Sistemas Operativos: Input/Output
Intro and HW/SW Interface

Pedro F. Souto (pfs@fe.up.pt)

April 14, 2012

Agenda

Introduction

I/O Controllers and I/O Buses

Modes of Data Transfer

Additional Reading

Agenda

Input/Output Devices

- Wide variety of I/O devices

Device	Type	Data rate
Keyboard	Human-interface	10 byte/s
Mouse	Human-interface	100 byte/s
Modem	Communication	56 kbit/s
ISDN line	Communication	128 kbit/s
Laser printer	Human-interface	100 kbyte/s
Ethernet	Communication	10 Mbit/s
USB	Bus	12 Mbit/s
40× CD-ROM	Storage	6 Mbyte/s
Fast Ethernet	Communication	100 Mbit/s
EIDE (ATA-2)	Storage	16.7 Mbyte/s
XGA Monitor	Human-interface	60 Mbyte/s
Gigabit Ethernet	Communication	1 Gbit/s
PCI bus	Bus	528 Mbyte/s
HyperTransport Bus	Bus	25.6 Gbyte/s

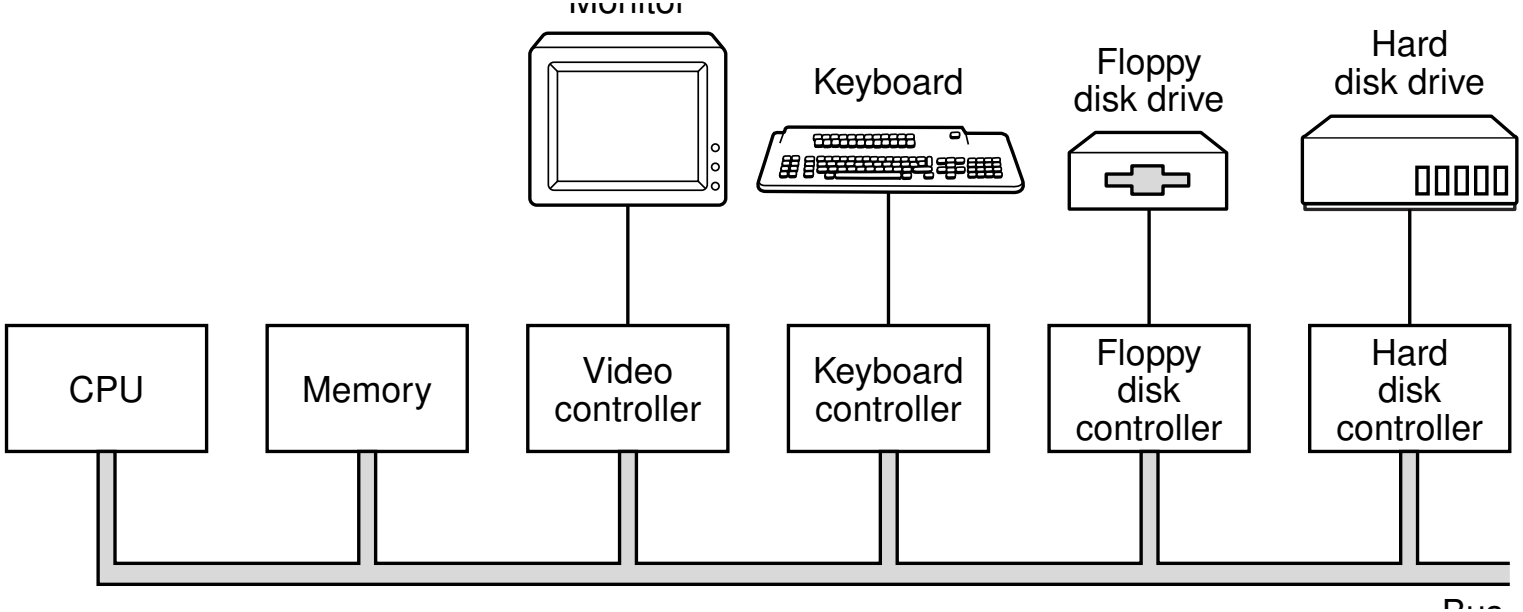
Agenda

I/O Controllers (1/3)

- An I/O controller is an electronic component that
  - Controls the operation of the I/O device
  - Some examples:
    - Network card
    - Video card
    - Hard disk controller
    - UART (serial port controller)
- The OS interfaces with the I/O controller
  - It is also known as **adapter**

  but not with the device itself
  - Nevertheless, the developer of a **device driver** needs to have a fairly detailed knowledge of the device's operation

I/O Controllers (2/3)



- I/O controllers have 4 sets of registers

  Control registers for configuring and controlling the device's operation

  Status registers for monitoring the state of the device and of the operations it performs

