

# Detailed summaries of the classes of Arquiteturas de Alto Desempenho 2024/2025

## Tomás Oliveira e Silva

Theoretical lectures:

Date	Summary
23/09	Rules. Overview of the main features of a modern processor.
30/09	TBD
07/10	TBD
14/10	TBD
21/10	TBD
28/10	TBD
04/11	TBD
11/11	TBD
18/11	TBD
25/11	TBD
02/12	TBD
09/12	TBD
16/12	TBD

Practice classes:

Date	Summary
23/09 and 24/09	Comparison of C, Java, Python, Fortran, web assembly, and compiled Python (codon).
30/09 and 01/10	Code parallelization using pthreads.
07/10 and 08/10	Code parallelization using OpenMP.
14/10 and 15/10	Code parallelization using sockets (server/client).
21/10 and 22/10	Code parallelization using MPI (Message Passing Interface).
28/10 and 29/10	Code parallelization using CUDA and OpenCL (part 1).
04/11 and 05/11	Code parallelization using CUDA and OpenCL (part 2).
11/11 and 12/11	First assignment.
18/11 and 19/11	First assignment.
25/11 and 26/11	Simulation of combinational logic described in VHDL.
02/12 and 03/12	Simulation of sequential logic described in VHDL.
09/12 and 10/12	Second assignment.
16/12 and 17/12	Second assignment.

**[Preliminary notes]** These summaries do not replace attending (and paying attention) to the theoretical lectures. To better organize the contents of these extended summaries, sometimes we took the liberty of moving things around: something reported as being lectured on a given day may actually have been lectured in the previous or in the next week. They also contain extra information not presented in lectures.

## September 23, 2024 ([tablet](#))

**[Grades]** One written exam in January (grade  $T$ ), and reports of two practice assignments (grades  $A_1$  and  $A_2$ ). The written exam has 12 equally valued questions, the best 10 of which will contribute to  $T$ . The practice grade is  $P = \max\left(\frac{A_1+A_2}{2}, \frac{3A_1+2A_2}{5}\right)$ . The final grade is  $F = \frac{T+P}{2}$ . It is necessary to have  $T \geq 7$ , and  $P \geq 7$  to pass.

**[MIPS pipeline overview]** The MIPS pipeline has 5 stages:

**IF** Instruction Fetch: reads instructions, one by one, from a dedicated memory. Increments the program counter.

**ID** Instruction Decode: reads registers, sign- or zero-extends an immediate value.

**EX** Execute: performs an arithmetic or logical operation. The arithmetic operation may be the computation of a memory address. Also, a separate unit computes a branch target address.

**MEM** Memory access: performs a read (load) from a second dedicated memory area, or a write (store) to the dedicated memory area.

**WB** Write Back: commit a new register value to the register file.

The usage of separate address spaces for instructions and data (Harvard architecture) is not practical.

**[Protection levels]** Modern processors execute code in several protection levels (called rings). Ring 0 is for kernel code (operating system and drivers): all instructions are allowed but some may be virtualized. In Intel/AMD, ring 3 is for user programs (for normal users

and even for super users): some instructions are privileged and if the program tries to execute them the program is terminated (the illegal instruction signal is raised). Ring -1 is for hypervisors (handle virtualized instructions). Ring -2 is for system management mode (SMM). Ring -3 is for the Management Engine (ME). These last two ring levels are usually only engaged in the BIOS.

**[Address translation (virtual addresses)]** All modern processors have a virtual memory mechanism. All addresses of a program are virtual addresses. Even the kernel uses virtual addresses (that is what makes virtualization possible). On each memory access the virtual address is translated into a physical address.

**[Caches]** On a modern processor the memory area used in the IF stage is the level 1 instruction cache (L1I), and the one used in the MEM stage is the level 1 data cache (L1D). The two caches fetch/store data in a unified level 2 cache (L2U), that stores both instructions and data (thus, instructions and data share a common address space — von Neumann architecture).

The processor may also have more than one execution pipeline (each pipeline has its own level 1 and level 2 caches). If so, a larger capacity level 3 cache (L3U) is often used. Besides caching data from several pipelines, it also guarantees the coherence of the information in the lower level caches.

Modern Intel/AMD processors can store the decoded instructions in a special cache (Op Cache). This is so because due to their variable length the instructions may take some time to decode.

There exists a special cache, called a Translation Lookaside Buffer (TLB) to cache virtual to physical address translations.

Usually, the caches are accessed using physical addresses.

**[Prefetch]** Modern processors have memory prefetch units that monitor the address access patterns used by the program and prefetch data into the data caches before it is needed. They detect strided memory accesses (arithmetic progressions).

**[Superscalar architecture]** The IF stage of the pipeline of modern processors can fetch more than one instruction in one go. This takes advantage of the wide data path between the instruction cache and the IF unit. The EX and MEM stages also have many execution units. Some may perform arithmetic or logical operations, others may perform floating point operations, others may compute memory addresses, and others may perform loads from and stores to memory. The WB stage is sometimes called the instruction retirement unit, and commits the changes made by the instructions in program order.

All this allows the pipeline to execute more than one instruction at the same time, provided, of course, that the instructions work on independent data. For example, in the following code

```
d = a + b; // 1
e = b + c; // 2
f = a + c; // 3
g = d + e; // 4, depends on 1 and 2
```

the first three statements can be performed in parallel, but the fourth has to wait for the completion of the first two (it is usually not necessary to wait for the instructions to retire, the use of data forwarding paths makes the results available from the execution units as soon as possible). Compilers are aware of this when they generate executable code, For each code block, they construct a so-called dependency graph, and output assembly code that takes advantage of any possible data-level parallelism in the source code.

**[Speculative execution]** Conditional branches are big a problem for the pipeline. The next instruction to be fetched after a conditional branch depends on a test whose outcome will be known only several clock cycles later. Modern processors employ a technique called speculative execution to try to ameliorate this problem: it will continue to fetch and (speculatively) execute instructions. The idea is to use a so-called branch predictor to attempt to predict the outcome of the conditional branch. If the predictor is successful, the IF unit fetched the right instructions and things will proceed at full pace. If it fails, something that will only be known some clock cycles later, the instructions that

entered the pipeline after the mispredicted conditional branch have to be discarded. The penalty is several (up to about 10) wasted clock cycles.

The branch predictor has to be tightly coupled with the level 1 instruction cache. Branch predictors base their decision on the past behavior of the conditional branch.

**[Out-of-order (OoO) execution]** Modern processors can execute independent instructions out of order. This can happen when an instruction takes many cycles to complete. If some instructions that follows it do not depend on its side effects they are executed out of order. They are, however, retired in order. For example, in the following code

```
c = a / b;      // 1, slow (many clock cycles)
d = a + 2 * b; // 2, fast (1 clock cycle)
e = 2 * a + b; // 3, fast
g = d + e;     // 4, depends on 2 and 3, fast
```

the last three statements with very likely finish execution (but are not retired) before the first statement finishes execution. Each pipeline of a modern processor can have many (> 50) instructions in-flight.

Although there exist a fixed number of architectural registers in the instruction set architecture (for example, 16 general purpose registers for 64-bit Intel/AMD processors), due to the many instructions that can be in-flight, the pipeline internally has physical space for many more registers. When each instruction is issued it is assigned to it, using a technique called register renaming, some of the available physical registers. When the instruction is retired, the contents of the destination physical register is transferred to the architectural register file, that holds the official state of the program.

Register renaming is useful to break false dependency chains. For example, in the following code

```
m = a[i];      // 1
m |= 3;        // 2, depends on 1
a[i + 1] = m; // 3, depends on 2
m = a[i + 2]; // 4, does not depend on 2
```

statement 4 can theoretically start execution at the same time as statement 1. Because  $m$  will reside in a given architectural register, without register renaming statement 4 would have to wait for the completion of statement 2 (previous last change of  $m$ ). With register renaming, it can be issued at the same time as statement 1.

**[SIMD]** Besides the regular registers, modern processors also have a set of wide registers. Each wide register stores a small vector of integer or floating point values. For example, a 128-bit register can store a vector of sixteen 8 integers, or a vector of eight 16-bit integers, or a vector of four 32-bit integers, or a vector of two 64-bit integers. The arithmetic or logical operations on these wide register work in parallel. Hence it is possible, for example using 128-bit registers, to add in parallel, in one clock cycle, say, four 32-bit integers.

**[SMT]** Modern processors also allow two or more execution flows on the same pipeline (this is called Simultaneous MultiThreading, SMT; Intel calls it hyperthreading, but that is just a marketing term). The threads that execute on the same pipeline share almost all of its resources. The exception is the register file: each thread has its own register file. SMT can make good use of execution units of the pipeline when one execution flow stalls due to a lengthy operation (say, a memory read, or a partial pipeline flush due to a mispredicted conditional branch).

**[Many cores]** Finally, all modern processors have multiple execution pipelines (one execution pipeline is one physical core). Due to simultaneous multithreading, one physical core will give rise to two or more virtual cores (vCPUs in cloud parlance). Modern server processors can have more than 100 physical cores!

**[All of the above]** All the features described above makes writing a super fast program for a modern processor a challenging task. The compiler takes care of some of the details, but to make a program super fast, the programmer should be aware of what may slow a program down. In particular, the size of the data caches is something that sometimes has to be taken in consideration. Also, if the problem to be solved allows

it, SIMD instructions and multiple execution flows (threads) should be used. This last point will be explored in depth in the practice classes.

The figure on the next page, extracted from a recent [AMD presentation at Hop Chips 2024](#), depicts the general architecture of a modern server processor pipeline. The floating point part of the pipeline is displayed in violet, and the integer part in light blue. Although there exist 16 architectural integer registers, there exist 240 physical integer registers! The floating point situation is even more extreme: 32 SIMD architectural registers and 384 physical registers. In each clock cycle, the pipeline can handle up to 6 integer instructions, 4 address calculations (the AGU is an Address Generation Unit), and 4 floating point instructions. In each clock cycle, it can also handle 4 read data transfers and 2 write data transfers from/to the L1 data cache.

# “Zen 5” Microarchitecture Overview

2 Threads/Core

NextGen Branch Predictor

Caches

- I-Cache: 32KB, 8-way; 2x 32B fetch/cycle
- Op-Cache: 6K inst; 2x 6-wide fetch/cycle
- D-Cache: 48KB, 12-way; 4 mem ops/cycle
- L2-Cache: 1MB, 16-way

Dual I-Fetch/decode pipes, 4 inst/pipeline

8 ops/cycle dispatched to Integer or FP

Execution capabilities

- 6 integer ALU
- 4 AGU, 4 addresses to LS per cycle
- 4 FP ops/cycle; 2cycle FADD

TLBs

- L1: 64entry ITLB, 96entry DTLB
- L2: 2K ITLB; 4K DTLB everything but 1G

