

# COMSM1302

## Overview of Computer Architecture

### Lecture 5 – Storage



# In this lecture

## Foundations

- Data representation, logic.

## Building blocks

- Transistors, transistor based logic, simple devices, **storage**.

## Modules

- Hex modules, memory, simple controller and processor.

## Programming

- Assembly, assembler, language, compilation phases, boot-strapping.

## Bigger systems

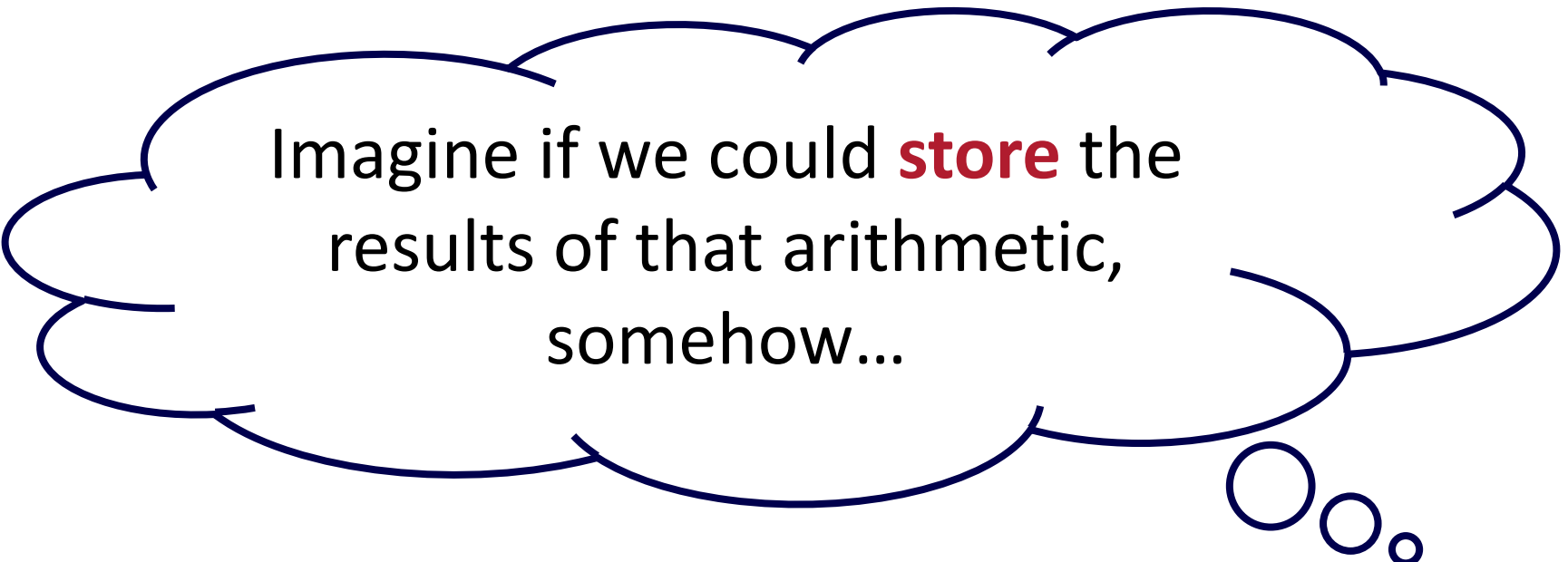
- ARM & Thumb, I/O, protecting shared systems, memory hierarchy, multi-processors, networks.

## Wrap-up

- More examples, historical computers, contemporary systems.

# Previous lecture

- We can do **basic arithmetic** with a **bunch of NAND gates**!



Imagine if we could **store** the results of that arithmetic, somehow...

# Combinatorial vs. sequential logic

- So far, everything we've done is **combinatorial** logic.
  - Input signals are combined in various ways to produce output signals.
  - We build more complex systems by **connecting blocks together**.
- If we **change inputs**, the **output changes** shortly afterwards.
  - Signals take some (very small) time to propagate.

# Combinatorial vs. sequential logic

- What if we wanted to perform the following arithmetic:

$$24 + 15 + 100$$

- But we can only add two numbers together.
- Create a sequence:

$$24 + 15 = X$$

$$X + 100 = Y$$

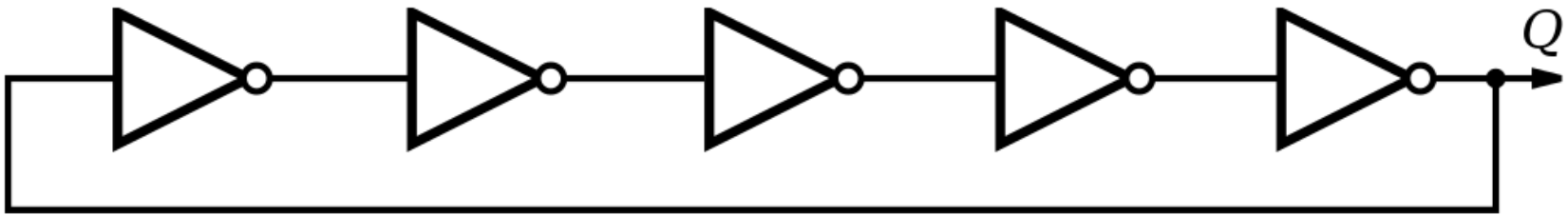
- We want to take the **output** of our **first addition** and make it one of the **inputs** to our **second addition**.

**It's not *quite* that simple.**

# 🔥 Ring Oscillator



What does the below device do?



