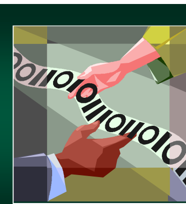
The Basics

Part 1

Binary Numbers



Bit of This and a Bit of That

What is a Number?

- We use the Hindu-Arabic Number System
  - positional grouping system
  - each position represents a power of 10
- Binary numbers
  - based on the same system
  - use powers of **2** rather than 10

Base 10 Number

The number 1783 is ...

10^4	10^3	10^2	10^1	10^0
10000	1000	100	10	1
0	1	7	8	3

1000 + 700 + 80 + 3 = 1783

Binary Number Example

The number 1010 1001 is ...

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	0	1	0	1	0	0	1

128 + 32 + 8 + 1 = 169

Binary Number Example

The number 1101 1011 is ...

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	1	0	1	1	0	1	1

128 + 64 + 16 + 8 + 2 + 1 = 219

Hexadecimal Numbers

- Writing out long binary numbers is cumbersome and error prone
- As a result, computer scientists often write computer numbers in hexadecimal
- Hexadecimal is base-16
  - We only have 0...9 to represent digits
  - So, hexadecimal uses **A...F** to represent **10...15**

Hexadecimal Numbers

Hex	Decimal	Binary	Hex	Decimal	Binary
0	0	`0000`	8	8	`1000`
1	1	`0001`	9	9	`1001`
2	2	`0010`	A	10	`1010`
3	3	`0011`	B	11	`1011`
4	4	`0100`	C	12	`1100`
5	5	`0101`	D	13	`1101`
6	6	`0110`	E	14	`1110`
7	7	`0111`	F	15	`1111`

Hex Example

The number **`A2C`** is ...

16^3	16^2	16^1	16^0
4096	256	16	1
0	**A**	**2**	**C**

$$(10 \times 256) + (2 \times 16) + (12 \times 1) = 2604$$

Converting Binary to Hex = Easy

- Since $16 = 2^4$, a single hex character can represent a total of 4 bits
- Byte can be represented with only 2 hex digits!
- When looking at raw data, editors, called Hex Editors, display data as groups of 2 hex digits

5				**C**			
0	1	0	1	1	1	0	0

Bits and Bytes

- Everything in a *modern* computer is stored using combination of ones and zeros
- *Bit* is one binary digit
  - either 1 or 0
  - shorthand for a bit is b
- *Byte* is a group of 8 bits
  - e.g. `0010 0100`
  - shorthand for a byte is B

The Byte

Nibble

0	1	1	0	1	1	0	1

bit

Hex & Binary Notation

It gets confusing quick, so let's prepare

Hex & Binary Notation

- Hexadecimal and binary notations use the same digits we use for decimal
- So, some numbers look like valid hex, decimal and binary
- Many programming languages have notation that specifies the base

Hex & Binary Notation

- For example is **`101`** …
  - binary value 5*?*
  - decimal value 101*?*
  - hexadecimal value 257*?*
- This, obviously, can become problematic

Examples for 00101110_2 ($2E_{16}$)

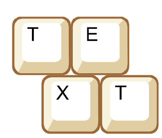
Notation	Binary	Hexadecimal
Subscript	`00101110`$_2$	`2E`$_{16}$
C++, Java, C#	`0b00101110`	`0x2E`
Visual Basic .NET	`&b00101110`	`&h2E`
VHDL	`2#00101110#`	`16#2E#`
AT&T Assembly	`0b00101110`	`0x2E`
Intel Assembly	`0y00101110`	`0x2E or 2Eh`
Verilog	`8'b2E`	`8'h2E`

Text in Programming Languages

Press Any Key to Continue

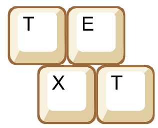
Characters

- Computer often store and transmit textual data
- Examples:
  - punctuation
  - numerals 0 – 9
  - letter
- Each of these symbols is called a *character* and are the basis for written communication



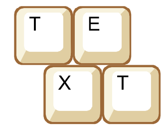
Characters

- Processors rarely know what a "character" is, and instead store each as an integer
- In this case, each character is given a unique value
- The letter "A", for instance, could have the value of 1, "B" is 2, etc…



Characters

- Characters and their matching values are a *character set*
- There have been many characters sets developed over time



