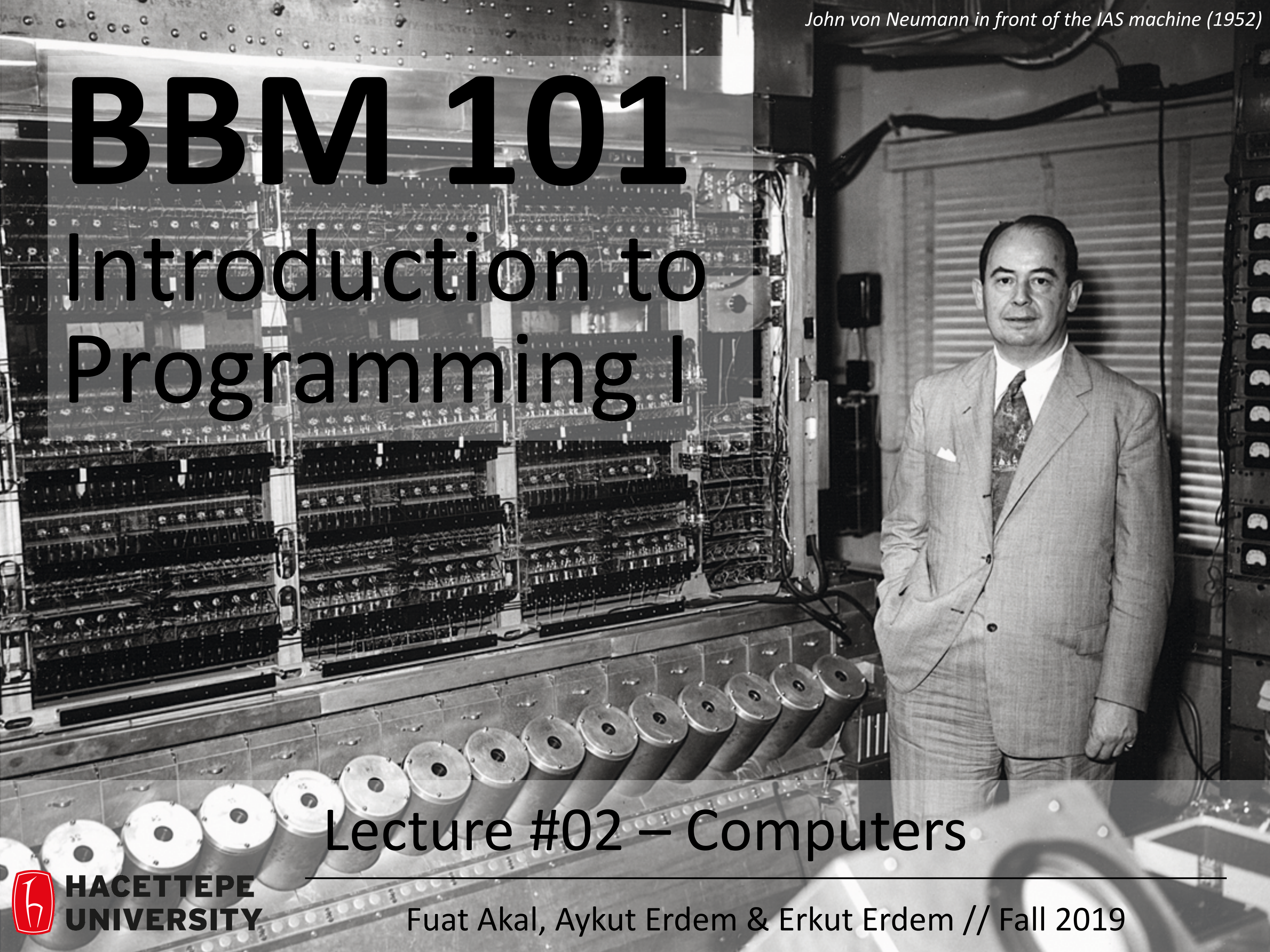


John von Neumann in front of the IAS machine (1952)

BBM 101

Introduction to Programming I

Lecture #02 – Computers

HACETTEPE UNIVERSITY

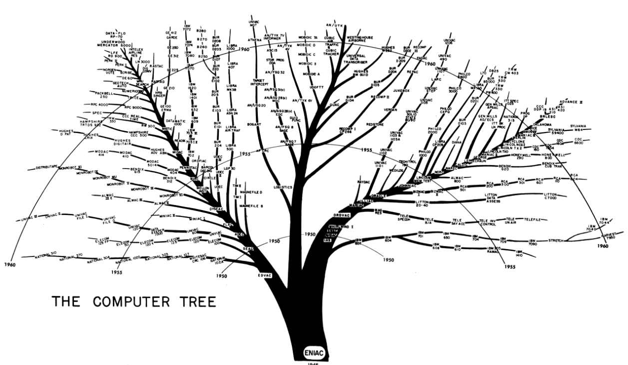
Fuat Akal, Aykut Erdem & Erkut Erdem // Fall 2019

Last time… **What is computation**

Computer science is about logic, problem solving, and creativity

Fixed Program Computers

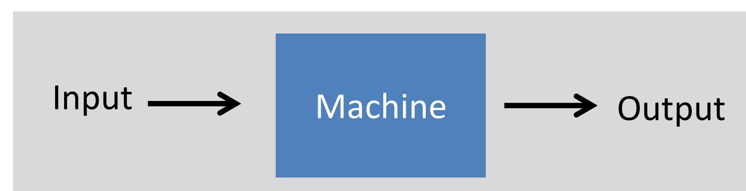
- Abacus
- Antikythera Mechanism
- Pascaline
- Leibniz Wheel
- Jacquard's Loom
- Babbage Difference Engine
- The Hollerith Electric Tabulating System
- Atanasoff-Berry Computer (ABC)
- Turing Bombe



- **Declarative knowledge**
  - Axioms (definitions)
  - Statements of fact
- **Imperative knowledge**
  - How to do something
  - A sequence of specific instructions (what computation is about)

Stored Program Computers

- Problem solving



- What if input is a machine (description) itself?
- Universal Turing machines
  - An abstract general purpose computer

Lecture Overview

- Building a Computer

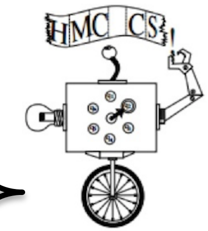
- The Harvey Mudd Miniature Machine (HMMM)

Disclaimer: Much of the material and slides for this lecture were borrowed from
— Gregory Kesden's CMU 15-110 class
—David Stotts' UNC-CH COMP 110H class
—Swami Iyer's Umass Boston CS110 class

Lecture Overview

- Building a Computer

- The Harvey Mudd Miniature Machine (HMMM)



Read the reference book

CS for All, by C. Alvarado, Z. Dodds, G. Kuenning & R. Libeskind-Hadas

Disclaimer: Much of the material and slides for this lecture were borrowed from
— Gregory Kesden's CMU 15-110 class
— David Stotts' UNC-CH COMP 110H class
— Swami Iyer's Umass Boston CS110 class

Lecture Overview

- Building a Computer
- The Harvey Mudd Miniature Machine (HMMM)

Building a Computer

- Numbers
- Letters and Strings
- Structured Information
- Boolean Algebra and Functions
- Logic Using Electrical Circuits
- Computing With Logic
- Memory
- von Neumann Architecture

Numbers

- At the most fundamental level, a computer manipulates electricity according to specific rules
- To make those rules produce something useful, we need to associate the electrical signals with the numbers and symbols that we, as humans, like to use
- To represent integers, computers use combinations of numbers that are powers of 2, called the base 2 or **binary representation**
  - **bit = 0** or **1**
    - False or True
    - Off or On
    - Low voltage or High voltage

