UNIT 6: MEMORY HIERARCHY

Estructura de Computadores (Computer Organization)
Course 2018/2019
ETS Ingeniería Informática
Universitat Politècnica de València

Unit goals

- To understand the concept and motivation of memory hierarchy
- To learn how cache memory operates
- To understand how cache design parameters impact the overall performance of the memory system
- To understand the basic processor mechanism to support virtual memory efficiently

Unit contents

- 1. Cache memory
  - Basic concepts
  - Mapping schemes
  - Write policies
  - Replacement algorithms
  - Multilevel caches
  - Examples
- 2. Virtual memory
  - Concept and motivation
  - Virtual addressing
  - Address translation
- 3. An example combining cache and VM
  - MIPS R2000 and the DECstation 3100

Bibliography

- D. Patterson, J. Hennessy. *Computer organization and design. The hardware/software interface*. 4th edition. 2009. Elsevier
  - Chapter 5 (sections 5.1 to 5.5)
- W. Stallings. *Computer Organization and Architecture. Designing for Performance*. 7th edition. 2006. Prentice Hall
  - Chapter 4 (4.2)
- C. Hamacher, Z. Vranesic, S. Zaky. *Computer Organization*. 5th edition. 2001. McGraw-Hill
  - Chapter 5 (5.5 to 5.7)

1. Cache memory

- Basic concepts
- Mapping schemes
- Write policies
- Replacement algorithms
- Multilevel caches
- Examples

Motivation

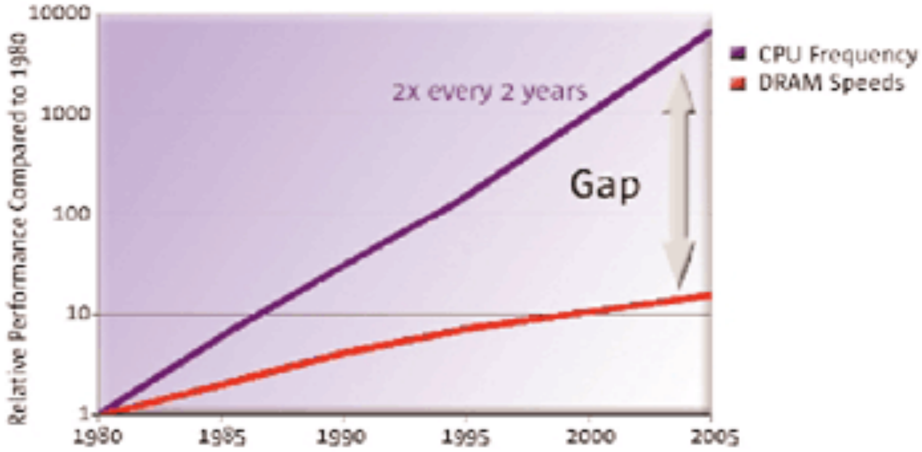
- Ideally, memory should be both large and fast
  - However, the larger the slower; the faster the smaller
- Other factors need also be considered
  - Cost per bit, power consumption, reliability…
- No single device is ideal under all the requirements

Technology	Access time (typ.)	Cost/GB
SRAM	0.5 .. 2.5 ns	$500 .. $1000
DRAM	50 .. 70 ns	$10 .. $20
Flash	5,000 .. 50,000 ns	$0.75 .. $1
Magnetic disk	5,000,000 .. 20,000,000 ns	$0.05 .. $0.10

Year 2012

The memory-CPU performance gap

- Performance increase rate:
  - Processors: 60% per year
  - Memory: 7% per year
- As a result, the *performance gap* grows



Intel Xeon 3.8 GHz
T_{clock} = 0.263 ns

≈ ×140

DDR3-1600
Access time (t_{RCD}+CL) ≈ 37.5 ns

- The gap is filled with one or more levels of faster RAM between CPU and Main Memory, at the top of the *memory hierarchy*

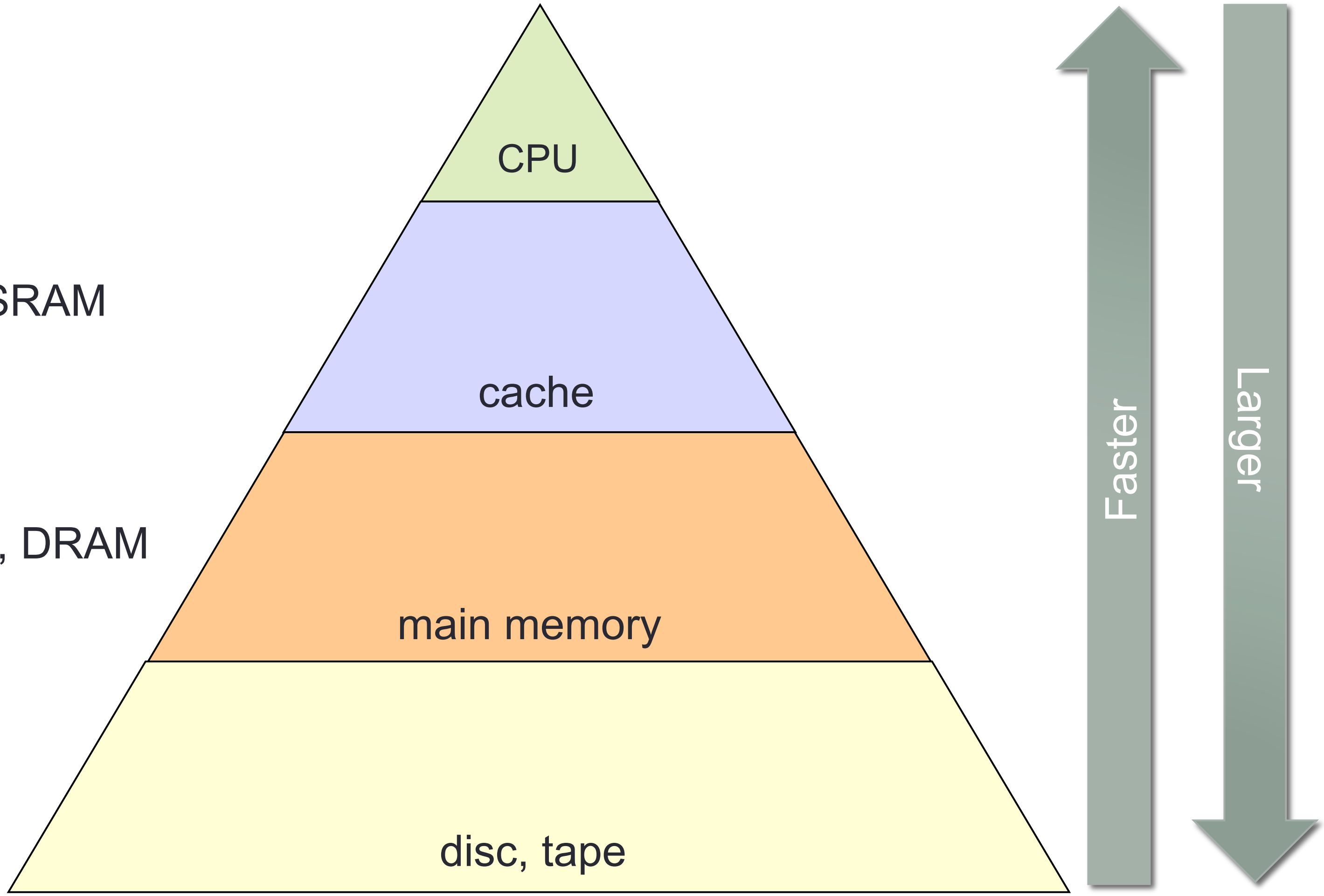
Memory hierarchy

CPU registers

On-chip/off-chip SRAM

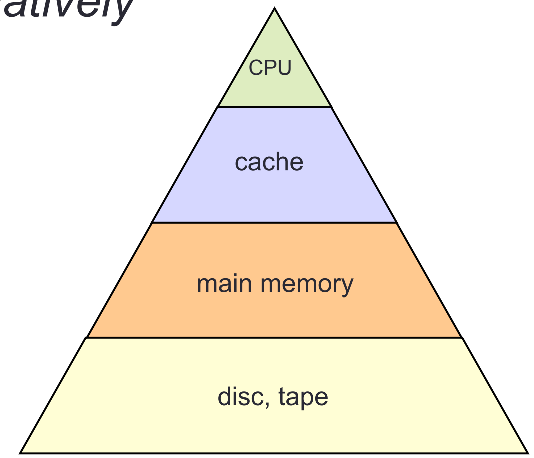
Memory modules, DRAM

Mass storage devices



Memory hierarchy: why does it work

- Higher levels contain code/data with higher probability of being accessed
- Lower levels contain code/data with lower probability of being accessed
- How is this probability determined?
  - Most memory accesses target the fastest levels thanks to the empirical principle of locality:
    - *During a given time interval, programs tend to access a relatively small portion of neighbouring addresses*

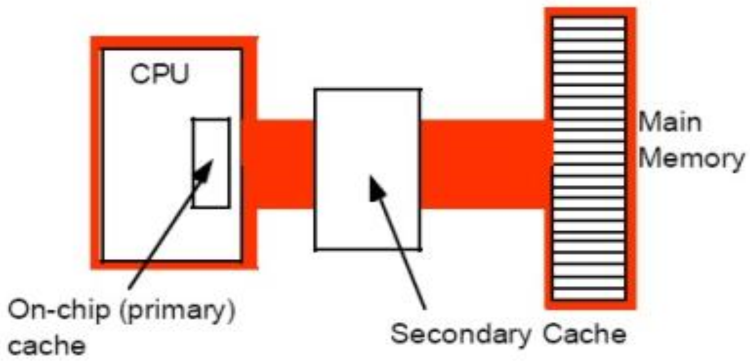


Memory hierarchy: why does it work

- Two forms of locality:
  - Temporal locality: when an item is referenced, it tends to be referenced again soon
    - e.g., instructions in loops
    - The 20-80 rule: *20% of the code takes 80% of a program's execution time*
  - Spatial locality: when an item is referenced, items in neighbouring addresses tend also to be referenced soon
    - e.g., a sequence of instructions, vector arithmetic
    - Favours accessing main memory in blocks and secondary storage in pages
- Empirically, computer programs do exhibit locality
  - And that's why a memory hierarchy works

What is cache memory

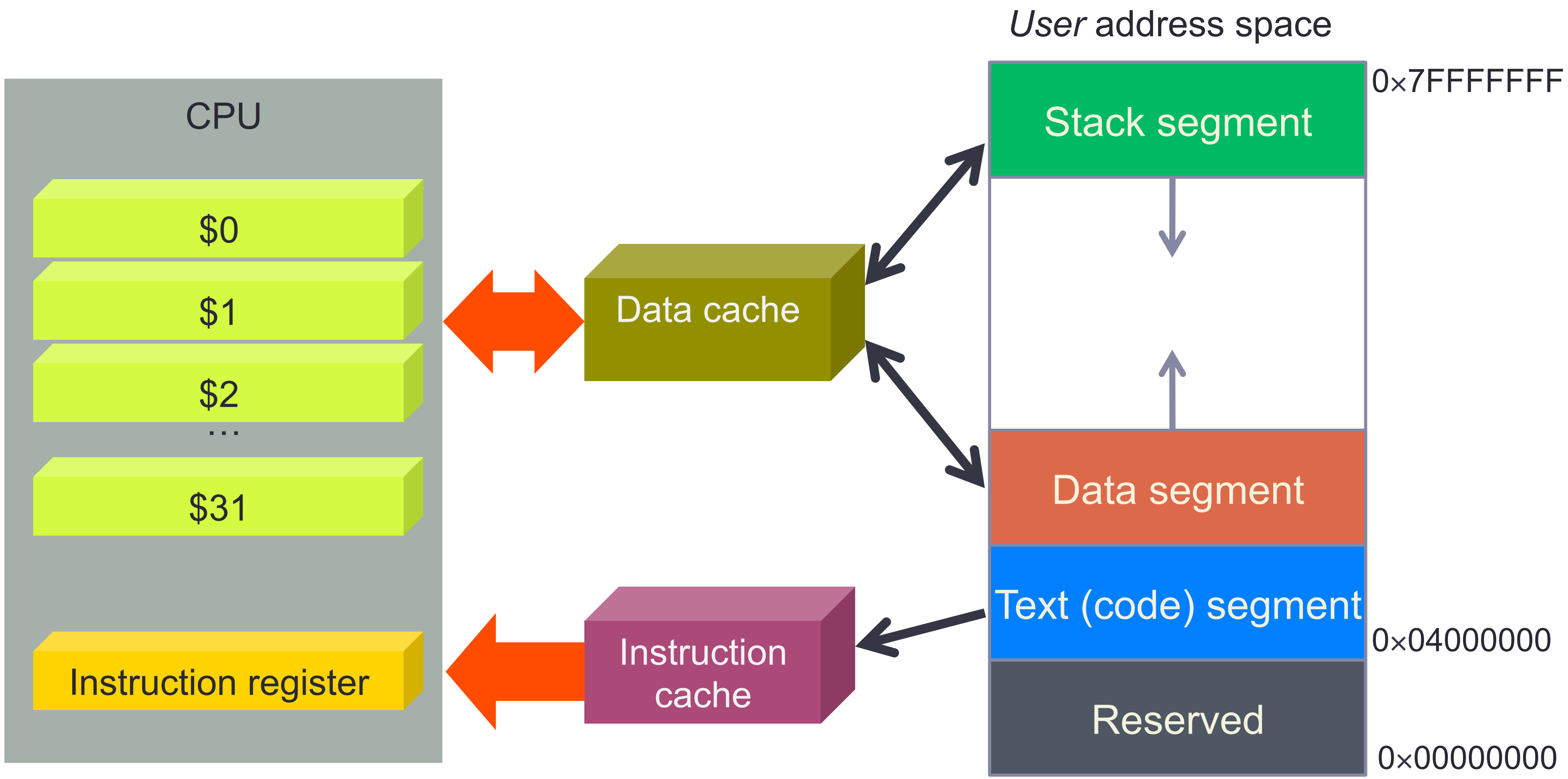
- A relatively small but fast SRAM memory
- Goal: to reduce the *average* access time to memory (means: take advantage of locality)
- Used both for code and data
- Often, several levels of cache (L1, L2…) of increasing capacity / decreasing speed
- Working principle
  - Data/instructions are first fetched from cache
    - Cache hit: the info resides in cache → fast access
    - Cache miss: info is not in cache → need to access next level



Unified and split caches

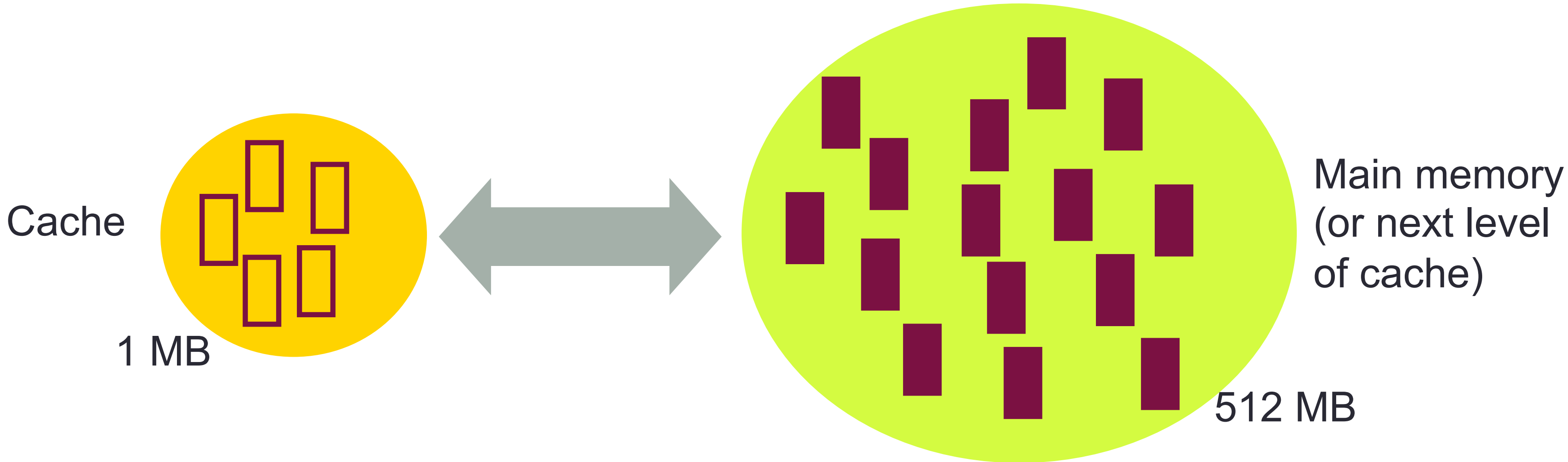
- A unified cache contains both data and instructions
- A split cache is formed by separate data and instruction caches
- Discussion
  - Split caches produce slightly lower hit rates, but also…
  - … reduce hardware complexity and make it possible to access data and code in parallel. This advantage overcomes the lower hit rate
- In practice, L1 is usually split; L2 and further, unified

