Lecture 5 – Memory

Dr Tianxiang Cui

Outline

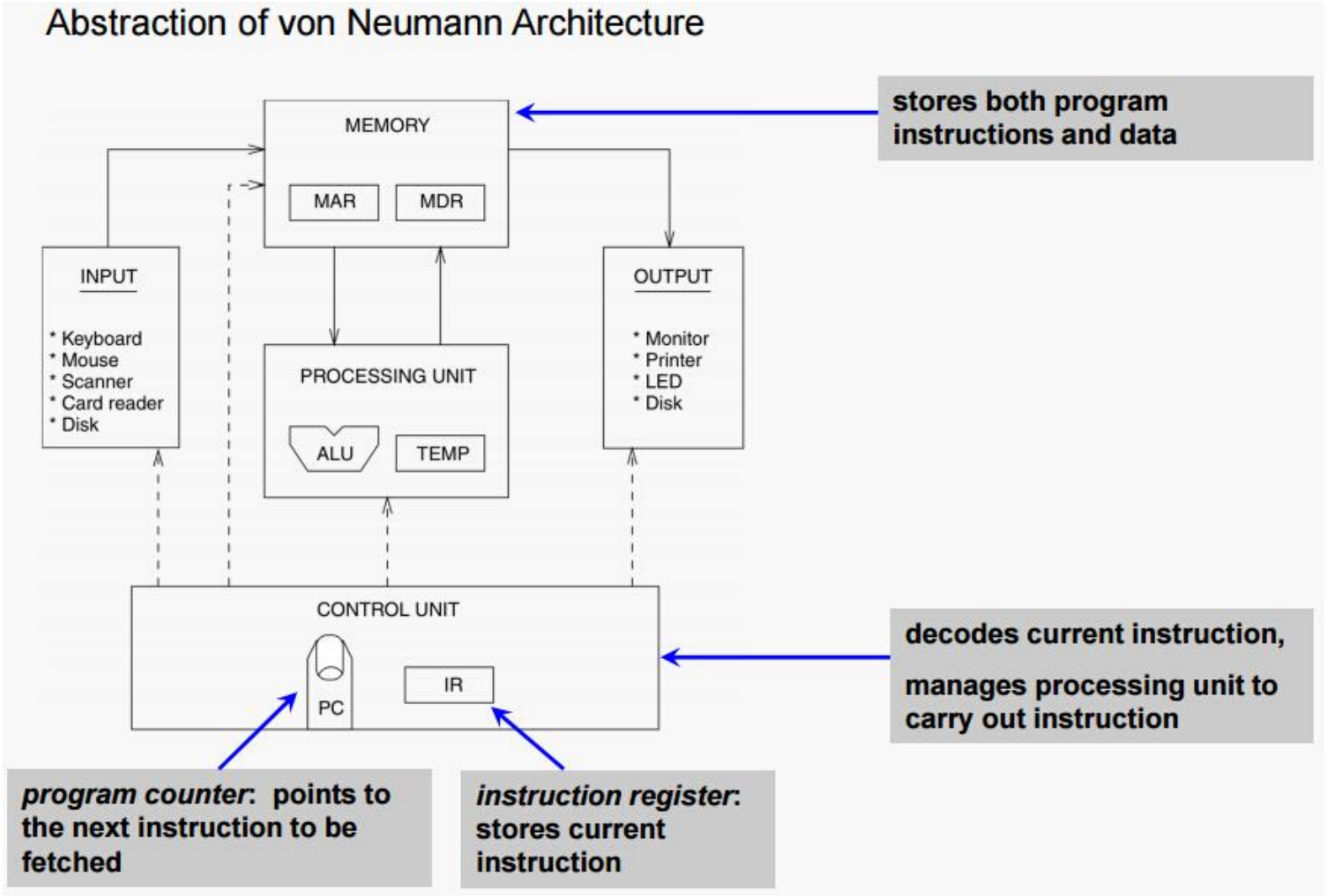
- Program Execution
- Cache
- Hack Computer
- CS Pioneers

Learning Outcome

- To be able to understand program execution process
- To be able to understand caching principle
- To be able to understand simple Hack computer architecture

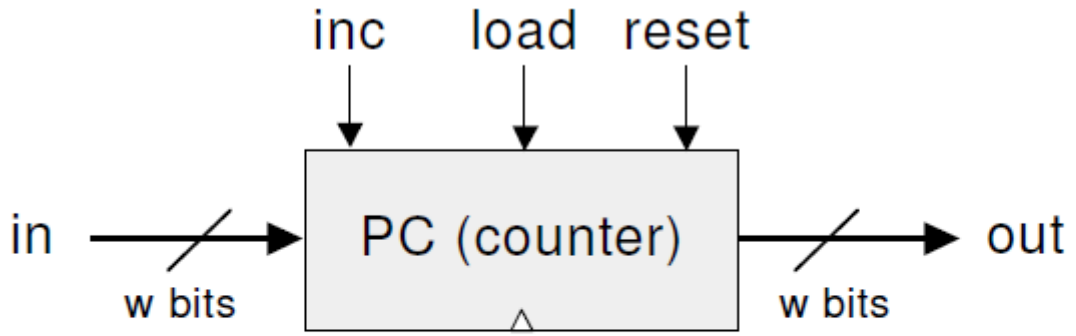
Program Execution

- A basic computer consists of a CPU, which runs programs, memory, a bus, and other I/O devices
- Memory stores programs, as well as data
- Thus, programs must go from memory to the CPU where it can be executed
- Programs consist of many instructions which are the 0's and 1's that tell the CPU what to do
- The format and semantics of the instructions are defined by the Instruction Set Architecture (ISA)
  - 80x86, MIPS, ARM, PowerPC



Program Counter

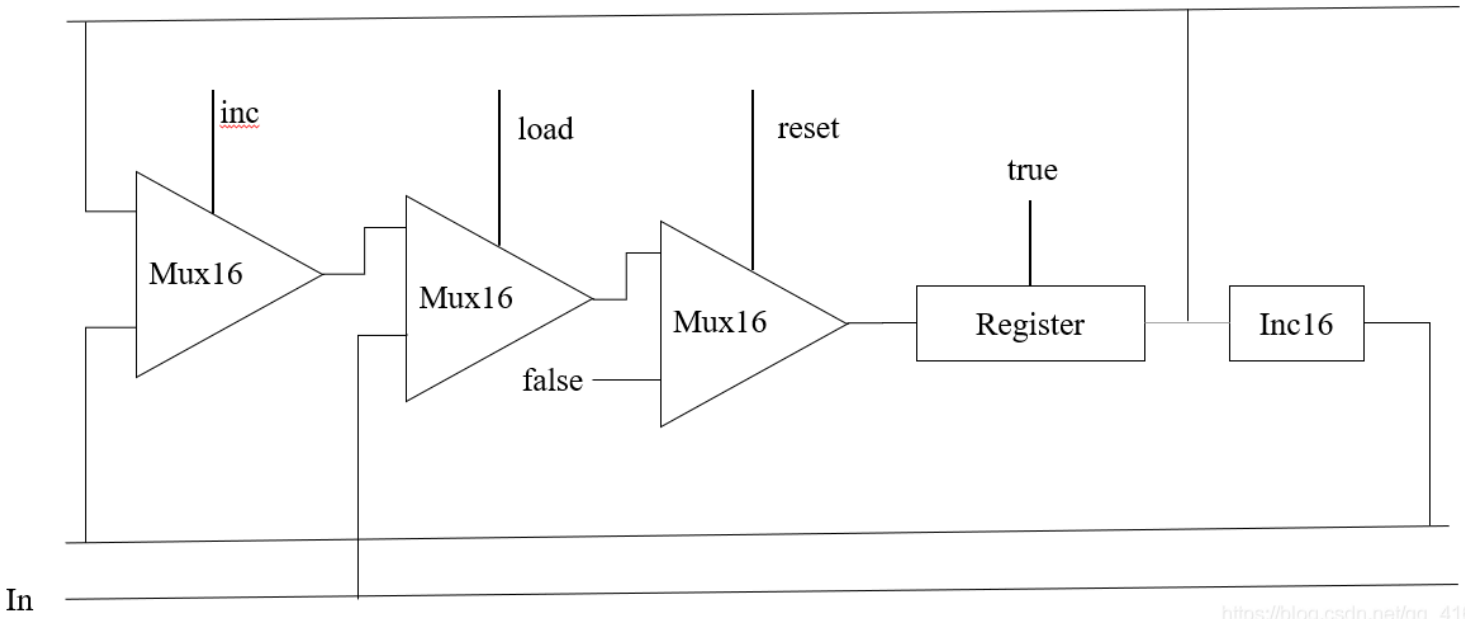
- Program counter emits the address of the next instruction.
  - To start/restart the program execution: PC=0
  - No jump: PC++
  - Unconditional jump: PC=A
  - Conditional jump: if (cond.) PC=A else PC++
- Implementation requirements:
  - Counting
  - Put counter in correct working mode based on the 3 different control bits




```
If reset(t-1) then out(t)=0
   else if load(t-1) then out(t)=in(t-1)
        else if inc(t-1) then out(t)=out(t-1)+1
             else out(t)=out(t-1)
```

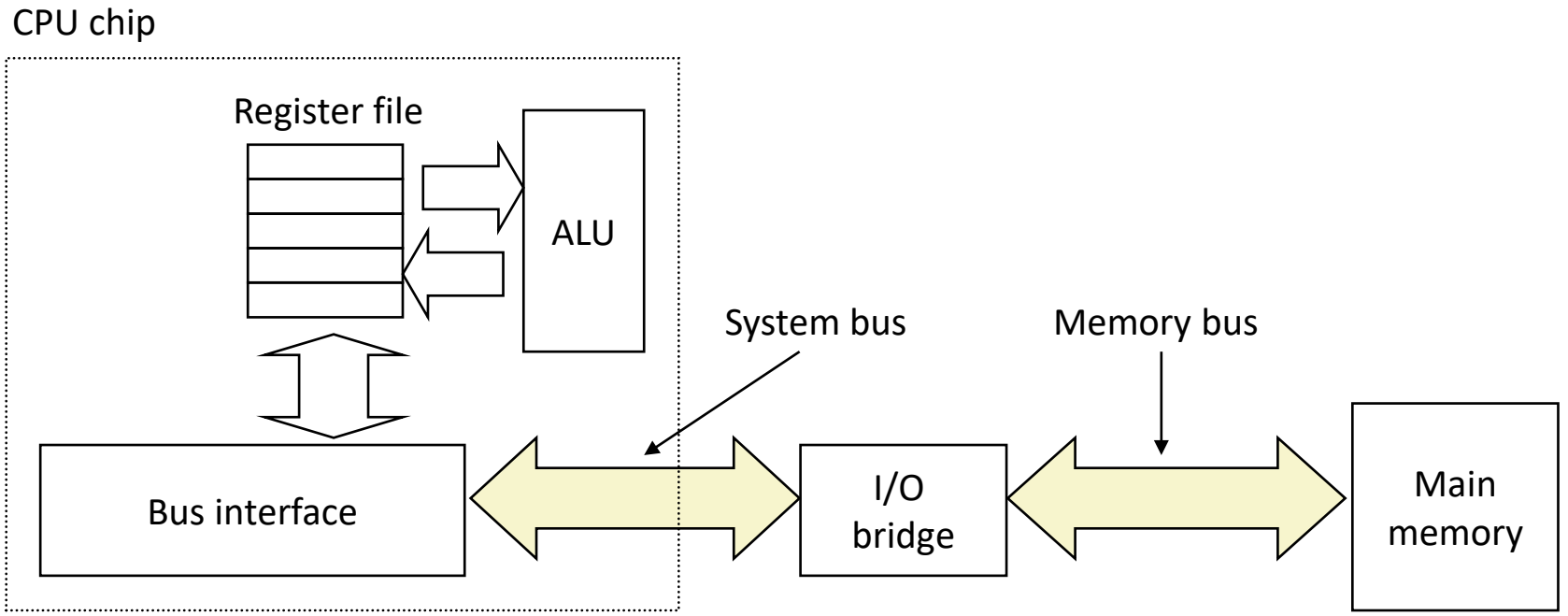
Program Counter

- inc:
  - determine whether to perform the increment operation
  - inputs: the data before/after increment
- load:
  - determine whether to load the input data
  - inputs: in, out from Mux16(inc)
- reset:
  - determine whether to reset the data (0)
  - inputs: false, out from Mux16(load)
- register:
  - load the memory address of instruction to be executed next



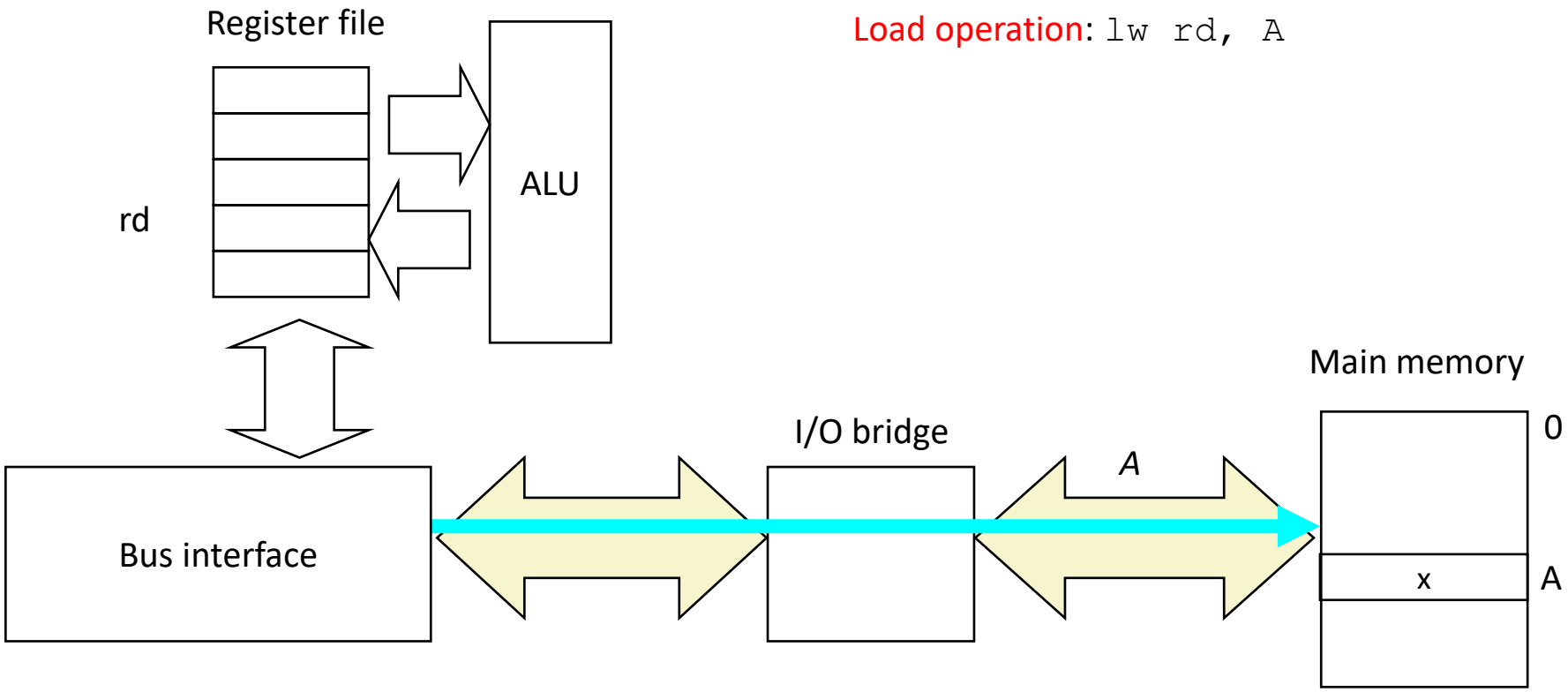
Bus Structure

- A bus is a collection of parallel wires that carry address, data, and control signals
- Traditional bus structure is used to connect CPU and memory
- Buses are typically shared by multiple devices



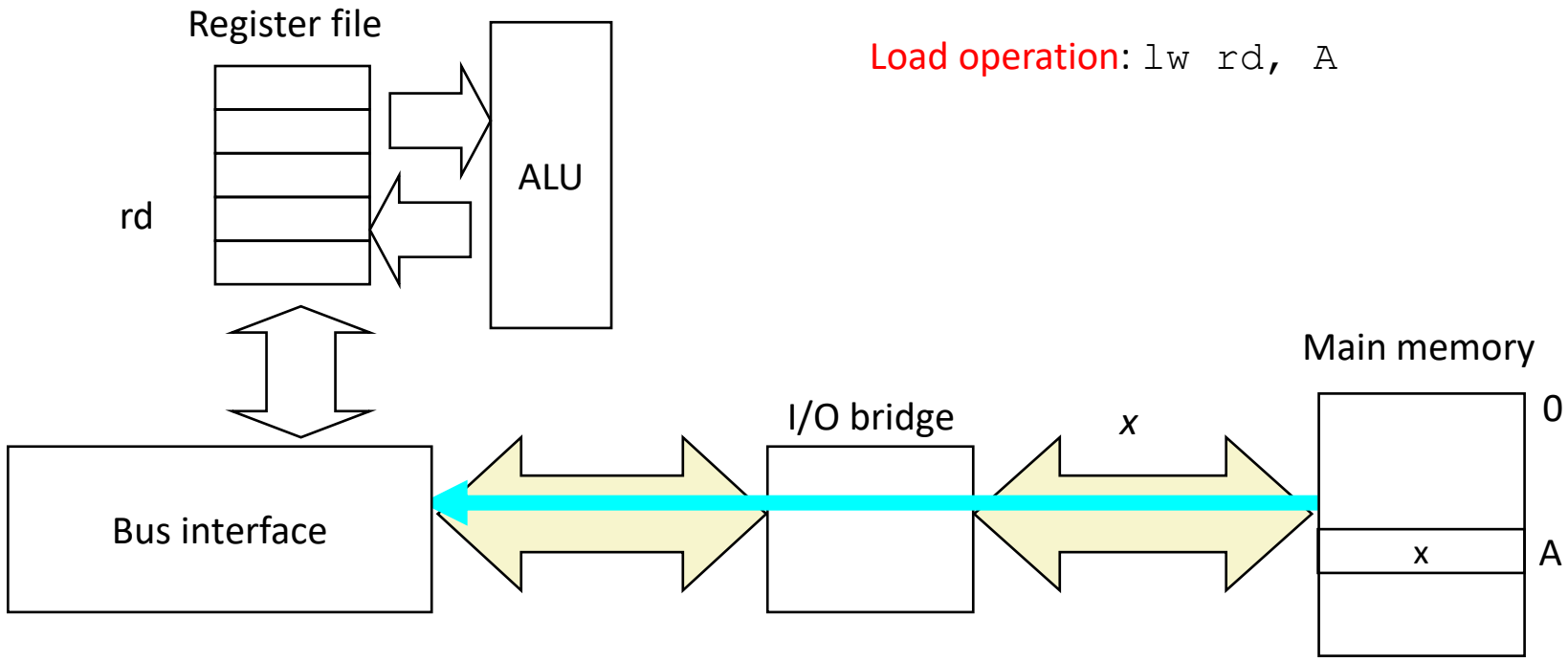
Memory Read Transaction

- CPU places address A on the memory bus



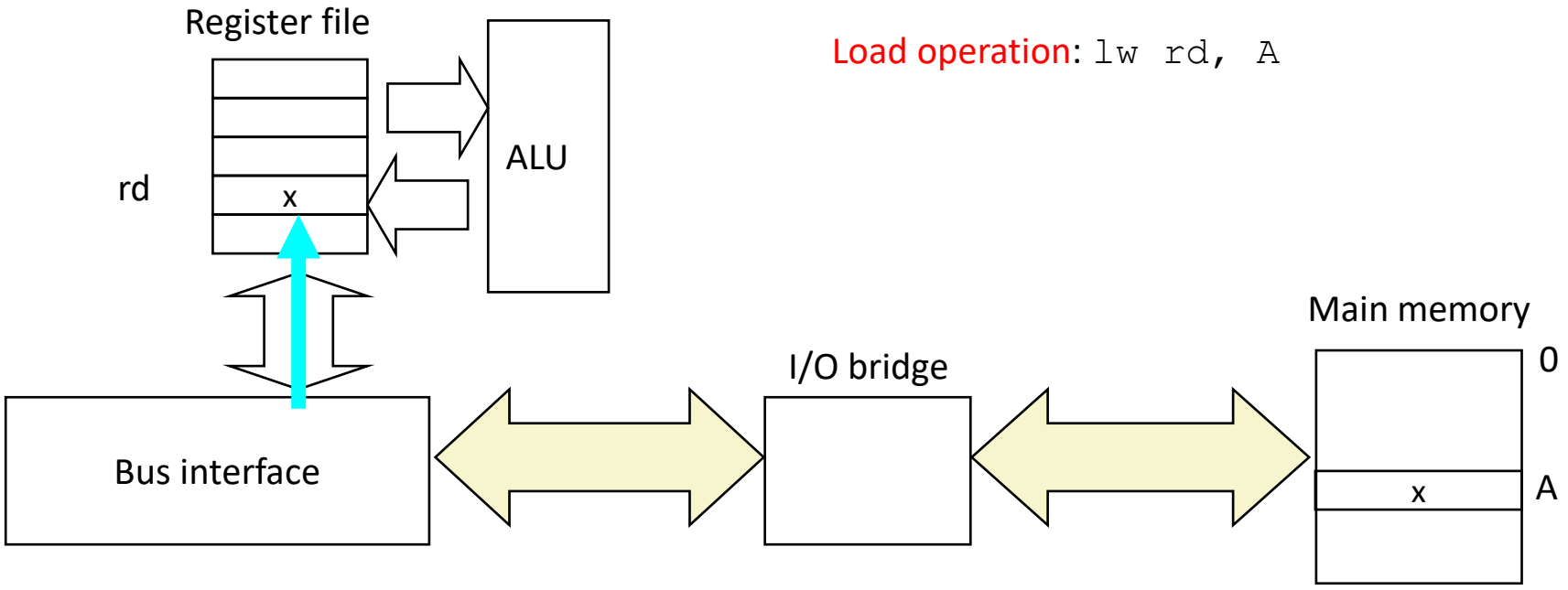
Memory Read Transaction

- Main memory reads A from the memory bus, retrieves word *x*, and places it on the bus



Memory Read Transaction

- CPU read word *x* from the bus and copies it into register rd.



