Hardware (P)Review

Gerd Liefländer

WT 2008/09

System Architecture Group

Literature

- Bacon, J.: Operating Systems (3)
- Davis, W.: Operating Systems (2)
- Hennessy, J.: Computer Architecture, 2003
- Nehmer, J.: Grundlagen moderner Betriebssysteme (2)
- **Messmer, H.P.: PC Hardwarebuch: Aufbau, Funktionsweise, Programmierung, Addison Wesley, 7. Auflage, 2004**
- Stallings, W.: Computer Organization and Architecture, 2003
- Patterson, D.: Computer Organization and Design: The Hardware/Software Interface
- *Schröder-Preikschat, W.: SOS I, Ch. III Organisation von Rechensystemen, Uni Erlangen*
- Silberschatz, A.: Operating System Concepts (2)
- Tanenbaum, A.: Modern Operating Systems (1)

Recommended Links

http://www.desy.de/user/projects/C++/courses/cc/Tutorial/

http://wwwh.eng.cam.ac.uk/help/tpl/languages/java/cuedjavanotes/CUEDC++.html

http://www.cee.hw.ac.uk/~rjp/Coursewww/

http://newtech.maconstate.edu/academics/ITEC4266Assignments.asp

http://salsa.cit.cornell.edu/cs213-sp01/lectures.html

Slides from Theo Ungerer (Uni Augsburg):
Systemnahe Informatik und Kommunikationssysteme

Agenda

- Introduction, Motivation
- HW Overview and Classification
- Computer Organization
  - Hardware/Software Hierarchy
  - Semantic Gap
- Computer Components
- Processor (CPU)
- Memory Hierarchy
- Caching and Locality
- Exception/Interrupt
- Physical-I/O

Note: See related HW courses on the web

Motivation

Why Hardware Review?

- Parts of OS control the hardware (HW), e.g.
  - Parts of OS kernel are HW dependent
- HW helps to control the OS and its applications
- Modern HW supports parallelism

⇒ OS must deal with

- Real-time applications
  - Need for additional HW support
  - Control of special HW

HW Abstraction versus HW Ignorance

- Designing and implementing an OS without knowing your HW

  ⇒ *inefficient system*

  ⇒ *insecure system*

OS = <A, B, C, D, E>

- Designing an OS without abstraction

  ⇒ *inflexible system*

  ⇒ *non portable system*

OS-hacking took place too often

Overview and Classification

Classification

Architectures

Trends

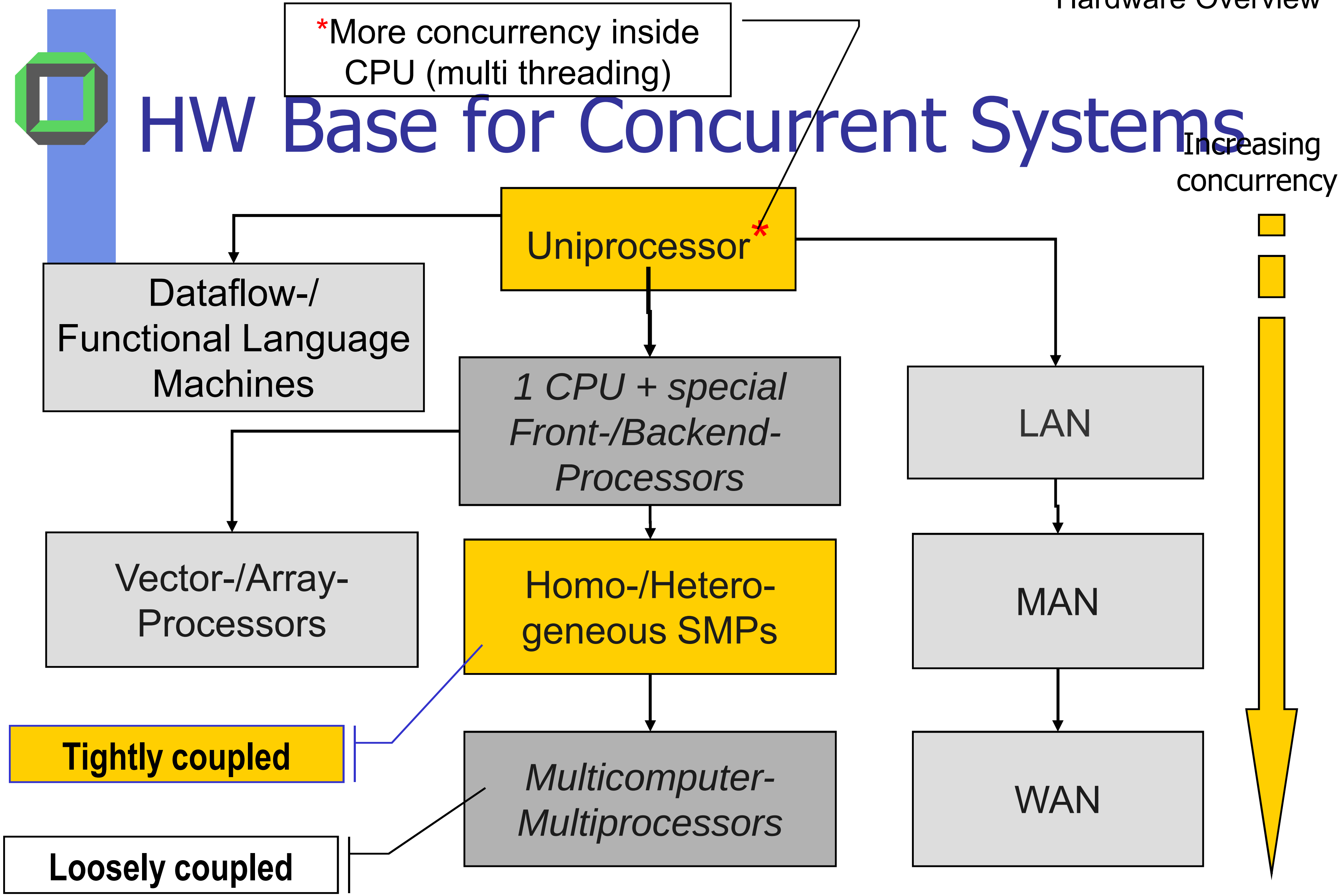
Flynn’s System Classification

	SINGLE INSTRUCTION STREAM	MULTIPLE INSTRUCTION STREAM
Single Data Stream	**SISD** (Single Processor, e.g.most PCs)	MISD[1]
Multiple Data Stream	SIMD[2] (Super-/Array-Computer, e.g. Cray, ICL DAP)	**MIMD** (SMPs (e.g. WSs), DS, Networks, …)

[1] Only few real HW-architectures of this type

[2] Not in this course

HW Base for Concurrent Systems



*More concurrency inside CPU (multi threading)

Uniprocessor*

Dataflow-/ Functional Language Machines

1 CPU + special Front-/Backend-Processors

LAN

Vector-/Array-Processors

Homo-/Hetero-geneous SMPs

MAN

Tightly coupled

Multicomputer-Multiprocessors

WAN

Loosely coupled

Increasing concurrency

Classification of Parallel Computers

- Homogeneity of
  - HW/ OS/ application
- Synchrony
  - Bulk synchronized, loosely synchronized
- Interaction mechanisms
  - Shared variables, message passing
- Address space
  - Shard memory, distributed memory, uniform-, non uniform access
- Memory model
  - EREW, CREW, CRCW / consistency

EREW = exclusive read and exclusive write
CREW = concurrent read and exclusive write

Classification of Parallel Computers

Pragmatic:

- Shared memory multiprocessors
  - Hyper Threaded ~ 2 – 8 threads
  - SMP ~ 2 – 64 CPUs
  - NUMA, CC-NUMA
- MPP (Massively parallel)
  - NUMA, CC-NUMA
  - Message passing MP, NORMA
- Cluster ~ 8 K CPUs

no

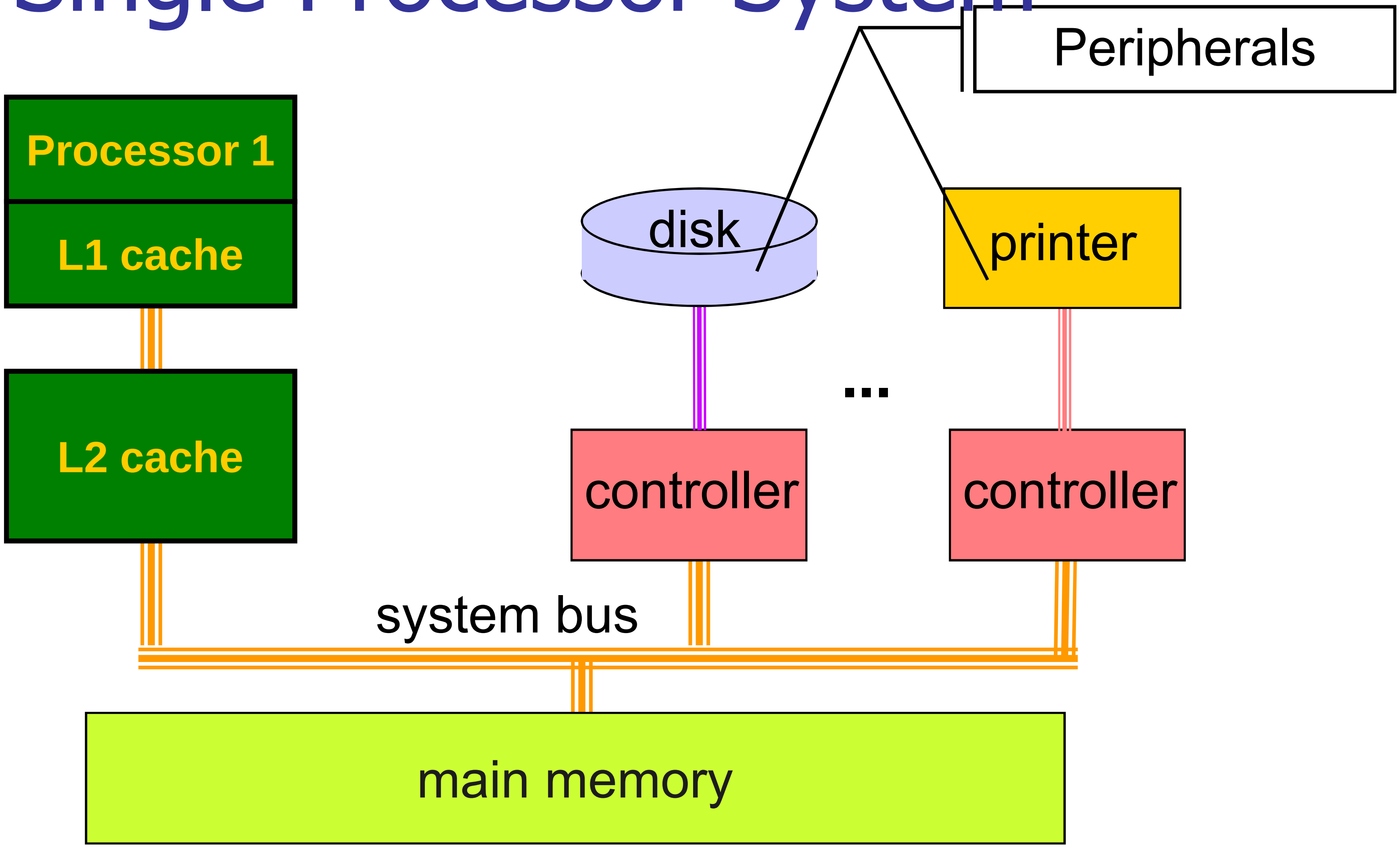
NORMA = No Remote Memory Access, i.e.
interaction with other processors via messages

Computer Architecture

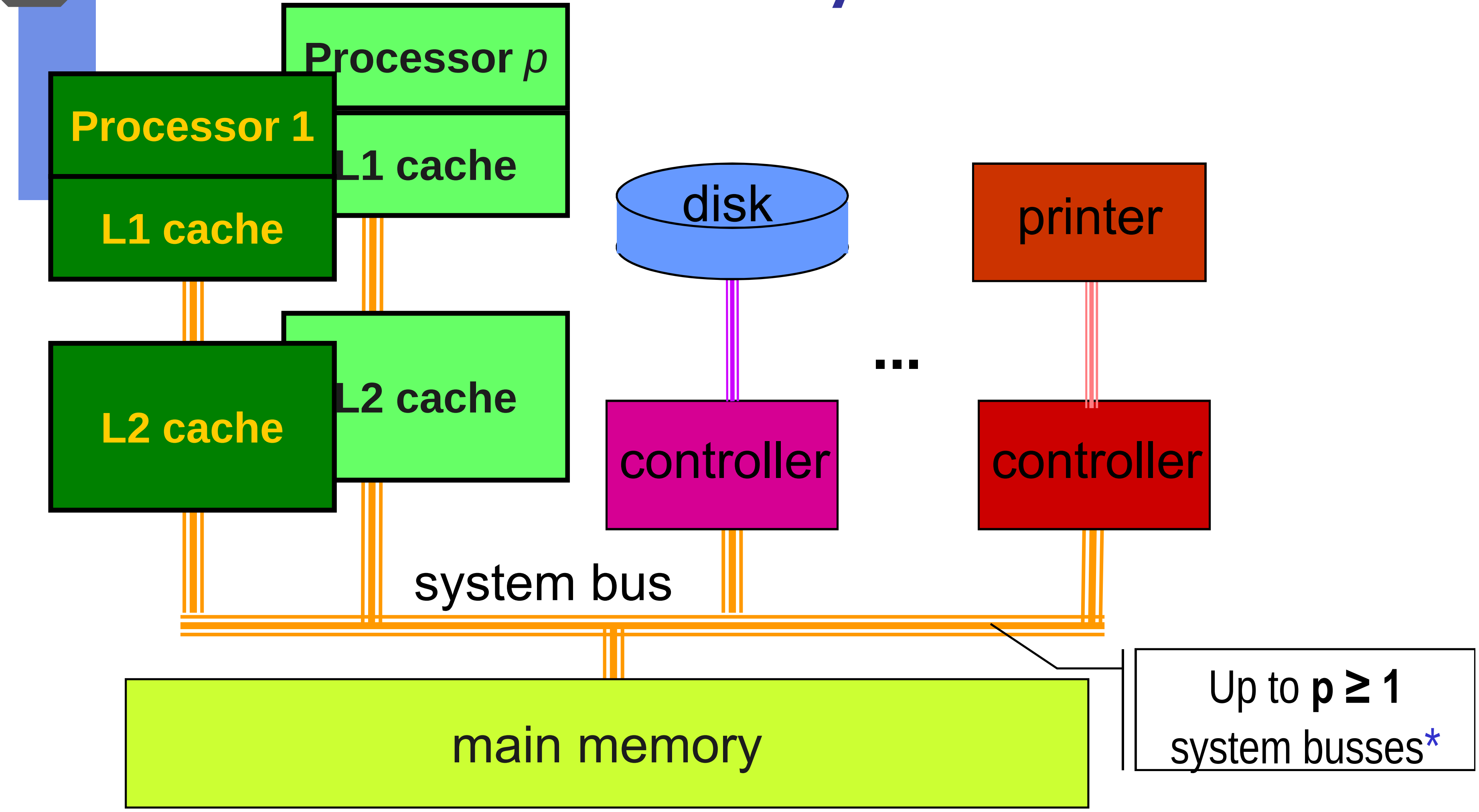
Computer Architectures

- Single Processor System
- Multi Processor System (SMP)
- Distributed Systems (LANs …)

