Pipelining

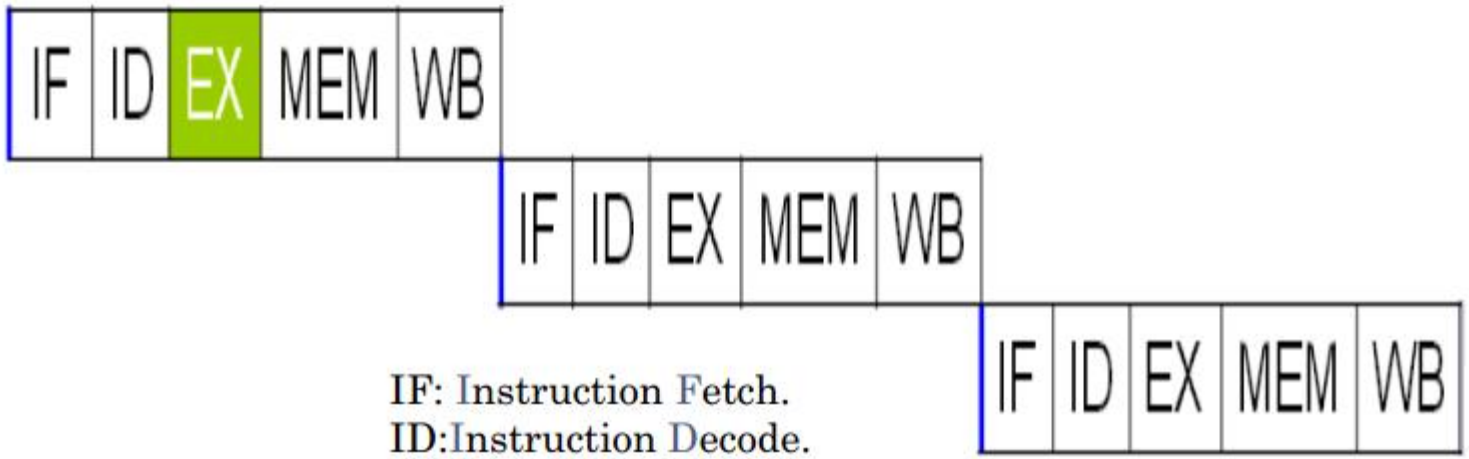
Increasing the CPU Throughput

Pipelining

- A technique used in advanced microprocessors where the microprocessor begins executing a second instruction before the first has been completed.
- A Pipeline is a series of stages, where some work is done at each stage. The work is not finished until it has passed through all stages
- The pipeline is divided into segments and each segment can execute it operation concurrently with the other segments. Once a segment completes an operations, it passes the result to the next segment in the pipeline and fetches the next operations from the preceding segment

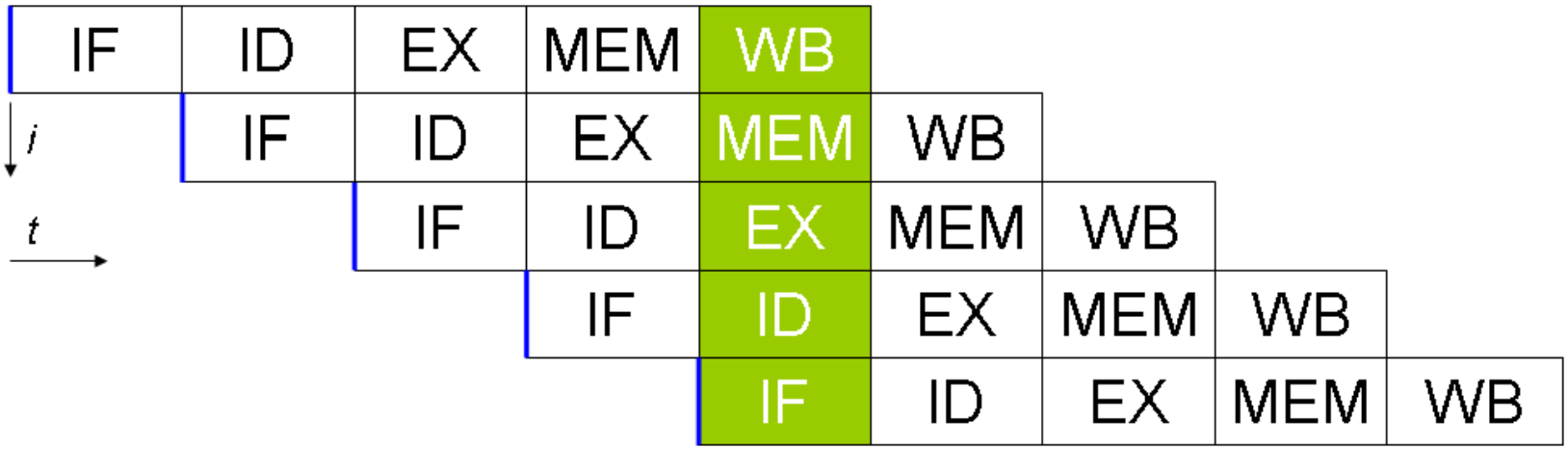
Without Pipelining

- Without pipelining the instructions are executed linearly.
- Until an instruction has completed its execution the next instruction does not start execution



With Pipelining

- An instruction starts the execution even though the previous instruction has not yet completed its execution.



