AIC NOTES

UNIT 1: INTRODUCTION TO COMPUTER ARCHITECTURE

THEME 1.- CONCEPT OF COMPUTER ARCHITECTURE

1.-CONCEPT OF COMPUTER ARCHITECTURE

The **computer architecture** defines the hardware of a computer attending to its requirements, as a design trade-off between performance, available technology and cost. It encompasses with the following levels:

- **Instruction Set Architecture (ISA)**: It includes everything that must be known by assembler programmers.
- **Processor organization**: It describes the logical elements enabling the execution of instructions.
- **Implementation**: It defines the computer implementation.

The **computer enginee**r task is:

- Consider the expected requirements.
- Identify existing technology, energy and cost limitations.
- Provide the best possible design.

So, to carry on these tasks, the computer engineer must quantify performance, cost and other features, compare and select.

2.-COMPUTER REQUIREMENTS

In order to design a computer, an architect **must** consider:

- The type of **required computer**.
- The degree of **compatibility** with other existing computers.
- **Operating system requirements**.
- **Market standards**.

3.-TECHNOLOGY, POWER CONSUMPTION AND COST

The **number of transistors** per chip **increases** per year, but we are achieving a limit. The number of transistors **grows quadratically** with the reduction of their feature size. Transistor speed **increases linearly** with the reduction of their feature size.

The **size of a transistor** is **decreasing** within time. But with this, it appears a **rise of propagation delay in interconnections**, that is, we need more clocks to reach all the chip. Also, the **leakage current increments** with the decrease of feature size, and so, the heat. Thus, we have a **heat dissipation problem** too.

In order to **reduce** the **power consumption**, we **decrease** the **supply voltage** down to a minimum needed.

Some implications related to power consumption and heat dissipation are:

- Microprocessor power (current) **distribution**
- **Heat dissipation** and **cooling**
- Development of **new materials** to reduce the leakage current

The factors that decrease the cost of components are:

- **Learning curve**: The cost of a component decreases over time, since throughput increases.
- **Sales volume**: Doubling the sales volume decreases costs by 10%.

The design costs are:

- **Factory costs** are inversely proportional to **feature sizes**.
- **Size of a design team** is inversely proportional to **feature size**.

4.-EVOLUTION OF PROCESSOR PERFORMANCE

Period I (1978–1986). Performance grows at a rate of 25% per year, mainly due to technology enhancements.

Period II (1986–2003). Performance grows at a rate of 52% per year, due to technology enhancements and **architectural improvements**.

Period III (2003–2011). Performance grows at a rate of 23% per year. Limits in ILP, Memory latency, power consumption, and heat dissipation -> performance improvement can only be achieved through parallelism.

Period IV (2011–2015). Performance grows at a rate of 12% per year due to technological and parallelism limitations (Amdahl's law).

Period V (2015–2018). Negligible performance growth (6.5% per year):

- End of Moore's law.
- Technology does not improve energy efficiency.
- Reaching the limits of parallelism.

Current situation:

- Performance and energy efficiency through specialization -> use of domain-specific accelerators
- Research and Development of new technologies.

Architectural improvements:

- **RISC architecture**: Simple instructions that execute very fast. Simple hardware.
- **Pipelining**: Splitting the instruction cycle into phases that are executed concurrently.
- **ILP exploitation**: Execution of instructions out of program order.

- **Parallelism**:
    - **DLP Data-Level Parallelism**: Performing an operation on multiple data.
    - **TLP Thread-Level Parallelism**: Performing several tasks in parallel.

Some architectural ideas are motivated by the technology available:

- **Caches**: the speed difference between the processor and the memory motivated the introduction of the cache memory between the processor and memory. The continuous growth of this difference has motivated the use of various levels of cache in the designs.
- **Stages to propagate signals**.
- **Chip multiprocessors**: we can put several simpler processors that operate at lower frequency and lower voltage.

5.-TYPES OF COMPUTERS

During the 70's, as first microprocessors were under construction, two types of computers were commercially available:

- **Mainframes**: very big and expensive computers only affordable to big corporations.
- **Minicomputers**: medium-size computers, largely used at universities.

Currently, most relevant computers are: Mobile devices, Embedded Systems, Personal computers, Servers, Clusters, Supercomputers.

5.1.-MOBILES DEVICES

The main features of **mobile devices** are:

- Very **limited energy consumption**.
    - Battery powered.
    - No forced cooling.
- **Responsive** and **predictable design**.
    - Need of multimedia GUIs.
    - Video frames and audio block must be processed on time.
- **Reduced main memory capacity**.
    - Optimized code (small footprint).
- **Secondary memory** of **flash** type.

5.2.-EMBEDDED SYSTEMS

Mobile devices are a concrete case of **embedded system**. The differences with mobile devices are:

- Wide spectrum of design and performance requirements.
- Specific software developed by manufacturers.

5.3.-PERSONAL COMPUTER

The main features of **personal computers** are:

- **Well-balanced numerical** and **graphical computing power**.
- **Optimal relation** between **performance** and **cost**.
- **Wide spectrum of configurations**.

5.4.-SERVERS

A **server** is a computer providing services in a network. Use to be part of the computer infrastructure of a corporation. Its main features are:

- **Availability** is critical.
- **Scalable** design.
- Design for **responsiveness**.
- **Throughput** is a key attribute.

5.5.-CLUSTER

A **cluster** is a collection of computers, each one with its own operating system, interconnected through a network. It is externally seen as a single computer. Its features are:

- Significative investment in **power supply** and **cooling**.
- **Scalability**: the system can be easily adapted to changes in workloads.

A **large scale cluster** provides support to the big internet services. Its main features are:

- Need of a **huge Internet bandwidth** and **secondary storage**.
- **Good balance** between **performance** and **cost**.
- **Dependability-oriented design**: availability is critical, but low-cost components are used, thus, there is a need of redundancy.

5.6.-SUPERCOMPUTER

Supercomputers are machines designed for high performance, despite their cost. Their main features are:

- Execution of **large scientific distributed applications** with limited user interaction.
- Very **high floating point arithmetic throughput**.
- **Cluster-based supercomputers** are becoming more and more common.

THEME 2.- PERFORMANCE EVALUATION

1.-PERFORMANCE DEFINITION

The response time or **execution time** is the time to complete a task.

The **throughput** is the number of operation/executions completed per time unit.

The **relation** between both concepts is:

$$Throughput = \frac{1}{Execution\ time}$$

When two computers (X and Y) are compared, the slowest one Y is taken as reference.

In order to compare computers X and Y, a workload must be selected in order to measure their respective performances under similar execution conditions.

The relation **S** is computed as:

$$S = \frac{T_y}{T_x} = \frac{P_x}{P_y} = 1 + \frac{n}{100}$$

S represents the **speedup** and it must be interpreted as follows:

- "X is S times faster than Y".
- "X is n % faster than Y".

** S < 1 is WRONG *

2.-QUANTITATIVE PRINCIPLE OF COMPUTER DESIGN

Execution time equation:

$$T_{exec} = \frac{sec}{program} = \frac{nº\ instructions}{program} * \frac{cycles}{nº\ instructions} * \frac{sec}{cycle} = I * CPI * T$$

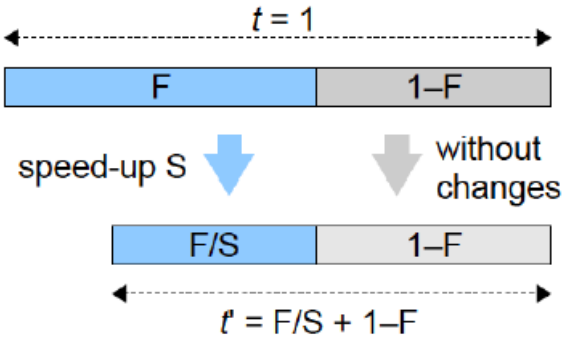
All three parameters are related:

- I = f(instruction set architecture, compiler).
- CPI = f(instruction set architecture, organization).
- T = f(technology, organization).

It is not possible to reduce one of these parameters without affecting the others.

** 100% question in first partial *

Amdahl's law is based on the question: how does a change in a part of a system/process affect the whole?



Where **F** is the fraction of time affected by the improvement and **S** is the applied speed-up or the factor improving F.

The **execution time** is:

$$t' = t * (1 - F) + \frac{t}{S} * F$$

The resulting **global speed-up** S' will be:

$$S' = \frac{t}{t'} = \frac{1}{(1 - F) + \frac{F}{S}}$$

The fraction of time that is not improved (**1 - F**) defines an upper **bound** to the **maximum reachable** speed-up.

Amdahl's law can be generalized for multiple fractions (n), each of them being accelerated with a different speedup. Additionally, a given speedup Si can be the result of combining several independent speedups.

The relation between cost and performance measures if that an improvement on performance is worth it in relation with the price.

