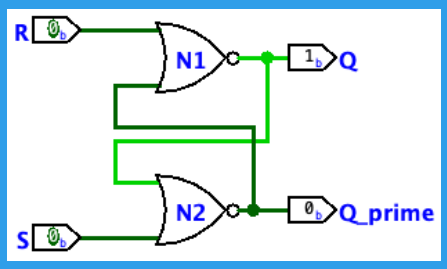
Computer Systems



Sequential Circuits

CS51 – Fall 2025

Today we will continue seeing circuits, but this time sequential circuits.

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**.

2

All circuits we have seen so far are 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 which not only depend on current input but have memory components that allow them to utilize prior inputs, too. And to establish some vocabulary regarding memory: Storing data into memory is called writing while retrieving data from memory is called reading.

SR-Latches

Let's see the simplest memory circuit we can build, an SR-latch.

SR-latch

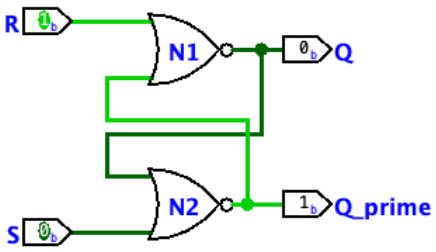
- We start by building an SR-latch which *latches onto* one bit of information.
- An SR-latch is a basic sequential circuit with two binary inputs, S and R, called set and reset, respectively, and two binary outputs, Q, and its complement Q'.
- When $S = 1$ and $\mathrm{R} = 0$, it **sets** Q to 1 (and its complement Q' to 0).
- When reset, $R = 1$ and $\mathrm{S} = 0$ 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'.

4

An SR-latch "latches" onto one bit of information. It is a basic sequential circuit with two binary inputs, by convention S and R, called set and reset, respectively, and two outputs, Q and its complement Q'. SR-latches follow these rules:
When $S = 1$ and $\mathrm{R} = 0$, it sets Q to 1 (and its complement Q' to 0).
When reset, $R = 1$ and $\mathrm{S} = 0$ 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'.



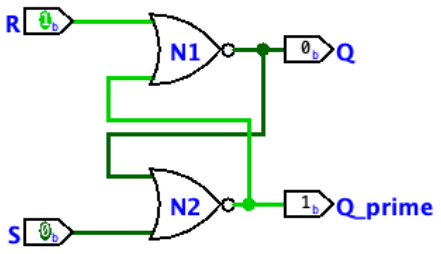
5

To build 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		



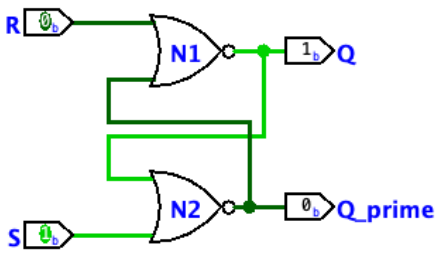
6

Case I will be R=1 and 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'.

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		



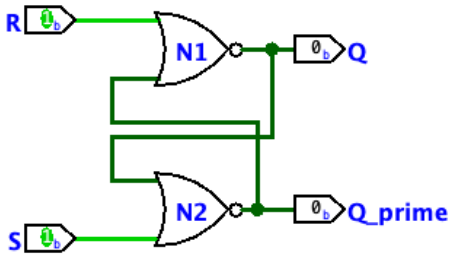
7

Case II is when R=0 and 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.

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		



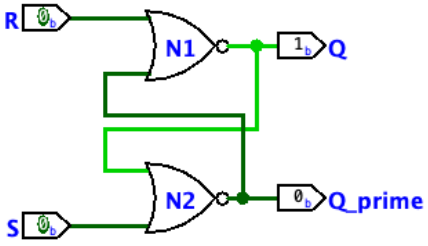
8

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.

