

Inf2C - Computer Systems

Lecture 14

I/O

Boris Grot

School of Informatics
University of Edinburgh



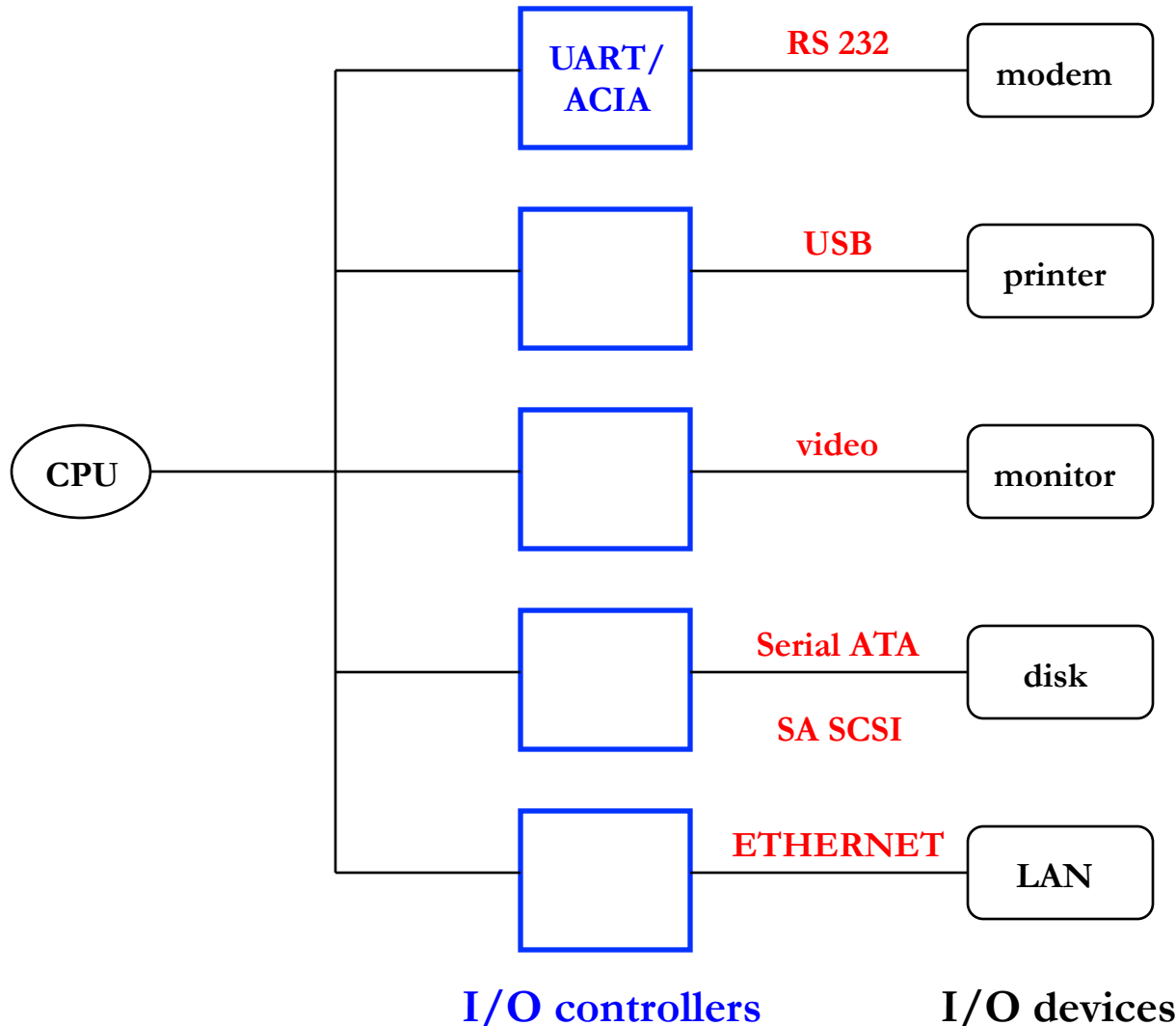
Previous lecture: Exceptions & processor mgmt