Single Processor System



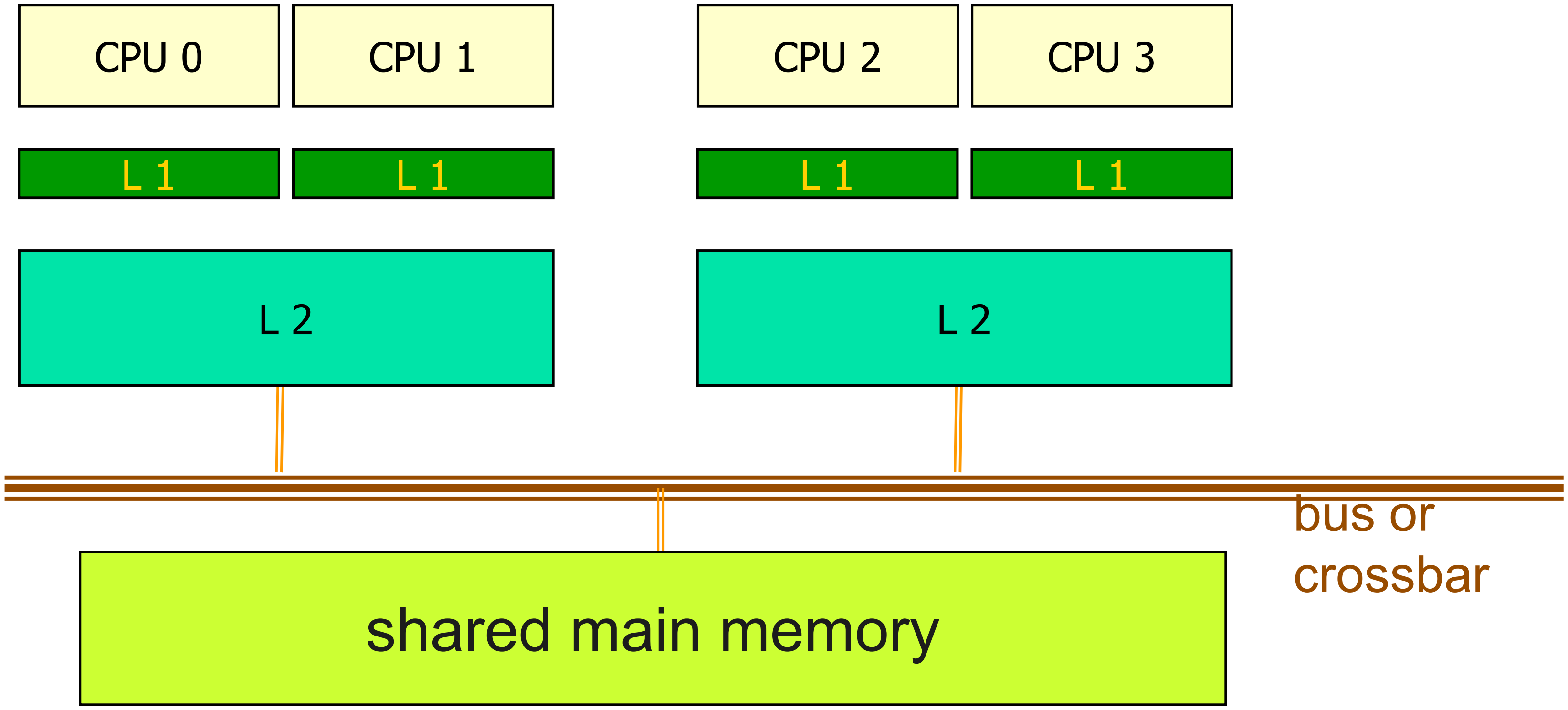
Multi Processor System



*trend to high-speed interconnection media

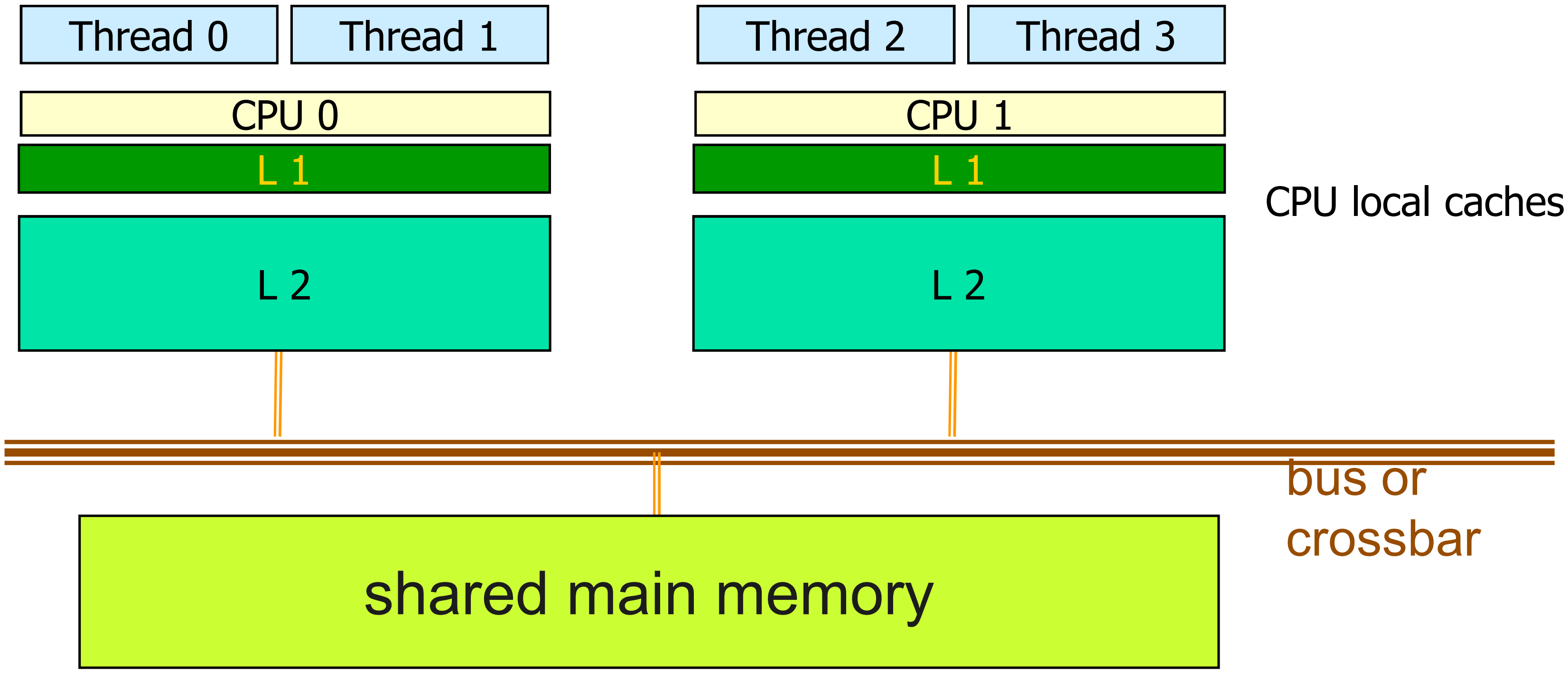
SMP Implementations

- Multicore SMP



SMP Implementations

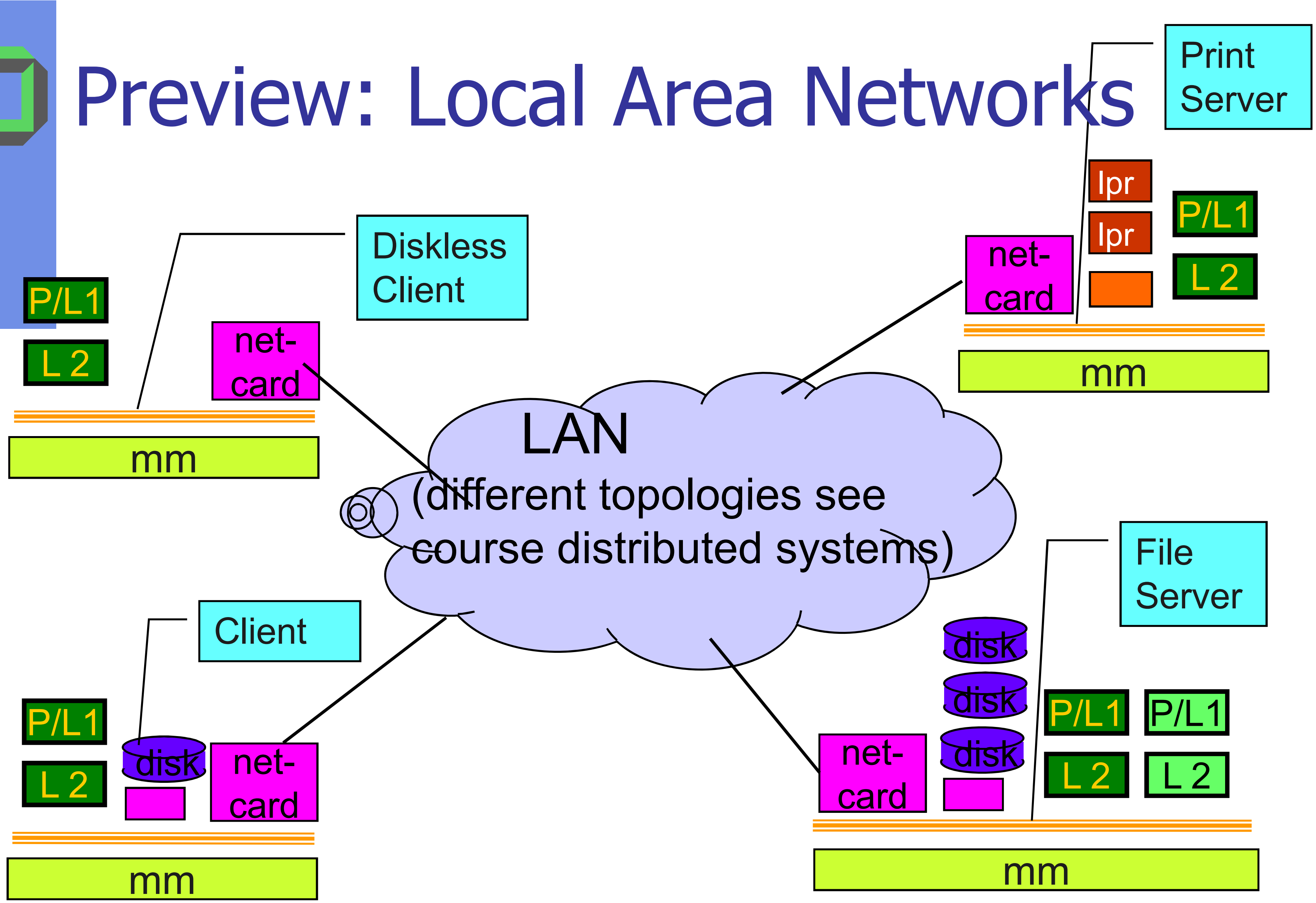
- Hyperthreaded SMP



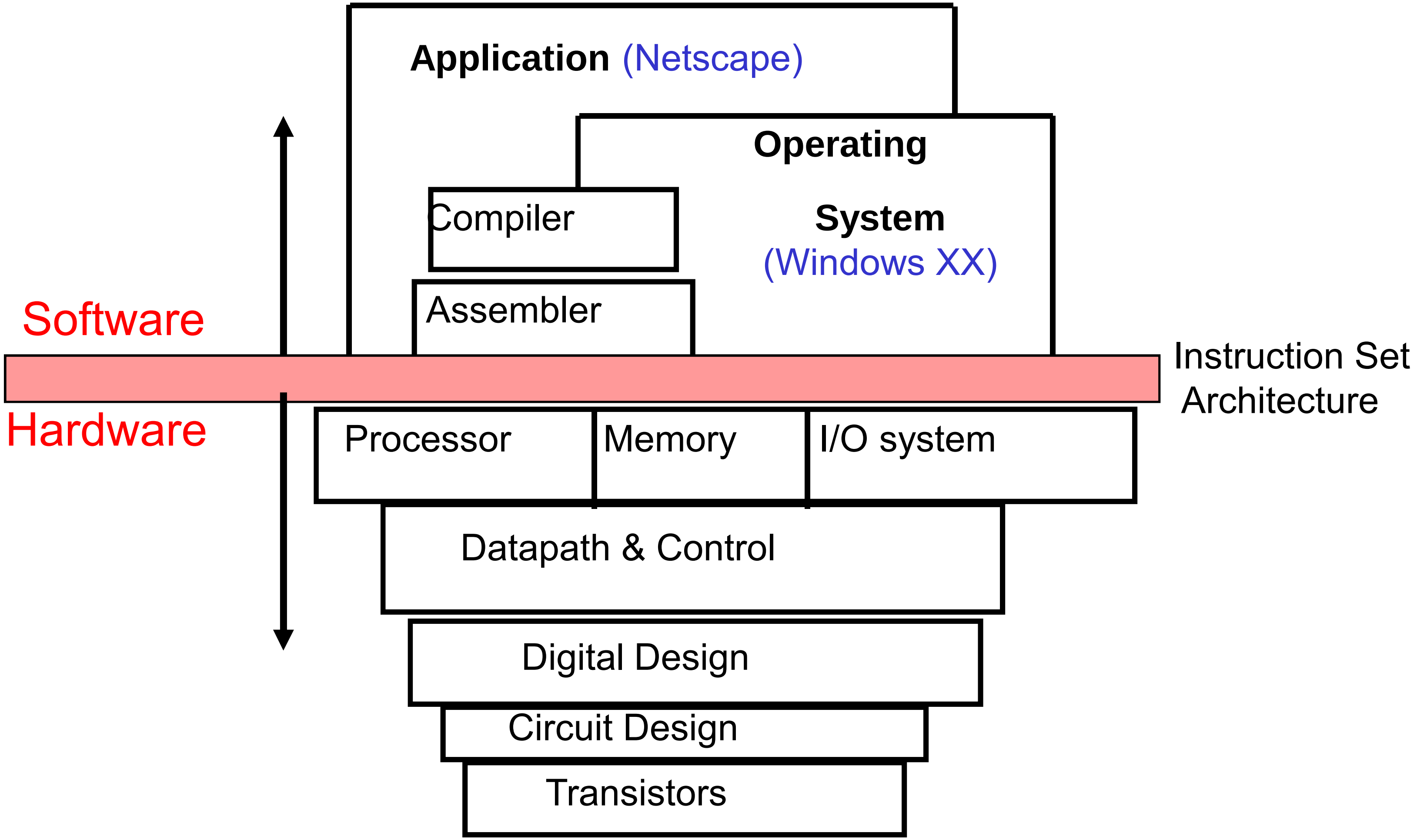
Caches

- Cache Architecture
- Cache Coherence
  - Write Through
  - Write Back
  - Snooping Protocols
    - MESI
    - MOESI
    - …
  - Memory Pinning
- Don’t miss it!!!
- *Why?*
  - It’s very interesting
  - You’ll need it in the examination
  - Liedtke et al: OS controlled cache predictability for real-time systems
  - Liedtke: Caches versus object allocation
  - …

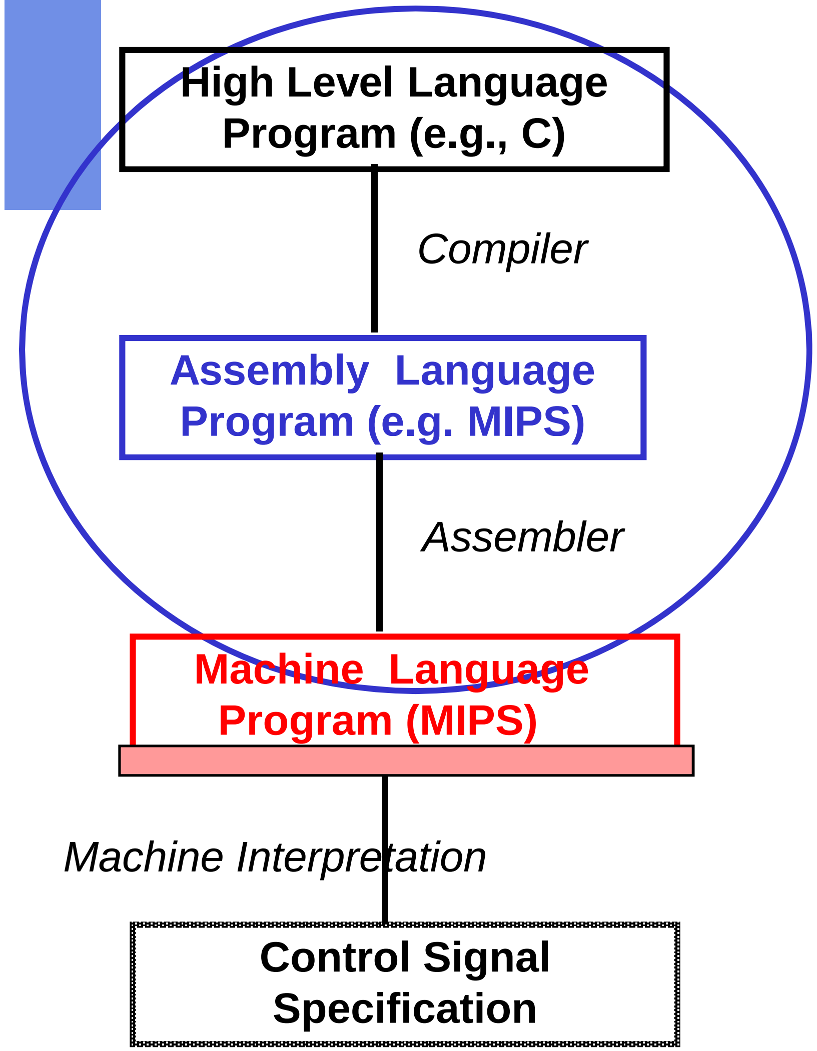
Preview: Local Area Networks



Computer Architecture



Levels of Representation



High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g. MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

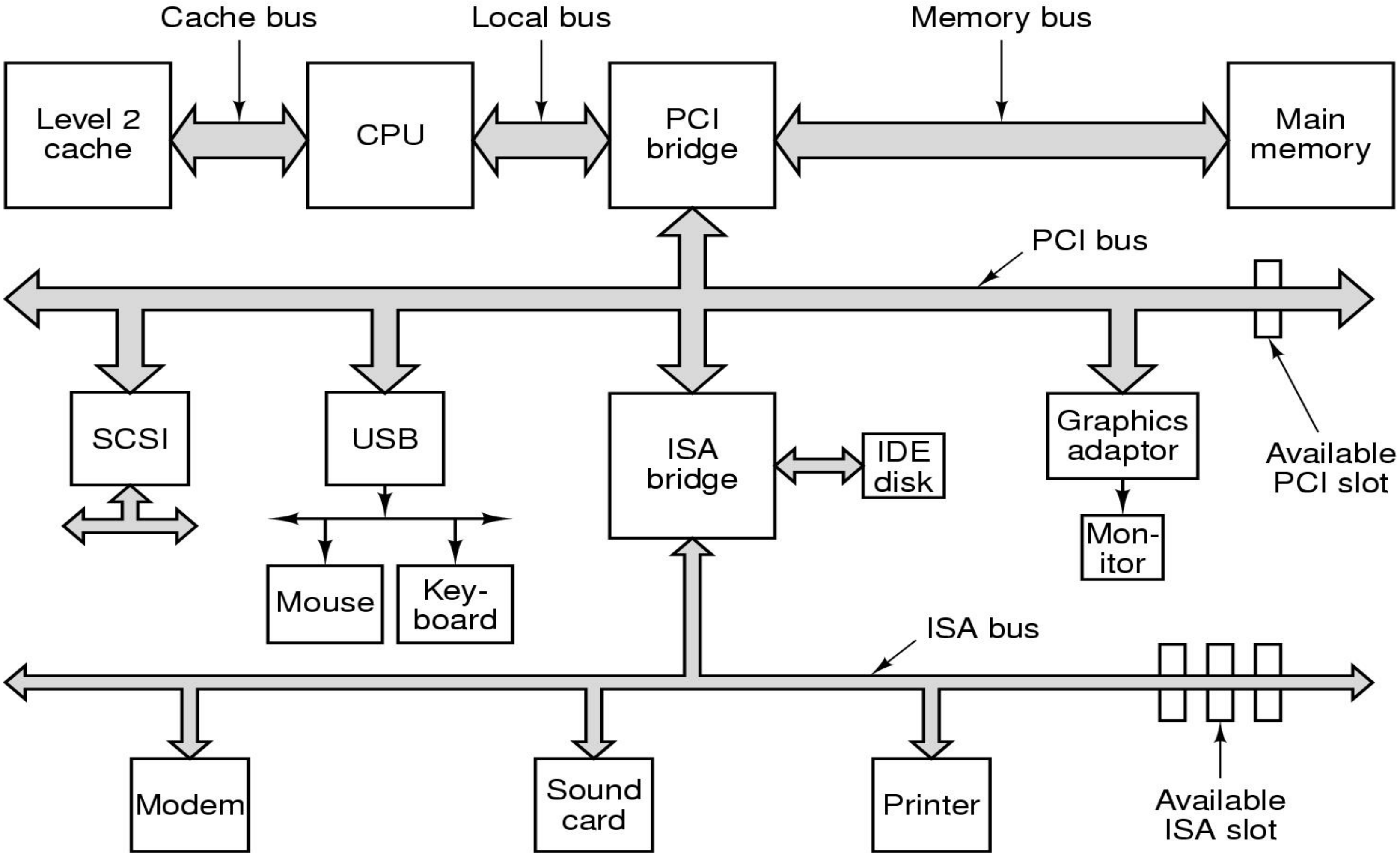
Control Signal Specification

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw$t0,0($2)
lw$t1,4($2)
sw$t1,0($2)
sw$t0,4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Architecture of a Pentium System



Trend in CPU Design?

- CISC → RISC → VLIW* → ?
- Concurrent execution on CPU
  - Pipelining
  - Superscalar Execution
  - Explicit parallel instruction set computer (EPIC)
  - Simultaneous multi-threading (SMT)
  - Speculative Execution

*Very Long Instruction Word, 1986 at IBM Watson Research Center, Idea: Expressing a program as a sequence of tree-instructions

What do Computers do?

- Computers manipulate representations of things, i.e. they interpret information!
- *What can you represent with N bits?*
  - 2^N things
  - Numbers! Characters! Pixels! Dollars! Positions! Instructions!
  - Depends on what operations you do with them

HW Components

CPU

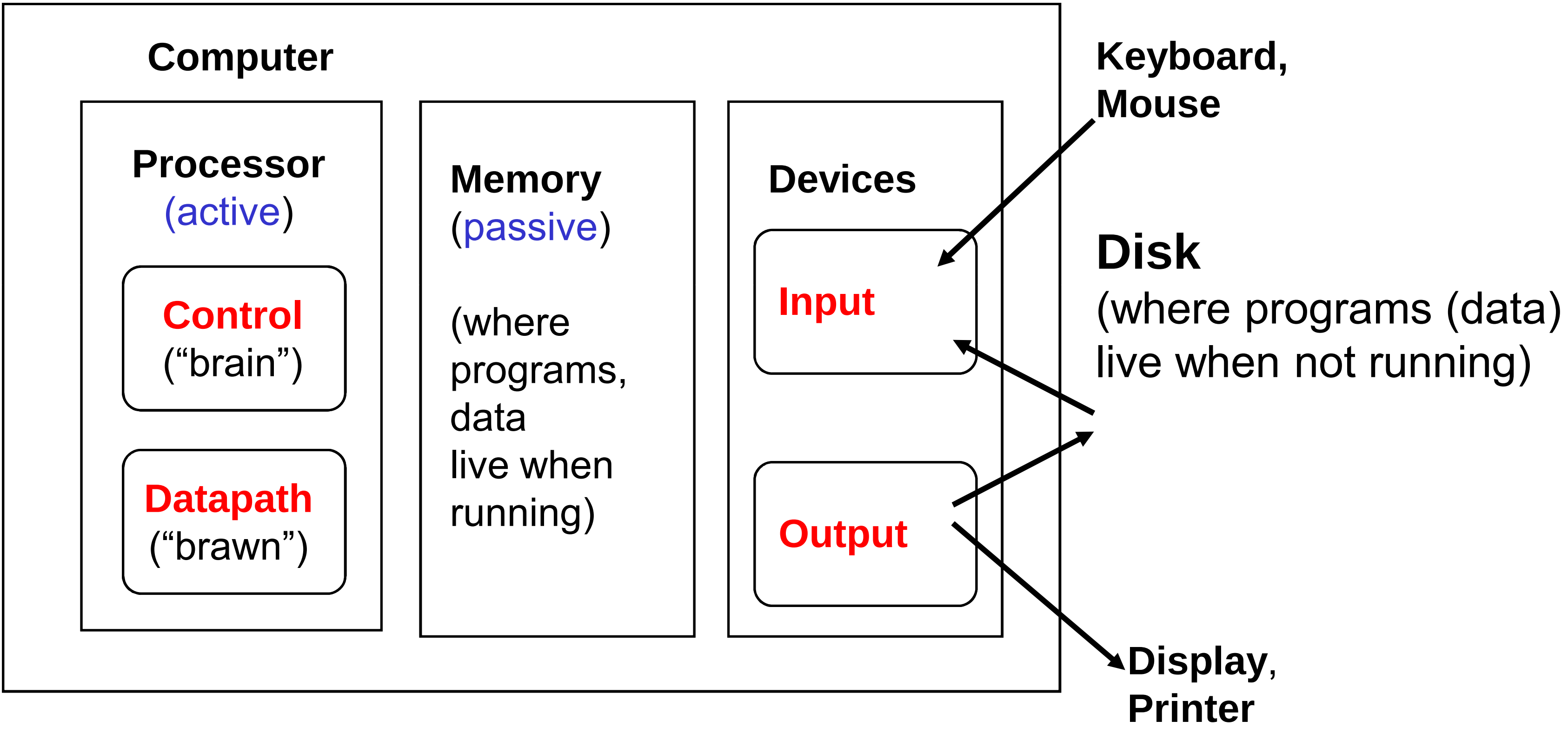
Memory

Interconnection

I/O Controller

I/O Devices

Components of a Computer



Computer

Processor
(active)

Control
("brain")

Datapath
("brawn")

Memory
(passive)

(where
programs,
data
live when
running)

Devices

Input

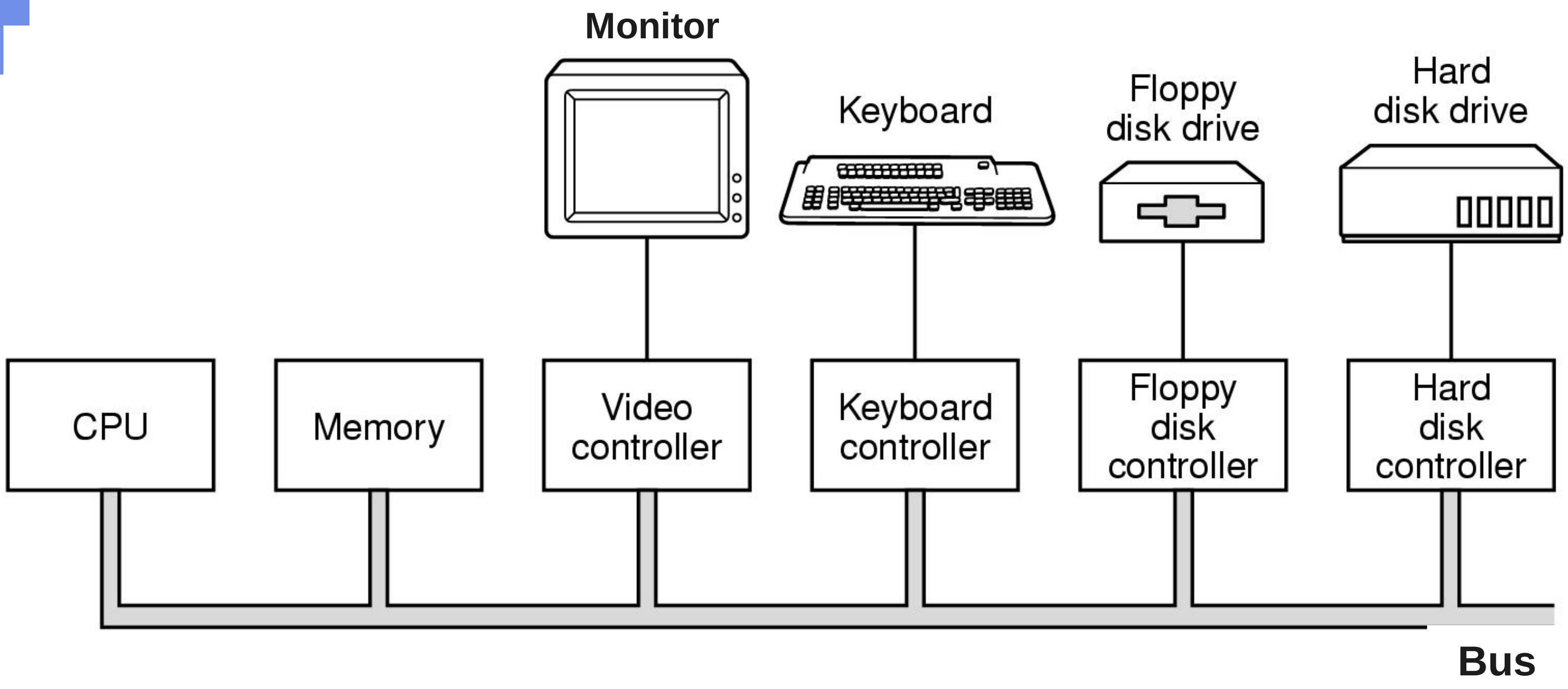
Output

Keyboard,
Mouse

Disk
(where programs (data)
live when not running)

Display,
Printer

Example: PC-Architecture



Basic Components

- Processors (CPUs)
- Memory hierarchy: (e.g. disk, RAM, caches)
- Interconnection (e.g. buses, cross bars)
  - Control and/or data lines
- I/O: controllers, channels, I/O-processors
  - hardware controlling devices and transporting data between devices, consisting of an extra CPU + memory, e.g. IDE controller, keyboard controller, network card, DMA, Timer / real-time clock, UART (RS232C, V.24, modem…)
- Peripheral Devices
  - disk, printer, keyboard, mouse, monitor, speaker, microphone, ...

