

01 The Digital World

01.1 Logic and Numbers

01.2 Computer Architecture

01.1 Intro to logic and numbering systems

- Modern digital hardware
- Digital functions and logic symbols
- Voltages and 0 1 x z – tri-state buffer
- Positional number systems: decimal, binary, hexadecimal
- Grouping of bits
- Signed numbers
- Symbols: ASCII and Unicode

01.1 Intro to logic and numbering systems

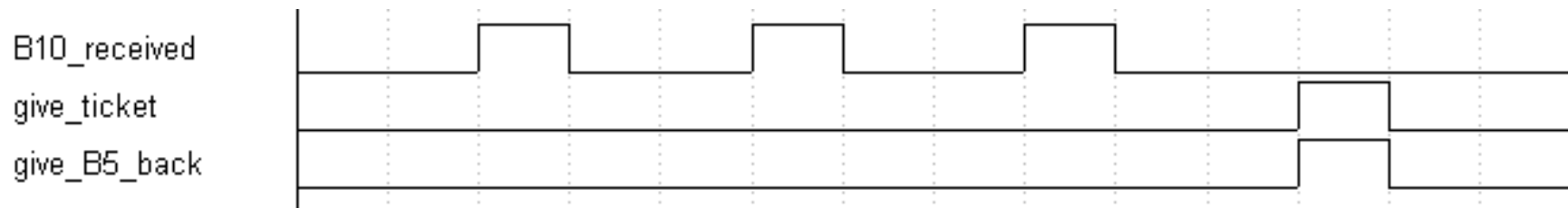
What is digital hardware?

- A system that physically store and process digital data.
- Digital data : two possible states/values
 - 0 = FALSE (F) = Low voltage (L)
 - 1 = TRUE (T) = High voltage (H)also called binary data.
- Modern implementation use voltages to indicate logic 0 (Ground) or logic 1 (+V).

Digital Signals (waveforms / timing diagram)

- signal: a function of time
- digital signal: value of function is 0 or 1
 - 0 = low, 1 = high

Example: Buying a 25-Baht BTS Ticket

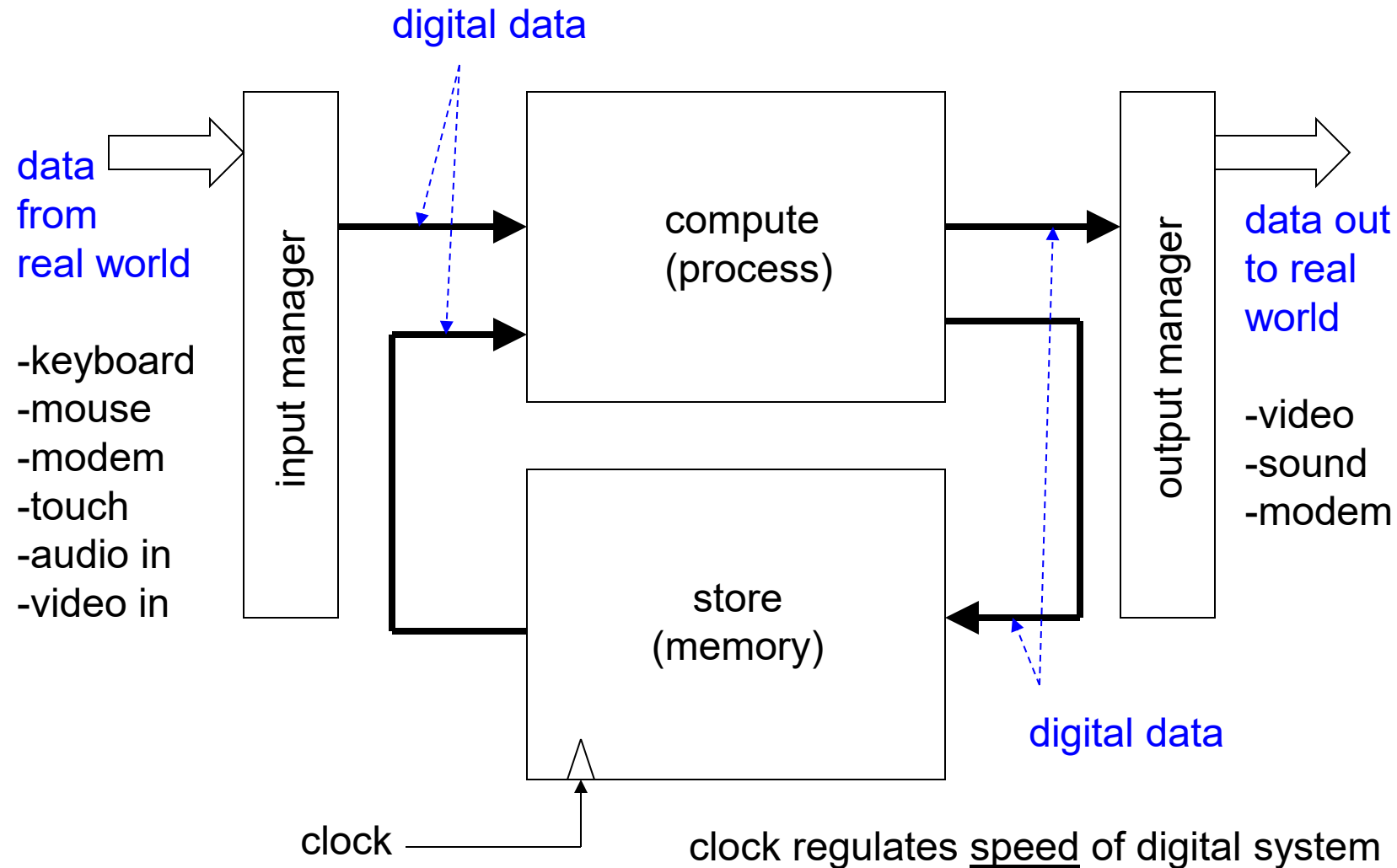


Gotcha1: Some signals are undefined, unknown, or we simply don't care for it. We use X instead of explicitly writing 0 or 1. (More later)

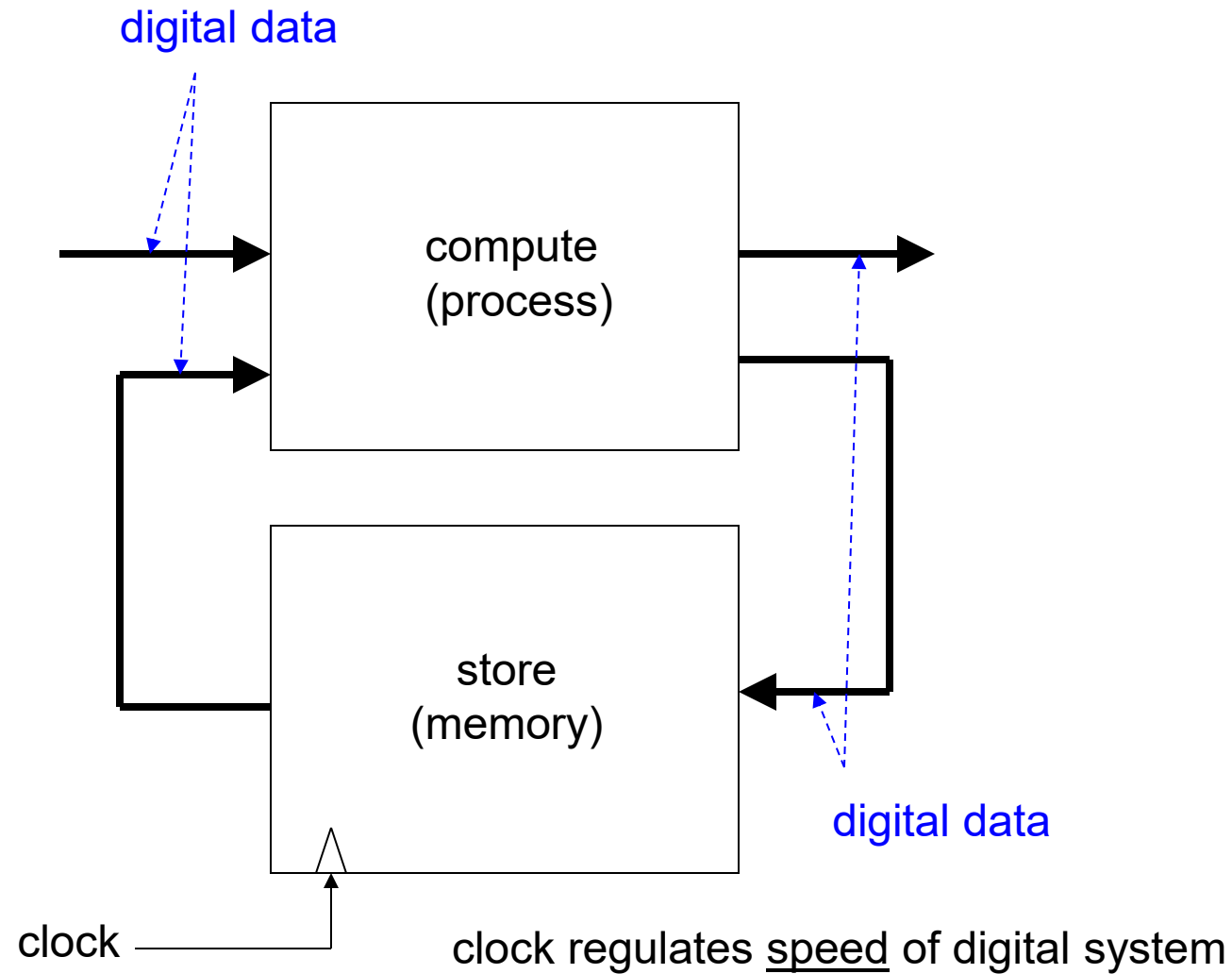
Gotcha2: Some signals are not connected to a voltage source (to make it 0 or 1). We use Z for those signals

(Z = float, high output impedance, disabled) (More later)

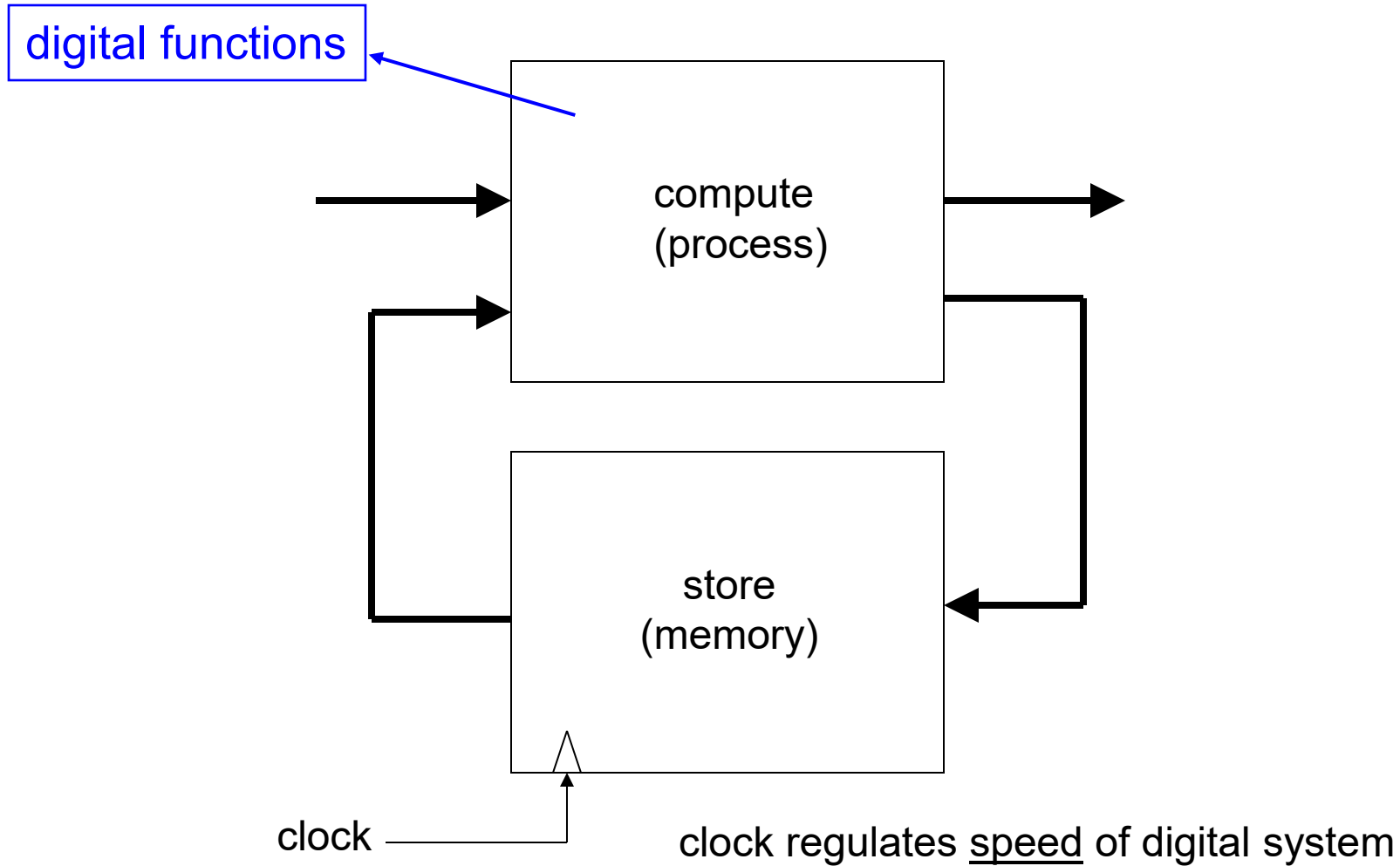
Organization of digital hardware:



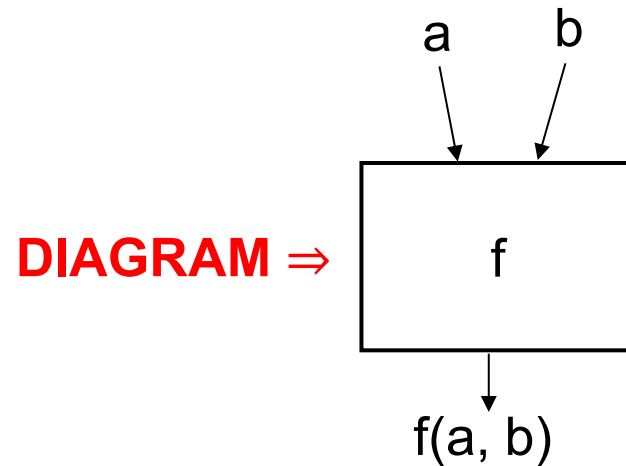
Organization of digital hardware:



Organization of digital hardware:



Digital functions (binary functions) has 2 views:



binary function

$$f(a, b) = ab \quad \Leftarrow \text{EQUATION}$$

only 4 possibilities
 $ab = 00, 01, 10, 11$

a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table

A table that lists all possible inputs to a function and the corresponding output

Truth table completely describes the function

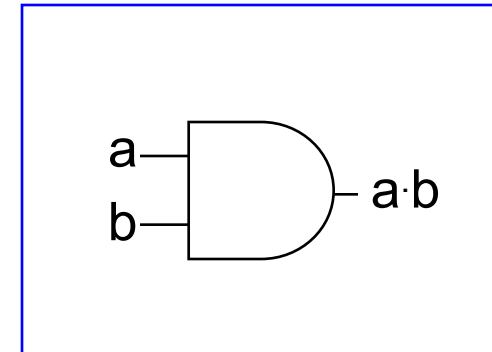
Some famous functions:

AND

a	b	$f = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

$$f = a \cdot b$$

AND gate symbol

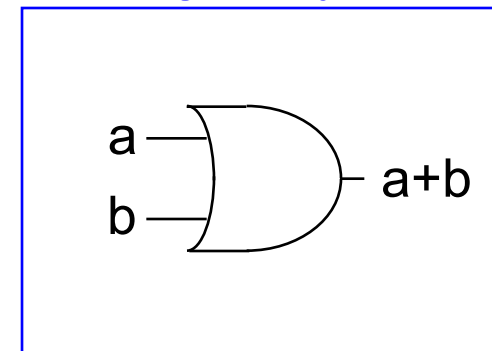


OR

a	b	$f = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

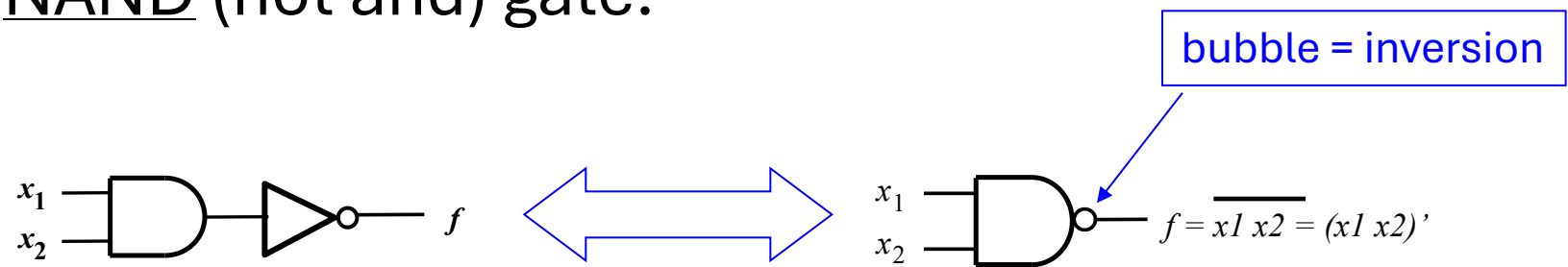
$$f = a + b$$

OR gate symbol



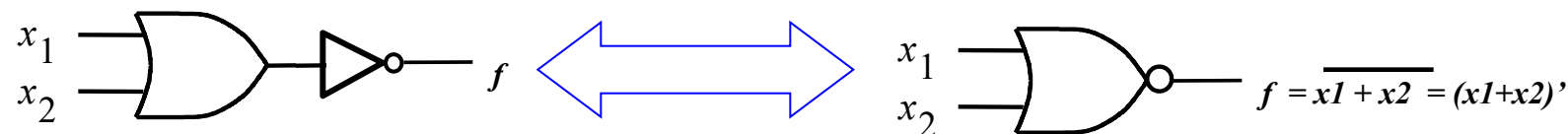
More gates:

NAND (not and) gate:



NOR (not or) gate:

[BV]



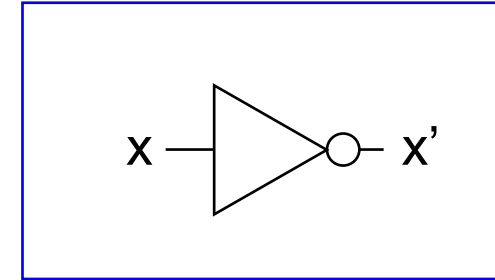
X1	X2	X1 X2	(X1 X2)'	X1+X2	(X1+X2)'
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	0	1	0

NOT (invert)

function:

input x can be
either 0 or 1

x	NOT x
0	1
1	0



inverter (NOT gate) symbol

Symbols for NOT x: \overline{x} x' $\sim x$ $!x$ (x bar, x prime, tilde x, bang x)

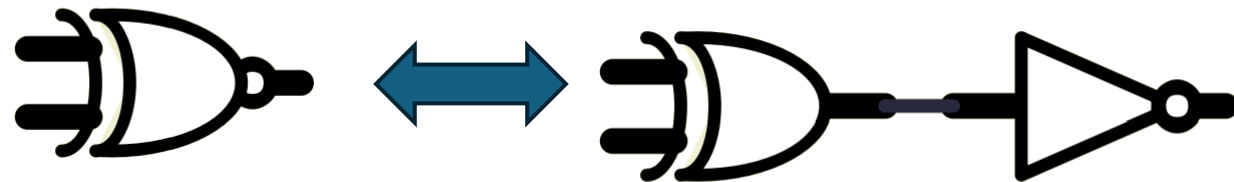
XOR Function

a	b	$f = a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0



XOR gate

XNOR gate is XOR followed by inverter

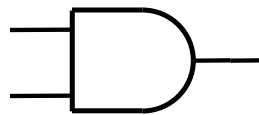


XNOR gate

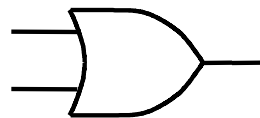
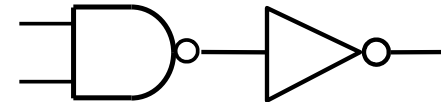


Why NAND's and NOR's?

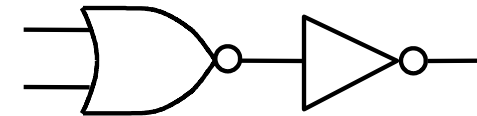
- In real logic circuits, NOT, NAND, and NOR tend to be “basic” gates (easiest to build)
- In real logic circuits, AND and OR are actually NAND or NOR followed by NOT!



is actually



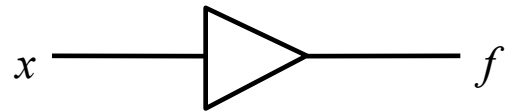
is actually



[BV]

Even more gates 2:

Buffer or Follower (BUF): output = input



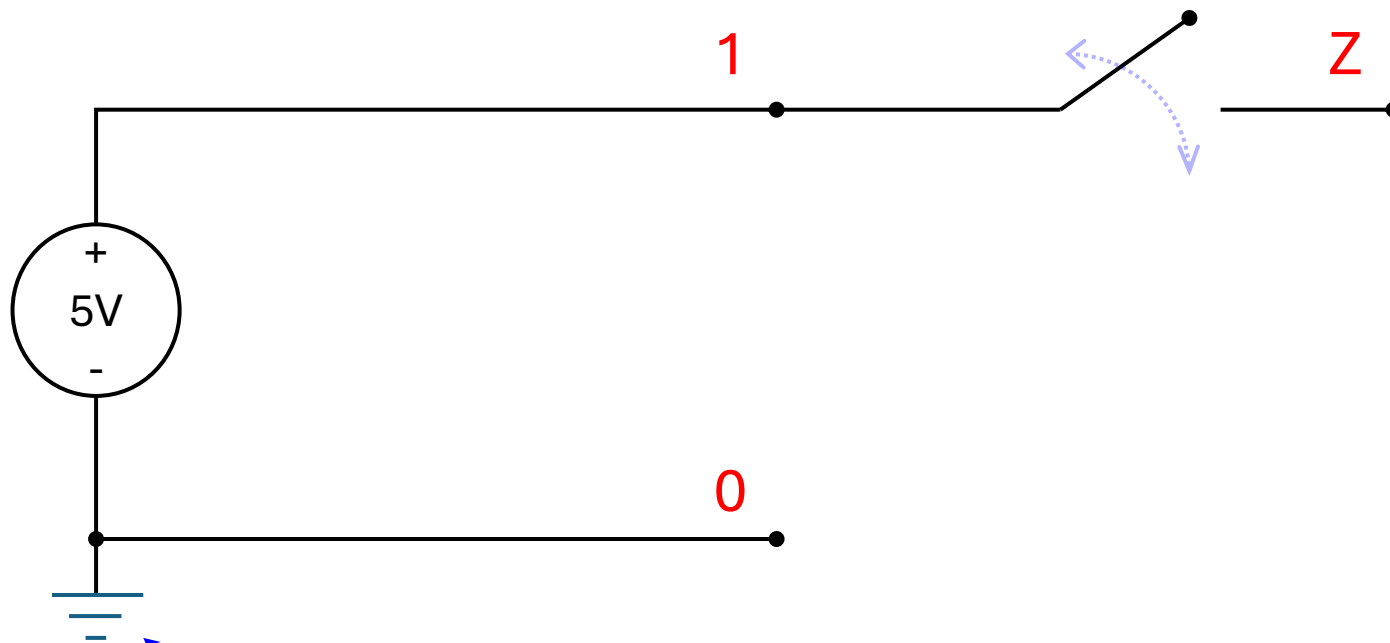
Graphical symbol

Truth table

x	f
0	0
1	1

What physically are 0 & 1?

Modern digital systems use voltages to represent 0 or 1.



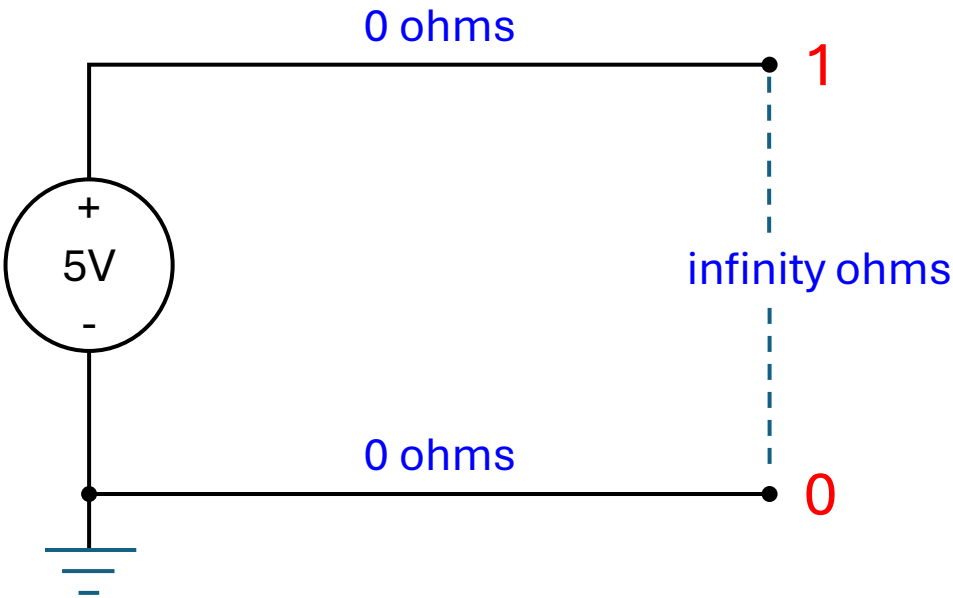
Besides 0 & 1. Some electrical nodes are **not connected** to any voltage source. Those nodes are called Z. Other names are tri-state, high-Z, high-impedance, or floating.

This is called a “ground” (GND) symbol. We define this point to be 0 volts. All other voltages are measured relative to this point.
Analogy: height of a mountain. Where do you define 0 to be?



What physically are 0 & 1?

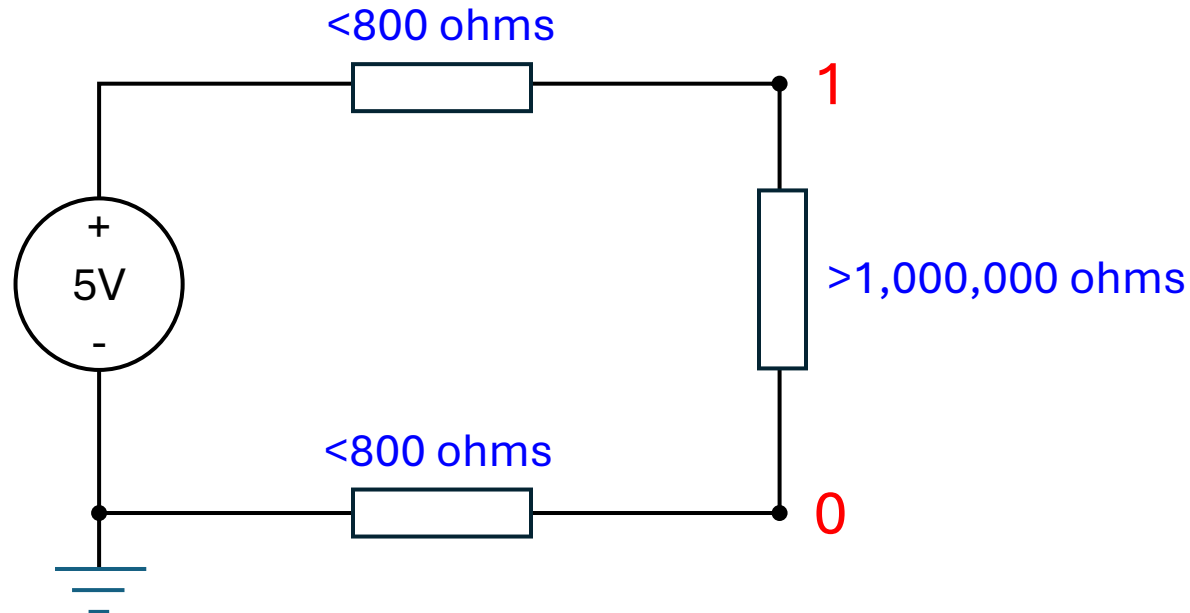
Ideally:





What physically are 0 & 1?

In reality (for off-chip signals):



Conclusion: **1** is still very close to 5V and **0** is still very close to 0V

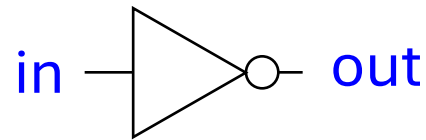


So, I only need 0's and 1's?

Just 0's and 1's are not enough to model logic.

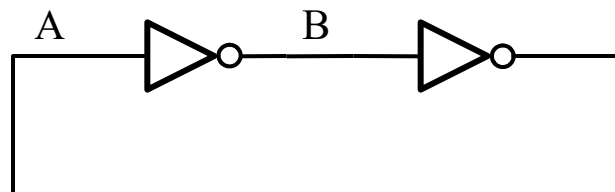
Counterexample:

Recall inverter:



in	out
0	1
1	0

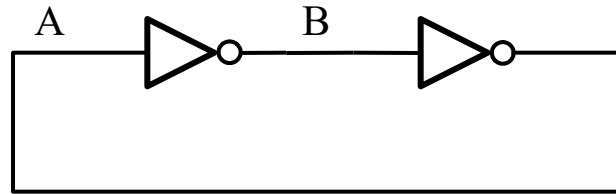
and now hook up this circuit:





So, I only need 0's and 1's?