  Input data registers (or buffers) for data transfer from the device

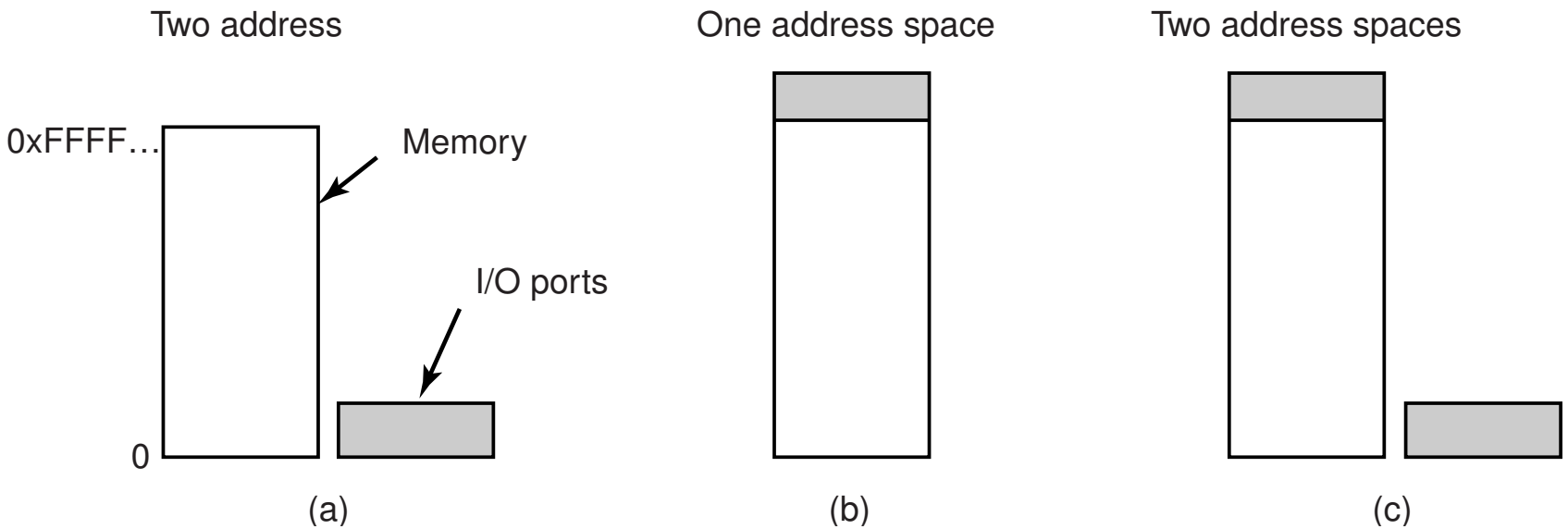
  Output data registers (or buffers) for data transfer to the device

I/O Controllers (3/3)

- Access to these registers may be via:

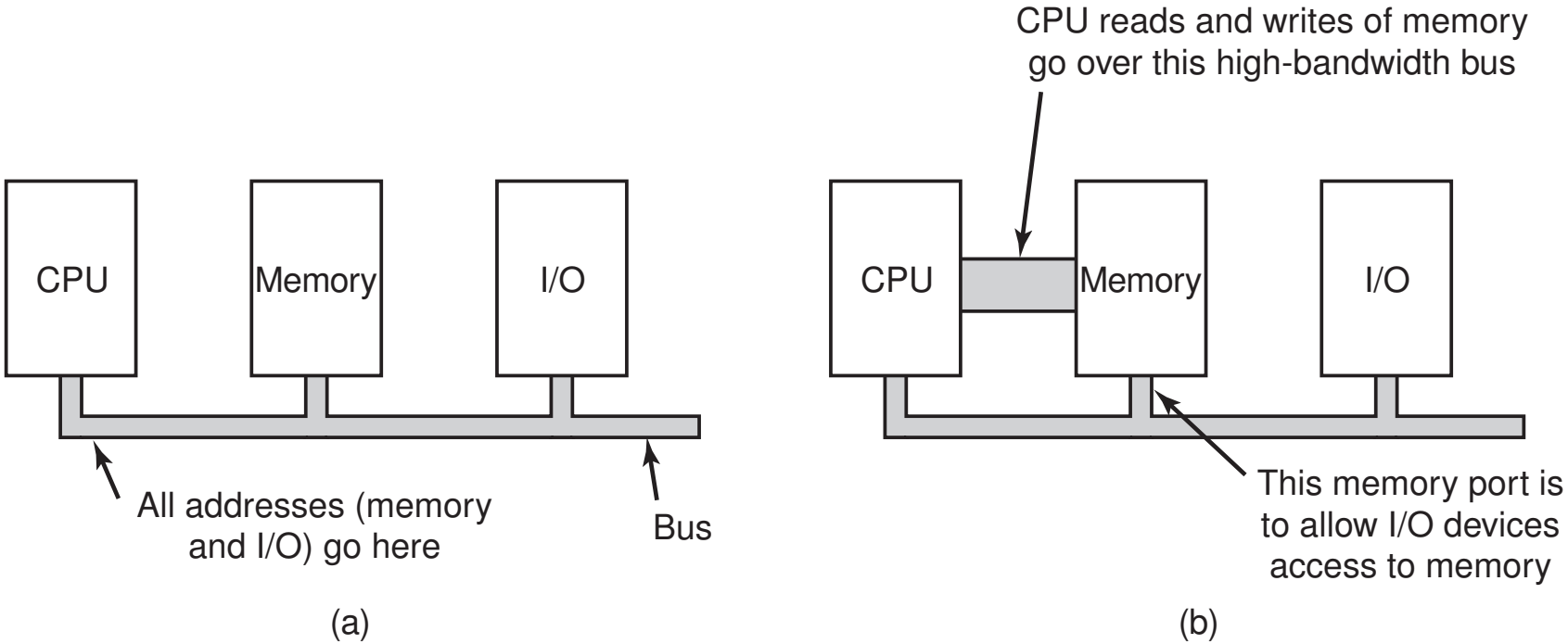
  Memory-mapped I/O
  - Allows to use any memory access instruction for I/O
  - There are some issues with caching and VM (to discuss later)

