Computer Systems and Professional Practice

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 1 - I/O Mechanisms

Topic Outline

Memory mapped I/O

Synchronisation and transferring data with an I/O device

Busy-wait polling techniques

Handshaking protocols for data synchronisation

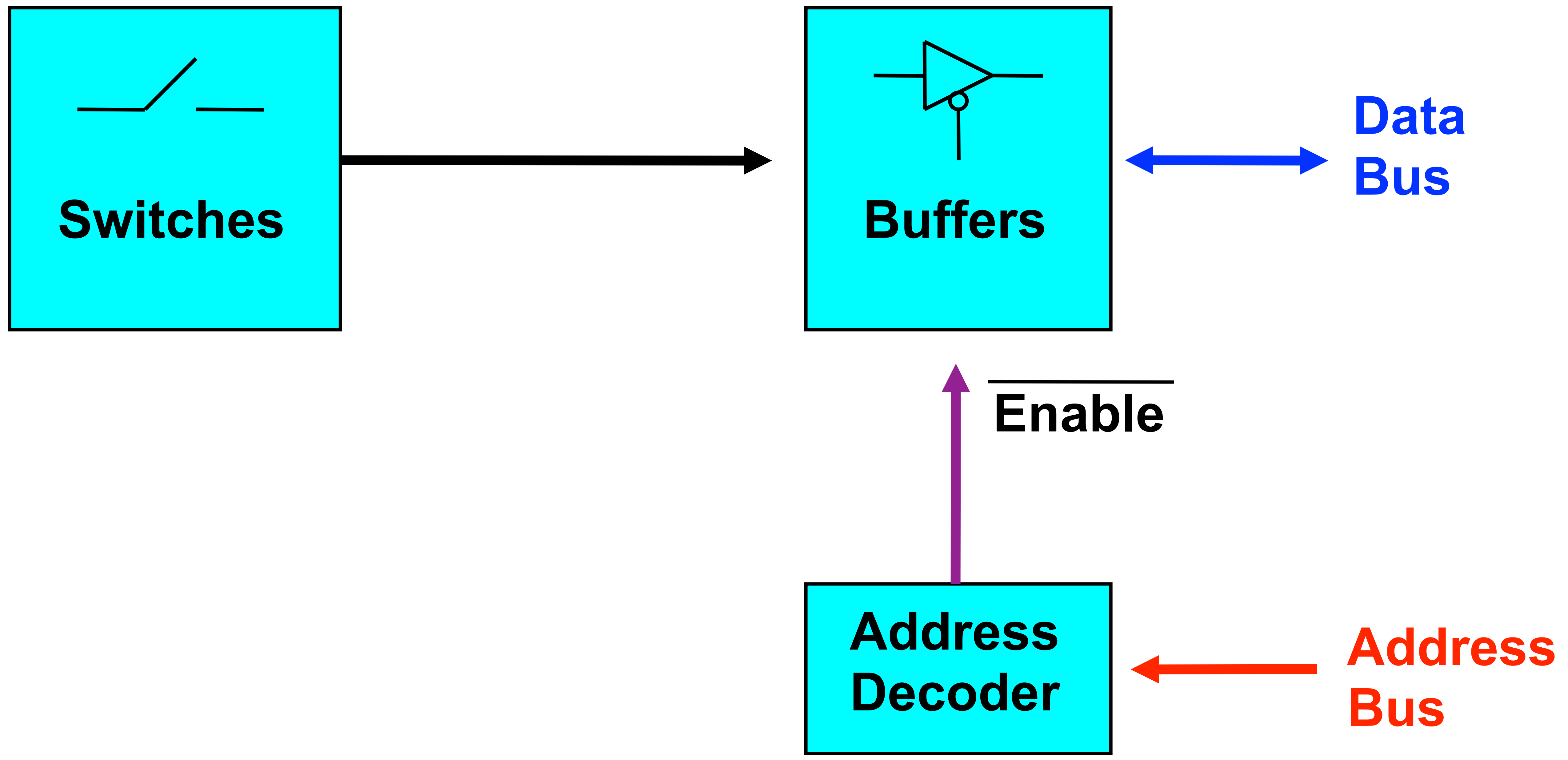
Interrupts and the principles of interrupt driven I/O

The differences between I/O DMA data transfer and programmed I/O

Memory Mapped I/O

Input Example

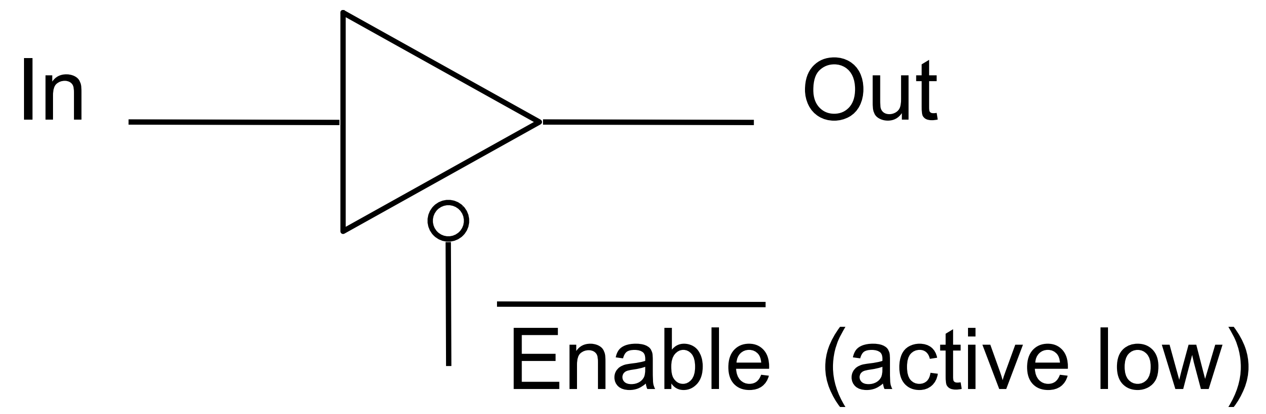
We've previously seen the principles underpinning the diagram below in relation to digital logic circuits



Reminder - Three-state Logic

The logic components considered so far have two possible logic states, i.e., 1 or 0

There is a further gate, termed a three-state buffer, whose output can be placed in a third state, known as UNCONNECTED



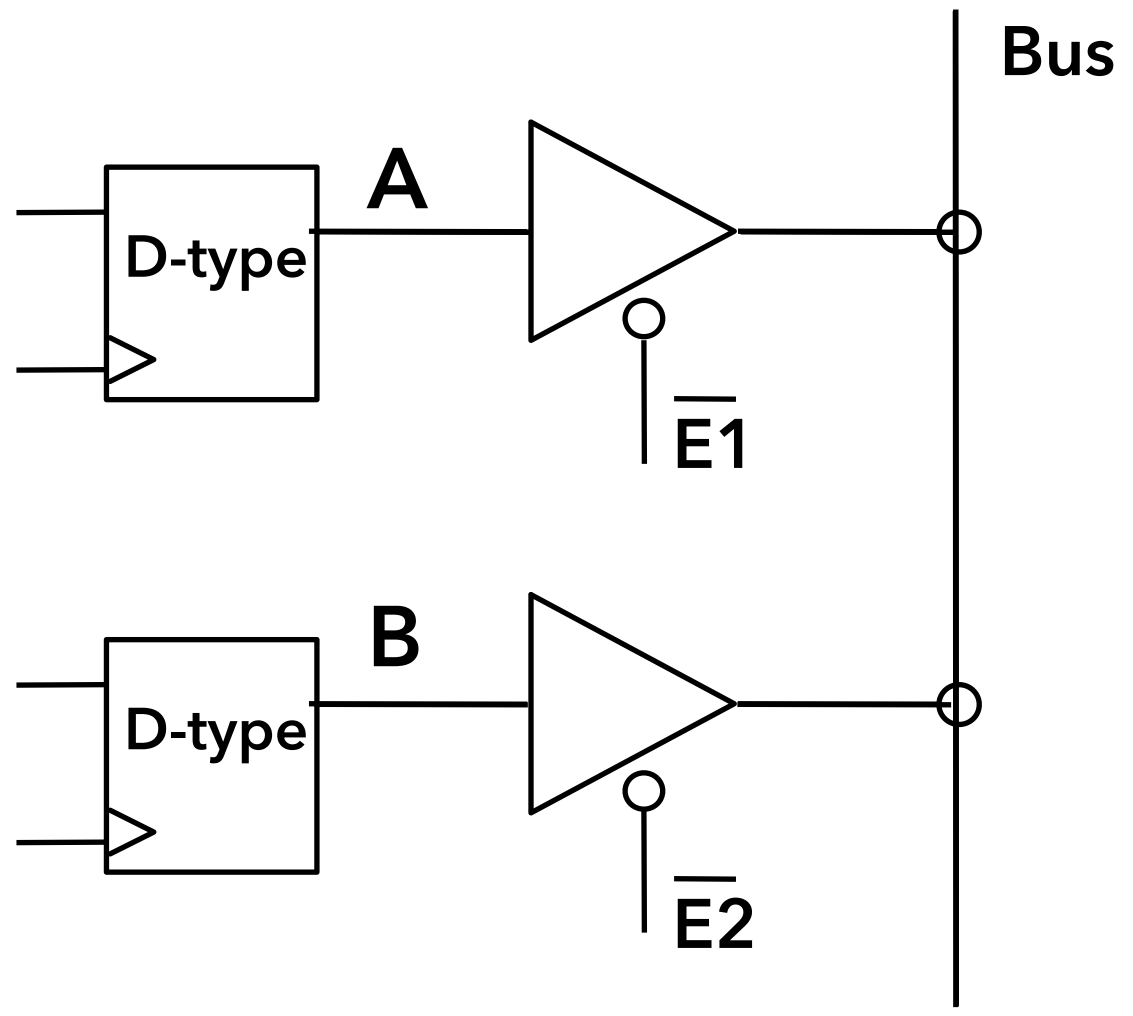
When $\overline{\text{Enable}}$ is high, the input is disconnected from the output