Fetch-Decode-Execute Cycle

- At some level, every programmable processor implements a **fetch-execute cycle**
- Automatically implemented by processor hardware, allows processor to move through program steps
- Fetch — The opcode for the instruction is fetched from memory
- Decode — Opcode decoded to work what parts of the CPU are needed
- Execute — CPU processes the instruction
- And repeat for the next instruction

Fetch-Execute Algorithm

Repeat {

Fetch (PC):

- Fetch the instruction word (at PC)
- Instruction decoded
- Calculate next instruction address

Execute (ALU, Registers and Control):

- Read operands
- Executes the operations
- Write/store results

}

Fetch

- The instruction is fetched from (physical) memory
  - i.e. RAM
- Here are some details:
  - The address of the instruction to be fetched defined by the PC (program counter)
  - Memory system places the required value onto the data bus
  - The instruction is copied from memory to IR (instruction register)
  - Loaded into instruction cache
    - Cache is a (very) fast memory

Program Counter	Instruction Register
Holds the **memory address** of the next instruction to be executed	Holds the **actual instruction**
After fetching a instruction the address value **is increased by 1**	After fetching a instruction, it **contains the next instruction**
A CPU **locates** the instructions in the memory from here	A CPU **decodes and executes** the instructions from here

Decode

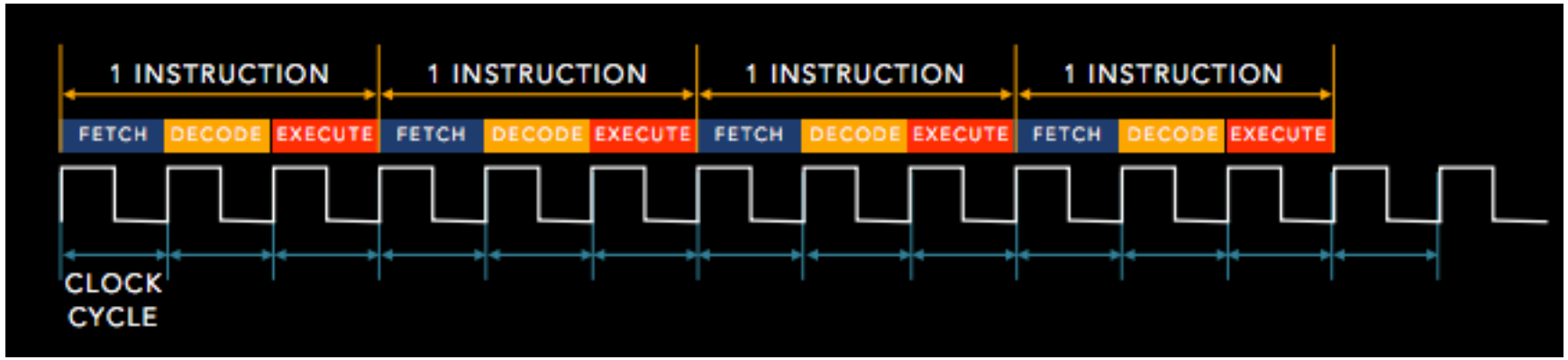
- In order to figure out what the instruction should do, it needs to be decoded
- Hardware in the CPU analyses the opcode
- Part of the decoding process fetches the input operands
  - E.g, if you have an add instruction (opcode) that adds the contents of register 1 and 2, and places the result in register, then the values of register 1 and 2 need to be fetched to perform the addition
- Also need to decide which parts of the CPU need to be engaged to process instruction and in which order

Execute

- Executes the steps need for the specific instruction
- It could take more than one step
- Might need to access memory again

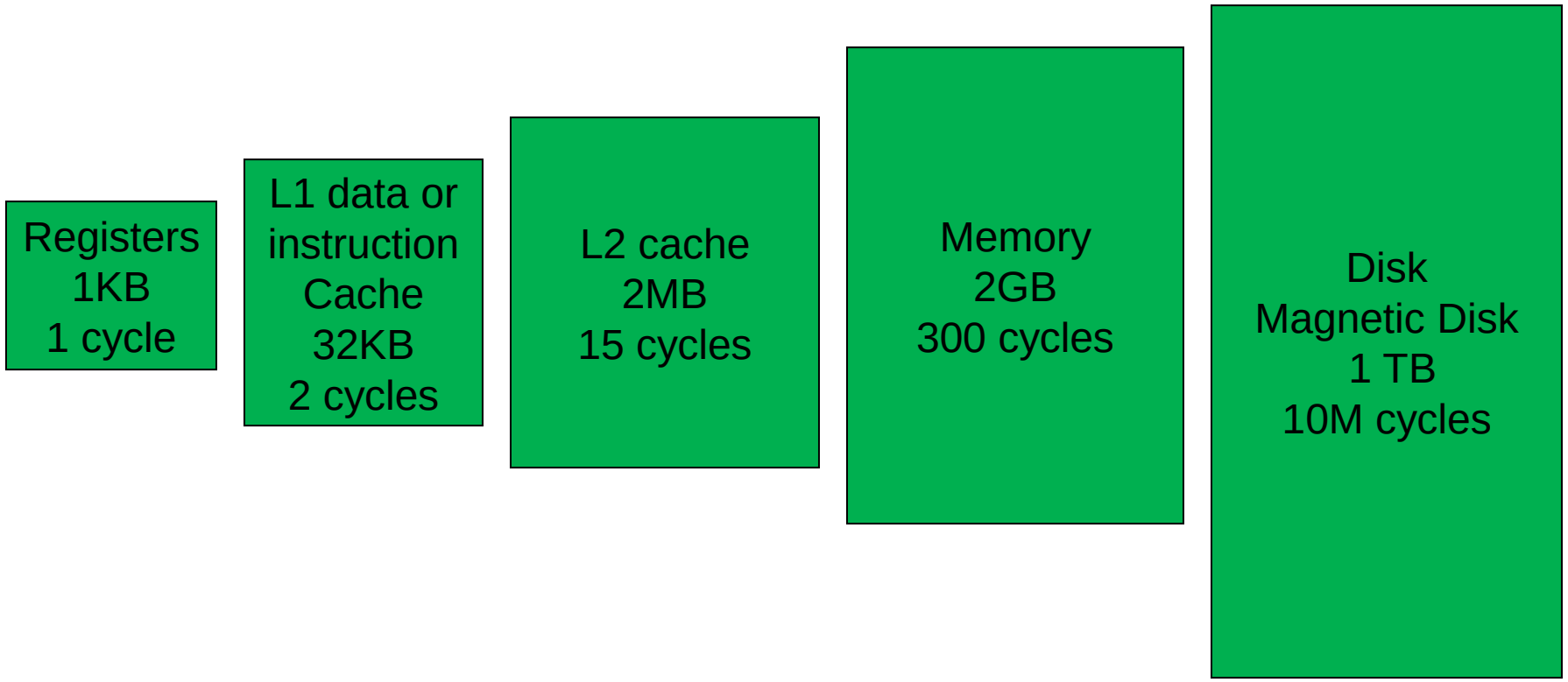
Fetch-Decode-Execute Cycle

- Each of these steps are synchronized to the CPU's clock cycles
- Let's assume that each step takes one cycle and happens in series
  - Fetch takes one cycle
  - Decode takes one cycle
  - Execute takes one cycle
- What is the minimum time to execute an instruction?



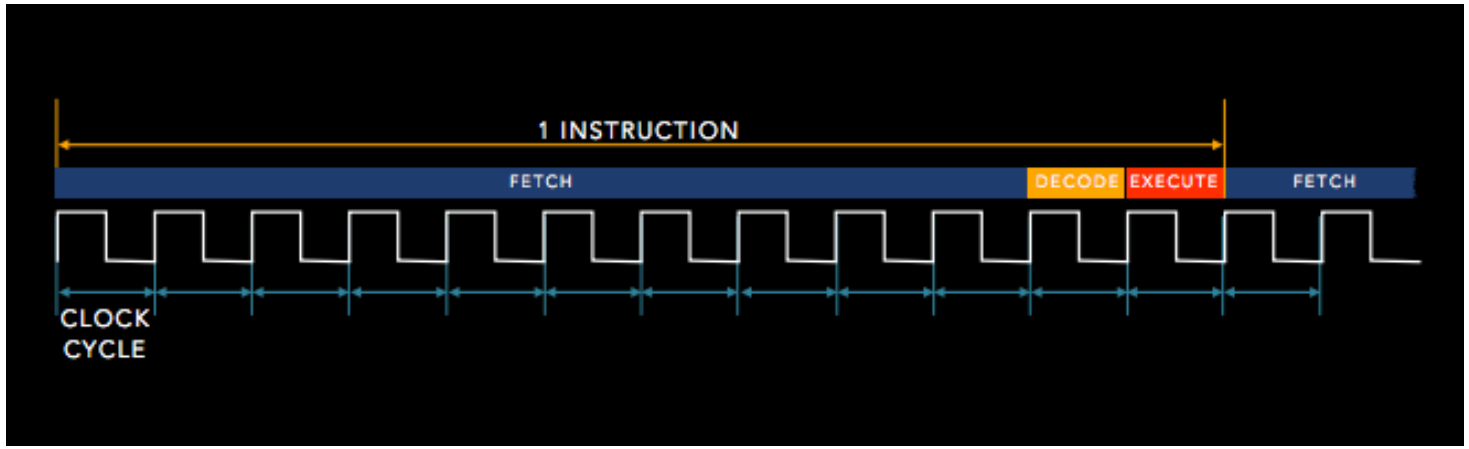
Recall: Memory Hierarchy

- As it goes further, capacity and latency increase



RAM Speed

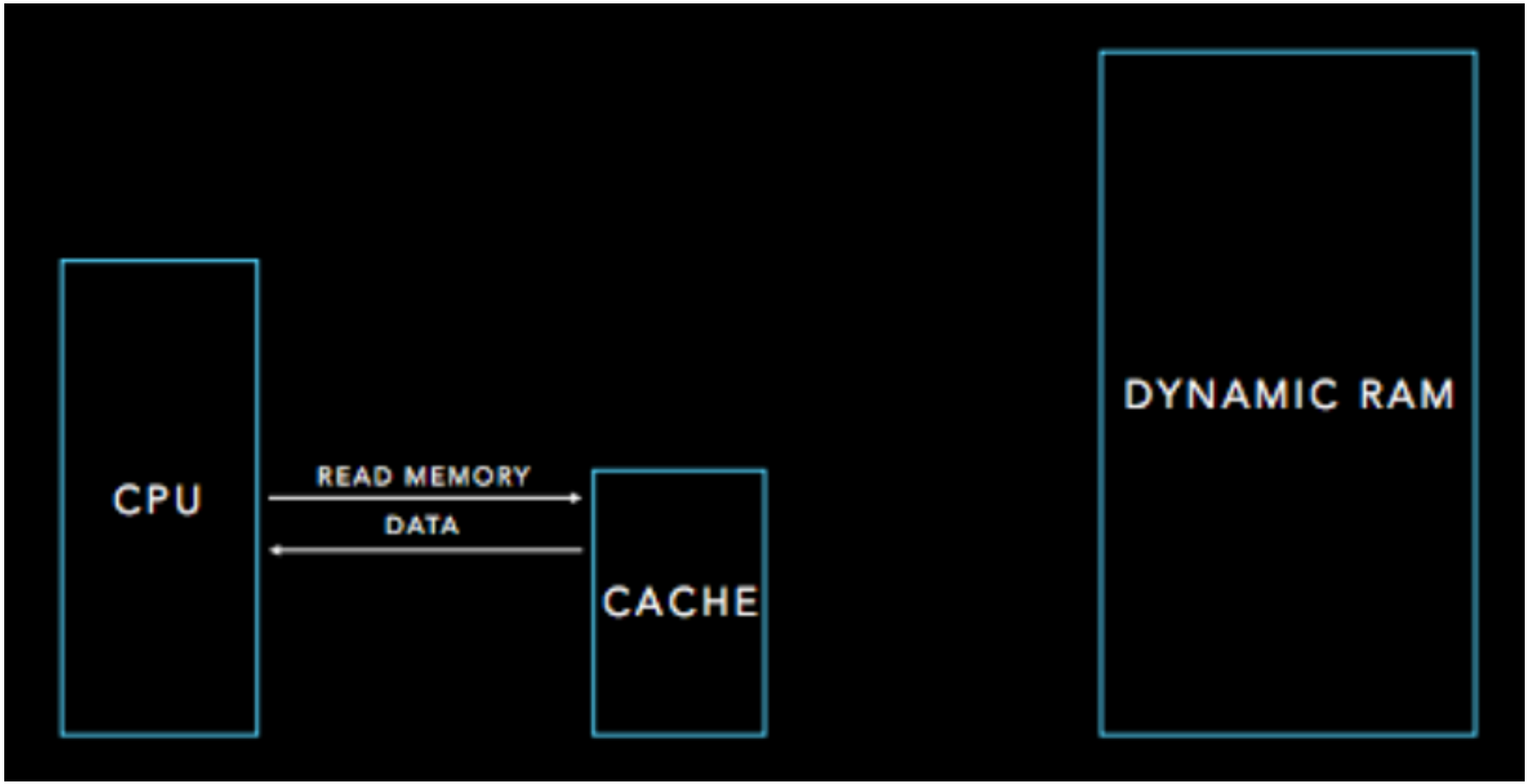
- Unfortunately, accessing RAM isn't instant
  - It's fast, but not particularly fast
- Modern RAM will take about 10-20ns to get access a piece of data
  - Known as the memory latency
- Will affect both instruction access and data access
  - Thus, affects fetch
- Realistic fetch in previous example may look like:



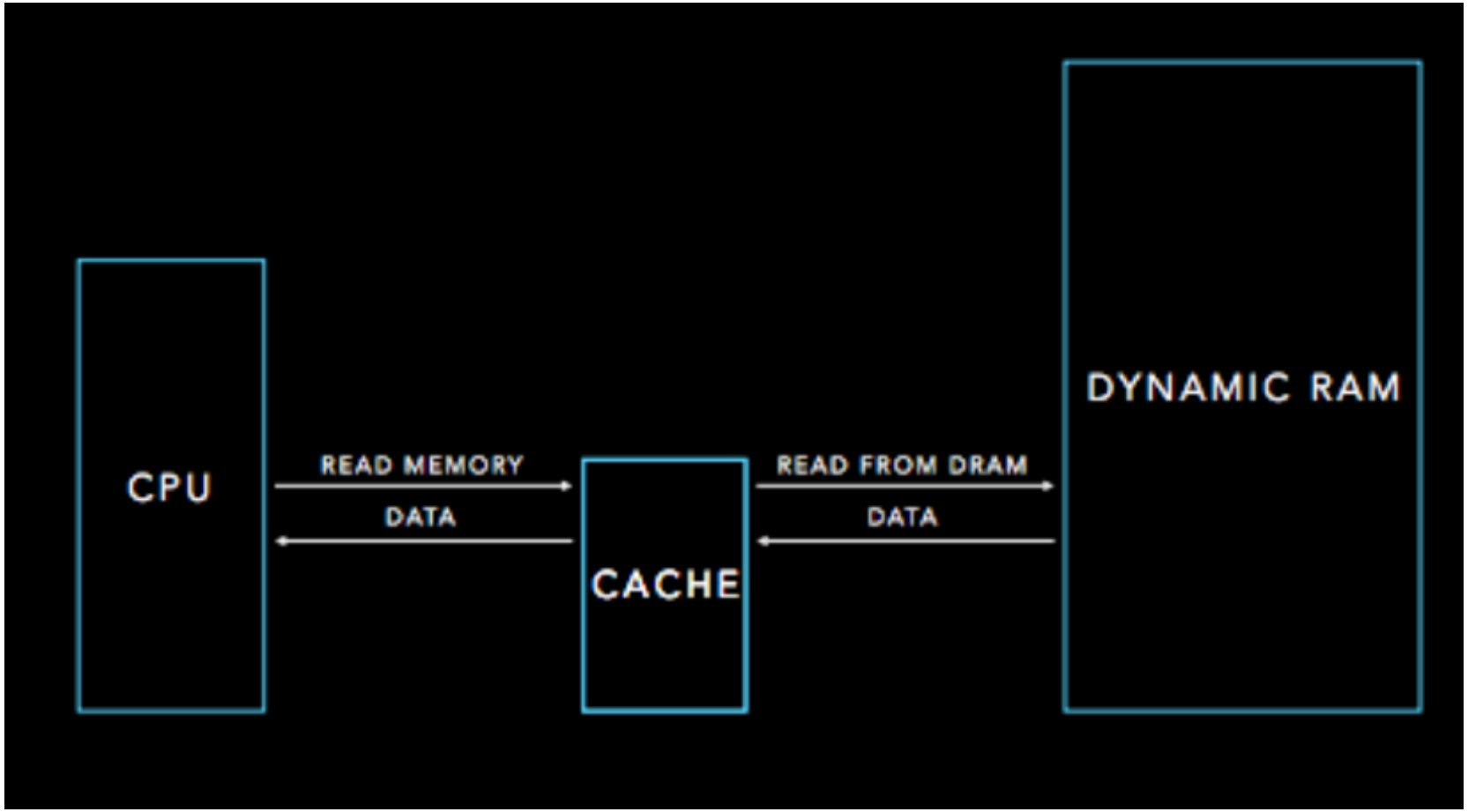
Speed Things Up: Cache

- Types of RAM:
  - SRAM (static RAM)
    - Generally faster and requires less dynamic power
    - More expensive to produce
    - Often used as cache memory for the CPU in modern computers
  - DRAM (dynamic RAM)
    - Slower
    - Less expensive to produce
- Use DRAM as our main block of memory as before
- CPUs addresses are referring to this block of memory as before
- However, the CPU does not talk directly to the DRAM
- Rather, we have a small piece of static RAM in between
  - This is called a **cache** and it stores a **temporary copy** of the instructions

