Computers

Computer Structure

Grau en Ciència i Enginyeria de Dades

Facultat d’Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

2021-2022 Q2
Creative Commons License

This work is under a Creative Commons Attribution 4.0 Unported License

The details of this license are publicly available at https://creativecommons.org/licenses/by-nc-nd/4.0
• Computer structure. Components
  • Processor – The chip
    • CPU / core
    • Multi- and many-core
    • Simultaneous multithreading / hyperthreading
  • Memory – and memory hierarchy
    • Data / instruction caches
    • Main memory
  • Input/Output components
From a Computer System to the Processor

- A computer system consists of hardware and systems software...
  - ...to run application programs
- Components can be different, but the concept remains the same
Computer Architecture

- **Computer Architecture**
  - Set of rules that state how **hardware and software components** are combined and interact with each other to work as a functional device

- **Von-Neumann* Architecture (mostly used)**
  - The design states how instructions and data are obtained and processed:
    1) The CPU receives instructions and data
       - From an input device or memory
    2) CPU processes instructions and data
    3) Results are sent out
       - To an output device or memory

- **Harvard Architecture**
  - Alternative approach (e.g. DSPs)

Processors

- M Chips
  - N cores/chip
  - T threads/core
- LLC – last level cache memory
Processors

• What do we need?
  • A program – sequence of instructions
    • Or multiple sequences... if concurrent/parallel
  • Data – operands should reach the instructions

• Exercise... where should we store instructions and data?
• Exercise... how do we generate executable programs?
Processor Architecture

- **Instruction Control Unit**
  - Manage Instruction Execution

- **Register File**
  - Word-sized storage closest to the CPU

- **Cache Memories**
  - Mem hierarchy to reduce CPU-mem gap

- **Bus Interface**
  - Circuitry for memory transactions

- **ALU (Arithmetic/Logic Unit)**
  - Simple operations

- **Other Units**
  - Complex and/or accelerated operation-based units
Hardware Thread

• Each hardware thread independently...
  • Fetches instructions*
  • Decodes
  • Issues load memory accesses*
  • Executes*
  • Stores results*

* When executing a single thread per core, then such a thread has all core resources available!
  - Memory bandwidth
  - Functional units

• Multithreading
  • Execute multiple threads in parallel
Thread-Concurrency

• The Operating System manages concurrency (multiple simultaneous activities) and take advantage of the different levels of parallelism abstraction

• From the thread (i.e. execution unit of a running program) concurrency point of view
  • Multi-processor
    • Multiple processors managed by the same Operating System
  • Multi-core
    • Multiple CPUs integrated in the same chip
  • Multi-thread
    • A CPU executes multiple flows of control
    • Some Hw resources are replicated
      • E.g. program counter, registre files, simple ALU
    • Some Hw resources are shared
      • E.g. complex ALU, FPU
Executing Instructions

• The **Instruction Control Unit**...
  • **Fetches** the next sequence of instructions
    • What-if it is not clear what is the very next instruction?
  • Instruction **Decoder** translates from instructions to operations or microoperations
    • Depends on the ISA (Instruction Set Architecture)
  • **Register File** smallest and fastest memory
    • It is at the top of the memory hierarchy
  • **Retirement Unit** keeps track of processing to commit results in the right order

• The **Execution Unit**...
  • **dispatches** the requested primitive operations to the respective functional units
  • Functional Units are the ones **perform the operation**
    • Load, Store, ALU, FP, ...
Instruction Set Architecture (ISA)

• The Instruction **Set Architecture (ISA)** defines...
  • Instruction types and format
  • Operation encodings
  • Addressing modes
  • Registers
  • Parallelism approaches (e.g. SIMD)
  • ...

