Chapitre 2

Architecture du TI C6678

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

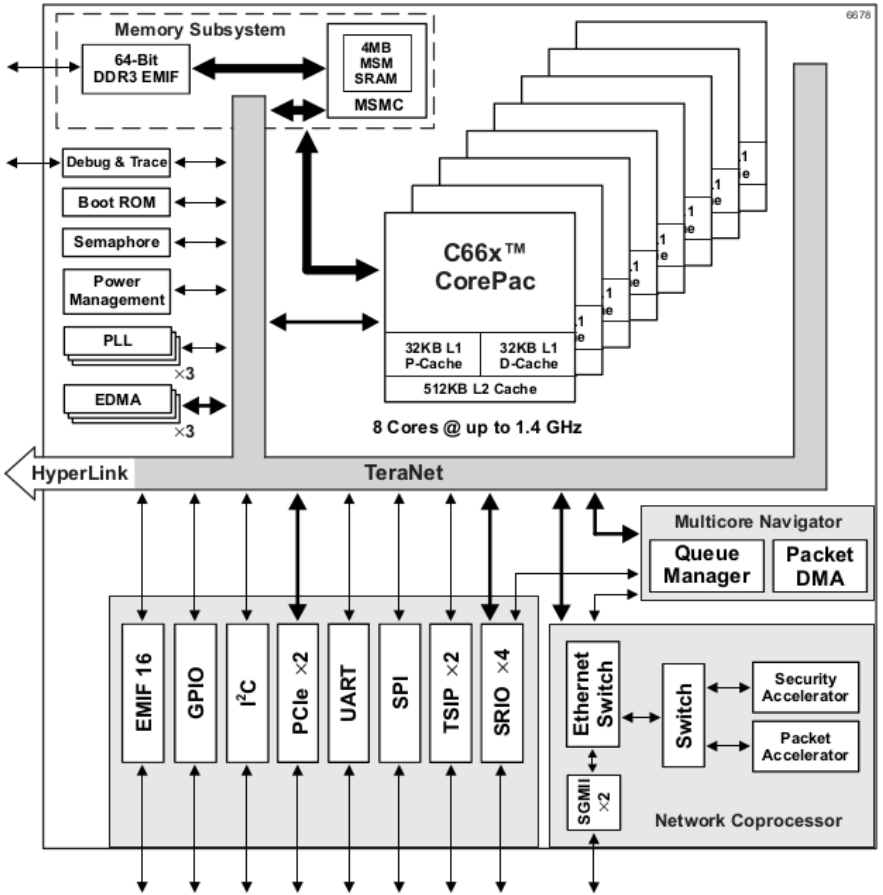
Caractéristiques du processeur et du cœur

Le **C6600** de Texas Instruments un est DSP multicœur avec une architecture CPU homogène.

Il contient **8 CPU VLIW et RISC-like**, cadencés jusqu'à 1,4 GHz.

→ 44.8 GMAC/core for fixed point @1.4 GHz

→ 22.4 GFLOP/core for floating point @1.4 GHz

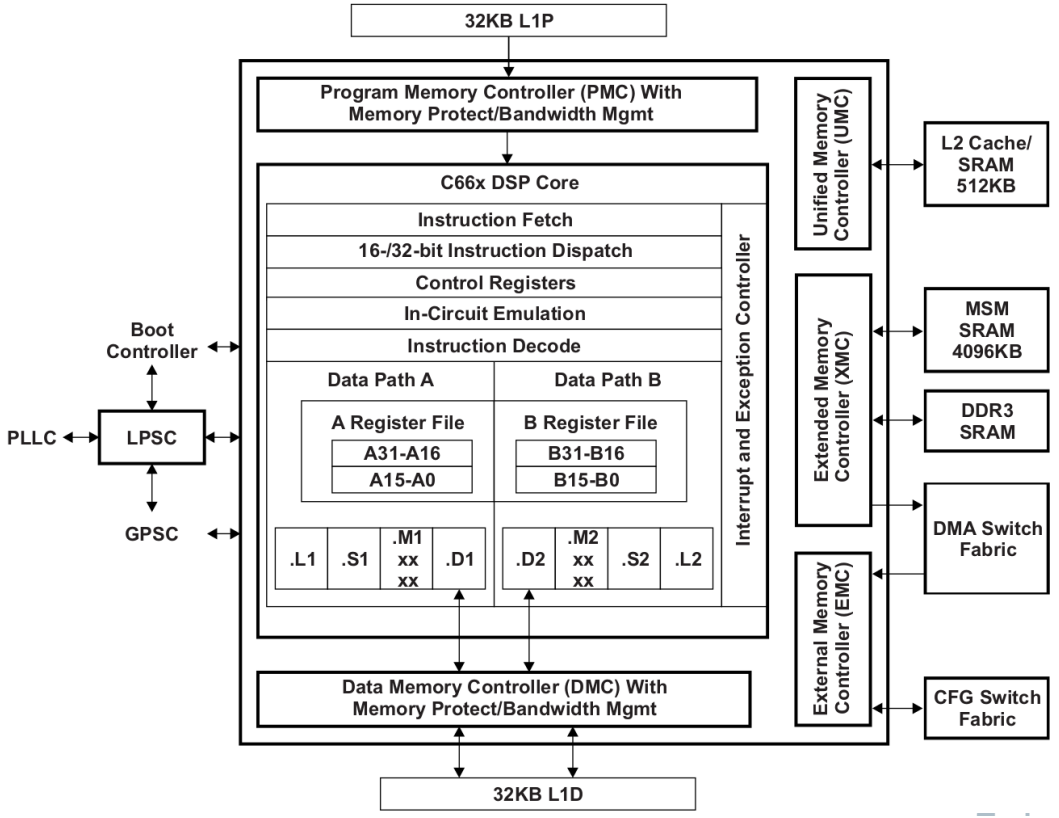


TMS320C6678 functional block diagram

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Chaque **cœur CorePac C66xx** est constitué de :

- 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)

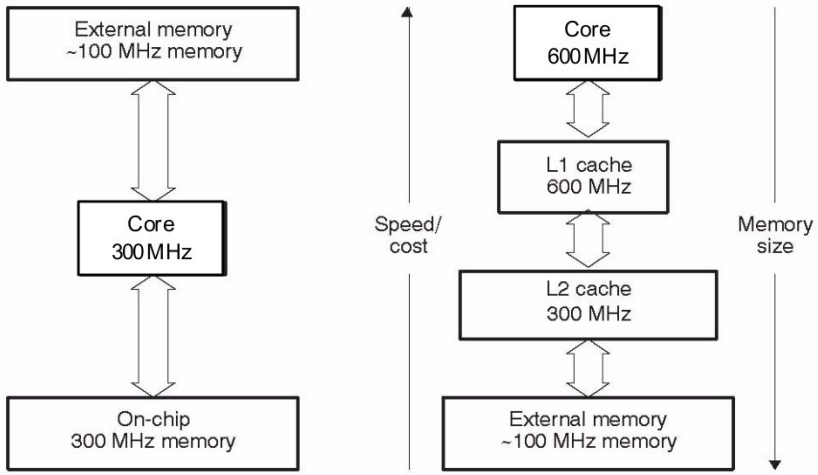


TMS320C66x CorePac DSP Block Diagram

Chaque **cœur** possède ses propres mémoires caches :

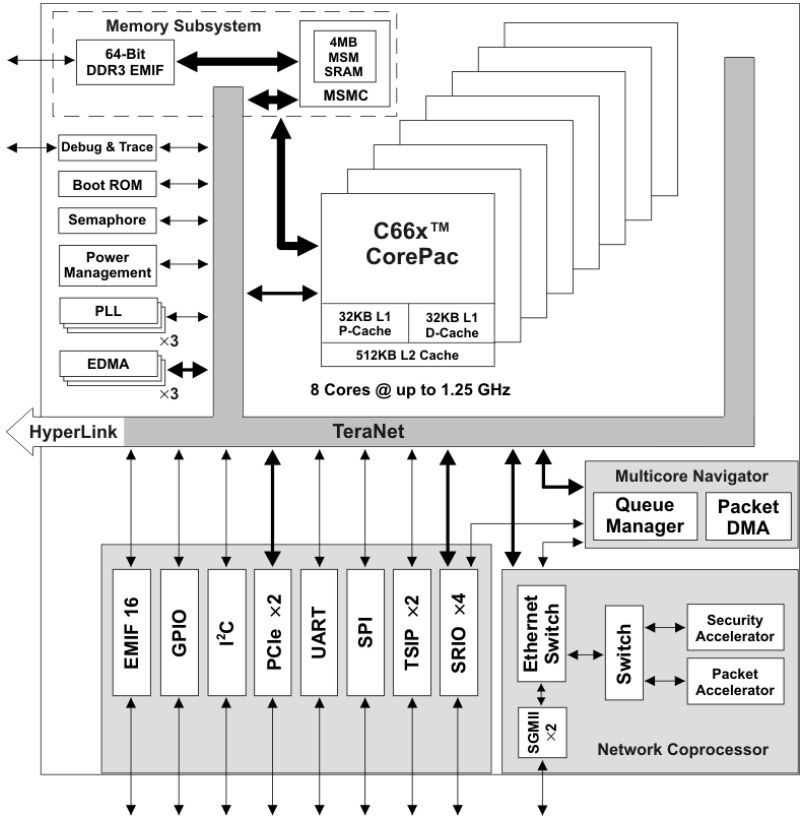
- 32 kB L1P cache memory
- 32 kB L1D cache memory
- 512 kB L2 cache memory

Figure 1-1 Flat Versus Hierarchical Memory Architecture



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

De plus, chaque cœur peut accéder à **4 MB de mémoire partagée** (*Multicore Shared Memory*, MSM), configurable en tant que mémoire cache ou SRAM adressable.



