AIC NOTES 2

UNIT 2: PIPELINED COMPUTERS

THEME 4.-DYNAMIC AND SPECULATIVE INSTRUCTION SCHEDULING

1.-BASIC CONCEPTS

Hardware increases ILP by reordering instructions at runtime:

- Independent instructions are simultaneously executed in the pipelined unit.
- Dependent instructions are sequentially executed.

Until now, when instruction i stalls, no following instruction j can proceed, even when j is independent from those under execution and the operator required by j is idle.

Hardware must be able to issue instructions following the stalled one, so instruction execution order is dynamically modified, thus avoiding stalled instructions to affect the following ones.

The **advantages** of this are:

- **Simplifies** the design of the compiler.
- **Efficient resolution of dependencies** that are unknown at compile time.
- Enables the **efficient execution of any code**, despite any existing optimization for another pipelined instruction unit, so it has an efficient binary compatibility.

The **drawback** is that **hardware** becomes **more complex**.

2.-DYNAMIC INSTRUCTION SCHEDULING

The **goals of dynamic instruction scheduling** are:

- Preventing instruction stalls at ID stage.
- Simultaneous execution of independent instructions.
- Correct detection and management of dependent instructions.
- Allow out-of-order execution.

Modifications on the pipelined instruction unit:

- **Issue stage**: when the ID stage decodes a multicycle instruction, the instruction is sent to the Issue stage.
    - If the operator is available and all the instruction operands are ready, the instruction is issued.
    - If the operator required by the instruction is not available, the instruction stalls.
    - If any operand is not available (due to a data dependency with another instruction), the instruction is stalled.

- For the sake of effectiveness, a **load/store operator** is incorporated to the design:
  - Cache access time may be longer than one cycle.
  - Cache misses only affect this operator, without stalling all the following instructions. In addition, when a cache miss occurs, it has a longer access time.
  - Dynamic instruction scheduling also applies to load and store instructions.
  - Detection of dependencies in memory access instructions.

Dependent instructions are stalled at:

- The **Issue stage**, so this stops the instruction decoding.
- The **corresponding operator**.
- A data structure associated with the corresponding operator, that is, the **reservation station**. A physical operator plus the reservation stations offers several virtual operators.

When the dependency disappears, the instructions are resumed following the next steps:

1. At the Issue stage:
   a. The instruction is copied to a virtual operator, configuring the path to be followed by the instruction result, ”marking” the corresponding destination register.
   b. If any of the operands is not available, the path to be followed by those operands from their current location to the corresponding virtual operator is configured.
2. When all the operands for an issued instruction have reached the corresponding virtual operator and the physical operator is free, the instruction is executed.
3. When the instruction ends its execution (Writeback stage), its result is forwarded to the configured instruction destination according to the path defined in 1.

3.-DEPENDENCY GRAPH

Some **considerations**:

- Dynamic instruction scheduling requires keeping, during the execution of instructions, a representation of unsolved data dependencies.
- These dependencies relate virtual operations among them and with register files.
- Each time an instruction is issued (Issue stage), new dependencies are added to the graph.
- Each time an instruction traverses the Writeback stage, it broadcasts its result, thus solving some dependencies, which are then deleted from the graph.

Structural hazards are solved by means of **virtual operators**. The execution only starts if the required physical operator is free.

RAW are solved by **chaining those operands operators involved in the hazard**.

WAW are solved by making that the **last issued instruction is the only one that effectively writes** in the register involved in the hazard.

WAR are avoided by **building the graph during the Issue stage**.

4.-SPECULATIVE INSTRUCTION EXECUTION

The problem with branch prediction techniques is that: the branch can be predicted, but **the branch condition usually takes very long to be computed**, so by the time the condition and the effective branch address are computed, instructions that follow the branch and have started their execution may have already completed execution. If so, **they cannot be cancelled**.

Speculation is a technique that enables partial end even **complete** execution of instructions following a branch, before computing the branch condition. It takes into account that branch conditions may have been misspredicted, thus requiring execution rollback:

- Does not wait for the completion of branch instructions.
- Bets on a given branch behavior.
- Executes instructions before knowing whether they must be executed or not, so they have a "speculative" execution.
- If the prediction turns out to be incorrect, performed actions should not have any effect on the result of program execution.
- But if the prediction is confirmed, performed actions should be committed.

Speculative instructions should not alter program execution, so the result of program execution with and without speculation must be the same:

- Data dependencies among instructions must be respected.
- Speculative instructions should not modify neither registers nor "alive" memory locations.
- Speculative instructions should not generate new exceptions until the branch prediction is confirmed, that is, until speculative instructions become "regular ones".

5.-HARDWARE-BASED SPECULATION

At runtime, branch conditions (taken/not taken) are predicted, and subsequent instructions are "provisionally" fetched and executed, but the state of the machine remains unchanged (no writing on registers or memory locations, no exception generation) **until the branch condition is confirmed**.

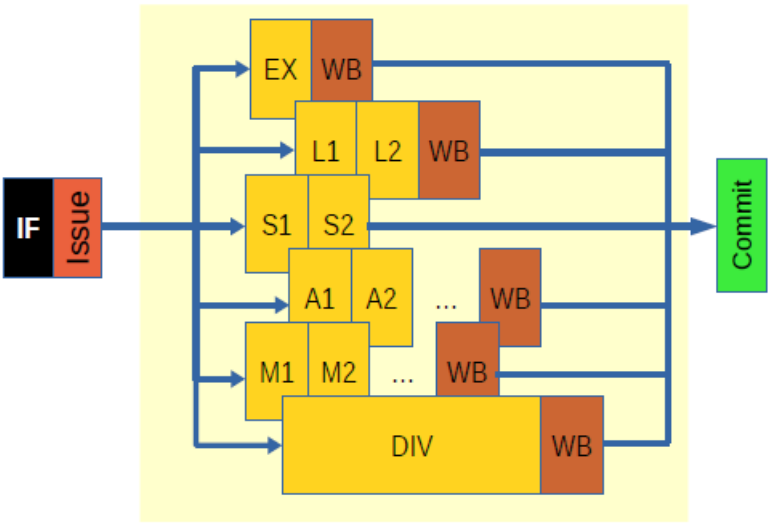
Implementation:

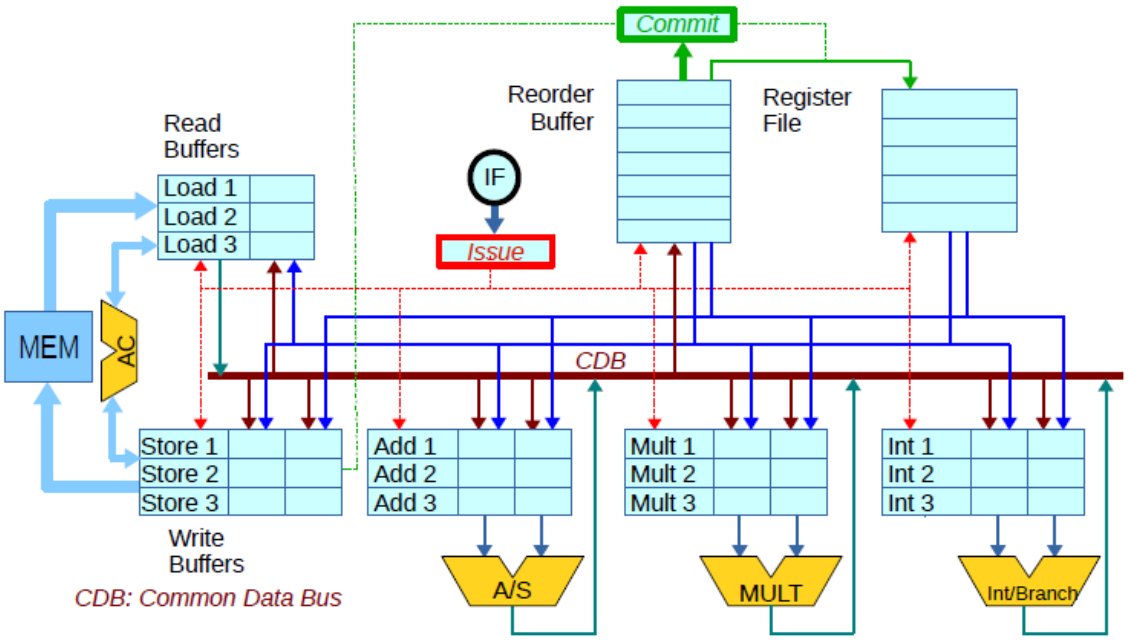
1. Dynamic branch prediction for selection of instructions to execute.
2. In-order instruction fetch (IF).
3. In-order instruction decode and issue (ID + Issue -> I).
4. Out-of-order instruction execution (EX).
5. **In-order** instruction **completion**.

An additional stage is required for instruction completion, that is, the **Commit stage**:

- In-order instruction arrival at this stage.
- Register/memory updates and exception management performed at this stage.
- When an instruction reaches the Commit stage, it is the oldest one in the processor pipeline:
  - All previous branches have been resolved and no previous instruction has provoked any exception.
  - No subsequent instruction has modified the state of the machine, and thus, they can be cancelled in case of need.
- Only those instructions that have been correctly speculated can reach the Commit stage.

Datapath:





Components of the datapath:

- **General purpose registers**.
- **Operators**: each operator has an associated data structure with several entries ("reservation stations"). Each entry contains either a stalled or a running instruction. Thus, each operator implements several "virtual operators".
- **Read and Write Buffers**: they contain data provided by / written to memory. They are managed by load/store units.
- **Common data bus**: fed by all units capable of generating results. It interconnects all the elements able to read data. Access to this bus is arbitrated when several units try to transfer data at the same time.
- **Reorder Buffer**.

Transfers through the common **data bus**:

- **Elements writing results on the bus broadcast its value and a "code" identifying the instruction producing that value**:
  - Virtual operators.
  - Read buffers.
- **Elements reading data from the bus have a variable "mark" that identifies the instruction they are waiting for**:
  - Reservation stations
  - Write buffers
- **Broadcasting a result**: when an element places a data on the bus, it also writes its code. All the elements whose mark matches such code, read the data.

The **Reorder buffer or ROB**:

- When an instruction is issued, a ROB entry is allocated for it. The entry number is used as a mark to tag the required destinations.
- When an instruction reaches the WB stage, it broadcasts its result to the reservation stations, but it writes its result into its ROB entry, not in the destination register. Then, if an instruction has a data dependency with a following one, the latter instruction will obtain its operands from the ROB, and not from registers.
- If an instruction provokes an exception, the event is annotated in its corresponding ROB entry.
- When the oldest instruction in the reorder buffer reaches the Commit stage:
  - It is checked whether it has raised an exception. If this is the case, the corresponding handler is executed.
  - The instruction result is copied from its reorder buffer entry to the corresponding destination register or memory location.
  - The reorder buffer entry is released.
- If a branch is incorrectly predicted, when it reaches the Commit stage it flushes the reorder buffer. Then, speculative instructions that have been incorrectly fetched after the branch:
  - Do not write into their destination register (they do not confirm their execution).
  - Do not originate any exception.

Tomasulo's algorithm with speculation, data structures:

- **Registers (Regs)**:
  - Value (value).
  - Reorder buffer entry (rob).
- **Operators**: floating point, integer and branches, with several reservation stations (RS), each one including:
  - Busy bit (busy).
  - Operation to be executed (op).
  - Operand 1 - Value (Vj).
  - Operand 1 - Mark (Qj).
  - Operand 2 - Value (Vk).
  - Operand 2 - Mark (Qk).
  - Reorder buffer entry (rob).
  - Result (result).
- **Read buffers (RB)**:
  - Busy bit (busy).
  - Memory address (with displacement addressing mode):
    - Base register - Value (Vj).
    - Base register - Mark (Qj).
    - Displacement (disp).
    - Computed address (addr) and its valid bit (valid).
  - Reorder buffer entry (rob).
  - Result (result).
- **Write buffers (WB)**:
  - Busy bit (busy).
  - Operand - Value (Vk).
  - Operand - Mark (Qk).
  - Memory address (with displacement addressing mode):
    - Base register - Value (Vj).
    - Base register - Mark (Qj).
    - Displacement (disp).
    - Computed address (addr) and its valid bit (valid).
  - Confirmation bit (conf).
- **Reorder buffer (ROB)**:
  - Busy bit (busy).
  - Instruction PC (PC)..
  - Instruction (instr): Branch (no result), store (result goes to memory) or ALU/load (result goes to a register).
  - Destination (dest): Register number, write buffer entry (store) or target address (branches).
  - Value (value): Value to store or condition (cond) (branches).
  - Exception identifier, when required.
  - State (state): Indicates whether the instruction has reached the
  - WB stage.
  - Prediction (pred): Predicted condition (branches).

Some **comments**:

- The ROB provides a space to store the instruction results, so it **dynamically renames registers**.
- The ROB entry number allows chaining results between instructions with unsolved data dependencies.
- Reservation stations store information for instructions since they are issued (I) until they complete execution (WB).
- Reservation stations also **monitor** the common data bus looking for operands required by on-hold instructions.
- During the WB stage, reservation stations directly write their result or the generated exception information into the corresponding ROB entry.
- The ROB does not monitor the common bus (for a large ROB size, this monitoring would require too many comparators).
- During the Commit stage, the result in the ROB entry is copied to the destination register, even when any subsequent instruction has blocked the register (since it may end up not being executed). However, in that case the register is not released (since the ROB entry indicated in the register rob field is used to correctly chain dependent instructions).

