

The CDC 6504 Microprocessor

Dr. Charles R. Severance

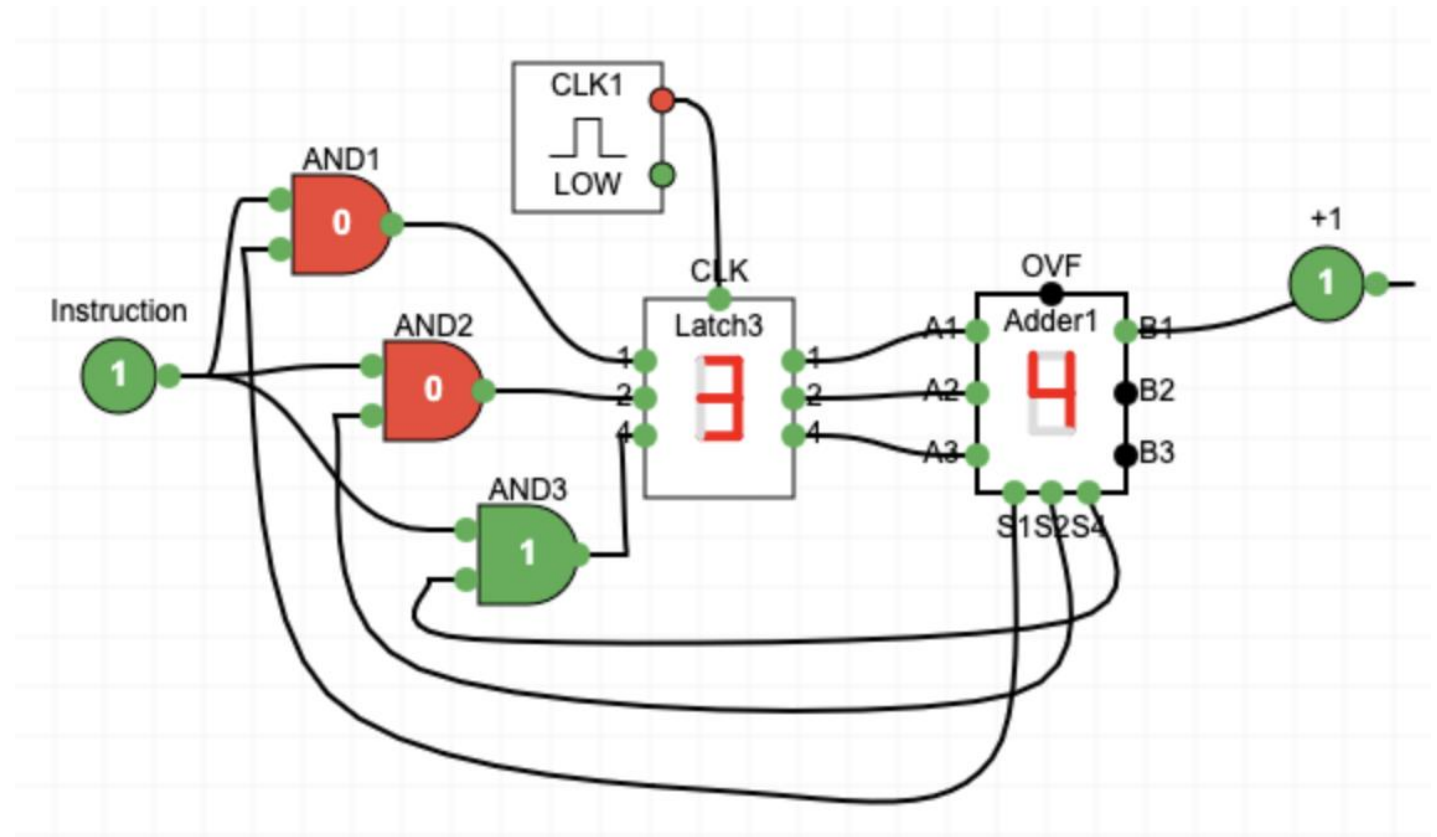
online.dr-chuck.com

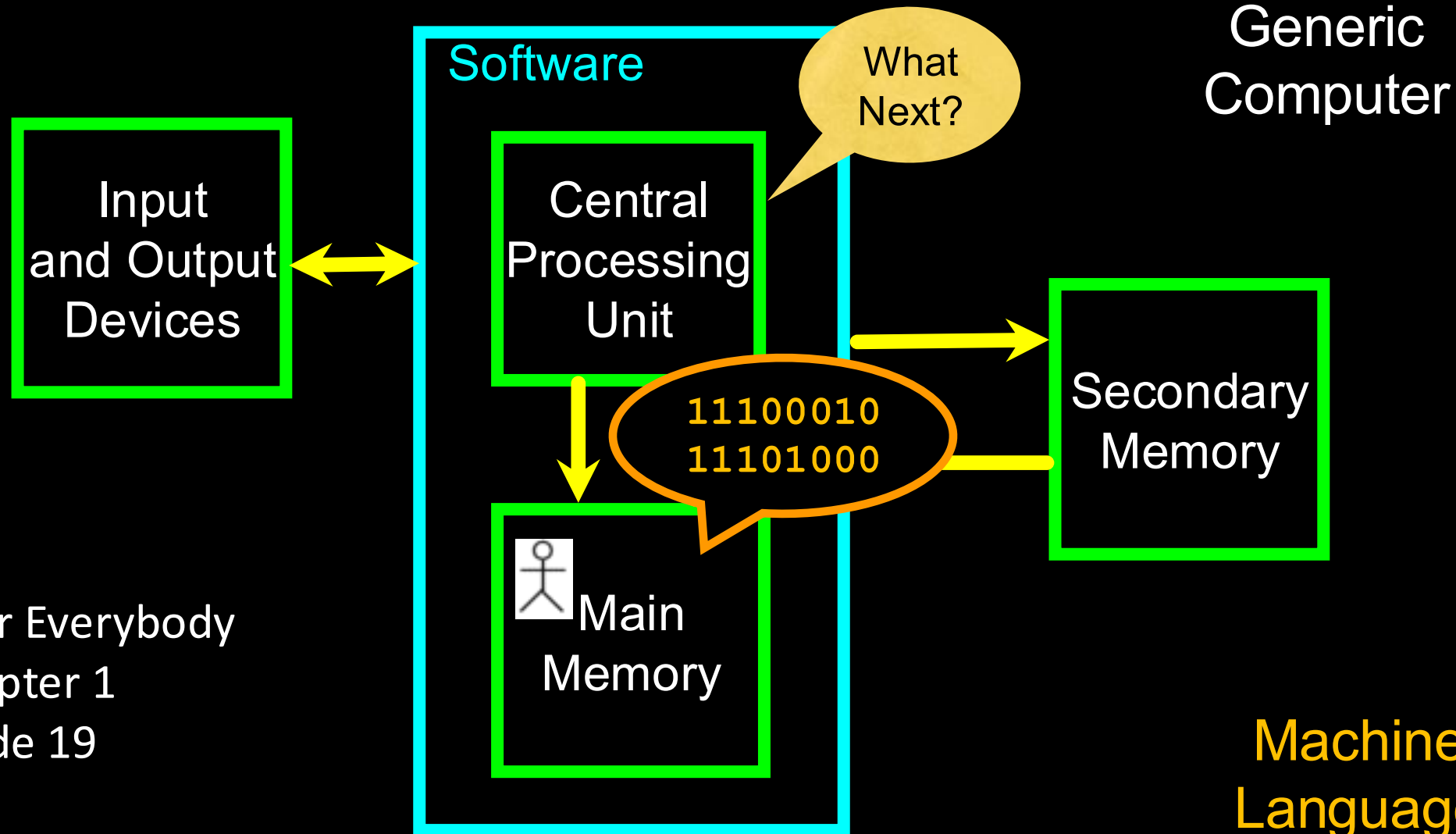
Outline

- Start from our "two instruction CPU" from the previous section
- Hexadecimal Notation
- Review: What is Programming from PY4E
- Real CPUs – Intel x86, ARM, and MOS 6502
- The CDC6504 – Mostly 6502 with homage to CDC 6500
 - Architecture
 - Machine Language
 - Assembly Language

Review: What is missing?

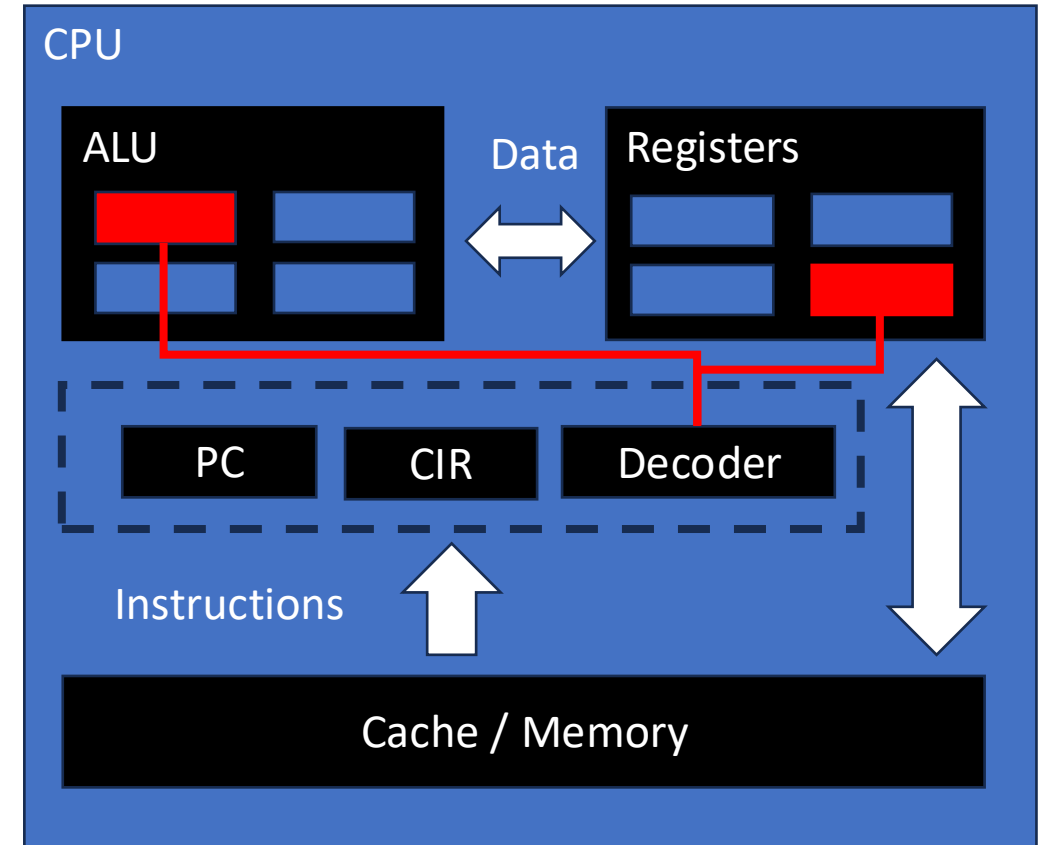
- The instructions are not being read from memory.
- It is like there is a switch on the front of the computer and it chooses between the two instructions
- We want a flexible sequence of many combinations of instruction sequences(a.k.a. Software)





Fetch-Decode-Execute-Cycle

- Program Counter (PC) Register – Address of the next machine code instruction
- Current Instruction Register (CIR)
- The decoder looks at the bits of the CIR and using gates activates the correct ALU Element and register
- The Clock causes the logic when to compute and when to copy results back to the register



https://en.wikipedia.org/wiki/Binary_decoder

Machine Language Instructions

- As part of the documentation for a CPU, we are given the bit patterns of the machine code for each of its instructions
- There is usually a symbolic language that translates to machine language (assembler)

Assembly	Opcode	Description
CLX	11100010	Clear X register ($X = 0$).
INX	11101000	Increment X register by 1
LDX #value	10100010	Load X register with immediate (constant) value
LDX \$address	10100110	Load X register from zero-page memory address
STX \$address	10000110	Store X register to zero-page memory address

Let's learn Base-16 (Hexadecimal)

Base 16 is easy to convert to bits and much less verbose than Base-2

Recall Base-2

- Base-2 to Base-10 for numbers less than 8
- A.k.a. 3-bit numbers

4s 2s 1s

$$1 \ 1 \ 0 = 6$$

Base-10	Base-2
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

The first problem with Base-16

- We only have 0-9 and we need to be able to represent 0-15 in a single base-16 digit
- Thankfully we have emoji's
 - 10 🦊
 - 11 🙌
 - 12 🙏
 - 13 🙋 (also Rock on!)
 - 14 🍀
 - 15 🖐

Base-10	Base-2	Base-16
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	🦊
11	1011	🙌
12	1100	🙏
13	1101	🙋
14	1110	🍀
15	1111	🖐

AI suggested these emoji's

- AI suggested emoji's
 - 10 🕒
 - 11 🕒
 - 12 🕒
 - 13 🕒 (1PM is 1300 hours)
 - 14 🕒 (2PM is 1400 hours)
 - 15 🕒 (3 PM is 1500 hours)
- American Sign Language(ASL) uses one hand but requires motion
- British Sign Language is different

Base-10	Base-2	Base-16
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	🕒
11	1011	🕒
12	1100	🕒
13	1101	🕒
14	1110	🕒
15	1111	🕒

No emojis in 1940



- We only have 0-9 as numbers and we need to be able to represent 0-15 in a single base-16 digit
- So we used A-F to represent 10-15 for base-16 numbers

Base-10	Base-2	Base-16
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Base-10 and Base-16 (Hexadecimal) numbers

- Need a cheat sheet

Base-10	Base-16
10	A
11	B
12	C
13	D
14	E
15	F

Base-10

$$\begin{aligned} 123_{10} &= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 \\ &= 1 \times 100 + 2 \times 10 + 3 \times 1 = 123_{10} \end{aligned}$$

Base-16

$$\begin{aligned} 7B_{16} &= 7 \times 16^1 + 11 \times 16^0 = \\ &= 112 + 11 = 123_{10} \end{aligned}$$

Base-16 to/from Base-2

- Since 2 and 16 are even powers of 2, each Base-16 represents exactly four base-2 digits so we can convert digit by digit



Hex is just more succinct Base-2

Base-10	Base-2	Base-16
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Keep it simple for now

- Go through base-10 if it helps

8s 4s 2s 1s

$$1\ 0\ 1\ 0 = 10_{10} = A_{16}$$

$$1\ 0\ 0\ 0 = 8_{10} = 8_{16}$$

$$1\ 1\ 1\ 1 = 15_{10} = F_{16}$$

Base-10	Base-2	Base-16
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Chuck's not-so-quick Guide to Base-16

- $F_{16} = 15_{10} = 1111_2$ (Easy one less than 16)
 - $E_{16} = 14_{10} = 1110_2$ (one less than 15)
 - $A_{16} = 10_{10} = 1010_2$ (8+2)
 - $B_{16} = 11_{10} = 1011_2$ (8+2+1)
 - $C_{16} = \text{😞} \text{ 🧠 } \text{👉} = 8+4 = 1100_2$
 - $D_{16} = \text{😞} \text{ 😞 } \text{👆} \text{ 👇 } \text{🧠} \text{ 📝 } \text{😡}$
-
- AI: What is a good way to be able to do hex to base 2 conversions in your head?

