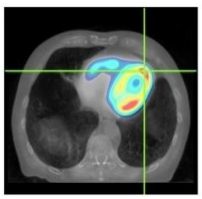
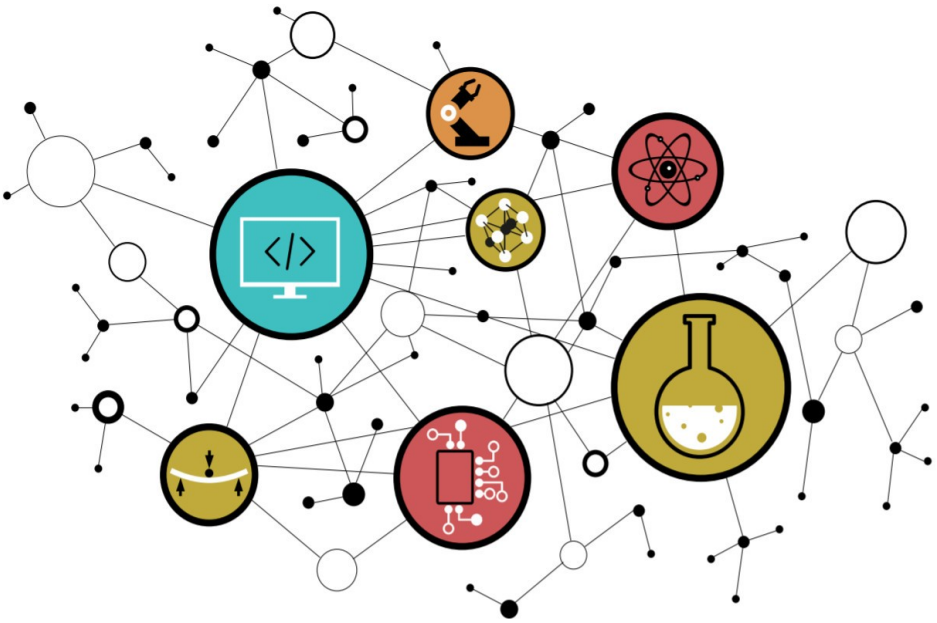
ARCHITECTURES POUR LE CALCUL

COURS





CONTACTS

Équipe enseignante

hugo descoubes - COURS
hugo.descoubes@ensicaen.fr
+33 (0)2 31 45 27 61

Isabelle Lartigau
isabelle.lartigau@ensicaen.fr

Emmanuel Cagniot
emmanuel.cagniot@ensicaen.fr

ENSICAEN
6 boulevard Maréchal Juin
CS 45053
14050 CAEN cedex 04

RESSOURCES

Les différentes ressources numériques sont accessibles sur la plateforme pédagogique de l'ENSICAEN. Télécharger l'archive complète de travail **opt.zip**

https://foad.ensicaen.fr/course/view.php?id=117

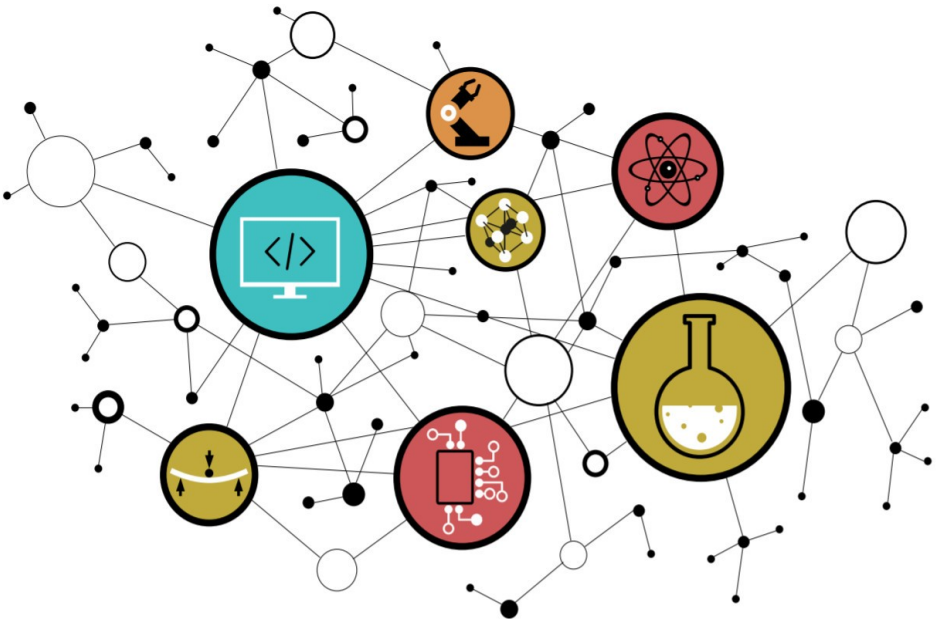
ÉVALUATION

- Examen de pratique sur ordinateur (1h30)

L'évaluation de la compétence se fera sur machine personnelle ou machine école et portera sur les points suivants :

- Création d'un projet sous IDE CCS. A l'image du projet présent dans cm/eval/examen_nom
- Optimisation d'une fonction algorithmique élémentaire (cf. trame de TP)
  - écriture en C canonique
  - écriture ASM C6000 canonique
  - écriture ASM VLIW
  - écriture de l'algorithme optimisé avec l'une des techniques avancée suivante :
    - Vectorisation en langage C par programmation intrinsèque
    - Pipelining software en ASM C6000
    - Vectorisation en base 2 ou 4 en ASM C6000

DIVERSITÉ DES ARCHITECTURES PROCESSEURS



Chapter 1

Diversity of Processor Architectures

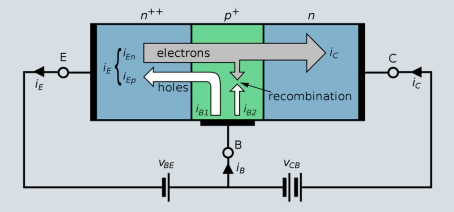
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2021-2022

Quick reminder

1947: Invention of the Bipolar Junction Transistor →

by Bardeen, Schokley and Brattain (Bell labs), Nobel Prize winners

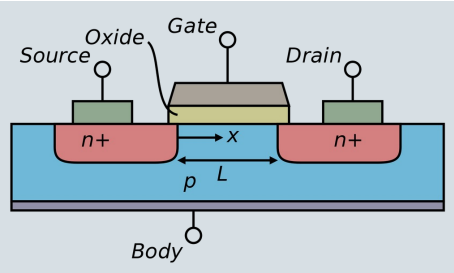


1958/1959: Creation of Integrated Circuits

by Texas Instruments (hybrid IC), then Fairchild (true monolithic IC)

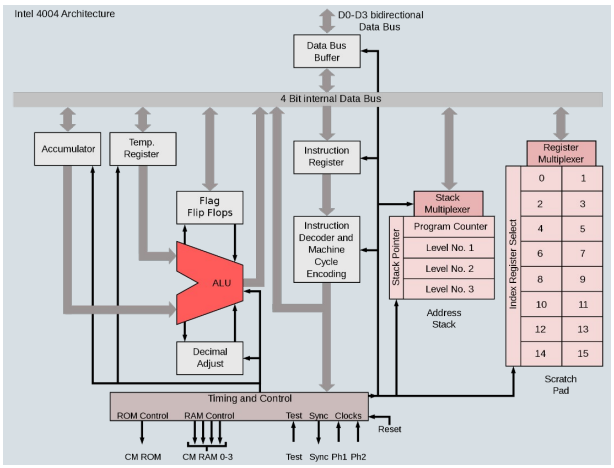
1960: Invention of the MOS Field-Effect Transistor →

by Mohammed Atalla and Dawon Kahng



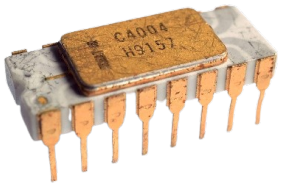
The first ever commercialised processor is the Intel 4004 in 1971.

It has 2,300 transistors with a 10 µm etching process (4-bit processor, 16 pins, 740 kHz, 90 kIPS or kilo-Instructions Per Second).

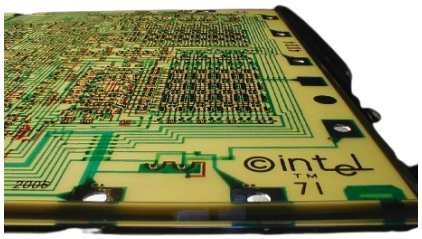


Intel 4004 architecture

Intel 4004 integrated circuit

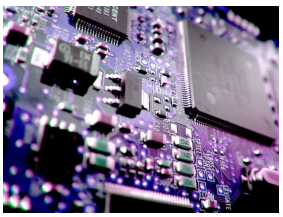


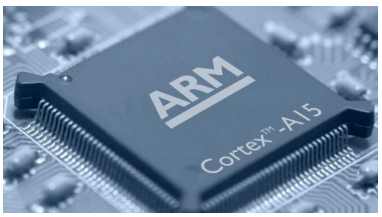
Intel 4004 die

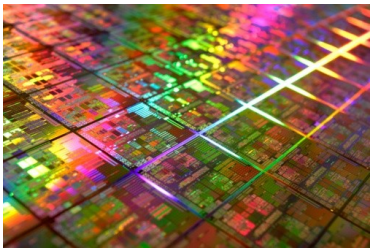


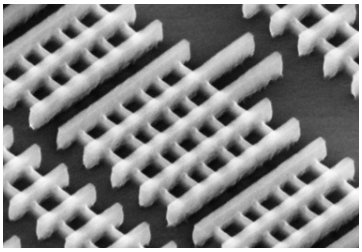
Ever since, processors have evolved following natural selection.

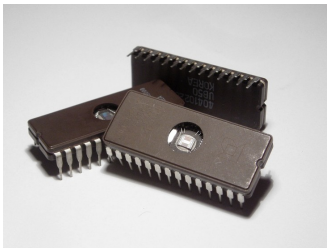
Those that matched specific needs improved while others disappeared from markets and research labs.

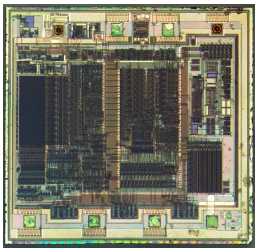












As for animals and plants, the evolution process of processors is never-ending. New processor architectures are likely to born in the next few years!

Let's take a look at the current processor architectures.

MCU **AP** **GPP** **SoC / SoB** **FPGA** **DSP** **(GP) GPU**

General architecture
Control processors

Hybrid architectures

Specialised architectures
Coprocessors or Calculus processors

MCU	AP	GPP	SoC / SoB	FPGA	DSP	(GP) GPU
Micro Controller Unit	Application Processor	General Purpose Processor	System on Chip / Board - FPGA-AP - FPGA-MCU - GPP-GPU - AP - MCU-analog	Field Programmable Gate Array	Digital Signal Processor	Graphics Processing Unit General Purpose GPU

Computer

CONTROL **CALCULUS**

General architecture
Control processors

Hybrid architectures

Specialised architectures
Coprocessors or Calculus processors

MCU	AP	GPP	SoC / SoB	FPGA	DSP	(GP) GPU
Micro Controller Unit	Application Processor	General Purpose Processor	System on Chip / Board - FPGA-AP - FPGA-MCU - GPP-GPU - AP - MCU-analog	Field Programmable Gate Array	Digital Signal Processor	Graphics Processing Unit General Purpose GPU

CPU **LOGIC** **CPU**

MCU
MICROCONTROLLER UNIT

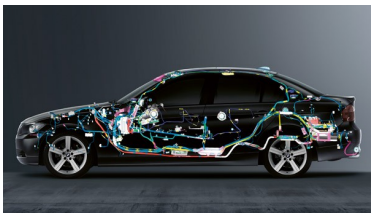
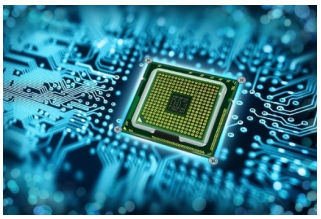
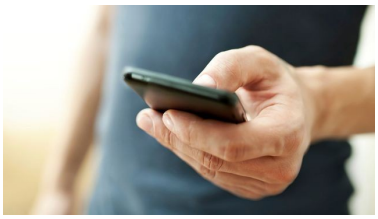
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Applications
Architectures
Designers and products
Market shares

Applications

MCUs (Microcontroller Units, fr: *micro-contrôleurs*) are the most common processors in our environment (talking about quantity).

We use about 200 processors every day, without even being aware!



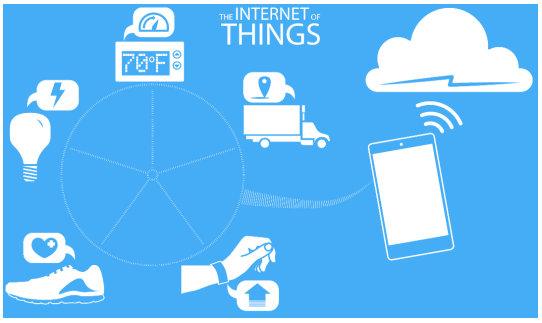


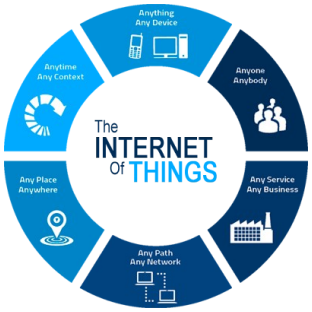
MCUs are control processors that are dedicated to the supervision of electronic processes. They control their input/output interfaces with their application-custom embedded firmware.

They aim for markets applications that require low-cost, low-consumption, small-size, and big production volumes.

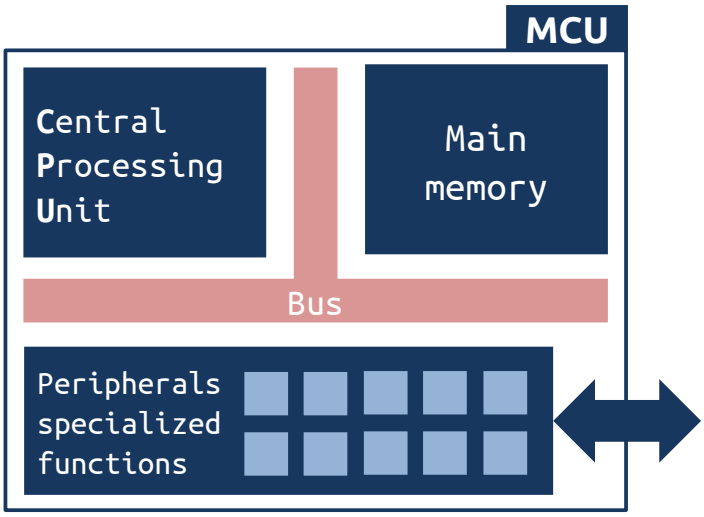
The IoT (Internet of Thing, fr: *objets connectés*) is the major market for MCUs. The IoT is the Internet extension to physical world objects and places. It is considered as the third Internet evolution and has been therfore named « Web 3.0 ».

With 3.6 billions of active connections in 2015, 11.7 billions in 2020 and 30 billions planned in 2025, the IoT counted for 18% of MCUs population in 2019 and will be around 29% in 2025.



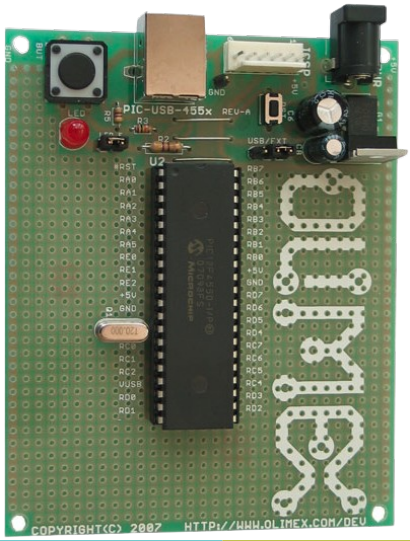


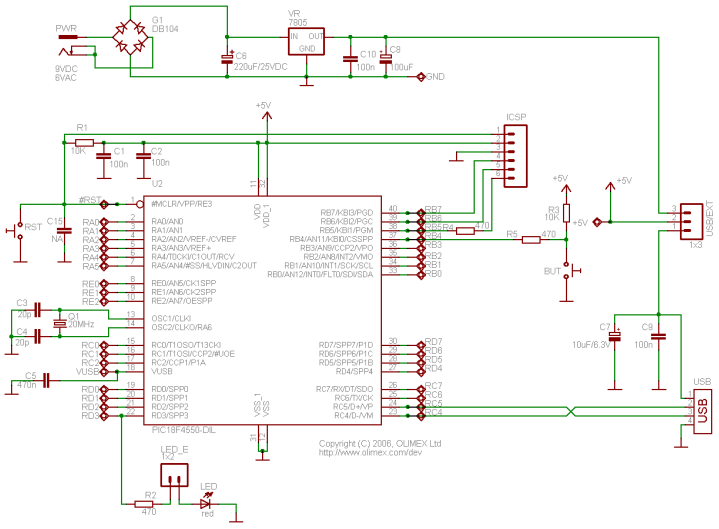
MCU processors are digital systems integrated onto an Integrated Circuit.
They are designed to be stand-alone (no need for external RAM, HDD …).



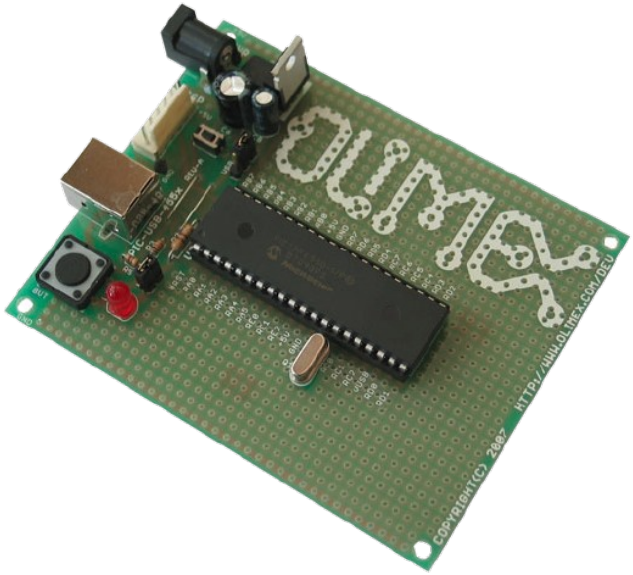
Example of a schematic that uses a Microchip's PIC18 MCU.

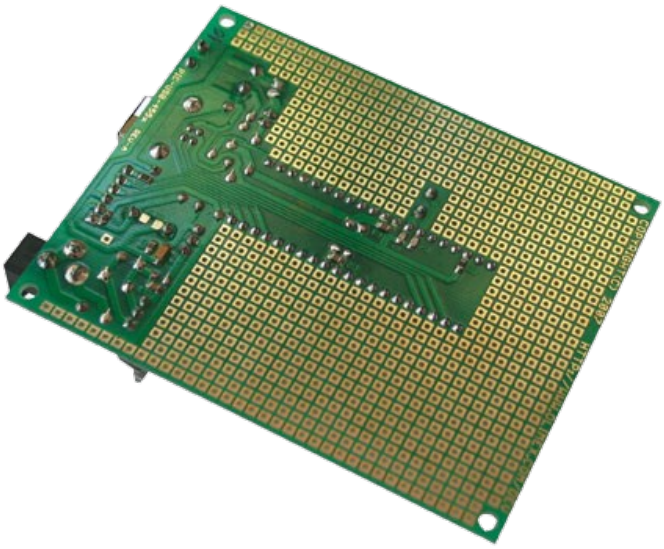
Olimex PIC-USB-4550 board.





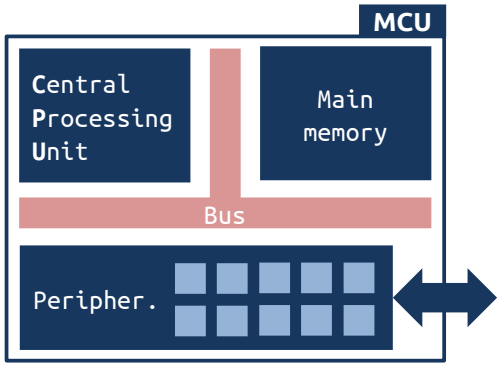
Exercise: link these board devices to the schematic in the previous slide.



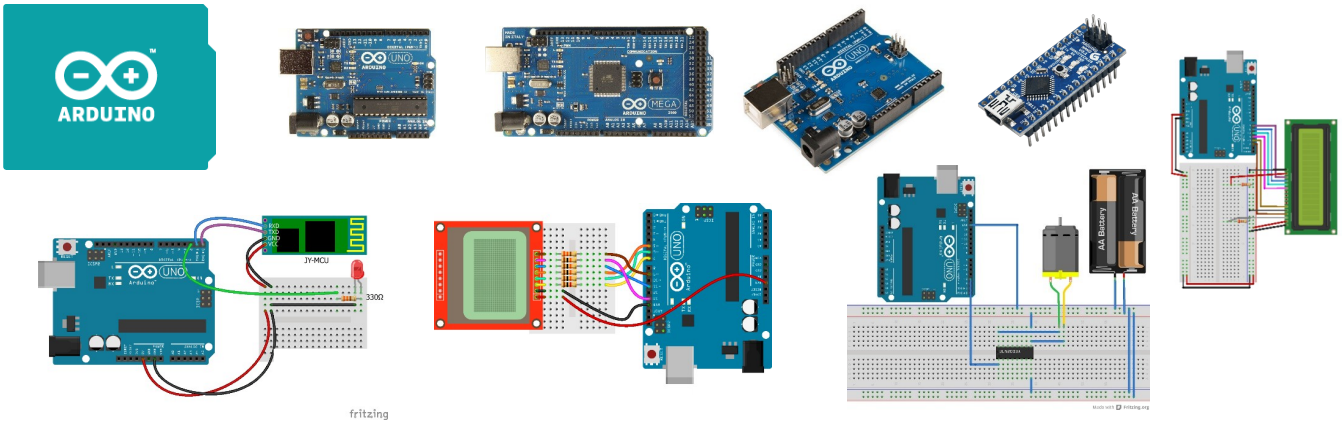


There is a big number of MCU products from various designers and foundries, each made for different uses.

MCUs from the same family possess the same CPU and associated buses. The ISA (Instruction Set Architecture, fr: *jeu d'instructions*) and the toolchain are therefore similar. The difference between same-family MCUs resides in the peripherals set and the memory resources.



The Arduino project is certainly the most famous MCU-based electronic project. However it is too user-friendly (too magic, too many hidden things) and is not used in professional environments, which is why it is not studied in engineer schools.

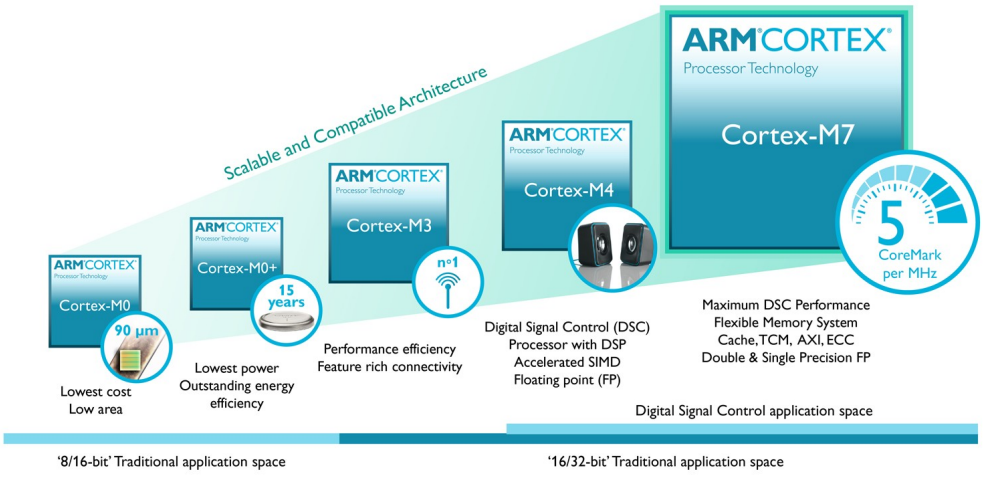


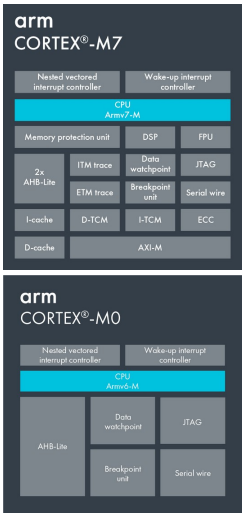
Even though the MCU market is very competitive, the vast majority of MCU founders (e.g. STMicroelectronics, Renesas, Texas Instruments, NXP, …) use similar CPU architectures: the Cortex-M family, designed by the British company ARM

This guaranties an access to reliable development tools, libraries and software services. Some tools can also be open-source (IP / Graphical / USB / Bluetooth, stack, RTOS, …).

ARM offers the Cortex-M series, with 'M' standing for "MCU".

This includes a whole family of MCU cores that are suitable for a wide range of applications.

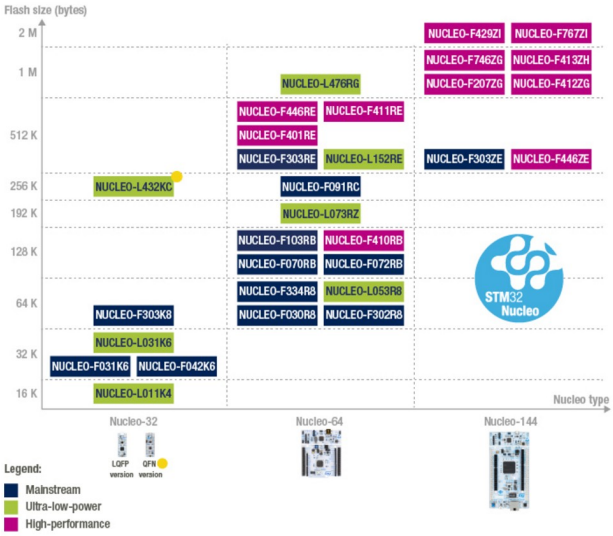


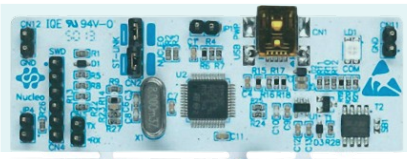


As an example let's take a look at the range of STM32. Those are 32-bit MCUs based on a Cortex-M core.