Character Sets

- ASCII
  - 7 bits – 128 characters
  - uses a full byte, one bit is not used
  - created in the 1967
- EBCDIC
  - Alternative system used by old IBM systems
  - Not used much anymore

ASCII Chart

Control characters

	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	sp	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
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

Useful Control Characters

	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	sp	!	"	#	$	%	&	'	(	)	*	+	,	–	.	/
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

ASCII Codes

- Each character has a unique value
- The following is how "OMG" is stored in ASCII

	Binary	Hex	Decimal
O	0100 1111	4F	79
M	0100 1101	4D	77
G	0100 0111	47	71

ASCII Codes

- ASCII is laid out very logically
- Alphabetic characters (uppercase and lowercase) are 32 "code points" apart

	Binary	Hex
A	01000001	41
a	01100001	61

ASCII Codes

- $32 = 2^5$
- Uppercase and lowercase letters are just 1 bit different
- Converting between the two is easy

	Decimal	Hex	Binary
A	65	41	01000001
a	97	61	01100001

ASCII: Number Characters

- ASCII code for 0 is 30h
- The characters 0 to 9 can be easily converted to their binary values
- Notice that the binary value is stored in the lower nibble

0	0011 0000
1	0011 0001
2	0011 0010
3	0011 0011
4	0011 0100
5	0011 0101
6	0011 0110
7	0011 0111
8	0011 1000
9	0011 1001

ASCII: Number Characters

- Character → Binary
  - clear the upper nibble
  - Binary-And 0000 1111
- Binary → Character
  - set the upper nibble to 0011
  - Binary-Or 0011 0000

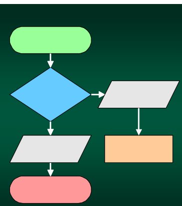
0	0011 0000
1	0011 0001
2	0011 0010
3	0011 0011
4	0011 0100
5	0011 0101
6	0011 0110
7	0011 0111
8	0011 1000
9	0011 1001

Unicode Character Set

- ASCII is only good for the United States
  - Other languages need additional characters
  - Multiple competing character sets were created
- Unicode was created to support every spoken language
- Developed in Mountain View, California

Unicode Character Set

- Originally used 16 bits
  - that's over 65,000 characters!
  - includes every character used in the World
- Expanded to 21 bits
  - 2 million characters!
  - now supports every character ever created
- Unicode can be stored in different formats

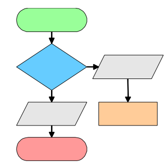


What are Programs?

It's all just a bunch of bytes

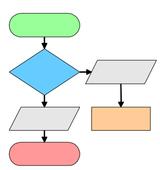
High-Level Programming

- You are used to writing programs in high level programming languages
- Examples:
  - C#
  - Java
  - Python
  - Visual Basic



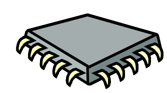
High-Level Programming

- These are *third-generation languages*
- They and are designed to isolate you from architecture of the machine
- This layer of abstraction makes programs "portable" between systems



Computer Processors

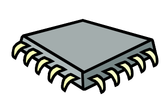
- The *Central Processing Unit (CPU)* is the most complex part of a computer
- In fact, it is the computer
- It works far different from a high-level language



Computer Processors

- Over time, thousands of processors were developed
- Examples:
  - Intel x86
  - IBM PowerPC
  - MOS 6502
  - ARM

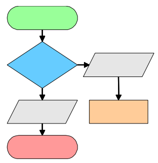
Instructions



- Processors do not have the constructs you find in high-level languages
- Examples:
  - Blocks
  - If Statements
  - While Statements
  - … etc

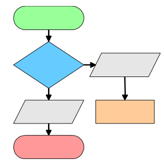
Instructions

- Processors can only perform a series of simple tasks
- These are called *instructions*
- *Examples:*
  - add two values together
  - move a value
  - jump to a memory location

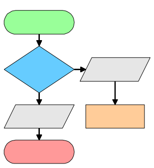


Instructions

- These instructions are used to create all logic needed by a program
- We will cover how to do this during the semester



Processor Instruction Set



- A processor's *instruction set* defines all the available instructions
- The instructions and their respective formats are very different for each processor

Registers

Where the work is done

Registers

- In high level languages, you put active data into variables
- However, it works quite different on processors
- All computations are performed using *registers*

What – exactly – is a register?