- Exceptions: interrupt normal program flow and require servicing by the CPU
 - Internal “traps” (e.g., syscall)
 - external “interrupts” (e.g., keyboard click)
- Exception handling
- Processor protection modes (user vs kernel)
- Processor management via time-sharing

Examples of I/O Devices

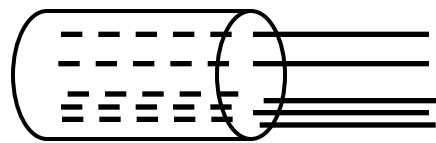
Device	Behaviour	Partner	Data Rate (Mbit/sec)
Keyboard	Input	Human	0.001
Mouse	Input	Human	0.004
Voice input	Input	Human	0.26
Laser printer	Output	Human	3.2
Graphics	Output	Human	800-8000
Magnetic disk	Storage	Machine	800-3000
Network/LAN	Input or output	Machine	100-40,000 (40Gbit/sec)

Lecture 14: I/O Controllers & Devices



Example: RS232 Serial Interface

- I/O controller: UART (Universal Asynchronous Receiver Transmitter)
 - ACIA (Asynchronous Communications Interface Adapter) also used
- Used for modems and other serial devices
- Physical Implementation:
 - 2 signal wires (one for each direction) + ground reference + status signals



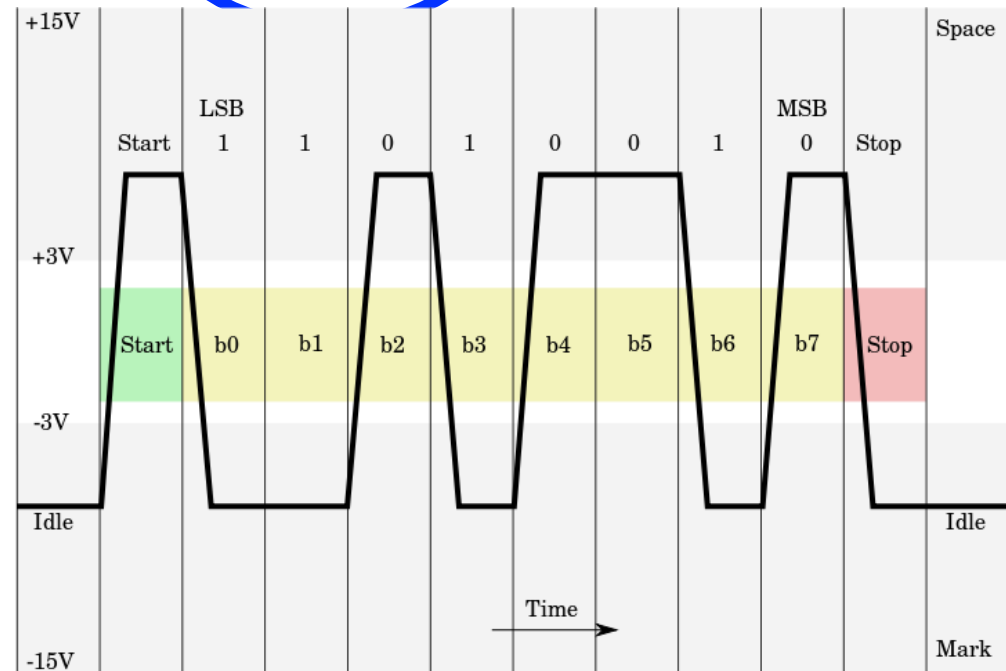
Tx data wire (CPU to I/O device)
Rx data wire (I/O device to CPU)
GROUND wire, status signals

