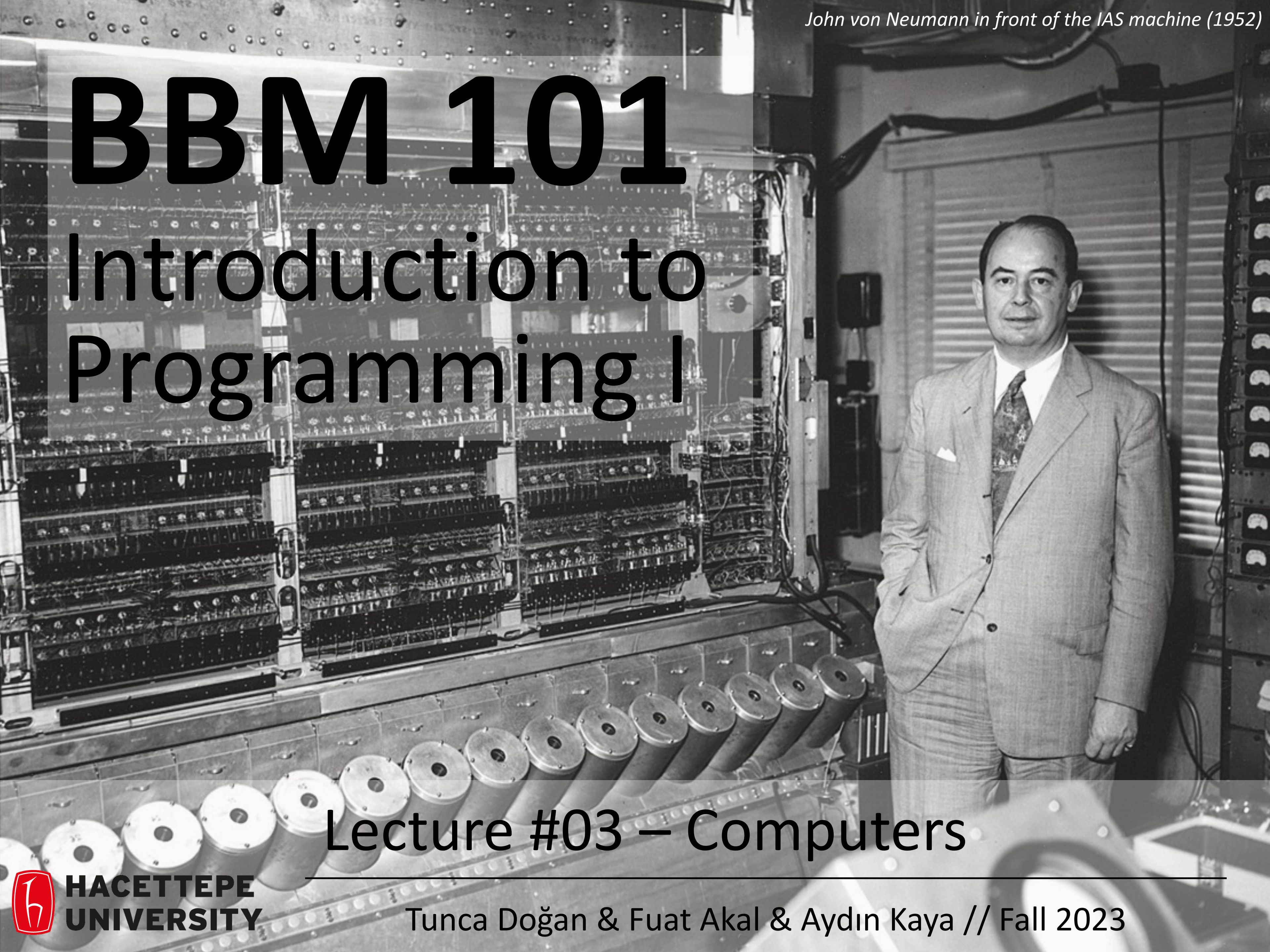


John von Neumann in front of the IAS machine (1952)

BBM 101

Introduction to Programming I

Lecture #03 – Computers

Tunca Doğan & Fuat Akal & Aydın Kaya // Fall 2023

HACETTEPE UNIVERSITY

Last time... **Introduction to Algorithms**

- An *algorithm* is a recipe for solving a problem.

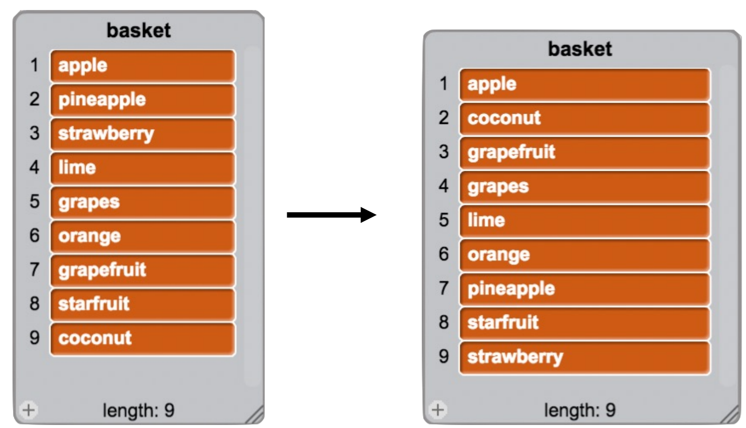
Problem Specification
Input: Some stuff!
OUTPUT: Information about the stuff!

