THE MEMORY SYSTEM

1. Components of the Memory System

2. The Memory Hierarchy

3. Cache Memories

4. Cache Organization

5. Replacement Algorithms

6. Write Strategies

7. Virtual Memory

Components of the Memory System

- Main memory: fast, random access, expensive, located close to the CPU.
  Is used to store program and data which are *currently manipulated* by the CPU.

- Secondary memory: slow, cheap, direct access, located remotely from the CPU.

Problems with the Memory System

What do we need?

We need memory to fit very large programs and to work at a speed comparable to that of the microprocessors.

Main problem:

- ❒ microprocessors are working at a very high clock rate and they need large memories;
- ❒ memories are much slower than microprocessors;

Facts:

- ❒ the larger a memory, the slower it is;
- ❒ the faster the memory, the greater the cost/bit.

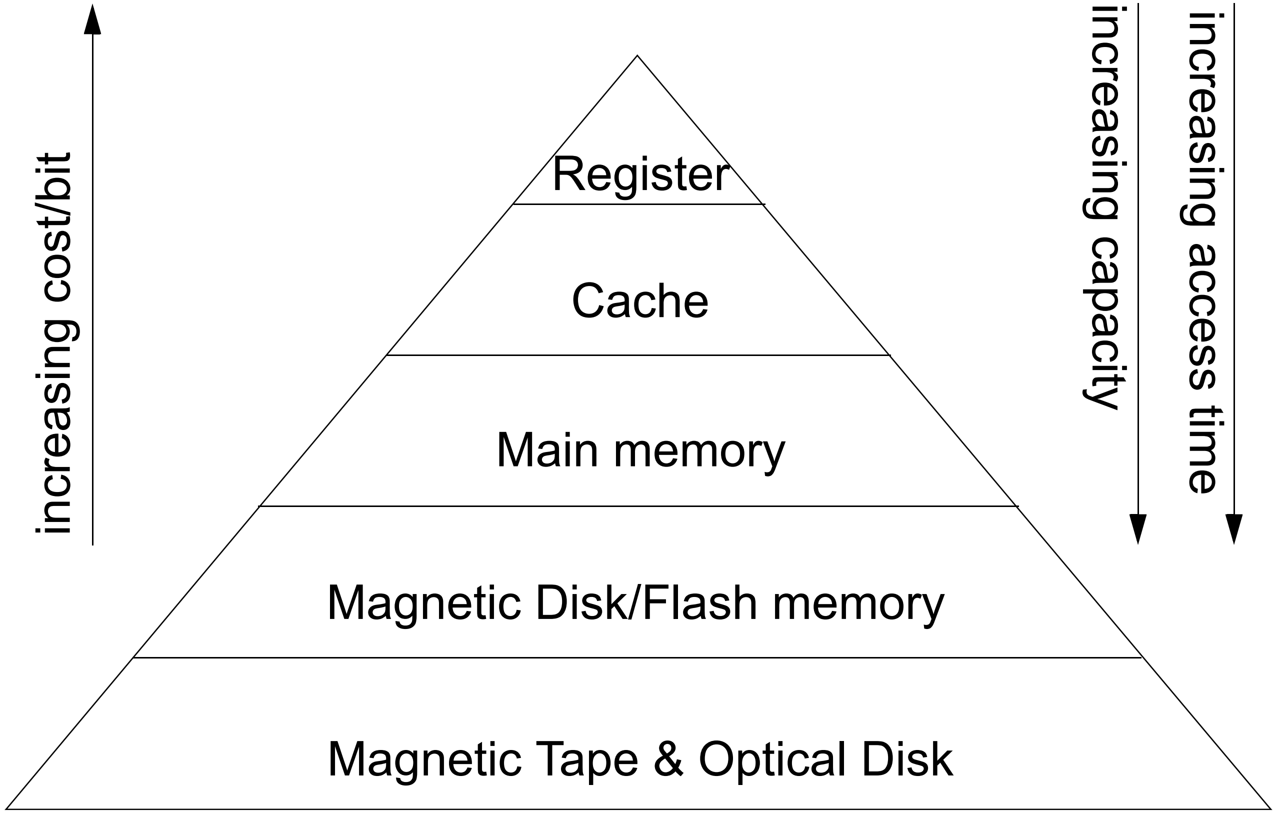
A Solution

It is possible to build a composite memory system which combines a *small, fast memory* and a *large slow main memory* and which behaves (most of the time) like a large fast memory.

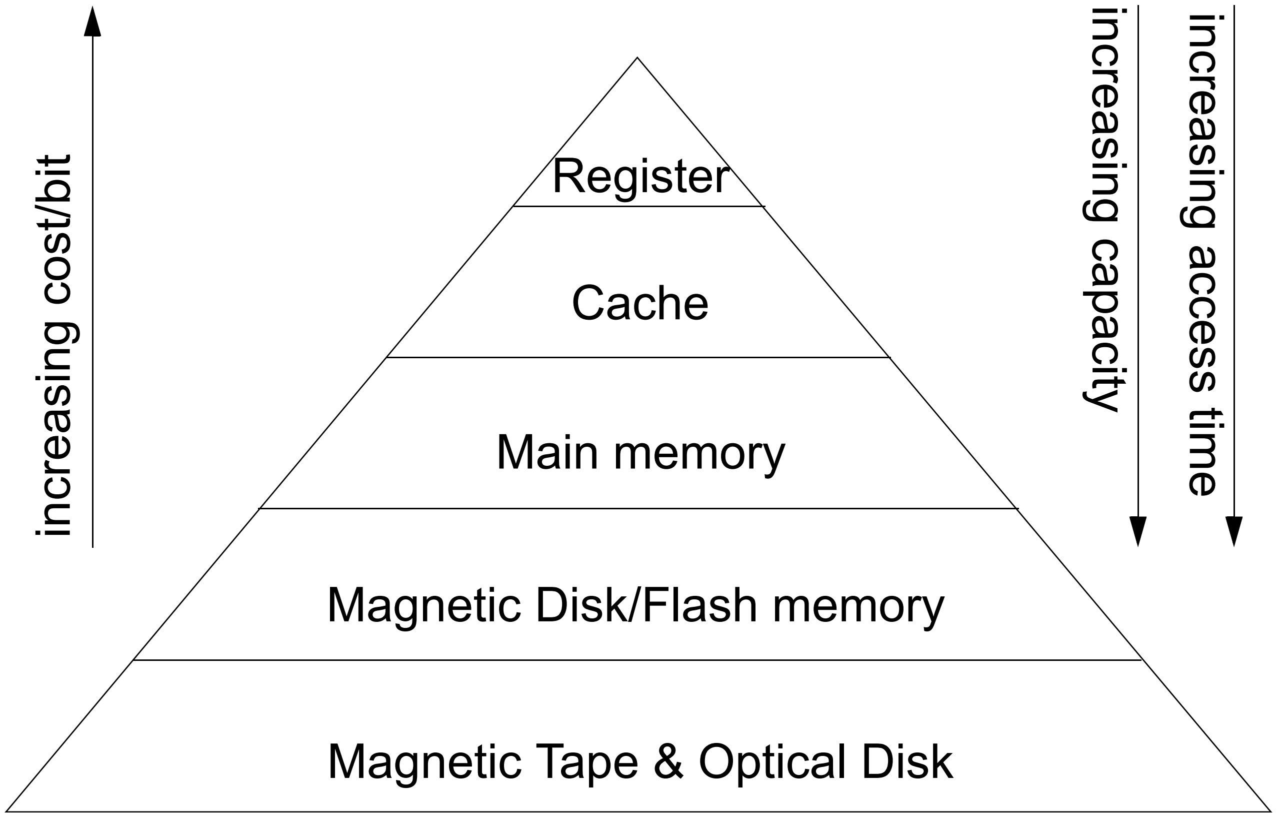
The two level principle above can be extended into a *hierarchy of many levels* including the secondary memory (disk store).

The effectiveness of such a memory hierarchy is based on property of programs called the *principle of locality*.

The Memory Hierarchy



The Memory Hierarchy



The key to the success of a memory hierarchy is if data and instructions can be distributed across the memory so that most of the time they are available, when needed, on the top levels of the hierarchy.