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 1 for Q'. N_1 receives two 1 inputs through R and Q' 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



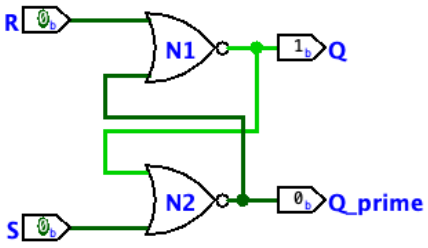
9

Last case, 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'. We truly have two options.
If 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.
If Q=1, N_2 produces 1 for Q'. N_1 receives two 1 inputs through R and Q' so it outputs 1 for Q as we had assumed

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



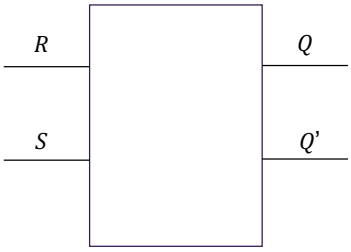
10

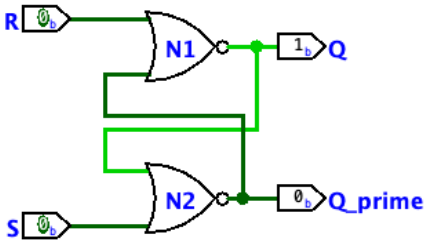
Putting all this together, suppose Q has some known prior value, which we will call $Q_{previous}$, before we enter Case IV. $Q_{previous}$ is either 0 or 1 and represents the state of the system.

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.

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:





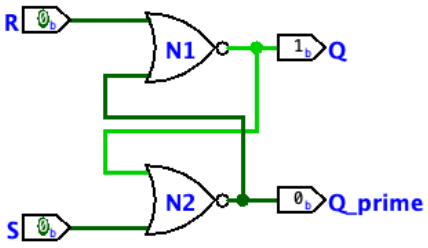
11

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

From SR-latch to D-latch

- SR-latches are easy to build but they can be awkward because they behave weirdly when both S and R are 1.
- They also conflate *what* and *when*. Setting one of the inputs to 1 determines not only what the state should be but also when it should change.
- We prefer circuits that separate the *when* and *what*.
- To solve this, we will move next to a circuit called D-latch that expands S-R latches to remember 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



12

SR-latches are easy to build but they can be awkward because they behave weirdly when both S and R are 1. They also conflate *what* and *when*. Setting one of the inputs to 1 determines not only what the state should be but also when it should change. We prefer circuits that separate the *when* and *what*.
To solve this, we will move next to a circuit called D-latch that expands S-R latches to remember one bit of information.

D-Latches

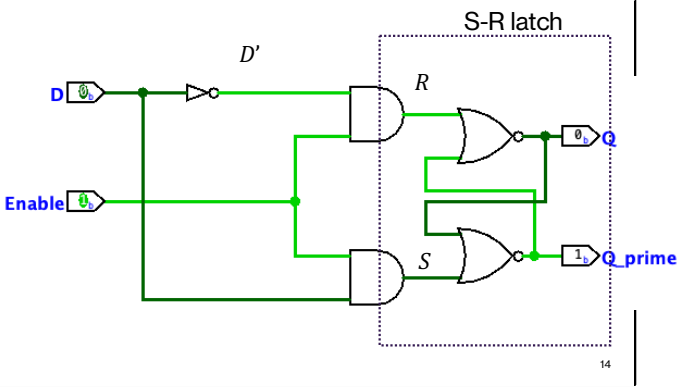
13

Let's look into D-latches

D-latch

- A D-latch has two binary inputs:
  - The data input, D, controls *what* the next state should be.
  - The Write Enable input, $Enable$, controls *when* the state can be changed by making memory available for writing in certain conditions.
- When $\text{Enable} = 1$, the latch is **transparent** and the data at D flows through to Q, writing to Q whatever value D has.
- When $\text{Enable} = 0$, the latch is **opaque** and blocks the data D from flowing to Q. Q remembers its previous value, $Q_{previous}$ until the next time that $\text{Enable} = 1$.
- We can build a D-latch off an S-R latch.

Input		Intermediate			Output	
Enable	D	D'	S	R	Q	Q'
0	0	1	0	0	$Q_previous$	$Q'_previous$
0	1	0	0	0	$Q_previous$	$Q'_previous$
1	0	1	0	1	0	1
1	1	0	1	0	1	0



D-latch has two binary inputs:

The data input, D, controls what the next state should be.

The Write Enable input, $Enable$, controls when the state can be changed by making memory available for writing in certain conditions.

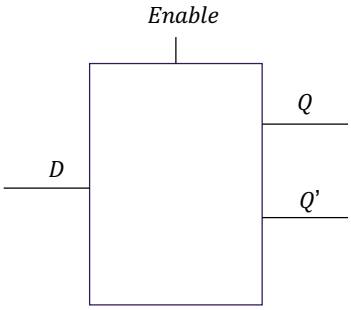
When Enable=1, the latch is transparent and the data at D flows through to Q, writing to Q whatever value D has.

