Chapter 3

TI C6678's Architecture

2021-2022

TMS320C6678 PROCESSOR

Processor and Core specifications

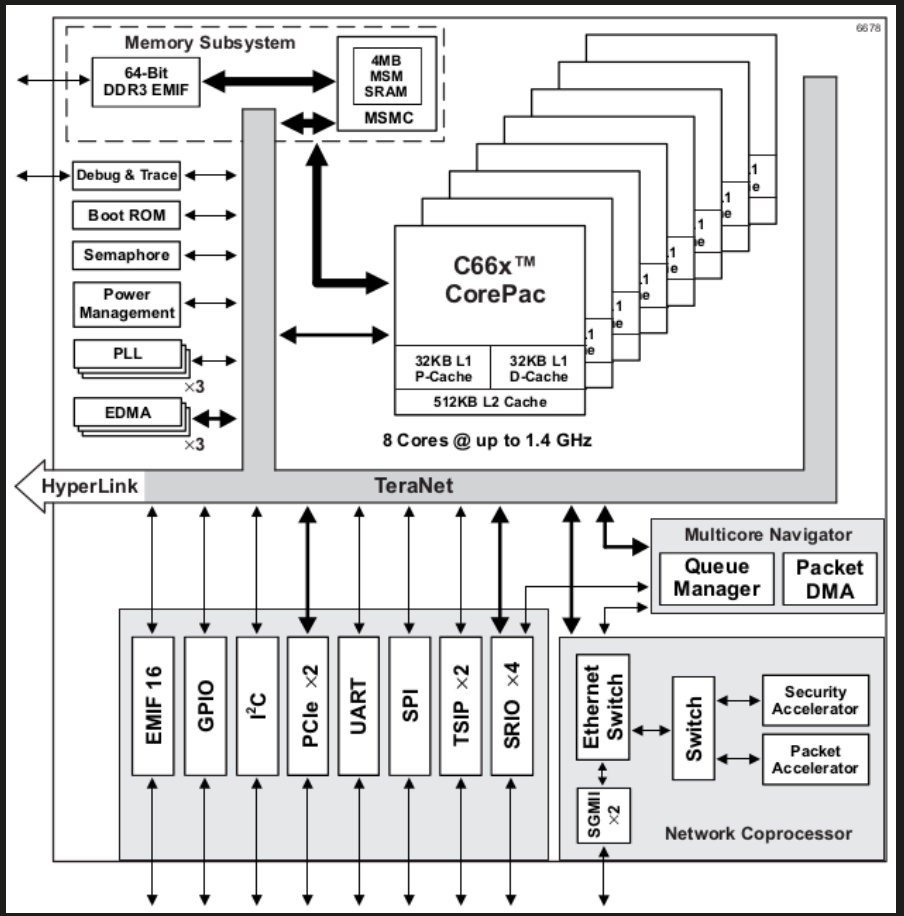
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

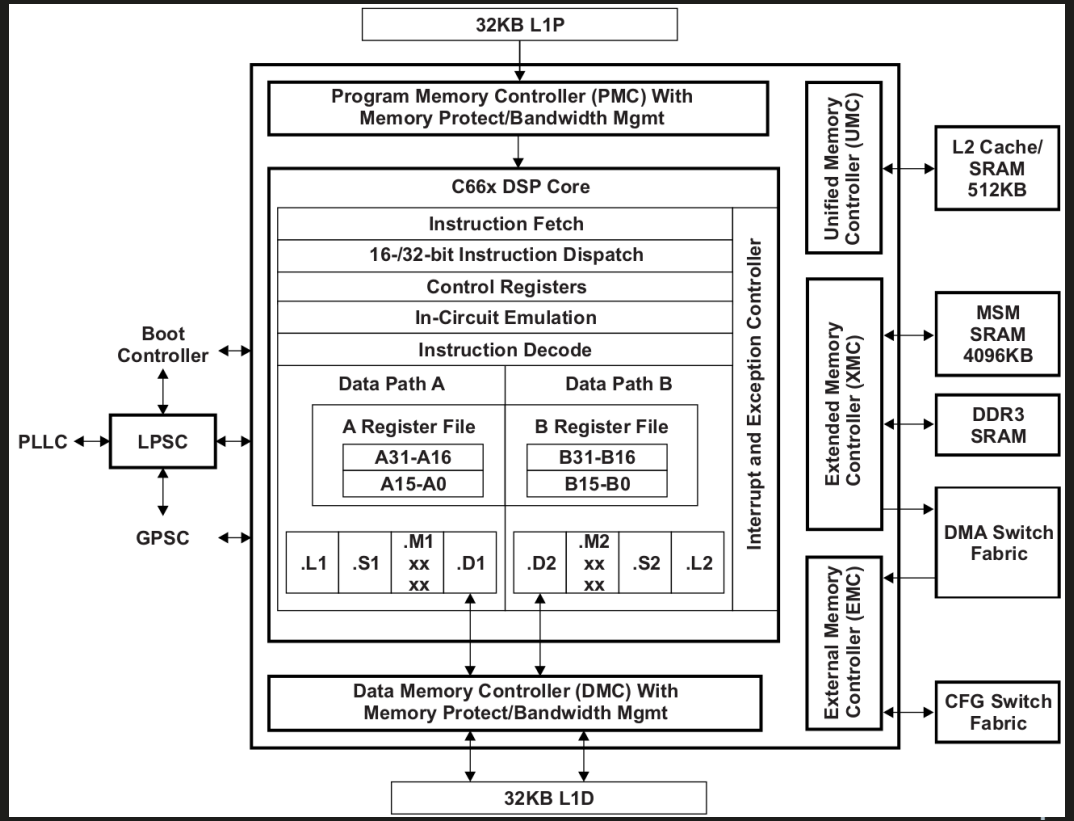


TMS320C6678 functional block diagram

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

The C66x CorePac consists of several components:

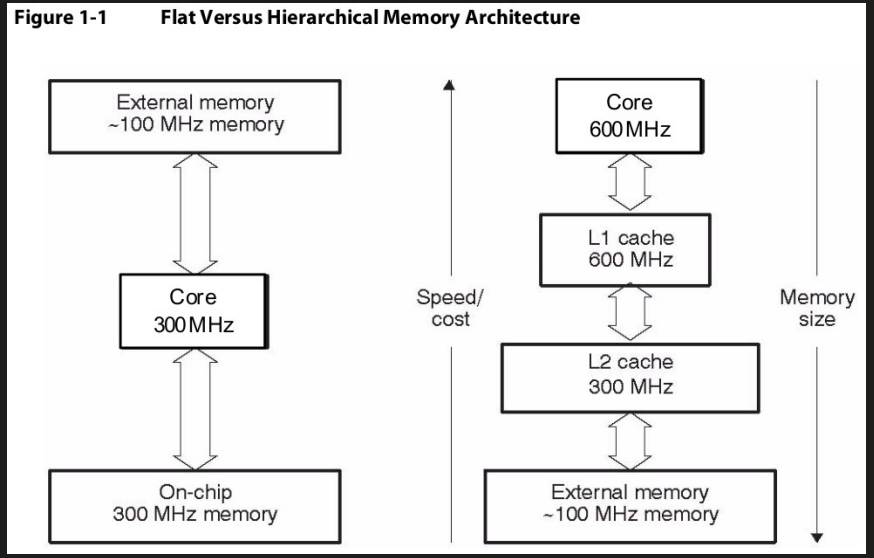
- The C66x DSP and associated C66x CorePac core
- Level-one and level-two memories (L1P, L1D, L2)
- Data Trace Formatter (DTF)
- Embedded Trace Buffer (ETB)
- Interrupt Controller
- Power-down controller
- External Memory Controller
- Extended Memory Controller
- A dedicated power/sleep controller (LPSC)



TMS320C66x CorePac DSP Block Diagram

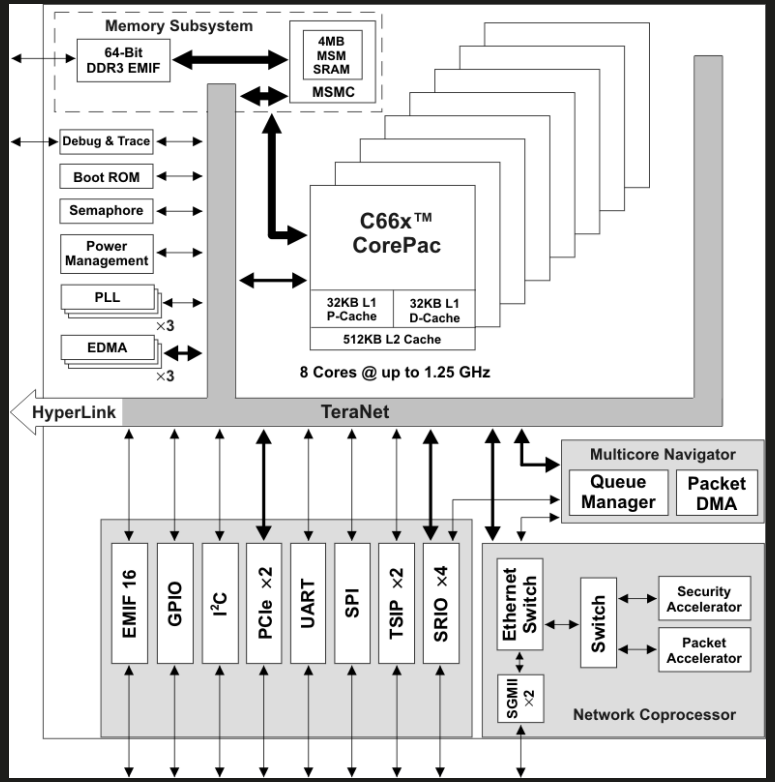
Each core has its own cache memories :