After power up, sometimes you see A=1 and B=0, but sometimes you see A=0 and B=1.



What's the value of A after power up?

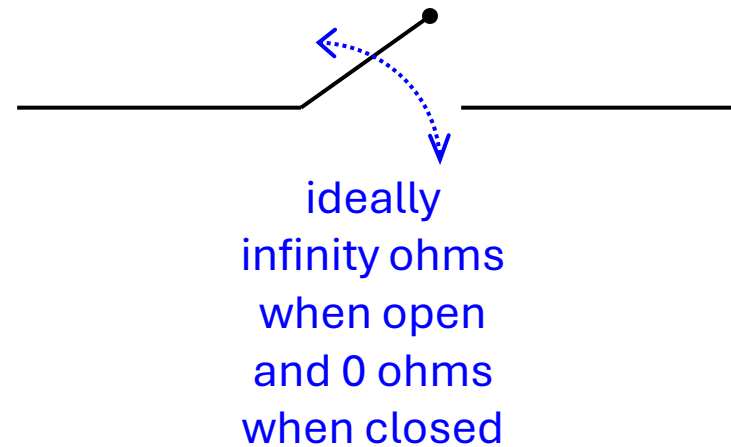
- we don't know (and sometimes we don't care)
- So, we call this unknown X, and X **MUST BE** either 0 or 1



So, I only need 0's and 1's and X's?

No. You need one more. Because we can build this circuit, which is an open-close switch that can be controlled.

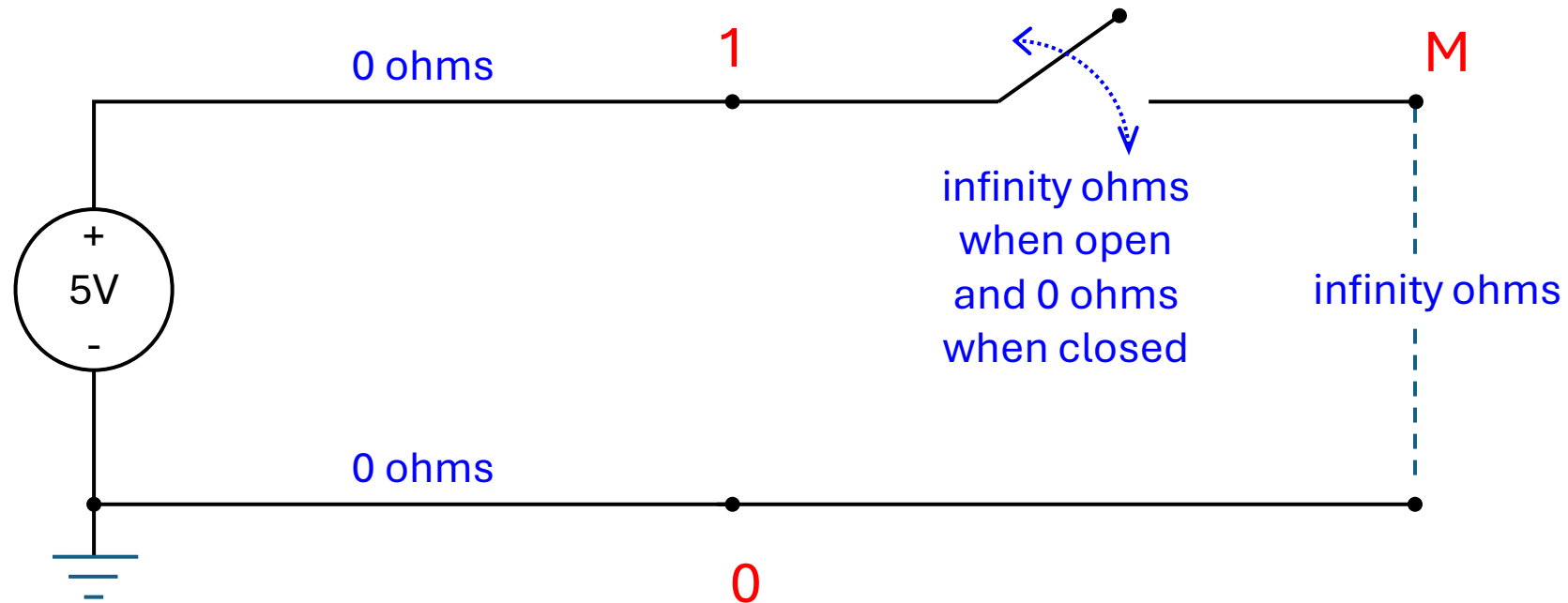
Technical name for this circuit is called transmission gate.





So, I only need 0's and 1's and X's?

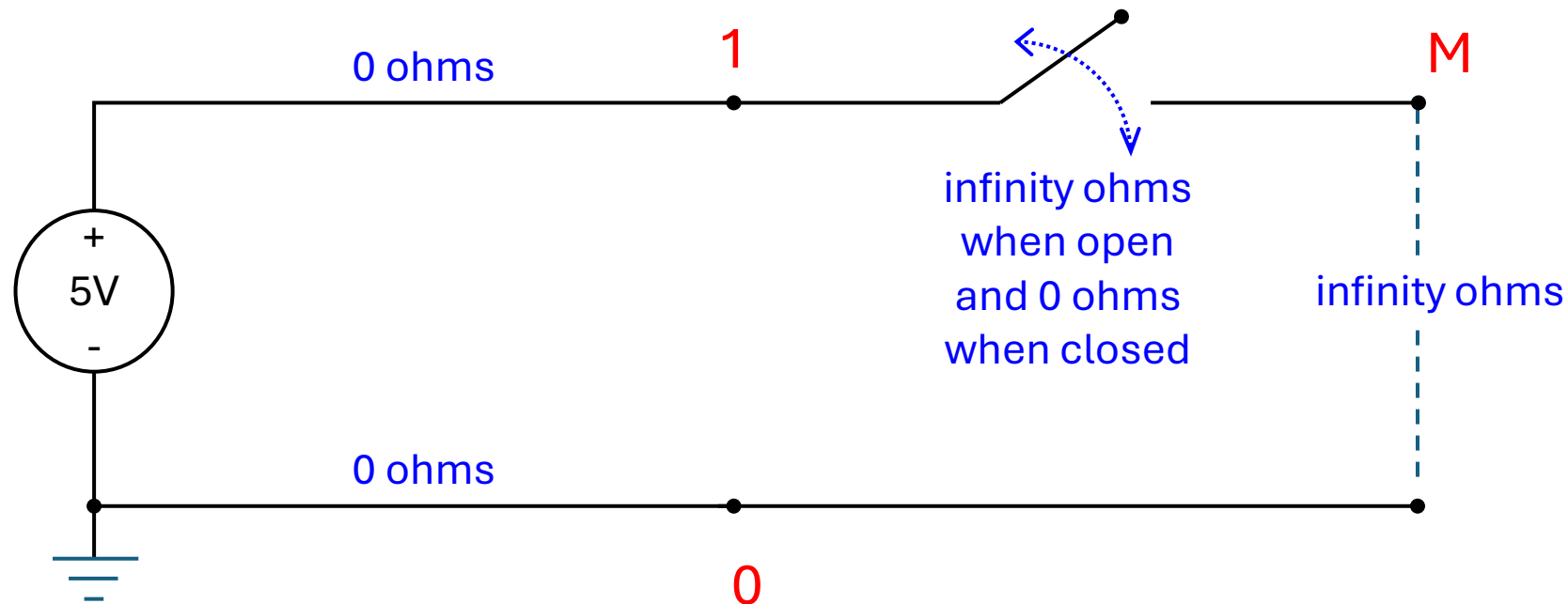
Now, if we have transmission gate, what's the logic value of M?
M is not 0. M is not 1. M is not X (remember X must be 0 or 1).





So, I only need 0's and 1's and X's?

M is just floating, and there's no voltage defined for M. In this case, we call M floating, high-impedance, in-tri-state, high-Z, or Z.



So, there are 0, 1, x, and z

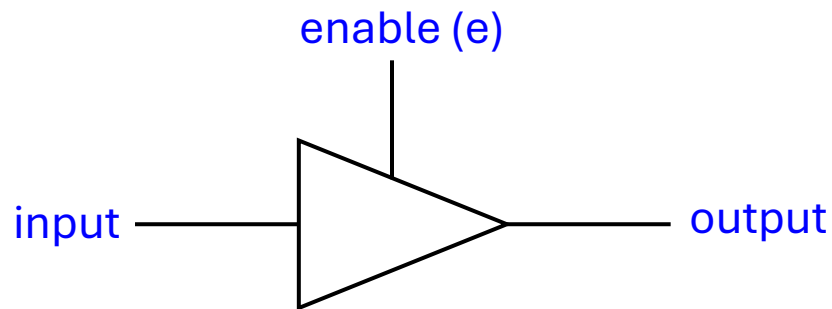
value	what it is
0	connected to GND by a low impedance (resistance) connection
1	connected to 5V by a low impedance (resistance) connection
x	must be 0 or 1 , but it is unknown to us, or we may not care whether it is 0 or 1. <u>This only exists in simulation.</u> In real world, x can always be measured and known.
z	is not connected to GND or 5V. It is essentially floating. no voltage defined.

In real life digital circuits, every electrical node can be categorized as 0, 1, z. Although we have x only in simulation, it is very important that we designate them as x (learn why in future courses).



Example of z: tri-state buffer

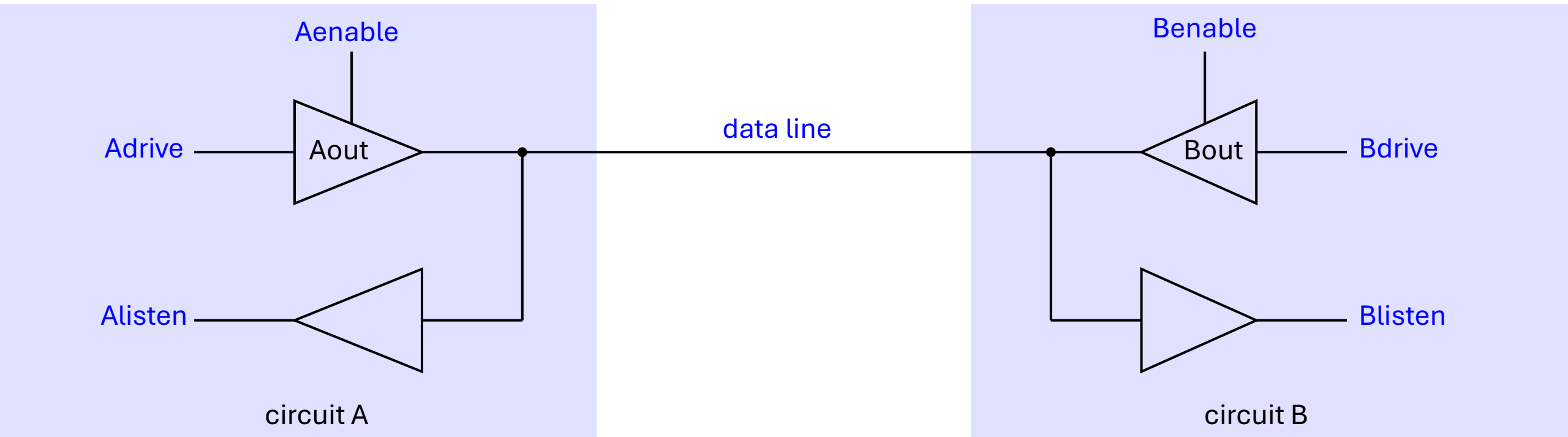
The following circuit is called a tri-state buffer:



enable	input	output
0	0	Z
0	1	Z
1	0	0
1	1	1

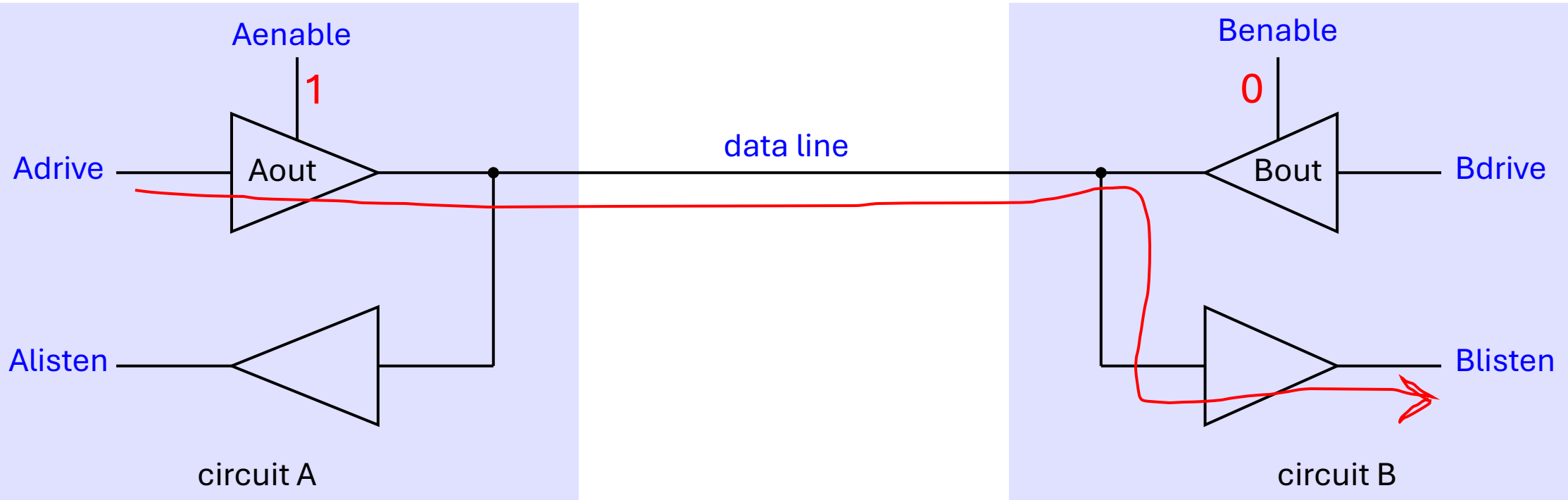


Bi-directional communication





Bi-directional communication

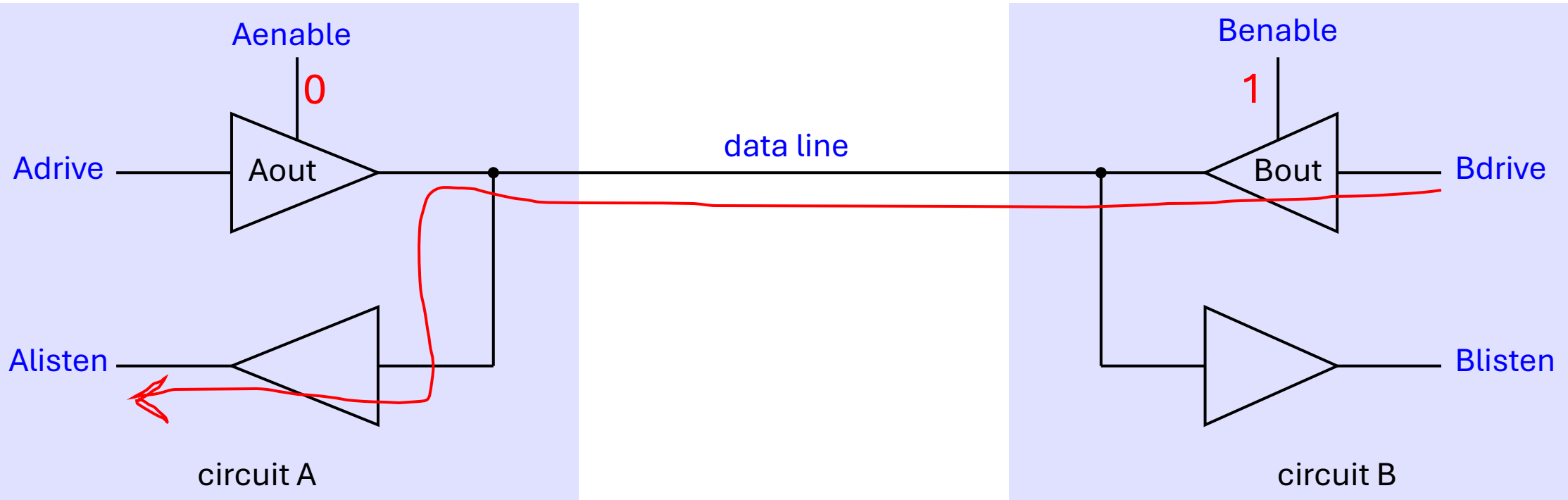


If A wants to talk to B:

- set Aenable to 1, → Aout follows Adrive
- set Benable to 0, → Bout is disconnected
- data line will take value of Adrive → Blisten



Bi-directional communication



If B wants to talk to A:

- set **Aenable** to 0, → **Aout** is disconnected
- set **Benable** to 1, → **Bout** follows **Bdrive**
- data line will take value of **Bdrive** → **Alisten**

intentionally left blank

01.1 Intro to logic and numbering systems

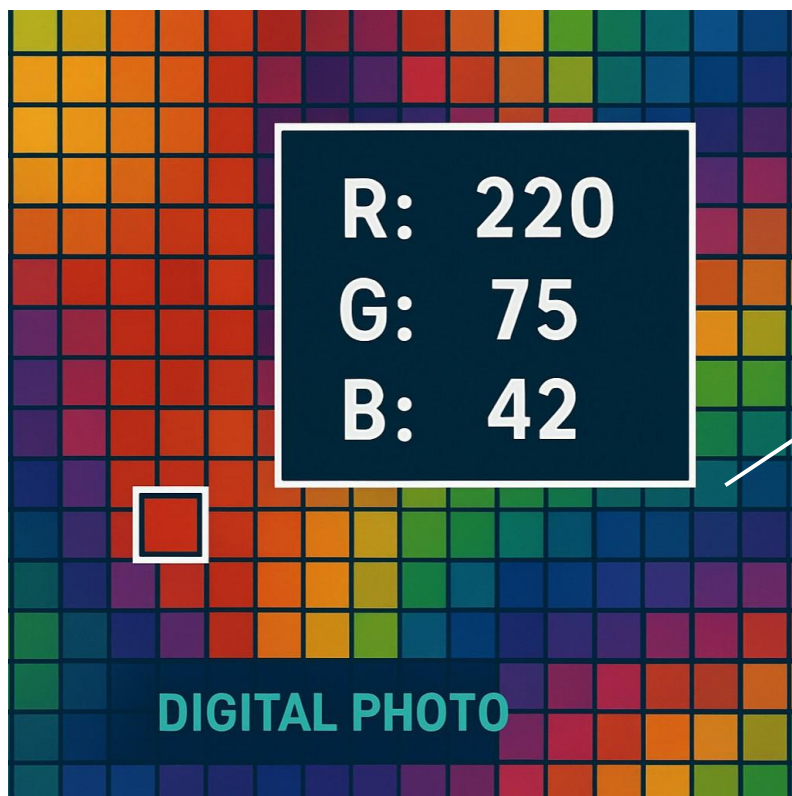
Numbers are very important for computers:

- It is what we mostly use computers for: $+$ $-$ \times \div $\sqrt{}$ $\log()$
- It is also used to represent data from the real world:
 - **Pictures**
 - **Video**
 - **Audio**
 - **Stereo**
 - **Surround sound**



01.1 Intro to logic and numbering systems

- **Pictures** are coded as pixels. Each pixel has (red, green, blue) numbers
 - **Video** is just a sequence of pictures. So, they're all numbers.



<-- Not a real-world picture

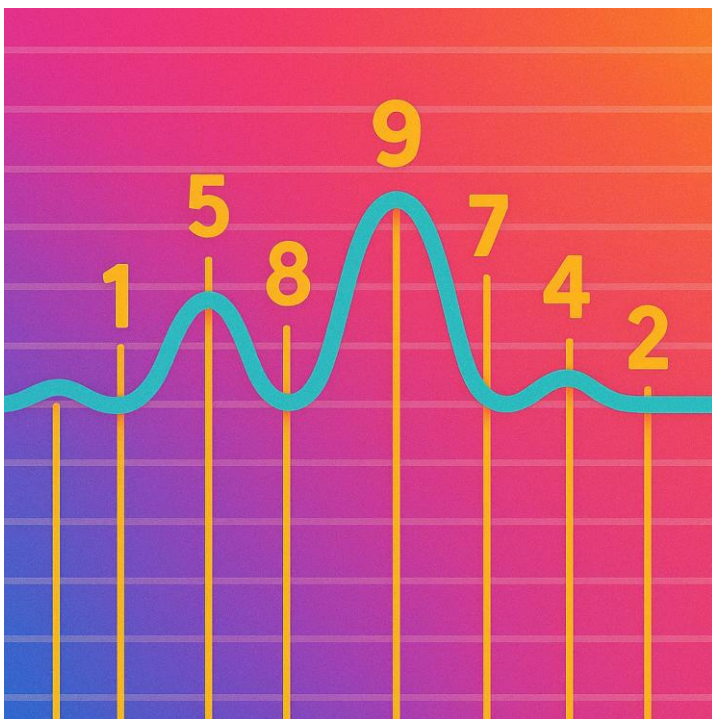
From the real world:

iPhone 17 Pro back cameras have 48 million pixels
4k TV has 3840 x 2160 pixels (8.3 million pixels)



01.1 Intro to logic and numbering systems

- **Audio** is amplitude sampling from the audio waveform.
 - **Stereo** is 2 independent audio channels.
 - **Surround sound** has many independent audio channels.



<-- Not a real-world audio signal

From the real world:

- CD audio has 2 tracks (left and right)
- Each track's audio waveform is sampled 44,100 times per second.
- Each sample can have 65,536 different values. ($\text{SNR} \approx 6.31 \times 10^9$)

Positional Number System, Bin, and Hex

What do you mean by 7512?

$$7512_{10} = 7 \times 10^3 + 5 \times 10^2 + 1 \times 10^1 + 2 \times 10^0$$

7 is called the most significant digit

2 is called the least significant digit

This is called a positional number system. Each symbol 0...9 carries different weights depending on location.

This is also called a base-10 (decimal) number, because we have 10 symbols (0...9) and each position is 10x more value than the position to its right. We (optionally) write “10” subscript to denote that the number is decimal.

$$\text{Likewise, } 12.87_{10} = 1 \times 10^1 + 2 \times 10^0 + 8 \times 10^{-1} + 7 \times 10^{-2}$$

Positional Number System, Bin, and Hex

Digital systems provide 0's and 1's, so we can build the same positional number system with 2 symbols (0, 1), and each position is 2x more value than the position to its right. So,

$$110.01_2 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^{-2} = 6.25_{10} \quad \text{You get the idea...}$$

1011011011010₂

This is the most significant binary digit (bit) – MSB

This is the least significant binary digit (bit) -- LSB

What bases are used in digital systems?

- **base 2** (0,1) called **binary** because we have only 0's and 1's
- **base 8** (0, 1, 2, 3, 4, 5, 6, 7) called **octal**. Each octal digit can be turned into 3 binary digits
- **base 16** (0...9, A, B, C, D, E, F) called **hexadecimal** (**hex** for short). Each hex digit can be turned into 4 binary digits

01.1 Intro to logic and numbering systems

How to quickly convert positive binary to decimal?

- Recall $mnpqrs_2 = m \cdot 2^5 + n \cdot 2^4 + p \cdot 2^3 + q \cdot 2^2 + r \cdot 2^1 + s \cdot 2^0$
 $= m \cdot 32 + n \cdot 16 + p \cdot 8 + q \cdot 4 + r \cdot 2 + s$

Let's try:

$$\begin{array}{cccccc} & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 2 = 32 + 8 + 4 + 1 = 45_{10} \\ 0 & 1 & 1 & 0 & 1 & 0 & 2 = 16 + 8 + 2 = 36_{10} \end{array}$$

This also works in reverse: Find the largest one and keep subtracting...

$$\begin{array}{l} 39 = 32 + (7) = 32 + 4 + (3) = 32 + 4 + 2 + 1 = 1 \ 0 \ 0 \ 1 \ 1 \ 1 \\ 57 = 32 + (25) = 32 + 16 + (9) = 32 + 16 + 8 + 1 = 1 \ 1 \ 1 \ 0 \ 0 \ 1 \end{array}$$

01.1 Intro to logic and numbering systems

Hexadecimal (base 16):

- base 16 has 16 symbols 0=0, ..., 9=9,
A=10, B=11, C=12, D=13, E=14, F=15

$$\begin{aligned} 2A7F_{16} &= 2 \times 16^3 + 10 \times 16^2 + 7 \times 16^1 + 15 \times 16^0 \\ &= 10879_{10} \end{aligned}$$

Convert back to base 16:

$$\begin{aligned} 10879 / 16 &= 679 \text{ remainder } 15 (\underline{\mathbf{F}}) \\ 679 / 16 &= 42 \text{ remainder } 7 (\underline{\mathbf{7}}) \\ 42 / 16 &= \underline{\mathbf{2}} \text{ remainder } 10(\underline{\mathbf{A}}) \end{aligned}$$

Read back from bottom to top = $2A7F_{16}$

01.1 Intro to logic and numbering systems

Decimal	Binary	Octal	Hexadecimal
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

We use mostly binary, decimal, and hex

Why do we use the above three systems?

binary: we build digital (0,1) systems

hex: we don't want to write long numbers. Hex converts to/from binary easily.

decimal: we (humans) use decimal math!



01.1 Intro to logic and numbering systems

Converting from Hex to Binary:

$$2A7F_{16} = \text{????}_2$$

Convert each hex digit into 4 binary digits:

$$\begin{array}{cccc} 2 & A & 7 & F \\ = 0010 & 1010 & 0111 & 1111_2 \text{ Done!} \end{array}$$

Converting from Binary to Hex

$$10110101001001_2 = \text{????}_{16}$$

Group binary in 4 from the back,

$$\begin{array}{ccccccc} 10 & 1101 & 0100 & 1001 & = & 0010 & 1101 & 0100 & 1001 \\ & & & & = & 2 & D & 4 & 9_{16} \end{array}$$

Decimal	Binary	Octal	Hex
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

01.1 Intro to logic and numbering systems

Base	lectures / textbooks	Notation in C99	Note / Warnings
binary	11101101_2	Not available in C99	<u>We will provide some helper</u>
octal	307412_8	0307412	Must begin with 0
decimal	7692 7692_{10}	7692	Must not begin with 0. <u>No commas.</u>
hex	$8A39_{16}$ $8a39_{16}$	$0x8a39$ $0X8A39$ $0x8A39$ $0X8a39$	Must begin with 0x or 0X. ABCDEF are ok. abcdef are also ok.

Recommendations:

1. For binary, use hex. (after some practice, you'll see binary) You can use our helper as well (more info later).
2. Avoid octal, unless there is a very good reason to use it.
3. For hex, begin with 0x (Easier to see than 0X)

01.1 Intro to logic and numbering systems

Grouping:

1 binary digit = a **bit** (**B**inary digi**T**)

8 binary digits = a **byte** (represented by 2 hex digits)

There are others: **half-word**, **word**, **double-word**, but no standard

Sizing	Decimal	Approx	Hex	Conventional Name	IEC binary prefix name
2^{10}	1,024	a thousand	0x400	Kilo (10^3)	Kibi (Ki)
2^{20}	1,048,576	a million	0x100000	Mega (10^6)	Mebi (Mi)
2^{30}	1,073,741,824	a billion	0x40000000	Giga (10^9)	Gibi (Gi)
2^{40}	1,099,511,627,776	a trillion	0x100000000000	Tera (10^{12})	Tebi (Ti)
2^{50}	1,125,899,906,842,624	thousand trillion	0x400000000000000	Peta (10^{15})	Pebi (Pi)

01.1 Intro to logic and numbering systems

- Smallest “addressable” unit in most computers today is a byte (8 bits)
- We want to represent numbers in a byte. $\rightarrow \text{-----}_2$
 - Use positional number system

1. Unsigned integer in a byte (8 bits) $\rightarrow \text{-----}_2$

Minimum value is $00000000_2 = 0x00 = 0$

Next value is $00000001_2 = 0x01 = 1$

...

Almost largest value is $11111110_2 = 0xFE = 254$

Largest value is $11111111_2 = 0xFF = 255$

01.1 Intro to logic and numbering systems

1. Unsigned integer

In 2 bits, we can represent $\{00_2, 01_2, 10_2, 11_2\}$ or $[0, 3]$ in steps of 1.

In 4 bits, we can represent $[0, 2^4-1]$ or $[0, 15]$ in steps of 1.

In a byte (8 bits), we can represent $[0, 2^8-1]$ or $[0, 255]$ in steps of 1.

In 2 bytes(16 bits), we can represent $[0, 2^{16}-1]$ or $[0, 65535]$ in steps of 1.

In **N** bits, we can represent what number ranges?

But we also want negative numbers! What do we do?

01.1 Intro to logic and numbering systems

Negative number representation:

Used to be so many standards, but only two remain widely used.

1. 2's complement number
2. Signed magnitude number

Our class's little CPU can only use 2's complement number.



