CG2271

Real-Time Operating Systems

Lecture 2

I/O Systems

colintan@nus.edu.sg

NUS National University of Singapore

School of Computing

NUS National University of Singapore
School of Computing

Learning Objectives

- **To understand and know what input/output (I/O) systems are, and how to program them.**
- **WHY:**
  - Accessing I/O is fundamental to real-time systems.
    - ✓ **Reading sensors to get information about the environment.**
    - ✓ **Writing to actuators to manipulate the environment.**
  - Much of microcontroller programming is about reading sensors, performing computations and writing the results to actuators.

NUS National University of Singapore School of Computing

Introduction

- **In the last lecture we looked briefly at what is in a real-time system.**
  - Sensors on the input end.
  - Controllers in the middle.
  - Actuators on the output end.
- **In this lecture we will look in more detail at these ideas:**
  - Input/Output Systems.
  - Input/Output Programming Techniques.
  - Basic Concepts.

I/O Systems

SOME PRELIMINARIES

Assembly Language

- **It's easier to explain some concepts in assembly language, and assembly language is essential for some tasks.**
  - Assembly language: This is a very low-level processor dependent language that actually manipulates the hardware on the processor.
    - ✓**Example instructions include ADD, SUB, MUL, DIV, CMP, BLT, BGT, BEQ, JMP etc. E.g. to do "for(i=0; i<20; i++)", we have:**

```
        LI R0, 0
        LI R1, 20
Loop:   ADDI R0, R0, 1
        CMP R0, R2
        BLT Loop
```

Assembly Language

- **Registers:** A microprocessor has special locations called “registers” to store data to be operated on, and to store results. A typical CPU has between 4 and 32 registers, usually labeled R0, R1, .. R31. We have just seen an example on the previous page. ☺
- **Interrupts:** Sometimes hardware needs to get the CPU’s attention. It issues an “interrupt” causing the CPU to suspend whatever it’s doing and attend to the hardware.

I/O Systems

COMMUNICATION WITH EXTERNAL SYSTEMS

Communicating with External Devices

- **For a processor to do anything useful, it must be able to communicate with the outside world.**
  - Some examples of external devices include:
    - ✓**Input devices:**

      Temperature sensors, light sensors, skid sensors, pitot tubes + static ports, etc etc
    - ✓**Output devices:**

      piezo-electric alarms, LCD/LED displays, actuators, servos, etc etc.

Communicating with External Devices Desktops/Servers vs. Microcontrollers

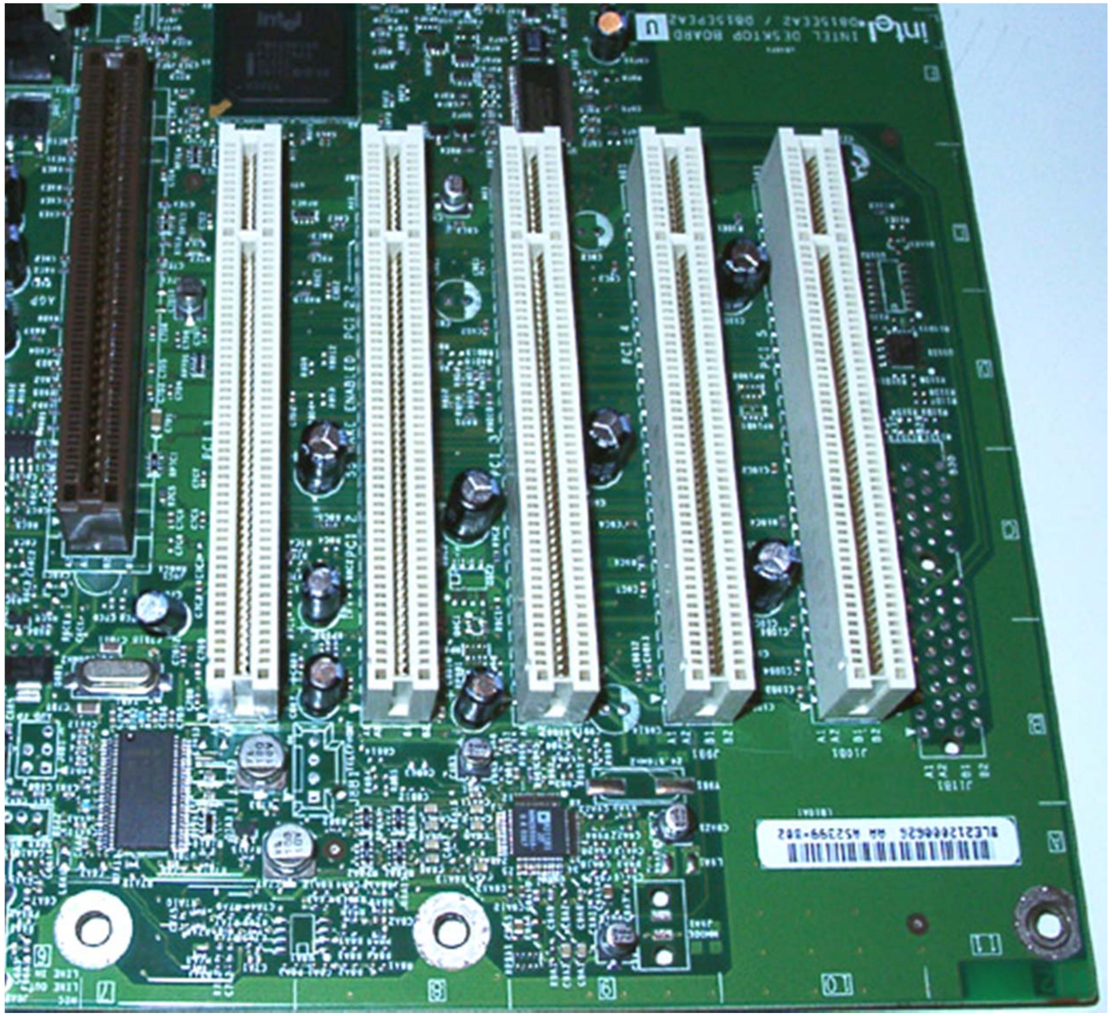
- **I/O design in desktop and server class machines are different from microcontroller class machines.**
  - Desktop and Servers (including Notebooks)
    - ✓**General purpose, meant to "interface" with a large and (at design time) unknown set of devices.**
    - ✓**Design is much more open-ended, not much built into the processor.**

