Computer Architecture and Networks

Personal summary by Gilberto Lem

Contents

1 Performance Evaluation of Computer Systems

Performance will mostly refer to speed. Speed can be seen from two views:

- **User's view:** A is faster than B if a program takes less time on A
- **Computing centre's view:** A is faster than B if A has more throughput (number of completed tasks within a timeframe)

In general, one can measure the time that a given program takes with the `time` command in UNIX. It will return 3 values:

- **Execution Time:** Total time required for a task, including memory access, disk access and I/O
- **User CPU Time:** Time spent in processor running the program's code
- **System CPU Time:** Time spent running code in the operating system kernel on behalf of the orgram

To evaluate performance, we can use:

- Measurements during execution
- Hardware Parameters
- Runtime measurements
- Theoretical/Analytical/Simulated models

We will elaborate on the last three:

1.1 Hardware Parameters

In theory, we could get the CPU-Time of a given problem with the following model:

$$CPU\,time = IC * CPI/CF$$

where

IC: Instruction Count. Number of Instructions in the program.
Depends on the ISA (chapter 3), the compiler technology, and the program-specific requirements.

CPI: Average number of clock cycles per Instruction.
Depends on the Microarchitecture (chapter 3), ISA and the program.

CF: Clock frequency (number of cycles per second)
Depends on semiconductor technology and Microarchitecture.

Other relevant parameters that are implicit in the previous model are the **MIPS** (Millions of instructions per second) and the **FLOPS** (Floating Point Operations per Second).

1.2 Runtime Measurements of Existing Programs

Benchmarks are already-made-codes that can be used to measure execution time. There exists a great variety of benchmarks for a great variety of necessities, from finite element simulations to number theory. You find them as source code and have to compile them, so it is important to take into account the quality of the compiler and software you are using.

1.3 Theoretical/Analytical/Simulated models

One of the most simple models to evaluate computer systems, specifically the impact that the improvement of a certain component has on the overall performance of the computer, is **Amdahl's Law**.

It assumes that some program parts are accelerated by a factor of N, and that these parts represent a factor of f of total runtime.

$$Acceleration = \frac{1}{1 - f + \frac{f}{N}}$$

2 Computer Architecture Basics

Computer Architecture describes how a computer system is organized and implemented. How you organize your hardware components, and what are the conventions they follow to function. It comprises the Instruction Set Architecture (ISA), the Microarchitecture, the Logic Design and the Implementation.

2.1 Computing

Transistors are electronic devices that can be used to control an electrical "switch" with a very tiny current, i.e. with a tiny current you can turn on or off a bigger current that goes through it. If you arrange transistors in a certain way, you can manage to create logic gates (NOT, NAND, NOR). A machine could be made of complex combinations of those logic gates, using them to execute binary operations and with that follow instructions.

Binary language is the most fundamental (and only) way of communication that we can have with this kind of machine. All instructions that we can give the machine have to be in binary, and it is called **Machine code.**

Assembly code is the easy to read interpretation of machine code. There is a one to one matching, one line of assembly code equals one line of machine code. For example:

Machine code	**Assembly code**
000000110101 =	Store 53

For representing big amounts of binary numbers, sometimes it is convenient for the user to use **Hexadecimal digits**, as 1 hexadecimal digit represents 4 binary digits, which makes it more compact and a little easier to read.

2.2 Storing

Transistors are powerful tools for computing, but they cannot store information. However, you can arrange a circuit of logic gates in a way that can store 1 bit. This is achieved through feedback loops. One can connect some of the inputs of the circuit to some of its outputs, and with that the system changes its stable state depending on the independent input and the previous state. This is called a **Flip-Flop**, and by arranging millions of them could manage to make a **memory**.