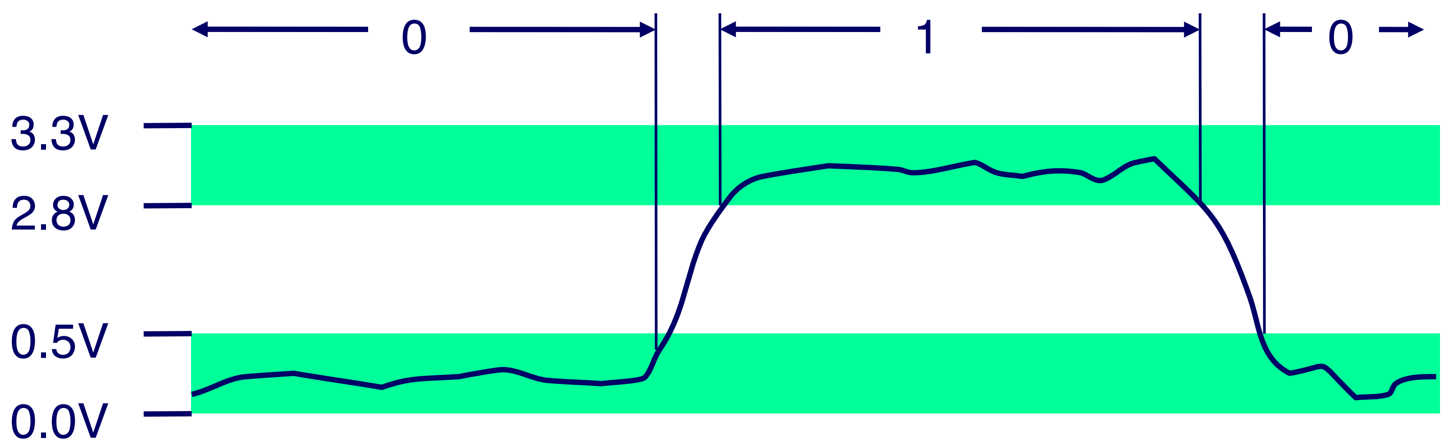


Image from: R.E. Bryant, D.R. O'Hallaron, G. Kesden

Numbers

- With four consecutive powers 2^0, 2^1, 2^2, 2^3, we can make all of the integers from 0 to 15 using 0 or 1 of each of the four powers
- For example, $13_{10} = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1101_2$; in other words, 1101 in base 2 means $1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$
- Analogously, 603 in base 10 means $603_{10} = 6 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0$ and 207 in base 8 means $207_8 = 2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0 = 135_{10}$
- In general, if we choose some base $b \geq 2$, every positive integer between 0 and $b^d - 1$ can be uniquely represented using d digits, with coefficients having values 0 through $b-1$
- A modern 64-bit computer can represent integers up to $2^{64} - 1$

Numbers

- Arithmetic in any base is analogous to arithmetic in base 10
- Examples of addition in base 10 and base 2

$$\begin{array}{rrr} & 1 & \\ & 1 & 7 \\ + & 2 & 5 \\ \hline & 4 & 2 \end{array} \qquad\qquad \begin{array}{rrrrr} & 1 & 1 & & \\ & & 1 & 1 & 1 \\ + & & 1 & 1 & 0 \\ \hline & 1 & 1 & 0 & 1 \end{array}$$

- To represent a negative integer, a computer typically uses a system called two's complement, which involves flipping the bits of the positive number and then adding 1
- For example, on an 8-bit computer, 3 = 00000011, so −3 = 11111101

Numbers

- If we are using base 10 and only have eight digits to represent our numbers, we might use the first six digits for the **fractional part** of a number and last two for the **exponent**
- For example, 31415901 would represent $0.314159 \times 10^1 = 3.14159$
- Computers use a similar idea to represent fractional numbers

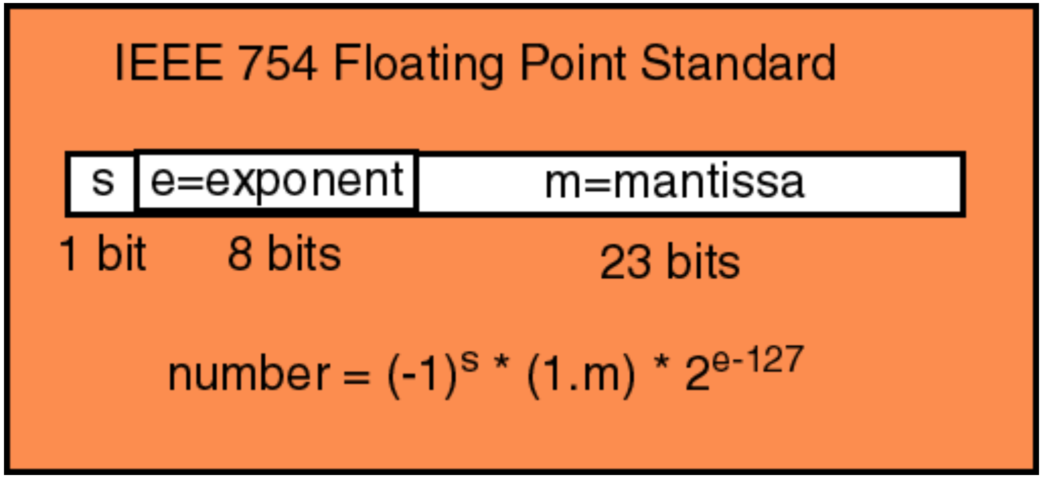


Image from: c-jump.com

Letters and Strings

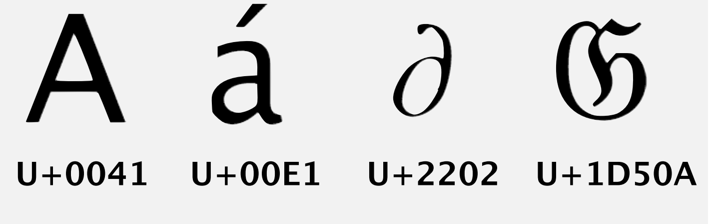
- In order to represent letters numerically, we need a convention on the encoding
- The American National Standards Institute (ANSI) has established such a convention, called ASCII (American Standard Code for Information Interchange)
- ASCII defines encodings for the upper- and lower-case letters, numbers, and a select set of special characters
- ASCII, being an 8-bit code, can only represent 256 different symbols, and doesn't provide for characters used in many languages

	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

Hexadecimal to ASCII conversion table

Letters and Strings

- The International Standards Organization’s (ISO) 16-bit Unicode system can represent every character in every known language, with room for more
- Unicode being somewhat wasteful of space for English documents, ISO also defined several “Unicode Transformation Formats” (UTF), the most popular being UTF-8



Unicode characters

Letters and Strings

- Emojis are just like characters, and they have a standard, too

Smileys & People															
face-positive															
№	Code	Browser	Appl	Googᵈ	Twtr.	One	FB	FBM	Sams.	Wind.	GMail	SB	DCM	KDDI	CLDR Short Name
1	U+1F600	😀	😀	😀	😀	😀	😀	😀	😀	😀	😀	—	—	—	grinning face
2	U+1F601	😁	😁	😁	😁	😁	😁	😁	😁	😁	😁	😁	😁	😁	beaming face with smiling eyes
3	U+1F602	😂	😂	😂	😂	😂	😂	😂	😂	😂	😂	😂	—	😂	face with tears of joy
4	U+1F923	🤣	🤣	🤣	🤣	🤣	🤣	—	🤣	🤣	—	—	—	—	rolling on the floor laughing
5	U+1F603	😃	😃	😃	😃	😃	😃	😃	😃	😃	😃	😃	😃	😃	grinning face with big eyes
6	U+1F604	😄	😄	😄	😄	😄	😄	😄	😄	😄	😄	😄	—	—	grinning face with smiling eyes
7	U+1F605	😅	😅	😅	😅	😅	😅	😅	😅	😅	😅	—	😅	—	grinning face with sweat
8	U+1F606	😆	😆	😆	😆	😆	😆	😆	😆	😆	😆	—	😆	—	grinning squinting face
9	U+1F609	😉	😉	😉	😉	😉	😉	😉	😉	😉	😉	😉	😉	😉	winking face

⋮

- Full Emoji List, v5.0
  https://unicode.org/emoji/charts/full-emoji-list.html

Letters and Strings

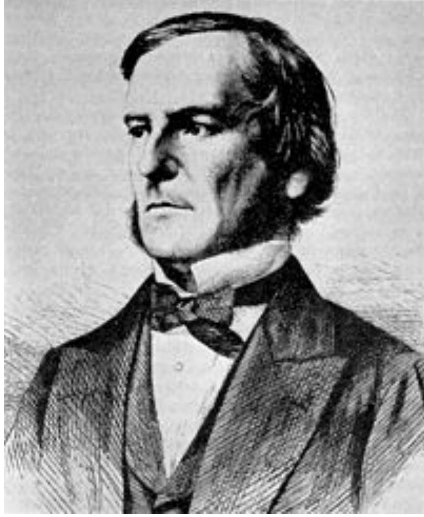
- A string is represented as a sequence of numbers, with a “length field” at the very beginning that specifies the length of the string