01 Intro to logic and numbering systems

1. Signed Magnitude:

Reserve the most significant bit (MSB) for sign. Use 0 for positive, and 1 for negative

The rest is positional number system

Example1: Given a byte (8 bits),

$$\begin{aligned} 122_{10} &= 111_1010_2 \\ +122_{10} &= 0_111_1010_2 \\ -122_{10} &= 1_111_1010_2 \end{aligned}$$

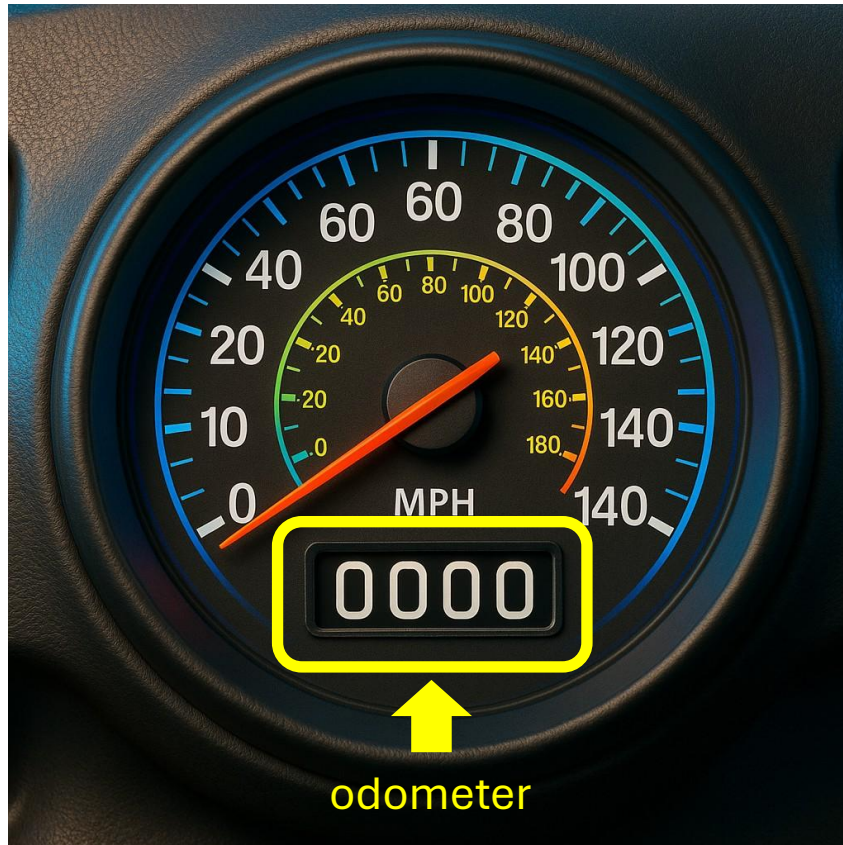
leftmost bit

rightmost bit is called least significant bit (LSB)

sign size (magnitude)

01 Intro to logic and numbering systems

2. 2's complement number: imagine a car's odometer in binary



when you hit “reset”, the meter reads

0000_2 (0)

- drive forward 1km, it reads

0001_2 (+1)

- drive forward another km, it reads

0010_2 (+2)

- drive forward another km, it reads

0011_2 (+3)

- drive forward another km, it reads

0100_2 (+4)

For the negative number part:

now hit “reset”, so the meter reads

0000_2 (0)

- drive backward 1km, it reads

1111_2 (-1)

- drive backward another km, it reads

1110_2 (-2)

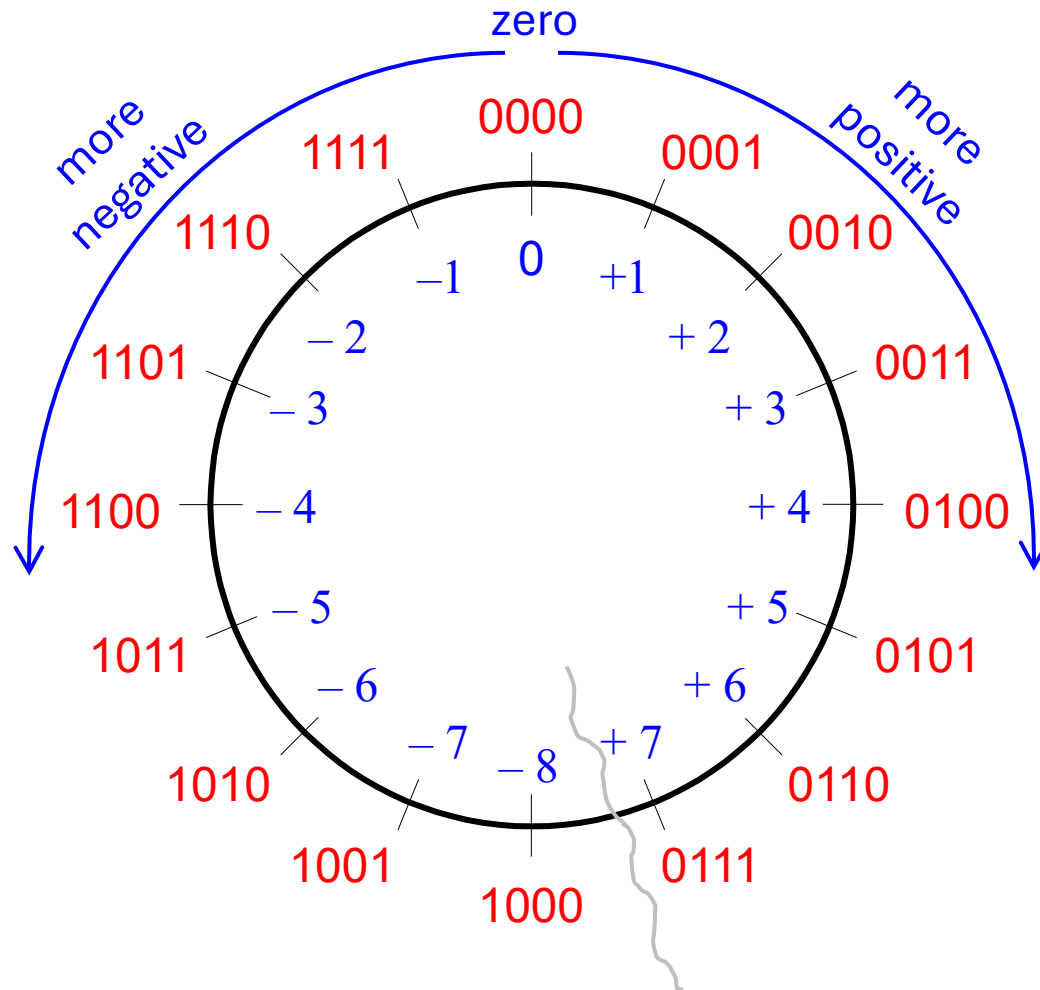
- drive backward another km, it reads

1101_2 (-3)

- drive backward another km, it reads

1100_2 (-4)

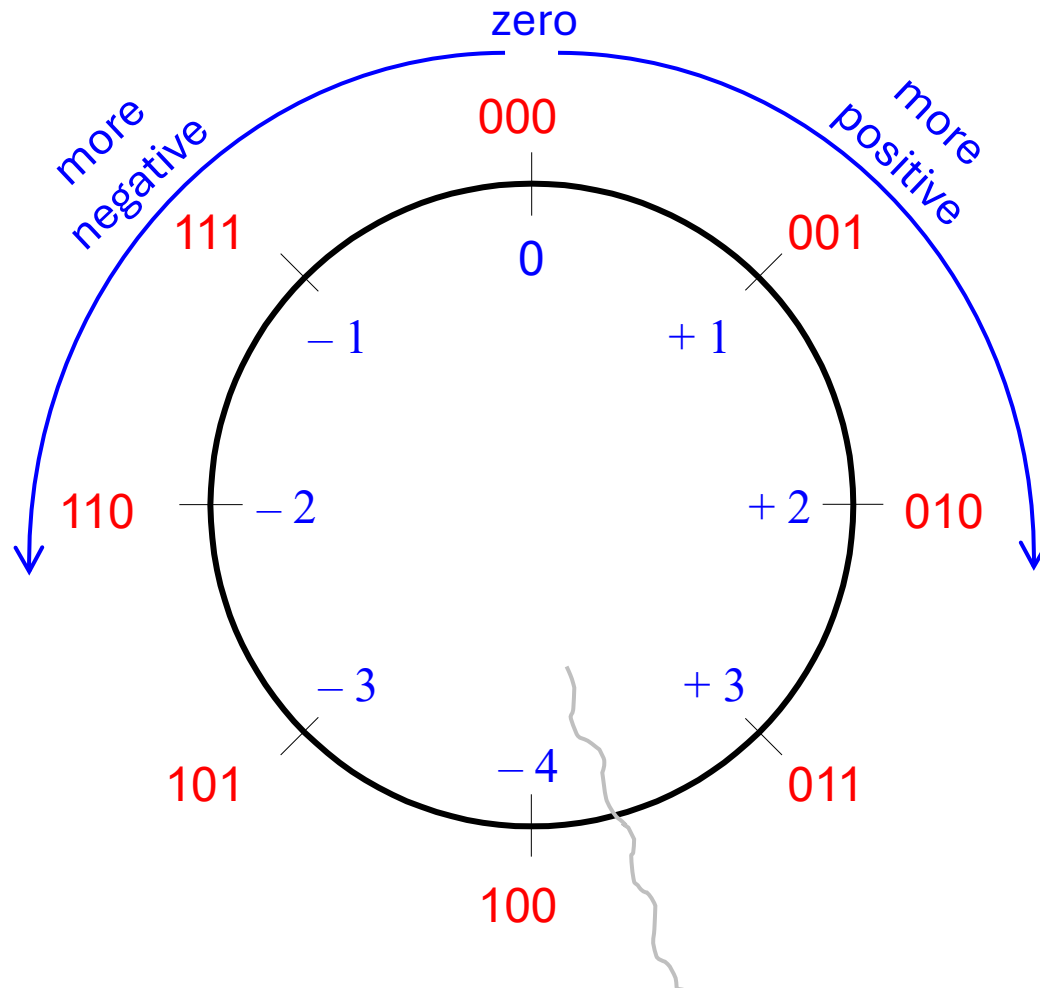
01 Intro to logic and numbering systems



4-bit 2's complement “code” vs “value”

- no longer a positional number system
- with 4-bit, we have $2^4 = 16$ **different codes (in red)** and we map these 16 codes to **16 different values (blue)**
- we still want MSB to be the sign bit, with 0 = positive and 1 = negative
- so, the largest positive is **0111 (+7)**
- the “largest magnitude” negative number is **1000 (-8)**
- so, 4-bit 2's complement cover values $[-8, 7]$ in steps of 1.

01 Intro to logic and numbering systems



2's complement “code” vs “value”

- 3-bit 2's complement cover values $[-4, 3]$
- 4-bit 2's complement cover values $[-8, 7]$
- 5-bit 2's complement cover values $[-16, 15]$
- 8-bit 2's complement cover values $[-128, 127]$
- n -bit 2's complement cover values?

Why we use 2's complement:

- Easy to build add/subtract logic (later course)
- Positive numbers match positional number system
- Overflow from addition can be detected easily

number too big to be represented in original size



01 Intro to logic and numbering systems

How to invert (multiply by -1) 2's complement number?

Convert 001100 (positive) to negative: Invert every bit then add 1

001_100 → this is $4+8 = +12$

110_011 (inversion of 001_100)
+1

1 1 ← carry

110_100 DONE → This is code for value -12 in 2's complement

Convert negative to positive: Invert every bit then add as well

100_101 → This is a negative number because MSB is 1, but what's its value?

011_010 (inversion of 100_101)

+1

011_011 → This is $16+8+2+1 = 27$. So, 100_101 is code for value -27

01 Intro to logic and numbering systems

What else do we need besides numbers?

We need scripts: A-Za-z!@#\$%^&*() ฟหกด愛してます اضطرابات بدون के॥ 이팝პოპულარული

And emojis: 🚀 🌌 💡 📁 🖋️ 📅 ✅ 100 🌟

ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) Codes

- The original standard. The initial, **7-bit** character encoding standard (128 characters) code 0x00-0x7F
- Covers English letters, digits, basic punctuation, (and control characters).

Unicode - new worldwide standard, set by Apple/Xerox. First version appears in 1991.

- **Backward Compatible:** The first 128 characters are identical to ASCII.
- Uses a unique number (e.g., U+0041) for every character. May require > 1-byte to represent
- Supports all scripts, technical symbols, and **emojis**.
- That's why Windows / MacOS / Linux / Android / iOS can display all languages and same emojis. 😊

ASCII Table (partial)

Available in cheat-sheet

- Remember: Computers use bytes (8-bits) so codes run from 0-255 (0x00 - 0xFF), but not all values are “printable”
- Note that codes 0-31 (0x00 – 0x1F) and codes 127-255 (0x7F – 0xFF) are not printable characters
- Code 127 (0x7F) is used for *backspace* (⏮) and is not printable.
- Code 0x00 is called **NUL** (nothing) and is special. It is used to “terminate” (signify the end) of a continuous block of symbols.

We will see 0x00 (NUL)’s importance later

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
space	32	0x20	@	64	0x40	`	96	0x60
!	33	0x21	A	65	0x41	a	97	0x61
"	34	0x22	B	66	0x42	b	98	0x62
#	35	0x23	C	67	0x43	c	99	0x63
\$	36	0x24	D	68	0x44	d	100	0x64
%	37	0x25	E	69	0x45	e	101	0x65
&	38	0x26	F	70	0x46	f	102	0x66
'	39	0x27	G	71	0x47	g	103	0x67
(40	0x28	H	72	0x48	h	104	0x68
)	41	0x29	I	73	0x49	i	105	0x69
*	42	0x2A	J	74	0x4A	j	106	0x6A
+	43	0x2B	K	75	0x4B	k	107	0x6B
,	44	0x2C	L	76	0x4C	l	108	0x6C
-	45	0x2D	M	77	0x4D	m	109	0x6D
.	46	0x2E	N	78	0x4E	n	110	0x6E
/	47	0x2F	O	79	0x4F	o	111	0x6F
0	48	0x30	P	80	0x50	p	112	0x70
1	49	0x31	Q	81	0x51	q	113	0x71
2	50	0x32	R	82	0x52	r	114	0x72
3	51	0x33	S	83	0x53	s	115	0x73
4	52	0x34	T	84	0x54	t	116	0x74
5	53	0x35	U	85	0x55	u	117	0x75
6	54	0x36	V	86	0x56	v	118	0x76
7	55	0x37	W	87	0x57	w	119	0x77
8	56	0x38	X	88	0x58	x	120	0x78
9	57	0x39	Y	89	0x59	y	121	0x79
:	58	0x3A	Z	90	0x5A	z	122	0x7A
;	59	0x3B	[91	0x5B	{	123	0x7B
<	60	0x3C	\	92	0x5C		124	0x7C
=	61	0x3D]	93	0x5D	}	125	0x7D
>	62	0x3E	^	94	0x5E	~	126	0x7E
?	63	0x3F	_	95	0x5F	DEL	127	0x7F

01 The Digital World

01.1 Logic and Numbers

01.2 Computer Architecture

01.2 Intro to computer architecture

A very basic modern computer architecture

- Turing machine
- von Neumann machine
- Input/Output (I/O)
- CPU
 - CPU internals: ALU, PC, stack pointer, status, and data registers
 - CPU externals: address bus, and data bus
 - endianness of multi-byte storage
- Real CPU: Z80



02 Intro to computer architecture

2 Great Heroes of Modern Computers: Alan Turing



wikipedia

1936 – invents the abstract “Turing machine,” defining the limits of computation – what can and cannot be computed.

1940-45 – At Bletchley Park, builds the “bombe,” cracking Enigma and shortening WWII. Featured in the movie *The Imitation Game* (2014)

1946-50 – Designs ACE (electronic computer), proposes the Turing Test, and codes the first computer chess.

The Turing Award is considered “the Nobel Prize of Computing”

Famous recipients:

1966 – Alan Perlis – building compilers

1983 – Dennis Ritchie – co inventor of Unix and C

2004 – Vint Cerf & Bob Kahn – inventor of TCP/IP (internet)

2015 – Whitfield Diffie & Martin Hellman – asymmetric cryptography



02 Intro to computer architecture

2 Great Heroes of Modern Computers: John von Neumann



1925-30 – does foundation work for quantum mechanics and game theory.

1944 – Co-creates game theory, shaping economics and Cold-War strategy.

1945 – Drafts the stored-program “von Neumann architecture” still used in every computer.

1945-52 – Builds the IAS machine at Princeton, first electronic stored-program computer.

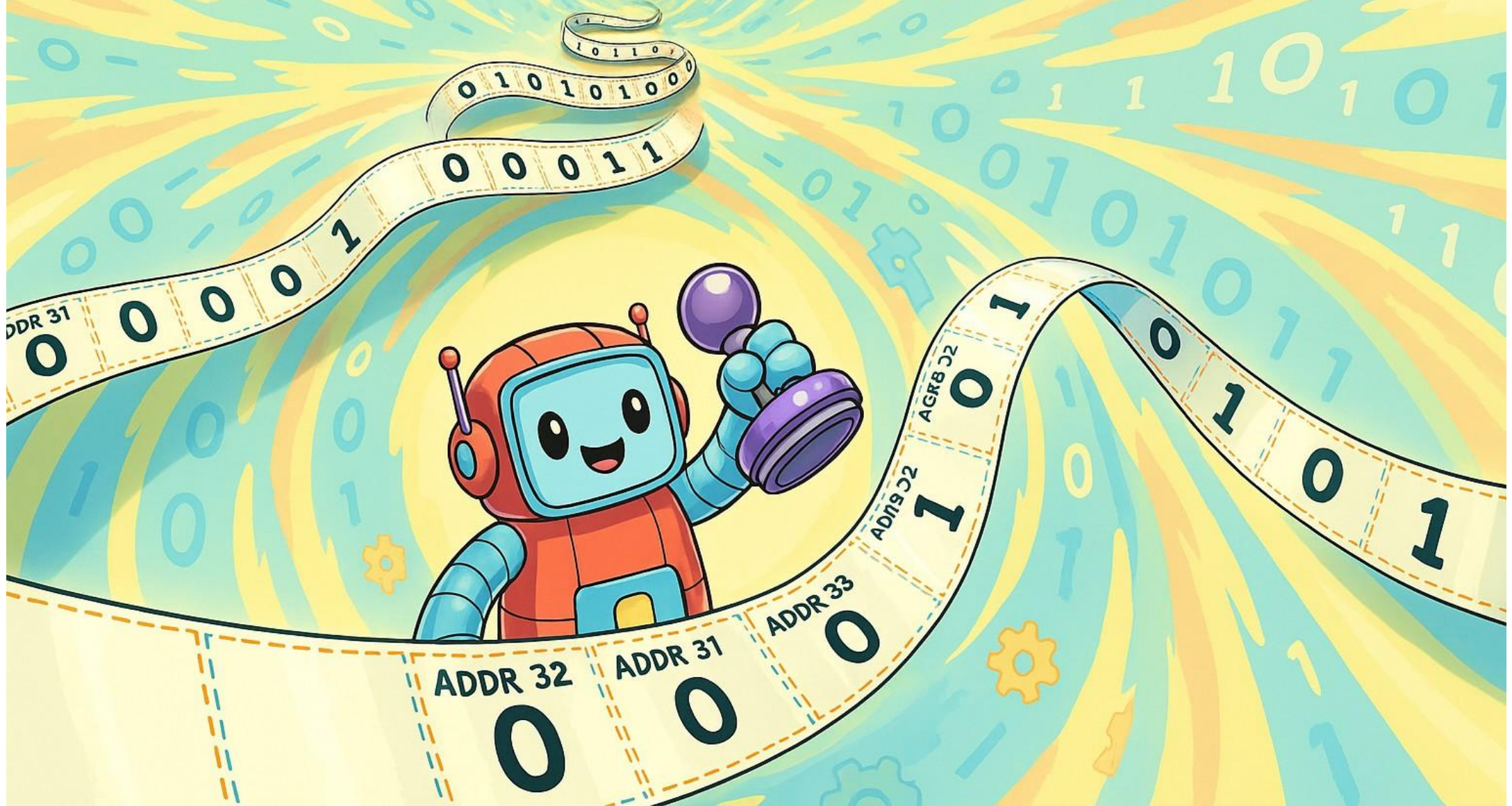
1945-55 – invents Monte-Carlo simulation and H-bomb codes, making computing core to science and defense.

The Turing Machine (1936)

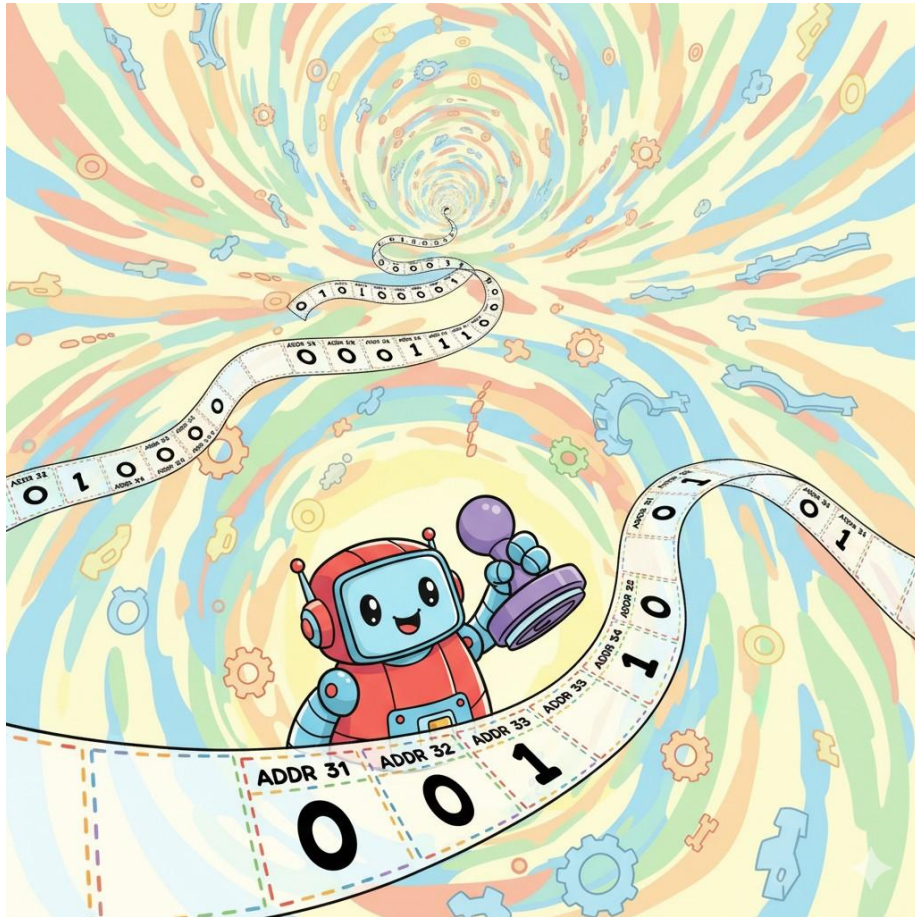
app

comp arch

logic




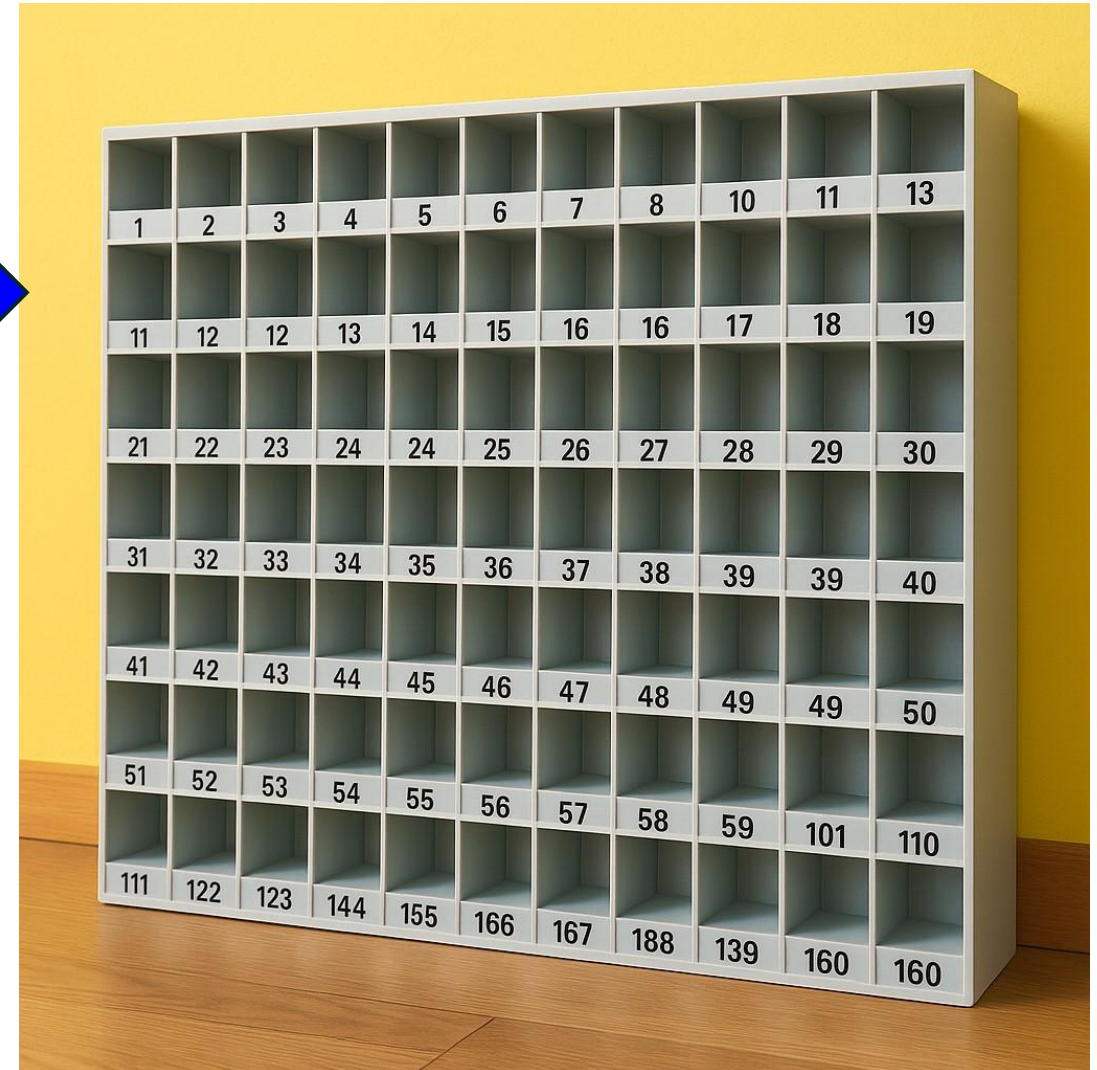
The Turing Machine



- A **theoretical, abstract model** of computation, not physical
- Has an **infinitely long tape** divided into cells, each holding a symbol, and a read/write "head" (shown as robot)
- A read/write "robot" moves along the tape one cell at a time (**sequential access**)
- Robot has a simple memory that stores its current "mode" and a "rulebook" to tell what it should do based on current symbol on the tape. So, it can compute and store a little bit.
- Defines the fundamental limits of what can be computed (**Turing Completeness**)
- Foundation for **computation theory** and complexity.

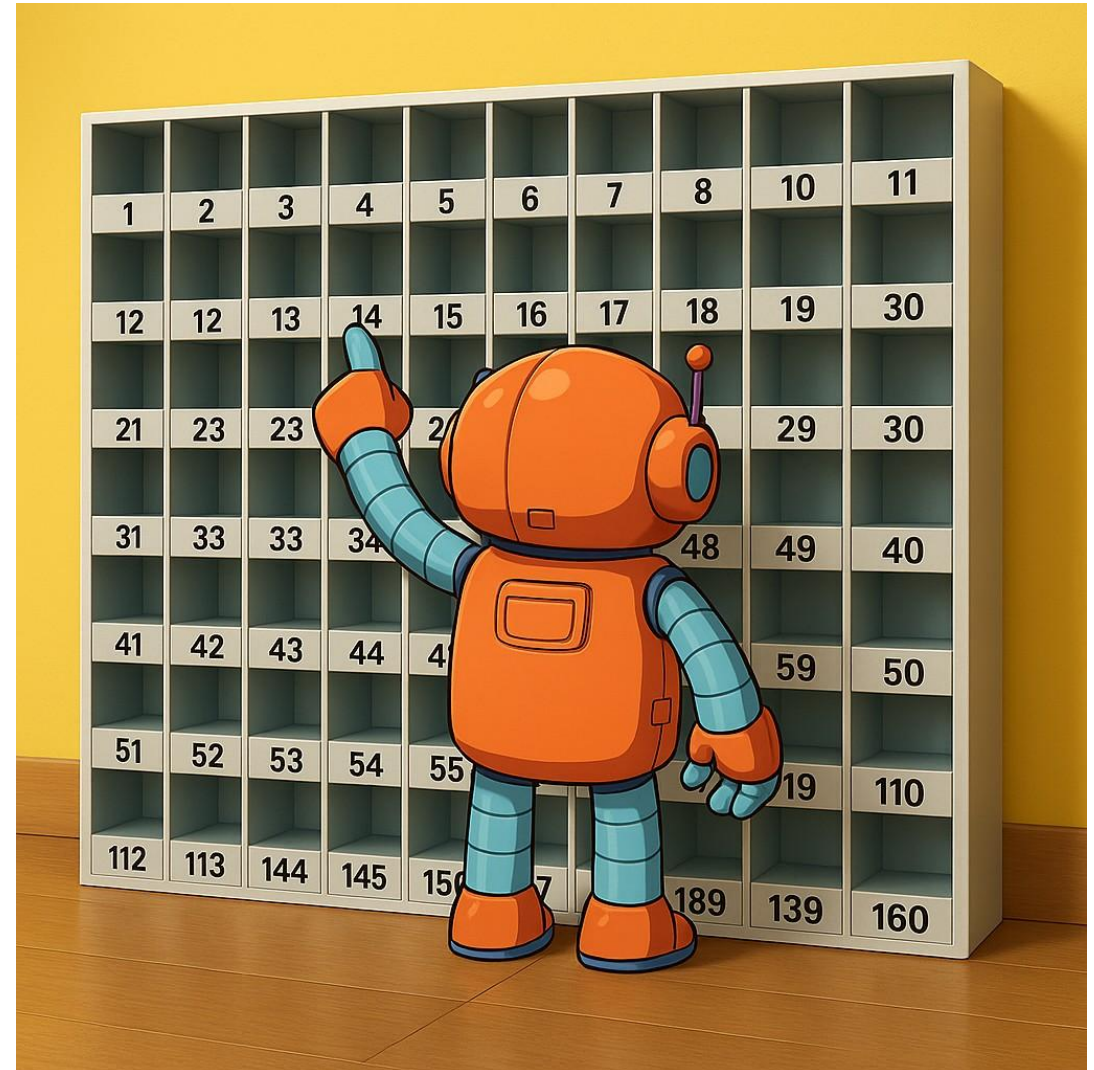
The von Neumann architecture (1945)

- John von Neumann is aware of this “Turing machine”
- An **infinitely long tape** is not physically possible
- But someone invented this: 
(not the physical mailbox, but an electronic one)
- We can use the mailbox to simulate a long tape
- Each mailbox has a unique address
 - By being digital, addresses are written in binary
- Each mailbox can store data
 - By being digital, data are read/written in binary
- Instead of rolling the tape forward/backward, we can access the data in the address we want at will.
This is called **Random Access Memory (RAM)**



The von Neumann architecture

- Now, let that Turing robot stand in front of **RAM**
- The robot can read data from any address
- The robot can write data to any address
- The data read can contain instructions for the robot. These instructions can also tell it to write data to some addresses
- Those instructions are called **programs**
Today we call them by their functionality, like operating-system, apps, web browser, games
- This little robot (remember, it can remember a little bit, and compute a little bit) is called the **Central Processing Unit (CPU)**
- Voila! A modern computer can be built!

