Architecture Interne des Micro-contrôleurs

M. Briday

année 2024/2025

Contents

CENTRALE NANTES

Contents

CENTRALE NANTES

Objectif

- Comprendre l’architecure générale d’un processeur - flot de données, flot de contrôle
- Déterminer les éléments matériels pour accélérer la vitesse de traitement:
  - fontionnement, limites
  - prise en compte au niveau logiciel

Progression

- ISA RISC-V / assembleur
- de l’assembleur au matériel (base)
- optimisation:
  - pipeline
  - prédiction de branchement
  - hiérarchie mémoire

Contents

CENTRALE NANTES

Objective

- fill the gap between the software (SW) part and the hardware (HW)
- what is the interface between SW and HW?
- what can be done to improve software perf?

Guiding principles in computer architecture 1/2

- *Moore's Law*: integrated circuits double each 18-24 months (since 1965...). It's getting more difficult nowadays
- *Abstraction*: We need different levels of representation to understand the whole complexity (for both SW and HW parts)
- Set priority on the *common case fast*, at the expense of the rare case. (not always the case, specially for real time systems!)
- *Dependability via redundancy*. A basic approach when the hardware fails.

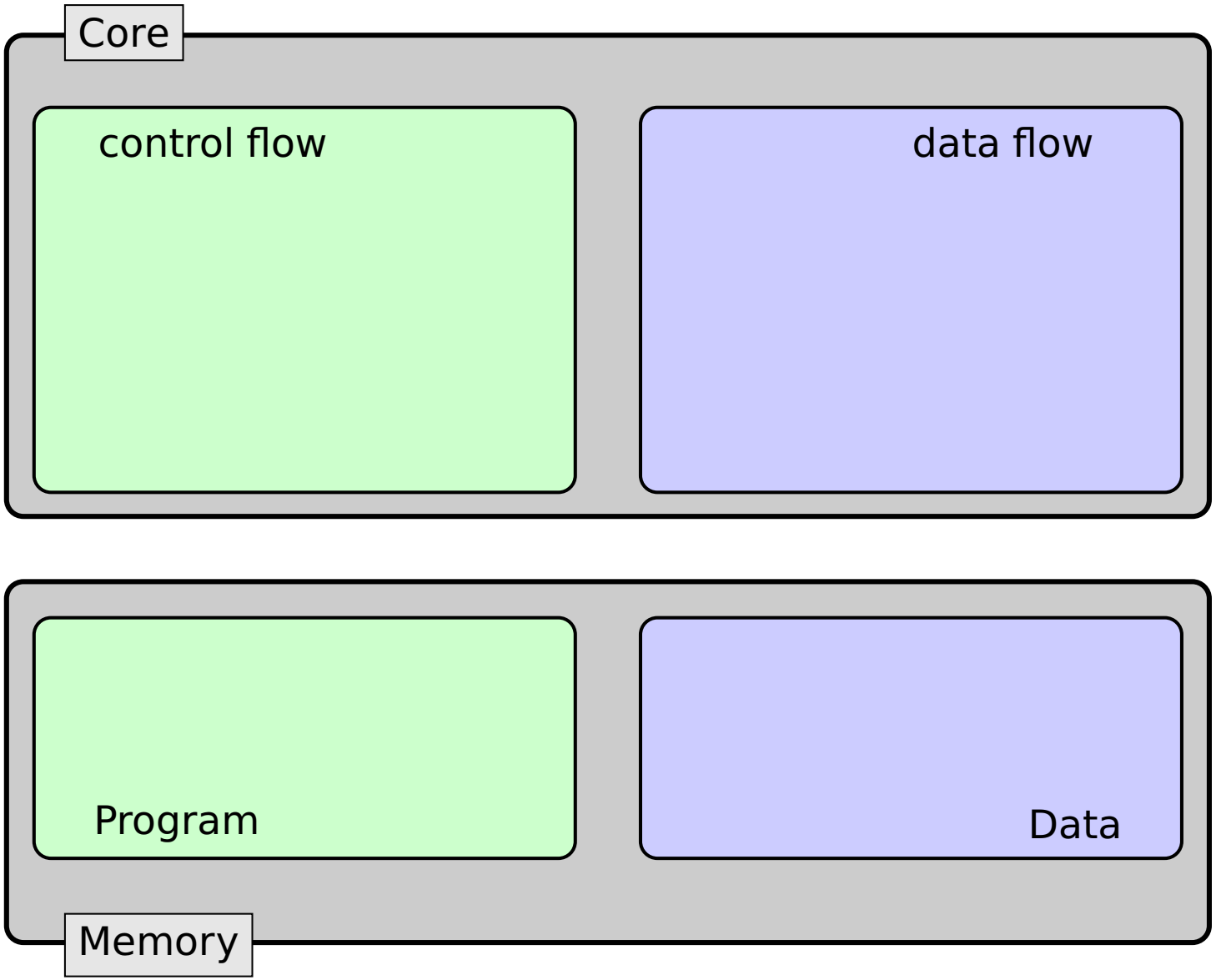
CENTRALE NANTES

Guiding principles in computer architecture 2/2

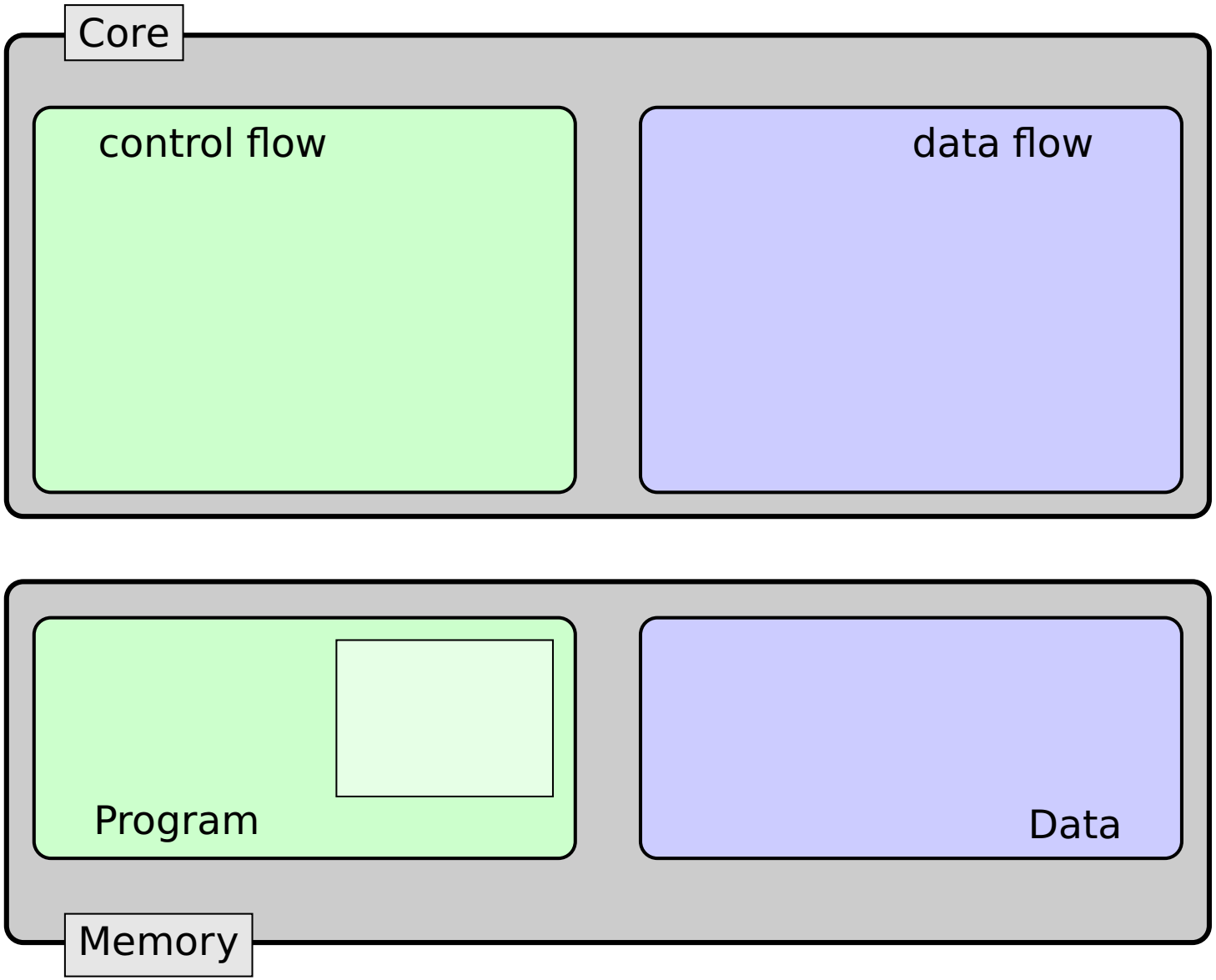
- Use *parallelism*: The hardware execution is naturally in parallel
- Use *pipelining*: split tasks in multiple subtasks and share them for different jobs.
- *Prediction*: in some cases, it is more efficient to try to guess a result and work on it than waiting for the safe data. Specially if the misprediction is not so expensive.
- *Memory hierarchy* is used to attenuate the speed difference between the core and the memory.