ASCII – 8 Bits

- ASCII Characters are 8-bits and can be nicely represented with two hex digits (a.k.a. nibbles)

American Standard Code for Information Interchange

<https://en.wikipedia.org/wiki/ASCII>

<http://www.catonmat.net/download/ascii-cheat-sheet.png>

<https://www.ca4e.com/tools/ascii/>

Dec	Hex	Oct	Bin	Char	Dec	Hex	Oct	Bin	Char	Dec	Hex	Oct	Bin	Char	Dec	Hex	Oct	Bin	Char
0	0x00	000	00000000	NUL	32	0x20	040	01000000	space	64	0x40	100	10000000	@	96	0x60	140	11000000	`
1	0x01	001	00000001	SOH	33	0x21	041	01000001	!	65	0x41	101	10000001	A	97	0x61	141	11000001	a
2	0x02	002	00000010	STX	34	0x22	042	01000010	"	66	0x42	102	10000010	B	98	0x62	142	11000010	b
3	0x03	003	00000011	ETX	35	0x23	043	01000011	#	67	0x43	103	10000011	C	99	0x63	143	11000011	c
4	0x04	004	00000100	EOT	36	0x24	044	01000100	\$	68	0x44	104	10000100	D	100	0x64	144	11000100	d
5	0x05	005	00000101	ENQ	37	0x25	045	01000101	%	69	0x45	105	10000101	E	101	0x65	145	11000101	e
6	0x06	006	00000110	ACK	38	0x26	046	01000110	&	70	0x46	106	10000110	F	102	0x66	146	11000110	f
7	0x07	007	00000111	BEL	39	0x27	047	01000111	'	71	0x47	107	10000111	G	103	0x67	147	11000111	g
8	0x08	010	00010000	BS	40	0x28	050	01010000	(72	0x48	110	10010000	H	104	0x68	150	11010000	h
9	0x09	011	00010001	TAB	41	0x29	051	01010001)	73	0x49	111	10010001	I	105	0x69	151	11010001	i
10	0x0A	012	00010010	LF	42	0x2A	052	01010010	*	74	0x4A	112	10010010	J	106	0x6A	152	11010010	j
11	0x0B	013	00010011	VT	43	0x2B	053	01010011	+	75	0x4B	113	10010011	K	107	0x6B	153	11010011	k
12	0x0C	014	00010100	FF	44	0x2C	054	01010100	,	76	0x4C	114	10010100	L	108	0x6C	154	11010100	l
13	0x0D	015	00010101	CR	45	0x2D	055	01010101	-	77	0x4D	115	10010101	M	109	0x6D	155	11010101	m
14	0x0E	016	00010110	SO	46	0x2E	056	01010110	.	78	0x4E	116	10010110	N	110	0x6E	156	11010110	n
15	0x0F	017	00010111	SI	47	0x2F	057	01010111	/	79	0x4F	117	10010111	O	111	0x6F	157	11010111	o
16	0x10	020	00100000	DLE	48	0x30	060	01100000	0	80	0x50	120	10100000	P	112	0x70	160	11100000	p
17	0x11	021	00100001	DC1	49	0x31	061	01100001	1	81	0x51	121	10100001	Q	113	0x71	161	11100001	q
18	0x12	022	00100010	DC2	50	0x32	062	01100010	2	82	0x52	122	10100010	R	114	0x72	162	11100010	r
19	0x13	023	00100011	DC3	51	0x33	063	01100011	3	83	0x53	123	10100011	S	115	0x73	163	11100011	s
20	0x14	024	00100100	DC4	52	0x34	064	01100100	4	84	0x54	124	10100100	T	116	0x74	164	11100100	t
21	0x15	025	00100101	NAK	53	0x35	065	01100101	5	85	0x55	125	10100101	U	117	0x75	165	11100101	u
22	0x16	026	00100110	SYN	54	0x36	066	01100110	6	86	0x56	126	10100110	V	118	0x76	166	11100110	v
23	0x17	027	00100111	ETB	55	0x37	067	01100111	7	87	0x57	127	10100111	W	119	0x77	167	11100111	w
24	0x18	030	00110000	CAN	56	0x38	070	01110000	8	88	0x58	130	10110000	X	120	0x78	170	11110000	x
25	0x19	031	00110001	EM	57	0x39	071	01110001	9	89	0x59	131	10110001	Y	121	0x79	171	11110001	y
26	0x1A	032	00110010	SUB	58	0x3A	072	01110010	:	90	0x5A	132	10110010	Z	122	0x7A	172	11110010	z
27	0x1B	033	00110011	ESC	59	0x3B	073	01110011	;	91	0x5B	133	10110011	[123	0x7B	173	11110011	{
28	0x1C	034	00110100	FS	60	0x3C	074	01110100	<	92	0x5C	134	10110100	\	124	0x7C	174	11110100	
29	0x1D	035	00110101	GS	61	0x3D	075	01110101	=	93	0x5D	135	10110101]	125	0x7D	175	11110101	}
30	0x1E	036	00110110	RS	62	0x3E	076	01110110	>	94	0x5E	136	10110110	^	126	0x7E	176	11110110	~
31	0x1F	037	00110111	US	63	0x3F	077	01110111	?	95	0x5F	137	10110111	_	127	0x7F	177	11110111	DEL



Unicode 9.0 Character Code Charts

[SCRIPTS](#) | [SYMBOLS](#) | [NOTES](#)<http://unicode.org/charts/>

Find chart by hex code:

Go

Related links: [Name index](#) [Help & links](#)

Scripts

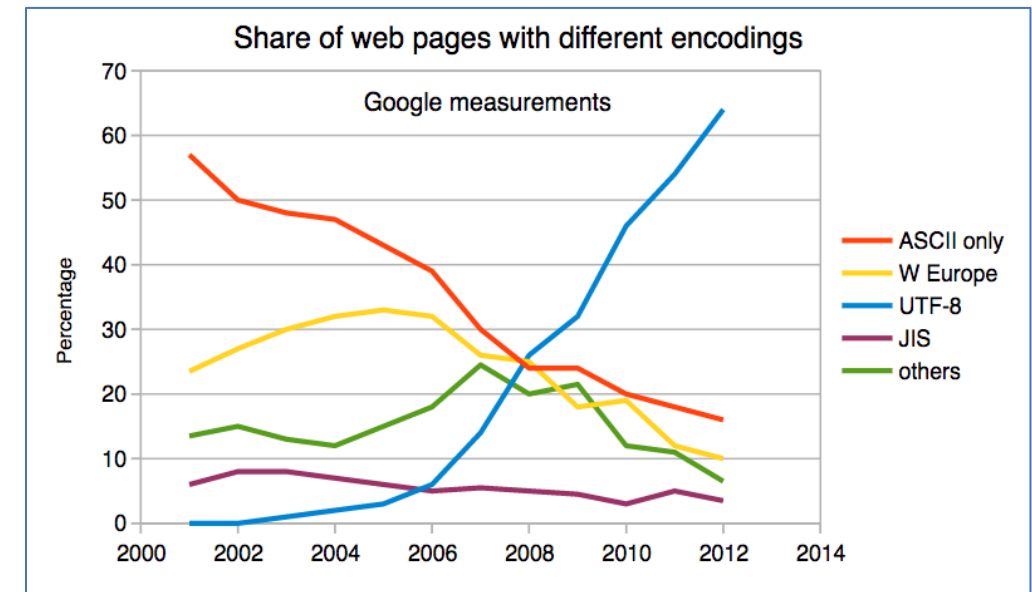
European Scripts	African Scripts	South Asian Scripts	Indonesia & Oceania Scripts
Armenian	Adlam	Ahom	Balinese
Armenian Ligatures	Bamum	Bengali and Assamese	Batak
Caucasian Albanian	Bamum Supplement	Bhaiksuki	Buginese
Cypriot Syllabary	Bassa Vah	Brahmi	Buhid
Cyrillic	Coptic	Chakma	Hanunoo
Cyrillic Supplement	Coptic in Greek block	Devanagari	Javanese
Cyrillic Extended-A	Coptic Epact Numbers	Devanagari Extended	Rejang
Cyrillic Extended-B	Egyptian Hieroglyphs (1MB)	Grantha	Sundanese
Cyrillic Extended-C	Ethiopic	Gujarati	Sundanese Supplement
Elbasan	Ethiopic Supplement	Gurmukhi	Tagalog
Georgian	Ethiopic Extended	Kaithi	Tagbanwa
Georgian Supplement	Ethiopic Extended-A	Kannada	East Asian Scripts
Glagolitic	Mende Kikakui	Kharoshthi	Bopomofo
Glagolitic Supplement	Meroitic	Khojki	Bopomofo Extended
Gothic	Meroitic Cursive	Khudawadi	CJK Unified Ideographs (Han) (35MB)
Greek	Meroitic Hieroglyphs	Lepcha	CJK Extension-A (6MB)
Greek Extended	N'Ko	Limbu	CJK Extension B (40MB)
Ancient Greek Numbers	Osmanya	Mahajani	CJK Extension C (3MB)
Latin	Tifinagh	Malayalam	CJK Extension D
Basic Latin (ASCII)	Vai	Meetei Mayek	CJK Extension E (3.5MB)
Latin-1 Supplement	Middle Eastern Scripts	Meetei Mayek Extensions	(see also UniHan Database)
Latin Extended-A	Anatolian Hieroglyphs	Modi	CJK Compatibility Ideographs

Multi-Byte Characters (a quick aside...)

To represent the wide range of characters computers must handle we represent characters with more than one byte