• ISA complexity trade-offs: Semantic Gap
  • Closer to the high-level language vs Closer to the hardware control signals
    • Reduced Instruction Set Computing (**RISC**), such as MIPS, ARM, RISC-V
    • Complex Instruction Set Computing (**CISC**), such as Motorola 68K, early x86
    • Hybrid between CISC-RISC, such as more recent x86
    • Others...
  • Impact on coding and complexity of compiler, among other features of the microarchitecture

• **Assembly language** is the text form of the machine code (low-level instructions and registers)
Register File

• Small amount of memory available in the CPU
• Fastest way to access to data
• Different size of registers (depend on ISA)
  • 64-bit, 32-bit, but also sub-registers to access 16 and 8 bits
• Different categories according to the purpose
  • General-Purpose Registers (GPRs)
    • E.g. x86-64 has 16 full-width GPRs and 52 subregisters
      • RAX, EAX, AX, AH, AL
  • FPU Registers
  • Others sets of registers, among others...
    • memory addressing support (e.g. segment), pointers (e.g. Instruction Pointer), control
SIMD Parallelism

- **SIMD Parallelism**: Single Instruction Multiple Data
  - Example based on x86-64
    - SSE (Streaming SIMD Extensions, 128-bit); AVX2 (Advanced Vector eXtensions, 256-bit); AVX512 (512-bit)
    - Registers + Instructions

- SISD, **SIMD**, MISD, MIMD
  - Single vs Multiple
  - Instruction; Data

**SSE Data Types (16 XMM Registers)**

- _m128
- _m128d
- _m128i
- _m128i
- _m128i
- _m128i

**AVX Data Types (16 YMM Registers)**

- _mm256
- _mm256d
- _mm256i

256-bit Integer registers. It behaves similarly to _m128i. Out of scope in AVX, useful on AVX2

4x 32-bit float
2x 64-bit double
16x 8-bit byte
8x 16-bit short
4x 32-bit integer
2x 64-bit long
1x 128-bit quad
4x 32-bit float
4x 64-bit double
ALU (Arithmetic and Logic Unit)

• From Lesson 1 – Data Representation and Basic Operations
ALU (Arithmetic and Logic Unit)

- Example of ALU with 4-bit opcodes

- C: Carry flag
- V: Overflow flag
- S: Signal flag
- Z: Zero flag

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>Src0 + Src1</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>Src0 − Src1</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>Src0 * Src1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>Src0 / Src1</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>Shift left (Src0) by Src1</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>Shift right (Src0) by Src1</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>Rotate left (Src0) by Src1</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>Rotate right (Src0) by Src1</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>Src0 AND Src1</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>Src0 OR Src1</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>Src0 XOR Src1</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>NOT(Src0)</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>NOT(Src1)</td>
</tr>
<tr>
<td>1 1 x x</td>
<td>Reserved for future use</td>
</tr>
</tbody>
</table>
Register + ALU/FPU

- Operations take different latencies depending on complexity and optimizations.
Example of code

```
rrmovq %rcx, %rax
- 0 as cc: always
- 1 as reg: %rcx
- 0 as reg: %rax
addq %rdx, %rax
subq %rbx, %rdi
- 0 as fn: add
- 1 as fn: sub
jl 0x84
- 2 as cc: l (less than)
- hex 84 00... as little endian Dest: 0x84
rrmovq %rcx, %rdx
rrmovq %rax, %rcx
jmp 0x68
```

Example obtained from slides of University of Virginia (link)
Hardware Organization

- **Processor or Central Processing Unit (a.k.a. CPU)**
  - Execute instructions stored in memory

- **Main Memory**
  - Volatile Storage (e.g. RAM)

- **Buses**
  - Wired connections
  - Chunks of bytes

- **I/O Bridge**
  - Interconnection Hub
  - Northbridge/Southbridge

- **I/O Devices**
  - Device, Controller, Adapter
  - Non-volatile Storage (e.g. Disk)
Example of a Virtual Assistant Teardown

I/O Slot (USB)
Main Memory
Main Processor (CPU)
Microphone

https://www.briandorey.com/post/amazon-echo-input-teardown
Example of Multiprocessor Motherboard

- **Motherboard**: the main circuit board in which components are connected to

  - Processor slots (CPU Sockets)
  - Memory slots (RAM)
  - Storage slots (SATA)
  - Network slot (Ethernet)
  - I/O slots (PCIe)
  - I/O slots (USB)

[Source](https://community.mellanox.com/s/article/understanding-numa-node-for-performance-benchmarks)
Table of Contents

