Advanced Computer Architectures

Claudio Di Salvo based on Lorenzo Rossi's Note

2022/2023

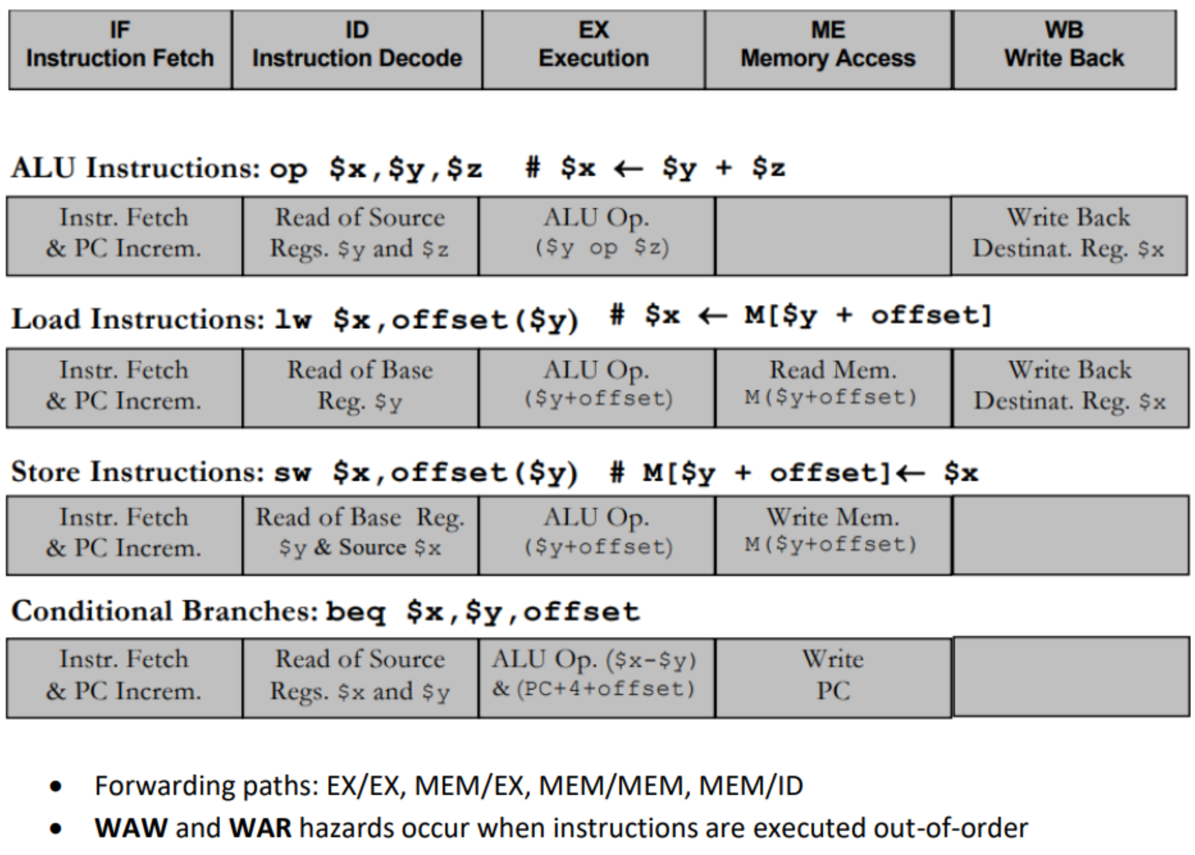
Last update: 2023-06-04

Contents

1 Introduction to Computer Architectures

1.1 MIPS Pipeline

The MIPS Pipeline is a 5 stages micro processor that can be rappresented by this image:



IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back

ALU Instructions: `op $x,$y,$z  # $x ← $y + $z`

Instr. Fetch & PC Increm.	Read of Source Regs. `$y` and `$z`	ALU Op. (`$y op $z`)		Write Back Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)  # $x ← M[$y + offset]`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back Destinat. Reg. `$x`

Store Instructions: `sw $x,offset($y)  # M[$y + offset]← $x`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`	

Conditional Branches: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC	

- Forwarding paths: EX/EX, MEM/EX, MEM/MEM, MEM/ID
- **WAW** and **WAR** hazards occur when instructions are executed out-of-order

Figure 1: This rappresent the 5 stages and the execution of the different type of operation

1.1.1 Optimization of MIPS Pipeline

The most important optimization that an MIPS Processor can have are:

- Forwarding path (Already wrote in 5)
- Register File that can be read and wrote in the same clock cycle.
- Branch prediction (Will be explained later)

1.1.2 Stages in *MIPS* pipeline

The 5 stages in the *MIPS* pipeline are:

1. **Fetch** - `IF`

   → Instruction fetch from memory
2. **Decode** - `ID`

   → Instruction decode and register read
3. **Execute** - `EX`

   → Execute operation or calculate the address

4. **Memory access** - `ME`

   → Access memory operand

5. **Write back** - `WB`

   → Write the result back to register

Each instruction is executed after the previous one has completed its first stage, and so on. When the pipeline is filled, five different activities are running at once. Instructions are passed from one unit to the next through a storage buffer. As each instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along.
The stages are usually represented in Figure 2.

`IF`	`ID`	`EX`	`ME`	`WB`

Figure 2: Stages in *MIPS* pipelines

1.2 Flynn Taxonomy

Created in 1996 and upgraded in 1972, it provides the first description of a computer.

- *SISD* - **single** instruction, **single** data
  - **Sequential** programs
  - **Serial** (non parallel) computer
  - **Deterministic** execution
  - Only **one instruction stream** is being executed at a time
- *MISD* - **multiple** instructions, **single** data
  - Multiple processors working in **parallel** on the same data
  - **Fail safe** due to high redundancy
  - The same algorithm is programmed and implemented in different ways, so if one fails the other is still able to compute the result
  - **No practical market configuration**
- *SIMD* - **single** instruction, **multiple** data
  - Each processor receives **different data** and performs the **same operations** on it
  - Used in fields where a single operation must be performed in many different pieces of information *(like in image processing)*
  - Each instructions is executed in **synchronous** way on the same data
  - Best suited for specialized problems characterized by a high degree of regularity, such as graphics or images processing
  - **Data level parallelism** *(DLP)*
- *MIMD* - **multiple** instructions, **multiple** data
  - **Array of processors** in parallel, each of them executing its instructions
  - Execution can be **asynchronous** or **synchronous**, **deterministic** or **non-deterministic**
  - The **most common** type of parallel computer

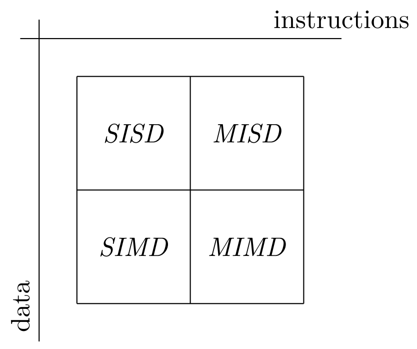


Figure 3: Flynn Taxonomy

1.3 Hardware parallelism

There are different types of hardware parallelisms:

- **Instruction Level parallelism** - *(ILP)*
  - Exploits data level parallelism at modest level through **compiler techniques** such as pipelining and at medium levels using speculation
- **Vector Architectures** and **Graphic Processor Units**
  - Exploit data level parallelism by applying a single instruction to a **collection of data** in parallel
- **Thread level parallelism** - *(TLP)*
  - Exploits either data level parallelism or task level parallelism in a coupled hardware model that allows **interaction among threads**
- **Request level parallelism**
  - Exploits parallelism among largely decoupled tasks specified by the programmer or the OS

Nowadays, heterogeneous systems *(systems that utilize more than one type of parallelism)* are commonly used among all commercial devices.

1.4 Pipeline and Hazard

The pipeline *CPI (clocks per instruction)* can be calculated taking account of:

- **Ideal** pipeline *CPI*
  - measure of the maximum performance attainable by the implementation
- **Structural stalls**
  - due to the inability of the *HW* to support this combination of instructions
  - can be solved with more *HW* resources
- **Data hazards**
  - the current instruction depends on the result of a prior instruction still in the pipeline
  - can be solved with *forwarding* or *compiler scheduling*
- **Control hazards**
  - caused by delay between the `IF` and the decisions about changes in control flow *(branches, jumps, executions)*
  - can be solved with *early evaluation, delayed branch, predictors*

The main features of the pipeline are:

- **Higher throughput** for the entire workload
- Pipeline rate is limited by the **slowest** pipeline stage
- Multiple tasks operate **simultaneously**
- It **exploits parallelism** among instructions
- Time needed to *"fill"* and *"empty"* the pipeline reduces speedup

1.5 Pipeline hazards

An **hazard** is a fault in a pipeline. A hazard is created whenever there is a dependence between instructions that are close enough such that the overlap introduced by pipelining would change the order of access to the operands involved in said dependence. It prevents the next instruction in the pipeline from executing during its designated clock cycle, reducing the performance from the ideal speedup.
There are 3 classes of hazards:

1. **Structural** hazards, due to attempting to use the same resource from different instructions at the same time
2. **Data** hazards *stalls*, due to attempting to use a result before it is ready
   - *Read after write - RAW*
     → the instruction tries to read a source register before it is written by a previous instruction
     → it's also called *dependence* by compilers
     → caused by an actual need for communication
   - *Write after read - WAR*
     → the instruction tries to write a destination register before it is read by a previous instruction
     → it's also called *anti dependence* by compilers
     → caused by the reuse of the same register for different purposes
   - *Write after write - WAW*
     → the instruction tries to write before it is written by a previous instruction
     → it's also called *output dependence* by compilers
     → caused by the reuse of the same register for different purposes
3. **Control** hazards *stalls*, due to the request of deciding on the next instruction to execute before the condition itself is evaluated

Data stalls may occur with instructions such as:

- *RAW* stall:

```
r3 := (r1) op (r2)
r5 := (r3) op (r4)  // r3 has not been written yet
```

- *WAR* stall:

```
r3 := (r1) op (r2)
r1 := (r4) op (r5)  // r1 has not been read yet
```

- *WAW stall*:

```
r3 := (r1) op (r2)
r3 := (r6) op (r7) // r3 has not been written yet
```

1.5.1 Solutions to data hazards

There are many ways in which data hazards can be solved, such as:

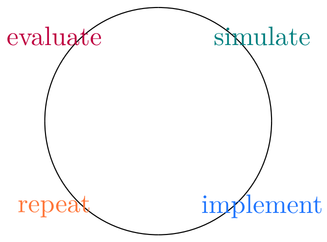
- **Compilation** techniques
  - **Insertion of `nop` instructions**
  - **Instructions scheduling**
    - the compiler tries to avoid that correlating instructions are too close

        - it tries to insert independent instructions among correlated ones
        - when it can't, it inserts `nop` operations
- **Hardware** techniques
    - **Insertion** on *bubbles* or *stalls* in the pipeline
    - Data **forwarding** or bypassing

Both the compilation and the hardware techniques will be analyzed in depth in Section 8.3.

2 Performance and cost

There are multiple types *(classes)* of computers, each with different needs: the performance measurement is not the same for each of them. *Price, computing speed, power consumption* can be metrics to measure the performance of a computer.
Programming has become so complicated that it's not possible to balance all the constraints manually; while the computational power has grown bigger than ever before, energy consumption is now a sensible constraint.
The computer engineering methodology is therefore described as:



1. *evaluate* system by bottlenecks
2. *simulate* new designs and organizations
3. *implement* next generation systems
4. *repeat*

Figure 4: Computer engineering methodology

The picked qualities may change according to the use case or the user themself, but 2 metrics are normally used:

1. **Elapsed time**
   - `execution time` $=$ `time end` $-$ `time start`
   - More relevant for system user
2. **Completion rate**
   - `completion rate` $=$ `number of jobs` $\div$ `elapsed time`
   - More relevant for the system designer

2.1 Response time vs throughput

Is it true that `throughput` $= 1 \div$ `average response time`? The answer can be given only if it's clear if there's an overlap between core operations.
If there is, then

$$\texttt{throughput} > 1 \div \texttt{average response time}$$

With pipelining, **execution time** of a single instruction is **increased** while the **average throughput** is **decreased**.

2.2 Factors affecting performance

A few of the factors affecting the performance are:

- Algorithm complexity and data sets
- Compiler
- Instructions set
- Available operations
- Operating systems
- Clock rate
- Memory system performance
- I/O system performance and overhead
  - this it's the least optimizable factor, the main focus is then to optimize all the others

The locution X **is** n **times faster than** Y can be expressed as:

$$\frac{ExTime\ (Y)}{ExTime\ (X)} = \frac{Performance\ (X)}{Performance\ (Y)} = Speedup\ (X, Y)$$
$$Performance(X) = \frac{1}{ExTime\ (x)}$$

So, in order to optimize a system, it's necessary to focus on common sense (a *sadly* valuable quality). While making a design trade-off one must favour the frequent case over the infrequent one.
For example:

- Instructions fetch and decode unit is used more frequently than the multiplier
  - it makes sense to optimize **instructions fetch unit** first
- If database server has 50 disks processor, storage dependability is more important than system dependability
  - it makes sense to optimize **storage dependability** first

2.2.1 Amdahl's law

As seen before, the speedup due to the enhancement E is:

$$Speedup\ (E) = \frac{ExTime\ w/o\ E}{ExTime\ w/\ E} = \frac{Performance\ w/\ E}{Performance\ w/o\ E}$$

Suppose that enhancement E accelerates a fraction F of the task by a factor S and the remainder of the task is unaffected. Amdahl's law states that:

$$ExTime_{new} = ExTime_{old} \times \left[(1 - F) + \frac{F}{S}\right]$$
$$Speedup = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1 - F) + {}^{F}/_{S}} = \frac{S}{S - SF + F}$$

2.2.2 *CPU* time

CPU time is determined by:

- **Instruction Count** - *IC*:
  - The number of executed instructions, not the size of static code
  - Determined by multiple *factors, including algorithm, compiler, ISA*
- **Cycles per instructions** - *CPI*:
  - Determined by *ISA* and *CPU* organization
  - Overlap among instructions reduces this term
  - The *CPI* relative to a process `P` is calculated as:

$$CPI(P) = \frac{\texttt{\# of clock cycles to execute P}}{\texttt{number of instructions}}$$

- **Time per cycle** - *TC*:
  - It's determined by technology, organization and circuit design

Then, *CPU* time can be calculated as:

$$CPU_{time} = T_{clock} \cdot CPI \cdot N_{inst} = \frac{CPI \cdot N_{inst}}{f}$$

Note that the *CPI* can vary among instructions because each step of the pipeline might take different amounts of time. The factors that can influence the *CPU* time are shown in Table 1.

	IC	*CPI*	*TC*
Program	✖		
Compiler	✖	(✖)	
Instruction set	✖	✖	
Organization		✖	✖
Technology			✖

Table 1: Relation between factors and *CPU* time

2.2.3 Other metrics

There are other metrics to measure the performance of a *CPU*:

- *MIPS* - millions of instructions per second

$$MIPS = \frac{\texttt{number of instructions}}{\texttt{execution time} \cdot 10^6} = \frac{\texttt{clock frequency}}{\texttt{CPI} \cdot 10^6}$$

  - the higher the *MIPS*, the faster the machine
- *Execution time*

$$T_{execution} = \frac{\texttt{instruction count}}{\texttt{MIPS} \cdot 10^6}$$

- *MFLOPS* - floating point operations in program
  - assumes that floating points operations are independent of the compiler and ISA
  - it's not always safe, because of:
    - missing instructions *(e.g. FP divide, square root, sin, cos, . . .)*
    - optimizing compilers

3 Cache Memories

In computing, a cache is a hardware or software component that stores data so that future request for that data can be server faster. The data stored in a cache might be the result of an earlier computation or a copy of data that are stored elsewhere.

3.1 Why use cache?

The reasone that why we use cache is because we need to improve performance of memory, because the time access of it is always slower than CPU's clock cycle. So our goal is:

- Provide the user the illusion to use a memory that is fast and large at the same time.
- Provide the data to processor at high frequency.

3.2 How a cache is made?

A cache i made up of block, that is a fixed-size colloction of data containg the requested word.
The number of *Cache blocks* is calculated from *Cache Size* and *Block Size* by the following formula:

$$\textit{Number Of Blocks} = \frac{CacheSize}{BlockSize}$$

To understand how a cache works we need to introduce two crucial thing

- *Temporal Locality*, it's tells us that we are likely to need this word again in near future.
- *Spatial Locality*, rappresent the trend that if we need to use a memory space there is an high probability that we need again an near memory space.

The cache exploits both types of predictability.

3.3 Definition of cache

If we need to use a cache we need to know some terminology.

- **Hit Rate**, is the number of memory access that find the data in the upper level[1] with respect to the total number of memory accesses.
$$HitRate = \frac{Hits}{MemoryAccess}$$
- **Hit Time**, time to access the data in the upper level of the hierarchy, including the time nedeed to understand if the attempt is and hit or a miss.
- **Miss Rate**, number of memory access that don't find the data in the upper level.[2]
$$MissRate = \frac{Miss}{MemoryAccess}$$
- **Miss Penalty**, time needed to access the lower level and to replace the block in the upper level, with this date we can calculate ***Miss Time*** by this formula:
$$MissTime = HitTime + MissPenalty$$
- **Average Memory Access Time** $AMAT$, is described by this formula.
$$AMAT = HitTime + MissRate \cdot MissPenalty$$

[1]There are a lot of level of memory, also in cache
[2]The sum of $MissRate + HitRate = 1$ must be this

3.4 Cache Structure

Eache entry in the cache must follow a structure that is:

- **Valid bit** indicate if the position contains valid data.
- **Cache tag** containts the value that univocally identifies the memory addres corrisponding to the stored data.
- **Cache Data** containt the data.[3]

In cache the most common problem is the block placement, because we need to find the corrispondence between the memory address of the block and the cache address of the block, there are 3 principal types:

- Direct Mapped, means that each memory location corresponds to one and only one cache location. The cache address of the block is given by to bigger block:
  - Block address, that contains tag and index
  - Block offeset, that contains word offset and byte offset
- Fully Associative, the memory block can be placed in any position of the cache, the index doesn't exist in the memory address.
- N-Way Set Associative, the cach is composed of sets each set composed of n blocks, the sets are calculated by the following formula:

$$NumberOfSets = \frac{CacheSize}{BlockSize \cdot N}$$

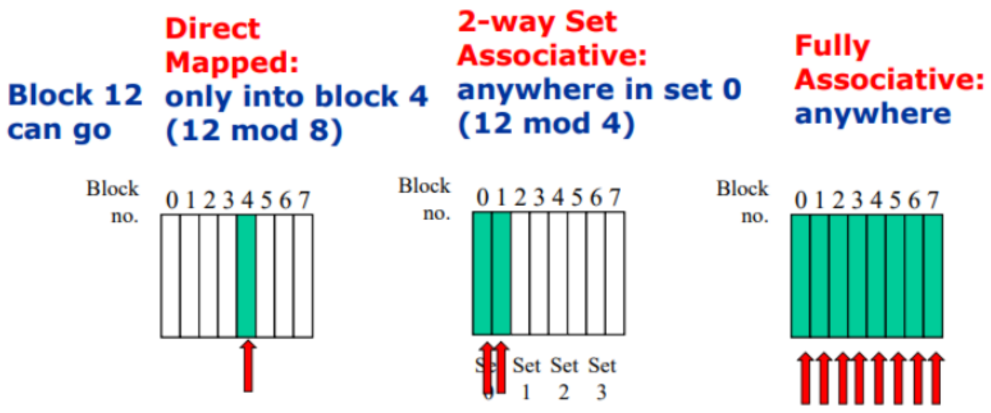

Block 12 can go
Direct Mapped: only into block 4 (12 mod 8)
2-way Set Associative: anywhere in set 0 (12 mod 4)
Fully Associative: anywhere
Block no.
0 1 2 3 4 5 6 7
Set 0
Set 1
Set 2
Set 3

Figure 5: Recap of cache memory

[3]grazie al cazzo

3.5 How to improve Cache?

The increment of cache throughput or a hit rate is an important job in CPU development.

3.5.1 Increment of associativity

This type of improvement will reduce the miss rate, but is a double sword[4] brings a higher cost and increment of hit time caused by the bigger number of sets.

3.5.2 Block Replacement

In case of a miss in a *Fully associative cache*, we need to decide which block to replace, in *set-associative*, we need to select among the blocks of the selected set, meanwhile, in *a direct mapped cache*, we must replace the selected one.
For every type of cache, except for directly mapped, we need to create strategies to change, there are three principal types:

- Random
- Least Recently Used
- FIFO

But the choosing strategy is not enough for a sufficient fix, we need a ***Write Policy***

3.6 Write Policy

This policy defines how to write in a selected block.

3.6.1 Write-Through

The information is written to both the block in the cache and the block in the lower level

3.6.2 Write-Back

The information is written ***only*** to the block in the cache, with this policy we need to introduce a **Dirty bit**

3.6.3 Main Difference

The main difference between these two policies can be reassumed in this simple scheme:

- Write Back:
  - The block can be written by the processor at the frequency at which the cache, and not the main memory can accept it
  - Multiple writes to the same block require only a single write to the main memory
- Write Through:
  - Simpler to be implemented, but it needs to create a *Write buffer* (3.6.4)
  - The read miss is cheaper because they do not require any writing to lower level
  - The memory is always up to date

3.6.4 Write Buffer

Insert a **FIFO buffer** to not wait for lower-level memory access, so in this way, the processor writes data to the cache and writes buffer, and the memory controller will write all the content into the main memory. There's still a problem, the saturation of this buffer.

3.7 Write Miss Option

This option explains what to do if a write miss occurs.

[4]Lama a doppio taglio? Non lo so sono mezzo analfabeta

3.7.1 Write Allocate

Allocate a new cache line in the cache then write, is a double write to cache, which usually means that you must do a "read miss" to fill in the rest of the cache line.
Usually is used with *a Write-Back* policy

3.7.2 No-Write allocate

Simply send write data to lower-level memory, and don't allocate a new line.
Usually is used with *Write-Through* policy

3.8 Memory Hierarchy

The goal of a CPU developer is to give the illusion of unlimited amounts of memory with low latency, but fast memory(*SRAM*) cost a lot with respect to cheaper and slower like *DRAM*.
To jump out of this problem we organize the memory system into a hierarchy, with *Temporal and Spatial Locality* we can ensure that nearly all references can be found in the smaller memories, reaching our goal.