Information flow in a split cache



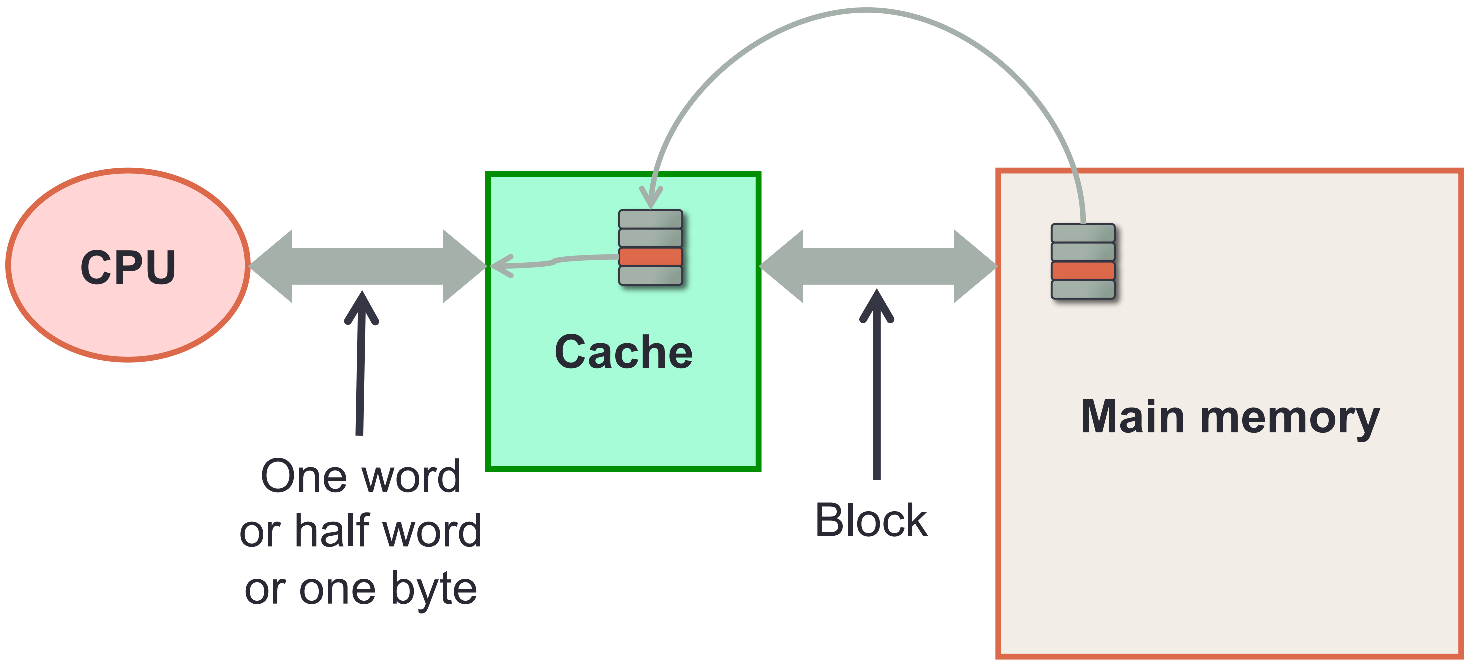
Cache – Main memory relationship

- Transfers between cache and main memory always occur in blocks (typically, sets of 4, 8 words)
  - Same applies to transfers between levels of cache
  - Remember SDRAM is optimized for this kind of accesses
- Number of blocks. Example:
  - 1 MB cache; 512 MB main memory
  - Block size: 4 × 32b words (16 Bytes)
    - Cache contains 65,536 blocks ($2^{20} / 2^4 = 2^{16}$ = 64 KBlocks)
    - Main memory contains 33,554,432 blocks ($2^{29} / 2^4 = 2^{25}$ = 32 MBlocks)



How locality is exploited

- When there is a read miss, the full block containing the missing word is brought to cache and copied to a cache slot or line
- Due to locality, there is now a higher probability of future hits



Hit rate and average access time

- Time to access data in cache: T_{hit}
- Time to access data otherwise: T_{miss}
  - It holds that $T_{hit} << T_{miss}$
- The hit rate is the ratio between hits and total accesses:

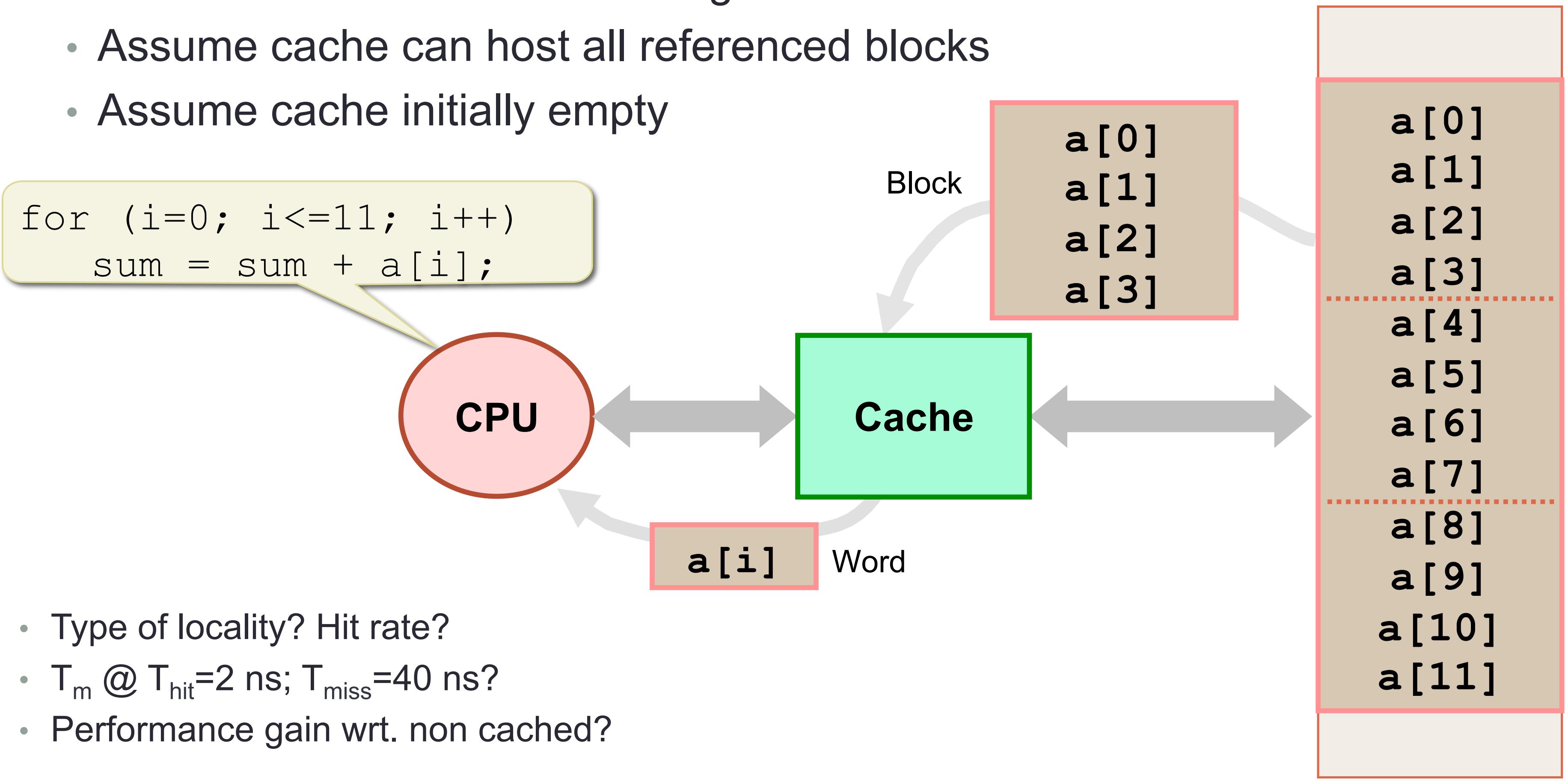
$$\boxed{H = \frac{\text{Number of hits}}{\text{Number of accesses}} = \frac{\text{Number of accesses} - \text{Number of misses}}{\text{Number of accesses}}}$$

- Conversely, the miss rate is defined as 1 – H
- The average access time is given by:

$$\boxed{T_m = H \times T_{hit} + (1 - H) \times T_{miss}}$$

Example

- Reading array `a[]` of twelve 32 b integers
  - Assume `i` and `sum` reside in registers
  - Assume cache can host all referenced blocks
  - Assume cache initially empty



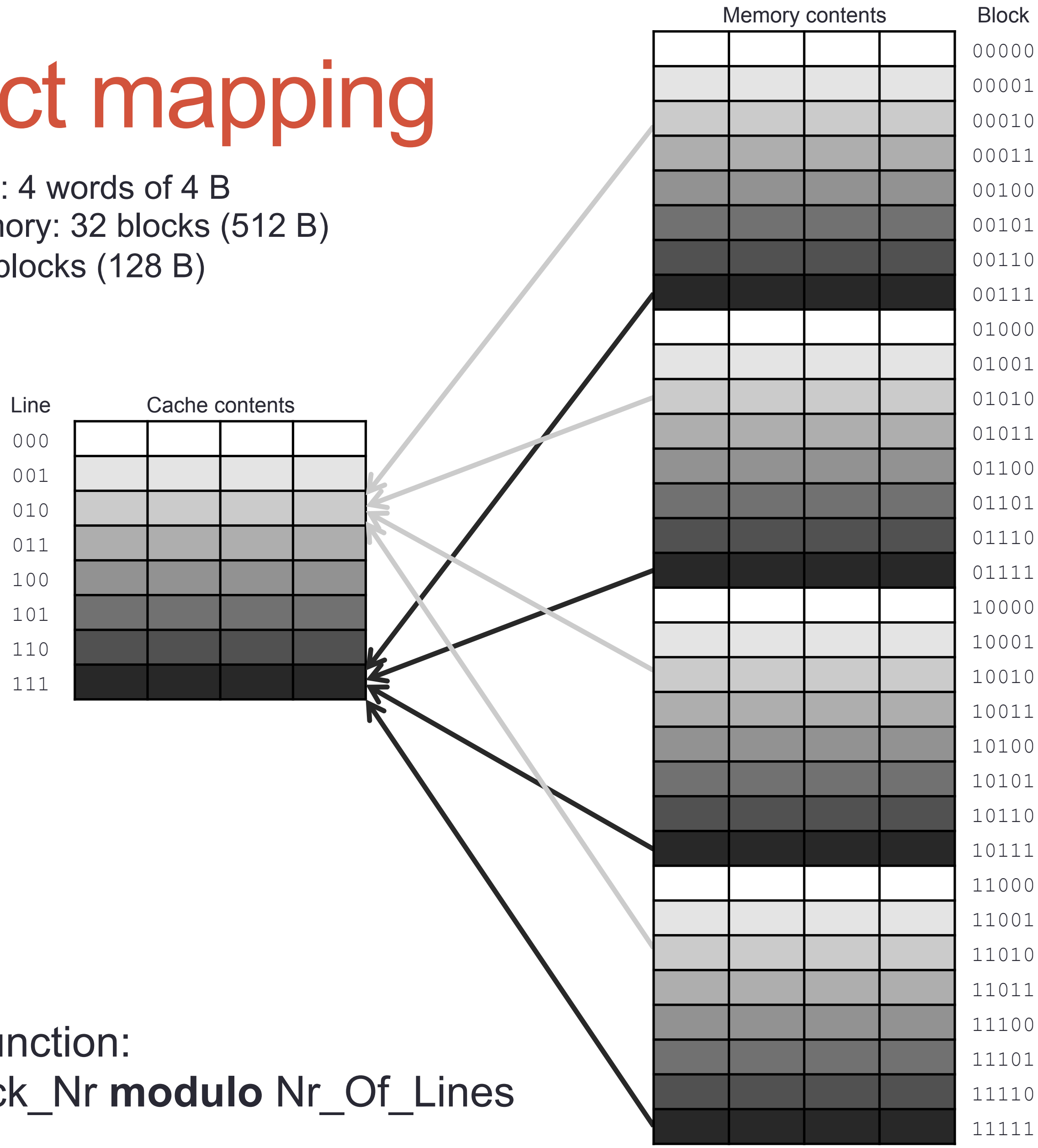
- Type of locality? Hit rate?
- T_m @ T_{hit}=2 ns; T_{miss}=40 ns?
- Performance gain wrt. non cached?

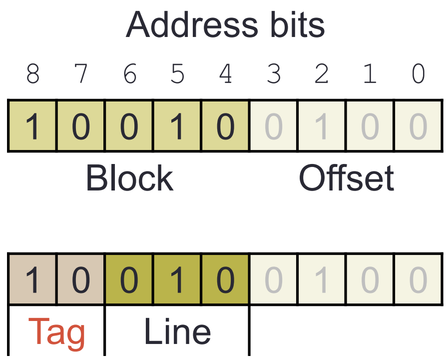
Cache mapping

- Two fundamental questions arise
  - How do we know whether an access is a hit or a miss?
  - In case of hit, where does the block reside in cache?
- The cache mapping scheme determines the answers to both questions
- Depending on mapping, a memory block can be found…
  - …only in one cache line: direct mapping
  - …in any cache line: fully associative mapping
  - …only in a subset of all cache lines: set-associative mapping
- Knowing the mapping scheme and the block address, we will be able to determine the block location in cache, or to signal a cache miss

Direct mapping

Block size: 4 words of 4 B
Main memory: 32 blocks (512 B)
Cache: 8 blocks (128 B)





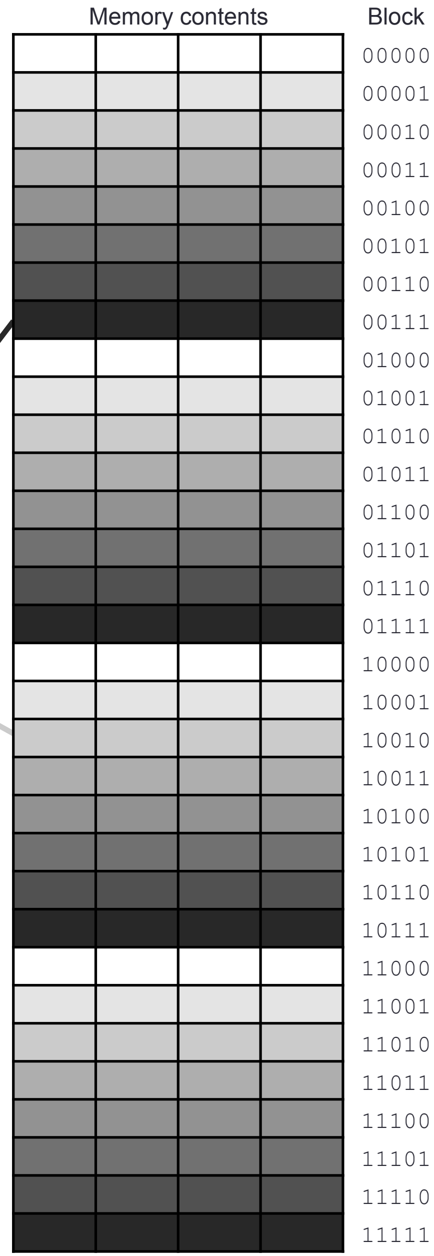
Line: Where to find me
Tag: My ID

Mapping function:
Line = Block_Nr **modulo** Nr_Of_Lines

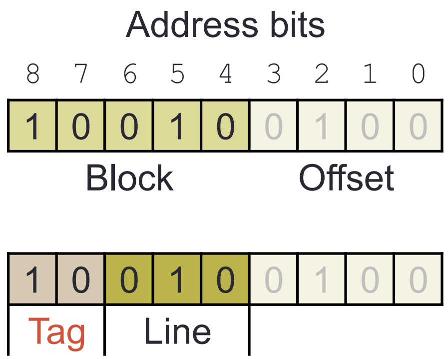
Direct mapping

Block size: 4 words of 4 B
Main memory: 32 blocks (512 B)
Cache: 8 blocks (128 B)

Line	V	Tag	Cache contents			
000	Y	00				
001	Y	00				
010	Y	10				
011	Y	11				
100	Y	11				
101	N					
110	N					
111	Y	00				



- Tags need be stored in cache, one tag per cache line
- An additional Valid (V) bit identifies empty lines in cache



Line: Where to find me
Tag: My ID

Hands on:
draw the rest of arrows, considering the tags' values

Accessing cache

- Hands on: Give the cache state after each read access in the following sequence. The cache is initially empty.