- For example, in ASCII the sequence 99, 104, 111, 99, 111, 108, 97, 116, 101 translates to the string “chocolate”, with the length field set to 9

Structured Information

- We can represent any information as a sequence of numbers
- Examples
  - A picture can be represented as a sequence of pixels, each represented as three numbers giving the amount of red, green, and blue at that pixel
  - A sound can be represented as a temporal sequence of “sound pressure levels” in the air
  - A movie can be represented as a temporal sequence of individual pictures, usually 24 or 30 per second, along with a matching sound sequence

Boolean Algebra and Functions



- Boolean variables are variables that take the value `True` (1) or `False` (0)

- With Booleans 1 and 0 we could use the operations (functions) `AND`, `OR`, and `NOT` to build up more interesting Boolean functions

- A truth table for a Boolean function is a listing of all possible combinations of values of the input variables, together with the result produced by the function

- Truth tables for AND, OR, and NOT functions

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

x	NOT x
0	1
1	0

Boolean Algebra and Functions

- Any function of Boolean variables, no matter how complex, can be expressed in terms of $\texttt{AND}$, $\texttt{OR}$, and $\texttt{NOT}$
- Consider the proposition "if you score over 93% in both midterm and final exams, then you will get an A"
- The truth values for the above proposition is given by the "implication" function ($x \implies y$) having the following truth table

x	y	$x \implies y$
0	0	1
0	1	1
1	0	0
1	1	1

- The function can be compactly written as $\texttt{NOT}\ x\ \texttt{OR}\ x\ \texttt{AND}\ y$ (or $\bar{x} + xy$)

Boolean Algebra and Functions

- The minterm expansion algorithm, due to Claude Shannon, provides a systematic approach for building Boolean functions from truth tables
- Minterm expansion algorithm
  1. Write down the truth table for the Boolean function under consideration
  2. Delete all rows from the truth table where the value of the function is 0
  3. For each remaining row, create something called a "minterm" as follows
     - For each variable that has a 1 in that row, write the name of the variable. If the input variable is 0 in that row, write the variable with a negation symbol to `NOT` it
     - Now AND all of these variables together
  4. Combine all of the minterms for the rows using `OR`

Boolean Algebra and Functions

- For the implication function, the minterm expansion algorithm applied as follows

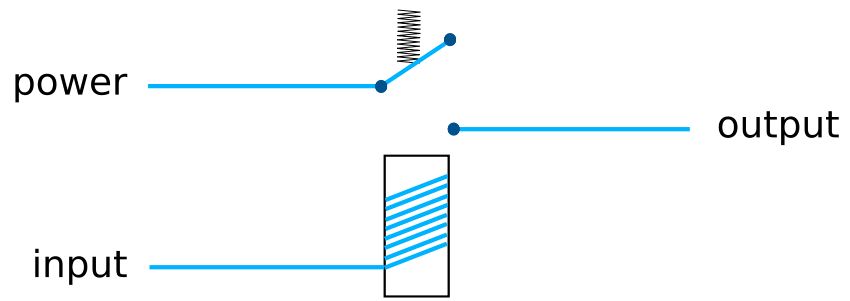
x	y	$x \implies y$	minterm
0	0	1	$\bar{x}\bar{y}$
0	1	1	$\bar{x}y$
1	0	0	
1	1	1	xy

produces the Boolean function $\bar{x}\bar{y} + \bar{x}y + xy$, which is equivalent to the simpler function $\bar{x} + xy$

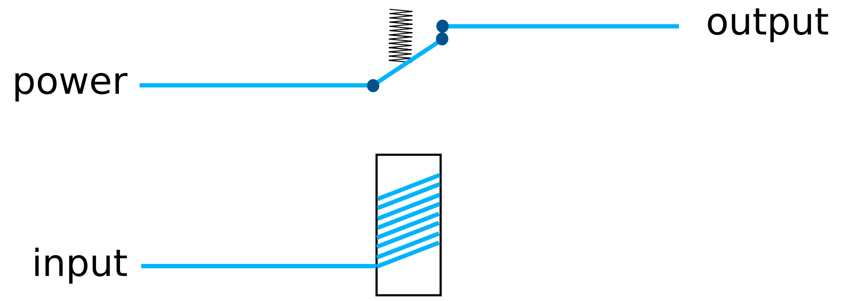
- Finding the simplest form of a Boolean function is provably as hard as some of the hardest (unsolved) problems in mathematics and computer science

Logic using Electrical Circuits

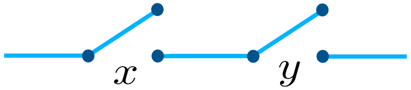
- An electromechanical switch in which when the input is off, the output is "low" (0), and when the input is on, the output is "high" (1)

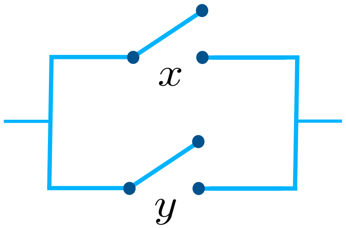


- The NOT gate constructed using a switch that conducts only when the input is off



- The AND and OR gates for computing x AND y and x OR y, constructed using electromechanical switches

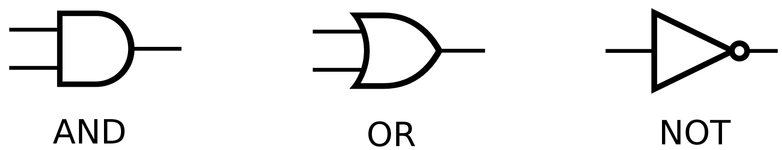




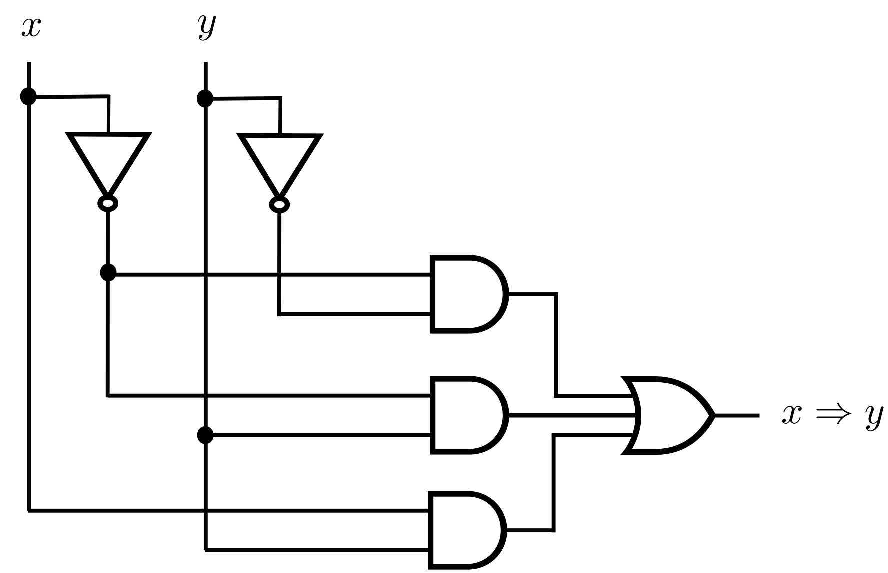
Logic using Electrical Circuits

- Computers today are built with much smaller, much faster, more reliable, and more efficient transistorized switches

- Since the details of the switches aren't terribly important at this level of abstraction, we represent, or "abstract", the gates using the following symbols



- A logical circuit for the implication function $\bar{x}\bar{y} + \bar{x}y + xy$



Computing with Logic

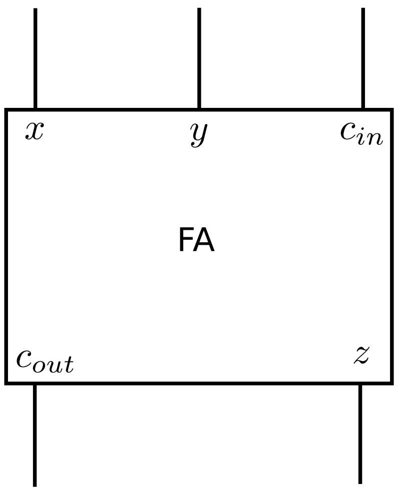
- A truth table describing the addition of two two-bit numbers to get a three-bit result

x	y	$x + y$
00	00	000
00	01	001
00	10	010
$\vdots$	$\vdots$	$\vdots$
01	10	011
01	11	100
$\vdots$	$\vdots$	$\vdots$
11	11	110

