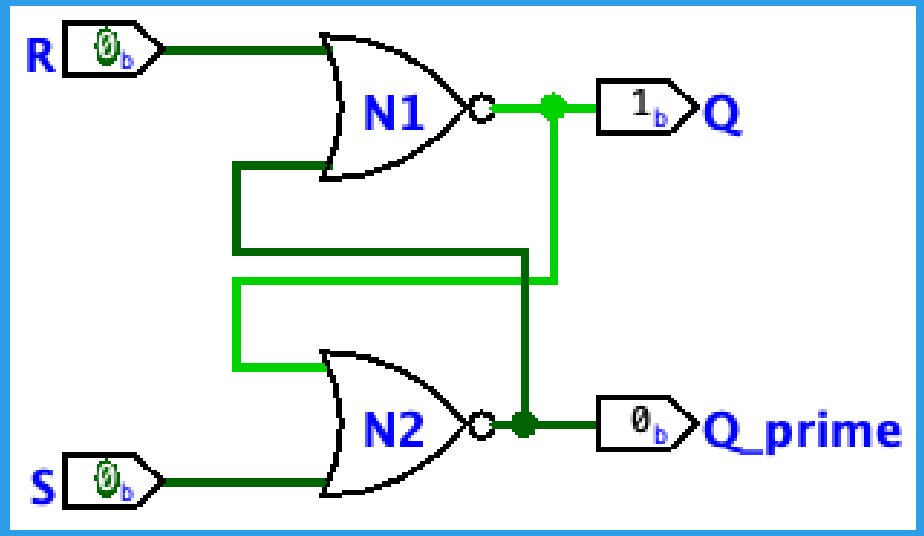
Computer Systems



Sequential Circuits

CS51 – Fall 2025

Sequential circuits

- Last lecture, all circuits we saw were **combinational**: their output depends on the present inputs and there is no notion of memory of past inputs.
- Beyond combinational circuits, there are also **sequential circuits**.
- Sequential circuits not only depend on current input but have memory components that allow them to utilize prior inputs, too.
- Storing data into memory is called **writing** while retrieving data from memory is called **reading**.

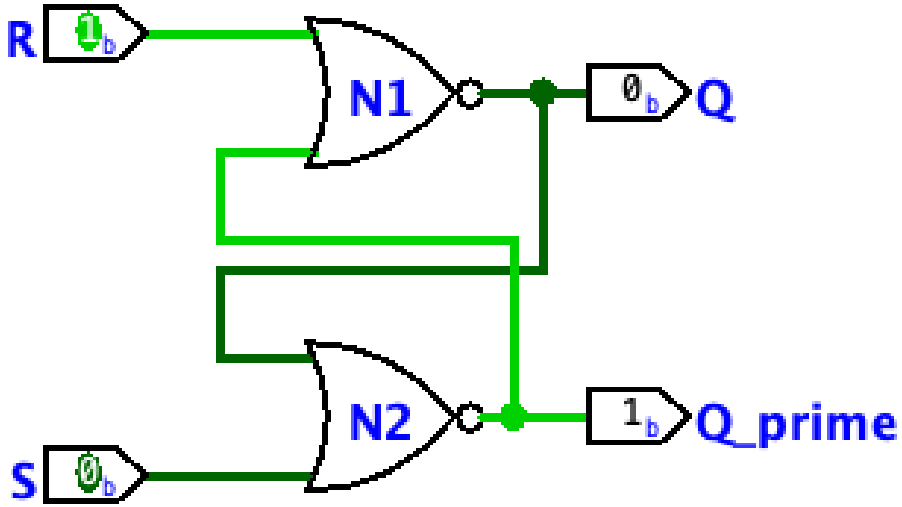
SR-Latches

SR-latch

- We start by building an SR-latch which *latches onto* one bit of information.
- An SR-latch is a basic sequential circuit with two binary inputs, S and R, called set and reset, respectively, and two binary outputs, Q, and its complement Q'.
- When $S = 1$ and $\mathrm{R} = 0$, it **sets** Q to 1 (and its complement Q' to 0).
- When reset, $R = 1$ and $S = 0$ it **resets** Q to 0 (and its complement Q' to 1).
- When both set and reset are zero, that is $S = 0$ and $\mathrm{R} = 0$, Q remembers its old value (that is, act as memory!!)
- The case when both set and reset are 1, that is $S = 1$ and $\mathrm{R} = 1$, is weird and we output 0 for both Q and Q'.

Building an SR-latch

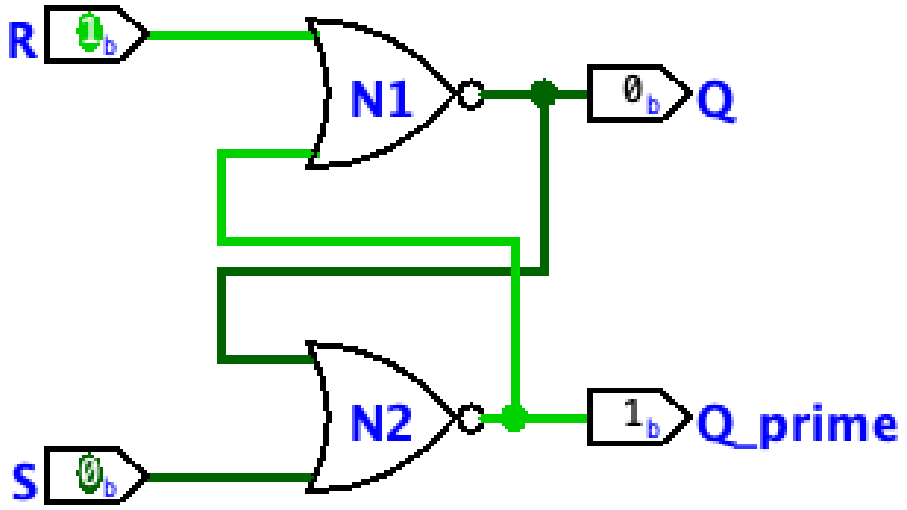
- We will use two **cross-coupled** NOR gates. For the first NOR gate, one of the inputs will be R. Similarly, for the second NOR gate, one of the inputs will be S.
- Cross-coupling means that the second input for the first NOR gate will be the output of the second NOR gate and vice-versa.
- Here's a visualization of the circuit. We will next build the truth table for all combinations of the inputs R and S and their impact on Q and its complement Q'.