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



"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

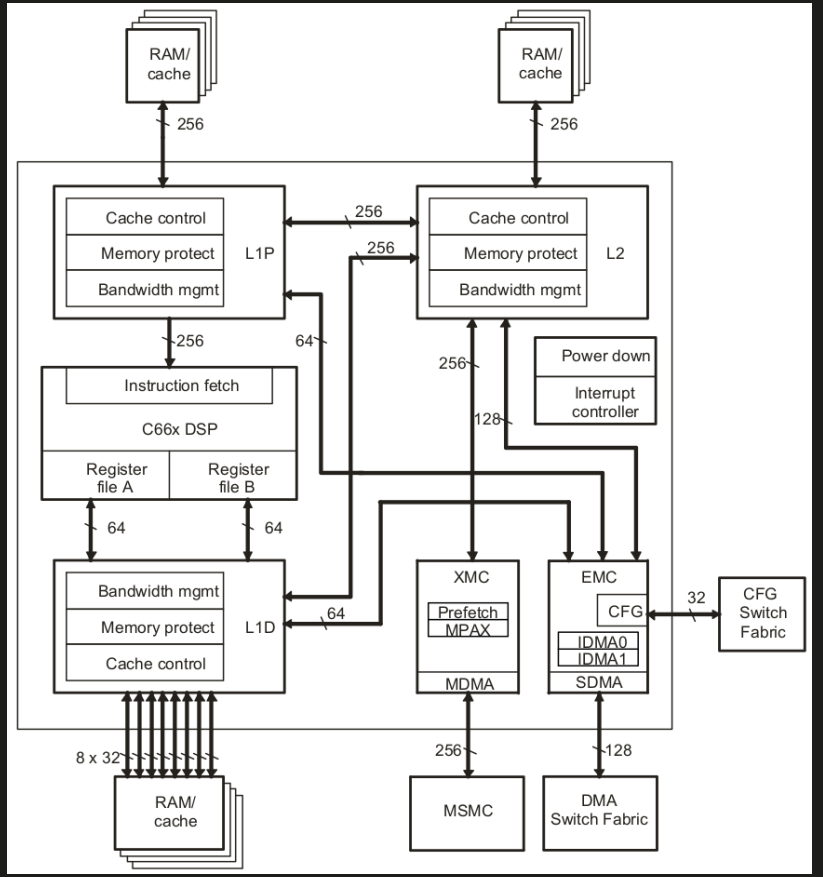
Core Architecture

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.

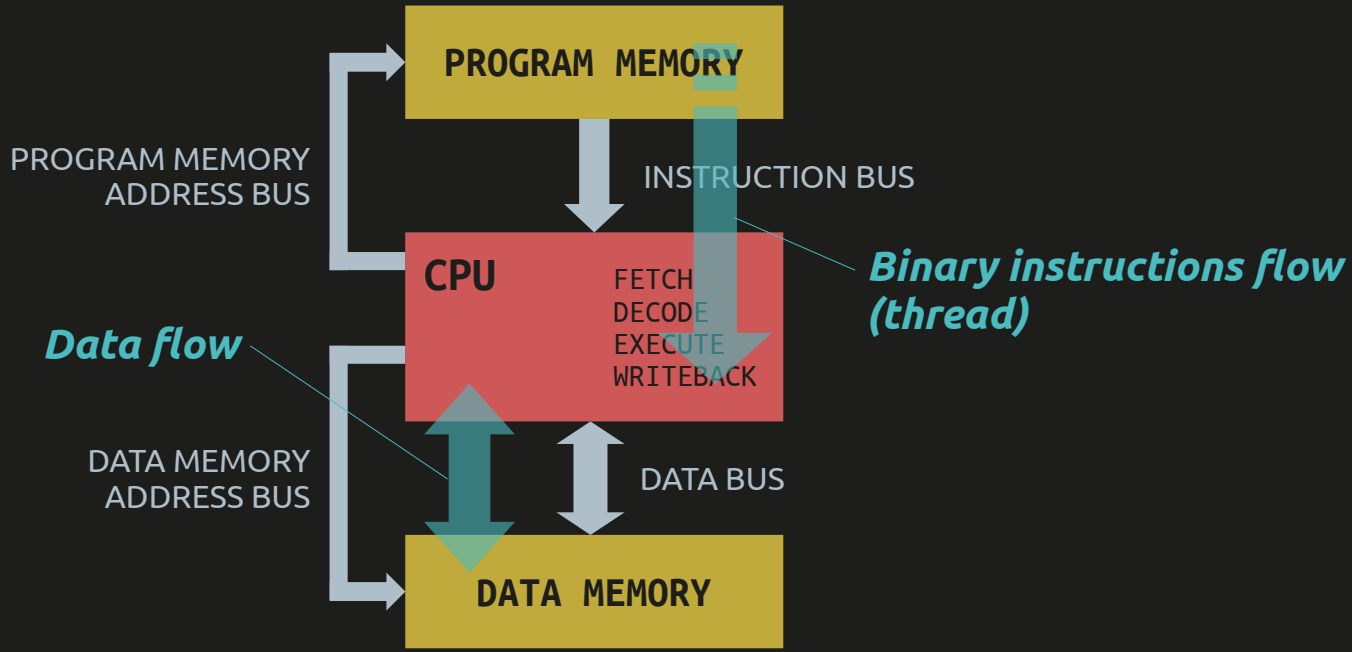


C6600 HARDWARE PIPELINE

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

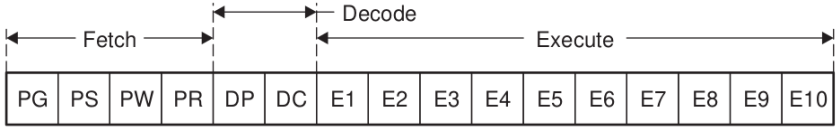
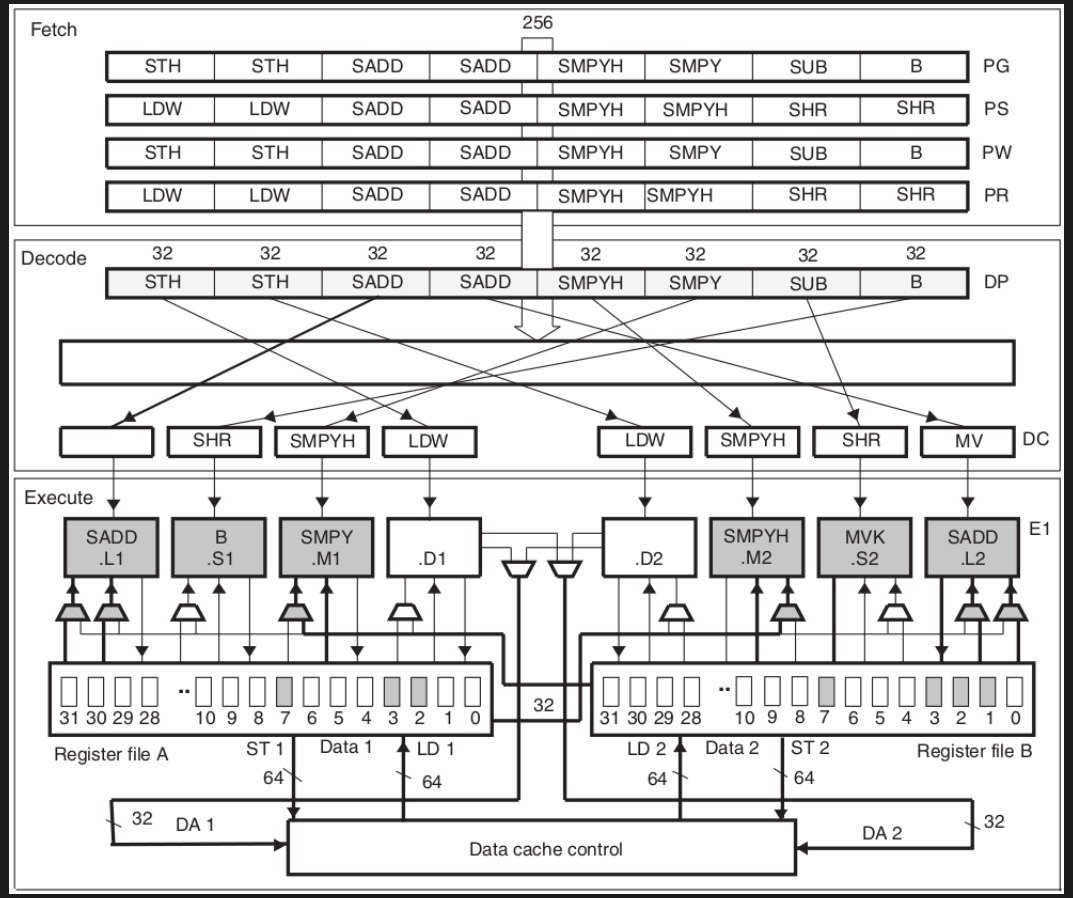


Figure 5-6 Pipeline Operation: One Execute Packet per Fetch Packet

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.