      Consists just of "address", "data" and "control" lines.
    - ✓**A "bridge" circuit is instead designed around the processor to provide I/O options.**

      Generally limited to PCI, PCMCIA (notebooks), AGP/PCIe (specialized forms of PCI for video cards) and USB.
    - ✓**This open-endedness can lead to much more complicated designs in the devices to be attached.**

Communicating with External Devices Desktops/Servers vs. Microcontrollers

- **Desktops and Servers:**



Communicating with External Devices Desktops/Servers vs. Microcontrollers

- **I/O design in desktop and server class machines are different from microcontroller class machines.**
  - Microcontrollers
    - ✓**Generally a larger, but more specialized set of input/output options.**

      See list on next slide.
    - ✓**These are BUILT INTO the processor, rather than on a circuit surrounding the processor.**
    - ✓**This results in simpler interfaces with the external devices.** ☺

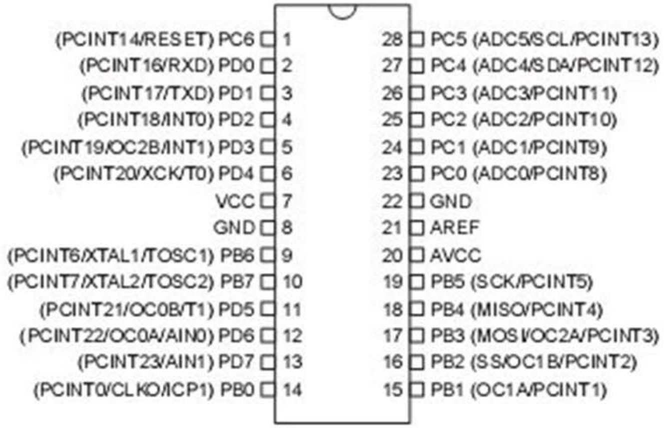
Communicating with External Devices Desktops/Servers vs. Microcontrollers

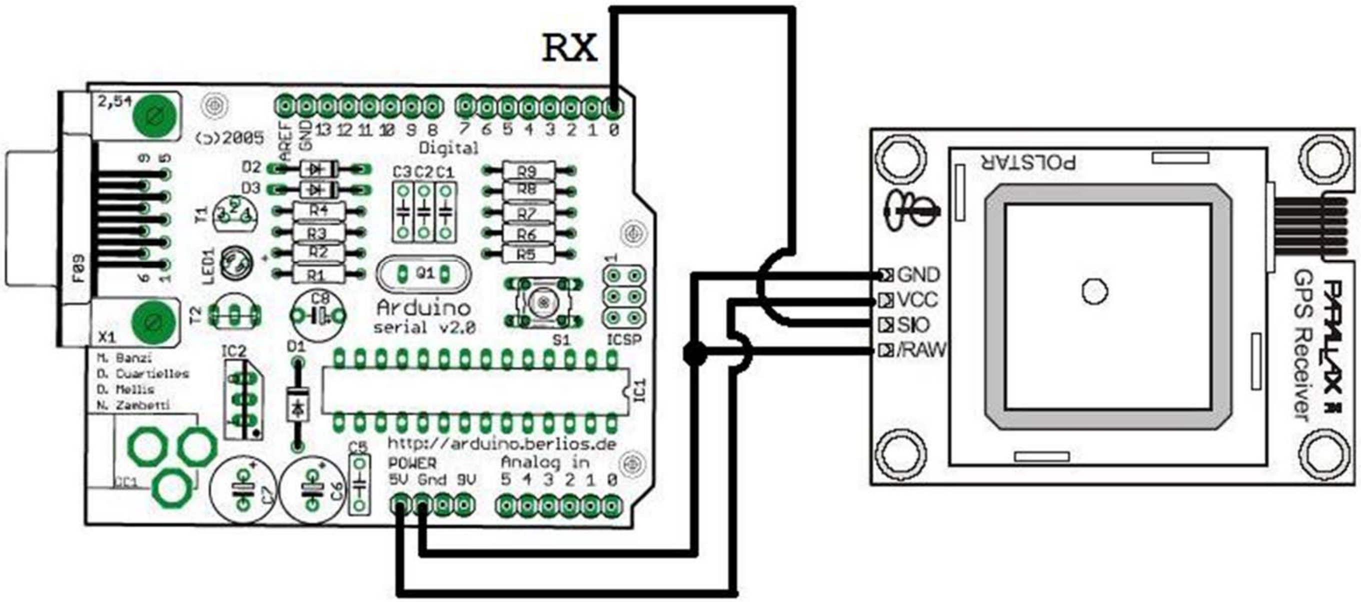
- Typical I/O “ports” in a microcontroller.
  - ✓**I2C/SPI – For communicating with other integrated circuits.**