SR-latch – Case I: $R = 1, S = 0$

- Remember, NOR is the negation of the OR gate: it produces TRUE/1 only when both its inputs are FALSE/0.
- The NOR gate N_1 sees that R is 1 so it produces 0 for Q.
- The NOR gate N_2 sees both Q and S being 0 so it produces 1 for Q'.

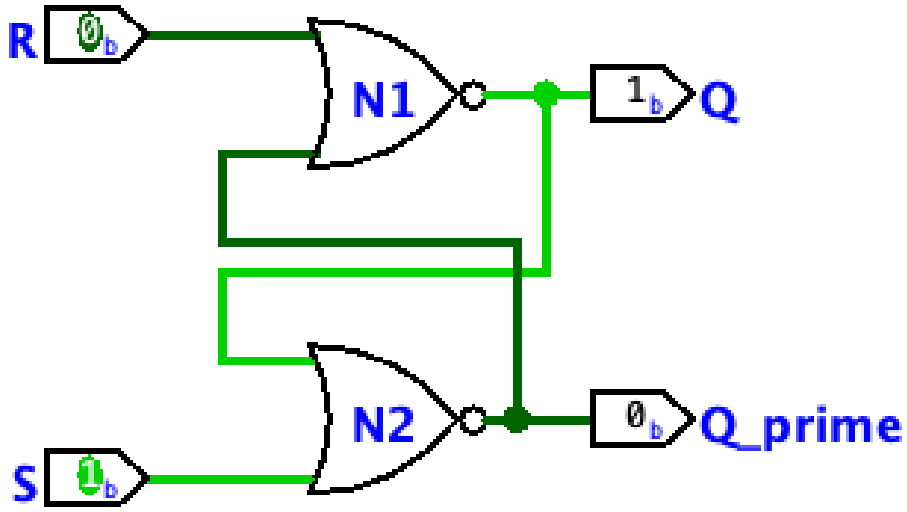
Case	Input		Output	
	S	R	Q	Q'
IV	0	0		
I	0	1		
II	1	0		
III	1	1		



SR-latch – Case II : $R = 0, S = 1$

- The NOR gate N_1 receives the inputs of R and Q'. R is 0 but we don't know the value of Q' yet.
- The NOR gate N_2 sees S being 1 so it produces 0 for Q'.
- Going back to N_1, since both R and Q' are 0, it produces 1 for Q.

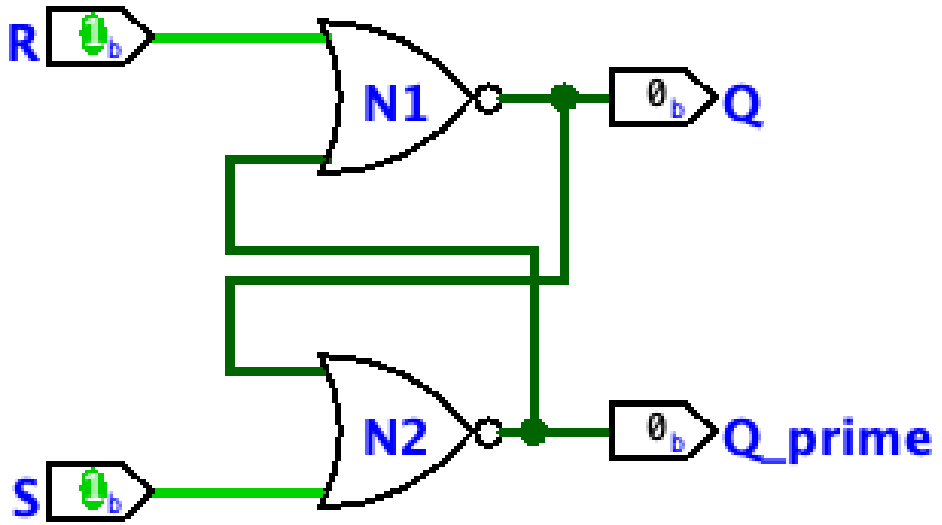
Case	Input		Output	
	S	R	Q	Q'
IV	0	0		
I	0	1	0	1
II	1	0		
III	1	1		



SR-latch – Case III : $R = 1, S = 1$

- Both NOR gates N_1 and N_2 receive at least one input being 1 (R and S, respectively) so they can immediately produce 0 for Q and Q', respectively.

