SISTEMI EMBEDDED

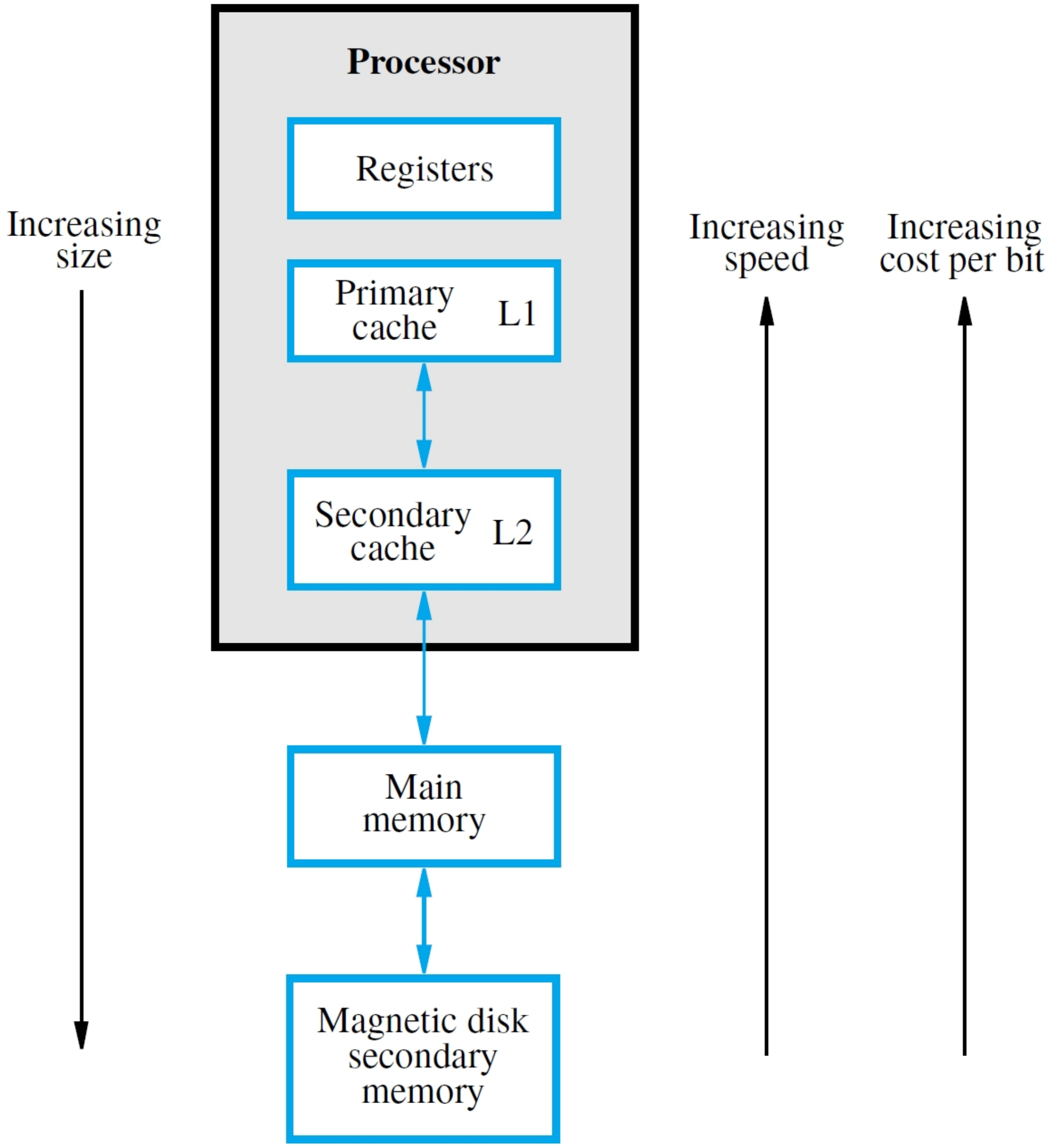
Computer Organization
Memory Hierarchy, Cache Memory

Federico Baronti

Last version: 20160524

Memory Hierarchy

- Ideal memory is fast, large, and inexpensive
- Not feasible with current memory technology, so use **memory hierarchy**
- Exploits **program behavior (*locality of reference*)** to make it *appear* as though memory is on average fast and large



Caches and Locality of Reference

- The cache is between processor and memory
- Makes large, slow main memory *appear* fast
- Typical program behavior involves executing instructions in loops and accessing data array
- Effectiveness is based on ***locality of reference***
  - Temporal locality: instructions/data that have been recently accessed are likely to be *again*
  - Spatial locality: *nearby* instructions or data are likely to be accessed after current access