• Computer structure. Components
  • Processor – The chip
    • CPU / core
    • Multi- and many-core
    • Simultaneous multithreading / hyperthreading
  • Memory – and memory hierarchy
    • Data / instruction caches
    • Main memory
  • Input/Output components
Memory Hierarchy

- **RAM**: Basic unit storage is a cell (one bit per cell)
  - Static RAM (SRAM) for **cache hierarchy**
  - Variants of Dynamic (DRAM) for **main memory**
    - E.g. SDRAM, DDR(2, 3, 4) SDRAM
    - SRAM faster (~10x) and more expensive (~100x) than DRAM
Memory Hierarchy

- **RAM**: Basic unit storage is a cell (one bit per cell)
  - Static RAM (SRAM) for cache hierarchy
  - Variants of Dynamic (DRAM) for main memory
    - E.g. SDRAM, DDR(2, 3, 4) SDRAM
  - SRAM faster (~10x) and more expensive (~100x) than DRAM

- **Caching**: use smaller but faster storage devices as staging of items stored in larger and slower mems
  - Usually processors comprise from L0 (registers) up to L3
  - Reduce the memory
  - Exploiting Locality...
Locality

- The locality principle is the tendency to access items that are close to other items in space or time
  - **Spatial Locality**: from a given access to a memory address, nearby memory addresses will be referenced as well
  - **Temporal Locality**: from a given access to a memory address, the same address will be referenced again within a short period of time

- If there is high spatial locality...bring not only the requested bytes, but also the very next bytes

- If there is high temporal locality...keep the requested bytes in closer levels of the memory hierarchy

- Parallelism approaches also exploit locality
  - E.g. SIMD
Example of Locality

• What types of localities can you see in the following code snippet..
  • for instruction references?
  • for data references?

```c
func (vec, n)
{
    tmp = 0;
    for (i=0; i<n; i++)
        tmp += vec[i];
    return tmp;
}
```
Cache Organization

- A cache consists of...
  - An array of sets
  - A set contains one or more lines
  - A line comprises...
    - A Valid bit
    - Few tag bits
    - A block of Bytes of data

- A given memory address is decomposed
  - Mem address of \( m \) bits
    - \( t \) bits for tag
    - \( s \) bits for set
    - \( b \) bits for block offset

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>B−1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ E \text{ lines per set} \]

\[ S = 2^s \text{ sets} \]

\[ \text{Cache size: } C = B \times E \times S \text{ data bytes} \]
Cache Hierarchy

- **Instruction** cache, **data** cache, and **unified** cache

Examples

**AMD**
- Private L1
- Shared L2
- Shared L3

**Intel Core i7**
- Private L1
- Private L2
- Shared L3

<table>
<thead>
<tr>
<th>Cache type</th>
<th>Access time (cycles)</th>
<th>Cache size (C)</th>
<th>Assoc. (E)</th>
<th>Block size (B)</th>
<th>Sets (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 i-cache</td>
<td>4</td>
<td>32 KB</td>
<td>8</td>
<td>64 B</td>
<td>64</td>
</tr>
<tr>
<td>L1 d-cache</td>
<td>4</td>
<td>32 KB</td>
<td>8</td>
<td>64 B</td>
<td>64</td>
</tr>
<tr>
<td>L2 unified cache</td>
<td>10</td>
<td>256 KB</td>
<td>8</td>
<td>64 B</td>
<td>512</td>
</tr>
<tr>
<td>L3 unified cache</td>
<td>40–75</td>
<td>8 MB</td>
<td>16</td>
<td>64 B</td>
<td>8,192</td>
</tr>
</tbody>
</table>
Write Strategies

• What if there is a write hit?
  • Write-through
    • As soon as there is a write in the k-level of cache, it is sent (updated) to the k+1 level
    • Simpler to implement, but increases bus traffic
    • A buffer (write-buffer) can be used to improve memory update performance
    • Read misses are lower cost because it does not trigger a write
  • Write-back
    • When a write is performed in the k-level of cache, it is NOT sent to the k+1 level. It is deferred as long as possible
    • Fewer transfer \(\rightarrow\) more available bandwidth to memory for I/O devices