Nr	Address (binary)
1	0 1010 0000
2	0 1010 0100
3	0 1010 1000
4	1 0110 0100
5	1 1001 0100
6	1 1010 0100
7	1 1001 1000
8	0 1110 0000
9	0 0011 0011
10	0 1100 1010

Initial state

Line	V	Tag	Cache contents
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Final state – after access 10

Line	V	Tag	Cache contents
000	N		
001	Y	11	Block 11001
010	Y	11	Block 11010
011	Y	00	Block 00011
100	Y	01	Block 01100
101	N		
110	Y	01	Block 01110
111	N		

Accessing cache

- Accesses 6 and 8 show miss-and-replacement situations
  - The newest block replaces an old one
- In a direct-mapped cache there is only one possible slot per block, hence the replacement algorithm is straightforward
- In associative caches (later) there are more chances and we'll need to select the *victim* line

Tag size

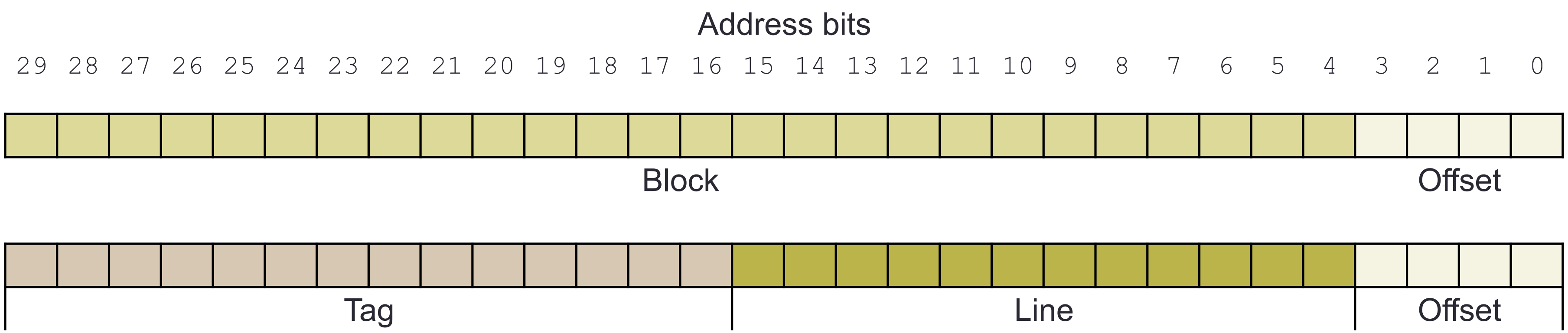
- Tags in our toy example are only 2-bit long. For the same block size, what's the tag size for a 64 KB cache and a main memory of 1 GB?

Blocks of 16 B (4 words of 4 B each):

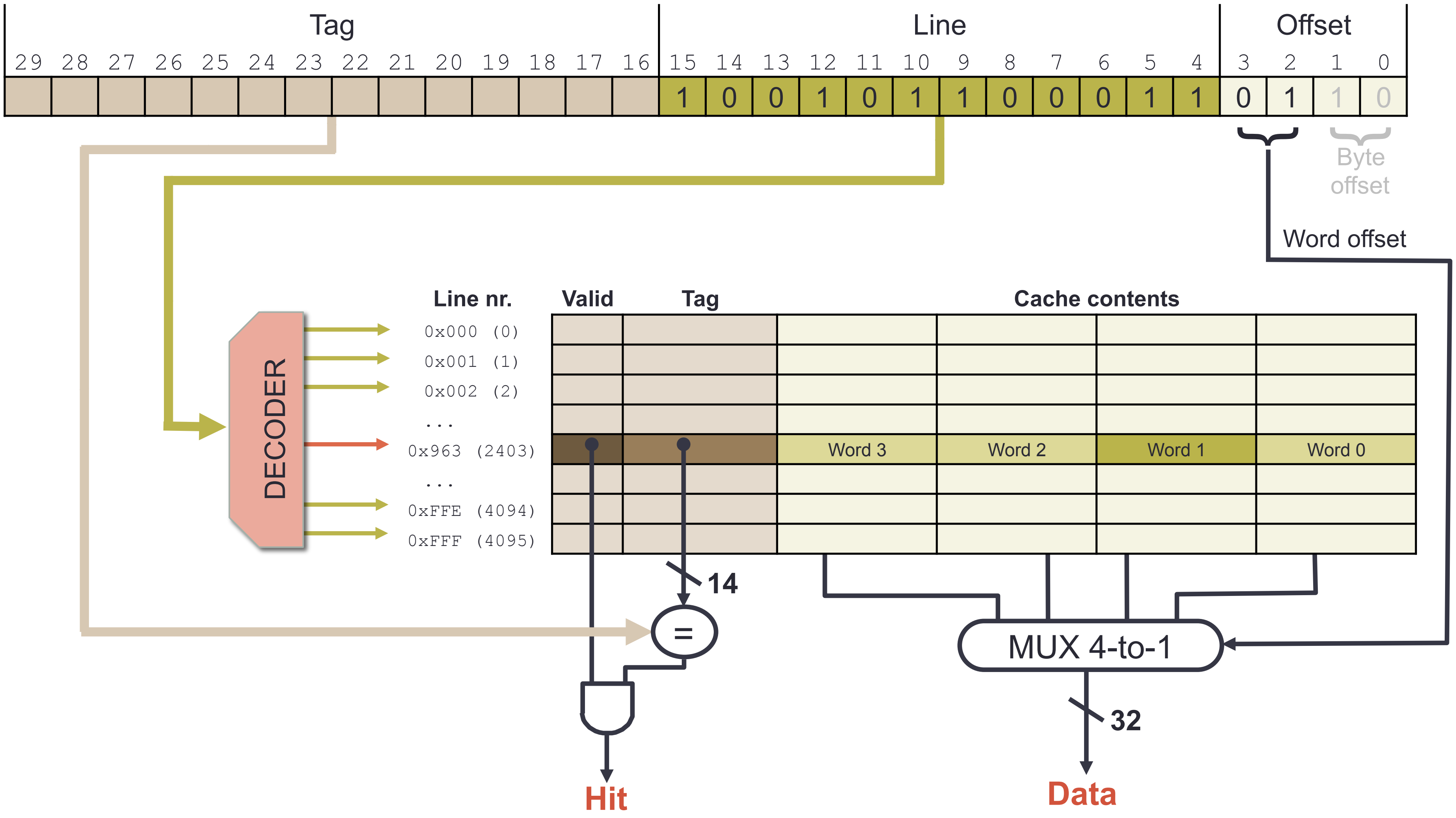
→ cache has 64 KB / 16 B = 4 K lines → 12 bits identify line

→ main memory has 1 GB / 16 B = 64 M blocks → 26 bits identify block

→ hence 26 – 12 = 14 bits are needed for the tag



Direct mapping implementation



Control information in cache

- The Valid and Tag fields are stored in memory
- Length of tags depends on cache size and address size
- Amount of control information in our example:
  - 4096 lines * (1 Valid bit + 14 Tag bits) = 61,440 b = 7,680 B
- The naming convention is to exclude the size of control fields, and to count only the size of data
  - So our cache is still named a 64 KB cache, although it needs a total of 65,536 + 7,680 = 73,216 B
  - Note that the control information takes 7,680 / 73,216 = 10.5 %

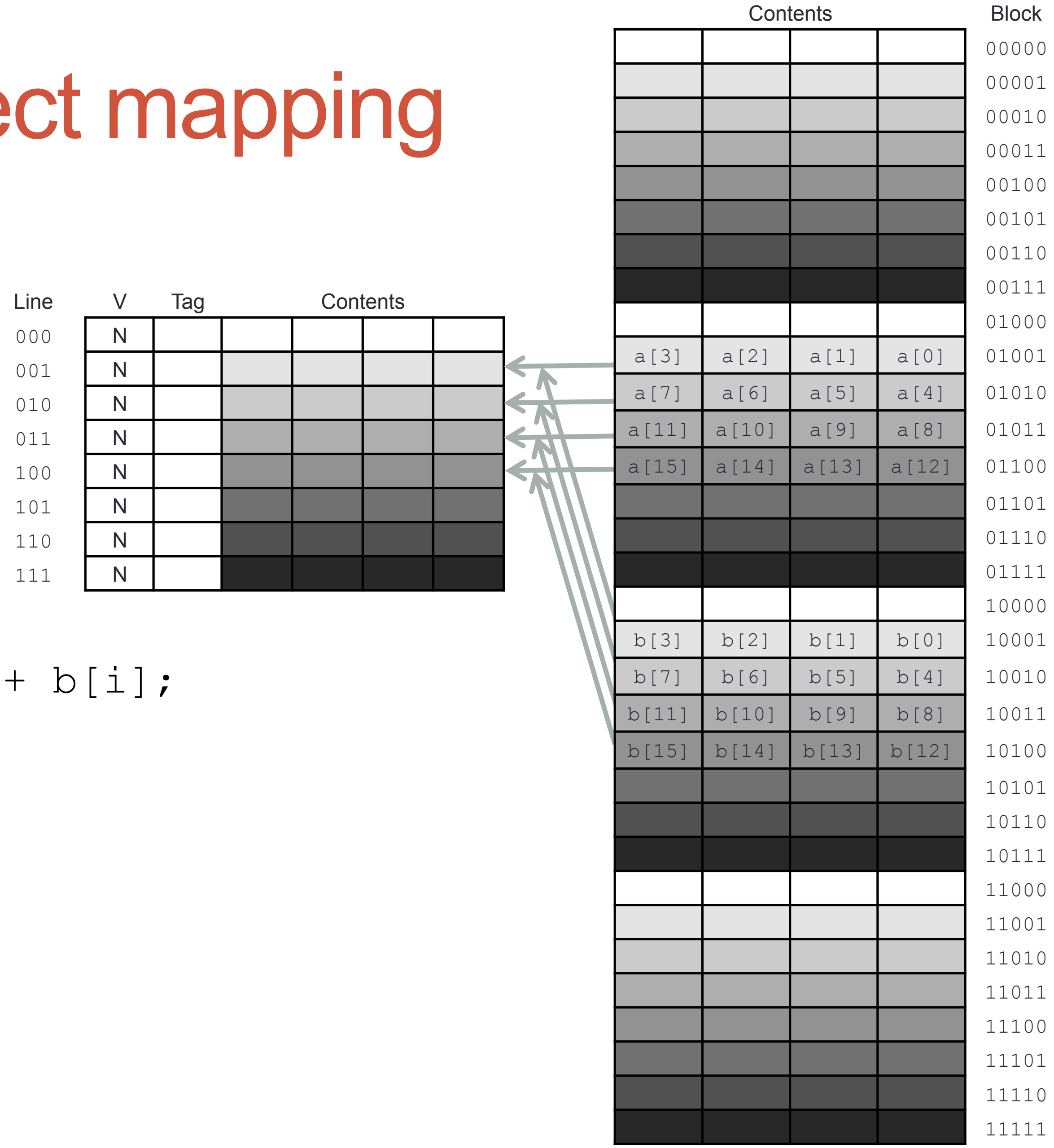
Hit rate and direct mapping

- Direct mapping gives only one choice for a block to be allocated in cache
- This has an impact on the attainable hit rate
- Pathological case:
    - Accessing two arrays in conflicting blocks (next slide)

Hit rate and direct mapping

Assume:

- `a[]` uses blocks 0x09..0x0C
- `b[]` uses blocks 0x11..0x14
- `i` and `sum` reside in registers
- Split cache




```
for (i=0; i<=15; i++)
    sum = sum + a[i] + b[i];
```

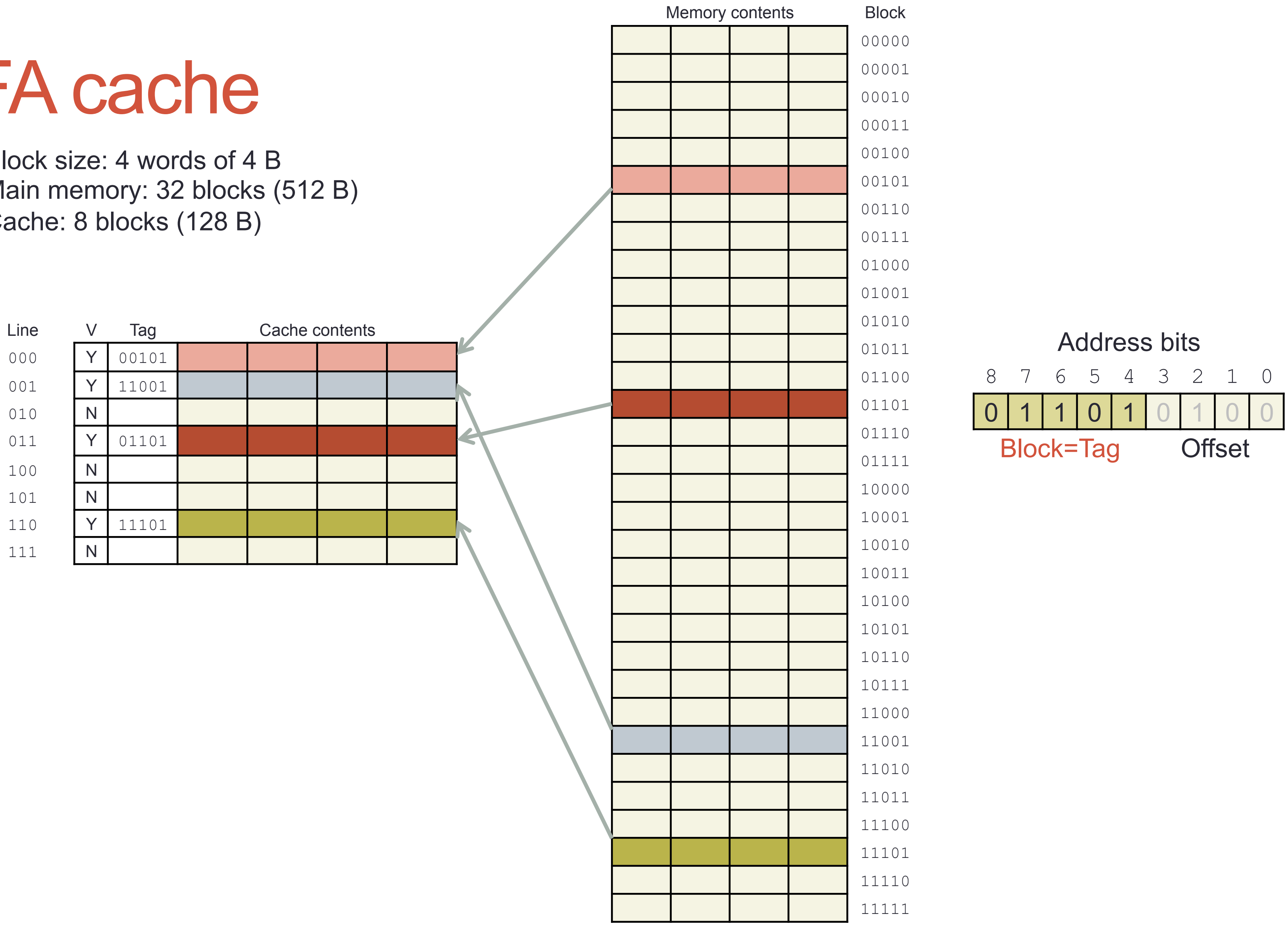
Hit rate?