More Cache Concepts

- To exploit spatial locality, transfer *cache block* (or *line*) with multiple adjacent words from memory
  - Later accesses to nearby words are fast, provided that cache still contains the block
- *Mapping function* determines where a block from memory is to be located in the cache
  - Direct or Associative mapping
- When cache is full, *replacement algorithm* determines which block has to be removed from the cache

Cache Operation

- Processor issues Read and Write requests as if it were accessing main memory directly
- But control circuitry first checks the cache
  - If desired information is present in the cache, a *read* or *write hit* occurs
- For a read hit, main memory is not involved; the cache provides the desired information
- For a write hit, there are two approaches:
  - Write-back or Write-through

Handling Cache Writes

- *Write-through protocol*: update cache & memory. Memory is always updated.
- *Write-back protocol*: only update the cache; memory updated later when block is replaced
  - *Write-back* scheme needs *modified* or *dirty bit* to mark blocks that are updated in the cache and need to be written in the main memory when they are replaced
- If same location is written repeatedly, then *write-back* is much better than write-through
  - Block memory update is often more efficient, even if writing back unchanged words

Handling Cache Misses

- If desired information is not present in cache, a *read* or *write* *miss* occurs
- For a read miss, the block with desired word is transferred from main memory to the cache
- For a write miss under write-through protocol, information is written to the main memory
- Under write-back protocol, first transfer block containing the addressed word into the cache. Then overwrite specific word in cached block

Mapping Functions

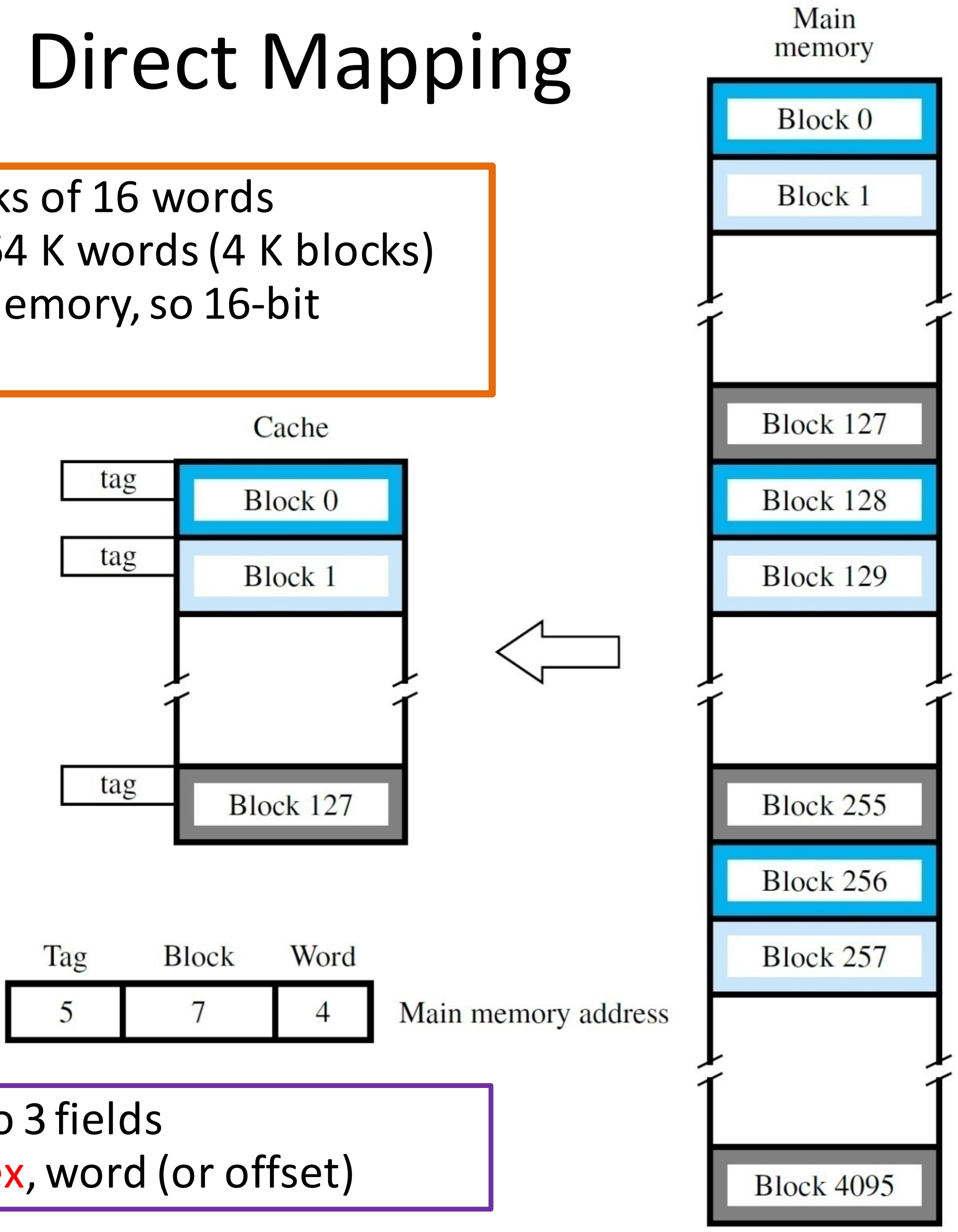
- Block of consecutive words in main memory must be transferred to the cache after a miss
- The *mapping function* determines the location of a block in the cache
- Three mapping functions:
  - Direct, Associative and Set Associative Mapping
- Let’s consider the following scenario:
  - Cache with 128 blocks of 16 words
  - Main memory with 64 K words (4 K blocks), *word*-addressable, so 16-bit address

