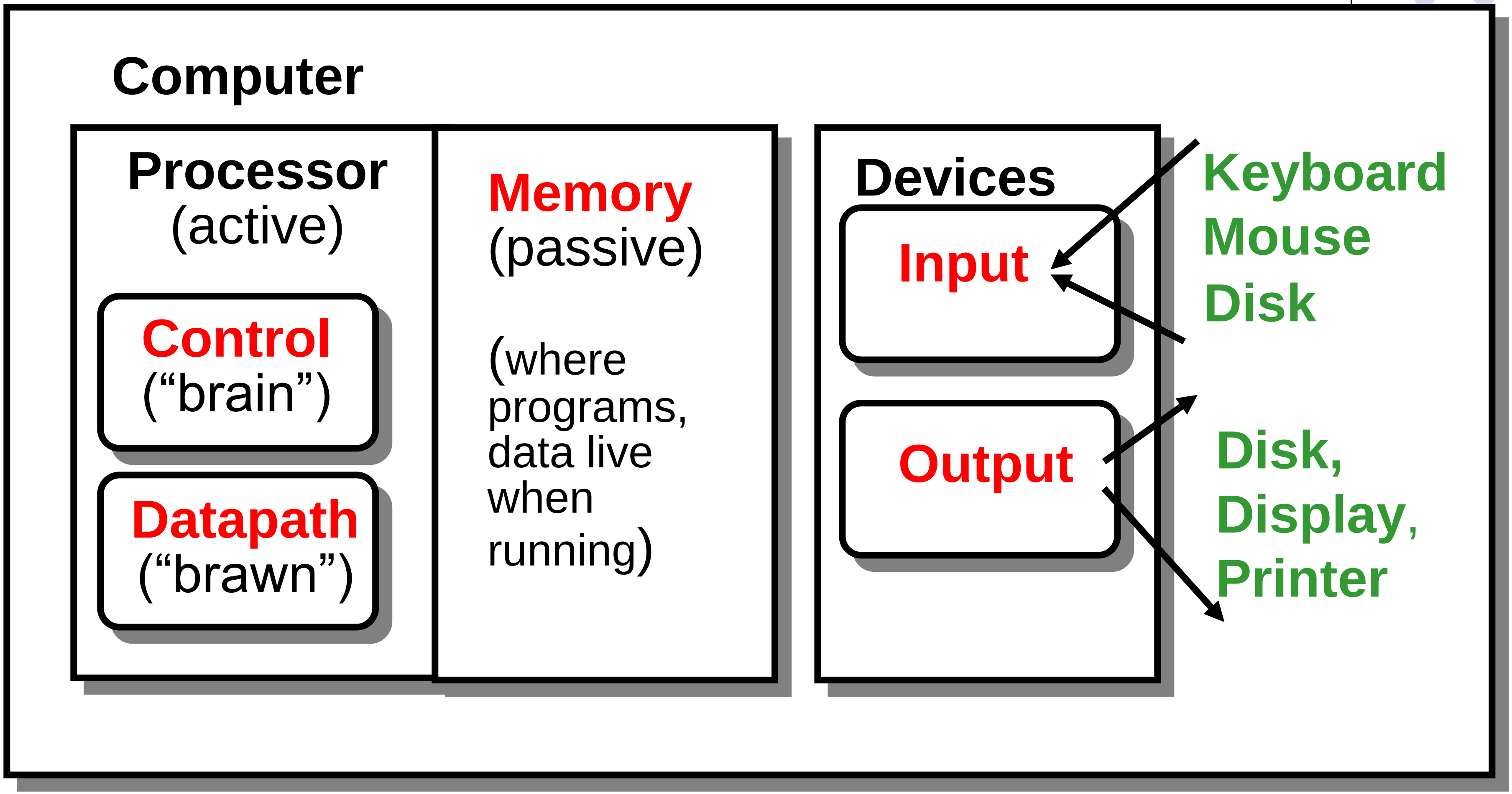
Buses and Parallel Input/Output

Lecturer: Sri Parameswaran
Notes by: Annie Guo

Lecture Overview

- Buses
  - Computer buses
- I/O Addressing
  - Memory mapped I/O
  - Separate I/O
- Parallel input/output
  - AVR examples

Five Components of Computers



Buses

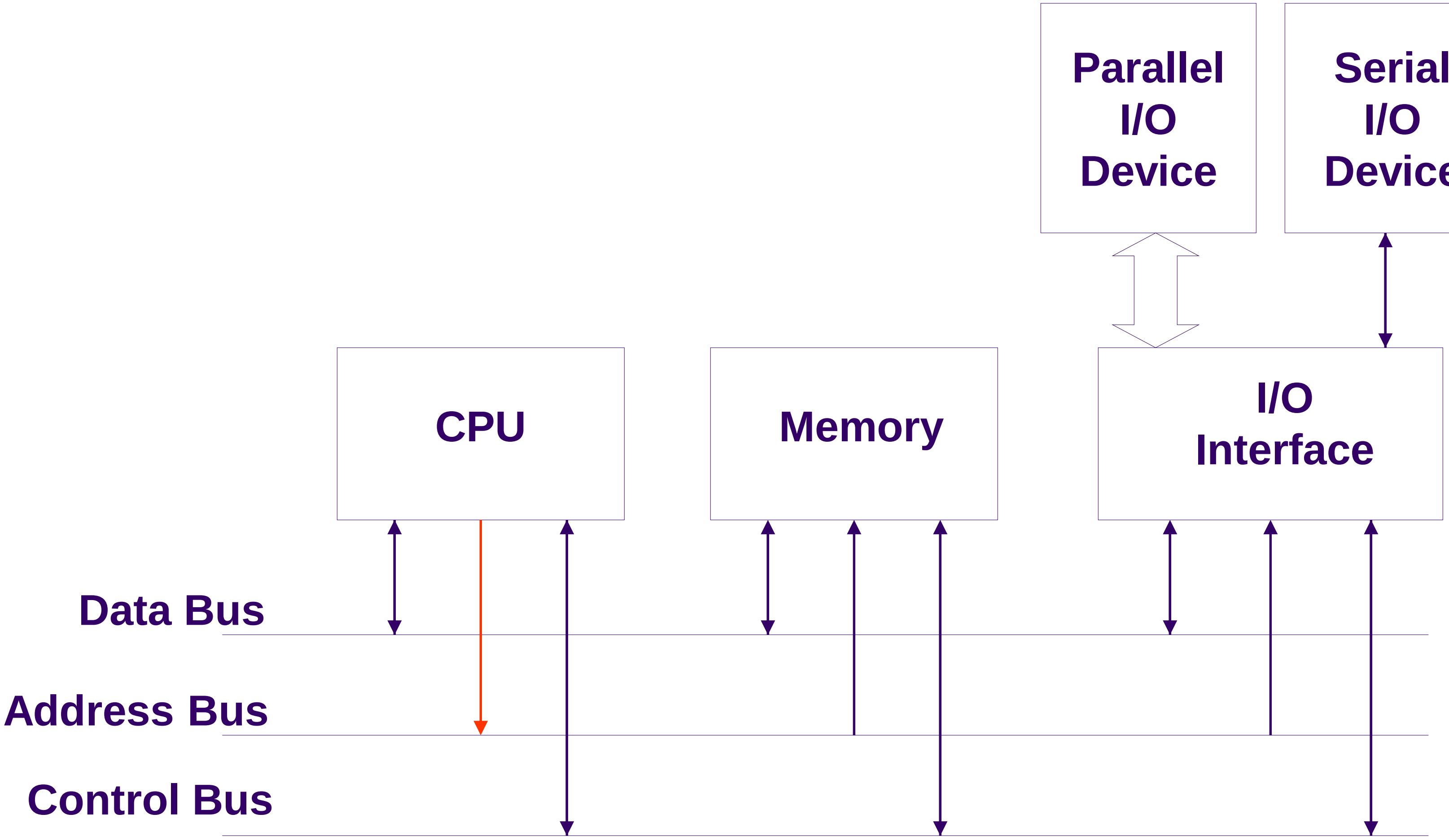
- A collection of wires through which data is transmitted from one of sources to destinations

- All buses consist of three parts:
  - data bus
    - transfer actual data
  - address bus
    - transfer information about where the data should go.
  - control bus
    - transfer control signals

Characteristics of Buses

- For system or higher level designs, buses can be characterized in
  - Bus width (in bits)
    - Determines how much data can be transmitted at a time. E.g. 16 bits, 32 bits
  - Clock speed in MHz
    - Determines how often data can be transferred on the busses

Typical Computer Bus Structure

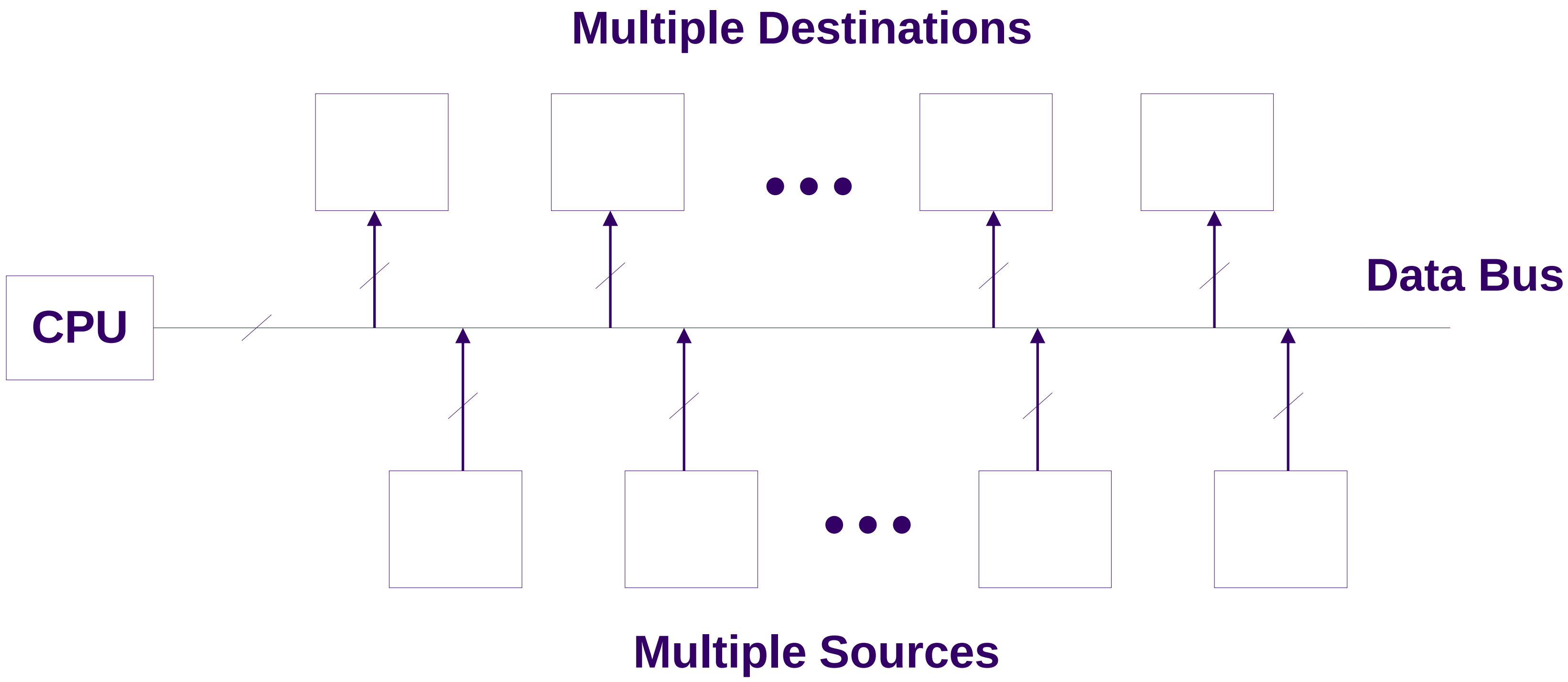


Computer Buses

- CPU is connected to memory and I/O devices via data, address and control buses.
- Data bus is bi-directional and transfers information (memory data and instructions, I/O data) to and from CPU.
- Address bus is most often unidirectional because the CPU is the only source of addresses.
- Control bus carries all control signals required to control the operation of the data transfer.

Computer Buses (cont.)

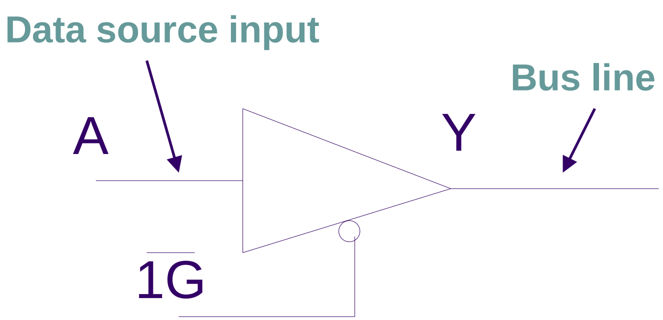
- Each line of a bus has multiple sources and destinations. The bus transfers data from one source each time.



Input Interface

- Connects multiple data sources
  - Only one source data is sent to the bus at a time
- Often implemented with three-state buffers for data buses
  - For example,
    - a parallel, eight-bit input data is connected to eight three-state gates whose enable lines are tied together
    - When the data is to sent to the bus the eight three-state gates are enabled.
- The open-collector gate is often used for control signals such as request for interrupts
  - Since one way switch is often required.

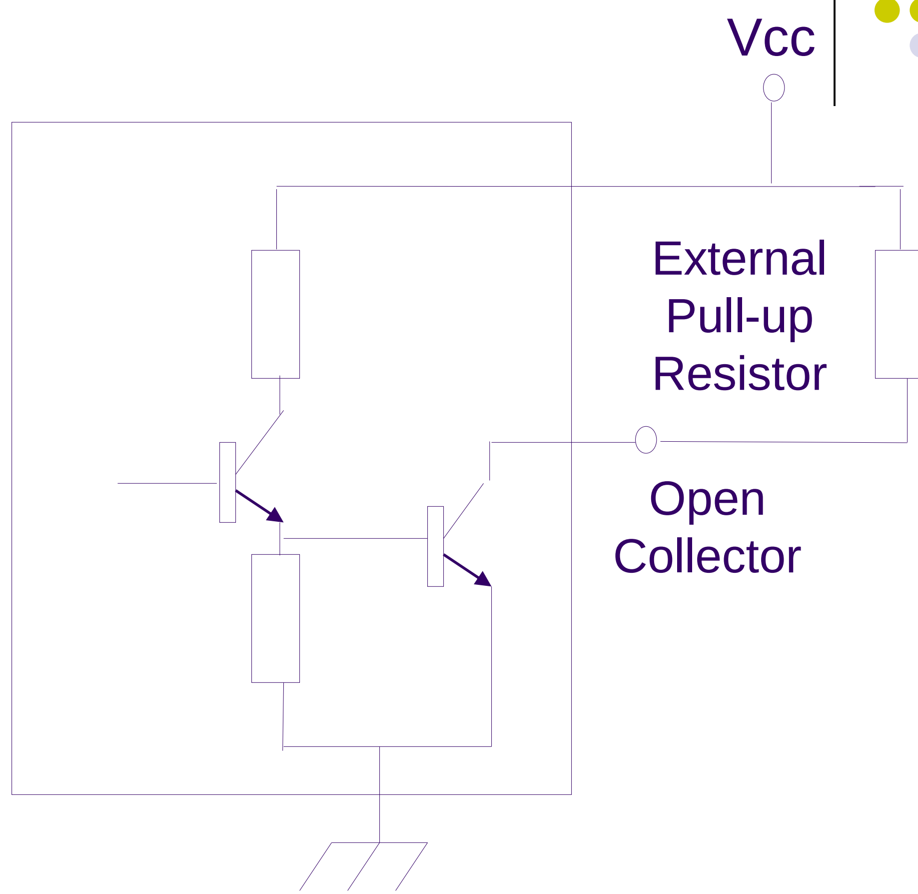
Typical Bus Interface Gates



$\overline{1G}$	A	Y
0	0	0
0	1	1
1	0	X
1	1	X

High Impedance

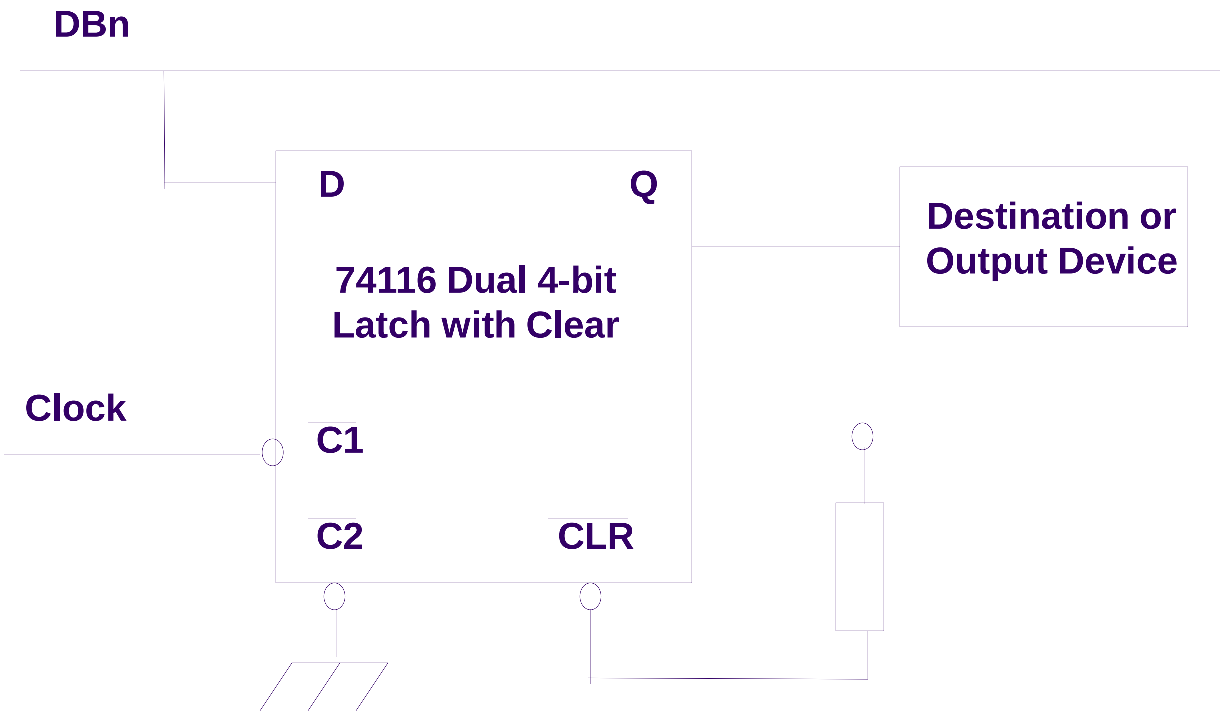
(a) Three-state gate



(b) Typical open-collector gate

Output Interface

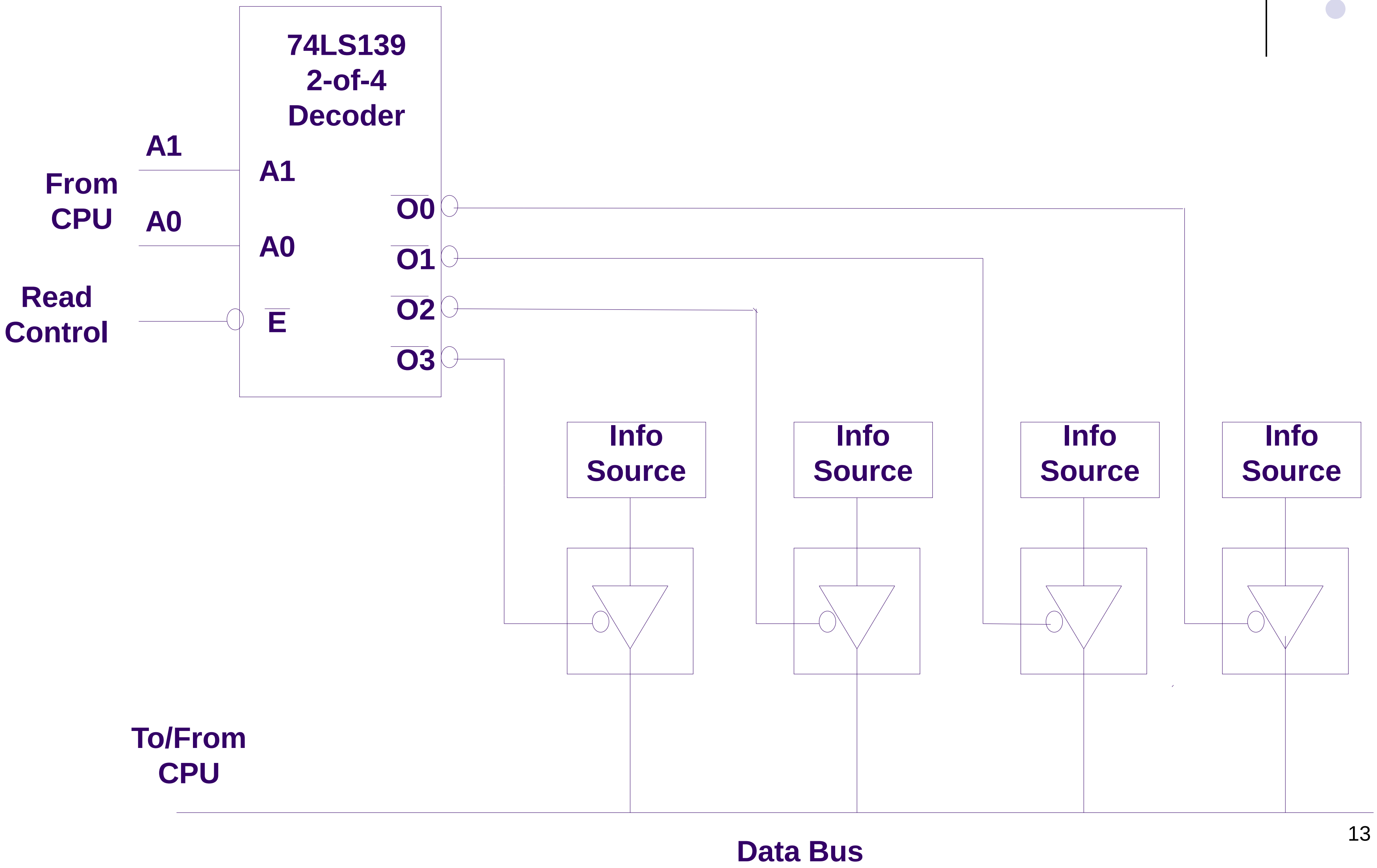
- The output interface between the data bus and a destination or output device contains a latch.



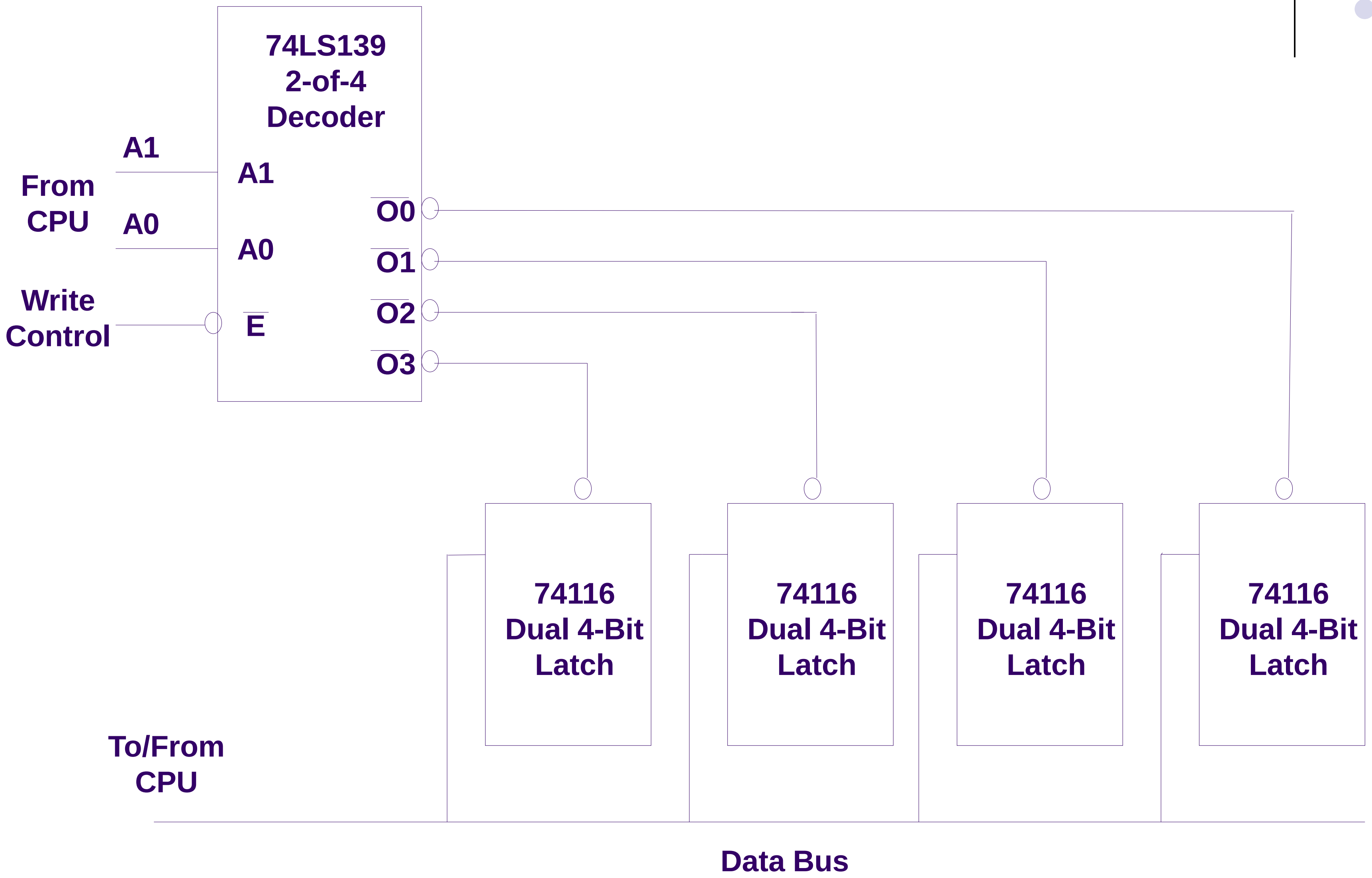
Address Decoding

- The interface must provides the ability for CPU to select one of many sources and destinations.
  - The address decoder is used.

Address Decoding for Input Devices



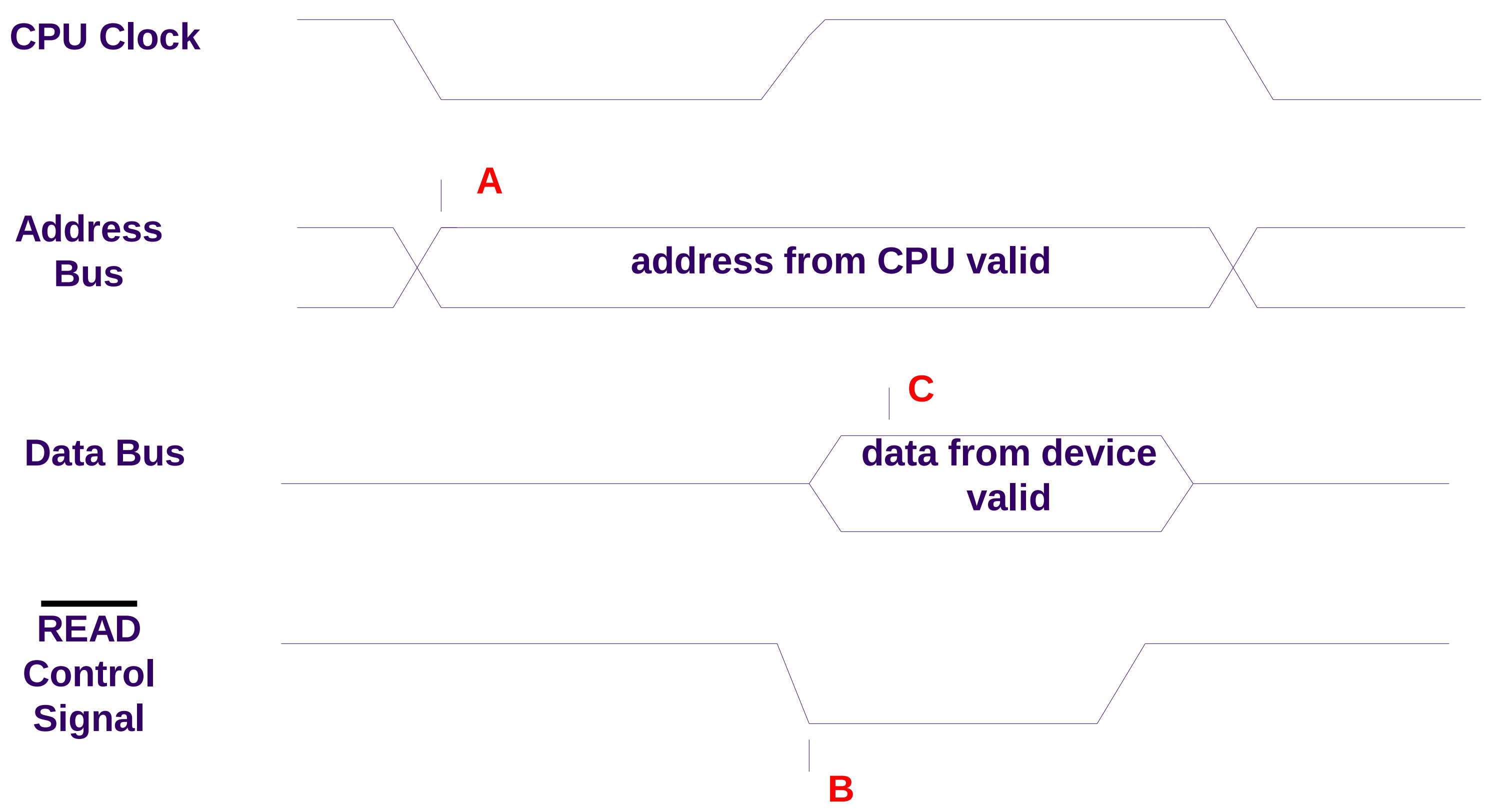
Address Decoding for Output Devices



CPU Timing Signals

- CPU must provide timing and synchronization so that the transfer of information occurs at the right time.
  - CPU has its own clock.
  - I/O devices may have a separate I/O clock.
  - Typical timing signals include $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$.

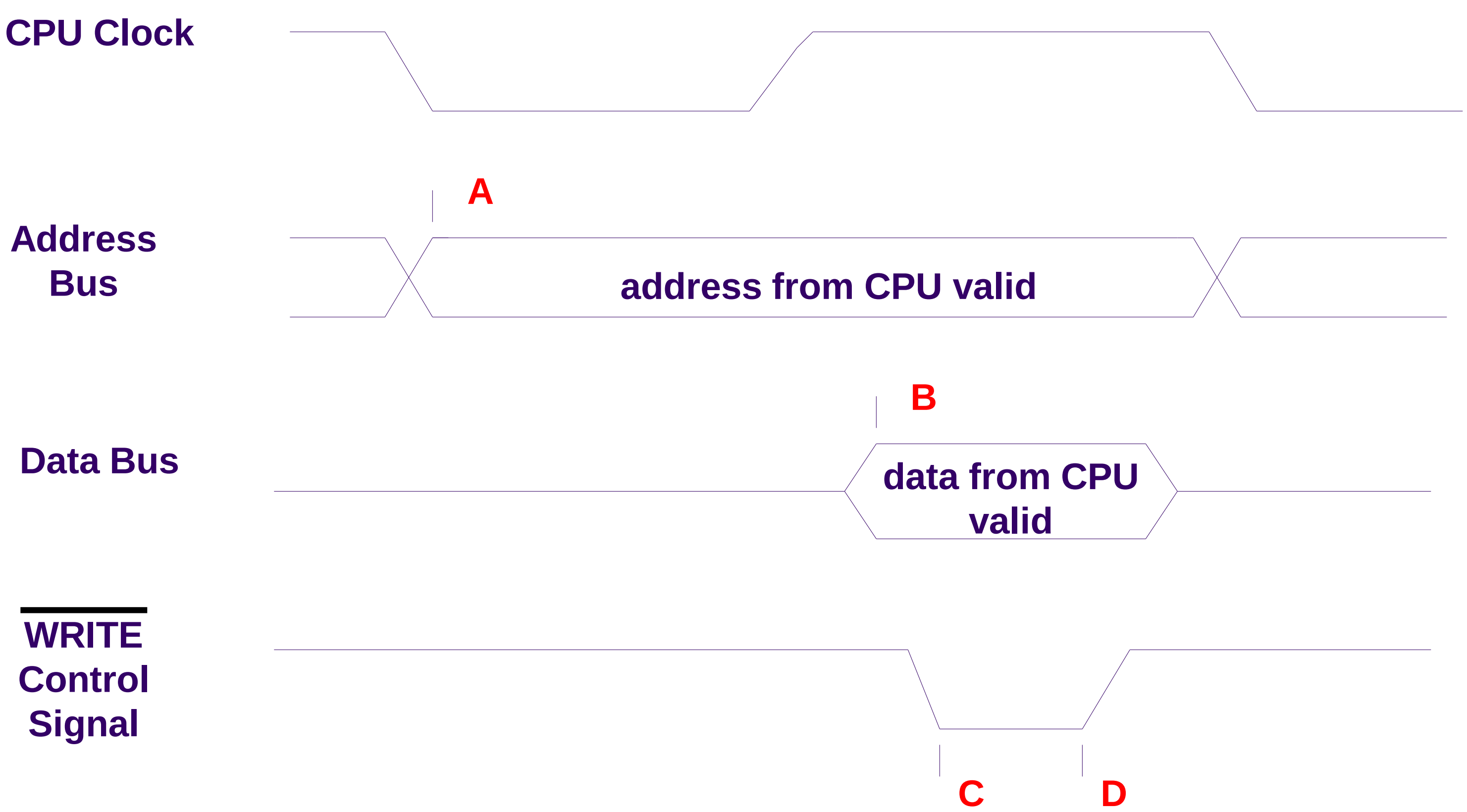
Typical CPU Read Cycle



Typical CPU Read Cycle

- CPU places the address on the address bus at point A.
- The control signal $\overline{\text{READ}}$ is asserted at point B to signal the external device that CPU is ready to take the data from the data bus.
- CPU reads the data bus at point C whether or not the input device has made it ready
  - If NOT, some form of synchronization is required.

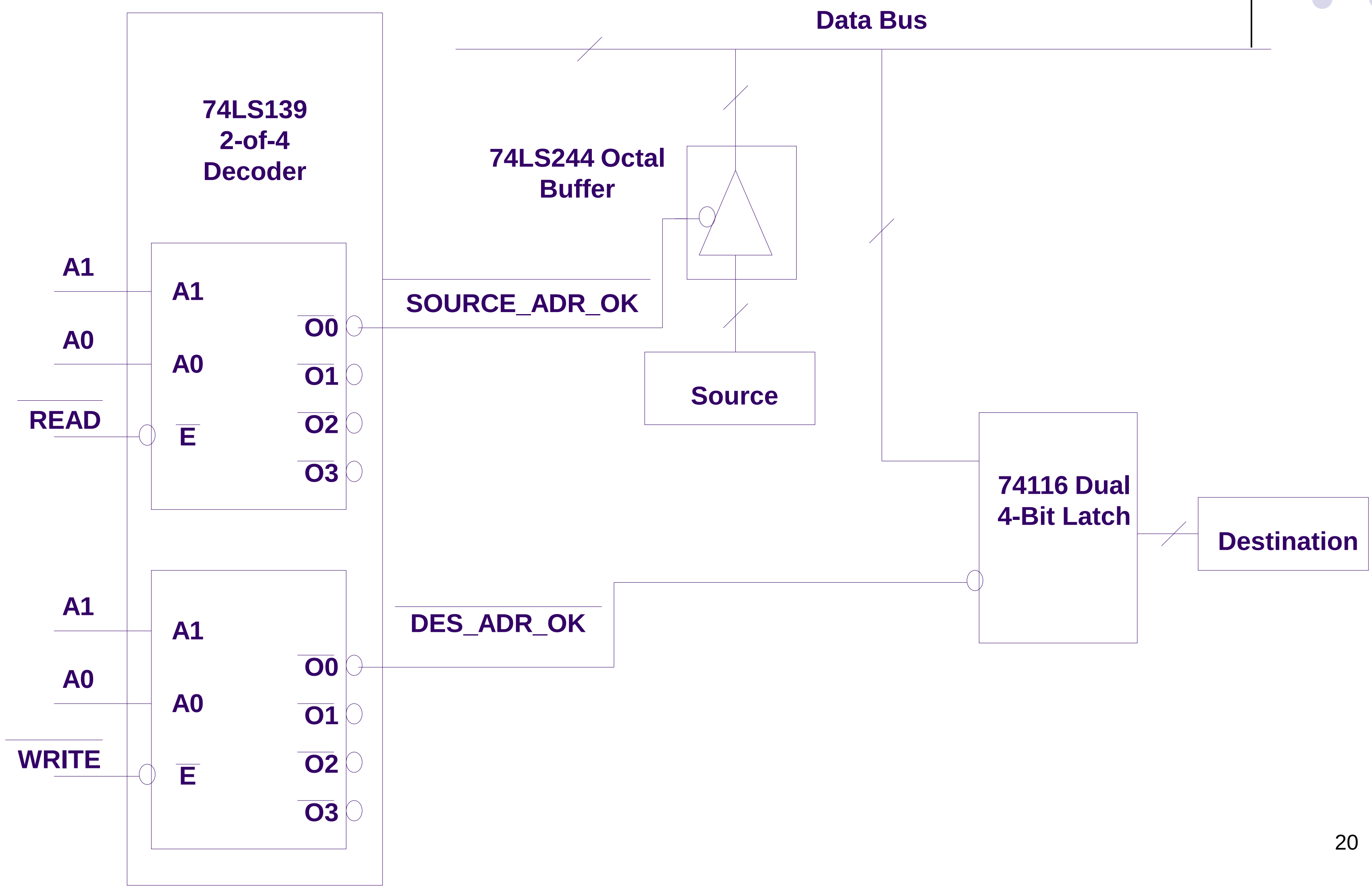
Typical CPU Write Cycle



Typical CPU Write Cycle

- CPU places the address on the address bus at point A.
- The data bits are supplied by CPU at point B.
- The control signal $\overline{\text{WRITE}}$ is asserted by CPU at point C to signal the external device that the data is ready to be taken from the data bus.
  - This signal is used to create the clock to latch the data at the correct time.
- Depending on the type of latch and when $\overline{\text{WRITE}}$ is asserted, the data may be captured on the falling edge or rising edge.

Complete I/O Interface



Complete I/O Interface (cont.)

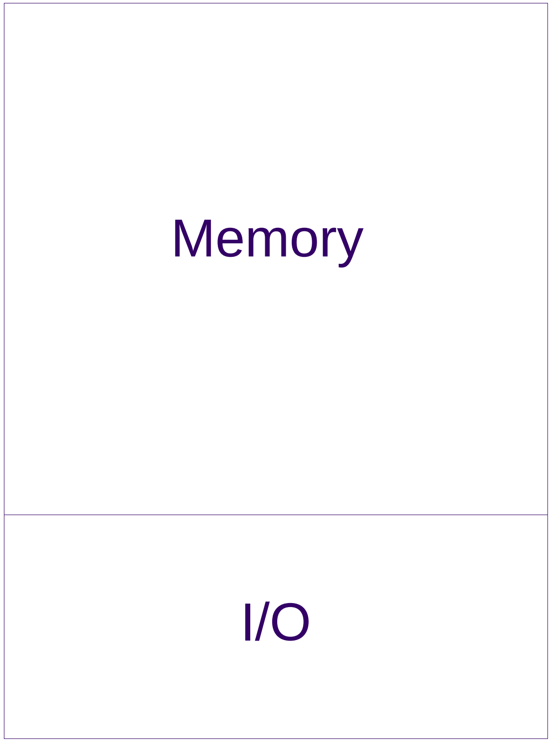
- $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$ control the enable ($\overline{\text{E}}$).
- The three-state enables and the latch clock signals are not asserted until the correct address is on the address bus AND the correct time in the read or write cycle has arrived.

I/O Addressing

- If the same address bus is used for both memory and I/O, how does hardware distinguish between memory reads/writes and I/O reads/writes?
  - Two approaches:
    - Memory-mapped I/O.
    - Separate I/O.
  - AVR supports both.

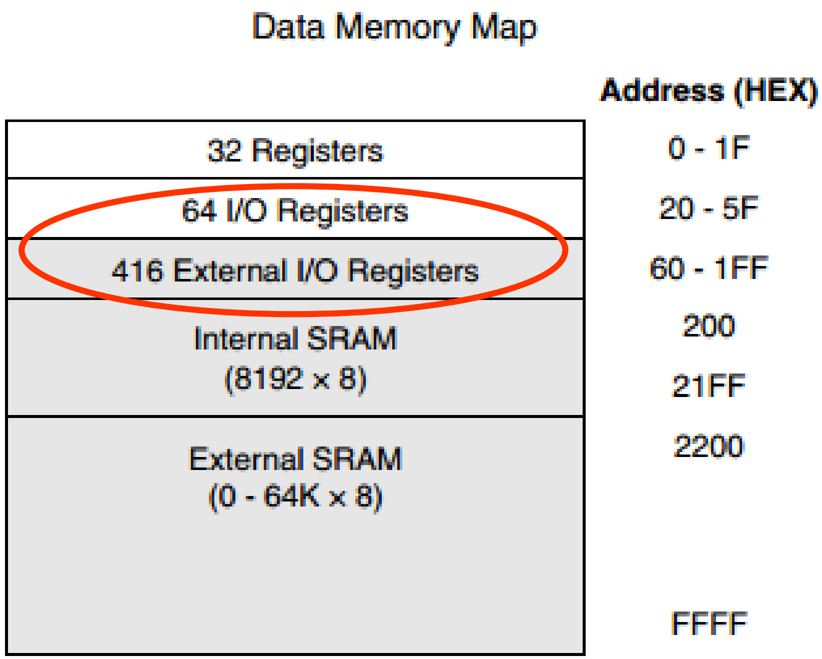
Memory Mapped I/O

- The entire memory address space is divided into memory space and I/O space.



AVR Memory Mapped I/O

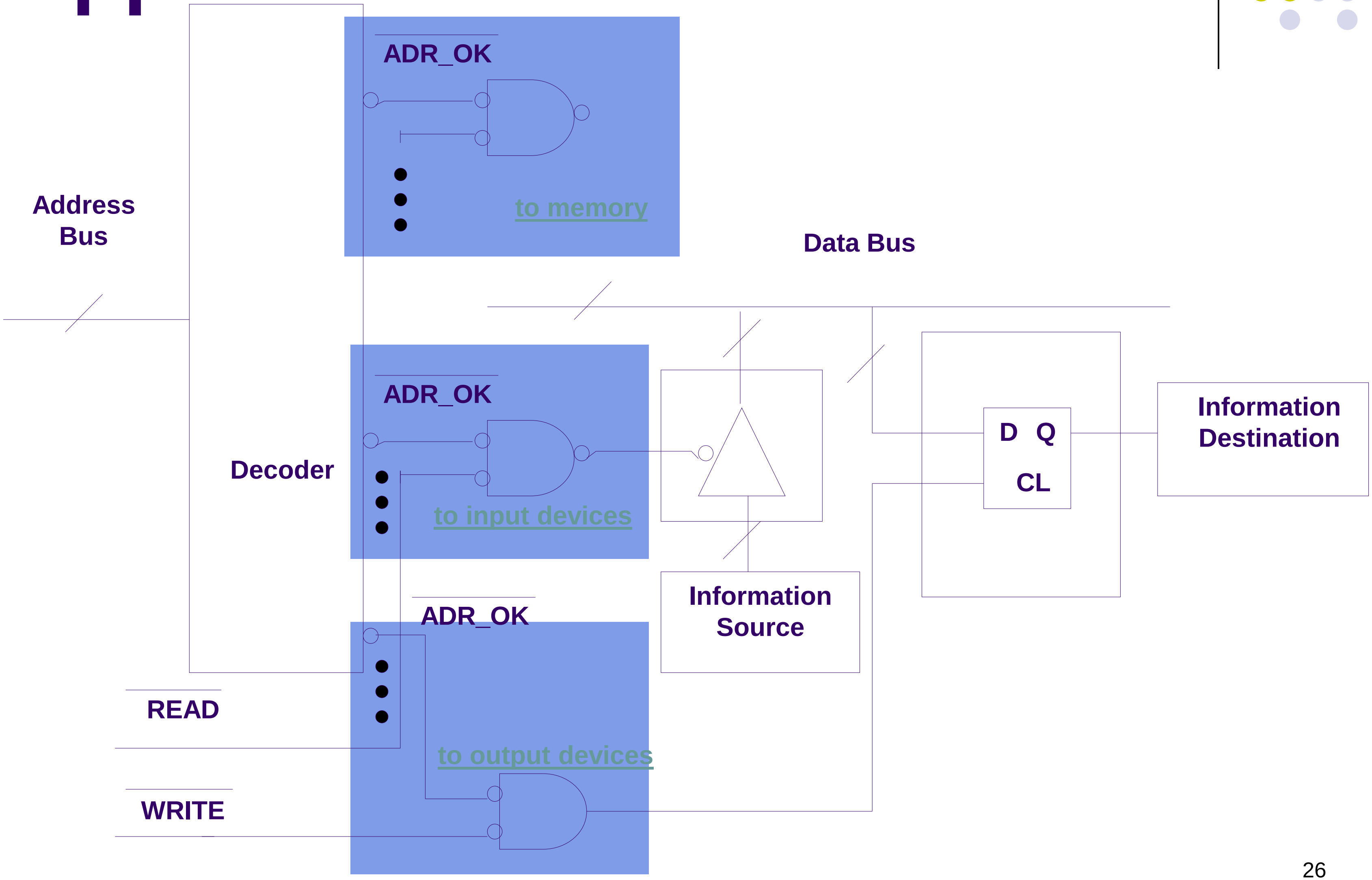
- In AVR, 480 I/O registers are mapped into memory space 0x0020 ~ 0x01FF
  - 1 byte each
- With such memory addresses, accesses to the I/O registers use memory access type of instructions.



Memory Mapped I/O (cont.)

- Advantages:
  - Simpler CPU design.
  - No special instructions for I/O accesses.
- Disadvantages:
  - I/O devices reduce the amount of memory space available for application programs.
  - The address decoder needs to decode the full address bus to avoid conflict with memory addresses.

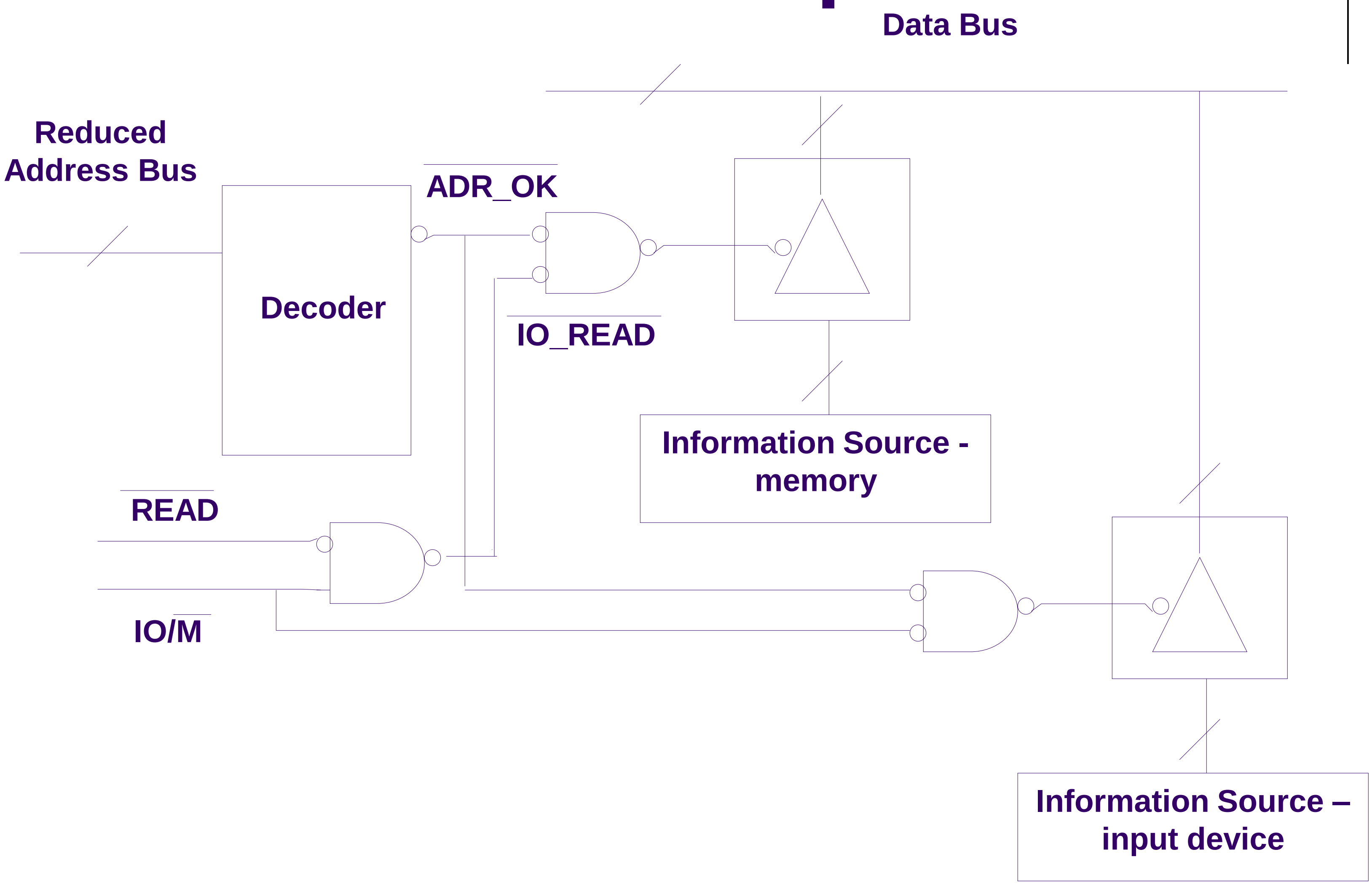
I/O Interface for Memory-Mapped I/O



Separate I/O

- Two separate spaces for memory and I/O.
  - Less expensive address decoders than those needed for memory-mapped I/O (Why?)
- Additional control signal, called $IO/\overline{M}$, is required to prevent both memory and I/O trying to place data on the bus simultaneously.
  - $IO/\overline{M}$ is high for I/O use and low for memory use.
- Special I/O instructions are required.

I/O Interface for Separate I/O



Separate I/O (cont.)

- In AVR, the first 64 I/O registers can also be addressed with separate addresses 0x00 ~ 0x3F
  - 1 byte each
- With such separate addresses, the I/O registers are accessed using I/O specific instructions.
  - E.g. `in` and `out`

I/O Synchronization

- CPU is typically much faster than I/O devices.
- Therefore, synchronization between CPU and I/O devices is required.
- Two synchronization approaches:
  - Software synchronization.
  - Hardware synchronization.

Software Synchronization

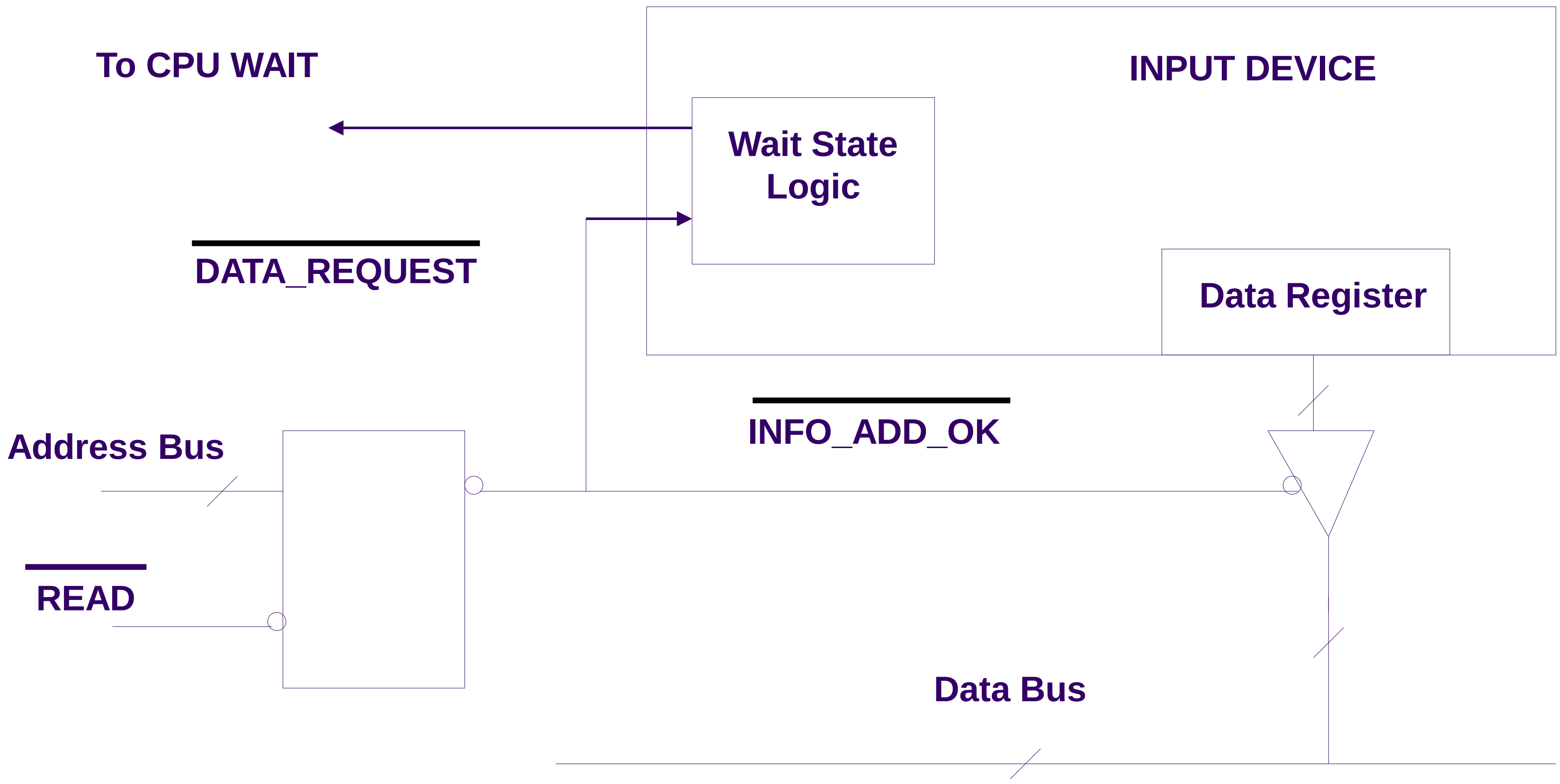
Two software synchronization approaches:

- Real-time synchronization
  - Uses a software delay to match CPU to the timing requirements of the I/O device.
    - The timing requirement must be known
    - Sensitive to CPU clock frequency.
    - Wastes CPU time.
- Polling I/O
  - A status register, with a DATA_READY bit, is added to the device. The software keeps reading the status register until the DATA_READY bit is set.
    - Not sensitive to CPU clock frequency.
    - Still waste CPU time, but CPU can do other tasks.

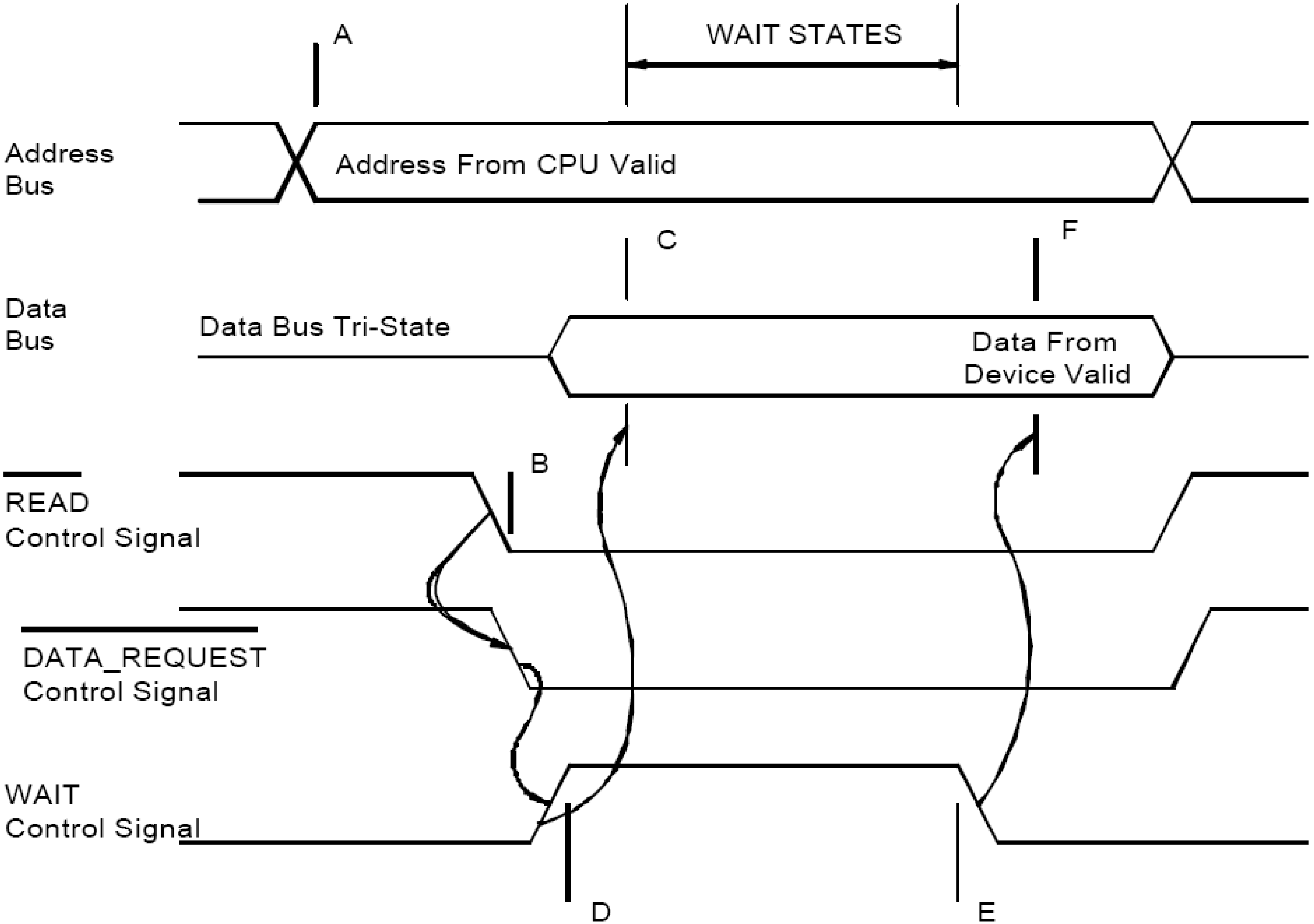
Handshaking I/O

- A hardware synchronization approach with control signal READY or WAIT.
  - For an input device, when CPU is asking for input data, the input device will assert WAIT if the input data is NOT available. When the input data is available, it will deassert WAIT. While WAIT is asserted, CPU must wait until this control signal is deasserted.
  - For an output device, when CPU is sending output data via the data bus, the output device will assert WAIT if it is not ready to take the data. When it is ready, it will deassert WAIT. While WAIT is asserted, CPU must wait until this control signal is deasserted.

Input Handshaking Hardware



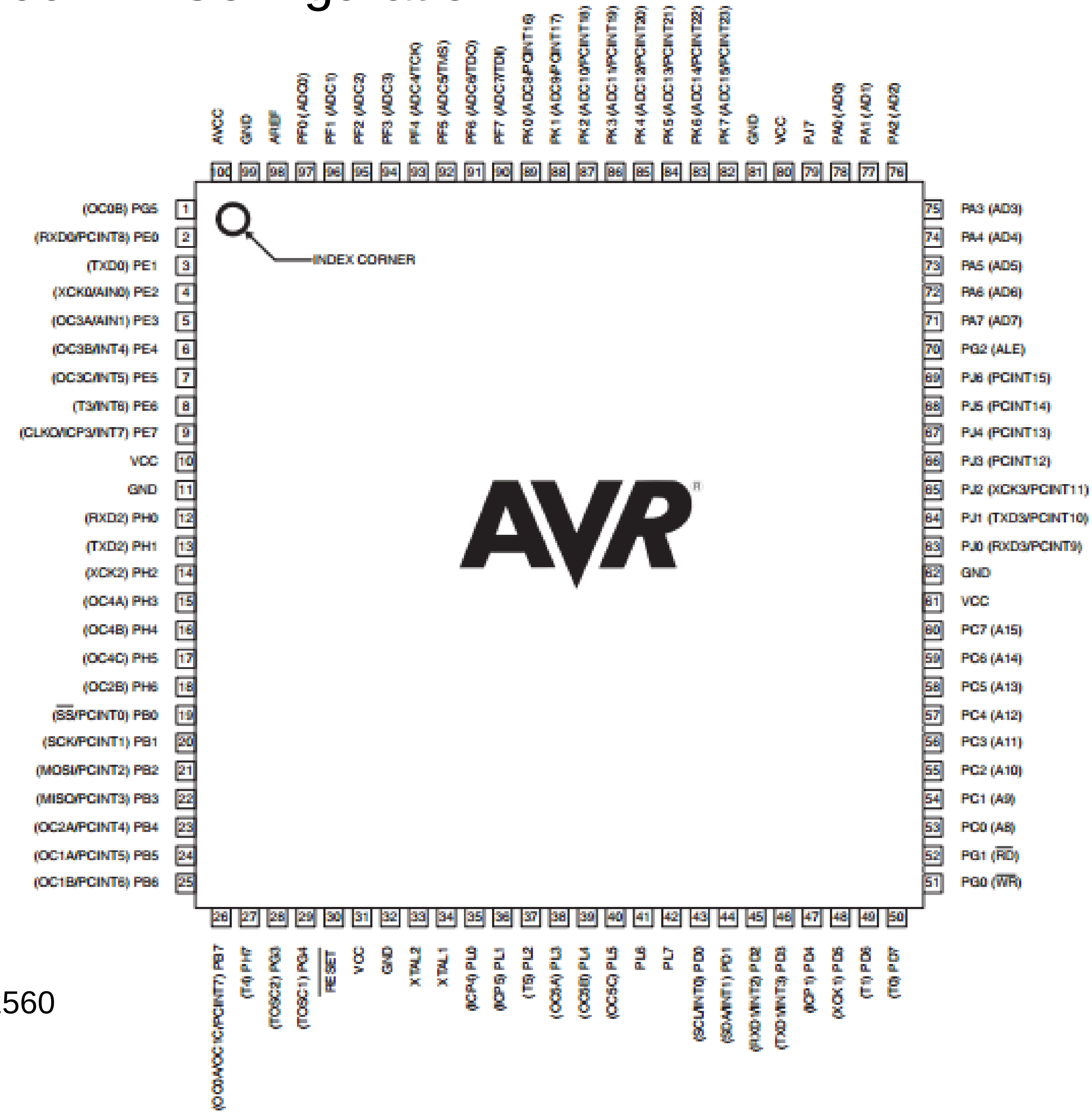
Read Cycle with Wait States



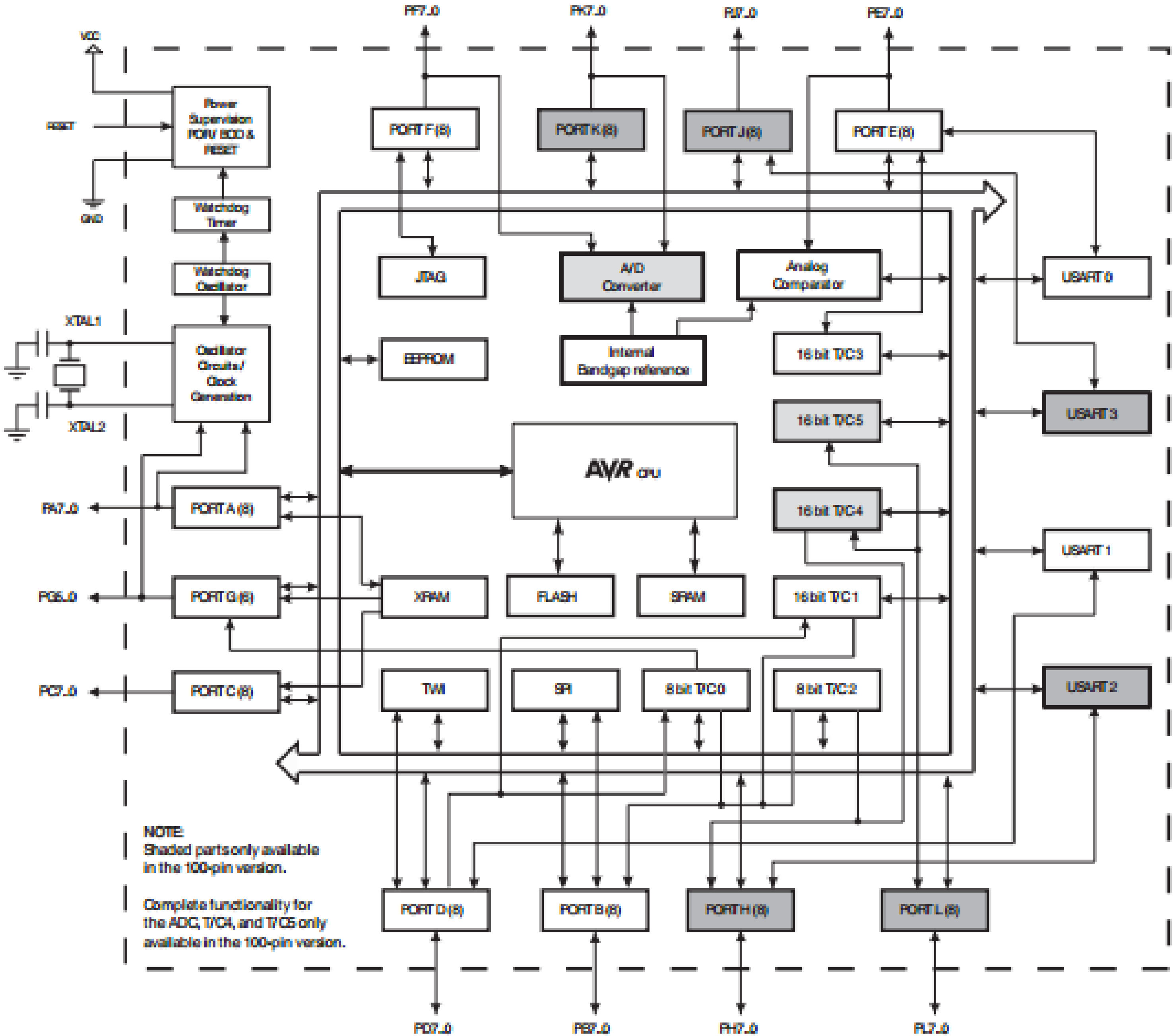
Parallel Input/Output in AVR

- Communication through ports
- There are two special instructions designed for parallel input/output operations
  - `in`
  - `out`

Atmega2560 Pin Configuration



Source: Atmega2560 Data Sheet



Source: Atmega2560 Data Sheet

AVR PORTs

- Can be configured to receive data or send out data
- Include physical pins and related circuitry to enable input/output operations.
- Different AVR microcontroller devices have different port design
  - ATmega2560 has 100 pins, most of them form eleven ports for parallel input/output.
    - Port A to Port H, Port J to Port L
    - Three I/O memory addresses (in data memory) are allocated for each port
      - PORTx for data register
      - DDRx for data direction register
      - PINx for port input pins

Load I/O Location to Register

- Syntax: ***in Rd, A***
- Operands: $0 \le d \le 31$, $0 \le A \le 63$
- Operation: Rd ← I/O(A)
- Words: 1
- Cycles: 1
- Example:

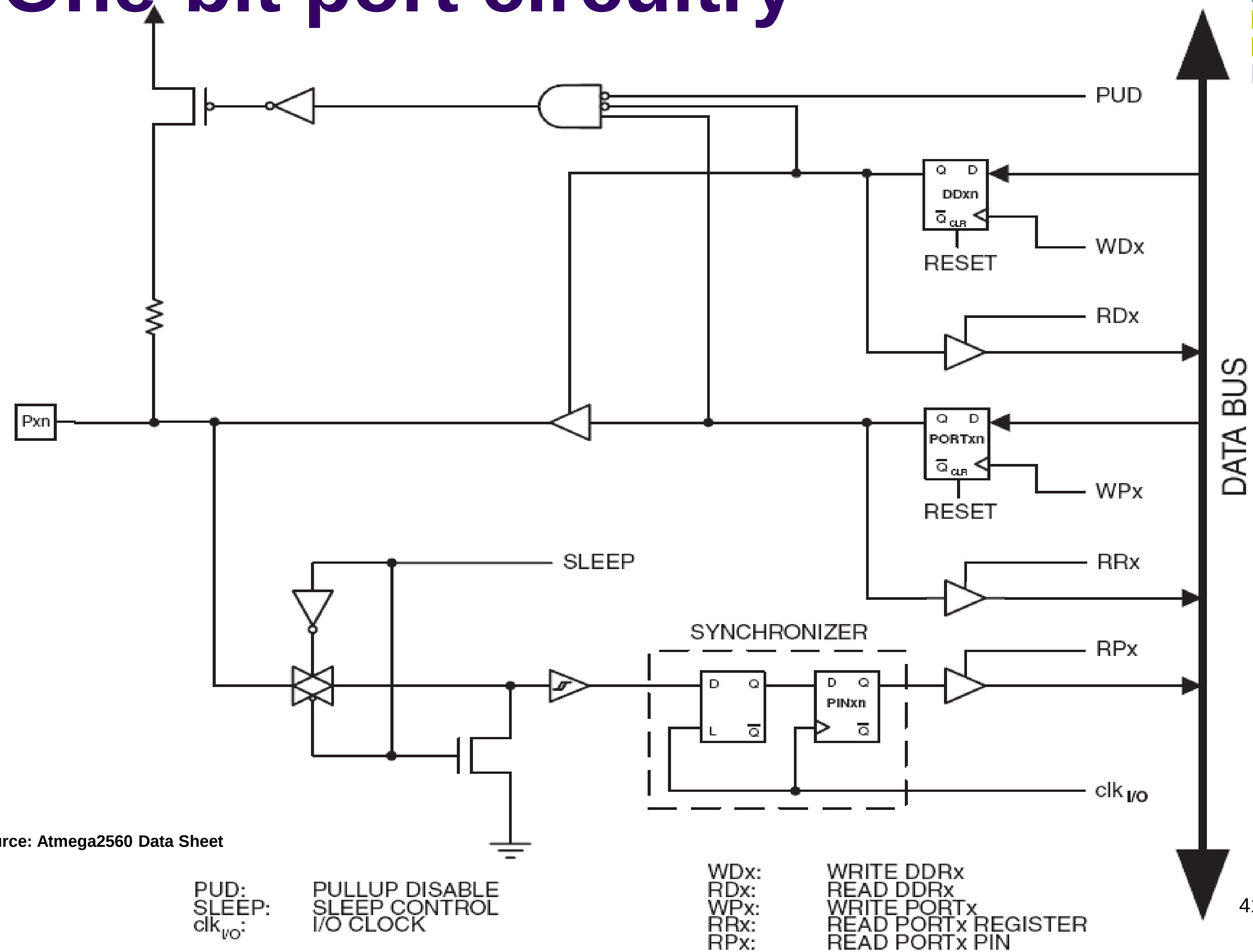
```
in r25, 0x00     ; read from port A
```

Store Register to I/O Location

- Syntax: ***out A, Rr***
- Operands: $0 \le r \le 31$, $0 \le A \le 63$
- Operation: I/O(A) ← Rr
- Words: 1
- Cycles: 1
- Example:

```
out 0x02, r16        ; write to port A
```

One-bit port circuitry



Source: Atmega2560 Data Sheet

How does it work?

- Each port pin consists of three register bits
  - DDxn, PORTxn, and PINxn.
    - DDxn bits are accessed at the DDRx I/O address,
    - PORTxn bits at the PORTx I/O address
    - PINxn bits at the PINx I/O address.
- The DDxn bit in the DDRx Register selects the direction of this pin.
  - If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin.

How does it work? (cont.)

- When the pin is configured as an input pin, the pull-up resistor can be activated/deactivated.
- To active pull-up resistor for input pin, PORTxn needs to be written logic one.

Sample code for output

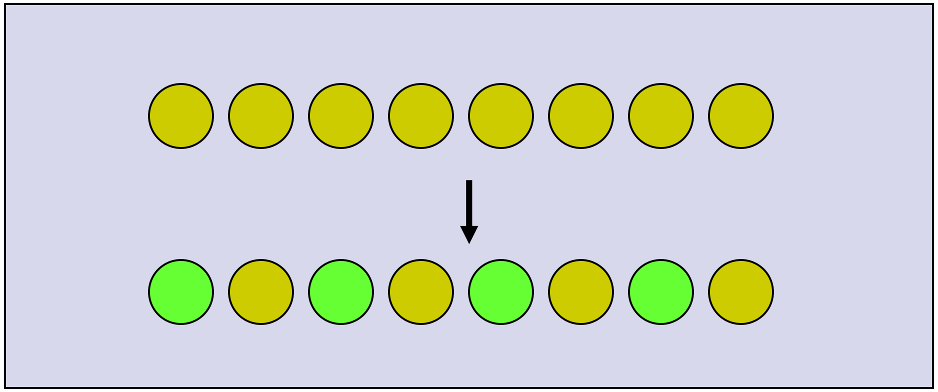
```
.include "m2560def.inc"

      clr   r16         ; clear r16
      ser   r17         ; set r17
      out   DDRA, r17   ; set Port A for output operation

      out   PORTA, r16  ; write zeros to Port A
      nop               ; wait (do nothing)
      out   PORTA, r17  ; write ones to Port A
```

Sample code for input

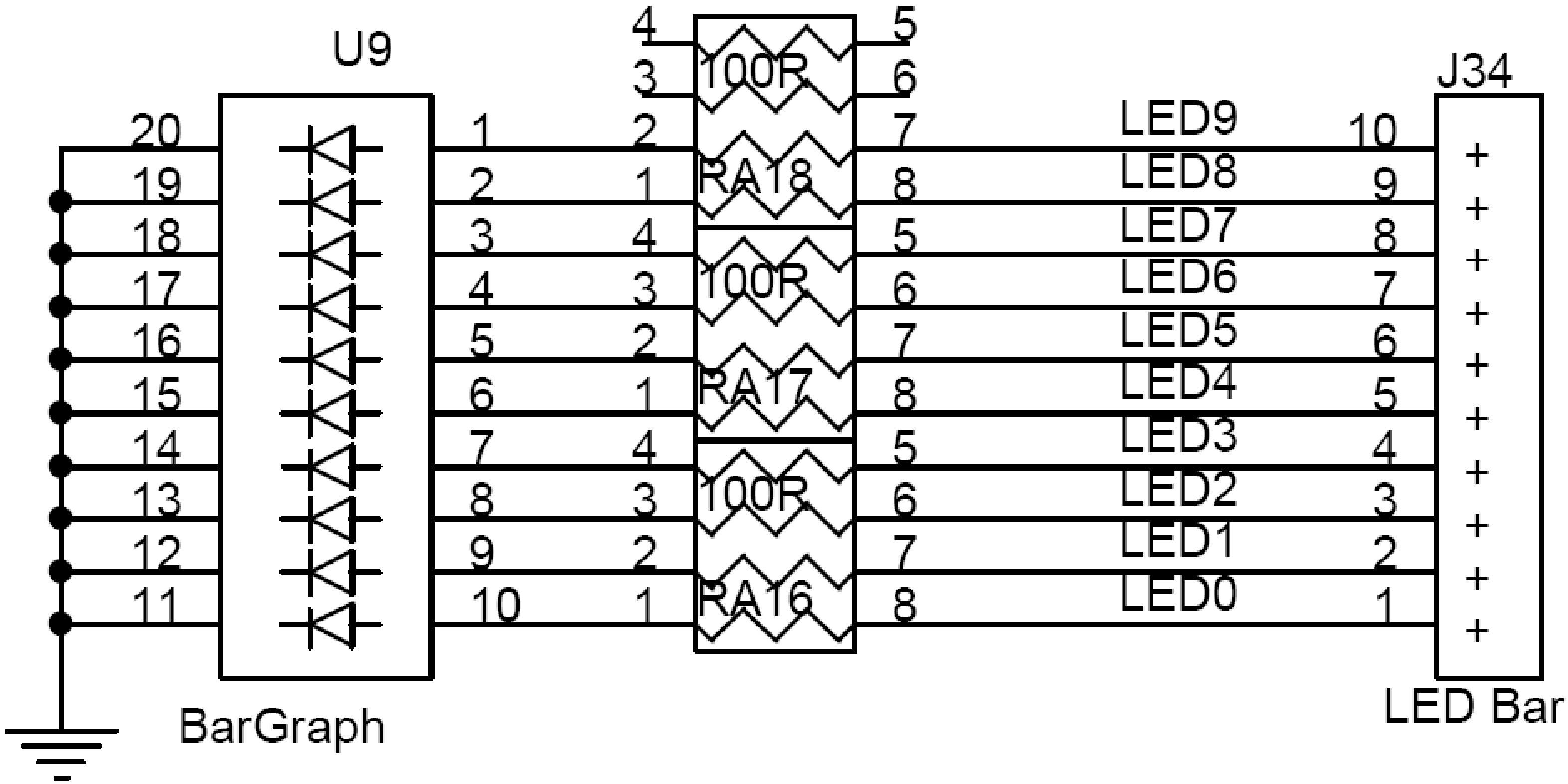
```
.include "m2560def.inc"

        clr     r15
        out     DDRA, r15       ; set Port A for input operation

        in      r25, PINA       ; read Port A
        cpi     r25, 4          ; compare read value with constant
        breq    exit            ; branch if r25=4
        ...
exit:   nop                     ; branch destination (do nothing)
```

Example 1

- Design a simple control system that can control a set of LEDs to display a fixed pattern.



LED and its operation



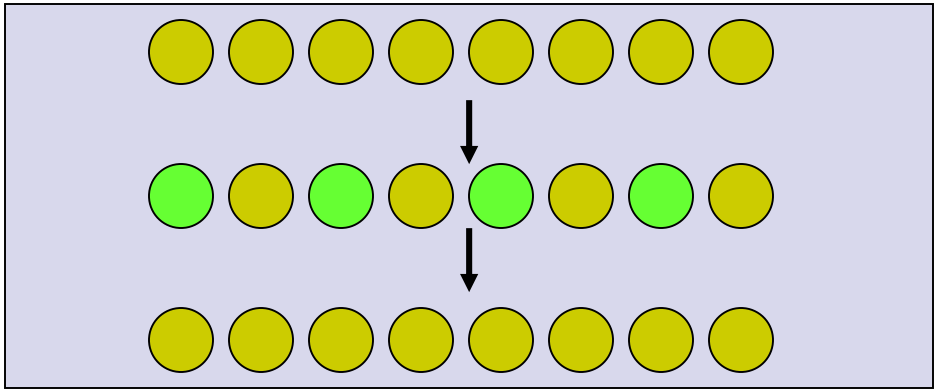
Example 1 (solution)

- Consists of a number of steps:
  - Set a port for the output operation, each pin of the ports is connected to one LED
  - Write the pattern value to the port so that it drives the LEDs to display the related pattern.

```
.include "m2560def.inc"
       ser r16
       out DDRA, r16        ; set Port A for output

       ldi r16, 0xAA        ; write the pattern
       out PORTA, r16
end:
       rjmp end
```

Example 2

- Design a simple control system that can control a set of LEDs to display a fixed pattern for *one second then turn the LEDs off.*



Example 2 (solution)

- Consists of a number of steps:
  - Set a port for the output operation, each pin of the ports is connected to one LED
  - Write the pattern value to the port so that it drives the display of LEDs
  - *Count one second*
  - *Write a pattern to set all LEDs off.*

Counting one second

- Assume we know that the clock cycle period is 1 ms (very very slow, not a real value). Then we can write a program that executes

$$1/10^{-3} = 1 \times 10^{3}$$

  single cycle instructions.

- Execution of the code will take 1 second if each instruction in the code takes one clock cycle.
- An implementation is given in the next slide

Code for one second delay

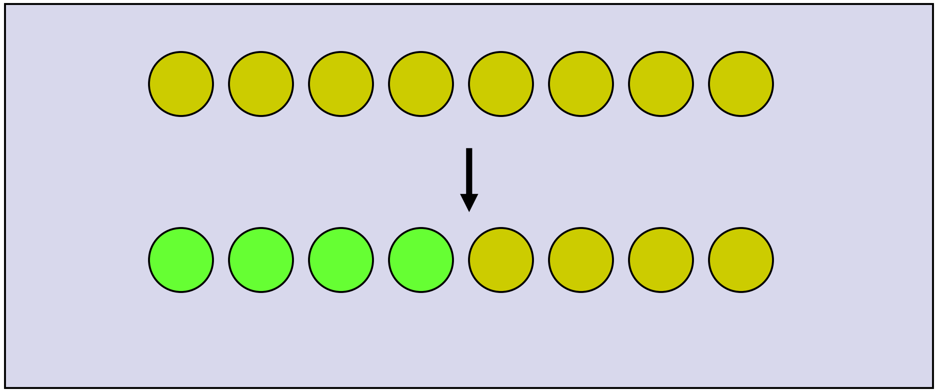
```
.include “m2560def.inc”
.equ loop_count = 124
.def iH = r25
.def iL = r24
.def countH = r17
.def countL = r16
.macro oneSecondDelay
       ldi countL, low(loop_count)   ; 1 cycle
       ldi countH, high(loop_count)
       clr iH                        ; 1
       clr iL
loop:  cp iL, countL                 ; 1
       cpc iH, countH
       brsh done                     ; 1, 2 (if branch)
       adiw iH:iL, 1                 ; 2
       nop
       rjmp loop                     ; 2
done:
.endmacro
```

Code for Example 2

```
.include “m2560def.inc”
        ser r15
        out DDRA, r15            ; set Port A for output

        ldi r15, 0xAA            ; write the pattern
        out PORTA, r15
        oneSecondDelay           ; 1 second delay
        ldi r15, 0x00
        out PORTA, r15           ; turn off the LEDs
end:
        rjmp end
```

Example 3

- Design a simple control system that can control a set of LEDs to display a fixed pattern *specified by the user.*



Example 3 (solution)

- Consists of a number of steps:
  - Set the switches and connect the switches to the pins of a port
  - Set the port for input
  - Read the input
  - Set another port for the output operation, each pin of the ports is connected to one LED
  - Write the pattern value to the port so that it drives the display of LEDs

Code for Example 3

```
.include “m2560def.inc”
        clr r17
        out DDRC, r17         ; set Port C for input
        ser r17
        out PORTC, r17        ; activate the pull up
        in r17, PINC          ; read pattern set by the user
                              ; from the switches
        ser r16
        out DDRA, r16         ; set Port A for output

        out PORTA, r17        ; write the input pattern
end:
        rjmp end
```

Example 4

- Design a simple control system that can control a set of LEDs to display a pattern specified by the user *during the execution*.

Example 4 (solution)

- One solution is the processor continuously checking if there is an input to read. If there is, then read the input and go to next task, otherwise the processor stays waiting for input. Such an approach to handle dynamic input is called ***polling***.

Code for Example 4

- Set an extra input bit for signal from user when the input is ready.

```
.include “m2560def.inc”

        cbi DDRB, 0                 ; clear Port B bit 0 for input

waiting:sbis PINB, 0                ; if yes skip to the next instruction
        rjmp waiting                ; waiting

        clr r17
        out DDRC, r17               ; set Port C for input
        ser r17
        out PORTC, r17              ; activate the pull up
        in r17, PINC                ; read pattern set by the user
                                    ; from the switches
        ser r16
        out DDRA, r16               ; set Port A for output

        out PORTA, r17
end:    rjmp end
```

Reading Materials

- Chapter 7: Computer Buses and Parallel Input and Output. Microcontrollers and Microcomputers by Fredrick M. Cady.
- ATmega2560 Data Sheet.
  - AVR CPU Core
    - PORTS

Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
   - Arithmetic and logic instructions
     - `ser`
   - Data transfer instructions
     - `in`, `out`
   - Bit operations
     - `sbi`, `cbi`
   - Program control instructions
     - `sbic`, `sbis`
   - MCU control instructions
     - `nop`

Homework

2. To make the AVR processor skip an amount of time without doing anything, we use nop instruction in the program shown in the example in this lecture. Can we use `mov Rd, Rd` to replace the nop instruction? Any difference?

Homework

3. One of very common functions a microcontroller application usually has is timing control. The function below is such a timing control function. Convert it to assembly program.

```
static int iSeconds, iMinutes:
void timing-control(void) {
        ++iSeconds;
        if (iSeconds >= 60) {
                iSeconds = 0;
                ++iMinutes;
                if (iMinutes > 30) {
                        //do something
                        //and reset the timer
                }
        }
}
```