Processor

Registers

Instructions

Modes

Simplified Processor

- Fetching instruction
- Executing instruction
- Moving data
- Manipulating data

Sophisticated processors may include

- Hyper Threading
- Instruction Pipelining
- Out-Of-Order Execution
- Explicit Parallel Instruction Set

Not in this course

Simplified API of a CPU

- Instruction Set
- General and Special Registers
- n ≥2 Execution or Processor Modes

Instruction Set

- Privileged Instructions
  - If executed in *user mode*

    → *HW raises an exception*
- Non-Privileged Instructions
  - Executable in *user-* as well as in *kernel-mode*

Give some examples

Why do we need privileged instructions?

General and Special Registers

- control / status registers

Instruction Pointer
Instruction Register
Status Register or
Processor Status Word
...

- user-visible registers

data registers
address registers
Index register
segment registers
stack pointer
condition codes / flags

OS Basics

- OS kernel = bunch of code sitting around in RAM, waiting to be executed
    - Triggered by system calls, exceptions, or interrupts
- “Kernel” gets control when system is booting
    - Depends a lot on the underlying hardware
    - Roughly, most PC have a BIOS (basic in/output system) that can access primitive hardware devices
        - Disk, keyboard, display
    - When powered on, BIOS has a small program that knows how to load a program from some I/O device(s), e.g. first try
        - Floppy, then CD ROM, then disk
- The program loaded by the BIOS is the kernel
    - Sometimes it’s actually a somehow simpler, prototype kernel
    - This prototype kernel knows how to load the ultimate kernel

Booting or Computer Startup

- Bootstrap program is loaded at power-up or reboot
  - Typically stored in ROM or EEPROM, generally known as firmware
  - Initializes all aspects of the system
  - Loads OS-kernel and starts its execution

Two[1] Execution Modes

- Kernel Mode
- User Mode
- *Why?*
- *What mode switches are typical?*

[1] ∃ architectures with more than 2 execution modes

User versus Kernel Mode

- *What makes a kernel different from user programs?*
  - Only kernel programs can execute privileged instructions
- Examples of privileged instructions:
  - Access I/O devices
    - Poll for I/O, perform DMA, catch hardware interrupt
  - Manipulate the MMU and memory state
    - Set up page tables, load/flush TLB
  - Configure various mode bits
    - Interrupt priority level, software trap vectors
  - Call HALT instruction
    - Put CPU into low-power or idle state until next interrupt
- Enforced by CPU
    - CPU checks current protection level on each instruction

Boundary Crossing[1]

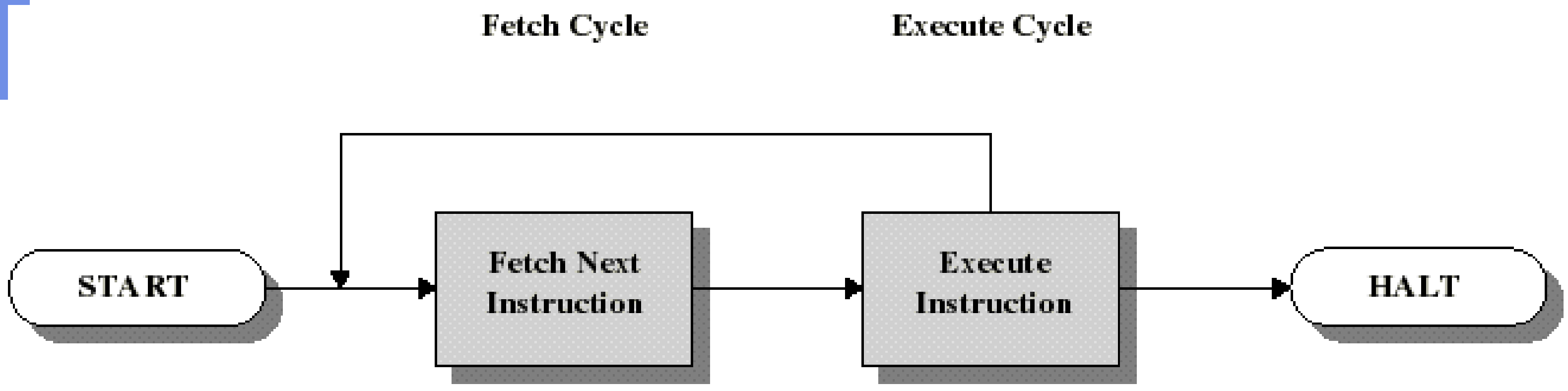
- User-to-kernel: *How does the kernel get control?*
  - At boot time: kernel loaded as the first OS program
  - System call: explicit call by an application into the OS
  - Exception, e.g. "division by Zero"
  - Hardware interrupt
  - Software interrupt
- Kernel-to-user: *How does an application gets control?*
  - OS sets up registers, protection domains, and MMU for the application to run the very first time
  - OS returns from a kernel activity and jumps to the next instruction of an application

[1] crossing user-kernel-boundary costs some/many cycles

Protection Rings

- ∃ multiple levels of protection domains, e.g.
  - x86 has 4 protection rings
    - Code in a less privileged ring can not directly call code in a more privileged ring
    - Ring 0 = OS kernel (~ kernel mode)
    - Ring 3 = application code (~ user mode)
    - Rings 1+2 can be used for less-privileged OS code
      - Third party device drivers

Basic Instruction Cycle



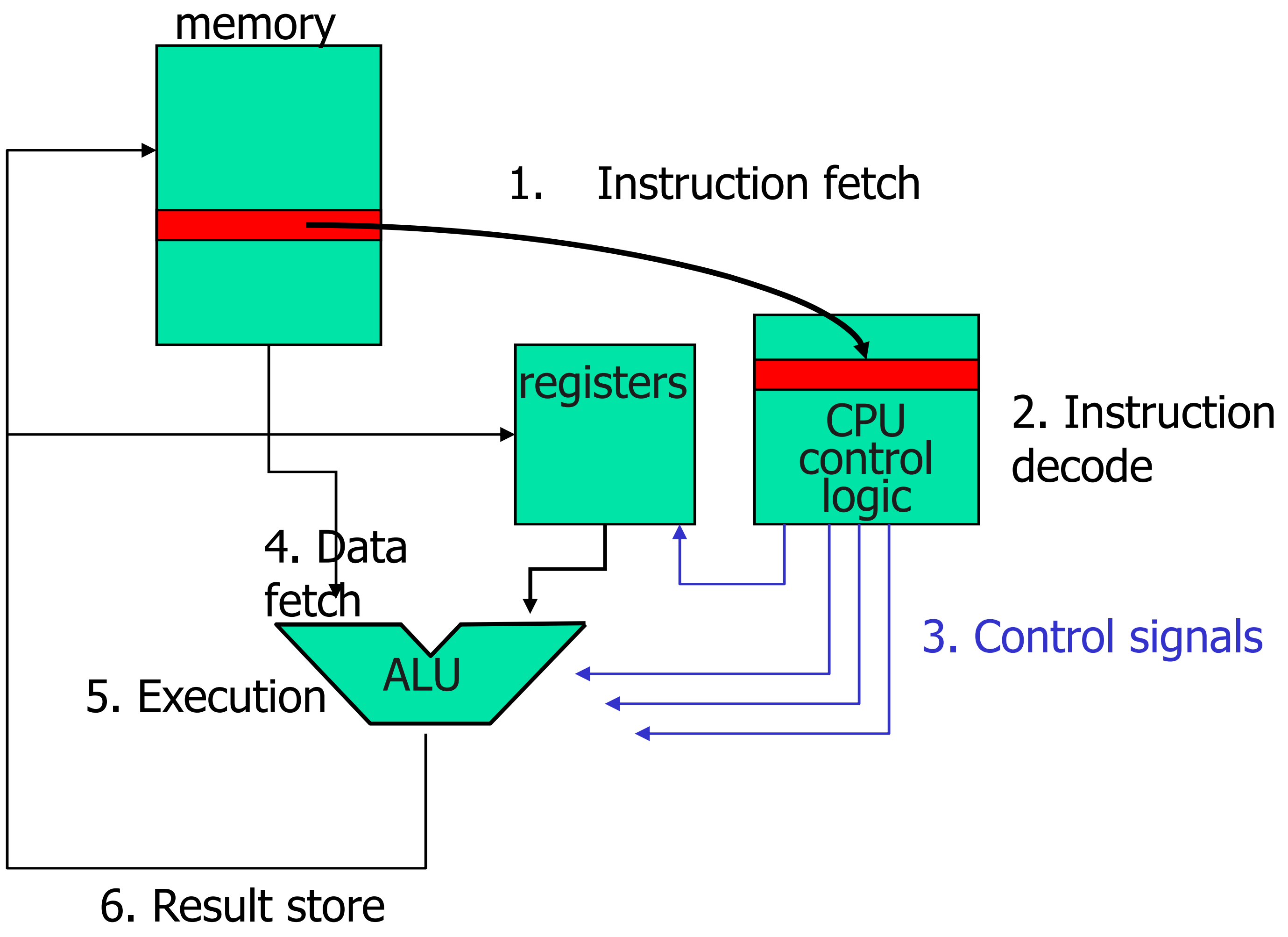
Basic Instruction Cycle

CPU fetches the next instruction (with operands) from memory.

CPU executes the instruction

Instruction Pointer (IP) holds address of the instruction to be fetched next, automatically incremented after each fetch

Execution Steps



1. Instruction fetch

2. Instruction decode

3. Control signals

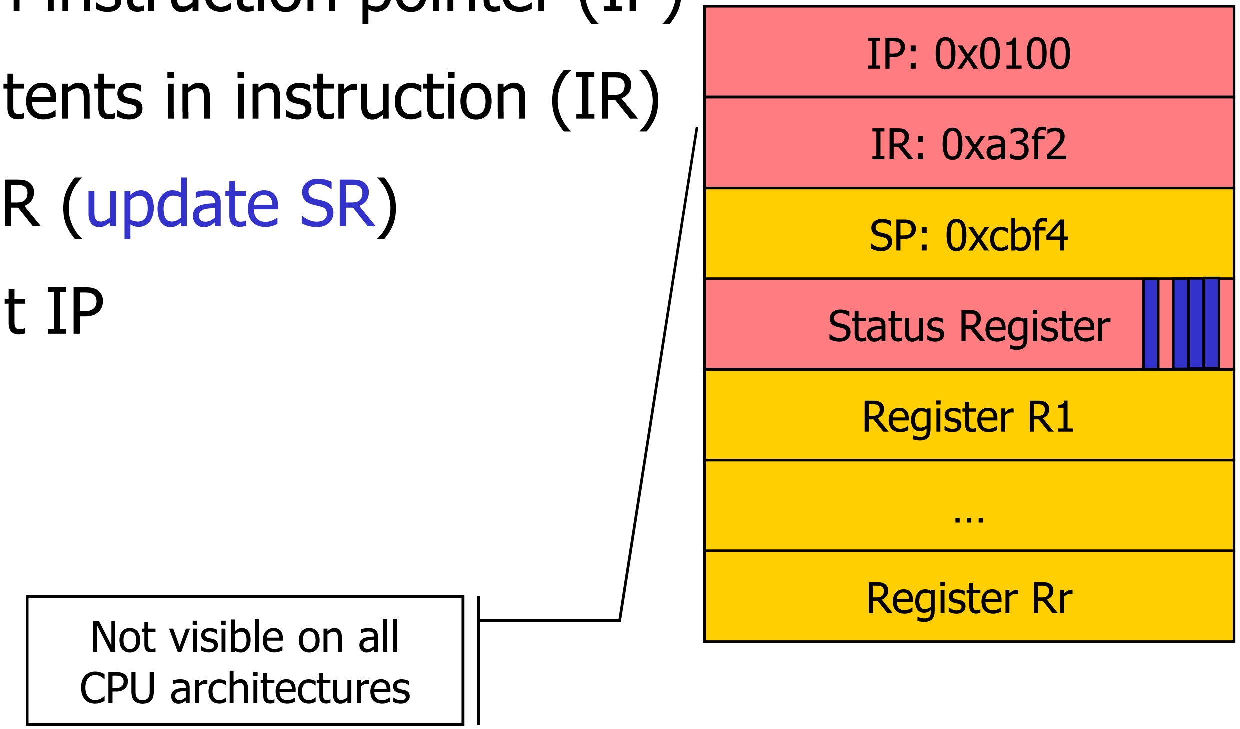
4. Data fetch

5. Execution

6. Result store

Simple Model of Computation

- Fetch-execute cycle
  - Load memory contents from address in instruction pointer (IP)
  - Store contents in instruction (IR)
  - Execute IR (update SR)
  - Increment IP
  - Repeat



IP: 0x0100

IR: 0xa3f2

SP: 0xcbf4

Status Register

Register R1

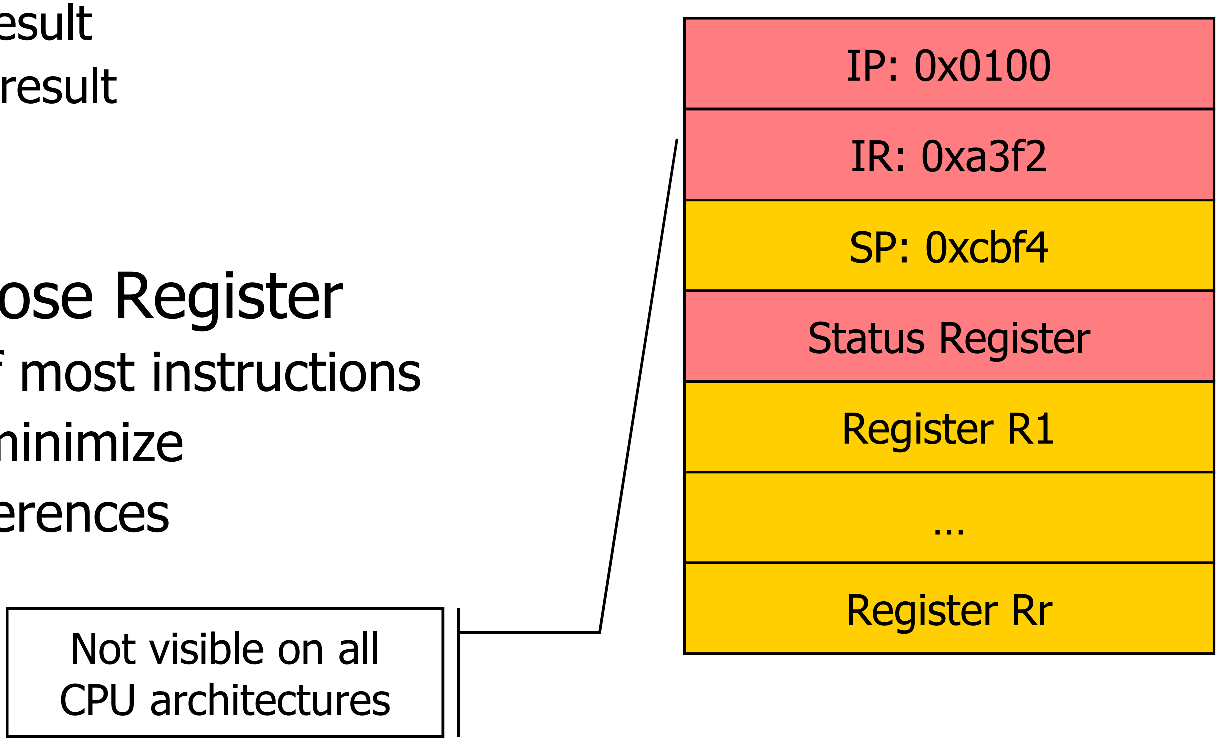
…

Register Rr

Not visible on all CPU architectures

Simple Model of Computation

- Stack Pointer
- Status Register
  - Condition Code
    - Positive result
    - Negative result
    - Zero
  - Mode bit(s)
- General Purpose Register
  - Operands of most instructions
  - Enables to minimize memory references


IP: 0x0100

IR: 0xa3f2

SP: 0xcbf4

Status Register

Register R1

…

Register Rr

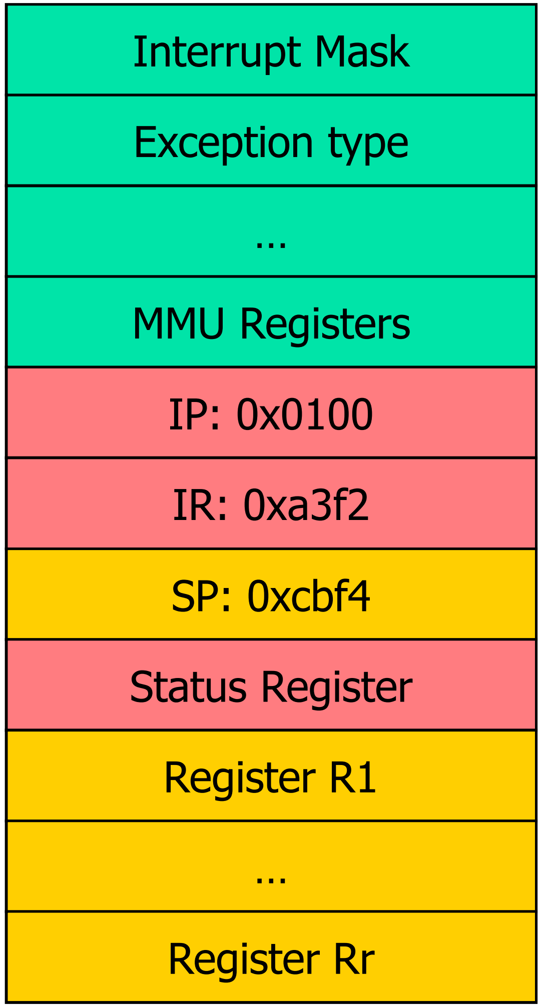
Not visible on all CPU architectures

Privileged Mode Operation

- To protect OS execution

  m ≥ 2 modes are available
  - Kernel (system) mode
    - All instructions are executable
    - All registers are accessible
  - User mode
    - Only "safe" subset of instructions are executable, e.g. not allowed:

      `disable interrupts`
    - Only "safe" registers are accessible



Interrupt Mask

Exception type

…

MMU Registers

IP: 0x0100

IR: 0xa3f2

SP: 0xcbf4

Status Register

Register R1

…

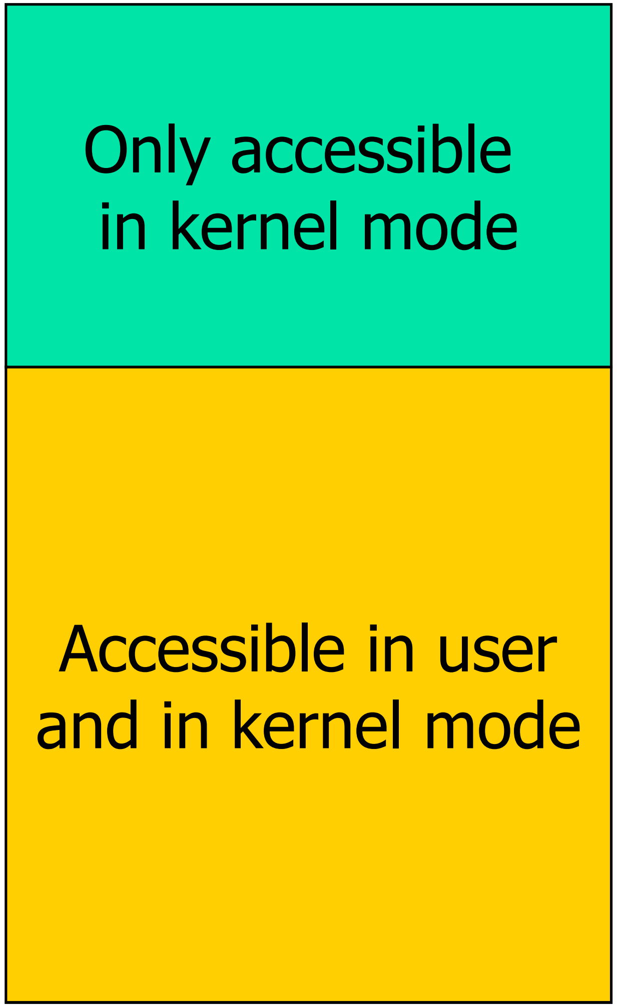
Register Rr

Safe Instructions and Registers

- Instructions and registers are safe if
  - Only affect application itself
  - Cannot be used to interfere with
    - OS
    - Other applications
  - Cannot be used to violate OS policy

Privileged Mode Operation

- Accessibility of addresses in an address space changes according to execution mode
  - To protect kernel code and kernel data
- Result: In principle, no application can harm the kernel



Only accessible in kernel mode

Accessible in user and in kernel mode

Atomicity

What does it mean?

