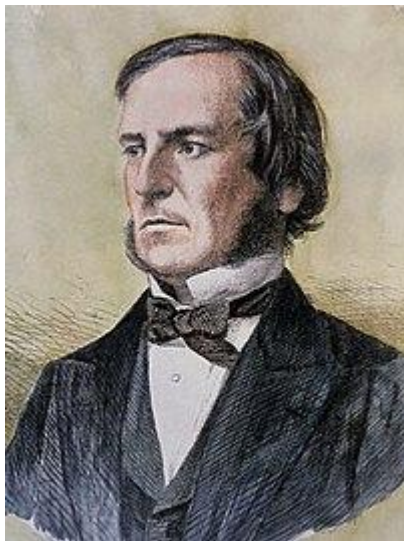


ICDS Spring 2025

Computer Memory

Digital circuits and circuits for storage

Recap: George Boole's invention



- Boolean algebra \rightarrow to express logics like math
 - Boolean variables: two possible values
 - Boolean operators:
 - NOT, (denoted by " \neg ")
 - AND, (denoted by " \times " or " \wedge ")
 - OR, (denoted by " $+$ " or " \vee ")

George Boole, 1815-1864.
The founder of Boolean Algebra

From Boolean algebra to Electrical Circuits



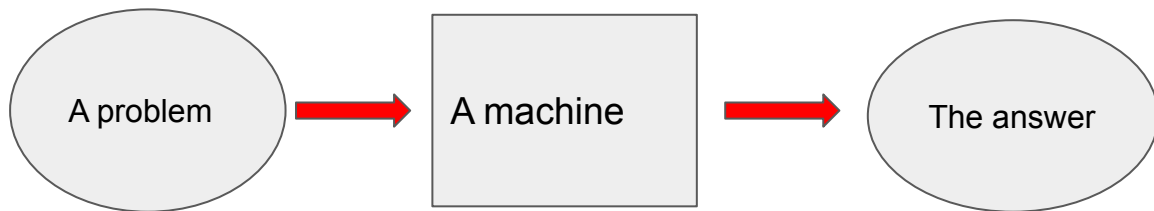
Boolean operators can be implemented by electrical circuits (and so expressions) → Compute logics by circuits

- No mechanical device any more

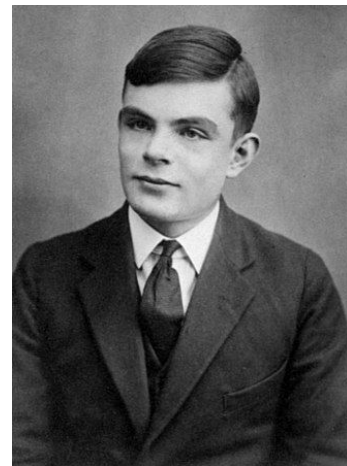
Claude Elwood Shannon, 1916-2001.
“The father of information theory”

Turing machine

- It is an ideal model that maps an input problem to its answer.



- It is a “function” that can represent a solution with several simple operations. (A simulation of human’s calculation)
- Programming (a Turing machine) is to describe the solution (algorithm) with these simple operations available in the machine.

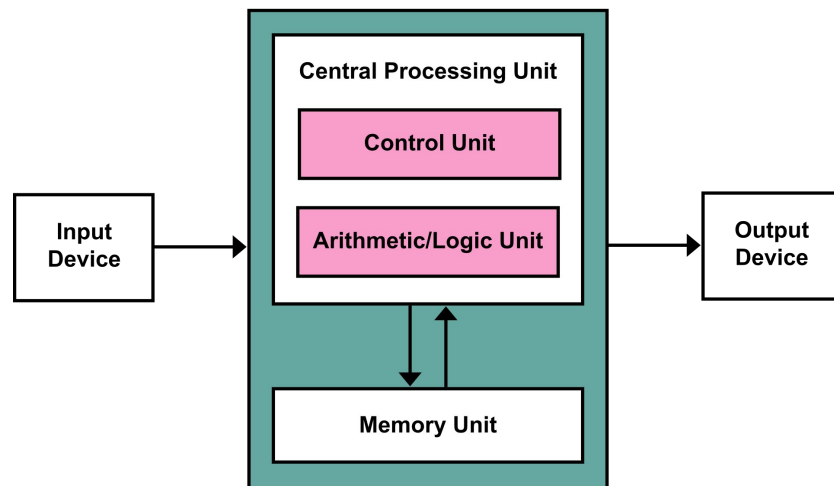


Alan Turing, 1912 - 1954

Von Neumann Architecture

Von Neumann architecture: a Turing machine made by electric circuits.

- Tape → memory
- Head → CPU
- State table → programs

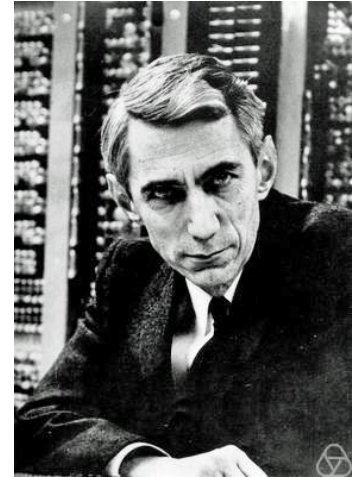


Von Neumann architecture

Agenda

- **Logic gates**
- Types of memory
- Memory Circuits
- Memory Hierarchy
- Locality

Digital circuits

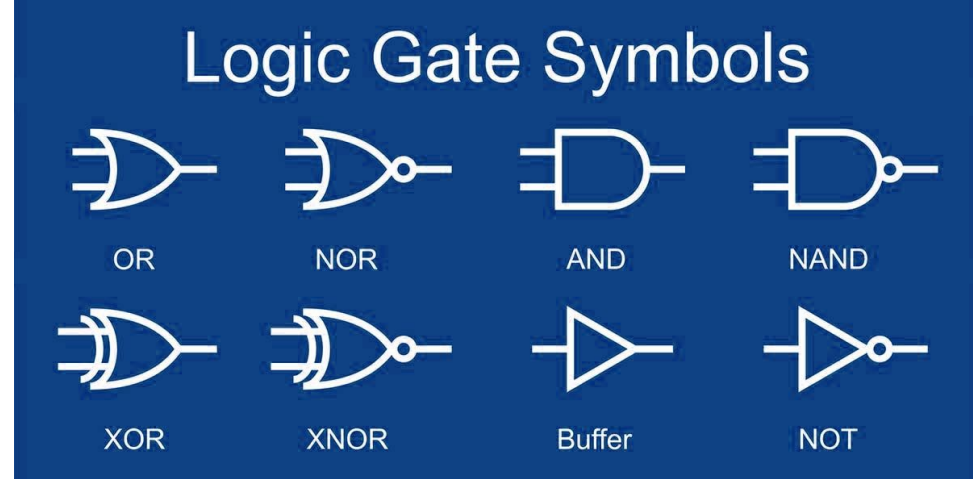


Claude Elwood Shannon, 1916-2001.
“The father of information theory”

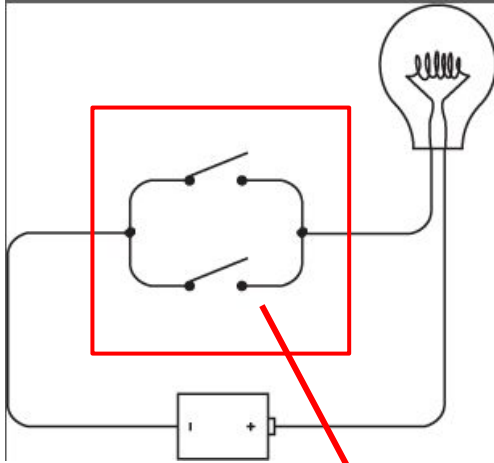
Logic gates

Logic gates are simple digital circuits that

- taking one, or two binary inputs
- producing a binary output
- working like Boolean operators



Example: circuit of the OR gate



- Two switches represent the inputs,
- Two states: on and off
- Connected parallel
- Any input is on, then the output is on → logic OR



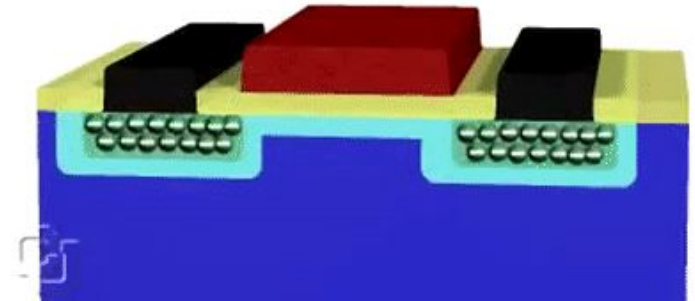
In today's computers, switches are usually constructed out of [transistors](#).