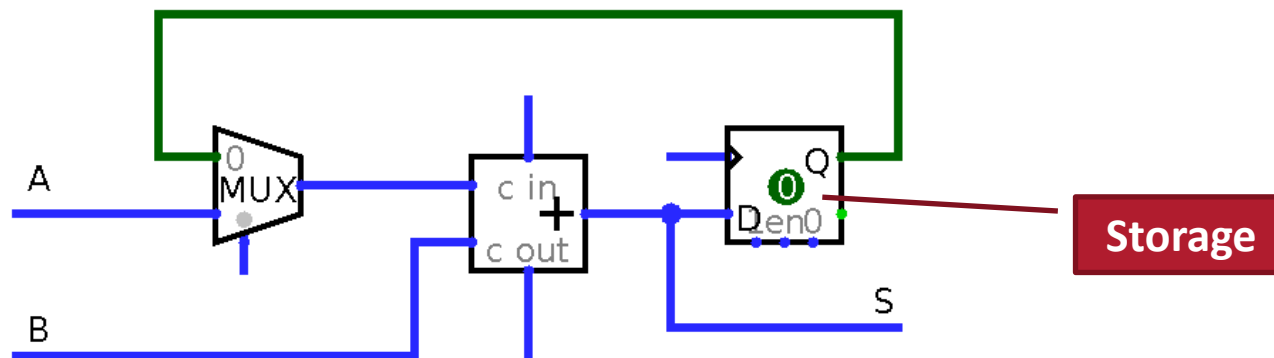
# Ring oscillator

- The output oscillates between 1 and 0.
  - How fast?
- $$F_{osc} = \frac{1}{N \times 2 D_{inv}}$$
- N is the number of inverters.
- $D_{inv}$  – the inverter delay, is determined by:
    - Wire length
    - Device size
    - Voltage
    - Temperature
    - Material
    - ...

# 🔥 Combinatorial vs. sequential logic

- To enforce a sequence reliably, we can:
  - Store result values
  - Control when values are stored
- This allows us to build a sequential system, combining **storage** and **combinatorial logic**.

**Below: A *prototype* we'll explain, develop and complete!**

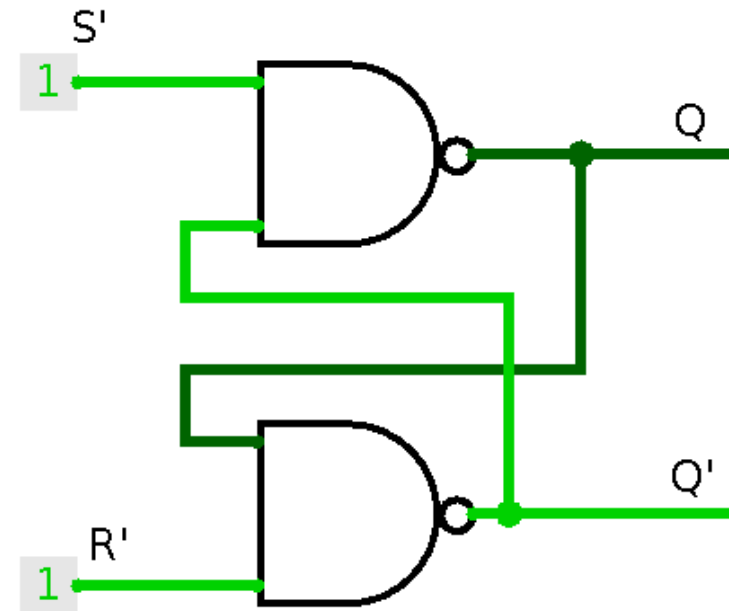




# 🔥 SR-latch

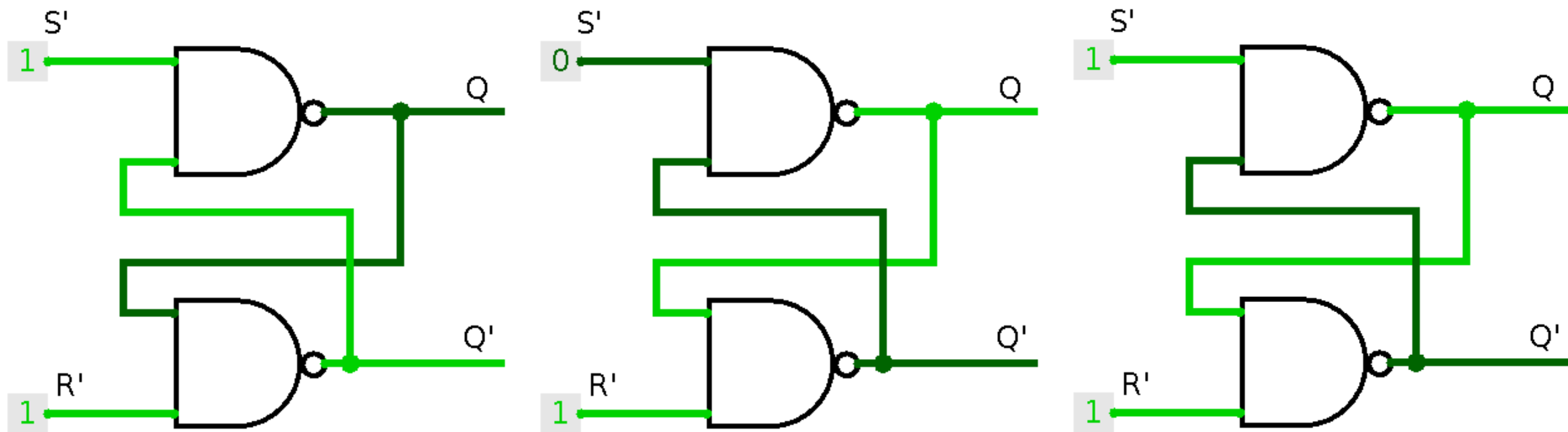


- Two NAND gates
- Input signals are **active-low**.
  - 1 = False, 0 = True
  - Denoted with top bar or tick.
- Set the latch
  - $S'$
- Reset the latch
  - $R'$