CENTRALE NANTES

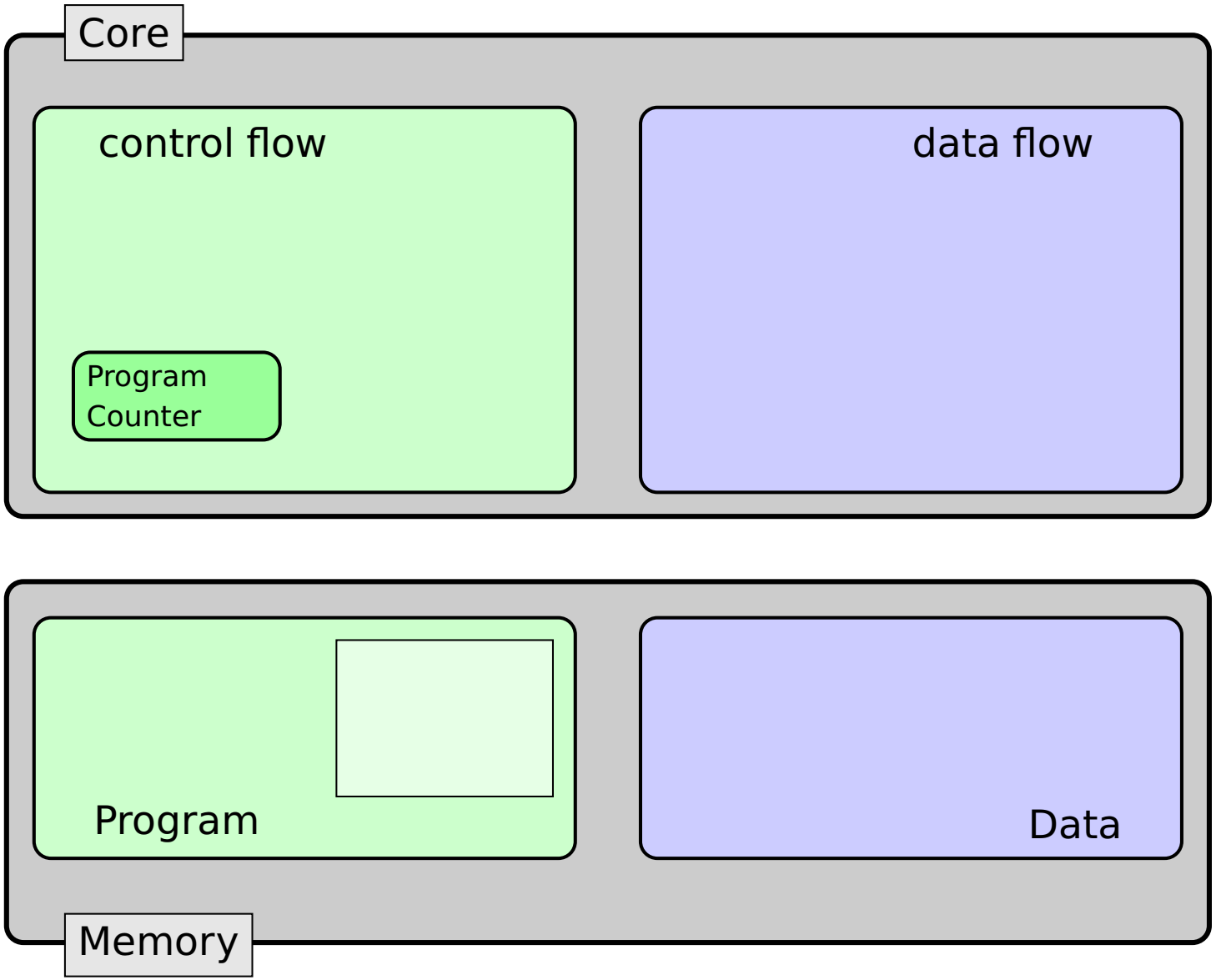
Architecture: The big picture



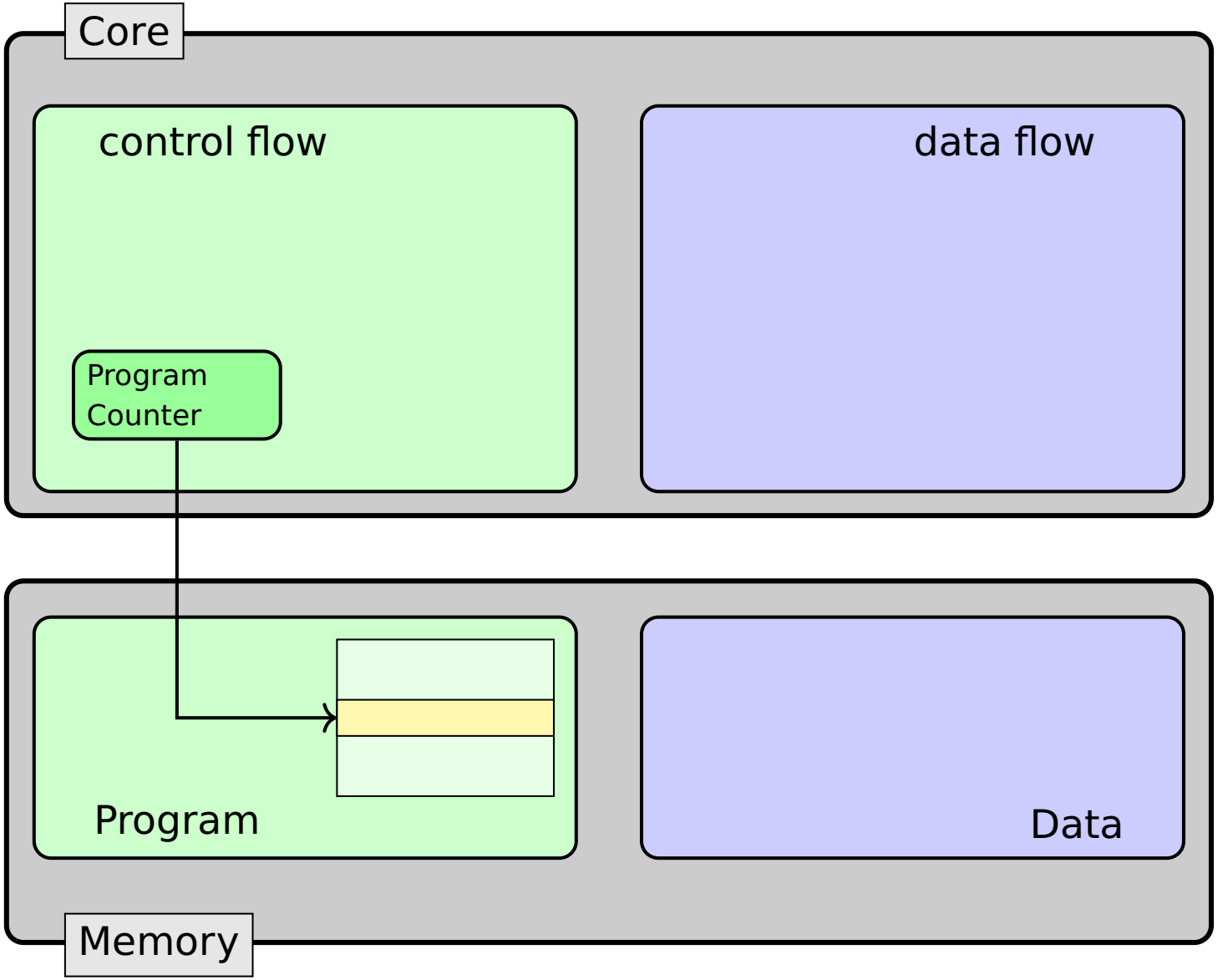
Architecture: The big picture



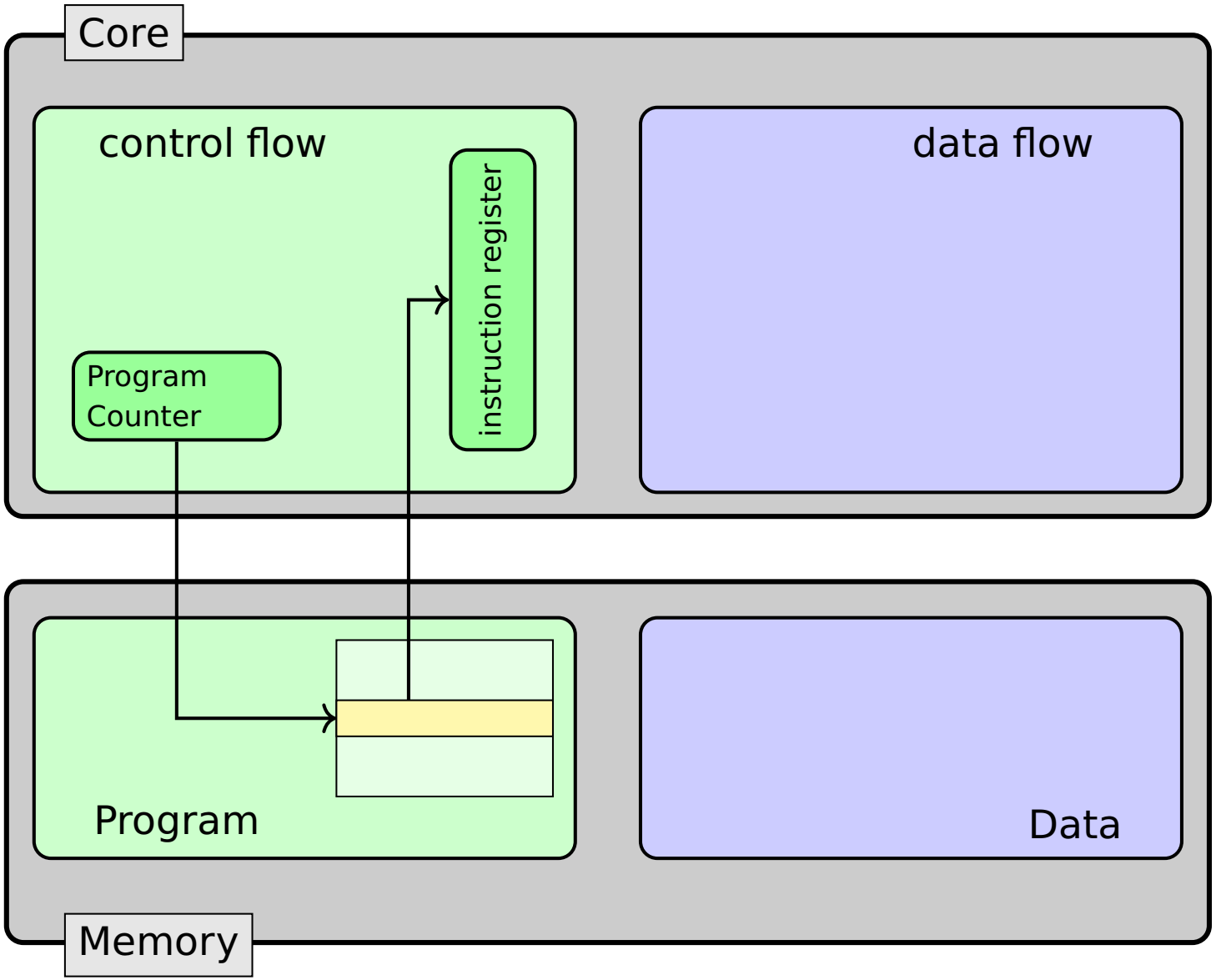
Architecture: The big picture



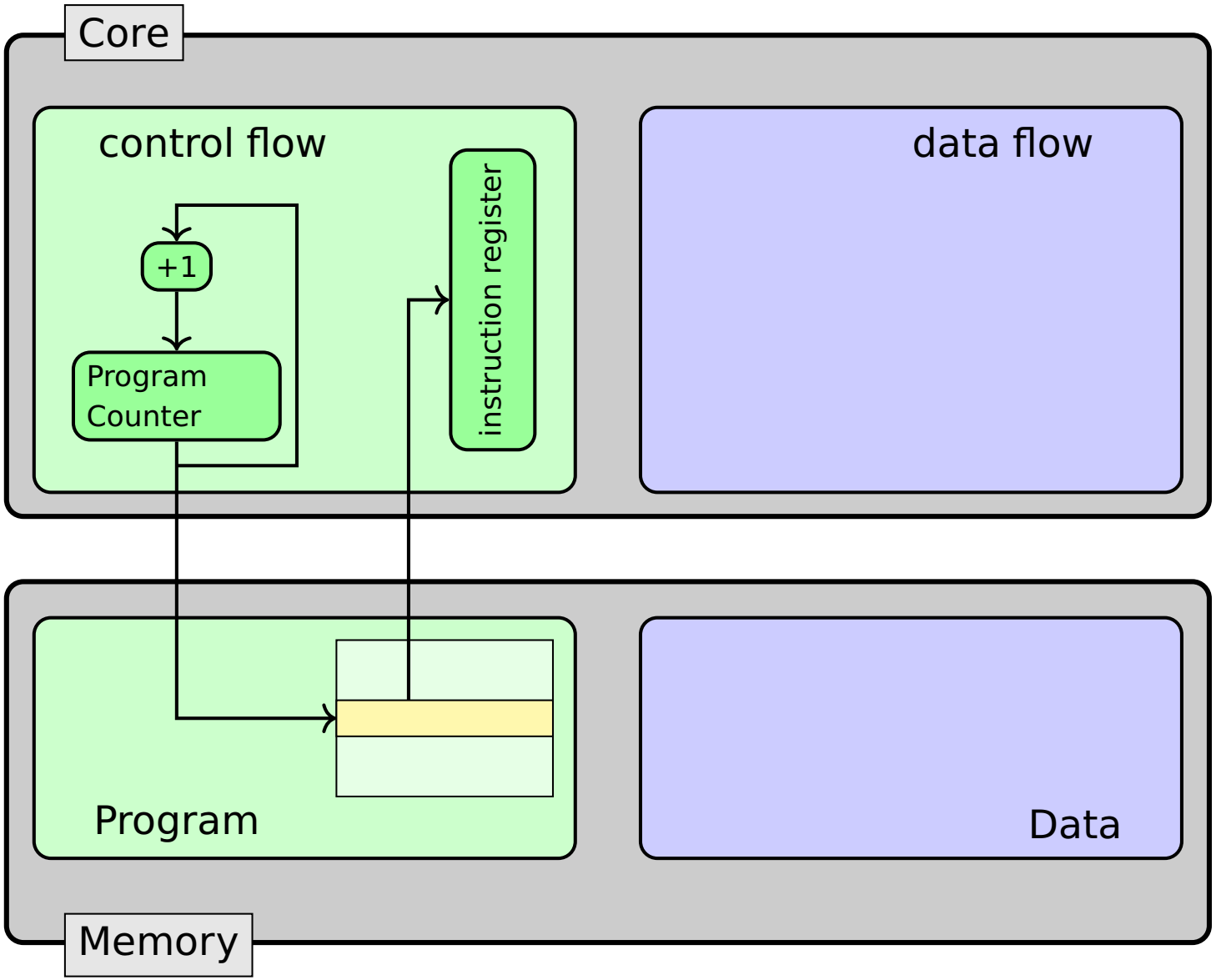
Architecture: The big picture



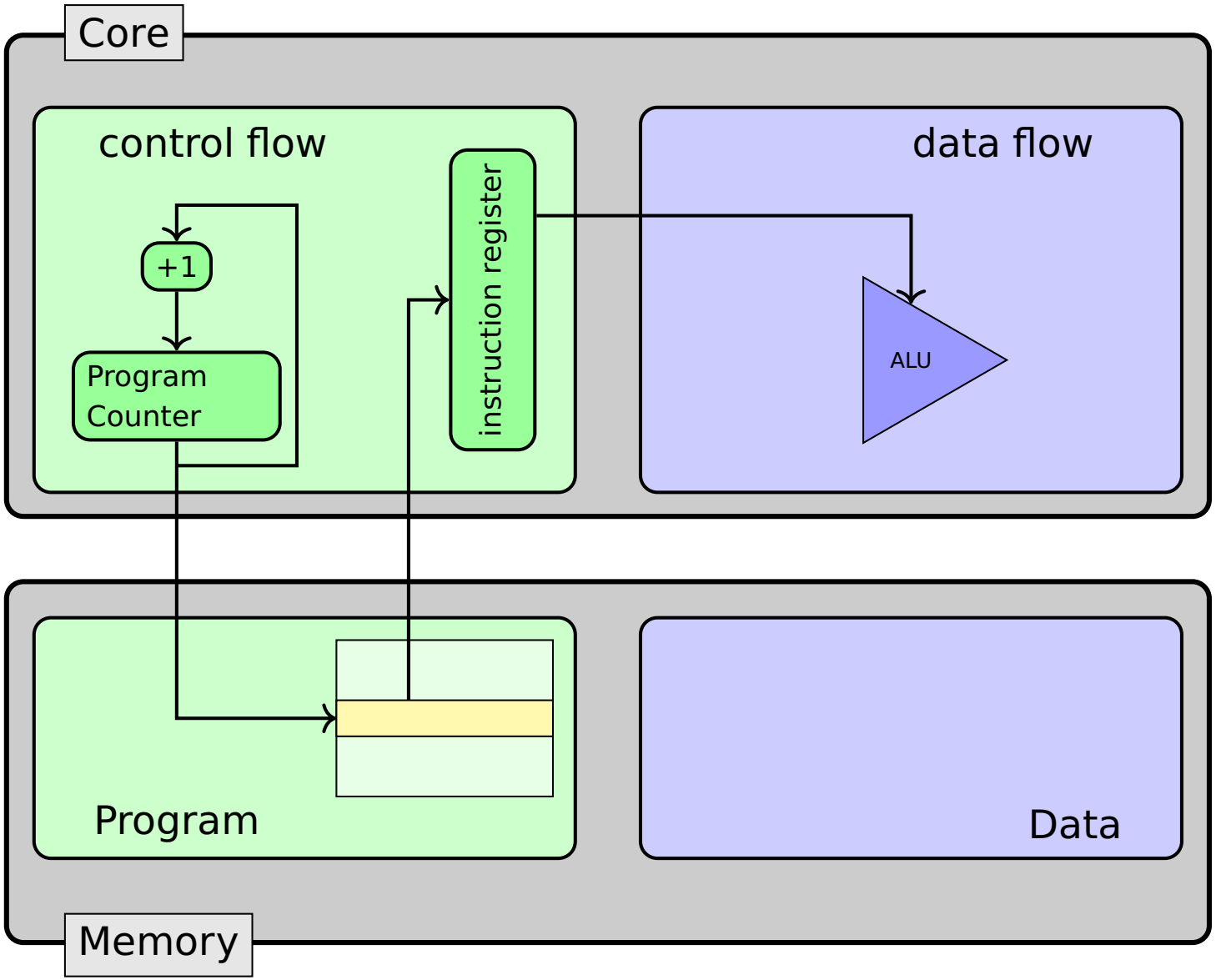
Architecture: The big picture



Architecture: The big picture

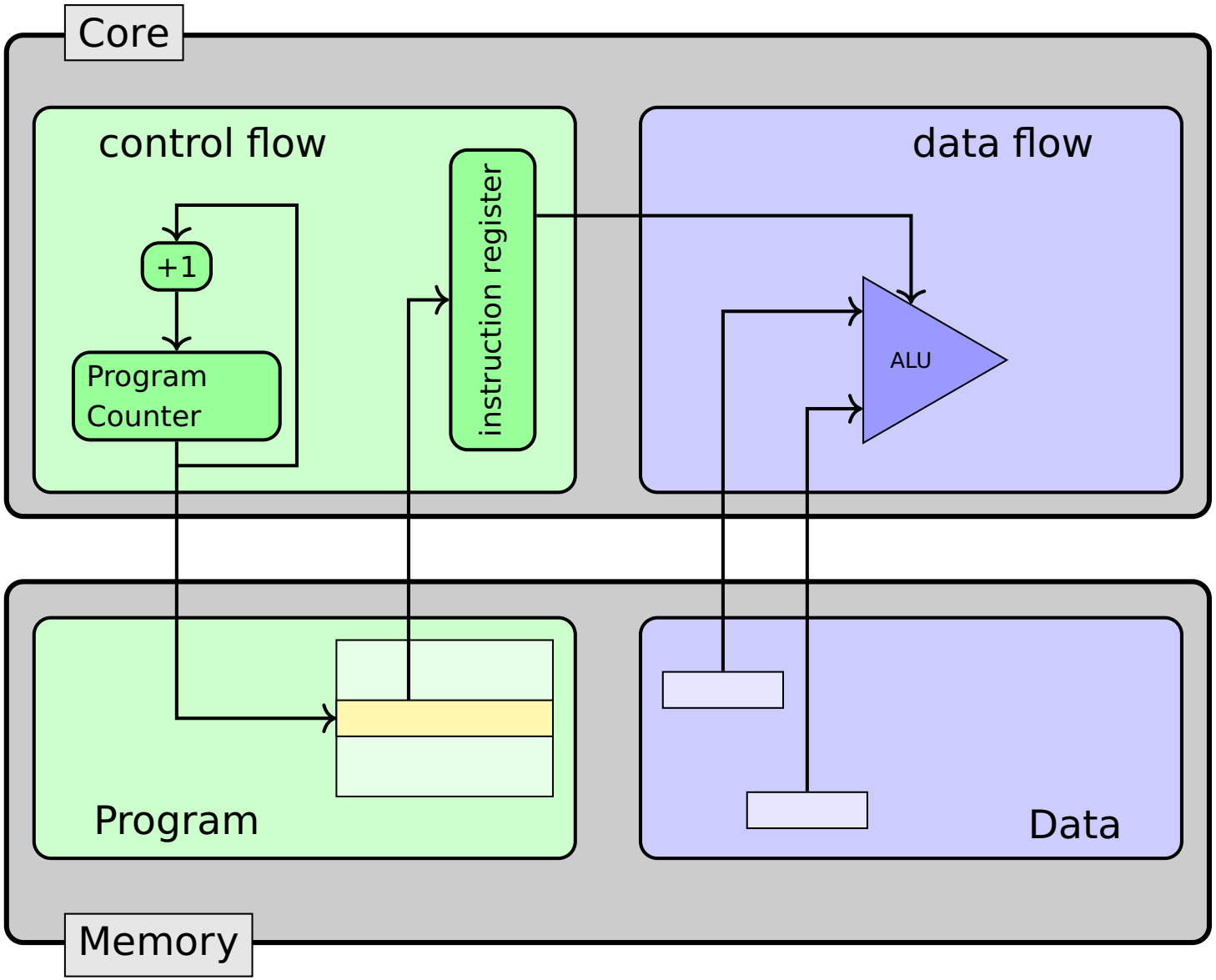


Architecture: The big picture



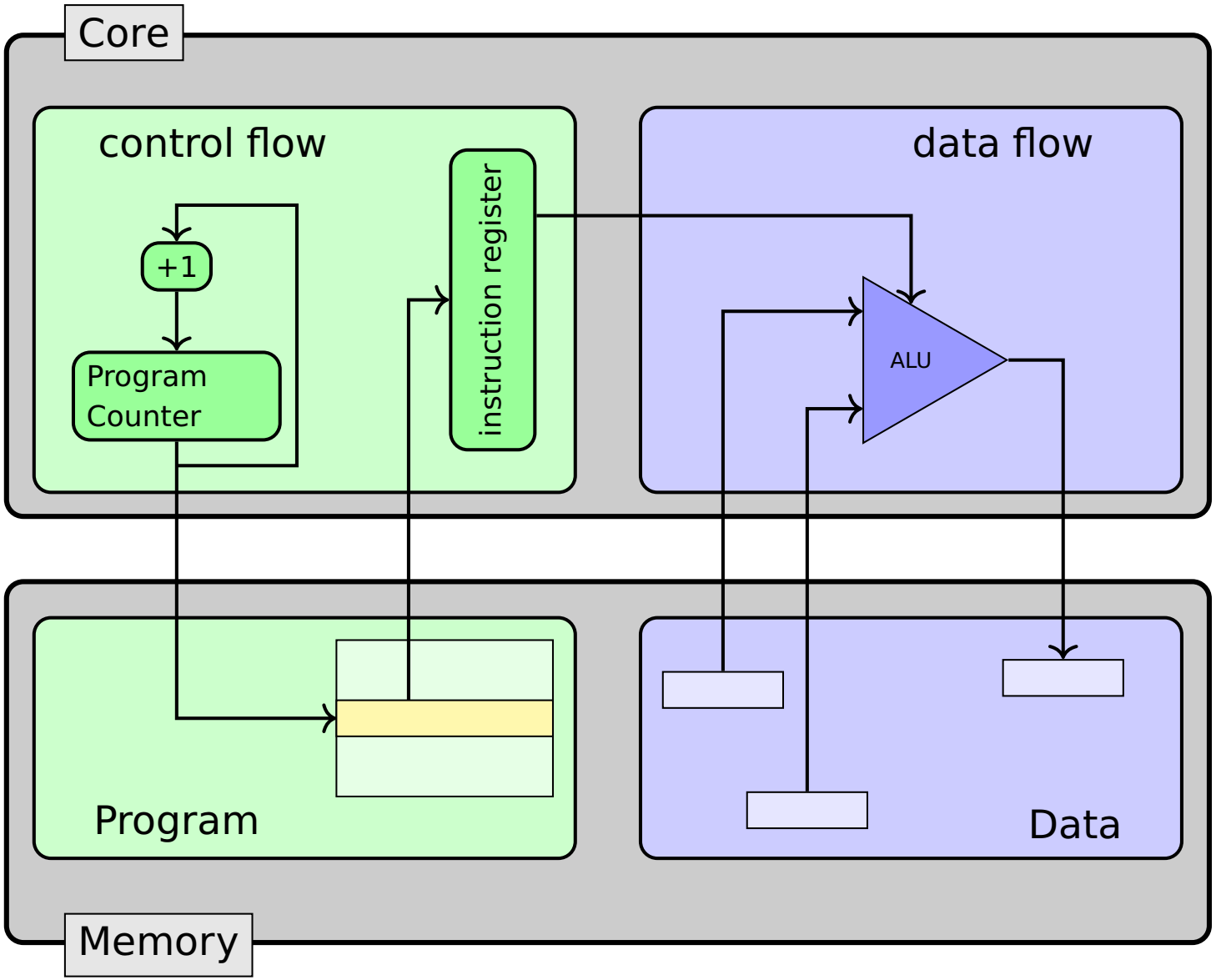
ALU: *Arithmetic and Logic Unit*

Architecture: The big picture



ALU: *Arithmetic and Logic Unit*

Architecture: The big picture



ALU: *Arithmetic and Logic Unit*

Architecture: The big picture

What we need:

- memory for instructions and data (R/W).
- high speed memory for intermediate results (registers)
- data path (manipulating data)
- control path (manipulating instruction execution)

2 main approaches:

- *Von Neumann* architecture: One main memory bus
- *Harvard* architecture: 2 memory busses for instructions and data.

CENTRALE NANTES

Instruction Set Architecture ?

Instruction set

An instruction is a basic word of computer's language. The instruction set is the vocabulary, i.e. all the available instructions for an architecture.