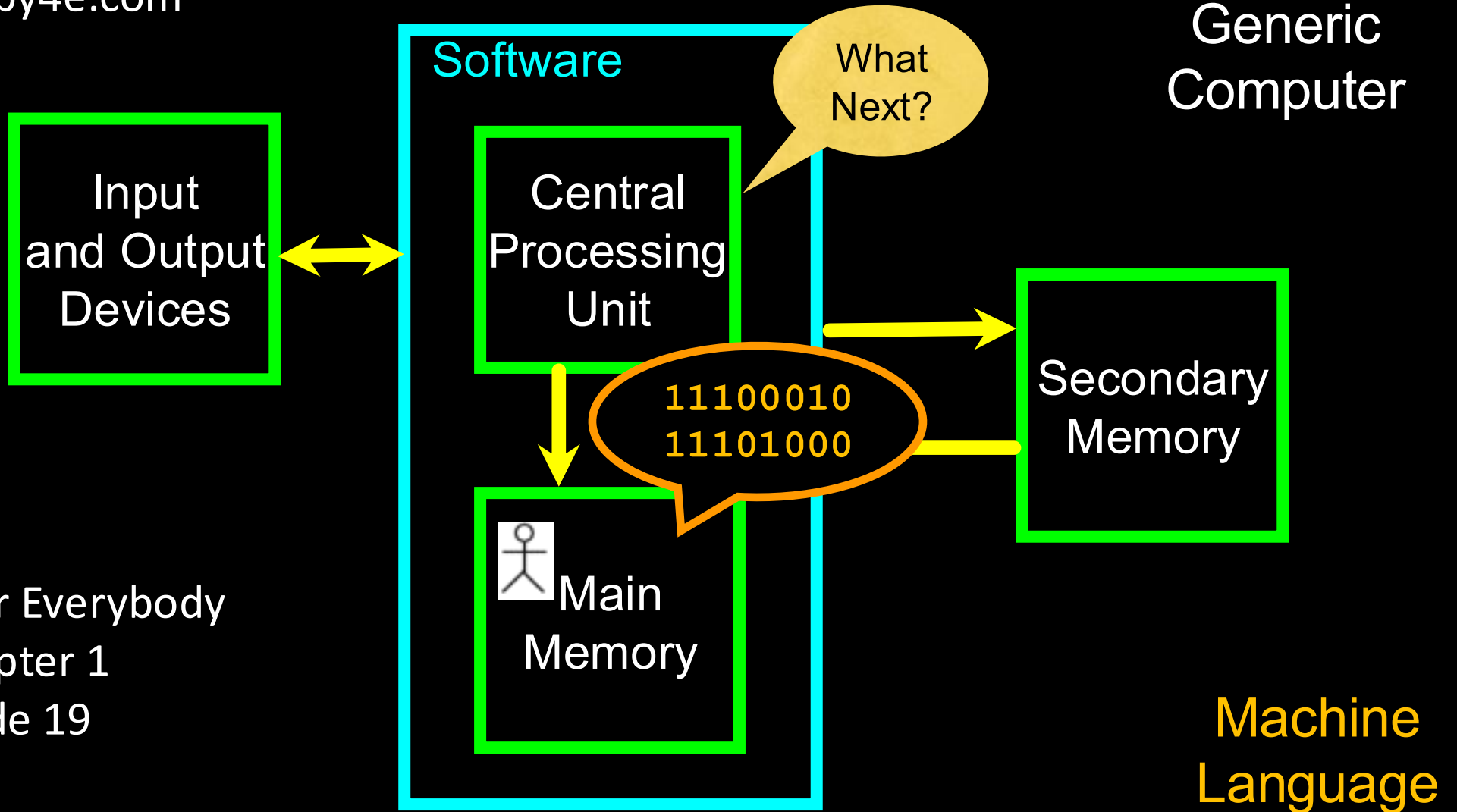
<https://en.wikipedia.org/wiki/UTF-8>

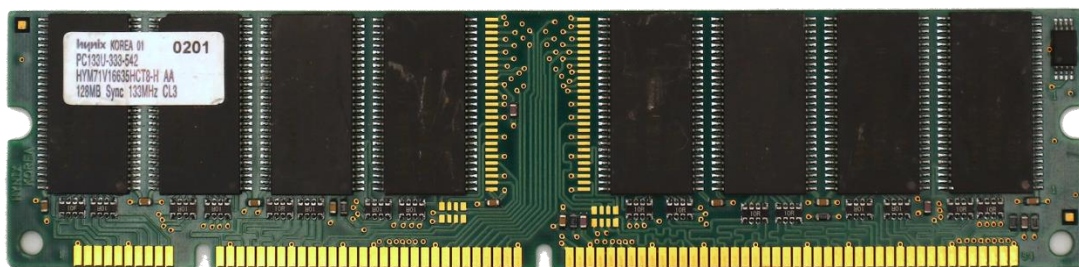
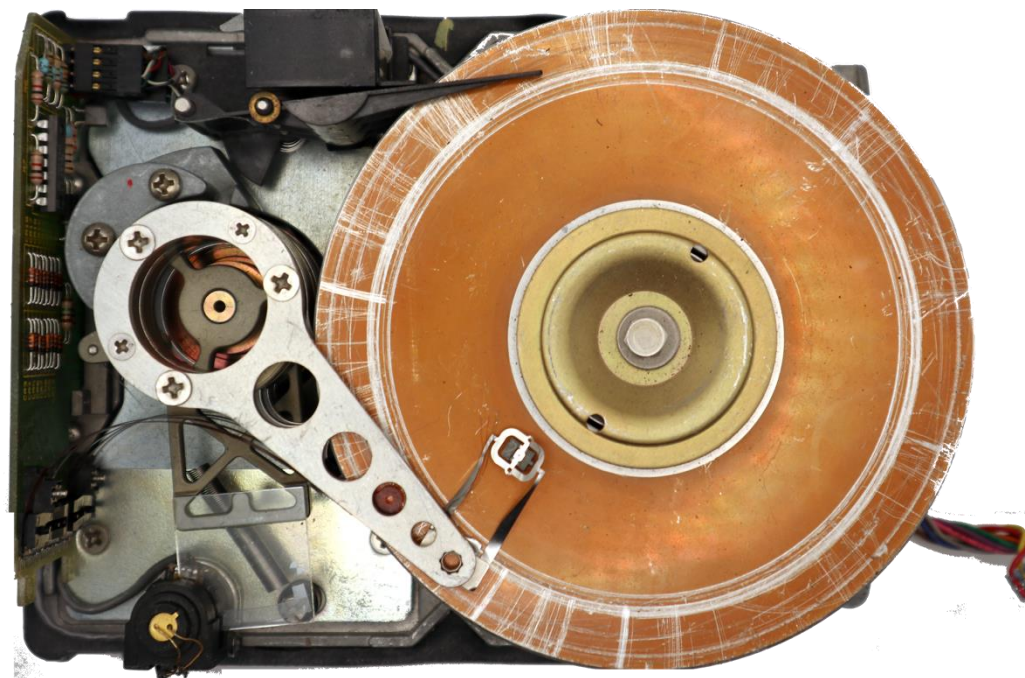
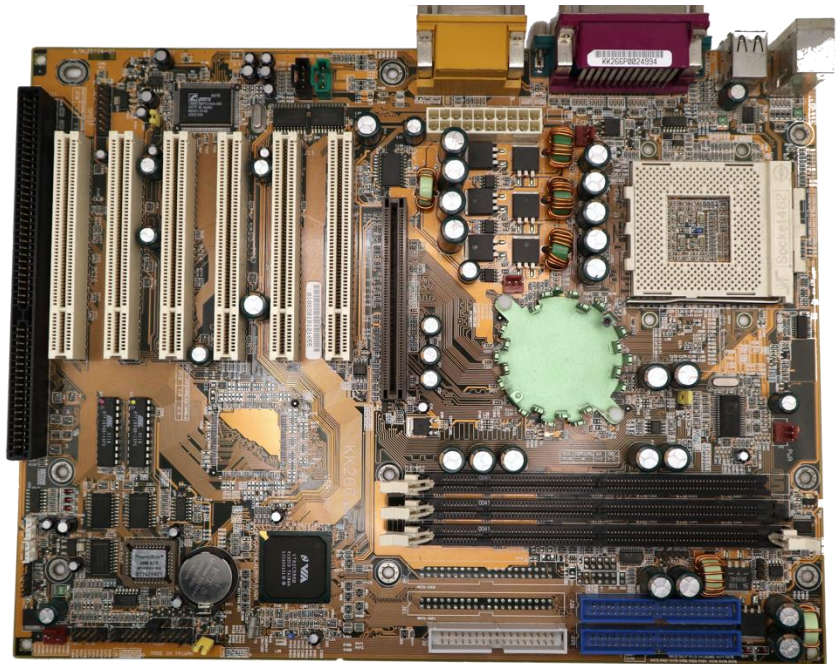
- UTF-32 – Fixed Length - Four Bytes
- UTF-16 – Fixed length - Two bytes
- UTF-8 – 1-4 bytes
 - Upwards compatible with ASCII
 - Automatic detection between ASCII and UTF-8
 - UTF-8 is recommended practice for encoding data to be exchanged between systems



What is Programming?

Hint: In the end any program we can write can also be represented in machine language. Machine code will both be simpler and a lot more complex. There is a reason we built languages like C and Python.

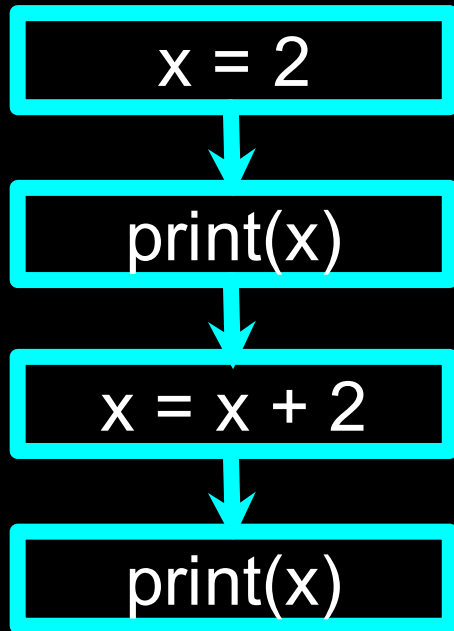




Program Steps or Program Flow

- Like a recipe or installation instructions, a program is a **sequence** of steps to be done in order.
- Some steps are **conditional** - they may be skipped.
- Sometimes a step or group of steps is to be **repeated**.
- Sometimes we store a set of steps to be used over and over as needed several places throughout the program

Sequential Steps



Program:

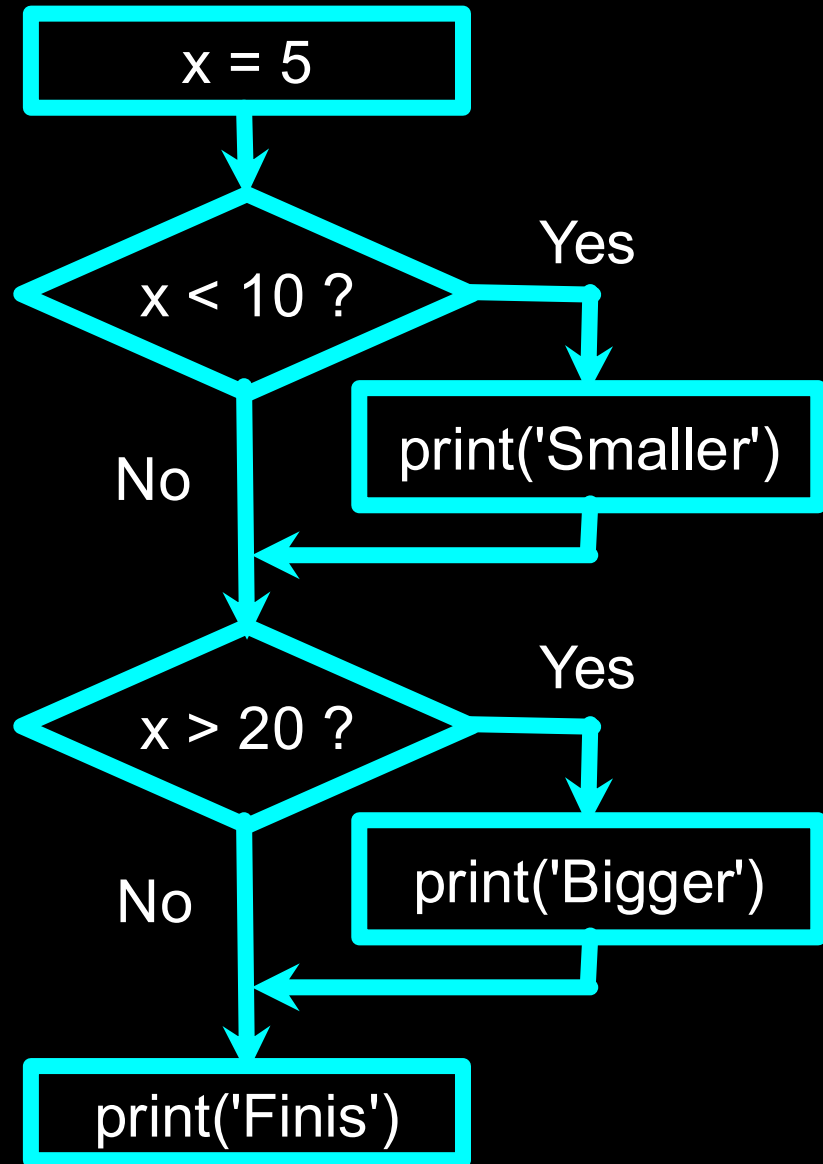
```
x = 2  
print(x)  
x = x + 2  
print(x)
```

Output:

2
4

When a program is running, it flows from one step to the next. As programmers, we set up “paths” for the program to follow.

Conditional Steps



Program:

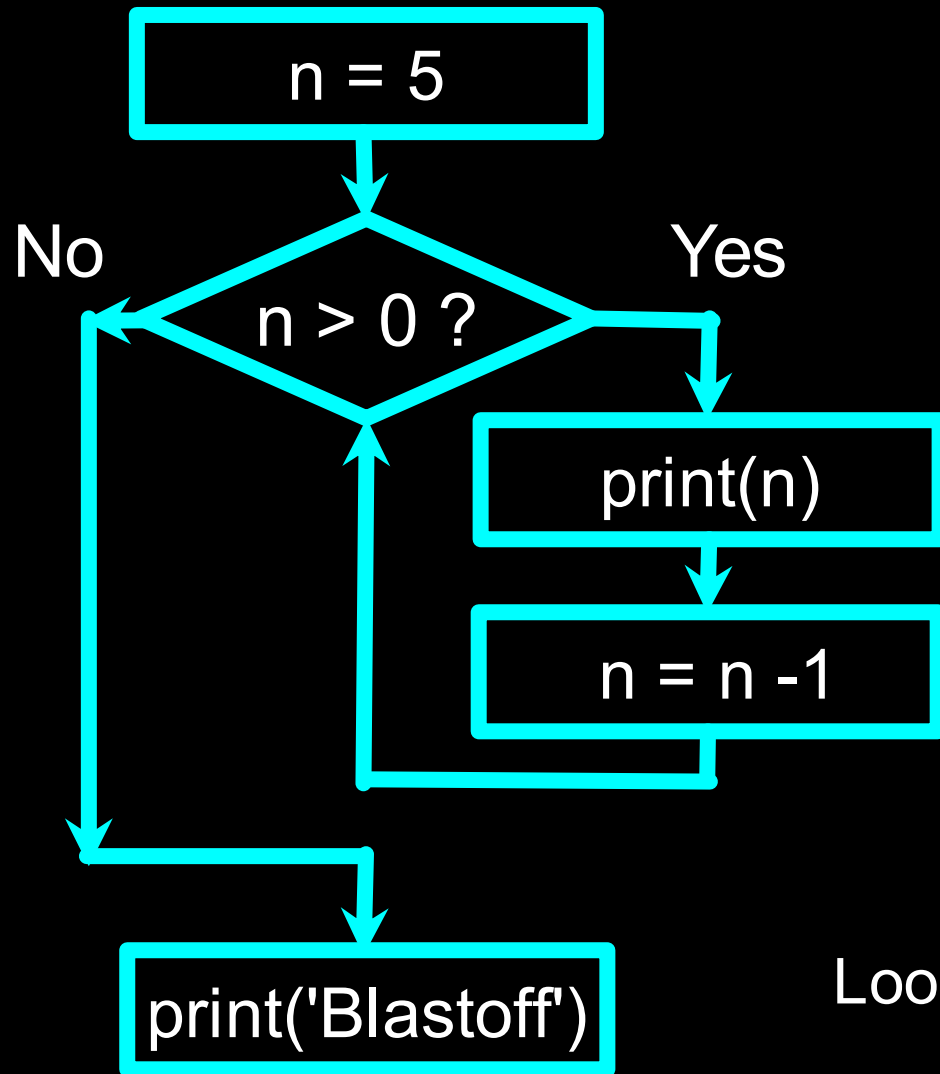
```
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')

print('Finis')
```

Output:

Smaller
Finis

Repeated Steps



Program:

```
n = 5
while n > 0 :
    print(n)
    n = n - 1
print('Blastoff!')
```

Output:

5
4
3
2
1
Blastoff!

Loops (repeated steps) have **iteration variables** that change each time through a loop.