Fully associative mapping

- A more flexible mapping scheme may provide a higher hit rate by reducing misses
- Direct mapping is at one extreme of the possibilities
- The other extreme is fully associative mapping
  - Any memory block can be placed in any cache slot
    - Placement of blocks in cache is only limited by cache capacity, and not by how data are mapped in memory
- To find a block in a fully associative cache, all cache entries have to be checked, because the block could reside anywhere

FA cache

Block size: 4 words of 4 B
Main memory: 32 blocks (512 B)
Cache: 8 blocks (128 B)



Fully associative mapping

- Although the most flexible mapping policy, full associativity has practical limitations
  - The amount of control data required, since tags are larger
  - The need to search the entire cache for a block (next slide)
- Amount of control data:

Blocks of 16 B (4 words of 4 B each); 64 KB cache; 1 GB addressing space:

→ cache has 64 KB / 16 B = 4 K lines

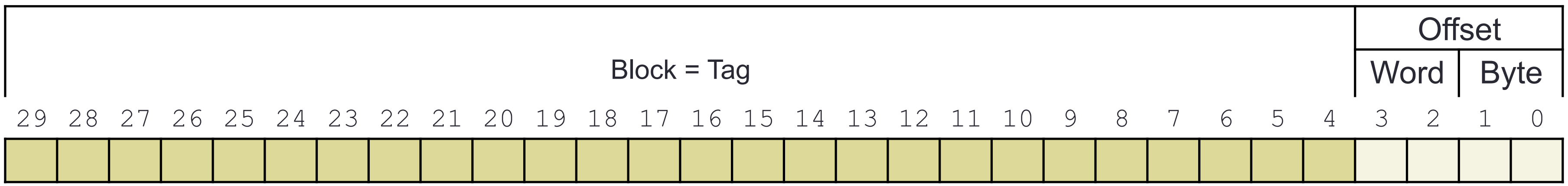
→ main memory has 1 GB / 16 B = 64 M blocks → 26 bits identify block

→ The amount of control information becomes:

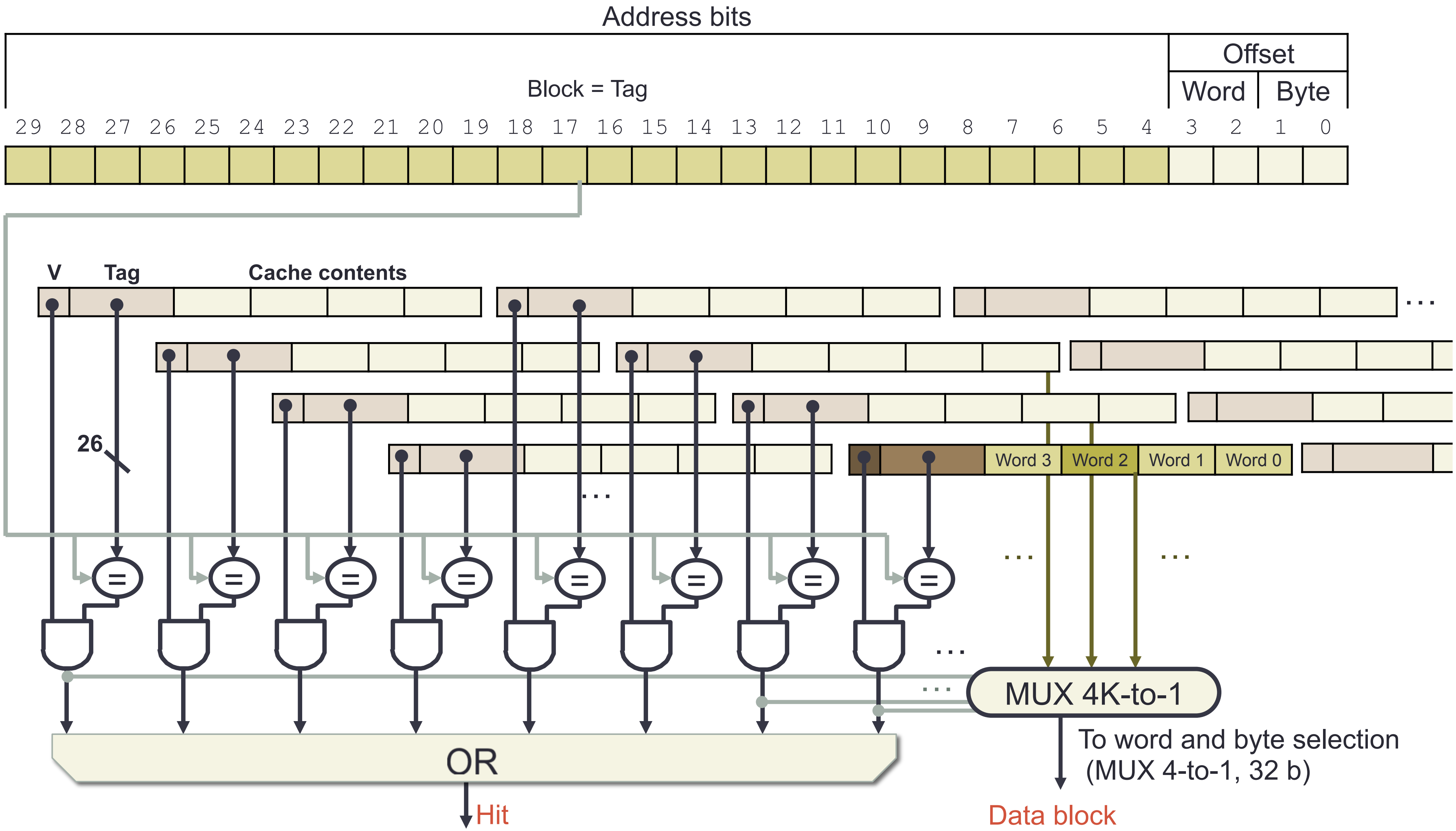
4 K lines × (1 Valid bit + 26 Tag bits) = 110,592 bits = 13,824 Bytes

→ which represents 13,824 / (13,824 + 65,536) = 17.4 % (vs. 10.5 % with direct mapping)

Address bits



Fully associative mapping implementation



Fully associative mapping - discussion

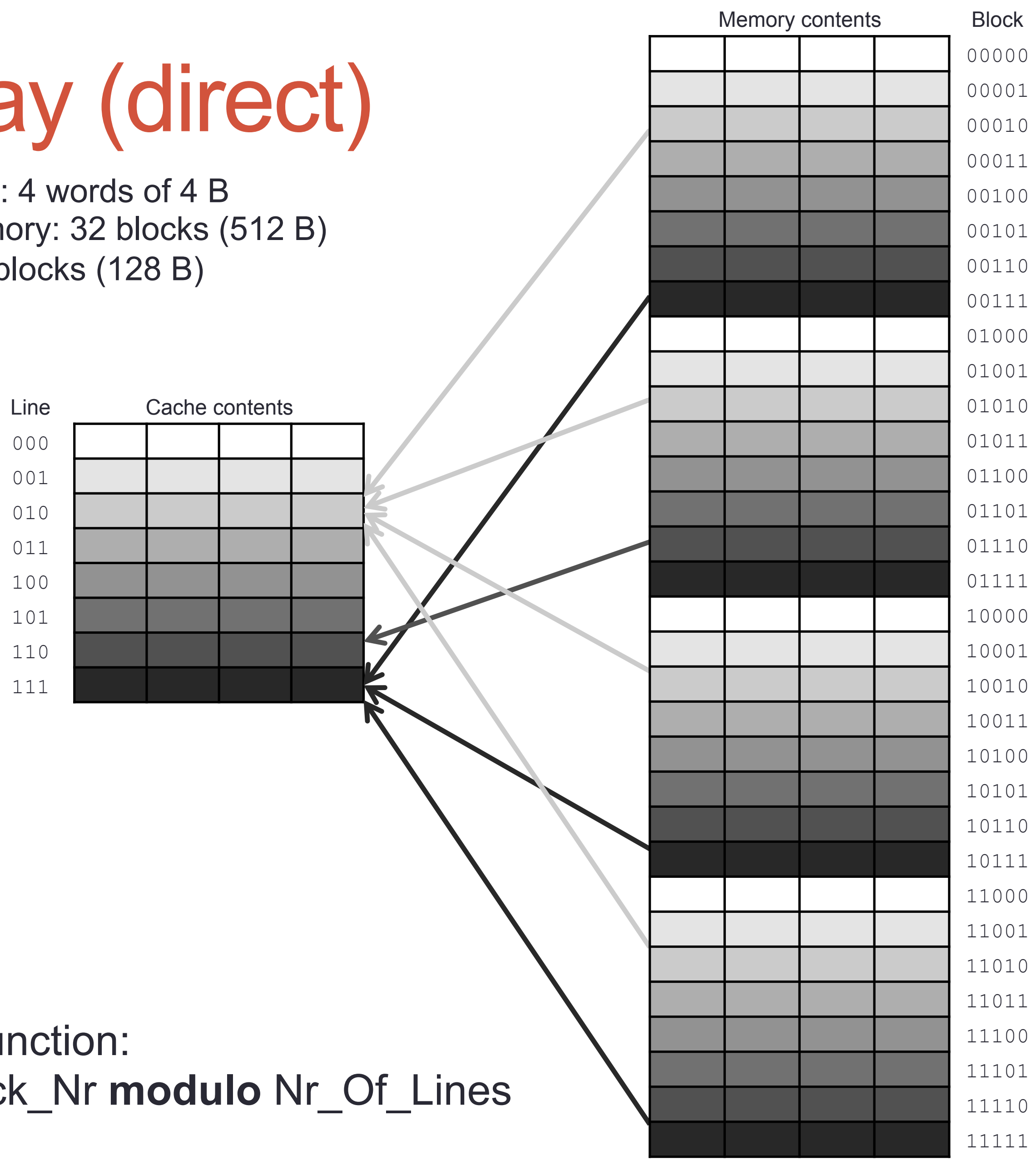
- On the positive side:
  - There are no conflict misses in a fully associative mapped cache
    - Misses occur only at startup and due to lack of free lines (capacity misses)
- On the negative side:
  - Needs more storage for control info
  - High hardware cost
- Conclusion:
  - Fully associative caches are only practical for caches of limited capacity (ie. a reduced number of lines)
    - And then capacity misses are more likely to occur…
- But there is an alternative between direct and fully associative mapping…

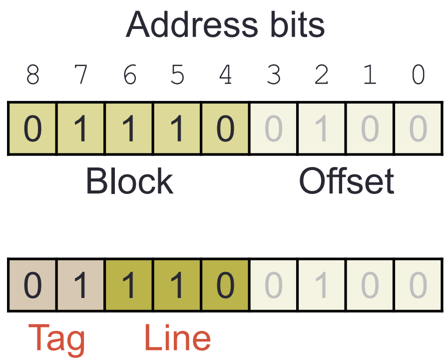
Set-associative mapping

- In a set-associative mapped cache, each memory block maps to a unique set of cache locations
  - The particular set can be derived from the block address
- In an n-way set-associative cache, lines are grouped in sets, each of n lines
- Each set of an n-way cache offers n alternative lines for the corresponding blocks
- In other words, a block is directly mapped to a set, and then all lines in the set are searched for a match
- All mapping schemes are variations of set-associativity
  - Direct mapping = 1-way set associative
  - Fully associative = n-way set associative, for a cache of n lines

1-way (direct)

Block size: 4 words of 4 B
Main memory: 32 blocks (512 B)
Cache: 8 blocks (128 B)



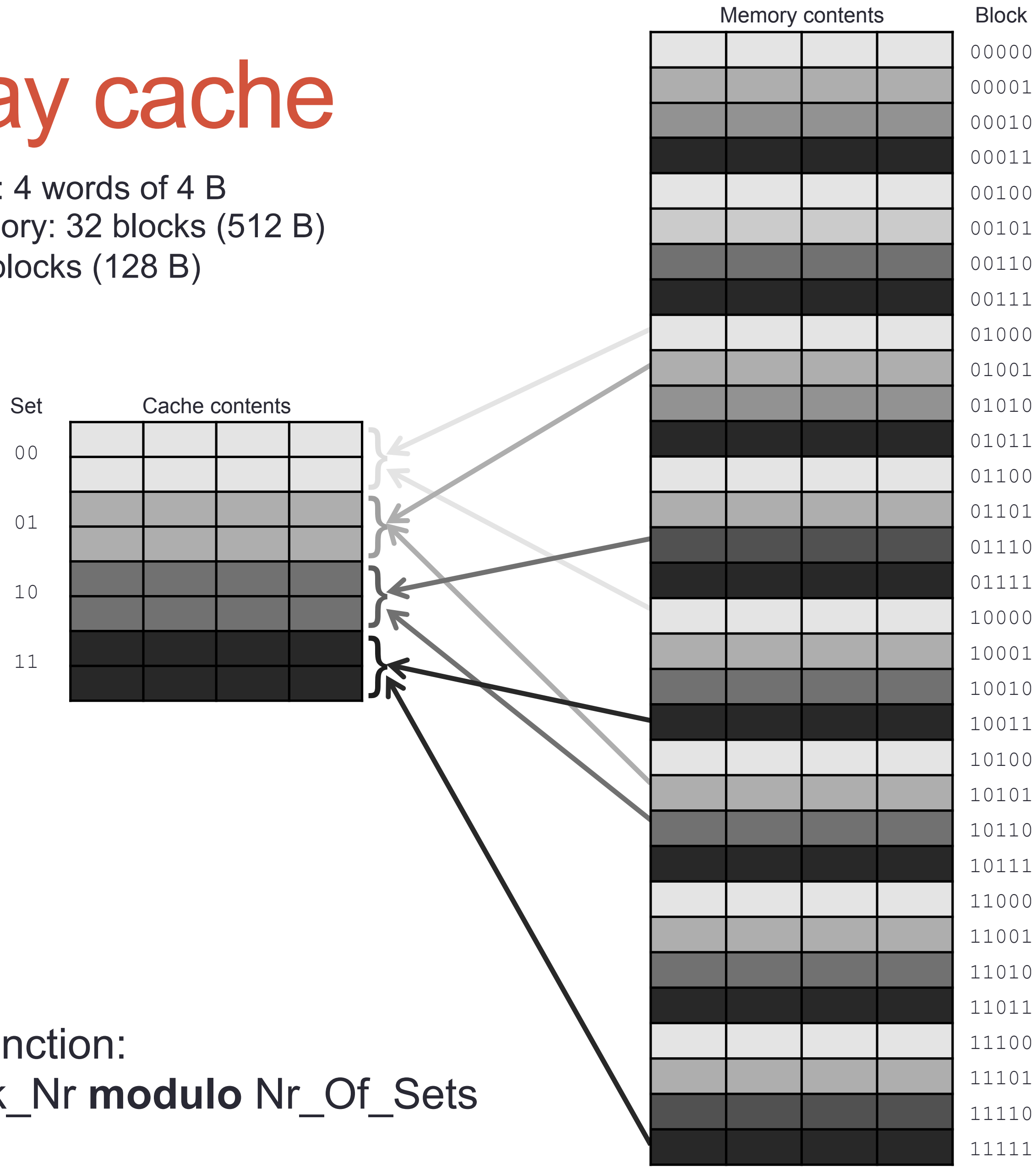


Line: Where to find me
Tag: My ID

Mapping function:
Line = Block_Nr **modulo** Nr_Of_Lines

2-way cache

Block size: 4 words of 4 B
Main memory: 32 blocks (512 B)
Cache: 8 blocks (128 B)