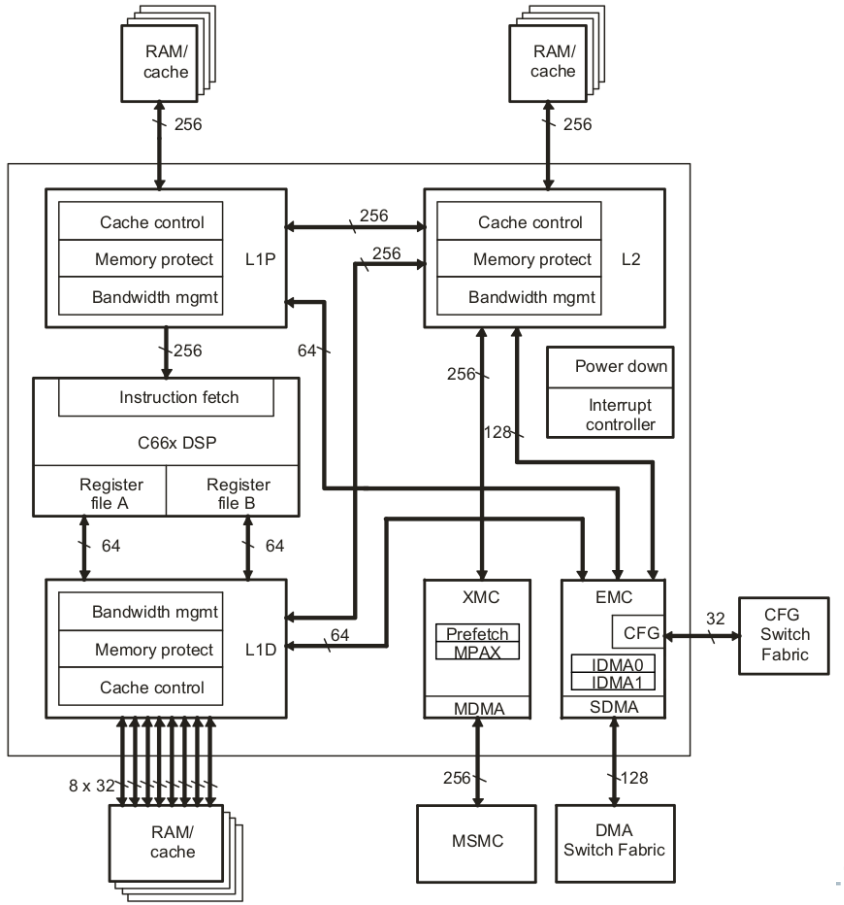
- Multicore Shared Memory Controller (MSMC)
  - 4096KB MSM SRAM Memory Shared by Eight DSP C66x CorePacs
  - Memory Protection Unit for Both MSM SRAM and DDR3_EMIF

Architecture du cœur CorePac

Le IDMA (*Internal Direct Memory Access*) est un **contrôleur DMA** propre à chaque CorePac.

Il est configurable et entièrement accessible par le développeur.

Le IDMA peut assurer le transfert de données entre mémoires locales au cœur, ou entre mémoires locales et périphériques (*peripheral configuration space*, CFG).

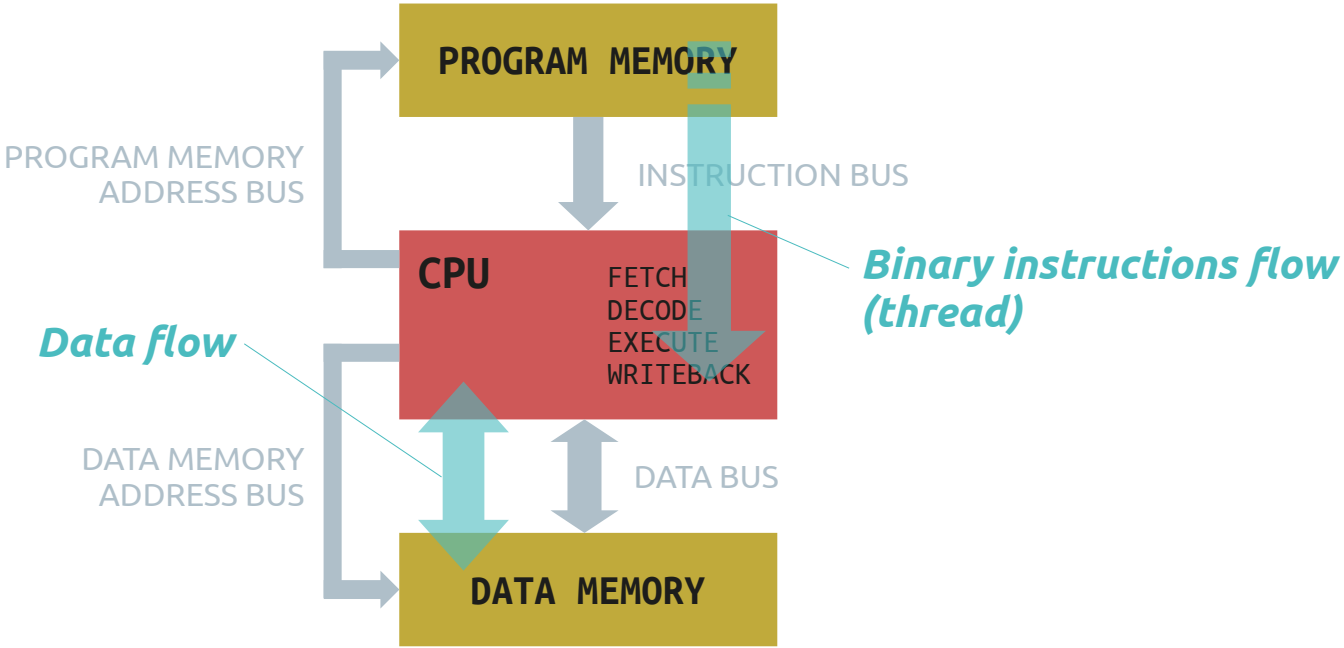


C6600 HARDWARE PIPELINE

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Rappel : un CPU est une machine séquentielle, mais il est capable de traiter plusieurs instructions à la fois grâce aux étages de son pipeline matériel.



Le pipeline du C6600 comporte 16 étages (appelés *phases*)

Figure 5-5 Pipeline Phases

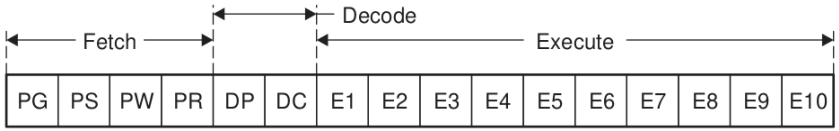
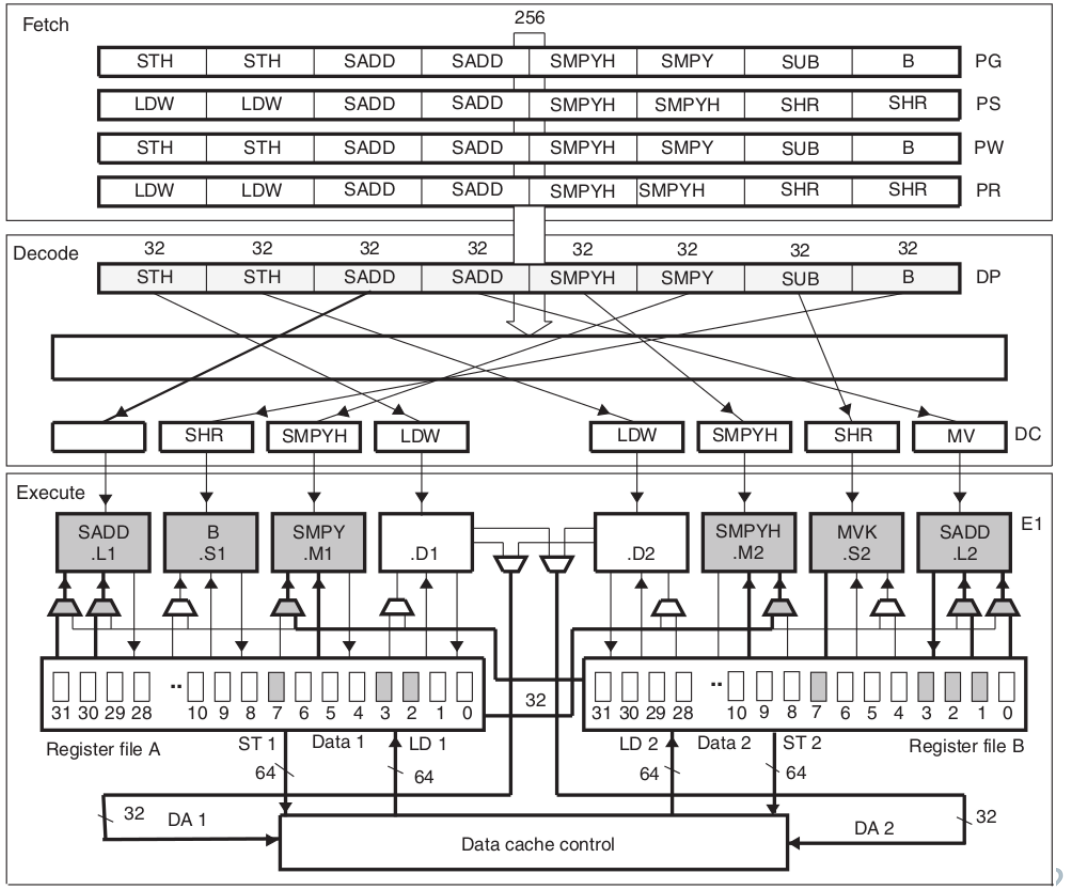


Figure 5-6 Pipeline Operation: One Execute Packet per Fetch Packet

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

Les CPU de la famille C6600 sont munis d'une **pipeline matériel VLIW (*Very Long Instruction Word*)**.