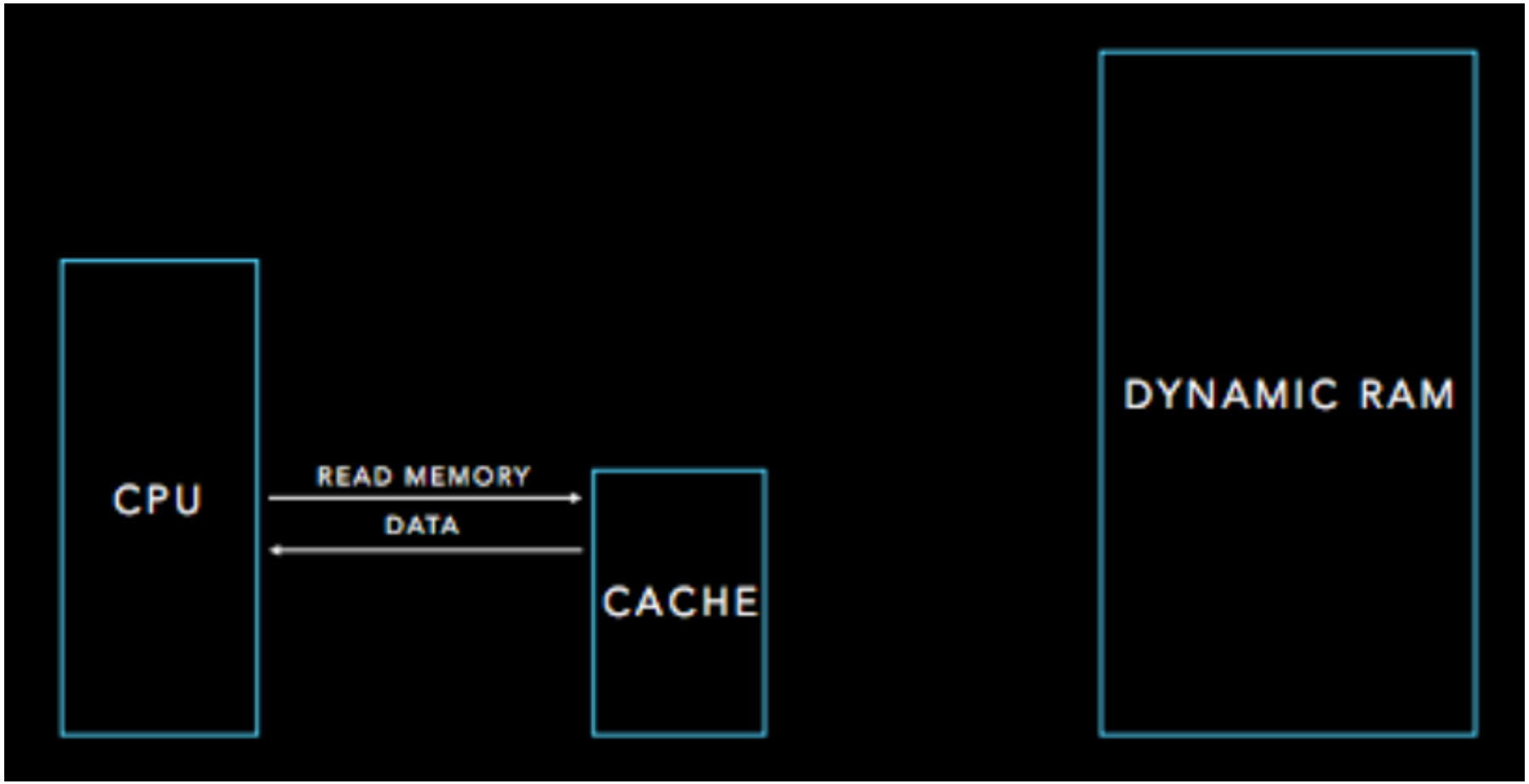
Cache Basic Principle – In Cache



Cache Basic Principle – Not In Cache



Cache Basic Principle – In Cache Again



Multiple Levels of Memory

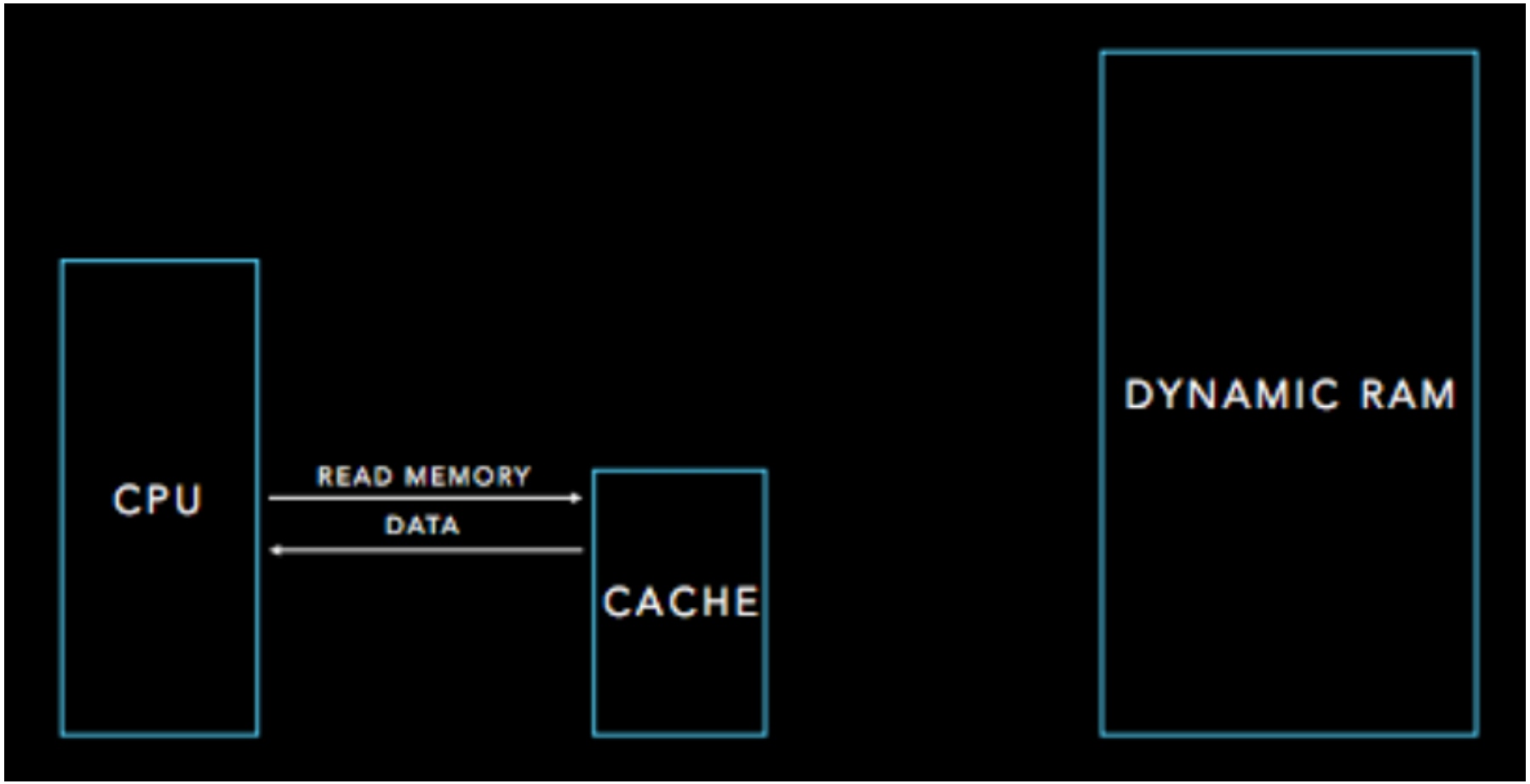
- CPU first looks in the small fast static RAM cache
- If not found look into DRAM
  - And make copy into cache
- If not in DRAM, OS can drag it in from disk
  - And make copy in DRAM
- Aim is to try and satisfy as much of the requests as possible from the cache

Caching

- Memory used for the cache is likely to be much smaller than DRAM
- Divided into indivisible units called cache lines
- Size of a cache line depends on the implementation of the cache
- When cache is updated, we always cache a complete cache line
  - Can access smaller pieces of data from the cache but we always fetch a complete cache line
- **Cache hit** – if a piece of data is found in the cache, it is known as a cache hit
- **Cache miss** – if a piece of data is not found in the cache, it is known as a cache miss

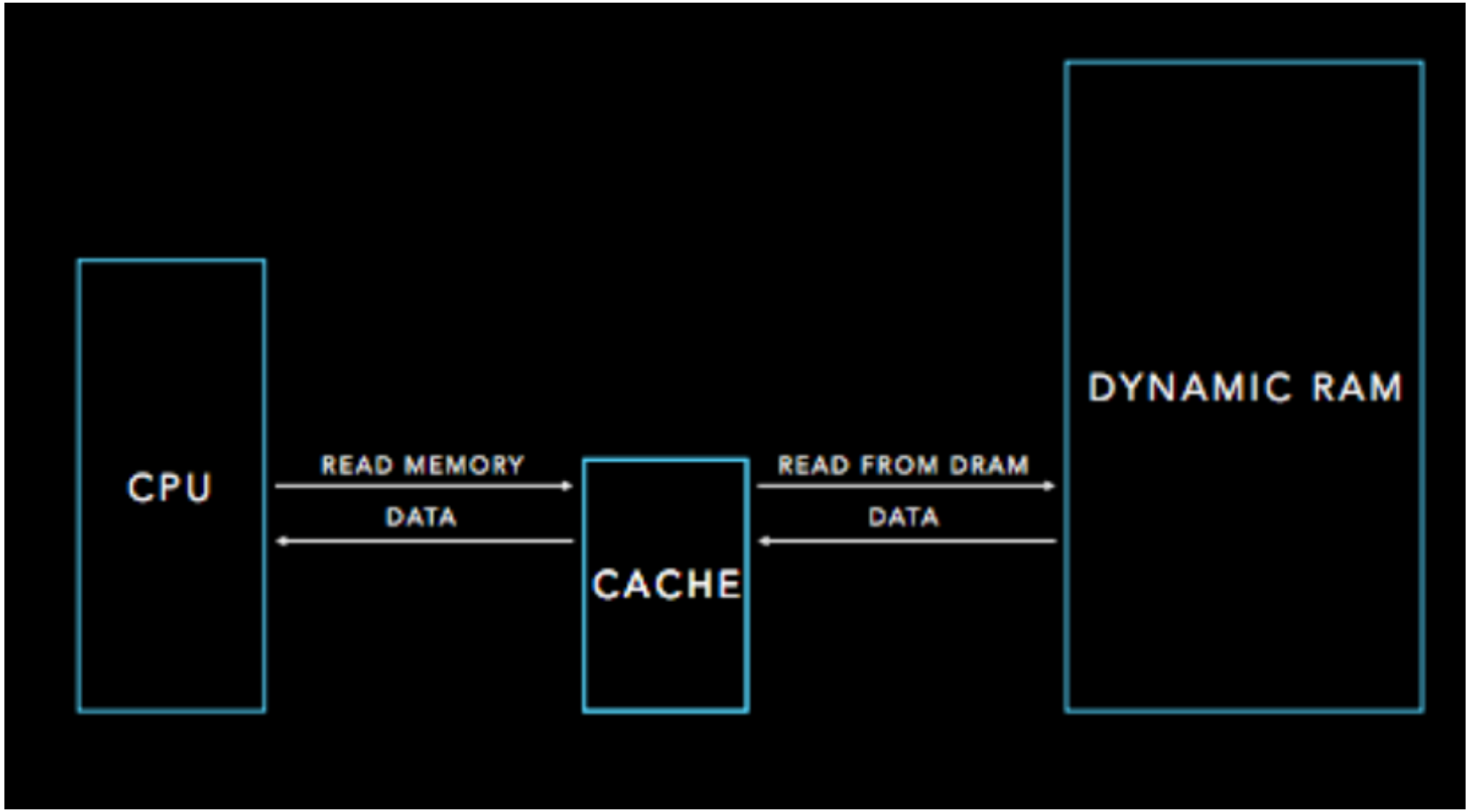
Cache Hit – Fast

- A piece of data is found in the cache – fast



Cache Miss – Slow

- A piece of data is not found in the cache – slow



Hit or Miss Measures

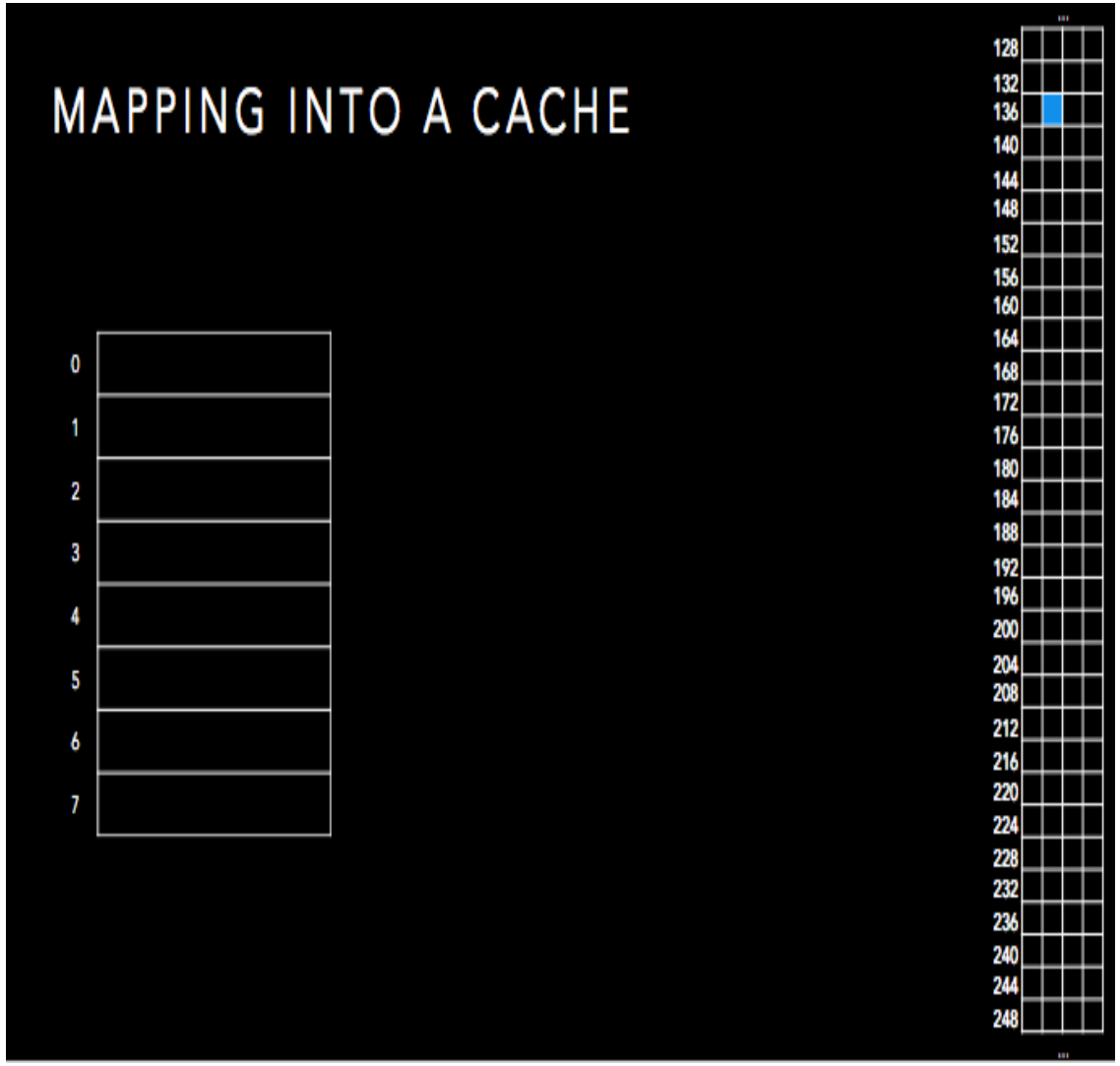
- Can measure the performance of cache
  - Hit rate – fraction of memory accesses that are cache hits
  - Miss rate – 1 - hit rate, the fraction of memory access that are not found in the cache
- Hit Time
  - Time required to access the cache, including determining whether the access is a hit or miss
- Miss penalty
  - Time required to fetch a cache line into the cache from memory, including the time to
    - Access the cache line
    - Transmit the data to the cache
    - Insert the data into the cache
    - Pass the data back to the CPU

Cache Example

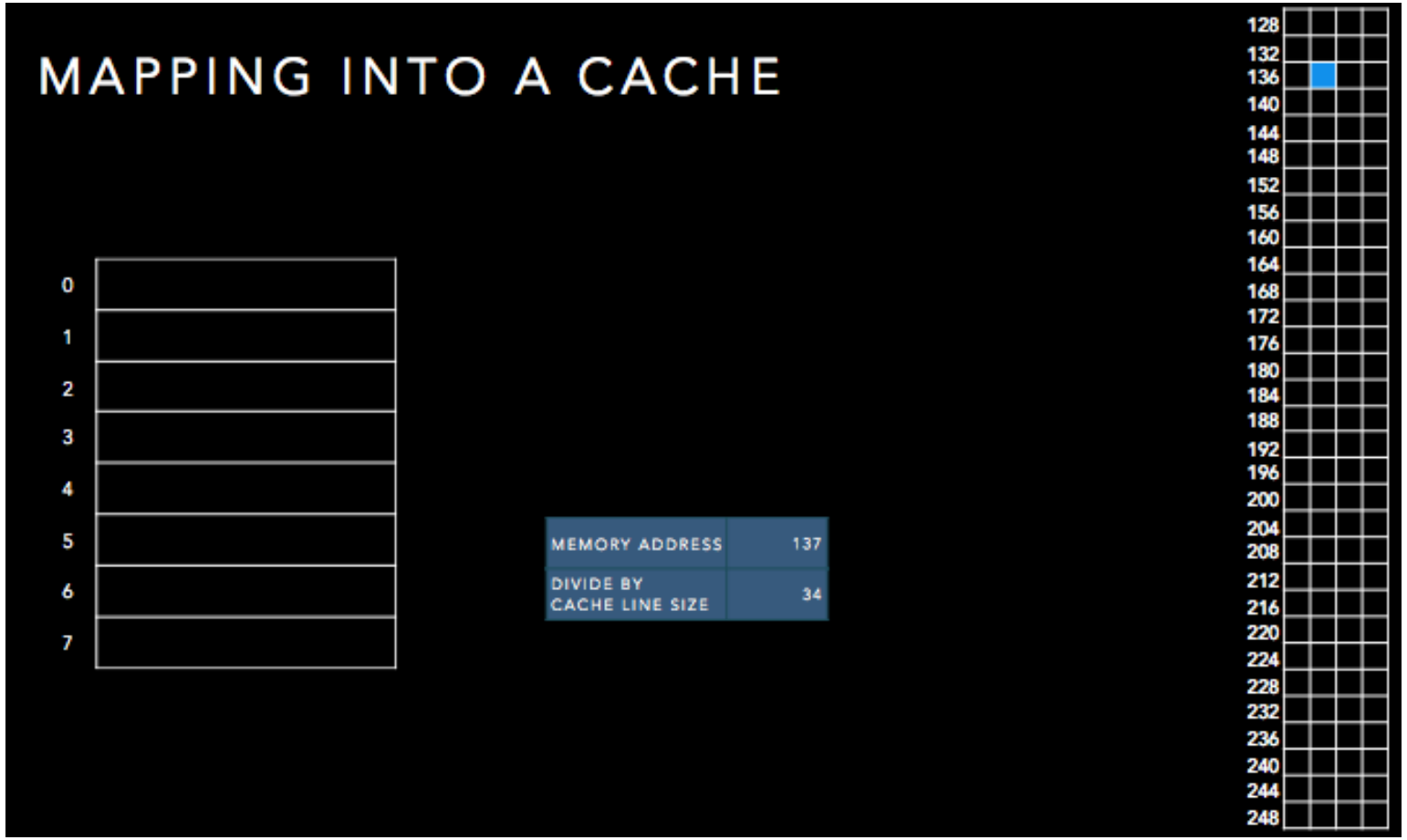
- Consider a simple cache, with the cache line size of 32 bits
  - Assume there are only eight entries in the cache
  - Assume that each word in main memory can only map to one possible position in the cache (also called a direct-mapped cache)
- Location in the cache is almost always based on the address in main memory
- One possible algorithm used to map address to a specific cache line
  - First, divide address by cache line size
    - Address in memory refers to individual bytes, but we only cache whole cache lines
  - Second, find the cache line
    - Take the remainder from dividing the above result by the size of the cache (in cache lines/blocks)

Cache Example

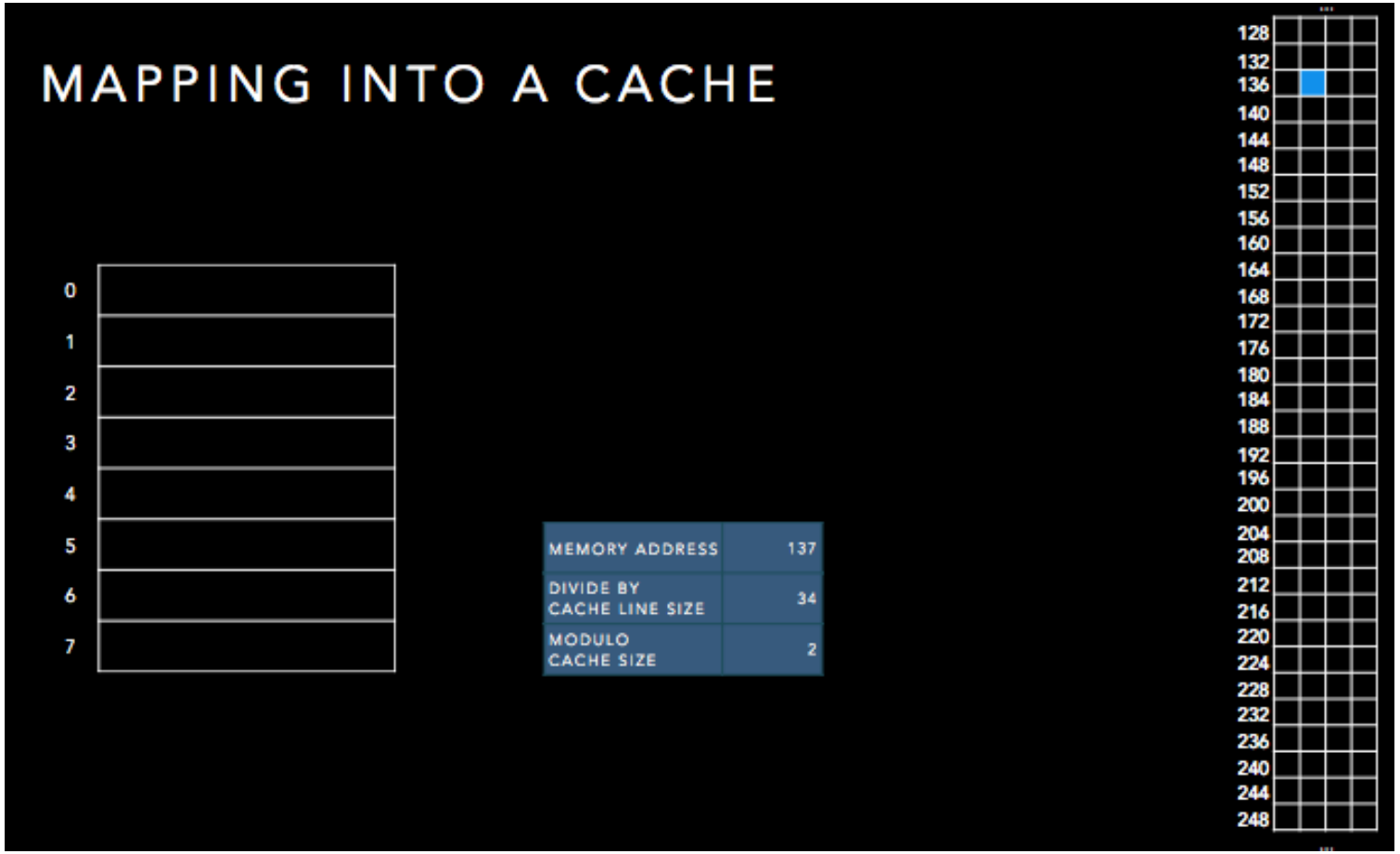
- Example memory location 137
- Divide by the cache line size
  - Since the address in memory refers to individual bytes, the cache line size should be also in bytes
  - 137/4 = 34
- Take the remainder from dividing the above result by the size of the cache (in cache lines/blocks)
  - 34 mod 8 = 2
  - Memory location 137 maps in to cache line 2