RS232 modem: bits and wires



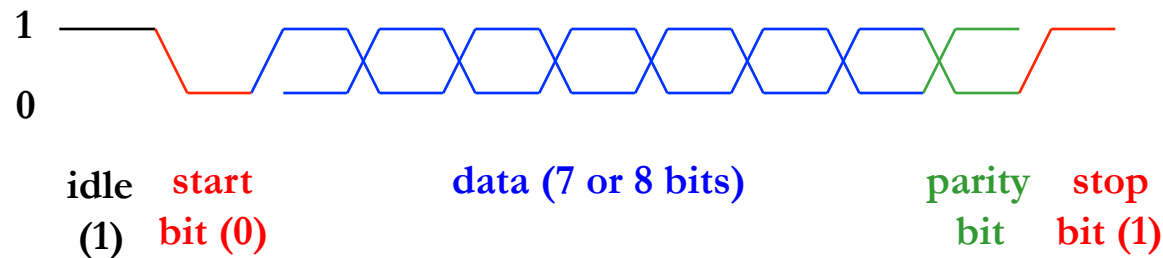
Oscilloscope trace of voltage levels for an ASCII "K" character (0x4B) with 1 start bit, 8 data bits, 1 stop bit.

Source: wikipedia



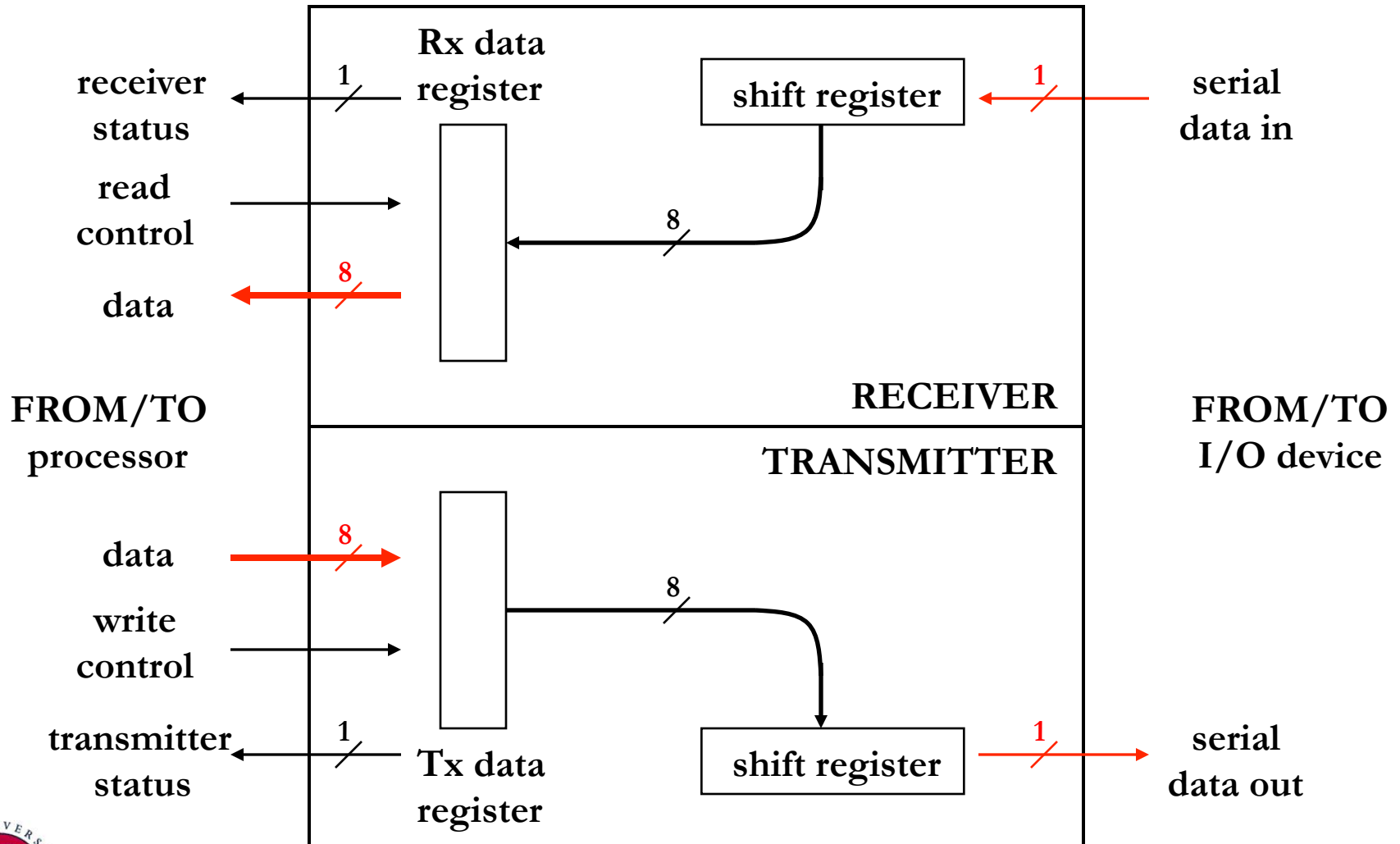
Example: RS232 Serial Interface

- Encoding:
 - 1 character = 10 or 11 bits (including signaling)
 - Idle state is represented by a constant “1”



- Parity: for detection of transmission errors
 - odd → total number of “1”s (including parity bit) is odd
 - even → total number of “1”s is even

UART Controller



Connecting CPU and I/O Controllers

