MASTER DEGREE in EMBEDDED COMPUTING SYSTEMS
A.Y. 2016 - 2017

Computer Architecture - Notes

Computer Architecture - Notes

Lesson 01 - 01/03/2017

- Professor: Antonio Prete
- Email: a.prete@ing.unipi.it

Books

- Computer Architecture, Fifth Edition: A Quantitative Approach, John L. Hennessy, David A. Patterson
- Advanced Computer Architecture and Computing, S.S. Jadhav
- Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors, Jean-Loup Baer
- Parallel computer organization and design, M. Dubois, M. Annavaram, P. Stenstrom

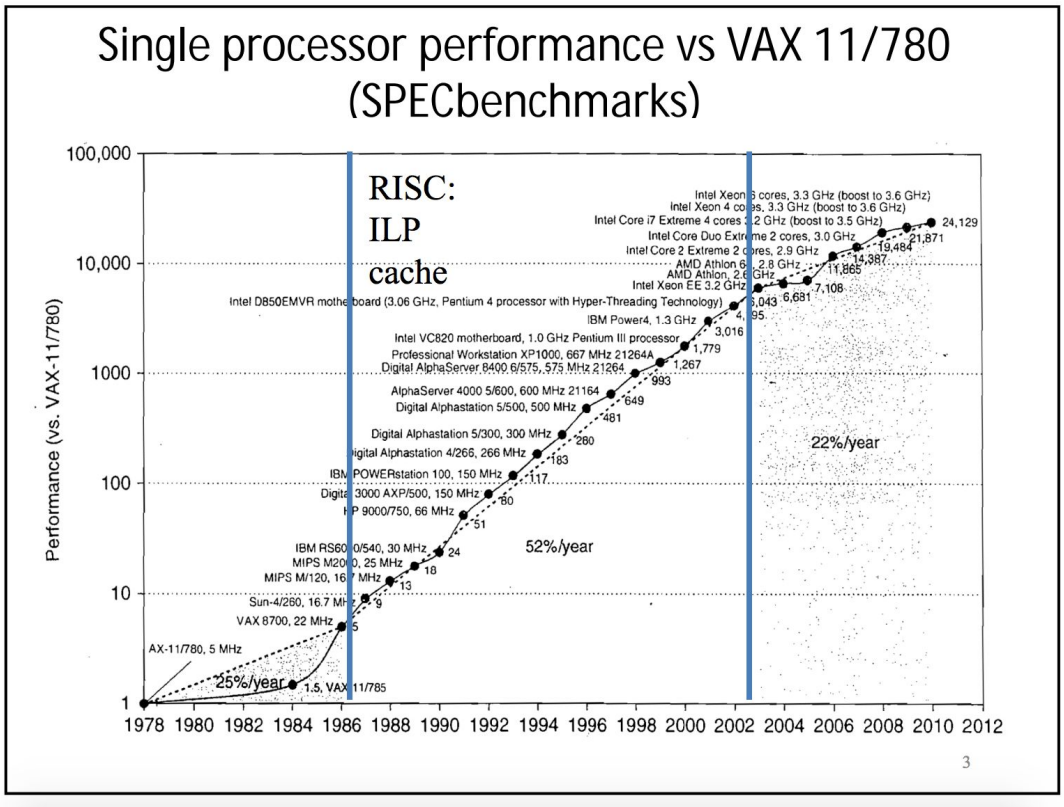
What the course is about

- Define classes of computers, field and feature that a computer must have. Used to understand the features.
- Technology trends (important) any change inside the technology can change our computer system.
- Understand main resources (memory subsystem) 3 levels: from single instruction to a set of multiple instructions at the same time.
- And so on..

The presentations will cover a single architecture, a list is available in the slides, but many other arguments can be covered of course. The logical organization of the presentation must be designed in order to reach the goal of “knowledge”.

Introduction

Let’s start talking about single processor performance of microprocessor. To evaluate difference of performance we take as starting point the **Vax 11/780** processor. The VAX is an ideal computer with performance “1”. A computer with performance 2 will require half the time that would need with a VAX computer.

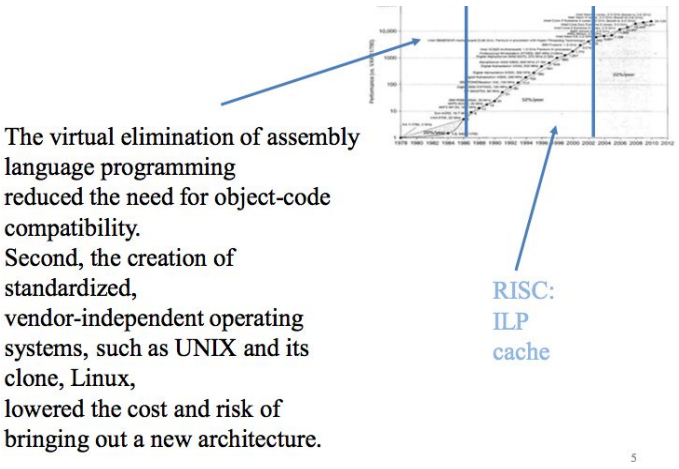


The slide can be divided in three parts.

In the first part (the one on the left) the ability of processors constructor was to increase rate of clock acting on the integrated electronics. In fact all the resources needed by a single processor have been moved inside a single chip (everything else outside, like memory etc.) and this way of working can increase clock performance, because the delay reduce needed to reach these components from the cpu is reduced.

In the middle part of the slide the improvement of processor was focused on RISC approach. Constructors wanted to reduce the gap between machine level instructions and low level instructions. But this is not a good idea for the programmers. So the idea was to design microprocessor using only the most used instruction. That's why we have a RISC: the typical program structure can be used again and then mapped in a new architecture, using only the most used instructions by a programmer. Another task was to parallelize the instruction (**ILP**: **I**nstruction **l**evel **p**arallelism) and use the cache to limit the memory and bus use. But the cache don't improve rate of processor, it's only an organization thing. Cache is useless for processing. But it's able to obtain better performances. The cache is the natural consequence of the way programmers usually solve big problems. Since it is not possible to solve big problems using a little program, usually programmers divide these bigger programs in smaller modules that are accessed frequently over time. These modules can be stored in cache to reduce latency introduced by memory access. This is a good idea only because the programmer works this way, the design of the machine follows the programmer way of working.

In the third part of slide constructor are focused on multicore/multithread processing. The idea is to use more core on same instruction/memory to improve rate. Sharing memory space between more than one process using more than one core, and at the same time more threads of the same process.



We can see that the growth in the three part of image it's not the same and nowadays the important task it's not to improve rate of clock but only to save energy.

Classes of computers

Now we can consider the classes of computer or better "the different way to use computers".

- Personal Mobile Device (PMD), start phones, tablet computers
- Desktop Computing
- Servers
- Clusters / Warehouse Scale Computers, "Software as a Service (SaaS)"
- Embedded Computers (industrial or consumer application)

It's important to identify the main features of each class in order to develop in the right direction.

Personal mobile device

Wireless devices with multimedia user interfaces such as cell phones, tablet computers.

Main features:

- The price for the consumer complete product is a few hundred dollars.
- Power consumption has to be limited because of:
    - use of batteries
    - use of plastic packaging
    - absence of a fan cooling.
- Energy and size requirements lead to use of Flash memory for storage.

Typical application:

- Applications are often Web-based and media-oriented,
- Responsiveness and predictability are key characteristics for media applications.
- Key characteristics in many PMD applications are:
    - the need of an efficient use of energy (problems in heat dissipation) and
    - the need to minimize memory (is a big portion of system cost), in particular the code size.

Desktop Computing

Desktop Computing spans from low-end netbooks to high-end workstations.

- The trend of desktop market is to be driven by price-performance optimization
- The increasing use of Web-centric, interactive applications launches new challenges to performance evaluation

They are very similar to smartphone requirements but here we don't have problem of space nor resource. The problem here is the price/quality ratio.

Servers

Servers have become the backbone of large scale enterprise computing, replacing the traditional mainframe. They are not web-oriented, servers required some characteristics:

- Availability, Servers must operate seven days a week, 24 hours a day.
- Scalability, the ability to scale up the computing capacity, the memory, the storage, and the bandwidth is crucial
- Efficient throughput, the overall performance, in terms of transactions per minute or Web pages served per second, is crucial

Clusters/Warehouse-Scale Computers

Clusters are collection of desktop computers or servers connected by local area networks acting as a single larger computer.

- Warehouse-scale computers (WSCs) are designed so- that tens of thousands of servers can act as one.
- Each node runs its own operating system, and nodes communicate using a networking protocol.

Price-performance and power are critical:

- 80% of the cost of a $90M warehouse is associated to power and cooling of the computers inside
- The computers and networking must be replaced every few years.

Supercomputers are different because they emphasize floating-point performance and by running large, communication-intensive batch programs that can run for weeks at a time.

Thus in this situation it's important the take in advance strategy to reduce power consumption, temperature or reuse part of some energy utilization. Inside the microprocessor we need some mechanism to manage this things.

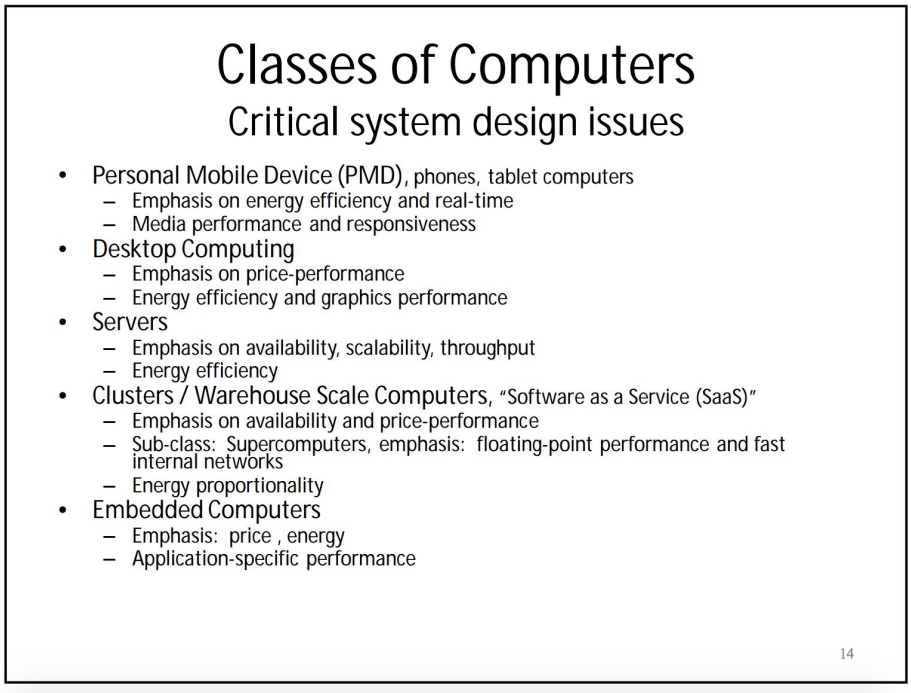
Embedded Computers

Embedded computers are present in everyday used machines: microwaves, washing machines, most of printers, most of networking switches, and all cars contain simple embedded microprocessors. The ability to run third-party software is the line between no-embedded and embedded computers.
It's not easy to define an embedded/no embedded computers but we can say the main important embedded computer is the microcontroller.

- Embedded computers have a range of processing power and of cost very large:
  - 8-bit and 16-bit processors (> a dime),
  - 32-bit microprocessors (>$5),
  - high-end processors for network switches ($100).
- The main goal is to reach the required performance at the minimum price.
- Microcontroller, a chip with off-the-shelf microprocessors and other special-purpose hardware.

Critical system design issues

So in the end this slide can summarize the main critical system design issues of each class.



It's fundamental to implement systems with an high level of scalability. For example think about a bank informatic system. The same mechanism has to work in a little bank of town and in the main location. So the scalability is the ability of the system to adapt to growing of the system without changing the architecture. The solution has to be designed and specialized to a particular situation without changing software, just changing/adding hardware resources. This is uneasy, because there are many problems like for example

bottlenecks, because they reduce global performance, reducing scalability range for an application. In a desktop computer the bottleneck for scalability is the BUS (the shared memory mechanism!). Adding memory and CPU's it's not so much to improve computer performance. The way to resolve the problem a possible solution is to increase the cache memory and implement a different way to access to memory. The cache has a smaller access delay with respect of the real memory one. In a network of computers a better software design strategy may reduce the traffic on a shared link, increasing global performances, for example including some redundancy of informations between the peers.

The main way to resolve some problems that we have just seen is the parallelism in application. But it's not easy to organize a parallel application. First let's define parallel and distributed systems.

Parallelism

Parallelism in application

The first step is to know the difference between distributed and parallel systems. A **parallel** solution is usually needed because the single processor cannot provide enough computational power to realize an application. A **distributed** solution is needed because the problem itself is distributed (in different places for example). A distributed system thus is a feature of the application because the application can be distributed in some different physical places. A parallel system instead it's not a feature of the application!

Parallelism in application can be of two different types:

- **Data-Level Parallelism** (DLP) is due to many data that can be processed at the same time.
- **Task-Level Parallelism** (TLP) is due to tasks of work created to operate independently and largely in parallel.

An example of data-level parallelism is a video encoding process! A task-level parallelism example can be a service-offer mechanism such as a web server (in general some internet services), or a DBMS.

In the first case, some examples can be video processing, graphics, in general many data that need to be processed in the same way.

For the second one, typically a general service (like a web server, etc.) is based on this: each request leads to another process. In general, Internet services are based on this idea and also databases.

Parallel Architectures

Parallel architectures helps the system to reduce cost of management, power and time. The way to solve, to offer process power to this parallelism are:

- Instruction-Level Parallelism (ILP): pipeline and speculative execution

- Vector architectures/Graphic Processor Units (GPUs): by applying a single instruction to a collection of data in parallel
- Thread-Level Parallelism: management of parallel threads
- Request-Level Parallelism: management of tasks

ILP: based in general on the pipeline, but also on speculative execution. Executing instructions separately, the result have to be stored in a common space, called commit space.

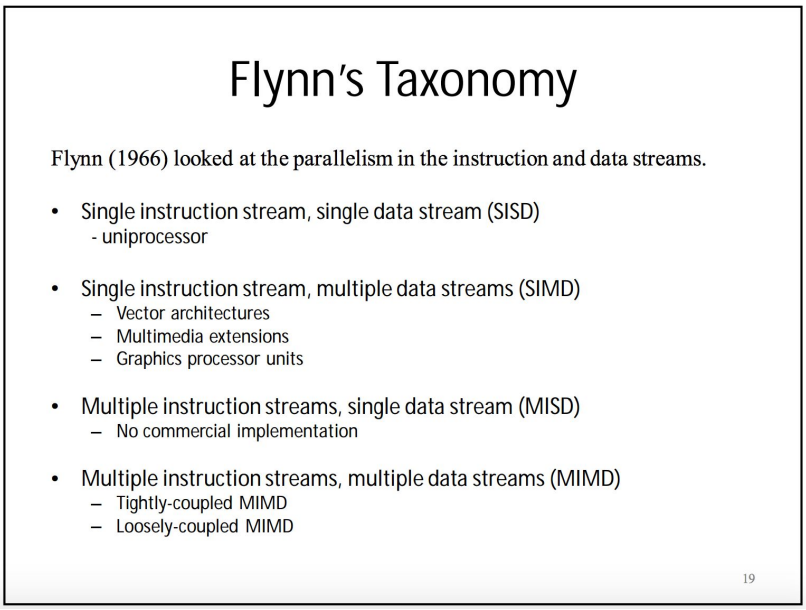
Vector and GPUs: Same code on a set of data in parallel.

Thread-level: many flows of execution inside a single process. This reduces cost of managing processes using threads instead, because they share the code and the memory. Typically you can find this solution inside a server to obtain better performances. Thread are useful when code it's the same for more threads. But it's not easy to design an application with thread structure.

RLP: the number of requests can drive the parallelism of the machine such as in web servers, when requests grow to much, we can actuate a mechanism to reduce or kill some requests/task. In general the number of processes/threads changes depending on the number of requests. This can be done to reduce power consumption or to reduce the overhead introduced by a large number of tasks.

Flynn taxonomy

In 1966 Flynn derive a classification for Task and Data parallelism:



This classification is very old but we can use to create an idea of device that are developed in this way.
The first item (SISD) architecture nowadays remain only for simple microcontroller of embedded system. The second (SIMD) is the most widely used. The last (MIMD) in general is a parallel solution distributed.

Main problem of Pipeline: to be efficient, the pipeline must be full for most of the time, so jumps are a big problem. Solutions may be speculative execution or branch prediction tables.

In this case is important to consider two main things.

- The first, is needed to have all the process in execution at the same time, hence is important to split the main application task in some little tasks. And each task have a processor. Processors cannot be free (**balancement**).
    - Traditionally, in operating systems this is obtained using a scheduler, but this strategy can be obtained only if we have a shared memory on which each processor can read and write. The scalability for this solution is very complex because shared memory/bus are bottlenecks!
    - In clusters of server in order to obtain balancement I need to use the network (has a cost).
- The second problem is that to obtain the highest performance it's fundamental to reduce the cost in communication. The communication is needed but it's an overhead.

It is possible to move an application from a system to another if the application is able to balance the workload (balancing plus communication overhead and cost).

Lesson 02 - 02/03/2017

Selecting the best architecture

Cpu's are strictly designed for the type of devices that they have to serve. This is strictly true for embedded computing systems, in which CPU are design not only to execute instruction but with some adding hardware facilities.

What can be the way to design the computer that fit our needs? One way can be try our application in some different systems and choose the one that had the shortest time response. But this way of reasoning cannot be followable for example if we have to design the application or if we don't have the possibility to test our application.

That's where **benchmarks** come into the scene. Benchmarks aren't only charts of execution time, they include multiple factors and in general anything that we consider to be relevant in terms of performance. Benchmarks are already calculated by scientist and IT people and we can use benchmark to have an idea of the response of some system. Another factor can be the cost of a system. The goal can be minimize the money spent for the service or the energy used in time.

The execution time is not in general a goal. For example in the case of a server the goal is the throughput (requests completed over the cost of power consumption). Different applications designed for the same class may have different goals. For the same system we could have multiple goals (one for the customer and one for the company). You must define performances for the global system considering all the users, because they will have different needs.

This means we have to predefine requirements that our systems need and only after starting to design phase. Killer feature for class of device can be:

- Personal Mobile Device (PMD), phones, tablet computers
  - Emphasis on energy efficiency and real-time
  - Media performance and responsiveness
- Desktop Computing
  - Emphasis on price-performance
  - Energy efficiency and graphics performance
- Servers
  - Emphasis on availability, scalability, throughput
  - Energy efficiency
- Clusters / Warehouse Scale Computers, "Software as a Service (SaaS)"
  - Emphasis on availability and price-performance
  - Sub-class: Supercomputers, emphasis: floating-point performance and fast internal networks
  - Energy proportionality
- Embedded Computers
  - Emphasis: price , energy
  - Application-specific performance

Another performance requirements can involved bandwidth and latency.

- Bandwidth or throughput
    - Total work done in a given time
    - 10,000-25,000X improvement for processors
    - 300-1200X improvement for memory and disks
- Latency or response time
    - Time between start and completion of an event
    - 30-80X improvement for processors
    - 6-8X improvement for memory and disks

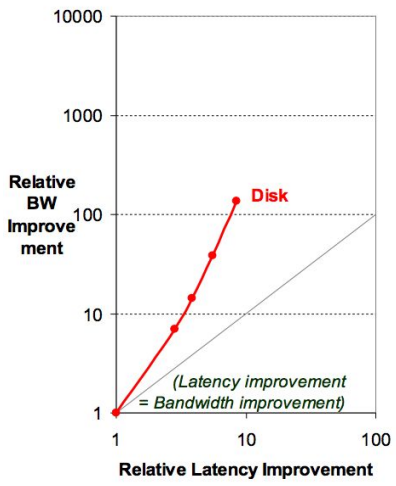
Let's see in which direction we can work to improve bandwidth and latency!

Improve bandwidth and latency

A way to reduce Latency? Cache.
A way to increase bandwidth? Increase bits per single transaction.
Let's see in particular!

Disks

In Architecture, a disk can become a very heavy load for the performance if I want a very low latency. The application must be organized in order to limit this problem. In this image below we can see how bandwidth for disks was improved is the last 20 years.



Bandwidth or throughput
Total work done in a given time
Mbytes/second

300-1200X improvement for disks

Latency or response time
Time between start and completion of an event
average disk access time in milliseconds

6-8X improvement for disks

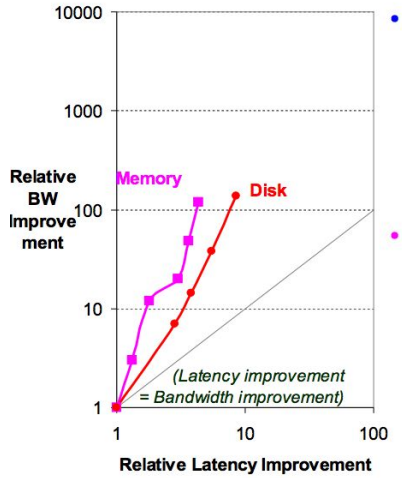
An example of a typical application that has to deal with latency of disks is the Operating System. In a disk the bandwidth can improved changing the arm speed or the rotation speed. Or organizing files in better way. But it's not sufficient or there are physical limit. Another way is to use a hierarchical way of memorize data such cache and RAM. For example, still talking about disk we can see that in years the difference is very high.

Hard disk	3600 RPM	5400 RPM	7200 RPM	10,000 RPM	15,000 RPM	15,000 RPM
Product	CDC WrenI 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453	Seagate ST3600057
Year	1983	1990	1994	1998	2003	2010
Capacity (GB)	0.03	1.4	4.3	9.1	73.4	600
Disk form factor	5.25 inch	5.25 inch	3.5 inch	3.5 inch	3.5 inch	3.5 inch
Media diameter	5.25 inch	5.25 inch	3.5 inch	3.0 inch	2.5 inch	2.5 inch
Interface	ST-412	SCSI	SCSI	SCSI	SCSI	SAS
Bandwidth (MBytes/s)	0.6	4	9	24	86	204
Latency (ms)	48.3	17.1	12.7	8.8	5.7	3.6

But not only the disk speed or bandwidth is important. In a device it's important the memory. Let's see in the same graph in last 20 years how memory throughput is improved.

Memory

For the RAM, increasing the throughput is easy (I can use a bigger bus or increase also the number of MB per chip, reducing its size) but the real problem is latency. And the execution time of an application is directly connected with the memory latency. The solution to this problem is, again, introducing another smaller and faster memory in the hierarchy, being the cache.



- Performance Milestones
- Memory Module: 16bit plain DRAM, Page Mode DRAM, 32b, 64b, SDRAM, DDR SDRAM

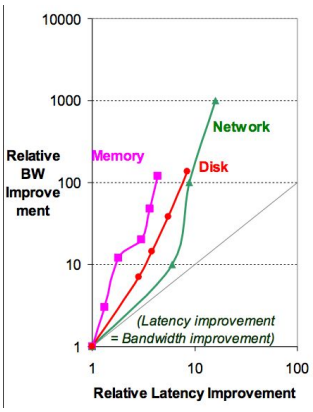
It's like before, we can use cache memory to reduce memory latency! Every memory of the system is simply a level of cache between the disk and the cpu. We have littler and littler caches (reducing latency) because caches consume much power. As before let's compare with real data.

Memory module	DRAM	Page mode DRAM	Fast page mode DRAM	Fast page mode DRAM	Synchronous DRAM	Double data rate SDRAM	DDR3 SDRAM
Module width (bits)	16	16	32	64	64	64	64
Year	1980	1983	1986	1993	1997	2000	2010
Mbits/DRAM chip	0.06	0.25	1	16	64	256	2048
Die size (mm^2)	35	45	70	130	170	204	50
Pins/DRAM chip	16	16	18	20	54	66	134
Bandwidth (MBytes/s)	13	40	160	267	640	1600	16,000
Latency (ns)	225	170	125	75	62	52	37

We have to talk about network. Also consider that network has a latency too (not too much).

Network

The increase of throughput of network is very very important. But is important also the reduction of the latency.



- Performance Milestones
- Ethernet: 10Mb, 100Mb, 1000Mb, 10000 Mb/s (16x,1000x)
- Memory Module: 16bit plain DRAM, Page Mode DRAM, 32b, 64b, SDRAM, DDR SDRAM (4x,120x)
- Disk: 3600, 5400, 7200, 10000, 15000 RPM (8x, 143x)

(latency = simple operation w/o contention
BW = best-case)

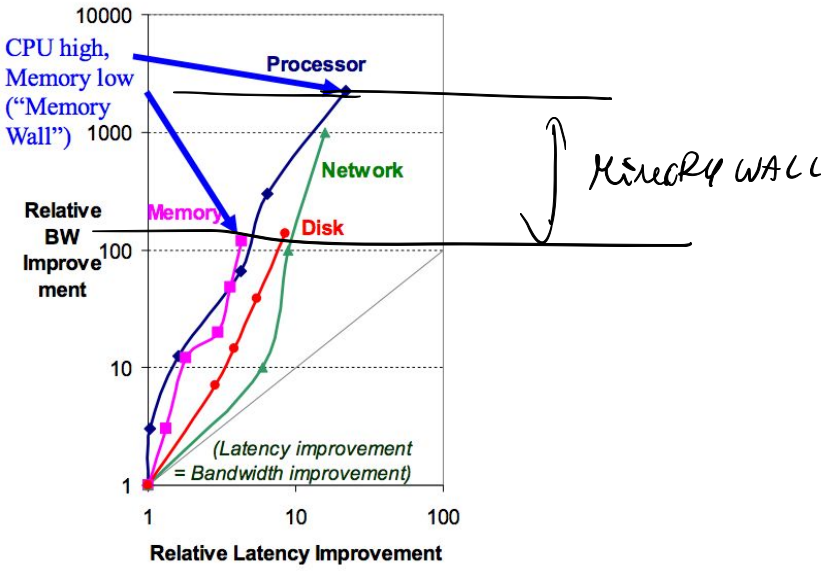
And as before comparing

Local area network	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Ethernet	100 Gigabit Ethernet
IEEE standard	802.3	803.3u	802.3ab	802.3ac	802.3ba
Year	1978	1995	1999	2003	2010
Bandwidth (Mbits/sec)	10	100	1000	10,000	100,000
Latency (µsec)	3000	500	340	190	100

Last but not in list the CPU

CPU

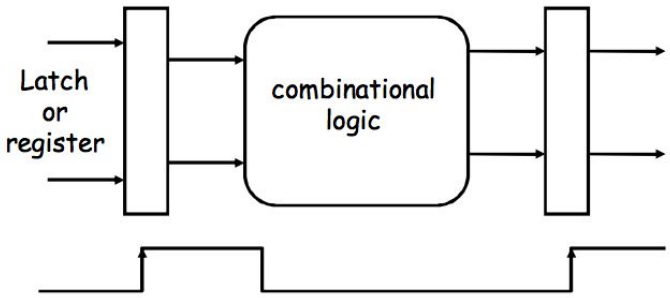
In terms of improving CPU are arrived to very high level. The problem is that memory are not really improved over years and this is the main problem designing a device! In general is possible to reduce latency using throughput. For example the branch predictor use the idea that the prediction of next step/jump very often is correct and save the time of read in memory. But when the prevision is not correct we have to modify the last previous executed instruction. But throughput of CPU is higher than memory so we can use this fact!



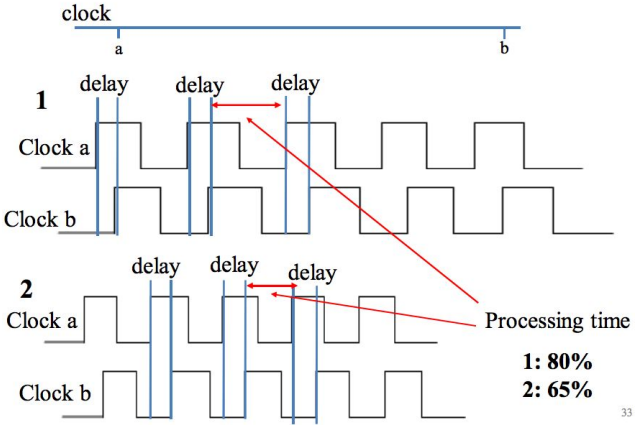
Also is important to consider scalability, hence the need of critical hardware parts must be limited. The latency is the most important problem in designing a device.

Microprocessor	16-bit address/ bus, microcoded	32-bit address/ bus, microcoded	5-stage pipeline, on-chip I & D caches, FPU	2-way superscalar, 64-bit bus	Out-of-order 3-way superscalar	Out-of-order superpipelined, on-chip L2 cache	Multicore OOO 4-way on chip L3 cache, Turbo
Product	Intel 80286	Intel 80386	Intel 80486	Intel Pentium	Intel Pentium Pro	Intel Pentium 4	Intel Core i7
Year	1982	1985	1989	1993	1997	2001	2010
Die size (mm^2)	47	43	81	90	308	217	240
Transistors	134,000	275,000	1,200,000	3,100,000	5,500,000	42,000,000	1,170,000,000
Processors/chip	1	1	1	1	1	1	4
Pins	68	132	168	273	387	423	1366
Latency (clocks)	6	5	5	5	10	22	14
Bus width (bits)	16	32	32	64	64	64	196
Clock rate (MHz)	12.5	16	25	66	200	1500	3333
Bandwidth (MIPS)	2	6	25	132	600	4500	50,000
Latency (ns)	320	313	200	76	50	15	4

Let's start talking about CPU in particular.
It's possibile to increase the bandwidth/latency increasing the clock cycle frequency? Nowadays it's not a good idea. What is a clock?



Clock is used to synchronize levels of logic. A value start from a latch, pass through a logic net and it's exposed in output. But the combination logic has some delay, 0-delay doesn't exist! And this delay must be considered! In past increasing clock seems a very good idea because reduce the delay and the time execution. But increase the clock frequency means increase power consumption. It must be balanced.



Lesson 03 - 03/03/2017

The CPU asks two kind of services from the memories: read and write. An operation is efficient if is asynchronous: read operations cannot be asynchronous, while writes can. That's why the bus is designed in order to guarantee better performances in read mode.

The performance of a bus depends from the percentage of operations that are asynchronous. A read can be seen as an asynchronous operation only in the case of the **prefetch mechanisms**, which overlaps the execution of the current instruction with the execution of the following instructions (using branch prediction for increase performances).

In the same way, the prefetch of the subsequent part of a file. This operation can be automatically performed in advance assuming that the next block that will be needed by the program will be the following block to the current one. The strategy is to reduce the throughput to reduce the latency.

The final consideration is that is important to organize the as an asynchronous set of operation in order to overlap the cost of communication with the work of the processor so sometimes we need to keep some synchronization to increase performance.

Measuring performance

Let's start with the lesson. Today we'll speak about measuring performance.

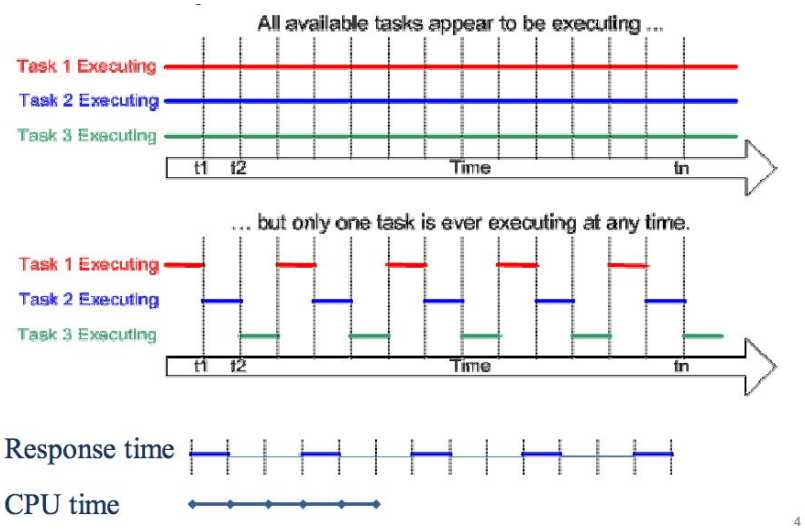
- Typical performance metrics:
  - Response time
  - Throughput
- Speedup of X relative to Y
  - Execution time$_Y$ / Execution time$_X$
- Execution time
  - Wall clock time: includes all system overheads
  - CPU time: only computation time
- Benchmarks
  - Kernels (e.g. matrix multiply)
  - Toy programs (e.g. sorting)
  - Synthetic benchmarks (e.g. Dhrystone)
  - Benchmark suites (e.g. SPEC06fp, TPC-C)

If I want to compare two microprocessor I have to used numbers obtained compared same type of data for example an application on the same systems. Obviously the execution time can be measured in different way, taking account overheads or not. Execution in different instant of time maybe can be different because the workload of the system can vary. **Performance aren't the same for all users involved in a system.** Typically, the end users of a shared service for example want that the service minimizes its response time, whereas

the owner of the service wants to increase its throughput, in order to serve more end user in the same time. So how can we obtain value comparable? Maybe without an OS working only with the kernel or throwing off the cache memory to obtain the worst case execution. But this is not a scientific way of working. In this way benchmark can help us! Benchmark are number calculated with constraint in a way to throw off every details and take in account only application response time.

