ELEC-H-473

Microprocessor architecture

Caches

Lecture 05

Dragomir MILOJEVIC
dragomir.milojevic@ulb.ac.be

2013

Previously on ELEC-H-473

- Reasons that cause pipeline inefficiency :
  - ✓ Variance in execution time (e.g. CISC or memory)
  - ✓ Hazards
    - ❖ Data
    - ❖ Control
- Techniques that improve IPC of the pipeline
  - ✓ Register renaming
  - ✓ Data forwarding/out-of-order execution of instructions
  - ✓ Branch prediction
- Software link → loop unrolling

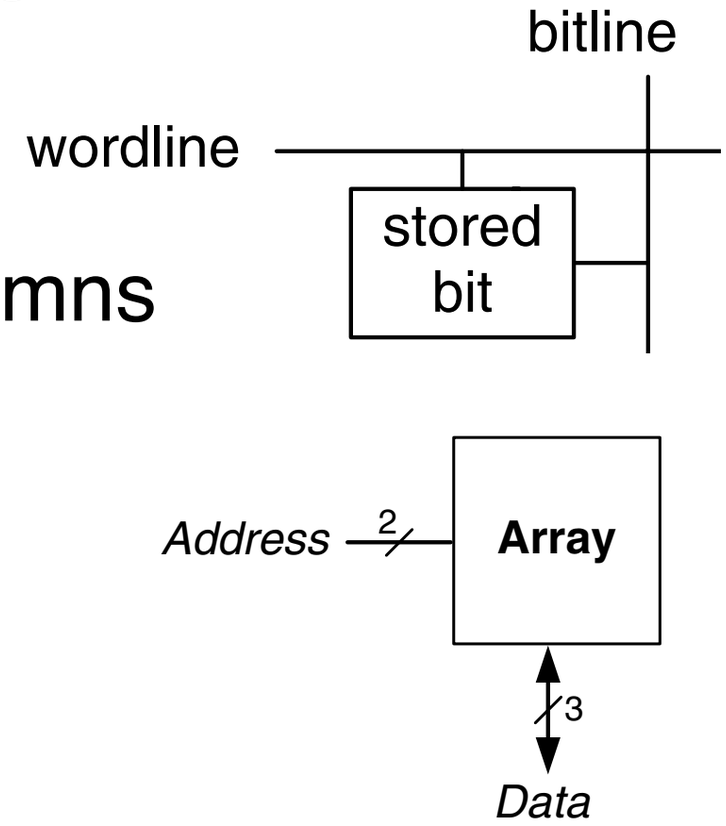
Today

- Memory technology
- Memory organisation
- Memory hierarchy
- Cache principals
- Virtual memory

Memory technology

Memory structure and organisation

- We start by introducing **bit-cell** storage, i.e. the smallest memory element capable of memorising one bit of info
- To build larger memory structures we use **2D memory arrays** of bit-cells organised in n rows and m columns (in all n x m cells)



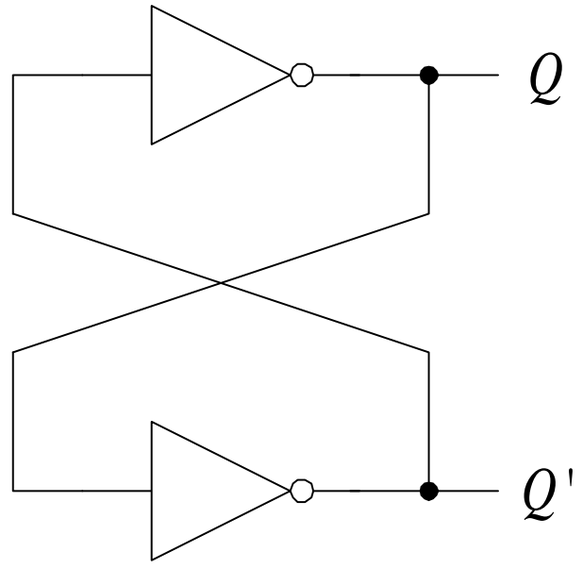
- Each row is identified with unique address, and only one row can be R/W accessed at a time
- When accessed address activates one row of the array and
  - ✓ places the data from the bus into the array (WRITE)
  - ✓ places the data on the bus from the array (READ)

Bit-cell technology

- Many technology options exist to store one bit of information (and many others are in currently in development)
- Two major classes, depending on the the information loss on power down : **volatile** and **non-volatile**
- Key volatile technologies:
  - ✓ **Flip-flop —** Basic storage element for any sequential logic circuit; fastest of the three, but most expensive in terms of area (typically used to build the FSMs)
  - ✓ **SRAM —** Static Random Access Memory (random → any data address can be accessed in the same amount in time); performance/price in the middle (targets smaller storage entities)
  - ✓ **DRAM —** Dynamic Random Access Memory low cost/bit but at the expense of access time (massive storage)

Latches & Flip-flops

- **Bistable —** Sequential circuits having only two stable states that are controllable by their inputs



- If the bistable element is sensitive to a level it is called **latch**

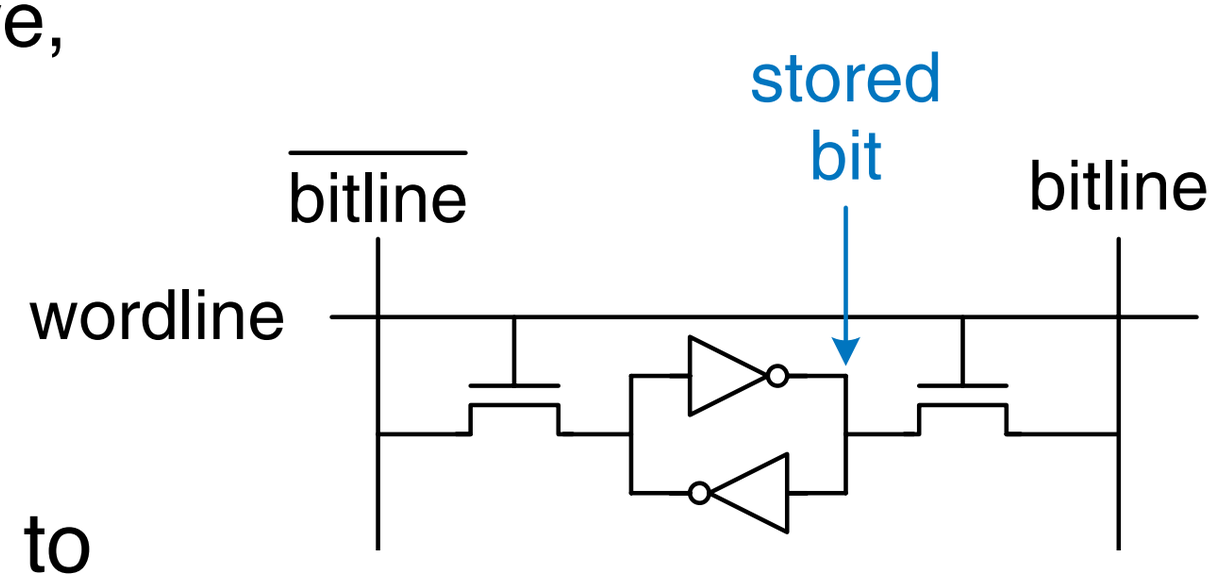
- If the bistable element is sensitive to an edge, it is called **flip-flops**

- For FF, the edge is produced by a rising/falling edge of the periodic signal (clock)