Direct Mapping

- Simplest approach uses a fixed mapping: mem. block *j* → cache block(*j* mod 128)
- Only one unique location for each mem. block
  - Two blocks may contend for same location even if the cache is not fully utilized
  - New block always overwrites previous block

Direct Mapping

Cache with 128 blocks of 16 words
Main memory with 64 K words (4 K blocks)
Word-addressable memory, so 16-bit address



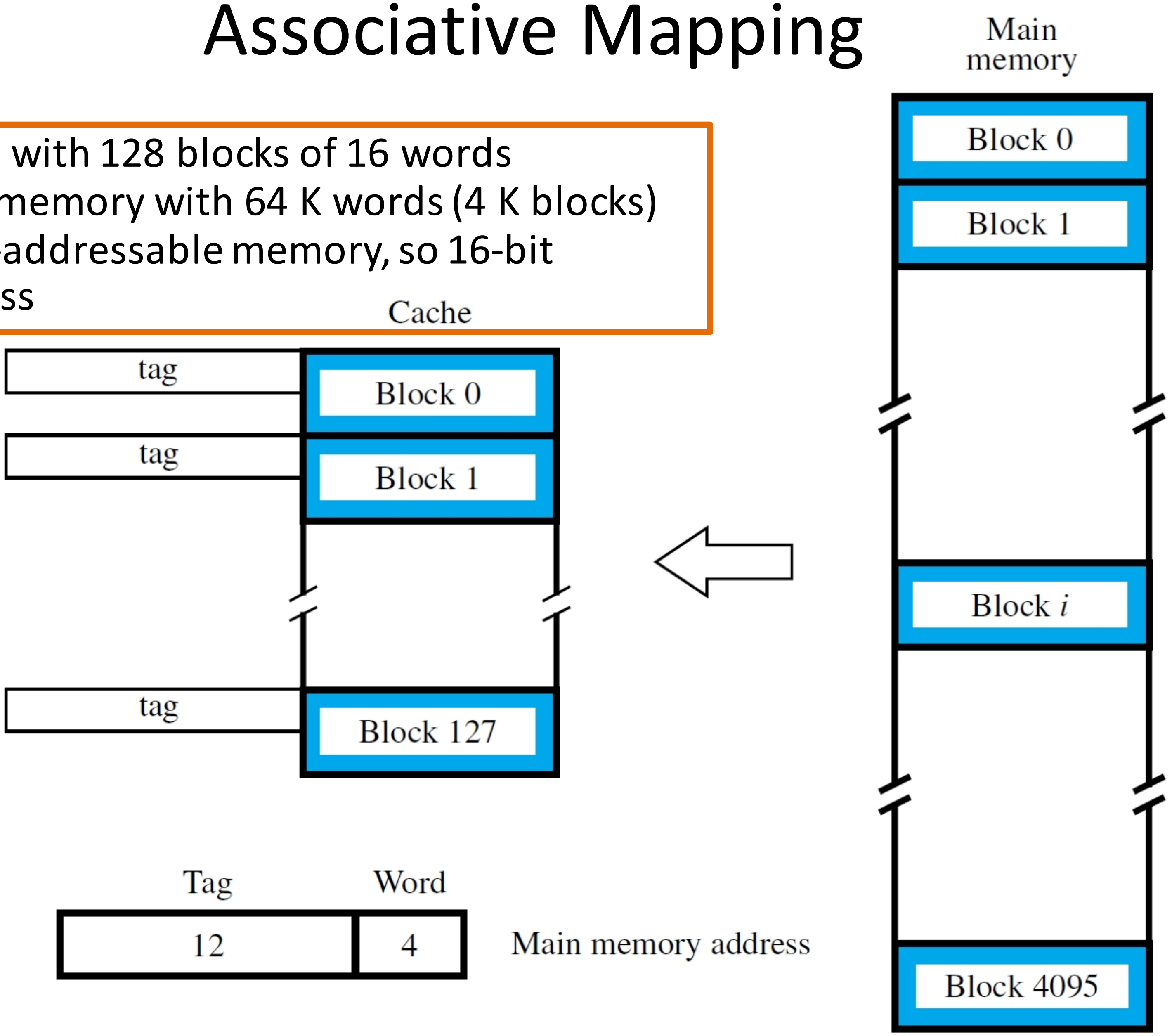
Address is divided into 3 fields
tag, block or line index, word (or offset)

