Memory Hierarchy

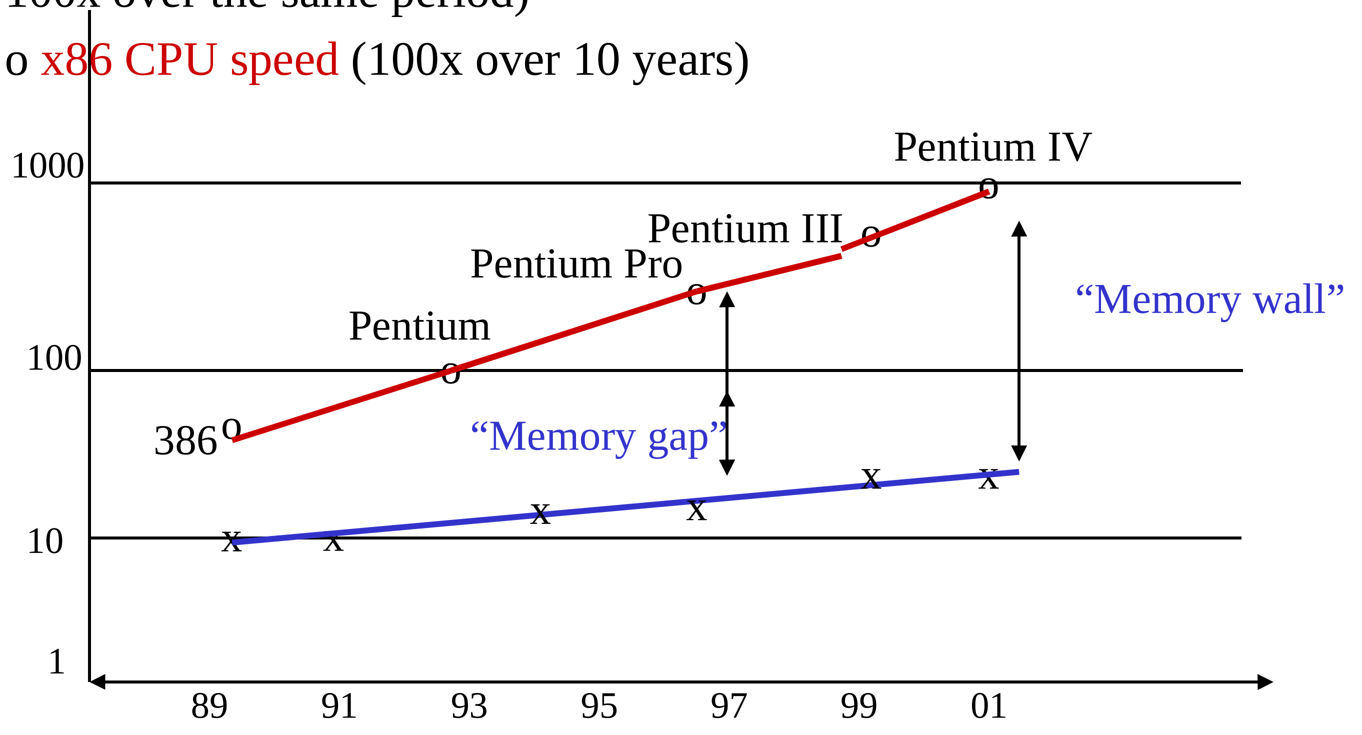
- Memory: hierarchy of components of various speeds and capacities
- Hierarchy driven by cost and performance
- In early days
  - Primary memory = main memory
  - Secondary memory = disks
- Nowadays, hierarchy within the primary memory
  - One or more levels of caches on-chip (SRAM, expensive, fast)
  - Generally one level of cache off-chip (DRAM or SRAM; less expensive, slower)
  - Main memory (DRAM; slower; cheaper; more capacity)

Goal of a memory hierarchy

- Keep close to the ALU the information that will be needed now and in the near future
  - Memory closest to ALU is fastest but also most expensive
- So, keep close to the ALU *only* the information that will be needed now and in the near future
- Technology trends
  - Speed of processors (and SRAM) increase by 60% every year
  - Latency of DRAMS decrease by 7% every year
  - Hence the *processor-memory gap* or the *memory wall* bottleneck

Processor-Memory Performance Gap

- x Memory latency decrease (10x over 8 years but densities have increased 100x over the same period)
- o x86 CPU speed (100x over 10 years)



Typical numbers

Technology	Typical access time	$/Mbyte
SRAM	3-20 ns	$50-200
DRAM	40-120ns	$1-10
Disk	milliseconds $\approx 10^6$ ns	$0.01-0.1

Principle of locality

- A memory hierarchy works because text and data are not accessed randomly
- Computer programs exhibit the *principle of locality*
  - *Temporal locality*: data/code used in the past is likely to be reused in the future (e.g., code in loops, data in stacks)
  - *Spatial locality*: data/code close (in memory addresses) to he data/ code that is being presently referenced will be referenced in the near future (straight-line code sequence, traversing an array)