3.-MEASURING PERFORMANCE

Performance is measured with a program or a collection of programs that are likely relevant to the user. The source code of such program(s) must be available. It must be compiled for each computer under analysis. The best option is real programs.

The alternative options are:

- **Kernels**: fragments of code extracted from real programs.
- **Toy benchmarks**: simple programs with well-known execution results.
- **Synthetic benchmarks**: programs written in order to represent the average program typically run in a system.

Benchmark suites:

- **Basic structure**: typically kernels and real non-interactive programs defined to measure performance attending to a given user profile.
- **Must be easy to update**: programs in the suite must represent at any moment the type of tasks typically run in the system by a regular users. Benchmarks are periodically updated.
- **Reproducibility**: measures must be reproducible and all details must be clearly defined, that is, hardware and software.

A **feature of a good time metric** is that the average value must be directly proportional to the execution time. Some examples:

- **Total execution time** (sum of execution times):

$$T_T = \sum_{i=1}^{n} Time_i$$

- **Arithmetic mean**:

$$T_A = \frac{1}{n} \sum_{i=1}^{n} Time_i$$

- **Sum of weighted execution times**:

$$T_W = \sum_{i=1}^{n} W_i * Time_i$$

  Where w_i represents the frequency of program I in the considered workload.

- (**SPEC**) the geometric mean of execution times normalized with a reference machine, that is, R times faster than the reference:

$$R = \sqrt[n]{\prod_{i=1}^{n} \frac{Time_{ref}}{Time_i}}$$

4.-OTHER PERFORMANCE METRICS

MIPS (Millions of Instructions Per Second):

$$MIPS = \frac{number\ of\ instructions\ executed}{T_{execution} * 10^6} = \frac{I}{I * CPI * T * 10^6} = \frac{1}{CPI * T * 10^6} = \frac{f}{CPI * 10^6}$$

It is an intuitive measure **proportional** to **performance**. It **does not account** for the **number of executed instructions**. It depends on the considered programs. Different programs execute different instructions, of different complexity and execution time.

It **depends** on the **instruction set**. The same program executes different number of instructions in each machine, according to the complexity of its instruction set, so it is not suitable for comparing machines with different instruction set. It may be inversely proportional to performance.

MFLOPS (Millions of Floating point Operations Per Second):

$$MFLOPS = \frac{num.\ of\ floating\ point\ operatins\ in\ the\ program}{T_{execution} * 10^6}$$

It **accounts** for **operations** instead of instructions: the execution of the same program on different architectures will require a different number of instructions but the same number of FP operations. It cannot be applied to programs that are not carrying out any floating point operations. This is the case of text processors and compilers.

It depends on the FP instruction set of each computer, which is not always the same, so the number of FP operations is not kept constant. The solution is to rely on source code FP operations. Different programs execute different FP operations and they usually have different costs.

THEME 3.- INSTRUCTION SET ARCHITECTURE DESIGN

1.-GENERAL ASPECTS RELATED TO INSTRUCTION SETS

Instruction sets act as interfaces between programs and datapaths of processors. Instructions results from program compilation, so those instructions rarely used by compilers are useless. The experience in program compilation shows that, although programs can be very complex, most of them are very simple. A fast execution of simple instructions is possible using simple datapaths, so the CPI and the period decreases.

The main features of **instructions sets** are:

- **Basic principle**: the **common** case must be **efficient**, but the **uncommon** only has to be **correct**.
- **Orthogonality**: whenever it makes sense, operations, addressing modes and data types must be independent. This simplifies the generation of code.
- **Provision of primitives instead of solutions**: avoid the inclusion of solutions directly supporting high-level instructions. Attending to their specificity, such solutions only work for particular programming languages. They provide more or less functionality than necessary. Instead, instruction sets must provide to compilers primitives rather to solutions for the generation of optimized code.
- **Principle ¨one for all¨**: either there is only one way to make something, or all ways are possible. This reduces the cost of computing each alternative.
- **Integrate instructions operating on constants**: some values are already known at compile-time.

2.-TYPES OF INSTRUCTIONS SETS

The **classic criteria** is that the storage of operands is in the CPU. Instructions compute on input data and produce a result. Input data and results can be stored in memory and CPU.

There exist three **paradigms**:

- **Stack**: input data and results are stored in the stack. (Implicit) operands are stored in the stack head and replaced by the result.
- **Accumulator**: there is an (implicit) accumulator register storing one of the operands and the final result.
- **General purpose registers**: Operands and results are in a register file. All operands must be explicitly referenced.

Current instruction sets use a **general purpose** register file. The paradigm of processor with general purpose register file and addressable memory allows an efficient compilation of programs.

2.1.-GENERAL PURPOSE REGISTERS

Two relevant parameters:

1. n **operands** in ALU instructions (2 or 3).
    a. If n = 2, then one operand acts as source and destination.
    b. If n = 3, then there are two sources and one destination.
2. **Number** m <= n of **memory addresses** in ALU instructions

Cases:

- If m = 0, load/store reg–reg computers (typically m = 0; n = 3).
    - All instructions work with register data and store results in registers.
    - Only load and store instructions exchange information with memory.
- If m < n, reg–mem computers (typically m = 1; n = 2).
- If m = n, mem–mem computers (typically m = 3; n = 3).

The choice impacts the execution time Tex = I * CPI * T. There are two main types of instructions sets:

- IA (Intel Architecture).
- RISC.

Their main features are:

	Intel Architecture	**RISC**
Prog. model	*R-M* (Register-memory)	*L/S* (Load/Store)
Registers	Few, variable length	A lot, fixed length
Instr. format	Variable	Fixed (32 bits)

2.2.-L/S AND R-M MODELS

RISC L/S Model:

1. ALU instructions compute only on registers.
2. ALU instructions include three operands.
3. Load and Store instructions exchange data between registers and memory.
4. The amount of work performed by all instructions is similar (a computation or a memory access).

IA R-M Model:

1. Instructions work on registers, or with an operand in memory.
2. Instructions have two operands.
3. The MOV instruction exchange data R ⇔ M and R ⇔ R
4. The amount of work performed by instructions is quite diverse.

These models affect the execution time by:

RISC L/S Model:

1. A program has more instructions to carry out the same work, so the number of instructions raises.
2. Simpler format, easier decoding, so the period (T) decreases.
3. The amount of work per instruction remains similar, so the CPI decreases.

IA R-M Model:

1. A program integrates less instructions to perform the same work, so the number of instructions decreases.
2. Variable format, more complex decoding, so the period (T) raises.
3. Heterogeneous amount of work per instructions, so the CPI increases.

3.-REGISTERS AND OPERAND TYPES

RISC has a big number of registers, all of them with the same length. Regular instructions use the whole content of registers. L/S instructions transform integer types whenever it is necessary.

IA has a few registers that are adapted to the available data types. Each ALU instruction has a different operation code for each data type.

Changing the word length is simpler in RISC computers.

4.-INSTRUCTIONS ENCODING

Instructions are stored in memory according to a format indicating the operation to carry out (operation code) and the operands. There are two types of format:

- **Fixed**: all instructions are encoded using the same number of bits.
    - It **easies fetching** and **decoding** of instructions
    - Sometimes, it **wastes bits** in the format, since all instructions do not require the same amount of bits to be encoded.
- **Variable**: the required number of bits for encoding varies attending to the instruction type.
    - **Space** taken by instructions (and programs) is **optimized**.
    - It makes more **complex fetching** and **decoding** of instructions.

The number of bits of the format limits the space devoted to each instruction field, which limits the number of available variants. There are two number of instructions formats:

- **One format**: the correspondence between format bits and fields is always the same.
    - **Easies** instruction **decoding**.
    - Sometimes, **format bits are wasted**, since not all instructions require all the available fields.

- **Multiple formats**: each format can have different fields and establishes a correspondence between such fields and the format bits.
    - It allows a **better adjustment** of those bits taken by each instruction and the required fields.

5.-MEMORY ADDRESSING

Physical reading/writing units (words) contain $W = 2^w$ addressable units (bytes). Words are referred through the address of their least significant byte, so the word with address A contains the bytes with addresses A, A + 1, … ,A +W – 1. Two alternatives, with no important consequence are:

- **IA Little endian**: byte with address A is located at the least significant word position.
- **RISC Big endian**: byte with address A is located at the most significant word position.

Instruction sets provide access to units of 1, 2, … , 2^w bytes. There are two alternatives:

- **RISC Aligned access**: the address of the object of 2^i bytes is multiple of 2^i.
- **IA Non-aligned access**: there is no restriction. An instruction can access contiguous byte in two consecutive words, and its execution will demand two physical accesses to memory. The impact on the HW complexity makes the CPI and the period (T) increase.

Addressing modes specify instruction operands. The support of sophisticated addressing modes reduces the number of program instructions, but have a more complex hardware, so the CPI and/or the period (T) increase.