• What if there is a write miss?
  • No-write-allocate (a.k.a. write around)
    • Data is not loaded to the cache, but it is written directly to the next level of cache
    • Data is only loaded on read misses
  • Write-allocate (a.k.a. fetch on write)
    • Data is loaded to cache, followed by a write-hit operation
    • Tries to exploit spatial locality of writes, but increases traffic

• Usually...
  • …lower levels of cache are write-through; Higher levels of cache are write-back
  • …Write-through are typically no-write-allocate; write-back are typically write-allocate
Requirements: Consistency & Coherency

- Accesses reference main memory locations, not cache locations
  - Cache memories are transparently managed by the hardware
  - **Memory coherency**: any read from any cpu to a particular memory @, returns the most recently written value to that @
  - **Memory consistency**: ensure writes to different memory @ will be seen in the correct order from all cpus

- There are protocols (e.g. snoopy) to fulfill such requirements

```c
void add_vectors( float * c, float * a, float * b )
{
    int i;
    for (i=0; i < N; i++) c[i] = a[i] + b[i];
}
```
Cache Miss

• Miss penalty: additional time due to a miss in the k-Level of cache

• Types of cache misses
  • **Cold (compulsory) miss**...
    • occur when the cache is empty
  • **Conflict miss**...
    • occur when multiple items collide in the same block
      • E.g. \((i \mod 4)\)...0, 4, 8, 12, 16... all these items collide in the block 0
  • **Capacity miss**...
    • Occur when the **working set** (i.e. active cache blocks) is bigger than the cache blocks

• Direct impact of
  • **Placement policy**: determines where the block goes
  • **Replacement policy**: determines what block is evicted to put the new one
Detailed Memory Access (MISS)

- Load instruction, data is not on the caches
- Also, fetching instructions, instructions are not on the caches(*)

(*) L1 Insn would be used for fetching instructions
Detailed Memory Access (HIT)

- Load instruction, data present in L1 Data
- Also, fetching instructions, instruction present in L1 Insn (*)

(*) L1 Insn would be used for fetching instructions
Detailed Memory Access (Eviction)

• Cache management is a complex hardware feature
  • What happens when the cache is already full of data... and the core needs to bring more?
  • Cache eviction... Last recently used data may be evicted to the next cache level
Let’s see few examples

- We have a simple computer with the following internal structure:
  - Level 1 cache
    - 8 cache lines
    - 8 bytes per cache line
    - 64 bytes total
Example 1

• We run the following program:

  ```c
  double V[16];       // using addresses to 0x100 to 0x17f
  int i;               // using a CPU register
  
  for (i=0; i < 16; i++)
      V[i] = 0.3;
  ```

• Determine the specific **hits, misses** and **capacity conflicts** that happen in the L1 data cache

• Assume there is no interaction with a possible L1 Instruction cache
Example 1

- We run the following program:

```c
double V[16]; // using addresses 0x100 to 0x17f
int i; // using a CPU register

for (i=0; i < 16; i++)
    V[i] = 0.3 + (double) i;
```

<table>
<thead>
<tr>
<th>i</th>
<th>V[i]</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.3</td>
<td>0x0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>1.3</td>
<td>0x0108</td>
</tr>
<tr>
<td></td>
<td></td>
<td>miss</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
<td>0x0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>miss</td>
</tr>
<tr>
<td>3</td>
<td>3.3</td>
<td>0x0118</td>
</tr>
<tr>
<td></td>
<td></td>
<td>miss</td>
</tr>
</tbody>
</table>

![Diagram showing memory access and cache behavior](image-url)
Example 1

• We run the following program:

```c
double V[16]; // using addresses 0x100 to 0x17f
int i; // using a CPU register

for (i=0; i < 16; i++)
    V[i] = 0.3 + (double) i;
```