- Option 1: connect the I/O Tx and Rx registers directly into some special CPU I/O registers → not flexible
- Option 2: keep I/O registers in separate I/O controller and connect CPU to I/O controller through special I/O bus → expensive, not flexible

I/O bus:

- data lines (8 bits)
- control lines (READ and WRITE signals),
- address lines (some few bits) → each I/O controller is assigned a range of addresses for its registers

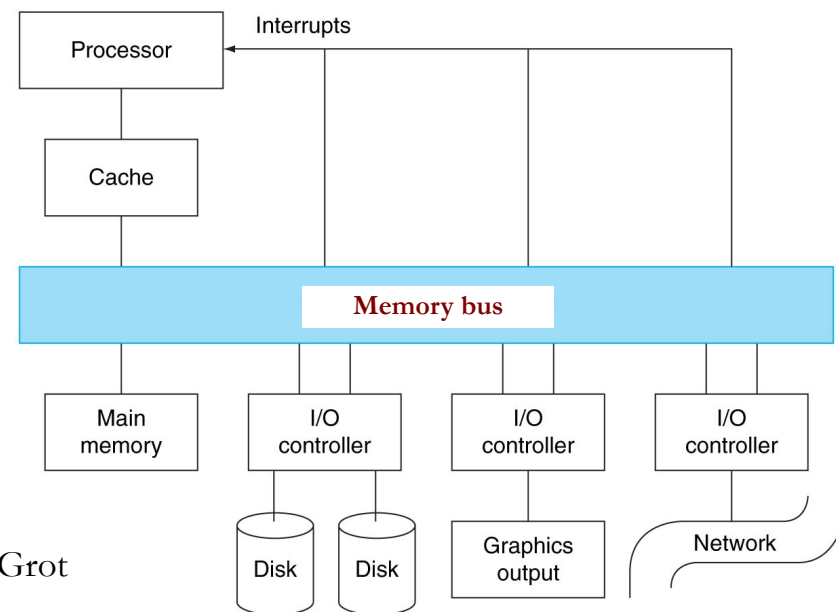
Data is accessed through special I/O loads and stores

Connecting CPU and I/O Controllers

- Option 3: keep I/O registers in I/O controller and connect the CPU to I/O controller through memory bus