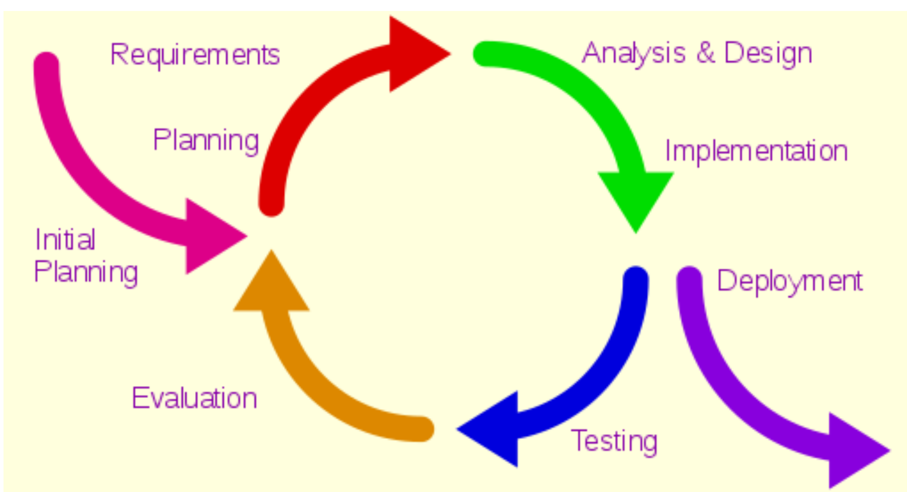
Search Problem

- *Input:*
  - a list of objects
  - a specific object
- *Output:*
  - True if the object is in list
  - False if the object is *not* in list

Sorting Problem

- *Input:*
  - a collection of orderable objects
- *Output:*
  - a collection where each item is in order





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
- 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

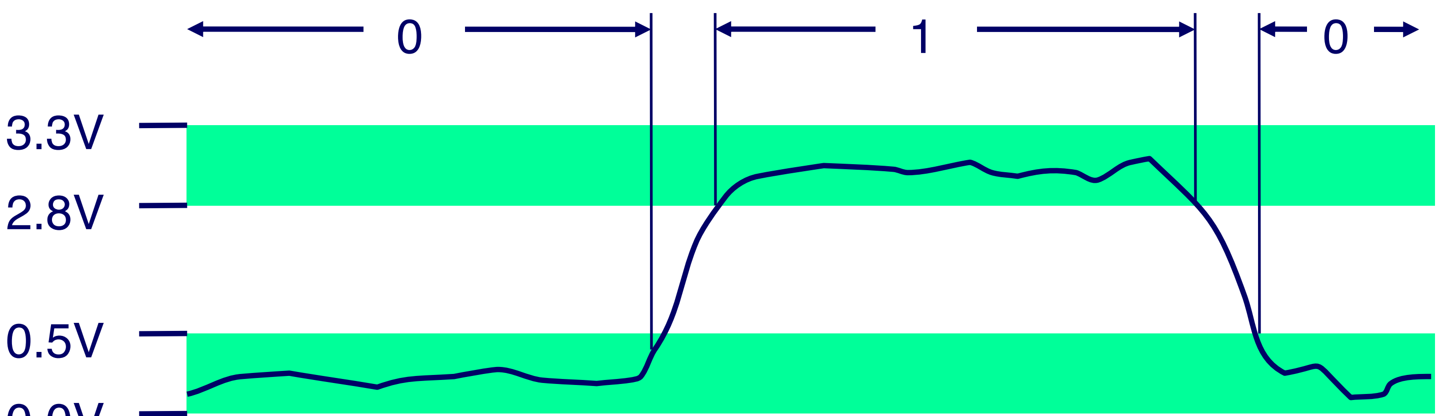


Image from: R.E. Bryant, D.R. O'Hallaron, G. Kesden

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

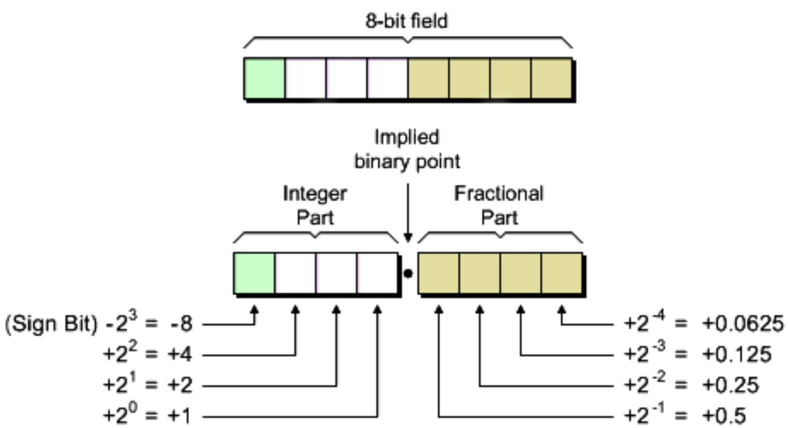
- Arithmetic in any base is analogous to arithmetic in base 10
- Examples of addition in base 10 and base 2

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

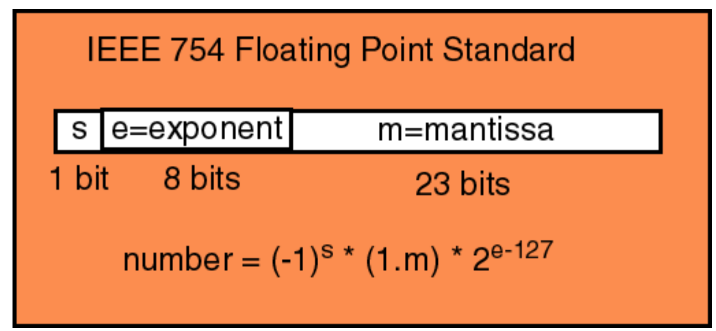
- To represent a negative integer, a computer typically uses a system called **two's complement**, which involves flipping the bits of the positive number and then adding 1
- For example, on an 8-bit computer, 3 = 00000011, so
  −3 = 11111101

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



An 8-bit field representing a fixed-point number



$$0\underbrace{10000001}_{2^7+2^0=129}\underbrace{10100000000000000000000}_{2^{-1}+2^{-3}=0.625} = -1^0 \times 1.101_2 \times 2^{129-127}$$
$$= 1.625 \times 2^2$$
$$= 6.5$$