- Used for FSMs or to pipeline whatever part of the IC

SRAM cell

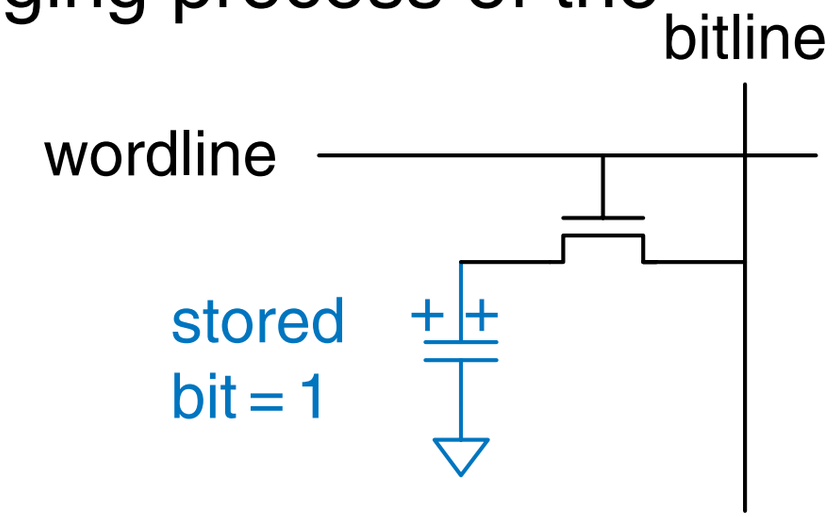
- The bit of information is stored in **cross-coupled inverters**
- The cell can be in one of the three states :
  - ✓ Idle, READ or WRITE
- Two nMOS transistors control the READ/WRITE states :
  - ✓ When the wordline is active, both nMOS are ON and
  - ✓ the bitline state is transferred from/to inverter couple



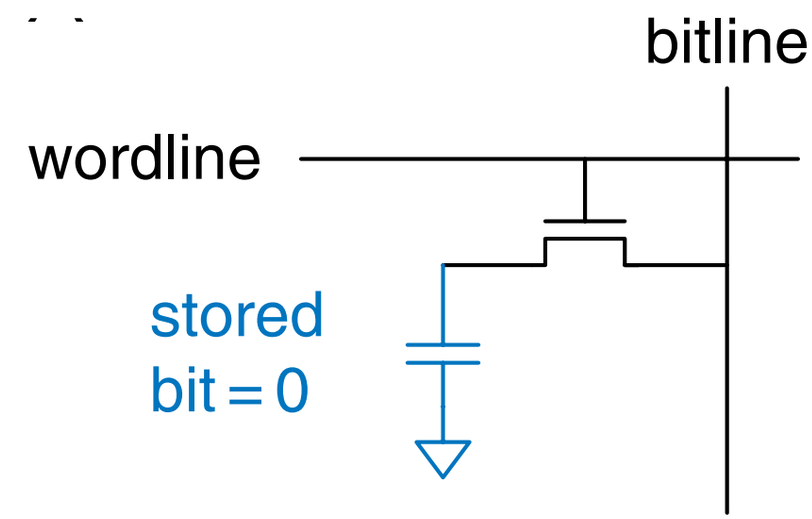
- Auto-refreshing — immune to noise because the inverters will restore the value that is possibly lost
- 6T(ransistor) configuration : 2 control + 2x2 for inverters

DRAM cell

- One bit of information is stored in the form of **capacitor charge**
- The nMOS transistor controls the charging process of the capacitor
- When the wordline is active
  → nMOS is on → the bit value
  is stored from the bitline (WRITE)
  or placed on the bitline (READ)



- Capacitor = physical component
  → there is **leakage**, the content
  is **lost** after some time
  - ✓ Anyhow after the read operation
    the content is lost too



- DRAM needs refreshment at regular rate even when not read, hence the name

Bit-cells : cross-comparison

	FF	SRAM	DRAM
Transistor count /bit	~20	6	1
Speed	<0.5	~1-10ns	100ns

- In reality the SRAMs performance parameters (delay and power) are function of the memory size
- Bigger the SRAM, bigger the access time is …
- This is going to influence the **memory subsystem** — subsystem, because we do not have just one memory instance !

Memory hierarchy — properties

- Multiple memories structured in **levels**
- Levels trade-off area (memory size) vs. access time and this is cost/performance driven
- Orders of magnitude difference in today's systems

	Reg File	L1	L2	L3	Main	Disk
Typical size	< 1kB	< 256kB	<24MB	< 2MB	< 512 GB	> 1TB
Access time (ns)	<0.5	0.5 - 25			<250	$5x10^6$
Managed by	compiler/ core	HW	HW	HW	OS	OS

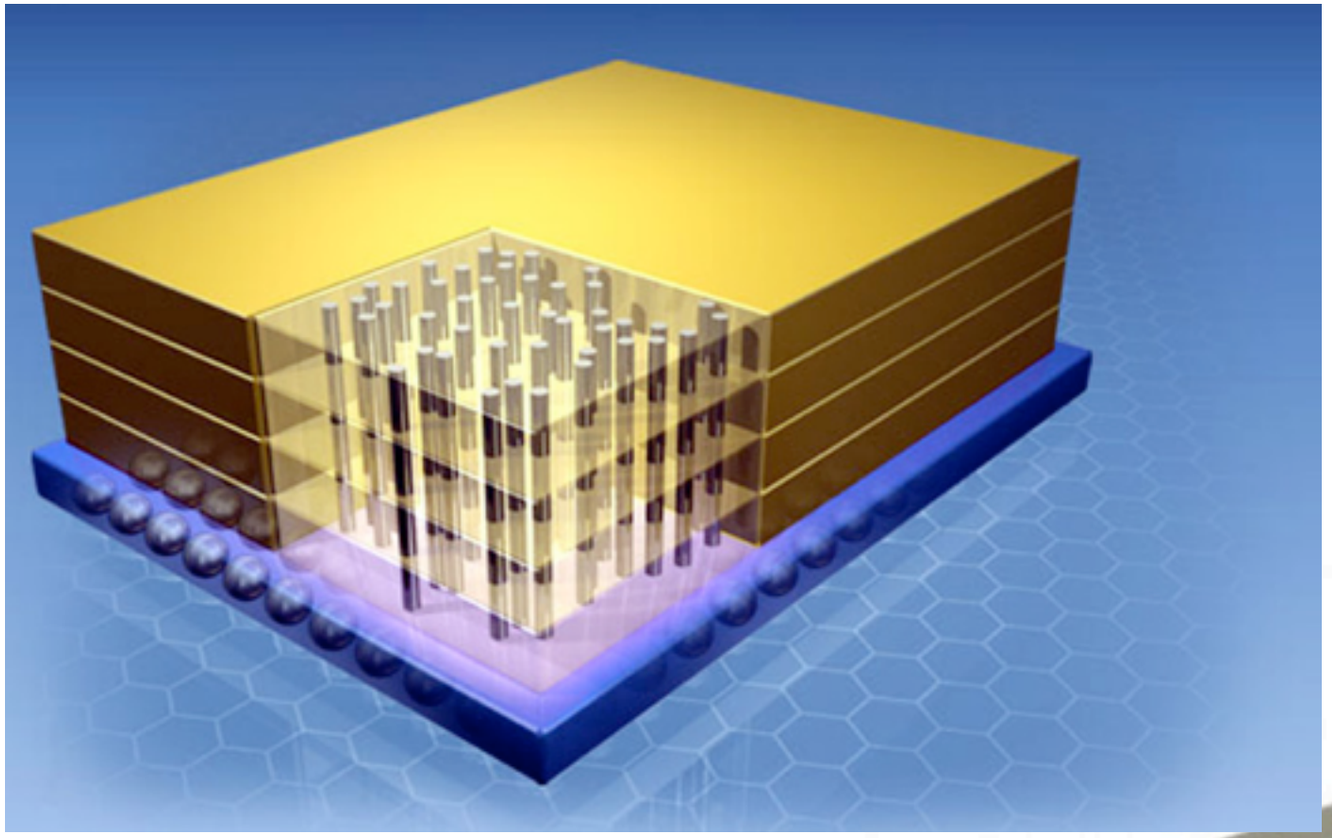
Main memory

- Today main memory is built using DDRs DRAMs :
  - ✓ Double Data Rate Synchronous Dynamic Random-Access Memories
- Double rate : 64 bits at both rising and falling edges
- But even with increasing interface frequency the bandwidth remains insufficient for some applications :

