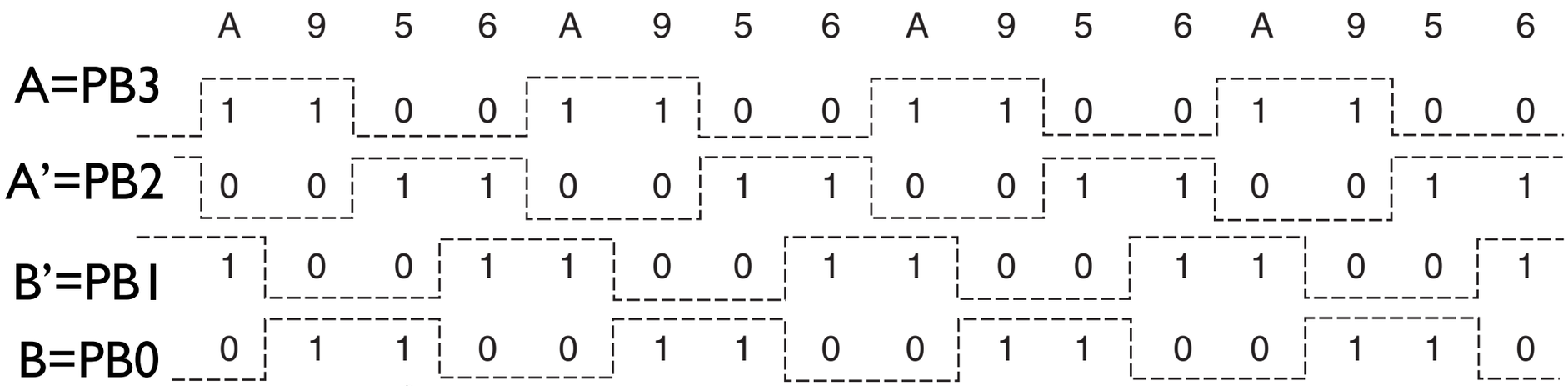
DC Motors III

ECE 3710

If you can't beat them, arrange to have them beaten

- George Carlin

ex: stepper in C



have to hold sequence for some time

A B A' B'
1 1 0 0
0 1 1 0
0 0 1 1
1 0 0 1

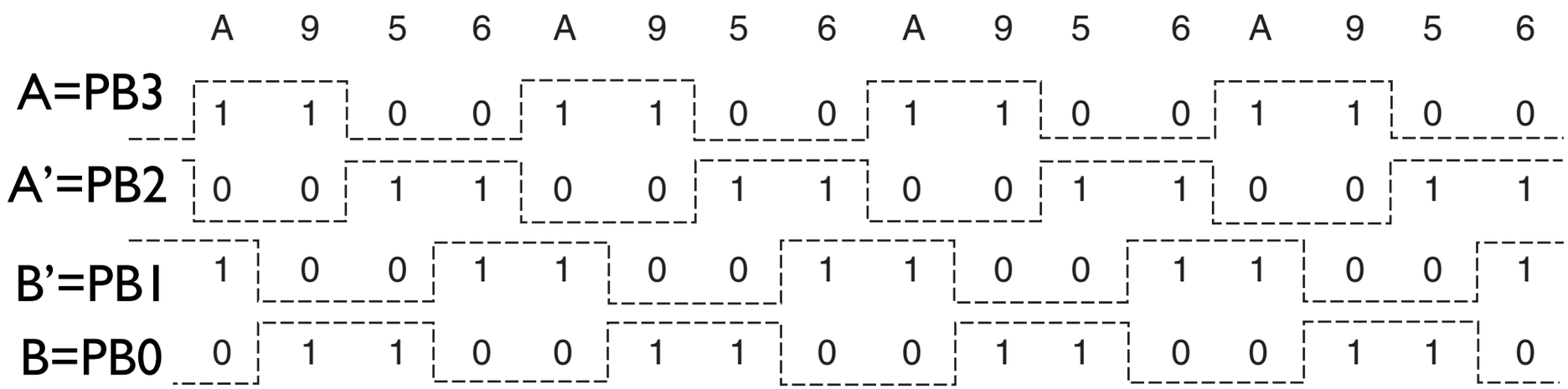
just write sequence to port

rearrange

A A' B B'
1 0 1 0 = 0xA
1 0 0 1 = 0x9
0 1 0 1 = 0x5
0 1 1 0 = 0x6

use timer ISR
(PWM better)

ex: stepper in C




```
// gpio port b base
unsigned char *PB = (unsigned char *) 0x40005000;
unsigned char STEPS[] = {0x0A,0x09,0x05,0x06};
unsigned char I = 0;

void SysTick_Handler(void)
{
   PB[0x3FC] = STEPS[I]; //go clockwise

   I = (I+1)%4; //better than an if-else-statement
}
```

ex: stepper in C

RPMs:

1. SysClk = 12 MHz
2. timer expiration = (255+1)*1/12e6=21.3 us
   (single step)
3. four steps per rev. = 4*21.3=85.33 us
4. 1/85.33e-6*60 = 703152 RPM

a bit fast

Memory-mapped External Peripherals I

ECE 3710

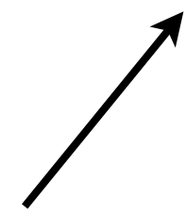
interfacing:

sending/receiving data to/from external devices

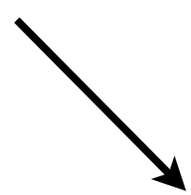
(take as example)

think about devices ⟶ external memory, lcd, rtc,dac

it should do something with data



we want to interface with a device



send and receive data

old way: tedious

(we're responsible for everything)

requirements:

1. tx/rx data
2. where it should go/come from
3. what is it

what should the device do with it

external memory mapping:

1. data written to uC memory is sent to external device
2. data read from uC memory comes from external device

uC is configured to handle:

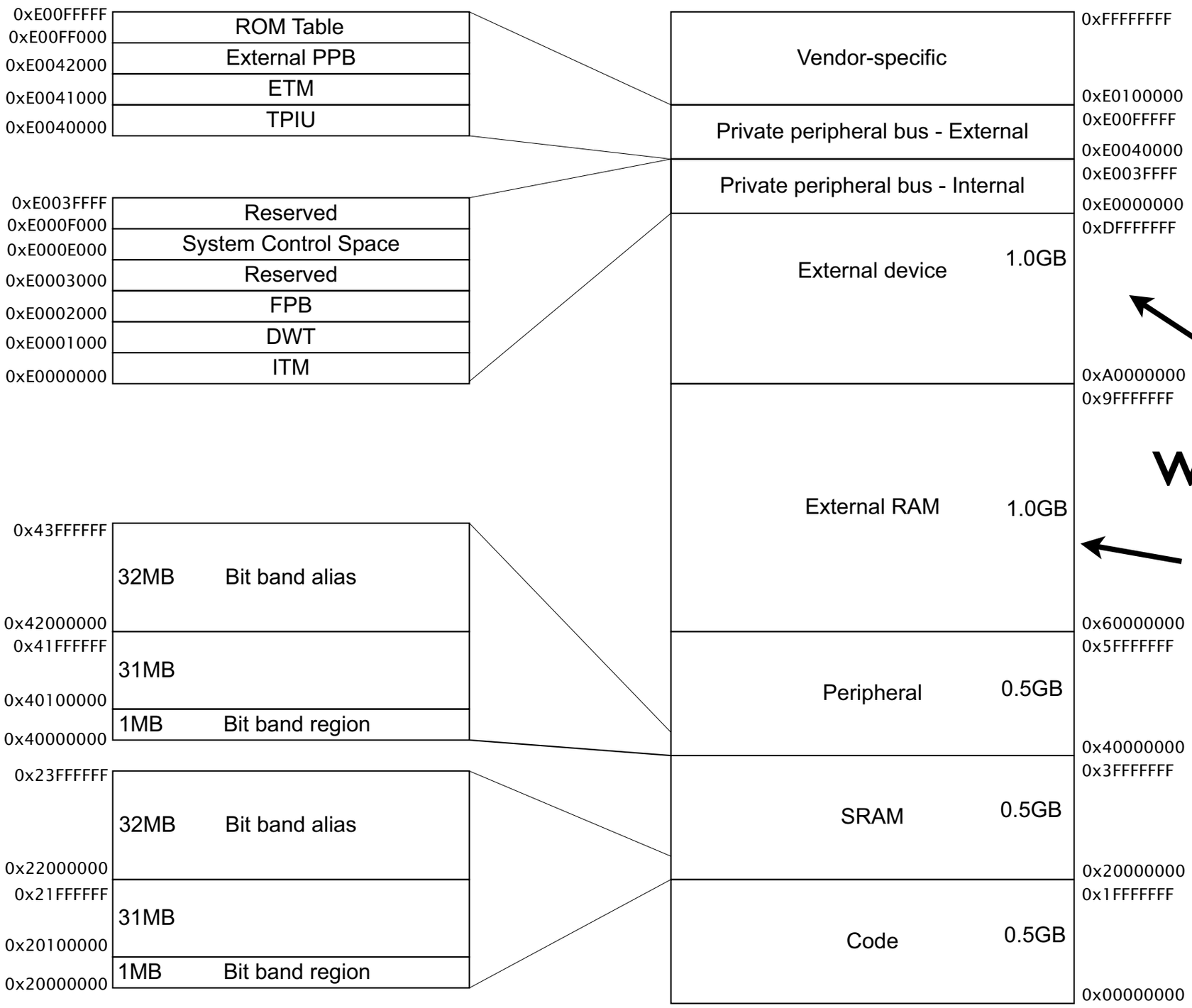
1. timing
2. pin assignments for data/addr
3. sending data/addr
4. receiving data

result: external device appears local

(reads write don't require your code to set/read pins directly)

ARM memory map

which addresses point to which things

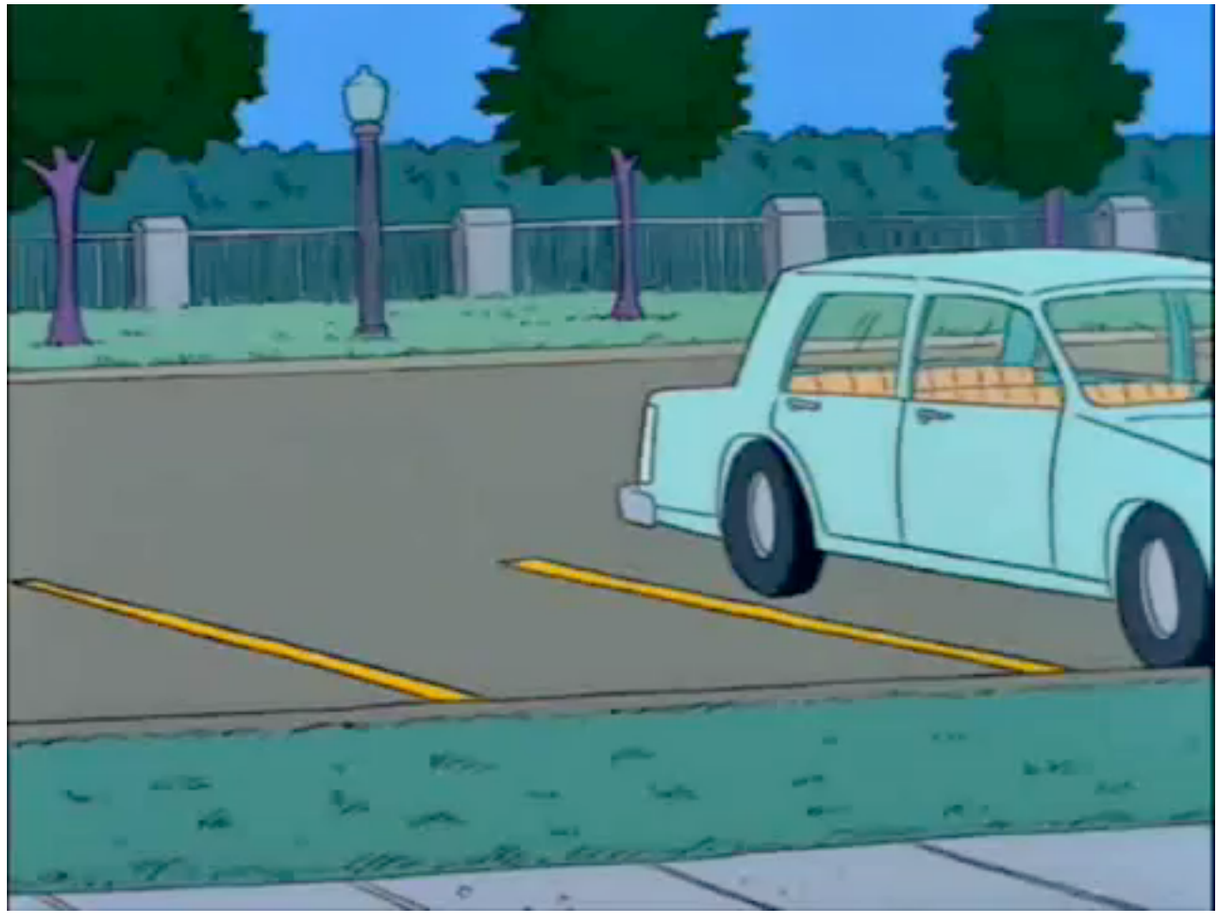


writes here directed at external devices

addr sent on i/o pins; data sent/recv on i/o pins

what was external, now

internal: this is good...



