “Hex” means six, and “dec” means ten
- Hexadecimal represents numbers in base 16
- We use the letters A to F as extra “digits”

- 0 1 2 3 4 5 6 7 8 9 A B C D E F (Hex)

- 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 (Decimal)

- Note:
  - Write ‘x’ after a hex number, or 0x before to distinguish it from a decimal one
    - 20x is not the the same value as 20 (decimal)...
  - Each 4 bit nibble translates into exactly one hexadecimal digit
    - 0x34 -> 0011 0100,      34 decimal -> 0010 0010
    - So a byte translates to 2 hexadecimal digits

# Hexadecimal

- How the extra digits are used:

0 decimal is 0x0	...
9 decimal is 0x9	15 decimal is 0xF
10 decimal is 0xA	16 decimal is 0x10

- Conversions follow familiar principles (see previous lecture)
- Converting from hex to decimal: consider (say) 0xA1F

210  
A1F

A x 16<sup>2</sup> + 1 x 16<sup>1</sup> + F x 16<sup>0</sup> which is

10 x 256 + 1 x 16 + 15 x 1 = 2,591

- So it's useful to know the first few powers of 16:
  - 16<sup>1</sup> = 16, 16<sup>2</sup> = 256, 16<sup>3</sup> = 4096, 16<sup>4</sup> = 65536, ...
  - (Note 16<sup>n</sup> is a way of writing 16<sup>n</sup>)

# Decimal to Hex

- Decimal to hex: just the same method as before!
  - Repeatedly divide the base into the number
  - Then write down the remainders in reverse order
- For example, convert 39 to hex:
  - $39 / 16 \rightarrow 2 \text{ remainder } 7$
  - $2 / 16 \rightarrow 0 \text{ remainder } 2$
- So 39 decimal is **27x**
- Converting 55 gives **37x**: so we can now see the similarity between 39 and 55, previously obscured by writing them in decimal
- Another example: 175 to hex:
  - $175 / 16 \rightarrow 10 \text{ remainder } 15 = \text{Fx}$
  - $10 / 16 \rightarrow 0 \text{ remainder } 10 = \text{Ax}$
- So 175 decimal is **AFx**
  - And this is easy to convert to binary:
  - 1010 1111

0010 0111

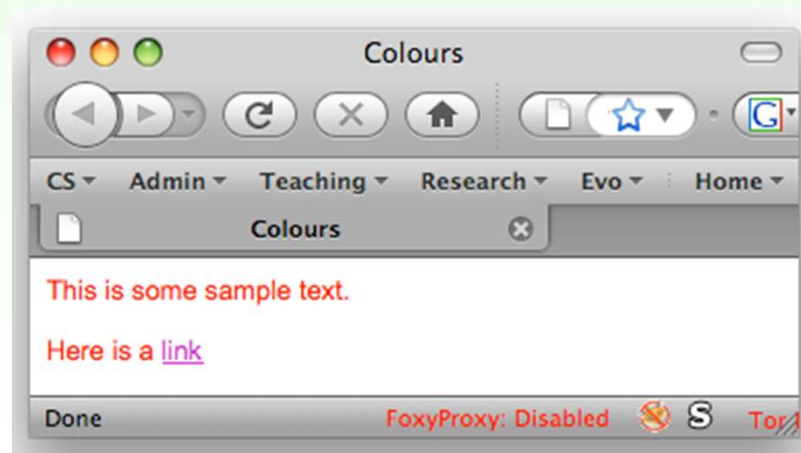
0011 0111

# Hexadecimal: A Practical Use

- When writing web pages, hexadecimal is used to specify colours (background colours, text colours, etc.)
- Colours are written using the RGB colour model: (why? discuss)
  - one byte each for the amounts of red, green and blue
  - 0-255 in decimal, **00x - FFx** in hexadecimal
- Examples of RGB colours:
  - 00x,00x,00x is black
  - FFx,00x,00x is **red**
  - C0x,00x,C0x is **purple** (mix of red and blue)
  - FFx,FFx,FFx is **white**
- Not clear *at all* from the decimal value for this 24 bit number!
- RGBA: A (alpha channel) is transparency, again 8 bits,
  - 00x fully transparent
  - FFx fully opaque.

# An example web page

```
<!DOCTYPE html>
<html>
<head><title>colours</title></head>
<body bgcolor="#ffffff" text="#ff0000" link="#c000c0">
this is some sample text.
<p>
here is a
<a href=nowhere.html>link</a>
</body>
</html>
```





# Moving on from integers

- We now know how to store integers in binary (and how to write these in hexadecimal)
- In subsequent lectures we shall look further at numbers (addition, subtraction, negative integers, fractions)
- But what about
  - characters (text) ?
  - sounds ?
  - Images ?
- *Everything* that a computer is storing is stored as a string of binary digits
- How it is interpreted depends on the program using these binary strings
  - Which means the same string can be interpreted in many different ways

# Characters

- Characters are fairly straightforward, at least for English
  - (but less so for languages with accents (ö, ç, á, etc) and even less so for languages using pictograms!)
- The system assigns a different small number to each character (and when it prints them, it knows to print the character not the number)
- How many characters are there?
  - A ... Z, a ... z, 0 ... 9, space gives us 63
- ... and then there are some punctuation characters ..: .,:@' (){}[]\$%^&\* etc.
- So we need somewhere near at least 80 different codes
- So we can hold them all in seven bits
  - So long as there are fewer than 128 altogether

