POLITECNICO
DI TORINO

Recap Computer Architectures (02LSEOV)

Jacopo Nasi
Computer Engineer
Politecnico di Torino

I Period - 2017/2018

11 dicembre 2017

Indice

License

Acknowledgments

Questo breve riepilogo non ha alcuno scopo se non quello di agevolare lo studio di me stesso, se vi fosse di aiuto siete liberi di usarlo.
Le fonti su cui mi sono basato sono quelle relative al corso offerto (**Computer Architectures (02LSEOV)**) dal Politecnico di Torino durante l'anno accademico 2017/2018.
Non mi assumo nessuna responsabilità in merito ad errori o qualsiasi altra cosa. Fatene buon uso!

1 Introduction Computer Design

1.1 Computer Evolution

The first general-purpose computer was created in the late 40s. What now we can buy for 500$ is equivalent (performance) to what could be bought for about $1M in 85'.
During the years the performance growth was not linear, as you can see in figure 1, during the first 10 years the annual increase was around 25-30%/year, from the late 80s to the 2000 the growth is increased around 50%/year and, in the last few years it decrease to the 22%. Why this change during the increase?
The manufacturers have found a lot of physical problem related to the creation

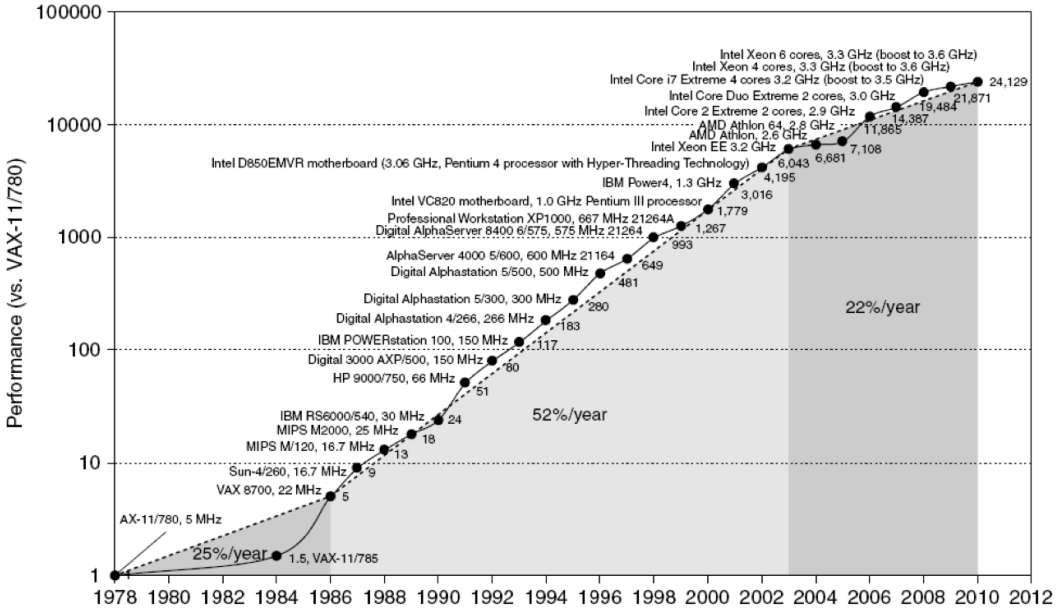


Figura 1: CPUs Growth over years

of new products, this problem are mainly related to:

- Power-Issue.
- Lower instruction-level parallelism.
- Unchanged memory lantecy.

in fact, since 2004, the major industry have changed the conceptual ways to desing processors, switching from single to multi-core architectures. We can say that, in anytime, this growth is incredible an is due to improvements in technology, microprocessor architecture and software development. Since the multi core introduction the major prefers to investe on multicore system rather than develop faster CPU.

1.2 Designing

There are 5 main market areas:

- **Personal Mobile Device** (PMD): Smarthphone, tablet. They are focused in energy efficency and real-time app.
- **Desktop Computing**: From PC to workstation and the main pourpose is optimize the price-performance ratio.
- **Server**: Larger-scale and more reliable computing services.
- **Cluster - WAS**:Emphasis on availability, price-performance and power consumption.
- **Embedded Computers**: Fastest growing portion of PCs market. All special-purpose computer-based application, from cheap to high-end processors.

There are to **Classes of Parallelism**:

- **Data-Level** (DLP): Many data items that can be operated on at the same.
- **Task-Level** (TaskLP): Many task of a work can operate independently.

The first solution allow the processor to split the data operation over multiple cores, you can for example divide an array of n elements over 4 core and, if T is the computational time needed for the entire array, the final time will be T/4 plus a little time for the reunion of the data. It split the one task on different data.
The TLP instead is able to manage multiple task over the same data, this is the common behavior of the actual system (pipelining techniques).
There are different Parallel Architectures:

- **Instruction-level** (ILP): The is modestly use the data-level parallelism.
- **Vector and GPU**: It exploits DLP.
- **Thread-level** (TLP): It exploits DLP and TaskLP.
- **Request-level** (RLP): Exploits parallelism among decoupled (not-related) tasks.

The designing of a new computer involve important analysis to the main pourpose of it, you need to study which attributes are important for the new machine and you need to maximizes performance and matching cost and power constraints. During the last decade the PC design took advantage of architectural and technology improvements, the performance increase is more than a

factor of 15 on what would have been obtained by relying solely on technology.

The **Moore's Law** says that: *The number of transistors that can be integrated into a single chip doubles every 18/24 months.* Until now the law has worked, as you can see in the figure 2 and, probably, it will work for other time.