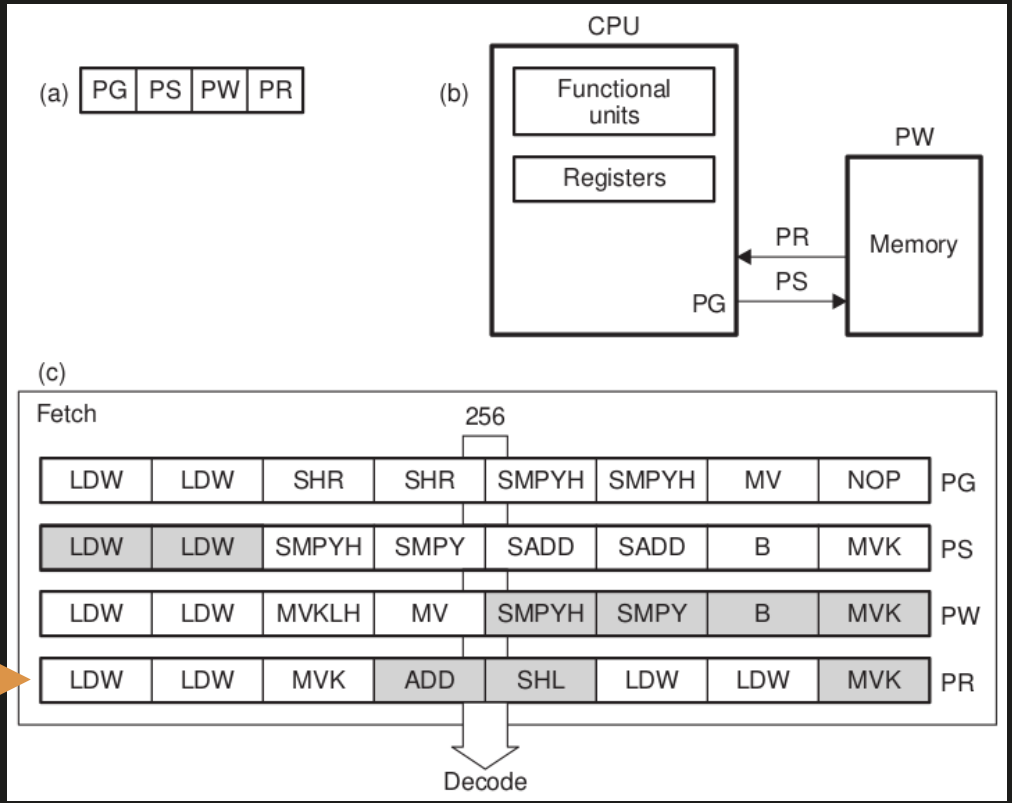
Pipeline Phases Block Diagram

FETCH stage

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)

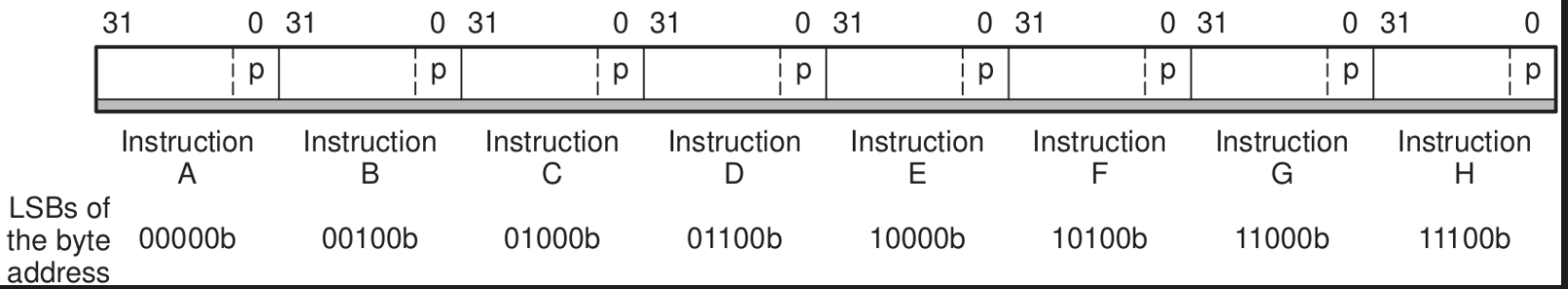


Each instruction has a 32-bit fixed size (RISC-like instruction set).

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



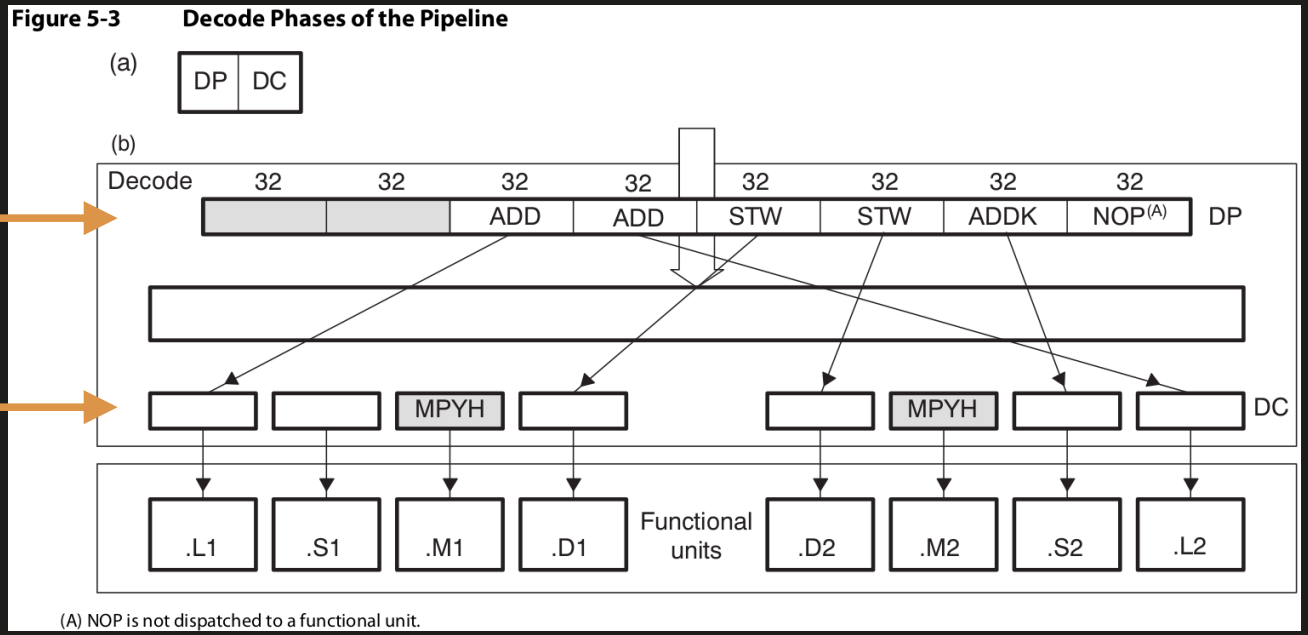
DECODE stage

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.

Dispatch

Decode

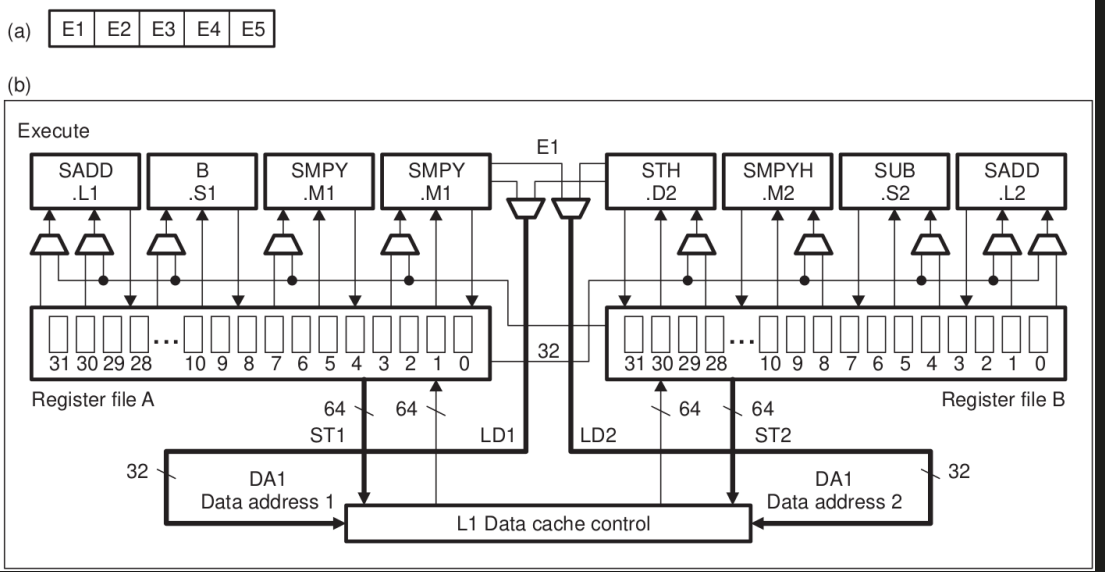


The VLIW execution stage has 8 SIMD Execution Units (or Functional Units).