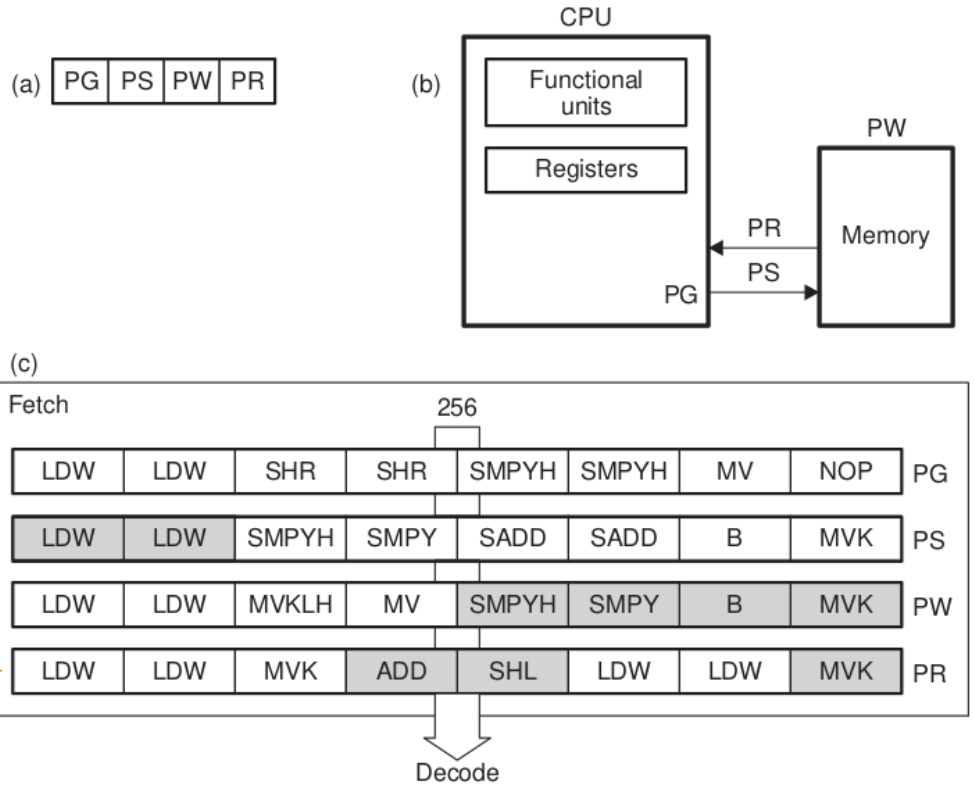
Avec ses 8 unités d'exécution, il peut exécuter jusqu'à 8 instructions à la fois.



Pipeline Phases Block Diagram

L'étage **FETCH** est divisé en quatre phases :

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



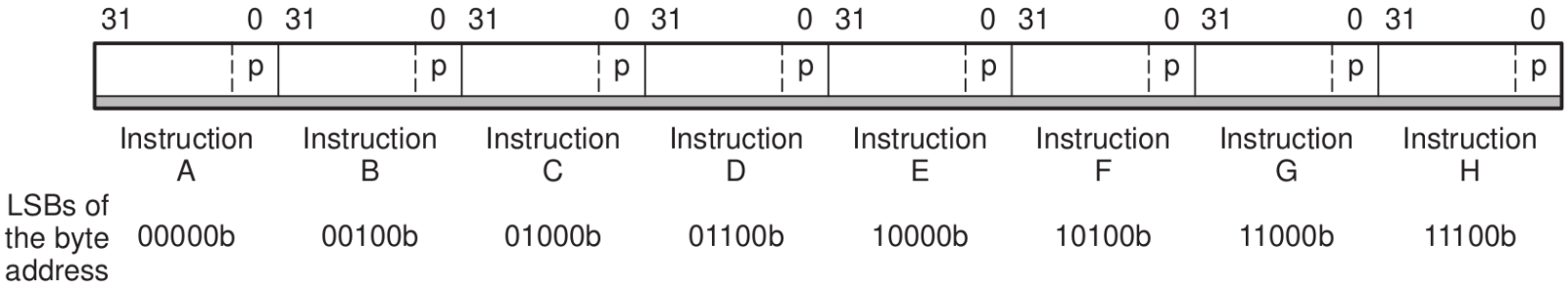
Chaque paquet récupéré peut faire jusqu'à 8 mots (8 x 32 bits = 256 bits).

Chaque instruction a une taille fixe de 32 bits (**jeu d'instructions RISC-like**).

Le tout dernier bit de chaque instruction se nomme P (*Parallel*) et est affecté à '0' ou '1' soit pendant la phase de compilation, soit explicitement par le développeur (en langage d'assemblage).

En lisant ce bit, l'étage de **FETCH** sait s'il doit également récupérer le mot suivant (c-à-d l'instruction suivante), dans une limite de 8 instructions à la fois.

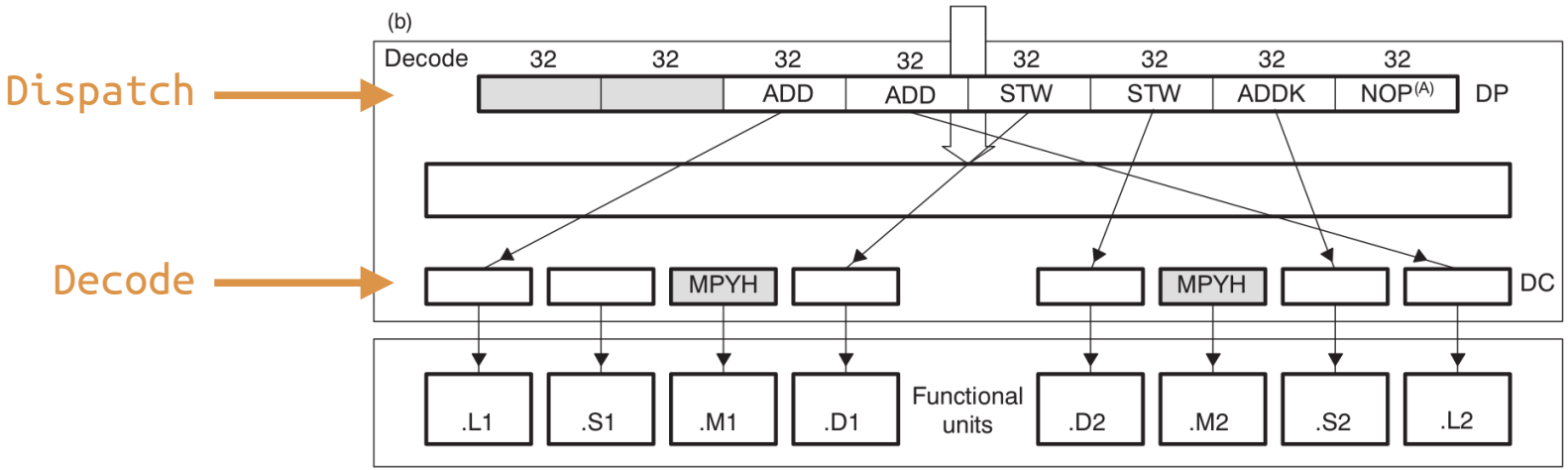
Figure 3-3 **Basic Format of a Fetch Packet**



Avec les instructions arrivant par paquets, l'étage **DECODE** se fait en deux phases :

1. La phase *Dispatch* redirige les instructions vers leur Unité d'Exécution appropriée
2. Chaque Unité d'Exécution dispose de sa propre unité de décodage.

Figure 5-3 Decode Phases of the Pipeline



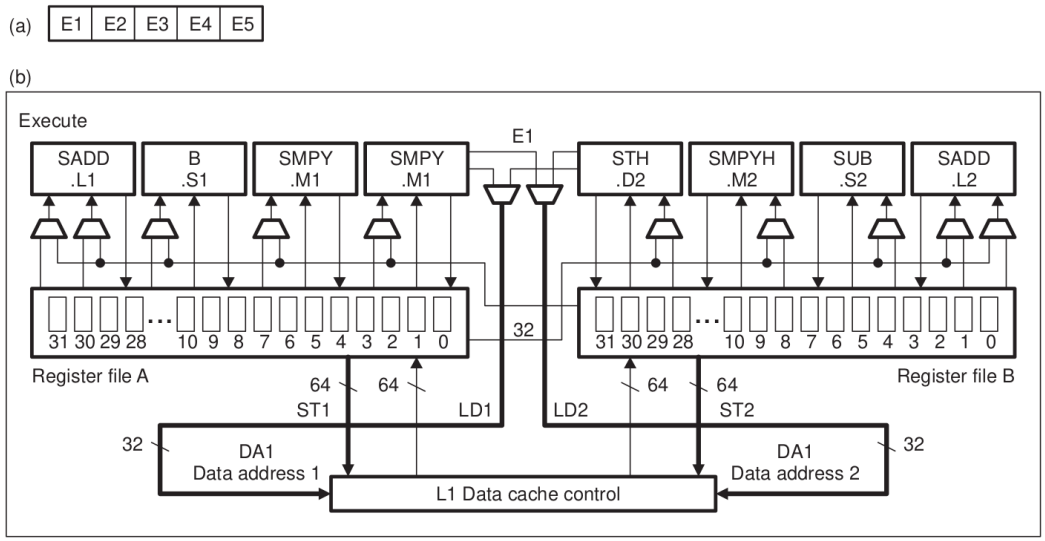
(A) NOP is not dispatched to a functional unit.

L'étage **EXECUTE** possède 8 Unités d'Exécutions (ou *Functional Units*).

Les Unités d'Exécution sont appelées .L1, .S1, .M1, .D1, .L2, .S2, .M2, .D2 et certaines instructions ne peuvent être exécutées que par certaines EU.

Les EU sont réparties en deux côtés symétriques, chacun ayant sa propre file de registres 32-bits.

Figure 5-4 Execute Phases of the Pipeline



Les instructions suivent alors le pipeline dédié à leur unité d'exécution.

