

Computer Organization and Architecture: A Pedagogical Aspect
Prof. Jatindra Kr. Deka
Dr. Santosh Biswas
Dr. Arnab Sarkar
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 26
Direct-mapped Caches: Misses, Writes and Performance

(Refer Slide Time: 00:28)

Memory – Physical Issues

- Memory technologies – Access time, cost per GB
 - SRAM
 - 0.5 - 2.5 ns, \$2000 - \$5000 per GB
 - DRAM
 - 50 - 70 ns, \$20 - \$75 per GB
 - Magnetic disk
 - 5 - 20 ms, \$0.20 - \$2 per GB

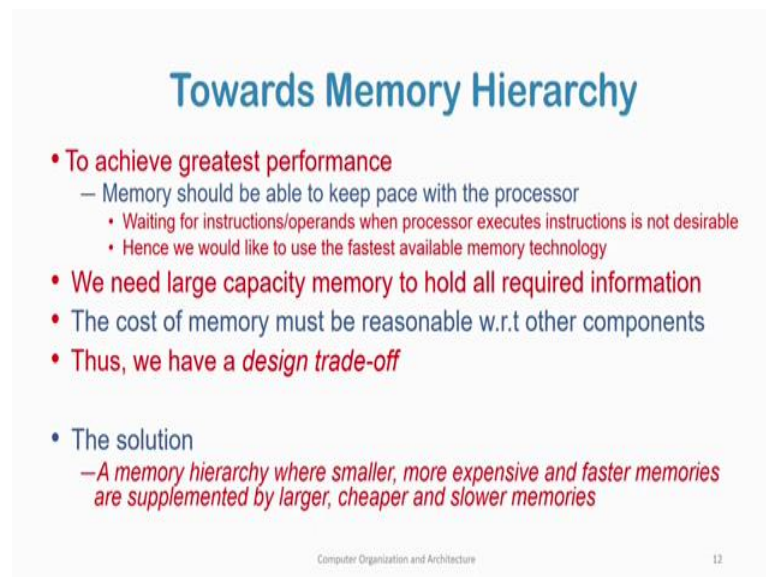
Computer Organization and Architecture 11

Unit 1 part 2: We ended part 1 of unit 1 by saying that we have different memory technologies which vary in terms of their access times and cost per GB. For example, we said that SRAMs are very fast and its speed is about one 0.5 to 2.5 nanoseconds, its access time; that means, it is on an average about one-tenth as fast as the processor ok. However, the cost per GB of this type of memories is also very huge. The cost per GB is about 2000 dollars to 5000 dollars.

Then we have DRAMs which are about 150 to 100 times slower than SRAMs; that means, to bring a certain amount of data a data unit a word from DRAM the processor will require about hundreds of processor cycles to do so. The speed of a DRAM is typically in the range of 50 to 70 nanoseconds; that is the access time is in the range of 50 to 70 nanoseconds. But, it is also about hundred times cheaper than SRAMs. So, the typical cost of DRAM units range in between 20 dollars to 75 dollars per GB.

Magnetic disks or hard disks are far cheaper; about 1000 times cheaper than DRAMs being only about 0.2 to 2 dollars per GB. However, it is also about 1000 times 100 times, 100 to 1000 times slower than DRAM units. Its access times ranges in between 5 to 20 milliseconds. So, to bring a data word from the hard disk, the processor requires tens of thousands of processor cycles.