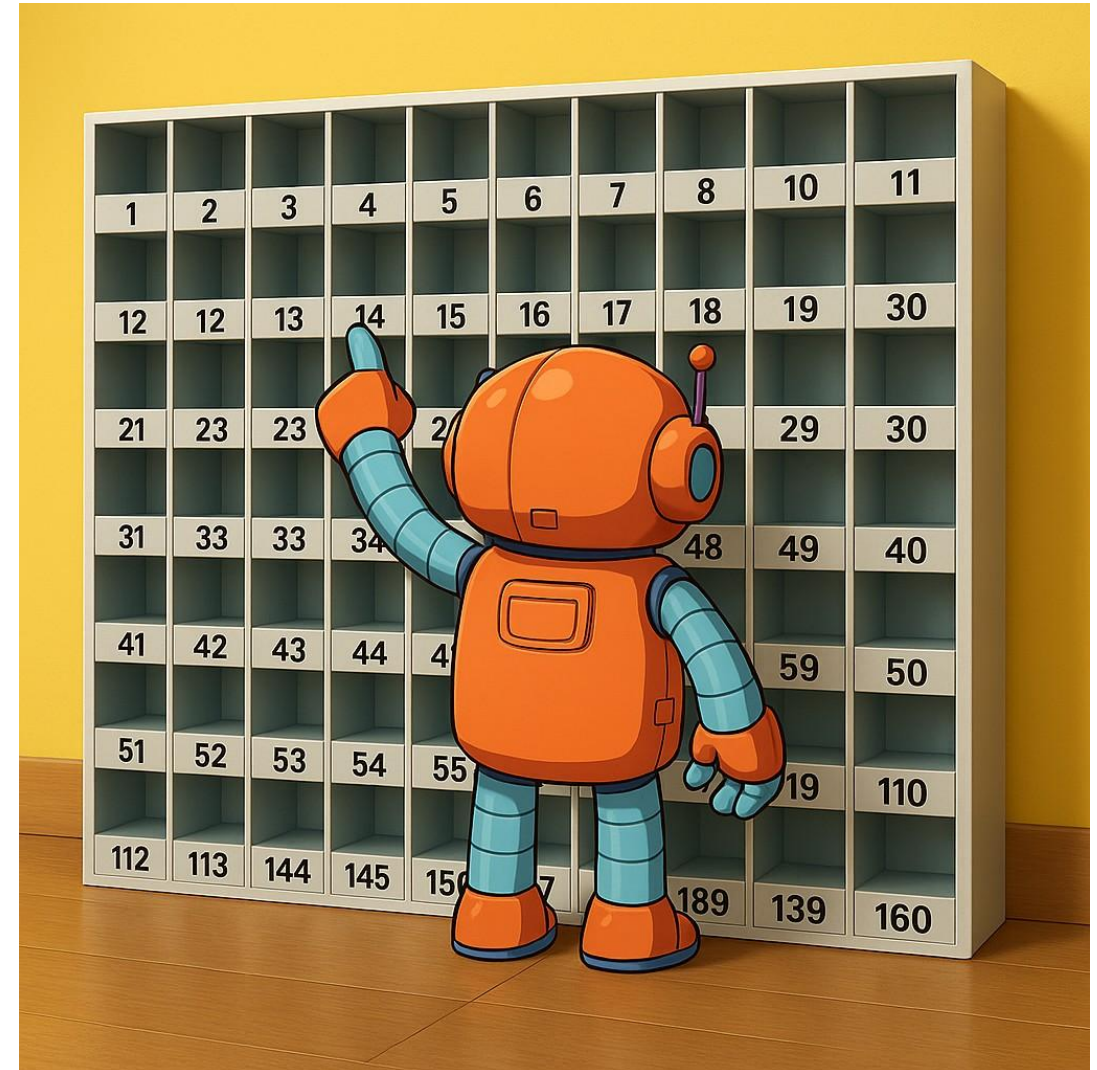


The von Neumann architecture and I/O

Real computers have **Input/Output (I/O)**

How does a computer interact with the world outside?

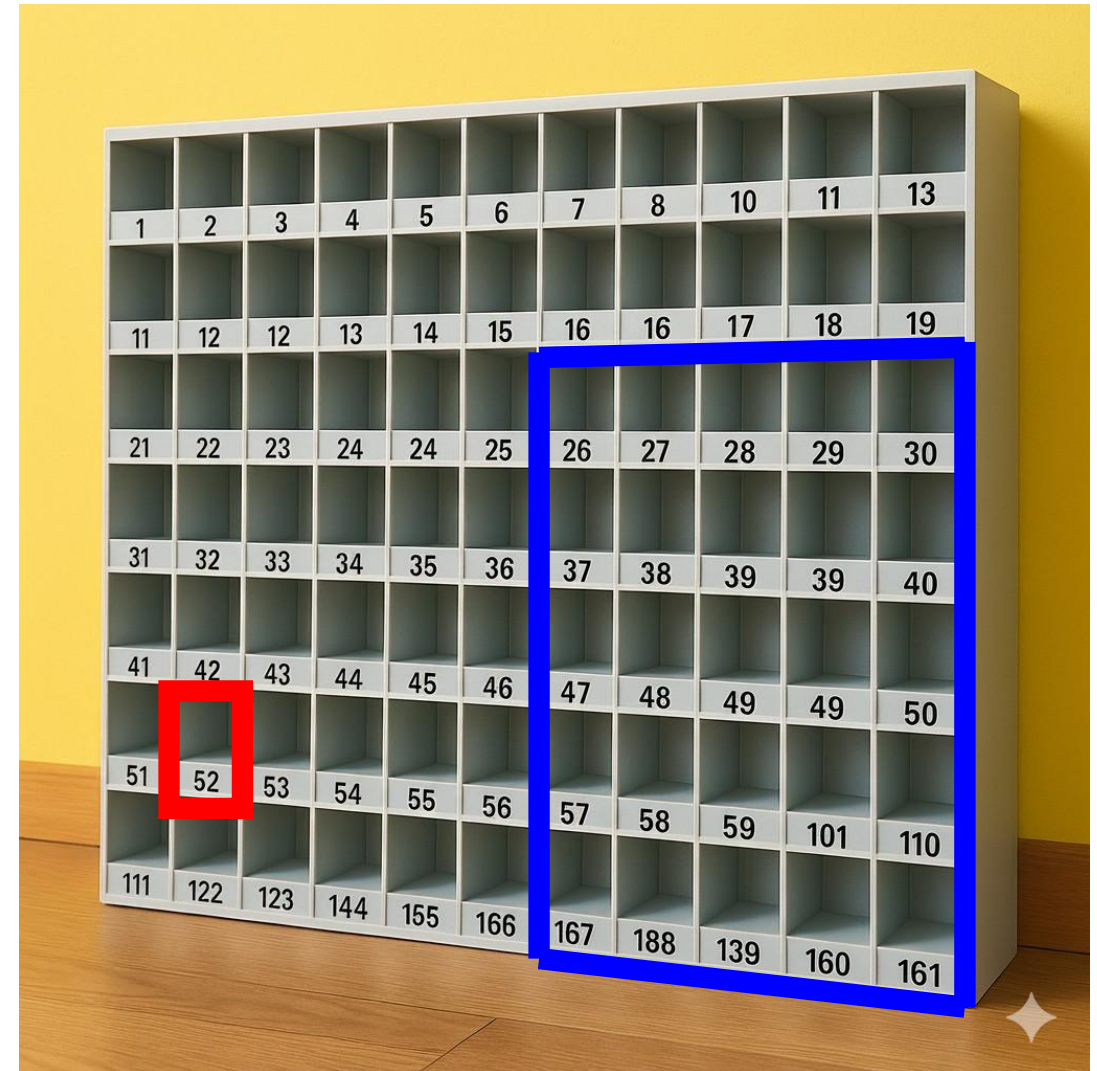
- Most addresses can be written to and read from **ONLY** by the CPU, but...
- Some addresses are **volatile** – data in those addresses can change without the robot's actions. The changes come from outside world. These are **inputs** to the computer.
- Some addresses are **write-only** – the robot can write data to them but cannot read data from them reliably. Some types of (not all) **outputs** are write-only.
- Some addresses cannot be written to, and data in them remain constant. These boxes are called **Read-Only Memory (ROM)**



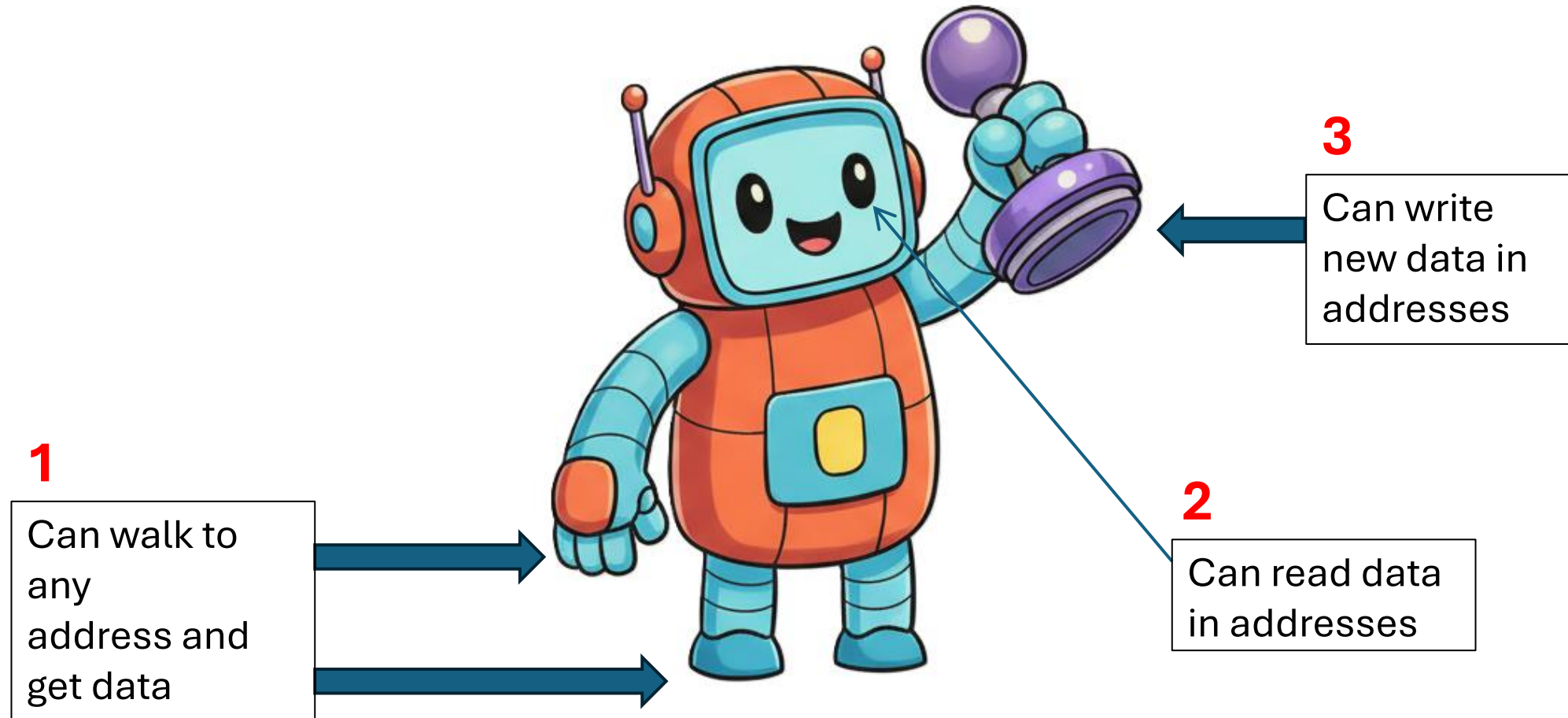
The von Neumann architecture and I/O

Examples of I/O:

- Red box: read from a **keyboard**
every time we read from address 52, we know the code of last key that has been pressed. If no key has been pressed between after the last time we read from the keyboard, we get data 0x00 (ASCII code for NUL).
- Blue boxes: **video frame buffer**
Every data we write to any address within the big deep blue boxes will show up on the screen, and our screen is 5x5 pixels. Address 26 is the top left corner and address 161 is the pixel on bottom right. Convention: 0x00 = black, 0x80 = mid-gray, 0xFF = white

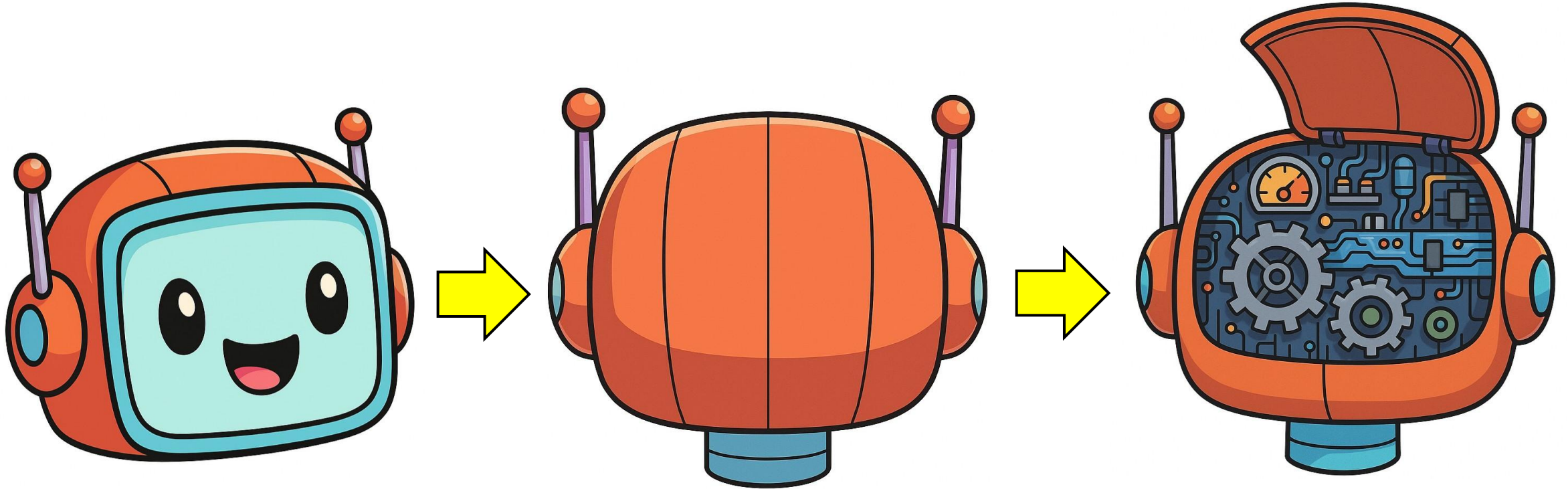


The internals of the robot (CPU)



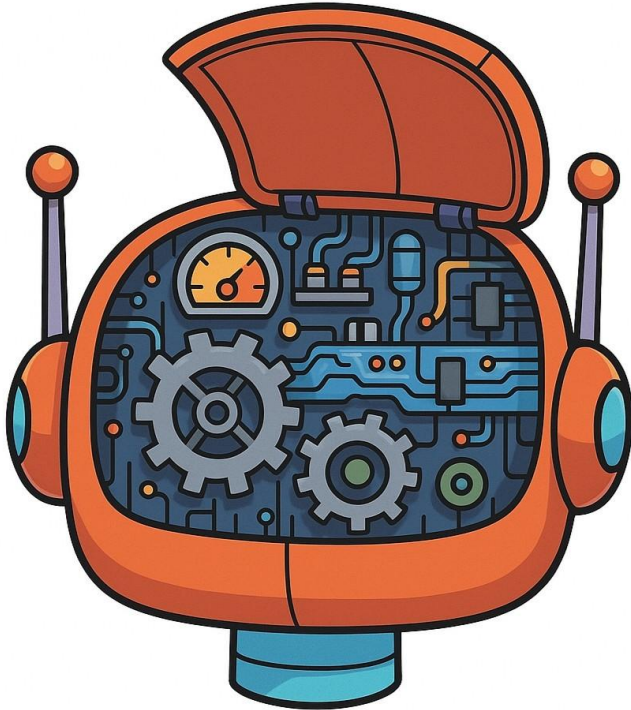
1 2 3 are interfaces to the outside world, known as **system bus**

The internals of the robot (CPU)



Has a little bit of
computing ability
and some memory
- called the **core**

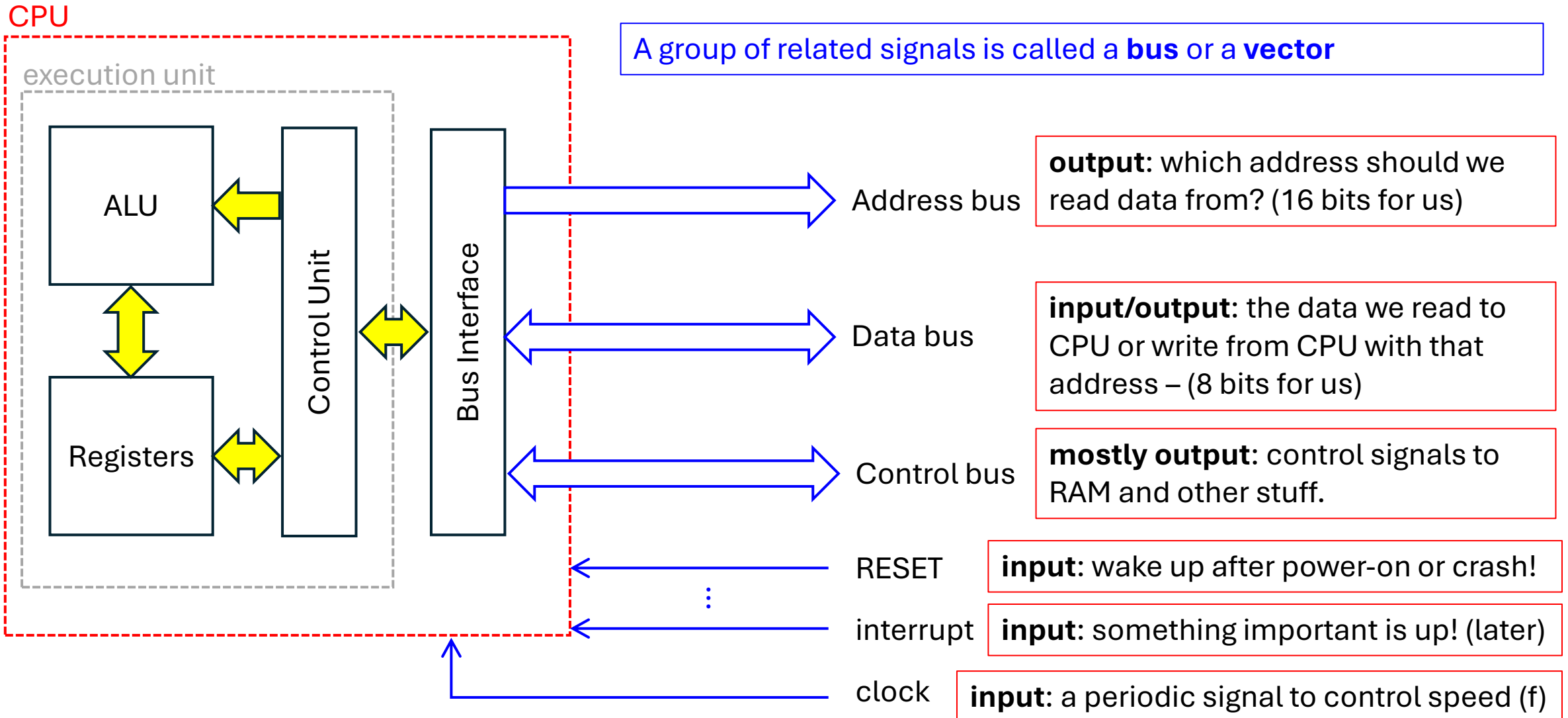
The internals of the robot (CPU)



The **core** has 4 major components

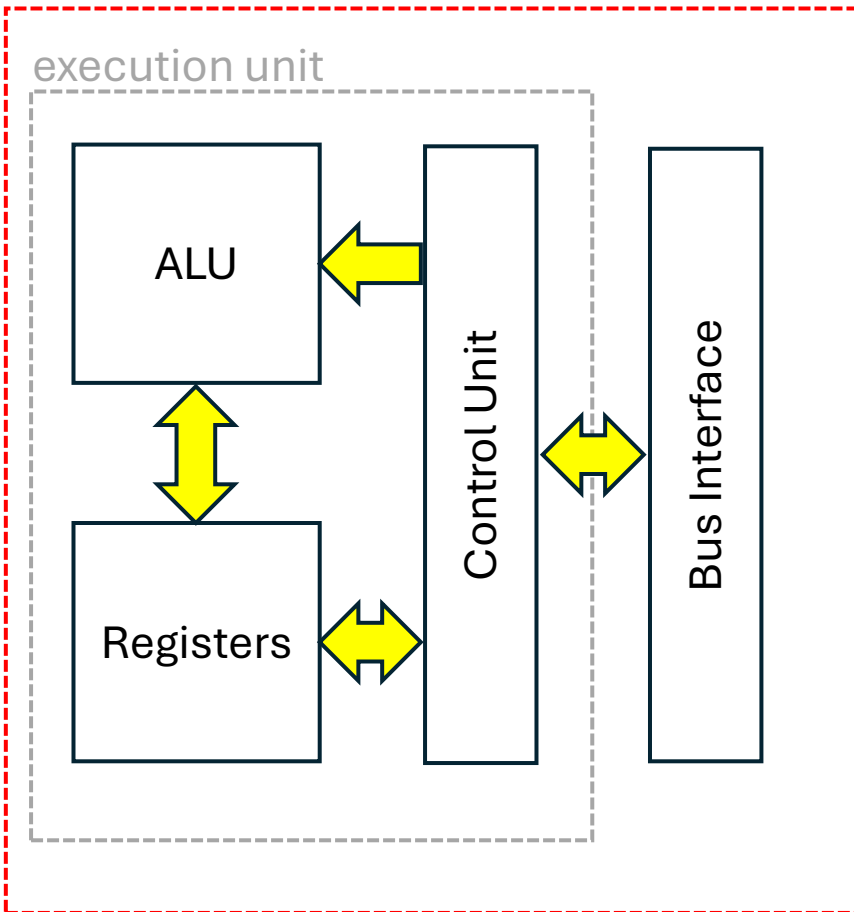
1. **ALU** or Arithmetic and Logic Unit (the “compute” part)
2. **registers** – small scratch memory the core can use
 - **data registers** – used to hold data for the ALU
 - **status registers** – some useful information about previous results
 - **program counter** – what address am I running the program from right now?
3. **control unit** – orchestrate the working of core components
all above are called **execution unit (EU)**
and finally,
4. **bus interface unit** – to communicate with the **system bus**

The internals of the CPU



The internals of the CPU

CPU



What CPU for this course?

- ✗ modern SoC (intel Core, AMD Ryzen, Apple M4)?
 - very complex. will take months just to understand the basics
 - you won't be able to DIY these.

