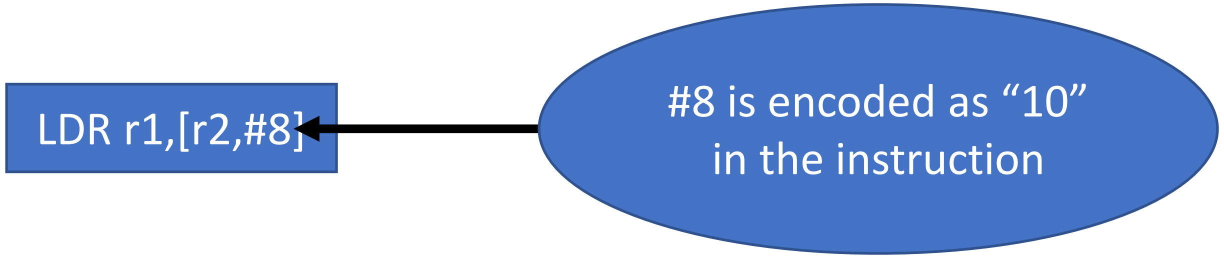
Module 2

Alignment, multiplexing and debouncing

Tim Rogers 2022

Remember alignment?

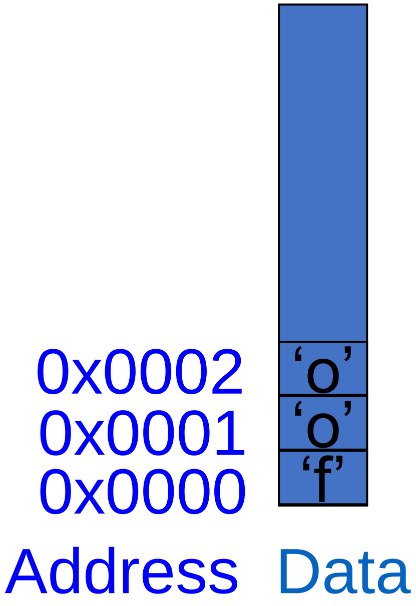
- LDR Rt, [Rn, #imm]
  - #imm must be divisible by 4.
  - Contents of Rn must be divisible by 4.
  - Rn + #imm must be divisible by 4.
    - Necessary for ***alignment***.
  - imm is encoded into instruction as a 5-bit value that can represent multiples of 4 from 0 – 124.



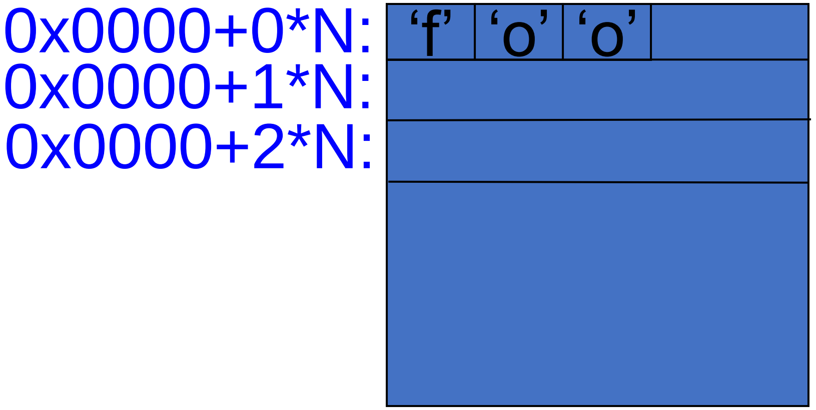
Why are machines like this?

- Blame memory!

Programmer's View:

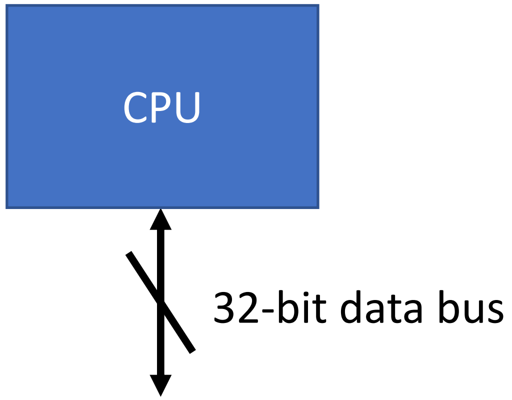


Hardware:

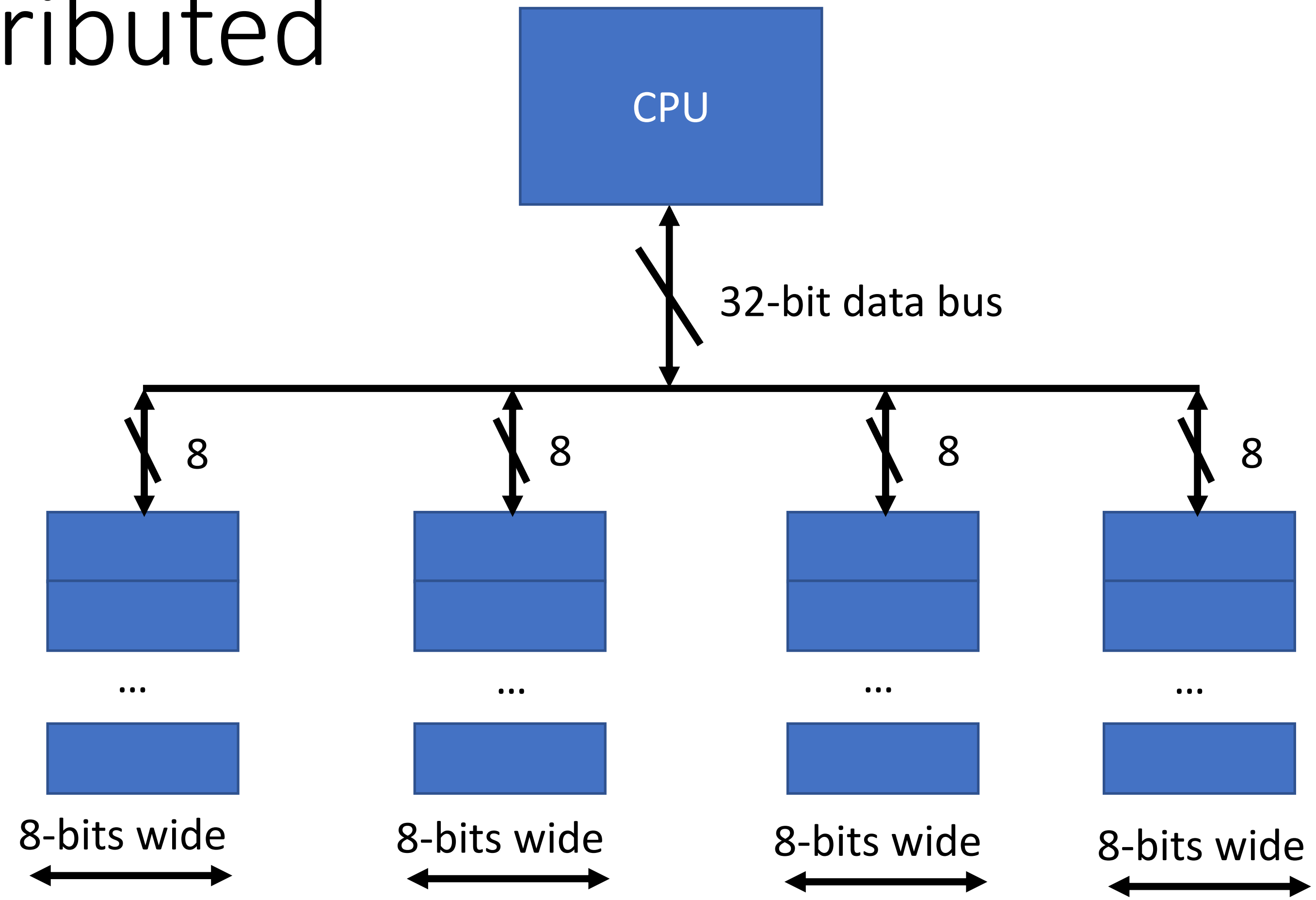


(N depends upon HW)