  I/O instructions such as `in/out`
  - The system uses different address spaces for memory and for I/O
  - Protection is simplified by making these instructions privileged



I/O Buses (1/3)

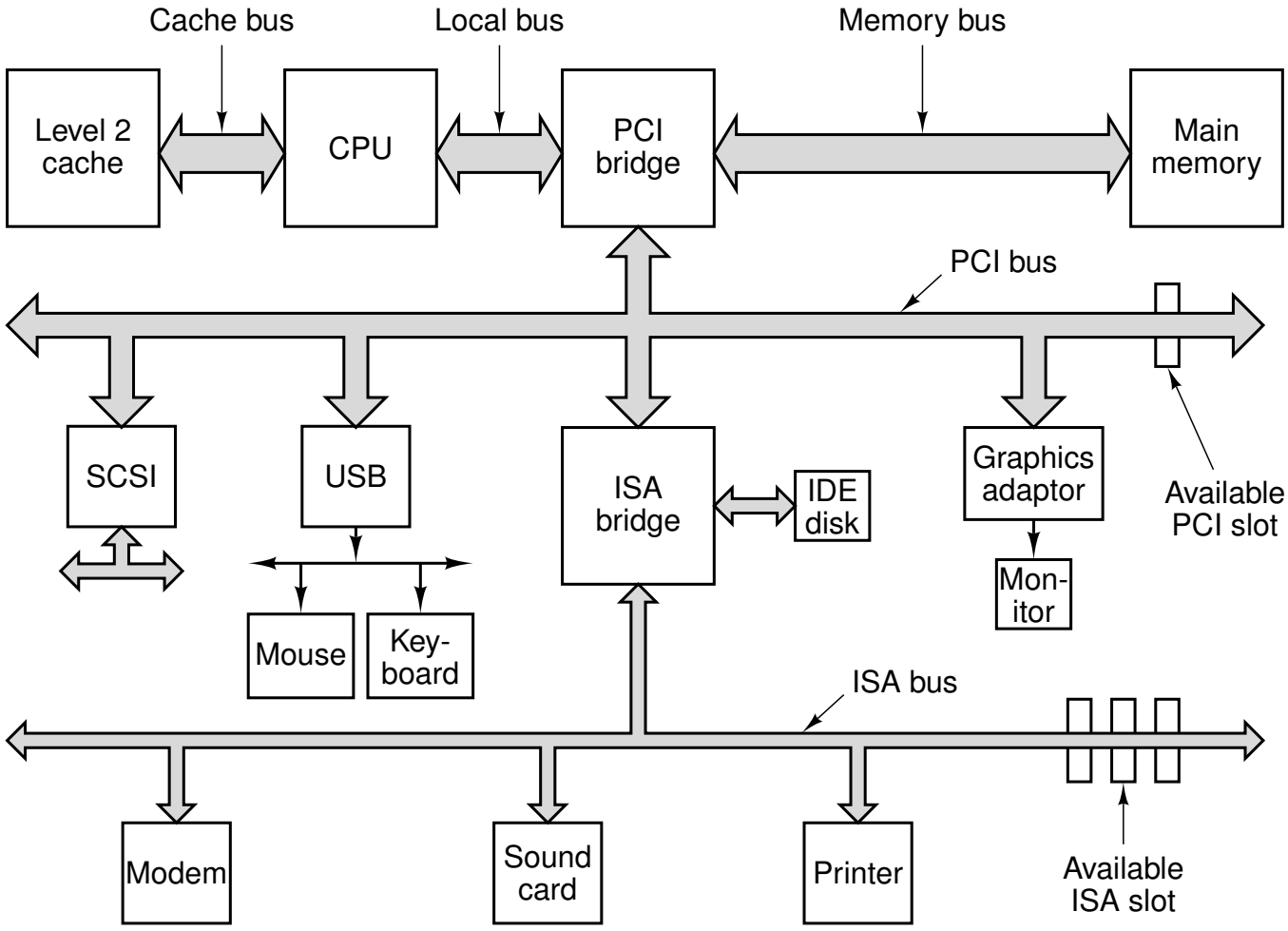
- Except for slow processors, I/O controllers and memory use different buses