# Character Encoding Systems

- There are several standards for encoding characters as numbers. Some major examples:
  - **ASCII** (American Standard Code for Information Interchange) is a 7-bit code
  - ISO 8859 (International Organization for Standardization) codes are 8-bit, primarily for Western languages
  - EBCDIC (Extended Binary Coded Decimal Interchange Code) is 8-bit, several variants again for Western languages
  - **Unicode** (see later)
- We will use ASCII for historical reasons, because it is very widely used (e.g. email messages) and because it is (?) the standard for plain text
  - (perhaps was would be more accurate)
  - Because Unicode is becoming much more common

# ASCII (and Extended ASCII)

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00																
10																
20	sp	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
80	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
90	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	×	f
A0	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	«	»
B0	—	—	—			Á	Â	À	©			+	+	¢	¥	+
C0	+	-	-	+	-	+	ã	Ã	+	+	-	-		-	+	¤
D0	õ	Ð	Ê	Ë	È	ì	Í	Î	Ï	+	+	—	—		Ì	¬
E0	Ó	ß	Ô	Ò	õ	Õ	µ	þ	Þ	Ú	Û	Ü	Ý	Ý	—	¬
F0		±	—	¾	¶	§	÷	¸	°	²	·	¹	³	²	—	

# Collating (sorting) order

- ASCII is a 7-bit code
- “Extended” ASCII is an 8-bit code that includes strange characters like the corners of boxes: you may ignore these and just focus on ASCII
- ASCII and other encodings have desirable properties:
  - ‘A’ before ‘B’ before ... ‘Z’
  - ‘a’ before ‘b’ before ... ‘z’
  - ‘0’ before ‘1’ before ... ‘9’ (notice that the character ‘0’ is not the same thing as the number zero!)
  - a usefully simple relationship between upper and lower-case letters: in ASCII the relationship is that the lower-case letters have bit 5 set
    - ‘A’ is **41x**, 0100 0001 binary
    - ‘a’ is **61x**, 0110 0001 binary

# Characters

- The ASCII characters in the range `00x ... 1Fx` are “non-printing”: they are “control characters”
  - Historically, they were separators, tabs, ringing a bell on a teletype,...
- If you hold down the `Ctrl` key and a letter-key together, you get a character whose code is the letter's code minus `40x` (i.e. with bit 6 set to 0)
  - `'A'` is `41x`, `^A` is `01x`
  - `'Z'` is `5Ax`, `^Z` is `1Ax`
- These characters have names like `CR`, `LF`, `TAB`, `EOF`, `EOT`, `BEL`
  - `CR` is `0Dx`, `^M`
  - `LF` is `0Ax`, `^J`
  - `TAB` is `09x`, `^I`
  - `BEL` is `07x`, `^G`
- The reasons for these names are sometimes clear (`LF` = line feed, `TAB` = tab character) and sometimes obscure (`CR` = carriage return, `EOT` = end of transmission, `BEL` = ring teletype bell)



# Unicode

- As we have seen, there are many more symbols than just in English. Many European languages have additional marks (e.g. őçäéè etc.). And many languages use a different script (Greek, Russian, Arabic, Korean, Japanese, Hebrew). In addition, many languages use pictograms (e.g. Chinese)
- Unicode is an international standard to be able to represent all of the characters used in the world's major languages: see <http://www.unicode.org>

*Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.*  
(from <http://www.unicode.org>)

- More or less universally adopted
- See <http://www.unicode.org/consortium/members.html>
- A major task: look at Unicode version 8, for example.  
-<http://www.unicode.org/versions/Unicode8.0.0/>

# Unicode continued

- It was originally a 16-bit code (so  $2^{16}$  symbols possible)
  - Is this enough?
- Now exists in three forms, UTF8, UTF16, UTF32
  - In UTF32, each code point is embedded in a 32 bit word.  
**Unicode code space is 0 to  $10FFFF_{16}$**   
**Simplest, but not space efficient**
  - In UTF 16, basic multilingual plane (codes from 0 to  $FFFF_{16}$ ) are represented in 16 bits  
**Supplementary characters use two 16 bit elements**
  - In UTF8 a variable width coding is used that maintains transparency with ASCII
- eg
  - $05D0_{16}$    $20AC_{16}$  
- Java uses Unicode to represent characters; C does not.  
see <http://www.unicode.org/>



# Strings of characters

- Text is just a sequence of characters that are stored in adjacent locations
- Suppose we have a 32-bit (4 bytes) value in memory, with hex representation

46 72 65 64x

- This *could* be a single integer value (a big one!), or it could be a series of ASCII characters

46x	'F'
72x	'r'
65x	'e'
64x	'd'

# Strings

- A String will usually have an area of memory allocated to store it but it may not use all of that memory allocation. We need some method of indicating how much space the actual String is using.
- Strings are usually either “counted” or “null-terminated”
  - Counted : 0446726564x
  - Null (0) terminated : 4672656400x

46	72	65	64	00	00	00	00	00	00	00	00	00	00	00	00
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

- Java uses (hidden) “null-terminated” strings
  - So do most modern languages!
  - Counting tends to restrict string length
- C uses ‘\0’, ie. null termination.

## Bit patterns are interpreted using conventions

- So, to repeat,
  - a portion of memory will **always** have some bit-pattern generally random at switch-on time!
  - there are various (many) conventions about how to interpret bit patterns
  - we have seen two of these: e.g. in one byte the bit pattern 0100 0001 might represent 'A' or the number 65, or , ...
  - the programmer (usually via some programming language) decides which of the (many) available conventions will be applied to a given portion of memory
- For example, a programmer might write the following declarations:

```
int i;  
char c;
```
- Just looking at the bit pattern in memory, you will not know its intended use. You would need to look at the program to understand how to interpret it.

**End of Lecture**