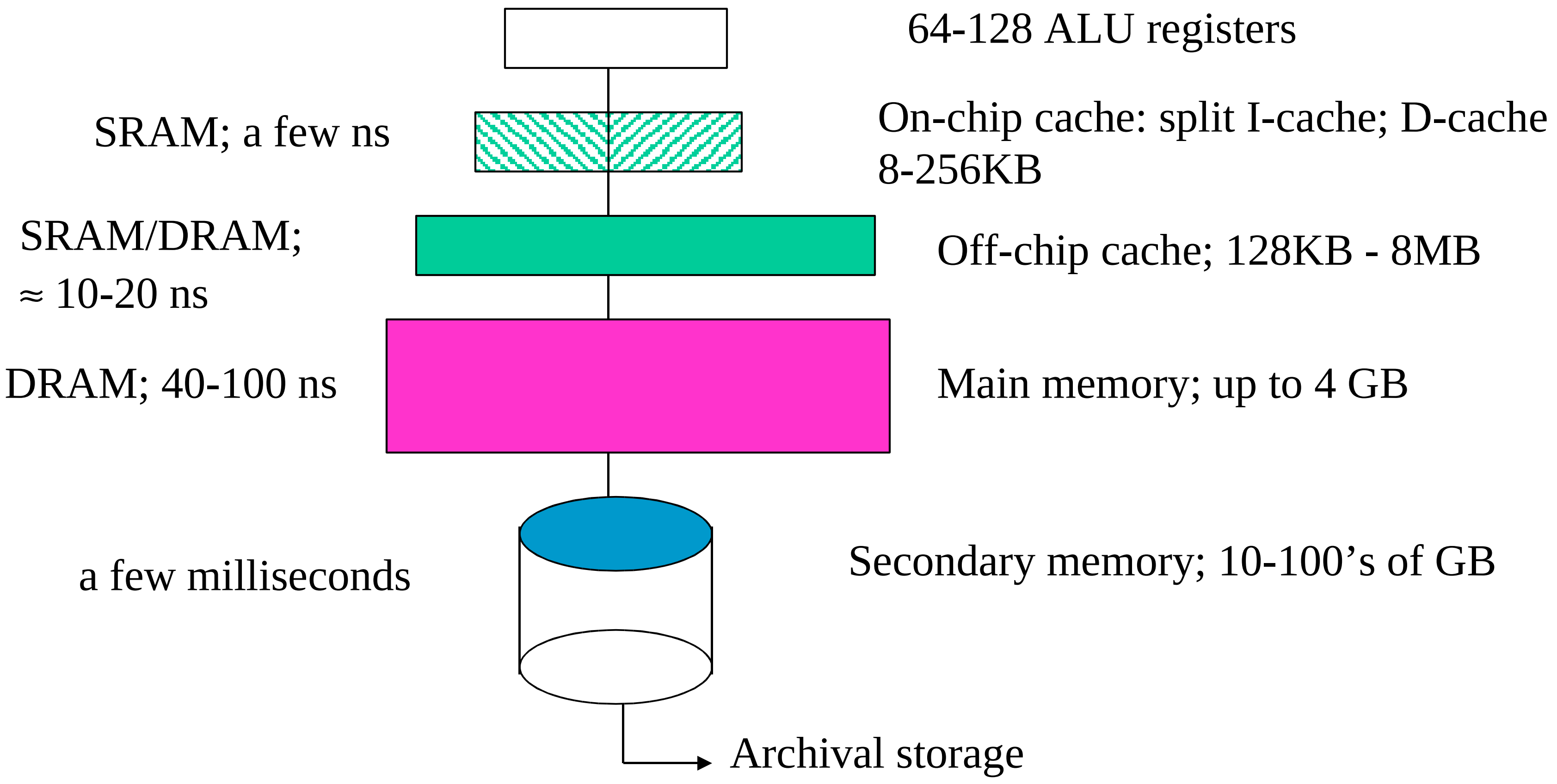
Caches

- Registers are not sufficient to keep enough data locality close to the ALU
- Main memory (DRAM) is too “far”. It takes many cycles to access it
  - Instruction memory is accessed every cycle
- Hence need of fast memory between main memory and registers. This fast memory is called a *cache.*
  - A cache is much smaller (in amount of storage) than main memory
- Goal: keep in the cache what’s most likely to be referenced in the near future

Basic use of caches

- When fetching an instruction, first check to see whether it is in the cache
  - If so (*cache hit*) bring the instruction from the cache to the IR.
  - If not (*cache miss*) go to next level of memory hierarchy, until found
- When performing a load, first check to see whether it is in the cache
  - If cache hit, send the data from the cache to the destination register
  - If cache miss go to next level of memory hierarchy, until found
- When performing a store, several possibilities
  - Ultimately, though, the store has to percolate to main memory

Levels in the memory hierarchy

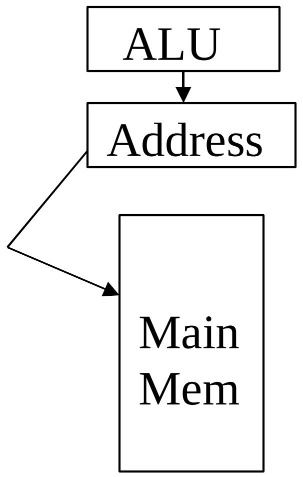


Caches are ubiquitous

- Not a new idea. First cache in IBM System/85 (late 60's)
- Concept of cache used in many other aspects of computer systems
  - disk cache, network server cache etc.
- Works because programs exhibit locality
- Lots of research on caches in last 20 years because of the *increasing gap* between processor speed and (DRAM) memory latency
- Every current microprocessor has a cache hierarchy with at least one level on-chip

Main memory access (review)

- Recall:
  - In a Load (or Store) the address in an index in the memory array
  - Each byte of memory has a unique address, i.e., the mapping between memory address and memory location is unique