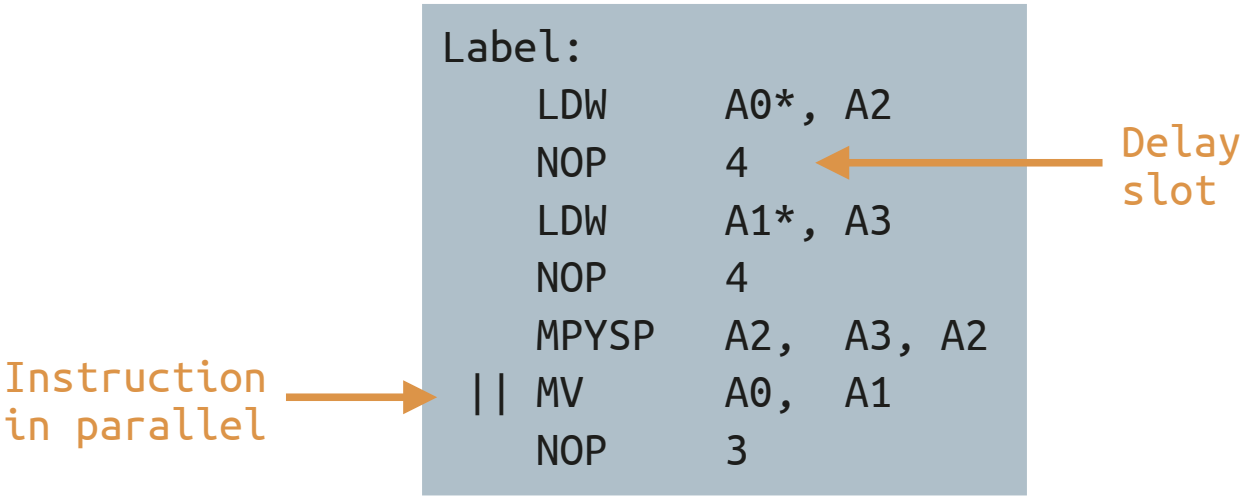
Table 5-1 Operations Occurring During Pipeline Phases (Part 2 of 2)

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.**

Les instructions prenant plus d'un temps de cycle sont suivies par un *delay slot*, écrit avec une instruction **NOP (*No Operation*)**.

La valeur du *delay slot* (nombre de NOP) correspond au temps que met l'instruction à traverser l'Unité d'Exécution.



À titre d'exemple, voici un extrait de documentation pour l'instruction MPYSP.

4.212 MPYSP

Multiply Two Single-Precision Floating-Point Values

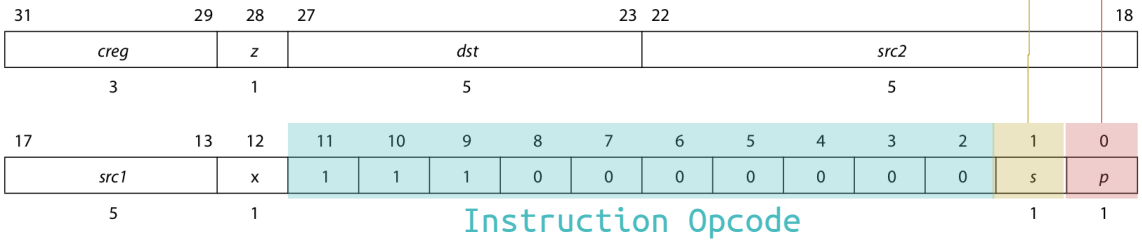
Syntax **MPYSP** (.unit) *src1, src2, dst*

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

PROGRAMMER UN CPU VLIW

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Exemple

Regardons comment un **CPU VLIW (*Very Long Instruction Word*)** fonction en se focalisant uniquement sur ses unités d'exécution.

Débutons avec un code en assembleur canonique.

```
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

Maintenant, trions les instructions par dépendance des données.

On trouve trois lots d'instructions indépendants.

```
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
```

Ces branches peuvent théoriquement être exécutées en parallèle sans problème de cohérence des données (placement parallèle arbitraire).

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
```

Cependant, nous devons prêter attention aux dépendances matérielles !

8 CPU cycles

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

Same unit

4 CPU cycles

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

1 CPU cycle

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

Ré-écriture du code (*refactoring*)

Ré-écrivons (*refactoring*) le code pour rendre possible l'exécution en parallèle.

8 CPU cycles

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

5 CPU cycles

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

1 CPU cycle

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

Voici donc les versions canonique et optimisée du même 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
```

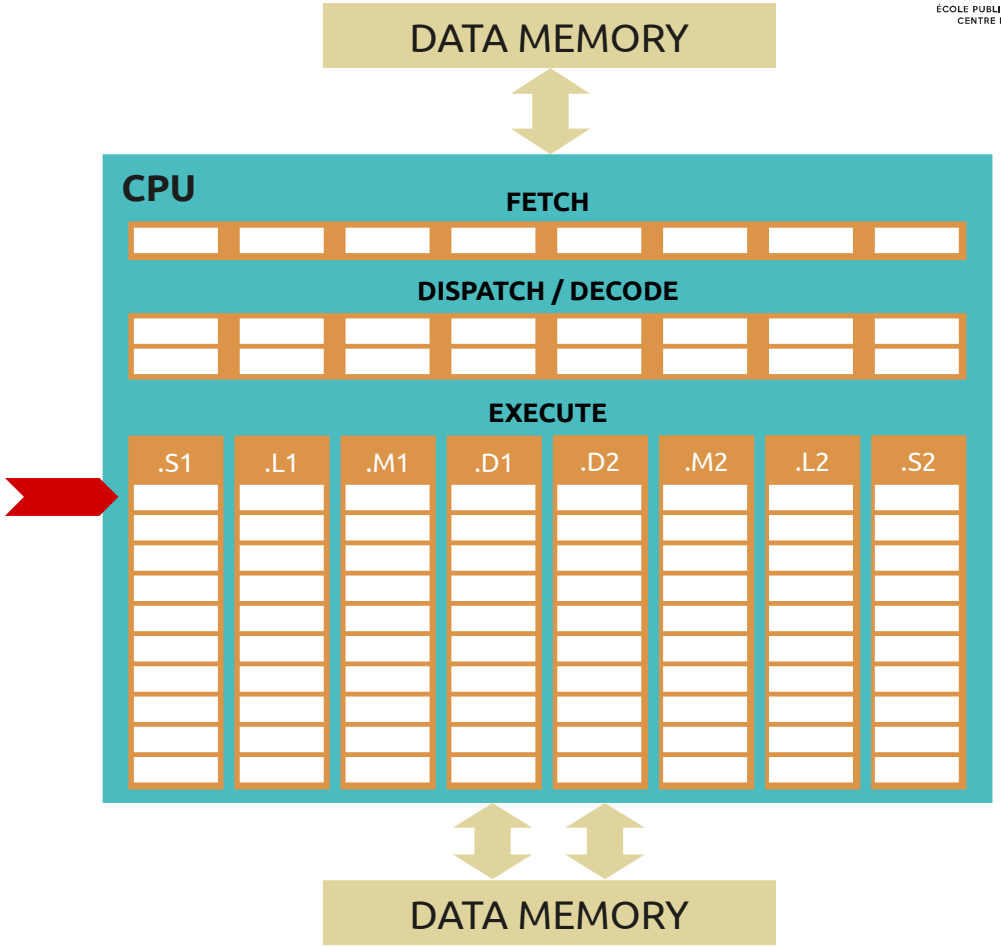
Optimized asm – 8 CPU cycles