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 selected set of special characters
- **ASCII**, being an 8-bit code, can only represent 256 different symbols, and does not provide for characters used in many languages

USASCII code chart

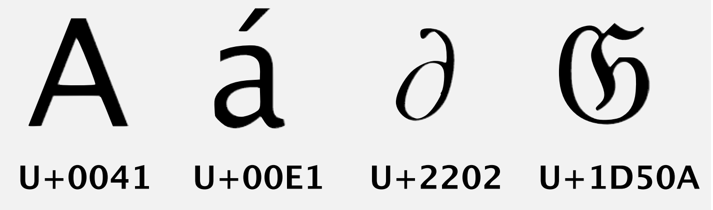
b_4	b_3	b_2	b_1	Row \ Column	0 ($b_7b_6b_5$ = 000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

	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	Goog[d]	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

Binary	Oct	Dec	Hex		
110 0001	141	97	61		a
110 0010	142	98	62		b
110 0011	143	99	63		c
110 0100	144	100	64		d
110 0101	145	101	65		e
110 0110	146	102	66		f
110 0111	147	103	67		g
110 1000	150	104	68		h
110 1001	151	105	69		i
110 1010	152	106	6A		j
110 1011	153	107	6B		k
110 1100	154	108	6C		l
110 1101	155	109	6D		m
110 1110	156	110	6E		n
110 1111	157	111	6F		o
111 0000	160	112	70		p
111 0001	161	113	71		q
111 0010	162	114	72		r
111 0011	163	115	73		s
111 0100	164	116	74		t

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

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 some place other than the next instruction