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



C6600 HARDWARE PIPELINE

EXECUTION stage

Each EU has its own VLIW pipeline.

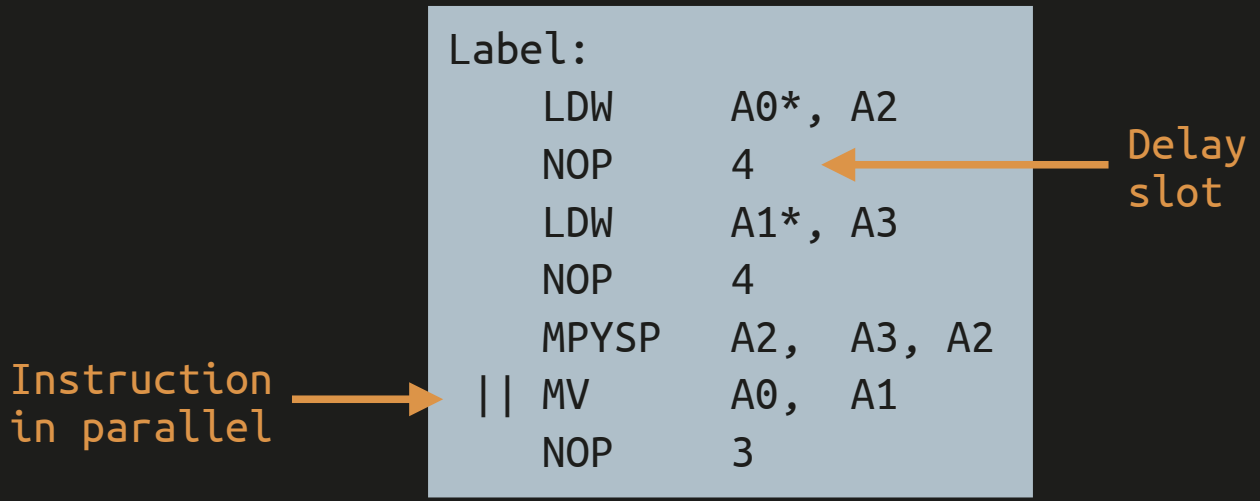
Table 5-1 Operations Occurring During Pipeline Phases (Part 2 of 2)

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

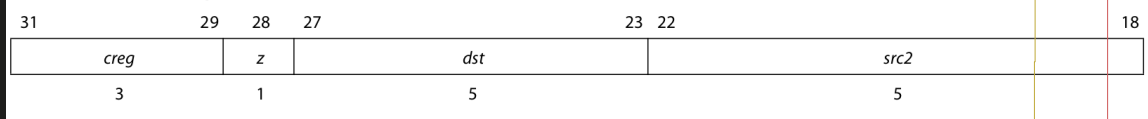
Syntax **MPYSP** (.unit) *src1, src2, dst*

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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

PROGRAMMING A VLIW CPU

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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
```

Rewriting code

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
```

5 CPU cycles

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

1 CPU cycle

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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

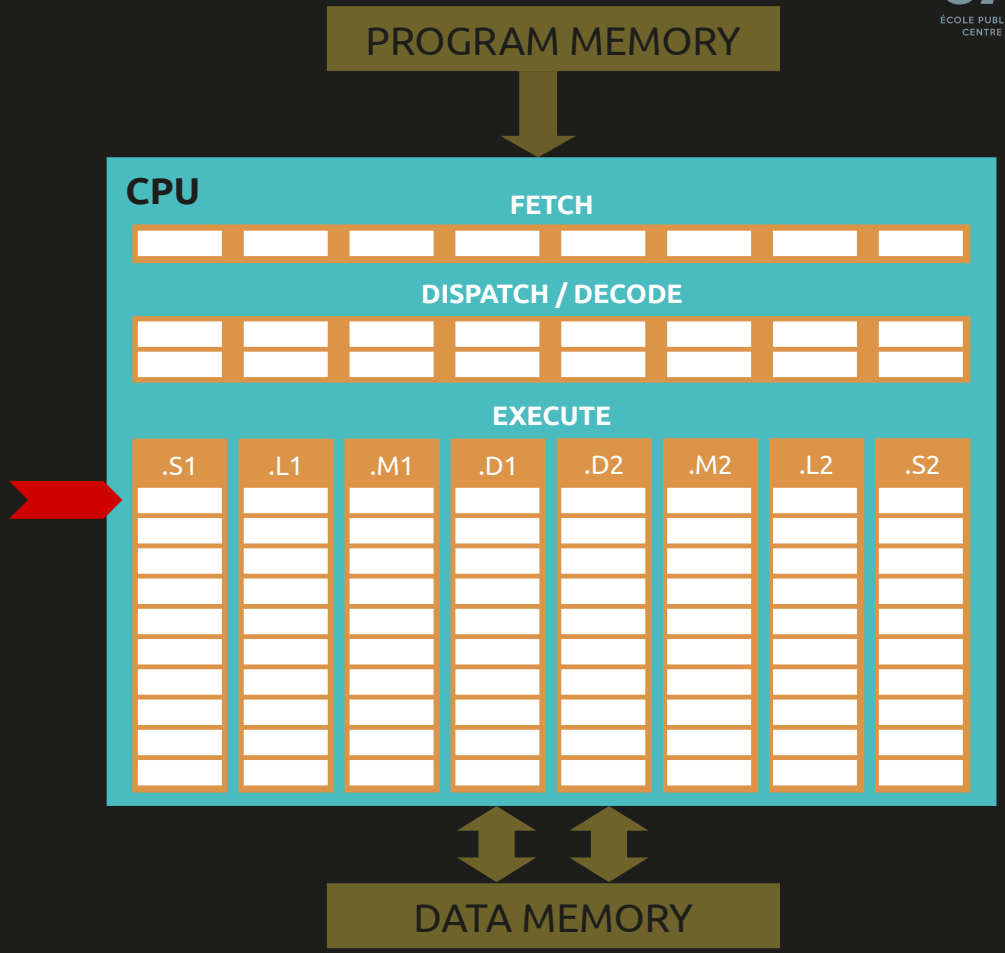
```
   MPYSP   .M1     A2, A3, A4
   NOP             2
   FADDSP  .S1     A0, A1, A0
   ADDSP   .S1     A2, A4, A2
   NOP             2
   MV      .D1     A0, A1
|| MV      .D2     B9, B7
```

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
    ...
```

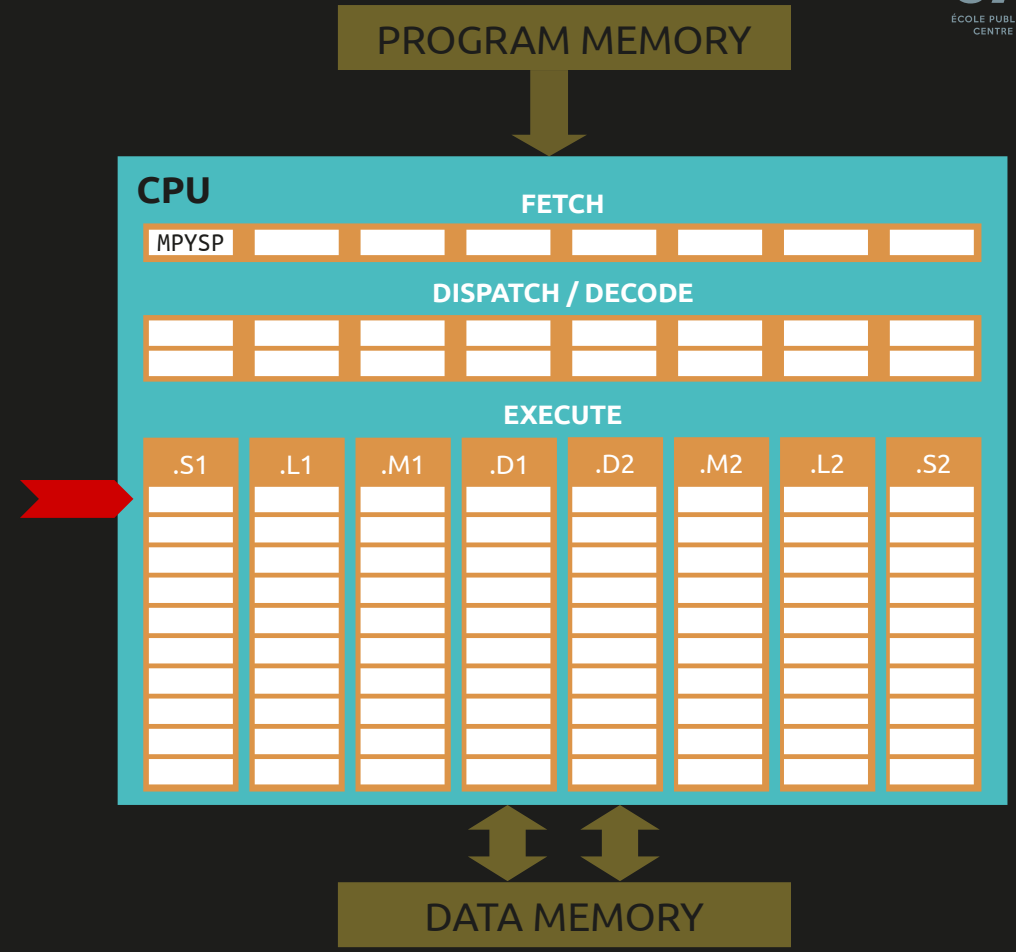


PROGRAMMING A VLIW CPU

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
    ...
```