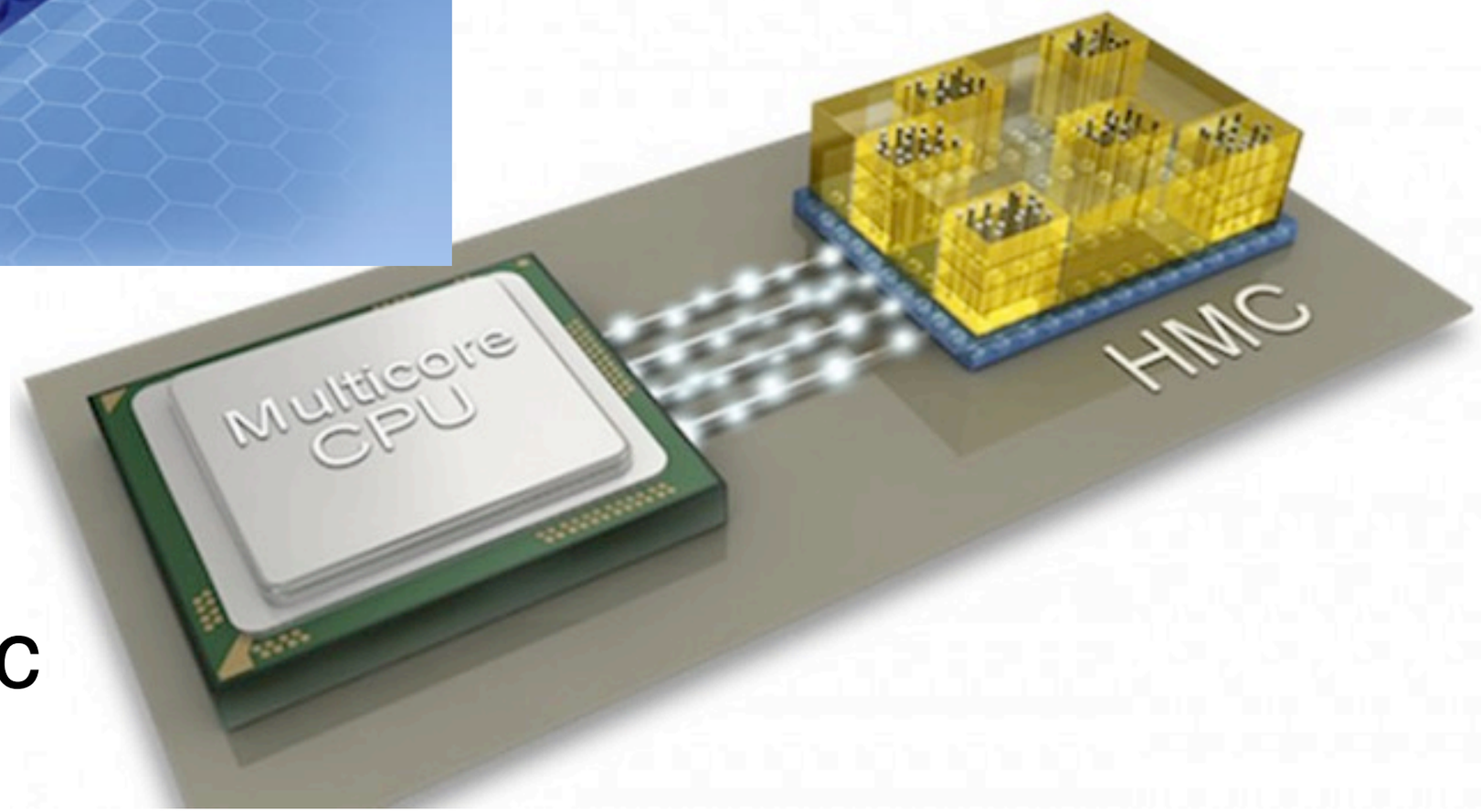
✓ → 3D stacked DRAM memories (Hybrid Memory Cube)

Names	Frequency	Transfer Rate	Max Bandwidth
DR	100 MHz	200 MT/s	1.6 GB/s
DDR2	400 MHz	800 MT/s	6.4 GB/s
DDR3	800 MHz	1600 MT/s	12.8 GB/s

Hybrid memory cube



Multiple DRAM dies stacked in the same package



Bottom die
→ CTRL Logic

Memory organisation

Memory access

- We need to define the **smallest unit of memory access**
- The choice is made at architecture level (often based on cost/ performance trade-off)
- The size of the smallest data unit will impact the amount of data allocated to a single address seen by SW
- If the smallest unit of access is one byte, then in order to construct more complex data types (like 64 bit integers that are maybe more common then unsigned bytes) we need to perform multiple memory accesses (8 in this case)
- To increase memory bandwidth parallelism is required and bytes are in general packed in **words** — one data access concerns one address / one packed word

Memory access

- From HW perspective reading/writing data will be performed at the scale of the word to increase memory access efficiency
- Single transfer is never a good idea
  - ✓ CPU to memory communication is protocol based
  - ✓ Establishing a communication can be time consuming
  - ✓ When you make a connection you want to transfer as much data as you can
    - ❖ **Burst vs. single word transfer**
- But after all we want a General Purpose Computer and all data types possible
  - ✓ What about memory access to data types smaller then word?

Memory alignment and padding

- If the addresses are multiples of the word size, then we say that the data is **aligned**
- If the data is smaller then data word we have a problem …
- In a 64-bit memory system accessing 8-bit data types, assume a data array

  ✓ We have to extract the actual byte out of the word information (expensive in logic and especially time); space overhead although not critical because of the DRAM $/byte