Transistors

We can control the states (i.e., on/off) of the switch by the gate voltage.

- High voltage: switch is on
- Low voltage: switch in off
- 5 volts: the standard voltage of the chips and drivers in computers (while some are 1.5 v or 3.3 v)

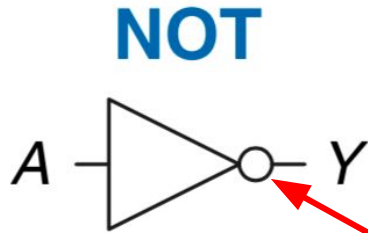
If you would like to know more about [transistors](#), you may watch this [video](#).



NOT gate

A NOT gate has one input, A, and one output, Y.

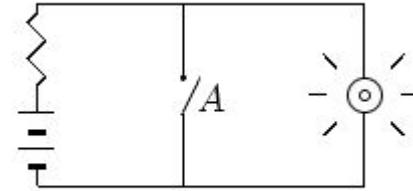
- The output is the inverse of its input.
 - If A is False, then Y is True.
 - If A is True, then Y is False.



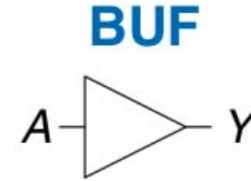
$$Y = \bar{A}$$

This circle means “not/inversion”.

A	Y
0	1
1	0



(a.) NOT gate



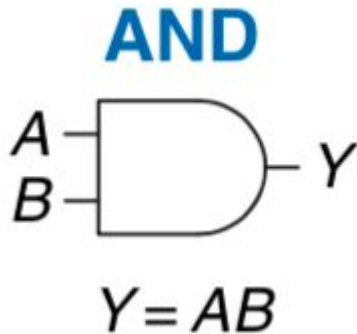
A	Y
0	0
1	1

Without the circle, the gate is called buffer, which simply copies the input to the output.

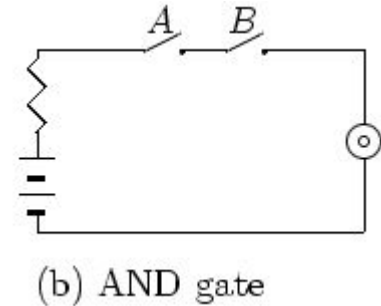
AND gate

The AND gate has two inputs, A and B, and one output, Y.

- It produces a True, if and only if both A and B are True.
- Otherwise, the output is False.



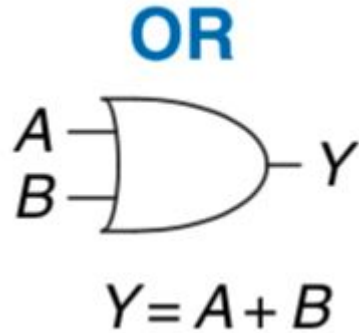
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



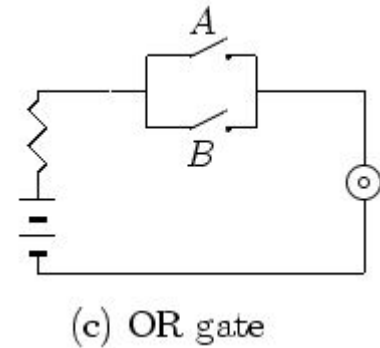
OR gate

The OR gate has two inputs A and B, and one output Y.

- It produces a True, if either A or B (or both) are True.
- Otherwise, Y is False.



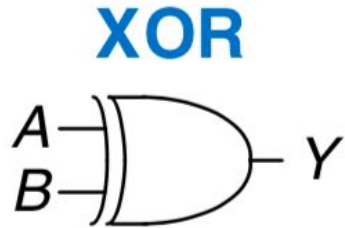
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



XOR gate

The XOR gate has two inputs A and B, and one output Y. (pronounced as “exclusive OR”)

- Y is True if A or B, but not both, are True.



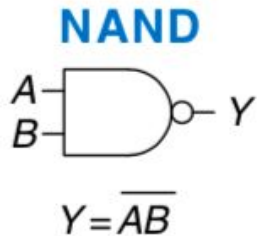
$$Y = A \oplus B$$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

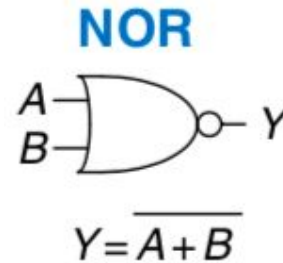
Other Two-input gates

Any gate can be followed by a bubble to invert its operation.

- The NAND gate performs NOT AND.
- The NOR gate performs NOT OR.

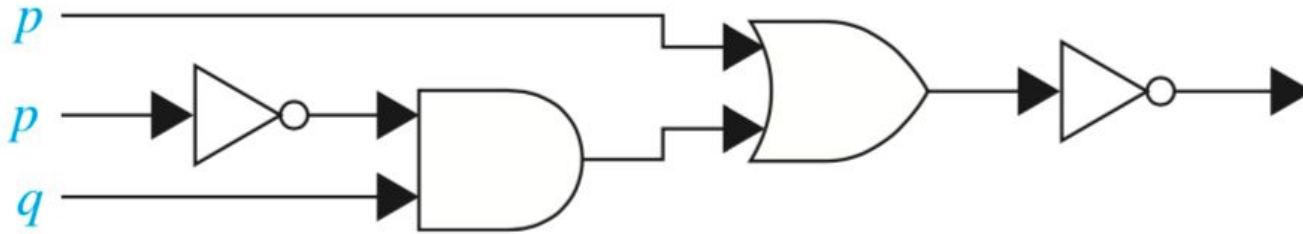


A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Boolean expression of the circuits



What is the boolean expression of this circuit?

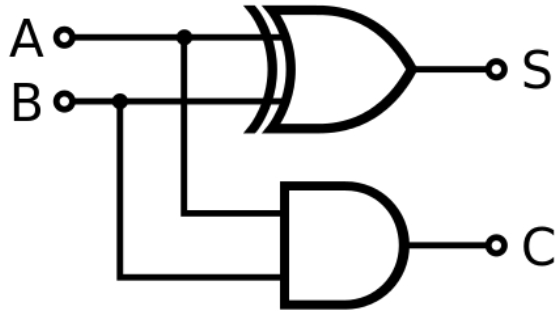
- A. $p \times (p + q)$
- B. $\neg(p + (p \times q))$
- C. $p + (\neg p \times q)$
- D. $\neg(p + (\neg p \times q))$

Scan to answer!

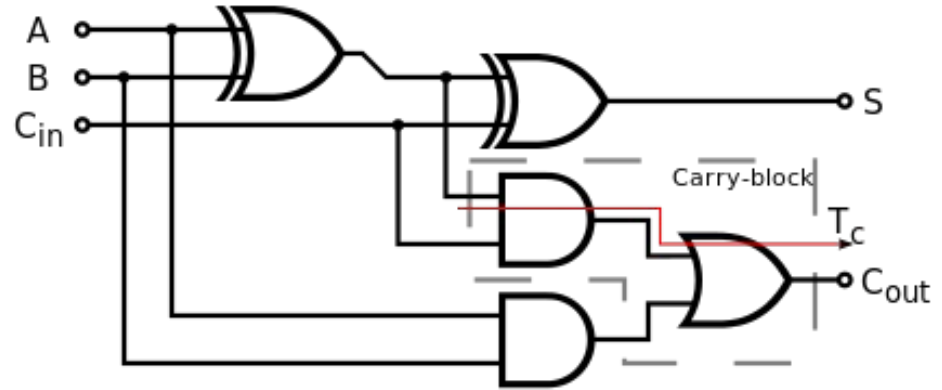


Circuits for adding numbers

What about subtraction?