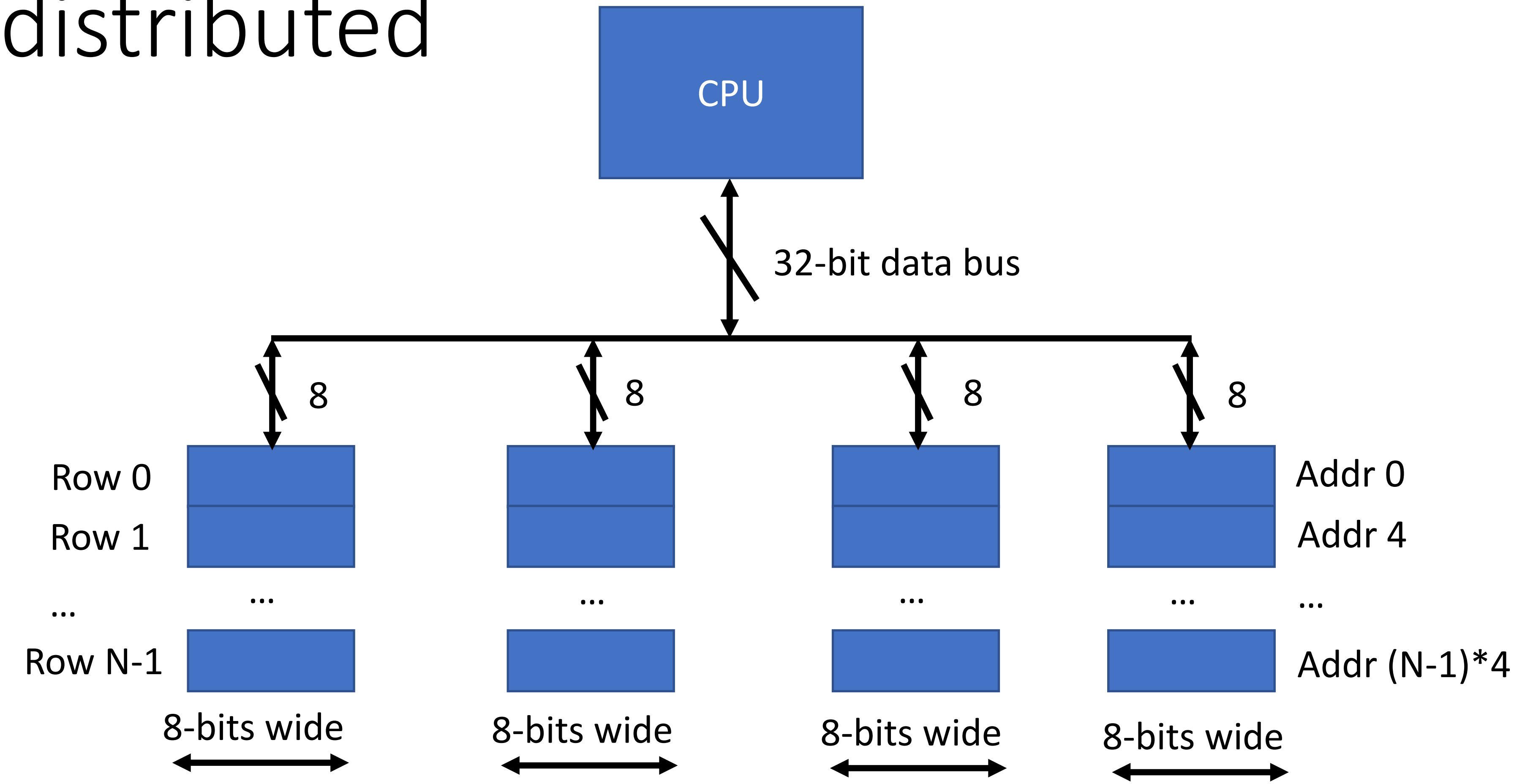
Real memories are rectangular and distributed



Real memories are rectangular and distributed

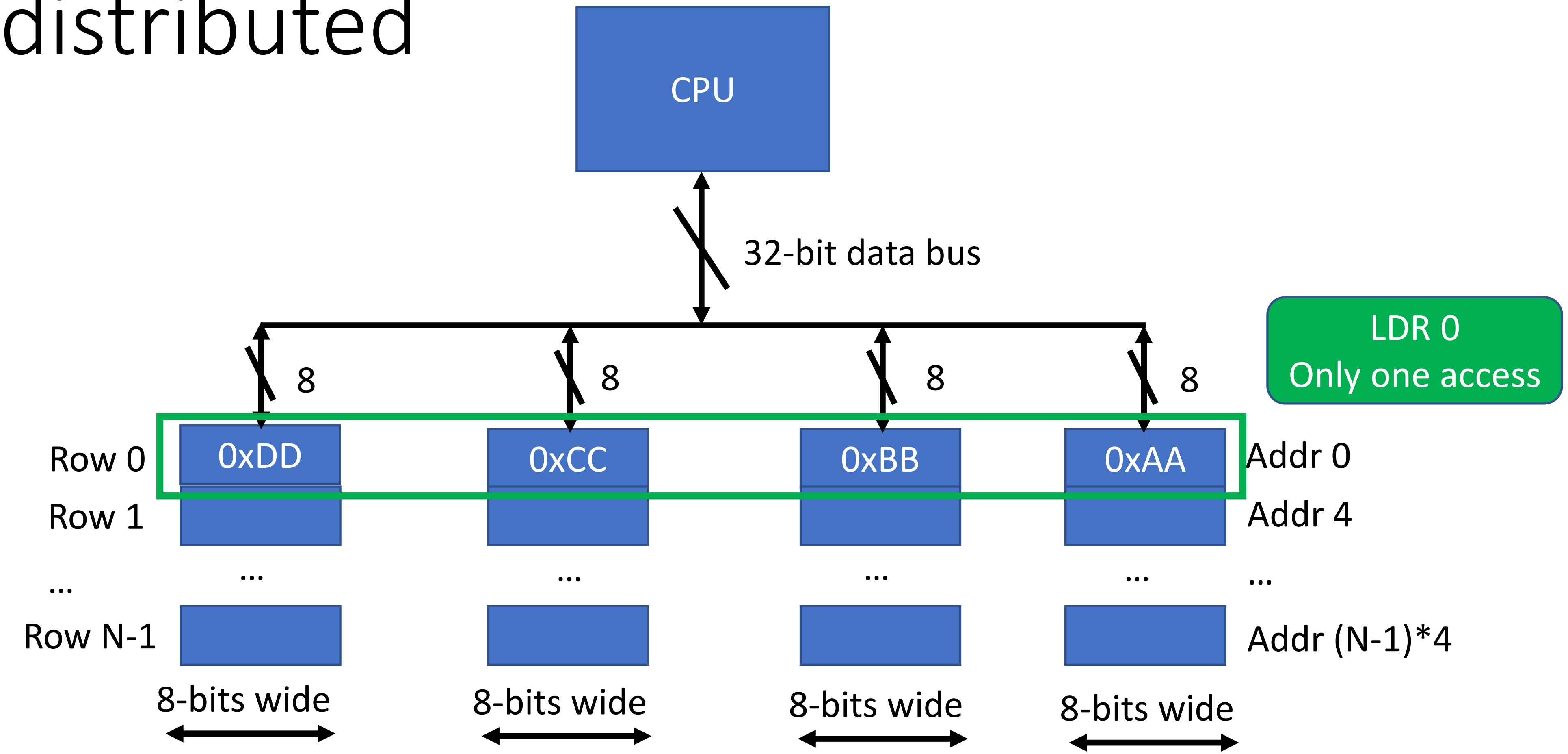


Real memories are rectangular and distributed



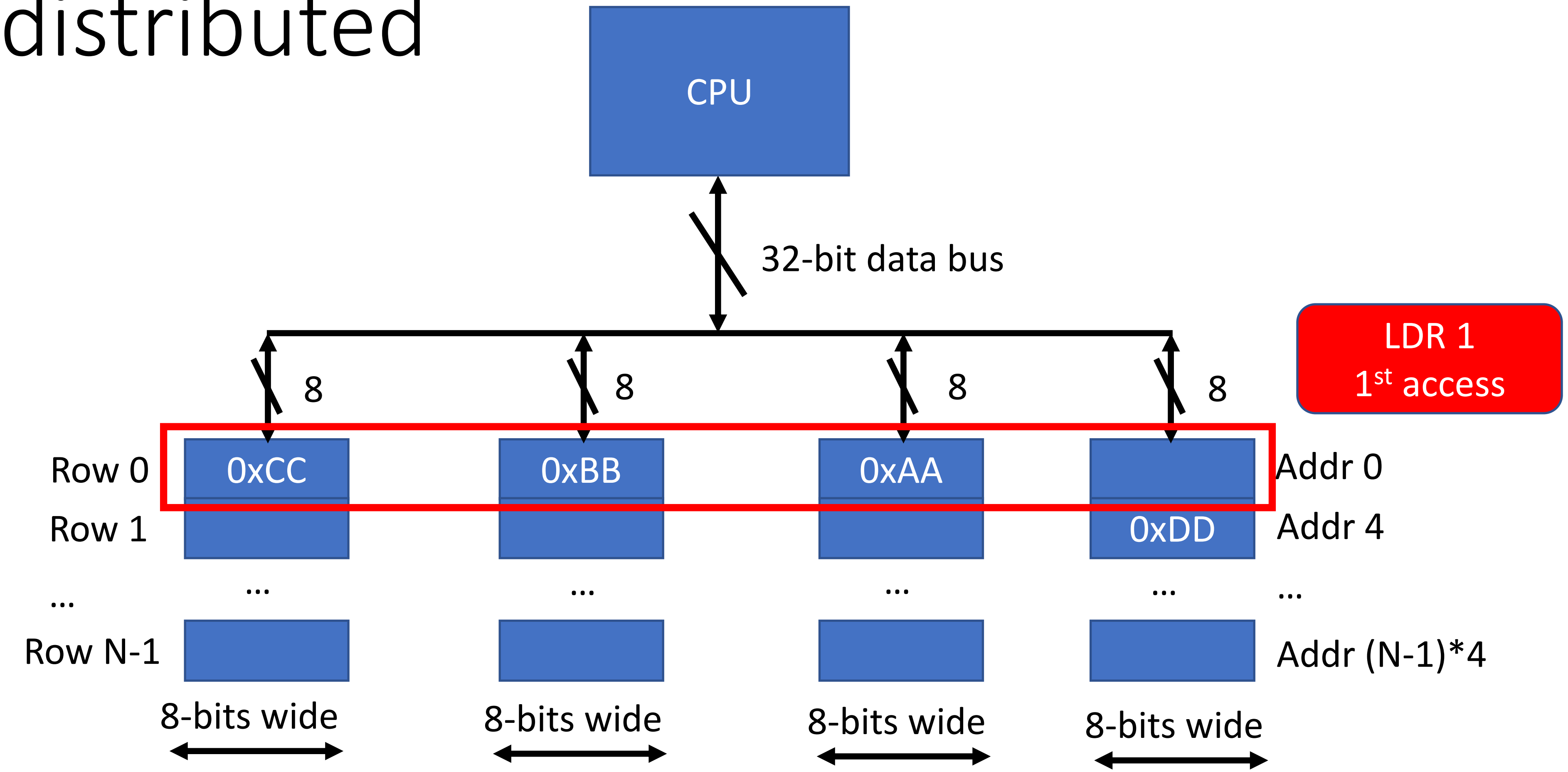
- **Can only read from one row at a time**

Real memories are rectangular and distributed



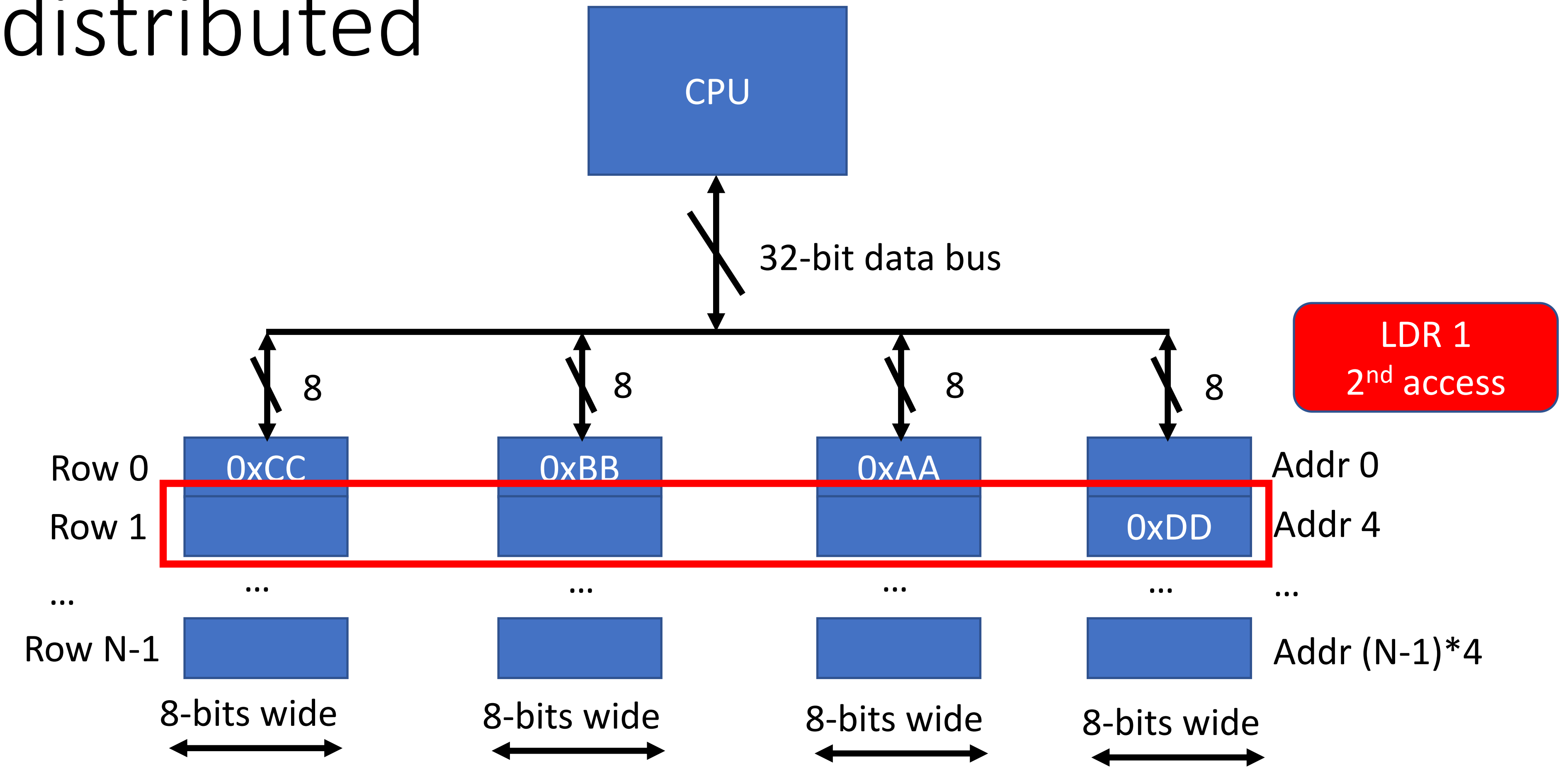
- **Can only read from one row at a time**

Real memories are rectangular and distributed



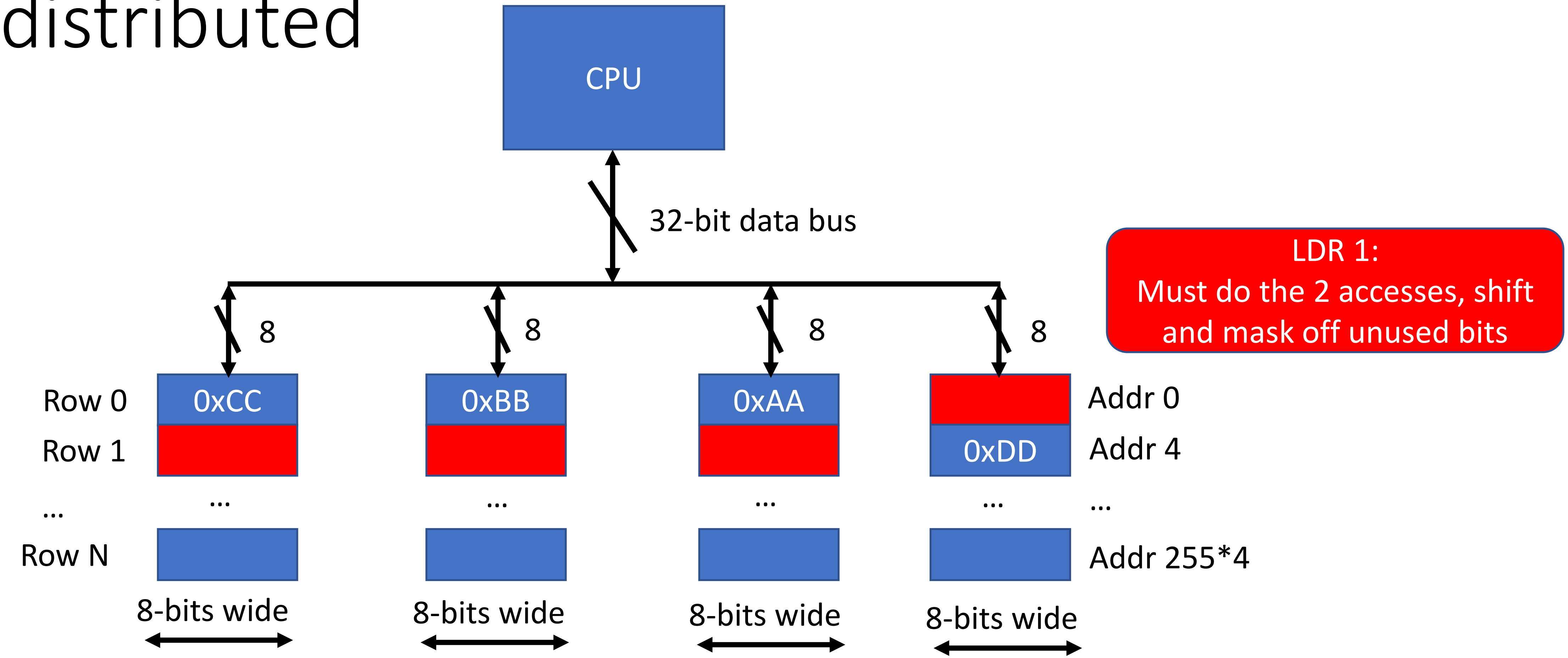
- **Can only read from one row at a time**

Real memories are rectangular and distributed



- **Can only read from one row at a time**

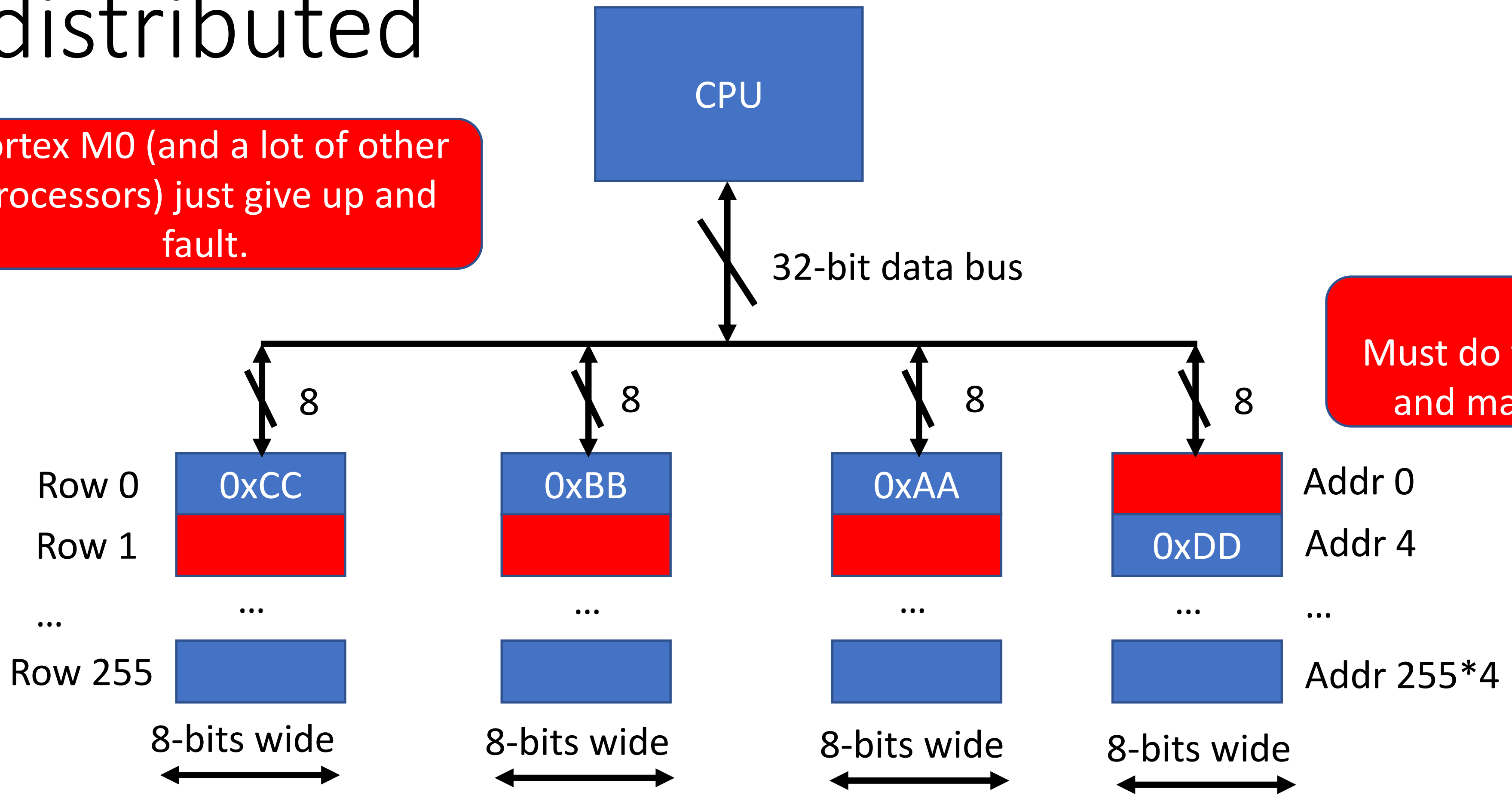
Real memories are rectangular and distributed



- **This is doable, but slower and requires specialized hardware.**

Real memories are rectangular and distributed

Cortex M0 (and a lot of other processors) just give up and fault.



LDR 1:
Must do the 2 accesses, shift and mask off unused bits

- **This is doable, but slower and requires specialized hardware.**

Alignment in practice

- Early RISC CPUs had alignment constraints.
  - Easy to cause problems.
  - Parsing, network protocol handling often lead to cases where you want to do unaligned access.
- CPUs that support unaligned access are easier to program.
  - Not doing so led to lost sales.
  - Most modern CPUs support unaligned memory accesses.
    - Including higher-end ARM CPUs.
- Cortex-M0 is architected to not support unaligned accesses.
  - The instructions and registers won’t support arbitrary unaligned accesses.
  - Consider the SP register...

SP and alignment

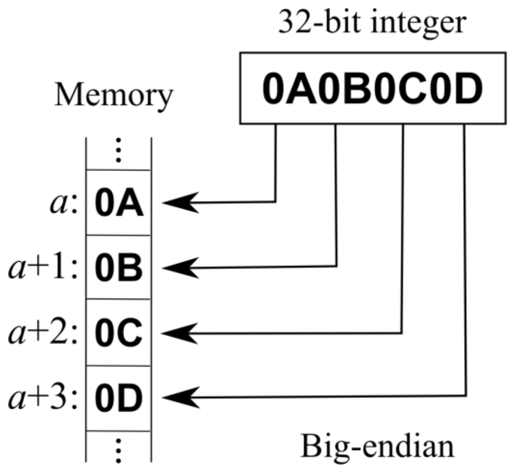
```
// sp is unmodified
//
mov  r7, sp
adds r7, #1
mov  sp, r7
```

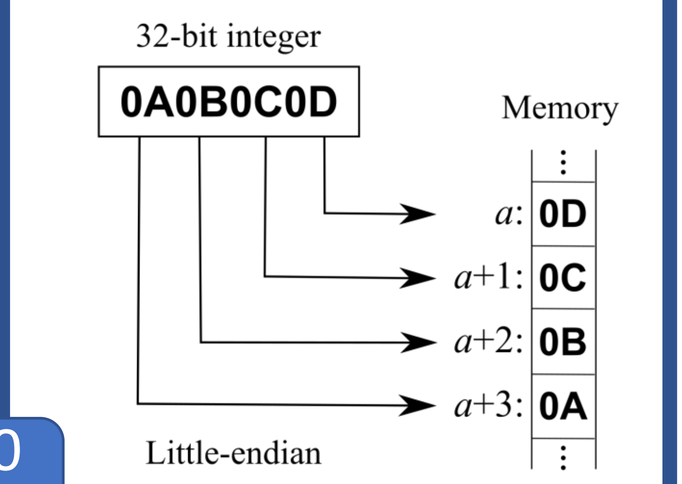
- Can the SP register contain an odd number?
  - No.
  - The two least significant bits of SP are hard-wired to 0.
  - The SP value is unchanged by these instructions.

Endianness

Recall

- Word storage
- Word = bytes (0x400,0x401,0x402,0x403)?
- Word = bytes (0x403,0x402,0x401,0x400)?
  - It depends…
- Big endian: MS Byte at address xxxxxx00b
  - e.g., IBM, SPARC
- Little endian: MS Byte at address xxxxxx11b
  - e.g., Intel x86
- Mode selectable
  - e.g., PowerPC, MIPS,ARM (generally)





ARM Cortex-M0 (ECE 362)

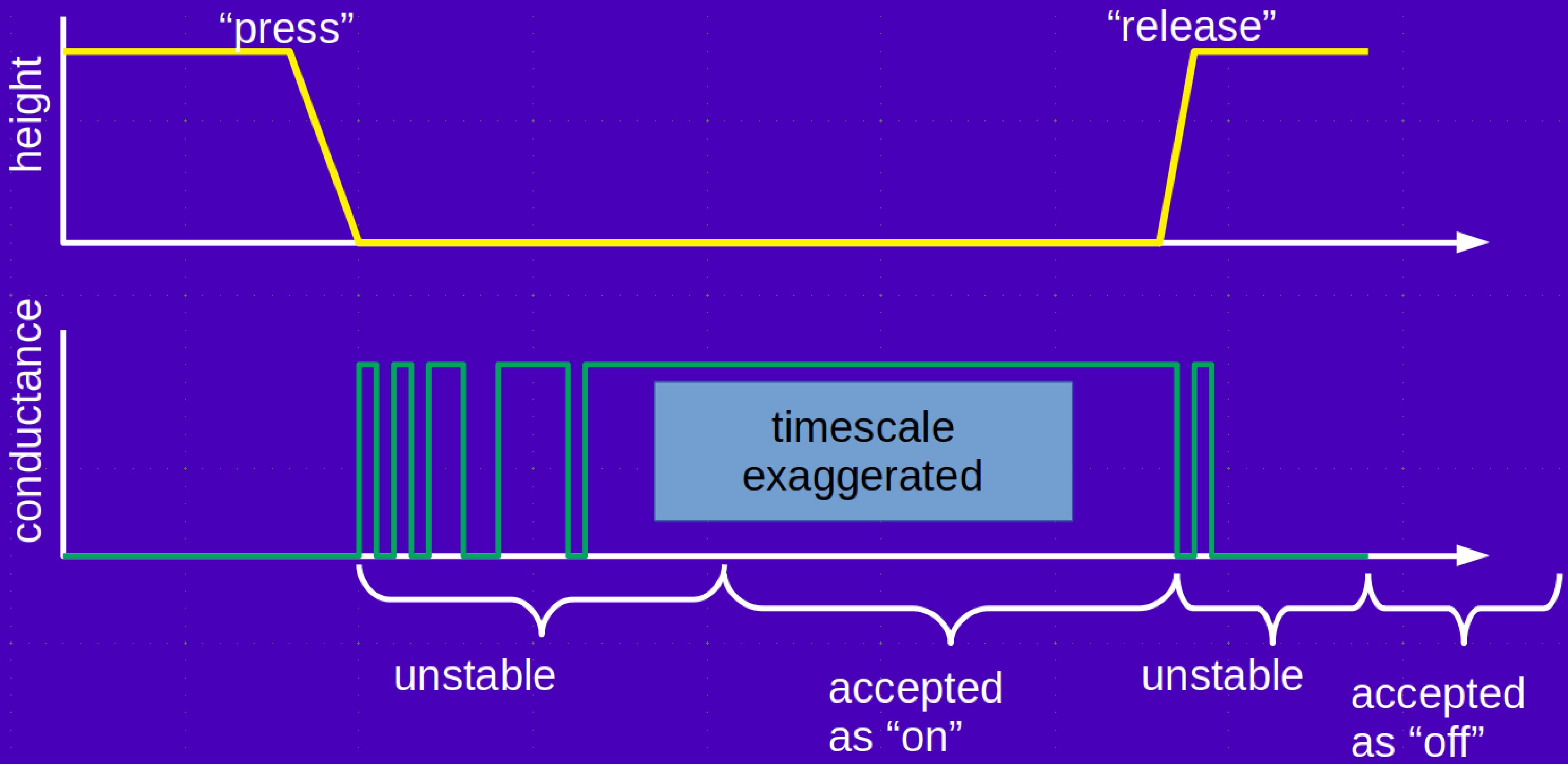
Some endianness notes

- ARM Cortex-M0 endian has an instruction to change endianness
  - REV: will reverse the 4 bytes in a 32-bit integer
- Most of the time you never really have to worry about endianness
  - Only matters when trying to access partial words

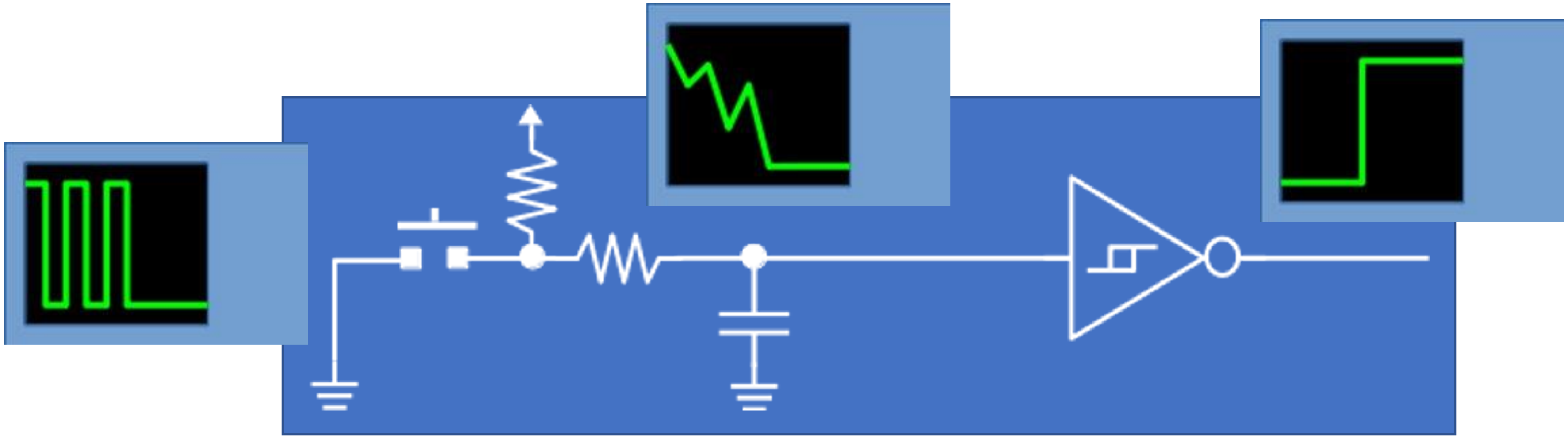
In the real world, everything bounces

- Most mechanical switches consist of a conductive plate that closes a circuit between two contacts.
  - Press the switch, and ***<u>bounces</u>***.

What does a bounce look like?



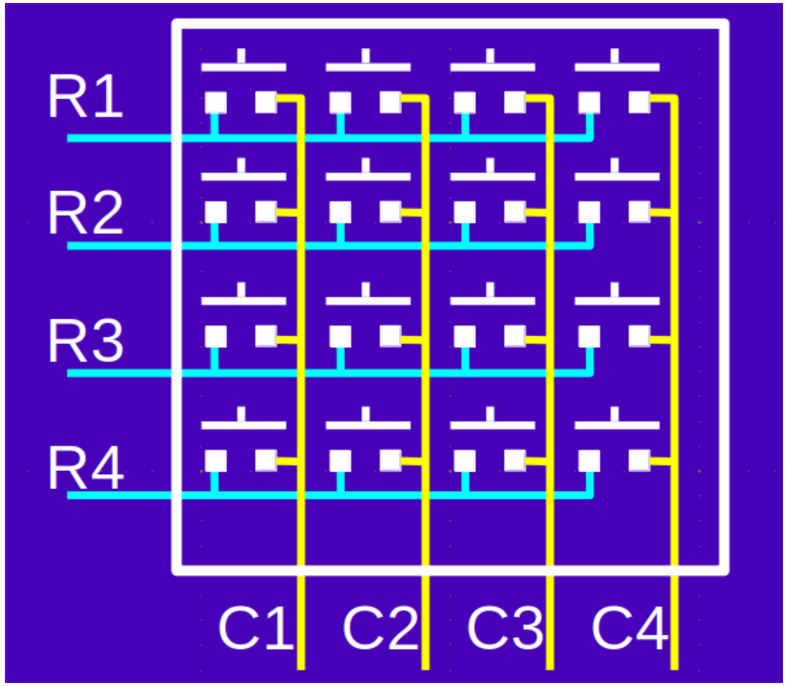
How to debounce one button electrically



- The on-off-on bouncing of the switch is "smoothed" by the R-C network. Normally, the slow rise and fall time causes problems for digital inputs.
  - The Schmitt Trigger doesn’t mind slow inputs.
  - As long as the RC constant is much larger than the bounce time, the output is a bounce-free digital signal.

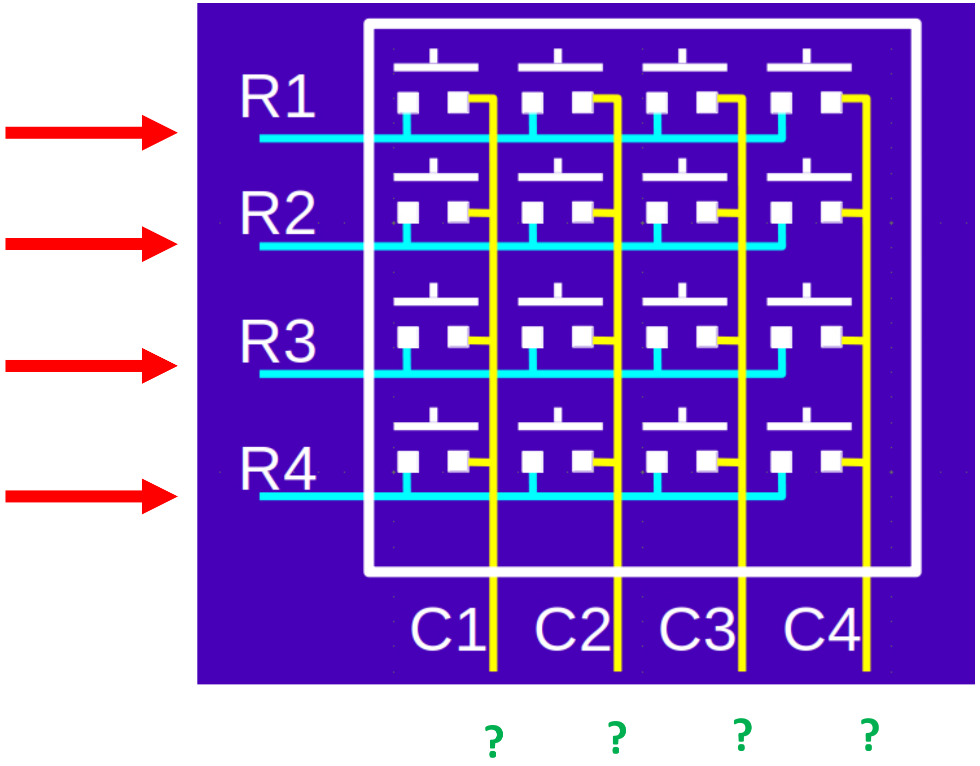
Keypad matrix

- Not many situations call for a single button.
- Most of the time, you have a matrix of keys.
  - For 16 buttons, you don't want to waste 16 pins (and 16 Schmitt inverters) to read them all. Arrange them in a matrix.
  - And they still bounce.
  - You must scan them.
  - You don't have to watch every button all the time. Just check each rapidly enough to notice a push soon after it happens.



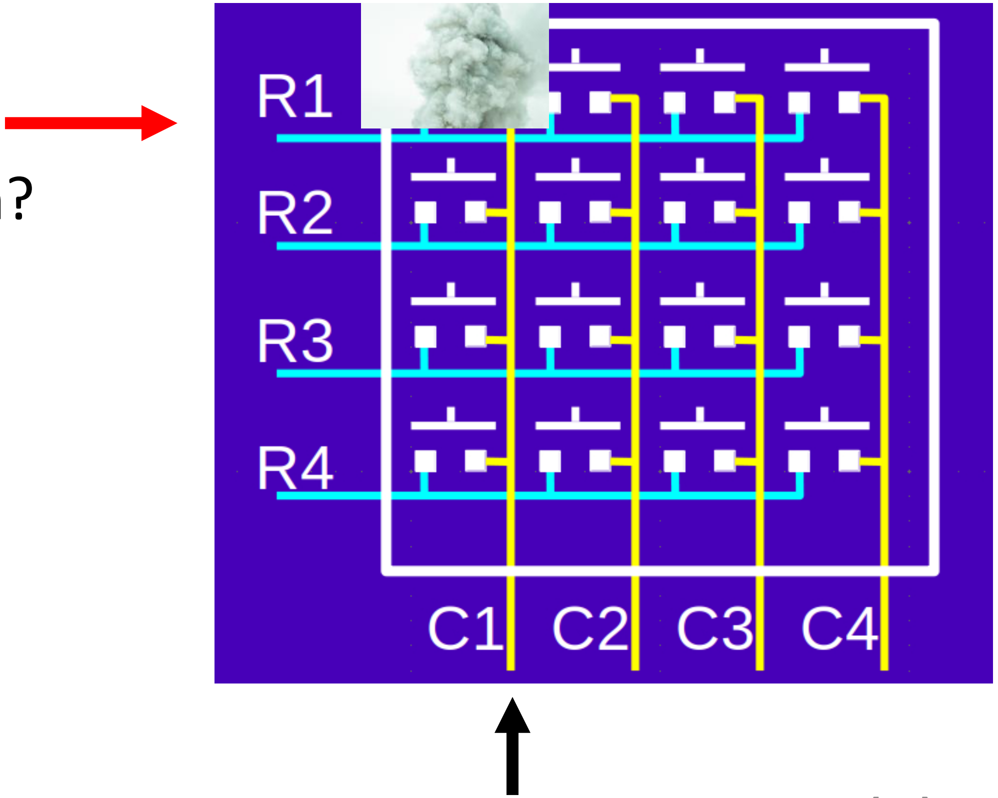
How to scan keys

- Apply voltage to one row.
- Check for voltage on columns.
- Turn off voltage.
- Turn on voltage for next row.
- And so on...



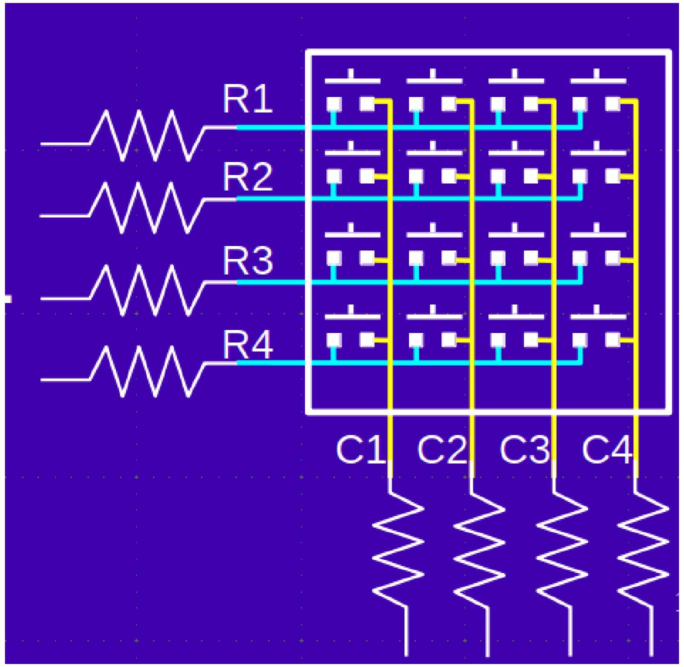
Watch out!

- You use the STM32 pins to scan by changing pins from input to output. But what if you...
  - Apply power to one row...
  - Apply ground to one column...
  - And then you push the upper-left button?



Safety first

- At least put some resistors here… to limit how much current flows through a button when you make a mistake.



Use of timer/interrupt

- The work of scanning can be done incrementally using a timer interrupt.
- On each interrupt, the ISR will:
  - read all the columns
  - put the value read for each key of the current row into its own history byte
  - turn off the voltage for the row
  - turn on the voltage for the next row (for the next ISR invocation)
  - return

- Why do it in this order?
  - You could turn on a row and immediately read the columns, but this way gives the voltage on the row/column connection time to settle in between ISR invocations.

Debouncing a matrix

- As key matrix is scanned, keep track of what the last 8 values read for each key. (1 for currently pressed, 0 for currently released)
  - left shift its latest reading into a byte of memory called a history byte.
- If key idle for a long time, the byte for the key will be 00000000
- The first time a key is pressed, its history will become ***00000001***
- If it bounces, it may be 00000101 or 00010101
- After it is pressed and stable for a long time, it will be 11111111
- The moment it is first released, it will be ***11111110***
- If it bounces on release, it may be 11111010 or 11101010

Detection

- To detect a press or release, search all the history bytes that represent the keys:
  - ***00000001***: key pressed
  - ***11111110***: key released
  - ignore any other values

How Quickly Should We Scan?

- Much faster than keys can be pressed and released.
  - It’s possible to repeatedly press and release a single button 10 times per second (maybe).
- Slower than the total bounce time for any key.
  - Don’t scan so fast that you can read 00000001 multiple times for a single (bouncing) press.
- If a button can bounce for 10ms, and we scan one of four rows every 1ms, then the worst possible history byte for a single press would be 00000101. (Individual bounces separated by 2 bits.)

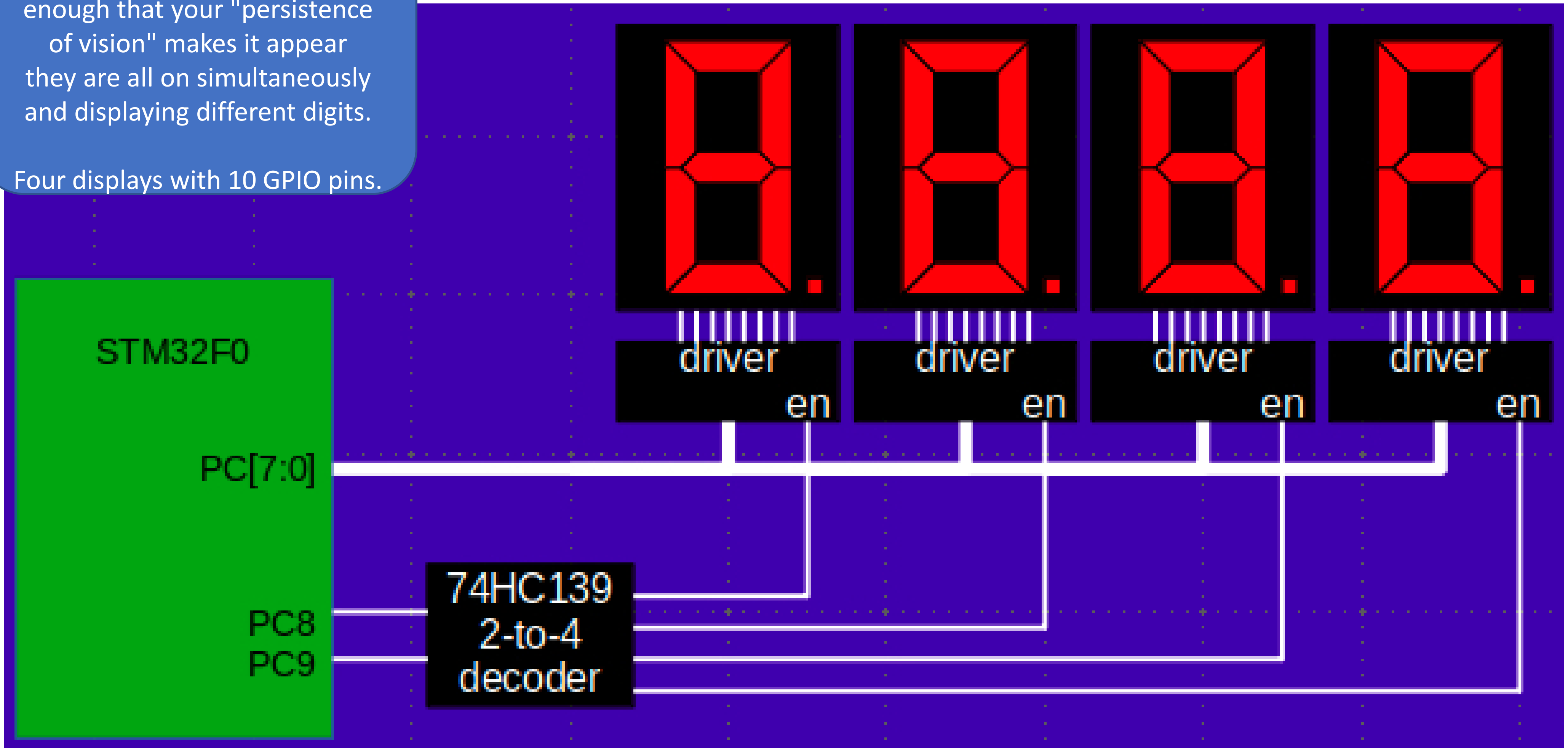
Output Multiplexing

- Key scanning is a specific example of input multiplexing and encoding.
  - If you use microcontrollers, you may spend a lot of time doing things like this.
- Another example: driving displays.
- There are eight 7-segment displays in your lab kit.
  - You do not want to use 64 STM32 pins to drive segments individually.
  - You can multiplex them with far fewer pins.
  - They are already configured in two groups of 4 to allow this.

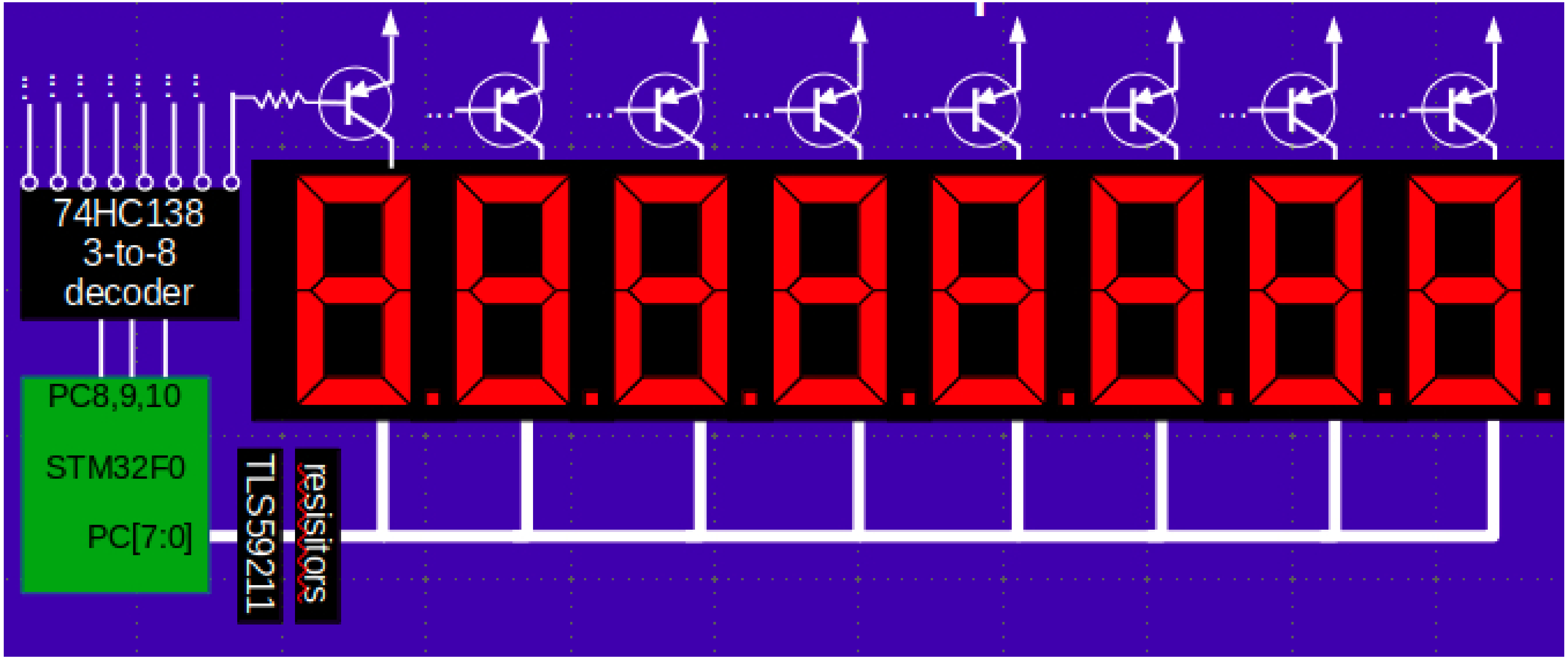
Multiplexing example

Turn on one display at a time.
Rotate through them rapidly enough that your "persistence of vision" makes it appear they are all on simultaneously and displaying different digits.

Four displays with 10 GPIO pins.

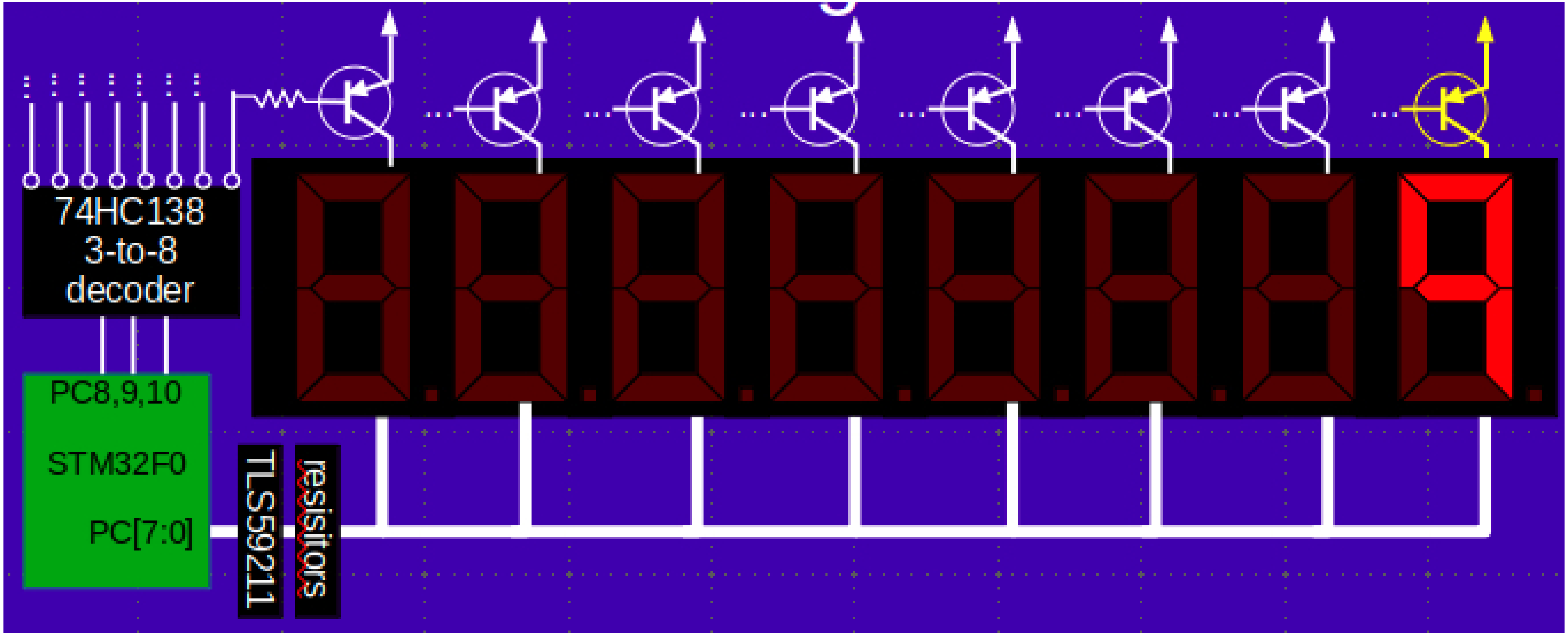


Better example

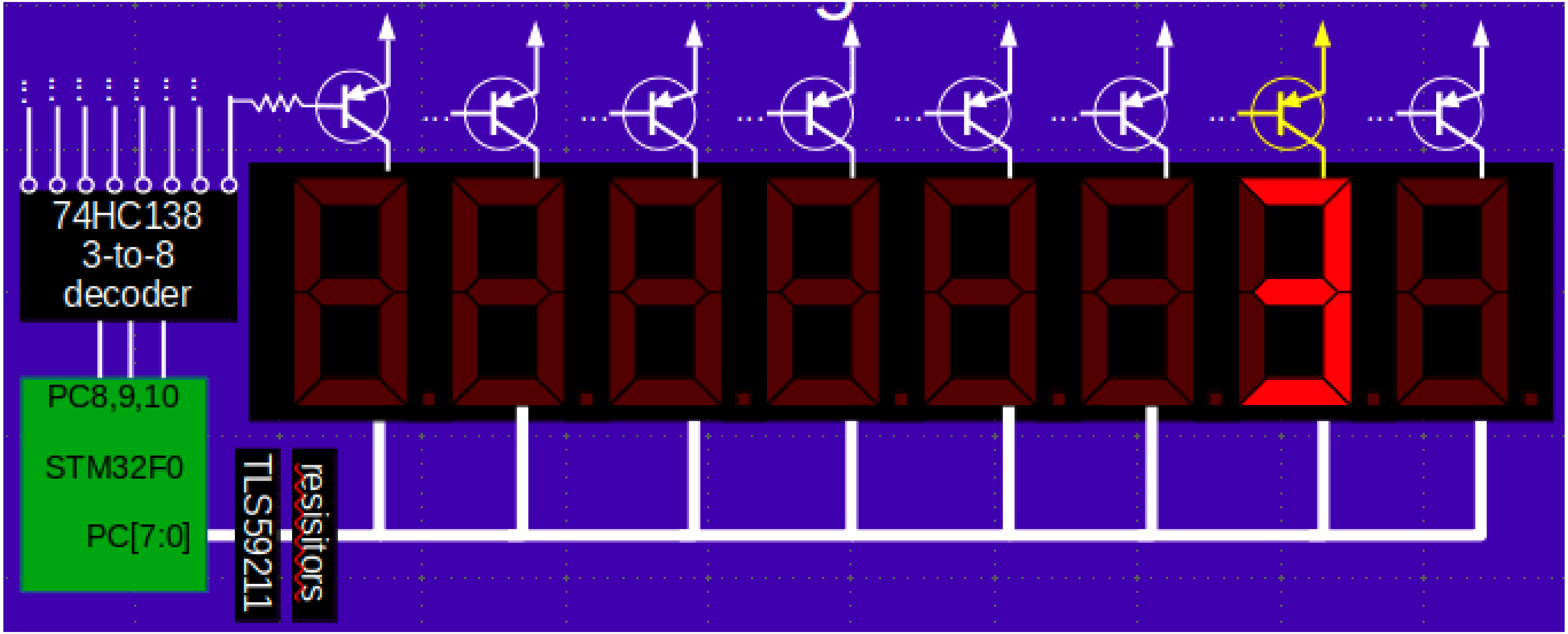


Only using 11 pins on the STM32
(Generally, log2(digits) + 8 pins)

Selecting one digit at a time



Selecting one digit at a time



A lot of multiplexing in uController systems

- Only so many pins, so many things to do!
- We'll use the multiplexing input and output systems shown in this lecture in multiple lab experiments