- ∃ atomic instructions, used to implement locks, e.g.
  - Test-And-Set (TAS), if word contains a given value, set to new value
  - Compare_And-Swap (CAS), if word equals value, swap old value with new value
  - …
- ∃ atomic operations
  - …
- ∃ atomic program sections
  - see critical section

Hint: Read http://www-128.ibm.com/developerworks/library/pa-atom/

Memory

Memory Hierarchy

Caching

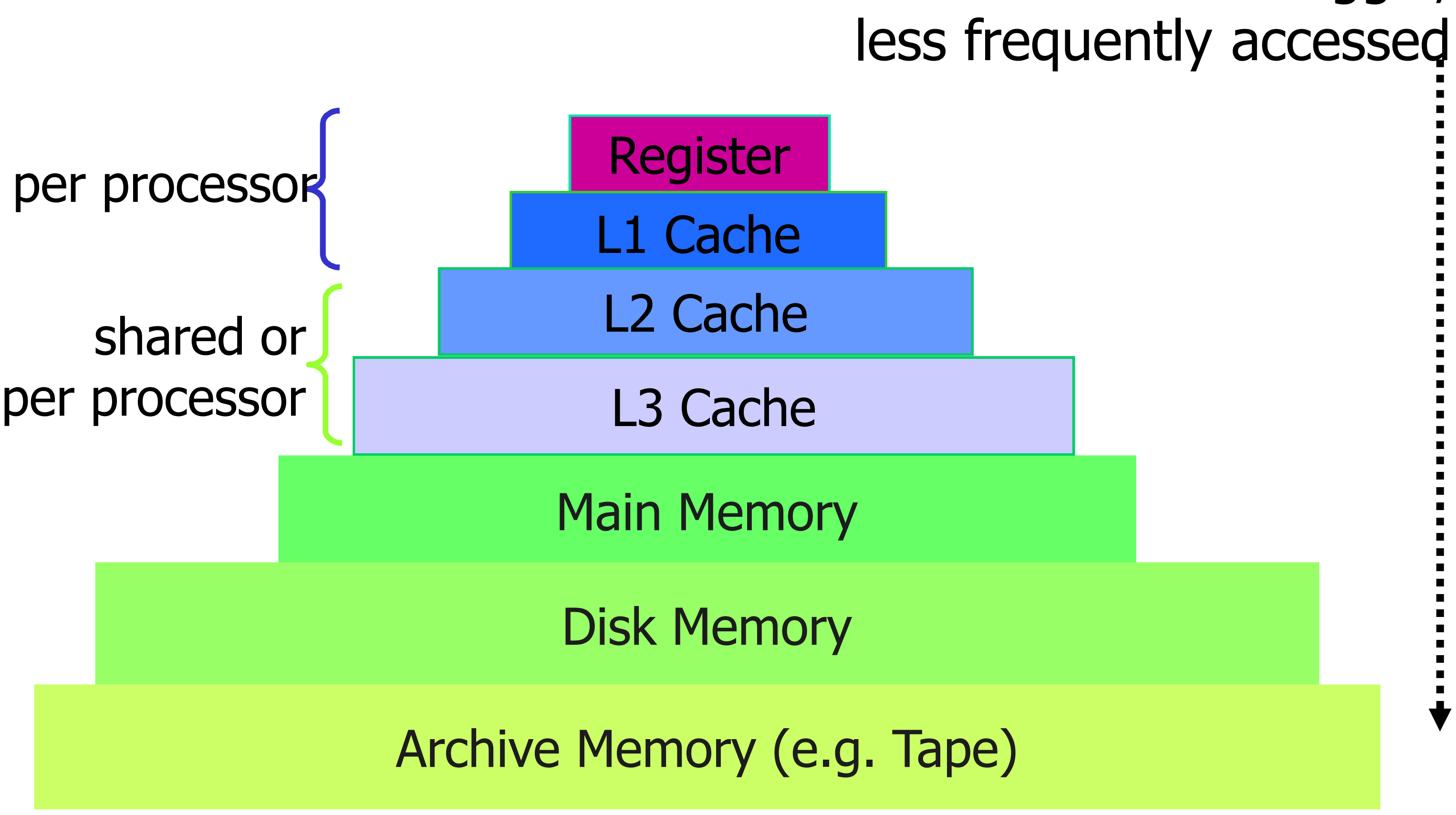
Locality

Protection

Why Memory Hierarchy?

- ∃ huge performance gap between modern CPUs and main memory (RAM)
- ∃ a principle to overcome this gap
  - Locality
    - Temporal locality
    - Spatial locality

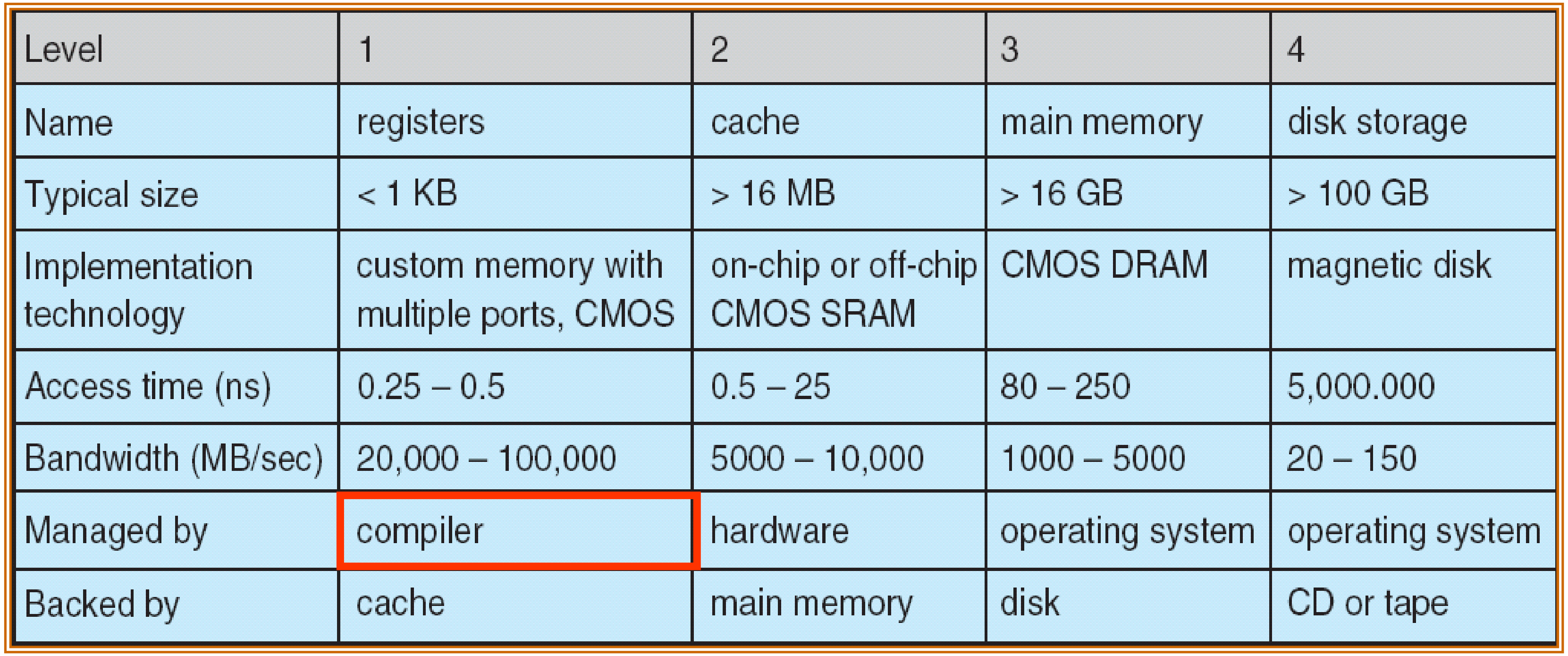
Memory Hierarchy



Note: Parts of main memory may be used as a disk cache

Some Newer Numbers[1]

- Movement between levels of storage hierarchy can be explicit or implicit



Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

[1] Silberschatz et al: OS Concepts, 7th Edition

Unix Original Machine (1969)

- PDP-7 = an 18 bit machine with a cycle time of 1.75 µs

  24 KB ~ practical limit of core memory at that time. PDP-7 first DEC computer designed for automated wire wrapping.
- Prize: ~ 72.000 $

Ritchie & Thompson + PDP 11

Unix needed 16 KB*

Users could only get 8 KB for their applications

*at that time a really tiny OS

Registers*

- Registers
  - General purpose
  - Floating point
  - Multimedia
  - Special (instruction pointer, status)
- Typical for RISC architectures:
  - 32 general purpose (32 bit or 64 bit)
  - 32 floating point (64 bit IEEE)
  - Multimedia (64, 128, or 256 bit)
- Intel
  - IA 32 Pentium
    - 8 general purpose, 8 floating point (or 8 multi media)
  - IA 64 Itanium
    - 128 general purpose, 128 floating point

*Register windows

Observation

- To support the execution of programs, faster -but limited caches- are useful to keep up with modern CPUs
- "Natura non saltat" ~ evolution takes time

  ~ program execution has some locality

  i.e. what's in the caches will be reused

Principle of Locality

Observation:

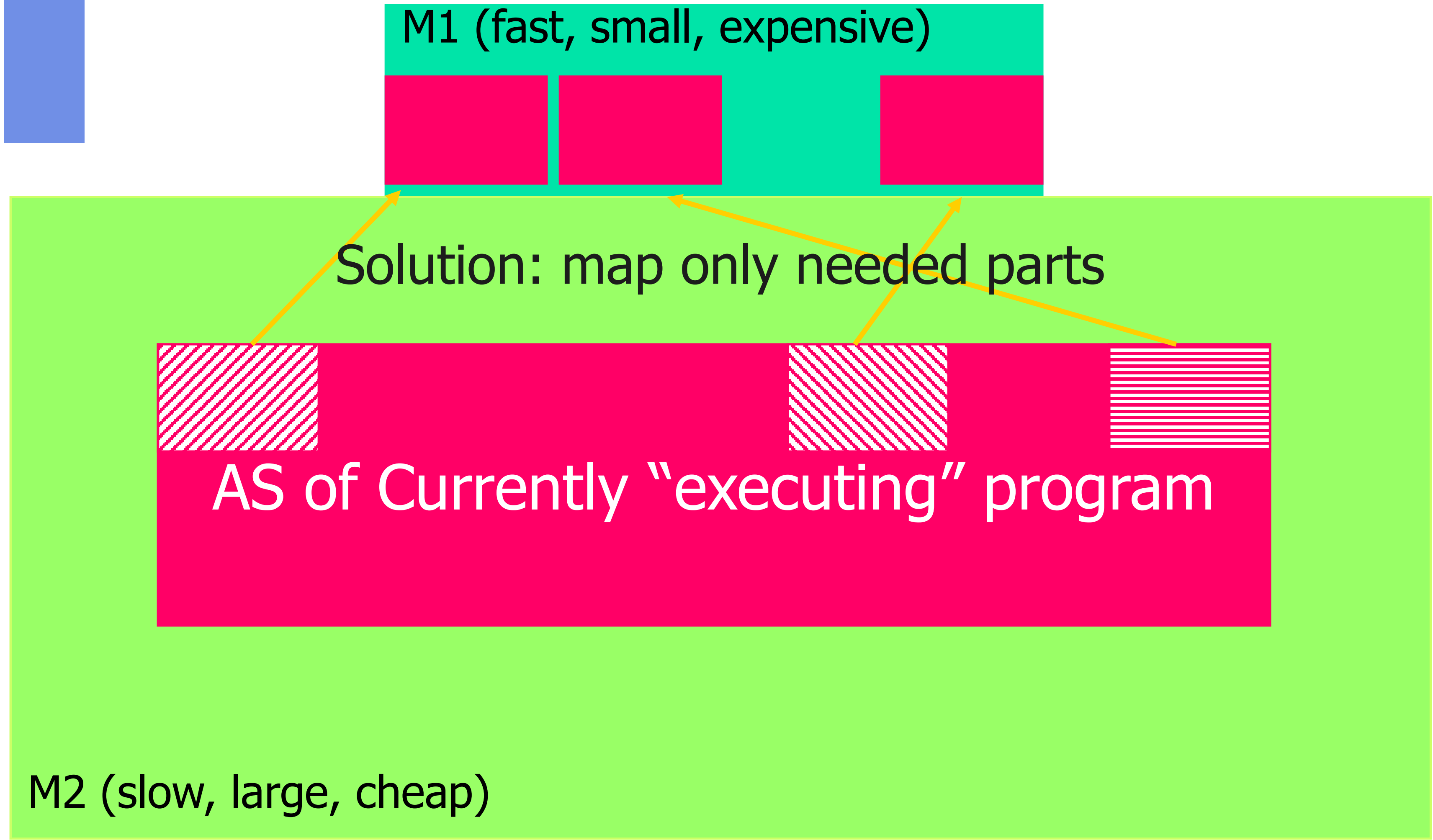
10 % of code does 90% of work*

**Principle a “law of nature” ?*

Types of Locality

- Spatial Locality
  - near addresses are accessed next
    - instructions ahead
    - local variables of a procedure
    - next elements of an array / a structure
- Temporal locality
  - Frequently used addresses (bursts)
    - instructions inside a loop
    - frequently called procedures
    - “important” variables
    - top of the stack

Locality and Memory Hierarchy?


M1 (fast, small, expensive)
Solution: map only needed parts
AS of Currently "executing" program
M2 (slow, large, cheap)

Analysis of a 2-Level Memory

(write-through cache)

$$T_{avr} = (1\text{-MissRate})*T_{cache} + \text{MissRate} * T_{ram}$$

T_{cache} = 1 ns

T_{ram} = 400 ns

Example

MissRate: 5 %

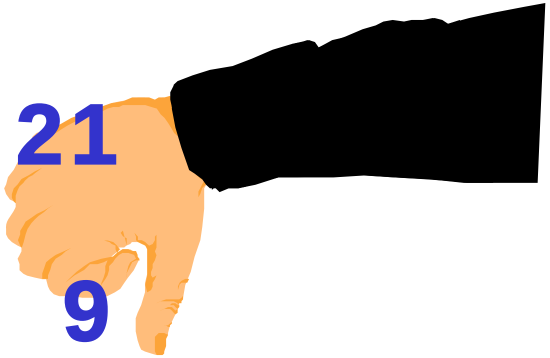
ns

T_{avr} ~ 21

2 %

ns

T_{avr} ~ 9



Simplified Analysis of 3-Level Memory

(write-through caches, inclusion property assumed)

$$T_{avr} = T_{L1} + Miss_{L1} * T_{L2} + Miss_{L2} * T_{ram}$$

T_{L1} = 1 ns $\quad$ $Miss_{L1}$: 5% $\quad$ 3%

Example

T_{L2} = 20 ns $\quad$ $Miss_{L2}$: 0.5% $\quad$ 0.3%

T_{ram} = 400 ns

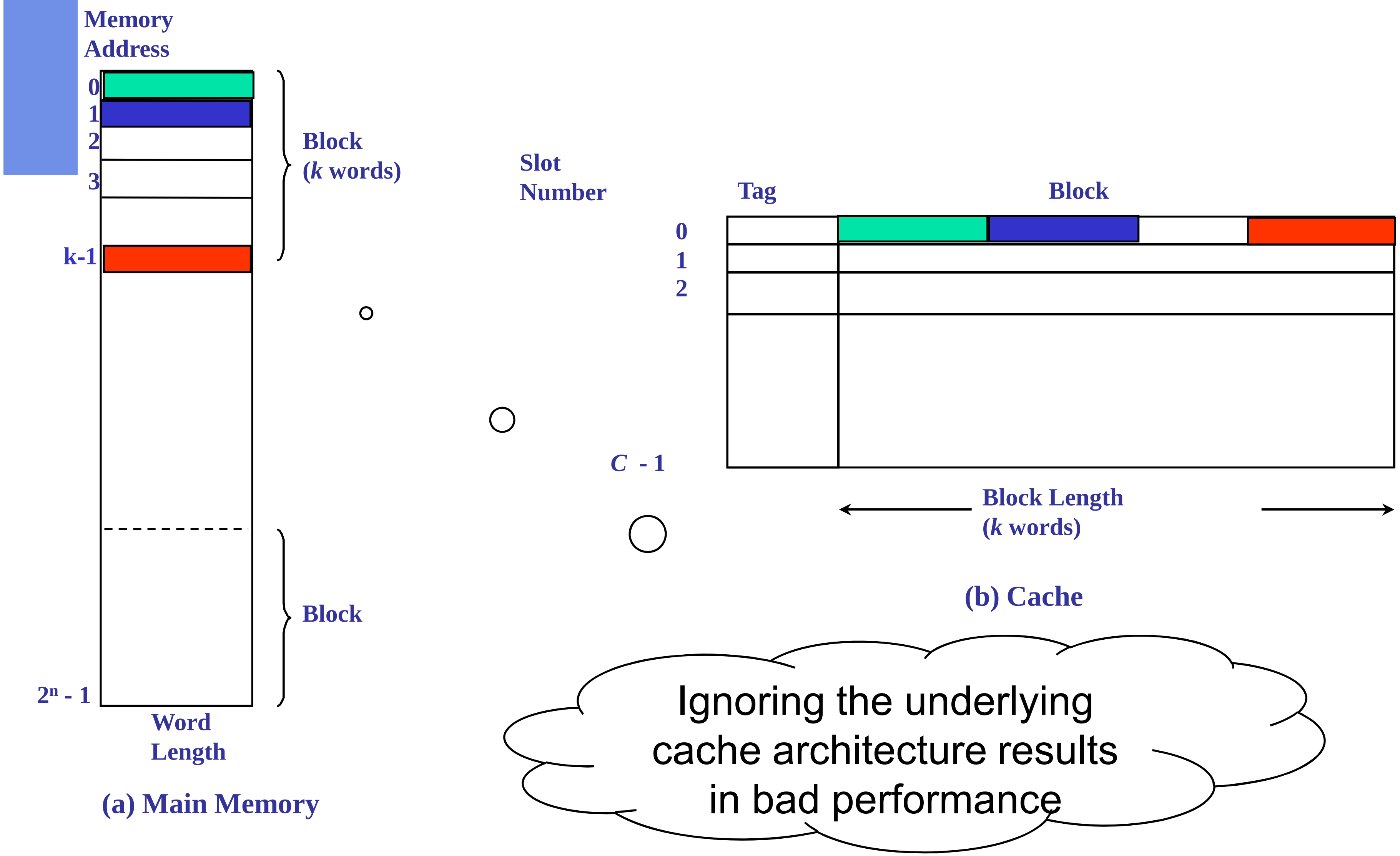
$T_{avr} = ?$

Cache Design Parameters[1]

Size	8K - 265K (L1), 64K - 8M (L2), 1 M – 16 M (L3)
Block Size	(32 … 128 B)
Access Time	(1 … 10 ns)
Mapping	(full associative, n-way-associative, direct-mapped)
Replacement	(LRU, FIFO,...)
Write Policy	(write through, write back)
Additionals	*(victim cache, exclusive ~)*