Instruction set examples:

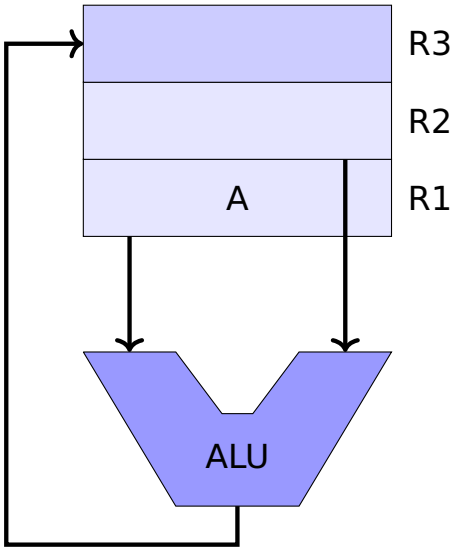
Architecture	**Instruction set**
Intel 8086 .. 80486, Pentium, Core i3,5,7,9...	Intel 80x86
Cortex M4	ARMV7e-M
Apple Axx, Qualcomm Snap-dragon (since 808)	ARMv8-A
Microchip PIC32	MIPS32
Microchip AVR	AVR8
RV32I, RV64xx, ...	RISC-V

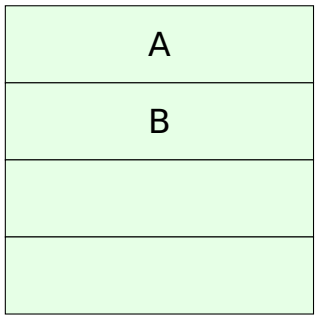
Instruction Set Architecture

The binary code is in fact a series of elementary instructions ordered in machine language. One elementary instructions can do:

- *data* manipulation:
  - arithmetic or logical operation such as addition, subtraction, AND, OR, ...
  - move data (memory, registers)
- *control flow* manipulation: a branch (to another program location);
- some specific operations (sleep, operations mode, . . .).

Instruction Set Architecture Architecture classes





Different architecture classes:

- stack: ALU connected on 2 fixed memory locations
- accumulator based: 1 memory location is fixed
- register-memory: 1 memory location can be in global memory
- **load-store**: ALU connected to registers only.

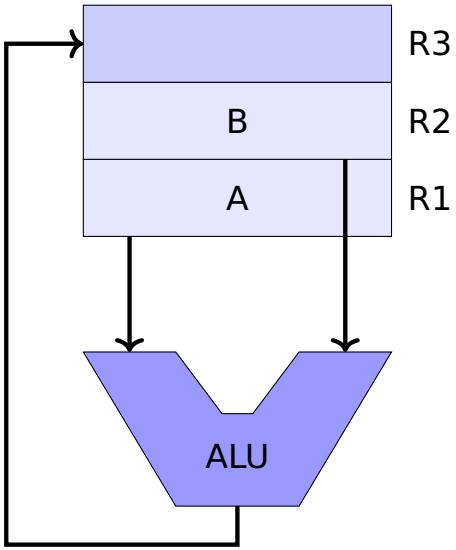
Steps for an addition of A and B (in memory). Results get back in memory C.

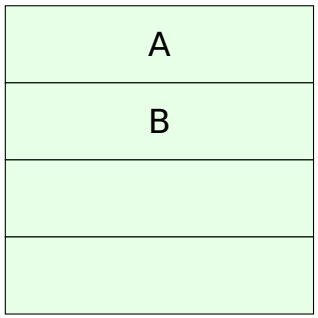
register (Load-Store)

- load R1,A

example: alpha (1992), ARM (1985), MIPS (1985), PowerPC (1992), SPARC (1986), SuperH (1990), . . .

Instruction Set Architecture Architecture classes





Different architecture classes:

- stack: ALU connected on 2 fixed memory locations
- accumulator based: 1 memory location is fixed
- register-memory: 1 memory location can be in global memory
- **load-store**: ALU connected to registers only.

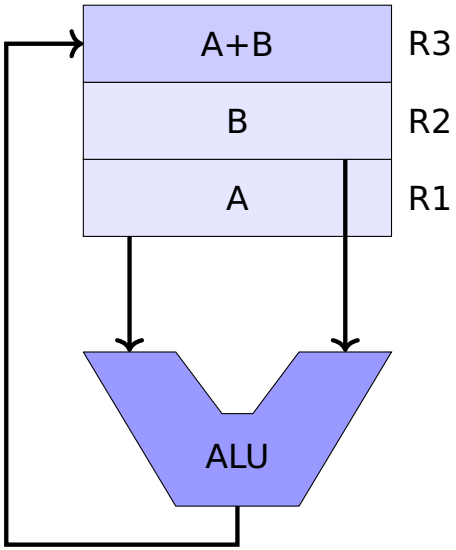
Steps for an addition of A and B (in memory). Results get back in memory C.

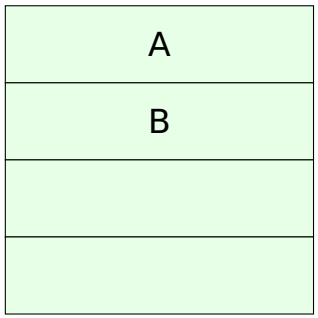
register (Load-Store)

- load R1,A
- load R2,B

example: alpha (1992), ARM (1985), MIPS (1985), PowerPC (1992), SPARC (1986), SuperH (1990), . . .

Instruction Set Architecture Architecture classes





Different architecture classes:

- stack: ALU connected on 2 fixed memory locations
- accumulator based: 1 memory location is fixed
- register-memory: 1 memory location can be in global memory
- **load-store**: ALU connected to registers only.

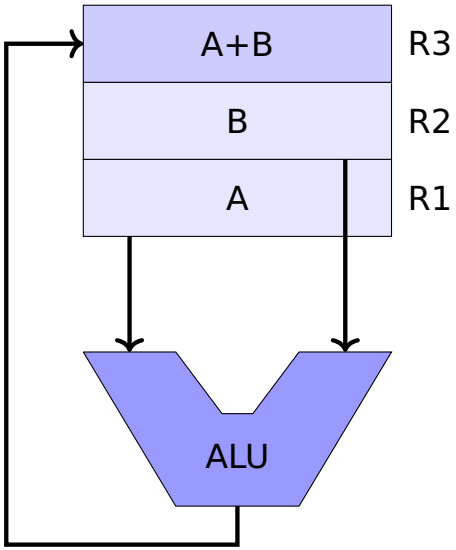
Steps for an addition of A and B (in memory). Results get back in memory C.

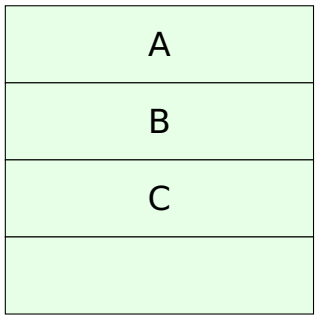
register (Load-Store)

- load R1,A
- load R2,B
- add R3,R1,R2

example: alpha (1992), ARM (1985), MIPS (1985), PowerPC (1992), SPARC (1986), SuperH (1990), . . .

Instruction Set Architecture Architecture classes





Different architecture classes:

- stack: ALU connected on 2 fixed memory locations
- accumulator based: 1 memory location is fixed
- register-memory: 1 memory location can be in global memory
- **load-store**: ALU connected to registers only.

Steps for an addition of A and B (in memory). Results get back in memory C.

register (Load-Store)

- load R1,A
- load R2,B
- add R3,R1,R2
- store R3,C

example: alpha (1992), ARM (1985), MIPS (1985), PowerPC (1992), SPARC (1986), SuperH (1990), . . .

Addressing modes

mode	**Example**	**Meaning**	**usage**
Register	add r1,r2	$r1 \leftarrow r1 + r2$	when a value is in a register
Immediat	add r1,#4	$r1 \leftarrow r1 + 4$	for constants
reg. indirect	add r1,(r2)	$r1 \leftarrow r1 + mem[r2]$	accessing using a pointer
displacement	add r1,100(r2)	$r1 \leftarrow r1 + mem[r2 + 100]$	accessing local var
indexed	add r1,(r2+r3)	$r1 \leftarrow r1 + mem[r2 + r3]$	in array: r2: base of array, r3 index
direct/abs	add r1,(100)	$r1 \leftarrow r1 + mem[100]$	static data, SFR
mem indirect	add r1,@r2	$r1 \leftarrow r1 + mem[mem[r2]]$	if r2 is the address of a pointer p, gets *p
autoincrement	add r1,(r2)+	$r1 \leftarrow r1 + mem[r2]$ $r2 \leftarrow r2 + d$	loops with an array d is the size of an elt
autodecrement	add r1,-(r2)	$r2 \leftarrow r2 - d$ $r1 \leftarrow r1 + mem[r2]$	idem

CISC vs RISC

CISC: Complex Instruction Set Computer

- instructions that perform complex operations (memory to memory, ...)
- code reduction - efficient when programming in asm
- many adressing modes
- needs few registers, with dedicated usage
- heavy decoding phase

In fact, 80% of the program needs only 20% of the instruction set.

a compiler that uses only `mov` instructions:
`https://github.com/xoreaxeaxeax/movfuscator`

CENTRALE NANTES

CISC vs RISC

RISC: Reduced Instruction Set Computer

- simple instructions: load/store
- instruction size is fixed.
- can be easily pipelined
- efficient for compiler optimisations
- more registers (general purpose)

We focus on the ARMv7-M instruction set (a mostly 'RISC' architecture)

CENTRALE NANTES

Example: instruction average on the 80x86 (CISC)

The top 10 instructions for the 80x86 for the five benchmark programs of *SPECInt92*:

rank	80x86 instruction	% of total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	mov reg-reg	4%
9	call	1%
10	return	1%
total		**96%**

Example: instruction average on the 80x86 (CISC)

The top 10 instructions for the 80x86 for the five benchmark programs of *SPECInt92*:

rank	80x86 instruction	% of total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	mov reg-reg	4%
9	call	1%
10	return	1%
total		**96%**

Most of the code of CISC use only a very small subset of the ISA.

Basic Core Architecture - Registers

On a RISC architecture, registers are used to:

- stores temporary variables to compute: *General Purpose Registers*;
- store the address of the next instruction to execute: *Program Counter*;
- store the address of the stack: the *Stack Pointer*
- stores the status of UAL operations (overflow, carry, zero, negative): the *status register*.

RISC

Few addressing modes, few instructions. The instructions are simple and memory accesses are only done through load/store instructions.

CENTRALE NANTES

VLIW

Certains processeurs permettent d'encoder plusieurs opérations (accès mémoire, opération entière, flottante, . . .) dans la même *instruction*

- Chaque operation utilise un *slot* de l'instruction
- l'architecture doit garantir qu'il n'y a pas de dépendance entre les opérations. . .

Contrainte

il faut ordonnancer correctement les instructions pour exploiter ce parallélisme. C'est le rôle du compilateur.

VLIW

Exemple: Intel Itanium (EPIC IA-64).
problème:

- explosion de la taille de code
- complexité du compilateur

Mais aussi. . .

- difficulté de la prédiction de branchement
- latence mémoire non prévisible (complexité du cache)

Si bien que l'architecture fut renommée *Itanic* par ses détracteurs.

Équation de performance

Un processeur est un système synchrone, cadencé à une certaine fréquence. Le temps CPU (CPU_{time}) est:

$$CPU_{time} = CPU_{clock\ Cycles\ for\ a\ program} \times Clock\ cycle\ time$$

Ou

$$CPU_{time} = \frac{CPU_{clock\ Cycles\ for\ a\ program}}{Clock\ freq}$$

Équation de performance

Un processeur est un système synchrone, cadencé à une certaine fréquence. Le temps CPU (CPU_{time}) est:

$$CPU_{time} = CPU_{clock\ Cycles\ for\ a\ program} \times Clock\ cycle\ time$$

Ou

$$CPU_{time} = \frac{CPU_{clock\ Cycles\ for\ a\ program}}{Clock\ freq}$$

On définit le nombre de cycles par instruction (CPI):

$$CPI = \frac{CPU_{clock\ Cycles\ program}}{instructions\ count}$$

CENTRALE NANTES

Équation de performance

Ainsi:

$$CPU_{time} = instructions\ count \times CPI \times Clock\ cycle\ time$$

et:

$$CPU_{time} = \frac{seconds}{Program} = \frac{instructions}{Program} \times \frac{Clock\ cycles}{Instruction} \times \frac{seconds}{Clock\ cycle}$$

À noter

La performance du processeur dépend de ces 3 critères, de manière équivalente: 10% d'amélioration sur un des 3 critères améliore de la même manière les performances.

Équation de performance

- $\frac{instructions}{Program}$: nombre d'instruction: dépend du *compilateur*, et du *jeu d'instruction*.

Équation de performance

- $\frac{instructions}{Program}$: nombre d'instruction: dépend du *compilateur*, et du *jeu d'instruction*.
- $\frac{Clock\ cycles}{Instruction}$ (ou CPI): dépend du *jeu d'instruction* et de l'*architecture interne*

Équation de performance

- $\frac{instructions}{Program}$: nombre d'instruction: dépend du *compilateur*, et du *jeu d'instruction*.
- $\frac{Clock\ cycles}{Instruction}$ (ou CPI): dépend du *jeu d'instruction* et de l'*architecture interne*
- $\frac{seconds}{Clock\ cycle}$ (fréquence de fonctionnement): dépend de la *technologie* (finesse de gravure) et de l'*architecture interne*.

CENTRALE NANTES

Équation de performance

Dans la pratique, le nombre de cycle par instruction (*CPI*) dépend du type d'instruction. Soit n le nombre de type d'instructions:

$$CPU_{time} = (\sum_{i=1}^{n} IC_i \times CPI_i) \times Clock\ cycle\ time$$

et on obtient le CPI moyen:

$$CPI = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{instruction\ count}$$

Exemple

on considère un programme, dont:

- 25% des instructions sont à virgule flottante (FP), avec un CPI de 4 (en moyenne)
- CPI = 1.33 pour les autres instructions (75%)
- Quel est le CPI moyen sur l'application?

Parmis les opérations FP, l'instruction FPSQRT requiert 20 cycles, et est présente à 2%.

- 2 conceptions sont évaluées:
  - la première diminue le CPI de FPSQRT à 2
  - la seconde diminue globalement le CPI des instructions FP à 2.5
- quelle est la meilleure approche?

Contents

The RISC-V Instruction Set

- Origine: Université de Berkeley (US), 2010
- Une ISA libre et accessible gratuitement à destination des universitaires et de l'industrie;
- Gestion par la fondation RISC-V (`https://riscv.org`)
- Typique de plusieurs architectures RISC modernes

Pas d'implémentation de réference

RISC-V maintient simplement un document PDF!

The RISC-V Instruction Set

- 3 ISA en fonction de la taille des adresses: RV32I, RV64I, RV128I
- RV32I: moins de 40 instructions
- RV32E: versions réduite à 16 registres pour l'embarqué
- instructions codées sur 32 bits, sauf extension C.
- certaines extensions ne sont pas encore ratifiées (*draft*).

Name	**Extension**
I	basic Integer instructions
M	integer Multiply/divide
A	Atomic
F	single precision Floating point (FP)
D	Double precision FP
G	General purpose (=IMAFD)
Q	Quad precision FP
C	Compressed ISA (16/32 bits)
B	*Bit manipulation*
T	*Transactional Memory*
J	*Dynamically Translated Languages*
V	*Vector Operations*
...	

Dans ce cours, on s'intéresse uniquement au jeu d'instruction RV32I qui définit 39 instructions.

Intérêt

- Architecture modulaire: ISA avec des extensions
- Architecture de base RV32I est très réduite
- Utilisation des chaines de compilations classiques (gcc, llvm, . . .)
- Possibilité d’ajouter des instructions spécifiques (sécurité, accélérateurs, . . .)
- Industriels: royalty free / patent free.

Écosystème

- Compilation: GCC, LLVM
- Simulation: Spike (référence), QEMU, Gem5
- Debug: openOCD
- OS: Linux, sel4, freeRTOS, Zephyr, ...

Spécificités

- Modulaire: beaucoup d'extensions
- standard ouvert
- les décisions principales sont motivées et expliquées:

Although 64-bit address spaces are a requirement for larger systems, we believe 32-bit address spaces will remain adequate for many embedded and client devices for decades to come and will be desirable to lower memory traffic and energy consumption. In addition, 32-bit address spaces are sufficient for educational purposes. A larger flat 128-bit address space might eventually be required, so we ensured this could be accommodated within the RISC-V ISA framework.

Figure: justification de l'architecture 32 bits (p.4 - risc-V spec)

(voir: justification architecture modulaire p.4)

RISC-V registres - instructions

voir fichier **riscv-user-isa.pdf** À noter:

- 32 registres + le compteur programme: `PC`
- le registre `x0` est câblé à 0
- par défaut, la version de base n'offre pas de registre d'état (de l'ALU): `CSR`: Control and Status Register
- certaines instructions sont des pseudo-instruction: c'est un synonyme. Exemple: `li rd, immm` est en fait une addition.
- registres de 32, 64 ou 128 bits.
- les registres ont 2 noms: `x0...x31` génériques, et `ra, sp, gp, ...` en fonction de l'ABI (voir p.44)

Example of assembly code

A C code is translated to assembly code, thanks to the compiler.
Columns in assembly listing are:

- program memory address (hexadecimal);
- instruction code (hexadecimal);
- instruction mnemonic (textual representation)

```c
#include <stdint.h
uint32_t b[10];

int main(void) {
    for(int i=0;i<10;i++)
    {
        b[i] = i;
    }
    return 0;
}
```

```
00000018 <main>:
18: 800007b7            lui  a5,0x80000
1c: 00078793            mv   a5,a5
20: 00000713            li   a4,0
24: 00a00693            li   a3,10
28: 00e7a023            sw   a4,0(a5) # 80000000
2c: 00170713            addi a4,a4,1
30: 00478793            addi a5,a5,4
34: fed71ae3            bne  a4,a3,28 <main+0x10>
38: 00000513            li   a0,0
3c: 00008067            ret
```

Exercice

On considère dans les exercices le jeux d'instruction RV32IM (instructions entières+multiplications/divisions), sauf indication contraire.
En vous servant des instructions arithmétiques et logiques, donner le code ASM pour réaliser:

- `x1 <- x2+x4*x5`
- `x2 <- 2*x4+3`
- `x2 <- 5*x4` (avec l'ISA RV32I).
- `x2 <- x1 + 0x12345678`

CENTRALE NANTES

Exercice

On considère dans les exercices le jeux d'instruction RV32IM (instructions entières+multiplications/divisions), sauf indication contraire.
En vous servant des instructions arithmétiques et logiques, donner le code ASM pour réaliser:

- `x1 <- x2+x4*x5`
- `x2 <- 2*x4+3`
- `x2 <- 5*x4` (avec l'ISA RV32I).
- `x2 <- x1 + 0x12345678`

Dans la pratique pour le dernier exemple, on pourra écrire:
`li x2,0x12345678`, et l'assembleur redécoupera en 2 instructions de base.