RISC Simplifying:

- Two versions of ALU instructions available:
    - R-format: Rd <= Rs op Rt.
    - I-format: Rd <= Rs op X.
- **Range of available values**: a solution that must balance the number of bits taken and the magnitude of the constants required. 16 bits in the MIPS. Solutions are also provided to easy the work with bigger immediate values.
- Main memory accesses must be performed using the displacement addressing mode: X(Rn):
    - Using X=0 results in the register indirect addressing mode.
    - Using Rn = $zero results in the absolute mode.
- **Displacement range**: solution that must balance the size of the necessary displacements and the number of required bits. 16 bits in the MIPS.

6.-OPERAND AND OPERATIONS

Normally the type of an operand is included in the operation code, although some old computers provided operands and labels representing their types.

Most commonly supported types are: char (8 bits – ASCII), integer (8-byte, 16-half-word, 32-word y 64 bits-double word, all of them in 2’s complement), floating point (single-precision- 32 bits and double precision -64 bits, in the IEEE 754 standard).

There exist a great variety of operations:

- **Integer arithmetic**, **binary logic**.
- **Transfer**: load/store or move.
- **Control**: branches and jumps, call/return.
- **System**: OS calls, virtual memory management.
- **Floating point**: arithmetic and real-integer conversions.
- **Decimal**: arithmetic and decimal-char conversions
- **Strings**: transference, comparison, search.
- **Multimedia**: vertex y pixels operations, compression/decompression, SIMD instructions.

7.-CONTROL FLOW

There are three types of control instructions:

- **Conditional branches** (branch).
- **Unconditional branches** (jump).
- **Call/return to/from a procedure**.

Statistics of use:

- Jump, call and return represent 1/3 , and are always taken.
- Conditional branches. Other 1/3 corresponds to loops, which are nearly 100% of the times taken. The remaining 1/3 of the branches are taken in 50 % of the cases.

The most likely alternative for a branch is to take it: 5/6 are taken while only 1/6 are not.

Addressing modes:

- **PC-relative**:
    - Destination use to be near the current instruction, so relative addresses take few bits.
    - Number of bits for encoding the (relative) displacement typical values are 16–20 bits in conditional branches and 26 bits in unconditional ones.
- **Indirect link**: useful if the branch destination is unknown at compilation time.
    - Statements selecting one over several alternatives.
    - Virtual methods in OO-languages.
    - Exchange of functions as parameters for other functions.
    - Dynamically linked libraries.
- **Jump and Link**:
    - Used to invoke subprograms (call).
    - It is a branch using a Relative-PC or indirect link addressing mode and storing the return address in a register.
    - The subprogram return is performed using an indirect link addressing mode. The address (link) stored by the jump instruction in the corresponding register is now used to return.

There are several alternatives to specify **branch conditions**:

- **Conditions codes**:
  - The instruction set defines an “state” that is modified according to the state of the last ALU operation.
  - Typically, there are some condition codes or flags.
  - Branch instructions simply check condition codes.

  The drawbacks of this alternative are:

  - Generation of condition codes is not trivial and requires space inside the chip.
  - The modification of condition codes by all instructions poses some problems for code reordering and for multiple issuing of ALU instructions.
- **Explicit check** (MIPS): result of operations is explicitly checked using specific instructions. Conditions codes do not exist.

Conditional instructions consist of adding an additional operand, the condition, to conventional instructions:

- If the condition is TRUE, the instruction executes normally.
- If the condition is FALSE, the instruction behaves as a NOP.

The usefulness is that it removes branch instructions, so it reduces instructions for simple flow control structures.

The **limitations of conditional instructions** are:

- They spend clock cycles, and therefore, they contribute to I, even if the condition is not met, then an excessive use of conditional instructions may end up increasing I.
- If the control flow leads to the execution of nested conditional sentences, the use of conditional instructions becomes less effective.
- CPI may be increased, or the clock frequency may be reduced.

8.-MIPS64 INSTRUCTION SET

Arithmetic/logical	*Operations on integer or logical data in GPRs; signed arithmetic trap on overflow*
DADD, DADDI, DADDU, DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB, DSUBU	Subtract, signed and unsigned
DMUL, DMULU, DDIV, DDIVU, MADD	Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends
DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV	Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic
SLT, SLTI, SLTU, SLTIU	Set less than, set less than immediate, signed and unsigned

Floating point	*FP operations on DP and SP formats*
ADD.D, ADD.S, ADD.PS	Add DP, SP numbers, and pairs of SP numbers
SUB.D, SUB.S, SUB.PS	Subtract DP, SP numbers, and pairs of SP numbers
MUL.D, MUL.S, MUL.PS	Multiply DP, SP floating point, and pairs of SP numbers
MADD.D, MADD.S, MADD.PS	Multiply-add DP, SP numbers, and pairs of SP numbers
DIV.D, DIV.S, DIV.PS	Divide DP, SP floating point, and pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C.__.D, C.__.S	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR*
LB, LBU, SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH, LHU, SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW, LWU, SW	Load word, load word unsigned, store word (to/from integer registers)
LD, SD	Load double word, store double word (to/from integer registers)
L.S, L.D, S.S, S.D	Load SP float, load DP float, store SP float, store DP float
MFC0, MTC0	Copy from/to GPR to/from a special register
MOV.S, MOV.D	Copy one SP or DP FP register to another FP register
MFC1, MTC1	Copy 32 bits to/from FP registers from/to integer registers

Control	*Conditional branches and jumps; PC-relative or through register*
BEQZ, BNEZ	Branch GPRs equal/not equal to zero; 16-bit offset from PC + 4
BEQ, BNE	Branch GPR equal/not equal; 16-bit offset from PC + 4
BC1T, BC1F	Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4
MOVN, MOVZ	Copy GPR to another GPR if third GPR is negative, zero
J, JR	Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
ERET	Return to user code from an exception; restore user mode

9.-SIMD INSTRUCTIONS

From 1975 to 2000 (approx.), several companies commercialized vector supercomputers, with instructions that operated on vectors.

When the integration scale was large enough, microprocessors extended their ISA with SIMD instructions that can operate on short vectors.

SIMD (Single Instruction - Multiple Data) instructions operate on registers that contain several packed data:

- They support different data types: 8, 16, 32, 64-bit integers, and single and double precision floating point.
- The number of data contained in a register is:

$$n = \frac{register\ size}{size\ of\ data\ type}$$

- The registers have a fixed size, but the instruction set allows packing data of different sizes.

A significant percentage of the additional transistors in each new generation of processors is devoted to increasingly powerful vector instructions and larger vector register files.

Compilation is based on data types and expressions to generate code. There are three ways to insert SIMD instructions into the code:

- **Automatic vectorization**: the programmer does not explicitly describe the parallelism. An advanced compiler extracts parallelism from scalar source code.
- **Using SIMD data types**: the programmer defines the relevant variables with a specific data type and uses them in common language expressions.
- **Intrinsic functions**: the language has a library of SIMD functions. The programmer uses them explicitly.

UNIT 2: PIPELINED COMPUTERS

THEME 1.-PIPELINED INSTRUCTIONS UNITS

1.-CONCEPT OF PIPELINING

A process is decomposed into several subprocesses. Each subprocess is independently executed in an autonomous module. Each module works concurrently with the rest.

Latches keep data stable during the time required by a module to perform its function. A clock synchronizes the advance of data among stages. The clock defines:

- When new data can enter the pipelined unit.
- The time available for each stage to perform its function.

Clock period:

- **Ideal case**: same delay in all the modules.

$$T = \frac{D}{k}$$

  D: original circuit delay.

  k: number of stages.

- **Real case**: modules with variable delays, latches and clock skew.

$$T = \max_{i=1;k}(t_i) + T_R + T_S \geq \frac{D}{k}$$

  T_i: delay of module i.

  T_R: delay of the inter-stage register.

  T_S: skew of clock.

Speed-up:

$$S = \frac{T_{np}}{T_p}$$

  T_{np}: time to process n instructions in the original unit.

  T_p: idem in the pipelined unit.

- **Ideal case**:

$$S = \frac{D}{\tau} = k$$

- **Real case**:

$$S = \frac{D}{\tau} \leq k$$

Throughput:

- **General expression**:

$$\chi = \frac{n}{T}$$

n: processed data.

T: required processing time for n data.