    SPI is used for communicating with SD cards.
  - ✓**CAN – For communicating across many different devices in a vehicle.**
  - ✓**Analog-to-digital converters – For converting between voltages and digital values.**
  - ✓**PWM ports – For converting from digital to analog values.**
  - ✓**Ethernet ports – For communicating across a local area network.**
  - ✓**USB ports – For communicating with PCs.**
  - ✓**…**

Communicating with External Devices Desktops/Servers vs. Microcontrollers

- **Microcontrollers:**





Communicating with External Devices

- **To communicate with these devices, there must be:**
  - A channel for communication
    - ✓**Typically the address, data and control buses.**

      Address bus: Electrical lines that contain the addresses of memory or devices to be accessed.

      Data bus: Electrical lines that carry data to and from memory or other devices.

      Control bus: Electrical lines that tell the memory or device to either READ the data on the bus or WRITE data to the bus.
  - An agreed method for communication
    - ✓**A "protocol" that specifies which signal lines are to be de/asserted, how many clock cycles should go by before data appears etc.**
    - ✓**This is usually specified in a "timing diagram".**

I/O Systems

TYPES OF I/O PORTS

Types of I/O Ports

- **Two main types of I/O ports in DESKTOP and SERVER class systems.**
  - More common in large desktop and server systems:
    - ✓**Memory mapped I/O**
    - ✓**Direct mapped I/O**
- **Micro-controllers are designed a little differently.**
- **We will look at DESKTOP and SERVER class systems first.**

Communicating with External Devices

- **Typically every device connected to the CPU is given a unique identifier:**
  - This serves as the "address" (memory mapped) or "I/O port" (direct mapped) of the device.
  - A CPU will write to this address or I/O port to send data to the device, and read from it to get data from the device.
    - ✓**Similar in idea to reading/writing memory.**

Types of I/O Ports
Desktops and Servers

- **Two main types:**
  - Memory mapped I/O:
    - ✓**I/O identification numbers come from the normal memory addressing range of the CPU**

      E.g. memory locations 8192, 8193, 8194 and 8195 are to identify devices that read from temperature sensors 0 and 1, activate an alarm and call the fire department respectively.
    - ✓**The CPU sees the devices as part of its memory**

NUS National University of Singapore
School of Computing

Desktops and Servers
Memory Mapped I/O

- **Memory Mapped I/O example:**
  - Read from temperature sensors 0 and 1 and write to memory locations 400 and 401

```
LI R0, 8192  ; Address of sensor 0
LW R0, (R0)  ; Read from temp sensor 0
LI R1, 400          ; Address to write to
SW R0, (R1)  ; Write temperature
LI R0, 8193  ; Address of sensor 1
LW R0, (R0)  ; Read from temp sensor 1
ADDI R1, R1, 1      ; Increment R1 to 401
SW R0, (R1)  ; Write temperature to 401
```