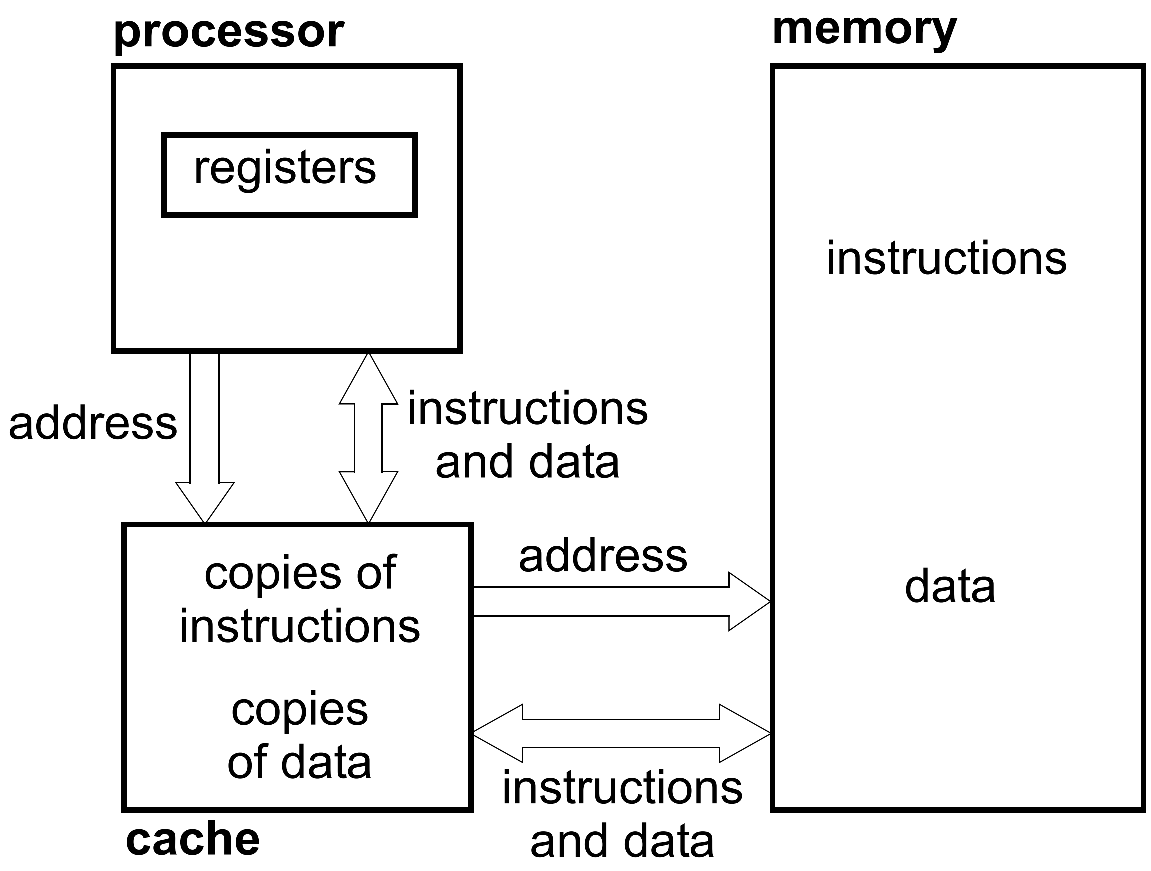
The Memory Hierarchy

- The data which is held in the registers is under the direct control of the compiler (or of the assembler programmer).

- The contents of the other levels of the hierarchy are managed automatically:
  - migration of data/instructions to and from caches is performed under hardware control;
  - migration between main memory and backup store is controlled by the operating system (with hardware support).

Cache Memory

A cache memory is a small, very fast memory that retains copies of recently used information from main memory. It operates transparently to the programmer, automatically deciding which values to keep and which to overwrite.



Cache Memory

- The processor operates at its high clock rate only when the memory items it requires are held in the cache.

  ⇓

  The overall system performance depends strongly on the proportion of the memory accesses which can be satisfied by the cache

- An access to an item which is in the cache: *hit*

  An access to an item which is not in the cache: *miss*.

  Proportion of memory accesses that are satisfied by the cache: *hit ratio*

  Proportion of memory accesses that are not satisfied by the cache: *miss ratio*

- The *miss ratio* of a well-designed cache: few %

Cache Memory

What we want by using caches is to obtain *high memory performance;* this means short access time:

<u>Average access time T_s</u> :

$$T_s = H \times T_1 + (1 - H) \times (T_1 + T_2) = T_1 + (1 - H) \times T_2$$

- T_1 : Access time to cache
- T_2 : Access time to main memory
- H : Hit ratio

⬇

Increased hit ratio will reduce average access time; with high hit ratio (close to 1), average memory access time converges towards the cache access time.

Cache Memory

- Cache space (~KBytes) is much smaller than main memory (~GBytes);

↓

Items have to be placed in the cache so that they are available there when (and possibly only when) they are needed.

Cache Memory

- Cache space (~KBytes) is much smaller than main memory (~GBytes);

⬇

Items have to be placed in the cache so that they are available there when (and possibly only when) they are needed.

How can this work???

Cache Memory

- Cache space (~KBytes) is much smaller than main memory (~GBytes);

  ⬇

  Items have to be placed in the cache so that they are available there when (and possibly only when) they are needed.

- The answer is: *locality*

  During execution of a program, memory references by the processor, for both instructions and data, tend to cluster: once an area of the program is entered, there are repeated references to a small set of instructions and data.

  - ❐ Temporal locality (locality in time): If an item is referenced, it will tend to be referenced again soon (instructions inside a loop).
  - ❐ Spacial locality (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon (a sequence of instructions under execution; elements of an array during sorting).

Cache Memory

Problems concerning cache memories:

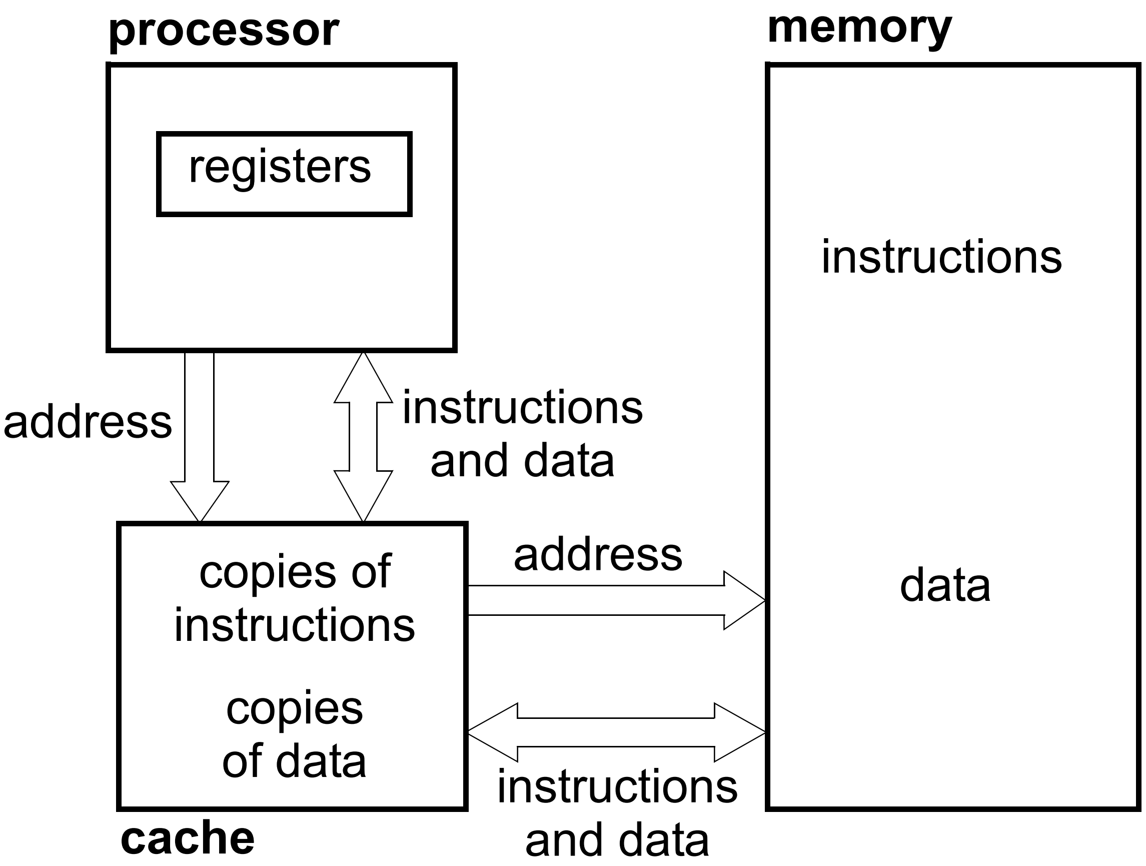
❒ Instruction and data in the same cache or not?

❒ How to determine at a *read* if we have a miss or hit?

❒ If there is a miss where to place the new item in the cache? Which information should be replaced?

❒ How to preserve consistency between cache and main memory at *write*?