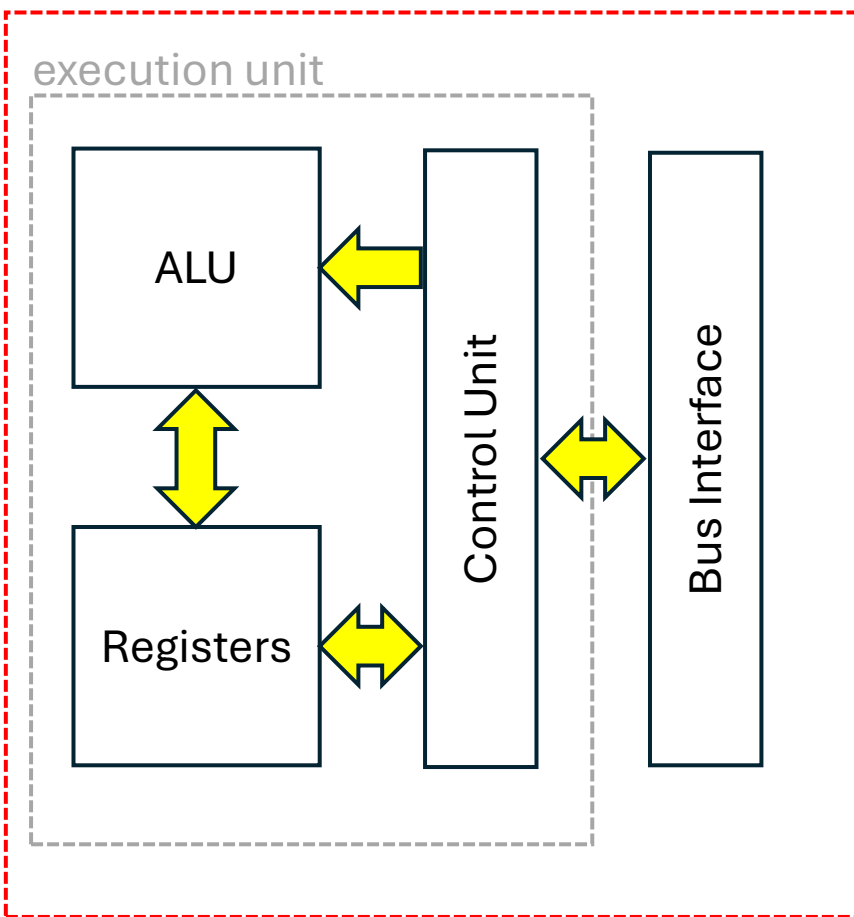
- ✓ Zilog Z80 CPU (1976)
 - easy enough to understand and use by students of this level
 - one of the most popular CPUs of early 1980s
 - simple hardware. only needs +5V to run
 - superset of Intel 8080
 - used heavily in PCs and arcade games
 - easy for C compiler to work with
 - you can still DIY it today

Z80 bus interface characteristics:

- **16-bit address bus** A15...A0, so address goes from 0x0000-0xFFFF
- **8-bit data bus** (1-byte) D7...D0, so data in an address has values in range of 0x00-0xFF

The internals of the CPU

CPU



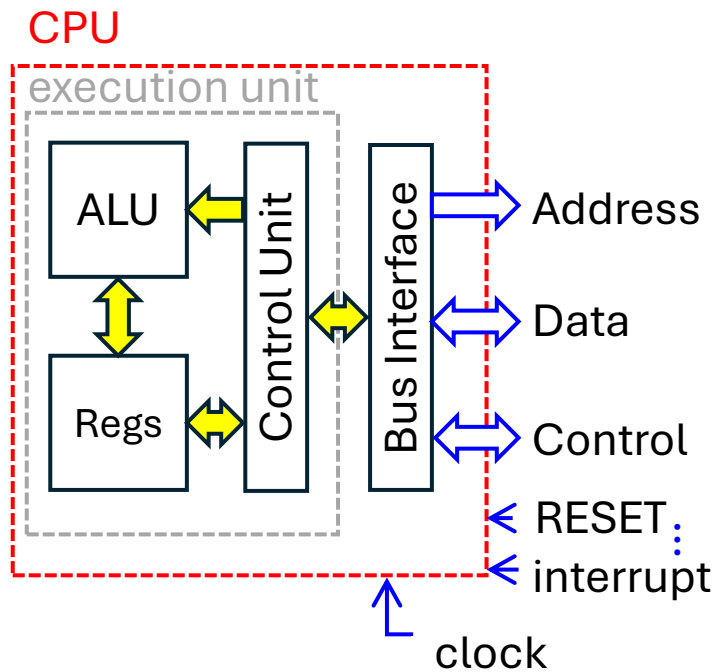
- **16-bit address bus** A15...A0, so address goes from 0x0000-0xFFFF
 $2^{16} = 2^6 \cdot 2^{10} = 64K = 65,536$ addresses
- **8-bit data bus** (1-byte) D7...D0, so data in an address has values in range of 0x00-0xFF → 8-bit = 1 byte
- **Z80 CPU supports 64k Bytes of memory**

Important Registers inside Z80

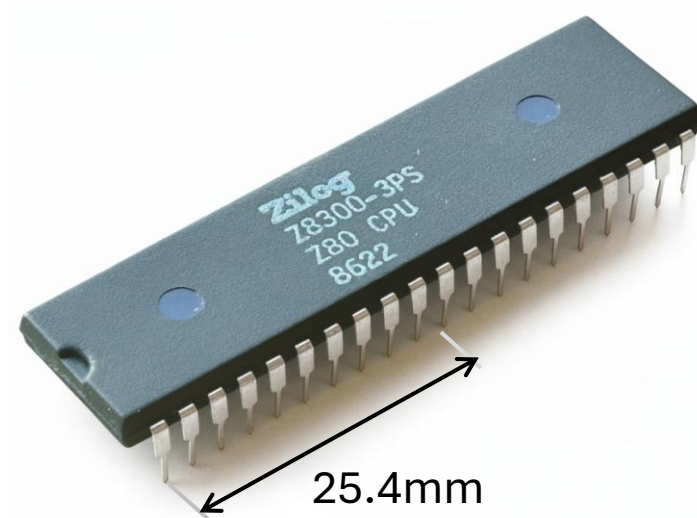
- **PC** (Program Counter) = 16-bit = Which address is the next command?
- **SP** (Stack Pointer) = 16-bit = which address is current the top of the stack? (more later)
- **Data registers** – 8-bit, but can be “paired” to be 16-bit
 - A, B, C, D, E, H, L – each one is 8-bit, but can be “joined” to store 16-bit data, like BC, DE, HL (left: upper 8, right: lower 8)
- **F** (Flags Register) = 8-bit. The info from previous calculation, like 7(MSB):Sign, 6:Zero, 4:Half Carry, 2:Parity/Overflow, 0(LSB):Carry
- **Other Misc Registers: IX, IY, I, R**

The CPU

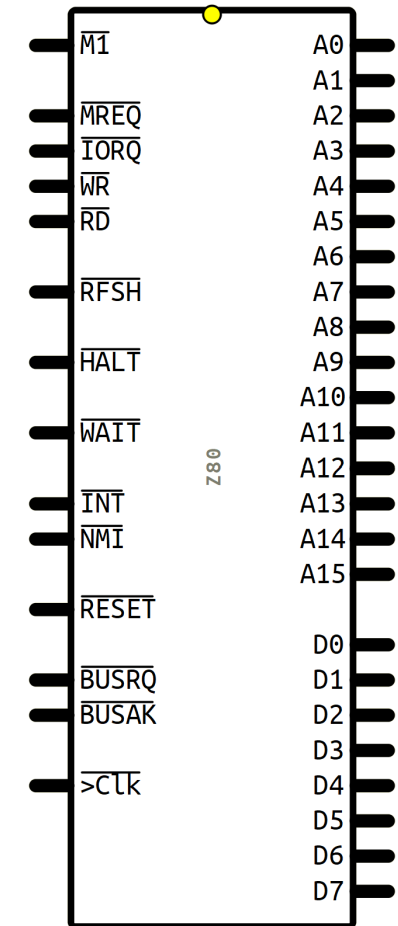
Conceptual



Physical



Logical (Schematics)



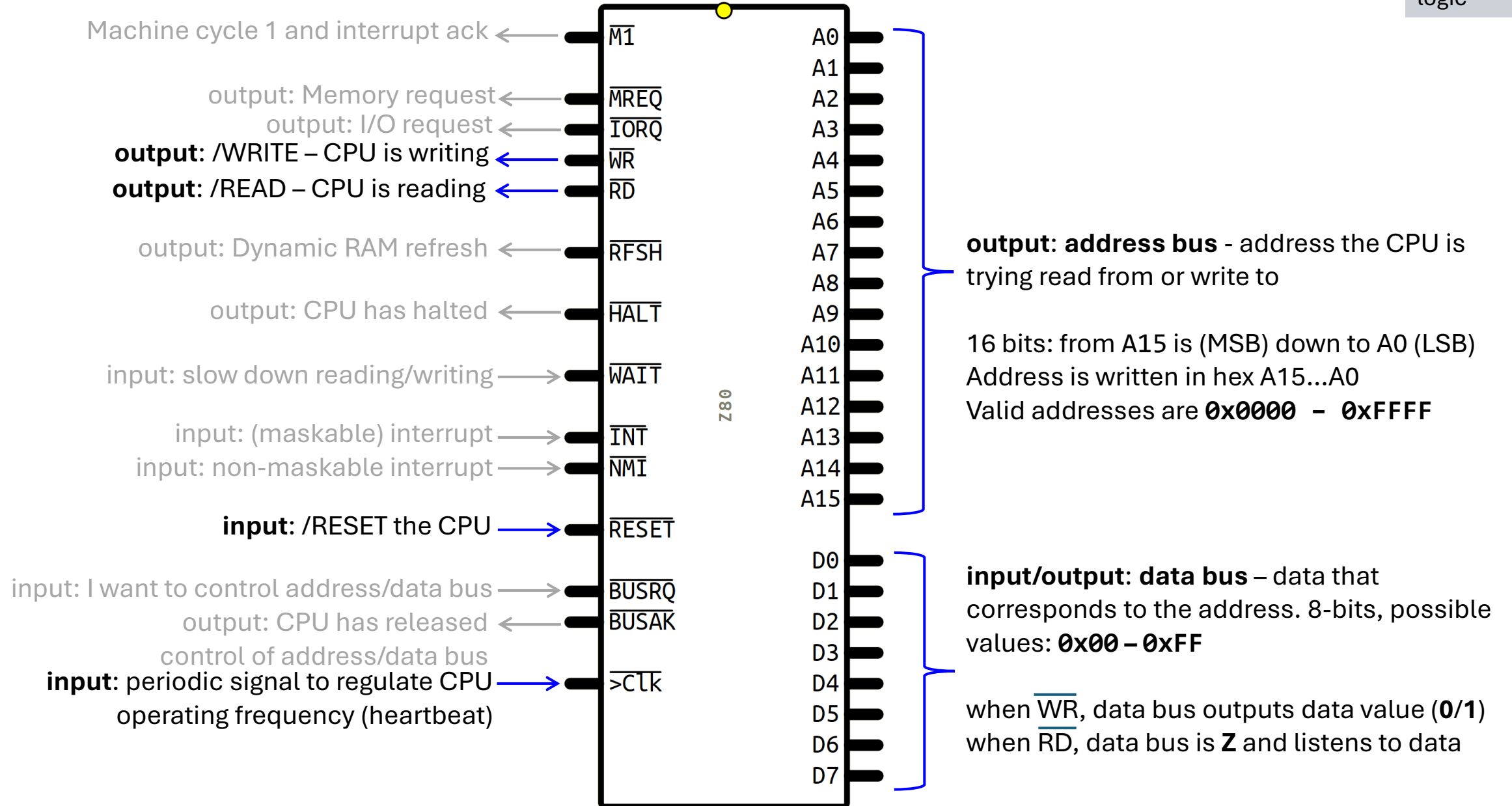
Z80 CPU

app

comp arch

logic

Black / Blue = MUST KNOW
GRAY = DON'T WORRY





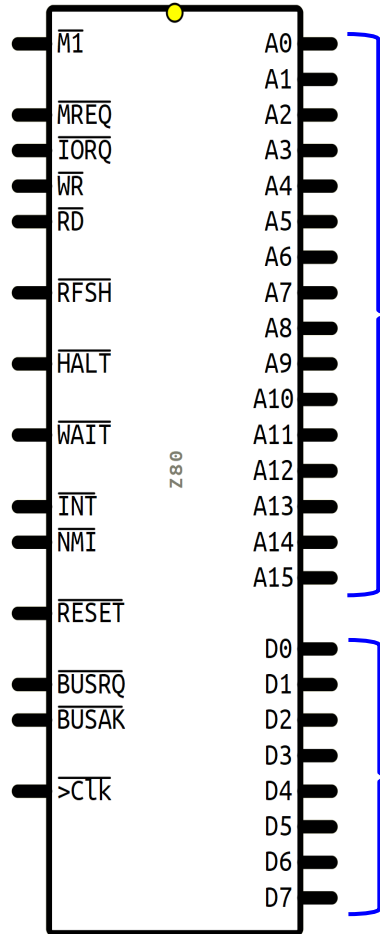
Z80 address/data bus timings

app

comp arch

logic

[From Z80 CPU User Manual, page 10]



output:
address bus
0x0000 –
0xFFFF

input/output:
data bus
0x00 – 0xFF

when you see “fuzzy” or “hatches”
value, it usually means **X** (unknown/
don’t care, but is 0/1)

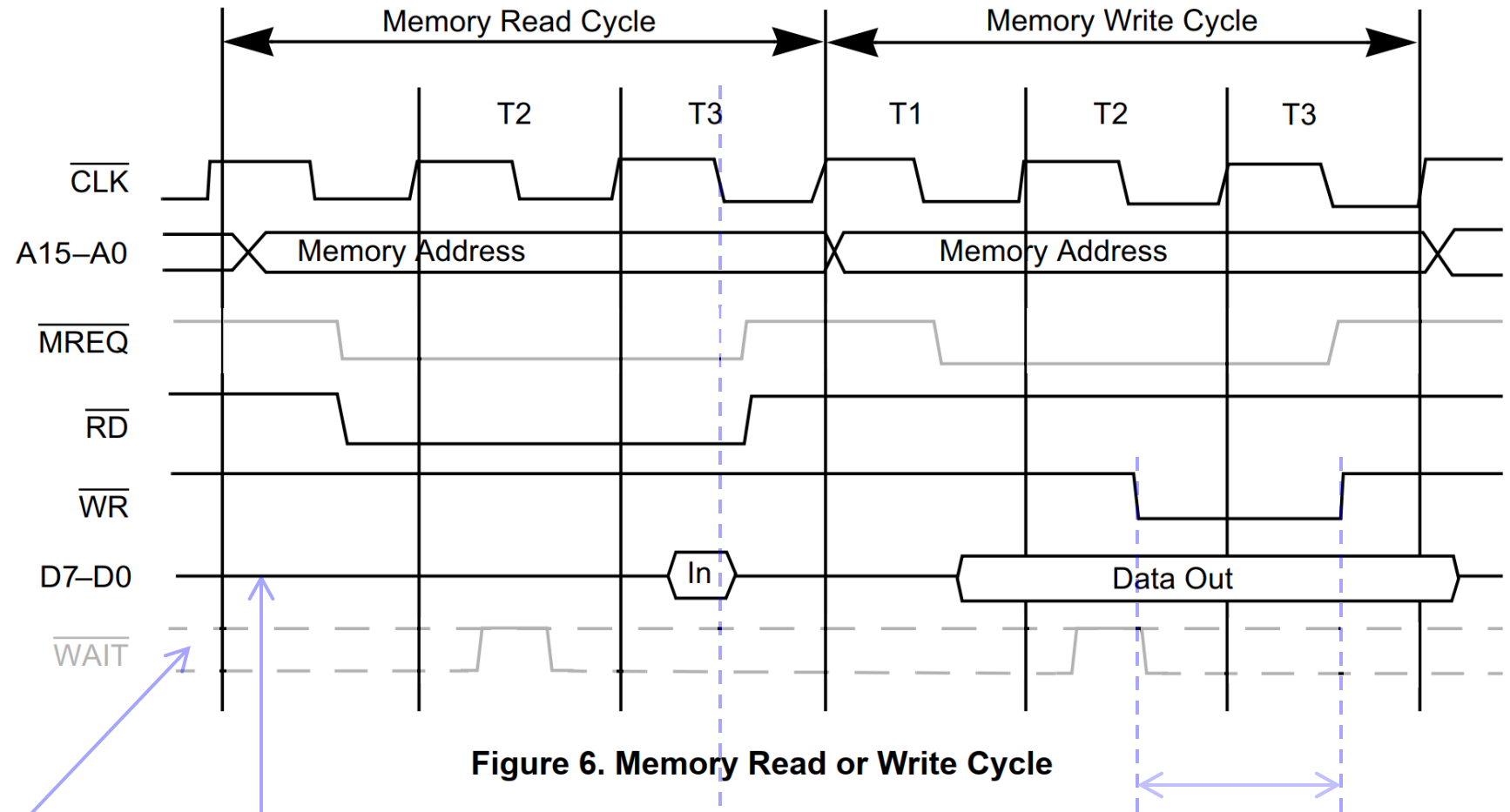


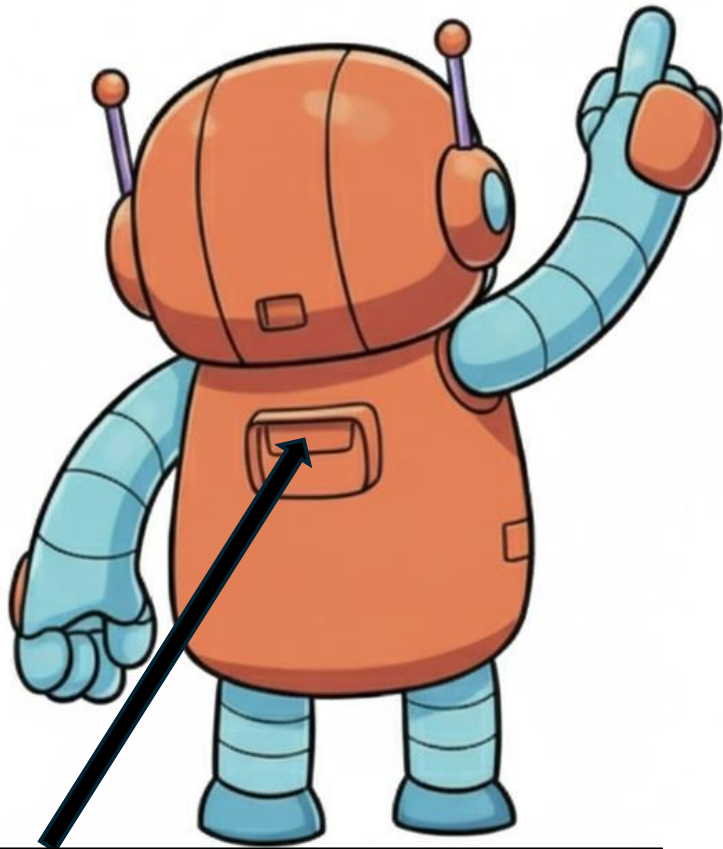
Figure 6. Memory Read or Write Cycle

when you see “in the
middle” value, it
usually means **Z**

data from address is
read by CPU right at
this moment

data to write to the
address is valid
during the entire WR

Z80 interrupts and magic number



There are **emergency** buttons on back of robot. Pushing them makes robot stop running programs and...

That emergency button(s) is called an interrupt.

- 2-types, *maskable* and *non-maskable*
- maskable = there's a command for robot to ignore it
- non maskable = robot **MUST** respond by moving its finger to a "magic address" and read commands there.

Magic number = some constant value made up by engineer/designer. there may or may not be a good reason behind it.

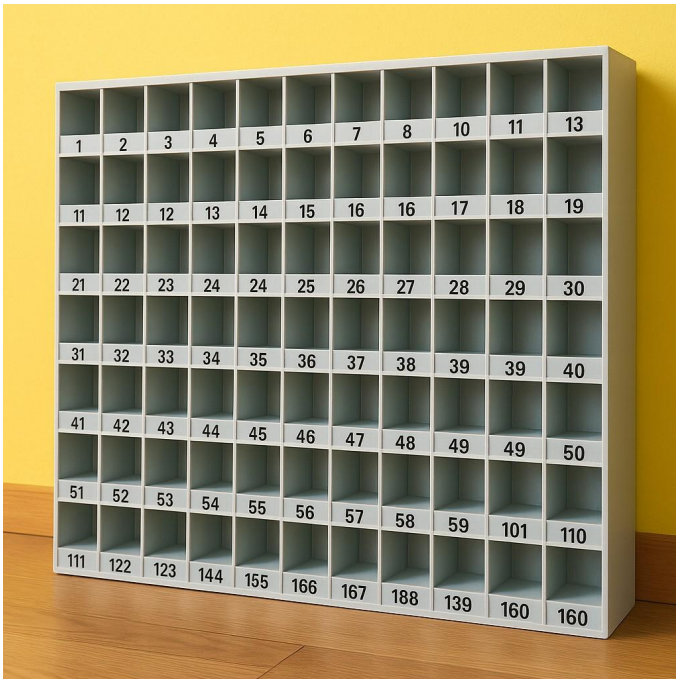
Magic numbers for Z80 CPUs:

- when CPU comes out of reset, program counter is set to $0x0000$ (so, the designer wants CPU to start working from address $0x0000$)
- When CPU encounters an NMI (non-maskable interrupt), it will finish its current instruction, push current PC onto the stack (more later), and will set PC to $0x0066$ and start running the program from that address.

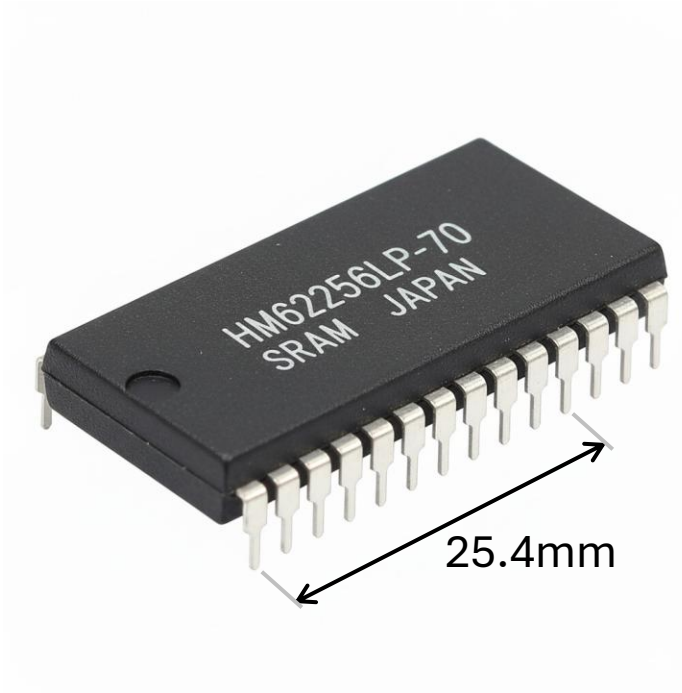
So, $0x0000$ = reset vector, $0x0066$ = NMI vector – both are **magic numbers** – chosen by Z80 designer because they "make sense."

The real random-access memory (RAM)

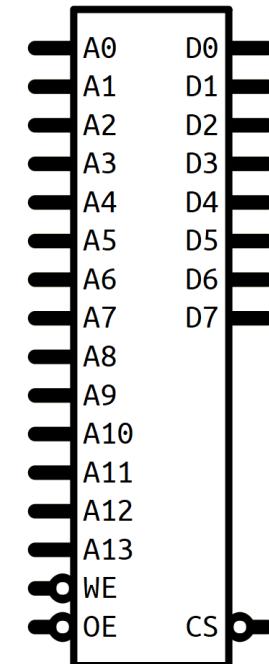
Conceptual



Physical



Logical (Schematics)



The real random-access memory (RAM)

Logical (Schematics)

Address bus (input)

A13 is MSB

A0 is LSB

address is written in hex

A13, A12, ..., A0

So, address range is from
0x0000 to 0x3FFF

Data bus (can be either inputs or outputs)

D7 is MSB - D0 is LSB

data in hex

D7, D6, ..., D0

So, data can take values from
0x00 to 0xFF

○ WE = $\overline{\text{WRITE_ENABLE}}$

○ OE = $\overline{\text{OUTPUT_ENABLE}}$

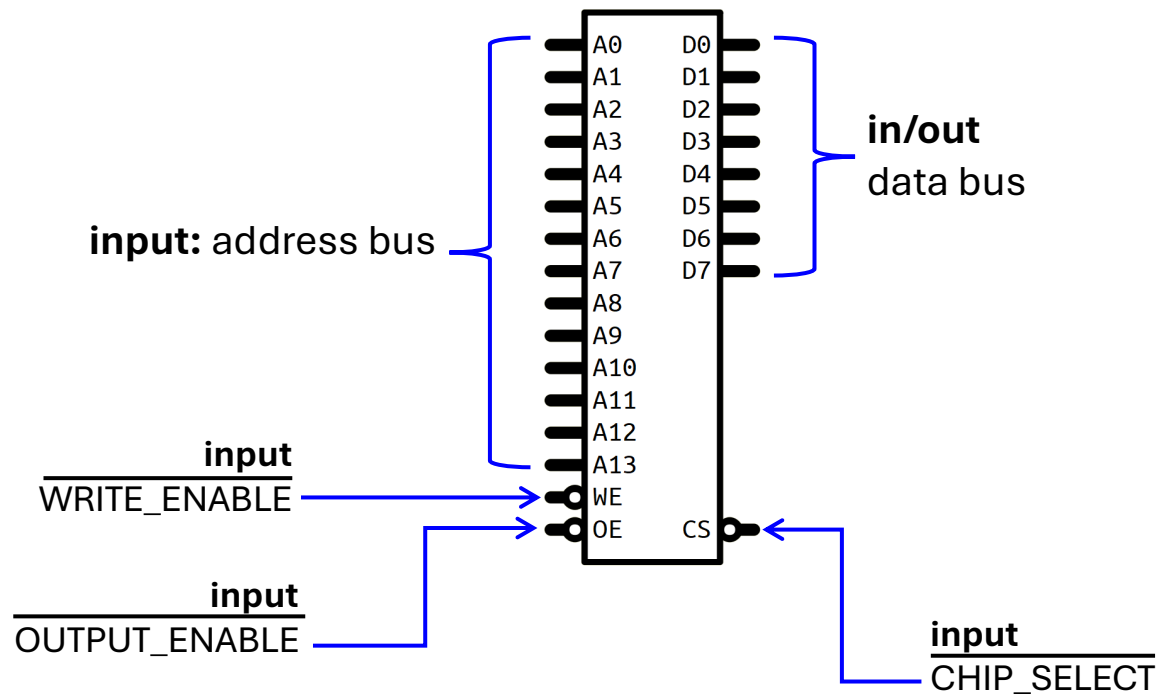
○ CS = $\overline{\text{CHIP_SELECT}}$

/CS /OE /WE
(CS_ OE_ WE_)
are all inputs.
They are control signals
for SRAM behavior.

This type of **RAM** is called **Static RAM (SRAM)**. There's Dynamic RAM (**DRAM**) as well. Some DRAM is also synchronous, with clock input (**SDRAM**). Some SDRAM are Double-edge triggered, aka Double Data Rate. (**DDR-SDRAM**)

The static random-access memory (SRAM)

Logical (Schematics)



RAM control signal truth table

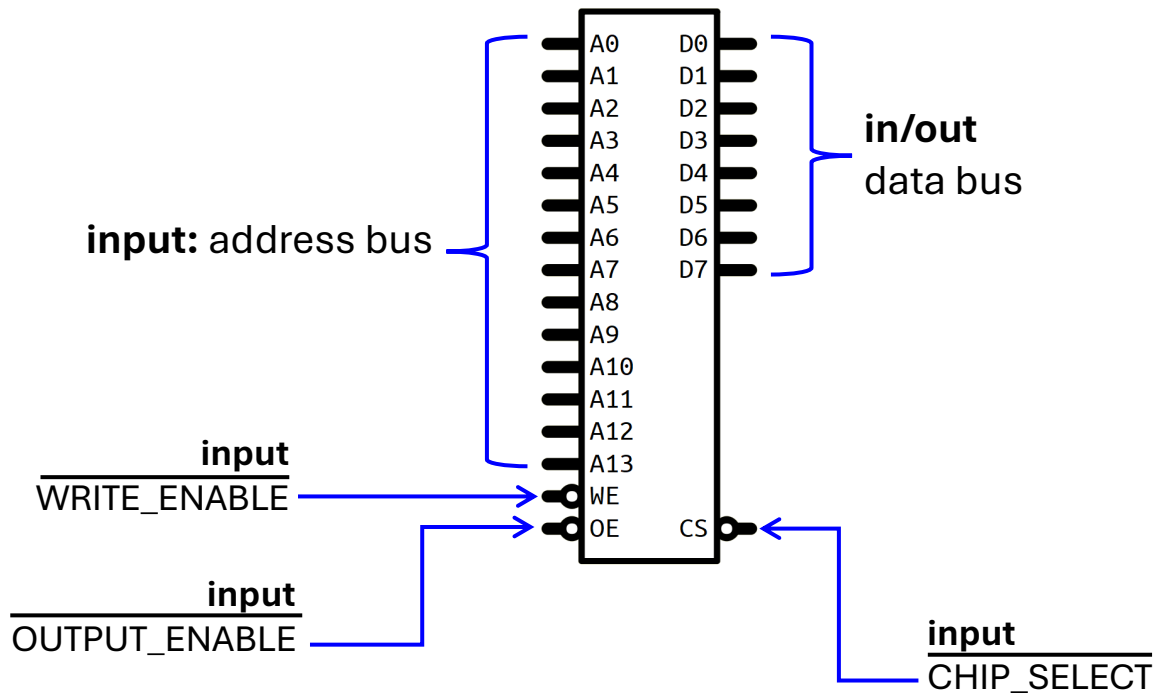
/CS	/OE	/WE	Meaning	D7-D0
1	x	x	chip is deselected	high Z
0	1	1	selected, but not enabling output	high Z
0	0	1	read from an address	data out from address
0	x	0	write to an address	data in to address

The way to read the table above:

1. CS_ (sometimes written as CS_) has the highest priority. must assert it (set it to 0) to use this memory chip
2. OE_ has the 2nd highest priority. Assert CS_ and OE_ makes sure we are reading.
3. WE_ has the least priority. Assert CS_, de-assert OE_, and assert WR_ to write ← why is it done this way?

The static random-access memory (SRAM)

Logical (Schematics)

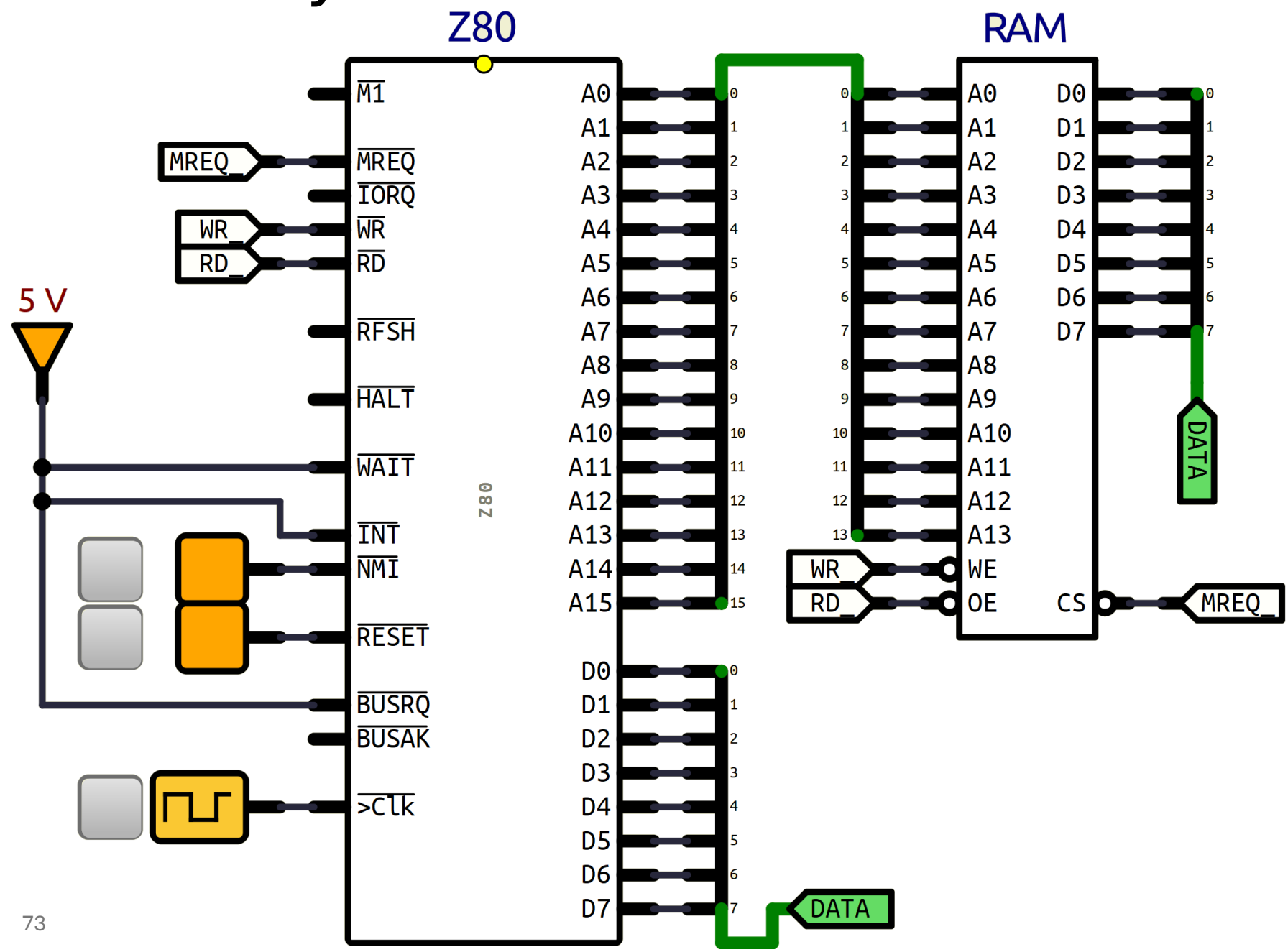


What is the “size” of this SRAM?