<table>
<thead>
<tr>
<th>i</th>
<th>V[i]</th>
<th>Address</th>
<th>Cache Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.3</td>
<td>0x0120</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>5.3</td>
<td>0x0128</td>
<td>miss</td>
</tr>
<tr>
<td>2</td>
<td>6.3</td>
<td>0x0130</td>
<td>miss</td>
</tr>
<tr>
<td>3</td>
<td>7.3</td>
<td>0x0138</td>
<td>miss</td>
</tr>
<tr>
<td>4</td>
<td>4.3</td>
<td>0x0140</td>
<td>???</td>
</tr>
</tbody>
</table>

RAM memory

L1 data cache
Example 1

• We run the following program:

```c
double V[16]; // using addresses 0x100 to 0x17f
int i; // using a CPU register

for (i=0; i < 16; i++)
    V[i] = 0.3 + (double) i;
```

![Diagram of RAM memory and L1 data cache showing cache misses and capacity conflicts](39)
Example 1

- We run the following program:

```c
double V[16]; // using addresses 0x100 to 0x17f
int i; // using a CPU register

for (i=0; i < 16; i++)
    V[i] = 0.3 + (double) i;
```

<table>
<thead>
<tr>
<th>i</th>
<th>V[i]</th>
<th>$0x_{100 + i}$</th>
<th>Miss &amp; c.c.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.3</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.3</td>
<td>0x0010</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
<td>0x0020</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.3</td>
<td>0x0030</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.3</td>
<td>0x0040</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5.3</td>
<td>0x0050</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6.3</td>
<td>0x0060</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7.3</td>
<td>0x0070</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8.3</td>
<td>0x0080</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9.3</td>
<td>0x0090</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10.3</td>
<td>0x00a0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11.3</td>
<td>0x00b0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12.3</td>
<td>0x00c0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13.3</td>
<td>0x00d0</td>
<td>miss &amp; c.c.</td>
</tr>
</tbody>
</table>

...
Example 2

• Now we change the program to use «float» single precision values:

```c
float V[32];  // using addresses 0x100 to 0x17f
int i;       // using a CPU register
```

```c
for (i=0; i < 32; i++)
    V[i] = 0.3 + (float) i;
```

• What is the behaviour of this program regarding the use of the cache (hits, misses, capacity conflicts)?

• and regarding the RAM memory?
Example 3

Consider this processor architecture running a single process with multiple software threads

SW1 running in HW1
SW2 running in HW2
Example 3

- Shared L1, line cache size 128 (0x080) bytes
- Shared L2, line cache size 256 (0x100) bytes

<table>
<thead>
<tr>
<th>Ordre d’execució</th>
<th>Core</th>
<th>HW Thread</th>
<th>Instr/dada</th>
<th>@ Memòria</th>
<th>L1 Insn</th>
<th>L1 Data</th>
<th>L2 Insn/Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>Dada</td>
<td>0x002000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>Dada</td>
<td>0x002080</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>Dada</td>
<td>0x002100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001018</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001020</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>Data</td>
<td>0x002180</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 3

- Shared L1, line cache size 128 (0x080) bytes
- Shared L2, line cache size 256 (0x100) bytes

<table>
<thead>
<tr>
<th>Ordre d'execució</th>
<th>Core</th>
<th>HW Thread</th>
<th>Instr/dada</th>
<th>@ Memòria</th>
<th>L1 Insn</th>
<th>L1 Data</th>
<th>L2 Insn/Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>Dada</td>
<td>0x002000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>Dada</td>
<td>0x002080</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>2</td>
<td>Dada</td>
<td>0x002100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001018</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001210</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>2</td>
<td>Data</td>
<td>0x002280</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 3

- Shared L1, line cache size 128 (0x080) bytes
- Shared L2, line cache size 256 (0x100) bytes

<table>
<thead>
<tr>
<th>Ordre d’execució</th>
<th>Core</th>
<th>HW Thread</th>
<th>Instr/dada</th>
<th>Memòria</th>
<th>L1 Insn</th>
<th>L1 Data</th>
<th>L2 Insn/Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001000</td>
<td>Miss 0x1000</td>
<td>--</td>
<td>Miss 0x1000</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>Dada</td>
<td>0x002000</td>
<td>--</td>
<td>Miss 0x2000</td>
<td>Miss 0x2000</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001008</td>
<td>Hit 0x1000</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>Dada</td>
<td>0x002080</td>
<td>--</td>
<td>Miss 0x2080</td>
<td>Hit 0x2000</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001010</td>
<td>Hit 0x1000</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>Dada</td>
<td>0x002100</td>
<td>--</td>
<td>Miss 0x2100</td>
<td>Miss 0x2100</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001018</td>
<td>Hit 0x1000</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>Instrucció</td>
<td>0x001020</td>
<td>Hit 0x1000</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>Data</td>
<td>0x002180</td>
<td>--</td>
<td>Miss 0x2180</td>
<td>Hit 0x2100</td>
</tr>
</tbody>
</table>
Example 3

- Shared L1, line cache size 128 (0x080) bytes
- Shared L2, line cache size 256 (0x100) bytes

<table>
<thead>
<tr>
<th>Ordre d’execució</th>
<th>Core</th>
<th>HW Thread</th>
<th>Instr/dada</th>
<th>@ Memòria</th>
<th>L1 Insn</th>
<th>L1 Data</th>
<th>L2 Insn/Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001000</td>
<td>Hit 0x1000</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>Dada</td>
<td>0x002000</td>
<td>--</td>
<td>Hit 0x2000</td>
<td>--</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001008</td>
<td>Hit 0x1000</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>Dada</td>
<td>0x002080</td>
<td>--</td>
<td>Hit 0x2000</td>
<td>--</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001010</td>
<td>Hit 0x1000</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>2</td>
<td>Dada</td>
<td>0x002100</td>
<td>--</td>
<td>Hit 0x2100</td>
<td>--</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001018</td>
<td>Hit 0x1000</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>2</td>
<td>Instrucció</td>
<td>0x001210</td>
<td>Miss 0x1200</td>
<td>--</td>
<td>Miss 0x1200</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>2</td>
<td>Data</td>
<td>0x002280</td>
<td>--</td>
<td>Miss 0x2280</td>
<td>Miss 0x2200</td>
</tr>
</tbody>
</table>
Example cache performance impact (1/6)

- ACCESS(i,j)
  - Row-wise access
    - Locality of access

- ACCESS(j,i)
  - Column-wise access
    - Matrix transposed
    - Accesses are disjoint between each other – low locality
Example cache performance impact (2/6)

- ACCESS(i,j)

\[ \text{mat}^{(5 \times 5)} \]

\[ \begin{array}{cccccc}
(i=0) & (i=1) & (i=2) & (i=3) & (i=4) \\
\end{array} \]

ACCESS(j,i)

\[ \text{mat}^{(5 \times 5)} \]

\[ \begin{array}{cccccc}
(j=0) & (j=1) & (j=2) & (j=3) & (j=4) \\
\end{array} \]
Example cache performance impact (3/6)

- ACCESS(i,j)

  ![Diagram of ACCESS(i,j)]

  - mat (5x5)
  - j
  - (i=0) (i=1) (i=2) (i=3) (i=4)
  - ~5 misses

- ACCESS(j,i)

  ![Diagram of ACCESS(j,i)]

  - mat (5x5)
  - j
  - (j=0) (j=1) (j=2) (j=3) (j=4)
  - >5 misses (capacity)
Example cache performance impact (4/6)

- ACCESS(i,j) ACCESS(j,i)

- Matrix 10240 x 10240, single precision floating point values
- MN4 processor, Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz
- Initialization times
  - loop =0: 148 ms
  - loop >0: 51ms
  - loop =0: 465 ms
  - loop >0: 361ms
- Cache friendly
  - Spatial and temporal locality
- Cache unfriendly
  ~Temporal locality, no spatial locality
Example cache performance impact (5/6)

- Cache accesses
  - loop = 0

```
callq <malloc>  // rax -> mat
loop: ...
  movups %xmm0, (%rax)
  add $0xa000, %rax
  cmp %rax, %rdx
  jne loop
```

![Diagram of cache memory hierarchy showing cache misses and stalls in instruction pipeline.](image)
Example cache performance impact (6/6)

• Cache accesses
  • loop > 0

1) miss/hit!  
2) miss/hit!  
3) miss/hit!

Core

To Main Memory

Variable execution time (examples)

1 cycle
20 cycles
80 cycles

150 cycles
To sum up...
Table of Contents

• Computer structure. Components
  • Processor – The chip
    • CPU / core
    • Multi- and many-core
    • Simultaneous multithreading / hyperthreading
  • Memory – and memory hierarchy
    • Data / instruction caches
    • Main memory
  • Input/Output components
Input/Output components

• The I/O Bus extends the access to
  • Accelerators (GPUs, FPGAs)
  • Disks
  • Network
  • Human-Machine Interface Peripherals
Bus and I/O access

• A bus is a collection of wires
  • Carry address, data, and control signals

• The CPU can communicate with I/O devices
  • Memory mapped I/O technique
    • A device is mapped to a particular mem @ (a.k.a. I/O port)

• DMA: Direct Memory Access
  • The peripheral can reads/writes directly from/to memory addresses
Access to accelerators/devices/peripherals

• Accesses to device configuration
  • Uncached – do not access the caches
    • Property of the memory mapping

• Accesses to device memory
  • Through specialized Direct Memory Access (DMA) engines
Accelerators

- Devices attached for computation
  - Need to transfer
    - code and data
    to/from the device
  - Need to start/stop execution
  - Synchronization

- Configuration through mapped memory space
  - Use specific addresses to access the device’s configuration registers
  - Access only allowed from the OS
Specialized Processing Units for Data Science

• GPU stands for Graphical Processing Unit
  • Some of them include “Tensor Cores”
    • Special hardware to accelerate computation of Tensors

• GPUs can address particular requirements of vast computing
  • Large memory bandwidth requirements
    • Take advantage of optimized PCIe bandwidth
      • Several GB/s
  • High computational power
    • Very large thread-parallelism
      • Thousands of cores

• There are other data analytics accelerators
  • E.g. FPGA, ASIC, TPU
Sata and HMI Peripherals

- Disk Drives
- Video
  - With special video memory
- USB
  - Keyboard, mouse...
Disk Drives

• Nonvolatile memory to save data
  • Retains stored values (data) even when there is no power
    • There are other nonvolatile memories in addition to disk drives (e.g. ROM)

• Similar vs Different components
  • Impact on performance and capacity
How are data physically saved in disks?

- Any storage device needs to organize the pool of memory
  - E.g.: DVD, hard-disk, pen-drive, etc.

**Capacity:** Max number of bytes recorded in a disk

\[
\text{Capacity} = \frac{\text{# bytes}}{\text{sector}} \times \frac{\text{average # sectors}}{\text{track}} \times \frac{\text{# tracks}}{\text{surface}} \times \frac{\text{# surfaces}}{\text{platter}} \times \frac{\text{# platters}}{\text{disk}}
\]

- **Sector:** The smallest unit of data that can be read/written
  - **Defined by the hardware**
  - Fixed size (typically 512 Bytes)

Some parameters that impact on performance

**Speed:** Revolution Per Min.
  - e.g. 5400 RPM-15000 RPM

**Max Bandwidth:** Bytes Per Sec.
  - e.g. few MB/s – hundreds MB/s
  - (Mbps is not equal to MBps)
Networking

• Send/receive information to
  • Servers
  • Network-attached disks
• Protocols
  • Low-level – ethernet packet
  • High-level – TCP/IP
• Control based on memory mapped configuration registers
  • Access from the OS
• Data transfers based on DMA engines
Virtual Machine (VM)

- A software package virtualizes all physical resources of a computer
  - Virtualized resources can be emulated (simulates the real behavior) or linked to access real resources of the host machine
  - Multiple VMs can run on the same host. Everyone can run a different Operating System
Bibliography

• Computer Systems – A Programmer’s perspective (3rd Edition)
  • Randal E. Bryant, David R. O’Hallaron, Person Education Limited, 2016
  • https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004062589706711
    • Several Chapters

• Computer Organization and Design (5th Edition)
  • D. Patterson, J. Hennessy, and P. Alexander
  • http://cataleg.upc.edu/record=b1431482~S1*cat
    • Several Chapters