- **Non-pipelined unit**:

$$\chi_{ns} = \frac{n}{T_{ns}} = \frac{n}{nD} = \frac{1}{D} \text{ results/s}$$

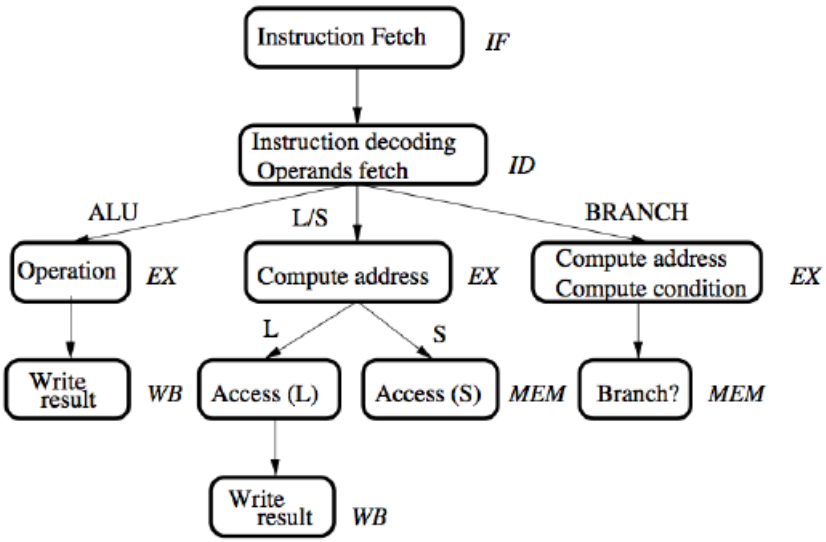
- **Pipelined unit**:

$$\chi_S = \frac{n}{T_s} \approx \frac{n}{n\tau} = \frac{1}{\tau} \text{ results/s}$$

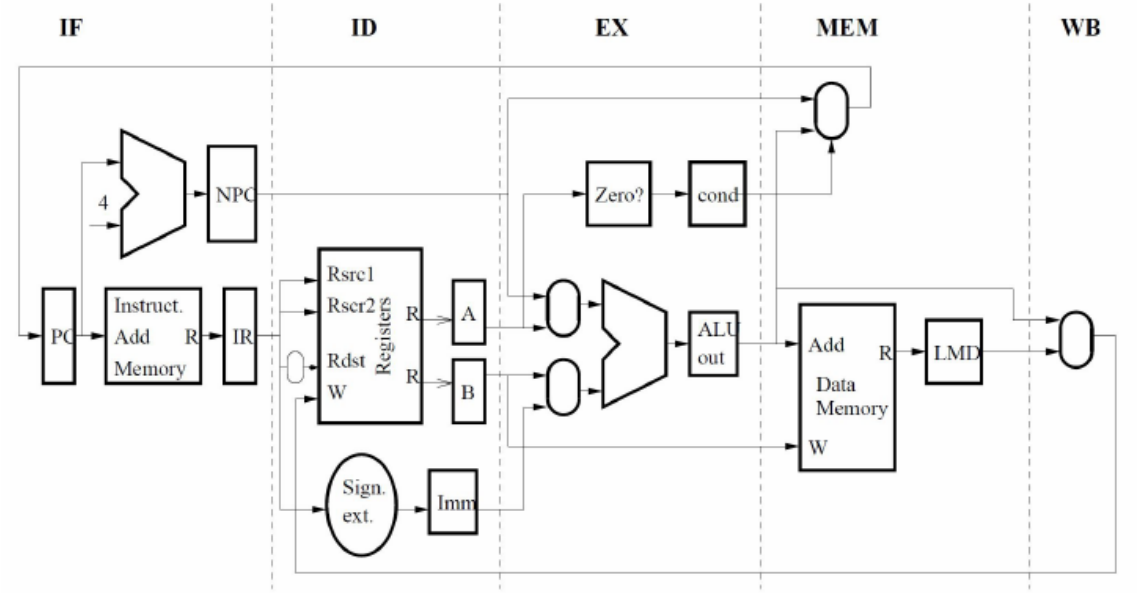
1 result per clock cycle

2.-THE INSTRUCTION CYCLE

MIPS instruction cycle:



MIPS datapath:



3.-INTRUCTION CYCLE PIPELINE

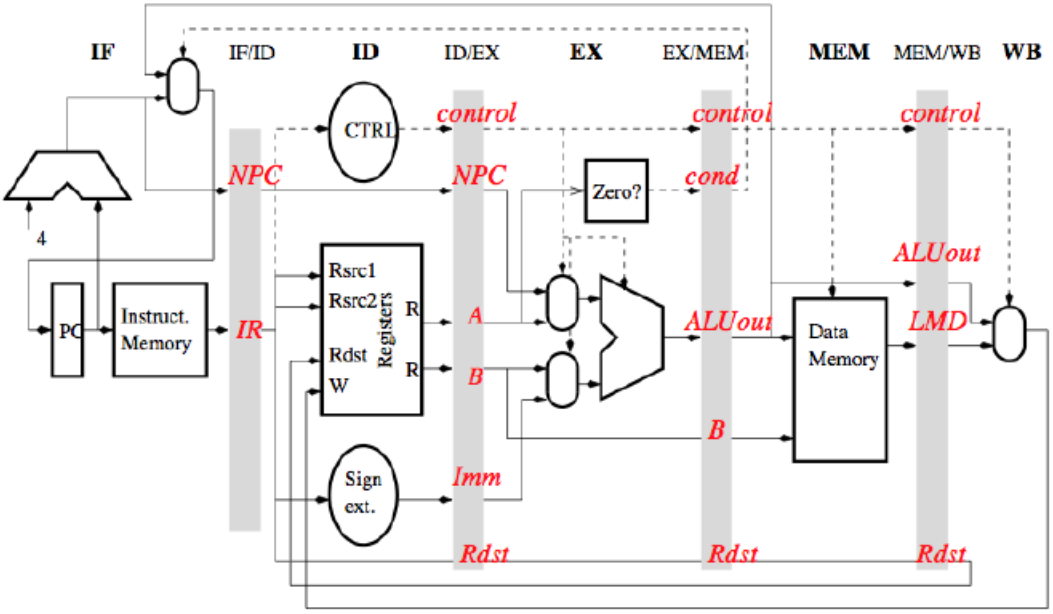
The problem of pipeline is that stages executed by instructions varies from one instruction to another. The solution is to assume that all instructions traverse the same stages:

- IF, ID, EX, MEM, WB.
- Some instructions do nothing in some stages.

Hardware requirements:

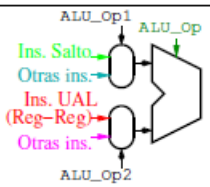
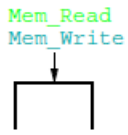
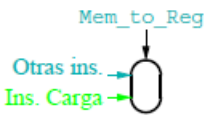
- In a single cycle there are 5 instructions simultaneously in execution, so we have to avoid hazards in functional units.
- There is one operator (EX) and one adder (IF).
- Separated instruction (IF) and data (MEM) caches.
- Register files with two simultaneous read (ID) and one write (WB) ports.
- Cache access time remains constant, but the required bandwidth is 5 times larger.

Pipelined datapath:



4.-CONTROL OF THE PIPELINED INSTRUCTION CYCLE

Control signals:

IF	ID	EX	MEM	WB
Mem_Read PC_Src	Reg_Read	ALU_Op ALU_Op1 ALU_Op2	Mem_Read Mem_Write	Reg_Write Mem_to_Reg
				

Required control signals in EX, MEM and WB stages are generated at stage ID and they travel through the pipelined unit. Identifier of destination register follows the same path as the data to be written into. Both information are simultaneously provided to the register file. PC management logic is moved to stage IF (PC is incremented every cycle).

Events and signals in the pipelined instruction cycle:

	All instructions
IF	IF/ID.IR ← Mem[PC] IF/ID.NPC ← PC+4 if EX/MEM.cond then PC ← EX/MEM.ALUout else PC ← PC+4
ID	ID/EX.A ← Regs[IF/ID.$IR_{6..10}$] ID/EX.B ← Regs[IF/ID.$IR_{11..15}$] ID/EX.Imm ← ((IF/ID.IR_{16})^{16} ## IF/ID.$IR_{16..31}$ ID/EX.NPC ← IF/ID.NPC ID/EX.IR ← IF/ID.IR

	ALU Instr. Reg–Reg/Reg–Imm
EX	EX/MEM.IR ← ID/EX.IR EX/MEM.ALUout ← ID/EX.A op ID/EX.B ‖ EX/MEM.ALUout ← ID/EX.A op ID/EX.Imm EX/MEM.cond ← 0
MEM	MEM/WB.IR ← EX/MEM.IR MEM/WB.ALUout ← EX/MEM.ALUout
WB	Regs[MEM/WB.$IR_{16..20}$] ← MEM/WB.ALUout ‖ Regs[MEM/WB.$IR_{11..15}$] ← MEM/WB.ALUout

	Load/Store
EX	EX/MEM.IR ← ID/EX.IR EX/MEM.ALUout ← ID/EX.A + ID/EX.Imm EX/MEM.cond ← 0 EX/MEM.B ← ID/EX.B
MEM	MEM/WB.IR ← EX/MEM.IR MEM/WB.LMD ← Mem[EX/MEM.ALUout] ‖ Mem[EX/MEM.ALUout] ← EX/MEM.B
WB	Regs[MEM/WB.$IR_{11..15}$] ← MEM/WB.LMD ‖

	Branch BEQZ/BNEZ
EX	EX/MEM.ALUout ← ID/EX.NPC + ID/EX.Imm EX/MEM.cond ← ID/EX.A op 0 ‖ ≠ 0

5.-HAZARDS

A **hazard** is a situation leading to the execution of some instructions generating results that are not consistent with the ones produced by the non-pipelined datapath, thus, a loss of binary compatibility. Hazards originated by two or more instructions simultaneously present in the pipelined instruction unit.