- A *register* is a location, on the processor itself, that is used to store temporary data
- Think of it as a special global "variable"
- Some are accessible and usable by a programs, but many are hidden

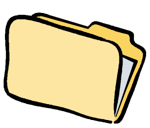
What are registers used for?

- Registers are used to store anything the processor needs to keep to track of
- Designed to be *fast!*
- Examples:
  - the result of calculations
  - status information
  - memory location of the running program
  - and much more…

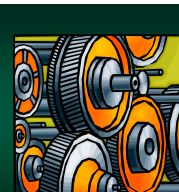
General Purpose Registers

- *General Purpose Registers (GPR)* don't have a specific purpose
- They are designed to be used by programs – however they are needed
- Often, you must use registers to perform calculations

Register Files



- All the related registers are grouped into a *register file*
- Different processors access and use their register files in very different ways
- Some processors support multiple files

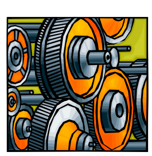


Machine Language

The raw bytes of your program

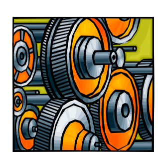
Machine Language

- The instructions, that are *actually* executed on the processor, are a series of bytes
- In this raw binary form, instructions are stored in *machine language*



Machine Language

- Each instruction is in a compact binary form
- Easy for the processor to interpret and execute
- Some instructions are take more bytes than others – not all are equal



Instruction Encoding

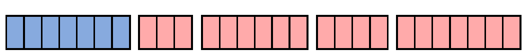
- Each instruction must contain <u>everything</u> the processor needs to know to do something
- So, if you want it to add 2 registers, it has to specify *which* ones

Operation Codes

- Each instruction has a <u>unique</u> *operation code (Opcode)*
- This value that specifies the <u>exact</u> operation to be performed by the processor
- Assemblers use friendly names called *mnemonics*

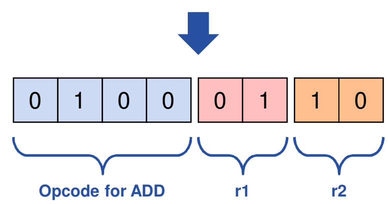
Typical Instruction Format

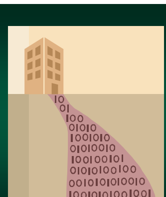
- The opcode is, typically, followed by various *operands* – what data is to be used
- These can be register codes, addressing data, literal values, etc…



Machine Code Example (not x86)

```
ADD %r1, %r2
```



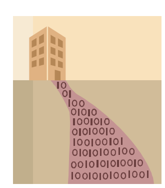


What is Memory?

Its… um…. I forgot….

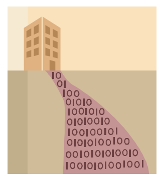
Computer Memory

- Programs access and manipulate memory far more than you realize
- So, understanding it…
  - is vital to becoming a great assembly programmer
  - and understanding computer architecture



What is Memory?

- Memory is essentially an enormous array
- It is also, sometimes, referred to as *storage*
- It stores both running programs and their related data



Memory Addresses

- Memory is divided into a storage locations that can hold 1 byte (8 bits) of data
- Each byte has an *address*
  - unique value that refers to that specific byte
  - used to locate the exact byte the processor wants

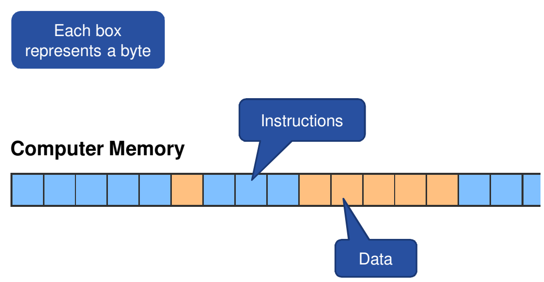
	Memory
0	01000100
1	01000011
2	01101111
3	01101111
4	01101011

What is Memory?

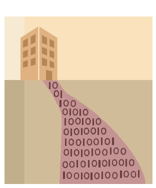
- So, each address is conceptually the same as the "index" in array terminology
- … and you will write access memory as would an array

	Memory
0	01000100
1	01000011
2	01101111
3	01101111
4	01101011

Memory Contains Data & Programs



Memory Contains Data & Programs



- Data and instructions are just binary numbers (stored in a series of bytes)
- …and are stored together
- Appreciating this is vital to understanding computer architecture