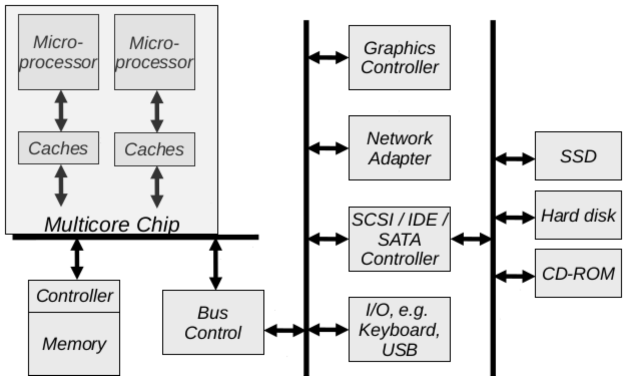
However not all memory is built with Flip-Flops. Sometimes other technologies as capacitors are used.

2.3 Von Neumann Architecture:

Computing and storing are the most basic functions one would desire in a computer, the only thing that is left is the channel of communication with the user. This is called **Input/Output unit (I/O)**. The Von Neumann Architecture puts this together to make a model that represents the most basic model of a computer having 4 main units. In practice, the first two are always implemented in the same unit, called the Central Processing Unit (CPU or microprocessor [1]).

- **Arithmetic Unit:** Performs operations on binary numbers.
- **Control Unit:** The computer has to have a way of monitoring the instruction that it is executing. The Control Unit contains an instruction register[2] (stores the instruction being executed) and program counter (holds the location in memory of the current instruction).
- **Main Memory:** Stores data and instructions.
- **Input/Output unit**

This architecture however does not represent a full computer. The following is a more complete example:



[1]Microprocessors can be used for any application, and can be optimized for each one. They are everywhere e.g. PC's, cars, printers, home entertainment systems, cellphones, etc.

[2]Register Files consist of internal memory within the microprocessor. It is very limited in storage but very fast. Its width is often equal to the width of processor.

3 Instruction Set Architecture (ISA)

If we want to communicate with the computer, we have to stablish conventions for what certain binary code means. The ISA consists on the set of instructions that the processor can understand. It is basically a contract the processor uses to communicate with software. It includes more than just instructions, it includes the "language" of the processor, including grammar, and:

- Supported data formats and their storage
- How to access operands (Execution Models)
- The instructions per se, which contains:
  - Privileged facilities in the microprocessor (Protection Mechanisms)
  - Encoding instructions

On the other side, **microarchitecture** is the way the ISA is implemented in the processor.

3.1 Supported Data Formats and their Storage

The ISA has to contain a way to discern between different types of data, as integers, characters, unsigned integers, etc, and thus, each data type is represented by a different **Data Format**. The formats (in binary form) have concrete restrictions on size, which depends on the architecture width (currently most used 64 bits, i.e. each register for addresses in memory has 64 bits).

For example, in the IA-32, an ISA for intel microprocessors, signed integers can be of 1 Byte, 2 Bytes (word), 4 Bytes (double word), 8 Bytes (Quadword), and the first bit is always the sign bit.

So far, only integer types are supported natively by the microprocessor. Other types, as floating point numbers, need to be emulated in software. However, more modern architectures contain special support. They can have a specific unit for floating point, or they can have extra instructions.

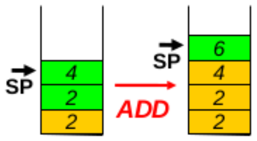
3.2 Execution Models

The ISA has operations defined. Each operation requires 1 or 2 operands, and has a result. The Execution Model defines the place these operands and result are located, the way they are accessed, as well as the types and quantity of operands that can be processed. Here are some examples:

Stack Model In the Stack Model all the operands are on the Stack (see box below), and the result is stored again on the Stack. A problem here is that Stack is a part of Memory, and Memory is slow (it is far away from the processor and one need to look for the desired data with the address). This is the Execution Model of Java Virtual Machine (JVM). This model is mostly used for two objectives: storing temporary values and storing the program counter (pointer to the current instruction) when calling functions.

Example Operation: C = A+B

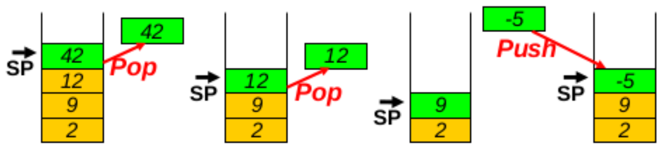
Assembly: Pop A, Pop B, Add C,A,B, Push B, Push A, Push C.