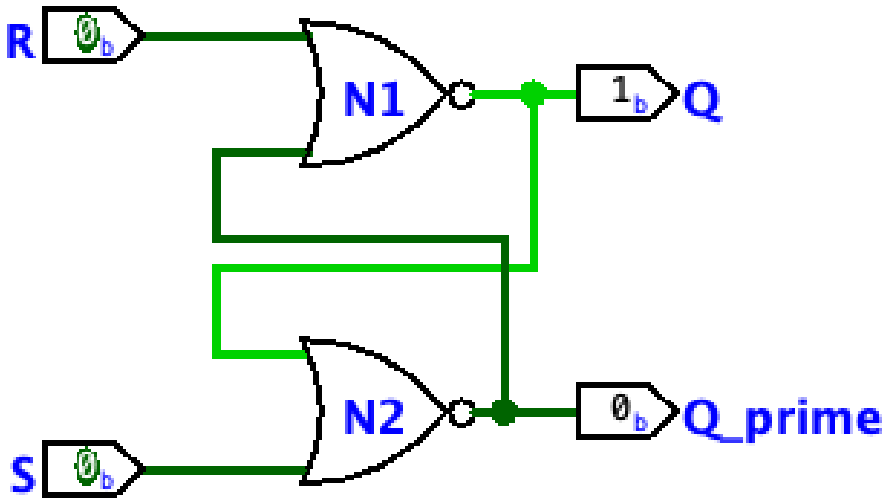
Case	Input		Output	
	S	R	Q	Q'
IV				
I	0	1	0	1
II	1	0	1	0
III	1	1		



SR-latch – Case IV : $R = 0, S = 0$

- Both NOR gates N_1 and N_2 receive at least one input being 0 (R and S, respectively) and are stuck because they both don't know what happens with their other input, Q and Q'.
- **Case IVa: $Q = 0$.**
  - Because both S and Q are 0, N_2 produces 1 for Q'. N_1 receives one 1 input through Q' so it outputs 0 for Q as we had assumed.
- **Case IVb: $Q = 1$.**
  - Because Q is 1, N_2 produces 0 for Q'. N_1 receives $R = 0$ and $Q' = 0$ so it outputs 1 for Q as we had assumed.

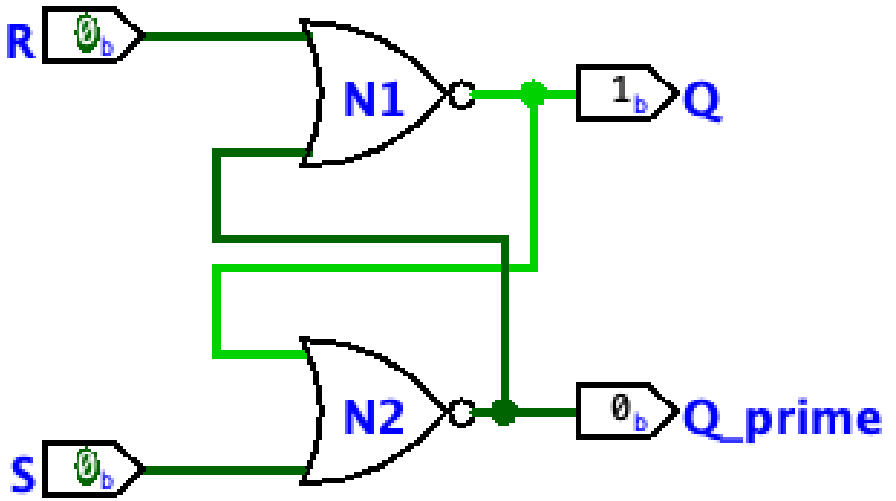
Case	Input		Output	
	S	R	Q	Q'
IV	0	0		
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0



SR-latch – Case IV : $R = 0, S = 0$

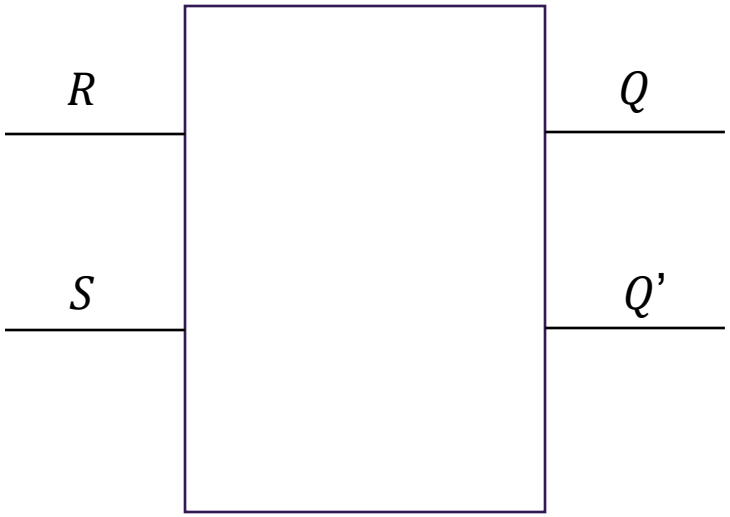
- Putting all this together, suppose Q has some known prior value, which we will call $Q_{previous}$, before we enter Case IV.
- $Q_{previous}$ is either 0 or 1 and represents the state of the system.
- When R and S are both 0, Q will remember this old value, $Q_{previous}$, and Q' will be its complement, $Q'_{previous}$.
- This circuit has memory! It remembers one bit of information.