  - Even, when the controllers are memory-mapped
    - In this case memory addresses have to be passed to the I/O bus
  - The bus on which a controller is, usually is not transparent to the **device driver** developer

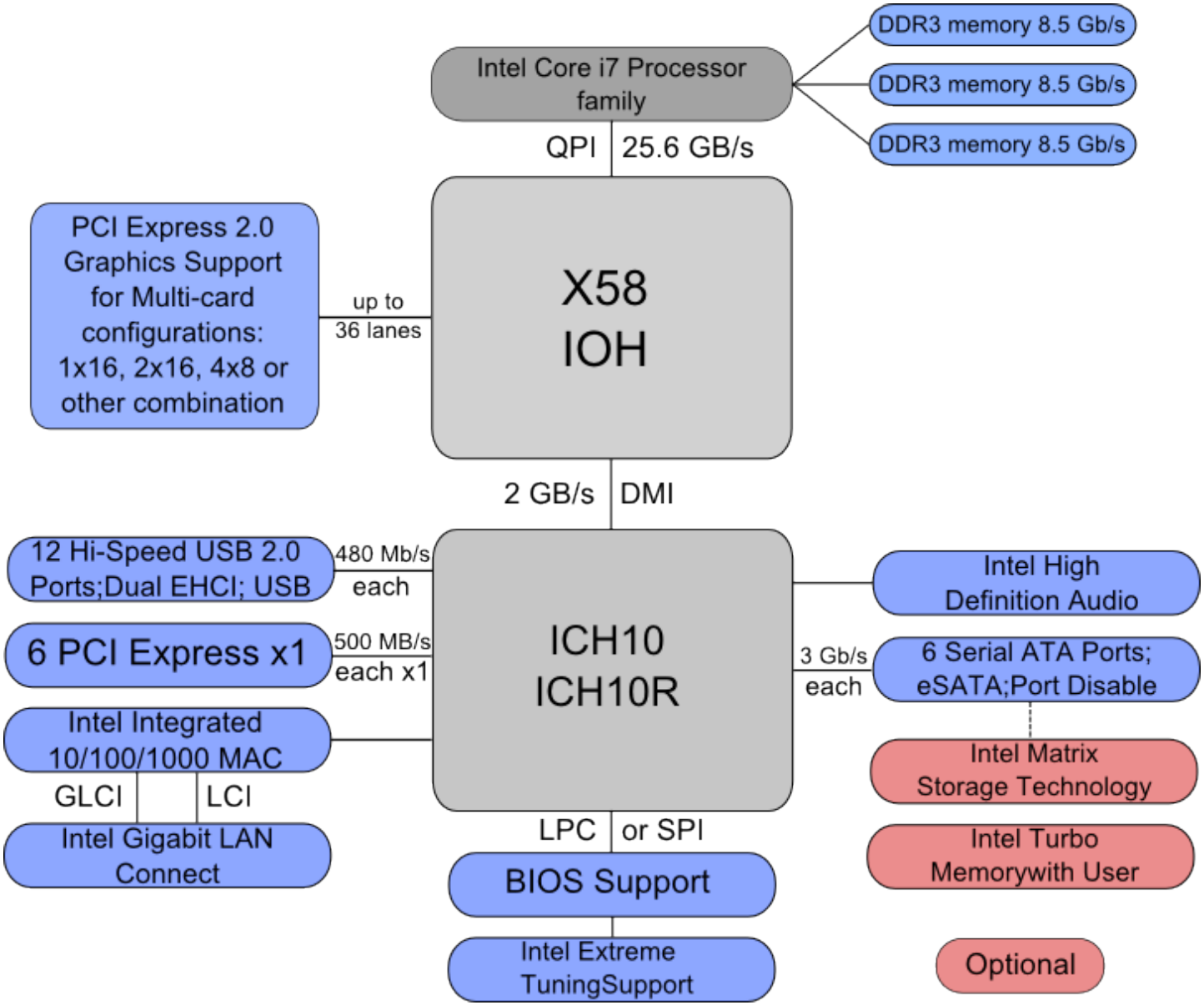
I/O Buses (2/3)

- On a PC, there are typically 3 types of buses:



  - It is up the PCI bridge (or better the north-bridge) to filter the addresses

I/O Buses (3/3)