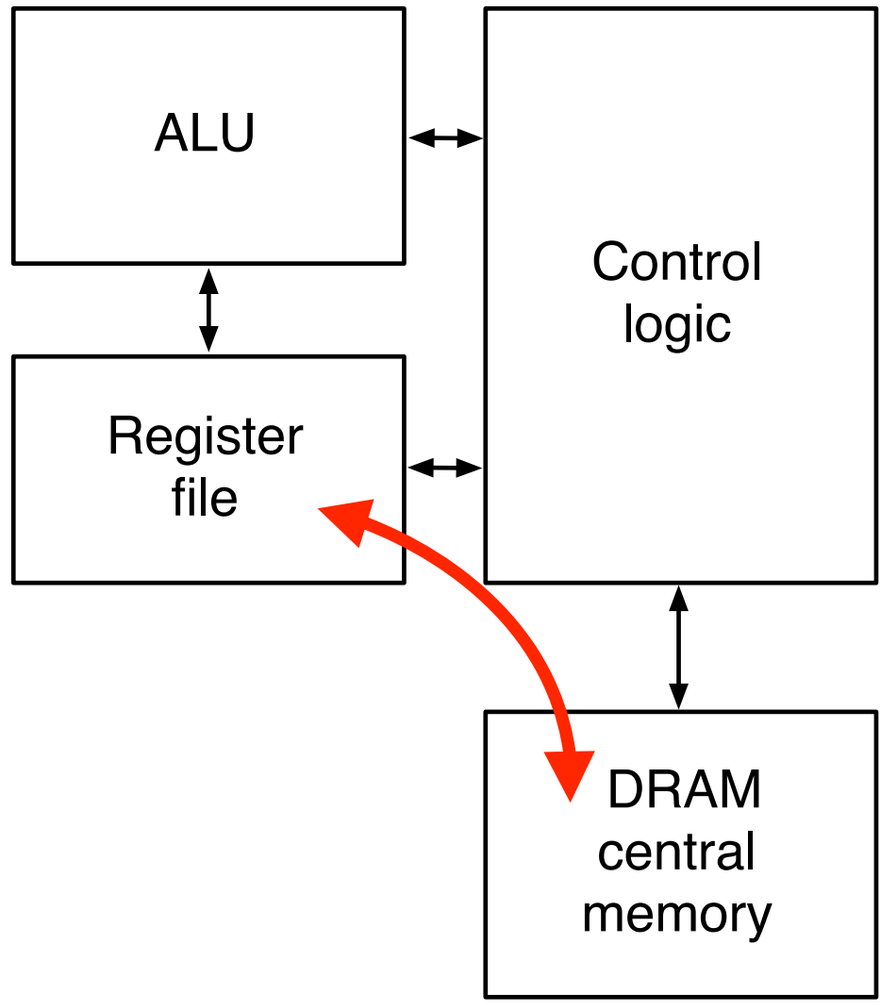
  ✓ We **align** all 8-bit data addresses to 64-bit words

    ❖ In that case, assuming that our word goes at the first byte of the word, other 56 bits need to have some kind of dummy values (can be zero!) — **data structure padding** → filling out the remaining space with useless data

Memory hierarchy

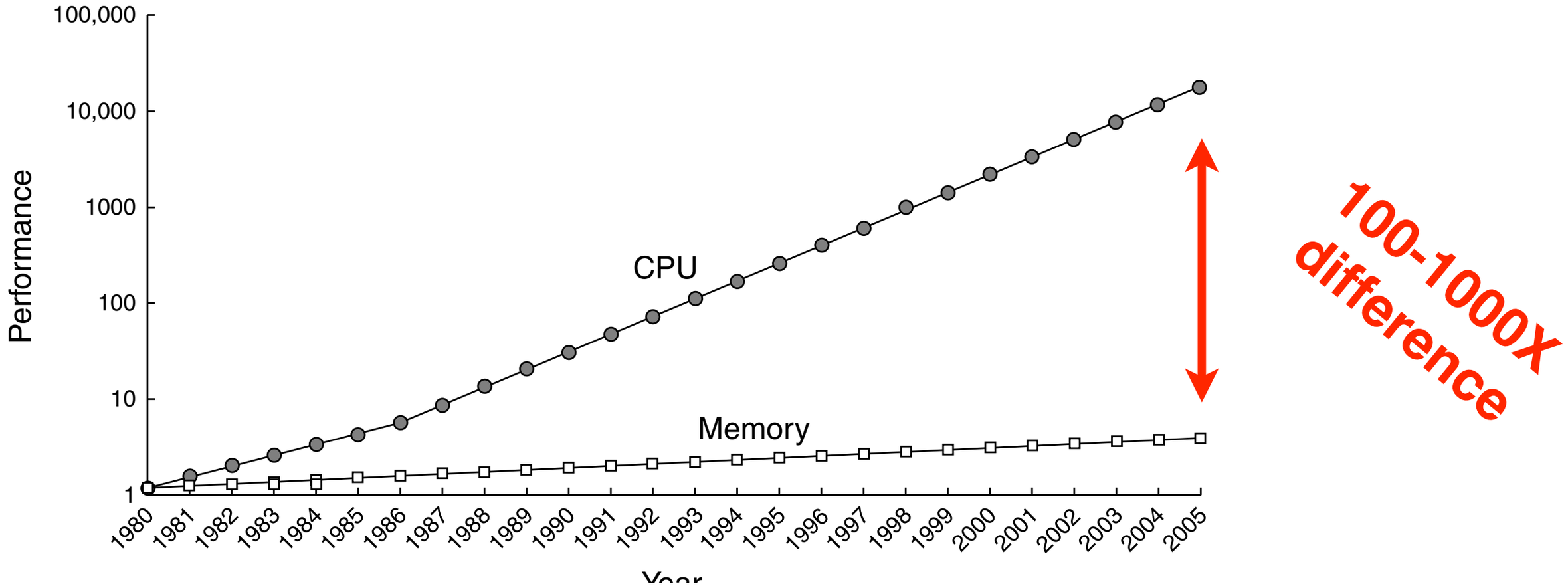
One central memory doesn't work

- Imagine that the Register File is directly connected to the main memory
- If the CPU needs data (this is true for program data and instructions), it needs to fetch it directly from the main memory
- In order to have balanced execution, and not to stall the processing or the memory IO, we need to compute and access data equally fast



This is not the case due to technology reasons

Uneven scaling of logic vs. memory



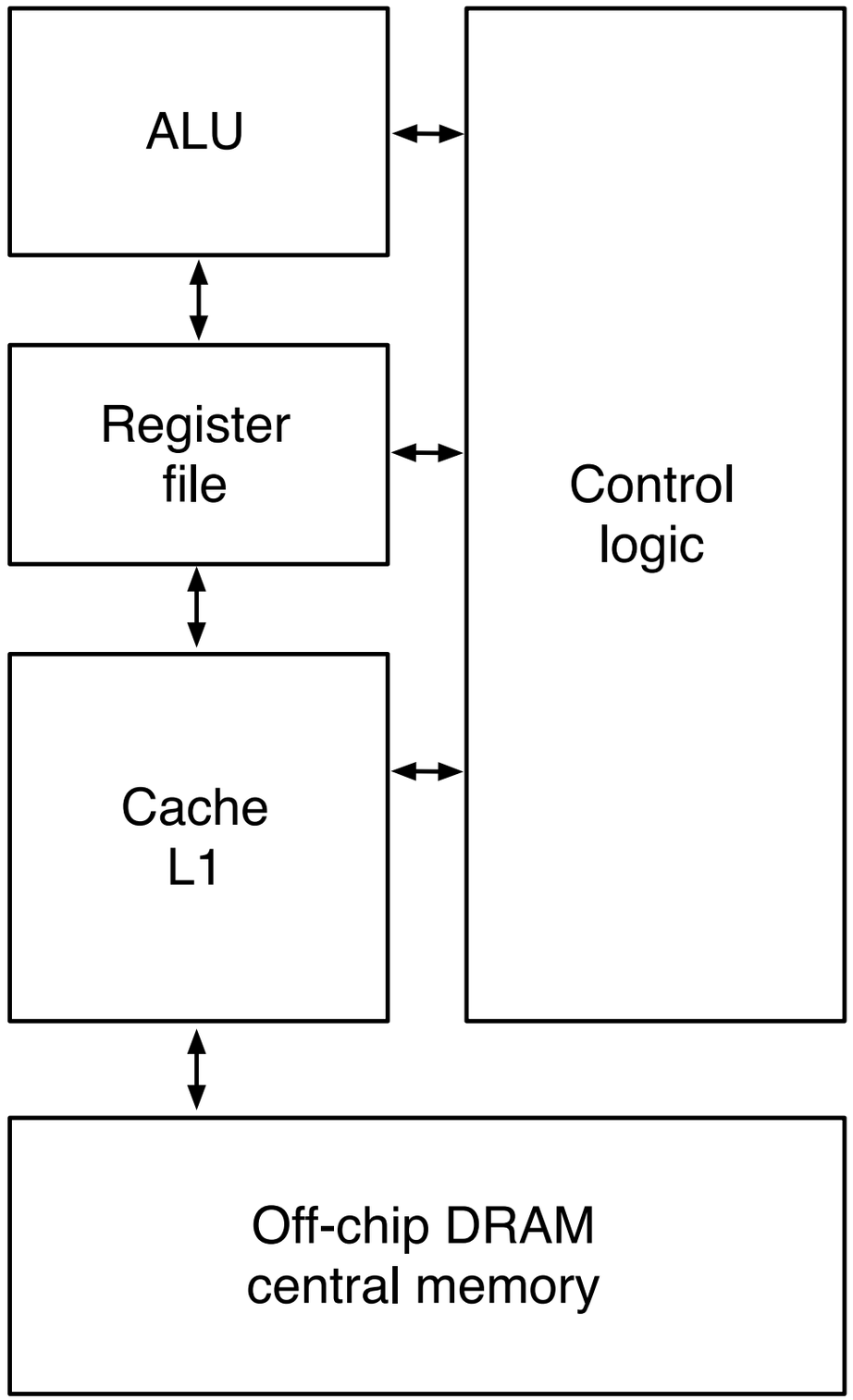
If the CPU needs to wait for the data, it is said that the CPU is stalled, we waste plenty of useful CPU cycles