When $\overline{\text{Enable}}$ is low, the input is connected to the output

Reminder - Three-state Buses

Can use three state buffers to allow different sources of data onto a common BUS

For example, consider a 1-bit wide bus



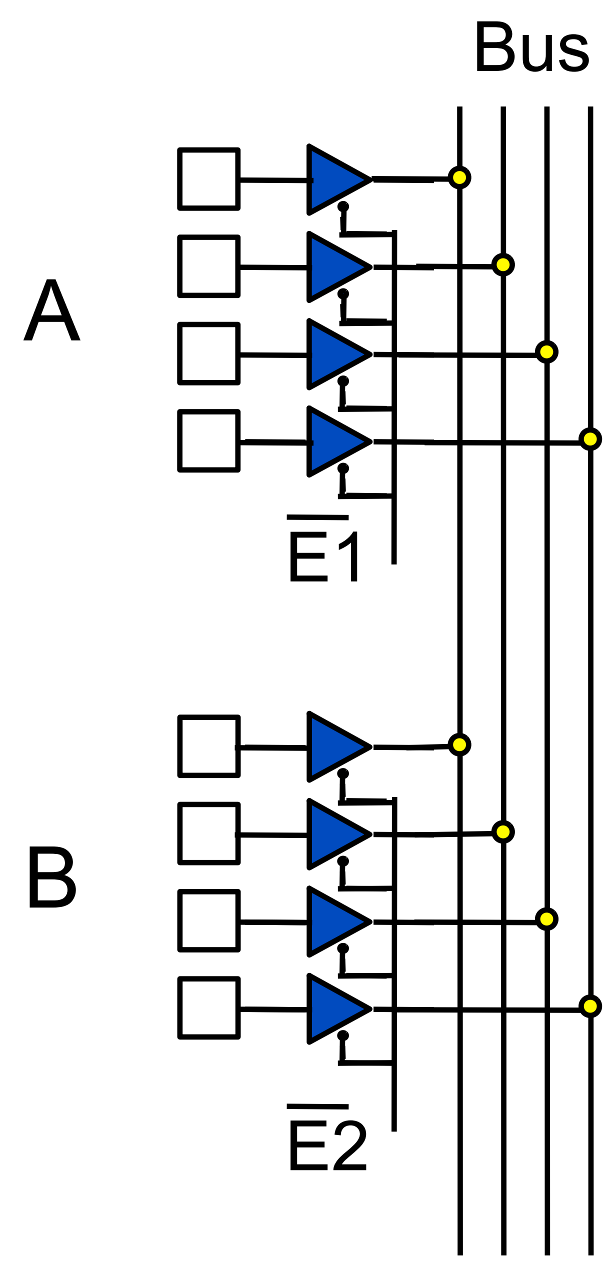
When $\overline{E1}$ = 0,
A will be placed on the Bus

When $\overline{E2}$ = 0,
B will be placed on the Bus

N.B. Should not have $\overline{E1}$=0 and $\overline{E2}$ = 0 at the same time!

Reminder - Multi-bit Bus

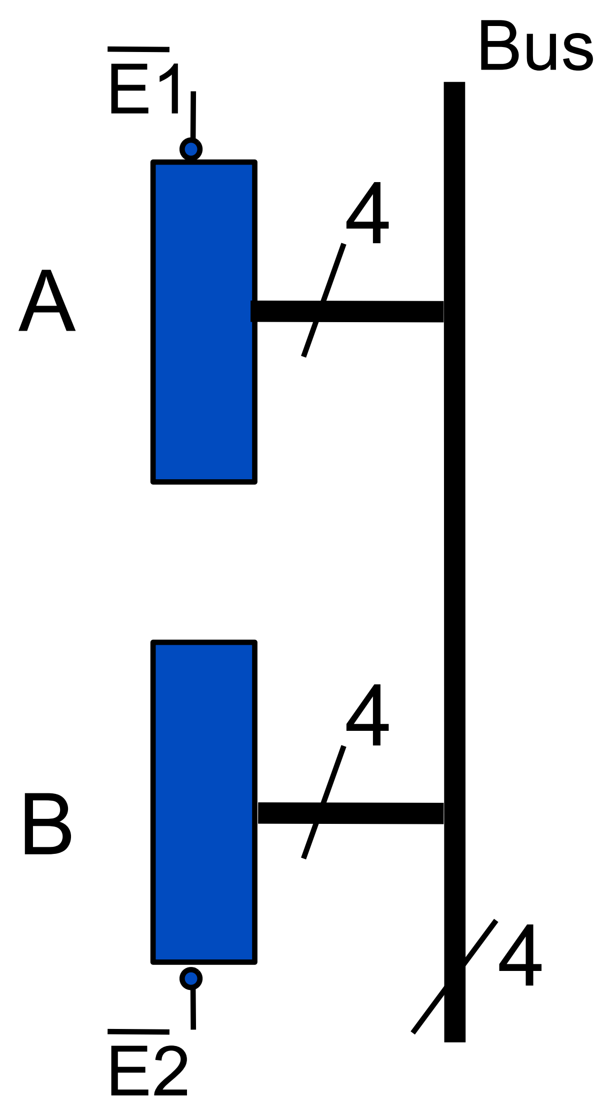
We can consider a 4-bit bus



When $\overline{E1}$ = 0,
A will be placed on the Bus

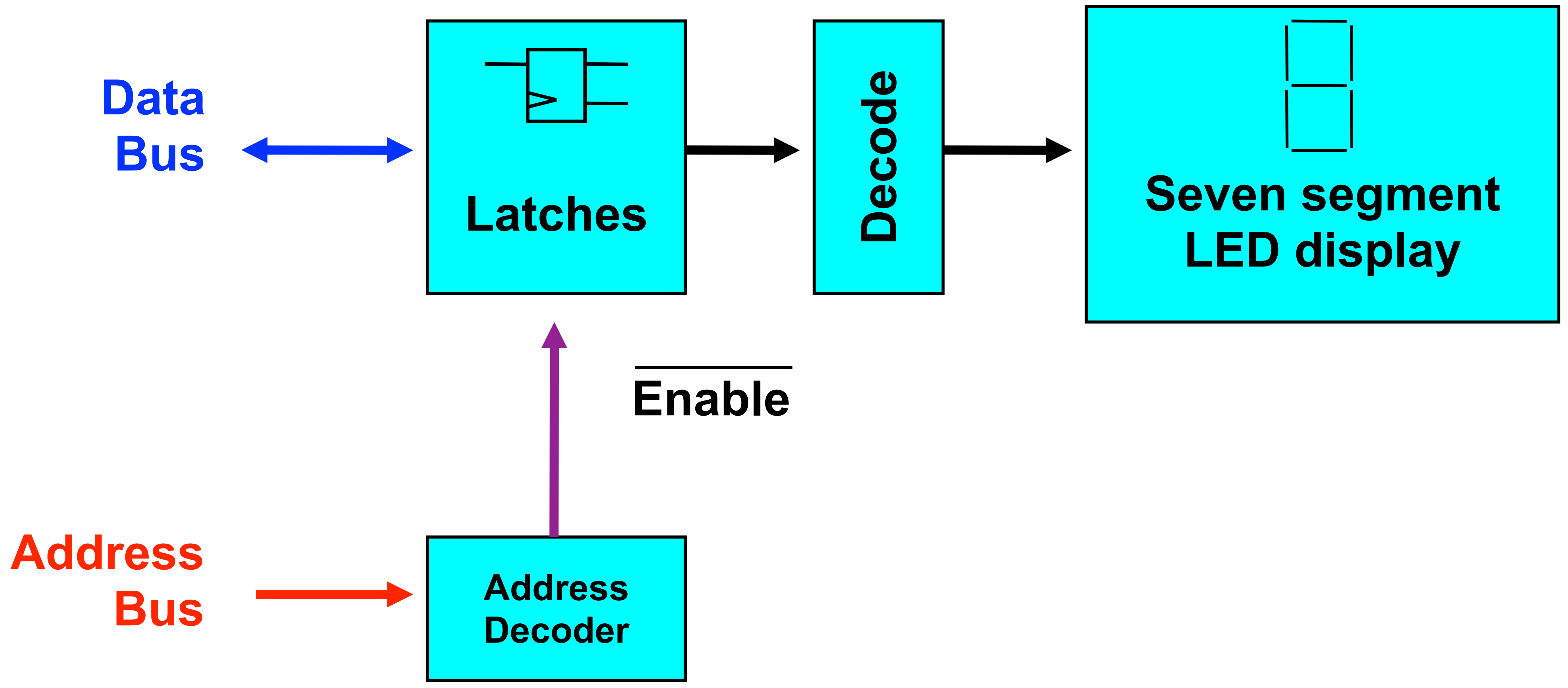
When $\overline{E2}$ = 0,
B will be placed on the Bus