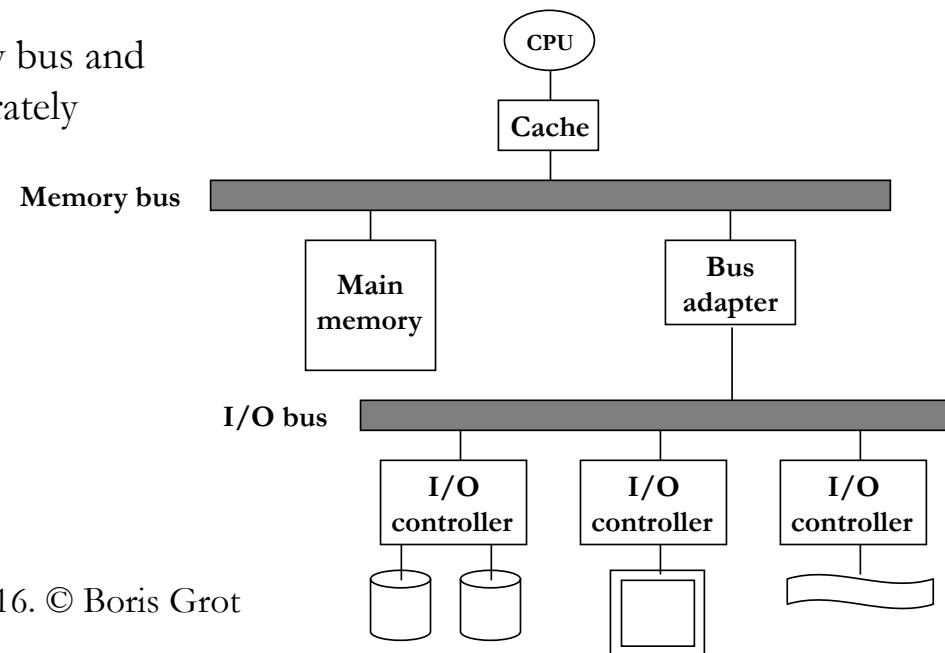
Memory mapped I/O:

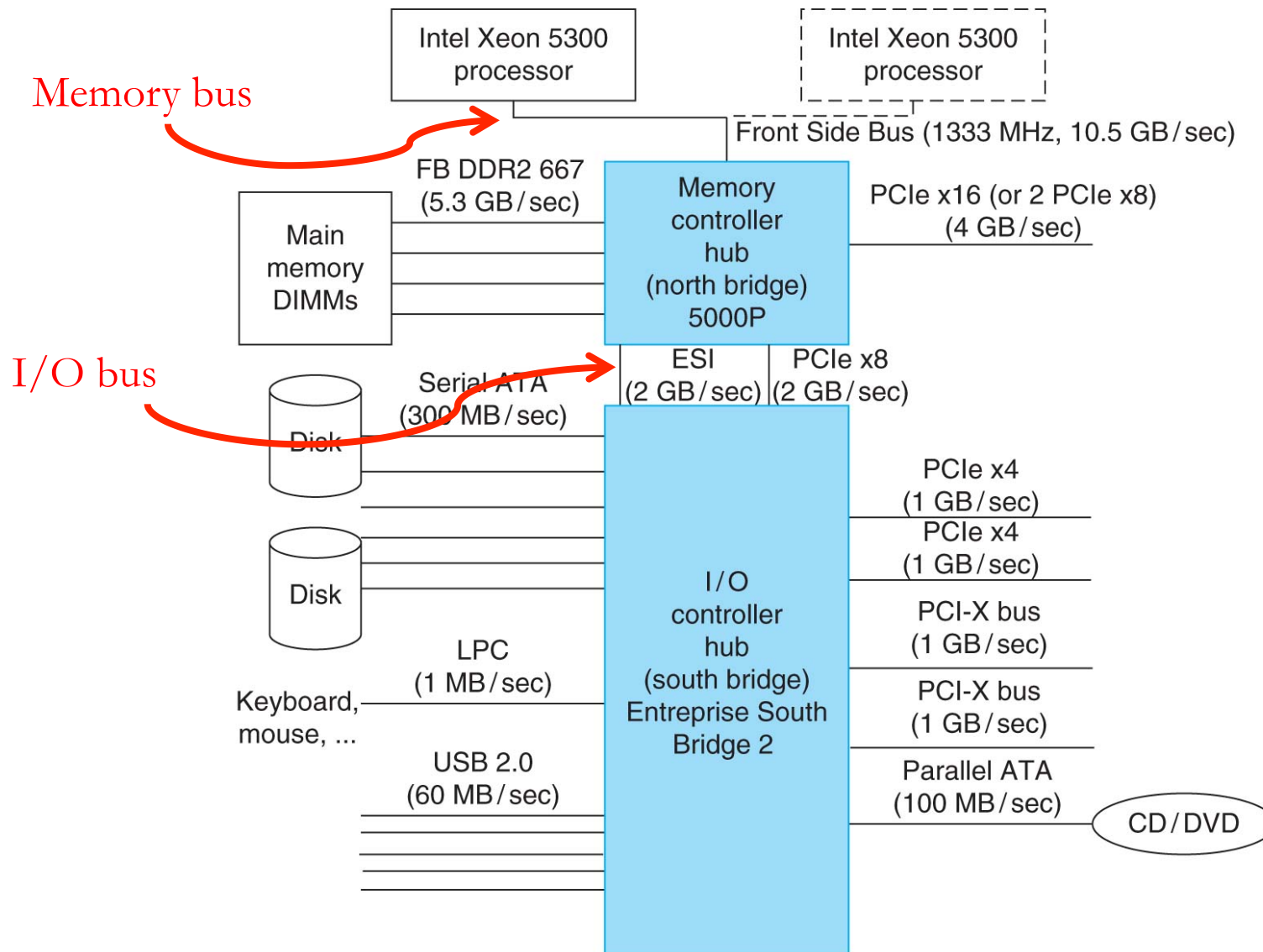
- I/O controller registers (data and control) are mapped to a dedicated portion of memory
- Good news: **accessed with regular load and store instructions**
- Bad news: Takes bus bandwidth away from CPU-memory