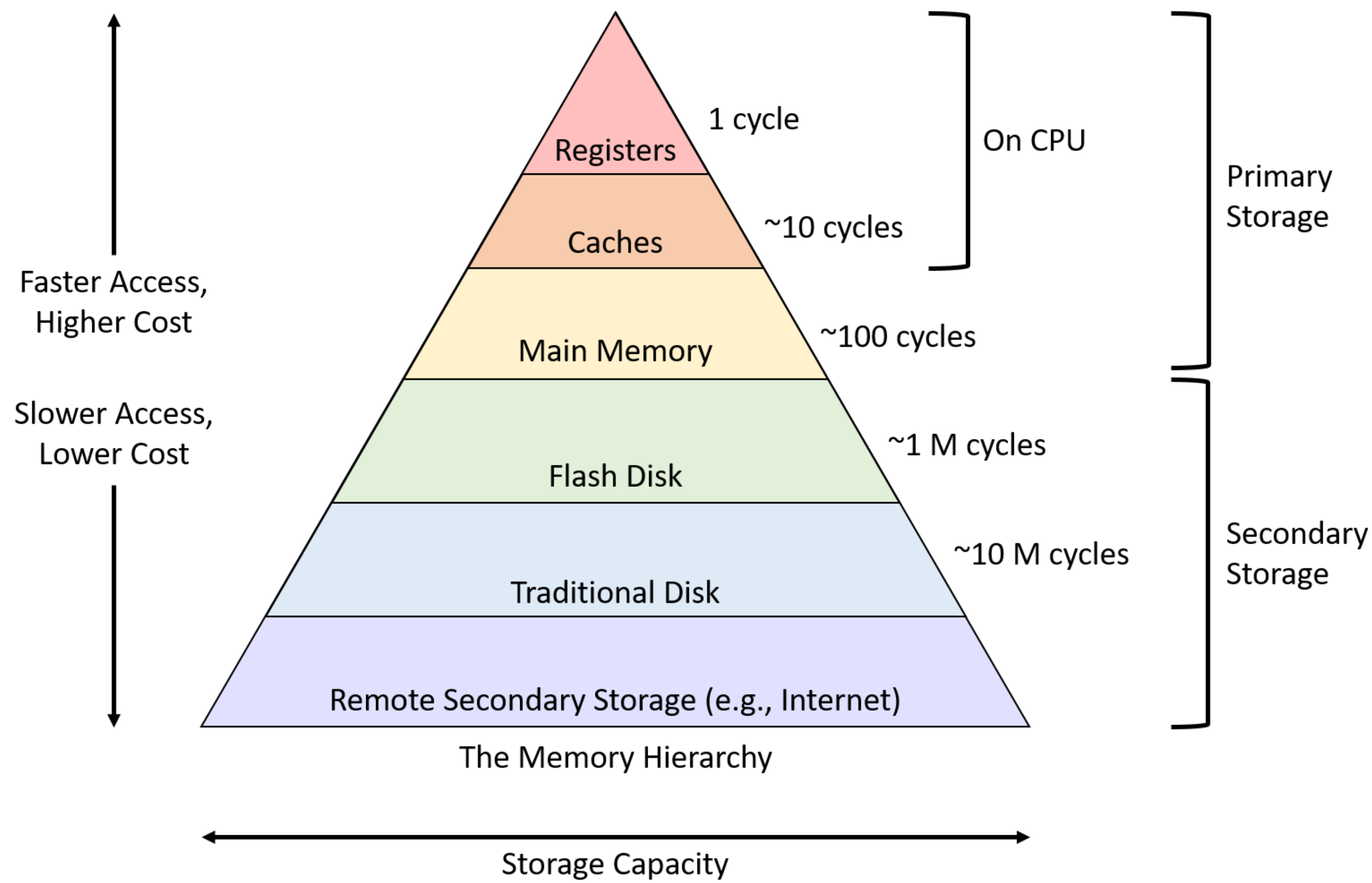
Stored Program Concept

- 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

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

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

Memory Hierarchy

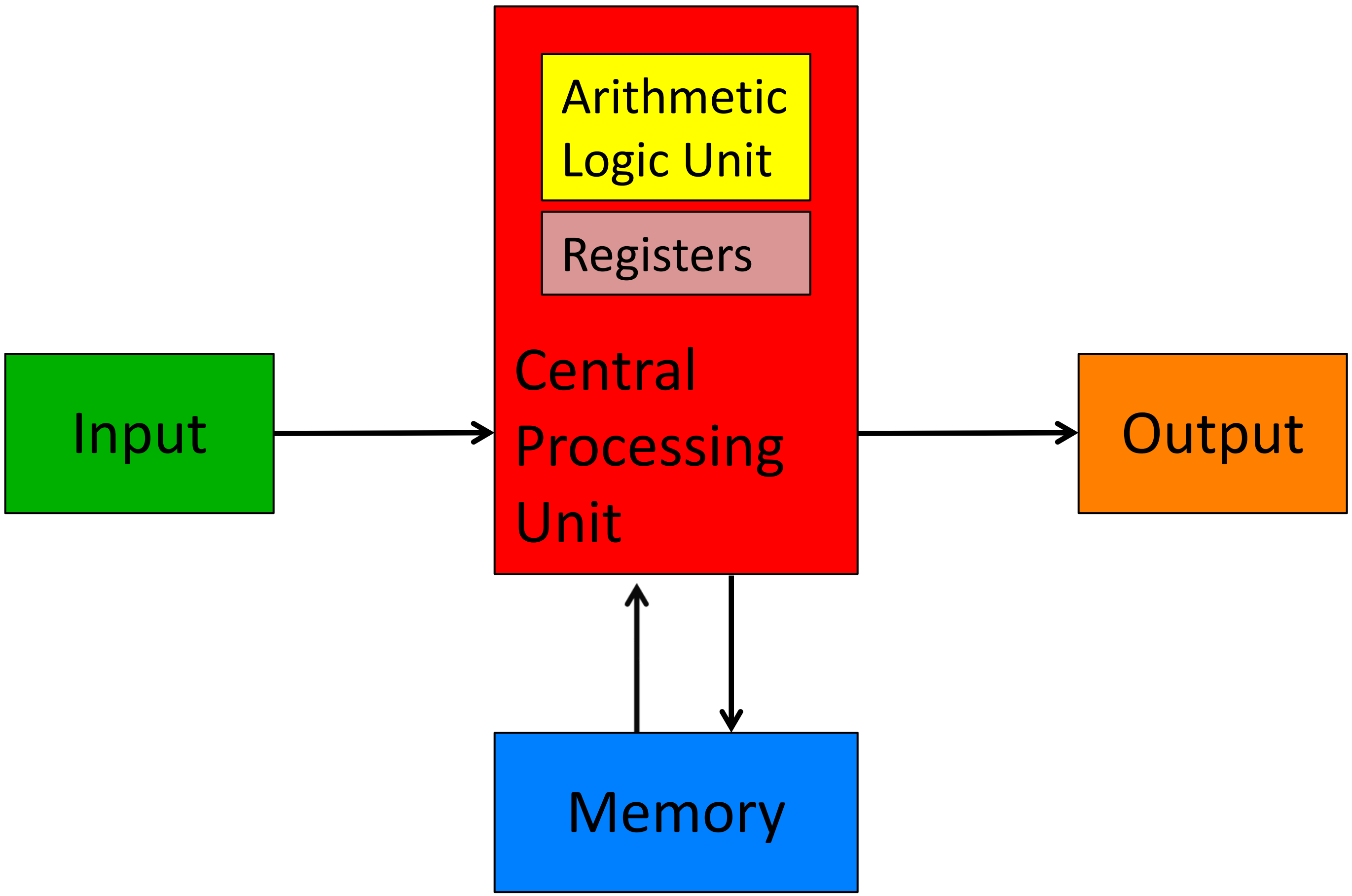


The Memory Hierarchy

https://diveintosystems.org/book/C11-MemHierarchy/mem_hierarchy.html

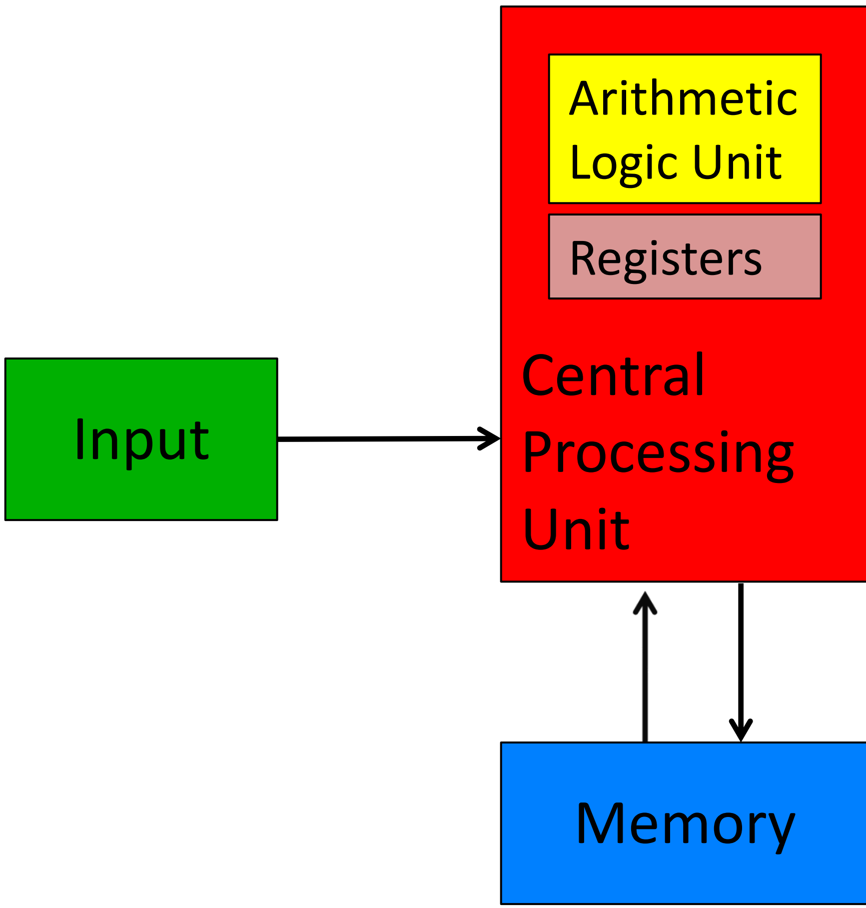