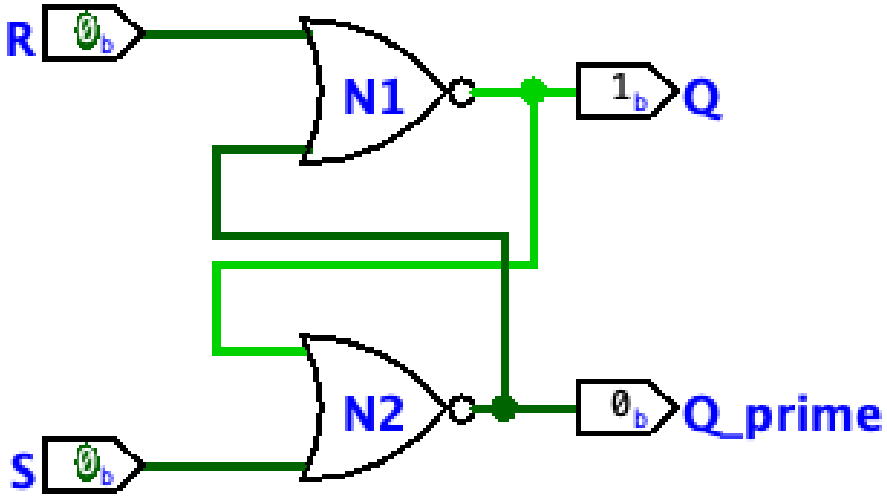
Case	Input		Output	
	S	R	Q	Q'
IV	0	0	$Q_previous$	$Q'_previous$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0



SR-latch - Abstraction

- As an application of abstraction, instead of seeing the full circuit on the right, you might encounter the following symbol for SR-latches:





Registers

From latches to registers

- One latch can store one bit of information.
- There are more advanced circuits that store 1 bit of information, such as D-latches and flip flops.
- If we group together n such circuits that store 1 bit we have a **register**.
- The **width** of the register is the n bits we can store which corresponds to n underlying circuits that each stores 1 bit of information.
- Registers are the key building block of sequential circuits.
- Registers today are 32 or even 64 bits and are built directly into a CPU.
- Remember that they store a relatively small number of bits, are extremely fast, and can be synchronized with a clock.

Memory

When you need more memory

- The sequential circuits we have seen so far (latches, flip-flops, and registers) have memory components that can store small amounts of data from a single to a relatively small number of bits.
- Computers require a lot more memory and they do so efficiently by utilizing **memory arrays**, two-dimensional arrays of **memory cells**, each storing one bit of information:

0	1	0	1	1	1	0	1
1	1	0	0	0	0	1	1
0	1	1	1	1	0	0	1
1	1	0	1	1	1	1	1
0	1	1	0	0	0	0	1
1	1	1	0	1	1	1	1
1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	0

Abstraction of structure of memory arrays

- You can think of a memory array as a two-dimensional array of 2^{N} rows and 2^{M} columns.
- The **depth** of a memory array is the number of rows known as the number of **words**.
- The **width** of a memory array is the number of columns, also called the word size.
- This means that our general memory array contains 2^{N} 2^{M}-bit words.
- The size of a memory array is given as depth×width= 2^{N}x 2^{M}.
- This is a 2^{3}-word× 2^{2} memory array, or simply an 8×4 array.

Data			
0	1	0	0
1	0	0	0
1	1	0	1
0	1	0	0
1	0	1	0
0	1	1	1
1	0	1	0
0	1	1	1

Address of a specific entry

- We said that the memory array consists of 2^{N} rows and 2^{M} columns.
- We can use the number N to build an **address**, an N-bit number that will specify which word to read/write.
- For example, in our 8x4 array, we would need 3 bits to create an address to reference each one of these words.
- For example, the address 010 refers to the word 1101 in the third row.

Address	Data			
000	0	1	0	0
001	1	0	0	0
010	1	1	0	1
011	0	1	0	0
100	1	0	1	0
101	0	1	1	1
110	1	0	1	0
111	0	1	1	1

Units of memory

- **Bit**: the smallest unit of memory.
- **Nibble**: 4 bits, half a byte
- **Byte**: 8 bits, the basic unit of measuring memory. Notation is B, e.g., 8B is 8 bytes or 64 bits.
- **Kilobyte**: 10^3 bytes $= 1{,}000$ bytes. Notation is KB.
  - In base 2, 1KiB (kibibyte) = 2^{10} bytes
- **Megabyte**: 10^3 Kilobytes = $1{,}000{,}000$ bytes. Notation is MB.
  - In base 2, 1MiB (mebibyte) = 2^{20} bytes
- **Gigabyte**: 10^3 Megabytes = $1{,}000{,}000{,}000$ bytes. Notation is GB.
  - In base 2, 1GiB (gibibyte) = 2^{30} bytes
- **Terabyte**: 10^3 Gigabytes = $1{,}000{,}000{,}000{,}000$ bytes. Notation is TB.
  - In base 2, 1TiB (tebibyte) = 2^{40} bytes

Calculating the size of memory

- We just said that the size of memory is depth×width= 2^N x 2^M, where N=the number of bits in one memory address and 2^M is the number of bits in a word.
- Let's say we have a memory with 1024 words and each word is 16 bits. How big is the memory in bytes, when 1 byte is equal to 8 bits?
- 1024 words can be written as a power of two as 2^{10} words.
- So far, we have the size of the memory being $2^{10} \times 16$ bits.
- 1 byte is equal to 8 bits. So that means the size of the memory is $2^{10} \times 2$ bytes.
- 2^{10} bytes ≅ 1 KB, 2^{20} bytes ≅ 1 MB, and 2^{30} bytes ≅ 1 GB.
- Therefore, our memory's size is 2 KB.