```
    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
```

Exécution du programme

Optimized asm – 8 CPU cycles

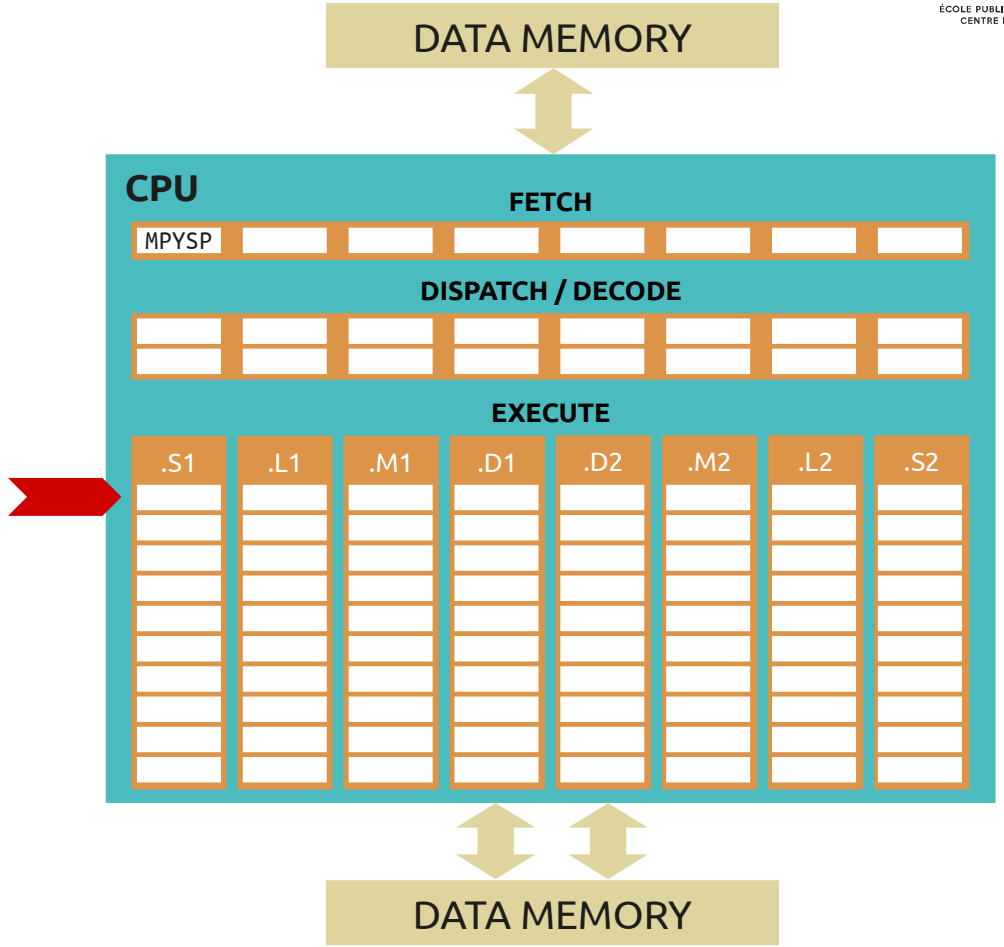
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

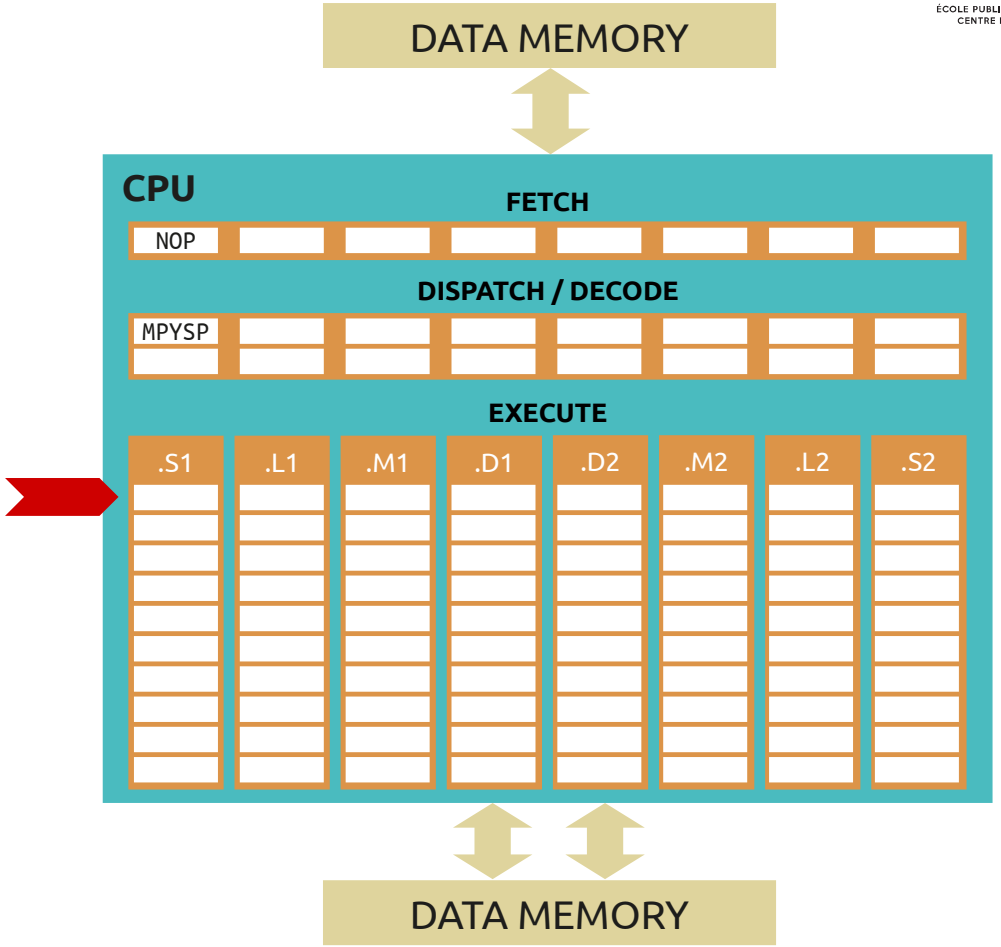
```
    ...
    MPYSP     .M1      A2, A3, A4
    NOP               2
    FADDSP    .S1      A0, A1, A0
    ADDSP     .S1      A2, A4, A2
    NOP               2
    MV        .D1      A0, A1
||  MV        .D2      B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

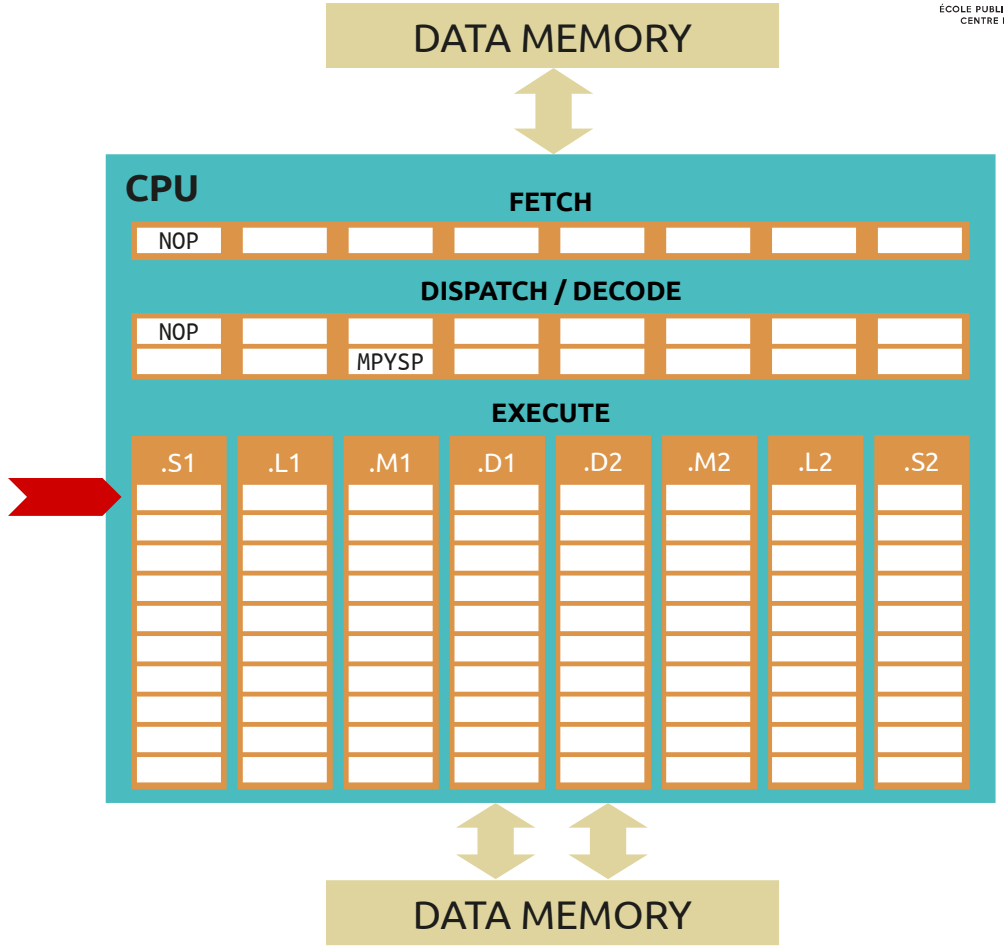
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

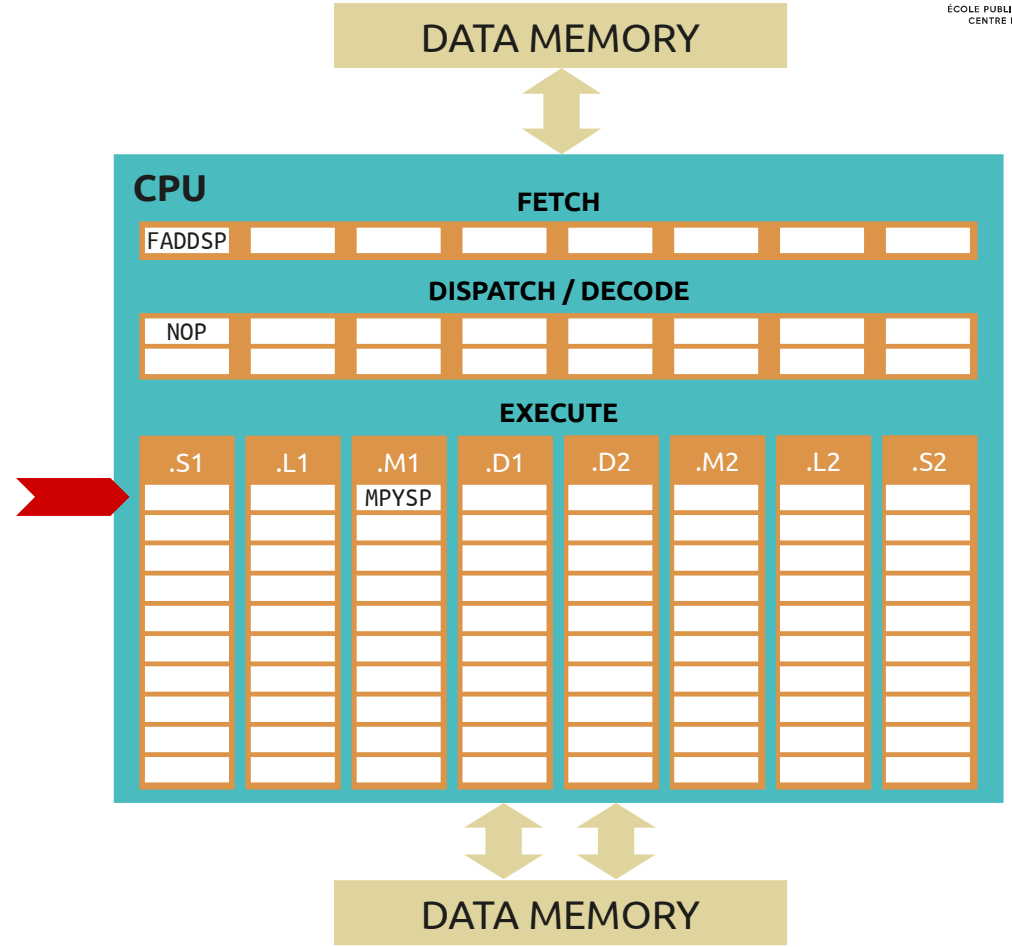
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