Cache Access for a Load or an Instr. fetch

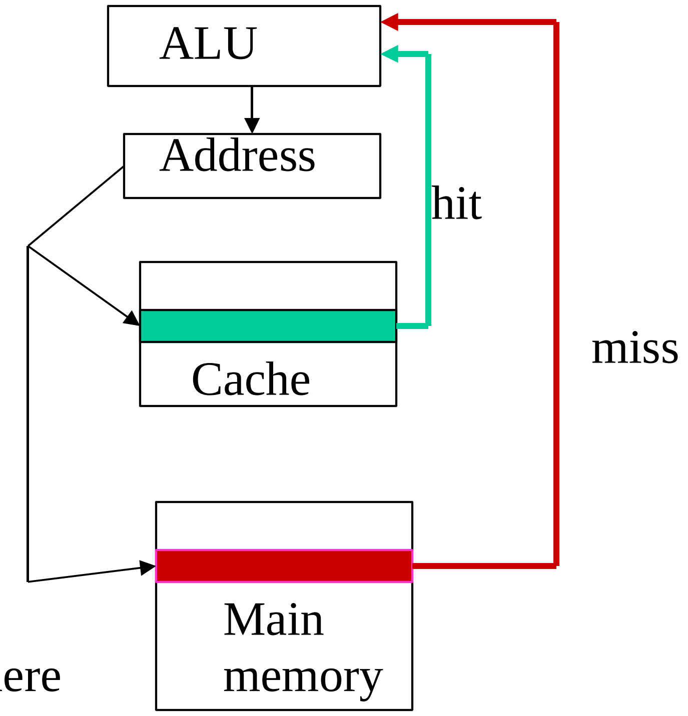
- Cache is much smaller than main memory
  - Not all memory locations have a corresponding entry in the cache at a given time
- When a memory reference is generated, i.e., when the ALU generates an address:
  - There is a look-up in the cache: if the memory location is *mapped* in the cache, we have a *cache hit.* The contents of the cache location is returned to the ALU.
  - If we don’t have a cache hit (*cache miss*), we have to look in next level in the memory hierarchy (i.e., other cache or main memory)

Cache access

How do you know where to look?

How do you know if there is a hit?

Main memory is accessed only if there was a cache miss



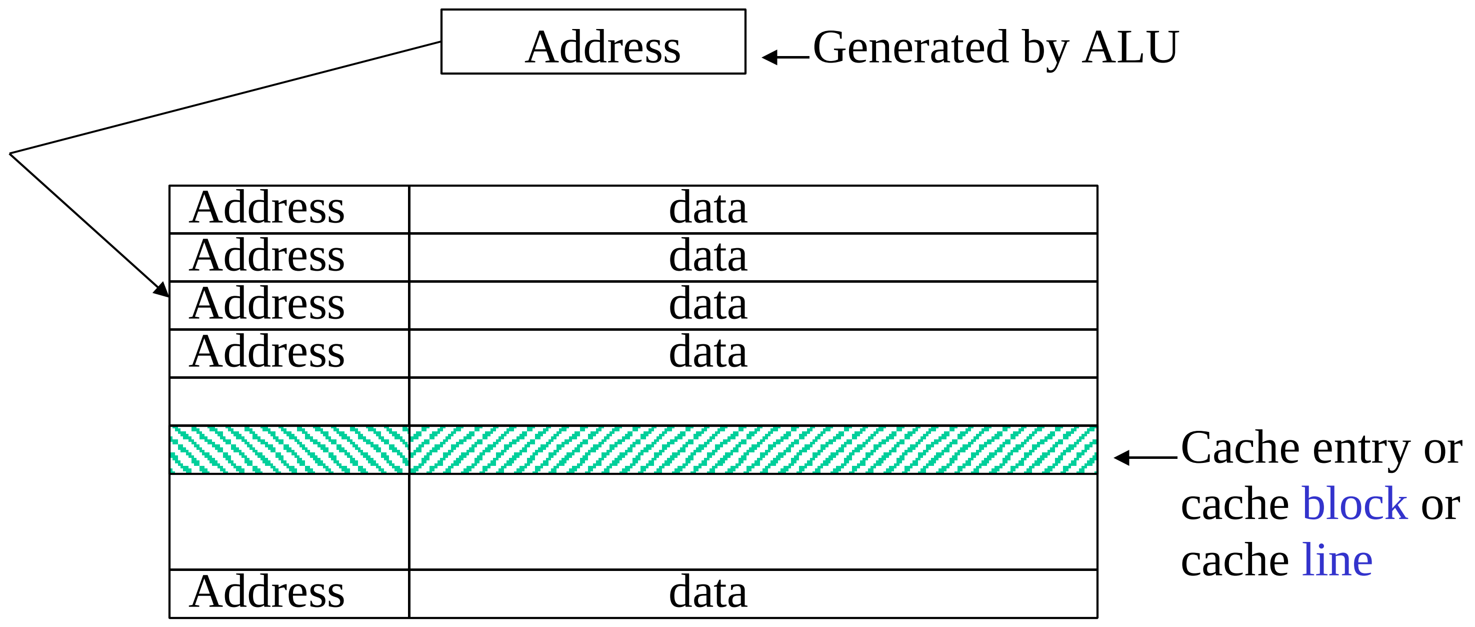
Some basic questions on cache design

- When do we bring the contents of a memory location in the cache?
- Where do we put it?
- How do we know it’s there?
- What happens if the cache is full and we want to bring something new?
  - In fact, a better question is “what happens if we want to bring something new and the place where it’s supposed to go is already occupied?”

Some “top level” answers

- When do we bring the contents of a memory location in the cache? -- The first time there is a cache miss for that location, that is “*on demand*”
- Where do we put it? -- Depends on *cache organization* (see next slides)
- How do we know it’s there? -- Each entry in the cache carries its own name, or *tag*
- What happens if the cache is full and we want to bring something new? One entry currently in the cache will be *replaced* by the new one

Generic cache organization



Address
or tag

If address (tag) generated by ALU = address (tag) of a cache entry, we have a cache hit; the data in the cache entry is good

Cache organizations

- Mapping of a memory location to a cache entry can range from full generality to very restrictive
  - In general, the data portion of a cache block contains several words
- If a memory location can be mapped anywhere in the cache (full generality) we have a *fully associative* cache
- If a memory location can be mapped at a single cache entry (most restrictive) we have a *direct-mapped* cache
- If a memory location can be mapped at one of several cache entries, we have a *set-associative* cache

How to check for a hit?