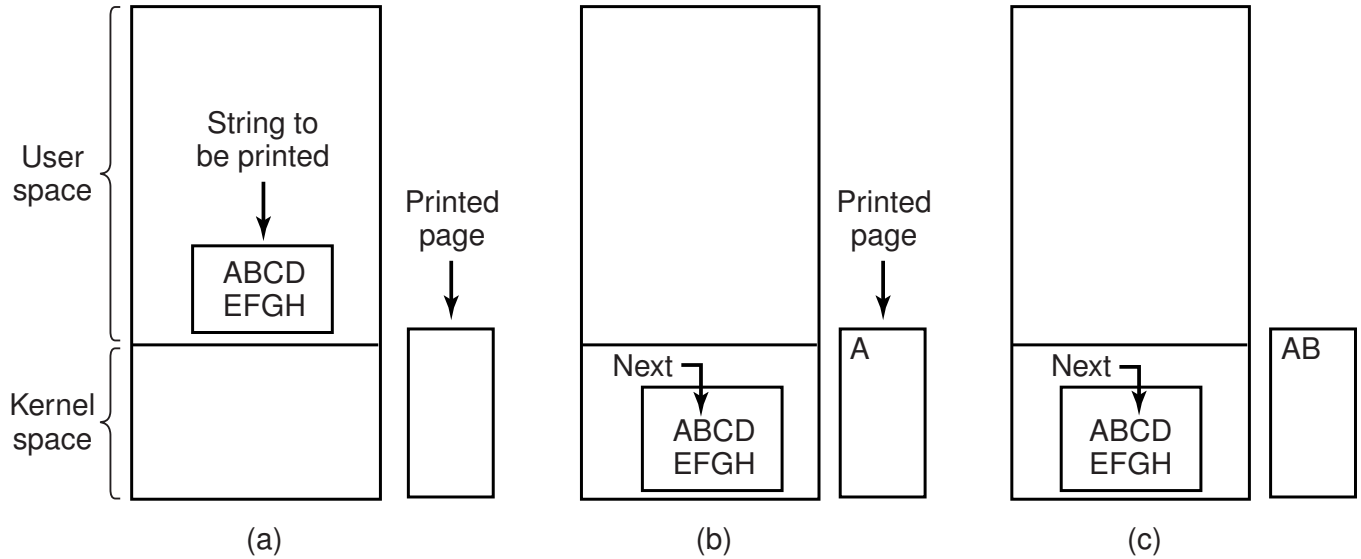


source: http://commons.wikimedia.org/wiki/File:X58_Block_Diagram.png

Agenda

Programmed I/O

- The CPU transfers the data between memory and the controller's data registers

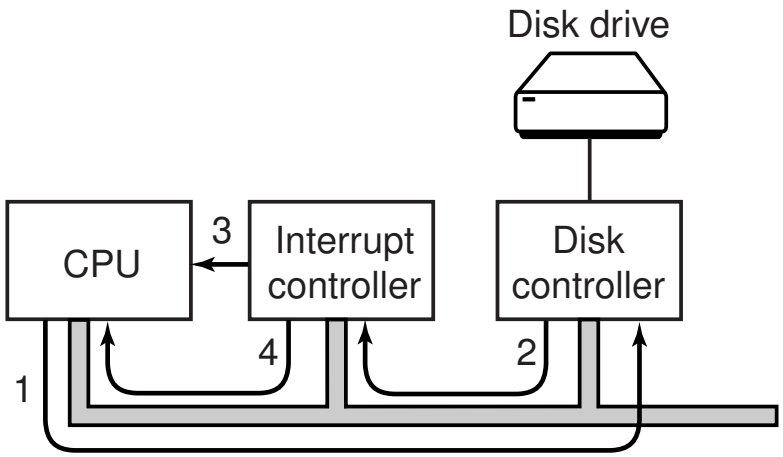


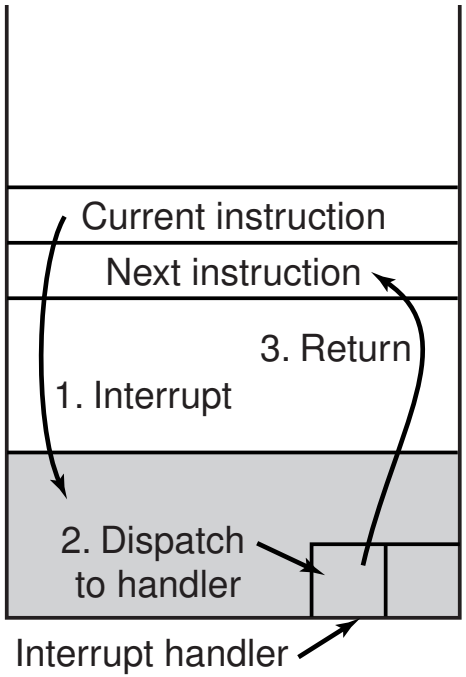

```
copy_from_user(buffer, p, count);   /* copy data to kernel */
for( i = 0; i < count; i++ ) {      /* for all characters */
   while(*printer_status_reg != READY); /* wait for printer */
   *printer_data_reg = p[i];        /* output next character */
}
```

- The CPU must **poll** the controller to find out if the device is ready for the next operation
  - Often, it must also check whether the operation succeeded

I.e. we have busy waiting

Interrupt-Driven I/0 (1/4)