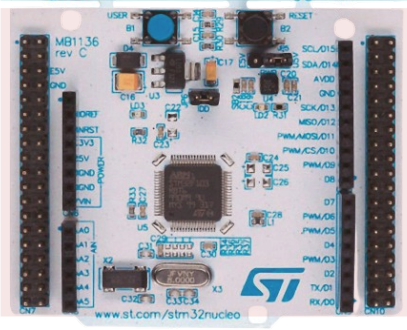
They are designed by the French-Italian company STMicroelectronics, which also is the main European manufacturer.

The STMicroelectronics Nucleo project offers low-cost (≈ €10) evaluation boards that use ARM-based MCUs and industrial development tools.





-Power supply
-Programmer (JTAG emulator)



-Target MCU
-Switch and LED
-External ports
-Shields connectors
-Arduino shield connectors

Nucleo-64

Let's take a look at an annual markets study.

ASPENCORE

2019 Embedded Markets Study

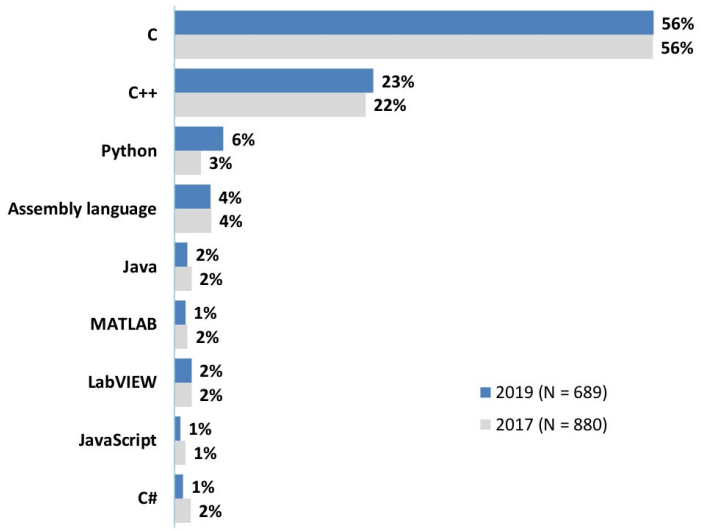
Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments

March 2019

Presented By: EETimes embedded

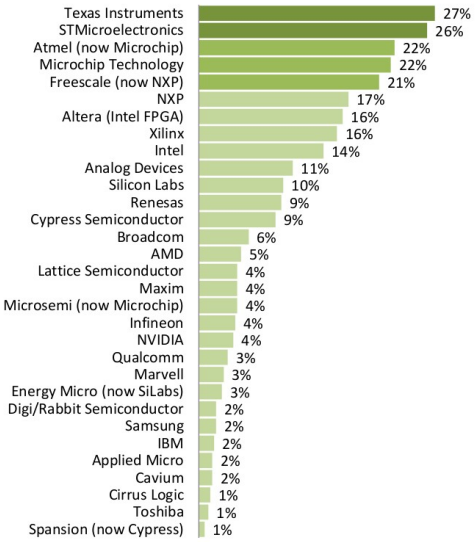
Market shares

My *current* embedded project is programmed mostly in:



Market shares

Please select the processor vendors you are currently using.

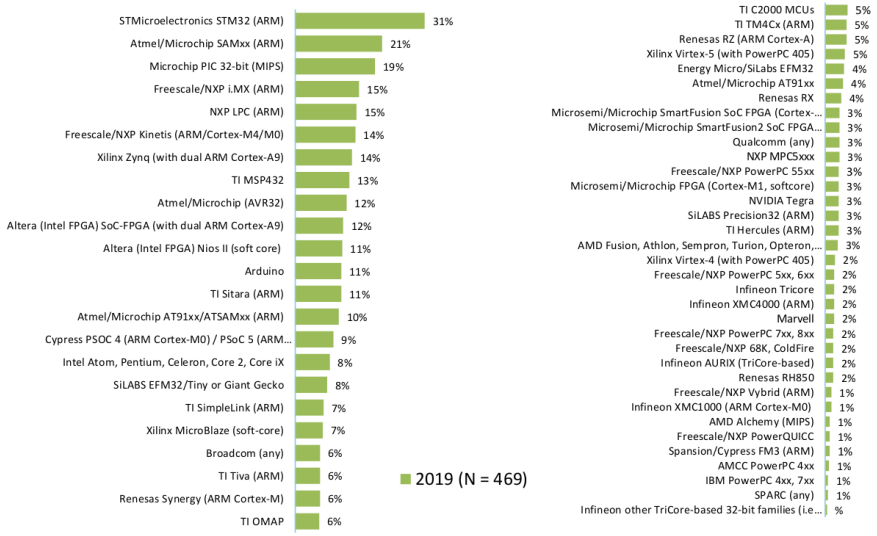


Merged Brands Combined	%
Microchip/Atmel/Microsemi (Net)	40
NXP/Freescale (Net)	28
Intel/Altera (Net)	26
Silicon Labs/Energy (Net)	10
Cypress/Spansion (Net)	9

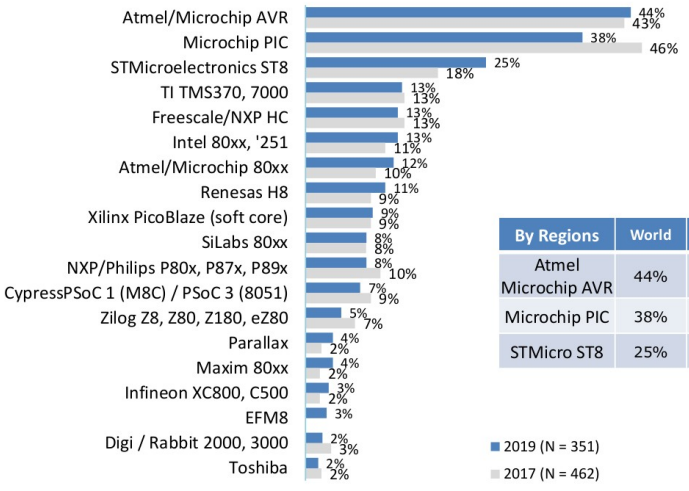
Top Four Brands by Region:
Americas: TI, Microchip, STMicro, Atmel
EMEA: STMicro, NXP, TI, Atmel
APAC: TI, Atmel, Freescale, STMicro

2019 (N = 458)

Which of the following 32-bit chip families would you consider for your next embedded project?



Which of the following 8-bit chip families would you consider for your next embedded project?



By Regions	World	Americas	EMEA	APAC
Atmel Microchip AVR	44%	44%	**52%**	39%
Microchip PIC	38%	41%	**43%**	**23%**
STMicro ST8	25%	22%	**31%**	28%

GPP
GENERAL PURPOSE PROCESSORS

Applications
Architecture
Motherboards
Superscalar processor

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Applications

GPP (General Purpose Processors) have a complex CPU architecture that gives them a great adaptability especially for executing non-optimised programs.

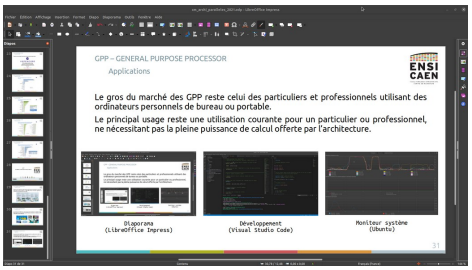
Most of the time, those programs contain sequential code with a lot of tests and function calls, which are difficult to accelerate.

```
444         prev = NULL;
445         for (mpnt = oldmm->mmap; mpnt; mpnt = mpnt->vm_next) {
446                 struct file *file;
447
448                 if (mpnt->vm_flags & VM_DONTCOPY) {
449                         vm_stat_account(mm, mpnt->vm_flags, -vma_pages(mpnt));
450                         continue;
451                 }
452                 charge = 0;
453                 if (mpnt->vm_flags & VM_ACCOUNT) {
454                         unsigned long len = vma_pages(mpnt);
455
456                         if (security_vm_enough_memory_mm(oldmm, len)) /* sic */
457                                 goto fail_nomem;
458                         charge = len;
459                 }
460                 tmp = kmem_cache_alloc(vm_area_cachep, GFP_KERNEL);
461                 if (!tmp)
462                         goto fail_nomem;
463                 *tmp = *mpnt;
464                 INIT_LIST_HEAD(&tmp->anon_vma_chain);
465                 retval = vma_dup_policy(mpnt, tmp);
466                 if (retval)
467                         goto fail_nomem_policy;
```

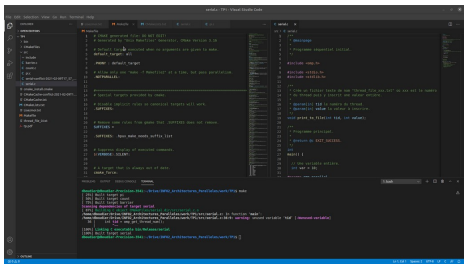
root/kernel/fork.c - www.kernel.org

Their target market are personal and professional computer and laptops.

Thus their main usage is for general applications (i.e. not specific) for personal and professional uses. Most of the time that does not require all the computing power that is really available



Slideshow
(LibreOffice Impress)

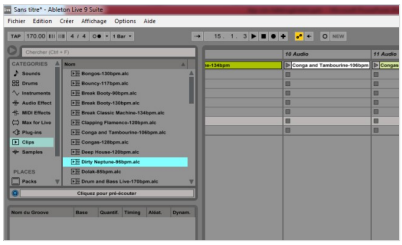


Development
(Visual Studio Code)

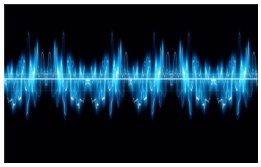
System monitor
(Ubuntu)

Of course some applications are likely to need full capability of the hardware, even though they are not the most common ones.

One can think of audio, image and video processing or software development as well-known examples.



Audio editing (Ableton)



Audio processing

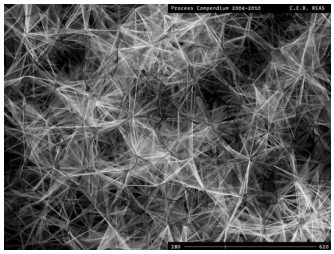


Image processing

Industrial applications are a historical part of GPP uses.

They are typically encountered on control tasks or specialised calculus functions. This market tends to use integrated solutions, such as AP (Application Processor), SoC (System on Chip), DSP (Digital Signal Processor), FPGA (Field Programmable Gate Array) ...

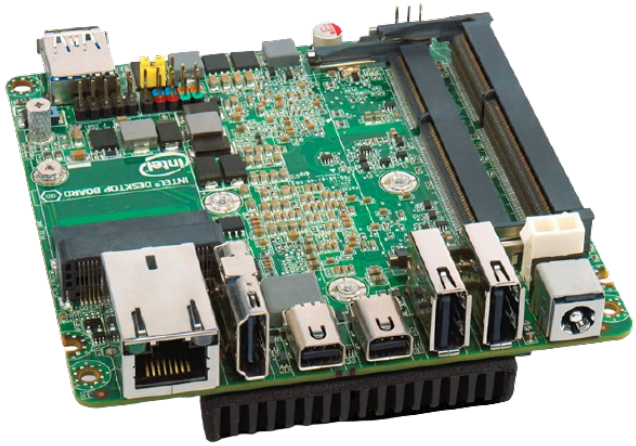
Radar GM400
(Thalès)

Rafale
(Dassault)

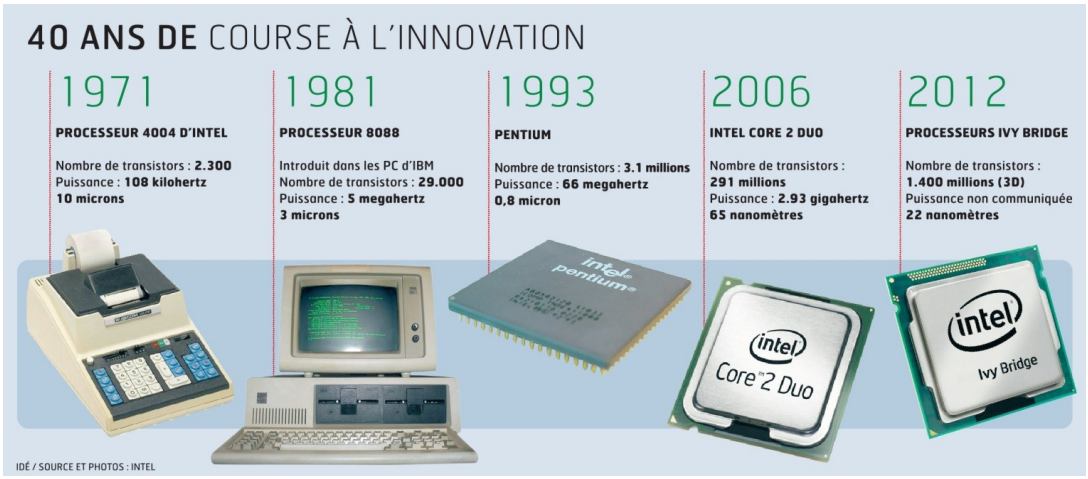
Automatic bollard
Box j200

Please note that GPPs can also be used in embedded systems applications.

For instance this is the NUC Core i5, an Intel motherboard.

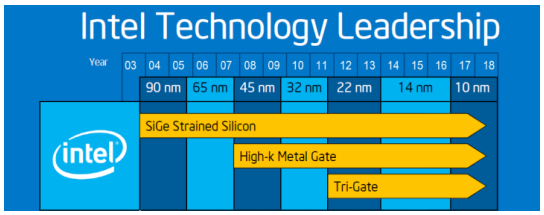


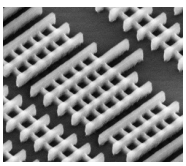
Let's have a look on major Intel architectures. Note that Intel is the historical and current leader of GPP market, but it is also the leader of semi-conductors market.

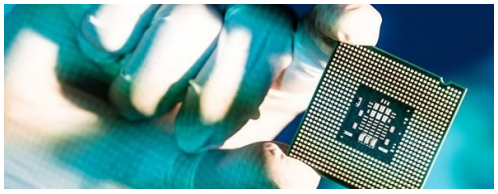


Today's leading GPP architectures ar the Intel Core i3/i5/i7/i9 families.

However there are many other actors and manufacturers aiming for different markets.

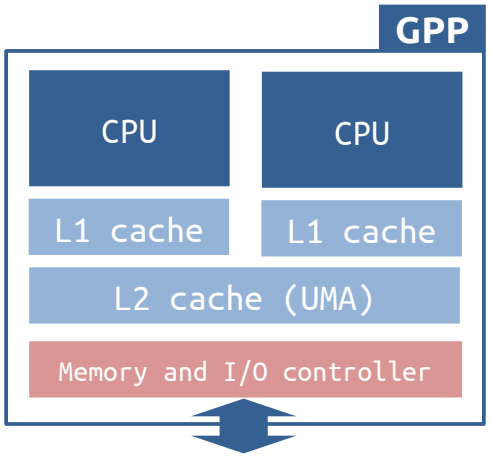




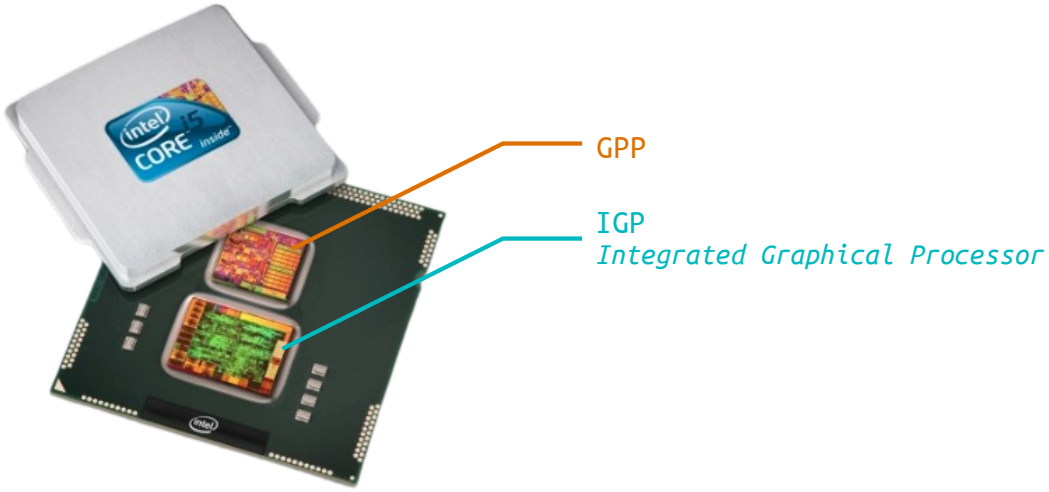


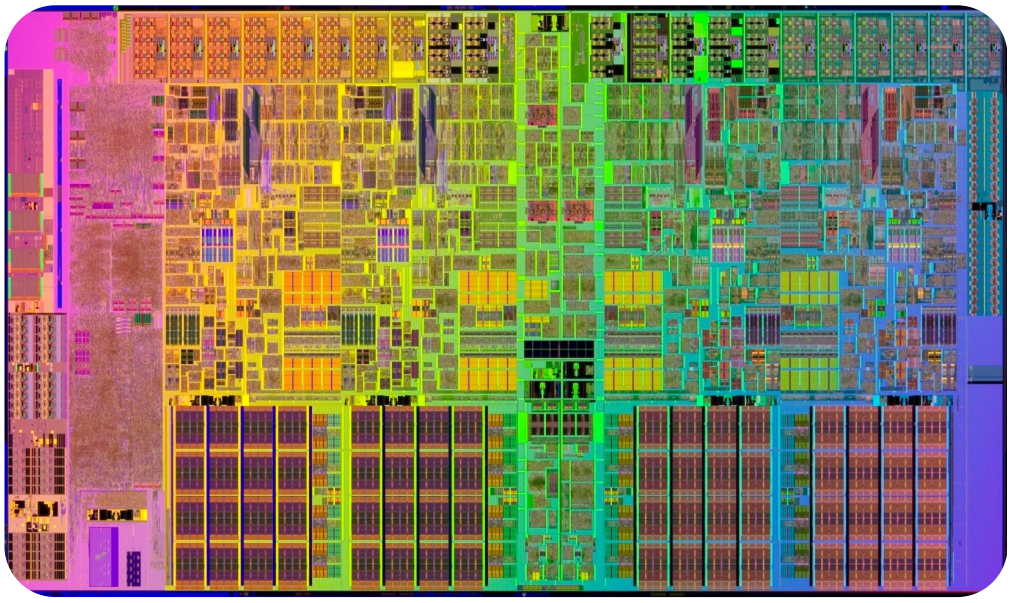
A GPP consists of a are processing element, with no main memory.

A GPP possesses one or several CPU (of same architecture) that are associated with their cache memories. They use an UMA (Uniform Memory Access) and and interface controller.



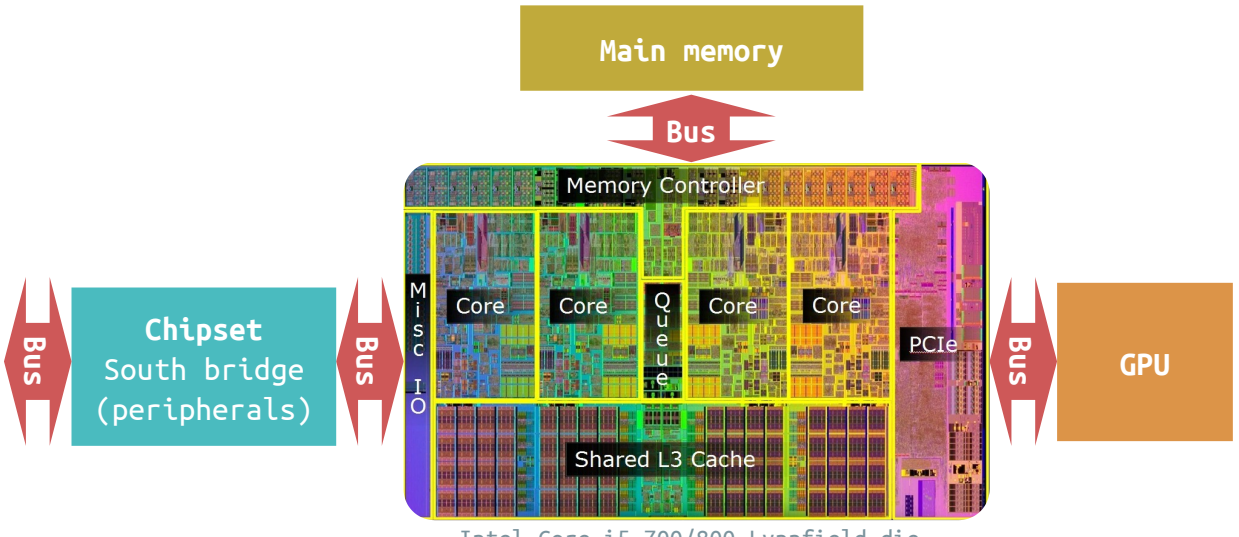
Example of the Intel Core i5 family.





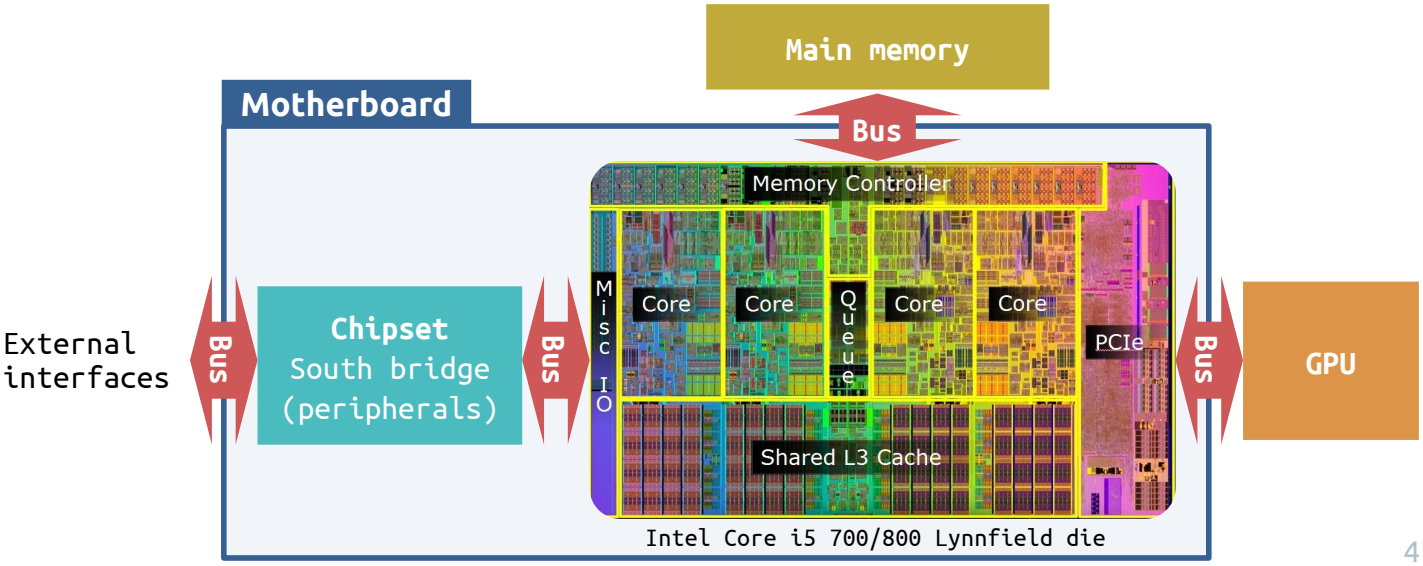
Intel Core i5 700/800 Lynnfield die

GPP integrated into a motherboard



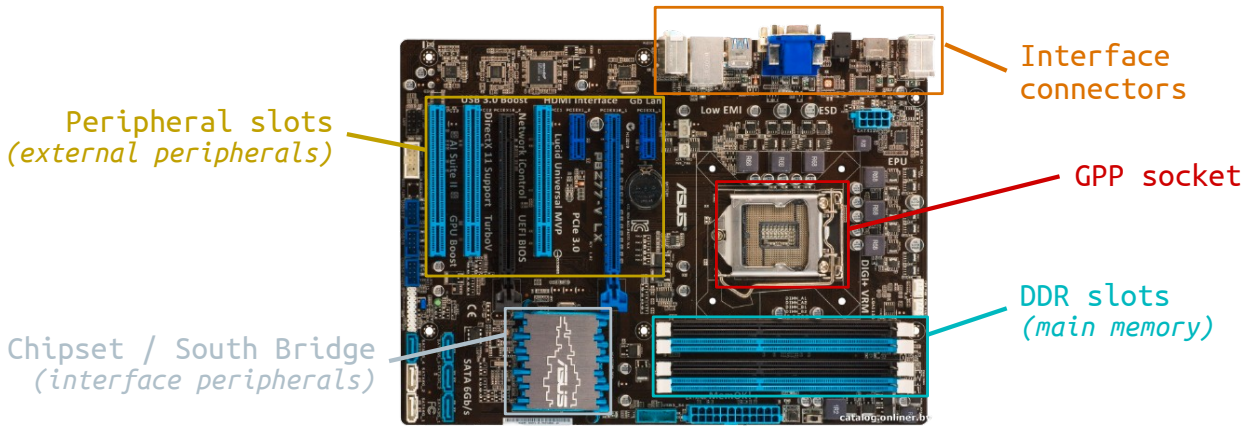
Intel Core i5 700/800 Lynnfield die

GPP integrated into a motherboard



A GPP must be carried onto a motherboard, on which main memory (RAM) and external interface peripherals will be placed.

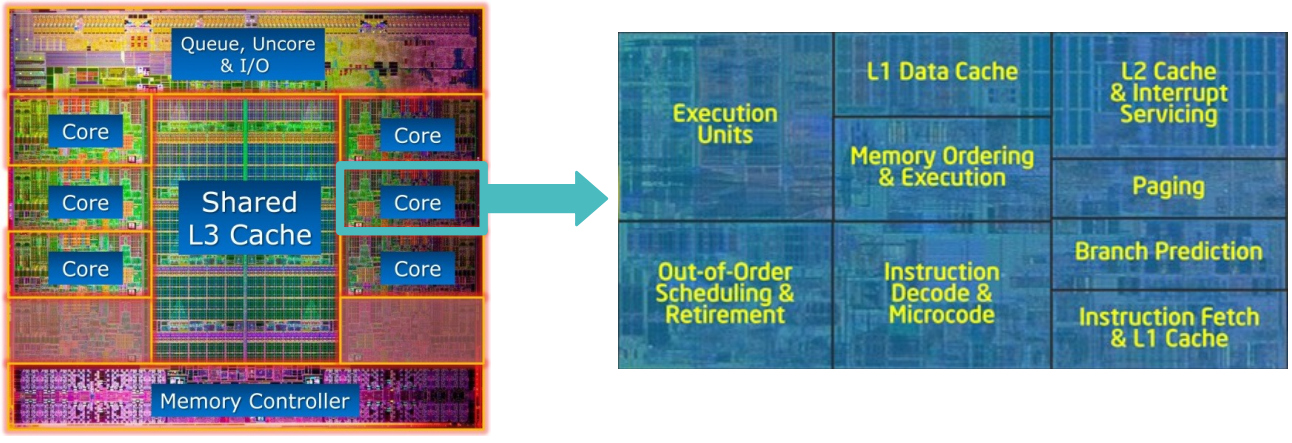
Example of a motherboard from ASUS, second leader of world market in 2016.