Half adder



Full adder

Note:

- The ways to build half adder and full adder are not unique.
- If the circuits represent the same truth table, they are equivalent.

Agenda

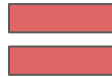
- Logic gates
- **Types of memory**
- Memory Circuits
- Memory Hierarchy
- Locality

Computer Memory

- Memory is a kind of circuits for storing data and instructions, like **a data warehouse**.
- There are various types of memory, adapting for different functions and specifications.

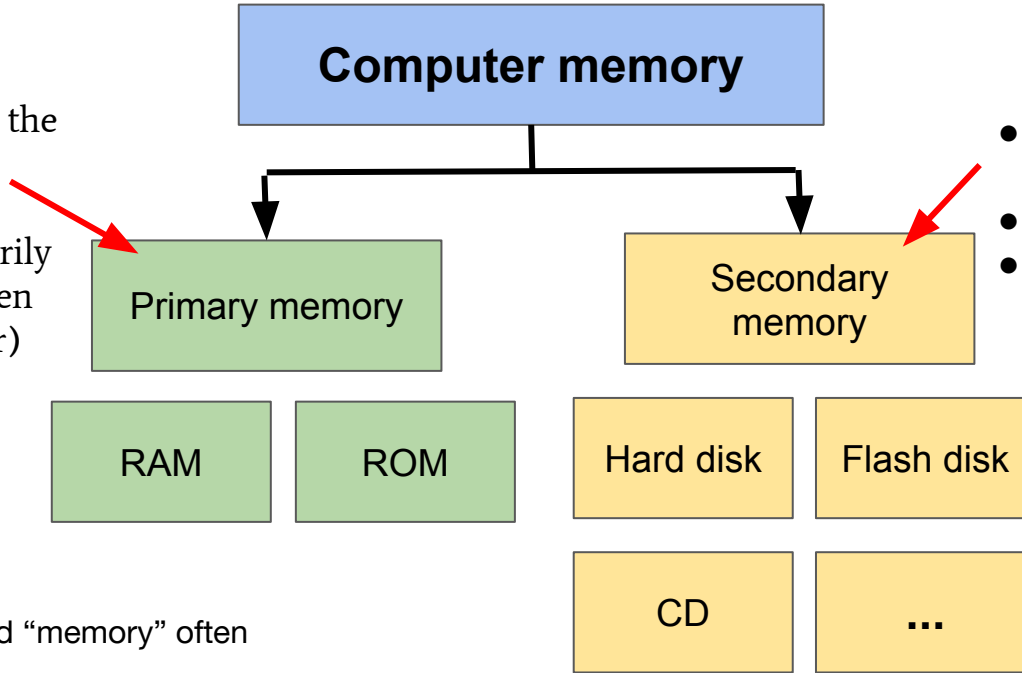


Not that different...



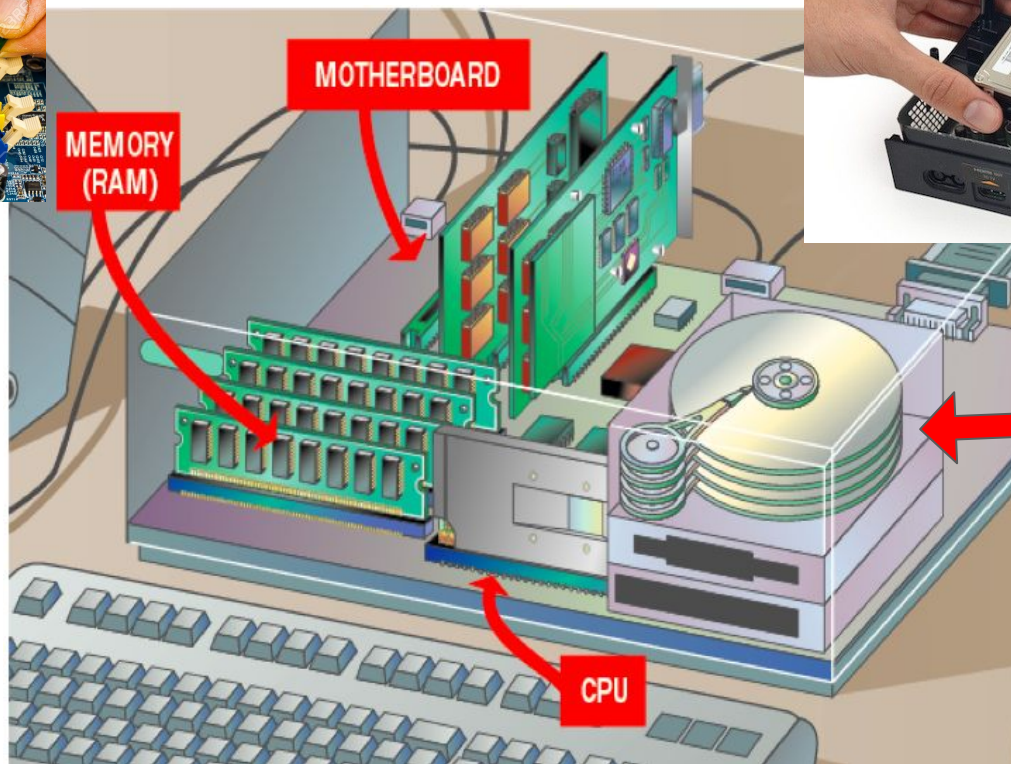
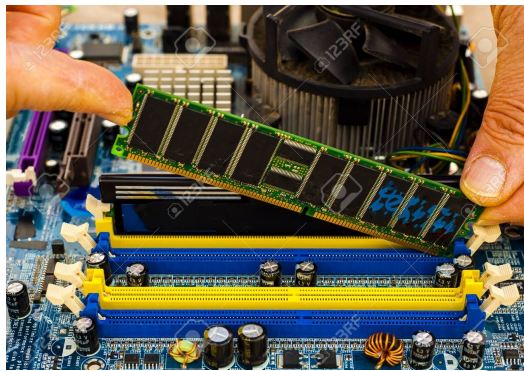
Memory types

- Immediate access by the processor
- Very fast
- Data stored temporarily (RAM loses data when cutting off the power)



- Not directly accessible by the CPU
- Slow
- Data stored permanently

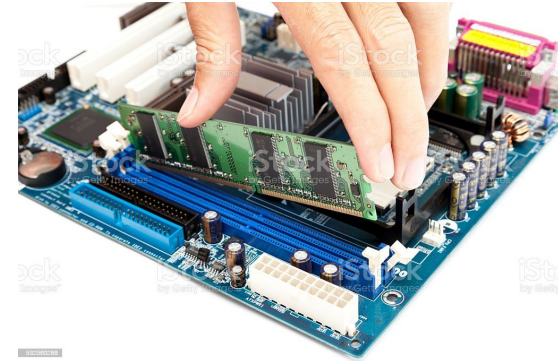
Note: in practice, the word “memory” often refers to the RAM.



Hard Drive

Primary memory

- RAM (Random access memory):
 - Being accessed by the processor when needed (that's why it is called “random access”);
 - **Volatile** (Data will be erased once power is off);
 - Storing data (and functions) for temporary use
 - E.g., the operating system is a “function” that helps you manipulate your computer. Every time you turn on your machine, it will be loaded into the RAM.



In the early days, computer memory could **not** be accessed randomly. (Magnetic tapes were read **sequentially**.)