Stored Program Concept

- “Fetch-Decode-Execute” cycle



Stored Program Concept

- "Fetch-Decode-Execute" cycle

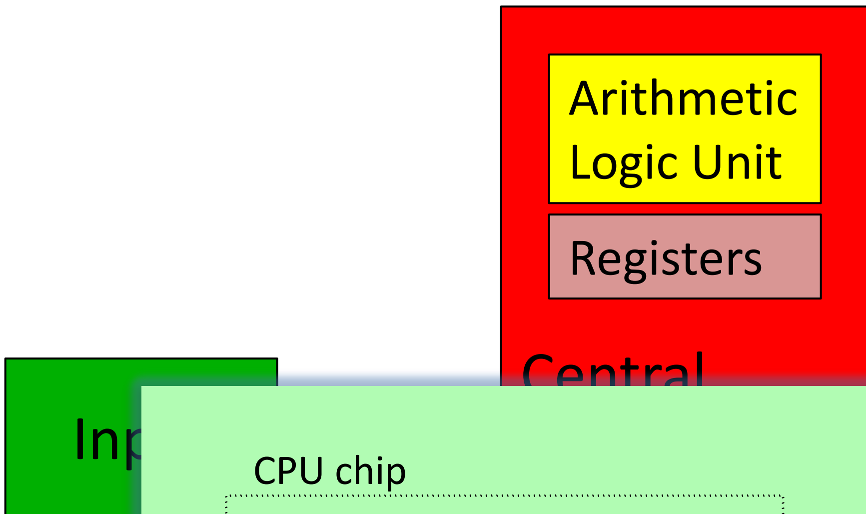


Central Processing Unit (CPU)

- The CPU is where all the computation takes place
- ALU + Control = Processor
- The CPU has devices such as adders, multipliers, etc. for doing arithmetic. It has a small amount of (scratch) memory called registers