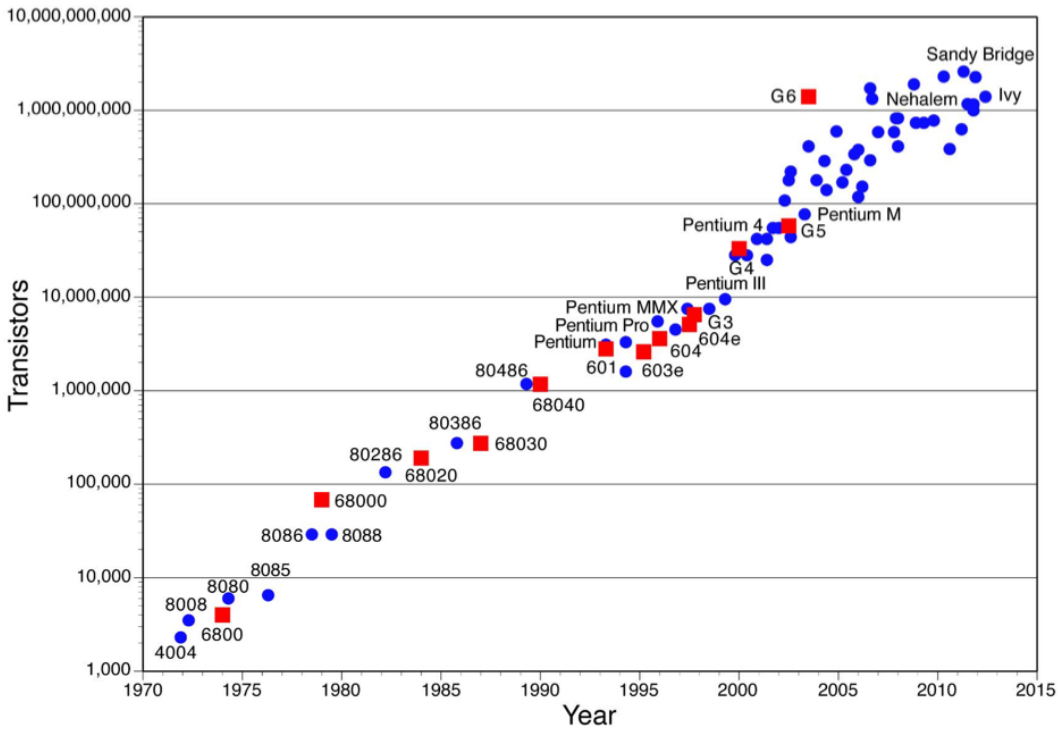


Figura 2: Transistors growth on CPUs

Cost During the evaluation of the IC manufacturing cost it is important not to forget the impact of yeild, the percentage of products that pass the test phase. The production process for every product undergoes an evolution which normally leads to an improvement in yield (learning curve). The cost decrease is due to yield increase. Probably the most important part of the manufacturing cost is related to the validation and testing procedures.

Other designing problem The continuos increase of the system complexity and device integration leads to porblem with power consumption, for the Power and for the Energy (manly for portable devices). Until now the greater power consumption contribution is due to the trasnsistor switching phase, related to the physical formulas the solution applied is trying to reduce the voltage.
Another important factor is the **Dependability** that is the quality of the

system to deliver a correct service, is traditionally very high, but it can be lowered by software or hardware bugs both during the production or in designing phase. During this years the safety-critical areas where the microprocessor have relevant importance are increased, initialy only space, avionics and nuclear, but now we have rail-road, automotive, biomedical, telecom, ecc...
The dependability is often measured using:

- **Mean Time To Failure** (MTTF): 1 failure in one billion hours.
- **Mean Time Between Failures** (MTBF)
- **Mean Time To Repair** (MTTR)

This three measures are related by the formula: $MTBF = MTTF + MTTR$

Performance they be analyzed under multiple point of view, *Time between start and completion of an operation, Total amount of work done in time unit, ecc....* The UNIX system provide 4 different values:

- Elapsed time
- CPU Time
  - User
  - System

The evaluation of often performed letting work a computer and observing its behavior. Unfortunately, the choice of the application severely affects the performance and is not too easy looking the result in a correct way. The main solution is using benchmarks to compare different system running the same operation and production comparable times of execution. The benchmark sets are normally composed of:

- Kernels
- Program Fragments
- Applications

There are multiples solution to analyze the performance one of the most common is the **Amdahl's law** and it based to the comparisons with the older version of the same product. The speedup is calculated by this formula:

$$Speedup = \frac{Performance_{enhancement}}{Performance_{NOenhancement}} \tag{1}$$

the value depends on two factors:

- $Fraction_{Enhanced}$: the fraction of the computation time that take advantages of the enhancement.

- $Speedup_{Enhanced}$: the suze of the enhancement on the part it affects.

A more complete formula is:

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{Enhanced}) + \frac{Fraction_{Enhanced}}{Speedup_{Enhanced}}} \tag{2}$$

an example could be useful:

An enhancement makes one machine 10 times faster for 40% of the programs the machine runs. Which is the overall speedup?
Fraction = 0.4, Speedup = 10.

$$Speedup_{overall} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56 \tag{3}$$

Another important designing evaluation is measuring the time required to execute a program, the are severals approaches:

- Observing the real system: Not easy to be evaluated.
- Simulation: It could be really expensive.
- CPU Equation.

the latest solution use a provided formula to evaluate the CPU time:

$$CPU_{Time} = CLK_{Time} * \sum_{i=1}^{n} CPI_i * IC_i \tag{4}$$

where:

- CPI_i: Number of clock cycles required by instruction i (depends on hardware and instr set).
- IC_i: Number of times instruction i is executed in the program (depends instruction set and compiler).
- CLK_{Time}: Inverse of clock frequency (depends technology).

in the pipelined processor, CPI_i may vary for a given instruction, therefore the evaluation becomes much harder.

2 Instruction Set Principles

2.1 Introduction

The Instruction Set Architecture (ISA) is hoe the computer is seen by the programmer or the compiler. There are different kind of design and they can be selected by different characteristics. The CPUs are often classified according to the type of their internal storage:

- Stack: Is the simplest one it can't accessing memory during operations.
- Accumlator: Similar to 8051 can access the memory during operation.
- Registers
  - Register-memory: Similar to 8086, have a 16-bit register.
  - Register-Register (load-store)
  - Memory-memory (no real cases)

Nowadays all processors are General-Purpose Register (GPR) this because registers are faster than memoery and are easier for a compiler to use. The CPUs are also classifiable by the number of operant per ALU instruction (2 or 3) and number of memory operand per ALU.

2.2 Memory

The memory has a fundamental work in the CPU world, there are several different type of it and of course different type to how accessing it.
There are many different **memory addresses** that can be implemented:

- **Little Endian**: Byte with lower address at the least significant position. The addresses of the data is that of the LSB.
- **Big Endian**: Byte with lower address at the most significant position. The addess of the data is that of the MSB.
- **Aligned**: Allowing only aligned accesses to memory could is a limitation in terms of performance.
- **Misaligned**: This solution require hardware and performance overhead.

A memory position can be accessed in different ways:

- **Register**: (ADD R4, R3) when a value is in a register.
- **Immediate**: (ADD R4, #3) for constants.
- **Displacement**: (ADD R4, 100(R1)) accessing local variables.
- **Deferred or Indirect**: (ADD R4, (R1)) accessing using a pointer or a computed address.
- **Indexed**: (ADD R3, (R1 + R2)) useful in array addressing: R1=array base, R2=index.
- **Direct or Absolute**: (ADD R1, (1001)) useful for accessing static data.