Cache Example

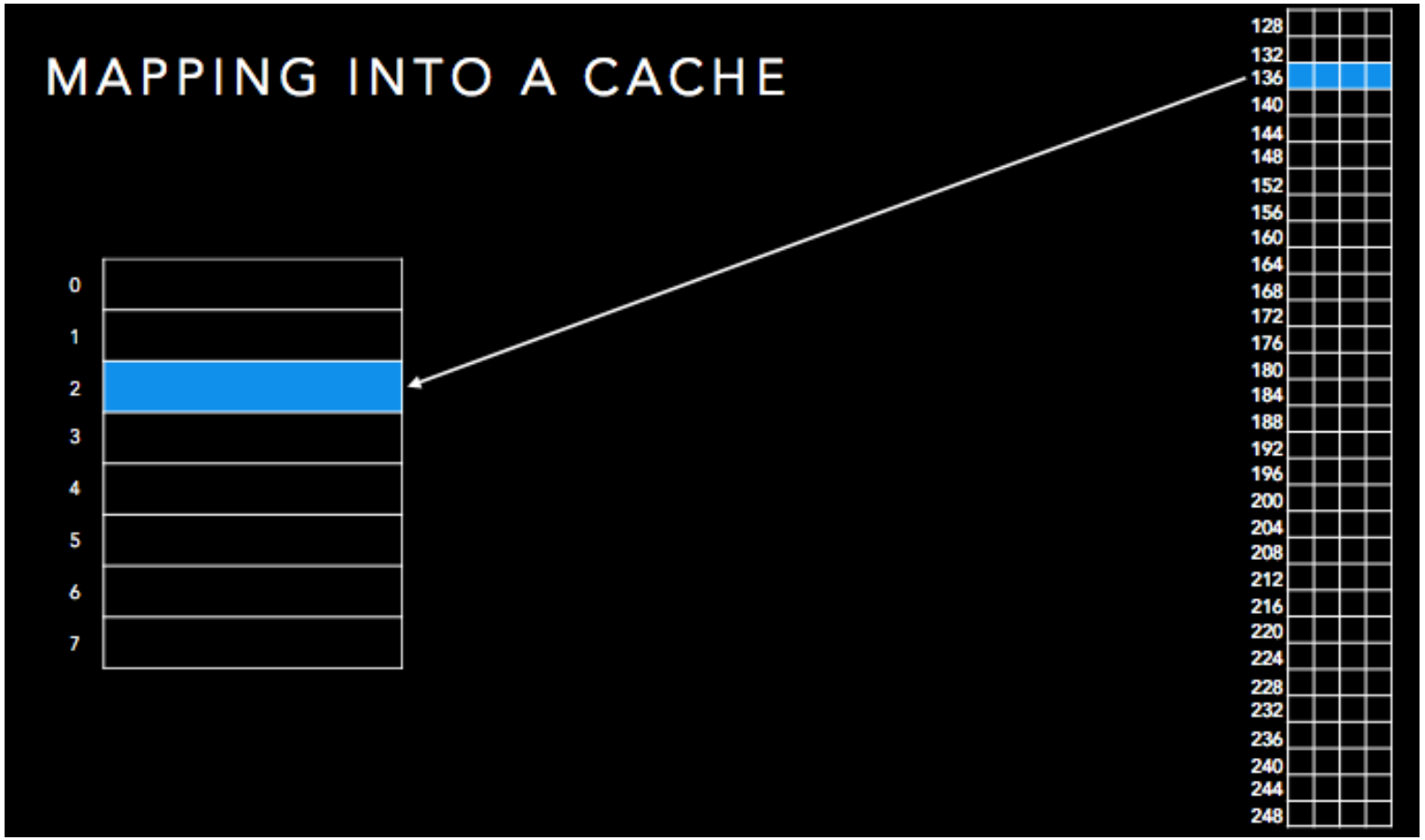


Cache Example

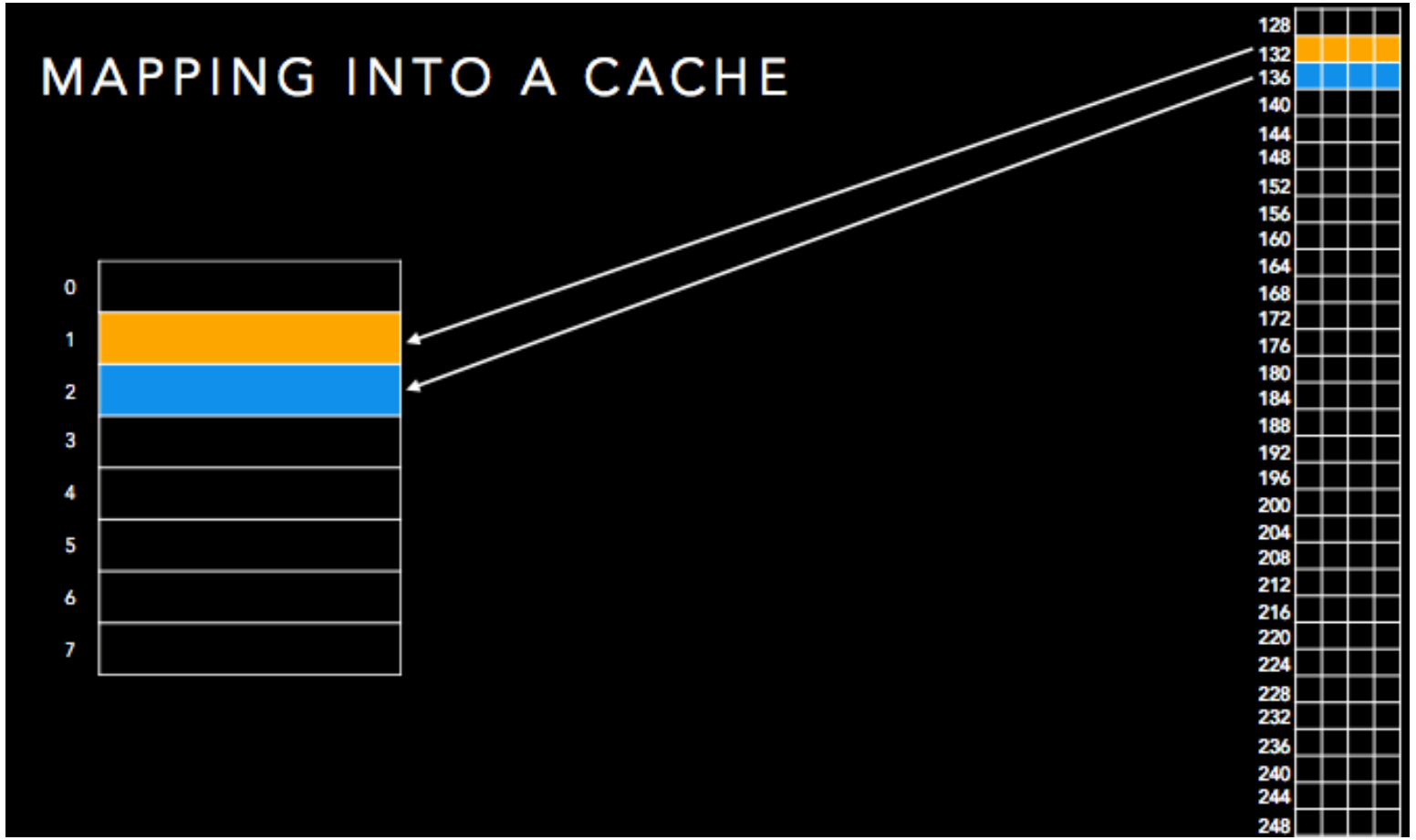


Cache Example

- Memory location 137 maps in to cache line 2

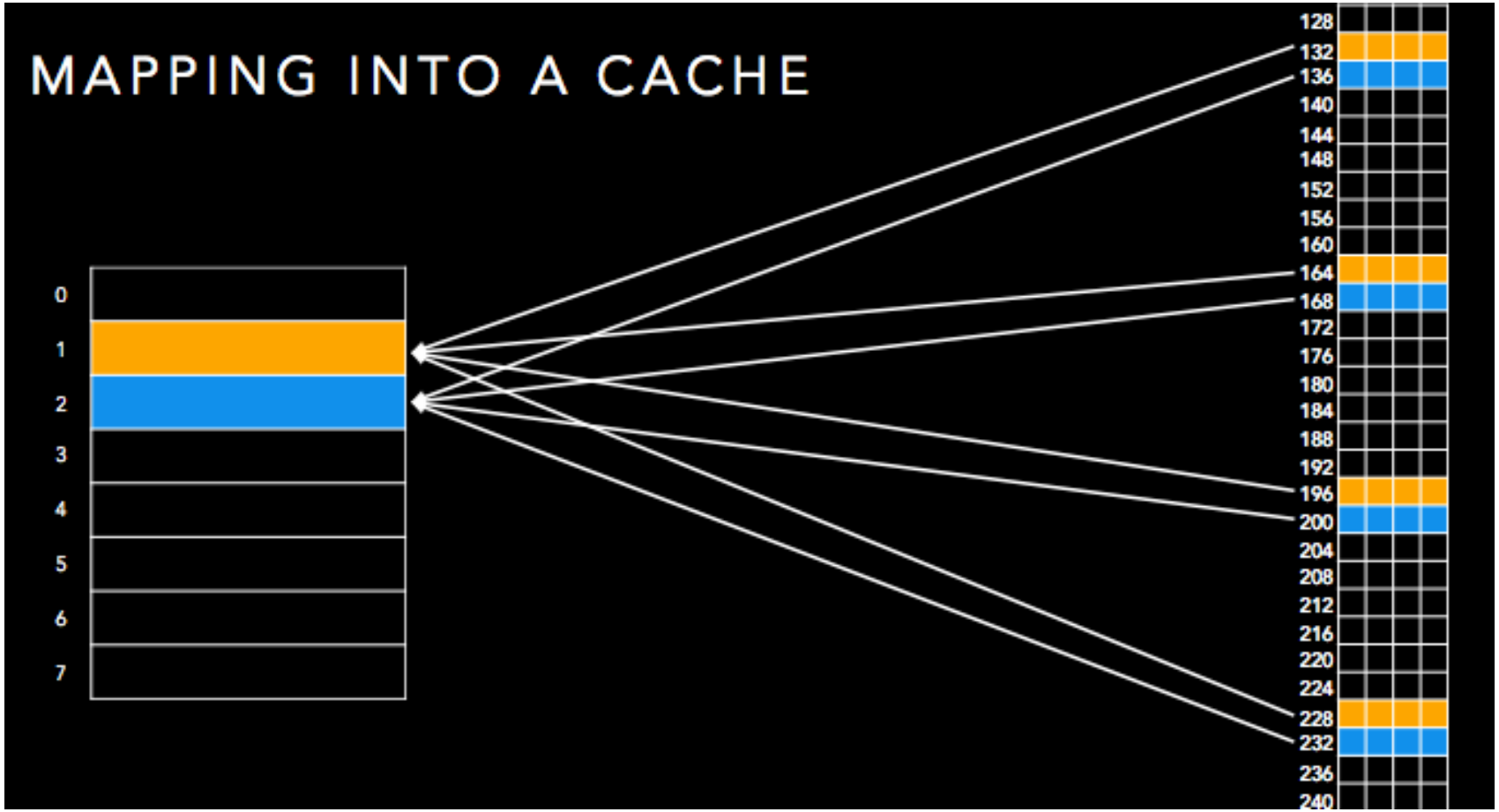


Cache Example: Another Address



Problems?

- Many memory addresses will map to the same cache line



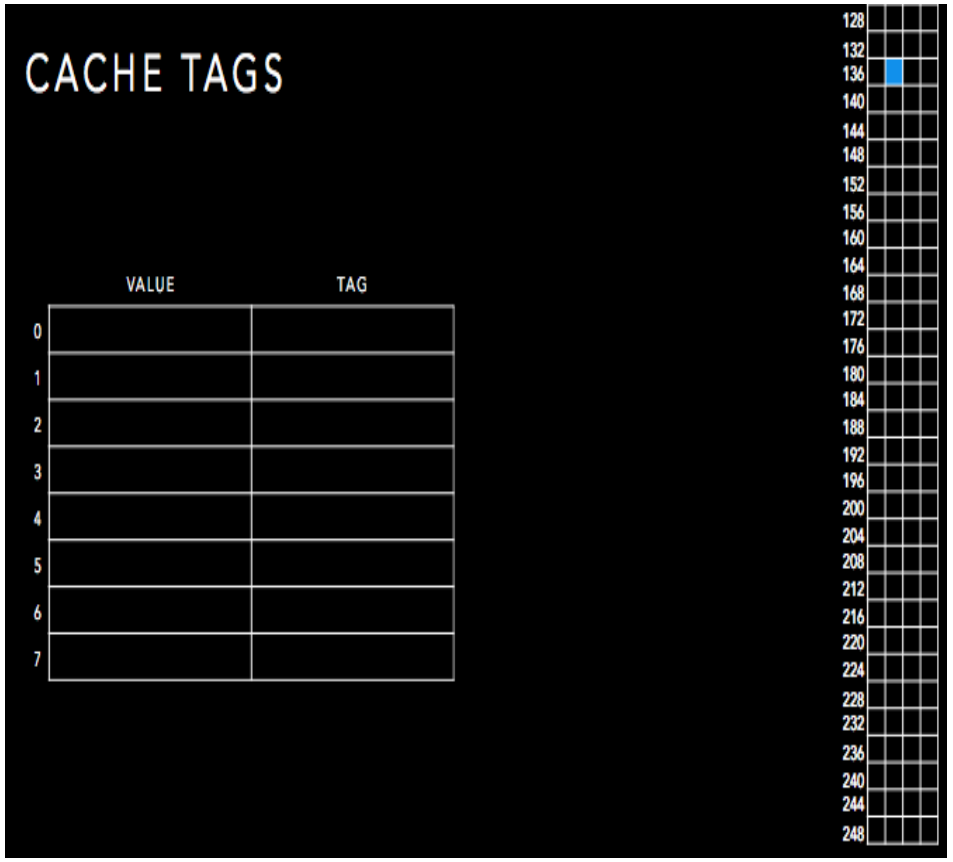
Cache Tags

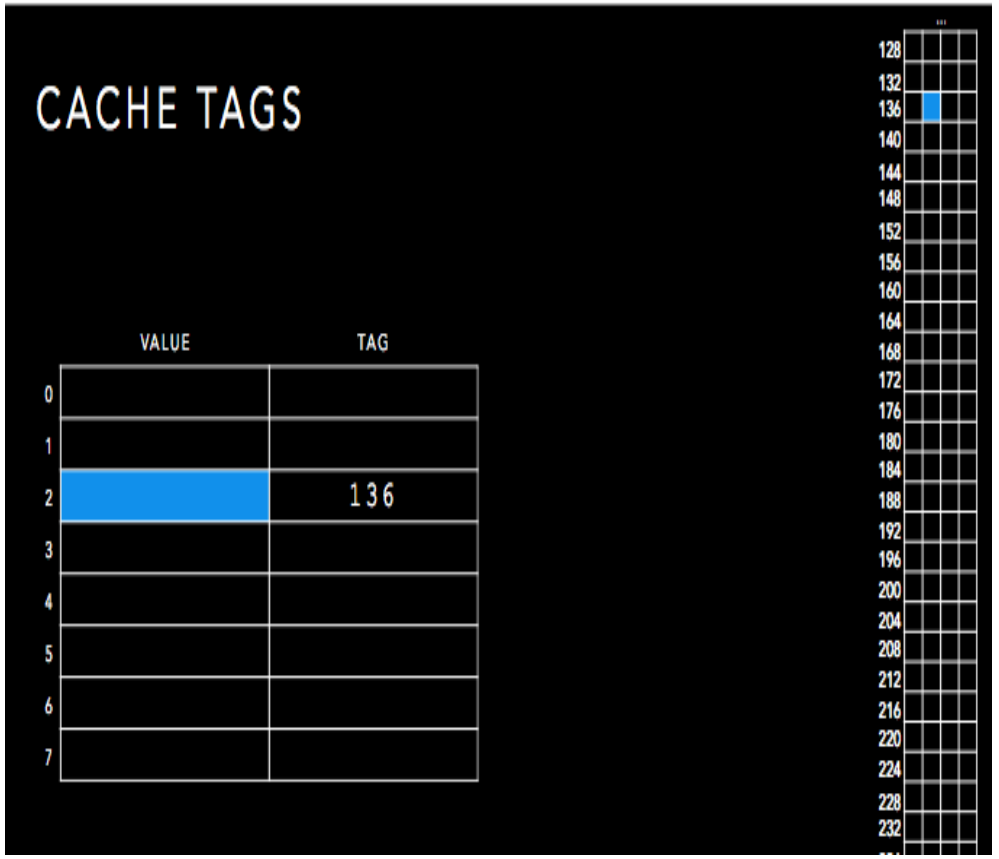
- Many memory addresses will map to the same entry in the cache
- If a cache has an entry, we need to make sure we are getting the correct value for the memory address we want
- We tag each cache entry with the address of where it came from
- Don’t necessarily need to store the complete address
  - Cache entry itself will encode some of it

Cache Tags

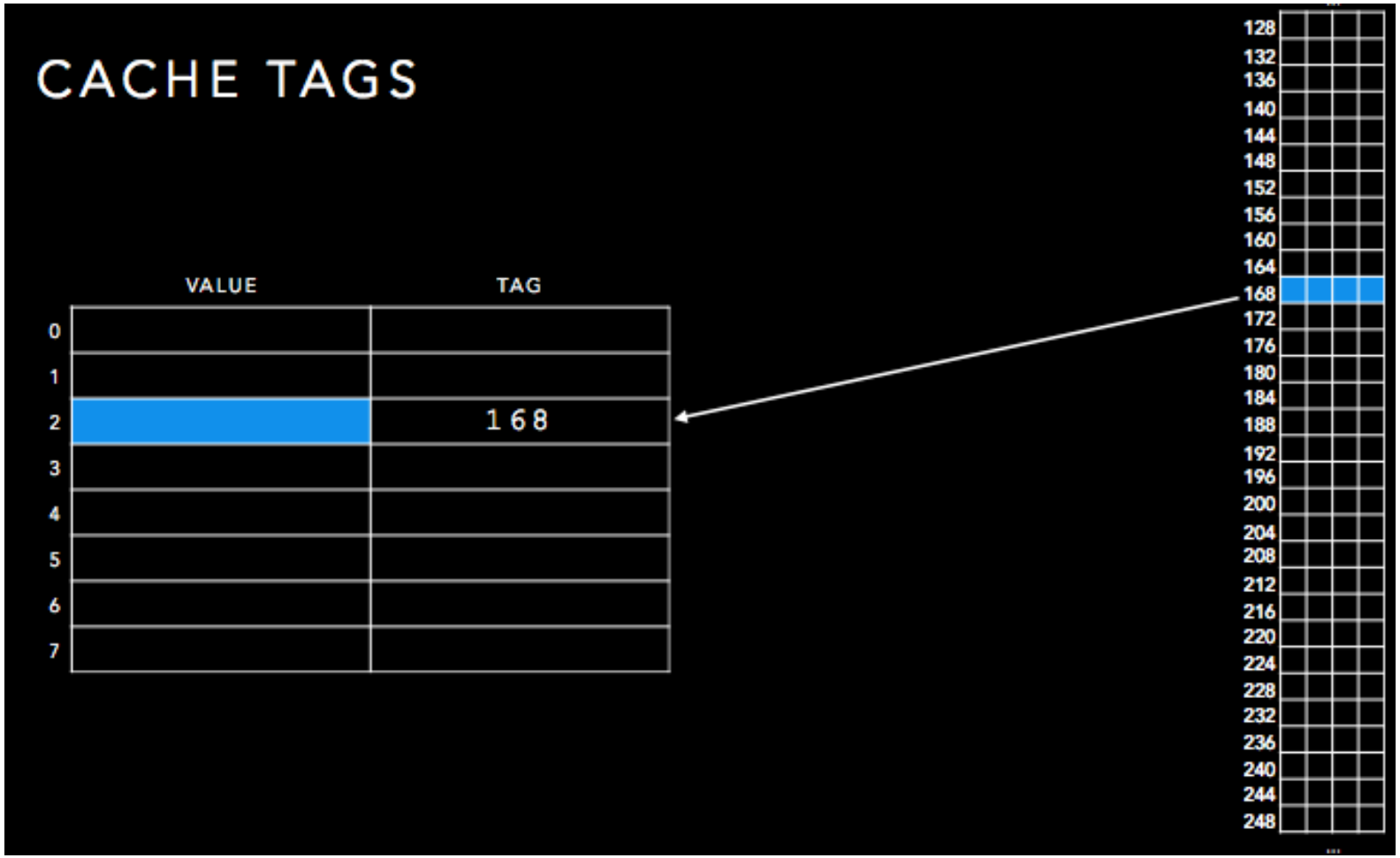
- When we access an address, look at the specific cache entry
- If the tag matches the tag for the address in the cache
  - The memory is cached, so we can return it
- If the tag doesn’t match
  - Fetch the value from main memory
  - And update the cache

Cache Tags: Example





Cache Tags: Example



One More Detail: The Valid Bit

- When started, the cache is empty and does not contain valid data
- We should account for this by adding a valid bit for each cache block
  - When the system is initialized, all the valid bits are set to 0
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1
- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity

The Hack Computer

- A 16-bit Von Neumann platform
- The instruction memory and the data memory are physically separated
- Screen: 512 rows by 256 columns, black and white
- Keyboard: standard
- Designed to execute programs written in the Hack machine language

The Hack Computer: Main Parts

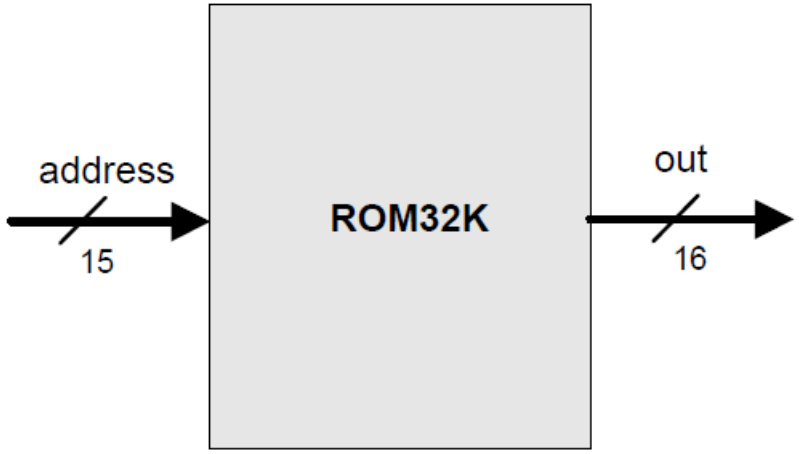
- Instruction memory (ROM)
- Memory (RAM)
  - Data memory
  - Screen (memory map)
  - Keyboard (memory map)
- CPU
- Computer (the logic that holds everything together)