GPPs have CPU said to be superscalar. Processors with this type of CPU pipeline are generally characterised by the implementation of the following hardware accelerating mechanisms:

- *Out-Of-Order* execution stage: instructions are not executed in the programmed order. A hardware scheduler looks for dependencies on data, the intermediate results are stored in other registers and instructions are executed in another order (in comparison to the "programmed" order).
- Branch-prediction stage: use statistics and counters to estimate the success rate of a test statement (if, else, for, while, …)
- RISC-like execution stage: even if the ISA (Instruction Set Architecture) is CISC.

Die of a Core i7 CPU (Intel Sandy Bridge generation).



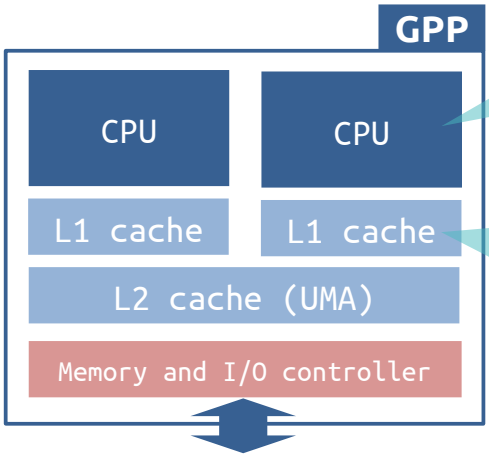
Intel Core i7

Sandy Bridge CPU/Core

However, GPP's great adaptability and hardware complexity leads to a lack of determinism and performance when it comes to the execution of specific algorithms.

For GPPs, the calculation power is simply not good when compared to the power consumption and the price.

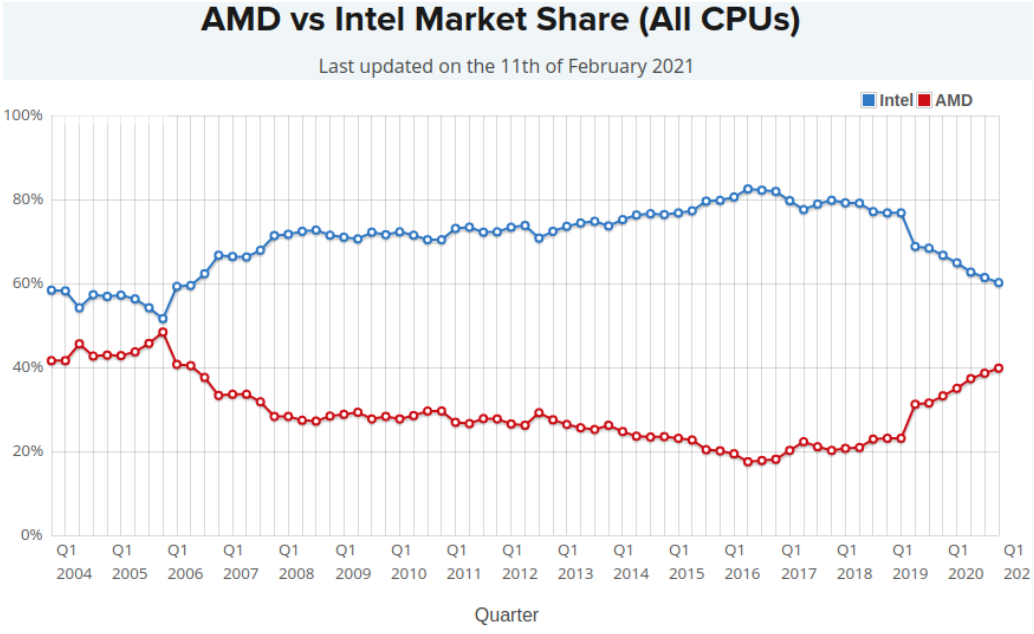
GPPs are designed to support an high-end OS (Operating System, fr: *Système d'exploitation*) and to execute application code. As already mentioned, they are not specialised for signal, image, audio and video processing for instance.



Superscalar CPU
- Out-Of-Order execution
- Branch prediction
- Not determinist
- Bad (calculus power) / (Watt x Cost) ratio

Memory
- Uniform Memory Access (UMA)
- Cache memory:
  - → Fast transfer technologies
  - → Copy information from main memory (DATA or INST.)
  - → Cache controllers for keeping data up to date
  - → Not determinist

Market shares: Intel vs. AMD



https://www.cpubenchmark.net/market_share.html

AP
APPLICATION PROCESSOR

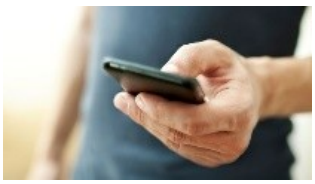
Applications
Architecture
Qualcomm
ARM

AP – APPLICATION PROCESSOR

Applications

The AP (Application Processor) market is recent and has started with mobile phones and tablets.

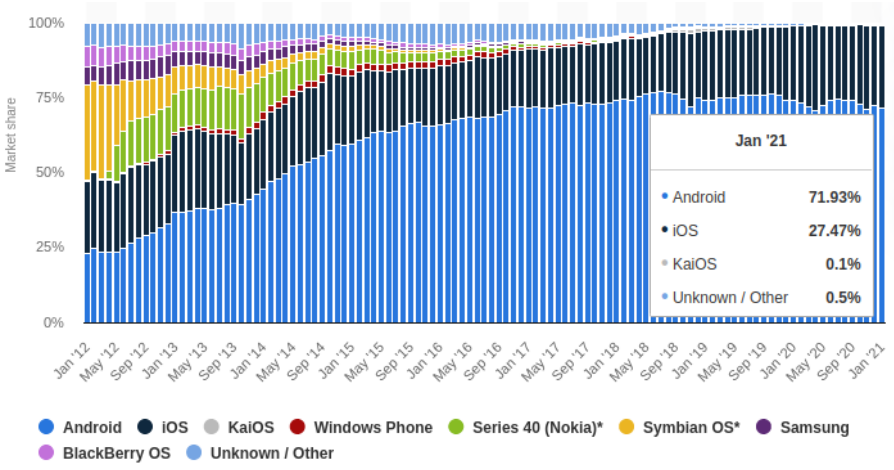
APs embed many functionalities and hardware services, and even SoC (System on Chip).






Mobile phones is the main target market for APs.

This market has led to an overwhelming use of the Android operating system in 2016 (Android is a Linux-kernel based OS).



Source:
Statista 2021

AP – APPLICATION PROCESSOR
Applications

However application processors are seen in many other embedded systems as well, whatever the final application: consumer, defence, transport, …

In those cases they are usually embedded with an operating system and a graphical interface.

Freebox Revolution

Sony X94C 4K television

Cook tablet
(EOLANE, made in Caen)

In most cases, APs are used by high-level operating systems.

On those markets, GNU/Linux systems and customs versions reign supreme.

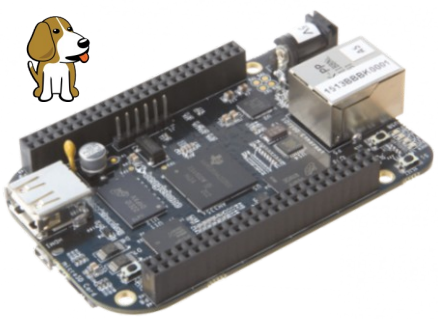
Example of EOLANE (French, #2 in Europe): industrial platform working with a Freescale iMX6 SoC/AP based on a GNU/Linux system.

Here are the two major solutions of user-oriented AP-based boards:

Raspberry Pi (Broadcom BCMxxxx SoC) and Beaglebone (TI AM335x SoC) projects.

These solutions are also based on GNU/Linux operating systems.

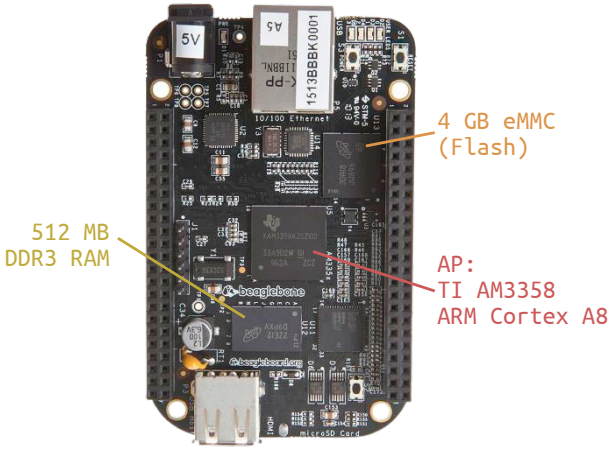
They are more likely to be used for prototyping stages or in a teaching environment, but cannot be industrialised. However hardened versions exist.



An application processor has one or several superscalar generalist CPUs. Their work is to execute the high-level operating systems (virtual or real) and application codes.

An AP may also have many calculus specialised functions (such as GPU, DSP, cryptography, …), an evolved peripheral set and an internal memory. However the latter is not capable of containing the operating system but has a bootloader instead.

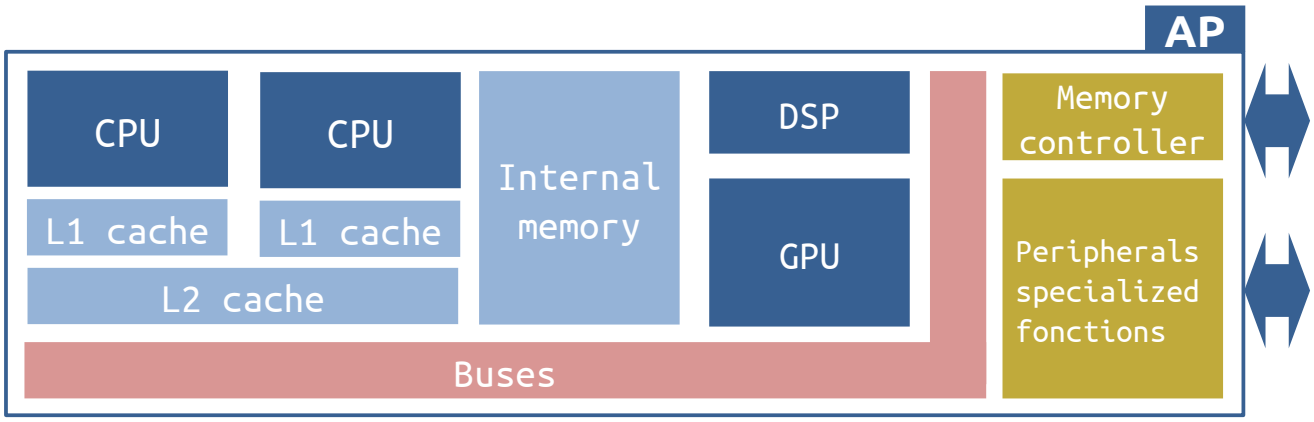
As a consequence a DDR volatile main memory and a remanent mass storage (MMC, eMMC, SDcard) must both be added as external components.



https://beagleboard.org/black

APs are fully operational systems in an integrated circuit (heterogeneous architecture).

Nonetheless main memory must be added as an external component.

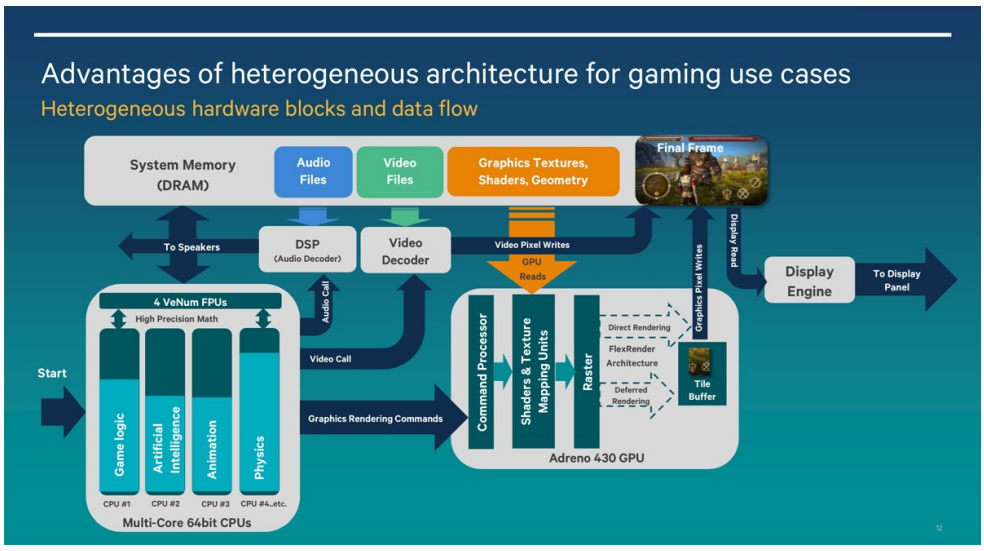


Contrary to MCUs, which contain all hardware services in a single chip, application processors require an important unitary cost and are therefore no the best solution for low-cost or large-quantities productions.

Yet if the application needs evolved interface and/or connectivities, MCUs are not suitable any more because of their low performances. APs then become the best solution.

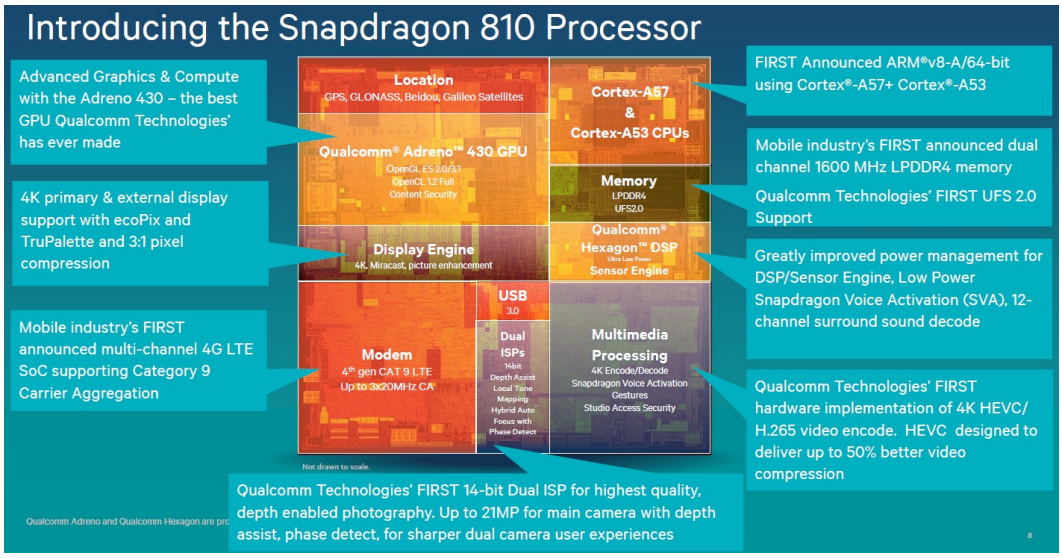
GPP	AP	MCU
+ RAM + hard drive + mother board	+ RAM + MMC + PCB	+ PCB
$$$	$$	$

Observe the point of a heterogeneous architecture for video games applications.



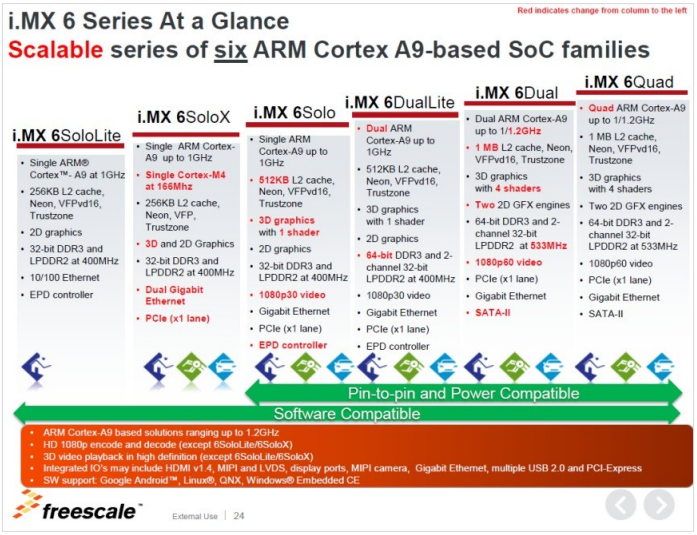
The market leader is Qualcomm.
This is due to its Snapdragon family dedicated to mobile phones market.

Internal architecture and hardware functionalities of the Qualcomm Snapdragon 810.

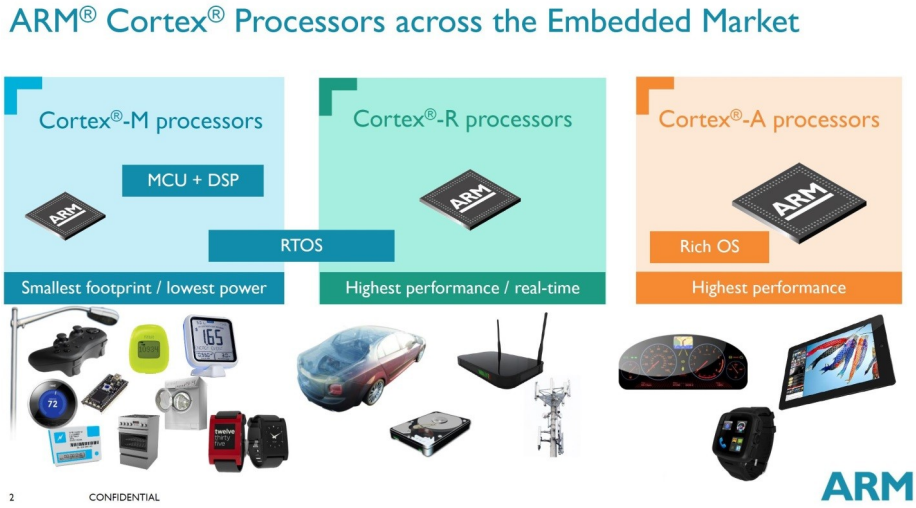


Les deux leaders du marché hors terminaux mobiles sont Texas Instruments et Freescale, deux fondeurs offrant de larges communautés d'utilisateurs.

Observons la famille i.MX6 de Freescale :



Outside of the mobile phones market, the ARM Cortex-A is the leading architecture in embedded markets. The 'A' stands for "Application".



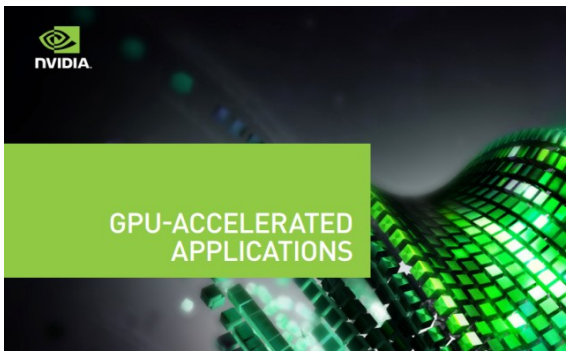
GPU
GRAPHICS PROCESSING UNIT

Applications
Architecture
Nvidia products
Markets

Applications

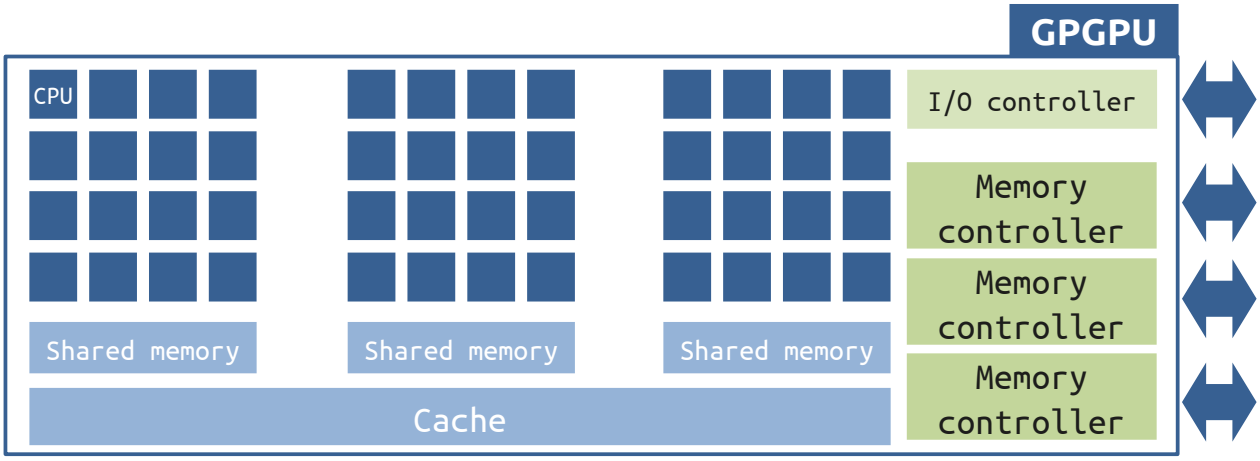
GPUs (*Graphics Processing Unit*) are specialised co-processors dedicated for high intensity calculus and processing.

The term of GPGPU (General Purpose GPU) appeared in the last few years. It relates to massive computing in very sense. Applications are diverse: finance, research, science, medical imagery, video games, …



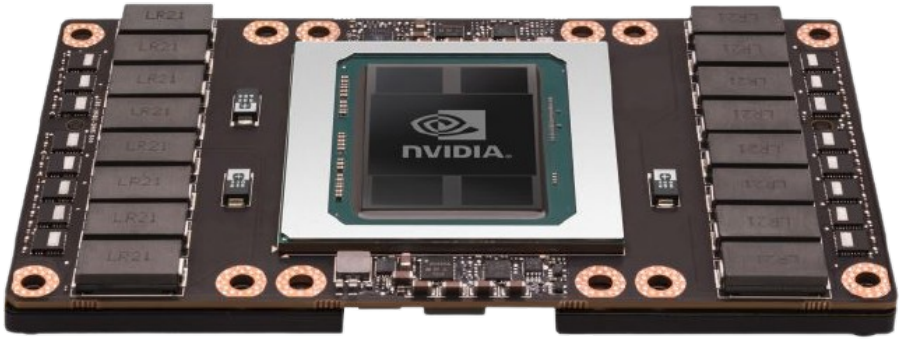
http://www.nvidia.com/content/gpu-applications/PDF/gpu-applications-catalog.pdf

GPU possess a shared NUMA (Non Uniform Memory Access), allowing a cloning of data to be processed and a execution parallelism. They integrated a massively parallel architecture.



Let's take a look at the Tesla P100 board characteristics. It has been produced by Nvidia in 2016 and it is dedicated to the then most advanced data centres.

The GPU is a Nvidia GP100.

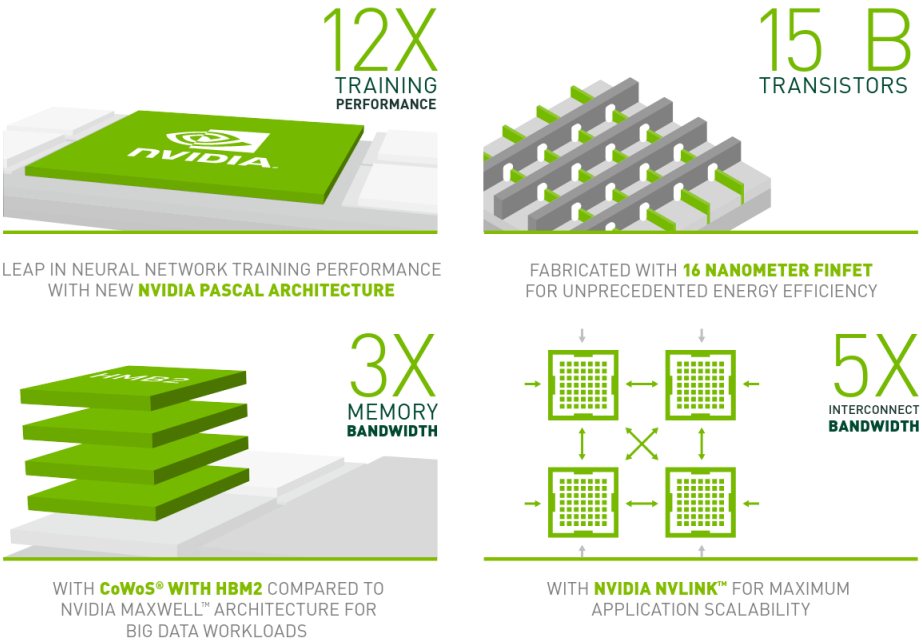


SPECIFICATIONS	
GPU Architecture	**NVIDIA Pascal**
NVIDIA CUDA® Cores	**3584**
Double-Precision Performance	**5.3 TeraFLOPS**
Single-Precision Performance	**10.6 TeraFLOPS**
Half-Precision Performance	**21.2 TeraFLOPS**
GPU Memory	**16 GB CoWoS HBM2**
Memory Bandwidth	**732 GB/s**
Interconnect	**NVIDIA NVLink**
Max Power Consumption	**300 W**
ECC	**Native support with no capacity or performance overhead**
Thermal Solution	**Passive**
Form Factor	**SXM2**
Compute APIs	**NVIDIA CUDA, DirectCompute, OpenCL™, OpenACC**

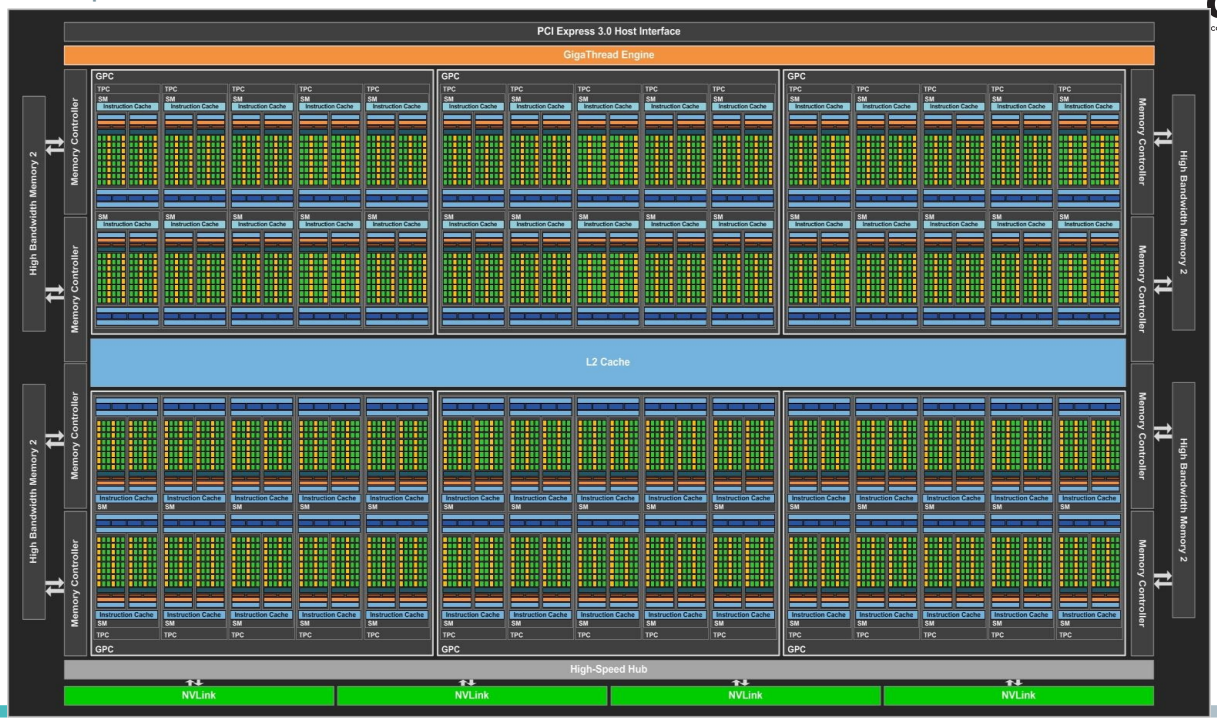
TeraFLOPS measurements with NVIDIA GPU Boost™ technology

https://www.nvidia.com/fr-fr/data-center/tesla-p100/

Nvidia products: Pascal architecture



Nvidia products: GP100 GPU architecture



The Nvidia GP100 GPU in a nutshell

- 6 Graphics Processing Clusters
- 30 Texture Processing Clusters (5 / GPC)
- 60 Streaming Multiprocessors (2 / TPC)
- 3840 single precision cores (64 / SM)
- 1920 double precision units (32 / SM)
- 240 texture units (4 / SM)
- 8 memory controllers
  - 8 x 512 KB = 4096 KB L2 cache
  - 4 pairs that control HBM2 DRAM

Note : the Tesla P100 board uses only 56 SMs out of the 60 available in the GP100 GPU.