- It has 14-bit address bus:
 - A13 is MSB
 - A0 is LSB
- So, it has $2^{14} = 16,384$ addresses, which means it has 16k addresses
- Each address has 8-bit data (8-bit = 1 byte)
- Data runs from D7 (MSB) down to D0 (LSB)
- So, this SRAM has capacity of 16k addresses, and each address is 1 byte
- So, this is a **16kB SRAM**
- More precisely written as 16k x 1-byte, denoting that it has 16k addresses

Real world data: one photo taken by your smart phone (2025) is about 1,000-5,000kB
→ You need 100-300 of these chips to store

our first system



This is called a schematic

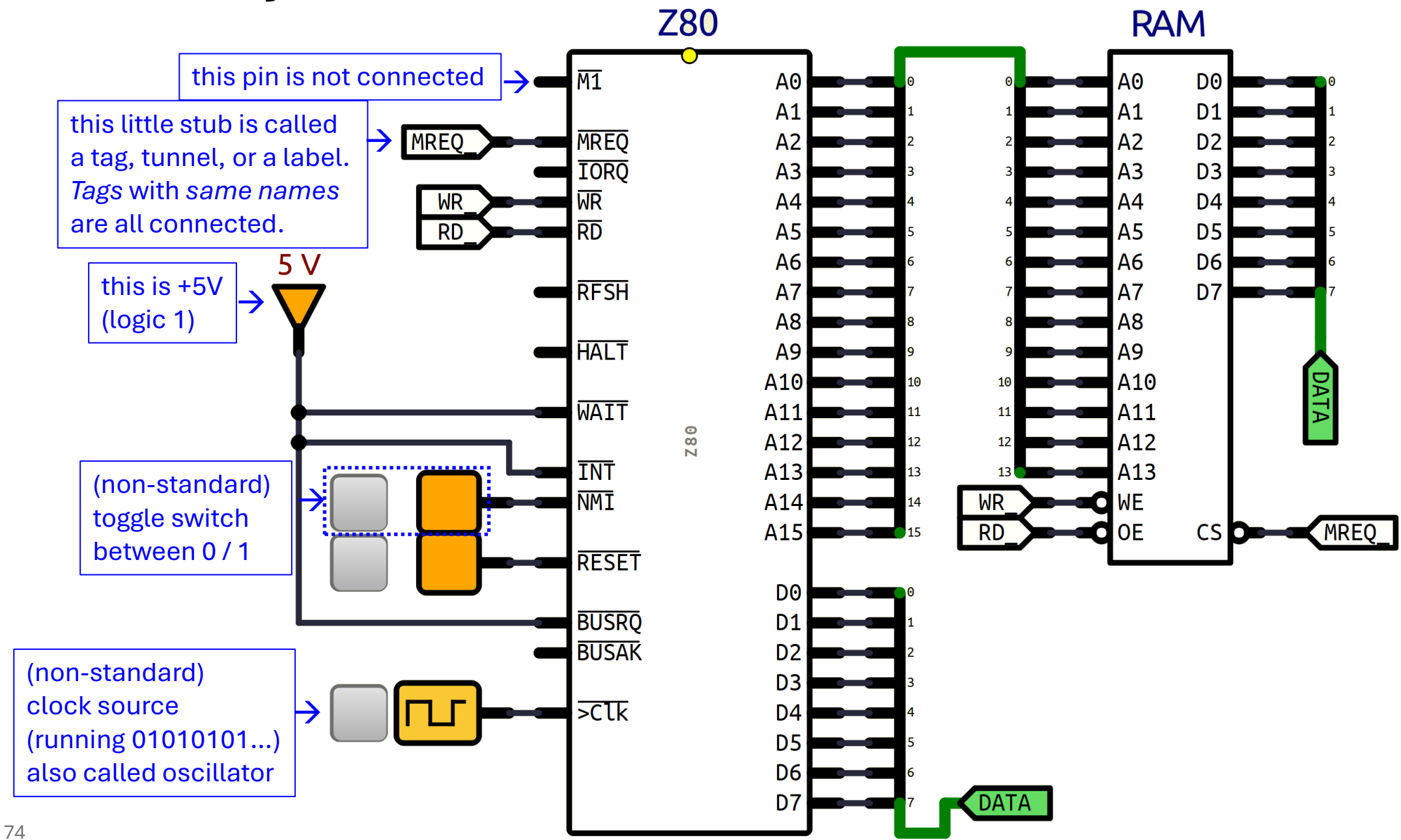
- shows all actual signal connections, but gives abstraction to bus (vector)
- sometimes omits power supply and GND (ground) connections, except where needed.
- usually NOT a real physical layout
- Every active computer chip (Z80 CPU and RAM) needs power (5V DC) and GND (ground) connections.

This schematic actually works in our simulation software.

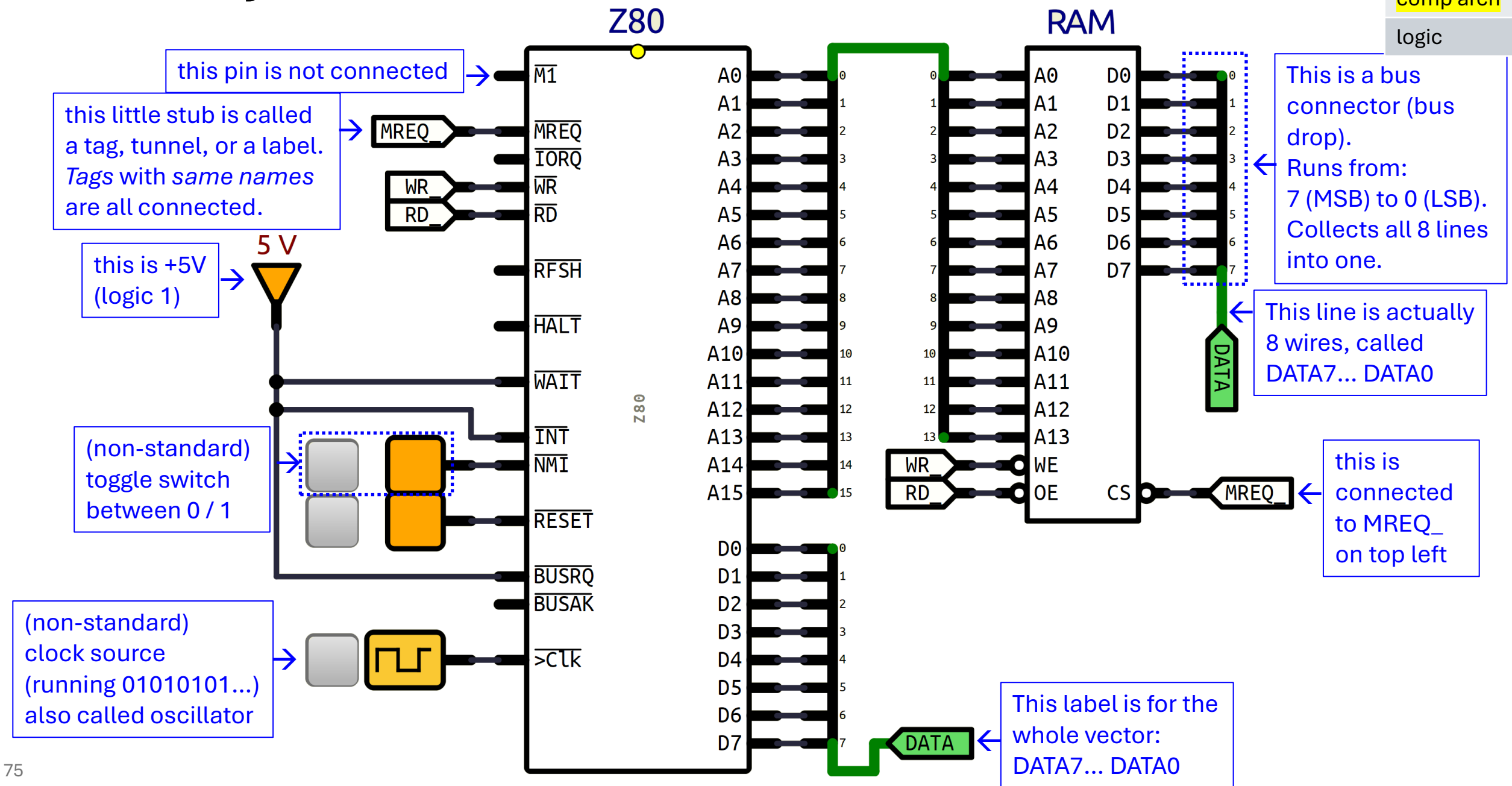
There's no convention for color of the schematics, but there are conventions for symbols.

Why no color? color printing is expensive.

our first system

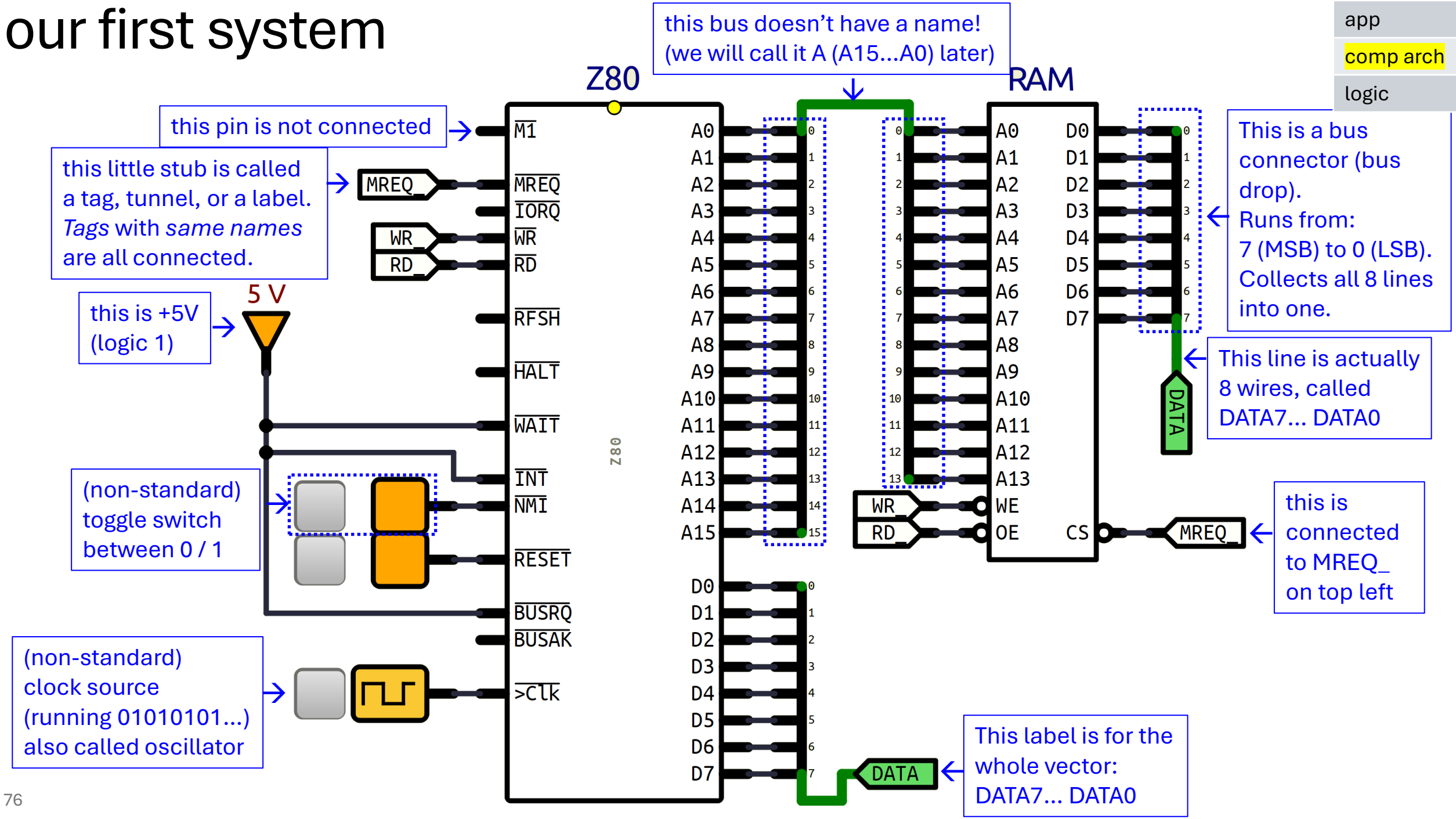


our first system



our first system

app
comp arch
logic



our first system – the C code running inside

app

comp arch

logic

```
#include <stdint.h>

#define ADDR0 0x0060U
#define ADDR1 0x0062U
#define NMI_COUNT 0x0064U

void nmi_isr (void) __critical __interrupt {
    uint8_t *count = (volatile uint8_t *) NMI_COUNT;
    uint8_t v = *count;

    *count = *count + 1;
}

void main(void) {
    uint8_t *count0 = (volatile uint8_t *) ADDR0;
    uint8_t *count1 = (volatile uint8_t *) ADDR1;

    *count0 = 0;
    *count1 = 0;

    while (1) {
        for (uint8_t i = 0; i < 3; i = i + 1) {
            *count1 = *count1 + 1;
        }
        *count0 = *count0 + 1;
    }
}
```

QUICK NOTE:

The code on this page is written to illustrate the points that are made in the next slide. It's intentionally **NOT WRITTEN to the coding guidelines and coding style for this course.**

The guidelines were broken to allow you to see the points of hardware-software interaction.

our first system – the C code running inside

app

comp arch

logic

```
#include <stdint.h>
```

`#include` is called a *Compiler Directive* in C – this one will “include” some easy-to-use integer types

```
#define ADDR0 0x0060U  
#define ADDR1 0x0062U  
#define NMI_COUNT 0x0064U
```

`#define` is called a *Macro* in C – think about it as “find and replace.” Every time `ADDR0` is typed, `0x0060U` replaces it. “U” means unsigned number.

```
void nmi_isr (void) __critical __interrupt {  
    uint8_t *count = (volatile uint8_t *) NMI_COUNT;  
    uint8_t v = *count;  
  
    *count = *count + 1;  
}
```

`__critical __interrupt` are called *Compiler Extensions*. This is non-standard. `__interrupt` means this function is an interrupt service function and `__critical` means it is for a non-maskable interrupt (NMI). These service functions are also called *interrupt service routine* (ISR) – hence the function name `nmi_isr`

```
void main(void) {  
    uint8_t *count0 = (volatile uint8_t *) ADDR0;  
    uint8_t *count1 = (volatile uint8_t *) ADDR1;  
  
    *count0 = 0;  
    *count1 = 0;  
  
    while (1) {  
        for (uint8_t i = 0; i < 3; i = i + 1) {  
            *count1 = *count1 + 1;  
        }  
        *count0 = *count0 + 1;  
    }  
}
```

C programs always start its execution from the `main` function.

Code overview: inside `main`, `count0` points to `ADDR0` (`0x0060`) and `count1` points to `ADDR1`. we increment `count1` 3 times before incrementing `count0` once, and repeat...

but if there's an NMI signal, we stop everything and jump to `nmi_isr`, which increments `NMI_COUNT` (`0x0064`) by one, and return to wherever it came from.

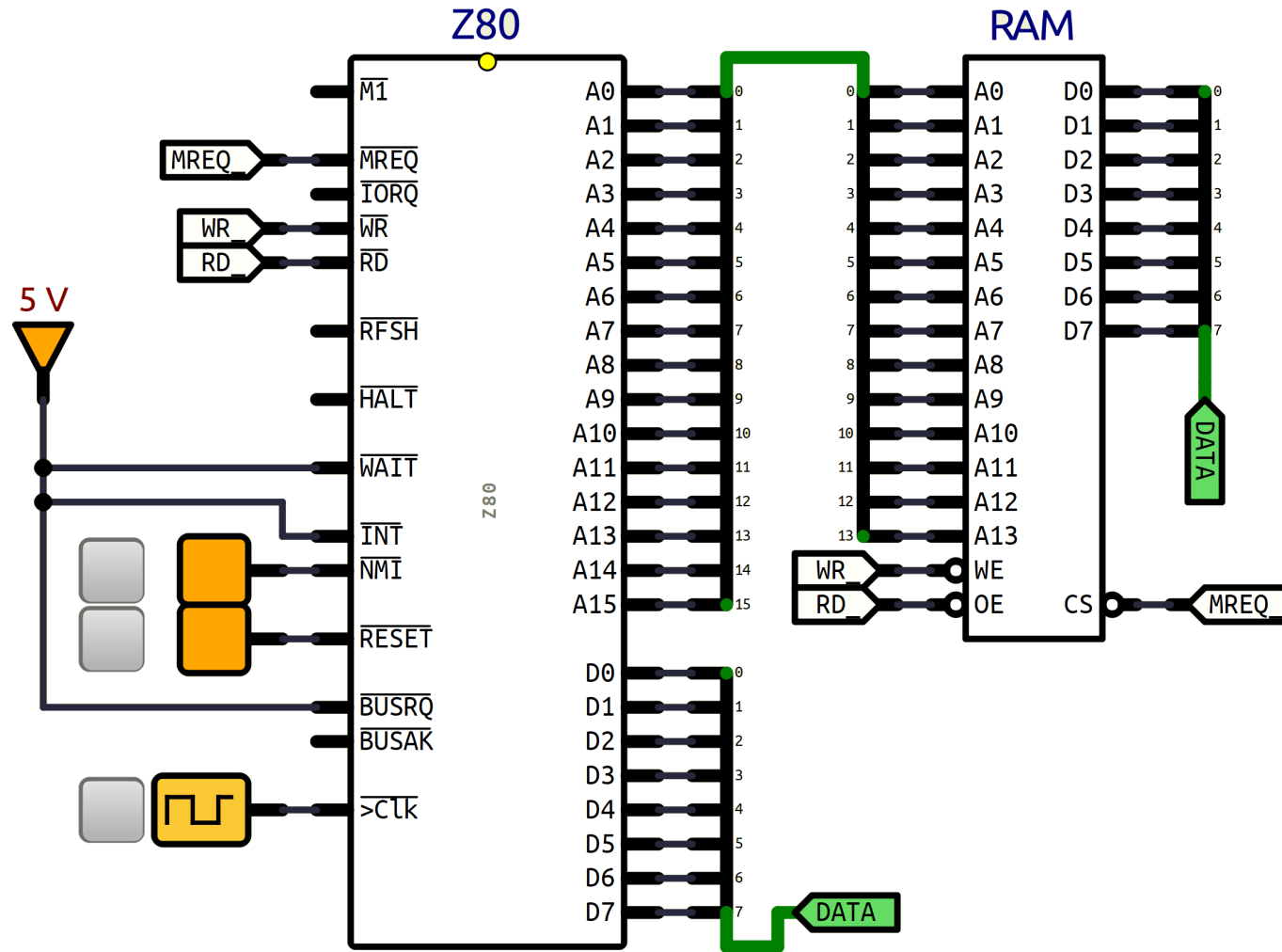
our first system demo

DEMO TIME

app

comp arch

logic



[00-101]

```
#include <stdint.h>
```

```
#define ADDR0 0x0060U
#define ADDR1 0x0062U
#define NMI_COUNT 0x0064U
```

```
void nmi_isr (void) __critical __interrupt {
    uint8_t *count = (volatile uint8_t *) NMI_COUNT;
    uint8_t v = *count;

    *count = *count + 1;
}
```

```
void main(void) {
    uint8_t *count0 = (volatile uint8_t *) ADDR0;
    uint8_t *count1 = (volatile uint8_t *) ADDR1;

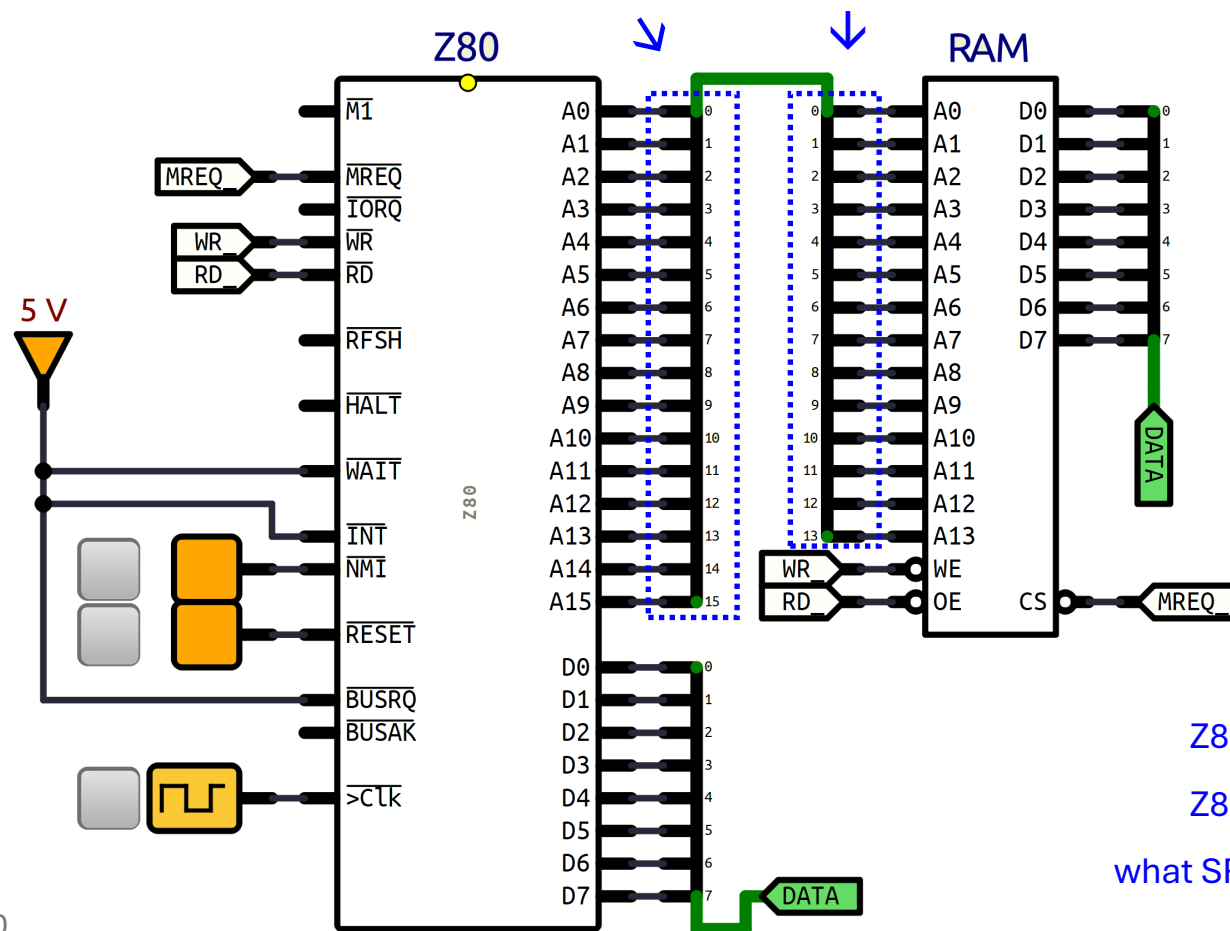
    *count0 = 0;
    *count1 = 0;

    while (1) {
        for (uint8_t i = 0; i < 3; i = i + 1) {
            *count1 = *count1 + 1;
        }
        *count0 = *count0 + 1;
    }
}
```



logic

We have a 14-bit address bus on SRAM side (A13...A0)



Surprising? No.

Z80 CPU supports $2^{16} = 64\text{kB}$ of memory
we could “buy” just one SRAM of 16kB capacity

Z80 will output 16 bits, but A15 and A14 are ignored by SRAM. (SRAM doesn't care about A15 and A14)

[illegible]

Z80 CPU can output 16-bit address (0x0000–0xFFFF)

SRAM can only take 14-bit address (0x0000-0x3FFF)

If we limit ourselves to only `0x0000 – 0x3FFF`, will we be ok?
MAYBE. We have what's called an *address-aliasing problem*:

Example: CPU trying to read from 0x0000 and 0x4000:

[illegible]



our first system (and we have a problem already)

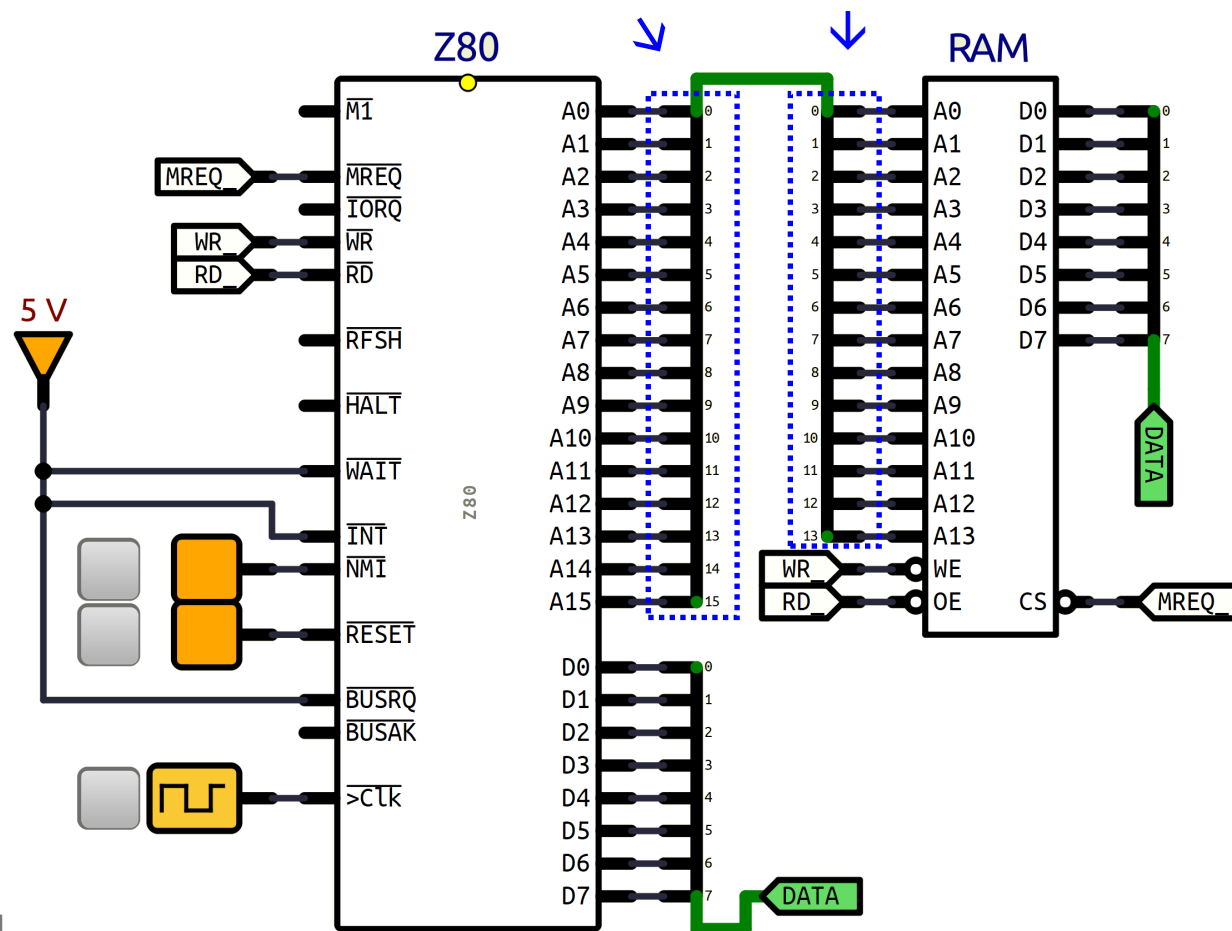
app

comp arch

logic

We have a 16-bit address bus on Z80 CPU side (A15...A0)

We have a 14-bit address bus on SRAM side (A13...A0)



address-aliasing:

CPU address **aliases**:

A bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Z80 0x0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Z80 0x4000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Z80 0x8000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Z80 0xC000	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
what SRAM sees			0	0	0	0	0	0	0	0	0	0	0	0	0	0

CPU reads from 0x4000 gets the same data as 0x0000.
 CPU writes to 0x8000 writes data means writing to 0x0000.
 Since A15 and A14 are ignored by SRAM,
 $0x0000 = 0x4000 = 0x8000 = 0xC000$
 Why? We have 2 bits (A15, A14) as don't-cares \rightarrow we have 2^2 aliases.