What is the solution?

- Insert smaller, but faster memories on the instruction/data path between CPU & the main memory — **cache**
- Cache holds a copy of the main memory content and can deliver it to the CPU much faster
- When the CPU access new data (or new instruction) it will first look into the cache:

➡ if this operation succeeds — **cache hit** — main memory latency is hidden to the CPU that continues computations
→ **the CPU is not stalled**

➡ if not — **cache miss** — the data needs to be fetched from the main memory,
→ **the CPU is stalled,** we loose CPU cycles



Why caching works ?

- Cache hits are possible because of **locality** :
  - ✓ Same data locations are accessed closely in time
    → **Temporal locality**
    - ❖ Typical example: loop counter
  - ✓ Data that will be accessed in the near future is generally not that far away from the current data
    → **Spatial locality**
    - ❖ Typical example: data arrays
- How local data is, will depend on:
  - ✓ the problem itself
  - ✓ the algorithm used to solve the problem
  - ✓ program implementation
- And also the way the cache is controlled …

How good is the solution ?

- Obviously:
  - ✓ **More cache hits** → less CPU stalls → **better performance**
  - ✓ More caches misses → more stalls → worse performance
- One wants to trade-off:
  - ✓ Performance — by maximising cache hits,
  - ✓ Cost — by minimising the cache size
    - ❖ Implementing big on-chip caches is costly (making SRAMs using technologies dedicated to logic is not that efficient)
- These two criteria are orthogonal !
  - ✓ Complex compromises need to be made … but more memory wins because apps are memory hungry

Performance metrics

- Memory stall cycles = IC x Misses/Instruction x Miss cost
- Miss cost will depend on :
  - ✓ **Memory access time**
    - ❖ How much time is required to get one data
  - ✓ **Hit rate**
    - ❖ How many times the actual access returned the right data
  - ✓ **Latency**
    - ❖ Time elapsed between request and first data being available
  - ✓ **Bandwidth**
    - ❖ The amount of time necessary to deliver a block of data
- To count misses : cache simulators that log memory access traces → these can be analysed later using tools Cacti, VTune…)

Memory hierarchy

What does cache memory stores ?

- Obviously useful information : **instructions and/or data** that are simply called **cache entries**
- Each **cache entry** is structured in fields :
  - ✓ **tag** — a part of the data main memory address
  - ✓ **data block** — the actual data
  - ✓ **flags** — indicating the status of the data block
    - ❖ **valid** — if the valid data has been loaded
    - ❖ **dirty** — if the data has been modified by the CPU (this means that some process of main memory update should take place)
- Data transfers are done from the main memory to the cache memory using **cache lines**, the size of the line depending on the hardware choices

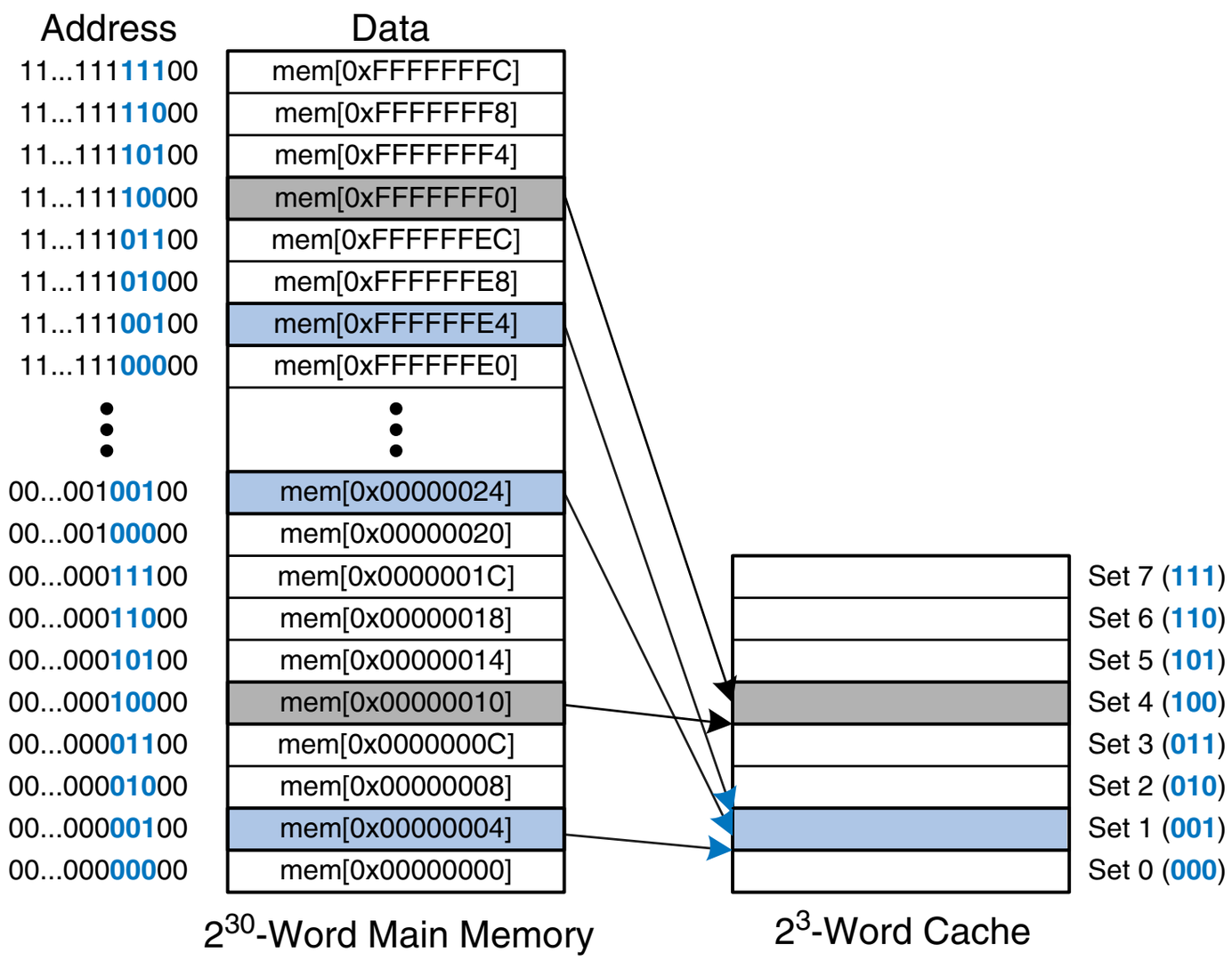
Main questions related to cache

- Where to place a given block of data in the cache?
  - ✓ **Block placement**
    - ❖ The question is how to map main memory space onto cache memory space
- How do you find a given data in the cache?
  - ✓ **Block identification**
    - ❖ We need to find the way to quickly and at lowest HW cost identify where in cache is the data
- Which data block should be evicted to make room for new ones?
  - ✓ **Block replacement**
    - ❖ Cache is of finite, in principal small size, so we need to erase unused data to enable storage of newer and more useful data