(Refer Slide Time: 02:26)



Towards Memory Hierarchy

- To achieve greatest performance
 - Memory should be able to keep pace with the processor
 - Waiting for instructions/operands when processor executes instructions is not desirable
 - Hence we would like to use the fastest available memory technology
- We need large capacity memory to hold all required information
- The cost of memory must be reasonable w.r.t other components
- Thus, we have a *design trade-off*

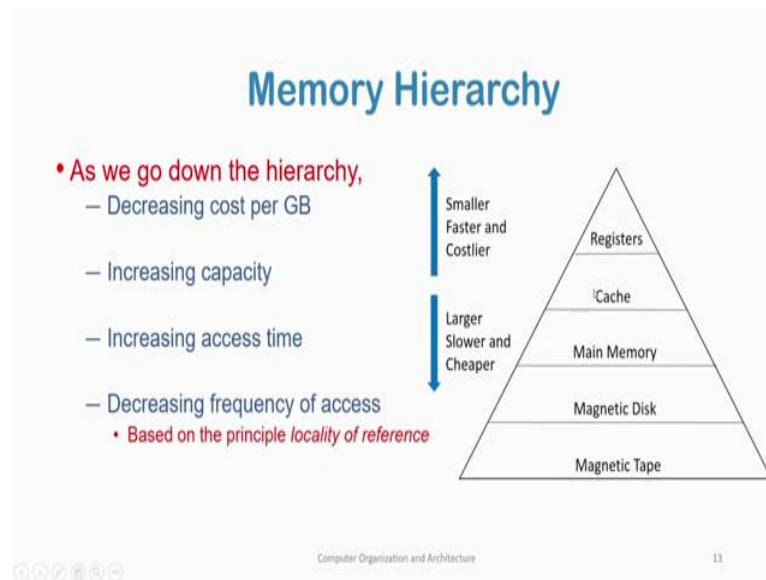
- The solution
 - *A memory hierarchy where smaller, more expensive and faster memories are supplemented by larger, cheaper and slower memories*

Computer Organization and Architecture 12

So, to achieve the best performance what would we desire? We would desire a very large capacity memory which can hold all our programs and data and which works at the pace of the processor. That means, if a processor requires a memory word in one cycle it is available in the processor from memory in the next cycle itself. However, in practice we saw the cost and performance parameters and it is difficult to achieve. So, to achieve the greatest performance memory should be able to keep pace with the processor. It is not desirable to wait for instruction slash operands when the processor executes instructions. And hence we would like to use the fastest available memory technology. We also need a large capacity memory to hold all our required information.

However, the cost of memory must be reasonable with respect to other components. Hence we understand that we have a design trade off. So, although the faster memories the mem although SRAMs are very fast in terms of access time, they are also very costly. The solution is to have memory hierarchy where smaller more expensive and faster memories are supplemented by larger, cheaper and slower memories.

(Refer Slide Time: 04:18)



Therefore, we have registers in the processor we typically have a few dozens of these registers and registers operate at the same speed as that of the processor. However, they are very expensive and we cannot have a large number of registers in the processor. Next in the hierarchy is cache. As I told it is about one tenth as fast as the processor speed; however, it is also very costly. Then we have the main memory which is slower than cache memory about hundreds of times slower than the processor speed and; however, its cost is cheaper than that of the cache. We have magnetic disks which are much cheaper than the main memory. However, its access times are also much slower and so on.

So, as we go down the hierarchy we have decreasing cost per GB, increasing capacity because it is cheaper we can have more capacity, more amount of that memory. We, but we also have increasing access times as we go down the hierarchy memories become slower. And we have decreasing frequency of access based on and this phenomenon that we have that we are able to have decreasing frequency of access towards in memories which are down the hierarchy is based on the principle of the locality of reference.

(Refer Slide Time: 06:00)

Principle of Locality

- **Programs access a small portion of the memory at a given time**
 - Programs typically contain a number of loops and subroutines
 - Within a loop/subroutine, a small set of instructions are repeatedly accessed
- **Temporal locality**
 - Items accessed recently are likely to be accessed again
 - eg. instructions in a loop
- **Spatial locality**
 - Items near to those accessed recently are likely to be accessed soon
 - eg. sequential access of data from array

Computer Organization and Architecture 14

Principle of the locality of reference is based on the fact that programs tend to access data and instructions and data in clusters, in the vicinity in the near vicinity at a of a given memory location. So, programs access a small portion of memory at a given time. Why? Because programs typically contain a large number of loops and subroutines, and within a loop or a subroutine a small set of instructions are repeatedly accessed. These instructions again tend to access data in clusters. So, there are two distinct principles in the locality of reference. Temporal locality which says that items accessed recently are likely to be accessed again. For example, the instructions within a loop.