Notice: two different executions on the same machine can lead to different execution times, due to cache memory and other stuff (kernel, etc.). Benchmark can be useful to get an idea about performance obtained by a specific applications by considering a set of specific procedures.

Response time it's completely different from CPU time! Response time can depends other tasks! Let's see the difference in the image below!



Evaluating speed of hardware, software, application on so on has become an important argument of discussion in the last years, and a consortium has born to help the comparing.

Benchmarks

Some consortiums evaluate performance for microcontrollers/kernels/libraries in order to use them for embedded applications, smartphones, and so on and so forth. When evaluation is completed, a report is produced. It is important to have a report for each component that can affect execution time. The general idea about benchmarks is that the evaluation can be repeated in any moment, so the report must include all the features and all the components used by the evaluation.

A final mark is computed based on all the results among all the evaluations executed. It is important to have a final value, but also values for each execution. This because some applications do not need some features (like for example floating point and so on), which are included in the global final score.

The way to evaluate the global value isn't easy, because I need to consider the score for each solution. I need first of all to decide which value I will use as a score for each feature and then the way I'm gonna combine these values together.

Benchmarks can influence the market of course. In some cases, some companies try to manipulate them. Many times consortium changes the kernel code each year, because a very easy way to obtain a good benchmark is to develop a version of this program well optimized and to store it inside the compiler. When I try to compile the benchmarking program, the compiler gives me the optimized version instead of the original one.

Comparing embedded microprocessors

EEMBC: the web site

On EEMBC we can found different devices and hardware comparing, focused on system in which the had to work. On slides we can found other details about the EEMBC site. An example of report of a benchmark can be

AutoBench 1.1 Production Silicon Benchmark Scores

All Benchmark Scores are Certified by EEMBC to ensure credibility

Export Report to Excel

	Freescale i.MX31-532MHz
Certification Report	View Certification Report to see complete certification data
Type of Platform	Hardware/Production Silicon
Type of Certification	Out-of-the-box
Certification Date	01/21/08
Benchmark Notes	
Hardware Type	Production silicon
Native Data Type	32-bit
Architecture Type	RISC
L1 Instruction Cache Size (kbyte)	16
L1 Data Cache Size (kbyte)	16
External Data Bus Width (bits)	32
Memory Clock (mhz)	133 MHz
Memory Configuration	4-1-1-1
L2 Cache Size (kbyte)	128 KB
L2 Cache Clock	266 MHz
Compiler Information	
Compiler Model and Version	Arm Realview Compilation Tools v3.1 build 569
Floating Point	Hardware

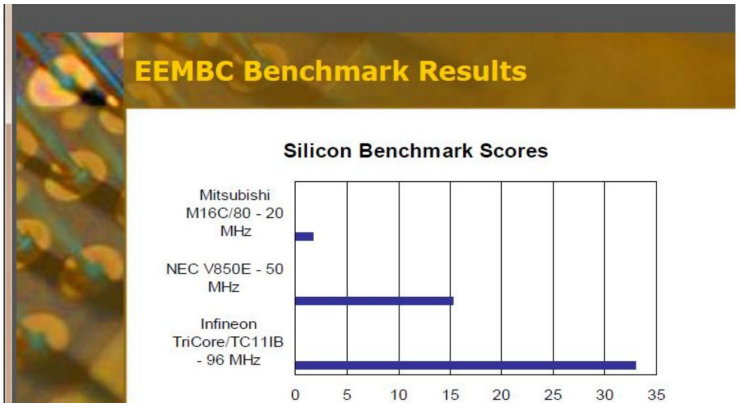
nchmark Scores	Iterations per second
Angle to Time Conversion	528925.03
Basic floating point	303782.18
Bit Manipulation	11307.02
Cache Buster	2671384.64
Response to Remote Request(CAN)	3098804.15
Fast Fourier Transform (Auto/Indust. Version)	614.51
Finite Impulse Response Filter (Auto/Indust. Vers)	190303.63
Inverse discrete cosine transform	10860.30
Infinite Impulse Response Filter	137289.71
Inverse Fast Fourier Transform (Auto/Indust. Vers)	659.47
Matrix arithmetic	1039.70
Pointer Chasing	11708.92
Pulse Width Modulation	2179341.27
Road Speed Calculation	2188980.08
Table Lookup and Interpolation	244956.19
Tooth To Spark	101902.74
AutomarkTM	258.3

Each detail of the simulation is reported cause simulation must be repeated in time. It's important also to have a final score the valuate the device taking in account each value. Actually the final value it's only an indication, for select the correct device that we need the value in feature that we must have in our system must be selected correctly.

On EEMBC the score reported for each device is a single-number figure of merit calculated by taking the geometric mean of the individual AutoBench scores and dividing by 307.455. This normalization factor (307.455) is derived from the lowest score in this category on December 5, 2000. Scores for each of the individual benchmarks within this suite allow designers to aggregate the benchmarks to suit specific application requirements. To calculate a geometric mean, multiply all the results of the tests together and take the nth root of the product, where n equals the number of tests.

$$Score = \sqrt[nth]{a * b * c * d \ldots}$$

Example of possible comparison results:



The AndEBench™ benchmark provides a standardized, industry-accepted method of evaluating Android platform performance

Samsung Galaxy S III (GT-I9300)	9,623		Google Nexus 7 (Nexus 7)	8,572
Google Nexus 4 (Nexus 4)	11,281		Google Nexus 10 (Nexus 10)	6,648
HTC One X	6,154		Sony Xperia S (LT26i)	4,354
Samsung Galaxy S II (GT-I9100)	4,231		Samsung Galaxy Note 10.1 (GT-N8000)	9,801
Samsung Galaxy Note II (GT-N7100)	11,222		Asus Transformer Pad TF300T	7,784
Google Nexus (Galaxy Nexus)	4,343		Amazon Kindle Fire HD 7 (KFTT)	4,210
Asus PadFone 2 (PadFone 2)	11,278		Amazon Kindle Fire HD 8.9 (KFJWI)	5,449
Motorola RAZR i (XT890)	5,907		Samsung Galaxy Tab 2 7.0 (GT-P3110)	3,224
HTC One X+	6,528		Samsung Galaxy Tab 2 7.0 (GT-P3100)	3,557
HTC One S	5,299		Acer Iconia Tab (A500)	2,715

Comparing desktop systems

To evaluate desktop computers another consortium was born. **Standard Performance Evaluation Corporation** that operate Processor-intensive benchmarks and graphics intensive benchmarks. SPEC created a benchmark set focusing on processor: SPEC CPUxxxx. The year (xxxx) on the name of the benchmark specifies the version of the program used to evaluate, all the conditions like the options in the compiler and at the same time the microprocessor used as a reference processor, to compare all the others.

The elapsed time in seconds for each of the benchmarks is given and the ratio to the reference machine is calculated: a **Sun UltraSparc II** system at 296 MHz. The metrics are calculated as a Geometric Mean of the individual ratios, and each ratio is based on the median execution time from three runs.

For graphics performance the **SPECgpc** is a totally new graphics performance evaluation software. Among the major changes are a new GUI, fully updated viewsets traced from newer versions of applications, larger models, and advanced OpenGL functionality such as shading and vertex buffer objects.

Viewperf 11.0

Precision Workstation R5500 NVIDIA Quadro 6000		
Test #	Weight (%)	Frames/sec
1	12.00	49.20
2	12.00	21.80
3	14.00	19.80
4	14.00	33.20
5	12.00	106.00
6	12.00	83.40
7	12.00	71.30
8	12.00	123.00
Weighted Geometric Mean = 50.68		

Comparing database systems

For database manager system there is another consortium. **Transaction Processing Council (TPC).** Of course other metrics and programs need to be used for different applications:

- TPC-C simulates a complex query environment.
- TPC-H models ad hoc decision support.
- TPC-E is a new On-Line Transaction Processing (OLTP) workload that simulates customer accounts.

The three different way of calculate benchmark want to measure performance on systems evaluating different things. For example the first application TPC-C want to measure latency on disk (read and write), the second the computational power. But for test a DBMS I need a dataset and some users that use the dataset. Everything is simulated by the benchmark engine.

Performance evaluation

Now I want to decide which metrics use to evaluate performances (number of instructions, clocks, etc.). In order to compare and evaluate performances of the system it can be used also the CPI metric, which is the number of clocks for each instruction. Usually many RISC instructions are needed to compute a complex instructions and processor is able to execute many RISC operations at the same time via pipeline.

Let's begin with the basis, we want to understand the link among execution time and response time of some components. The first rule is that the CPU time is exactly clock cycles for a program multiply by the clock cycle time divided by the rate

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

Then we can use the CPI that is the number of clocks for each instruction (in general a minimum value):

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

In the same way we can evaluate the number of clock cycles. So the final value can be obtained by the number of instructio of specific for the number of cycles:

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

The formula shows that in order to reduce the CPU Time you can work on the program, on the clock and on the instructions used.

Different instruction types having different CPI's:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

Finally we can use the calculated value to calculate the CPU time of a process.

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

Often this number cannot be significantly because the CPU time depends from different factor. For example if the program uses floating point instructions or not.

Principles of Computer Design

There are some parameters to take in account while designing a "computer".

- Take Advantage of Parallelism
    - multiple processors, disks, memory banks, pipelining, multiple functional units
- Principle of Locality
    - Reuse of data and instructions
- Focus on the Common Case
    - Amdahl's Law

Parallelism

Parallelism takes advantage of the multi processor, balancing the workload among all the available processors. The first step to reach good performance is to use parallelism to minimize the communication overhead by considering the waiting to have data. If I want to minimize the overhead of communication we need to consider at the same time the speed of connection and the amount of data. For example to send a word from a processor to another in a multiprocessor system the cost is similar to acces in the main shared memory. The cost to send a word on a network is similar to the speed of the network. we can have a ratio between this two technology that is about 10 000. Then it's better to minimize the overhead in communication that depends on the speed of the link.

So the application should have a low amount of data exchanged. If my application is designed to have many data exchanged among processes I have to consider a multiprocessor instead of a multi computer structure.

I can have a good solution in a network if the cluster is based on the idea of exchange data reducing the size (of what?) whereas in a multiprocessor you can use big data exchanged because in many cases it is more simple to exchange the address where the data is stored (instead of making a copy of the data).

Principle of locality

The second factor is the principle of locality. Temporal locality states that recently accessed items are likely to be accessed in the near future. Spatial locality says that items whose addresses are near one another tend to be referenced close together in time. If application are well-designed most used instruction/data are "near" in the memory structure. This can help in reuse of data and instruction. Most used data and instruction can be saved in a memory faster and nearest the processor. But the size of this type of memory typically is little compared to central memory.

Principle of Locality

- **Reuse of data and instructions:**
  - Temporal locality states that recently accessed items are likely to be accessed in the near future.
  - Spatial locality says that items whose addresses are near one another tend to be referenced close together in time.

Focus on common case

The last case can be the main strategy to solve problem in case that our system doesn't meet the requirements.

The two types of locality needs different type of cache organization! We have to focus on the Common Case. We have to decide:

- what the frequent case is and
- how much performance can be improved by making that case faster.

$$\text{Speedup overall} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

- **Fraction_e**, *The fraction of the computation time in the original solution that can be converted to take advantage of the enhancement*
- **Speedup_e**, The improvement gained by the enhanced execution mode for Fraction=e,

(Execution time for entire task using the enhancement when possible) **=**
(Execution time for **fraction of task** without using the enhancemen) +
(Execution time for **fraction of task** using the enhancement) =

$$(1\text{-Fraction\_e}) * T\_old + Fraction\_e * \frac{T\_old}{Speedup\_e}$$

$$\text{Speedup overall} = \frac{1}{(1\text{-Fraction\_e}) + \frac{Fraction\_e}{Speedup\_e}}$$

36

Solution	Enhancement	Computation	Waiting for I/O	Overall time
Original	...	7 s	3 s	10 s
Disks	I/O Speedup = 3	7 s	1 s	8 s
CPU 1	CPU speedup =2	3,5 s	3 s	6,5 s
CPU 2	CPU speedup = 3	2,33 s	3 s	5,33 s

The program needs several seconds for computation and several seconds waiting for I/O operations. To increase performance we can use a disc with more speed or change CPU using a faster one. This can be shown us by a performance debugger. A performance debugger is able to show where the execution time is used, is able to show the list of procedures which are time consuming and that should be optimized to improve performances.

Lesson 04 - 08/03/2017

Prof. De Vitis Gabriele Antonio.
We have 10 hours with De Vitis to talk about interesting argument on Computer Architecture.

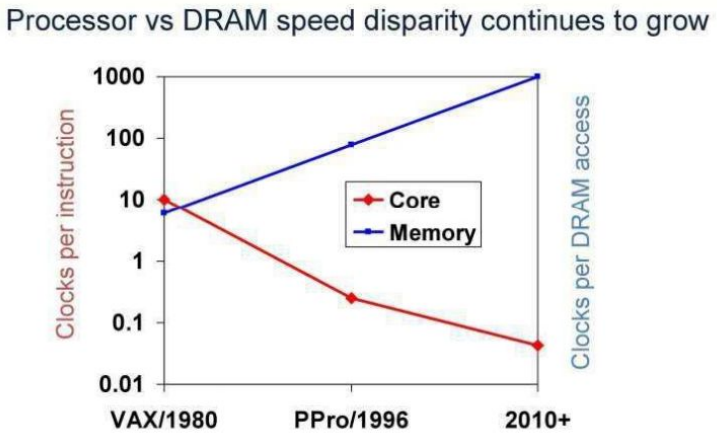
Cache memory

The inventor was Wilkes and in 1965 he talked about dynamic storage allocation and slave memory.

M. V. Wilkes, "Slave Memories and Dynamic Storage Allocation,"
***IEEE Transactions on Electronic Computers*, vol. EC-14, no. 2,**
pp. 270-271, April 1965.

Why cache?

Why is so important cache memory in a calculator? The performance of the cpu and the memory grows exponentially but not with the same speed. So the idea is that the memory is always slower.



Furthermore, now a CPU can do multiple instructions at the same time (pipeline, multicore, and so on) but the CPU needs 1K clock cycle to load a data from the DRAM and this is a problem because an instruction has to wait for fetching the instruction itself and then load the data on which it operates. If the CPU is able to complete an instruction in 1 clock and then it has to wait 1000 for the loading of a data there is a useless utilization.

In the PC we have different kind of memories; fastest memories are always expensive. We can create a simple solution to increase the power of the memory. The idea is that we can build a memory system that we have all the previous memory set in order to have slower memories (that are cheaper) as the same velocity of the more expensive memories.

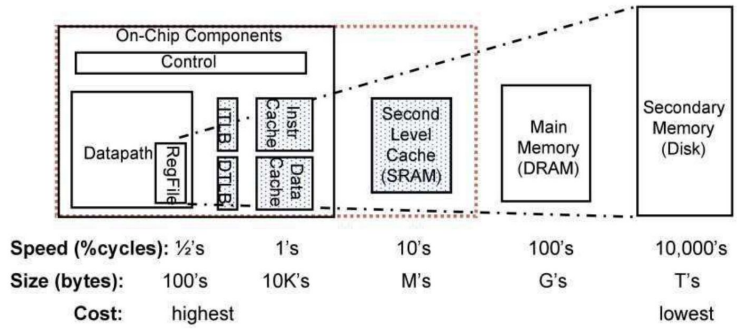
Static RAM (SRAM)
0.5ns – 2.5ns, $2000 – $5000 per GB

Dynamic RAM (DRAM)
50ns – 70ns, $20 – $75 per GB

Magnetic disk
5ms – 20ms, $0.20 – $2 per GB

Ideal memory
– Access time of SRAM
– Capacity and cost/GB of disk

We can create a simple organization of memory that we can consider hierarchical. Cache can create a hierarchy of memories, each littler than the other but way faster, in order to reduce the accesses to the bigger and slower memories.



This is possible due to the principles of **locality**. We can take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology. What is the principle of **locality**?

Temporal locality
If a memory location is referenced then it will tend to be referenced again soon.

➥ Keep most recently accessed data items closer to the processor
e.g., instructions in a loop, induction variables

Spatial locality
If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon.

➥ Move blocks consisting of contiguous words closer to the processor
e.g., sequential instruction access, array data

We can create a memory hierarchy. We can save all the datas in the main disk and when the CPU access to this data I copy these data in the near and faster memory. So when I load a software in an hard disk the OS loads the software in the RAM. Then I load a instruction from the software and so I can load this instruction in a cache memory near to the cpu. This is the basic idea and it works. When the CPU accesses some data, I copy it in a cache memory so I can use it (and the locations near it) in the next future. This is valid both for DRAM (coming from Hard Drive) and for actual cache (coming from DRAM). Caches use SRAM for speed and technology while main memory uses DRAM for size.

Caches use SRAM for speed and technology compatibility
- Fast (typical access times of 0.5 to 2.5 nsec)
- Low density (6 transistor cells), higher power, expensive
- Static: content will last "forever" (as long as power is left on)

Main memory uses DRAM for size (density)
- Slower (typical access times of 50 to 70 nsec)
- High density (1 transistor cells), lower power, cheaper
- Dynamic: needs to be "refreshed" regularly (~ every 8 ms)
  - consumes1% to 2% of the active cycles of the DRAM
- Addresses divided into 2 halves (row and column)
  - RAS or Row Access Strobe triggering the row decoder
  - CAS or Column Access Strobe triggering the column selector

Let's define some words:

- **Block** (or line): the minimum unit of information that is present (or not) in cache
- **Hit rate**: the fraction of memory accessed found in a level of memory hierarchy
- **Miss rate**: the fraction of memory accessed not found in a level of memory hierarchy (1 - hit rate)
- **Miss penalty**: is the cost (amount of time) needed to replace a block with another that produced a miss

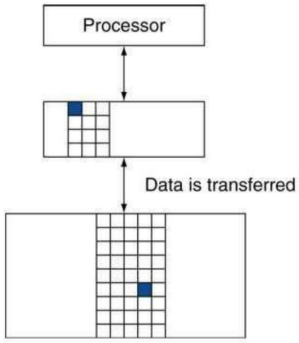
If a block is present in a upper level of cache we have a **hit**. If the data is not present then we have a **miss** and we request the data from the lower level, copying it in the upper level.

Block (aka line): unit of copying
- May be multiple words

If accessed data is present in upper level
- Hit: access satisfied by upper level
  - Hit ratio: hits/accesses

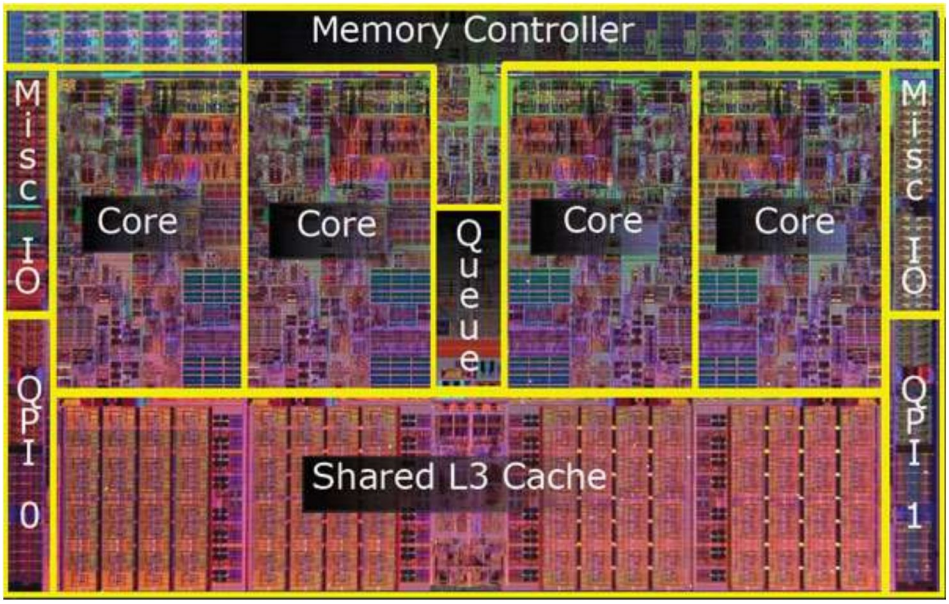
If accessed data is absent
- Miss: block copied from lower level
  - Time taken: miss penalty
  - Miss ratio: misses/accesses = 1 – hit ratio
- Then accessed data supplied from upper level

So the idea of the cache is the we need some level of memory and cache is the memory closest che CPU! The idea is that we have a data in the second memory disk and the cpu wants to access it.



The CPU has to search it in all the memories. Every time the cpu spends to search the data and to load it this is the miss penalty.

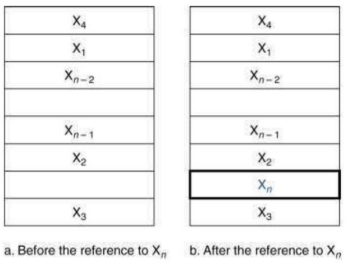
Example of cache occupied space in an Intel i7.



How cache work

The main idea for the CPU is this:
CPU has to access to a block (given by multiple words). If accessed data is in the present in the upper level we can read from there otherwise, if we had a miss, we can search the data in the lower level and then copy it in the upper level. Of course if the data is not present in the lower level we have to go down through the memories in order to find the data. We have to perform an iterative operation.

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses $X_1, \ldots, X_{n-1}, X_n$



- How do we know if the data is present?
- Where do we look?

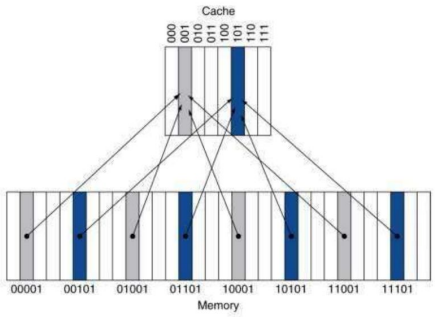
But how do we know if data is present in cache? Where we have to look-up? We have to take in account that cache memory is smaller than RAM! Usually cache are inside the processor to access it faster. In general in a cache we have a certain number of blocks, but how CPU can load it?

We will see a simple way to load data from the cache. A simple technique could be:
we have 8 blocks in cache and so, if I want to load the first block of main memory in cache, I can bring the block in the first block of the cache and so on until the last block.
When the ninth arrives we can put it in the first block of the cache and so on. The simple technique can be to copy data in circular way. When the cache memory is full we use the module operator to restart from the first. This technique is called **direct mapped**.
In order to compute the address in cache memory we need to:

$$(address \in RAM)\ mod\ (\#blocks \in cache)$$

In few words the cache keeps information about the blocks in it using the low-order address bits that characterize that block. A first choice would be to assign to a set of blocks always the same block (**direct mapped cache**).

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

The CPU calculate the correspond block in cache but in the same block we can have different blocks. We need to store the information and we need a tag value, that contains the information about what block is contained in the memory. At the bootstrap the cache memory has random values and so we need also another value that says if the information in the cache is valid or not. So in the cache we have a sort of table with index, valid bit, tag and finally the block itself.

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
    - Actually, only need the high-order bits
    - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

If the CPU wants to access the data in the cache we can compute the address in the cache as in the previous description. Then we need to check the bit valid and compare the tag with the tag of our data address to understand if the data is present. If the block in the cache is empty (V = 0) or if the tag does not correspond we need to go in the RAM memory and copy the data in the cache block. If the tag is different when we copy the new block and we override the previous one destroying it.

In general when we want to access a block we can divide the address in 3 parts:

- **Tag** (part on the left of the address)
- **Index** (central part of the address)
- **Offset** (end of the address)

The number of bits of the offset depends on the size of the block. The number of bits if index depends on the number of blocks in cache memory. Finally the other part is given by the tag.

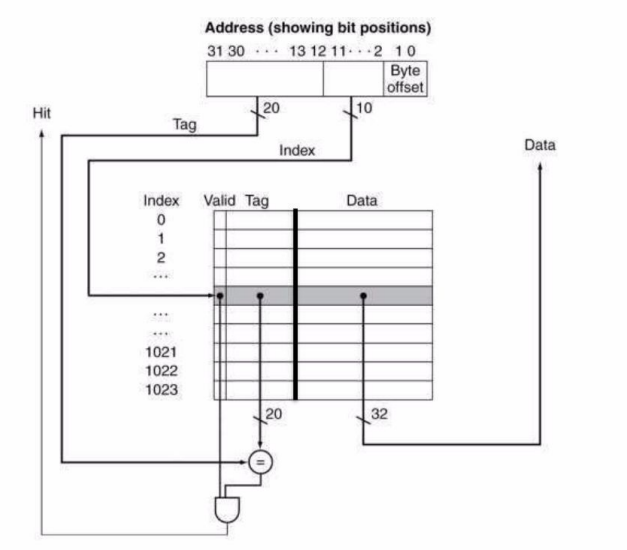
Let’s see an example on how direct mapped cache works! Let us suppose we work with with an 8-blocks cache. Each block can contains only 1 word.

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

At the beginning cache doesn't contain any data (random or 0's). All valid bit are 0 (**N**). Let's start with the assumption that each block contain only 1 word.

// ESEMPIO

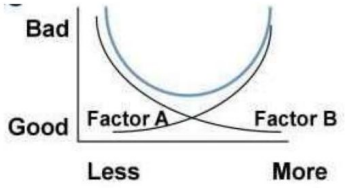
Direct map is resumed in the figure below.



How many blocks can be put in a cache? How much large can be a block? If a cpu access to a data will access a near data and we can create larger blocks. But if we have larger blocks we have few blocks. If a cpu accesses to a data in the future it will access to a same data and so for this property it's better lots blocks and so they should be smaller. If we increase the size of the block we have less miss penalty. The best choice is to have block with small size. According to the temporal locality, we would like smaller blocks. Because the program tends to access to the same locations so I'd like to have more smaller blocks. If I want to take into account the spatial locality I'd like bigger blocks. If I consider the miss rate and the cache size we can see that increase the number of block size I decrease the miss rate but it's also true that if we increase the block size the miss rate grows.
Increasing the block size decrease the miss rate (due to spatial locality) but increase (due to temporal locality).

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer blocks
    - More competition ⇒ increased miss rate
  - Larger blocks ⇒ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

The composition of these two factors builds up a curve that shows the miss rate for a particular cache. This however is not true for all softwares, some software may take more advantage of spatial locality while others could use a lot temporal locality. The best cache cannot be decided in an absolute way.



In general every time we have a miss we need to stop the cpu from continuing the pipeline execution, Then it has to access the block in the main memory and copy it into the cache.
Compulsory misses are miss due to an access to a data for the first time. This miss cannot be avoided. By other types of miss can be avoided. For instance if a data was in cache and it was override we can avoid this type of miss. In fact these misses are due by the conflict of the location, in fact lots of blocks in memory share the same block in cache.
In different kind of caches. In the direct access mapped cache we can search the data only in a block (no in others). We can have different caches in which a block can be in more than one block in cache.

Associativity of a cache and multiple ways

We can have different kinds of access policies to a cache instead of direct mapping. For example in direct mapping a single data will always appear in the same block. In an associative cache the same data can be loaded in multiple locations (in different moments of course) in a so called **Associative Cache**.

Allow for more flexible block placement:

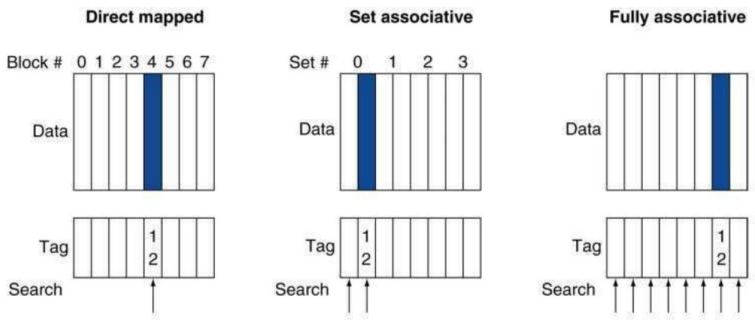
- In a direct mapped cache a memory block maps to exactly one cache block
- At the other extreme, we can allow a memory block be mapped to *any* cache block – fully associative cache

The difference with the previous architecture is that we can a certain number of blocks in cache at the same index. So for the same index field we can have at least 2 fields for tag and 2 for the data. The idea to search the data is exactly equal to previous one (we divide the address as in the previous case). The difference is that now we have to compare the tag for each block that can be in cache. When all the blocks are busy we need to choose one of the blocks to override it (we could add new blocks for the same index to avoid this problem, of course we have a bounded dimension of the cache to take in account). **We have a fully associative cache when the block of the main memory can be in a whatever block in the memory**. In this case we have no indexes and we need to compare all the tags in the of all the blocks in the cache. In this case the number of conflicts is dastrically decreased, because in this case we are using all the capacity of the cache to save a block. Every miss is caused by the fact that the cache is full and another block as to be destroyed. In a **Fully Associative Cache** can be in any block of the multi-cache, a miss in the cache can be caused only by the capacity of the cache, not because there is no space for a block when the cache actually is not full (there cannot be conflict misses).

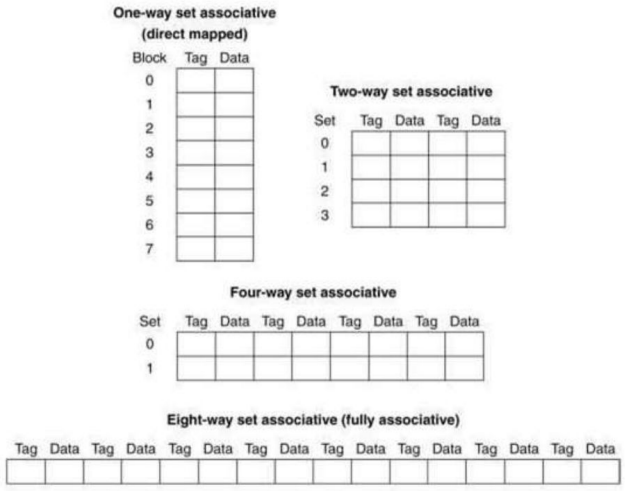
- A compromise is to divide the cache into sets each of which consists of n "ways" (n-way set associative).
  - A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)

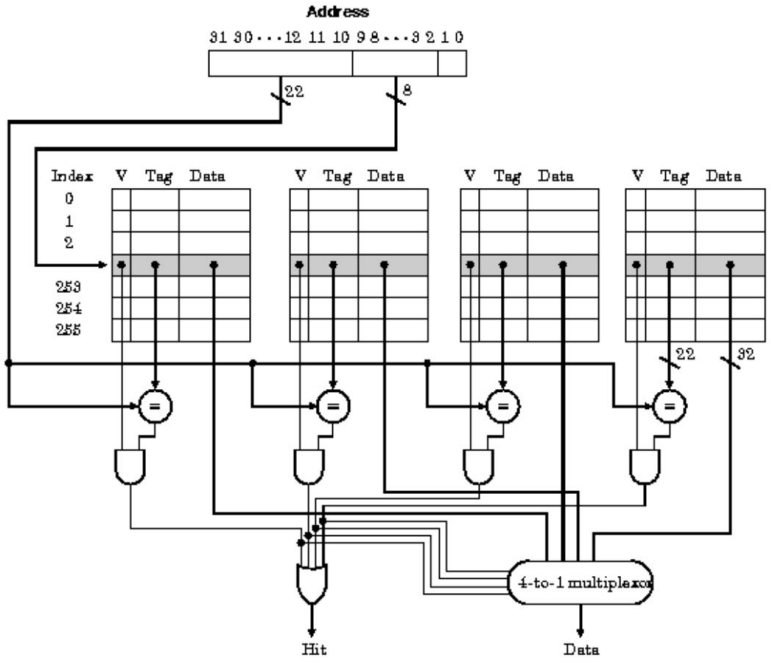
This could be extended even more using a **Set of Associative Caches** (each with the same policy).