Instruction Memory

- The ROM is pre-loaded with a program written in the Hack machine language
- The ROM chip always emits a 16-bit number:

```
out = ROM32K[address]
```

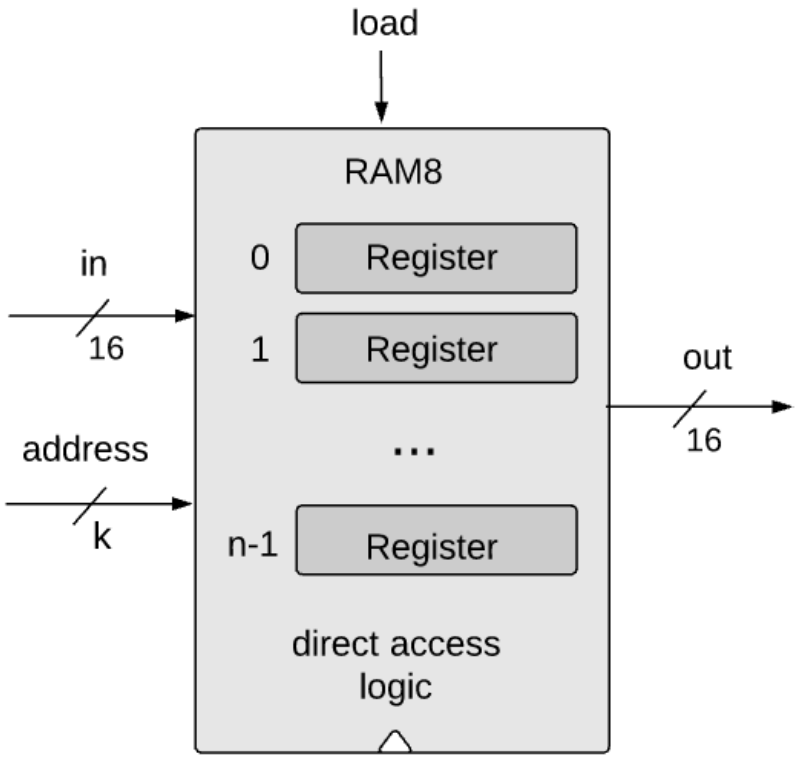
- This number is interpreted as the current instruction



Recall: Memory (RAM)

- Random-Access Memory (RAM)
  - Traditionally packaged as a chip
  - Basic storage unit is normally a cell (one bit per cell)
  - Multiple RAM chips form a memory
- RAM should be able to **access randomly chosen** words, with no restriction in the order in which they are accessed
  - Assign each word in the n-register RAM a unique address (an integer between 0 to n-1)
  - Given an address j, the individual register with the address j can be selected

RAM



Architecture:

A sequence of *n* addressable registers, with addresses 0 to *n*-1

Address width:

$k = log_2 n$ (# of bits that is needed to store different add. for each reg)

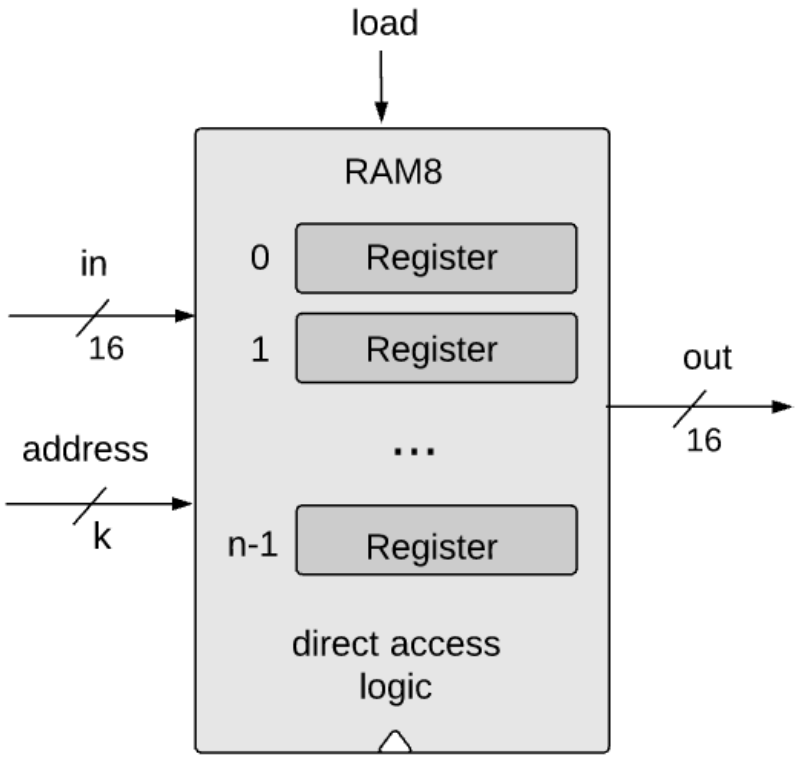
For 8 registers, k =3

For 16 registers, k =4

Word width:

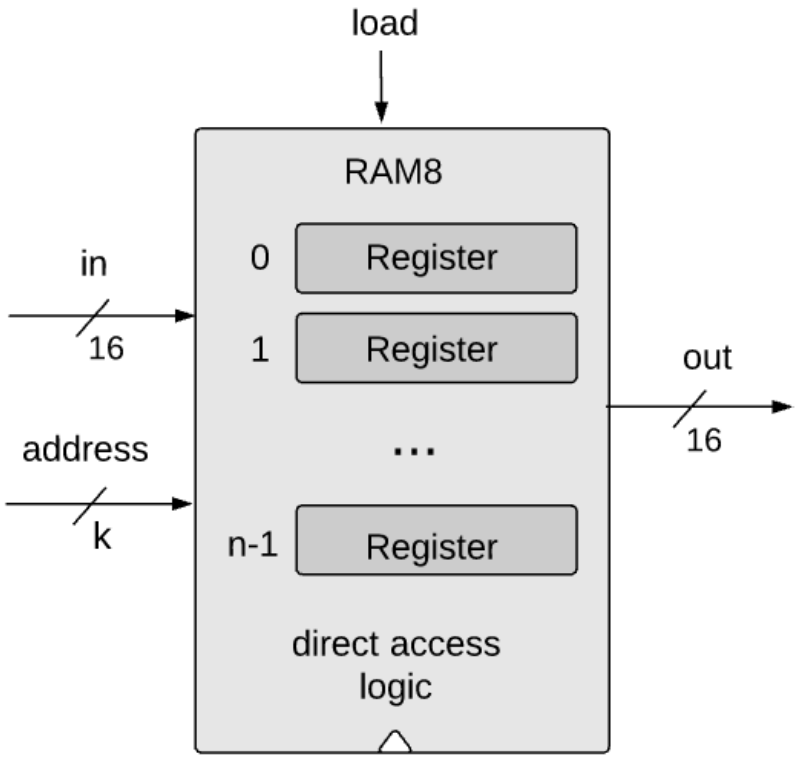
No impact on the RAM logic

A family RAM chips



chip name	*n*	*k*
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

RAM: abstraction



At any given point of time:

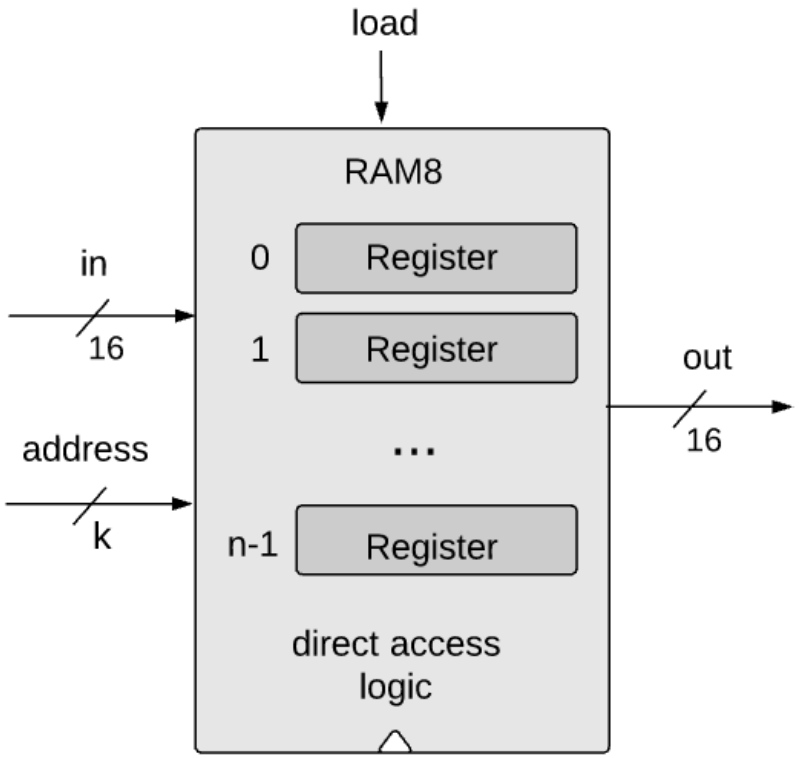
- *one* register in the RAM is selected
- all the other registers are irrelevant

To read `Register` *i* :

set `address` = *I*

Result:
`out` emits the value of `Register` *i*

RAM: abstraction



At any given point of time:

- *one* register in the RAM is selected
- all the other registers are irrelevant

To set `Register` *i* to *v* :

set `address` = *I*

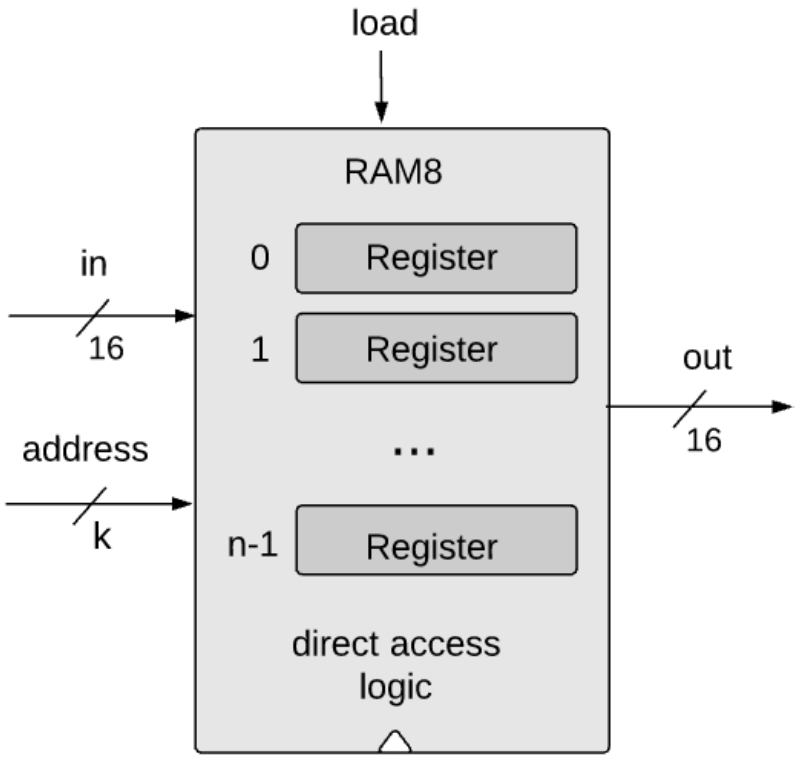
set `in` = *v*

set `load` = 1

Result:

- The state of `Register` *i* becomes *v*
- From the next cycle onward, `out` emits *v*

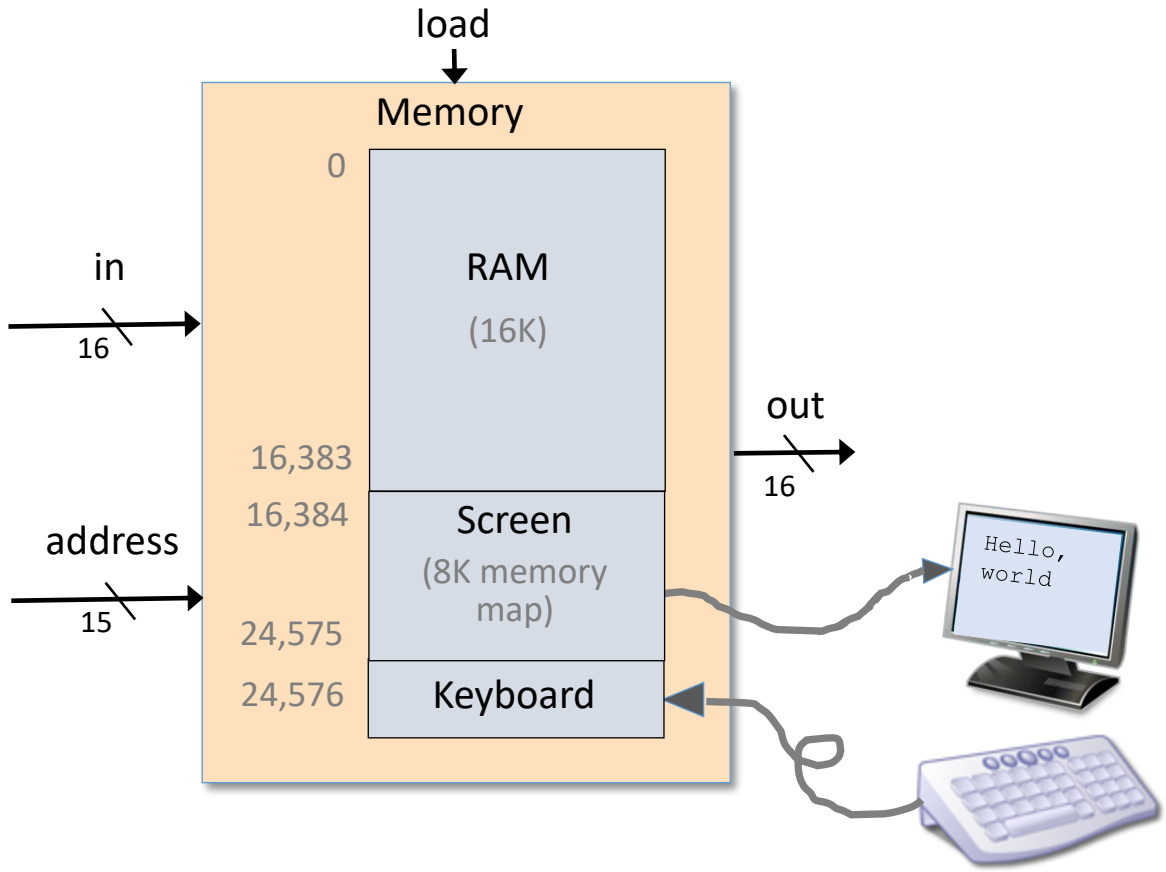
RAM: abstraction



Why "Random Access Memory"?

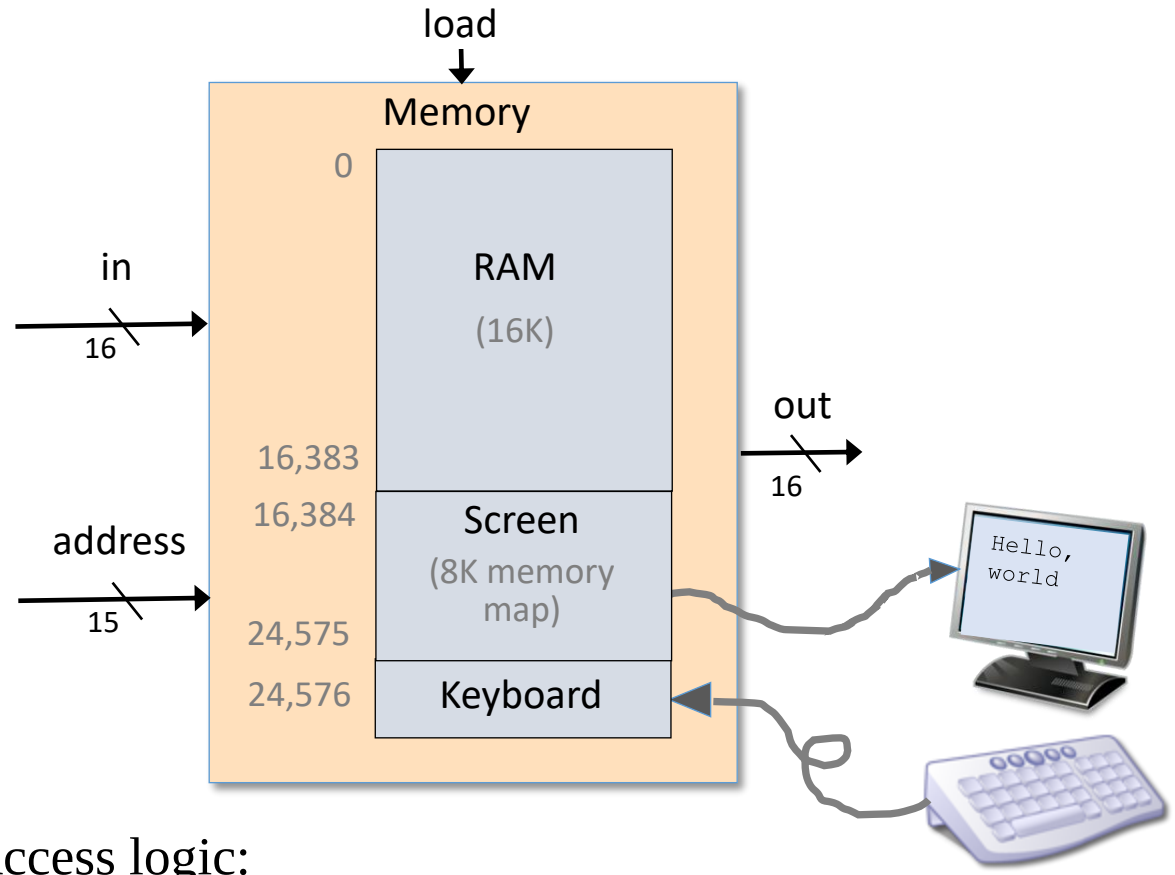
Irrespective of the RAM size (n),
every randomly selected register can be accessed "instantaneously",
at more or less the same time.

Look ahead - Memory: implementation



- Address 0 to 16383: data memory, used to record or recall values (variables, objects, arrays, etc.)
- Address 16384 to 24575: screen memory map, used to write to the screen (or read the screen)
- Address 24576: keyboard memory map, used to read which key is currently pressed

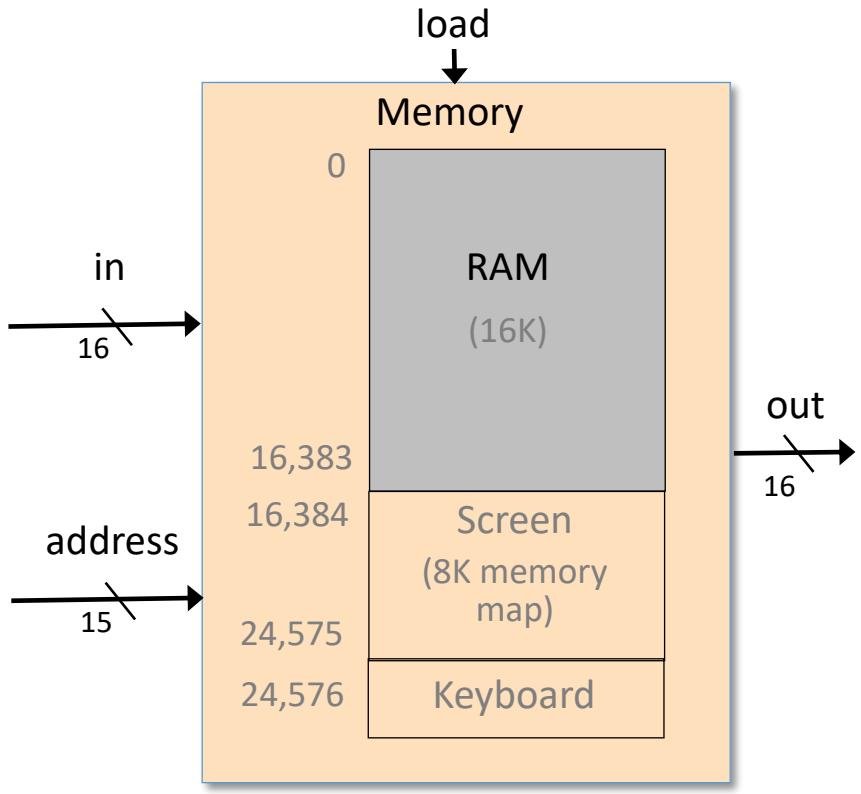
Look ahead - Memory: implementation

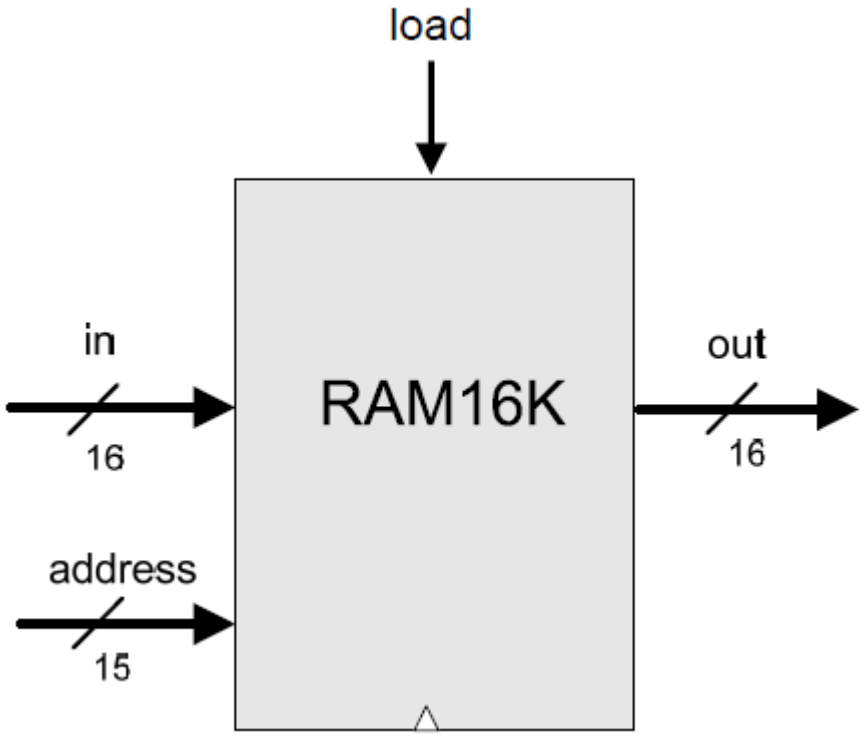


Access logic:

- Access to any address from 0 to 16,383 results in accessing the RAM16K chip-part
- Access to any address from 16,384 to 24,575 results in accessing the Screen chip-part
- Access to address 24,576 results in accessing the keyboard chip-part
- Access to any other address is invalid.