Unified Instruction and Data Cache



processor

registers

address

instructions and data

memory

instructions

data

copies of instructions

copies of data

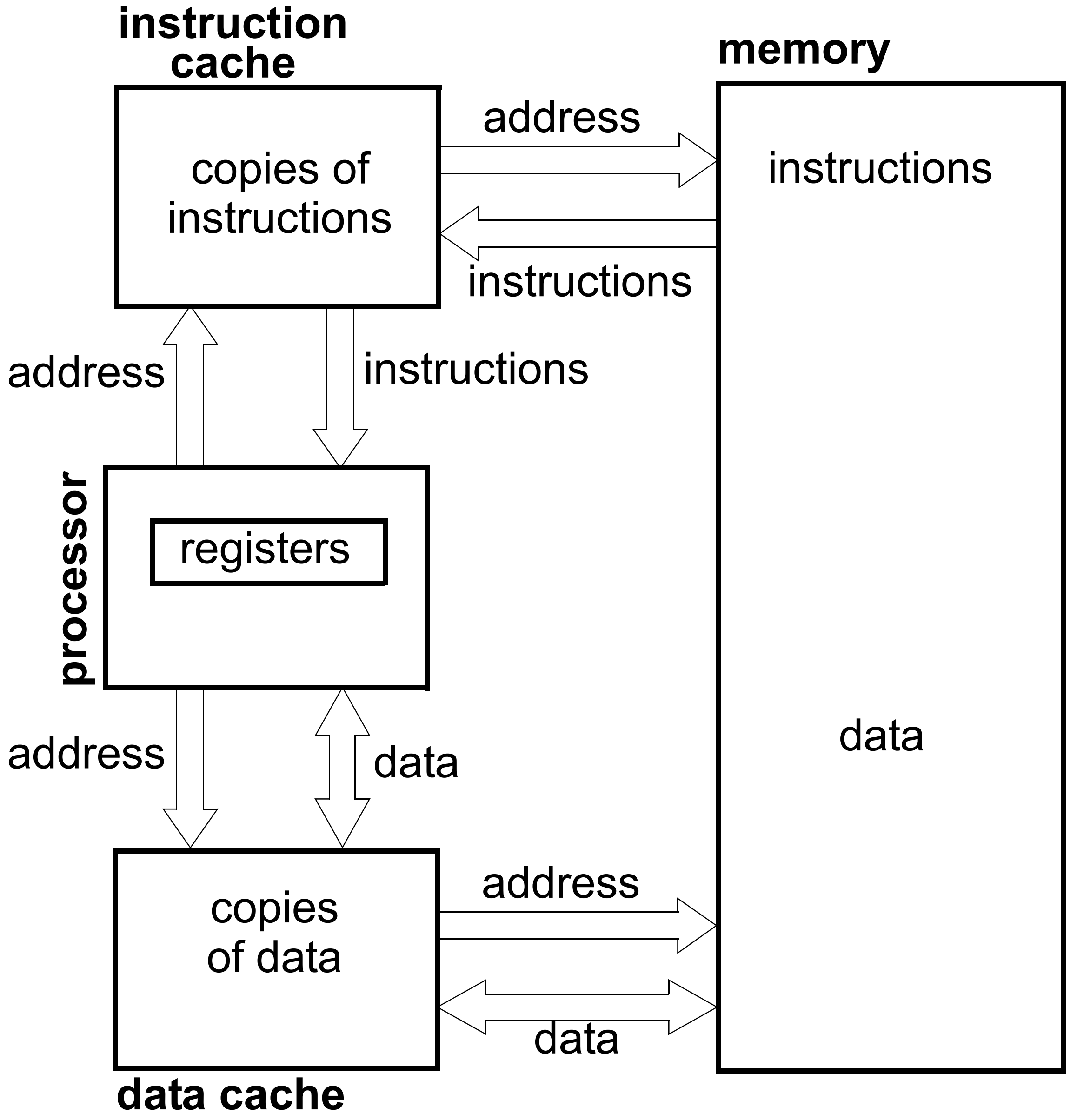
cache

address

instructions and data

Separate Data and Instruction Caches

It is common to split the cache into one dedicated to instructions and one dedicated to data.



Separate Data and Instruction Caches

- Advantages of unified caches:
  - they are able to better balance the load between instruction and data fetches depending on the dynamics of the program execution;
  - design and implementation are cheaper.

- Advantages of split caches (Harvard Architectures):
  - competition for the cache between instruction processing and execution units is eliminated $\Rightarrow$ instruction fetch can proceed in parallel with memory access from the execution unit.

Cache Memory

Problems concerning cache memories:

- ❐ Instruction and data in the same cache or not?
- ❐ How to determine at a *read* if we have a miss or hit?
- ❐ If there is a miss where to place the new item in the cache? Which information should be replaced?
- ❐ How to preserve consistency between cache and main memory at *write*?

Cache Organization

Example:

- a cache of 64 Kbytes
- data transfer between cache and main memory is in *blocks* of 4 bytes; we say the cache is organized in *lines* of 4 bytes;
- a main memory of 16 Mbytes; each byte is addressable by a 24-bit address (2^{24}=16M)

⬇

- the cache consists of 2^{14} (16K) lines
- the main memory consists of 2^{22} (4M) blocks

Cache Organization

Example:

- a cache of 64 Kbytes
- data transfer between cache and main memory is in *blocks* of 4 bytes; we say the cache is organized in *lines* of 4 bytes;
- a main memory of 16 Mbytes; each byte is addressable by a 24-bit address (2^{24}=16M)

⇓

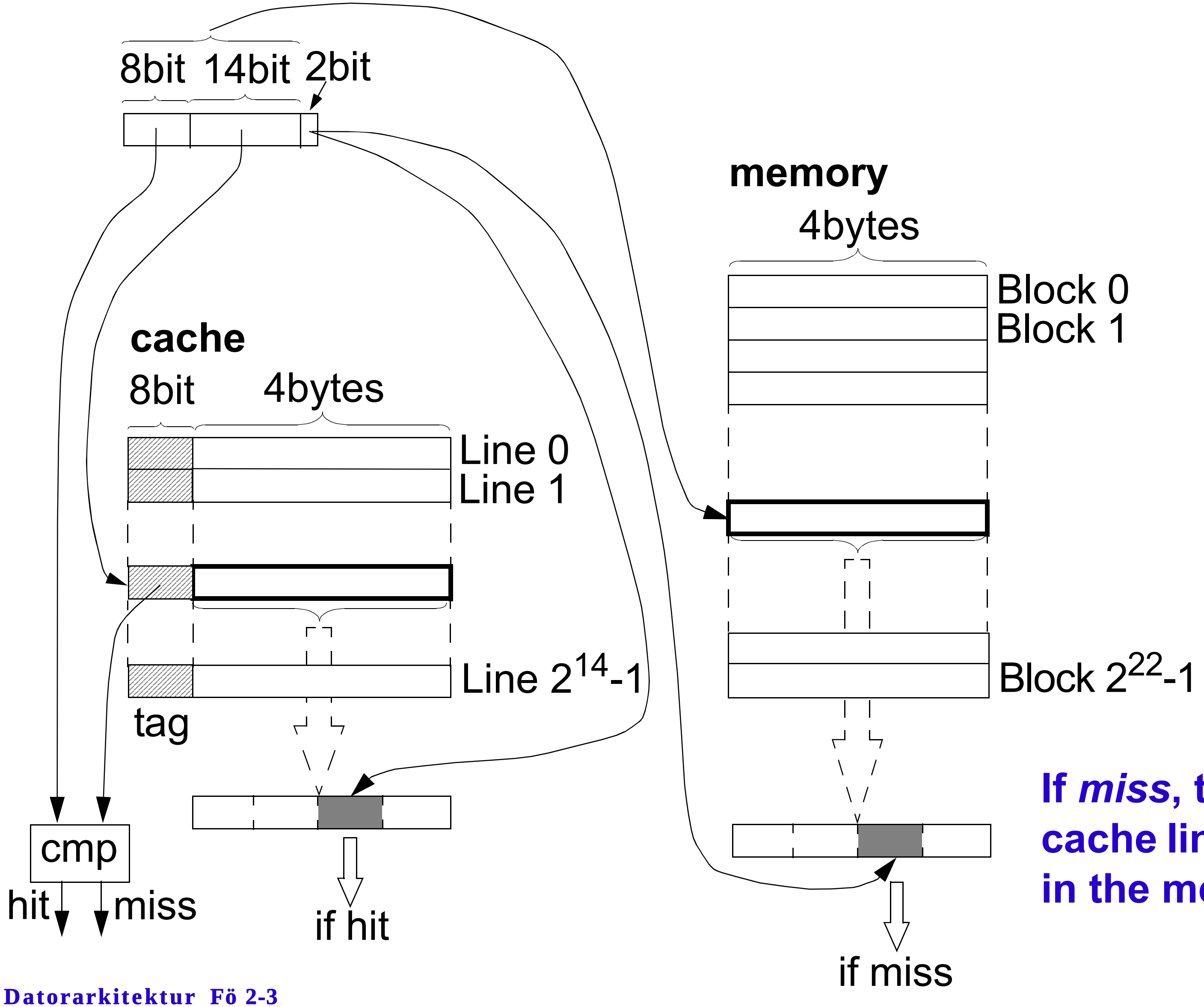
- the cache consists of 2^{14} (16K) lines
- the main memory consists of 2^{22} (4M) blocks

Questions:

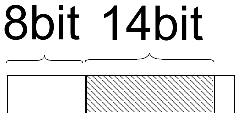
- when we bring a block from main memory into the cache where (in which line) do we put it?
- when we look for the content of a certain memory address
  - in which cache line do we look for it?
  - how do we know if we have found the right information (*hit*) or not (*miss*)?

Direct Mapping

- 14 bits are needed to identify the cache line
- 22 bits are needed to address a block in main memory
- *tag* size is 22 - 14 = 8 bits



If *miss*, the block is placed in the cache line corresponding to the 14 bits in the memory address of the block:



Direct Mapping

- A memory block is mapped into *a unique* cache line, depending on the memory address of the respective block.

- A memory address is considered to be composed of three fields:

    1. the least significant bits (2 in example) identify the byte within the block;
    2. the rest of the address (22 bits in example) identify the block in main memory; for the cache logic, this part is interpreted as two fields:

        2a. the least significant bits (14 in example) specify the cache line;

        2b. the most significant bits (8 in example) represent the *tag*, which is stored in the cache together with the line.

- Tags are stored in the cache in order to distinguish among blocks which fit into the same cache line.

Direct Mapping

Advantages:

- simple and cheap;
- the tag field is short; only those bits have to be stored which are not used to address the cache (compare with the following approaches);
- access is very fast.

Disadvantage:

- A given block is assigned into a fixed cache location ⇒ a given cache line will be replaced whenever there is a reference to another memory block which fits to *the same* line, regardless what the status of the other cache lines is.