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs[1]	5040	6840	10600
Peak FP64 GFLOPs[1]	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm²	601 mm²	610 mm²
Manufacturing Process	28-nm	28-nm	16-nm FinFET → TSMC

[1] The GFLOPS in this chart are based on GPU Boost Clocks.

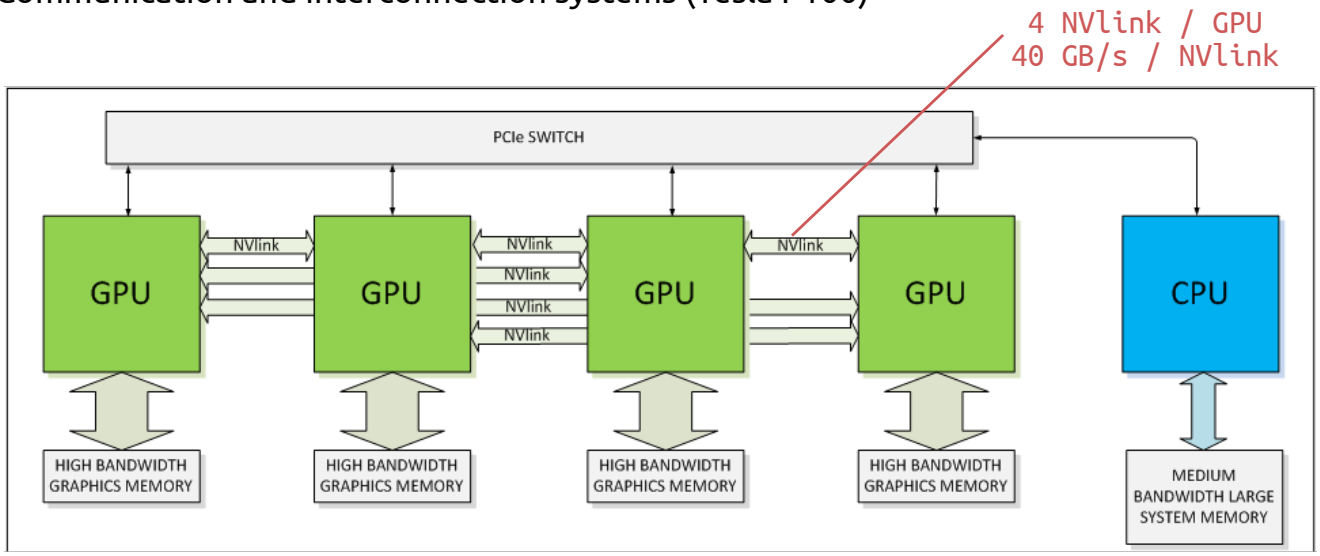
GPUs integrate a large number of classical pipeline CPUs but with vectorial SIMD execution units.

EU = Execution Unit
SIMD = Single Instruction Multiple Data

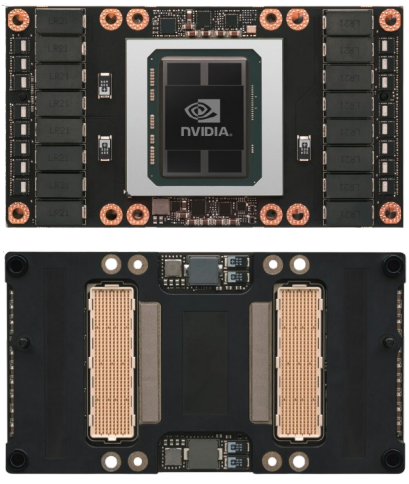
GPC = Graphics Processing Cluster
TCP = Texture Processing Cluster
SM = Streaming Multiprocessor (multithreaded processor)

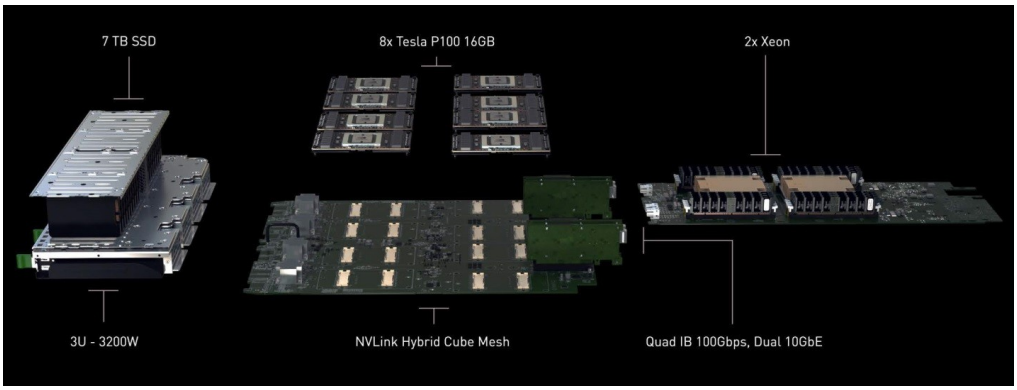
Warp = thread of SIMD instructions
DP = Double Precision
LD/ST = Load/Store
SFU = Special Function Unit
Tex = Texture

Communication and interconnection systems (Tesla P100)

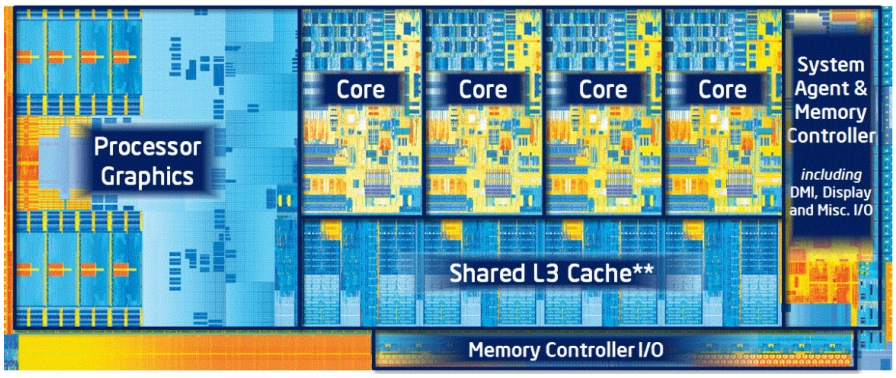


Example of an application using the Nvidia Tesla P100 board.



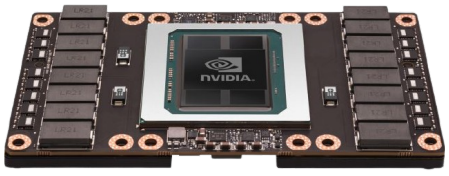


The undisputed leader of the GPU/IGP market is Intel, thanks to their graphics co-processors IGPs (Integrated Graphics Units) embedded in a wide range of their GPPs (more than 70% of market shares in 2016).



Nonetheless the leader of high-performance external solutions in the American company Nvidia.

Tesla K20C



Tesla P100

DSP
DIGITAL SIGNAL PROCESSOR

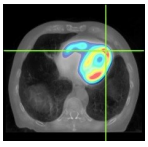
Applications
Architecture
Texas Instruments

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

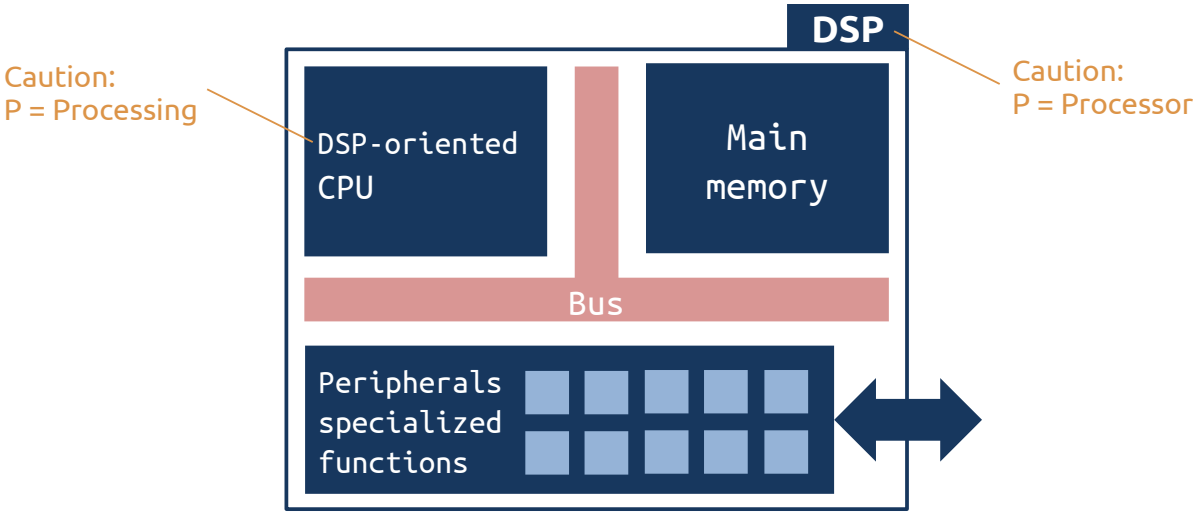
Applications

DSPs (Digital Signal Processors) are dedicated to applications with Digital Signal Processing (fr: Traitement numérique du signal).





DSPs are very close to MCUs: they are autonomous systems.
However their CPU is specialised for signal processing and calculus.

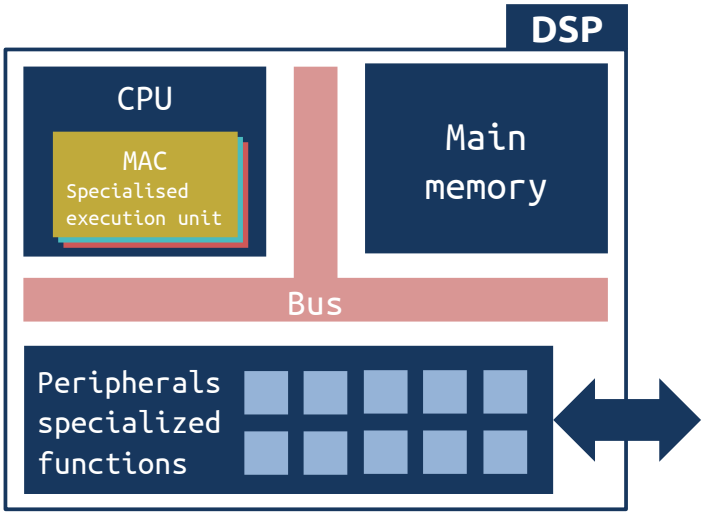


DSP's CPUs possess execution units dedicated for MAC (Multiply Accumulate) or SOP (Som Of Products) operations. These are elementary operations met in almost every signal processing algorithm.

Expansion of the Danielson-Lanczos Lemma to 8 terms:

$$F(n) = \sum_{k=0}^{N/8-1} x(8k)e^{\frac{-j2\pi kn}{(\frac{N}{8})}} + W_{\frac{N}{4}}^{n} \sum_{k=0}^{N/8-1} x(8k+4)e^{\frac{-j2\pi kn}{(\frac{N}{8})}} +$$
$$W_{\frac{N}{2}}^{n} \sum_{k=0}^{N/8-1} x(8k+2)e^{\frac{-j2\pi kn}{(\frac{N}{8})}} + W_{\frac{N}{2}}^{n} W_{\frac{N}{4}}^{n} \sum_{k=0}^{N/8-1} x(8k+6)e^{\frac{-j2\pi kn}{(\frac{N}{8})}} +$$
$$W_{N}^{n} \sum_{k=0}^{N/8-1} x(8k+1)e^{\frac{-j2\pi kn}{(\frac{N}{8})}} + W_{N}^{n} W_{\frac{N}{4}}^{n} \sum_{k=0}^{N/8-1} x(8k+5)e^{\frac{-j2\pi kn}{(\frac{N}{8})}} +$$
$$W_{N}^{n} W_{\frac{N}{2}}^{n} \sum_{k=0}^{N/8-1} x(8k+3)e^{\frac{-j2\pi kn}{(\frac{N}{8})}} + W_{N}^{n} W_{\frac{N}{2}}^{n} W_{\frac{N}{4}}^{n} \sum_{k=0}^{N/8-1} x(8k+7)e^{\frac{-j2\pi kn}{(\frac{N}{8})}}$$

CPU with MAC/SOP dedicated execution units. The ISA (Instruction Set Architecture) contains specific instructions for working with these EUs.

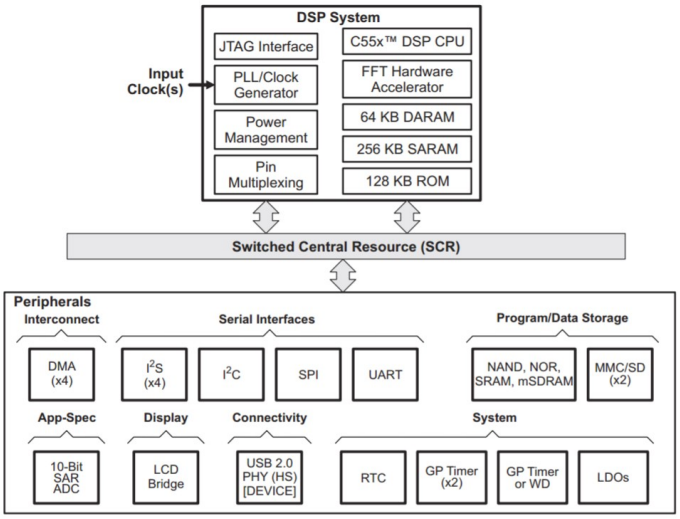


MAC = SOP

MAC : Multiply-Accumulate
SOP : Som of Products

ISA : Instruction Set Architecture
EU : Execution Unit

This is the Texas Instruments C5500 DSP, one of the leading DSP solutions.



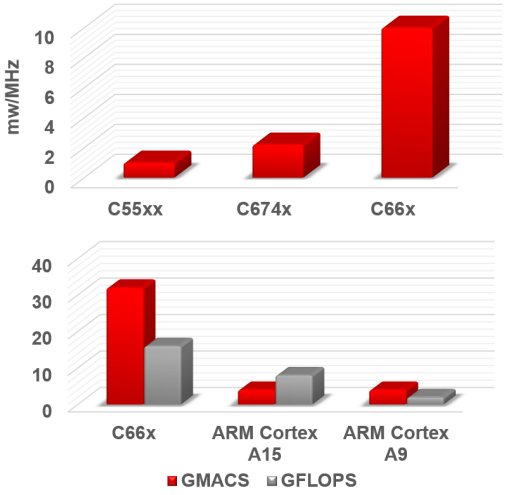
Here is an extract of the C5500 datasheet, with a summary of its characteristics.

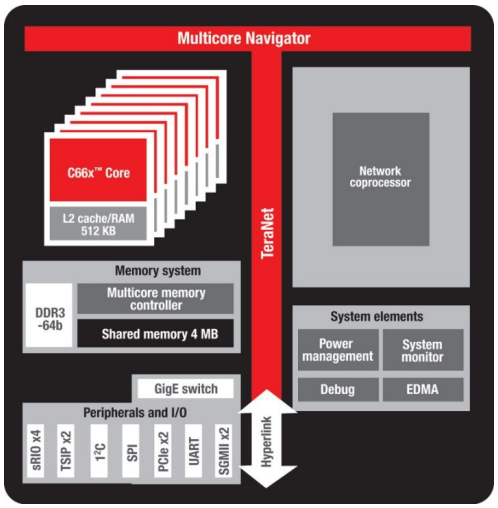
1.1 Features

- CORE:
  - High-Performance, Low-Power, TMS320C55x Fixed-Point Digital Signal Processor
    - 20-, 10-ns Instruction Cycle Time
    - 50-, 100-MHz Clock Rate
    - One or Two Instructions Executed per Cycle
    - Dual Multiply-and-Accumulate Units (Up to 200 Million Multiply-Accumulates per Second [MMACS])
    - Two Arithmetic and Logic Units (ALUs)
    - Three Internal Data and Operand Read Buses and Two Internal Data and Operand Write Buses
    - Software-Compatible with C55x Devices
    - Industrial Temperature Devices Available
  - 320KB of Zero-Wait State On-Chip RAM, Composed of:
    - 64KB of Dual-Access RAM (DARAM), 8 Blocks of 4K x 16-Bit
    - 256KB of Single-Access RAM (SARAM), 32 Blocks of 4K x 16-Bit
  - 128KB of Zero Wait-State On-Chip ROM (4 Blocks of 16K x 16-Bit)
  - Tightly Coupled FFT Hardware Accelerator

https://www.ti.com/lit/ds/symlink/tms320c5533.pdf

Let's switch to the Keystone C6600. This Texas Instruments DSP is one of the highest performances in the current market.

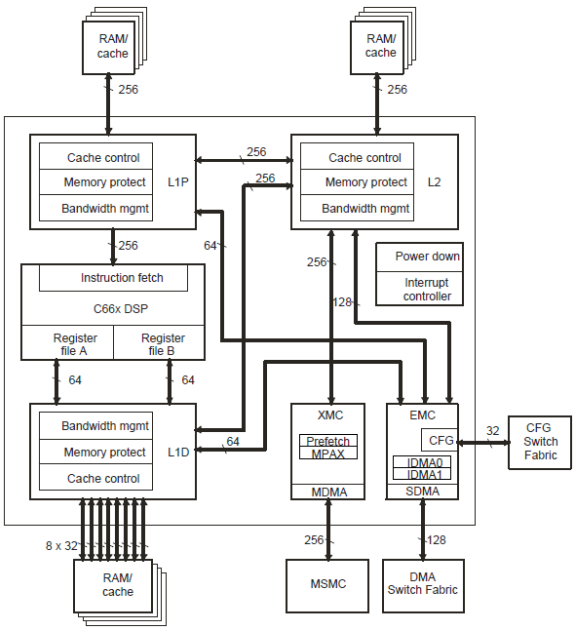




Texas Instruments C6600 CorePac.

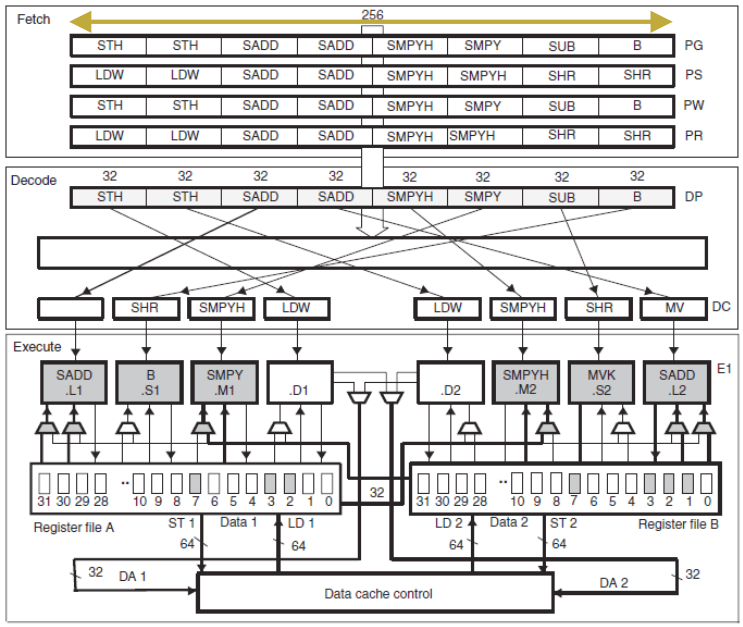
Memory configurable as cache memory or addressable SRAM with no bandwidth loss.

UMA or NUMA models configurable for each core.



C6600 core with:

- 14-stage VLIW hardware pipeline (*Very Long Instruction Word*)
- software pipeline with a max width of 8 instructions



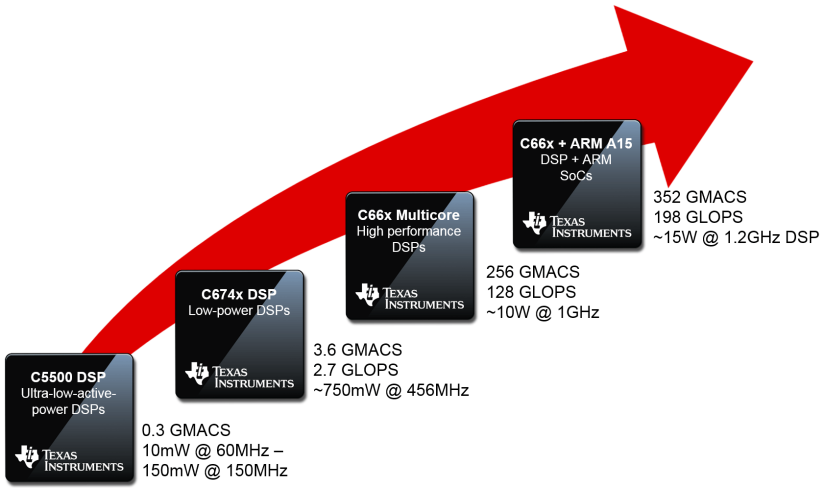
These DSPs are designed for both parallel and daisy-chain work.

Parallel configuration is suitable for massive parallel processing whereas daisy-chain configuration is more suitable for deep processes algorithms.

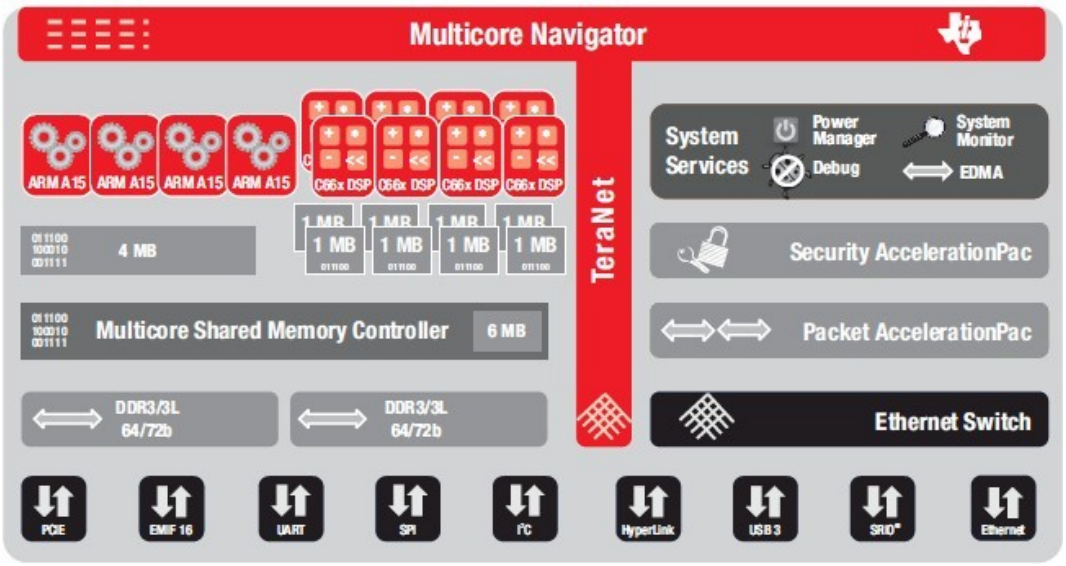
Advantage of using daisy-chain configuration:

That’s not all, TI also offers the Keystone II family. It consists of an AP-SoC with application processors dedicated for digital signal processing applications.

The main target is the telecommunications area.



KeyStone
Multicore DSP+ARM®
Texas Instruments



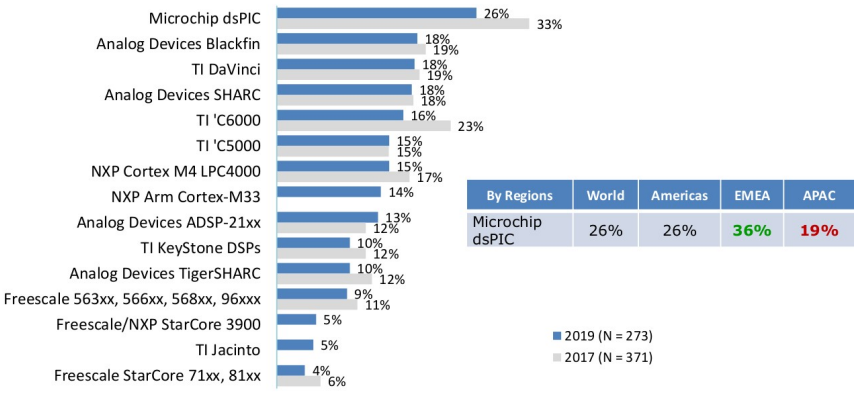
The historical and current leader is by far Texas Instruments.
TI was the first company to design DSP in 1982.