UNIVAC TAPE DRIVE



Primary memory

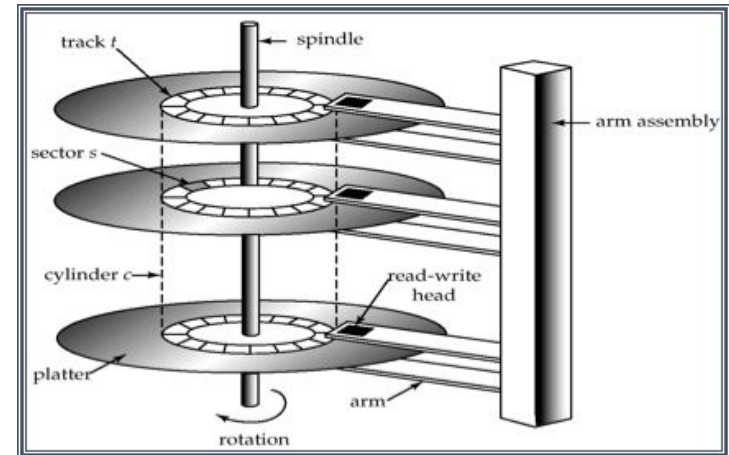
- ROM (Read only memory):
 - Data can only be accessed and read, no overwritten, or modified (today some ROM can be modified)
 - **Nonvolatile**, a permanent storage of data (regardless of power supply)
 - Used as bootable devices (e.g., BIOS)



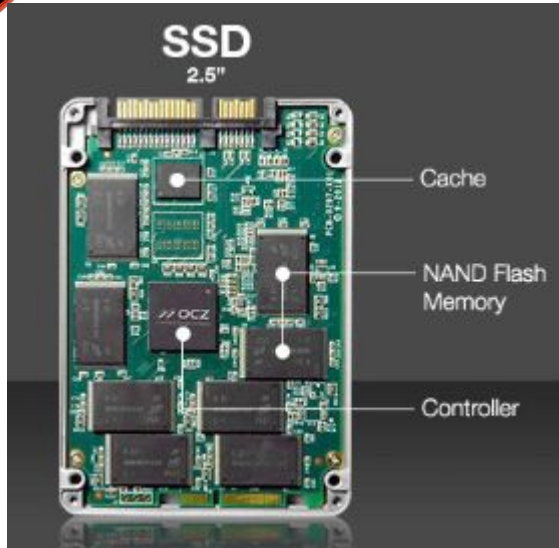
Secondary memory: Magnetic disk

Hard Disk Drive (HDD)

- Magnetic coating on platter to store the data
- Files are stored in sectors (usually, one file in consecutive sectors)
- Loading a file → Random access, needs to move the read-write head (slow)
- Cheap; good for massive storage



Secondary memory: Flash drive and SSD



- SSD (Solid state drive) use NAND flash memory to store data.
 - Much faster than HDD
 - Not reliable for long-term archiving

Flash memory stores information by sending electronic signals to the storage medium and electrons are trapped within the isolator (tiny chamber of silicon dioxide). Data will be retained after power is off.

Comparison: RAM and Hard Disk

	RAM	Disk
Capacity	~GB	~TB (x1000 bigger)
\$ per Gigabyte	~\$8	~\$0.04 (x200 cheaper)
Access time	~100ns	~10ms (x10k slower)
Access pattern	Random	Random (but read a big chunk is much faster)
Durability	Volatile: gone if power is off	Nonvolatile

Agenda

- Logic gates
- Types of memory
- **Memory Circuits**
- Memory Hierarchy
- Locality

Memory Circuits

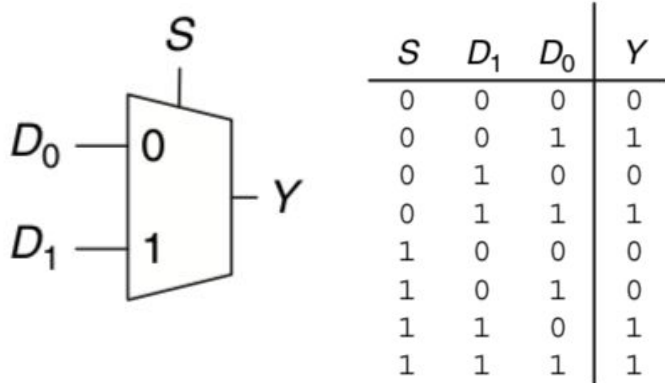
Several useful building blocks

- Mux: a selector
 - A circuit that can choose one of its inputs as the output.
- Decoder: a lookup table
 - A circuit that has N inputs and 2^N outputs, and it asserts exactly one of its outputs depending on the patterns of the inputs.
- Flip-flop: a kind of “memory”
 - A circuit that can “remember” its last state.

Multiplexer (Mux)

- It is a selector.
- choose an input as the output according to the control signal S.

2:1 Multiplexer and its truth table



There are two inputs D_0 and D_1 , a control input S , and one output Y .

The mux chooses between D_0 and D_1 based on the values of S .

- If $S = 0$, $Y = D_0$
- If $S = 1$, $Y = D_1$

Decoder

A decoder has N input and 2^N outputs.

- a “lookup table”
- each pattern of inputs associates to an output

2:4 decoder and its truth table

		2:4 Decoder							
				A_1	A_0	Y_3	Y_2	Y_1	Y_0
A_1 A_0	11			0	0	0	0	0	1
	10			0	1	0	0	1	0
	01			1	0	0	1	0	0
	00			1	1	1	0	0	0

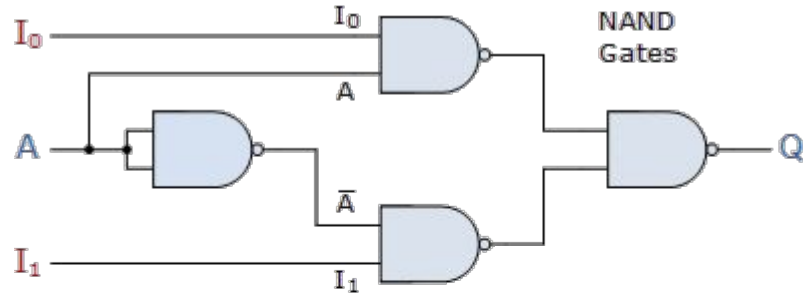
In the 2:4 decoder,

- when $A_{1:0} = 00$, $Y_0 = 1$
- when $A_{1:0} = 01$, $Y_1 = 1$
- when $A_{1:0} = 10$, $Y_2 = 1$
- when $A_{1:0} = 11$, $Y_3 = 1$

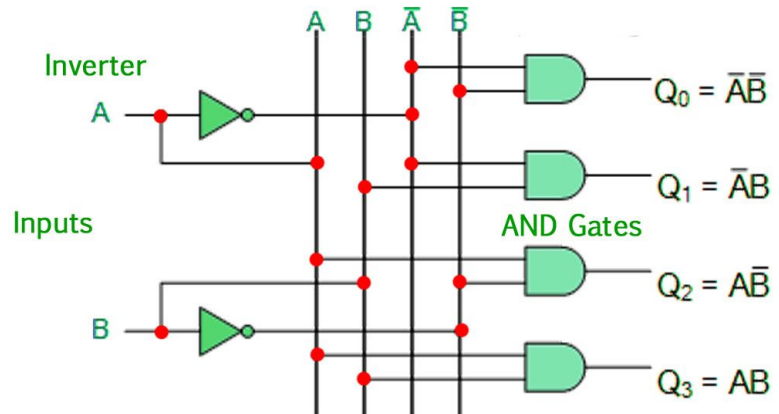
Supplementary page: to be skipped

- How are multiplexer and decoder implemented by logic gates?

- Mux/selector:



- Decoder/lookup table:



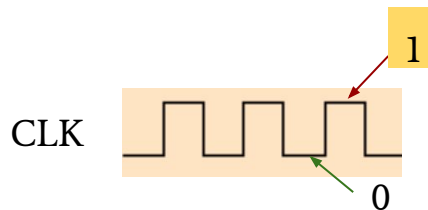
The clock signal

- A clock signal is a signal that **swings** between a **high** and a **low** state.
- It defines the “time” in a computer, and so different circuit components can coordinate with each other.



Intel® Core™ i9-9900KS Processor (16M Cache, Up to 5.00 GHz)

- 16 MB Intel® Smart Cache
- 8 Cores
- 16 Threads
- 5.00 GHz Max Turbo Frequency
- 9th Generation



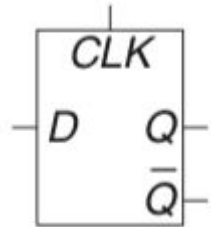
<https://www.intel.com/content/www/us/en/gaming/resources/cpu-clock-speed.html>

D latch (a special selector)

D latch is a circuit with two inputs: D and CLK.