- Building a corresponding circuit using the minterm expansion algorithm is infeasible — adding two 16-bit numbers, for example, will result in a circuit with several billion gates

Computing with Logic

- We build a relatively simple circuit called a full adder (FA) that does just one column of addition



x	y	c_{in}	z	c_{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

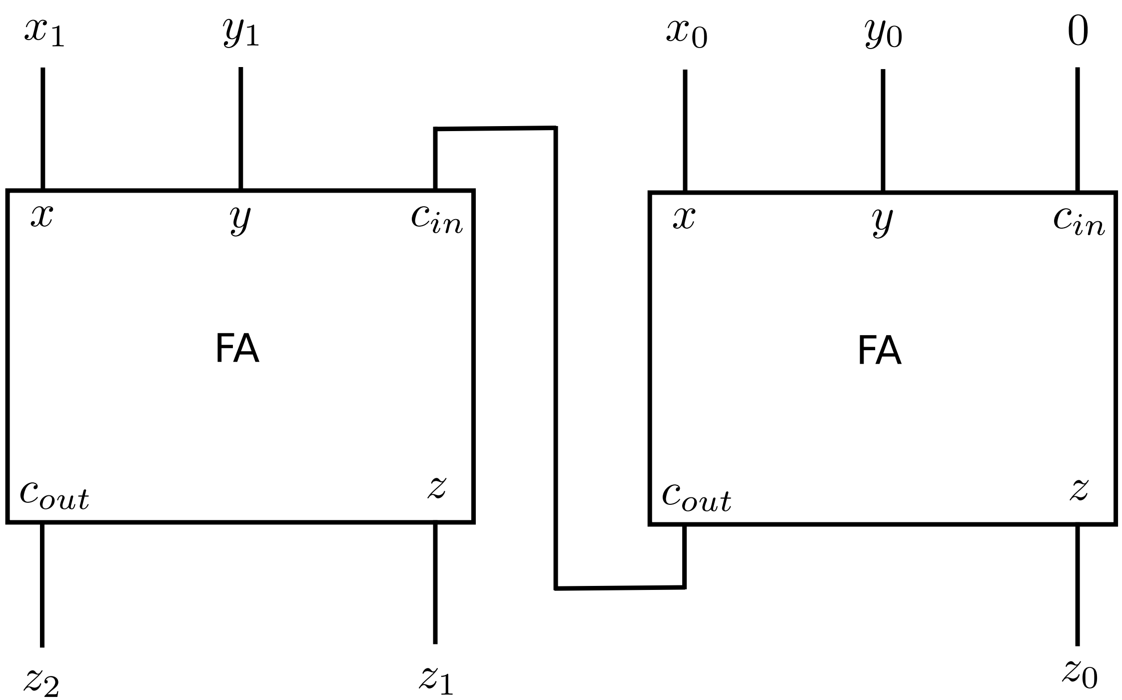
- The minterm expansion principle applied to the truth table for the FA circut yields the following Boolean functions

$$z = \bar{x}\bar{y}c_{in} + \bar{x}y\bar{c}_{in} + x\bar{y}\bar{c}_{in} + xyc_{in}$$

$$c_{out} = \bar{x}yc_{in} + x\bar{y}c_{in} + xy\bar{c}_{in} + xyc_{in}$$

Computing with Logic

- We can "chain" *n* full adders together to add two *n*-bit numbers, and the resulting circuit is called a ripple-carry adder
- A 2-bit ripple-carry adder

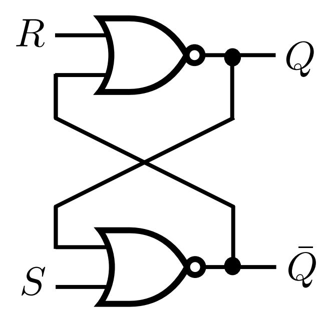


Memory

- Truth table for a `NOR` gate (`OR` followed by `NOT`)

x	y	x `NOR` y
0	0	1
0	1	0
1	0	0
1	1	0

- A latch is a device that allows us to “lock” a bit and retrieve it later
- By aggregating millions of latches we have the Random Access Memory (RAM)
- A latch can be constructed from two `NOR` gates as shown below



where the input S is known as “set” while the input R is known as “reset”

Recall: Stored Program Concept

- Stored-program concept is the fundamental principle of the ENIAC's successor, the EDVAC (Electronic Discrete Variable Automatic Computer)
- Instructions were stored in memory sequentially with their data
- Instructions were executed sequentially except where a conditional instruction would cause a jump to an instruction someplace other than the next instruction

Stored Program Concept

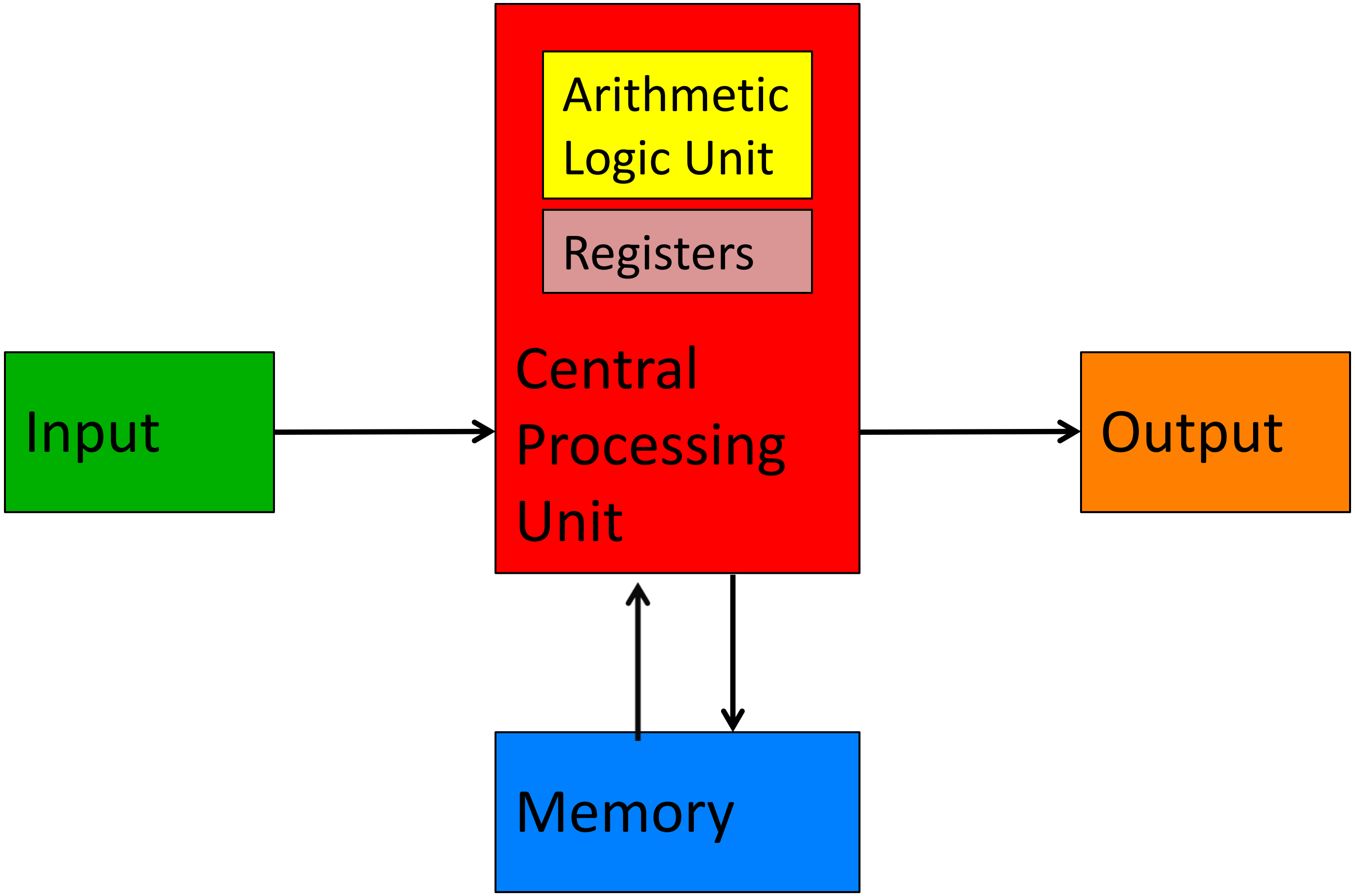
- Mauchly and Eckert are generally credited with the idea of the stored-program