Hazard **types**:

- **Data**: the result of one instruction is used as input data in the following one(s).
- **Control**: branch instructions modify the flow of instructions.
- **Structural**: two or more instructions want to use the same resource.

Hazard **solutions**:

- **Stall insertion**: stop the instruction originating a hazard, and the following ones, but continue with the execution of those preceding such instructions (otherwise the hazard will never disappear). During these cycles no instruction is fetched (stalls). It has a performance degradation:

$$CPI_S = CPI_{S_{ideal}} + \frac{stalls}{instruction} = 1 + \frac{stalls}{instruction} > 1$$

- **Datapath modification**: modify the hardware to dynamically solve hazards. It reduces the number of stalls required to solve the problem. It is a complete solution of the problem (it is sometimes impossible).
- **Compiler modification**: avoid the problem by avoiding the generation of certain sequences of instructions. The main drawback is that the binary compatibility with datapaths that accept any sequence of instructions is lost.

6.-DATA HAZARDS

Some instructions rely on the results of previous ones, so pipelining the instruction unit may change operand access order. The solution to this data hazards is to delay those operations originating the hazard.

There is another alternative, **datapath modification**:

- **Multiport register file** supporting simultaneous read and write (-1 stall cycles).
- **Modification of the access time** to the register file (-1 stall cycles). Register write is done in the first part of the cycle and register read in the second. There is no problem if the register file access time is less or equal than the half of the clock period and it simplifies the design of the register file: simultaneous read and write operations are not required.

** Possible exam question: control logics *

Control logic of **stall insertion**:

- First cycle:

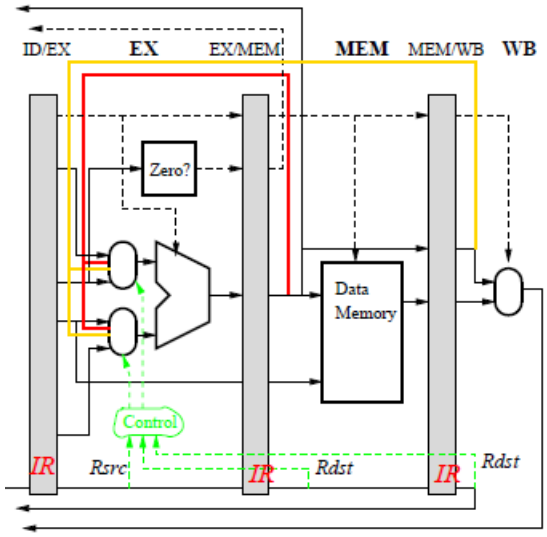
```
if ((ID/EX.IR_CODOP = "ALU") and
    (IF/ID.IR_CODOP = "ALU") and
    ((ID/EX.IR_Rdst = IF/ID.IR_Rsrc1) or (ID/EX.IR_Rdst = IF/ID.IR_Rsrc2)
then
    IF.stall, ID.stall, ID.nop
```

- Second cycle:

```
if ((EX/ME.IR_CODOP = "ALU") and
    (IF/ID.IR_CODOP = "ALU") and
    ((EX/ME.IR_Rdst = IF/ID.IR_Rsrc1) or (EX/ME.IR_Rdst = IF/ID.IR_Rsrc2
then
    IF.stall, ID.stall, ID.nop
```

Forwarding is based in the idea of that, if there are two consecutive instructions with a data hazard, the second one really needs the data at the beginning of execution cycle, and not at decodification. In order to do this, we add a bus from the ALU output (MEM stage) to its inputs (EX stage) + control logic), so we are doing like a "**short-circuit**" from MEM to EX. When the third instruction is considered, the hazard can be solved in a similar way, implementing a short-circuit between WB and EX.

The forwarding implementation is made by adding buses and multiplexors in the ALU input.



Control logic of **forwarding**:

- Shortcircuit from MEM to EX:
  - Rsrc1:

```
if ((EX/MEM.IR_OPCODE = "ALU") and
    (ID/EX.IR_OPCODE = "ALU") and
    (EX/MEM.IR_Rdst = ID/EX.IR_Rsrc1))
then
    Shortcircuit MEM-to-EX (high part of ALU inputs)
```

  - Rsrc2:

```
if ((EX/MEM.IR_OPCODE = "ALU") and
    (ID/EX.IR_OPCODE = "ALU") and
    (EX/MEM.IR_Rdst = ID/EX.IR_Rsrc2))
then
    Shortcircuit MEM-to-EX (low part of ALU inputs)
```

- Shortcircuit from WB to EX:
  - Rsrc1:

```
if ((MEM/WB.IR_OPCODE = "ALU") and
    (ID/EX.IR_OPCODE = "ALU") and
    (MEM/WB.IR_Rdst = ID/EX.IR_Rsrc1))
then
   Shorcircuit WB-to-EX (high part of ALU inputs)
```

  - Rsrc2:

```
if ((MEM/WB.IR_OPCODE = "ALU") and
    (ID/EX.IR_OPCODE = "ALU") and
    (MEM/WB.IR_Rdst = ID/EX.IR_Rsrc2))
then
    Shorcircuit WB-to-EX (low part of ALU inputs)
```

When a load-dependent and consecutive instruction appears, even using a shortcircuit from WB to EX, it is necessary to insert 1 stall cycle. The required control logic for this is:

```
if ((ID/EX.IR_OPCODE = "LOAD") and
    (IF/ID.IR_OPCODE = "ALU") and
    (ID/EX.IR_Rdst = IF/ID.IR_Rsrc))
then
    ID.stall, IF.stall, ID.nop
```

7.-CONTROL HAZARDS

In the instruction flow, some instructions modify the value of the PC. Branch instructions modify the PC in the MEM stage. When this cycle is reached, 3 instructions have already been issued.

One solution is to insert stalls whenever a branch instruction is decoded, but it has a loss of performance. The control logic to this insertion is:

```
beqz r1,dest    IF ID EX ME WB
<dest>             if if if IF ID EX ME WB
```

```
if ((IF/ID.IR_OPCODE = "Branch") or
    (ID/EX.IR_OPCODE = "Branch") or
    (EX/MEM.IR_OPCODE = "Branch"))
then
    IF.stall, IF.nop
```

Another solution is **fixed prediction** (datapath modification):

- **Predict-not taken**: assume the branch is not taken, so the instructions following the branch are correct. At the end, if the branch is taken, these 3 instructions must be cancelled. These instructions must not modify the computer state.
- **Predict-taken**: assume the branch is taken, so once the destination address is known, new instructions are fetched. If the branch is finally not taken, such instructions are aborted. It is only useful if the destination address is known before the branch condition not useful in the case of the MIPS.

Control logic of Predict-not-taken:

```
if (EX/MEM.cond) then
    IF.nop, ID.nop, EX.nop
```

In order to reduce the number of cycles between fetching the branch instruction and computing the branch destination we do (datapath modification):

- Computation of destination address moves from EX to ID, so we need an additional adder.
- Evaluation of the condition moves from EX to ID.
- Update the PC in:
  - Stage EX: num. of cycles = 2.
  - Stage ID (at its end): num. of cycles = 1. In stage ID the new PC is known.

Control logic for **updating the PC in the ID** stage:

```
  beqz r1,dest    IF ID EX ME WB
  <sgte>             IF X
  <dest>                IF ID EX ME WB
if (IF/ID.IR_OPCODE = "Branch")
then
    if (Regs[IF/ID.IR_Rsrc] op 0)
    then
        IF.nop
        PC <- IF/ID.NPC + IF/ID.Imm
    else
        PC <- PC + 4
```

These changes have an impact on the clock period: ID becomes the slowest stage, so branch condition must be simple. This modification may be incompatible with the requirement of reading the registers in half a cycle.

Data hazards where branches are involved require the use of short-circuits in the ID stage and/or the insertion of stalls.

