

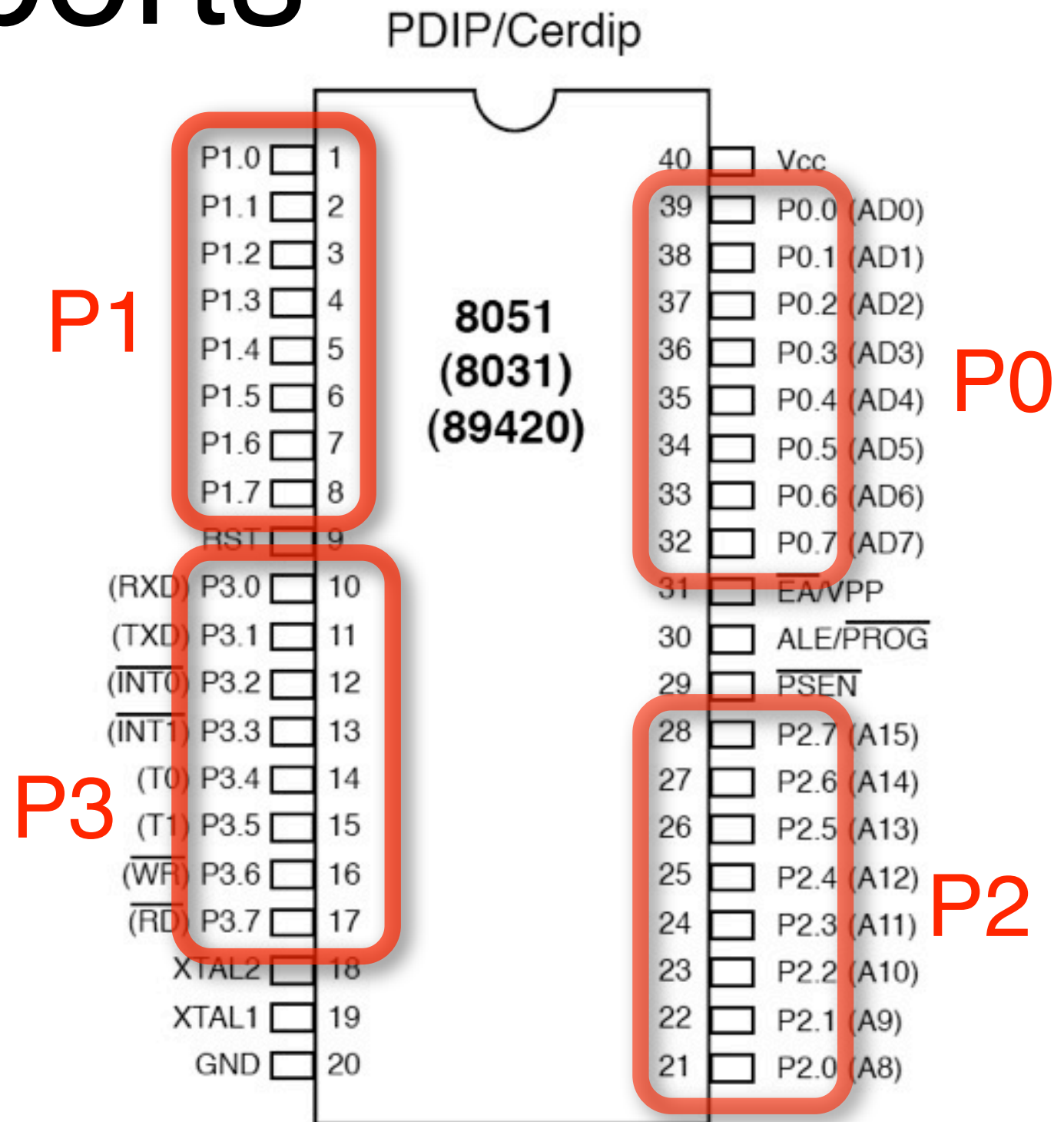
# 8051 General- Purpose I/O Ports

# General Purpose I/O

- Software-controllable pin
- Read and write like registers
- Issues
  - Bit or byte addressable, depending on ISA
  - Directionality - how to specify?
  - Driving strength
  - Overloading with other types of I/O functions

# 8051 ISA: Four I/O ports

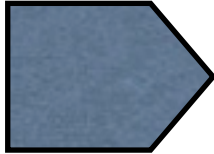
- 8-bits each  
P0, P1, P2, P3
- Ports are like registers
- Difference:  
values tied to the pins
- Bit addressable

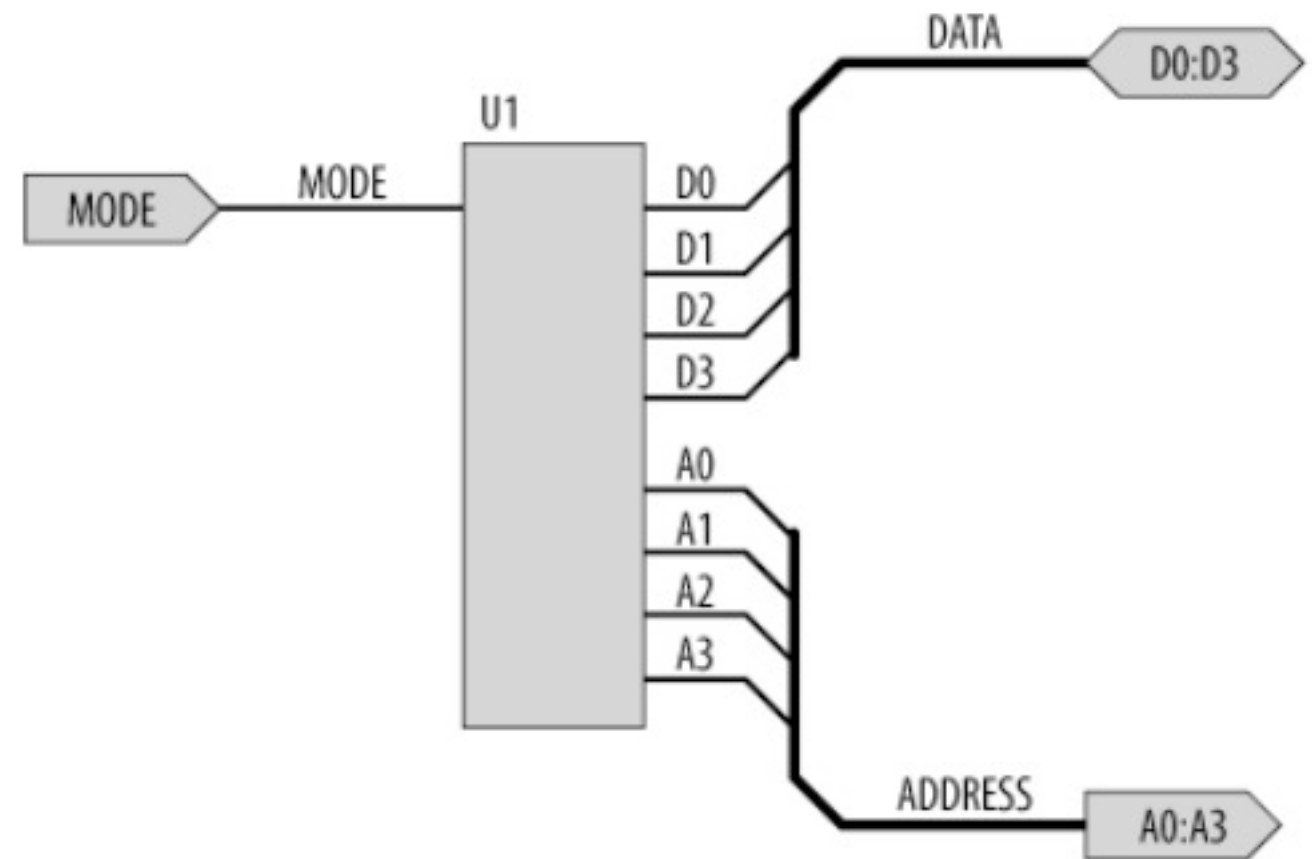


# I/O pin configurations

- Input/output direction
- Pin granularity configuration
- Drive Strength
- Enable and Select of pull resistors
- Interrupt enable
- Status flags

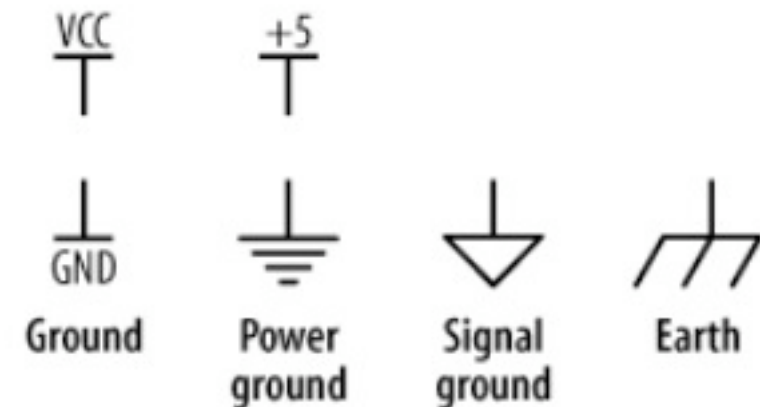
# Bus and Ports

- Bus:
  - group of wires
  - drawn as thick wires
- Ports: 
  - connection to/from another schematic
  - e.g., MODE, DATA, ADDRESS