THEME 5.-MULTIPLE INSTRUCTION ISSUE

1.-INTRODUCTION

To **reduce the time of execution**, we can:

- **Superscalar processors**: **reduce** the **average number of clock cycles per instruction** CPI. Increases the number of issued instructions per clock cycle.
- **VLIW processors**: **reduce** the **number of executed instructions**. Increase the work performed by each instruction.
- **Superpipelined processors**: **reduce** the **clock cycle** T. Pipeline the instruction cycle in more stages (longer pipeline).

The **ILP**, Instruction Level Parallelism, exploited by the machine is **limited** by:

- The **ILP of the program**.
- The **machine hardware constraints**.

This **lowers** the **utilization** of the functional units and **reduces** the potential **performance**.

The alternative is to **support** the **execution of several threads**, that is, to exploit the **TLP**, Thread Level Parallelism. This can be done by means of:

Replicating machine resources: multicore processors, multiprocessors.

Sharing machine resources: multithreaded processors.

2.-SUPERSCALAR PROCESSORS

In superscalar processors, **m instructions can be issued at each clock cycle**:

$$CPI = \frac{1}{m}$$

The **m** term is the **number of ways in the superscalar computer**. This change **increases** the **performance** by a **factor of m**.

The **hardware requirements** are:

- **Simultaneous access** to several **instructions** (IF) and simultaneous access to **multiple data** in memory. To support this, the cache memory must supply m words per clock cycle, so it has to be a **wider** cache or a cache built out of **multiple modules**.
- **Multiple instructions decoding** (I). We need a m instructions decoder. It checks dependencies among the m fetched instructions, and among these and in-flight instructions. It is more complex/faster decoder. It provides multiple decoding cycles.
- **Simultaneous read access to several operands** (I). Thus, we need multiple read ports in the register file and multiple read ports in the ROB.
- **Simultaneous execution of several instructions** (EX). Different functional units working in parallel are required.
- **Several instructions at WB stage**. We need multiple common data buses and multiple write ports in the ROB.

- **Commit several instructions at the same time** (C). Several instructions should be able to Commit simultaneously and we require multiple write ports in the register file.

The **consequences** of all these changes are:

- **Hazard likelihood increases**. In addition to hazards among in-flight instructions, hazards may also occur among the m instructions issued at the same cycle.
- **Increases the penalty** associated to hazards. A single stall prevents the issue of m instructions.
- **Additional problem with branches**:
  - The **branch target address** may be **misaligned** in the cache, thus the IF stage does not provide m valid instructions after the branch.
  - The **branch instruction** may **not be the last one** in the group of m instructions, so the rest of instructions must be discarded.

The instruction set of the original pipelined processor is not modified, so superscalar processors provide **binary compatibility** with the original pipelined processor.

Since there is a problem with the hardware cost, constraints are imposed to the type of instructions that can be simultaneously executed, that is, a **non-uniform superscalar processor**.

The **consequences** of this are:

- Appearance of structural hazards
- The compiler can help by grouping structural hazard-free instructions
- Binary compatibility may not be so efficient.

3.-MULTITHREADED PROCESSORS

Multithreading provides support for multiple flows of instructions aiming at:

- **Increasing the throughput of processors running lots of independent programs**, that is that each thread is a different application.
- **Reducing the execution time of programs having multiple independent threads**, that is that each thread is a fragment of a parallel program.

TLP can be much more cost efficient than ILP. There are two types of TLP architectures: **multiprocessors** and **multithreaded processors**.

Multithreading **allows avoiding most stalls** by **switching to another thread**. Thus, the **instruction throughput increases** and if **both threads** belong to the **same application**, the **execution time is reduced**.

The **hardware requirements** for multithreaded processors:

- Multithreading: most processor components are shared by a set of threads.
- Per-thread state is replicated (private components):
  - Register file.
  - Program counter.
  - Page table.

- The memory is shared through the virtual memory mechanisms.
- Thread switching is performed by processor mechanisms, which is much faster than OS context switching.
- The application must contain several threads, either identified by the compiler or by the programmer.

There are three **types of multithreading approaches**:

- **Fine-grain multithreading**.
- **Coarse-grain multithreading**.
- **Simultaneous multithreading**.

3.1.-FINE-GRAIN MULTITHREADING

In **fine-grain multithreading**, **thread switching is performed every clock cycle**, interleaving the execution of the threads. Interleaving is often done in a **round-robin** fashion, skipping threads that have no ready instructions at that time. It requires very fast thread switching.

The **main advantage** is that it hides the throughput losses that arise from both short and long latencies.

The **main disadvantage** is that it slows down the execution of individual threads, since thread switching is done on a per-cycle basis.

3.2.-COARSE-GRAIN MULTITHREADING

In **coarse-grain multithreading**, **thread switching** is only done on **long-latency stalls**, such as L2 or L3 cache misses.

The **advantages** of this are:

- It does not require fast thread switching. Usually, the instruction unit is flushed, and instructions are fetched from the new thread.
- It does not delay the execution of a single thread, since thread switching only occurs when a thread cannot make forward progress.

The main **disadvantage** is the limited ability to overcome throughput losses, specially from short stalls.

3.3.-SIMULTANEOUS MULTITHREADING

In **simultaneous multithreading**, instructions from different threads can be issued at the same time, trying to use all the available resources.

Out-of-order superscalar processors already **implement mechanisms that support the execution of more than one thread**:

- Register renaming provides a way to uniquely identify all the instruction operands, thus allowing instructions from different threads to be executed without mistaking the operands.
- Dynamic instruction scheduling takes care of dependencies among instructions within each thread.
- Out-of-order execution helps improving resource utilization.
- A private (physical or virtual) ROB for each thread is required.

4.-VLIW PROCESSORS

VLIW (Very Long Instruction Word) processors are processors with a very long instruction format, coding several (p) operations in an instruction. Static instruction scheduling (compiler-based technique) is applied. In these processors, **performances is improved by a factor of p**.

The **consequences** of these changes are:

- The hardware does **not apply dynamic instruction scheduling**.
- The compiler extracts the instruction level parallelism, packing several independent operations (or NOPs, when they are not found) in a single instruction, that is, **special compilation techniques**.
- They are **not binary compatible** with the original scalar machines, neither with other VLIW machines with different hardware.
- If the compiler optimization is not good or the code presents low ILP, the generated code size can be larger than the conventional one.

5.-SUPERPIPELINED PROCESSORS

In **superpipelined processors**, the **clock cycle is t times smaller** than the one of the original pipelined computer. The number of stages in the instruction cycle is kt, but the cycle is shorter. The **performance is increased by a factor of t**.