3.9 Classification of Cache Miss

The cache miss could be categorized into three major types:

- Compulsory Misses, the first access to a block that is not in the cache, is also called ***Cold Start.***
- Capacity Misses, if the cache cannot contain all the blocks needed during the execution of a program, capacity misses will occur due to blocks being replaced and later retrieved.
- Conflict Misses can occur only in set-associative or direct mapped, this occurs because a block can be replaced and later retrieved when other blocks map to the same location in the cache.

4 Improve cache performance

There are three ways to improve the cache performance, and all of these refer to a reduction of something in the ***AMAT*** calculation.

4.1 Reducing the miss rate

There are different ways to reduce the miss rate:

1. Increase the cache capacity, but this brings an increase in cost, hit time, and power consumption.
2. Increasing the block size, this increase misses penalty and conflict misses if the cache is small.
3. Higher Associativity, this increase hit time, area, power consumption, and cost.
   Is possible to use the ***MultiBanked caches***, this type of cache introduces a sort of associativity, by organizing the cache as independent banks to support simultaneous access to increase the cache bandwidth.
4. Victim Cache is a small fully associative cache used as a buffer to place data discarded from the cache to better exploit temporal locality, is placed between the cache and its refilling path. Is checked on a miss to see if it has the required data before going to lower-level memory, if the block is found the victim block and the cache block are swapped.
5. Pseudo-Associativity and Way Prediction:
   - Way prediction uses extra bits to predict for each set which of the n-ways[5] to try on the next cache access.
   - Pseudo-associativity[6] divides the cache into two banks in a sort of associativity.
6. Hardware Pre-Fetching of instruction and data, to exploit locality pre-fetch next instruction before they are requested by the processor, this solution relies on extra memory bandwidth that can be used without penalty. But if pre-fetching interferes with demand misses it can lower performance.
7. Software Pre-Fetching data, compiler inserts prefetch *LOAD* instruction to load data in registers/cache before they are needed, this adds an overhead due to LOAD instruction.
8. Compiler Optimizations, the idea is to apply profiling on SW application, the use profiling info to apply code transformation.
   - Reordering instruction to reduce conflict misses.
   - Merging Arrays to improve spatial locality
   - Loop Interchange to improve spatial locality by changing loops nesting access data in the order stored in memory
   - Loop Fusion to improve spatial locality by combining two independent loops
   - Loop Blocking improves temporal locality by accessing the "Sub-Block" of data repeatedly.

4.2 Reducing the miss penalty

1. Read Priority over Write on Miss, giving higher priority to reading misses overwrites to reduce the miss penalty, the write buffer must be properly sized. This approach can complicate memory access because the write buffer might hold the updated value of a memory location needed on a read miss.
   - Write Through with write buffers might generate *RAW* conflicts with main memory read on cache misses, so we need to read the contents of the write buffer on a read miss if there are no conflicts we let the memory access continue sending the read miss before the write, otherwise, the read miss has to wait until the flush of the write buffer.
   - Write Back, instead of writing the dirty block to memory we could copy the dirty block to a write buffer then do the read miss and after write to memory, this reduces CPU stalls.
2. Sub-Block placement, we move the sub-block instead of the full block. We need valid bits, but we can't exploit enough spatial locality.
3. Early Restart and Critical Word First, usually the CPU needs just one word of the block on a miss, so we don't wait for the full block to be loaded before restarting the CPU.
   - Early restart, request the words in normal order from memory, but as soon as the requested word of the block arrives, send it to the CPU to let it continue execution while filling the rest of the block.

[5]Si USA solo nei n-way set-associative
[6]Questo solo in direct mapped

- Critical Word First, request the missed word first from memory and send it to the CPU as soon as it arrives to let the CPU continue execution while filling the rest of the cache block.

These two technique is generally used only for large blocks and spatial locality tend to want the next sequential words.

4. Non-Blocking Cache allows the data cache to continue to supply cache hits during a previous miss.
    - Hit under Miss reduces the effective miss penalty by working during a miss instead of stalling CPU on misses, requires out-of-order execution CPU, and is a sort of out-of-order pipelined memory access.
    - Hit under Multiple Miss or Miss under Miss, may further lower the effective miss penalty overlapping multiple misses, requiring multiple memory banks to serve multiple misses. This increases the complexity of the cache controller.
5. Introduce a second-level Cache, large enough to capture many accesses that would go to main memory reducing the effective miss penalty.
6. Merging Write Buffer, each entry of the write buffer can merge words from different memory addresses, trying to reduce stalls due to a full write buffer.

4.3 Reducing the hit time

1. Fast hit times via small and simple L1 Cache, direct mapped, small and simple on-chip L1 cache.
2. Avoiding Address Translation, avoiding virtual address translation during indexing of cache, if the index is a physical part of the address can start tag access in parallel with address translation so that can compare to the physical tag.
3. Pipelined writes, to pipeline tag check and update cache data as separate stages, the delayed write buffer must be checked on reads either complete write or read from write buffer.
4. Small sub-blocks for write-through, if most write are 1-word, and sub-block are sized in 1-word write-through always write sub-block and tag immediately.
    - Tag Match and valid bit set, writing the block was proper, and nothing was lost by setting a valid bit on again.
    - Tag Match and valid bit not set, writing data into the sub-block makes it appropriate to turn the valid bit on.
    - Tag mismatch, this is a miss and will modify the data portion of the block.

5 Pipeline

5.1 Complex in-order pipeline

While using floating points operations, a few questions might arise:

> *"What happens, architecture-wise, when mixing integer and floating point operations? How are different registers handled? How can GPRs (general purpose registers) and FPRs (floating point registers) be matched?"*

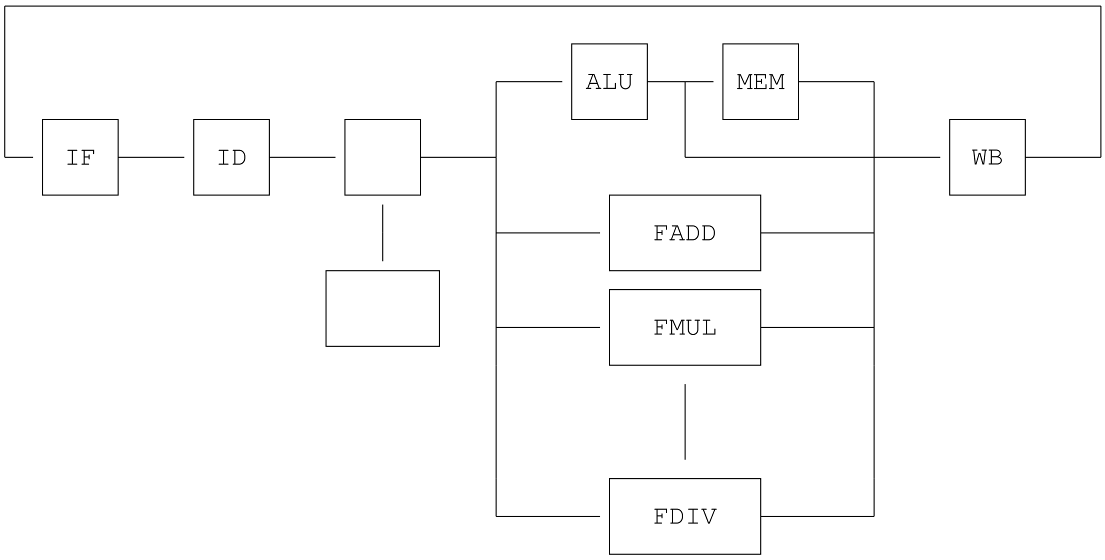


Figure 6: Complex-in order pipeline

In the complex in-order pipeline there isn't a single *ALU* anymore but the execution of floating point operations is split between a number of *Functional Units*. The *issue* stage detects conflicts while accessing any of them and it's able to delay the execution *(by using stalls)* of instructions in case of errors.
The pipeline is now constituted by 6 stages, normally represented as in Figure 7.

IF	ID	IS	EX	ME	WB

Figure 7: Pipeline stages with issue

Pipelining becomes complex when we want high performance in the presence of:

- **Long latency** or partially pipelined floating point units
- **Multiple functions** and memory units
- Memory systems with **variable access time**
- Precise **exception**

Formally, all the different executions must be balanced.

The main issues are found in:

- **Structural conflicts at the execution stage** if some *FPU* or memory unit is not pipelined and takes more than one cycle
- **Structural conflicts at the write-back stage** due to variable latencies of different Functional Units (or *FUs*)

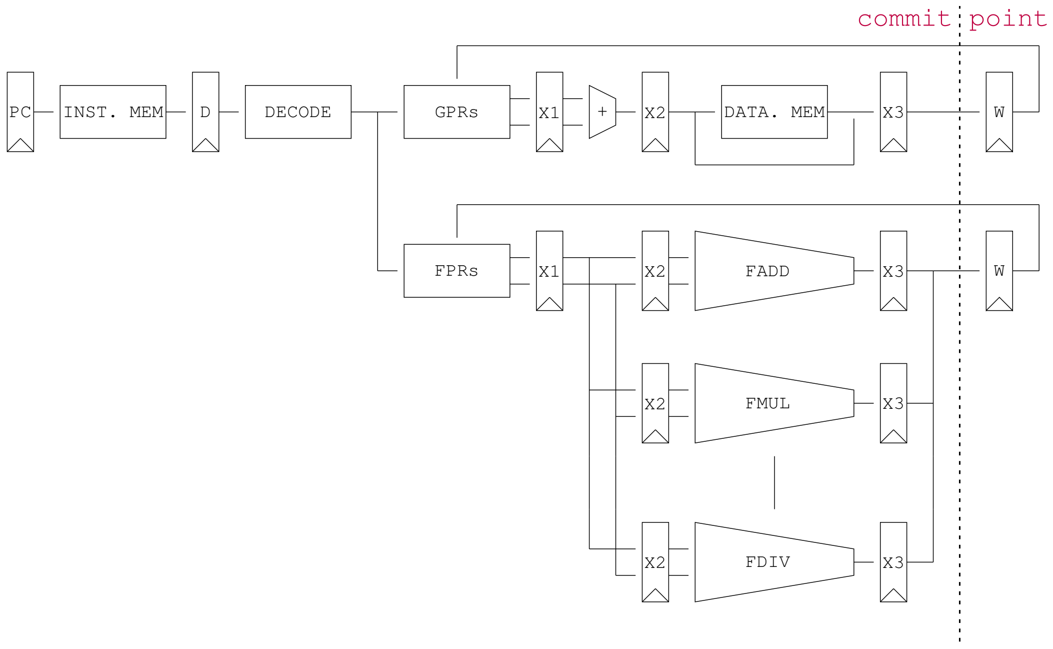


Figure 8: Complex pipelining

- **Out of order write hazards** due to variable latencies of different *FUs*
- Hard to handle exceptions

A possible technique to handle write hazards without equalizing all pipeline depths and without bypassing is by delaying all writebacks so all operations have the same latency into the `WB` stage. In this approach, the following events will happen:

- Write ports are **never oversubscribed**
    - one instruction *in* and one instruction *out* for every cycle
- Instruction commit happens **in order**
    - it simplifies the precise exception implementation

How is it possible to prevent increased write-back latency from slowing down single-cycle integer operations? Is it possible to solve all write hazards without equalizing all pipeline depths and without bypassing them?

5.2 Instructions issuing

To reach higher performance, more parallelism must be extracted from the program. In other words, dependencies must be detected and solved, while instructions must be scheduled to achieve the highest parallelism of execution compatible with available resources.
A data structure keeping track of all the instructions in all the functional units is needed. In order to work properly, it must make the following checks before the `issue` stage to dispatch an instruction:

1. Check if the **functional unit** is available
2. Check if the **input data** is available
    - *Failure in this step would cause a RAW*
3. Check if it's safe to write to the **destination**
    - *Failure in this step would cause a WAR or a WAW*
4. Check if there's a **structural conflict** at the `WB` stage

Such a suitable data structure would look like in Table 2.
An instruction at the `issue` stage consults this table to check if:

name	*busy*	*op*	*destination*	*source 1*	*source 2*
`int`					
`mem`					
`add 1`					
`add 2`					
`add 3`					
`mult 1`					
`mult 2`					
`div`					

Table 2: Data structure to keep track of *FUs*

- The *FU* is **available** by looking at the *busy* column
- A *RAW* can arise by looking at the *destination* column for its **sources**
- A *WAR* can arise by looking at the *source* columns for its **destinations**
- A *WAW* can arise by looking at the *destination* columns for its **destinations**

When the checks are all completed:

- An entry is **added** to the table if no hazard is detected
- An entry is **removed** from the table after `WB` stage

Later in the course *(Section 8.3.2)*, this approach will be discussed more in-depth.

5.3 Dependences

Determining **dependences** among instructions is critical to defining the amount of parallelism existing in a program. If two instructions are dependent, they cannot execute in parallel: they must be executed in order or only partially overlapped.
There exist 3 different types of dependences:

- **Name** Dependences
- **Data** Dependences
- **Control** Dependences

While hazards are a property of the pipeline, dependences are a property of the program. As a consequence, the presence of dependences does not imply the existence of hazards.

5.3.1 Name Dependences

Name dependences occurs when 2 instructions use the same register or memory location *(called name)*, but there is no flow of data between the instructions associated with that *name*. Two types of name dependences could exist between an instruction `i` that precedes an instruction `j`:

- **Antidependence**: when `j` writes a register or memory location that instruction `i` reads; the original instruction ordering must be preserved to ensure that `i` reads the correct value
- **Output Dependence**: when `i` and `j` write the same register or memory location; the ordering of the original instructions must be preserved to ensure that the value finally written corresponds to `j`

Name dependences are not true data dependences, since there is no value *(no data flow)* being transmitted between instructions. If the name *(either register or memory location)* used in the instructions could be changed, the instructions do not conflict.

Dependences through memory locations are more difficult to detect (this is called the *"memory disambiguation"* problem), since two different addresses may refer to the same memory location. As a consequence, it's easier to rename a **register** than rename a **memory location**. It can be done either **statically** by the compiler or **dynamically** by the hardware.

5.3.2 Data Dependences

A data or name dependence can potentially generate a data hazard (*RAW* for the former or *WAR* and *WAW* for the latter), but the actual hazard and the number of stalls to eliminate them are properties of the pipeline.

5.3.3 Control Dependeces

Control dependences determine the ordering of instructions. They are preserved by two properties:

1. **Instructions execution in the program order** to ensure that an instruction that occurs before a branch is executed at the right time *(before the branch)*
2. **Detection of control hazards** to ensure that an instruction *(that is control dependent on a branch)* is not executed until the branch direction is known

Although preserving control dependence is a simple way to preserve program order, control dependence is not the critical property that must be preserved.

6 Branch Prediction

The main goal of the **branch prediction** is to evaluate as early as possible the outcome of a branch instruction. Its performance depends on:

- The **accuracy**, measured in terms of the percentage of incorrect predictions given
- The **cost of an incorrect prediction** measured in terms of time lost to execute useless instructions *(misprediction penalty)* given by the processor architecture
  - the cost increases for deeply pipelined processors
- **Branch frequency** given by the application
  - the importance of accurate branch prediction is higher in programs with higher branch frequency

There are many methods to deal with performance loss due to branch hazards:

- **Static** branch prediction techniques: the actions for a branch are fixed for each branch during the entire execution
  - used in processors where the expectation is that the branch behaviour is highly predictable at compile time
  - can be used to assist dynamic predictors
- **Dynamic** branch prediction techniques: the actions for a branch can change during the program execution

In both cases, care must be taken not to change the processor state until the branch is definitely known.

6.1 Static techniques

Usually it's done at compile time, because is used whene the target application has an highly predictiable branch. There are 5 commonly used branch prediction techniques:

- Branch **always not taken**
- Branch **always taken**
- Backward **taken forward not taken**
- Profile **driven prediction**
- Delayed **branch**

6.1.1 Branch Always Not Taken

The branch is always assumed as **not taken**, thus the sequential instruction flow that has been fetched can continue as if the branch condition was not satisfied. If the condition in the state `ID` will result as not satisfied *(and the prediction is correct)* performance can be preserved.
If the condition in stage `ID` will result in satisfied *(and the prediction is incorrect)* the branch is taken: the next instruction already fetched is flushed *(turned into a nop)* and the execution is restarted by fetching the instruction at the branch target address. There is a one-cycle penalty, so 4 nop.

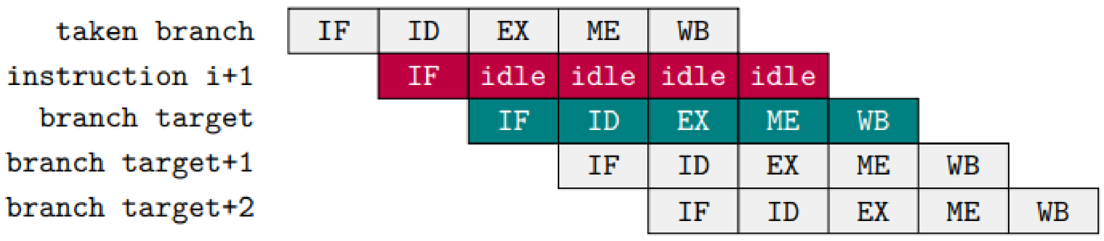


Figure 9: Rappresentation of missprediction

6.1.2 Branch Always Taken

An alternative scheme is to consider **every branch as taken**: as soon as the branch is decoded and the branch target address is computed, the branch is assumed to be taken and the fetching and the execution stages can begin at the target.
The predicted-taken scheme makes sense for pipelines where the branch target is known before the actual outcome. This is not the case within *MIPS* pipeline *(where the branch target is known only after the operation outcome)*, so **there is no advantage in this approach**.

6.1.3 Backward Taken Forward Not Taken

The prediction is based on the branch direction:

- **Backward** going branches are predicted as **taken**
    - the branches at the end of loops are likely to be executed most of the time
- **Forward** going branches are predicted as **not taken**
    - the **if** branches are likely not executed most of the time

6.1.4 Profile Driven Prediction

The branch prediction is based on profiling information collected from earlier runs. This method can use compiler hints, and it's potentially more effective than the other ones; however, it's also the most complicated between the 5 as it needs additional hardware.
This technique will be explored in Section 6.2.

6.1.5 Delayed Branch

The compiler statically schedules an independent instruction in the branch delay slot, which is then executed whether or not the branch is taken.
If the branch delay consists of one cycle (as in *MIPS*), there's only one delay slot; almost all processors with delayed branches have a single delay slot, as it's difficult for the compiler to fill more than one of them.
If the branch:

- **Is untaken**: the execution continues with the instruction after the branch
- **Is taken**: the execution continues at the branch target

The compiler's job is to make the instruction placed in the branch delay slot valid and useful. There are three ways in which the branch delay slot can be scheduled:

1. From **before**
2. From **target**
3. From **fall through**

These methods will be better analyzed in the following paragraphs.

In general, the compilers can fill about **half** of the delayed branch slots with valid and useful instructions, while the remaining slots are filled with `nop`. In deeply pipelined processors, the delayed branch is longer than one cycle: many slots must be filled for every branch, thus it's more difficult to fill each of them with *useful* instructions.

The main limitations on delayed branch scheduling arise from:

- **The restriction on the instruction** that can be scheduled in the delay slot
- **The ability of the compiler** to statically predict the outcome of the branch

To improve the ability of the compiler to fill the branch delay slot, most processors have introduced a **cancelling or nullifying branch**. The instruction includes the direction of the predicted branch:

- When the branch **behaves as predicted**, the instruction in the branch delay slot is **executed normally**
- When the branch **is incorrectly predicted**, the instruction in the branch delay slot is **flushed** *(turned into a `nop`)*

With this approach, the compiler does not need to be as conservative when filling the delay slot.

6.1.5.1 From before

The branch delay slot is scheduled with an independent instruction **from before the branch**.
The instruction in the branch delay slot is always executed, whether the branch is taken or not. An illustration of this strategy is represented in Figure 10.

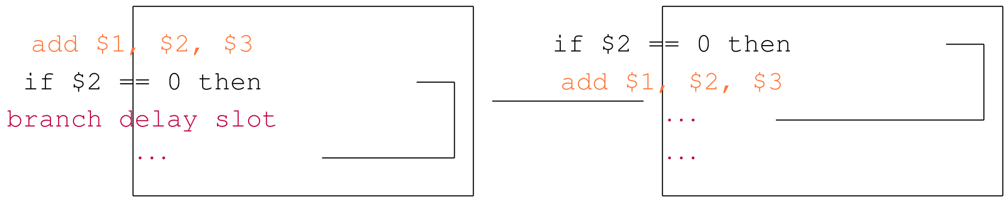


Figure 10: From before

6.1.5.2 From target

The use of a register in the branch condition prevents any instructions with that register as a destination from being moved after the branch itself. The branch delay slot is scheduled from **the target of the branch** *(usually the target instruction will need to be copied because it can be reached by another path)*.
This strategy is preferred when the branch is taken with high probability, such as loop branches **(backward branches)**. An illustration of this strategy is represented in Figure 11.