- **Indirect**: (ADD R1, @(R3)) if R3 is the addr. of a pointer p, then the mode is like p.
- **Autoincrement or decrement**: (ADD R1, (R2)±) useful for stepping through array within a loop.

Is important to choosing the correct addressing mode, one can obtain some important consequences, from reduing the number of instruction to increasing the architecture.
There are a lot of instruction that can be done, looking at the 80x86 frequency we can see that most most used are *Load, Conditional Branch, Compare and Store* that are around the 70% of the total operation performed.
The instruction set encoding depends on which instruction compose the instruction set and which addressing modes are supported. When a high number of addressing modes is supported, and address specifier is used to specified the addressing mode. When the number of addressing modes is low, they can be encoded together with the opcode. The different encoding are:

- **Variable**: Any number of operands, operation with variable length, lower performance and minimum code size.
- **Fixed**: Fixed number of operands, need address specifier, fixed instruction length, maximum performance but larger code size.
- **Hybrid**: Multiple format specified by the opcode and it allows a trading-off between code size and performance.

Assembly-level programs are now produced by compilers only. The CPU designer and the compiler writer must interact and cooperate. A crucial part taken by the compiler is the optimization of the register using, it easier to solve it when the number of register is higher (>16).

3 Pipelining

3.1 Introduction

Pipelining is an a solution to execute multiple instructions at the same time overlapping it. The different units (also called *stage* or *segment*) are completing different part of different instruction in parallel. The figure 3 is an example of this techniques. The **throughput** of a pipelined processor is the number of instructions which exit the pipeline in the time unit. All the pipeline stages are synchronized (they proceed to executing the new task all together); the necessary time for executing one step is called machine cycle and, normally, corresponds to one clock cycle. Of course the duration of a machine cycle is determined by the slowest stage. CPI means clock cycles per instruction. The ideal pipeline have all stages perfectly balanced, in this way the $TRP_{pipelined} = TRP_{unpipelined} * n$ where n is the number of pipeline stages.

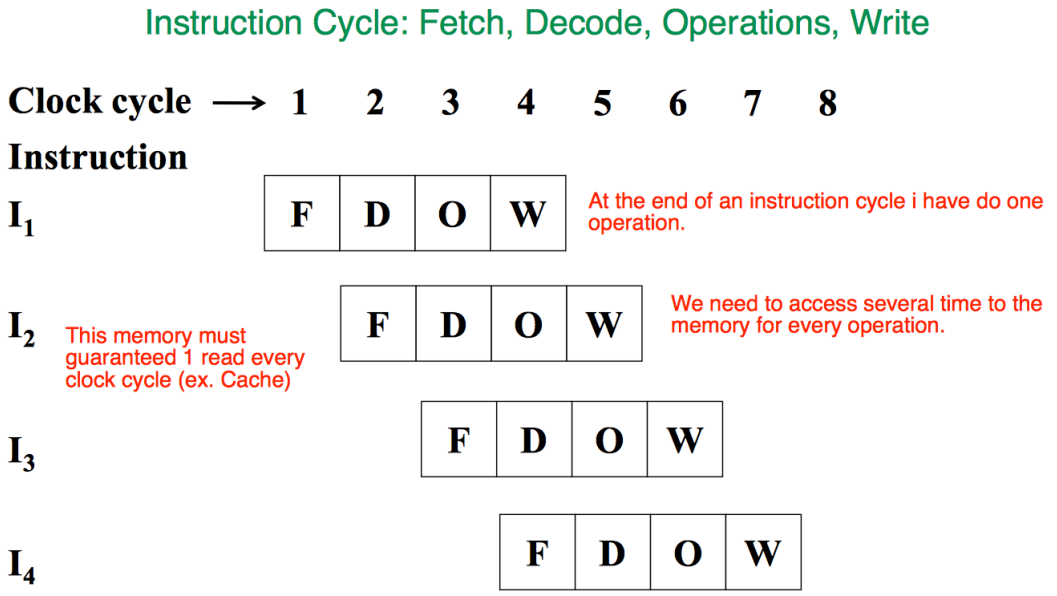


Figura 3: Pipeling example

3.2 behavior

The execution of each instruction may be composed of at most five clock cycles:

- **IF**: Fetch the instruction from the memory address of the PC and save it in the IR.
- **ID**: Decode the instruction just fetched.
- **EX**: Execute the instruction.
- **MEM**: Memory access and brach completion.
- **WB**: Write-back.

with the unpipelined processor all instruction require 5 CC, the only optimization to reduce the average CPI is completing the ALU instruction during the MEM cycle, hardware resources could be optimized avoiding duplications.
The pipelined version instead can be more powerful beacause a new instrcution is started at each clock cycle and different resources work on different instruction at the same time. There are several consideration to keep in mind during the development of this units; at every clock cycle, each resource can be used for one purpose only, this means that:

- Separate instruction and data memories must be used.
- The register file is used in two stages: for reading (second half of CC) in ID stage and for writing (first half od CC) in WB stage.

- The PC (*Program Counter*) must be changed in the IF stage, What about branches?

A little view of the system behavior in figure 4.