The **term t** is the **superpipelined degree**: number of substages each original stage is divided into.

The **consequences** of these changes are:

- There is no need for replicating execution units, but pipelining becomes more “complex”.
- Higher clock frequency, so higher overhead on intermediate registers and clock skew problems.
- Indeed, not all the stages of the original pipelined processor must be superpipelined but only the slowest ones.

UNIT 3: MEMORY SUBSYSTEM

THEME 1.-PERFORMANCE OF THE MEMORY SUBSYSTEM

1.-MEMORY HIERARCHY: A QUICK REVIEW

In **memory hierarchy**, memory is organized into several levels, where each level is smaller, faster, and more expensive than the lower level.

This hierarchy achieves **good performance** due to the **principle of locality**: programs tend to reuse code and data.

- **Temporal locality**: recently accessed data items will be accessed again soon.
- **Spatial locality**: items with nearby addresses tend to be accessed shortly after, thus, we use block organization.

It is an **efficient solution** since technologies with different characteristics are used:

- Fast but expensive for performance. Used only for small caches.
- Dense but cheaper (cost per bit) for capacity.

The requirements vary according to the computer type.

The **cache** is the first level of memory hierarchy.

- **Hit**: processor finds the requested data in cache.
- **Miss**: processor doesn't find the requested data in cache. The block that contains the requested word is fetched from main memory to cache.

Cache management (hits, misses, and replacements) is performed by hardware.

If the system supports **virtual memory**, the objects referenced by a program can be located either in main memory or disk. The addressing space is divided into pages that must be in memory to enable access to them. On a page miss, the entire page is transferred from disk to main memory. Page misses are handled by software and do not stall the processor. The processor switches context, running another task while the disk is accessed.

2.-CACHE STRUCTURE AND OPERATION: A QUICK REVIEW

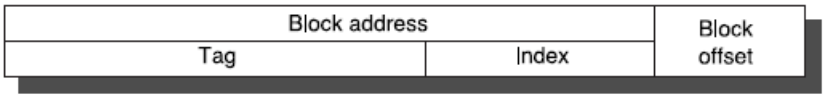
The **cache memory characteristics** are:

- **Block placement**.
- **Block identification**.
- **Replacement policy**.
- **Write policy**.

The **block placement** can be:

- **Direct-mapped**: a block can only be stored in a certain cache line.
- **Fully-associative**: a block can be stored in any cache line.
- **n-way set-associative**: a block can only be stored in a certain set (consisting of n lines).

To **identify the block**, each block stored in the cache has an **associated tag**. To know if a block is in cache, the tag field of the block address is compared with the tags in the target set. The parts of an address issued by the processor are:



The **valid bit (V)** indicates whether a line contains valid information. Only tags having the valid bit set are compared.

To **compare the tags**, we make this in **parallel** with **all the tags in the set**, thus, we need only a single comparison for direct mapping. It is **not necessary to include the block offset** in the comparison because the block is present or not as a whole.

For a given cache size, increasing the associativity increases the number of blocks per set and decreases the number of sets. So, the size of the index and tag decreases and increases, respectively.

On a cache miss, the referenced block is fetched from main memory. If the target set is full, the **block that is removed** depends on the **different strategies**:

- Trivial in direct-mapped caches (just a single candidate).
- (Set-)associative caches:
    - **LRU**: Less Recently Used. Exploits temporal locality.
    - **Pseudo LRU**. Less costly than LRU, similar performance for a high number of ways.
    - **Random**: the candidate is randomly selected. Easy to implement. Useful for structures with poor locality.

Only one data item of the block is modified when **writing**. Then the block must also be updated in main memory (MM). There are **different strategies in case of hit**:

- **Write-through**: data is written in both cache and MM. It is easier to implement, and main memory is always up to date.
- **Write-back**: data is written just in cache. The "dirty" blocks (dirty bit set) are updated in MM when they are replaced. It consumes less memory bandwidth than Write-through.

The are also **different strategies in case of miss**:

- **Write allocate**: the block is brought to cache. Then, it is written by proceeding like in a cache hit. It is usually combined with write-back.
- **No-write allocate**: the block is not brought to cache. It is just modified in the lower level where it is currently stored. It is usually combined with write-through.

Useful formulas:

- Block address = n bits + m bits offset.
- 2^m = number of lines.
- 2^i = cache size / (block size * num. ways).
- i bits index.
- m – i = tag.

3.-PERFORMANCE EVALUATION OF THE MEMORY SUSBSYSTEM

The **average access time**:

$$T_{access} = Ht + MR * MP$$

Where:

- Ht: cache hit time.
- MR: miss rate.
- MP: miss penalty.

The **execution time equation** is **extended** to account for the cache latency:

$$T_{ex} = T_{ex\ cpu} + T_{extra\ memory}$$

Where $\mathbf{T_{ex\ cpu}}$ includes the **cache hit time** and $\mathbf{T_{extra\ memory}}$ the time to **handle misses**.

Formulas:

- $T_{ex\ cpu} = I * CPI * T.$
- $T_{extra\ memory} = Memory\ stall\ cycles * T.$
- $Memory\ stall\ cycles = Num.\ misses * Miss\ penalty = M * MP.$
- $Num.misses = Instructions * \frac{Accesses}{Instructions} * Miss\ rate = I * API * MR.$

Replacing all:

$$T_{extra\ memory} = I * API * MR * MP(in\ cycles) * T$$

Considering that read (R) and write (W) accesses have different miss rate and miss penalty:

- RPI; RMR; RMP: average number of accesses per instruction, miss rate, and miss penalty, respectively, for read accesses.
- WPI; WMR; WMP: average number of accesses per instruction, miss rate, and miss penalty, respectively, for write accesses

$$T_{extra\ memory} = (I * RPI * RMR * RMP * T) + (I * WPI * WMR * WMP * T)$$

Where:

$$RMR = \frac{RM}{I + loads}\ and\ WMR = \frac{WM}{stores}$$

If the penalties are identical, we can compute the unified miss rate (MR'). It can be calculated as:

$$MR' = \frac{RPI * RMR}{API} + \frac{WPI * WMR}{API}$$

On the other hand, if we consider that we have separate caches for instructions (I) and data (D), with different miss rates and miss penalties:

- IPI; IMR; IMP: average number of accesses per instruction, miss rate, and miss penalty, respectively, for instruction cache accesses.
- DPI; DMR; DMP: average number of accesses per instruction, miss rate, and miss penalty, respectively, for data cache accesses.

$$T_{extra\ memory} = (I * IPI * IMR * IMP * T) + (I * DPI * DMR * DMP * T)$$

Where:

$$IMR = \frac{IM}{I}\ and\ DMR = \frac{DM}{loads + stores}$$

Finally, we can consider separate caches (I + D) as well as different miss parameters for read and write accesses by combining the corresponding formulas. For the instruction cache we only need to consider read accesses. Let D_R and D_W represent read and write accesses, respectively, to the data cache:

$$T_{extra\ memory} = (I * IPI * IMR * IMP * T) + (I * D_RPI * D_RMR * D_RMP * T) + (I * D_WPI * D_WMR * D_WMP * T)$$

Where:

$$IMR = \frac{IM}{I}, D_RMR = \frac{D_RM}{loads}\ and\ D_WMR = \frac{D_WM}{stores}$$

THEME 2.-CAHE PERFORMANCE OPTIMIZATIONS

1.-REDUCING THE MISS PENALTY

There are different techniques to reduce the miss penalty:

- **Multilevel caches**.
- **Critical word first** and **Early restart**.
- **Combined write buffers**.