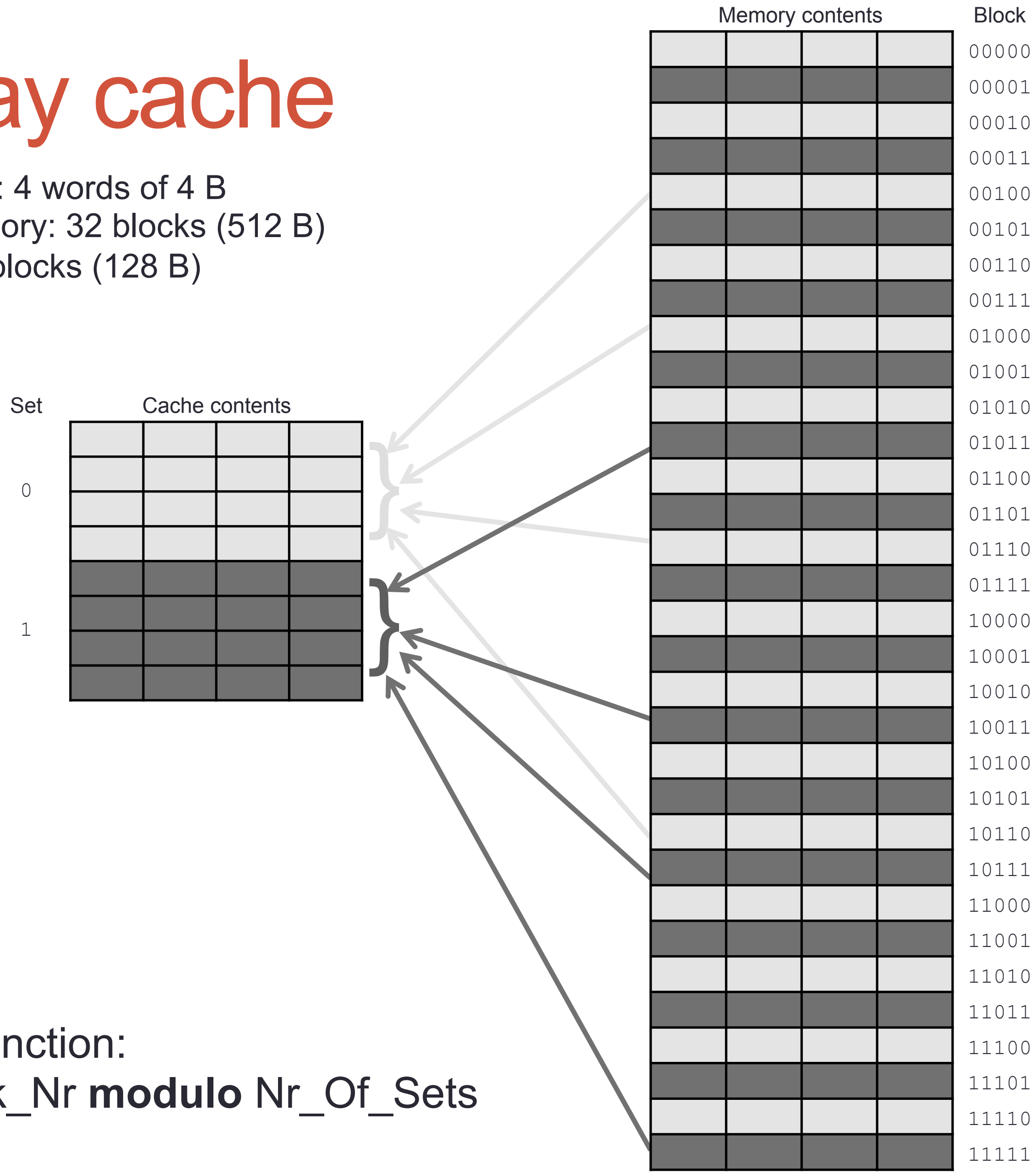


Set: Where to find me
Tag: My ID

Mapping function:
Set = Block_Nr **modulo** Nr_Of_Sets

4-way cache

Block size: 4 words of 4 B
Main memory: 32 blocks (512 B)
Cache: 8 blocks (128 B)



Address bits

8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	1	0	0

Block Offset

0	1	1	1	0	0	1	0	0

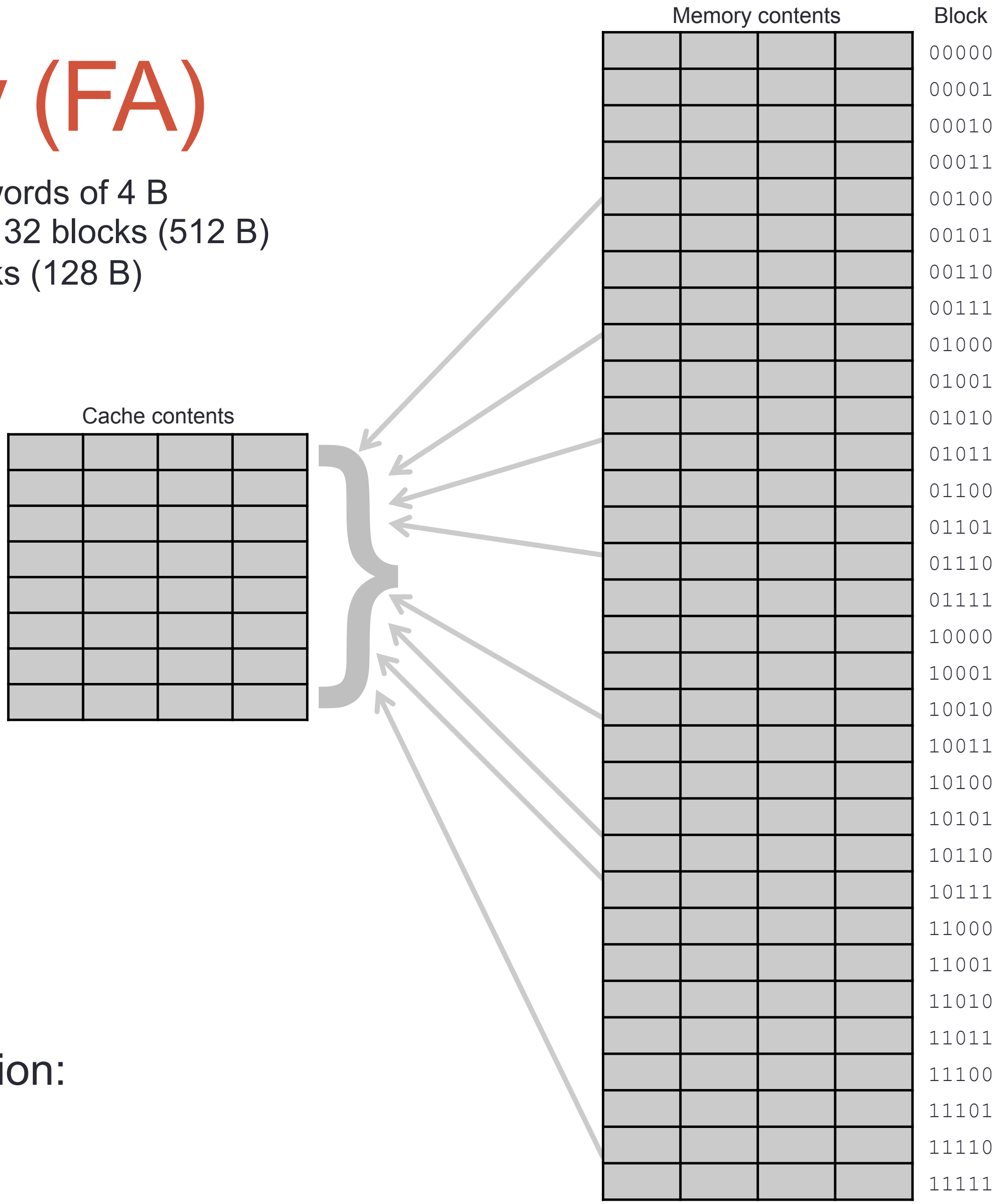
Tag Set

Set: Where to find me
Tag: My ID

Mapping function:
Set = Block_Nr **modulo** Nr_Of_Sets

8-way (FA)

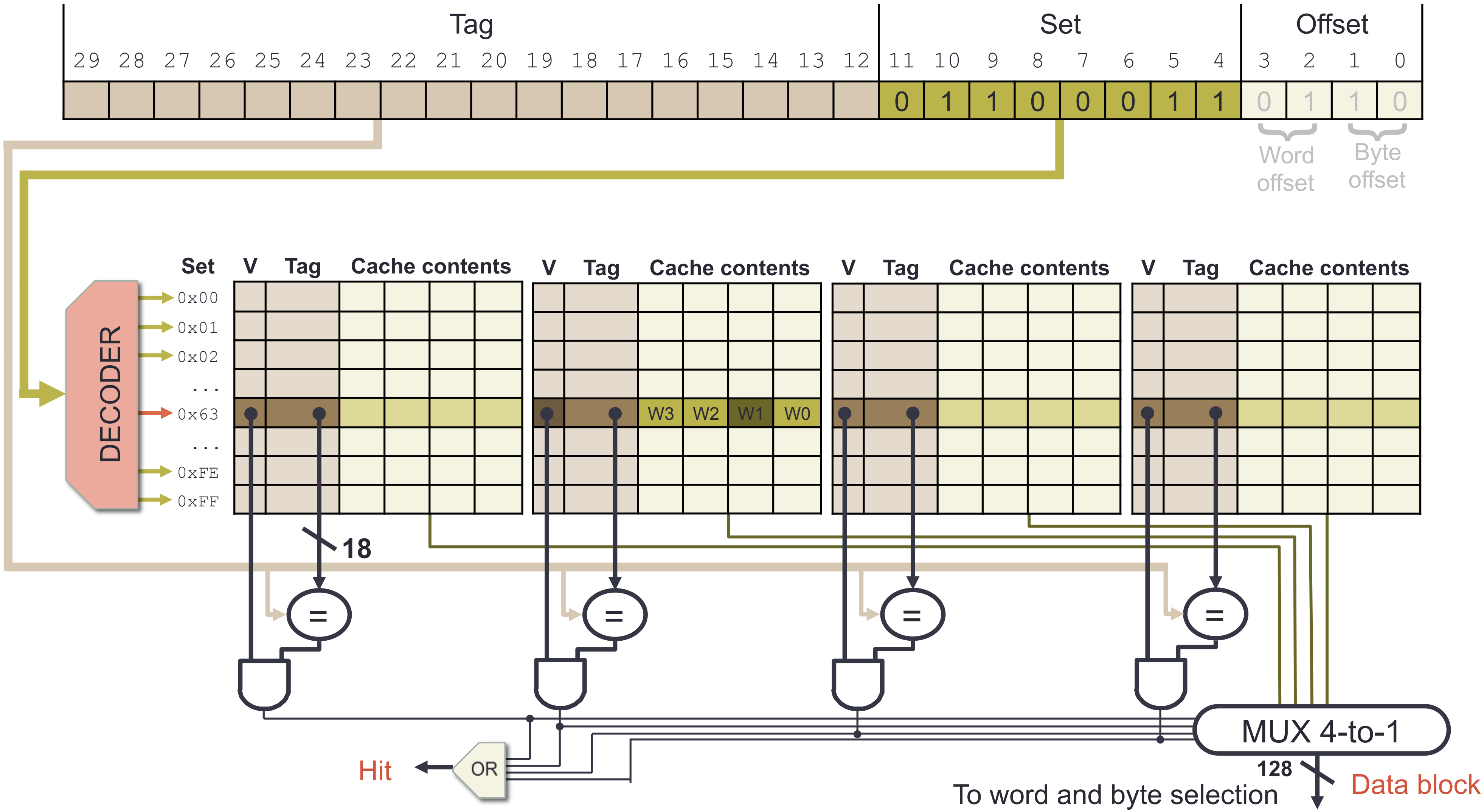
Block size: 4 words of 4 B
Main memory: 32 blocks (512 B)
Cache: 8 blocks (128 B)



Find me anywhere
Tag: My ID = Block nr.

Mapping function:
None

Set-associative implementation (4-way)



Implementation comparative

- Cache of 1 K Lines of four 32-bit words each (16 B)
- 1 GB addressing space (64 M Blocks in main memory)

Direct mapping (1-way)

- Set (Line): 10 bits
- Tag: 16 bits
- Control: 17 Kb (min)
- 1 deco 10-to-1024
- 1 comp 16 b
- 1 MUX 4-to-1, 32b

4-way associative

- Set: 8 bits
- Tag: 18 bits
- Control: 19 Kb (min)
- 1 deco 8-to-256
- 4 comp 18 b
- 1 MUX 4-to-1, 32 b
- 1 MUX 4-to-1, 128 b

Fully associative (1K-way)

- Set: 0 bits
- Tag: 26 bits
- Control: 27 Kb (min)
- No decoder
- 1024 comp 26 b
- 1 MUX 4-to-1, 32 b
- 1 MUX 1024-to-1, 128 b

Hands on: fill the gaps

CPU		Cache memory				
Addr.	W	Size	# Ways	Block size	# Lines	# Sets
32	32	32 KB	2	32 B		
32	32	8 KB			256	64
24	32	32 KB	1			
24	32		128	32 B		
36	64	32 KB	2	64 B		
36	64	8 KB				

Address bits		
Tag	Set	Offset
9	10	5
19	0	5
25	5	6

Calculating the hit rate

```
          .data 0x10000000                    # Data segment
V:        .word 2,6,5,7,8,3,4,1,9,0           # 10 words
          .globl __start

          .text 0x00400000                    # Code segment
                                              # 9 words
__start:
          lui  $t0, 0x1000
          addi $t1, $zero, 10
          addi $a0, $zero, $zero
loop:     lw   $t2, 0($t0)
          add  $a0, $a0, $t2
          addi $t2, $t2, 1                    # Useless...
          addi $t0, $t0, 4
          addi $t1, $t1, -1
          bnez $t1, loop
```

Addresses given by the CPU

Data segment

```
0x10000000
0x10000004
0x10000008
0x1000000C
0x10000010
0x10000014
0x10000018
0x1000001C
0x10000020
0x10000024
```

10 read accesses
(caused by `lw`)

Code segment

```
0x00400000
0x00400004
0x00400008
0x0040000C
0x00400010
0x00400014
0x00400018
0x0040001C
0x00400020
```

Addresses of loop instructions

3+10×6=63 read accesses
(caused by instruction fetch)

Hit rate – data cache

- Assume 32 KB direct cache, with block size = 16 Bytes
  - Data takes three memory blocks
  - 17-bit tag; 11-bit line, 4-bit offset

```
            Tag                Line     Offset
00010000000000000 00000000000 0000
00010000000000000 00000000000 0100
00010000000000000 00000000000 1000
00010000000000000 00000000000 1100

00010000000000000 00000000001 0000
00010000000000000 00000000001 0100
00010000000000000 00000000001 1000
00010000000000000 00000000001 1100

00010000000000000 00000000010 0000
00010000000000000 00000000010 0100
```

$$H = \frac{10-3}{10} = 0.7 = 70\%$$

Hit rate – Code cache

- Assume now 64 KB direct cache with block size = 16 Bytes
  - 18-bit tag; 12-bit line; 4 bit offset

```
       Tag               Line      Offset
000000001000000  000000000000  0000
000000001000000  000000000000  0100
000000001000000  000000000000  1000
000000001000000  000000000000  1100

000000001000000  000000000001  0000
000000001000000  000000000001  0100
000000001000000  000000000001  1000
000000001000000  000000000001  1100

000000001000000  000000000010  0000
000000001000000  000000000000  1100
000000001000000  000000000001  0000
000000001000000  000000000001  0100
```

loop instructions

$$H = \frac{63-3}{63} = 0.95 = 95\%$$

Types of misses

The following are mutually exclusive:

- **Compulsory misses**
  - Occur on the first reference to a block of memory – block has not yet been in cache
  - Impossible to get rid of these, regardless of mapping.
- **Capacity misses**
  - Occur when cache is not large enough to hold every block needed by a program
  - A larger cache reduces capacity misses, but larger capacity → slower
- **Conflict misses**
  - Only in direct or set-associative caches, but not in full-associative caches
  - Occur when my corresponding set is full, but not the cache
  - Larger number of ways reduces conflict misses, but larger n-ways → slower

Handling reads

- Read hit
  - We get the data in the cache access time, ie., within the same processor cycle of the load instruction
- Read miss
  - We need to access main memory (or the next level…) and pay the corresponding time penalty
  - We'll assume a simple model whereby the miss penalty equals the access time of the next level
    - For a DRAM, $t_{RCD} + t_{CL}$
    - For a next-level cache, its access time

Handling writes

- Writes are not so simple…
- Write hit:
  - If we write to cache only, then memory and cache may become inconsistent

Instruction	Cache line for X	Mem location X
`la $a0,X`	--	0x15
`lw $t0,($a0)`	0x15	0x15
`addi $t0,0x10`	0x15	0x15
`sw $t0,($a0)`	0x25	0x15

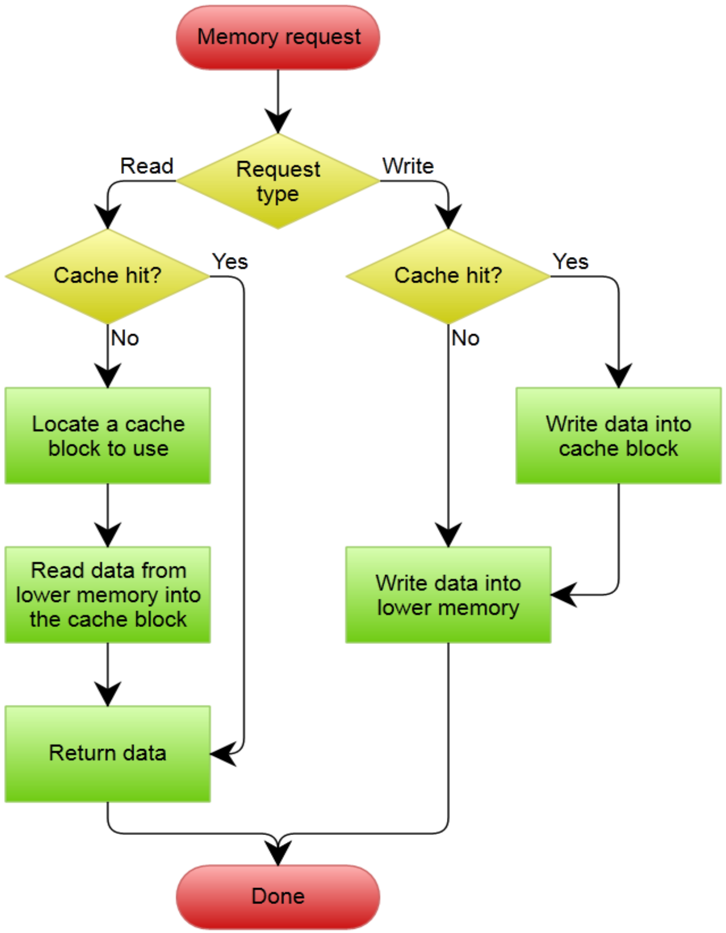
Write policies

- Policies for handling write hits
  - Write through
    - Data is written both to cache and memory (or next level)
    - Simple to implement, but affects performance
      - The time penalty can be mitigated by using write buffers
  - Write back
    - Data is written to cache only
    - Data is updated in next level when the block is replaced in cache
      - This requires an additional control bit, the so called dirty bit, that marks the block as modified
    - More complex to implement, but better performance