- BUT: John von Neumann publishes a draft report that describes the concept and earns the recognition as the inventor of the concept
  - "von Neumann architecture"
  - A First Draft of a Report of the EDVAC published in 1945
  - http://www.worldpowersystems.com/J/EDVAC/

von Neumann,
Member of the Navy
Bureau of Ordinance
1941-1955

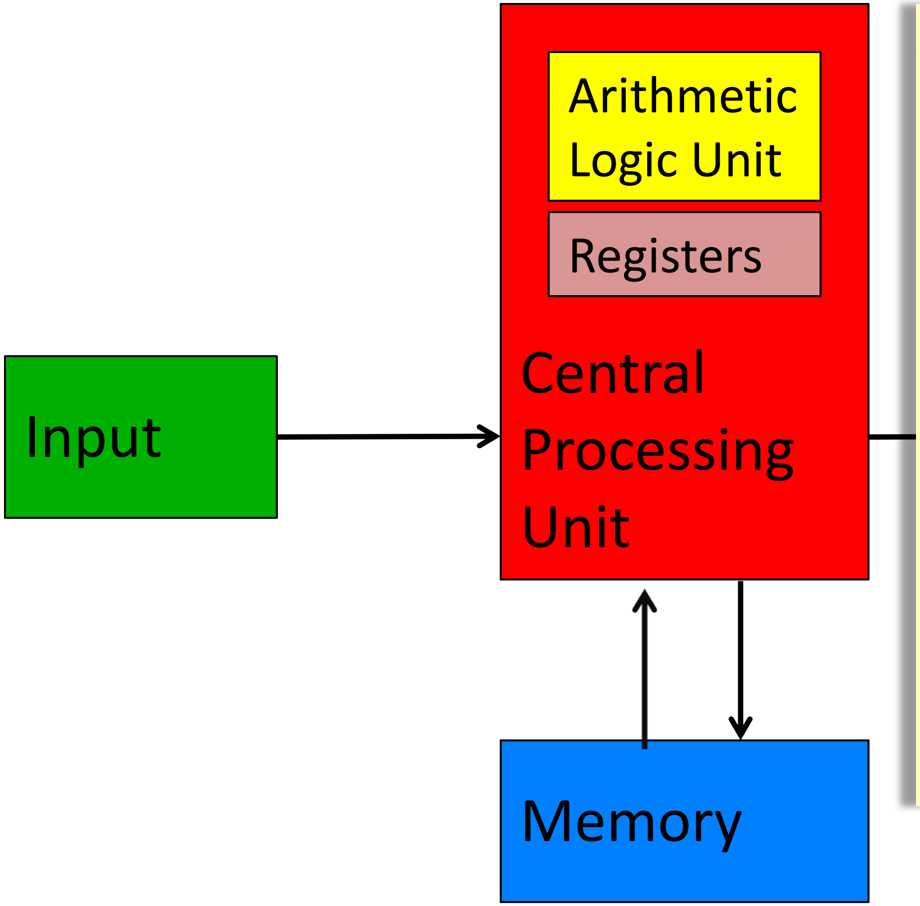
Stored Program Concept

- "Fetch-Decode-Execute" cycle



Stored Program Concept

- "Fetch-Decode-Execute" cycle

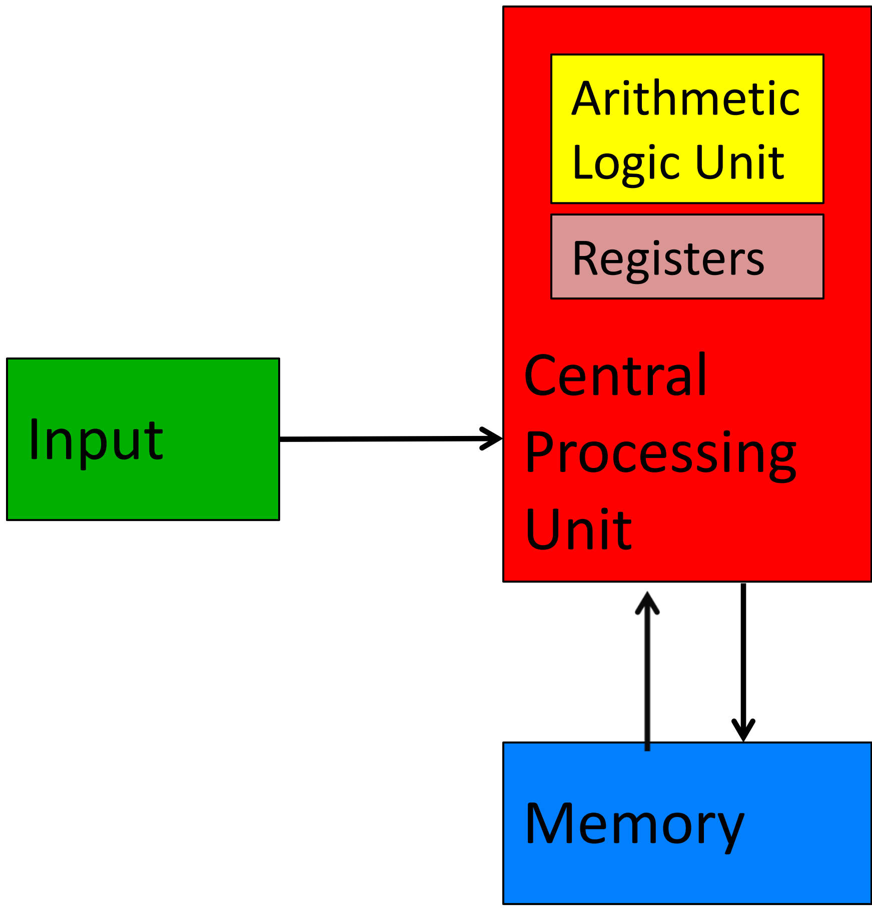


Central Processing Unit (CPU)

- In a modern computer, the CPU is where all the computation takes place
- The CPU has devices such as ripple-carry adders, multipliers, etc. for doing arithmetic. In addition, it has a small amount of (scratch) memory called registers
- The computer's main memory, which allows storing large amounts of data, is separate from the CPU and is connected to it by wires on the computer's circuit board

Stored Program Concept

- “Fetch-Decode-Execute” cycle

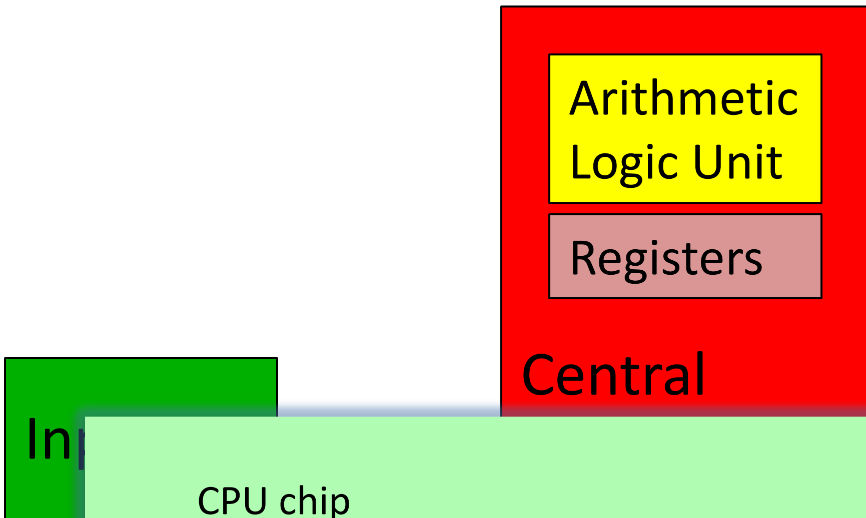


Central Processing Unit (CPU)

- ALU + Control = Processor
- Registers. Storage cells that holds heavily used program data
- Without address, specific purpose
- e.g. the operands of an arithmetic operation, the result of an operation, etc.

