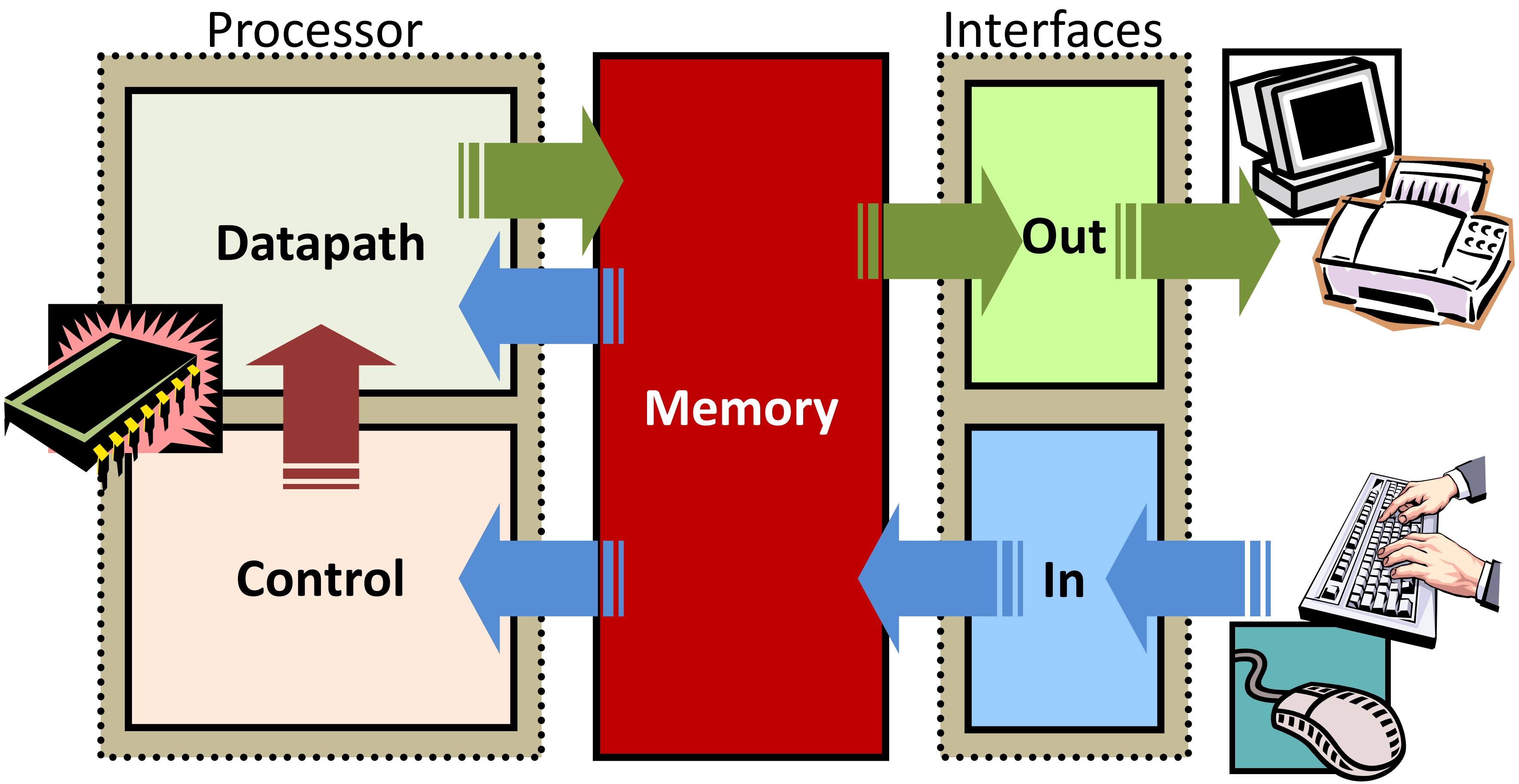
CS-200
Computer Architecture
—
Part 2b. Processor, I/Os, and Exceptions
Inputs and Outputs

Paolo Ienne
<paolo.ienne@epfl.ch>

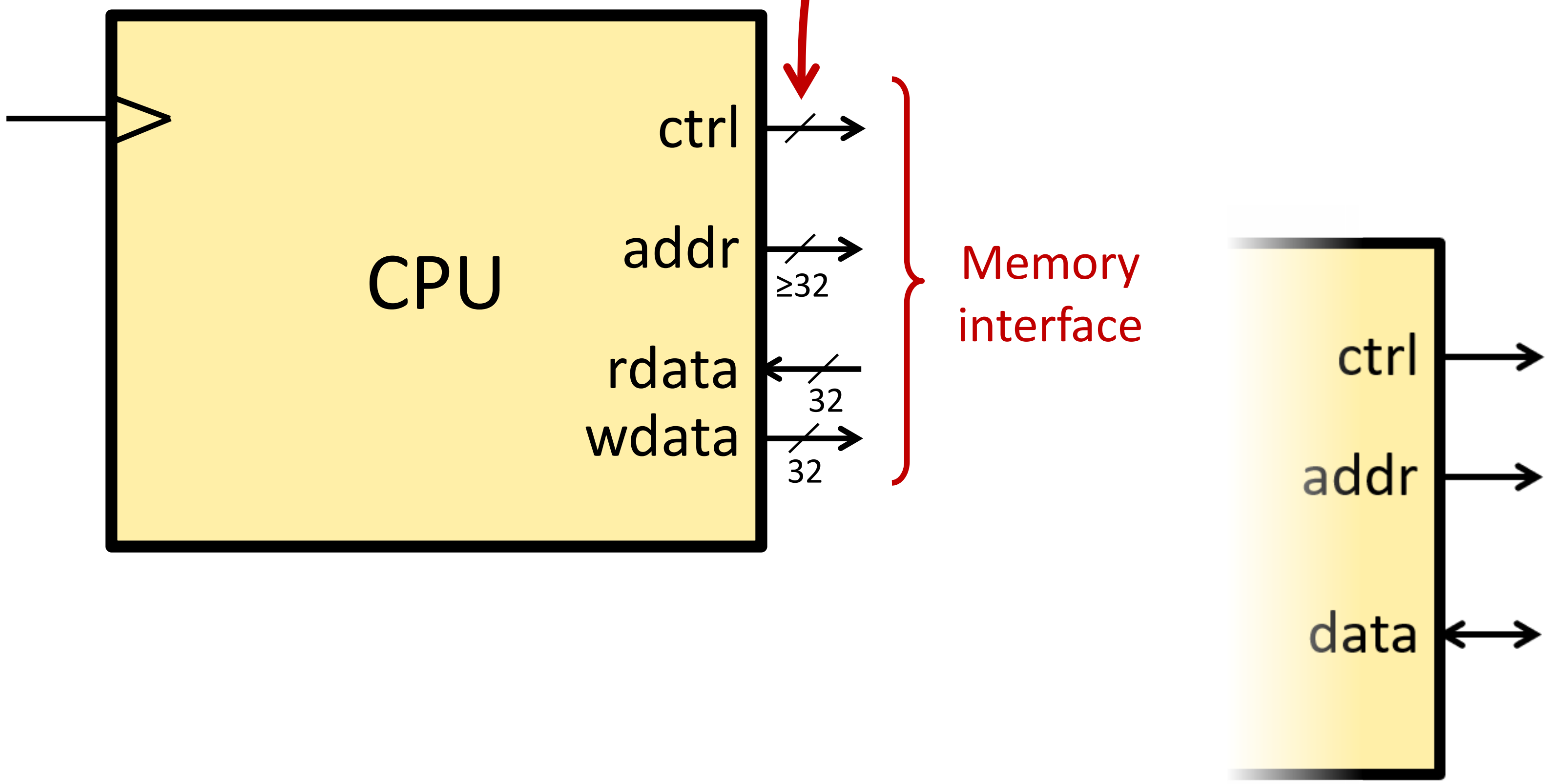
The Five Classic Components of a Computer