Data Memory





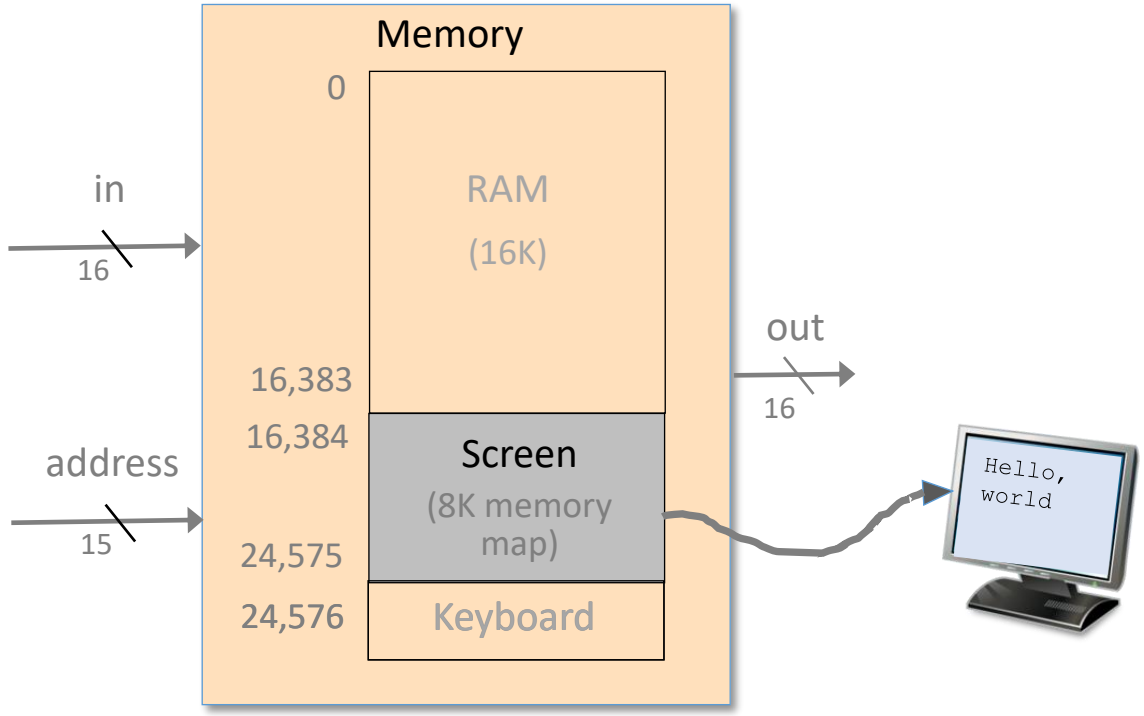
To read RAM[k]:

- set address to k
- read data

To write RAM[k]=x:

- set address to k,
- set in to x,
- set load to 1,
- run the clock

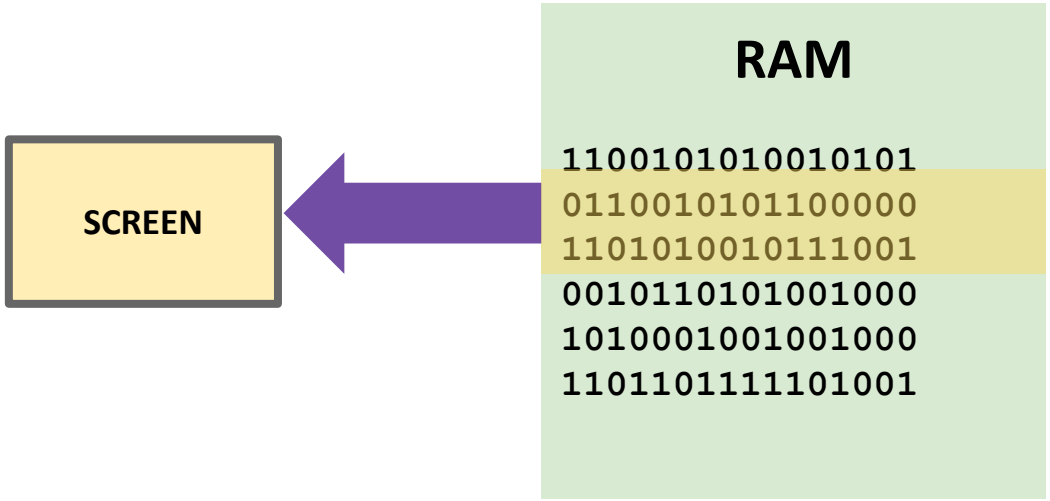
Screen

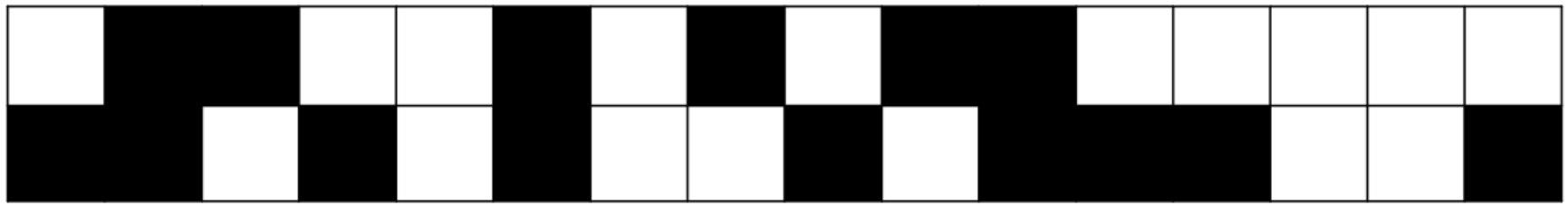


- In the Hack platform, the screen is implemented as an 8K 16-bit RAM chip with a side effect of refreshing.
- The Screen chip has a basic RAM chip functionality:
  - read logic: `out = Screen[address]`
  - write logic: `if load then Screen[address] = in`
- Side effect: Continuously refreshes a 256 by 512 black-and-white screen device

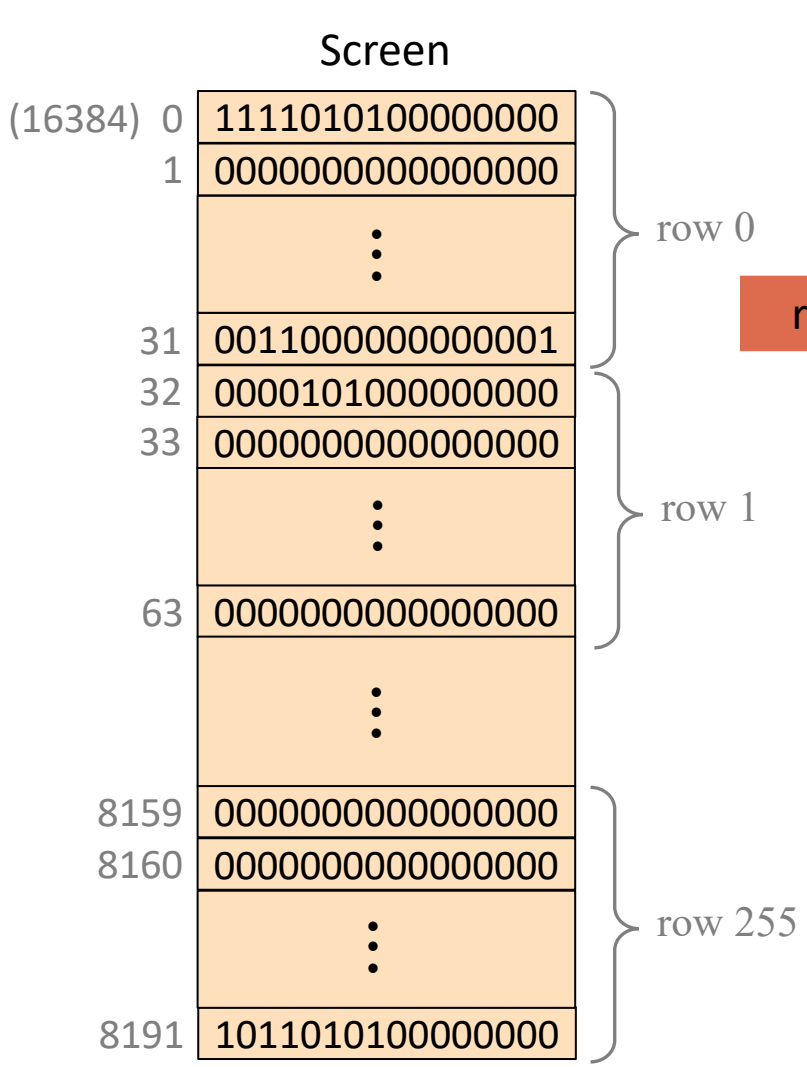
Screen memory map

- The bit contents of the Screen chip is called the “screen memory map”
- Each bit of the screen memory map corresponds to one pixel (1 = black, 0 = white)

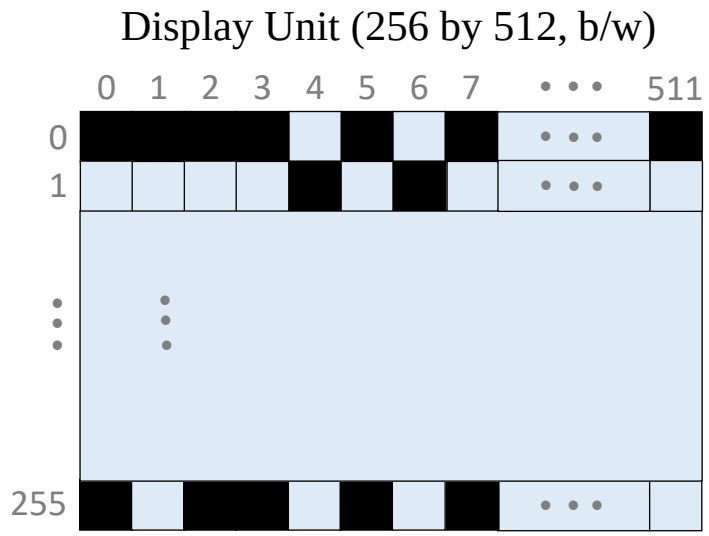




Screen memory map



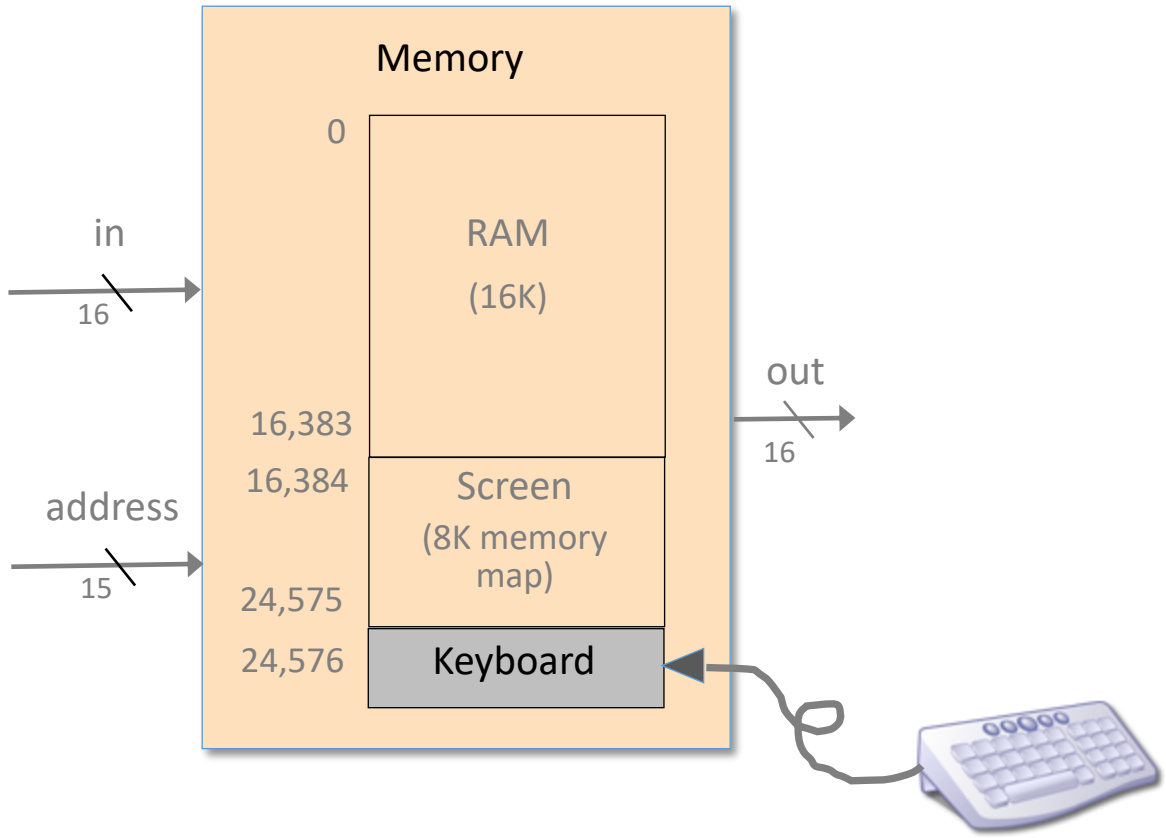
refresh



- To set pixel `(row,col)` black

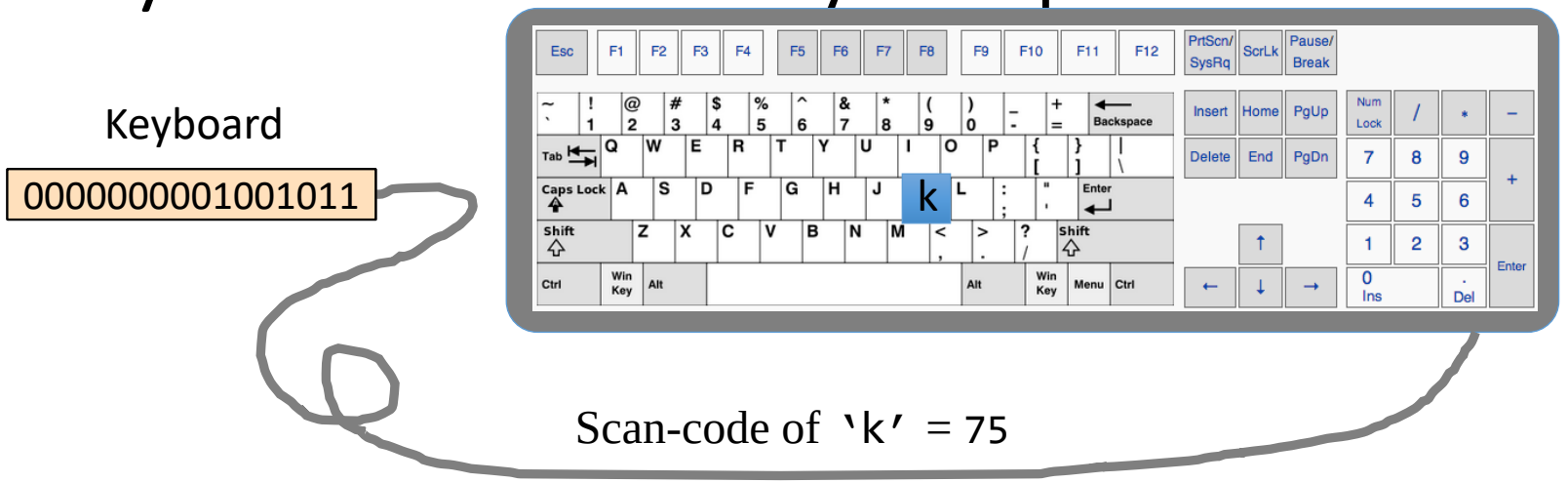
1. `word=Screen[32*row + col/16]` (`RAM[16384 + 32*row + col/16]`)
2. Set the `(col%16)-th` bit of `word` to 1
3. Commit `word` to the RAM

Keyboard



- A 16-bit register is used to keep the key stroke
- When a key is pressed on the keyboard, the key's scan code appears in the keyboard memory map.

Keyboard memory map



The `Keyboard` chip emits the scan-code (16-bit value) of the currently pressed key, or `0` if no key is pressed.

Scan Codes

key	code
(space)	32
!	33
“	34
#	35
$	36
%	37
&	38
‘	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

