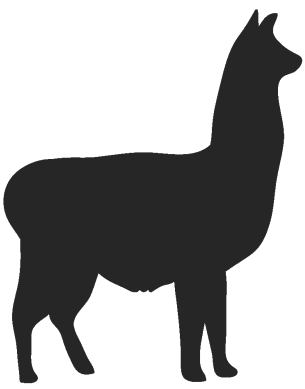
Advanced Computer Architectures

Lorenzo Rossi and everyone who kindly helped!

2021/2022

Last update: 2022-09-04



no alpaca has been harmed while writing these notes

Contents

Introduction

This document contains the notes for the *Advanced Computer Architectures* course, relative to the *2021/2022* class of *Computer Science and Engineering* held at *Politecnico di Milano*.

- Teacher: *Donatella Sciuto*
- Support teacher: *Davide Conficconi*
- Textbook: *Hennessey and Patterson, Computer Architecture: A Quantitative Approach*

By comparing these notes with the material *(slides, video lessons)* provided during the lectures, you will find a lot of discrepancies in the order in which they are represented.
This might look like a bizarre *(if not completely stupid)* choice, but it has been deliberate as I have found the lessons quite confusing in their ordering and how each topic was introduced. You will still find each of the topics explained during the lessons, including something more *(sometimes)*.

If you find any errors and you are willing to contribute and fix them, feel free to send me a pull request on the GitHub repository found at github.com/lorossi/advanced-computer-architectures-notes.

A big thank you to everyone who helped me!

Lorenzo Rossi

1 Introduction to the Computer Architectures

1.1 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
  - Same algorithm programmed and implemented in different ways, so if one fails the other are 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 informations *(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

MIMD an *SIMD* architectures will be further explained in sections 9 and 10. An illustration of the different architectures is displayed in Figure 1.

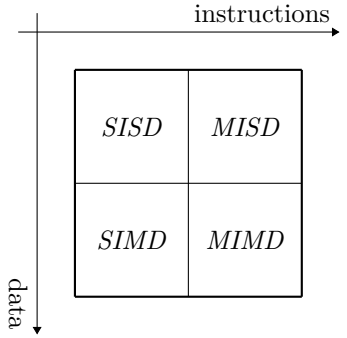


Figure 1: Flynn Taxonomy

1.2 Hardware parallelism

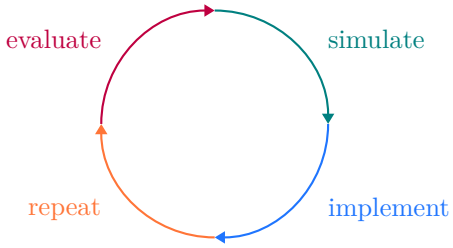
There are different types of hardware parallelisms:

- **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.

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 such:



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

Figure 2: Computer engineering methodology

There are more constraints not contained in this models, such as technology trends.

When one computer is faster than another, what quality is being addressed? **It depends on what it's important.**
The picked qualities may change according to the use case or the user itself. 2 metrics are normally used:

1. Computer system user
   - **le** for program execution
   - $\texttt{execution time} = \texttt{time end} - \texttt{time start}$
2. Computer center manager
   - **le**
   - $\texttt{completion rate} = \texttt{number of jobs} \div \texttt{elapsed time}$

2.1 Response time vs throughput

Is it true that $\texttt{throughput} = 1 \div \texttt{average response time}$? The answer can be given only if it's clear if there's overlapping 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 set
- 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, one must focus on the common sense. *Sadly*, it is a 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 multiplier, so it makes sense to optimize it first
- If database server has 50 disks processor, storage dependability is more important than system dependability so it has to be optimized 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. The Amdahl's law states that:

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

2.2.1.1 Corollary

The best possible outcome of the *Amdahl's law* is:

$$Speedup = \frac{1}{1 - F}$$

> *"If an enhancement is only usable for a fraction of task, we can't speed up the task by more than the reciprocal of* 1 *minus the fraction"*

The *Amdahl's law* expresses the law of diminishing returns. It serves as a guide to how much an enhancement will improve performance and how to distribute resources to improve the cost over performance ratio.

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 pipeline might take different amounts of time. The factors that can influence the *CPU* time is shown in Table 1.

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

Table 1: Relation between factors and *CPU* time

2.2.2.1 *CPU* time and cache

In order to improve the *CPU* time, an **instruction cache** can be used. Using a more realistic model, while calculating *CPU* time, one must must also account for:

- The execution *CPI* - CPI_{EXE}
- The miss penalty - MISS_P
- The miss rate - MISS_R
- The memory references - MEM

Then the $\text{CPI}_{\text{CACHE}}$ can be calculated as:

$$\text{CPI}_{\text{CACHE}} = \text{CPI}_{\text{EXE}} + \text{MISS}_P \cdot \text{MISS}_R \cdot \text{MEM}$$

2.2.3 Other metrics

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

- *MIPS* - million of instructions per second

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

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

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

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

- optimizing compilers

Furthermore, the execution time is compared against test programs that:

- Are chosen to measure performance defined by some groups
- Are available to the community
- Run on machines whose performance is well known and documented
- Can compare to reports on other machines
- Are representative

2.3 Averaging metrics

The simplest approach to summarizing relative performance is to use the total execution time of the n programs. However, this does account for the different durations of the benchmarking programs.
3 different approaches using means can be described as shown in Table 2.

metric	*type of mean*
times	arithmetic
rates	harmonic
execution time	geometric

Table 2: Mean approaches

3 Multithreading and Multiprocessors

3.1 Why multithreading?

Why is multithreading needed?

- *80's*: expansions of superscalar processors
  - In the 80's, people were writing languages in high level programming languages
  - Since compiler optimization was not good enough, it was needed to improve the software translations by making *CPU* instructions that were more similar to high level instructions
  - In order to improve performances, the **pipeline** was introduced
    - it sends **more than one instructions at a time**
    - more instructions completed in the same clock cycle
    - it's kind of a hardware level implicit parallelism
- *90's*: decreasing returns on investments
  - Since all the parallelism was implemented by the hardware *(or, at most, the compiler)*, there was no effective way to manually handle the performance
  - There were many different issues:
    - issue from 2 to 6 ways, issue out of order, branch prediction, all lowering from 1 *CPI* to 0.5 *CPI*
    - performance below expectations
    - these performance issues led to delayed and cancelled projects
  - All the previous improvements were due to the shrinking size of the transistors, which was slowly speeding down
    - the number of transistors followed Moore's law, doubling each 18 to 24 months
    - the frequency and the performance per core were not, due to interferences and energy problems
- *2000*: beginning of the multi core era
  - Since increasing the *CPU* frequency could not be achieved any more, the only solution left was to increase the number of threads in every processor
  - This implied that there was a need of introducing a software way to handle the parallelism, in harmony with an enhanced hardware

Motivations for the paradigm change:

- Moderns processors fail to utilize execution resources well enough
  - There's no single culprit:
    - Memory conflicts
    - Control hazards
    - Branch misprediction
    - Cache miss
    - ...
  - All those problems are correlated and there's no way of solving one of them without affecting all the others
  - There's the need for a general latency-tolerance solution which can hide all sources of latency: **parallel programming**

3.1.1 Parallel Programming

Explicit parallelism implies structuring the applications into concurrent and communicating tasks. Operating systems offer systems to implement such features: **threads** and **processes**.
The multitasking is implemented differently basing on the characteristics of the *CPU*:

- **Single** core
- **Single** core with **multithreading** support

- **Multi** core

In multithreading, multiple threads share the functions units of one processor via overlapping. The processor must duplicate the independent state of each thread:

- Separate copy of the register files
- Separate `PC`
- Separate page table

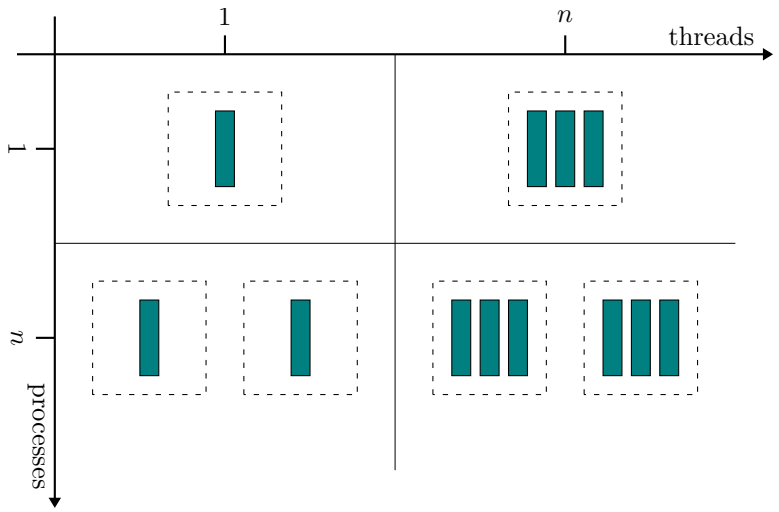


Figure 3: Multiplicity of processes and threads

The memory is shared via the virtual memory mechanisms, which already support multi processes. Finally, the hardware must be ready for fast thread switch: it must be faster than full process switch *(which is in the order of hundreds to thousands of clocks)*.
There are 2 apparent solutions:

1. **Fine grained** multi threading
    - Switches from one thread to another at each instructions by taking turns, skipping when one thread is stalled
    - The executions of more threads is interleaved
    - The *CPU* must be able to change thread at every clock cycle.
    - For n processes, each gets $1/n$ of *CPU* time and n times the original resources are needed
2. **Coarse grained** multithreading
    - Switching from one thread to another occurs only when there are long stalls in the active process
    - Two threads share many resources
    - The switching from one thread to the other requires different clock cycles to save the context

Disadvantages of multithreading:

- for short stalls it does not reduce the throughput loss
- the *CPU* starts the execution of instructions that belongs to a single thread
- when there is one stall it is necessary to empty the pipeline before starting the new thread

Advantages of multithreading:

- in normal conditions the single thread is not slowed down

Could a processors oriented ad *ILP* exploit *TLP*?

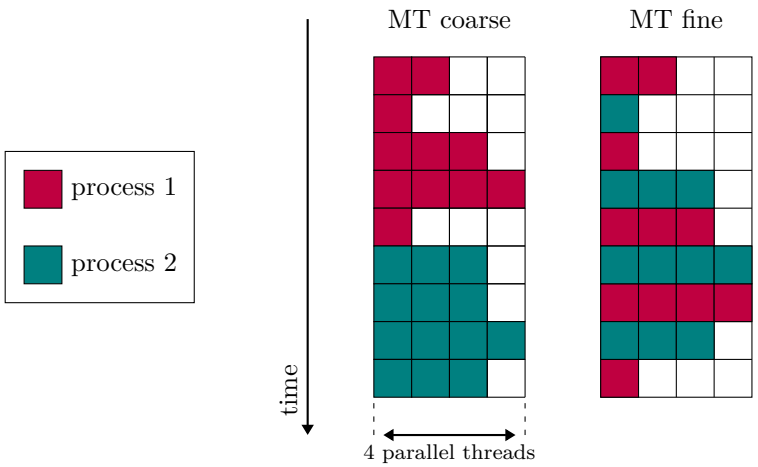


Figure 4: Comparison between fine and coarse multithreading. Each column shows the evolution of 2 different processes spread on 4 different threads over a set amount of time. A filled square illustrates a thread occupied by a process, while the empty ones represent an empty `(idle)` thread.

- **Thread level parallelism**, simultaneous multithreading
  - Uses the resources of **one superscalar processor** to exploit simultaneously *ILP* and *TLP*
  - A *CPU* today has **more functional resources** than what one thread can if fact use
  - Simultaneously **schedule instructions** for execution from all threads
- A *CPU* that can handle these needs must be built
  - A **large set of registers** is needed, allowing multiple process to operate on different data on the same registers
  - **Mapping table for registers** is needed in order to tell each process where to write data
  - Each processor can manage a **set amount of threads**
- This is the most flexible way to manage multithreading but it requires more complex hardware

Comparison between many multithreading paradigms is shown in Figure 5.

3.1.2 Further improvements

It's difficult to increase the performance and clock frequency of the single core. The longest pipeline stage can be split in multiple smaller stages, allowing an higher throughput.
This concept is called **deep pipeline** and has a few drawbacks:

- **Heat dissipation** problems due to the increased number of components
- More stages imply **more faults** since sequential instructions are likely related
- **Transmissions delay** in wires start to get relevant
- **Harder design** and verifications by the hardware developers

3.2 Parallel Architectures

A **parallel computer** is a collection of processing elements that cooperate and communicate to solve large problems in a rapid way.
The aim is to replicate processors to add performance and not design a single faster processor. Parallel architecture extends traditional computer architecture with a communication architecture.
This concept needs:

- **Abstractions** for HW/SW interface
- **Different structures** to realize abstractions easily

Refer to Flynn Taxonomy *(Section 1.1)* for more details about these architectures.

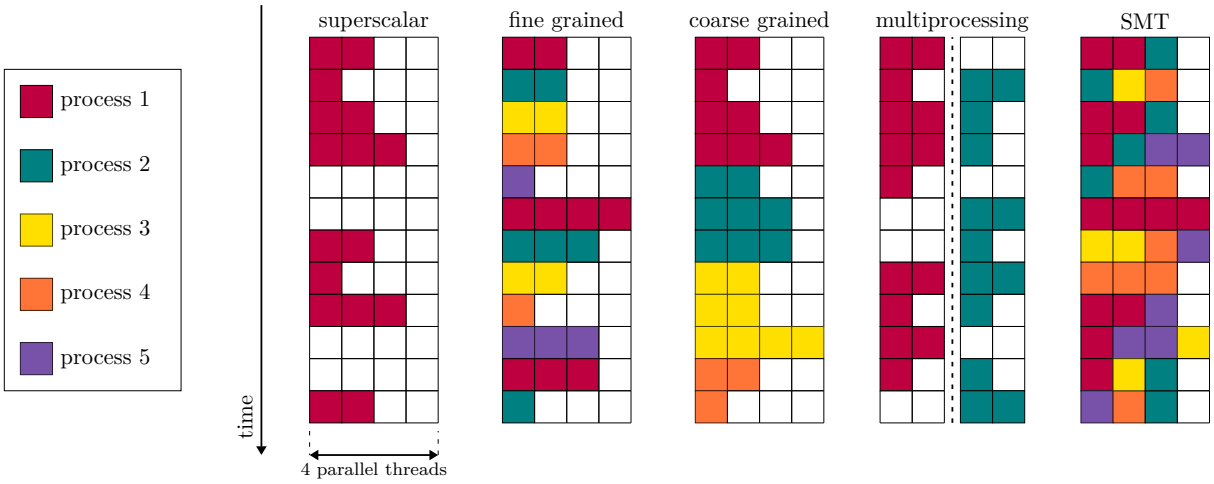


Figure 5: Multithreading comparison. Each column shows the evolution of 5 different processes spread on 4 different threads over a set amount of time. A filled square illustrates a thread occupied by a process, while the empty ones represent an empty `(idle)` thread.