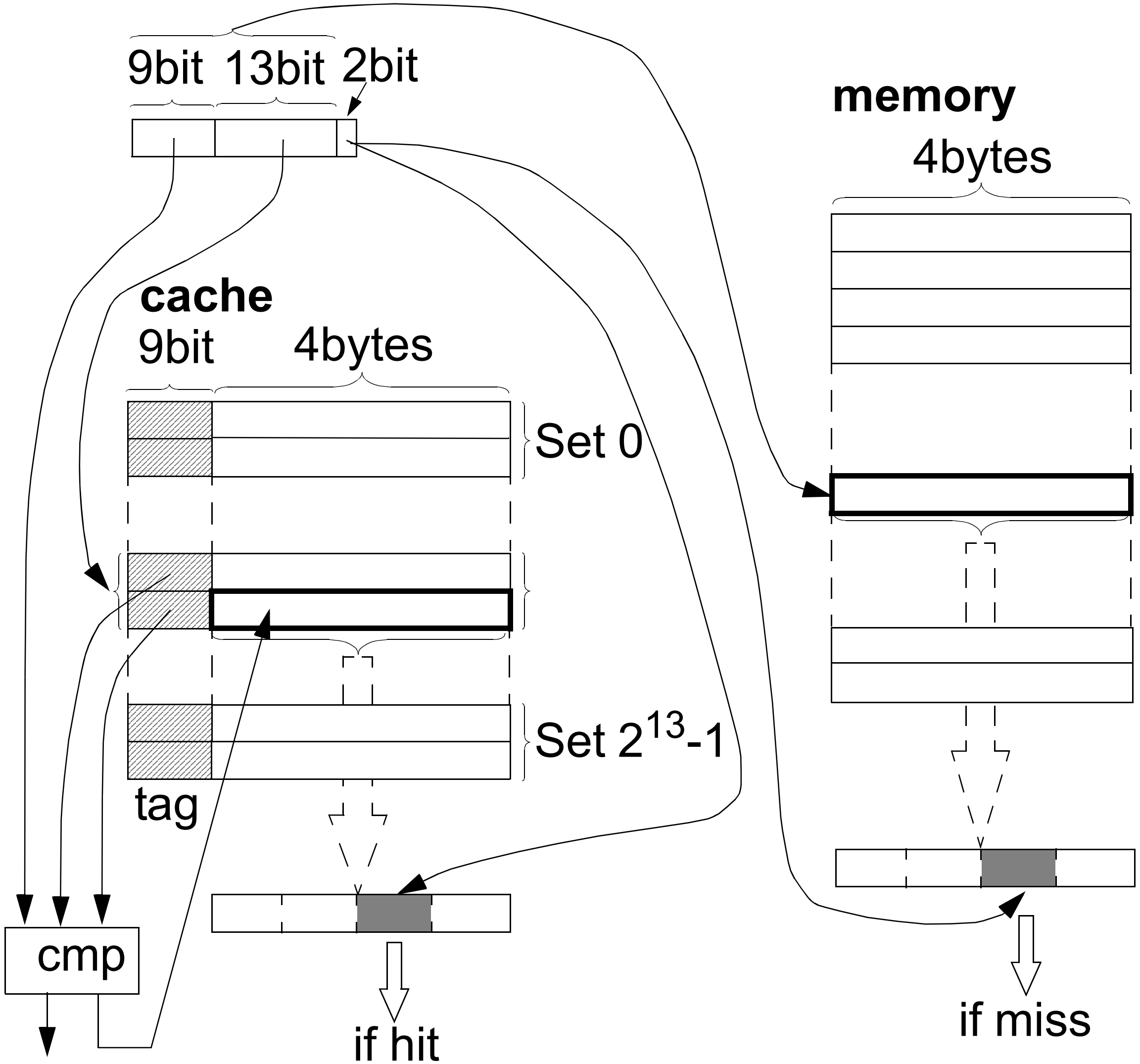
⇓

This can produce a low hit ratio, even if only a very small part of the cache is actually used.

Set Associative Mapping

Two-way set associative cache

- 13 bits are needed to identify the cache set
- 22 bits are needed to address a block in main memory
- *tag* size is 22 - 13 = 9 bits



If *miss*, the block is placed in one of the two cache lines in the set corresponding to the 13 bits field in the memory address. The *replacement algorithm* decides which line to use.

Set Associative Mapping

- A memory block is mapped into *any of the lines* of a set. The set is determined by the memory address, but the line inside the set can be any one.

- If a block has to be placed in the cache, the particular line of the set will be determined according to a *replacement algorithm*.

- The memory address is interpreted as three fields by the cache logic, similar to direct mapping.
  However, a smaller number of bits (13 in example) are used to identify the *set of lines* in the cache ⇒ the tag field will be larger (9 bits in example).

- Several tags (corresponding to all lines in the set) have to be checked in order to determine if we have a hit or miss. If we have a hit, the cache logic finally points to the actual line in the cache.

- The number of lines in a set is determined by the designer;
  2 lines/set: two-way set associative mapping
  4 lines/set: four-way set associative mapping
  8 ....

Set Associative Mapping

- Set associative mapping keeps most of the advantages of direct mapping:
  - short tag field
  - fast access
  - relatively simple

- Set associative mapping tries to eliminate the main shortcoming of direct mapping; a certain flexibility is given concerning the line to be replaced when a new block is read into the cache.

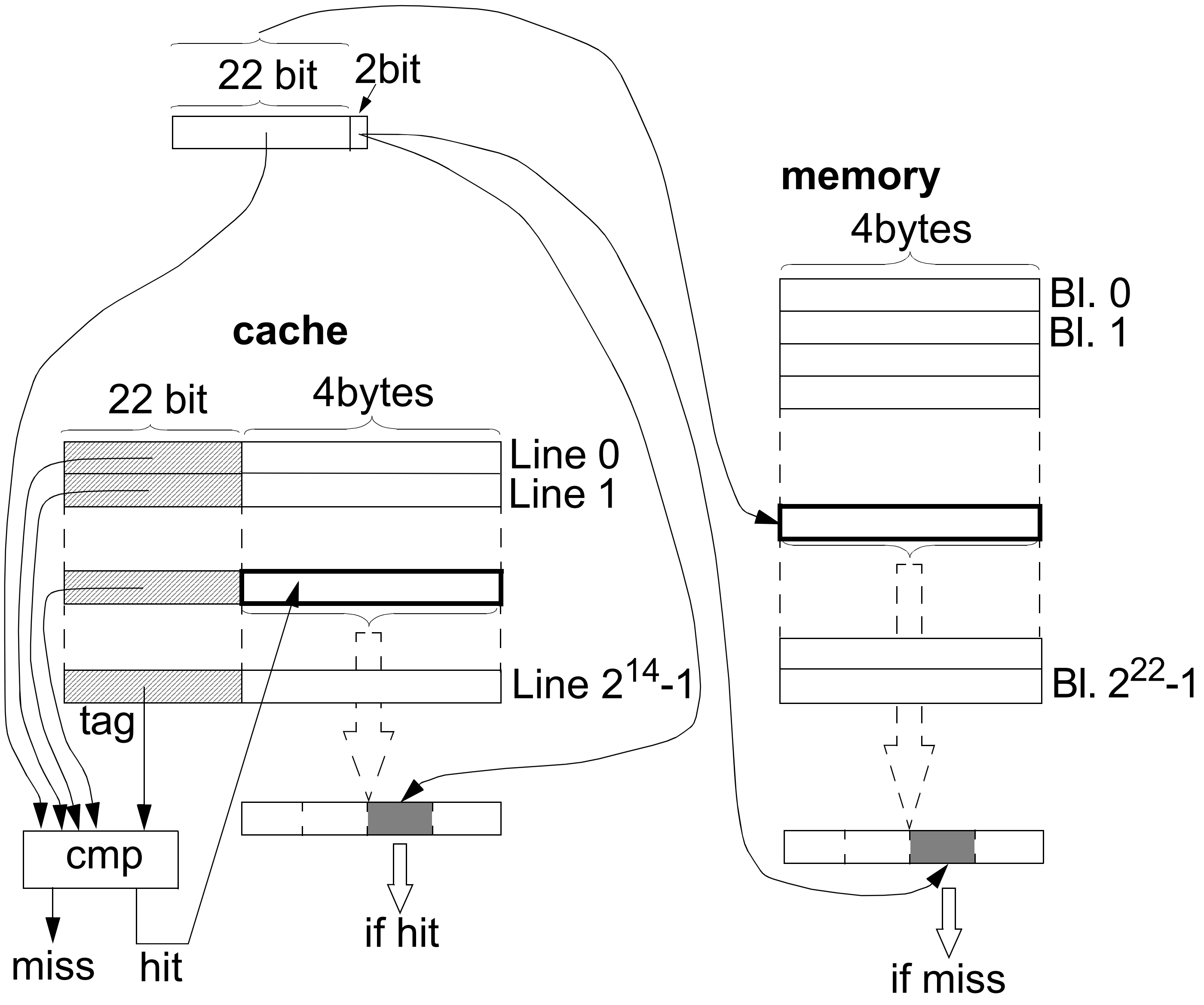
- Cache hardware is more complex for set associative mapping than for direct mapping.

- In practice 2 and 4-way set associative mapping are used with very good results.

- If a set consists of a single line ⇒ *direct mapping*;

  If there is one single set consisting of all lines ⇒ *associative mapping*.

Associative Mapping



If we had a miss, the block will be placed in one of the 2^{14} cache lines. The *replacement algorithm* decides which line to use.

Associative Mapping

- A memory block can be mapped to any cache line.

- If a block has to be placed in the cache, the particular line will be determined according to a *replacement algorithm*.