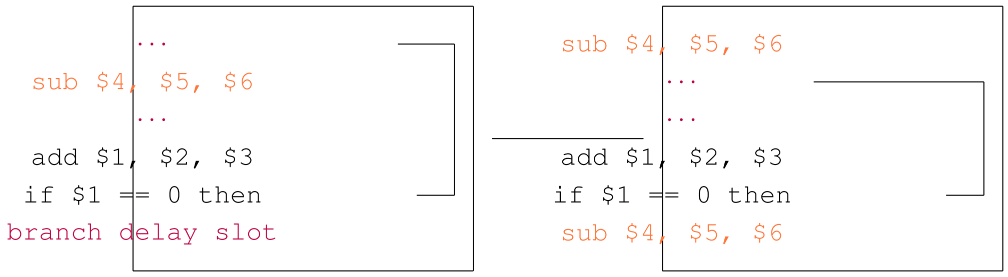


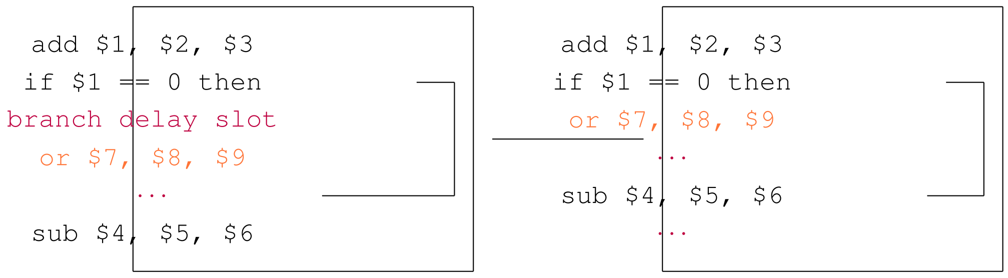
Figure 11: From target

6.1.5.3 From fall through

The use of a register in the branch condition prevents any instructions with that register as a destination from being moved after the branch itself *(like what happens in the from target technique)*. The branch delay slot is scheduled from **the not taken fall through path**.
This strategy is preferred when the branch is not taken with high probability, such as **forward branches**. An illustration of this strategy is represented in Figure 12.
In order to make the optimization legal for the target, it must be ok to execute the moved instruction when the branch goes in the expected direction; the instruction in the branch delay slot is executed but its result is wasted *(if the program will still execute correctly)*.
For example, if the destination register is an unused temporary register when the branch goes in an unexpected direction.

6.2 Dynamic Branch Prediction

Basic idea: use the past branch behaviour to predict the future.
Hardware is used to dynamically predict the outcome of a branch: the prediction will depend on the behaviour of the branch at **compile time** and will change if the branch changes its behaviour during **run time**.



```
add $1, $2, $3
if $1 == 0 then
branch delay slot
or $7, $8, $9
...
sub $4, $5, $6
```

```
add $1, $2, $3
if $1 == 0 then
or $7, $8, $9
...
sub $4, $5, $6
...
```

Figure 12: From fall through

Dynamic Branch Prediction is based on two interacting mechanisms:

1. **Branch Outcome Predictor** *(BOP)*
   - used to predict the direction of a branch *(taken or not taken)*
2. **Branch Target Predictor** *(BTP)*
   - used to predict the branch target address in case of taken branch

These modules are used by the *Instruction Fetch Unit* to predict the next instruction to read in the instruction cache *(also called I-cache)*:

- Branch **is not taken**: `PC` is incremented
- Branch **is taken**: *BTP* gives the target address

6.2.1 Branch Target Buffer

The **Branch Target Buffer** is a cache storing the predicted branch target address for the next instruction after a branch. The *BTB* is accessed in the `IF` stage using the instruction address of the fetched instruction *(a possible branch)* to index the cache.
The typical entry of the *BTB* is shown in Figure 13. The predicted target address is expressed as `PC`-relative.

address of a branch instruction	predicted destination address

Figure 13: Typical entry of the *BTB*

6.2.2 Branch History Table

The **Branch History Table** contains 1 *bit* for each entry that says whether the branch was recently taken or not. It is indexed by the lower portion of the address of the branch instruction.
The prediction is a hint that it is assumed to be correct and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back; the pipeline is then flushed and the correct sequence is executed.
The table has no tags *(every access is a hit)* and the prediction bit could have been put there by another branch with the same LSBs. The 1-bit branch history table only considers the last status of the branch *(either taken or not taken)*. It is a simple *FSA* where a misprediction will change the current value.
A **misprediction** occurs when:

- The prediction is **incorrect** for that branch
- The same index has been referenced by two different branches, and the **the previous history refers to the other branch**

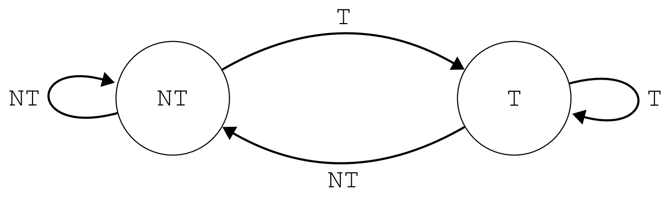


Figure 14: 1-bit *BHT* as *FSA*

- to solve this problem it's enough to increase the number of rows in the *BHT* or to use a hashing function *(such as GShare).*

In a loop branch, even if a branch is almost always taken and then not taken one, the 1-bit *BHT* will mispredict **twice** *(rather than once)* when it is not taken. That situation causes two wrong predictions:

- At the **last loop iteration**
    - the loop must be exited
    - the prediction bit will say `TAKE`
- While **re-entering the loop**
    - at the end of the first iteration the branch must be taken to stay in the loop
    - the prediction bit will say `NOT TAKE` because the bit was flipped on the previous execution of the last iteration of the loop

To fix this kind of behaviour, the 2-bit *BHT* was introduced.

6.2.3 *2*-bit Branch History Table

By adding one bit to the *BHT*, the prediction must miss twice before it is changed. In a loop branch, there's no need to change the prediction for the last iteration.
For each index in the table, the 2 bits are used to encode the four states of a *FSA*.

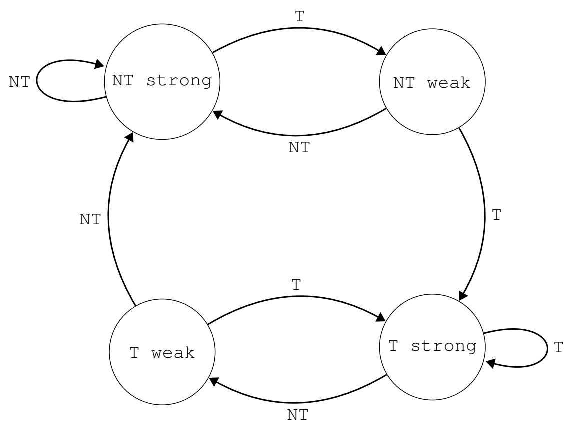


Figure 15: 2-bit *BHT* as *FSA*

6.2.4 *k*-bit Branch History Table

It's a generalization: n-bit saturating counter for each entry in the prediction buffer.
The counter can take on values between 0 and 2^{n-1}. When the counter is greater than or equal to one-half of its maximum value, the branch is predicted as taken. Otherwise, it's predicted as untaken.

As in the 2-bit scheme, the counter is incremented on a taken branch and decremented on an untaken branch. Studies on n-bit predictors have shown that 2 bits behave almost as well *(so using more than* 2 *bits is almost useless).*

6.3 Correlating Branch Predictors

Basic idea: the behaviour of recent branches is correlated, that is the recent behaviour of other branches rather than just the current branch that we are trying to predict can influence the prediction of the current branch.
The **Correlating Branch Predictors** are predictors that use the behaviour of other branches to make a prediction. They are also called 2-*level Predictors.* A $(1, 1)$ Correlating Predictor denotes a 1-bit predictor with 1-bit of correlation: the behaviour of the last branch is used to choose among a pair of 1-bit branch predictors.

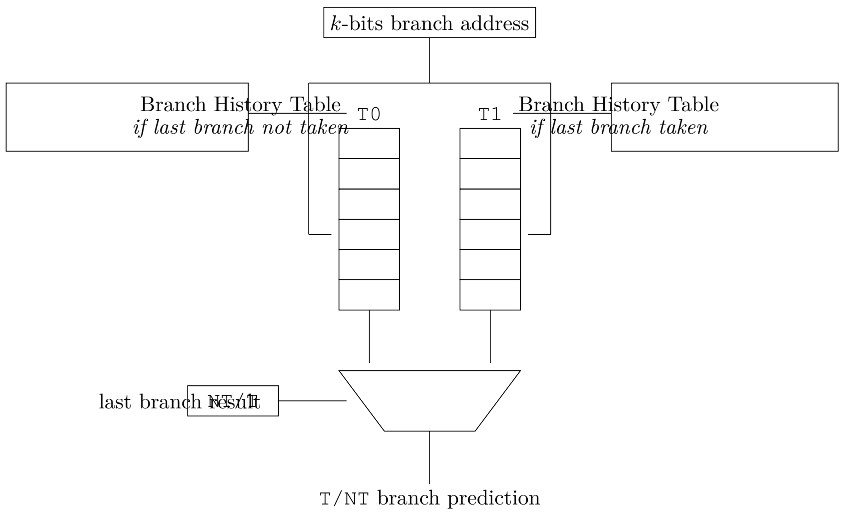


Figure 16: Structure of the *Correlating Branch Predictors*

6.3.1 *(m, n)* Correlating Branch Predictors

In general, (m, n) correlating predictors records last m branches to choose from 2^m *BHTs*, each of which is a n-bit predictor.
The branch prediction buffer can be indexed by using a concatenation of low order bits from the branch address with m-bit global history *(i.e. global history of the most recent m branches, implemented with a shift register).*

6.3.1.1 A *(2, 2) Correlating Branch Predictor*

A $(2, 2)$ correlating predictor has 4 2-bit Branch History Tables. It uses the 2-bit global history to choose among the 4 *BHTs*.

- Each *BHT* is composed of 16 entries of 2-bit each
- The 4-bit branch address is used to choose four entries *(a row)*
- 2-bit global history is used to choose one of four entries in a row *(one of the four BHTs)*

6.3.1.2 Accuracy of Correlating Predictors

A 2-bit predictor with no global history is simply a $(0, 2)$ predictor.
By comparing the performance of a 2-bit simple predictor with 4000 entries and a $2, 2$ correlating predictor with 1000 entries, we find out that the latter not only outperforms the 2-bit predictor with the same number of total bits but also often outperforms a 2-bit predictor with an unlimited number of entries.

6.4 Two Level Adaptive Branch Predictors

The first level history is recorded in one *(or more)* k-bit shift register called *Branch History Register (BHR)* which records the outcomes of the k most recent branches. The second level history is recorded in ore *(or more)* tables called *Pattern History Table (PHT)* of two bits saturating counters.
The BHR is used to index the PHT to select which 2-bit counter to use. Once the two-bit counter is selected, the prediction is made using the same method as in the two bits counter scheme.

6.4.1 GA and GShare Predictors

The **GA Predictor** *(Genetic Algorithm Predictor)* is composed of a *BHT (local predictor)* and by one or more *GAs (local and global predictor)*:

- The *BHT* is indexed by the low order bits of the `PC` *(the branch address)*
- The *GAs* are a 2-level predictor: *PHT* is indexed by the content of *BHR (from global history)*

The **GShare Predictor** is a local `XOR` global information, indexed by the exclusive `OR` of the low order bits of *PC (branch address)* and the content of *BHR (global history)*.

6.5 Domanda quiz

Branch prediction is always advantage?
On average i have a set of branch and i will be always advantage, because maybe the prediction can be wrong and I have to pay 1 clock cycle, but i want to optimize the big picture not the single instruction.

7 Exception Handling

Interrupts are special events that alter the normal program execution by requesting the attention of the processor. They can be raised either by an **internal** or an **external** system, and they are usually unexpected or rare from the program's point of view.
When they are raised, a special routine called **Interrupt Handler** will take care of stopping and resuming the flow of the code, while addressing the exception.

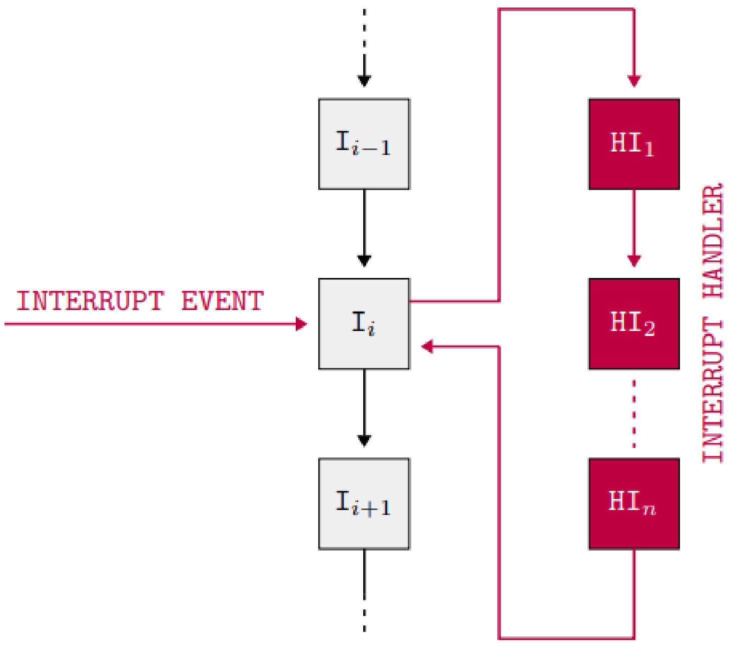


Figure 17: Interrupt event

The interrupts can be:

- **Asynchronous**: created by an **external event**, for example:
  - input or output device service request
  - timer expiration
  - power disruption or hardware failure
- **Synchronous**: created by an **internal event**, for example:
  - undefined *opcode*, arithmetic overflow, *FPU* exception
  - privileged instruction, misaligned memory access
  - virtual memory exceptions
    - page faults
    - *TLB* misses
    - protection violations
  - traps
    - system calls
    - jumps to kernel

Synchronous events are also called **Exceptions**.

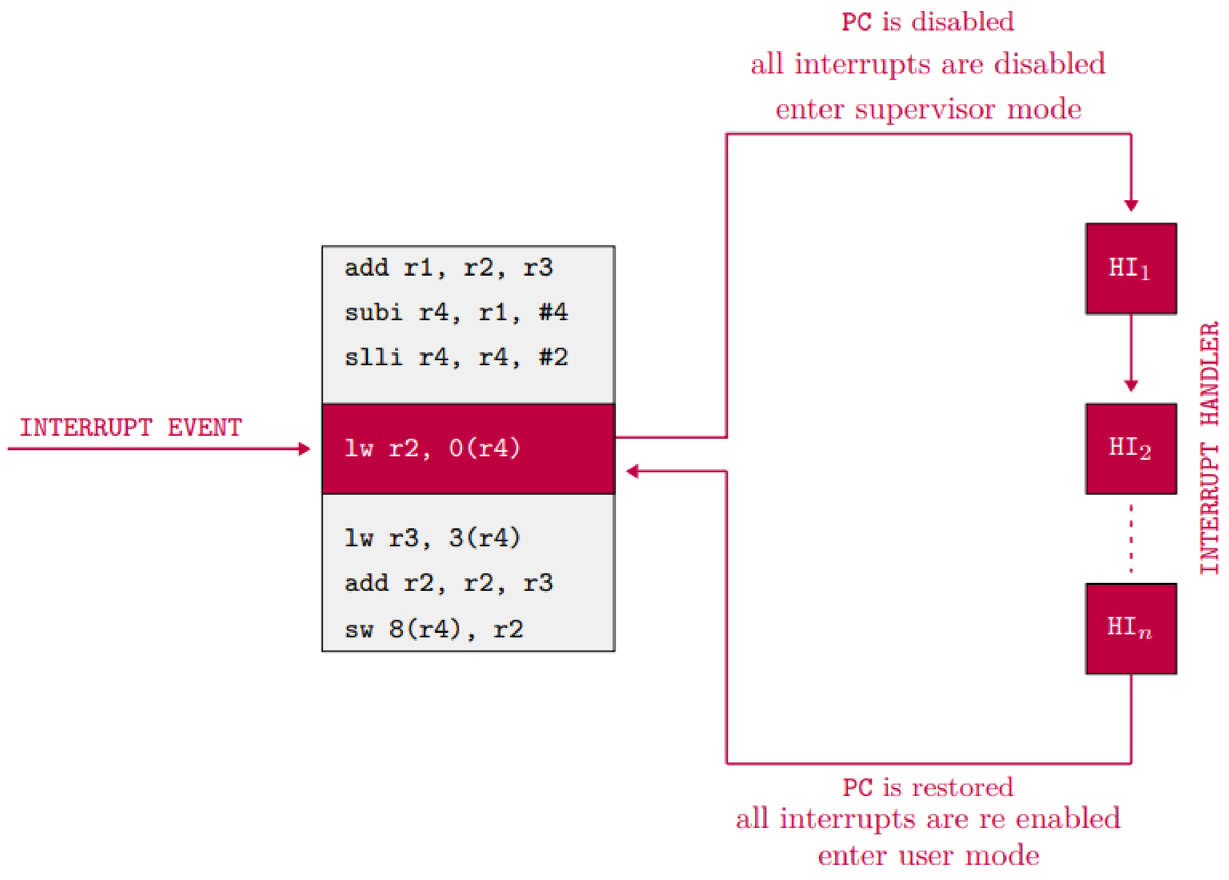


Figure 18: Operation of an interrupt

7.1 Precise Interrupts

An interrupt or exception is considered precise if there is a single instruction *(or interrupt point)* for which all instructions before that one have committed their state and no following instructions *(including the interrupting one)* have modified any state.
This effectively implies that the execution can be restarted at the interrupt point and continue correctly.

This kind of interrupt is desirable because:

- Many types of interrupts or exceptions **need to be restartable**
- It's **easier to figure out what actually happened** and what caused the exception

While restartability does not require preciseness, it makes it a lot easier to restart the execution by:

- Reducing the **number of states** to be saved if the process has to be unloaded
- Making **quicker restarts** due to the faster interrupts

The operation of an interrupt is shown in Figure 18.

7.2 Classes of Exceptions

Exceptions can be divided into classes:

- **Synchronous** and **Asynchronous**:
    - Asynchronous exceptions are caused by devices external to the *CPU* and memory and are easier to handle as they can be addressed after the current instruction
- **User requested** and **Coerced**:
    - User requested are predictable, they are treated as exceptions because they use the same mechanisms that are used to save and restore the state and are handled after the instruction has been completed
    - Coerced exceptions are caused by some hardware event, not under the control of the program
- **User Maskable** and **User Nonmaskable**:
    - The mask controls whether the hardware responds to the exception or not
- **Within** vs **Between** instructions:
    - Exceptions that occur between instructions are usually synchronous as they are triggered by instruction. The instruction must be stopped and restarted
    - Asynchronous that occur between instructions arise from catastrophic situations and cause program termination
- **Resume** and **Terminate**
    - With a terminating event, the program execution always stops
    - With resuming events, the program execution continues after the interrupt

7.2.1 Asynchronous Interrupts

The **Asynchronous Interrupts** work by:

1. Invoking the **Interrupt Handler**
    - an *I/O* device requests attention by asserting one of the prioritized interrupt request lines
2. When the processor decides to **address the interrupt**, it has to:
    1. stop the current program at instruction $\texttt{I}_i$
    2. complete all the instructions up until $\texttt{I}_{i-1}$ *(precise interrupt)*
    3. save the `PC` of instruction $\texttt{I}_i$ in a special register called *EPC*
    4. disable interrupts and transfer control to a designated interrupt handler running in the kernel mode

Then the operation is handled by the **Interrupt Handler**, which will:

3. **Save** the *PC* before enabling interrupts to allow nested interrupts
    - it needs an instruction to move the `PC` into *GPRs*
    - it needs a way to mask further interrupts at least until the `PC` can be saved
4. **Read** a *status register* that indicates the cause of the interrupt
5. **Use** a special indirect jump instruction (`RFE`, *Return From Exception*) which:
    - enables interrupts
    - restores the processor to the user mode
    - restores hardware status and control state

7.2.2 Synchronous Interrupts

A **Synchronous Interrupt** is caused by a **particular** instruction. Generally, the instruction **cannot be completed** and needs to be **restarted** after the exception has been handled. This procedure requires undoing the effect of one or more partially executed instructions.
In case the interrupt was raised by a system call trap *(a special jump instruction involving a change to privileged kernel mode)*, the instruction is considered to have been completed.

7.3 Precise interrupts in 5 stages pipeline

Exceptions may occur at different stages in the pipeline, due to the out-of-order executions. For instance, *arithmetic exceptions* occur in `EX` stage, while *TLB faults* occur in `IF` or `ME` stages.
The same issue arises while handling interrupts, as the pipeline must be interrupted as little as possible.
This problem can be solved by tagging instructions in the pipeline as *"cause exceptions or not"* and waiting until the end of `ME` stage to flag exceptions. Then:

- Interrupts become marked `NOP` *(like bubbles)* that are placed into pipeline instead of an instruction
- Interrupt conditions are assumed to be persistent in case of flushed `NOP` instructions
- A clever `IF` stage might start fetching instructions from the interrupt vector
    - this step is complicated by the need for supervisor mode switch, saving multiple `PC` and more similar issues

In detail, exceptions are handled by:

- Holding exceptions flags in pipeline until the commit point *(`ME` stage)*
- Overriding newer exceptions for given instructions with older exceptions
- Injecting external interrupts at commit point, overriding other interrupts
- If an exception happens at commit, updating *CAUSE* and *ECP* registers, killing all stages and injecting handler `PC` into `IF` stage

While dealing with an in-order pipeline might be easy, generating a precise interrupt when instructions are being executed in an arbitrary order is not as easy. In his famous paper, *Jim Smith* proposes 3 methods for getting precise interrupts:

- In-order instruction completion
- Reorder buffer
- History buffer

7.3.1 Speculating on Exceptions

There are 3 technique to speculate on exceptions:

- **Prediction** mechanism
    - → exceptions are so rare that predicting no exceptions is surprisingly accurate
- **Check Prediction** mechanism
    - → exceptions are detected at the end of the instruction execution pipeline
    - → dedicated hardware for different exception types
- **Recovery** mechanism
    - → only write architectural state at commit point, so partially executed instruction can be thrown away after exception
    - → exception handler is launched only after the pipeline is flushed

8 Instruction Level Parallelism - *ILP*

The objective of the *ILP* is to improve the *CPI*, with the ideal goal of 1 *cycle per instruction*.
The *ILP* implies a potential overlap of execution among unrelated instructions. This is achievable if there are no hazards.

8.1 Multi-Cycle Pipeline

Pipeline going in order, we keep one instruction at the time in sequential order. On instruction from sequential flow, one by one.
We can have two cases:

- Execution stage: could be multiple cycle. For example: Multiply or divide operation in the alu require more the 1 clk cycle.
- Memory stage: require multiple cycle , this are caused by cache misses, this could cause that it will take 10 or 20 clock cycles.

8.1.1 In-order commit

This means that i do not care in which order they are executed, if one is faster then the other, but i need to commit with the flow order.
Otherwise, if I use out-of-order i need to implement some logic to accept those type of commit, to preserve the absense[7] of conflict like *WAR* *WAW*

8.1.2 Difference between MIPS and Multi-ISSUE

In *MIPS* decode and access to the register were in the same *CLK*, now we add another stage that is goal is to issue the instruction and pipeline in the correct *Functional Unit*.
In this way I can commit in out-of-order with all of the problems that occurs.

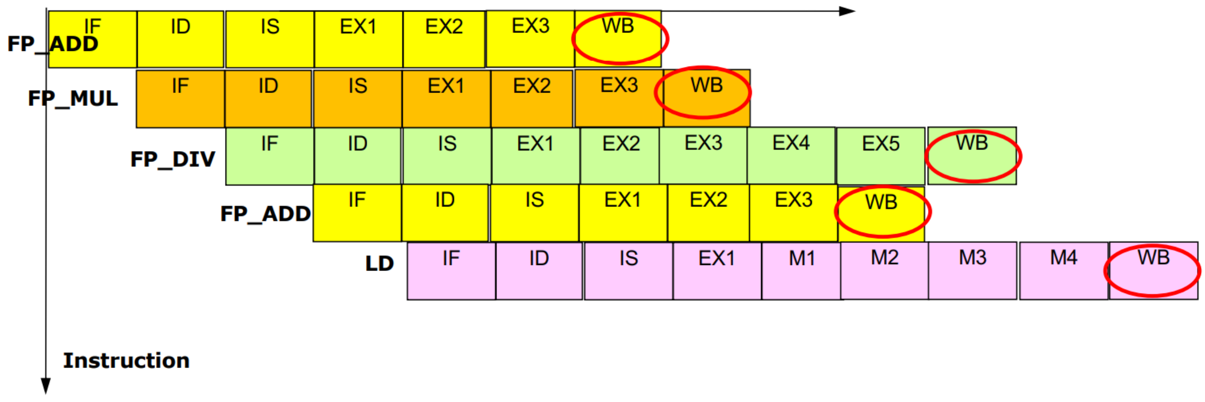
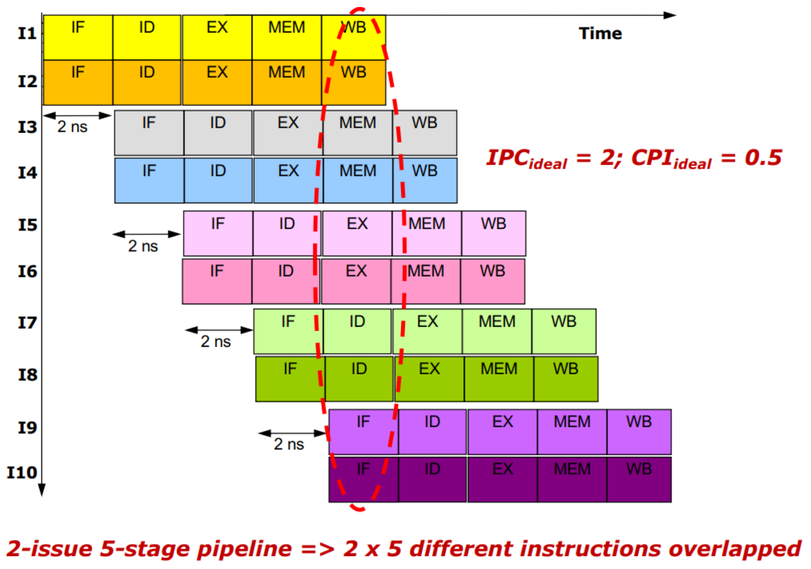


Figure 19: In-order issue / out-of-order commit

[7]assenza? forse

8.2 Dual-Issue Pipeline

At each clock we will fetch and decode two instruction, but can be more, so we have to modify our register file to handle this request. Since we ave a single thread / single process we have to use a register file with 4



I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
IF
ID
EX
MEM
WB
Time
2 ns
IPC_ideal = 2; CPI_ideal = 0.5
2-issue 5-stage pipeline => 2 x 5 different instructions overlapped

Figure 20: Dual Issue pipeline

read/write port to write 2 results.
Also the fetch of instruction cache is double the size. Also forwarding path are more complicated.

8.3 Strategies to support *ILP*

There are main two software strategies to support *ILP*:

1. **Dynamic** scheduling: depends on the hardware to locate parallelism
2. **Static** scheduling: relies on the software to identify potential parallelism

Usually, hardware-intensive approaches dominate desktop and server markets.

8.3.1 Dynamic scheduling

The hardware reorders the instruction execution to reduce pipeline stall while maintaining data flow and exception behaviour.

Properties of the Dynamic Scheduling:

1. Instructions are fetched and **issued in program order**
2. Execution begins **as soon as operands are available**, possibly out of order execution
3. Out of order execution introduces possibility or *WAR* and *WAW* data hazards
4. Out of order execution implies **out of order completion**
    - a *reorder buffer* is needed to reorder the output

Advantages of dynamic scheduling:

- It enables handling some cases where dependencies are **unknown** at compile time
- It **simplifies** the compiler complexity
- It allows compiled code to run **efficiently** on a different pipeline

Disadvantages:

- A significant increase in **hardware complexity**
- Increased **power consumption**
- Could generate **imprecise exception**

8.3.2 Scoreboard

A specific data structure is needed to solve data dependences without specialized compilers. The first implementation of such hardware is found in the **Scoreboard**, created in *1963*.
Its key idea is to allow instruction behind stalls to proceed, with the result of a 250% speedup with regards to no dynamic scheduling and a 170% speedup with regards to instructions reordering by the compiler. It has the downside of having a **slow memory** *(due to the absence of cache)* and **no forwarding hardware**. Furthermore, it has a low number of *FUs* and it does not issue structural hazards.
It solves the issue of data dependencies that cannot be hidden with bypassing or forwarding due to the hardware stalls of the pipeline by allowing out-of-order execution and commit of instructions.

The scoreboard centralizes hazard management. It can avoid them by:

- Dispatching **instructions** in order to functional units provided there's no structural hazard or *WAW*
    - a **stall** is added on structural hazards *(when no functional unit is available)*
    - there can be only one pending write to each register
- Instructions **wait** for input operands to avoid *RAW* hazards
    - as a result, it can execute out-of-order instructions
- Instructions **wait** for output register to be read by preceding instructions to avoid *WAR* hazards
    - results are held in functional units until the register is freed

The scoreboard is operated by:

1. **Sending** each instruction through it
2. **Determining** when the instruction can read its operands and subsequently start its execution
3. **Monitoring** changes in hardware and deciding when a stalled instruction can execute
4. **Controlling** when instruction can write results

As a result, a new pipeline is introduced, where the `ID` stage is divided into two parts:

1. *issue*, where the instruction is decoded and **structural hazards** are checked
2. *read operands*, where the operation waits until there are no **data hazards**

Finally, the scoreboard is structured in three different parts:

1. **Instruction** status
2. **Functional Units** status
    - fields indicating the state of each *FUs*:
        - `Busy` - indicates whether the unit is busy or not
        - `Op` - the operation to perform in the unit
        - `Fi` - the destination register
        - `Fj`, `Fk` - source register numbers
        - `Qj`, `Qk` - functional units producing source registers
        - `Rj`, `Rk` - flags indicating when `Fj`, `Fk` are ready
3. **Register** result status
    - indicates which functional unit will write each register
    - it's `blank` if no pending instructions will write that register

An illustration of the new pipeline is represented in Figure 21, while the structure of the scoreboard is represented in Figure 22.

ID		EX	WB
issue	*read operands*	*execution*	*write back*

Figure 21: Pipeline introduced by the *Scoreboard*

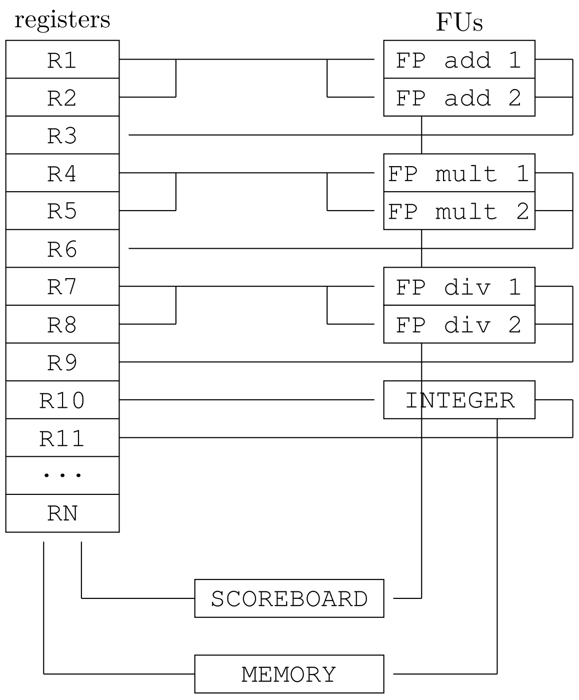


Figure 22: Structure of the *Scoreboard*

8.3.2.1 Four stages of Scoreboard Control

The four stages of scoreboard control are:

- **Issue**: instructions are decoded and structural hazards *(WAW)* are checked for
    - instructions are issued in program order for hazard checking
    - if the *FU* for the instruction is free and no other active instruction has the same destination register, the scoreboard issues the instruction and updates its internal data structure
    - if a structural or *WAW* hazard exists, then the instruction issue stalls and no further instructions are issued until they are solved
- **Read operands**: expiration of data hazards is awaited, then operands are read
    - a source operand if available if no earlier issued active instruction will write it or a functional unit is writing its value into a register
    - when the source operands are available, the scoreboard tells the *FU* to proceed to read the operands from the registers and begin execution
    - *RAW* hazards are resolved dynamically, instructions could be sent out of order
    - there's no data forwarding in this model
- **Execution**: the *FUs* operate on the data
    - when the result is ready, the scoreboard it's notified
    - the delays are characterized by **latency** and **initiation interval**
- **Write result**: the execution is finished
    - once the scoreboard is aware that the *FU* has completed the execution, it checks for *WAR* hazards
        - if no *WAR* hazard is found, the result is written
        - otherwise, the execution is stalled
    - **issue** and **write** stages can overlap

This structure creates a few implications:

- *WAW* are **detected** *(and the pipeline is stalled)* until the other instruction is completed
- There's **no register renaming**
- **Multiple instructions must be dispatched** in the execution phase, creating the need for multiple or pipelined execution units
- Scoreboard **keeps track of dependences** and the state of the operations

8.3.3 Tomasulo algorithm

The Tomasulo algorithm is a **dynamic** algorithm that allows execution to proceed in presence of dependences. The key idea behind this algorithm is to **distribute** the control logic and the buffers within *FUs*, as opposed to the scoreboard *(in which the control logic is centralized)*.
The operand buffers are called **Reservation Stations** *(RS)*: each instruction is also an entry to a *RS* and its operands are replaced by values or pointers *(a technique known as **Implicit Register Renaming**)* in order to avoid *WAR* and *RAW* hazards.
Results are then dispatched to other *FUs* through a *Common Data Bus*, communicating both the data and the source. Finally, `LOAD` and `STORE` operations are treated as *FUs*, as *RS* are more complex than architectural registers to allow more compiler-level optimizations.

8.3.3.1 Structure of the Reservation Stations

The Reservation Station is composed by 5 **fields**:

- `TAG` - indicating the *RS* itself
- `OP` - the operation to perform in the unit
- `Vj`, `Vk` - the value of the source operands
- `Qj`, `Qk` - pointers to the *RS* that produces `Vj`, `Vk`

    - its value is zero if the source operator is already available in `Vj` or `Vk`
- `BUSY` - indicates the *RS* is busy

In this description, only one between the `V`-field and the `Q`-field is valid for each operand.

Furthermore, a few more components are needed, such as:

- **Register File** and the **Store** buffers have a *Value* (`V`) and a *Pointer* (`Q`) field
    - `Q` corresponds to the number of the *RS* producing the result (`V`) to be stored in *Register File* or *Store* buffers
    - if `Q = 0`, no active instructions is producing a result and the *Register File (or `STORE`)* buffer contains the wrong value
- **Load** and **Store** buffers have an *Address* (`A`) field, with the former having also a *Busy* field *(`BUSY`)*
    - the `A` field holds information for memory address calculation: initially contains the instruction offset, while after the calculation it stores the effective address

8.3.3.2 Stages of the Tomasulo Algorithm

The Tomasulo algorithm is structured in 3 different stages: **Issue**, **Execute** and **Write**.

In more detail:

1. **Issue** stage:
    - Get an instruction `I` from the queue
        - if it is an *FP* operation, check if any *RS* in empty *(i.e. check for any structural hazard)*
    - Rename the registers
    - Resolve *WAR* hazards
        - if `I` writes *R*, read by an already issued instruction `K`, `K` will already know the value of *R* or knows that instruction will write into it
        - the *Register File* can be linked to `I`
    - Resolve *WAW* hazards
        - since the in-order issue is used, the *Register File* can be linked to `I`
2. **Execute** stage:
    - When both the operands are ready, then the operation is executed. Otherwise, watch the *Common Data Bus* for results.
        - by delaying the execution until both operands are available, *RAW* hazards are avoided
        - several instructions could become ready in the same clock cycle for the same *FU*
    - `LOAD` and `STORE` are two-step processes:
        - effective address is computed and placed in `LOAD/STORE` buffer
        - `LOAD` operations are executed as soon as the memory unit is available
        - `STORE` operations wait for the value to be stored before sending it into the memory unit
3. **Write** stage:
    - When the result is available, it is written on the *Common Data Bus*
        - it is then propagated into the *Register File* and all the registers *(including `STORE` buffers)* waiting for this result
        - `STORE` operations write data to memory
        - *RSs* are marked as available

8.3.3.3 Focus on `LOAD` and `STORE` in Tomasulo Algorithm

`LOAD` and `STORE` instructions go through a functional unit for effective computation before proceeding to their respective load and `STORE` buffers. `LOAD` take a second execution step to access memory, then go to *Write* stage to send the value from memory to *Register File* and/or *RS*, while `STORE` complete their execution in their *Write* stage. All write operations occur in the write stage, thus simplifying the algorithm.

A `LOAD` and a `STORE` instruction can be done in a different order, provided they access different memory locations. Otherwise, a *WAR (interchange in load-store sequence)* or a *RAW (interchange in store-load sequence)* may result in a *WAW (if two stores are interchanged)*. However, `LOAD` instructions can be reordered freely.
To detect such hazards, data memory addresses associated with any earlier memory operation must have been computed by the *CPU*.

`LOAD` instructions executed out of order with previous `STORE` assume that the address is computed in program order. When the `LOAD` address has been computed, it can be compared with `A` fields in active `STORE` Buffers: in case of a match, the load is not sent to its buffer until conflicting `STORE` completes.
Store instructions must check for matching addresses in both `LOAD` and `STORE` buffers. This is a **dynamic disambiguation** and, opposing to the static disambiguation, is not performed by the compiler. As a drawback, more hardware is required to perform these operations: each *RS* must contain a fast associative buffer, because single *CDB (Common Data Bus)* may limit performance.

8.3.3.4 Tomasulo and Loops

Tomasulo algorithm can **overlap** iterations of loops due to:

- **Register Renaming**
    - **multiple iterations** use **different physical destinations** for registers
    - **static register names are replaced** from code with dynamic registers *"pointers"*, effectively increasing the size of the register file
    - **instruction issue is advanced** past integer control flow operations
- **Fast branch resolution**
    - integer unit must *"get ahead"* of floating point unit so that multiple iterations can be issued

8.3.3.5 Comparison between Tomasulo Algorithm and Scoreboard

The main advantages of the Tomasulo algorithm over the scoreboard are:

- Control and buffers are **distributed** with *FUs*
    - *FUs* buffers are called **reservation stations** and have pending operands
- Registers in instructions are **replaced** by values or pointers to *RS*
    - avoids *WAR* and *WAW* hazards
    - since there are more *RS* than registers, there's a higher optimization than compilers alone can do
- The result are **propagated** from *RS* to *FU* via *Common Data Bus*
    - the value is propagated **to all *FUs***
- `LOAD` and `STORE` instructions are treated as *FUs* with *RSs*
- Integer instructions can go past **branches**, allowing *FP* ops beyond basic block in *FP* queue

8.4 Limits of *ILP*

In order to execute more than one instruction at the beginning of a clock cycle, two requirements must be satisfied:

1. Fetching more than one **instruction per clock cycle**
    - this task is completed by the *Fetch Unit*

- there is no major problem provided the instruction cache *(I-cache)* can sustain the bandwidth and can manage multiple requests at one
2. Decide on data and control **dependences**
    - *dynamic scheduling* and *dynamic branch prediction* are needed

Superscalar architectures paired with compiler scheduling can achieve such speeds.

A few requirements must be satisfied to start an ideal machine:

1. **Register renaming**
    - by using an infinite number of virtual registers, all *WAW* and *WAR* hazards are avoided
2. **Branch prediction**
    - by using a perfect predictor, no branch is ever mispredicted
3. **Jump prediction**
    - all jumps are perfectly predicted
    - a machine with perfect speculation and an infinite buffer of instructions is needed
4. **Memory address alias analysis**
    - addresses are known and a `STORE` can be moved before a `LOAD` if their addresses are different
5. 1 **cycle latency** for all instructions
    - an unlimited number of instructions can be issued each clock cycle

8.4.1 Initial assumptions

Furthermore, a few **initial assumptions** must be made:

- *CPU* can issue an **unlimited number of instructions**, looking arbitrarily far ahead in the computation
- There's **no restriction on types of instructions** that can be executed in one cycle *(including loads and stores)*
- All *FUs* **have unitary latency**: any sequence of depending instructions can issue on successive cycles
- All *LOAD* and *STORE* execute in 1 cycle, thanks to perfect caches

8.4.2 Limits dynamic analysis

Dynamic analysis is necessary to approach perfect branch prediction, and it cannot be achieved at compile time. A perfect dynamic scheduled *CPU* should:

1. **Look arbitrarily far ahead** to find a set of instructions to issue and predict all branches perfectly
2. **Rename all registers** in use, avoiding all *WAR* and *RAW* hazards
3. Determine whether there are **data dependences among instructions** in the issue packet, renaming them if necessary
4. Determine and handle all **memory dependences among issuing instructions**
5. **Provide** enough replicated **functional units** to allow all ready instructions to issue

8.5 Static Scheduling

Compilers can use sophisticated algorithms for code scheduling to exploit *ILP*. The amount of parallelism available within a basic block *(a straight line code sequence with no branches in except to the entry and no branches out except at the exit)* is quite small, and data dependence can further limit the amount of *ILP* that can be exploited within a basic block much less than the average basic block size. To obtain substantial performance enhancements, *ILP* must be exploited across multiple basic blocks *(i.e. across branches)*.
The static detection and resolution of dependencies are accomplished by the compiler, so they are avoided by code reordering. The compiler outputs dependency-free code.

Limits of static scheduling:

- **Unpredictable** branches
- Variable memory **latency** *(due to unpredictable cache misses)*
- Huge increase in **code size**
- High compiler **complexity**, because all of the scheduling logic is related to compiler[8].

8.6 SuperScalar architectures

This type of architectures need a complex hardware, because they are static scheduled. This means a large area and energy consumption, this need a large instruction window.
The data dependency are managed at execution time.

8.7 VLIW architectures

The **Very Long Instruction Words** *(VLIW)* is a particular architecture made specifically to fetch more instructions at a time. The *CPU* issues multiple sets of operations *(single unit of computations, such as `ADD`, `LOAD`, `branch`, ...)* called **instructions**. Those are meant to be intended to be issued at the same time and the compiler has to specify them completely.

Its features include:

- **Fixed number** of instructions *(between* 4 *and* 16*)*
- The instructions are scheduled by the **compiler**
  - the hardware has **very limited control** on what is going on
  - the instructions are going to have a **very low dependency**
- The operations are put into wide **templates**
- **Explicit** parallelism
  - parallelism is found at compile time, not run time
  - the compiler is responsible for parallelizing the code, not the designer
- Single **control flow**
  - there's only one `PC`
  - only one instruction is issued each clock cycle
- Low hardware **complexity**
  - there's no need to to perform *scheduling* or *reordering* on hardware level
  - all operations that are supposed to begin at the same time are packaged into a single instruction
  - each operations slot is meant for a fixed functions
  - constant operation latencies are specified

There are multiple **functional units** *(FUs)* that are going to execute instructions in parallel. An illustration of the inner working instruction level is represented in Figure 23, while the pipeline level is represented in Figure 24.

8.7.1 Compiler responsibilities

The **compiler** has to schedule the instructions *(via **static scheduling**)* to maximize the parallel execution:

- It can exploit *ILP* and *LLP (Loop Level Parallelism)*
- It is necessary to **map the instructions** over the machine functional units
- This mapping must account for **time constraints and dependences** among the tasks themselves

The idea behind the static scheduling in *VLIW* is to utilize all functional units *(FUs)* in each cycle as much as possible to reach a better *ILP* and therefore higher parallel speedups.

[8]FLC potrebbe servire a qualcosa

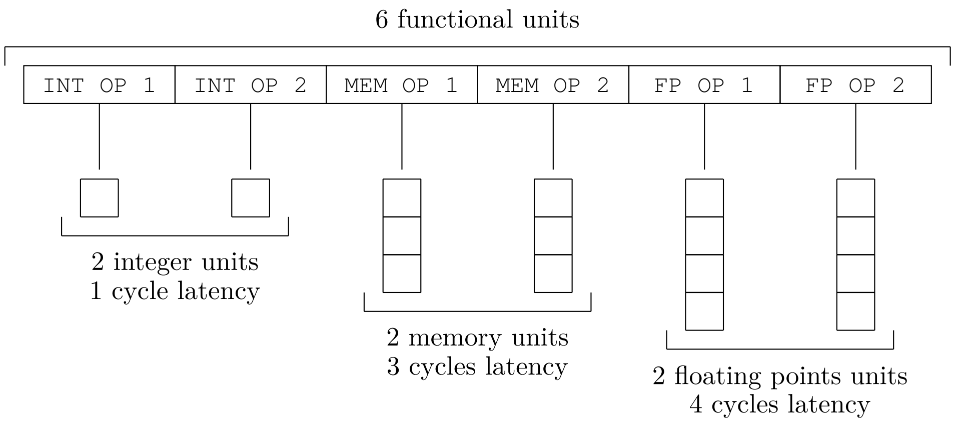


Figure 23: *VLIW* - instructions level

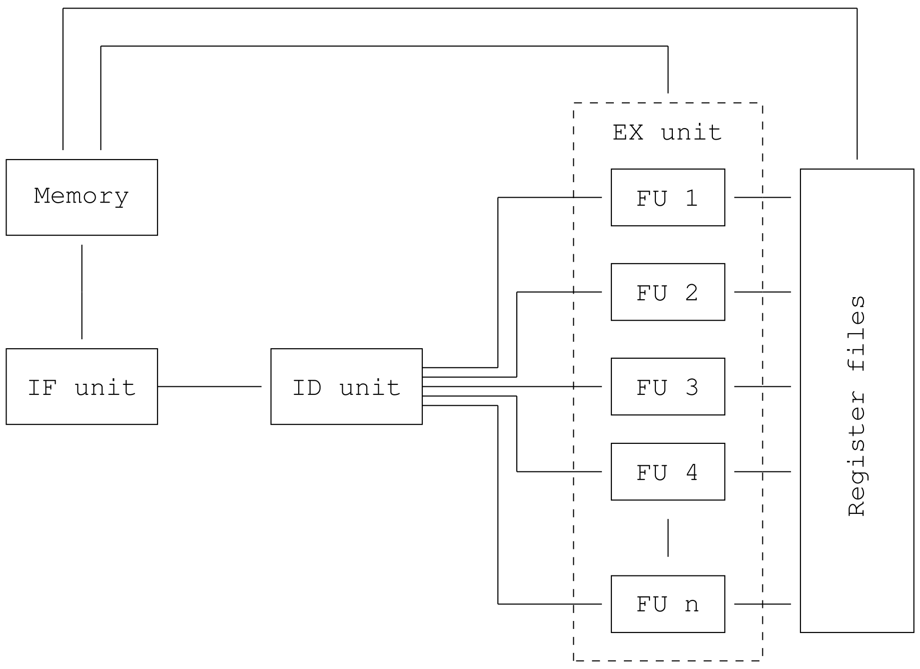


Figure 24: *VLIW* - pipeline level

8.7.2 Basic Blocks and Trace Scheduling

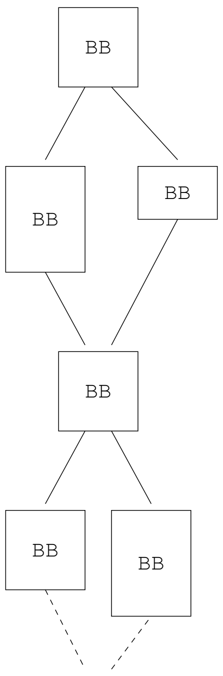
Compilers use sophisticated algorithms to schedule code and exploit *ILP*. However, the amount of parallelism available in a single **basic block**, as previously pointed out, is quite small; furthermore, **data dependence** can limit the amount of *ILP* that can be exploited to less than the average block size.
A **basic block** *(BB)* is defined as a sequence of straight non-branch instructions.
In order to obtain substantial performance enhancements, the *ILP* must be exploited across multiple blocks (*i.e.* among branches). An illustration of the structure of *BB* can be found in Figure 25a.

A **trace** is a sequence of basic blocks embedded in the control flow graph. It must not contain loops but it can include branches.
It's an **execution path** which can be taken for a certain set of inputs. The chances that a trace is executed depend on the input set that allows its execution. As a result, some traces are executed much more frequently than others.
The tracing scheduling algorithm works as follows:

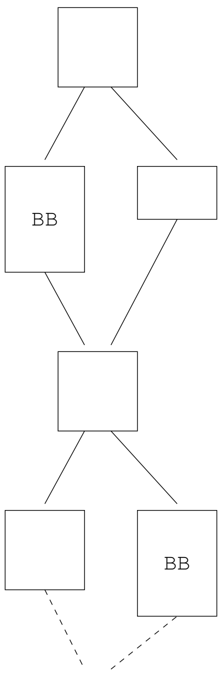
1. Pick a **sequence of basic blocks** that represents the most frequent branch path
2. Use **profiling feedback** or compiler heuristics to find the common branch paths
3. **Schedule** the whole trace at once
4. Add **code to handle branches** jumping out of trace

Scheduling in a trace relies on basic code motion but it could also use globally scoped code by appropriately *renaming* some blocks. Compensation codes are then needed for **side entry points** *(i.e. points except beginning)* and **slide exit points** *(i.e. points except the ending)*.
Blocks on non-common paths may now have added overhead, so there must be a high probability of taking common paths according to the profile. However, this choice might not be clear for some programs.
In general, compensation codes are not easy to generate for entry points.

A comparison of scheduled and unscheduled codes can be found in Figure 25.



(a) Basic blocks



(b) Trace scheduled code

Figure 25: Comparison of scheduled and unscheduled code

8.7.2.1 Code motion and Rotating Register Files in Trace Scheduling

In addition to the need for compensation codes, there are a few more restrictions on the movement of a code trace:

A) The **data flow** of the program must not change
B) The **exception behaviour** must be preserved

In order to ensure (A), the **Data** and **Control** dependency must be maintained. Furthermore, control dependency can be eliminated using **predicate instructions** *(via Hyperblock scheduling)* and branch removal or by using **speculative instructions** *(via Speculative Scheduling)* and speculatively moving instructions before branches.
Finally, Trace Scheduling within loops requires lots of registers, due to the duplicated code: to solve this issue, a new set of registers must be allocated for each iteration.
This solution is achieved via the use of **Rotating Register Files** *(RRB)*. The address of the *RRB* register points to the base of the current register set. The value added onto a local register specifier gives a physical register number.

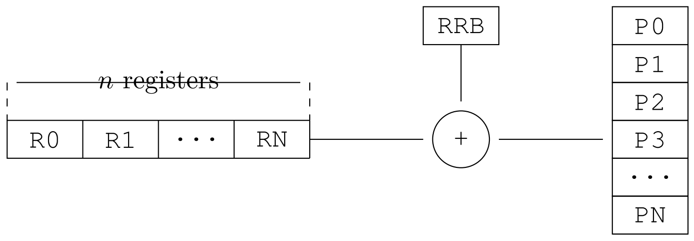


Figure 26: Rotating Register File

8.7.3 Pros and cons of *VLIW*

Pros:

✓ **Simple** *HW*
✓ It's **easy** to increate the number of FU
✓ Good **compilers** can efficiently **detect parallelism**

Cons:

✕ **Huge number of registers** to keep active each FU, each needed to store operands and results
✕ **Large data transport** capabilities between:
  - FUs and register files
  - register files and memory

✕ **High bandwidth** between instruction cache and fetch unit
✕ **Large code** size

8.7.4 Static Scheduling methods

The static scheduling methods used in the *VLIW* are:

- Simple code **motion**
- **Loop unrolling** and loop peeling - *Paragraph 8.7.4.1*
- Software **pipelining** - *Paragraph 8.7.5.1*
- Global code **scheduling** *(across basic blocks)*

- **Trace** scheduling - *Paragraph 8.7.5.2*
- **Superblock** scheduling
- **Hyperblock** scheduling
- **Speculative** Trace scheduling

8.7.4.1 Loop unrolling

8.7.5 Loop-Carried dependency

To properly understand this section we need to catch this type of dependency.
They are really simple, to better understand see this code:

```
for(int i = 1; i< N-1;i++){
    v[i] = v[i-1] + 50;
}
```

In this case we are using `v[i-1]` that is modified in the cycle before of the *i-esimo* cycle[9] if a loop-carried dependecy occurs YOU DON'T HAVE TO APPLY AN LOOP-UNROLLING[10]
Examine this snippet of code:

```
for (int i = 0; i < N; i++)
  B[i] = A[i] + C;
```

the inner loop gets *unrolled* in order to execute 4 iterations at once:

```
for (int i = 0; i < N; i += 4) {
  B[i] = A[i] + C;
  B[i + 1] = A[i + 1] + C;
  B[i + 2] = A[i + 2] + C;
  B[i + 3] = A[i + 3] + C;
}
```

A final clean-up is needed to take care of those values of `N` that are not multiples of the unrolling factor *(4 in this example)*.
This technique has the drawbacks of creating **longer code** and **losing performance** due to the costs of starting and closing each iteration. Furthermore, trace scheduling cannot proceed beyond a loop.
With this technique, we have to be very careful with the index, because this can be catastrophic.

An illustration of the performance improvements can be found in Figure 27.

8.7.5.1 Software pipelining

The programs can be pipelined in order to increase performance and reduce the overall cost of the startup and wind down phases from once per iteration to once per loop.

An illustration of the performance improvements can be found in Figure 28.

8.7.5.2 Trace scheduling

As discussed in Section 8.7.2, Trace Scheduling
To increase the performance in these situations, techniques based on loop unrolling are needed. Traces scheduling schedules traces in order of decreasing probability of being executed. As such, the most frequently executed traces get better schedules.

[9]Semplicemente quando uso un dato fatto in un altro ciclo ho un grosso rischio
[10]Potrebbe servire anche in FLC, perchè se è presente una lista va sempre strotolata

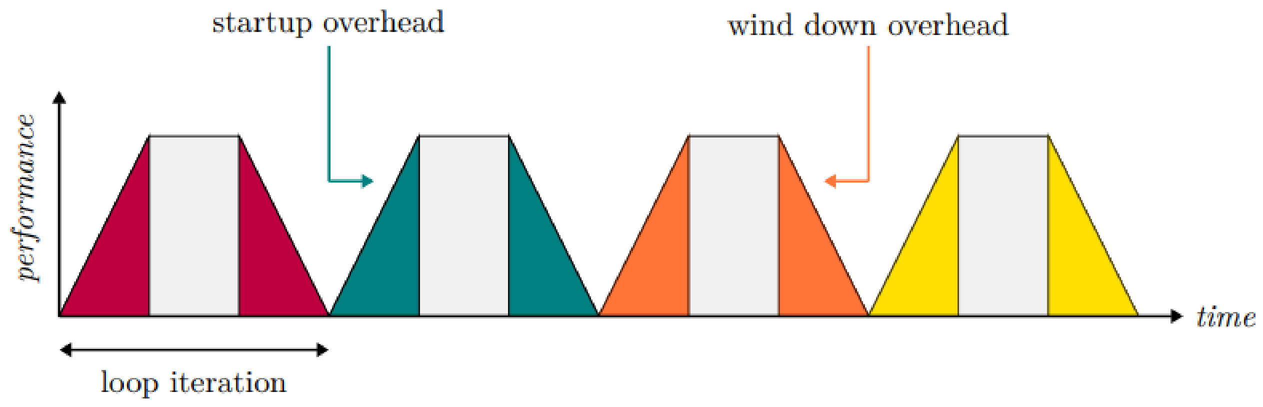


Figure 27: Performance improvement of loop unrolling

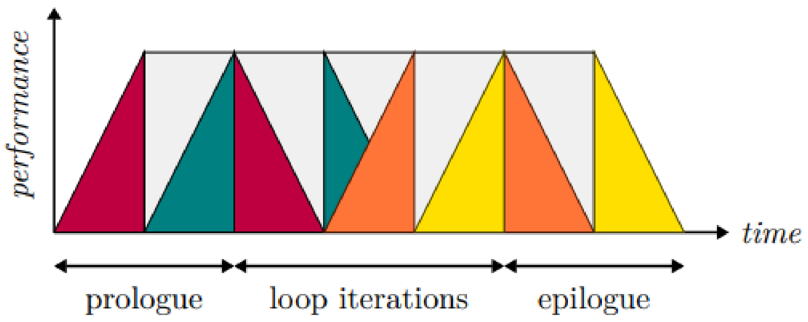


Figure 28: Performance improvement of software pipelining

9 Hardware Based Speculation

The Hardware Based Speculation combines 3 ideas:

- **Dynamic Branch Prediction** to choose which instruction to execute
- **Dynamic Scheduling** to support out-of-order execution while allowing in-order commit
    - prevents irrevocable actions such as register update or exception taking until an instruction commits
- **Speculation** to execute instructions before control dependencies are resolved

The outcome of the branches is speculated, and then the program is executed as if the speculation was correct. Mechanisms are necessary to handle incorrect speculation: the hardware speculation extends dynamic scheduling beyond a branch *(i.e. beyond the basic block)*.

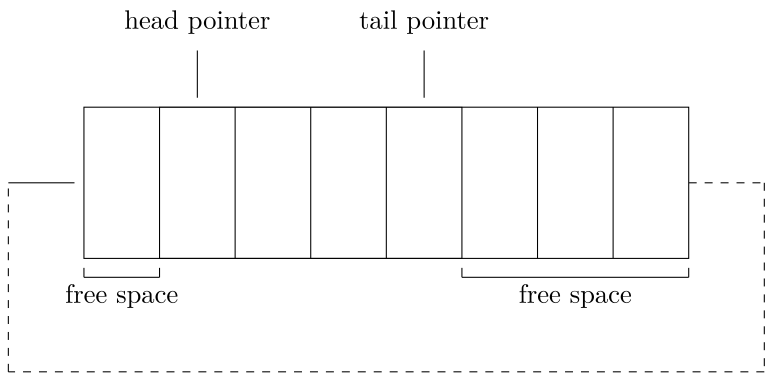
9.1 Reorder Buffer

The **Reorder Buffer** *(ROB)* holds instructions in *FIFO* order, exactly as issued: when the instructions are complete, their results are placed back in the *ROB*.
Operands are supplied to the other instructions between their completion and their commit, while results are tagged with *ROB* buffer number instead of the reservation station.
During the instruction commit, the values in the head of the *ROB* are placed in registers. This guarantee a speculative execution, so we can execute operation before their real time of execution [11], but if the speculation goes wrong? Nothing happen, because the completion state is separed from commit state so we can undo everything.
With this hardware speculation we can obtain an in-order commit with out-of-order execution, manteining a precise interrupt/execption handling

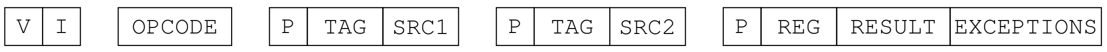
ROB is structured as a circular buffer with **head** and **tail** pointers. Entries between those two are valid and they are allocated and freed whenever an instruction enters or leaves the *ROB*.
Its main entries are:

- The **type** of the instruction
- The **destination** register of the result
- The **result** of the instruction
- If any **exception** was raised
- The **program counter** `PC`
- If the **instruction** is **ready**
- If the **branch** is **speculative**

[11] Ex. un ciclo while, eseguo la prima istruzione del ciclo ma potrei aver raggiunto la condizione per uscire e quindi quell'istruzione non serve, ecco questo è l'esecuzione speculative delle cose.



(a) *ROB* structure



(b) *ROB* entry

Figure 29: Structure of the *ROB* and its entries

The structure of an entry contains:

- The `TAG`, given by the index in the *ROB*
- The new instructions are **dispatched** to free slot
- Instruction space is **reclaimed** when done by setting the `P` bit
- In **dispatch** stage:
  - non-busy source operands are read from register files and copied to `SRC` fields where the `P` bit is set
  - busy source operands copy `TAG` of producer and clear the `P` bit
  - the `V` bit is set valid
- In **issue** stage:
  - the `I` bit is set valid
- On **completion**:
  - source tags are searched and the `P` bit is set
  - result and `EXCEPTION` flags are written back to *ROB*
- On **commit**:
  - `EXCEPTION` flags are checked
  - `RESULT` is copied into register files
- On **trap**:
  - machine and *ROB* are flushed
  - `FREE` pointer is set as `OLDEST`
  - the execution resumes from `HANDLER`

In order to separate the completion stage from the commit stage, the *ROB* must hold register results from the former until the latter.
Each entry, allocated in program order during decode, holds:

- The **type** of the instruction
- The **destination** register specifier and value *(if any)*
- The **program counter**, `PC`
- The **exception** status *(often compressed to save memory)*

They buffer completed values and exception states until the in-order commit point. Completed values can be used by dependencies before reaching that point.

9.1.1 Tomasulo's Algorithm and *ROB*

Tomasulo's Algorithm needs a buffer for all the uncommitted results. It's structured as a *ROB*.
Each entry of the *ROB* contains 4 fields:

1. **Instruction Type** field, indicating if:
    - the instruction is a *branch (and has no destination result)*
    - the instruction is a `STORE` *(and has a memory address destination)*
    - the instruction is a `LOAD` or *ALU* operation *(and has a register destination)*
2. **Destination** field, supplying:
    - the register number *(for `LOAD` and ALU instructions)*
    - the memory address *(for `STORE` instructions)*
3. **Value** field, holding the value of the result until the instruction commits
4. **Ready** field, indicating if the instruction has been completed and the value is ready

- [12]The *ROB* replaces all store buffers, because `STORE` execute in two steps
    - the second step happens when the instruction commits
- The renaming function of the reservation stations is completely replaced by the *ROB*
    - Tomasulo provides **Implicit Registers Renaming**, because user registers are renamed to reservation station tags
- Reservation stations now only queue operations *(and their relative operands)* to *FUs* between the time they issue and the time they begin their execution stage
- Results are tagged with the *ROB* entry number rather than with the *RS* number
    - each *ROB* assigned to instruction must be tracked in the *RS*
- All instructions **excluding incorrectly predicted branches** and **incorrectly speculated `LOAD`** commit when reaching head of *ROB*
    - when an incorrect prediction or speculation is indicated, the *ROB* is flushed and execution restarts at the correct successor of branch
    - speculative actions are easily undone
- Processors with *ROB* can dynamically execute while maintaining a precise interrupt model
    - if instructions *I* causes an interrupt, the *CPU* waits until *I* reaches the head of the *ROB* to handle it, flushing all other pending instructions

9.1.2 Steps in Tomasulo Speculative

The 4 steps of the Speculative Tomasulo Algorithm are:

1. **Issue** or **dispatch** - instruction is loaded from *FP* operation queue. If the reservation station and the reorder buffer slot are free, then:
    - the instruction is issued
    - the operands are sent
    - the destination buffer is reordered
2. **Execution** - operands are operated upon
    - when both operands are ready, the instruction is executed
    - if they aren't, the *Common Data Bus* is watched for the results
    - when both operands are in the reservation station, the instruction is executed
    - *RAW* are checked
3. **Write result**
    - result is written on the *Common Data Bus* to all awaiting *FUs* and *ROBs*
    - all the *RS* are set to available

[12]Esercizi su ROB

4. **Commit** or **graduation** - when the instruction at the head of the *ROB* and the results are present, then:
    - the register *(or memory page)* is updated with the result
    - the instruction is removed from the *ROB*
    - mispredicted branches are flushed from the *ROB*

Furthermore, 3 possible **commit** sequences are possible:

- **Normal commit**:
    - the instruction reaches the head of the *ROB*
    - the result is found in the register
    - the instruction is removed from the *ROB*
- **Store commit**:
    - same as **Normal commit** but memory *(and not a register)* is updated
- **Instruction is a branch with incorrect prediction**:
    - the speculation was wrong
    - *ROB* is flushed *(the operation is called "graduation")*
    - execution restarts at the correct successor of the branch

9.2 Explicit Register Renaming

Explicit Register Renaming is a technique that uses a physical register file that is larger than the number of registers specified by the *ISA*.
The key *idea* behind this technique is to allocate a new physical destination register for every instruction that writes; it's very similar to a compiler transformation called **Static Single Assignment** *(SSA)* but at hardware level. It removes all chances of *WAR* or *WAW* hazards while allowing complete **out of order** completion.
It works by keeping a translation table that maps each *ISA* register to a physical register: whenever a register is written, the corresponding entry on the map is replaced with a new register from the **freelist**. A physical register is considered free when not used by any active instruction.
A simple illustration of this technique is shown in Image 30.

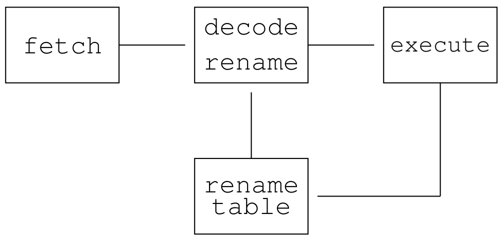


Figure 30: Explicit Register Renaming

Explicit Renaming Support includes:

- **Rapid access** to a table of translations
- A physical register file that has **more registers** than specified by the *ISA*
- Ability to figure out which physical registers are **free**
    - if no register is free, the issue stage is stalled

The advantages of Explicit Renaming Support are:

- Decouples **renaming** from **scheduling**
    - pipeline can behave exactly like *"standard"*, Tomasulo like or scoreboard *(among the others)* pipeline
    - physical register file holds committed and speculative values
    - physical registers are decoupled from rob entries

        - there's no data in the *ROB*
- Allows data to be fetched from **a single** register file
    - there's no need to bypass values from reorder buffer
    - this can be important for balancing the pipeline

9.2.1 Multiple Issue

Often it's necessary to modify the issue logic to handle two or more instructions at once, including possible dependencies between instructions. The biggest bottleneck is found in dynamically scheduled superscalar processors:

- The **processor needs additional logic** to handle the issue of every possible combination of dependent instructions in the same clock cycle
- The **number of combinations increases with the square with respect to the number of instructions** that can be issued in the same cycle
    - it's difficult to implement a logic supporting more than 4 instructions

The basic approach is as follows:

1. A **reservation station** and a *ROB* entry is assigned to each instruction in the following issue bundle
    - if this is not achievable. only a subset of instruction is considered in sequential order
2. All **dependences** between the instructions are **analyzed**
3. If an instruction in the bundle **depends on an earlier instruction of the same bundle**, the assigned *ROB* number is used to update the reservation table for the current instruction

All these operations are done in parallel in a single clock cycle.
Similarly, the issue logic must behave as follows:

1. Enough physical space for the entire issue bundle is **reserved**
2. The issue logic determines what **dependences exists in the bundle**:
    - if a dependence **does not exist** within the bundle:
        - the register renaming structure is used to determine the physical register that holds the result on which instruction depends
        - the result is from an earlier issued bundle and the register renaming table will contain the correct register number
        - if an instruction depends on an **instruction that is earlier** in the bundle:
            - the reserved physical register in which the result will be placed is used to update the information for the issuing instruction

Once again, all these operations are done in parallel in a single clock cycle.

10 Multithreading

The parallel architecture aims to replicate processors to add performance, rather than designing a faster processor; it extends the traditional architecture by adding a communication layer between processors. New abstractions *(for both hardware and software interfaces)* and different structures to realize them are needed.

ILP architectures *(like superscalar and VLIW)* support fine-grained, instruction level, parallelism but they fail to support large-scale parallel systems. Multiple-issue CPUs are very complex and thus extracting more parallelism is getting more and more difficult as time goes on due to a steep increase in the hardware complexity. A partial solution can be found by extracting parallelism at higher levels: multiple microprocessors are connected in a complex system, extracting parallelism at the process and thread levels. Parallelism can be achieved by:

- **Data Level** Parallelism - *DLP*
  - many data items can be processed at the same time
- **Task Level** Parallelism - *TLP*
  - tasks can be executed in parallel and independently

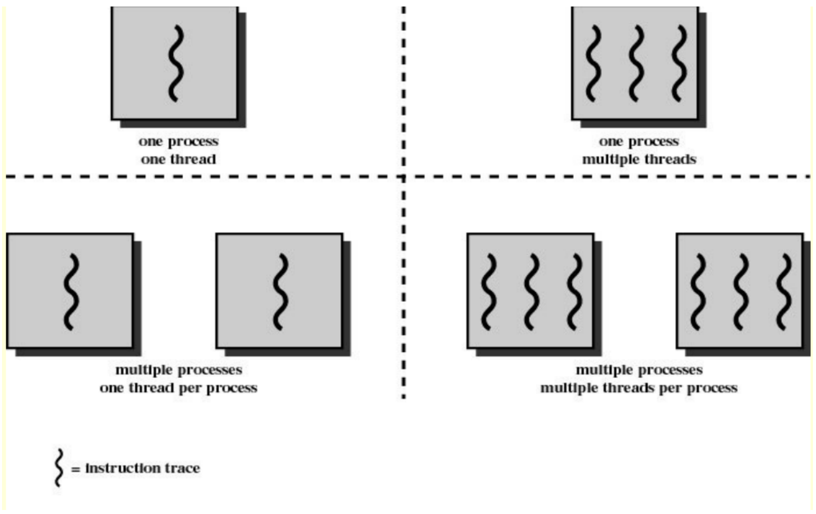


Figure 31: Molteplicity process/thread

10.1 Thread

Thread are created by the programmer or by the OS, this would be for a portion of the code that could be parallelize[13], this need a complication in HW, because every thread need his own Program counter and register file, but not the functional unit of the processor[14]. This allows multiple threads to share the functional unit to improve the throughput of the CPU. The memory address space can be shared too, but we will see that this will need coherence logic.

[13]Ex. un ciclo for senza loop-carried dependency
[14]ATTENZIONE ALLA DIFFERENZA TRA THREAD HW E THREAD SW

10.2 Types of multithreading

1. Coarse-grained Multithreading This will hide long stall caused by memory miss.

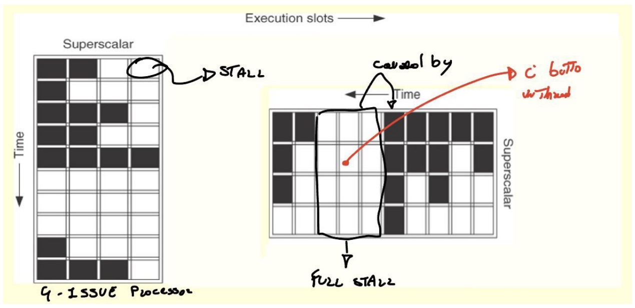


Figure 32: We are putting a new thread while the T1 is full stalled

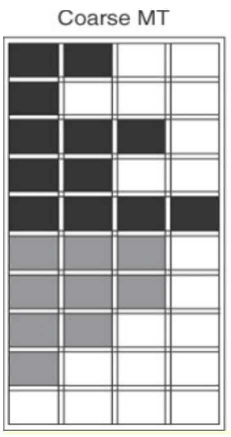


Figure 33: Coarse

2. Fine-Grained, this type of Multithreading check each instruction if one thread is stalled in a round-robin way, skipping stalled thread trying to eliminate empty slots. this is an effective way to hide long-latency

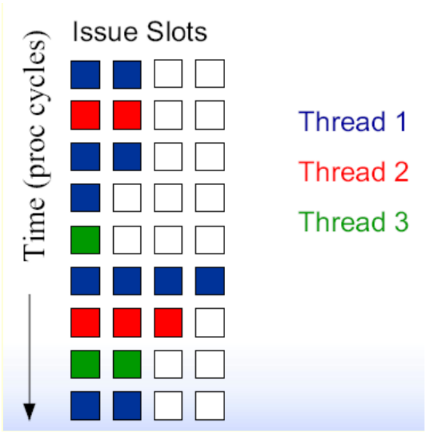


Figure 34: Fine-Grained MT

event and keep CPU busy. In this method, we have to discuss the difference between SW Thread and HW Thread.

- SW Thread, thisispe of thread is managed from the OS with a context switch and is not physically in the core.
- HW Thread are physically on the chip and they are not managed from OS

So I.E. every HW thread can handle multiple SW threads.

3. Simultaneous Multithreading is the more complex type This technique comes naturally when we use

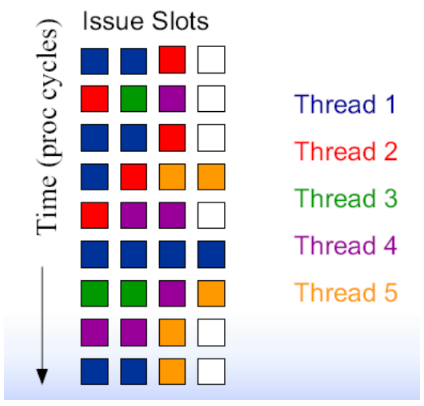


Figure 35: Simultaneous Threading

dynamic scheduling and multiple issue processor. With this, we have a sizeable scalable processor that can improve a lot the performance with regards to the other component this improve the performance without improving a lot the power consumption.

11 Multiprocessor

Right now many processors are running multi-core solutions, so this topic is very important to understand the state of the art. We are entering ***MIMD*** architecture, in fact, every processor has its own data and instruction, and every processor could have multithreading or not.

11.1 Connection of processors

By increasing the number of processors we can see that will be an increase in cost. There are 2 main connections:

1. Single bus, mostly used with a low number of processors
2. Interconnection networks, mostly used with a high number of processor

11.1.1 Single bus multiprocessor

It's an evolution of *Von Neumann* architecture that contains multiple nodes Each node has its own cache and

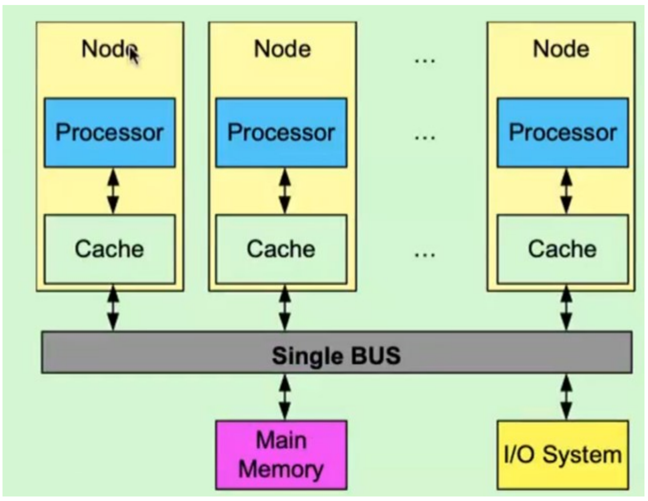


Figure 36: Single bus

processor.
We have a single bus so if one node need to send a communication the bus will be occupied and no more node can communicate in the same bus.

11.1.2 Network-connected

In this case, we have a memory attached to the processor and the connection could be a single bus, but in the most of the case we will get a more complex network.

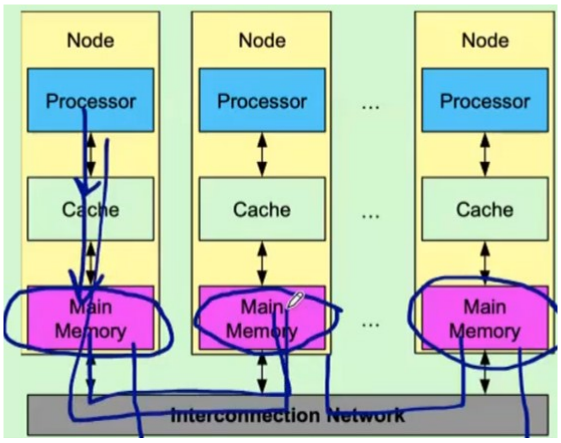


Figure 37: Network connected

11.2 Network Topologies

There are multiple topologies

- Single-bus

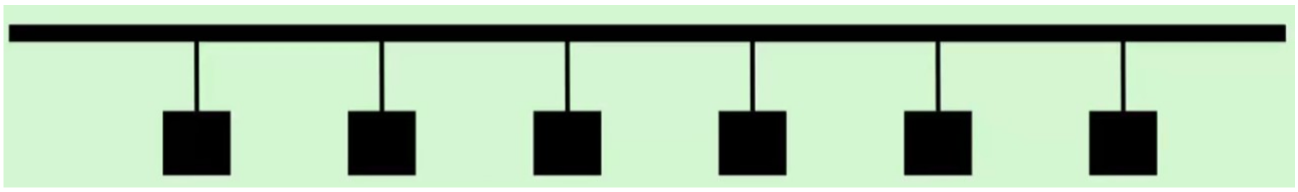


Figure 38: Single bus topology

- Ring, this topology can do simultaneous transfers. The communication between nodes needs to pass through intermediate nodes to reach the final destination.

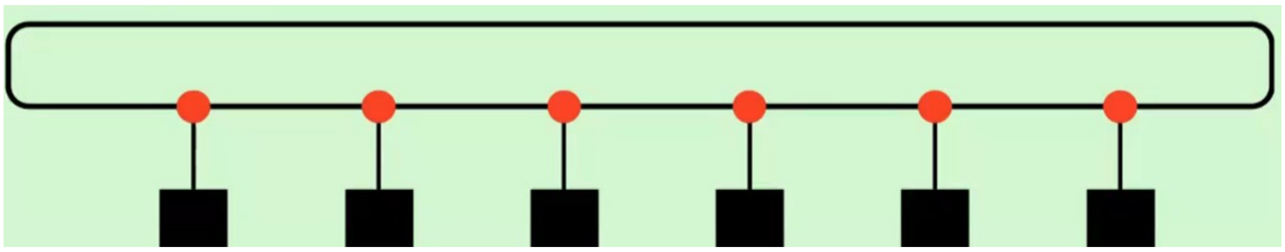


Figure 39: Ring topology

11.2.1 Network Performance Metrics

To measure the performance we need to introduce a formula:
P = number of nodes
M = number of links
b = bandwidth of a single link
That in the best case will be:

$$M * b$$

so the number of links is multiplied by the bandwidth of each link. But the world unfortunately is not in the best scenario, so we have to consider the worst. This is calculated by cutting in an half the nodes[15]

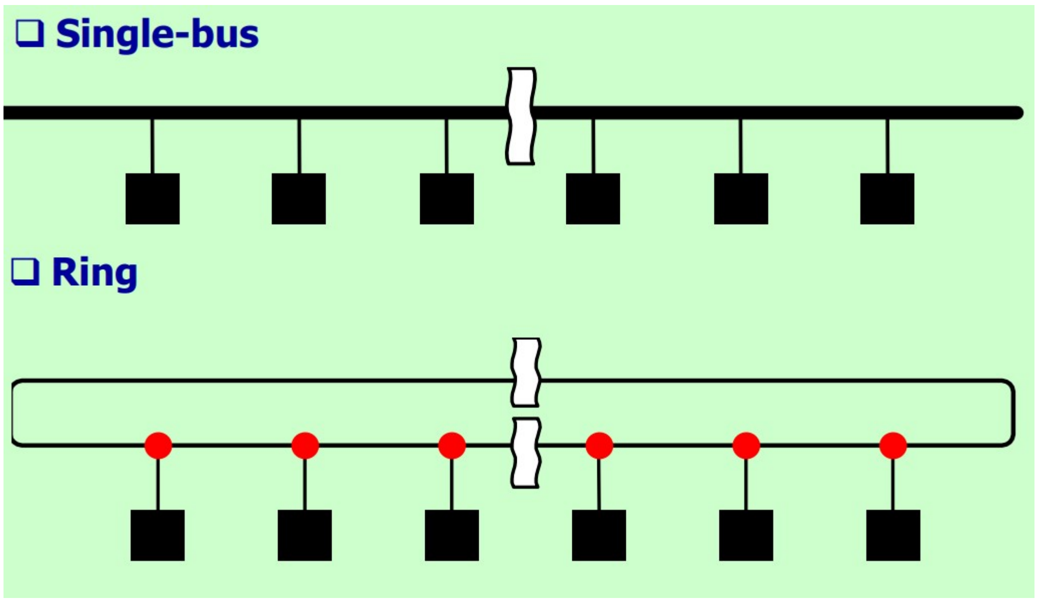


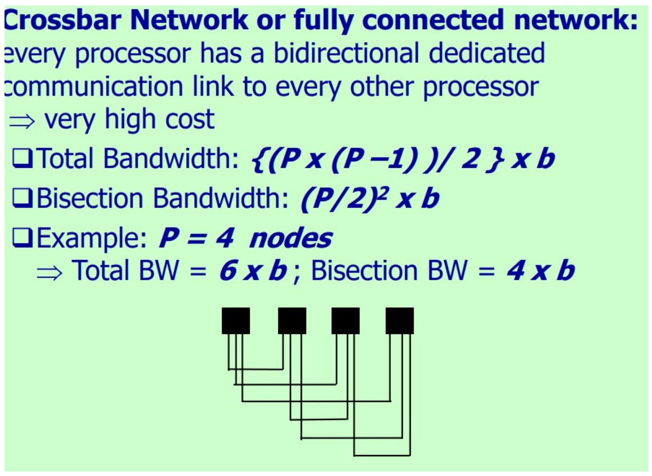
Figure 40: Bisection

So we will get a situation more or less like this

	Total Bandwidth	Bisection Bandwidth
Single Bus	b	b
Ring	Pb	$2b$

Let's now continue the network connection:

- Crossbar Network

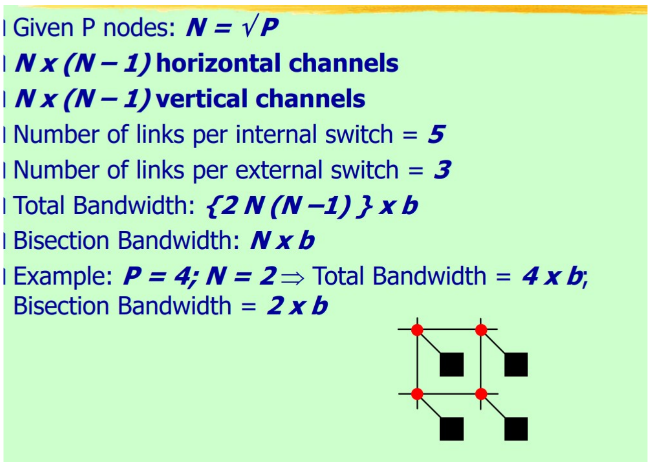


Crossbar Network or fully connected network:
every processor has a bidirectional dedicated communication link to every other processor
⇒ very high cost
❑ Total Bandwidth: $\{(P \times (P-1))/2\} \times b$
❑ Bisection Bandwidth: $(P/2)^2 \times b$
❑ Example: $P = 4$ nodes
⇒ Total BW = $6 \times b$; Bisection BW = $4 \times b$

Figure 41: Crossbar

[15] i quadrati neri

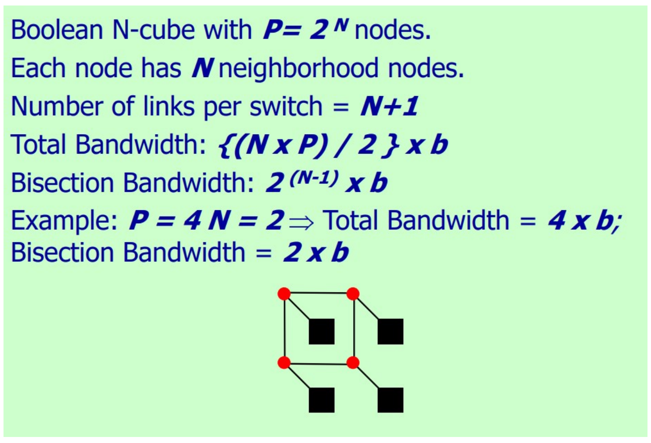
- Bidimensional Mesh



- Given P nodes: $N = \sqrt{P}$
- $N \times (N-1)$ **horizontal channels**
- $N \times (N-1)$ **vertical channels**
- Number of links per internal switch = **5**
- Number of links per external switch = **3**
- Total Bandwidth: $\{2N(N-1)\} \times b$
- Bisection Bandwidth: $N \times b$
- Example: $P = 4; N = 2 \Rightarrow$ Total Bandwidth = $4 \times b$; Bisection Bandwidth = $2 \times b$

Figure 42: Mesh

- HyperCube



Boolean N-cube with $P = 2^N$ nodes.

Each node has N neighborhood nodes.

Number of links per switch = $N+1$

Total Bandwidth: $\{(N \times P) / 2\} \times b$

Bisection Bandwidth: $2^{(N-1)} \times b$

Example: $P = 4$ $N = 2 \Rightarrow$ Total Bandwidth = $4 \times b$; Bisection Bandwidth = $2 \times b$

Figure 43: HyperCube

11.3 Shared of Data

We are in a bivio, we can choos two type with his pros and cons.

- Single logically shared address space, with this idea any processor can make a reference to any memory through loads/stores instruction. In this case, one address is the same for every processor in the chip.

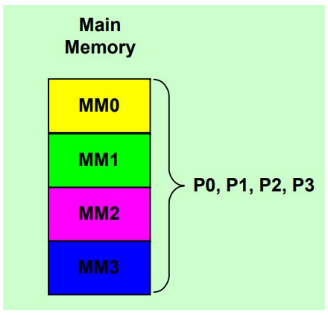


Figure 44: Single logically shared address

This model imposes the ***Cache Coherence problem*** among processors. To be clear shared memory does not mean that there is a single centralized memory, it can be distributed all over the chip in all the processor[16].
It's also called *OpenMPI*.

- Multiple address spaces, in this idea every processor has its own bank memory, and the communication among those must be through send/receive message with an ***Message Passing Architectures***, the address space is logically disjointed so 2 same addresses in 2 different processors refer to a different memory location.

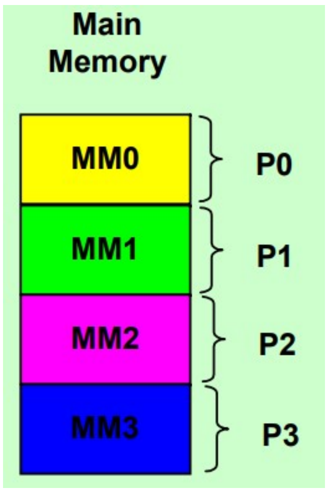


Figure 45: Multiple Address

This method will not cause any problem with cache coherency. It's also called *MPI*

WE ARE LOOKING JUST AT MEMORY AND FROM A LOGICAL POINT OF VIEW NOT THEIR PHYSICAL POINT OF VIEW

[16]ocio che è una domanda

11.4 Physical memory organization

How is placed this memory?
We can have two types of organization

1. Centralized Memory (UMA), in this organization the access time to a memory location is ***uniform*** for all the processors.

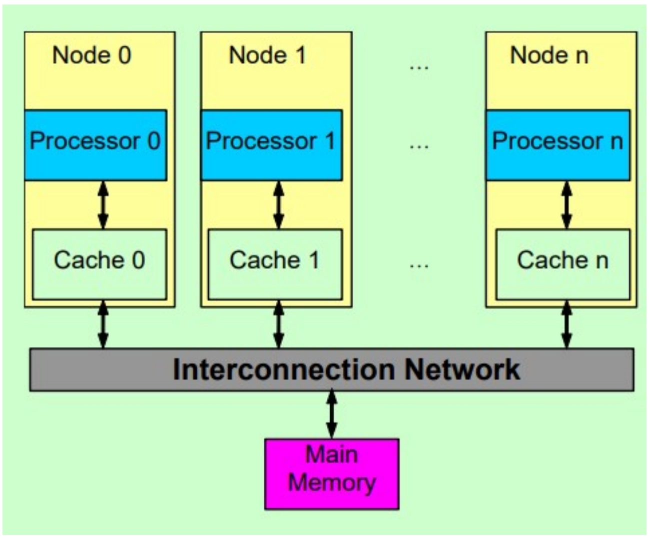


Figure 46: UMA

2. Distributed Memory (NUMA), the memory is physically divided into modules distributed on every single processor, and the memory location access time ***IS NOT UNIFORM*** for all processors.

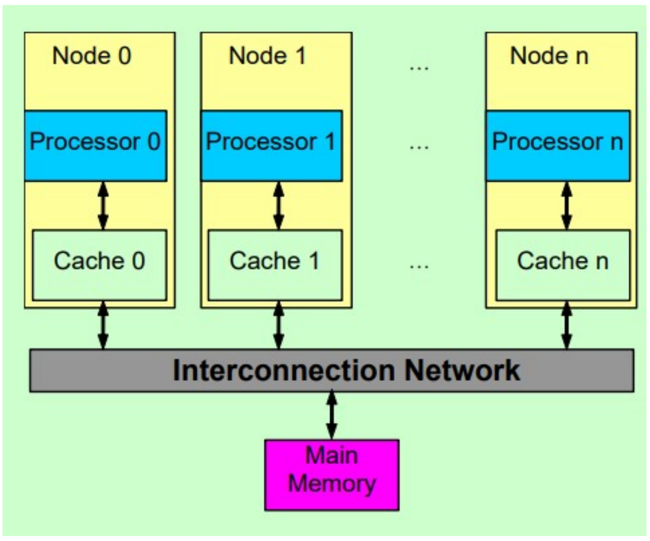


Figure 47: NUMA

The concept of address space and the physical memory organization are *orthogonal* to each other, so I can create all 4 options:

1. UMA / single address

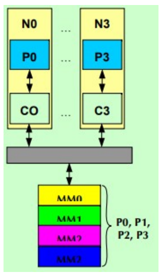


Figure 48: UMA / single address

2. NUMA / Single Address

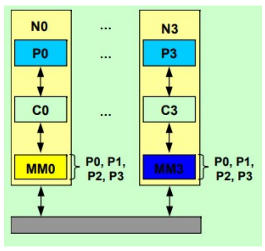


Figure 49: NUMA / Single Address

3. UMA / Multiple address

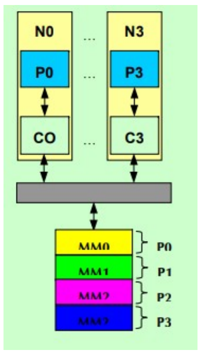


Figure 50: UMA / Multiple addresses

4. NUMA / Multiple address

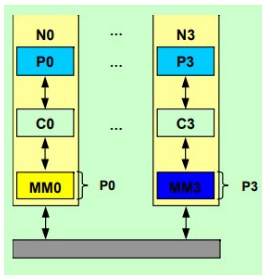


Figure 51: NUMA / Multiple address

11.4.1 Centralized and shared

Most of the existing multicores are single-chip with a small number of nodes with a single bus and Centralized shared address memory.
As we said in the shared address memory we have some problems on cache coherency, try to imagine a solution that will be discussed later.

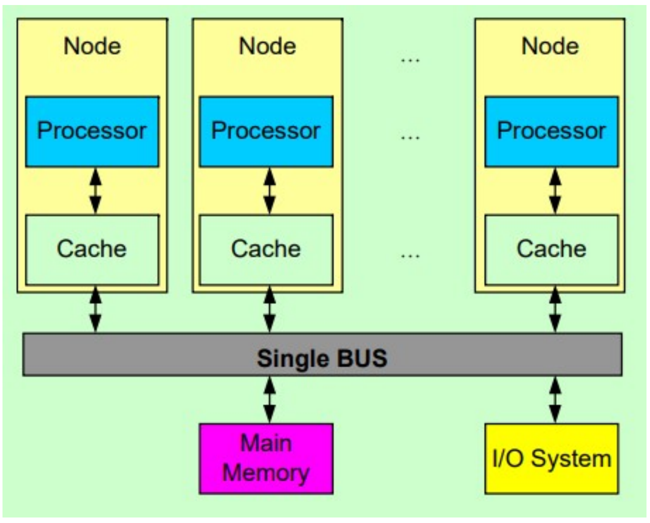


Figure 52: UMA / Single address

The problem is we have multiple caches, and I've potentially multiple copies of my data, if we are just reading there is no problem, but if someone has a beautiful idea to write?
I have a problem keeping coherency to the main memory and the other cache, we already saw a solution with cache and main memory, now we have to expand *Write Through* and *Write Back* to the horizontal problem.

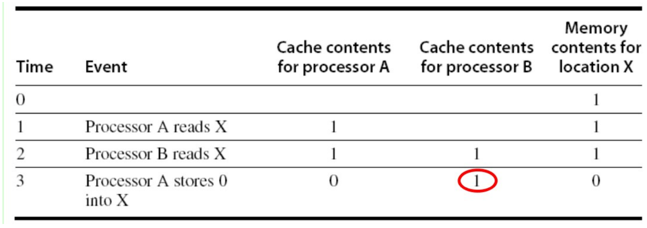
11.5 Programing a multiprocessor

When we are very skilled programmers, we think that with a programming language, we can exploit the world, but not every time is like this, in fact, if we are going to program a multiprocessor architecture we need to do a checklist and thinks first about our architecture how is made and after how made our program.
In fact, we have to manage a programming model because we will get multiple jobs and programs. We have to understand which type of communication between the node is present so if we have to use a shared address space (OpenMP) or a message passing protocol. In the end, is possible to change the paradigm that uses *GPU or Vector processor*[17]to get a single instruction in multiple data.

[17]ne parliamo dopo

12 Cache Coherency

This is a problem caused by shared cache in a multiprocessor architecture, we already know how to fix cache coherency in a vertical way[18], but now we discuss how to keep it coherence with all the nodes among a chip.
When shared data is cached, the corresponding value may be replicated in multiple caches. While reducing the access latency *(and the relative cost of memory bandwidth)*, this replication provides a reduction of shared data contention read by multiple processors simultaneously The use of multiple copies of the same data introduces the aforementioned problem of cache coherency.
Consider the following sequence of operations performed by a *shared data SIMD* with 2 processors:



Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Figure 53: Cache coherency problem

What does B read?
The two processors see different values for the same shared variable.
We need to understand how to keep the coherent and consistent the data, there exist 2 main methods:

1. Snooping protocols
2. Directory based protocols

12.1 Snooping protocols

All cache controllers monitor *(or snoop)* the BUS to determine whether they have a copy of the data requested in the block or not. Every cache that has a copy of the shared data also has a copy of the sharing status of the block, removing the need of keeping a centralized state.
All requests for shared data are sent to all processors. This solution requires broadcast since the cache controllers must know the sharing status of all the data blocks; as such, it's particularly suitable for **Centralized Shared Memory Architectures**, especially for small-scale multiprocessors with single BUS.

- The cache controller **snoops all transactions** on the shared BUS
    - the BUS is merely a broadcast medium
- If a block contains the **address tag** of a variable contained in the cache, then:
    - the processor must *take action*, either **invalidating**, **updating** or **supplying** the value
    - the action depends on the **state** of the block and on the protocol

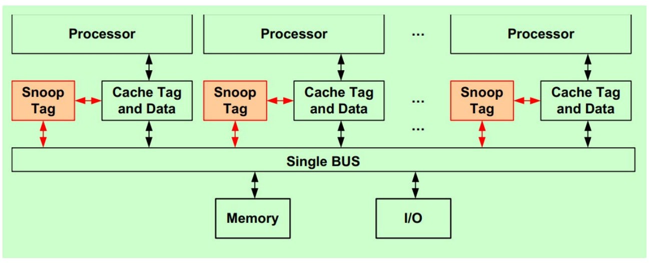


Figure 54: Snooping architecture

Since every BUS transaction checks the cache address tags, this operation can interfere with the processor operations, causing stalls when the variable is not available in the cache.

[18]dalla cache alla main memory

To reduce the number of interferences with the processor's accesses to the cache, the tag portion of the address is duplicated for snooping activities. An extra read port is also added to the address tag portion of the cache.

When a **miss** happens, the following actions are performed:

- In case of a **write** operation:
    1. the address is invalidated in all other caches before the write is performed
    2. this process is called **Write-Invalidate Protocol**
- In case of a **read** operation:
    1. if a dirty copy is found in some cache, a write-back is performed before the memory is read
    2. this process is called **Write-Update Protocol** or **Write-Broadcast Protocol**

12.1.1 Write-Invalidate Protocol

The writing processor issues an invalidation signal over the `BUS` to cause all copies in other caches to be invalidated before changing its local copy: at this point, it is free to update the local data until another processor asks for it.
All caches on the `BUS` check to see if they have a copy of the data and if so, they must invalidate the block containing it. This scheme allows multiple **readers but** only a **single writer**: the `BUS` is used only on the first write to invalidate all other copies while subsequent writes do not result in `BUS` activity.

Different operations of the protocol:

- Basic **Bus Based** Protocol
    - each processor keeps track of cache and state
    - all transactions over `BUS` are snooped
- Writes **invalidate** all other caches
    - multiple readers can read the same block
    - a single `write` invalidates all other copies
- Each block in cache has **two states**
    - the state of a block is a p-vector of states
    - hardware state bits are associated with blocks that are in the cache
    - other blocks can be seen as being in *invalid* in that cache

Finally, on a `WRITE` operation, all other copies of the same data are **invalidated**. The `BUS` itself is used to serialize the access to the data, as the `WRITE` operations cannot complete until exclusive `BUS` access is obtained. This protocol is used with *Write-back* and reduces the bus traffic.

12.1.2 Write-Update Protocol

In the **Write-Update protocol**, when a `WRITE` operation is performed, all the redundant copies of the data are **invalidated**. The processor that performs the write also has the duty of updating the main memory and all the other processors' memory, by broadcasting the new value over the `BUS`. All caches check if they have a copy of the data and if so, all copies are updated with the new value. This scheme requires the continuous broadcast of all `WRITE` operations to the shared data.
This protocol is similar to the **Write-Through** because all writes are sent over the `BUS` to update copies of the shared data, but it has the advantage of making the new values appear in caches sooner *(thus reducing the latency)*. In case of a `READ` miss, the memory is always up to date. It's used with *Write-thought*.

12.1.2.1 Write-through vs Write-back

The main difference between the two protocols is:

- **Write-through**: the memory is always up to date
- **Write-back**: the caches must be snooped into until the most recent copy is found

The write-through protocol is simple, as every `WRITE` operation is observable: each one of them goes on the `BUS`, and as such only one `WRITE` can take place at a time in any processor. As a downside, it uses a lot of bandwidth.

12.1.3 Invalidate vs Update

Before choosing one of the two protocols, a basic question on the program behaviour must be asked:

> *"Is a block written by one processor later read by others before it is overwritten?"*

Then, the two protocols can be compared:

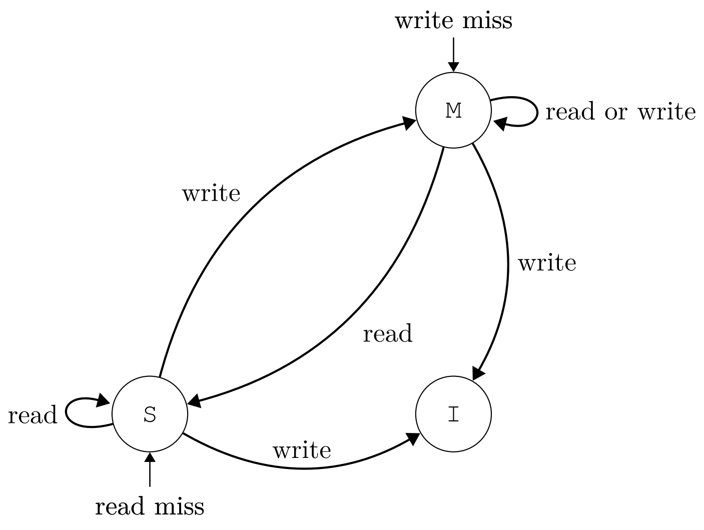
- **Invalidate**
  - ✔ yes: readers will take a miss
  - ✖ no: multiple writes can be performed without added traffic and old copies will be cleared out
- **Update**
  - ✔ yes: misses on later references will be avoided
  - ✖ no: multiple useless updates will be broadcast over the `BUS`

In the same way, most commercial cache-based multiprocessors use:

- **Write-Back Caches** to reduce `BUS` traffic, allowing more processors on a single `BUS`
- **Write-Invalidate Protocol** to preserve `BUS` bandwidth

12.1.3.1 Cache State Transition Diagram

This transition diagram *(represented in Figure 55a)* is applicable to the **MSI Protocol** *(Modified, Shared, Invalid)*. Each cache line has state bits, relative to the state of the data.



(a) State Transition Diagram

M	S	I	ADDRESS TAG

(b) Structure of a cache line

Figure 55: *MSI* Protocol Implementation

12.1.4 MESI Protocol

The basic *MSI protocol* is not completely suitable to be applied to the real world, since it has a few limitations:

- Operations are not **atomic**
  - *deadlocks* and *race conditions* may be introduced
  - **solution**: the processor sends invalidate can hold `BUS` until other processors receive the message
- The system is **hard to extend**
  - **solution**: add an exclusive state to indicate a clean block in only one cache *(MESI Protocol)*

A modified version of the *MSI* protocol is the *MESI* protocol. It implements the Write-Invalidate protocol, with the additional **shared** state.
Each cache block can then be in one out of 4 states:

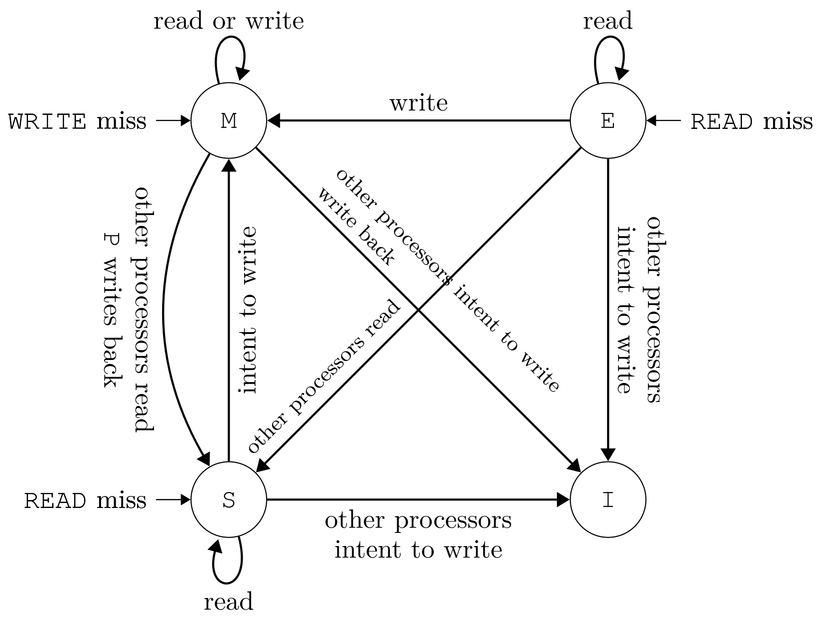
1. **Modified**: the block is dirty and cannot be shared, the cache has the only copy and it's writeable
2. **Exclusive**: the block is clean and the only copy is in the cache
3. **Shared**: the block is clean and other copies of the block are in the cache
4. **Invalid**: the cache contains invalid data

The **exclusive** state distinguishes between **exclusive** *(writable)* and **owned** *(written)* states.
To support this protocol, all cache controllers snoop on a special `BUS` called `BusRd`. The issuer chooses between the **shared** and **exclusive** states, and the `BUS` is used to send the message to the other caches.
Characteristics of the states:

- In both **shared** and **exclusive** states the memory has an up-to-date version of the data
- A write to an **exclusive** block does not require sending the invalidation signal on the `BUS`, since no other copies of the block are in the cache
- A write to a **shared** block implies the invalidation of the other copies of the block in the cache

↱	*Modified*	*Exclusive*	*Shared*	*Invalid*
Line valid?	✓	✓	✓	✗
Copy in memory	has to be updated	valid	valid	invalid
Other copies in other caches?	✗	✗	maybe	–
Can be written?	✓	✓	✗	✓
Can be read?	✓	✓	✓	✗
On read	nothing happens	nothing happens	nothing happens	transition to *Exclusive* if data **is not** on other caches; transition to *Shared* if data **is** on other caches
On write	nothing happens	transition to *Modified*	transition to *Modified*	transition to *Modified*, other copies transition to *Invalidate*

Table 3: State of cache lines with *MESI* Protocol



(a) State Transition Diagram

M	E	S	I	ADDRESS TAG

(b) Structure of a cache line

Figure 56: *MESI* Protocol Implementation

12.2 Directory based protocol

Here the sharing state of a block of physical memory is kept in just one location called *Directory*. Each entry in the directory is associated to each block in the main memory. This protocol is mainly used in ***Distribute shared-memory*** architecture, where the directory is distributed on the nodes to avoid bottleneck.
This does not mean that if you use ***Centralized shared-memory*** architecture you cannot use this protocol. There is an overall mapping of each memory block address to each node, so everyone will know the home node, the node where reside the memory block, and the directory[19].
Everyone in the system knows who refers if need a specific block. Because is a distributed protocol it will use the Message Passing Protocol paradigm, so the protocol is based on send/receive instruction and this will improve the scalability of the system.

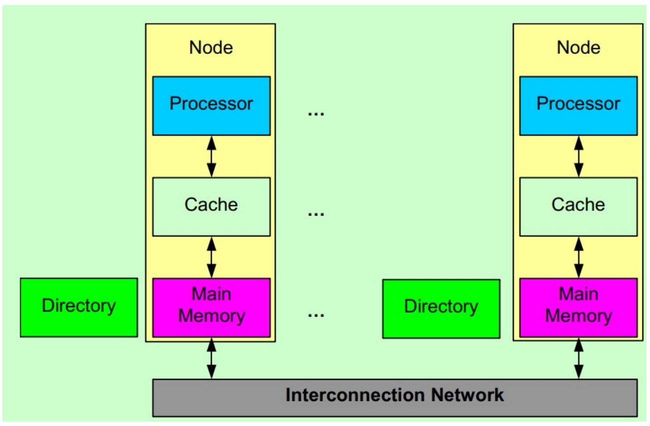
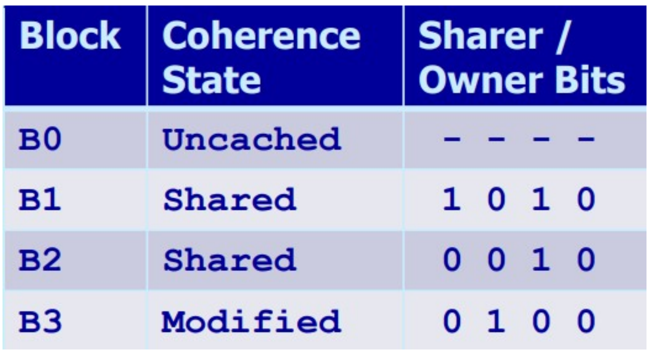


Figure 57: Directory Based protocol

12.2.1 Directory

The directory is the part of the protocol that contains the state and the information of the cache block. The state of a directory could be:

- Uncached, that means nobody has fetched this data
- Shared, there are one or more copies around the chip. How holds the copy is determined by the sharer bit, a series of bit that holds a flag that says ***"This node has this block updated"***
- Modified, there is only one copy around the chip, and the owner, who modified the block, is specified in the sharer bit



Block	Coherence State	Sharer / Owner Bits
B0	Uncached	- - - -
B1	Shared	1 0 1 0
B2	Shared	0 0 1 0
B3	Modified	0 1 0 0

Figure 58: In this picture we can see what's explained before.

[19]dopo ne parliamo

12.2.1.1 Types of Nodes

There are 3 types of nodes:

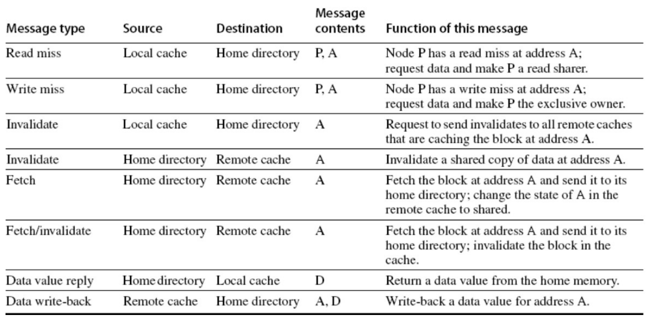
- Home Node, is the one that holds the memory location.
- Local Node, where a request came from.
- Remote Node, where there is a copy of the cache block.

The *Local Node* can be *Home Node* and vice-versa, but this will be an intranode[20] transaction.
The *Remote Node* can be the *Home Node* and vice-versa.
The *Local Node* and the *Remote Node* **MUST BE DIFFERENT**.

12.2.2 How do they talk to each other?

As mentioned before they talk with the message, but suppose we are referring to figure 58 and we are *Node2* we want to read *B3*, what we are supposed to do?
We must contact the *Home node*, by sending a specific message and he will handle all the stuff.

12.2.2.1 Type of Message

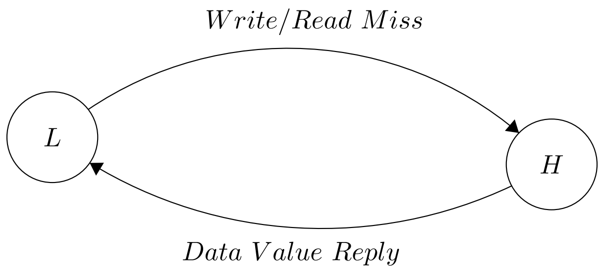


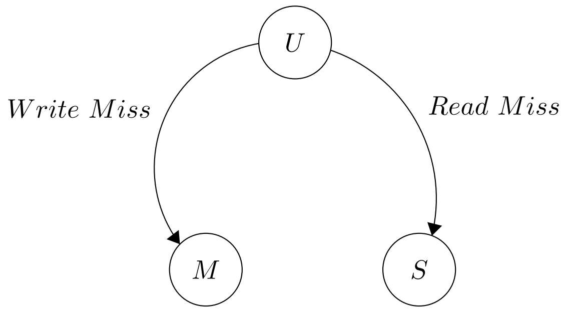
Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

Figure 59: All types of message

12.2.2.2 Uncached

When a directory block is in the uncached block we can only have a miss, no matter if we write or read.
So in this case we will send an *Write/Read Miss* from Local to Home, and it should answer with a *Data Value Reply.*

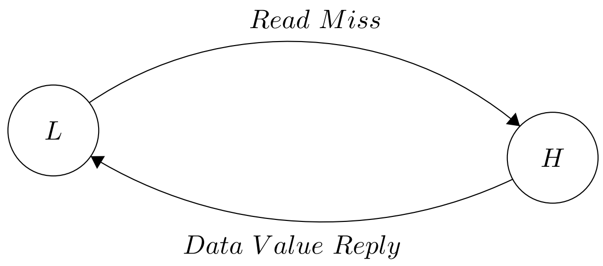




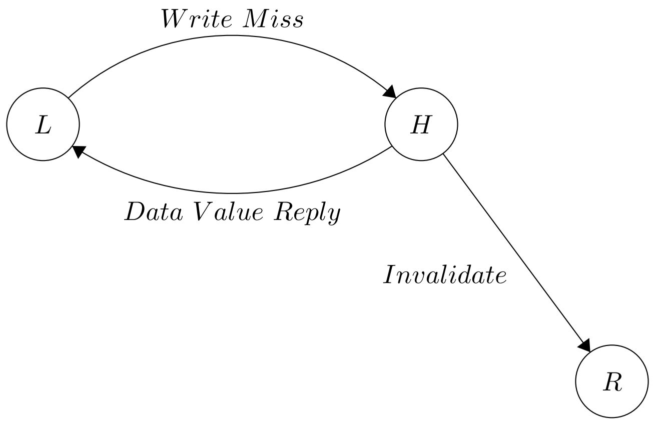
[20]Su se stessa

12.2.2.3 Shared

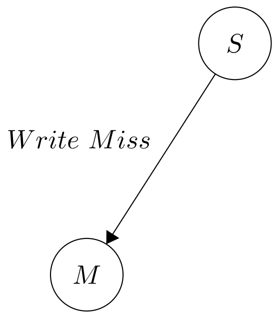
In this case, there will be a difference between read and write.
If we perform a read miss we will get an data value reply from *Home Node*



And the state will not be modified.
The symphony changes if we perform a *Write Miss*, in fact, the home node must invalidate all the copies in the system.

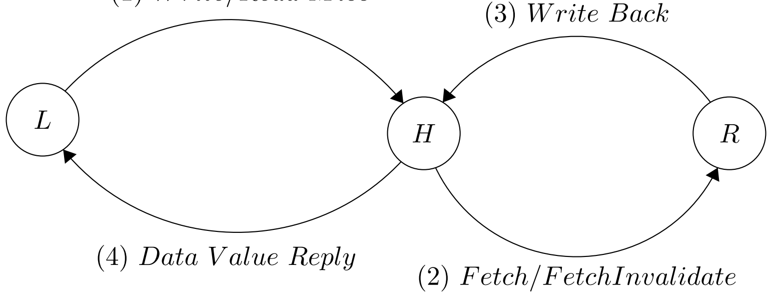


And the state will change as follows:

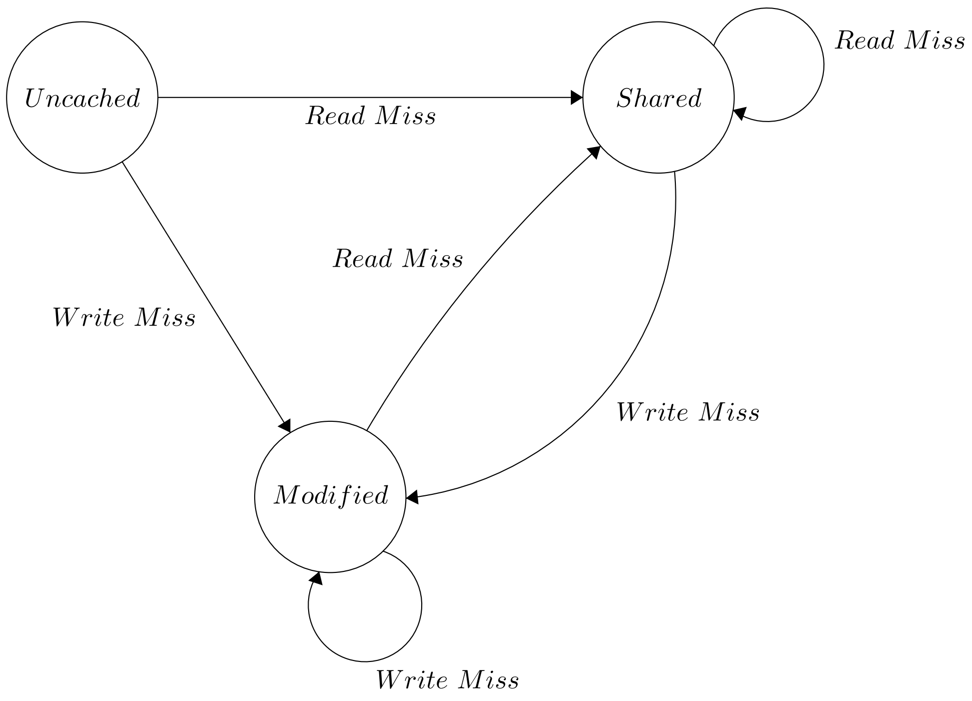


12.2.2.4 Modified

In this state, we have modified data so every time someone performs an action on this data the Home node must fetch it. But there is still a difference between write and read. In fact the first will also invalidate after the fetch.



12.2.2.5 Final FSM of the directory based



13 Vector Processor

Vector processors have high-level operations that work on linear arrays of number *(also called vectors).* A language that can handle vectors *(and not scalar values)* is needed as well.
Basic idea:

- **Read** sets of data elements into *vector registers*
- **Operate** on those registers
- **Disperse** the results back into memory
- Adaptable to **different data types**
  - a vector size can be seen as 64 64-bit elements, 32 128-bit elements, . . . This is because we will adapt our architecture to the application's size.
- A single instruction operates on **vectors** of data
  - the result involves many registers to register operations
  - used to hide memory latency
  - leverages memory bandwidth

In this type of architecture, we have a radical change, now we have a controller of the architecture, but the processing element is distributed and they can access some data: The use of this paradigm is very useful for

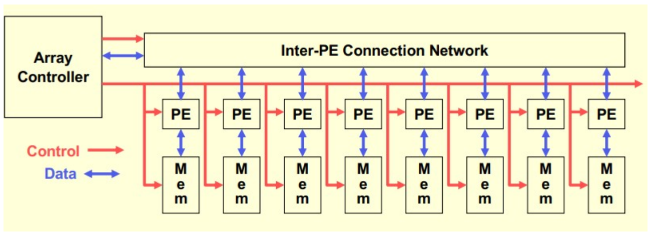


Figure 60: SIMD Architecture

exploiting significant Data level parallelism.

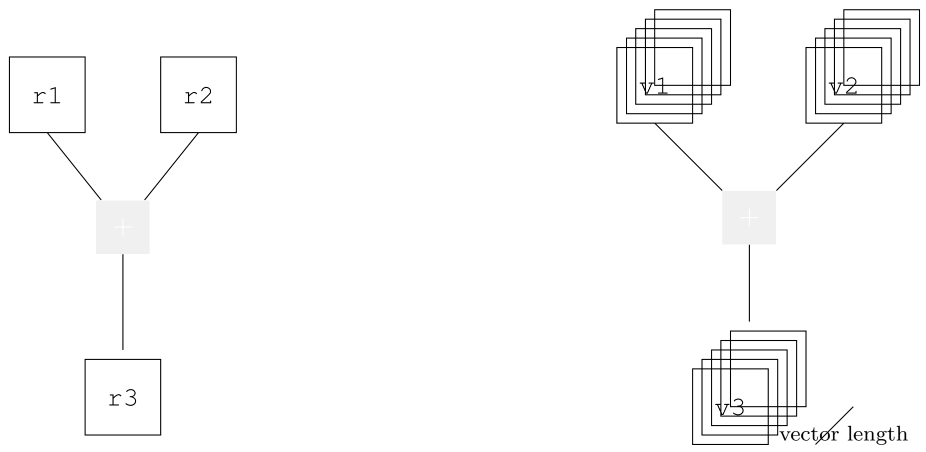
13.1 How is made a Vector processor

A vector processor is made by:

- Load/store Architecture
- Vector Register
- Vector Instruction
- Hardwired Control
- Highly Pipelined Functional Unite
- Interleaved Memory System
- No Data Cache
- No Virtual Memory

From this is created the ***VMIPS*** that will be discussed later.
The idea is to combine multiple scalar models at one time To perform this idea we must be sure that every element is independent of each other[21].
The use of this paradigm is very useful for exploiting significant **Data level parallelism** with more energy efficient than **MIMD.**

[21]NO LOOP CARRIED



```
add r3, r1, r2
```
(a) Scalar model, 1 operation at a time

```
add.vv v3, v1, v2
```
(b) Vector mode, n operations at a time

Figure 61: Comparison of scalar and vector models

13.2 Vector processing

A vector processor consists of a pipelined scalar unit and vector units. There are 2 styles of vector architectures:

- **Memory-Memory** vector processors: all vector operations are memory to memory
- **Vector-Register** processors: all vector operations are between vector registers *(except* `LOAD` *and* `STORE`*)*

We will use *the Vector-Register* processor approach.

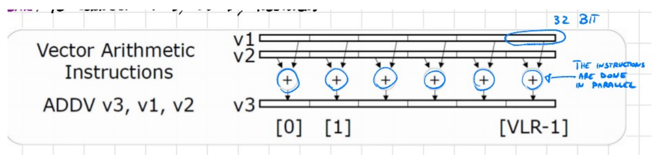

32 BIT
Vector Arithmetic Instructions
ADDV v3, v1, v2
v1
v2
v3
THE INSTRUCTIONS ARE DONE IN PARALLEL
[0]
[1]
[VLR-1]

Figure 62: Arithmetic Instruction

13.2.1 VMIPS

The *VMIPS* is an architecture loosely based on *CRAY-1* supercomputer. It features:

- 8 **Vector registers**
  - each register holds a 64 elements vector with 64 bits per element
  - the register file has at least 16 read ports and 8 write ports
- **Vector functional units**
  - fully pipelined so they can start a new operation every cycle
- **Vector load store unit**
  - fully pipelined so they can read one word per clock cycle
- 32 general-purpose **scalar registers**
- 32 floating point **scalar registers**
- Cross-bar to connect FUs, LSUs, and register

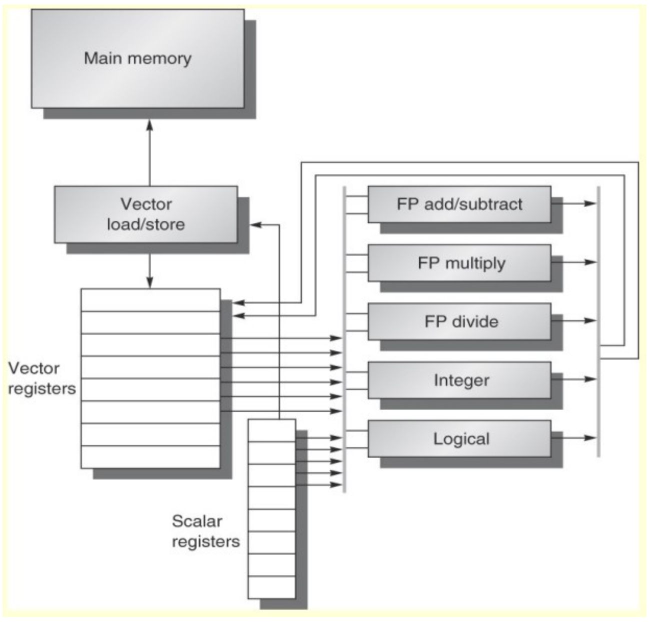


Figure 63: VMIPS architecture

13.3 DAXPY Operation

This operation is from well known type of operation

$$B = a * X + B$$

It's used in ML applications but also in multimedia.
Example of vector code[22]:

```
// C code
for (i = 0; i < 64; i++)
  C[i] = A[i] + B[i];
```

```
  // Scalar Code
  LI R4, //64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

```
// Vector Code
LI VLR, //64
LV V1, R1
LV V2, R2
ADDV.D V3,V1,V2
SV V3, R3
```

In the scalar version, we will have 9 instructions in the loop for 64 elements and 2 initialization instruction

$$(64 \cdot 9) + 2 = 578 \quad instruction \quad per \quad loop$$

In the vectorial version, we have a slightly reduce of instructions, we will have 5 instructions per 64 elements so

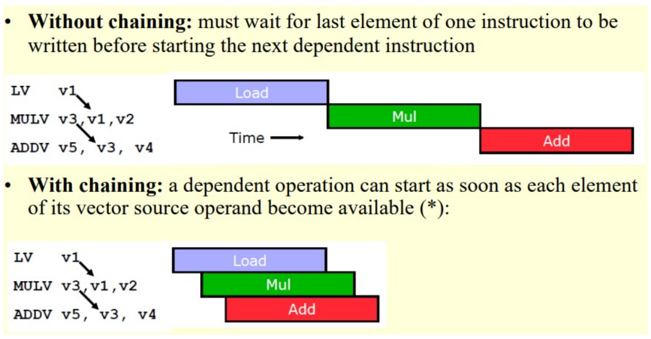
$$5 \cdot 64 = 321 \quad Clock$$

But this could be better if we put some optimization.

[22]OCIO ALLE LOOP CARRIED

13.3.1 Vector Execution Time

VMIPS functional units consume one vector element per clock cycle, so the execution time of one vector instruction is approximately the vector length.
To simplify the calculations, the **Convoy notion** has been introduced: it considers a set of vector instructions that could potentially execute together *(generating no structural hazards)*: for a vector of length n and m convoys in a program, $n \cdot m$ clock cycles are needed.



- **Without chaining:** must wait for last element of one instruction to be written before starting the next dependent instruction

```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```

- **With chaining:** a dependent operation can start as soon as each element of its vector source operand become available (*):

```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```

Figure 64: With or Without Chaining

With this optimization, we go from 321 Clock to 69 Clock. This optimization will also be safe in terms of data dependency because we start use data when is already solved.

13.3.2 Advantages over scalar

- **Vectors operations chaining**
  - a vector operation can start as soon as the individual elements of its vector source operand become available
  - the results of the first functional unit in the chain are forwarded to the second functional unit
  - implemented by allowing the processor to read and write a particular vector register at the same time provided it is to different scalar elements
  - flexible chaining: allow a vector instruction to chain to essentially any other vector instruction, assuming that it does not generate any structural hazard
- **Pipeline stalls greatly decreased**
  - vector version: only the first element of the vector must be stalled, and after that, the results can come out at every cycle
  - a scalar processor can try to get a similar effect through loop unrolling but it cannot get the dynamic instruction count to decrease
- **The operations are executed in parallel**, along multiple lines

13.3.3 Multiple Lanes

Instead of generating an element per clock cycle in one lane, I can spread the elements into multiple lanes to improve the vector performance

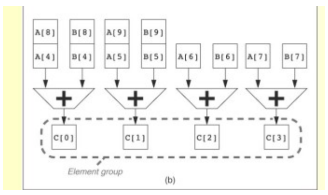


Figure 65: Multiple Lane

13.3.4 Vector Lenght Controller

The Maximum Vector Lenght(MVL) is the physical length of the vector register in a machine, but if we have some differences?
We can have 2 cases:

- Vector smaller, with the register *Vector-Lenght Register* we can control the length of any vector operation and we can set a smaller value than MVL.
- Vector Unknown at compile time, we shoud use a technique called *Strip Mining*, to restructure the code in a ***Remainder***[23] and in different blocks equal to MVL.

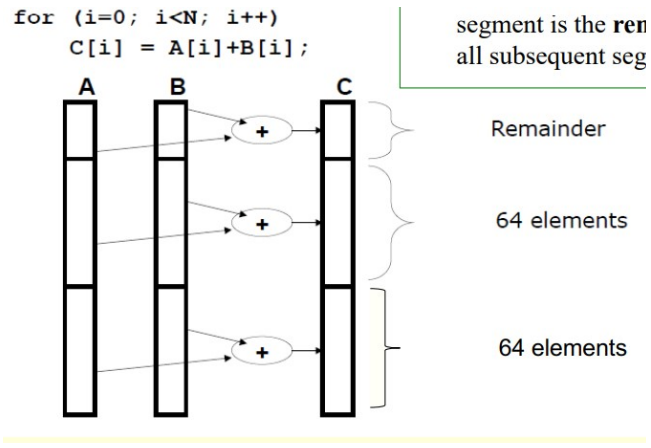


Figure 66: Strip Mining

13.3.5 Vector Mask Register

What if there is a control statement in our loop?
We can use a ***Vector Mask Register***(VMR), to "disable" some elements to not perform the operation in the element that does not satisfy those conditions.
Every Vector instruction operates **ONLY** to the vector elements whose corresponding mask is set to 1.

13.3.6 Stride

It's called **the stride** of an array *(also referred to as increment, pitch, or step size)* the number of locations in memory between the beginnings of successive array elements, measured in bytes or units of the size of the array's elements. The stride **cannot be smaller** than the element size but can be larger, indicating extra space between elements.
This idea is used in example when I'm using a matrix and I want to use it by the column and not by the row, so I've to put a load instruction with a stride, to jump into the element I need.

[23] $N mod MVL$

13.3.7 Scatter and Gatter

This operation is used in sparse matrix, this type of matrix is very common in ML.

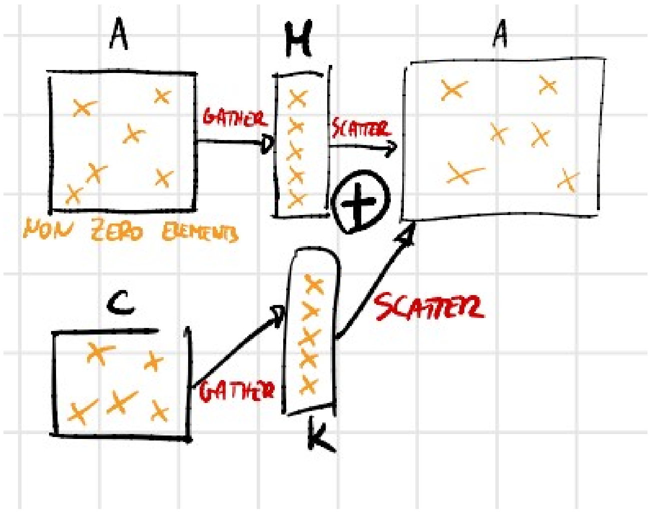


Figure 67: Scatter and gatter

14 GPU

Originally *(in the mid '90s)*, *GPUs* were dedicated fixed functions for generating 3D graphics, including high-performance *(SP)* floating point units. They provided workstation-like graphics for PCs while the user had no real way to program them, but they could merely configure the pipeline.
Over time, more programmable features were added to the *GPU*, enabling millions of vertices to be rendered in a single frame with very constrained programming models.
Some users noticed they could do general-purpose computation by mapping input and output data to images and computation to vertex and pixel shading computations. It was however a very difficult programming model as the programmer had to exploit the graphics pipeline to perform general computation.

14.0.1 General Purpose *GPUs - GP-GPUS*

In 2006, *NVIDIA* introduced *GeForce 8800 GPU* supporting a new programming language, called `CUDA` *("Compute Unified Device Architecture")*. Subsequently, the other companies in the industry started pushing for `OpenCL` *("Open Computing Language")*, a vendor-neutral version of the same language available for multiple platforms.
`CUDA` takes advantage of the *GPU* computational performance and memory bandwidth to accelerate some kernels for general-purpose computing. The host *CPU* issues data-parallel kernels to *GP-GPU* device for execution.

`CUDA` programming model:

- The **programmer writes** a serial program that calls parallel **kernels**
- A **kernel executes** in parallel across a set of parallel **threads**
- The **programmer organizes** these threads into thread block and grids of **thread blocks**
- A **thread block** is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared memory space private to the block
- A **grid** is a set of thread blocks that may be executed independently and thus may execute in parallel *(or any order)*
  - thread **creation**, **scheduling** and **termination** are handled by **underlying hardware**
  - `CUDA` model **virtualizes** the processor

14.0.2 Hardware execution model

- The *GPU* is built from **multiple parallel cores**, each one containing a multithreaded *SIMD* processor with multiple lanes but without scalar processor
  - each tread block executes on **one core**
  - some newer models feature a **scalar unit**
- The *CPU* **sends the whole grid over to the** *GPU*, which distributes thread blocks among cores
  - the programmer is **not aware** of the number of cores

GPUs use a **SIMT** *(Single Instruction Multiple Thread)* model, where individual *(scalar)* instruction streams for each `CUDA` thread are grouped for *SIMD* execution on the hardware. *NVIDIA* groups 32 threads (μT) into a **warp**.
Warps are multithreaded on cores, and each of them is managed by the hardware: one warp composed by 32 threads μT represents a single thread in the hardware. Multiple threads are then interleaved in execution on a single core to reduce latencies to memory and FUs.
A single thread block can contain multiple warps *(up to $512\mu T$ in `CUDA`)*, all mapped into a single core; multiple blocks can also execute on a single core. Individual parallel threads of a warp start together but are free to branch and execute in **parallel**.
SIMT model gives the illusion of many independent threads running on a single core, but for efficiency's sake, the programmer must try and keep each μT as aligned as possible, in *SIMD* fashion.

Simple `if-then-else` instructions are compiled into predicated execution, equivalent to vector masking ; more complex control flow code must be compiled into branches.
Hardware tracks which μT take or don't take branches: if all go in the same direction, the hardware can execute the block in *SIMD* fashion; otherwise, a mask vector indicating *taken* or *not taken* is created.
The *not taken* paths keep running under the mask, while the *taken* paths `PC` and mask are pushed into a hardware stack to be executed later.

An illustration of the hardware execution model is shown in Figure 68.

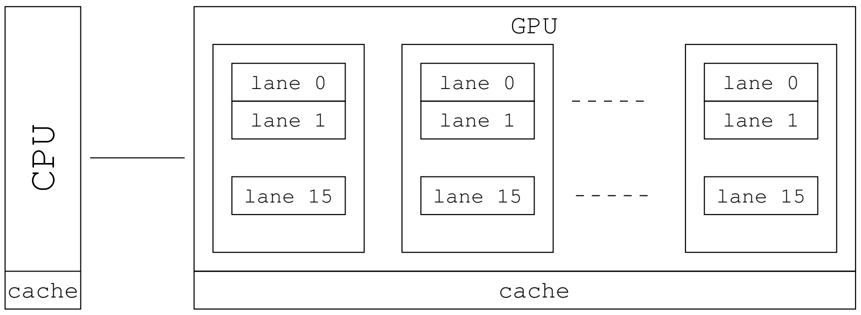


Figure 68: Hardware execution model