Connecting CPU and I/O Controllers

- Option 4: connect I/O controllers to I/O bus and the I/O bus to the memory bus through a bus adapter
 - Off-load the I/O from the memory bus: multiple I/O devices appear as a single device to the memory bus)
 - Pros:
 - better performance: slow I/O bus doesn't clog up the fast memory bus
 - higher flexibility: add/remove devices without impacting the high-performance processor-memory interface
 - Modularity and extensibility: memory bus and I/O bus technology can evolve separately





Organization of the Memory & I/O system on an Intel server using the Intel 5000P chip set.

Polling and interrupt-based I/O

How do we check I/O status (e.g., key clicked)?

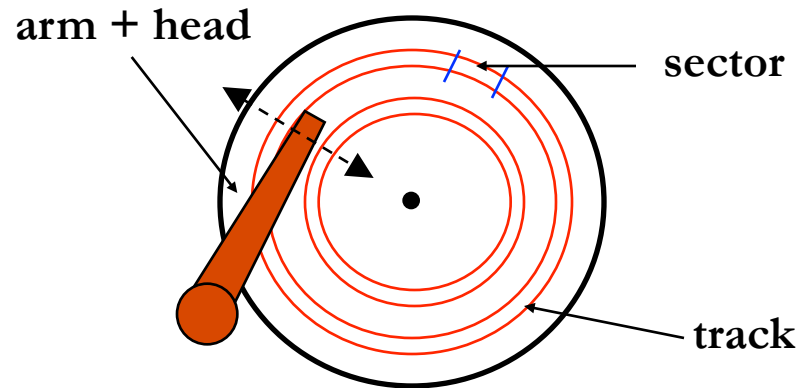
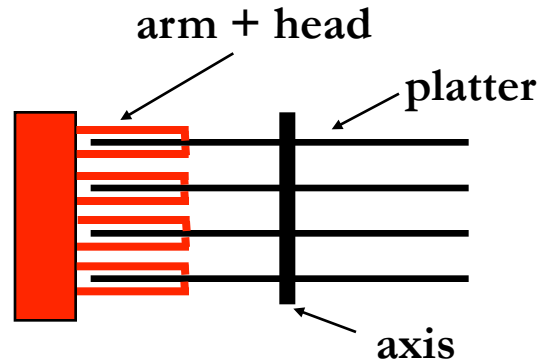
- Option 1: Polling
 - User process calls OS at regular intervals to check status of I/O operation
 - Wasteful for events that happen very infrequently (e.g., keyboard clicks)
- Option 2: Interrupt
 - I/O controller interrupts user process to signal an I/O event
 - Heavy-weight (break out of user code into kernel mode)