Here is the range of Texas Instruments processors.

Microcontrollers (MCUs)		ARM®-based Processors			Digital Signal Processors		
16-bit Ultra Low Power MCU	32-bit Real-Time MCU	32-bit ARM MCU	32-bit ARM Processors for Performance Applications	Application Processors	Singlecore DSP	Multicore DSP	Ultra Low Power DSP
• MSP430™	• C2000™	• TMS570 Cortex® R4 • RM4 Cortex® R4F • TMS470M Cortex® M3 Automotive	• Sitara™ Cortex A and ARM9 • KeyStone Cortex® A15 and Cortex® A15 + DSP	• OMAP™ Processors • DaVinci™ Video Processors	• C6000™ Power Optimized	• KeyStone Multicore DSP+ARM • C6000™ Multicore	• C5000™

http://processors.wiki.ti.com/index.php/Main_Page

Which of the following DSP chip families would you consider for your next embedded project?



By Regions	World	Americas	EMEA	APAC
Microchip dsPIC	26%	26%	36%	19%

EXECUTION MODELS

Classifying processors according to their execution model

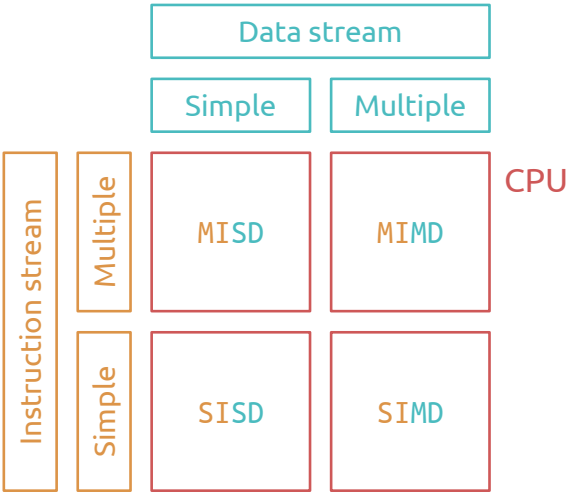
SISD – SIMD – MISD – MIMD

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

The next slides are not intended for proper lecturing.
However you'll hear those terms quite a lot, so here are a few slides about execution models.

Flynn's classification (1972)



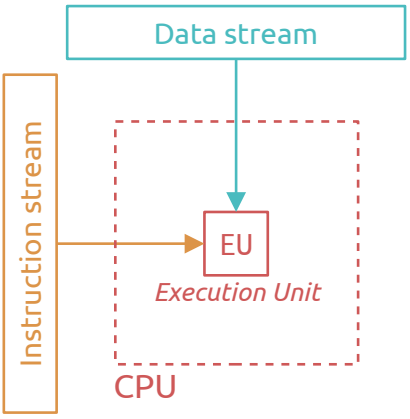
Simple data stream : each operand contains only one piece of data (one memory cell per operand).

Multiple data streams : each operand contains multiple pieces of data (a fixed-size array per operand).

Single instruction stream : the CPU can execute one instruction at once (sequential execution).

Multiple instruction streams : the CPU can execute multiple instructions at once, either using data parallelism (e.g. *forall* loop) or using control parallelism (e.g. parallel sections).

SISD – Single Instruction stream, Single Data stream



The processor execute one instruction at once, each instruction operand containing a single memory cell.

This is the typical mono-processor architecture:

→ Von Neumann architecture

→ MCUs and old GPP generations

→ Sequential processor (no parallelism)

→ Scalar processor

→ A single piece of data (a single memory cell) for each operand

SISD – Single Instruction stream, Single Data stream

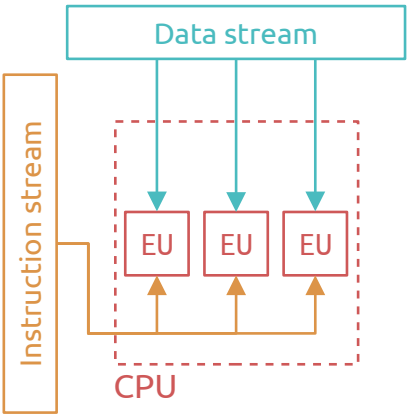
Example: TI C6600 assembly language
Adding two floats

```
; Single Precision ADD
ADDSP     A17, A5, A5

; Result:
; A5 = A5 + A17
```

Example canonical C:
Adding two floats

```
float a, b ;

// Initialising a and b ...

a = a + b ;
```

SIMD – Single Instruction stream, Multiple Data streams



The same instruction will be executed by multiple EUs, each processing its own piece of data. It means the whole CPU will execute a single instruction on multiple pieces of data.

Parallel architecture with centralised control unit:

→ Vectorial processor

→ GPU

→ Intel SSE and AVR instructions set architecture for x86

SSE = Streaming SIMD Extension (SSE, SSE2, SSE3, SSE4)

AVR = Advanced Vector Extensions (AVX, AVX2, AVX512)

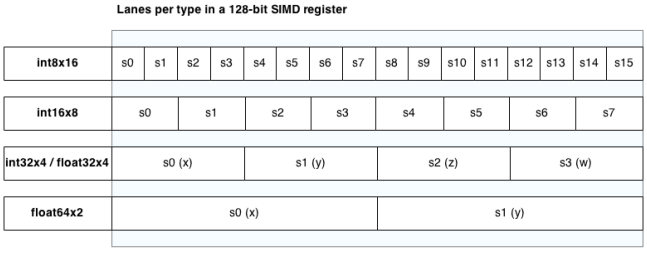
SIMD – Single Instruction stream, Multiple Data streams

Example: TI C6600 assembly language
Adding two couples of floats

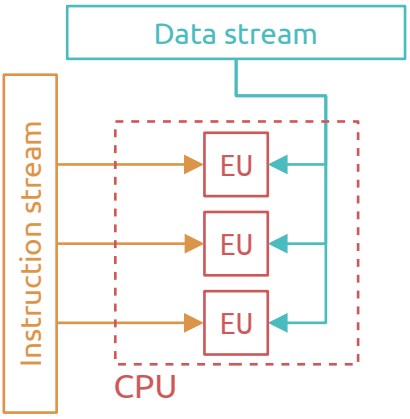
```
; Dual ADD Single Precision
DADDSP    A21:A20, A25:A24, A25:A24


; Result:
; A25 = A25 + A21
; A24 = A24 + A20


; Just like the SSE for Intel, the C6600
; DSP has a C extension (C functions)
; for vectorial instructions
```

Example: x86 SSE C, adding four couples of floats

```
float A[N], B[N], C[N] ;

for( int i = 0 ; i < N ; i += 4 ) {
    __m128 reg_b = _mm_load_ps( &B[i] );
    __m128 reg_c = _mm_load_ps( &C[i] );
    __m128 reg_a = _mm_add_ps( reg_b , reg_c ) ;
    __mm_store_pd( &A[i] , reg_a );
}
```



MISD – Multiple Instruction streams, Single Data stream



Each EU execute its own instruction, with single pieces of data.

Few practical applications

→ code redundancy (for detection of execution errors)

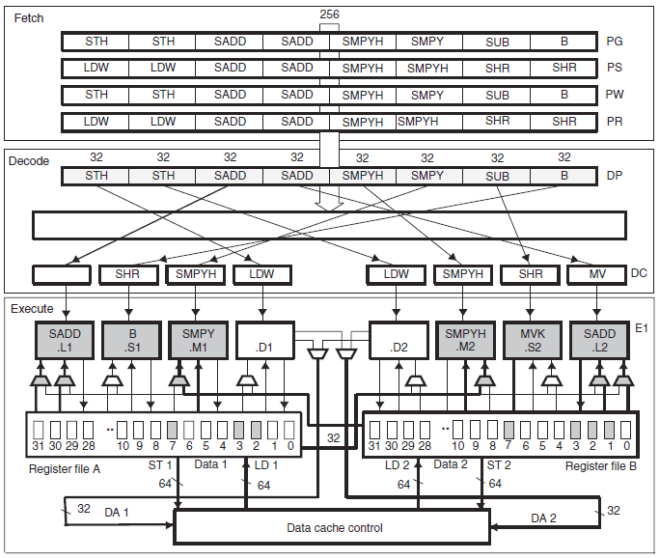
→ VLIW processors (*Very Long Instruction Word*)

e.g. C66xx Texas Instruments DSP

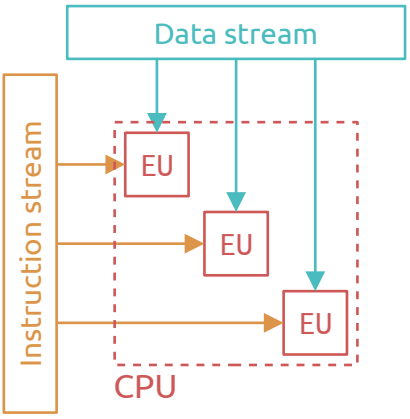
MISD – Multiple Instruction streams, Single Data stream

Example: TI C6600 assembly language
Simultaneously adding and multiplying

```
; ADD Single Precision
; MULTIPLY Single Precision
     ADDSP     A3, A9, A3
||   MPYSP     B3, B9, B3


; The pipes (||) explicitly indicate that
; instructions must be executed in parallel
; (use of software pipeline)

; Result
; A3 = A9 + A3
; B3 = B9 + B3
```



MIMD – Multiple Instruction streams, Multiple Data streams



Each EU executes its own instructions flow on their own data flow.

Execution Unit can be grouped as a cluster.

Parallel architectures with independent control units

→ Super-scalar processors

→ Any modern GPP: x86-x64 (CISC), Cortex-A (RISC)

→ Includes use of SPMD (*Single Program, Multiple Data*)

MIMD – Multiple Instruction streams, Multiple Data streams

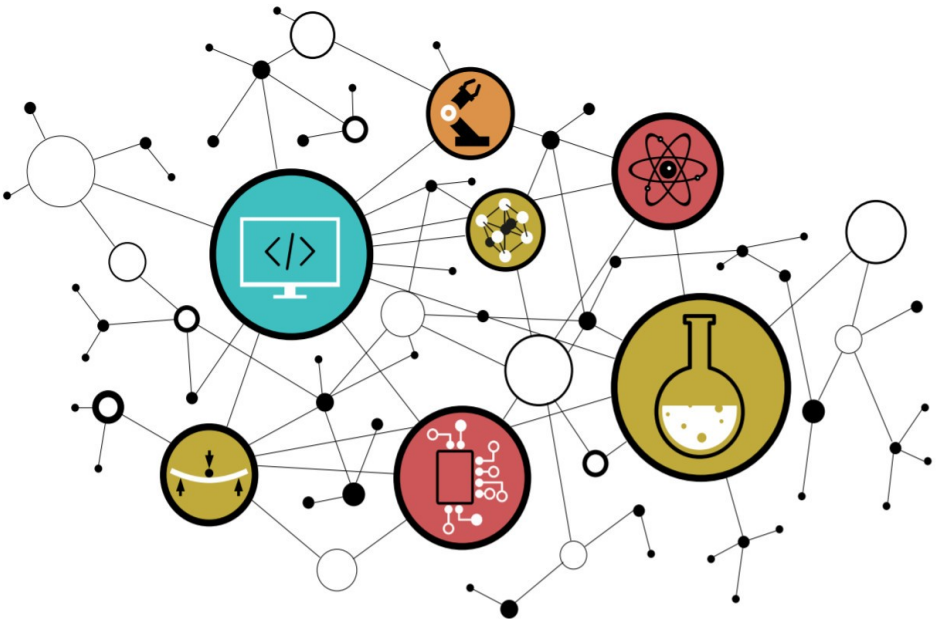
Example: TI C6600 assembly language
Simultaneously adding and multiplying two different couples of data

```
; Dual ADD Single Precision
; Dual SUBSTRACT Single Precision
     DADDSP    A21:A20, A25:A24, A25:A24
||   DSUBSP    B25:B24, B23:B22, B23:B22


; The pipes (||) explicitly indicate that
; instructions must be executed in parallel
; (use of software pipeline)

; Result
; A25 = A25 + A21
; A24 = A24 + A20
; B23 = B25 - B23
; B22 = B24 - B22
```

Example: C and OpenMP
Parallelisation of for loop

```
#pragma omp parallel reduction(+:acc)
{

    #pragma omp for schedule(static)
    for( k = 0; k < size; k ++ )
    {
      acc += A[i * size + k] * x[k];
    }
}
```

APPLICATION VS ALGORITHME



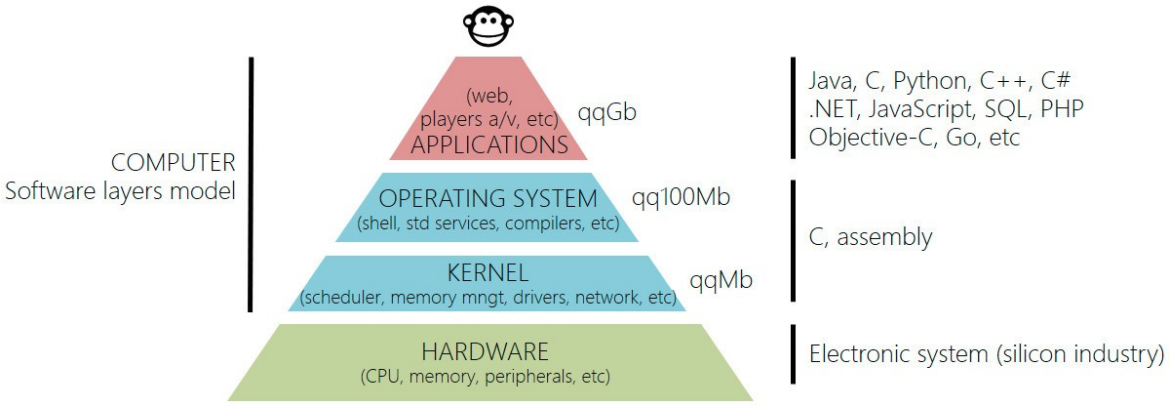
Chapter 2

Choosing a specialized CPU

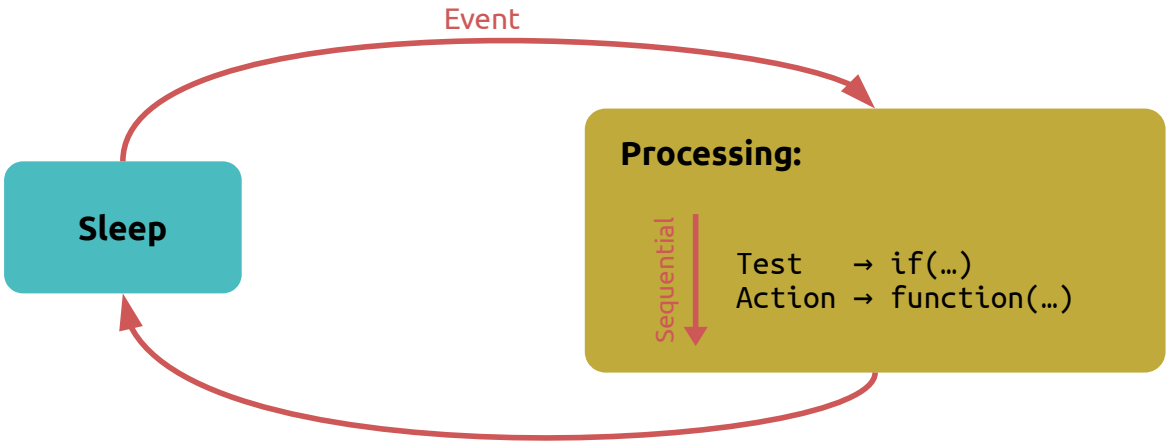
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2021-2022

The objective of an application is to fulfill specifications (or requirements).



About 90 % of the time, the processing consists of a **simple supervision**.



→ Opt for MCU, AP or GPP architectures

From time to time the function to process might be an algorithm,
i.e. apply a processing to a certain amount of data (information).

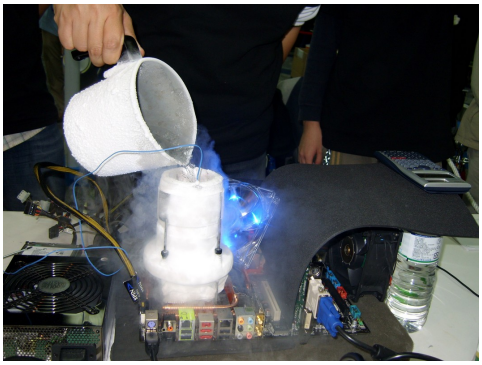
Algorithm examples: search, sort, digital signal processing (audio, radar, comms, …), ...

The first choice of processor should always be a general-purpose processor.

However if it does not match the specifications, it is wise to switch to a processing-specialized architecture so that we can:

- Reduce the processing time
- Reduce the code size and/or its memory footprint

Note that switching to a specialized processor should be justified with measurements.



Take for example the DFT algorithm:

For ONE frequency sample

$$S(k) = \sum_{n=0}^{N-1} s(n) \times e^{-j2\pi k \frac{n}{N}}$$

Sum

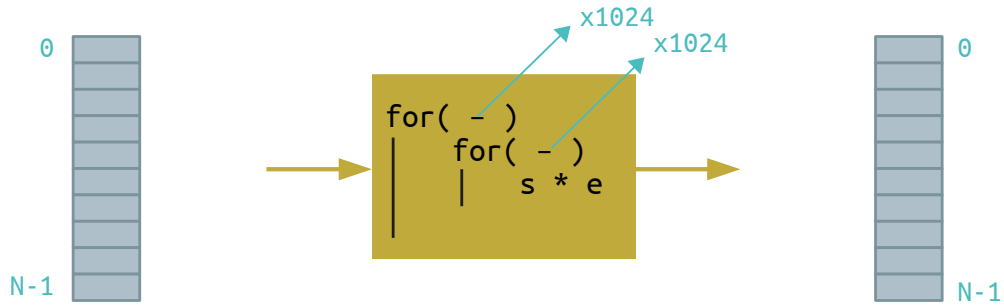
Product

Sum Of Product (SOP)
Multiply-Accumulate (MAC)

- Each product is independent from another
- → Parallelism available!
- Same for the processing every single frequency sample

MATH

SOFT

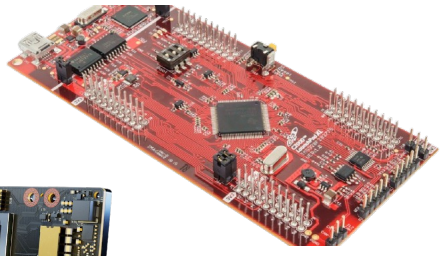


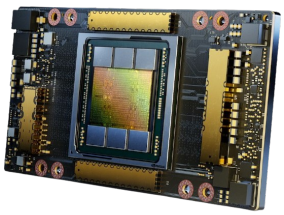
Finally, choose the CPU according to your needs.

DSP: low-power, low-cost, very low-level development (C, asm)

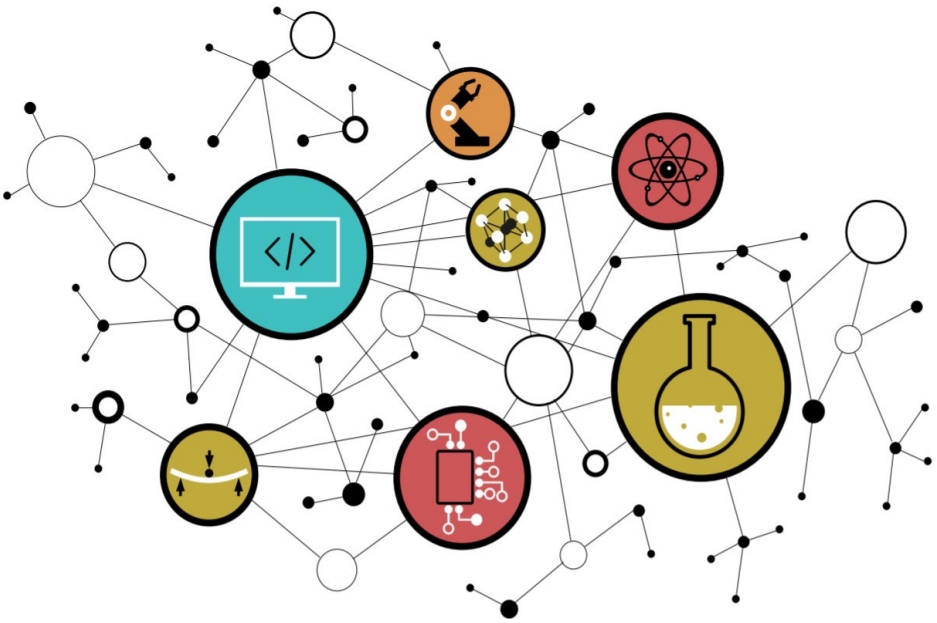
GPU: high-power, high-cost, high-level development (C++, OpenMP, Cuda, …), high-parallelism potential

MPPA: Massively Parallel Processor Array, not widespread yet, but huge potential (dispatch cores to specific algorithms).





ARCHITECTURE CPU C6678 VLIW DE TEXAS INSTRUMENTS



Chapter 3

TI C6678's Architecture

2021-2022

TMS320C6678 PROCESSOR

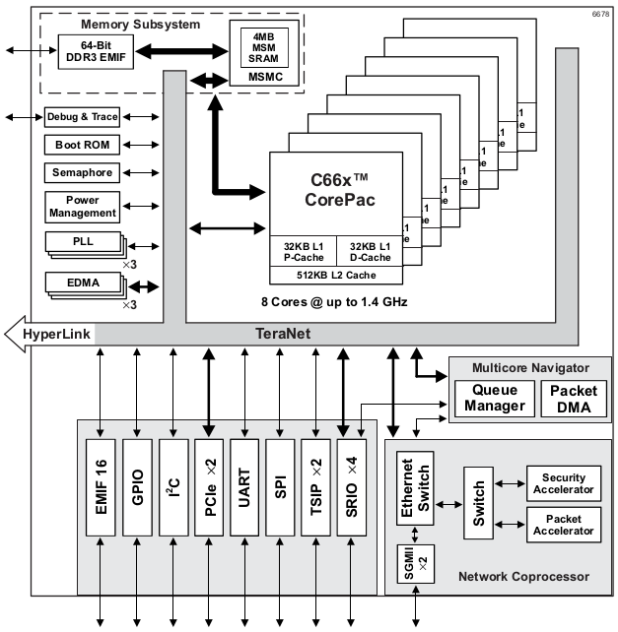
Processor and Core specifications

The TI C6600 is a multicore DSP with a homogeneous CPU architecture.

It includes 8 RISC-like VLIW CPUs that can be clocked up to 1.4 GHz.

→ 44.8 GMAC/core for fixed point @1.4 GHz

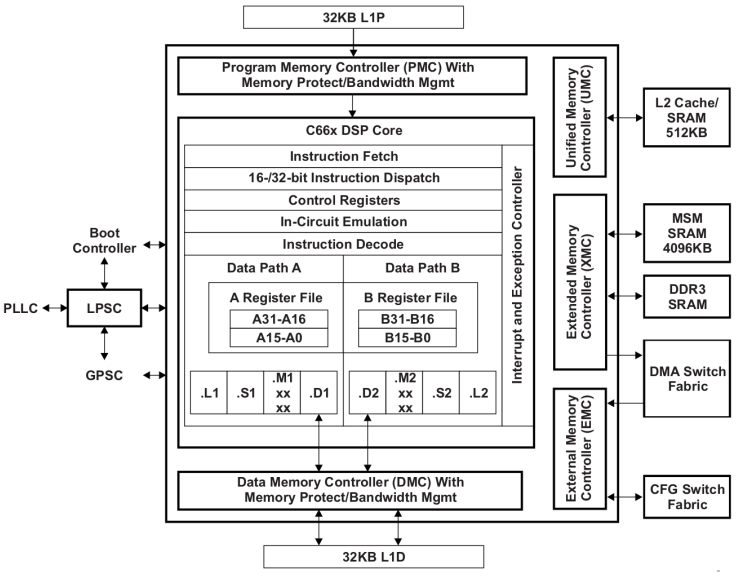
→ 22.4 GFLOP/core for floating point @1.4 GHz



TMS320C6678 functional block diagram

The C66x CorePac consists of several components:

- The C66x DSP and associated C66x CorePac core
- Level-one and level-two memories (L1P, L1D, L2)
- Data Trace Formatter (DTF)
- Embedded Trace Buffer (ETB)
- Interrupt Controller
- Power-down controller
- External Memory Controller
- Extended Memory Controller
- A dedicated power/sleep controller (LPSC)

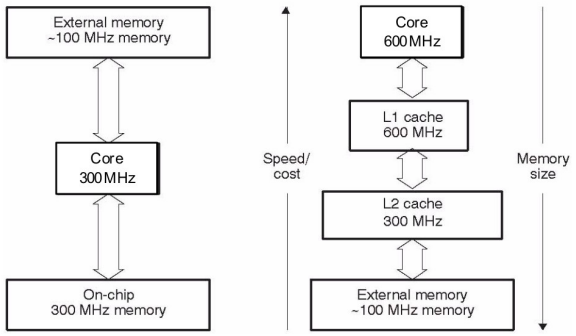


TMS320C66x CorePac DSP Block Diagram

Each core has its own cache memories :

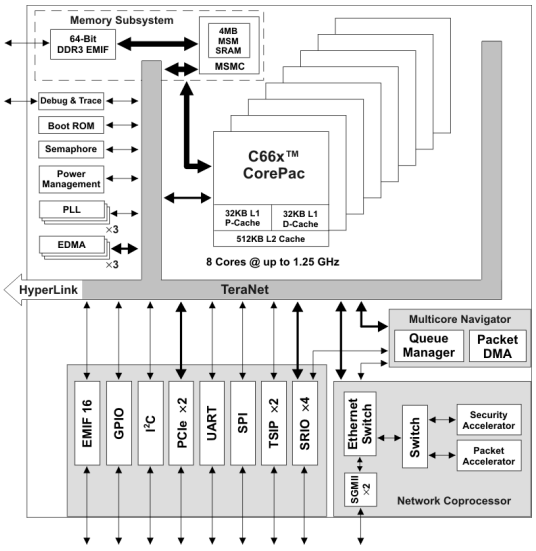
- 32 kB L1P cache memory
- 32 kB L1D cache memory
- 512 kB L2 cache memory

Figure 1-1 Flat Versus Hierarchical Memory Architecture



"Why Use Cache?"
Texas Instruments,
SPRUGY8-November 2010

Also, all cores can access to a 4 MB multicore shared memory (MSM), which can be configured either as a cache memory or as an addressable SRAM.