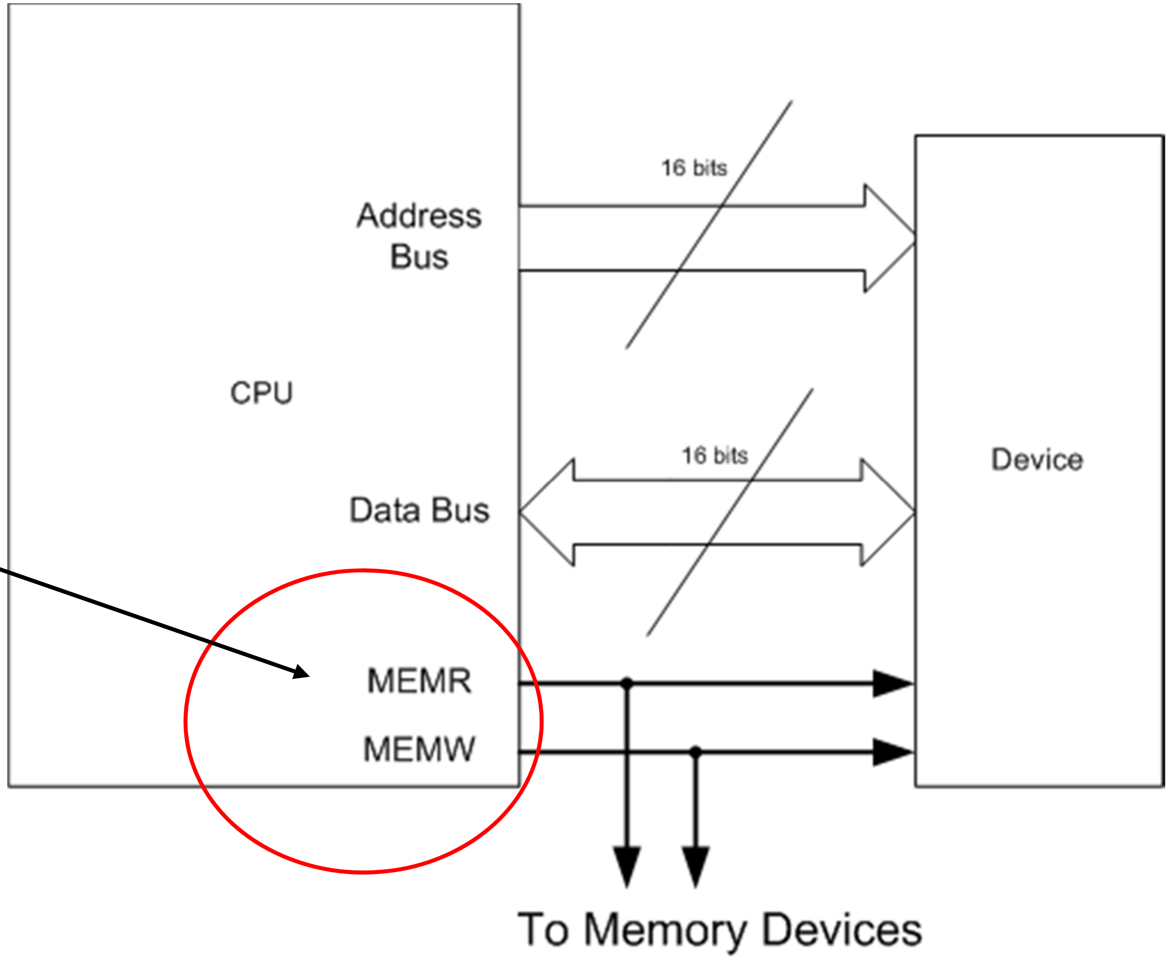
Desktops and Servers
Memory Mapped I/O

- **Important points:**
  - The port numbers for reading sensors 1 and 2 (addresses 8192 and 8193) are treated exactly the same way as memory addresses (e.g. 400 and 401).
  - Obviously memory ranges reserved for I/O ports *cannot* be used as normal memory.
    - ✓**The CPU actually loses some memory addressing range.**
  - It is however a simple and attractive scheme.

NUS National University of Singapore
School of Computing

Desktops and Servers
Memory Mapped I/O

Single MEMW and MEMR line shared with memory and I/O device



Desktops and Servers
Direct Mapped I/O

- **Second type is Direct Mapped (or Isolated) I/O:**
  - The addressing range of the I/O devices is completely separate from memory.
  - This means that the CPU does not lose any memory addressing range at all
  - AND I/O devices potentially get an equally large addressing range!

Desktops and Servers
Direct Mapped I/O

- **BUT:**
  - CPUs supporting direct mapped I/O must provide:
    - ✓**Two separate sets of READ/WRITE signals:**
      - *One set for memory*
        - MEMW and MEMR
      - *One set of I/O*
        - IOW and IOR

Desktops and Servers
Direct Mapped I/O

- ✓**New commands to read/write the I/O devices**

  Can no longer use LW/SW or similar commands.

  These would cause you to read or write memory instead of I/O devices!

- ✓**Typically these instructions take the form of:**

  IN R0, portnum ; Read the device at portnum and write to register R0

  OUT R0, potrnum ; Write R0 to the device at portnum

Desktops and Servers
Direct Mapped I/O

- **Direct Mapped I/O example:**
  - Read from temperature sensors at port numbers 8192 and 8193
  - Write to memory locations 8192 and 8193

```
IN R0, 8192       ; Read sensor 1
LI R1, 8192
SW R0, (R1)       ; Write to memory
                  location 8192
ADDI R1, R1, 1 ; Inc. to location 8193
IN R0, 8193       ; Read sensor 2
SW R0, (R1)       ; Write to memory location
                  ; 8193
```