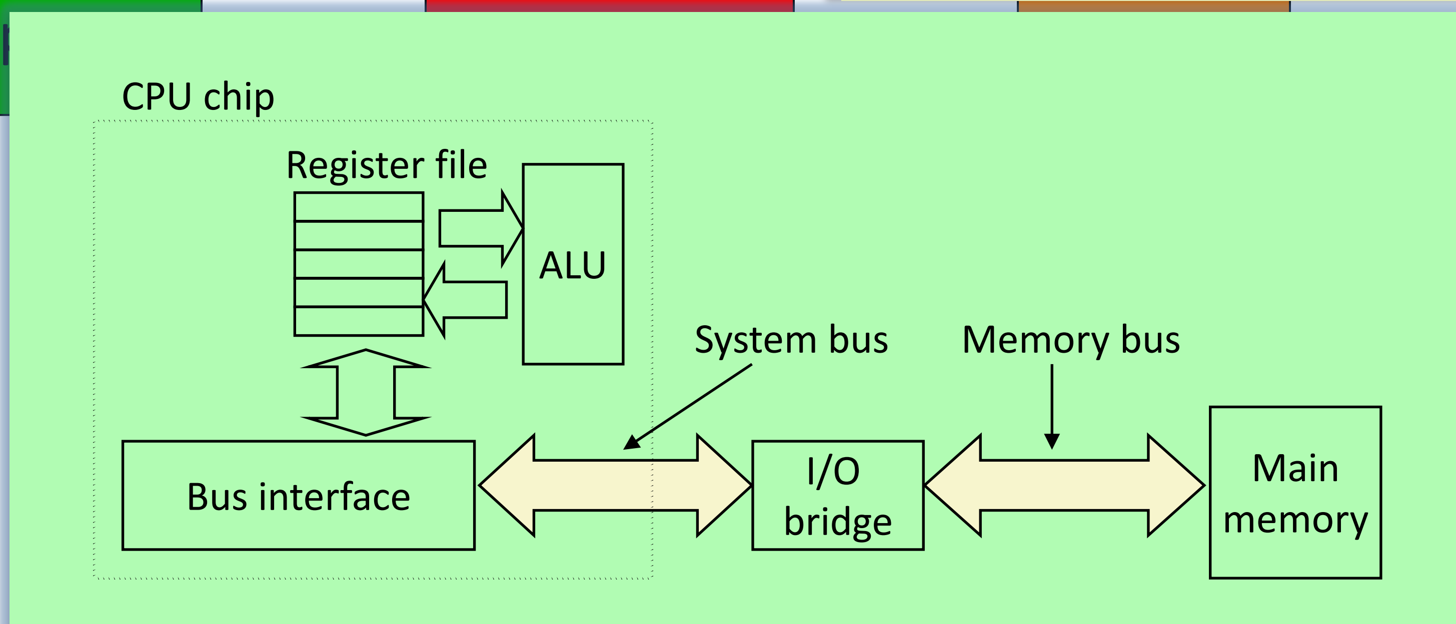
Stored Program Concept

- “Fetch-Decode-Execute” cycle



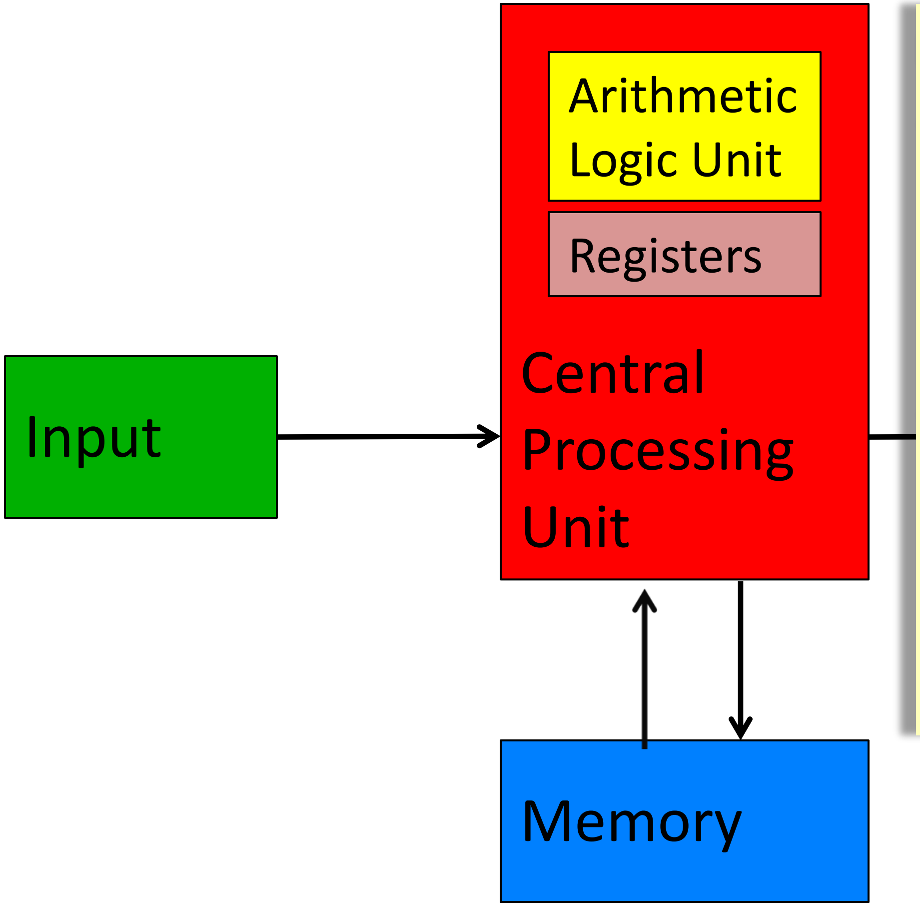
BUS

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



Stored Program Concept

- "Fetch-Decode-Execute" cycle

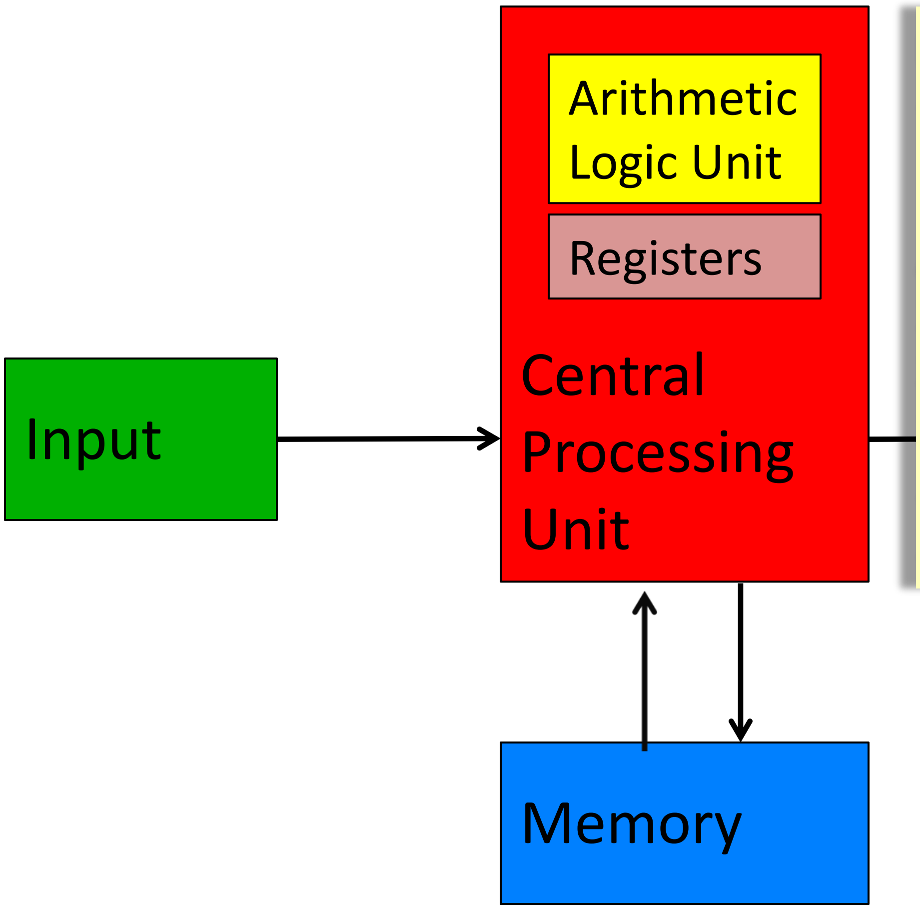


Memory

- A program, which is usually a long list of instructions, is stored in the main memory, and is copied, one instruction at a time, into a register in the CPU for execution
- The CPU has two special registers: a **program counter** that keeps track of the location in memory where it will find the next instruction and an **instruction register** that stores the next instruction to execute