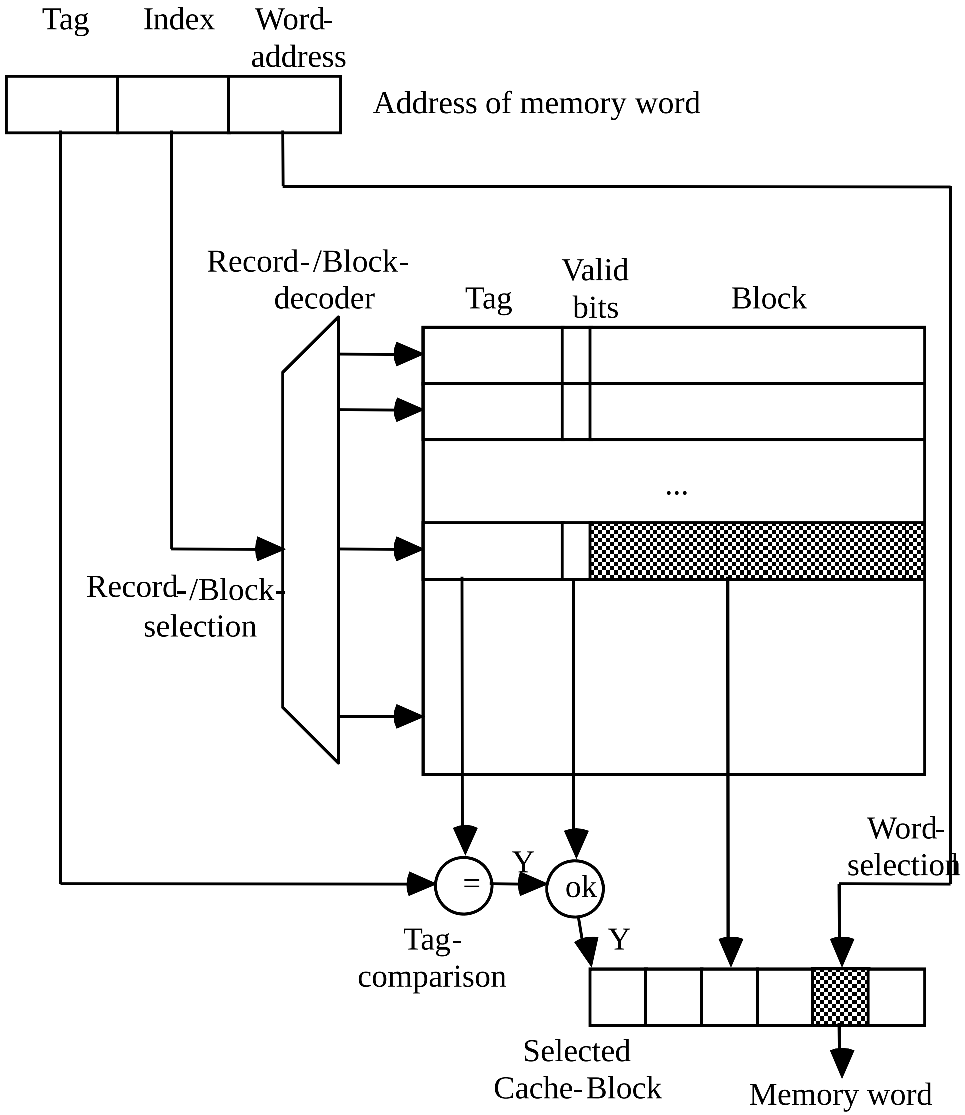
[1] Adjust to current values

Cache Memory Architecture

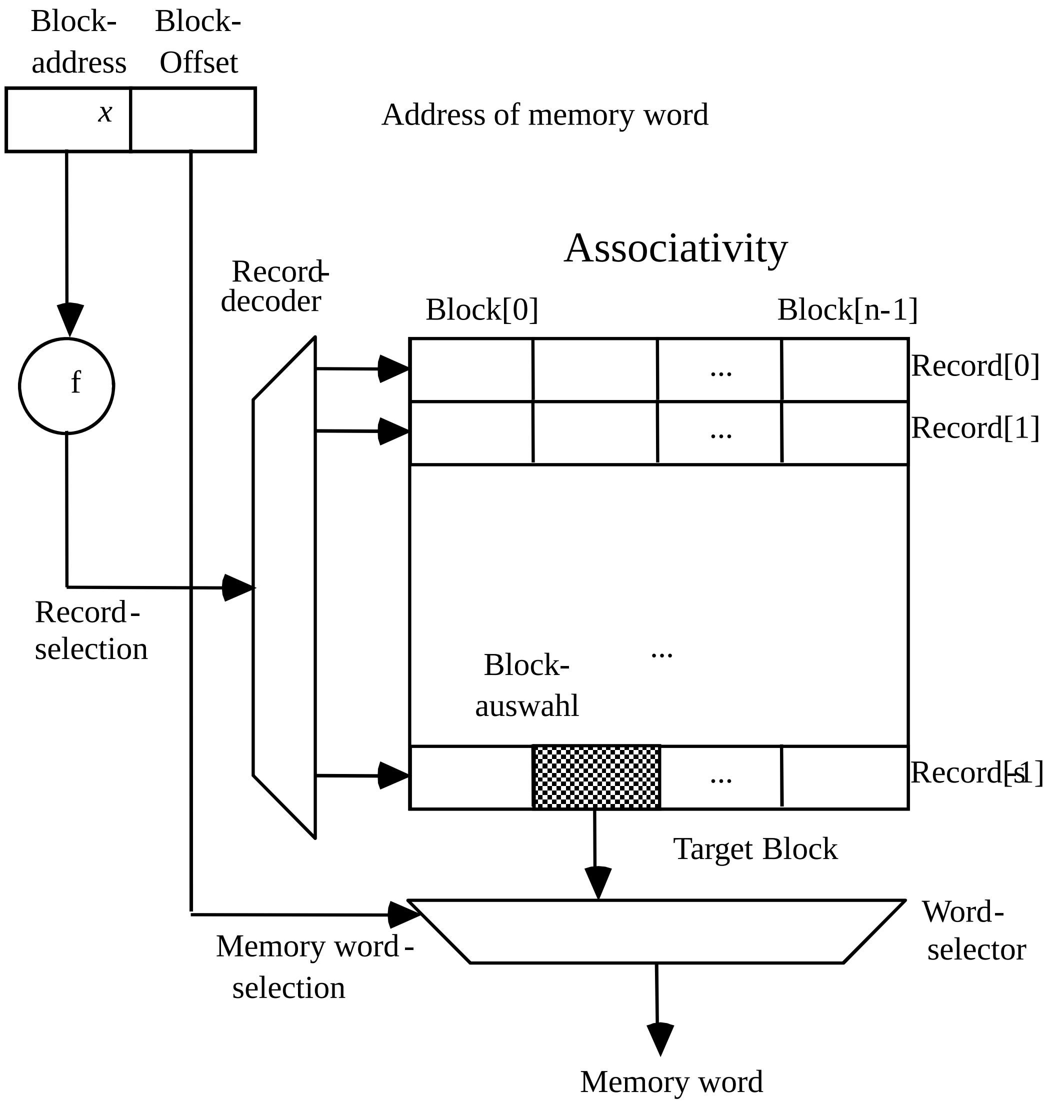


Ignoring the underlying cache architecture results in bad performance

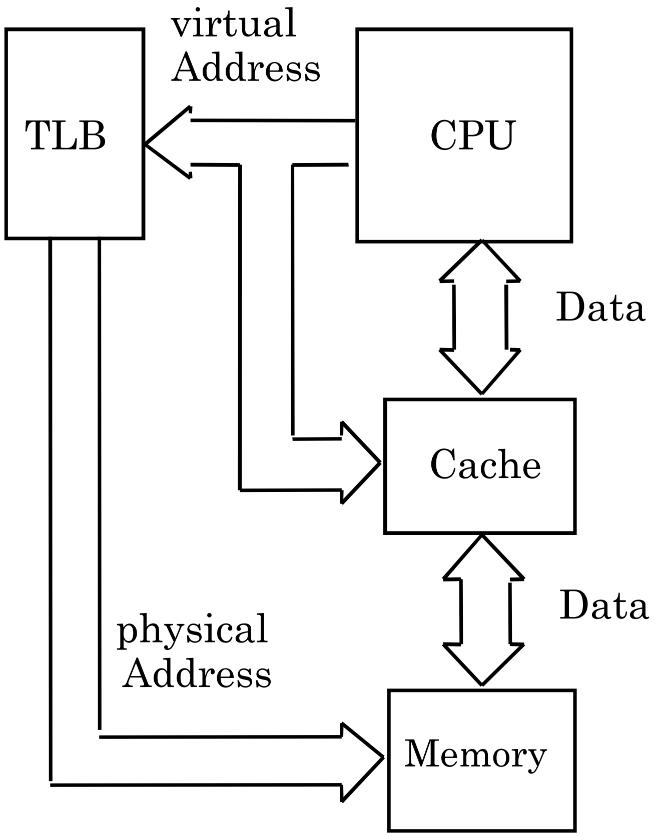
Direct Mapped Cache



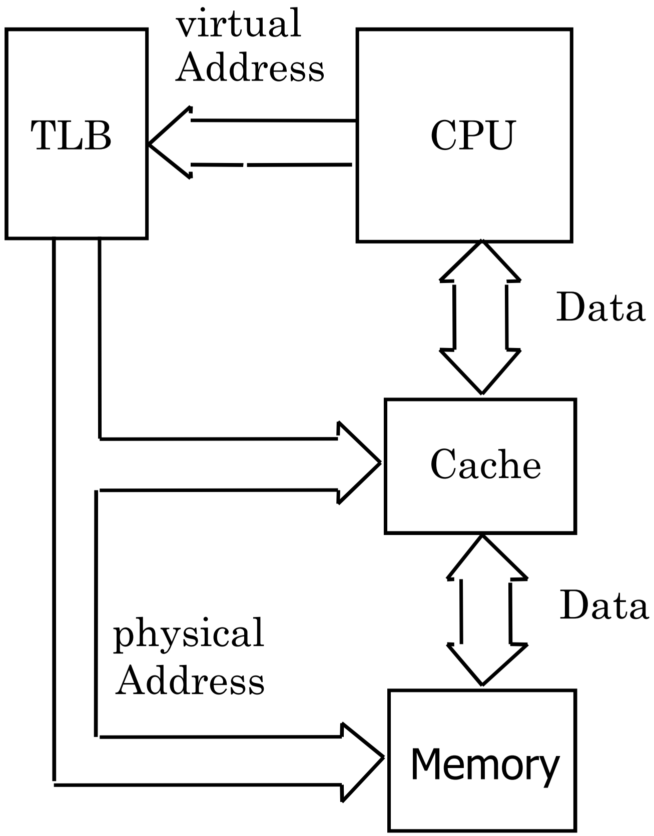
N-Way Set Associative Cache



Cache Addressing Schemes



(a) Virtual Cache



(b) Physical Cache

Read **www.ee.umd.edu/~blj/papers/micro18-4.pdf**

Cache Memory Design

- Cache Size
    - Even mall caches have an impact on performance
- Cache Line Size (Block size)
    - Unit of data exchanged between cache and main memory
    - Hit means the information was found in the cache
    - Larger block size ⇒ better hit rate, until probability of using newly fetched data becomes less than the probability of reusing data that has been moved out of cache
    - Miss means data is not present in a cache line

Cache Memory Design

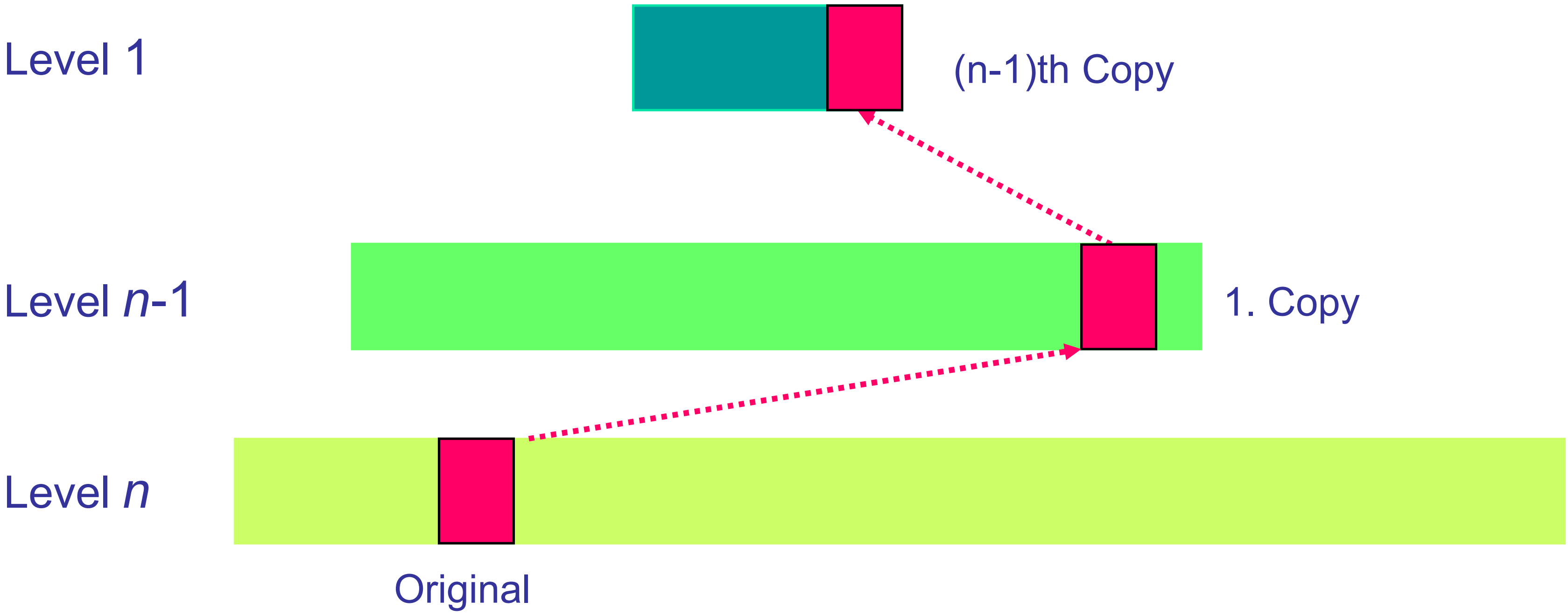
- Mapping function
  - determines cache location, a new block will occupy
- Replacement algorithm
  - determines the block, which has to be replaced
  - 2 commonly used methods
    - Least-Recently-Used (LRU) algorithm
    - FCFS (in victim caches)

Cache Memory Design

- Write policy determines
  - when a block of cache is written to main memory
  - can occur every time a cache line is updated (write-through policy)
  - can occur only when the cache line has to be replaced (write-back policy)
    - Minimizes main memory operation
    - Leaves main memory in an obsolete state

Buffering/copying within a “write-back” memory hierarchy

Assumption: Original of data item within Memory Level *n*



Level 1

(n-1)th Copy

Level *n*-1

1. Copy

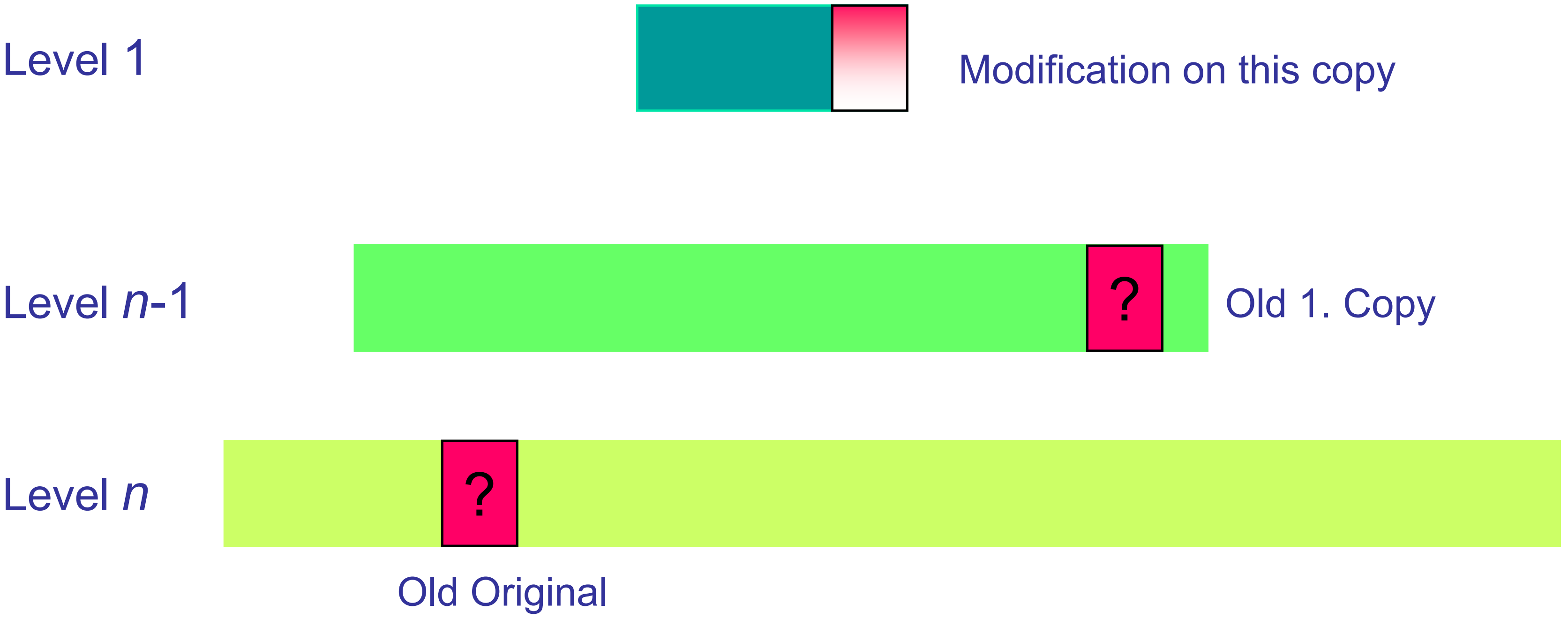
Level *n*

Original

Observation: Before accessing the first time, this data item has to be copied several times

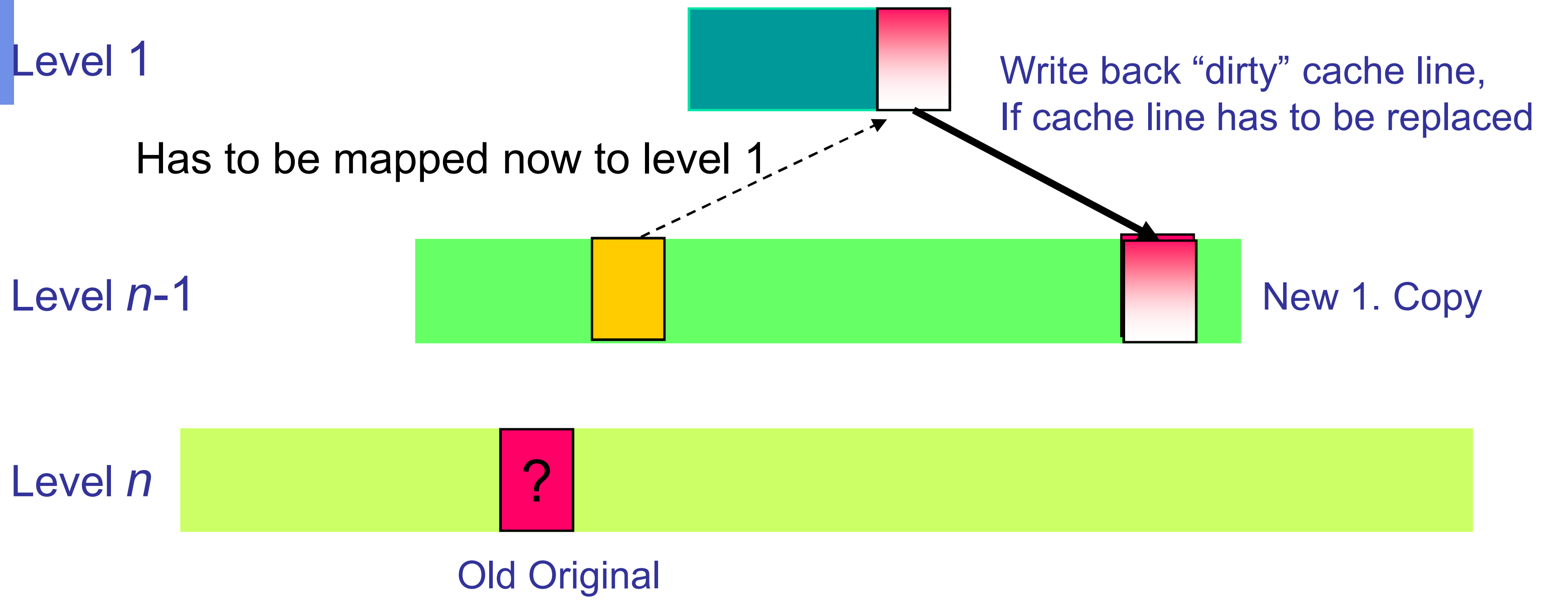
Buffering/copying within a “write-back” memory hierarchy

Assumption: Original of data item within Memory Level *n*



Observation: Modifying a data item on the uppermost memory level affects data consistency.