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) *modulo* (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)



For a cache with 8 entries:





We can have a miss for **compulsory** (first access), **conflict** miss (when we have to override someone with the same index) or a **capacity** miss (if the cache is full completely and so the definition is in general “capacity *miss is the miss that cannot be avoided in fully associative caches, where we cannot find conflict miss but only capacity miss*”).

On cache hit, CPU proceeds normally

On cache miss
- Stall the CPU pipeline
- Fetch block from next level of hierarchy
- Instruction cache miss
  - Restart instruction fetch
- Data cache miss
  - Complete data access

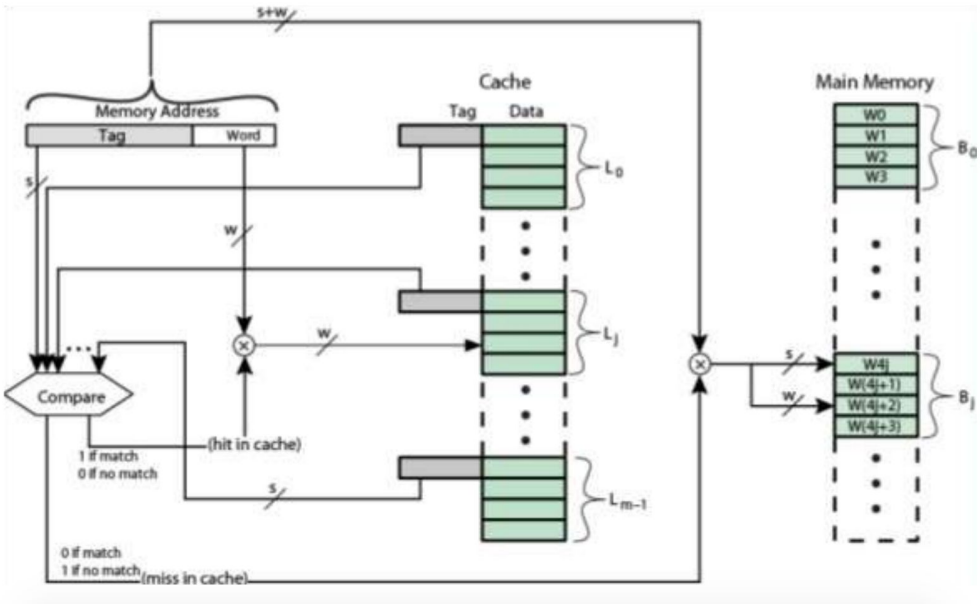
If we have more associativity we need more time to access the location. A problem is that if we want translate an address we need to access the memory twice, director and . So in order to access a location the CPU has to access twice in the memory and this cost is big in terms of access time.

Compulsory (cold start or process migration, first reference):
- First access to a block, “cold” fact of life, not a whole lot you can do about it. If you are going to run “millions” of instruction, compulsory misses are insignificant
- Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)

Capacity:
- Cache cannot contain all blocks accessed by the program
- Solution: increase cache size (may increase access time)

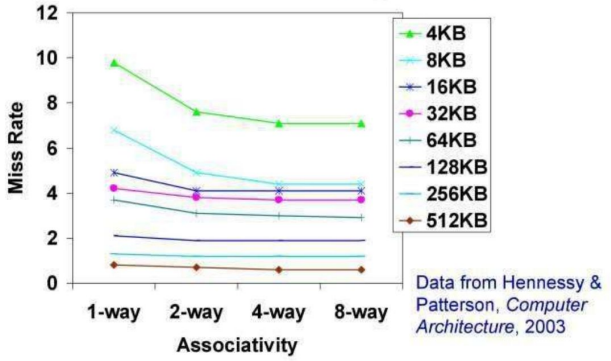
Conflict (collision):
- Multiple memory locations mapped to the same cache location
- Solution 1: increase cache size
- Solution 2: increase associativity (stay tuned) (may increase access time)

We can introduce the **TLB** to translates the addresses. In this case we can use a fully associative cache for the TLB because we can use an expensive solution. It is implemented in this way because it is better in terms of performances, the other caches are not implemented in this way because this implementation has a great cost for all the comparator and so they are expensive.



In the CPU we have the prefetch unit that uses the sequential locality (the says that we will access the next location in the future probably). The sequential locality is present also in data stream when we use arrays of course. In general in application in which the part of the program that performs I/O operations the cache is useless in general.

The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation



Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

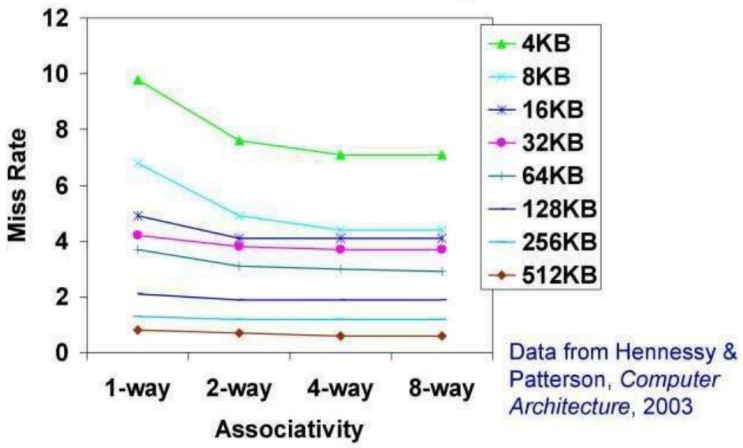
We can observe that the miss rate decreases with the number of associativity, in particular we reduce the number of conflict miss. This number goes to 0 with fully associative caches. If we increase the size of the cache we reduce the number of capacity miss.
We can reduce the misses but we need to pay in terms of complexity (for example in terms of comparators).

Lezione 05 - 09/03/2017

Resume of previous lesson conclusion

Let's resume the lesson of yesterday.
There are different type of miss:

- compulsory miss cannot be avoided (constant for every type of cache)
- capacity miss caused by the used of small size caches
- conflict miss when two blocks have to be placed on the same block in cache



Increasing the number of associativity we can reduce the number of conflict miss. The number of conflict miss can be reduced to 0 if it's used a full associativity cache. The size miss can be reduced to 0 only if the cache it's big as memory (that's impossible!).
Clearly increasing the n-way of associativity increase the complexity of hardware construction (because we need a comparator for every block of cache and this cost is very very high).

Replacement policy

When CPU need to access the main memory, cache is updated and one of the current block is substituted with a block from the main memory. How can CPU can perform that replacement?

We have four different way:

- For direct mapped cache
    - no choice because the block is chosen statically (prefixed position in the cache)
- For set of associative
    - prefer non-valid entry if there is one or choose among entries in the set
    - Least recently used (LRU)
    - Random (obviously this policy is not optimal)

For direct mapped cache we have no choice, block in cache have prefixed position in cache.

In associative cache we can use different policy to replace the current block with the new one. The first (easiest) can be random way. Clearly simple but not efficient although for high associativity can provide performance approximately equal to LRU. Another way can be round robin. The most used the LRU (least recently used). This is the one used most because we need only few information and it's efficient (order that the blocks in cache are accessed the information about the "use" is changed during an hit and is reset during a replacement and the solution to update this information should be based only on 1 write, no time to perform a read and a write).

The LRU algorithm choose to substitute the block unused for the longest time. Clearly it's not simple to keep the information of when the block was used without wasting memory and computation time. Also increasing the associativity, manage the LRU information become almost unhandable. For this reasons typical is used a pseudo-LRU that uses a smaller number of bit to maintain the information required by the algorithm.

For sure we can perform two type of operation, the **read** and the **write** operation. When we perform a write operation how does cache works? If CPU have to perform a write operation at the end the value in main memory have to be updated. Thus when CPU have to update the value in main memory, there are in principle three different policy:

- Write-through with fetch
- Write-through without fetch
- Write-back

Write-through policy

The first is the write-through. In the same time CPU update simultaneously cache and main memory. The problem is main memory is slower than cache memory, so we can implement a sort of "write buffer" in order to permit to the CPU to perform operation immediately. And often is implemented as cache memory.

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also updates memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

Another policy is the write-back. In this way the block in main memory is updated only when the corresponding block in cache is replaced. CPU write only in cache but when the block

must be replaced from cache it's updated also in main memory. The problem is that if we have a lot of cores or CPU and someone read a value from the memory it can be not the most recent version. Hence one update operation can be lost. A solution can be that the controller of the cache is able to listen to the operations in bus and when he percepe that a CPU want to read a non-valid value, the controller invalidates the operation and check for right value.

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

Write Allocation

But what happen if we have a miss during a write operation? With the write back CPU has to fetch the main memory in cache and update the value in cache. Instead with the write through we can fetch the block and update both the values or don't fetch the block at all and update directly the value in main memory! So we have three policy, the combination of write through with the two policy and the write back. For the write back we don't have any choice!

- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

It's important to note that the write policy of cache is fundamental and inflate the performance of the system! Write operation in fact use a bus cycle incrementing the throughput of the bus! Also another thing to note is the wire from CPU and cache can support 32 bit (64 bit) transfer while the cache to memory bus can reach 256 bit. In this case write through is slower that write back!

A bad cache can have 5 % as a miss rate (for a read). The percentages of writes for a program is about 20-50% and since the previous choices provide a solution for reads, a write back can provide a solution also for the write.

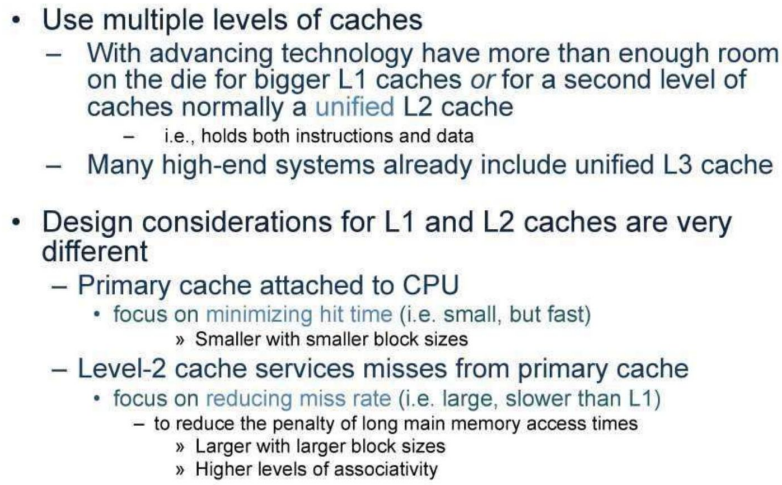
Measuring Performances

Let’s try to measure single performance:

Important to remember:
AMAT = Hit Time + Miss Rate * Miss Penalty (cost expended by the CPU in order to acquire the block from main memory and put it in cache)

Multilevel

In history CPU have different level of cache, and the reason are many. At the beginning decide to include in the package of CPU a small cache and permit to another company to implement another memory of cache or not. This implementation can be usable for lot of reason, for example for policy reason access, using the first level of cache with associative way and the second level with direct blocking. Another reason can be that we can put for example private values for a thread or a process in a nearest cache and put shared variables in L2 cache (outside core or CPU). So that changes in a thread do not invalid values in the second cache, while updating a global variable updates the other cache too. Multi level cache can be inclusive or exclusive. In inclusive pattern the cache is contained into another one. Hence is called **inclusive cache** if L1 is a subset of the second cache L2 (total size of the cache is the size of L2). The exclusive pattern instead state the first level of cache is not contained the second. Hence If L1 is not a subset of the second cache L2 we talk about exclusive cache.



This policy is very very strategic in multithread and multiprocessor environment in order to avoid too many operations on the level of cache L2.

In the first policy L1 contains a subset of L2, thus L1 contains all the L2 memory plus another block (maybe the olds discarded). In Inclusive caching, updates in L1 should be committed on L2 too. We can have a miss in L1 and a hit in L2: in this case we can update the block from L2 and update the value in L1 cache. In L1 we have to replace a block.

If we have a miss in L2 too we have to take the block from the main memory and in order for the property of inclusive caching to be valid always we put the block first in the L2 cache and then in the L1 cache

In the second policy L1 does not contain L2. So everything depends on the data modified. Suppose that L1 and L2 contain different elements (no element is shared).

$$Size(cache) = Size(L1) + Size(L2)$$

If we have a miss in L1 but a hit in L2, the block that is in L2 is taken and put in L1, while the block in L1 will be moved in L2 (exchange). Moreover if we have a miss in L1 and another miss in L2, then the block is taken from main memory and put in L1, a block in L1 is put in L2 and a block in L2 is write back in main memory.

Primary cache
– Focus on minimal hit time
L-2 cache
– Focus on low miss rate to avoid main memory access
– Hit time has less overall impact
Results
– L-1 cache usually smaller than a single cache
– L-1 block size smaller than L-2 block size

Let's talk about AMAT.

$$t_{access} = t_{hitL1} + p_{missL1} \cdot t_{penaltyL1} \leftarrow (p \text{ is a probability})$$

$$t_{access} = t_{hitL1} + p_{missL1} \cdot t_{penaltyL1} \leftarrow (p \text{ is a probability})$$

$$t_{penaltyL1} = t_{hitL2} + p_{missL2} \cdot t_{penaltyL2}$$

thus

$$t_{access} = t_{hitL1} + p_{missL1} \cdot (t_{hitL2} + p_{missL2} \cdot t_{penaltyL2})$$

Advanced CPU

A CPU can execute other operations while waiting for a block to be acquired from the main memory (waiting is not the best solution). In these architectures the effect of miss on execution time depends on the program data flow thus is much harder to analyse and system simulations are needed. Other instructions can be executed using a pipeline with reservation stations.

Out-of-order CPUs can execute instructions during cache miss

- Pending store stays in load/store unit
- Dependent instructions wait in reservation stations
  - Independent instructions continue

Effect of miss depends on program data flow

- Much harder to analyse
- Use system simulation

Victim Cache

Contains only the values that are removed by another cache, but not the same values. L2 in exclusive cache is a victim cache. If we have a miss but we decide to remove the "wrong" block (a block that will be referred again soon) but if we moved it in a victim cache we can retrieve it a little faster than retrieving it from main memory.

Instead of completely discarding each block when it has to be replaced, temporarily keep it in a victim buffer.

Rather than stalling on a subsequent cache miss, the contents of the buffer are checked on a subsequent miss to see if they have the desired data before going to the next lower-level of memory.

- Small cache (e.g., 4 to 16 positions)
- Fully associative
- Particularly efficient for small direct mapped caches (more than 25% reduction of the miss rate in a 4kB cache).

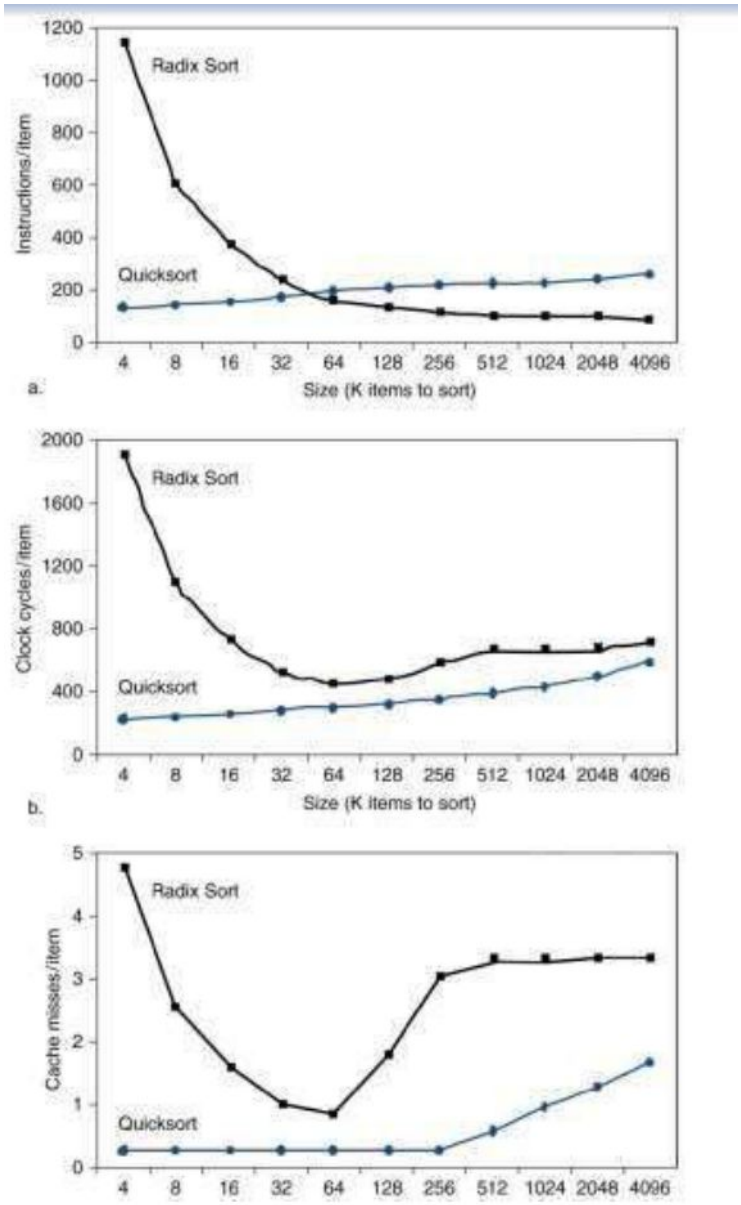
Split cache

Another way of organize cache is the split cache. This is a cache divided in two sub caches, a part of cache contain only instruction, the other only data. We can have I-cache for instruction and D-cache for data. How the cache understand if CPU want data or instruction? The processor can know what he wants to retrieve depending on the segment or its state when retrieving the data (in fetch for sure I want an instruction, while if I am in a read operand or write result the CPU is handling a data). A unified cache has both data and instruction. It’s better the normal cache with instruction and data mixed or the split cache? Well it depends from the software we want to execute! Because if we have a small software that handle some data, we can take advantage of having instruction in the block. The unified choice could be better if we have a balancing between amount of code and amount of data

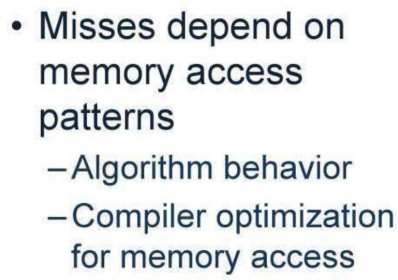
Performance

In general performance depends. Depends from some architectural factor but also depends from algorithm behavior. In order to build the perfect system we have to perform benchmark. For example many algorithms normally executed in computers are sorting algorithms and thus we can check the performances with both and choose the best. Also the compiler optimizations for memory access should be taken into account when designing a cache.

Exists a lot of sort algorithm (bubble sort, insertion sort, selection sort, quick sort, merge sort and so on). Here we have a comparison with Radix sort and Quicksort.



In this slide we can see how cache behaves managing the sorting of an array depending on algorithm used. Clearly compiler can help to increase the performance organizing data in different way.



Thus if a want to reduce the time to hit in cache? For sure

- Smaller cache
- Direct mapped cache (it's easiest to manage)
- Smaller blocks (the multiplexer is smaller)
- For write
    - no write allocate - no hit on cache, just write on the buffer
    - write allocate - to avoid two cycles pipeline writes via a delayed write buffer to cache

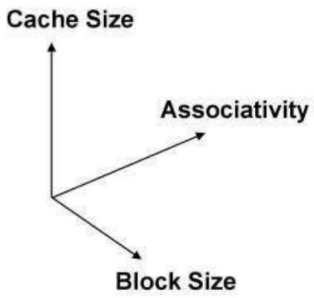
If I want to reduce the miss rate

- A bigger cache (increment the probability of found something in cache)
- More flexible placement (for example increasing associativity)
- Larger blocks (same reason of the first, 16 to 64 bit typical)
- Victim cache (small buffer holding most recently discarded block)

Conclusion

In conclusion, we've seen the cache and what are the important feature?
There are several interacting number of component that change the factor of increasing performance:

- Cache size
- Block size
- Associativity
- Replacement policy
- Write through vs write back
- Write allocation



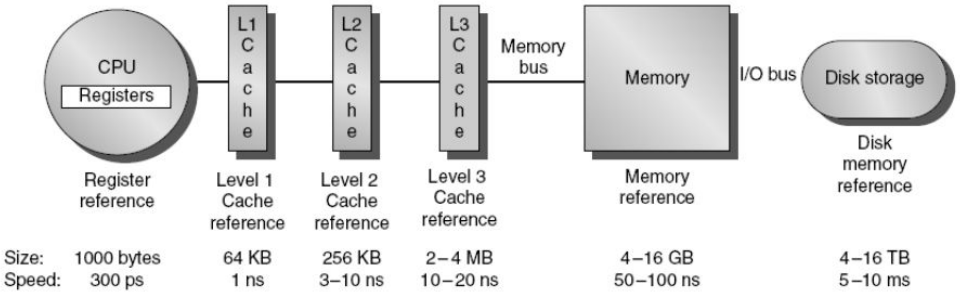
Lesson 06 - 10/03/2017

Memory Hierarchy

The memory is the main problem in computer design. The latency of the memory can be up to 100 times more than the latency of CPU. Also in year memory dimension has become bigger than past and this require overhead in handling. Often the power consumption make impossible to solve completely this problem. The programmer wants very big memories with a faster access time. Thus how can computer designer approach to resolve this problem?

An economical solution is a memory hierarchy, which takes advantage of locality and tradeoffs in the cost-performance of memory technologies. The principle of locality, says that programs do not access all code or data uniformly:

- locality occurs in time (temporal locality)
- and in space (spatial locality).



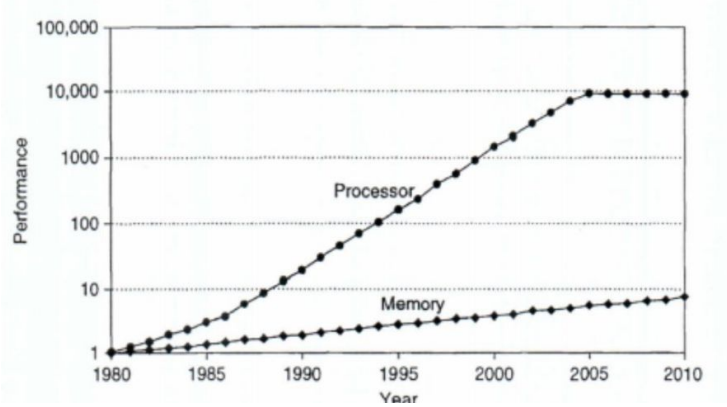
Different level of hierarchy of memory can help saving power consumption per bit and reduce latency. The very first memory used by the computer is the register, next we have the cache, then the memory and the virtual memory on which some copies of processors are on the disk. As we can see, these memories reduce the power consumption for each bit going from the registers to the disks. But at the same time, the access cost increases in computational terms.

Let’s enlightened the main difference between a register and the cache from the logical point of view: a register can store the value of a specific variable, while the cache is able to store copies of memory blocks, which can be code or variables. The main difference however is that the register is managed by the program, while the cache is not (in fact the cache is transparent for the programmer, is automatic).

In fact we can have some microcontrollers on which there is an high speed memory that can be used as a cache or as an high speed memory. The difference is that the cache is automatically managed by the controller, whereas the memory is managed by the program itself, which stores the most used data in it “manually”. The initialization program decides if

that memory can be used as high speed memory or as cache memory. This is typical of many microcontrollers.

However, even with performance increasing in terms of latency of operations is much slower than the processor, even with cache and all other stuff. Obviously we have to consider that the processor is organized in a parallel way, and we can also have more than one core per processor. In the last years architecture of CPU is changed among number of cores, organization, bandwidth. The memory has increased so much in size but not so much in speed! How we do?



Let's see an example of **Intel i7 CPU**.

Intel Core i7 can generate two data memory references per core each clock cycle

- with four cores and a 3.2 GHz clock rate, the i7 can generate a peak of 25.6 billion 64-bit data memory references per second,

- to a peak instruction demand (for four cores) of about 12.8 billion 128-bit instruction references per second;
- total peak bandwidth of 409.6 GB/sec

This processor can generate two data memory references per core each clock cycle, so the processor can generate a peak of 25.6 billion memory references per second. At the same time, the system read instructions to satisfy 4 processors inside. For this purpose you can use a big bus. The total peak bandwidth is about 409.6 GB/s. In the same time the bandwidth offered by DRAM memory is about 25 GB/sec (6%). How this incredible bandwidth is reached and how can we increase?

- Organize the system in a different way, using pipeline and using the cache in pipeline mode (on the bus at the same time, there are two memory transfers, on the first the bus transports the address and in the second the bus transports the data; to increase performances, while transferring the data, the address of the next referenced datum is transferred on the address bus)
- Use multiple levels of cache

- Using separate instruction and data cache at the first level, so that the processor can access at the same time both the instruction and the data on which it operates
- Using a first level private for each processor and a second level shared among processors, in order to minimize the changes in memory due to multiprocess.

On the bus in the same there are two memory transfer. In the first cycle the CPU send the address, in the second the CPU read/write the data. Using the pipeline organization only in one cycle address and data are passed. There are two different group of wire in order to achieve pipeline. Also **another level of cache** is used (or another 2 level of cache!) in order to minimize the conflict.

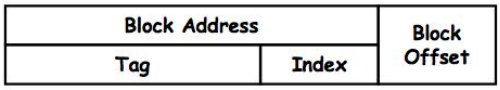
In same case maybe we can use the write buffer. Using a write buffer is strategic because the write buffer can reduce the time needed by the processor for write operations, because more subsequent writes for the subsequent addresses can be performed in one single operation on the low level memory.This increase the bandwidth in case of a high number of writes. For example the write of 4 byte in the same address can be satisfied with only 1 byte in main memory. This solution is used in ARM architecture. In multi-process environment often we found the same thing but there is the well known problem of many copies of data maybe not all updated. In multiprocessor environment in fact is called invalidation buffer. This solution is more complex in multiprocessor because we can have the more copies of the same memory block, one for each different private cache.

Cache

Let's examine in depth the cache discussion. We all remember the basics of cache.

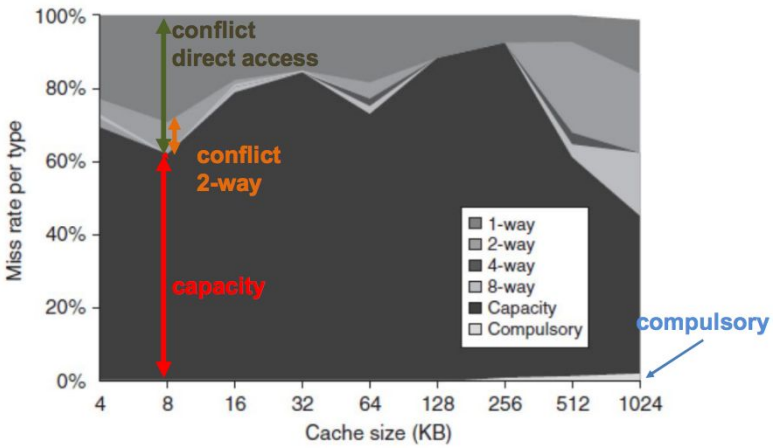
When a word is not found in the cache, a *miss* occurs:

- Fetch word from lower level in hierarchy, requiring a higher latency reference
- Lower level may be another cache or the main memory
- Also fetch the other words contained within the *block*
  - Takes advantage of spatial locality
- Place block into cache in any location within its *set*, determined by address
  - block address MOD number of sets



Miss rate is the fraction of cache access that result in a miss. The principal causes of misses are

- Compulsory (first reference to a block)
- Capacity (blocks discarded and later retrieved)
- Conflict (program makes repeated references to multiple addresses from different blocks that map to the same location in the cache)



This however changes if we change the structure of the program in execution.
You can change cache organization by considering the typical application executed on that processor: very good idea in embedded environment. In any case, the idea to work on the capacity miss is good, because the number of misses due to capacity is very relevant. At the same time a good idea would be to work on conflict misses for specific applications.

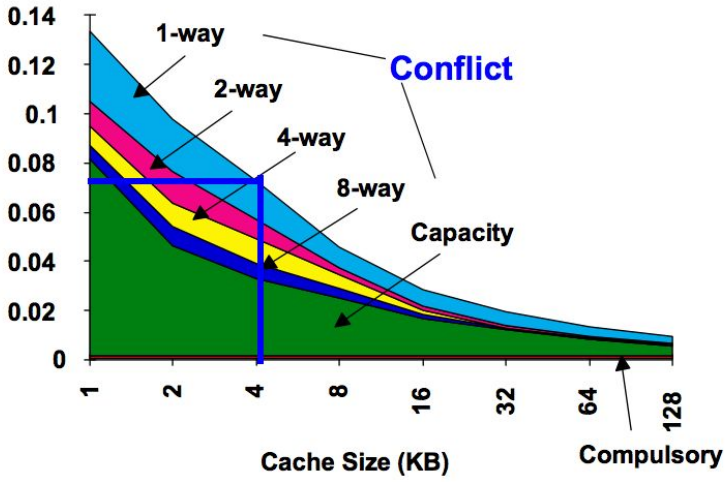
The same situation considering the absolute value. A compulsory miss can offer more ways to store a specific cache block: from this point of view, an associative cache would be the best, but at the same time if I want to obtain a specific percentage miss I have two possible solutions:

- bigger one way associative cache
- smaller two way associative cache

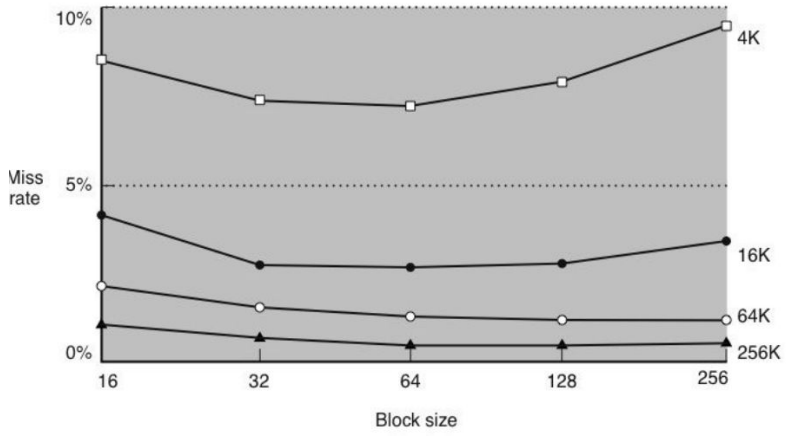
The conflict miss can be reduced by offering an high number to store the specific data. In this graphic maybe the 8-way associative cache seems better but there are a main difference between these two solutions:

- global size (linked to the power consumption and to the space, physical size of the chip)
- At the same time, a 8 way associative cache means a cache organization very complex, so we have to consider also power consumption of such complex solution based on 8 way associative caches

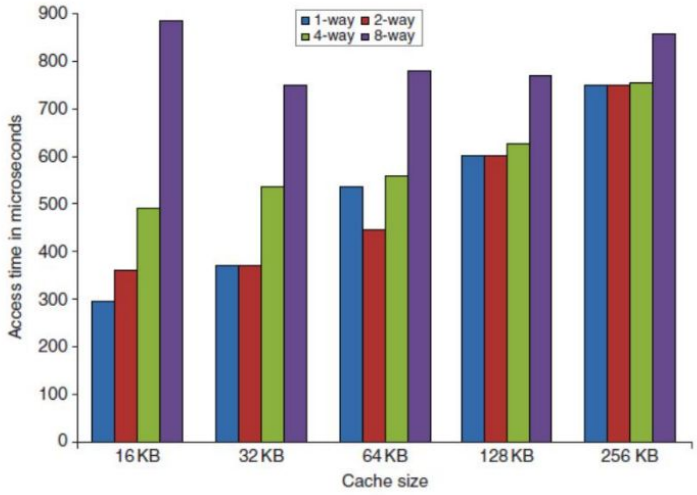
miss rate 1-way associative cache size X
=
miss rate 2-way associative cache size X/2



The slide below instead show the the size the influence the miss rate. The miss rate depends also on the block size, because a small size for each cache block will use temporal locality, because for temporal locality I need more register with low size. Whereas increasing the block size would support the spatial locality, so we have to take a solution in the middle. Hence also the block size take part in miss rate! The global optimum point between the two depends also on the global size of the cache. Thus benchmark help us to decide how dimension our memory.