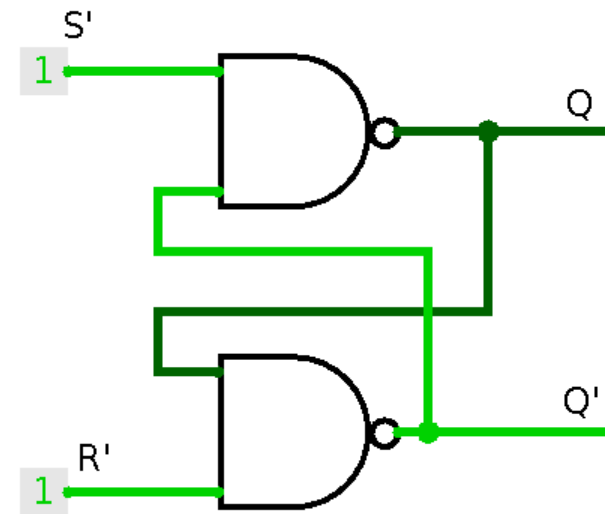
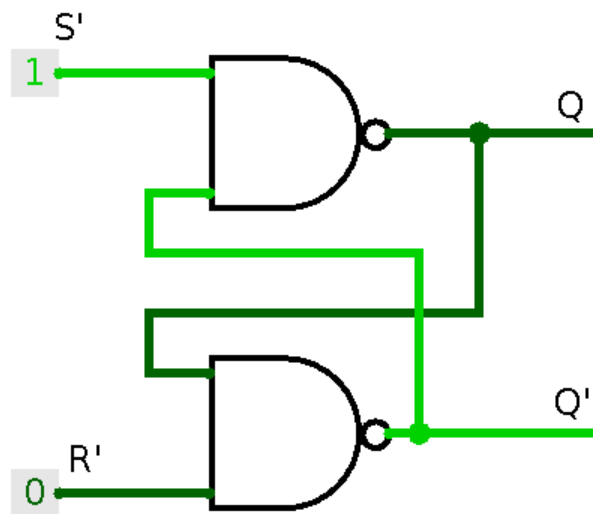
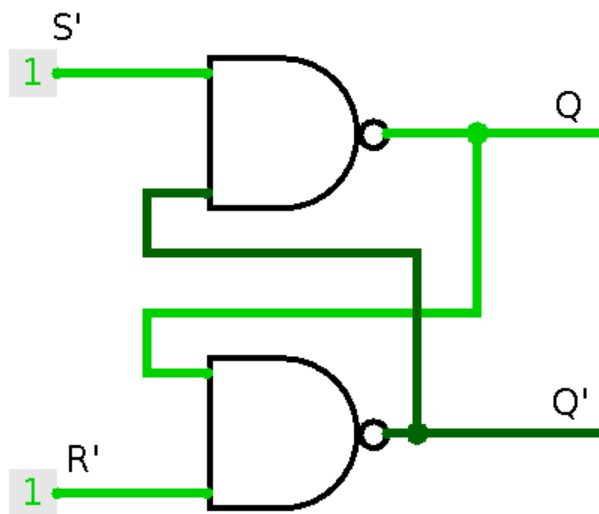
# 🔥 Setting the latch

- When  $S'$  transitions to low,  $Q$  is **set high** (and  $Q'$  its inverse).
- Upon  $S'$  returning to high...  $Q$  retains the same high value.



# 🔥 Resetting the latch

- When  $R'$  transitions to low,  $Q$  is **reset low**.
- Upon  $R'$  returning to high...  $Q$  retains the same low value.



# SR inputs



S'	R'	Q
0	0	Not allowed
0	1	1
1	0	0
1	1	Hold

- There are **three valid** combinations of  $S'$  and  $R'$  for the SR NAND latch.
- Both  $S'$  and  $R'$  should **not be active** (low) together.

Why?

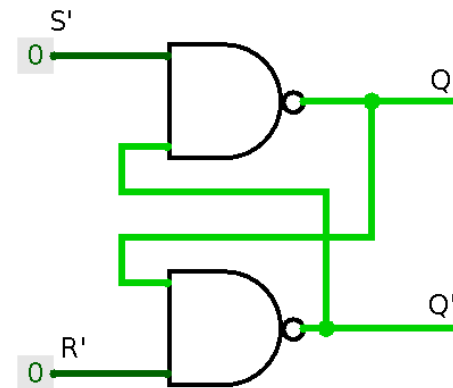
# SR inputs



S'	R'	Q	Q'
0	0	1	1
0	1	1	0
1	0	0	1
1	1	Q_prev	Q'_prev

- If both  $S'$  and  $R'$  are active, the output **doesn't make any sense**.

$$S' = 0, R' = 0, \mathbf{Q = Q'}$$



# Different latch types

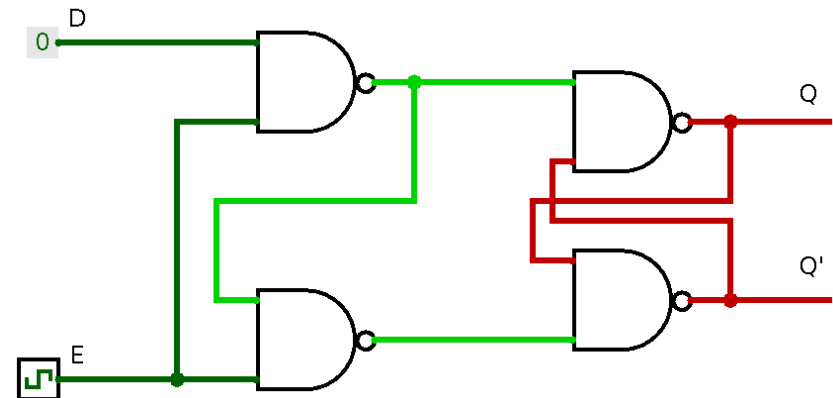
- We've seen SR (Set-Reset) latch
- Other latches:
  - **D latch**
  - JK latch
  - Earle latch