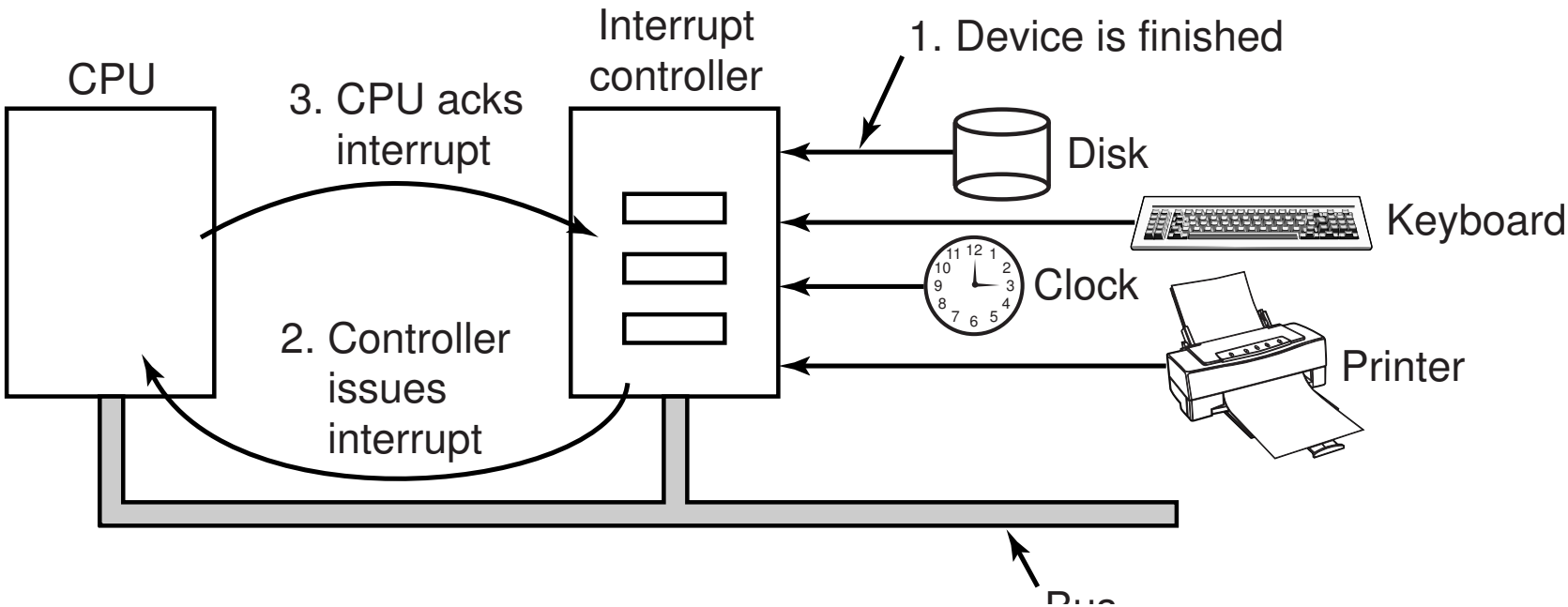


(a) (b)

- Relies on the processor's interrupt mechanism
- When the device
  - completes a command
  - detects an event (e.g. reception of a packet)

  it generates an HW interrupt

Interrupt-Driven I/0 (2/4)



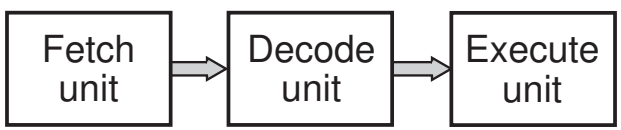
- So that interrupt processing is more efficient, many processors use **vectored interrupts**
  1. The CPU and the interrupt controller execute a HW handshake protocol
     - So that the CPU learns which device generated the interrupt
  2. The CPU then invokes the appropriate **interrupt handler**

Interrupt-Driven I/0 (3/4)

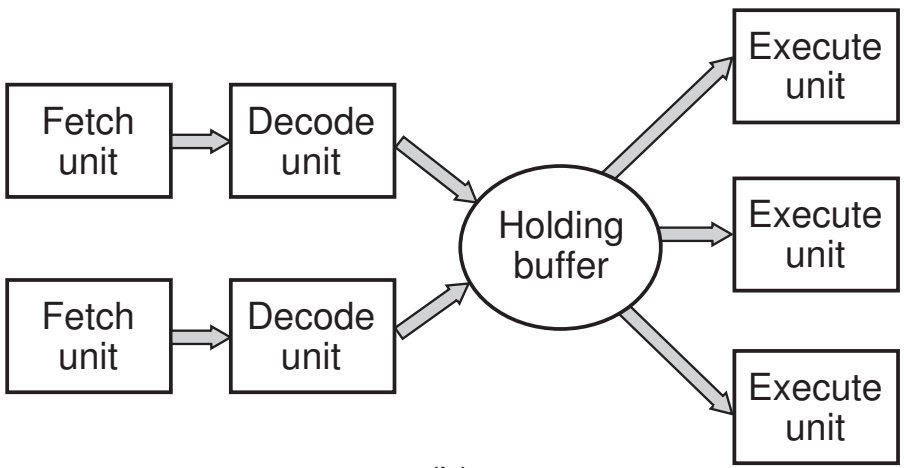
- Before executing the interrupt handler, the CPU automatically stores some of its state in the stack