key	code
0	48
1	49
…	…
9	57

:	58
;	59
<	60
=	61
>	62
?	63
@	64

key	code
A	65
B	66
C	…
…	…
Z	90

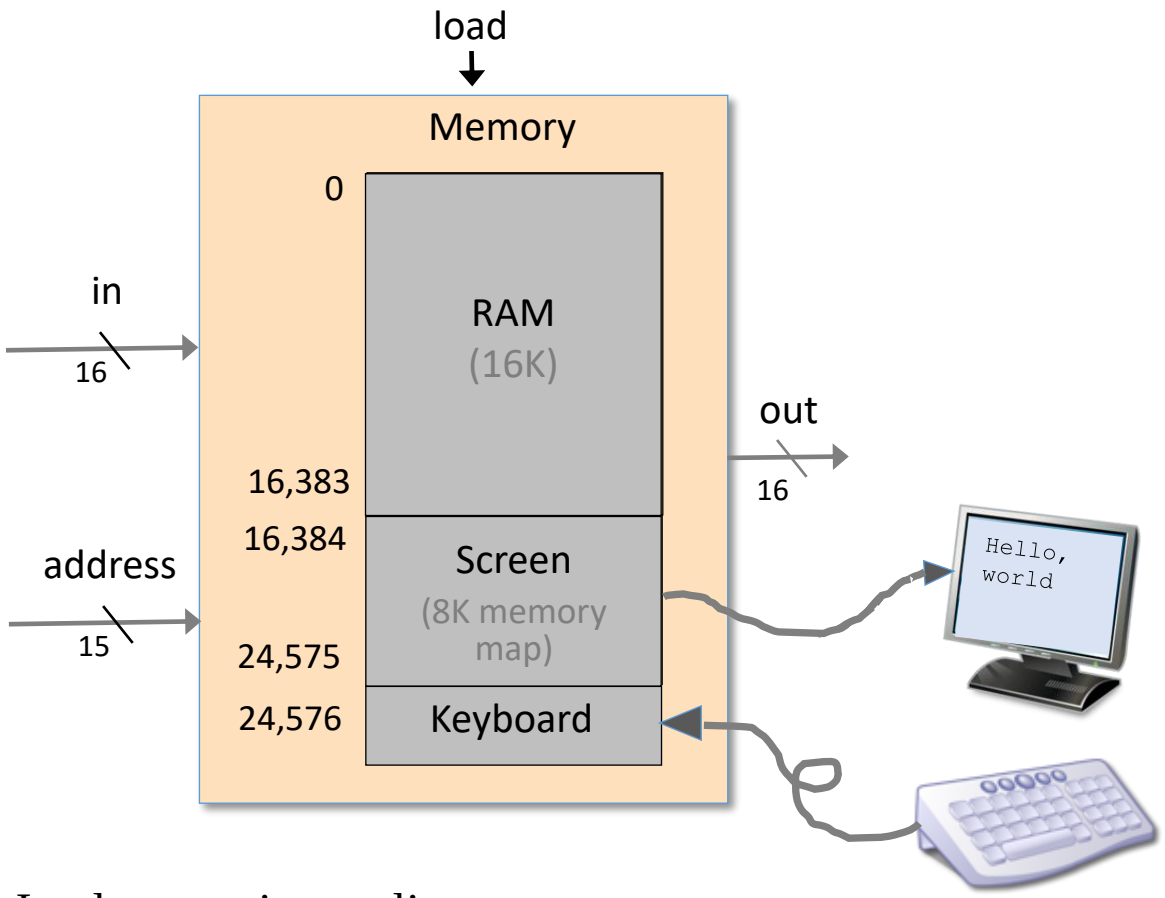
[	91
/	92
]	93
^	94
_	95
`	96

key	code
a	97
b	98
c	99
…	…
z	122

{	123
|	124
}	125
~	126

key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
…	…
f12	152

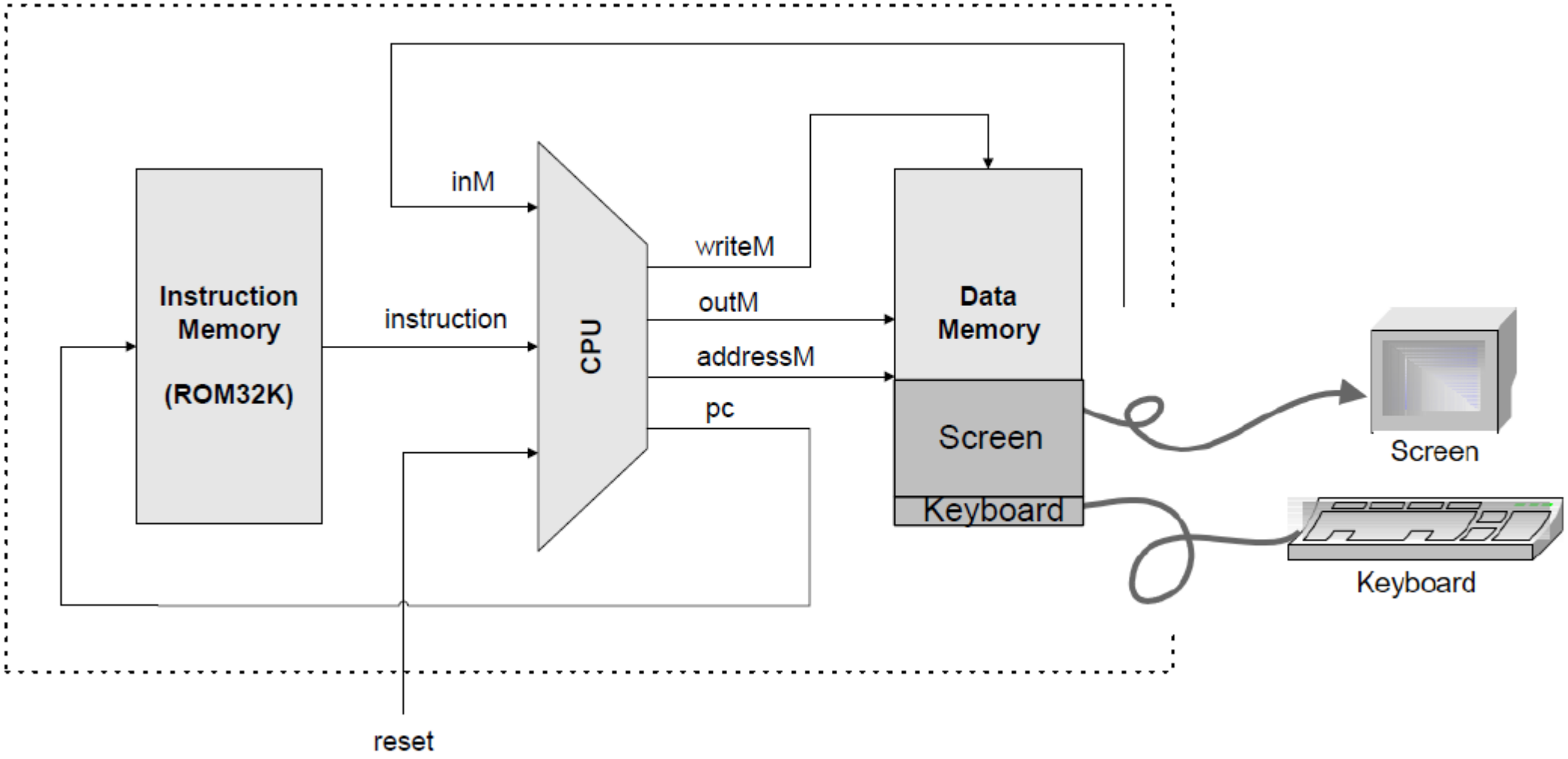
Memory implementation



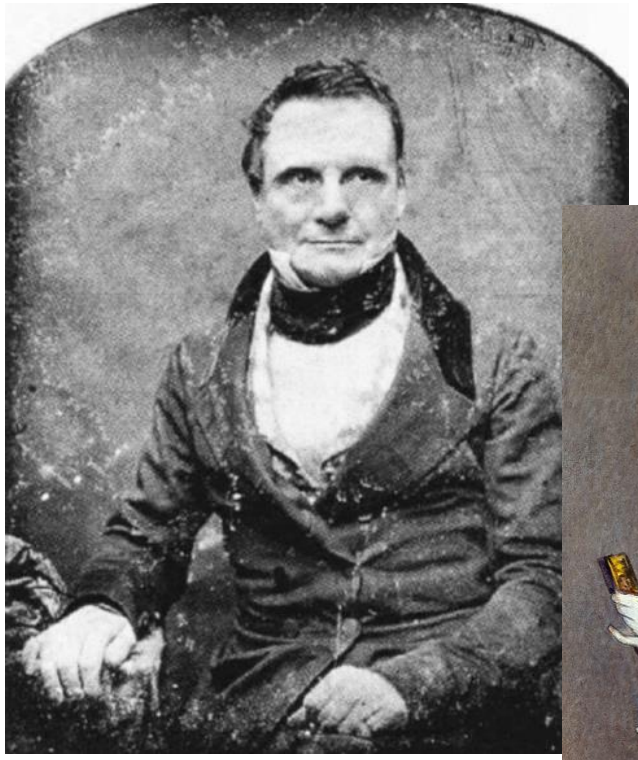
Implementation outline:

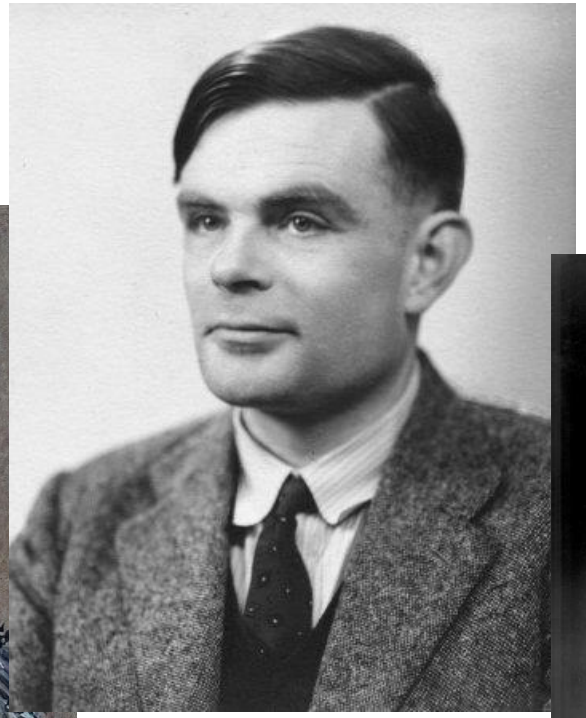
- Uses the three chip-parts `RAM16K`, `Screen`, and `Keyboard`
- Routes the `address` input to the correct `address` input of the relevant chip-part.

The Hack Computer (Put Together)



Computer Science Fundamentals…..How did we get to the current architecture?



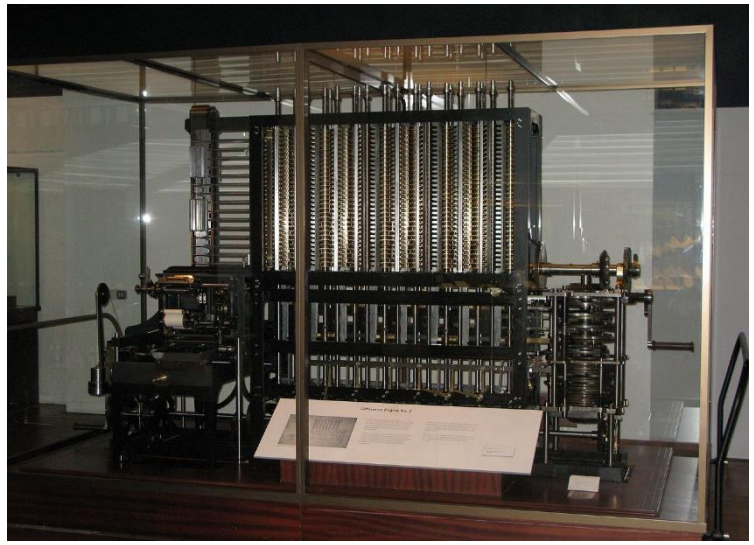


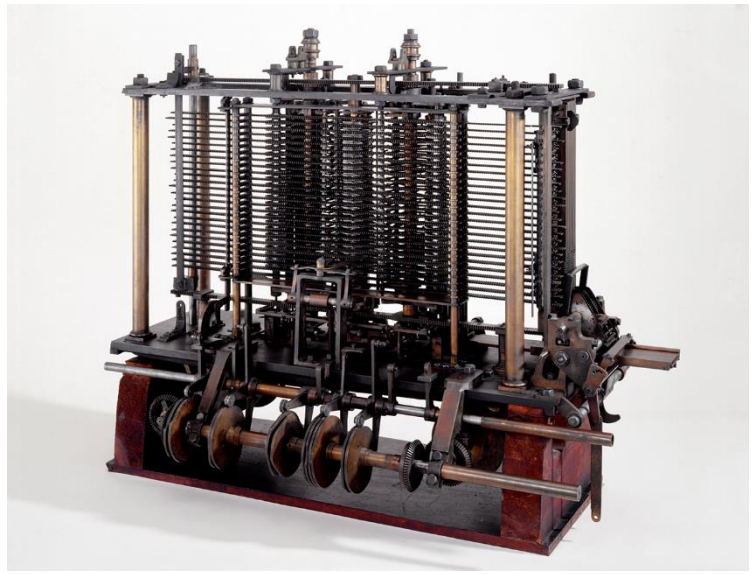
Charles Babbage (1791 – 1871)

- English polymath (mathematician, philosopher, inventor and mechanical engineer)
- Originated the concept of a digital programmable computer

Charles Babbage (1791 – 1871)

- Difference engine: an automatic mechanical calculator designed to tabulate polynomial functions (eg. $x^2 - 2x + 3$) …2,3,6,11,18…..)
- Analytical Engine: Incorporated an arithmetic logic unit, control flow in the form of conditional branching and loops, and integrated memory, making it the first design for a general-purpose computer (Turing Complete)





Ada Lovelace (1815-52)

- Was born Ada Byron, the daughter of Lord Byron.
- She was educated to be a mathematician because her mother worried she might turn out to be a poet like her father. (Poets were the 19th century punk rockers)
- Lovelace worked closely with Charles Babbage on the design of his ‘analytical engine’, a machine for performing mathematical calculations
- Regarded as the first computer programmer
  - The first to recognize that the machine had applications beyond pure calculation
  - Published the first algorithm intended to be carried out by such a machine
- Died aged 36, buried in Nottingham (next to her father)

Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data.			Working Variables.									Result Variables.			
						$^1V_1$ 0 0 0 1 [1]	$^1V_2$ 0 0 0 2 [2]	$^1V_3$ 0 0 0 4 [n]	$^0V_4$ 0 0 0 0 [ ]	$^0V_5$ 0 0 0 0 [ ]	$^0V_6$ 0 0 0 0 [ ]	$^0V_7$ 0 0 0 0 [ ]	$^0V_8$ 0 0 0 0 [ ]	$^0V_9$ 0 0 0 0 [ ]	$^0V_{10}$ 0 0 0 0 [ ]	$^0V_{11}$ 0 0 0 0 [ ]	$^0V_{12}$ 0 0 0 0 [ ]	$^0V_{13}$ ..... 0 0 0 0 [ ]	$^1V_{21}$ B_1 in a decimal fraction. [B_1]	$^1V_{22}$ B_2 in a decimal fraction. [B_2]	$^1V_{23}$ B_3 in a decimal fraction. [B_3]	$^0V_{24}$... 0 0 0 0 [B_7]
1	×	$^1V_2 \times {}^1V_3$	$^1V_4, {}^1V_5, {}^1V_6$	$\begin{cases} {}^1V_2 = {}^1V_2 \\ {}^1V_3 = {}^1V_3 \end{cases}$	$= 2n$	...	2	n	$2n$	$2n$	$2n$											
2	−	$^1V_4 - {}^1V_1$	$^2V_4$	$\begin{cases} {}^1V_4 = {}^2V_4 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 2n-1$	1	...	...	$2n-1$													
3	+	$^1V_5 + {}^1V_1$	$^2V_5$	$\begin{cases} {}^1V_5 = {}^2V_5 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 2n+1$	1	...	...	...	$2n+1$												
4	+	$^2V_5 + {}^2V_4$	$^1V_{11}$	$\begin{cases} {}^2V_5 = {}^0V_5 \\ {}^2V_4 = {}^0V_4 \end{cases}$	$= \dfrac{2n-1}{2n+1}$	...	...	...	0	0	...	...	...	...	...	$\dfrac{2n-1}{2n+1}$						
5	÷	$^1V_{11} \div {}^1V_2$	$^2V_{11}$	$\begin{cases} {}^1V_{11} = {}^2V_{11} \\ {}^1V_2 = {}^1V_2 \end{cases}$	$= \dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1}$	...	2	...	...	...	...	...	...	...	...	$\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1}$						
6	−	$^0V_{13} - {}^2V_{11}$	$^1V_{12}$	$\begin{cases} {}^2V_{11} = {}^0V_{11} \\ {}^0V_{13} = {}^1V_{13} \end{cases}$	$= -\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} = A_0$	...	...	...	...	...	...	...	...	...	...	0	.....	$-\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} = A_0$				
7	−	$^1V_3 - {}^1V_1$	$^1V_{10}$	$\begin{cases} {}^1V_3 = {}^1V_3 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= n-1\ (=3)$	1	...	n	...	...	...	...	...	...	$n-1$							
8	+	$^1V_2 + {}^0V_7$	$^1V_7$	$\begin{cases} {}^1V_2 = {}^1V_2 \\ {}^0V_7 = {}^1V_7 \end{cases}$	$= 2+0=2$	...	2	...	...	...	...	2										
9	+	$^1V_6 + {}^1V_7$	$^2V_{11}$	$\begin{cases} {}^1V_6 = {}^1V_6 \\ {}^0V_{11} = {}^2V_{11} \end{cases}$	$= \dfrac{2n}{2} = A_1$	...	...	...	...	...	$2n$	2	...	...	...	$\dfrac{2n}{2} = A_1$						
10	×	$^1V_{21} \times {}^2V_{11}$	$^1V_{12}$	$\begin{cases} {}^1V_{21} = {}^1V_{21} \\ {}^2V_{11} = {}^2V_{11} \end{cases}$	$= B_1 \cdot \dfrac{2n}{2} = B_1 A_1$	...	...	...	...	...	...	...	...	...	...	$\dfrac{2n}{2} = A_1$	$B_1 \cdot \dfrac{2n}{2} = B_1 A_1$	.....	B_1			
11	+	$^1V_{12} + {}^1V_{13}$	$^2V_{13}$	$\begin{cases} {}^1V_{12} = {}^0V_{12} \\ {}^1V_{13} = {}^2V_{13} \end{cases}$	$= -\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} + B_1 \cdot \dfrac{2n}{2}$ ...	...	...	...	...	...	...	...	...	...	...	.....	0	$\left\{-\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} + B_1 \cdot \dfrac{2n}{2}\right\}$				
12	−	$^1V_{10} - {}^1V_1$	$^2V_{10}$	$\begin{cases} {}^1V_{10} = {}^2V_{10} \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= n-2\ (=2)$	1	...	...	...	...	...	...	...	...	$n-2$							
13	−	$^1V_6 - {}^1V_1$	$^2V_6$	$\begin{cases} {}^1V_6 = {}^2V_6 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 2n-1$	1	...	...	...	...	$2n-1$											
14	+	$^1V_1 + {}^1V_7$	$^2V_7$	$\begin{cases} {}^1V_1 = {}^1V_1 \\ {}^1V_7 = {}^2V_7 \end{cases}$	$= 2+1=3$	1	...	...	...	...	...	3										
15	+	$^2V_6 + {}^2V_7$	$^1V_8$	$\begin{cases} {}^2V_6 = {}^2V_6 \\ {}^2V_7 = {}^2V_7 \end{cases}$	$= \dfrac{2n-1}{3}$	...	...	...	...	...	$2n-1$	3	$\dfrac{2n-1}{3}$									
16	×	$^1V_8 \times {}^3V_{11}$	$^4V_{11}$	$\begin{cases} {}^1V_8 = {}^0V_8 \\ {}^3V_{11} = {}^4V_{11} \end{cases}$	$= \dfrac{2n}{2} \cdot \dfrac{2n-1}{3}$	...	...	...	...	...	...	...	0	...	...	$\dfrac{2n}{2} \cdot \dfrac{2n-1}{3}$						
17	−	$^2V_6 - {}^1V_1$	$^3V_6$	$\begin{cases} {}^2V_6 = {}^3V_6 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 2n-2$	1	...	...	...	...	$2n-2$											
18	+	$^1V_1 + {}^2V_7$	$^3V_7$	$\begin{cases} {}^2V_7 = {}^3V_7 \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= 3+1=4$	1	...	...	...	...	...	4										
19	+	$^3V_6 + {}^3V_7$	$^1V_9$	$\begin{cases} {}^3V_6 = {}^3V_6 \\ {}^3V_7 = {}^3V_7 \end{cases}$	$= \dfrac{2n-2}{4}$	...	...	...	...	...	$2n-2$	4	...	$\dfrac{2n-2}{4}$	...	$\left\{\dfrac{2n}{2} \cdot \dfrac{2n-1}{3} \cdot \dfrac{2n-2}{3} = A_2\right\}$						
20	×	$^1V_9 \times {}^4V_{11}$	$^5V_{11}$	$\begin{cases} {}^1V_9 = {}^0V_9 \\ {}^4V_{11} = {}^5V_{11} \end{cases}$	$= \dfrac{2n}{2} \cdot \dfrac{2n-1}{3} \cdot \dfrac{2n-2}{4} = A_3$	...	...	...	...	...	...	...	...	0								
21	×	$^1V_{22} \times {}^5V_{11}$	$^2V_{12}$	$\begin{cases} {}^1V_{22} = {}^1V_{22} \\ {}^5V_{12} = {}^2V_{12} \end{cases}$	$= B_3 \cdot \dfrac{2n}{2} \cdot \dfrac{2n-1}{3} \cdot \dfrac{2n-2}{3} = B_3 A_3$	...	...	...	...	...	...	...	...	...	...	0	$B_3 A_3$	.....	.....	B_3		
22	+	$^2V_{12} + {}^2V_{13}$	$^3V_{13}$	$\begin{cases} {}^2V_{12} = {}^0V_{12} \\ {}^2V_{13} = {}^3V_{13} \end{cases}$	$= A_0 + B_1 A_1 + B_2 A_2$	...	...	...	...	...	...	...	...	...	...	.....	0	$\left\{A_3 + B_1 A_1 + B_2 A_2\right\}$			
23	−	$^2V_{10} - {}^1V_1$	$^3V_{10}$	$\begin{cases} {}^2V_{10} = {}^3V_{10} \\ {}^1V_1 = {}^1V_1 \end{cases}$	$= n-3\ (=1)$	1	...	...	...	...	...	...	...	...	$n-3$							

Here follows a repetition of Operations thirteen to twenty-three.