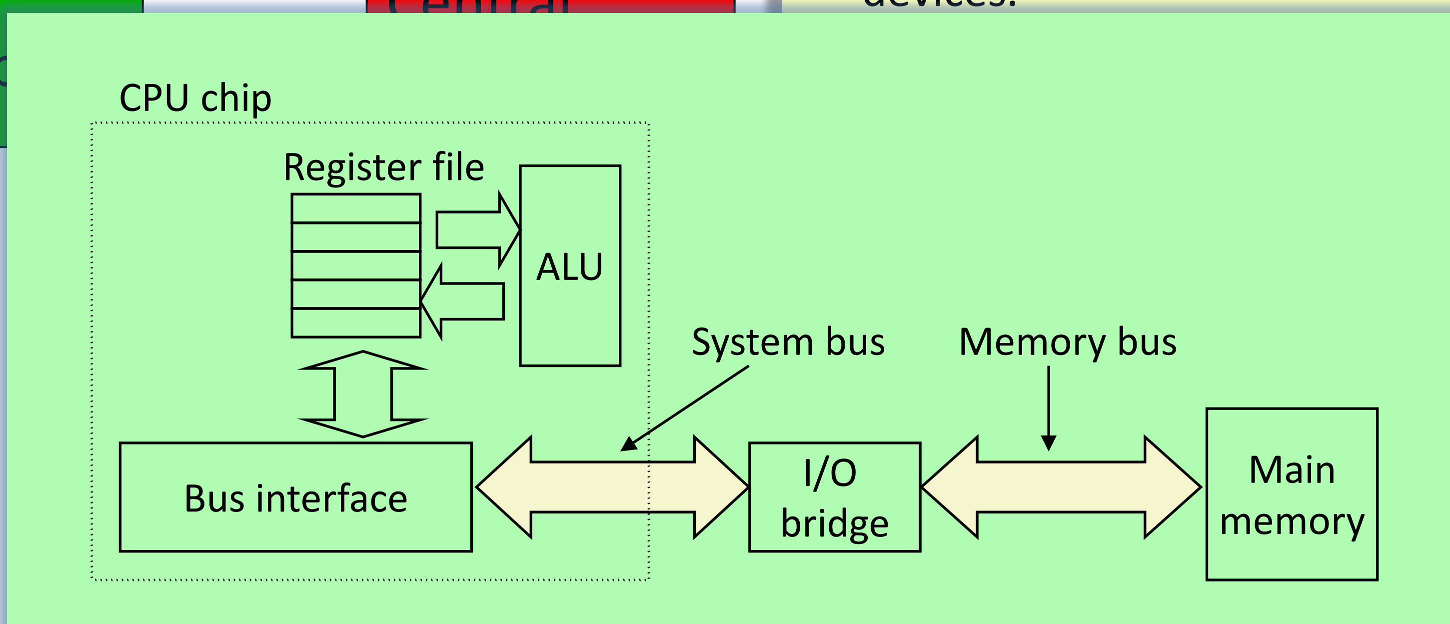
Stored Program Concept

- “Fetch-Decode-Execute” cycle

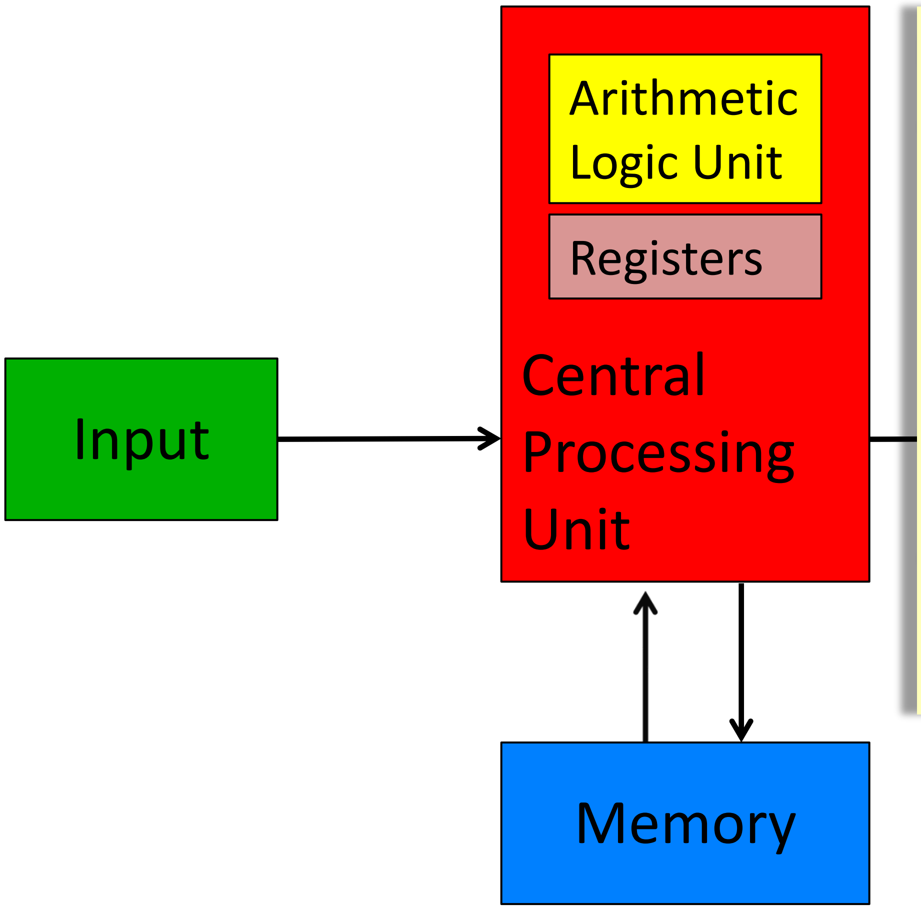


BUS

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



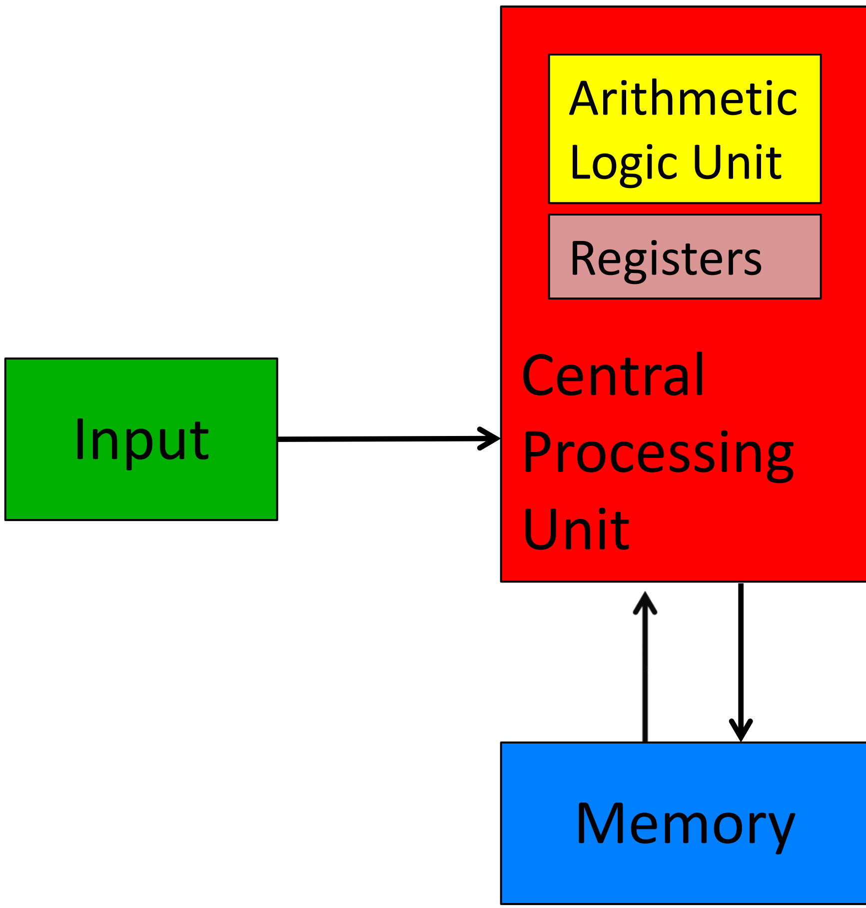
Stored Program Concept



Memory

- A program, a long list of instructions, is stored in the main memory,
  - One instruction at a time, taken into register in the CPU for execution
- The CPU has some special and general purpose registers.
- **Program counter** is the most important register that keeps track of the instructions to be executed.