The Stack is a data structure with LIFO principle (the Last In is the First in getting Out). It has two instructions:

- Push: gets a value into the Stack
- Pop: gets the last value from the Stack

A Stack structure is present in basically all architectures, for both data and addresses, as a part of the Main Memory (RAM). There is also a special purpose register, called the Stack Pointer (SP), which points to the current last value on the Stack.



Accumulator Model This consists on using always a special register that is made for arithmetic results. It works for single source operands, and has a limited flexibility.

Example Operation: C = A+B

Assembly: Load A, Add B, Store C.

This would store A in the Accumulator, then add B and store the result in the accumulator and then store C.

Register-Register Model Also called Load/Store Architecture, it allows 3 Operands, all of them from Registers. Its instruction format is simple, but it has a high instruction count.

Example Operation: C = A+B

Assembly: Load R1,A / Load R2,B / Add R3,R1,R2 / Store C,R3

Memory-Memory Model Here all operands are in memory. It allows 3 operands per instruction, and you do not need to bother about registers. However, memory is slow, and the instruction formats are irregular.

Example Operation: C = A+B
Assembly: Add C,A,B

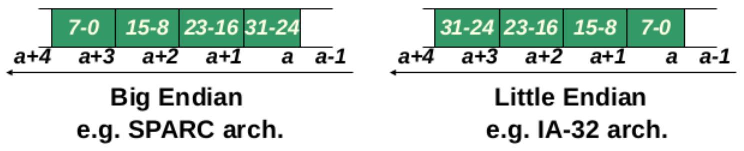
Memory-Register Model It has one operand in main memory and one in registers. It takes two operands and the first one is overwritten. A disadvantage is that one operand is destroyed.
Example Operation: C = A+B
Assembly: Load R1,A / Add R1, B / Store C, R1

3.2.1 Addressing of Memory

All execution models require transfer from/to main memory. This main memory is organized as equal cells (Linear addressing). There are several addressing schemes (i.e. the way the information is requested in the instruction):

- **Direct Operand:** When there is a constant operand, so there is no requirement of addressing scheme, *e.g. Add #5 (sums the number 5).*
- **Register Addressing:** The operand is in Register, and you use the register number in the instruction, *e.g. Add R3.*
- **Direct Addressing:** Operand is in Memory and you use an address in the instruction, to point to it, *e.g. Add 0x0F13.*
- **Register Indirect Addressing:** The operand is in Memory. The address of the operand is in the register. You use the register number in the instruction, *e.g. Add *R3.*
- **Register Indirect Addressing with Displacement:** The same as before but in the instruction you add also a displacement to change the address, *e.g. Add (*R3+d)*

Endianness: Refers to the way of storing data which is > 1 byte in memory. The address is based on individual bytes, thus you use several addresses to store one word. Memory addresses grow to the left. For one 4 Byte value, you can store your bits in the same direction or backwards. The same direction is called Little Endian, and the opposite is called Big Endian.



3.3 Types of Instructions

- **Arithmetical and Logical Functions:** Arithmetic (`+,-,*,/,not,++,--`), Logical `(&,|,xor)` and Comparisons `(gt, lt, gte, lte, eq, neq).`

Flags
There exist some special purpose registers that store useful values for computations, they are called Flags. In them you store some additional result values, e.g. overflow, result of comparisons. These are examples of the most important flags:

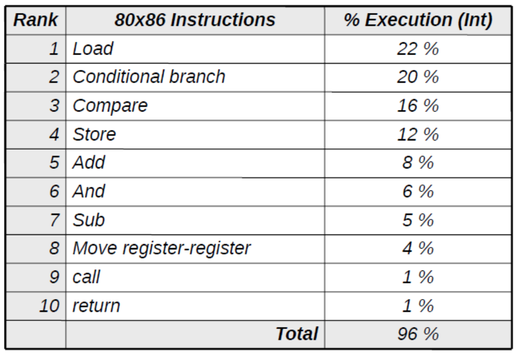
- ZERO: Last result was 0
- NEG: Last operation was negative
- CARRY: Last bit for adder
- OVERFLOW: Last operation caused overflow

- **Data Transfer Instructions:** They move data to and from storage: `Move, Load, Store, Push, Pop`
- **Control Flow Instructions:** The processor works on an instruction stream, however we can get away from this stream: `Jump, Call, Return, Conditional commands` [3].
- **I/O operations (in some architectures)**
- **System Instructions:** They include system management instructions[4], e.g. setup of initial execution or access to internal parameters. Specific example: Loading of stack pointer.