3.3 *SIMD* architecture

The characteristics of the *SIMD* architectures *(Single Instruction Multiple Data)* are:

- **Same instruction** executed by **multiple processors** using different data streams
- Each processor has its own data memory
- **Single instruction memory** and **single control processor** to fetch and dispatch instructions
- Processors are typically **special purpose**
- A **simple** programming model

The programming model features:

- Synchronized units
  - a single program counter
- Each unit has its own addressing registers
  - each unit can use different data address

Motivations for *SIMD*:

- The cost of the control unit is shared between all execution units
- Only one copy of the code in execution is necessary
- All the computation is fully synchronized

In real life:

- *SIMD* architectures are a mix of *SISD* and *SIMD*
- A host computer executes sequential operations
- *SIMD* instructions sent to all the execution units, which has its own memory and registers and exploit an interconnection network to exchange data

SIMD architectures will be explored in depth in Section 10.

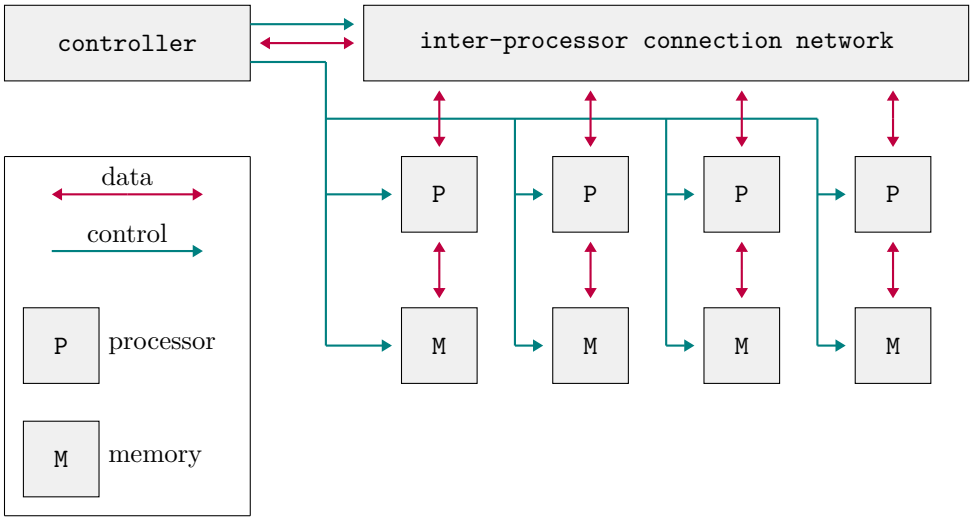


Figure 6: *SIMD* architecture

3.4 *MIMD* architecture

MIMD architectures are flexible as they can function as:

- **Single user** machines for high performance on one application
- Multiprogrammed multiprocessors running many tasks **simultaneously**
- Some combinations of the two aforementioned functions

They can also be built starting from standard CPUs

Each processor fetches its own instructions and operates on its own data. Processors are often off the shelf microprocessors, with the upside of being able to build a scalable architecture and having an high cost performance ratio.
To fully exploit a *MIMD* with n processors, there must be:

- At least n threads or processes to execute
  - those independent threads are typically identified by the programmer or created by the compiler
- Thread level parallelism
  - parallelism is identified by the software *(and not by hardware like in superscalar CPUs)*

MIMD machines can be characterized in 2 classes, depending on the number of processors involved:

- Centralized shared-memory architectures
  - At most a few dozen processors chips *(less than* 100 *cores)*
  - Large caches, single memory multiple banks
  - This kind of structures is often **symmetric multiprocessors (SMP)** and the style of architecture is called **Uniform Memory Access (UMA)**
- Distributed memory architectures
  - Supports **large processor count**
  - Requires **high bandwidth** interconnection
  - It has the disadvantage of the high volume of data communication between processors

SIMD architectures will be explored in depth in Section 9.

4 Pipeline recap

The pipeline *CPI (clocks per instruction)* can be calculated as the sum 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 pipelining 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

4.1 Stages in *MIPS* pipeline

The 5 stages in the *MIPS* pipeline are:

1. **Fetch** - `FE`
   → Instruction fetch from memory
2. **Decode** - `ID`
   → Instruction decode and register read
3. **Execute** - `EX`
   → Execute operation or calculate address
4. **Memory access** - `ME`
   → Access memory operand
5. **Write back** - `W`
   → Write result back to register

Each instructions 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 an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along.
The stages are usually represented like in Figure 7.

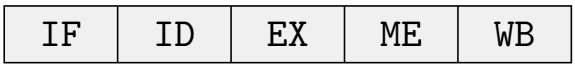


Figure 7: Stages in *MIPS* pipelines

4.2 Pipeline hazards

An **hazard** is a fault in a pipeline. They are 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 the operands involved in said dependance. They prevent the next instruction in the pipeline from executing during its designated clock cycle, reduce the performance from the ideal speedup.
There are three classes of hazards:

1. **Structural** hazards, due to attempt to use the same resource from different instructions at the same time
2. **Data** hazards *stalls*, due to attempt 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 the 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 attempt to make a decision 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
```

4.2.1 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 6.1.

4.3 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?"*

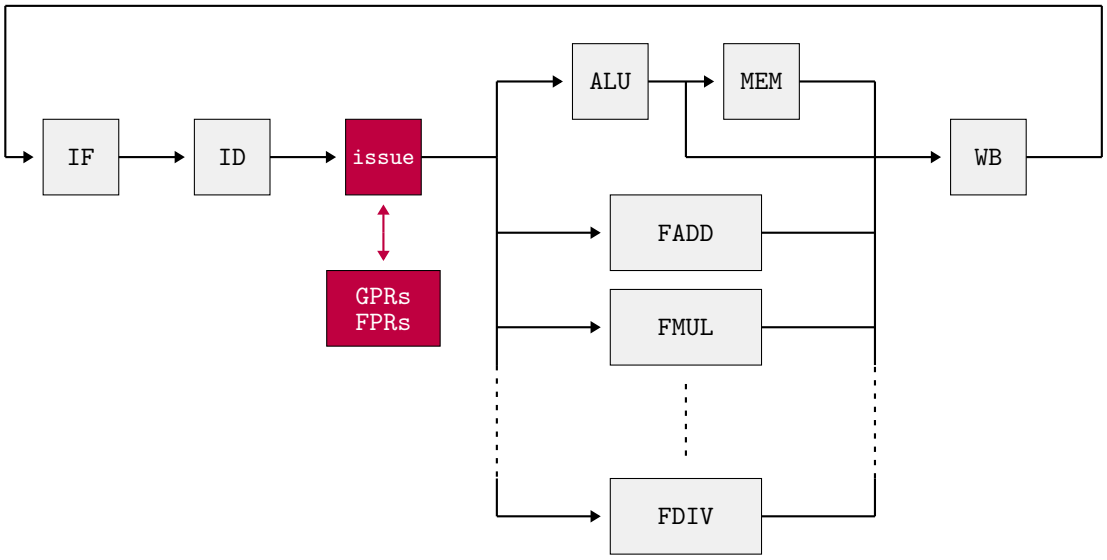


Figure 8: Complex-in order pipeline

In the complex in order pipeline there isn't a single ALU any more 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 usings stalls)* of an instructions in case of errors.
The pipeline is now constituted by 6 stages, normally represented like in Figure 9.

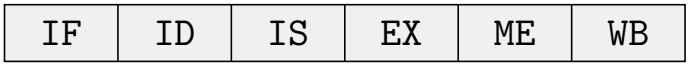


Figure 9: Pipeline stages with issue

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**

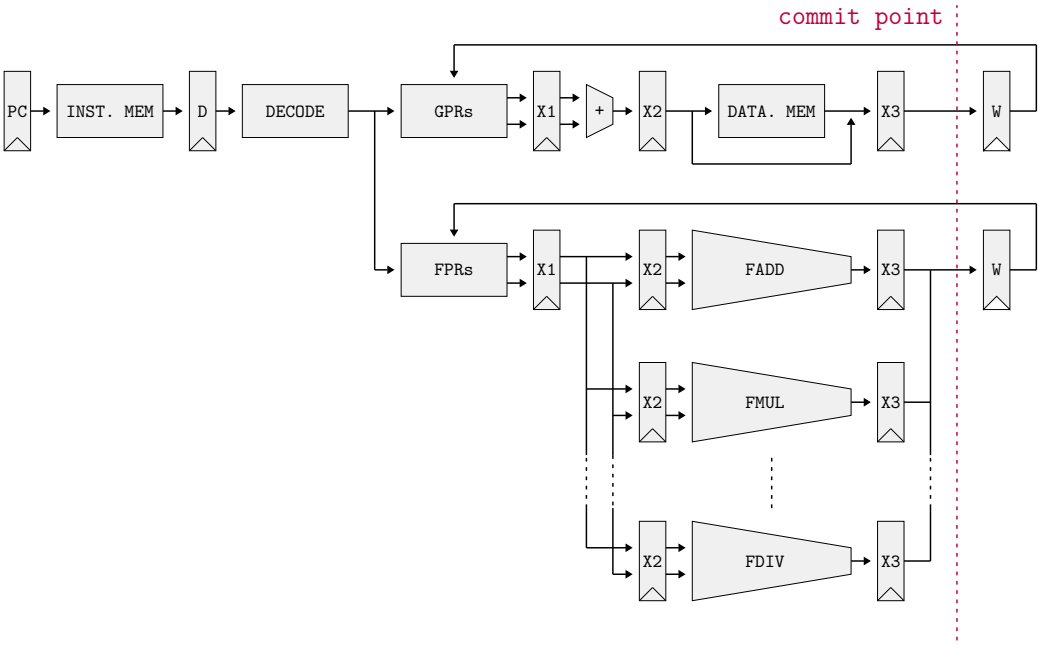


Figure 10: Complex pipelining

Formally, all the different executions must be balanced.

The main issues are:

- **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*)
- 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?

4.4 Instructions issuing

To reach higher performance, more parallelism must be extracted from the program. In other words, dependences must be detected and solved, while instructions must be scheduled as to achieve 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 in order to dispatch an instruction:

1. Check if the **functional unit** is available

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

Table 3: Data structure to keep track of *FUs*

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 3.
An instruction at the `issue` stage consults this table to check if:

- 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 6.1.2)*, this approach will be discussed more in depth.

4.5 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.

4.5.1 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 type 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 original instructions ordering 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 apparently different addresses may refer to the same memory location. As a consequence, it's easier to rename a **register** than renaming a **memory location**. It can be done either **statically** by the compiler or **dynamically** by the hardware.

4.5.2 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.

4.5.3 Control Dependeces

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

1. **Instructions execution in 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.

5 Branch Prediction

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

- The **accuracy**, measured in terms or 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 the 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.

5.1 Static techniques

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*

Each one of these techniques will be discussed in the following Sections *(from 5.1.1 to 5.1.5)*.

5.1.1 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 state `ID` will result not satisfied *(and the prediction is correct)* performance can be preserved.
If the condition in stage `ID` will result 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.

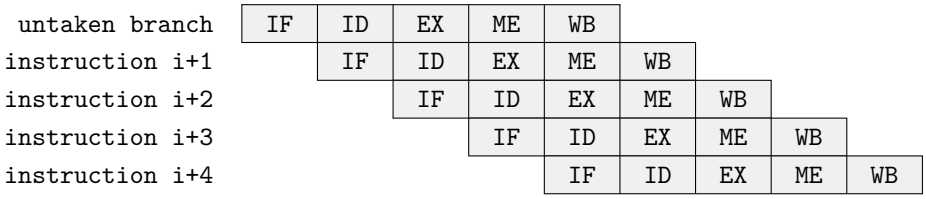


Figure 11: Branch always not taken: success

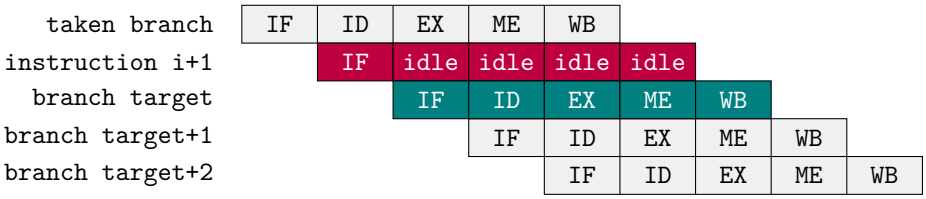


Figure 12: Branch always not taken: fail

5.1.2 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. In *MIPS* pipeline, the branch target address is not known before the branch outcome, so **there is no advantage in this approach**.

5.1.3 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 the 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

5.1.4 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.

5.1.5 Delayed Branch

The compiler statically schedules and 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 a one delay slot. Almost all processors with delayed branch have a single delay slot, as it's difficult for the compiler to fill more than one slot.
If the branch:

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

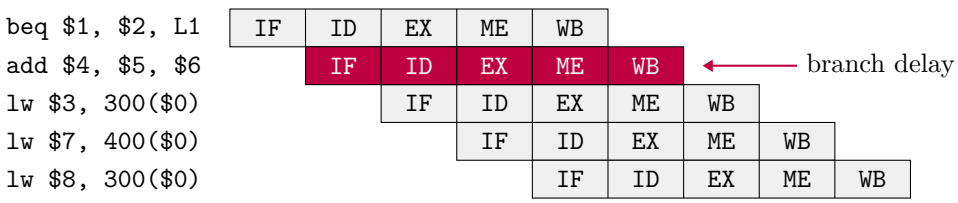


Figure 13: Delayed branch

The compiler 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 better analyzed in the following paragraphs.

In general, the compilers are able to 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 the 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 processor 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.

5.1.5.1 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 14.

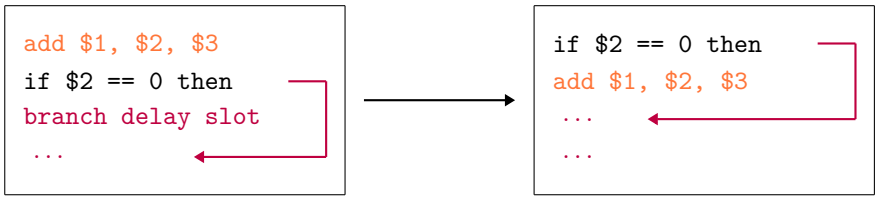


Figure 14: From before

5.1.5.2 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 15.

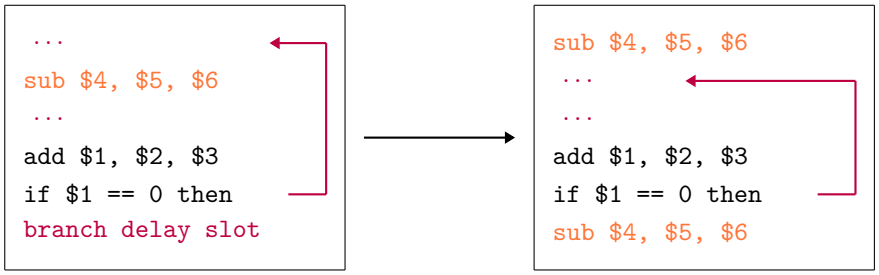


Figure 15: From target

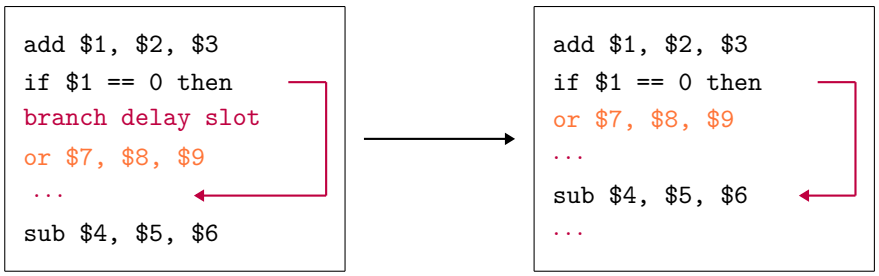


Figure 16: From fall through

5.1.5.3 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 16.
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 the unexpected direction.

5.2 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 wil depend on the behaviour of the branch at run time and will change if the branch changes its behaviour during execution.

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

5.2.1 Branch Target Buffer

The **Branch Target Buffer** is a cache storing the predicated 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 17. The predicted target address is expressed as `PC`-relative.

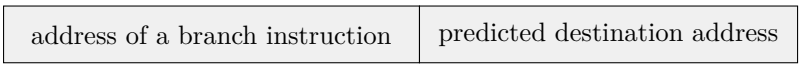


Figure 17: Typical entry of the *BTB*

The structure and operation of the *BTB* is shown in Figure 18.

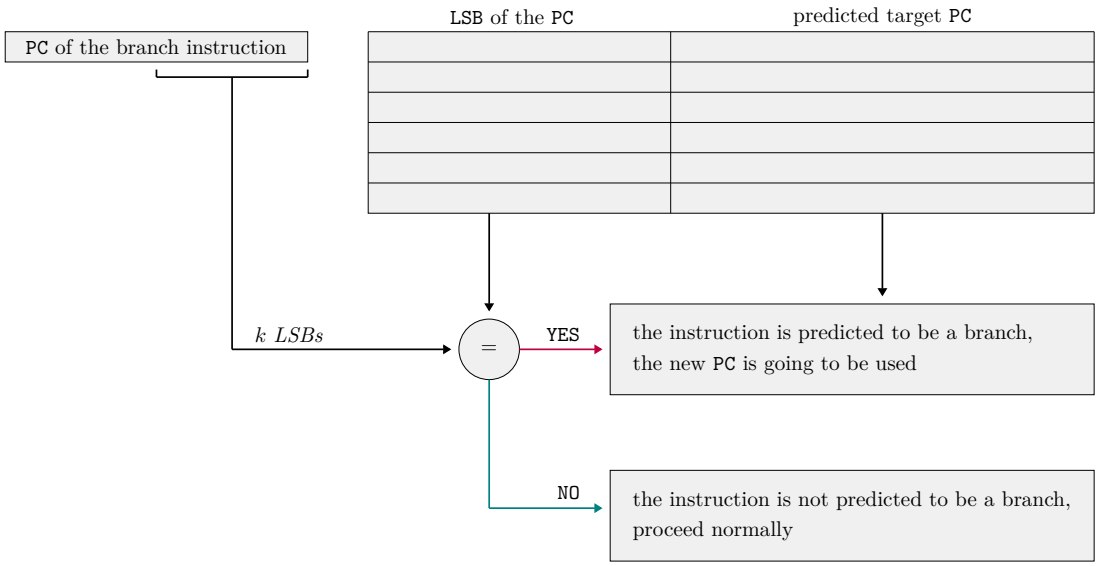


Figure 18: Structure of the *BTB*

5.2.2 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 structure of the *BTB* is shown in Figure 19.

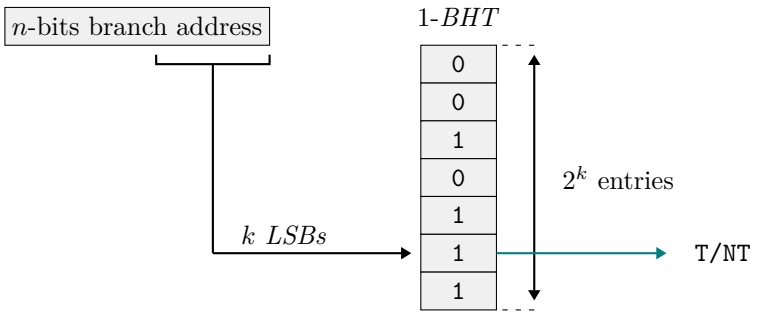


Figure 19: Structure of the *BHT*

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. Then the pipeline is 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. Its structure is shown in Figure 20.

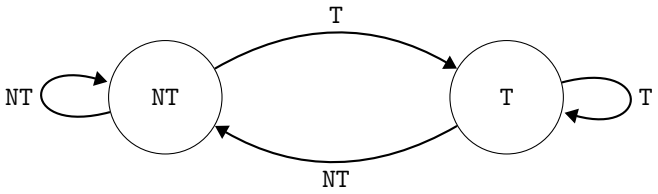


Figure 20: 1-bit *BHT* as *FSA*

A misprediction occurs when:

- The prediction is **incorrect** for that branch
- The same index has been referenced by two different branches, and the **previous history refers to the other branch**
    - to solve this problem it's enough to increase the number or rows in the *BHT* or to use a hashing function (such as in *GShare*).

In a loop branch, even if a branch is almost always taken and then not taken one, che 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 previous execution of the last iteration of the loop

In order to fix this kind of behaviour, the 2-bit *BHT* was introduced.

5.2.3 *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*. Its structures is represented in Figure 21.

5.2.4 *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 its almost useless)*.

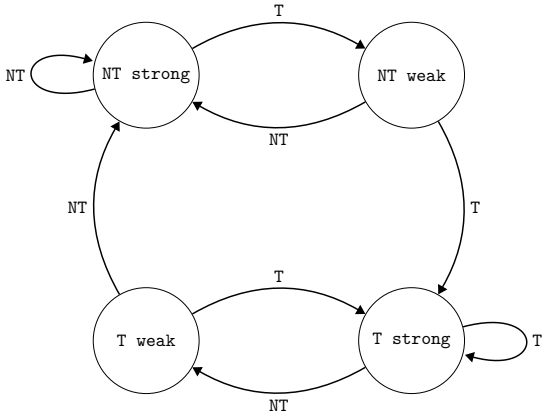


Figure 21: 2-bit *BHT* as *FSA*

5.3 Correlating Branch Predictors

Basic idea: the behaviour of recent branches are 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*. Their scheme is represented in Figure 22.
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.

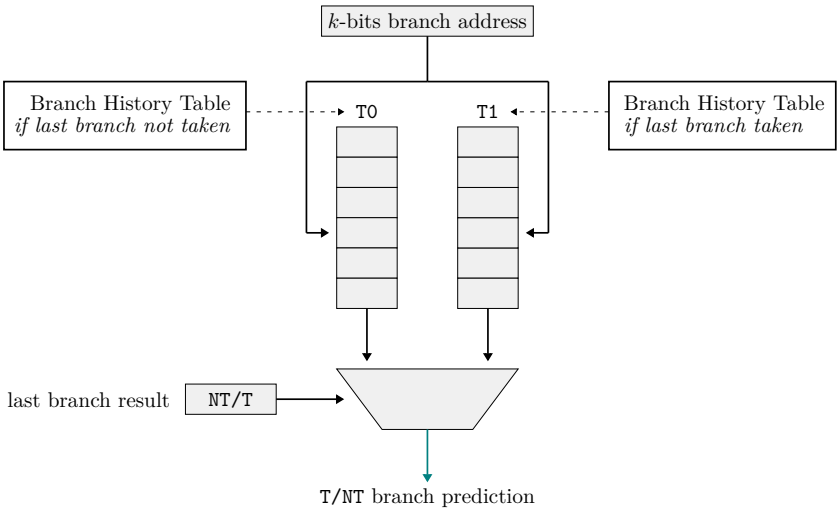


Figure 22: Structure of the *Correlating Branch Predictors*

5.3.1 *(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)*.
The general structure of a (m, n) *CBP* is represented in Figure 23.

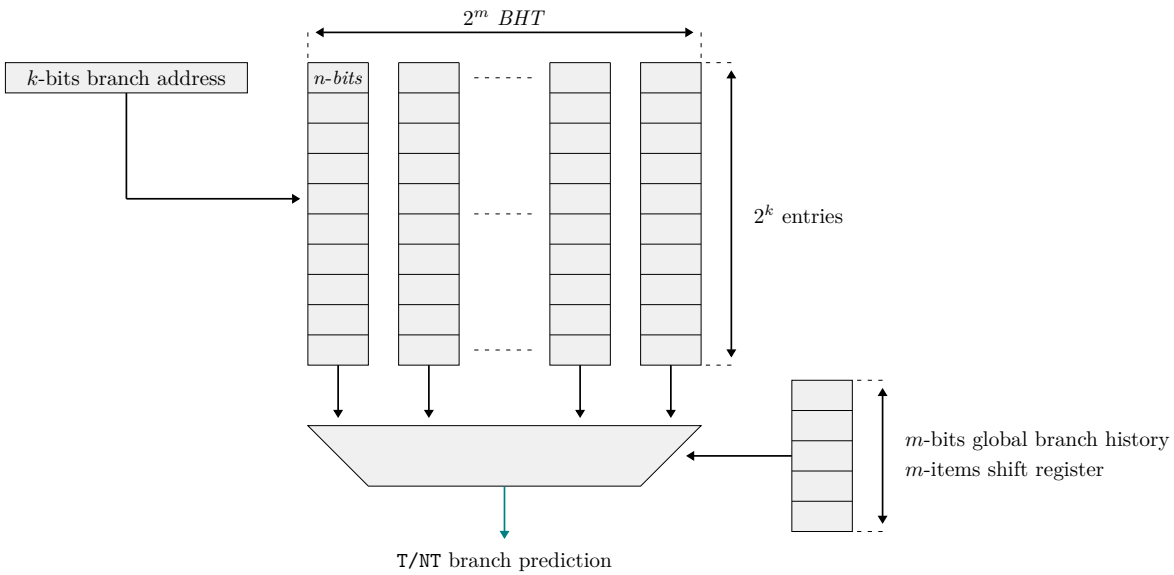


Figure 23: Structure of the *(m, n) Correlating Branch Predictors*

5.3.1.1 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)*

5.3.1.2 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.

5.4 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 bit 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 bit counter scheme.

5.4.1 GA Predictor

The **GA 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` (branch address)
- The GAs are a 2-level predictor: *PHT* is indexed by the content of *BHR* (global history)

The structure of a *GA* predictor is represented in Figure 24.

5.4.2 GShare Predictor

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).
The structure of a *GShare* predictor is represented in Figure 25.

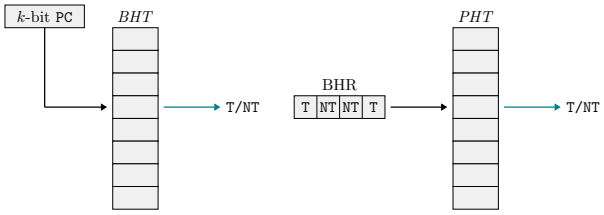


Figure 24: GA Predictor

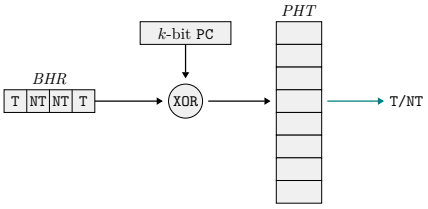


Figure 25: GShare Predictor

6 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 goal is only achievable if there are no **hazards** *(as described in Section 4.2)*.

Two properties are critical to program correctness *(and normally preserved by maintaining both data and control dependences)*:

1. **Exception behaviour**: preserving exception behaviour means that any changes in the ordering of instructions execution must not change how exceptions are raised in the program
2. **Data flow**: actual flow of data values among instructions that produces the correct results and consumes them

6.1 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.

6.1.1 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

The two main techniques used by hardware to minimize stalls are:

- **Forwarding**
  - The result from the `EX/MEM` and the `EX/WB` pipeline registers is **fed back** to the `ALU` inputs
  - If the forwarding hardware detects that the previous `ALU` operation has **written the register corresponding to a source for the current** `ALU` operation:
    - control logic selects the forwarded result as the `ALU` input
    - the value read from the register file is discarded
  - The `ALU` needs **multiplexers** that allow it to select the correct inputs from the pipeline
  - This technique can be generalized to include **passing a result directly to the functional unit** that requires it
    → in that case, a result is forwarded from the pipeline register corresponding to the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit
- **Stalling**
  - Since not all potential data hazards can be solved by bypassing, so a piece of hardware called *pipeline interlock* is added
  - When it detects a hazard, it **stalls** the pipeline until that hazard it solved
  - The stalls are often referred to as *"bubbles"*

Advantages of dynamic scheduling:

- It enables handling some cases where dependences 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**

6.1.2 CDC6600 Scoreboard

As discussed earlier *(Section 4.4)*, a specific data structure is needed to solve data dependences without specialized compilers. The first implementation of such an an hardware is found in the **CDC6600 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 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 on structural hazards.
Is solves the issue of data dependences 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 the 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 in 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 26, while the structure of the scoreboard is represented in Figure 27.

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

Figure 26: Pipeline introduced by the *Scoreboard*

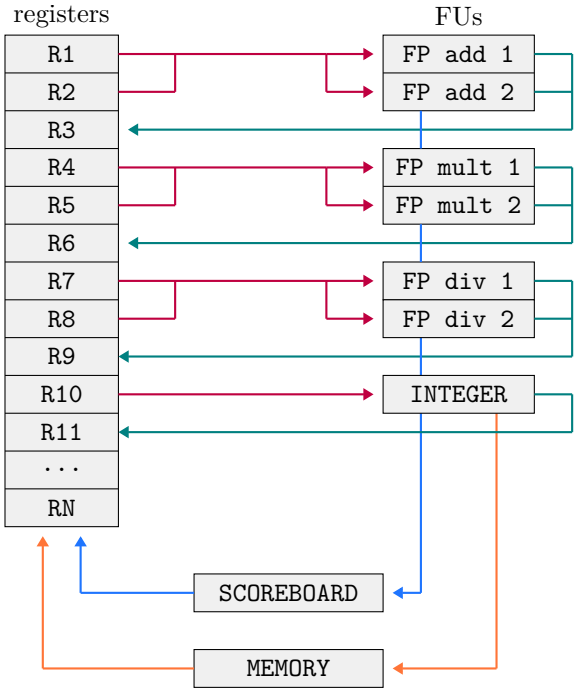


Figure 27: Structure of the *Scoreboard*

6.1.2.1 Four stages of Scoreboard Control

The four stages of the 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 is 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 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

6.1.3 Tomasulo algorithm

The Tomasulo algorithm is a **dynamic** algorithm that allows execution to proceed in presence of dependences. It was invented at IBM 3 years after *CDC 6600* scoreboard with the same goal.
The key idea behind this algorithm is to **distribute** the control login and the buffers within *FUs*, as opposed to the scoreboard *(in which the control logic is centralized)*.
The operand buffers are called **Reservation Stations** (or *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.

6.1.3.1 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

6.1.3.2 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

6.1.3.3 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 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 *(generating a WAW if two stores are interchanged)*. However, `LOAD` instructions can be reordered freely. In order 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, 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.

6.1.3.4 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

6.1.3.5 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 an 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

6.2 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 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 are able to achieve such speeds.

A few requirements must be satisfied in order 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 an 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 instruction can be issued each clock cycle

6.2.1 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

6.2.2 Limits dynamic analysis

Dynamic analysis in 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 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

6.2.3 Limits on window size

Size of the window size affects the number of comparisons needed to determine *RAW* dependences. The number of comparisons that are needed with infinite register is:

$$2\sum_{i=1}^{n-1} i = 2 \cdot \frac{(n-1)n}{2} = n^2 - n$$

For a window size of 2000 almost 4 **million comparisons** are needed; a more realistic window with a size of 50 instructions still needs 2450 comparisons.
Today's *CPUs* have constraints deriving from the limited number of register, the search for dependent instructions and the in order instructions issue.

6.2.4 Other limits of modern *CPUs*

- Number of *FUs*
- Number of *buses*
- Number of ports for the register file

All these limitations impose that the maximum number of instructions that can be issued, executed or committed in the same clock cycle is much smaller than the window size.
In real life, the maximum size of issue width is capped at 6. Increasing the issue rate above this value *(i.e. at 12)* would require the *CPU* to:

- issue 3 or 4 data memory accesses per cycle
- resolve 2 or 3 branches per cycle
- rename and access more than 20 registers per cycle
- fetch between 12 and 24 instructions per cycle

The complexities of implementing these capabilities is likely to mean sacrifices in the maximum clock rate. Power consumption is nowadays an issue and it would grow too much.

The key question to answer is whether a technique is **energy efficient** enough:

> *"Does it increase power consumption faster than it increases performances?"*

Multiple issue processors techniques are all energy **inefficient**, as:

- Issuing multiple instructions incurs some overhead in logic that grows faster than the issue rate grows
- A growing gap between peak issue rates and sustained performance is introduced, increasing energy per performance ratio

6.3 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. Data dependence can further limit the amount of *ILP* that can be exploited within a basic block to 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 dependences is 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**

6.4 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 includes:

- 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 28, while at pipeline level is represented in Figure 29.

6.4.1 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.

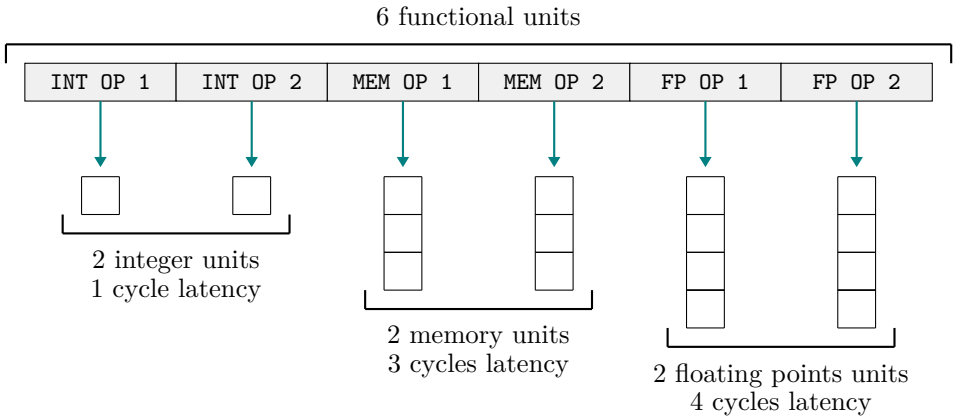


Figure 28: *VLIW* - instructions level

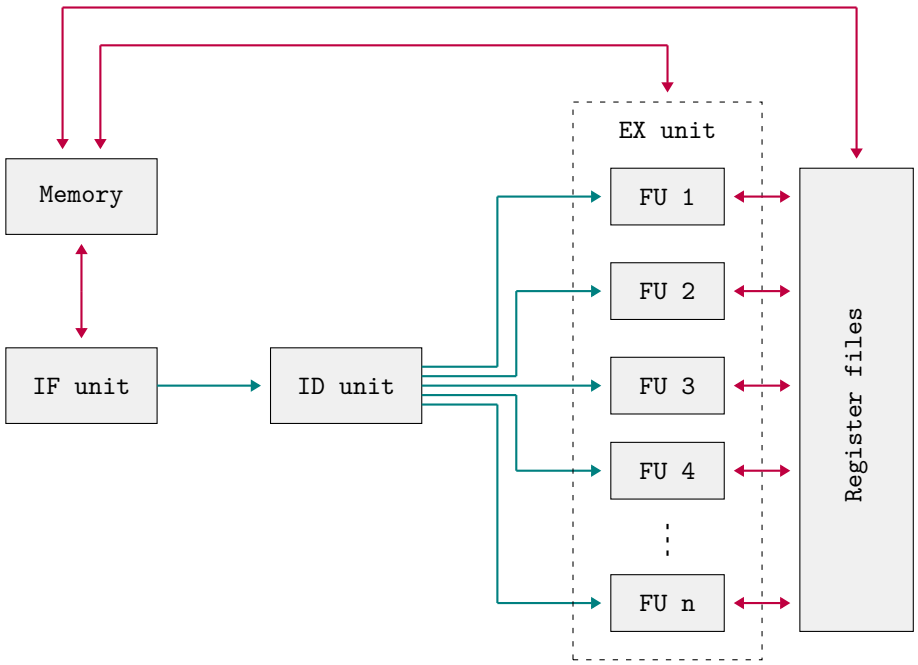


Figure 29: *VLIW* - pipeline level

6.4.2 Basic Blocks and Trace Scheduling

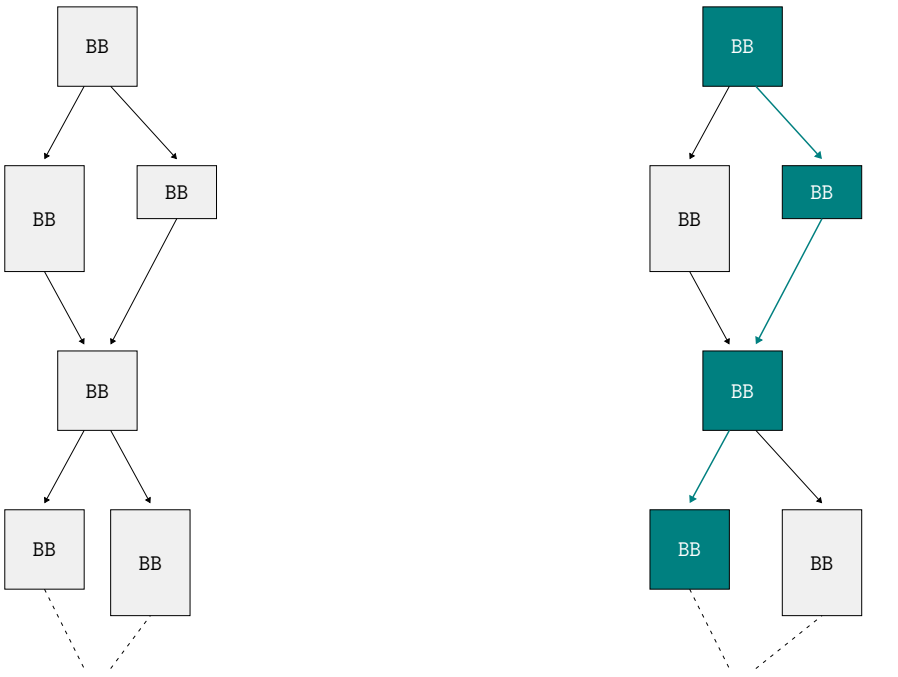
Compilers use sophisticated algorithms to schedule code and exploit *ILP*. However, the amount of parallelism available in a single **basic block**, as previously pointed out, is quite small; furthermore, **data dependence** can limit the amount of *ILP* that can be exploited to less than the average 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 30a.

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 actually executed depends 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 ending)*.
Blocks on non common paths may now have added overhead, so there must be an high probability of taking common paths according 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 code can be found in Figure 30.



(a) Basic blocks

(b) Trace scheduled code

Figure 30: Comparison of scheduled and unscheduled code

6.4.2.1 Code motion and Rotating Register Files in Trace Scheduling

In addition to the need of 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 require lots of registers, due to the duplicated code: in order to solve this issue, a new set of register 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 physical register number.

An illustration of the *RRB* is shown in Figure 31.

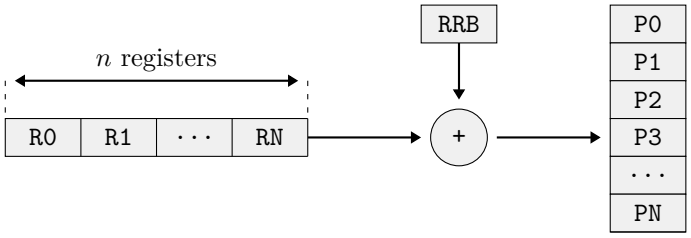


Figure 31: Rotating Register File

6.4.3 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

6.4.4 Static Scheduling methods

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

- Simple code **motion**
- **Loop unrolling** and loop peeling - *Paragraph 6.4.4.1*
- Software **pipelining** - *Paragraph 6.4.4.2*

- Global code **scheduling** *(across basic blocks)*
  - Trace scheduling - *Paragraph 6.4.4.3*
  - Superblock scheduling
  - Hyperblock scheduling
  - Speculative Trace scheduling

6.4.4.1 Loop unrolling

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.

An illustration of the performance improvements can be found in Figure 32.

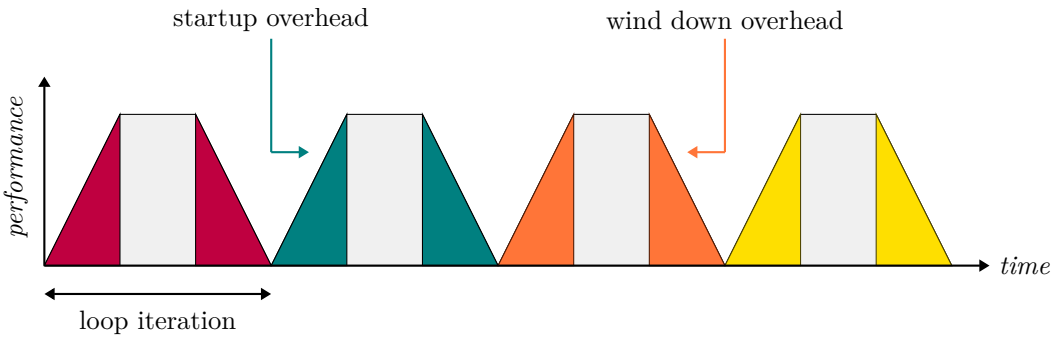


Figure 32: Performance improvement of loop unrolling

6.4.4.2 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 33.

6.4.4.3 Trace scheduling

As discussed in Section 6.4.2, Trace Scheduling does not support loops.
In order 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, most frequently executed traces get better schedules.

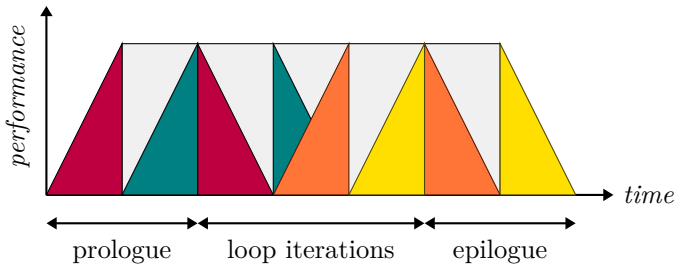


Figure 33: Performance improvement of software pipelining

7 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 dependences are resolved

The outcome of the branches is speculated, then the program is executed as if 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)*.

Generally speaking, Hardware Based Speculation raises power consumption but lowers execution time by more than it increases the average power consumption. The total energy consumed may be less depending on the number of instructions incorrectly executed.

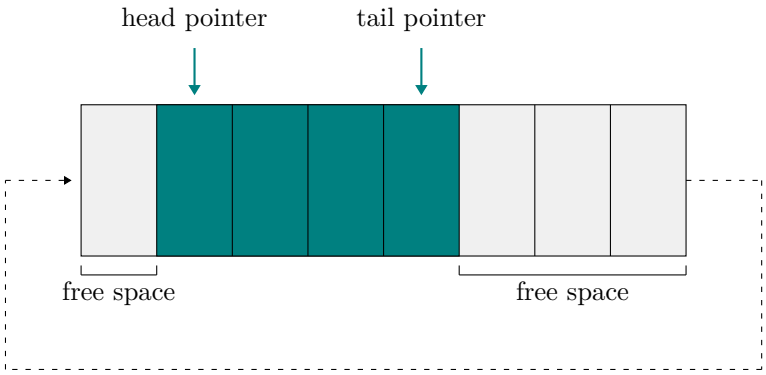
7.1 Reorder Buffer

The **Reorder Buffer** (or *ROB*) holds instructions in *FIFO* order, exactly as issued: when the instructions 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.

ROB are 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 include:

- 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**

The structure of the *ROB* is represented in Figure 34 along with its main entries.



(a) *ROB* structure

(b) *ROB* entry

Figure 34: 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 slots while
- 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 dependences before reaching that point.

7.2 Interrupts and Exceptions with hardware speculation

Hardware Speculation greatly increases the complexity of control dependences. In fact, branch prediction may not be enough to keep an high level of *ILP*.
Special care must be ensued while handling **interrupts** and **exceptions**, because:

- All the instruction **before** the event must be completed
- All the instructions **after** the even must behave as they have never started

Since branch prediction might be wrong, the issue is now to update the processor state accordingly. Finally, out of order completion, post interrupt and mispredict writebacks change the state of the program.
To solve this problems executed instructions are held in *ROB* until they are no longer speculative. The instruction commit is then *in order*: exceptions and interrupt are handled by not being addressed until the instruction that caused them is recorded in the *ROB*.
Temporary storage *(shadow registers and store buffers)* is then needed to hold all results before commit Operands are supplied as well between execution complete and commit.
Once the instruction is committed, its result is placed in the destination register. Therefore, it's easy to undo speculated instructions on exceptions and interrupts.
A scheme of this structure is represented in Figure 35.

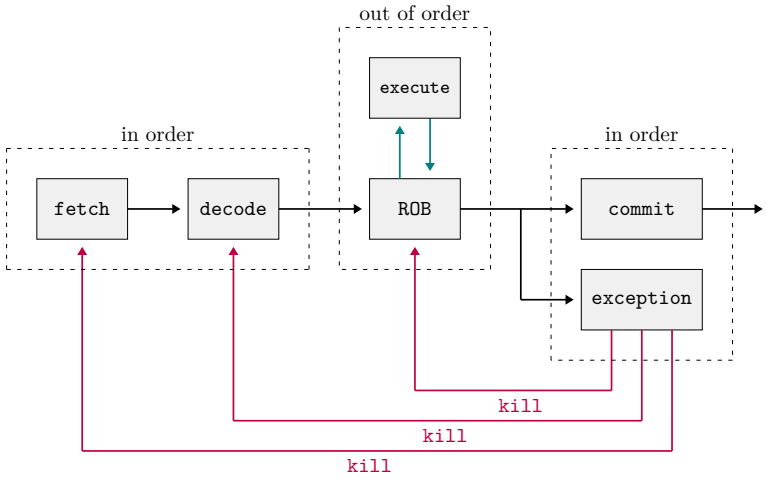


Figure 35: Scheme of the in order commit for precise exceptions

7.3 Steps in the Speculative Tomasulo Algorithm

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 reservation station, the instructions 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

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 in 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 correct successor of the branch

7.3.1 Tomasulo's Algorithm and *ROB*

The 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 completed and the value is ready

Further observations:

- 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 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

7.3.2 Exception handling

The use of a *ROB* with in order instruction commit provides precise exceptions: they are handled by ignoring them until they are ready to commit.
3 different scenarios can then be identified at commit stage:

- If a speculated instruction raises an exception, then the exception itself is recorded in the *ROB*
- If a branch misprediction arises and the instruction should not have been executed, then the exception is flushed along with the instruction when the *ROB* is cleared
- If the instruction reaches the head of the *ROB*, then it's no longer speculative and it should be addressed

7.3.3 Hardware support for Memory Disambiguation

In order to keep track of all stores to memory in program order, a buffer is needed. Its duties include:

- Keep track of addresses and results and when they become available
- Keep a *FIFO* ordering, so stores are retired in program order

While issuing a `LOAD`, the current head of store queue must be recorded in order to know which `STORE` instructions are ahead. When the address is actually found, the `STORE` queue is checked:

- If any `STORE` prior to `LOAD` is waiting for its address:
  → the `LOAD` is stalled
- If any `LOAD` address matches any previous `STORE` address (found by *associative lookup*):
  → a **memory induced** *RAW* hazard is created
  → if the `STORE` value is available, then it's returned
  → if the `STORE` value is not available, then the *ROB* number of the source is returned

All the actual `STORE` instructions commit in order, so there's no need to check for *WAR* or *WAW* hazards throughout memory.

7.4 Explicit Register Renaming

Explicit Register Renaming is a technique that uses physical register file that is larger than the number of register 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 36.

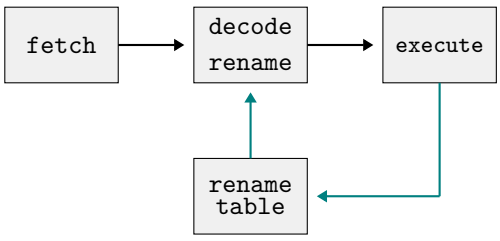


Figure 36: Explicit Register Renaming

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 registers 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 **single** register file
  - there's no need to bypass values from reorder buffer
  - this can be important for balancing the pipeline

7.4.1 Unified Physical Register File

In order to enable the *CPU* to explicitly rename registers, a **Unified Physical Register File** is created.
It works by interfacing *Registers* to *Functional Units*:

- It renames all architectural registers into a **single physical register file** during decode, without reading their values
- Functional units read and write from single Unified Physical Register File holding **committed** and **temporary registers** in execution
- It commits only updates the mapping of architectural registers to the physical registers, **without actually moving data**

The **Renaming Map** is a simple data structure that supplies the physical register number of the register that currently corresponds to the requested architectural register. At the **Instruction Commit** stage the renaming table is updated permanently to indicate that the physical register, holding the destination value, corresponds to the actual architectural register.
A schematic of the Unified Physical Register File is shown in Image 37.

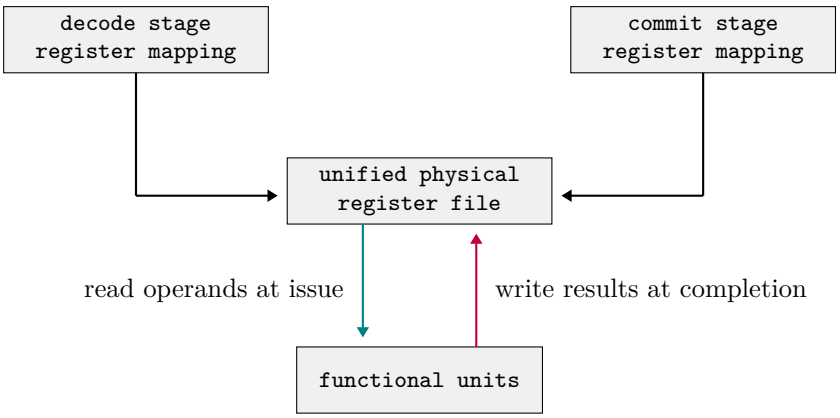


Figure 37: Unified Physical Register File

By using the Unified Physical Register File, the mapping between an architectural register and a physical one **is not speculative**. Any physical register being used to hold the older value of the architectural register is freed.
Deallocating registers is a more complicated:

- Before freeing up a physical register, checks that it is **no longer being used** now and in the near future, must be made
- A physical register corresponds to an architectural register **until it is rewritten**
- There may be **pending uses** of the physical register: the processor must check if any source operand corresponds to that register in the *FU* queue
    - if it does not appear, it can be allocated
    - otherwise, the processor can wait until another instructions that writes the same architectural register commits. This is an easy way to implement a solution that might cause slight delays

A simplified implementation of the pipeline with a Physical Register File is shown in Image 38.

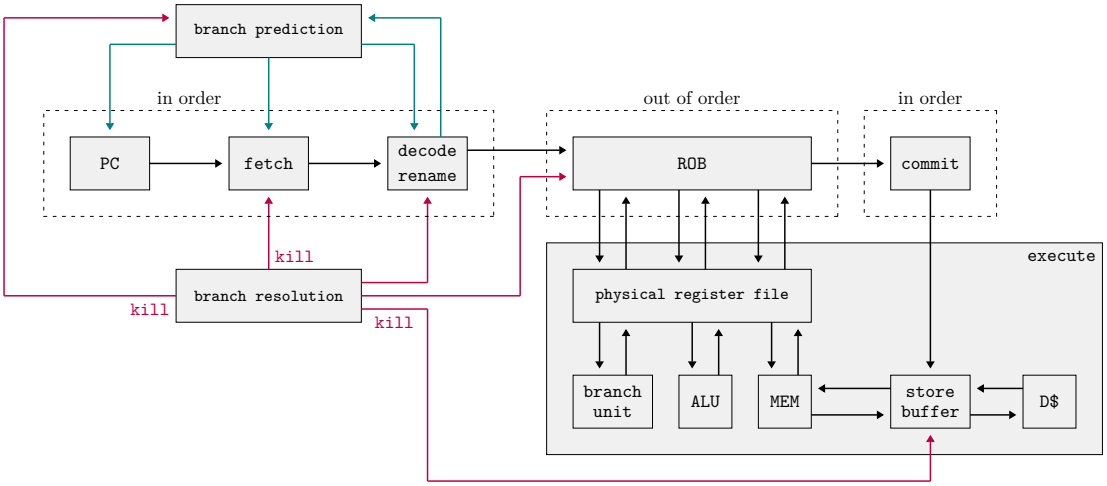


Figure 38: Pipeline Design with Physical Register File

7.5 Explicit Register Renaming and Scoreboard

The scoreboard architecture can be easily adapted to be coupled with Explicit Register Renaming. The 4 stages now become:

- **Issue** - decode instruction, check for structural hazards and allocate new physical register for result
    - instructions are issued in program order for hazard checking
    - no instruction is issued if there are no physical registers
    - no instruction is issued if there is a structural hazard
- **Read operands** - wait until no hazard happens and then read operands
    - all real dependences (*RAW* hazards) are resolved in this stage, since the processor waits for instructions to write back data
- **Execution** on operands
    - the *FU* begins execution upon receiving operanda
    - when the results are read, the scoreboard is notified
- **Write result**
    - the execution of the instruction ends here

Thanks to the Explicit Renaming there's no need to check for *WAR* or *WAW* hazards.

7.5.1 Multiple Issue

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

- The **processor needs additional logic** to handle issue of every possible combination of dependent instructions in the same clock cycle
- Since the number of possibilities increases with the square of the number of instructions that can be issued in one clock 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.
In a similar manner, 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.

7.5.2 Superscalar Register Renaming

- During decode, instructions allocate a new physical destination register
- Source operands are renamed to physical register with newest value
- Execution unit only sees physical register numbers
- RAW hazards must be checked between instruction issuing in the same cycle
    - this operation can be done in parallel using rename lookup

An illustration of the architecture in the two-issue Superscalar Register Renaming is found in Image 39.

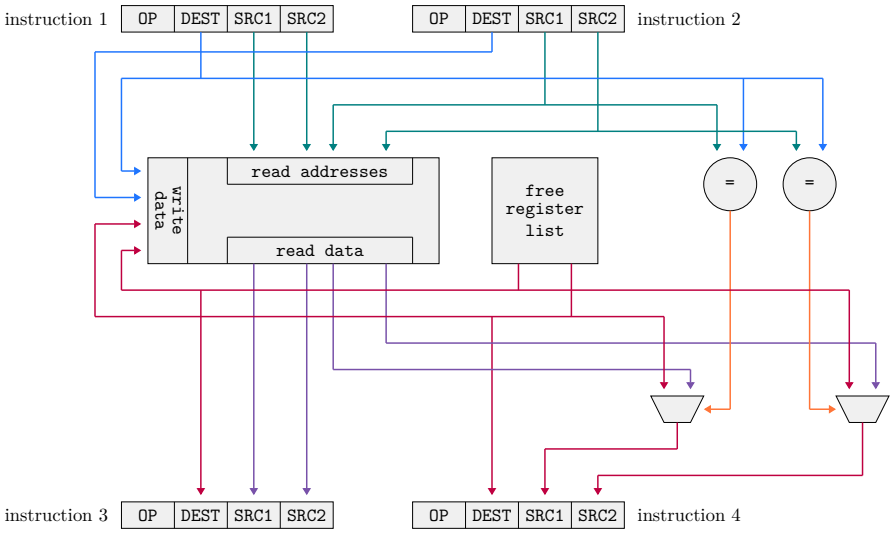


Figure 39: Two issue Superscalar Register Renaming

8 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.

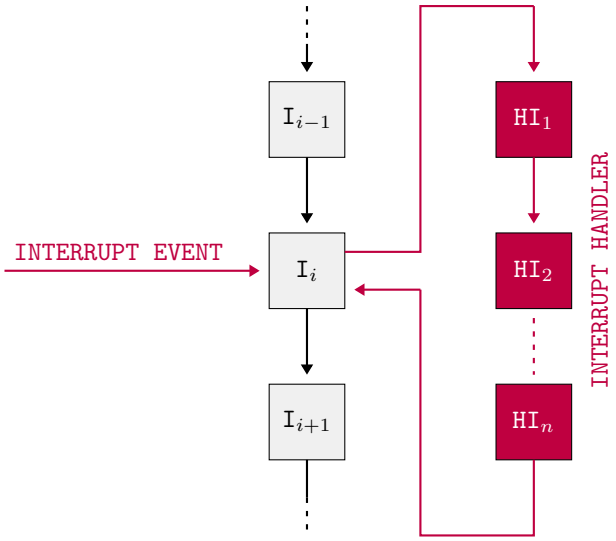


Figure 40: Interrupt event

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**.

8.1 Precise Interrupts

An interrupt or exception is considered precise if there is a single instruction *(or interrupt point)* for which all instruction 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.

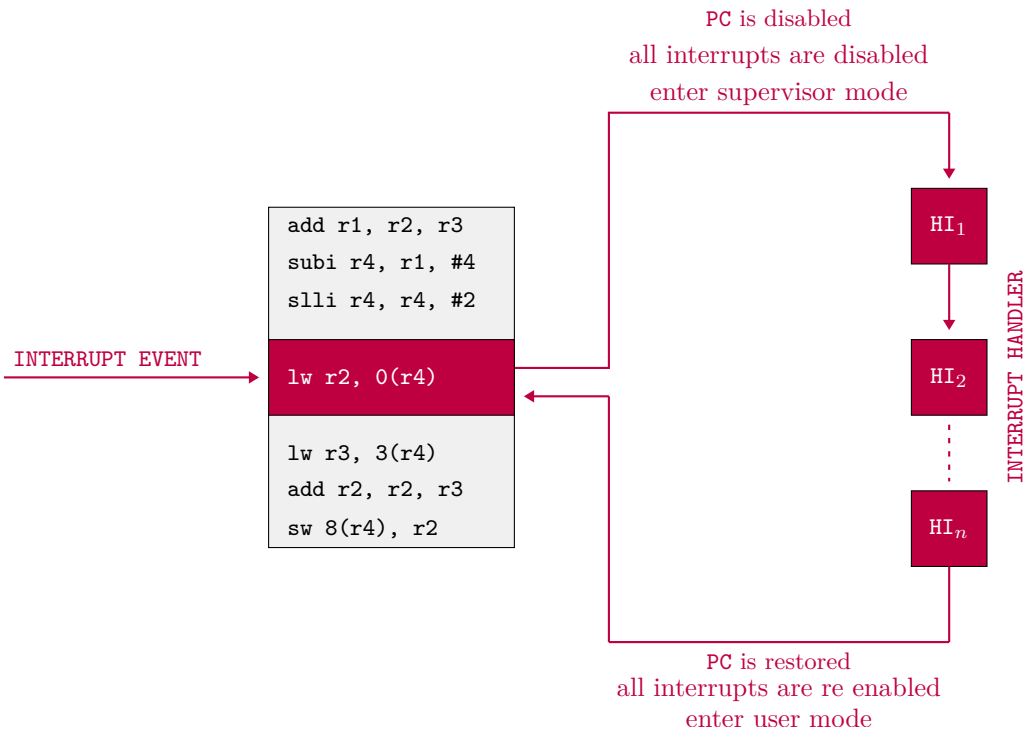


Figure 41: Operation of an interrupt

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 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 41.

8.2 Classes of Exceptions

Exceptions can be divided into classes:

- **Synchronous** and **Asynchronous**:
  - Synchronous 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 handled after the instruction has completed
  - Coerced exceptions are caused by some hardware event not under 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 terminating event, the program execution always stops
    - With resuming events, the program execution continues after the interrupt

8.2.1 Asynchronous Interrupts

The **Asynchronous Interrupts** work by:

1. Invoking the **Interrupt Handler**
    - an *I/O* device request 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**, that 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

8.2.2 Synchronous Interrupts

A **Synchronous Interrupt** is cause 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.

8.3 Precise interrupts in 5 stages pipeline

Exceptions may occur at different stages in 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.
These problem can be solved by tagging instruction in pipeline as *"cause exceptions or not"*, and wait until end of `ME` stage to flax 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 instruction from the interrupt vector
    - this step is complicated by 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 a 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

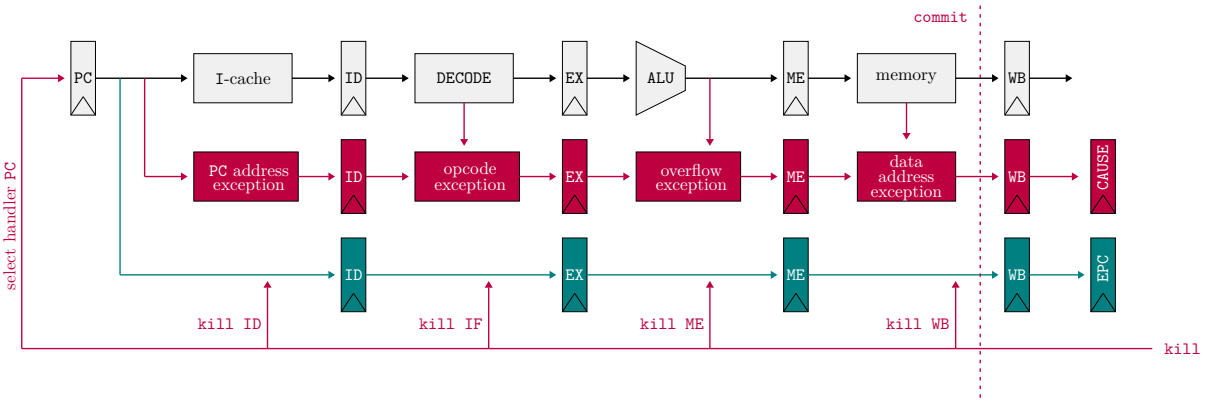


Figure 42: Interrupt handling in 5 stages pipeline - *i'm not sure what this Figure exactly meant to represent*

While dealing with in-order pipeline might be easy, generating precise interrupt when instructions are being executed in 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

8.3.1 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 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 pipeline if flushed

9 MIMD and parallel architectures

A definition of **parallel architectures**, as defined by *Almasi* and *Gottlieb* in 1989 is:

> *"A collection of processing elements that cooperatres and communicated to solve large problems fast."*

The aim of the parallel architecture is to replicate processors to add performance, rather than designing a faster processor; parallel architecture 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.

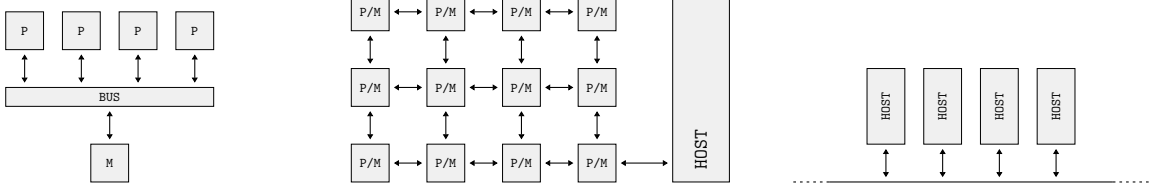
ILP architectures *(like superscalar and VLIW)* support fine grained, instruction level, parallelism but they fail to support large scale parallel systems. Multiple issue CPUs are very complex and thus extracting more parallelism is getting more and more difficult as time goes on due to a steep increase 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 process and thread level. 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 independently

9.1 Example of *MIMD* machines

Examples of *MIMD* machines are:

- **Symmetric** multiprocessor machines
  - **multiple processors** in box with **shared memory** communication
  - every processor runs a copy of the *OS*
  - *represented in Figure 43a*
- **Non uniform** shared memory machine, with separate *I/O* through host
  - **multiple processors**, each with **local memory** and scalable network
  - extremely light *OS* on each node providing simple services
  - network accessible host of *I/O*
  - *represented in Figure 43b*
- **Cluster** machines
  - many **independent machines** connected with general network
  - communication happens through messages
  - *represented in Figure 43c*



(a) Symmetric multiprocessor (b) Non uniform shared memory (c) Cluster

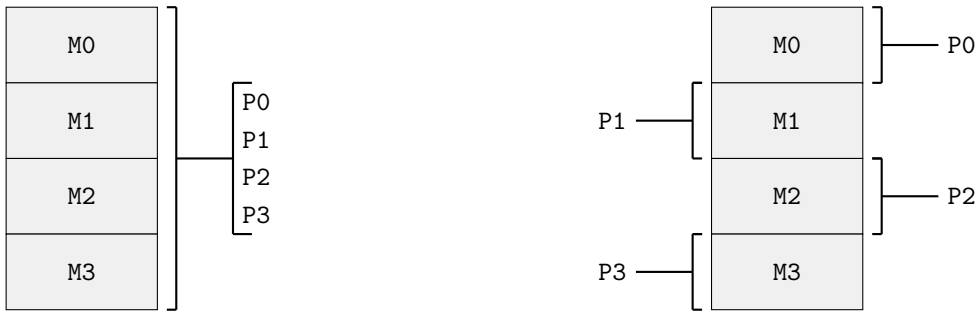
9.2 Memory sharing between processors

MIMD architectures can be divided in 2 **classes**, depending on the number of processors involved; this categorization, in turn, dictates the memory organization an interconnection strategy.

- **Centralized** shared memory architectures
  - at most a **few dozen processor** chips *(less than* 100 *cores)*
  - **large caches**, single memory split in multiple banks
  - often called **Symmetric Multiprocessors** *(SMP)*
  - the architecture is called **Uniform Memory Access** *(UMA)*
- **Distributed** memory architectures
  - able to support **large count of processors**
  - requires high bandwidth **interconnection**
  - data communication among processors slows down the operations

The **Memory Address Space Model** can be divided in two categories:

- **Single** logically shared address space
  - a memory reference can be made by **any processor** to **any memory location**
  - the **address space is shared** among processors
    → the **same physical address** on multiple processors refers to **the same memory** location
  - model used in **shared memory** architectures
  - *represented in Figure 44a*
- **Multiple** and private address space
  - the processors communicate among them through **message passing**
  - the **address space is logically disjoint** and cannot be addressed by different processors
  - → the s**ame physical address** on different processors refers to **different memory** locations
  - model used in **message passing** architectures
  - *represented in Figure 44b*

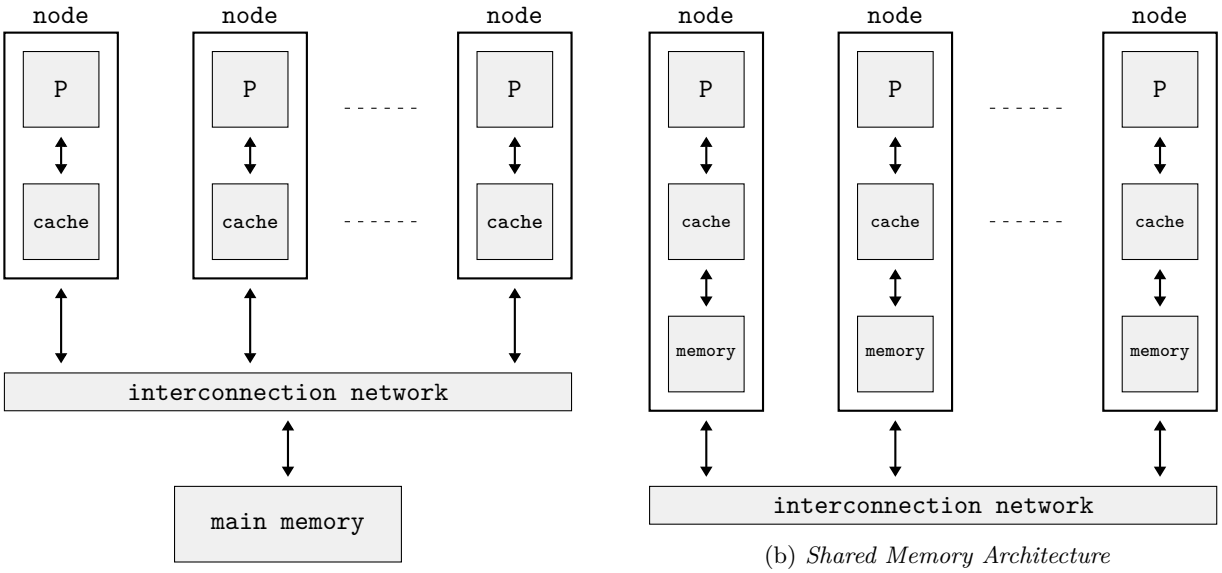


(a) Single address (b) Multiple address

Figure 44: Illustration of single and multiple address spaces. `P0`, `P1`, `P2` and `P3` are processors, `M0`, `M1`, `M2` and `M3` are memory locations.

Furthermore, the physical memory can be organized to be either:

- **Centralized**
  - also called *Unified Memory Architecture*, or *UMA*
  - *represented in Figure 45a*
- **Distributed**
  - also called *Shared Memory Architecture*, or *SMA*
  - *represented in Figure 45b*



(a) *Unified Memory Architecture*

(b) *Shared Memory Architecture*

Figure 45: Comparison between *UMA* and *SMA*

It must be noted that the concept of **addressing space** *(single or multiple)* and the **physical memory organization** *(centralized or distributed)* are **orthogonal** to each other: multiprocessor systems can have single addressing space and distributed physical memory.
The main differences among them are shown in Table 4.

↱	*single address space*	*multiple address space*
communication	through shared variables	through message passing
communication management	implicit	explicit
advantages	no single centralized memory	no cache coherency problems

Table 4: Main differences between single and multiple address spaces

9.3 Communication Models

As mentioned before, the communication between processors can happen either via **shared memory** *(also called shared variables)* or via **message passing**.
Main advantages and disadvantages of each method:

- **Shared memory**
  - ✓ Low latency
  - ✓ Easier to program
  - ✓ Easier to use hardware controlled caching
  - ✓ Most common choice among uni processors and small scale multiprocessors
  - ✓ Implicit communication
  - ✓ Low overhead when cached
  - ✓ Communication happens using `LOAD` and `STORE` operations, with low software overhead
  - ✗ Complex to build in a way that scales well
  - ✗ Requires synchronizing the processors
  - ✗ Hard to control data placement within the cache
- **Message passing**
  - ✓ Less hardware
  - ✓ Easier to design
  - ✓ Focus on local, non costly, operations
  - ✓ Explicit communication
  - ✓ Easier to control data placement due to lack of automatic caching
  - ✗ Message passing overhead can be quite high
  - ✗ More complex to program
  - ✗ Introduces the problem of the reception technique *(polling and interrupts)*

Operations of the two models:

- **Shared memory**:
  - Program is a collection of threads of control, can be created **dynamically**
  - Each thread has a set of private variables and a set of **shared variables**
  - Threads communicate implicitly by writing and reading **shared variables**
  - Threads coordinate by synchronizing on **shared variables**
- **Message passing**:
  - Program is a collection of named processes, fixed at **startup time**
  - There's no **shared data** between processes
  - Logically shared data is **partitioned** over local processes
  - Processes communicate by explicitly **sending** and **receiving** messages
    - the most common software interface to pass message is called Message Passing Interface *(MPI)*

Furthermore, message passing among parallel processors introduces two more issues, more relevant in large scale systems:

1. All data layout must be handled by **software**
   - data cannot be retrieved remotely without explicit communication
2. Message passing has a high software **overhead**
   - early machines had to invoke the *OS* on each message *(spending* $100\,\mu s \div 1\,ms$ *per message)*
   - user level access to network interface has **dozen of cycles** of overhead
   - while sending messages might be cheap, **receiving messages is expensive** due to the introduction of polling or interrupt mechanisms

To try and partially solve those two problems, the **Bus Based Symmetric Shared Memory** has been introduced: it is now the most common communication model in the parallel world. It can be scaled easily to large scale systems and has an high throughput, as it offers:

- **Fine grained** resources sharing

- **Uniform** access via `LOAD` and `STORE` instructions
- **Automatic data movement** and coherency in cache replication
- **Cheap** and powerful extensions
- **Normal uniprocessor mechanisms** to access data, extending memory hierarchy to support multiple processors

9.3.1 Cache Coherency

The cache is fast memory used to reduce latency of access to data: as such it's a valuable resource as can be used both for shared and local data. There are 2 main categories of shared memory machines:

1. **Non cache coherent**
2. **Cache coherent**

Both will work with any data placement, eventually resulting in slow executions, but critical portions of the code can be optimized to be sped up. In large scale systems, the logically distributed shared memory can is implemented as physically distributed memory modules.
The cache, however, needs introduce a new problem: **cache coherency**. Informally, it is described as:

> *"Any write to a cache line must be followed by a read from the same cache line. Furthermore, all writes are seen in proper order"*

2 main categories of shared memory architectures are identified:

1. **Private data**, used by a single processor
2. **Shared data**, used by multiple processors to communicate

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 already mentioned problem of cache coherency.
Consider the following sequence of operations performed by a *shared data SIMD* with 2 processors:

1. Processor `A` **reads** the shared variable `x`, loading it into its **cache**
2. Processor `B` **reads** the shared variable `x`, loading it into its **cache**
3. Processor `A` **stores** `0` into `x`
4. Processor `B` **reads** `x`

What does `B` read? **The two processors see different values for the same shared variable.** This is a common problem with write back caches, because the value written back to memory depends on the circumstances of the access itself: processes accessing main memory may see old values of the same variable.
Alternatively the cache can be bypassed, forcing the processor to read the value from main memory. This solution however slows down significantly the access to memory, increasing the latency and requiring more bandwidth.
At the same time, it's not possible to require that the `READ` of `B` can instantaneously know that `A` has performed a `STORE` operation. This problem is called **memory consistency** and it's complementary to the cache coherency:

- **Coherence** defines the behaviour of reads and writes to the same memory location
- **Consistency** defines the behaviour of reads and writes to different memory locations

The coherency must be preserved while writing and reading data from the cache:

- **Reading does not create issues**, as multiple copies of the same data are kept in the cache
- **Writing** is an operation that must be done in a way that **guarantees coherency**
  - a processor must have **exclusive access** to the cache while writing
  - all processor must **receive the new values** after a `write` operation
  - coherency protocols must **locate all the caches** that share an object to be written
  - a write to a shared data can cause:
    1. **invalidation** of all other copies
    2. **update** of all shared copies

9.3.2 Coherent caches

A program running on multiple processors will normally have copies of the same data in several caches. In a coherent multiprocessor, the cache provides:

- **Migration** of shared data
  - a data item can be **moved to a local cache** and used in a transparent manner
  - the **latency** to access the item **is increased**
  - the **bandwidth** to access the shared memory **is increased**
- **Replication** of shared data
  - a data item is **replicated in several caches**
  - the **latency** to access the item **is reduced**
  - the **bandwidth** to access the shared memory **is reduced**
  - the **contention** to access the shared memory **is reduced**

In order to solve the two problems, a class of hardware based techniques called **Cache Coherence Protocols** has been introduced: it works by tracking the status of any sharing of a data block. There are 2 classes of protocols:

1. **Snooping** protocols
2. **Directory Based** protocols

9.3.3 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 and it's particularly suitable for **Centralized Shared Memory Architectures**, especially for small scale multiprocessors with single `BUS`.

One of the first implementations of this protocol has been proposed by *Goodman* in 1983. The cache snoops upon *DMA* transfers, doing *"the right thing"* when it is necessary:

- 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

A simple example of the hardware implementation of the snooping protocol is represented in Figure 46.

Since every `BUS` transaction check the cache address tags, this operation can interfere with the processor operations, causing stall when the variable is not available in the cache.
To reduce the interference 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**

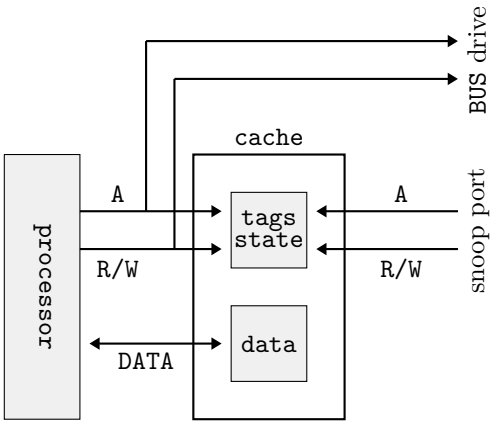


Figure 46: Snooping Protocol

9.3.3.1 Write-Invalidate Protocol

The writing processor issues and 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 block 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.

9.3.3.2 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.

9.3.3.3 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.

9.3.3.4 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 of 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

9.3.3.5 Cache State Transition Diagram

This transition diagram *(represented in Figure 47a)* is applicable to the **MSI Protocol** *(Modified, Shared, Invalid)*. Each cache line has state bits, relative to the state of the data.

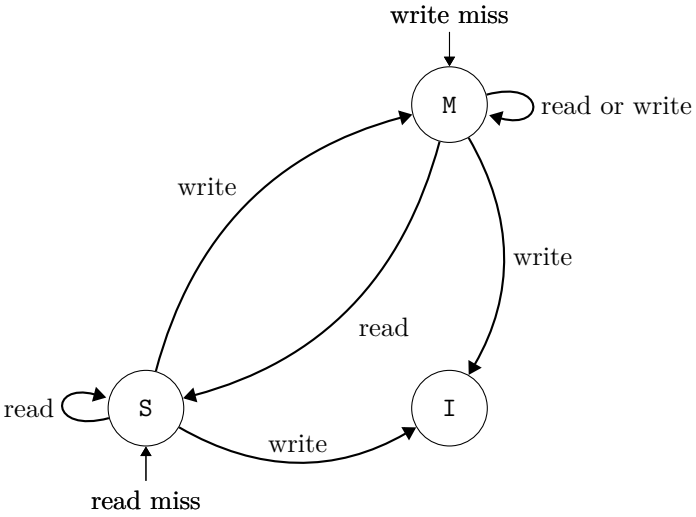
9.3.4 Snoopy Cache Variations

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 processor receive the message
- The system is **hard to extend**
  - **solution**: add an exclusive state to indicate 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 cache
4. **Invalid**: the cache contain invalid data



(a) State Transition Diagram

(b) Structure of a cache line

Figure 47: *MSI* Protocol Implementation

The **exclusive** state distinguishes between **exclusive** *(writable)* and **owned** *(written)* states.
In order 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 to send the invalidation signal on the `BUS`, since no other copies of the block are in cache
- A write to a **shared** block implies the invalidation of the other copies of the block in cache

An illustration of the `BUS` is shown in Figure 48, while the state of the cache lines with *MESI* is represented in Table 5. The transition diagram and relative cache line are represented in Figure 49.

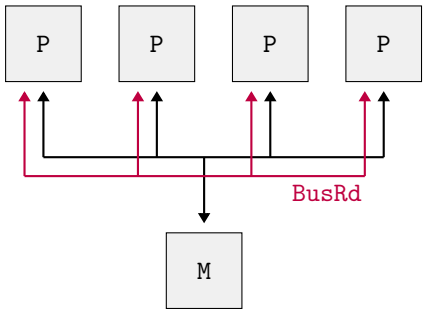


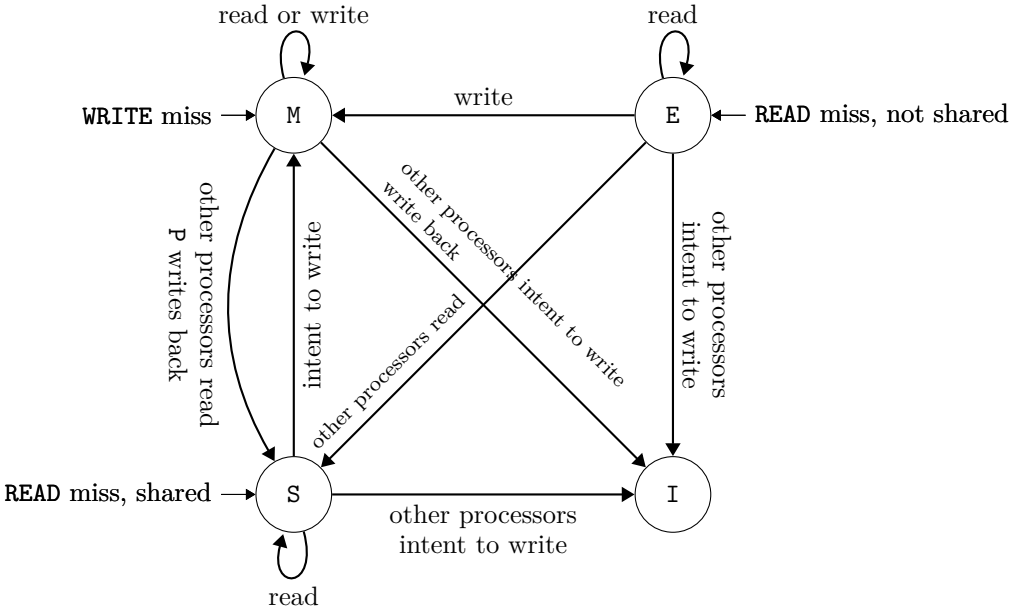
Figure 48: Hardware support for *MESI*

9.3.5 Optimized Snoop with Level-2 Caches

To improve the reading speed (thus reducing the access latency), often processors have two level caches, both on the same chip:

↱	*Modified*	*Exclusive*	*Shared*	*Invalid*
Line valid?	✓	✓	✗	✓
Copy in memory	has to be updated	valid	valid	–
Other copies in other caches?	✗	✗	maybe	maybe
A write on this line	access the `BUS`	access the `BUS`	access the `BUS`, update the cache	direct access to the `BUS`

Table 5: State of cache lines with *MESI* Protocol



(a) State Transition Diagram

M	E	S	I	ADDRESS TAG

(b) Structure of a cache line

Figure 49: *MESI* Protocol Implementation

- A `L2` cache, large
- A `L1` cache, small
  - entries in `L1` must be in `L2`
  - an invalidation in `L2` **implies** an invalidation in `L1`
  - *snooping* on `L2` does not affect `L1` bandwidth

When a `READ` miss for a value is detected in a value, a read request for the value is placed on the `BUS`. The cache containing the value needs to supply and change its status to shared. The main memory may respond to the request as well.

An illustration of this hardware setup is shown in Figure 50

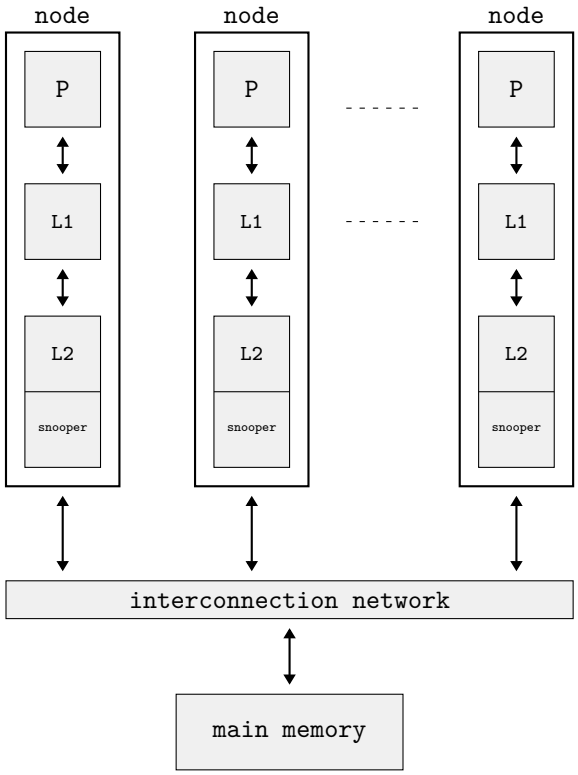


Figure 50: Hardware support for *MESI* with Level-2 Caches

9.3.5.1 False sharing

A cache line might contain more than one value, and cache coherence is not maintained at the line level or at word level.
Suppose processor $\mathtt{P}_1$ writes $\mathtt{word}_i$ and processor $\mathtt{P}_2$ writes $\mathtt{word}_k$ at the same line address. The block may be invalidated many times unnecessarily, since both the addresses share a common block. Such event is called **false sharing**.

An illustration of such a memory block is shown in Figure 51.

M	E	S	I	line address	$\mathtt{word}_i$	$\mathtt{word}_{i+1}$	$\cdots$	$\mathtt{word}_n$

Figure 51: False sharing in *MESI*

9.3.6 Memory consistency

The memory is defined as **consistent** when all caches in each processor read the same value associated to the same shared variables.
Likewise, according to *Lamport*, a system is sequentially consistent if:

> *"The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program."*

The **sequential consistency** *(or SC)* is then defined as the arbitrary order preserving interleaving of memory references of sequential programs.

In order to guarantee the **memory consistency**, the result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved. The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed.

9.3.6.1 Relaxed Memory Models

In order to preserve sequential consistency, all possible read/write orderings must be preserved:

$$R \to W \quad R \to R \quad W \to R \quad W \to W$$

The relaxed models are defined by which of these four sets of orderings they relax:

- **Total store** ordering relaxes $W \to R$
- **Partial store** ordering relaxes $W \to W$
- Relaxing $R \to W, R \to R$ ordering yields a **variety of models** depending on how synchronization operations enforce ordering *(thus releasing consistency)*

Not all dependences assumed by *SC* are supported, so the software has to explicitly insert additional dependences where needed according to the memory model.

9.3.7 Synchronization

The need for **synchronization** arises whenever there are concurrent processes in a system *(even in uniprocessor systems)*. 2 main classes of synchronization are identified:

1. **Producer-Consumer**, a *consumer* process must wait until the *producer* process has produced data
2. **Mutual Exclusion**, only one *process* uses a resource at a given time

The synchronization of two processes makes sense only if they communicate data: consistent view is needed only when one process shares its updates to others, and it's only needed to ensure that each process get updated after they acquire access to the shared data.

9.3.8 Performance of Symmetric Multiprocessors

Performance of cache in **Symmetric Multi Processors** *(SMP)*, is a combination of:

- **Uniprocessor** cache miss traffic
- **Traffic** caused by communication, resulting in invalidations and cache misses
- Coherence **misses** *(communication misses)*

True sharing misses arise from the communication of data through the cache coherence mechanism:

- **Invalidation** due to first write to shared line
- Read by anther CPU on the **same** line in **different** cache

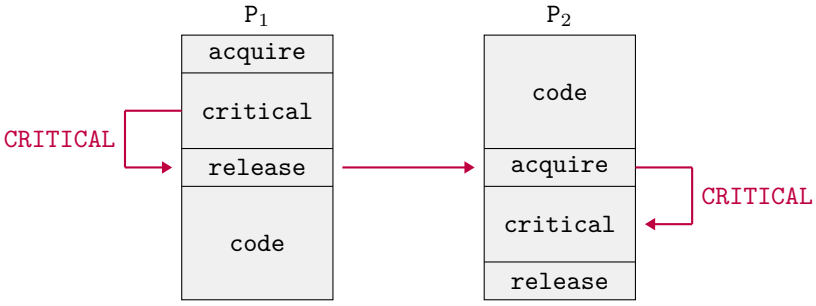


Figure 52: I don't even know what this is meant to be. It was on the slides. I'm just kinda giving up.

- **Miss** occurs if line size were of 1 word

False sharing misses when a line is invalidated because some word in the line, other than the one being read, is written into:

- Invalidation does not cause a new value to be communicated, but only **causes an extra cache miss**
- Line is **shared**, by no word in line is actually shared
- **Miss** would not occurs if line size were of 1 word

9.3.9 Scaling broadcast coherence

When a processor gets a miss, it must probe every other cache. This causes a limit in the scale up, because it implies higher communication traffic and more snoop bandwidth into tags.
The bandwidth can be improved by using multiple interleaved *BUSES* with interleaved tag banks. However they don't scale to large number of connections, so point-to-point networks can be used with large number for large scale systems. The limitation would now be represented by the tag bandwidth while broadcasting snoop request.
In real world applications, most snoops fail to find a match and, as such, there's no scalability with respect to the number of processors.

9.3.9.1 Extension of coherence protocols

Shared memory `BUS` and snooping bandwidth create a bottleneck for scaling symmetric multiprocessors. In order to solve this problem, the following modifications are proposed:

- Duplication of tags
- Directory are placed in the outermost cache
- Crossbars or point to point networks with banked memory are used to connect the cache

An illustration of this extension is shown in Figure 53.

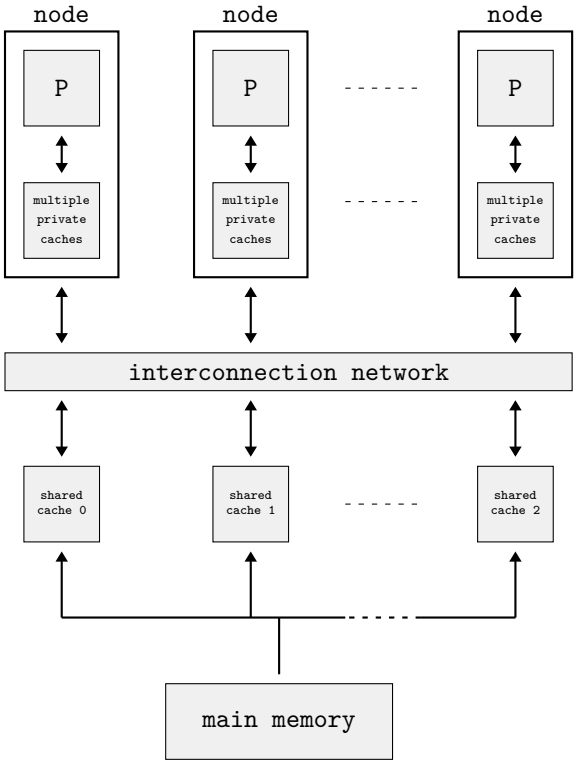


Figure 53: Extension of coherence protocols

10 SIMD

3 variations of the *SIMD* model are commonly used in parallel processors. They are:

- **Vector** architectures
- **SIMD** extensions
  - *MMX* - Multimedia Extension
  - *SSE* - Streaming *SIMD* Extension
  - *AVX* - Advanced Vector Extension
- **Graphics Processor Units** *(GPUs)*
  - heterogeneous architectures: require system processor and system memory in addition to *GPU* and graphics memory

10.1 SIMD vs MIMD

SIMD architectures can exploit significant data level parallelism for:

- **matrix-oriented** scientific computing
- **media-oriented** image and sound processors

SIMD is generally more energy efficient than *MIMD*, as it only needs to fetch one instruction per data operation. This feature makes *SIMD* attractive for personal mobile devices. Finally, *SIMD* allows programmers to continue to think sequentially, as opposed to *MIMD (where programmers need to think in parallel)*.

10.2 Resurgence of DLP

The convergence of application demands and technology constraints drivers the architecture choice. New applications, *(such as graphics, machine vision, speech recognition, machine learning, ...)* require large numerical computations that are often trivially data parallel. *SIMD* based architectures *(such as vector-SIMD, subword-SIMD, SIMT, GPUs)* are the most efficient choice for these applications.
It is widely accepted that *DLP* will account for more mainstream parallelism growth than *TLP* in the next decade.

10.3 Supercomputers

Definition of a supercomputer:

- Fastest machine the world at a given task
- A device that can turn a compute bound problem into a I/O bound problem
- Any machine costing $\$30M+$ capitalism is a blessing in disguise
- Any machine designed by *Seymour Cray* damn son what a chad

The *CDC600*, a computer designed by *Seymour Cray* and built by *Cray Research* in 1964, is regarded as the first supercomputer.

10.4 Vector Architectures and Vector Processing

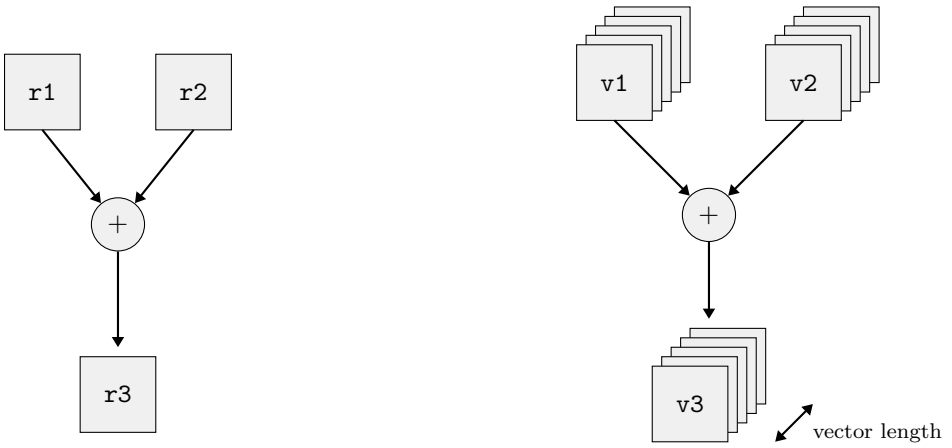
Vector processors have high level operations that work on linear arrays of number *(also called vectors)*. A language that can handle vectors *(and not scalar values)* is needed as well.

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-bits elements, 32 128-bits elements, ...

- A single instruction operates on **vectors** of data
  - the result involves many registers to register operations
  - used to hide memory latency
  - leverages memory bandwidth

It's called **stride** of an array *(also referred to as increment, pitch or step size)* the number of locations in memory between beginnings of successive array elements, measured in bytes or in 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.



(a) Scalar model, 1 operation at a time

(b) Vector mode, n operations at a time

Figure 54: Comparison of scalar and vector models

10.4.1 Vector processing

A vector processor consists of a pipelined scalar unit and a 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`)*

The execution is done by using a **deep pipeline**, allowing very fast clock frequency and higher speeds; since elements in the vectors are independent, there are no hazards and the pipelines are always full.
Vectors applications are not limited to scientific computing, as they are used in:

- Multimedia Processing
- Standard benchmarks kernels
- Lossy and Lossless Compression
- Cryptography and Hashing
- Speech and handwriting recognition
- Operating systems and networking
- Databases
- Language run time support

The structure of the Vector Unit is represented in Figure 55.

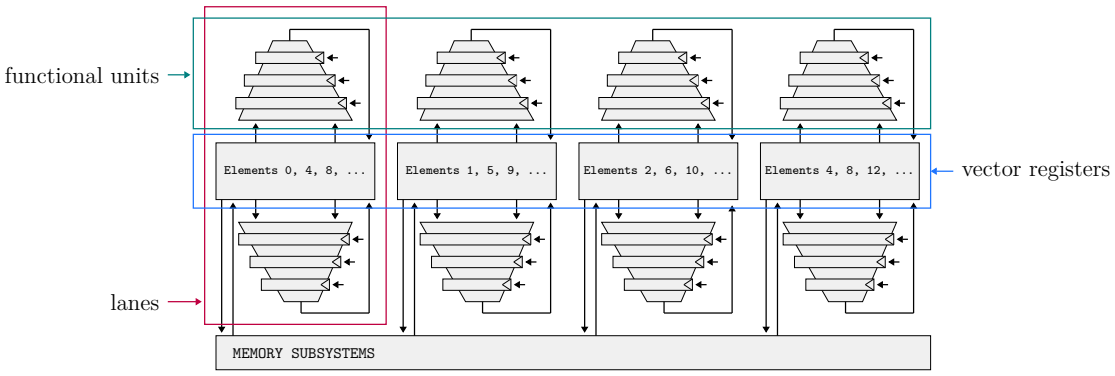


Figure 55: *Vector Unit* structure

10.4.2 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**

Example of vector code:

```
// 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
```

10.4.3 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.

10.4.3.1 Vector startup time

The vector unit has a startup time penalty. It is composed by two factors:

- **Functional unit** latency *(time through the pipeline)*
- **Dead** time or recovery time *(time before another vector instructions can start travelling the pipeline)*

The representation of the pipeline during startup is shown in Figure 56.

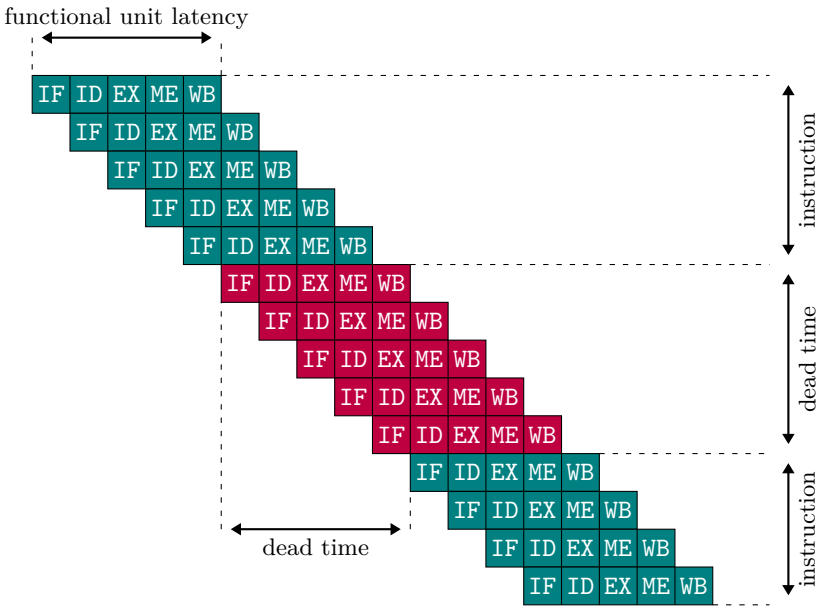


Figure 56: Vector Unit startup time penalty

10.4.4 Interleaved Vector Memory System

The access to each item in a memory bank is interleaved. The bank busy time is defined as the time before bak is ready to accept the next request.
The *Cray-1* had 16 banks with 4 cycles bank busy time and 12 cycles latency.

10.4.5 Automatic code vectorization

The code vectorization is a process that automatically detects the best way to reorder the operation sequencing. It is executed at compile time via heuristic algorithms and it requires extensive loop dependence analysis.

10.4.5.1 Stripmining

Problem: vector registers have **finite length**.
Solution: **break loops** into pieces that fit in registers. This technique is called **stripmining**.

Code example:

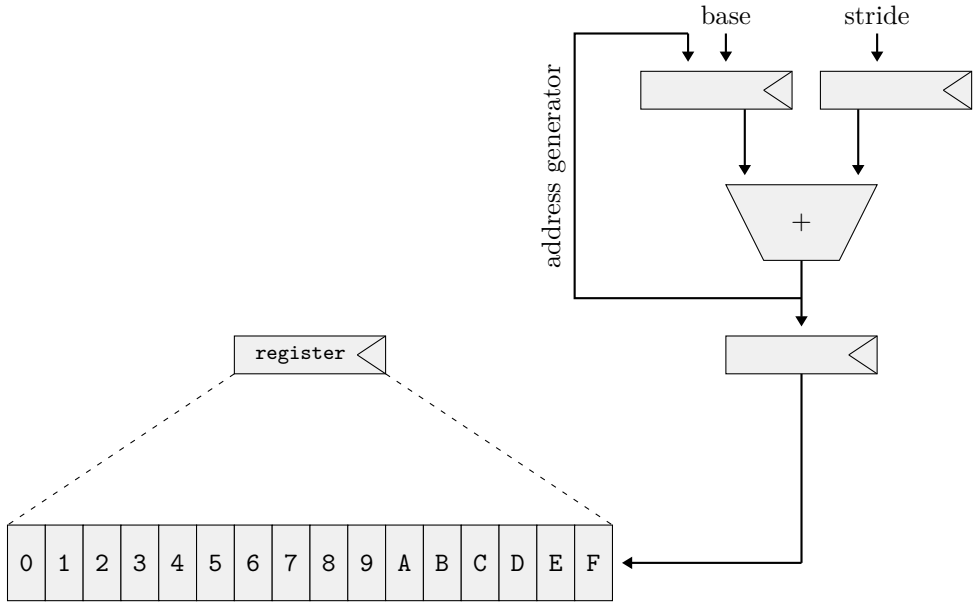
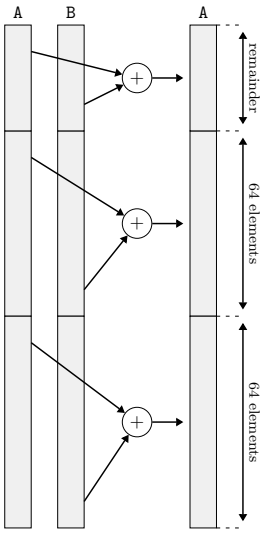


Figure 57: Interleaved Vector Memory System

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

```
andi x1, xN, 63 // N mod 64
setvlr x1 // Do remainder
loop:
vld v1, xA
sll x2, x1, 3 // Multiply by 8
add xA, x2 // Bump pointer
vld v2, xB
add xB, x2
vfadd.d v3, v1, v2
vsd v3, xC
add xC, x2
sub xN, x1 // Subtract
li x1, 64
setvlr x1 // Reset full length
bgtz xN, loop // loop again
```



10.4.5.2 Vector Conditional Execution

Problem: **vectorize** loops with conditional code.
Solution: add **vector masks** to registers:

- vector version of predicate registers, 1 bit per element and maskable vector instructions
- vector operations become bubble *(or `NOP`)* at elements where mask bit is clear

Two different implementation can be used to execute the vector instructions:

- Simple implementation: execute all N operations, turn off result writeback according to mask
- Density time implementation: scan mask vectors and only execute elements with non zero mask

Code example:

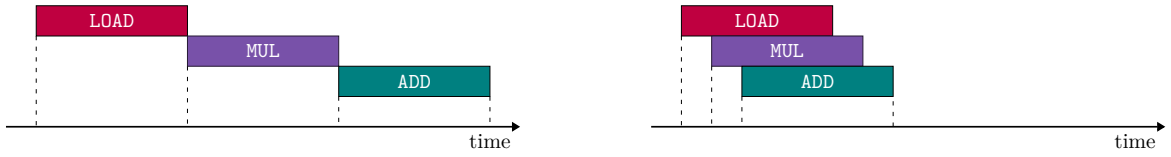
```
for (i = 0; i < N; i++)
  if (A[i] > 0)
    A[i] = B{i}
```