- D is the *data* input, controls the state (i.e., the value) of the output Q.
- CLK is the *enable* input, controls when Q should change.

D latch symbol



Truth table of D latch

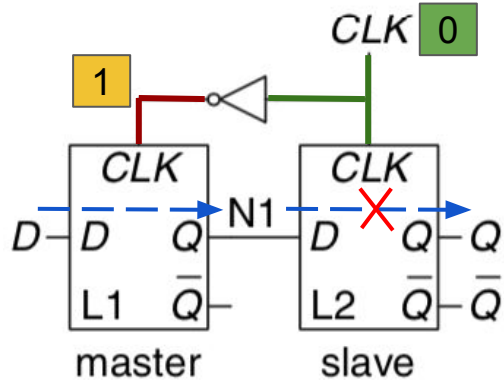
Case	CLK	D	Q
I	0	X	Q_{previous}
II	1	0	0
III	1	1	1

When **CLK = 0**, the latch is **opaque**; whatever D is, Q keeps its previous state.

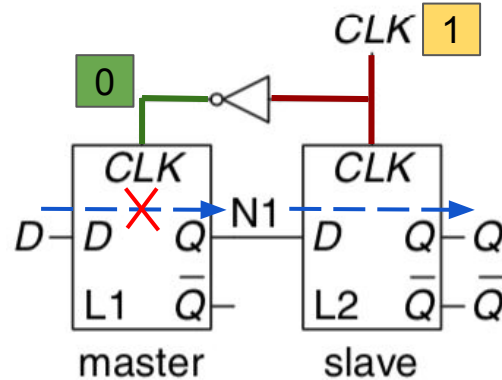
When **CLK = 1**, the latch is **transparent**; Q is changed to D.

Flip-flop

A flip-flop is built from two D latches, controlled by a common clock signal (CLK).



- When $CLK = 0$,
 - the master latch is transparent and the slave is opaque.
 - D goes through to $N1$.



- When $CLK = 1$,
 - the master goes opaque and the slave becomes transparent.
 - D is cut off from $N1$
 - $N1$ goes through to Q .

The output of a flip-flop at $CLK=1$ is its input at $CLK=0$.

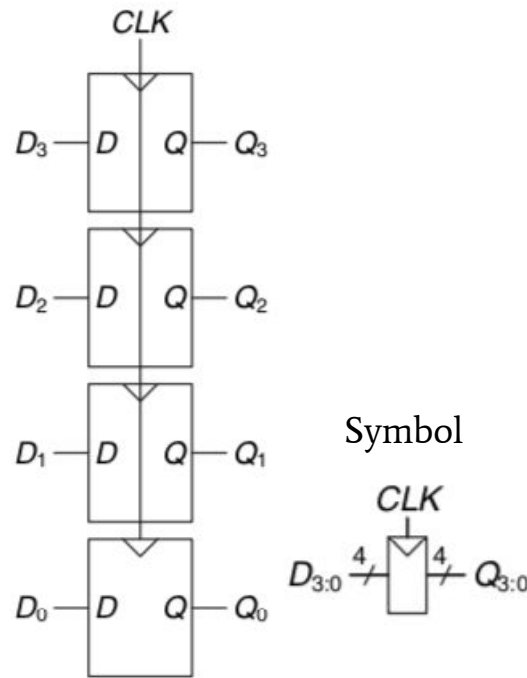
Registers (memory inside the CPU)

A register is a bank of flip-flops that share a common CLK input.

- All bits of the registers are updated at the same time.
- If there is N flip-flops, the register is call N-bit register.
 - e.g., if a register has 32 flip-flops, we call it 32-bit register; the length of its input/output is a 32-bit long binary string.

Registers are components of the CPU.

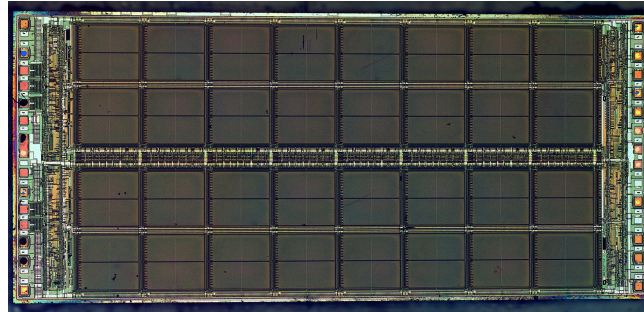
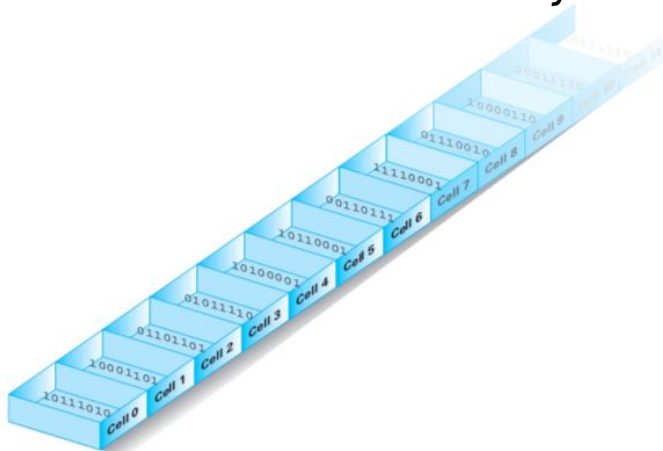
- A CPU is equipped with a group of registers
- A 32-bit processor means its registers are 32-bit registers.
- Processors can rapidly access data stored in registers (They work at the same frequency).



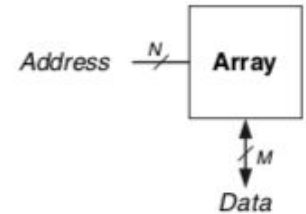
A 4-bit register

Let's have look at RAM

- The memory is organized in cells.
 - a cell contains some bits (usually, 8 bits)
- Each cell has a unique address.
 - We locate the cell by its address
 - Address is a binary string



Symbol of memory array



How does RAM work?

The decoder finds the cell of the input address. (i.e., turn on the word line accordingly)

Address $\xrightarrow{2}$

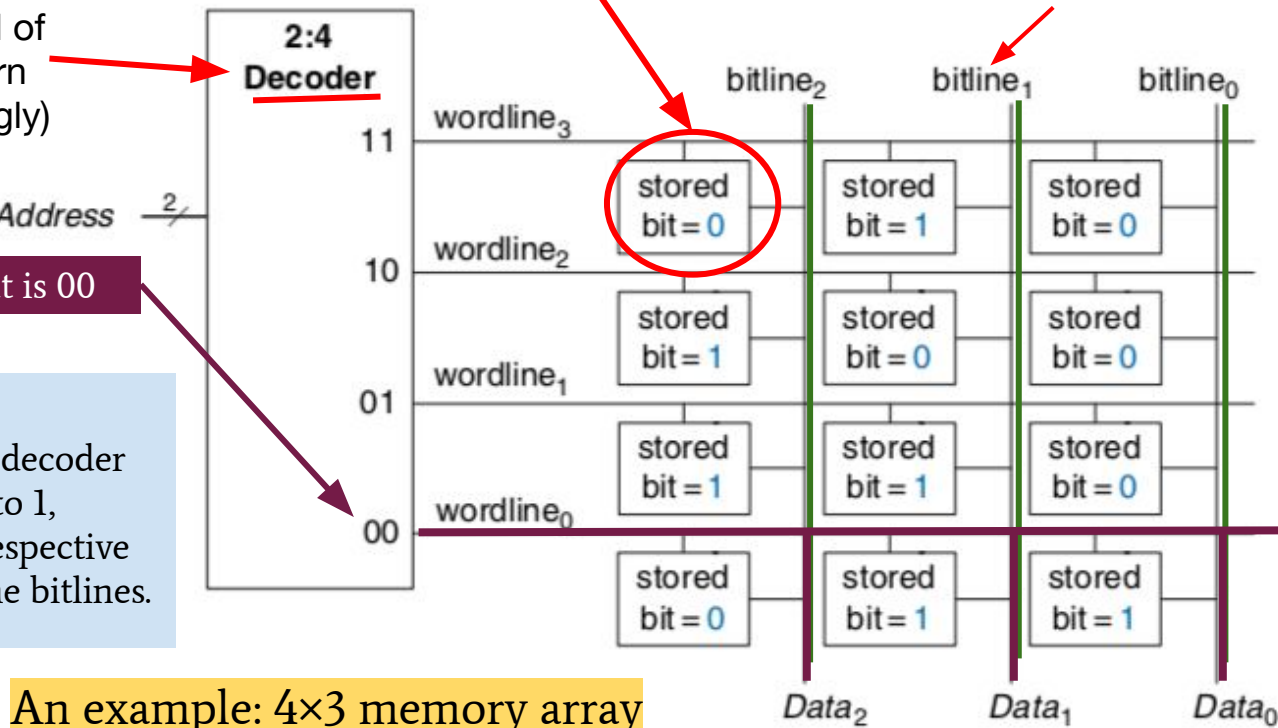
When input is 00

Reading the memory at 00:

When the input address is 00, the decoder will set its output at “wordline 0” to 1, which activates the bit cells in the respective rows. The stored bit transfers to the bitlines.