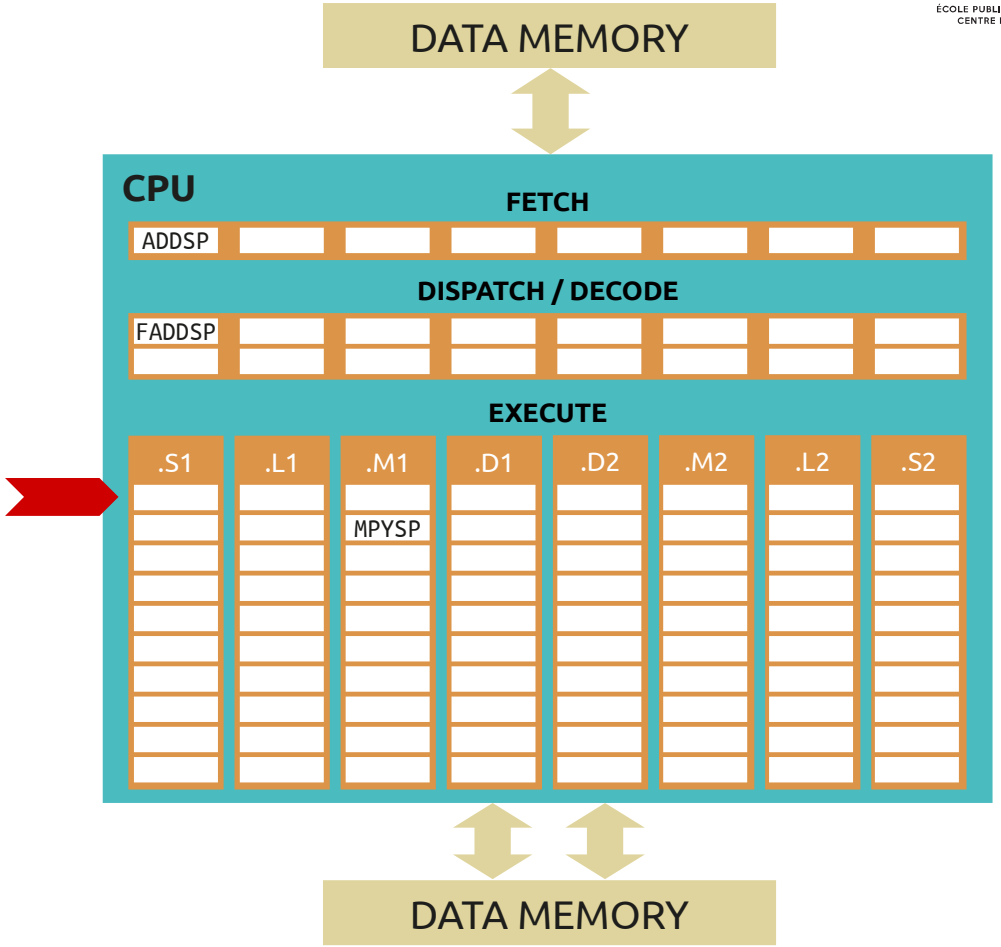
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

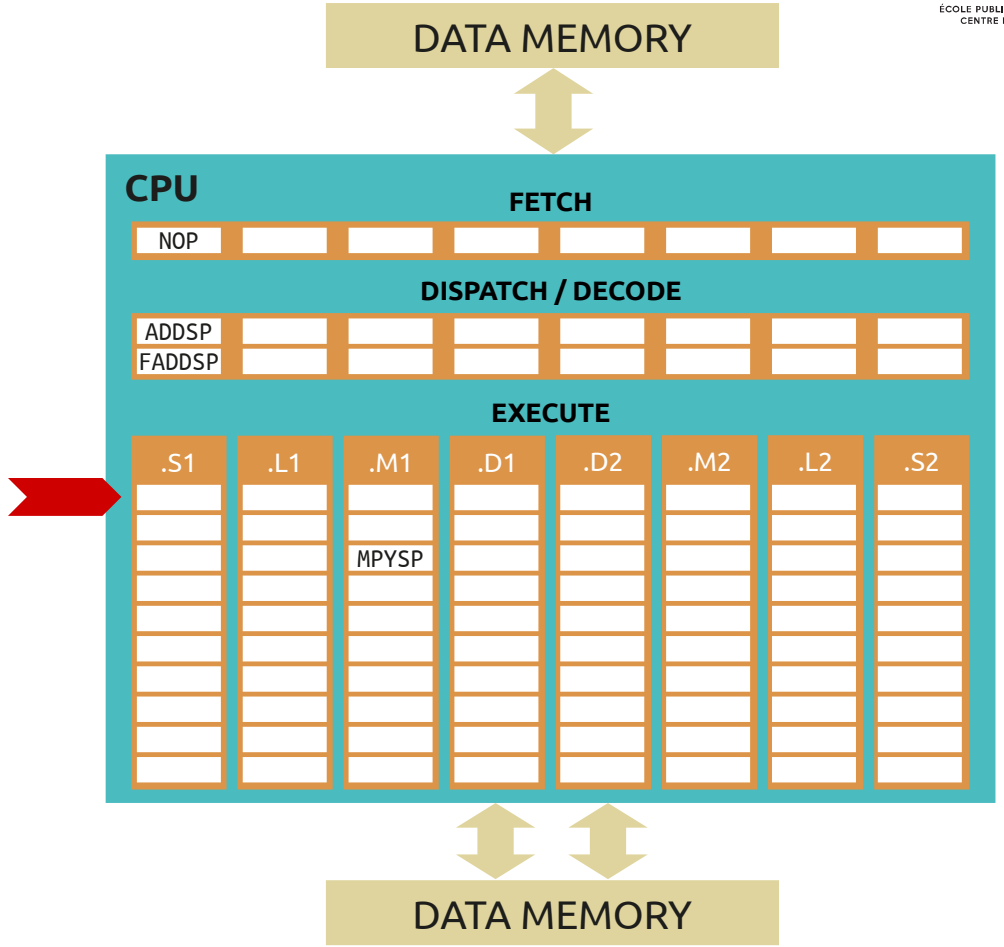
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

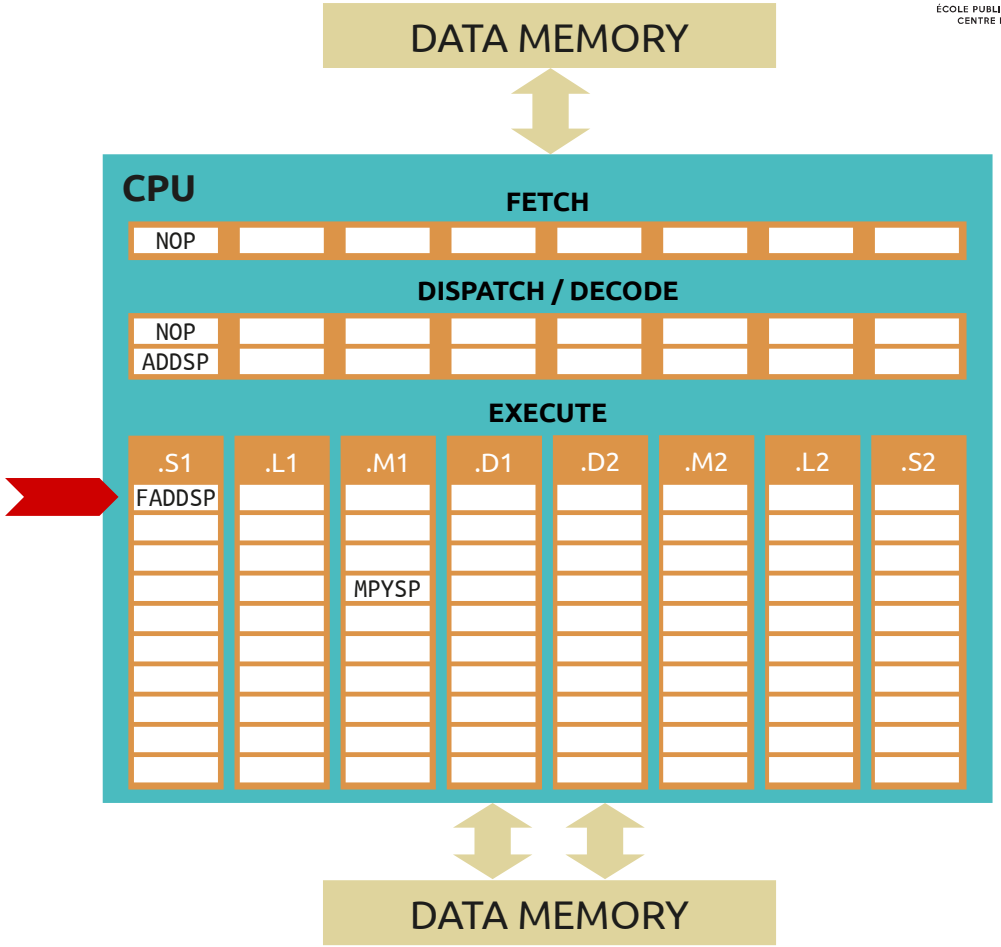
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

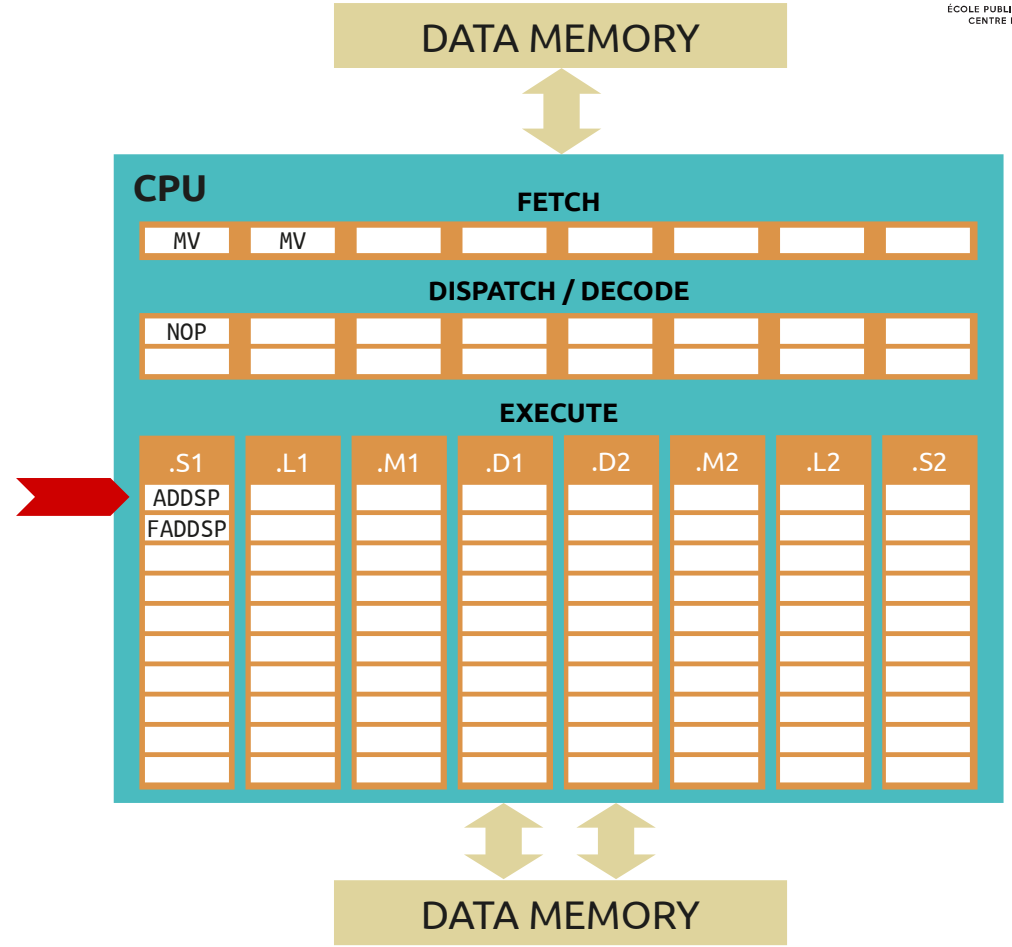
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