24	+	$^4V_{13} + {}^0V_{24}$	$^1V_{24}$	$\begin{cases} {}^4V_{13} = {}^0V_{13} \\ {}^0V_{24} = {}^1V_{24} \end{cases}$	$= B_7$	...	...	...	...	...	...	...	...	...	...	.....	.....	.....	.....	.....	.....	B_7
25	+	$^1V_1 + {}^1V_3$	$^1V_3$	$\begin{cases} {}^1V_1 = {}^1V_1 \\ {}^1V_3 = {}^1V_3 \\ {}^4V_6 = {}^0V_6 \\ {}^4V_7 = {}^0V_7 \end{cases}$	$= n+1 = 4+1 = 5$ ..... by a Variable-card. by a Variable card.	1	...	$n+1$	...	...	0	0										

```python
from fractions import Fraction

# my implementation of the algorithm
def bernoulli():

  # results
  Bs = list()

  # start at n=1
  n = 1
  # calculate the sequence
  while True:

    # result = A0
    r = -Fraction(2 * n - 1, 2 * n + 1) / 2

    # A1 = n
    A = n
    # for each B[k] already determined calculate the corresponding A[k]
    for (i, B) in enumerate(Bs):
      if i > 0:
        # multiply in the 2 additional terms
        j = 2 * i - 1
        A *= Fraction(2 * n - j, 2 + j)
        j += 1
        A *= Fraction(2 * n - j, 2 + j)
      # add A[k] * B[k] into the result
      r += A * B

    # the computed bernoulli number is -r
    B = -r
    # return the number
    yield B
    # add it to the result list
    Bs.append(B)
    # increase n
    n += 1

# run N iterations of the algorithm
N = 10

# allow N to be specified as a command line argument
import sys
if len(sys.argv) > 1:
  N = int(sys.argv[1])

k = 1 # ada's numbering (normal numbering is k + 1)
K = 2 * N + 1 # max k
for r in bernoulli():
  print("B[{k}] = {r}".format(k=k, r=r))
  k += 2
  if k == K: break
```

$$B_m^- = \sum_{k=0}^{m} \sum_{v=0}^{k} (-1)^v \binom{k}{v} \frac{v^m}{k+1}$$

$$B_m^+ = \sum_{k=0}^{m} \sum_{v=0}^{k} (-1)^v \binom{k}{v} \frac{(v+1)^m}{k+1}.$$

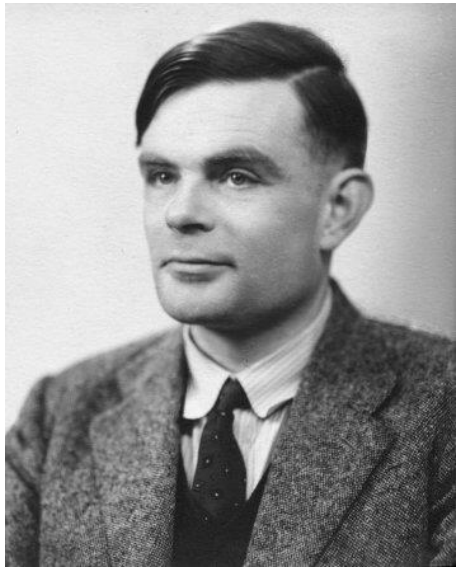
- (Python version)
- Probably true to say it was the first example of a **complex** program

https://enigmaticcode.wordpress.com/tag/bernoulli-numbers/

Universality

Same **hardware** can run many different **software** programs

Theory



Alan Turing:

Universal Turing Machine

Practice

John Von Nuemann:

Stored Program Computer

Hilbert's Problems

- In 1900, David Hilbert, the German mathematician, proposed 23 problems in mathematics
  - Ten of them were presented at the Paris conference of the International Congress of Mathematicians
  - Complete list was published later
  - The most important problems in mathematics – all unsolved at that time
- #2 – Prove that the axioms of arithmetic are consistent
- Reframed in 1928:

1. Is mathematics complete?
2. Is mathematics consistent?
3. Is mathematics decidable?

Gödel's Incompleteness Theorems

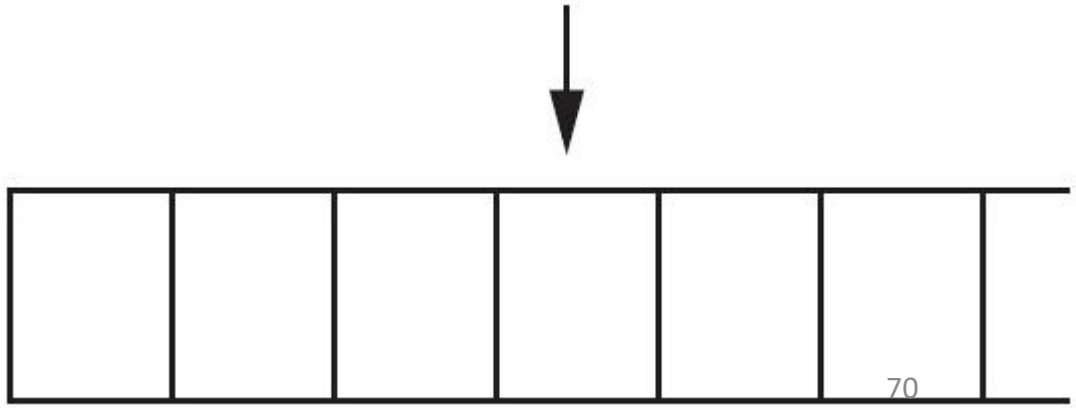
- In 1931, Kurt Gödel, the Austrian mathematician, proposed the solutions to (1) and (2)
- Gödel's incompleteness theorems
- Is mathematics complete? (Can derive every formula that is true)
  - **If the system is consistent, it cannot be complete – there will always be statements that are true but not provable**
- Is mathematics consistent? (Does not contain any internal contradictions)
  - **The consistency of the axioms cannot be proven within the system (a system cannot demonstrate its own consistency)**

Entscheidungsproblem

- Is mathematics decidable? – now known as the **entscheidungsproblem** (German for "decision problem")
- Is there a general algorithm to determine whether a mathematical conjecture is true or false?
- Church & Turing (independently, 1935-1937)
  - Showed the “decision problem” is unsolvable
  - Turing proved this by imagining a “Universal Machine”
  - This result is now known as Church's Theorem or the Church-Turing Theorem

Turing Machine

- Theoretical machine (imaginary / pencil and paper)
- Infinite tape separated into positions (squares)
- Each position can contain a symbol or be blank
- Tape head points to current position
  - Can read symbol
  - Can alter symbol / move – following a rule
  - The set of “rules” is a “program”
- Given some input the Machine will halt (or not) in a particular state
- Used to show the decision problem can’t be solved
  - E.g., the “halting problem”

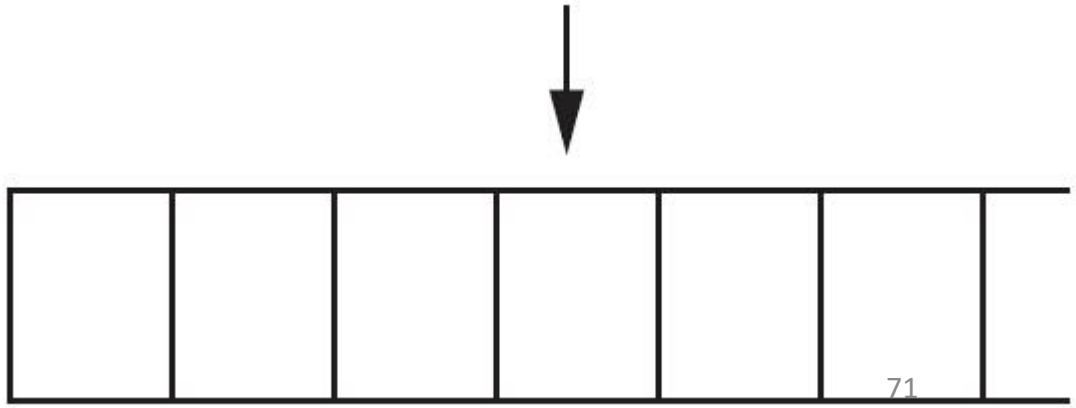


Turing Machine

- The tape may serve as:

1. The input device (the input string is assumed to be on the tape initially)
2. The memory available for use during the computation
3. The output device (the output, if it is relevant, is the string of symbols left on the tape at the end of the computation

- A Turing machine will have two **halt** states, one denoting acceptance and the other rejection
- If a TM decides to accept or reject the input string, it stops. But it might not decide to do this, and so it might continue **moving forever**



The Halting Problem

- Determine, from a description of an arbitrary program and input, whether the program will finish running (**halt**) or continue to run forever (**not halt**)
- Turing 1936: a **general algorithm** to solve the Halting Problem for **all possible** program-input pairs **cannot exist**
- Key part of proof: mathematical definition of computer and program (Turing Machines)
- The Halting problem is **undecidable** over Turing Machines

Genius of Turing

- TM is a fantastic abstract idea to solve a difficult abstract problem
- So, a small group of pure mathematicians (logicians) were very impressed...
- But who else cares? Or even understands?
- In 1936: not many people!
- In fact, it was possible that Turing's ideas would be lost in the pages of a pure mathematics journal for many years
- Until…

World War II

- In 1939, the WW2 began
- It involved the vast majority of the world's countries—including all of the great powers
- Majority of world’s population directly affected
- Estimated that more than 50 million killed during WW2
- Main instigators:
  - Germany in Europe (with their allies, e.g., Italy)
  - Japan in Pacific region

1940

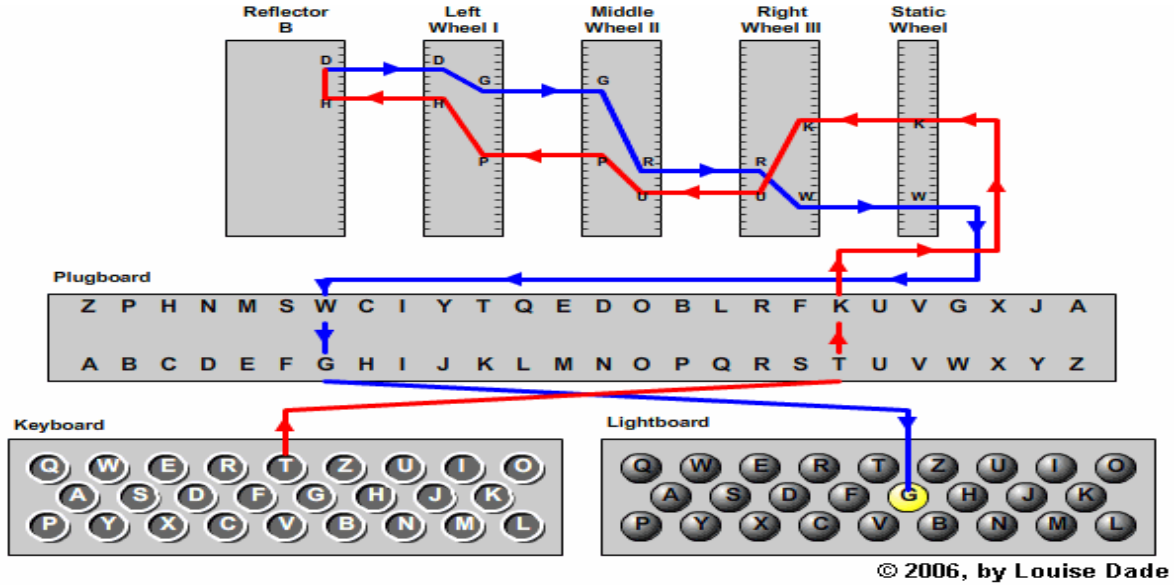
- The situation:
- Europe in hands of Germany (and their allies)
- Britain looking likely to fall next
  - “Battle of Britain” (1963 British planes against 4074 German planes)
- Japan dominating Pacific region
  - (China, Korea, Malaysia, Singapore, Vietnam, Thailand, Indonesia, etc…)
- USA, USSR (Russia) not yet involved
  - And do not want to get involved!

The Secret War

- While battle raged in the open a “secret” war was being fought
- Military messages are sent encrypted
- If messages can be deciphered, possible to know what the enemy is planning
- British (and their allies) goal: To decrypt German (and Japanese) code
- How hard can it be?

Enigma Machine

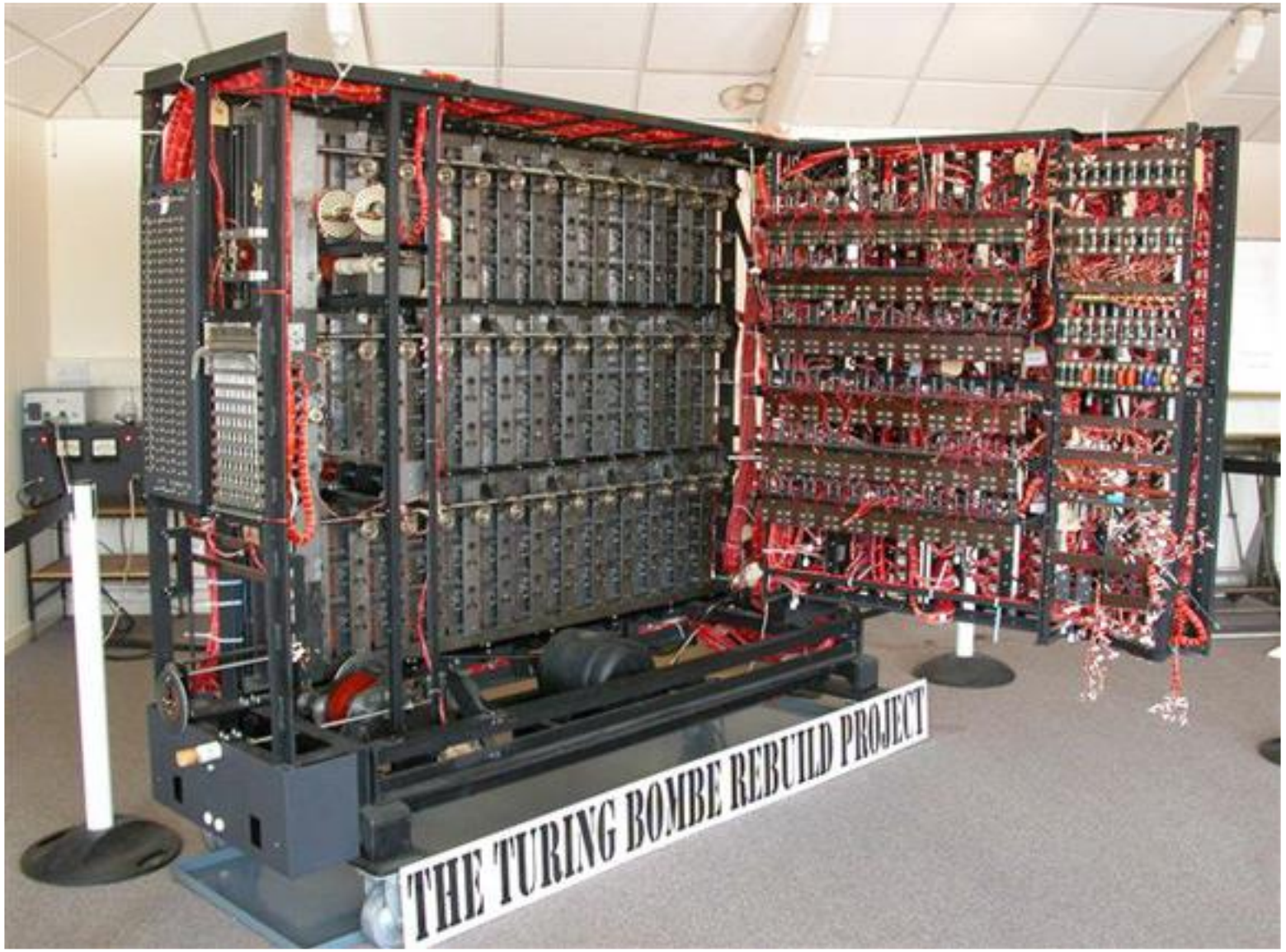
- To encrypt German messages
- “Unbreakable”
- 158 x10^{18} (158 million million million) settings!
- Settings changed every day!
- Japan had something similar
  - Purple machine, based on Enigma



Breaking Enigma

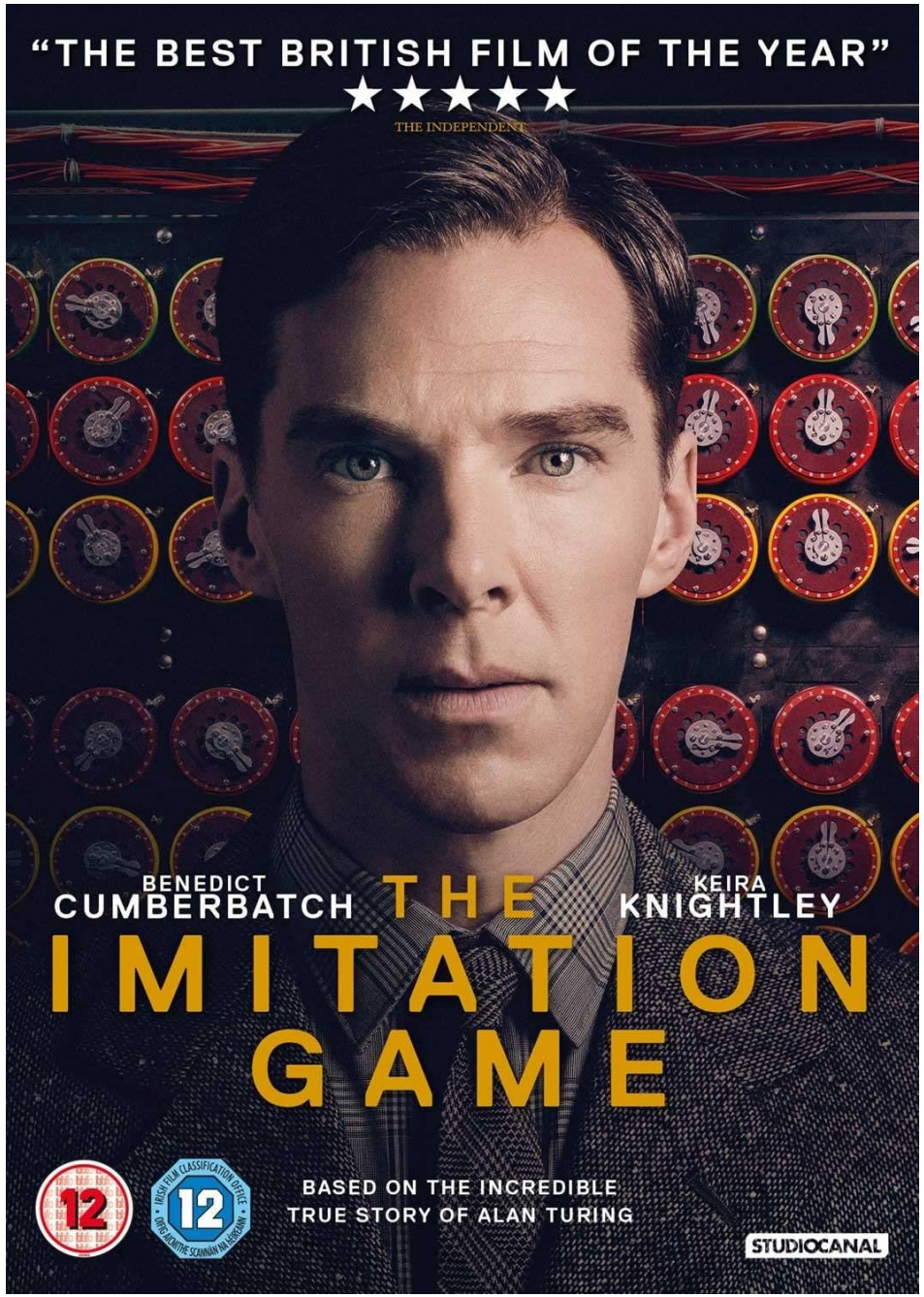
- Bletchley Park was the central site for British codebreakers during WW2
- Teams of people (thousands):
  - Capturing German messages
  - Attempting to decrypt messages
- Top secret
  - Nobody knows this is happening
- Difficult problem. Struggling to succeed
- Turing and others (this was not done alone, despite what movies may say) built a machine to break the enigma code
  - Using abstract concepts from his Universal machine

Breaking Enigma



Breaking Enigma

- Breaking the “unbreakable” code
- Success! Enigma was broken
  - Japanese codes also broken
- Estimated shortening of war by 2 years
- Estimated 14-21 million lives saved!
- Outcome:
  - The first “computer”
  - An actual, physical computing device



Computer Science Born

- After the war ended (1945), computers began to be built and studied in universities
- Although the effort in Bletchley Park was secret
  - People who worked on breaking Enigma had learned a lot about how to build computing machines
  - They developed this knowledge at universities
  - Turing went to Manchester University, England, UK

Turing Test

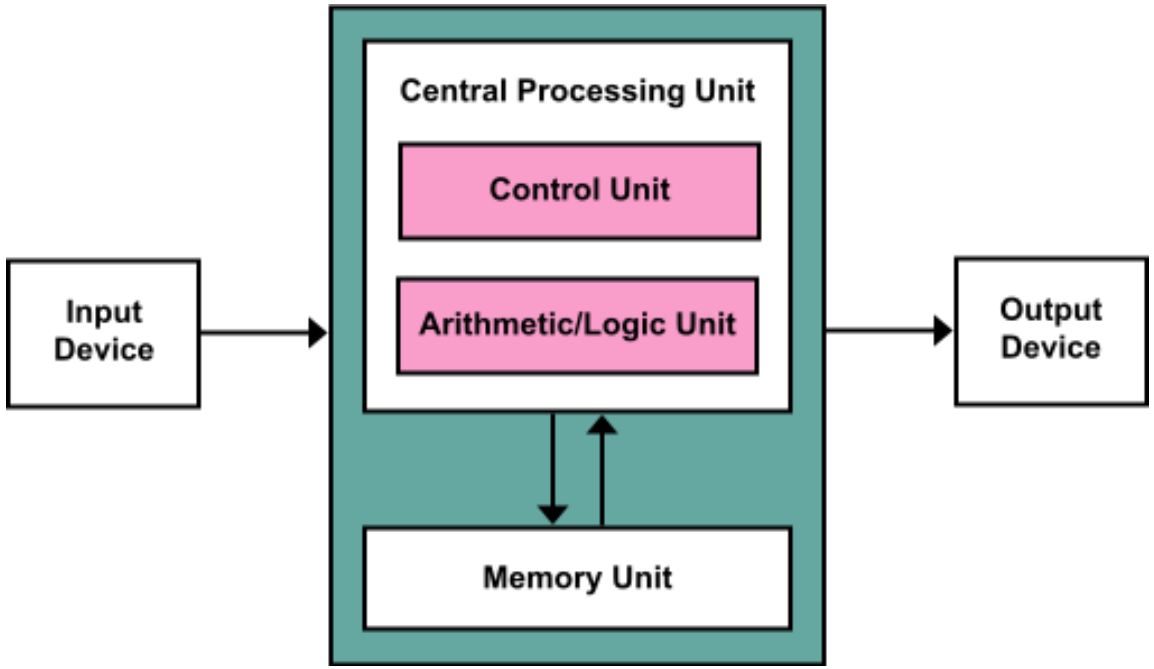
- Turing had lots of visionary ideas
- Considered computer “intelligence”
  - One of the pioneers of Artificial Intelligence (AI)
- How to decide whether a computer is intelligent – the Turing Test
  - Identify whether you are talking to a computer or a person?
- One of the most important people in the history of computer science
- https://www.youtube.com/watch?v=yv_8dx7g-WA

John Von Neumann

- Von Neumann
  - Mathematics (foundations of mathematics, functional analysis, ergodic theory, group theory, representation theory, operator algebras, geometry, topology, and numerical analysis)
  - Physics (quantum mechanics, hydrodynamics, and quantum statistical mechanics),
  - Economics (game theory)
  - Computing (Von Neumann architecture, linear programming, self-replicating machines, stochastic computing)
  - Statistics

Von Neumann Architecture

- Von Neumann Architecture, 1945 (von Neumann model or Princeton architecture):
  - A processing unit that contains an arithmetic logic unit and processor registers
  - A control unit that contains an instruction register and program counter
  - Memory that stores data and instructions
  - External mass storage
  - Input and output mechanisms
- The term "von Neumann architecture" has evolved to mean any stored-program computer in which an instruction fetch and a data operation cannot occur at the same time



Summary

- Program Execution
  - PC
  - Fetch-Decode-Execute
- Cache
- Hack Computer
  - Instruction memory (ROM)
  - Data memory (RAM)
- CS Pioneers
  - Turing
  - Neumann