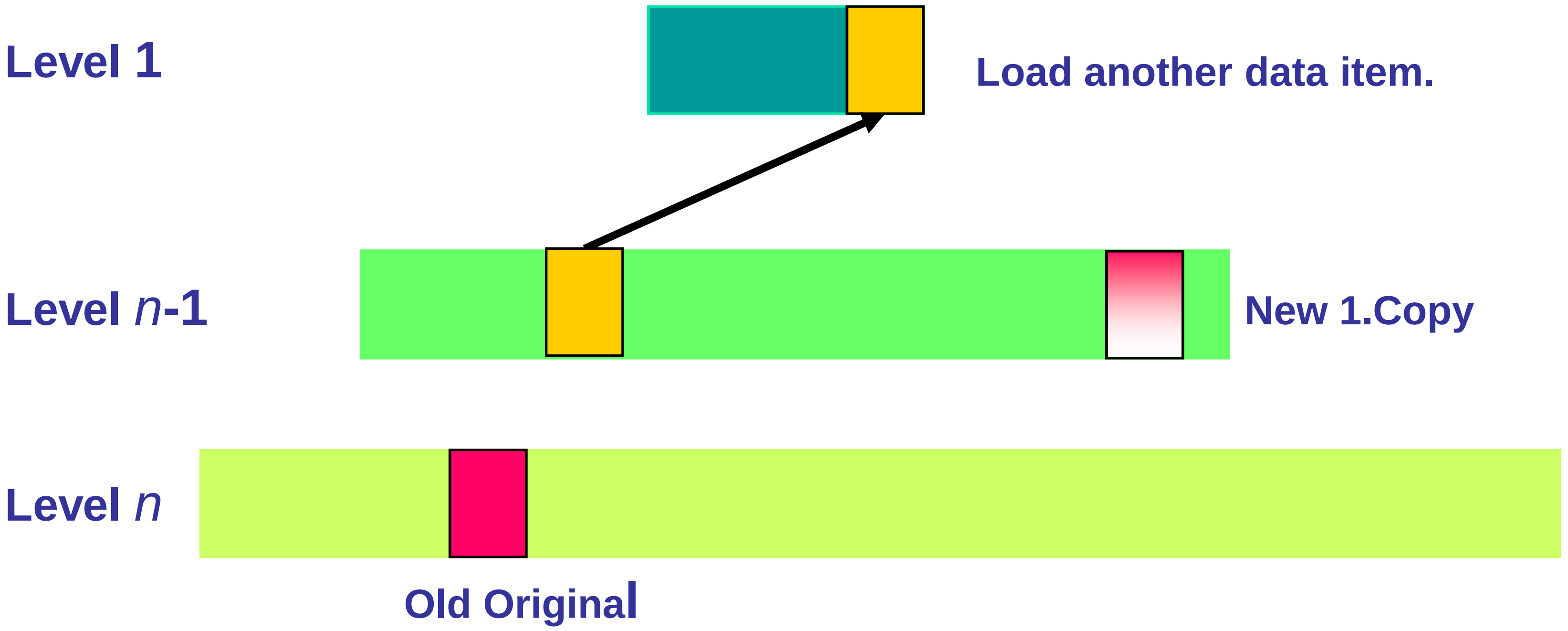
Buffering/copying within a “write-back” memory hierarchy


Level 1
Write back “dirty” cache line,
If cache line has to be replaced
Has to be mapped now to level 1
Level *n*-1
New 1. Copy
Level *n*
?
Old Original

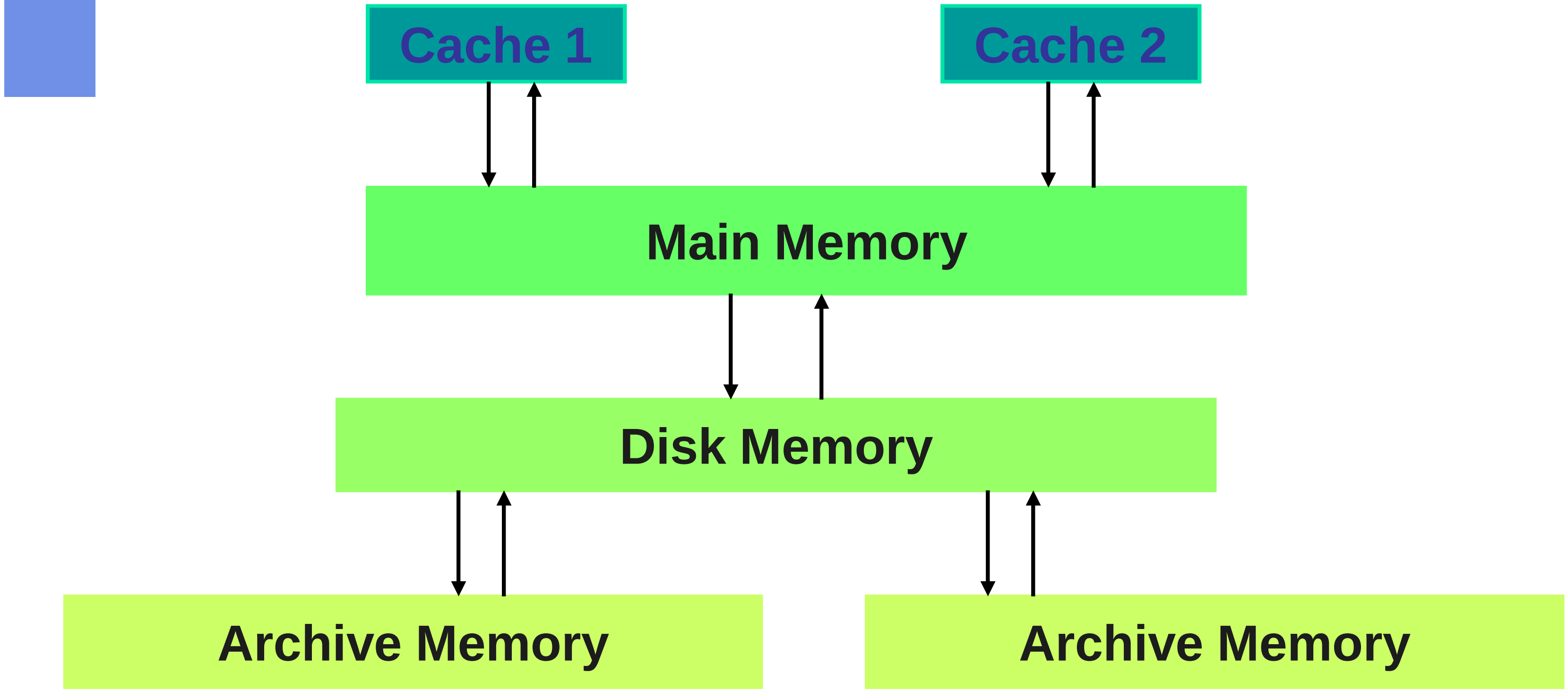
<u>Observation:</u> Propagation of modified data to lower levels can be done at once or stepwise, but has to be done only, if the upper dirty cache line has to replaced

Buffering/copying within a “write-back” memory hierarchy

Assumption: Original of data item within Memory Level *n*



Branching Memory Hierarchy



Questions: Why 2 archive memories? Additional levels with similar branches?
Additional consistency problems?

Specific Literature

- J. Liedtke „Caches Versus Object Allocation“, In 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA, October 1996.

- J. Liedtke „Potential Interdependencies between caches, TLBs, and memory management schemes“, Arbeitspapiere der GMD, No. 962, 1995

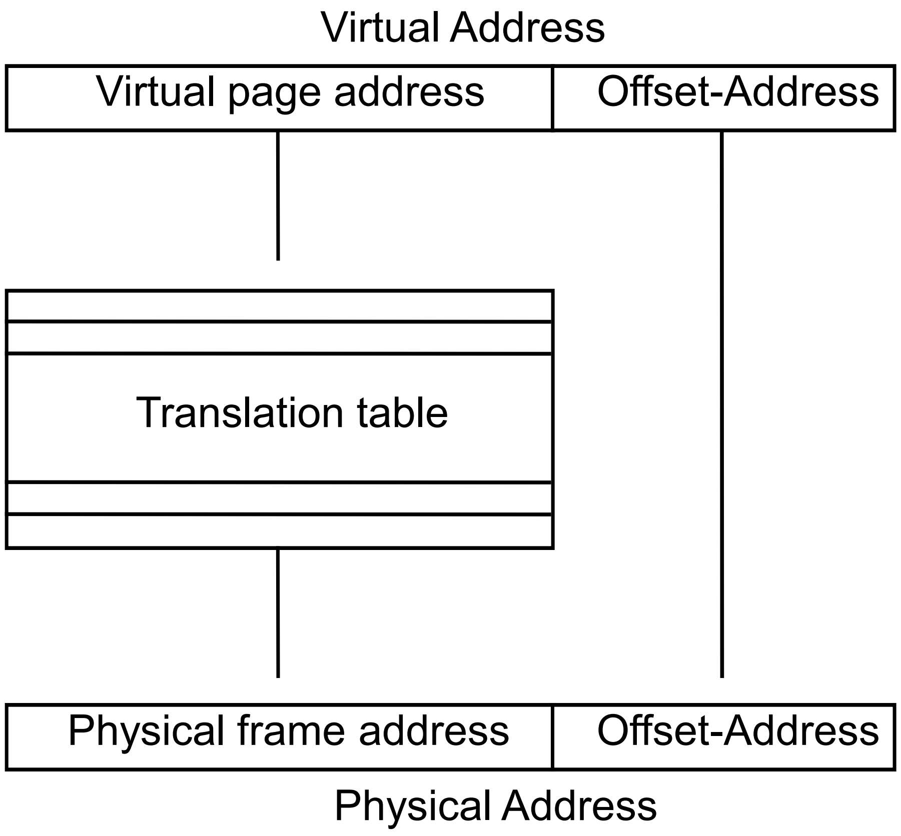
Memory Protection

- OS must protect programs from each other and protect itself from buggy or malicious applications
- Solution: Concept of Address Space
- Simple scheme: HW offers base and limit register
  - Base register indicates begin of application’s memory space
  - Limit register indicates end of application’s memory space
- Limitations:
  - Memory must be allocated as a contiguous block
    - Might waste memory when program finishes
  - Does not allow applications to share memory directly
    - Example: n copies of Mozilla share its program code in RAM

Address Space & Virtual Memory

- OS creates a separate address space AS per application task or per system task
- Assumption: almost each (part of an) AS is relocatible within the physical RAM
  - Sometimes a specific AS or parts of it have to be located at specific physical addresses
- System needs a HW address transformation unit (MMU) to translate efficiently logical “task” addresses into physical RAM addresses

Address Translation

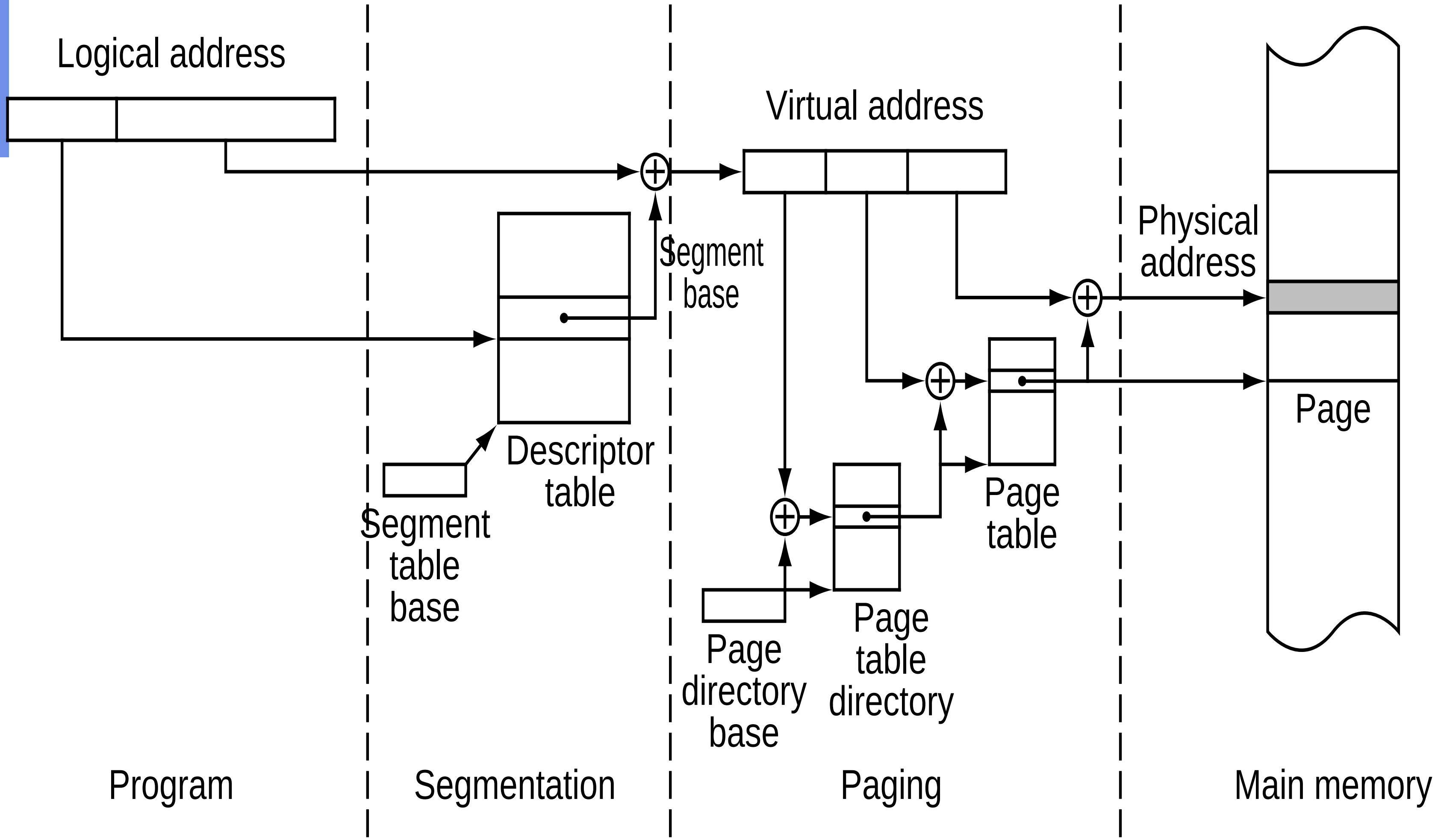


Note:
In most VM systems you need at least one translation table per AS

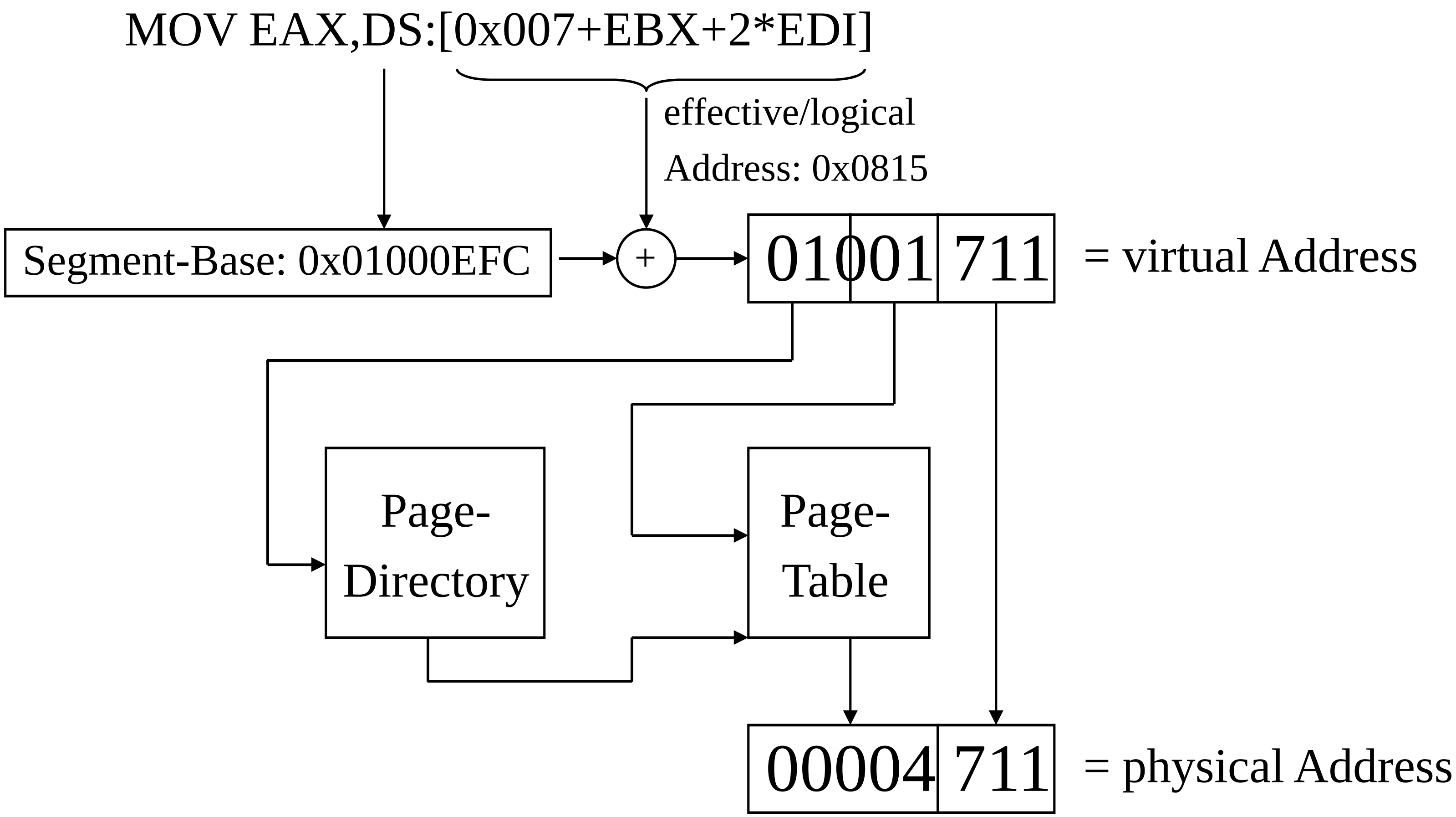
Translation Table(s)

- Translation often done using more than 1 translation table per AS, e.g.
  - Segment- and page table (Intel)
  - Multi-level page tables

Address Translation (Intel 80486)



Address Translation IA 32



Translation Look-Aside Buffer (TLB)

- Parsing translation tables by software is slow
- HW offers a quicker solution, e.g. TLB
  - TLB contain the most recently used pairs of:
    - Virtual page address, physical frame address,
  - TLB implementation via a associative cache
  - TLB sizes contain 32, 64 or even 128 entries
- MMU contains TLB + some other control bit to enable quick access control

Interacting HW Components

How do CPU & Peripherals interact?

- Special (privileged) instructions
- Interrupts

I/O Control

- *How does OS initiate an I/O operation?*
  - Special instructions
    - **`in`** and **`out`** on x86 machines
  - Memory-mapped I/O
    - Access hardware state as memory addresses
    - Requires MMU to translate certain addresses to I/O bus access
  - Usually start I/O by writing a command to a HW register
    - Read block from disk at sector xyz into memory address 0815
- *How does OS realize that I/O has finished?*
  - Polling: read value from the result or status register of the device
  - Interrupt: Hardware signal that causes OS to get control and run the corresponding interrupt handler

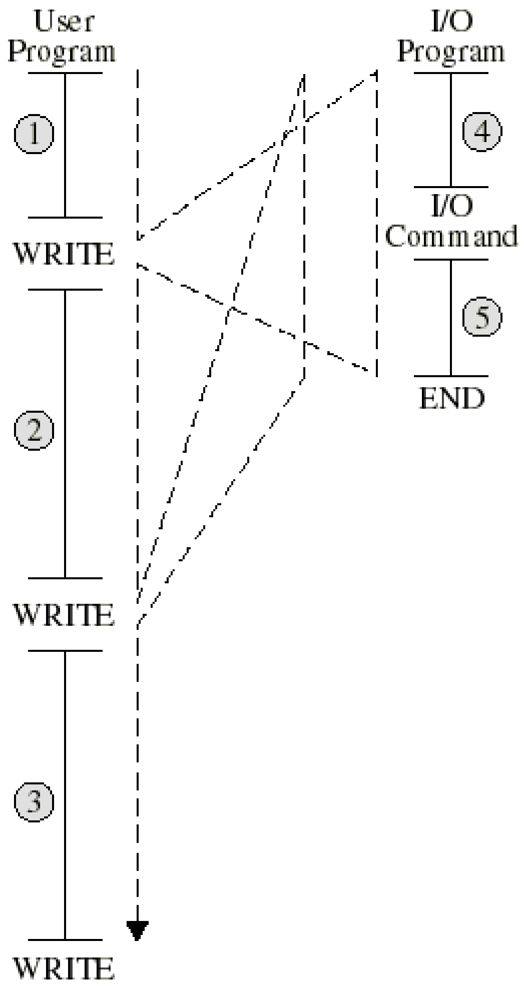
CPU waits for I/O to complete

A WRITE system call transfers control to the printer driver (I/O program).

Printer driver prepares I/O module for printing (4).

CPU has to WAIT for the print-operation to complete.

Printer driver finishes in (5) reporting status of I/O operation.



Exceptions and Interrupts

- 2 major classes of special situations
  - Synchronous exception (e.g. trap)
  - Asynchronous interrupt (e.g. end of DMA action)
- ∃ additional differences concerning
  - Source
  - Predictability
  - Reproducibility
- Handling exceptions or interrupts
  - must be done in time and
  - is processor specific

Exception

Synchronous CPU event

- origin: current instruction
- erroneous exceptions:
  - invalid pointer
  - division by 0

=> typically program is aborted

- non-erroneous exceptions:
  - page fault
  - breakpoint

⇒ typically handled by OS, transparent to user program

Interrupts

Asynchronous event

- origin: I/O device, timer…
- unrelated to instruction stream
- Most interrupts are caused by I/O-completion, etc. (discussed later)
- Failures on devices cause a few interrupts (paper jam, device malfunction, ...)

Trap

- Synchronous, predictable, reproducible
- An isolated program executing on the same CPU with the same input data will "always" trap at the same instruction, e.g.
  - Unknown instruction
  - Buggy instruction (e.g. division by 0)
  - Wrong addressing mode
  - Address (space) violation
  - System call
  - Page fault (in case of local paging policies)

discuss ?