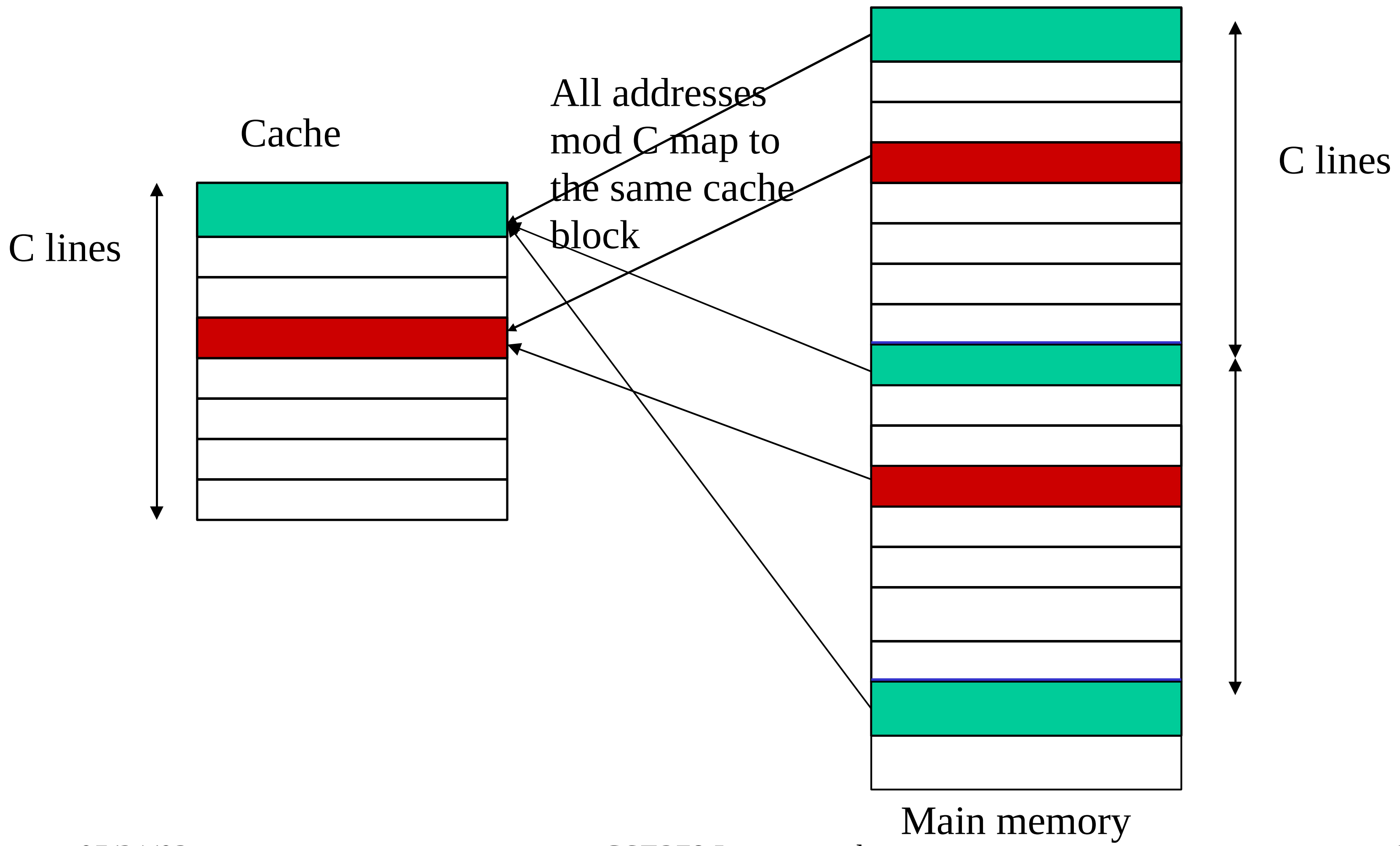
- For a fully associative cache
  - Check all tag (address) fields to see if there is a match with the address generated by ALU
  - Very expensive if it has to be done fast because need to perform all the comparisons in parallel
  - Fully associative caches do not exist for general-purpose caches
- For a direct mapped cache
  - Check only the tag field of the single possible entry
- For a set associative cache
  - Check the tag fields of the set of possible entries

Cache organization -- direct-mapped

- Most restricted mapping
  - *Direct-mapped* cache. A given memory location (block) can only be mapped in a single place in the cache. Generally this place given by:

(block address) mod (number of blocks in cache)