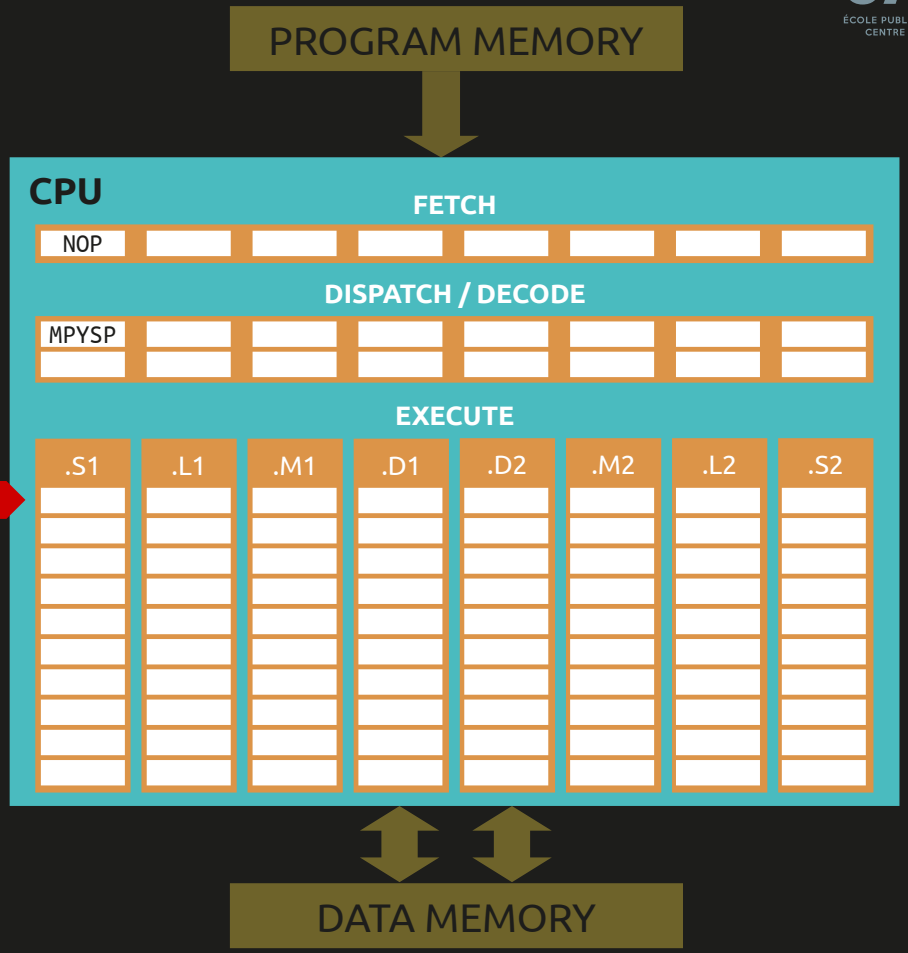


PROGRAMMING A VLIW CPU

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
    ...
```

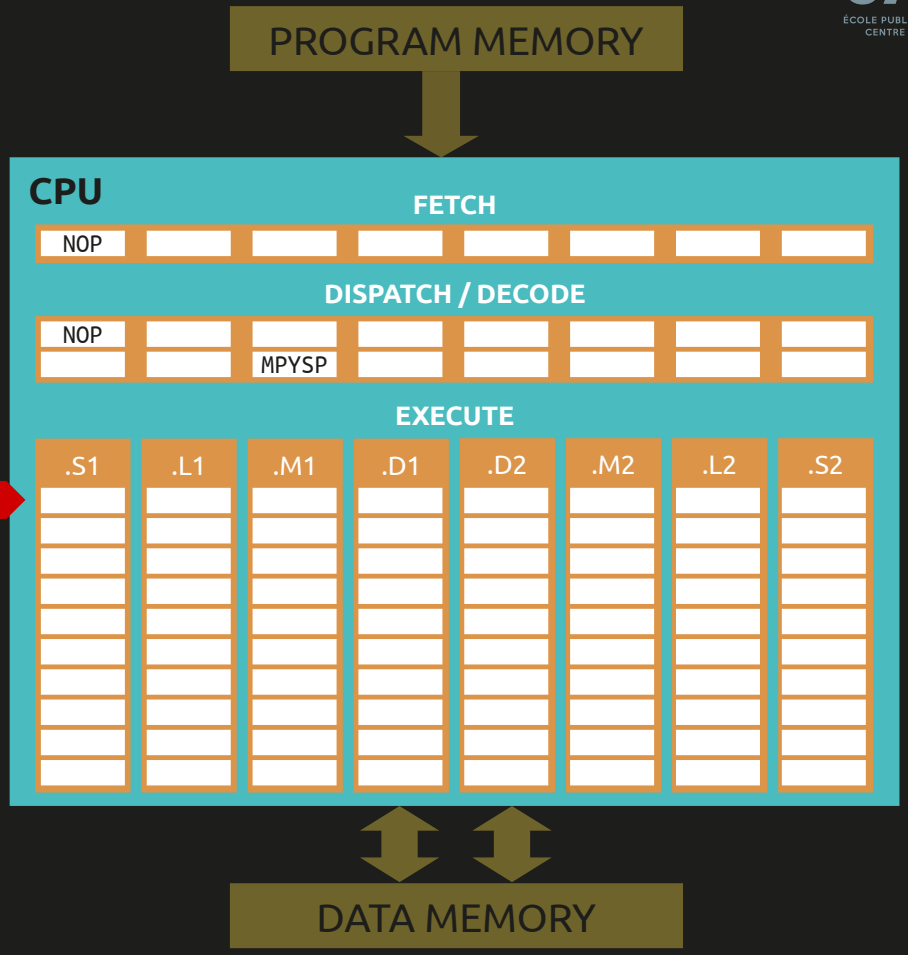


PROGRAMMING A VLIW CPU

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
    ...
```