USB example to bring things together

- USB devices don't generate interrupts
 - Keeps cost low
- Computer can't afford to keep polling the multitude of USB ports
 - Would lose too much performance for nothing
 - Not polling often enough not an option: will lose data
- USB controller on the CPU side does the polling (simple FSM) and generates an interrupt to inform the CPU
 - This functionality is in the Southbridge in the picture on slide 13



Hard Disks



- 1-4 platters per drive
2 surfaces per platter
10-50k tracks per surface
100-500 sectors per track
512B – 4KB per sector
- Spinning speed:
 - 5400-15000 rpm
(90-250 revs per sec)



Disk Performance

- Total time of a disk operation is divided in two parts:
 1. **Access time:** time to get head into position to read/write data
 $\text{access time} = \text{seek time} + \text{rotational latency}$
 - **Seek time:** time to move head to appropriate track ($< 10\text{ms}$)
 - **Rotational latency:** time to wait for appropriate sector to arrive underneath the head ($< 10\text{ms}$)
Dependent on spinning speed
 2. **Transfer time:** time to move data to/from disk
 $\text{transfer time} = \text{time to transfer 1 byte} * \text{number of bytes of data}$
 - Dependent on both spinning speed and recording density
 - 75-125 MB/s (changing very slowly – limited by mechanics)



Disk Controllers

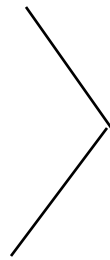
- Disk controller inside disk unit → responsible for all mechanical operation of disk + interface with CPU
- I/O registers:
 - Exchange data and control between CPU and disk controller
 - Command register → tells disk controller what to do next

e.g. Seek n

Read Sector m

Write Sector m

Format Track



High-level commands

Using a Disk Controller

- Step 1: user program requests data from a file
- Step 2: OS file system determines sector(s) to be accessed
- Step 3: OS disk handler issues **Seek** command and CPU goes to work on some other process (multi-tasking)
- Step 4: I/O controller interrupts CPU to signal completion of seek
- Step 5: OS disk handler issues **Read Sector** command and CPU goes to work on some other process
- Step 6: I/O controller interrupts CPU to signal data ready
- Step 7: OS disk handler transfers data to/from disk
- Step 8: go to step 3 or 5 and repeat until all data transferred