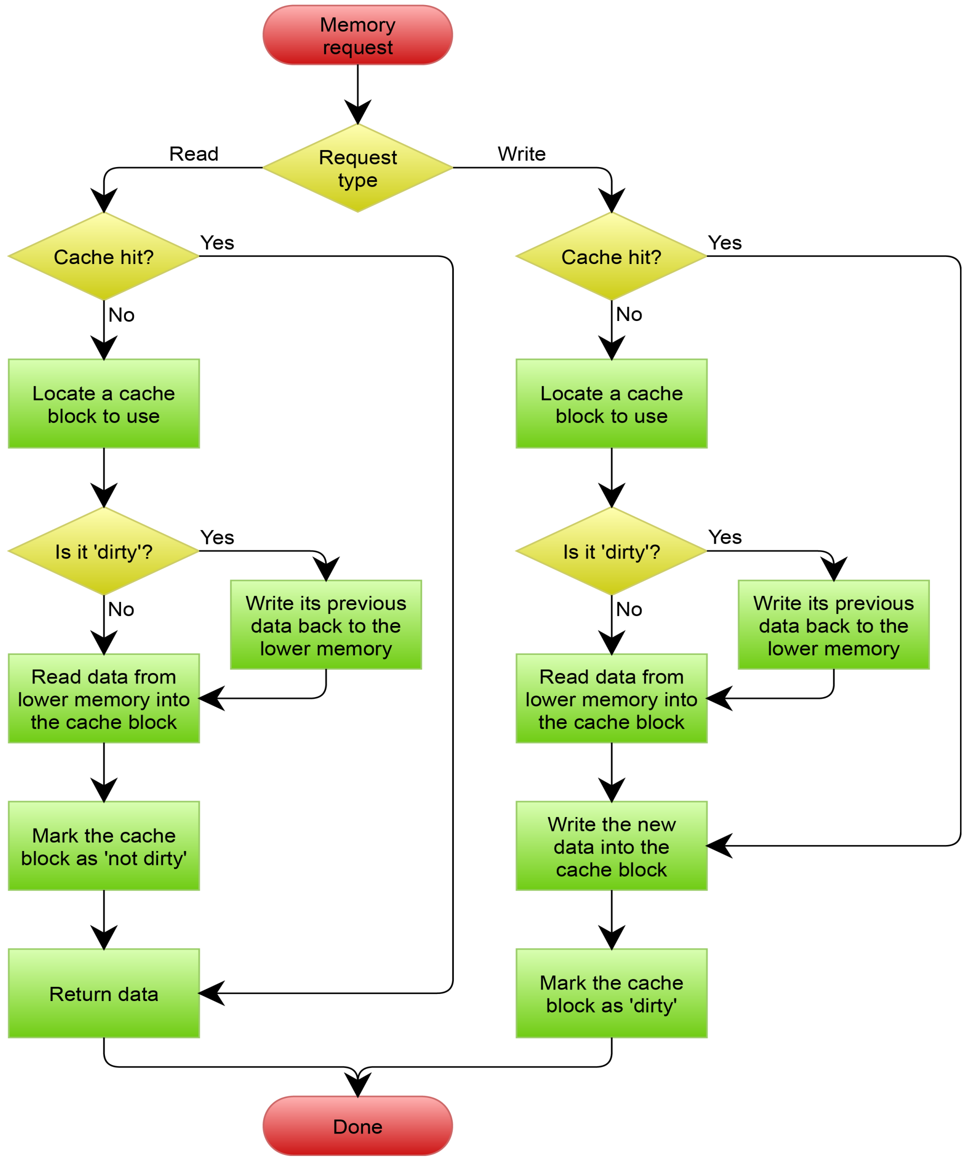
Write policies

- Policies for handling write misses
  - Write allocate
    - Miss handling is similar to read miss: offending block is brought to cache
  - No write allocate
    - Data is only written in next level; the written block is **not** brought to cache
- Write hit and write miss policies are usually paired as:
  - Write through and no write allocate
  - Write back and write allocate

Summary of policies



Write through with no write allocation



Write back with write allocation

Images taken from wikipedia:cache

Replacement algorithms

- Goal: to decide which block to replace when a set is full
- Implemented by hardware (should not be too complex)
- Recall that direct mapping (1-way) does not require a victim selection since there is only one possibility
- Most used algorithms
  - Random
    - A random victim is selected. No guarantee for optimal decision. Simple to implement. Performs only slightly worse than LRU
  - LRU, Least Recently Used
    - Needs to time-stamp all lines upon every single access
      - Requires n bits for the time stamp, where $n = \log_2(\text{Number\_Of\_Ways})$
    - Its complexity grows with the number of ways
      - Only practical for 2- or 4-way caches

LRU replacement in detail

- Each line (or way) in the set has an associated counter of n bits, with $n = \log_2(\text{Number\_Of\_Ways})$
- Counters are maintained as follows:
  - Upon a cache hit to line j
    - Add 1 to all counters of valid lines whose values are strictly lower than j's
    - Set j's counter to 0
  - Upon a cache miss on a set that is full
    - Select the line with the highest counter as the victim (say line j)
    - Add 1 to all counters except j
    - Set j's counter to 0

Review of control information

- We have considered two pieces of control data so far
- Finally we may need the following fields:
  - Valid: One bit.
  - Tag: Depends on sizes of cache and lower level, and mapping
  - Dirty: One bit. Only for write-back caches
  - LRU: n bits, $n = \log_2 (Nr\_Ways)$. Only if LRU replacement is used
- All these bits must be replicated for all cache lines

Improving performance

- Average access time is given by

$$\boxed{T_m = H \times T_{hit} + \left(1 - H\right) \times T_{miss}}$$

- Possible improving approaches
  - Reduce T_{hit}
    - A technological issue
  - Reduce the miss rate (1-H)
    - Increasing associativity is limited by implementation complexity
  - Reduce the miss penalty
    - Use multilevel caches to reduce T_{miss} (L1, L2, L3)
    - Additionally, use wider busses between cache levels

Multilevel caches

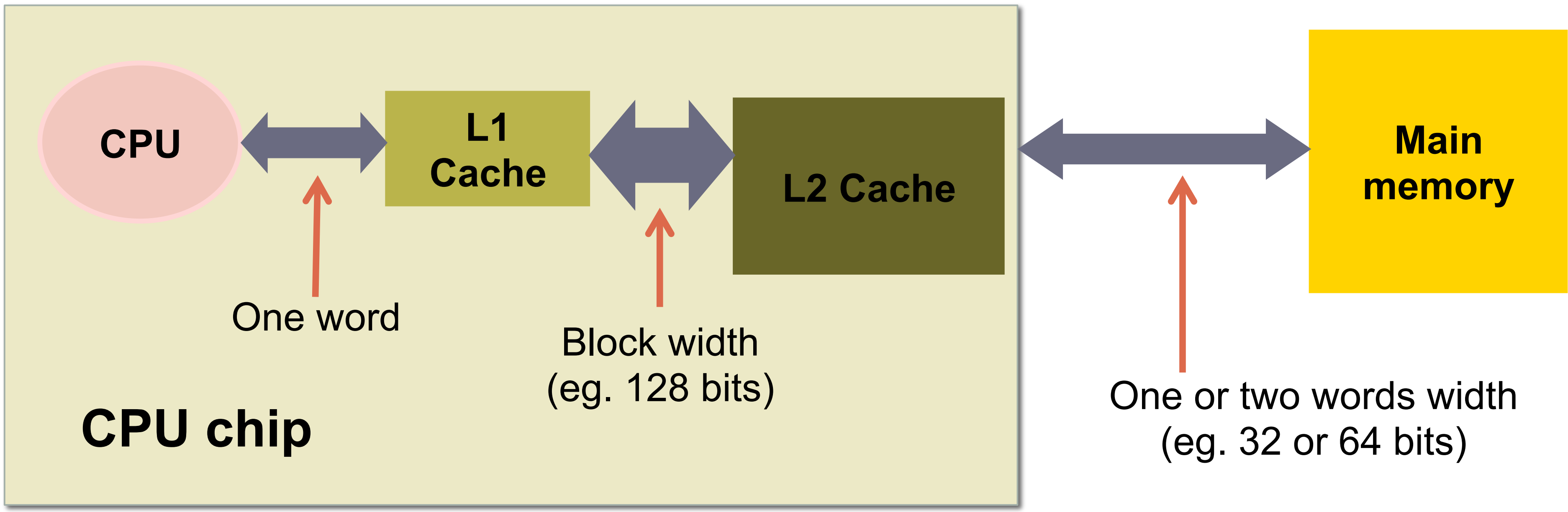
- L2 cache: larger, but slower than L1
- Now T_m will be:

$$\boxed{T_m = H_1 \times T_{L1} + \left(1 - H_1\right) H_2 \times T_{L2} + \left(1 - H_1\right)\left(1 - H_2\right) T_{miss}}$$

- Expected benefit? Assume:
  - L1: H_1 = 90%; Access time = 1 CPU clock cycle
  - L2: H_2 = 90%; Access time = 10 cycles
  - Main memory: Access time = 100 cycles
- Without L2: T_m = 0.9×1 + + 0.1×100 = 10.9 cycles
- With L2: T_m = 0.9×1 + 0.09×10 + 0.01×100 = 2.8 cycles
- So T_m is reduced to only 25.7%

Multilevel caches

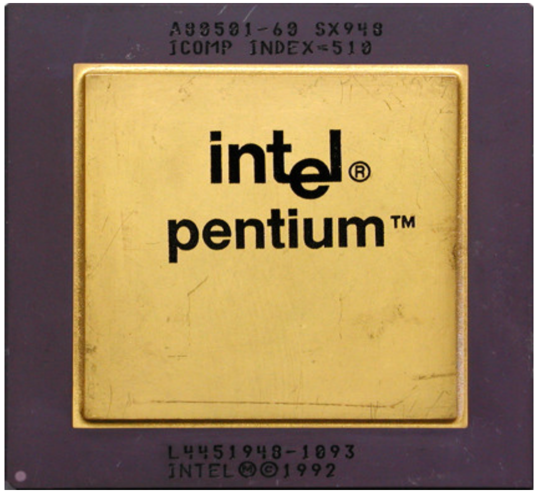
- When L2 is on the CPU chip, the number of pins is not an issue and the L1-L2 bus can be as wide as one full block
- This reduces the miss penalty on L2

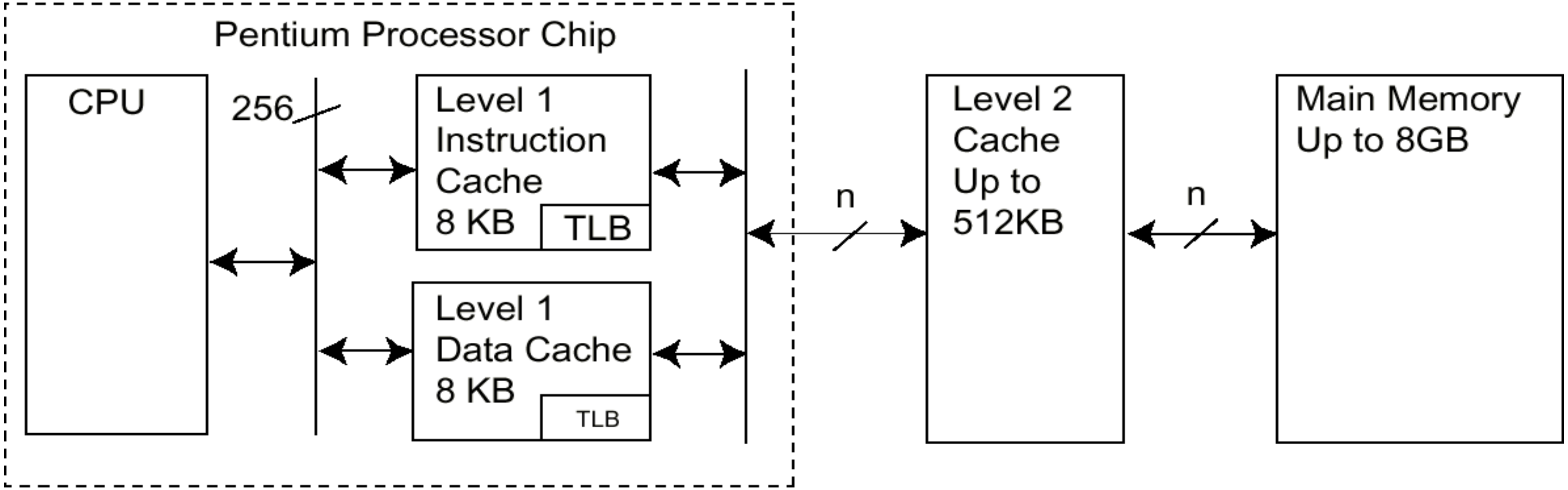


Multilevel caches

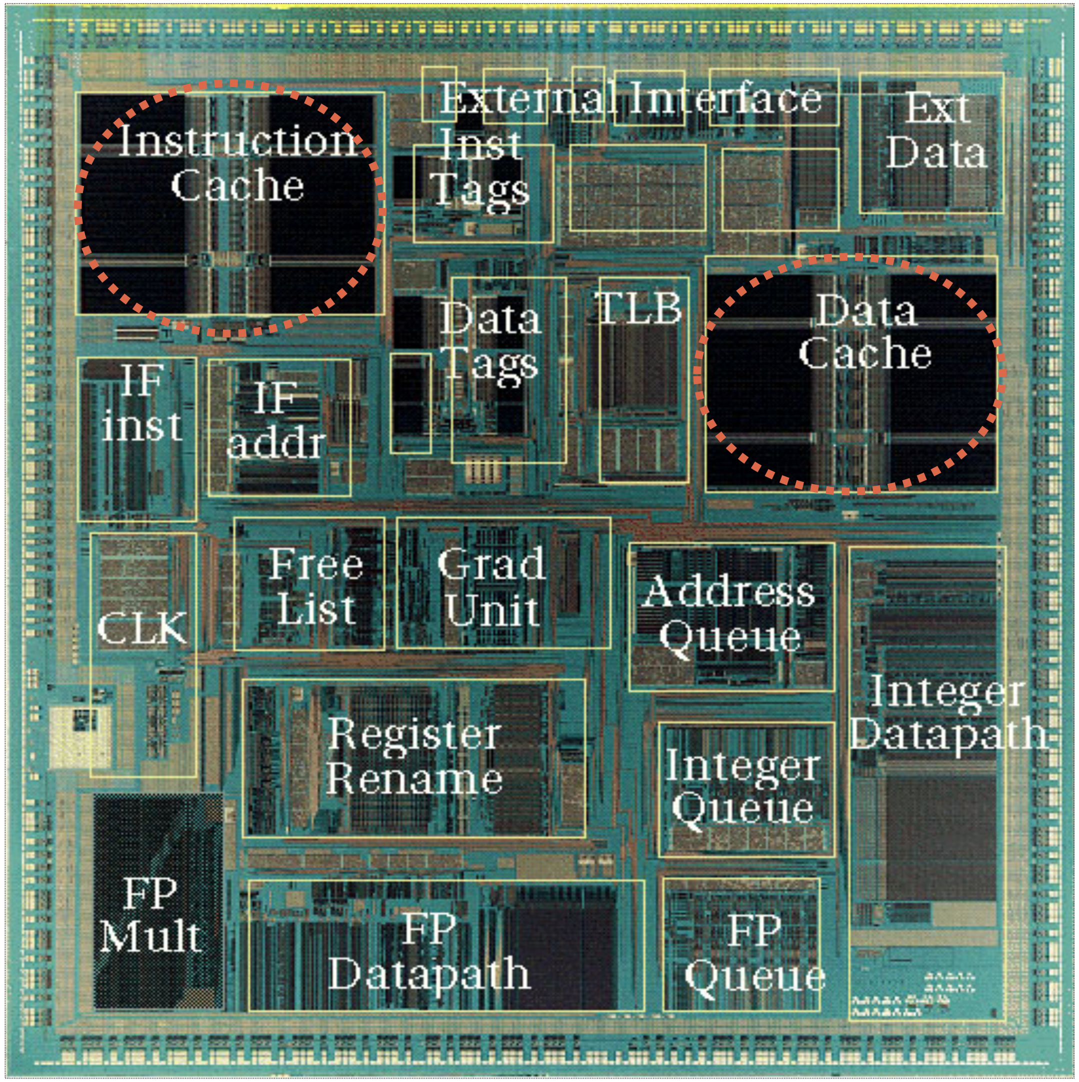
- Handling misses in a multilevel system. Two options:
  - Sequential
    - An L1 miss is followed by an L2 access
    - The miss penalty is $T_{L1} + T_{L2}$
  - Parallel
    - Both L1 and L2 are accessed in parallel
    - An L1 hit aborts the L2 access
    - The miss penalty is T_{L2}
  - Unless otherwise specified, we'll assume parallel access