# D-latch

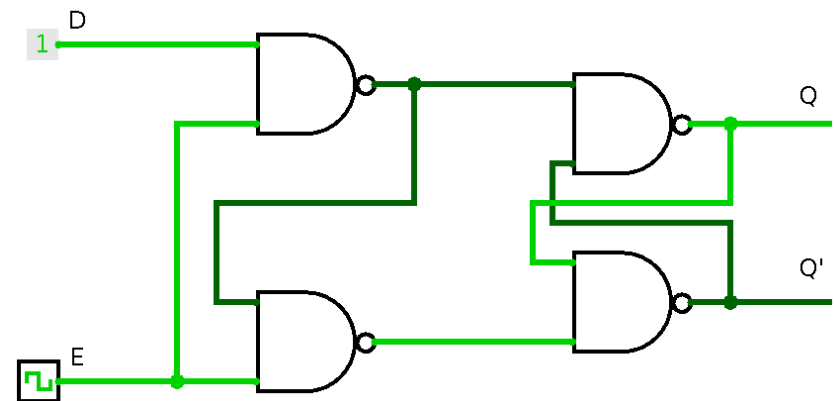
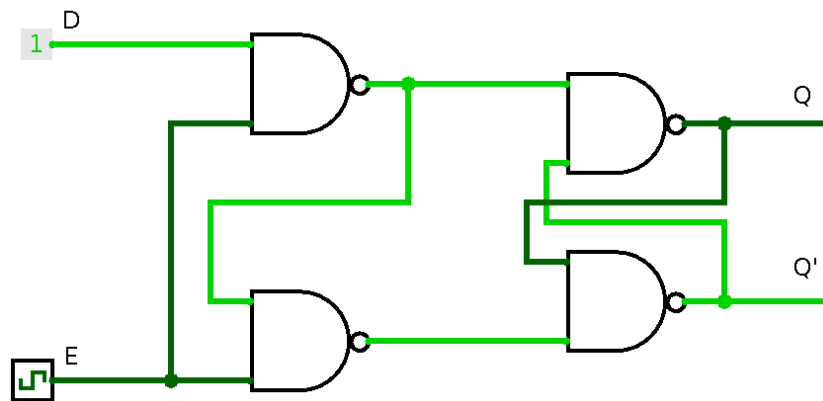
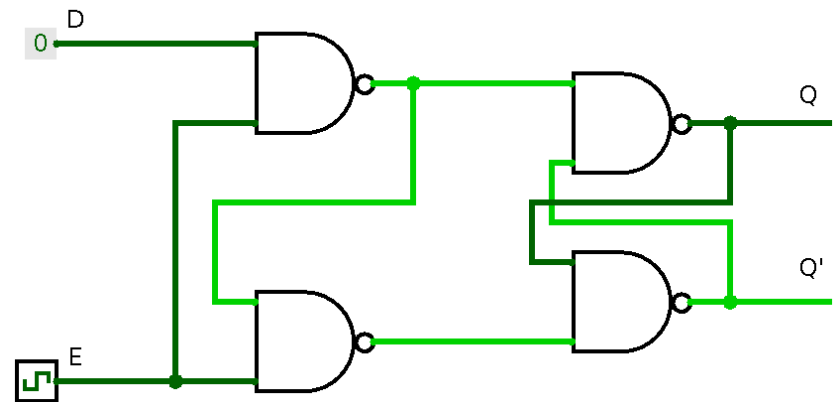
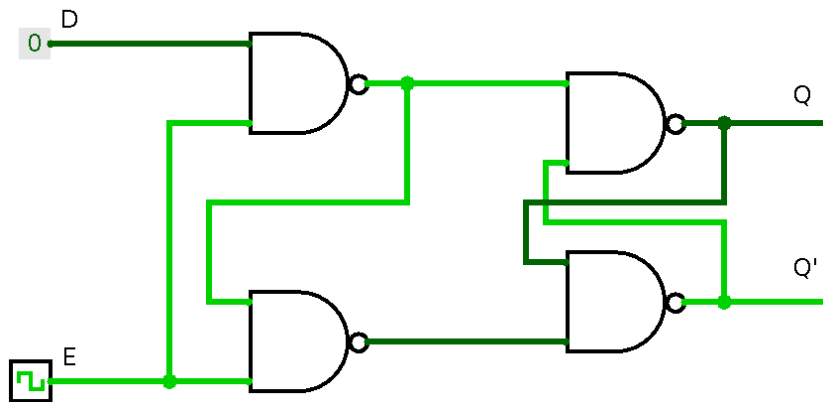


- Instead of separate set/reset inputs, one input, **D**, specifies the **data** value, and a second, **E**, **enables** propagation.
- If E is low, the previous output values are retained.
- No forbidden inputs.
  - At what cost?