Pipelining Requirement

- Pipelined organization requires sophisticated compilation techniques.
- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- Multiple tasks operating simultaneously using different resources
- Pipeline rate limited by slowest pipeline stage
- Each pipeline stage is expected to complete in one clock cycle

Pipeline Performance

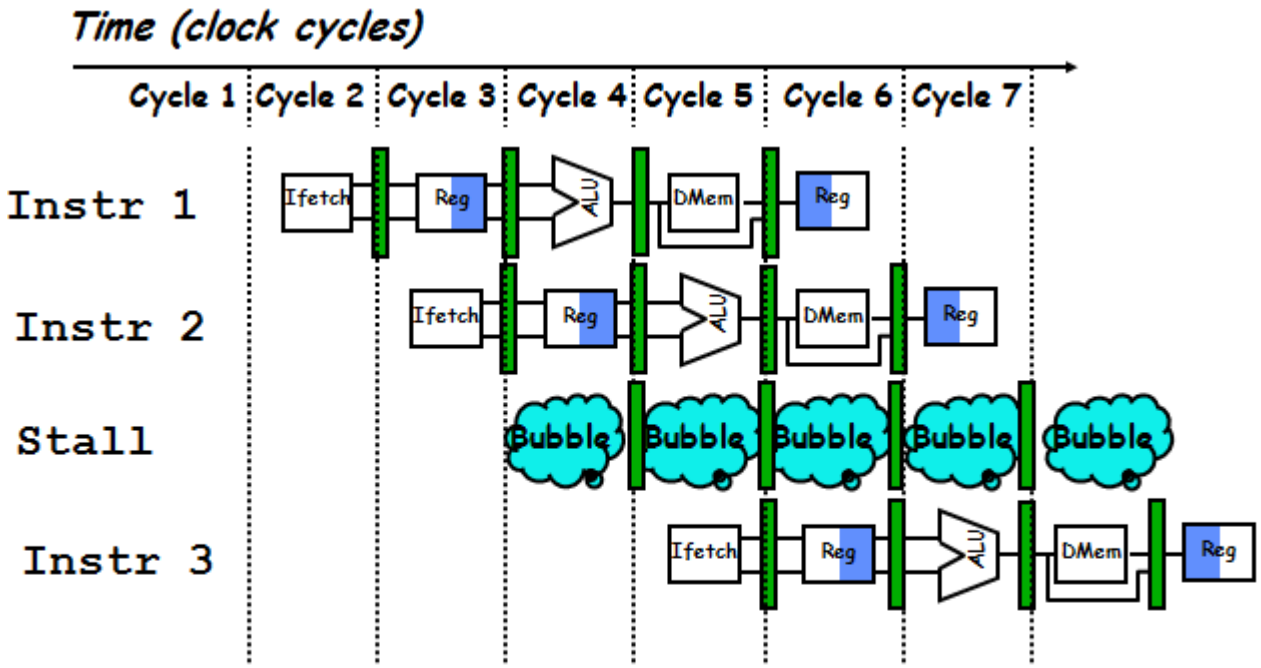
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases
- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- Suppose we execute 100 instructions
- Single Cycle Machine
  - 50 ns/cycle x 1 CPI x 100 inst = 5000 ns
- Ideal pipelined machine
  - 10 ns/cycle x (1 CPI x 100 inst + 5 cycle drain) = 1050 ns

Pipeline performance

- An *n*-stage pipeline has the potential to increase the throughput by *n* times.
- However, the only real measure of performance is the total execution time of a program.
- Higher instruction throughput will not necessarily lead to higher performance.
- Two questions regarding pipelining
  - How much of this potential increase in instruction throughput can be realized in practice?
  - What is good value of *n*?

Problems With Pipelining

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle and introduce stall cycles
- Stalling involves halting the flow of instructions until the required result is ready to be used. However stalling wastes processor time by doing nothing while waiting for the result.



Hazards in Pipelining

- Structural hazards: HW cannot support this combination of instructions - two dogs fighting for the same bone
- Data hazards: Instruction depends on result of prior instruction still in the pipeline
    - Data dependencies
- Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
    - Control dependencies

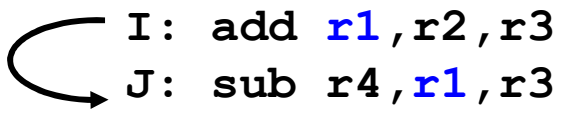
- Can always resolve hazards by stalling
- More stall cycles = more CPU time = less performance
  - Increase performance = decrease stall cycles