...and exciting

(to me at least)

external memory map

dedicated interface
(need to understand SDRAM)

external RAM:

SDRAM

external device:

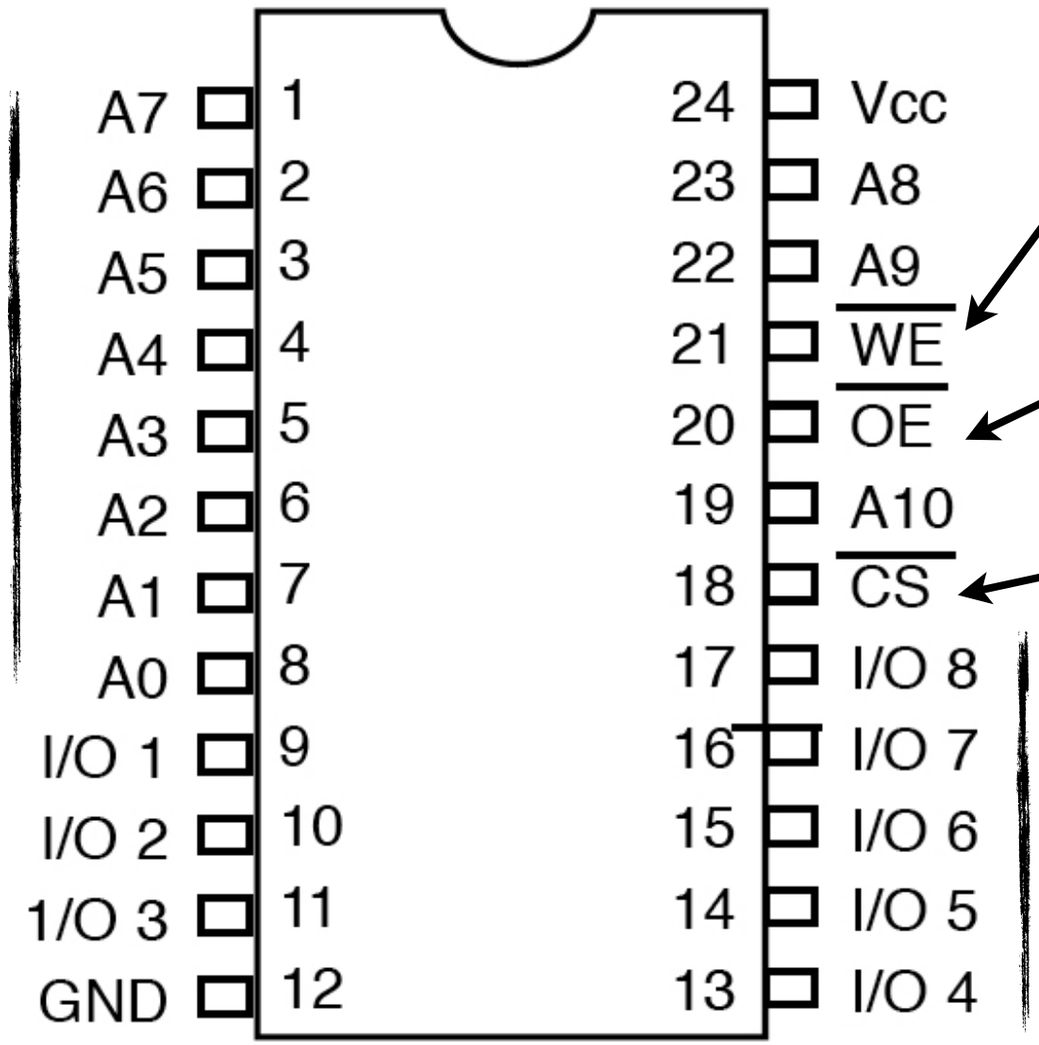
everything else

our focus

Q: how many bits for:

1. addr
2. data

example: SRAM



enable writing

enable reading

activate device for i/o (sometimes CE)

address
(2^8 bytes)

pins A0--7
(A8--A10 grnd)

data
(8 bits)

write `0xAA` to address `0x34`:

SRAM needs to see
(at same time)

```
  A[7:0]=0x34
I/O[7:0]=0xAA
  CS=0, WE=0
```

ex: uC configured for mm with 256x8 device

addr: 8 bits
data: 8 bits

write `0xCCAA` to
`0xA0001234`

a write to mm location:

1. addr = `0xA0001234 & 0xFF = 0x34`
1. data = `0xCCAA & 0xFF = 0xAA`

what device will see
(bits uC sets on pins)

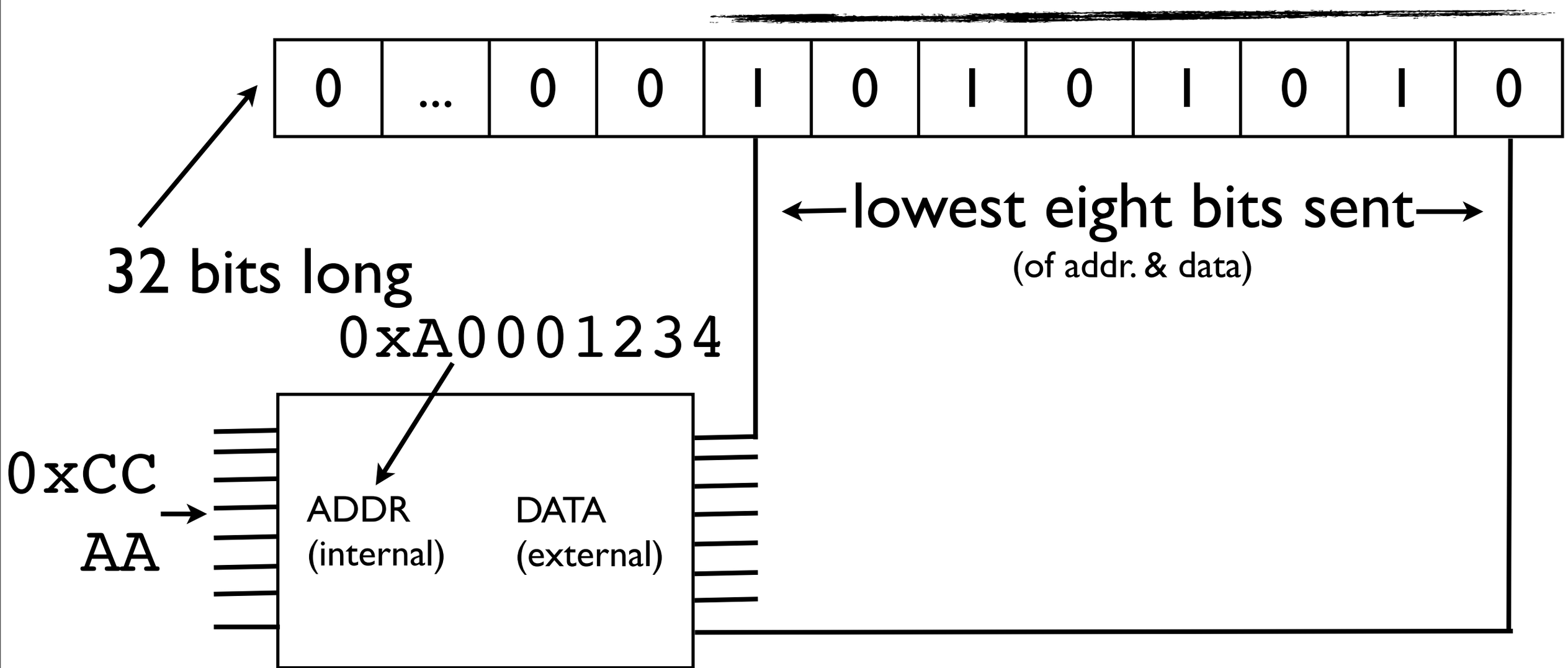
ex: uC configured for mm with 256x8 device

addr: 8 bits
data: 8 bits

write 0xCCAA to

`0xA0001234:`

data: 0xCCAA



MM means less
work for us:

determined by
uC

interfacing requirements (which pins):