A bit cell stores 1 bit of data.

Bitlines are connected to the motherboard



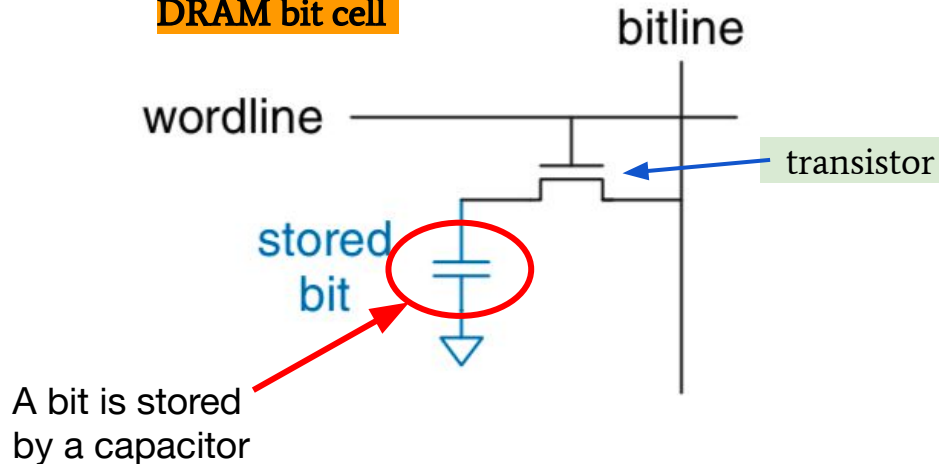
An example: 4×3 memory array

Two types of RAM

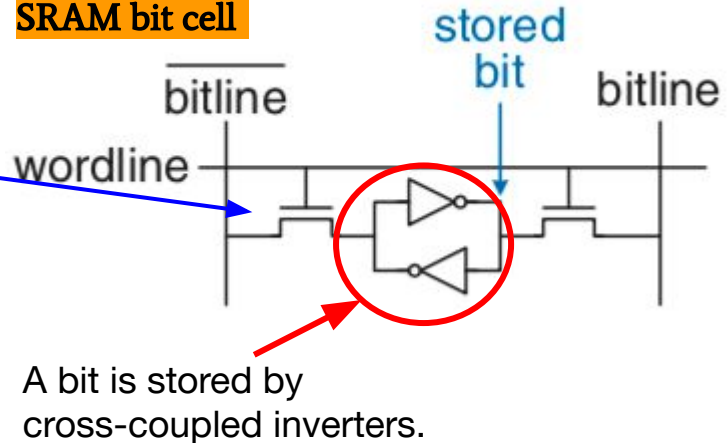
There are two major types of RAM: dynamic RAM (DRAM) and static RAM (SRAM).

- The **difference** is the circuit design of the bit cell.

DRAM bit cell



SRAM bit cell



RAM comparison

Memory Type	Transistor per Bit cell	Latency
Flip-flop in Register	~20	fast
SRAM	6	medium
DRAM	1	slow

The more transistors a device has, the more area, power, and cost it requires.

- Price: Flip-flop > SRAM > DRAM
- Speed: Flip-flop >> SRAM > DRAM

Accessing data in the memory (by the address)

- Data objects are stored in the memory cells (RAM)
- Memory cells have unique addresses (usually a very long number)
- (In programming) a variable \Rightarrow the memory address
 - A variable can be any type of data, e.g., int/float/list/tuple/function

```
In [24]: a = 1.2
```

```
In [25]: id(a)
```

```
Out[25]: 140318739206544
```

```
In [26]: b = [1, 2, 3]
```

```
In [27]: id(b)
```

```
Out[27]: 140318493164096
```

```
In [28]: def foo():  
...:     print('bar')  
...:
```

```
In [29]: id(foo)
```

```
Out[29]: 140318219064656
```

```
In [30]: c = foo
```

```
In [31]: id(c)
```

```
Out[31]: 140318219064656
```

- `id()`: the Python built-in function to show the ID of the variable, which is equal to the memory address.

Mutable and Immutable data types in Python

- immutable → the memory cell of the data is locked; modification is not allowed
- mutable → the memory cell is unlocked; you may change the values in the cell

```
In [1]: a = 1
```

```
In [2]: id(a)
```

```
Out[2]: 4311873888
```

```
In [3]: a = 2
```

```
In [4]: id(a)
```

```
Out[4]: 4311873920
```

a is immutable; if you change its value, Python will virtually associate it with another cell.

```
In [5]: a = [1]
```

```
In [6]: id(a)
```

```
Out[6]: 140318218561728
```

```
In [7]: a[0] = 2
```

```
In [8]: a
```

```
Out[8]: [2]
```


```
In [9]: id(a)
```

```
Out[9]: 140318218561728
```

a is mutable; we can modify the content in the cell.

A list inside a tuple

```
In [7]: tup = ("ICS", 2022, [1, 2, 3])
In [8]: tup[2][0] = 4
In [9]: tup[2][1] = 5
In [10]: tup[2][2] = 6
In [11]: tup
Out[11]: ('ICS', 2022, [4, 5, 6])
```



What happens:

- A tuple/list stores the addresses of the elements, not their exact values.
- By calling `tup[0]`, we visit the memory cells that store the “ICS”.
- `tup[2]` is a list, so, we can modify it.

- Mutable data types like channels, which allow us to “share” data inexplicitly. (like a magic teleporting door)

What will be printed?

```
def count_one(c=[0]):  
    c[0] += 1  
    return c[0]
```

```
print(count_one())  
print(count_one())  
print(count_one())
```

- To see why, you may insert the following inside the function

```
print("id of c:", id(c))
```

Modifiers and Pure Functions

- Functions which takes mutable variables as arguments and change them during execution are called **modifiers**, and the changes they make are called **side effect**
- A **pure function** does **not** produce side effects.

Agenda

- Logic gates
- Types of memory
- Memory Circuits
- **Memory Hierarchy**
- Locality

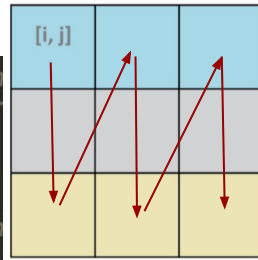
Memory hierarchy

Two ways to sum up the elements in a 2d array

Case I:

```
import numpy as np # package for generating random arrays
import timeit # package for counting the time

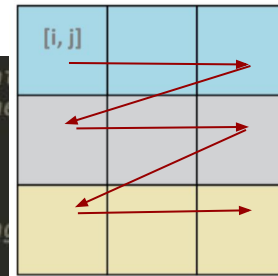
array_2d = np.random.rand(1000, 1000)
s = 0
## Case I: we sum up the elements along the columns
start = timeit.default_timer()
for i in range(1000):
    for j in range(1000):
        ## we fix the column index, change the row index
        s += array_2d[j,i]
stop = timeit.default_timer()
print("Case I")
print("Time for summing all elements:", stop-start)
```