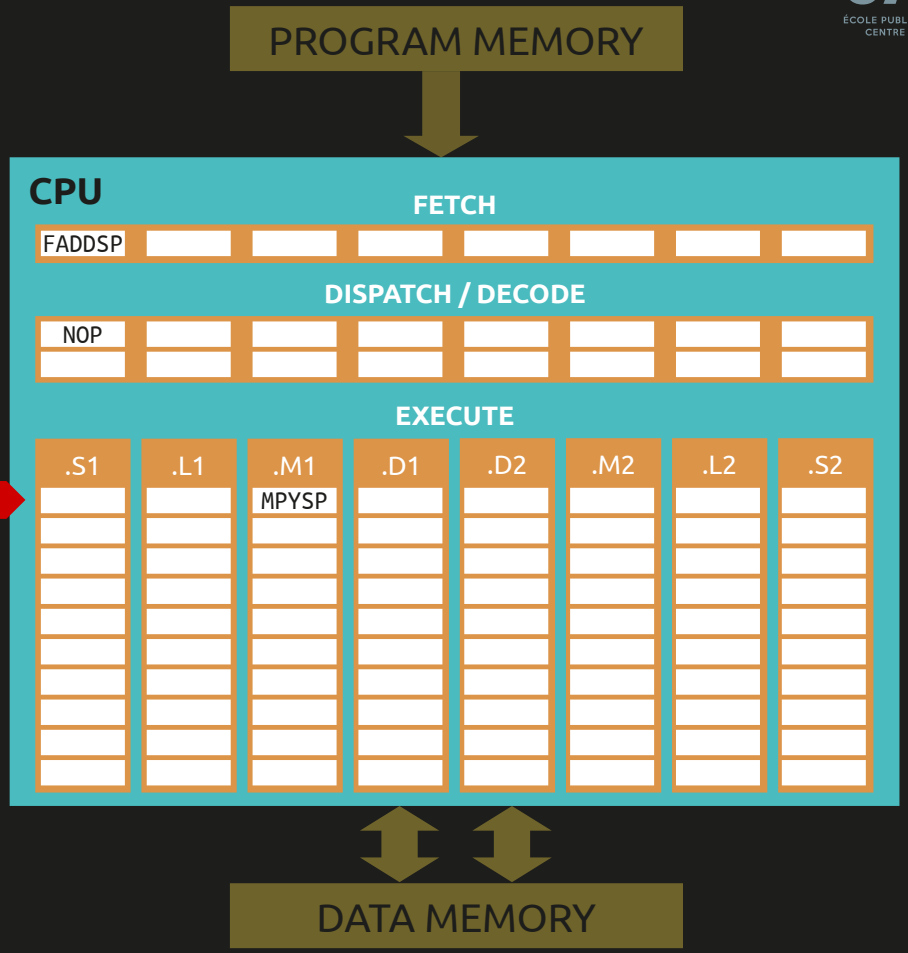


PROGRAMMING A VLIW CPU

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
    ...
```

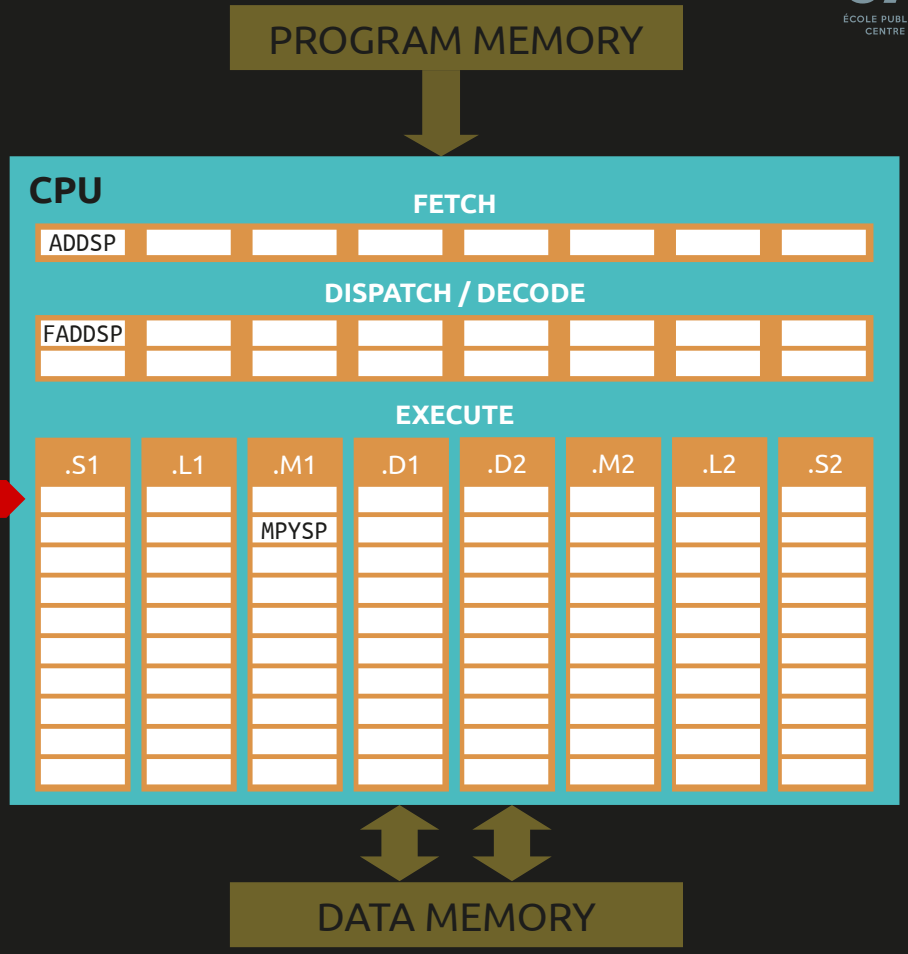


PROGRAMMING A VLIW CPU

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
    ...
```

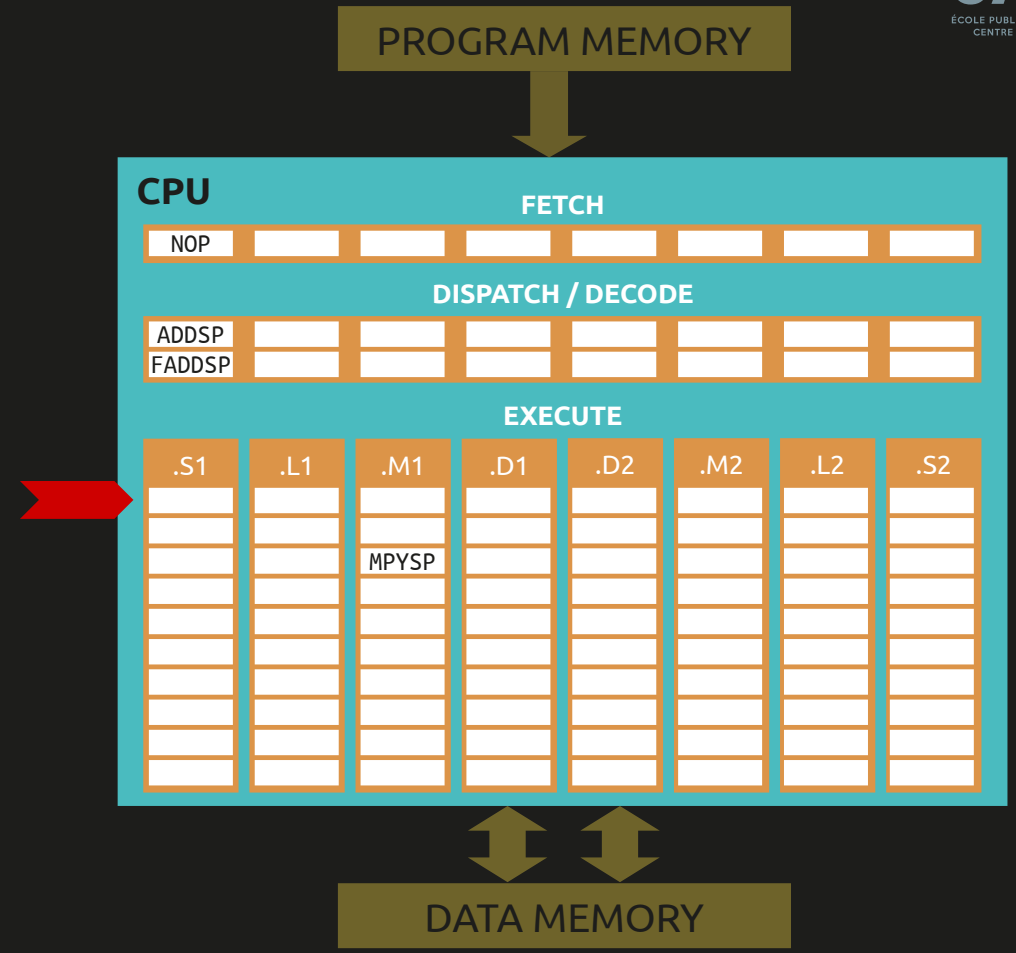


PROGRAMMING A VLIW CPU

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
    ...
```

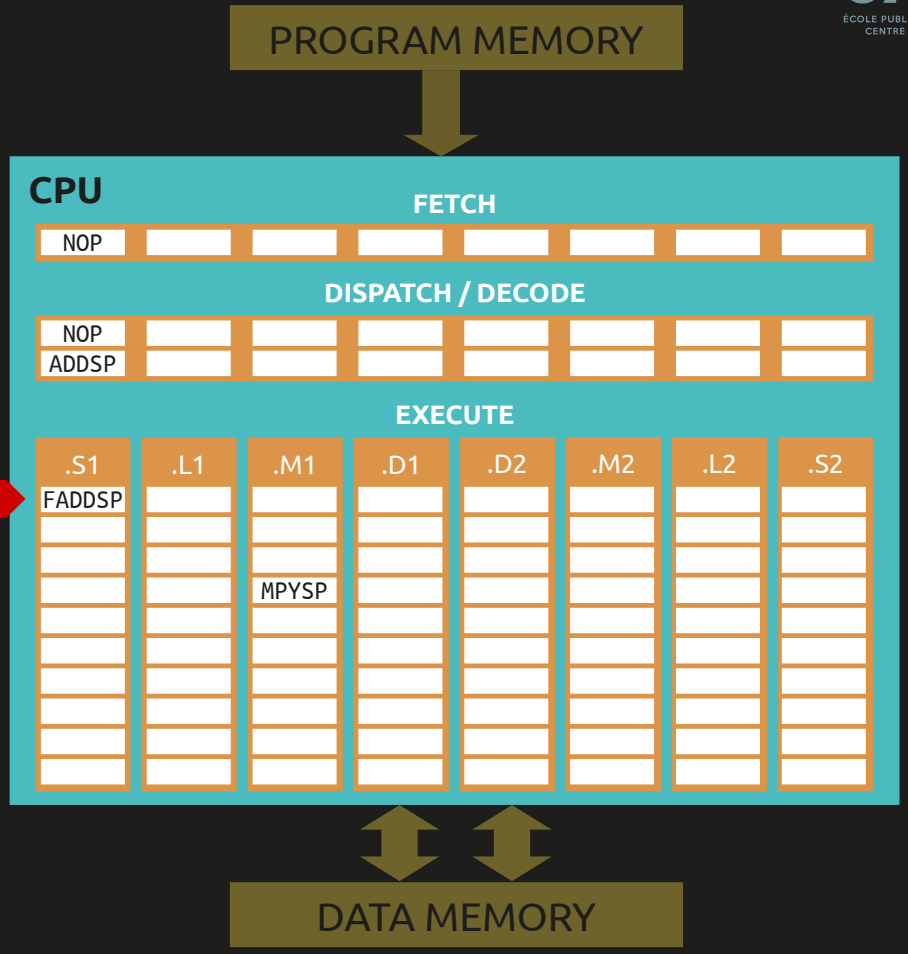


PROGRAMMING A VLIW CPU

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
    ...
```