1. address
2. data
3. enable: device/read/write

found on LM3S9B96

TI External Peripheral Interface:

1. host bus mode
2. data: 8 or 16 bits
3. addr: 8--28 bits
4. duplexed or non-duplexed mode
5. 32 pins (`EPI0Sx`)
6. mm addr starts at: `0xA0000000`

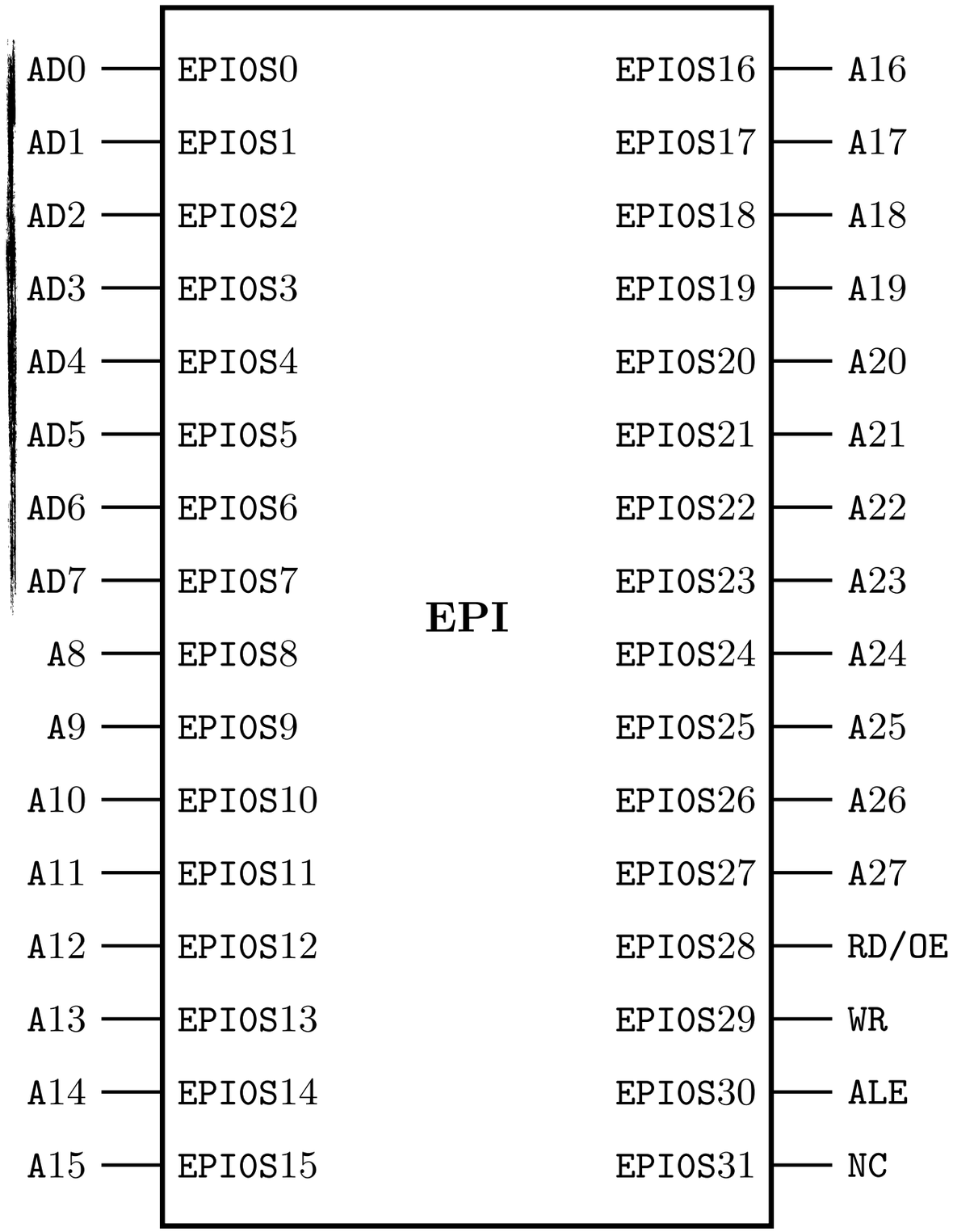
EPI: 8-bit multiplexed mode

8-bits

address and data

2^27 of something

(doesn't have to be bytes)