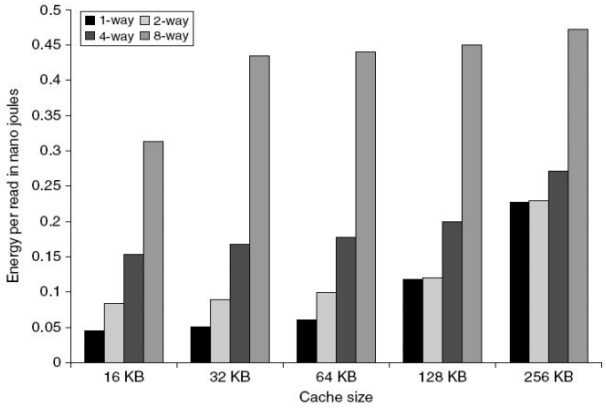


In the same time we must consider also the power consumption and the access time. Clearly considering the same technology for the solution. Hence in any solution, the access time for direct access cache, is very low whereas the access time for 8-way associative cache is 2 times the direct access time. So if I want to satisfy requirement of high speed of CPU I must use the 8-way associativity solution but this increase the power consumption. We need to find a tradeoff.



For small caches, to satisfy the high speed required by the processor I need to use a better electronic solutions (which consumes more power), while using larger caches doesn't change this very much. Again the picture depends on the features of specific programs, so I need to evaluate the behavior of a cache organization by using the typical application that will be executed on that microprocessor, which depends on the features of the processor, like instruction set, how the compiler translate the program, how the link works, how the main memory manager works and so on. Well in these way compiler, linker, loader can help us but it's not a stable situation. The variable in game are so many and in most of that we don't have control. The best organization of cache memory does not exist in absolute value, all depends on our requirement hardware and software.

Energy per read vs. size and associativity



Finally energy consumed per read increases at the increase of cache associativity. A direct access cache consumes less energy, whereas is not a good idea to use a 8 way associative cache from this point of view. Again the picture depends on the program.

Prediction

A way to reduce power consumption and preserve the speed of cache is to use an 8-way associative cache but power on the one of the channel only if we have to read/write the block in that channel! The reduce at all the power consumption but how can we know where in which channel the block is?

To improve hit time, predict the way to pre-set mux
- Mis-prediction gives longer hit time
- Prediction accuracy
  - > 90% for two-way
  - > 80% for four-way
  - I-cache has better accuracy than D-cache
- First used on MIPS R10000 in mid-90s
- Used on ARM Cortex-A8

Extend to predict block as well
- "Way selection"
- Increases mis-prediction penalty

We can use a predictor and only after a miss on the predicted way cache we can turn on the other ways. The access time increases, while the power consumption increases. This because when we look for the copy in the not predicted ways we have to way to turn them on. This solution is usually applied in embedded processors to reduce power consumption.

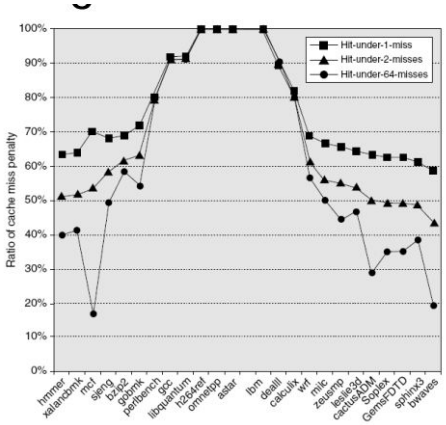
Non-blocking cache

When we have a miss, the cache has to load data from main memory (1000 times slower than cache memory). In traditional cache is a miss we have to wait to get the copy of data that is stored in the main memory. CPU has to wait at least one clock cycle but maybe can be more than one. Let's suppose that CPU has to read also the cell after. Another time lost! We can use non-blocking cache to reduce this lost time updating in one time the first level of cache and use the second level to work. Clearly we can't hide both level 1 and level 2 miss but in future surely the level 2 miss is no more present!

Allow hits before previous misses complete
- "Hit under miss"
- "Hit under multiple miss"

L2 must support this

In general, processors can hide L1 miss penalty but not L2 miss penalty



In this situation, the behavior of the cache in case of miss penalty is way better from the processor point of view. However the cache is unable to solve two misses at the same time. Again, percentage of hit/miss has a big impact on the performance, and this changes when changing program structure. In many programs this organization is useless, for example if they use very often the same value (time locality). Again, features of microprocessor, compiler and such are important. But the compiler doesn't know the cache organization usually.

Let's talk about compiler. The compiler at compile time don't know how the memory of device is organized! But we have to provide something because we need to improve organization of our compiled software section! This is done in embedded systems. Compiler are "personalized" for an architecture because is fundamental to use every resource at its 100%.

Multi Banked caches

Another way is to organize cache as independent banks to support simultaneous accesses. The most used processors (either for embedded and general purpose) organize caches in banks, storing subsequent blocks in different banks. This is used in order to obtain more data in a single transaction. This solution is used in main memory too! In fact the memory is divided in different bank on different channel. Or in DBMS the data is distributed on some disks in order to read/write in parallel way to reduce the latency.

- Interleave banks according to block address

Block address	Bank 0	Block address	Bank 1	Block address	Bank 2	Block address	Bank 3
0		1		2		3	
4		5		6		7	
8		9		10		11	
12		13		14		15	

– ARM Cortex-A8 supports 1-4 banks for L2
– Intel i7 supports 4 banks for L1 and 8 banks for L2

Compiler Optimizations

Introduction

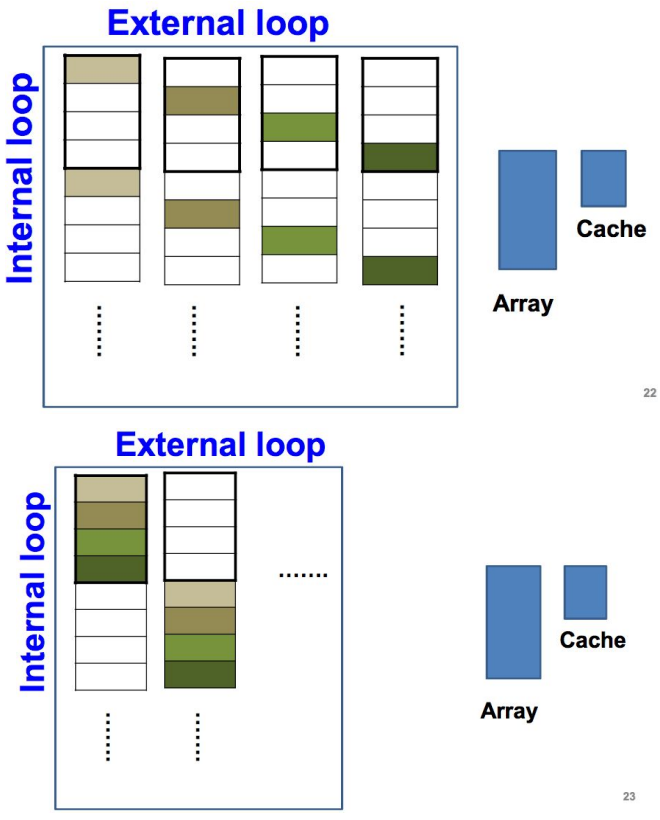
In the same time, we can consider also the software tools used to organize the program, reducing the access time to memory. The compiler can do his best to optimize compiled code. Two of most used solution regard

- Loop interchange
    - swap nested loops to access memory in sequential order
- Blocking
    - Instead of accessing entire rows or columns, subdivide matrices into blocks

The ways in which an array can be organized can be vertical or horizontal. The compiler can solve the problem copying the array in another array and then use the second one to access it in order to have a good sequence of access in cache, to minimize number of misses. For example, a specific program could explore a multidimensional array in such a way that each access will lead to a miss, but the compiler could produce an additional loop on which the array is organized in a different way, in order to minimize the number of misses later, grouping things accessed one after the other together. The only cost is the construction of the additional array. This could reduce also the number of cache blocks used by the loops. If some data won't be used in a loop, the number of cache blocks used can be reduced grouping stuff together as said, reducing also miss rate of other parts of the program. In few words, in blocking approach, instead of accessing entire rows or columns, matrices are subdivided into blocks. This requires more memory accesses but improves locality of accesses. But let's try to analyze in deep the two different solution.

Loop interchange

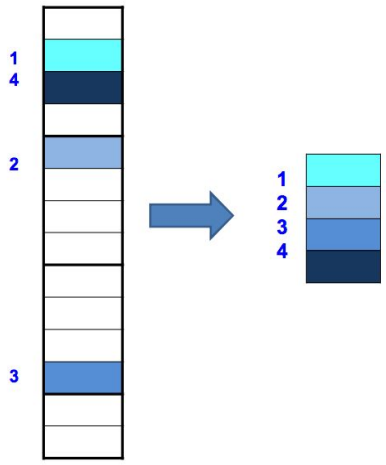
In loop interchange approach, nested loops are swapped to access memory in sequential order. Using loop interchange reduces access time, because the cache can store instructions for the loop and the data on which they operate. Let's see an example.



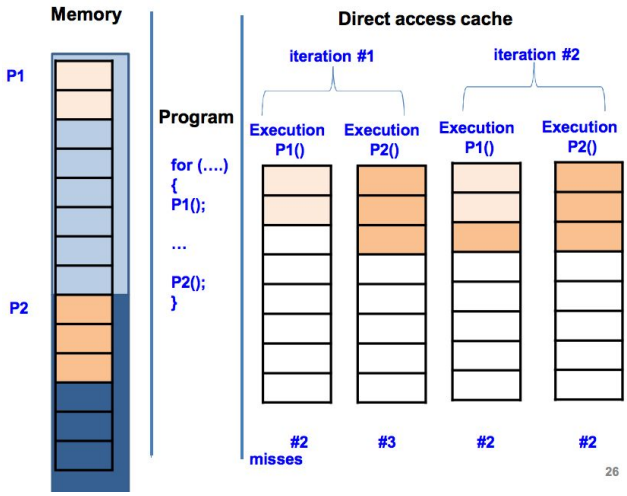
Here a 4 access miss become only 1 miss! And this at 0 cost. The only thing is that at compile time compiler must know we can do this thing. Also memory is saved.

Blocking approach

The second example instead is for blocking approach. Let's see how to improve the **temporal locality**. The programmer often declare variable and the OS use the stack in order to organize local variables of a procedure. This organization usually is randomic, by considering the definition of variable. But the compiler know where the variable ar declared (and stored in memory) and where it are used. Hence the compiler can re-group the variable in different way. If the compiler is able to know the typical sequence of access to local variables, it may organize the variables used more often in the same block, thus reducing the number of misses. Changing the virtual allocation of the stack reduces number of blocks that have to be accessed, reducing cache usage and misses. Let's see how with an example!



In this way instead of 4 cache block, only 1 is used!
Another example of compilation optimization for conflict miss. Conflict misses are very important in small caches or caches with a small number of ways.



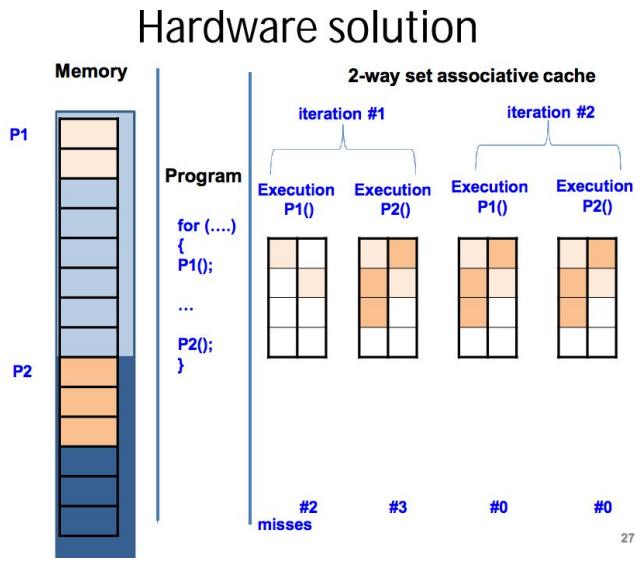
In different executions (of the same program) the program could ask for blocks in different pages of memory which should actually be stored in the same cache block (due to the definition of the direct access cache for example). In fact, let's suppose we have a program with a for and two different sub-program. With two associative cache in the second execution of the for, the cache block are overlapped! And is lost time for nothing, only because the software is organized in a bad way. The problem is that here we know the flow path but if don't know the flow path (depend from input!) how can we organize our problem to know it if behave well?

Let's recap a bit, in general software solution have these problems:

- the compiler has to analyze the software structure to reorganize it
- we considered the execution path of a specific program must be known, but it depends on the input of a program, so more than one input should be considered, to obtain the best solution for different inputs.

Usually, RT software needs to have a specific maximum execution time. In high performance solutions, the software needs the best execution time. Again the problem can be solved in hardware increasing the number of ways of a caches. So the number of ways needed for a specific program is linked to the number of locality areas inside the main memory for that program.

For this example solution can be two, changing the cache structure or change the organization of the routine.



In general n-way associative performance is linked to the way software is organized in section/segment. For example our software can be separated in three .text segment (one for main function, one for subroutine, one for system calls), and in three .data segment (one for global variables, one for local variables, one for kernel variable).

Hence locality areas in the code can be (in a small interval of time):

- main program

- routines and functions
- system calls

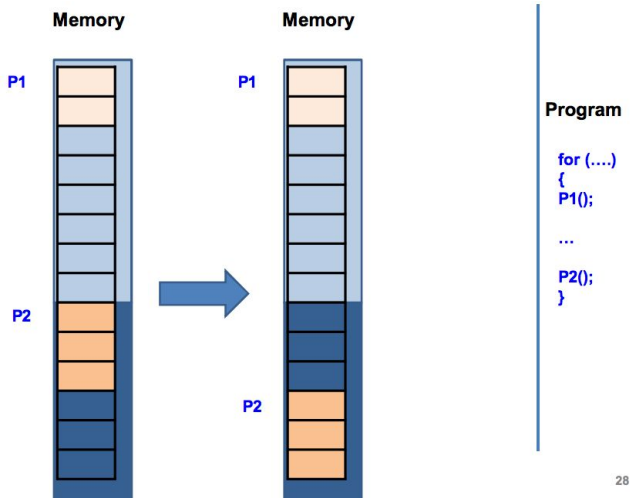
Locality areas in the data can be:
- global data
- local data
- data used by the kernel

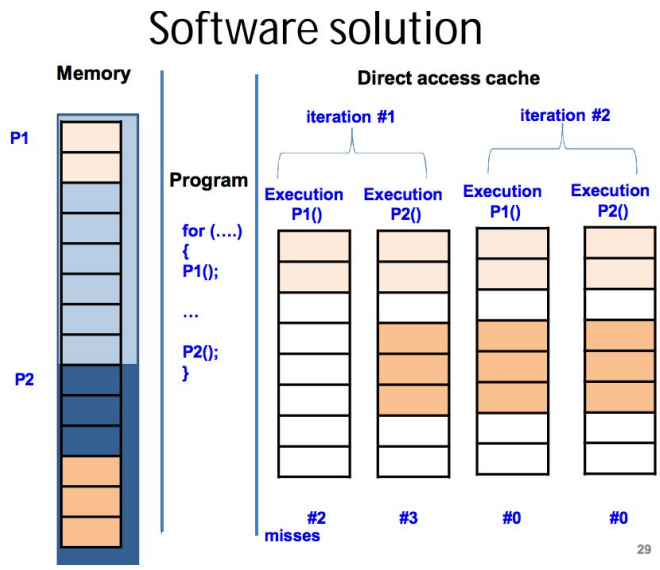
In this perspective the 6-way associative cache can help to reach the maximum number of hit! It seems easy but if we have some processor? It depends of the number of thread/processor. But if thread are from the same process (and so share data and code) it's perfect! The hit miss can reach high level on all the processor! In general, 4 ways associative cache is a good solution, 8 way is an excess for a single program.

The best solution is the one with 4 processors, each of them with 4 threads. Each thread has a different stack but shares data and code. So another important consideration is that the solution with thread will maximize the number of localities used by a program (because threads share resources like code and data).

In general the number of localities needed depends on the number of processors and number of threads per processor at the same time.

Now let's see instead the software solution. The solution work in way to reorganize the layout of a program (in main memory) so that conflicts doesn't happen in short times (blocks used often subsequently do not overlap). First we can change the organization of our function environment in stack. Changing the organization, this reflected on the cache organization! Perfect. This could reduce drastically the number of misses for subsequent loop executions.





It seems software solution it's better. Same performance without changing hardware. But it is only an illusion! Because the flow of execution depends on input and the organization of memory depends on program and compiler must know the organization.
All this factor often are not known.

The hardware strategy is better, we're sure that work properly.

ARM-Cortex A8

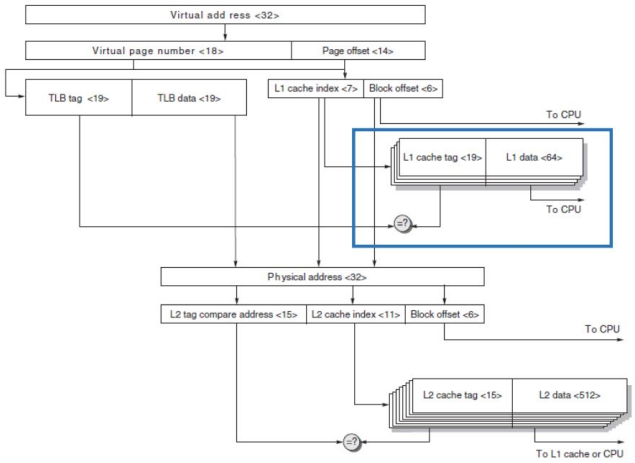
Now let's focus on ARM Cortex A8 architecture organization. We start from cache, that maybe is the most important feature. Consider that often in a microprocessor almost the 80% of power consumption is spent on cache management, namely to reduce memory latency!

The cache is organized as a two-level cache hierarchy using virtually indexed caches. The first level is organized in a pair of caches (instruction and data).

- The Cortex-A8 is a configurable core that supports the ARMv7 instruction set architecture
- Cortex-A8 IP core is used in the Apple iPad and smartphones by several manufacturers including Motorola and Samsung.
- Two-level cache hierarchy with the first level being a pair of caches (for I & D), each 16 KB or 32 KB.
- The optional second-level cache when present is eight-way set associative and can be configured with 128 KB up to 1 MB

We can have different model of the same processor based on different features and size of the cache inside. We can have the second level of cache that can vary according to the

usage in embedded problem. In embedded decisions it is very strategic the number of hardware component we use to reduce the cost and the power consumption. The organization is strategic in order to reduce these things. The 80% of the power consumption is due to cache because it is composed by very high speed element.



At the beginning there is the virtual address. The translation is based on an additional cache. The virtual page number is used to the cache of all the last translation done before (as a traditional cache). This cache will offer the translation or it produces a miss and we need to go in the table in order to go to the line and store it in the cache.

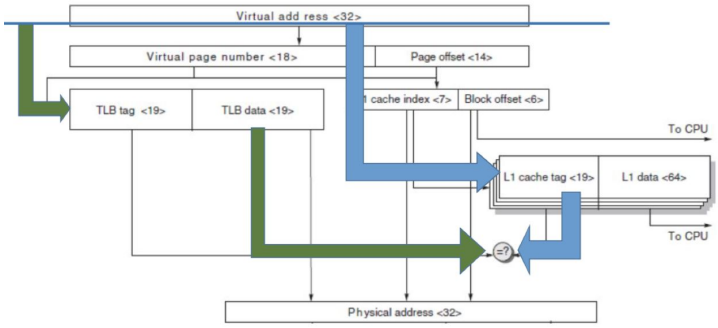
It is complex to use virtual address in real-time applications when we need to go to the tables. The first level of cache is accessed by the page offset, a miss can ask to access the second level and if there is a miss we need to go in the main memory. I want to minimize the access time. In general the idea is that at the first step of each operation we need to translate the address and then we enter in the caches.

The best solution I will use the translation only on case of the last cache. So the caches will use only virtual address. In this case a hit doesn't require a translation. Every time we have a context which we need to destroy the cache to be sure. We don't use this solution because there is the possibility that more virtual addresses can link to same area, so in terms of cache we need to detect this situation. I need to offer 2 positions in cache with same copy for the same shared areas. For example the area for the same kernel process.

There is a way to access to the cache and to translate the address by checking if the tag stored in the cache is the same of the tag translated by considering the same . We can use to access the cache the part of the virtual address that does not need a translation. The translation is done only to the page part, the offset is the same. We use the offset to access the cache and we use the other part . This operation can be done if the size of the virtual page is equal to the size of the cache. A virtual page is about 4 Kb, so the first cache level is 4 Kb blocks. A cache of 4Kb of blocks is very small, not a good idea.

We can use the set associative solution, for example in case of a page of 4Kb we can address to the offset and a cache organised with 2-ways will have 8Kb. I can have 16 Kb of cache in case of a 4-way set associative cache.

I can increase the offset by asking to the manger of the virtual address to have an additional bit (the last one) that is equal in physical and virtual address (in Intel we use this solution to increase the size of virtual page). 2 subsequent virtual pages will have the same address and so I can use the same virtual address for 2 physical address.



We use the virtual offset and then we use the virtual page id to obtain the physical address.

The first level of cache is used for data and cache of the processor. In parallel there is the translation in 2 steps, in the first there is a small cache and a miss in the first level means an access in the second level (2 levels of caches for the translation). The second level is composed by data and code of the same processor. The third is for all the processors. The delay due to the wire is strategic, it is impossible to increase the clock cycle but the solution is to offer a cache subsystem to have more used blocks near.

ARM microprocessor are characterized by 2 levels of caches. The first levels is used to store code and data. The second level can have different possible size (128Kb to 1Mb) so we can organize the cache according the specific application.

The virtual address is 32 bit and 14 bits for the offset and the rest for virtual page. The offset is used to access to the cache by using 7 bits to address the cache, 6 bit as offset of the block. If the final result is miss I can read the block in the second block. The second level is shared among instruction and data. In case of miss in second level the cache will read the block directly in main memory.

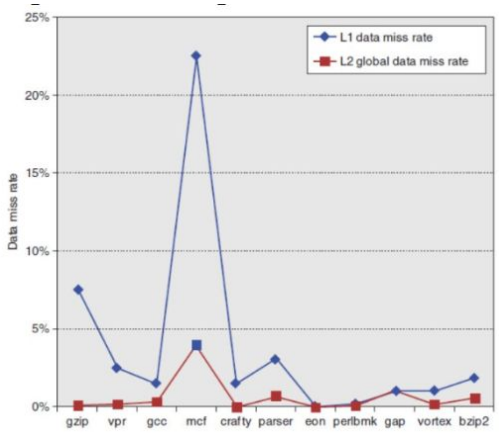
We can access at the same time the TLB and the cache and then we can compare the tag produced by TLB and cache. In this way we can understand if the information in the cache is right or not (remember that in cache we have the physical addresses).

The access to the cache has to be performed by using the bits from the offset of the virtual address. So in this case by using a 4 ways associative cache we can have a size fro the

cache of 8Kb for each ways. In general a page size of the virtual address must be greater than or at least equal to the size of the cache way (given by the cache size divided by the number of ways).

I will have 2 caches in the first level, one operating on the code request and another to the data. Of course these two sequence of these caches can have different features in terms of locality. So I can have different feature for these 2 caches. In general the code will have a sequential localities while the data temporal and spatial locality, so the size can be very different.

The miss rate depends on a specific program. Let's see the following performances:



The designer must organize the system in order to avoid the overlap these group of misses. A cache can produce a non-regular track even if it receives regular requests.
We can consider also the average access penalty, that is the additional clock cycle needed to compute the operation. Each penalty is divided in 2 parts, one due to the first level and the other due to the second. The access to the first level is low while the access to the second is very high.

The idea to minimize the pipeline penalty is to use the branch prediction. The penalty of the first level because there is mechanism to reduce this penalty, when a miss appends a CPU must wait in order to have the data of the code but if we use the prefetch we can minimize it, because when a miss will happen the machine has100 instruction ready to be executed. This is good for the code. But for the data is to move a load operation before at the point in which the data is used. The load operation of the data in register can be computed before, so the compiler translate the program in order to move the load operation before. But I need and I can move the operation if the new position doesn't create effects on the execution (the load operation reads the data, stores into a register and changes the flags) so this operation can destroy the current status of the flags during the execution of the program, the problem is to add a new operation load used to load the data without change the flags. So we can have 2 different kind of loads. The second level produces penalty that are relevant.
The core of ARM is an asynchronous sequential machine. The synchronous machine has the clock and the input and output are valid only in a portion of time. The clock is an input for

ARM so we can turn off and we can cut the cost of energy due to the dynamic operations due to the clock. In traditional system the idle is obtained by a loop.

Intel i7

The TLB is organized in 2 levels so I miss in the first level can be resolved by accessing the second level. A miss in the second level means to go into memory. The first level is divided into 2 parts for the data and the code:

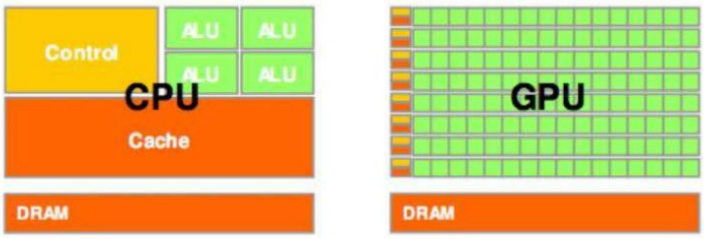
We have 3 levels of cache memory. The first level of cache contains data in 4 ways and data in 8 ways. A second level cache contains both instruction and data but they are not divided. The last level cache is shared among all the processors. We have 2 TLB, the second is shared among data and code. The last level must solve the need of all the CPU inside the core-

The processor ask for address, the cache is sending the data and in the same time the processor ask the address of the next operation (so we have at the same time address of the next operation with the data of the current operations).

In the current CPU, the instructions are not executed in order and the process is called out-of-order execution.

Lesson 07 - 15/03/2017

Today in the lesson will talk about the GPU. Meanwhile, why is needed a GPU? Can we use CPU to manage video and monitor? The main difference is that the CPU is designed to work with sequential memory organization.



While the CPU is optimized for sequential operations, GPUs are designed in order to execute typically many parallel operations. GPUs have many cores, because in order to handle graphics the operations that are typically performed are simple and are executed on a heavy load of data. So the architecture of the GPU is designed in order to meet this needs.

The design of a **CPU** is optimized for sequential code performance

- Out-of-order execution, branch prediction
- Large cache memories to reduce latency in memory access
- Multi core

The **GPU** instead

- Many core
- Massive floating point computation for video games
- Much larger bandwidth in memory access
- No branch prediction or too much control logic: just compute

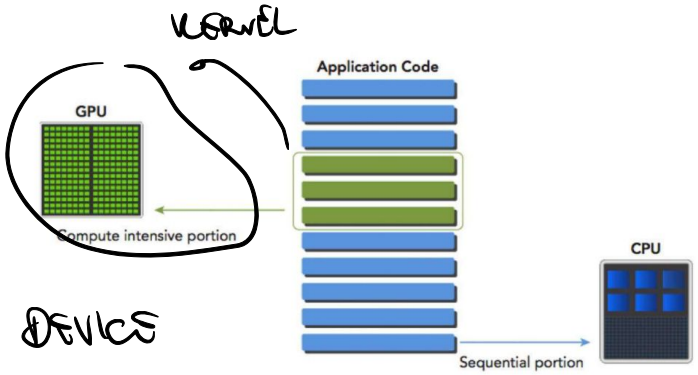
In fact the video handling doesn't need to perform control logic or branches, there are only lot of work that must be done in parallel way!

- GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well;
- One should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.
- We are going to deal with **heterogenous architectures**: CPUs + GPUs.

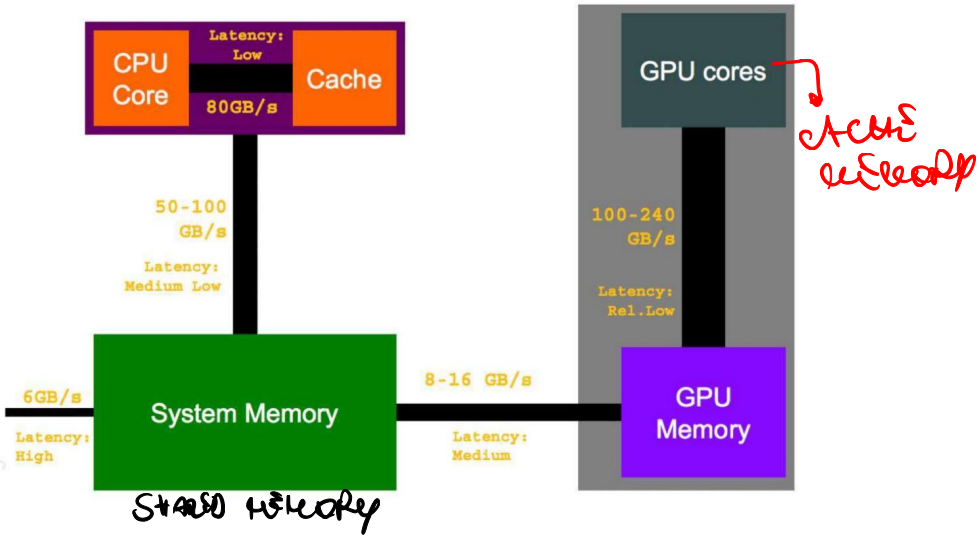
In Game developing, typically the floating point computations and parallel algorithms are executed on the GPU architecture, while the other sequential part of the software is performed by the CPU. This kind of programming style is called **Heterogeneous Computing**. In this point of view, the application is divided in two parts:

- Host Code - executed on CPU
- Device Code - executed on the GPU

CPU computing is good for control-intensive tasks, and GPU computing is good for data-parallel computation-intensive tasks. The CPU is optimized for dynamic workloads, marked by short sequences of computational operations and unpredictable control flow. GPUs aim at workloads that are dominated by computational tasks with simple control flow.



The latency is low and the speed is fast when accessing the cache from the CPU core. If the CPU has to acquire data from memory the latency is not really good (medium low), while the speed is not so bad.



In GPU system, we have low speed and low latency! Why this is needed? Why we need an higher value of speed using the same technology? In GPU systems, the speed at which GPU cores can acquire data from the GPU memory at a faster speed and a lower latency than related to a CPU-cache system. This because in the GPU we have a lot of cores, so

also a lot of caches. The speed of a single cache is multiplied by the number of caches in order to obtain the global speed, so more caches more speed.

If you perform access to GPU memory using a coalition access, you take advantage of the architecture of the GPU, using many threads to operate on many sequential data at the same time.

The very easy idea to increase speed in computing is to increase the number of processor. But the problem again is, more processors require more memory bandwidth. It's difficult nowadays to increase the memory bandwidth. Thus to minimize the require of memory cache are introduced. But this is not enough! The next step is to have a private channel between the memory and each processor. But each processor has to execute on different data not on the same data because data in this way is not consistence.

This is a problem of organization! Then the idea is same code of software is shared among processor and different data. The system is organized in this way, some processor share the same code. It is hard to design an application for which many processors use the same instruction at the same time. An example of applications like these are graphic applications.

Then we can use an heterogeneous solution in which we have a set of processor that share instructions and a set of processors that can share the same code (the idea is that threads the execute on the CPU share the instructions and they have to execute exactly that operations, while on the GPU every thread can do what he wants, each execution is independent).

Some processor will share the same instruction, but a set of these groups can share the same code. At the same time we can have the a processor has local variables and all the processor will share all the data in a main share memory. In this way we can obtain a parallel execution by asking a very low need of memory location. At the end, we need to look for the possible algorithm to organize it in a parallel way and we need to be able to split the global processing in a set of global threads in which some share the same instruction and others that share the same code.

We must be able to split the global processing in threads, by minimizing the communication between threads on the same processor and also minimizing the bandwidth that belong to the same group.

At the begin, a GPU was able only to perform only the graphical processes. Nowadays a GPU is a parallel machine inside the machine and in some case it is used also alone in embedded solution. On the market we have different solution because we can change as we want the architecture by considering process power, cost and so on.

In GPU architecture the application can be written in c, CUDA or what we want but it's created a new executable format (ptx) that is interpreted as a script by the GPU driver. This solution is useful in order to change the architecture of the GPU maintaining the compatibility on previous applications (that is not possible in CPU).

We have registers, caches private and shared for each core. There is also a special unit to execute some particular mathematical operation needed in graphic.