Stored Program Concept

- "Fetch-Decode-Execute" cycle



Memory

- Instructions, like data, can be encoded as numbers

Address	Contents
200	1000 0001 (ADD to R1)
201	0110 0110 (data value 102)
202	1001 0001 (ADD to R1)
203	0110 0110 (data at address 102)
204	1111 0111 (JUMP 7 bytes)

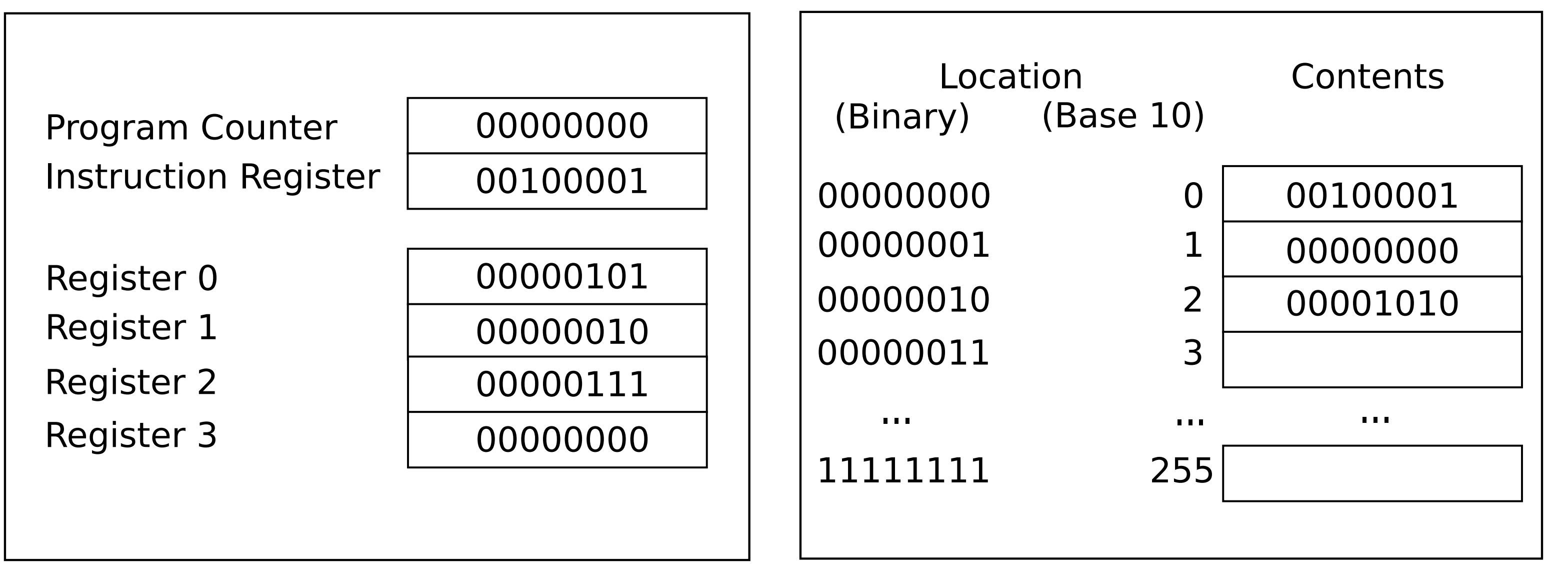
von Neumann Architecture

Opcode	Meaning
00	Add
01	Subtract
10	Multiply
11	Divide

- Let’s assume an 8-bit computer with only four instructions:
  - add, subtract, multiply, and divide
- Each of the instructions will need a number, which is called an operation code (or opcode), to represent it
- Next, let’s assume that our computer has four registers, numbered 0 through 3, and 256 8-bit memory cells
- An instruction will be encoded as: the first two bits represent the instruction, the next two bits encode the “destination register”, the next four bits encode the registers containing two operands
- For example, the instruction `add 3 0 2` (meaning add the contents of register 2 with the contents of register 0 and store the result in register 3) will be encoded as `00110010`

von Neumann Architecture

- Our computer operates by repeatedly performing the following procedure
  1. Send the address in the program counter (commonly called the PC) to the memory, asking it to read that location
  2. Load the value from memory into the instruction register
  3. Decode the instruction register to determine what instruction to execute and which registers to use
  4. Execute the requested instruction, which involves reading operands from registers, performing arithmetic, and sending the results back to the destination register
  5. Increment PC so that it contains the address of the next instruction in memory



Central Processing Unit (CPU) Memory

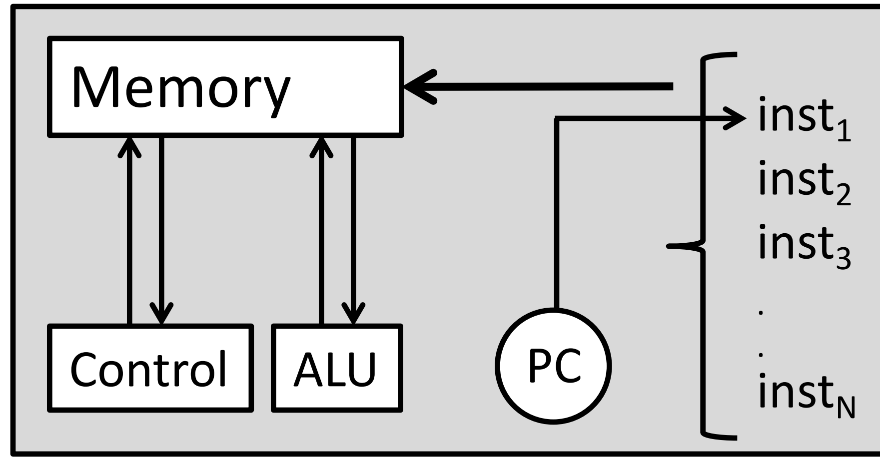
Assembly Language

- A low-level programming language for computers
- More readable, English-like abbreviations for instructions
- Architecture-specific
- Example:

```
MOV AL, 61h
MOV AX, BX
ADD EAX, 10
XOR EAX, EAX
```

Summary: Components of a Computer

- Sequential execution of machine instructions
  - The sequence of instructions are stored in the memory.
  - ■ One instruction at a time is fetched from the memory to the control unit.
  - They are read in and treated just like data.



- PC (program counter) is responsible from the flow of control.

- PC points a memory location containing an instruction on the sequence.

- Early programmers (coders) write programs via machine instructions.