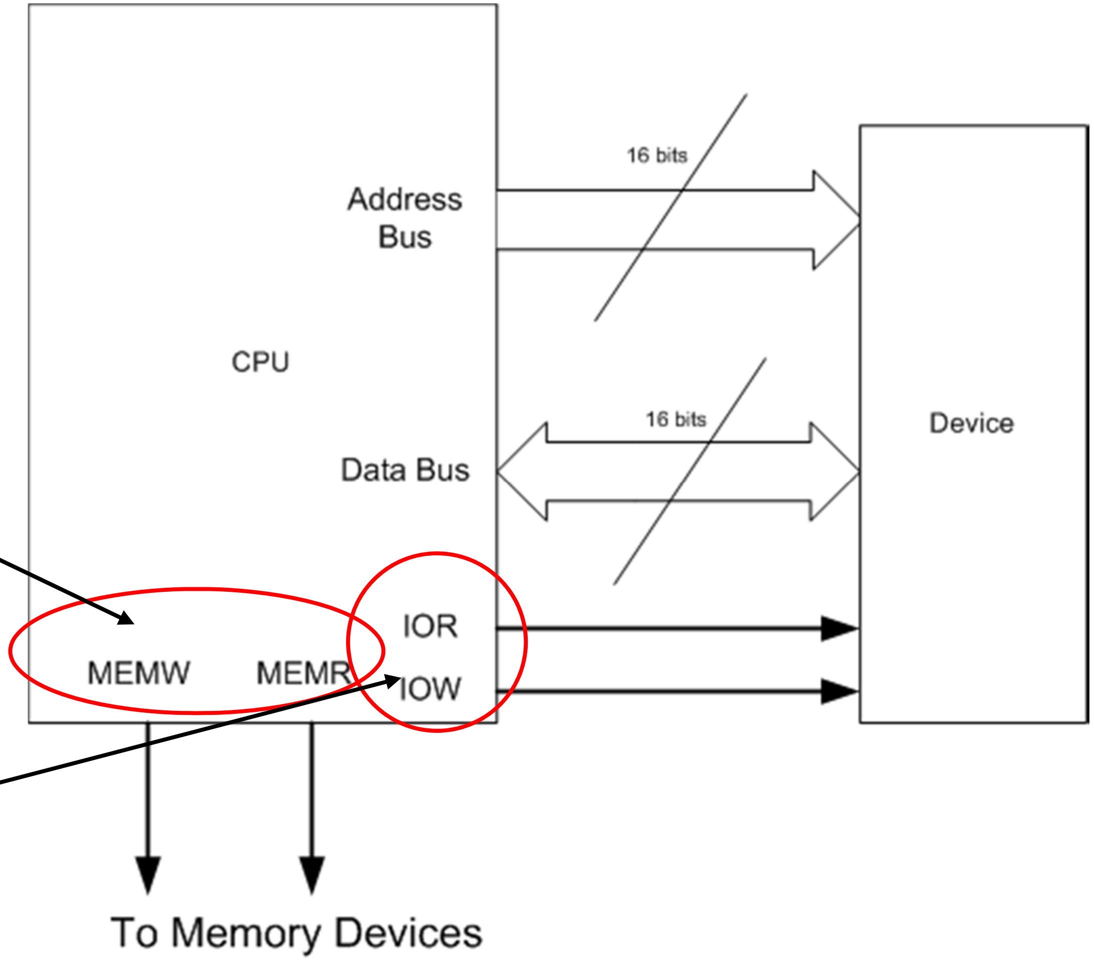
Desktops and Servers Direct Mapped I/O

- **Notice that:**
  - It does not matter that the sensors were at port numbers 8192 and 8193:
    - ✓**We can still read/write memory locations 8192 and 8193 because their addressing ranges are separate!**
  - We read/write memory using standard LW and SW instructions
  - We read/write I/O devices using special IN and OUT instructions.