Depending on how you do these → many options …

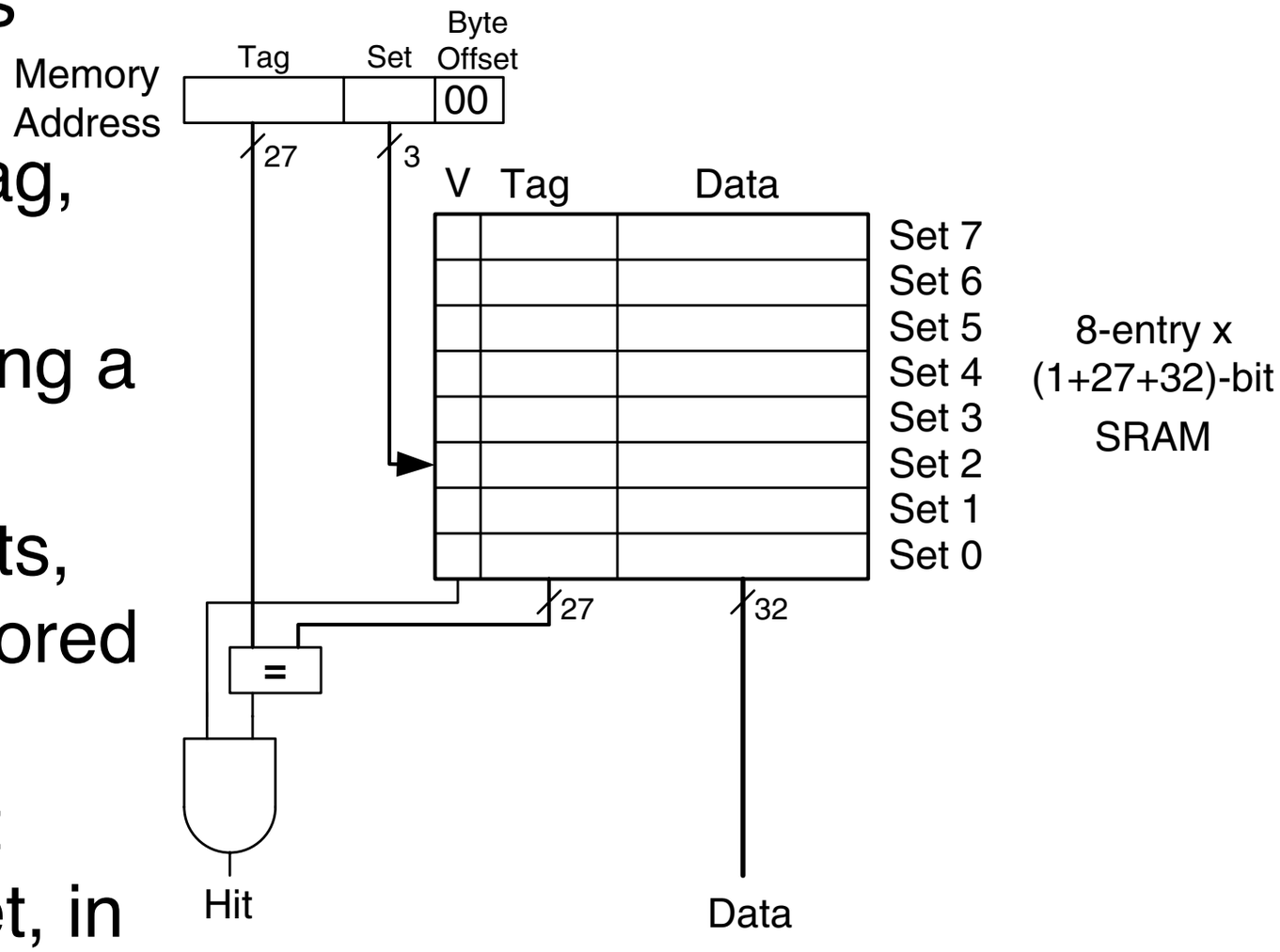
Direct mapping — simplest way

- We map 2^{30} main memory space into 2^3 cache memory space (there are 8 cache entries in all)
- Memory space uses 4 byte words and the addresses are 4 byte aligned (so word addresses are 4, 8, C, etc.)
- Last two bits indicate byte offset within the word
- The following 3 bits map the set address



Block identification — example

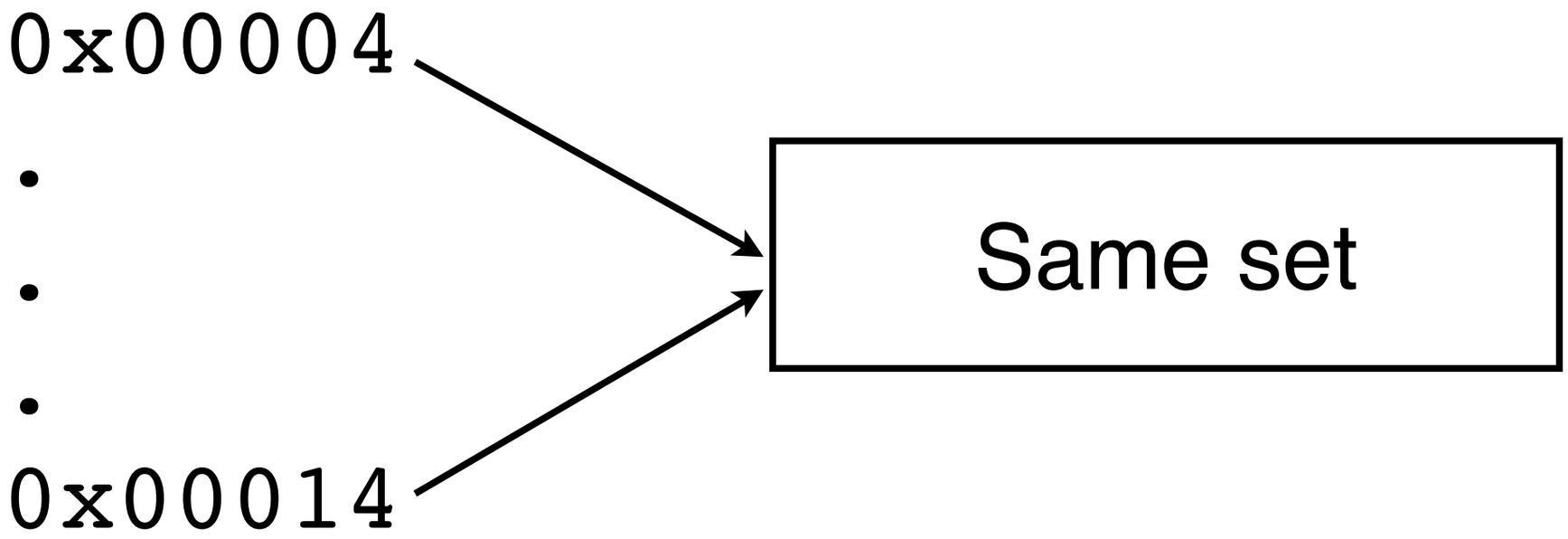
- Each entry, or set, contains one line consisting of 32 bits of data, 27 bits of tag, and 1 valid bit
- The cache is accessed using a 32-bit address
- The two least significant bits, the byte offset bits, are ignored for word accesses
- The next three bits, the set bits, specify the entry or set, in the cache that match the most significant bits



- A load instruction reads the specified entry from the cache and checks the tag and valid bits : if the tag is ok → bingo !