Example systems: Intel Pentium





Example systems: MIPS R10000



L1: code (32 KB, 2 ways)

data (32 KB, 2 ways)

L2: off-chip, 2 ways (0.5-16 MB)

Evolution of MIPS caches

ID-MHz	Year	L1				L2			L3		
		Code	Data	Map	Where	Size	Map	Where	Size	Map	Where
R3000-33	1988	32 KB	32 KB	Direct	Off-chip						
R4000-100	1991	8 KB	8 KB	Direct	On-chip	1 MB	Direct	Off-chip			
R10000-250	1995	32 KB	32 KB	2-way	On-chip	4 MB	2-way	Off-chip			
RM7000-250	1998	16 KB	16 KB	4-way	On-chip	256 KB	4-way	On-chip	1 MB	Direct	Off-chip

Virtual memory

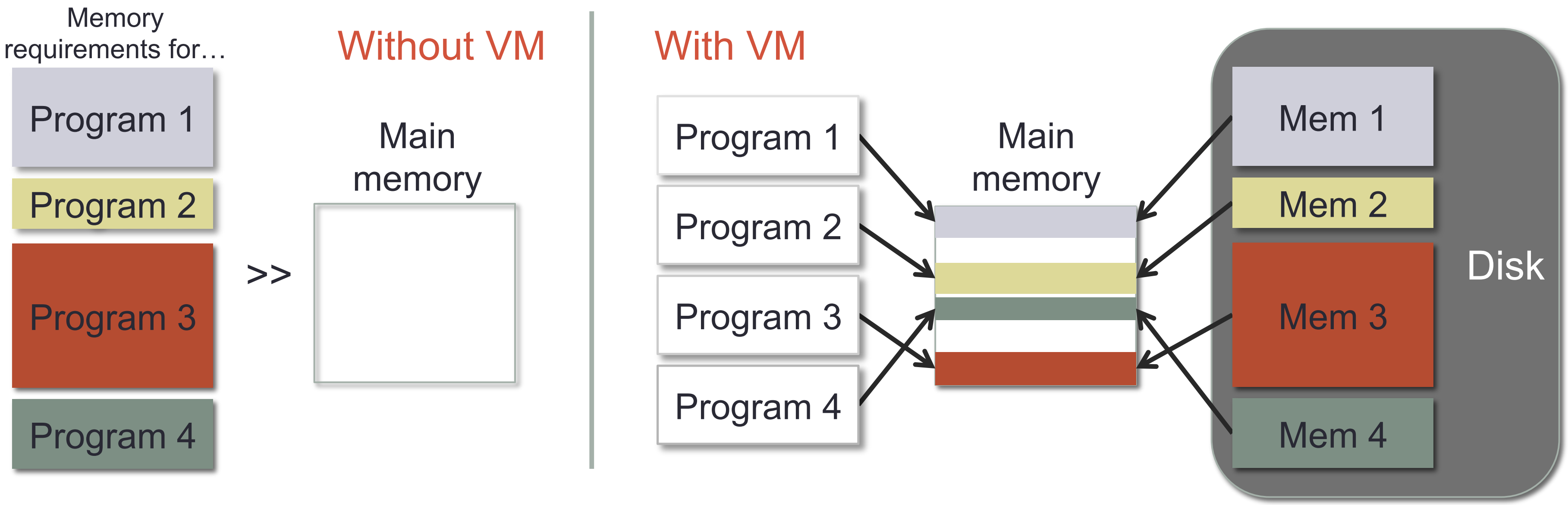
- Concept and motivation
- Virtual addressing
- Address translation

VM: concept and motivation

- The VM technique extends the addressable memory by making use of secondary storage devices – in practice, hard disks
- The motivation for using VM is twofold:
  - 1. Remove the programming burdens of a limited amount of main memory
  - 2. Allow efficient and safe sharing of memory among multiple programs
- 1. Before VM, if a program became too large for memory, programmers had to make it fit by allocating pieces of the total required memory (overlays) when they were needed

VM: concept and motivation

- 2. Multiprogramming: the capacity issue



- Without VM, there's no room in main (physical) memory
- With VM:
  - Disk serves as storage for all programs' needs
  - Only the active portions reside in physical memory – the rest remain on disk
    - Locality enables VM; VM enables efficient use of the CPU and memory

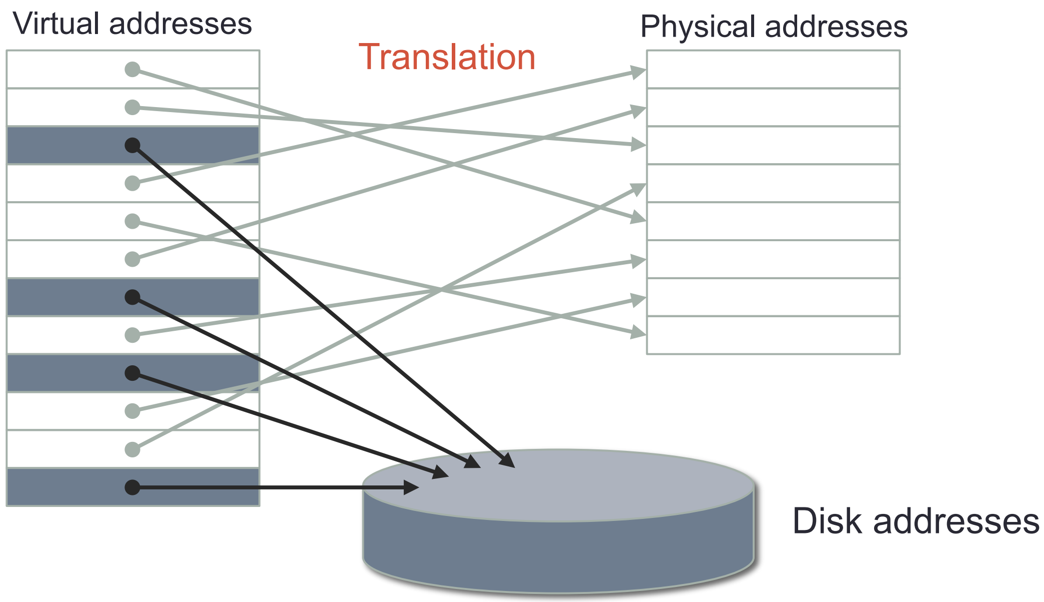
Virtual addresses

- Examples:
  - MIPS R2000 programs may use 32 bit addresses → 4 GB, but the physical memory available may be smaller
  - AMD Opteron X4 and Intel Core i7 use 48 bit addresses → 256 TB !

- Each program is compiled into its own address space
  - a separate range of addresses accessible only to this program
- The program's addresses are referred to as virtual addresses
  - VM implements the translation of virtual addresses to physical addresses
  - This translation enforces protection of a program's address space from other programs
    - The translation ensures that programs' addresses will not collide

Address translation (paging)

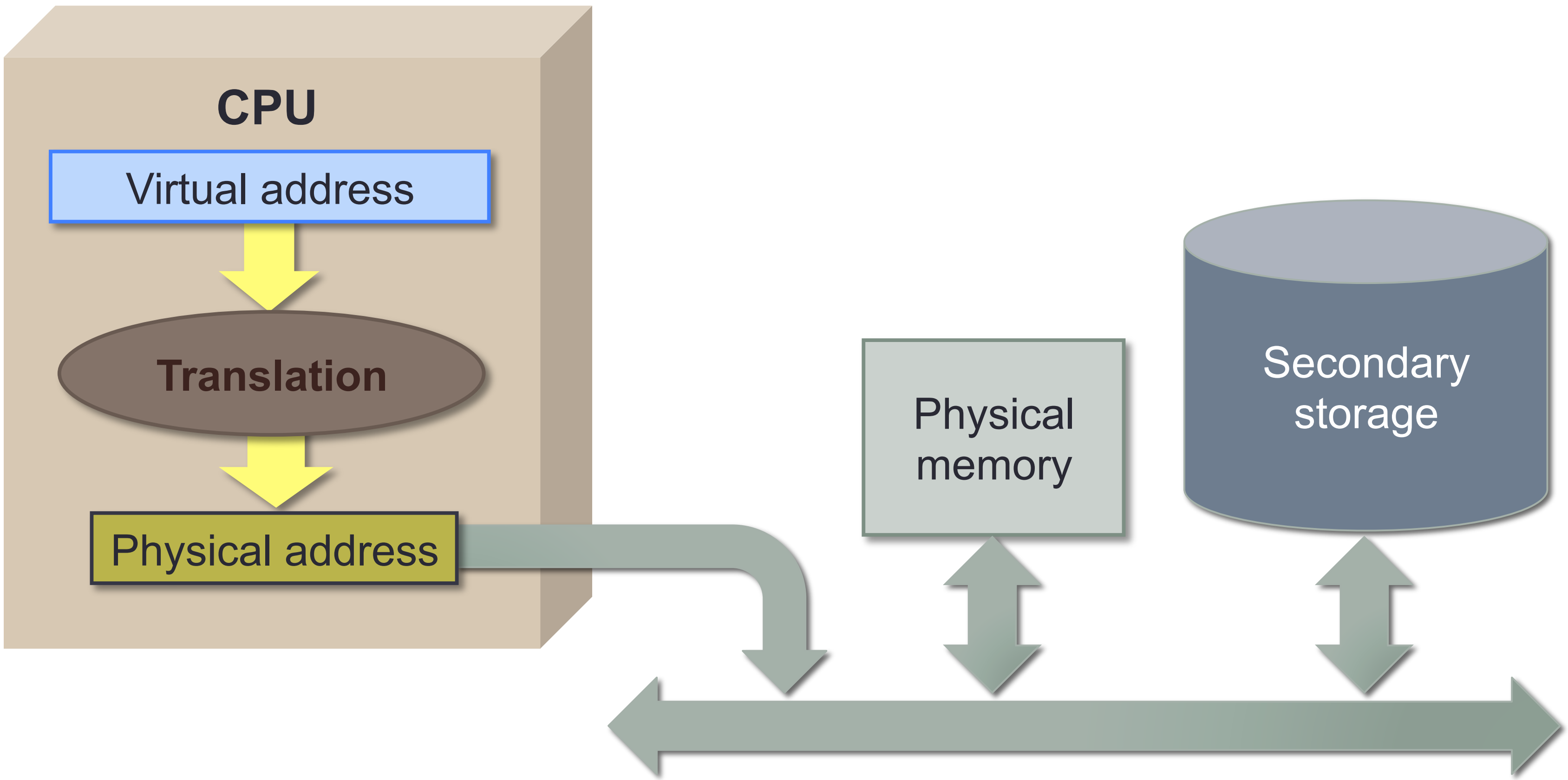
- Although the principles of caches and VM are resembling, the terminology used differs, for historical reasons:
  - A VM block is called a page – typical page size is 4 KB
  - A VM miss is called a page fault – a page fault implies a heavy penalty
- All virtual addresses need be translated to physical addresses



Further advantage:
A program's address space needs not be contiguous.
This simplifies program loading to the OS (reallocation)

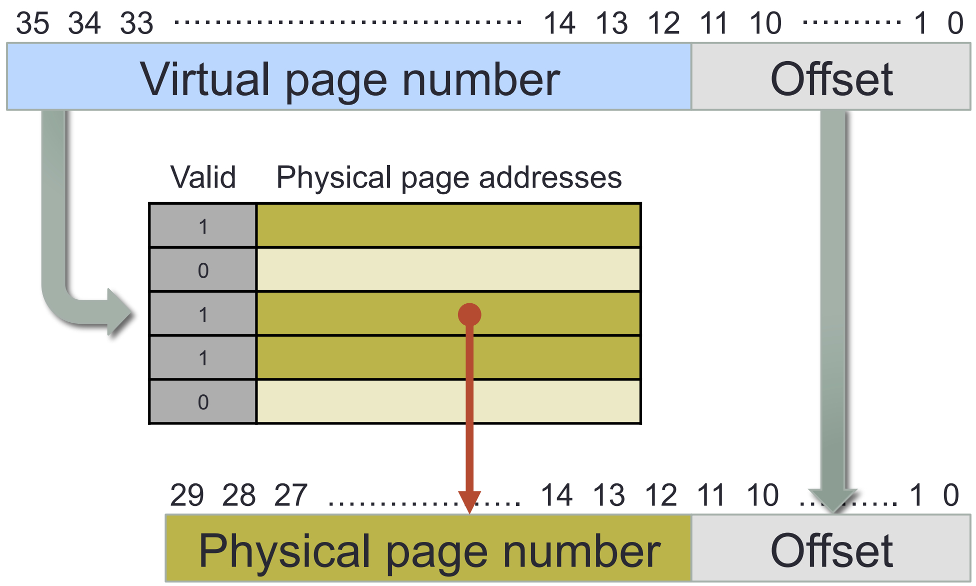
Address translation

- Programs' virtual addresses need be translated before they are sent through the physical address bus



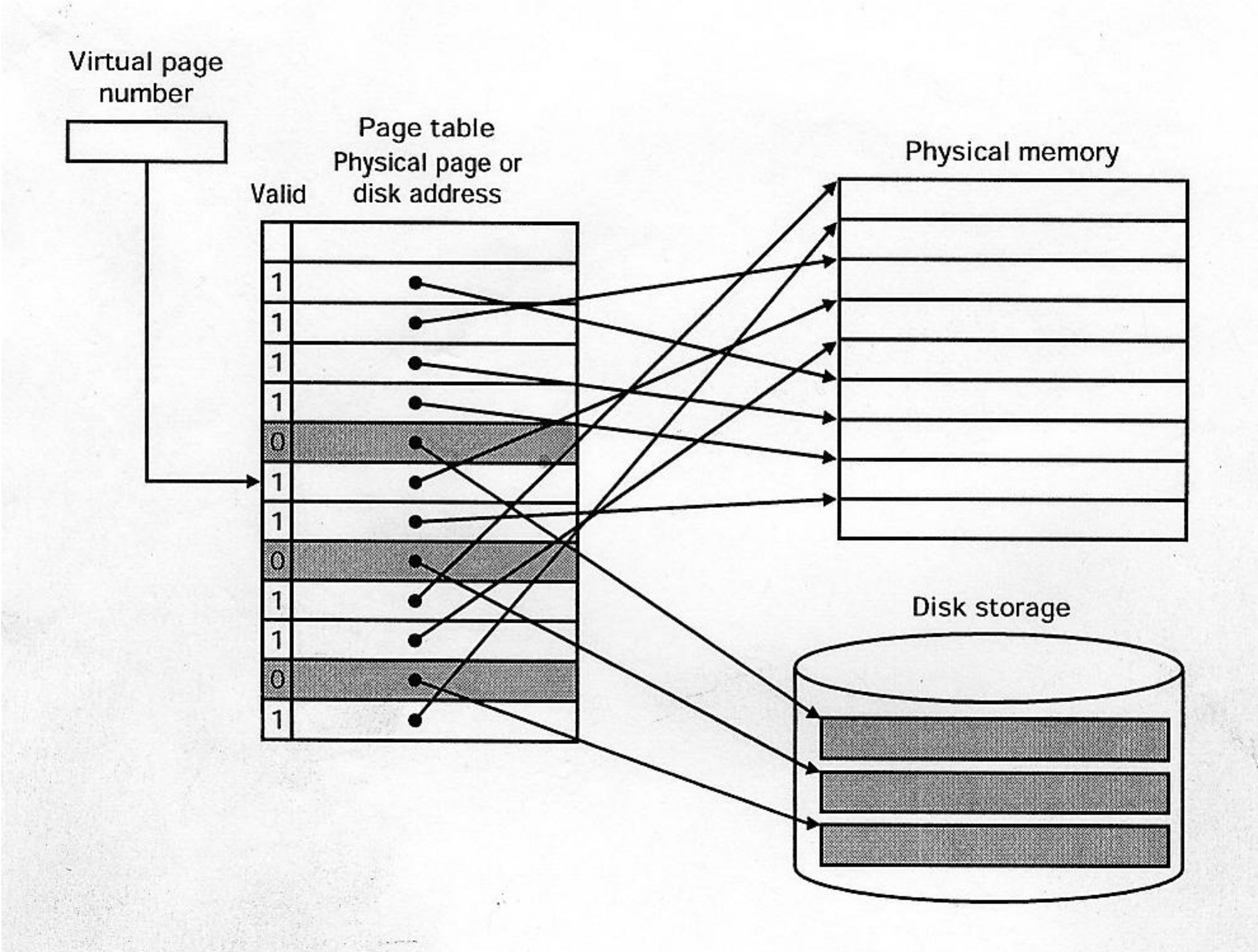
Translation: the page table

- Translation must be fast – it's in the critical path to mem. access
  - A page table in memory contains all needed translations
- A program's virtual address is structured in two fields:
  - Virtual page number (cache equivalent to block number)
  - Offset within the virtual page (cache equivalent to offset within block)