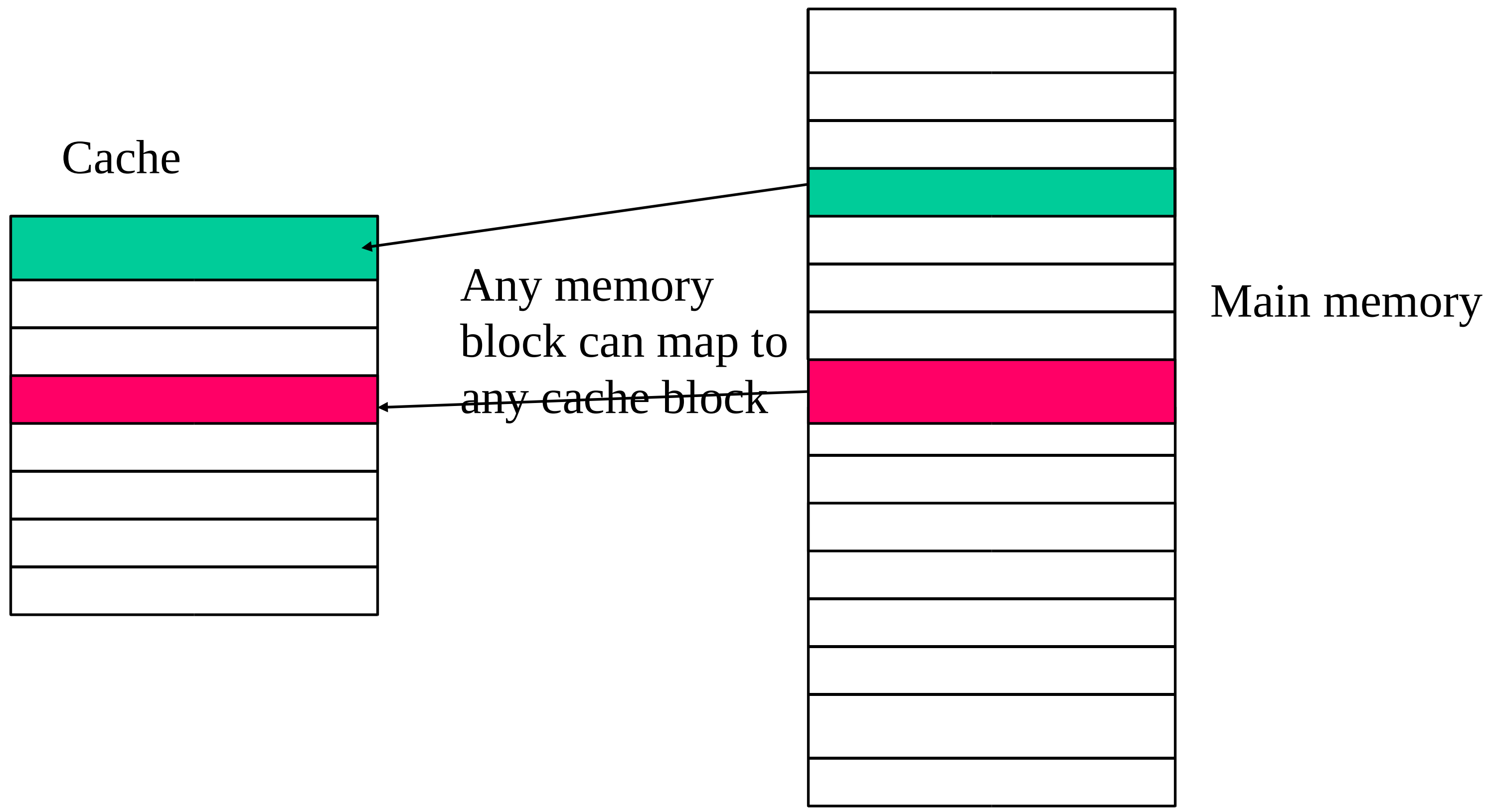
Direct-mapped cache



Fully-associative cache

- Most general mapping
  - *Fully-associative* cache. A given memory location (block) can be mapped anywhere in the cache.
  - No cache of decent size is implemented this way but this is the (general) mapping for pages from virtual to physical space (disk to main memory, see later) and for small TLB's (this will also be explained soon).