- The memory address is interpreted as two fields by the cache logic.
  The least significant bits (2 in example) identify the byte within the block; all the rest of the address (22 bits in example) is interpreted by the cache logic as a tag.

- All tags, corresponding to every line in the cache memory, have to be checked in order to determine if we have a hit or miss. If we have a hit, the cache logic finally points to the actual line in the cache.

  The cache line is retrieved based on a portion of its content (the tag field) rather than its address. Such a memory structure is called *associative memory*.

Associative Mapping

Advantages:

- ❐ Associative mapping provides the highest flexibility concerning the line to be replaced when a new block is read into the cache.

Disadvantages:

- ❐ complex
- ❐ the tag field is long
- ❐ fast access can be achieved only using high performance associative memories for the cache, which is difficult and expensive.

Cache Memory

Problems concerning cache memories:

- ❒ **Instruction and data in the same cache or not?**
- ❒ **How to determine at a *read* if we have a miss or hit?**
- ❒ **If there is a miss where to place the new item in the cache? Which information should be replaced?**
- ❒ **How to preserve consistency between cache and main memory at *write*?**

Replacement Algorithms

When a new block is to be placed into the cache, the block stored in one of the cache lines has to be replaced.

- **With direct mapping there is no choice.**

- **With associative or set-associative mapping a replacement algorithm is needed in order to determine which block to replace (and, implicitly, in which cache line to place the block);**
    - **with set-associative mapping, the candidate lines are those in the selected set;**
    - **with associative mapping, all lines of the cache are potential candidates;**

Replacement Algorithms

- Random replacement: One of the candidate lines is selected randomly.

All the other policies are based on information concerning the usage history of the blocks in the cache.

- Least recently used (LRU):

  The candidate line is selected which holds the block that has been in the cache the longest without being referenced.

- First-in-first-out (FIFO):

  The candidate line is selected which holds the block that has been in the cache the longest.

- Least frequently used (LFU):

  The candidate line is selected which holds the block that has got the fewest references.

Replacement Algorithms

- Replacement algorithms for cache management have to be implemented in hardware in order to be effective.

- LRU is the most efficient: relatively simple to implement and good results.
- FIFO is simple to implement.
- Random replacement is the simplest to implement and results sometimes are surprisingly good.

Cache Memory

Problems concerning cache memories:

- ❒ **Instruction and data in the same cache or not?**
- ❒ **How to determine at a *read* if we have a miss or hit?**
- ❒ **If there is a miss where to place the new item in the cache? Which information should be replaced?**
- ❒ **How to preserve consistency between cache and main memory at *write*?**

Write Strategies

The problem:

How to keep cache content and the content of main memory consistent without losing too much performance?

- **Problems arise when a write is issued to a memory address, and the content of the respective address is potentially changed.**

Write Strategies

- **Write-through**

  All write operations are passed to main memory; if the addressed location is currently hold in the cache, the cache is updated so that it is coherent with the main memory.

  ⇓

  For writes, the processor always slows down to main memory speed.

Write Strategies

- <u>Write-through</u>

  All write operations are passed to main memory; if the addressed location is currently hold in the cache, the cache is updated so that it is coherent with the main memory.

  ↓

  For writes, the processor always slows down to main memory speed.

- <u>Write-through with buffered write</u>

  The same as write-through, but instead of slowing the processor down by writing directly to main memory, the write address and data are stored in a high-speed write buffer; the write buffer transfers data to main memory while the processor continues it’s task.

  ↓

  Higher speed, more complex hardware

Write Strategies

- Copy-back

  Write operations update only the cache memory which is not kept coherent with main memory; cache lines have to remember if they have been updated; if such a line is replaced from the cache, its content has to be copied back to memory.

  ↓

  Good performance (usually several writes are performed on a cache line before it is replaced and has to be copied into main memory), more complex hardware.

Write Strategies

- **Copy-back**

  Write operations update only the cache memory which is not kept coherent with main memory; cache lines have to remember if they have been updated; if such a line is replaced from the cache, its content has to be copied back to memory.

  Good performance (usually several writes are performed on a cache line before it is replaced and has to be copied into main memory), more complex hardware.

Cache coherence problems are very complex and difficult to solve in multiprocessor systems!

Some Cache Architectures

- Intel 80486
  - a single on-chip cache of 8 Kbytes
  - line size: 16 bytes
  - 4-way set associative organization

- Pentium
  - two on-chip caches, for data and instructions.
  - each cache: 8 Kbytes
  - line size: 32 bytes (64 bytes in Pentium 4)
  - 2-way set associative organization
    (4-way in Pentium 4)

- PowerPC 601
  - a single on-chip cache of 32 Kbytes
  - line size: 32 bytes
  - 8-way set associative organization

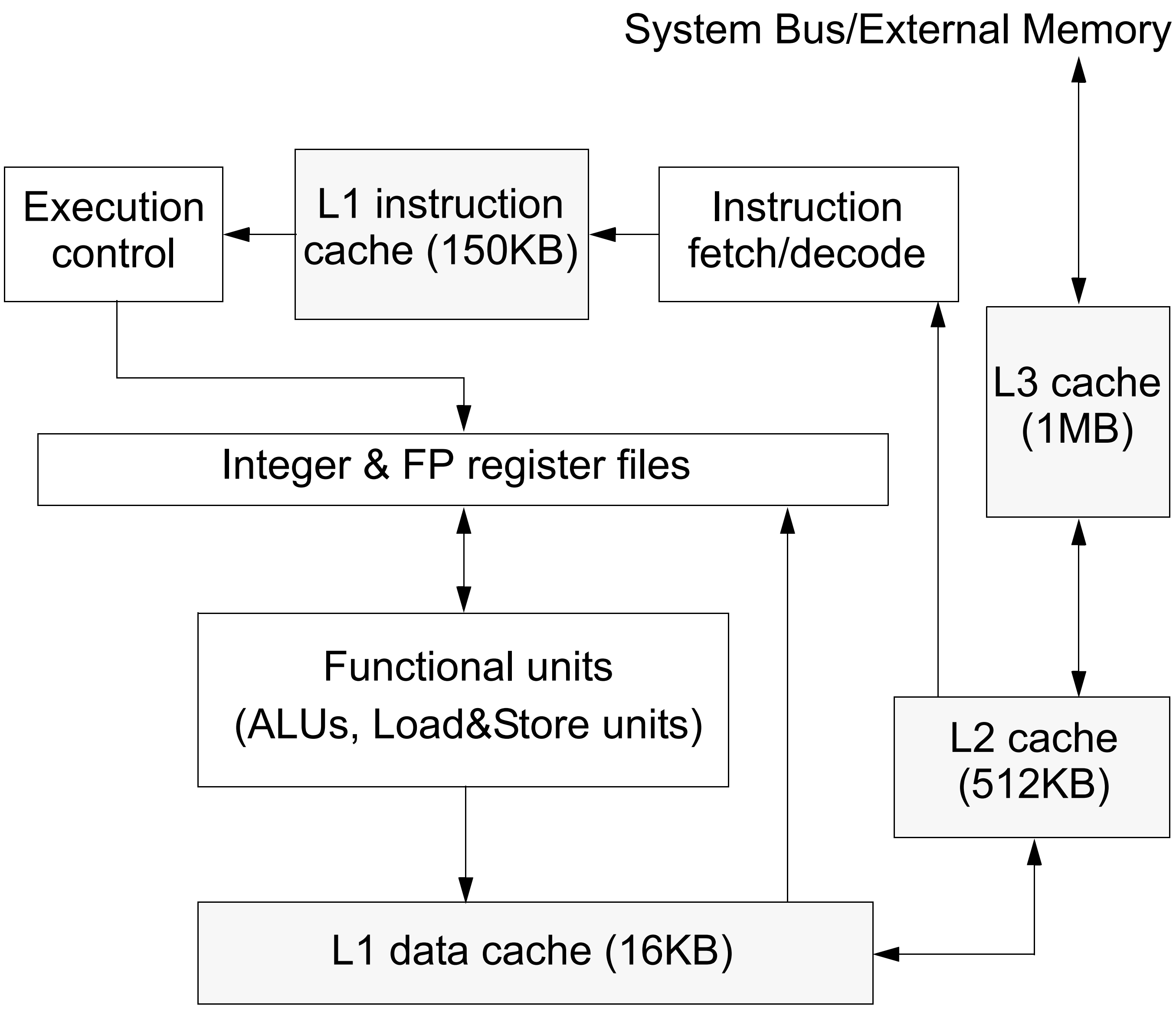
Some Cache Architectures

- PowerPC 603
  - two on-chip caches, for data and instructions
  - each cache: 8 Kbytes
  - line size: 32 bytes
  - 2-way set associative organization
    (simpler cache organization than the 601 but stronger processor)

- PowerPC 604
  - two on-chip caches, for data and instructions
  - each cache: 16 Kbytes
  - line size: 32 bytes
  - 4-way set associative organization

- PowerPC 620
  - two on-chip caches, for data and instructions
  - each cache: 32 Kbytes
  - line size: 64 bytes
  - 8-way set associative organization

Pentium 4 Cache Organization



Pentium 4 Cache Organization

- L1 data cache:
  16KB, line size: 64 bytes, 4-way set associative.
  Copy-back policy

- The Pentium 4 L1 instruction cache (150 KB, 8-way set associative) is often called *trace cache*.
  - The Pentium 4 fetches groups of x86 instructions from the L2 cache, decodes them into strings of microoperations (traces), and stores the traces into the L1 instruction cache.
  - The instructions in the cache are already decoded and, for execution, they only are fetched from the cache. Thus, decoding is done only once, even if the instruction is executed several times (e.g. in loops).
  - The fetch unit also performs address calculation and branch prediction when constructing the traces to be stored in the cache.