- Example:
  - Page size = 2^{12} B = 4 KB
  - 2^{36} B = 64 GB virtual space
  - 2^{30} B = 1 GB physical

Interpretation of the page table



Design choices for VM systems

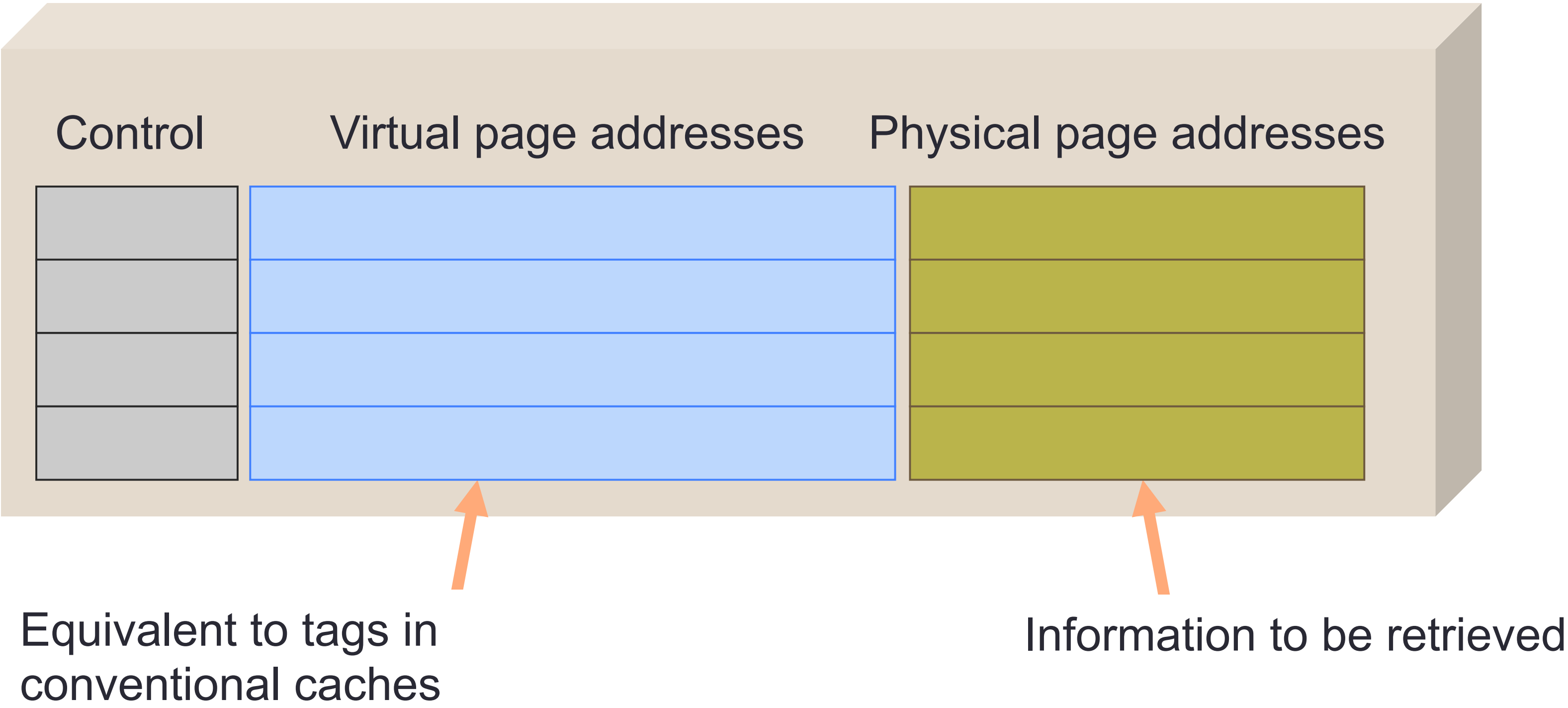
- Mostly motivated by the high cost of page faults
  - Disk access is about 50,000 times slower than memory access
- Pages should be large enough to amortize the high access time
  - Typically 4 KB to 16 KB
- Reducing the *fault rate* is crucial
  - The primary technique used is fully associative mapping
- Page faults can be handled in software
  - The overhead is small compared to disk latencies – replacement algorithms can be clever: increasing the hit rate pays back the cost
- Write through doesn't work – VM uses write back

Improving address translation

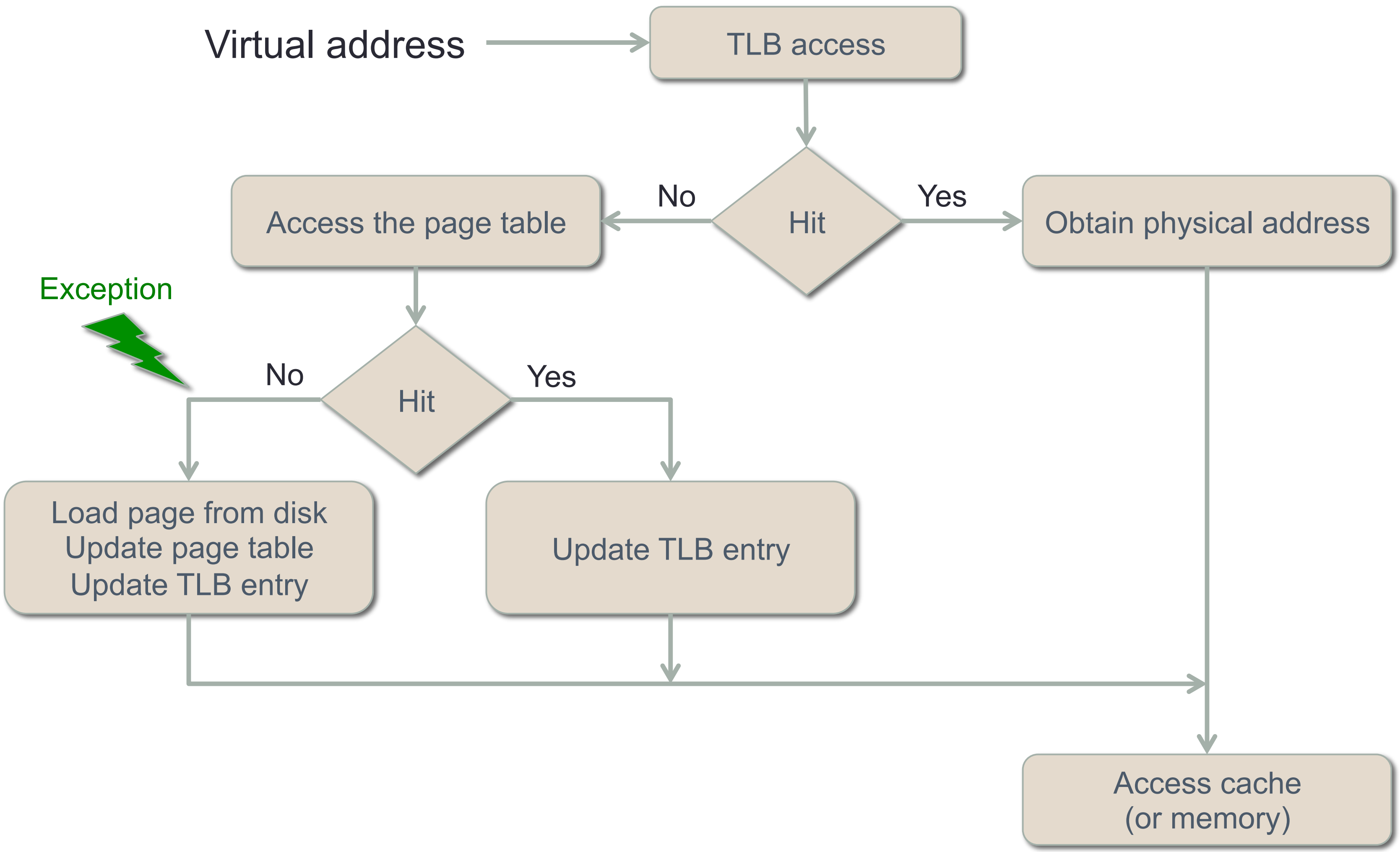
- The page table resides in main memory and contains, for each entry:
  - The physical page address (or the disk location of the page)
  - Valid bit
  - Dirty bit
  - Bits for the replacement algorithm
  - Permissions (read, write, execute)
- Two problems arise:
  - The page table may become vary large (and each process needs one)
    - This is solved with more efficient table organizations, such as hierarchical tables
  - Every access to memory requires two (translation + actual access)
    - This is solved with translation caches, AKA Translation Lookaside Buffer (TLB)

The TLB

- Locality enables TLB
  - Only a small portion of memory is in use at a particular time



Page table – TLB relationship



An example combining VM and cache

- MIPS R2000 and the DECstation 3100

Virtual memory in MIPS R2000

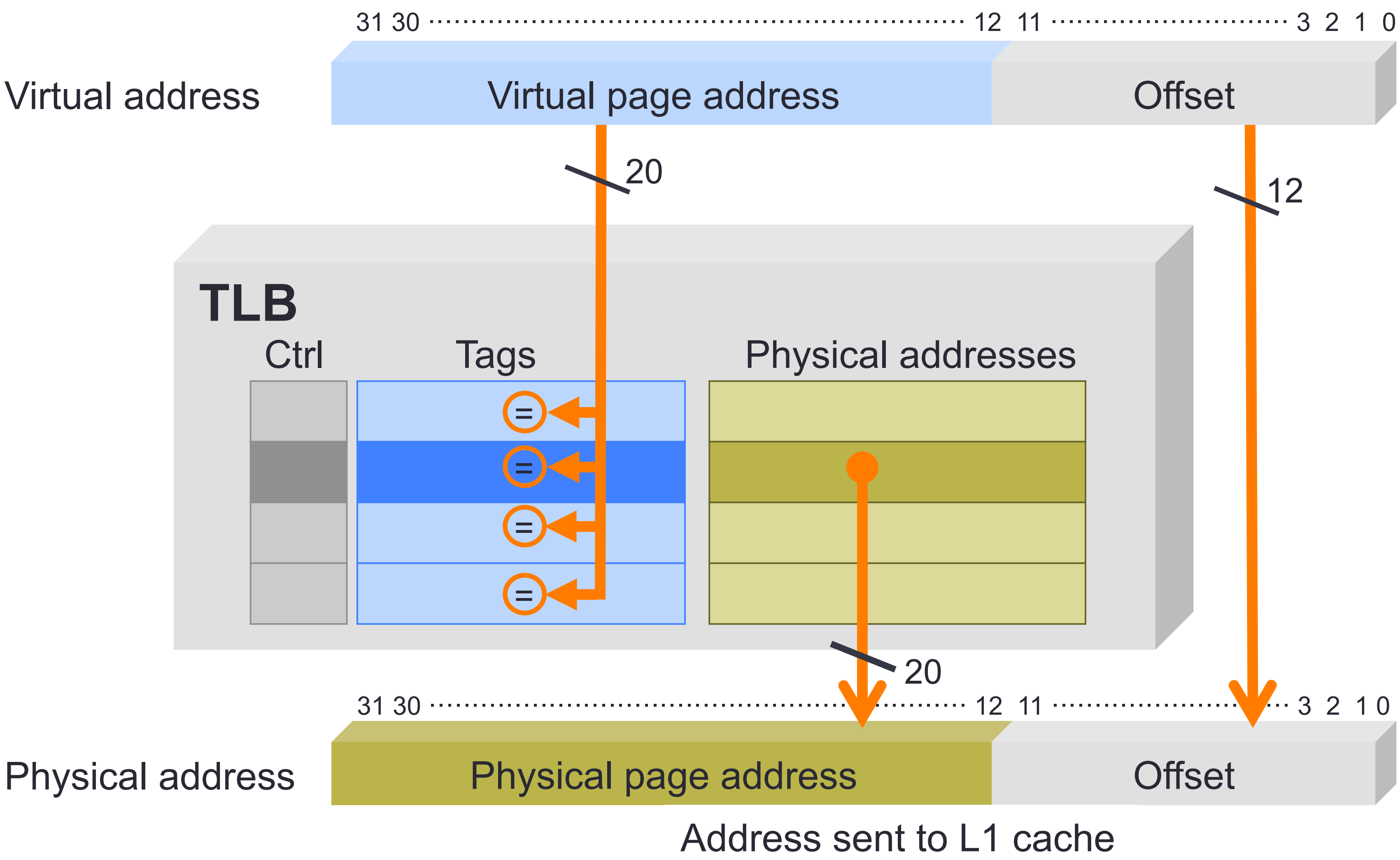
- Memory Management Unit (MMU) is in charge of handling VM
- Memory addressing
  - Page size: 4 KB
  - Physical addresses: 32 bits; Virtual addresses: 32 bits
    - Yes, this is also possible. Capacity is not increased, but VM provides flexibility for allocating memory to multiple processes in separate address spaces
  - On-chip TLB
    - Translates both code and data addresses
    - Fully associative
    - 64 entries (lines), each of 64 bits
      - A TLB entry:

Control	Virtual page (20 bits)	Physical page (20 bits)

TLB handling resources in MIPS R2000

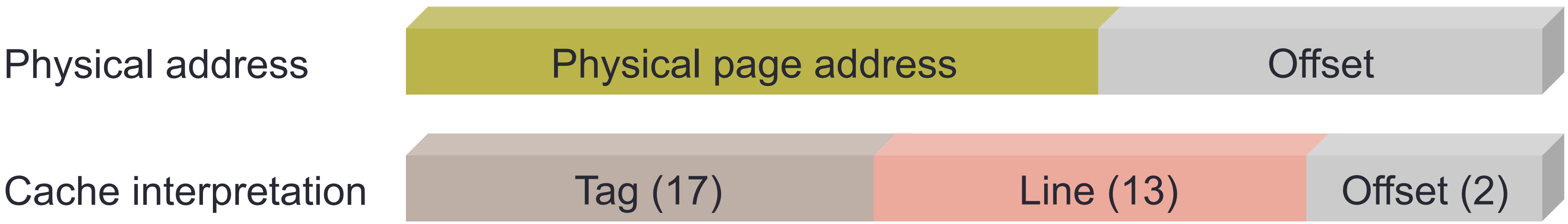
- All these resources are only visible in *kernel mode*
- Instructions
  - **`tlbr`** (TLB read)
  - **`tlbwi`** (TLB write index)
  - **`tlbwr`** (TLB write random – for random replacement)
- Special registers
  - **`EntryLo`** – **`EntryHi`**: LO and HI parts of data read/written to TLB
  - **`Index`**: 6 bits are used as index to access the TLB
  - **`Random`**: Contains a random value for replacement. It gets decremented on each CPU cycle

Virtual address translation in TLB



Cache in DECstation 3100

- Off-chip 32KB + 32 KB L1 split cache
- Direct mapping
- Block size of 4 B (one word)
- Write through, no write allocate



Cache access