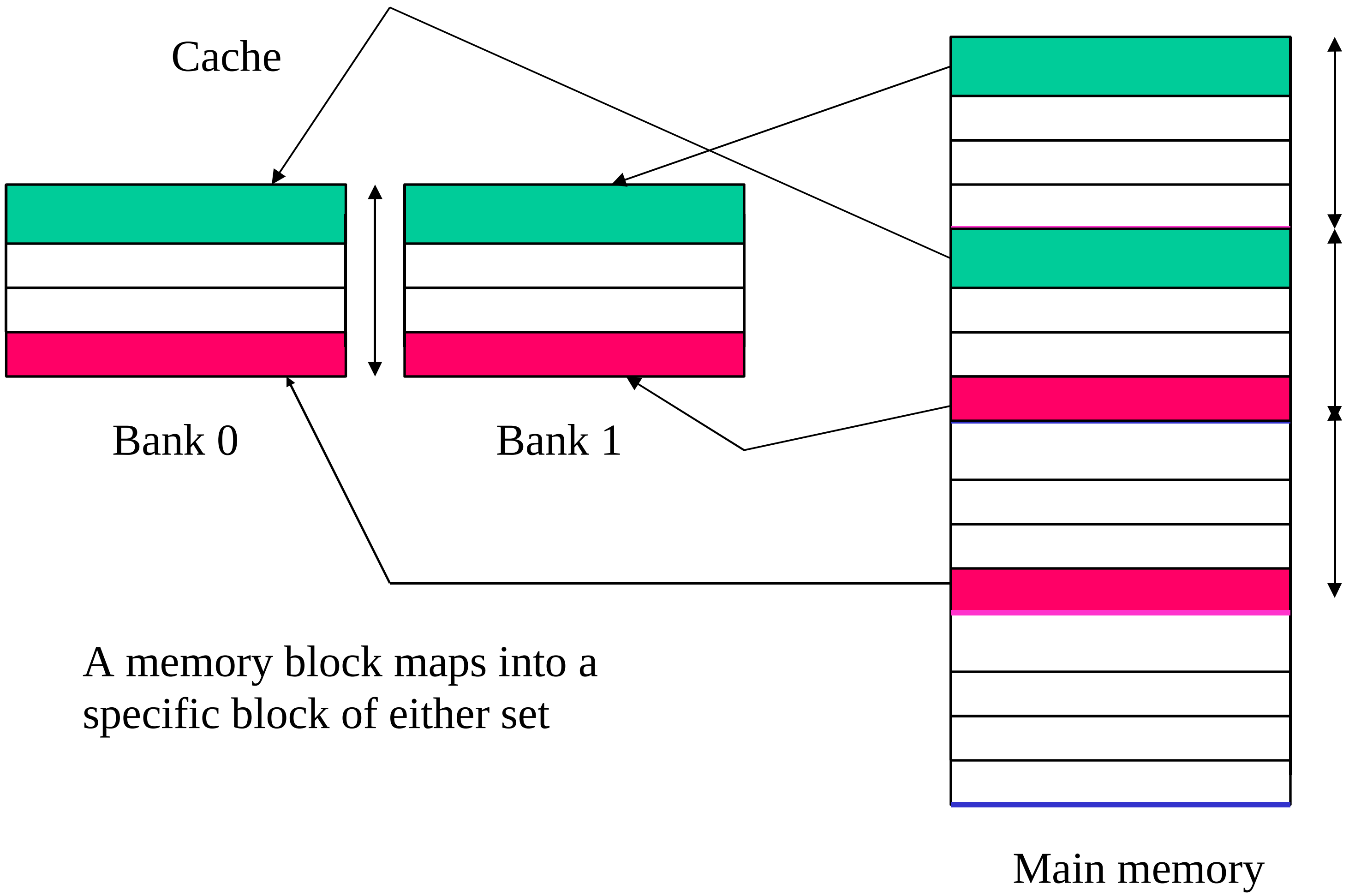
Fully-associative cache



Set-associative caches

- Less restricted mapping
  - *Set-associative* cache. Blocks in the cache are grouped into sets and a given memory location (block) maps into a set. Within the set the block can be placed anywhere. Associativities of 2 (two-way set-associative),4, 8 and even 16 have been implemented.
- Direct-mapped = 1-way set-associative
- Fully associative with m entries is m-way set associative

Set-associative cache

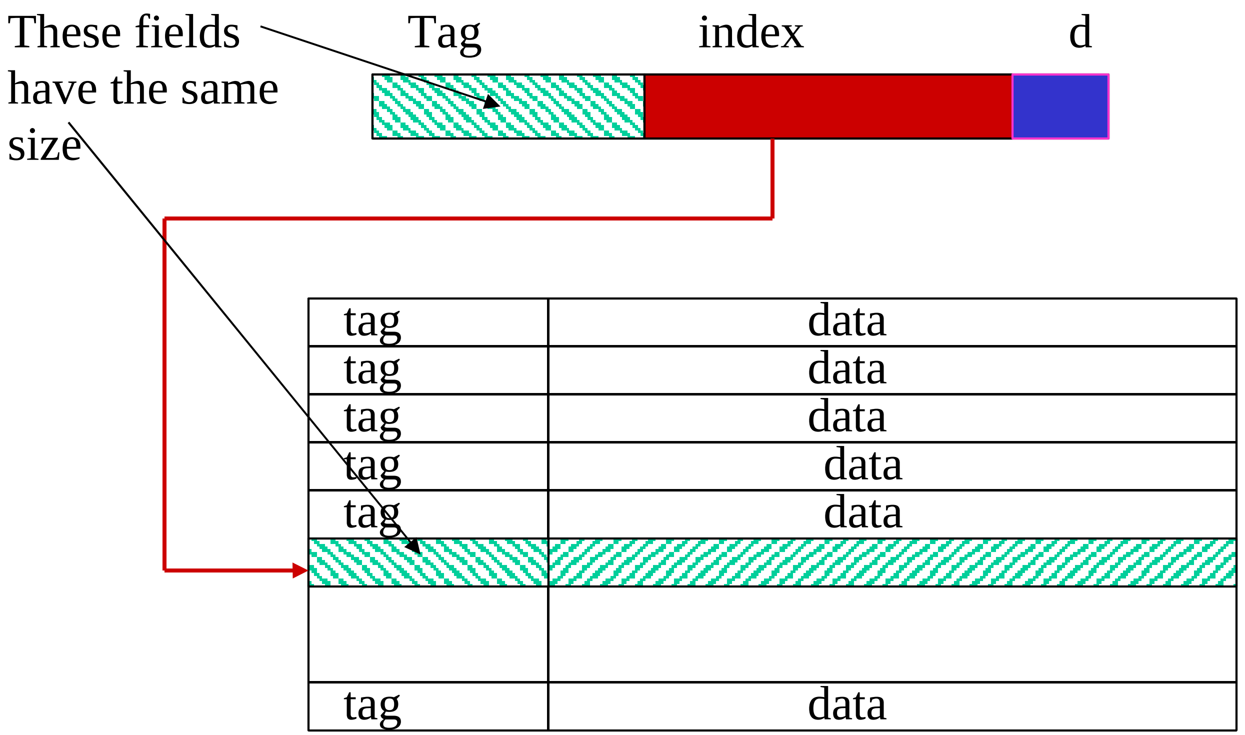


A memory block maps into a specific block of either set

Cache hit or cache miss?