In execution, the instruction follow this distribution in time:

[3]Conditional commands extract values from flag registers. For example, evaluating `a>b` would compute `a-b` and then use the value stored in the NEG flag.

[4]Not all users/processes should have access to system parameters. For this there exist **protection mechanisms**. There are two (or more) modes of execution of instructions: *User mode* has unprivileged instructions, and *Kernel mode* allows all instructions. The Operating System is executed in Kernel mode.



Rank	80x86 Instructions	% Execution (Int)
1	Load	22 %
2	Conditional branch	20 %
3	Compare	16 %
4	Store	12 %
5	Add	8 %
6	And	6 %
7	Sub	5 %
8	Move register-register	4 %
9	call	1 %
10	return	1 %
	Total	96 %

Based on a subset of the ***SPECint*** *codes (taken from Hen. & Pat.)*

3.4 Types of ISA

Depending on the complexity of the instructions defined, the ISA can be of two types:

CISC: Complex Instruction Set Computer

- Many instructions and many addressing schemes.
- Instructions varying in size
- Example Architectures: IA-32, IA-64, x86-64

RISC: Reduced Instruction Set Computer

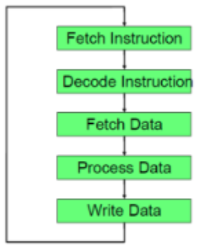
- Few fast instructions
- Fixed format (length)
- Many general purpose registers
- Example Architectures: ARM, SPARC, Alpha

4 Microprocessor Basics and Techniques

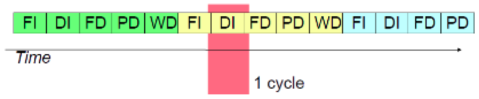
4.1 Pipelining

The basic task of the processor is to receive a stream of instructions, maybe modify the stream (e.g. loops), and execute each instruction.

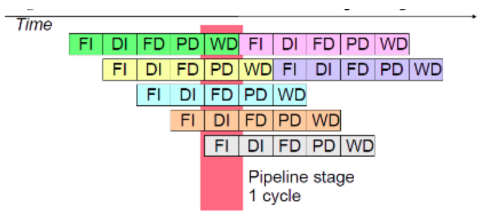
This process can be summarized in the following cycle[5]:



Each of the steps is carried out by a different component (unit) in the microprocessor. During one cycle only one of the units is activated.



This leads to a waste of resources of all the other units. We could overcome this by making use of the other units by executing other instructions. This is called "Pipelining"



It is notable that the more units you have (the longer your pipeline is) the more parallel processes you can have at the same time and the more efficient the system is. Thus an approach to increase productivity is to add more stages. There is a trend to make longer pipelines with more stages and increased clocks, however this increases potential conflicts.

It is also worth noting that the pipeline rate is limited by the slowest pipeline stage, and thus unbalanced length reduces speedup.

Here are some example pipeline lengths:

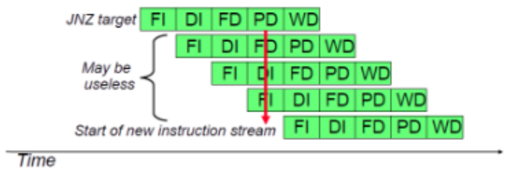
[5]This is a symplified cycle. A more complete cycle could include interrupts and exceptions

Processor	**Length (units)**
Intel Pentium II	12/14
Intel Pentium IV	22/24, 33
Sun Ultra II	6/9
Sun Ultra III	14/15
Modern x86 processors	~15

In real life not all phases are of equal length, and some instructions need additional phases. Pipelining is very well suited for RISC systems, although it is also used in CISC systems.