** 100% question in the exam *

Possible **data hazards**:

Instrucciones	Ejemplo	Cortocircuito	*stalls*	Fig.
ALU - ALU	DADD R1,R2,R3			
	DSUB R4,R1,R5	MEM to EX	0	1
	AND R7,R1,R6	WB to EX	0	2
Load - ALU	LD R1,20(R2)			
	DADD R3,R1,R4	WB to EX	1	2
ALU - Load/Store	DADD R1,R2,R3			
	LD R2,20(R1)	MEM to EX	0	1
	LD R3,40(R1)	WB to EX	0	2
ALU - Store	DADD R1,R2,R3			
	SD R1,20(R2)	WB to MEM	0	3
	SD R1,40(R2)	WB to EX	0	4
Load - Store	LD R1,20(R3)			
	SD R1,20(R2)	WB to MEM	0	3
	SD R1,40(R2)	WB to EX	0	4
Load - Load/Store	LD R1,30(R3)			
	LD R2,20(R1)	WB to EX	1	2
ALU - Branch	DSLT R1,R2,R3			
	BEQZ R1,loop	MEM to ID	1	5
ALU - Branch	DSLT R1,R2,R3			
	...			
	BEQZ R1,loop	MEM to ID	0	5
Load - Branch	LD R1,20(R2)			
	BEQZ R1,loop	–	2	–
Load - Branch	LD R1,20(R2)			
	...			
	BEQZ R1,loop	–	1	–

8.-CODE ARRANGEMENTS

The concept is the addition of a new step to the compilation process: code rearrangement. It enables getting by without circuits specifically designed for solving hazards, so it has a possible reduction of the CPI and t at the cost of binary compatibility loss. Even if those circuits exist, code rearrangement can prevent stalls, thus, there is a CPI reduction. If code rearrangement is not possible, NOP instructions are inserter. This makes a slightly increase of I and, in the worst case, similar performance to the insertion of stalls.

Even when using shortcircuits, an instruction just following a load and presenting a data hazard with such load always requires a stall. Then, we use delayed load, that is that the compiler ensures that no instruction following a load reads from the loaded register.

A **load delay** slot is the number of instructions after a load that cannot read from the loaded register. This number depends on the existing instruction pipeline. 1 cycle in the case of the MIPS.

In **Delayed branch**, the compiler places instructions after branches that will be executed regardless of whether branches are taken or not. Since these instructions will be always executed, they do not require any cancellation or the insertion of any stall. In general, the compiler selects instructions preceding the branch that do not present any hazard with the branch condition.

A **branch delay slot** is the number of instructions after the branch that will be always executed despite the branch will be taken or not. This value is equal to the branch latency. It is 1 cycle if the PC is updated during the ID stage.

The use of instructions presenting hazards with the branch condition is acceptable if such instructions have a high-probability of being executed. If it is not possible to find out such type of instructions, NOP's are used.

The instructions used to fill the delay slot come from:

Instruction	Conventional	Delayed Branch
Preceding	ADD R1,R2,R3 BEQZ R2,dst ...	... BEQZ R2,dst ADD R1,R2,R3
Following	ADD R2,R2,R3 BEQZ R2,dst ... OR R7,R8,R9 ... dst: SUB R4,R5,R6	ADD R2,R2,R3 BEQZ R2,dst OR R7,R8,R9 ... ... dst: SUB R4,R5,R6

At destination	ADD R2,R2,R3
	BEQZ R2,dst
	...
	...
	OR R7,R8,R9
	...
	dst: SUB R4,R5,R6
	AND R10,R11,R12
		ADD R2,R2,R3
		BEQZ R2,dst
		SUB R4,R5,R6
		...
		OR R7,R8,R9
		...
		dst: AND R10,R11,R12
		...

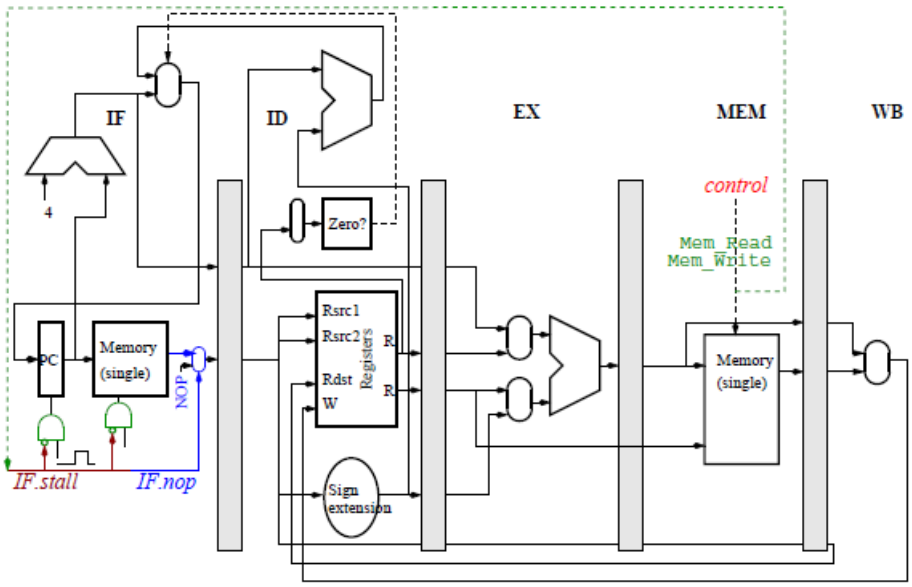
Strategy	Restrictions	Performance improvement
Preceding	Branch condition does not depend on the instruction.	Always
Following	If condition is met, it must not lead to any problem even when executed.	If the condition is not met.
At destination	If condition is not met, it must not lead to any problem even when executed. If the destination can be reached from other points, it is also necessary to copy the instruction to them.	If the condition is met.

9.-STRUCTURAL HAZARDS

The cause of **structural hazards** is due to that hardware does not allow all possible combinations between instructions in the unit. A resource has not been replicated enough. These are the **solutions**:

- **Datapath modifications**: replicate the resource in order to enable such combination of instructions. This increases the cost and replicating the resource is not always possible or makes sense.
- **Stall insertion**: delay the instructions generating the hazard. These stalls make a loss of performance. The best approach depends on the % of each type of structural hazard.

A great example of structural hazard is when we don't have a separated cache for data and instructions, thus we can't access to memory at the same time in ID and MEM. These would be the datapath modified for this case:



And this would be the control logic:

```
if ((EX/MEM.Mem_Read) or (EX/MEM.Mem_Write))
then
  IF.stall,  IF.nop
```

10.-EXCEPTIONS

There are several types of **interrupts** or **exceptions**:

- **Synchronous** vs. **asynchronous**. It is synchronous if the event is triggered at the same location on every program execution.
- **User-driven** vs. **raised to the user**.
- **Maskable** vs. **unmaskable**.
- D**uring the execution of an instruction** vs. **between instructions**.
- **Continue the program execution** vs. **end the program**.

The most difficult ones are those exceptions raised to the user that are provoked during the execution of instructions, when the program continues its execution.

MIPS possible **exceptions**:

Stage	*Exceptions*
IF	Instruction page fault, Misaligned access Violation of protection, E/S request
ID	Ilegal instruction, E/S request
EX	Arithmetic exception, E/S request
MEM	Data page fault, Misaligned access Violation of protection, E/S request
WB	E/S request

There are several instructions under execution when an exception occurs in a pipelined computer, thus, an incorrect behaviour is presented. The PC of instructions is only kept until ID.

First pipelined machines terminated the execution of programs by printing the PC of the current instruction in IF. They signalled approximately the instruction originating the exception, but it was an **imprecise exception**.

- A computer supports a precise behaviour in the presence of an exception if:
- Instructions preceding the one generating the exception correctly finish their execution.
- The instruction raising the exception and the following ones are aborted.
- After handler completion it is possible to restart the program from the instruction originating the exception.

Then, it is possible to identify the instruction raising the exception and the behaviour is identical to the one exhibited by a non-pipelined computer.

The **requirements** for **precise exceptions** in the MIPS are:

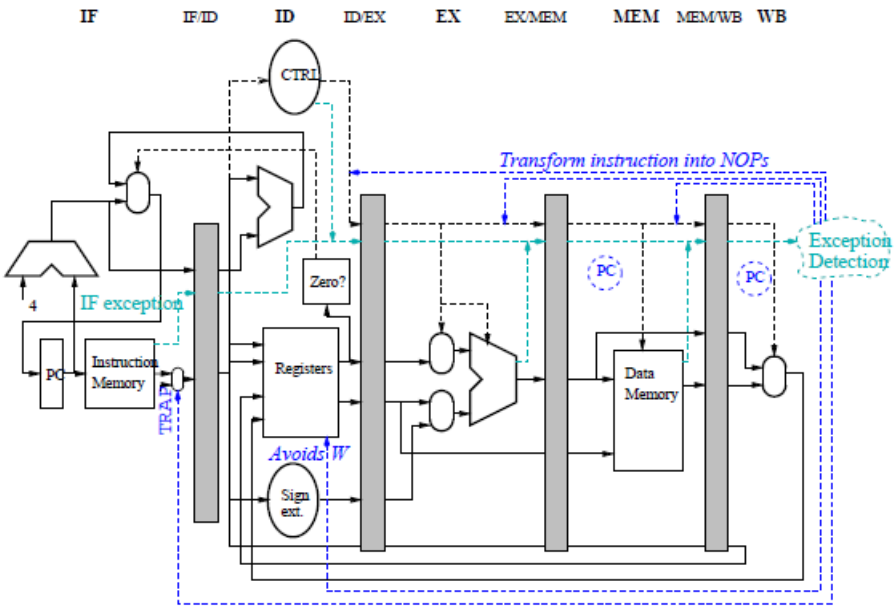
- **More than one exception** (up to 5) can be **raised** in the same or in **different clock cycles**.
- It is necessary to **take into account** how the **branch delay technique** works.