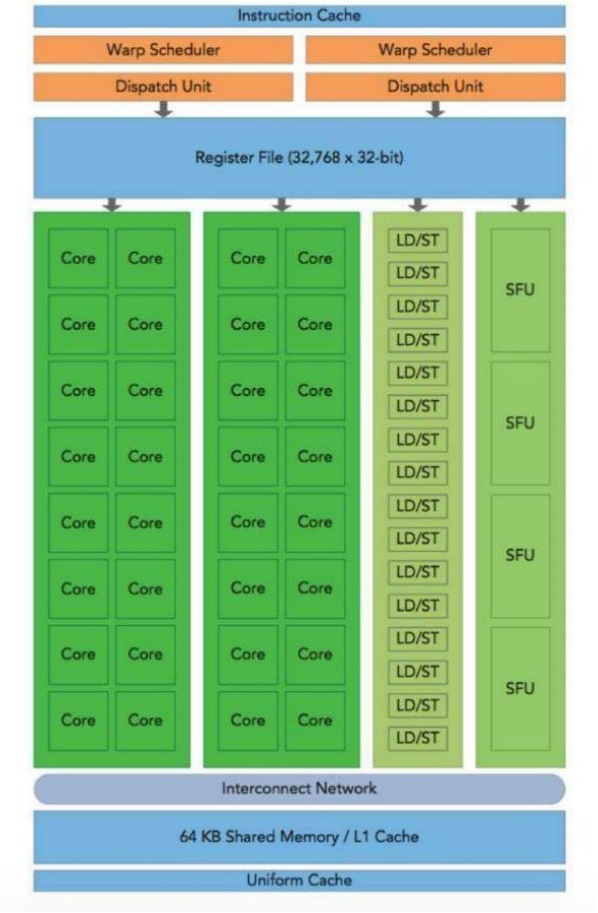
A CPU is optimized to execute a sequential code, so we have 2 cores with about 2 GHz. These cores in GPU work in frequency that order of the MHz. This is due to the fact that when we have lots of core we need to consider the power consumption. Generally in GPU system we need to use efficient cooling technique.

A thread is lighter because when we have context switch we don't' need to invalidate TLB or cache because they share memory. In GPU threads are lighter again because they don't do a lot of function, for example we don't need synchronization operations.

GPU Architecture overview

The architectures in years is changed and on the market there are very different solution talking about architecture in order to meet requirements of power consumption, parallelism, performance and so on. Is important to understand how we can change an architecture.

In GPU architecture, the application are in a new executable format, written in CUDA (file of extension PLX) and interpreted by the GPU (not compiled). This solution is good to maintain an abstraction between software and hardware and to port the code for other GPUs.



Here we can see the architecture of a NVIDIA GPU. The GPU is built around a scalable array of streaming processor (lot of cores). Why this solution is not implemented in CPU? Why in CPU the core are only 4 or maximum 8? At first sight seems GPU it's better and faster than CPU. This is completely false! Core of CPU work at high frequency such as 3 Ghz for each core. In GPU cores work at Mhz! The architecture that implement the high frequency is very expensive from the point of view of power and temperature. Thus for many cores it is impossible to implement an high frequency.

Let's return to organization of our software, we want to split program and memory data in smaller piece. The very good thing in organization of our software is the thread idea. But what are **Threads**?

Usually, GPU threads don't have many branches, many synchronization needs and so on.

> Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off CPU execution channels to provide multithreading capability. Context switches are slow and expensive.
>
> We deal with a few tens of threads per CPU, depending on HyperThreading.
>
> Threads on GPUs are extremely lightweight. In a typical system, thousands of threads are queued up for work. If the GPU must wait on one group of threads, it simply begins executing work on another.
>
> We deal with tens of thousands of threads per GPU.

Flynn taxonomy

Flynn in '60 divided the possible architecture based on the number of streaming instruction and the quantity of data handled.

	SI (Single Instruction)	MI (Multiple Instruction)
SD (Single Data)	SISD	MISD
MD (Multiple Data)	SIMD	MIMD

The most used architecture until now is the SISD one. A single instruction at time that work on a single cell of data. The MISD architecture doesn't exist in real, has not sense to work with a multiple instruction on the same data. The one that interest us now is the MIMD! We want to work with parallel instruction in the same time and clearly work on some data!

SISD Architecture

In SISD architecture,

SIMD Architecture

SIMD typically implement a lot of logical units and handle memory in a parallel way. Differently from SISD, SIMD needs only:

- 1 clock cycle to fetch and decode the instruction
- x clock cycle for data fetch
- 1 clock cycle for execution (on multiple data)
- x clock cycles for write back the result

The memory accesses are executed in parallel, which are way faster than SISD.

The first commercial CPU with SIMD instruction was presented in 1997.

- 1997: Pentium Pro MMX (Multimedia Extension)
- 1999: Pentium III with SSE x86 support
- 2001: SSE2
- 2004: SSE3
- 2006: SSE4

In order to manage these architecture, the instructions in machine code are generated by the compiler, so the compiler should be able to recognize some patterns which could be executed with the SIMD instruction set (instead of SISD) and translate them using it.

The multimedia extension was presented as an instruction support to multiple data handling. Hence we need new instruction set to implement this type of architecture and for example in case of SIMD architecture, MATLAB provide us different things.

- GPU is a SIMD (Single Instruction, Multiple Data) device → it works on “streams” of data
  - Each “GPU thread” executes one general instruction on the stream of data that the GPU is assigned to process
  - NVIDIA calls this model SIMT (single instruction multiple thread)

NVIDIA provide a new one architecture idea, the SIMT. SIMT means single instruction, multiple **threads**). The SIMT architecture is similar to the SIMD one. Both implement parallelism broadcasting same instruction on multiple execution units.

The SIMT architecture is similar to SIMD. Both implement parallelism by broadcasting the same instruction to multiple execution units.

A key difference is that SIMD requires that all vector elements in a vector execute together in a unified synchronous group, whereas SIMT allows multiple threads in the same group to execute independently.

However, in SIMT multiple threads in the same group can execute independently, not in a “synchronized” way sharing the single instruction that has to be executed.

- The SIMT model includes three key features that SIMD does not:
- Each thread has its own instruction address counter.
- Each thread has its own register state.
- Each thread can have an independent execution path.

Note that in SIMD there cannot be exceptions inside these loops or parallel operations, while in SIMT each thread has own instruction counter and independent register states and execution path. Performance decrease but it is possible.

Cuda language

CUDA language is an abstraction of GPU that is provided by NVIDIA. It’s a scripting language to write script that the GPU execute.

- It enables a general purpose programming model on NVIDIA GPUs. Current CUDA SDK is 8.0.
- Enables explicit GPU memory management
- The GPU is viewed as a compute **device** that:
  - Is a co-processor to the CPU (or **host**)
  - Has its own DRAM (global memory in CUDA parlance)

    Runs many threads in parallel

CUDA languages introduce a lot of innovation in classical C language. For example local function, basic synchronization function for threads (similar to barrier), a lot of new keyword and some runtime API in order to manage GPU memory.

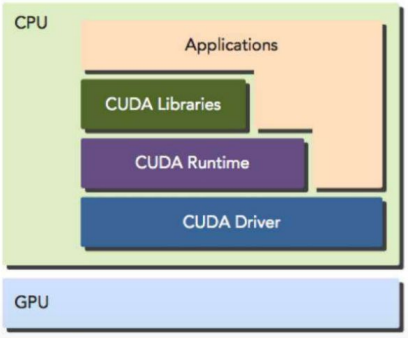
- The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives, application programming interfaces, and extensions to industry-standard programming languages, including C, C++, Fortran, and Python

- CUDA C is an extension of standard ANSI C with a handful of language extensions to enable heterogeneous programming, and also straightforward APIs to manage devices, memory, and other tasks.

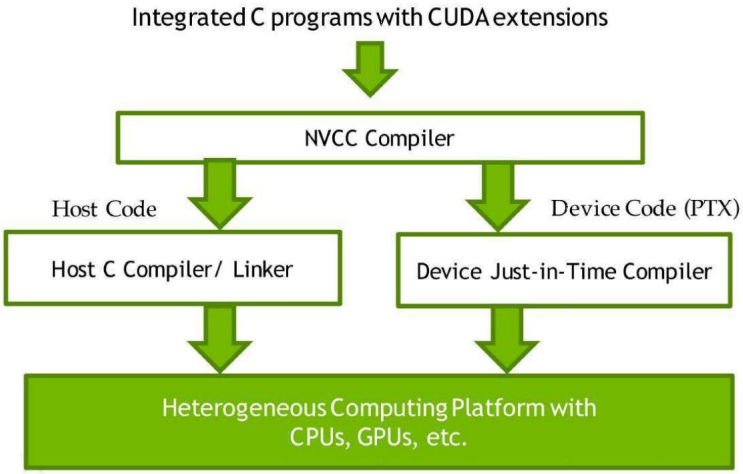
So we can use it only on NVIDIA. We have CUDA API that are divided in 2 levels:

- Driver API (require knowledge about GPU architecture)
- Runtime API

- CUDA provides two API levels for managing the GPU device and organizing threads:
  - CUDA Driver API
  - CUDA Runtime API
- The driver API is a low-level API and is relatively hard to program, but it provides more control over how the GPU device is used.
  The runtime API is a higher-level API implemented on top of the driver API. Each function of the runtime API is broken down into more basic operations issued to the driver API.



It is a mixed of host code and device code. The nvcc compiler divides the code and produce ptx instruction interpreted by CUDA driver at runtime.



It introduces lots of extensions to the C language. The function can be device, shared, global and so on. We can introduce keywords and we have . Synchronization means that when we have a thread that calls the execution until all the threads the share the same operation calls it.

The design of a program written in CUDA must follow this path:

- Allocate GPU memories
- Copy data from CPU to GPU memory
- Invoke the CUDA functions (called **kernel**) to perform program-specific computation
- Copy data back from GPU memory to CPU memory
- Destroy GPU memory

The copy from CPU to GPU and vice versa is very expensive, so it must be done only when is really needed and the best approach is that the program must overlap the initial execution of the program with the transfer phase.

CPU and GPU can be in overlapped execution. There is a synchronization between them to know when the GPU finishes its execution. All threads generated by a kernel during an invocation are called collectively a grid (the programmer can organize as he wants a grid).

Typical application that use this type of “architecture” are the program that process continuous flows of data.

Cuda functions

We can 3 different kind of functions:

- **Device function**
  - A device function is called by GPU and executed by the GPU.
- **Global function**
  - The global functions are called by the CPU but executed on the GPU and so we need a void function (a sort of remote procedure call).
- **Host function**

An host function is called by CPU and executed by the CPU. And also this type of programming language is useful not only for GPU

This is a simple hello world for CUDA.

```
__global__ void helloFromGPU() {
    printf("Hello World from GPU thread %d!\n", threadIdx);
}

int main(int argc, char **argv) {
    greet(std::string("Pinco"));

    helloFromGPU<<<1, 10>>>();

    // destroy and clean up all resources associated with current device
    //                                                  + current process.
    cudaDeviceReset(); // CUDA functions are async...
                       // the program would terminate before CUDA kernel prints
    return 0;
}
```

10 CUDA threads running on the GPU.
This uses is 1 grid.

These are the CUDA C language to allocate memory. It's like the c malloc!

- The CUDA programming model assumes a system composed of a host and a device, each with its own separate memory.
- Kernels operate out of device memory. To allow you to have full control and achieve the best performance, the CUDA runtime provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory.

Standard C function	CUDA C function
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

The device memory architecture is difference between CUDA and C. When an host invoke the kernel, it is created a grid in a block! For each block we can have tridimensional group of thread.

Sum for a vector. In classical sequential approach, the sum is repeated from 0 to N-1 elements of the vector. In GPU parallel approach, there are N threads that sum only one value!

- CPU:

```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
    for (int idx = 0; idx < N; idx++)
        C[idx] = A[idx] + B[idx];
}
```

- GPU

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C,
const int N) {
    int i = threadIdx.x;
    if (i < N)
      C[i] = A[i] + B[i];
}
```

This is completely different from sequential approach! Also

> Supposing a vector with the length of 32 elements, you can invoke the kernel with 32 threads as follows:
>
> ```
> sumArraysOnGPU<<<1, 32>>>(float *A, float *B, float *C, 32);
> ```

Organizing Threads

When a host calls a kernel in the CPU it creates a grid of blocks. Each block can be bidimensional (2 coordinates of threads) or three dimensional.

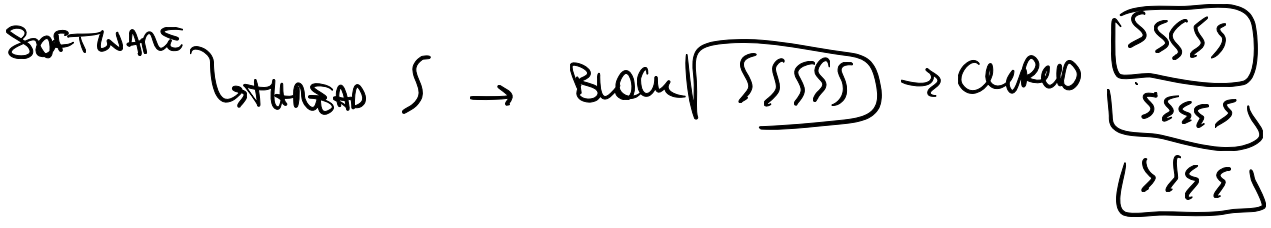
To know the position of a thread in a block and the position of the block in a grid we can use the keywords blockIdx.x, blockIdx.y and so on.

Using Threads

It's really common that threads on a cpu we have threads that read addresses in the blocks using their own addresses (first first, second second), this is called coalition access. The architecture of the GPU is optimized to execute these accesses faster and simultaneously, implementing a sort of network where the first address of the block is linked to the first cache of the cores, the second with the second and so on.

In these networks, all the reads can be performed in a few clock cycles, much less than the number that would be actually needed without it.

If some threads want to read elements in positions different from the previous defined one, the network doesn't support that access type so its performance will be much less.



Lesson 08 - 17/03/2017

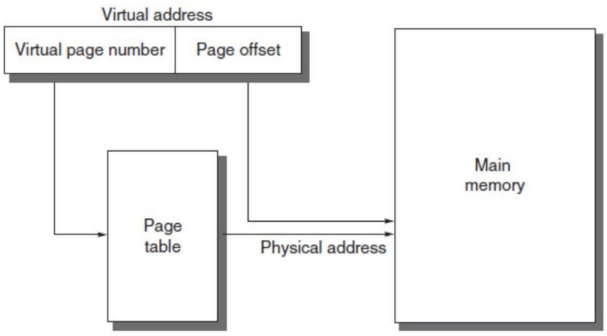
Virtual memory management

Today we will speak about the virtual memory management. Virtual memory allows to manage the memory on which a specific process is located, while the real process is allocated in physical memory.There must be a component which translates virtual addresses in physical addresses (MMU). This part of the system it is needed to translate a virtual address that comes from CPU to a physical address in memory.

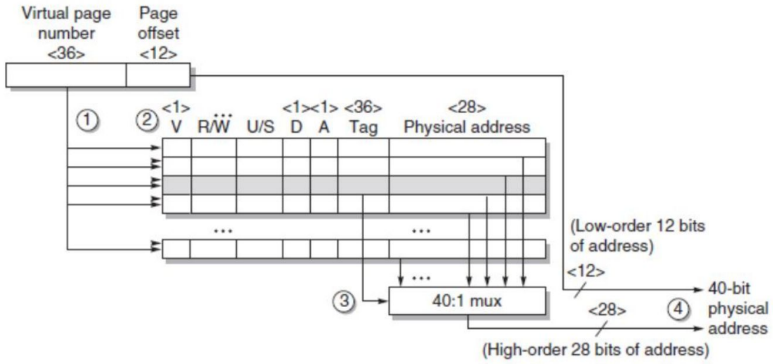
Why are we need a virtual memory? The strategic role of the virtual memory (and of the virtual memory management):

- There can be more processes that use the same program on different data, with same virtual addresses but different physical addresses. The program code and variables are saved in the HDD and in this way the RAM provide a sort of cache for HDD. There is also the possibility to share the code between one or more process.
- The space needed for all the process is bigger than the size of the physical memory, so all the processes are on the disk and some parts of these processes are copied in physical memory (cache). In order to manage virtual memory, each processor space is splitted in pages, so that only some pages are located in physical memory, while the others are stored in physical memory on demand. When there is no more space, pages have to be copied back into disk, overwriting them.
- Allow a process to share some memory space to the kernel, in order to send or receive messages, files and so on. The kernel and the processor will access to the same space using 2 different memory addresses, in this way you can also specify the possible operations a processor can or cannot execute on such space. The memory management supports a mechanism that control the operations on a specific address. Having more than one address specifies more than one way to access to the same physical space.

For translation however a table of translation is needed, if the table is not present how can we translate a virtual address into a physical address? Actually we need a table **for each process**, in fact we said that the virtual address of each process can be different from the virtual space of another. In the address table for the translation, there is stored information about the page on the disk that has to be replaced when popping out a page from physical memory. The table can be very big, because we need a register and a descriptor for each virtual page for all the processes that are in the system in a moment, again it is stored on the disk and loaded on demand on the physical memory.



To implement this translation we need an hardware support, this support is provided by a mechanism that derive from the old idea of MMU: the TLB.
In the figure below we can see that the virtual address is divided into two different part, one is used as offset and the other (the most significant one) to access the table of page.



The virtual address can be considered organized in 2 parts:

1. virtual page number - relative to the page table (which performs the translation)
2. page offset

On the TLB there are the latest translations:

- the physical address of the page
- the tag (as a cache, for checking the virtual page number)
- data about the access policies (V, RW, US, D, A)

How work in ARM Cortex architecture the idea of virtual memory and also how the cache is organized?

ARM Cortex microprocessors

The first level cache is one for instruction and one for data, the second level (when present) can have different possible sizes (from 128KB to 1MB). This part can be “configured” when choosing processor for the specific application you need to execute.

Then the virtual address is 32 bits, and the [...]

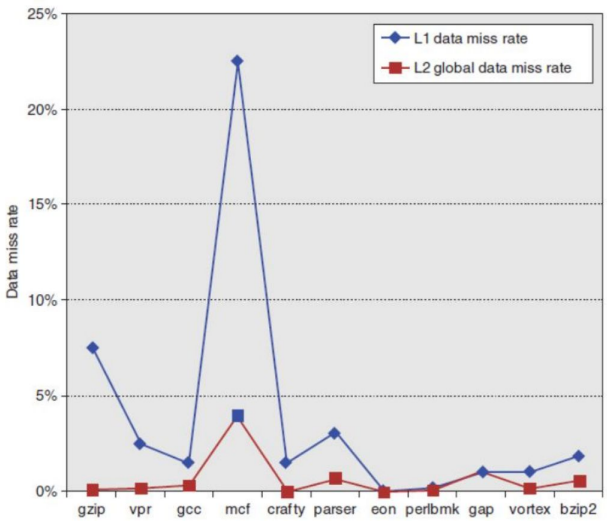
The offset is used (7 bits) to access the cache, 6 bit as offset of the block (?). If there is a miss on the first block I can read it from the second level (if present).

The procedure to access to cache memory accesses the cache and TLB in the same time and then checks if there is a hit, comparing the tag field from the cache to the tag field in the TLB. This allows to minimize the time needed to access a specific address.

The access to the cache checks only the bits from the offset field, the TLB checks the others. If both have an hit, the address we wanted is stored on the cache.

The page size for the virtual address must be equal to the size of the cache way size.

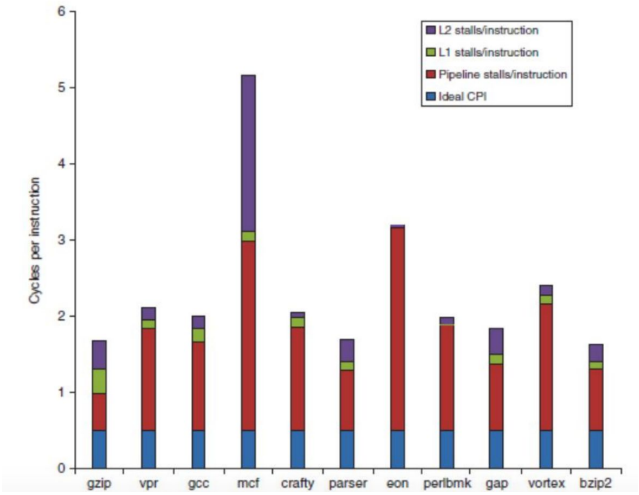
It is important to have 2 caches operating on the first level, one for the code and one for the data. These two sequences of addresses produced can have different features of localities, so in general these two caches will have different features. For the data spatial and temporal locality apply, while for the code usually it is more a thing of spatial locality.



This picture shows how the performance of level 1 and level 2 cache performance in an ARM architecture are affected by the algorithm used to do a work (I think for compress data!). We can see that for example there is a peak using the mcf in the level 1 cache.

Now let's talk about the bandwidth required by the CPU. The cache modifies the bandwidth required by the CPU. This number in fact is irregular in time. For example, when a program i started, surely there is a cache miss (that will be translated in a high memory bandwidth require). This can be extended for example when subroutine are called or when kernel function are called. Instead, in a loop, the cache memory have already collected the memory required. This is translated in a different bandwidth requirement in time.

We already know the CPI, there is the number of clock to perform an instruction. The idea in history is: one instruction, one cycle of clock. With cache memory this is no more true! Let's see an example.



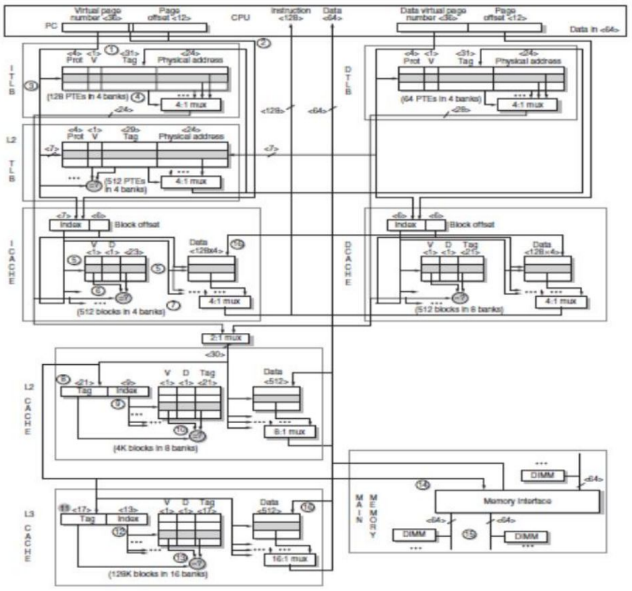
When the pipeline is empty, for example due to JMP, produce in a machine, in general, this penalty. Clearly the total penalty depends on what program do in its computation. The hardware way to reduce the total penalty is to implement a mechanism of branch prediction! The miss value on the L1 of cache is not important because there is a simple mechanism that can be used. The prefetch mechanism! And on the data? There is a way to reduce the quantity of operation needed in order to read data from memory? The way is to move a load operation before at the point of which at the data will be produced. The operation in fact is moved before.

This can introduce some problems, some registers are needed to manage this thing. I can move the operation only if the new position doesn't create effect on the execution. Do you know how this can affect the execution?

We have to reason about how the data are loaded from memory. The load operation not only carry data from memory to register but modifies the flag status. This can destroy all the program execution. We need to introduce another instruction: the LOAD without flag modify.

The Intel Core i7

The first important situation for this microprocessor is that the TLB is organized in two levels. A miss in the first level is managed accessing the second level. Obviously a miss in the second level is resolved with a memory request.



To have an idea of the organization, is very very complex! For each processor I have two first level cache, one for code and one for data. The last level of cache is shared among all processor.

Now let's see some features.

Characteristic	L1	L2	L3
Size	32 KB I/32 KB D	256 KB	2 MB per core
Associativity	4-way I/8-way D	8-way	16-way
Access latency	4 cycles, pipelined	10 cycles	35 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorihtm

The processor ask for address, the cache is sending the data, in the same time the processor will send the address of the next access. On the BUS in the same time is present the address of the next operation and the data. In this way I can overlap the address with the data. Remember this is the pipeline mode.

Lesson 09 - 22/03/2017

Cuda Lab

Introduction

Let's start this CUDA lab. We will use first an already written file to analyze some of basic API of CUDA. The first part of file regard the library required to execute the C code. The first, "cuda_runtime" is the one that permit to use the function provided by CUDA.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
```

The function "**addWithCuda**" is the function deputed on take variable from RAM and transfer on GPU memory, add values to each other and return the result. For add values are launched some thread in order to compute the sum in parallel. The data declared in main take place in RAM and we had to do some action to transfer the data.

```
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);

__global__ void addKernel(int *c, const int *a, const int *b){
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

In the main function two vector are instantiated, sum one with the other and the result printed on screen. The last thing is to close correctly the executable.

```
int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
        c[0], c[1], c[2], c[3], c[4]);

    // cudaDeviceReset must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }

    return 0;
}
```

Let's see how the function is working.Firstable we need to check if our environment is capable of use CUDA library. The function "cudaSetDevice" permit this. After we have to allocate the space in memory, like we do with dynamic memory in RAM. The function "cudaMalloc" is used for this purpose. After we need to copy stuff from RAM to GPU memory and for this the "**cudaMemcpy**" is used.

```
// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;
    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr,
             "cudaSetDevice failed!Do you have a CUDA-capable GPU installed?");
        goto Error;
    }

    // Allocate GPU buffers for three vectors (two input, one output)    .
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    // Copy input vectors from host memory to GPU buffers.
    cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<1, size>>>(dev_c, dev_a, dev_b);

    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr,
             "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
        goto Error;
```

```
    }


    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr,
             "cudaDeviceSynchronize returned error
                    code %d after launching addKernel!\n", cudaStatus);
        goto Error;
    }

    // Copy output vector from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

Error:
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);

    return cudaStatus;
}
```

Measuring performance

Now let's write something to calculate benchmark. Or better to calculate the time spent by an algorithm to compute something. We want to compare the time spent using GPU and the time spent using CPU.

```
#include "time.h"
…

clock_t begin = clock();

/* here do your time consuming */

clock_t end = clock();
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
```

With this code snippet we can calculate how many time our algorithm has used. If we use this program to execute

Matrix multiplication on GPU

Now let's try to measure the performance of GPU with an algorithm that is required sometimes: the multiplication of matrices. With the parallelism of GPU maybe the multiplication of two matrices should consume less time than the time required by CPU. Let's write the algorithm and test!

Instead of two cycles on i and j, the CUDA threading hardware generates all of the threadIx.x and threadIdx.y values for each thread. Each thread uses its threadIx.x and threadIdx.y to identify the row of Md and the column of Nd to perform the dot product operation.

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int width) {
	int tx = threadIdx.x;
	int ty = threadIdx.y;

	float PValue = 0;

	for(int k = 0; k < width; ++k) {
		float MdElem = Md[ty * width + k];
		float NdElem = Nd[k * width + tx];
		PValue += MdElem * NdElem;
	}

	Pd[ty * width + tx] = PValue;
}
```

Lesson 10 - 21/04/2017

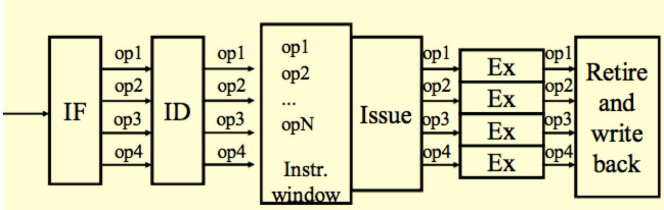
Instruction-level parallelism

Introduction

The idea of parallelizing instruction can be applied also to a sequential traditional program and we can obtain better performance without any effort of the programmer. Only the compiler and the architecture will work together in order to obtain the best performance when a specific program is executed. **It is very important that this result is obtained without effort of the programmer**.

Superscalar Microprocessor

The idea behind this technique is to split the program into different flows of execution and then perform them in parallel, according to certain methods. We want to minimize (not maximize) the CPI that is the number of CPU cycles needed to execute an instruction. Of course we have to consider the average number of cycles. In general, we want to have a set of pipelines, like in the following example where there are 4 pipelines, called 4-way superscalar (or pipelines):



IF: instruction fetch unit
ID: instruction decode unit
Ex: execution unit

This means that the pipeline can execute 4 instructions at the same time. In order to have all the lines (execution part of the ALU or the FPU) used in all clock cycles, we want to minimize the overhead needed to retrieve data at the deepest level of memory hierarchy.

An **Instruction Window** is a scheduler for instructions. Of course at the same time it is not possible to have an unit which can execute any kind of operation. We could have some modules that can execute operations on integers and others that can execute operations on floating point numbers. An execution pointer can execute only increments in order to count or move pointers.
At the end of the day we need 3 kinds of units:

- Arithmetic unit
- Incrementing units

- Floating point units

Which is the right quantity for each kind of unit? The number of execution unit for each feature limits the number of instructions that can be executed in parallel. The number of execution units has to be greater than the number of pipelines (4 pipelines → 6 units: 1 increment, 2 arithmetic and 2 floating points). It is important to have an execution part that can execute operations on some datas. These are ideal pipelines, real implementations may differ.

If I want to have in all cycles 4 instructions able to execute, I need 4 sequential instructions that can be executed in parallel. Finding these instructions in a program code at any time could be an uneasy task. The compiler must limit the number of instruction that must be executed in sequence.

Limitations

The compiler can be limited in this task by the organization of the algorithm: if the algorithm is strictly sequential it is not possible to solve this problem. In some other cases, the only limit are resources used by instructions. These resources are the registers, because in order to execute an operation, the data must be stored inside the register. Generally the compiler uses a general purpose register both to read data and write operation output. So the number of registers available to perform operations is a limit to this superscalar approach. Is it a good idea to increase the number of registers? The number of registers cannot be increased since addresses of register must be written inside instruction opcode. Increasing the number of register increases also the number of opcodes available to write a program but the number of bits per opcode is too short so it is not possible to do this.

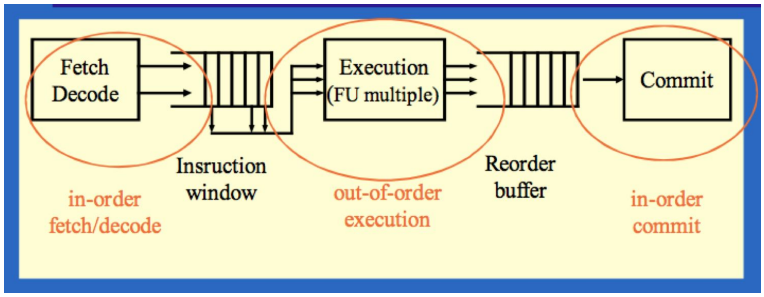
Register renaming

We can organize the pipeline to change the names of registers used by the compiler. This is the same idea of virtualisation at a very low level. The program is translated by considering that the program will store, in specific parts of the instruction, a “virtual address”, while the “physical address” of the register can be changed at run-time. An instruction can use R1 to store a value in another register and another instruction can store a value exactly in R1 from a logical point of view, but at run-time they will use two different physical registers. In this case we can reorganize the system adding a component which is able to change the name of registers used inside an instruction. This component performs the **registers renaming**.

Out-of-order execution

We must have the opportunity to execute a subsequent instruction before a precedent instruction. We must be able to execute the program in a different order, so this technology is called **out-of-order execution**, which is not a really correct word but it is called like this. Using a different order requires also another component which can perform reordering of the results in such a way that the final result is the same of the original sequential execution. Of course the results are written back in the proper order.

At the end of the day the pipeline is split in 3 zones:



The operations computed in the pipeline are:

- Renaming register
- Scheduling of instructions
- Out-of-order execution
- Commit of the results in order

Notice that in order to perform the reordering of the instructions, we need to transfer not a single instruction from the first level of the cache at time but we have to transfer directly one cache block (in 1 or 2 cycles at most). So the the fetching unit from the first level of the cache will fetch cache blocks.

Let's see the following situation:

```
1 - op R1,R2,R3
2 - op R3,R2,R4
3 - op R4,R3,R5
4 - op R2,R4,R3
5 - op R3,R2,R6
```

In this situation, there are 2 instructions that write in R3. It is not possible to execute instruction 2 before the last instruction that uses R3 as input value. This of course is not needed for the algorithm, this is needed for the run time environment that should be met by the compiler. Even if not met by the compiler, the runtime environment can also rename R3 and use another physical address to perform the second block. At the end of the day we can rewrite the registers in the following way:

It is important to store somewhere the original ordering and the renaming operations performed in order to reconstruct the final results that need to be committed at the reordering phase.