Case II:

```
import numpy as np # package for generating random arrays
import timeit # package for counting the time

array_2d2 = np.random.rand(1000, 1000)
s = 0
## Case II: we sum up the elements along the rows
start = timeit.default_timer()
for i in range(1000):
    for j in range(1000):
        ## We fix the row index, change the column index
        s += array_2d2[i,j]
stop = timeit.default_timer()
print("Case II")
print("Time for summing all elements:", stop-start)
```

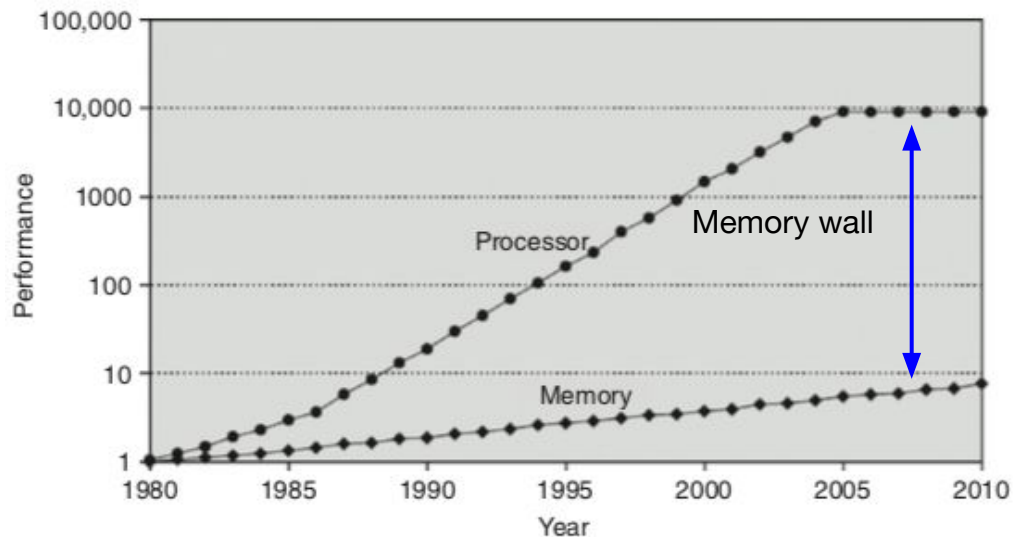


- Please run the codes on your laptop; which one takes more time? (Download them from Brightspace)
- The order to loop the elements matters → But RAM is randomly accessible. Why the time is different?

Memory wall: speed gap between CPU and memory

“Memory wall”: the growing disparity of speed between CPU and memory outside the CPU chip.

- It is the bottleneck that slows down the speed of computers.



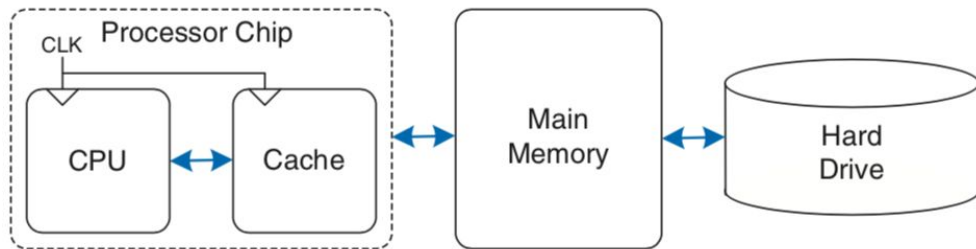
Diverging processor and memory performance: DRAM speed has improved 7%/year, whereas processor performance has improved at a rate of 25% to 50%/year.

How to overcome the memory wall

Trade-off among speed, size and price:

- Fast memory → storing the **copy** of data that will be used in the near future
- Slow memory → storing the data that may not be needed recently

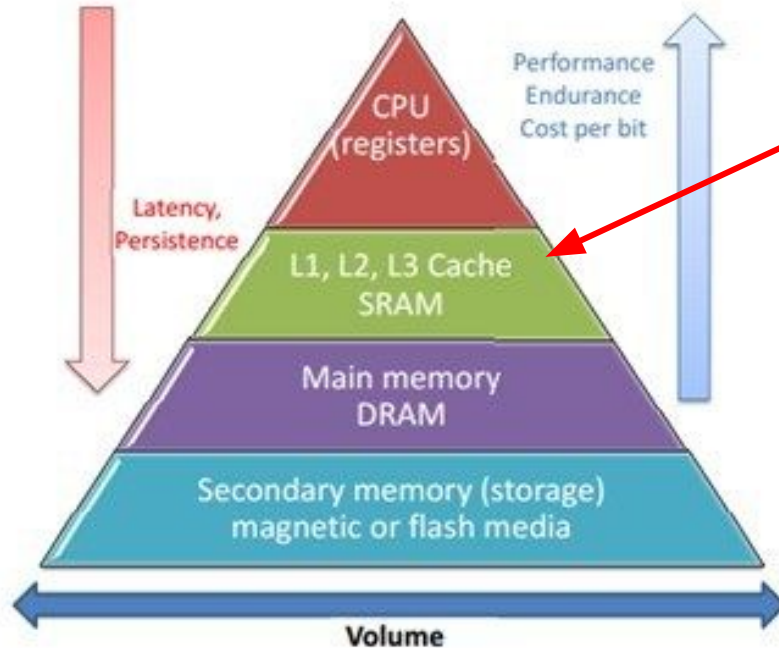
As a result, memory in our computers are built in a hierarchical structure



CPU searches data according to the following path:

CPU → cache → main memory → hard drive.

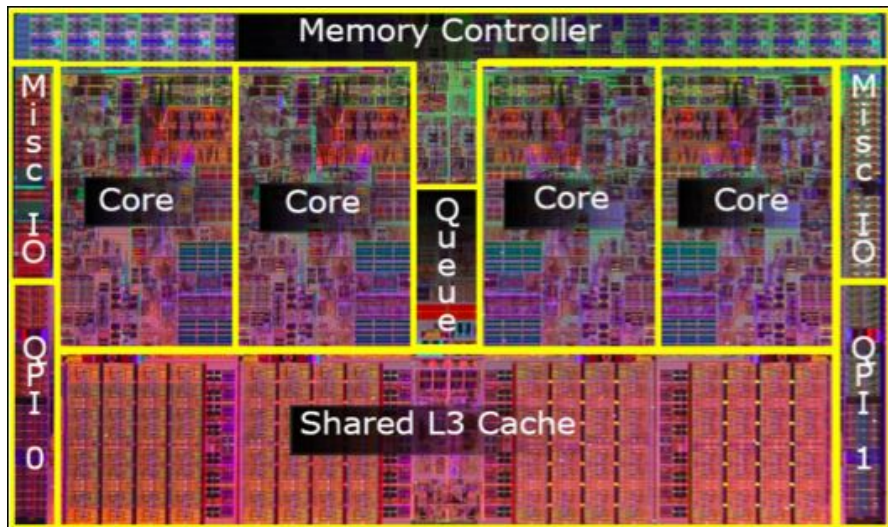
Memory hierarchy



Cache: a faster but smaller memory that built out of SRAM on the same chip as the processor.

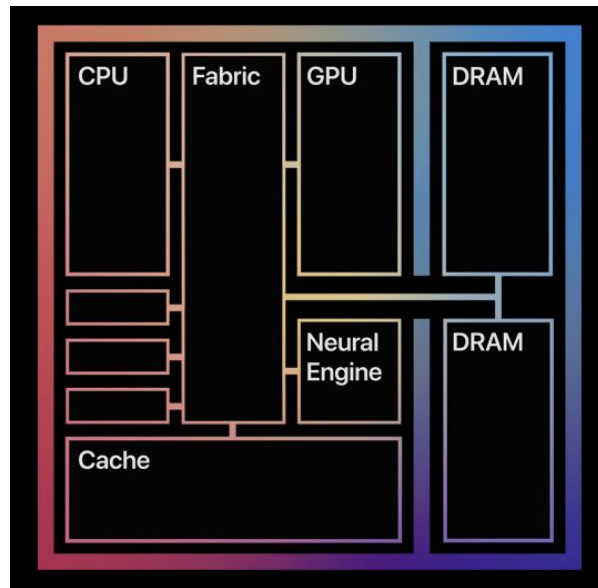
- Copying data used in near future into cache **in advance** can save CPU's memory access time

Examples of modern cores



Intel Nehalem:

- ON-chip cache resources:
 - For each core: L1: 32K instruction and 32K data cache, L2: 1MB
 - L3: 8MB shared among all 4 cores
- Integrated, on-chip memory controller (DDR3)



Apple silicon M1:

Unified memory shared across the entire system.