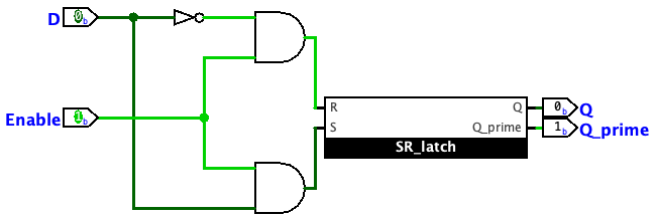
When Enable=0, the latch is opaque and blocks the data D from flowing to Q. Q remembers its previous value, $Q_previous$ until the next time that Enable=1.

We can build a D-latch off an S-R latch.

D-latch - Abstraction

- Using abstraction, we can see that the D-latch can be visualized with the S-R latch abstracted, as in the right.
- We also have a new symbol for D-latches:





15

Again, we can abstract the design of a D-latch, either by showing the SR_latch as a black box or by abstracting the whole design even further.

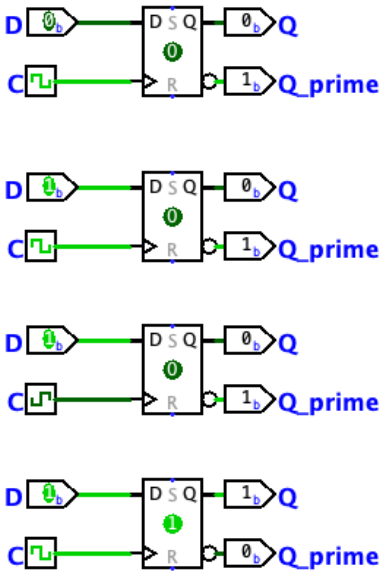
D-Flip Flops

16

D-latches are great but we can do even better.

From D-latches to D flip-flops

- The D-latch updates its state, Q, continuously when $\text{Enable} = 1$. This is known as **level-triggering**.
- It is useful to update the state Q only at a specific instant in time. Instead of a Write Enable line, we will use Clock which alternates between 0 and 1. Specifically, we will allow Q to be updated to whatever value D had only when the clock **rises** from 0 to 1. This is known as **edge-triggering**.
- D flip-flops are edge-triggered circuits and are composed from two back-to back D-latches controlled by a single clock.
- Whatever value was at D immediately before the clock rises from 0 to 1 gets copied to Q immediately after the clock rises from 0 to 1.
- All other times, Q remembers its old value $Q_{previous}$.



17

The D-latch updates its state, Q, continuously when Enable=1. This is known as level-triggering.

It is useful to update the state Q only at a specific instant in time. Instead of a Write Enable line, we will use Clock which alternates between 0 and 1. Specifically, we will allow Q to be updated to whatever value D had only when the clock rises from 0 to 1. This is known as edge-triggering.

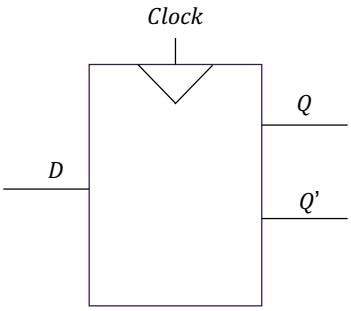
D flip-flops are edge-triggered circuits and are composed from two back-to back D-latches controlled by a single clock.

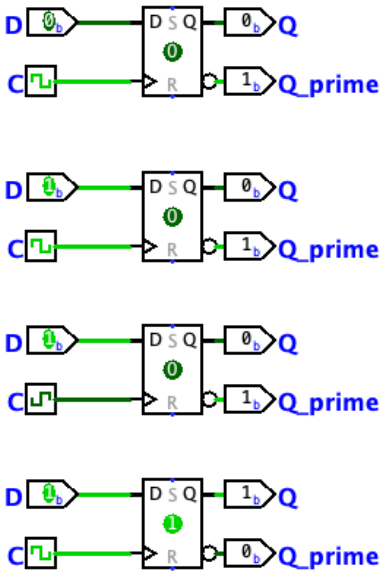
Whatever value was at D immediately before the clock rises from 0 to 1 gets copied to Q immediately after the clock rises from 0 to 1.

All other times, Q remembers its old value $Q_previous$.

D flip-flops - Abstraction

- The symbol for a D flip-flop is:





Again, we can abstract D flip flops.

Registers

We will next put together multiple flip flops to build registers.