Direct mapping cache — problem

- Because there is only one set, i.e. given address can be found on only one cache address, if two consecutive memory accesses target the **address pointing to the same set**, there will be a **conflict**
- The latest data will evict the previous data !
- If this occurs in a loop, we will have a systematic conflict that will lead to a systematic cache misses (and hence the performance loss)



Example

- Analyse the execution of the two programs on the right
- Assume direct cache mapping with 8 cache entries
- Answer :
  - ✓ What do these programs do?
  - ✓ What is the difference?
  - ✓ What is the miss rate for these programs?

```
mov     bx,35
mov     cx,10000000
loop:
    add    bx, [0x004]
    add    bx, [0x008]
    sub    cx, 1
jnz    loop
```

```
mov     bx,35
mov     cx,10000000
loop:
    add    bx, [0x004]
    add    bx, [0x024]
    sub    cx, 1
jnz    loop
```

Solution for conflicts in direct mapped

- Each main memory address could potentially reside on multiple locations in the cache memory : **associativity**
- A cache memory is said to be m-set associate if the data could be found in m different sets — this is the **degree of the associativity**
- Depending on the value of m we can have less to more associativity :

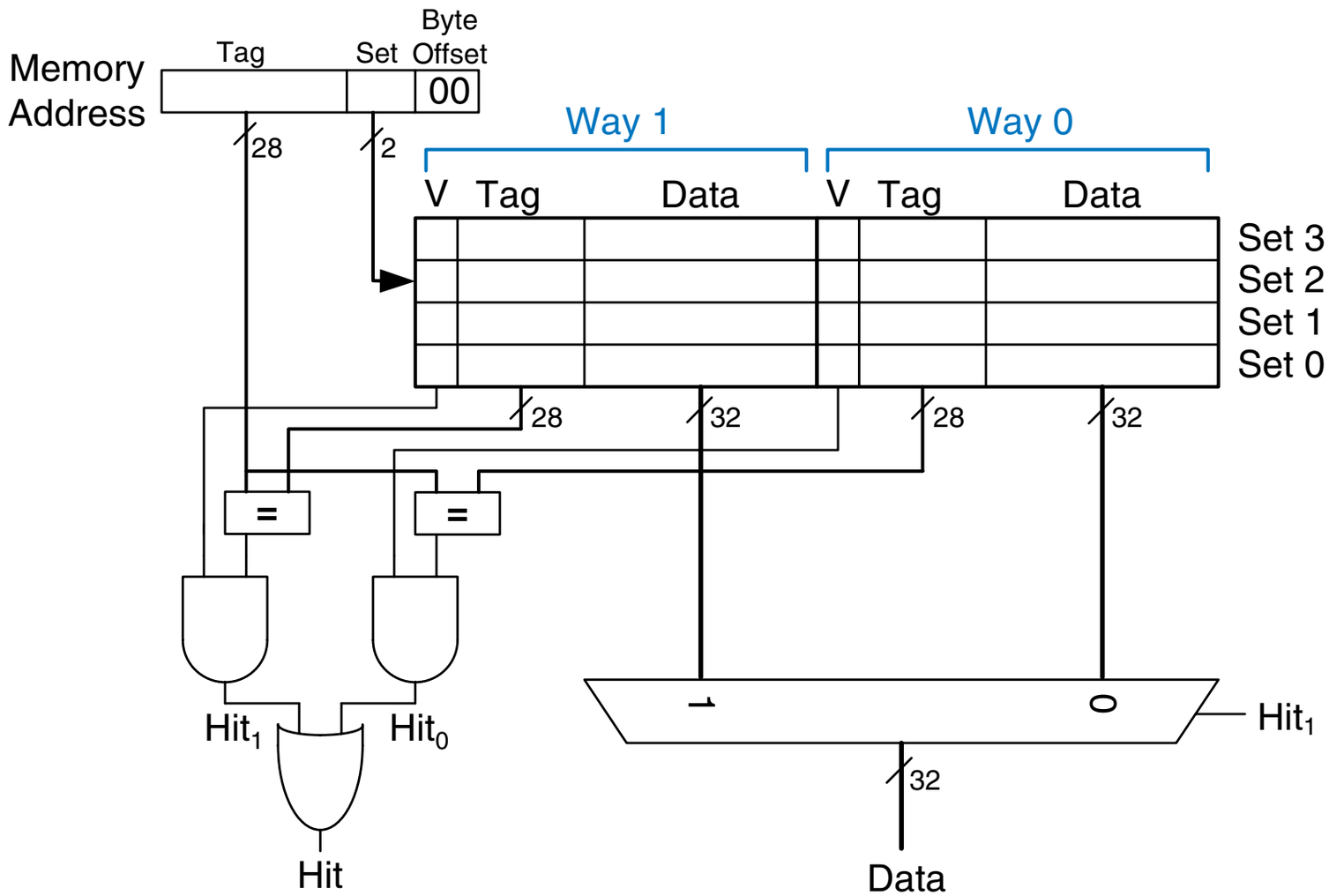
✓ **direct mapped cache** — 1-way set associate because a given address can be located on only one address in the cache

✓ **m-set** — data could be found in one of the m sets

✓ **fully associate** — given data can be placed anywhere in the cache

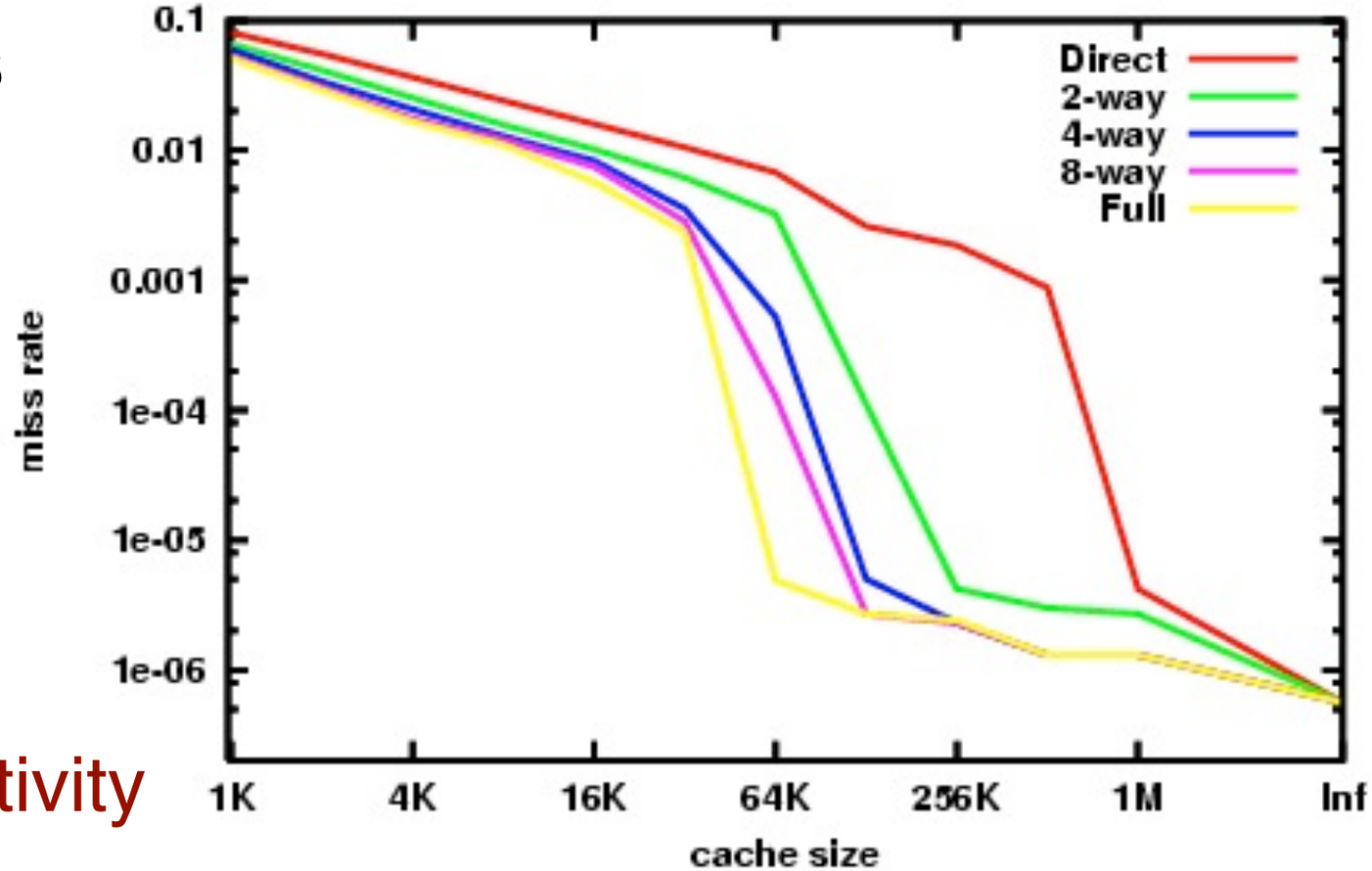
Example of 2-way set associate cache

- The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit
- If a hit occurs in one of the ways, a multiplexer selects data from that way
- In the example above now both 0X004 & 0X024 could remain in the cache



Set-associative cache performance

- Bigger caches improve hit rate (they reduce miss rate)
- More associativity means better probability to avoid address conflicts
- More associativity means also more complex search (8-way set associate cache needs to check 8 tags)
- Running a benchmarks gives you the idea
  - ✓ **Double the way has the effect of doubling the cache size !!!**
  - ✓ The effect is reduced for increasing associativity



Replacement policy

- In m-way set associative cache when cache miss occurs the controller will fetch a new cache line
- … and it need to decide where to store it
- So, how to decide where to store a new cache line ?
- In direct mapped caches, there is no choice, since there is only one address where this line can be stored
- What about m-way ?
- Which existing line in cache to replace ?
- How to make a good crystal ball?

Popular replacement policy algorithms

- **Random** — pick the candidate randomly
  - ✓ very easy to implement, but you might evict the data that you will need soon
- **Least Recently Used (LRU)** — If recently used data is to be reused, least recently data will not be used !
  - ✓ If this is true then erase the block that has been unused for the longest time
- **First In First Out (FIFO)** — LRU needs to keep track of what is going on → increased HW complexity. FIFO emulates LRU behavior at lower implementation cost

Performance of replacement algorithms

- LRU is the best for smaller caches, however no significant difference for bigger caches with random

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Figure C.4 Data cache misses per 1000 instructions comparing least-recently used, random, and first in, first out replacement for several sizes and associativities. There is little difference between LRU and random for the largest-size cache, with LRU outperforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using 10 SPEC2000 benchmarks. Five are from SPECint2000 (gap, gcc, gzip, mcf, and perl) and five are from SPECfp2000 (applu, art, equake, lucas,and swim). We will use this computer and these benchmarks in most figures in this appendix.

Write policy

- Similar procedure as with read: when data is supposed to be written, the address is first checked to see if it is cashed
- Again two situations can occur:
- **Hit** — the data is first written into cache
- **Miss** — the block of data corresponding to the address to be written is fetched from the main memory

- Depending on the way the main memory is updated:
- **Write though** — the main memory is immediately updated; this can be time consuming if the central memory is too slow (and especially, if this location is constantly updated)
- **Write back** — dirty bit associated with each block is set to 1, if there was a write to this block; the main memory is updated only when the data from the cache is evicted

Write policy

- Common optimization is to insert a **write buffer** between the cache memory and the main memory
- The buffer is filled in by the CPU and emptied by the memory (this is a FIFO)
- Since the write buffer stores the data and has the life on his own, it will not stall the CPU
- How good it is will depend of course of the FIFO depth : more is better but we need to keep the FIFO small (area penalty) → if we can not empty the FIFO fast enough : **buffer saturation** and we are facing again the same problem
- Solution : insert L2 that can match the emptying frequency imposed by the CPU

Instructions

- Previously we assumed data caches only
- Similar same principals are applied to instructions cache
- The only difference is that for instructions we are only in the READ mode

Further reduction of miss rate

- After all we still have cache misses ...
- Cache hierarchy design is sensitive, because there will be always causes for a cache miss that can be :
  - ✓ **Compulsory** — cache needs to be initialized every time → this is **unavoidable** (unless you do explicit prefetch of data in cache); it generally accounts for a small proportion of the total miss rate
  - ✓ **Conflicts** — there will be more requests then ways, this is application dependent but also implementation trade-off
  - ✓ **Capacity** — if the cache size is too small, data blocks will be systematically evicted and then reloaded, this one is most important
- Whatever is done in HW it is a choice, application has to know this and use the best it can whatever is there …

Example

- Take matrix multiplication : if matrix is to big, cache misses ...

```
for i in 0..n
  for j in 0..m
    for k in 0..p
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

- Cut the matrix in blocks of the cache size and do block-by-block computation to avoid cache capacity problems

```
for (ii = 0; ii < SIZE; ii += BLOCK_SIZE)
  for (kk = 0; kk < SIZE; kk += BLOCK_SIZE)
    for (jj = 0; jj < SIZE; jj += BLOCK_SIZE)
      for (i = ii; i < ii + BLOCK_SIZE && i < SIZE; i++)
        for (k = kk; k < kk + BLOCK_SIZE && k < SIZE; k++)
          for (j = jj; j < jj + BLOCK_SIZE && j < SIZE; j++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Virtual memory

Physical vs. virtual memory

- At the end of the day you need to store the information somewhere — **physical memory** (caches, main memory etc.)
- But you have all the freedom of the world to create the address space you like …
- **The question is how much of the memory OS will allocate to a newly generated process ?**
- You can limit this to all available, or a portion of the physical memory
- Another way would be to allow every program to see a memory space that correspond to all addressable memory space — **virtual memory**

Example

- Consider the following :

```
int size = 64 000 000 000

new double my_array[size];
```

- Will this work (BTW can you compute the size in bytes)?
  - ✓ In a system where OS see only the physical memory NO
- But if you assume that the HDD could be a potential memory extension, this is possible
- If the address bus of our architecture has 64 bits, the size of the address space is 2^{64}
- Virtual memory takes all addressable space

Virtual memory

- All this virtual memory space is allocated to every process running on the system (every application sees the maximum memory)
- Virtual memory is mapped to a physical memory using **address translation**
- Both virtual and physical memory are divided into pages (typically 4kB)
- Any virtual page can be found either in main memory or on HDD — whenever a programs access a virtual page :
  - ✓ Either it is in the memory → no penalty
  - ✓ Either it is on the HDD → the page needs to be transferred from HDD to the main memory → there is penalty