Variables

- A **variable** is a named place in the memory where a programmer can store data and later retrieve the data using the **variable** “name”
- Programmers get to choose the names of the **variables**
- You can change the contents of a **variable** in a later statement

x = 12.2

y = 14

x = 100

x

~~12.2~~ 100

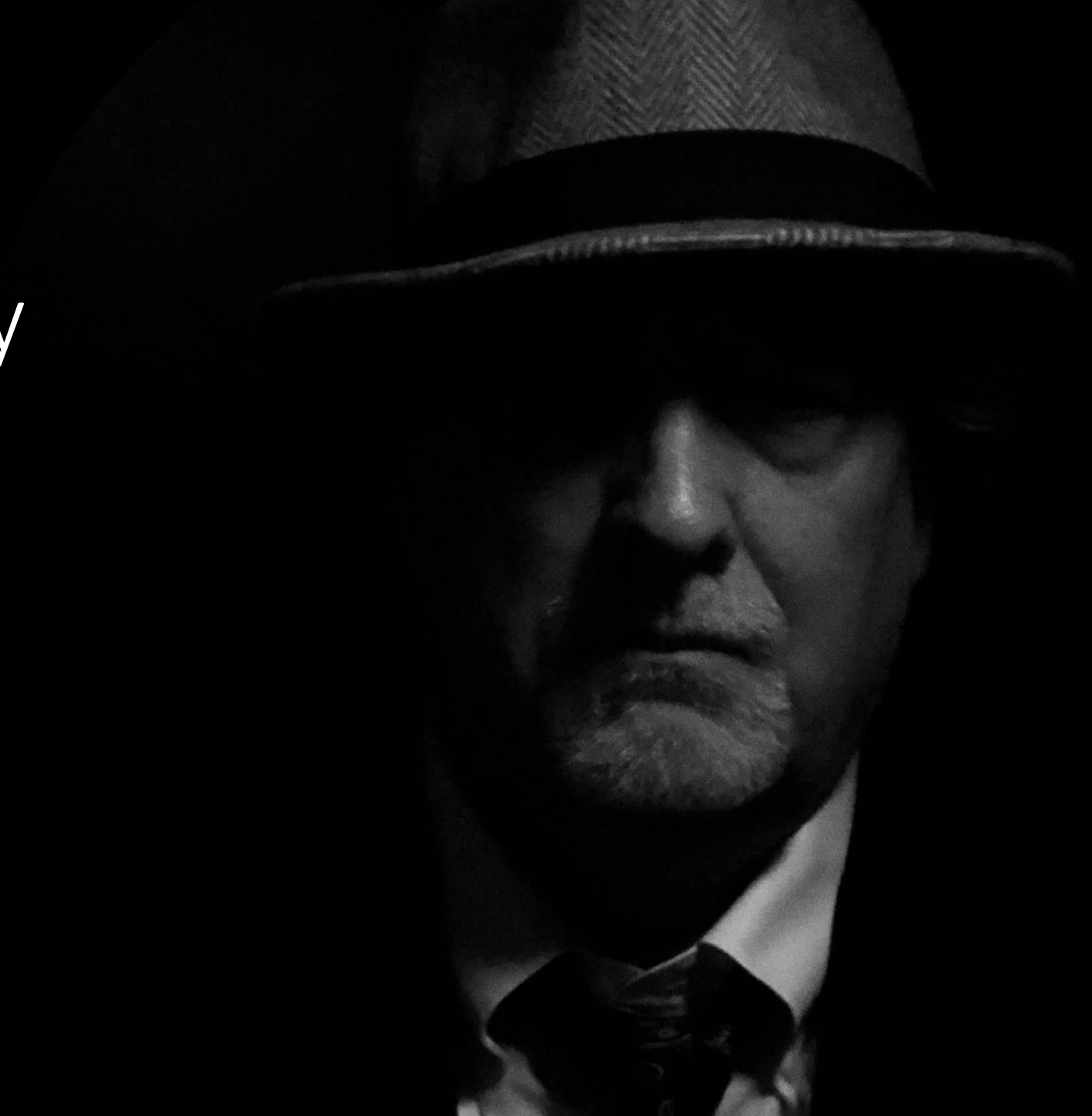
y

14

The True Story of Python Assignment Statements

Charles Severance

www.dj4e.com



What **IS** a Python variable?

- A Python variable is a symbolic name that is a **pointer** or **reference** to an object.
- We can examine what a variable is "pointing to" using the `id()` function in Python

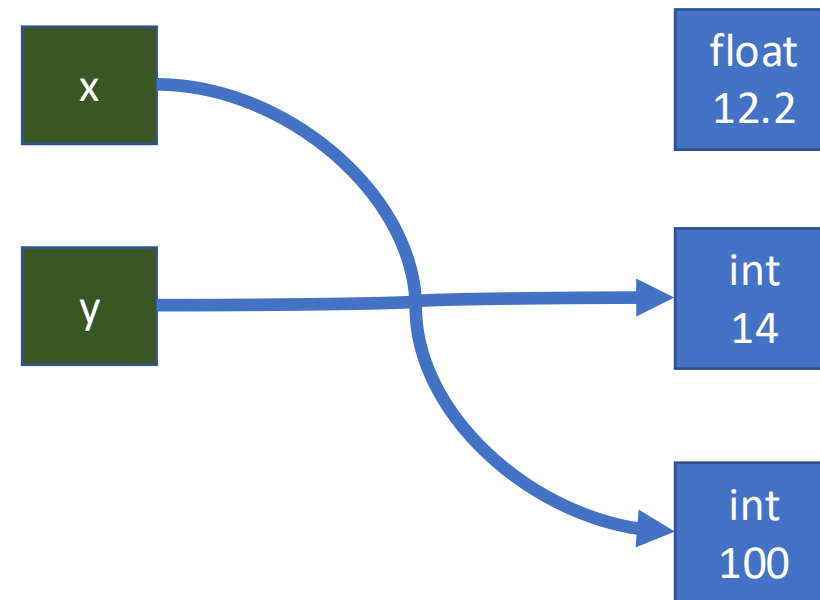
```
>>> x = 12.2
>>> y = 14
>>> print(id(x))
4334363728
>>> print(id(y))
4345045600
```



What **IS** a Python variable?

- A Python variable is a symbolic name that is a **pointer** or **reference** to an object.
- We can examine what a variable is "pointing to" using the `id()` function in Python

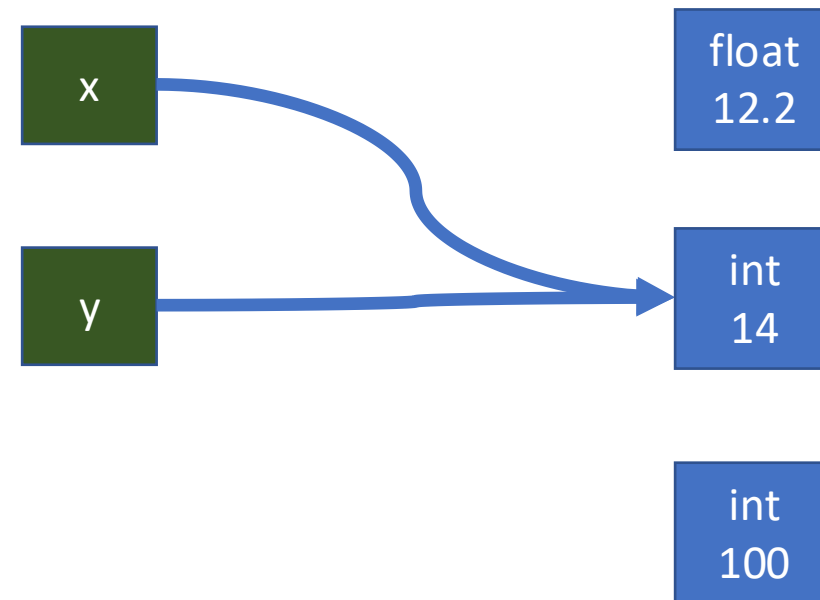
```
>>> x = 12.2
>>> y = 14
>>> print(id(x))
4334363728
>>> print(id(y))
4345045600
>>> x = 100
>>> print(id(x))
4345048352
```



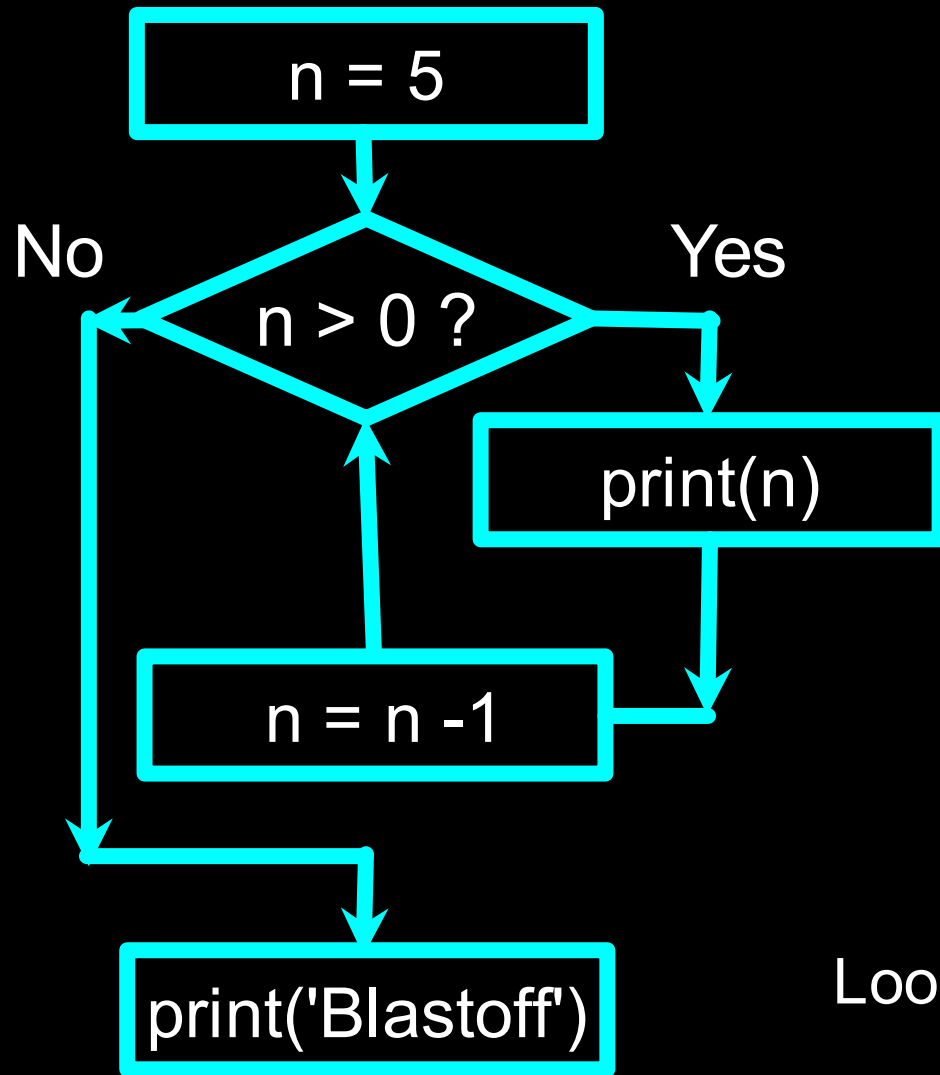
What **IS** a Python variable?

- A Python variable is a symbolic name that is a **pointer** or **reference** to an object.
- We can examine what a variable is "pointing to" using the `id()` function in Python

```
>>> x = 12.2
>>> y = 14
>>> print(id(x))
4334363728
>>> print(id(y))
4345045600
>>> x = 100
>>> print(id(x))
4345048352
>>> x = 14
>>> print(id(x))
4345045600
```



Loops in C



Program:

```
for (n=5; n>0; n--) {  
    printf("%d\n", n);  
    printf('Blastoff!\n');  
}
```

Output:

5
4
3
2
1
Blastoff!

Loops (repeated steps) have **iteration variables** that change each time through a loop.

The CDC6504 Architecture

A blend of two names to avoid confusion with the actual MOS6502. And they both have registers named "X"

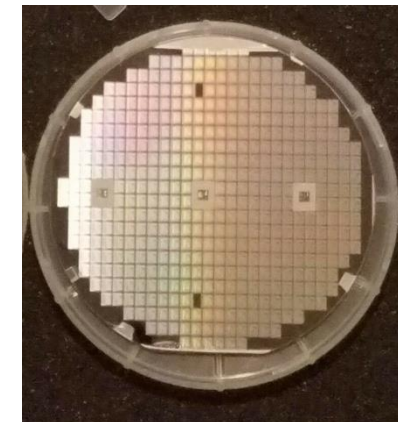
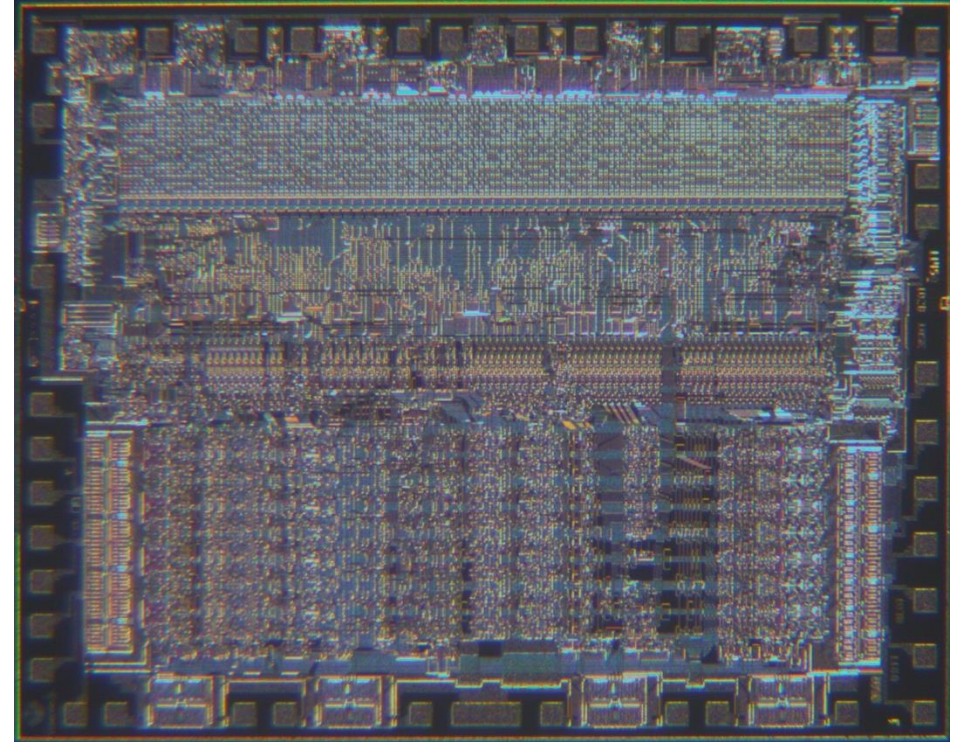
AI: Please compare the Z80 and 6502 in terms of historical impact