Interrupt Approach

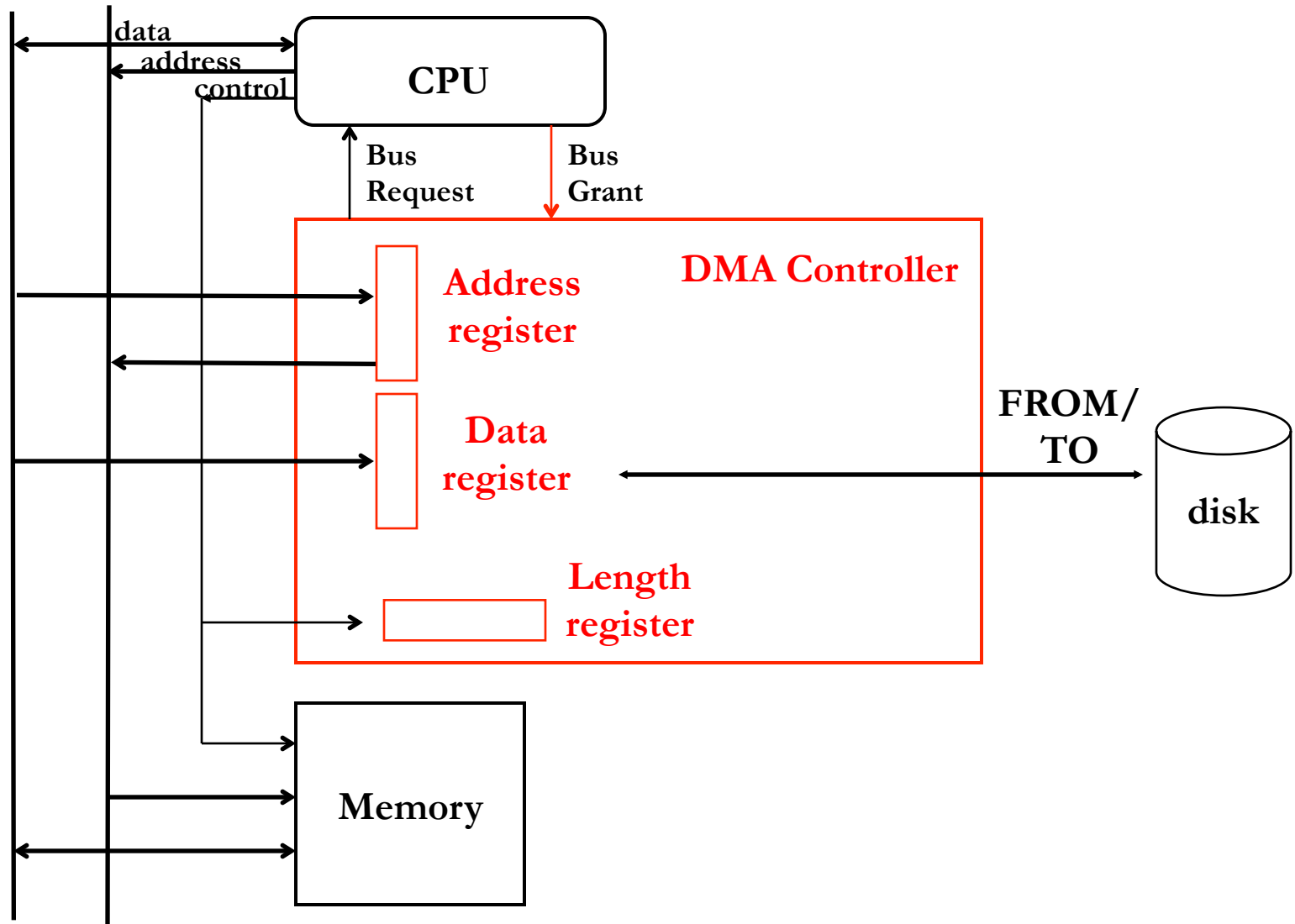
- CPU (through OS) has to issue individual commands to read every sector from disk
- Data transfer is very slow ($\sim 100\mu\text{s}$ for a 512 byte sector)
- Frequent interrupts detrimental to processor performance
 - Switch to OS, then switch back to the application (Lecture 13)
- Solution: Direct Memory Access (DMA)

Direct Memory Access (DMA)

- DMA controller: stateful device that sits on the memory bus and can independently transfer data between memory and disk
 - Setup by the processor for each transfer using memory-mapped registers inside the DMA controller
- DMA registers:
 - Address register → position in memory of next data to be read/written
 - Data register → temporary storage for data to be transferred
 - Length register → number of bytes remaining to be transferred
- Technical write-up of how this works for x86:
<http://www.brokenthorn.com/Resources/OSDev21.html>



DMA Organization



DMA Operation

- Step 1, 2: user program requests data, OS determines location of data on disk
- Step 3: OS disk handler issues **Seek** command and sets up DMA registers (address, length); CPU goes to work on another process
- Step 4, 5: I/O controller interrupts CPU, OS disk handler issues **Read Sector** command
- Step 6: I/O controller informs DMA controller that data is ready (no need to interrupt CPU)
- Step 7: DMA controller transfers data into memory; length register is decremented until all data is moved (advanced DMA controllers can access multiple tracks with a single operation)
- Step 8: DMA controller interrupts CPU to inform completion of DMA operation



Bus Arbitration

- DMA and CPU connect to memory bus → access must be somehow arbitrated to avoid conflicts
- Solution: additional logic (**bus arbiter**)
 - Authorizes CPU or DMA controller to access memory at any given time
- Two new wires on the memory bus:
 - Bus Request → asserted by the DMA controller when it requires the bus
 - Bus Grant → asserted by the CPU when it is not using the bus and thus the DMA controller can use it
 - In case of conflicting requests in the same cycle, CPU usually has priority