Lecture Overview

- Building a Computer
- The Harvey Mudd Miniature Machine (HMMM)

The Harvey Mudd Miniature Machine (HMMM)

- HMMM
- A Simple HMMM Program
- Looping
- Functions
- HMMM Instruction Set

HMMM

- A real computer must be able to
  - Move information between registers and memory
  - Get data from the outside world
  - Print results
  - Make decisions
- The Harvey Mudd Miniature Machine (HMMM) is organized as follows
  - Both instructions and data are 16 bit wide
  - In addition to the program counter and instruction register, there are 16 registers named `r0` through `r15`
  - There are 256 memory locations
- Instead of programming in binary (0's and 1's), we'll use assembly language, a programming language where each instruction has a symbolic representation
- For example, to compute `r3 = r1+r2`, we'll write `add r3 r1 r2`
- We'll use a program to convert the assembly language into 0's and 1's – the machine language – that the computer can execute

[1] http://shickey.github.io/HMMM.js

A Simple HMMM Program

triangle1.hmmm: Calculate the approximate area of a triangle.

```
0     read     r1        # Get base b
1     read     r2        # Get height h
2     mul      r1 r1 r2  # b times h into r1
3     setn     r2 2
4     div      r1 r1 r2  # Divide by 2
5     write    r1
6     halt
```

Assemble! →

```
----------------------
| ASSEMBLY SUCCESSFUL |
----------------------

0 : 0000 0001 0000 0001        0     read     r1        # Get base b
1 : 0000 0010 0000 0001        1     read     r2        # Get height h
2 : 1000 0001 0001 0010        2     mul      r1 r1 r2  # b times h into r1
3 : 0001 0010 0000 0010        3     setn     r2 2
4 : 1001 0001 0001 0010        4     div      r1 r1 r2  # Divide by 2
5 : 0000 0001 0000 0010        5     write    r1
6 : 0000 0000 0000 0000        6     halt
```

Simulate! →

```
4
5
10
```

Looping

- Unconditional jump (`jumpn N`): set program counter to address `N`

triangle2.hmmm: Calculate the approximate areas of many triangles.

```
0     read    r1        # Get base b
1     read    r2        # Get height h
2     mul     r1 r1 r2  # b times h into r1
3     setn    r2 2
4     div     r1 r1 r2  # Divide by 2
5     write   r1
6     jumpn   0
```

Simulate! →

```
4
5
10
5
5
12
<ctrl-d>

End of input, halting program execution...
```

Looping

- Conditional jump (`jeqzn rX N`): if `rX == 0`, then jump to line `N`

triangle3.hmmm: Calculate the approximate areas of many triangles. Stop when a base or height of zero is given.

```
0     read     r1         # Get base b
1     jeqzn    r1 9       # Jump to halt if base is zero
2     read     r2         # Get height h
3     jeqzn    r2 9       # Jump to halt if height is zero
4     mul      r1 r1 r2 # b times h into r1
5     setn     r2 2
6     div      r1 r1 r2 # Divide by 2
7     write    r1
8     jumpn    0
9     halt
```

Simulate! ➡

```
4
5
10
5
5
12
0
```

Looping

is_it_a_prime_number.hmmm: Calculate whether a given positive number is prime or not

```
0       read   r1          # read the number. Please enter positive integers.
1       setn   r2 2        # use this register for arithmetic operations with 2.
2       setn   r9 1        # use this register for arithmetic operations with 1.
3       sub    r15 r1 r9
4       jeqzn r15 17       # check if the number is 1
5       div    r3 r1 r2    # Divide to 2. The biggest divider (denominator) should (may) be this number.
6       nop                # there is no reason. Deleted a line, but too lazy to change all the line numbers.
        # the number is 2 or 3. So it is prime.
7       sub    r15 r3 r9
8       jeqz   r15    15
        # The number is not 1, 2 or 3. The main loop starts here------------------
9       mod    r15 r1 r3    # mod to check if the number is aliquot.
10      jeqzn r15   17     # it is not a prime number. Jump to line 17.
11      sub    r3 r3 r9     # subtract one from the divider
12      sub    r5 r3 r9     # subtract one, but on a different register to check the divider is 1 or not.
13      jeqz   r5 15        # we succesfully reduced the divider to 1. This is a prime number. Jump to line 15.
14      jumpn   9          # jump to the start of the main loop.
        #--------------------------------- Write 1 for prime numbers.
15      write  r9    # r9 is already 1.
16      halt
        #--------------------------------- Write 0 for non-prime numbers.
17      setn    r8 0
18      write  r8
19      halt
```

Simulate! →

```
17
1
```

Functions

- Call a function (`calln rX N`): copy the next address (aka return address) into `rX` and then jump to address `N`
- Return from a function (`jumpr rX`): set program counter to the return address in `rX`
- By convention, we use register `r14` to store the return address

square.hmmm: Calculate the square of a number N.

```
0     read     r1     # Get N
1     calln    r14 5  # Calculate N^2
2     write    r2     # Write answer
3     halt
4     nop             # Waste some space

# Square function. N is in r1. Result (N^2) is in r2. Return address is in r14.
5     mul      r2 r1 r1 # Calculate and store N^2 in r2
6     jumpr    r14      # Done; return to caller
```

Simulate! ➔

```
11
121
```

Functions

combinations.hmmm: Calculate $C(N, K)$ (aka N choose K) defined as $C(N, K) = N!/(K!(N-K)!)$, where $N!$ (N factorial) is defined as $N! = N \times (N-1) \times (N-2) \times \cdots \times 2 \times 1$, with $0! = 1$.

```
0     read     r3        # Get N
1     read     r4        # Get K
2     copy     r1 r3     # Calculate N!
3     calln    r14 15    # ...
4     copy     r5 r2     # Save N! as C(N, K)
5     copy     r1 r4     # Calculate K!
6     calln    r14 15    # ...
7     div      r5 r5 r2  # N!/K!
8     sub      r1 r3 r4  # Calculate (N - K)!
9     calln    r14 15    # ...
10    div      r5 r5 r2  # C(N, K)
11    write    r5        # Write answer
12    halt
13    nop                # Waste some space
14    nop

# Factorial function. N is in r1. Result is r2. Return address is in r14.
15     setn     r2 1      # Initial product
16     jeqzn    r1 20     # Quit if N has reached zero
17     mul      r2 r1 r2  # Update product
18     addn     r1 -1     # Decrement N
19     jumpn    16        # Back for more
20     jumpr    r14       # Done; return to caller
```

Simulate! →

```
5
2
10
```

Functions

Trace of the factorial function (N=4)

instruction	r1	r2
	4	
15 setn r2 1	4	1
16 jeqzn r1 20	4	1
17 mul r2 r1 r2	4	4
18 addn r1 -1	3	4
19 jumpn 16	3	4
16 jeqzn r1 20	3	4
17 mul r2 r1 r2	3	12
18 addn r1 -1	2	12
19 jumpn 16	2	12
16 jeqzn r1 20	2	12
17 mul r2 r1 r2	2	24
18 addn r1 -1	1	24
19 jumpn 16	1	24
16 jeqzn r1 20	1	24
17 mul r2 r1 r2	1	24
18 addn r1 -1	0	24
19 jumpn 16	0	24
16 jeqzn r1 20	0	24
20 jumpr r14	0	24

Trace of the program (N=5, K=2)

instruction	r1	r2	r3	r4	r5	r14
0 read r3			5			
1 read r4			5	2		
2 copy r1 r3	5		5	2		
3 calln r14 15	5	120	5	2		4
4 copy r5 r2	5	120	5	2	120	4
5 copy r1 r4	2	120	5	2	120	4
6 calln r14 15	2	2	5	2	120	7
7 div r5 r5 r2	2	2	5	2	60	7
8 sub r1 r3 r4	3	2	5	2	60	7
9 calln r14 15	3	6	5	2	60	10
10 div r5 r5 r2	3	6	5	2	10	10
11 write r5	3	6	5	2	10	10
12 halt	3	6	5	2	10	10

HMMM Instruction Set

- **System instructions**

`halt`	stop
`read rX`	place user input in register `rX`
`write rX`	print contents of register `rX`
`nop`	do nothing

- **Setting register data**

`setn rX N`	set register `rX` equal to the integer `N` (-128 to 127)
`addn rX N`	add integer `N` (-128 to 127) to register `rX`
`copy rX rY`	set `rX=rY`

- **Arithmetic**

`add rX rY rZ`	set `rX=rY+rZ`
`sub rX rY rZ`	set `rX=rY-rZ`
`neg rX rY`	set `rX=-rY`
`mul rX rY rZ`	set `rX=rY*rZ`
`div rX rY rZ`	set `rX=rY/rZ` (integer division; no remainder)
`mod rX rY rZ`	set `rX=rY%rZ` (returns the remainder of integer division)

HMMM Instruction Set

- **Jumps**

`jumpn N` set program counter to address `N`

`jumpr rX` set program counter to address in `rX`

`jeqzn rX N` if `rX==0`, then jump to line `N`

`jnezn rX N` if `rX!=0`, then jump to line `N`

`jgtzn rX N` if `rX>0`, then jump to line `N`

`jltzn rX N` if `rX<0`, then jump to line `N`

`calln rX N` copy the next address into `rX` and then jump to address `N`

- **Interacting with memory**

`loadn rX N` load register `rX` with the contents of address `N`

`storen rX N` store contents of register `rX` into address `N`

`loadr rX rY` load register `rX` with data from the address location held in register `rY`

`storer rX rY` store contents of register `rX` into address held in register `rY`