From flip-flops to registers

- One flip-flop can store one bit of information.
- If we group together n flip-flops we have a **register**.
- The **width** of the register is the number of flip-flops n or the n bits we can store.
- Registers are the key building block of sequential circuits.

20

One flip-flop can store one bit of information.

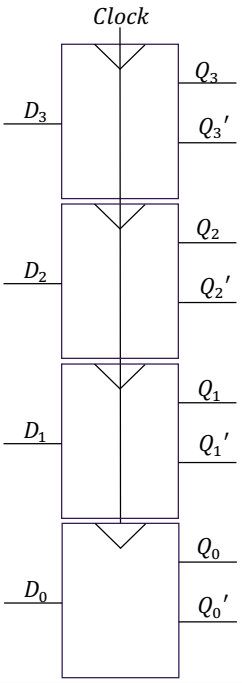
If we group together n flip-flops we have a register.

The width of the register is the number of flip-flops n or the n bits we can store.

Registers are the key building block of sequential circuits.

Registers

- To write to our register, we connect all of the flip flops on a single wire that connects to one $Clock$.
- On the right, you can see a schematic representation for a 4-bit register, with inputs $D_3 D_2 D_1 D_0$ and outputs $Q_2 Q_2 Q_1 Q_0$.
- When the $Clock$ rises to 1, the 4 bits $Q_2 Q_2 Q_1 Q_0$ are updated to whatever information is passed through the wires for the input data $D_3 D_2 D_1 D_0$.
- When the $Clock$ turns to 0 the 4 bits are saved in memory and are not altered until the $Clock$ rises again.



21

To write to our register, we connect all of the flip flops on a single wire that connects to one $Clock$.

On the right, you can see a schematic representation for a 4-bit register, with inputs D_3 D_2 D_1 D_0 and outputs Q_2 Q_2 Q_1 Q_0.

When the $Clock$ rises to 1, the 4 bits Q_2 Q_2 Q_1 Q_0 are updated to whatever information is passed through the wires for the input data D_3 D_2 D_1 D_0.

When the $Clock$ turns to 0 the 4 bits are saved in memory and are not altered until the $Clock$ rises again.

Registers

- 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.

22

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

Next, we will talk about memory in general.

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

24

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 as shown in this table.

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

25

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^Nx⟦ 2⟧^M.

This is a 2^3-word× 2^2 memory array, or simply an 8×4 array.

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 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

26

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 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.

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

27

Let's see what are the most common units of measuring the size of memory. We will often use the base 10 and base 2 definitions interchangeably.

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.

28

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?

29

Your turn

ANSWER - Calculating the size of memory

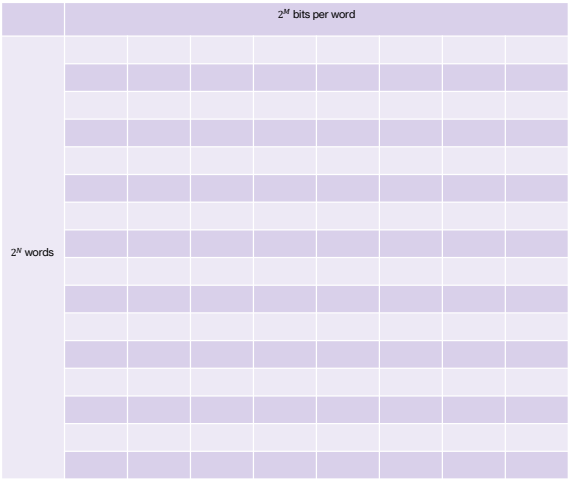
- Assume you have a memory with 2×2^{20} words, each word being 32 bits. What is the memory's size in bytes?
- Size = $2 \times 2^{20} \times 32$ bits = $2 \times 2^{20} \times 4$ bytes ≅ 8 MB.

30

Did you get 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.



31

Let's think now 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.