Stored Program Concept



Memory

- Instructions, like data, can be encoded as numbers

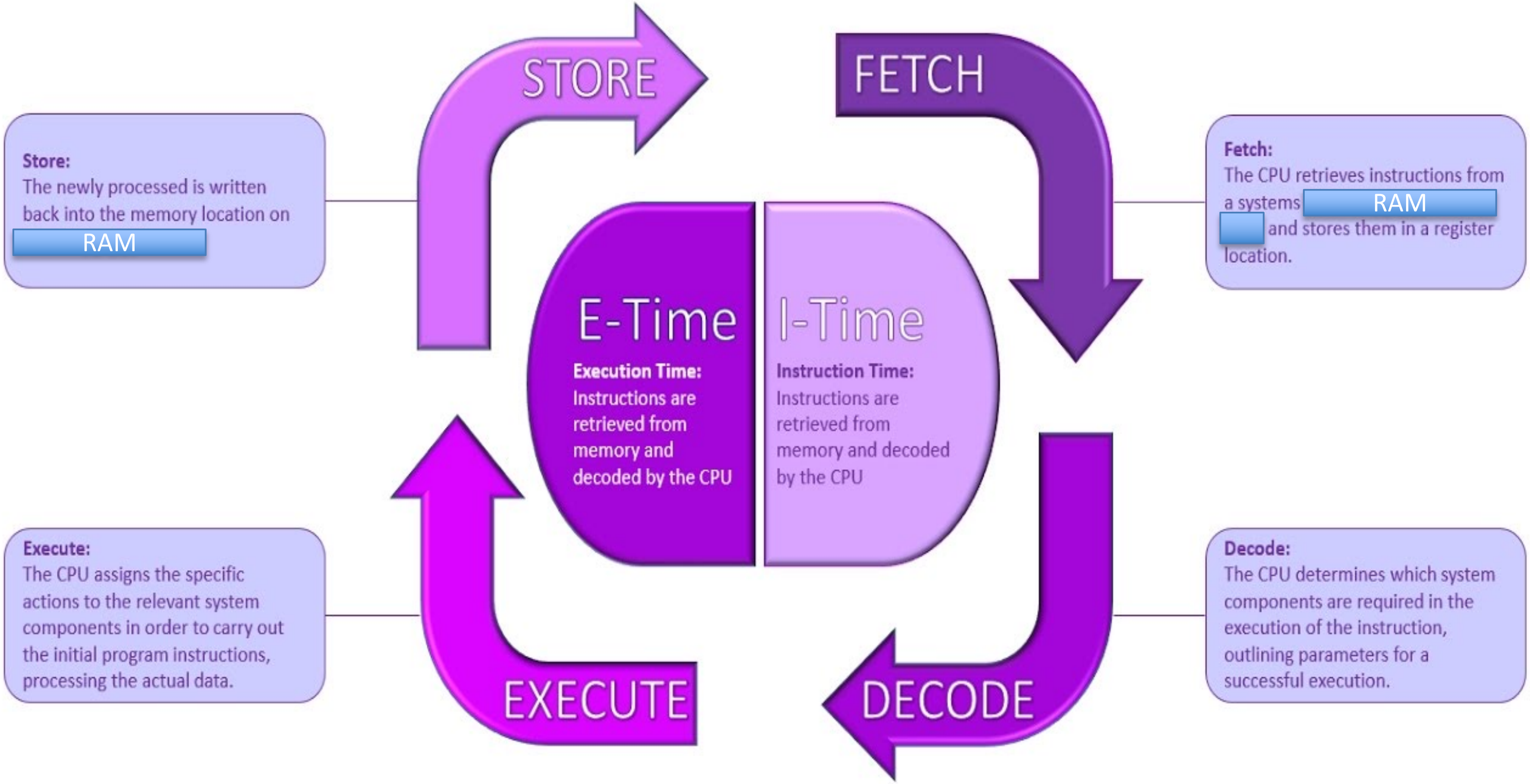
Address	Contents
200	1000 0001 (ADD to R1)
201	0110 0110 (data value 102)
202	1001 0001 (ADD to R1)
203	0110 0110 (data at address 102)
204	1111 0111 (JUMP 7 bytes)

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 and register 0 and store the result in register 3) will be encoded as `00110010`

The Fetch-Execute Cycle



Source: https://www.youtube.com/watch?v=xs5oq-i_rTc

Algorithm for Program Execution

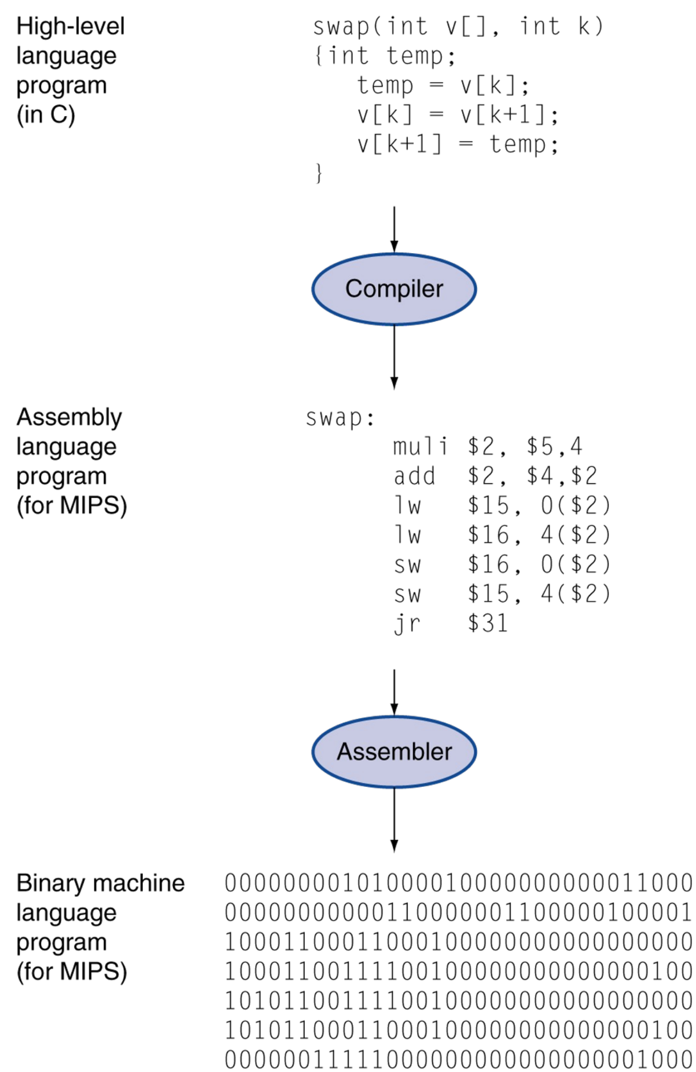
- PC (program counter) is set to the address where the first program instruction is stored in memory.