- You can not avoid a trap without having handled the cause of the exception

Interrupt

- Asynchronous, not predictable, not reproducible
- A peripheral device signals an interrupt to the CPU independently of the state of the currently running program on the CPU, e.g.
  - Signaling external events (e.g. a sensor)
  - End of a DMA operation
  - End of an I/O operation (e.g. disk transfer)

Example: *Trap or Interrupt?*

```
#include <stdlib.h>
float frandom () {
  return random()/random();
}
```

∃ chance that the executing process does a division by 0, ⇒ this happens synchronously if it happens, however

it is not predictable when it will happen, but

it is predictable where it will happen

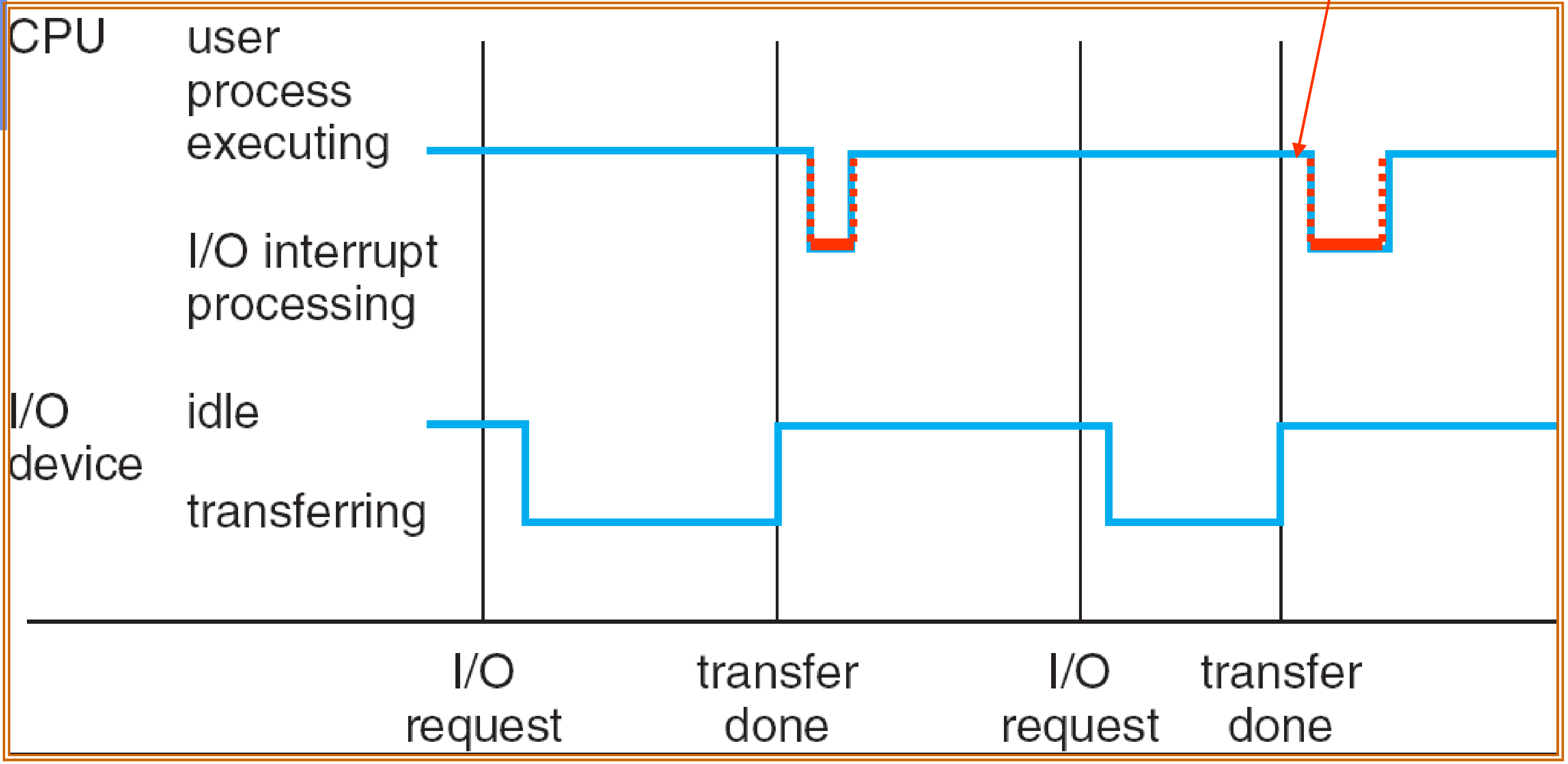
⇒ It is an exception

Exception or Interrupt Handling

- Potentially raising events:
  - Signal from peripheral devices, e.g.
    - End of disk input
    - Timer signal from a watch dog
  - Switching the protection domain (in case of a system call)
  - Programming errors (invalid address)
  - Memory overflow (e.g. stack overflow in case of an endless recursion)
  - Paging on demand (in case of a page fault)
  - Alarm signals from your HW (e.g. shortage of energy)
- The corresponding event handling has to be done during execution of the currently running program

Typical Interrupt Timeline

Delay of interrupt handling



Models of Event Handling

- Resumption model
  - Having handled the event, the interrupted program can resume its execution at the same instruction pointer
- Termination (aborting) model
  - If the cause of a event cannot be handled ⇒ severe error, i.e. the interrupted program has to be terminated
- Raising an exception or an interrupt involves a context switch
  - Currently running program → exception handler or
  - Currently running program → interrupt handler
  - We have to save the "context" of the interrupted program, otherwise no resumption
  - Goal: reduce overhead of saving and restoring contexts

Timing Effects

- Handling of each exception and of each interrupt slows down the execution of the interrupted program
- However, the result of a correct program is not endangered, but
- Real-time applications can fail

Hint:

In any case, each programmer should not rely on any timing conditions

Interrupt Control

- Interrupts happen at any time ⇒ cause problems in the kernel
  - User code is in the middle of modifying state shared with other programs
  - OS is been in the middle of performing a time-sensitive operation
    - E.g. zeroing out a newly allocated buffer of memory to pass to an application
  - OS and hardware must support synchronization of concurrent activities
    - Atomic operations: short sequences of instructions that can not be interrupted
      - e.g. READ-MODIFY-WRITE changes a variable in one atomic step
    - One approach: disable interrupts during atomic operations
      - What are the problems with this approach?
- Another approach: complex atomic instructions
  - Test-and-set
    - Read value into register and store 1 into
  - Load linked (LL) and store conditional (SC):
    - LL: load memory value into register
    - SC: only perform store if the value at the address had not changed since LL

Interrupts

- Many computers permit I/O modules to interrupt an activity on the CPU
- For this an I/O module just asserts an interrupt request line on the “control bus”
- Then CPU transfers control to an “Interrupt Handler” (normally part of the OS-kernel)
- CPU can prevent to be interrupted, by masking out or disabling interrupts

Interrupt Handler

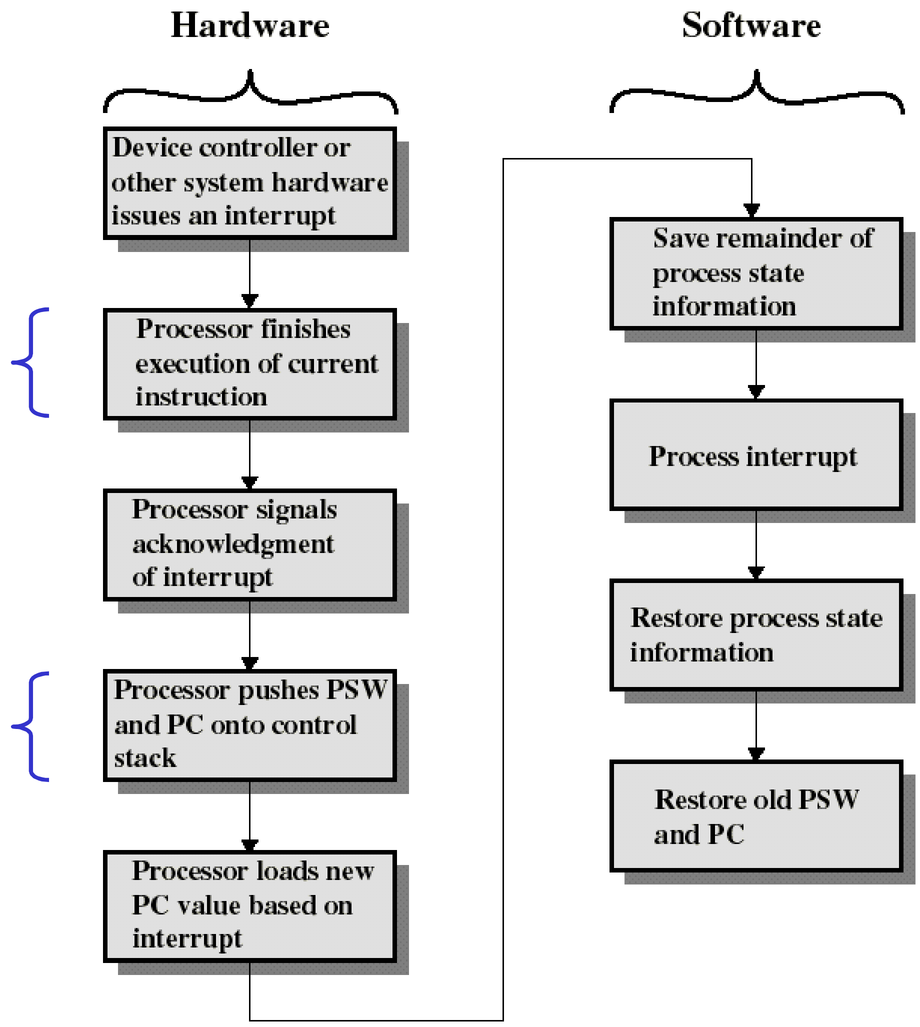
- A (peripheral) interrupt interrupts the currently executing program invoking the corresponding interrupt handler
- The interrupt handler might return* to the interrupted program ⇒
- Point of interruption can occur anywhere, so either HW or interrupt handler must save the state of the interrupted program (IP, PSW, registers ...) and restore it upon return

* *What program may run instead of?*

Interrupt Processing

Long instructions can be interrupted

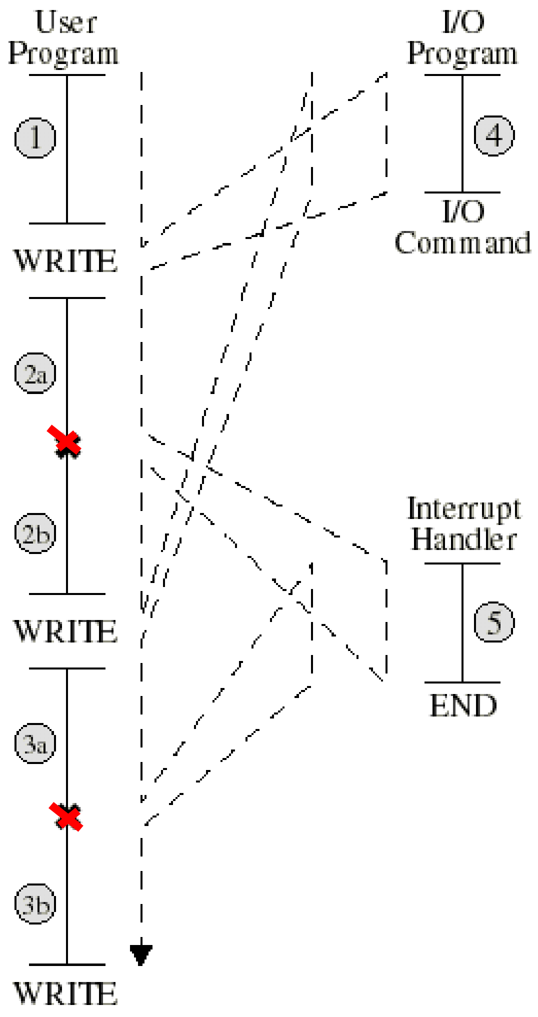
Some CPUs save context into shadow registers


Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

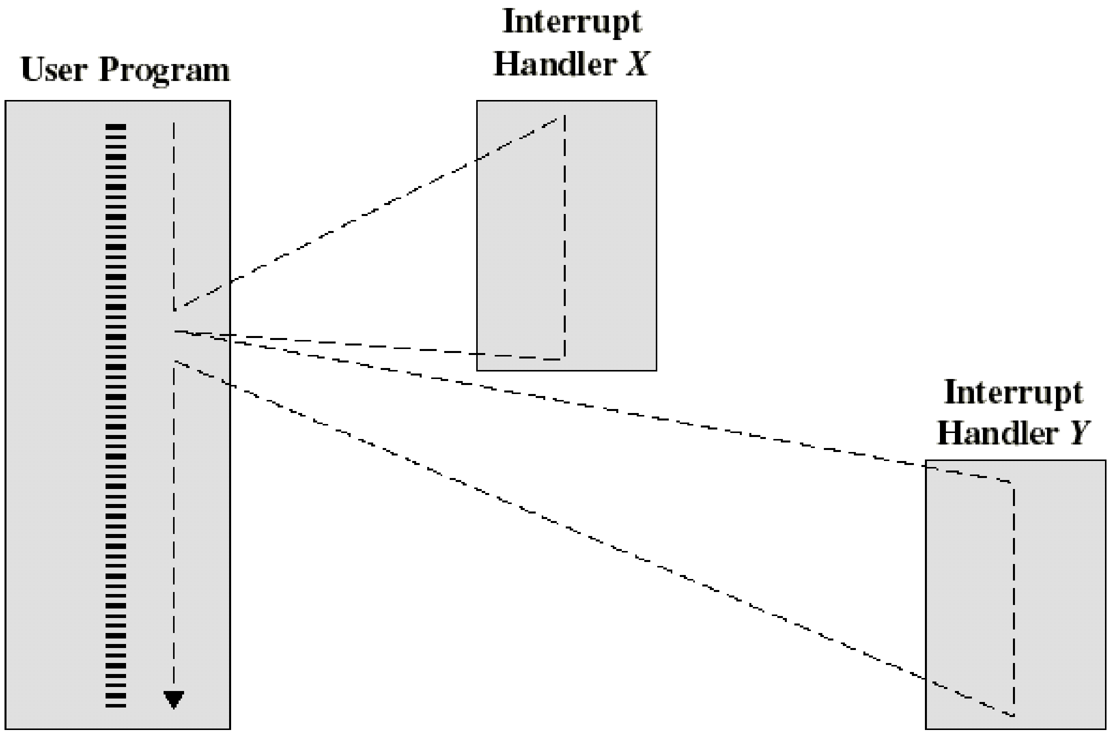
Interrupts improving CPU usage

I/O program prepares I/O module, issues I/O command (to a printer) I/O program branches back to the user program, user code gets executed during I/O operation (e.g. printing) ⇒ no waiting

User program gets interrupted (x) when I/O operation is done and branches to interrupt handler to examine status of I/O module Execution of user code resumes



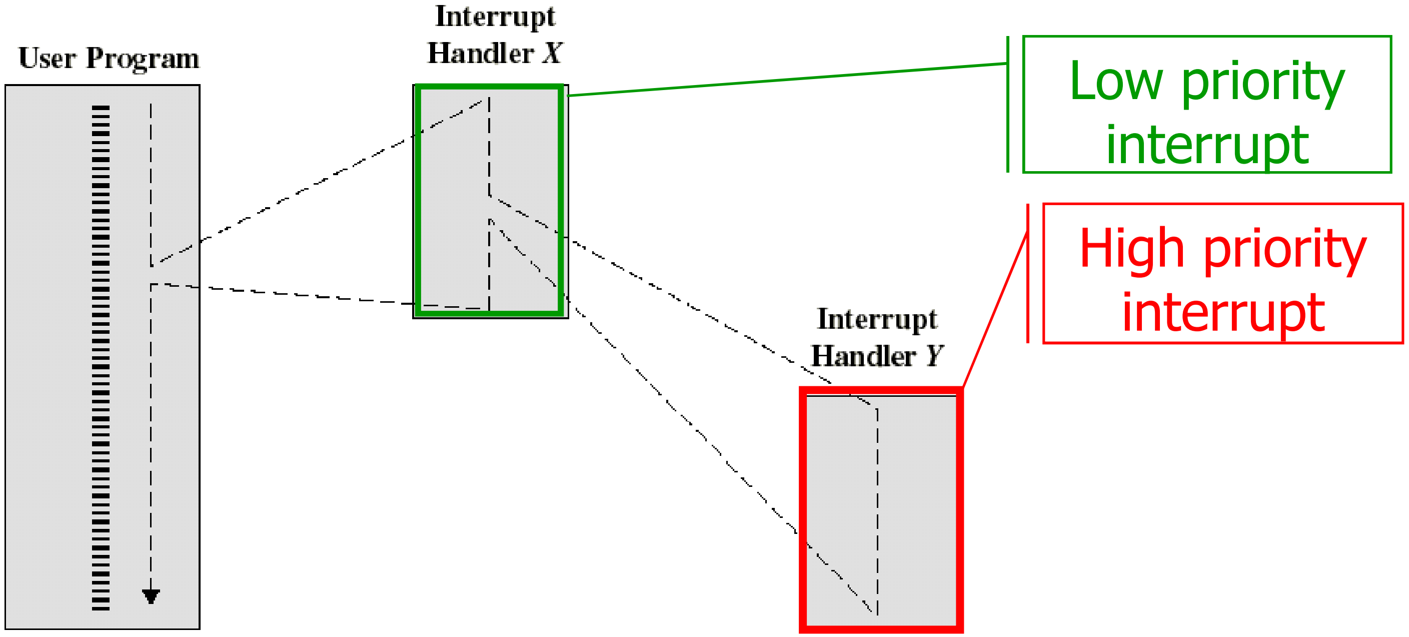
N>1 Interrupts: Sequential Order



Analysis:

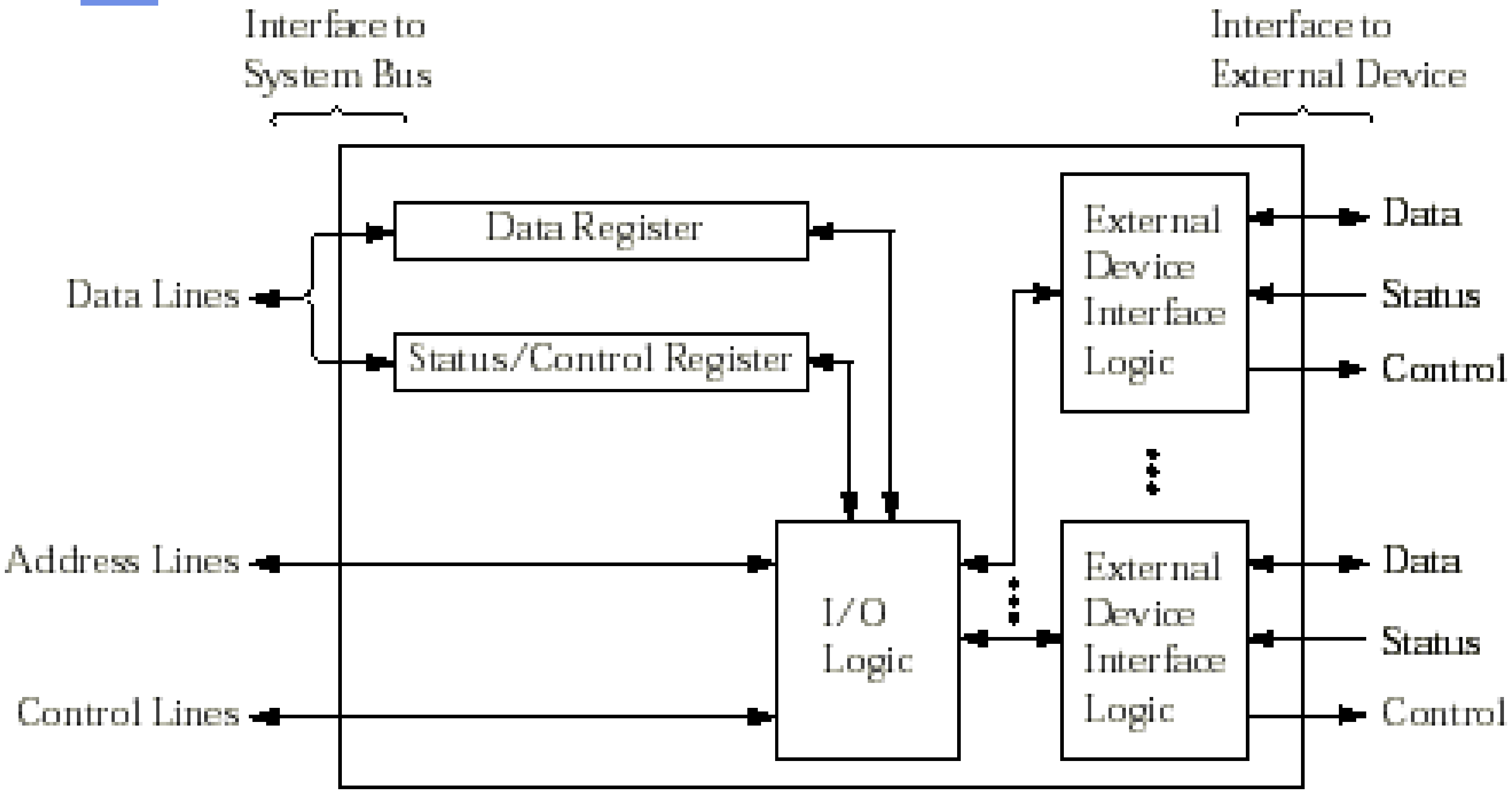
Disable all interrupts during interrupt processing. Interrupts remain pending until CPU enables interrupts again. After interrupt handler completed, CPU checks for further pending interrupts

N>1 Interrupts: Nested Interrupts



- Low-priority interrupt handling no longer delays high-priority interrupt processing. High-priority interrupt cause a low-priority interrupt handler to be interrupted.
- When input-data arrive from the network, it needs to be consumed quickly to make room for further incoming data.