So, if the instructions within in one iteration of the loop will be again accessed in the next iteration of the loop. And special locality in items near those accessed recently are likely to be accessed soon; for example, sequential access of data from an array. So, if you have a big array we tend to access data one by one from the array in sequence. So, how does this principle of the locality of reference helps to maintain this hierarchical memory organization?

(Refer Slide Time: 07:17)

Taking Advantage of Locality

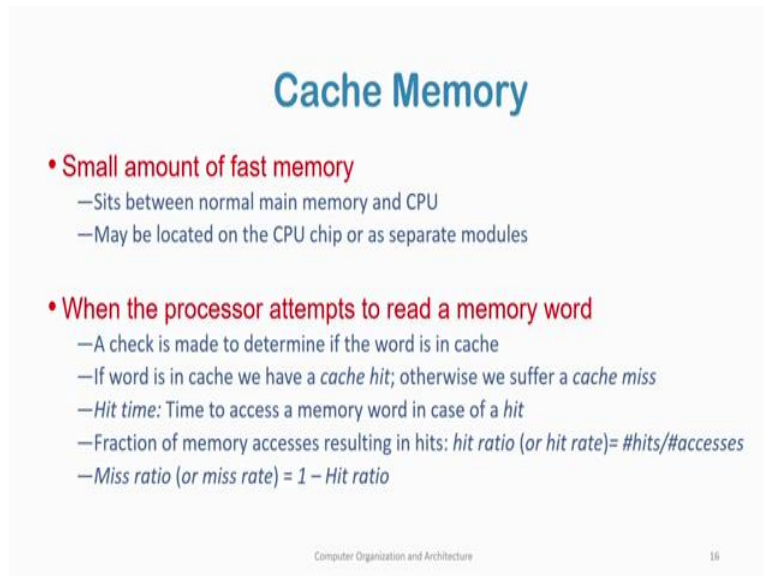
- Makes hierarchical organization of memory possible
 - Store everything in the magnetic disk
 - Copy recently accessed and near by data in small DRAM memory (main memory) from disk
 - Copy more recently accessed and near by data in a smaller SRAM memory (cache) from DRAM

Computer Organization and Architecture 15

So, principle of locality makes hierarchical organization of memory possible. So, how can we do that? For example, we can store everything we store everything in the magnetic disk. And then we copy recently accessed and nearby data in a small DRAM memory or the main memory. So, the main memory uses this technology of DRAM from the disk.

So, we have the magnetic disk which stores everything and whatever we require currently we access, we access it and store it in a DRAM or the main memory. Then whatever is still more recently accessed data and instructions are stored in an SRAM memory which is cache from the DRAM.

(Refer Slide Time: 08:20)



Cache Memory

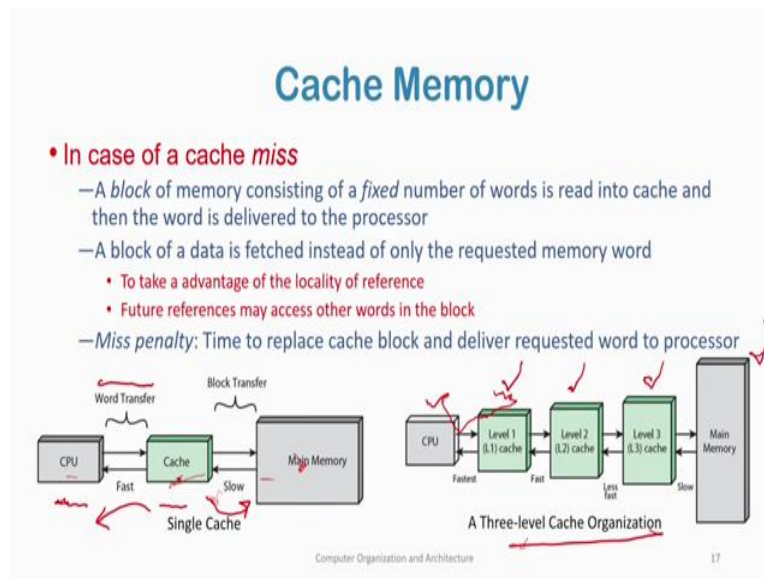
- **Small amount of fast memory**
 - Sits between normal main memory and CPU
 - May be located on the CPU chip or as separate modules
- **When the processor attempts to read a memory word**
 - A check is made to determine if the word is in cache
 - If word is in cache we have a *cache hit*; otherwise we suffer a *cache miss*
 - Hit time*: Time to access a memory word in case of a *hit*
 - Fraction of memory accesses resulting in hits: *hit ratio (or hit rate)* = $\frac{\text{hits}}{\text{accesses}}$
 - Miss ratio (or miss rate)* = $1 - \text{Hit ratio}$