But how can we know what data should be loaded into the cache? → some hardware constraints you need to know



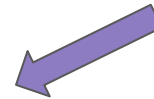
LADDER
MOVE UP

LADDER MANKAV GREENS

Railway Station - 1 km
MIMS Hospital - 1 km
Tali Temple - 2 km
Focus Mall - 4 km
R P Mall - 4 km
Kozhikode Beach - 3 km
Palayam - 3 km

**WHEN WE SAY
HEART OF THE CITY,
We mean it!**

Locality, locality, locality!



Principles of locality

- Temporal locality: recent accesses have shorter latency
- Spatial locality: nearby accesses have shorter latency

Your computers are designed to exploit these two forms of locality. Most data items are stored in RAM, but if a data item is recently used by a program


- it will be copied into the cache
- a few of its nearest neighbors (in the RAM) will be copied into cache

(These are educated guesses because such data items are highly likely to be used again in the next CPU clock cycle)

Letting your code access locality


- Temporal locality: recent accesses have shorter latency
 - **Program tips:** not all data is accessed equally often, keep those frequently accessed in inner loop

```
16 a = [0, 1, 2, 3, 4]
17 s0 = a[0]
18 s1 = s0 + a[1]
19 s2 = s1 + a[2]
20 s3 = s2 + a[3]
21 s4 = s1 + a[2] + a[3] + a[4]
```



Line 21 is an example that violates the temporal locality because `s1`, `a[2]`, and `a[4]` are not accessed most recently.

```
38 a = [0, 1, 2, 3, 4]
39 s = 0
40 for i in range(len(a)):
41     s += a[i]
```




Variable `S` is referenced in each iteration, so there is good temporal locality with respect to `S`.

Letting your code access locality


- Spatial locality: nearby accesses have shorter latency
 - **Program tips:** if you are accessing a big array (list), try to access them in order, avoid random access

```
38 a = [0, 1, 2, 3, 4]
39 idx = [1, 4, 0, 3, 2]
40 s = 0
41 for i in idx:
42     s += a[i]
```



This is a bad code example; elements are randomly picked, no use of spatial locality.

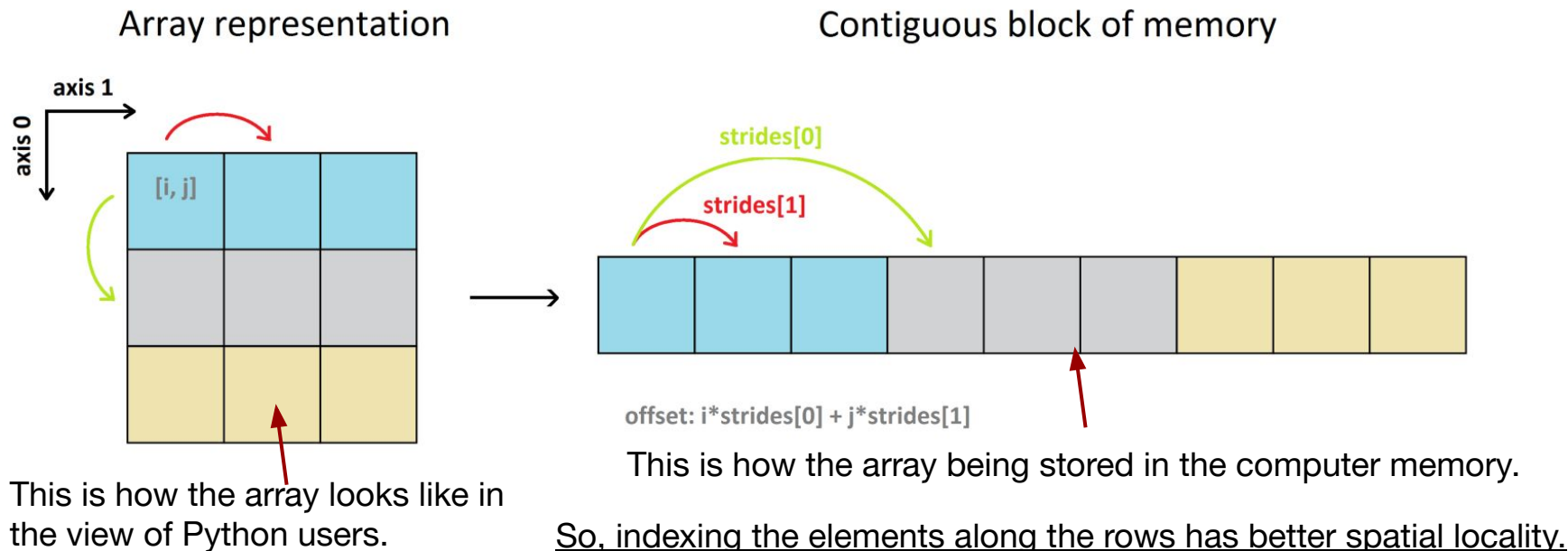
```
a = [x for x in range(0, 100)]
s = 0
for i in range(0, 100):
    s += a[i]
```



Good use of spatial locality; elements are accessed by incrementally +1

How a Numpy array is allocated in memory

Elements in a numpy array are stored along the rows, which means the location of an element is closer to its neighbors in row than those in column.



Maximize locality for better performance

There are two ways to sum up the elements in a 2d array.

Case I:

```
import numpy as np # package for generating the array
import timeit # package for counting the runtime

array_2d = np.random.rand(1000, 1000)
s = 0
## Case I: we sum up the elements along the column
start = timeit.default_timer()
for i in range(1000):
    for j in range(1000):
        ## we fix the column index, change the row index
        s += array_2d[j,i]
stop = timeit.default_timer()
print("Case I")
print("Time for summing all elements:", stop-start)
```

Case II:

```
import numpy as np # package for generating the array
import timeit # package for counting the runtime

array_2d2 = np.random.rand(1000, 1000)
s = 0
## Case II: we sum up the elements along the row
start = timeit.default_timer()
for i in range(1000):
    for j in range(1000):
        ## We fix the row index, change the column index
        s += array_2d2[i,j]
stop = timeit.default_timer()
print("Case II")
print("Time for summing all elements:", stop-start)
```

- Now, you should know why Case II is faster than Case I.
- Which program is better? (in the view of locality)

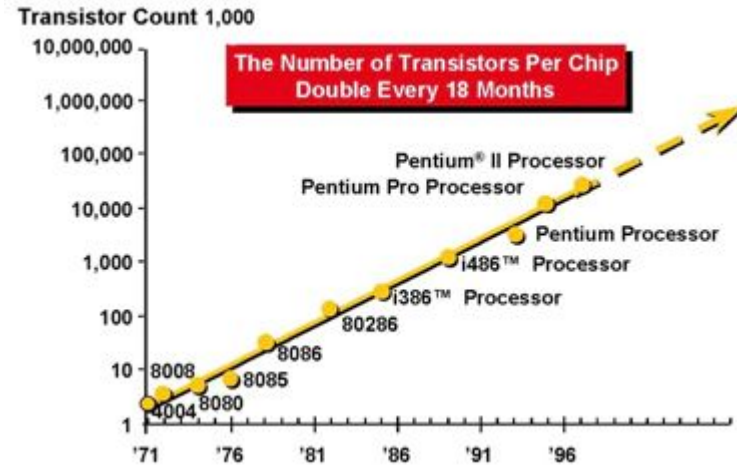
Appendix: The Moore's law



In 1965, Gordon Moore made a prediction that would set the pace for our modern digital revolution. From careful observation of an emerging trend, Moore extrapolated that computing would dramatically increase in power, and decrease in relative cost, at an exponential pace.

However, many people, including Gordon Moore, expect Moore's law will end by around 2025.

- Transistors eventually would reach the limits of miniaturization at atomic level.



Number of transistors on cost-effective integrated circuit double every 18 months.