The CPU

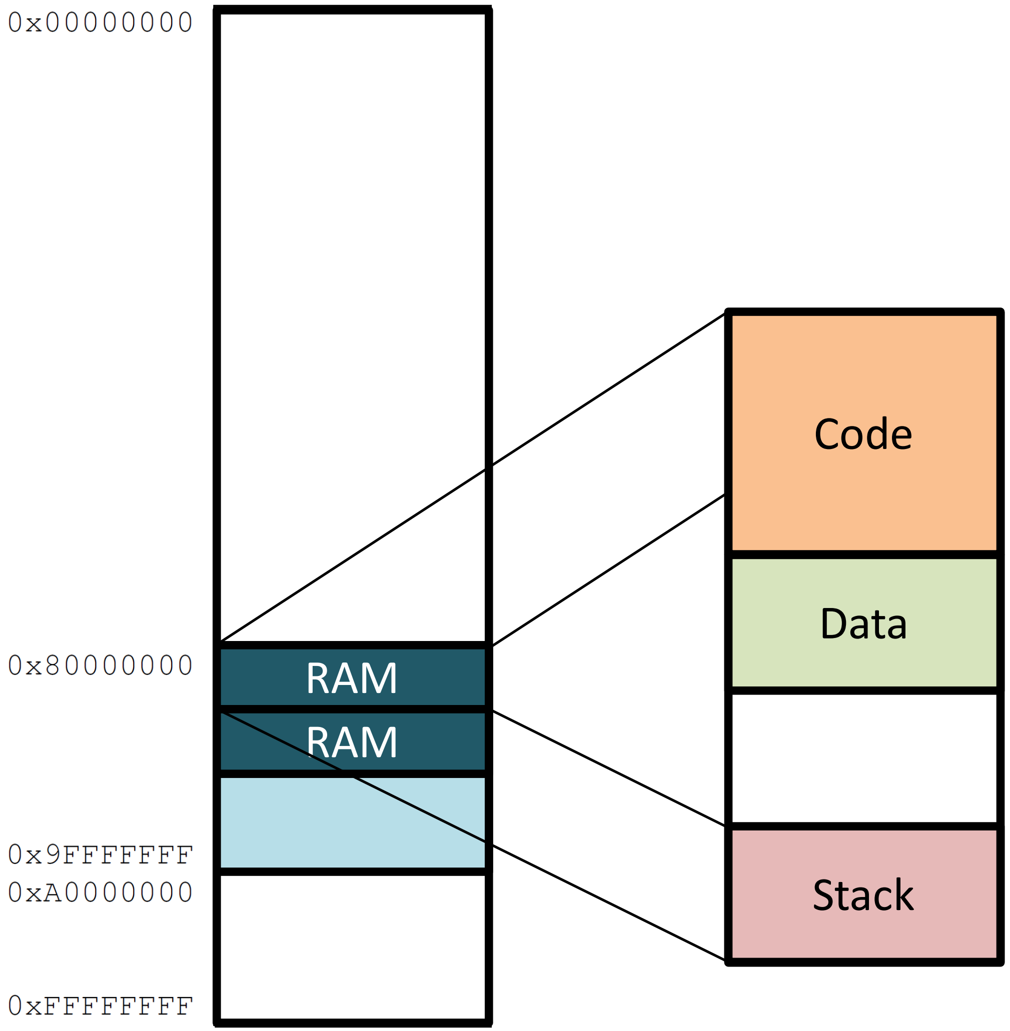
A “very” **sequential** component
(but from now on we may
omit the **clock** in diagrams)

Signals like
CE = Circuit Enable = address is valid
WE = Write Enable = access is a store