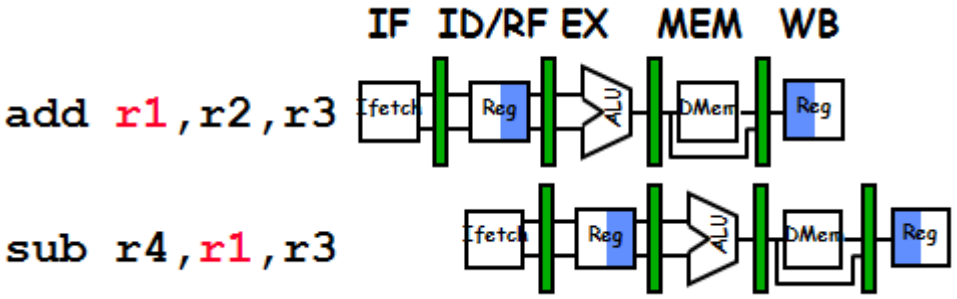
Data Dependencies

- Three types of data dependencies defined in terms of how succeeding instruction depends on preceding instruction
  - RAW: Read after Write or Flow dependency
  - WAR: Write after Read or anti-dependency
  - WAW: Write after Write

Data Dependencies RAW

- Read After Write (RAW)

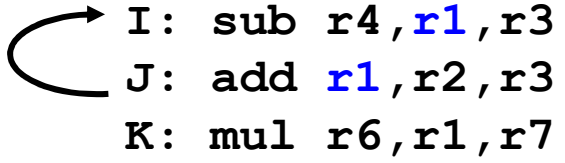


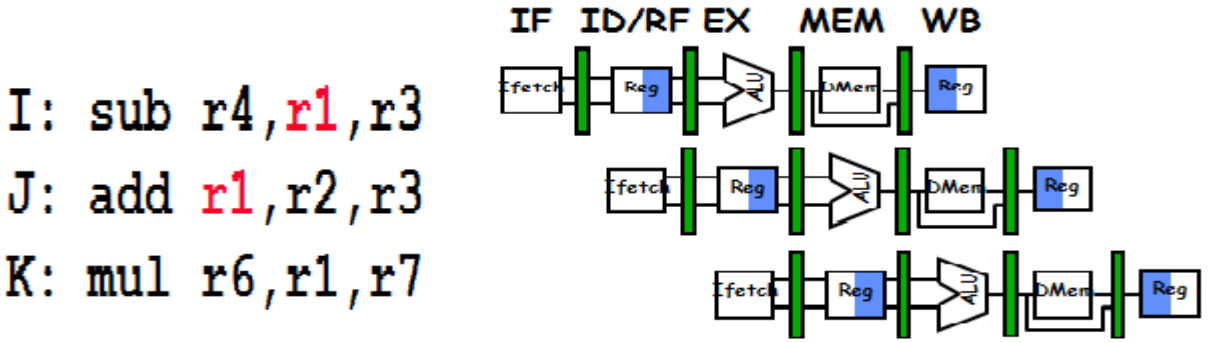


- Caused by a "Dependence" (in compiler nomenclature).

Data Dependencies WAR

- Write After Read (WAR)

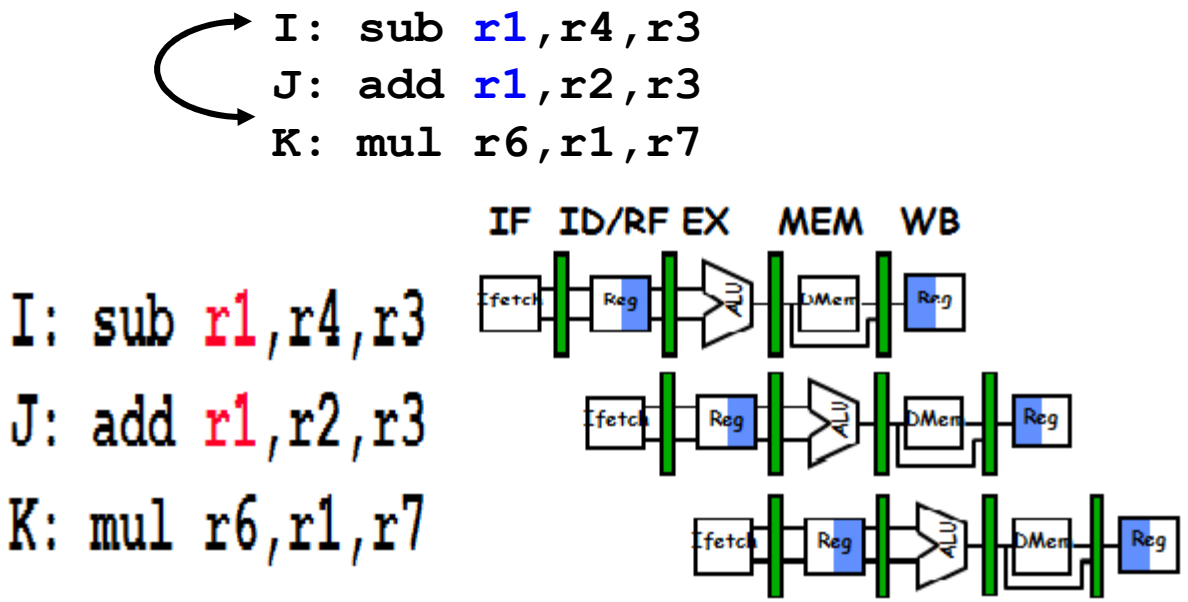
- 



- Called an "anti-dependence" by compiler writers.
  This results from reuse of the name "r1".

