# Advanced Computer Architecures notes

Elia Ravella

March 18, 2021

# Contents

Part I

# Introduction

## 1 Taxonomy of Computer Architectures

### 1.1 Flynn Taxonomy

Flynn defined four categories to group computing architectures. This so called taxonomy helps system architects to distinguish among possible models of processors or system themselves.

1. SISD: Single Instruction Single Data, uniprocessor systems;
2. MISD: Multiple Instructions Single Data, no commercial application;
3. SIMD: Single Instruction Multiple Data, low overhead, used in custom integrated circuits;
4. MIMD: Multiple Instruction Multiple Data, off the shelf micros

SISD is the simplest and most diffuse model of computation, one data stream in input and a *single* instruction executed per istant of time. SIMD introduces parallelism in a vector fashion: multiple processes execute a single instruction (the *same* instruction) on different data elements. Another type of parallelism: MIMD. Just SIMD with multiple different instructions per instant.

Part II

# Performance and Cost

## 2 Performance

### 2.1 Evaluating performance

We need to distinguish between how to measure performance of a computer architecture and what impacts it without affecting performance. The amount of memory used is not a performance evaluator, but it affects performance in some way. The power consumed, the latency and the execution time are. To evaluate the performance of a CA I must use all the performance indicator and find the right tradeoff. Also, context! A CA is inserted in a context that bounds its scope, so the environment itself shapes the way performance is evaluated
*Every performance indicator* says something about the CA. So, the whole system must be evaluated with a *whole set* of indexes that each tell a different thing about the system itself. This provides a full view of the system strenghts and weaknesses and highlights the things to optimize.

## 2.2 Quantifying the Design Process

"The frequent case" case: when designing a CA, the "frequent case" represents the set of conditions met *the most times* during a standard computation run. Optimizing the standard case means optimizing *singnificately* all the system behaviour.

**Amdahl's Law** Amdahl's law describes the concept of the speedup as a function of execution time. The set of formulas that describes the law is

$$EXTIME_{new} = EXTIME_{old} \times [(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}] \tag{1}$$

$$SPEEDUP_{overall} = \frac{EXTIME_{old}}{EXTIME_{new}} = (1 - Fraction_{enhanced} + \frac{Fraction_{enhanced}}{Speedup_{enhanced}})^{-1} \tag{2}$$

and calculates the speedup of a system taking into account all his part, the speedupped one and the invariate one.

## 2.3 Indexes on performance

Overview of performance indicators and measurements.

- Instruction Count: the effective number of instructions executed. This is influenced also by the program we're trying to execute.
- Tyme per Cycle: the clock rate. OBVIOUSLY influenced by the hardware architecture.
- MIPS and MFLOPS indicates already a very *program bound*-like perfomance indicator. They represents the number of integer/floating point operations that the architecture can carry out in a second. They're very synthetic indexes.
- CPI - Average CPI (clock per instruction) represents obviously yhe amount of clock cicles that are necessary to carry out a specific operation. The element that most impacts the CPI is the ISA: the alphabet we use shapes the words we can create.

Notice how execution time is not present. That's because *reducing execution time is the goal*, so execution time does not modify performance, it just *represents* it.

## 2.4 Benchmarking

Benchmarking is "running a fake program with the sole purpose to evaluate a system and find bottlenecks". It's a technical test. Workloading a CA means feeding it with a real work load to test what is a response to a non-syntethic stimulus.
Benchmarking must be

1. representative: we need to benchmark the right aspects of the architecture we're testing.

2. updated: old benchmark does not test the architecture well, because of the very architecture is changed and the benchmark is *not anymore representative* because it's *old.*

### 2.5 Energy, Power

Energy and power consumption are a "cost - like" performance indicator. The less the power, the less the cost of keep the architecture running, and tha's easy. Also, reducing the power consumed means *reducing the power needed from a battery* and this impacts a lot in the mobile scenario. Energy per task is a good indicator for energy consumption. The power used by an architecture is influenced by a ton of factors, not least the TDP (thermal design power). Also architecture-bound aspects (clock rate, voltages, busses) influence power consumption *heavily.*

# Part III
# Pipelining

## 3 MIPS architecture

Reference architecture for this course is MIPS.
MIPS is a RISC architecture (Reduced Instruction Set Computer) this means that the only possible operation (that are very optimized) are basic operations. This kind of architecture is the opposite of CISC (Complex ISC) where more complex operation are supported at chip level.
MIPS is a LOAD/STORE architecture. This means that data cannot come into the ALU from anywhere besides the CPU general purpose registers. MIPS uses 32-bit instructions that can be

- Register Register operations
- Register Immediate operations
- Branch operations
- Jump/Call operations

Operand lenght is 6-bit. We have 64 possible operations.