D	E	Q	Q'
0	0	Q_prev	Q'_prev
0	1	0	1
1	0	Q_prev	Q'_prev
1	1	1	0

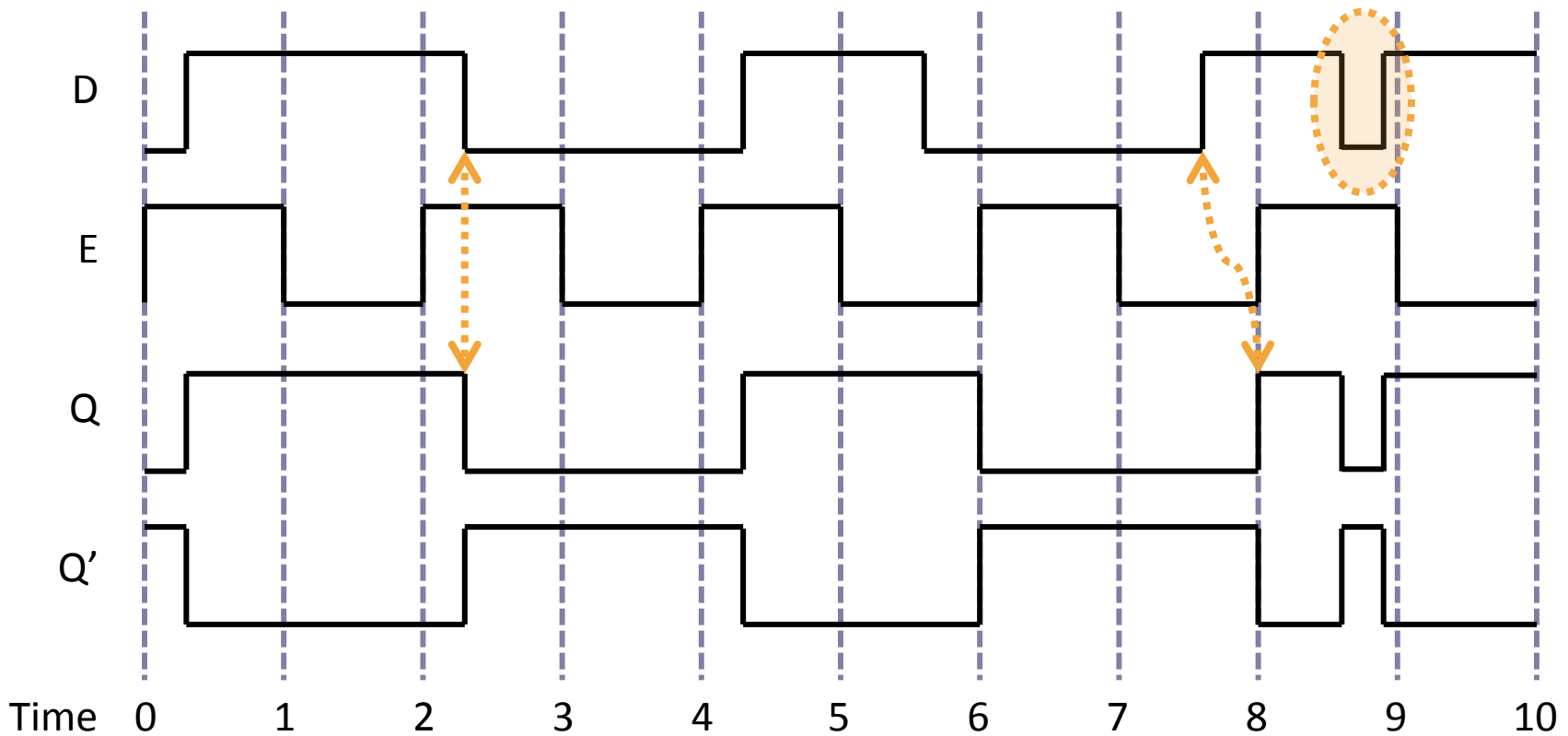


# 🔥 D-latch simulation



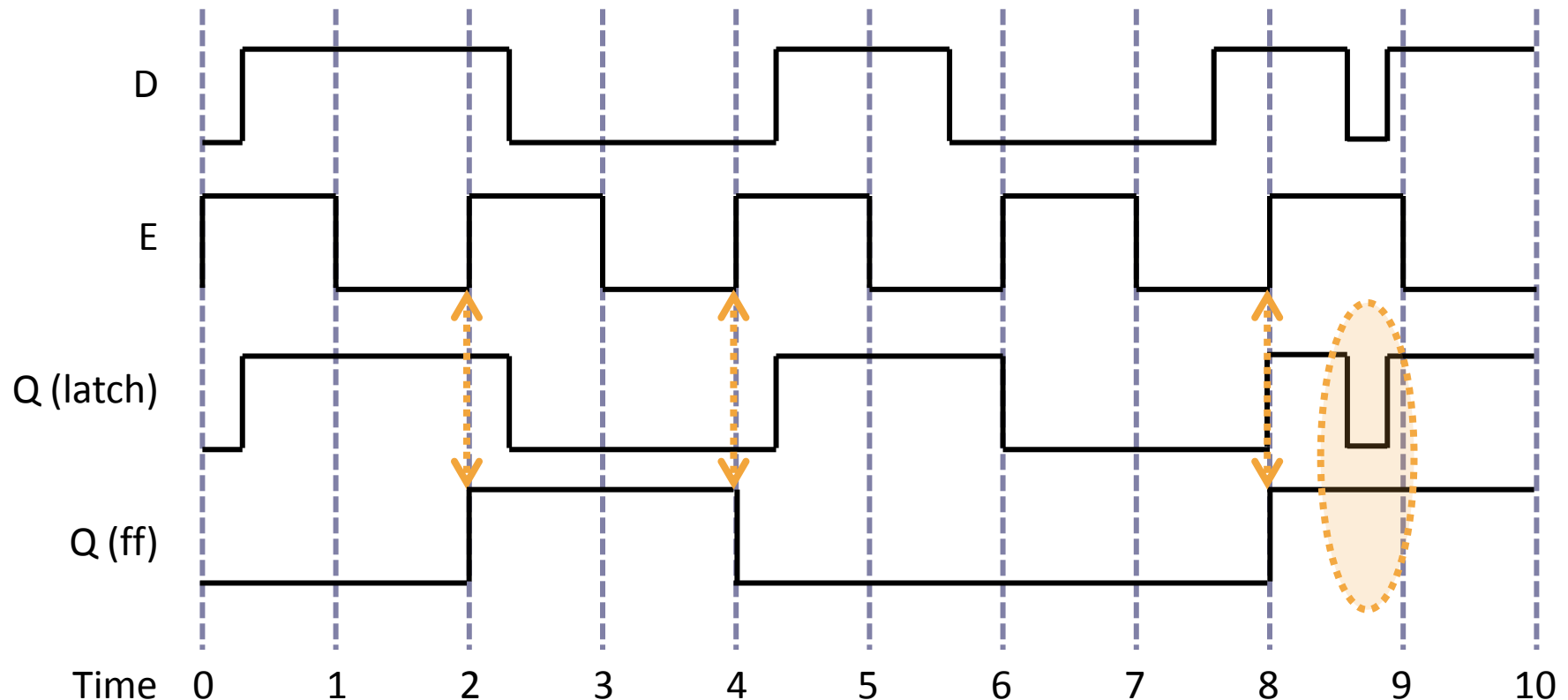