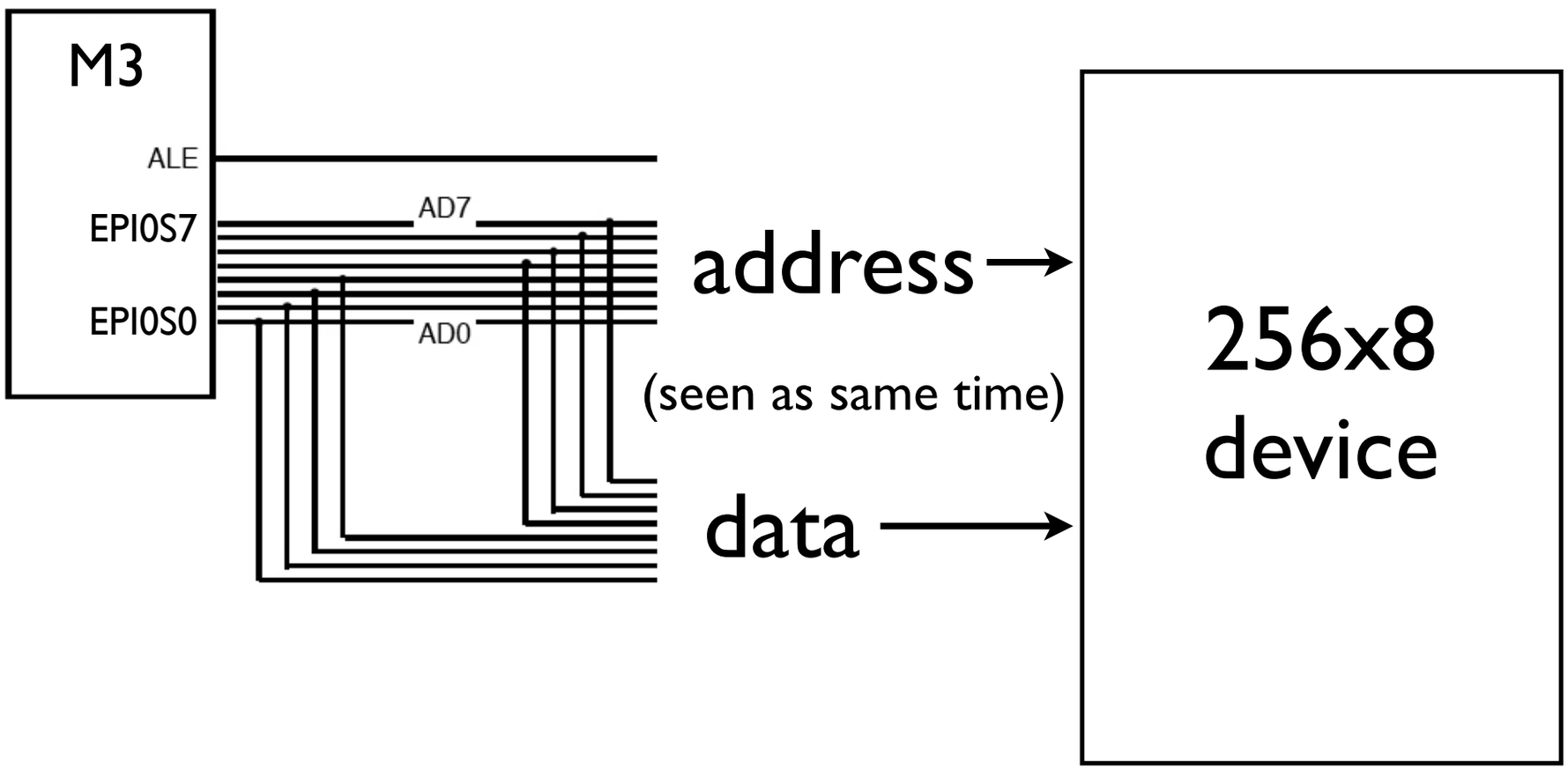


multiplexing: address and data on same pins

must demultiplex

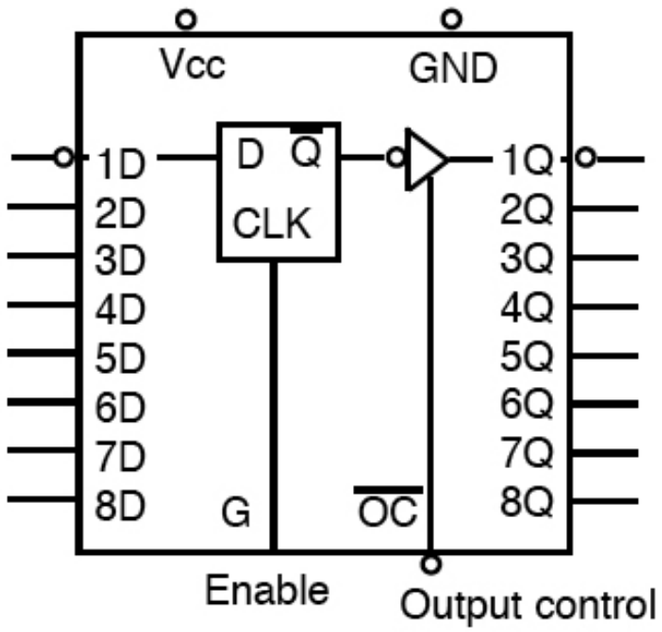
(find a way to separate)

q: why?



need a way to select data or address

address latch



Funtion Table

Output control	Enable G	Enable D	Output
L	H	H	H
L	H	L	L
L	L	X	Q0
H	X	X	Z

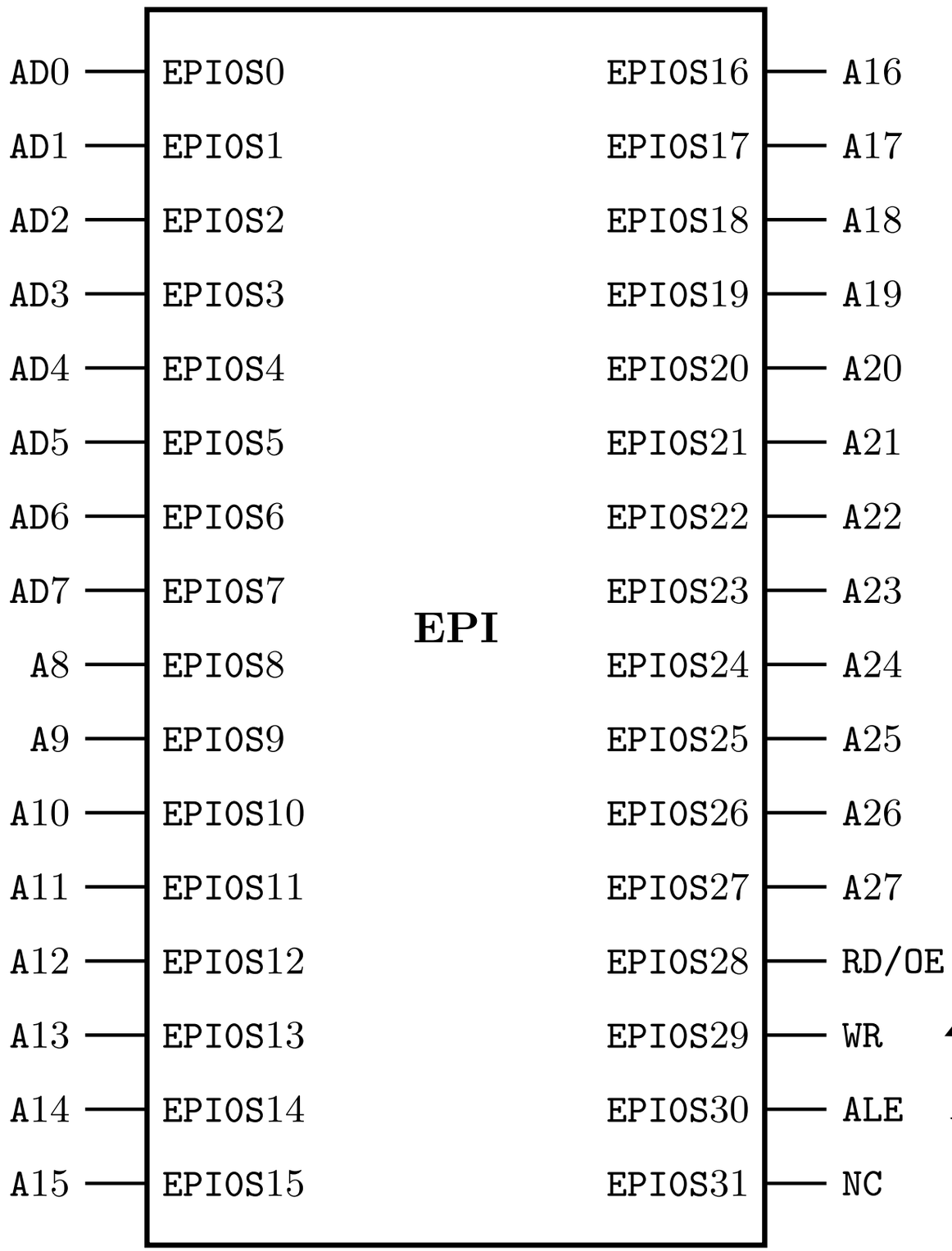
if OC=0

when G=1 Output=D

when G=0 Output=Q

stored value

EPI: 8-bit multiplexed mode



note: use excess addr pins for CS/CE

(can use as few as 8; free up pins)

at device:

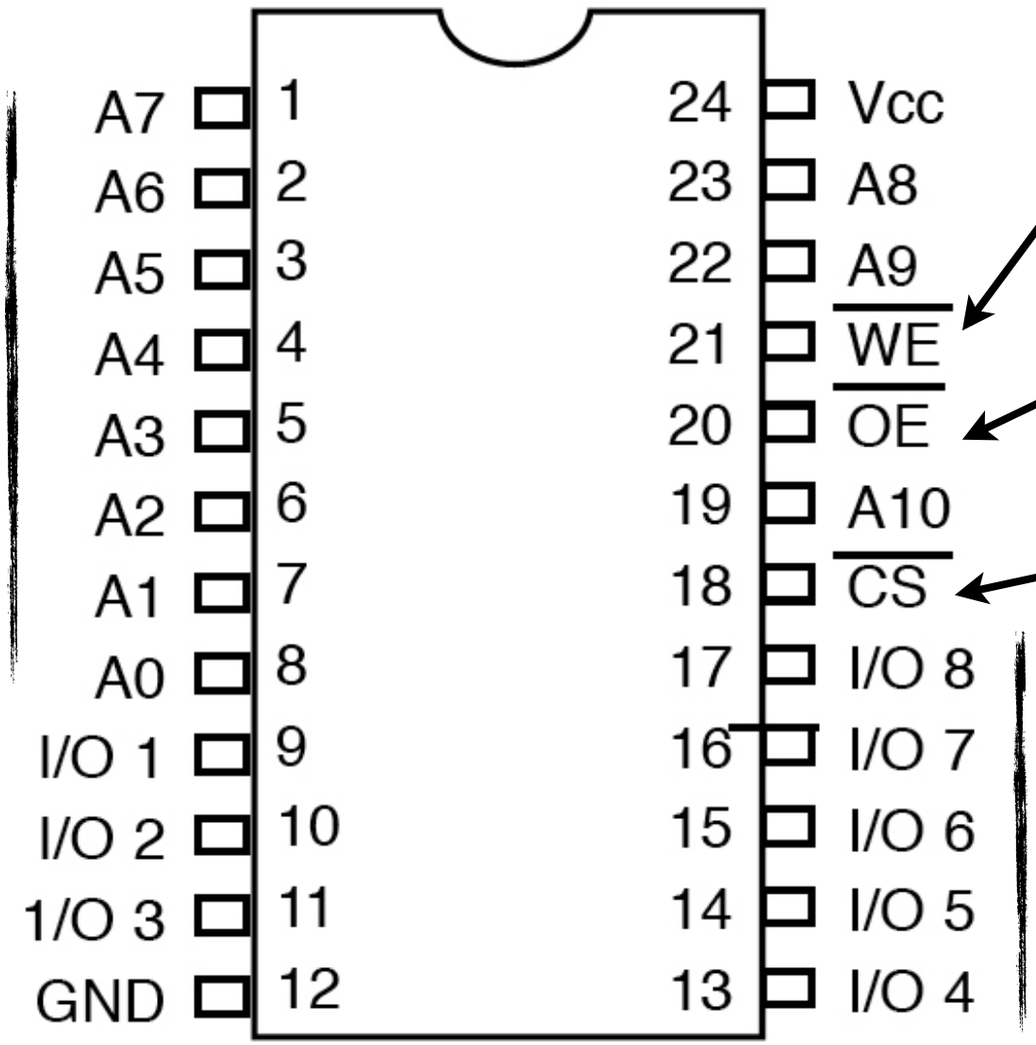
enable read

enable write

(often active low; configurable on uC)

address latch enable

example: SRAM w/TI EPI



enable writing

enable reading

activate device for i/o (sometimes CE)

address
(2^8 bytes)

pins A0--7
(A8--A10 grnd)

data
(8 bits)

write `0xAA` to address `0x34`:

SRAM needs to see
(at same time)

```
  A[7:0]=0x34
I/O[7:0]=0xAA
  CS=0, WE=0
```

to write data: first step

uC does:

1. outputs address

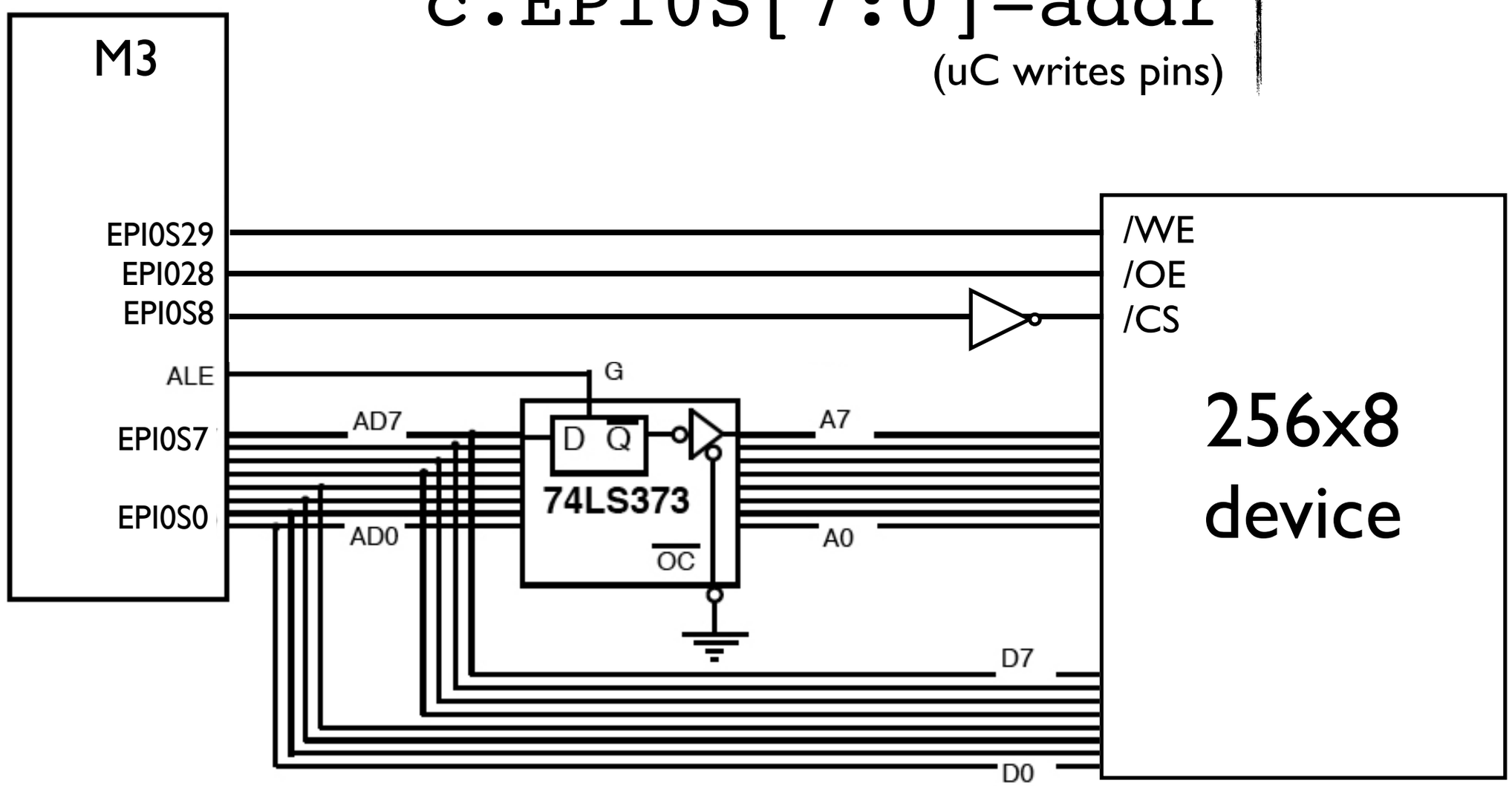
a. enable latch (`ALE=1`)