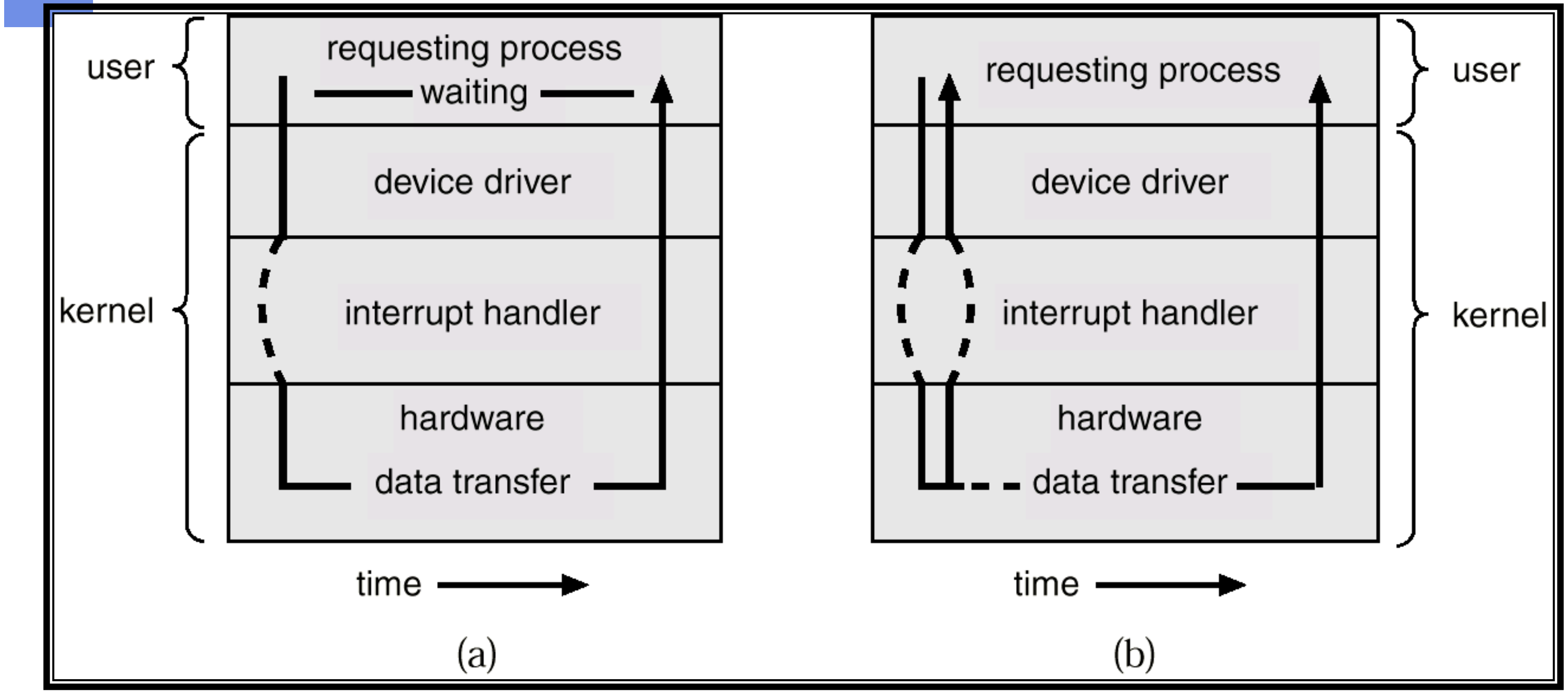
I/O Module Structure



Two Principal I/O Methods

Synchronous

Asynchronous



I/O-Interface Techniques

- Programmed I/O (polling)
- Interrupt Driven I/O
- Direct Memory Access (DMA)

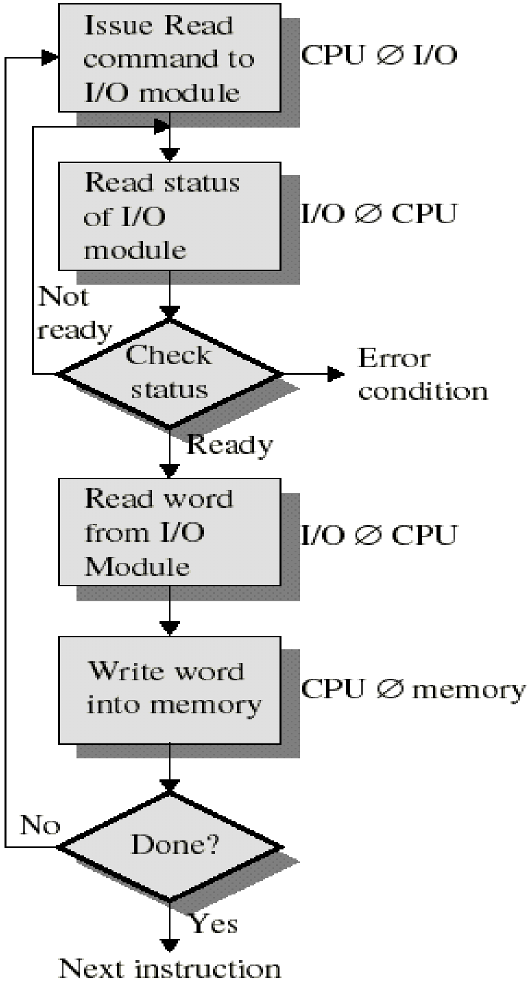
Discuss these techniques in detail in the tutorials!

Programmed I/O

I/O module performs action, on behalf of CPU

I/O module does not have to interrupt CPU when I/O is done

CPU is kept busy checking status of I/O module



Issue Read command to I/O module

CPU ∅ I/O

Read status of I/O module

I/O ∅ CPU

Not ready

Check status

Error condition

Ready

Read word from I/O Module

I/O ∅ CPU

Write word into memory

CPU ∅ memory

No

Done?

Yes

Next instruction

(a) Programmed I/O

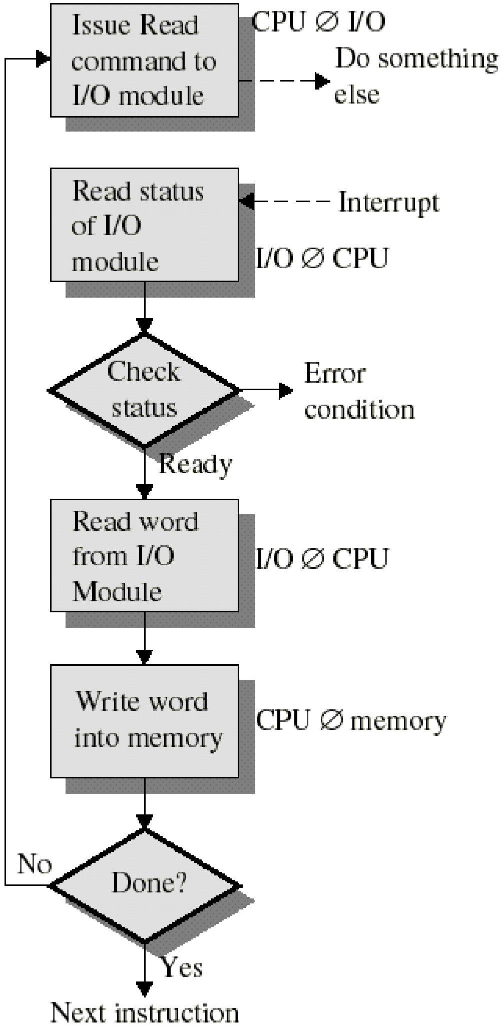
Interrupt-Driven I/O

CPU is interrupted when
I/O module is ready to transfer data,
e.g. if requested operation finished.

CPU is free to do other work in
the meantime ⇒ no busy waiting.
Well-suited for medium-grained event.

Bad for frequent fine-grain events:
⇒ high CPU costs per interrupt.

- Interrupt per incoming network packet might work
- Interrupt per byte is far too expensive.

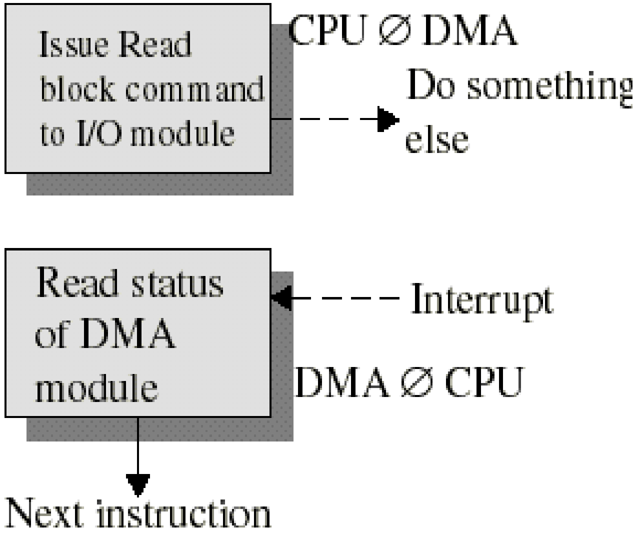


Direct Memory Access

CPU issues request to a DMA module (separate module or incorporated into I/O module)
DMA module transfer a block of data directly to/from memory, *not* through CPU)
Interrupt is sent when DMA is complete.

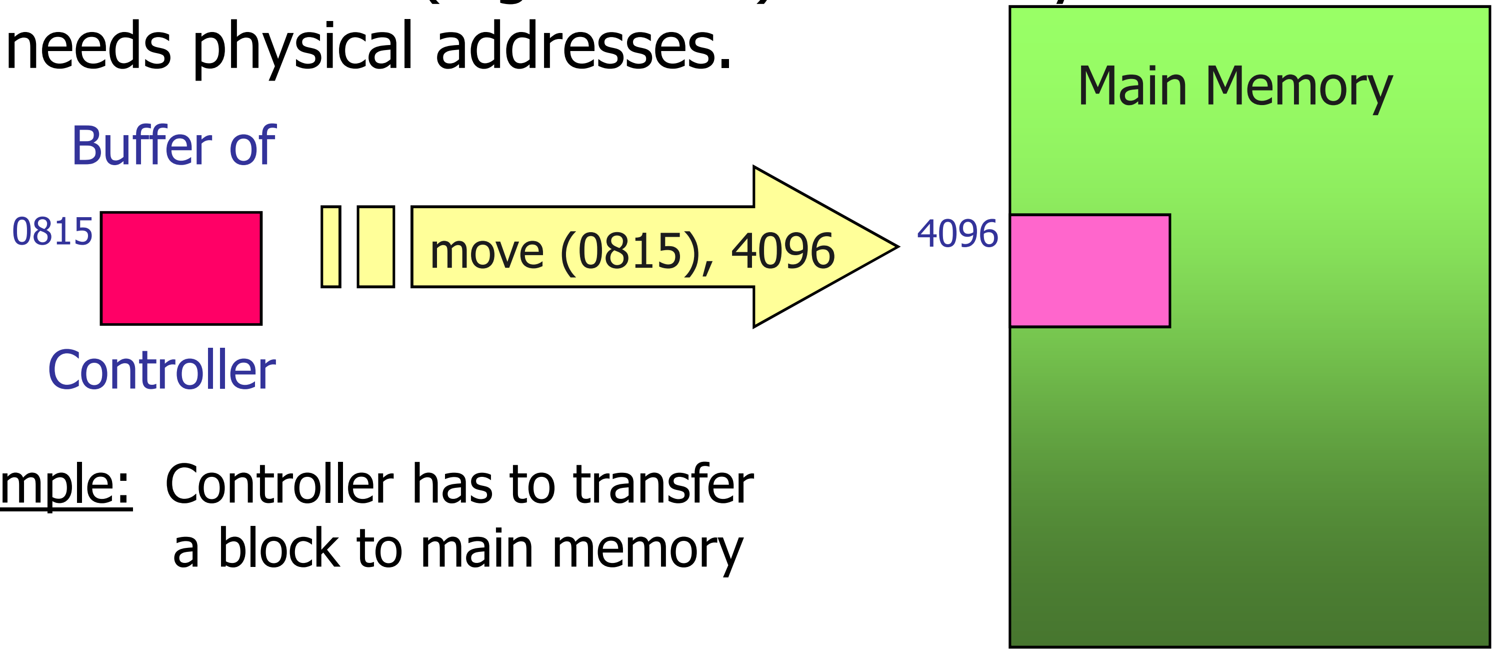
CPU is only involved at the beginning and at the end of the transfer, it is free to perform other jobs during data transfer.

<u>However note:</u>
DMA may put heavy load on memory bus
⇒ problem of cycle stealing


Issue Read block command to I/O module
CPU ∅ DMA
Do something else
Read status of DMA module
Interrupt
DMA ∅ CPU
Next instruction

Physical Addressing of RAM

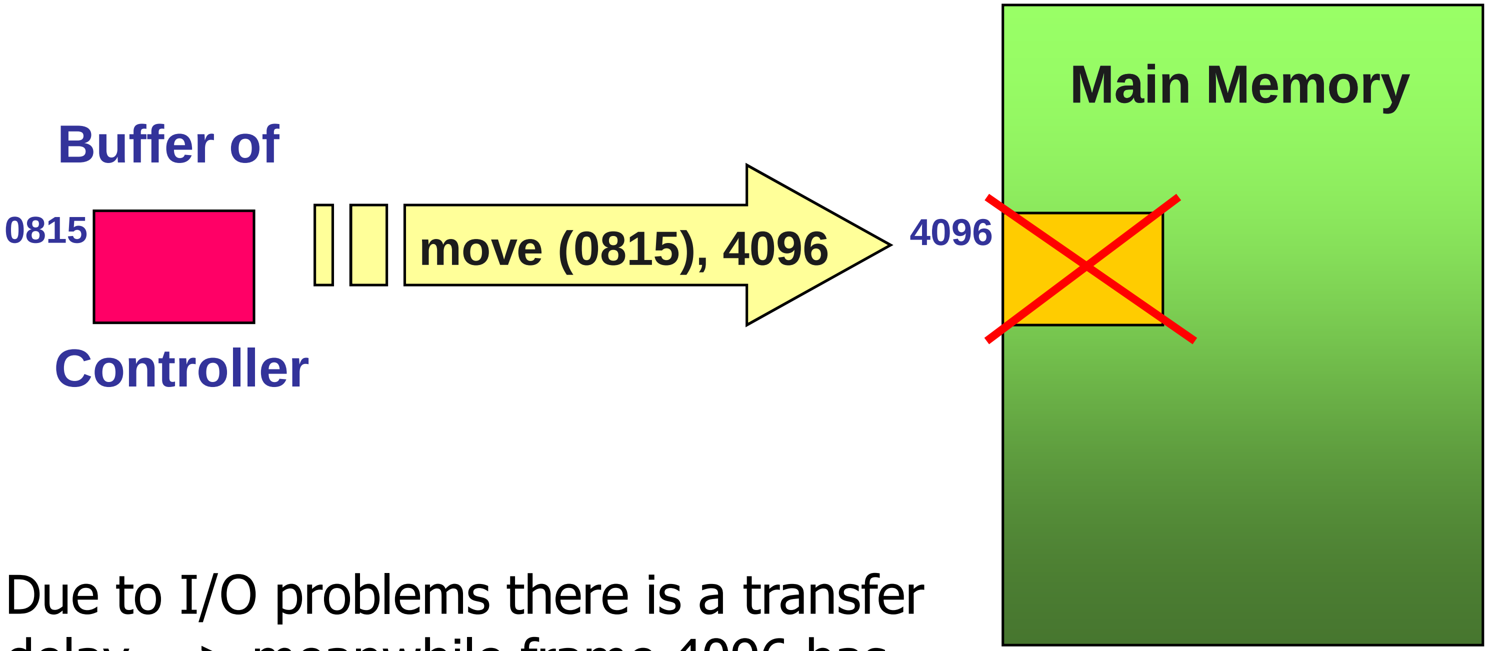
To access (either read or write) the desired buffer a device controller (e.g. a DMA) often only knows and needs physical addresses.



Example: Controller has to transfer a block to main memory

What may happen?

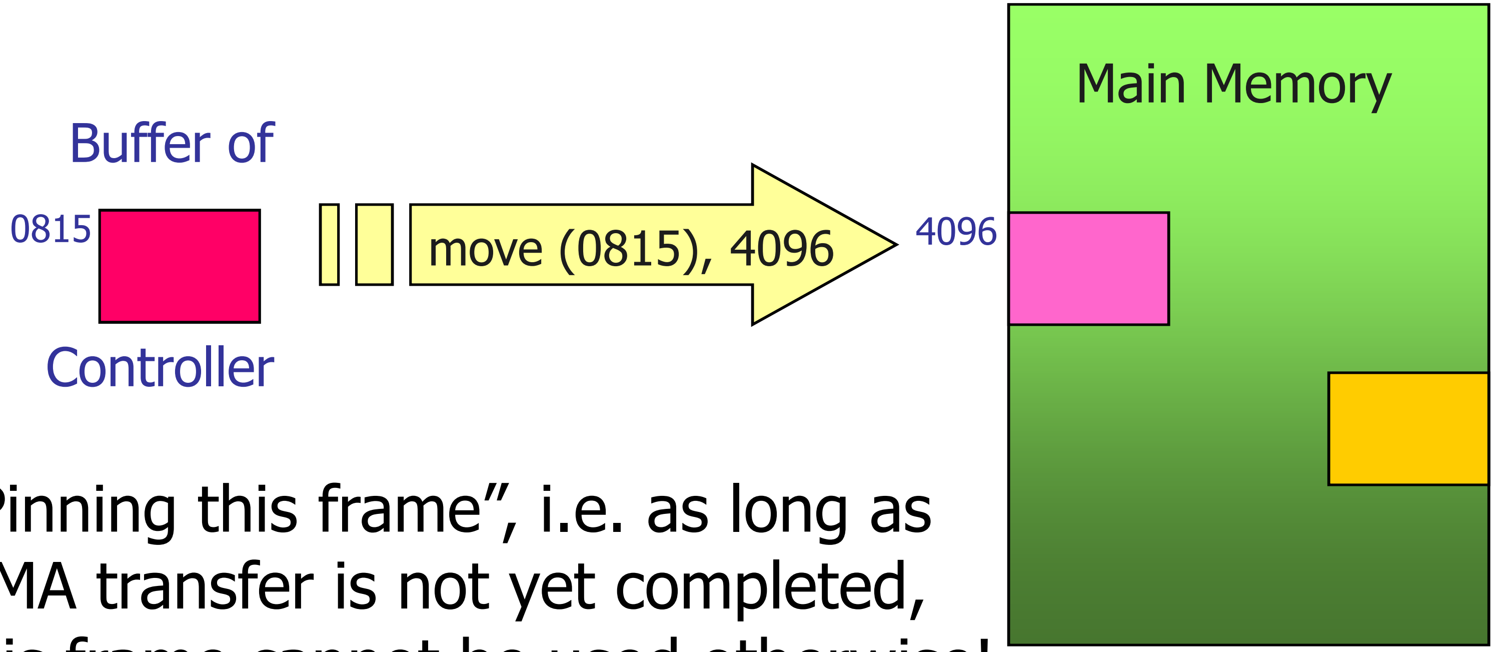
Physical Addressing of RAM



Due to I/O problems there is a transfer delay, => meanwhile frame 4096 has been used for some other activity because of paging or segmentation requirements.

How to solve this problem?

Physical Addressing of RAM



“Pinning this frame”, i.e. as long as DMA transfer is not yet completed, this frame cannot be used otherwise!

Pinning and unpinning are the required mechanism.

What policy can you establish upon it?

Application of Pinning

- To support DMA
- To support real-time tasks (see*)
  - *Can that method be abused?*
  - Think about a solution

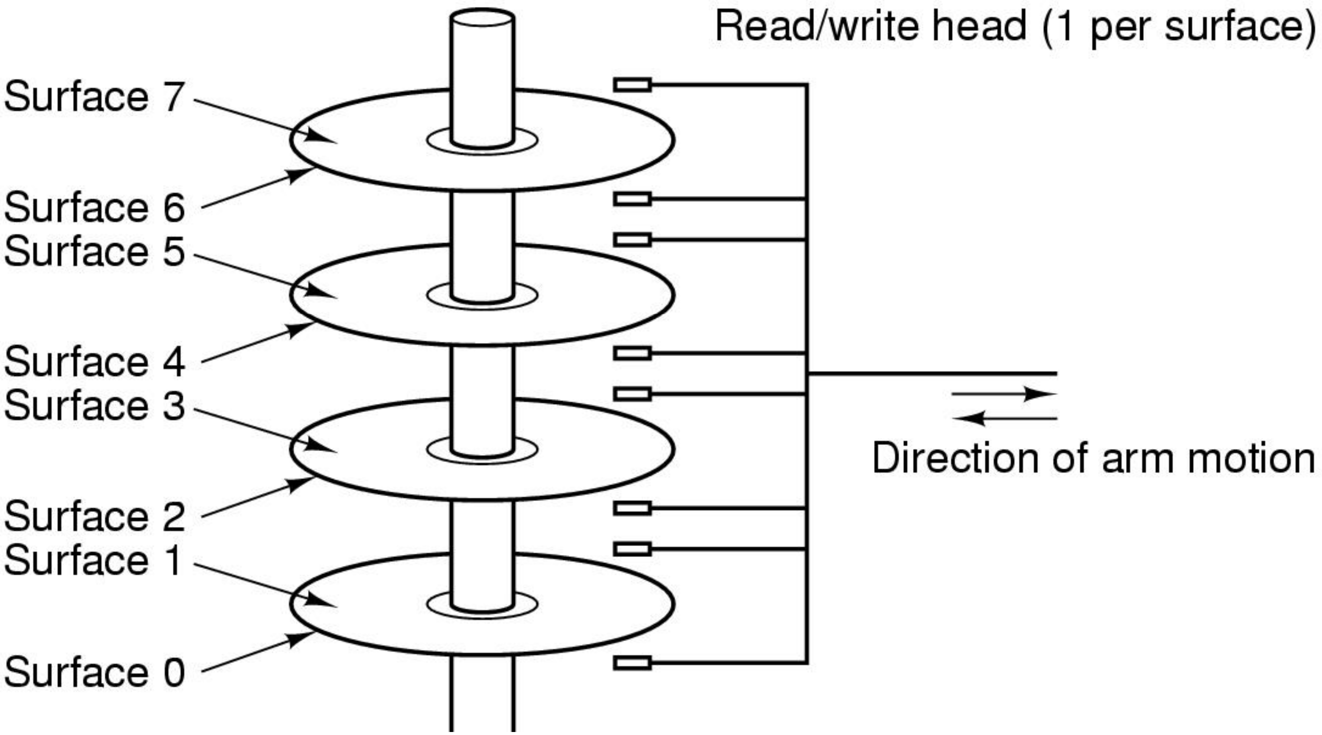
*J. Liedtke, V. Uhlig, K. Elphinstone, T. Jaeger, and Y. Park:

"How to Schedule Unlimited Memory Pinning of Untrusted Processes or Provisional Ideas About Service-Neutrality"

7th Workshop on Hot Topics in Operating Systems (HotOS),
Rio Rico, MA, March 1999

I/O Devices

HW Review (Disk)



Structure of a disk drive

Summary I/O Management

- Early computers had a static hardware configuration
  - All I/O devices known at boot time
  - Kernel was configured statically by the system administrator for the particular machine
- Today things are more complicated
  - There are hotplug devices
  - USB, Compact Flash, Firewire, etc.
- Even the PCI bus is configured on each boot cycle
  - Assign I/O addresses, interrupt lines to each device at boot time
  - Allows users to plug in new boards without reconfiguring the kernel
- OS must discover available devices (even on the fly)
  - USB controller interrupts OS when a device is added or removed
  - Device identifies itself with a special identifier
    - Vendor/device ID, device type, etc.
  - Kernel loads appropriate driver
    - Loadable kernel modules (Linux)
  - User’s access to the device is enabled
    - Remap /dev/mouse to the new USB mouse device