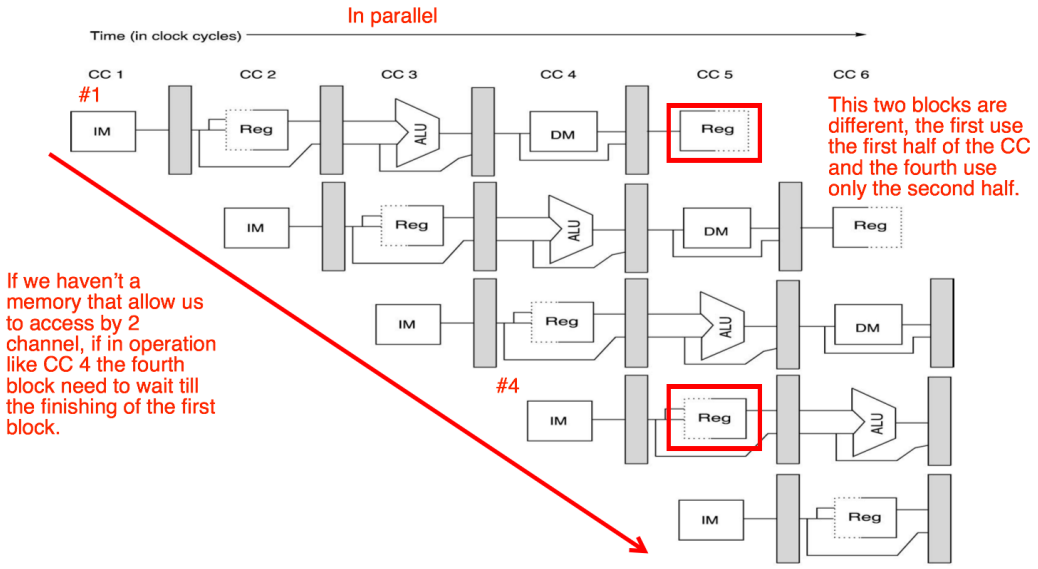


Figura 4: Evolution in time of Pipeling architecture

Performance The first puorpose of this architectural type is the performance increase without making single instruction faster. The single instruction are maded slower for the overhead introduced by the pipeline control. The limitations of a pipeline come from the necessity of balanced stages and pipeling overhead (pipeline register delay and clock skew).

3.3 Hazards

The hazards are situations that prevent the execution of an instruction during its designated clock cycle. There are 3 classes of hazards:

- **Structural**: Coming from resource conflict.
- **Data**: And instruction depends on the result of a previous instruction.
- **Control**:Depend on pipelining branches and other instructions that change the PC.

One way to dealing with hazards is to force the pipeline to stall, i.e., to block instructions for one or more clock cycles. When and instruction is stalled:

- The instructions following the stalled instruction are also stalled.
- The instructions preceding the stalled instruction continue.

A stall causes the introduction of a bubble in the pipeline.

Structural hazards they may happen when some pipeline unit is not able to execute all the operations scheduled for a given cycle. Example:

- A given unit is not able to complete its task in one clock cycle (division).
- The pipeline owns only one register-file write port, but there are cycles in which two register writes are required.
- The pipeline refers to a single-port memory, and there are cycles in which different instructions would like to access to the memory target.

Solving this problem implies adding or improving the hardware.

Data hazards overlapping the execution of instructions, as it is done by pipelining, changes the order of read/write accesses to operands. This can lead to wrong results and undeterministic behavior.
An example could be:

```
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
```

The second instruction will read the value of R1 before the first instruction can write the new value. This will cause a wrong result. And is the same of course for the third instruction.
If an **Interrupt effects** occurs during the execution of a critical piece of code (from the pov of data hazards) correctness may be restored. This may causa an undeterministic behavior.
There are two possibile solution for data hazards problems:

- The first is using the stall till the data will be available.
- Implementing the **forwarding** technique.

Forwarding A special hardware in the datapath detects when a previous ALU operation should write the register corresponding to the source of the current ALU. In this case, the hardware, selects the ALU result as the ALU input rather than the value from the register file. This part must be able to:

- Forward a data from any of the previously started instructions (the data can't be already writed in its final location).
- Not to forward anything, if the following instruction is stalled, or an interrupt occurred.

an example can be viewed in the figure 5.
The forwarding can be implemented:

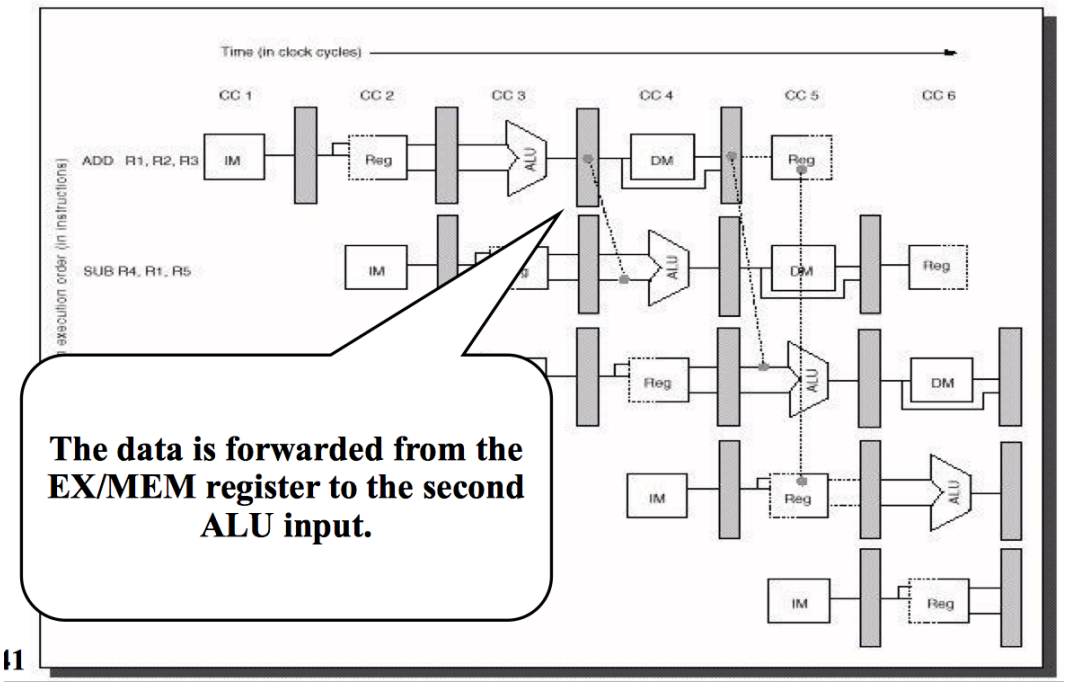


Figura 5: Forwarding Example

- From the ALU or data memory output...
- ...to ALU inputs, data memory inputs, or the zero detection unit (branch instruction).

and the logic must compare:

- The DEST fields of the IR in the EX-MEM and MEM-WB register with...
- ... the SRC fileds of the IR in the IF-ID, ID-EX and EX-MEM registers.

Causes of Data Hazards A hazards is created whenever there is dependence betweem instructions, and they are close enough that the overlap caused by pipeling would change the order of access to an operand. In general, this can happen for:

- Register Operands
- Memory Operands:
  - Acceses to memory by load and store are not made in the same stage.
  - execution can proceed while an instruction waits for a cache miss to be solved.

The forwarding not solve all potential data hazards, in this case to only solution is to use stalls.

Stalls At each clock cycle all test for detecting data hazards are performed (ID stage), if an hazard is detected two actions can be taken:

- The apporpriate forwarding is enabled.
- The instruction is stalled before entering in the wrong stage.