Architecture de type Load/Store

- toutes les opérations se font sur les registres internes (pas de manipulation directe en mémoire)
- instructions *Load* pour copier de la mémoire vers les registres
- instructions *R-type* et *I-type* pour faire les opérations
- instructions *Store* pour recopier en mémoire
- instructions de branchement (*Branch*, *Jump*)

RISC-V registres - instructions

Les registres sont banalisés (tous identiques sauf `x0`), mais l'ABI impose un sens à certains registres (on y reviendra après. . .)

- return address
- stack pointer
- global pointer
- thread pointer
- function arguments
- fp related registers

RISC-V modes d'adressage

Peu de modes d'adressages! (architecture RISC).

- immédiat:
  ```
  li x1, 10;x1 <- 10
  ```
- registre:
  ```
  add x1,x2,x3;x1 <- x2 + x3
  ```
- registre+offset:
  ```
  lw x1,10(x2);x1 <- mem[x2+10]
  ```
- PC+offset:
  ```
  auipc x2,0x80000 ;x2 <- pc+0x80000
  ```

CENTRALE NANTES

Appel de fonction

Pour un appel de fonction, il faut:

- enregistrer la valeur de retour
- modifier la valeur de pc (saut)

C'est le rôle de l'instruction `jal`:

```
jal x1,6c
```

qui va:

- sauver pc+4 dans x1
- sauter à l'adresse 6c (modifier pc)

Appel de fonction

À la fin de la fonction (retour à la fonction appelante), il suffit de sauter à la valeur précédemment enregistrée:

```
ret
```

L'instruction copie x1 dans pc.
l'instruction `ret` est en réalité une pseudo-instruction (`jalr x0, 0(x1)`).

Appel de fonction

À la fin de la fonction (retour à la fonction appelante), il suffit de sauter à la valeur précédemment enregistrée:

```
        ret
```

L'instruction copie x1 dans pc.
l'instruction `ret` est en réalité une pseudo-instruction (`jalr x0, 0(x1)`).

Mais pourquoi x1 et pas x2!?!

Appel de fonction

Exemple:

```
  jal x1,func1
  jal x1,func2
  jal x1,func3
# ..
func1:
  #code func1
  ret

func2:
  #code func2
  ret

func2:
  #code func3
  ret
```

Appel de fonction - ABI

L'ABI (*Application Binary Interface*) définit un ensemble de règles pour que différents binaires (issus de différents compilateurs) puissent co-exister:

- passage des arguments lors des appels de fonction (entrée, sortie)
- préservation des registres (ou non) dans les fonctions appelées
- gestion de la pile
- ...

Dans le RISC-V, les registres sont nommés de `x0` à `x31`, ou à travers leur fonction dans l'ABI.

ABI

Les registres généraux se distinguent entre:

- ceux qui sont sauvés par l'appelant (*Caller*):
  - ils doivent être sauvé par l'appelant si ces registres sont utilisés dans la fonction (sur la pile)
  - ils peuvent être utilisés sans restriction par l'appelé
- ceux qui sont sauvés par l'appelé (*Callee*)
  - l'appelant n'a pas à y faire attention
  - l'appelé doit faire une sauvegarde s'il veut utiliser le registre

CENTRALE NANTES

Registres - ABI

register	ABI name	description	saved by
x0	zero	hardwired zero	-
x1	ra	return address	caller
x2	sp	stack pointer	*callee*
x3	gp	global pointer	-
x4	tp	thread pointer	-
x5..7	t0..t2	temporary registers	caller
x8..9	s0..s1	saved registers	*callee*
x10..17	a0..a7	function arguments	caller
x18..27	s2..s11	saved arguments	*callee*
x28..31	t3..t6	temporary arguments	caller
pc	pc	program counter	-

CENTRALE NANTES

ABI

`ra`	c'est le *link register*, ou *return address*: x1
`sp`	*stack pointer*: x2
`gp`	*global pointer*. Pointeur de base pour l'accès aux données sur le tas: x3
`tp`	*thread pointer*. Pointeur de base pour l'accès aux données du thread courant: x4
`s0-s11`	saved register. Sauvés par l'*appelant*
`a0-a7`	arguments de fonction, sauvés par l'*appelé*
`t0-t6`	registres temporaires, sauvés par l'*appelé*

CENTRALE NANTES

Arguments de fonction

Les règles pour le passage de paramètres de fonctions permettent d'utiliser des bibliothèques issus de compilateurs différents:
Dans notre cas (classique) où les arguments sont sur 32 bits max:

- a_i: argument d'entrée i
- `a0,a1`: arguments de retour

Exercice

Implémenter la fonction en assembleur, en respectant l'ABI.

```
int f(int a, int b)
{
    return 5*a+4*a*b+b*b;
}
```

Mise au point. . .

Un outil en ligne pour voir le code généré par différents compilateurs: `https://godbolt.org/`

Variables globales

en assembleur, on indique les portions de données avec le prologue `.data`.
Ici ga est une adresse en mémoire, et `0x12345678` est la valeur à l'initialisation:

```
.data
ga:
    .word 0x12345678
```

La récupération de l'adresse (*l*oad *a*ddress):

```
la t0,ga
```

ceci se décompose en fait en 2 instructions par l'assembleur:

```
    auipc t0, ...
    addi t0, t0, ...
```

Exercice

Implémenter la fonction en assembleur, en respectant l'ABI.

```
int var; //variable globale
int f(int a, int b)
{
    var += 1;
    return 4*a*b+2*ga;
}
```

RISC-V: branchements

Les branchements conditionels utilisent le résultat d'une soustraction de l'ALU.
bne signifie *Branch if Not Equal*.

```
bne a4,a3,28 # PC <- 0x28 si a4 != a3
```

En fait, le circuit:

- calcule la différence a4-a3
- vérifie la condition (résultat non nul)
- si la condition est vraie, met à jour la valeur de PC

Conditions issues de l'ALU

L'ALU permet de faire les opérations de base, et permet de sortir généralement 4 bits d'état:

- `N`: le résultat de la dernière opération est *négatif* (MSB = 1);
- `Z`: résultat nul (*zero*).
- `C`: la retenue *carry* de la dernière opération (non signée)
- `V`: dépassement de la dernière opération signée (*oVerflow*);

Spécificité RISC-V

Généralement, l'évaluation de la condition et le branchement sont réalisés par 2 instructions distinctes sur les architectures RISC. On utilise alors un registre d'état (*status register*) pour enregistrer ces bits d'état.

CENTRALE NANTES

Instructions de branchement

En fonction de l'état de des 4 bits, on peut implémenter les instructions de branchement.
Pour le RISC-V, il n'y a pas par défaut de registre qui stocke ces bits d'état.
Les instructions de branchement réalisent une soustraction `rs2-rs1` et branchent directement:

```
beq  rs1,rs2,imm ; branch if equal
bne  rs1,rs2,imm ; branch if not equal

blt  rs1,rs2,imm ; branch if lower than (signed)
bge  rs1,rs2,imm ; branch if greater or equal than (signed)

bltu rs1,rs2,imm ; branch if lower than (unsigned)
bgeu rs1,rs2,imm ; branch if greater or equal than (unsigned)
```

CENTRALE NANTES

Instructions de branchement

Quels vont être les drapeaux NZCV pour les opérations (sur 8 bits)?

- `0x7A` et `0x7A`
- `0x7A` et `0x5C`
- `0x8A` et `0x5C`
- `0x5C` et `0x8A`
- `0x2C` et `0x72`

Instructions de branchement

Quels vont être les drapeaux NZCV pour les opérations (sur 8 bits)?

- `0x7A` et `0x7A` (C_2=0x86)
- `0x7A` et `0x5C` (C_2=0xA4)
- `0x8A` et `0x5C` (C_2=0xA4)
- `0x5C` et `0x8A` (C_2=0x76)
- `0x2C` et `0x72` (C_2=0x8E)

Instructions de branchement

Quels vont être les drapeaux NZCV pour les opérations (sur 8 bits)?

- `0x7A` et `0x7A` (C_2=0x86) $\Rightarrow$ R=00
- `0x7A` et `0x5C` (C_2=0xA4) $\Rightarrow$ R=1E
- `0x8A` et `0x5C` (C_2=0xA4) $\Rightarrow$ R=2E
- `0x5C` et `0x8A` (C_2=0x76) $\Rightarrow$ R=D2
- `0x2C` et `0x72` (C_2=0x8E) $\Rightarrow$ R=BA

Instructions de branchement

Quels vont être les drapeaux NZCV pour les opérations (sur 8 bits)?

- 0x7A et 0x7A (C_2=0x86) $\Rightarrow$ R=00 :0 1 0 1
- 0x7A et 0x5C (C_2=0xA4) $\Rightarrow$ R=1E :0 0 0 1
- 0x8A et 0x5C (C_2=0xA4) $\Rightarrow$ R=2E :0 0 1 1
- 0x5C et 0x8A (C_2=0x76) $\Rightarrow$ R=D2 :1 0 1 0
- 0x2C et 0x72 (C_2=0x8E) $\Rightarrow$ R=BA :1 0 0 0

Branch conditions

Mnemo	Meaning (integer)	Flags
EQ	Equal	
NE	Not equal	
GE	Signed greater than or equal	
LT	Signed less than	
GEU	Unsigned higher or equal	
LTU	Unsigned lower	

Branch conditions

Mnemo	Meaning (integer)	Flags
EQ	Equal	Z == 1
NE	Not equal	Z == 0
GE	Signed greater than or equal	
LT	Signed less than	
GEU	Unsigned higher or equal	
LTU	Unsigned lower	

Branch conditions

Mnemo	Meaning (integer)	Flags
EQ	Equal	Z == 1
NE	Not equal	Z == 0
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GEU	Unsigned higher or equal	
LTU	Unsigned lower	

Branch conditions

Mnemo	Meaning (integer)	Flags
EQ	Equal	Z == 1
NE	Not equal	Z == 0
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GEU	Unsigned higher or equal	C == 1 or Z == 1
LTU	Unsigned lower	C == 0 and Z == 0

Exercice

Proposer une implémentation pour la fonction: `int max(int a, int b);`

Note: il est possible de rajouter un *label* qui correspond à une adresse dans le code, avec le *:*.

```
label1:
    la t0,ga
    li t1, 0x12
label2:
    ...
```

Fonction?

une fonction n'est alors rien d'autre qu'un label suivi d'une liste d'instruction. On peut s'en servir en C: le nom de la fonction est alors un *pointeur de fonction*, *i.e.* l'adresse de sa première instruction!

exercice boucle for?

en C, la boucle `for` se décompose en

```
for(init; condition; update) {body}
```

et peut être implémentée:

```
init
jump test

forBody:
body
update

test:
si condition, jump forBody
```

Exercice

faire une fonction qui calcule la somme des n premiers entiers (itérativement, sans formule!)

```
int sumInt(int n)
{
  int result = 0;
  for(int i = 1; i <= n; i++)
  {
    result += i;
  }
  return result;
}
```

Note: on utilisera des registres pour les variables locales...

Exercice - accès indirect

Faire une fonction qui calcule le max dans un tableau passé en paramètre.

```
int tab[SIZE];
int maxTab(int *t){..}
```

Exercice

Faire une fonction qui calcule la taille d'une chaine de caractère:

```
int strlen(const char *);
```

Déclaration d'une chaine

On peut déclarer une chaine de caractère d'identificant `chaineCar`:

```
.data
chaineCar:
  .string "Ceci est une chaine de caractere, terminee par le
     caractere nul."
```

Exercice

Faire la fonction que compare 2 chaines de caractères, jusqu'à n caractères:

```
int strncmp(const char * first, const char * second, size_t
    length);
```

La valeur de retour est nulle si les 2 chaines sont identiques, est inférieure (resp. supérieure) à 0 si la chaine `first` est inférieure (resp. supérieure) au sens lexicographique.

Limites pour les appels de fonction...

Et si les fonctions sont imbriquées? exemple:

```
  jal ra,func1
  # ..
  func1:
    #debut code func1
    jal ra,func2
    #fin code func1
    ret #ERREUR!!!

  func2:
    #code func2
    ret
```

Ça ne fonctionne plus car la valeur de ra est écrasée dans l'appel de func2.

Limites pour les appels de fonction...

Mais plus encore:

- Comment sauvegarder des registres dans les fonctions?
- Comment utiliser des variables locales?
- comment gérer les fonctions récursives?

Il manque une structure de donnée pour mémoriser tout ça...

Stack

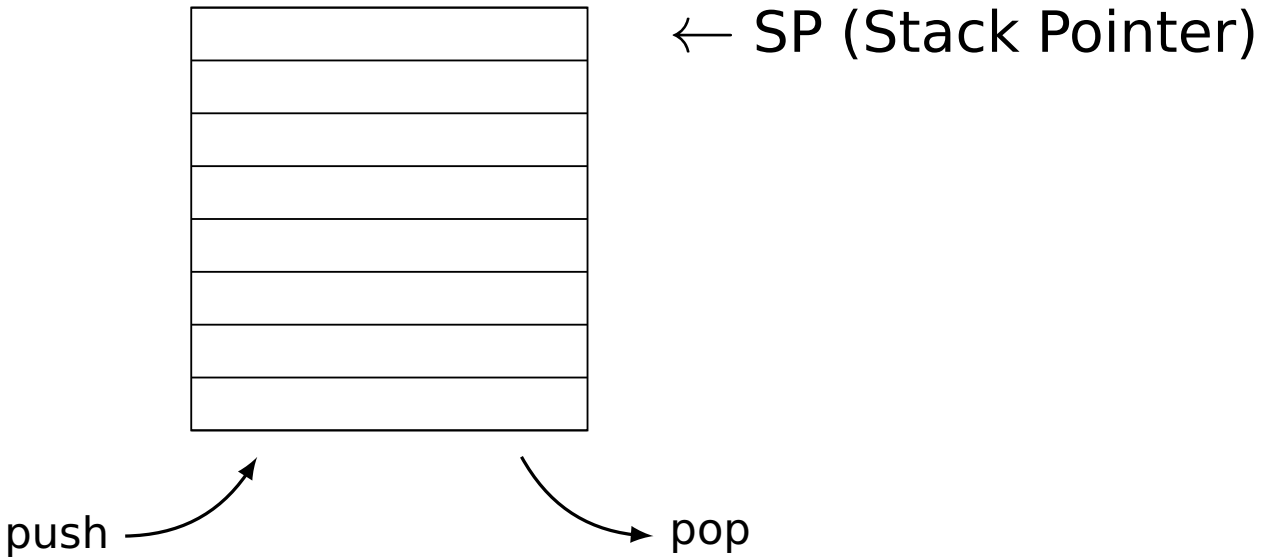
The stack is a memory zone used to store:

- local variables of a function;
- return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- push: insert a data on top of stack
- pop: get back data from stack



Stack

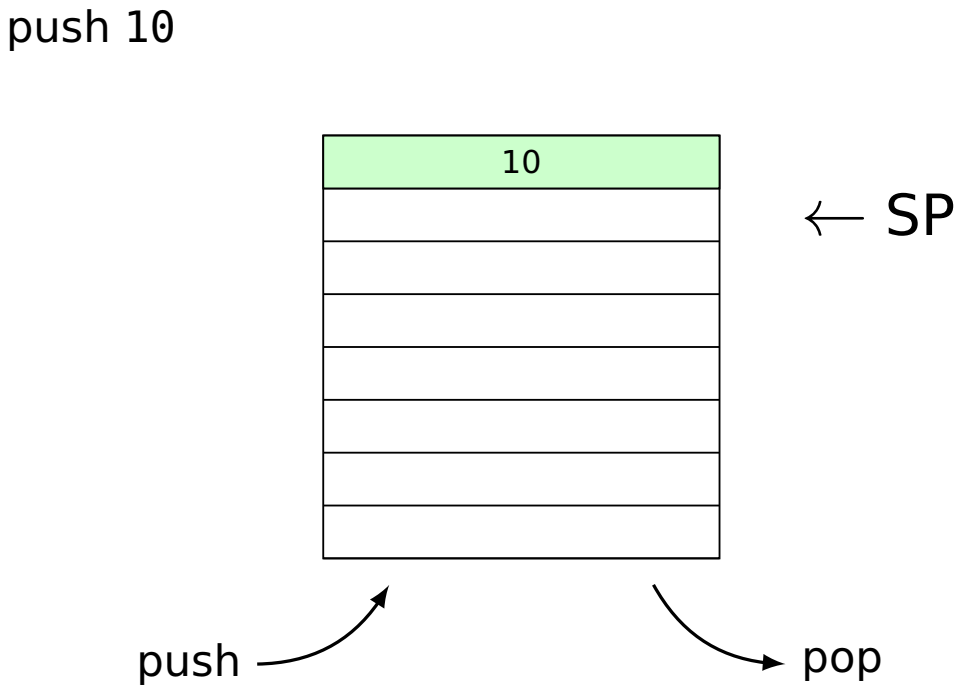
The stack is a memory zone used to store:

- local variables of a function;
- return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- push: insert a data on top of stack
- pop: get back data from stack



Stack

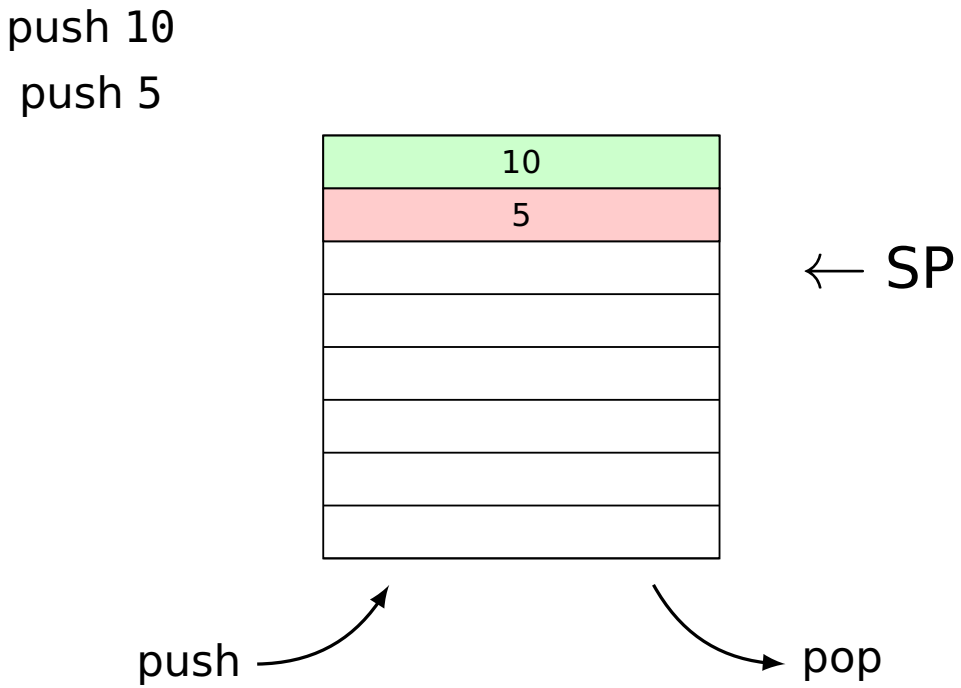
The stack is a memory zone used to store:

- local variables of a function;
- return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- push: insert a data on top of stack
- pop: get back data from stack



Stack

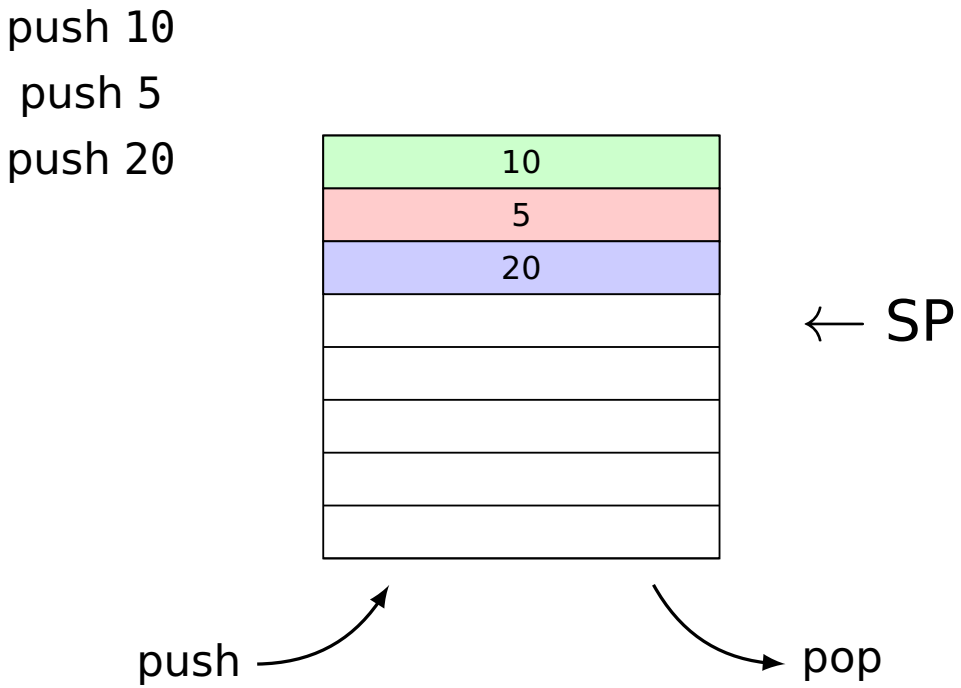
The stack is a memory zone used to store:

- local variables of a function;
- return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- push: insert a data on top of stack
- pop: get back data from stack



Stack

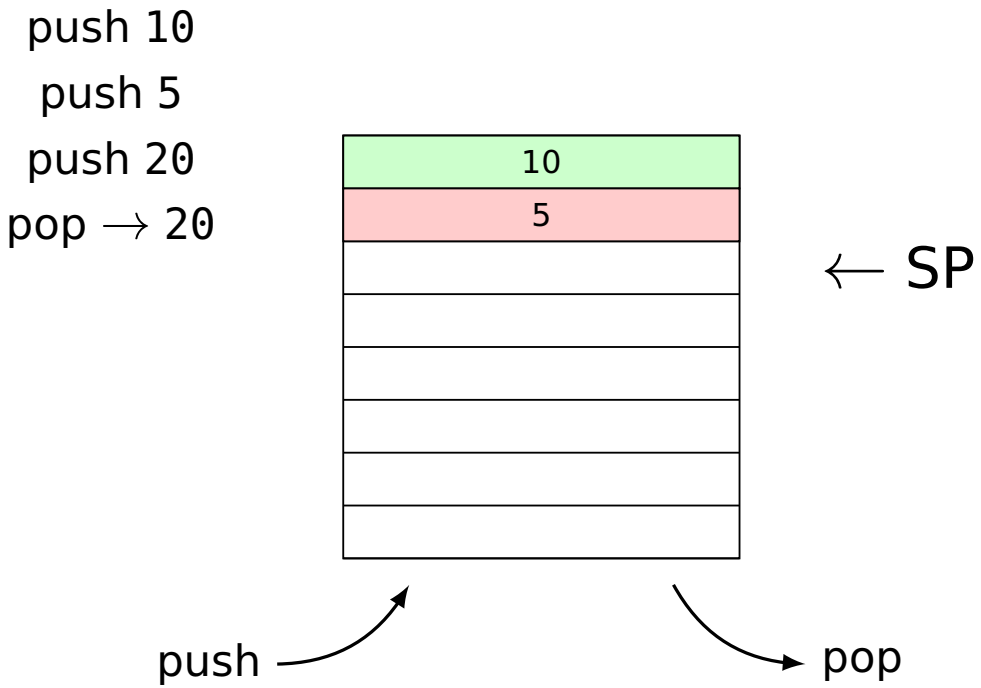
The stack is a memory zone used to store:

- local variables of a function;
- return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

- push: insert a data on top of stack
- pop: get back data from stack



Stack

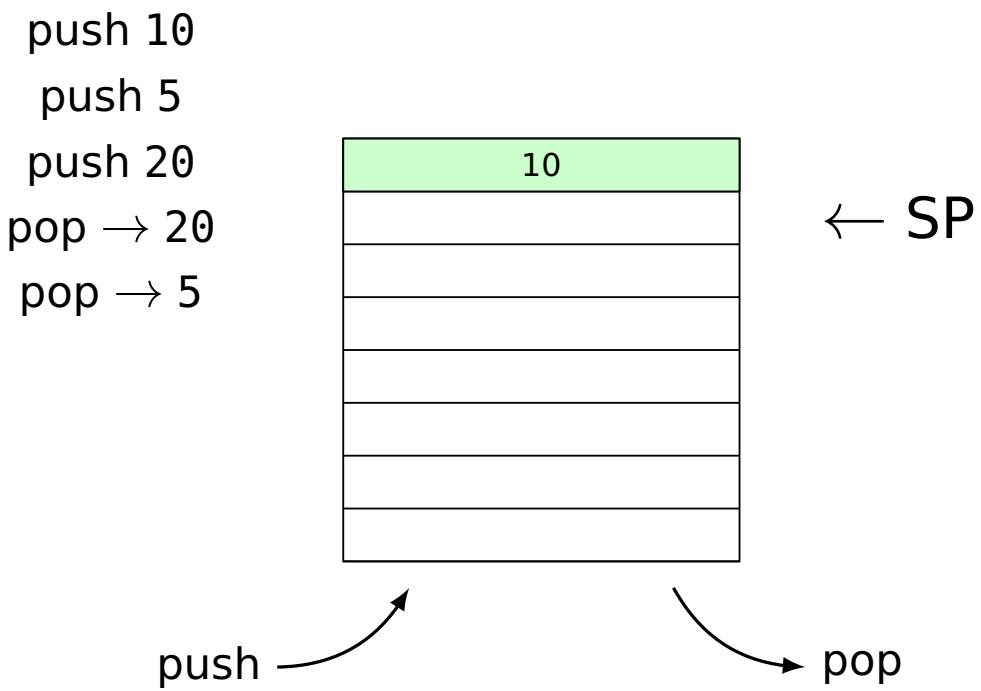
The stack is a memory zone used to store:

- local variables of a function;
- return address when calling a function;

The memory is managed as a *LIFO* (Last In First Out) container.

It basically uses 2 operations:

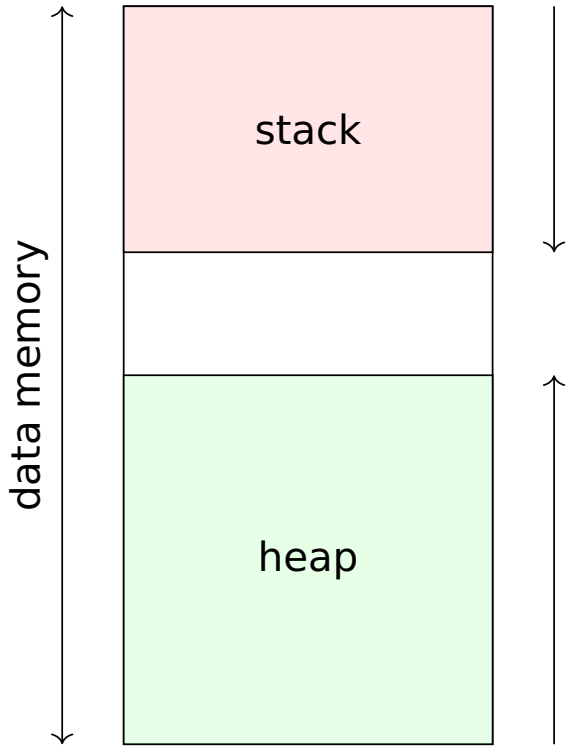
- push: insert a data on top of stack
- pop: get back data from stack



Organisation mémoire

Dans la pratique, la mémoire (RAM) utilisable est scindée en 2 zones:

- le tas (*heap*) qui grandit des adresses basses vers le haut
- la pile (*stack*) qui grandit des adresses hautes vers le bas



Mémoire

Ainsi, la pile stocke:

- les adresses des retours de fonctions
- les variables locales

Le tas stocke:

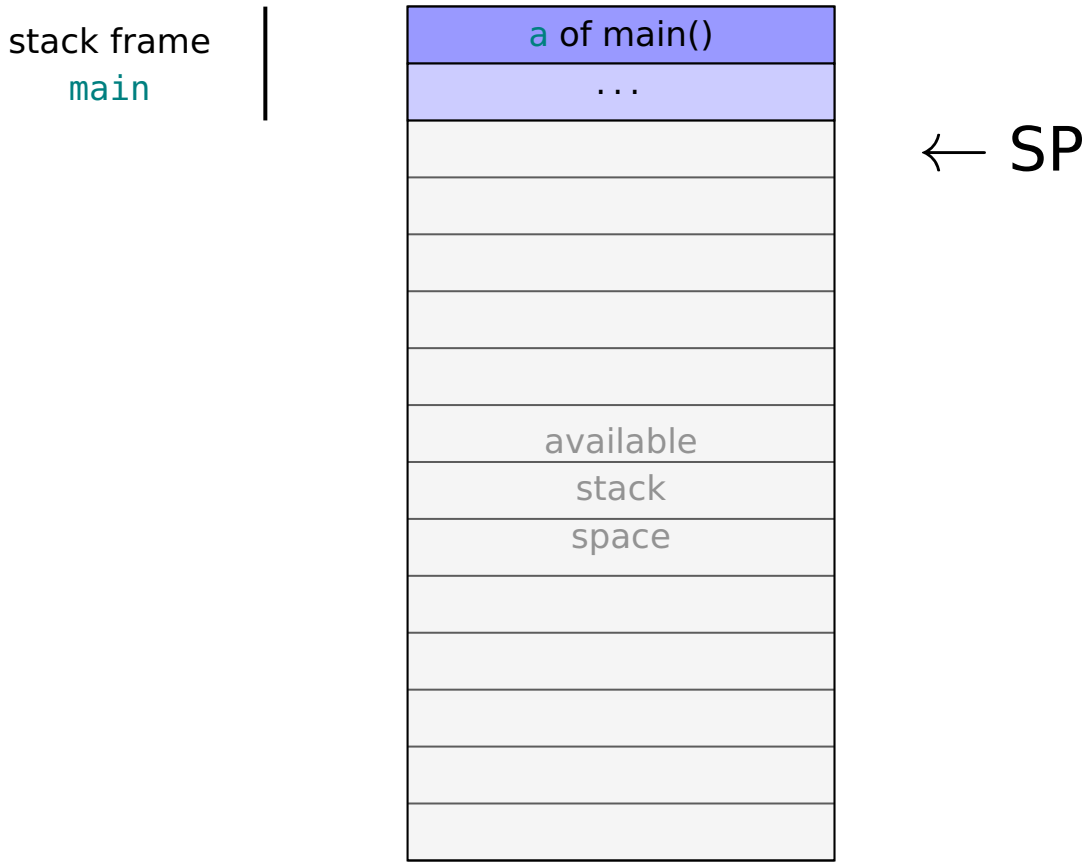
- les variables globales
- les données avec allocation dynamique (`malloc`, `new`). Quelque fois, ces fonctions sont cachées (vecteurs, listes, ...).

On essaie d'éviter que ces 2 zones se superposent :)

Pile

La pile stocke variables locales et adresses de retour (registre ra).Exemple d'appels imbriqués:

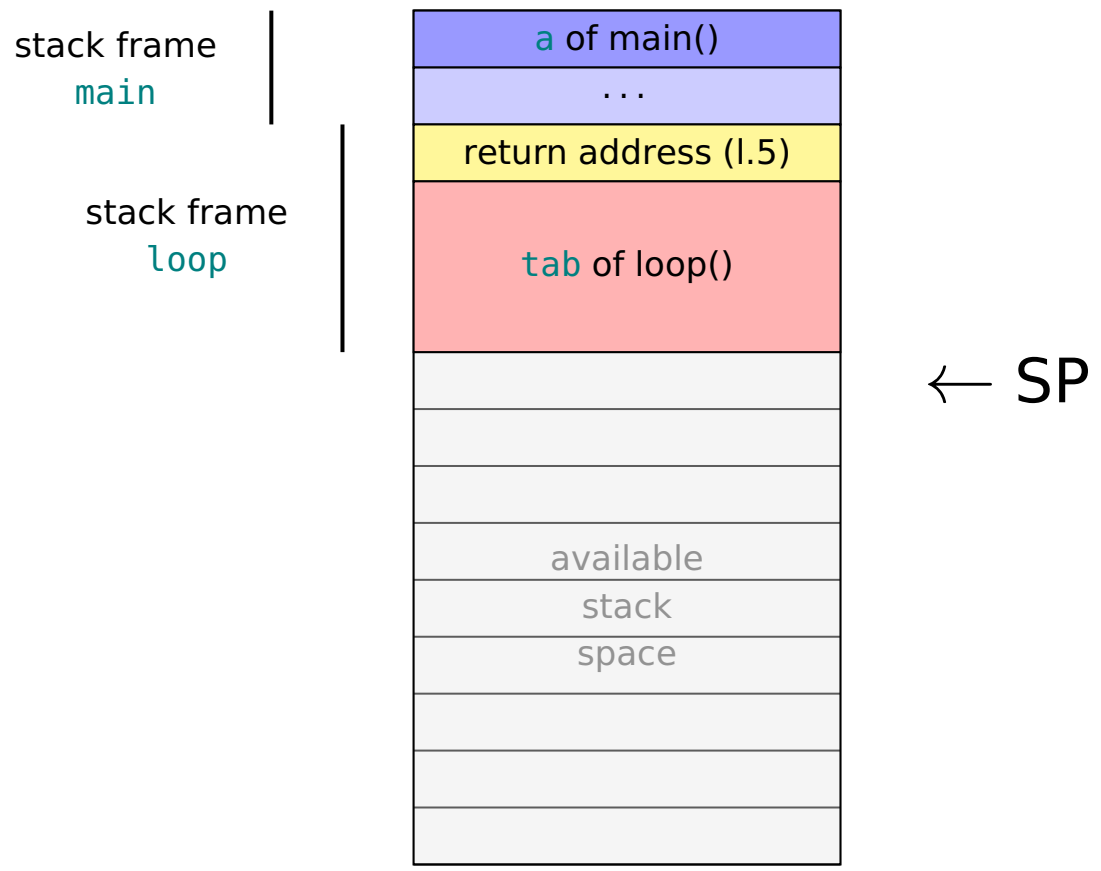
```
void main() {           1
    int a;              2
    while(1) {          3
        loop();         4
    }                   5
}                       6
                        7
void loop() {           8
    int tab[3];         9
    ...                 10
    readSensor();       11
}                       12
                        13
int readSensor() {      14
    int result;         15
    ...                 16
    return result;      17
}                       18
```



Pile

La pile stocke variables locales et adresses de retour (registre ra).Exemple d'appels imbriqués:

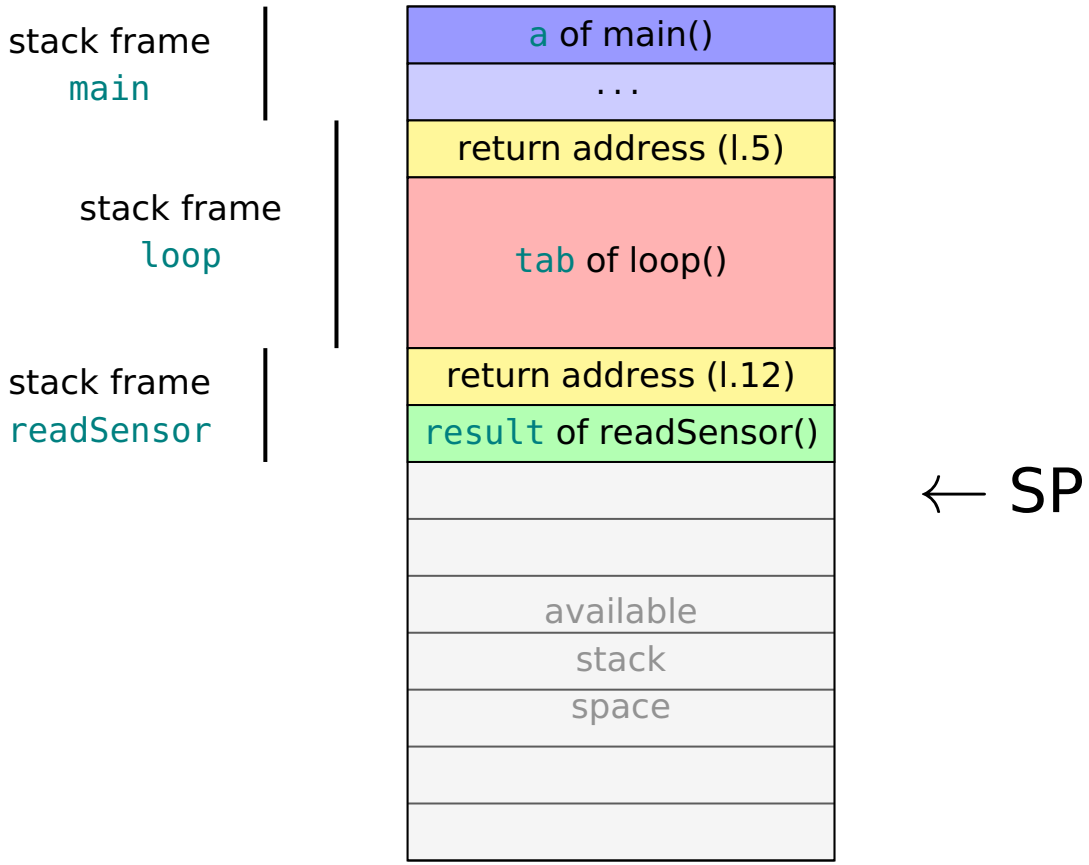
```
void main() {           1
    int a;              2
    while(1) {          3
        loop();         4
    }                   5
}                       6
                        7
void loop() {           8
    int tab[3];         9
    ...                 10
    readSensor();       11
}                       12
                        13
int readSensor() {      14
    int result;         15
    ...                 16
    return result;      17
}                       18
```



Pile

La pile stocke variables locales et adresses de retour (registre ra).Exemple d'appels imbriqués:

```
void main() {           1
    int a;              2
    while(1) {          3
        loop();         4
    }                   5
}                       6
                        7
void loop() {           8
    int tab[3];         9
    ...                 10
    readSensor();       11
}                       12
                        13
int readSensor() {      14
    int result;         15
    ...                 16
    return result;      17
}                       18
```



Stack

En fait, les processeurs modernes dotés d'une architecture RISC gèrent la pile à l'aide d'un *mode d'adressage indirect* et utilisent une mémoire de pile: *stack frame*.