Furthermore, we are assuming that each instruction is independent and that units can operate concurrently. In reality different kind of conflicts can occur:

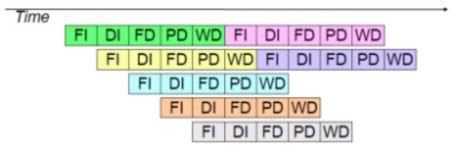
- **Resource Conflicts:** Two instructions may need the same physical resources at the same time
- **Data Conflicts:** One instruction may need data from a previous one. These could produce wrong results, since some operations could execute taking old values stored in the register.
  Solution with software: When the compiler detects them, or using appropriate code structuring.
  Solution with hardware: When it detects dependencies dynamically. The hardware could stop starting new instructions until the dependency is fulfilled (Pipeline Stall). Also, the hardware could forward results before the "Write data" phase, to make the process faster.
- **Control Conflicts:** Take for example with conditional branches. The result of the condition is not known until after the "Process Data" phase. When you get to this phase, many other instructions may have been started and partially executed. After the phase, the pipeline must be restarted and the results of the other instructions must be dropped.
  Solution with software: **Branch Delay Slots** fill this gap with instructions that will be executed in any case (this avoids pipeline stalls!)
  Solution with hardware: Restarting the pipeline, adding mechanisms to evaluate branch targets early, and with branch prediction



.

4.2 Superscalarity

In pure pipelining every unit exists once in a processor:

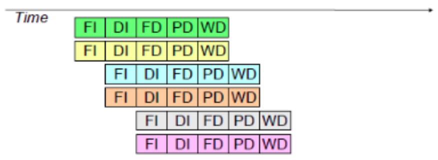


Superscalar Execution consists on replicating functional units, which can create several pipelines, start several instructions concurrently and with that speed up the code.

In practice this is not normally implemented as N pipelines. It has a flexible assignment of free units in each stage.

Naturally this has similar problems as pipelining, with data dependencies and control flows. Furthermore, it has some other problems, as resource conflicts when accessing functional units, assignments and reservation of units, and the need for complex hardware.

Some examples of this are the Intel Pentium II, the Intel Sandy Bridge and the AMD Bulldozer.

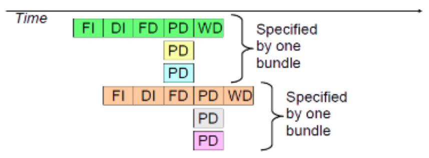


4.3 VLIW

Very Long Instruction Word (VLIW) is another architecture that implements parallelism, however unlike superscalarity it does not paralelize all the pipeline stages, but only the "Process Data" stage. All other phases work as usual. It makes on-chip parallelism explicit, specifying the task of each component of the processor at each instruction.

This allows the process of executing the instruction to be purely controlled by software (the compiler does the work before the execution), which reduces hardware complexity.

It can be combined with pipelining:

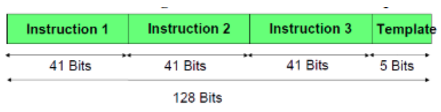


The more "famous" VLIW examples are the Transmeta Crusoe (x86 emulation) and the Intel Itanium / IA-64.

4.4 Intel Itanium

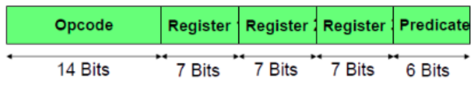
The **IA-64 Architecture** was the first 64 Bit processor of Intel. It had a 64 bit width, VLIW design, Predicated execution and Speculation. They called it EPIC (Explicitly Parallel Instruction Computing).

Each instruction was formed with a set of single instructions and a template.



Where each single instruction corresponds to one element of the processor, and the Template indicates if the instructions in a bundle can be executed simultaneously, or if one or more instructions are to be executed sequentially, or if neighboring bundles can be executed in parallel.

Each one of the single instructions have the following structure:



We have space for the operator, and 3 register operands (their register number) and a predicate.

4.4.1 Predicates

Most programs are written sequentially, and the compiler must parallelize this sequence. A common problem that emerges in this approach is the conditional branches. The path to take is unknown at compile time, and thus the processor must decide during runtime which instructions are to be fetched into pipeline. One way too overcome this is branch prediction, where the processor starts executing one of the branches before the result of the condition is known. However if a branch prediction goes wrong, instructions must be thrown away. This is worse if the pipeline is large.

Predicates are a way to solve this problem, getting rid of branches. Predicates are auxiliar registers that contain information about previous comparisons. Each comparison sets two separate predicate registers. The first predicate gets the result and the second gets the complement.
Take for example the following Itanium code:

```
comp.eq p1, p2 = r1, r2
```

If the value in `r1` and the value in `r2` are equal, then `p1=1`, `p2=0`, otherwise `p1=0`, `p2=1`.

Example

- If-Then-Else in C:
```
if (emp_status == ACTIVE) {
active_emps++;
total_payroll += emp_pay;
} else{
inactive_emps++;
}
```
- If-Then-Else in Itanium Assembler:
```
{
Cmp.ne p1 = rs, ACTIVE
(p1) br else
}
.label then
{
add rt = rt, rp
add ra = ra, 1
br join
}
.label else
{
Add ri = ri, 1
}
.label join
```
- With predication:
```
{
Cmp.eq p1, p2 = rs, ACTIVE
} {
(p1) add rt = rt, rp
(p1) add ra = ra, 1
(p2) add ri = ri, 1
}
```

4.4.2 Unconditional Compare

Predicates are also used for nested IF constructs, this is achieved through the **unconditional mode** of predication. Take for example the following instruction:

```
(p1) cmp.eq.unc p2, p3 = r1, r2
```

This is a comparison that begins with a predicate, and has the termination `.unc`, which represents the unconditional mode. The convention is the following: if (p1) == 1, it will execute as normal, but if (p1) ==0, it will set both predicate targets to 0.

Example

- Nested IF in C:
  ```
  if (a>b) { // block 1
  c++; // block 2
  } else {
  d+= c; // block 3
  if (e==f) { // block 3
  g++; // block 4
  } else {
  h-- // block 5
  }
  }
  ```
- With unconditional compare it is way more compact:
  ```
  {
  cmp.gt p1, p2 = ra, rb // block 1
  } {
  (p1) add rc = rc, 1 // block 2
  (p2) add rd = rd, rc // block 3, If a>b, (p3,p4)->0, because of
  the .unc
  (p2) cmp.eq.unc p3, p4 = re, rf // block 3
  } {
  (p3) add rg = rg, 1 // block 4 (This will not be executed)
  (p4) add rh = rh, -1 // block 5
  }
  ```

4.4.3 Paralell Comparisons

When you have several comparisons that are operated logically, the traditional approach is to treat the operations sequentially, however, you can parallelize the instructions. Logical operations here have the following behavior: they receive a condition to evaluate (compare relation) and target predicate registers:

- **AND** sets both target predicate registers to 0 if the compare relation is false. If the compare relation is true, it does nothing.
- **OR** sets both target predicate registers to 1 if the compare relation is true, and otherwise does nothing.

Example

- Parallel Compare in C:
  ```
  if (status == FOUND && record == search_key && no_error) {
  ...  // do something
  }
  ```
- Parallel Compare in Itanium:
  ```
  {
  cmp.eq.or p1, p2 = r0, r0 // init:  p1 = p2 = 1
  }
  {
  cmp.eq.and p1, p2 = rs, FOUND // status==FOUND
  cmp.eq.and p1, p2 = rr, rk //record==search_key
  cmp.ne.and p1, p2 = rn ,0 // no_error != 0
  }
  ```

As all the three conditions are AND's, then it does not matter the order they are computed, they all do the same, either remain with the same result, or set the result to 0. Only comparisons of the same type can be paralellized with this method.

4.4.4 Other Features of IA-64

Control Speculation (Speculative Loads): Move load instructions up to prevent memory latency. If speculation fails recovery code is needed (generated by the compiler).

Data Speculation: Try to guess about values/conditions to fasten-up code. If speculation fails recovery code is needed.