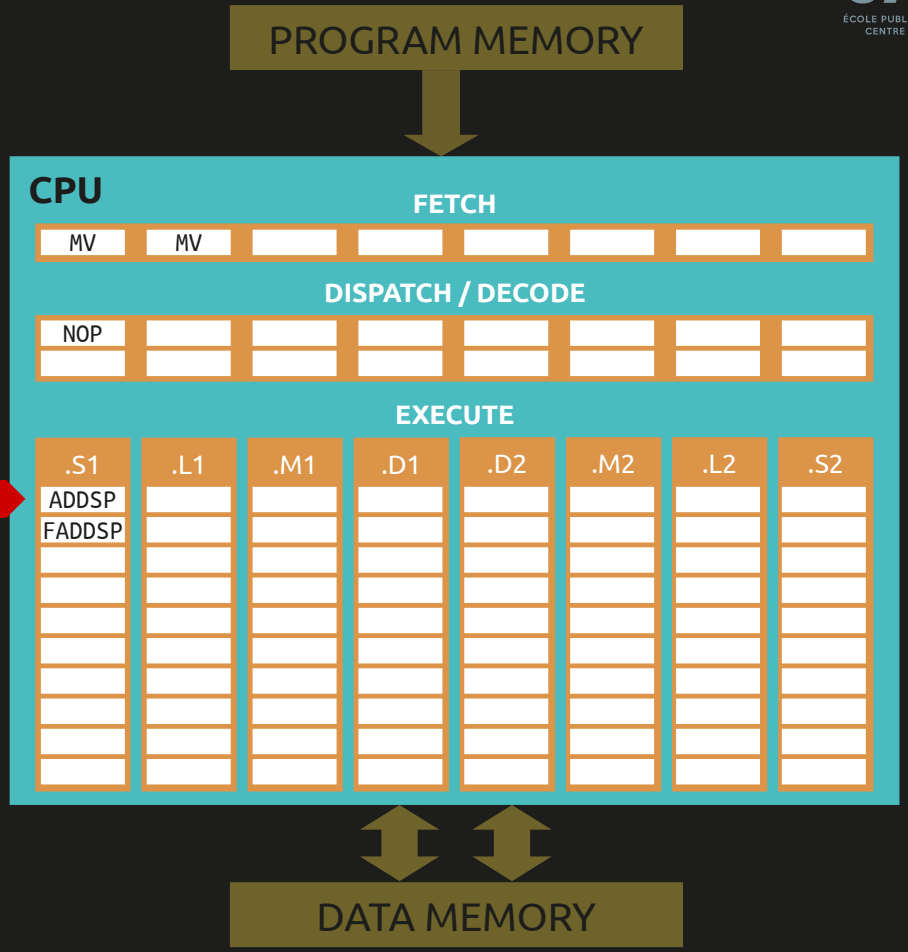


PROGRAMMING A VLIW CPU

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
    ...
```

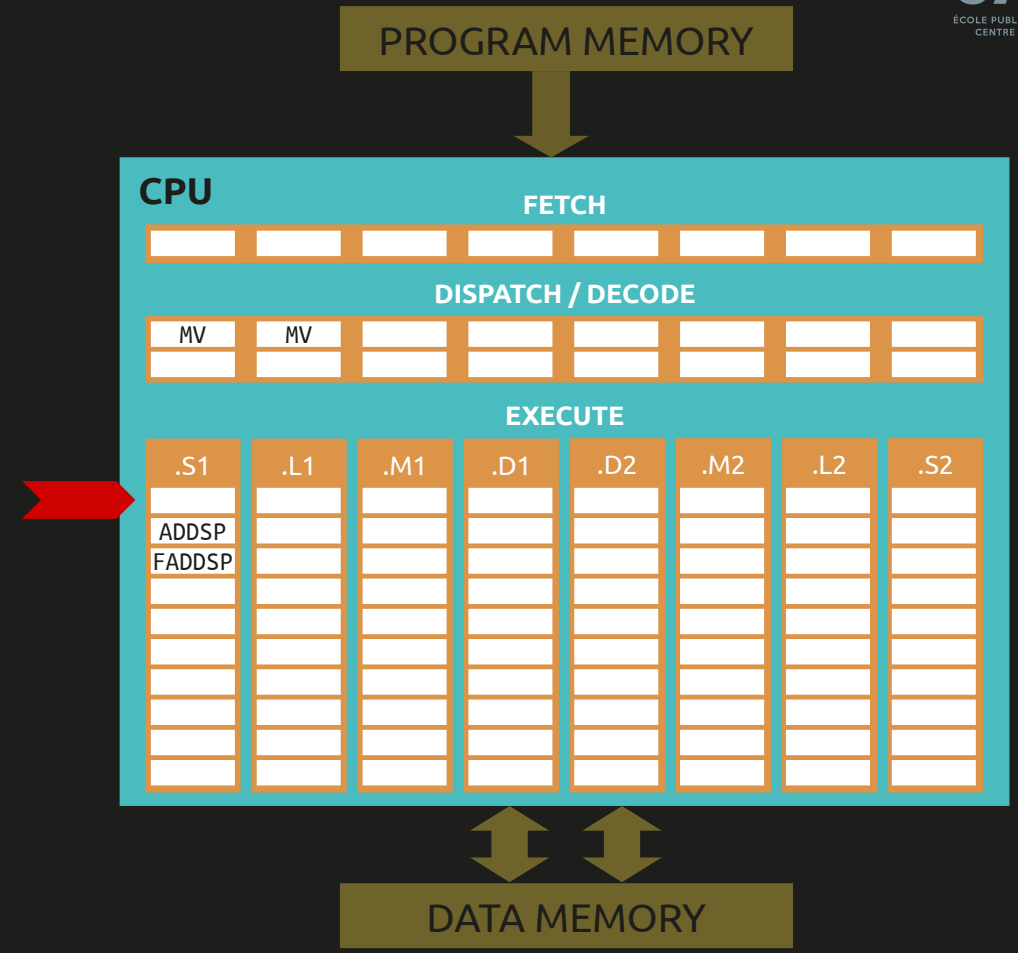


PROGRAMMING A VLIW CPU

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
    ...
```