The idea is to ensure a natural order when handling exceptions, thus, instructions must reach the last stage of the pipeline in order.

Steps:

1. Each instruction entering the unit is related to a register with as many bits as stages in the pipeline able to raise an exception. Such register travels through the pipeline together with the instruction.
2. If an exception is raised, the register bit of the corresponding stage becomes “1”. At the same time, the instruction generating the exception becomes a NOP.
3. During the last stage, register’s bits are checked If any bit is set, the following instructions become NOP and a TRAP is generated in the IF stage for the following cycle.
4. The PC of the instruction raising the exception is stored. If the branch delay technique is used the PC of delay slot+1 instructions must be also stored.
5. The exception handler takes the control.
6. Once the handler(s) ends its execution, the PC (or PCs) is (are) restored, thus following the execution from that point.

Datapath modified:



THEME 2.-STATIC INSTRUCTION SCHEDULING

1.-MULTICYCLE OPERATIONS

Floating point instructions and **some integer instructions** need **more time** at the EX stage. The **solutions** to this problem are:

- **Increase the clock period** of the instruction unit, but this makes that the machine slows down.
- Add **more hardware** to keep EX taking the same time required by simple instructions. This increases the cost and it is not always possible.
- Enable EX stage of complex operations to take **several clock cycles** (multicycle operations), so the clock period does not change.

As new instructions require specific hardware, new specialized operators are added instead of a single multifunction operator. During ID stage, the instruction is issued to the appropriate operator. Once the operation ends, it is sent to the MEM stage.

New **multicycle operators** can be either conventional or pipelined. The **typical parameters** are:

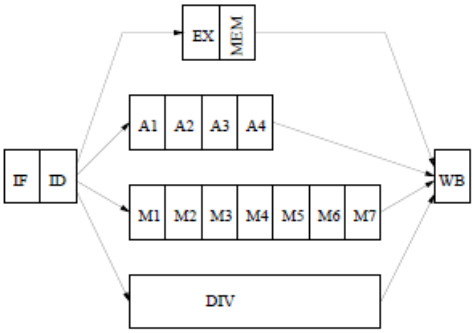
- **Latency** or **evaluation time** (time required to obtain the first result).
- **Initiation rate IR** (reciprocal of the time between results). If it is pipelined, IR = 1.

The new pipelined MIPS including new operators require:

- New integer-stage registers.
- A multiplexor to handle WB stage inputs.

Unnecessary structural hazards are avoided by ending the execution of stores in MEM.

Stages of this model:



When we use units with IR < 1 operation by cycle, structural hazards appear:

- **Two instructions shouldn't be in the same operator**, so the second instruction must wait at ID, by inserting stalls, until the operator becomes free. The disadvantages of this solution are that ID concentrates all the detection logic and as soon as one instruction is stalled at ID, no further instructions are fetched (IF).

- **Several instructions in WB**. It reduces the number of simultaneous accesses to the register file during the WB. One partial solution is to separate integer from floating point register files. Other solutions are:
    - Increase the number of ports: on average only one writing operation is carried out per cycle. This is not efficient.
    - Stall insertion: ID stage checks whether an instruction writes to the register file at the same time that another one does. Detection logic is concentrated at ID. Stalls at ID prevent fetching new instructions (IF).

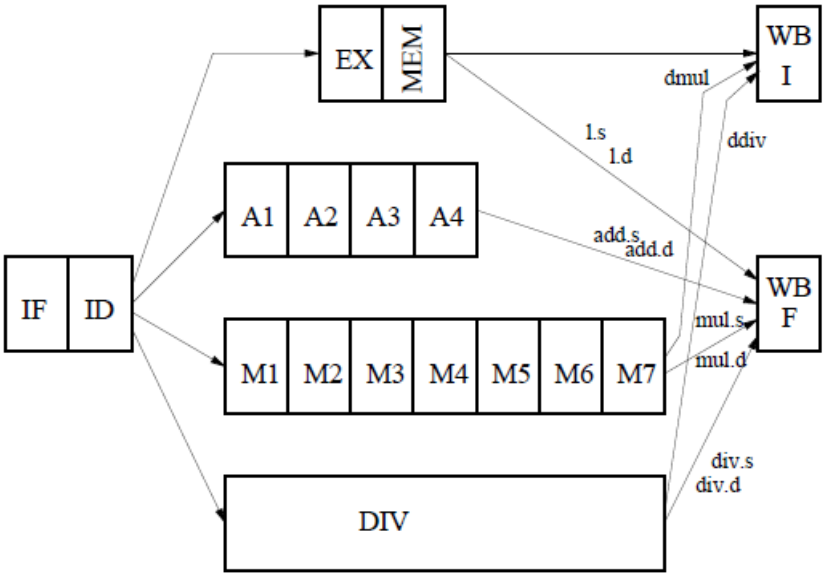
The **advantages** of **independent register files** for floats and integers are:

- **Structural hazards** are **reduced**.
- The **total number of registers** is **doubled**, without making decoding logic more complex, without increasing access time and without adding more bits to the format.
- **Register file bandwidth** is **doubled**, without adding more ports or increasing the access time.

The **drawbacks** are:

- Sometimes, it is necessary to **exchange information between both register files** (MFC0, MTC0 and MFC1, MTC1 instructions).
- The **number of registers** of each type is **limited** a priori.

Datapath with independent register files:



RAW (Read After Write) hazards (data) happen when an instruction produces a result required by a following one. The solution is to insert stalls until a short-circuit can be applied. Another time, this makes that the detection logic is concentrated at ID and stalls at ID prevent instructions fetching (IF).

With multicycle operations, the execution stage may take several clock cycles. This penalty (number of stalls) can be very important.

WAW (Write After Write) hazards happen when two close instructions write to the same register. The problem is that the order of write operations is not correct. The solution is to detect at ID and insert stalls. Although a good compiler should not generate code with two write operations to the same register without any intermediate read.

The presence of multicycle operations may alter the instruction execution order. When an exception occurs, some of the instructions following the one triggering the exception may have already finished.

2.-TYPES OF DEPENDENCIES

The problem is that pipelined unit with multicycle operations produce many stalls, so the CPI is not approximately one.

Instruction Level Parallelism or **ILP** is the potential overlapping in the execution of instruction sequences. It relies on the independence among the considered instructions. The relation is:

More ILP -> few conflicts -> few stalls -> Less CPI

Two instructions are independent if they can be simultaneously executed without any problem and they can be reordered. There are three **types of dependencies**:

- **Data dependency**.
- **Name dependency**.
- **Control dependency**.

Given two instructions i and j, j logically after I, a **data dependency** exists if:

- i produces a result used by j.
- There exists a data dependency between i and k, and k produces a result used by j. This chain can be as long as the program is.

Name dependencies occur when two instructions use the same register or memory location, but there is no data flow between them. Given two instructions i and j, j logically after i, the following name dependencies may occur:

- **Anti-dependency**. An anti-dependency occurs when an instruction requires a value that is later updated.
- **Output dependency**. An output dependency occurs when the ordering of instructions will affect the final output value of a variable.

An instruction is **control dependent** on a preceding instruction if the outcome of the latter determines whether the former should be executed or not. Every instruction in a program (except those at the beginning of the program) has a control dependency with some branch.

Summary:

Type of dependency	Hazard
Data dependency	RAW
Anti-dependency	WAR
Output dependency	WAW
Control dependency	Control hazard

3.-IMPROVING ILP

The goal is to increase the ILP of instructions currently under execution in the pipelined unit. Sequences of instructions between branch instructions are named **basic blocks**. We need to determine the amount of parallelism that can be extracted from the execution of the instructions in each basic block.

The statistics show that 15% of instructions are branches, so there are 6 or 7 instructions per basic block. Instructions in a basic block usually exhibit dependencies among them. The solution to this is to exploit the existing ILP among multiple basic blocks: instructions from different basic blocks are executed in parallel.

A particular case is the one concerning loop-level parallelism, which exploits the ILP existing among loop iterations by overlapping them.

In order to increase the ILP by overlapping basic blocks we use:

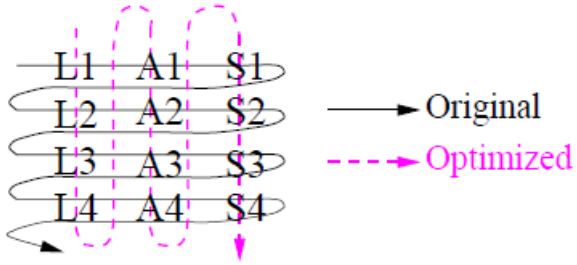
- **Static instruction scheduling**: the compiler reorders/modifies the code.
- **Dynamic instruction scheduling**: hardware reorders instructions at runtime.