- L2 cache:
  512 KB, line size: 128 bytes, 8-way set associative.
- L3 cache:
  1 MB, line size: 128 bytes, 8-way set associative.
- The L2 and L3 cache are unified instruction/data caches.
- Instruction/data are fetched from external main memory only if absent from L1, L2, and L3 cache.

ARM Cache Organization

- ARM3 and ARM 6 had a 4KB unified cache.
- ARM 7 has a 8 KB unified cache.
- Starting with ARM9 there are separate data/instr. caches:
  - ARM9, ARM10, ARM11, Cortex: up to 128/128KB instruction and data cache.
  - StrongARM: 16/16KB instruction and data cache.
  - Xscale: 32/32KB instruction and data cache.
- Line size: 8 (32bit) words, except ARM7 and StrongArm with 4 words.
- Set associativity:
  - 4-way: ARM7, ARM9E, ARM10EJ-S, ARM11
  - 64-way: ARM9T, ARM10E
  - 32-way: StrongARM, Xscale
  - various options: Cortex
- With the Cortex, an L2 internal cache is introduced
- Write strategy: write through with buffered write

Virtual Memory

The memory space needed and seen by a program is usually much larger than the available main memory.

↓

Only one part of the program fits into main memory; the rest is stored on secondary memory (such as hard disk).

- **In order to be executed or data to be accessed, a certain segment of the program has to be first loaded into main memory; in this case it has to replace another segment already in memory.**
- **Movement of code and data, between main memory and secondary storage, is performed automatically by the *operating system*. These techniques are called *virtual-memory* techniques.**

Virtual Memory

The memory space needed and seen by a program is usually much larger than the available main memory.

↓

Only one part of the program fits into main memory; the rest is stored on secondary memory (such as hard disk).

- **In order to be executed or data to be accessed, a certain segment of the program has to be first loaded into main memory; in this case it has to replace another segment already in memory.**

- **Movement of code and data, between main memory and secondary storage, is performed automatically by the *operating system*. These techniques are called *virtual-memory* techniques.**

- **The binary address issued by the processor is a *virtual (logical) address*; it considers a *virtual address space*, much larger than the physical one available in main memory.**

Virtual Memory Organization

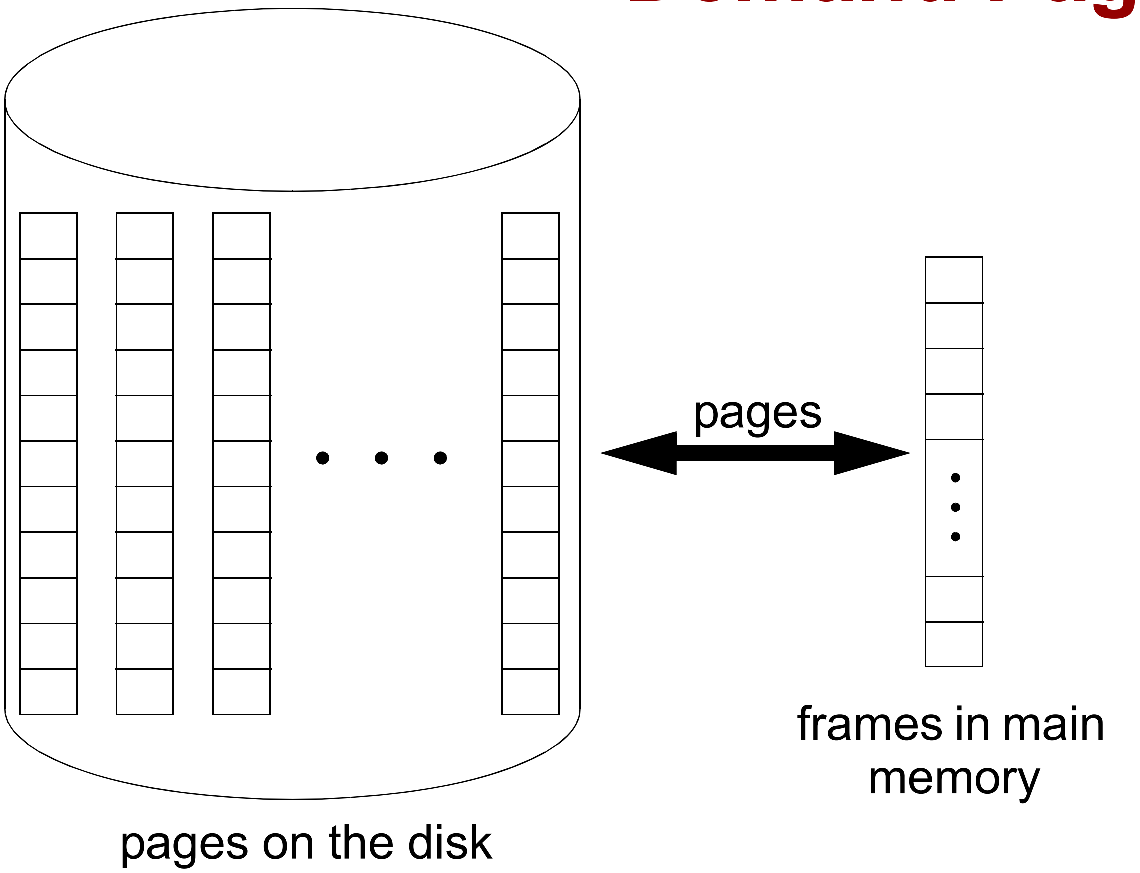
- The virtual memory space associated to a program (instructions + data) is divided into equal, fixed-size chunks called *pages*.

- Physical main memory is organized as a sequence of *frames*; a page can be assigned to an available frame in order to be stored (page size = frame size).

- The page is the basic unit of information which is moved between main memory and secondary memory by the virtual memory system.

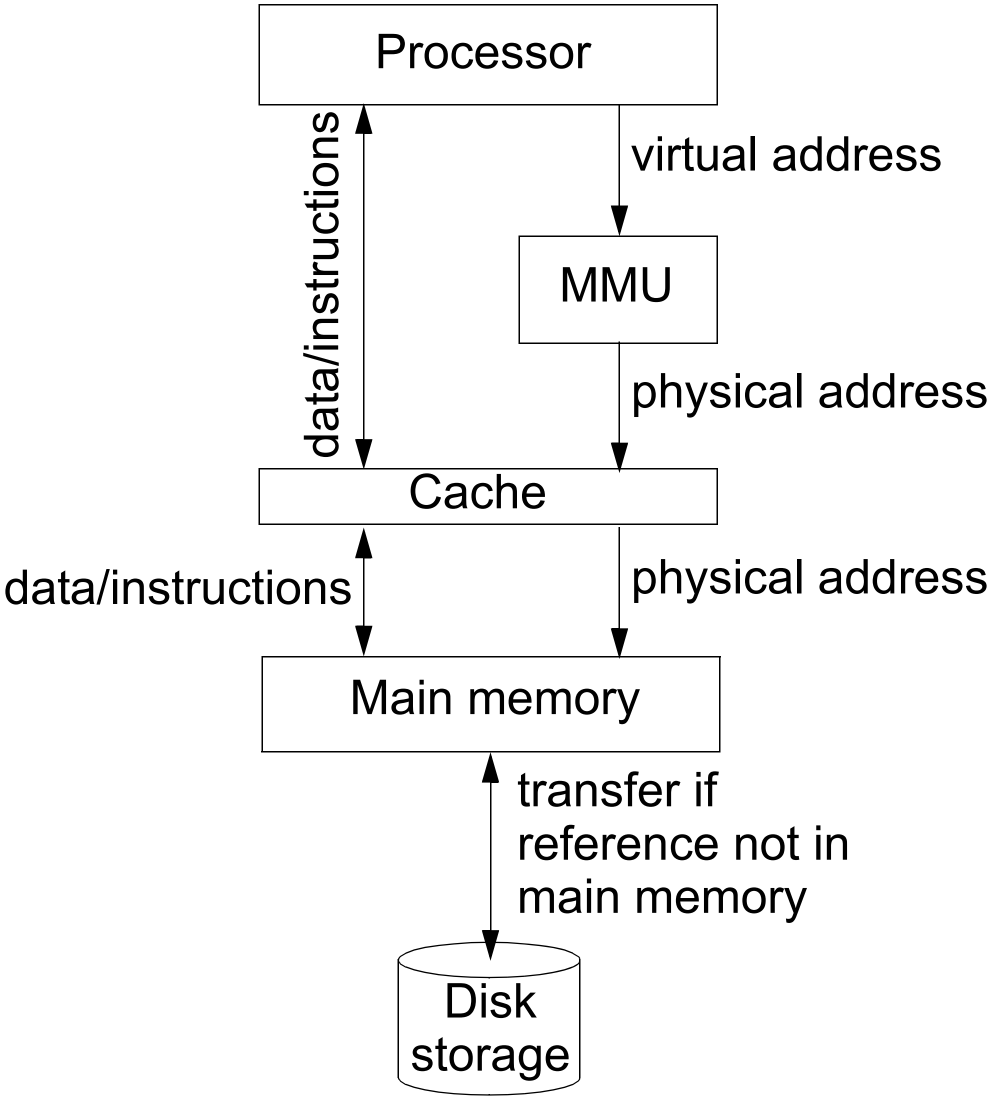
- Common page sizes are: 2 - 16Kbytes.