- Repeat until HALT instruction or fatal error
  - Fetch instruction
  - Decode instruction
  - Execute instruction

  End of loop

Levels of Program Code

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data



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
```

For example, the instruction below tells an x86/IA-32 processor to move an immediate 8-bit value into a register. The binary code for this instruction is 10110 followed by a 3-bit identifier for which register to use. The identifier for the *AL* register is 000, so the following machine code loads the *AL* register with the data 01100001.[17]

```
10110000 01100001
```

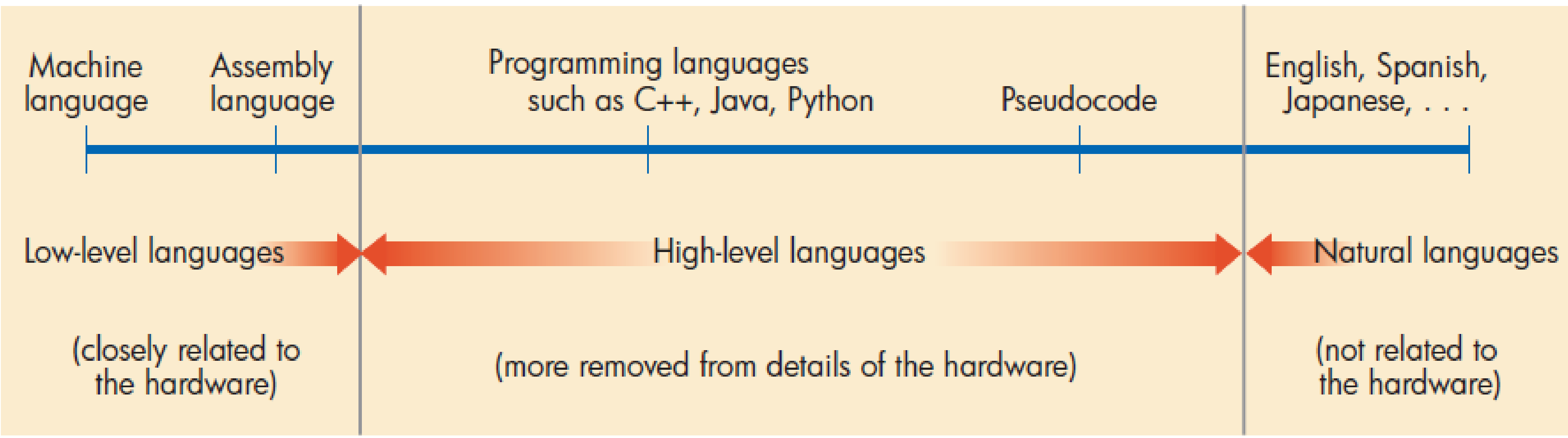
This binary computer code can be made more human-readable by expressing it in hexadecimal as follows.

```
B0 61
```

Here, `B0` means 'Move a copy of the following value into *AL*, and `61` is a hexadecimal representation of the value 01100001, which is 97 in decimal. Assembly language for the 8086 family provides the mnemonic MOV (an abbreviation of *move*) for instructions such as this, so the machine code above can be written as follows in assembly language, complete with an explanatory comment if required, after the semicolon. This is much easier to read and to remember.

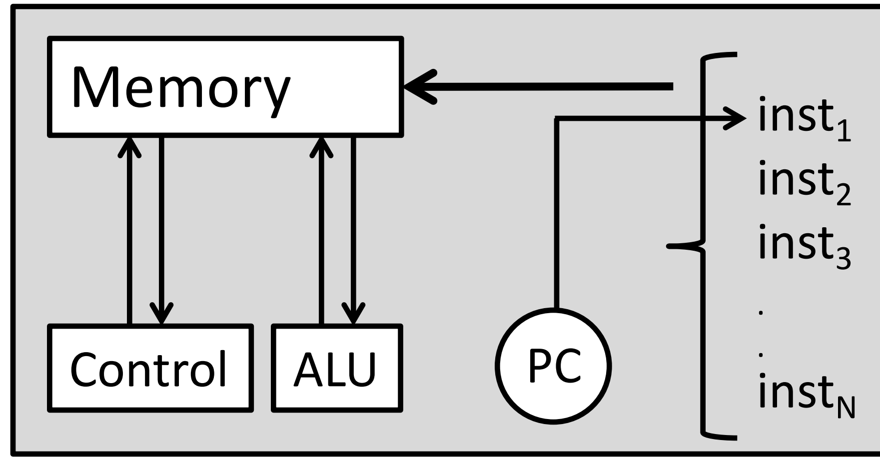
```
MOV AL, 61h       ; Load AL with 97 decimal (61 hex)
```

Continuum of Programming Languages



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 at a memory location containing the instruction being executed at the current time.

- Early programmers (coders) used to 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

The Harvey Mudd Miniature Machine (HMMM)

- Hmmm (Harvey Mudd Miniature Machine) is a 16-bit, 23-instruction simulated assembly language with 2^8=256 16-bit words of memory.

- In addition to the **program counter** and **instruction register**, there are 16 registers named `r0` through `r15`.

Hmmm assembly code

```
0     read     r1
1     read     r2
2     mul      r1  r1  r2
3     setn     r2  2
4     div      r1  r1  r2
5     write    r1
6     halt
```

→

Corresponding instructions in machine language

```
0000  0001  0000  0001
0000  0010  0000  0001
1000  0001  0001  0010
0001  0010  0000  0010
1001  0001  0001  0010
0000  0001  0000  0010
0000  0000  0000  0000
```

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 will write `add r3 r1 r2`
- We will 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 successfully 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`