PRACTICE TIME - Calculating the size of memory

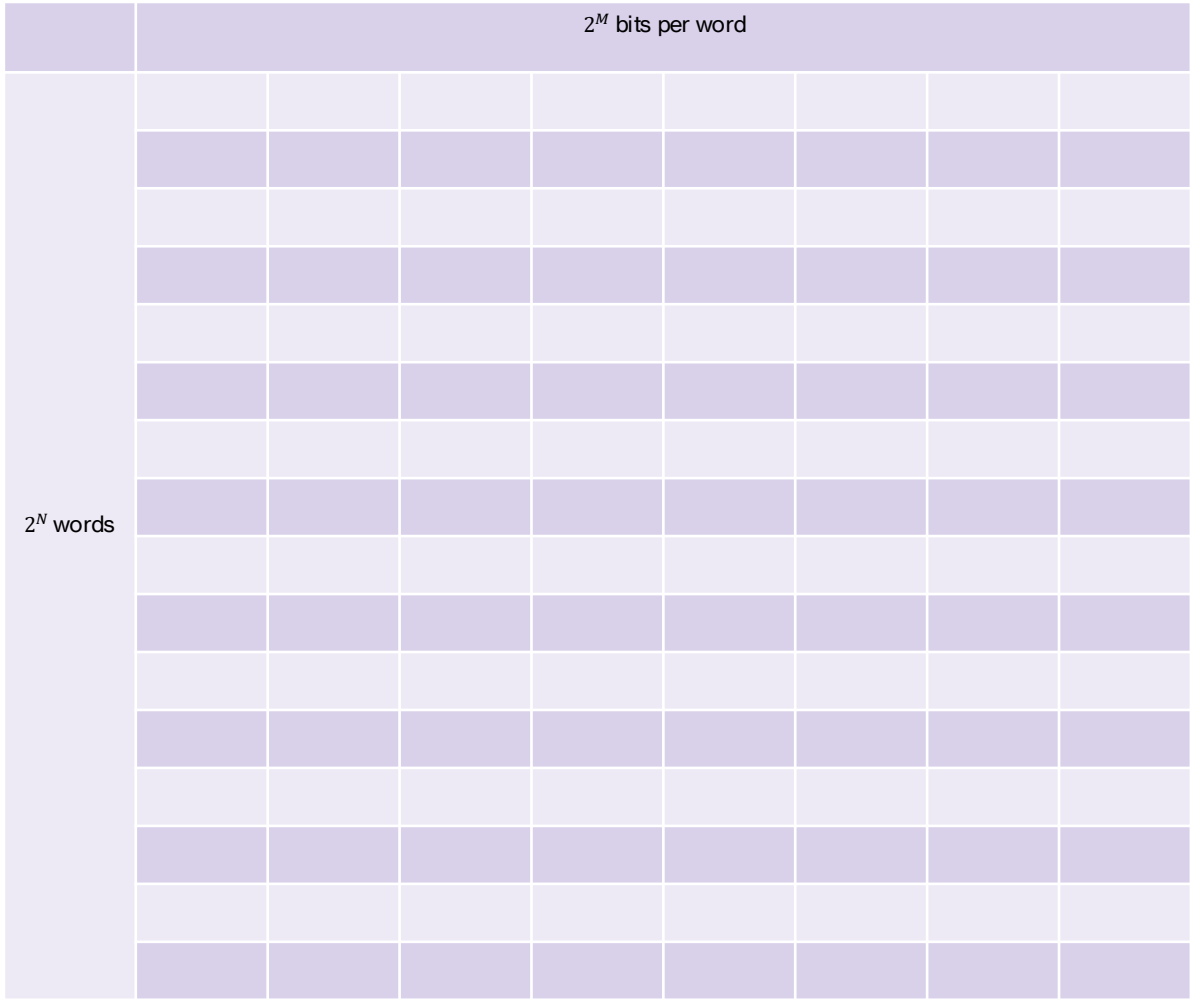
- Assume you have a memory with 2×2^{20} words, each word being 32 bits. What is the memory's size in bytes?

ANSWER - Calculating the size of memory

- Assume you have a memory with 2×2^{20} words, each word being 32 bits. What is the memory's size in bytes?
- Size = $2 \times 2^{20} \times 32$ bits = $2 \times 2^{20} \times 4$ bytes $\cong$ 8 MB.