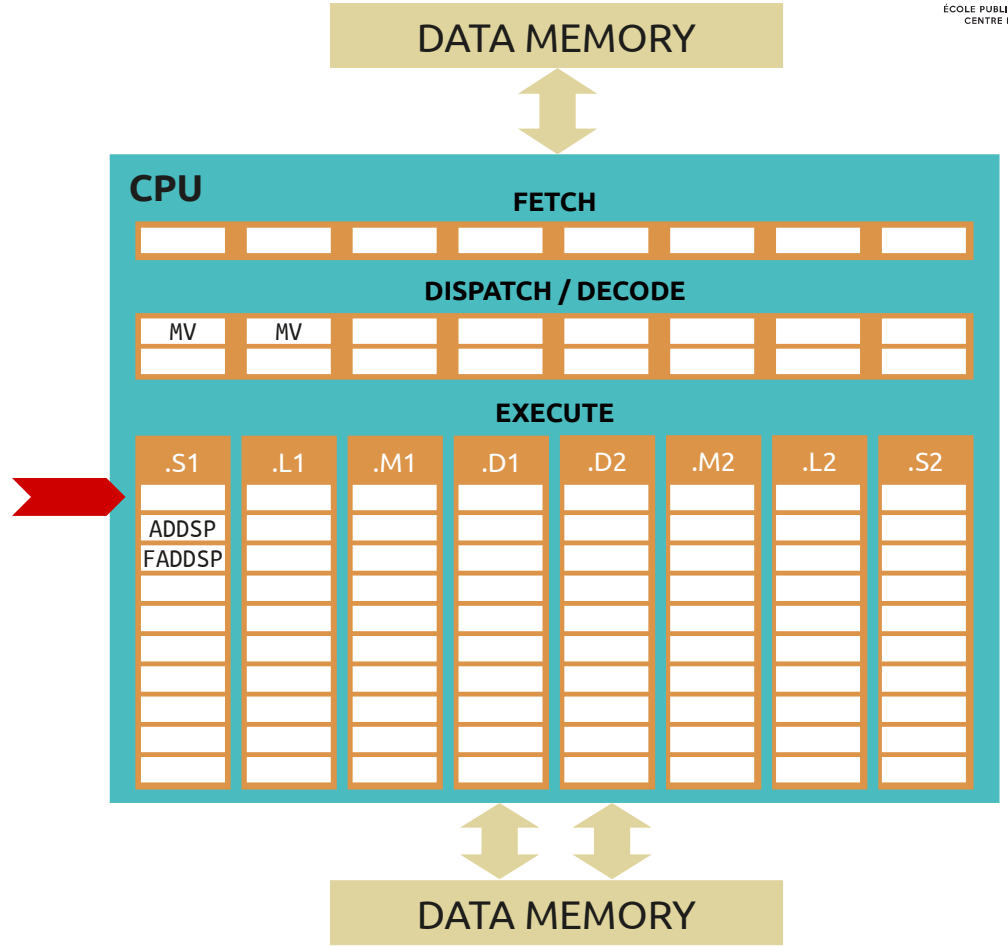
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

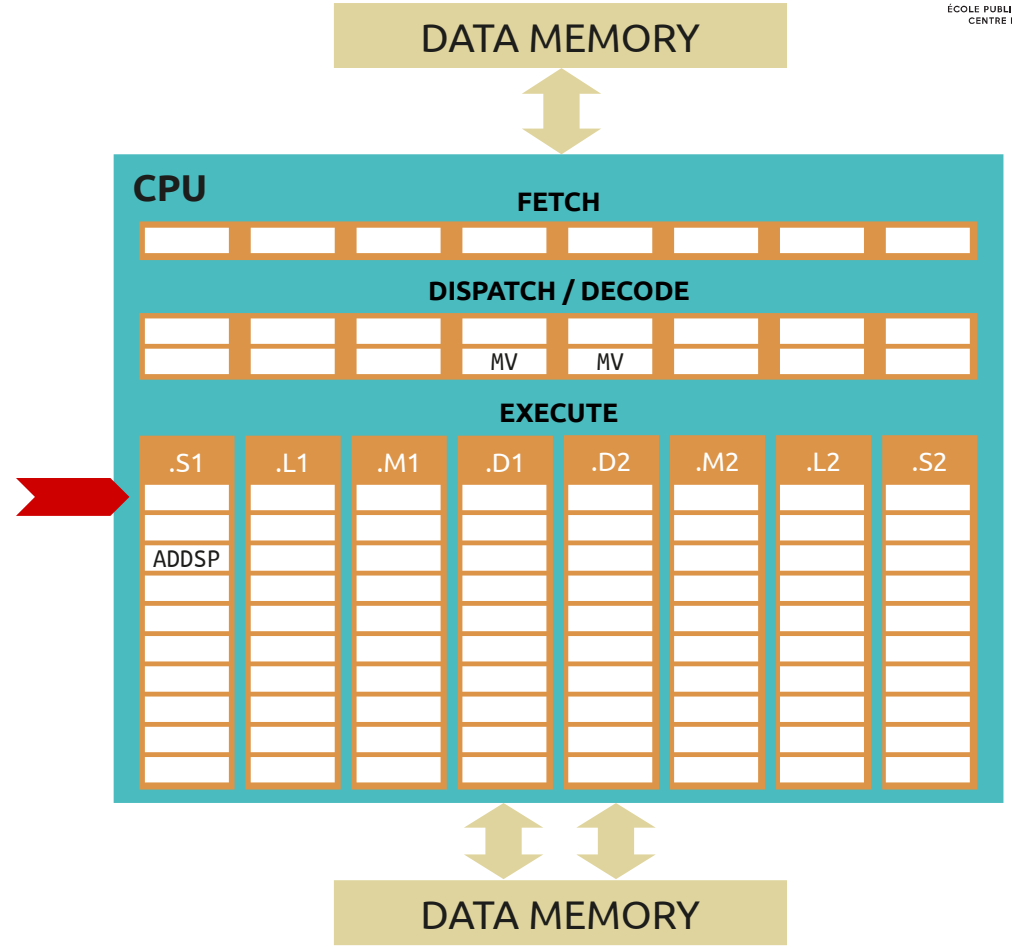
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

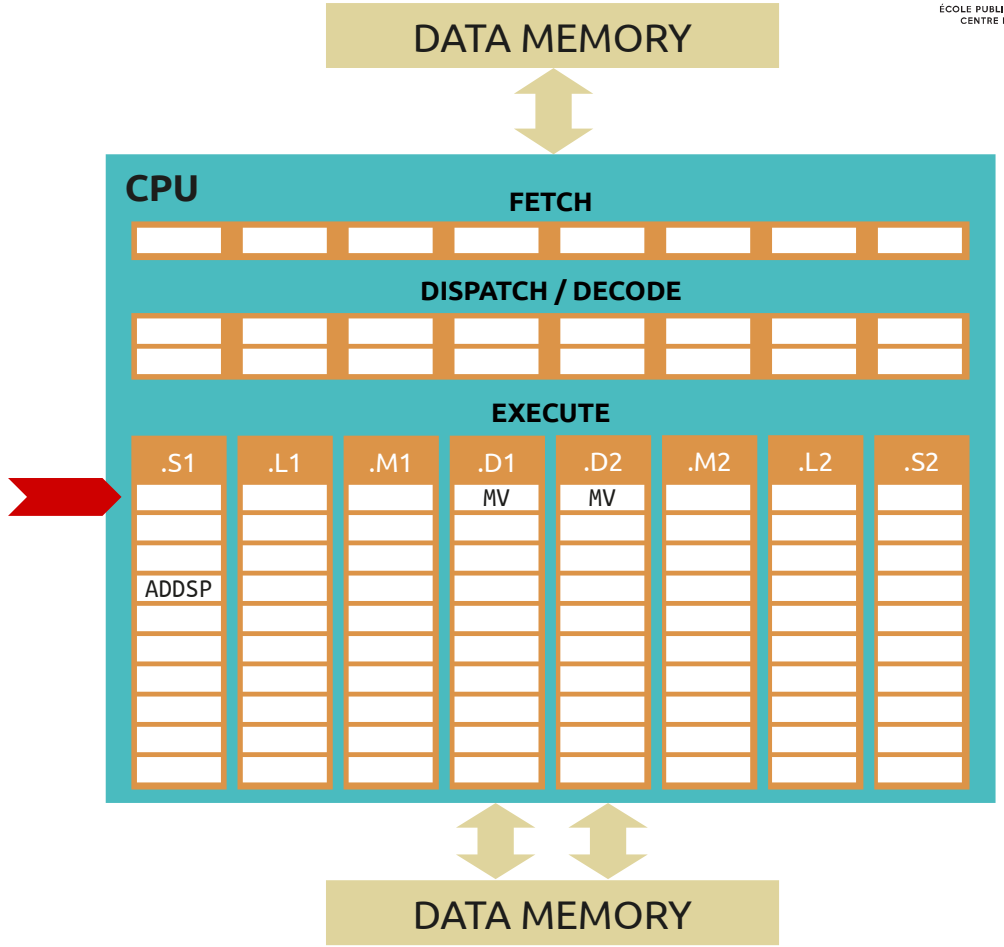
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