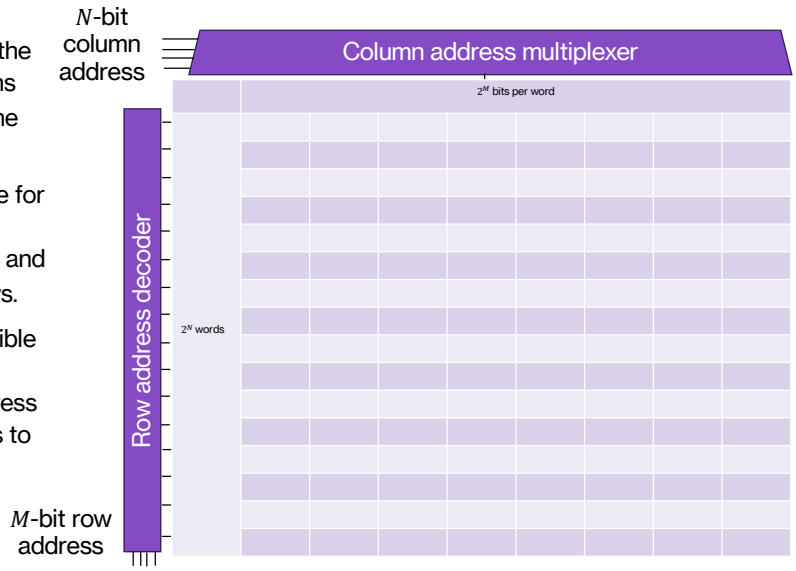
How many bits would we need to address a particular column?

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.

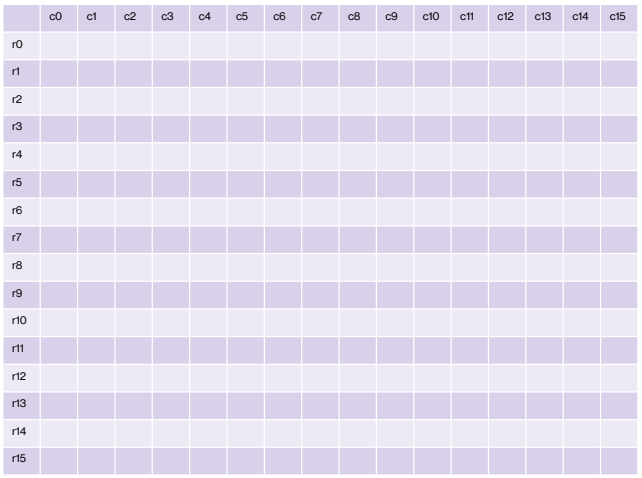


32

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.

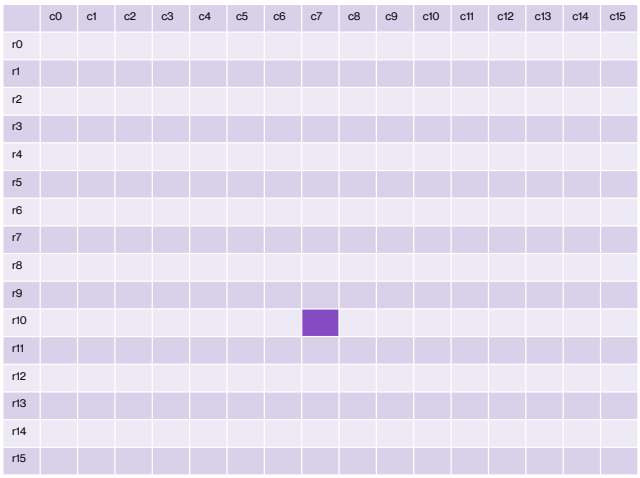


33

Here is an example with a 16x16 memory array.

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.

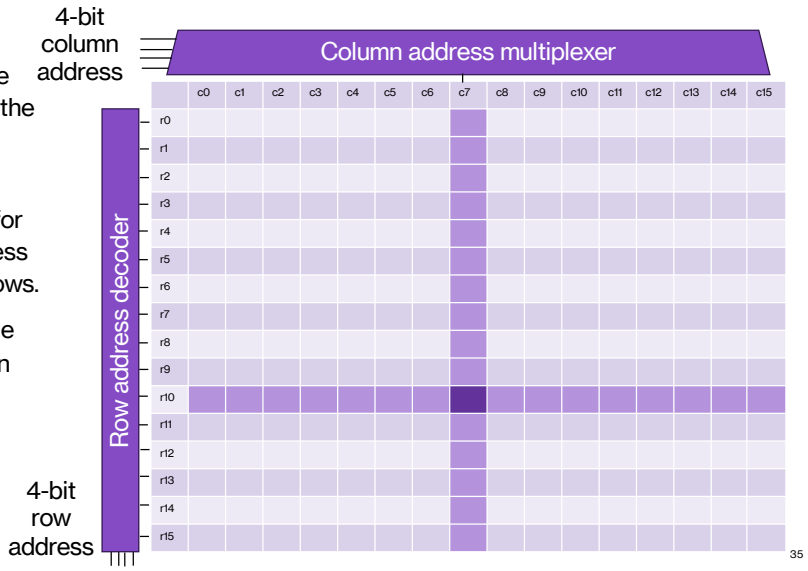


34

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.