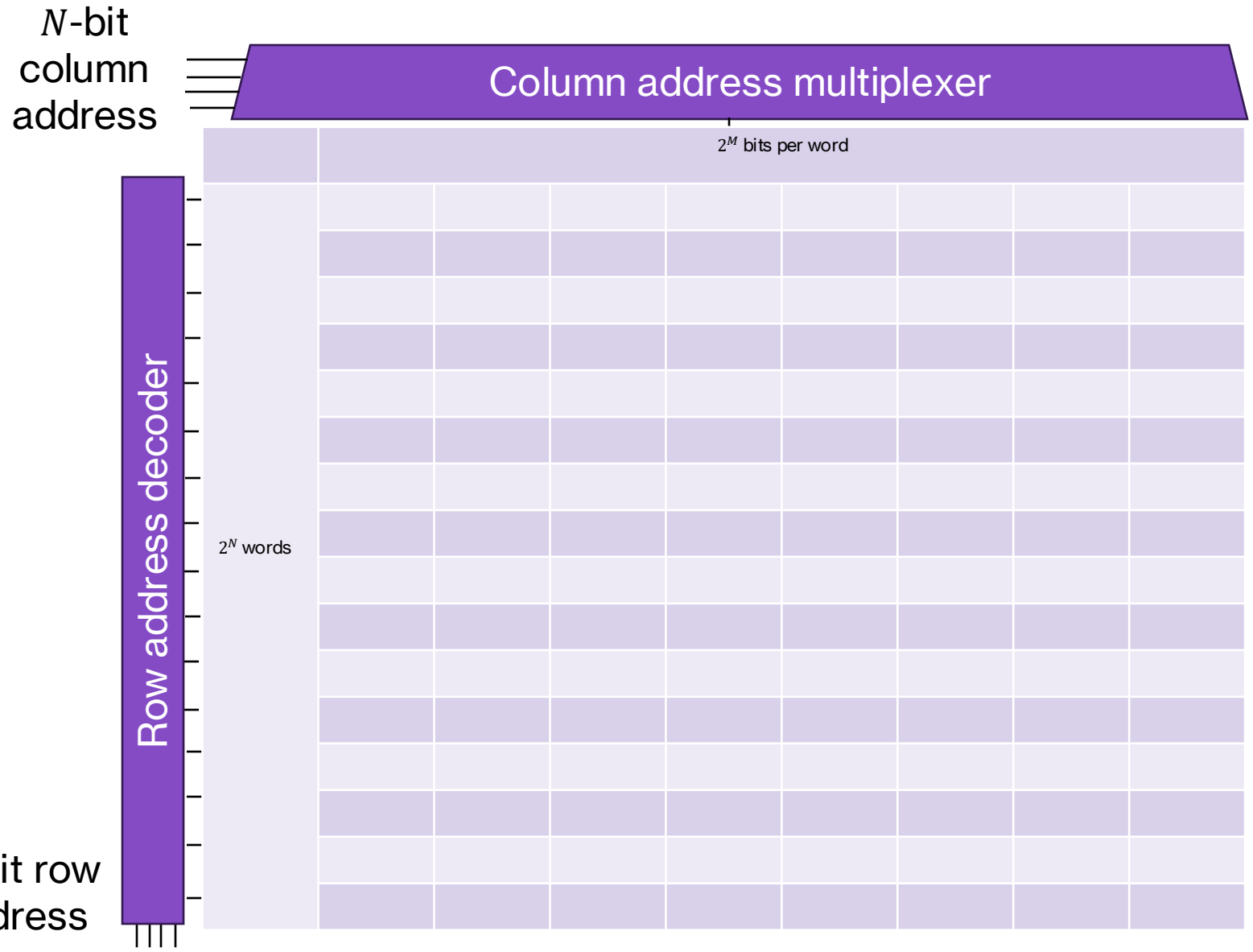
Implementing a memory array

- How do we go about implementing a 2^Nx 2^M memory array?
- We want to organize the words in 2^N rows and 2^M columns.
- We already saw that we can reference a particular row/word if we had and address of N bits. We will consider this number as the row address.
- How many bits would we need to address a particular column?
  - M. This is our column address
- We can pack together this $N + M$ bits into a single number that we can use to address a specific memory cell.



From addresses to actual memory cells

- To convert from an address to the actual memory cell that contains the desired bit, we will utilize one multiplexer and one decoder.
- The decoder will be responsible for taking the first N-bits that correspond to the row address and mapping it to one of the 2^N rows.
- The multiplexer will be responsible for taking the last M-bits that correspond to the column address and use them as selection lines to map to a single column.