- How to detect if a memory address (a byte address) has a valid image in the cache:
- Address is decomposed in 3 fields:
  - *block offset* or *displacement* (depends on block size)
  - *index* (depends on number of sets and set-associativity)
  - *tag* (the remainder of the address)
- The tag array has a width equal to *tag*

Hit detection (direct-mapped cache)



d corresponds to number of bytes in the block;

index corresponds to number of blocks in the cache;

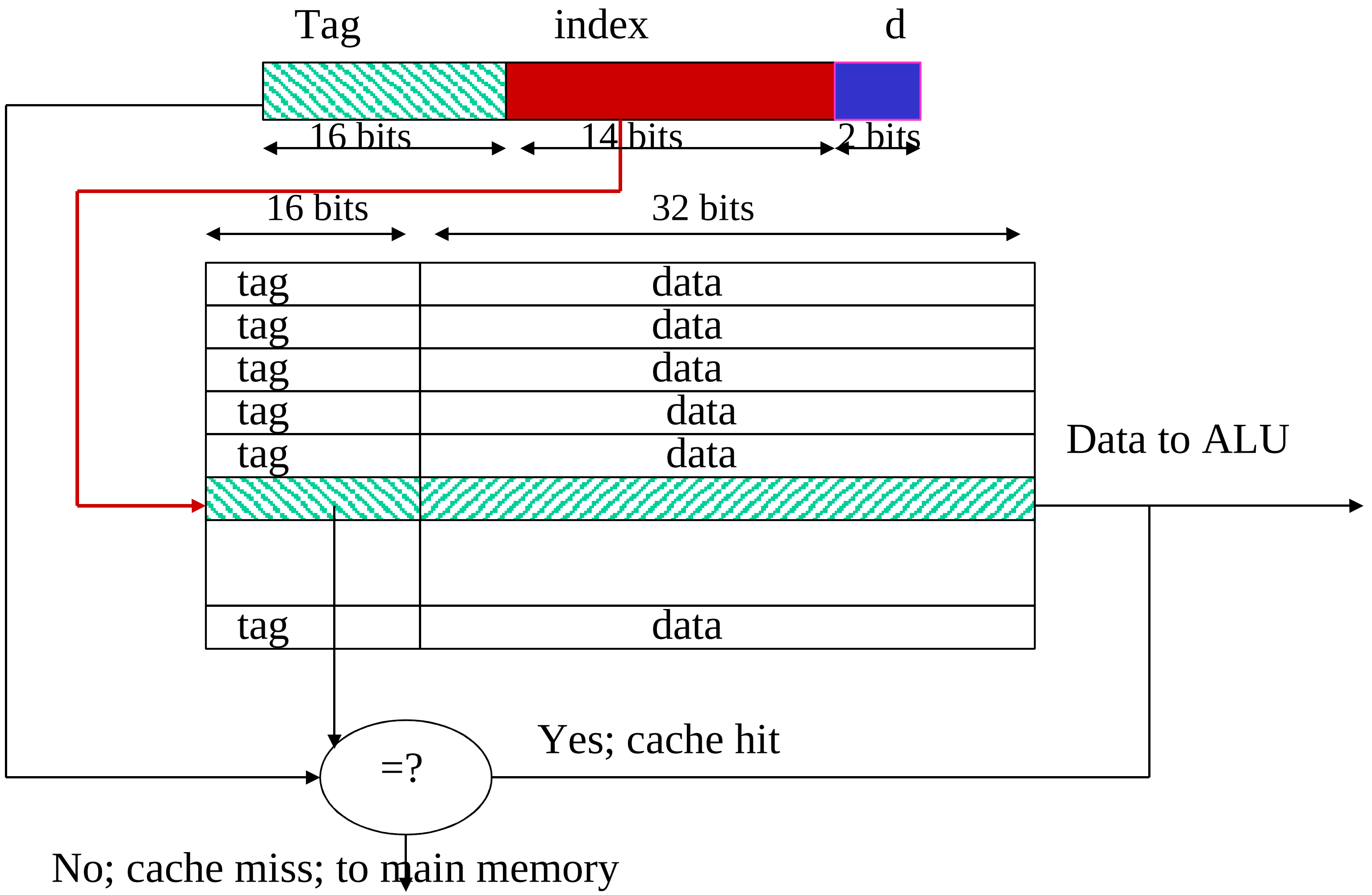
tag is the remaining bits in the address.