Computer Organization and Architecture

Cache memory: So now we begin our discussion on cache memory. So, cache memory as we said is based on the SRAM memory technology. It's a small amount of fast memory which sits between the main memory and the CPU and it may be located within the CPU chip or a separate modules which are plugged in on the motherboard. So, when the processor attempts to read a memory word from the main memory, it what does it do? It places the address of the memory word from where on the address bus. Then what is done? A check is made to determine if the word is in cache. If the word is in cache we have a cache hit otherwise we suffered a cache miss. What is the hit time? The time to access a memory word in memory word in case of a hit is the hit time. So, fraction of memory accesses resulting in hits is called the hit ratio or the hit rate and is defined as number of cache hits over a certain given number of accesses on the memory.

Miss ratio or miss rate is; obviously, 1 minus the hit ratio. In case of a cache miss a block of memory consisting of a fixed number of words is read into the cache and then the word is delivered to the processor. A block of memory is fetched instead of only the requested memory word to take advantage of the locality of reference. Future references may access other words in the block ok. A block of data is fetched instead of only the requested memory word to take advantage of the locality of references because future references may access other words in the block. And in that case when those future references are made we will have a cache hit. Miss penalty: the time to replace a cache block and deliver requested word to the processor is known as miss penalty.

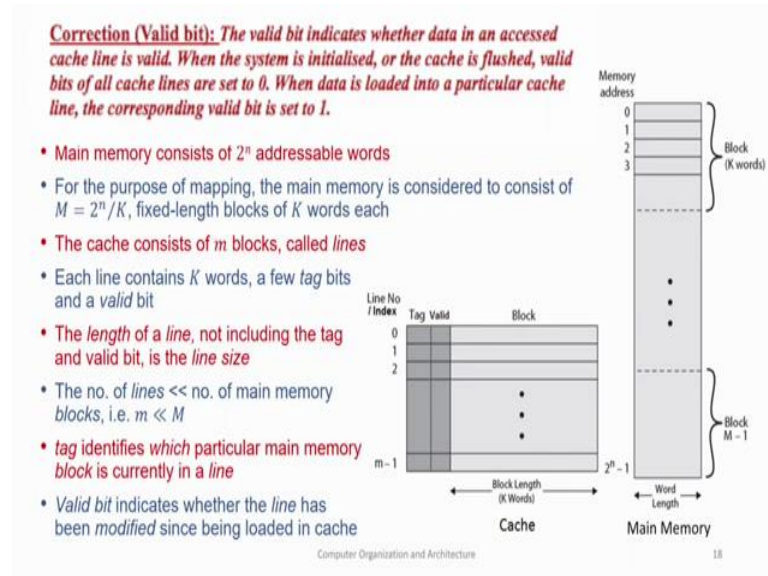
(Refer Slide Time: 09:50)



So, here in the figure on the left we see that CPU asks for a word from memory and if that word is present in cache, we send it back to the CPU and this is word transfer. If this word is not present in cache we have a cache miss and then we fetch a block from main memory and this block contains the desired word also. So, between cache and main memory we have block transfer whereas, between CPU and cache we have word transfer. The figure on the right shows that we may have different levels of cache not only a single level of cache.