This situation can be performed in two ways:

- Forcing all 0s in the ID-EX pipeline register (*nop* instruction).
- Forcing the IF-ID pipeline register to maintain the current value.

Control Hazards are due to branches, which may change the PC after the following instruction has been fetched already. In the case of conditional branches, the decision on whether the PC should be modified or not can be taken even later. In the MIPS implementation, the PC is written with the target address (if the jumo is taken) at the end of the ID stage.
A possible solution is based on stalling the pipeline as soon as a branch instruction is detected (ID stage) by:

- Decide earlier whether the branch has to be taken or not.
- Compute earlier the new PC.

Performance there are several techniques for reducing the performance degradation due to branches:

- **Freezing pipeline**: the pipe is stalled until the decision about the branch is know.
- **Predict untaken**: Assume the branch is not taken, avoid any change in the pipe, undo all performed if the branch is taken. *
- **Predict taken**: If the target address is know before the branch outcome, it may be possible to assume the branch as taken. *
- **Delayes branch**: Filling the slot after the branch instruction (branch-delay slot) with instructions which have to be executed no matter the branch outcome. Is a compiler task. Less used nowadays.

* This thecniques can be speeded by the compiler which maximizes the chance for the processor to make the right prediction.

3.4 Exception

The execption are fundamental for system. The are a lot of possible causes:

- I/O device request
- System call by user program
- Tracing instruction execution (like debug mode)
- Breakpoint
- Overflow and underflow
- FP anomaly
- Page fault
- Misaligned memoery accesses
- Undifined instruction
- Hardware malfunction
- Power failure

they can be calssified based on the type:

- **Synchronous - Asynchronous**: Same position in code - External devices.
- **User requested - Coerced**: Like procedures - Out of user control.
- **User maskable - Non maskable**: Force hardware not to answer to exception requests.
- **Within - Between instructions**: Generated by instruction itself.
- **Resume - Terminate**: Terminate or execute something than resume.

There are some machine, called restatable, the are able to handle an exception, save the state, and restart without affecting the execution of the program. Nowadays all processors are restatable.

Stopping execution when and exception occurs, the pipeline must execute the following steps:

- Force a trap instruction into the pipeline on the next IF stage.
- Until the trap is taken, turn off all writes for the instruction that raised the execption and for all the following instructions in the pipeline.
- When the exception-handling procedure receives control, it immediately saves the PC of the faulting instruction.

after the exception has been handled, special instructions return the machine from the exception by reloading the PC and restarting the instruction stream. A processor has precise execeptions if the pipeline can be stopped so that:

- The instructions just before the faulting instruction are completed.
- The instructions following the faulting instruction ca be restarted from scratch.

It could be really hard restarting after exception, but precise execption is a must for most architectures, at least for integer.

Contemporary exception for example we can have 2 different instruction LD that thrown an exeception during the MEM phase and a DADD that can generated an arthmetic exception during the EX phase. The handling could manage the first problem, and if its cause is removed, the second occurs. The situation can be different supposing that the DADD generate an error in the IF stage and the LD still in the MEM phase, this case generate before the DADD exception and then the other.
The are multiple solution, a good one could be:

- A status flag is associated to each instruction in the pipeline.
- If an instruction causes an exception, the status flag is set.
- If the status flag is set, the instruction can not perform any write operation.
- When an instruction reaches the last stage, and its status flag is set, an exception is triggered.

In some cases machines have instructions that change the state before they are committed (those using autoincrement addressing modes). If one of these instructions is aborted because of an exception, it leaves the machine state altered. Implementig precise exceptions could be really tough.
Instructions implicitly updating condition codes create complications:

- Data Hazards
- Save and restored in case of exception.
- Hardening the compiler work to filling delay slots between writing conditions and branch.

3.5 Multicycle Operations

Floating points units perform more complex operations than the integer ones. Forcing them to be executed in a single CLK means using a very slow clock or using really complex unit. The main solution is repeating the EX stage as many times as the instruction requires. An example for undestan the number can be viewed in this table below:

Functional Unit	Latency	Init Interval
Integer ALU	0	1
Data Memory	1	1
FP Add	3	1
FP/Int Multiply	6	1
FP/Int Divide	24	25

Latency: It is the number of cycles that should last between an instruction that produces a result and an instruction that uses the same result.
Initiation Interval: It is the number of cycles that must elapse between issuing two operations of the same type to the same unit.

Hazards can be more frequent due to the different structure of the EX stage. This because:

- Unpipelined divide unit, where several instructions could need it at the same time.
- The instructions have varying running times, the number of the register writes required in a cycle can be larger that 1.

solving this problem with more writes ports is normally too expensive, the solution is forcing a structural hazard stalling the instructions in the ID stage, or stalling it before the MEM or WB stage. This kind of operations can introduce long period of stall. This introduce new type of hazards, like **RAW** (Read After Write), **WAW** (Write After Write) and structural hazards involving the divide unit and the write port.
A solution could be that, before issuing an instruction to the EX phase, check whether it is going to write on the same register of an instruction still in the EX stage. In this case, stall the new instruction. Of course guarantee the precise exception become really hard, the possible solutions are:

- Accepting imprecise execeptions.
- Fast, but imprecise operating mode, and slow, but precise one.
- Buffering the results of each instruction until all the previously issued instructions have been completed.
- Forcing the FP units to early determine whether an instruction can cause an exception, and issuing further instructions only when the previous ones are guaranteed not to cause an exception.

An alternative solution is like the MIPS R4000 that implements a 8 cycle pipelined processors. This means more forwarding, increased load delay slot and increased branch delay slot.

3.6 Instruction Level Parallelism

The pipelines exploit the parallelism existing among instructions, which allows their execution in parallel. The highest the amount of ILP that can be found, the better the performance of the pipeline. There are two kind of approach:

- Static: Depending on the software (i.e compiler). Used in embedded system.
- Dynamic: depending on the hardware to locate parallelism. Used in desktop and server.