Data Dependencies WAW

- Write After Write (WAW)



- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".

Solution

- Problem:
  - i1: mul r1, r2, r3;
  - i2: add r2, r4, r5;
- Solution: Rename Registers
  - i1: mul r1, r2, r3;
  - i2: add r6, r4, r5;
- Register renaming can solve many of these
  - note the role that the compiler plays in this
  - specifically, the register allocation process--i.e., the process that assigns registers to variables

Control Hazards: Branches

- Instruction flow
  - Stream of instructions processed by Inst. Fetch unit
  - Speed of “input flow” puts bound on rate of outputs generated
- Branch instruction affects instruction flow
  - Do not know next instruction to be executed until branch outcome known
- When we hit a branch instruction
  - Need to compute target address (where to branch)
  - Resolution of branch condition (true or false)
  - Might need to ‘flush’ pipeline if other instructions have been fetched for execution

Solution to branch hazard

- Stall until branch direction is clear – flushing pipe
- Predict Branch Not Taken
  - Execute successor instructions in sequence
  - “Squash” instructions in pipeline if branch actually taken
  - 47% DLX branches not taken on average
- Predict Branch Taken
  - 53% DLX branches taken on average
- Delayed Branch
  - Define branch to take place AFTER a following instruction

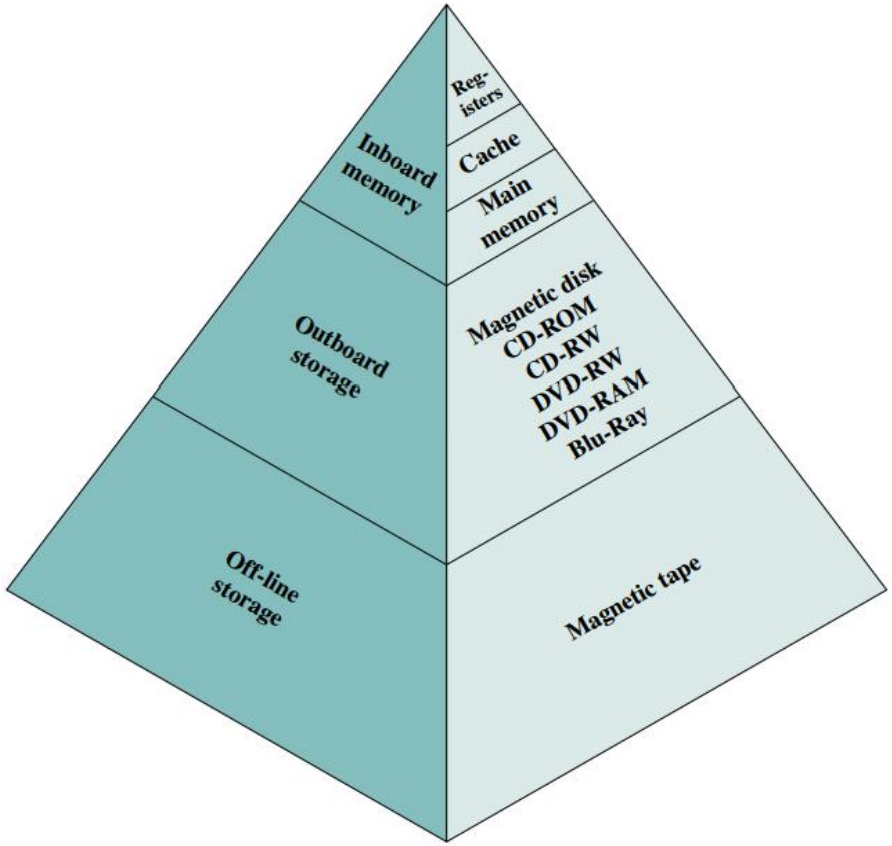
Memory Hierarchy

Memory

- The memory unit that communicates directly with the CPU is called main memory
- Devices that provide backup storage are called auxiliary memory
- Memory system can be classified according to the various characteristics ranging from the location, capacity, access method, performance, physical type, organization.
- The design constraint on a computer memory can be summed up by three questions
  - How Much?
  - How Fast?
  - How Expensive?
- There is a trade off among the three characteristics of memory capacity, access time and cost.
- The total memory capacity of a computer can be visualized as being a hierarchy of components from slow but high capacity auxiliary memory to relatively faster main memory, to even smaller and faster cache memory.