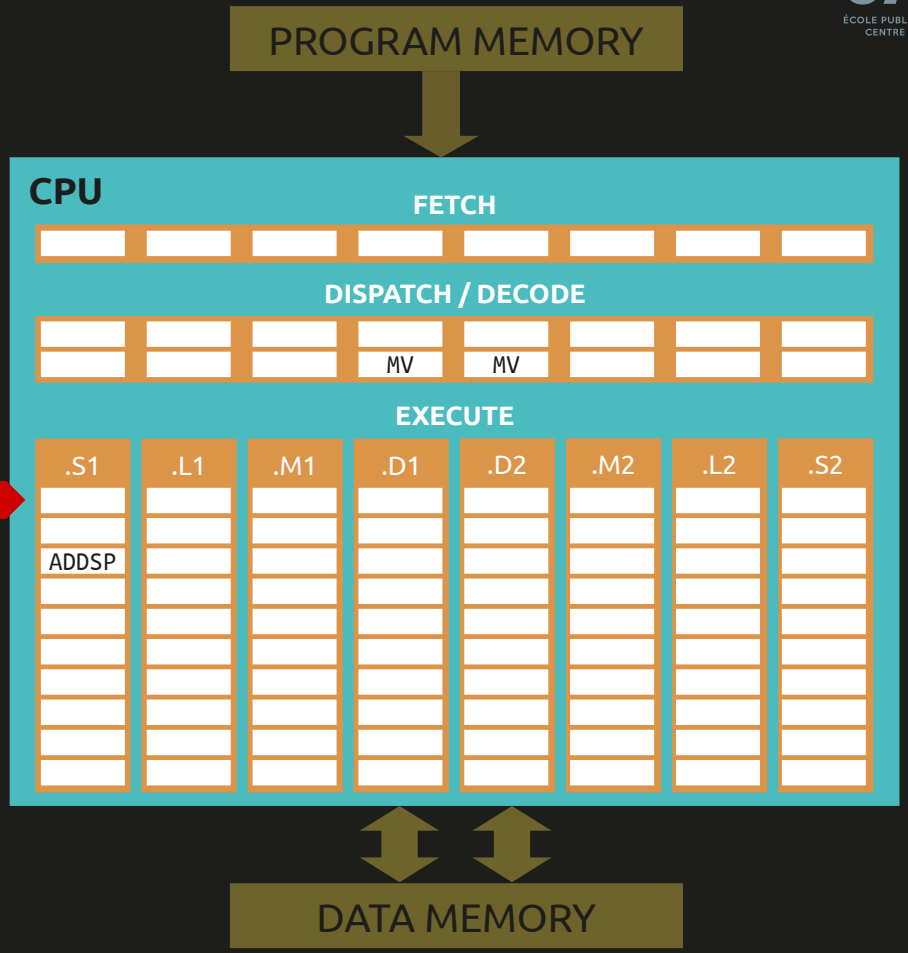


PROGRAMMING A VLIW CPU

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
    ...
```

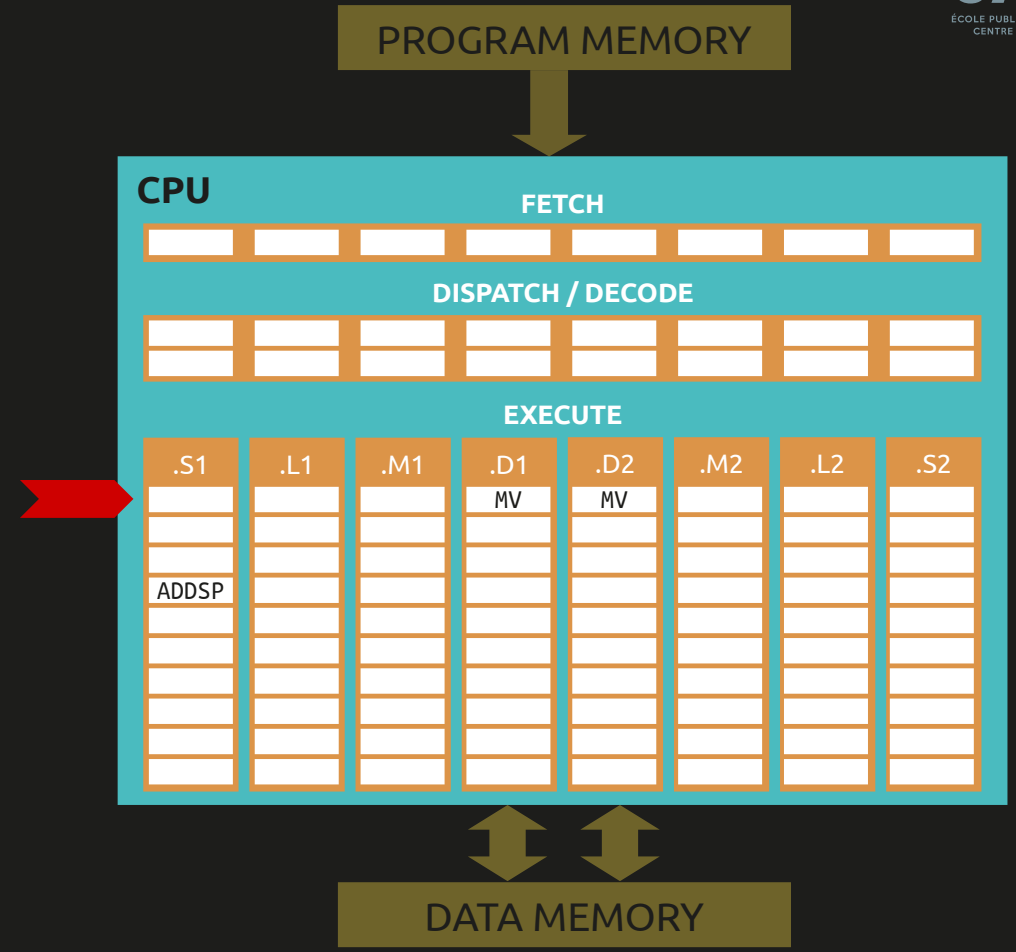


PROGRAMMING A VLIW CPU

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
    ...
```

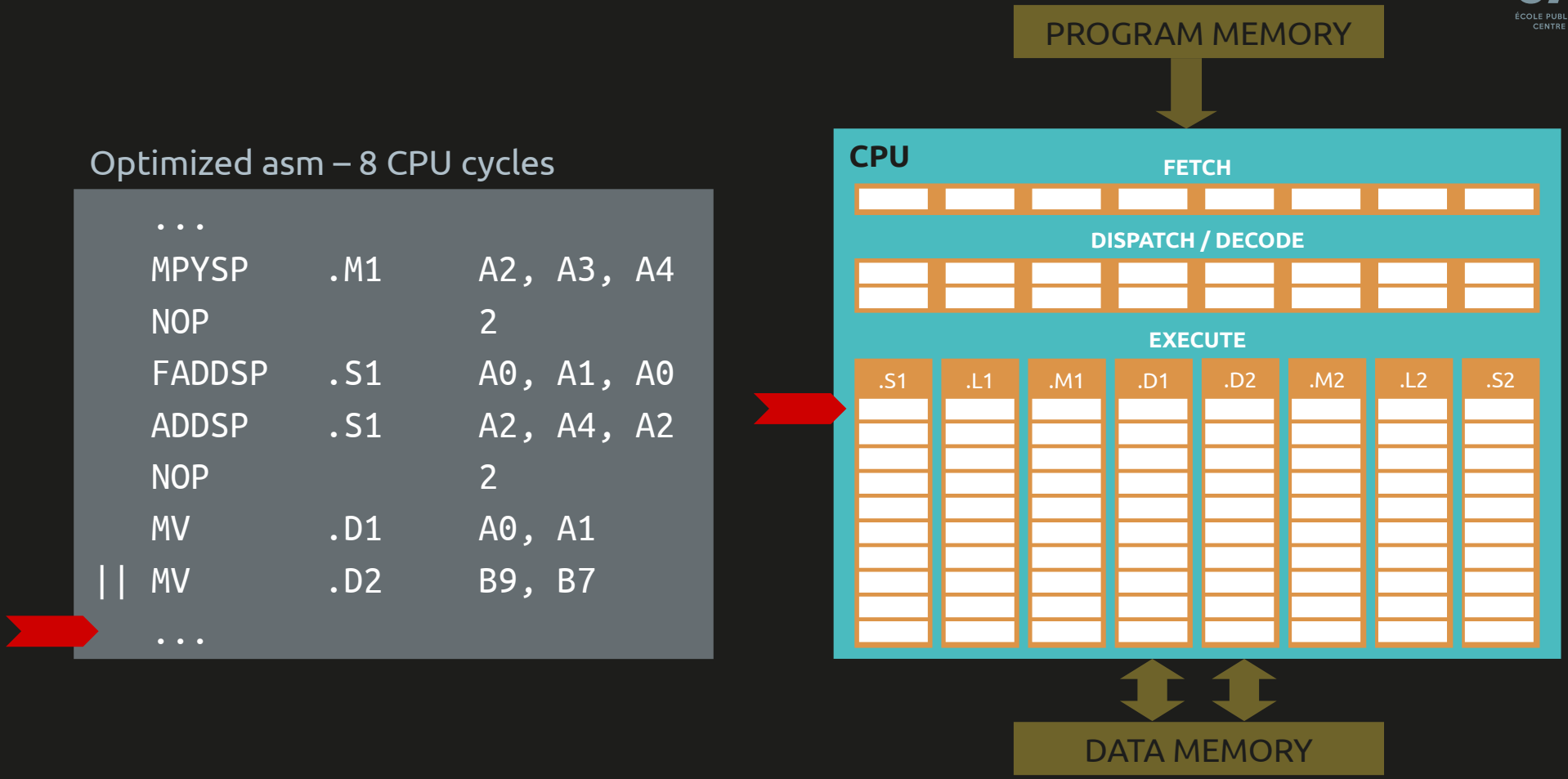


PROGRAMMING A VLIW CPU

Code execution

Optimized asm – 8 CPU cycles

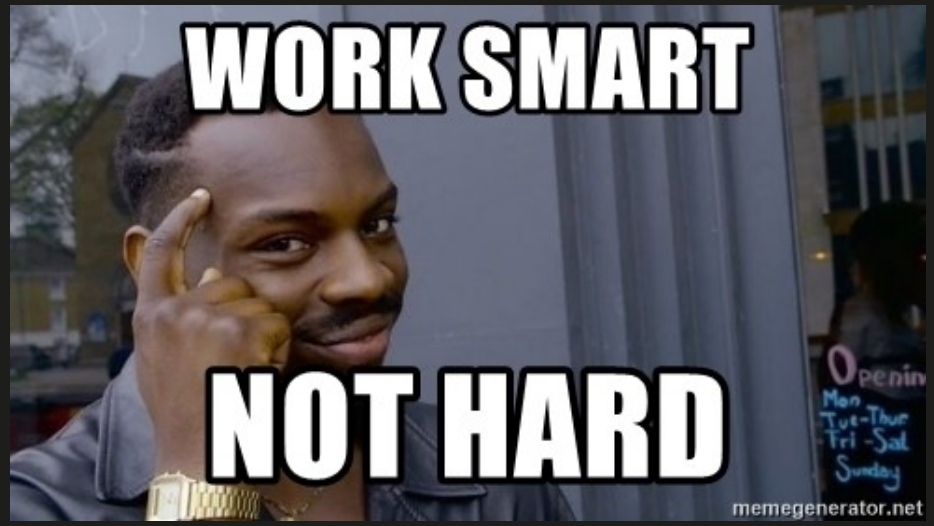
```
    ...
    MPYSP     .M1     A2, A3, A4
    NOP               2
    FADDSP    .S1     A0, A1, A0
    ADDSP     .S1     A2, A4, A2
    NOP               2
    MV        .D1     A0, A1
||  MV        .D2     B9, B7
    ...
```

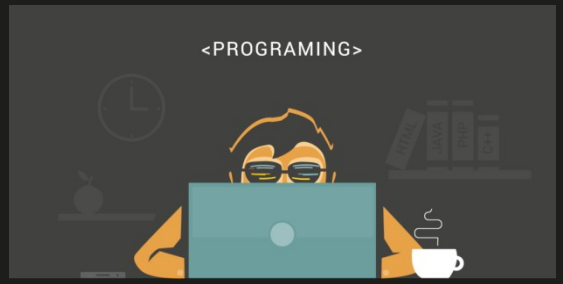


One particularity of VLIW processors is that their assembly code (and binary code as well) is out of order in the program memory, but they come out of the pipeline in order.

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

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





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

VLIW CPU properties

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.