- Il réserve de la mémoire lorsqu'il entre dans une fonction (mise à jour de sp, *i.e.* x2)

```
addi    sp,sp,-16
```

- Il accède ensuite aux variables à l'aide d'un décalage constant dans la fonction:

```
sw  ra,12(sp) ; store ra to [sp+12]
lw  a4, 8(sp) ; load a4 from [sp+8]
```

CENTRALE NANTES

Exemple

La fonction en C:

```c
int func()
{
    int a;
    int b;
    int c;
    ...
    a++;
}
```

peut donner en assembleur:

```
func:
    ; prologue: 3 entiers de
        32 bits
    addi sp, sp, -16
    ...
    lw t0,12(sp)
    addi t0,t0,1
    lw t0,12(sp)
    ..
    ;epilogue
    addi sp, sp, 16
```

Note

L'ABI impose un multiple de 16 octets pour l'alignement sur la pile (`Tag_RISCV_stack_align` dans la doc de l'ABI.)

Appel de fonction imbriqués.. la suite!

Il suffit de sauver la valeur de ra directement sur la pile:

```
func1:
  addi sp,sp,-16 #réserve de la place sur la pile
  #debut code func1
  sw ra,4(sp)   #sauve ra sur le pile
  jal ra,func2
  lw ra,4(sp)   #restaure ra de la pile
  #fin code func1
  addi sp,sp,16
  ret #ok, ra est cohérent!
```

Dans la pratique, on va plutôt sauvegarder systématiquement ra en debut de fonction (et le restaurer en fin de fonction) pour toutes les fonctions non feuilles.

Exercice pile

La suite de Fibonacci u_n est définie par:

$$\begin{cases} u_0 = 1 \\ u_1 = 1 \\ \forall n > 1, u_n = u_{n-1} + u_{n-2} \end{cases}$$

Implementez une version récursive de la suite: `int fibo(int n);`
On ne se soucie pas des optimisations ici (double appel récursif)!

Exercice bilan

- Implémenter une multiplication (sur RV32I) en 16x16 bits
  `uint32_t mult(uint16_t a, uint16_t b);`
- Implémenter une multiplication (sur RV32I) en 32x32 bits
  `uint64_t mult(uint32_t a, uint32_t b);`

CENTRALE NANTES

Exercice bilan

Implémenter la fonction suivante un assembleur, sans optimiser la variable locale (qui reste sur la pile):

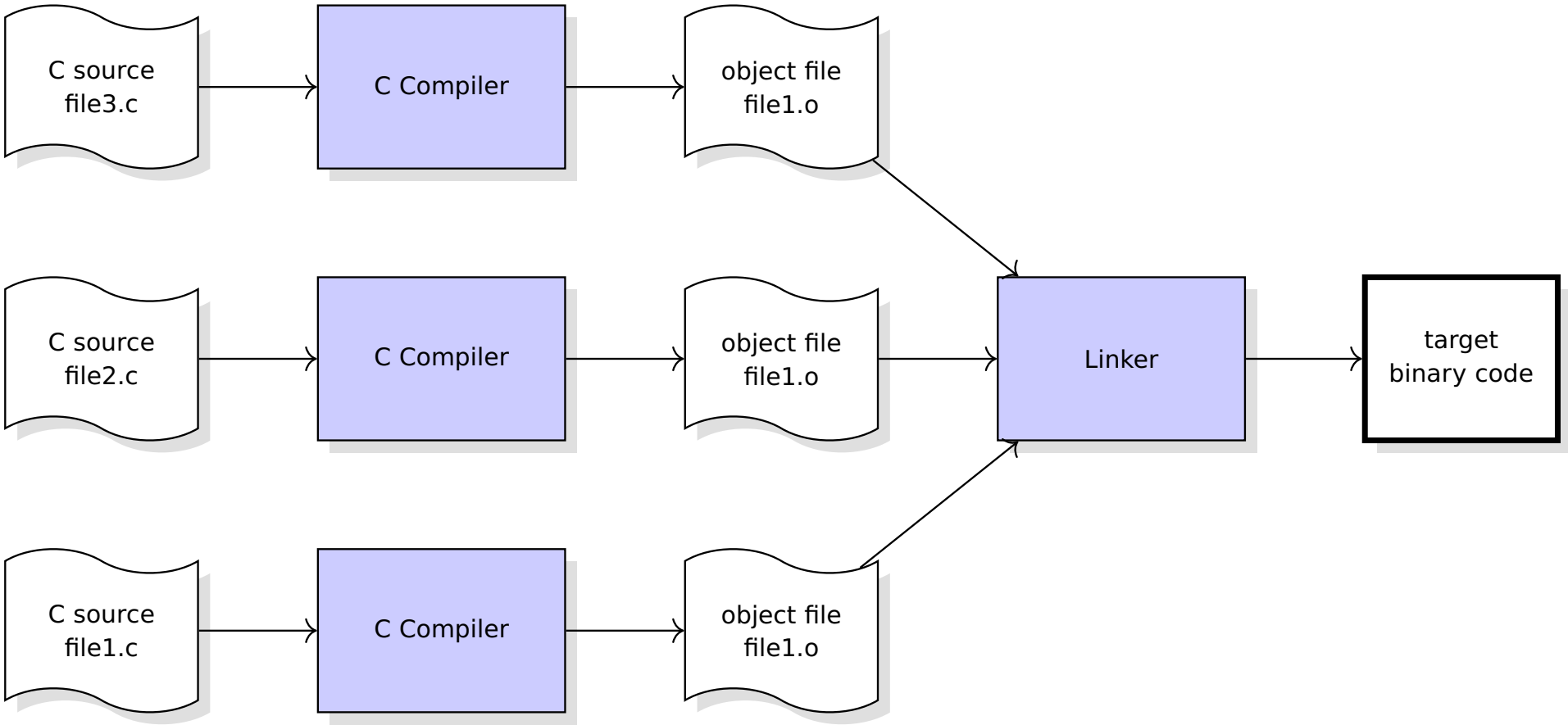
```
typedef struct {
    unsigned int priority;
    unsigned int state;
} taskEntry;

int initTask(taskEntry *entry)
{
    static int nbInit = 0;
    nbInit++;
    unsigned int priority;
    priority = computePrio(entry); //fonction à appeler.
    entry->priority = priority;
    entry->state = 0;
}
```

Contents

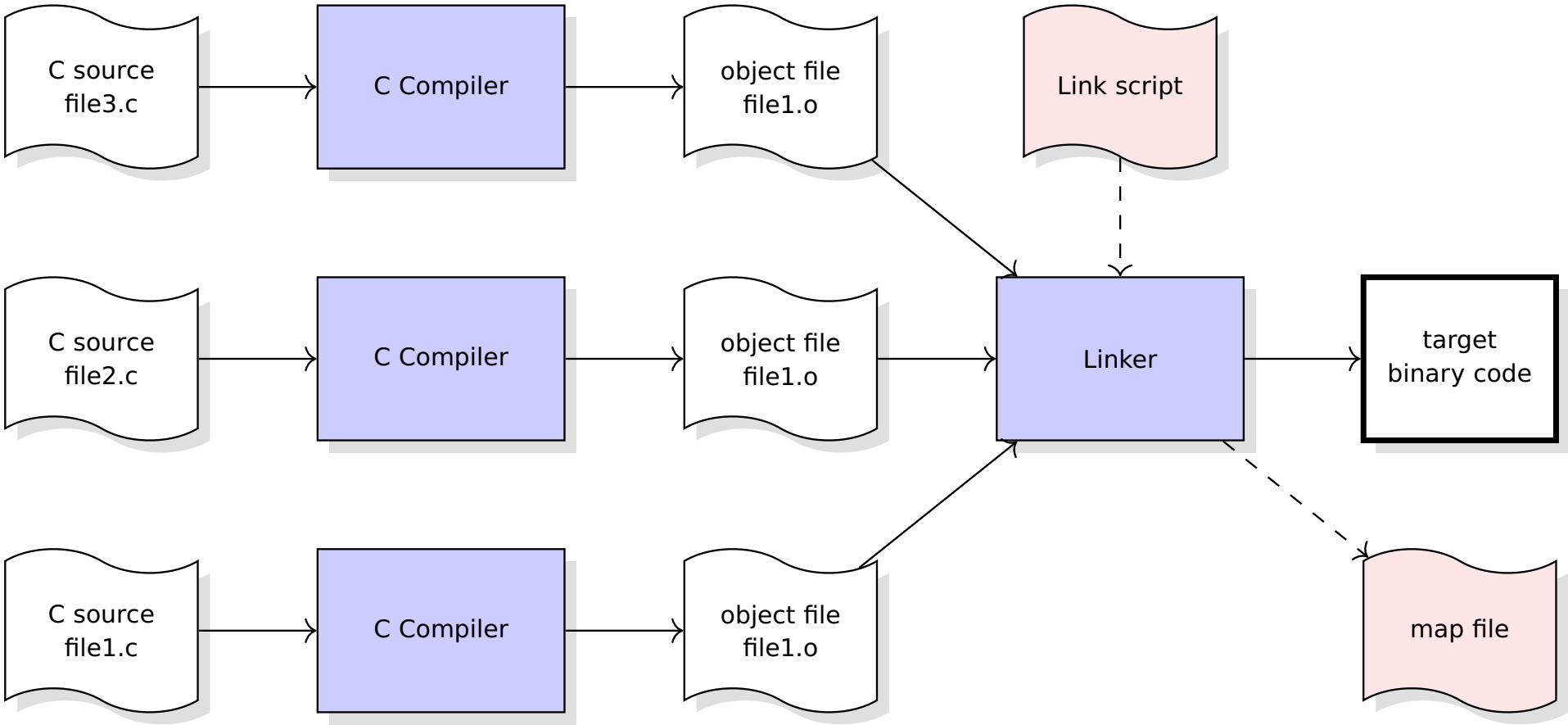
Chaine de compilation C

Compilation chain: getting the binary code from the C source code.



Chaine de compilation C

Compilation chain: getting the binary code from the C source code.



Chaine de compilation

- l'affectation des emplacements mémoire est fait dans l'étage d'édition des liens. Les règles de placement sont définies à travers le *link script*.
- le fichier `.map` permet de visualiser a posteriori l'emplacement choisi par le compilateur. Ça peut servir pour la mise au point.

Link script GCC

Le *link script* définit au moins une partie `MEMORY` et une partie `SECTIONS`.

- `MEMORY` définit les tailles, emplacements et droits d'accès de la mémoire.
- `SECTIONS` définit des sections du C qui sont mappées sur les zones mémoires définies dans `MEMORY`.

Les sections du C les plus communes sont:

- `.text`: le code
- `.data`: les données initialisées
- `.bss`: les données non-initialisées (init à 0).

On en trouve beaucoup d'autre en fonction des langages: `.ctors` (constructeurs en C++), `.dtors`, `.rodata` (constantes), `...`

CENTRALE NANTES

Link script GCC

On peut définir explicitement la section pour placer les variables ou le code:

```
static uint8_t stack[__STACK_SIZE]
    __attribute__ ((aligned(16), used, section(".stack")));

__attribute__((section(".truc")))
    uint32_t fctTruc(uint32_t value) {
}
```

Très utile pour placer les vecteurs d'interruptions notamment!

Link script GCC

Lors de l'édition des liens, les sections sont placées de manière incrémentales dans les sections les concernant. On peut agir sur le placement (alignement) et on peut récupérer l'adresse au debut/fin de section.
Ceci permet de définir des constantes/variables qui sont utilisables dans la partie en C.
Voir fichiers `startup.c` et `linkScript.ld` dans les TPs.

Démarrage de l'application

Au démarrage, avant le `main()`, il est nécessaire:

- initialiser le cœur (pile, *global pointer*)
- initialiser la section `.bss`
- initialiser la section `.data`
- appeler les constructeurs des objets globaux
- initialiser la libc
- appeler le `main()`.

Implémentation matérielle...

on dispose:

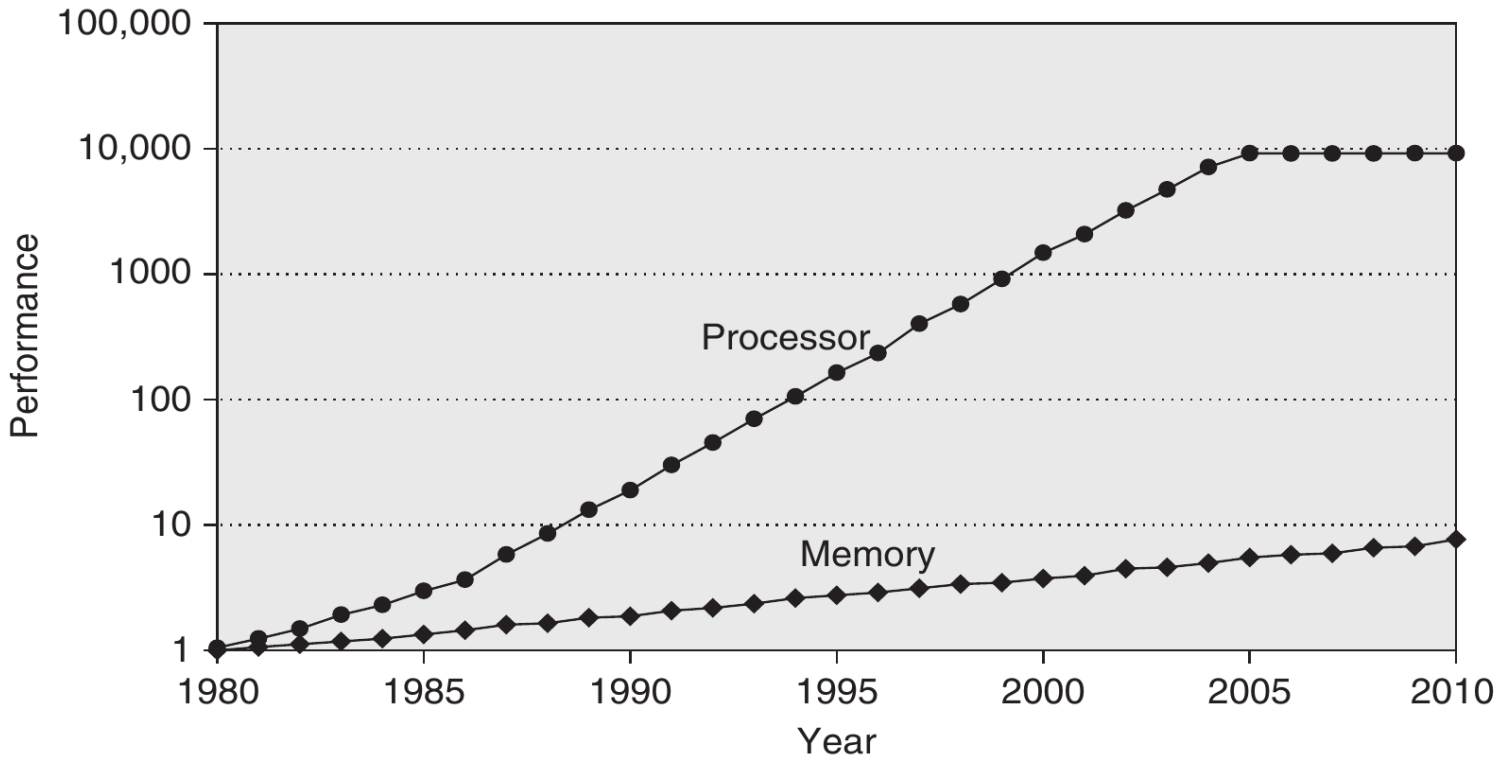
- d'une ALU: faire les opérations
- de la mémoire: programme/données séparés (architecture Harvard) ou non (architecture Von Neumann);
- d'un banc de registre (mémoire rapide, 2 lectures/1 écriture);
- d'un registre de compteur programme et de son additionneur.

Il ne reste *plus qu'à* connecter tout ça ensemble!!!