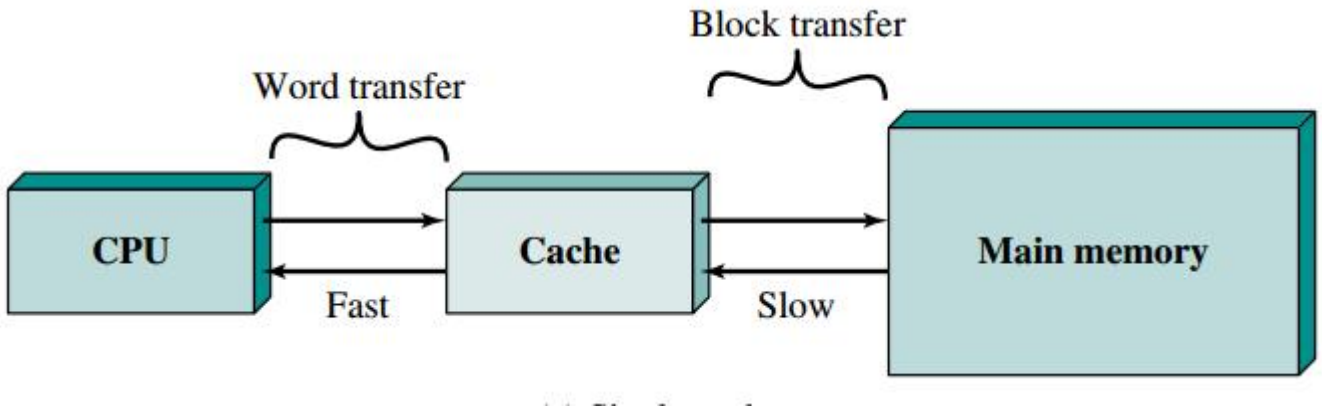
Memory Hierarchy

- The design is not to rely on a single memory component or technology but to employ memory hierarchy.
- Smaller, more expensive faster memories are supplemented by larger cheaper slower memories.
- The main factor in this organization is the decreasing frequency access of the memory by the processor.
- The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost.
- Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable magnetic disk, tape, and optical storage.
- The basis for this hierarchy is the principle known as **Locality of Reference**, a term for the phenomenon in which the same values, or related storage locations, are frequently accessed, depending on the memory access pattern



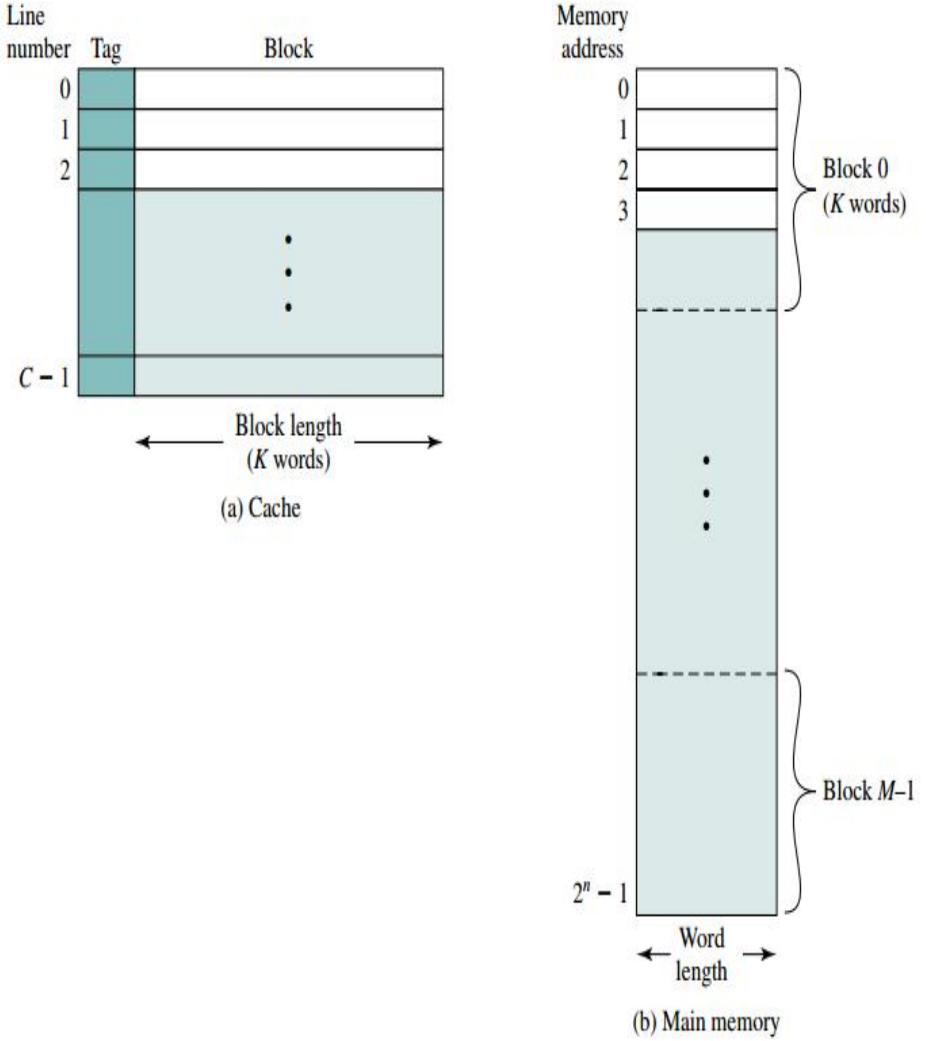
Cache Memory

- **CPU cache** is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory.
- The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations.
- When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor
- If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor
- Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block