MOS Technology 6502

- Launched in 1975 and cost \$25
- 4,528 NMOS Transistors
- 1Mhz

Apple I, Apple II, Commodore PET, Commodore VIC-20, Commodore 64 (6510), Atari 2600, Atari 400, Atari 800, Atari XL series, Atari XE series, BBC Micro, Acorn Atom, Acorn Electron, Nintendo NES (2A03), Oric-1, Oric Atmos, Philips P2000, Ohio Scientific Challenger 1P, KIM-1 ...

https://en.wikipedia.org/wiki/MOS_Technology_6502



Homage to the the CDC 6500

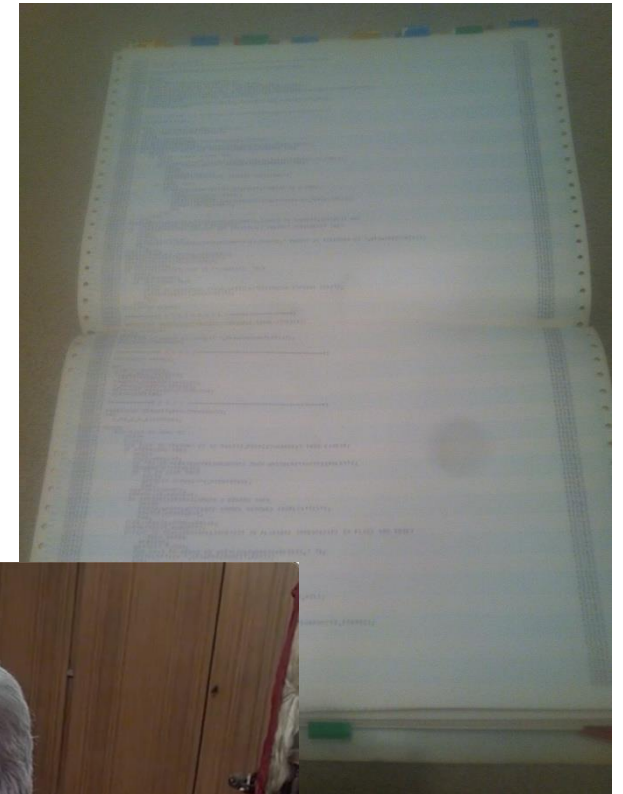
- Control Data Corporation
- Also in the mid-1970's
- Dr. Chuck's first computer
 - FORTRAN
 - Assembly Language



https://en.wikipedia.org/wiki/CDC_6000_series#Central_processor

Homage to the the CDC 6500

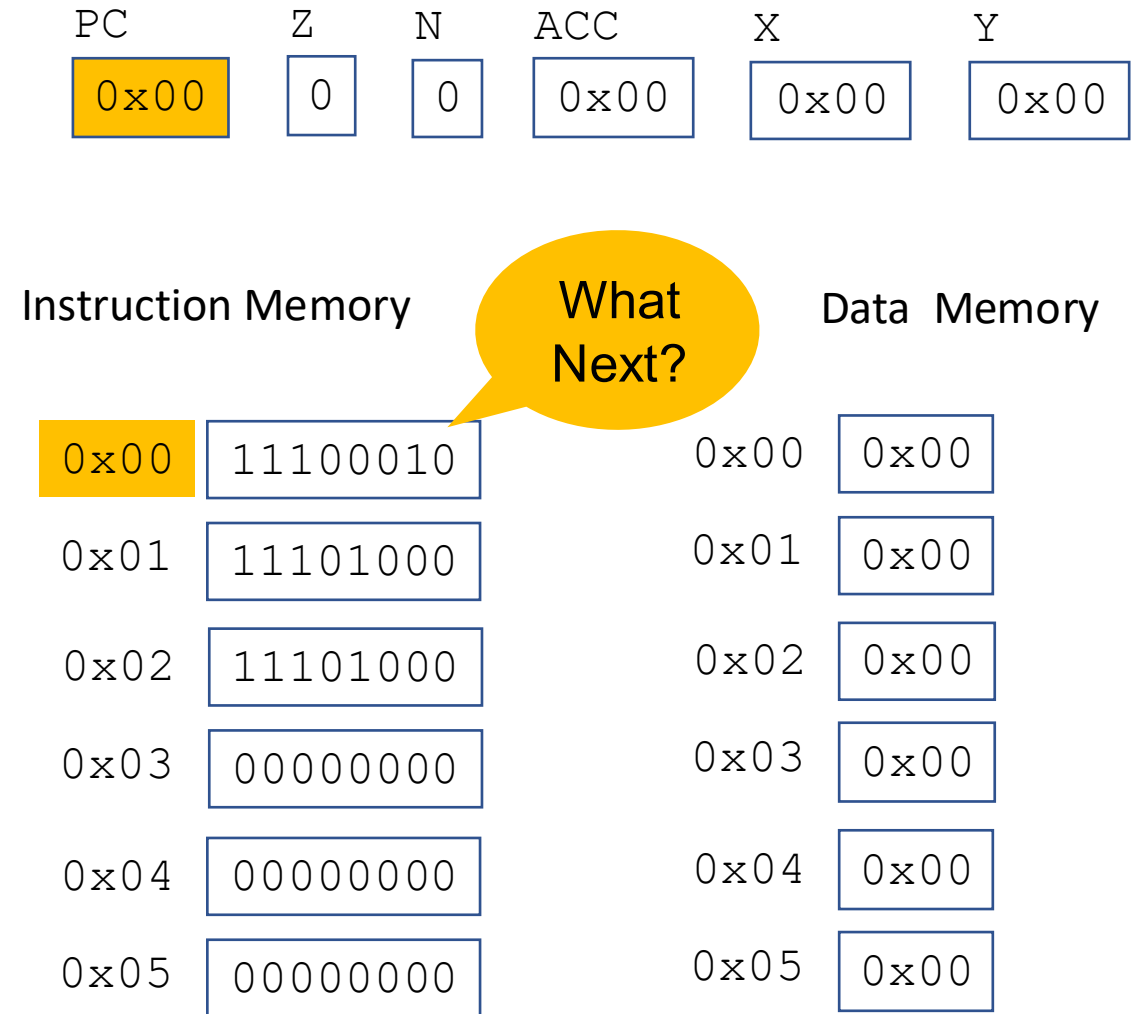
- Control Data Corporation
- Also in the mid-1970's
- Dr. Chuck's first computer
 - FORTRAN
 - Assembly Language
 - Pascal



https://en.wikipedia.org/wiki/CDC_6000_series#Central_processor

CDC6504 Architecture

- PC – Program Counter
- ACC – Accumulator register
- Z – Was the last result zero?
- N – Was the last result negative?
- X and Y – Index registers
- Instruction memory shown in base-2 for easy decoding
- Data memory shown in hex – will usually contain ASCII



Programming the CDC6504

- Assembly language is easier to read and translates directly to binary machine language
- Assembly language also has labels, so we don't need to hand-compute the hex value of data or instruction address locations

Assembly	Opcode	Description
CLX	11100010	Clear X register (X = 0).
INX	11101000	Increment X register by 1
LDX #value	10100010	Load X register with immediate (constant) value
LDX \$address	10100110	Load X register from zero-page memory address
STX \$address	10000110	Store X register to zero-page memory address

Our first Program

PC	Z	N	ACC	X	Y
0x00	0	0	0x00	0x00	0x00

0x00	11100010	CLX	; Clear X register	0x00	0x00
0x01	11101000	INX	; Increment X	0x01	0x00
0x02	11101000	INX	; Increment X again	0x02	0x00
0x03	00000000	BRK		0x03	0x00
0x04	00000000			0x04	0x00
0x05	00000000			0x05	0x00
0x06	00000000			0x06	0x00

Our first Program

PC	Z	N	ACC	X	Y
0x00	0	0	0x00	0x00	0x00

0x00	11100010	CLX ; Clear X register	0x00	0x00
0x01	11101000	INX ; Increment X	0x01	0x00
0x02	11101000	INX ; Increment X again	0x02	0x00
0x03	00000000	BRK	0x03	0x00
0x04	00000000		0x04	0x00
0x05	00000000		0x05	0x00
0x06	00000000		0x06	0x00

Our first Program

PC	Z	N	ACC	X	Y
0x01	0	0	0x00	0x01	0x00

0x00	11100010	CLX	; Clear X register	0x00	0x00
0x01	11101000	INX	; Increment X	0x01	0x00
0x02	11101000	INX	; Increment X again	0x02	0x00
0x03	00000000	BRK		0x03	0x00
0x04	00000000			0x04	0x00
0x05	00000000			0x05	0x00
0x06	00000000			0x06	0x00

Our first Program

PC	Z	N	ACC	X	Y
0x02	0	0	0x00	0x02	0x00

0x00	11100010	CLX	; Clear X register	0x00	0x00
0x01	11101000	INX	; Increment X	0x01	0x00
0x02	11101000	INX	; Increment X again	0x02	0x00
0x03	00000000	BRK		0x03	0x00
0x04	00000000			0x04	0x00
0x05	00000000			0x05	0x00
0x06	00000000			0x06	0x00

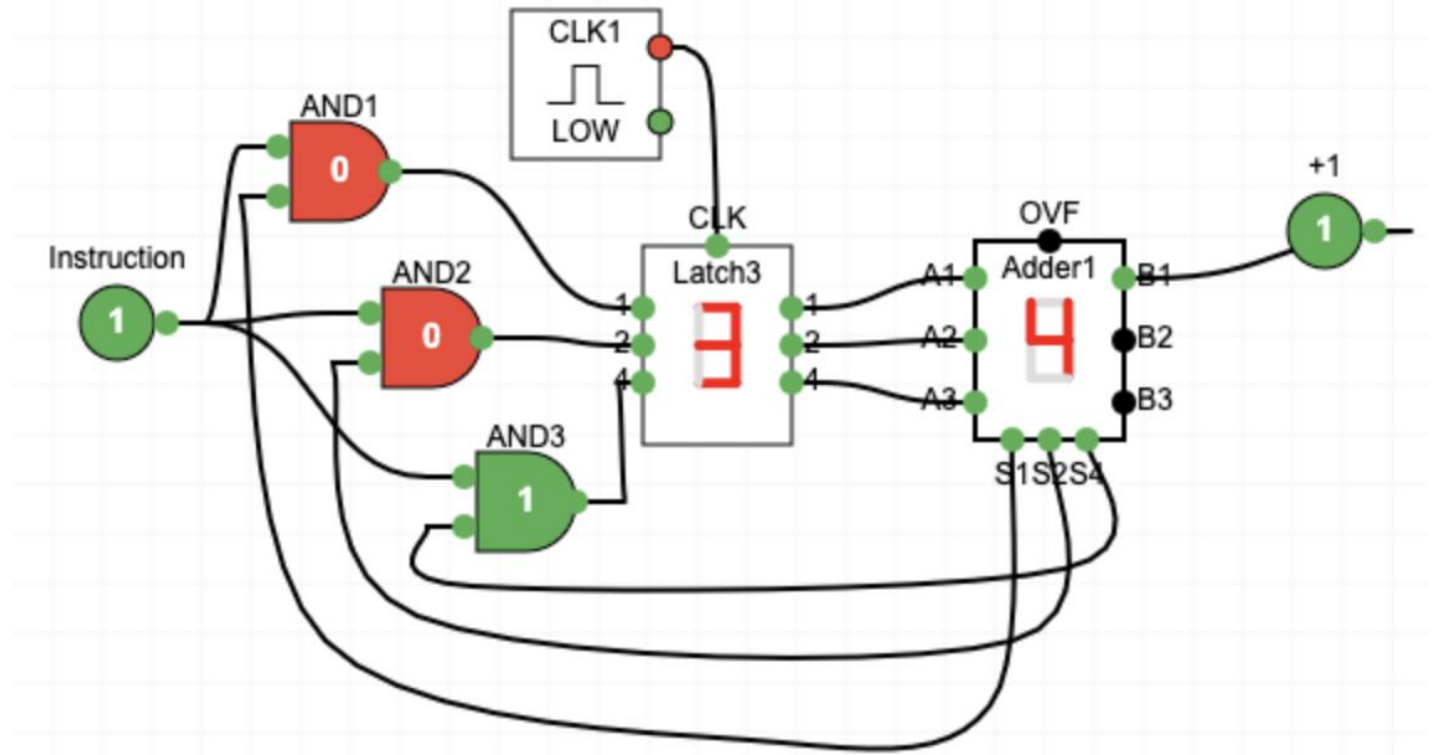
Our first Program

PC	Z	N	ACC	X	Y
0x03	0	0	0x00	0x02	0x00

0x00	11100010	CLX	; Clear X register	0x00	0x00
0x01	11101000	INX	; Increment X	0x01	0x00
0x02	11101000	INX	; Increment X again	0x02	0x00
0x03	00000000	BRK		0x03	0x00
0x04	00000000			0x04	0x00
0x05	00000000			0x05	0x00
0x06	00000000			0x06	0x00

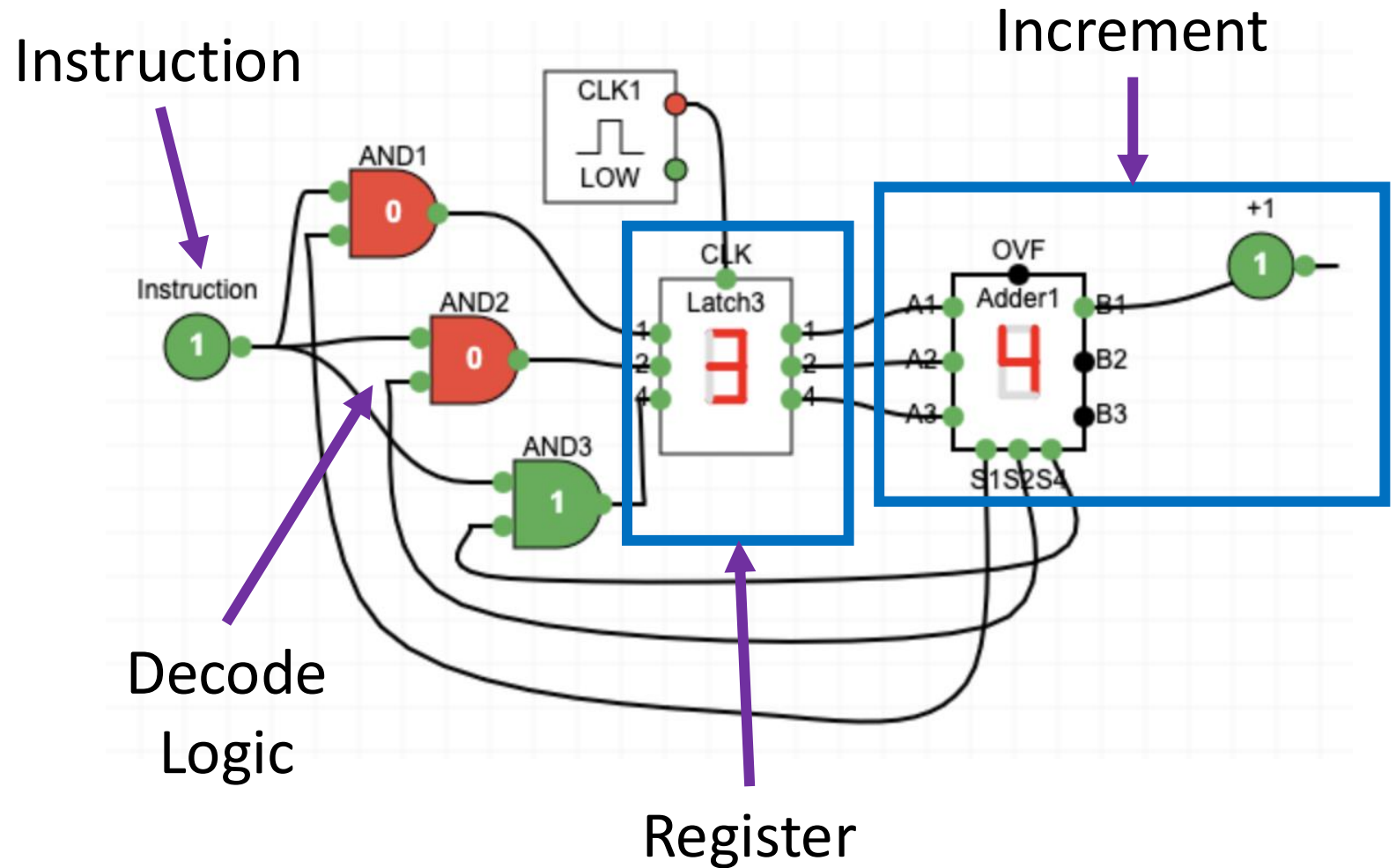
Our CPU from Logic, Gates, Transistors

- There is a lot of overlap between the CDC6504 and our two instruction CPU....



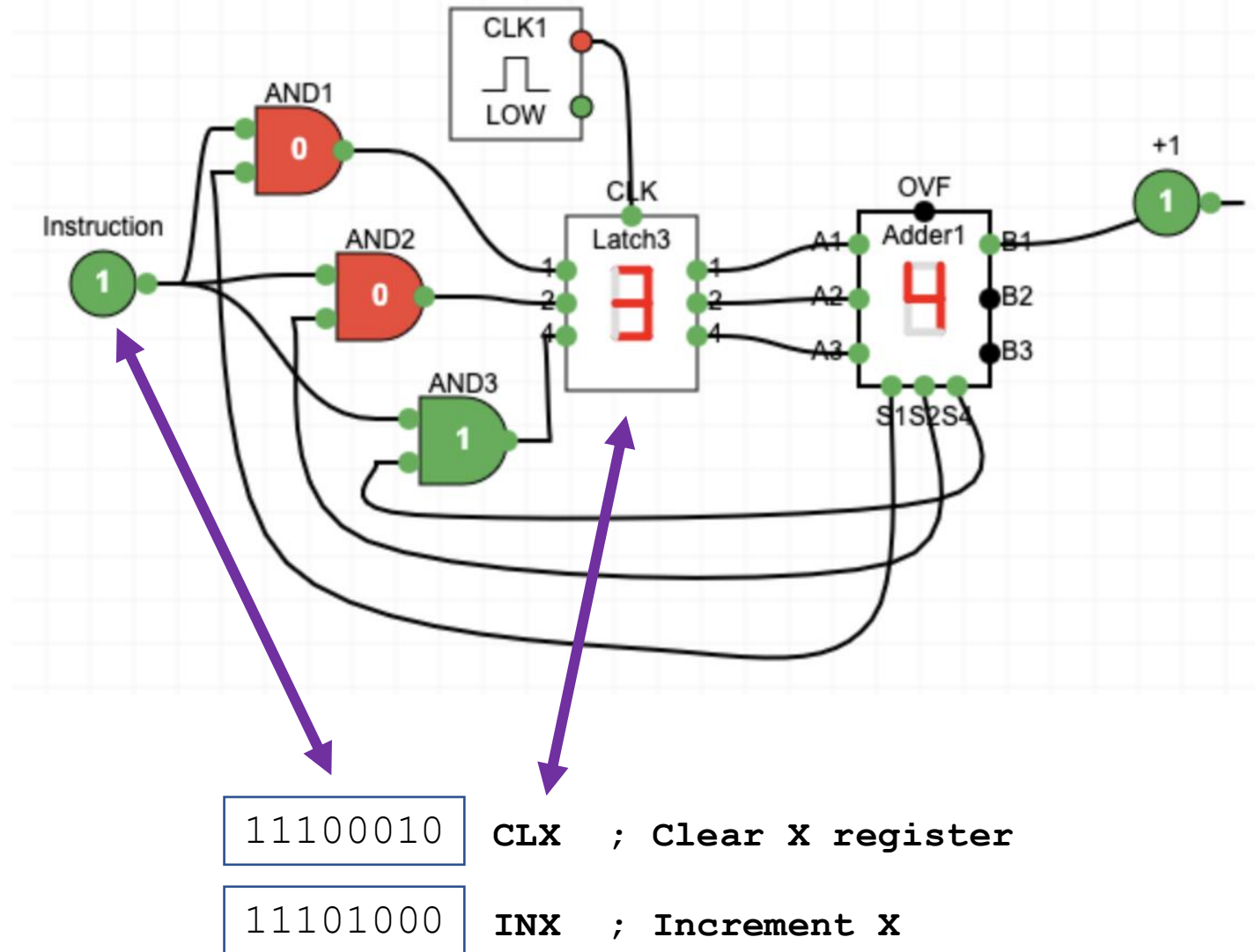
Our CPU from Logic, Gates, Transistors

- Our CPU implements two instructions
 - Zero Register
 - Increment Register
- There is no program counter because there is no instruction memory
- But....



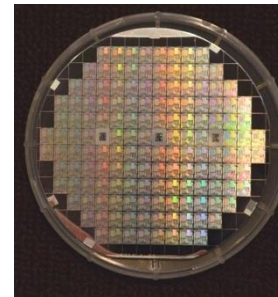
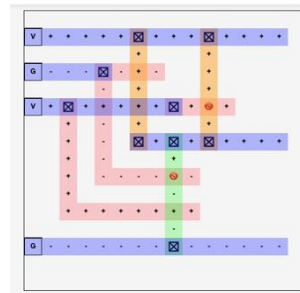
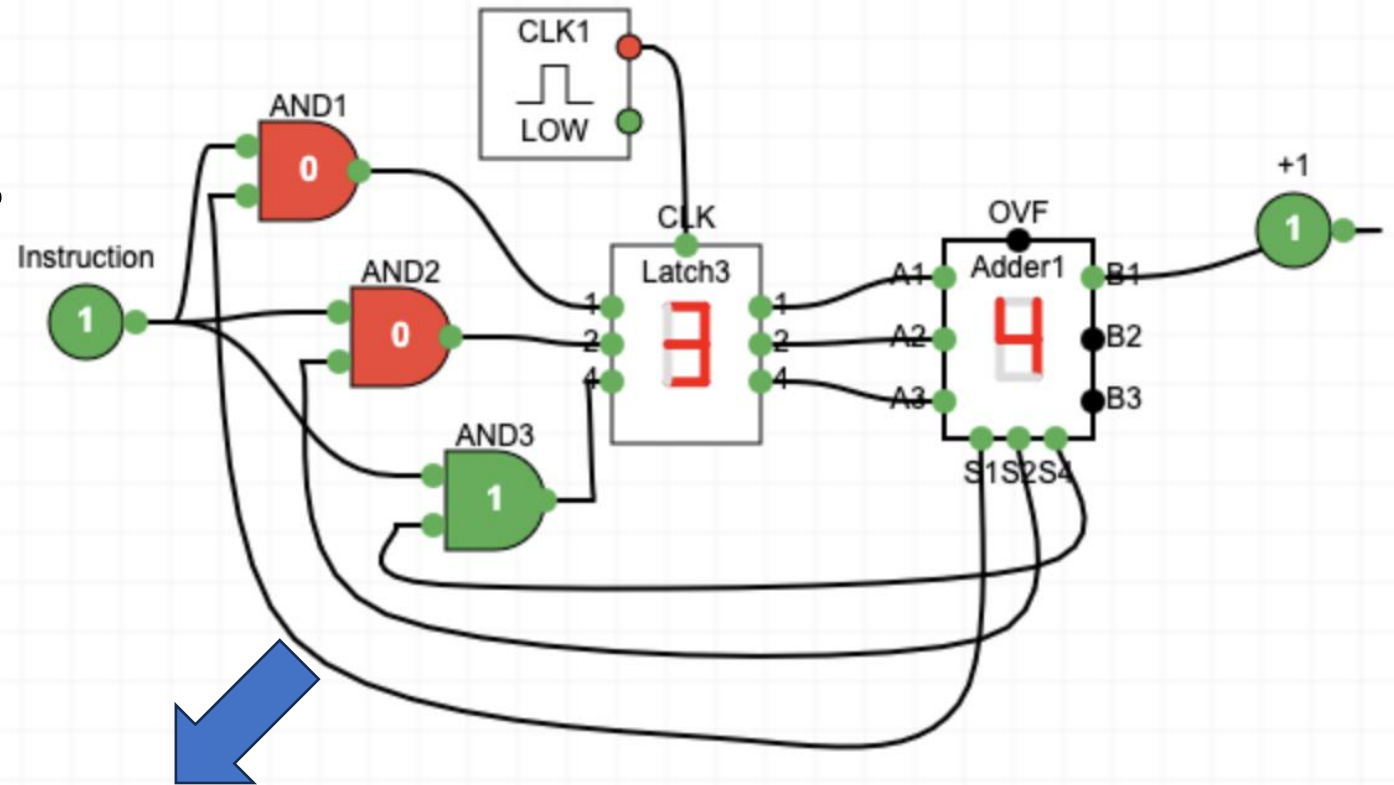
Looking closer

- These two instructions are the equivalent to the zero (clear the latch) and one (increment the latch) from our hand-built CPU
- The 6502 has 8-bit registers



If you had time...

- With what you know and and better design tools you *could* design a CDC6504
 - Logic
 - Memory
 - VLSI layout
 - Manufacture

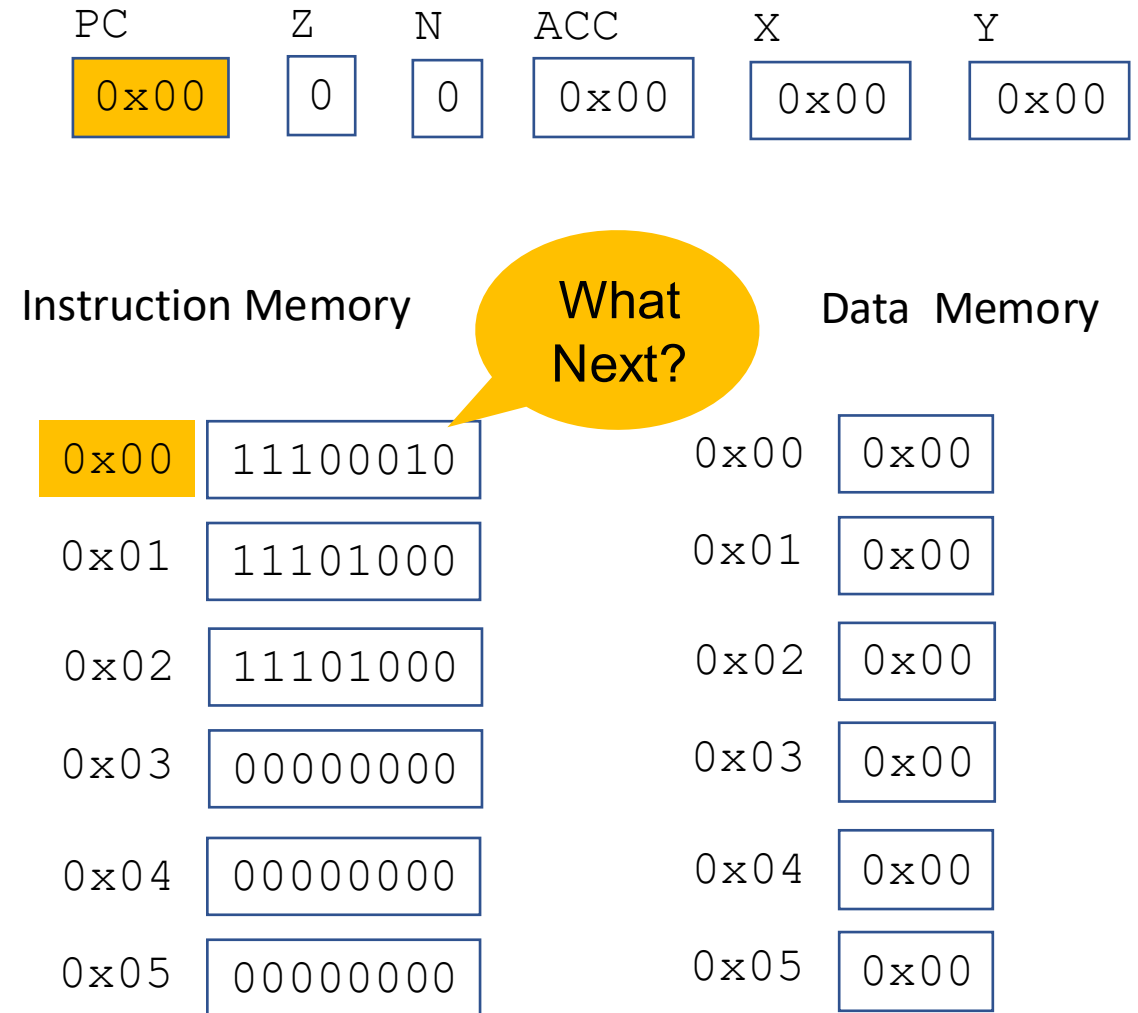


CDC6504 Machine Language

<https://www.ca4e.com/cdc6504/documentation.html>

CDC6504 Architecture

- PC – Program Counter
- ACC – Accumulator register
- Z – Was the last result zero?
- N – Was the last result negative?
- X and Y – Index registers
- Instruction memory shown in base-2 for easy decoding
- Data memory shown in hex – will usually contain ASCII



Index Instructions (8-bit)

- The X and Y registers are often used to control counted loops or to be an offset within an array.

Assembly	Opcode	Description
CLX	11100010	Clear X register ($X = 0$). Sets Z flag to 1, N flag to 0.
CLY	11000010	Clear Y register ($Y = 0$). Sets Z flag to 1, N flag to 0s.
INX	11101000	Increment X register by 1
INY	11001000	Increment Y register by 1
DEX	11001010	Decrement X register by 1
DEY	10001000	Decrement Y register by 1

16-Bit Instructions

- The byte immediately following the instruction can be a constant value or data memory address

Assembly	Opcode	Description
LDA #value	10101001	Load accumulator with immediate value (0-255, 0x00-0xFF, or 'A')
LDA \$address	10100101	Load accumulator using immediate value as memory address

0x00	10101001	LDA #'*'	0x02	10100101	LDA \$02
0x01	00101010	(the 42)	0x03	00000010	(the address 2)

Load / Store Instructions

Assembly	Opcode	Description
LDA #value	10101001	Load accumulator with immediate value (ACC = value)
LDA \$address	10100101	Load accumulator from zero-page memory address (ACC = memory[address])
LDX #value	10100010	Load X register with immediate value (X = value)
LDX \$address	10100110	Load X register from zero-page memory address (X = memory[address])
LDY #value	10100000	Load Y register with immediate value (Y = value)
STA \$address	10000101	Store accumulator to zero-page memory address (memory[address] = ACC)
STX \$address	10000110	Store X register to zero-page memory address (memory[address] = X)
STY \$address	10000100	Store Y register to zero-page memory address (memory[address] = Y)

Indexed Load / Store Instructions

- To handle array indexed operations, we can use X and Y as an offset starting at an immediate address

Assembly	Opcode	Description
LDA \$address,X	10110101	Load accumulator from indexed address ($ACC = \text{memory}[\text{address} + X]$)
LDA \$address,Y	10111001	Load accumulator from indexed address ($ACC = \text{memory}[\text{address} + Y]$)
STA \$address,X	10010101	Store accumulator to indexed address ($\text{memory}[\text{address} + X] = ACC$)
STA \$address,Y	10011001	Store accumulator to indexed address ($\text{memory}[\text{address} + Y] = ACC$)

Arithmetic Instructions

- Adding and subtraction and getting back a result is ACC only

Assembly	Opcode	Description
ADC #value	01101001	Add (immediate): $ACC = ACC + value$
ADC \$address	01100101	Add (from memory): $ACC = ACC + memory[address]$
SBC #value	11101001	Subtract (immediate): $ACC = ACC - value$
SBC \$address	11100101	Subtract (from memory): $ACC = ACC - memory[address]$

Comparison Instructions

- Comparison is like doing a subtraction, setting Z and N and then discarding the result of the subtraction.

Assembly	Opcode	Description
CMP #value	11001001	Compare accumulator with immediate value, sets Z, N flags
CMP \$address	11000101	Compare accumulator with memory[address], sets Z, N flags
CPX #value	11100000	Compare X register with immediate value, sets Z, N flags
CPX \$address	11100100	Compare X register with memory[address], sets Z, N flags
CPY #value	11000000	Compare Y register with immediate value, sets Z, N flags
CPY \$address	11000100	Compare Y register with memory[address], sets Z, N flags

There is no IF statement in Machine Code

- Every instruction that changes a register (and the compare instructions) sets the result flags
 - Z = 1 if the result is zero, 0 if the result is no zero
 - N = 1 if the result is negative and 0 if the result is positive
- The CMP instruction is like subtraction – it sets Z and N but ignores the result value of the subtraction

Result	Z	N
0	1	0
positive	0	0
negative	0	1

There is no IF statement in Machine Code

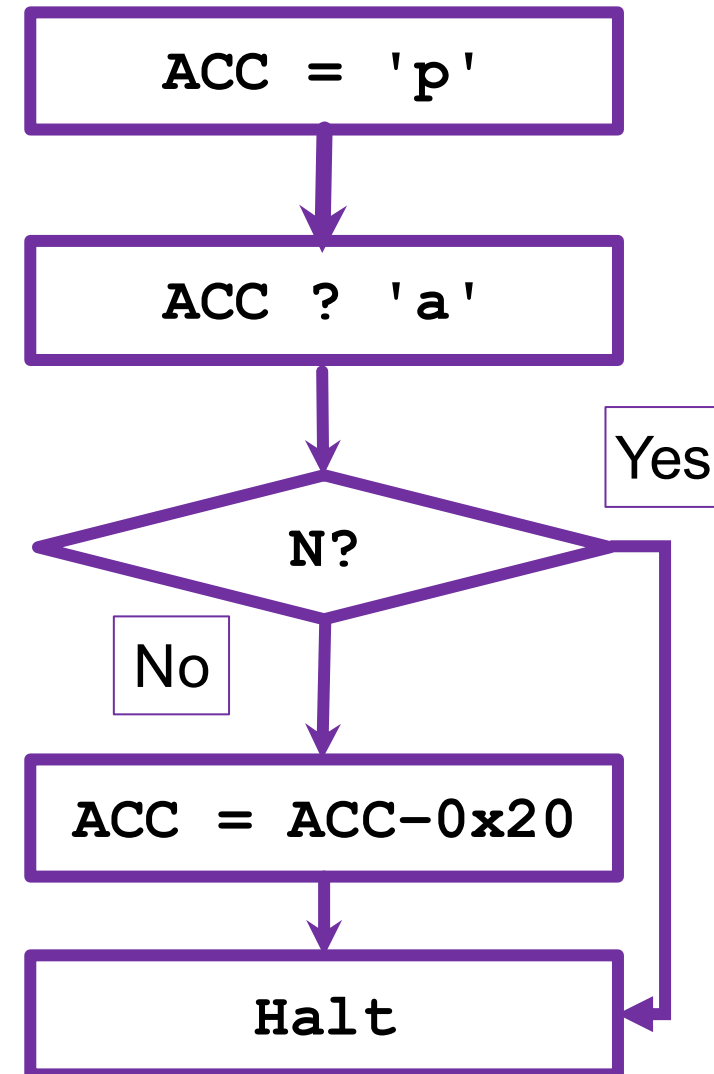
- The closest we have to an IF is a CMP + conditional jump

Assembly	Opcode	Description
BEQ \$address	11110000	Branch if equal (Z flag set)
BNE \$address	11010000	Branch if not equal (Z flag clear)
BMI \$address	00110000	Branch if minus (N flag set)
BPL \$address	00010000	Branch if plus (N flag clear)

```
LDA #'p'      ; ASCII value for 'p'
CMP #'a'      ; ASCII value for 'a'
BMI $08       ; Branch if minus (less than)
SBC #0x20     ; convert to uppercase
BRK           ; This is at 0x08
```

```
if acc < 'a' : goto skip
acc = acc - 0x20
skip: BRK
```


0x00	10101001	LDA #'p'
0x01	01110000	(the 'p')
0x02	11001001	CMP #'a'
0x03	01100001	(the 'a')
0x04	00110000	BMI \$08
0x05	00001000	(the 0x08)
0x06	11101001	SBC #0x20
0x07	00101010	(the 0x20)
0x08	00000000	BRK



Assembly gives us labels

- No need to compute instruction addresses by hand

```
LDA #'p'  
CMP #'a'  
BMI skip  
SBC #0x20  
skip:  
BRK
```

0x00	10101001	LDA #'p'
0x01	01110000	(the 'p')
0x02	11001001	CMP #'a'
0x03	01100001	(the 'a')
0x04	00110000	BMI 0x08
0x05	00001000	(the 0x08)
0x06	11101001	SBC #0x20
0x07	00101010	(the 0x20)
0x08	00000000	BRK

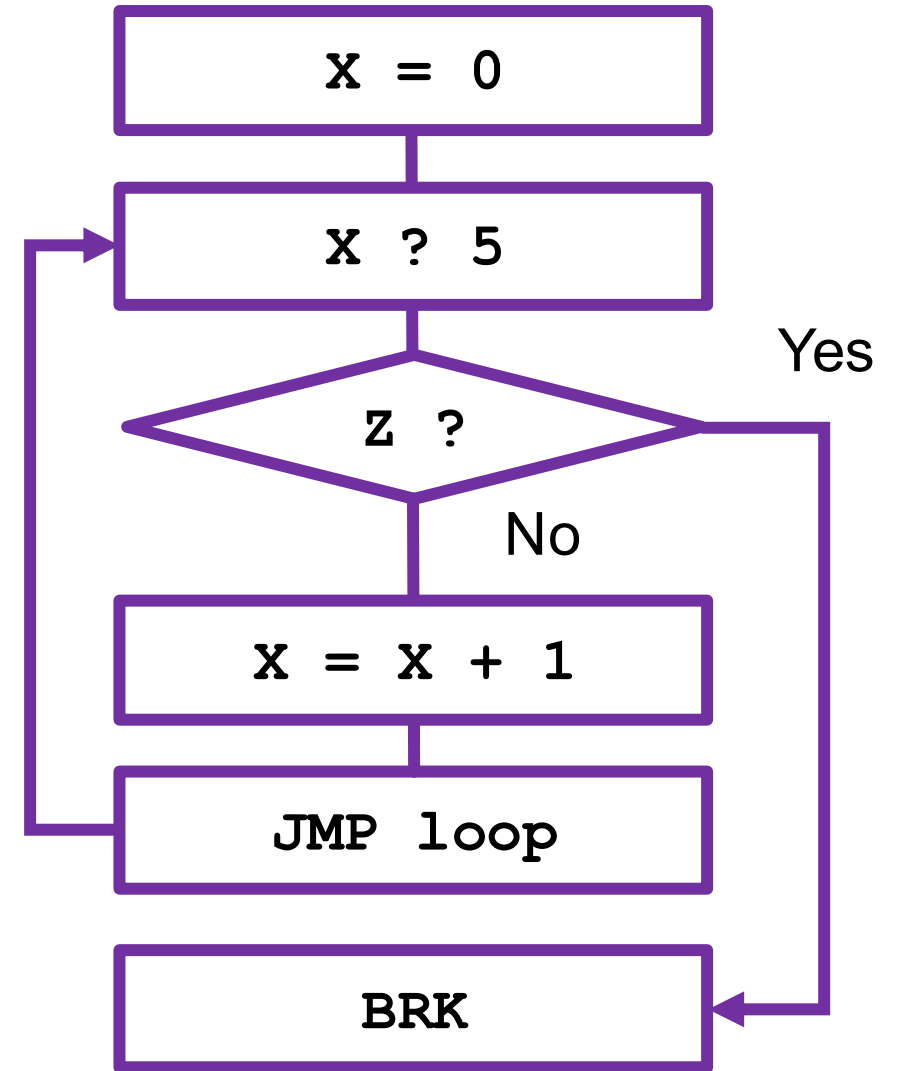
Jumps are far simpler than "if logic"

- You can think of the unconditional Jump as a SET of the Program Counter (PC) register
- A conditional jump is a conditional set of the PC register

Assembly	Opcode	Description
JMP \$address	01001100	Unconditional jump to absolute address (zero-page)
BEQ \$address	11110000	Branch if equal (Z flag set)
BNE \$address	11010000	Branch if not equal (Z flag clear)
BMI \$address	00110000	Branch if minus (N flag set)
BPL \$address	00010000	Branch if plus (N flag clear)

There is no "for" or "while" in machine language

```
CLX          ; Clear X register
loop:
CPX #5       ; Compare X to 5
BEQ end      ; Branch if equal (Z flag set)
INX          ; Increment X
JMP loop     ; Jump to loop
end:
BRK          ; Halt
```



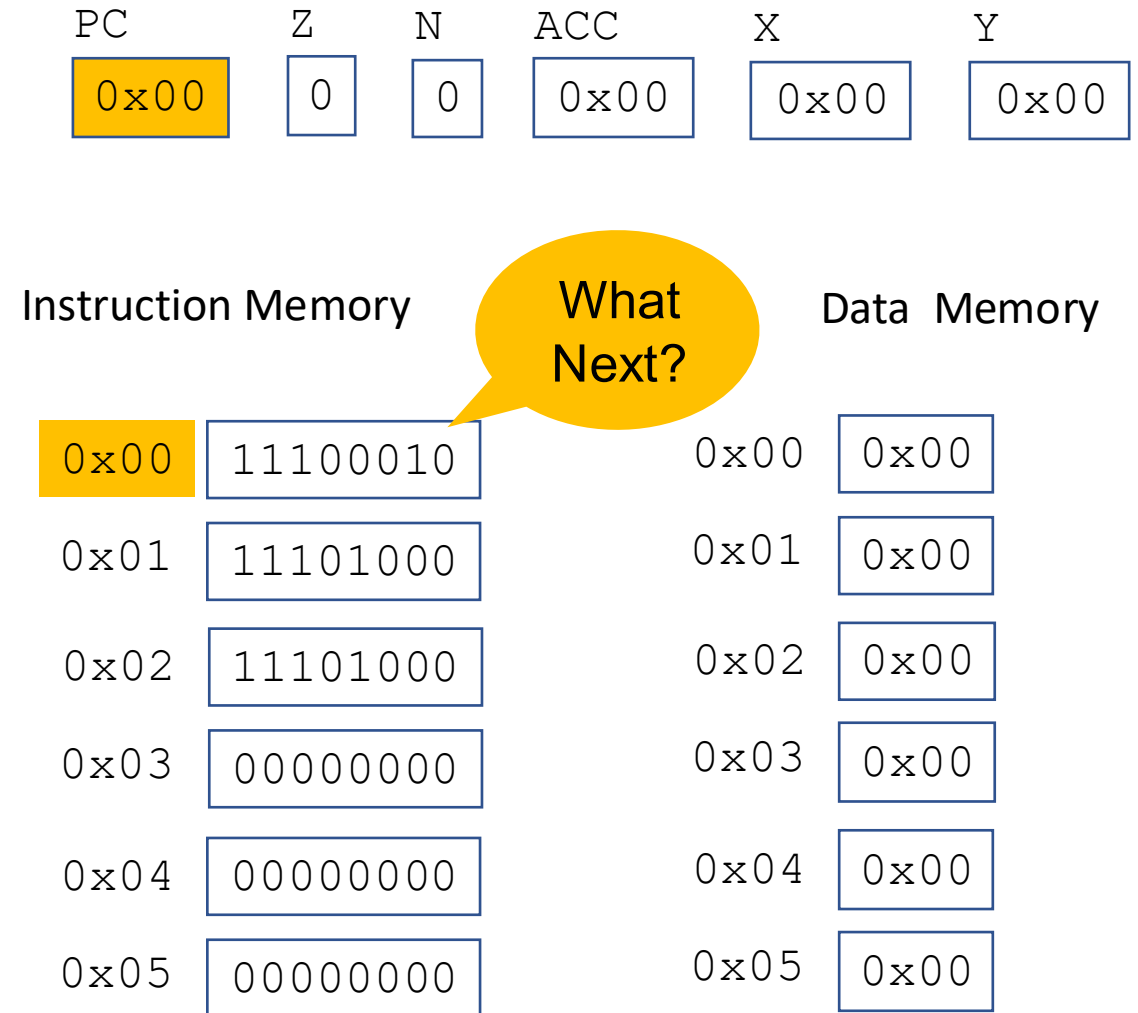
Assembly	Opcode	Description
CLX	11100010	Clear X register ($X = 0$). Sets Z flag to 1, N flag to 0.
INX	11101000	Increment X register by 1
LDA #value	10101001	Load accumulator with immediate value ($ACC = value$)
LDA \$address	10100101	Load accumulator ($ACC = memory[address]$)
STA \$address	10000101	Store accumulator ($memory[address] = ACC$)
ADC #value	01101001	Add (immediate): $ACC = ACC + value$
SBC \$address	11100101	Subtract (from memory): $ACC = ACC - memory[address]$
CMP #value	11001001	Compare accumulator with immediate value, sets Z, N flags
JMP \$address	01001100	Unconditional jump to absolute address (zero-page)
BEQ \$address	11110000	Branch if equal (Z flag set)
BMI \$address	00110000	Branch if minus (N flag set)
BRK	00000000	Break/Halt - Stop program execution

CDC 6504 – Assembly Language

<https://www.ca4e.com/cdc6504/documentation.html>

CDC6504 Architecture

- PC – Program Counter
- ACC – Accumulator Register
- Z – Was the last result zero?
- N – Was the last result negative?
- X and Y – Index registers
- Instruction memory shown in base-2 for easy decoding
- Data memory shown in hex – will usually contain ASCII



Data Directive – Assembly Only

Assembly	Description
DATA 'string'	Pre-populates data memory (not instruction memory) with a null-terminated string. Data is loaded starting at memory address 0.
DATA 0x00 0x0A ...	Pre-populates data memory (not instruction memory) with a space-separated list of hexadecimal values. Data is loaded starting at memory address 0.

```
BRK
DATA 'Hello World! '
```

-- Load Program --



Status: **Halted**

Printed: **Hello World!**

Add 27 + 15 and Store

```
LDA #27    ; Load 27 into accumulator
ADC #15    ; Add 15 (result = 42)
STA $00    ; Store result to memory[0]
BRK
```

Status: **Halted**

Printed: *

Reset

Step

Start

PC:

0x05

Z:

☐

N:

☐

ACC:

0x2A

X:

0x00

Y:

0x00

Instructions

0x00: 10101001

0x01: 00011011

0x02: 01101001

0x03: 00001111

0x04: 10000101

0x05: 00000000

Memory

0x00: 0x2A

0x01: 0x00

0x02: 0x00

0x03: 0x00

0x04: 0x00

0x05: 0x00

Status: **Halted**

Printed: **Hello**

Reset

Step

Start

PC:

0x08

Z:

☐

N:

☐

ACC:

0x6F

X:

0x00

Y:

0x00

Instructions

0x00:	10101001
0x01:	01001000
0x02:	10000101
0x03:	00000000
0x04:	10101001
0x05:	01100101
0x06:	10000101

Memory

0x00:	0x48
0x01:	0x65
0x02:	0x6C
0x03:	0x6C
0x04:	0x6F
0x05:	0x00
0x06:	0x00

Print Hello

LDA #'H'	; Load 'H' into ACC
STA \$00	
LDA #'e'	; Load 'e' into ACC
STA \$01	
LDA #'l'	; Load 'l' into ACC
STA \$02	
STA \$03	; Store 'l' again
LDA #'o'	; Load 'o' into ACC
STA \$04	
BRK	

Uppercase a String

```
CLX
loop:
LDA $00,X
BEQ done
CMP #'a'
BMI cont
SBC #0x20
STA $00,X
cont:
INX
JMP loop
done:
BRK

DATA 'Hello'
```

Status: Stopped
Printed: No output yet

Reset Step Start

PC: 0x00 Z: ☐ N: ☐

ACC: 0x00 X: 0x00 Y: 0x00

Instructions

0x00:	11100010
0x01:	10110101
0x02:	00000000
0x03:	11110000
0x04:	00001011
0x05:	11001001
0x06:	01100001
0x07:	00110000
0x08:	00000100

Instructions

0x10:	00000000
0x11:	00000000
0x12:	00000000
0x13:	00000000
0x14:	00000000
0x15:	00000000
0x16:	00000000
0x17:	00000000
0x18:	00000000

Memory

0x00:	0x48
0x01:	0x65
0x02:	0x6C
0x03:	0x6C
0x04:	0x6F
0x05:	0x00
0x06:	0x00
0x07:	0x00
0x08:	0x00

Uppercase a String

```
CLX
loop:
LDA $00,X
BEQ done
CMP #'a'
BMI cont
SBC #0x20
STA $00,X
cont:
INX
JMP loop
done:
BRK

DATA 'Hello'
```

Status: Halted
Printed: HELLO

Reset Step Start

PC: 0x10 Z: ☒ N: ☐

ACC: 0x00 X: 0x05 Y: 0x00

Instructions

0x00:	11100010
0x01:	10110101
0x02:	00000000
0x03:	11110000
0x04:	00001011
0x05:	11001001
0x06:	01100001
0x07:	00110000
0x08:	00000100

Instructions

0x10:	00000000
0x11:	00000000
0x12:	00000000
0x13:	00000000
0x14:	00000000
0x15:	00000000
0x16:	00000000
0x17:	00000000
0x18:	00000000

Memory

0x00:	0x48
0x01:	0x45
0x02:	0x4C
0x03:	0x4C
0x04:	0x4F
0x05:	0x00
0x06:	0x00
0x07:	0x00
0x08:	0x00

Uppercase a String (Python)

```
>>> "Hello".upper()  
'HELLO'  
>>>  
  
mem = "Hello"  
newmem = ""  
  
for xr in range(len(mem)):  
    acc = mem[xr:xr+1]  
    if acc >= 'a' and acc <= 'z' :  
        acc = chr(ord(acc) - ord('a') + ord('A'))  
  
    newmem = newmem + acc  
  
print(newmem)
```

Uppercase a String (C – array version)

```
#include <stdio.h>

int main() {
    register int xr,acc;
    char mem[6] = "Hello";

    for(xr=0;mem[xr];xr++) {
        acc = mem[xr];
        if ( (acc - 'a') < 0 ) continue;
        mem[xr] = acc - 0x20;
    }

    printf("%s\n", mem);
}
```

Uppercase a String (walkthrough)

```
CLX                ; for (X=0;mem[X];X++)
loop:
LDA $00,X          ; ACC = mem[$00 + X]
BEQ done           ; for (X=0;mem[X];X++)
                   ; Branch if Z flag is set

CMP #'a'           ; In ASCII 'A' < 'a'
BMI cont           ; continue to X++
SBC #0x20          ; ACC = ACC - 0x20
STA $00,X          ; mem[$00 + X] = ACC
cont:
INX                ; for (X=0;mem[X];X++)
JMP loop
done:
BRK

DATA 'Hello'
```

Memory

0x00:	0x48
0x01:	0x65
0x02:	0x6C
0x03:	0x6C
0x04:	0x6F
0x05:	0x00
0x06:	0x00
0x07:	0x00
0x08:	0x00

Memory

0x00:	0x48
0x01:	0x45
0x02:	0x4C
0x03:	0x4C
0x04:	0x4F
0x05:	0x00
0x06:	0x00
0x07:	0x00
0x08:	0x00

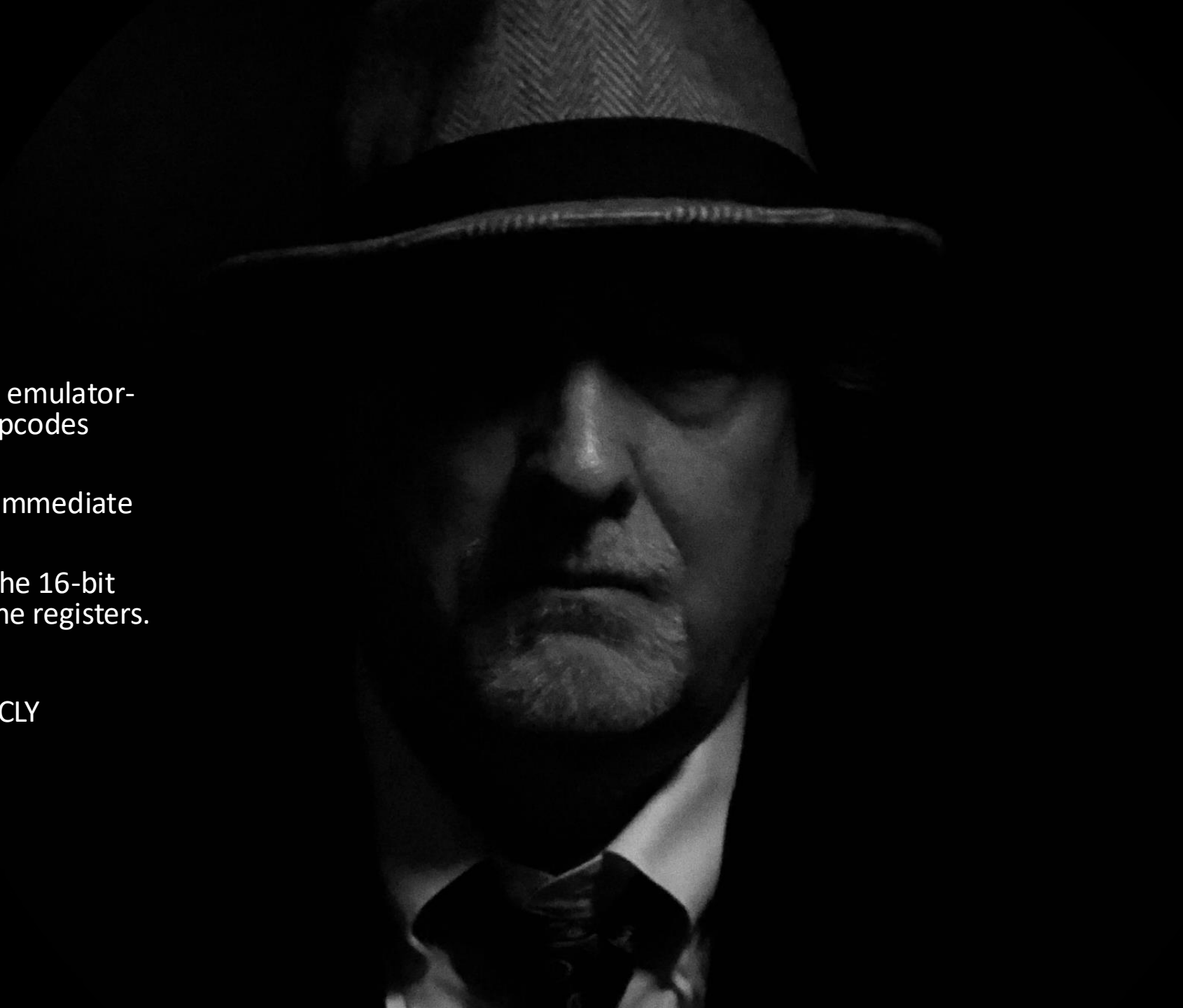
CLX is not really a 6502 instruction

The clear instructions (CLX, CLY) are 8-bit emulator-only instructions that use unused 6502 opcodes (0xE2, 0xC2).

I wanted to hold off introducing (16-bit) immediate instructions for the simplest examples.

In a real hardware 6502, you would use the 16-bit LDX #0 and LDY #0 instructions to clear the registers.

AI: Why didn't the 6502 include CLX and CLY instructions?



Summary

- Fetch-Decode-Execute (a.k.a. Fetch-Execute-Cycle)
- Explored the CDC6504 architecture, instruction set, assembly and machine languages

Acknowledgements / Contributions

These slides are Copyright 2025- Charles R. Severance (online.dr-chuck.com) as part of www.ca4e.com and made available under a Creative Commons Attribution 4.0 License. Please maintain this last slide in all copies of the document to comply with the attribution requirements of the license. If you make a change, feel free to add your name and organization to the list of contributors on this page as you republish the materials.

Initial Development: Charles Severance, University of Michigan School of Information

Insert new Contributors and Translators here including names and dates

Continue new Contributors and Translators here