Architecture and physical registers

The register name used by the compiler is called **architecture registers.** The architecture is the set of resources that the programmer can use. The registers actually present in the processor are called **physical registers**. The hardware must allocate a new physical register on each writing instruction on that register. Subsequent reads on that register will use the same name, until a new write is executed. When can I deallocate a physical register? When no one is using it anymore but the problem becomes how can we detect it? After the last read is executed on that instruction, it can be deallocated. For example at commit of an instruction, if after that instruction the following renames the register R3 we can then free the previously used one, since no one will use it again in the future. Future instructions will rely on the new physical register assigned to the logical name of R3. **The sequential order is then used to write back commits and free up resources**.

Notice that committing a result to an actual architectural register is not actually needed at all, since actual execution always refers to physical registers. So the commit is not actually needed, in fact it is only needed to free up resources that are not needed anymore. So in current processors, virtual registers have no real counterpart in the physical implementation, only physical registers are put into a processor.

The names of a register can be assigned to any physical register. So the only information that needs to be stored is the link between the architectural name and the physical name. The committed content of the register at any time is not present in any place.

Speculative execution

Instructions may have a field in which an execution condition can be stored, in order to execute a specific instruction only if that condition is met by the status register. This can execute a program without using any conditional jump. It is very easy to add processing units on a chip, the problem is the interaction with the memory. So we can execute all the instructions and then commit only the ones that actually were needed to obtain the final result. So if instruction 2, 3, 4 may be executed or not, depending on the result of instruction 1 on the status register, they are instead all executed and then in the commit phase, for

instance, only instruction 2 and 3 are accepted to commit. This method is called speculative execution. The performances are better than using the traditional conditional jumps, since more instructions may be executed, but they are also executed on more ALUs so it is fine. If an "if" is inside another "if", we can recursively apply this reasoning (not sure). So we end up with predicate arrays, in order to manage more conditions in sequence, like nested "if"s or loops.

Window Registers (Spark microprocessor)

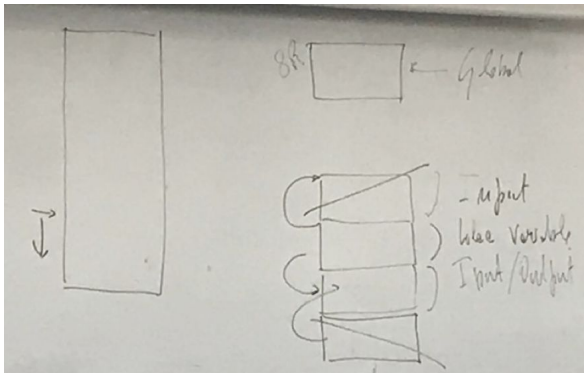
A different approach to renaming of registers is the one used in Spark microprocessors, produced by Sun (Oracle). This mechanism is based on **window registers**.

When the program is running, it may use four groups of registers called windows, storing in the:

- First group, the global variables (in 8 variables)
- Second group, the input values of the current procedure
- Third group, the local variables of the current procedure (typically stack variables)
- Fourth group, inputs and outputs of a possible subprogram called by the current program

When we want to call a subprogram, the compiler transfers the input values from the fourth block's registers into the second window. Actually, when the procedure is called, the system uses the fourth window as the new second window and then allocates a new window; the original second window is no more accessible. It is not overwritten. The new window is used to allocate local variables for the new procedure (new third window). When the procedure terminates, the new third window is destroyed and the original second/third windows are used. This can be applied only when the number of registers that a program can access is managed in blocks of 8. Allocation and deallocation is performed by means of a pointer, which is increased and decreased at procedure call/return.

If the stack of calls is bit, you can use all the "instruction file", the whole number of register (typically 1K register). At this point, registers are saved into the stack, in order to "create space" in the file register.



This idea was developed about 20 years ago, based on a project called RISC microprocessor.
In this approach, the compilers uses many subprograms to implement a program, each using 3-4 input parameters and 3-4 output parameters. Starting from this idea, Sun designers decided that this technique was able to perform many subprogram calls without using stack, which increases drastically the speed of the processor, since it doesn't need to access memory anymore.

Lesson 11 - 26/04/2017

Design for low power

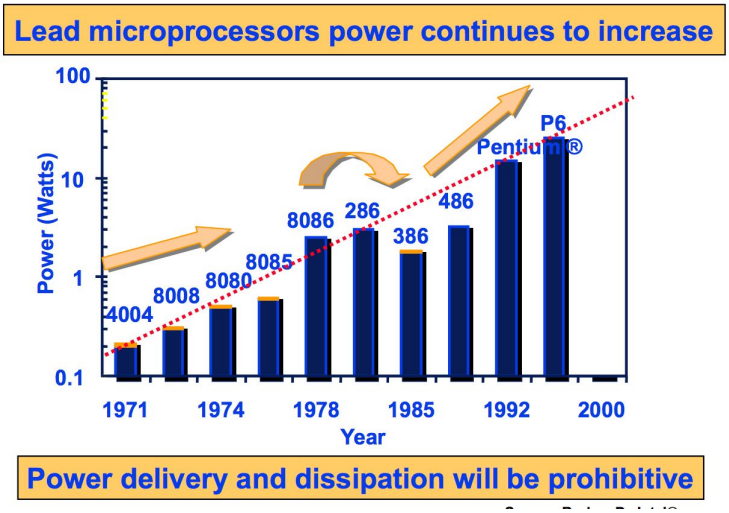
Introduction

We will address the topic of designing for low power. We will deal a little bit on why the power consumption are important by the point of view of electronics. The main reason why is so important the cost related to power consumption are

- Packing costs
- Power supply rail design
- Chip and system cooling costs
- Noise immunity
- Battery life
- Environmental concerns

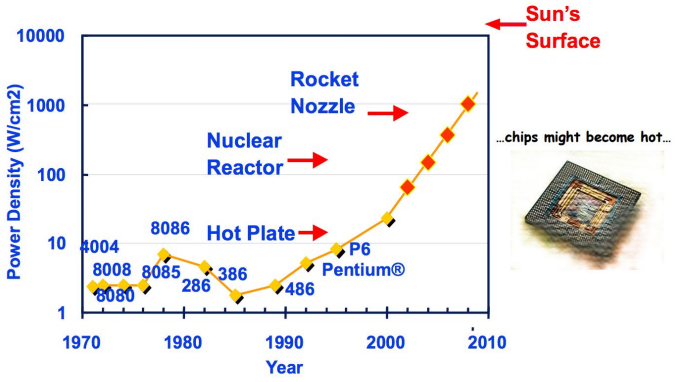
In fact 14% of total US commercial energy uses are related to technology. Office equipment (professional, government and banks) accounted for 14% of total US commercial energy usage in 2012. In US the energy star program is a program related to save energy. The Energy Star program is incorporating standby energy into its ratings. Standby energy in office equipment represents a significant hidden energy cost.

Let's start from a little bit of history, the evolution of processor from power perspective:

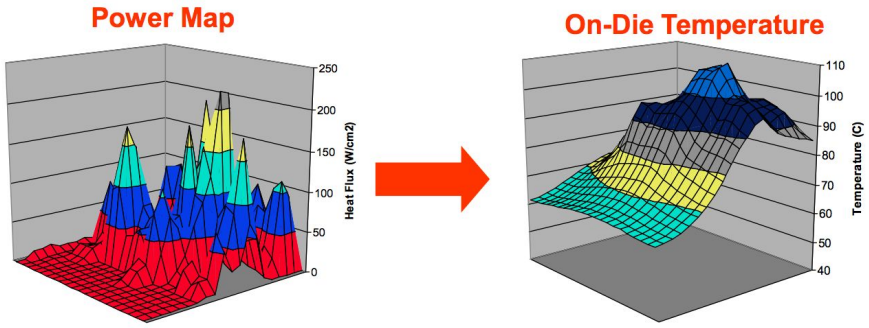


Until 2000, moore's law was exploited to gain speed and increase performance, increasing also power consumption of the processor. Also the power density (power per unit of area) increased a lot in the first evolution of the microprocessor.

At the certain point we were in a very difficult situation we when address the chip x power density. Going forward like this is unsustainable, because it needs too much power for normal usage.



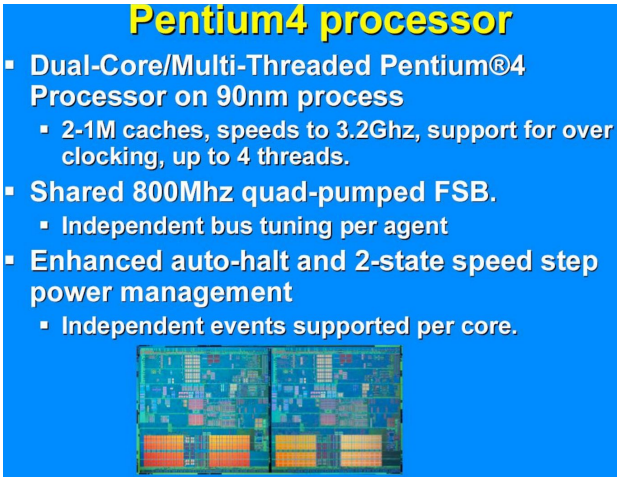
The highest peaks of power per area of a chip are related to the areas with higher temperature on die.



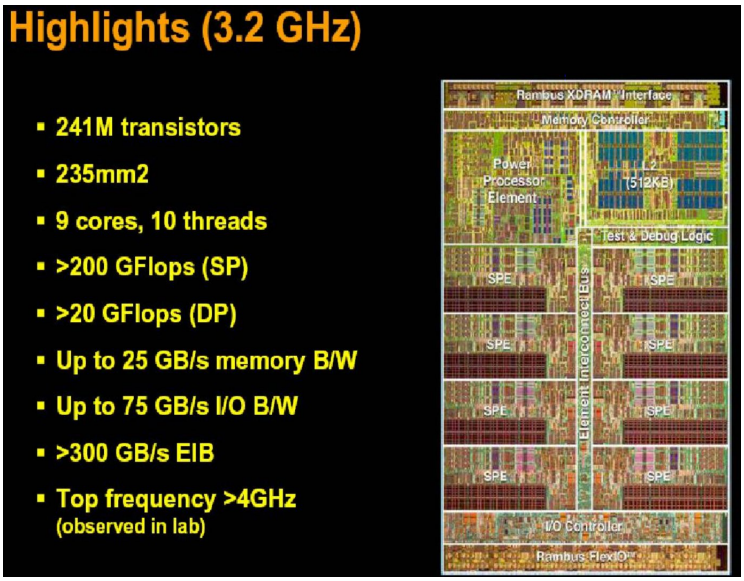
We have a different Power density distributed across the chip. Also silicon is not a good heat conductor. Max junction temperature is determined by hot-spots. Hence we have an impact on packaging, w.r.t. Cooling. This paradigm (follow the moore's law) is going to slow down for that kind of problem. If for a while we remove the cooling fan a CPU can burn up. The result is catastrophic.

For example, data centers should be placed in places where the average temperature is very low, so the cooling system can achieve same results with lower effort.
Around 2004 Intel stop to run to get faster frequency but introduces the multicore ideas in order to maximize the performance. The first was the Pentium 4 processor. The opportunity to implement more transistor per area was exploited to include two cores in a single chip.

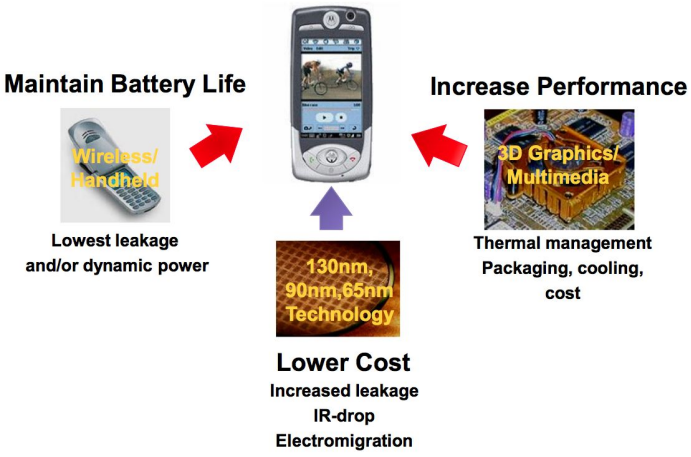
You are of course not doubling the performance, due to issues related to multi core technology.



Another example at that time is the first CPU used on PS3. They exploit the multicore solution in order to gain performance in graphics of the game.

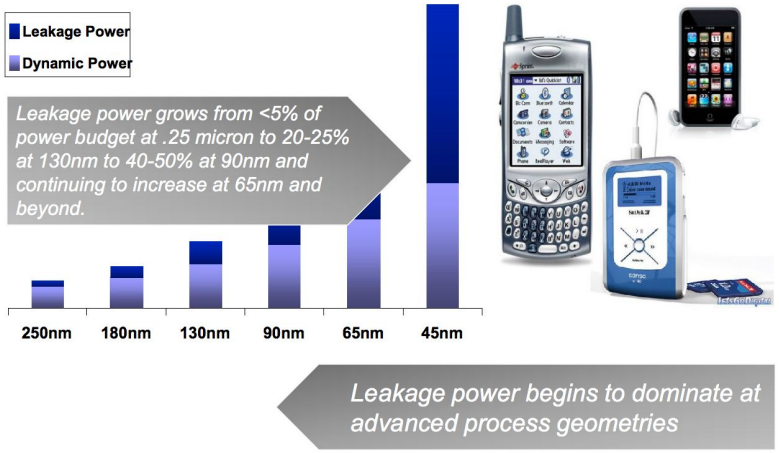


This problem takes dilemma in smartphone world. If we consider that we start from only voice service now we have something extraordinary as power of computation. Especially this is possible by the evolution of the “moore’s” law because we can build up extraordinarily small transistor. Also the recurrent cost are related to the area so the smartphone has not prohibitive cost.

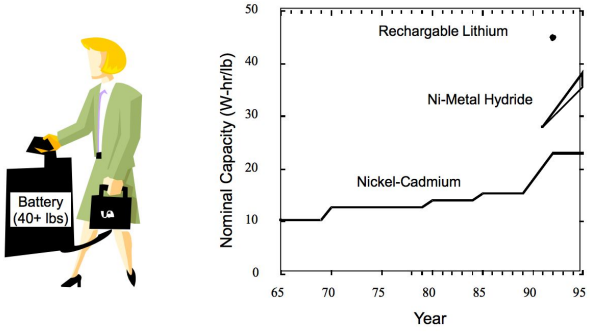


On top of that there is a strong push in direction of power needed to run application, game and so on. We need an high processing power that bring with it the power consumption! Here the problem is that the battery doesn't follow the same degree of evolution of CPU! So due to increased power consumption, also leakage power increased and battery let’s say without evolution in year, we have a big problem! Battery is a real issue now, we need always the cable to connect to alimentation.

We will see later the difference between leakage power and dynamic power, but for now it's enough to say that the percentage of leakage power (leakage means wasted power) is going to increase with the technology increasing dimension capacity.



Let's see we're ok with dynamic power because we ask for the CPU service and we are agreeing to consume power but the leakage power no! Because is wasted and it's a problem nowadays with 40-30 nm transistor!

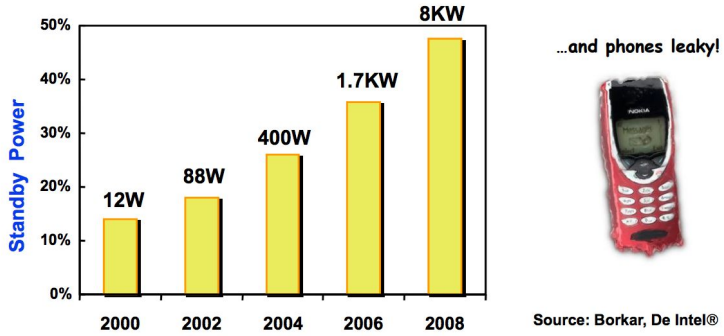


Expected battery lifetime increase
over the next 5 years: 30 to 40%

We say there is a gap from the increasing power consumption and battery evolution. The problem is also the weight and the dimension of battery, we want a mobile device!

Year	2002	2005	2008	2011	2014
Power supply V_{dd} (V)	1.5	1.2	0.9	0.7	0.6
Threshold V_T (V)	0.4	0.4	0.35	0.3	0.25

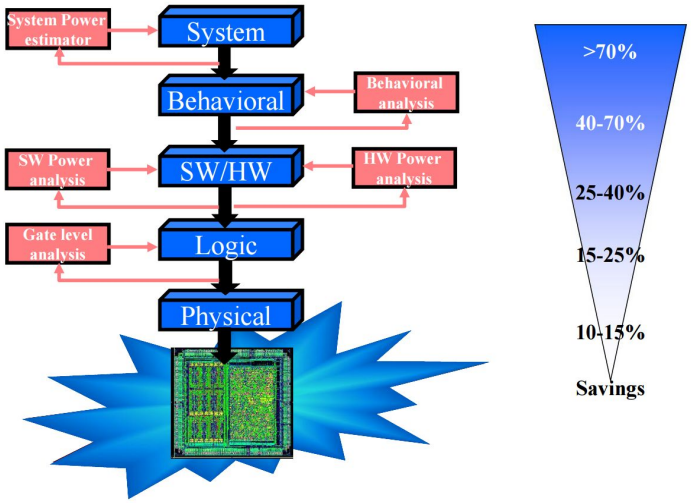
- Drain leakage will increase as V_T decreases to maintain noise margins and meet frequency demands, leading to excessive battery draining standby power consumption.



Leakage power is present also if we do anything so this is the standby power! We are interesting in what we can do when we have to design our system with the point of view of power consumption.

- **The challenge**
  - "To design an embedded system (HW and SW) that provides the target functionality with minimum power consumption"
- **The solution**
  - "From the system concept down to the implementation phase, adopt a design style that includes power consumption as a figure of merit, and exploit all the opportunities and techniques available at each design level to reduce it"

When we have to design especially an embedded system we have to reduce at the minimum power consumption and provide the maximum of CPU power. And this is a challenge. We have to consider power consumption at the beginning at the start step (system level description) and not at the end of design.
In fact we can end up in a circuit that burn up. This is extremely important there are some tools that analyze code, memory access and so on in order to choose the best solution to implement something!



When we abstract adding a level clearly our saving percentage is more but it's not real because we work at high level. When we are more closer of the real physical implementation we can real estimate the power saving. This slide shows that starting from a power consumption of 70% in theory at the highest level we can end up in a 10-15% save power at the physical level. Let's enter in details!

$$E = C_L V_{DD}^2 P_{0\to1} + t_{sc} V_{DD} I_{peak} P_{0\to1} + V_{DD} I_{leakage}$$

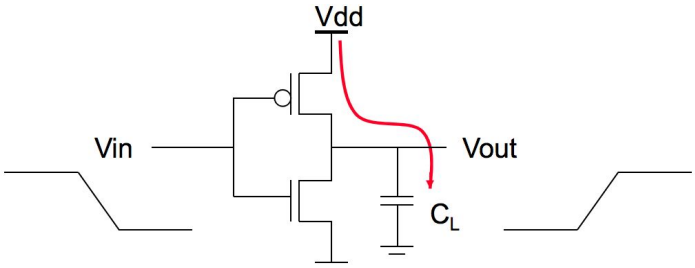
$$f_{0\to1} = P_{0\to1} * f_{clock}$$

$$P = C_L V_{DD}^2 f_{0\to1} + t_{sc} V_{DD} I_{peak} f_{0\to1} + V_{DD} I_{leakage}$$

Dynamic power | Short-circuit power | Leakage power

Dynamic power is related to switching activity of transistor, so it's something we need! We will see the short circuit power and we want to understand the leakage power! Let's start from dynamic power

Dynamic power consumption



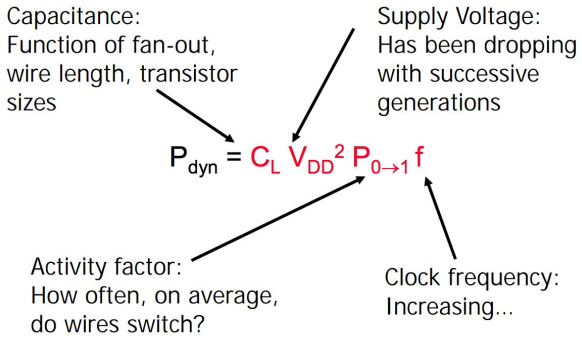
Overall, the power consumption has a formula which is proportional with a quadratic factor to the voltage supply and linearly to the switching activity.
If the output doesn't switch, the dynamic power consumption is zero, but of course we cannot invest in not switching to reduce power.

$$\text{Energy/transition} = C_L * V_{DD}^2 * P_{0\to1}$$

$$P_{dyn} = \text{Energy/transition} * f = C_L * V_{DD}^2 * P_{0\to1} * f \quad (f_{0\to1})$$

$$P_{dyn} = C_{EFF} * V_{DD}^2 * f \quad \text{where } C_{EFF} = P_{0\to1}\, C_L$$

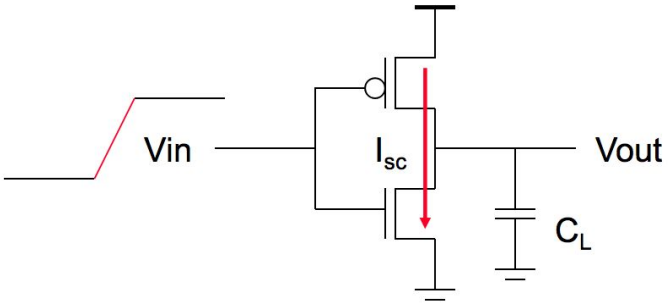
The very interesting thing is that the dynamic power consumption does not depend on size of chip but if we want to decrease the power dynamic is reduce each of that term. So we can reduce the capacitance (how?) and/or power supply. But we can't reduce power supply let's say random! We have to respect something. Because from the voltage depends the delay!



So we can act on some parameter. Firstable by the highest level point of view we have to deal with more performance oriented algorithm with lower number of switching and so on. In principle, from a given algorithm there are some solutions with less switching activity from the others. Or we can act on clock frequency. In fact this is why we don't speed up processor anymore! Just in time performance is a key technique to reduce power consunption as much as possible.

Short circuit power consumption

There is a certain current during switching activity that goes from power supply to ground through the 2 transistors, but it is not so important.



Finite slope of the input signal causes a direct current path between VDD and GND for a short period of time during switching when both the NMOS and PMOS transistors are conducting.

- Duration and slope of the input signal, t_{sc}
- I_{peak} determined by
  - the saturation current of the P and N transistors which depend on their sizes, process technology, temperature, etc.
  - strong function of the ratio between input and output slopes
    - a function of C_L

It is a function of time duration of the current spike from power supply to ground, switching activity, frequency and voltage supply (all linearly) but this is something we are not addressing too much.

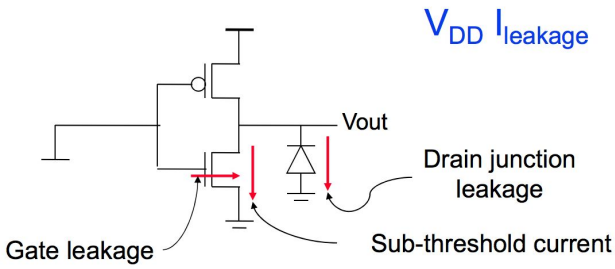
$$E_{sc} = t_{sc} V_{DD} I_{peak} P_{0\rightarrow1}$$

$$P_{sc} = t_{sc} V_{DD} I_{peak} f_{0\rightarrow1}$$

Let's go to leakage power that is the thing we want to know!

Leakage power consumption

Physical phenomena that contribute to it are leakage gates. In fact current through the oxide of the transistor, which should be zero, but going to 40 nm transistor tech statistically you have a current flowing through tunneling effect which is not zero. Then reverse leakage power from the gate.



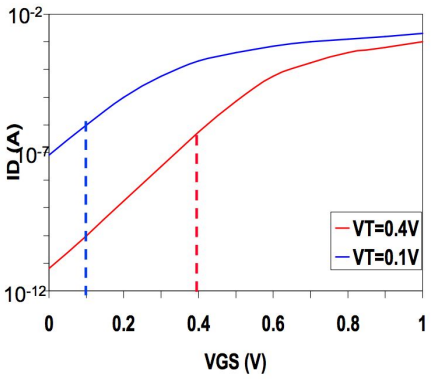
But the most important factor is the subthreshold current. For us the transistor is simply a switch. If the voltage is higher than the threshold current must flow, else not. This is not true!

And this effect is bigger is the channel of MOS is shorter. Also this factor is bigger with temperature. Temperature increase will increase these power consumption quadratically.

So we have to deal with temperature, we have to control it in order to not have high temperature and high leakage consumption. A circuit going faster and consuming less leakage is not possible.

So the transistor today have 2 different threshold. One is to use the full power of transistor but consuming more power. The second is to have a threshold higher but with less power consumption

Let's analyze the image. With threshold of 0.4 we expected that less of 0.4 the drain current must be 0 but is not 0! So we spend power. The same is for the blue curve but higher threshold lower current of drain not desirable.



- An 90mV/decade V_T roll-off - so each 255mV increase in V_T gives 3 orders of magnitude reduction in leakage (but adversely affects performance)

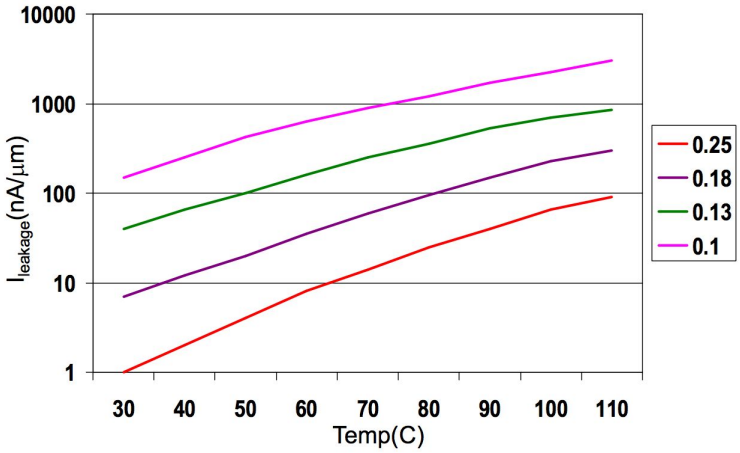
Speed and power consumption are the opposite so we must be clever in order to realize devices with this characteristic. For the same tech process you provide to designers different libraries. General/low power/ultra low power/higher performance.

	CL018 G	CL018 LP	CL018 ULP	CL018 HS	CL015 HS	CL013 HS
V_{dd}	1.8 V	1.8 V	1.8 V	2 V	1.5 V	1.2 V
T_{ox} (effective)	42 Å	42 Å	42 Å	42 Å	29 Å	24 Å
L_{gate}	0.16 µm	0.16 µm	0.18 µm	0.13 µm	0.11 µm	0.08 µm
I_{DSat} (n/p) (µA/µm)	600/260	500/180	320/130	780/360	860/370	920/400
I_{off} (leakage) (ρA/µm)	20	1.60	0.15	300	1,800	13,000
V_{Tn}	0.42 V	0.63 V	0.73 V	0.40 V	0.29 V	0.25 V
FET Perf. (GHz)	30	22	14	43	52	80

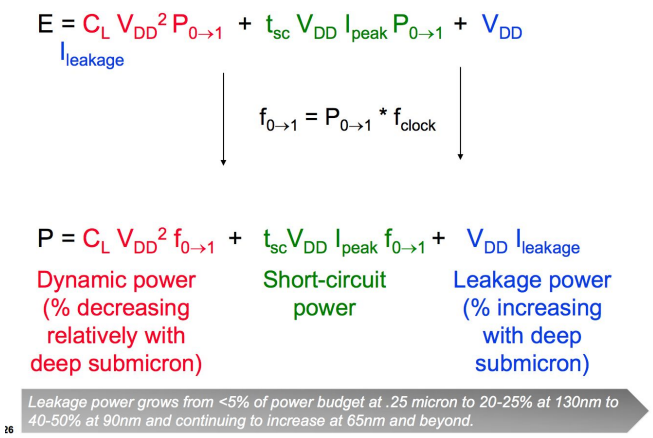
This is an example of a library of transistor. First see the relationship with V threshold and the leakage current. The name indicate the characteristic. LP stands for low power ULP ultra low power, HS high speed.

In fact we can see for example the leakage current of the last one is very high! Indeed, with ULP, the voltage supply is greater but leakage power 2 orders of magnitude less than the LP. However, speed of transistor in GHz cannot be the same in the two libraries.

The point we just mention before is that leakage power is increased exponentially with temperature and increases with smaller technologies.



If we consider the 0.25 we have an increasing from 30 degree temperature to 110. Temperature increasing is 2 order of magnitude! Summarizing we have this three terms and we will focus more on dynamic and leakage.



We deal with the first and the third! Some methods of optimizing guarantees throughput latency and someone that cannot guarantee it.

Latency is the delay from input to relevant output. Throughput is the maximum input we are able to process. This means that sometimes we have good solution in theory where we accept compromise (for example accept speed less performance) in order to reduce power consumption.

Let's see how we can address to this problem reasoning on the design. Sometimes, some solutions at design time we can reduce either the dynamic or leakage power consumption without any degradation in speed performance or we could have solutions which reduce speed performance to reduce power consumption.

Design phase

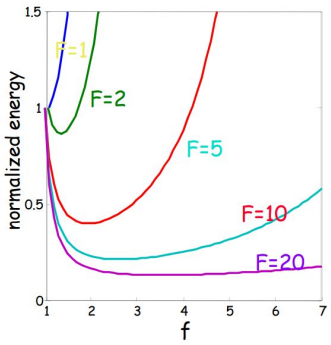
Sizing

This table provides a snapshot of the situation.

	Constant Throughput/Latency		Variable Throughput/Latency
Energy	Design Time	Non-active Modules	Run Time
Active	Logic Design Reduced V_{dd} Sizing Multi-V_{dd}	Clock Gating	DFS, DVS (Dynamic Freq, Voltage Scaling)
Leakage	+ Multi-V_T	Sleep Transistors Multi-V_{dd} Variable V_T	+ Variable V_T

We have a picture that represent a picture of result of rabay book, we consider an inspiration of this topic. When we have to build up something that has to drive an high capacitance

- Device sizing affects dynamic energy consumption
  - gain is largest for networks with large overall effective fan-outs (F = $C_L/C_{g,1}$)
- The optimal gate sizing factor (f) for dynamic energy is smaller than the one for performance, especially for large F's
  - e.g., for F=20, f_{opt}(energy) = 3.53 while f_{opt}(performance) = 4.47
- If energy is a concern avoid oversizing beyond the optimal



When you have to build up a driver to drive a high value capacitance, when you have to drive a lot of registers, you are bringing together a chain of inverters, where you can optimize the size of the inverters to reduce the power consumption.

Whatever it's the increase of the size between inverters and the threshold you need to overcome to drive a transistor, there are different values of frequency that can be used to lower down power consumption. This can be applied at design time. The two aims of energy and power consumption compete one with each other, so the frequencies that need to be selected to reduce one or the other are different.

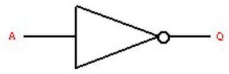
Indeed for us when we have to think about the design of integrated circuit all the information about leakage power and power consumption are defined in library that we will use to build the integrated. Here for instance we have some example of library and information provided by constructor of gate.

INVX1

www.ams.com

Conditions for characterization library **c35_CORELIBD_BC**, corner **c35_CORELIBD_BC_best**: Vdd= **3.63**V, Tj= **-50.0** deg. C .
Output transition is defined from **20%** to **80%** (rising) and from **80%** to **20%** (falling) output voltage.
Propagation delay is measured from **50%** (input rise) or **50%** (input fall) to **50%** (output rise) or **50%** (output fall).