Every address in this system has 4 aliases. Examples:
 $0x0123 = 0x4123 = 0x8123 = 0xC123$ (check for yourself)
 $0x2000 = 0x6000 = 0xA000 = 0xE000$ (check for yourself)
 $0x1EAD = 0x5EAD = 0x9EAD = 0xDEAD$ (check for yourself)
 $0x31F0 = 0x71F0 = 0xB1F0 = 0xF1F0$ (check for yourself)



our first system (and we have a problem already)

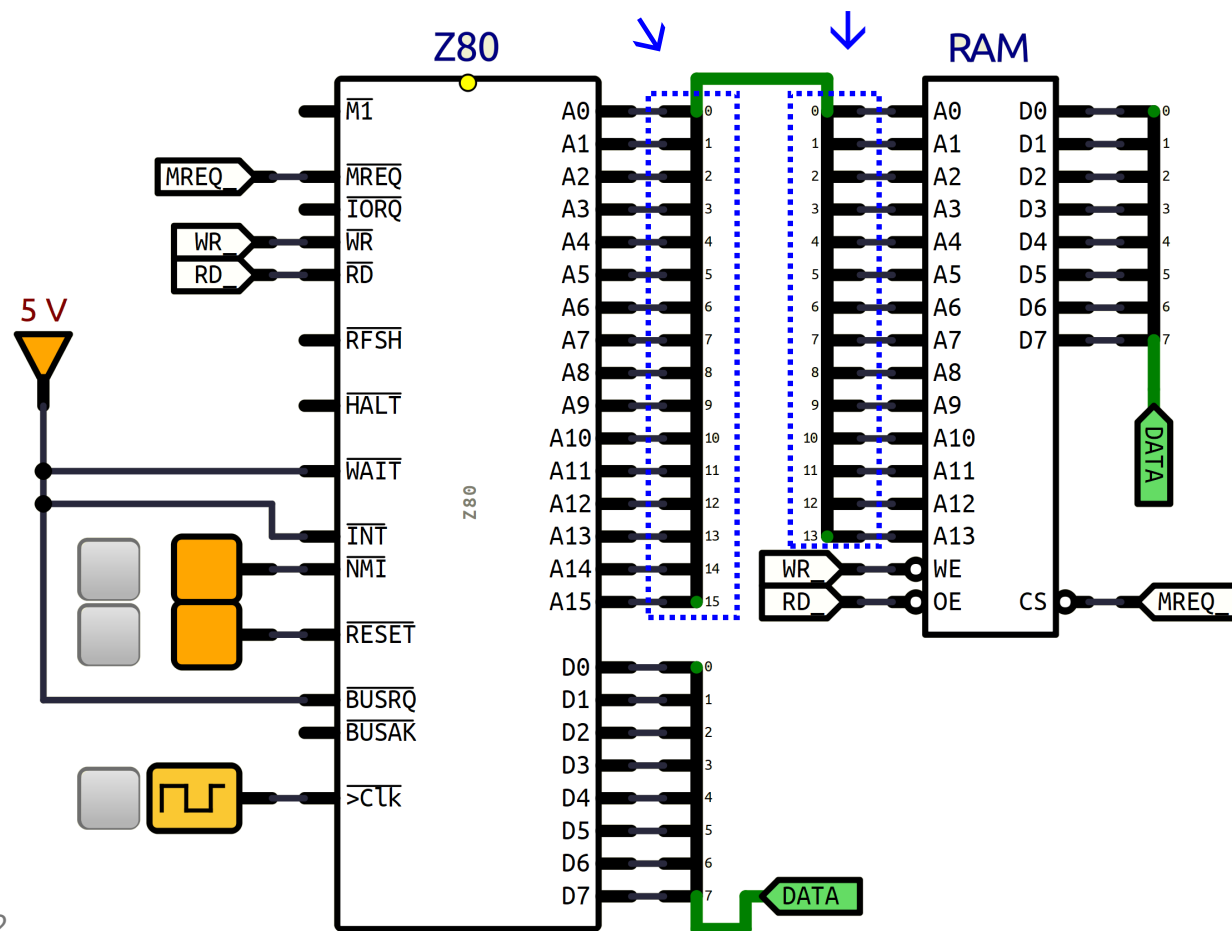
app

comp arch

logic

We have a 16-bit address bus on Z80 CPU side (A15...A0)

We have a 14-bit address bus on SRAM side (A13...A0)



full aliasing table:

SRAM ignores A15 and A14. A13...A0 connects ok

A	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Z80
SRAM	X	X

So, given that *mnp* are hex digits, so m, n, p each in [0-9A-F]

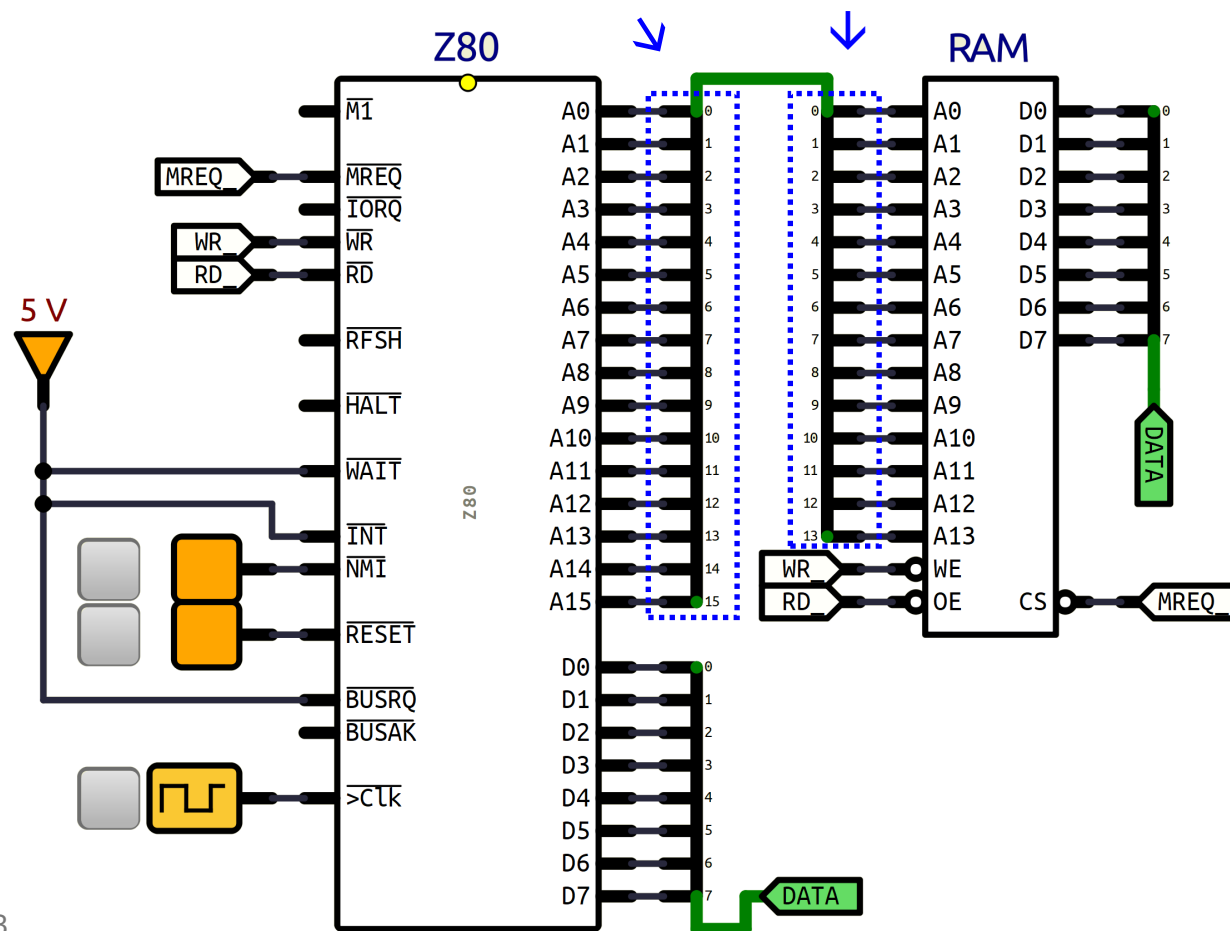
List of all aliases are:

- $0x0mnp = 0x4mnp = 0x8mnp = 0xCmnp$
- $0x1mnp = 0x5mnp = 0x9mnp = 0xDmnp$
- $0x2mnp = 0x6mnp = 0xAmnp = 0xEmnp$
- $0x3mnp = 0x7mnp = 0xBmnp = 0xFmnp$

(Goal: Must be able to understand the whole aliases above)



We have a 14-bit address bus on SRAM side (A13...A0)



If we:

- Make sure that the software running on Z80 only use address 0x0000 – 0x3FFF
- Don't connect anything else to “fight” with the RAM

Then we're ok!

But...

- We are only utilizing 16kB out of 64kB that's possible on the Z80
- We haven't got any I/O yet except that RESET_ and NMI_ buttons (and they're not really affecting our main, so it's not really an input).

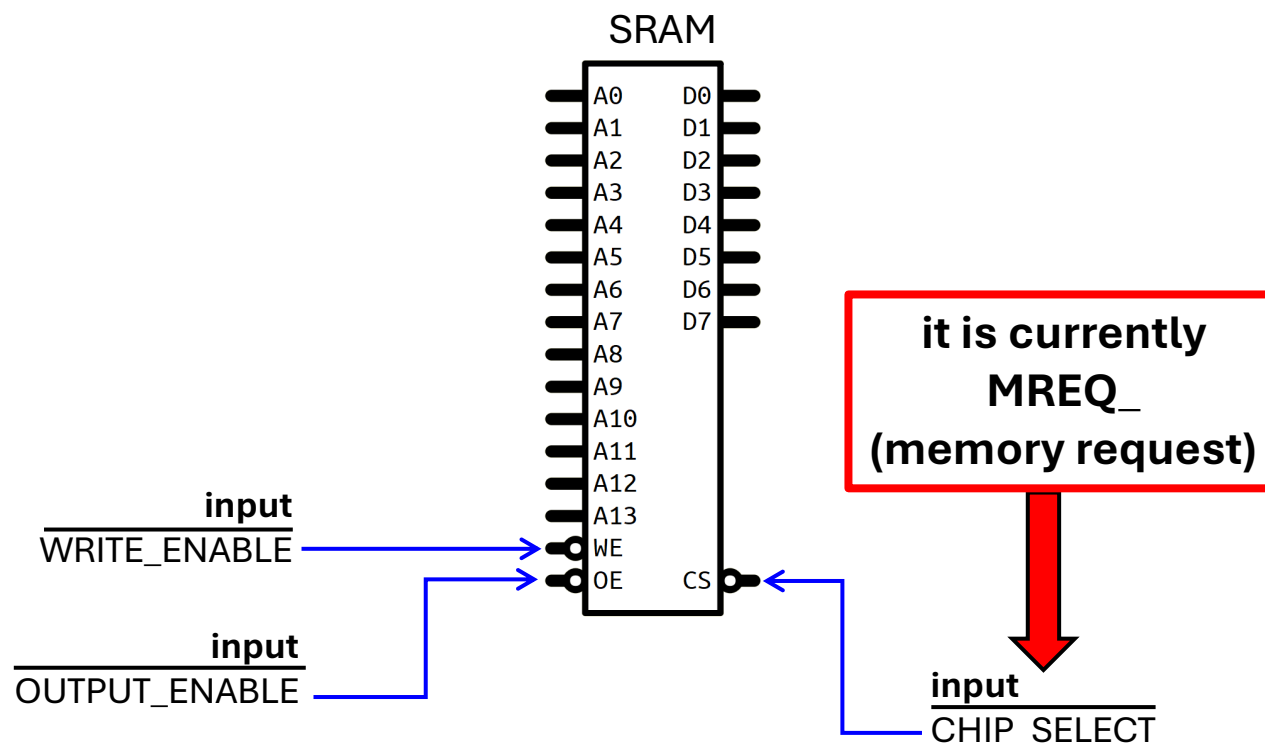
So, most of the time, we want to get rid of address-aliasing:

- It's a bad idea to have aliasing in the first place. We want each byte in RAM to have a unique address from the CPU
- Having aliasing can introduce nasty, hard to find bugs when we write software. Hardware should help software with that

Solution is called *address decoding*.



address decoding done right



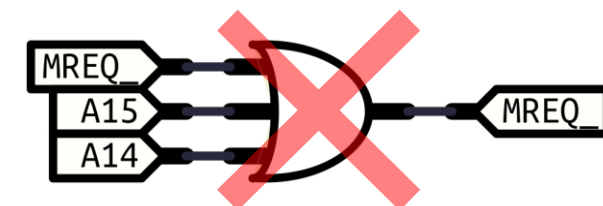
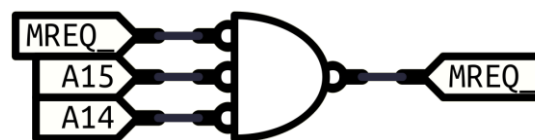
RAM control signal truth table

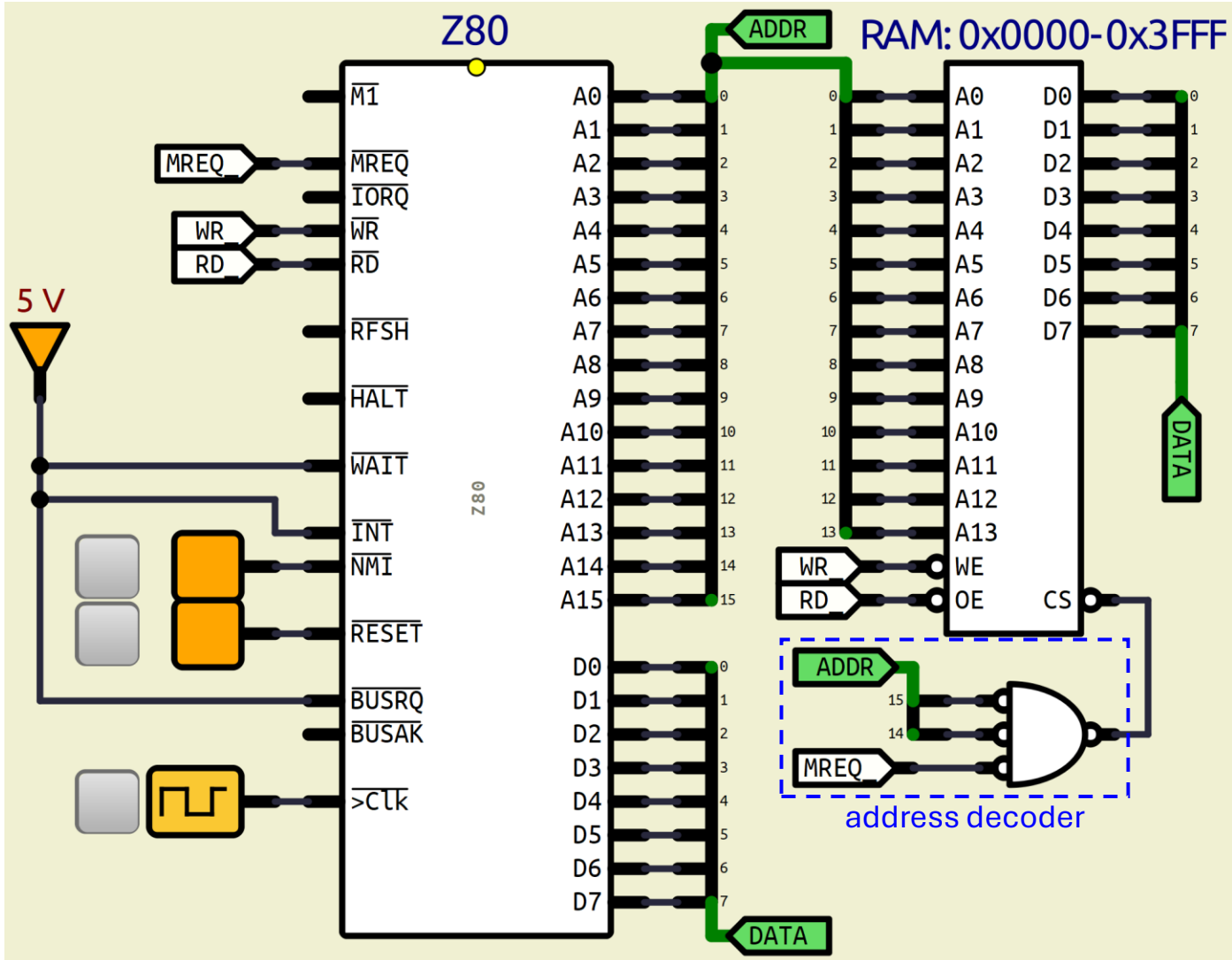
CS_	OE_	WE_	Meaning	D7-D0
1	x	x	chip is deselected	high Z
0	1	1	selected, but not enabling output	high Z
0	0	1	read from an address	data out from address
0	x	0	write to an address	data in to address

Modify CS_ so that CS_ is asserted only when *A15=0 and A14=0 and (MREQ_ asserted)*:

Old: CS_ = MREQ_

New: CS_ = $\sim(\sim\text{MREQ_} \& (\sim\text{A15} \& \sim\text{A14}))$





So, how is it now?

address decoding is now done properly:

[illegible]

This SRAM + decoder will only respond to address 0x0000-0x3FFF. All read/write to other addresses (0x4000-0xFFFF) now have **undefined behavior**, which is *a good thing*.

Why is undefined behavior of *out-of-bound* addresses(0x4000-0xFFFF) good?

- helps us catch bugs that read or write to undefined region
- helps us connect other devices in 0x4000-0xFFFF region



address decoding and RD_ and WR_ fix

app

comp arch

logic

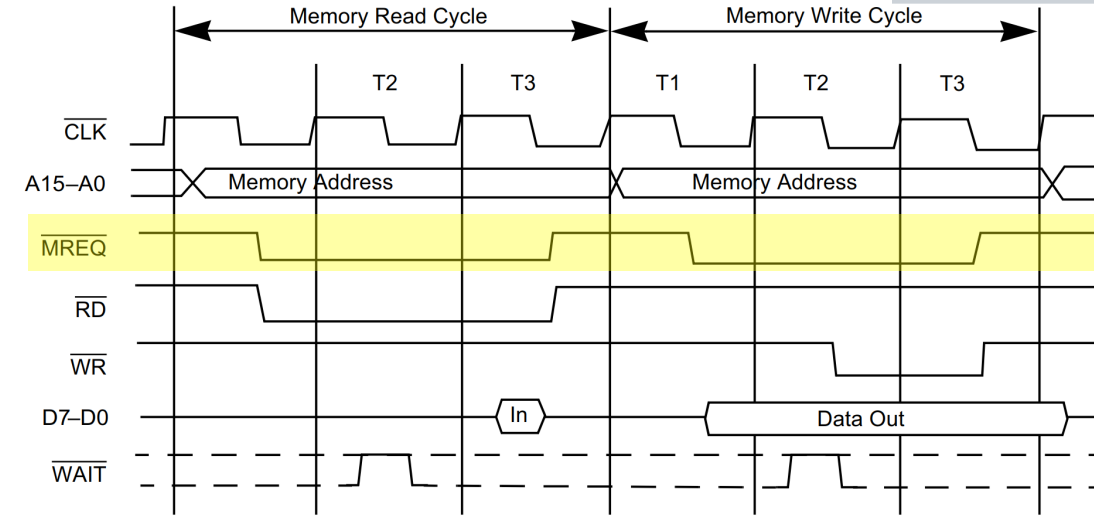
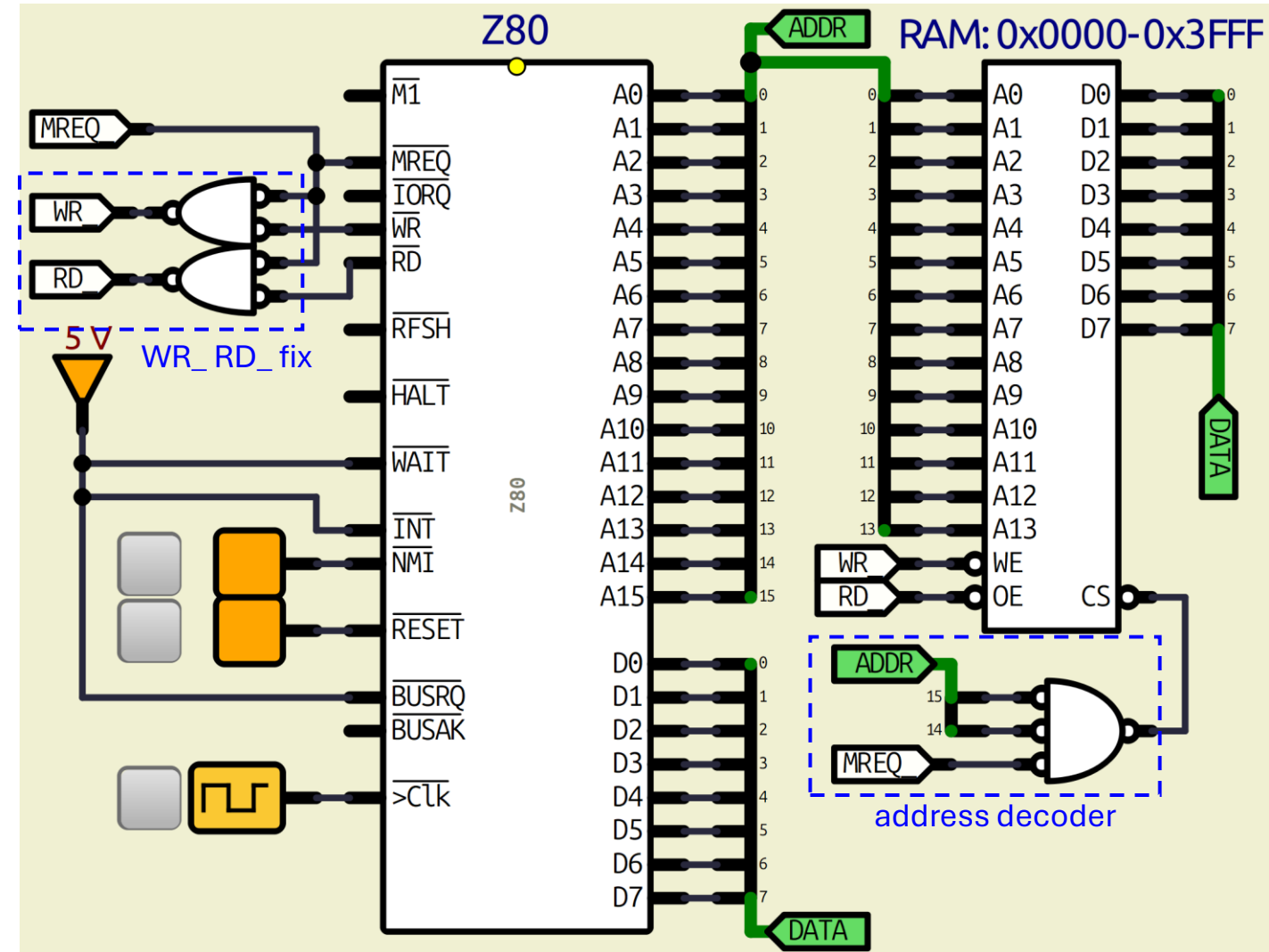


Figure 6. Memory Read or Write Cycle

So, what OTHER fix do we need? We need one more fix.

On Z80, there are times when the address bus is used for something else (won't talk about in this class). The address bus is only reliably accessing memory when MREQ_ is asserted

So, we need to make sure that RD_ and WR_ for all address-bus-accessible things is only asserted when MREQ_ is asserted.

Fix is (remember MREQ, RD, WR are all asserted low):

$$\text{ADDR_BUS_RD} = \text{MREQ} \ \& \ \text{RD} \quad \text{and}$$

$$\text{ADDR_BUS_WR} = \text{MREQ} \ \& \ \text{WR}$$

But for sake of simplicity, we still call them RD_ WR_ still.

Memory Map

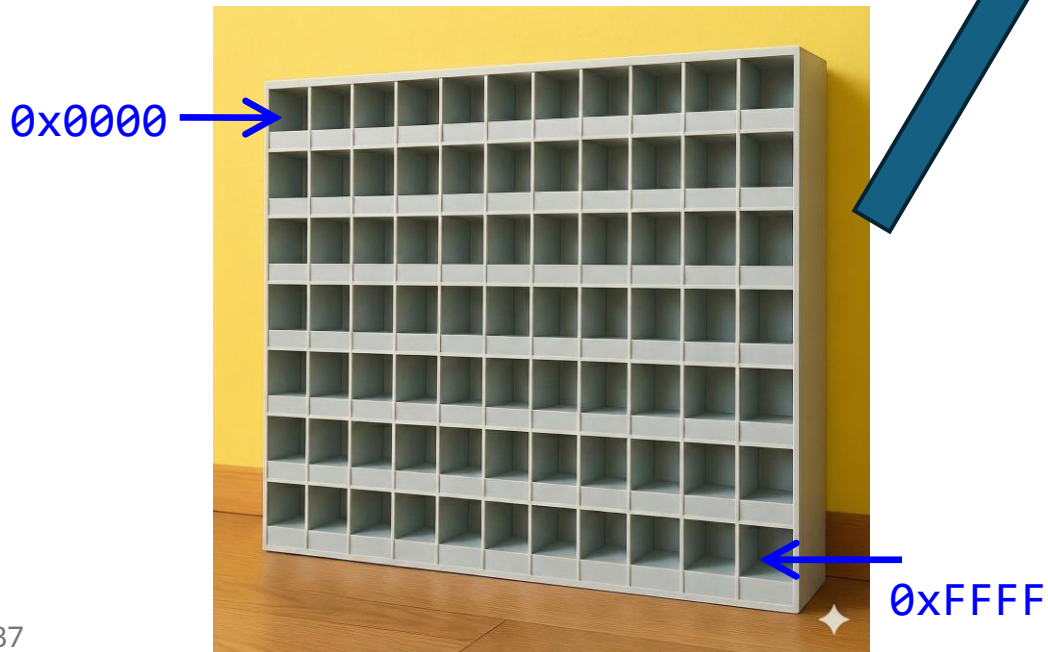
Remember Z80 CPU has 16-bit address bus.

So, possible addresses are 0x0000-0xFFFF

In a computer design, create a “memory map” to show where things are for all the possible addresses.

Think about this as having a map to show what each range of mailboxes are:

we will write memory map as one tall column instead of a mailbox now.