In static instruction scheduling, the compiler reorders/modifies the code to increase ILP by reducing/deleting existing dependencies and their effects (hazards and stalls). Some static instruction scheduling techniques are:

- **Loop unrolling**: moves dependent instructions away from each other and exploits ILP across several basic blocks.
- **Software pipelining**: reorders the code in order to move dependent instructions away from each other.

4.-LOOP UNROLLING

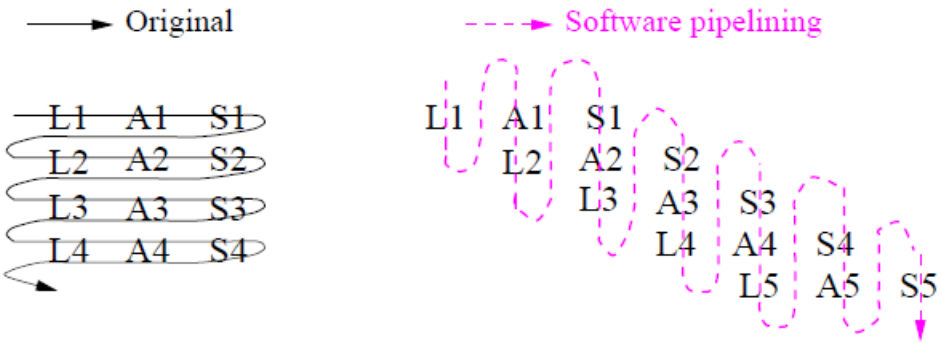
The **loop code** is **replicated several times**, thus reducing the number of iterations to be executed. It reduces the overhead produced by loop control instructions. Increasing the size of the basic block increases the freedom of the compiler to separate instructions exhibiting data dependencies:



It is necessary to perform register renaming to eliminate name dependencies. Dependent instructions have been separated by inserting as many instructions between them as the number of required stalls in the worst case, so no stalls are required.

5.-SOFTWARE PIPELINING

The basic idea is to **transform a loop with dependent instructions and independent iterations into another loop with dependent iterations and independent instructions**:



The name of the technique relates to its ability to process the original loop by simulating the behaviour of a pipelined unit:

- The “data” of the pipelined unit are the iterations.
- To avoid intermediate results rewriting, execution is performed from the last stage to the first one.

THEME 3.-DYNAMIC BRANCH PREDICTION

1.-INTRODUCTION

The goal of branch prediction is to **predict the behaviour of branches in order to have instructions fetched** (and even under execution) **by the time the final result of the branch becomes available**.

The behaviour of the different branch instructions in a program is not usually the same and the behaviour of a given branch instruction uses to vary across the program execution, thus, we must use dynamic branch prediction (supported by hardware).

2.-BRANCH PREDICTION BUFFERS (BPB)

The goal is to predict branch conditions. The data is the branch instruction address and the result is just **taken** or **not taken**.

The **mechanism** is:

- A table indexed by the least significant bits of the branch instruction address.
- Each table entry contains the prediction for that branch instruction.

The **table location** has several options:

- Small fast memory that is accessed in parallel with the instruction cache during IF.
- Adding prediction bits to each instruction cache block when using direct mapping.

The **predictor prediction algorithm** is:

- Finite state automaton.
- State changes according to the actual behaviour of the branch instruction.
- The prediction (“taken”/“not taken”) depends on the state.

The simplest case of this algorithm is the **one-bit predictor**.

Two branch instructions may share their least significant bits, so prediction may fail if their behaviour is different. The solution is to use more PC bits for the table index, but this increases the table size.

Branch instructions controlling **loop iterations** follow a **predictable pattern**:

- If the loop contains n iterations, the branch is taken n - 1 times, and only once it is not taken (for the last iteration).
- A one-bit predictor will fail twice: in the first iteration (it is not trained yet) and in the last one.

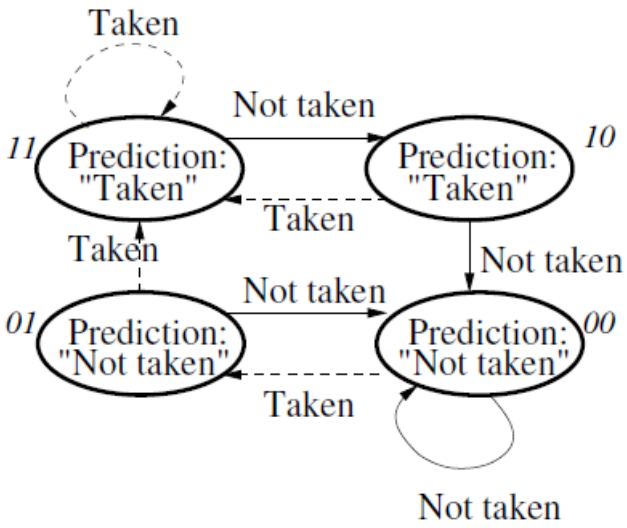
The prediction for an inner loop fails twice for each iteration of a outer loop. The solution to not fail is a two-bit predictor.

With **two-bit states**, the automaton has four states:

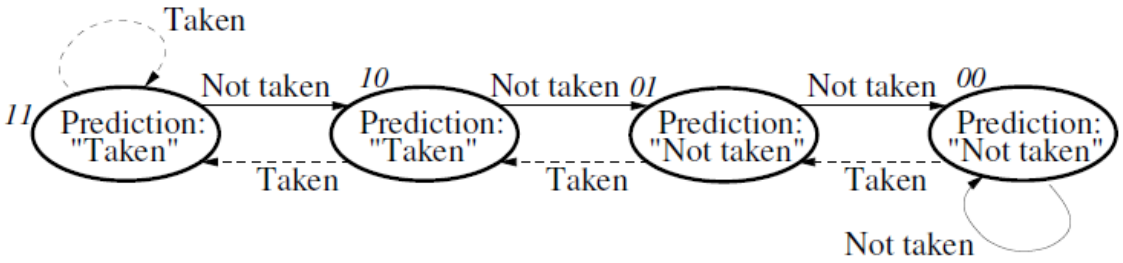
- Strongly not taken (state 00).
- Weakly not taken (state 01).
- Weakly taken (state 10).
- Strongly taken (state 11).

The basic idea is that a prediction must fail before being modified. The two most usual designs are:

- **Two-bit predictor with hysteresis**.



- **Two-bit predictor with saturation**. Prediction must fail 2^{n-1} times before being modified.



Another type of predictors is the **two-level** or **correlating predictors**. In addition to the behaviour of the current branch instruction (local history) they consider the behaviour of other branches (global history). A predictor (g: global; l: local) uses the behaviour of the last g branches in order to choose among 2^g l-bit predictors.

It also exists the **hybrid predictors**: several predictors plus a selection mechanism. The basic idea is that each predictor is suitable for different patterns and that by combining the use of different predictors and applying the most suitable one according to the needs, it is possible to increase the prediction accuracy.

The **selection mechanism** chooses the predictor that provided the best results up to now. The **implementation** of the selection mechanism is a table of suturing counters indexed by the branch instruction address.

3.-BRANCH TARGET BUFFERS (BTB)

The goal is to predict the branch condition and the branch target address. The data is the branch instruction address and the results are **taken** or **not-taken**, the branch target address (if taken) and whether or not the instruction is a branch.

It is a completely associative table with three fields:

- **Index**: branch instruction address.
- **Prediction**: "Taken" or "Not taken".
- **Address**: branch target address.

Events at stages of the instruction unit:

- IF If (EXISTS instruction in BTB table). // It is a branch.
    - If (prediction == "taken"). Start fetching instructions during next cycle from the address provided by the table. // Zero penalty cycles.
- ID Instruction decoding. If (instruction == branch).
    - Compute branch target address and condition.
    - If (EXISTS instruction in BTB table)
        - If (prediction != condition). Abort incorrectly fetched instructions. Update the prediction bit in the table entry. Start fetching instruction from the correct address
    - else // There was no entry in the table for the instruction. Add entry to the table for the instruction

The effective behaviour depends on the prediction accuracy: high predictor hit rate. In practice, the branch condition may depend on a previous long-latency instruction, and thus, it may take long to be computed. In this case, the improvement achieved by BTB would be much higher.

Implementation:

- The table stores the complete address of the branch instruction, since this address is required during IF, before decoding the instruction. A hit in the table only occurs if the instruction is a branch for sure.
- It is not mandatory to implement a fully associative table. In practice, it is a set-associative table (the least significant bits of the PC are used to index sets and they are not stored in the table).
- The BTB and the predictor can be decoupled:
    - BHT stores prediction bits
    - BTB stores the target address of the last "taken" branches. The size of the BTB can be small.

  If the prediction of the BHT is "taken" and there is a hit in the BTB, it starts fetching instructions from the target address.