Strength	1
Cell Area	29.120 um^2
Equation	Q = "!A"
Type	Combinational
Input	A
Output	Q



State Table	
A	Q
L	H
H	L

Capacitance [fF]	
A	2.8210

Propagation Delay [ns]					
Input Transition [ns]		0.01		4.00	
Load Capacitance [fF]		5.00	100.00	5.00	100.00
A to Q	fall	0.04	0.41	-0.36	0.63
	rise	0.06	0.67	0.83	1.88

Output Transition [ns]					
Input Transition [ns]		0.01		4.00	
Load Capacitance [fF]		5.00	100.00	5.00	100.00
A to Q	fall	0.04	0.54	0.70	1.43
	rise	0.08	1.03	0.62	1.58

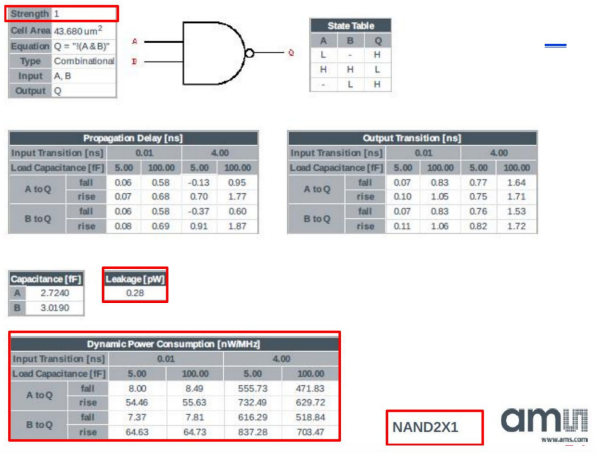
Dynamic Power Consumption [nW/MHz]					
Input Transition [ns]		0.01		4.00	
Load Capacitance [fF]		5.00	100.00	5.00	100.00
A to Q	fall	1.93	2.35	516.63	510.09
	rise	38.82	40.07	813.89	712.95

Leakage [pW]
0.26

If you are going to check, in previous technologies leakage power wasn't even mentioned, because it was negligible, but now it is not anymore for new technologies. At that time constructor not mention at all the leaked power because was a small number. Now it's mentioned always!

Let's see some example of Austriamicrosystems!

// SOME EXAMPLES ON SLIDE



Of course depending from the probability of switching from 0 to 1 and vice versa are different and depends from application.

- Switching activity, $P_{0\to1}$, has two components
  - A static component – function of the logic topology
  - A dynamic component – function of the timing behavior (glitching)

2-input NOR Gate

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

Static transition probability

$P_{0\to1} = P_{out=0} \times P_{out=1}$

$= P_0 \times (1-P_0)$

With input signal probabilities

$P_{A=1} = 1/2$

$P_{B=1} = 1/2$

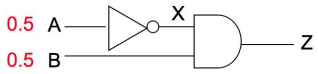
NOR static transition probability

$= 3/4 \times 1/4 = 3/16$

We have to analyze the various probability separately. For example in a NOR the out equal 1 is present only ¼ while the 0 is present ¾. The static transition probability is P0->1 = P0 * (1-P0). You then assign probabilities to input values and you obtain that P0->1 = 3/16.

If we are going to consider this for any kind of gate, we are able to analyze every kind of logic. You are then able to analyze any logic with respect to the static logic transition probability even for complex systems.

	$P_{0\to1} = P_{out=0} \times P_{out=1}$
NOR	$(1 - (1 - P_A)(1 - P_B)) \times (1 - P_A)(1 - P_B)$
OR	$(1 - P_A)(1 - P_B) \times (1 - (1 - P_A)(1 - P_B))$
NAND	$P_A P_B \times (1 - P_A P_B)$
AND	$(1 - P_A P_B) \times P_A P_B$
XOR	$(1 - (P_A + P_B - 2P_A P_B)) \times (P_A + P_B - 2P_A P_B)$



For X: $P_{0\to1} = P_0 \times P_1 = (1-P_A) P_A$

$= 0.5 \times 0.5 = 0.25$

For Z: $P_{0\to1} = P_0 \times P_1 = (1-P_X P_B) P_X P_B$

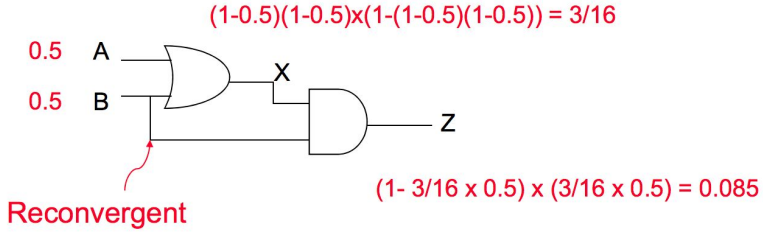
$= (1 - (0.5 \times 0.5)) \times (0.5 \times 0.5) = 3/16$

This means than when we have to evaluate the power of this gate we have to consider this terms as probability. And when we want to implement let's say a gate that realize some function maybe I have some different solution, so I have to take in account this thing in order to choose the better one.

If you want to achieve a given logic function, you can analyze 3 solutions, the static transition probabilities and then reduce dynamic power consumption by just considering different solutions with different probabilities.

We don't do this thing, maybe tools do things, we can't realize the probability with for example million of gate! Here for instance we have another issued

- Determining switching activity is complicated by the fact that signals exhibit correlation in space and time
  - reconvergent fan-out



$$P(Z=1) = P(B=1) \& P(A=1 \mid B=1)$$

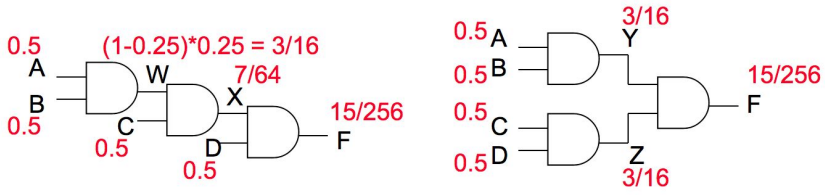
- Have to use conditional probabilities

What if a single signal in an input for more than one component in a single network?

We have conditional probabilities. As soon we use complex logic we need to introduce another probability tool to calculate probability value.

- Logic restructuring: changing the topology of a logic network to reduce transitions

AND: $P_{0\to1} = P_0 \times P_1 = (1 - P_A P_B) \times P_A P_B$



Chain implementation has a lower overall switching activity than the tree implementation for random inputs

Ignores glitching effects

Let's consider a chain solution or a tree solution. The two logic are implementing the same function. Let's consider the same static transition probability for all inputs in the two solutions. The one more convenient should be the chain implementation, because it has a lower overall switching activity.

These probability terms are related to switching activity, so we can apply them to calculate the dynamic power supply.

The probability value clearly is the same for the two realization, they realize the same logic function! And from the capacitance point of view the load is the same for each port. But we have to analyze the port in the chain. The two single port of the second solution have probability of 3/16 and the other 7/64 that is better!

Clearly for random value (are probability) and we don't' take in account glitching. This because the chain is more subject to glitches. Any glitch consumes energy, so the glitching probability is more for the chain probability.

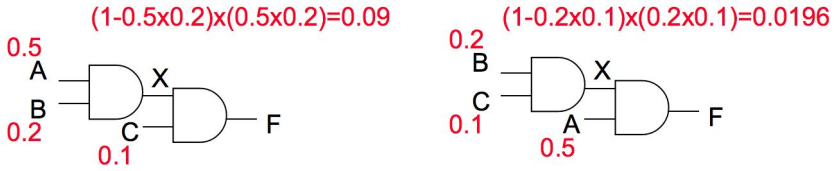
What is glitching? Consider the AND truth table

2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

When you move from 01 to 10, the output doesn't change, but if A and B have got a race and the computation goesh through 11, in between the output changes 2 times 0->1->0.

This is a glitch and this consumes power. To avoid these or consider these in power consumption we need to analyze the system more carefully.

Another important thing that we can implement using appropriate tool is input reordering. We can change the way in which we provide the input. For example input in the slide has re-order!



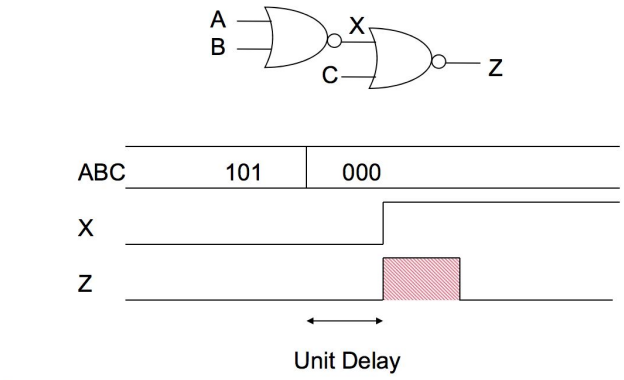
Input ordering is important too when they have different input static transition probability, even if the topology of the network is the same. A general rule is to put the higher static transition probability on the point nearer to the end of the chain (near to output).

In principle we have a quite big different from 0.09 to 0.0196. This just be careful on where provide the same rule! We do not' affect anyhow the speed but we are reducing of 1 order of magnitude the consumption.

Of course we can consider this kind of information depends on the application, because the inputs depend on the application you run on the device, so the test plan used to design the system should be closest as possible to a mockup application that will be executed on the device.

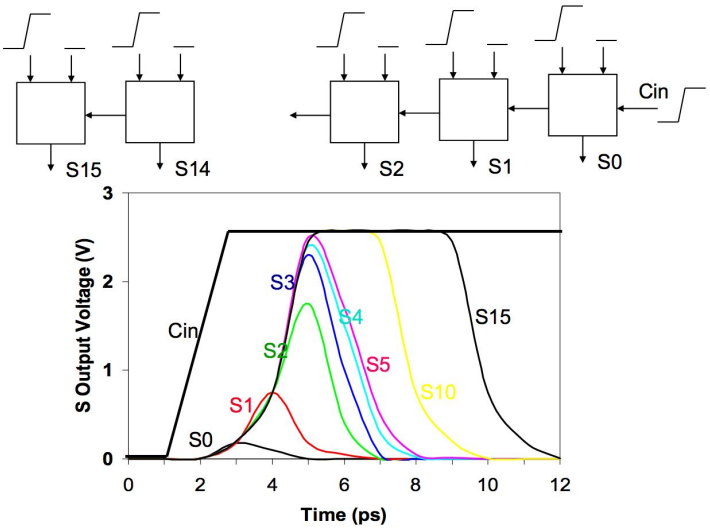
Another point is what we have already discussed. Consider that the gate can have a unit delay, we have a situation in which Z switch after some time. We have a glitch.

- Gates have a nonzero propagation delay resulting in spurious transitions or glitches (dynamic hazards)
  - glitch: node exhibits multiple transitions in a single cycle before settling to the correct logic value



This is totally wasted. Also the glitch is not a good thing, there is the risk to do some error. From the point of view of driving correctly. In digital synchronous design, glitches are ok, but they could be a problem from the functional behavior in asynchronous design.

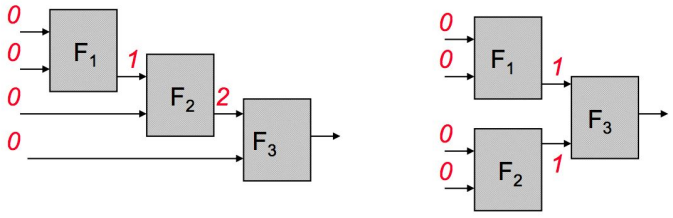
This is an example of glitching a ripple carry of 16 bit. All the area under the curve is wasted energy. The area of the voltage signal due to glitch is equal to the power which is wasted in the process.



In an RCA we could have a delay in the carry chain, so the outputs of the single full adders in the chain may be affected by glitches, with increasing area (more waste) the more we go into higher levels of the chain.

From this perspective we want to look into the working at compare at least two solution.

- Glitching is due to a mismatch in the path lengths in the logic network; if all input signals of a gate change simultaneously, no glitching occurs



So equalize the lengths of timing paths through logic

Between a chain and a tree, a tree is much better to reduce the glitches, which is against the static analysis that we did before! There is no silver bullet to this problem.

We have to consider the exact numbers of dynamic power consumption that are coming from topology or from the glitch.

And of course the tree has also lower delay. When inputs have the same delay, there could be less races, while if they are delivered with different delays there could be even more.

Multi VDD

So let's go we're going to analyze a solution to reduce power consumption with multi vdd.

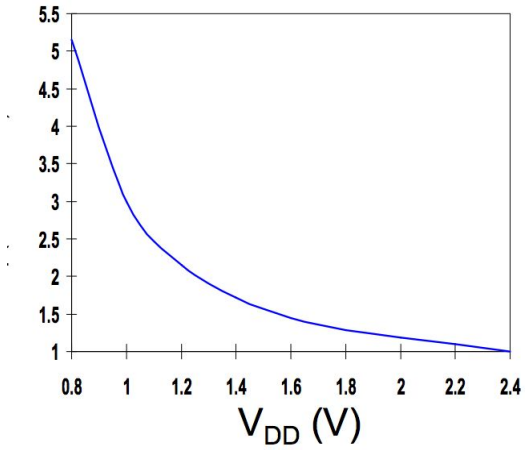
	Constant Throughput/Latency		Variable Throughput/Latency
Energy	Design Time	Non-active Modules	Run Time
Active	Logic Design Reduced V_{dd} Sizing Multi-V_{dd}	Clock Gating	DFS, DVS (Dynamic Freq, Voltage Scaling)
Leakage	+ Multi-V_T	Sleep Transistors Multi-V_{dd} Variable V_T	+ Variable V_T

Here we have if we remember the propagation delay high to low and low to high depends from the load.

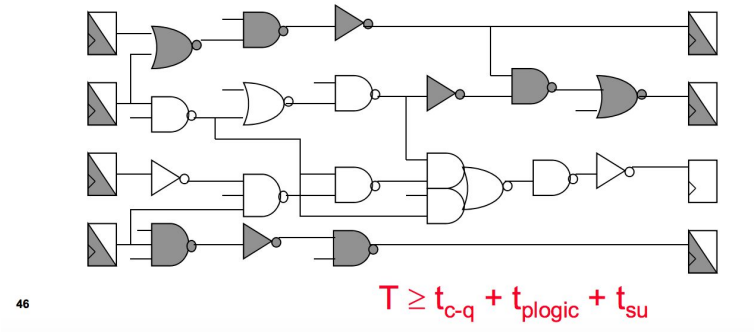
$$t_{p_{HL}} \propto \frac{K \cdot C}{\beta_n \cdot (V_{DD} - V_{Tn})}$$

$$t_{p_{LH}} \propto \frac{K \cdot C}{\beta_p \cdot (V_{DD} + V_{Tp})}$$

Hence VDD is there also in leakage power and dynamic power! Low the voltage is a very good thing! This solution can reduce both dynamic and leakage power by reducing the vdd, but the delay time increases! But we can't do in a random way.



Because the delay is increased is we reduce the VDD! So the idea is that reducing power supply reduce power consumption. I see that logic is slower. In order to not reduce too much the propagation I can reduce the power supply to non-critical path. Clearly I have the kept the non critical path non critical because if I lower too much the supply become critical! This will reduce overall power consumption maintaining the current overall speed of the network.



Example, the idea is there are a lot of path in which I have less logic than the central one. So I can reduce the power supply in order to balance the t propagation and lowering the power supply! So in the overall speed we don't have any difference!

We have to say indeed when I have to modify the power supply I can not put one after the other logic with different power supply! I cannot put one after the other logic components that work with different vdd levels. If I use two different Vdd values, components may not be able to speak one with each other.

In digital world I consider a threshold for high value (for example from 3.5 to 5 V is high) and one for low value. Hence I cannot use different power supplied logic because component must talk!

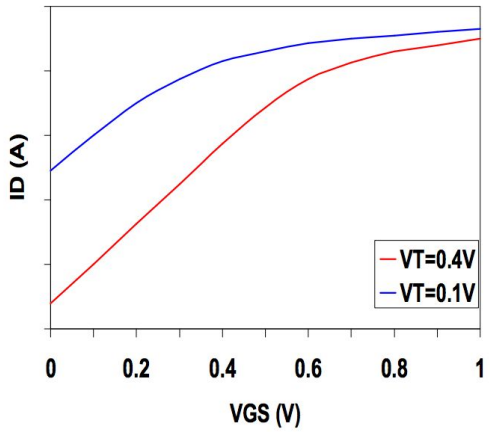
So in order to use multi vdd functionality I have to include something that is used to make talk logic block. In this case, in order to apply these solutions I need to increase the

complexity to introduce level converters, which have transistors and then increase a bit power consumption. And this component consumes energy so it must be calculated!

Multi threshold

	Constant Throughput/Latency		Variable Throughput/Latency
Energy	Design Time	Non-active Modules	Run Time
Active	Logic Design Reduced V_{dd} Sizing Multi-V_{dd}	Clock Gating	DFS, DVS (Dynamic Freq, Voltage Scaling)
Leakage	+ Multi-V_T	Sleep Transistors Multi-V_{dd} Variable V_T	+ Variable V_T

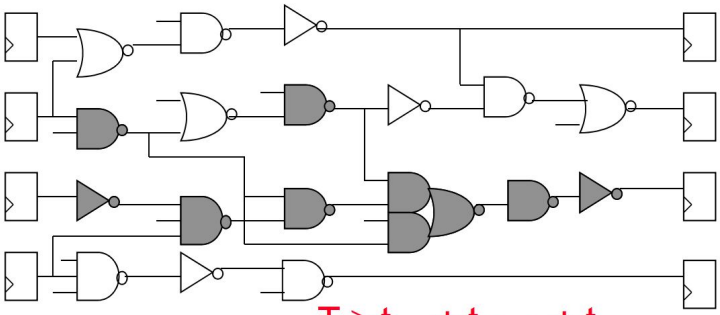
Transistors with different threshold voltages.



Also we can use difference threshold! As we see many minutes ago.

These delay are not put in the critical path. All the gray block are realized with transistors with lower threshold voltage in order to gain better performance. The others are with higher threshold voltage!

- Minimum energy consumption is achieved if all logic paths are critical (have the same delay)
- Use lower threshold on timing-critical paths
  - Assignment can be done on a per gate or transistor basis; no clustering of the logic is needed
  - No level converters are needed



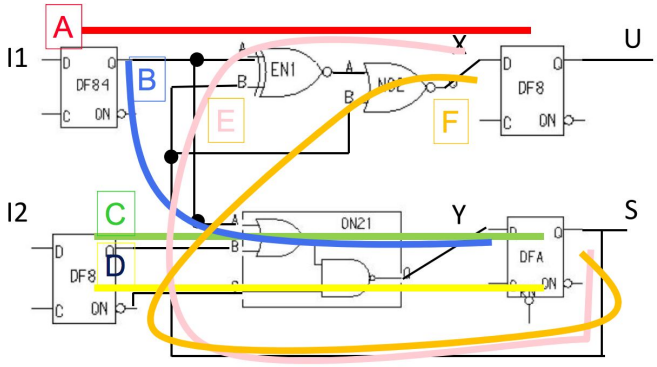
49

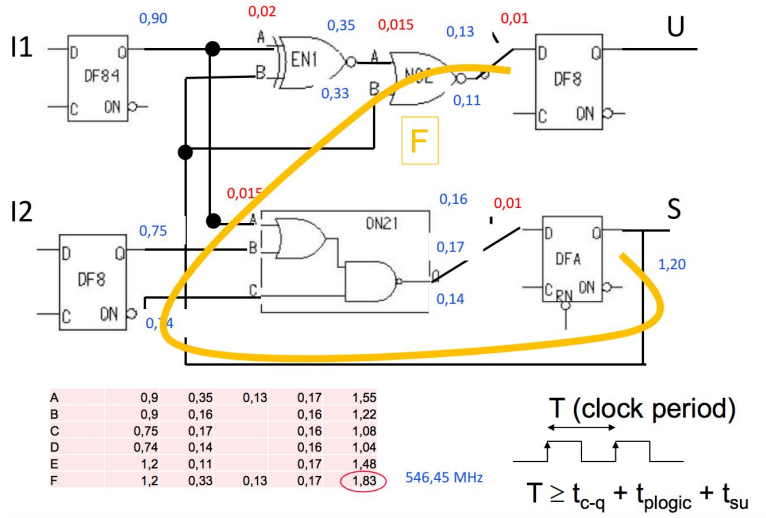
$$T \geq t_{c\text{-}q} + t_{plogic} + t_{su}$$

This is clear also from the formula of prograpation time! Having higher threshold voltage cause a longer propagation time. With the careful attention to this thing we can save up to 20% saving of energy.

- Determine the critical path(s) at design time and use low V_T devices on the transistors on those paths for speed. Use a high V_T on the other logic for leakage control.
  - A careful assignment of V_T's can reduce the leakage by as much as 80%

// EXAMPLE





We can't deteriorate too much the time in the non-critical path because I don't want to affect the overall speed of the circuit!

Low Power Techniques in Microarchitectures and Memories

Now few information on what can be done in micro-architecture level. We say we need to address the problem in design phase but here we want to see what really can be done in hardware to optimize!

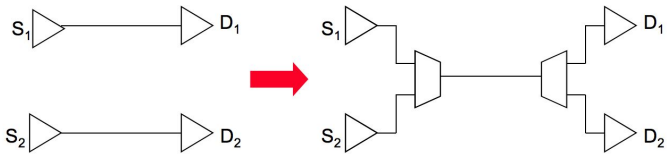
Bus multiplexing

Historically hardware talk each other with buses. Buses are characterized by high switching activity and high capacity loading. So they play an important role in energy consumption. So let's see something about.

- Buses are a significant source of power dissipation due to high switching activities and large capacitive loading
  - 15% of total power in Alpha 21064
  - 30% of total power in Intel 80386

In principle let us suppose to have a 32 bit bus or 64 bit bus. They are quite complex, the need an amount of area non neglectable.

- Share long data buses with time multiplexing (S_1 uses even cycles, S_2 odd)



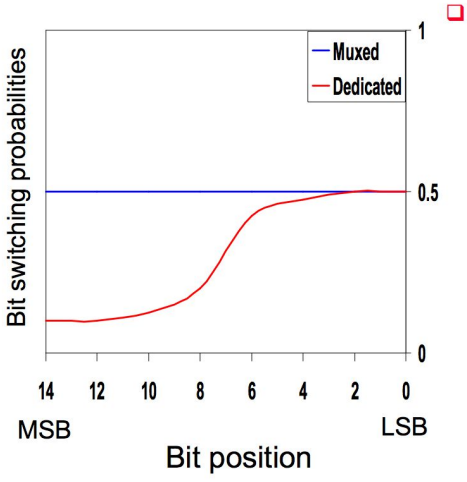
- But what if data samples are correlated (e.g., sign bits)?

So we can decide for example, if S1 and S2 use the same bus can share the bus using a multiplexer! Sharing long data buses between more than one different source/destination using a time multiplexing is a solution that can reduce very much power. This is very good only if the sources/destination work at different times.

If the data samples that transit on bus are completely random, the multiplexer don't deteriorate from the point of view of switching activity. And it's the same if the data are strongly correlated between S1-D1 and S2-D2, but if they are not correlated between the 2 buses this will deteriorate power consumption.

Imagine the S1-D1 is a videos stream of Rai1. A guys is speaking with blue background. The kind of information provided on the bus at each instant of time is a small modification of the previous one (change only the position of the mouth of the guy for example). And in S2-D2 a guy is speaking on Channel 5 with a green background.

Switching the two bus introduce a big switching activity! Instead keeping the bus separated, the switching activity is lower because the background is always the same. So we need to take in account this thing, is not convenient switch from Rai1 to Channel5!



- For a shared (multiplexed) bus advantages of data correlation are lost (bus carries samples from two uncorrelated data streams)
  - Bus sharing should **not be used** for positively correlated data streams
  - Bus sharing **may prove advantageous** in a negatively correlated data stream (where successive samples switch sign bits) - more random switching

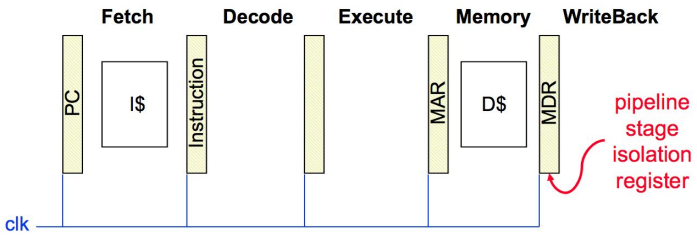
Pipeline

Another example is Pipeline.

- Glitches depend on the logic depth of the circuit - gates deeper in the logic network are more prone to glitching
  - arrival times of the gate inputs are more spread due to delay imbalances
  - usually affected more by primary input switching

You already know the glitch problem and we know that the depth of the logic increase the glitch probability. What we can do a system level is to include a pipeline register in order to reduce the depth of the logic. This solution can bring a reduction in power consumption.

- Reduce logic depth by adding pipeline registers
  - additional energy used by the clock and pipeline registers



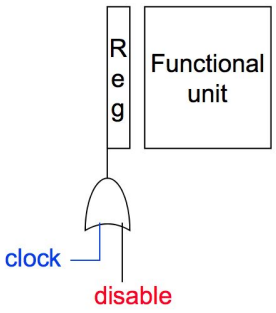
Clock gating

Another important thing is clock gating.

	Constant Throughput/Latency		Variable Throughput/Latency
Energy	Design Time	Non-active Modules	Run Time
Active	Logic Design Reduced V_{dd} Sizing Multi-V_{dd}	Clock Gating	DFS, DVS (Dynamic Freq, Voltage Scaling)
Leakage	+ Multi-V_T	Sleep Transistors Multi-V_{dd} Variable V_T	+ Variable V_T

If for instance we have a microcontroller architectures where we have the program counter, instruction and so on. Of course at the design time we have to think for example if we need a floating point unit. But we have to think also if we have the capability to switch off the floating point unit when is not used! This is done by gating the clock, we can switch a piece of hardware!

- Gate off clock to idle functional units
  - e.g., floating point units
  - need logic to generate **disable** signal
    - increases complexity of control logic
    - consumes power
    - timing critical to avoid clock glitches at OR gate output
  - additional gate delay on clock signal
    - gating OR gate can replace a buffer in the clock distribution tree



We must stop the clock with a simple gate, is fixed to 1 or 0 logic. Of course this reduce to 0 switching activity! We remove completely any power consumption dynamic. We have the leakage power because the logic is still there. We will see is that possible to remove also leakage but for now not.

Dynamic frequency and voltage scaling

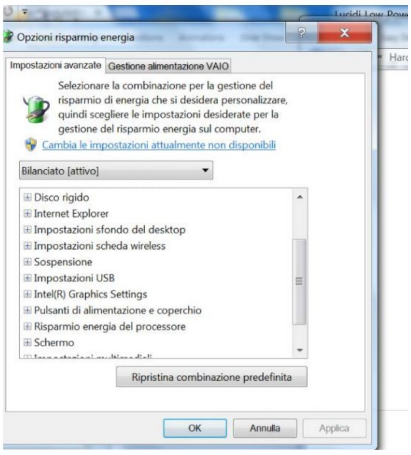
	Constant Throughput/Latency		Variable Throughput/Latency
Energy	Design Time	Non-active Modules	Run Time
Active	Logic Design Reduced V_{dd} Sizing Multi-V_{dd}	Clock Gating	DFS, DVS (Dynamic Freq, Voltage Scaling)
Leakage	+ Multi-V_T	Sleep Transistors Multi-V_{dd} Variable V_T	+ Variable V_T

Now let's see another solution. The technique are called dynamic freq scaling and dynamic voltage scaling. A couple of example. We have for example the Intel Speedstep and the TransMeta longrun.

These are two CPU design in order to change the power supply and the frequency. The philosophy is that if we are willing for instance to use our notebook for much time but we want less performance, we can (from PLL) change the CPU frequency! The dynamic power level is related of frequency so reducing frequency reduce consumption!

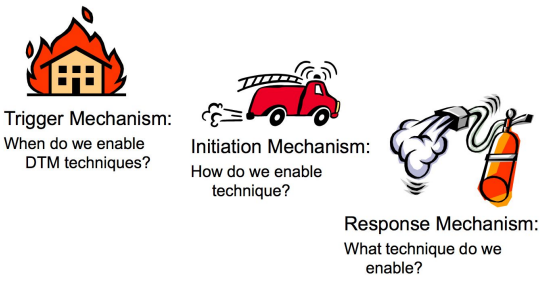
- Intel's SpeedStep
  - Hardware that steps down the clock frequency (dynamic frequency scaling – DFS) when the user unplugs from AC power
    - PLL from 650MHz → 500MHz
  - CPU stalls during SpeedStep adjustment
- Transmeta LongRun
  - Hardware that applies both DFS and DVS (dynamic supply voltage scaling)
    - 32 levels of V_{DD} from 1.1V to 1.6V
    - PLL from 200MHz → 700MHz in increments of 33MHz
  - Triggered when CPU load change is detected by software
    - heavier load → ramp up V_{DD}, when stable speed up clock
    - lighter load → slow down clock, when PLL locks onto new rate, ramp down V_{DD}
  - CPU stalls only during PLL relock (< 20 microsec)

In the Long Run we can also reduce the voltage! If we check on battery settings we have some and some method to reduce power consumption. So this feature are implemented in low level in order to provide to the user a GUI to control this things!



Clearly we can't change frequency and voltage every second, is power consuming! Every time you change clock frequency, in order to avoid data loss you need to stop everything, adjust parameters and then restart again.

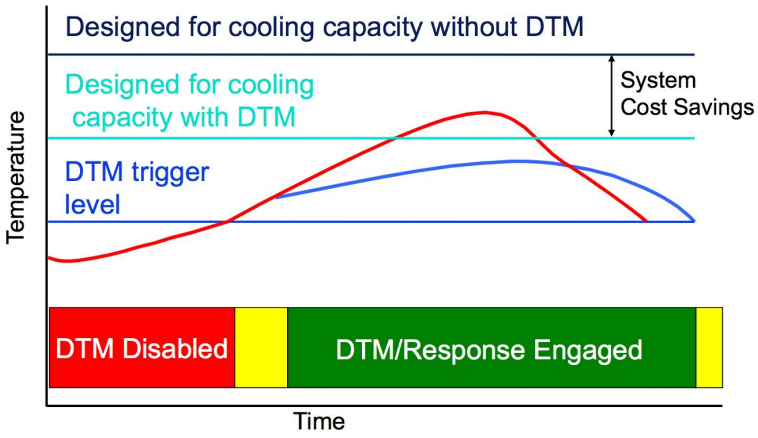
Dynamic thermal management



The topic now is how to deal with DFS and DVS in order to reduce the cost of cooling system? We want something to play it in dynamic way let's say at runtime! The previous mechanism (DFS, DVS) can also be used to manage dynamically the heat of the processor.

A simple cooling system, handling the heat for a microprocessor can be very expensive when there is no DTM, because it need to be able to handle higher temperatures for longer times. Also for a desktop computer, if we are using a fan basing cooling system or liquid cooling system cost are really different.

Let's start from history.
I'am in industrial plant. We have a system to track the plaint in 24h during the activity. Let us suppose the profile of workload is this kind of profile. This is something reasonable in industrial plan because we have moments of high workload. We have to reason at worst case scenario! **Red** line is the workload.



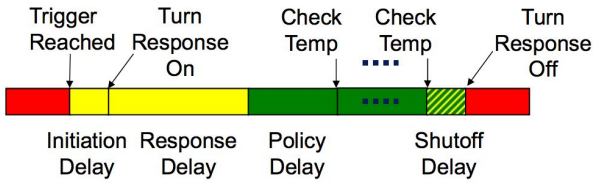
Clearly I have to be sure, I have to have a margin from the worst case. So I design the cooling system as the **black** line. But in the duration of all process of build something for example I can accept a little delay in order to save money! So I can decide to slow down the system.

When the situation is getting worse, trigger ask for the thermal management to reduce frequency and power supply. The overall process takes much time but I can use a system that is able to cool a bit less and save money!

How to do this? FIrstable we need a temperature sensor and typically nowadays temperature sensors are always on the die! Clearly we can modify the frequency clock and so on for a peak of temperature! I have to measure the temperature for an interval of time.

Of course, the width of the window of observation has to be defined carefully, not too long to avoid losing some time to reduce temperature or not to narrow, to avoid extra non needed work.

So we need a trigger mechanism, someone that reads the sensor and triggers an action and the actual action.



- Initiation Delay – OS interrupt/handler
- Response Delay – Invocation time (e.g., adjust clock)
- Policy Delay – Number of cycles engaged
- Shutoff Delay – Disabling time (e.g., re-adjust clock)

The delays needed to achieve the goal are shown in slide above.

Lesson 12 - 28/04/2017

Cache in multiprocessor environment

Introduction

Today we wanna investigate the use of cache in multiprocessor environment. The cache subsystem in a multiprocessor can deliver additional features like

- minimizing the traffic through main memory, in this way you can put on the same bus more processors
- minimize the access to the shared memory among two running processor because the fastest processor acquire copy of memory that will be used by also the second

In general the shared bus is the bottleneck by the point of view of performance if we talk about multiprocessor environment. Also there is a big problem we need to solve.The problem derives from the access to the shared copies in write.

Example.
Let us suppose we have two different processors, and each of one have a private cache. When a process run on one of the processor and requires something in memory, a block of memory will be copied on the respective cache.
When the process have to perform a write there are some paradigm.

- If process write also in the main memory, the use of bus is increased, reducing the possibility of other process to access to the memory.
- But also if we keep all local, other process can be aware of the changing.