### 3.1 Assembly basics

We'll see three classes of MIPS instructions:

- ALU: add "reg target" "reg source" "reg source", or sub "reg target" "reg source" "immediate"
- LOAD/STORE: lw "reg target" "offset register" (load word)
- Branches/Jumps: classic bne or bge or jmp operations

## 3.2 MIPS instruction execution

A fetch - decode - execute cycle in this architecture works this way:

1. Fetching the instruction means sending the PC content to the memory in order to access the memory location where the instruction is. Then, the PC is updated (PC + 4, each instruction take 32 bits).

2. Decoding the instruction: in this phase a fixed-field operation is put in place to deconde the instruction itself, and the registers needed for the computation are read.

3. Execute operation: the ALU calculates the result of the instruction supplied.

4. Memory Access and Write Back phases: in these two phases the CPU takes care of the memory access (LOAD or STORE instructions) and the registers update.

## 3.3 MIPS Chip Architecture

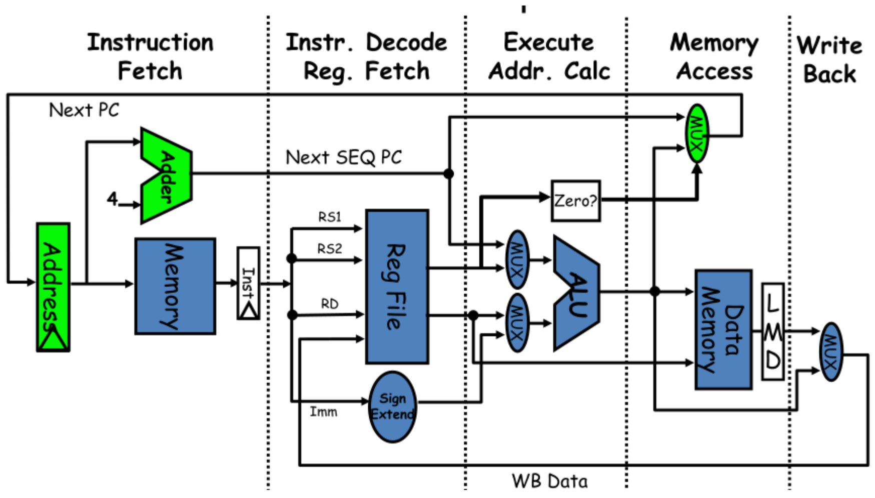


Figure 1: The MIPS data path with pipeline stages highlighted

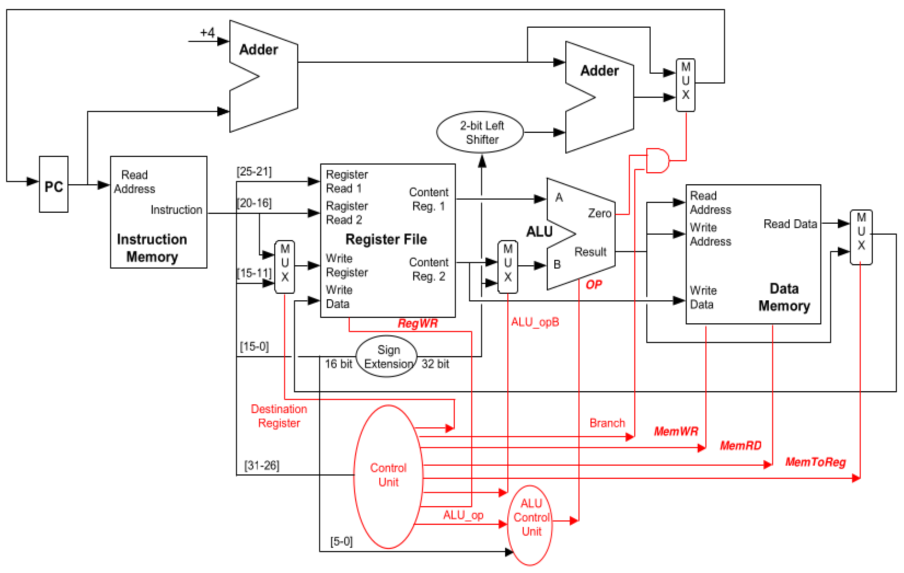


Figure 2: The MIPS full CPU, data path integrated with control unit

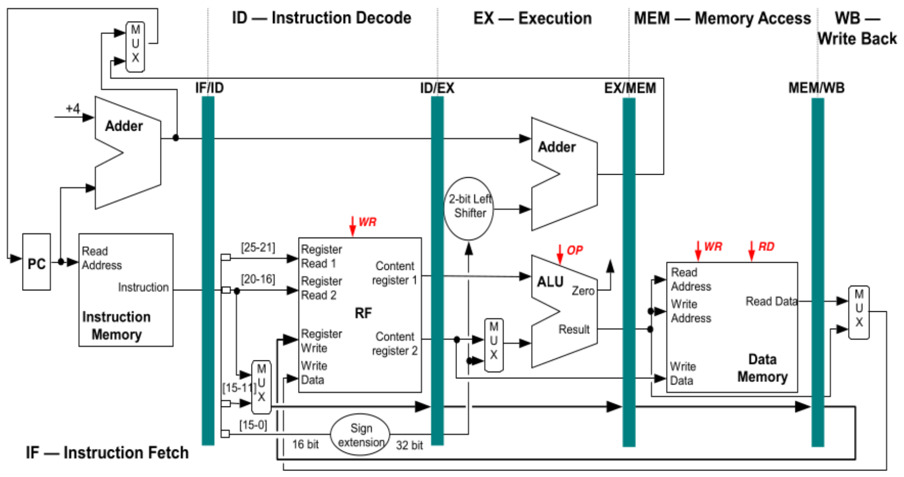


Figure 3: Pipelined MIPS

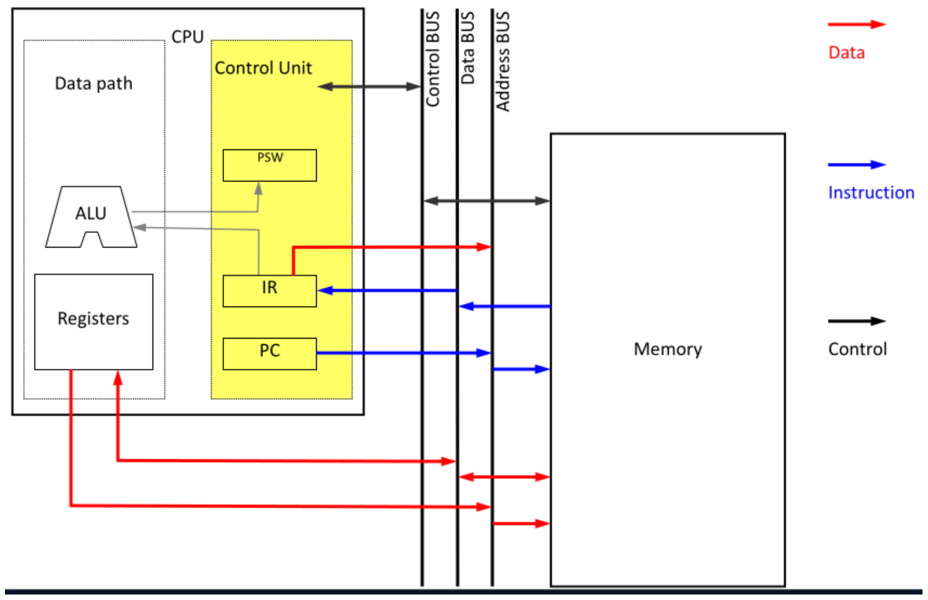
## 4 Data and Control

Every computer architecture is composed of 2 main parts: the one that computes data (datapath) and the one that controls the execution (control unit).

**Datapath** The datapath runs the house. All data pass through this module, and the registers here are used to pipeline operation.

**Control Unit** The CU guides the computations. It takes ”run time decisions” in order to pilot execution of intructions. The registers here are architecture-specific.

**Memory** And memory? "Slow primary memory" is connected with the CPU (so DP + CU) by means of buses. The structure is similar to:



Things to be noted:

1. the datapath does not connect to the control path
2. the control unit does not write on the data bus

## 5 Hazards

Pipelining is cool, parallele, throughput enhancing and overall faster. Does she have drawbacks? YES, hazards. Both the ID and the WB phases in the pipeline we have an access to the registers, that can overlap. This is usually overcome assigning (for example) rising edge of the clock to reading and falling edge to writing.
But what are hazards? An hazard is generated by dependency between two instructions. Hazards can be classificated in

1. Structural hazards: the registers example. Different phases of the same pipeline access simoultaneously the same resource.
2. Data hazards: attempting to *reading the future*: accessing a piece of data that's not be computed yet.
3. Control hazards: making a decision basing on a condition *not yet evaluated*.

## 5.1 Structural Hazards

No structural hazards in the reference architecture (edge - based synchronization)

## 5.2 Data Hazards

Utilizing a piece of data that is not ready is called the condition of data hazard (or data conflict). This is generally generated by two (or more) consecutive instructions that generate and use the same value.
We can further classify the types of data hazards:

**Read after Write** conflicts are generated by two consecutive operations, the first writes a value and the second reads it. In a pipelined environment, this can cause problems. This is the most common hazard.
This kind of conflicts can be handled at compilation time (the compiler itself detects a dependence of this kind and inserts delays in the execution ("bubbles" or "stalls") of the instructions to distanciate "enough" the read and write operation. The cost is incremented overhead and increased execution time. This approach can also be implemented *at architecture level.* The difference is that compiler will delay instructions with NOP instructions, while at architecture level clock cycle are just "skipped".
Another way to resolve this kind of conflicts is to reschedule instructions (taking advantage od the dependencies between them) in order to

1. distanciate enough the write-read conflicts
2. not inserting dead times

Forwarding, instead, is the technique that "bypasses" the WB phase and makes available the result of the ALU evaluation directly in the pipeline. So additional EX-EX, MEM-EX, MEM-ID paths are introduced in the architecture. This approach does not cover certain type of conflicts, like load - use hazards. This conflict, however, is the only one possible in the reference architecture.

**Write after Write** It's a overwriting conflict. This is a problem when a WB stage of a instructions happens *after* the WB phase of a *following* instruction.

**Write after Read** In a out-of-order scenario we can have a write operation of an instruction happens *after* the read operation of an instruction *preceding* her.

## 5.3 Control Hazards

Conditional instructions at assembly level are jumps (oversimplifying). The jump target address is computed just after the condition is evaluated (it can be PC + 1 or "TargetAddress").
In the reference pipeline the PC is updated in the last pipeline stage, BUT we can skip the MEM phase because the PC is updated in the MEM phase itslef. Tis kind of instructions (beq, bne) terminates after the MEM stage. This means that further instructions must wait for the MEM phase of the conditional

statement in order to be fired, so *at least 4 cycles*.
The standard procedure to deal with this kind of instructions structure is to *preload in the pipeline the "true" branch*, and then flush it if the condition is not met. Also, data forwarding reduces the number of stalls to be introduced: EX-ID path, for example, reduces the number of stalls from 3 to 2.

**Early Evaluation** What if we anticipate the condition evaluation *the most that we can*? We can move an ALU for conditions in the second stage of the pipeline (ID) in order to recompute faster the condition. This reduces the number of bubbles needed to 1.
This technique combined with the "betting on the true branch" reduces the scope of the pipeline flush needed in the case the bet result is bad. BUT early evaluation introduces data hazards when the same registers must be accessed by two consecutive instructions where the second one is a branch.

### 5.3.1 Branch Prediction

Ok, so we reduced the flush scope of a prediction to one instruction (with the early evaluation of branch condition). Can we go further? Yes, we can pre-load a branch of the conditional execution in order to anticipate the jump.
These techniques can be classified in two major families, static and dynamic, if they're fixed or acts basing on the data they've got.

#### Static Branch Prediction Techniques

- Branch Always Not Taken: the next instruction to be fetched is PC + 4. Easy.
- Branch Always Taken: next instruction to be fetched is the one computed by ID stage.
- Backward Taken Forward Not Taken: this techniques "try to predict basing on the direction of the jump". So the assumption is "forward jumps are conditions, we enters them; backward jumps are loops, we always take them".
- Profile Driven Prediction: statistic approach. Thismethod can use compiler hints.
- Delayed Branch: rescheduling. This approach takes an instruction "near" the branch instruction and reschedule it after the branch one, because either if the branch is taken or not *no penalty clock cycle* are to be taken. Obviously, the moved instruction must not be data dependent from the condition or the branch itself.

**Dynamc Branch Prediction Techniques** How can we enhance the "branch bet" choice at runtime? Basing on the data we gather at runtime, change the predicted branch. This is implemented with an associative table that links *PC bits with target addresses.* The whole predictor mechanism can be described by two parallel computations:

**Branch Outcome Prediction** BOP is the procedure that selects the actual label to jump to. If we're using a one-bit predictor, the behaviour is schematized by a super simple FSA that acts this way:

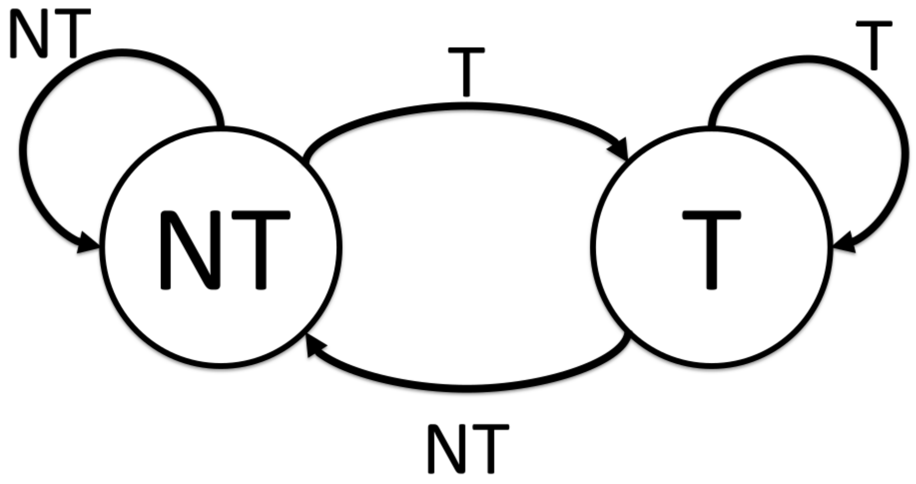
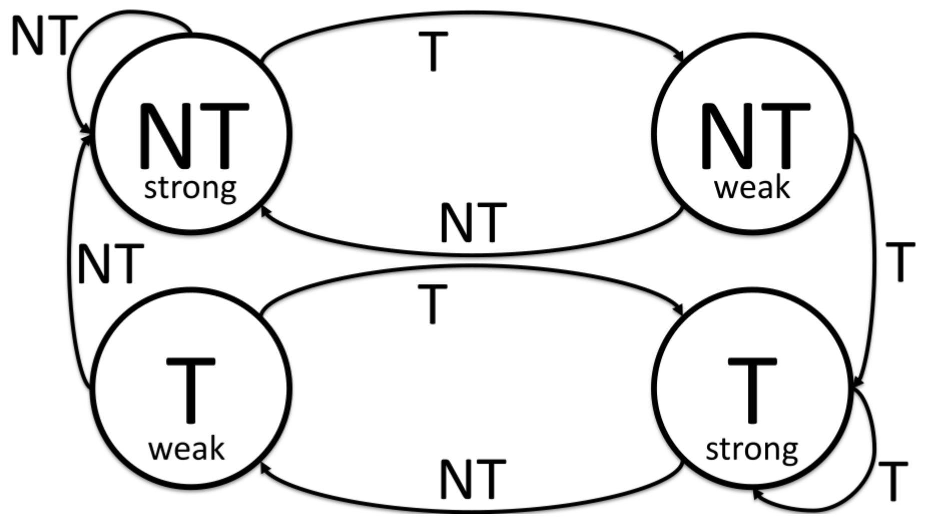


Figure 4: The NT stands for Not Taken, and T for Taken

This models the behaviour of the predictor of *which branch is to be taken.* Important remark: the BHT FSA may *overlap predictions* (in the case for example of nested loops) if the prediction is based on different loops. An enhanced model use a 2-bit register to switch state:



We could also add a third, fourth bit to the memory of the FSA, but sperimental data show that a 2-bit BHT predictor is the best tradeoff between complexity and accuracy.
Another approach is to lenghten the scope of the prediction: what if two threads of execution have a similar pattern in branches, but they're "unlucky"? We

could add a dedicated branch predictor that matches previous behaviours with the current one to look for patterns and better decide which path to take. This "multilevel predictors" relies on 2 factors: the lenght of the *behaviour branch* to be analyzed **and stored** and the number of bits used to discrimine the branch taken (so as the simple predictor: in fact, simple FSA predictors are (0, 2) correlating predictors).