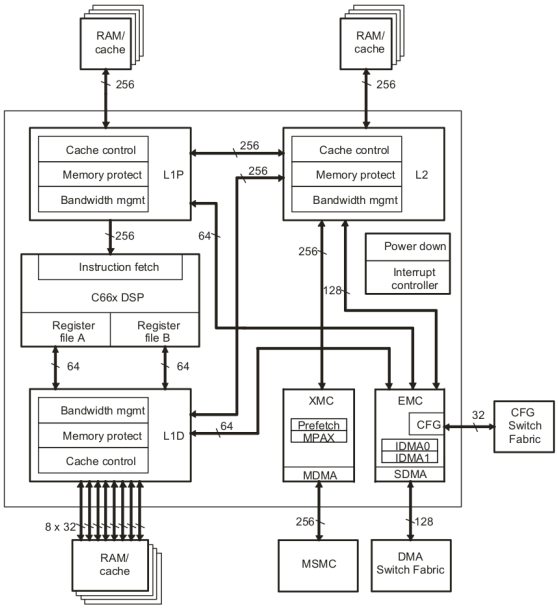
- Multicore Shared Memory Controller (MSMC)
  - 4096KB MSM SRAM Memory Shared by Eight DSP C66x CorePacs
  - Memory Protection Unit for Both MSM SRAM and DDR3_EMIF

The IDMA (Internal Direct Memory Access) is a DMA controller local to the CorePac.

It can be configured and is fully accessible by the developer.

It can handle data transfer between local memories, or between peripheral configuration space (CFG) and local memories.

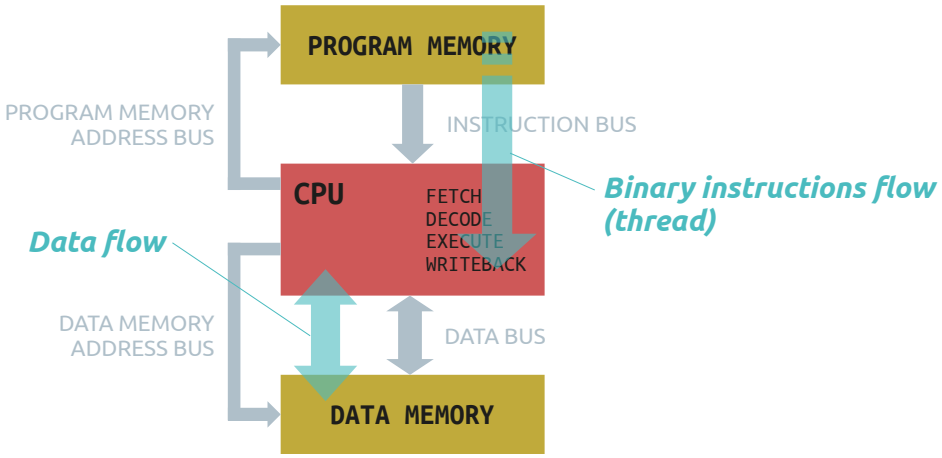
Local transfers to the CPU are determinist.



C6600 HARDWARE PIPELINE

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Reminder: a CPU is a sequential machine, but it can process simultaneously several instructions thanks to the stages of its hardware pipeline.



The C6600 pipeline has 16 stages (called phases)

Figure 5-5 Pipeline Phases

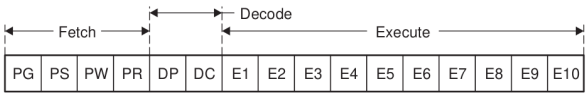
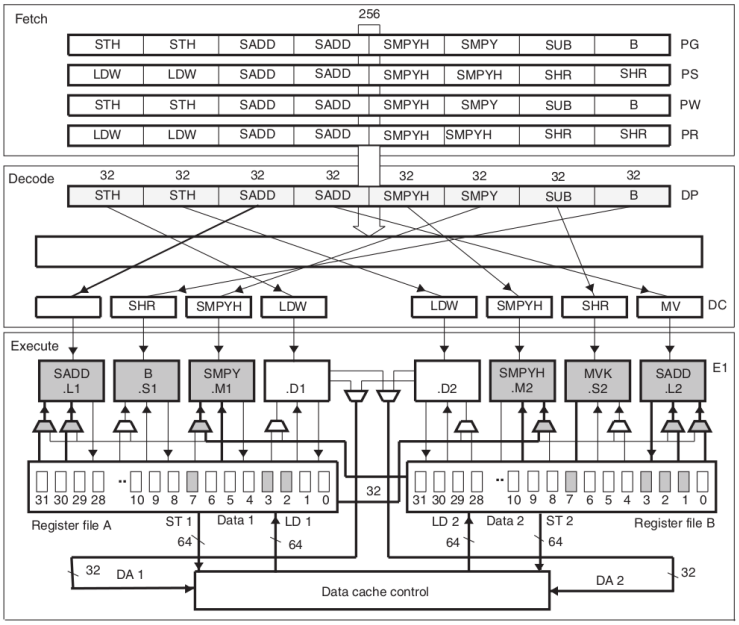


Figure 5-6 Pipeline Operation: One Execute Packet per Fetch Packet

Fetch Packet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	
n+1		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
n+2			PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9
n+3				PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8
n+4					PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7
n+5						PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
n+6							PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
n+7								PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+8									PG	PS	PW	PR	DP	DC	E1	E2	E3
n+9										PG	PS	PW	PR	DP	DC	E1	E2
n+10											PG	PS	PW	PR	DP	DC	E1

CPUs from the C6600 family are equipped with a VLIW (Very Long Instruction Word) hardware pipeline.

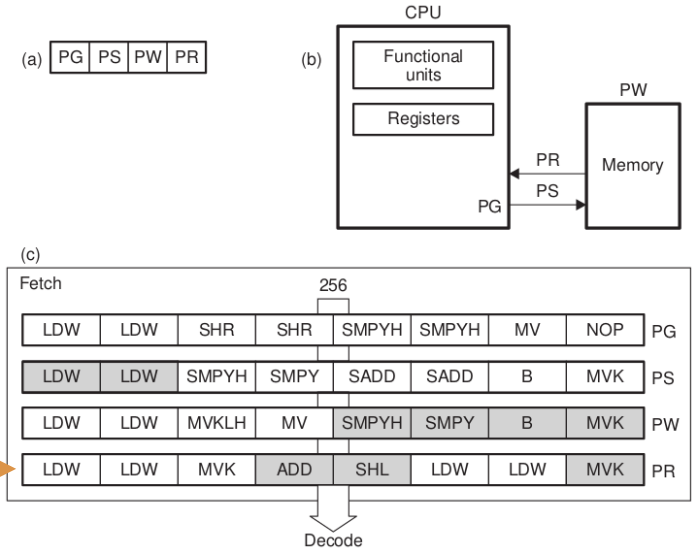
It can process up to 8 instructions at once with its 8 execution units.



Pipeline Phases Block Diagram

The FETCH stage is divided into four phases:

- PG: Program address generate
- PS: Program address send
- PW: Program access ready wait
- PR: Program fetch packet receive



Each fetch packet is 8-word long

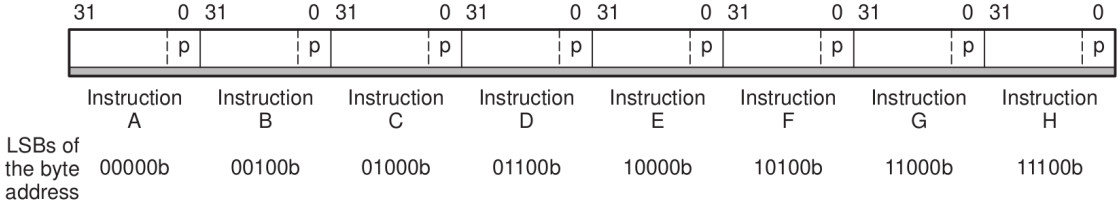
(8 x 32 bits = 256 bits)

Each instruction has a 32-bit fixed size (RISC-like instruction set).

The very last bit of each instruction is named P (Parallel) and is set to 0 or 1 either during the compilation phase or directly by the developer (in assembly language).

By reading this bit, the FETCH stage knows exactly how many instructions to search for, with a maximum of 8.

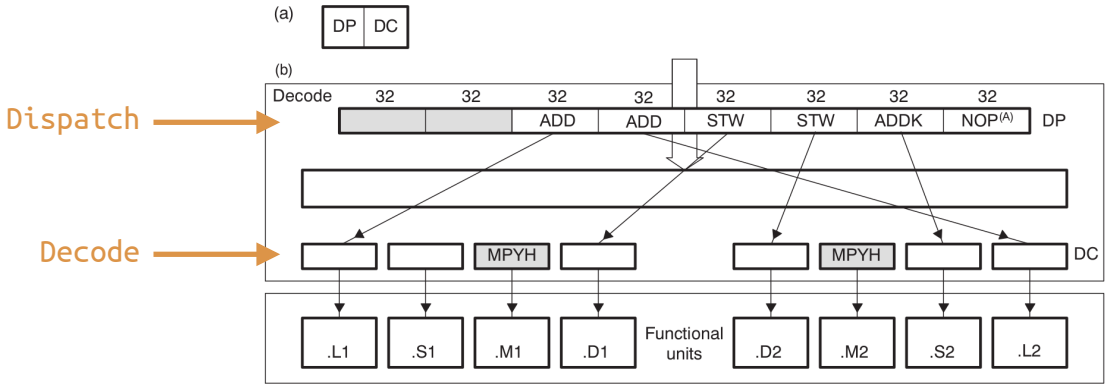
Figure 3-3 Basic Format of a Fetch Packet



With instructions arriving by packets, the decoding stage takes two phases.

1. The *Dispatch* phase redirects the instructions to their dedicated Execution Unit.
2. Each Execution Unit has its proper decoding unit.

Figure 5-3 Decode Phases of the Pipeline



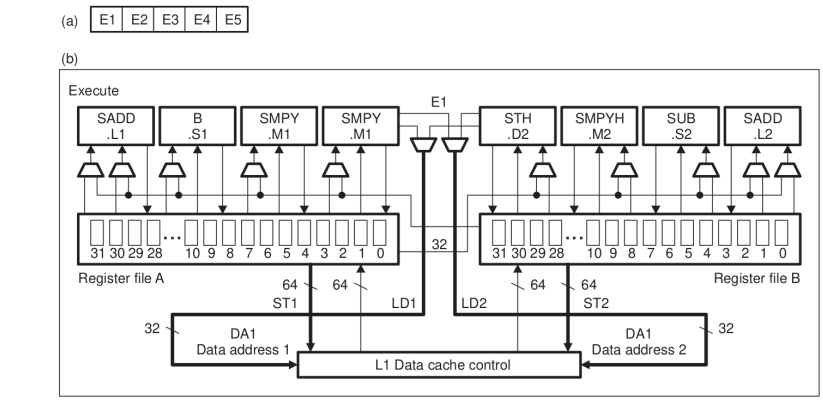
(A) NOP is not dispatched to a functional unit.

The VLIW execution stage has 8 SIMD Execution Units (or Functional Units).

The Execution Units are labeled .L1, .S1, .M1, .D1, .L2, .S2, .M2, .D2, and some instructions are EU-specific.

They are split into two symmetrical sides, each side having its own 32-bit register file.

Figure 5-4 Execute Phases of the Pipeline



Each EU has its own VLIW pipeline.

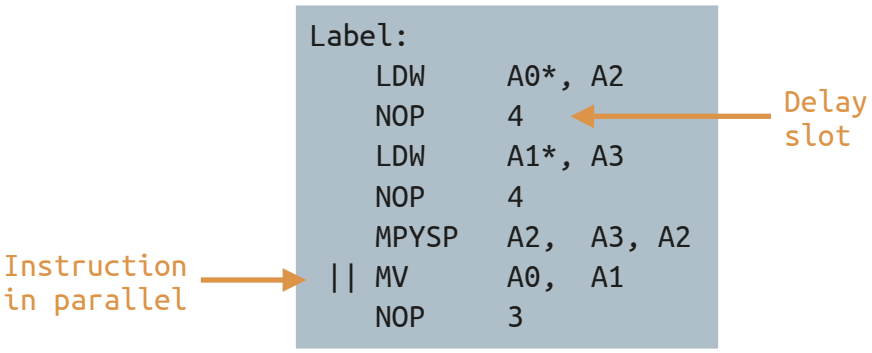
Table 5-1 Operations Occurring During Pipeline Phases (Part 2 of 2)

Stage	Phase	Symbol	During This Phase
Execute	Execute 1	E1	For all instruction types, the conditions for the instructions are evaluated and operands are read. For load and store instructions, address generation is performed and address modifications are written to a register file.[1] For branch instructions, branch fetch packet in PG phase is affected.[1] For single-cycle instructions, results are written to a register file.[1] For DP compare, ADDDP/SUBDP, and MPYDP instructions, the lower 32-bits of the sources are read. For all other instructions, the sources are read.[1] For MPYSPDP instruction, the *src1* and the lower 32 bits of *src2* are read.[1] For 2-cycle DP instructions, the lower 32 bits of the result are written to a register file.[1]
	Execute 2	E2	For load instructions, the address is sent to memory. For store instructions, the address and data are sent to memory.[1] Single-cycle instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.[1] For multiply unit, nonmultiply instructions, results are written to a register file.[2] For multiply, 2-cycle DP, and DP compare instructions, results are written to a register file.[1] For DP compare and ADDDP/SUBDP instructions, the upper 32 bits of the source are read.[1] For MPYDP instruction, the lower 32 bits of *src1* and the upper 32 bits of *src2* are read.[1] For MPYI and MPYID instructions, the sources are read.[1] For MPYSPDP instruction, the *src1* and the upper 32 bits of *src2* are read.[1]
	Execute 3	E3	Data memory accesses are performed. Any multiply instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.[1] For MPYDP instruction, the upper 32 bits of *src1* and the lower 32 bits of *src2* are read.[1] For MPYI and MPYID instructions, the sources are read.[1]
	Execute 4	E4	For load instructions, data is brought to the CPU boundary.[1] For multiply extensions, results are written to a register file.[3] For MPYI and MPYID instructions, the sources are read.[1] For MPYDP instruction, the upper 32 bits of the sources are read.[1] For MPYI and MPYID instructions, the sources are read.[1] For 4-cycle instructions, results are written to a register file.[1] For INTDP and MPYSP2DP instructions, the lower 32 bits of the result are written to a register file.[1]
	Execute 5	E5	For load instructions, data is written into a register.[1] For INTDP and MPYSP2DP instructions, the upper 32 bits of the result are written to a register file.[1]
	Execute 6	E6	For ADDDP/SUBDP and MPYSPDP instructions, the lower 32 bits of the result are written to a register file.[1]
	Execute 7	E7	For ADDDP/SUBDP and MPYSPDP instructions, the upper 32 bits of the result are written to a register file.[1]
	Execute 8	E8	Nothing is read or written.
	Execute 9	E9	For MPYI instruction, the result is written to a register file.[1] For MPYDP and MPYID instructions, the lower 32 bits of the result are written to a register file.[1]
	Execute 10	E10	For MPYDP and MPYID instructions, the upper 32 bits of the result are written to a register file.[1]

1. This assumes that the conditions for the instructions are evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.
2. Multiply unit, nonmultiply instructions are **AVG2, AVG4, BITC4, BITR, DEAL, ROT, SHFL, SSHVL,** and **SSHVR.**
3. Multiply extensions include **MPY2, MPY4, DOTPx2, DOTPU4, MPYHIx, MPYLIx,** and **MVD.**

Instructions with a execution time greater than one cycle is followed by a delay slot, written with a NOP instruction (No Operation).

The NOP instruction corresponds to the time of the instruction travelling through the current Execution Unit.



As an example, here is the documentation for the MPYSP instruction.

4.212 MPYSP

Multiply Two Single-Precision Floating-Point Values

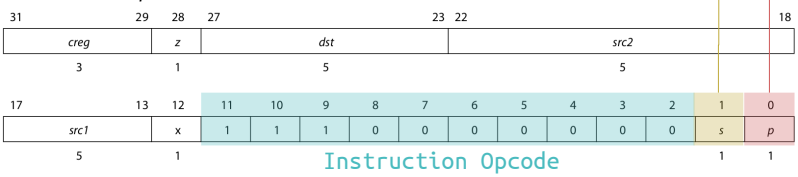
Syntax **MPYSP** (.unit) *src1, src2, dst*

unit = .M1 or .M2

Parallel

Side

Opcode



Opcode map field used...	For operand type...	Unit
src1	sp	.M1, .M2
src2	xsp	
dst	sp	

Instruction Type Four-cycle

Delay Slots 3

Functional Unit Latency 1

See Also MPY, MPYDP, MPYSP2DP

Example `MPYSP .M1X A1,B2,A3`

Execution unit

PROGRAMMING A VLIW CPU

Example code

Let's see how a VLIW (Very Long Instruction Word) CPU works by focusing on the execution units. We'll start with a canonical assembly code.

```
MPYSP   .M1     A2, A3, A4
NOP             3
ADDSP   .S1     A2, A4, A2
NOP             3
FADDSP  .S1     A0, A1, A0
NOP             2
MV      .D1     A0, A1
MV      .D2     B9, B7
```

13 CPU cycles

Now sort the instructions according to the data dependencies.
We'll get three instruction branches.

```
MPYSP   .M1     A2, A3, A4      8 CPU cycles
NOP             3
ADDSP   .S1     A2, A4, A2
NOP             3
FADDSP  .S1     A0, A1, A0      4 CPU cycles
NOP             2
MV      .D1     A0, A1
MV      .D2     B9, B7          1 CPU cycle
```


In theory, these branches can be executed in parallel.

8 CPU cycles

```
MPYSP   .M1     A2, A3, A4
NOP             3
ADDSP   .S1     A2, A4, A2
NOP             3
```

4 CPU cycles

```
FADDSP  .S1     A0, A1, A0
NOP             2
MV      .D1     A0, A1
```

1 CPU cycle

```
MV      .D2     B9, B7
```

However, we must pay attention to functional dependencies!

8 CPU cycles

```
MPYSP   .M1     A2, A3, A4
NOP             3
ADDSP   .S1     A2, A4, A2
NOP             3
```

4 CPU cycles

```
FADDSP  .S1     A0, A1, A0
NOP             2
MV      .D1     A0, A1
```

Same unit

1 CPU cycle

```
MV      .D2     B9, B7
```


We shall rewrite the code (refactoring) to make parallelism possible.

8 CPU cycles

```
MPYSP   .M1     A2, A3, A4
NOP             3
ADDSP   .S1     A2, A4, A2
NOP             3
```

4 CPU cycles

```
FADDSP  .S1     A0, A1, A0
NOP             3
MV      .D1     A0, A1
```

1 CPU cycle

```
MV      .D2     B9, B7
```

Here are the canonical and optimized versions of the same code.

Canonical asm – 13 CPU cycles

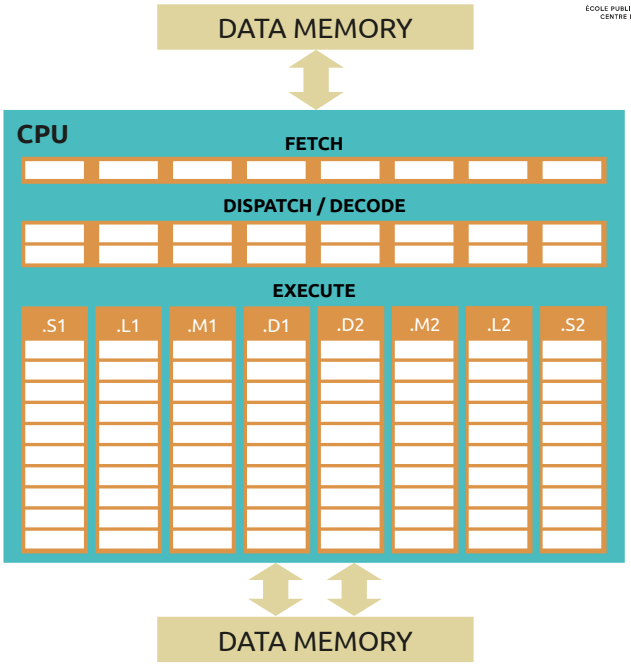
```
MPYSP   .M1     A2, A3, A4
NOP             3
ADDSP   .S1     A2, A4, A2
NOP             3
FADDSP  .S1     A0, A1, A0
NOP             2
MV      .D1     A0, A1
MV      .D2     B9, B7
```

Optimized asm – 8 CPU cycles

```
   MPYSP   .M1     A2, A3, A4
   NOP             2
   FADDSP  .S1     A0, A1, A0
   ADDSP   .S1     A2, A4, A2
   NOP             2
   MV      .D1     A0, A1
|| MV      .D2     B9, B7
```


Optimized asm – 8 CPU cycles

```
   ...
   MPYSP   .M1     A2, A3, A4
   NOP             2
   FADDSP  .S1     A0, A1, A0
   ADDSP   .S1     A2, A4, A2
   NOP             2
   MV      .D1     A0, A1
|| MV      .D2     B9, B7
   ...
```

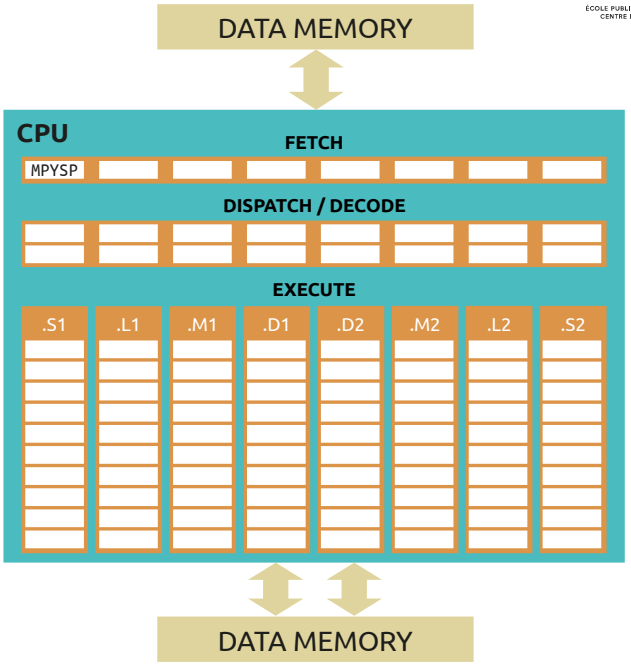


PROGRAMMING A VLIW CPU

Code execution

Optimized asm – 8 CPU cycles

```
   ...
   MPYSP    .M1     A2, A3, A4
   NOP              2
   FADDSP   .S1     A0, A1, A0
   ADDSP    .S1     A2, A4, A2
   NOP              2
   MV       .D1     A0, A1
|| MV       .D2     B9, B7
   ...
```

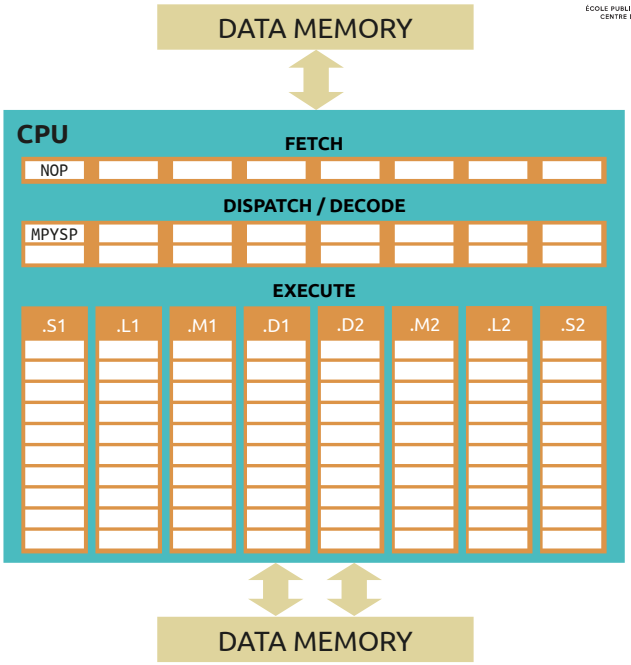


PROGRAMMING A VLIW CPU

Code execution

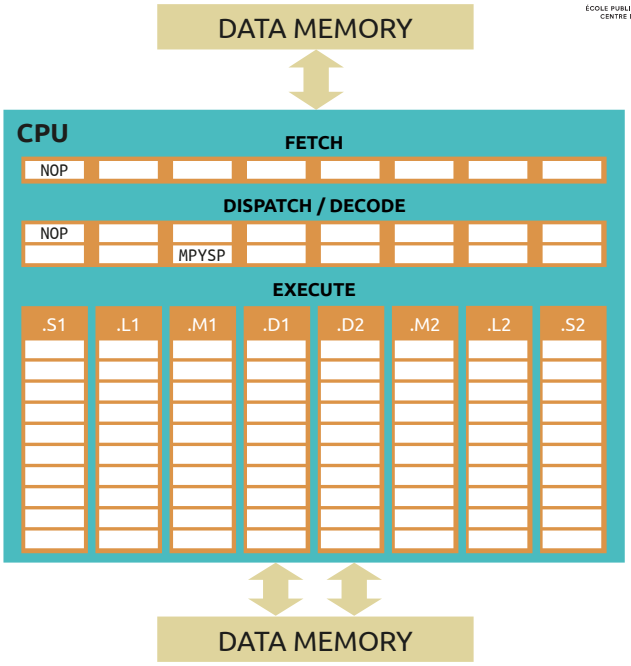
Optimized asm – 8 CPU cycles

```
   ...
   MPYSP    .M1     A2, A3, A4
   NOP              2
   FADDSP   .S1     A0, A1, A0
   ADDSP    .S1     A2, A4, A2
   NOP              2
   MV       .D1     A0, A1
|| MV       .D2     B9, B7
   ...
```