1.1.-MULTILEVEL CACHES

Another cache level (L2) is added between the cache and main memory:

- The L1 cache is designed for speed.
- The L2 cache is large to capture most of main memory accesses.

With this change the **access time** is:

$$T_{access} = Ht_{L1} + MR_{L1} * (Ht_{L2} + MR_{L2} * MP_{L2})$$

With multiple cache levels there are **two types of miss rate**:

- **Local miss rate** for each level:

$$\frac{Num.\,cache\ misses}{Total\ cache\ accesses} = MR_{L1} = MR_{L2}$$

- **Global miss rate**:

$$\frac{Num.\,cache\ misses}{Toal\ accesses} = (Cache\ L1)\ MR_{L1} = (Cache\ L2)\ MR_{L1} * MR_{L2}$$

From the miss rate point of view, the global miss rate of a two-level cache system is similar to the one for a single-level cache as large as the L2 but with a smaller L1 cache, so it is faster than the L2 alone.

There are some **design issues**:

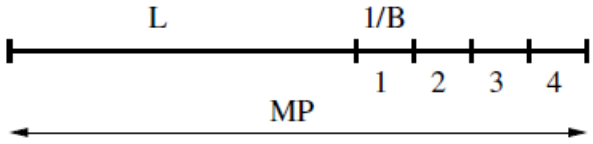
- The L1 cache speed affects the processor clock cycle. The solution is a small L1 cache and direct-mapped or set-associative with few ways.
- The L2 cache affects L1 miss penalty. The solutions is a large L2 cache, much larger than L1, and with many more ways.

There are **two different multilevel caches**:

- **Multilevel inclusion**: all the data in L1 cache are always in L2 cache. It helps to maintain coherence, thus, it is only necessary to check the lower level (L2). In some processors the L2 block size is larger than in L1, or the L3 with respect to the L2. In case of replacement at the lower level, for example, L2, all the blocks in the upper level (L1) that compose the L2 block must be invalidated, increasing MR_{L1}.
- **Multilevel exclusion**: there are not any replicated blocks in caches. They are located either in L1 or L2. It has a better space management. An L1 miss and L2 hit exchanges blocks between caches. It is interesting when the L2 cache is only slightly larger than the L1 cache. The coherence mechanism implementation is complex.

1.2.-CRITICAL WORD FIRST AND EARLY RESTART

The **miss penalty** MP **depends** on the memory **latency** L and **bandwidth** B:



$$MP = L + \frac{1}{B} \cdot n, \text{ where } n \text{ is the block size.}$$

The **latency** L or access time is the time elapsed since the address is sent to the memory until the first data is available at the memory output.

The **bandwidth** B is the speed at which data can be supplied.

The processor only needs the word of the block that caused the miss, so two solutions arise:

- **Early restart**: as soon as the requested word arrives, it is delivered to the processor to continue execution.
- **Critical word first**: the first accessed word is the one requested by the processor. The processor continues processing while the rest of the words in the block are loaded.

Performance improvement achieved with Critical word first and Early restart is larger when:

- Large cache blocks are used.
- The next memory instruction does not access the same block that is being loaded. In such a case, the second access must wait till block loading finishes.

1.3.-WRITE BUFFERS

With **write buffers**, the processor writes into the buffer without stalling execution, decoupling the execution from the memory write, which is handled by the controller.

With this change, a **problem** of **memory data dependencies** arises. The **solutions** are:

- Wait until the write buffer is emptied before reading data.
- Check if the requested address is in the write buffer and, if not, let the read continue (**load-bypassing**).
- If the requested address is in the write buffer, read the data from the buffer (**load-forwarding**).

If the **buffer** is **full** and an entry is needed, we suffer a **stall**. The **solution** to this is using **combined write buffers**. Each buffer entry refers to a set of consecutive addresses. The address is compared with those of valid buffer entries. If it matches, the data are combined with that entry. We have an efficient memory bandwidth usage since we write several words at a time instead of only one.

2.-REDUCING THE MISS RATE

Block miss classification:

- **Compulsory**: appears the first time a block is accessed. They usually represent a low percentage of the total misses.
- **Conflict**: appears when the target set is full but there is free space in other sets. Fully-associative caches do not have conflict misses, but require more hardware and may impact the clock frequency.
- **Capacity**: if the cache cannot store the entire working set for an application, active blocks are frequently replaced, thus producing capacity misses. These misses are reduced when cache size increases.

The techniques to reduce the miss rate are intended to reduce the global miss rate:

- **Adjusting the cache geometry**:
  - Block size.
  - Cache size.
  - Number of ways.
- **Way-prediction**.
- **Compiler optimizations**.

2.1.-ADJUSTING THE CACHE GEOMETRY

A larger **block size** leads to:

- Higher exploitation of spatial locality, so compulsory misses are reduced (Decreases MR)
- Less cache lines for a given cache size.
  - Conflict misses may rise (Increases MR).
  - Capacity misses may rise (Increases MR).
- Increases the miss penalty (MP) due to having to bring more words.

The block size must be a **trade-off** between **MR** and **MP**. Must be relatively large to amortize MP, but the miss rate must be reasonable. Some processors use a larger size in the Last-Level Cache (LLC).

A larger **cache size** leads to:

- Reduces capacity misses (Reduces MR).
- Increases the hit time (Ht).
- Increases area and power consumption.
- Idea: design L1 small with fast technology for performance, and L2 large for capacity, and use a power-aware technology for huge caches.

A higher **number of ways** leads to:

- Reduces conflict misses (Reduces MR).
- Requires more comparators (more energy consumption).
- The multiplexer has more entries, so it may increase the cache access time (Ht).

2.2.-WAY PREDICTION

The **objectives** of way prediction are:

- Reduce conflict misses without increasing Ht with respect to a direct-mapped.
- Reduce energy with respect to a set-associative cache.

The main idea is that a predictor predicts the way of the set that has to be accessed. Multiplexer control entries are accordingly set in advance to hide multiplexer latency, while tag and data arrays are being accessed. On misprediction, the remaining tags are compared. The cache is set-associative.

Hit time has **two different values**:

- **Prediction OK**: only one tag is compared, so low Ht (1 cycle).
- **Misprediction**: the remaining tags are compared, the predictor updated and the multiplexer properly set, thus, Ht is much higher (3 cycles).

2.3.-COMPILER OPTIMIZATIONS

The compiler generates an optimized code that reduces the miss rate.

Reduction of instructions miss rate:

- Reordering the groups of instructions so that they are stored in different sets to reduce conflict misses.
- Aligning the entry point of basic blocks with the beginning of a cache block. It improves spatial locality.

Branch straightening is a technique in which if the compiler predicts a branch to be taken, the code is modified to evaluate the opposite condition, and locate next to the branch instruction the code initially located at the target address. It improves spatial locality.

A technique to **reduce data misses** is **loop exchange**: loop nesting is exchanged to operate in the order in which data are stored in memory.

To **improve the temporal locality**, we use **blocking**: if arrays are accessed both by rows and by columns, it is better to operate on sub-matrices or blocks, trying to maximize the access to cached data before lines are replaced.

3.-MISS PENALTY AND MISS RATE REDUCTION THROUGH PARALLELISM

Techniques that **overlap execution of instructions with memory access**:

- **Non-blocking caches**.
- **Hardware-based prefetching of instructions and data**.
- **Compiler-controlled prefetching**.

3.1.-NON-BLOCKING CACHES

The cache services new requests while a previous miss is being handled. Alternatives:

- "**hit under miss**": the cache can only handle one miss at a time, but in the meantime, it is able to service hits.
- "**miss under miss**" or "hit under multiple misses": multiple misses can be handled at the same time.

3.2.-HARDWARE-BASED PREFETCHING OF INSTRUCTIONS AND DATA

The idea is to **look for information before it is required by the processor**. Pre-fetched information is stored in an external buffer, faster than main memory. Prefetch of instructions:

- **On a miss**, the processor retrieves two blocks: the required one, which is stored in the cache, and the following one, which is stored in the instructions buffer.
- **On a cache miss**, the block in the instructions buffer is read, and in case of hit, the next block is prefetched

Data prefetching is more complex than instruction prefetching. The block to be prefetched is computed by the prefetcher logic. Most current prefetchers detect non-unit regular stride patterns. Prefetch requests compete with cache misses when accessing to main memory, so MP may increase.

In a **compiler-controlled prefetching**, the compiler inserts prefetch instructions in order to request data before the processor requires them. Prefetch instructions must not generate neither virtual memory faults nor protection violation exceptions, thus, it is a **nonbinding fetch**. It makes sense in non-blocking caches and it is especially useful in loops.

The insertion of prefetched instructions increases the number of executed instructions, increasing the overhead. The focus must be placed on those accesses likely to result in block misses.

4.-REDUCING HIT TIME

Reducing hit time is very important since it affects the processor clock period. Techniques that do this are:

- **Simple and small caches**.
- **Avoid virtual memory translations in cache accesses**.
- **Cache pipelining**.

4.1.-SIMPLE AND SMALL CACHES

A large percentage of the time required to access the cache is devoted to compare the tag field of the accessed address with the tags stored in the cache. A possible way to reduce this time:

- **Small cache**: "the smaller the cache structure the faster" and it fits in the same chip the processor does (L1 and L2).
- **Simple cache**: direct mapped caches (this way the multiplexer is not required), or set-associative with few ways.

The **trends** are:

- **L1 caches**: small and simple. Emphasis on speed for performance. Reduced size and low associativity.
- **L2 cache**: much larger to avoid memory accesses. They are designed to be private (accessed by just one core) or shared by a subset of the cores.
- **L3 cache**: some processors include L3 as LLC. These caches are shared by several L2 caches. Designed to avoid accesses to main memory. In particular, they are useful to store shared variables. Low-power technologies are used to implement them.

4.2.-AVOID VIRTUAL MEMORY TRANSLATIONS IN CACH ACCESSES

Another component of the Ht is the time devoted to translating from the virtual memory address issued by the processor to the physical memory address. This translation is performed in the TLB. Once the translation completes, the cache access can be performed.

The idea is to **use the virtual address to access the cache**. Problems with this technique:

- **Protection**: this is part of the translation process of the virtual address to the physical one. The information from the TLB must be copied to the cache.
- **Processes**: each process owns its virtual memory space. When the context is switched, a given virtual address points to a different physical address, thus, the cache must be emptied with each context switch or process identifiers (PID) must be added to cache tags.
- **Aliasing**: a given physical address can be referred to by using two or more different virtual addresses. There may exist several copies of the same data that must be kept consistent (they must be identical).

A part of the page offset is used to index the cache. Cache tags and data are read in parallel to the translation. The imitation is the size of a direct-mapped cache or the number of sets in a set-associative cache cannot exceed the size of a virtual memory page.

4.3.-CACHE PIPELINING

Cache access is pipelined in stages to match the processor clock. A cache access takes several cycles but several instructions can overlap their cache accesses. It impacts the design of the processor pipeline: more stall cycles in branch and load instructions. This technique increases the instruction cache bandwidth rather than reducing its latency.

THEME 3.-MAIN MEMORY PERFORMANCE OPTIMIZATIONS

1.-MEMORY TECHNOLOGY AND PERFORMACE MODEL

Main memory supplies the requests of the cache and I/O subsystem. From the cache point of view, the **goal** is to **reduce miss penalty** (MP). If a **single data is accessed**, the **MP** is the **memory access time**. If **multiple data are accessed**, the MP:

$$MP = L + \frac{1}{B_W} B$$

The **latency**, L, is the time to satisfy the first access. The **bandwidth**, B_W, is the number of words transferred per time unit.

It is easier to increase the bandwidth than to reduce the latency.

1.1.-TRADITIONAL DRAM

In the **traditional DRAM**, due to pin count constraints, the address is multiplexed. First the row address is transmitted (RAS signal is activated to validate the address), then the column address (CAS signal is activated to validate it).

A single memory word is read/written every time memory is accessed. However, an entire row is internally read every time a word is accessed, later refreshing it. After accessing a word, the next access cannot start until the memory cycle is completed, then a precharge refreshes and closes a row.

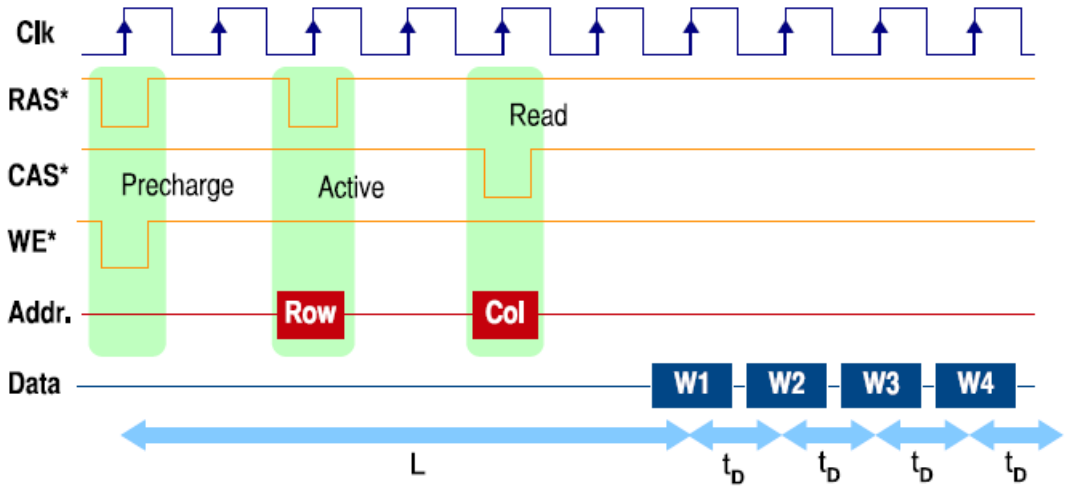
With **fast page mode**, if a row buffer is added, accesses to other words in the same row will be faster. Once the row is available, several column addresses can be read (or written) in sequence. Only the column address is required for each access.

1.2.-SDRAM

The Synchronous DRAM (**SDRAM**) **characteristics** are:

- SDRAMs are synchronous, so the **clock signal** is **sent** to **memory**. The **clock frequency** is defined by the **memory controller**. Timing is measured in **clock cycles**. The number of cycles required for each operation (sending addresses, accessing and transferring data) are read from a ROM in the SDRAM module and are used to configure the memory controller.
- SDRAM chips are **organized** as **one or more banks**. Each bank is a memory cell array. Once a bank row is activated, it is possible to read and/or write any column from it.
- **Burst mode**: SDRAMs use a self-incrementing counter and a mode register to determine the column address sequence after the first access to a row. This allows faster DRAM operations since the time to set up subsequent column addresses is removed.

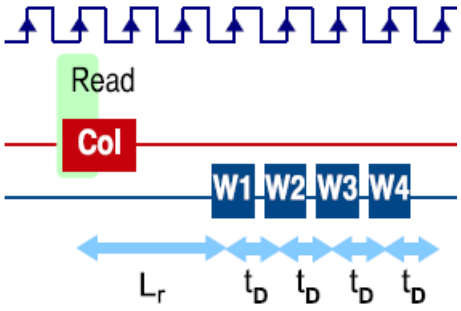
SDRAM read chronogram:



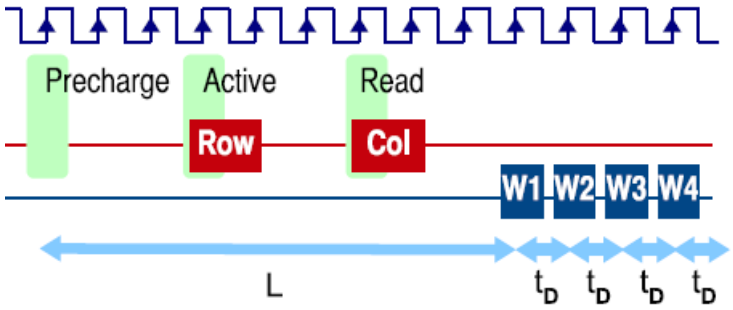
Some **implications** are:

- Cache and disk blocks are accessed in burst mode so that, once the first word is accessed, the remaining ones could be transferred at a high rate.
- A higher clock frequency increases data transfer rate but it does not reduce memory access time L for the first word in a burst.
- Memory access time L depends on memory timing parameters, which are specified as an integer number of clock cycles: the lowest number of cycles that allow the operation to complete. Thus, due to rounding effects, it may happen that a higher clock frequency leads to longer access time.
- As many rows as chip banks can be active at a time.
- Miss penalty depends on whether consecutive block accesses belong to the same row. This introduces variability in the miss penalty, making it dependent on memory access patterns.

If the row is already open:



If the row needs to be opened:



1.3.-SIMPLE MEMORY MODEL

Generic memory parameters:

- **L**: latency or access time (time to read the first word).
- **t_D:** time to transfer each word.
- **B_w**: bus bandwidth, measured in words/sec.

Consider a block size of B memory words. The **miss penalty MP** is:

$$MP(in\ seconds) = L + t_D * B = L + \frac{1}{B_W} * B$$

MP can also be expressed in clock cycles:

- **f**: Bus clock frequency.
- **L_c**: Latency, measured in cycles at frequency f.
- **B_{wc}**: Bus bandwidth, measured in words/cycle at frequency f.

$$MP(in\ cycles) = L_C + \frac{1}{B_{WC}} * B$$

$$MP(in\ seconds) = MP(in\ cycles)/f$$

In the **general case**, the requested row may not be open:

- Let ML (memory locality) be the probability that a cache miss requests a memory block that belongs to one of the open (activated) rows.
- In that case, access time is shortened since the corresponding row is already stored in a row buffer.
- Let L_r be the reduced access time or latency. Let L_{rc} be the value of L_r when measured in cycles at frequency f.

The **average miss penalty** is:

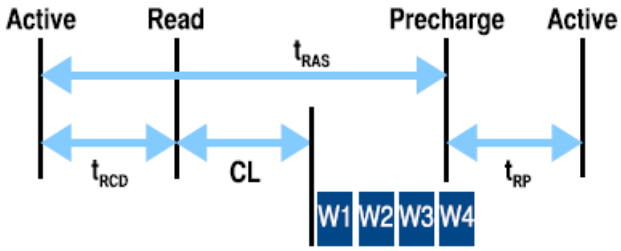
$$MP(in\ seconds) = L * (1 - ML) + L_r * ML + \frac{1}{B_W} * B = MP(in\ cycles)/f$$

$$MP(in\ cycles) = L_C * (1 - ML) + L_{rc} * ML + \frac{1}{B_{WC}} * B$$

1.4.-MEMORY TIMING PARAMETERS

Timing in commercial SDRAMs is defined by the clock frequency and by four timing parameters, separated by dashes. Those **parameters**, in order of appearance, are the following:

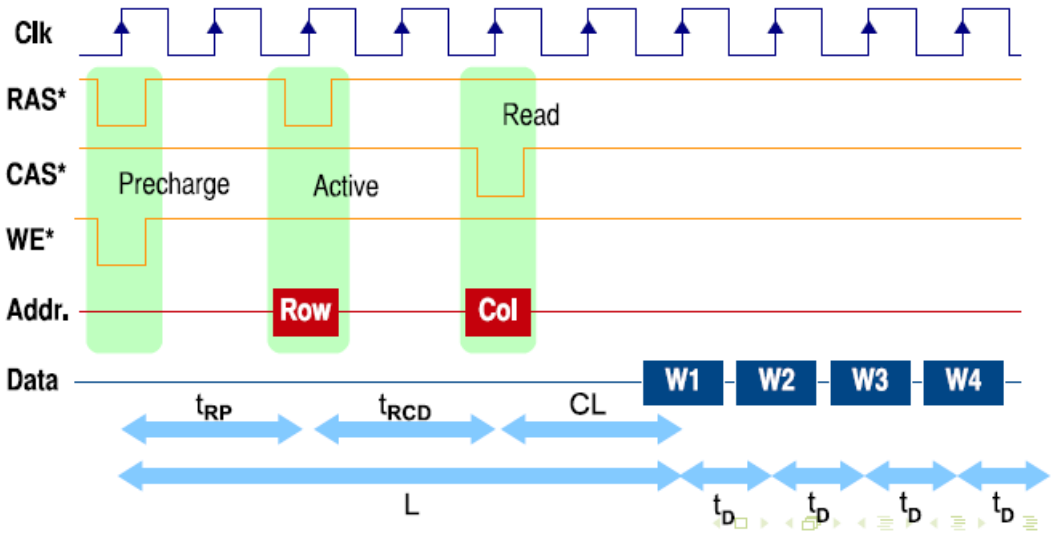
- **CL**: The time (in cycles) between sending a column address to the memory and the beginning of the data burst. This is the time it takes to read the first bit of memory from a DRAM with the correct row already open.
- **t_{RCD}**: The number of clock cycles required between the opening of a row of memory and accessing columns within it. The time to read the first bit of memory from a DRAM without an active row is t_{RCD} + CL.
- **t_{RP}**: The number of clock cycles required between the issuing of the precharge command and opening the next row. The time to read the first bit of memory from a DRAM with the wrong row open is t_{RP} + t_{RCD} + CL.
- **t_{RAS}**: The number of clock cycles required between a bank active command and issuing the precharge command. This is the time needed to internally refresh the row and overlapping with t_{RCD}. This fourth parameter is sometimes dropped when specifying memory timings.



The **computation of L and L_r from the timing parameters** is:

$$L_C = t_{RP} + t_{RCD} + CL\ cycles \text{ and } L_{rc} = CL\ cycles$$

Summary:



2.-ENHANCING SDRAM PERFORMANCE

MP can be **reduced** by **reducing L** and **B**, and by **increasing B_w** and **ML**. However:

- L is by far the largest contributor to MP.
- L remains roughly constant when increasing f, except for rounding effects, Lc increases linearly with f.
- ML mostly depends on memory access patterns, but it also depends on the number of banks.

The **techniques** to **improve the performance of main memory** are:

- Increase the bus width: **B decreases**.
- Increase B_{wc} while keeping f constant by transferring data at both the rising and falling edges of the clock signal: **B_w increases**.
- Increase the clock frequency f while keeping B_{wc} constant: **B_w increases**.
- Increase the number of memory banks: **ML increases**.
- Implement several memory controllers. Although this does not reduce miss penalty (unless addresses are interleaved), it allows several misses to be concurrently serviced. It also increases the total number of memory banks: **ML increases**.

2.1.-INCREASING BUS WIDTH

By making the memory bus wider, more than one word can be transferred at the same time, then the number of transfers is reduced. Since L_c is by far the largest contributor to MP, **little savings can be achieved by making the bus wider**.

Memories are arranged as modules Dual Inline Memory Module (**DIMMs**), each of them having one or more ranks. A **rank** consists of enough chips to complete 64 bits.

2.2.-DDR: DOUBLE DATA RATE

The idea is to **transmit data at both the rising and falling edge of the clock signal**. The bus works at the same speed, but the bandwidth is doubled. The maximum signalling frequency with respect to SDR (Single Data Rate) remains unchanged, thus being implementable without having to enhance technology. Internally, the number of accessed columns is doubled (2n-prefetch) as well as the width of the bus connecting memory banks with the data bus, thus, the internal clock frequency does not change. The result is that **B_{WC} is doubled**.

2.3.-INCREASING THE BUS CLOCK FREQUENCY

Memory bus clock frequency has been **increased over time**, also reducing voltage to reduce power consumption. Several techniques have been developed to keep signal integrity at higher clock frequencies. Clock frequency and voltage have been standardized by JEDEC.

Standard values are as follows:

- **DDR2**. The number of accessed columns is doubled with respect to DDR (4n-prefetch) as well as the width of the bus connecting memory banks with the data bus.
- **DDR3**. The number of accessed columns is doubled with respect to DDR2 (8n-prefetch) as well as the width of the bus connecting memory banks with the data bus.
- **DDR4**. Keeps 8n-prefetch. Banks are arranged into groups that are also addressable. Each group can be independently and concurrently accessed, regardless of the state of other groups. Memory buses have been replaced with channels with point-to-point links.

To report performance, the **notations** are:

- **DDRn-xxxx**, where xxxx indicates the transfer rate in Mtransfers/s. Bus clock frequency: f = xxxx/2 MHz.
- **PCn-yyyy**, where yyyy is the bus bandwidth in MB/s. Bus clock frequency: f = yyyy = (8 * 2) MHz.

2.4.-INCREASING THE NUMBER OF MEMORY BANKS

Current **SDRAM** chips **implement a large number of banks**. The **reasons** are:

- Several rows (one per bank) can be open at a time, thus increasing ML. To access a different open row, the bank address is supplied together with the column address. Accessing a different open row is as fast as accessing the same row again.
- For a given memory capacity, increasing the number of banks reduces the bank size, thus reducing latency.
- Smaller banks also imply faster address decoders.
- A bank design can be replicated, thus simplifying SDRAM chip design.

2.5.-INCREASING THE NUMBER OF MEMORY CONTROLLERS

Current high-performance processors implement multiple memory controllers. Each memory controller implements one or two channels to access one or more ranks of SDRAM DIMMs. Each DIMM implements multiple internal banks. The **total number of rows** that can be **simultaneously open** is the **number of memory controllers times the number of channels per controller times the number of ranks per channel times the number of banks per rank**.

A large number of open rows is necessary to minimize conflicts when multiple cores concurrently initiate memory accesses. This way, many memory accesses will hit on an open row, thus maximizing ML.