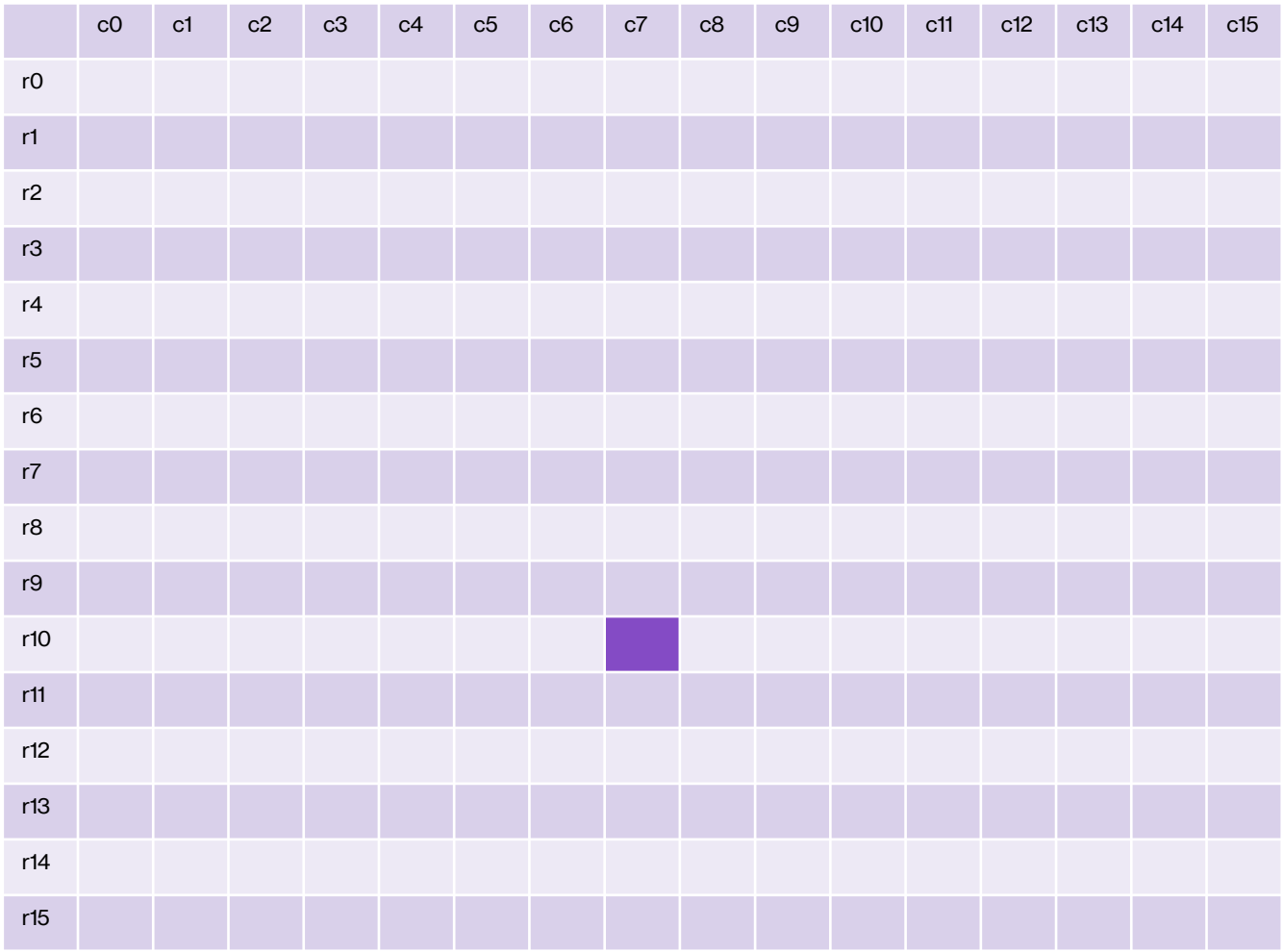
Example: Implementing a memory array

- Assume we implement a 16 x 16 memory array which can store 16 words of 16 bits each, that means it can overall store up to 256 bits.
- Since we have 16 rows and 16 columns, we will need 8 bits for our address: 4 bits to define the specific row and 4 bits to define the specific column.

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15
r0																
r1																
r2																
r3																
r4																
r5																
r6																
r7																
r8																
r9																
r10																
r11																
r12																
r13																
r14																
r15																

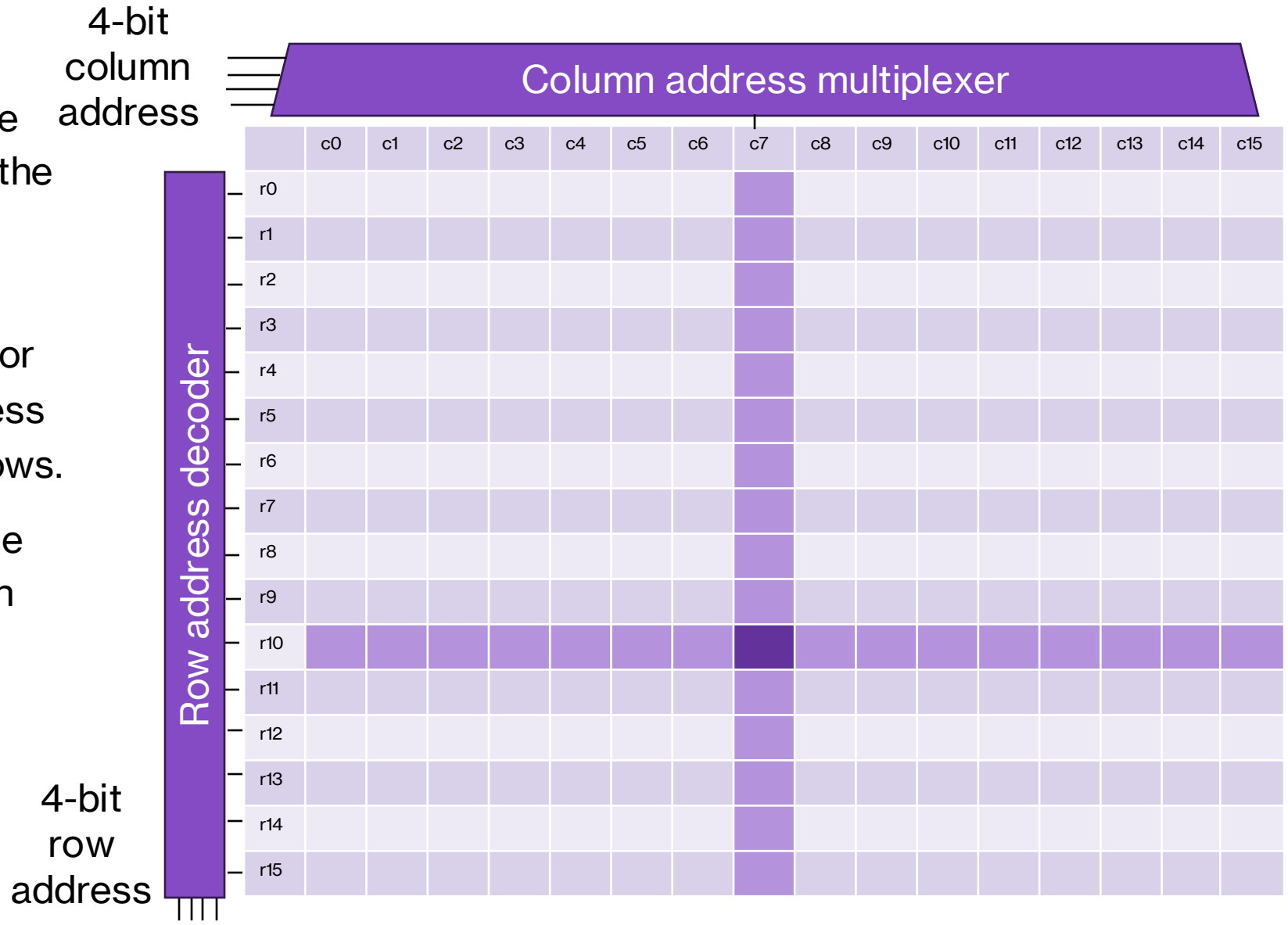
Addressing a memory cell

- For example, let's say we wanted to read the contents of the memory cell at row 10 and column 7.
- $10_{10} = 1010_2$ so we would store the row address as the 4-bit number 1010.
- $7_{10} = 0111_2$ the column address as the 4-bit number 0111.
- Putting it together, the address of the memory cell located at row 10 and column 7 is 10100111.