  Question which stack should be used? The user's process stack?
  - The SP contents may not be valid
  - It may point to the "end" of a page, leading to a **page fault**
- Another issue concerns the **interrupt precision**
  - In pipelined or superscalar architectures, the CPU does not execute one instruction at a time, but several

  This is a HW feature with major implications on the OS



(a)



(b)

Interrupt-Driven I/0 (4/4)

- The processor programs the I/O controller to execute the operation

```
copy_from_user(buffer, p, count);     /* copy data to kernel */
while(*printer_status_reg != READY); /* wait for printer */
*printer_data_reg = p[0];           /* output first character */
scheduler();                          /* wait for I/O */
```

- The I/O controller generates an interrupt when the operation is done
- The interrupt handler does what must be done on that event

```
if( count == 0 ) {
   unblock_process();
} else {
   *printer_data_register = p[i];
   count--;
   i++;
}
acknowledge_interrupt();
return_from_interrupt();
```

I/O with DMA (1/5)

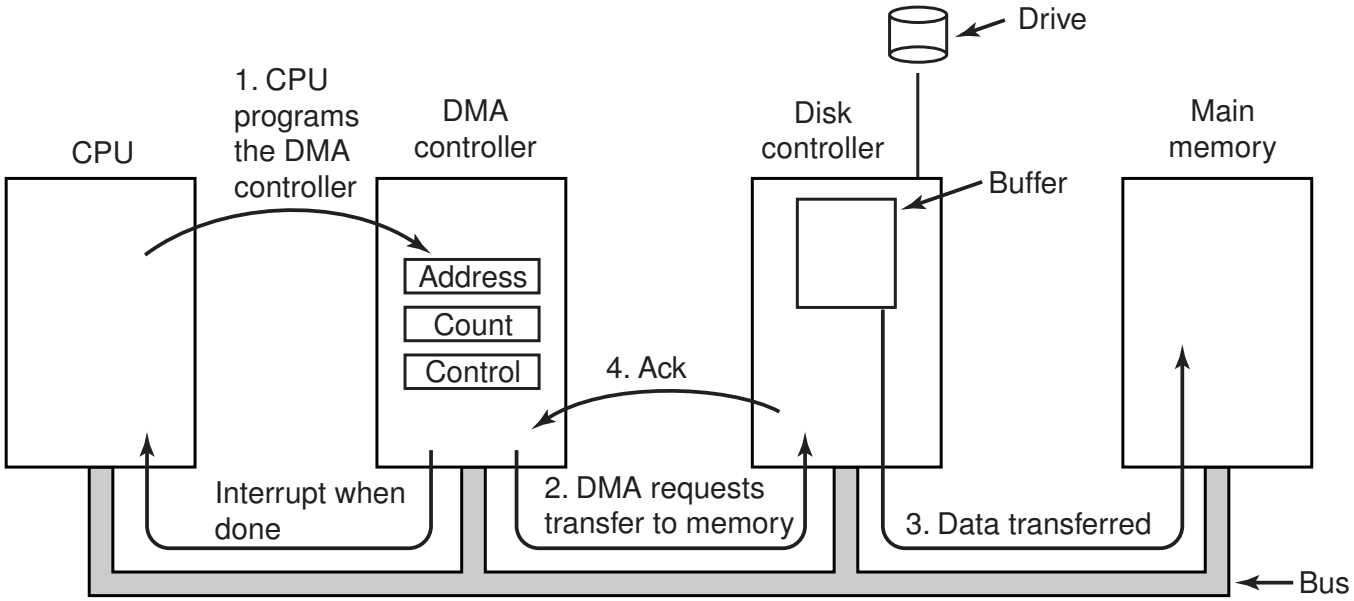
Problem Interrupt-driven I/O avoids busy waiting but the processor still has to transfer the data

- For large data blocks and high-speed devices such as disk accesses, interrupt-driven I/O would lead to a very high CPU utilization

Solution Use DMA, i.e. a dedicated processor that takes over of data transfer for the processor

- The processor has to:
  - configure the DMA controller for carrying out the data transfer
  - configure the I/O controller to carry out the operation
- The I/O controller signals the DMA controller when it is ready to transfer data
- The DMA controller requests bus access, and does the transfer when it is granted the bus
- The DMA controller interrupts the processor once the data block has been transferred