Or, can represent this more concisely:



Output Example

Again, we've previously seen the principles underpinning the diagram below in relation to digital logic circuits



Memory-mapped I/O

The same address bus is used to address both memory and I/O devices

Memory (including registers) on I/O devices are mapped to address values

- An I/O device can then operate as designed on the data given

When a CPU accesses a memory address, that memory address may be in physical memory (typically RAM) or be associated with the memory of an I/O device

Memory-mapped I/O Advantages

- Simpler than many alternatives
  - Particularly compared to port I/O (where a dedicated class of instructions are defined for performing I/O)
  - CPU requires less internal logic, which can help make the design and fabrication of a CPU cheaper
- Use of general purpose memory instructions
  - All addressing modes supported by a CPU are available to I/O

Memory-mapped I/O Disadvantages

Portions of memory address space must be reserved

Less of a concern as 64-bit processors, hence address spaces, have come to market

Still relevant where 16-bit (and sometimes 32-bit) processors are used, e.g., embedded and legacy systems

Synchronising with I/O Devices - Polling

Motivating Synchronisation

Most I/O devices are much slower than the CPU

Several factors to consider

- Read - Is there data to be read from the device?
- Write - Is the device ready to accept data?

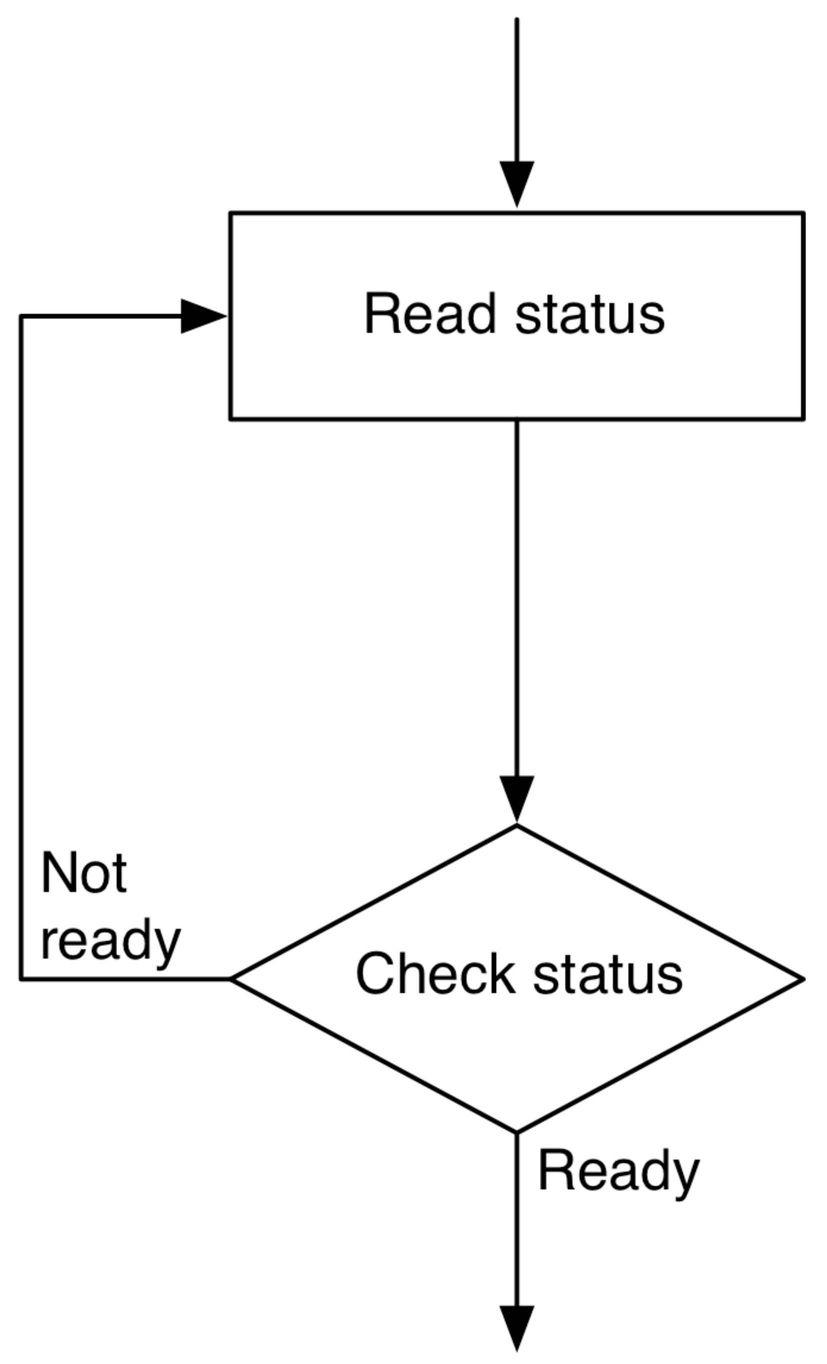
Consider a simple printer that takes a finite amount of time to print a character

- This time very much greater than the processor instruction time

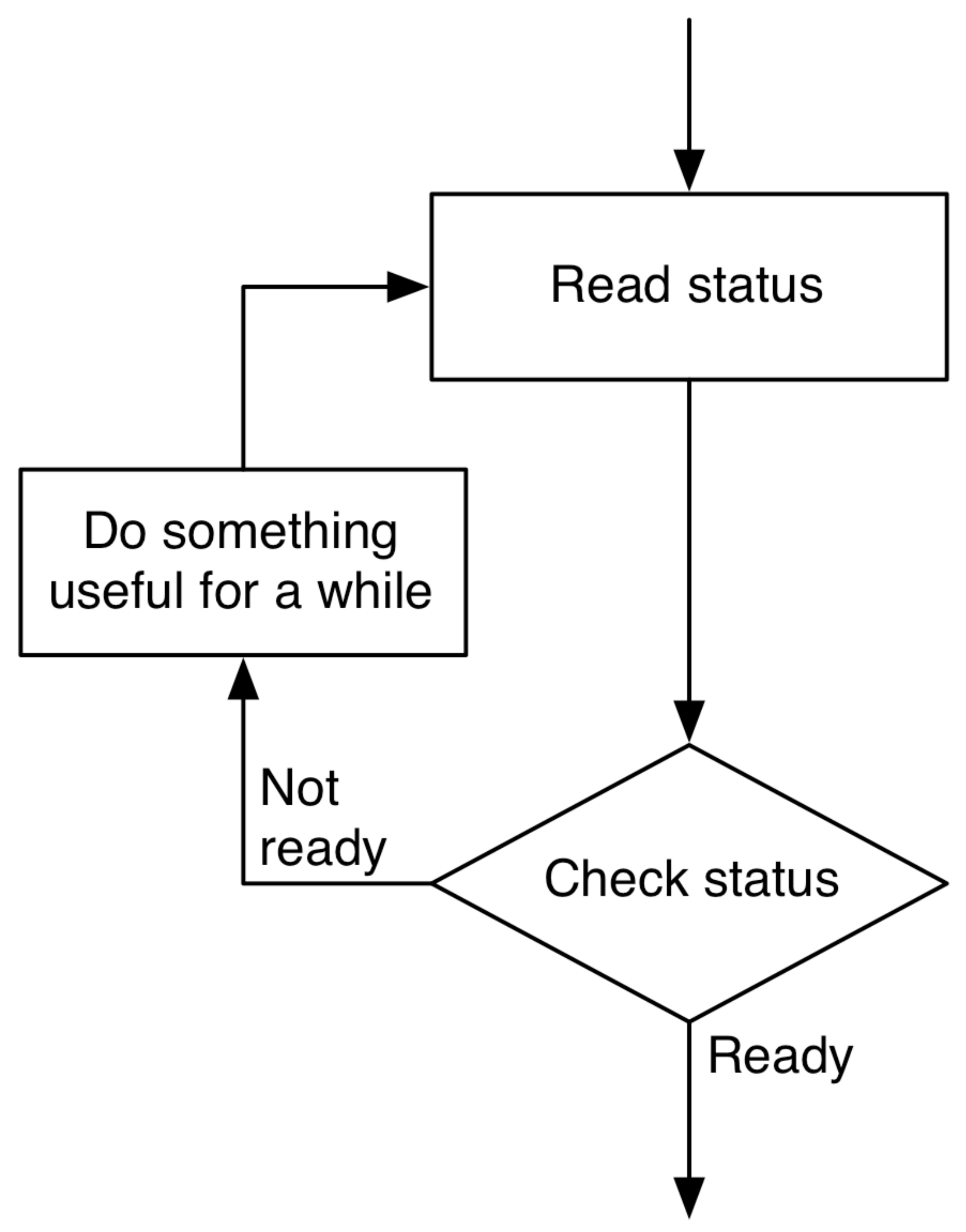
Polling and “Busy-wait” Polling

What are the practical advantages of these polling techniques?

What might we want to do whilst waiting for change in status?



Busy-wait polling



Polling, interleaved with another task

Polled I/O Advantages

- Simple software
    - A looping construct paired with some known checks
- Simple hardware
    - Support for notion of “ready” is all that is required

Polled I/O Disadvantages

Busy-wait polling wastes CPU time and consumes power

- Imagine you have a power constrained device, would this be where you want to spend your power?

Polling when interleaved with other tasks can lead to a significantly delayed response to device

- Not a problem for some I/O situations but a serious issue if you're working in a hard real-time context

But sometimes it is good enough!

Synchronising with I/O Devices - Handshaking

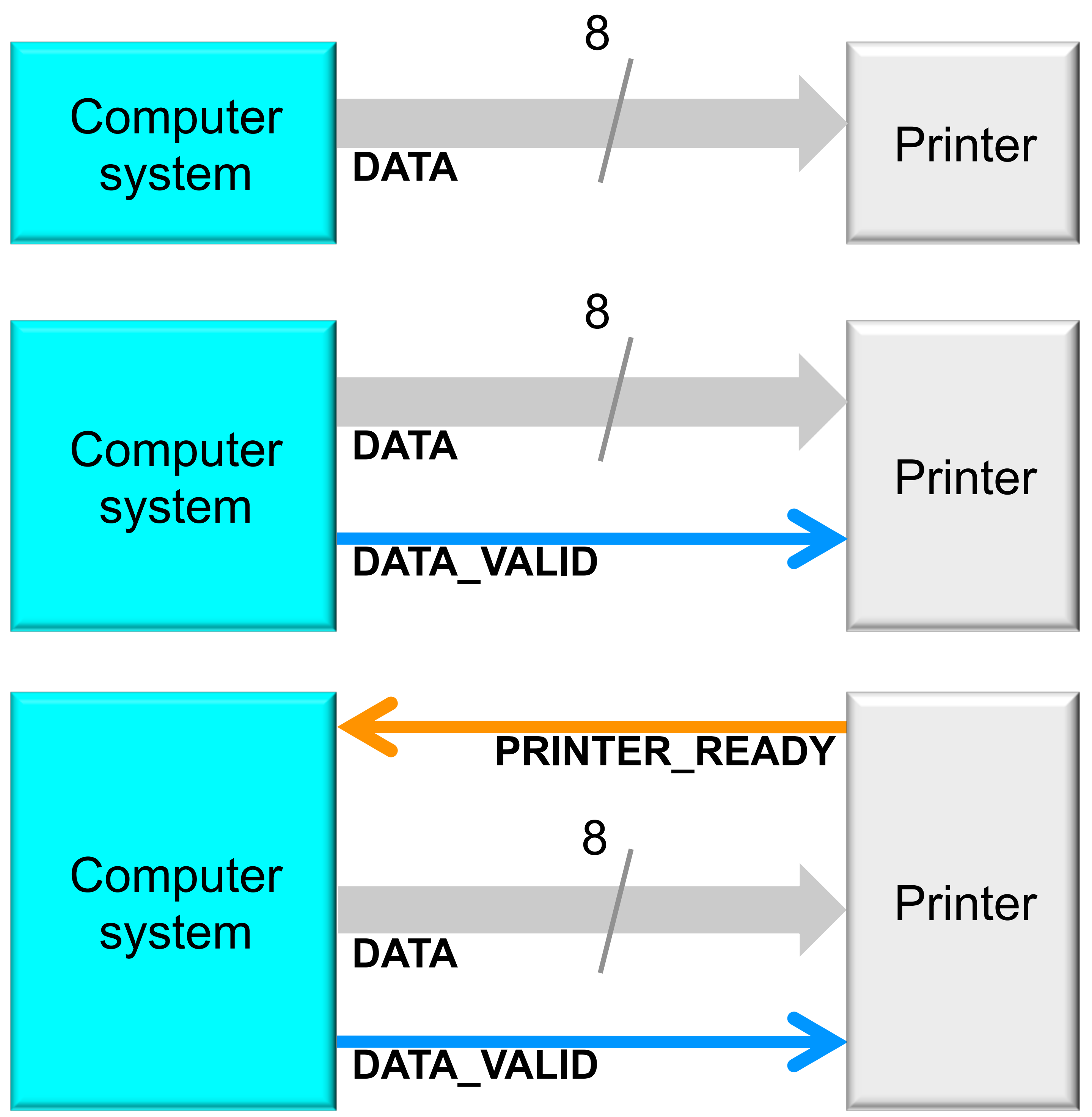
Motivating Handshaking

It seems that we're adding complexity - why?

Unsynchronised

Open-ended

Closed-loop
(Handshaking)

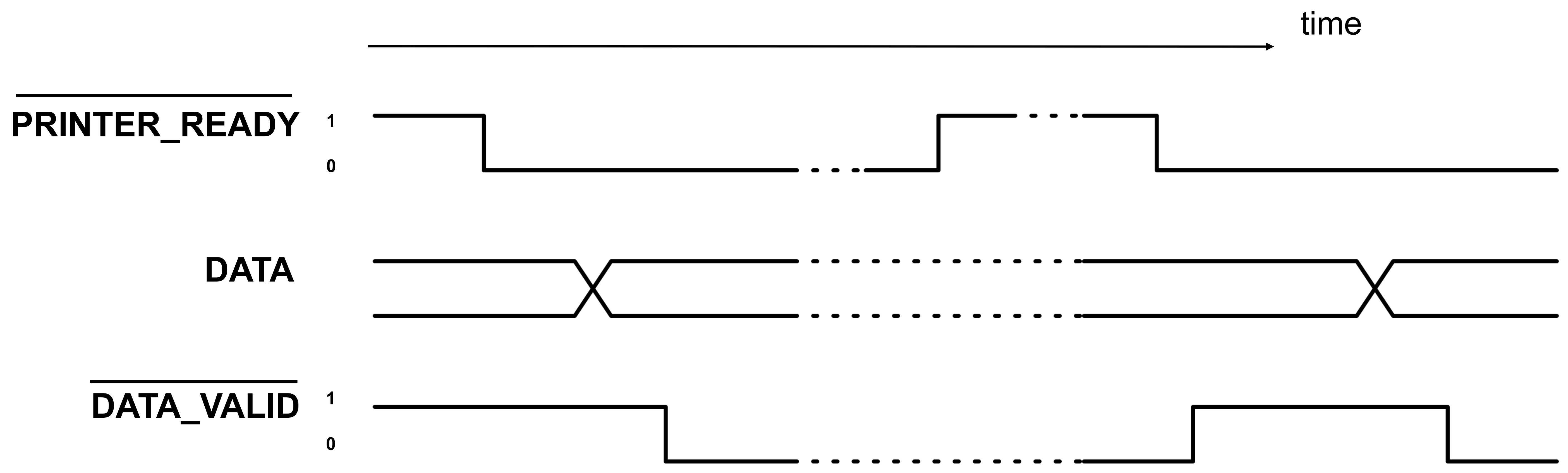


Handshaking - Timing Diagram

Computer system responds to the printer being ready by placing a new character on the data bus and signalling DATA_VALID

We could write code to get the CPU to do this

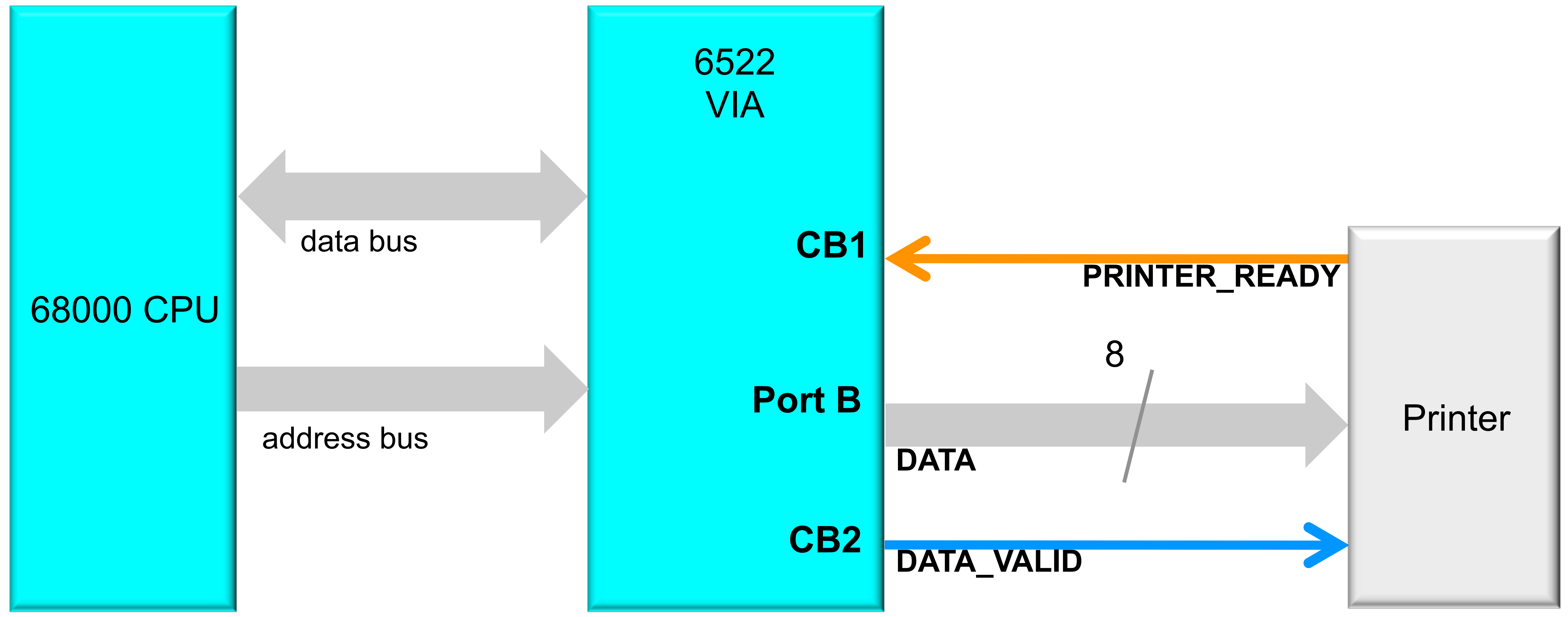
Alternatively, we could use the specialised hardware, which often requires fewer CPU instructions


time
$\overline{\text{PRINTER\_READY}}$
1
0
DATA
$\overline{\text{DATA\_VALID}}$
1
0

Example - Using the 6522 VIA for Handshaking

The 6522 Versatile Interface Adapter (VIA) is a (relatively old) I/O IC that supports handshaking - this is just an example!

This is more intuitive than it might seem - consider how you'd act out handshaking



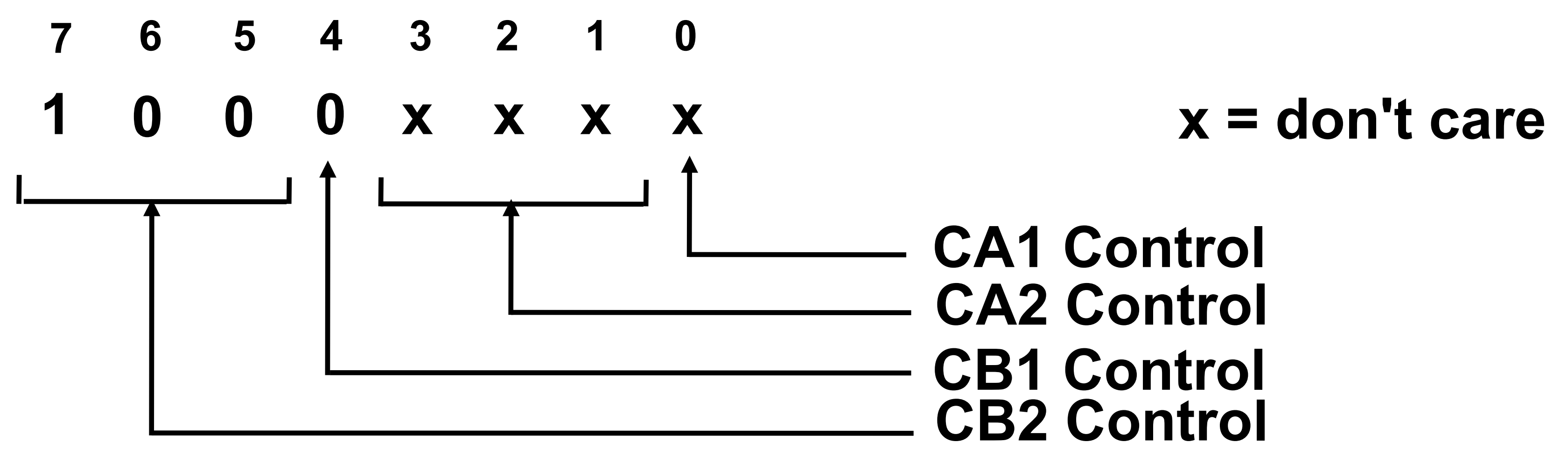
Setup for 6522 VIA Handshaking

There is a direct relationship between the diagram below and the block diagram we analysed previously

Note how setting bit values in particular locations allows us to determine function

Use: **PORT B as Output**
CB1 control line as PRINTER_READY
CB2 control line as DATA_VALID

Need to set up Peripheral Control Register (PCR) on VIA



How the 6522 VIA Performs Handshaking

PCR 1000xxxx

> “CB2 handshake output mode”
>
> “CB1 Interrupt Flag set by negative transition on CB1”

When PRINTER_READY is asserted, i.e., goes high to low

> VIA sets IFR4, which can be later read by CPU
>
> VIA sets DATA_VALID high, i.e., invalid

When CPU writes a value to ORB

> VIA sets DATA_VALID to low (i.e., valid)

In this example, the CPU will repeatedly poll IFR to see if PRINTER_READY - we are not (yet) using interrupts.

Busy-wait Printing Program with Handshaking

```
| Print characters A to Z on hypothetical printer connected as per previous slides.
MAIN:       clr.b     VIA_DDRA                          | set PORT A = inputs
            move.b #%11111111,  VIA_DDRB                | set PORT B = outputs
            move.b #%10000000,  VIA_PCR                 | initialise PCR
            move.b #%01111111,  VIA_IER                 | disable VIA interrupts
            move.b #$41, D0                             | start with ASCII ‘A’ = $41

| now… when PRINTER_READY goes high to low, VIA sets DATA_VALID=invalid, and also sets IFR bit 4: check if so…
LOOP:       move.b VIA_IFR, D1                          | read VIA interrupt flag byte, put it in D1
            and.b  #%00010000, D1                       | is bit 4 set?
            beq    LOOP                                 | if clear then not PRINTER_READY: loop
            move.b D0, VIA_ORB                          | PRINTER_READY, so output character
                                                        | (this write to ORB causes VIA to set DATA_VALID=valid)
                                                        | printer will eventually set PRINTER_READY high again
            add.b  #1, D0                               | increment to next character
            cmp.b  #$5B, D0                             | are we at the end of the alphabet…
      bne   LOOP                                        | …(ASCII ‘Z’ = $5A)? If not, do it again
```

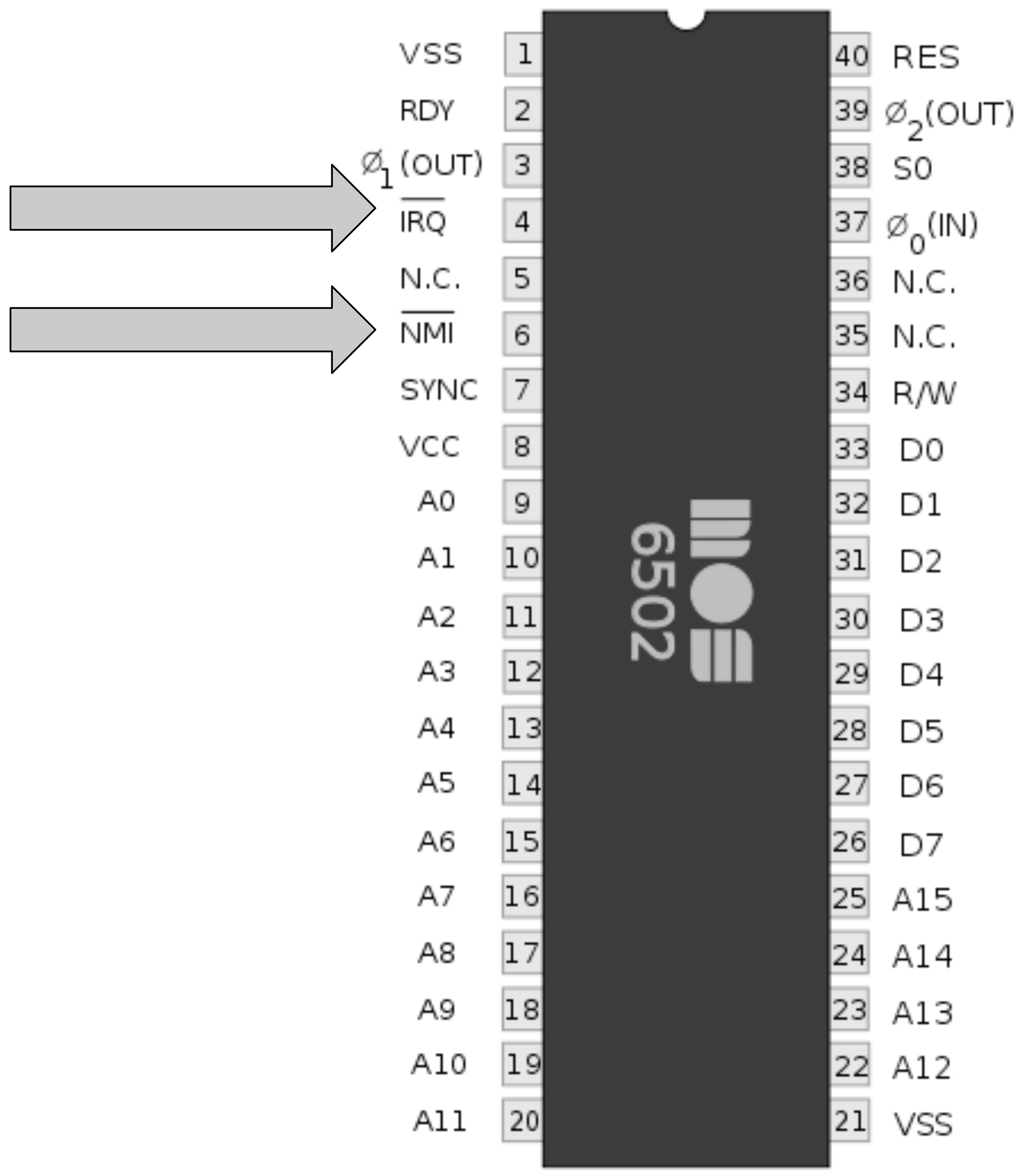
Synchronising with I/O devices - Interrupts

Interrupts - An Input on the CPU

Consider the 6502 processor, designed in 1975, used in the BBC micro and others

IRQ (Interrupt Request) and NMI (Non-maskable Interrupt) inputs

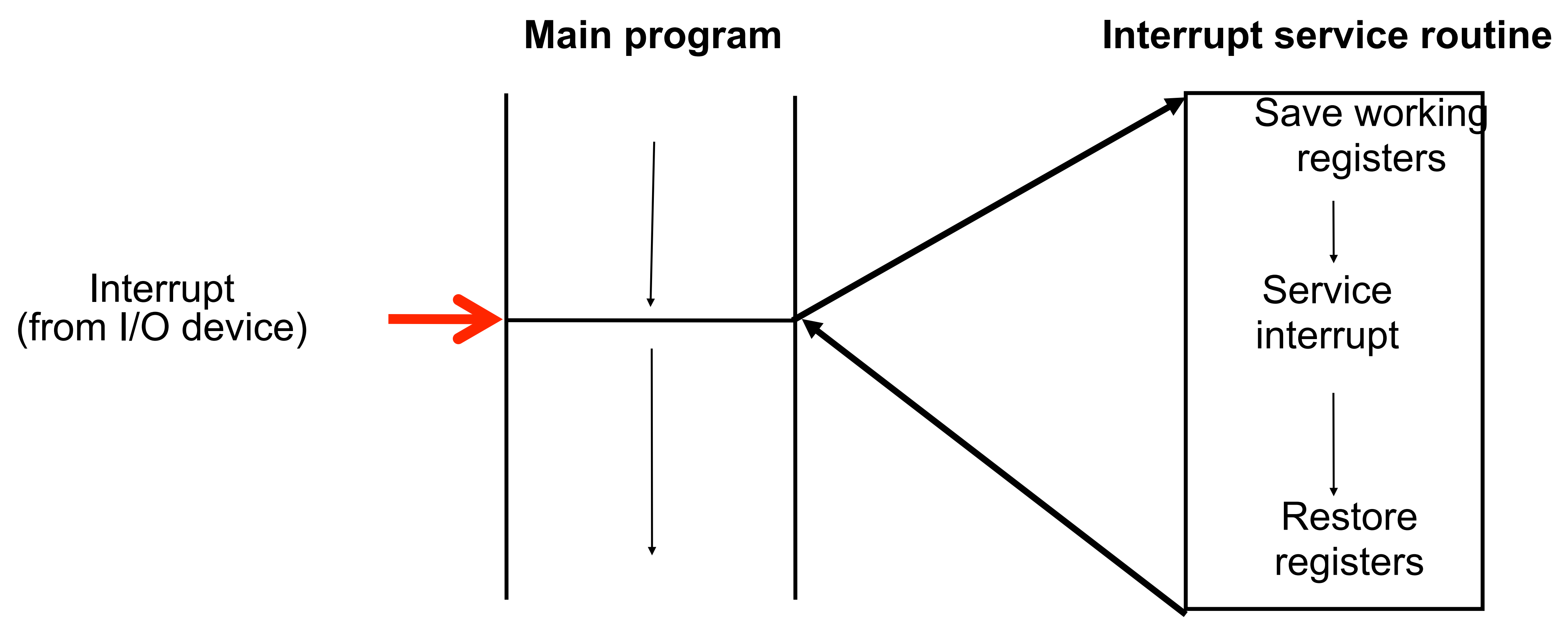
Code can disable response to IRQ, as it is only a request, but NMI cannot be disabled or "masked out"



Effect of an Interrupt Input

CPU normally executes instructions sequentially, unless a jump or branch is made

An interrupt input can force CPU to jump to a service routine



Can therefore make it appear (to humans) that the CPU is performing two or more tasks "simultaneously"

Interrupt Handling Sequence

1. External device signals interrupt

INTERRUPT RESPONSE

1. CPU completes current instruction
2. Push PC onto Stack
3. Push Status Register(s) onto Stack

   Context switch

4. Load PC with address of Interrupt Handler

RETURN FROM INTERRUPT

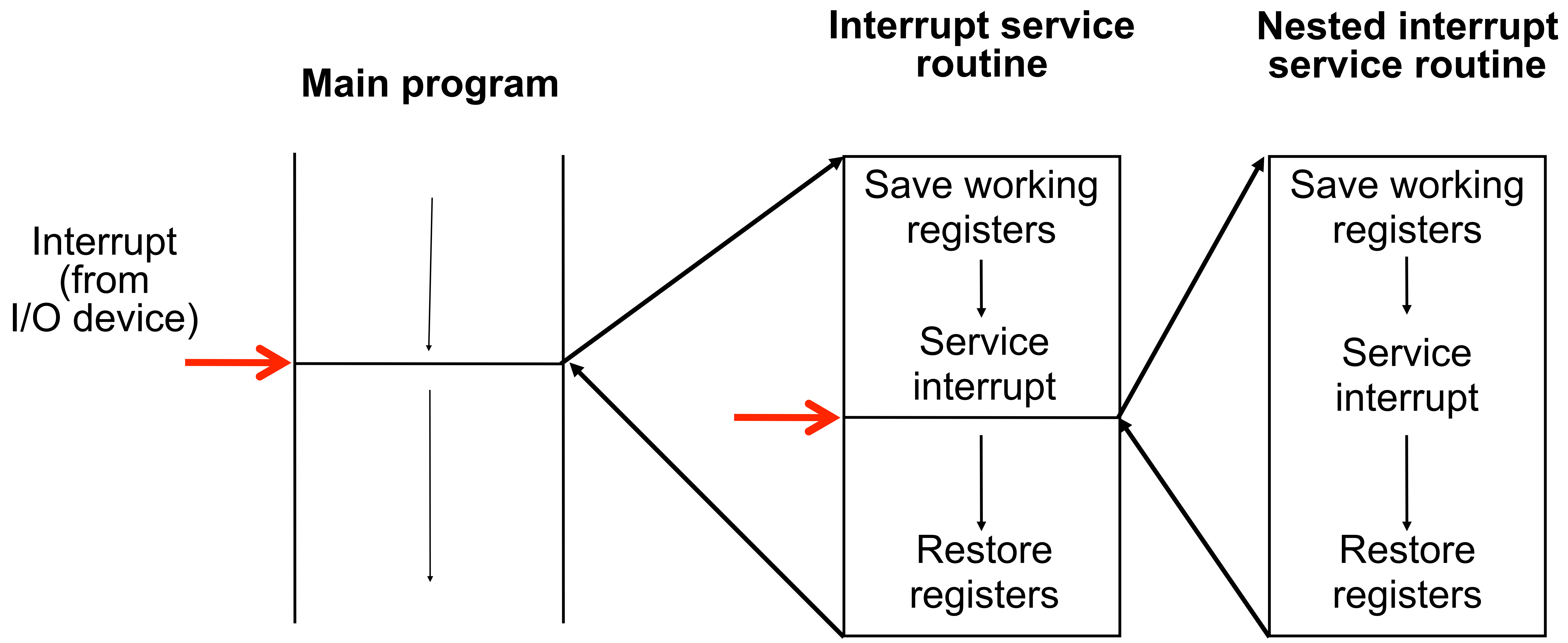
1. Pop PC from Stack
2. Pop Status Register(s) from Stack

   Context switch

3. Load PC with popped return address

Nested Interrupts

An interrupt service routine can be interrupted



Interrupts for I/O Examples

Switches can be connected to IRQ

- If using multiple switches we can OR them all together to form IRQ and then check the individual switch states within the service routine

A hard drive can generate an interrupt when data, requested some time earlier, is ready to read

A timer can generate an interrupt every 100ms and the service routine can then read an sensor input

A printer can generate an interrupt when it is ready to receive the next character to print, as we will see

Interrupts - Advantages and Disadvantages

Advantages

- Fast response
- No wasted CPU time / battery power

Disadvantages

- All data transfers still controlled by CPU
- More complex hardware and software

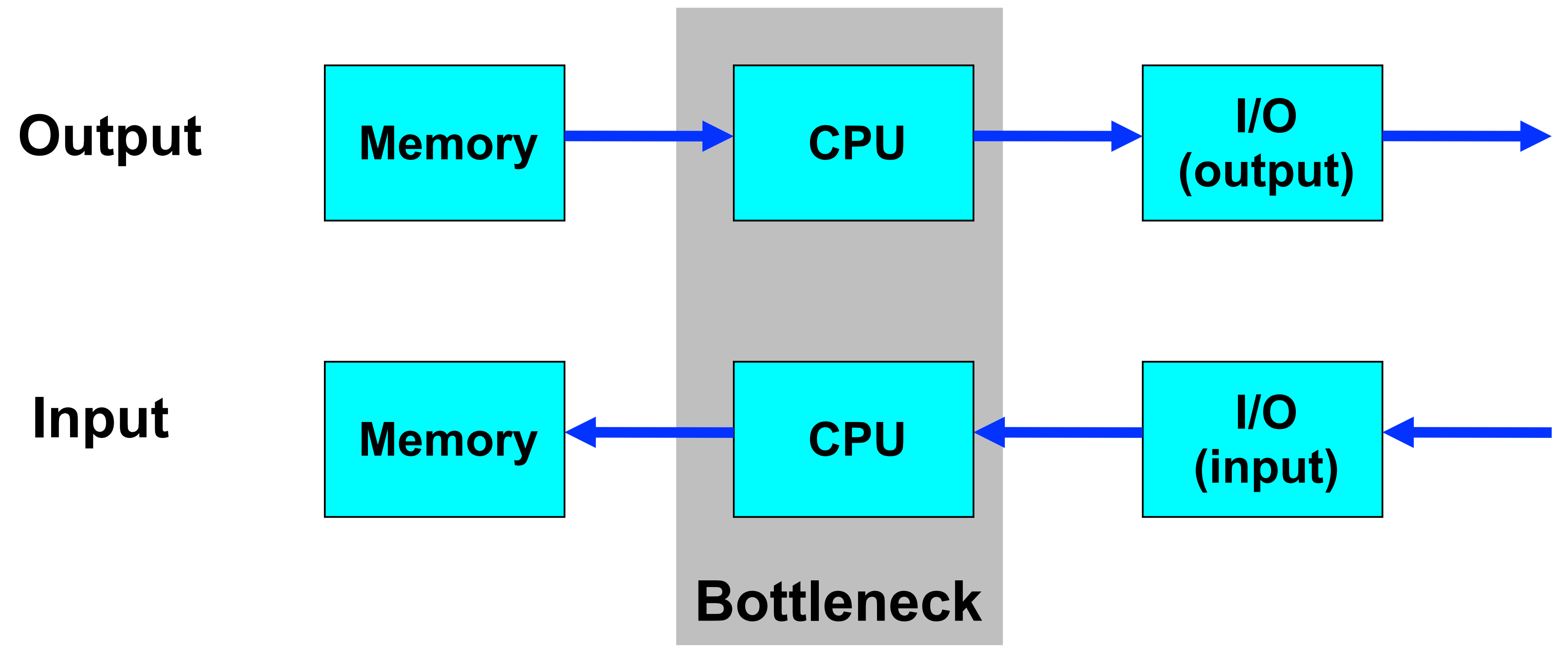
Understanding I/O mechanisms is vital - no single best!

Direct Memory Access (DMA)

Motivating Direct Memory Access (DMA)

The CPU is a bottleneck for I/O

Programmed I/O techniques, as in all the examples we’ve seen so far, are slowed down by the CPU



DMA avoids the CPU bottleneck, thus speeding up the transfer of data to memory

Why Use DMA?

DMA is used where large amounts of data must be transferred at high speed

Control of the system buses is surrendered by the CPU to a DMA Controller (DMAC).

- The DMAC is a dedicated device that controls the three system buses during the data transfer.
- The DMAC is optimised for one operation, i.e., data transfer.
- The CPU is more general purpose, it both transfers data and is a processor of information.

DMA-based I/O can be more than 10 times faster than CPU-driven I/O

DMA Operation

1. DMA transfer requested by I/O
2. DMAC passes request to CPU
3. CPU initialises DMAC

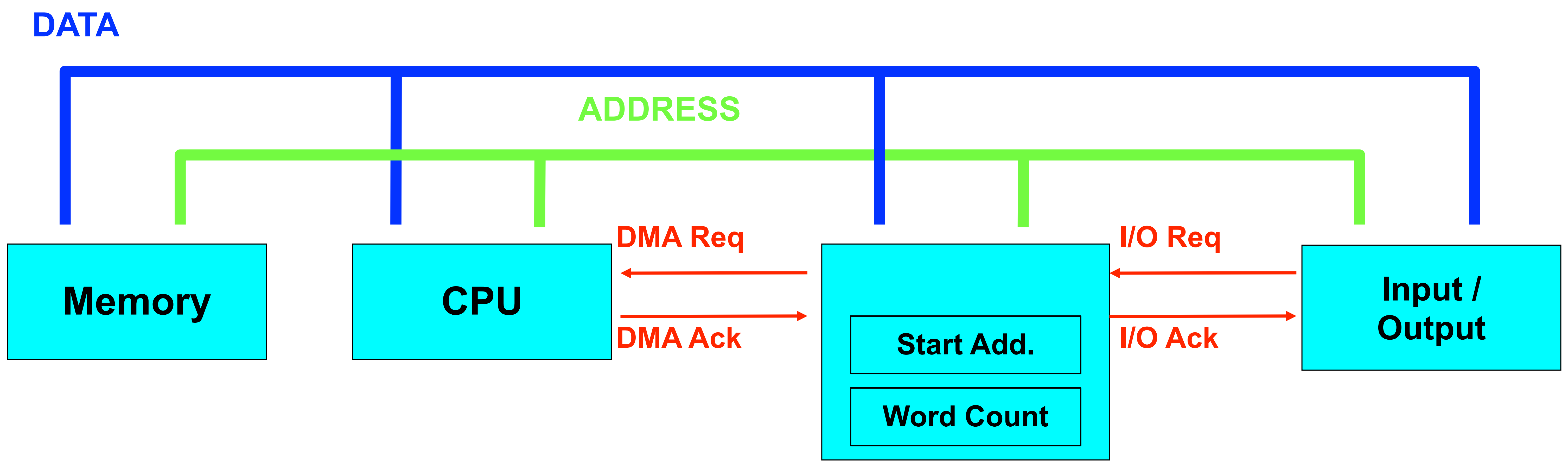
    i) Input or Output,

    ii) Start address → DMAC Address Reg,

    iii) Number of words to transfer → Count Reg,

    iv) CPU enables DMAC

4. DMAC requests use of system buses
5. CPU responds with DMA Ack when it's ready to surrender buses



DMA Modes of Operation

Cycle Stealing

DMAC uses the system buses when they are not being used by the CPU - usually by “grabbing” available memory access cycles not used by the CPU