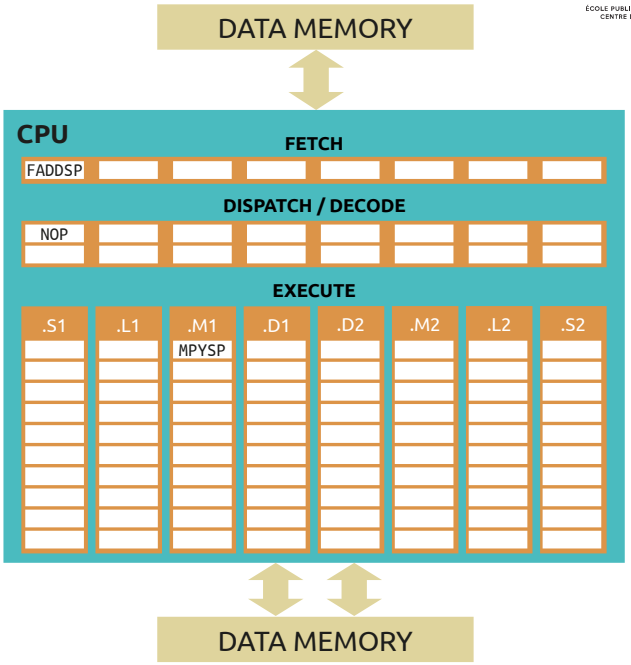
Optimized asm – 8 CPU cycles

```
    ...
    MPYSP    .M1     A2, A3, A4
    NOP              2
    FADDSP   .S1     A0, A1, A0
    ADDSP    .S1     A2, A4, A2
    NOP              2
    MV       .D1     A0, A1
||  MV       .D2     B9, B7
    ...
```



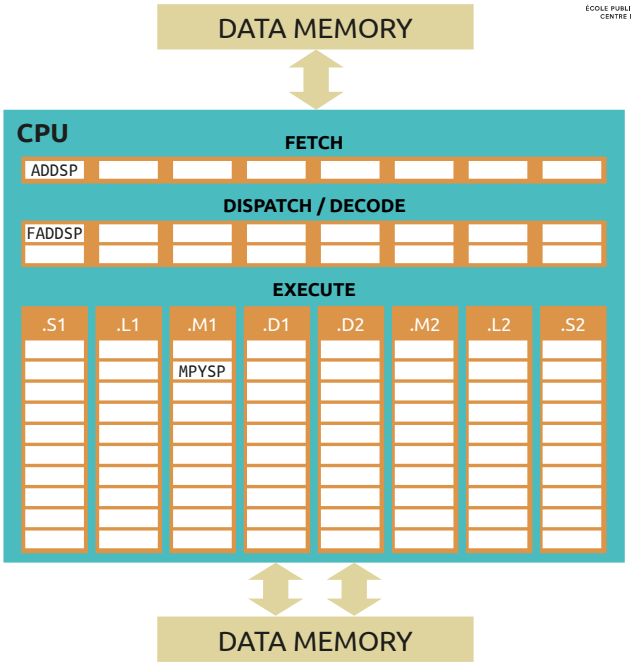
Optimized asm – 8 CPU cycles

```
    ...
    MPYSP    .M1     A2, A3, A4
    NOP              2
    FADDSP   .S1     A0, A1, A0
    ADDSP    .S1     A2, A4, A2
    NOP              2
    MV       .D1     A0, A1
||  MV       .D2     B9, B7
    ...
```



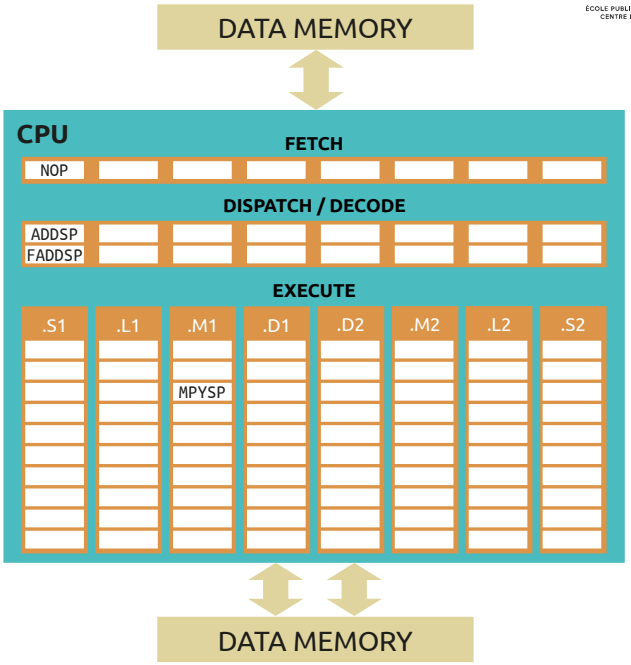
Optimized asm – 8 CPU cycles

```
    ...
    MPYSP    .M1     A2, A3, A4
    NOP              2
    FADDSP   .S1     A0, A1, A0
    ADDSP    .S1     A2, A4, A2
    NOP              2
    MV       .D1     A0, A1
||  MV       .D2     B9, B7
    ...
```



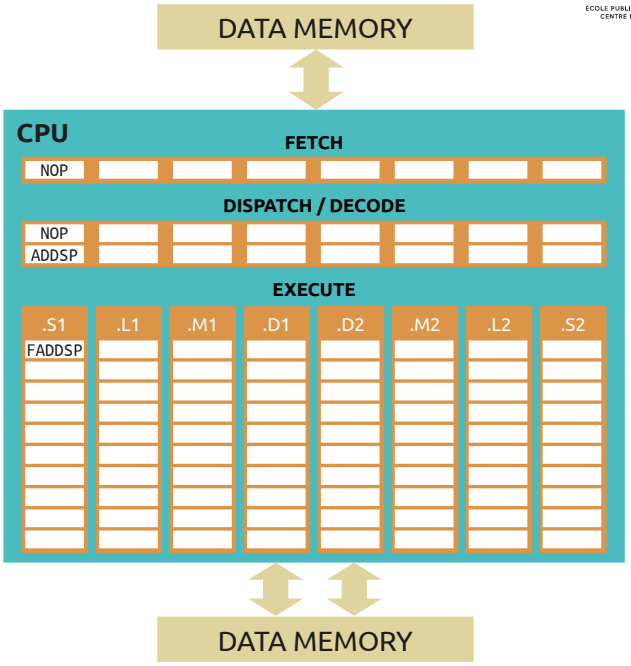
Optimized asm – 8 CPU cycles

```
    ...
    MPYSP    .M1     A2, A3, A4
    NOP              2
    FADDSP   .S1     A0, A1, A0
    ADDSP    .S1     A2, A4, A2
    NOP              2
    MV       .D1     A0, A1
||  MV       .D2     B9, B7
    ...
```



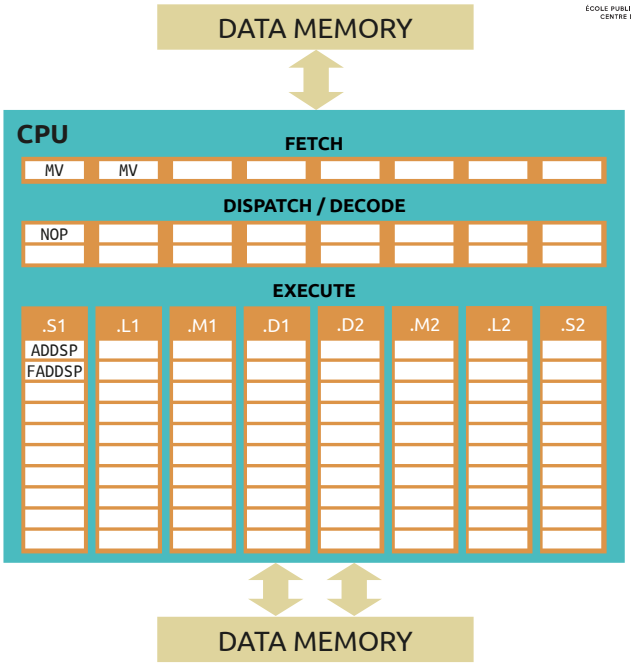
Optimized asm – 8 CPU cycles

```
   ...
   MPYSP    .M1     A2, A3, A4
   NOP              2
   FADDSP   .S1     A0, A1, A0
   ADDSP    .S1     A2, A4, A2
   NOP              2
   MV       .D1     A0, A1
|| MV       .D2     B9, B7
   ...
```



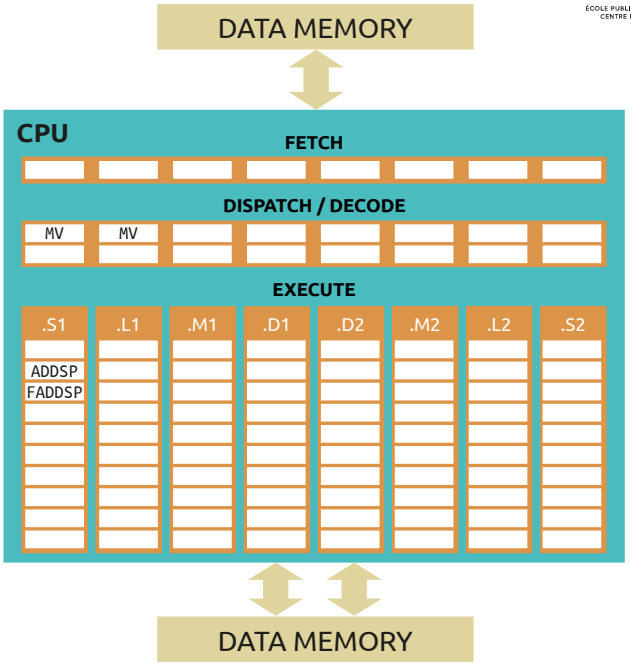
Optimized asm – 8 CPU cycles

```
   ...
   MPYSP    .M1     A2, A3, A4
   NOP              2
   FADDSP   .S1     A0, A1, A0
   ADDSP    .S1     A2, A4, A2
   NOP              2
   MV       .D1     A0, A1
|| MV       .D2     B9, B7
   ...
```



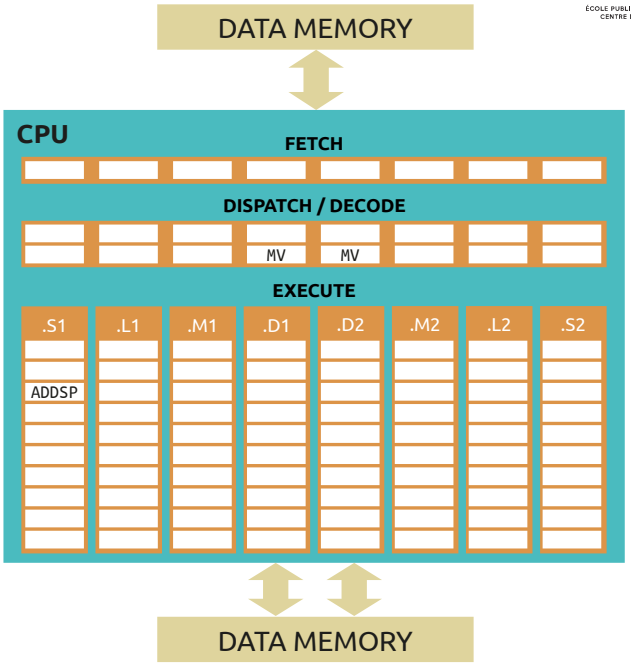
Optimized asm – 8 CPU cycles

```
   ...
   MPYSP    .M1     A2, A3, A4
   NOP              2
   FADDSP   .S1     A0, A1, A0
   ADDSP    .S1     A2, A4, A2
   NOP              2
   MV       .D1     A0, A1
|| MV       .D2     B9, B7
   ...
```



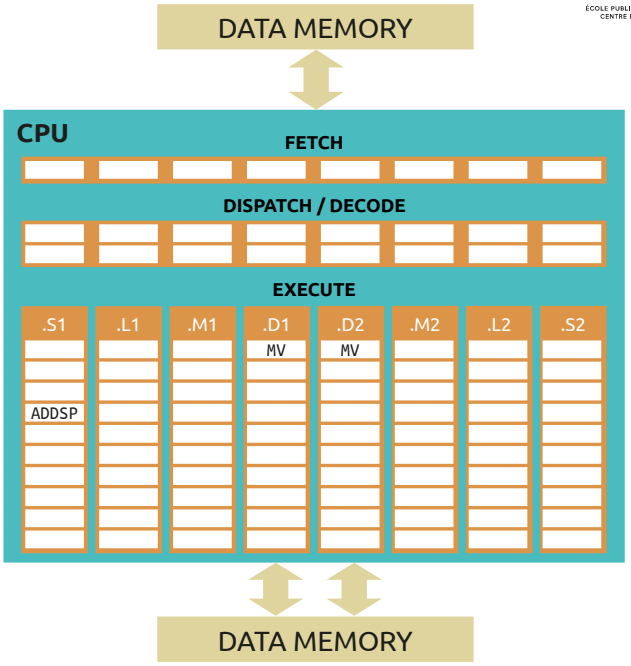
Optimized asm – 8 CPU cycles

```
   ...
   MPYSP    .M1     A2, A3, A4
   NOP              2
   FADDSP   .S1     A0, A1, A0
   ADDSP    .S1     A2, A4, A2
   NOP              2
   MV       .D1     A0, A1
|| MV       .D2     B9, B7
   ...
```



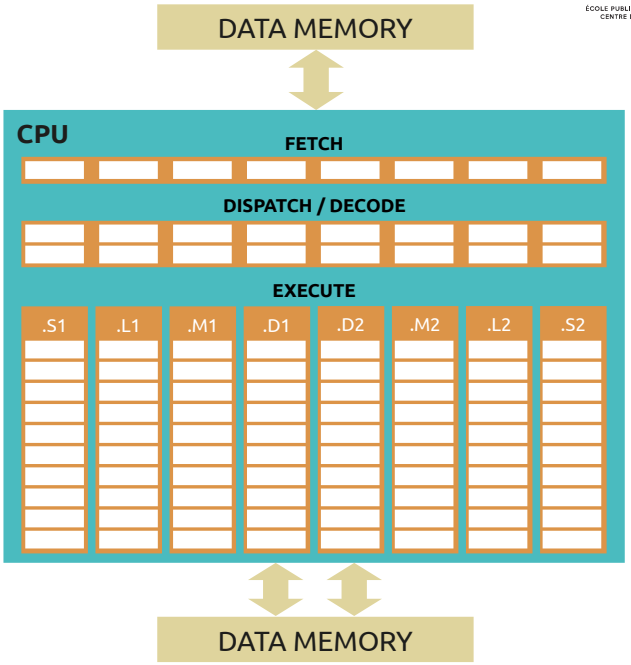
Optimized asm – 8 CPU cycles

```
   ...
   MPYSP    .M1     A2, A3, A4
   NOP              2
   FADDSP   .S1     A0, A1, A0
   ADDSP    .S1     A2, A4, A2
   NOP              2
   MV       .D1     A0, A1
|| MV       .D2     B9, B7
   ...
```



Optimized asm – 8 CPU cycles

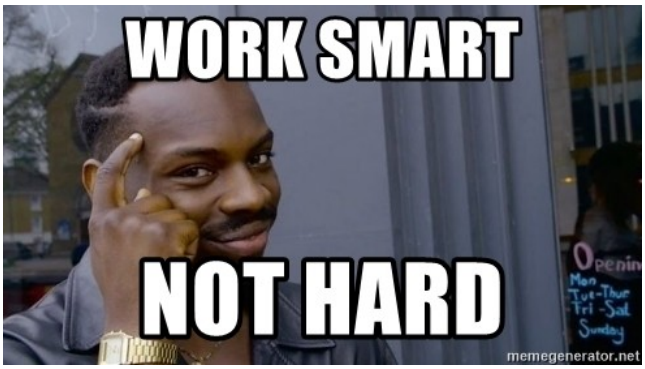
```
   ...
   MPYSP    .M1     A2, A3, A4
   NOP              2
   FADDSP   .S1     A0, A1, A0
   ADDSP    .S1     A2, A4, A2
   NOP              2
   MV       .D1     A0, A1
|| MV       .D2     B9, B7
   ...
```

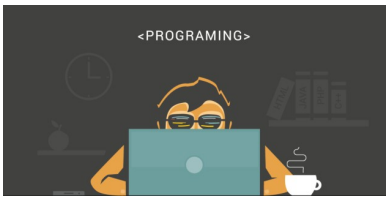


One particularity of VLIW processors is that their assembly code (and binary code as well) is out of order in the program memory, but they come out of the pipeline in order.

This very simple CPU has a very good performances/Watt ratio.

However, intelligence and skills belong to the developper and the toolchain.





VLIW CPU

- Intelligence bring by toolchain and engineer
- Memory program code is out of order
- Execution In Order
- Determinist
- Excellent performance/consumption ratio

Superscalar CPU

- Intelligence lies within the execution stage
- Memory program code is in order
- Execution is Out Of Order (OOO execution)
- Not determinist
- Bad performance/consumption ratio

On the one hand superscalar CPUs are designed to execute generic code with almost no optimisation and that includes lots of branches and tests. Keyword is genericity.

On the other hand VLIW CPUs must run target-dependant code in order to use their maximum capability. However this means architecture-specific code (no portability).

```
ptr_x2[l1] = xt1 * co1 + yt1 * si1;
ptr_x2[l1 + 1] = yt1 * co1 - xt1 * si1;
ptr_x2[h2] = xt0 * co2 + yt0 * si2;
ptr_x2[h2 + 1] = yt0 * co2 - xt0 * si2;
ptr_x2[l2] = xt2 * co3 + yt2 * si3;
ptr_x2[l2 + 1] = yt2 * co3 - xt2 * si3;
```

TI DSPLIB, FFT algorithm, floating point
Canonical implementation
→ PORTABLE

```
x_1o_x_0o = _daddsp(xh1_0_xh0_0, xh21_0_xh20_0);
x_3o_x_2o = _daddsp(xh1_1_xh0_1, xh21_1_xh20_1);

yt0_0_xt0_0 = _dsubsp(xh1_0_xh0_0, xh21_0_xh20_0);
yt0_1_xt0_1 = _dsubsp(xh1_1_xh0_1, xh21_1_xh20_1);
```

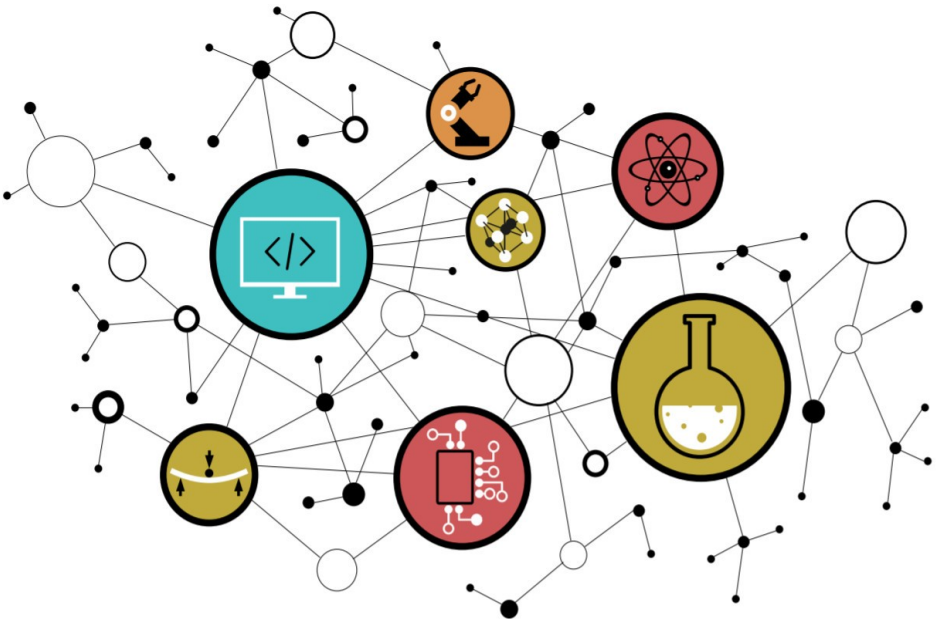
TI DSPLIB, FFT algorithm, floating point
Optimised implementation (intrinsec functions)
→ NOT PORTABLE

If one wants to use the full capability of a processor, he must master the hardware architecture as well as associated developping tools (i.e. toolchain).

Also one must be able to use math and rewrite the algorithm (and its implementation) with the aim of a code acceleration.

As a matter of fact, the most performant codes are most of the time not portable.

CONVOLUTION DISCRÈTE



Chapter 4

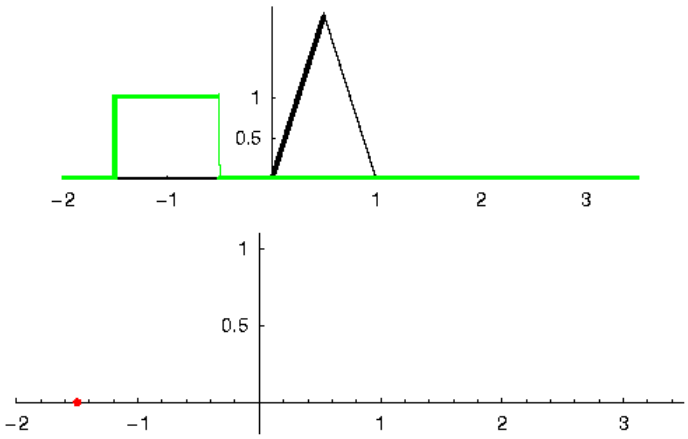
Lab's example algorithm

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2021-2022

Lab sessions will use a well known algorithm: the **discrete convolution**.

This algorithm has a very simple structure, but it is very difficult to accelerate without mathematical refactoring.



Let's have a look at the mathematical definition of the discrete convolution

$$y(k) = \sum_{k=0}^{Y} \sum_{j=0}^{N} a(j) \cdot x(k-j)$$

Where:

- x() is the input samples vector
- y() is the output samples vector
- a() is the coefficients vector
- Y is the output vector size
- N is the number of coefficients
- k is the index of the current sample

Before being coded in C onto the wanted processor, the algorithm is usually validated with prototyping and simulation tools, such as Matlab/Simulink.

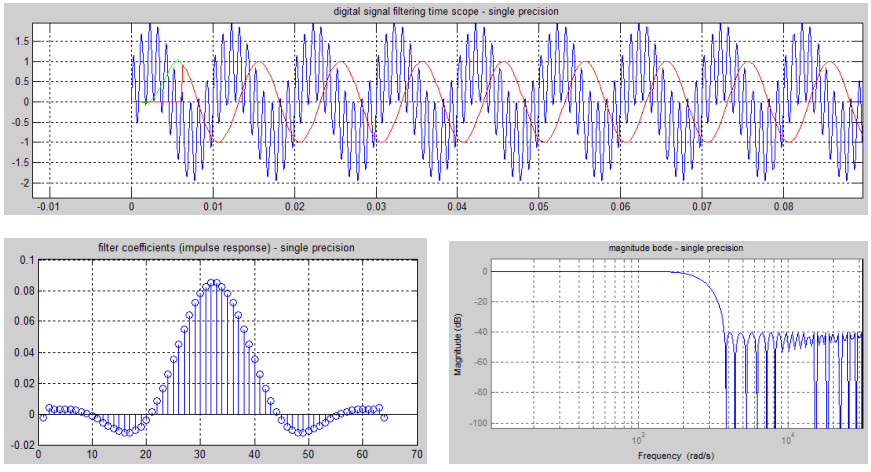
Validating the algorithm consists in coding its canonical implementation and check the input and output vectors values.

Here is the Matlab implementation of the discrete convolution algorithm.

```
function yk = fir_sp(xk, coeff, coeffLength, ykLength)
    yk = single(zeros(1,ykLength));     % output array preallocation

    % output array loop
    for i=2:ykLength
        yk(i) = single(0);

        % FIR filter algorithm - dot product
        for j=1:coeffLength
            yk(i) = single(yk(i)) + single(coeff(j)) * single(xk(i+j-1));
        end
    end
end
```

This code is given with lab materials

Observe some of the outputs suggested by Matlab sources, for a 64th-order FIR filter.



Matlab sources given with lab materials

Once the algorithm has been validated, it can be implemented in the processor.
First make a canonical C implementation, using **IEEE-754 single-precision floats**.

```
void fir_sp (   const float * restrict xk,   \
                const float * restrict a,    \
                float * restrict yk,         \
                int na,                      \
                int nyk){
    int i, j;

    for (i=0; i<nyk; i++) {
        yk[i] = 0;

        /* FIR filter algorithm - dot product */
        for (j=0; j<na; j++){
            yk[i] += a[j]*xk[i+j];
        }
    }
}
```


Another canonical C implementation.
This one is given by Texas Instruments in its library **dsplib**.

```
#pragma CODE_SECTION(DSPF_sp_fir_gen_cn, ".text:ansi");

#include "DSPF_sp_fir_gen_cn.h"

void DSPF_sp_fir_gen_cn(const float *x,
    const float *h,
    float *y,
    int nh,
    int ny)
{
    int i, j;
    float sum;

    for(j = 0; j < ny; j++)
    {
        sum = 0;

        // note: h coeffs given in reverse order: { h[nh-1], h[nh-2], ..., h[0] }
        for(i = 0; i < nh; i++)
            sum += x[i + j] * h[i];

        y[j] = sum;
    }
}
```

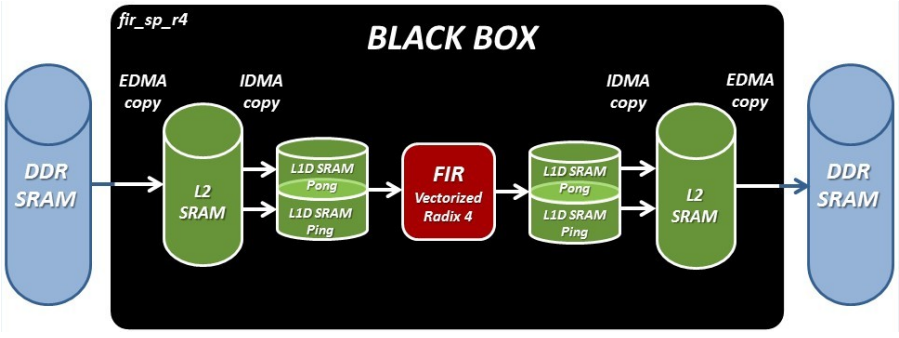
Another canonical C implementation, from the Texas Instruments **dsplib**.

But this time, it uses **16-bit signed integers** with the **Q1.15 format**.