The basic block is the group of instructions belonging to the same block, it means no branches in (except entry) and no branches out (excepts exit). The basic block is opmizable by the compiler by a little rescheduling. For example if we need to compute $a = b + c$
$d + e - f$ on solution could be the one in figure 6: that requires 14 clocks cycles.

```
LD   Rb, b      IF ID EX MEM WB                                  5
LD   Rc, c         IF ID EX  MEM WB                              1
ADD  Ra, Rb, Rc       IF ID  st  EX  MEM WB                      2
SD   Ra, Va              IF  st  ID  EX  MEM WB                  1
LD   Re, e                   IF  ID  EX  MEM WB                  1
LD   Rf, f                       IF  ID  EX  MEM WB              1
SUB  Rd, Re, Rf                      IF  ID  st  EX MEM  WB      2
SD   Rd, Vd                              IF  st  ID EX   MEM WB  1
```

Figura 6: First execution order

In figure 7 a better solution is implemented, this rescheduling requires only 12 clock cycles. For typical MIPS program the size of a basic block is between 4

```
LD   Rb, b      IF ID EX MEM WB
LD   Rc, c         IF ID EX  MEM WB
LD   Re, e            IF ID  EX  MEM WB
ADD  Ra, Rb, Rc          IF  ID  EX  MEM WB
LD   Rf, f                   IF  ID  EX  MEM WB
SD   Ra, Va                      IF  ID  EX  MEM WB
SUB  Rd, Re, Rf                      IF  ID  EX  MEM WB
SD   Rd, Vd                              IF  ID  EX  MEM  WB
```

Figura 7: Second execution order

and 7 instructions. Since these instructions are likely to be dependent one from the other, the amount of parallelism existing within a basic block is normally rather small.
COnsidering a loop any iteration of a loop could be independent on the others, so they can be overlapped. There are two ways for exploting the loop-level parallelism:

- Loop unrolling (static or dinamyc)

- SIMD

The first solution explicitly replicate the loop body in multiple times like in figure 8. The considerations are:

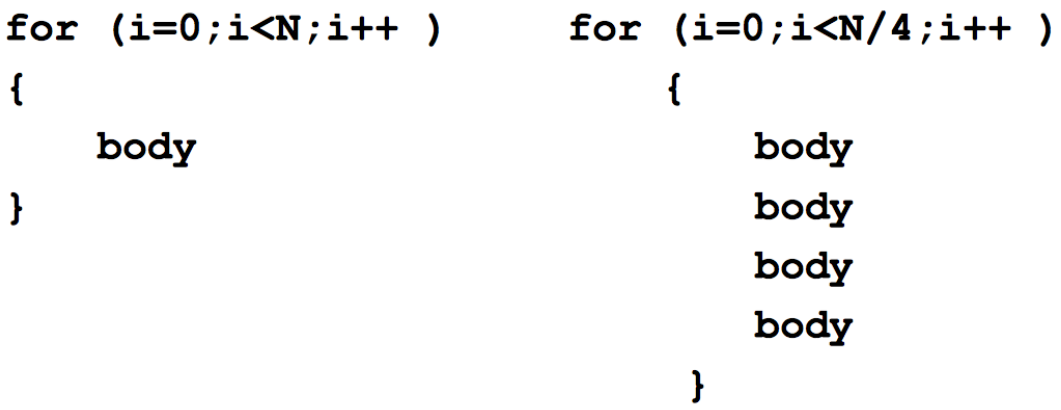


Figura 8: Loop unrolling

+ Reduced overhead due to the iteration control.

+ Wider body increase the chance for the compiler to exploit rescheduling to eliminate stalls.

- Size code increased.

The SIMD techniques it can be exploited in:

- Vector processors: A vector instruction operates on a set of data, instead of on a scalar data (as normal instructions).

- GPUs: Parallel acting over multiple data.

In both cases is necessary to evaluate the dependencies, if two instructions are not dependent, they can be executes in parallel without any stall. If they are not, they have yo be executed in order (or partly overlapped). There are 3 kind of dependencies: data, name and control.
An instruction i is **DATA** dependent on instruction j, if i produces a result that is used by instruction j, or if j is dependent on instruction k, and k is dependent on instruction i.
Detecting dependencies involving registers is easy. Detecting dependencies involving memory cells is much more difficult, because accesses to the same cell can look very different. Using a static techniques force the compiler to adopt a conservative approch, assuming that anu load instruction refers to the same cell of the previous store. This kind of dependencies can be only detected at the tun time, when the addresses are known.

A **NAME** dependencies occurs when two instructions refer to the same register or memory location (name) but there is no flow of data associated to the name. They can be of 2 types:

- *Antidependence*: Instruction j writes a register or memory location that instruction i reads, and instruction i is executed first.
- *Output dependence*: Both instruction i and j write the same register or memory location.

The name dependencies do not prevent from reordering involved instructions, provided that we change the register used by one of the two instructions, it can be exploited by compiler (statically) or by the processor (dynamically). A similar method can be implemented to solve name dependencies involving memory locations. Finding possibile hazards require carefully analyzing the code, taking also into account the effects of branches.
Another kind of dependecies is the control one, it occurs when an intruction depends on a branch. An instruction that is control dependent on a branch cannot be moved before the branch (so that its execution is no more controlled by the branch). Altought, an instruction that is not control dependent on a branch cannot be moved after the branch (so that its execution become dependent on the branch). Preserving the control dependencies is a sufficient condition for preserving the program correctness. The reverse is not always true! The critical properties for program correctness are:

- Exception behavior
- Data flow

Changing in the instruction order must not change how exception are raised. Anticipating the LD before the BEQZ in this piece of code:

```
DADDU R2, R3, R4
BEQZ R2, L1
LD R1, 0(R2)

L1: ...
```

is not possible because LD can cause an exception, no matter whether the branch is taken or not.
The data flow is easy to be understand, changing the order of some operations can change the final result of the software running therefor is important to pay attention on that problem.

3.7 Branch Prediction Techniques

Branches can potentially impact seriously on the pipeline, a solution can be to predict how branches will behave. Also in this case the are two main solution:

- Static Techniques: Handled by the compiler resorting to a preliminary analysis of the code.
- Dynamic Techniques: Implemented by the hardware based on the behavior of the code.

The **static** solution become useful when cobined with other static techniques, delayed branches and rescheduling to avoid data hazards. Th compiler may predict branch behavior in different alternative ways:

- Predict branches as taken.
- Predicting behavior depending on branch direction: Forward branch often untaken, Backward branches more often taken.
- Prediction based on earlier runs: Try to run a limited number of times, gather statistics and use it for prediction.

Using the dynamic technique, the hardware is in charge of prediction. Starting from the branch instruction address to active different prediction mechanisms. There are some difference between possibile prediction:

- Branch History Table (1 and 2 bit schemes).
- 2-level prediction schemes (correlating predictors).
- Branch-target buffer.