```
cvm // Turn on all elements
vld vA, xA // Load entire A vector
vfsgts.d vA, f0 // Set bits in mask register where A > 0
vld vA, xB // Load B vector into A under mask
vsd vA, xA // Store A back to memory under mask
```

10.4.6 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 decrease
- **The operations are executed in parallel**, along multiple lines

The advantage of vector chaining is represented in Figure 58.



(a) without vector chaining

(b) with vector chaining

Figure 58: Comparison of instruction executions with and without vector chaining

10.4.6.1 Vector Memory-Memory vs Vector Register Machines

- Vector memory-memory architectures *(VMMAs)* require greater main memory bandwidth because all operands must be read in and out of memory
- *VMMAs* make it difficult to overlap execution of multiple vector operations because dependencies on memory addresses must be checked
- *VMMAs* incur in greater startup latencies
- All major vector machines since Cray-1 have had vector register architectures

10.4.7 Multimedia Extension

The **multimedia extension**, also called *SIMD* extension. It features:

- very short vectors **added** to existing *ISAs* for microprocessors
- existing 64 bits registers **split** into $2 \times 32\,\texttt{b}$ or $4 \times 16\,\texttt{b}$ or $8 \times 8\,\texttt{b}$
- execution of the single instructions on **all the elements within the register** *(Figure 59)*

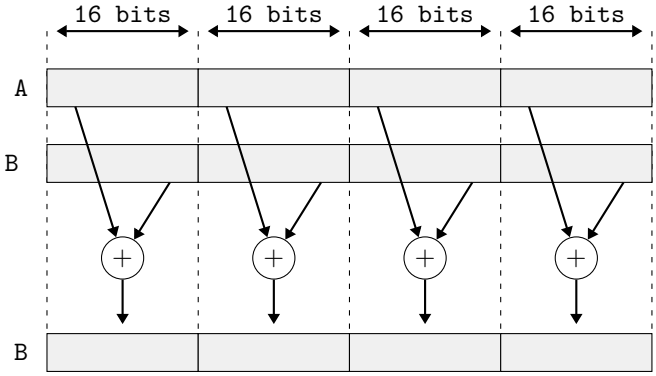


Figure 59: Operation of instructions in vector form, `A + B = C` where all the variables are 4 16 bits vectors

10.4.7.1 Multimedia Extensions vs Vector Architectures

Multimedia Extensions, compared to Vector Architectures, feature:

- **A limited instruction set**
  - no vector length control
  - no strided `LOAD` and `STORE` or `SCATTER` and `GATHER`
  - no unit-stride loads must be aligned to 64/128 bit boundary
- **A limited vector register length**
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increase register pressure
- **Trend towards fuller vector support in microprocessors**
  - better support for misaligned memory access
  - support of double precision *(64 bits floating point)*

10.5 Vector Computers recap

- Vectors provide **efficient execution** of data parallel loop codes
- Vector ISA provides **compact encoding** of machine parallelism
- Vector ISA **scales to more lanes** without changing binary code
- Vector registers **provide fast temporary storage** to reduce memory bandwidth demands and simplify dependency checking between vector operations
- Scatter/gather, masking, compress/expand operations **increase set of vectorizable loops**
- Requires **extensive compiler analysis** *(or programmer annotation)* to be certain that loops can be vectorized
- Full **long vector support** is still **only in supercomputers**, while microprocessors have limited packed or subword-*SIMD* operations support

10.6 Graphic Processing Units - *GPUs*

Originally *(mid-1990s)*, *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 million 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.

10.6.1 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 through shared memory space private to the block
- A **grid** is a set of thread block 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

10.6.2 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 together for *SIMD* execution on 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 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 *(technique already explored in Paragraph 10.4.5.2)*; 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 in order to be executed later.

An illustration of the hardware execution model is shown in Figure 60.

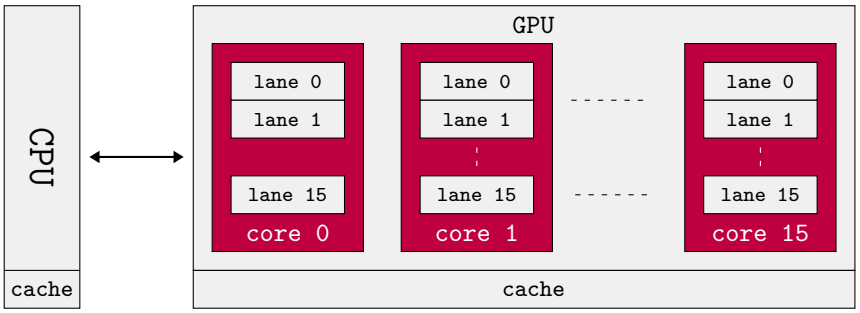


Figure 60: Hardware execution model