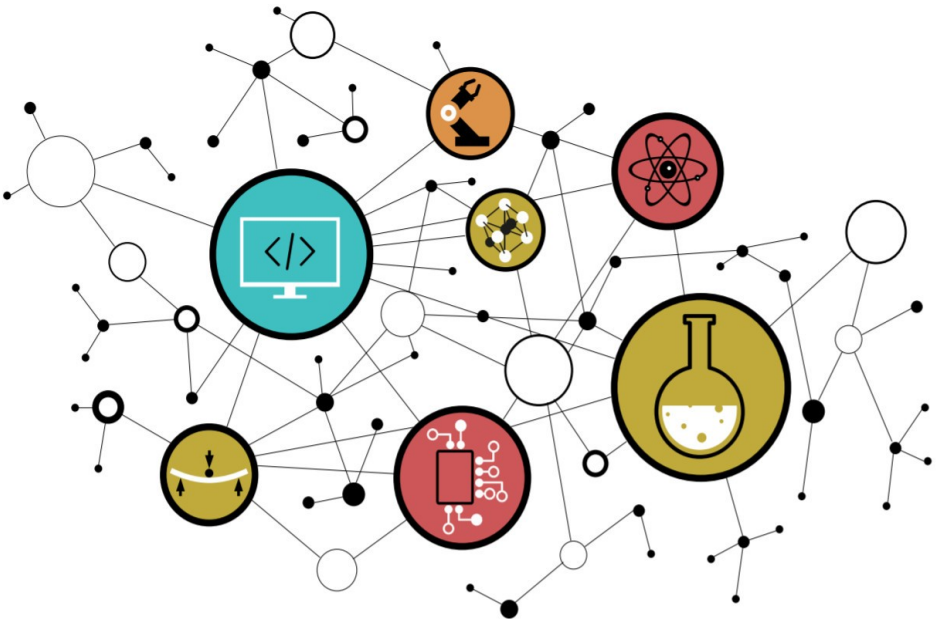
```
#pragma CODE_SECTION(DSP_fir_gen_cn, ".text:ansi");

#include "DSP_fir_gen_cn.h"

void DSP_fir_gen_cn (
    const short *restrict x,    /* Input array [nr+nh-1 elements] */
    const short *restrict h,    /* Coeff array [nh elements]      */
    short       *restrict r,    /* Output array [nr elements]     */
    int nh,                     /* Number of coefficients         */
    int nr                      /* Number of output samples       */
)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
}
```


The main goal of the lab sessions is to present a **generic methodology for optimizing algorithms for a specific architecture**.

In our case, we'll optimize a **discrete convolution algorithm for a TI C6678 DSP**.



ASSEMBLEUR C6000



Chapter 5

C6678's Assembly

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

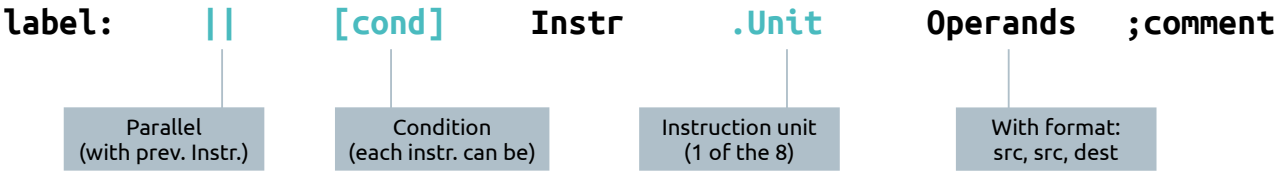
2021-2022

C6678 INSTRUCTION SET ARCHITECTURE

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

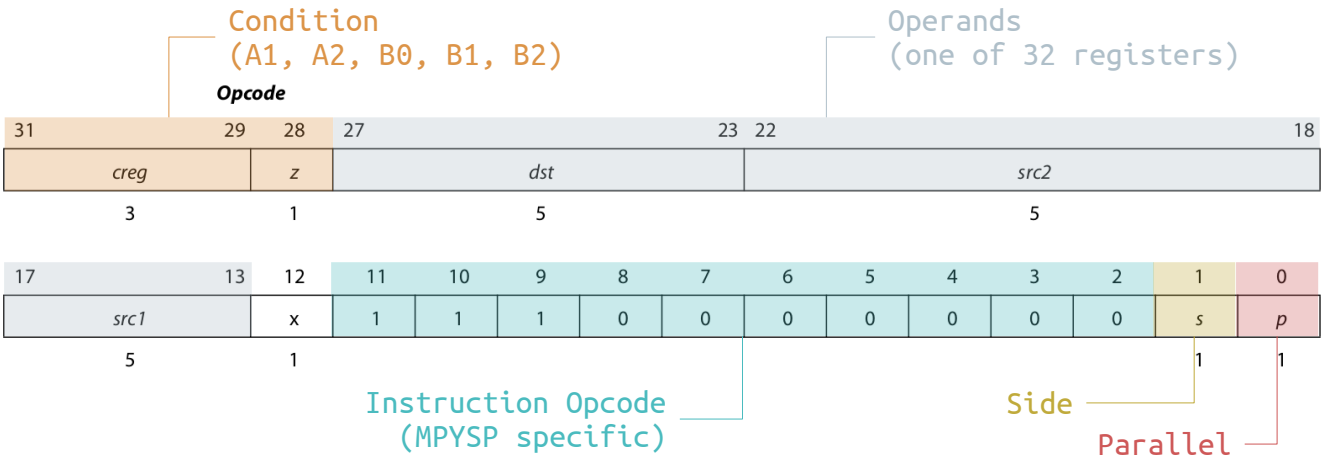
Let's see the different fields of an instruction line in assembly language for the Texas Instruments C6600 architectures.

Note that some fields are specific to VLIW architectures.

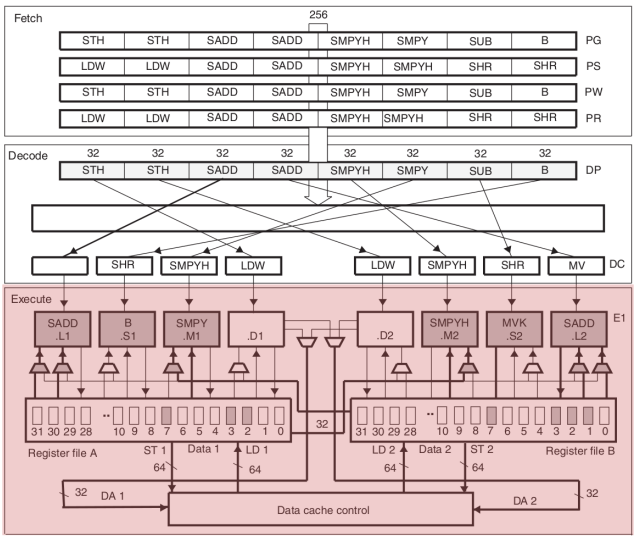


Remember that all fields of an instruction in assembly language correspond to a field in the binary code of the instruction (except for the label and the comment).

See for instance the MPYSP instruction.



In order to ease the understanding of the C6600 Instruction Set Architecture, we'll look at the effects of the assembly instructions onto the execution stage.



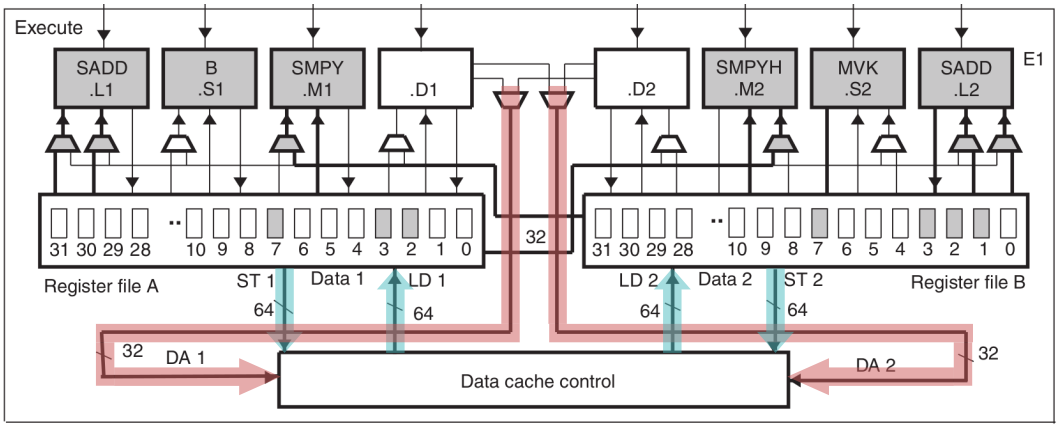
The **C6000 CPU has a Load-Store architecture**.

This means that some execution units (.D1 and .D2) are dedicated to memory access, and both of them have a direct access to the L1 cache memory (64-bit bus).

The other execution units are used for control and processing.

Data bus (64-bit)
L1 cache ↔ Reg file A/B

Memory address bus (32-bit)
.D1/.D2 → L1 cache



Only **3 addressing modes** are supported by the C6000 ISA.

Remind that addressing modes correspond to data manipulation strategies , as used by the instructions.

Being a calculation processor, the C6000 CPU heavily uses register addressing mode.

- **Register addressing**
  - 324 instructions (full ISA)
- **Indirect addressing**
  - 18 instructions (load/store instructions)
- **Immediate addressing**

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

DISCRETE CONVOLUTION ALGORITHM IN CANONICAL C6678 ASSEMBLY

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

For the remaining part of this lecture, we'll translate the C algorithm into a C6678 assembly language program.

The canonical C version of the program is on the next slide.

For educational purpose, we will ignore the delay slots that are associated to instructions execution time.

The absence of the NOP instructions will facilitate the understanding of the canonical assembly program.

Do not forget to add the delay slots when programming the target DSP!

This is the algorithm that will be studied during the lab sessions.
It's a canonical C implementation, using IEEE-754 single-precision floats.

```
void fir_sp (   const float * restrict xk,   \
                const float * restrict a,    \
                float * restrict yk,         \
                int na,                      \
                int nyk){
    int i, j;

    for (i=0; i<nyk; i++) {
        yk[i] = 0;

        /* FIR filter algorithm - dot product */
        for (j=0; j<na; j++){
            yk[i] += a[j]*xk[i+j];
        }
    }
}
```


Registers used for passing parameters through function call:

See "TMS320C6000 Optimizing Compiler V7.6" User's guide, Chapter "7.3 Register conventions"

```
void main(void)
{
    fir_sp( xk_sp, a_sp, yk_sp, A_LENGTH, YK_LENGTH );

    while(1);
}
```

Returned value in A4

B3 register used to save return address

A4 B4 A6 B6 A8

We decide to use those registers:

- i = B0
- j = A1
- yk (temp) = A5
- xk_sp = A19
- a_sp = B19
- xk = A9
- a = B9

The easiest way to translate the C into assembly language is to start from the main operation, inside the inner loop.

Like many other digital signal processing algorithms, the discrete convolution uses **MAC** (Multiply-Accumulate) or **SOP** (Sum Of Products) instructions.

First let's look at the MPYSP instruction.

4.212 MPYSP

Multiply Two Single-Precision Floating-Point Values

Syntax **MPYSP** (.unit) *src1, src2, dst*

unit = .M1 or .M2

```
fir_sp_asm:

        MPYSP .M1       A9,  B9, A17
```


The current example shows a cross path (i.e. the two sources are not from the same side).

This brings some limitations:

- The destination register and the execution unit must be on the same side
- Only one source register can be on the other side
- Add the 'x' suffix when specifying the execution unit

```
fir_sp_asm:

        MPYSP .M1x      A9,  B9, A17
```

Now we can use an addition instruction.

4.14 ADDSP

Add Two Single-Precision Floating-Point Values

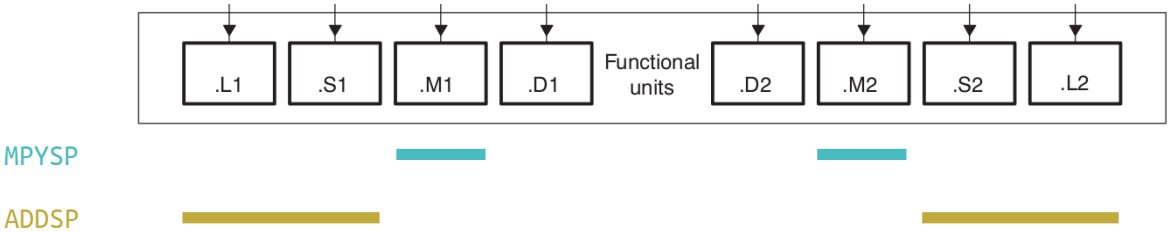
Syntax **ADDSP** (.unit) *src1, src2, dst*

unit = .L1, .L2, .S1, .S2

```
fir_sp_asm:

        MPYSP .M1x      A9,  B9, A17
        ADDSP .L1       A17, A5, A5
```

Well done!
You just implemented a MAC operation!

Use of execution units:



Move data from a CPU register to another one.

4.222 MV

Move From Register to Register

Syntax **MV** (.unit) *src2, dst*

unit = .L1, .L2, .S1, .S2, .D1, .D2

```
fir_sp_asm:
        MV   .L1      A8, B0

        MPYSP .M1x    A9,  B9, A17
        ADDSP .L1     A17, A5, A5
```


Before performing the MAC, we must load the cells values (stored in the L1 cache memory) into CPU registers.

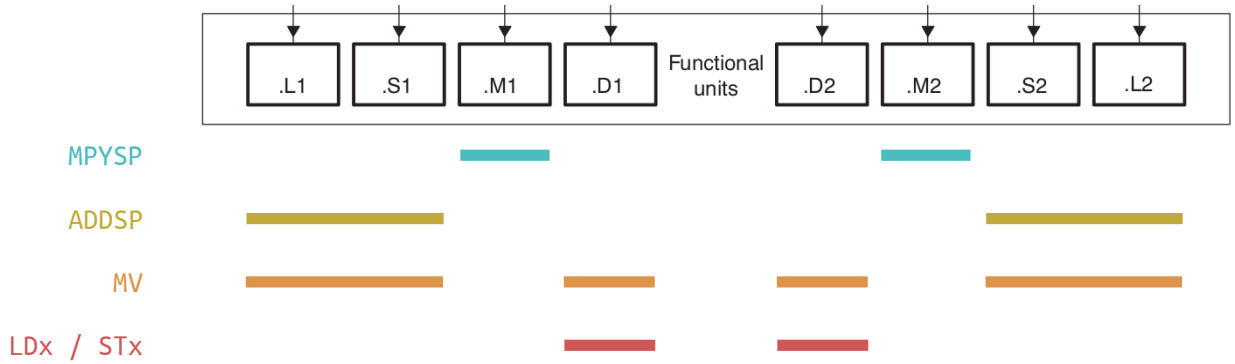
We must use one of the LDx (load) instructions:

- `LDB,  B  = Byte        = 1 byte  = char`
- `LDH,  H  = Half-word   = 2 bytes = short int`
- `LDW,  W  = Word        = 4 bytes = int, float`
- `LDDW, DW = Double-Word = 8 bytes = long, double`

The .D1 and .D2 executions units are dedicated to ST and LD instructions only.

```
fir_sp_asm:
        MV   .L1      A8, B0

        LDW   .D1     *A19,  A9
        LDW   .D2     *B19,  B9
        MPYSP .M1x    A9,  B9, A17
        ADDSP .L1     A17, A5, A5
```

Use of execution units:



Note that a pointer-like style is used.

In this example case, A19 and B19 registers contain each an address. The '*' character before the register name indicates the use of **indirect addressing mode**.

This is equivalent to the use of pointers in C.

```
fir_sp_asm:
        MV    .L1       A8, B0




        LDW   .D1       *A19,  A9
        LDW   .D2       *B19,  B9
        MPYSP .M1x      A9,  B9, A17
        ADDSP .L1       A17, A5, A5
```


Similarly to the pointers in C language, registers used in indirect addressing mode support **pre- and post-incrementations**.

Also, load and store operations can **be indexed with the [] notation**, like arrays in C.

Table 3-10 Indirect Address Generation for Load/Store

Addressing Type	No Modification of Address Register	Preincrement or Predecrement of Address Register	Postincrement or Postdecrement of Address Register
Register indirect	*R	*++R	*R++
		*- -R	*R- -
Register relative	*+R[*ucst5*]	*++R[*ucst5*]	*R++[*ucst5*]
	*-R[*ucst5*]	*- -R[*ucst5*]	*R- -[*ucst5*]
Register relative with 15-bit constant offset	*+B14/B15[*ucst15*]	not supported	not supported
Base + index	*+R[*offsetR*]	*++R[*offsetR*]	*R++[*offsetR*]
	*-R[*offsetR*]	*- -R[*offsetR*]	*R- -[*offsetR*]

To summarize:

The A19 and B19 registers contain the address of the current cell of a[] and xk[] arrays.

The two LDW instructions load 4 bytes from the L1 cache memory to the CPU registers A9 & B9.

The address value contained in A19 and B19 registers are incremented afterward, making these registers pointing to the next cell.

```
fir_sp_asm:
         MV    .L1       A8, B0




         LDW   .D1       *A19++,  A9
         LDW   .D2       *B19++,  B9
         MPYSP .M1x      A9,  B9, A17
         ADDSP .L1       A17, A5, A5
```

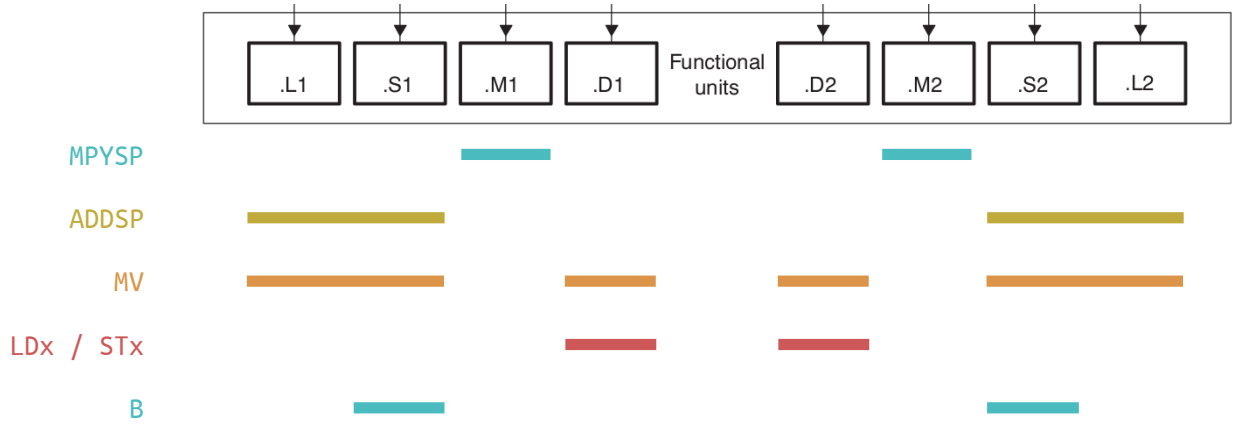
The C6000 family historically supported only one branch instruction: the **B (branch) instruction**.

It allows to perform function calls as well as all known control structures (if, for, while, …).

The B instruction uses .S1 and .S2 execution units.

```
fir_sp_asm:
        MV    .L1       A8, B0




fir_sp_asm_l2:
        LDW   .D1       *A19++,  A9
        LDW   .D2       *B19++,  B9
        MPYSP .M1x      A9,  B9, A17
        ADDSP .L1       A17, A5, A5

        B     .S1       fir_sp_asm_l2
```


Use of execution units:



A condition can be added to the execution of an instruction.

Five registers (A1, A2, B0, B1, B2) can be used as condition values.

Syntax:

- `[R]` = instruction executed if R ≠ 0
- `[!R]` = instruction executed if R = 0

All instructions can be executed conditionally.

```
fir_sp_asm:
         MV    .L1       A8, B0




fir_sp_asm_l2:
         LDW   .D1       *A19++,  A9
         LDW   .D2       *B19++,  B9
         MPYSP .M1x      A9,  B9, A17
         ADDSP .L1       A17, A5, A5

    [A1] B     .S1       fir_sp_asm_l2
```


Implement the internal loop's counter.

```
fir_sp_asm:
         MV    .L1       A8, B0



         MV    .L1       B6, A1


fir_sp_asm_l2:
         LDW   .D1       *A19++,  A9
         LDW   .D2       *B19++,  B9
         MPYSP .M1x      A9,  B9, A17
         ADDSP .L1       A17, A5, A5
    [A1] SUB   .L1       A1,  1,  A1
    [A1] B     .S1       fir_sp_asm_l2
```

Implement the external loop.

```
fir_sp_asm:
         MV    .L1       A8, B0

fir_sp_asm_l1:

         MV    .L1       B6, A1



fir_sp_asm_l2:
         LDW   .D1       *A19++,  A9
         LDW   .D2       *B19++,  B9
         MPYSP .M1x      A9,  B9, A17
         ADDSP .L1       A17, A5, A5
    [A1] SUB   .L1       A1,  1,  A1
    [A1] B     .S1       fir_sp_asm_l2



    [B0] SUB   .L2       B0, 1, B0
    [B0] B     .S1       fir_sp_asm_l1
```


The **return address** of a function is always given by the calling function through the **B3 register**.

```
fir_sp_asm:
         MV    .L1       A8, B0

fir_sp_asm_l1:

         MV    .L1       B6, A1



fir_sp_asm_l2:
         LDW   .D1       *A19++,  A9
         LDW   .D2       *B19++,  B9
         MPYSP .M1x      A9,  B9, A17
         ADDSP .L1       A17, A5, A5
    [A1] SUB   .L1       A1,  1,  A1
    [A1] B     .S1       fir_sp_asm_l2



    [B0] SUB   .L2       B0, 1, B0
    [B0] B     .S1       fir_sp_asm_l1

         B               B3
```

Final version

Without the required delay slots!

```
void fir_sp (   const float * restrict xk,   \
                const float * restrict a,    \
                float * restrict yk,         \
                int na,                      \
                int nyk){
    int i, j;

    for (i=0; i<nyk; i++) {
        yk[i] = 0;

        /* FIR filter algorithm - dot product */
        for (j=0; j<na; j++){
            yk[i] += a[j]*xk[i+j];
        }
    }
}
```

```
fir_sp_asm:
         MV    .L1       A8, B0

fir_sp_asm_l1:
         ZERO  .L1       A5
         MV    .L1       B6, A1
         MV    .L1       A4, A19
         MV    .L1       B4, B19

fir_sp_asm_l2:
         LDW   .D1       *A19++,  A9
         LDW   .D2       *B19++,  B9
         MPYSP .M1x      A9,  B9, A17
         ADDSP .L1       A17, A5, A5
    [A1] SUB   .L1       A1,  1,  A1
    [A1] B     .S1       fir_sp_asm_l2

         STW   .D1       A5,   *A6++
         ADD   .L1       A4, 4, A4
    [B0] SUB   .L2       B0, 1, B0
    [B0] B     .S1       fir_sp_asm_l1

         B                B3
```

A

- ABI : Application Binary Interface
- ADC : Analog to Digital Converter
- ALU : Arithmetic and Logical Unit
- AMD : Advanced Micro Devices
- ANSI : American National Standards Institute
- AP : Application Processor
- API : Application Programming Interface
- APU : Accelerrated Processor Unit
- ARM : Société anglaise fabless concevant des CPU RISC 32bits
- ASCII : American Standar Code for Information Interchange
- ASIC : Application Specific Integrated Circuit

B

- BP : Base Pointer
- BSL : Board Support Library
- BSP : Board Support Package

C

- CCS : Code Composer Studio
- CEM : Compatibilité ElectroMagnétique
- CISC : Complex Instruction Set Computer
- CMS : Composant Monté en Surface
- CPU : Central Processing Unit
- CSL : Chip Support Library

D

- DAC : Digital to Analog Converter
- DDR : Double Data Rate
- DDR SDRAM: Double Data Rate Synchronous Dynamic Random Access Memory
- DMA : Dual Inline Package (boîtier de composant)
- DMA : Direct Memory Access
- DSP : Digital Signal Processor
- DSP : Digital Signal Processing

E

- EDMA : Enhanced Direct Memory Access
- EUSART : Enhanced Universal Synchronous Asynchronous Receiver Transmitter
- EMIF : External Memory Interface
- EPIC : Explicitly Parallel Instruction Computing

F

- FPU : Floating Point Unit
- FLOPS : Floating-Point Operations Per Second
- FMA: Fused Multiply-Add

G

- GCC : Gnu Collection Compiler
- GLCD : Graphical Liquid Crytal Display
- GNU : GNU's Not UNIX
- GPIO : General Purpose Input Output
- GPGPU : General Purpose GPU
- GPP : General Purpose Processor ou MPU
- GPU : Graphical Processing Unit

I

- IA-64 : Intel Architecture 64bits
- I2C : Inter Integrated Circuit
- IC : Integrated Circuit
- ICC : Intel C++ Compiler
- ICC : Interface Chaise Clavier (main problem root)
- IDE : Integrated Development Environment
- IDMA : Internal Direct memory Access
- IHM : Interface Homme Machine
- IRQ : Interrupt ReQuest
- ISR : Interrupt Software Routine
- ISR : Interrupt Service Routine

L

- L1D : Level 1 Data Memory
- L1I : Level 1 Instruction Memory (idem L1P)
- L1P : Level 1 Program Memory (idem L1I)
- Lx : Level x Memory
- LCD : Liquid Crytal Display
- LRU : Least Recently Used

M

- MAC: Multiply Accumulate
- MCU : Micro Controller Unit
- MFLOPS : Mega Floating Point Operations Per Second
- MIMD : Multiple Instructions Multiple Datas
- MIPS : Mega Instructions Per Second
- MISD : Multiple Instructions Single Data
- MMACS : Mega MAC's Per Second (cf. définition MAC ci-dessus)
- MMU : Memory Managment Unit
- MPLABX : MicrochiP LABoratory 10, IDE Microchip
- MPU : Micro Processor Unit ou GPP
- MPU : Memory Protect Unit
- MPPA : Massively Parallel Processor Array

O

- OS : Operating System

P

- PC : Program Counter
- PC : Personal Computer
- PCB : Printed Circuit Board
- PIC18 : Famille MCU 8bits Microchip
- PIC : Programmable Interrupt Controller
- PLD : Programmable Logic Device
- POSIX : Portable Operating System Interface (norme IEEE 1003)
- PPC : Power PC

R

- RAM : Random Access Memory
- RISC : Reduced Instruction Set Computer
- RS232 : Norme standardisant un protocole série asynchrone
- RTOS : Real Time Operating System
- RTS : Real Time System

S

- SDK : Software Development Kit
- SIMD : Single Instruction Multiple Date
- SIP : System In Package
- SOB : System On Board
- SOC : System On Chip
- SOP : Sums of products
- SP : Stack Pointer
- SP : Serial Port
- SPI : Serial Peripheral Interface
- SRAM : Static Random Access Memory
- SSE : Streaming SIMD Extensions
- STM32 : STMicroelectronics 32bits MCU

T

- TI : Texas Instruments
- TNS : Traitement Numérique du Signal
- TSC : Time Stamp Counter
- TTM : Time To Market

U

- UART : Universal Asynchronous Receiver Transmitter
- USB : Universal Serial Bus

V

- VHDL : VHSIC Hardware Description langage
- VHSIC : Very High Speed Integrated Circuit
- VLIW : Very Long Intruction Word

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES