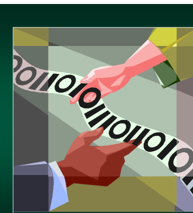
Processor Basics

Part 1

1

Binary Numbers



Bit of This and a Bit of That

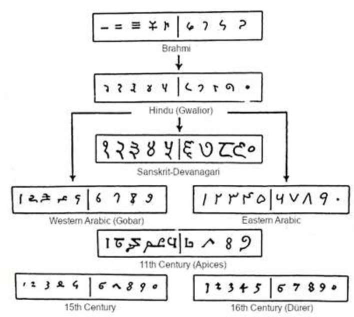
2

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 powers of 10

3

Evolution of a Genius System



4

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

5

Binary Number Example

The number 0100 1010 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
0	**1**	**0**	**0**	**1**	**0**	**1**	**0**

64 + 8 + 2 = 74

6

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

7

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, hex uses **A ... F** to represent **10 ... 15**

8

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`

9

Hex Example

The number A2C is ...

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

(10×256) + (2×16) + (12×1) = 2604

10

Converting Binary to Hex = Easy

- Since $16^{1} = 2^{4}$, a single hex character can represent a total of 4 bits
- To convert: treat every 4-binary digits as a single hexadecimal digit
- The conversion is direct!

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

11

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. **`1101 0100`**
  - shorthand for a byte is B

12

The Byte

Nibble

$\underbrace{0\ 1\ 1\ 0}_{\text{Nibble}}\ 1\ 1\ 0\ 1$

0	1	1	0	1	1	0	1

bit

Text in Programming Languages

Press Any Key to Continue

How Text Is Stored

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

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, "C" is 3, 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

19

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

20

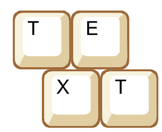
Evolution of ASCII

From ASCII to Unicode

21

Evolution of ASCII

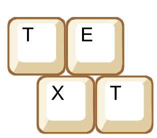
- The ASCII Character Set has gone through a few minor revisions this it was first created
- But, the 1967 standard is still used today



22

Evolution of ASCII

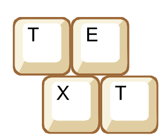
- The initial 1963 version didn't include lowercase letters
- As shocking as it sounds, there was a time where lowercase letters were considered obsolete
- All-caps was called "modern"



23

Evolution of ASCII

- Back in the 1960's – there were no computer monitors
- Nor did you enter programs using a keyboard
- Instead…
  - programs were inputted using punched cards or tape
  - all output was printed



24

Original 1963 ASCII Proposal

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOM	EOA	EOM	EOT	WRU	RU	BEL	FE0	HT	LF	VT	FF	CR	SO	SI
1	DC0	DC1	DC2	DC3	DC4	ERR	SYN	LEM	S0	S1	S2	S3	S4	S5	S6	S7
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																
7													ACK		ESC	DEL

25

1967 Revision Added Lowercase

Changed

New

			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
		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

26

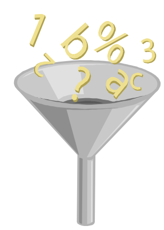
Control Characters

- Many of the control characters were designed for transferring data (SOH, STX, FS, GS, etc…)
- Other control characters we designed for printing

27

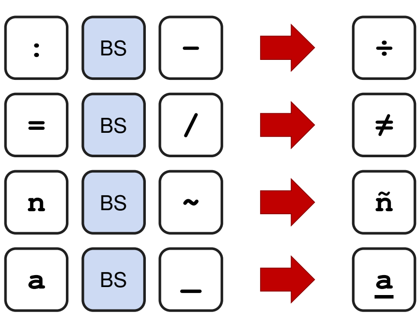
Backspace Control Character

- Printers, at the time, were basically classic typewriters
- The backspace character (BS) was used to print 2+ symbols on top of each other
- This would essentially create a new character on the paper



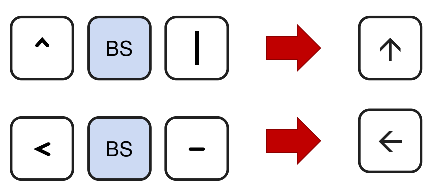
28

Power of the Backspace



29

Recreating the Removed Arrows



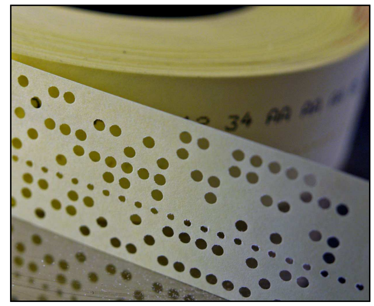
30

DEL Control Character

- You might have noticed an odd control character located at 7F
- This is the "Deleted" control character and was used with punched cards and tape

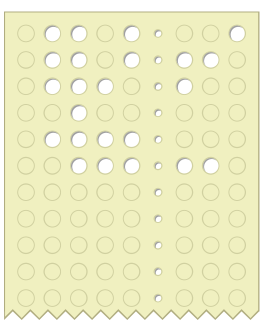
31

Punched Tape



32

Example Punched Tape: `int x;`

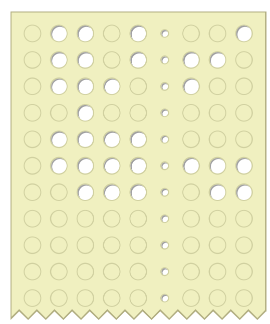


Student punched the wrong hole! They meant ;

Do they have to punch an entire new tape?

33

Example Punched Tape: `int x;`



Just punch all remaining holes. It becomes DEL and is ignored

34

ASCII Codes

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

	Decimal	Hex	Binary
A	65	41	01000001
a	97	61	01100001

35

ASCII Codes

- $32^1 = 2^5$
- There is just a 1-bit difference between uppercase and lowercase letters
- Printers can easily convert between the two

	Decimal	Hex	Binary
A	65	41	01000001
a	97	61	01100001

36

ASCII: Number Characters

- ASCII code for 0 is 30h
- Notice that the actual value of a number character is stored in the lower nibble
- So, the characters 0 to 9 can be easily converted to their binary values

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

37

ASCII: Number Characters

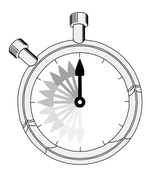
- Character → Binary
  - clear the upper nibble
  - Bitwise And: 0000 1111
- Binary → Character
  - set the upper nibble to 0011
  - Bitwise Or: 0011 0000

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

38

Times have changed…

- Computers have changed quite a bit since the 1960's
- As a result, most of these clever control characters are no longer needed
- Backspace, DEL, and numerous others are obsolete



39

Only Control Characters Still Used

	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

40

Unicode Character Set

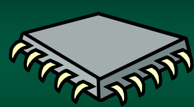
- ASCII is only good for the United States
  - Other languages need additional characters
  - Multiple competing character sets were created
- Unicode was created to support every spoken language
- Developed in Mountain View, California

41

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
  - … and emojis
- Unicode can be stored in different formats

42



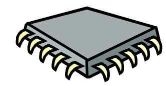
Processors

What are they? Besides awesome!

43

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



44

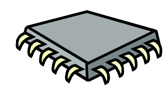
Computer Processors

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

45

Computer Processors

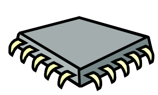
- Each processor functions differently
- Each is designed for a specific purpose – *form follows function*



46

Computer Processors

- But all share some basic properties and building blocks…
- Computer hardware is divided into two "units"
  1. Control Logic Unit
  2. Execution Unit



47

Control Logic Unit (CLU)

- *Control Logic Unit (CLU)* controls the processor
- Determines when instructions can be executed
- Controls internal operations
  - fetch & decode instructions
  - invisible to running programs

48

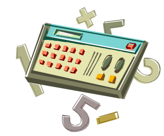
Execution Unit

- *Execution Unit (EU)* contains the hardware that *executes* tasks (your programs)
- Different in many processors
- Modern processors often use multiple execution units to execute instructions in parallel to improve performance

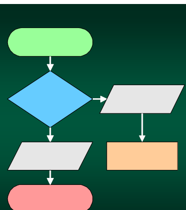
49

Execution Unit – The ALU

- *Arithmetic Logic Unit* is part of the Execution Unit and performs all calculations and comparisons
- Processor often contains special hardware for integer and floating point



50



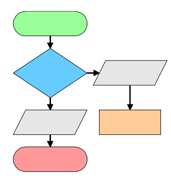
Instructions

It's all just a bunch of bytes

51

Instructions

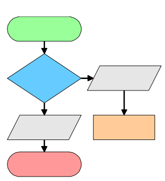
- You are used to writing programs in high level programming languages
- Examples:
  - C#
  - Java
  - Python
  - Visual Basic



52

High-Level Programming

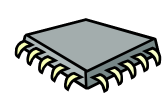
- These are *third-generation languages*
- They are designed to <u>isolate</u> you from architecture of the machine
- This layer of abstraction makes programs "portable" between systems



53

Instructions

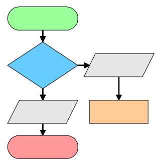
- Processors <u>do not</u> have the constructs you find in high-level languages
- Examples:
  - Blocks
  - If Statements
  - While Statements
  - … etc



54

Instructions

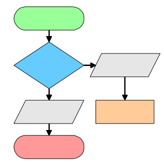
- Processors can only perform a series of simple tasks
- These are called *instructions*
- *Examples:*
  - add two values together
  - copy a value
  - jump to a memory location



55

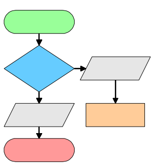
Instructions

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



56

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

57

Registers

Where the work is done

58

Registers

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

59

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

60

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…

61

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

62

Special Registers

- There are a number of registers that are used by the Control Logic Unit and cannot be accessed by your program
- This includes registers that control how memory works, your program execution thread, and much more.

63

Special Registers

- *Program Counter (PC)*
  - keeps track of the memory location of your running program
  - think it as the "line number" in your Java program – which one is being executed
  - it can be changed indirectly *(using control logic – which we will cover later)*

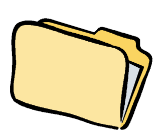
64

Special Registers

- *Status Register*
  - contains Boolean information about the processors current state
  - we will use this later, indirectly
- *Instruction Register (IR)*
  - stores the current instruction (being executed)
  - used internally and invisible to your program

65

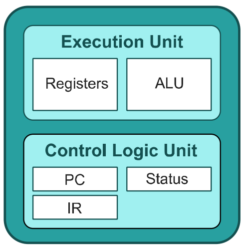
Register Files



- All the related registers are grouped into a *register file*
- Different processors access and use their register files in very different ways
- Sometimes registers are implied or hardwired

66

Components of a Processor



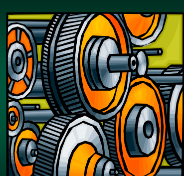
67

Please Wait

CSC 35 with begin shortly.

Please open the chat window.

68



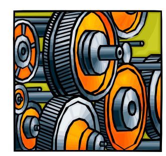
Machine Language

The raw bytes of your program

69

Machine Language

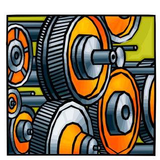
- The instructions, that are *actually* executed on the processor, are just bytes
- In this raw binary form, instructions are stored in *Machine Language (aka Machine Code)*



70

Machine Language

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



71

Instruction Encoding

- Each instruction must contain everything the processor needs to know to do something
- Think of them as functions in Java: they need a name and arguments to work

72

Instruction Encoding

- For example: if you want it to add 2 things…
- The instruction needs:
  - something to tell the processor to add
  - something to identify the two "things"
  - destination to save the result

73

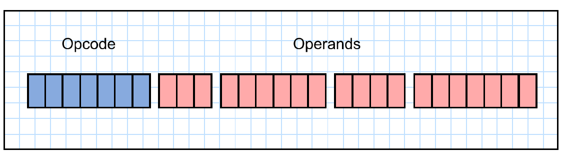
Operation Codes

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

74

Typical Instruction Format

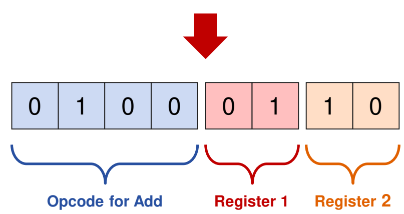
- The opcode is, typically, followed by various *operands* – what data is to be used
- These can be register codes, addressing data, literal values, etc…



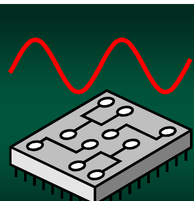
75

Machine Code Example (not x86)

```
ADD r1, r2
```



76



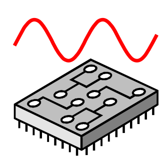
The System Clock

Tick-tock tick-tock tick-tock

77

The System Clock

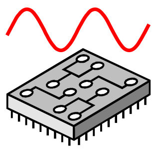
- The rate in which instructions are executed is controlled by the CPU clock
- The faster the clock rate, the faster instructions will be executed
- Measured in Hertz – number of oscillations per second



78

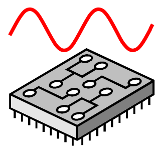
The Clock

- Computers are typically (and generically) labeled on the processor clock rate
- In the early 80's it was about 1 Megahertz – million clocks per second
- Now, it is terms of Gigahertz – billion clocks per second



79

Clock and Instructions



- Not all instructions are "equal"
- Some require multiple clock cycles to execute
- For example:
  - add can take a single clock
  - but floating-point math could require a dozen

80