From addresses to actual memory cells

- To convert from an address to the actual memory cell that contains the desired bit, we will utilize one multiplexer and one decoder.
- The decoder will be responsible for taking the 4-bits of the row address and mapping it to one of the 16 rows.
- The multiplexer will be responsible for taking the 4-bits of the column address as selection lines and to map to a single column.



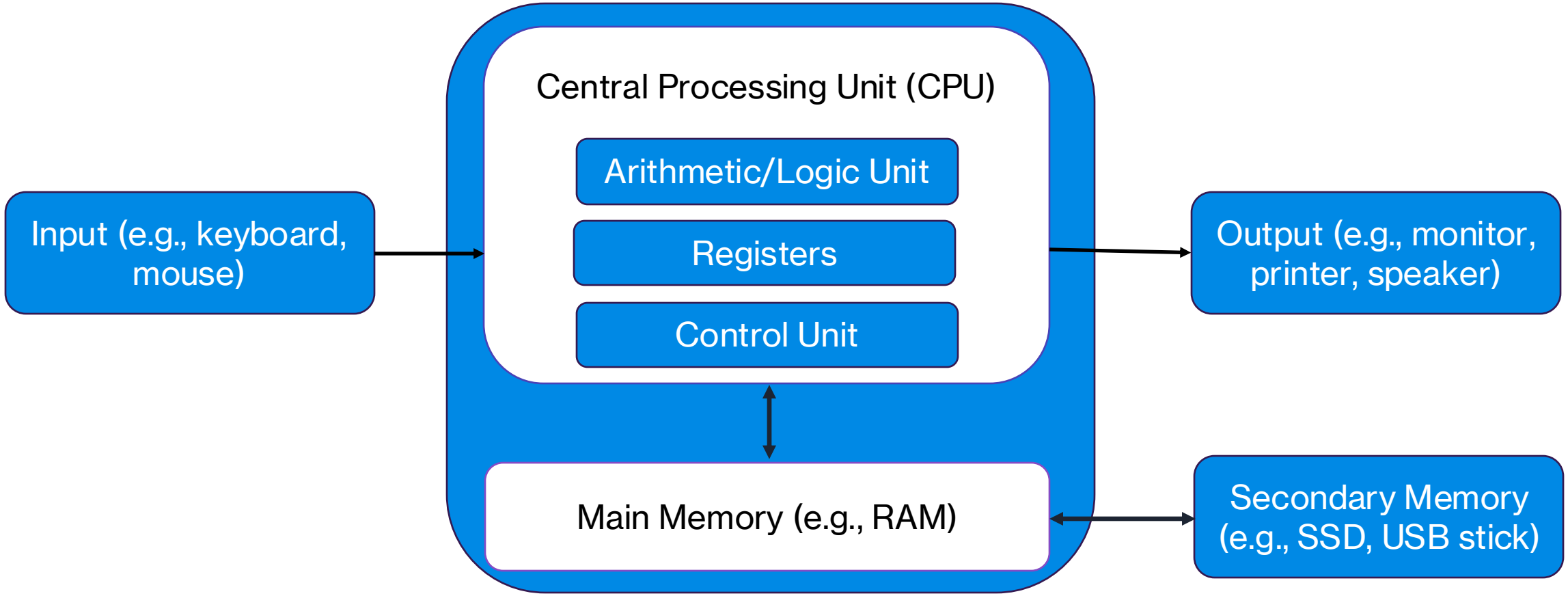
Memory core

- What we just built (with some simplifications) is one memory core.
- Memory cores are combined in bigger arrays to build increasingly bigger memory that can reach GBs.
- The actual physical component in each matrix cell can differ across implementations and affects how fast, compact, and expensive the resulting memory is.
- Some memories are built that way using a flip-flop or latch for each cell while others use capacitors to store the binary value of the cell.

Types of memory

- Memory can be divided in **RAM** (Random-Access Memory) and **ROM** (Read-Only Memory).
- RAM is considered **volatile**. This means that it loses its data when the power is turned off. It provides quick read and write access and is the main memory of your computer.
  - It was named random, because any data could be accessed as easily as any other regardless of where it was located.
- ROM is **non-volatile** and it retains its data even if the power is turned off. Although it is quick to read, it can either not be written (historically was written only at manufacturing time) or is very slow.
  - Memory in cameras, game consoles, laser printers is often ROM.
  - Your computer ROM is responsible for storing the BIOS (Basic I/O System) that instructs how to start your computer and loads the operating system into the RAM.

Revisiting the simplified view of a computer



Arithmetic Logic Unit (ALU)

- Performs arithmetic and logical operations.
- Arithmetic operations like adding A and B (ripple-carry adder!), increment/decrement of A by 1, etc.
- Logical operations, like bitwise AND/OR of A and B or logical and arithmetic shifting.