So, we have CPU followed by followed by a small a small very fast cache, followed by a level 2 cache which is slower than level one cache, but is also higher in capacity. We also may have level 3 cache which is higher in capacity than level 2 cache, but is also slower and then finally, we have the main memory.

(Refer Slide Time: 12:00)



Let us assume that we have an n bit address bus. Therefore, we have a main memory consisting of 2^n addressable words. For the purpose of mapping the main memory is considered to consist of $M = 2^n / K$ fixed length blocks of K words each. So, we have a main memory which consists of 2^n words or bytes and a block consisting of K words or bytes each. And each so the number of blocks we have in main memory is given by $2^n / K$. The cache contains m blocks called lines. Each line contains K words same as the same as the block size plus a few tag bits and a valid bit.

The length of a line not including the tag and valid bit is the line size. The number of lines in cache is much much less than the main memory block size; that is small m is much much less than M . The tag identifies which particular main memory block is currently in a line; so therefore, right. The valid bit indicates whether the line has been modified since being loaded in cache.

(Refer Slide Time: 13:38)

Mapping Function

- $m \ll M$
 - we need a mechanism for mapping main memory blocks to cache lines
- **Direct Mapping – The simplest**
 - Each main memory block may be mapped to a single unique cache line
 - The mapping: $i = j \text{ modulo } m$
 - i = cache line no., j = main memory block no., m = no. of cache lines

Computer Organization and Architecture

19

Since, m is much much less than M ; that is the number of lines in cache is much much less than the number of blocks in the main memory we need a mechanism for mapping main memory blocks to cache lines. Therefore, we have a mapping function. The simplest mapping function is called direct mapping. In this each main memory block may be mapped to a single unique cache line and the mapping function is given by $i = j \text{ modulo } m$; where i is the cache line number, j is the main memory block number and m is the number of cache lines.

So, in this example that we the figure that we have at the bottom the cache has 8 lines and the main memory has 16 blocks. And we see that blocks 0 and block number 16, block number 0 and blocks number 16 maps both map to cache line number 0. Similarly, block number 15 as well as block number similarly block number 7 and block number 15 both map to line number 7.

(Refer Slide Time: 15:03)

Direct Mapping

- For purposes of cache access
 - Each main memory address may be viewed as consisting of $(s + w)$ bits
 - w LSBs = identify unique word (or byte) within a main memory block
 - Block size = Line size = 2^w bytes
 - s MSBs = block id; one of 2^s main memory blocks
 - Given size of cache as $m = 2^r$
 - r = Determines line no. or cache index no.
 - $(s - r)$ MSBs of main memory address = size of tag
 - Thus the main memory address has three parts:

Tag	Index	Word
$s-r$ bits	r bits	w bits

Computer Organization and Architecture

For the purposes of cache access, when we want to read the cache, each main memory address may be viewed as consisting of $s + w$ bits ok. So, we have a main memory consisting of $s + w$ bits. So, here this is s and this is w . Each main memory address may be viewed as consisting of $s + w$ bits. In which the w LSBs, the least significant bits identify a unique word within or byte within a main memory block. The block size is equal to the line size and is 2^w bytes ok. Because there are w LSBs, we have 2^w addressable bytes within a block or line. The s MSBs equals to is the block id, the most significant s bits are the block id. So, it identifies one of 2^s main memory blocks. Given the size of cache equals to $m = 2^r$. So, we have let the number of lines in cache equals to m and this m it is equals to 2^r ; r determines the, determines a line number or the cache index number ok.

So, r bits are used to determine the line number or cache index number. So, $s - r$ bits or the MSBs $s - r$ MSBs of the main memory address gives the size of the tag field. And thus the main memory address has 3 parts; the tag field which is the $s - r$ MSBs, the next r bits identify a line in cache and the least significant w bits identify a word in the main memory.