Desktops and Servers Direct Mapped I/O

MEMW and MEMR lines only for memory devices

Separate IOW and IOR lines for I/O devices



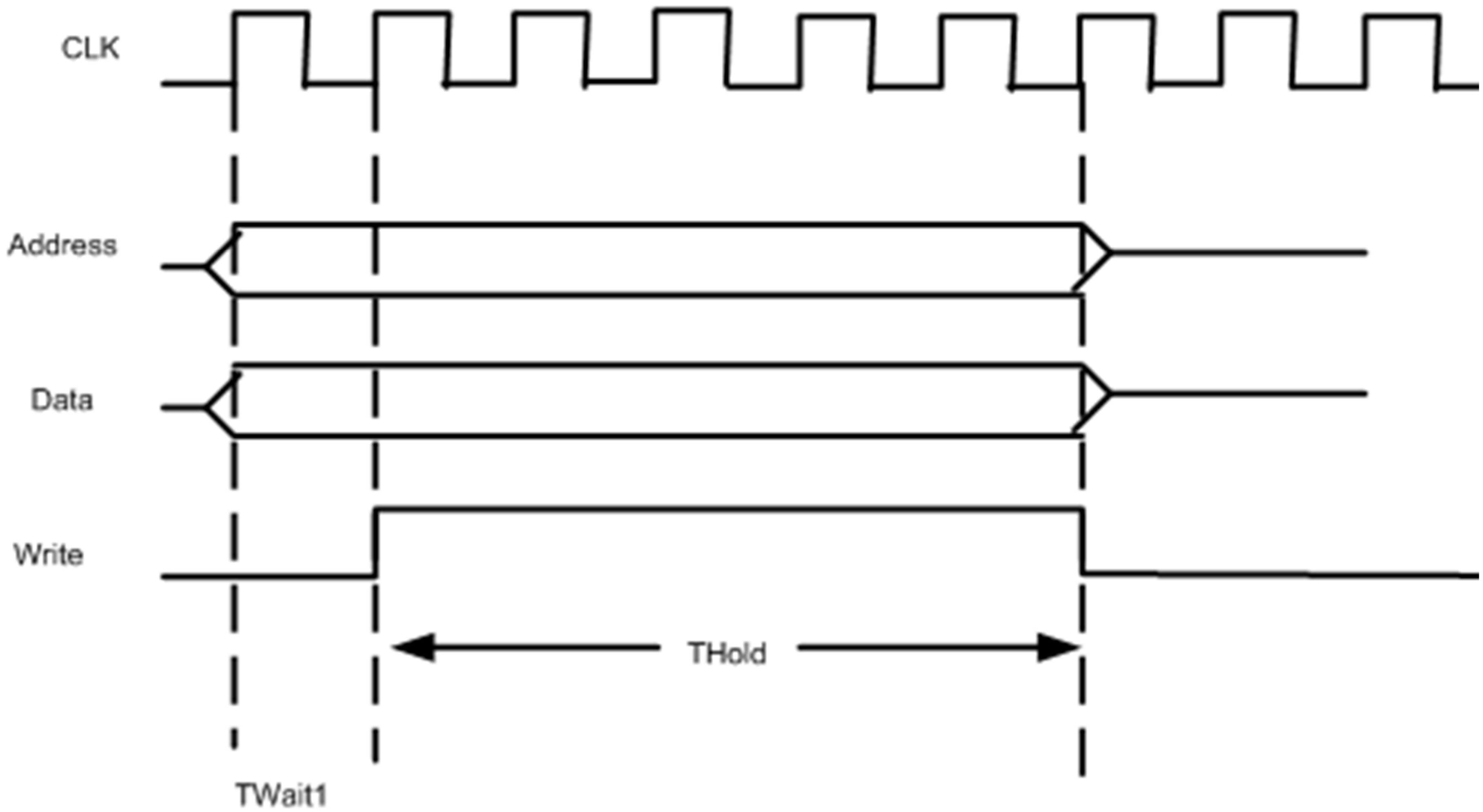
Microcontrollers
Register Mapped I/O

- **Register-mapped I/O is a variation of memory-mapped I/O.**
  - In memory mapped I/O, any location that is not used to store data can be used for I/O.
    - ✓**The corresponding device is activated when the "decoder" circuit matches the address on the address bus with the device's ID.**
  - In register-mapped I/O, the memory locations that are used for I/O is fixed.
    - ✓**These fixed locations are typically called "registers", which is different from CPU registers that you are used to.**
    - ✓**Suitable for microcontrollers as the set of peripherals is usually fixed.**

Timing Diagrams

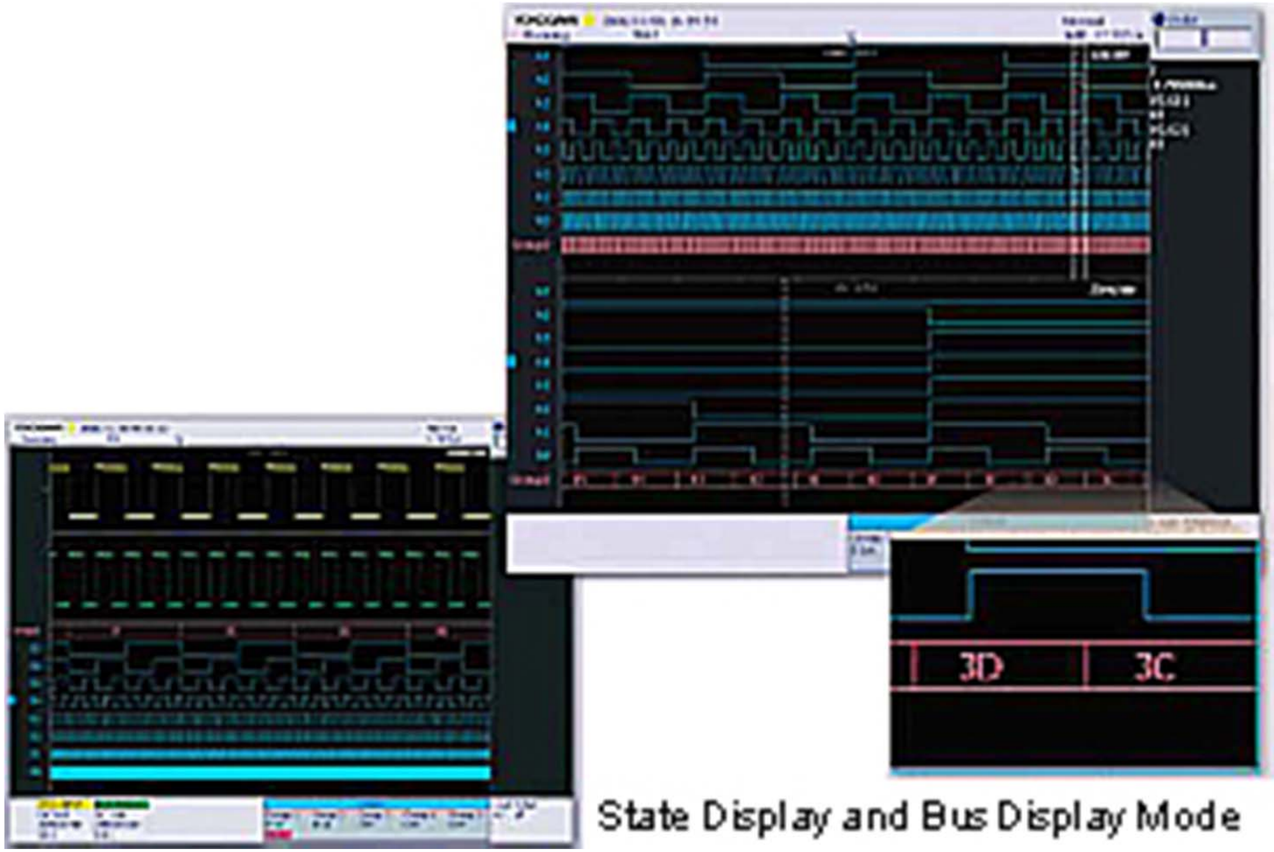
- **Typically a timing diagram specifies (in seconds) how long an event is supposed to occur after the occurrence of another event.**
  - E.g. to write to a particular RAM device:
    - ✓**Place address and data onto address and data buses respectively.**
    - ✓**After at least 25 ns, assert the WRITE signal (T_{WAIT1})**
    - ✓**Hold for at least 100 ns (T_{HOLD})**
    - ✓**Deassert WRITE signal, remove all other signals**
    - ✓**Wait for at least 50 ns before accessing device again (T_{WAIT2})**