# Waveform

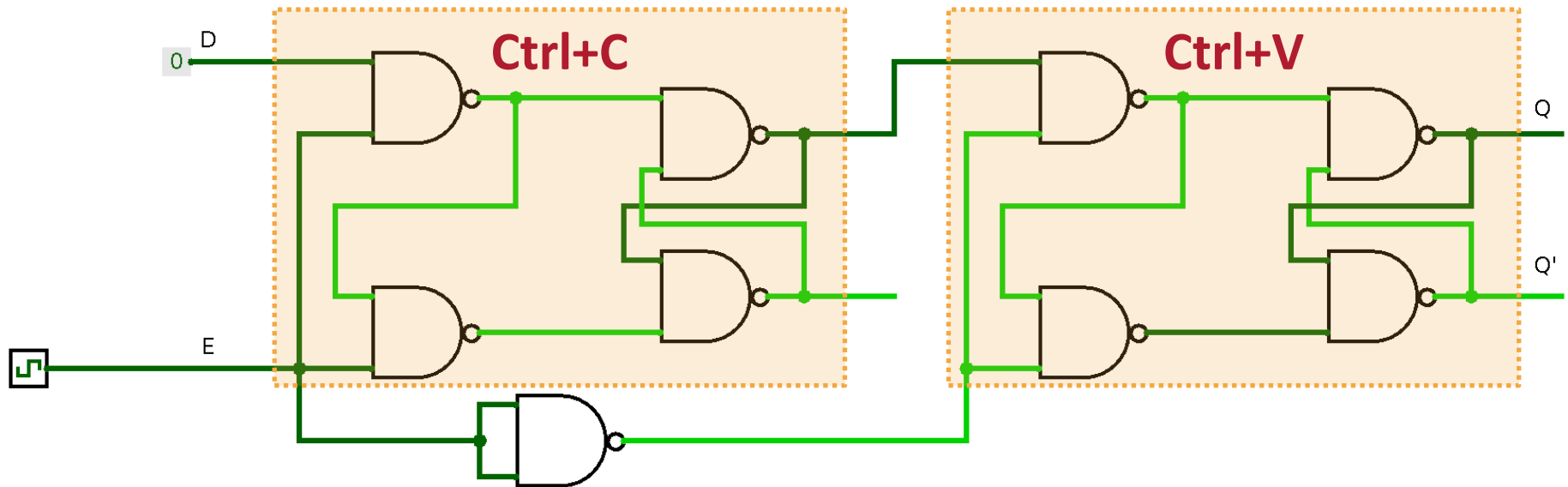


# 🔥 Level vs. edge triggering

- Latches are level sensitive.
- We can also build edge sensitive devices - Flip-flops