Burst Mode

DMAC requires system buses for extended transfer of large amount of data at high speed and “locks” the slower CPU out of using the system buses for a fixed time or until the transfer is complete or the CPU receives an interrupt from a device of greater priority

DMA Organisation

Possible to consider multiple organisations for the incorporation of DMA

Single bus, detached DMA

All modules (DMA, I/O, memory and processor) share the same system bus

The DMA module, mimicking the processor, uses programmed I/O to exchange data between memory and an I/O module through the DMA module

This implementation is straightforward to implement for most bus-based systems but is inefficient

As with processor-controlled I/O, each word transfer takes two bus cycles

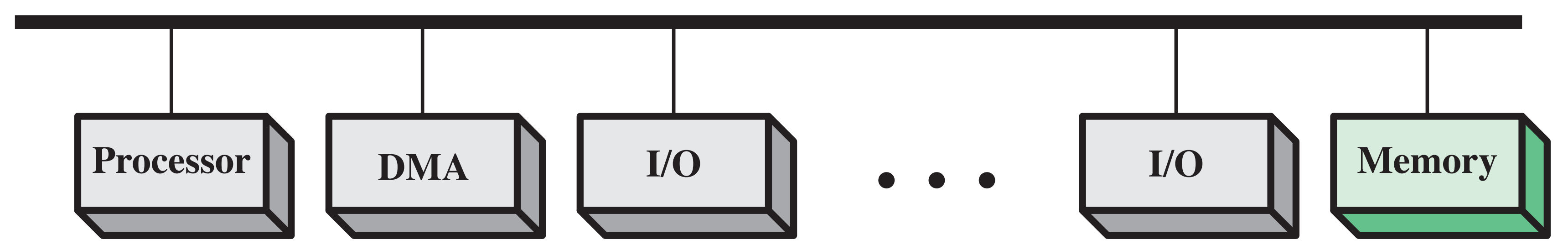
DMA Organisation

I/O Bus

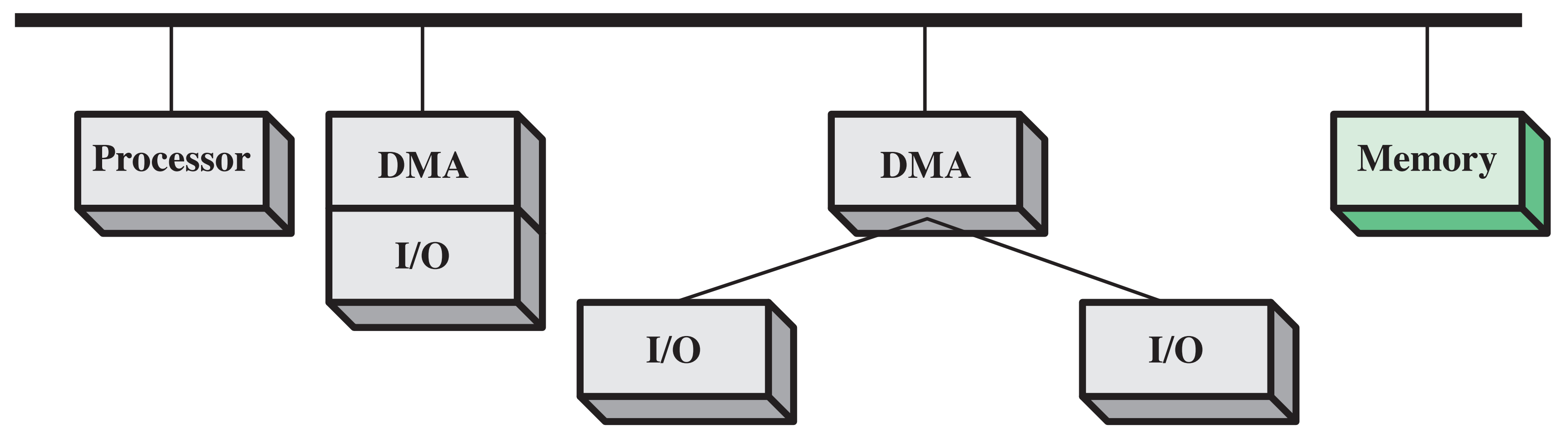
Reduces number of I/O interfaces in the DMA module to one

System bus the DMA module shares with the processor and memory is used only to transfer data to and from memory

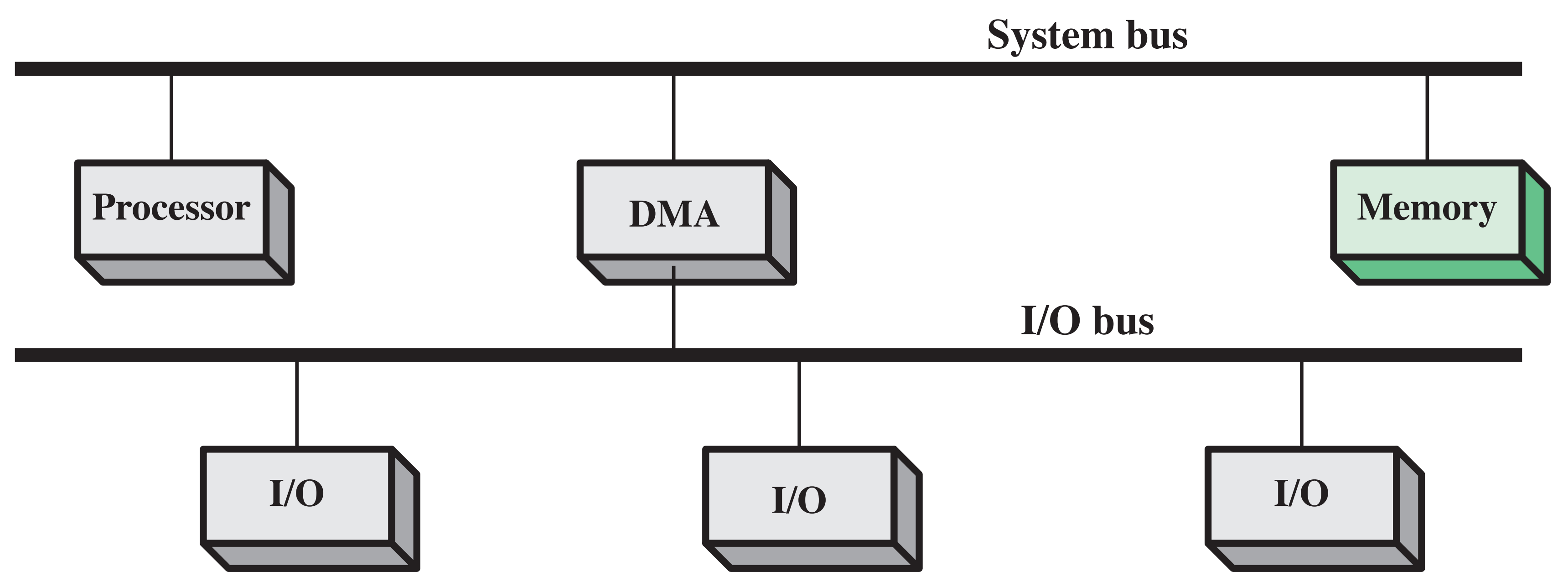
Reproduced from: W. Stallings, “Computer Organization and Architecture”, 9th Edition, Pearson, 2013



(a) Single-bus, detached DMA



(b) Single-bus, Integrated DMA-I/O



(c) I/O bus

I/O Mechanisms

Memory-mapped input and output devices can be accessed in the same way as RAM, at special address locations

Polled I/O is a technique for scheduling input and output, where the CPU repeatedly checks if this is necessary

I/O devices are usually much slower than the CPU, so **handshaking techniques** must be used to coordinate CPU and device to synchronise the transfer of data

Interrupts are a way to avoid polled I/O by diverting the CPU to a special I/O routine – quickly and only when necessary

If necessary, a **DMA** controller can be used instead of the CPU to transfer I/O data into/from memory - this can be faster than the CPU but at additional hardware cost

Which technique to use? Depends on amount of data, required transfer speed, budget, power, physical size, complexity - this is **computer system design**