Demand Paging



- The program consists of a large amount of pages stored on secondary memory; at any time, only a few pages need to be stored in main memory.
- The operating system is responsible for loading/replacing pages so that the number of *page faults* is minimized.
- We have a *page fault* when the CPU refers to a location in a page that is not in main memory; this page has then to be loaded and, if there is no available frame, it has to replace a page which previously was in memory.

Demand Paging

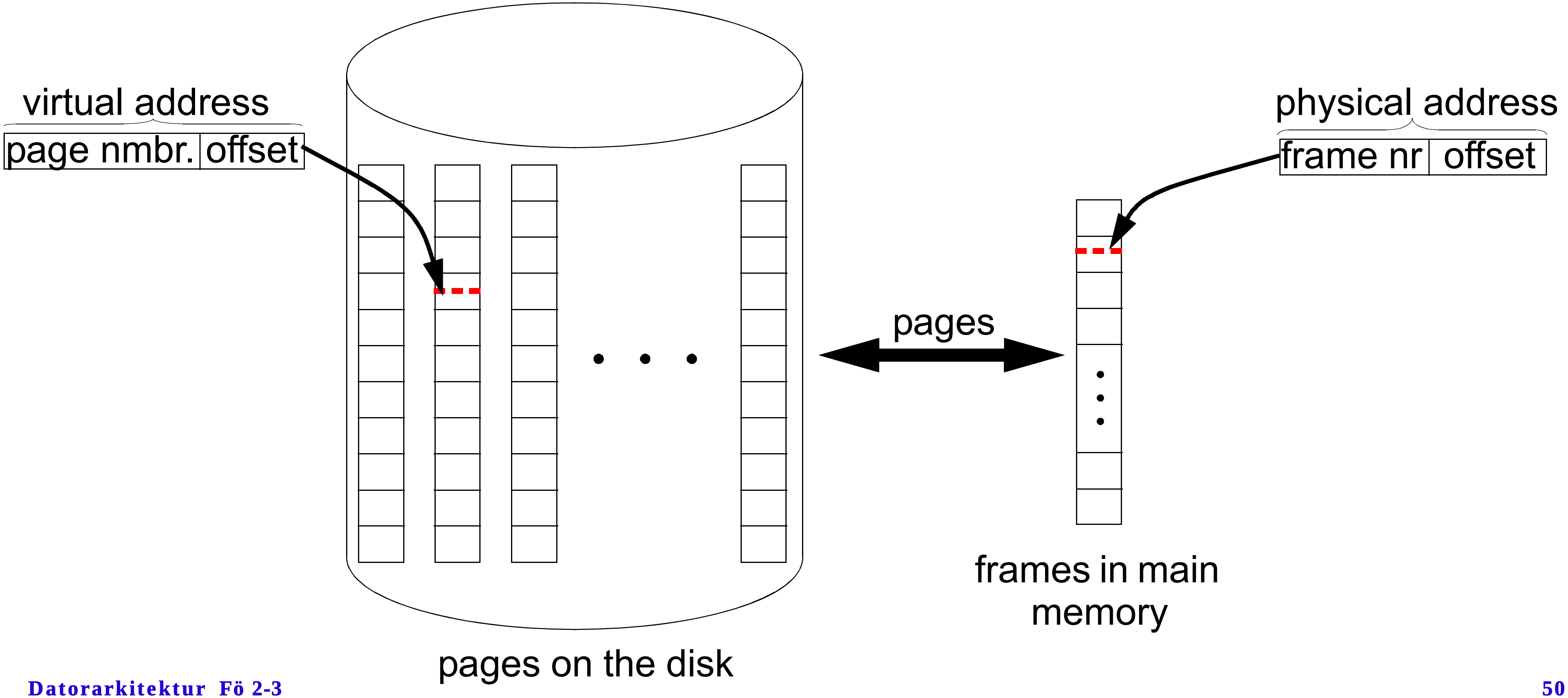


- If a *virtual address* refers to an item that is currently in the main memory, then the appropriate location is accessed immediately using the respective *physical address*; if this is not the case, the page containing the item has to be transferred first from secondary memory.

- A special hardware unit, *Memory Management Unit* (MMU), translates virtual addresses into physical ones.

Address Translation

- Accessing a word in memory involves the translation of a virtual address into a physical one:
  - ❐ *virtual address*: page number + offset
  - ❐ *physical address*: frame number + offset
- Address translation is performed by the MMU using a *page table*.



Address Translation

- Accessing a word in memory involves the translation of a virtual address into a physical one:
  - *virtual address*: page number + offset
  - *physical address*: frame number + offset
- Address translation is performed by the MMU using a *page table*.

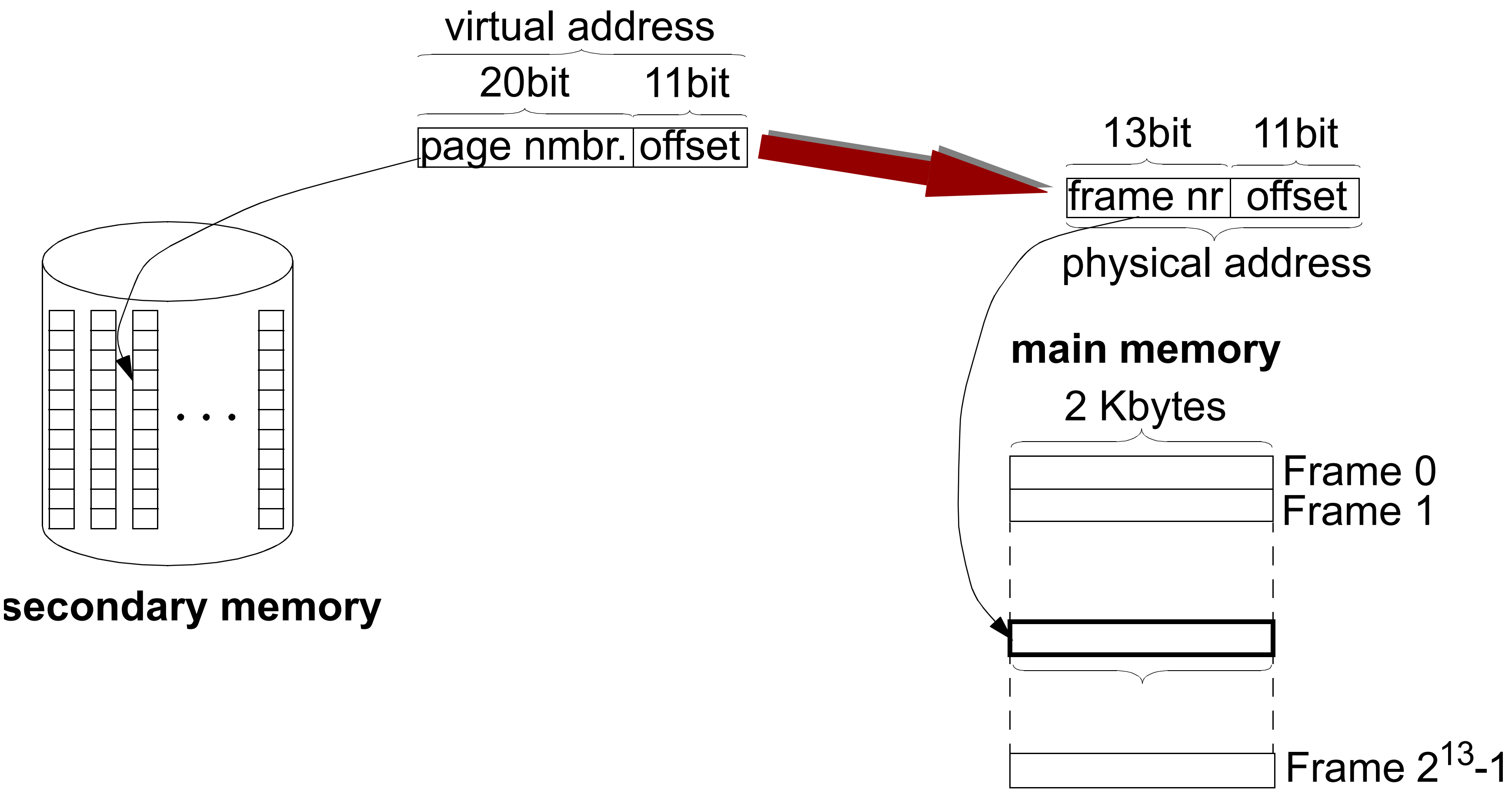
Example:

- Virtual memory space: 2 Gbytes
  (31 address bits; 2^{31} = 2 G)
- Physical memory space: 16 Mbytes (2^{24}=16M)
- Page length: 2Kbytes (2^{11} = 2K)

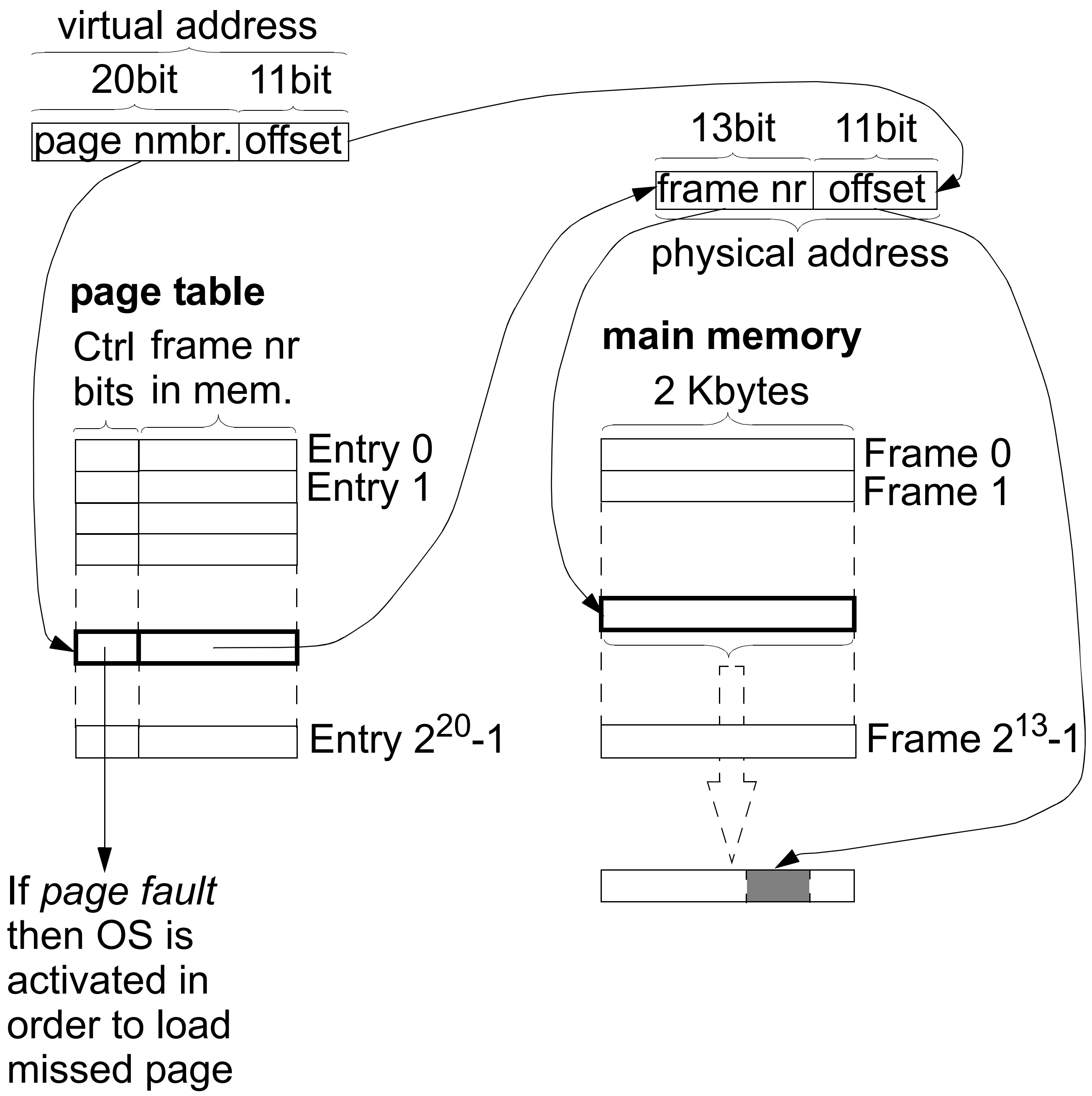
↓

Total number of pages: 2^{20} = 1M
Total number of frames: 2^{13} = 8K

Address Translation



Address Translation



If *page fault* then OS is activated in order to load missed page

The Page Table

- The page table has one entry for each page of the virtual memory space.
- Each entry of the page table holds the address of the memory frame which stores the respective page, if that page is in main memory.
- Each entry of the page table also includes some *control bits* which describe the status of the page:
  - whether the page is actually loaded into main memory or not;
  - if since the last loading the page has been modified;
  - information concerning the frequency of access, etc.

The Page Table

Problems:

- The page table is very large (number of pages in virtual memory space is very large).
- Access to the page table has to be very fast ⇒ the page table has to be stored in very fast memory, on chip.

⬇

- A special cache is used for page table entries, called *translation lookaside buffer* (TLB); it works in the same way as an ordinary memory cache and contains those page table entries which have been most recently used.

The Page Table

Problems:

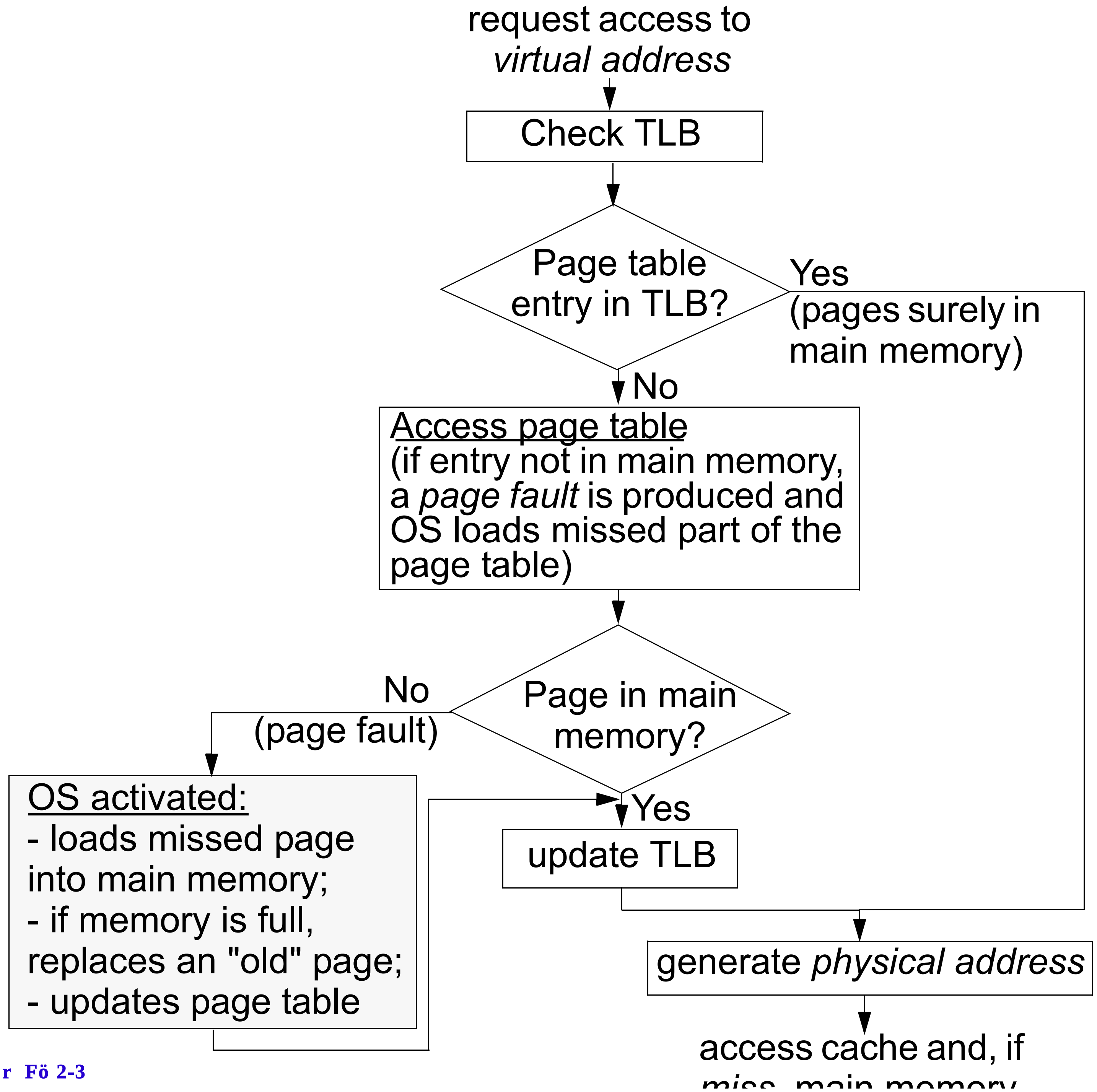
- The page table is very large (number of pages in virtual memory space is very large).
- Access to the page table has to be very fast ⇒ the page table has to be stored in very fast memory, on chip.

- A special cache is used for page table entries, called *translation lookaside buffer* (TLB); it works in the same way as an ordinary memory cache and contains those page table entries which have been most recently used.
- The page table is often too large to be stored in main memory. Virtual memory techniques are used to store the page table itself ⇒ only part of the page table is stored in main memory at a given moment.

The page table itself is distributed along the memory hierarchy:

- TLB (cache)
- main memory
- disk

Memory Reference with Virtual Memory and TLB



request access to *virtual address*

Check TLB

Page table entry in TLB?

Yes (pages surely in main memory)

No

Access page table (if entry not in main memory, a *page fault* is produced and OS loads missed part of the page table)

Page in main memory?

No (page fault)

OS activated:
- loads missed page into main memory;
- if memory is full, replaces an "old" page;
- updates page table

Yes

update TLB

generate *physical address*

access cache and, if *miss*, main memory

Memory Reference with Virtual Memory and TLB

- Memory access is solved by hardware except the page fault sequence which is executed by the OS software.

- The hardware unit which is responsible for translation of a virtual address into a physical one is the *Memory Management Unit* (MMU).

Page Replacement

- When a new page is loaded into main memory and there is no free memory frame, an existing page has to be replaced.
  The decision on which page to replace is based on the same speculations like those for replacement of blocks in cache memory.

  LRU strategy is often used to decide on which page to replace.

- When the content of a page, which is loaded into main memory, has been modified as result of a write, it has to be written back on the disk before its replacement.
  One of the control bits in the page table is used in order to signal that the page has been modified.