Timing Diagrams



Timing Diagrams

- **Actual bus signals, as seen on a digital bus analyzer:**



State Display and Bus Display Mode

I/O Systems

I/O PROGRAMMING

I/O Programming

- **The 3 types of I/O ports shown earlier correspond to how the hardware is designed.**
- **On the software side, there are 3 ways to access the I/O devices.**
  - These can work with any of the 3 types of I/O ports shown earlier.
  - The 3 ways to access I/O are:
    - ✓**Programmed I/O**
    - ✓**Interrupt-driven I/O**
    - ✓**Direct Memory Access**

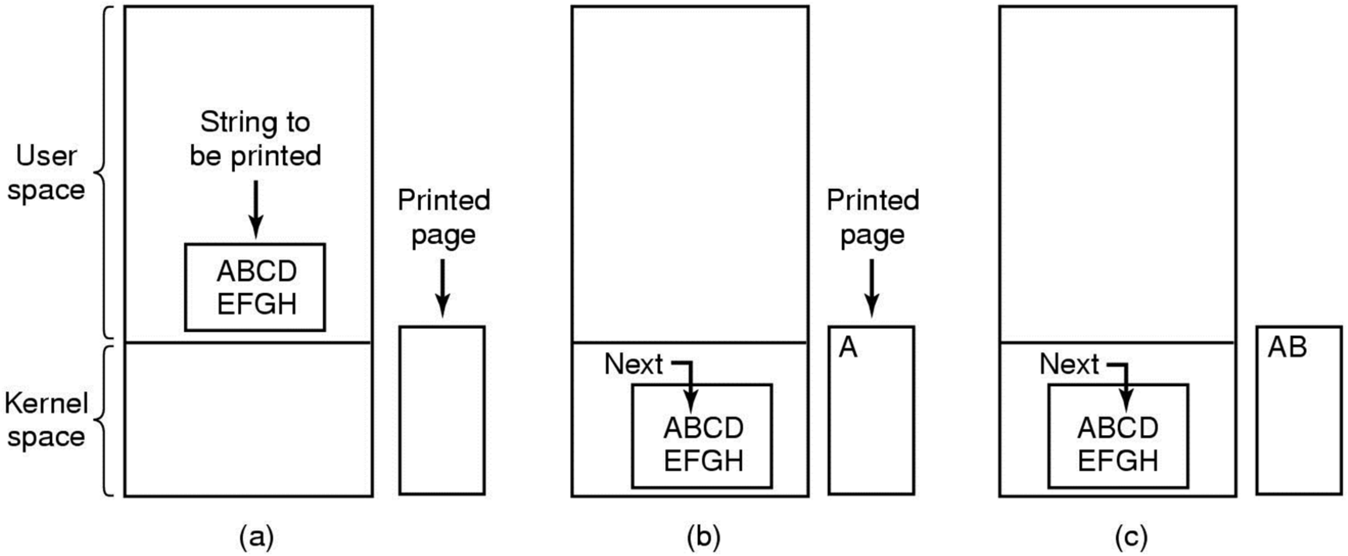
I/O Programming
Programmed I/O

- **This is the simplest form of I/O programming – The CPU does all the work.**
- **Consider a process that prints "ABCDEFGH" on the printer:**
  - User process acquires the printer by making a system call. This call returns an error or blocks if the printer is busy, until it is available.
  - The OS kernel then copies the string to be printed into its own buffer.

I/O Programming
Programmed I/O

- The OS then copies character by character onto the printer's latch, and the printer prints it out.
  - **Copy the first character and advance the buffer's pointer.**
  - **Check that the printer is ready for the next character. If not, wait.**
    - *This is called "busy-waiting" or "polling".*
  - **Copy the next character. Repeat until buffer is empty.**

I/O Programming
Programmed I/O




```
copy_from_user(buffer, p, count);                  /* p is the kernel bufer */
for (i = 0; i < count; i++) {                      /* loop on every character */
      while (*printer_status_reg != READY) ;       /* loop until ready */
      *printer_data_register = p[i];               /* output one character */
}
return_to_user( );
```

I/O Programming
Programmed I/O

- **Issue:**
  - It takes perhaps 10ms to print a character.
  - During this time, the CPU will be busy-waiting until the printer is done printing.
  - On a 96MHz processor like the ARM STR912, this is equivalent to wasting 960,000 instruction cycles!
    - ✓ **Assumption is that 1 instruction completes in 1 cycle. Realistic for a RISC processor.**

I/O Programming
Interrupt I/O

- **Since 960,000 instructions is a large number, why not let the CPU do other stuff while the printer is busy?**
  - After the string is copied to kernel space, the OS will send a character to the printer, then switch to a task.
  - When the printer is done, it will interrupt the CPU by asserting one of the "interrupt request" (IRQ) lines on the CPU.
  - This triggers a software routine called an "interrupt handler" , "interrupt service routine" or "interrupt service procedure" which will then load the next character. We will use the term "ISR".

NUS National University of Singapore
School of Computing

I/O Programming
Interrupt I/O

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
    i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

I/O Programming
Interrupt I/O

- **Issue:**
  - The printer interrupts the CPU every time it prints a character!
  - Each interrupt has overheads:
    - ✓**CPU must stop what it is doing, look up a vector table for the address of the appropriate ISR, then hand control to the ISR.**
    - ✓**ISR must save the registers, process the interrupt, and restore the registers.**
    - ✓**CPU then hands control back to the interrupted process.**

I/O Programming
Interrupt I/O

- **There are typically several hundred thousand to several million characters in a document!**
  - This means triggering millions of interrupts to print each document.
  - The overheads can be sizeable.
- **Would be ideal if we could just dump the document on the printer and worry only when it has finished printing everything!**
  - This is where Direct Memory Access comes in.