Solution.
1. So one possible solution is to write directly also in other private caches (propagate)! This is the fastest strategy! We will see how.
2. The second strategy is to invalidate other local cache copy in order to signal to other processor that data is no more valid.

It is important to understand that when I want to discuss a solution to a problem, I must evaluate the size of the problem. We have to evaluate the performance gain of each of the possible solution. If the problem is very low on performance, the solution has to provide a final big improvement in performance. The propagation solution is problematic from a performance point of view.

Let’s consider 100 accesses to the main memory, 20 writes and 80 reads operation. Cache can create a very low number of miss percentage. Let’s suppose 3 miss on 100 accesses. If I want to operate in traditional way, sending the writes on the main memory, on the bus I’d have 3 read operations to solve the misses and 20 writes through.

If I want to reduce this traffic I must work on these 20 writes, so the problem exists and solutions can provide a huge improvement in performance.

If I am able to use in a system asynchronous operation, I can increase the use of shared bus. The idea is to perform the write operation only when is needed, of course using the bus when is not used. It's strategic to manage in this way the operation.

If I want to use the first strategy (update caches of other processor), I must use a protocol that is called **coherence protocol**. The protocol must be able to ensure that all the copied on the system are updated as soon as possible. This protocol ensures consistency of caches data. In a multiprocessor I will have a coherence protocol to solve the consistency problem of the whole system.

Handling cache in multiprocessor environment

Snooping

Let us suppose I have a multiprocessor architecture and I want to use the first strategy (propagate the valid data value). How can I implement a coherence protocol?
All the operations on shared copies must be observable from a certain point on the system. To do so, each controller of a cache is able to **snoop** on the bus in order to update the local copies after write operations. However, this architecture is not used because the shared bus is the bottleneck, so growing the number of the processor this solution becomes highly inefficient. This solution is used for 3-4 processor maximum.

Shared cache

Another solution is the following, between the local caches and the memory we put a shared cache, so we can have high speed links between the local and the shared cache (putting them altogether in a single chip). Imagine I want to implement a machine with 100 processor. The architecture is thought as composition of shared and private memory. Each core has its private cache and the entire system has one shared cache. But again when the number of processors increases this becomes a problem.

Multi computer

Another solution wants that interaction among processors can be obtained by shared links, like a network. The informations are sent and received. This goes against the shared bus/memory mechanism in which load/store operations are used.
This solution is called **multi computer**. Each processor has a local memory and the interactions are made with a message passing scheme. In this scheme, we can have how many processor we want.

Again this solution present a problem. Remember what are the two rules:

1. In all moment each CPU must have work to do, and we must balance workload
2. We have to reduce at minimum the communication on links

In this scheme it is difficult to solve each of the problems:

1. Not easy to design an application that will produce a balanced workload for all processors.
2. Two processes that will communicate frequently should be placed on near processors, in order to have a lower communication overhead.

Firstable it's very difficult to balance the works of each CPU, also because for balance workload processor must talk each other! So problem 1 and 2 are in conflict each other!

For the programmer point of view the multicomputer approach is transparent, the programmer sees the load/store operations even in the multicomputer architecture, but the program is translated instead into a send/receive one.

The component act to balance the workload is the scheduler! We must have a general scheduler that balances workload! This solution is called distributed and shared machine.

Hybrid approach

Of course we can have another solution in which each processor of the machine is inside the same chip. The current solution in high speed machines is to put processors on a same chip and use a shared memory inside it, but there is no point in which different chips can observe all the operations in all the other chips. In this situation the problem is that we don't have a point in system in which we can observe all the operation performed (such as the shared bus) because each process do each work alone! So we must implement something in order to balance work! And this is possible by means of **directory protocol**. A Directory Protocol is able to send for each block the current state (only a copy exists or there are multiple copies) and a list of knowns that have a copy of this memory block.

Conclusion

Concluding we can say that to manage cache in multiprocessor environment we have to deal at least with two protocol. We have 2 different approaches to solve the problem of local copies inside local caches involved in write operations:

1. **Snooping** protocols, used in small multiprocessors, in which there is a point where all write operations performed by the system are observable.
2. **Directory** protocols, in a multi computer approach, where this technique is not applicable, there is a special component which stores informations about all memory blocks and the list of all processors which have this memory block.

The write can be managed in 2 ways:

1. Update of all copies
2. Invalidate on a write all the rest of the copies, in order to have only one updated copy on the system and no outdated copies.

At the end the coherence protocol is designed in order to perform the least number of actions that have to be done in order to obtain coherence. And is not important that the actions are really fast but is important to perform this action when the bus is free namely when the load is low! Because we don't want to make load heavier to the system!

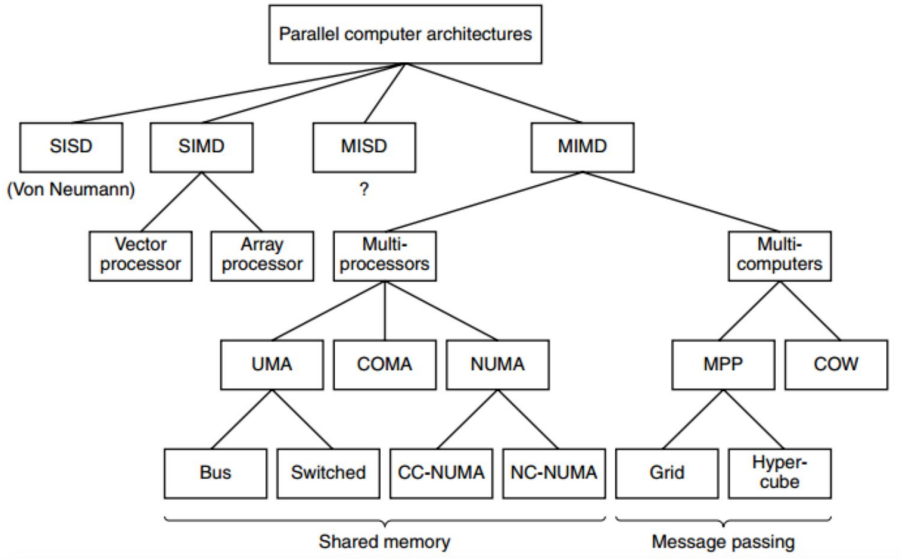
The memory consistency describe the last time in which we can perform an action on a single memory block of the system.

The idea is the following: two processes which want to operate on the same variable must perform the operations in critical sections with lock and unlock, so we can use these operations in order to obtain the last point in which all actions on these variables must be performed. I can use lock/unlock operation to identify the instant in which each process need shared data!

Is important to have at the beginning the problem in mind, now let's dive in details. This is a global vision of the problem and the solutions. We can find this topic on the book. This stuff is on Tanenbaum book in chapter 3.

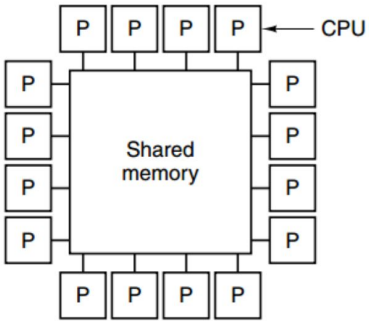
Multiprocessor architecture

Flynn taxonomy is based on two paradigm, stream of data and stream of instruction. We we talk about multiprocessor we talk about MIMD. MIMD architectures, multiple instructions on multiple data.



Multiple instructions executed means N CPUs attached to the memory. Any CPU can be attached to the memory in different way.

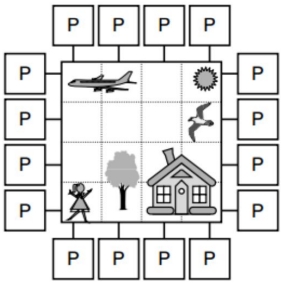
Multiprocessor



We have a single **shared memory** between all processors. Clearly the **LOAD/STORE** instruction are used to write and read inside the memory. This operation provide a sort of **communication** among processor. Clearly we have a single OS, a single scheduler and so on. The main advantage is that if we have a lot of tasks the OS (which is unique in this scheme) knows how to balance these tasks on all the processors that are around. Thanks to the single OS, we have a single memory map and a single process table.
If all the processors are similar among each other, and can all access IO devices independently, we call this schema Symmetrical Multi-Processor system (SMP).
If there is a special CPU which has privilege to access memory IO or has powerful resources, hardware acceleration mechanism and so on, we talk about Non Symmetrical.
We won't cover that

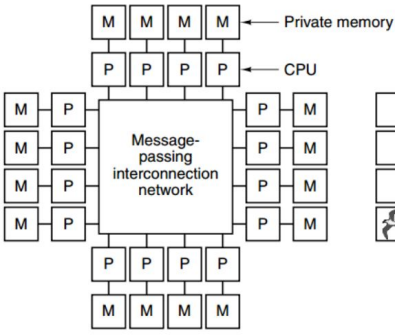
- Shared Memory
- LOAD/STORE instruction for communication
- One single copy of OS
- One single memory map and process table
- If no special CPU, Symmetric Multi-Processor system

Imagine an application that want to recognize images, various processor can communicate using the shared memory to access the same data!



In the example above each processor work on the same memory with other processor in order to recognize part of image!

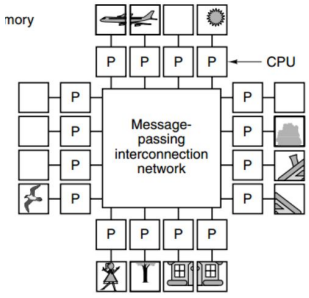
Multicomputer



If the private memory of processor are placed in different place for example, processor run with his own data. In the middle we have a channel that permit to CPU to talk among each other. This is a message passing paradigm (multicomputer). There are more OSes running on different processors, but in order to distribute this we need to have distributed process table and distributed memory model, so all CPUs can know all processes in order CPUs.

- Private Memory
- SEND/RECEIVE instruction for communication
- More copy of OS
- More Page Table and Process Table

If we use the same example of before we can see that each processor runs on his private memory and only later they can exchange the data using the network.



These implementations can have PROS and CONS.

Multiprocess:

- PRO: easy to program
- CONS: hard to build

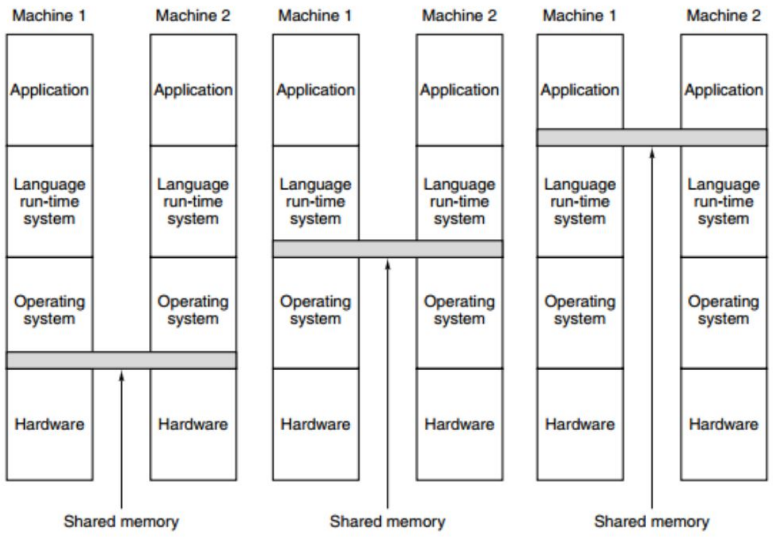
Multi-Computer:

- PRO: easy to build
- CONS: hard to program

The multiprocessor is easy to program, write/read and the balance of the task is handled by the OS. Difficult to manage (build) the shared memory efficiently.

The multi-computer are easy to build, we can assume are different computer attached with a network! But are hard to program, we have to imagine implementation of system with send/receive and balance the workload in this way is difficult.

To gain pros from the two paradigm we can use an hybrid system! We can have hybrid systems, in which there are shared memories among some processors, but not all. Processors are divided in sort of groups. However load and store instruction are not performed in hard way, they are translated in communication over the network.



Handle caching in multiprocessing architecture

First we will focus on multiprocessor. We have shared memory.
We said we have two main property we must maintain: **coherence** and **consistency**.

Coherence:
When we have data placed on different memory we have to guarantee that access to same address on private memory return the same data! Which one of all the copies has to be accessed? This is defined by coherence property. So coherence defines which values can be returned by a read operation.

Consistency:
Defines when a value that has been written will be returned by a read operation. If a processor modifies a variable, other the locations are not valid anymore, so the other processors would like to acquire the new value. But when can we update this?

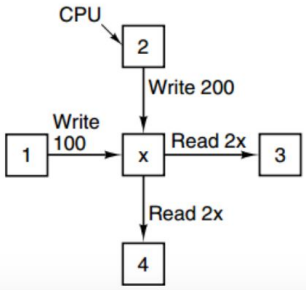
Informally, we talked about coherence memory system if any reading of a data provides the value of the written data more recently

Memory consistency

Consistency is a sort of contract between processors and memory, we have different kinds of consistency.

Strict consistency: any read to a location x always returns the value of the most recent write to x. Hard to implement in a multiprocessor architecture, since memory have to be managed in a fifo way.

Sequential consistency: in the presence of multiple read and write requests, some interleaving of all the requests is chosen by the hardware (nondeterministically), but all CPUs see the same order. Take this example, 4 CPUs, CPU1 and CPU2 perform 2 write operations on the same variable and CPU3 and CPU4 perform 2 read operations.There are then many kinds of instruction orderings which could be good from a sequential consistency point of view, even if the final result is different among these different orderings.



W100	W100	W200
W200	R3 = 100	R4 = 200
R3 = 200	W200	W100
R3 = 200	R4 = 200	R3 = 100
R4 = 200	R3 = 200	R4 = 100
R4 = 200	R4 = 200	R3 = 100

Processor consistency: Writes by any CPU are seen by all CPUs in the order they were issued. And, for every memory word, all CPUs see all writes to it in the same order. The local order of operations is guaranteed, but not the global ordering

Weak consistency: no order in write operation but synchronization point to avoid chaos.

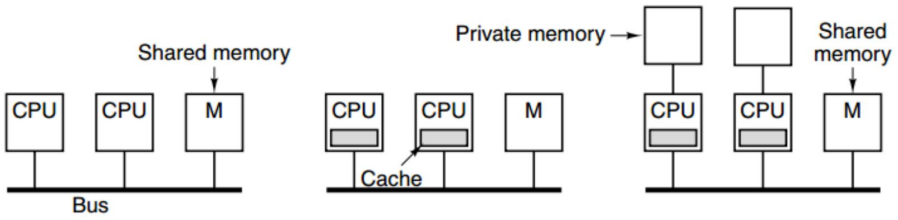
Release consistency: write before a release are performed before a new acquisition. The write operations on a critical section have to be performed before any new acquisition of the same resources.

Notice that different processors may have now different visions of the same memory block. Each processor must have however a consistent vision of the memory blocks for which he uses lock and unlock operations.

Uniform memory access

Now we talk about multiprocessor we can divide into another two classification. The first is UMA (Uniform Memory Access). We talk about UMA when all the processors have the same

time in order to access the memory. In the slide we can find three different implementation of UMA system.



- Two processor that don't have cache and a shared memory with a bus
- Two processor with its own private cache and a shared memory with a bus
- Two processor with its own private cache, each own private memory and a shared memory with a bus

Now let's focus on snooping protocol. The snooping protocol is used to maintain coherence of cache. How can we guarantee that a processor read the latest and correct value of some data? If we have the second scheme, how can we guarantee that a process accessing the memory accesses the latest write on a shared variable?

Snooping protocol

In **snooping**, each cache "listens" to the writes in main memory, so each process performs a write operation in write through so that the controller can "snoop" the write itself.

Action	Local request	Remote request
Read miss	Fetch data from memory	
Read hit	Use data from local cache	
Write miss	Update data in memory	
Write hit	Update cache and memory	Invalidate cache entry

If in cache we have a **read miss** we have to take the variable from the shared memory, so if the shared memory is consistent with the last write we are fine.
If we have a **read hit** in the cache we don't have to perform any subsequent read request to memory.

For **write miss** we have two cases.
If data is not present in local cache and not present in other caches a write in main memory is fine.
If data is not present in local cache and present in other caches, a write in main memory is fine, as long as other caches listen to the write on the bus.

For **write hit**, if data is present in local cache and present or not present in other caches a write in local cache could be fine or not, but since we can't know this for sure we need to

write in main memory, so that the other caches are able to invalidate if needed the local outdated copy.

So, all the writes send a request to the bus (write-through), in order for the snooping controllers to invalidate local copies. All reads can be performed as normally, since if a location is valid in a local cache its value is good.

MESI protocol

This is another protocol used in order to guarantee the coherence of memory. It is called MESI protocol. The difference is that here the operation are performed on local memory and not in the shared memory so every request are not sent to the bus.

Before talk about details we have to introduce some definitions about blocks that we will use:

- **Modified**: the entry is valid; memory is invalid; no copies exist.
- **Exclusive**: no other cache holds the line; memory is up to date.
- **Shared**: multiple caches may hold the line; memory is up to date.
- **Invalid**: the cache entry does not contain valid data

At the beginning, no cache block is valid. Then a block can become exclusive on the first access from a processor, and then shared if other processors access in read to that value.

Read operations of other processors can be snooped by the caches and then they update the state of that block as “shared” from “exclusive”.

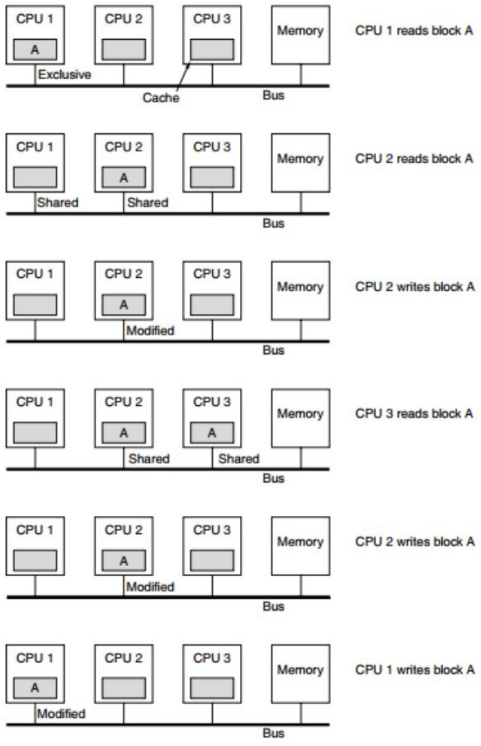
When a CPU wants to modify a block, he have to send a request to other CPU to inform them that their copies are invalid. The state of the block in these caches become “invalid” and the local cache has instead a state “modified” for the block.

If then a cpu wants to read that block, that read request can be intercepted by the cache controller by the cpu which has the “modified” block, which is the latest version of the block and can therefore send the correct value to the cache that requested it.

They both set then the state of the block as “shared”, while the main memory still has a “invalid” flag on that block.

The read operation is accepted by the cache of another processor, instead by the main memory.

So we can see on slide a simple example.



At the beginning, no cache block is valid. Then the block A can become exclusive on the first access from CPU1.

Now CPU2 reads block A and in this case the CPU2 send a read operation on the bus that can be performed by the cache controller of CPU1. Cache controller of CPU1 listen to the read request and change the state of the block from exclusive to shared.

Now assume that CPU2 want to modify the block A, he have to sent a request informing the other CPU that the block is invalidated. The state change from shared to modified. The state of the block in these caches become “invalid” and the local cache has instead a state “modified” for the block.

If then another CPU wants to read that block, for example CPU3, that read request can be intercepted by the cache controller by the CPU2 which has the “modified” block, which is the latest version of the block and can therefore send the correct value to the cache that requested it.

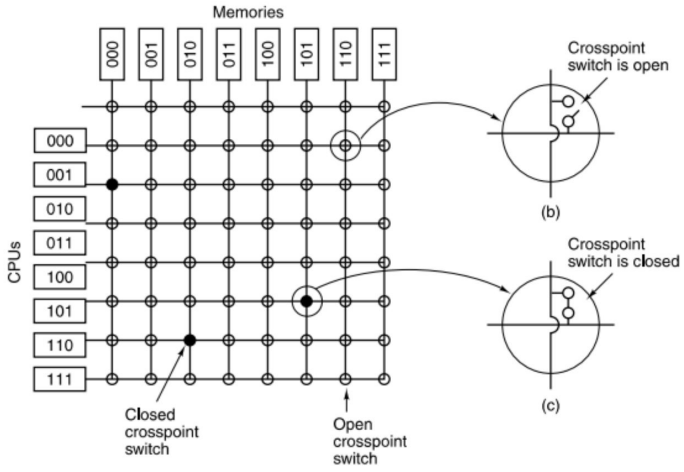
Now CPU2 sent the block to CPU3 and the state is shared. The read operation is called by CPU3 but in reality is performed by CP2 that send data!

When the memory is updated in MESI protocol implementation? The block in shared memory is updated only when the block will be deleted on the local cache! This is a write back implementation, memory is updated only when the block in the cache will be deleted (due to interference between cache blocks and so on), so if a processor wants to read a

value it may find an outdated value in the main memory. In order to avoid problems, notice that in this implementation, if processor want to perform a write operation, he must evaluate if the block is exclusive or shared. If the block is shared (implemented in hw with two bit in cache), the processor must modify the block only after sending a request on the bus that impose to other processor that have the same block in their cache in order to invalidate the data. The current cache has to be the sole owner of the block it wants to write. In this case, the main memory contains of course outdated versions of the data blocks, so caches need to intercept read operations that involve modified blocks that they own.

UMA implementation

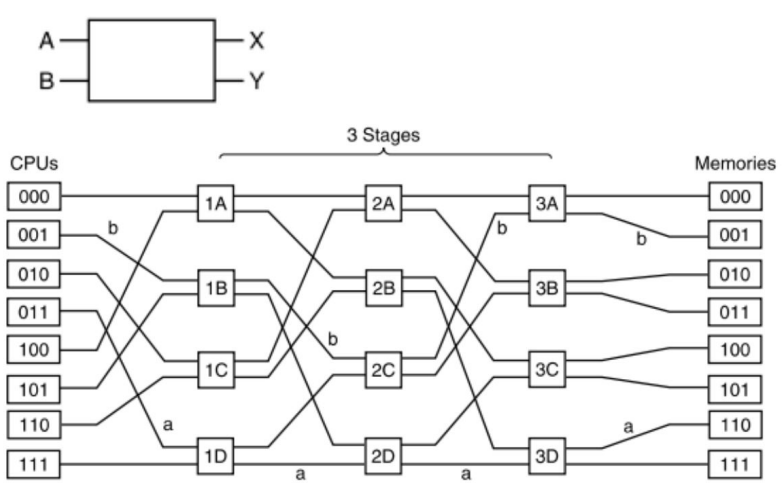
Now let's return to the implementation of UMAs computers. Usually are implemented with **crossbar interconnection**. Processors have to be connected with memories by "cross points" in a matrix, which can be opened or closed.



The complexity of this implementation is the number of crosspoint. If we have N cpu and K memories, the number of crosspoint is N * K. But the advantage of this implementation is that we have no competition when a CPU want to access to one memory!

We have 2^{NK} different combinations. But how many combinations won't cause interference? We have to deal with combinations! So N!/K!.

The complexity of the implementation is in the switch managing.The complexity is evaluated in the number of switches.

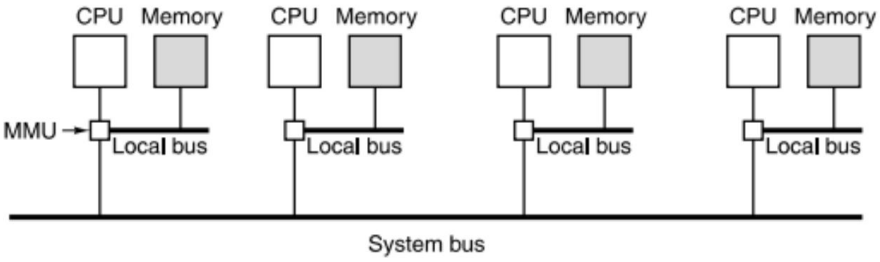


Switch	CPU	MEM
1A	X00	XXX
1B	X01	XXX
1C	X10	XXX
1D	X11	XXX
2A	XX0	0XX
2B	XX1	1XX
2C	XX0	0XX
2D	XX1	1XX
3A	XXX	00X
3B	XXX	01X
3C	XXX	10X
3D	XXX	11X

An omega switching network can connect CPUs and memories via a 3 stages switch, using a overall lower number of switches. The complexity of these networks is (N cpu N memories) Log2(N) levels and N/2 switches in any level. We may still have interference between requests if they require the same switches.

NUMA (Non Uniform Memory Access)

Now as last topic we will talk about NUMA architecture. Numa stands for Non-Uniform Memory Access. In the UMA we saw that the access time spent by each processor to acquire data is almost the same. Clearly can be interference but the difference of time access are not so big. In NUMA we have completely different access time. In this case the memory is not private for each processor. All the memory are shared among CPUs. Each processor see its memory and other processors memory. In this scheme, still all CPUs see a global address space, load and store operations are used. Also there are local and global bus. However, access to remote memory is slower than accessing local memories.



We have 2 buses: a local and a system bus. A cpu uses a local bus to access the local memory and global bus to access directly memories in other processors. Of course, access to memories different from the local one are performed in a much greater time than accesses in local memories, due to the competition to access the global bus and memories of other processors. This architecture is seen as multiprocessor because at the end the memory is shared.

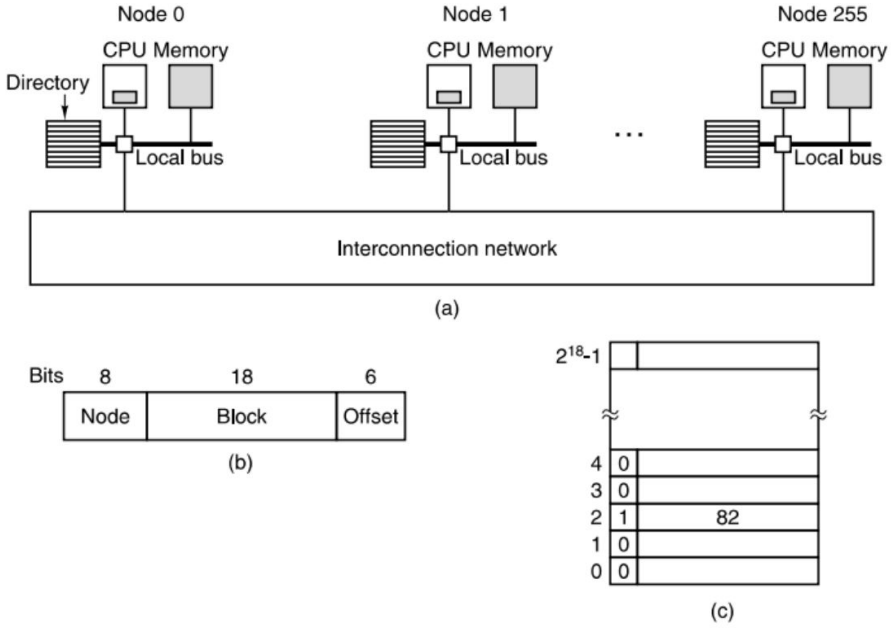
1. There is a single address space visible to all CPUs.
2. Access to remote memory is done using LOAD and STORE instructions.
3. Access to remote memory is slower than access to local memory.

The difference of timing access regards the fact that if some CPU wanna read a data that is local the request is executed fast, but if the CPU have to send a request to the system bus in order to get another CPU data access time increase a lot.

In this architecture the main problem is the execution time of a process is not predictable! A problem for these architectures is the non repeatability of the software running on it. We can experience completely different time execution.
Also the idea is that we want to access the nearest or the best memory that contain required data. There is an algorithm that is thought for this reason. Of course, if a block from a memory is accessed often by another processor, we can move that block also to the local memory of the other processor, to speed up subsequent operations. Many protocols are used to do so, but we won't see it.

Coherence in NUMA architectures

But how can we maintain the coherence in a NUMA architecture? We have a lot of memory that contains data. A snoopy algorithm is not useful on this type of architecture because we can't implement a cache controller that listen to the bus! The system bus often is not used!



The solution used in order to maintain coherence between memories, is to create a directory (a table in memory) that contains information of all the blocks of the memory.

A directory based multi-processor protocol informs the CPU of the state of its own memory. For each block contained in the local memory stores if the block contains the latest version of the block or if the latest value is contained in the cache of another processor.

The write operation is implemented as write-through, the memory is not updated, but caches among processors are updated in write-through.

The directory will reply to a read operation with either:

- the block of the cache if the latest version is present
- the number of the node that contains the latest version is the current version is outdated

MMU in this case are used only when a data is not present in the cache of a cpu (read or write miss). The MMU reads the address of the data that we want to read/write and it reads the node tag of the directory in order to establish where to find the data.

If the data is present in the local memory it sends the request to the local bus. In the other case it sends the request to the system bus.

Let's see an example:
At the beginning a memory of 1MB is used. Each processor have is own 1MB. The first CPU has the first 1MB, the second the second 1MB and so on. Now let's suppose the first CPU want to access to an address. The address is divided in three parts, the first indicate the node, the second the block numb and the third the offset. …..

Lesson 13 - 04/05/2017

MESI protocol - a brief review

To better understand the MESI protocol, we can see the table on the slide! MESI stands for Modified (M), Exclusive (E), Shared (S), Invalid (I) that are the possible states of a block in cache!

- **Modified**: the entry is valid; memory is invalid; no copies exist.
- **Exclusive**: no other cache holds the line; memory is up to date.
- **Shared**: multiple caches may hold the line; memory is up to date.
- **Invalid**: the cache entry does not contain valid data

First, let's say that the MESI protocol is a write-back protocol so we need to update the memory in case of write!

Suppose now we have a shared block and suppose we wanna read that block. For sure we have a hit, the block is present! But if I want to modify I want to invalidate all blocks of the other CPU because I modify the block. I send on the bus a MISS. If the other CPU receive a MISS must invalidate the block involved. After this the block can be modified and is put in Modified state. So the block can be modified directly if it is exclusive but is is shared (some other CPU have the same block in caches) I have to inform the other first.

Let's start from the beginning. All the cache blocks are valid. When a processor execute the first access (read operation) a block can become exclusive. Then, if other processors access (read) the same block, it becomes shared. Read operations of other processors can be snooped by the caches and then they update the state of that block as **shared** from **exclusive**. When a CPU wants to modify a block, he has to send a request to other CPUs to inform them that their copies become invalid. The state of the block in these caches become **invalid** and the local cache has instead a state **modified** for the block.

If then a cpu wants to read that block, the read request can be intercepted by the cache controller of the cpu which has the modified block (latest version of the block). This CPU can send the correct value to the cache that requested it. Then, both of them set the state of the block as shared, while the main memory still has an invalid flag on that block. The read operation is accepted by the cache of another processor, instead by the main memory.

MESI Protocol State Transition Table

State	*This Processor* Load	Store	*Other Processor* Load Miss	Store Miss
Invalid (I)	Miss → **S or E**	Miss → M	---	---
Shared (S)	Hit	Miss → M	---	→ I
Exclusive (E)	Hit	**Hit → M**	Send Data → S	Send Data → I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

MOESI Protocol

The only difference is the owner block.

- **Owner Block**
  - Multiple caches can have the same block but the memory is not updated.

The owner block owns the updated version of the block, while the memory has an outdated version.

	M	O	E	S	I
M	✗	✗	✗	✗	✓
O	✗	✗	✗	✓	✓
E	✗	✗	✗	✗	✓
S	✗	✓	✗	✓	✓
I	✓	✓	✓	✓	✓

State	This Processor Load	Store	Other Processor Load	Store
I	Miss -> S,E	Miss -> M	-	-
S	Hit	Miss -> M	-	→ I
E	Hit	Hit -> M	Send Data -> S	Send Data -> I
M	Hit	Hit	Send Data -> O(S)	Send Data -> I
O	Hit	Miss -> M	Send Data	Send Data -> I

MSI Protocol

In this case we have only:

- **Invalid block**
  - No valid data) contained
- **Shared block**
  - The block can be in one or in multiple caches and the main memory contains the value contained in this cache
- **Modified block**
  - The entry is valid while the value in memory is not valid anymore and we have no copies in other caches.

MSI Protocol State Transition Table

State	*This Processor* Load	*This Processor* Store	*Other Processor* Load Miss	*Other Processor* Store Miss
Invalid (I)	Miss → S	Miss → M	---	---
Shared (S)	Hit	Miss → M	---	→ I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

	M	S	I
M	✗	✗	✓
S	✗	✓	✓
I	✓	✓	✓