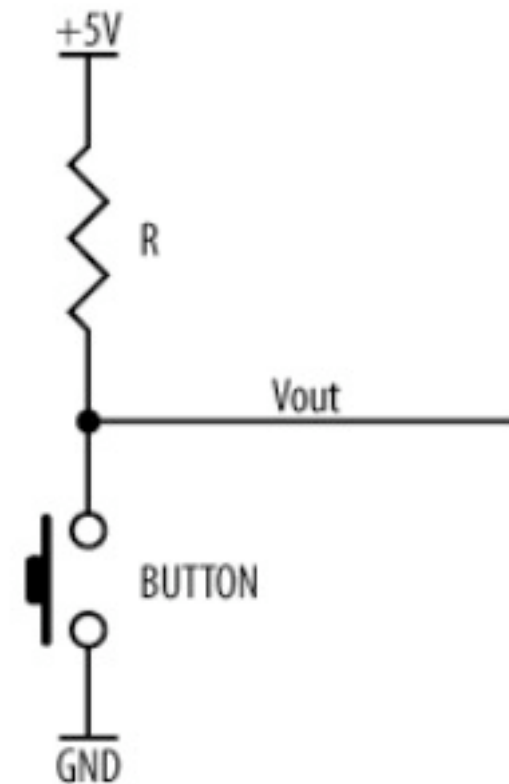
# Power and Ground

- Vcc or Vdd
  - (positive) supply voltage
  - could have multiple
- GND: "ground"
  - Digital, analog, earth, signal
- Some are connected for electrostatic discharge purpose



# Example button with pull-up resistor

- Unpressed: logic '1'
  - pulled up to 5V by R
- Pressed: logic '0'
  - Vout pulled to GND
  - R limits current flow ( $I$ ) to  $I = 5V / R$   
=> use big  $R$  to save power



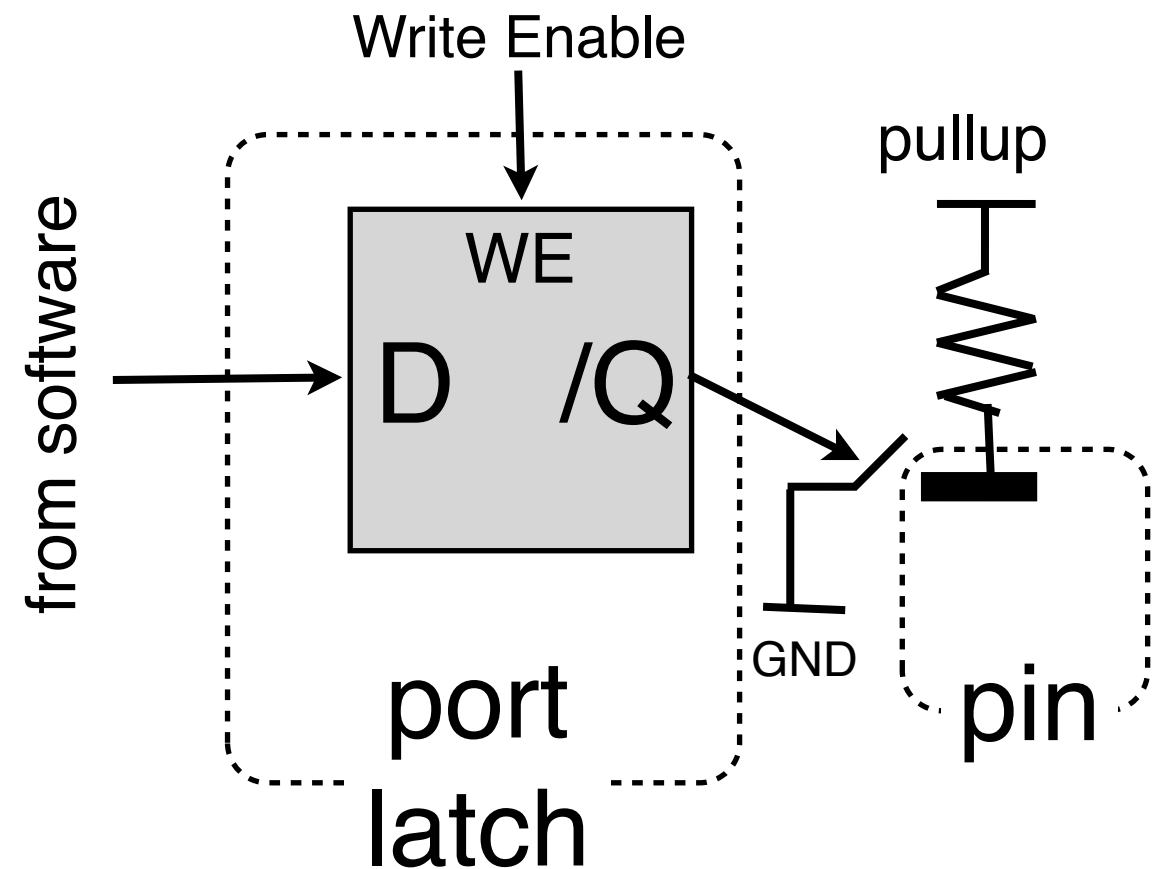
# Directionality - how to specify input or output?

- Option 1 (8051): Set bit = '1' for input
  - Open collector with pull-up
  - Always output, allowing input to override
- Option 2 (most others): set direction bit
  - Config. each port as either input or output
  - Should not mix input and output access



# What a port looks like on 8051?

- Port latch's /Q controls a pull-down to the pin
- Open collector output
  - P0: no internal pull-up
  - P1, P2, P3: internal pull-up
- Bit addressable and Byte addressable



Q=1 (/Q=0): pull up  
Q=0 (/Q=1): pull down

# Output: write to port latch

- Byte access

● `MOV P1, #05EH`

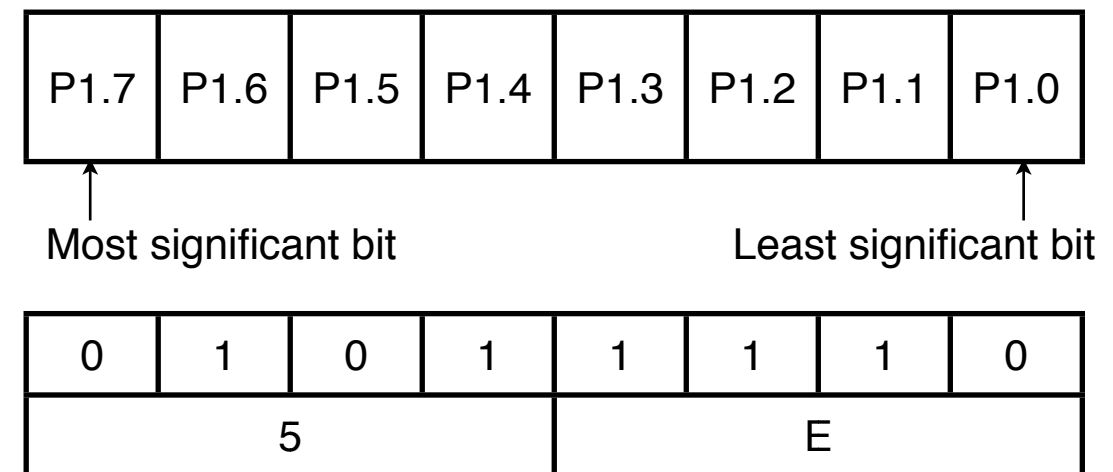
- big-endian bit order

- Bit access

● `SETB P1.1` ;; *sets port 1 bit 1*

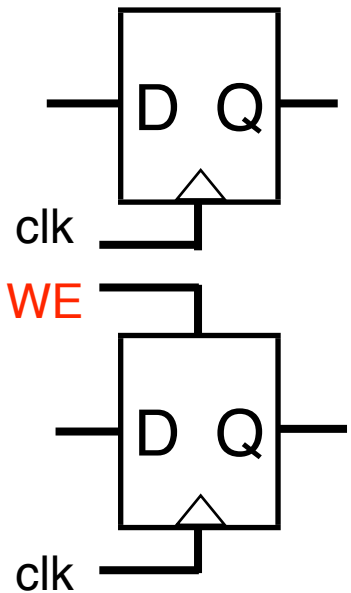
● `CLR P2.3` ;; *clears port 2 bit 3*

Port P1



# "Port Latch"

- Intel calls it "port latch"
- "latch" => hardware;  
"register" => software concept
- Digital hardware terminology
  - "latch" => grabs a new value each clock
  - "register" => has a *write enable*  
(both "latch" and "register" are hardware)