# 🔥 D-type flip-flop

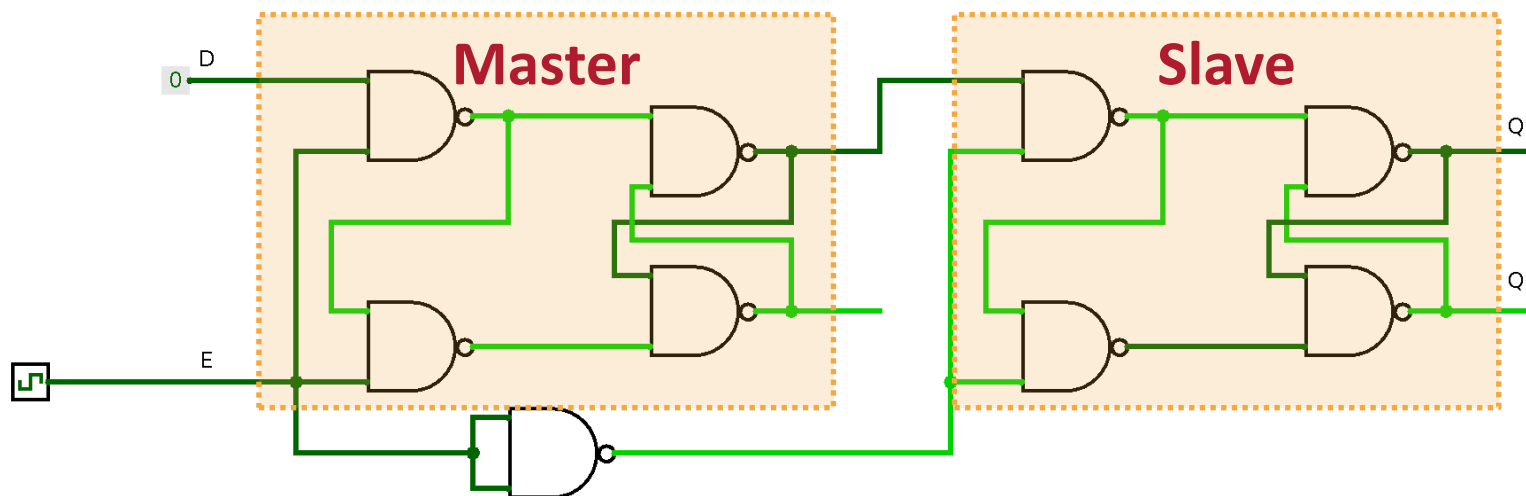


- Two D-type latches.
- Inverted enable signal to the second latch.

**Demo in Logisim**

# 🔥 D-type flip-flop

- The example shown is a master-slave D-type flip-flop.
- The master latch is enabled on E high
  - The slave is disabled, its output is **held**.
- The slave latch is enabled on E low
  - The master is disabled, its output is held.



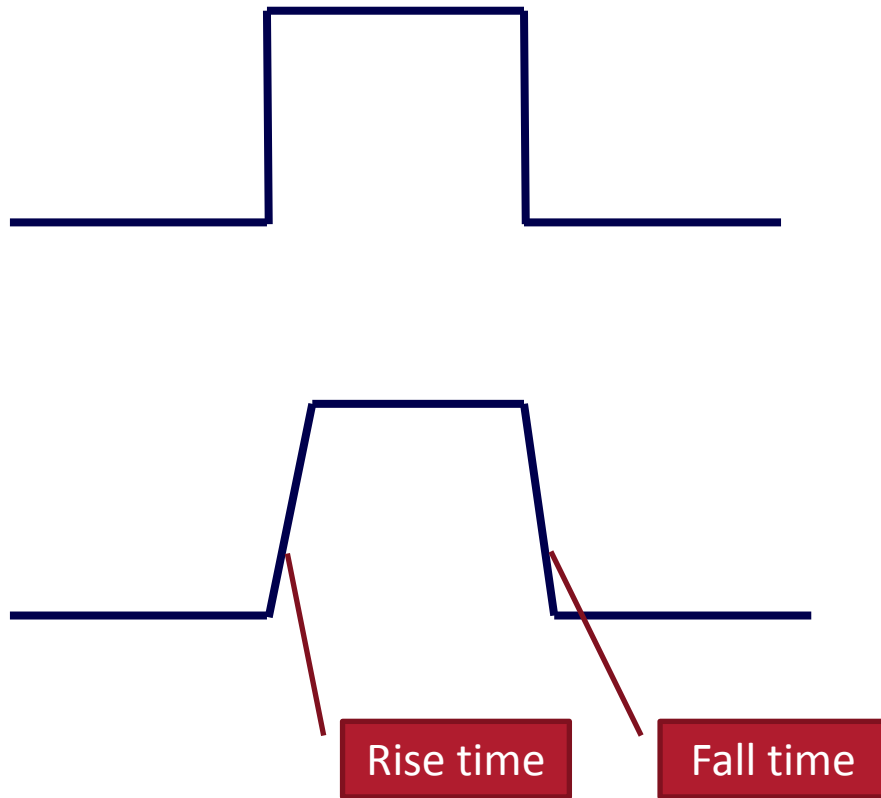
# D-type flip-flop transition



D	E >	Q
0	Rising	Q_prev
0	Falling	0
1	Rising	Q_prev
1	Falling	1

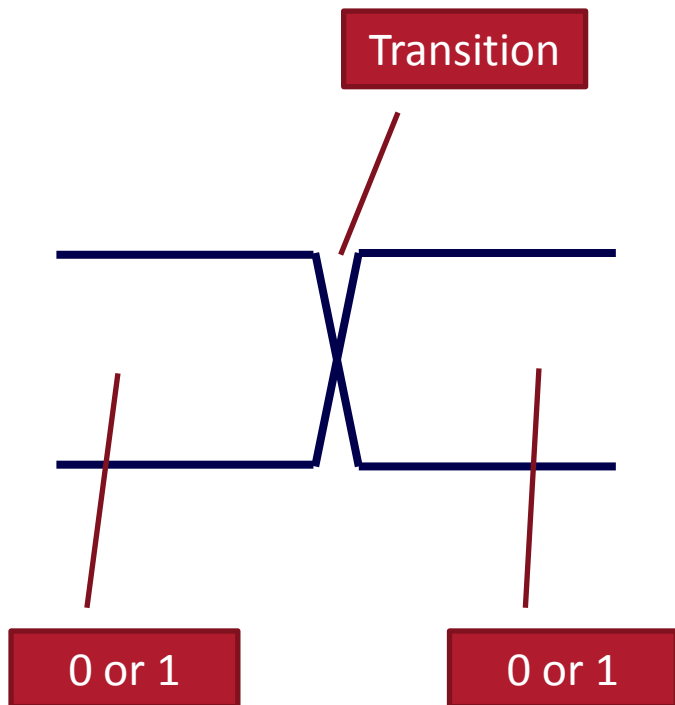
- We've built a **falling-edge** or **negative-edge triggered** flip-flop.
- How would we make it **positive-edge**?
  - How many gates have we used?
- There's **more than one way** to make a D-type FF.
  - Hint: Three SR NAND latches.