I/O Programming
Direct Memory Access.

- **In Direct Memory Access (DMA):**
  - A separate processor called a DMA Controller (DMAC) is used to transfer data between a device and memory.
- **In a DMA READ:**
  - The user process initiates the READ by making an OS system call, providing:
    - ✓**Information on which device to read. In the case of a disk drive, the user process will provide information of which part of the disk (side number, track number, starting block number) to read.**
    - ✓**Information on how many bytes (or blocks of bytes) to read.**

I/O Programming
Direct Memory Access.

- ✓**Starting address of the buffer to write to.**
- ✓**The process is blocked until the transfer is done.**

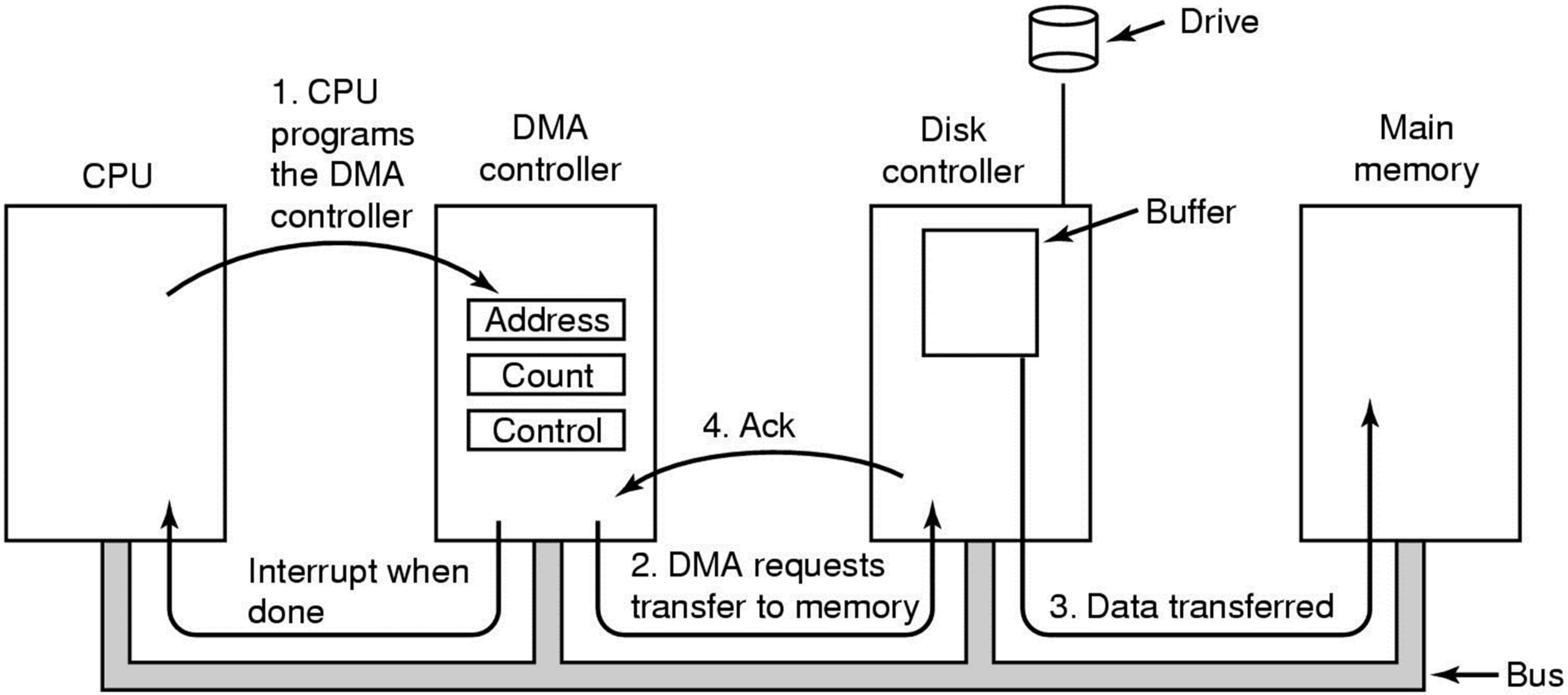
▪The OS then (using appropriate I/O ports) passes this information to the DMAC, and tells the DMAC to start transferring.

▪The OS then switches to another process, while the DMAC transfers the data independently of the CPU.

▪When the DMAC is done, it interrupts the CPU.

- ✓**The ISR for the DMAC then calls routines in the OS to move the blocked process to the READY queue.**

I/O Programming
Direct Memory Access.



NUS National University of Singapore School of Computing

I/O Programming
Direct Memory Access.

- **In our printer example:**
  - The user process makes an OS call with a pointer to the text buffer, and the number of characters to print.
  - The OS copies the text into its own buffers, then sets up the DMA transfer and blocks the calling process.
  - The OS initiates a DMA transfer, then hands control to another process.
  - When the DMAC is done, it interrupts the OS. The OS moves the calling process into the READY state.

I/O Programming
Direct Memory Access.

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

Summary

- **In this lecture we looked as some basic ideas:**
  - What is assembly language?
  - I/O Ports
  - Designing programs to access I/O
    - ✓**Programmed I/O**
    - ✓**Interrupt I/O**
    - ✓**Direct Memory Access I/O**