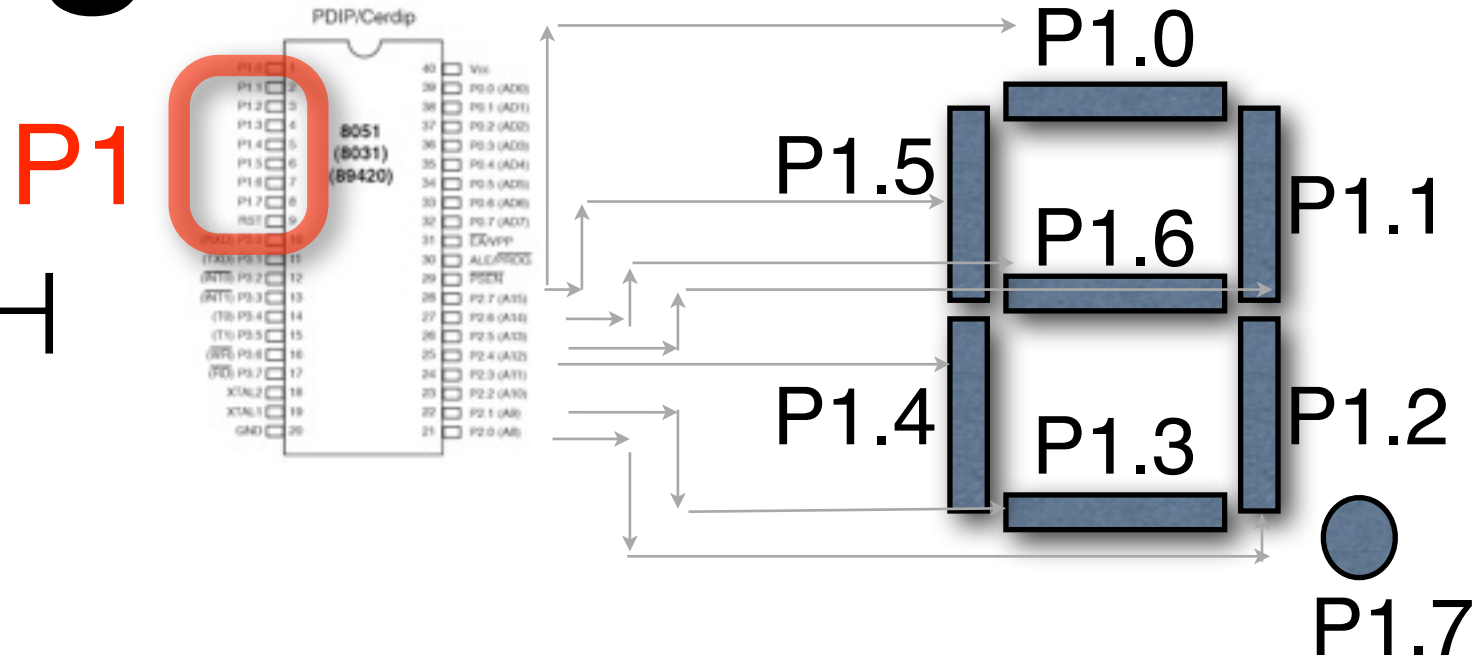
# e.g.: 7-segment LED

- To write "2"

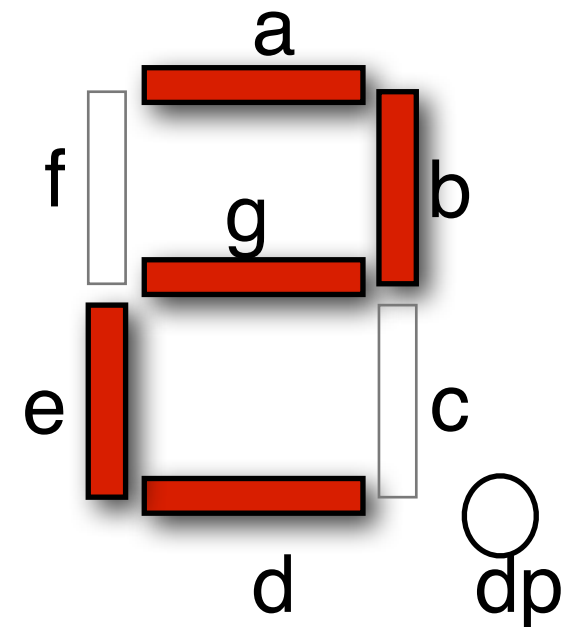
```
MOV P1, #0A4H
```

```
:: 1010 0100
```

```
:: 1 = off, 0 = on
```



- Wasteful in GPIO  
=> solution: decoder chip  
(74LS48) to convert from  
binary (4 bits) to 7-segment  
on/off.

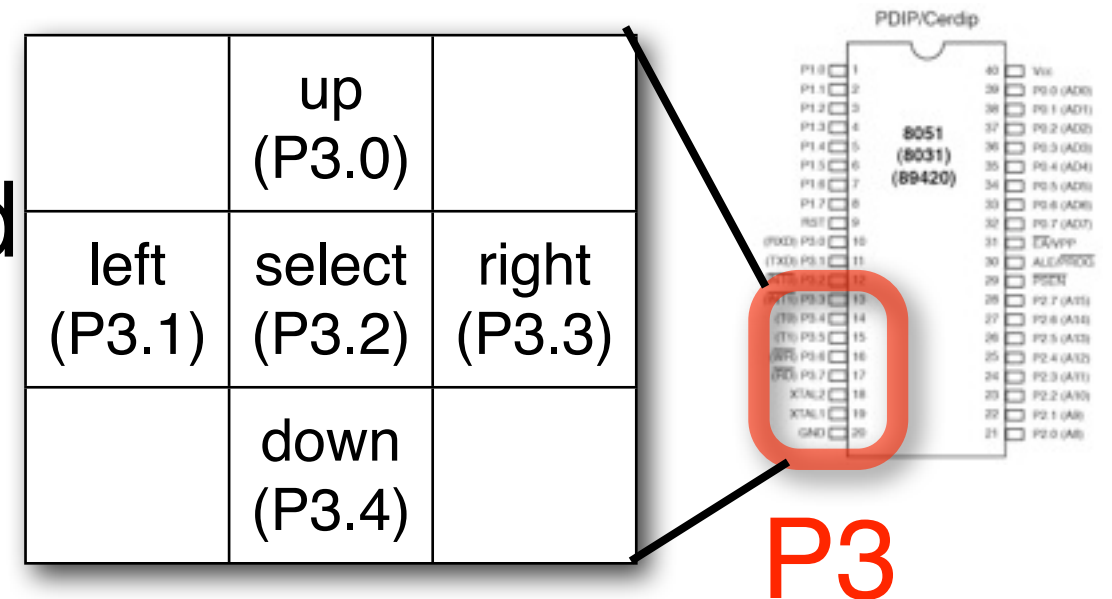


# Input: reading from pins

- Read *pin* value, *\*not\** port-latch value!!
  - `MOV A, P3` ;; read P3 pins, assign to A
- Actually, pin & port may have different values
  - A gets updated, but P3 unchanged!
- How to configure a port to input/output?
  - Actually, Intel-style port always outputs!
  - If you want input, **set port latch to 1**

# e.g.: Joystick input

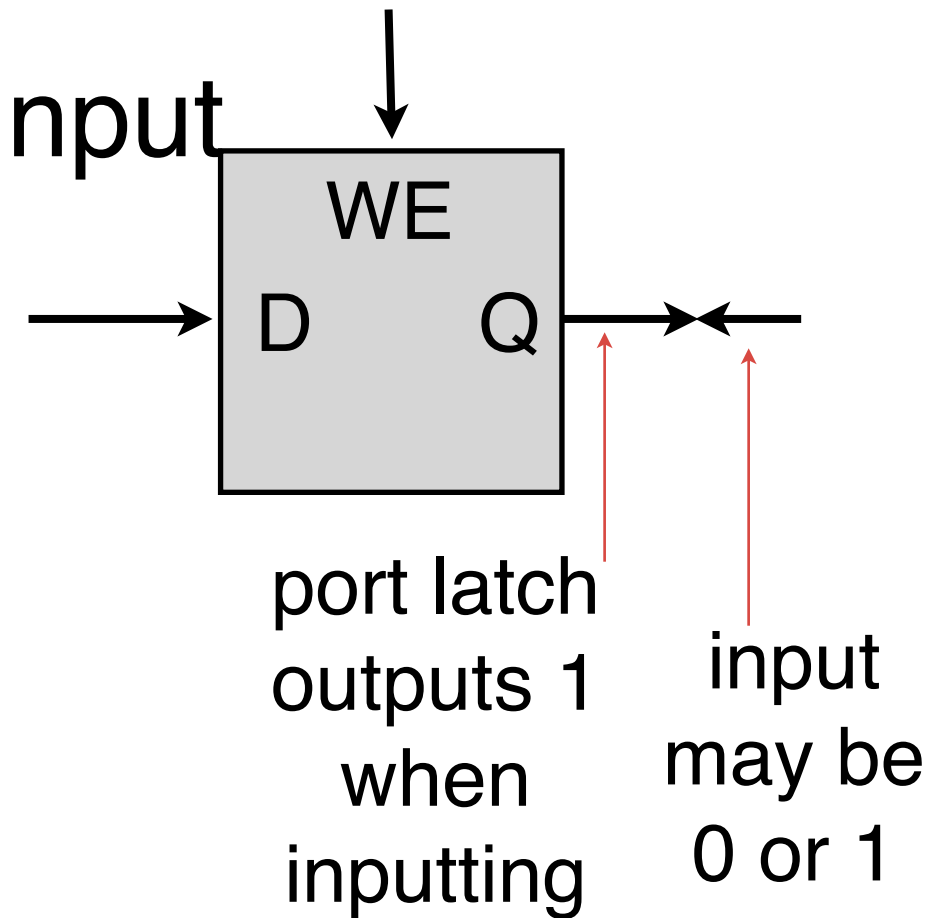
- First, P3 register should be 11111111
  - Power-up default
- Read P3 as a byte
  - `MOV A, P3`
- Read single bit (e.g. up)
  - `MOV C, P3.0`



bit value == 0  
if button is pressed!

# Conflict between port latch and pin

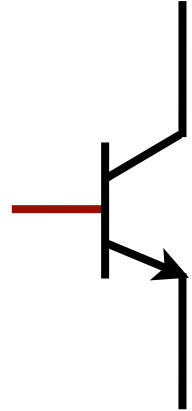
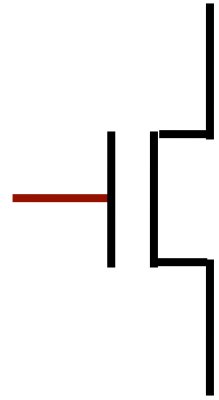
- *Port latch* outputs 1 when input
- *Pin* may input 0 or 1
- Isn't this a short circuit?
  - Actually, no
  - Input should "win"
- Solution: open collector / open drain
  - "weak 1" (overridable), "hard 0"



# Bipolar vs. MOS Transistors

transistors are power-controlled switches

Control signal = base or gate

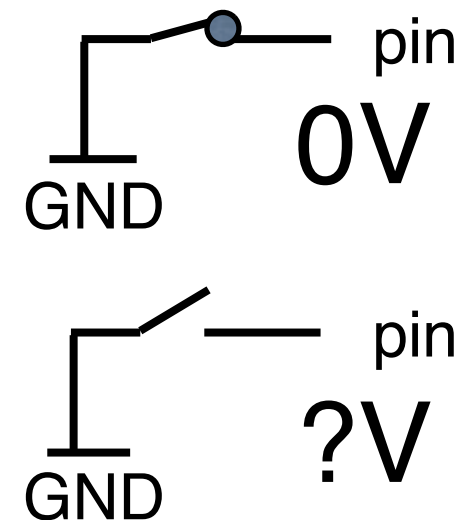
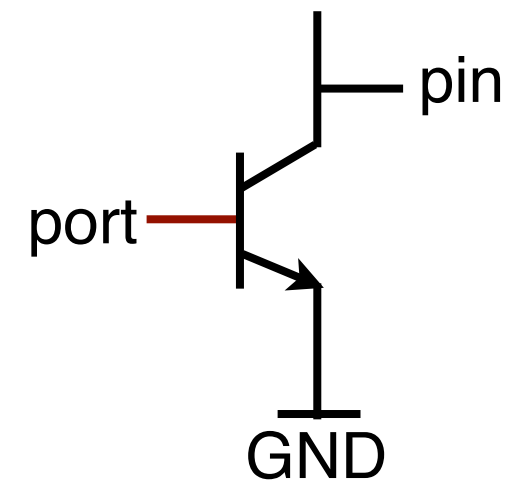
Bipolar transistor		MOS transistor	
collector		drain	
base		gate	
emitter		source	



# Open collector / Open drain output

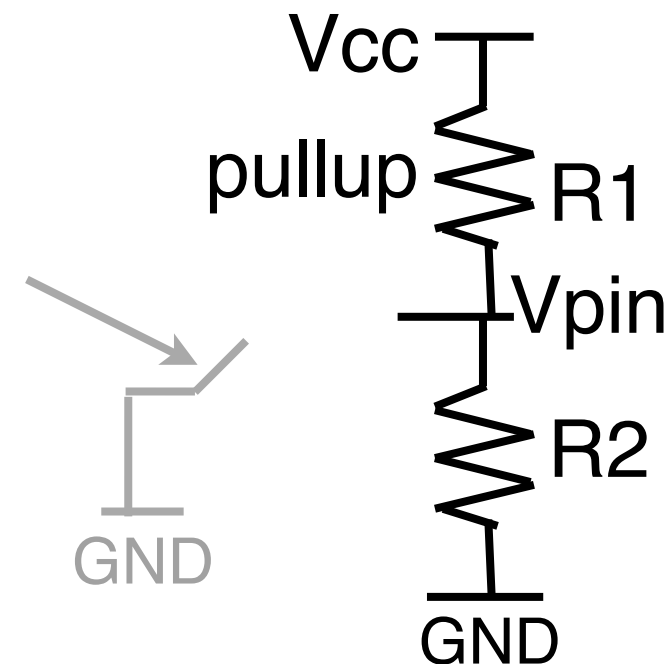
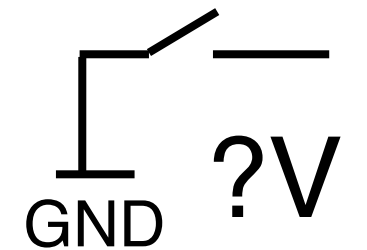
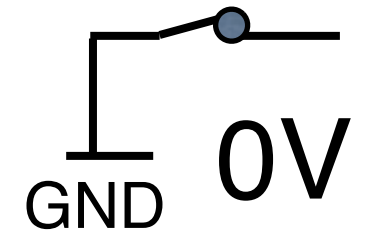
- Port = 0  
=> pin connect to ground (0V)
- Port = 1  
=> pin is disconnected
- Open-collector  
=> pin is in high impedance ("Z")  
or "tristated"

collector  
(open = disconnected)



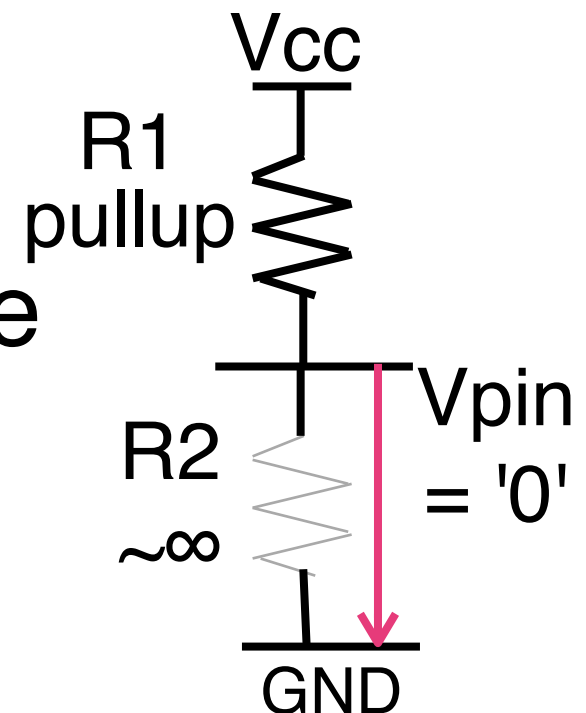
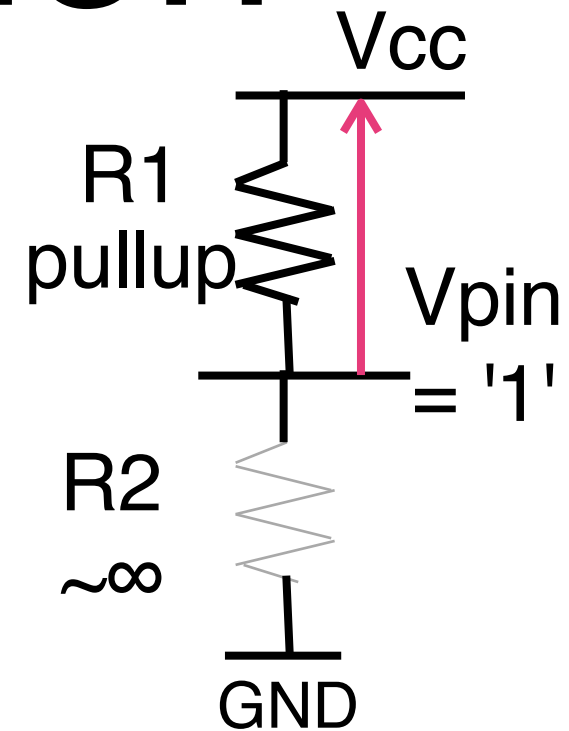
# Closed Collector vs. Open Collector

- Closed-collector means pull-up  
=> pin gets pulled up high
- Voltage divider
  - $V_{pin} = V_{cc} * R2 / (R1 + R2)$
  - When  $R2 \gg R1$ ,  $V_{pin} \sim V_{cc}$   
When  $R2 = 0$ ,  $V_{pin} \sim 0$

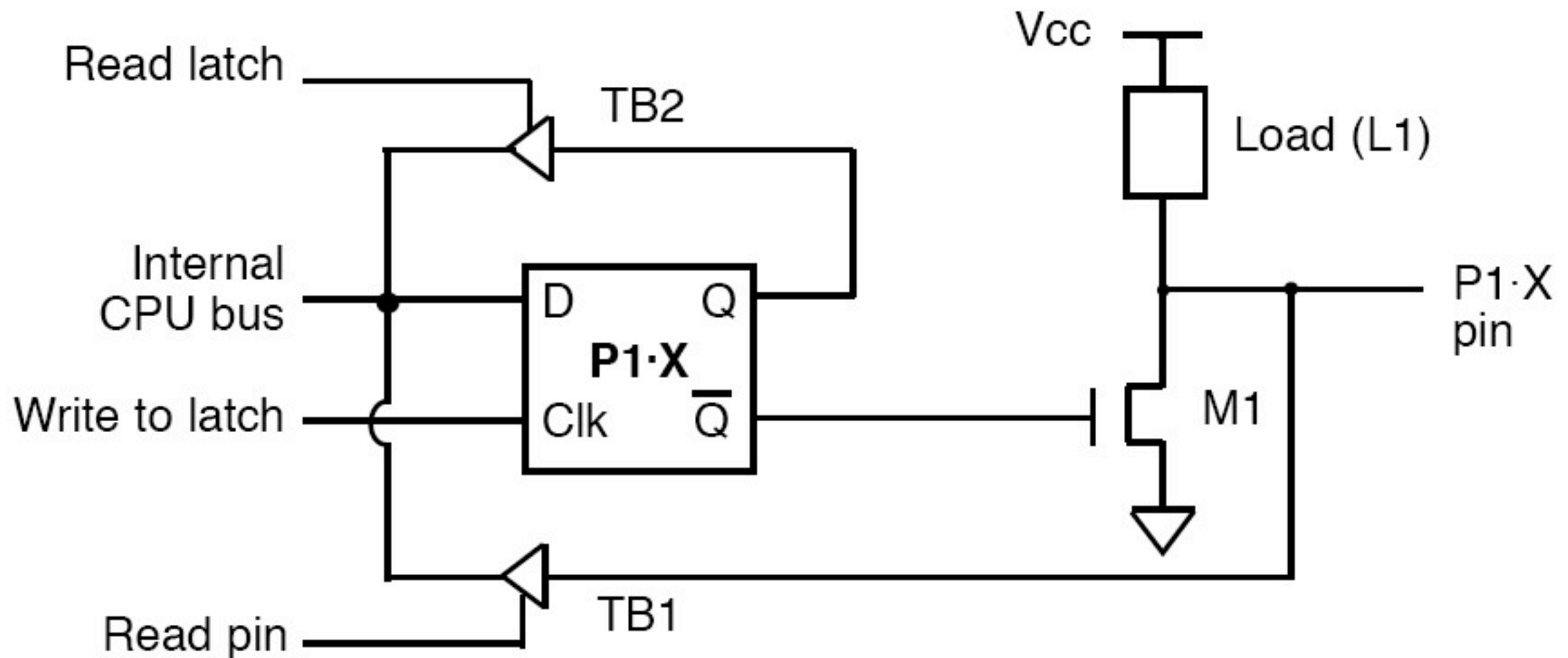


# Input value resolution

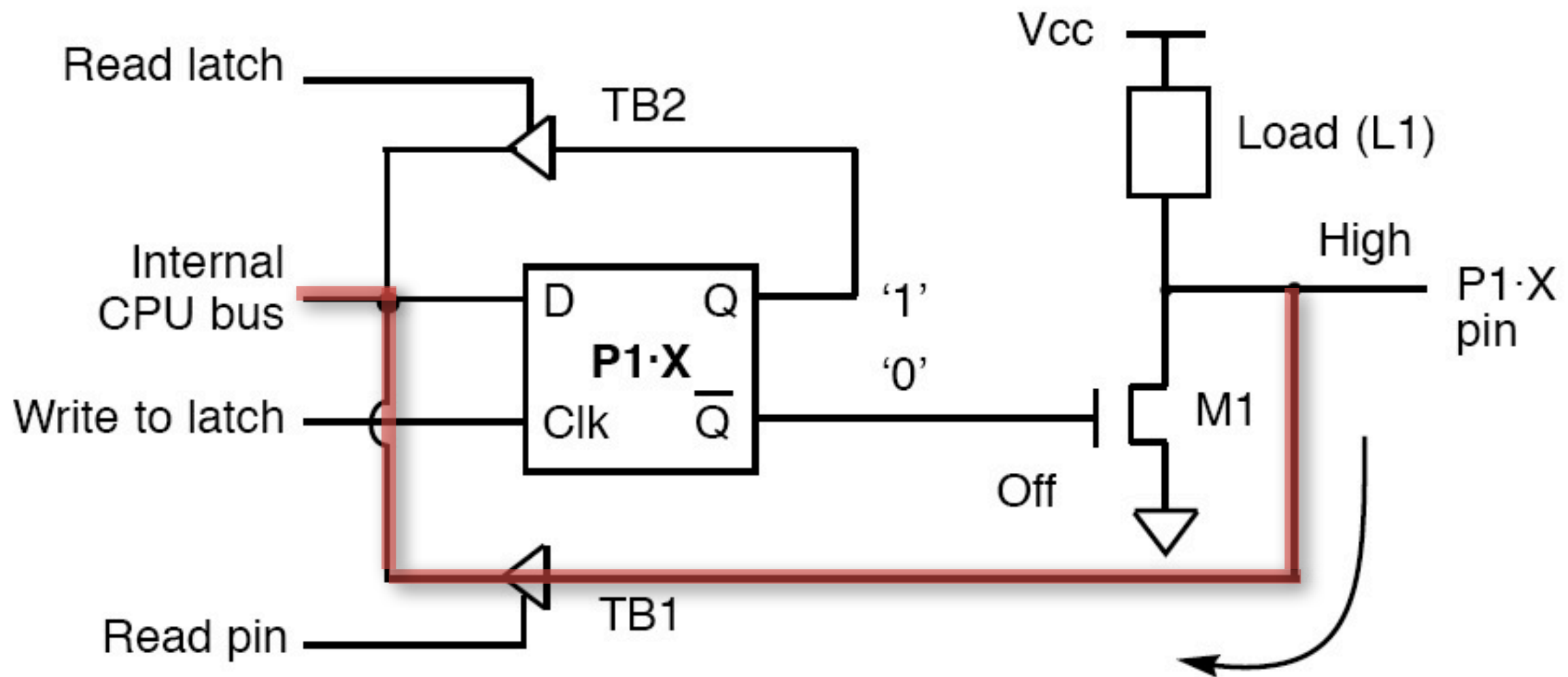
- Input pin = 1
  - No conflict (latch also pulled up)  
=> read value = 1
- Input pin = 0
  - register: pull-up to R1
  - Vpin is pulled low by outside device
  - R1 (10 K $\Omega$ ) draws current  
a price to pay, but logic works



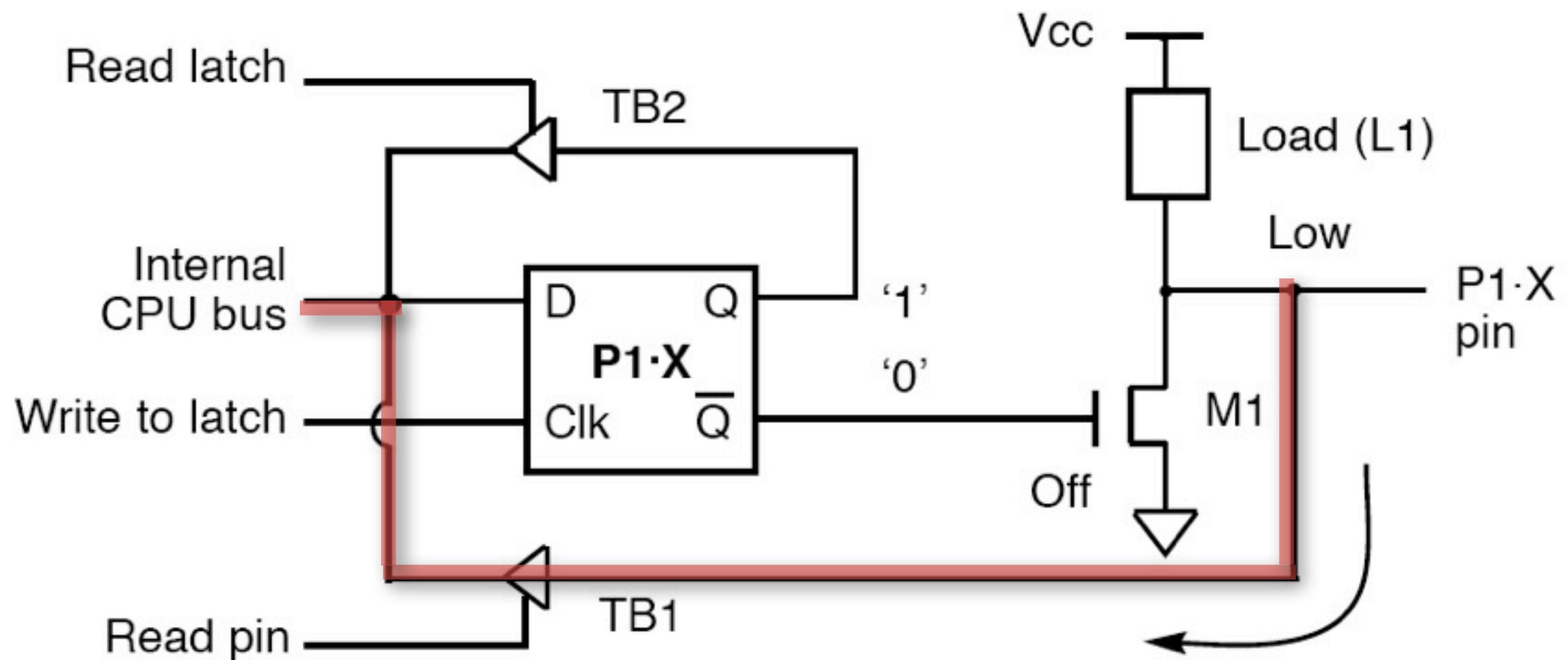
# Structure of Port P1



# Reading a High value at input

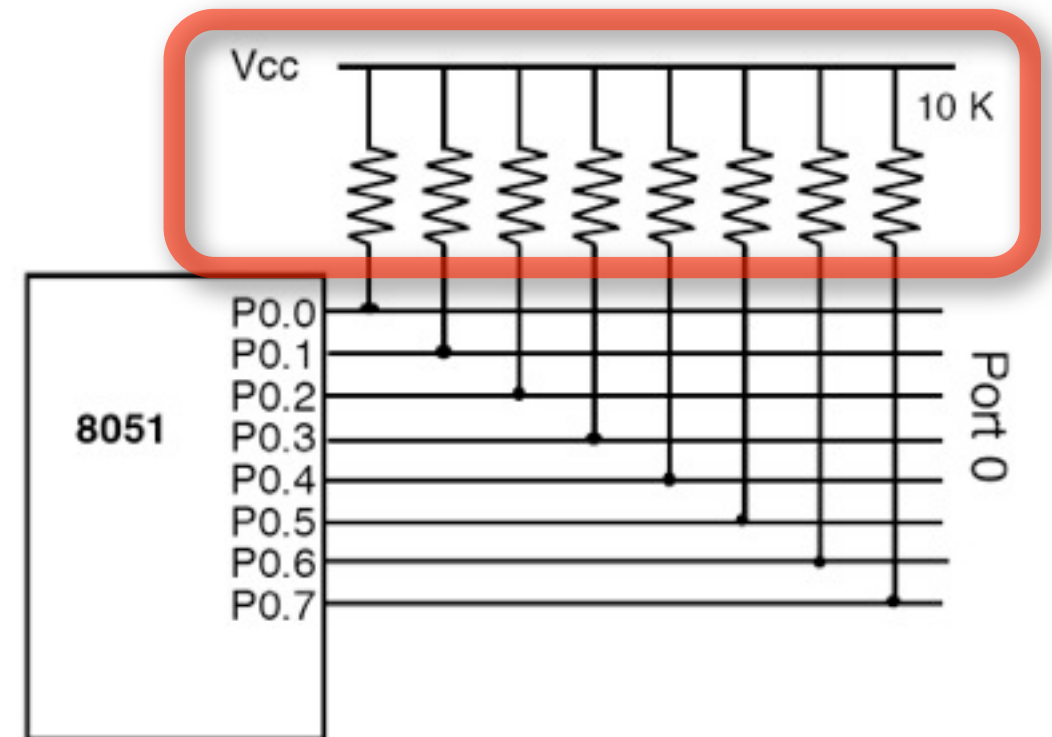


# Reading a Low value at Input pin



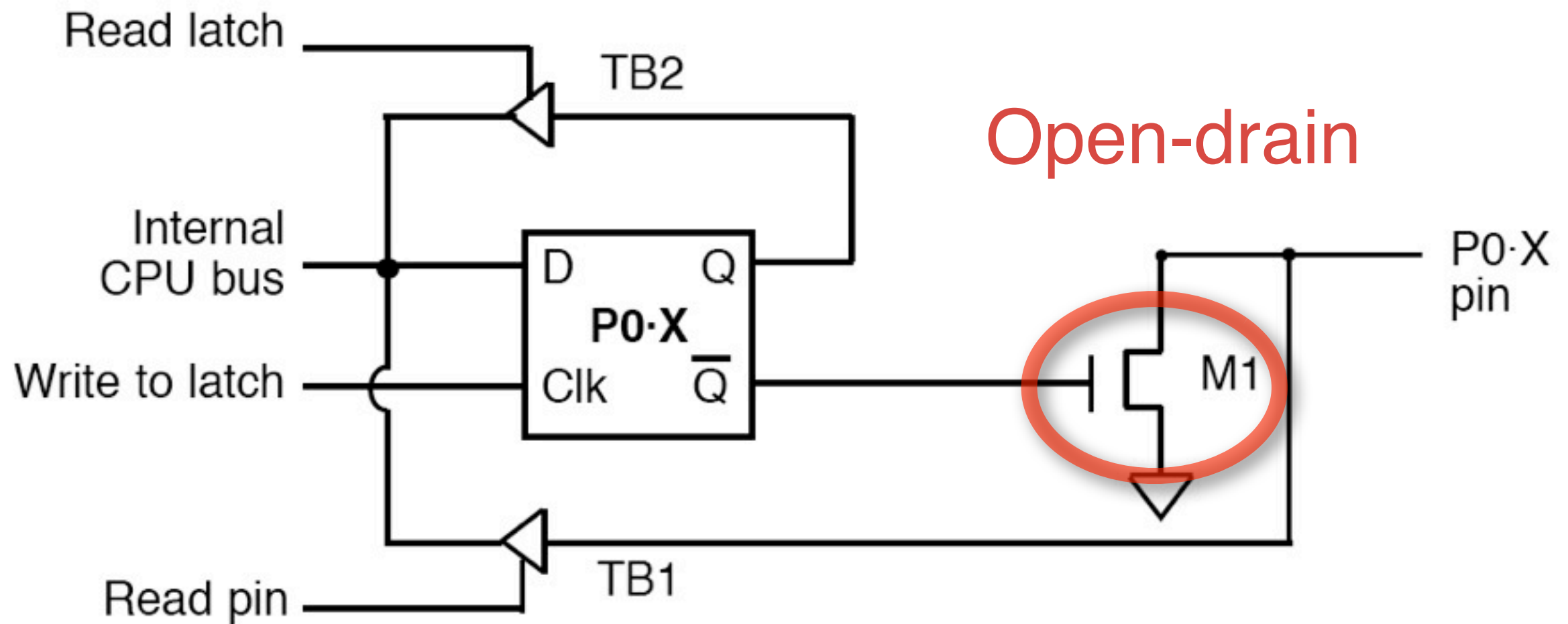
# Port P0 is different from other I/O ports

- No internal pull-up
- OK as input
- Output -- watch out!
  - 0 is a real, hard zero
  - 1 is a fake 1 (disconnect!)
- for output to 1
  - Add external pull-up resistors



external  
pullup

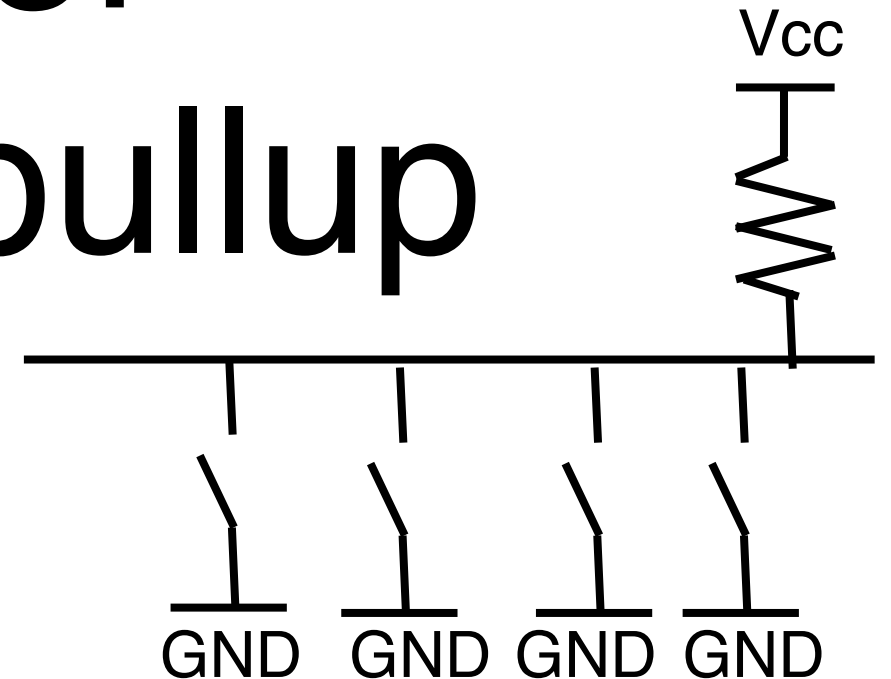
# Structure of Port P0





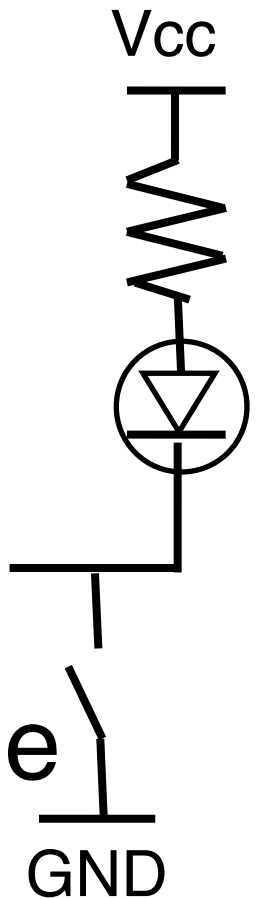
# Reasons for externalizing pullup

- Bus connection
  - Multiple microcontrollers
  - bus pull-up, each MCU open collector
  - "wired-AND" -- any zero pulls down bus
- Why not use built-in pull-up?
  - Could be too many pull-ups!
  - Pull-up too strong => hard to get clean 0



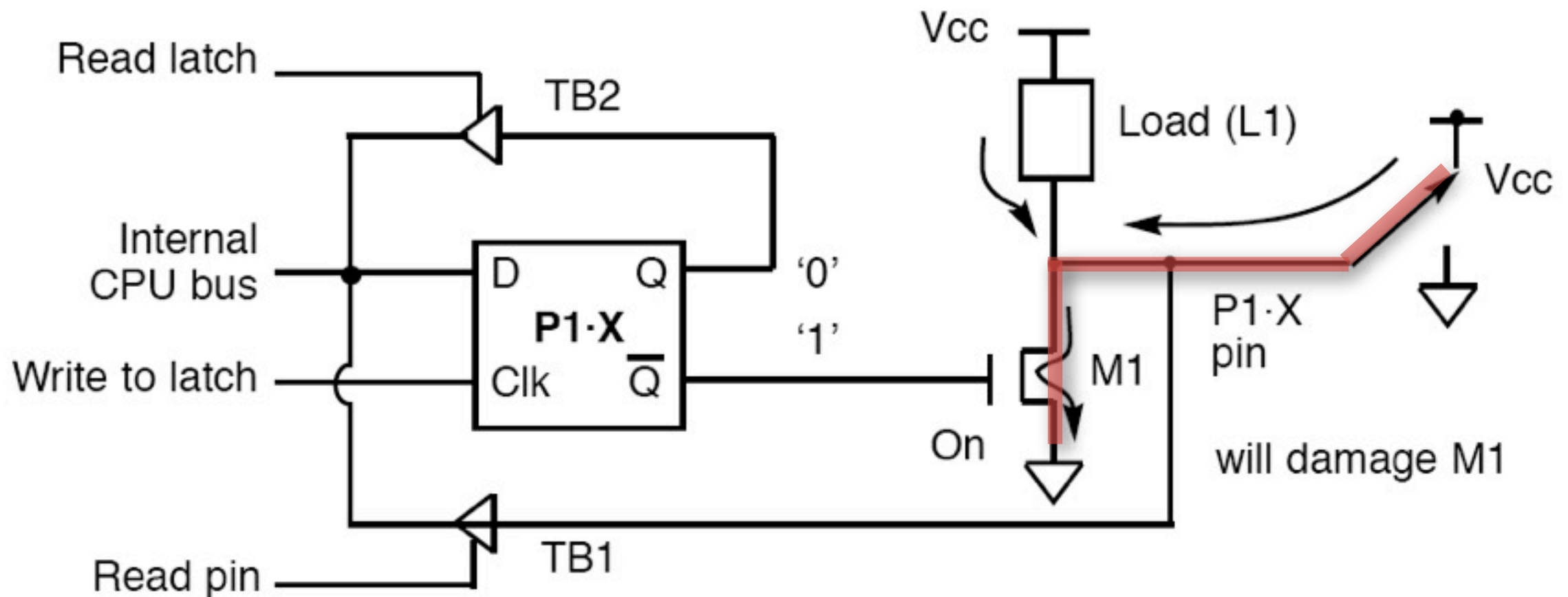
# Example use of output with external pullup

- E.g., an LED
  - Turns on when voltage difference  $> .7V$
  - I/O port voltage may be too high!
  - Voltage divider  $\Rightarrow$  better matches voltage
- External pull-up voltage may be higher than MCU's output voltage
- Easier to "sink" (consume) current than to "source" (provide) current



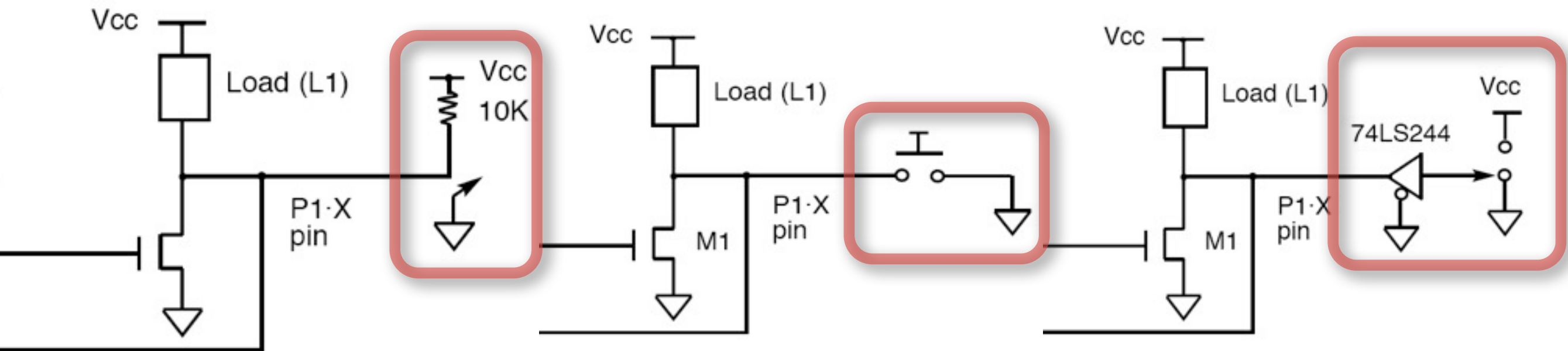
# What if port latch bit = 0 during input?

- Could potentially short circuit!
  - If input wire is tied to Vcc
- Solution: Always use pull-up



# Three ways to avoid short circuit in input

- Input switch with resistive pull-up
- Input switch to ground
- Input switch with a tristate

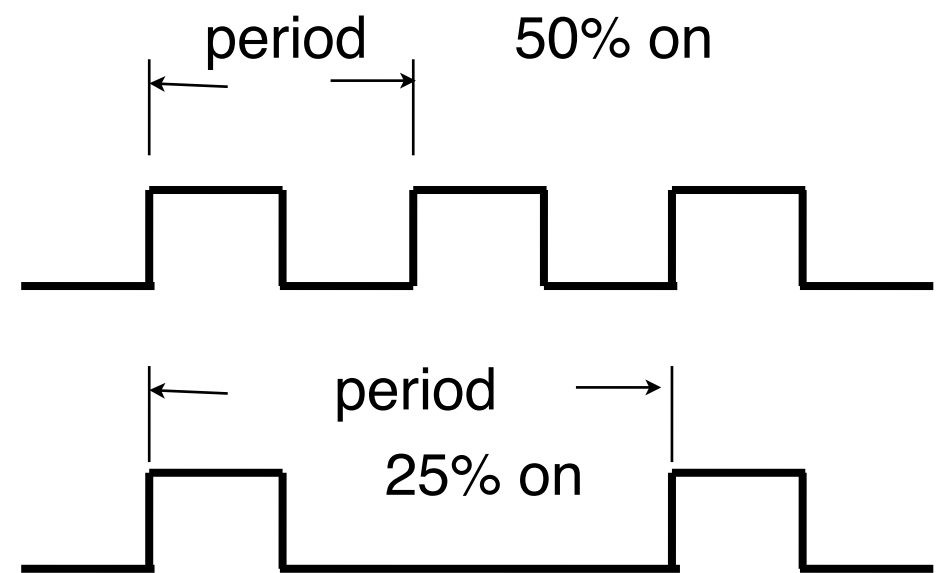


# I/O bit manipulation

- Use bit instructions
  - `CPL P1.2` ;; complement port 1 bit 2
  - `SETB P2.1` ;; assign port 2 bit 1 = 1
  - `CLR P3.7` ;; assign port 3 bit 7 = 0
- Not all MCU instruction sets support bit op
  - e.g., ATMEEL AVR: access whole 8-bit port
  - `P2 |= 0x02; /* set P2.1 */`
  - `P3 &= 0x7F; /* clear P3.7 */`
  - How to complement P1.2?

# Duty cycle

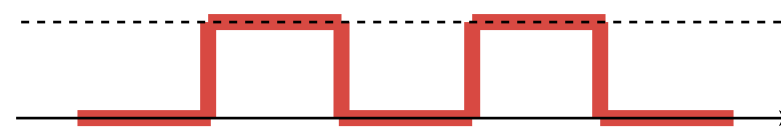
- Percentage on time in a square wave
  - 50% duty cycle  
=> 50% on, 50% off
  - 25% duty cycle  
=> 25% on, 75% off
- Orthogonal to frequency
  - each frequency can have diff. duty cycles



# Application of duty cycling

- PWM: Pulse width modulation
  - Use duty cycle to control average intensity
  - Frequency should be sufficiently high

- Example



50%  
duty cycle

- motor speed
- brightness of light



@hi freq.  
effectively  
looks like 50%  
amplitude (avg)

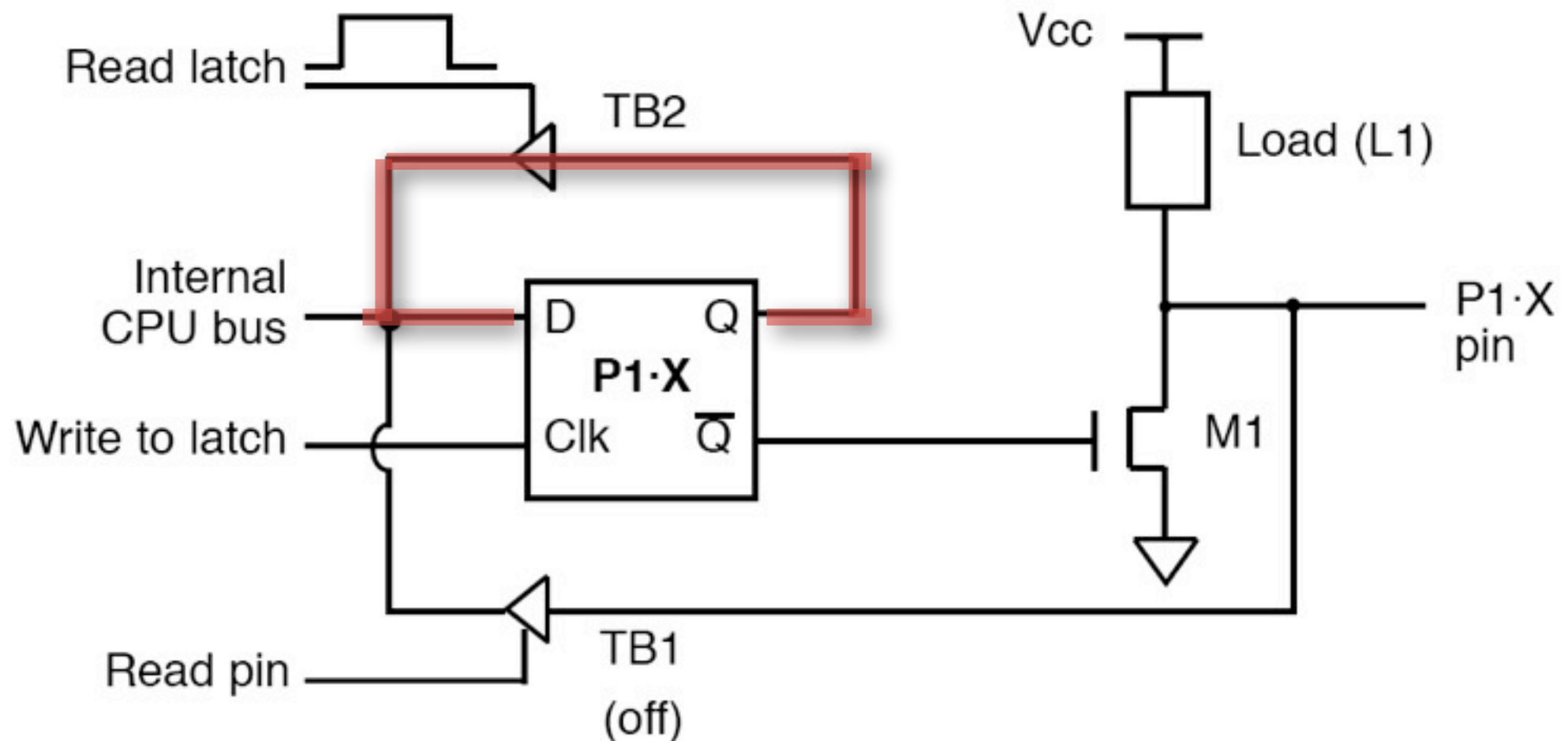
# Reading from pin vs. using port latch value

- **MOV** instructions (source is a port)
  - reading from pin. e.g., **MOV** A, P1
- **ANL** (And), **ORL** (Or), **XRL** (Xor),  
**JBC**, **DJNZ**, **CPL** (complement), **INC**, **DEC**
- Port latch serves as both a src and dest  
e.g., **ANL** P1, A means  $P1 = P1 \& A$   
**INC** P2 means  $P2++$



# Uses port-latch value instead of pin

ANL	<i>port, A</i>	CPL	<i>bit</i>
ORL	<i>port, A</i>	INC	<i>port</i>
XRL	<i>port, A</i>	DEC	<i>port</i>
JBC	<i>bit, target</i>	DJNZ	<i>port, target</i>



# Meaning of reading port latch

- `ANL P1, A`      `:: P1 := P1 & A`
- ``Read-Modify-Write" property
  - Read a port latch, modify the value, write back to the same port latch
  - All done "atomically" in one instruction
- Why is this a useful feature?

# Atomic operation

- w/out read-modify-right, need a sequence of 3 instructions:
  - `MOV A, P1`  
`ANL A, expression`  
`MOV P1, A`
- But! If an interrupt occurs, value of A may be changed by the interrupt handler!
- Atomic means you can be sure operation is consistent even if interrupted

# Addition revisited

- Depends on interpretation of the 8-bit data
  - as signed
  - as unsigned
  - as binary-coded decimals (BCD)
- Results mostly same for signed /unsigned
  - overflow interpretation is different
  - BCD needs to us Aux Carry, DA

# Unsigned byte addition with 2-byte sum

- after add, CY bit could be set (carry)
- use JNC to skip incrementing higher byte

	MOV	R0,#40H	:: p = starting addr
	MOV	R2, #5	:: 5 times
	CLR	A	:: lower order sum
	MOV	R7, A	:: higher order
Again:	ADD	A, @R0	:: sum := sum + *p
	JNC	Next	:: no carry, don't inc higher byte
	INC	R7	:: higher order ++
Next:	INC	R0	:: ++ pointer to next element
	DJNZ	R2, Again	:: while (--count);

# ADDC: add with carry

- Syntax: `ADDC A, [imm,dir,reg,@R]`
- Meaning:  $A = A + \text{arg} + \text{CY}$
- Usage:
  - add two lower byte and set CY
  - `ADDC` two higher bytes
- uses multiple bytes to represent a number

# BCD: binary coded decimal

- Represents decimal digits
  - Minimally: 4 bits (0000 to 1001 binary)
  - Unused combinations: 1010 to 1111 bin
- Examples

decimal	BCD high	BCD low	binary
17	1H (0001)	7H (0111)	11H
28	2H	8H	1CH
45	4H	5H	2DH

# Unpacked vs Packed BCD

- Unpacked: one byte per decimal digit
- Packed: one byte per TWO decimal digits
- Examples

decimal	unpacked two bytes		packed byte
17	01H	07H	17H
28	02H	08H	28H
45	04H	05H	45H



# AC: auxiliary carry flag

- Used for packed BCD arithmetic
  - $\text{CarryOut}(3) = \text{CarryIn}(4)$
  - i.e. from lower nibble to higher nibble
- E.g., 29H + 18H as BCD add

	AC=1	
29H	2H	9H
+) 18H	1H	8H
sum=	4H	1H

# DA instruction: decimal adjust (for addition)

- Assume: packed BCD
- works only after **ADD**; not after **INC**!
- postprocess to adjust sum to packed BCD
  - **if** sum bits[3:0] > 9 **or** AC=1 **then**  
sum bits[3:0] += 6
  - **if** sum bits[7:4] > 9 **or** CY = 1 **then**  
sum[7:4] += 6

# Example

- 29H + 18H as BCD add

	BCD	[7:4]	[3:0]
	29H	2H	9H
ADD	18H	1H	8H
sum		3H	1H
aux carry		AC=1	
DA		+1	+6
adjusted		4H	7H

# Checksum example

- What is checksum [of a series of bytes]
  - Add bytes together and discard CY
  - Take 2's complement of the sum byte
- Purpose: Ensure data is not corrupted
- How: add all bytes & checksum, should = 0  
(after discarding CY)

# Example for Checksum

- Data: [25H, 62H, 3FH, 52H]
- $\text{sum} \% 256 = 18\text{H}$ ,  
checksum = 2's complement = E8H
- To check:  
 $(\text{sum} + \text{E8H}) \% 256 = 0$   
 $\Rightarrow$  data is (probably not) corrupted
- if  $((\text{sum} + \text{CS}) \% 256 \neq 0)$   
 $\Rightarrow$  data *must be* corrupted

# Checksum program

- What is needed: a sum() routine:  
`char Sum(char array[ ], char length);`
- `Checksum = -Sum(array, n);`
- if you put Checksum at end of dataArray, then to check it, evaluate the condition  
`Error = Sum(array, n+1);`  
`/* 0 means no-error, 1 means error */`
- Issues: assume array is writable, in on-chip RAM

# Sum(char data[], char n)

## Assume R1=ptr, R2=n

Sum:	CLR	A	:: sum = 0;
H2:	ADD	A, @R1	:: sum += *dataptr;
	INC	R1	:: dataptr++;
	DJNZ	R2, H2	:: loop till n=0
	RET		

# Example calling Sum()

array	EQU	30H	:: on-chip address
COUNT	EQU	4	:: n (constant)
DATA	EQU	400H	:: addr. in code mem
	MOV	R1, #array	:: load pointer
	MOV	R2, #COUNT	:: load n
	ACALL	Sum	:: call

- On return from Sum(&array, COUNT), the accumulator contains the sum



# CAL\_CHECKSUM

- call the Sum() subroutine!

CAL\_CHECKSUM:

MOV	R1, #array	:: load array address
MOV	R2, #COUNT	:: load n
ACALL	Sum	:: call subroutine for sum
CPL	A	:: calculate 2's complement
INC	A	::
MOV	@R1, A	:: put at end of array

# TEST\_CHECKSUM

- Behavior: send 'B' for bad, 'G' for good to P1 to indicate if checksum is bad/good

TEST\_CHECKSUM:

```
MOV      R1, #array      ;; load pointer
MOV      R2, #COUNT+1   ;; load n+1
ACALL    Sum              ;; call Sum()
JZ       G1              ;; if A == 0: good
MOV      P1, #'B'        ;; else bad
RET
```

```
G1: MOV   P1, #'G'        ;; the good part
RET
```