# Timing



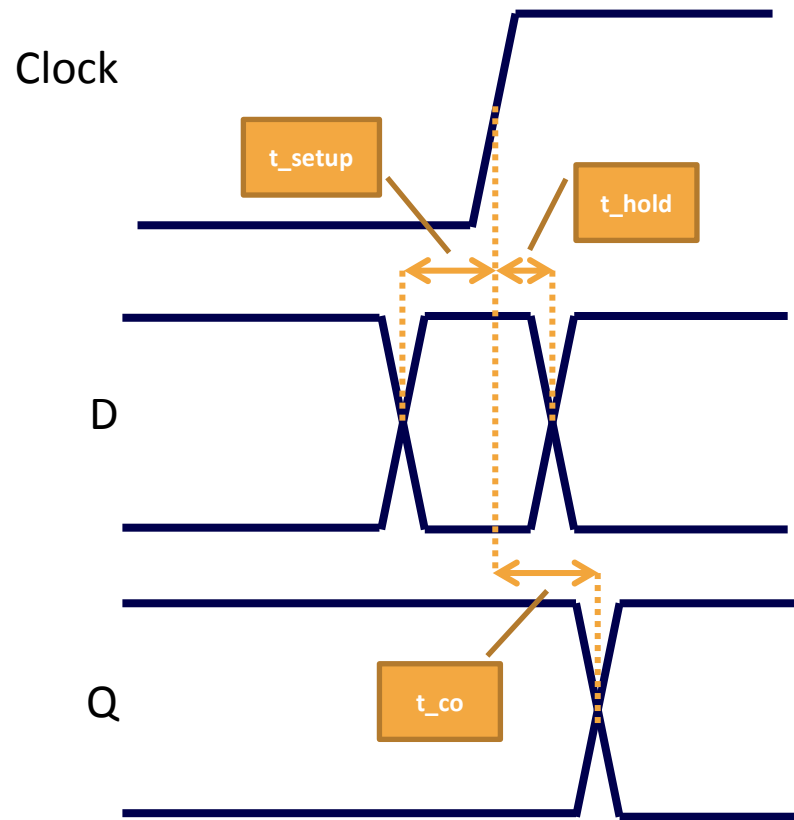
- We imagine signal transitions as instant.
- But they're not!
  - Wires and transistors have capacitances. They have a charge/discharge time.
  - Remember the Ring Oscillator.
- This results in rise & fall times.

# Timing



- When a signal changes there's a period of time where **we don't know its value**.
- So, if events (i.e. clock edges) happen at the wrong time **unexpected behaviour can occur**.

# Timing

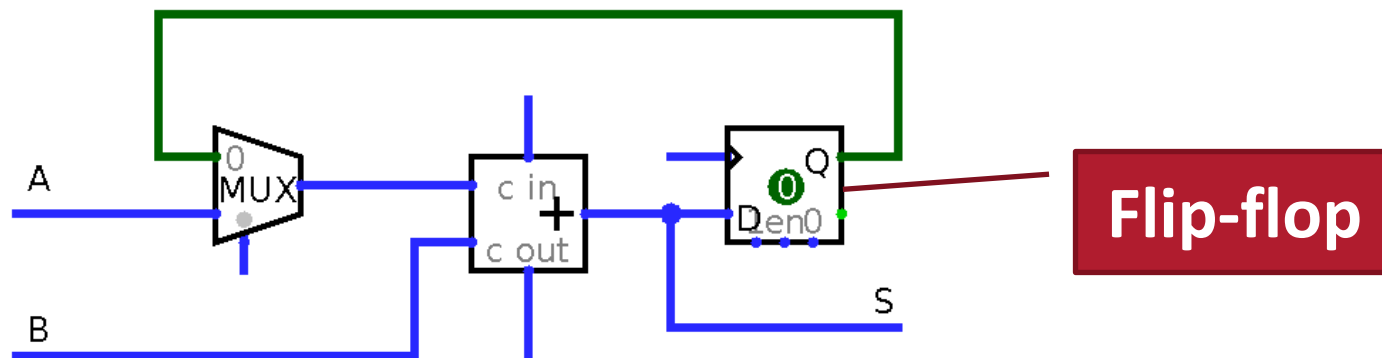


- We specify **timing constraints** to try to avoid these problems.
- Setup time ( $t_{\text{setup}}$ )
  - Signal should be settled for this amount of time **before** event.
- Hold time ( $t_{\text{hold}}$ )
  - Signal should remain settled for this long **after** the event.
- Clock-to-output ( $t_{\text{co}}$ )
  - Time between event and **output changing**.



# 🔥 Combinatorial vs. sequential logic

- To enforce a sequence reliably, we can:
  - Store result values
  - Control when values are stored
- This allows us to build a sequential system, combining **storage** and **combinatorial logic**.



# What have we got now?

- We have enough devices to perform **sequences of operations**.
  - Results can be **stored and reused** in the next operation.
  - We have to setup our inputs **by hand** at each stage in the sequence.
  - We have a **clock** that we have to operate **by hand**.
- We have flip-flops – single units of **memory**.
  - Soon we will have more complex memories.
  - And ways of controlling the system **without** manual (by hand) intervention.

# What next?



- Introduction to the hex modules
  - Building on the theory we've done so far, with a set of real devices.
- Brings us closer to **building our own processor**.

# Next lecture

## Foundations

- Data representation, logic.

## Building blocks

- Transistors, transistor based logic, simple devices, storage.

## Modules

- **Hex modules**, memory, simple controller and processor.

## Programming

- Assembly, assembler, language, compilation phases, boot-strapping.

## Bigger systems

- ARM & Thumb, I/O, protecting shared systems, memory hierarchy, multi-processors, networks.

## Wrap-up

- More examples, historical computers, contemporary systems.