The **BHT** is the simplest method. Is a small memory indexed by the lower portion of the address of the branch instruction, containing for each entry one or more bits recording whether the branch has been taken or not in the last execution.
Two-bit prediction schemes provide higher prediction capabilities. For every branch, two bits are mantained, and the prediction is changed only after missing twice. There is also the **n-bit** solution that us n bit for performing a similar behavior of the previous method. When the counter value is greater than one half of tis maximum value, the branch is predicted as taken, when it is lower it is predicted as not taken.
The correlating predictors is an approach (also called two-level predictors) base on exploting dependencies between the result of branches, where the the behavior of a branch is strongly dependent on the result of the previous ones. They use the behavior of the last m branches to choose from 2^m branch predictors, each of which is a n-bit predictor. The hardware implementation is very simple:

- The history of the most recent m branches is recorded in an m-bit shift register, where each bit records whether the branch was taken or not.

- The branch-prediction buffer is indexed using a concatenation of the low-order bits from the branch address with the m low-order history bits.

Reducing the negative effects of control dependencies requires knowing as soon as possible:

- Whether the branch has to be taken or not.
- The new value of the PC (if the branch is assumed to be taken).

The later issue is faced by introducing a **Branch-Target buffer** (or cache). Each entry of the BTB contains:

- **Address** of the considered branch.
- **Target** value to be loaded in the PC.

Using the BTB, the PC is loaded with the new value at the end of the IF stage, i.e., even before the branch instruction is decoded. The flow of this solution in figure 9.
The BTB could be used cobined with other strategy, like branch history table, if a two-bit prediction is adopted.

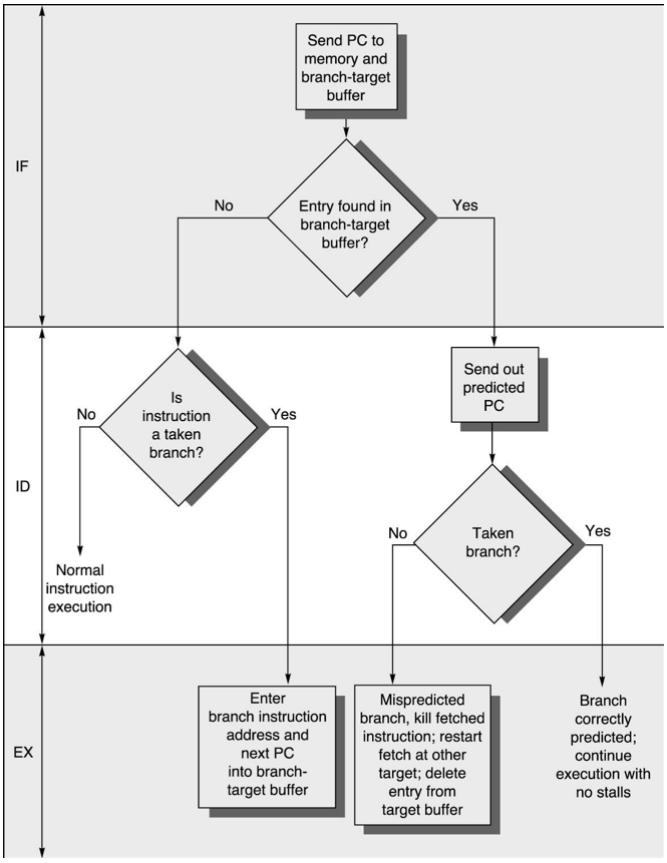


Figura 9: Branch-Target Buffer behavior

3.8 Dynamic Scheduling Techniques

The *dynamic scheduling* is a technique aimed at rearranging in hardware the instruction execution order to reduce the pipeline stalls while maintaining data flow and exception behavior. The advantages of DS are:

- Identifies some dependencies that are unknow at compile time.
- Simplifies the compiler job.
- Allows the processor to tolerate unpredictable delays.
- Allows to run the same code on different pipelined processors.

Starting from this example:

```
DIV.D F0, F2, F4
ADD.D F10, F0, F8
SUB.D F12, F8, F14
```

where the ADD instruction need to wait the end of the DIV for a data dependency, we can think about some DS solutions.
One possible solution for the proposed example is to remove the constraints of *inorder instruction execution*. This implies some considerations, it may introduce:

- WAR hazards
- WAW hazards
- Imprecise execeptions

If the out-of-order execution is allowed, precise exception handling is impossible. This may mean that at the time when the exception is raised:

- An instruction before that raising the exception still has to be completed, or
- An instruction after that raising the exception has been already completed.

In both cases resuming program execution after exception handling becomes really difficult.
Splitting ID stage it useful to support out-of-order execution, it is splitted in two parts:

- Issue: decode instruction, check for structural hazards.
- Read Operands: Wait until no data hazards, then read operands.

The issue stage reads instructions from a register or a queue (written by the IF stage). Instruction issue is performed in-order. After the instruction are issued, they may wait for operands, and then (when operands are available) sent to the EX stage. They can be stalled or bypass each other while in the read operand stage. Therefor, they can enter execution out-of-order.
If the processor includes multiple functional units, multiple isntructions may be in execution at the same time. Instructions can also bypass each other while in the execution stage. Therefor, they can leave execution out-of-order.

Hardware schemes ont of the first solutions to achieve the dynamic scheduling was proposed in 1967 from an IBM architect, Robert Tomasulo. The key ideas in the Tomasulo's algorithm are:

- Track the operands availability.
- Introduce register renaming.

The **reservation stations** are the key novelty in the Tomasulo's approch. They have several functions:

- They buffer the operands of instructions waiting to issue; operands are stored in the station as soon as they are available.
- They implement the issue logic.
- The univocally identify and instruction in the pipeline: pending instruction designate the reservation station that will provide them with an input operand.

Using the register renaming allow to eliminate WAW and WAR hazards. Each time an instruction is issued, the register specifiers for the pending operands are renamed to the names of the reservation stations in charge of computing them. An example in figure: The reservation stations related to each functional

Instruction	$f0	$f2	$f4	$f6	Renamed Instruction
initial values	V0	V1	V2	V3	————————
mul.d $f6, $f0, $f2				V4	mul.d V4, V0, V1
div.d $f4, $f2, $f0			V5		div.d V5, V1, V0
add.d $f0, $f6, $f2	V6				add.d V6, V4, V1

Figura 10: Register Renaming

unit control when an instruction can begin execution at that unit. Results are passed directly to other functional units, rather than going through the registers. All results from the functional units and from memory are sent to the *Common Data Bus*, which:

- Goes everywhere, execept to the load buffer.
- Allows all units waiting for an operand to load it simultaneously when it is available.