If tag(gen. address) = tag(entry pointed by index in cache), we have a hit

Example of a direct-mapped cache

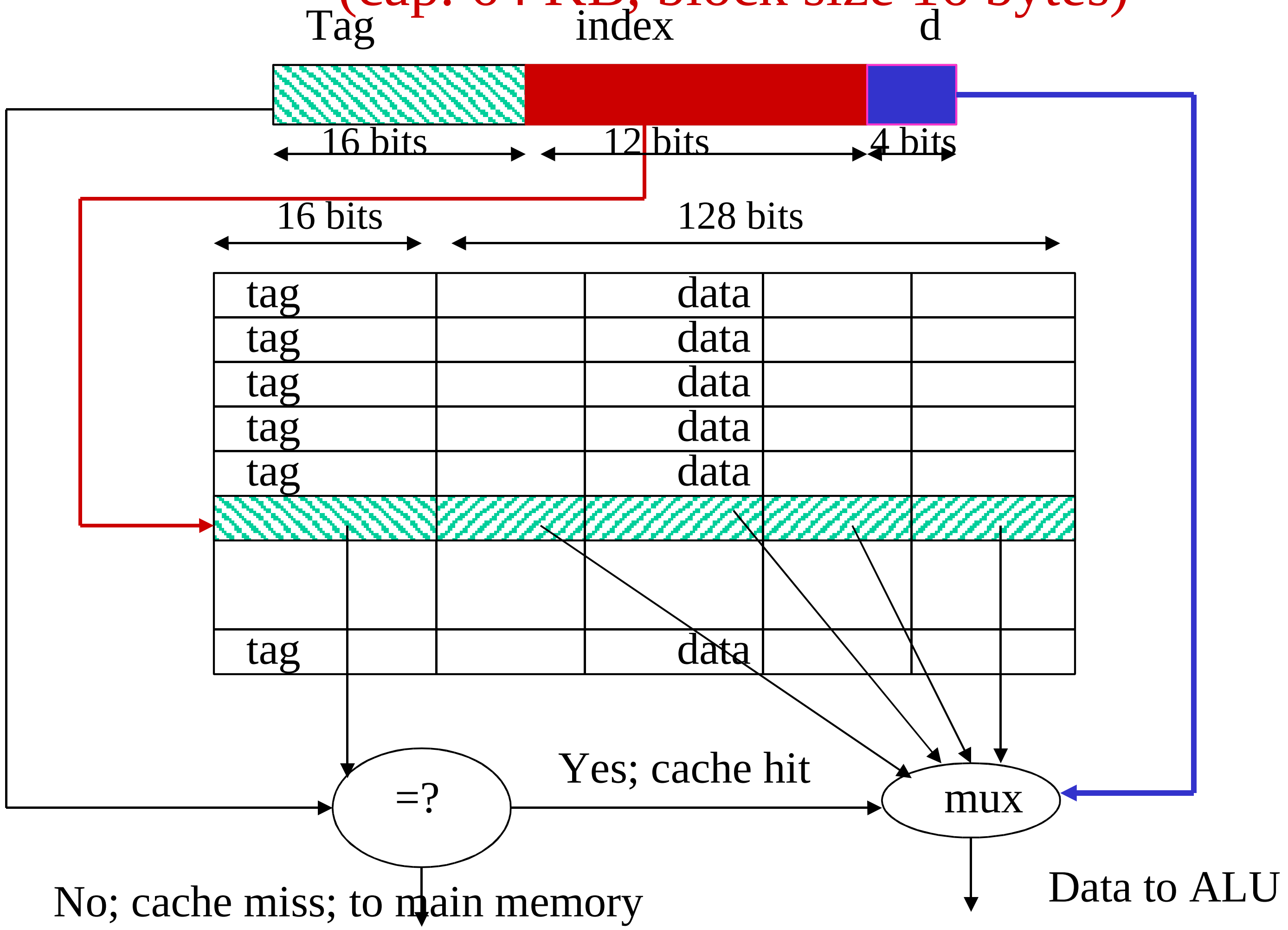
- DEC Station 3100
  - 64 KB (when giving the *size*, or *capacity*, of a cache, only the data part is counted)
  - Each block is 4 bytes (*block size*); hence 16 K blocks (aka lines)
  - displacement field: *d* = 2 bits (d = $\log_2$ (block size))
  - index field: *i* = 14 bits (i = $\log_2$ (nbr of blocks))
  - tag field : *t* = 32 - 14 -2 = 16 bits

Dec 3100


Tag
index
d
16 bits
14 bits
2 bits
16 bits
32 bits
tag
data
tag
data
tag
data
tag
data
tag
data
tag
data
Data to ALU
=?
Yes; cache hit
No; cache miss; to main memory

Exxample of a Cache with longer blocks

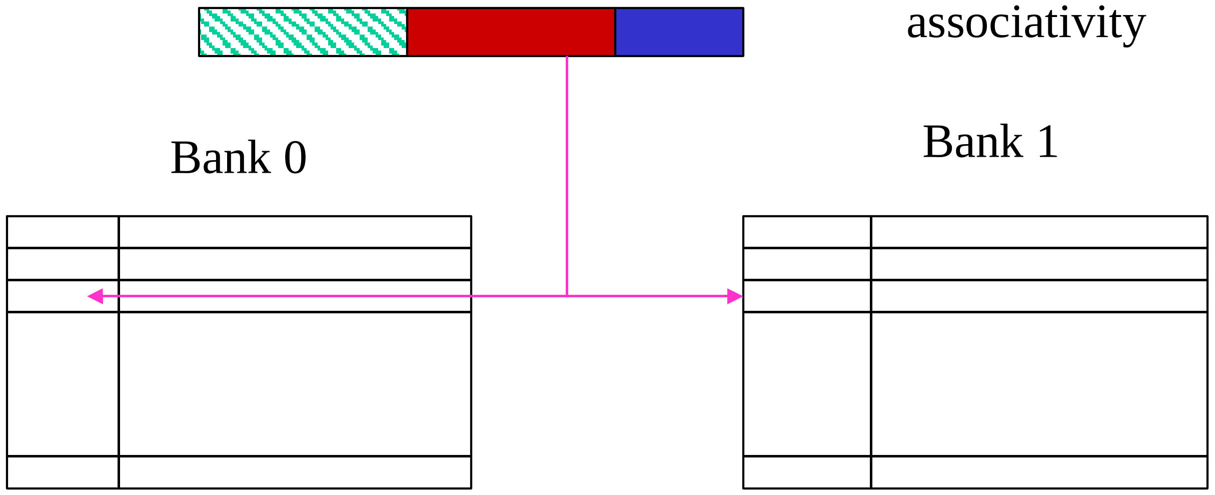
(cap. 64 KB, block size 16 bytes)



Why set-associative caches?

- Cons
  - The higher the associativity the larger the number of comparisons to be made in parallel for high-performance (can have an impact on cycle time for on-chip caches)
- Pros
  - Better hit ratio
  - Great improvement from 1 to 2, less from 2 to 4, minimal after that

Set associative mapping



<span style="color:red">Index</span> = log (number of blocks)/ associativity

Note: we need one comparator per bank + mux to see if we have a hit.