OUR SYSTEM MEMORY MAP

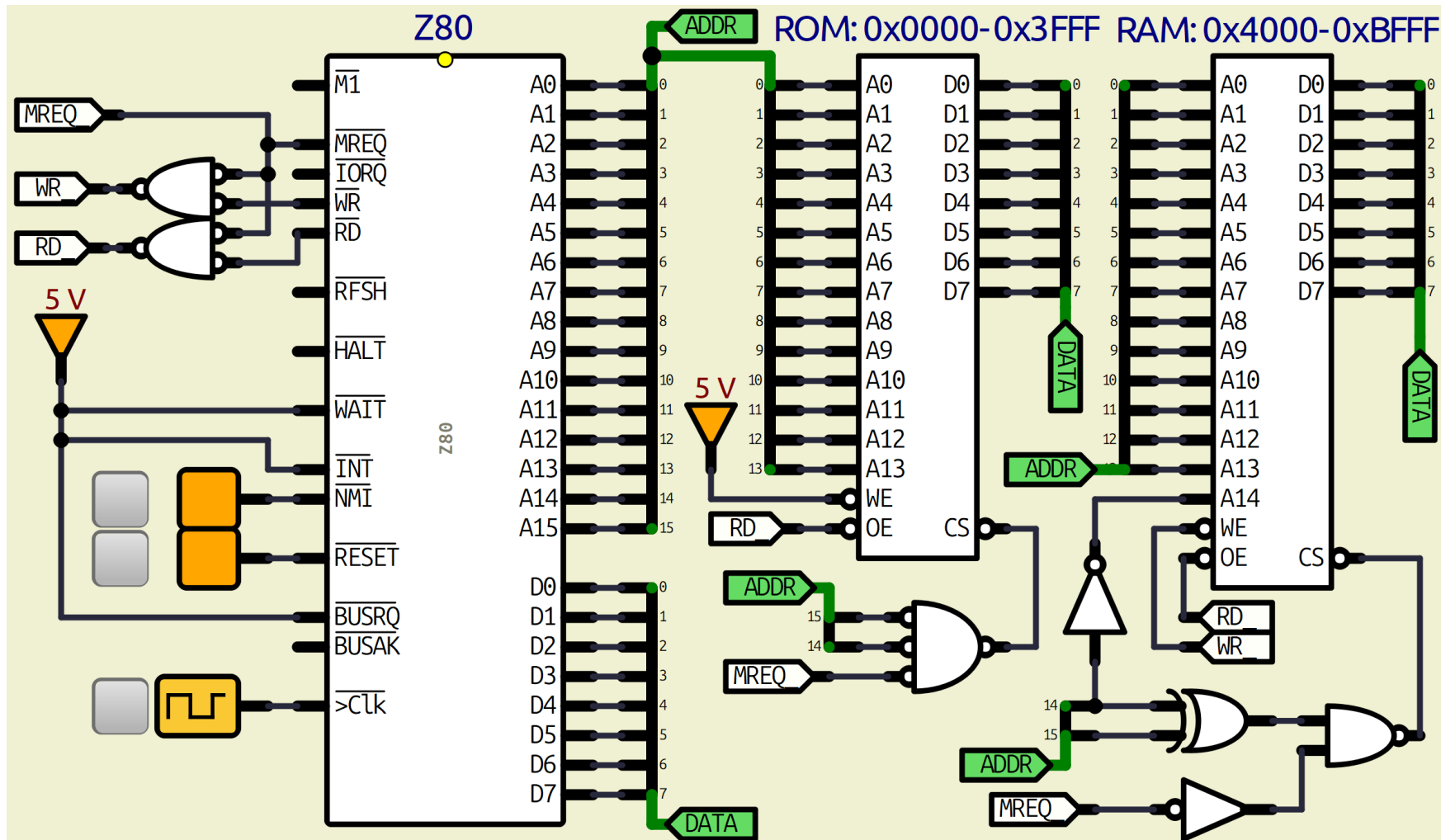
Address	What's there?	app
		comp arch
		logic
0x0000	RESET jumps here (Z80 magic)	
0x0066	NMI_ jumps here (Z80 magic)	
0x0100 0x3FFF	program starts here (our magic) last address for program	
0x4000 ... 0x40FF	Your (student's) safe memory zone. Free to write to and read from these addresses without danger	
0x4100 ... 0x41FF	Instructor memory zone. Danger zone! You may corrupt the whole system writing to these addresses.	
0x4200 ...	program data space. contains all data types compiler produce automatically	
0xBFFF	top of the stack (more later). it grows upwards.	
0xC000 ...	I/O zone. this is not RAM. we use these addresses to communicate to the outside world.	
	I/O devices can be 1. READ ONLY – write does nothing 2. WRITE ONLY – read may return garbage 3. READ and WRITE – can read or write	
0xFFFF		

Memory Map

app

comp arch

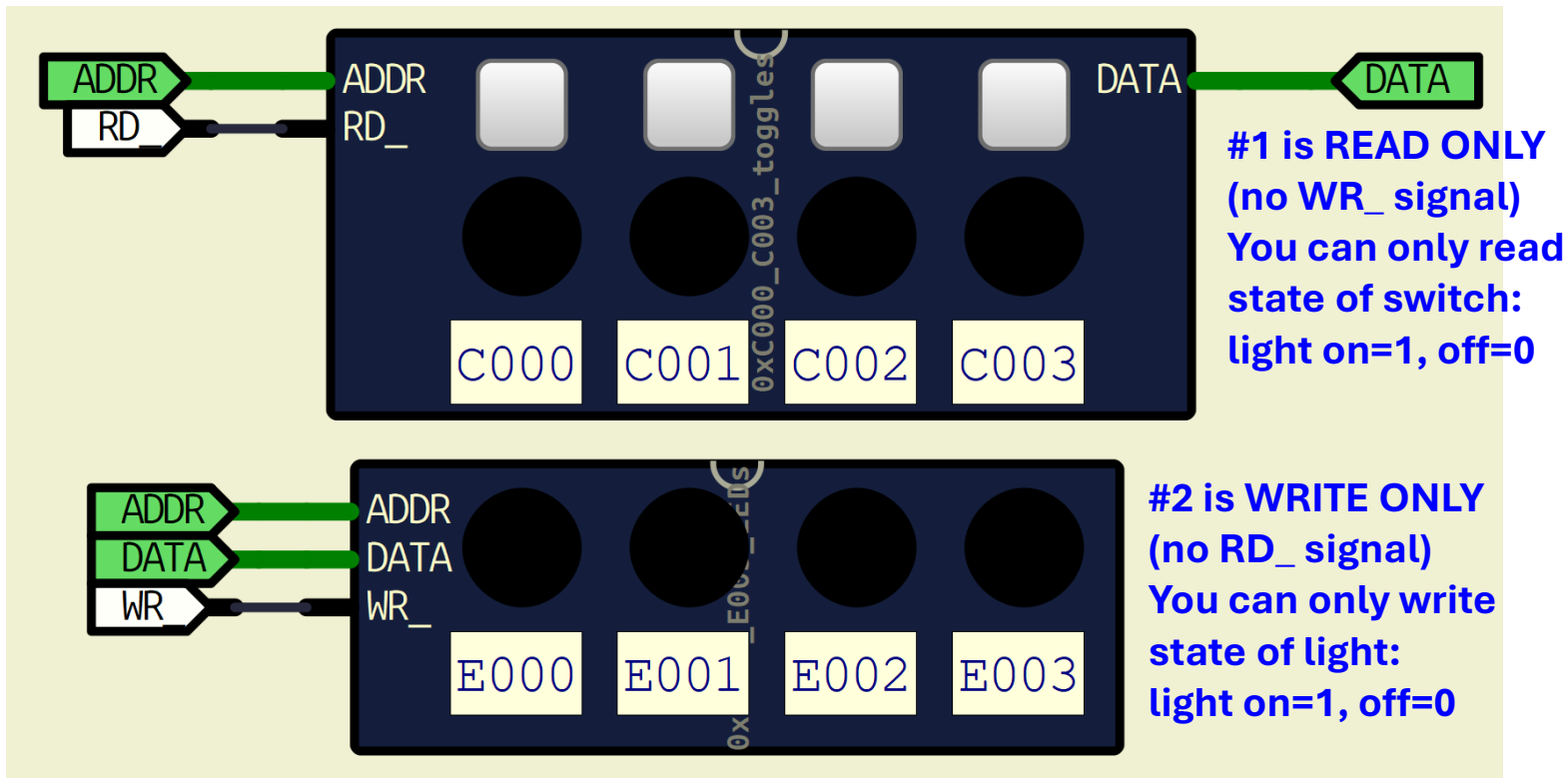
logic



We will do 3 things with out previous hardware design:

- Changing the first RAM from 0x0000-0x3FFF to read only (ROM)
- Add another RAM from 0x4000-0xBFFF to read/write data from our program
- Add I/O
 - Add 4 toggle switches (0/1) at address 0xC000-0xC003.
 - Add 4 LEDs at 0xE000-0xE003

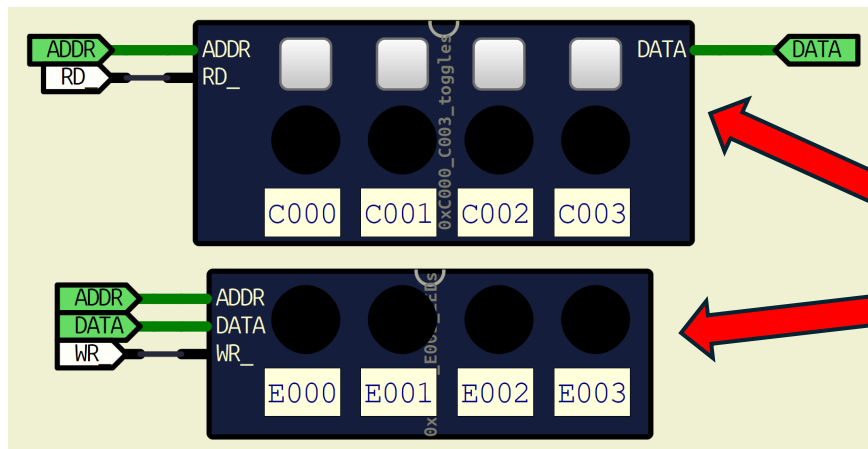
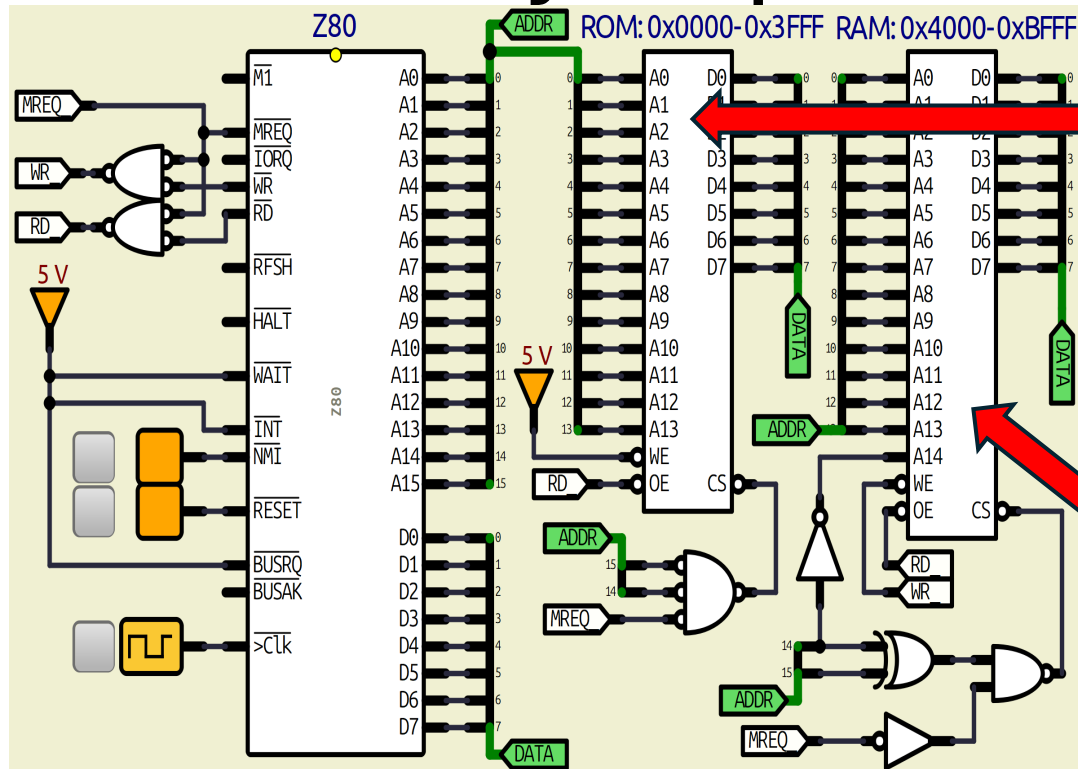
Memory Map



We will do 3 things with out previous hardware design:

- Changing the first RAM from 0x0000-0x3FFF to read only (ROM)
- Add another RAM from 0x4000-0xBFFF to read/write data from our program
- Add I/O
 - #1 • Add 4 toggle switches (0/1) at address 0xC000-0xC003.
 - #2 • Add 4 LEDs at 0xE000-0xE003

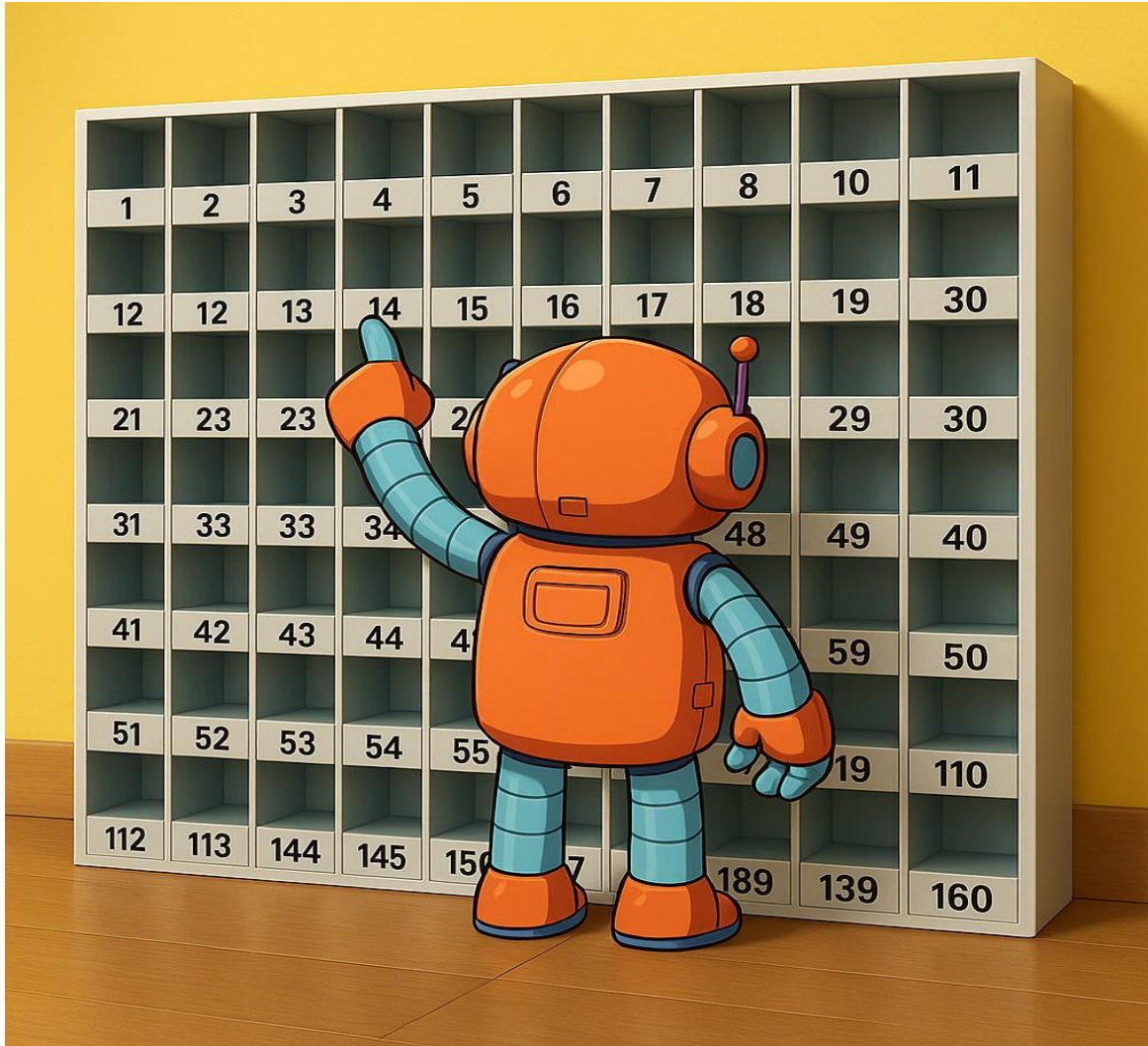
Memory Map



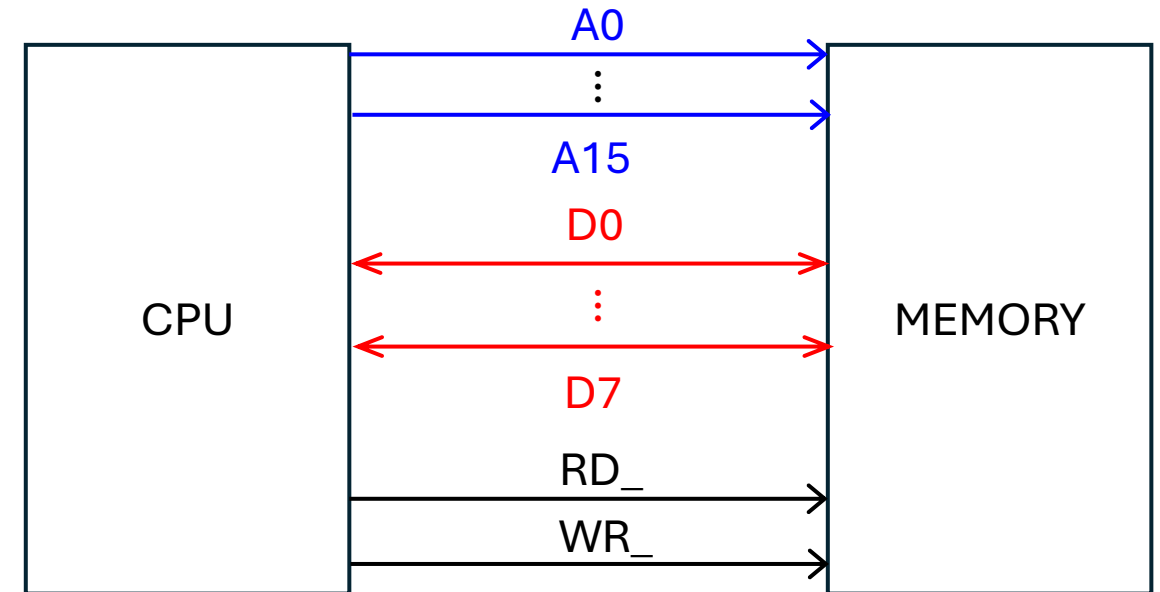
Address	What's there?	app
0x0000	RESET jumps here (Z80 magic)	comp arch
0x0066	NMI_ jumps here (Z80 magic)	logic
0x0100 0x3FFF	program starts here last address for program	
0x4000 ... 0x40FF	Your (student's) safe memory zone. Free to write to and read from these addresses without danger	
0x4100 ... 0x47FF	Instructor memory zone. Danger zone! You may corrupt the whole system writing to these addresses.	
0x4800 ...	program data space. contains all data types compiler produce automatically	
0xBFFF	top of the stack (more later). it grows upwards.	
0xC000 ...	I/O zone. this is not RAM. we use these addresses to communicate to the outside world.	
	I/O devices can be	
	1. READ ONLY – write does nothing	
	2. WRITE ONLY – read may return garbage	
0xFFFF	3. READ and WRITE – can read or write	

Summary: How CPUs talk to "memory"

Concept:



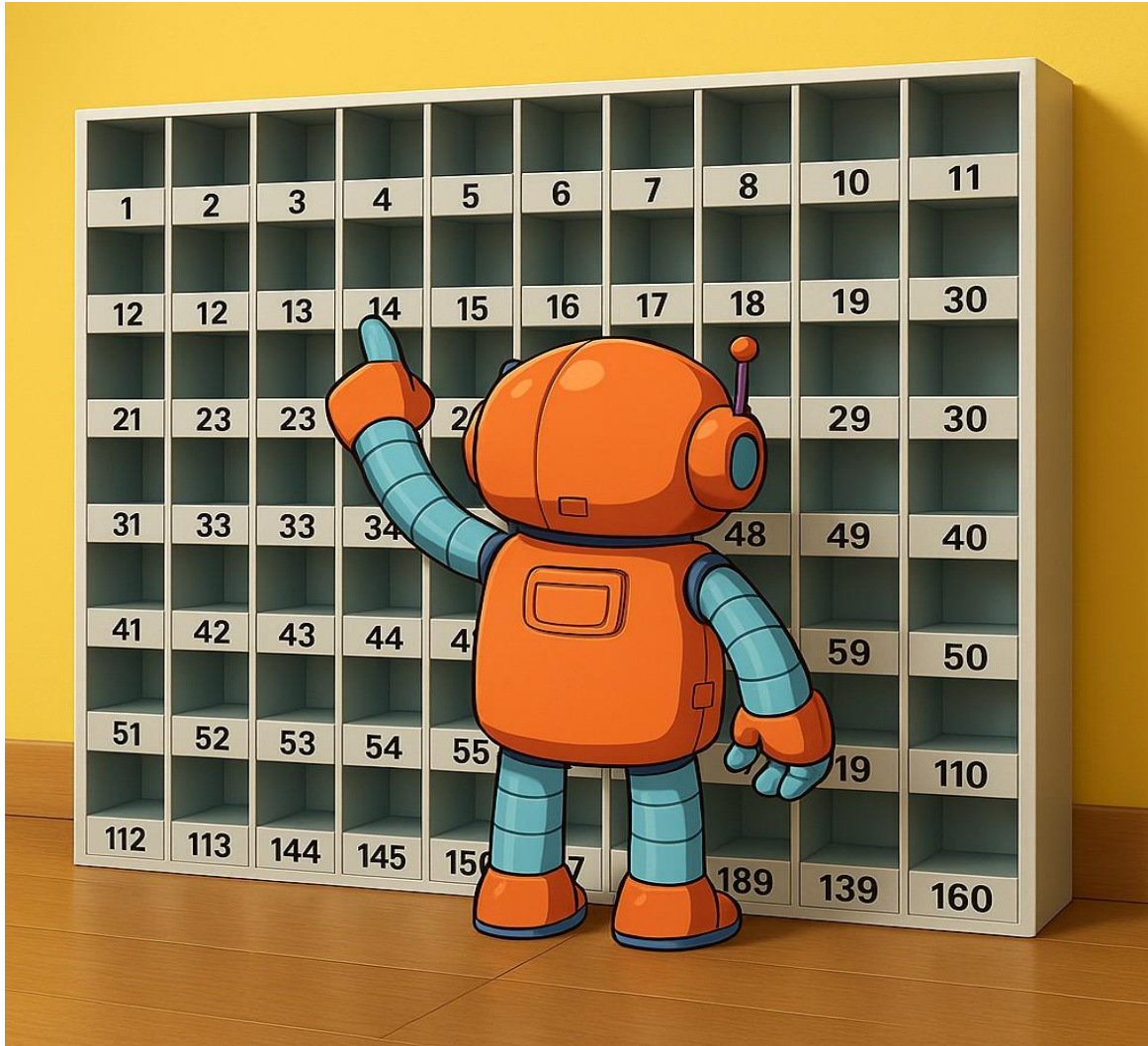
Reality:



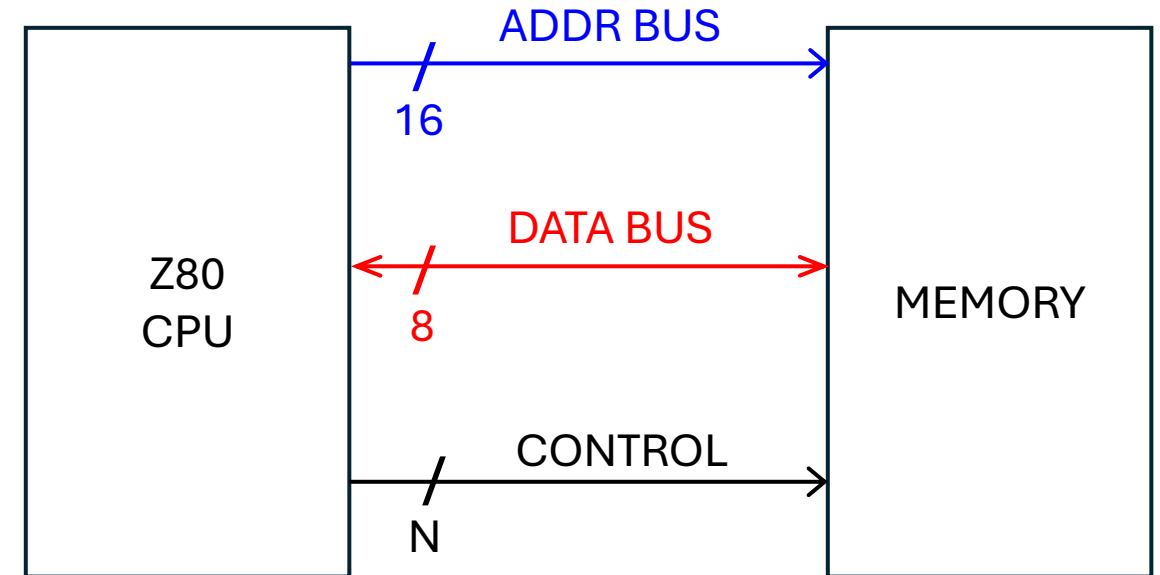
- **ADDR:** 16 wires from CPU to MEMORY. CPU always "talks" (assert data)
- **DATA:** 8 wires *between* CPU and memory.
 - If CPU talks (write), MEMORY listens. → WR_ asserted
 - If CPU listens (read), MEMORY talks. → RD_ asserted
- **CONTROL:** misc signals from CPU. RD_ WR_

Summary: How CPUs talk to "memory"

Concept:



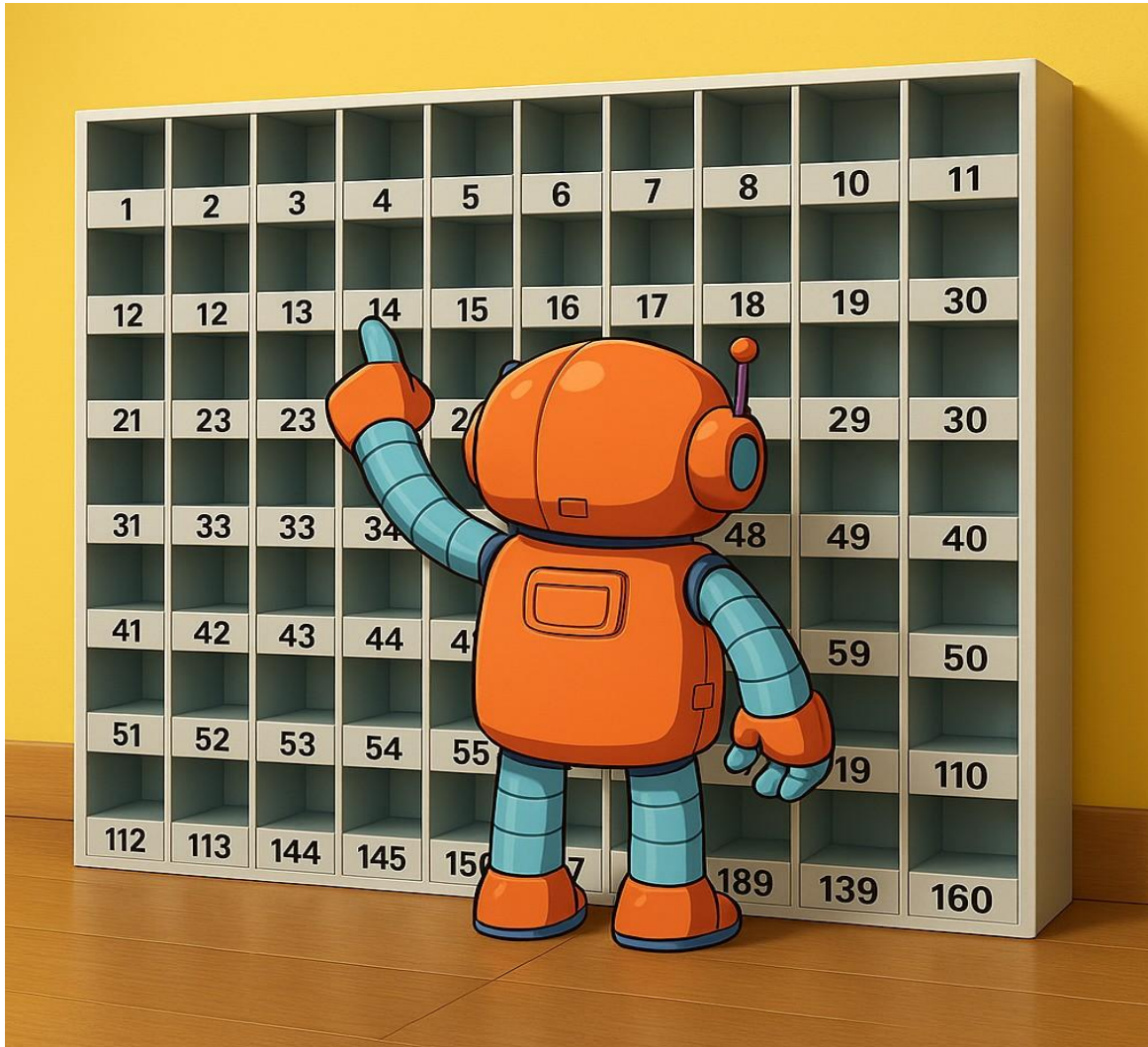
Reality:



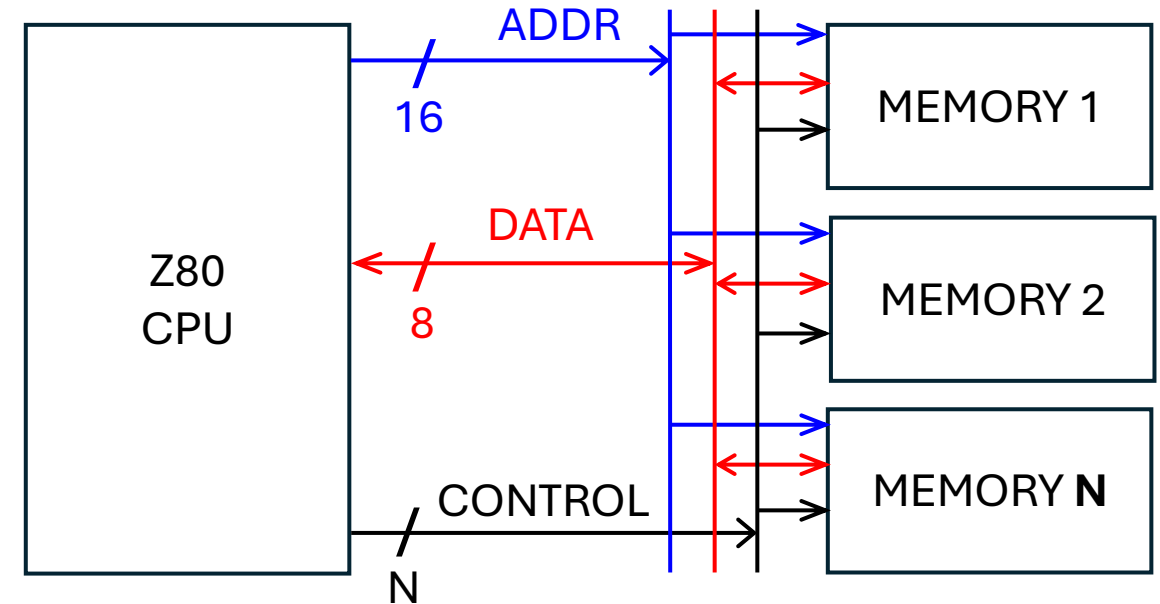
- **ADDR BUS:** 16 wires from CPU to MEMORY
- **DATA BUS:** 8 wires *between* CPU and memory
- **CONTROL BUS:** misc signals from CPU to "memory"

Summary: How CPUs talk to "memory"

Concept:



Reality (many memory blocks):



- **ADDR BUS:** 16 wires from CPU to MEMORY
- **DATA BUS:** 8 wires *between* CPU and memory
- **CONTROL BUS:** misc signals from CPU to "memory"

All above signals are connected to *every* memory elements- that's why they're called *buses*.

Each signal group has more than 1 bit – called a *vector*.

01 Workshop

Goals: go to <https://github.com/kongkrit/BareMetal-C>

- Follows all the setup documents on repo page.

Homeworks:

1. Read the “C99 Coding Guidelines” and “C99 Style Guide” on repo.
2. TBD