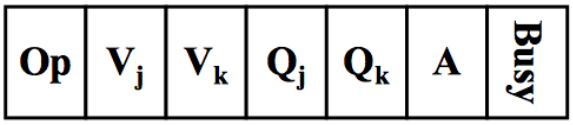
Issue get an instruction from the instruction queue (implementing FIFO strategy):

- If no reservation station is available, a structural hazard occurs, and the instruction stalls until a reservation station becomes available.
- If there is an empty reservation station, send the instruction to it:
  - If the operands are available, they are sent to the reservation station.
  - If not, the functional units responsible fo their generation are recorded.

Flow In case of normal instructions, when an operand appears on the CDB, it is read by the reservation unit. As soon as all the operands for an instruction are available in the RU, the instruction is executed. For the load and store operations, the first step is, as soon as the base register is available, the effective address is computed and written in the load/store buffer. The second step will be, for the load instruction: execute as soon as memory is available, for the store instructions, wait the operand to be written and then executed as soon as memory is available. In case of branches, to preserve exception behaviour, no instruction is allowed to initiate execution until all branches that precede the instruction in program order have completed. When the result of an instruction is available, it is immediately written on the CDB, and from there is the register and functional units waiting for it.

Each reservation station is associated with an **identifier**. It identify also the operand the instruction will produce. Instructions requiring that operand identify it using the identifier. The identifiers play the role of names of virtual registers that ca be used to implement register renaming.

RS Fields the reservation station is divided in more field like in figure below:



where:

- Op: Operation to be performed.

- V_j, V_k: Values of the source operands.
- Q_j, Q_k: Reservation stations that will produce a source operand.
- A: Used in l/s buffer only. Immediate field and then effective address.
- Busy: Status of the reservation station.

The **register file** contains, for each element, one field Q_i; that is the number of the reservation station that contains the instruction whose result should be stored in the register. If is *null*, no currently active instructions computing a result destinated to this register.

In figure 11 an example of the reservation station for this code:

```
L.D  F6, 34(R2)
L.D  F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F8, F2, F6
DIV.D F10, F0, F6
ADD.D F6, F8, F2
```

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					45 + Regs[R3]
Add1	yes	SUB		Mem[34 + Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

Figura 11: Reservation station of the example

The register file is represented in figure

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Figura 12: Reservation station of the example

There are several advantages using this solution:

- The hazard detection logic is distruited.
- Stalls for WA and WAR hazards are elimnated.

- Loop unrolling not required, naturally performed in parallel.

there are also some disadvantage:

- High complxity hardware (including an associative buffer for each RS).
- The CDB can be a bottleneck.

3.9 Hardware-based Speculation

The HBS is a technique for reducing the effects of control dependences in a processor implementing dynamic scheduling. If a processor supports branch prediction with DS, it *fetches* and *issues* instructions, as if the branch prediction was always correct. If a processor support BHS, it also *executes* them.
The HBS combines three ideas:

- Dynamic Branch Prediction
- Dynamic Scheduling
- Speculation

Is such way, the processor implements **data flow execution**: *Operations execute as soon as their operands are available.*

Architecture it implements the basic Tomasulo's architecture, extending it to support speculation.
There are two different steps in instruction execution:

- The computation of results and their bypassing to other instructions.
- The update od register file and memory, which is only performed when the instruction is no longer speculative (instruction commit); in this way, in-order commitment is implemented.

Is introducce a new datastructure, the **ReOrder Buffer** (ROB), it contains the instruction results while the instruction didn't commit yet. It provides additional virtual registers and integrates the store buffer existing in the origianl Tomasulo's architecture.
In the TM's architecture already computed results are read from the register file. With speculation, data may be read:

- from the ROB, if the producing instruction didn't commit yet.
- from the register file, otherwise.

Each entry of the ROB has four fields:

- Instruction Type: Branch, store, or register.
- Destination: Register number, or memory address.
- Value: Contains the value when the instruction has completed but still didn't commit.
- Ready: Indicates whether the instruction completed its execution.

Steps this solution provides four execution steps. The first is the **Issue** (or Dispatch) step:

1. An instruction is extracted from the instruction queue id there is:
   - an empty reservation station and
   - an empty slot in the reorder buffer.

   If this not the case, the instruction issue is stalled.
2. The operands for the instruction are sent to the reservation station, if they are in the register file or in the reorder buffer.
3. The number of the reorder buffer entry for the instruction is sent to the reservation station to tag the instruction (and its results, when they will be written on the CDB).

The **execute** step flow is:

- The instruction is executed as soon as all the required operands are available (Avoid RAW hazards).
- Operands are possibly taken from the CDB as soon as another instruction produces them.

The lenght of this step varies depending on the instruction type (e.g. 2 for load instructions, 1 for integer instruction and different values for FP instructions).

Another important step is the **Write Result**:

1. A soon as it is available, the result is put on the CDB (together with the tag identifying the instr.) and sent to the ReOrder Buffer.
2. Any RS waiting for the result reads it.
3. The RS is marked as available.

The last step is the **Commit** (or Completion) step:
The ReOrder Buffer is ordered according to instructions original order.
As soon as one instruction reached the head od the buffer:

- If it is a mispredicted branch, the buffer is flished, and the execution is restarted with the correct successor of the instruction.
- Otherwise, the result is written in the register or in memory (in case of a store).

In both case the ROB, implemented as a *circular buffer*, entry is marked as free.

Since a dynamic renaming is implemented and memory updating occurs inoder WAW or WAR hazards can't occurs. RAW hazards instead are prevented by:

- Enforcing the program order while computing the effective address of a load wrt all earlier store instruction, and
- not allowing a load to initiate its second step if any active ROB entry occupied by a store has a destination field matching the A field of the load.

Th **store** instructions write to memory when they commit, only. Therefor, their input operand is required when they commit, rather then in the WR stage. This means that the ROB should have a further field, specifying where the input operand for each store instruction should come from.
Exceptions are not executed as soon as they are raised, but they are stored in ROB. When the instruction is committed, the possible exception is executed, and the following instructions flushed from the buffer.If the instruction is flushed from the buffer, the exception is ignored. Full precise exception handling is thus supported.

When a time-expensive event (e.g. second-level cache miss, TLB miss) occurs speculatively, some processors wait for its execution until the event is no more speculative. On the other side, low-cost events (e.g. first-level cache miss) are nomally executed speculatively.