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.

36

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.

37

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.

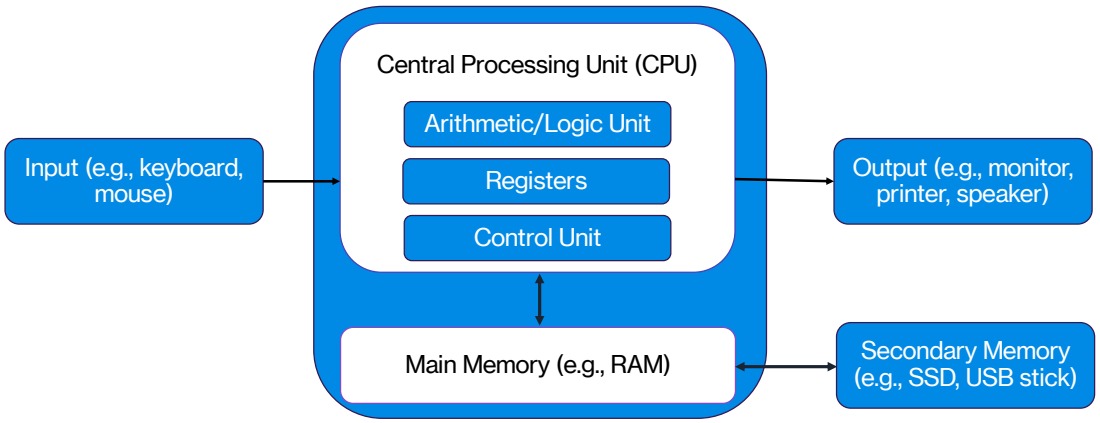
It was named random, because any data could be accessed as easily as any other regardless of where it was located.

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



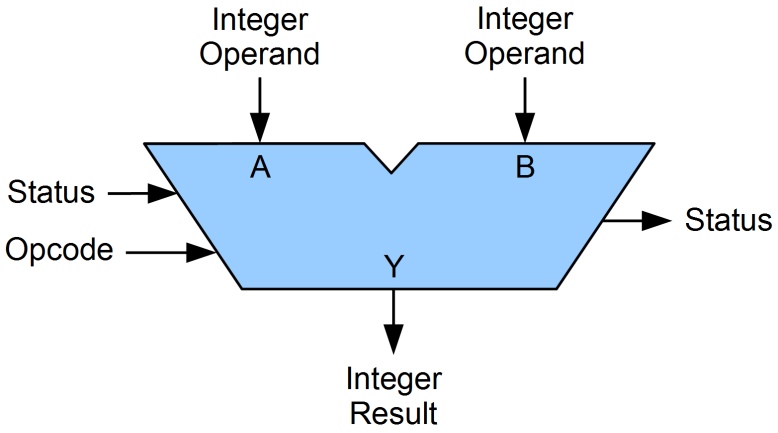
38

A simplified view of a computer would show that it comprises mostly of two components. A central processing unit, i.e. the CPU, and memory (also known as primary or main memory). You can think of the CPU as the brains of your computer. It consists of three main components. An arithmetic/logic unit, known as the ALU, that performs arithmetic and logic operations, processor registers that supply information to the ALU and store the results that the ALU outputs, and a control unit that orchestrates the fetching of data from the main memory and which coordinates the ALU and the registers. The main memory stores current data, programs, and instructions that can be accessed directly by the CPU. Any data that should not be changed is stored in a special type of main memory called ROM (read-only memory). For example, the programs that your operating system needs when it boots are stored in ROM. In contrast, RAM (random-access memory) is like our computer's short-term memory for data and programs that need to be read and written but is volatile, that is its information is wiped if power is removed. Secondary memory stores programs and data permanently, without having to worry about what happens when we power off our computer. Secondary memory comes mostly in the form of solid-state drives (SSDs) these days. They are smaller, faster, and less noisy than the older hard-disk drives (HDDs). Other examples of secondary memory are usb

drives, memory cards, CDs/DVDs. All this exchange of information can be supplemented by input that the user provides through their keyboard, mouse, etc, and it can be directed to output devices such as the monitor, printers, speakers, etc.

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.



39

If we zoom into the CPU, we would see the ALU which 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.