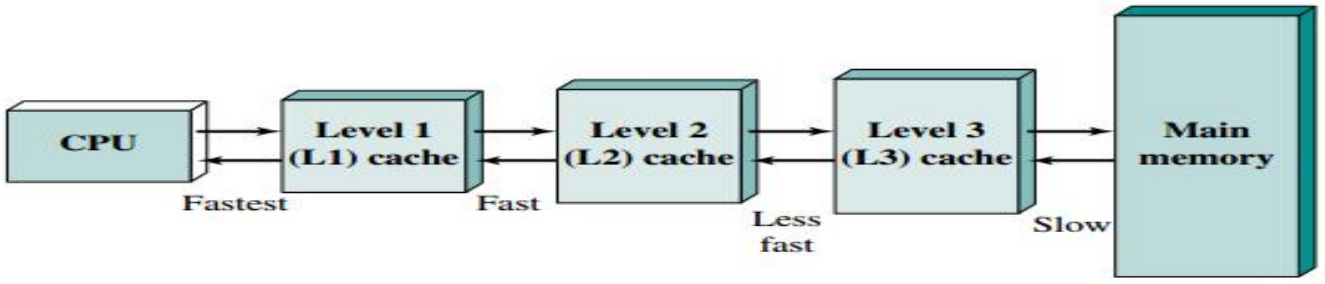
Cache Memory Structure

- Main memory consists of up to 2^n addressable words, with each word having a unique n bit address.
- This memory is considered to consist of a number of fixed length blocks of K words.
- Thus there are M= 2^n / K blocks in main memory
- The cache consists of D blocks called lines.
- Each line contains K words plus a tag of few bits.
- Each line include control bits to indicate whether the line has been modified or not.
- The number of lines D < M main memory blocks
- The processor generates the read address of a word to be read.
- If the word is contained in the cache it is delivered to processor.
- Otherwise the block containing the word is loaded into the cache from RAM and the word is delivered to the processor.



Types of Cache Memory

- A computer can have several different levels/types of cache memory.
- The level numbers refers to distance from CPU where Level 1 is the closest.
- All levels of cache memory are faster than RAM.
- The cache closest to CPU is always faster but generally costs more and stores less data then other level of cache.
- **Level 1 (L1) Cache**
  - It is also called primary or internal cache. It is built directly into the processor chip. It has small capacity from 8 K to 128 Kb
- **Level 2 (L2) Cache**
  - It is slower than L1 cache. Its storage capacity is more, i-e. From 64 Kb to 16 MB. The current processors contain advanced transfer cache on processor chip that is a type of L2 cache. The common size of this cache is from 512 kb to 8 Mb.
- **Level 3 (L3) Cache**
  - This cache is separate from processor chip on the motherboard. It exists on the computer that uses L2 advanced transfer cache. It is slower than L1 and L2 cache. The personal computer often has up to 8 MB of L3 cache.



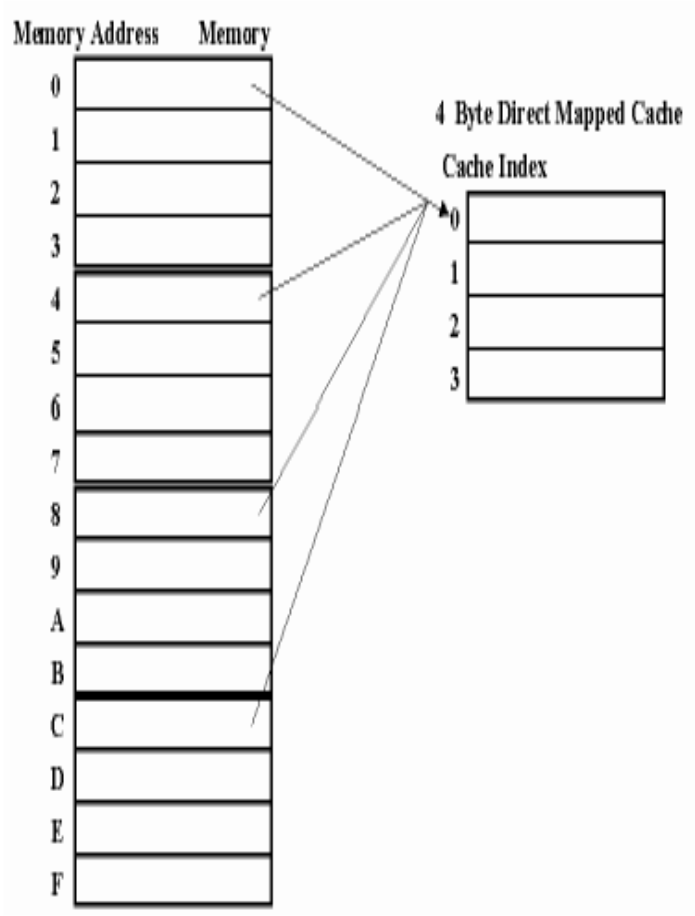
Three-level cache organization

Cache Mapping

- Since there are few cache lines to hold the memory blocks, an algorithm is needed for mapping the main memory blocks into cache lines.
- The choice of mapping functions dictate how the cache is organized.
- Three main techniques are used
  - **Direct**
  - **Associative**
  - **Set associative**