Register Rotation: It renames the registers during execution to prevent dependencies where there are no dependencies. For example:

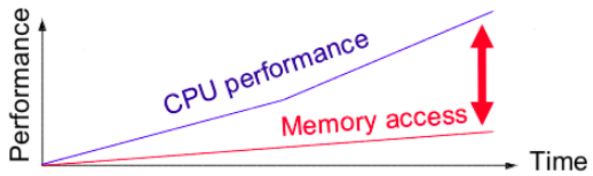
1. LOAD R1, 1024

2. ADD R1, R1, 2
3. STORE R1, 1024
4. LOAD R1, 2048
5. ADD R1, R1, 4
6. STORE R1, 2048

This produces a dependency between lines 4-5-6 and lines 1-2-3, however, this could be avoided and thus parallelized if in lines 4-5-6 we used R2 instead of R1.

5 Caches

The CPU-Memory Bus is the connection between main memory and CPU. All data and instructions need to go through it. The processor has to wait until data is present before a command can be executed. This memory speed increases 7-15% per year with the development of technology, however, the CPU performance increases 50-100% per year. This limitation by memory is called the "Memory Wall".



One of the reasons of this phenomenon is that memory is too far away physically from the processor.

Caches are memories closer to the CPU which hold subsets of the main memory. They have lower latency (time to transfer the information), higher bandwidth (amount of information transferred per unit of time) and are close to the CPU (on the same chip). Data can be loaded in the cache and then be reused in later accesses, so it can save time of repetitive slow accesses to main memory.

Caches use different memory technology than main memory. Here are the most important differences between them:

Caches	**Main Memory**
SRAM	DRAM
FlipFlops	Capacitors
	Have to be recharged periodically
Fast and Expensive	Slow, cheap and very small

Here is some of the basic cache terminology:

- **Cache hit:** Data is requested and it is already in the cache
- **Cache miss:** Data is requested and it is not in the cache, thus it has to be retrieved from memory
- **Cache size:** Size of cache in Bytes
- **Cache line size/length:** Caches do not store individual bytes, they store blocks of bytes. The length is how many bytes has each element of the cache. Cache lines are equally sized. Memory cells are bundled into blocks equivalently. The typical size of lines are 32 or 64 bytes.
- **Replacement policy:** Defines which cache line to evict if we have no more space.

- **Write back caching:** In this modality, all the writes are stored in cache, and if they are evicted they are written back to main memory.
- **Write through caching:** Data is written directly to main memory.
- **Cache Layers:** There could be several independent levels of caches, where higher levels are faster but smaller.
  CPU - L1 Cache[6] *- L2 Cache - L3 Cache - Main Memory*
  For my computer: L1 32kB, l2 256 kB, L3 6MB.
- **Full Inclusion:** Means that all the contents of smaller caches are included in bigger caches. For example all the contents of L1 being included in L2.
- **Additional bits:** Each cache line has special purpose bits with additional information:
  - Valid bit: The line contain a copy of a memory block?
  - Tag: Which one?
  - Dirty bit: Has there been a write access to this line? (used for parallelization)

5.1 Cache Associativity

For the cache to work we have to map and monitor which blocks of main memory are in which lines of the cache. There are several schemes for this. Suppose we have n blocks in memory and m blocks (lines) in the cache; and we use j to denote the index of the memory block and i to denote the index of the cache line.

Direct-Mapped Cache: In this mapping each block in memory can be mapped to only one line in cache. One easy way to map this is taking the module `i = j mod m`.

This implies low hardware complexity and a fixed (but not optimal) replacement strategy.

k-way Set Associative Cache: Here the cache is divided into v sets, where each one can hold k lines. One block in memory would be mapped to only one set, however, there is freedom within the set to occupy k different places. Modern architectures use generally sets with 8 or 12 cache lines.

Full Associative Cache: Here any block in main memory can be mapped to any cache line, and has to specify a replacement strategy. It would be similar to having a set of m lines.

This implies high hardware complexity.

[6] *L1 Caches are often split: L1-i for instructions and L1-d for data. This reduces conflicts originated on the different access patterns for instructions/data, and also allows optimisations by locating the to L1 caches closer to their relevant functional units (decoder for L1-i, and load/store units for L1-d).*