Computer Organization and Networks

(INB.06000UF, INB.07001UF)

Chapter 9: Memory Systems

Winter 2023/2024

TU Graz

Stefan Mangard, www.iaik.tugraz.at

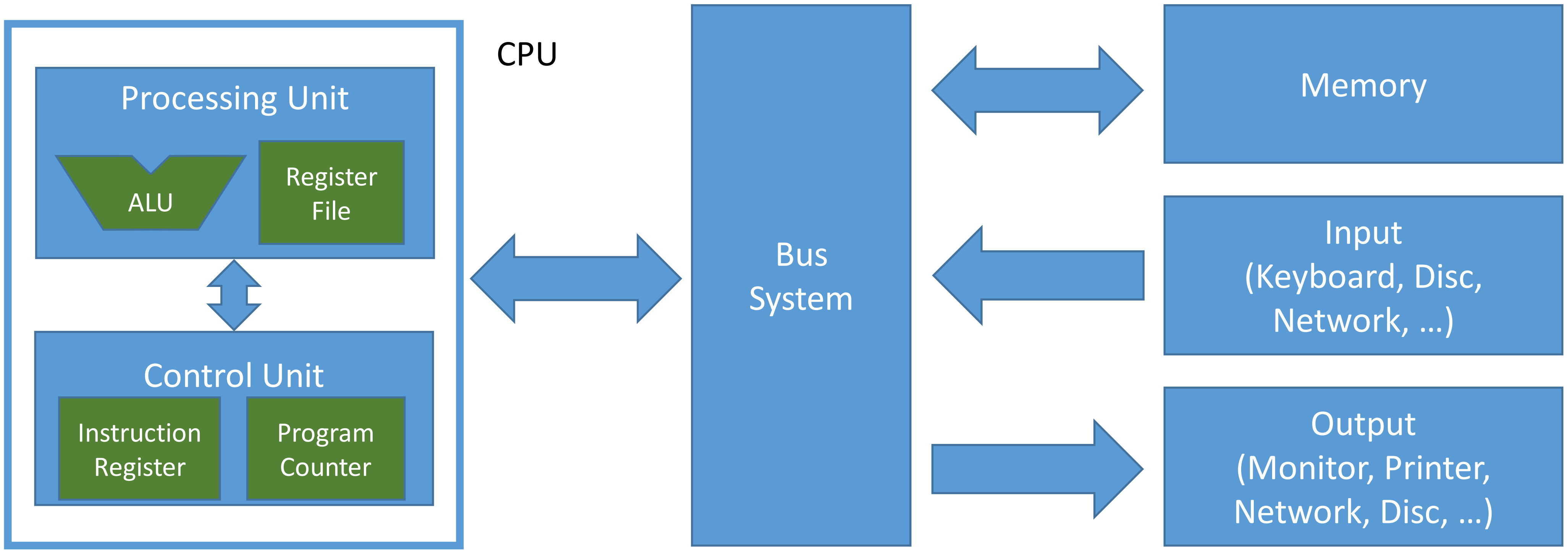
Note

- These slides use figures from the book “Digital Design and Computer Architecture RISC-V Edition” by Sarah L. Harris and David Harris. These figures are kindly provided by Elsevier and the Authors for educational purposes.

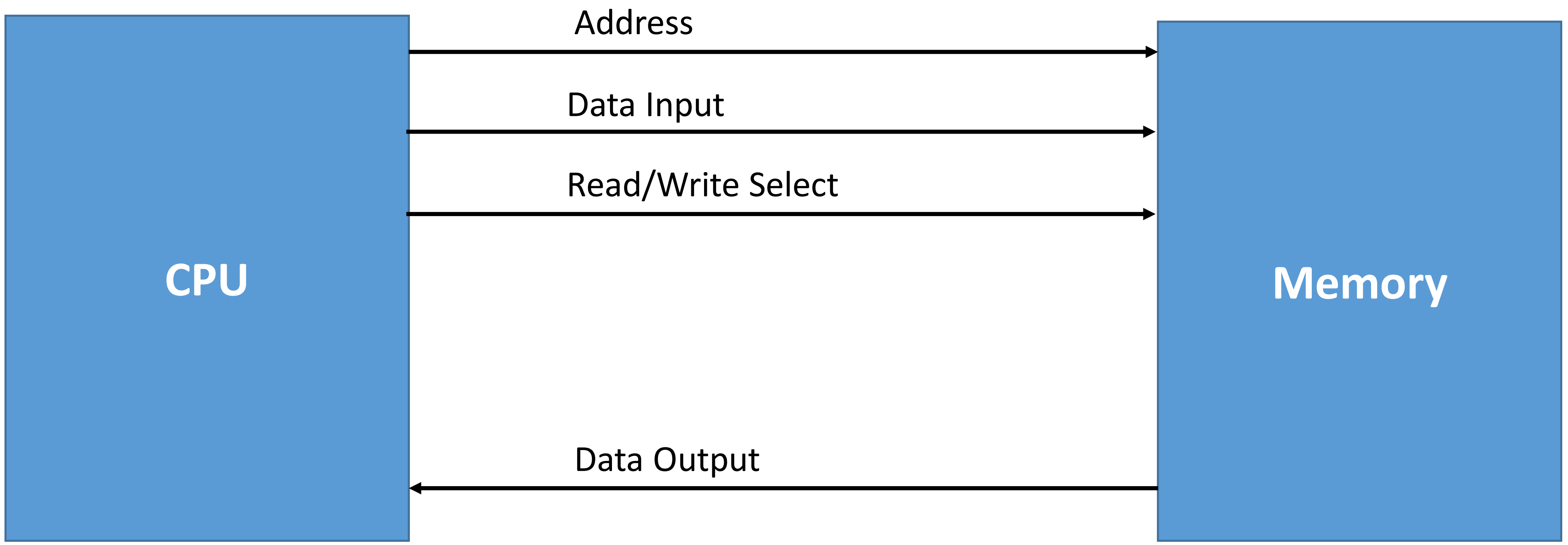
- The figures from the book have the caption “HH, F8.x” in the slides, where x stands for the figure number in chapter 8 of the book

- For all figures with this caption, it holds: “Figure Copyright © 2022, Elsevier Inc. All rights Reserved”

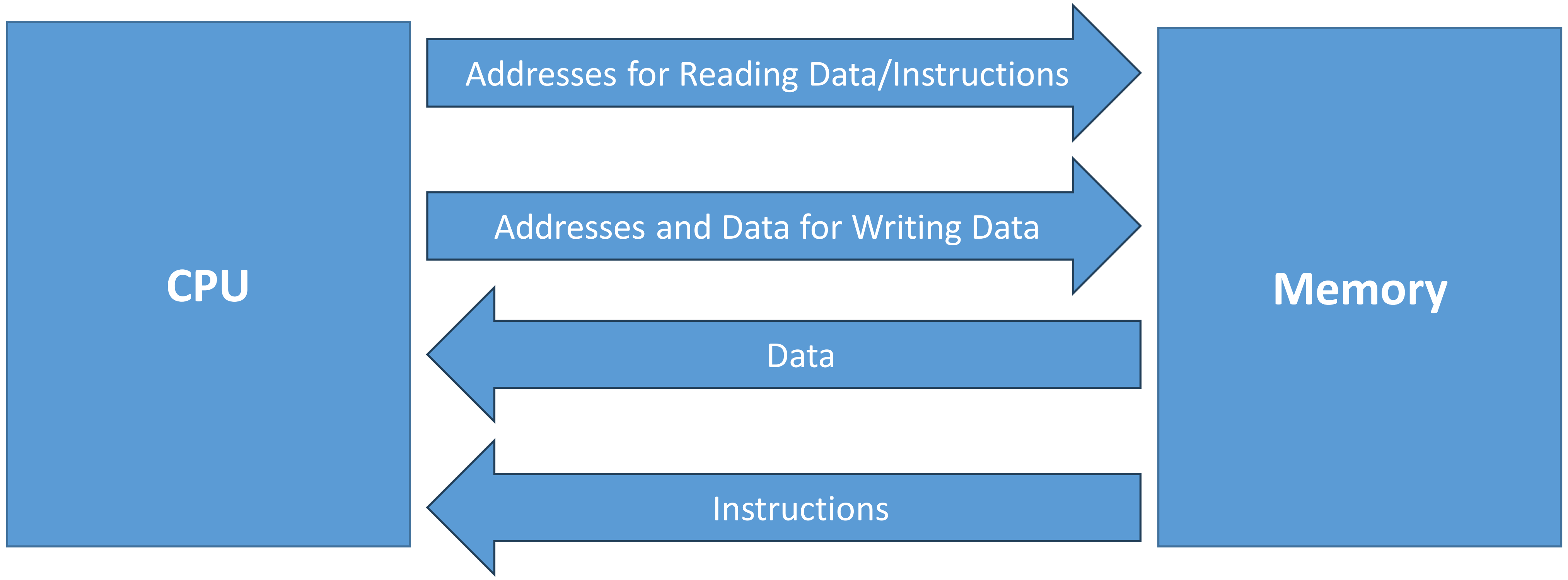
Von Neumann Model



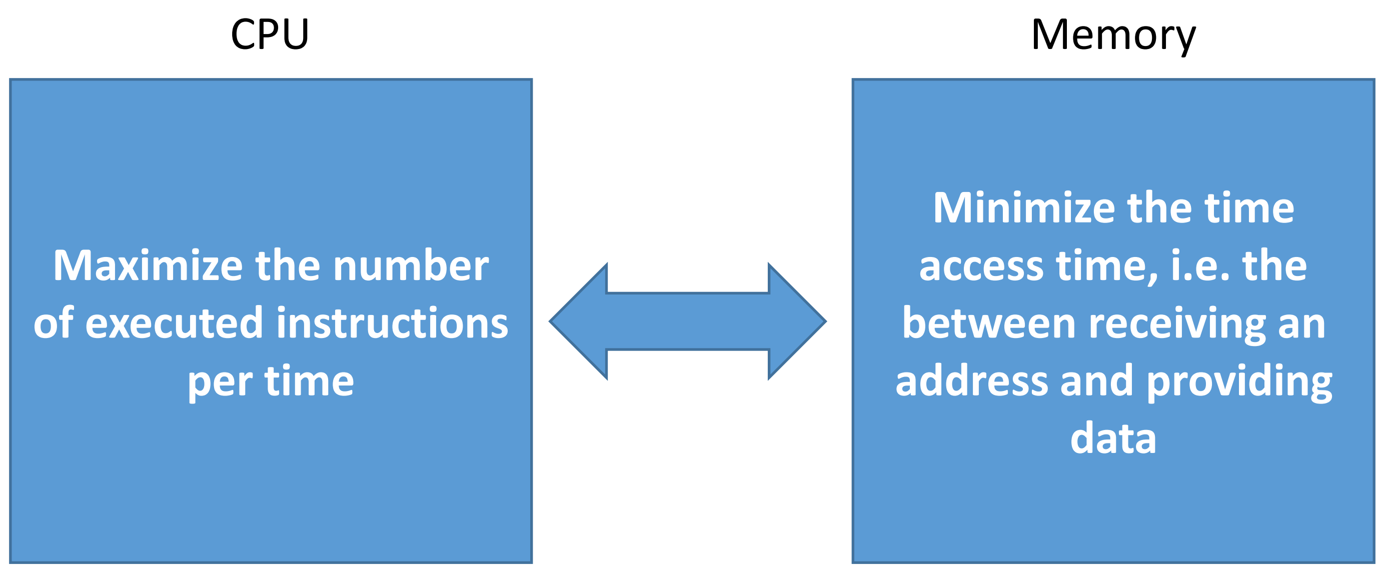
The Interface Between CPU and Memory



The Interface Between CPU and Memory


CPU
Addresses for Reading Data/Instructions
Addresses and Data for Writing Data
Data
Instructions
Memory

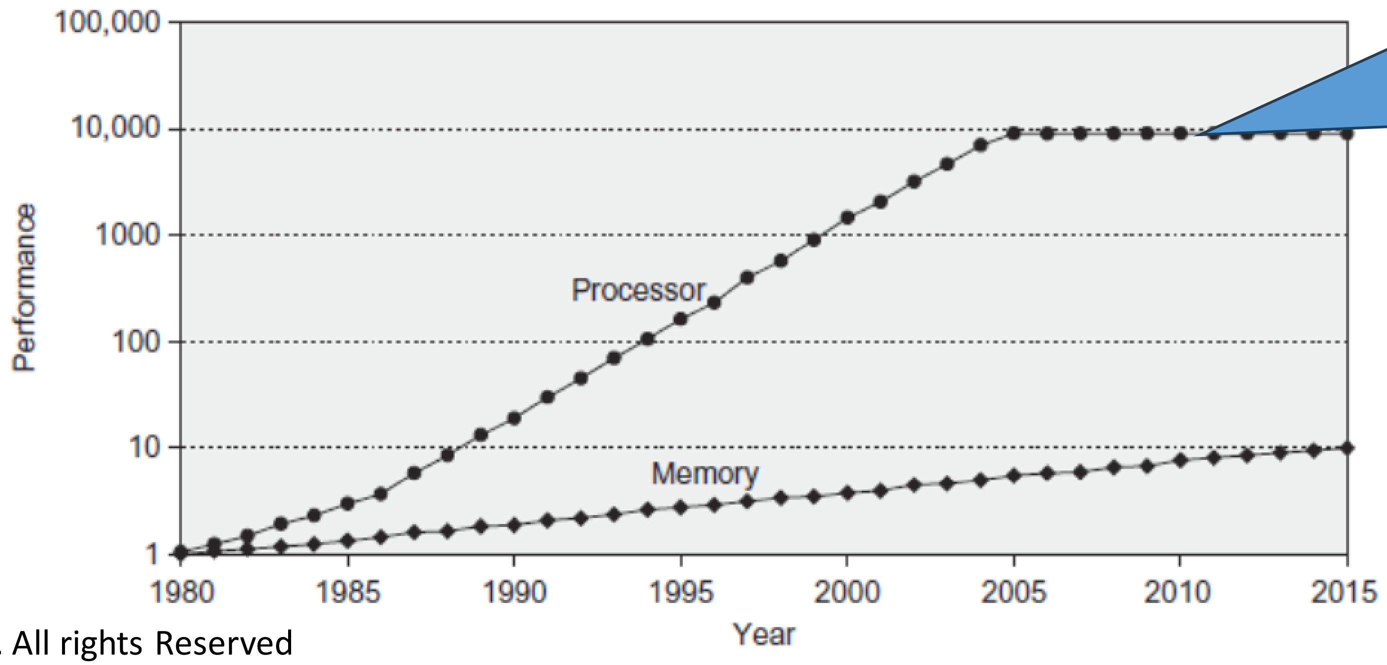
We want Speed –
Design Goals of CPU and Memory


CPU
Memory
Maximize the number of executed instructions per time
Minimize the time access time, i.e. the between receiving an address and providing data

- The speed of the CPU and the memory needs to be match
- If the CPU is faster than the memory, the CPU needs to wait for the memory to deliver instructions/data
- If the memory is faster than the CPU, the full speed of the memory is not used and limited by the speed of the CPU

Who is Faster?

- In 1980, we had the situation that it in one clock cycle of the CPU, it was possible to do one memory access
- CPU speed and memory speed developed differently since then



Performance gain on the CPU side is not through increased clock frequency, but due to architecture (e.g. parallelism)

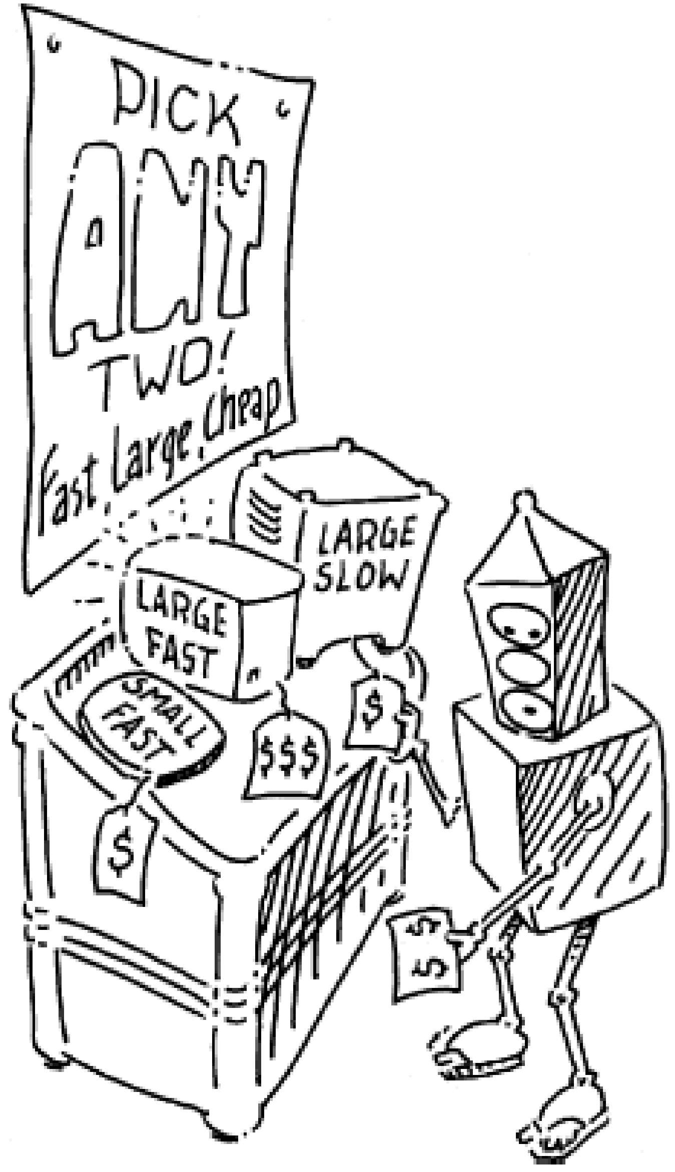
How can we provide a memory system that matches the performance requirements of today's CPUs?

Desired Memory Properties

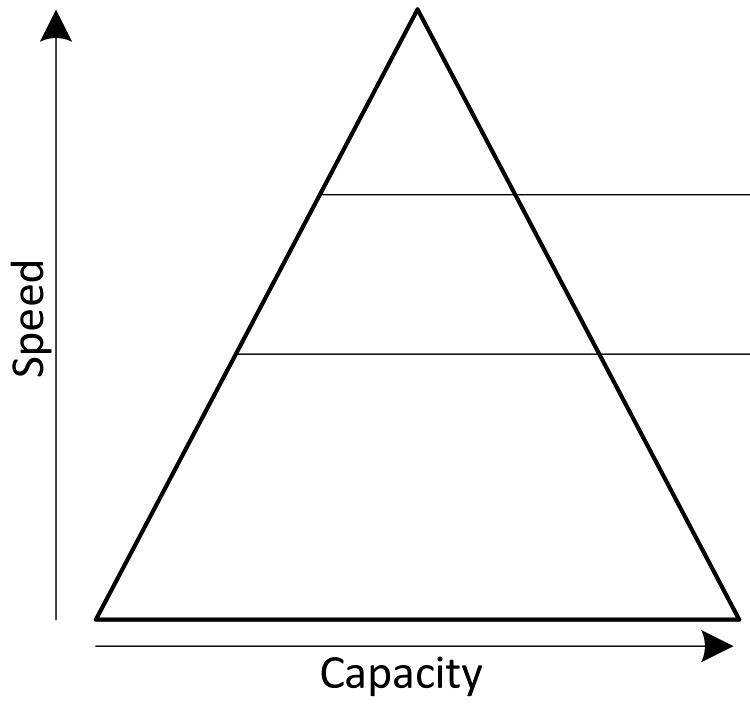
- Fast (maximum speed):
  - The time between sending the address and receiving/writing data (the latency) should be as short as possible
  - The amount of data that can be read/written per time (the bandwith) should be as high as possible
- Large (maximum size)
  - There should be as many data words as possible stored in memory
- Cheap (minimal cost)
  - The costs to build the memory should be as small as possible

The Reality on “Fast-Large-Cheap”

- We can only get two of the desired properties at the same time
- The requirements contradict each other
    - Larger is slower
    - Faster is more expensive
    - …



Properties of Different Memory Technologies



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$100	0.2 - 3	100+
DRAM	$3	10 - 50	30
SSD	$0.10	20,000	0.05 - 3
HDD	$0.03	5,000,000	0.001 - 0.1

Typical Characteristics of memories as of 2021 (based on HH, F8.4)

Properties of Memory Accesses

- **Temporal Locality**
  - Memory accesses have locality in time
  - If data used recently, likely to use it again soon
  - Example:
    - When performing computations, there is a set of input and intermediate variables and that are typically accessed several times during a computation
    - Instructions in a loop are accessed multiple times
    - …

- **Spatial Locality**
  - Memory accesses have locality in space
  - If data used recently, likely to use nearby data soon
  - Examples:
    - Instructions are read from memory one after the other in case of a linear code sequence
    - Data words are read one after the other when copying data or when reading/writing arrays
    - …

Hierarchical Memory Design

The properties

- of the different memory technologies(speed, size, cost) and
- the temporal/special locality of memory access

lead to the idea of a **hierarchical memory design**

Instead of building one single memory to serve the requests of the CPU …

Hierarchical Memory Design

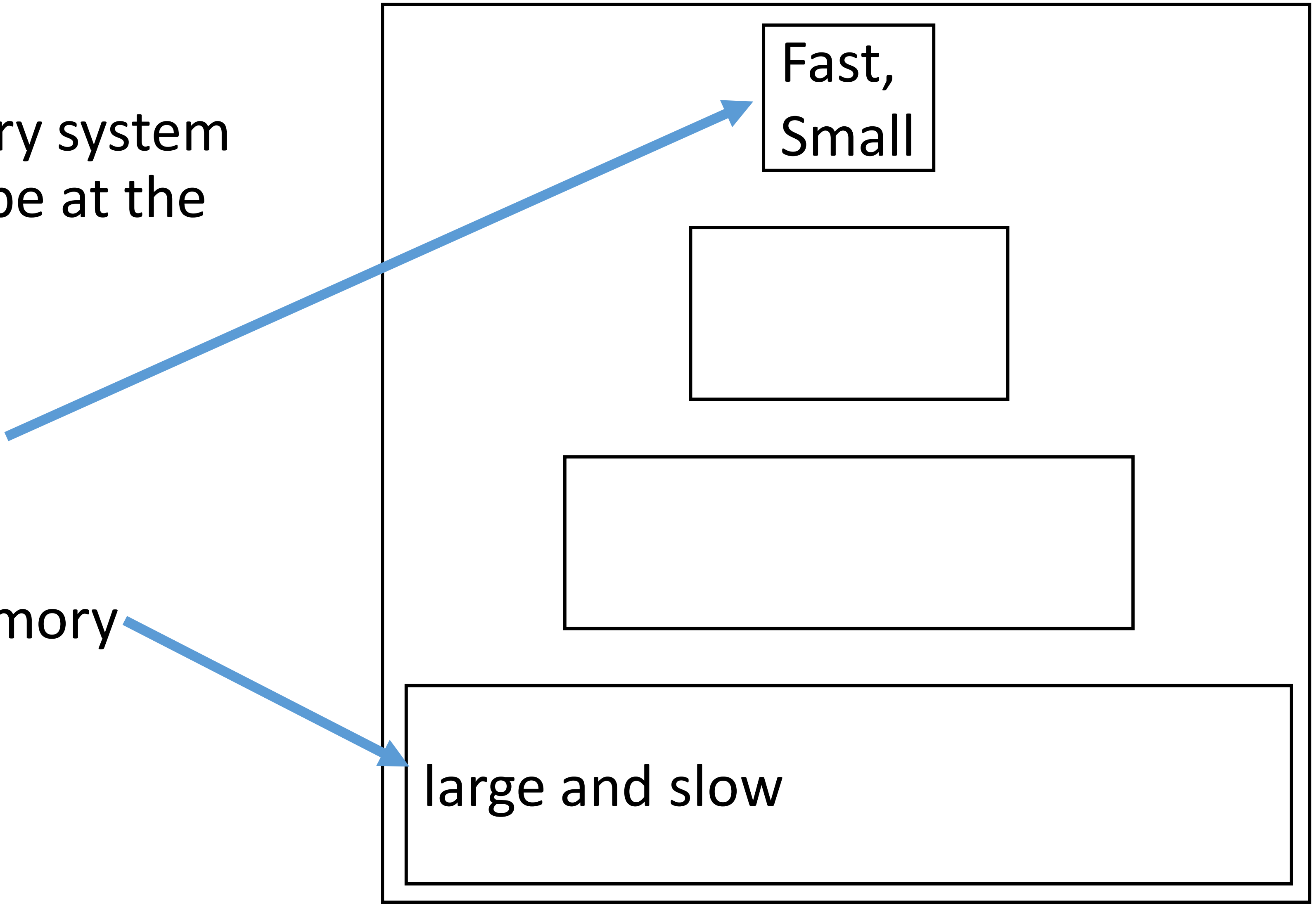
… we combine all available types of memory in order to “create the illusion” that the CPU is connected to memory that is

FAST & LARGE & CHEAP

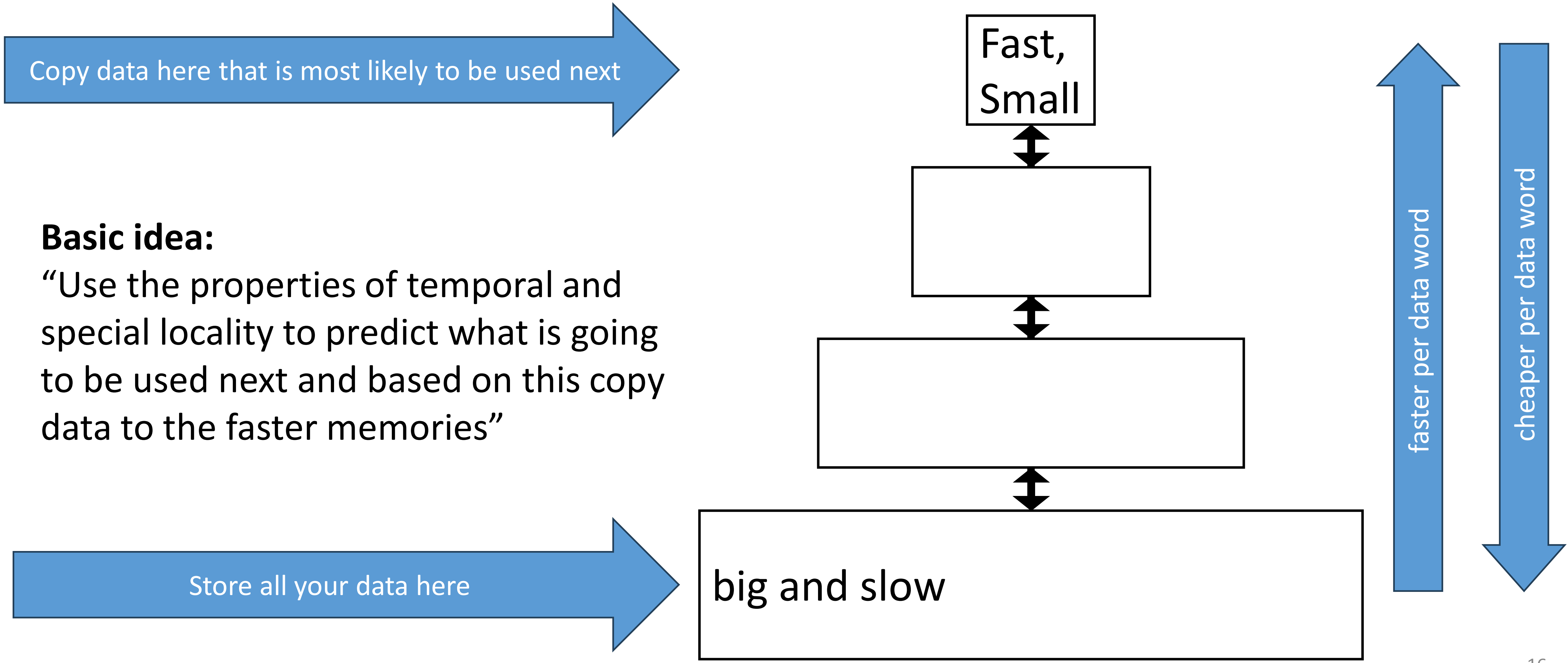
Hierarchical Memory Design

The goal is to build a memory system that appears to the CPU to be at the same time

- as a fast as small memory
- as large as the biggest memory



Memory Hierarchy – Basic Idea



Basic idea:
"Use the properties of temporal and special locality to predict what is going to be used next and based on this copy data to the faster memories"

Analogy – From the Library to your Desk

Three storage locations for books (different size, different access times)

- On your desk
- In your bookshelf
- In the library

Disclaimer:
This example is for illustration only.
This would not be healthy.

You can continuously study at your desk, if you have friends that do the following for you:

- Friend 1: Brings all books that are relevant for all courses that you enroll in a semester to your bookshelf at home
- Friend 2: Every day puts the books that you need for the courses of the day on our desk

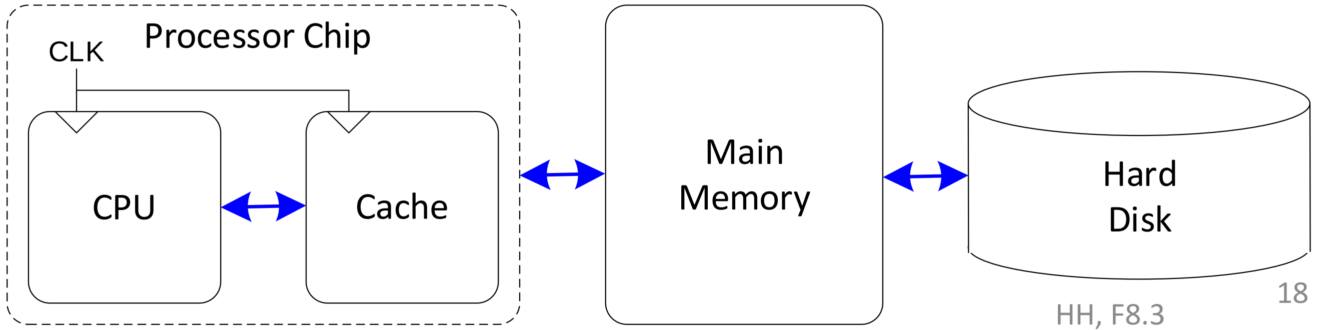
As long as you follow predictable patterns it will hold:

- Your desk (that is immediately accessible) appear as large as the library
- Only if your friends did not predict correctly what you need, you will need to wait

Caches – Realizing a Memory Hierarchy

- **Cache:**
  - Generic term referring to any structure that memorizes frequently used data
  - Idea: Instead of performing slow operations repeatedly again, store the result of these operations
  - Example: Cache of a browser

- **Caches for processor designs**
  - Idea: Build small SRAM memories next to the CPU as a cache for data in main memory
  - Modern CPUs typically have multiple layers of caches



HH, F8.3

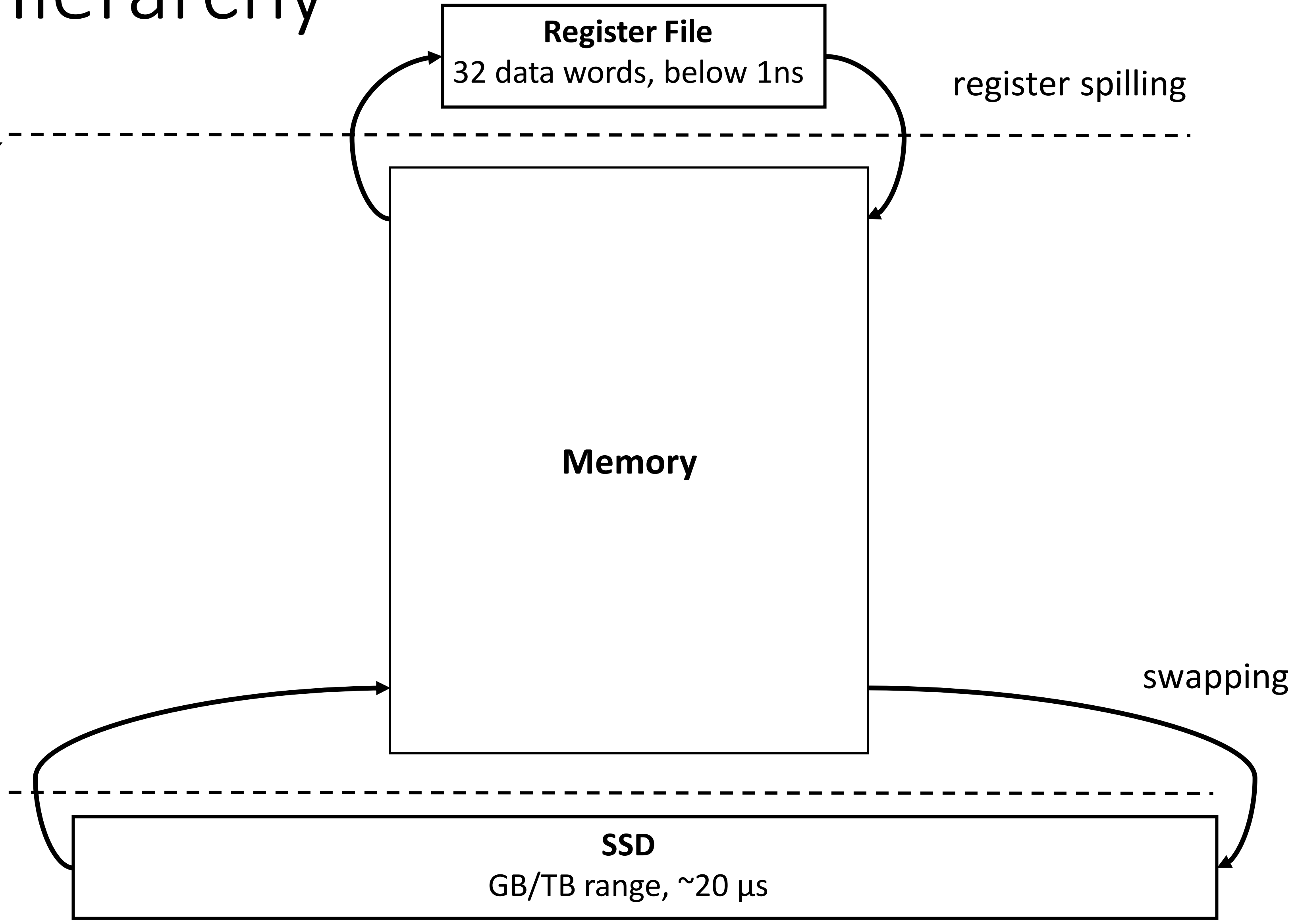
How to Manage Caches

- **Manual:** The Programmer explicitly decides when which data is moved between the different memory levels
- **Automatically in Software:** A piece of software (e.g. the operating system) implements an algorithm for an automated caching strategy
- **Automatically in Hardware:** The Hardware transparently for the software moves data between different levels of memory

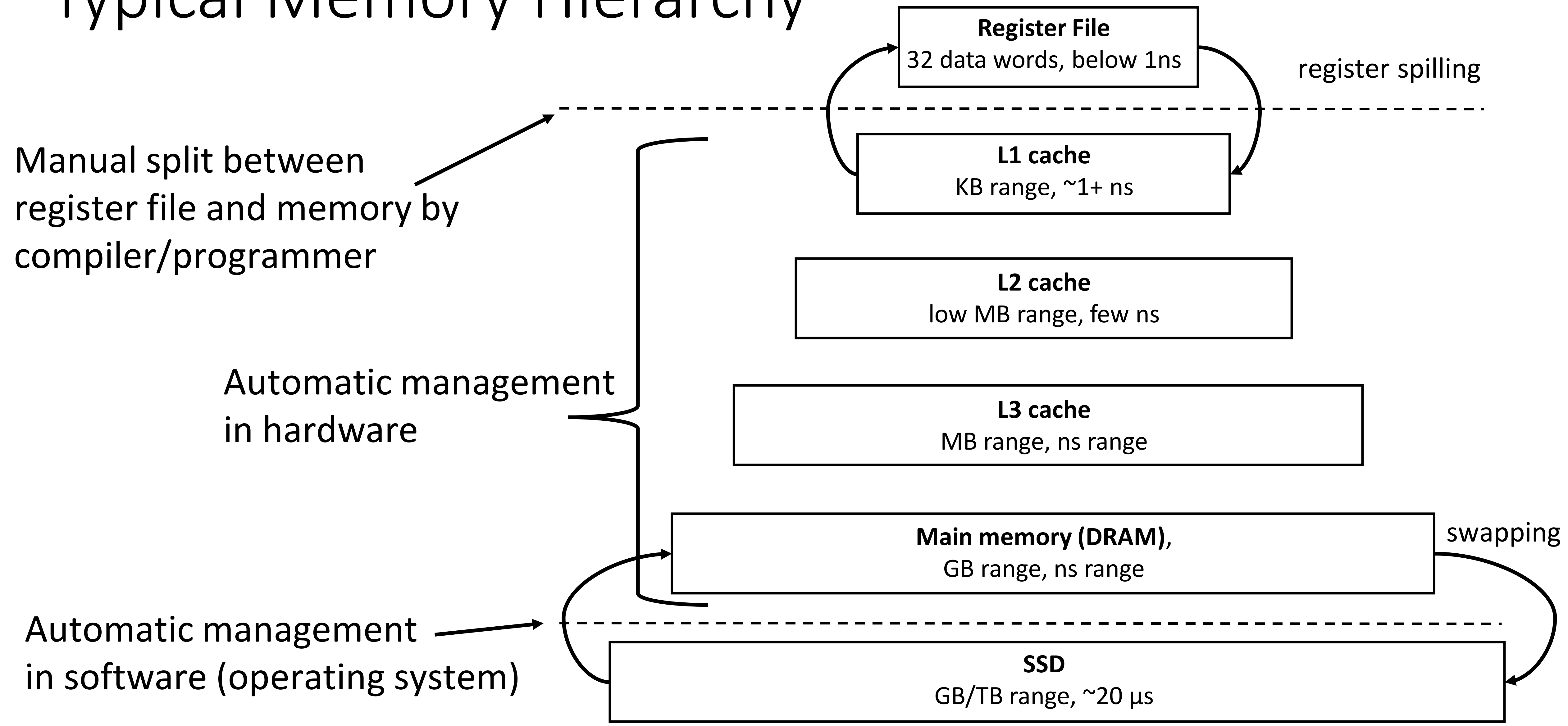
Typical Memory Hierarchy

Manual split between register file and memory by compiler/programmer

Automatic management in software (operating system)



Typical Memory Hierarchy



Memory Performance

- **Result of a memory access at a particular level can be**
  - **Hit:** data found in that level of memory hierarchy
  - **Miss:** data not found (must go to next level)

- **For each level there is a hit and a miss rate**
  - Hit Rate = # hits / # memory accesses
    = 1 – Miss Rate
  - Miss Rate = # misses / # memory accesses
    = 1 – Hit Rate

Memory Performance

- **Result of a memory access at a particular level can be**
  - **Hit:** data found in that level of memory hierarchy
  - **Miss:** data not found (must go to next level)

- **For each memory level i there is a hit and a miss rate**
  - Hit Rate (h_i) = # hits / # memory accesses
  - Miss Rate (m_i) = # misses / # memory accesses
  - It holds $h_i + m_i = 1$

- **For each memory level i there is a**
  - Memory access time t_i
  - Average memory access time (AMAT) / Perceived access time T_i

- **Calculating the perceived access time (T_i)**
  - $T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$
  - $T_i = t_i + m_i \cdot T_{i+1}$

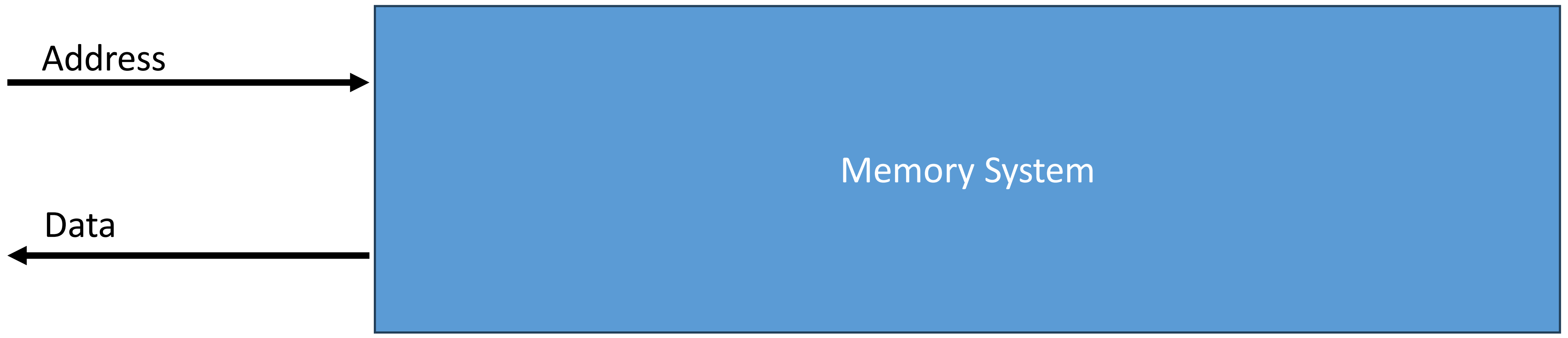
Goal and Design Considerations

On each memory level it holds $T_i = t_i + m_i \cdot T_{i+1}$

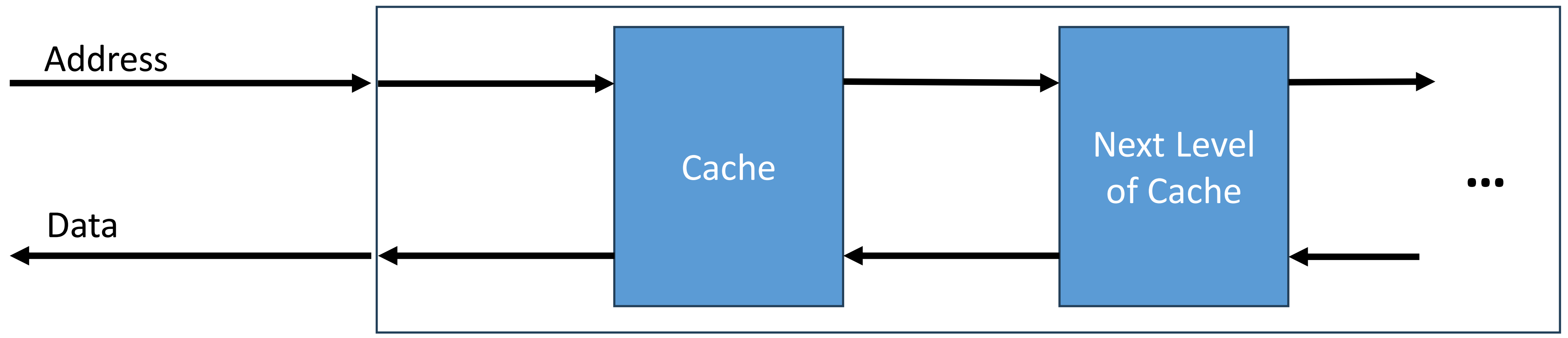
- **Goal:**
  - Minimize T_1 (This is the average memory access time observed by the CPU); Ideally T_1 should not increase much over t_1
- **Considerations to reach the goal**
  - Keep m_i low:
    - Increase the size of the cache (potentially increasing production cost or increasing the access time t_i)
    - Lower m_i through smart cache management (better predict what is needed next)
  - Keep T_{i+1} low:
    - At each level look for the best tradeoff of size, access time, and cost
    - Introduce additional cache hierarchies

Designing Caches

Cache Setting for Read Operations



Cache Setting for Read Operations



- **Cache**
  - Receives an address
  - Checks if data for this memory location is stored
  - If yes, data is delivered
  - If not, data is requested from the next cache level and stored in the cache (to have it in the cache for future accesses – motivated by temporal locality)

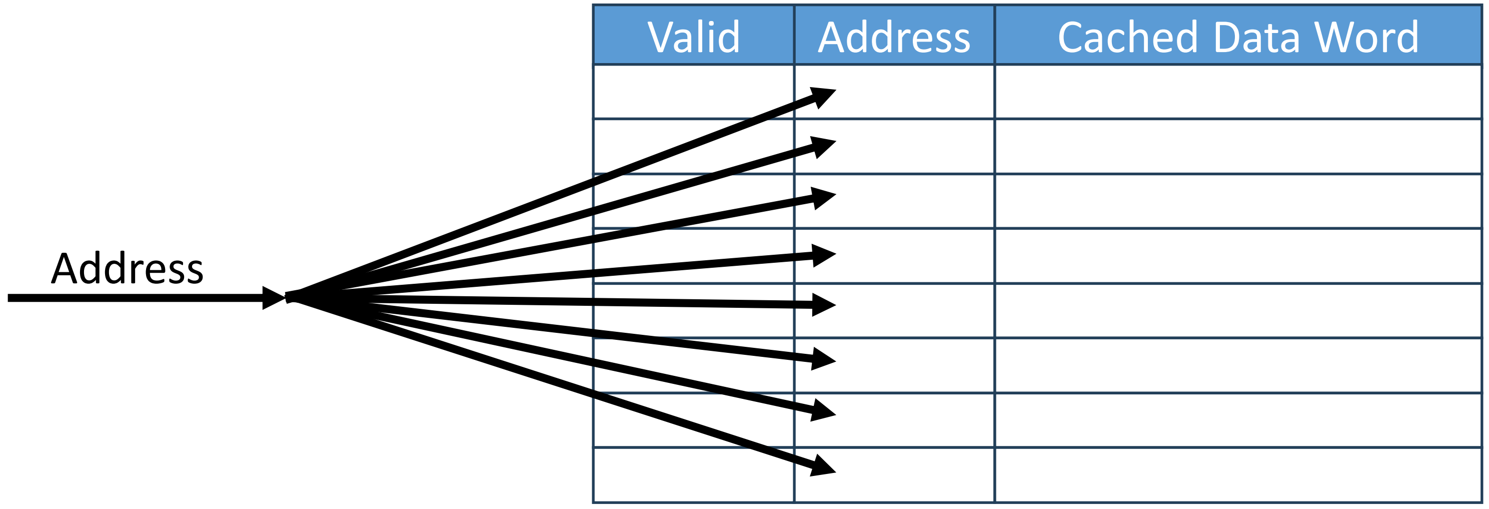
Metadata Stored in a Cache

A cache not only needs to store data, but also metadata

- **Information on the address of the stored data blocks:** Upon receiving an address, the cache needs to be able to determine if the block containing that address is in the cache or not

- **Bookkeeping data:** Additional metadata is necessary e.g. to keep information on valid blocks and to implement cache replacement policies

Naïve Approach



Assume

- a 32-bit CPU sending a 32-bit address
- a cache is able to store 8 data words
- a cache storing the triple of (a valid bit, the address of the data word in main memory, the cached data word)

Behavior

- The cache is empty at the beginning (the valid bits for all entries are 0)
- When the cache receives an address, the cache searches all 8 storage locations to determine if data for this address is stored or not (valid bit needs to be set and address needs to match)
- If there is match, data is delivered
- If there is a miss, data is looked up in the next level of the memory hierarchy and stored in the cache on an empty slot. In case the cache is full (8 valid entries) a cache entry needs to evicted

Observations

Let’s make this more efficient!

- It is expensive to have data transfers between caches at the data word level (many transfers)
- It is expensive to search through the entire cache for every access (assume a cache state that stores megabytes of data)
- We need some replacement policy to determine which data word to evict, in case we need to add an entry to a cache that is already full (8 valid entries)

Basics Design Principles of Caches

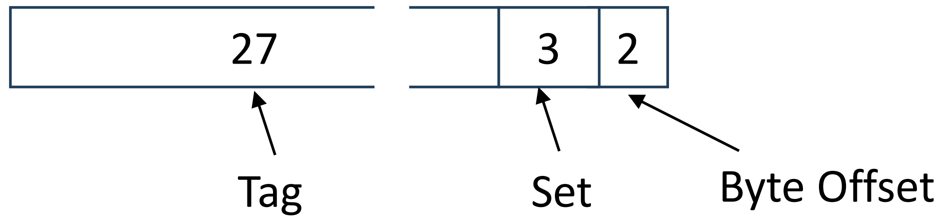
- **Store larger blocks and transfer larger blocks of memory between different levels of the memory hierarchy** instead of single words (this is motivated by spatial locality)
    - Capacity (C): The number of data words that can be stored in the cache
    - Block Size(b): The size of the blocks; A cache can store $B=C/b$ blocks
    - Block or Cache Line: This refers to the content of the data block

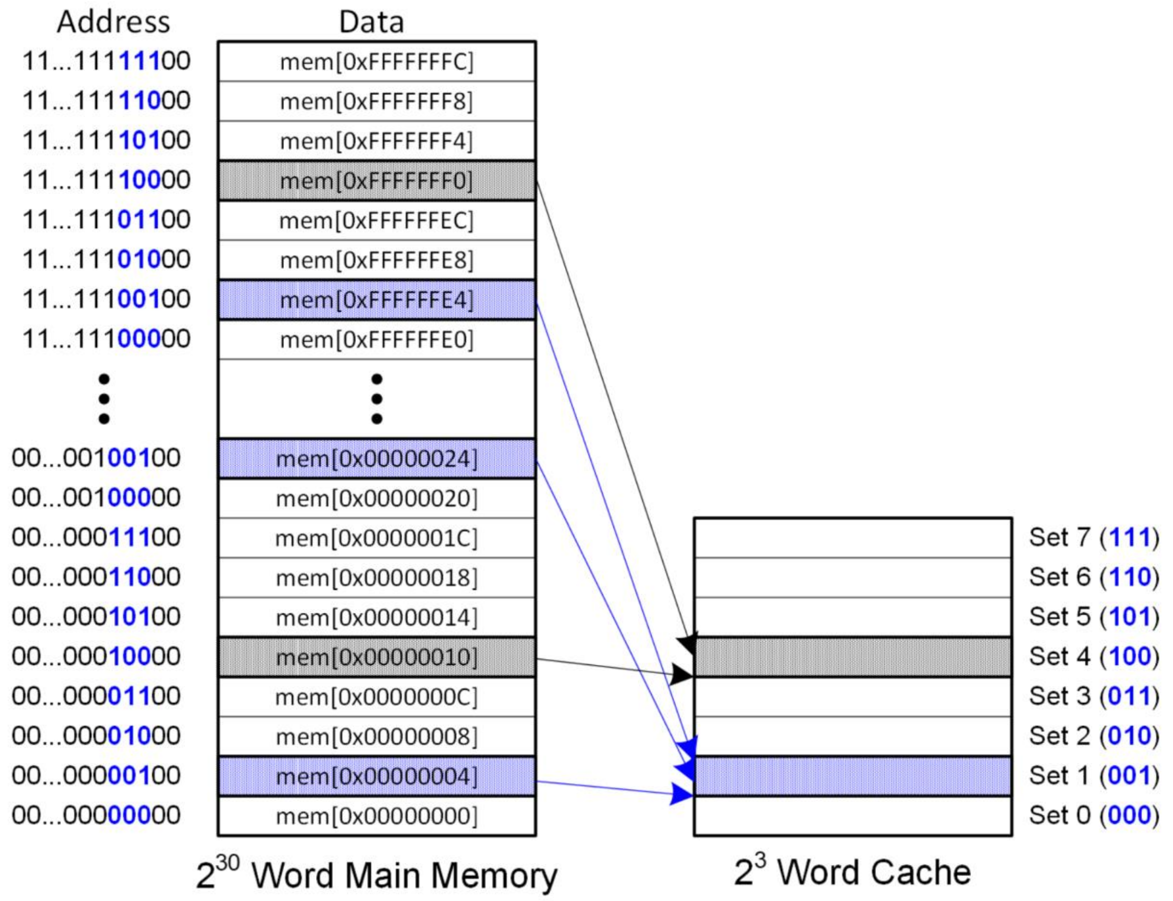
- **Avoid that the entire cache needs to be searched upon access**
    - Cache sets (S): Split the cache into cache sets such that each address of the main memory maps to exactly one set of the cache
    - In the simplest case $S = B$. In this case each storage location for a block in the cache corresponds to a different cache set. This is called a “directly mapped cache”.

Directly Mapped Cache

- The mapping of addresses to cache sets needs to be efficient → we use bits of the address as index for the cache set

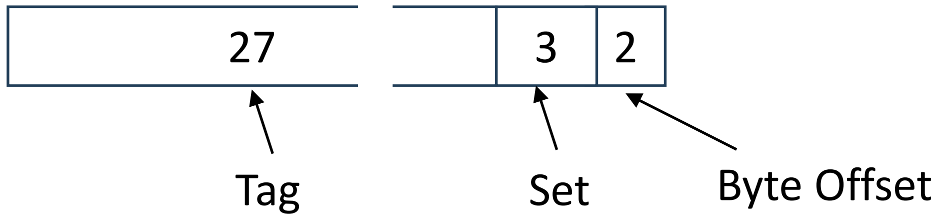
- Example:
  - 1 block = 1 word
  - Cache with 8 blocks
  - The 32-bit address is therefore interpreted as follows



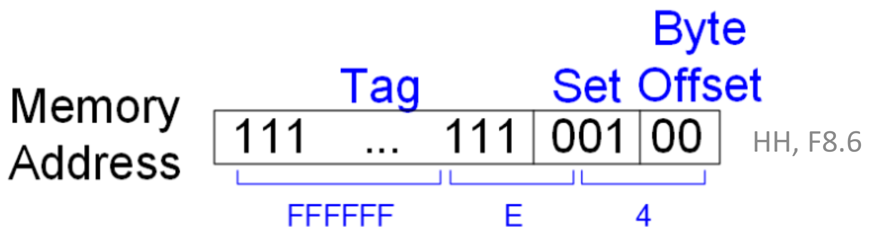


HH, F8.6

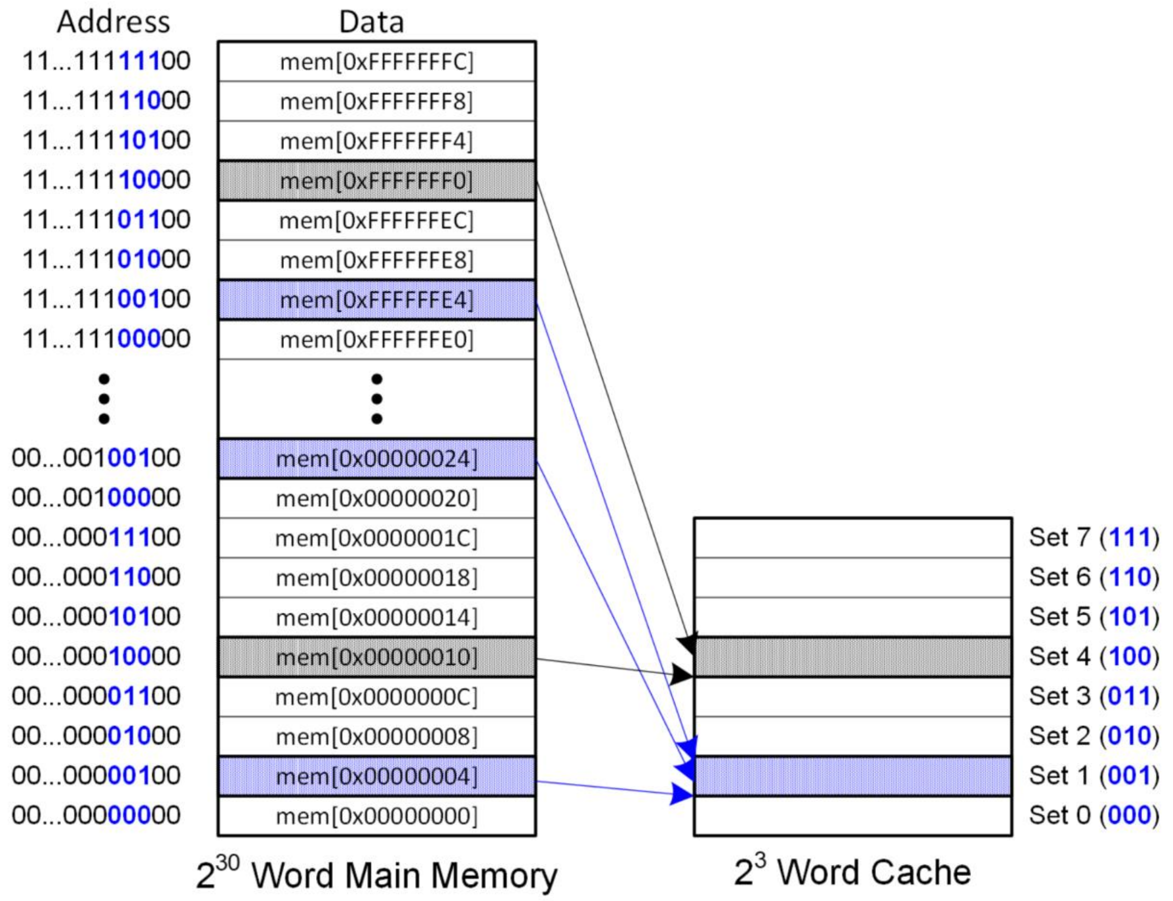
Directly Mapped Cache



Example mapping for address 0xFFFFFFE4:



This address maps to set 001



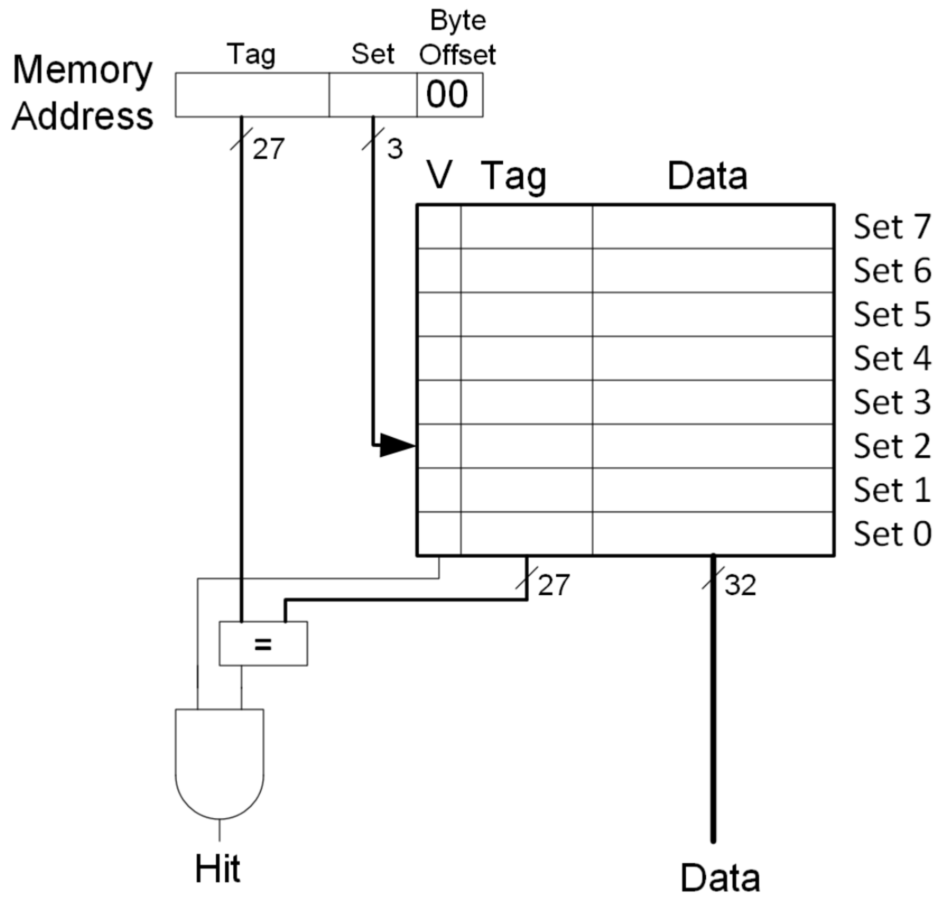
HH, F8.5

Implementation of Directly Mapped Caches

- The cache stores the triplet (valid bit, tag, data) for each entry

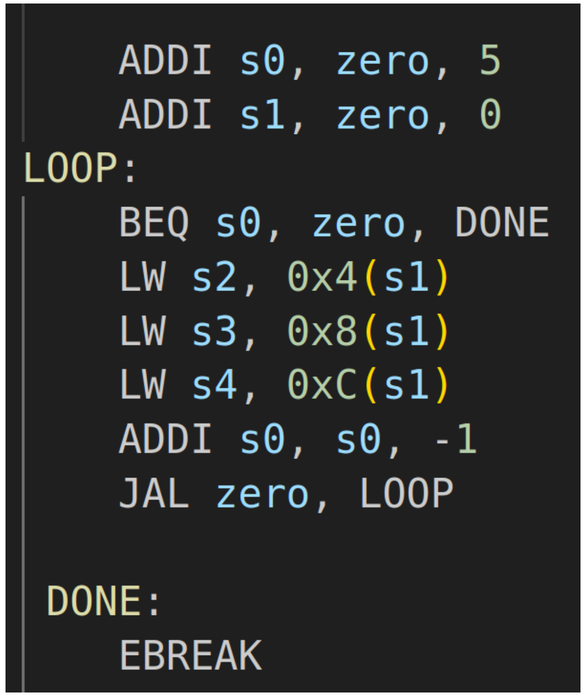
- Read operation:
  - The 3 bits of the set field of the address are used as index to read (valid bit, tag, data) from the cache memory

  - In case of a hit, the data is delivered to the CPU.

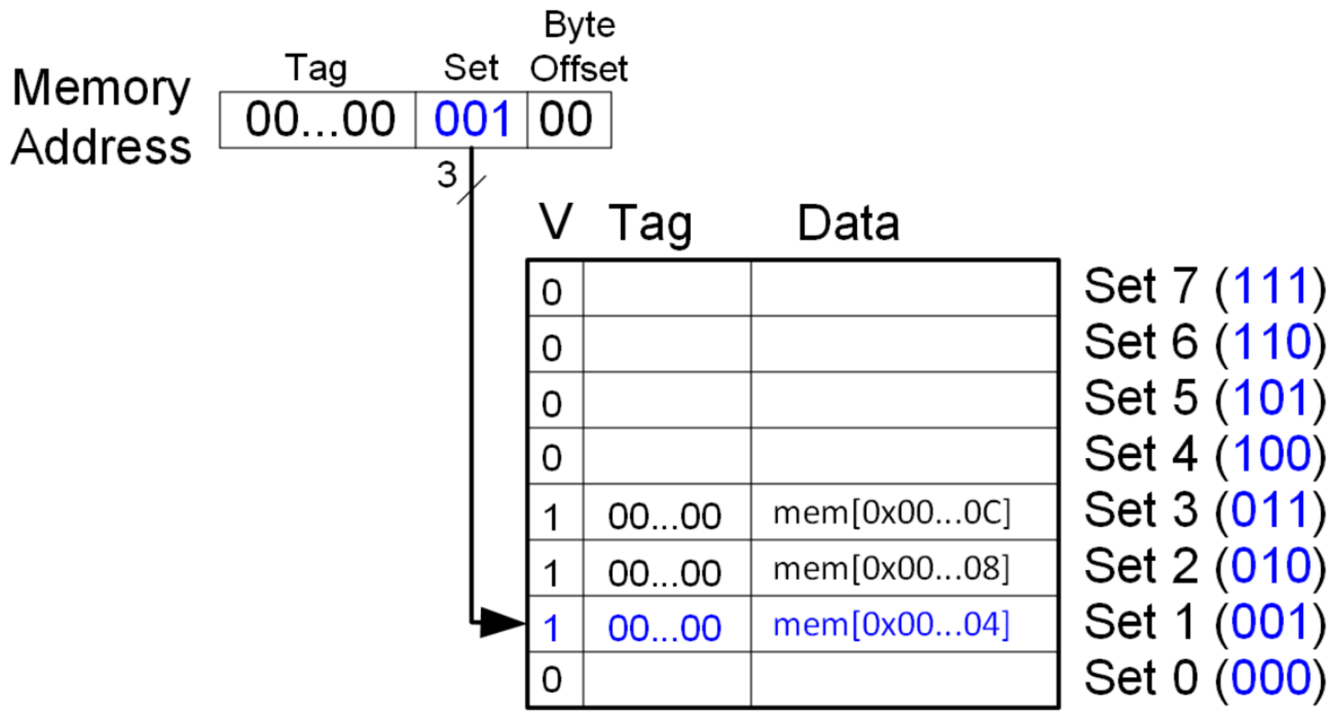


HH, F8.7

Performance of a Directly Mapped Cache



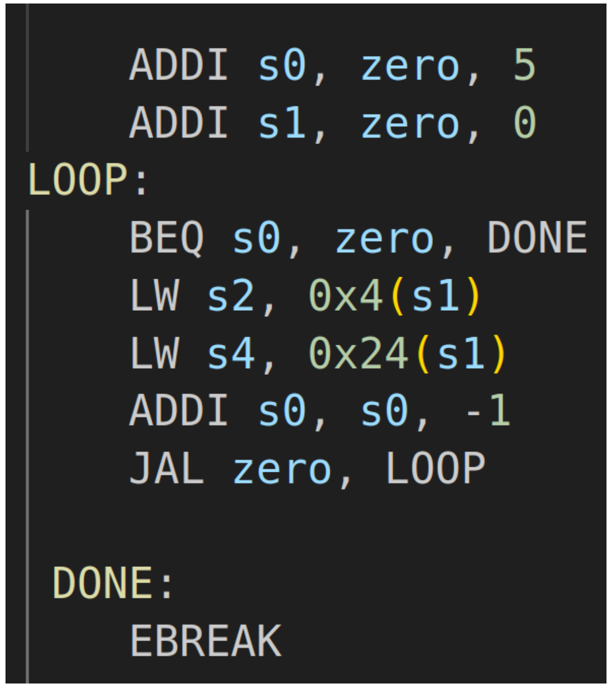
```
    ADDI s0, zero, 5
    ADDI s1, zero, 0
LOOP:
    BEQ s0, zero, DONE
    LW s2, 0x4(s1)
    LW s3, 0x8(s1)
    LW s4, 0xC(s1)
    ADDI s0, s0, -1
    JAL zero, LOOP

 DONE:
    EBREAK
```



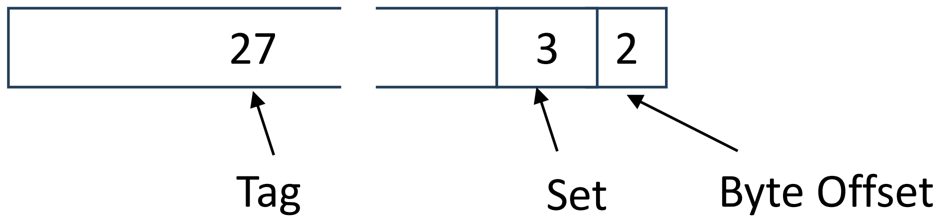
HH, F8.8

- During the first loop execution, all loads lead to a miss
- During the four subsequent executions, the data is in the cache
- Statistics: 15 loads in total, 3 miss, 12 hit → 12/15 = 80% hit ratio

Performance of a Directly Mapped Cache



```
    ADDI s0, zero, 5
    ADDI s1, zero, 0
LOOP:
    BEQ s0, zero, DONE
    LW s2, 0x4(s1)
    LW s4, 0x24(s1)
    ADDI s0, s0, -1
    JAL zero, LOOP

 DONE:
    EBREAK
```



Note: Address 0x4 and 0x24 both map to cache set 1:

0x4: 000 001 00
0x24: 001 001 00

- During the first loop execution, all loads lead to a miss
- Also during subsequent executions, there is no hit
- Statistics: 10 loads in total, 10 miss, 0 hit → 0% hit ratio

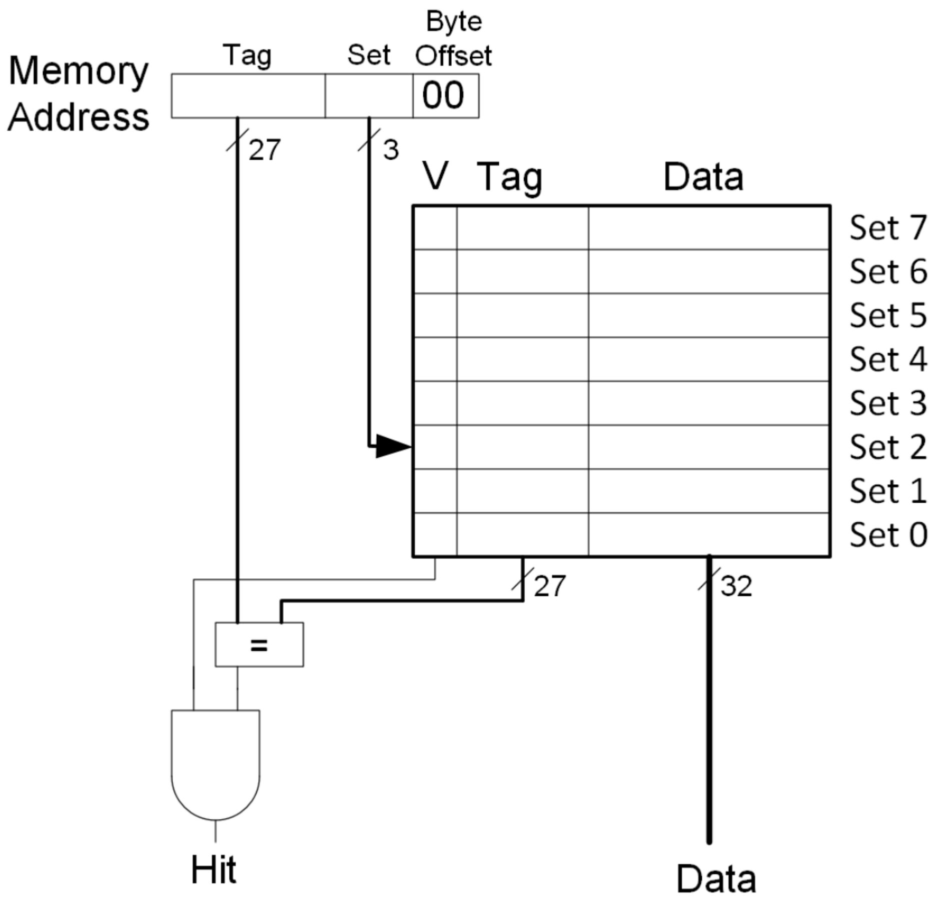
Drawback of Directly Mapped Caches

- There is only one storage location in the cache for all memory elements that map to the same cache set
  → two blocks that map to the same set cannot be in the cache at the same time.

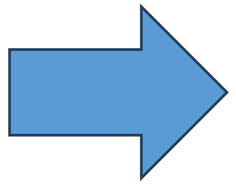
- This can lead to low hit rates (even 0% in case of alternating accesses to addresses mapping to the same cache set)

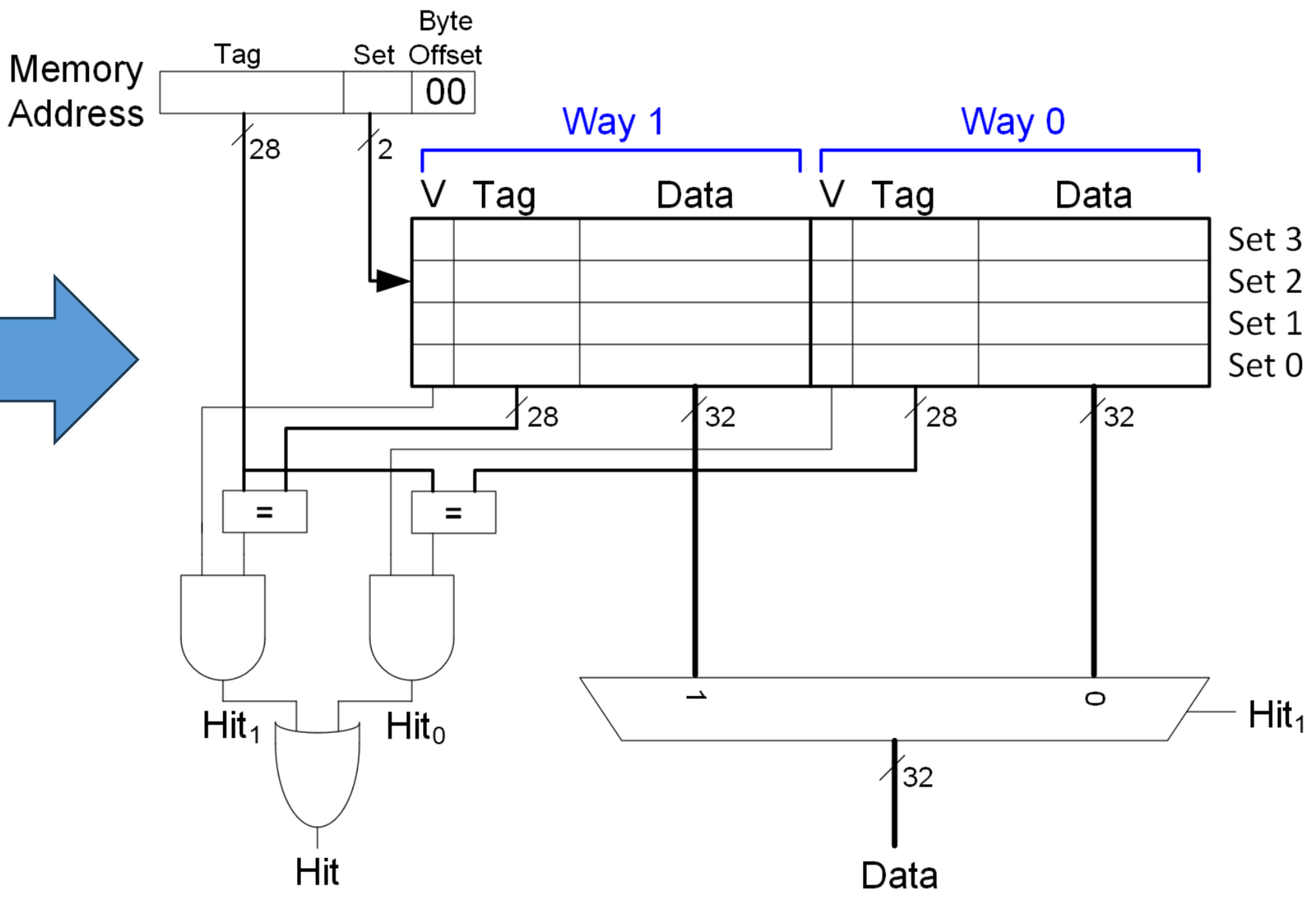
→ We introduce “Set Associative Caches”, which provide multiple storage locations per set

We “Reshape” the Cache to “Two Caches With Half the Size”



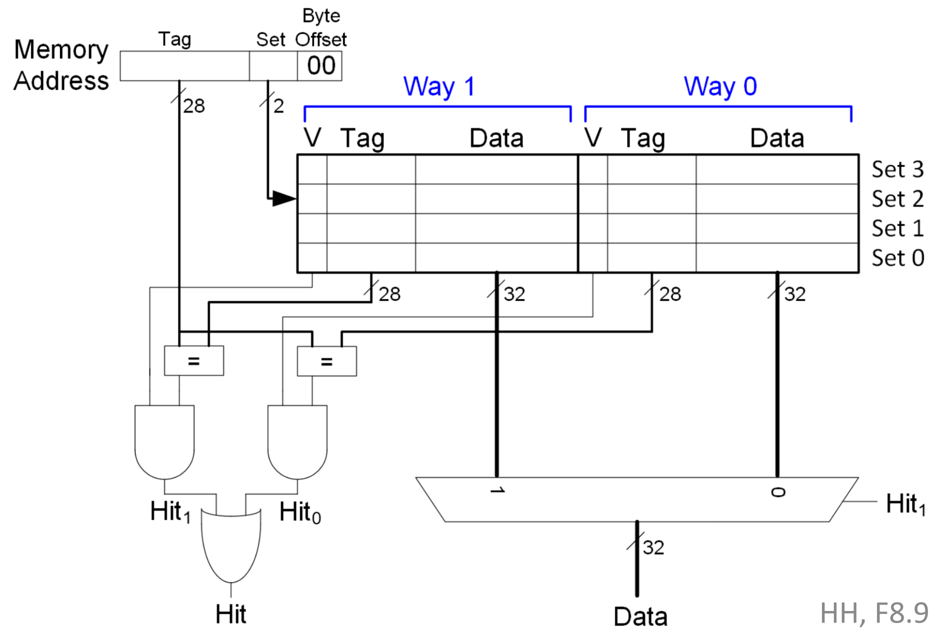
HH, F8.7





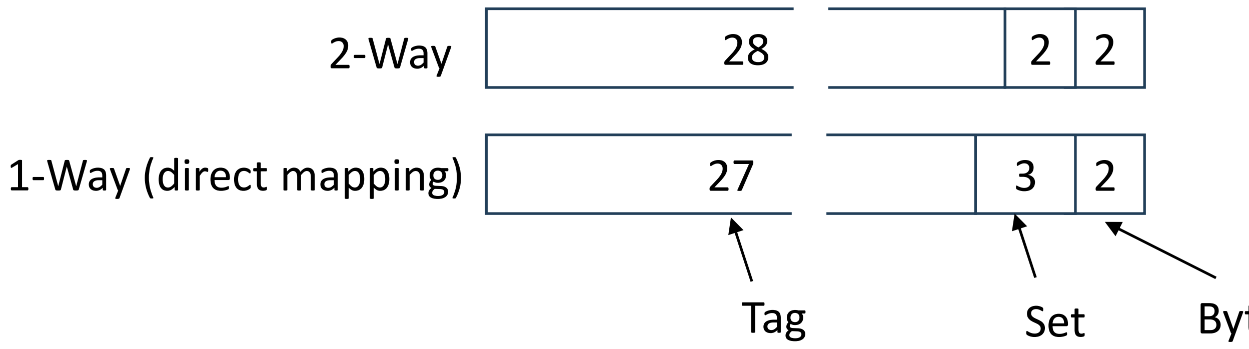
HH, F8.9

Set Associative Caches

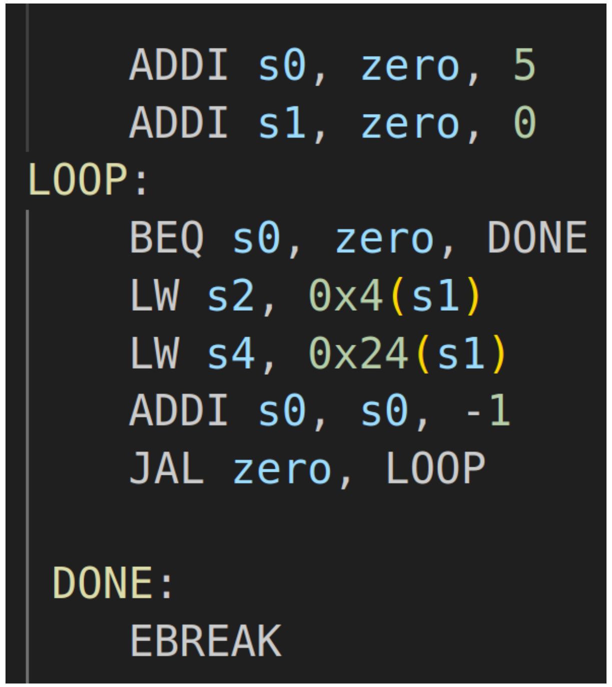


- An N-Way set associative cache, provides N storage locations for each set.
- Each storage location is called "way"
- Upon access, the hardware searches in all ways for the cached data
- Example cache:
  - Number of blocks (B): B = 8
  - Ways (N): N = 2
  - Sets (S): S = 4
  - Each block stores one word

Note the difference to the direct mapping: We now have only 4 sets
→ 2 instead of 3 bits for set indexing; tag size 28 instead of 27

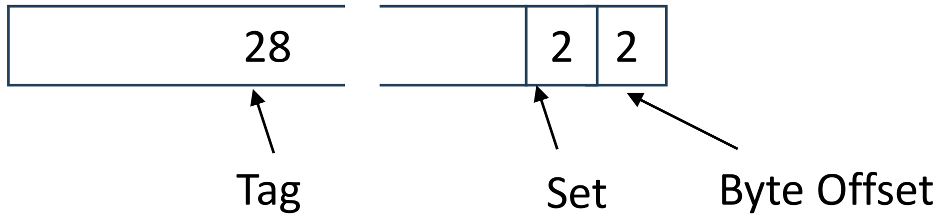


Performance – Repetition of Example



```
    ADDI s0, zero, 5
    ADDI s1, zero, 0
LOOP:
    BEQ s0, zero, DONE
    LW s2, 0x4(s1)
    LW s4, 0x24(s1)
    ADDI s0, s0, -1
    JAL zero, LOOP

 DONE:
    EBREAK
```

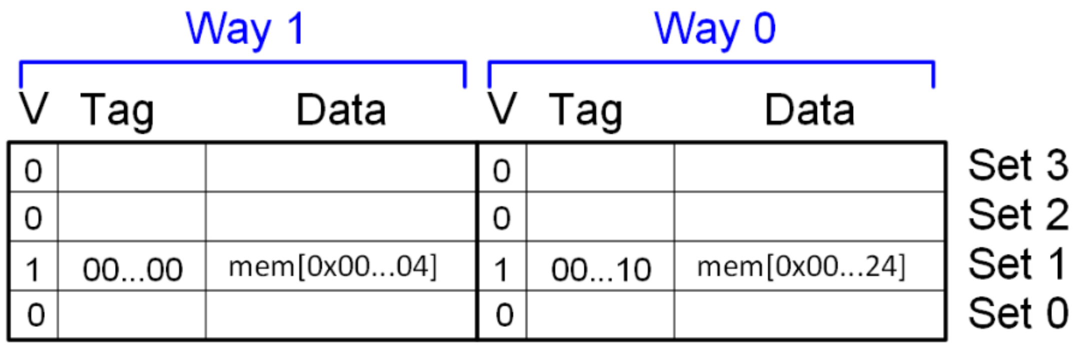
After the first loop iteration, all loads lead to a cache hit



Address 0x4 and 0x24 still map to cache set 1:

0x4: 0000 01 00
0x24: 0010 01 00

However, we now have two ways for storage:

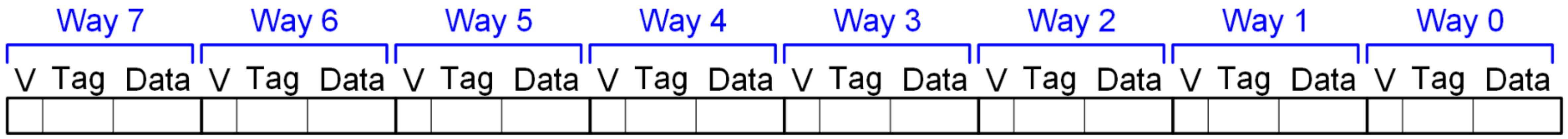


Way 1 V	Way 1 Tag	Way 1 Data	Way 0 V	Way 0 Tag	Way 0 Data	
0			0			Set 3
0			0			Set 2
1	00...00	mem[0x00...04]	1	00...10	mem[0x00...24]	Set 1
0			0			Set 0

HH, F8.10

A Fully Associative Cache

- If a cache consists of a single set with B ways, this is called an “fully associative cache”. It holds N = B, where B is the number of blocks in the cache
- In a fully associated cache every address can be cached at every location
- Fully associative caches are typically only done for small cache sizes (higher number of ways → higher number of comparators → higher power consumption and higher latency)

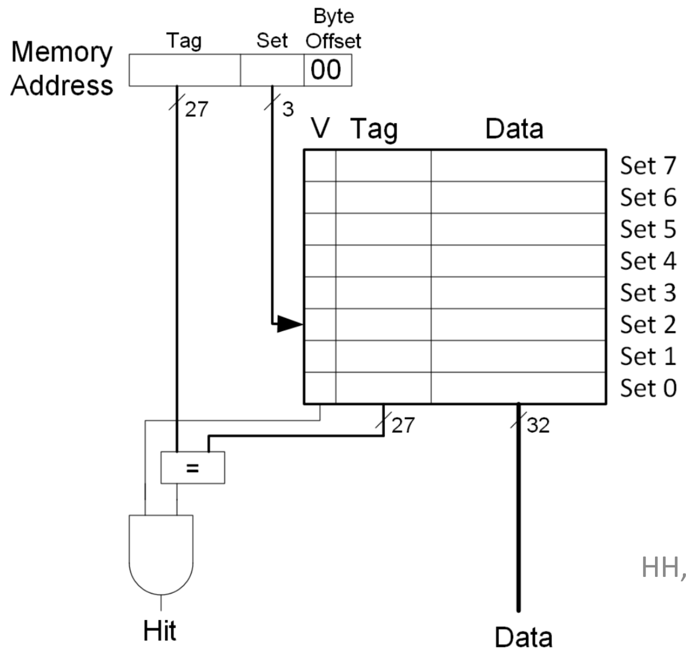


HH, F8.11

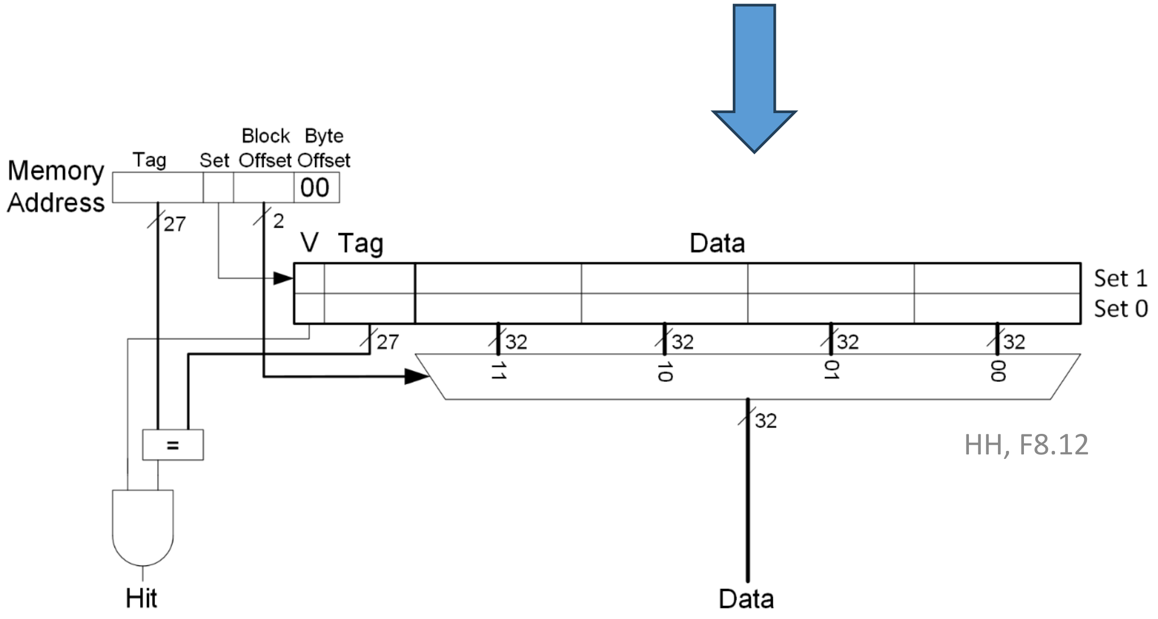
Changing the Block Size (b)

- Motivated by spatial locality, increasing the block size is another parameter to improve the hit ratio

- Example cache:
  - Block size (b): b = 4 words
  - Total number of blocks (B): B = 8 words
  - Ways (N): N = 1
  - Sets (S): S = 2

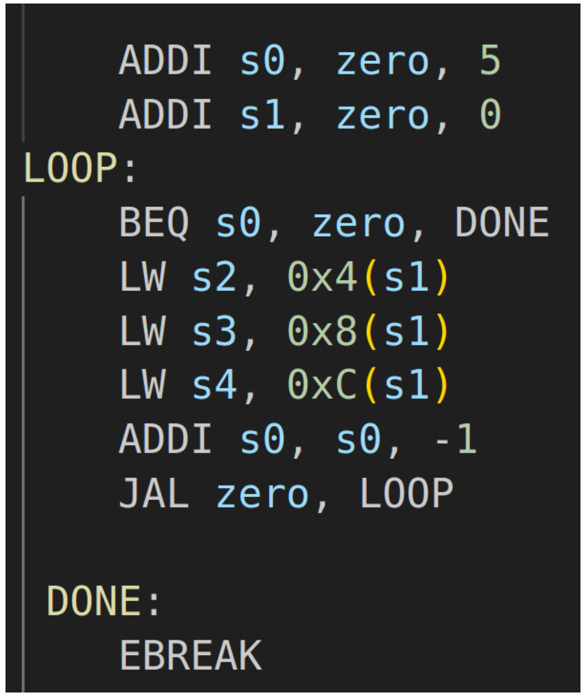


HH, F8.7

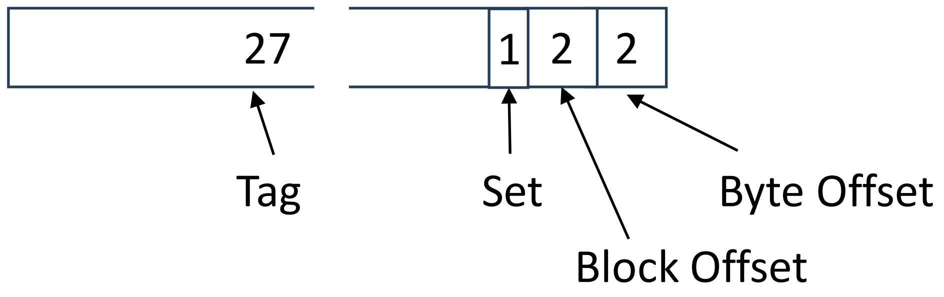


HH, F8.12

Increased Block Size – Performance Example



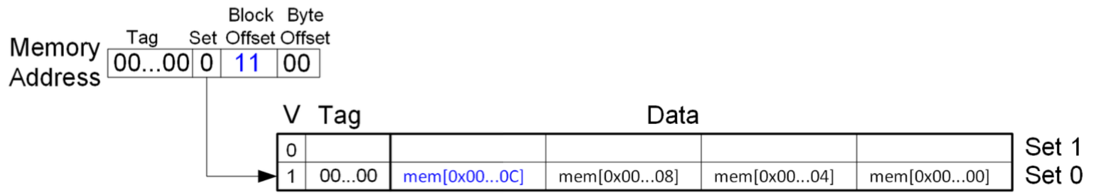
```
    ADDI s0, zero, 5
    ADDI s1, zero, 0
LOOP:
    BEQ s0, zero, DONE
    LW s2, 0x4(s1)
    LW s3, 0x8(s1)
    LW s4, 0xC(s1)
    ADDI s0, s0, -1
    JAL zero, LOOP

 DONE:
    EBREAK
```



Address 0x4, 0x8, and 0xC map to cache set 0:

0x4: 000 0 01 00
0x8: 000 0 10 00
0xC: 001 0 11 00

However, there is no eviction as the three words are in the same block:



HH, F8.14

After the first loop iteration, all loads lead to a cache hit

Summary of Parameters for Different Cache Organizations

We define a cache through

- Capacity (C)
- Block size (b) – also called length of a cache line
- Number of ways (N)

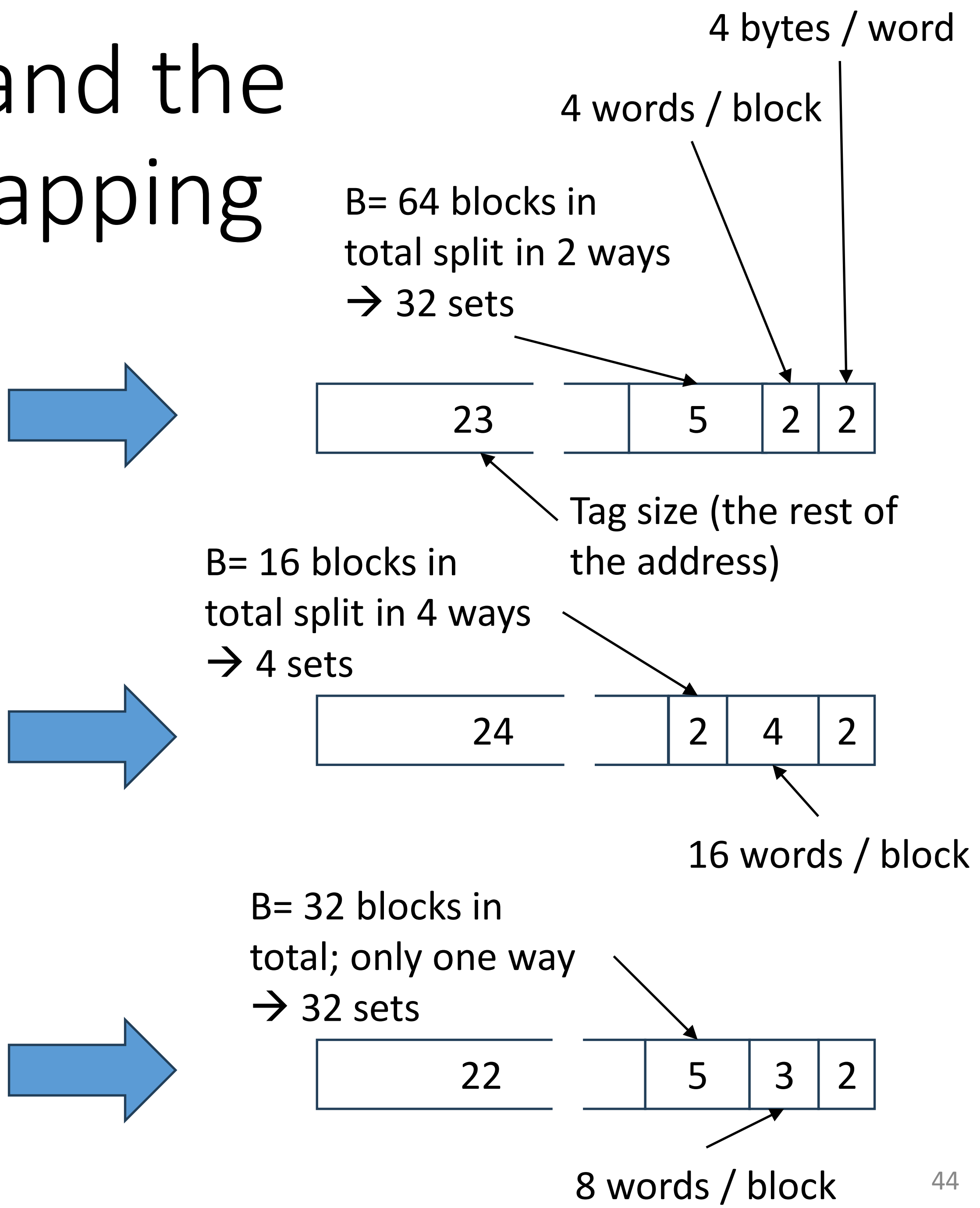
- The number of blocks (B) in in a cache is given by B = (C / b)

- The parameters are selected such that the number of sets (S=B/N) is a power of two

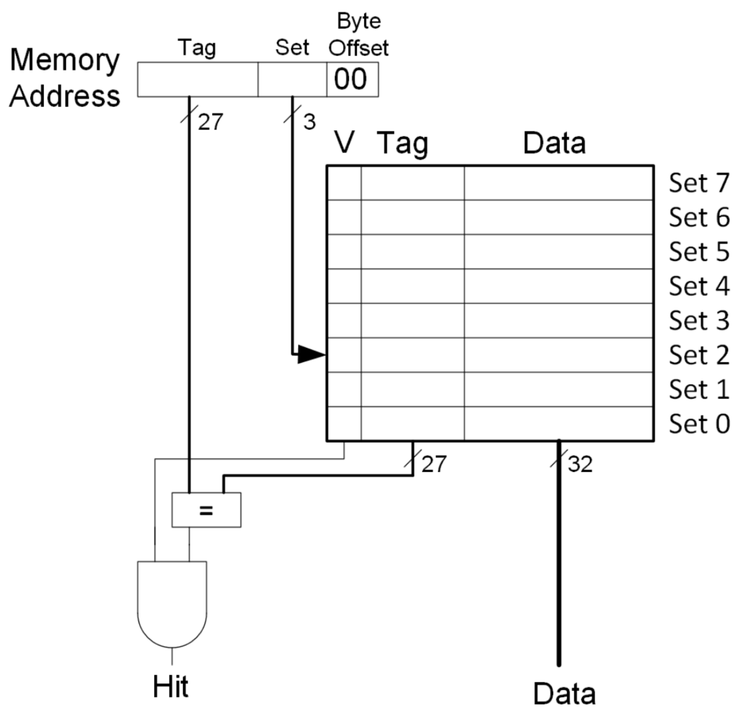
Cache Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapping	1	B
Set Associative	1 < N < B	B/N
Fully Associative	B	1

Examples of Caches Sizes and the Corresponding Address Mapping

- Example 1 (1024 byte Capacity)
  - Block size (b): 16 byte
  - Ways (N): 2 ways
- Example 2 (1024 byte Capacity)
  - Block size (b): 64 byte
  - Ways (N): 4 ways
- Example 3 (1024 byte Capacity)
  - Block size (b): 32 byte
  - Ways (N): 1 way


4 bytes / word
4 words / block
B= 64 blocks in total split in 2 ways → 32 sets
23
5
2
2
Tag size (the rest of the address)
B= 16 blocks in total split in 4 ways → 4 sets
24
2
4
2
16 words / block
B= 32 blocks in total; only one way → 32 sets
22
5
3
2
8 words / block

Options When "Doubling the Cache Size"



HH, F8.7

- Assume you have the simple cache we had at the beginning
- Assume we can afford the cost to double the size of data storage

- **Design Options:**
  - Double the number of sets (Cache stays a 1-way cache with same block size) → +1 bit in set selection
  - Double the block size (Cache stays a one 1-way cache with same number of sets) → +1 bit for Block offset
  - Double the number of ways (Number of sets and block size stays) → no change in the address decoding

Replacement Policies

Which Block to Replace?

- In case of a cache miss, a new block needs to be stored in the cache
- In case, there are invalid blocks, use these blocks first
- In case all are valid, one block needs to be evicted. There are several possible replacement policies:
  - Random
  - FIFO
  - Least recently used
  - Not most recently used
  - Least frequently used?
  - …

Implementing an LRU (Least Recently Used) Policy

- Goal
  - Evict the block that was least recently accessed
- 2-way set associative cache
  - Add one bit of metadata to indicate which way has been access least recently
- N-way set associative cache
  - Implementing LRU perfectly is typically too expensive to be implemented for caches with 4+ ways
  - Note: LRU is an approximation to predict locality and not necessarily the best possible replacement policy anyway

Alternatives Policies

- Random
  - Just replace a block randomly
- Pseudo-LRU
  - Split the ways of a cache into two groups and track which of the two groups has been used most recently
  - Upon eviction select a random block from the group that was least recently used
- **Note**
  - The hit rate in practice strongly depends on the executed code
  - Random replacement policies also lead to good hit rates as other replacement strategies are also not perfect (e.g. when the program is working on more memory that fits into the cache, there is continuous eviction)

Handling Write Opertions

Two Options for Write Handling

- **Write-Back Cache**
  - **Idea**
    - When data is written to memory, update only the cache and do not update further up in the memory hierarchy
    - Write to the next cache level when the cache line is evicted
  - **Pros**
    - In case of multiple writes to the same block, this is more efficient
  - **Cons**
    - Needs a “dirty bit” in the cache to indicate whether block has been written to or not
    - More complex design

- **Write-Through Cache**
  - **Idea**
    - Update the value in the cache and update the next level of the memory hierarchy
  - **Pros**
    - Simpler design
  - **Cons**
    - No combination of writes
    - More transfers between the memory

Two Options for Allocation

- **Allocate On Miss**
  - **Idea**
    - Transfer a memory block into the cache, if there is a write on the block
  - **Pros**
    - Can combine writes
    - Simpler design (read and write have the same behavior)
  - **Cons**
    - Can lead to more memory transfers

- **No Allocation on Miss**
  - **Idea**
    - Don’t transfer a block into main memory upon write
  - **Pros**
    - Uses less cache space
  - **Cons**
    - No combination of writes

Which Cache to Use Where?

Which Type of Cache to Use Where in the the Memory Hierarchy?

- **Options for Cache Organization**
  - Cache Size
  - Associativity
  - Block Size

- **Additional Options for Implementation**
  - Replacement policy
  - Write Handling (write back vs. write through)
  - Separate cache for instructions and data vs. unified cache
  - Exclusive vs. inclusive caches (in case of exclusive caches, data is not duplicated across cache layers – implies more complex cache management)
  - Separate implementation of tags and data in a tag memory and a data memory
  - …

Goal

The goal of the overall memory system is to minimize the average memory access time (AMAT)

→ minimize cache latency and minimize the number of cache misses at each level

Cache Misses

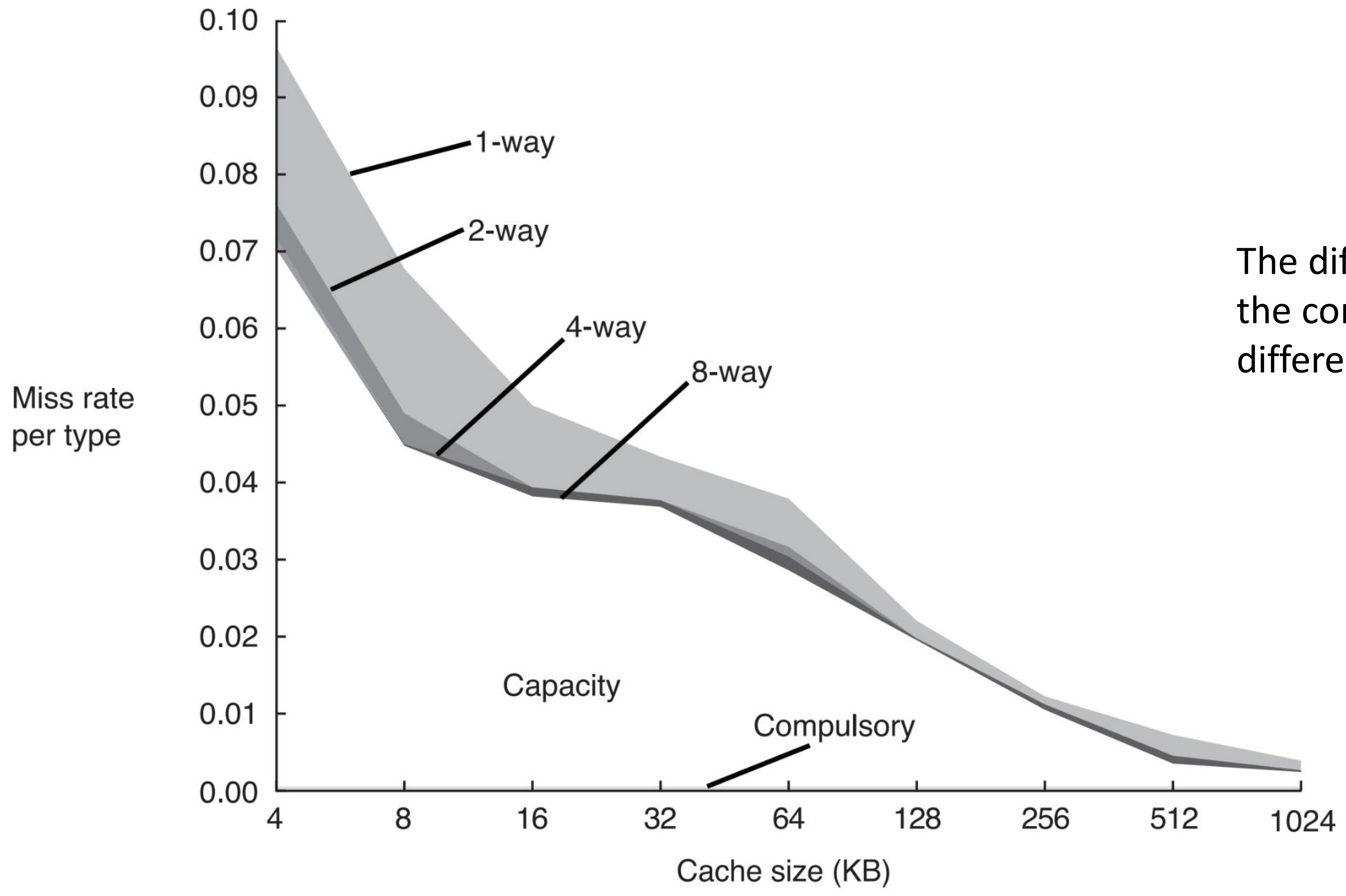
Reasons for Cache Misses

- **Compulsory Misses:** Cache misses that occur independent of the cache design
- **Capacity Miss:** Cache misses that occur because the cache can’t store all needed data concurrently
- **Conflict Misses:** Cache misses that occur because different addresses map to the same set and that then evict blocks that are still needed by a program

Effect of Parameters

- Increasing the block size can reduce compulsory misses (spatial locality), but can increase the conflict misses
- Increasing the capacity can decrease capacity and conflict misses, but not compulsory misses
- …

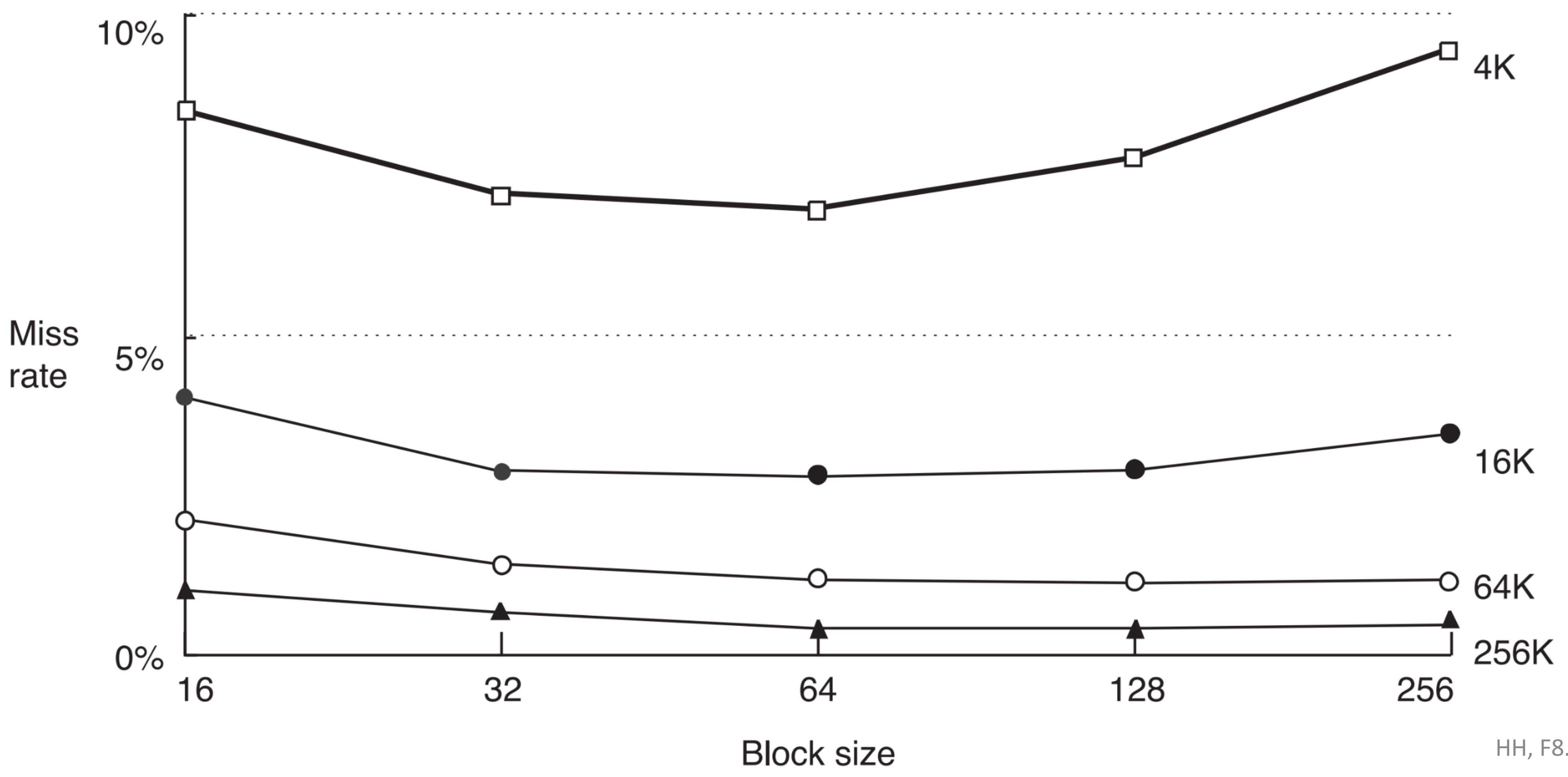
Effect of Associativity



The different gray parts show the conflict misses for the different levels of associativity

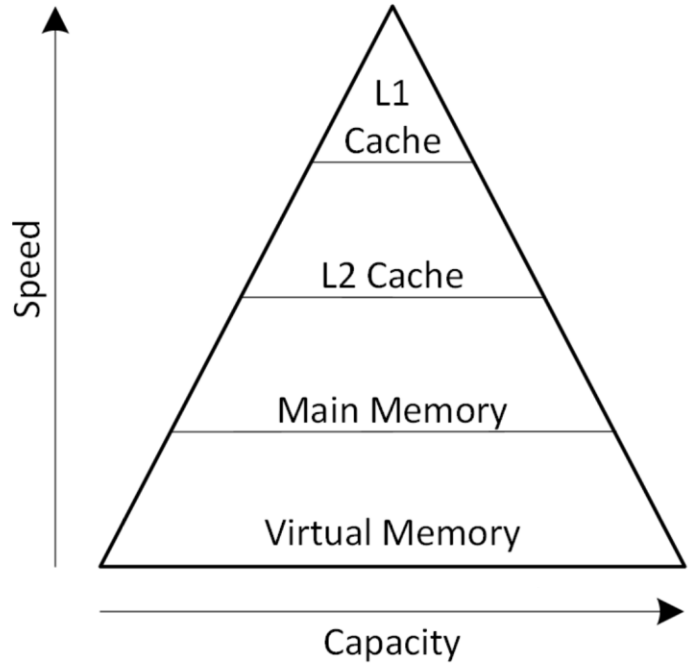
HH, F8.17

Effect of Block Size



HH, F8.18

General Considerations



HH, F8.16

- **L1 Cache**
  - This is the point where latency matters the most latency overrules all other properties
  - Latency needs to be aligned with clock rate of CPU
  - On most systems, there is a separate data and instruction cache (avoiding mutual eviction of data and instructions)

- **L2 and higher levels**
  - Typically unified caches (data and code in the same cache)
  - Latency less and less dominant with each level (allows larger size, allows doing tag access and data access sequentially, …)

Product Examples

- **Intel (13th Generation)**
  - L1 (P cores): 12-way 48KB for data; 8-way 32KB for instructions
  - L1 (E cores): 8-way cache; 32KB for data; 64KB for instructions
  - L2 (P cores): 10-way non-inclusive cache; 1.25MB
  - L2 (E cores): 16-way non-inclusive cache; 2MB
  - L3: 12-way non-inclusive cache ; up to 3MB per core (shared between all cores)
  - Lenth of cache line: 64 bytes

- **Apple M2**
  - L1 (P cores): 192KB instruction and 128 KB data cache
  - L1 (E cores): 128KB instruction and 64 KB data cache
  - L2: 16MB (shared)
  - L3: 8 MB – 96 MB (shared)

Notation - P cores: optimized for performance; E cores: optimized for energy efficiency

Cache Coherency

- In multiprocessor systems, at least the first-level cache is not shared between cores → This can lead to coherence problems: processor 1 reads value A; processor 2 reads value A; processor 1 writes value A; processor 2 reads value A (from its L1 cache)

- It is necessary to ensure **cache coherence**
  - A read by a processor A from a location X that follows a write by a processor A to location X returns the value that was written by processor A in case there was no write on location X by another processor

  - A read by a processor A from a location X that follows a write to location X by processor B (with no writes made by any other processor), must return the value written by processor B (given that there has been sufficient time between the write and the read).

  - Writes to the same memory location are seen in the same order by all processors (Serialization)

Examples of Cache Access Patterns

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array is stored on the stack at memory location 0x36; all other variables are in registers;
- Assume an empty stack the beginning of the execution of the loop

```
char array[4] = {1, 2, 3, 4};
int sum = 0;
int prod = 1;

for(int i = 0; i < 4; ++i)
        sum = sum + array[i];
for(int i = 0; i < 4; ++i)
         prod = prod * array[i];
```

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array is stored on the stack at memory location 0x36; all other variables are in registers;
- Assume an empty stack the beginning of the execution of the loop

```
char array[4] = {1, 2, 3, 4};
int sum = 0;
int prod = 1;

for(int i = 0; i < 4; ++i)
        sum = sum + array[i];
for(int i = 0; i < 4; ++i)
         prod = prod * array[i];
```

Addresses for memory accesses:

Hex	Binary
0x36	00 110 110
0x37	00 110 111
0x38	00 111 000
0x39	00 111 001
0x36	00 110 110
0x37	00 110 111
0x38	00 111 000
0x39	00 111 001

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array is stored on the stack at memory location 0x36; all other variables are in registers;
- Assume an empty stack the beginning of the execution of the loop

```
char array[4] = {1, 2, 3, 4};
int sum = 0;
int prod = 1;

for(int i = 0; i < 4; ++i)
        sum = sum + array[i];
for(int i = 0; i < 4; ++i)
         prod = prod * array[i];
```

Tag

Block index

Offset

Addresses for memory accesses:

Hex	Binary
0x36	00 110 110
0x37	00 110 111
0x38	00 111 000
0x39	00 111 001
0x36	00 110 110
0x37	00 110 111
0x38	00 111 000
0x39	00 111 001

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array is stored on the stack at memory location 0x36; all other variables are in registers;
- Assume an empty stack the beginning of the execution of the loop

```
char array[4] = {1, 2, 3, 4};
int sum = 0;
int prod = 1;

for(int i = 0; i < 4; ++i)
        sum = sum + array[i];
for(int i = 0; i < 4; ++i)
         prod = prod * array[i];
```

Addresses for memory accesses:

Hex	Binary	
0x36	00 110 110	M
0x37	00 110 111	
0x38	00 111 000	
0x39	00 111 001	
0x36	00 110 110	
0x37	00 110 111	
0x38	00 111 000	
0x39	00 111 001	

Cache Miss. The block is copied from memory into the data cache. The tag memory stores 00 at index 110.

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array is stored on the stack at memory location 0x36; all other variables are in registers;
- Assume an empty stack the beginning of the execution of the loop

```
char array[4] = {1, 2, 3, 4};
int sum = 0;
int prod = 1;

for(int i = 0; i < 4; ++i)
        sum = sum + array[i];
for(int i = 0; i < 4; ++i)
         prod = prod * array[i];
```

Addresses for memory accesses:

Hex	Binary	
0x36	00 110 110	M
0x37	00 110 111	H
0x38	00 111 000	
0x39	00 111 001	
0x36	00 110 110	
0x37	00 110 111	
0x38	00 111 000	
0x39	00 111 001	

We access the same block as before → Cache Hit

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array is stored on the stack at memory location 0x36; all other variables are in registers;
- Assume an empty stack the beginning of the execution of the loop

```
char array[4] = {1, 2, 3, 4};
int sum = 0;
int prod = 1;

for(int i = 0; i < 4; ++i)
        sum = sum + array[i];
for(int i = 0; i < 4; ++i)
         prod = prod * array[i];
```

Addresses for memory accesses:

Hex	Binary	
0x36	00 110 110	M
0x37	00 110 111	H
0x38	00 111 000	M
0x39	00 111 001	
0x36	00 110 110	
0x37	00 110 111	
0x38	00 111 000	
0x39	00 111 001	

Access to a new block → cache miss

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array is stored on the stack at memory location 0x36; all other variables are in registers;
- Assume an empty stack the beginning of the execution of the loop

```
char array[4] = {1, 2, 3, 4};
int sum = 0;
int prod = 1;

for(int i = 0; i < 4; ++i)
        sum = sum + array[i];
for(int i = 0; i < 4; ++i)
        prod = prod * array[i];
```

Addresses for memory accesses:

Hex	Binary	
0x36	00 110 110	M
0x37	00 110 111	H
0x38	00 111 000	M
0x39	00 111 001	H
0x36	00 110 110	
0x37	00 110 111	
0x38	00 111 000	
0x39	00 111 001	

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array is stored on the stack at memory location 0x36; all other variables are in registers;
- Assume an empty stack the beginning of the execution of the loop

```
char array[4] = {1, 2, 3, 4};
int sum = 0;
int prod = 1;

for(int i = 0; i < 4; ++i)
        sum = sum + array[i];
for(int i = 0; i < 4; ++i)
        prod = prod * array[i];
```

Addresses for memory accesses:

Hex	Binary	
0x36	00 110 110	M
0x37	00 110 111	H
0x38	00 111 000	M
0x39	00 111 001	H
0x36	00 110 110	H
0x37	00 110 111	H
0x38	00 111 000	H
0x39	00 111 001	H

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array1 is stored at memory location 0x36; array2 is stored at memory location 0x70; all other variables are in registers;
- Assume an empty stack the beginning of the execution

```
char array1[4] = {1, 2, 3, 4};
char array2[4] = {1, 2, 3, 4};
int sum_prod = 0;

for(int i = 0; i < 4; ++i)
        sum_prod = sum_prod + array1[i] * array2[i];
```

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array1 is stored at memory location 0x36; array2 is stored at memory location 0x70; all other variables are in registers;
- Assume an empty stack the beginning of the execution

```
char array1[4] = {1, 2, 3, 4};
char array2[4] = {1, 2, 3, 4};
int sum_prod = 0;

for(int i = 0; i < 4; ++i)
        sum_prod = sum_prod + array1[i] * array2[i];
```

Addresses for memory accesses:

Hex	Binary
0x36	00 110 110
0x70	01 110 000
0x37	00 110 111
0x71	01 110 001
0x38	00 111 000
0x72	01 110 010
0x39	00 111 001
0x73	01 110 011

Example: Cache Access Patterns of C Code

- Assume a 256 bytes of main memory, a directly mapped data cache with 64 bytes, and a block size of 8 byte
- Assume array1 is stored at memory location 0x36; array2 is stored at memory location 0x70; all other variables are in registers;
- Assume an empty stack the beginning of the execution

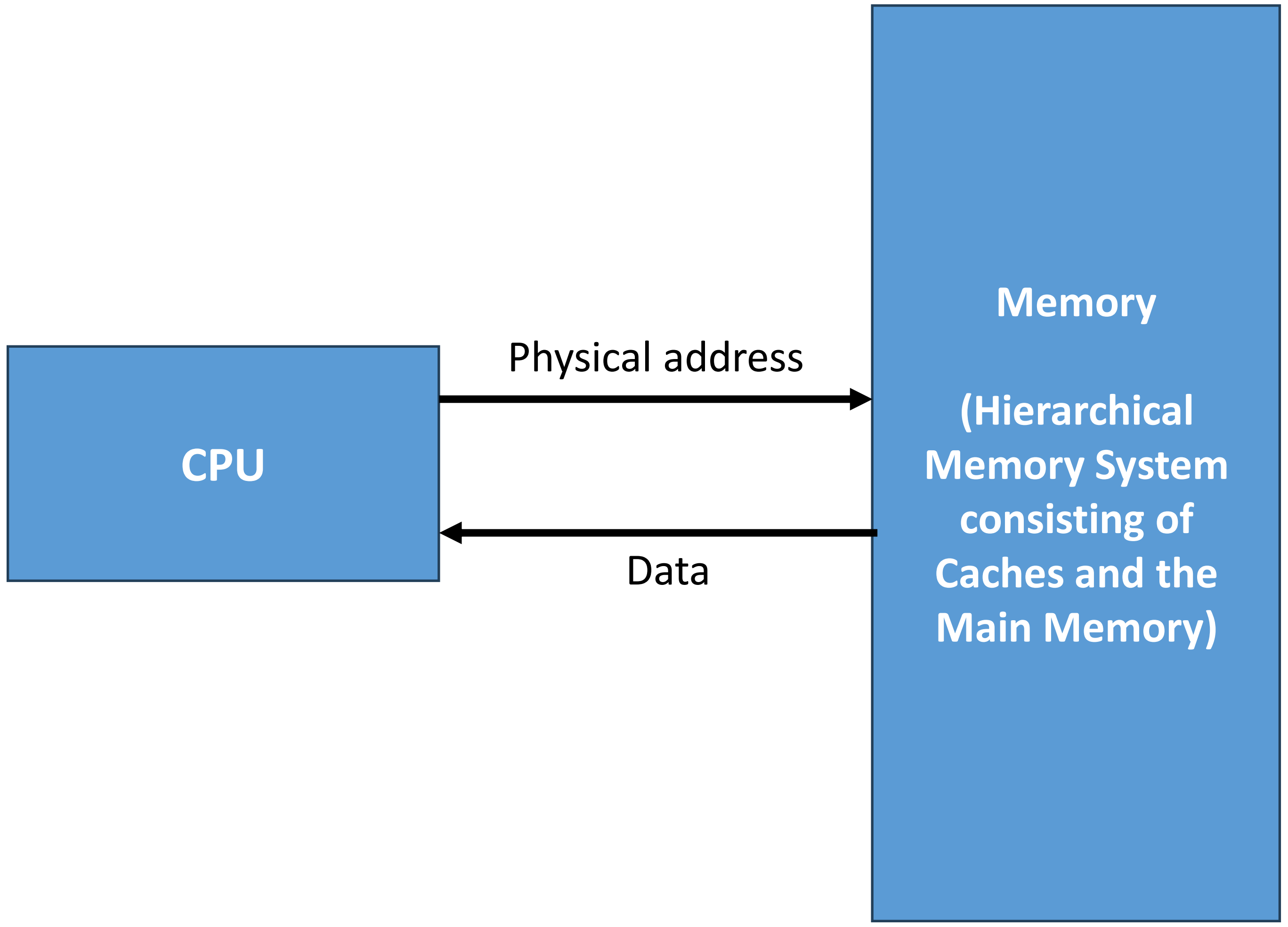
```
char array1[4] = {1, 2, 3, 4};
char array2[4] = {1, 2, 3, 4};
int sum_prod = 0;

for(int i = 0; i < 4; ++i)
        sum_prod = sum_prod + array1[i] * array2[i];
```

Addresses for memory accesses:

Hex	Binary	
0x36	00 110 110	M
0x70	01 110 000	M
0x37	00 110 111	M
0x71	01 110 001	M
0x38	00 111 000	M
0x72	01 110 010	H
0x39	00 111 001	H
0x73	01 110 011	H

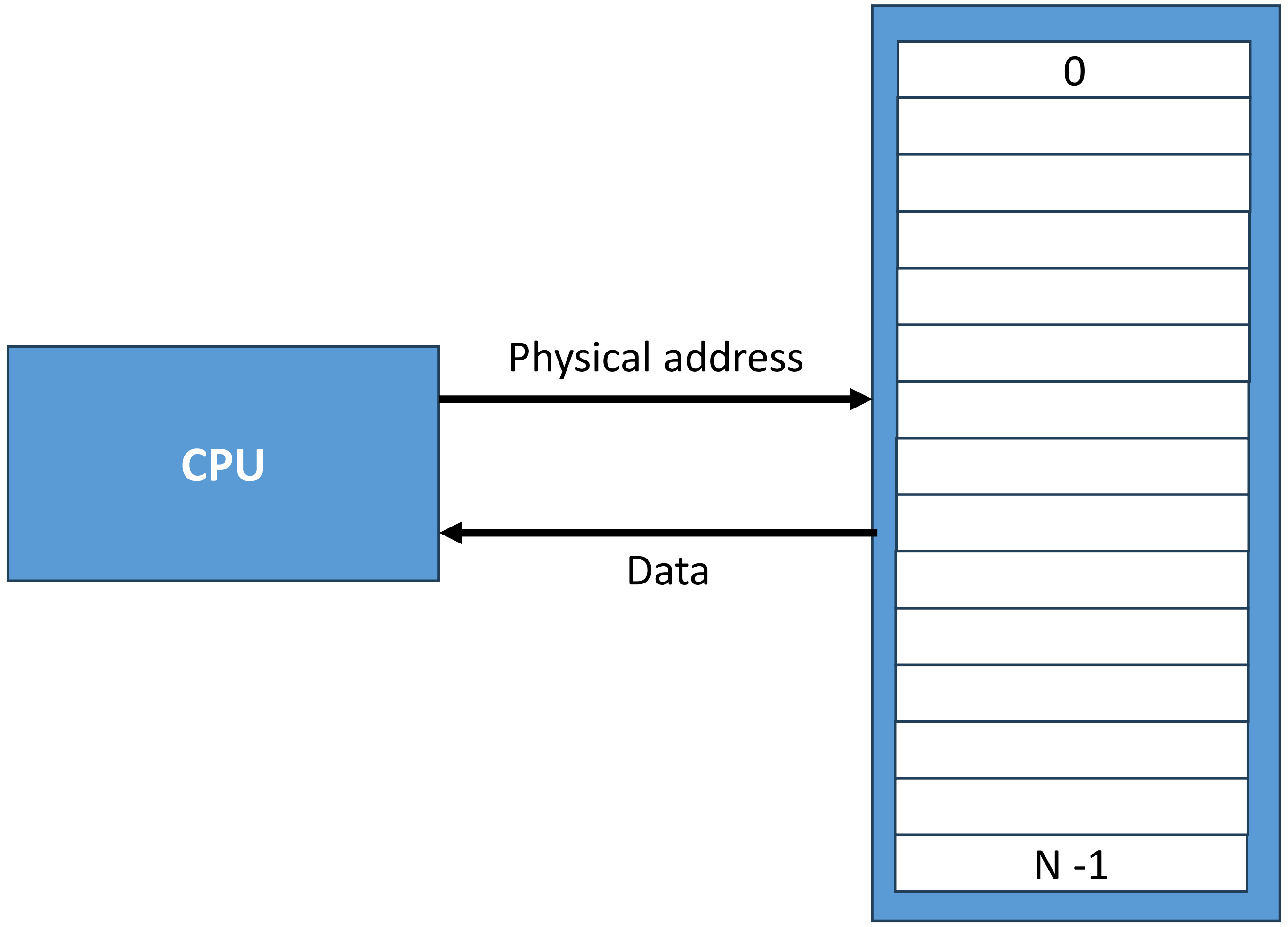
Virtual Memory

Systems Directly Accessing Physical Memory


CPU
Physical address
Data
Memory
(Hierarchical Memory System consisting of Caches and the Main Memory)

Programmer’s View

- The CPU sends a physical address
- The address “runs” through the memory system and finally a data value is returned from the address (location) requested by the CPU



Observations from a Programmer's Perspective

Observe

- The programmer needs to manage the memory layout (i.e. what is where in the memory)
- The memory addresses are a part of the program → changing an address means changing the program

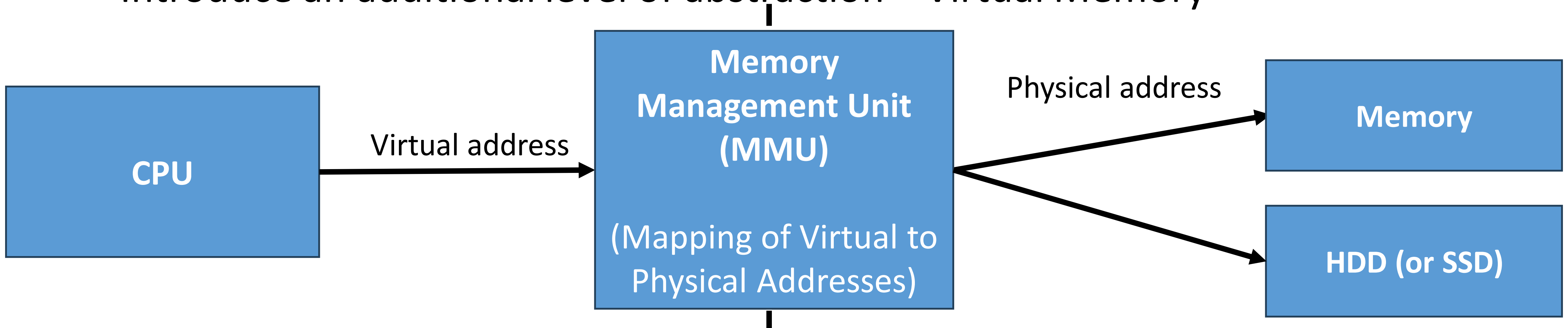
Difficulties

- Difficult to cope with devices with different memory sizes
  - If you extend memory on a device, this may impact the programming
  - If you execute the same program on two different machines with different amounts of memory, this is likely the code will not be located at the same physical location
- Difficult to support code and data relocation
- Difficult to support data/code sharing across different programs
- Difficult to support multiple processes

→ Direct physical memory access is mainly used on small embedded devices

Virtual Memory

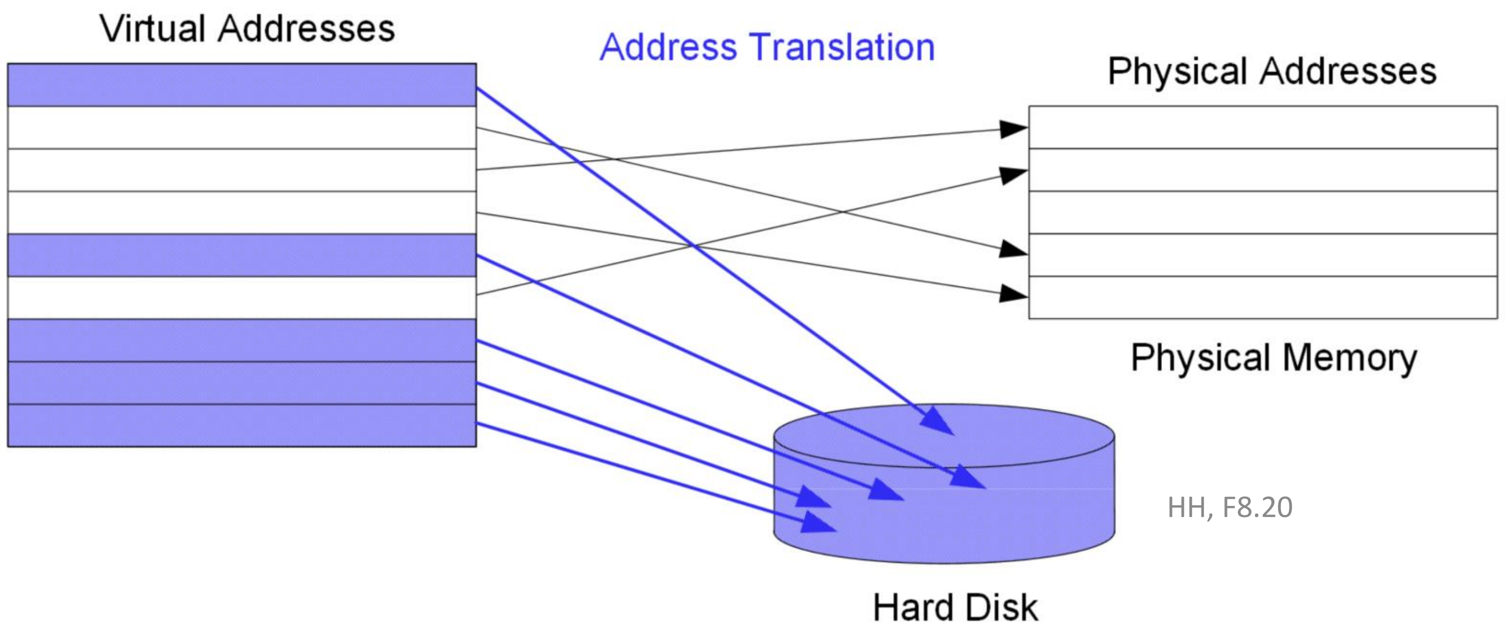
- **Idea**
  - Make the programmer's view of memory independent of the memory that is physically available and independent of physical locations of storage
  - Make memory appear as an almost "infinite resource" to the programmer
- **Basic Concept**
  - Introduce an additional level of abstraction – Virtual Memory


CPU
Virtual address
Memory Management Unit (MMU)
(Mapping of Virtual to Physical Addresses)
Physical address
Memory
HDD (or SSD)

Properties and Benefits

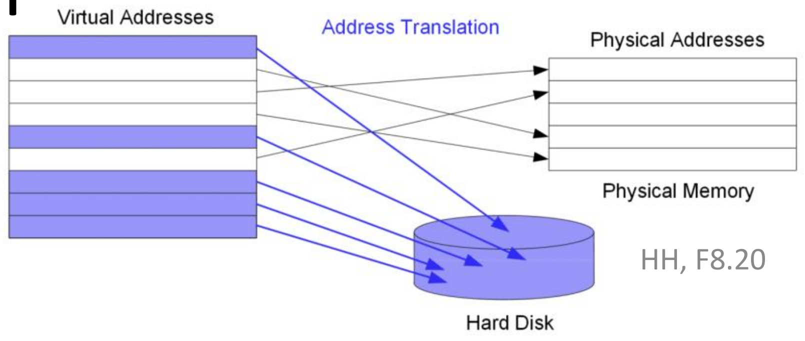
- **Properties**
  - The programmer does not work with physical addresses any more
  - Each process has its own mapping of virtual to physical addresses
  - The HDD is added as an additional memory to the memory hierarchy (slowest, but largest)

- **Benefits**
  - Relocation
  - Sharing of memory between processes
  - Isolation of processes



HH, F8.20

Basics On the Address Translation



HH, F8.20

- Address translation is done at the level of pages
  - Virtual memory is divided into virtual pages
  - Virtual pages are mapped to physical pages (also called frames) via the page table

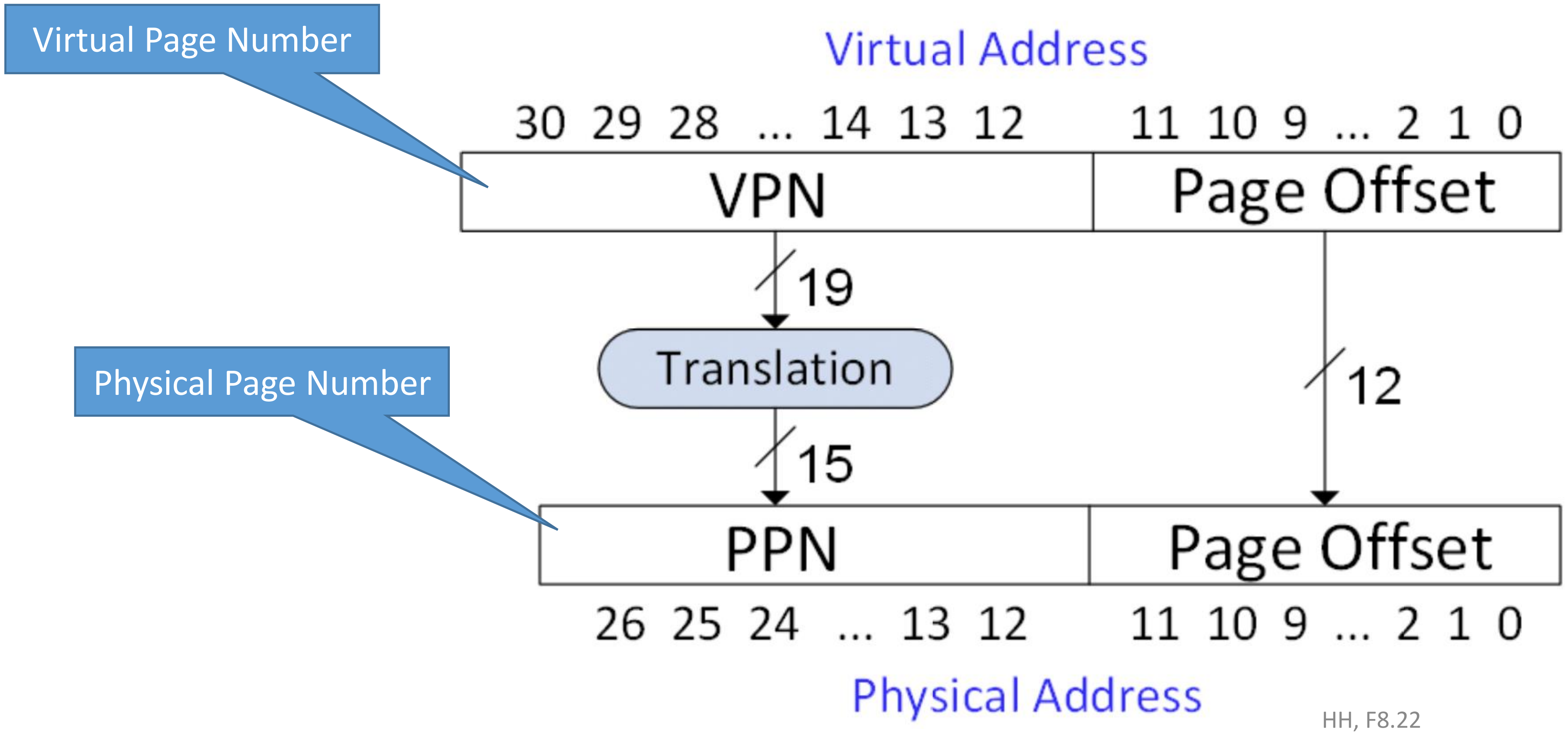
- Main memory acts as fully associative cache for the HDD (managed by OS)
  - Every virtual page can map to every physical page
    - Note: The approach of using a page table is different than the approach we used in the HW-managed caches → we do not need to search through all physical memory locations

  - A memory access to a virtual page that is not mapped to a physical page, leads to a page fault
    - In case the page is located on the HDD, the OS brings the page into main memory – the OS defines the replacement strategy for pages in main memory
    - In case of an access is outside the address space of the process, this leads to a segmentation fault

Analogies for Caches and Virtual Memory

Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

High-Level View on Address Translation

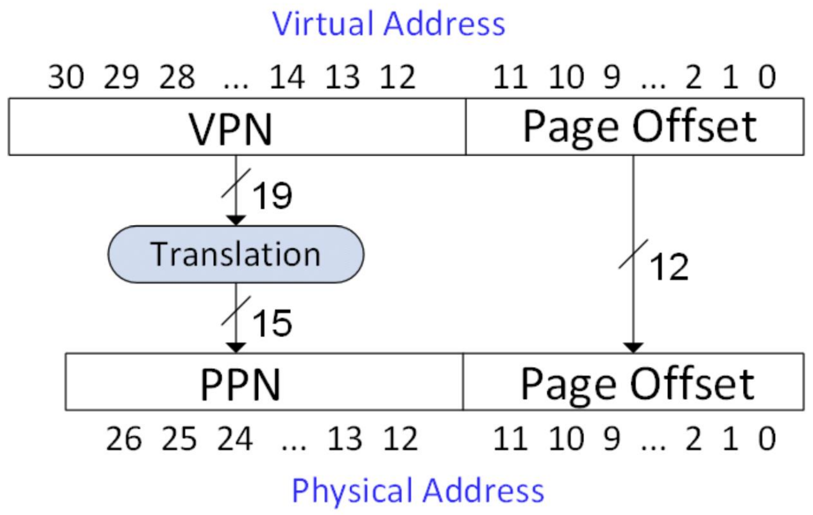


HH, F8.22

Address Translation - Example

- **System**
  - Virtual Memory Size: 2 GB = 2^{31} bytes
  - Physical Memory Size: 128 MB = 2^{27} bytes
  - Page Size: 4KB = 2^{12} bytes

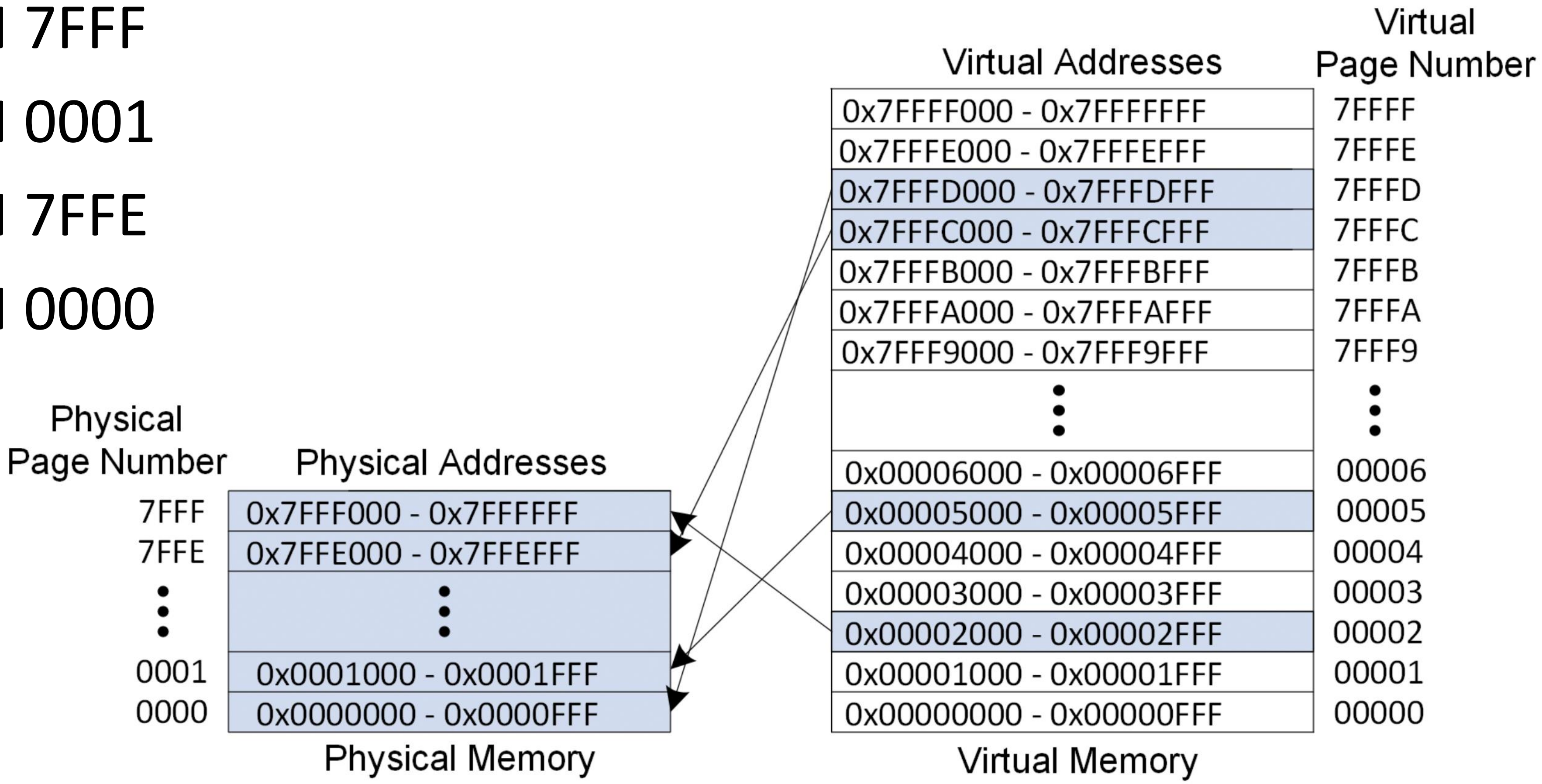
- **Number of Pages**
  - Virtual address: **31** bits
  - Physical address: **27** bits
  - Page offset: **12** bits
  - # Virtual pages = $2^{31}/2^{12}$ = $\mathbf{2^{19}}$ (VPN = 19 bits)
  - # Physical pages = $2^{27}/2^{12}$ = $\mathbf{2^{15}}$ (PPN = 15 bits)



HH, F8.22

Example Mapping VPN → PPN

- VPN 00002→ PPN 7FFF
- VPN 00005→ PPN 0001
- VPN 7FFFC→ PPN 7FFE
- VPN 7FFFD→ PPN 0000



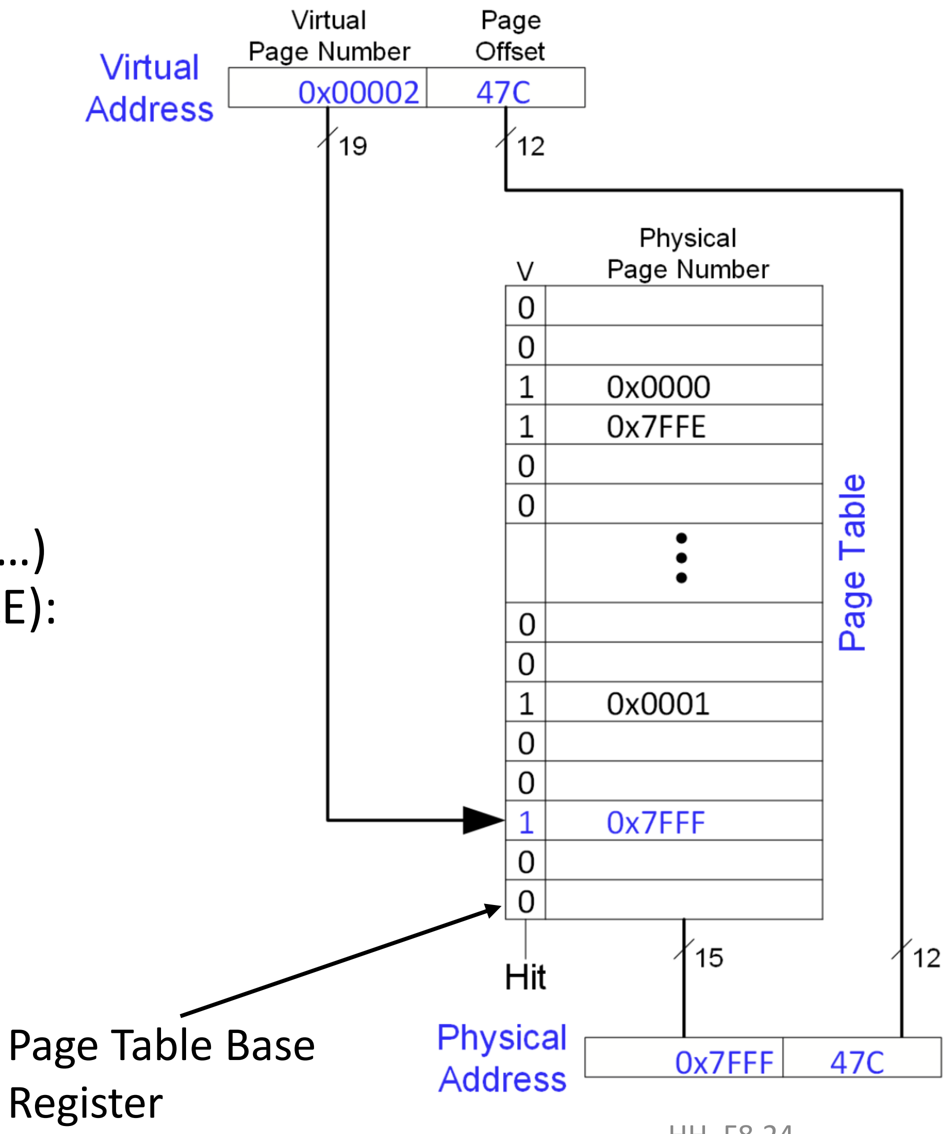
HH, F8.21

The Page Table

The page table maps virtual to physical addresses

- Page Table
  - Is indexed via the virtual page number
  - Contains a valid bit and the physical address for every page table entry (PTE)
  - Contains also additional metadata (replacement, dirty, …)
  - Typical page table entry size on 32-bit systems (PTE_SIZE): 1 word (4 bytes)

- Storing the page table
  - Page table is large
  - Page table is stored in physical memory
  - The Page Table Base Register specifies location of the page table in physical memory (satp on RISC V, CR3 on Intel)

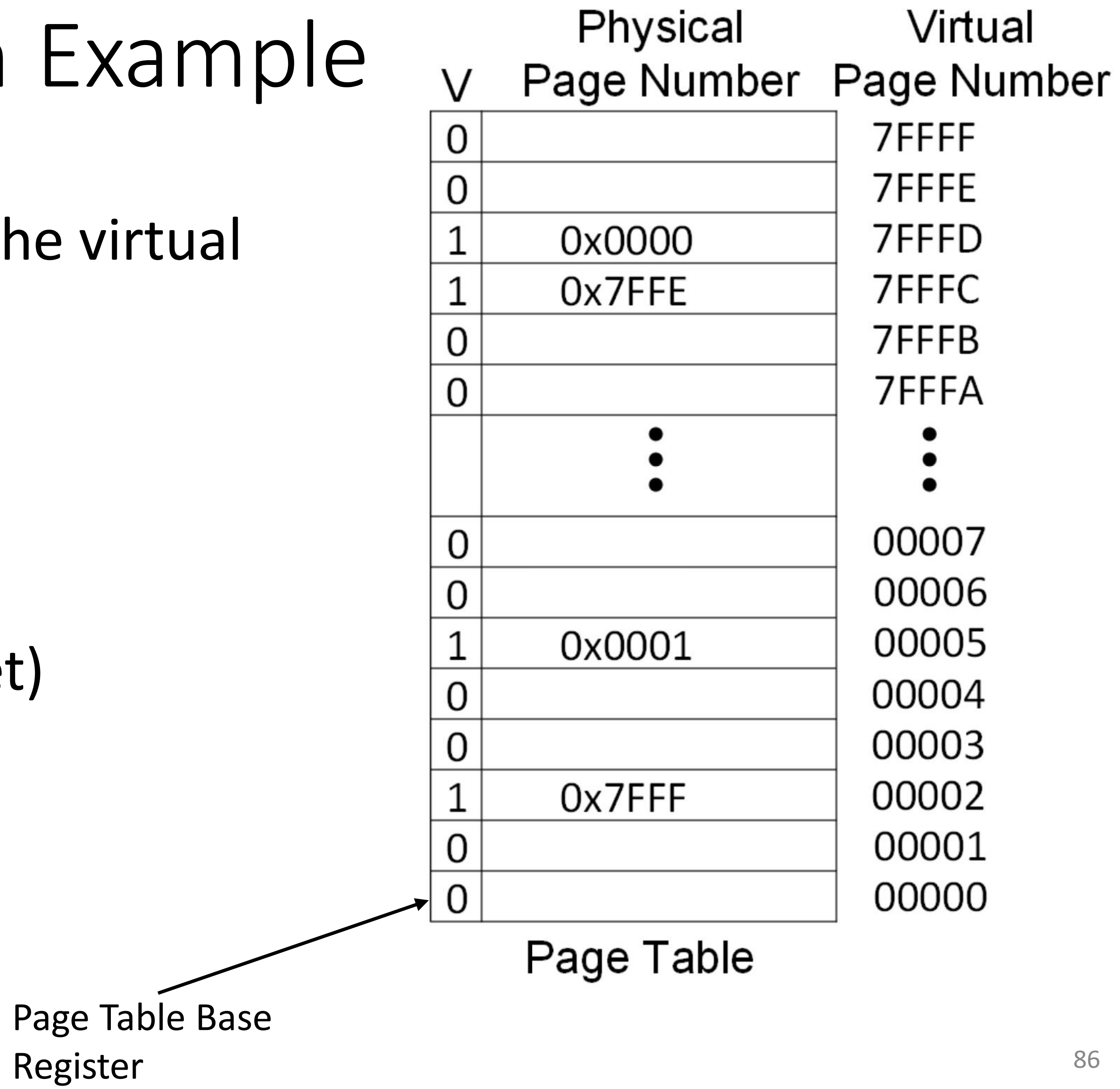


HH, F8.24

Page Table Translation Example

- What is the physical address of the virtual address 0x5F20?
- We first need to find the VPN
- The page size is 4KB (12 bit offset)

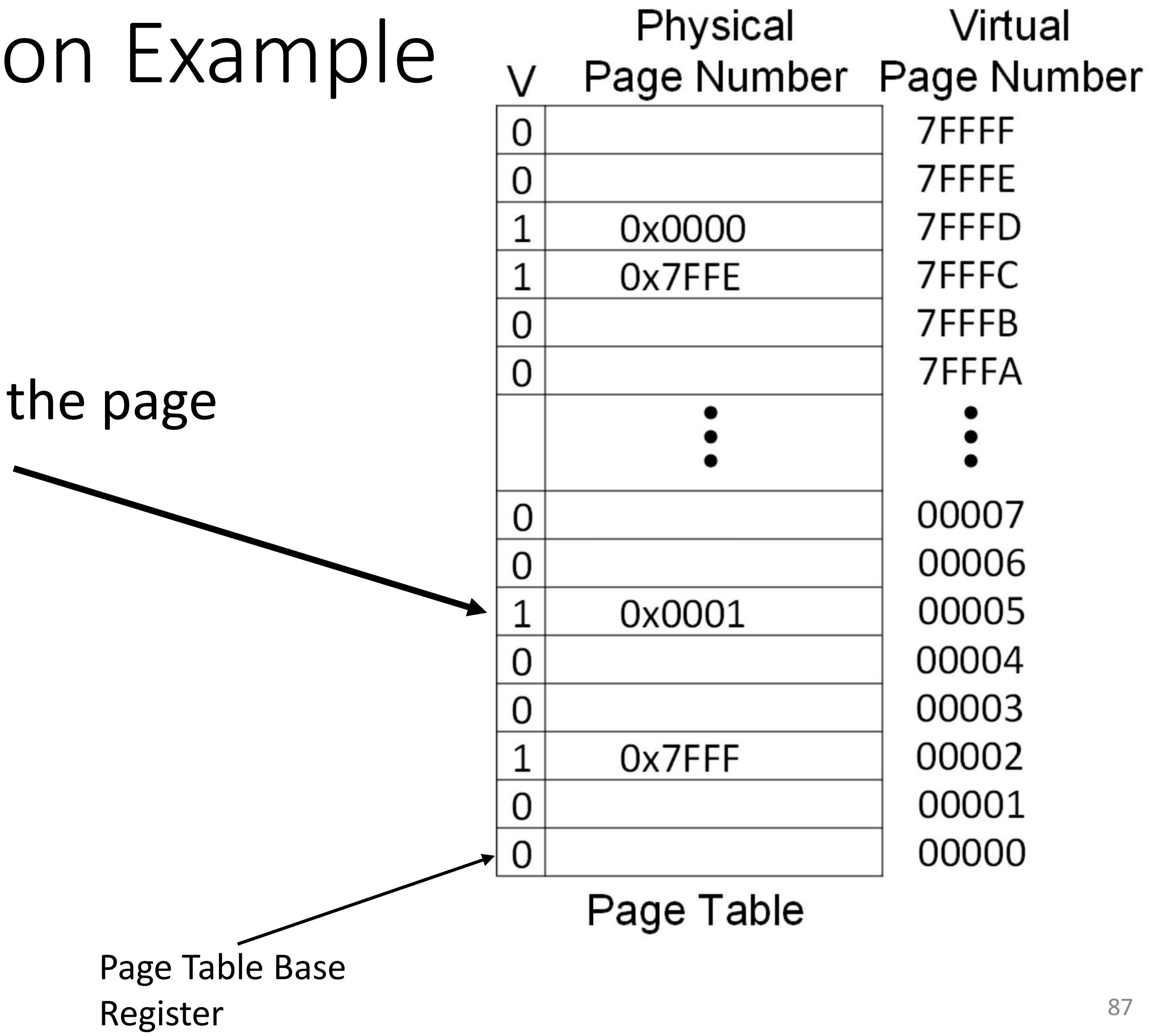
→ VPN = 0x5 ~~F20~~



V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table Translation Example

- VPN = 5
- Calculation of the address in the page table: PTBR + VPN*PTE_SIZE

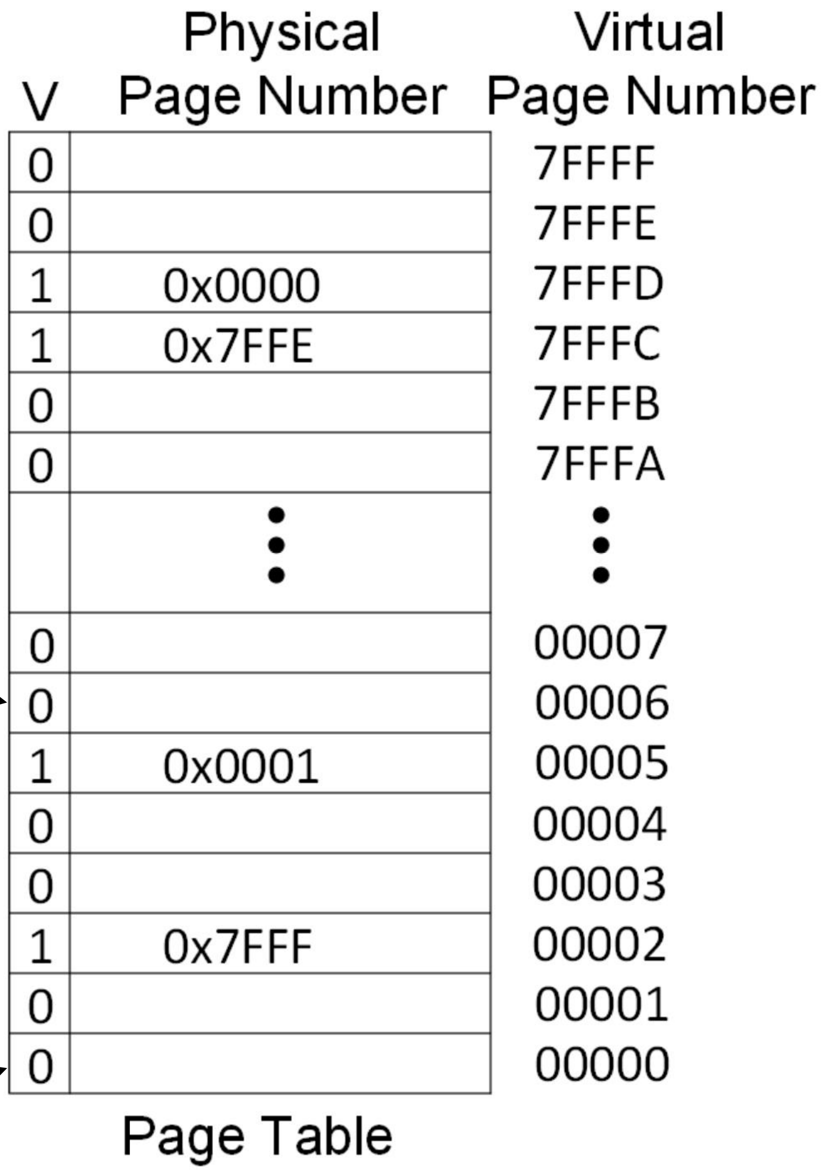


V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table Translation Example

- What is the physical address of the virtual address 0x61FF?

- This entry is not valid and leads to a page fault. It needs to be swapped from the disk into memory



V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table Base Register

Doing a Memory Access (Load/Store)

1. The CPU sends a virtual address to the memory management unit (MMU)
2. MMU determines the PPN by performing a load from (PTBR + VPN*PTE_SIZE)
3. If the entry is not valid, swap the page from the HDD to main memory
4. If the entry is valid, concatenate the offset of the virtual address to the PPN to receive the physical address of the data
5. Perform a load/store based on the physical address

→Note: Each memory access now requires two memory accesses:

1x load from page table

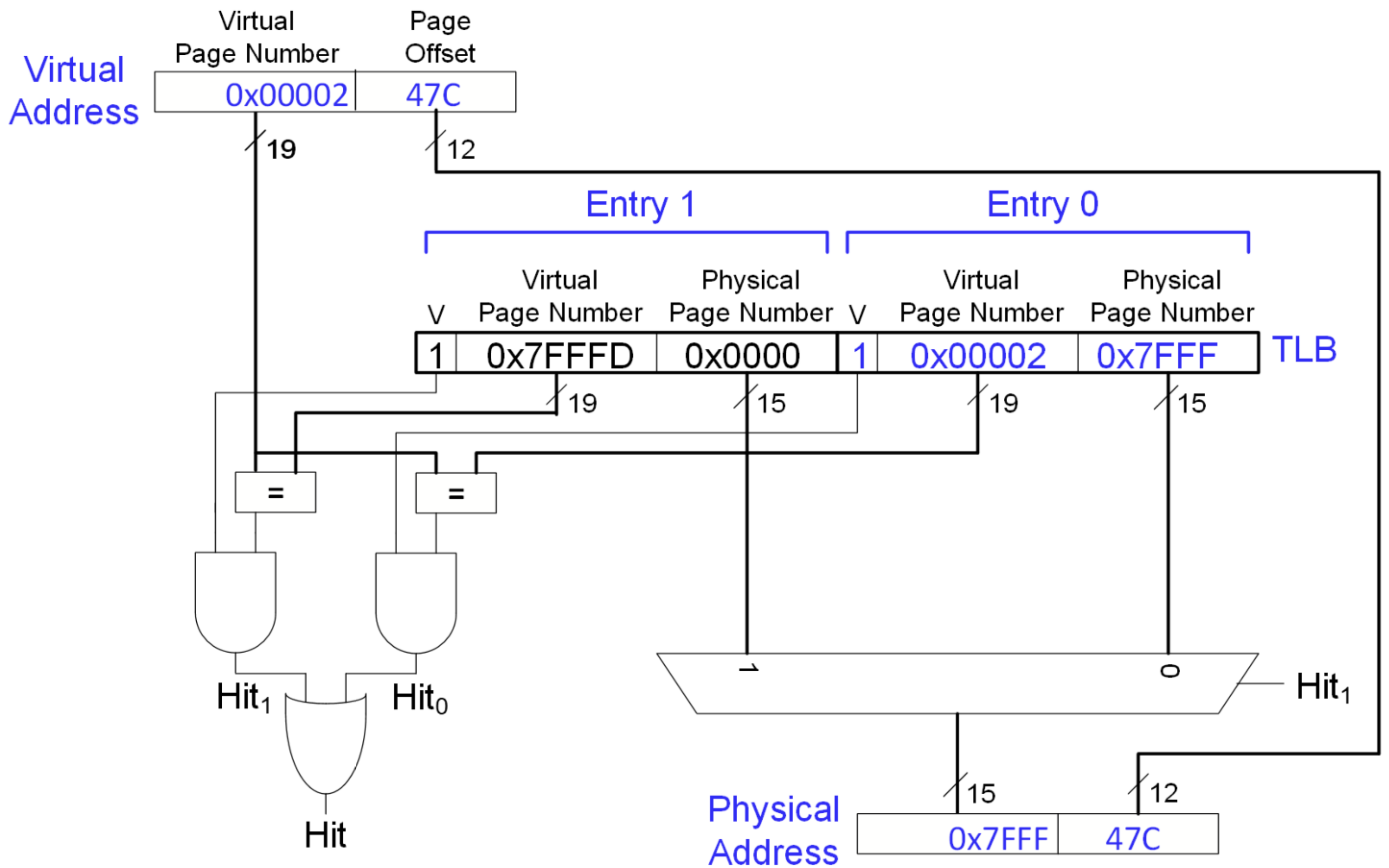
1x the actual load/store operation

(+potentially swapping)

Translation Lookaside Buffer (TLB)

- If it was necessary to do two memory operations for every load/store, this would have a huge performance impact
- The Translation Lookaside Buffer (TLB) is a Cache for page-table entries
- Highly/Fully associative
- Typically 16 – 512 entries
- TLBs achieve very high hit rates (95% - 99%) → only one memory operation for a load/store and no additional load. Only in case of a TLB miss, the page table entry needs to be fetched from memory.
- TLBs achieve very high hit rates because of high spatial/temporal locality

TLB Structure

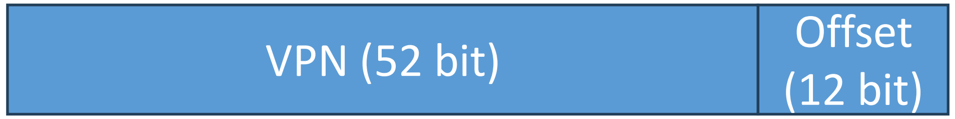


HH, F8.25

The Size of a Page Table

- How large is a page table?

- Consider a 64-bit system with 4KB pages



→2^{52} page table entries; each entry has 4 bytes → 2^{54} bytes for storing the page table for one process

→ Keeping this large table in memory is not practical, and we need to optimize

Multi-Level Page Tables

- **Motivation**
  - The virtual memory space is typically used only sparsely (a process typically does not use all its virtual memory space)
  - A hierarchical page-table scheme (tree-like structure) only requires to keep a smaller first-level page table in physical memory

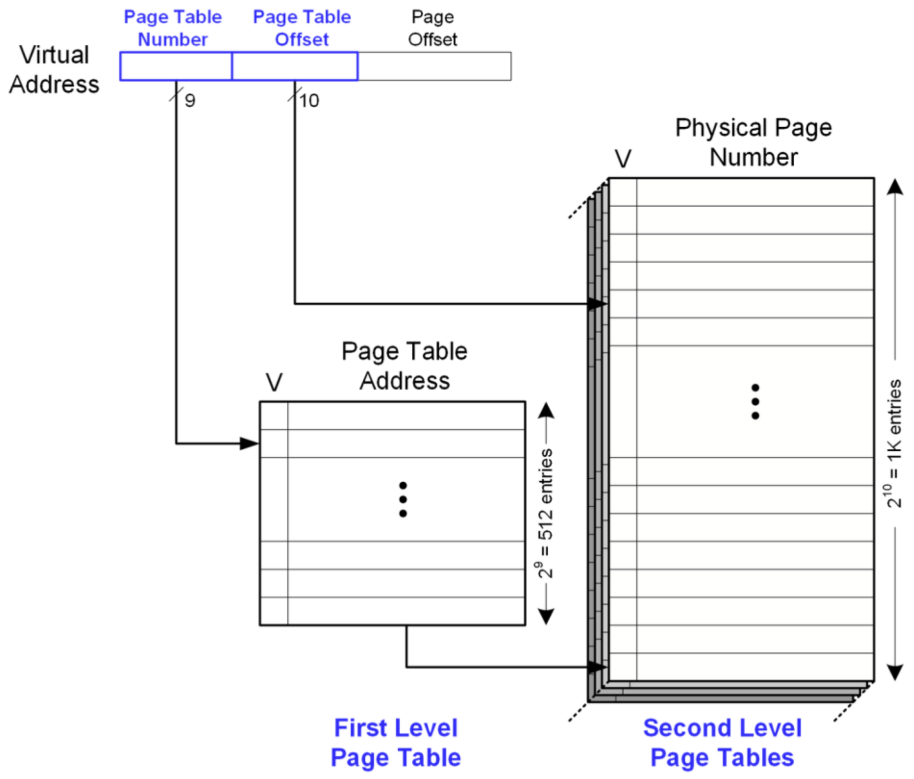
- On 32-bit systems we typically use 2-level page tables (it is up to 5 levels on 64-bit systems)

Example of 2-Level Paging

- Assume a virtual address space of 2GB (31 bit):
  - 9 bit page table number (first level)
  - 10 bit page table offset (second level)
  - 12 bit page offset

- Note: Typical Programs will not need all entries in the first-level page table

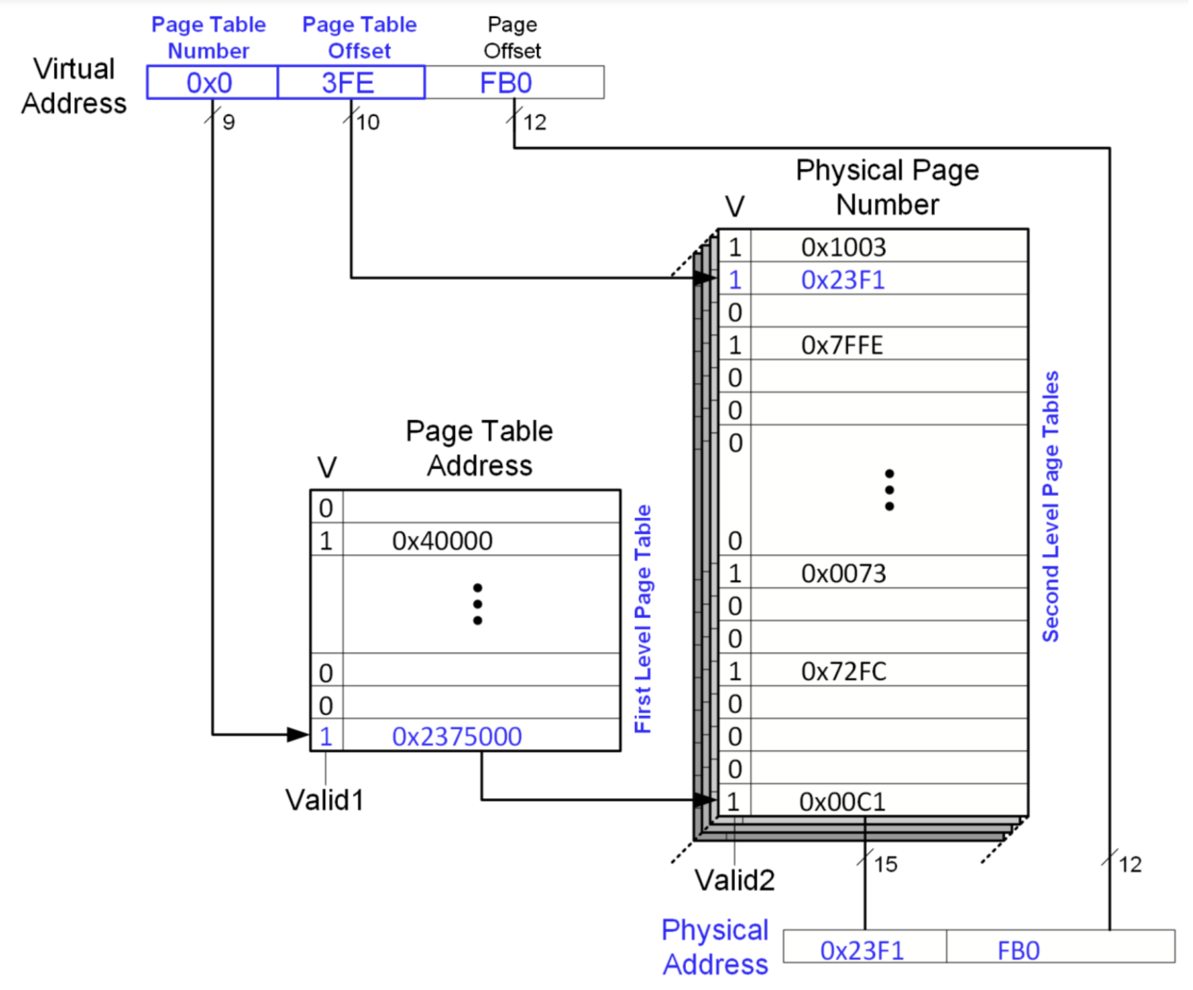
  → page tables on the second level only need to be allocated for those memory parts that are actually needed



Example Lookup

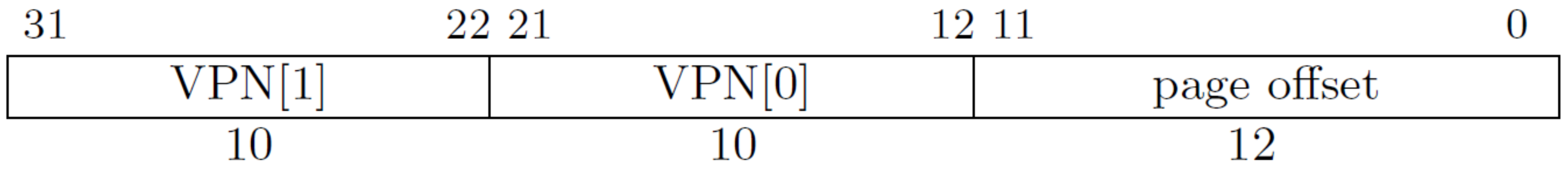
What is the physical address of virtual address 0x03FEFB0?

- We determine the page table number and use this to index the first-level page table
- This leads to 0x2375000 as physical address for the second-level page table.
- We access the second-level page-table at index 0x3FE and receive the physical page number 0x23F1
- The physical address is 0x23F1FB0



Example from the RISC-V Manual (32 bit)

- Pointer Layout for sv32 mode:

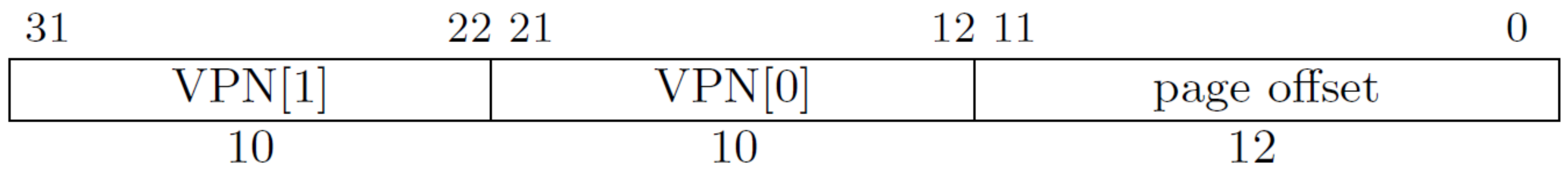


- This mode supports 4KB pages and 4MB megapages
  - For 4KB pages, we have two levels of paging (VPN[1] and VPN[0])
  - For 4MB megapages, the second level is skipped and the first table directly translates the virtual address to a physical address (Note: 22 bit correspond to 4MB)

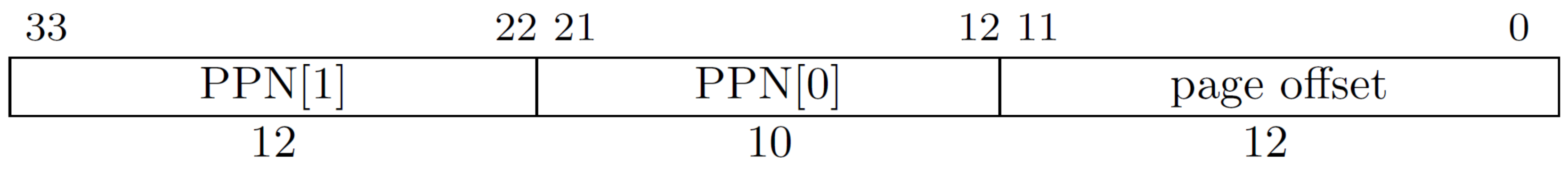
Example from the RISC-V Manual (32 bit)

- Sv32 allows to translate a 20-bit virtual page number (VPN) to a a 22-bit physical page number (PPN)

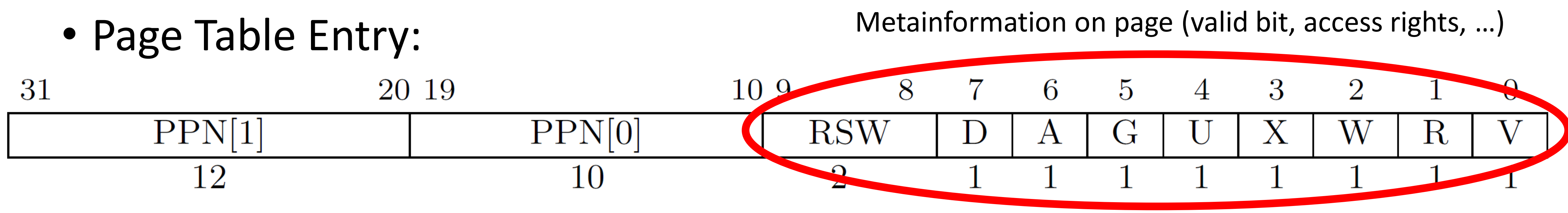
- Virtual address:



- Physical address:



- Page Table Entry:



Summary on Virtual Memory

- Virtual memory
  - provides a “fresh empty memory” for each process
  - increases memory capacity by adding the HDD as an additional level of the memory hierarchy; The operating system manages swapping
- Mechanisms
  - The memory management unit maps virtual addresses to physical address via page tables (implemented in a hierarchical manner to save space in memory)
  - The page table entries contain metadata defining e.g. access rights
  - The TLB acts as cache for page table entries (doing the mapping of virtual to physical page number)

Combining Virtual Memory and Caches

Virtual Memory and Caches

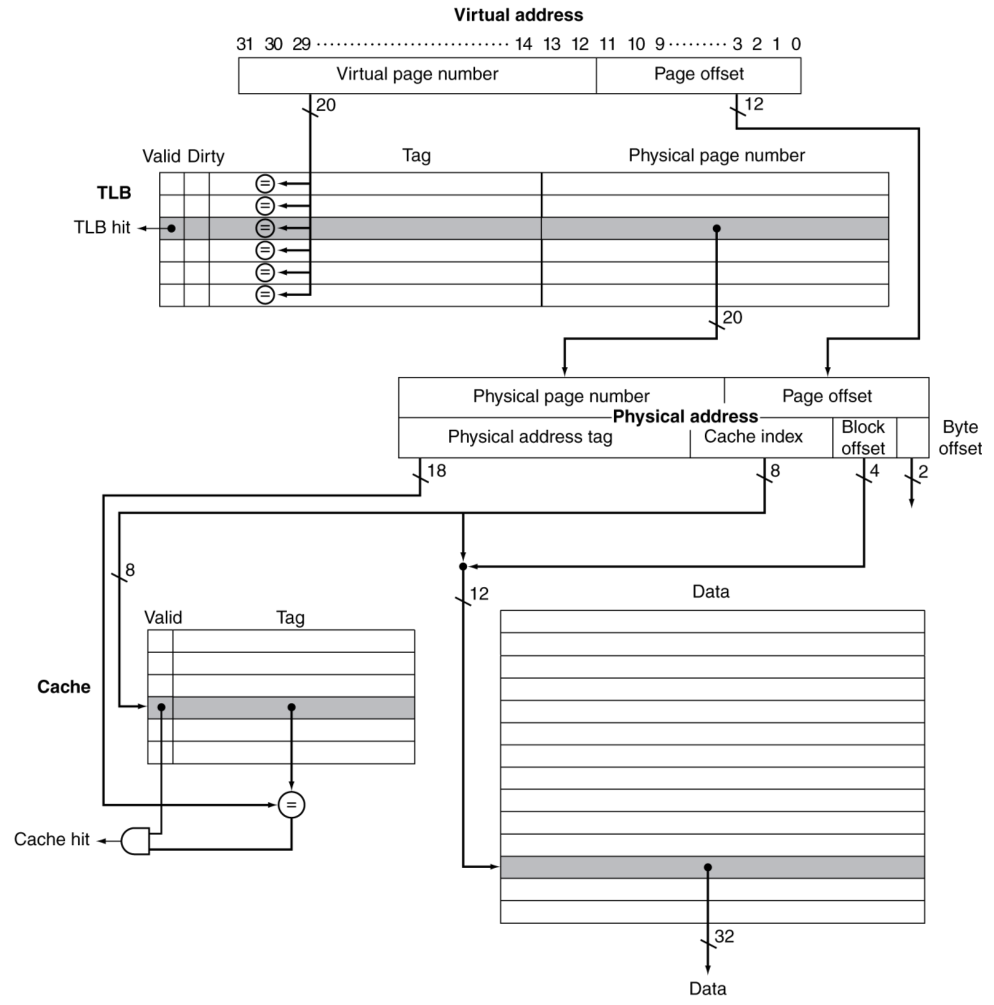
- In this chapter, we have built an abstraction for physical memory that consists of a memory hierarchy with different levels of caches

- We have then added another layer of abstraction by introducing virtual memory that is mapped to physical memory via an MMU

→All the memory lookups for page tables that don't lead to a TLB hit, will run through the memory system with its caches

Note: On large systems with multi-level paging, multiple levels of TLB caches, and multiple levels of caches for the physical memory this creates quite some interactions …

Example System with 1-level Paging, 1 TLB Cache and 1 Cache for Physical Memory



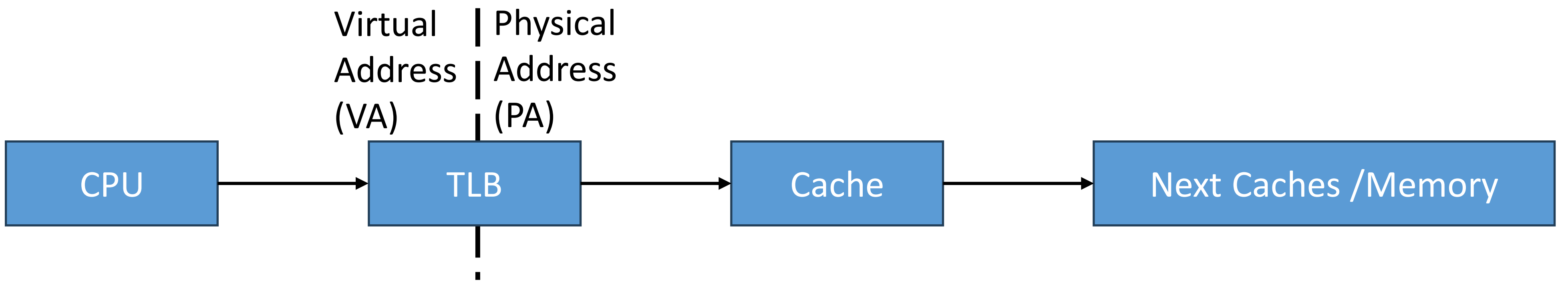
- A memory access can encounter
  - **A TLB miss →** the MMU need to read the page table from memory to get the physical address
  - **A Page Fault →** the operating system needs to swap the page from HDD to physical memory
  - **A Cache Miss →** data needs to loaded from main memory into the cache

- In best case, there is a hit on all types of accesses
- In the worst case, there is a miss on all types of accesses

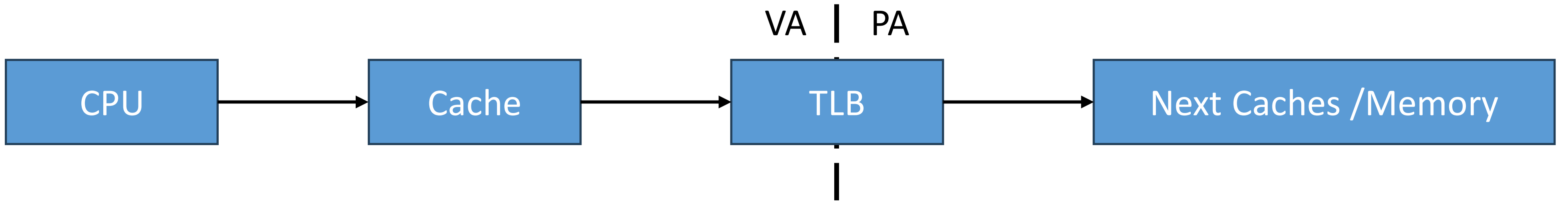
Optimizing Speed – Positioning of the Cache

- Misses on TLB and cache accesses lead to significant overheads
- Is it possible to do other arrangements of TLB and caches for better speed?
- The example shown corresponds to what is called a “Physically indexed and Physically Tagged” (PIPT) Cache → The cache only works with physical addresses

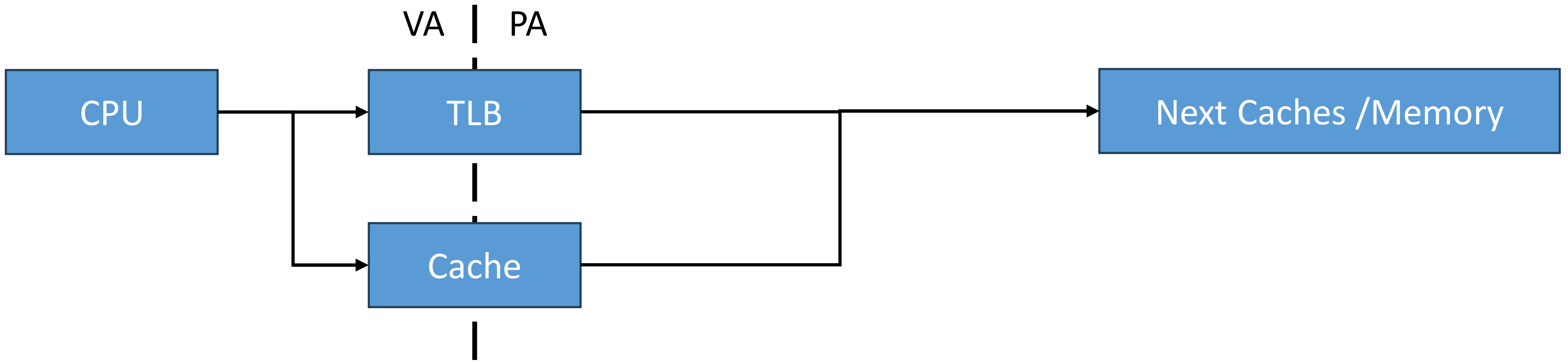
Optimizing Speed – Positioning of the Cache



"Physically indexed Physically Tagged" (PIPT)



"Virtually indexed Virtually Tagged" (VIVT)



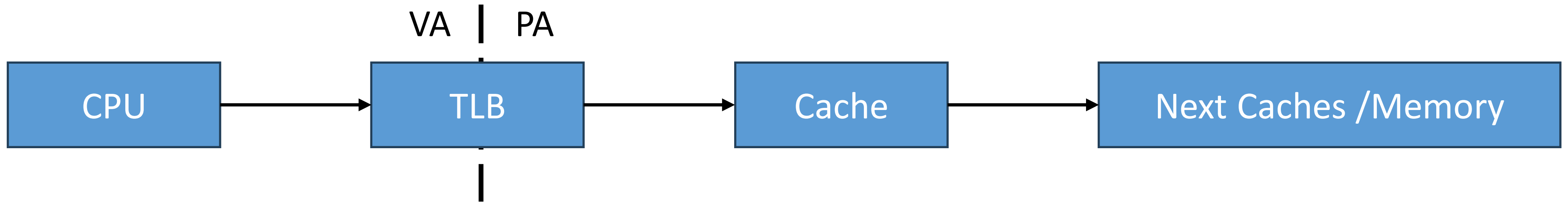
"Virtually indexed Physically Tagged" (VIPT)

Physically indexed Physically Tagged Cache (PIPT)

- Cache works as the caches we have discussed at the beginning of this slide set

- Drawback:

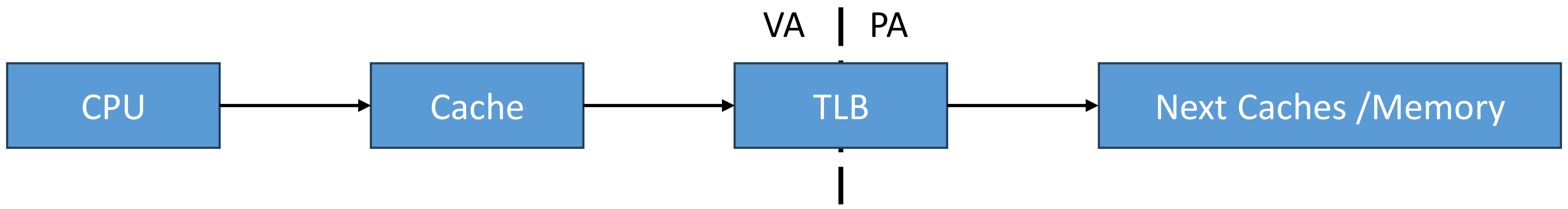
We need to wait with the cache access until the TLB delivers the physical address before we can access the cache



Virtually Indexed Virtually Tagged Cache (VIVT)

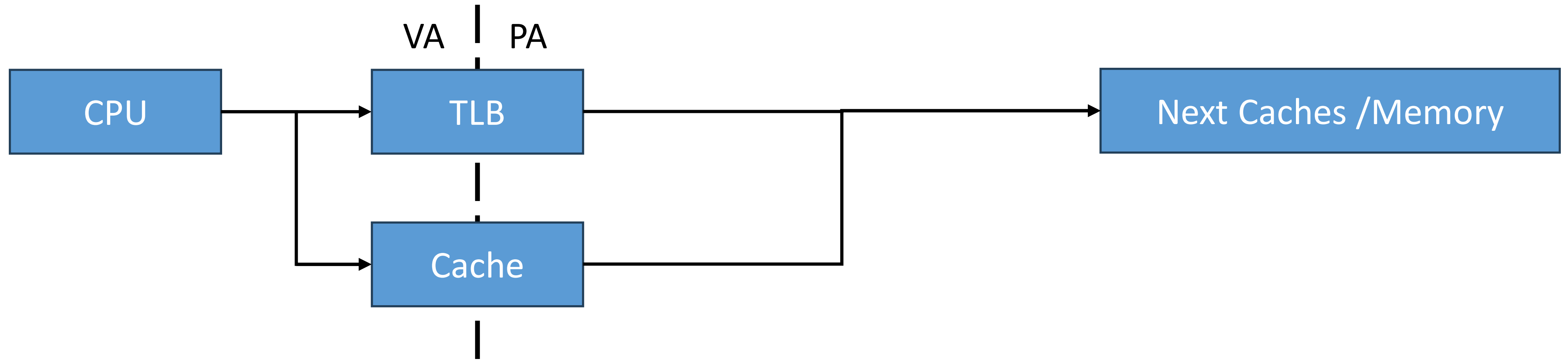
- Cache works on virtual addresses only – no need to wait for TLB
- TLB access is only needed in case of a cache miss
- Drawback:

Different processes have different mappings (virtual tag is not unique) → shared memory can be in cache multiple times. This aliasing needs to be handled.



Virtually indexed Physically Tagged Cache (VIPT)

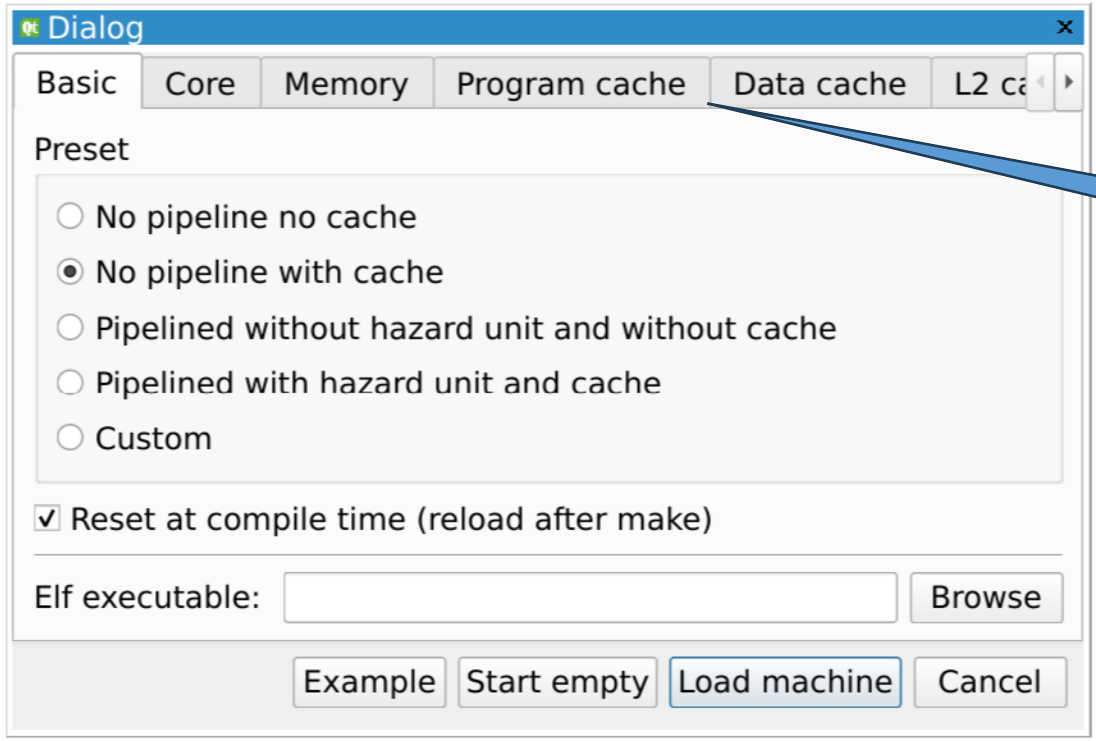
- The virtual address is used to index the cache, but the physical address is used for the tagging of the cache
- Advantage: Lookups of TLB and Cache can be started in parallel.
- Disadvantage: Aliasing may occur if a process accesses the same physical memory via two different virtual addresses
- Typical compromise:
  - Keep the size of the cache so low that the tagging is just done using address bits of the page offset (which is the same for virtual and physical addresses)
  - Example: 4KB Page size (12 bit used for offset) → Build caches that are not larger than 4KB * (Number of Ways)



Simulating Caches with QtRVSim

QtRVSim

- QtRVSim allows configuring different caches for the basic CPU (non-pipelined) that we have designed earlier in the lecture



Preset

- ( ) No pipeline no cache
- (●) No pipeline with cache
- ( ) Pipelined without hazard unit and without cache
- ( ) Pipelined with hazard unit and cache
- ( ) Custom

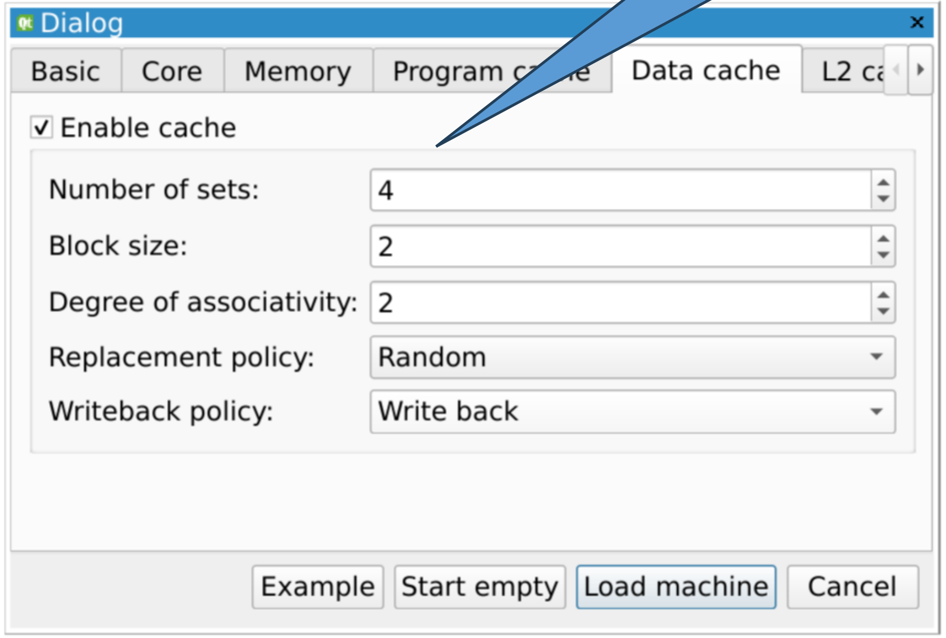
☑ Reset at compile time (reload after make)

Elf executable:

Configure Your Instruction and Data Caches here

QtRVSim

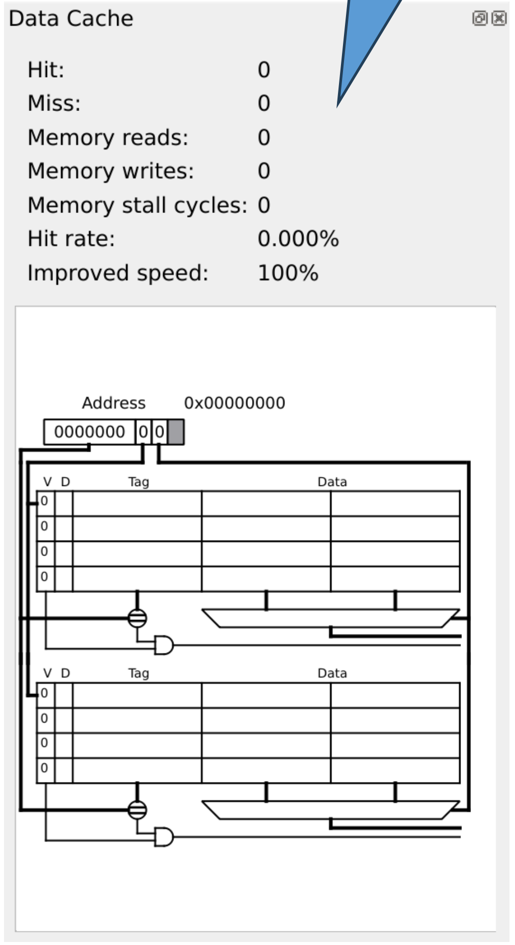
Configuration



☑ Enable cache

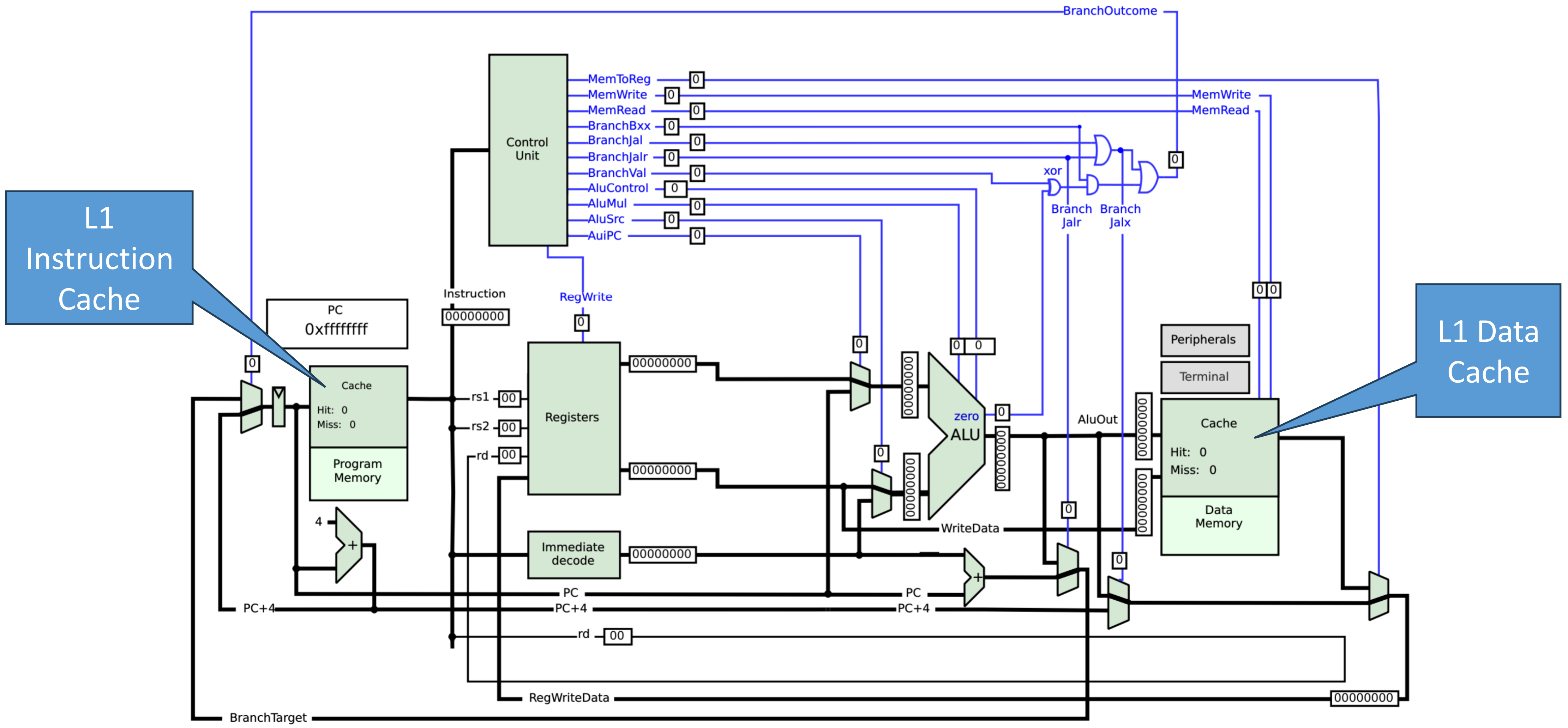
Number of sets:	4
Block size:	2
Degree of associativity:	2
Replacement policy:	Random
Writeback policy:	Write back

Watch activity and content during the simulation



Hit:	0
Miss:	0
Memory reads:	0
Memory writes:	0
Memory stall cycles:	0
Hit rate:	0.000%
Improved speed:	100%

QtRVSim


L1 Instruction Cache
L1 Data Cache
BranchOutcome
Control Unit
MemToReg
MemWrite
MemRead
BranchBxx
BranchJal
BranchJalr
BranchVal
AluControl
AluMul
AluSrc
AuiPC
xor
Branch Jalr
Branch Jalx
PC
0xffffffff
Instruction
00000000
RegWrite
Cache
Hit: 0
Miss: 0
Program Memory
rs1
rs2
rd
Registers
Immediate decode
zero
ALU
AluOut
Peripherals
Terminal
Data Memory
WriteData
PC+4
BranchTarget
RegWriteData

Examples

- Look at the examples repo and simulate the code examples in the directory

  con09.01_QtRVSim_cache_examples

- Use the following files as starting point – change the files, the cache configuration, add more load operations, add store operations, … to see effects on the hit ratio of the cache:
  - **cache_example_1.S**
  - **cache_example_2.S**

Errata

This errata slide lists typos that have been fixed after the recording of the lecture during winter term 2023/2024

- 29.01.2024:
  - Slide 44: 25 $\rightarrow$ 24
  - Slides 103 – 106: PT $\rightarrow$ PA