Direct mapping

- It is simplest technique
- It maps each block of main memory into only one possible cache line.
- ***i = j modulo D***
- ***i = cache line number***
- ***j = main memory block***
- ***D = number of lines in cache***
- Thus if D = 16 then first 16 blocks get mapped to respective line the 17 block gets mapped to line 1 block 18 to 2 and so on
- Main disadvantage is that there is a fixed cache location / line for any given block. Thus is a program references words from two different blocks that maps to the same line, then the blocks will be continually swapped in the cache.



Associative Mapping

- It overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of cache.
- Since a block can be in any one of the cache lines a tag in the cache line is used to determine whether the block is present in the cache.
- With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache
- The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.

Associative Mapping

An address is partitioned to

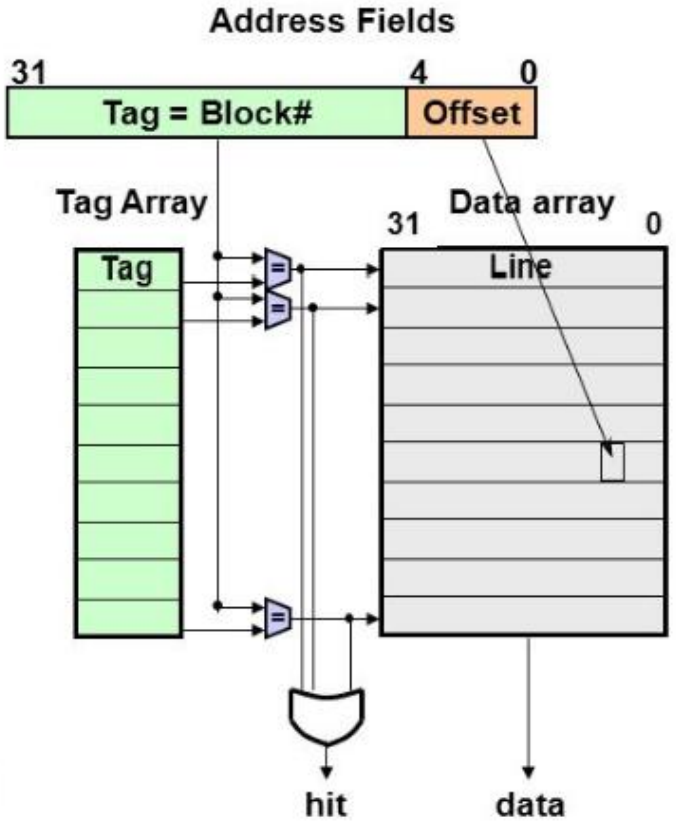
- offset within block
- block number

Each block may be mapped to each of the cache lines

- Lookup block in all lines

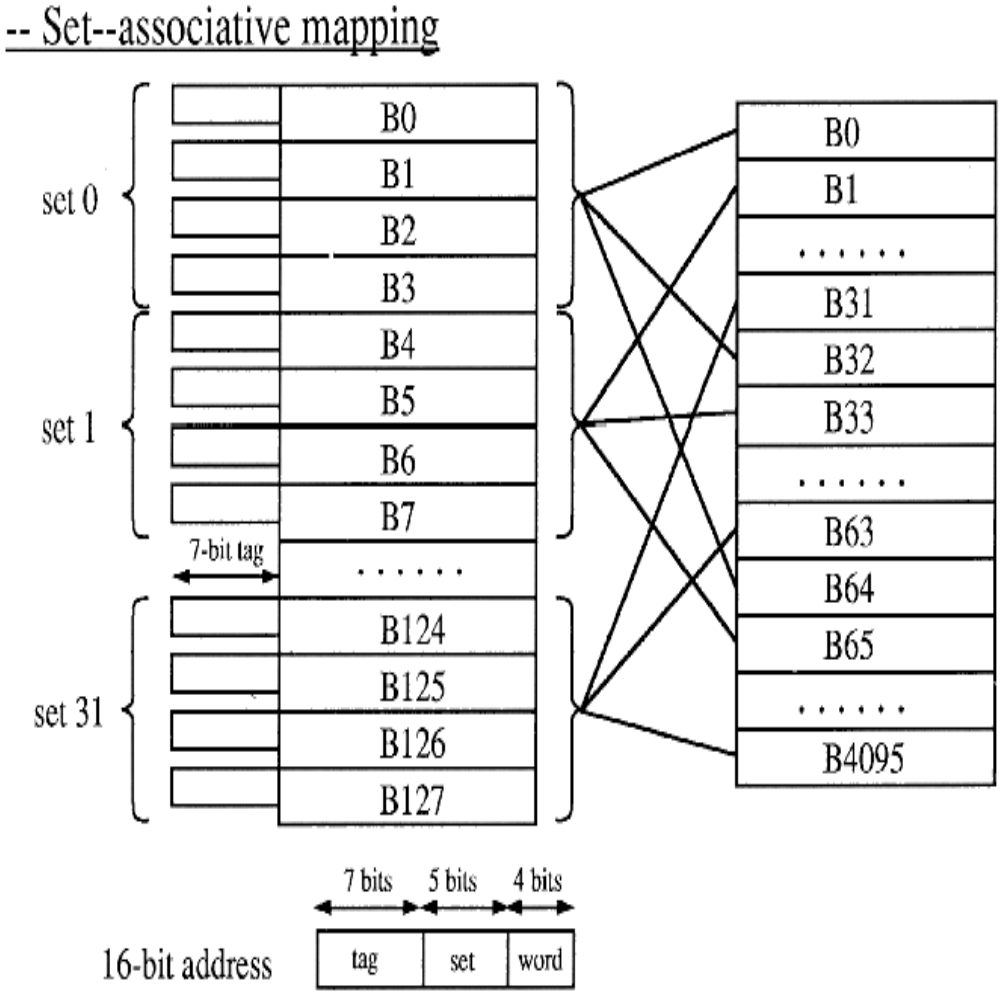
Each cache line has a tag

- All tags are compared to the block# in parallel
- Need a comparator per line
- If one of the tags matches the block#, we have a hit
  - Supply data according to offset



Set associative line

- Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing
their disadvantages.
- The cache consists of a number sets, each of which consists of a number of lines.
- With this a block B_j gets mapped into one of the lines in set j.
- For set associative mapping each word in main memory maps into all the cache line in a specific set.



Replacement Algorithms

- Once a cache has been filled, when a new block is brought into cache one of the existing block must be replaced.
- For direct mapping there is only one choice.
- For associative and set associative mapping a replacement algorithm is needed.
- 4 algorithms are the most common
  - LRU least recently used
  - FIFO first in first out
  - LFU least frequently used
  - Random

Replacement used

- **LRU**
  - It is the most common algorithm used.
  - It replaces the block in the cache that has been in the cache longest with no references.
  - It is easy to implement for associative mapping.
  - When a line is referenced, it moves to the front of the list.
  - For replacement the line a the back is used.
- **FIFO**
  - It replaces the block in the cache that has been there for the longest.
  - It is implemented as a round robin technique
- **LFU**
  - It replaces that block in the cache that has experienced the fewest references.
  - It is implemented by using a counter to keep track of the number of references for a particular line.
  - The line with smallest count is replaced with a new block
- **Random**
  - It simply picks a random line from the candidate lines
  - Random replacement provides only slightly inferior performance according to research

Write Policy

- When a system writes data to cache, it must at some point write that data to the main memory
- The timing of this write is controlled by what is known as the write policy
- A variety of write policies, with performance and economic trade-offs, is possible
- There are two problems to contend with
  - First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid
  - if the I/O device has altered main memory, then the cache word is invalid.
- A more complex problem is if there are multiple processors and each processor has its own cache. Then a word altered in one cache is invalid in other cache

Write Policy

- There are two basic approaches to
  - **Write Through**
    - All write operations are made to main memory as well as to the cache, ensuring that the main memory is valid
    - Any other module can monitor traffic to main memory to maintain consistency within its own cache.
    - Its main disadvantage is that it generates large memory traffic and may create a bottle neck.
  - **Write Back**
    - Updates are only made to the cache.
    - When an update occurs a dirty/use bit associated with the line is set
    - When a block is replaced it is written to the main memory if and only if its dirty bit is set.
    - The problem with this is that the main memory is invalid hence access to the main memory by I/O can only be allowed through the cache.
    - This makes complex circuitry and potential bottleneck.

Performace

- An excellent indicator of the effectiveness of a particular implementation of the memory hierarchy is the success rate in accessing information at various levels of the hierarchy.
- The successful access to data in the cache is called a **hit**
- The number of hits as a fraction of all the attempted accesses is called **hit rate**
- The **miss rate** is the number of misses stated as the fraction of attempted accesses
- Performance is affected by the actions that must be taken after a miss.
- The extra time needed to bring the desired information into the cache is called **miss penalty**
- This penalty reflects that the processor is stalled.
- The miss penalty is reduced if efficient mechanisms for transferring data between various sections of the hierarchy are implemented.
- The hit rate depends on the design of cache and on the instructions and data access patterns of the program.

Performance

- Assume that there are 100 access, 10 clock cycles are needed to access main memory , 15 cycles are needed to load a block into cache, Assume 90% hit ratio. With 2 clock cycle to assess cache.
- Time without cache
    - 100 * 10 = 1000 cycle
- Time with cache
    - (90 * 2) + (10 * 15) = 330