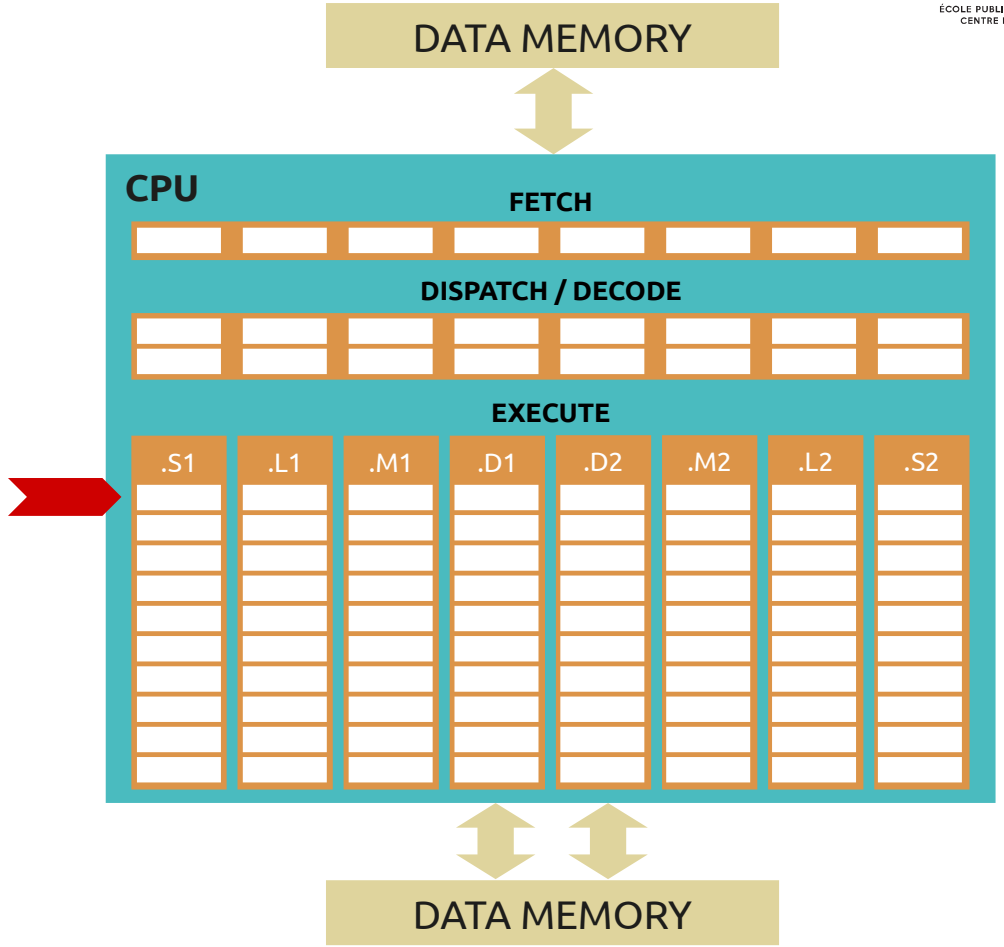
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```



Exécution du programme

Optimized asm – 8 CPU cycles

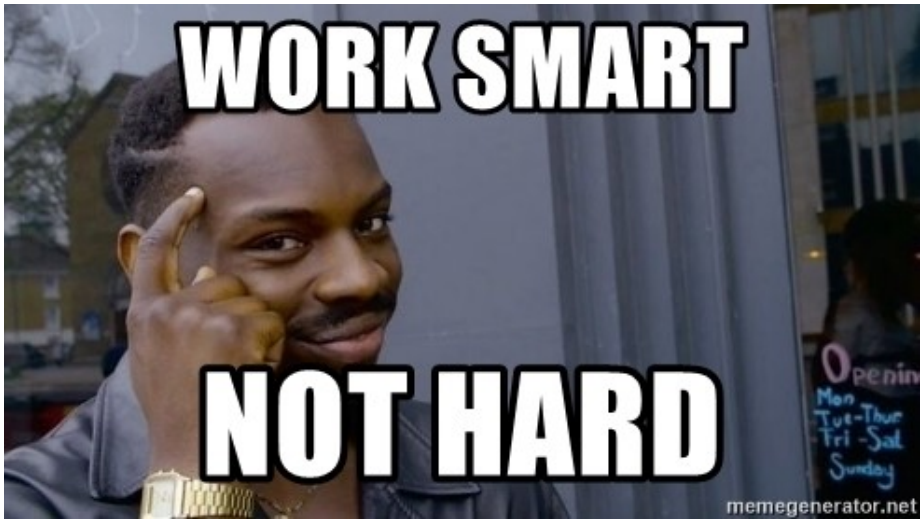
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```

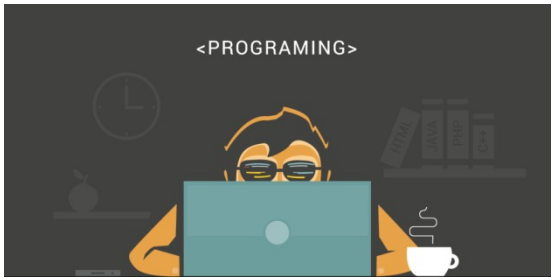


Une particularité des **processeurs VLIW** est que leurs instructions sont dans le désordre en mémoire programme, mais sortent du pipeline dans l'ordre !

Ce fonctionnement très basique donne à ces CPU un très bon ratio performances/Watt.

Mais les performances reposent entièrement sur l'intelligence et les compétences du développeur et/ou de la chaîne de compilation !





ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

Les **CPU superscalaires** sont conçus pour exécuter du code générique faiblement optimisé, remplis de tests et de sauts. → **Généricité**

Les **CPU VLIW** doivent exécutés du code spécifique à une cible afin d'exploiter le maximum de ses capacités. → **Code spécifique** (non portable)

```
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

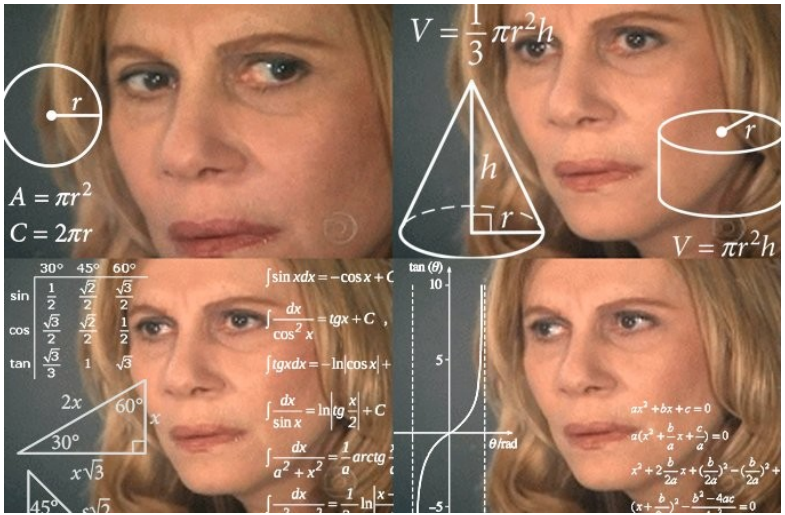
```
x_1o_x_0o = _daddsp(xh1_0_xh0_0, xh21_0_xh20_0);
x_3o_x_2o = _daddsp(xh1_1_xh0_1, xh21_1_xh20_1);

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

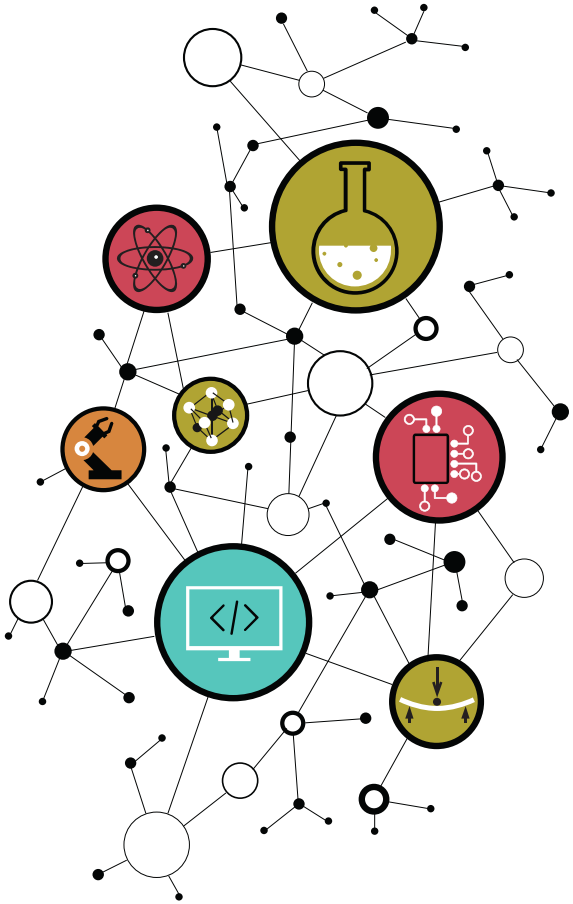
Exploiter le plein potentiel d'un processeur ne peut se faire qu'à travers la **maîtrise de l'architecture matérielle** ainsi que des outils de développements associés (*toolchain*).

Il est essentiel de maîtriser les mathématiques derrière les algorithmes afin de pouvoir les ré-écrire (et ré-implémenter) dans l'optique d'une accélération du code.

Notons que, généralement, les codes les plus performants ne sont pas portables.



CONTACT



Dimitri Boudier – PRAG ENSICAEN

dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)