b. `EPI0S8=1`

c. `EPI0S[7:0]=addr`

(uC writes pins)

we do not set these pins; uC does



to write data: second step

uC does:

2. outputs data

a. disable latch (`ALE=0`)

b. `EPI0S8=1`

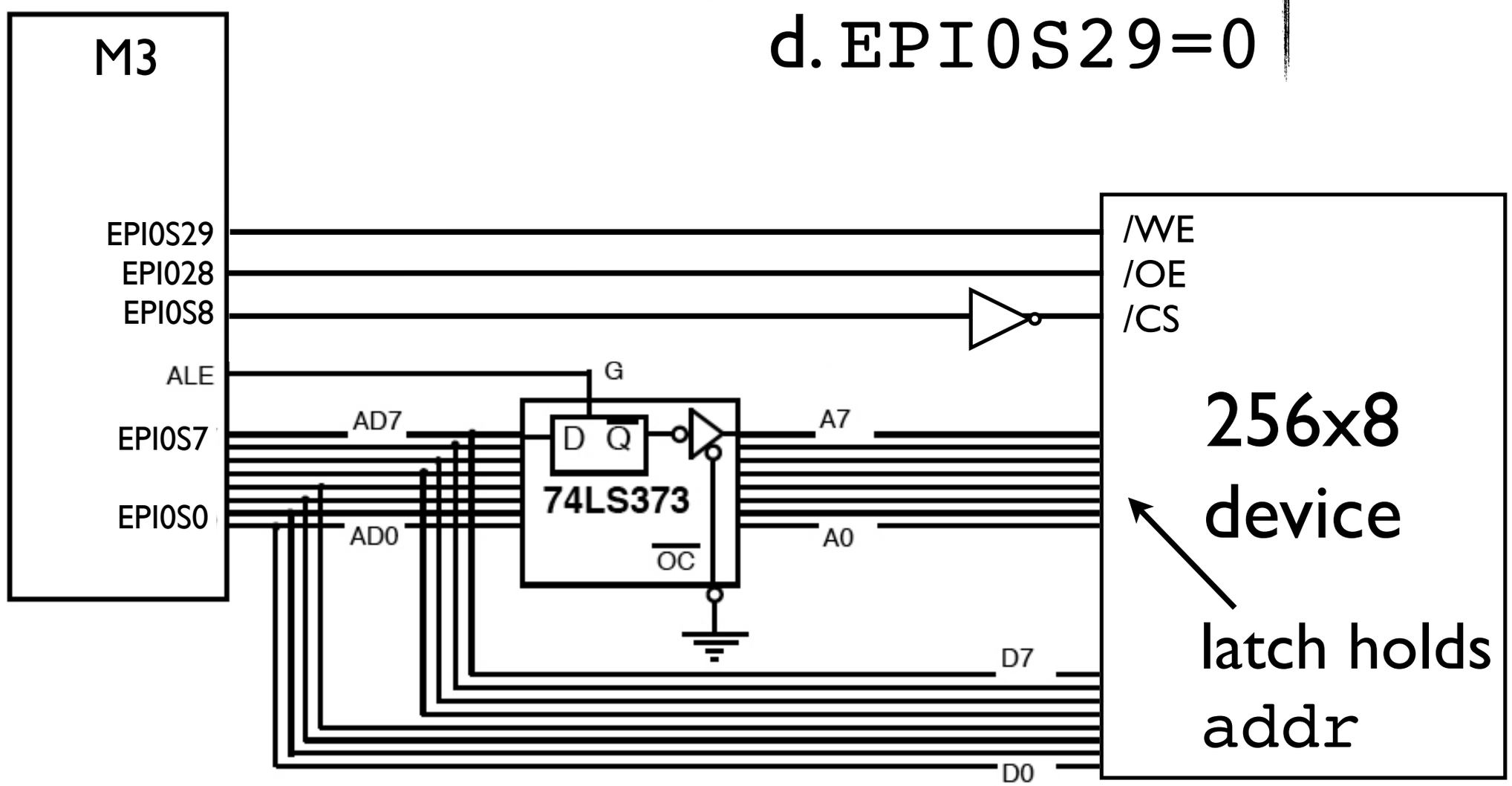
c. `EPI0S[7:0]=data`

(uC writes pins)

d. `EPI0S29=0`

we do not set these pins; uC does

free up bus for data



to read data: first step

uC does:

1. outputs address

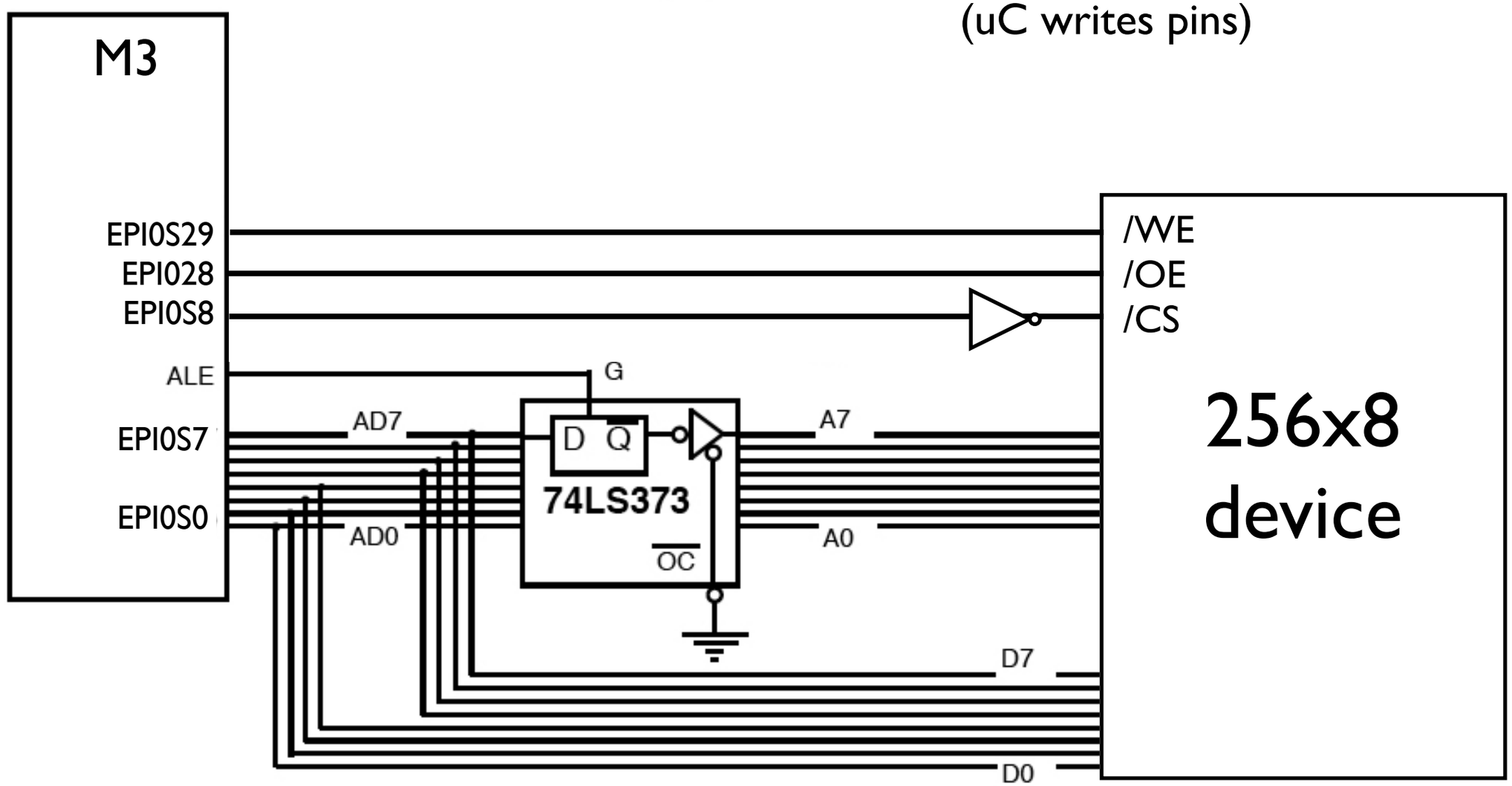
a. enable latch (`ALE=1`)

b. `EPI0S8=1`

c. `EPI0S[7:0]=addr`

(uC writes pins)

we do not set these pins; uC does



to read data: second step

uC does:

2. reads data

free up bus for data

a. disable latch (`ALE=0`)

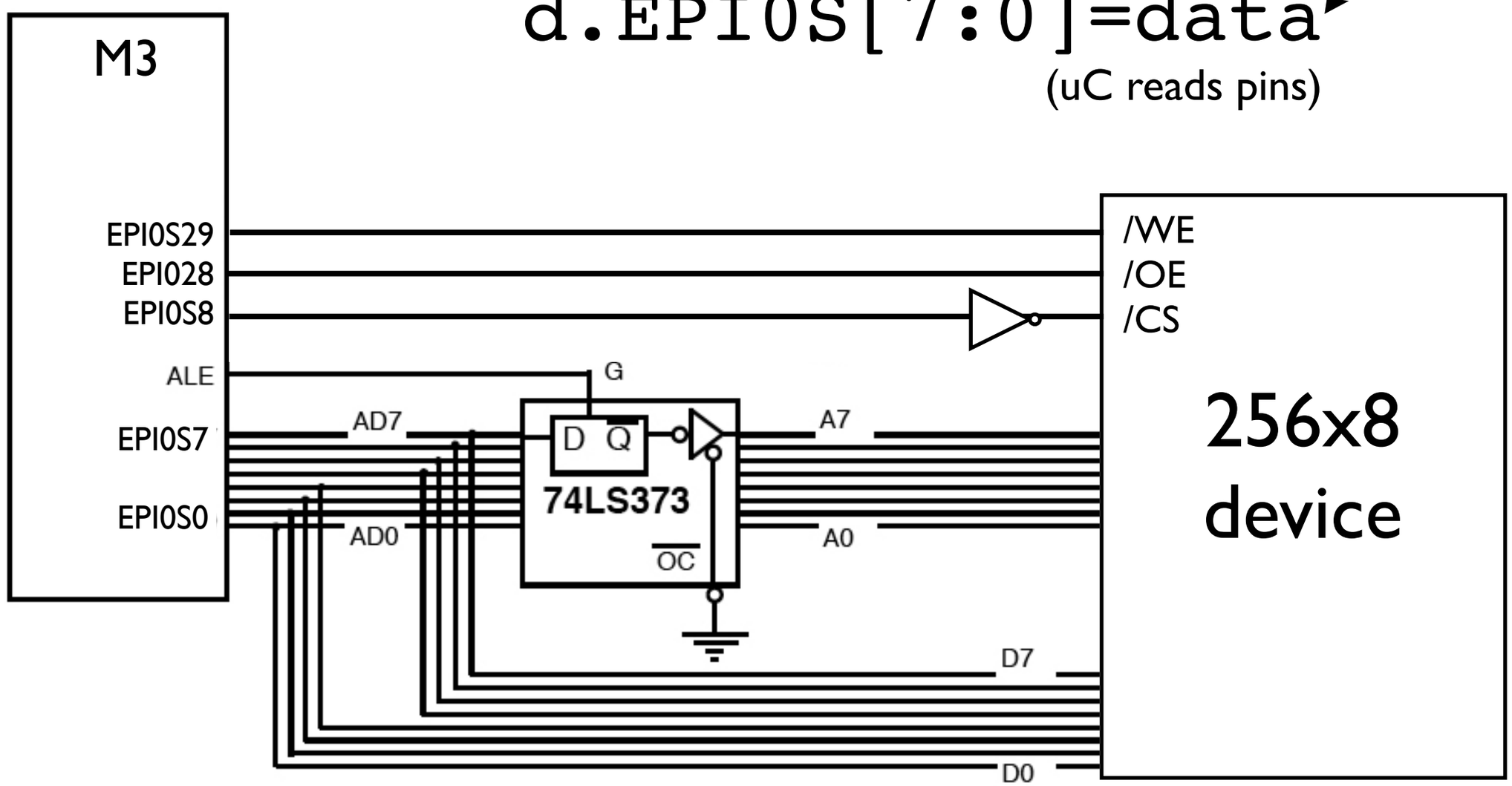
b. `EPI0S8=1`

c. `EPI0S28=0`

d. `EPI0S[7:0]=data`

(uC reads pins)

device puts data on bus



reasoning about
the process:

MM is only useful if we don't have to
handle pins ourselves