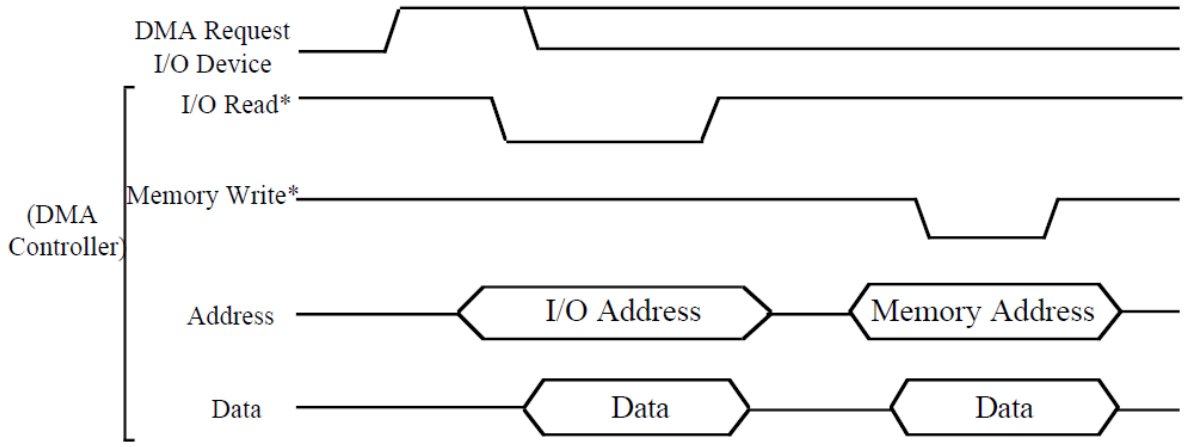
I/O with DMA (2/5)



- The CPU has to program the DMA controller with:
  - The memory address with the data buffer
  - The number of bytes/words to transfer
  - Other parameters such:
    - direction of data transfer (read/write)
    - bus mode: cycle stealing vs. burst mode
    - transfer mode: flyby vs. fetch-and-deposit

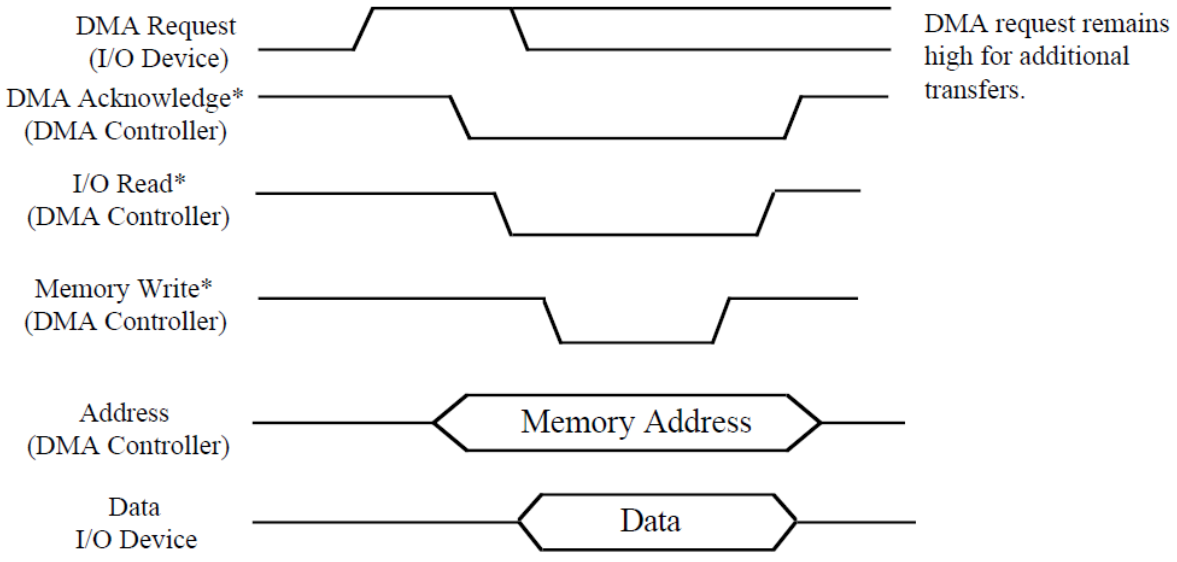
I/O with DMA (3/5)

Fetch and deposit DMA can be also used for memory-to-memory transfers



source: National Instruments

Flyby DMA



I/O with DMA (4/5)

- The code for initiating the DMA operation (to print a buffer), might be:

```
copy_from_user(buffer, p, count); /* copy data to kernel*/
set_up_DMA_io();                  /* set up for DMA operation *
scheduler();                      /* wait for I/O */
```

- The DMA controller generates an interrupt when the data transfer is done to avoid busy-waiting
- The interrupt handler for the DMA controller might be something like:

```
acknowledge_interrupt();
unblock_process();
return_from_interrupt();
```

I/O with DMA (5/5)

- A DMA controller capabilities vary widely:
  - May handle more than one data transfer simultaneously
  - May handle more than one data buffer per DMA request
- A system may have more than one DMA controller
  - Usually, each I/O bus has its own DMA controller, which is integrated into the bus controller chip

Issues

- Processor data cache coherency
- Physical memory addresses vs virtual memory addresses (later)

Agenda

Additional Reading

Sistemas Operativos

- Sections 11.1: *Objectivos do Subsistema de E/S*
- Subsection 11.5.1: *Communicação entre o Gestor e o Periférico*

Modern Operating Systems, 2nd. Ed.

- Section 5.1: *Principles of I/O Hardware*

Operating Systems Concepts, 7th. Ed.

- Section 13.1: *Overview*
- Section 13.2: *Hardware*