Associative Mapping

- Full flexibility: locate block anywhere in cache
- Block field of address no longer needs any bits
- Tag field is enlarged to encompass those bits
- Larger tag stored in cache with each block
- For hit/miss, compare all tags simultaneously in parallel against tag field of given address
- This *associative search* increases complexity
- Flexible mapping also requires appropriate replacement algorithm when cache is full

Associative Mapping

Cache with 128 blocks of 16 words
Main memory with 64 K words (4 K blocks)
Word-addressable memory, so 16-bit address

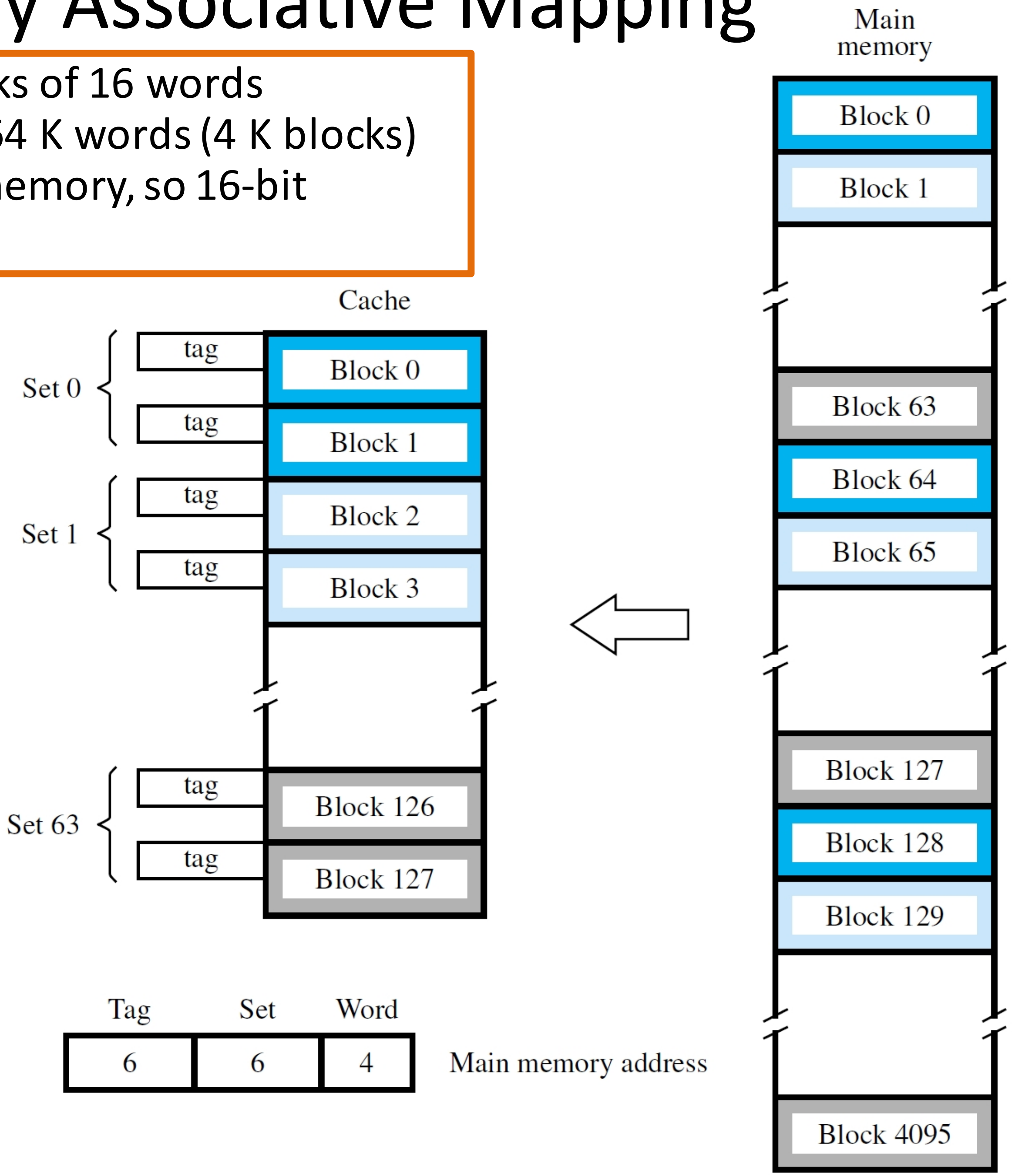


Set-Associative Mapping

- Combination of direct & associative mapping
- Group blocks of cache into *sets*
- Block field bits map a block to a unique set
- But any block within a set may be used
- Associative search involves only tags in a set
- Replacement algorithm is only for blocks in set
- Reducing flexibility also reduces complexity
- *k* blocks/set $\rightarrow$ *k*-way set-associative cache
  - Direct Mapping corresponds to 1-way
  - Associative Mapping corresponds to all-way

2-way Associative Mapping

Cache with 128 blocks of 16 words
Main memory with 64 K words (4 K blocks)
Word-addressable memory, so 16-bit address



Stale Data

- Each block has a valid bit, initialized to 0
- No hit if valid bit is 0, even if tag match occurs
- Valid bit set to 1 when a block placed in cache
- When power is turned on, all valid bits are set to 0
- Because of DMA, main memory can change w/o read or write performed by the processor
  - Invalid a cache block when corresponding block in memory is modified by DMA
  - If write-black transfers block from cache to memory before starting DMA that has such a block as source. This action can be achieved by flushing the cache.

LRU Replacement Algorithm

- Replacement is trivial for direct mapping, but needs a method for associative mapping
- Consider temporal locality of reference and use a ***least-recently-used*** *(LRU)* algorithm
- For *k*-way set associativity, each block in a set has a counter ranging from from 0 to *k*-1, which is updated w/ the following rules:
  - Hitting on a block clears its counter value to 0; others originally lower in set are incremented, and all the others remain unchanged
  - When a miss occurs and the set is not full, the counter of the new block is set to 0 and all the others are increased by one
  - When a miss occurs and the set is full, replace the block w/ counter = *k*-1, set its counter to 0 and increment by one all the other counters

Hit Rate and Miss Penalty

- Performance of a memory hierarchy are determined by the *hit rate* and the *miss penalty*
- *Hit rate* depends on the cache size and its organization (mapping function, block size)
- *Miss penalty* includes the time to detect the miss, transfer one block from the main mem. to the cache and eventually the requested word to the proc. It depends on the main memory access time, which is usually much larger for the first word of the block than for the remainder ones.
  - Let's assume that the cache access time is 1 clock cycle, the access for the first word in mem. is N_{first} = 7 *cycle* and for the following words N_{more} = 1 *cycle*, and the block size B = 8 *word*.
  - Then, the miss penalty
    $N_{miss} = (1 + 1x\, N_{first} + (B-1)x\, N_{more} + 1) - 1 = 15$,
    where 1 one cycle is for detecting the cache miss and another for providing the requested word to the proc.

Effect on Pipelining Performance

- Assume that: freq. of *cache* misses during fetch $p_{\text{miss-fetch}} = 5\ \%$, freq. of *cache* misses during mem access $p_{\text{miss-mem}} = 10\ \%$, freq. of Load and Store instr. $p_{\text{LD-ST}} = 30\ \%$. Then, :
  - $\delta_{\text{cache-miss}} = N_{\text{miss}} (p_{\text{miss-fetch}} + p_{\text{LD-ST}} \text{ x } p_{\text{miss-mem}})$
    $= 15\text{x}(0.05 + 0.03) = 1.2$
  - $F = R/2.2 = 0.45\text{R}$
- W/o cache, i.e., $p_{\text{miss-fetch}} = 100\ \%$ and $p_{\text{miss-mem}} = 100\ \%$, mem access time penalty is $N_{\text{first}} - 1$ *cycle*
  - $\delta_{\text{mem}} = (N_{\text{first}} - 1)\text{x}(1 + p_{\text{LD-ST}}) = 7.8$
  - $F = R/8.8 = 0.11\text{R}$
- Cache improves performance by a factor of 4.

References

- C. Hamacher, Z. Vranesic, S. Zaky, N. Manjikian "Computer Organization and Embedded Systems," *McGraw-Hill International Edition*
  - Chapter VIII: 8.5 – 8.7.1