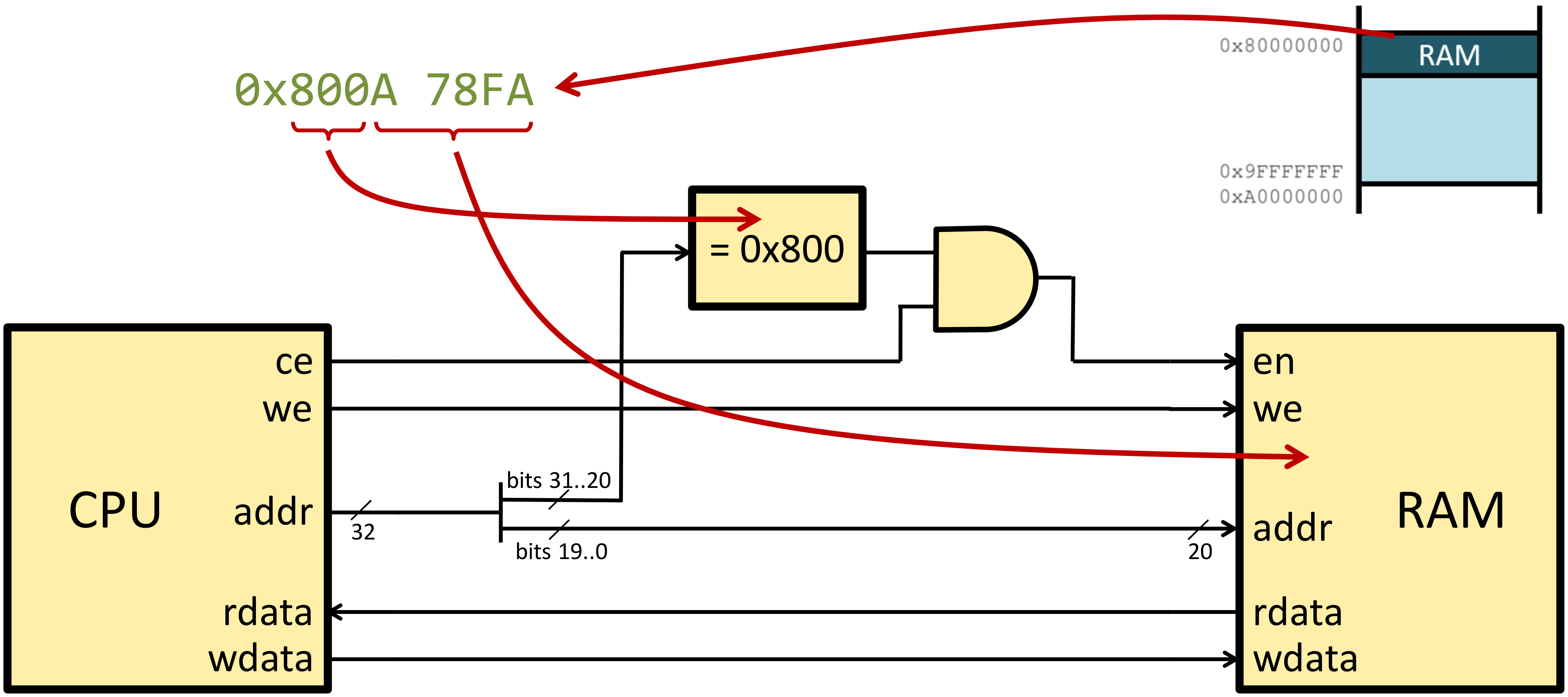


Single data bus
for read and write
(needs **tristate**!)

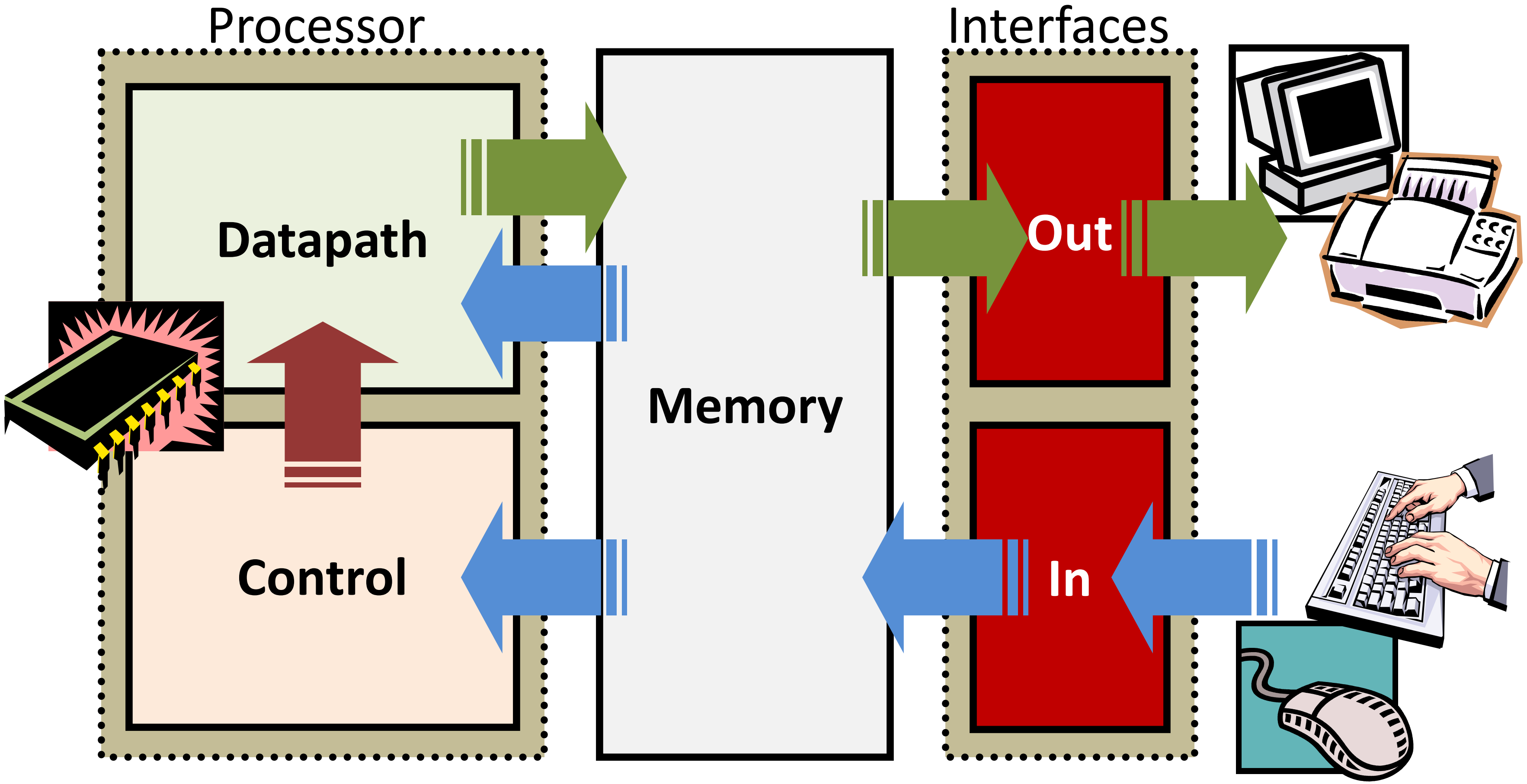
Physical Memory Map



Connecting CPU and Memory



The Five Classic Components of a Computer

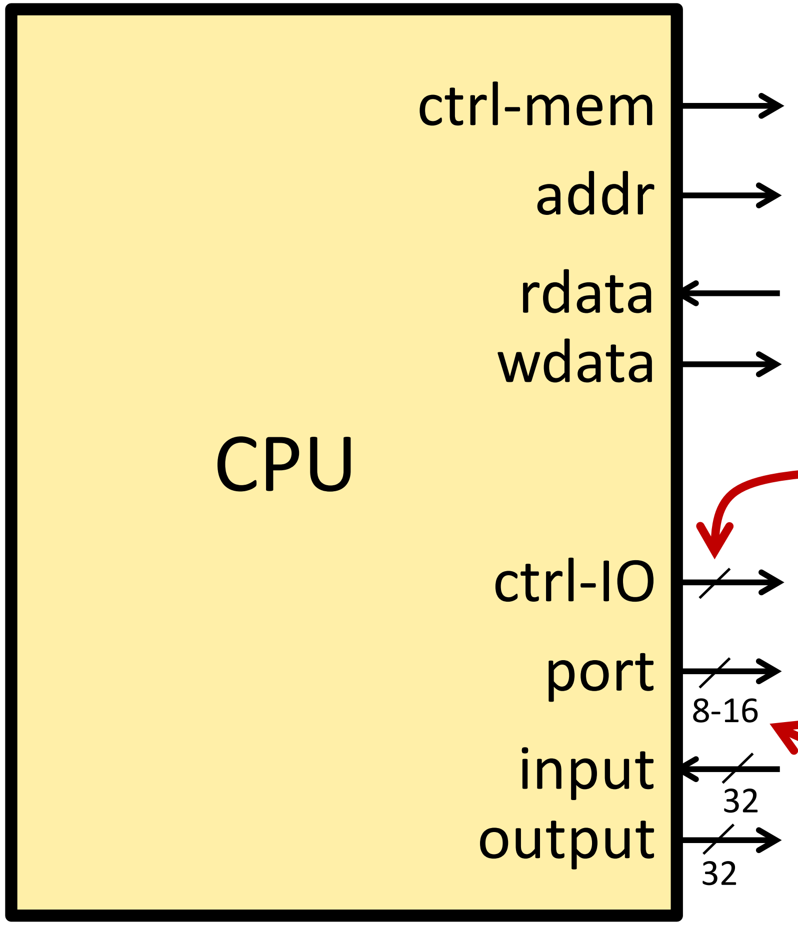


Input/Output Devices (I/Os)

Type	Peripheral	Data Rate
Human Interaction	Keyboard	~kbps
Human Interaction	Mouse	~kbps
Generic	Serial Port (RS-232)	115.2 kbps (max)
Generic	Parallel Port (LPT)	150 kbps
Generic	USB 4.0	20-40 Gbps
Generic	Bluetooth 5.0	2 Mbps
Generic	PCIe 4.0	16 Gbps per lane
Storage	SATA III (HDD/SSD)	6.0 Gbps
Storage	NVMe (PCIe 4.0)	64 Gbps (4-lane)
Networking	Ethernet (10BASE-T)	10 Mbps
Networking	10 Gigabit Ethernet (10GBASE-T)	10 Gbps
Networking	Wi-Fi 6 (802.11ax)	Up to 9.6 Gbps
Displays	VGA (analog video)	0.6-1.5 Gbps (approx.)
Displays	HDMI 2.1	48 Gbps
Optical Discs	CD-ROM	150 KB/s (1x) - 7.68 MB/s (52x)
Optical Discs	DVD-ROM	1.32 MB/s (1x) - 21.1 MB/s (16x)
Optical Discs	Blu-ray	4.5 MB/s (1x) - 54 MB/s (12x)

Accessing I/Os: Port Mapped I/O (PMIO)

- Create a **new interface** similar to the memory one



New instructions (e.g., **x86** but seldom used):

```
IN register, port
OUT port, register
```

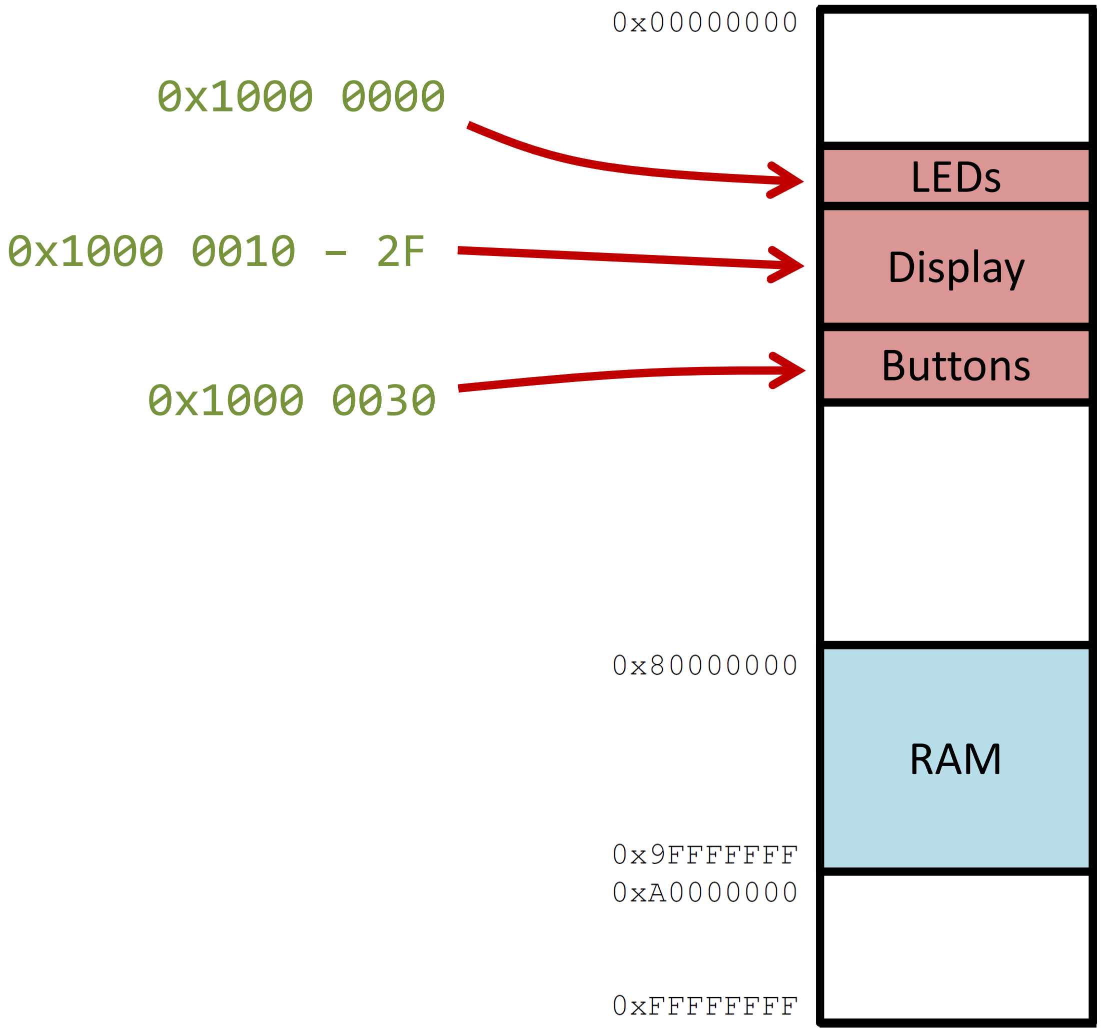
e.g., `IN AL, keyboard`

CE = Circuit Enable = port number is valid

OE = Output Enable = I/O access is an output

Maybe not 32 bits for we have only a few peripherals…

Accessing I/Os: Memory Mapped I/O (MMIO)

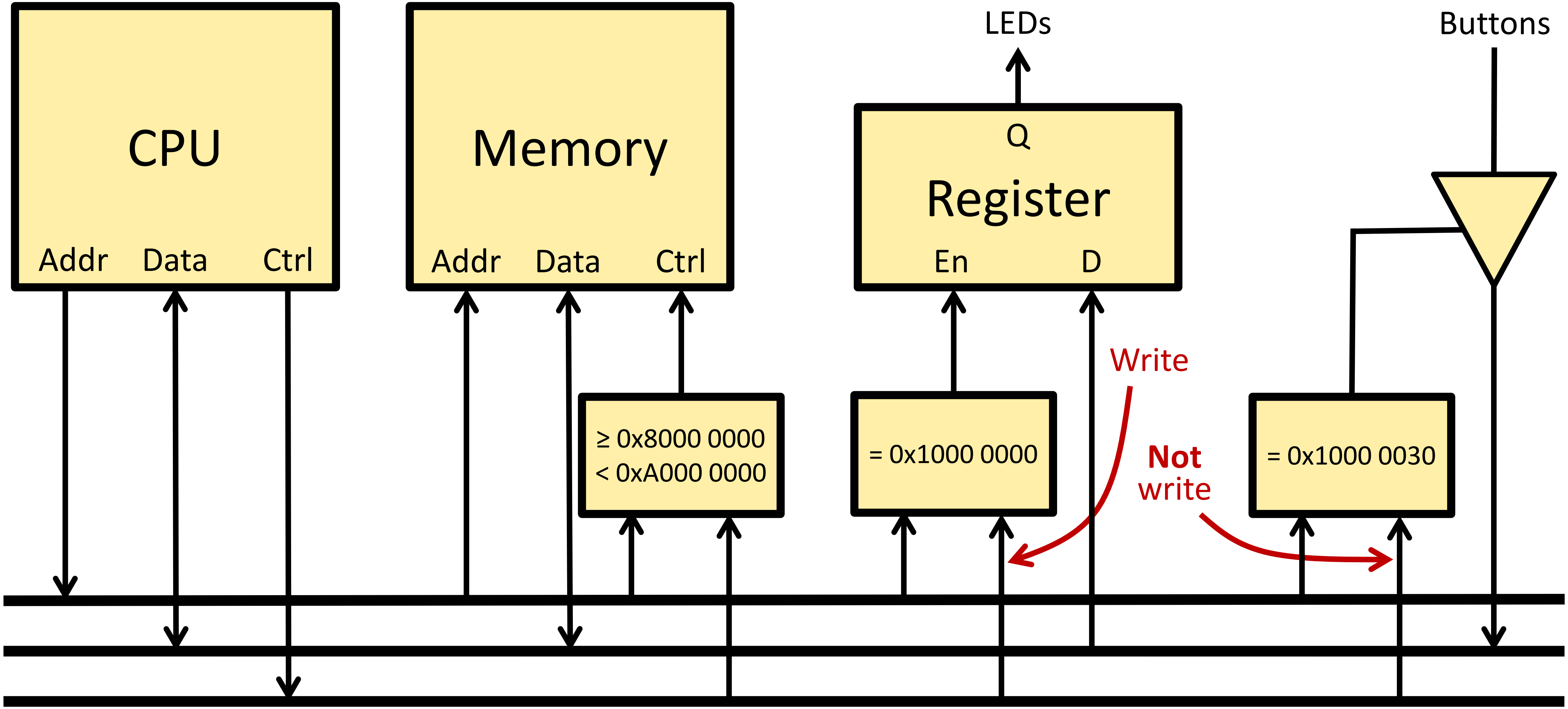


No special hardware needed in the CPU

No special instructions needed:

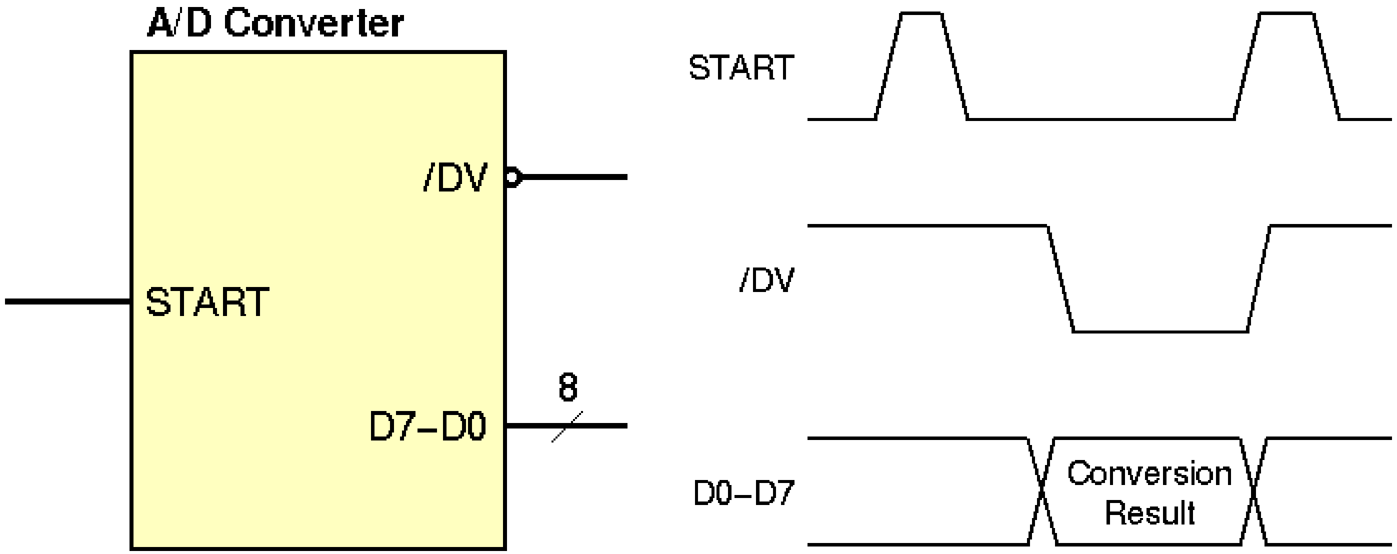
```
lui   t0, 0x10000   # pointer to I/Os
sw    t1, 0(t0)     # write LEDs
lw    t2, 0x30(t0) # read buttons
```

Accessing I/Os: Memory Mapped I/O (MMIO)



Example:
A/D Converter

- Signals:
  - **Start** (START): input; when active begins a new conversion
  - **Data Valid** (/DV): output; when active, D7—D0 are valid
  - **Data** (D7—D0): output; last conversion result



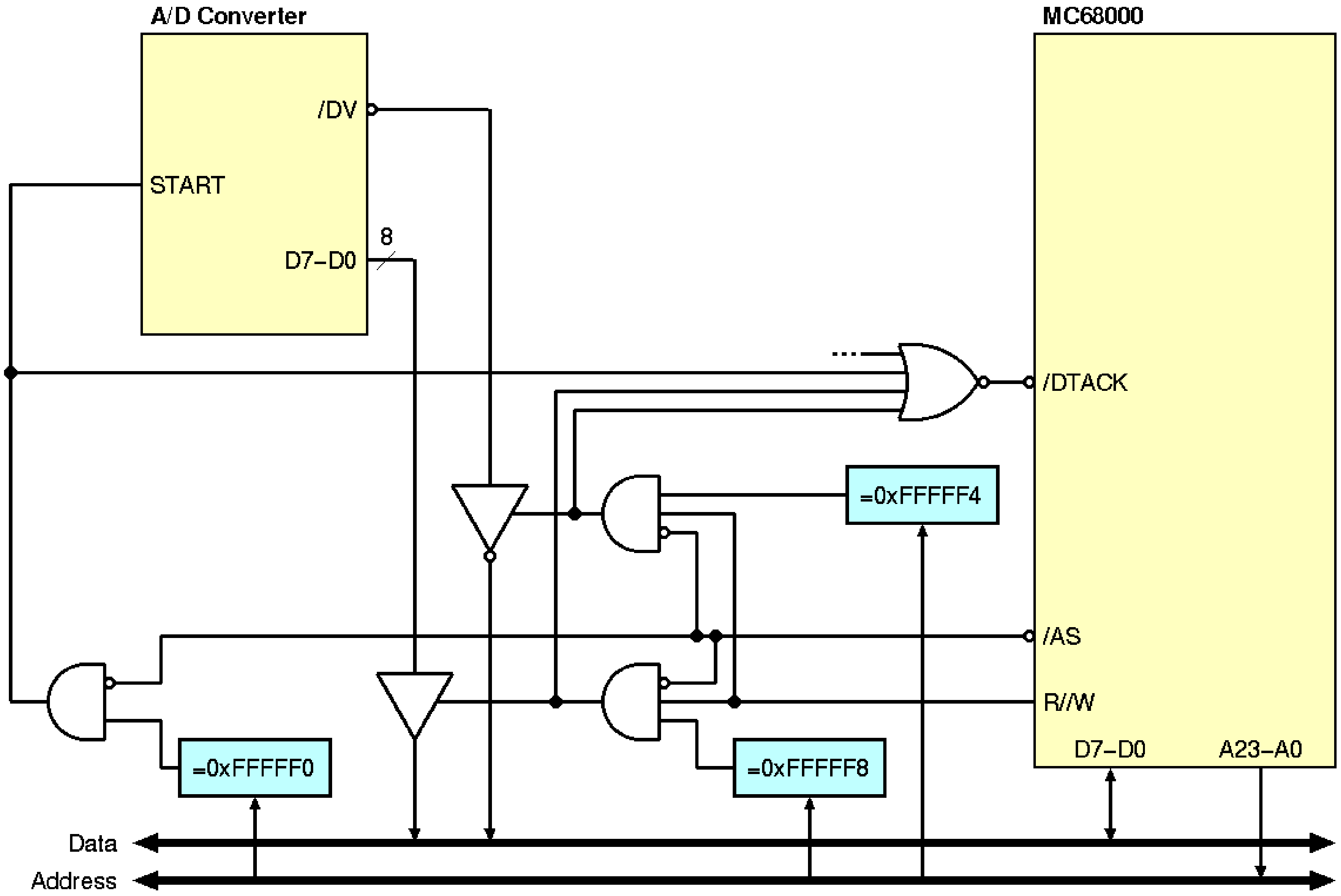
Example:
Simple Bus Interface

- Suppose that a 8-bit processor has the following signals:
  - **Address** (A23—A0): output; address bus
  - **Data** (D7—D0): input/output; data bus
  - **Address Strobe** (/AS): output; signals the presence of a valid address on the Address bus during a memory access cycle
  - **Read/Write** (R//W): output; signal the direction of the data flow
  - **Data Acknowledge** (/DTACK): input; must be activated at the end of a memory access, when the written data have been latched or the read data are ready
- Similar but not identical to the MC68000
- Just an example but already more complex than busses described so far (/DTACK)

Example:
Memory-Mapped Interface

- Connect the A/D converter described in the previous slide so that:
  - **Any access** (R or W) to address `0xFFFFF0` starts a **new conversion**
  - The **Data Valid** signal can be read by the processor at address `0xFFFFF4` (bit 0)
  - The **result of the conversion** can be read by the processor at address `0xFFFFF8`

A/D Converter: Circuit

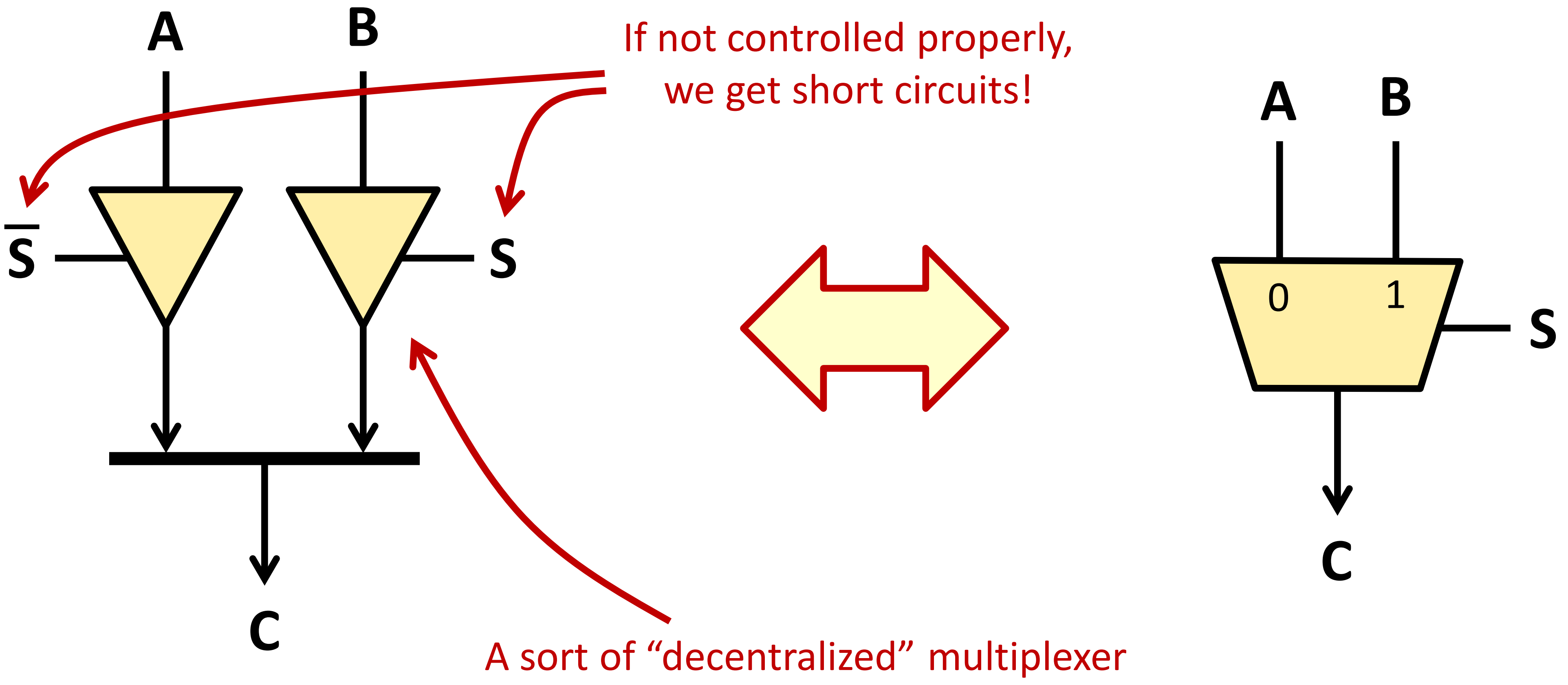


A/D Converter: Software

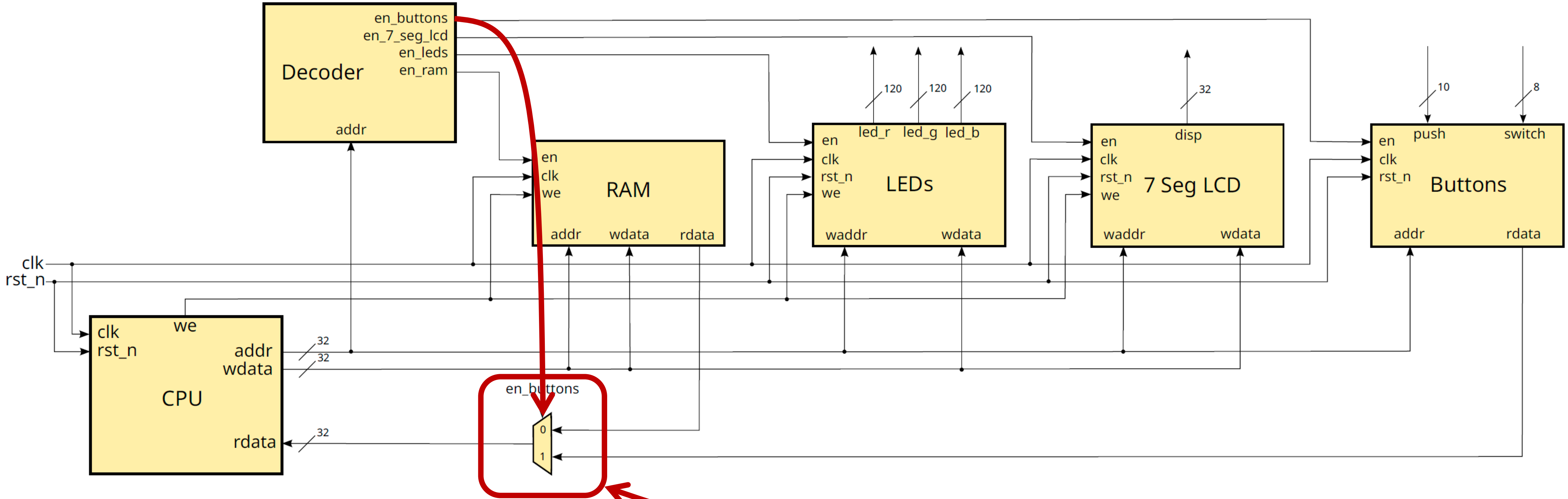
```
read_adc:   lui    t1, 0xfff
            addi   t1, t1, 0xff0          # t1 = 0xffffff0
            sw     zero, 0(t1)            # start conversion

poll:       lw     t0, 4(t1)              # t0 = DV signal
            beqz   t0, poll               # wait until done

end:        lw     a0, 8($t1)             # a0 = A/D output
            ret
```

What Do These Tristate Buffers Do?

- What is their **logic function**?



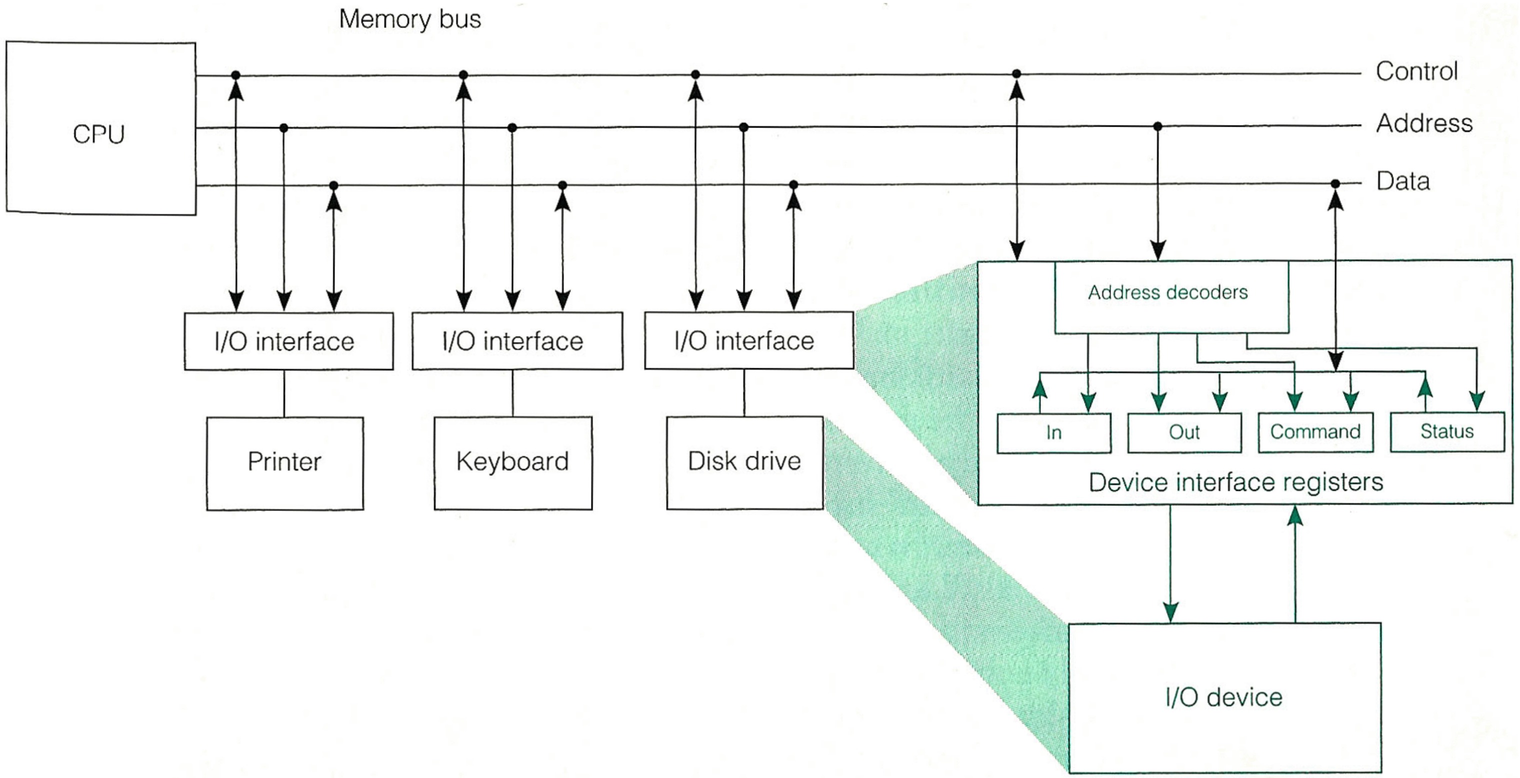
Your System in Lab B



Exactly the same function as the tristate buffers in the previous example

Programmed I/Os

Many peripherals are more developed programmable systems and have a set of registers which the processor reads and writes (a) to **send** and **receive data** and (b) to **issue commands** and **read the status**



Source: Heuring & Jordan, © Addison Wesley 1997

A Classic UART

- **UART** = Universal Asynchronous Receiver-Transmitter
- One of the **simplest and most common communication** peripherals, typically used today to connect terminals to embedded devices
- Our UART has a **simple programmed I/O interface** with four registers:
  - A **control register** for the processor to configure the UART
    - Bit 7 must be set to 1 for the UART to be enabled
    - Bits 2..0 configure the communication speed (e.g., `0b001` for 9600 baud)
  - A **status register** for the processor to check the status of the UART
    - Bit 1 is 1 if there are data available
    - Bit 0 is 1 if the UART is ready to send data
  - A **data input** register where the received data are available to the processor
  - A **data output** register where the processor places data to send

A Classic UART

Configure and enable the UART

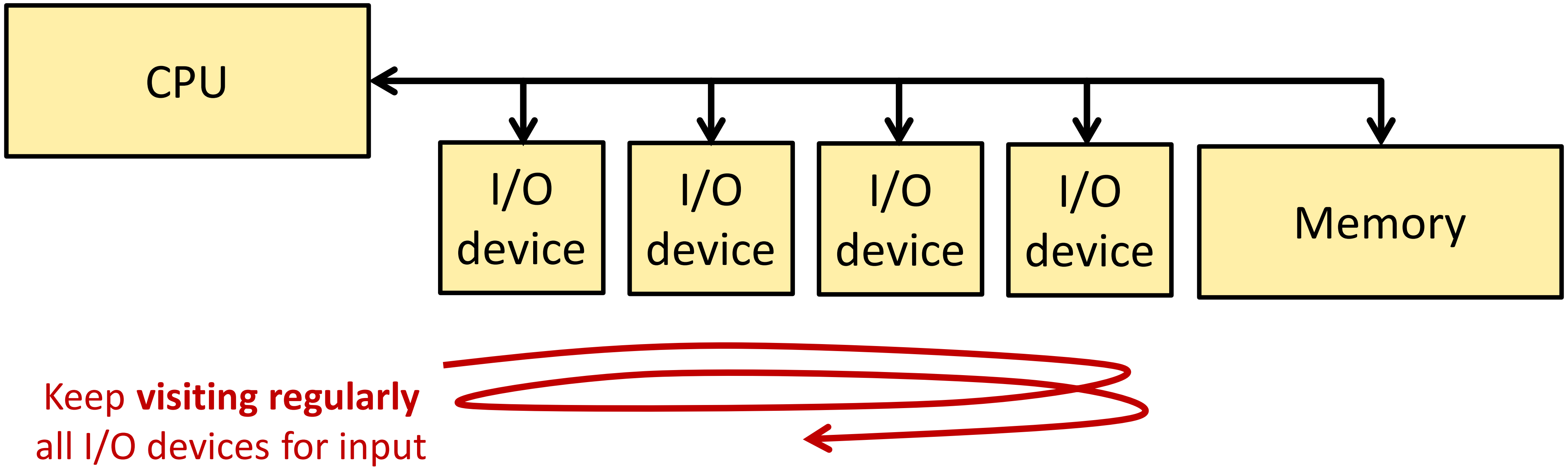
Wait until we can send a new character…

…and send it

```
UART_CTRL_ADDR     = 0x10000000  # UART status register address
UART_ENABLE_BIT    = 0x80        # Enable bit (bit 7)
UART_SPEED_9600    = 0x01        # Speed setting for 9600 baud (4 bits, [3:0])
UART_STATUS_ADDR   = 0x10000004  # UART status register address
TX_READY_BIT       = 0x01        # Transmitter ready bit (bit 0)
UART_DATAIN_ADDR   = 0x10000008  # UART data input (receive) register address
UART_DATAOUT_ADDR  = 0x1000000C  # UART data output (send) register address

send_string:
    li   t0, UART_CTRL_ADDR      # Get UART control address
    li   t1, UART_STATUS_ADDR    # Get UART status address
    li   t2, UART_DATAOUT_ADDR   # Get UART data address
    li   t3, UART_ENABLE_BIT     # Get enable bit (0x80)
    li   t4, UART_SPEED_9600     # Get speed setting (0x01)
    or   t4, t3, t4              # Combine enable and speed bits
    sw   t4, 0(t0)               # Configure using the UART control register

next_char:
    lb   t5, 0(a0)               # Load first byte of the string
    beqz t5, finish              # If byte is zero (null terminator), finish

check_tx_ready:
    lw   t6, 0(t1)               # Load UART status register
    andi t6, t6, TX_READY_BIT    # Check if TX_READY_BIT is set
    beqz t6, check_tx_ready      # If not ready, loop back and check again

    sw   t5, 4(t2)               # Store the character in UART data register
    addi a0, a0, 1               # Increment string pointer (move to next char)
    j    next_char               # Jump back to send the next character

finish:
    ret                          # Return when the string is done
```

I/O Polling

- How do we know if **a peripheral has data** for us (key pressed, packet arrived, etc.)?



- **Very expensive**: if the device is fast and requires immediate action, the processor must spend too much time to check **frequently**