Contents

Contrainte technologique

la vitesse des processeurs augmente beaucoup plus vite que la vitesse d'accès à la mémoire



Écart de performances entre les latences des requètes mémoires (mono-cœur) et la latence d'accès à la DRAM. source: Computer Architecture A

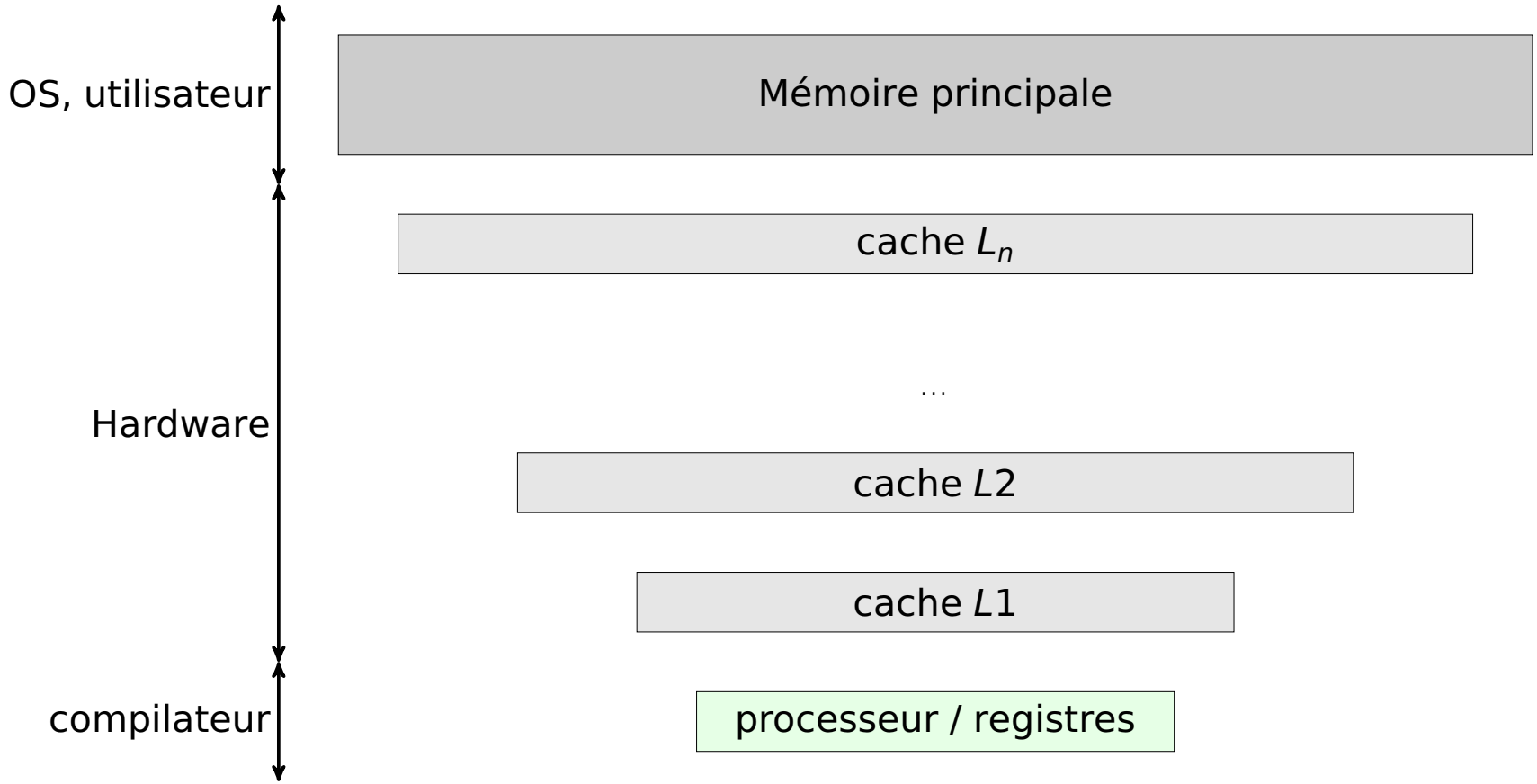
Hiérarchie mémoire

Si on ne fait rien. . . le processeur passerait l'essentiel du temps à attendre les données mémoire.
Autre constat: la mémoire rapide est chère!

- registre: $< 1Ko$ - 0.15 à 0.3 ns
- cache (SRAM): quelques Ko à quelques Mo - 0.5 à 15 ns
- mémoire centrale (DRAM): quelques Mo à quelques Go - 30 à 200ns

Hiérarchie mémoire

Le principe consiste à mettre une partie des données en *mémoire cache*, pour accélérer les accès.



Hiérarchie mémoire

Principe de localité

- localité temporelle: les données/codes qui ont été utilisées ont une forte probabilité d'être réutilisées dans un futur proche;
- localité spatiale: les données/codes qui sont proches ont beaucoup de chance d'être accédées dans un futur proche

La mémoire cache consite donc à

- copier un *bloc* mémoire vers la cache pour profiter de cette localité spatiale/temporelle.

CENTRALE NANTES

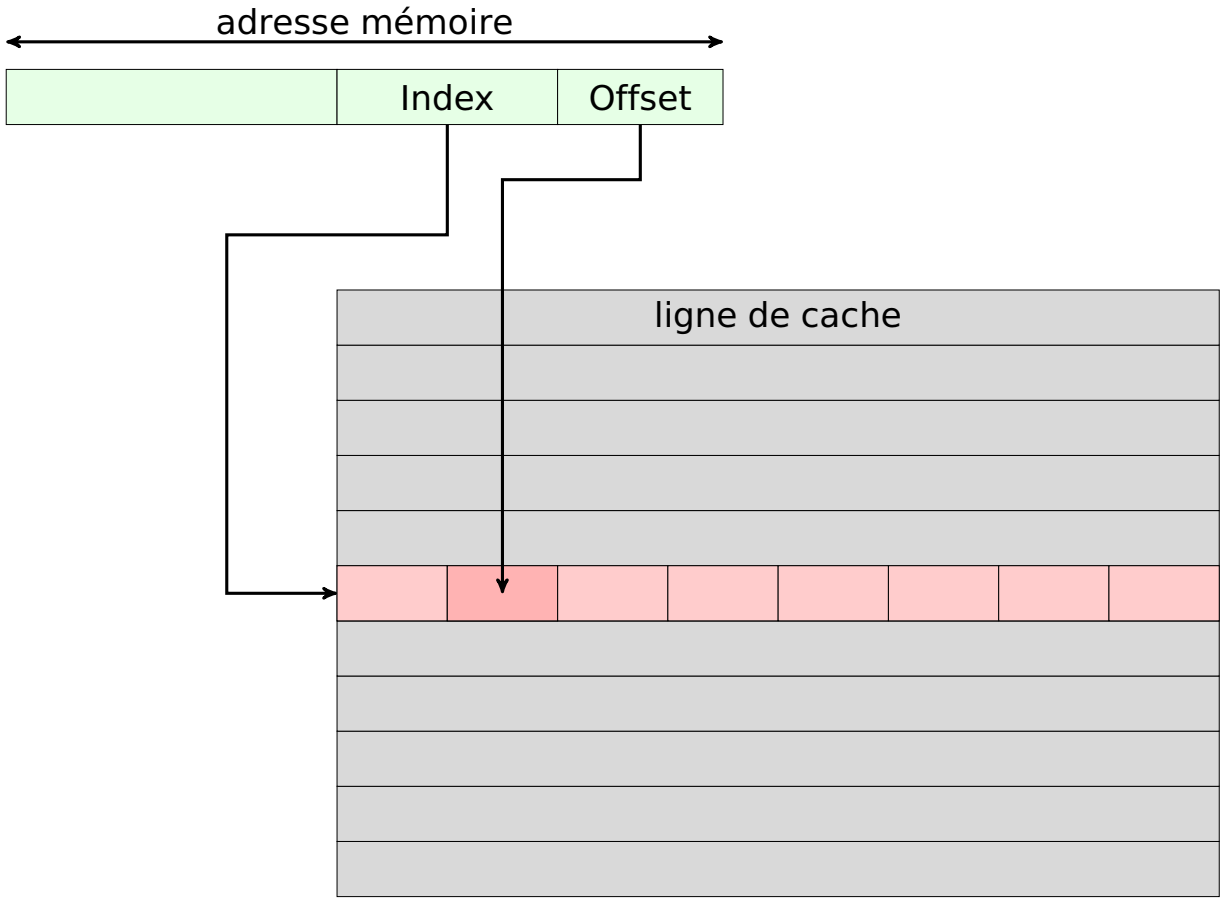
Hiérarchie mémoire

Plusieurs questions:

- Q1 Où placer un bloc dans la cache? *placement*
- Q2 Comment trouver un bloc en cache? *identification*
- Q3 Comment remplacer un block par un autre? *remplacement*
- Q4 Comment faire pour une écriture? *statégie d'écriture*

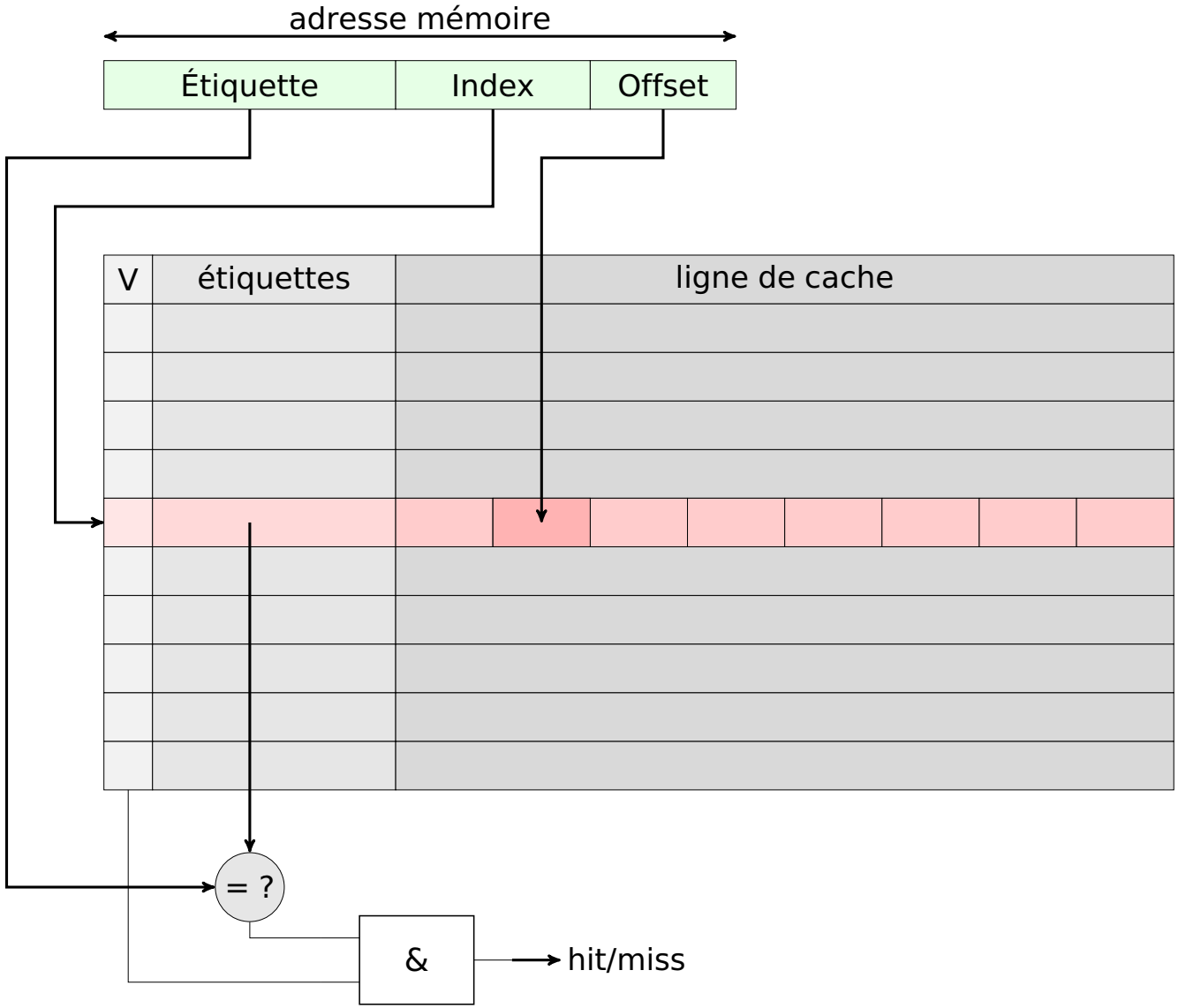
Placement / identification

Principe de base. Accès direct



Placement / identification

Principe de base. Accès direct



Placement / identification

on décompose l'adresse:

- l'*offset* donne le numéro du mot dans la ligne;
- l'*index* donne la ligne de cache;
- l'*étiquette* (*tag*) permet d'identifier si l'accès est ok

Exemple: Quelle est la configuration (taille des étiquettes, index offset) pour un cache de 32Ko, avec des lignes de 64 octets. Quelle quantité de mémoire effective?

Placement / identification

2 cas possibles lors de l'identification:

hit la donnée est dans la cache, elle est utilisée de suivante

miss la donnée n'est pas dans la cache, toute la ligne est récupérée en mémoire et est placée en cache.

Placement / identification

On dispose du code:

```
uint32_t tab[20];
uint32_t sum = 0;
for(int i=0;i<20;i++)
{
    sum += tab[i];
}
```

Les données `tab` et `sum` sont en ram, à partir de l'adresse `0x1234`. On a un cache de 32Ko, lignes de 32 octets.

- comment est rempli le cache, en considérant `sum` optimisé par le compilateur (dans un registre), ou avec placement en mémoire, juste à la suite du tableau.
- quel taux de hit/miss? lors de l'exécution du code?

Placement / identification

Deuxième exemple:

```
uint32_t tab1[32768];
uint32_t tab2[32768];
uint32_t sum = 0;
for(int i=0;i<..;i++)
{
    sum += tab1[i]+tab2[i];
}
```

- quel taux de hit/miss? lors de l'exécution du code?

Placement / identification

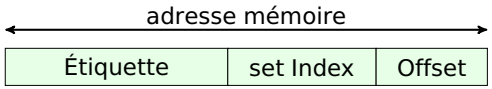
Limitation de l'accès direct

un cache à accès direct *direct mapped* est rigide: il n'y a qu'un seul emplacement possible pour une adresse donnée.

Et si on mettait à la place n caches (plus petits) en parallèle?

Placement / identification

cache associatif à 2 voies:

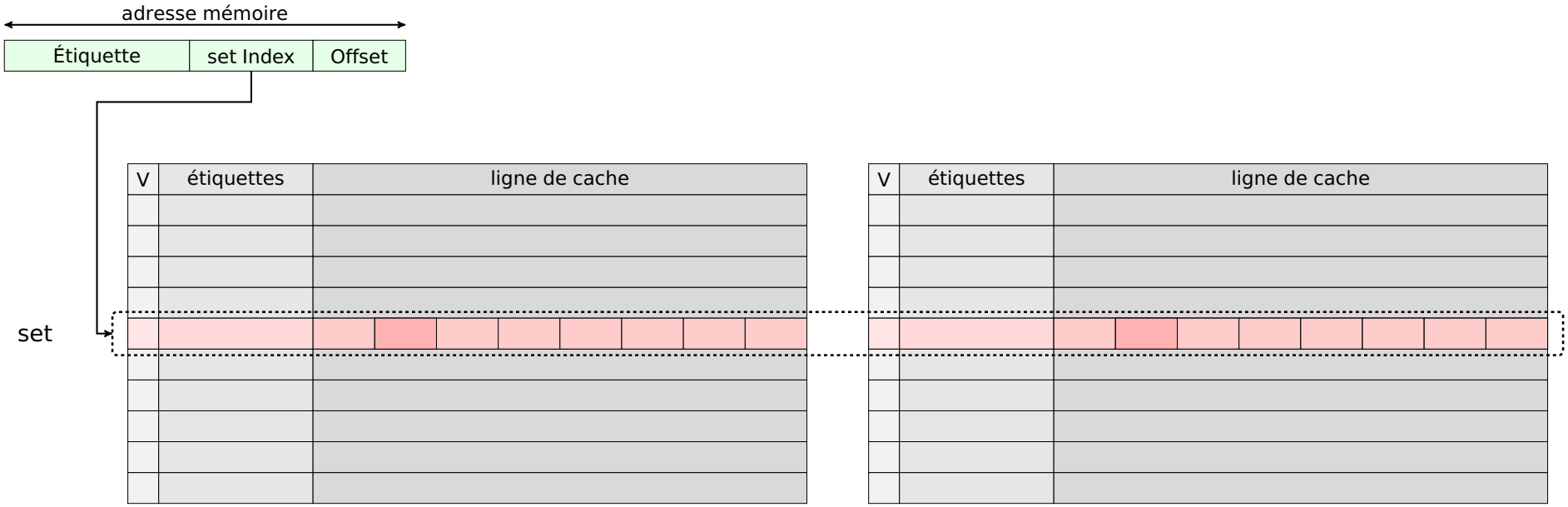


V	étiquettes	ligne de cache

V	étiquettes	ligne de cache

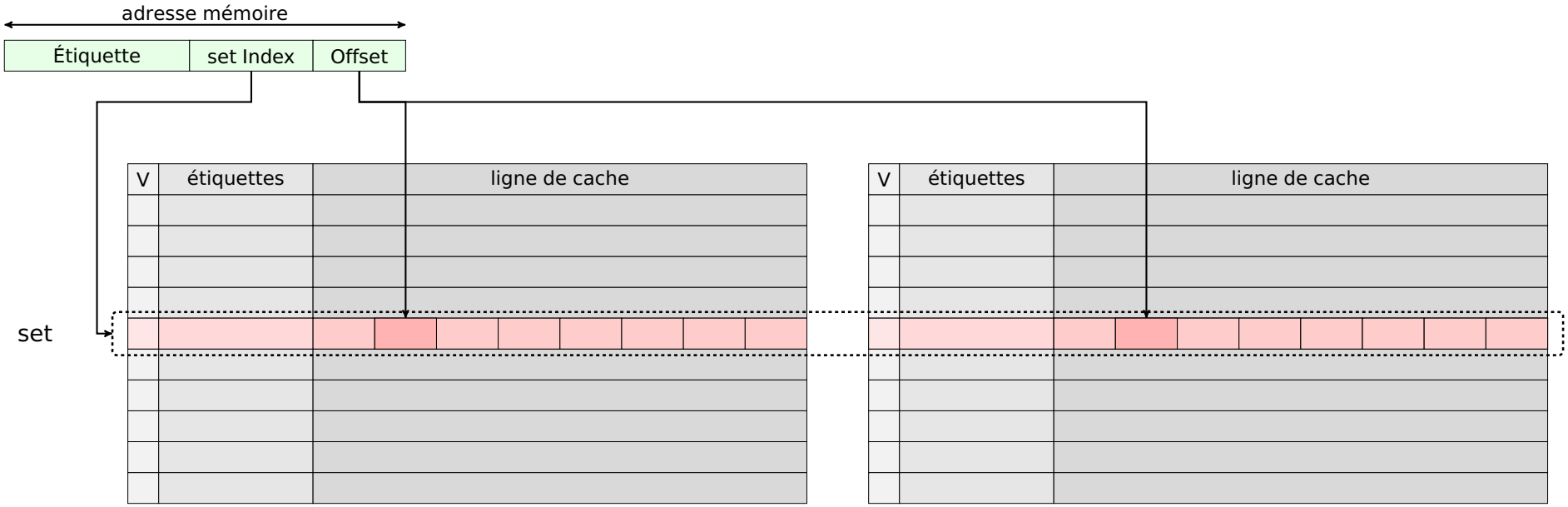
Placement / identification

cache associatif à 2 voies:



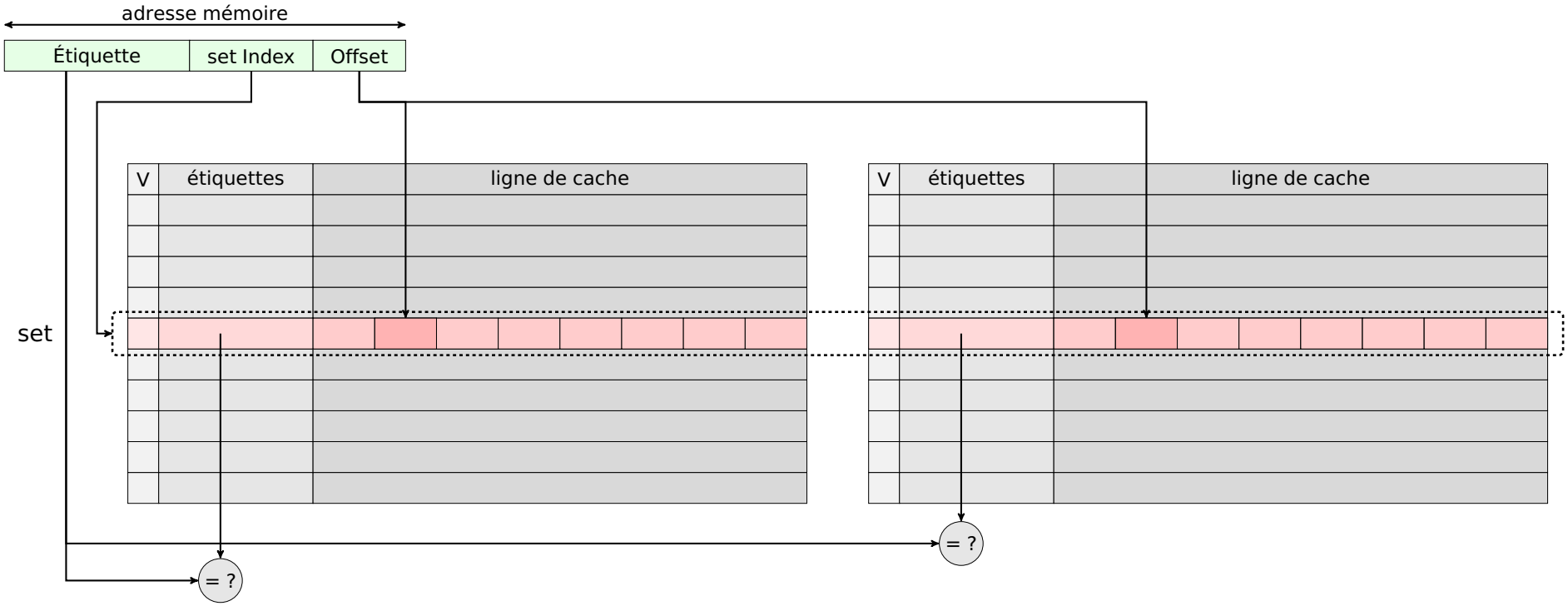
Placement / identification

cache associatif à 2 voies:



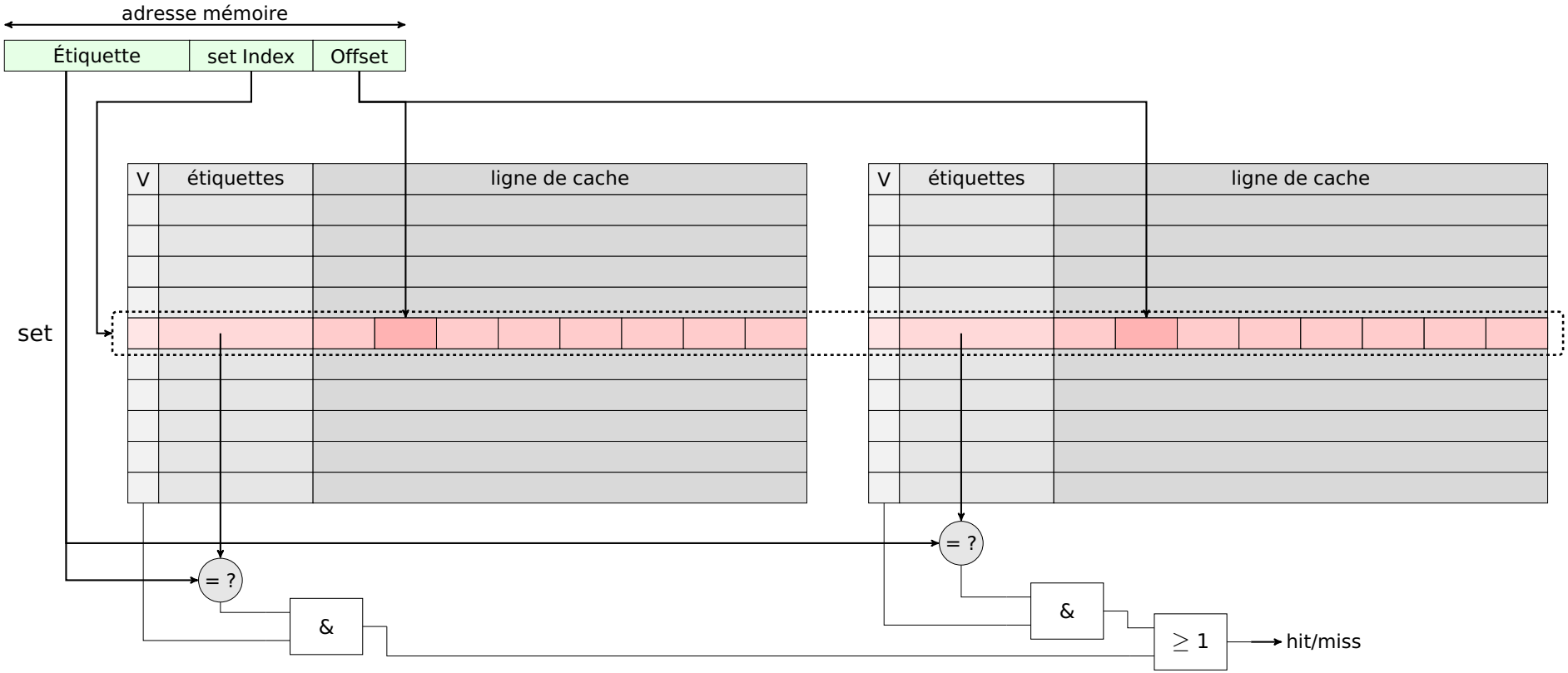
Placement / identification

cache associatif à 2 voies:



Placement / identification

cache associatif à 2 voies:



Placement / identification

On trouve généralement des cache associatifs à 2 voies ou à 4 voies:

- l'index permet maintenant de déterminer un ensemble de lignes (*set*)
- la position de la donnée dans le cache n'est plus figée!

Si le cache est *complètement associatif*, chaque ligne peut aller n'importe où!

Pour n voies:

- Il y a n emplacements pour chaque ligne, donc pour une même taille de cache, il y a n fois moins de lignes!

exemple: cache associatif à 4 voies, 32Ko, 64 octets/ligne. tailles des étiquettes, index, ...

Placement / identification

re-exemple:

```
uint32_t tab1[32768];
uint32_t tab2[32768];
uint32_t sum = 0;
for(int i=0;i<..;i++)
{
    sum += tab1[i]+tab2[i];
}
```

- quel taux de hit/miss? lors de l'exécution du code?

Remplacement

La politique de remplacement avec un cache associatif est plus complexe!
Quelle ligne remplacer en cas de conflit?
On parle de politique de remplacement:

LRU : *Last Recently Used*. La ligne avec l'accès le plus ancien est remplacée (localité temporelle);

FIFO : La ligne la plus ancienne dans le cache est remplacée;

aléatoire : en fait, c'est pseudo-aléatoire...

La *LRU* donne les meilleurs résultats... mais au prix de rajout de matériel important dans le cache pour conserver l'ordre des accès.
La politique *random* n'est pas du tout souhaitée pour les sytèmes critiques!! le résultat n'est pas prédictible.

Stratégie d'écriture

Que faut-il faire dans le cas d'une écriture? 2 stratégies sont possibles:

Write-through La donnée est écrite *à la fois* en cache et dans le niveau supérieur.

Write-back La donnée est écrite *uniquement* en cache. La ligne est annoté (*dirty bit*), pour ré-ecriture quand la ligne sera exclue du cache

Stratégie d’écriture

Compromis:

Write-through

- la donnée est écrite en mémoire en asynchrone (pendant que le programme continue).
- plus de traffic mémoire
- plus simple à implémenter
- cohérence de données avec les niveaux supérieurs (surtout en cas multi-cœur!)
- requis pour les I/Os!!

Write-back

- moins de traffic mémoire si plusieurs mots sont écrits
- lors d’un défaut de cache (*miss*), il faut d’abord réécrire la ligne en mémoire avant de charger la nouvelle.
- consommation moindre (embarqué)

CENTRALE NANTES

Stratégie d'écriture

Et si la donnée à écrire n'est pas dans le cache?

Write allocate La ligne est allouée en cache lors du *write miss*. C'est le comportement *naturel*

No-write allocate la ligne est modifiée directement en mémoire

un simulateur en ligne de cache *facile à utiliser*:
`https://courses.cs.washington.edu/courses/cse351/cachesim/`

Et dans les systèmes embarqués?

- le cache améliore énormément les performances moyennes. . . mais le *pire cas* est très difficile à prévoir!
- dans certain cas, on préfère *déasctiver* le cache pour privilégier la *prédictibilité* temporelle de l'exécution
- souvent un cache d'instruction uniquement. C'est assez prévisible!

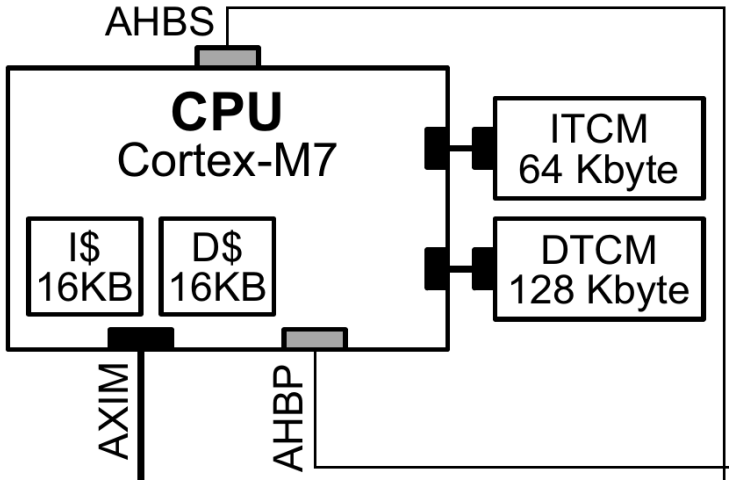
Et dans les systèmes embarqués?

Pour les données, on remplace le *cache* par une mémoire *scratchpad*:

- ce n'est plus le matériel qui gère cette zone mémoire
- elle est à la charge du programmeur qui doit gérer sa zone mémoire tampon.

Exemple: STM32H745/55/47/57xx devices

C'est un STM32 avec un Cortex-M7 et un cortex-M4. Zoom sur le M7 (*ref manual*):



- `I$` et `D$` sont les caches d'instructions/données
- `ITCM` et `DTCM` sont les *data and instruction tightly coupled RAMs*, qui permettent des accès à la vitesse du processeur (*0 wait state*).

Exemple: STM32H745/55/47/57xx devices

sur Cortex-M7, les caches sont configurables par le fondeur (taille), avec les caractéristiques:

- cache de donnée associatif 4 voies, lignes de 32 octets. Configuration *Write-Through* par zones.
- cache d'instruction associatif 2 voies, lignes de 32 octets
- chargement d'une ligne *critical word first*, et non bloquant.
- remplacement? *If all the cache lines in a set are valid, to allocate a different address to the cache, the cache controller must evict a line from the cache.*. Ça laisse rêveur!

source: Technical Reference manual ARM Cortex-M7 processor

CENTRALE NANTES

Contents

Principe de base

Au lieu de considérer l'exécution d'une instruction comme une étape indivisible, on découpe l'instruction en plusieurs *micro étapes*. Exemple:

Fetch Récupérer le code de l'instruction en mémoire, mise à jour du compteur programme (`PC += 4`)

Decode Décoder l'instructions (registres utilisés, fonction ALU, immédiats, extensions de signe), lecture des registres.

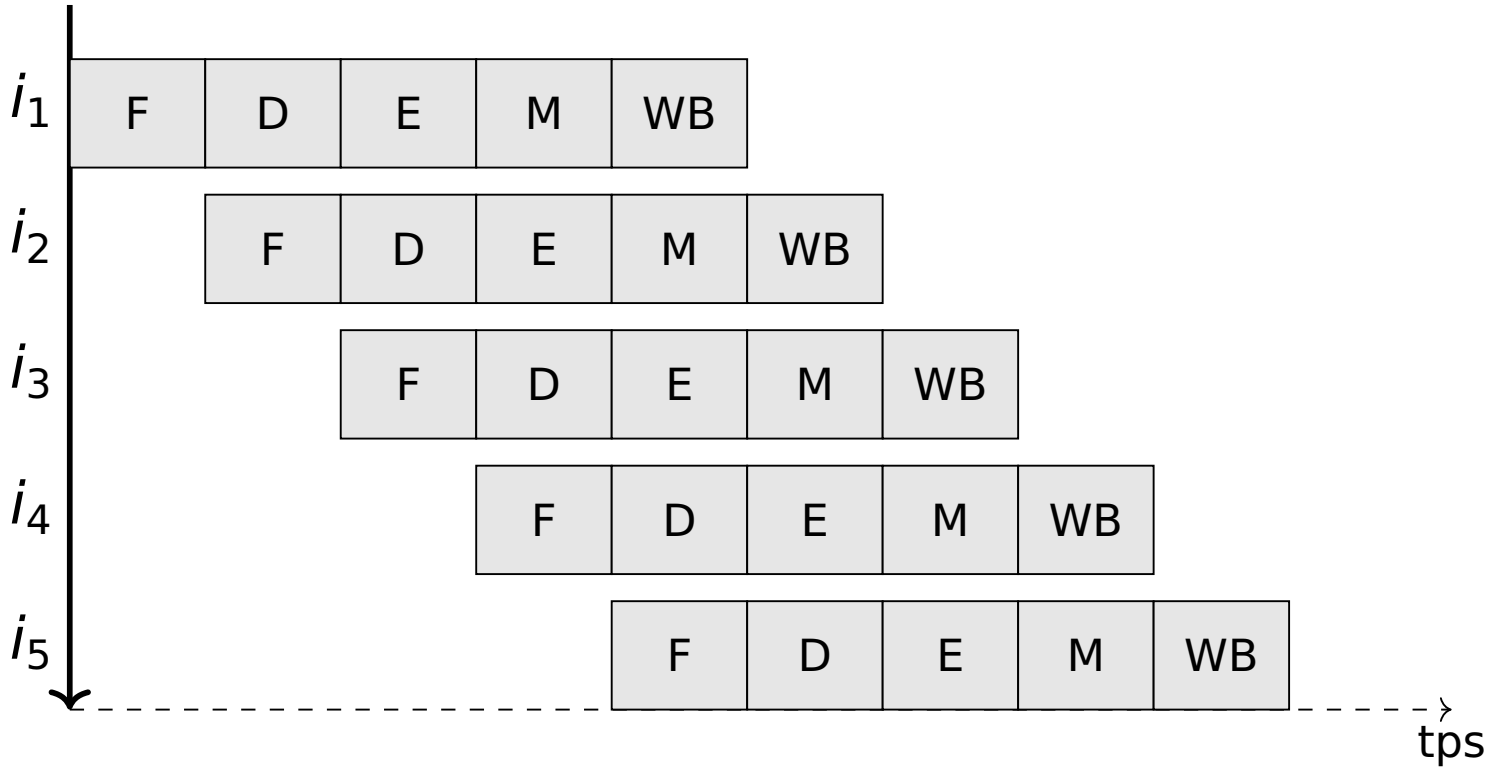
Execute Exécuter l'instruction (registre/registre ou registre/immediat), calculer les adresses (offset+base)

Mem Faire un accès mémoire (approche Load/Store):

Write Back Écrire le résultat (registre destination) des instructions registre/registre et des *load*s.

Principe de base

Le *pipeline* est une technique d'implémentation où l'exécution de plusieurs instructions se recouvrent.



Exemple d'exécution d'instructios avec un pipeline de 5 étages

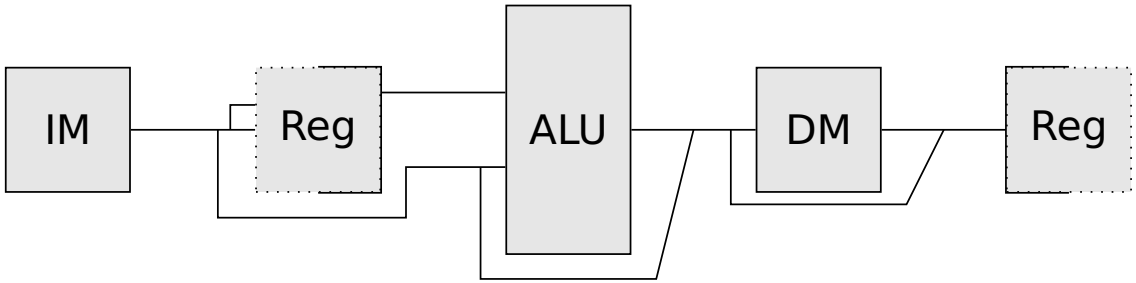
Principe de base

on parle alors de:

- *débit*: nombre d'instruction qui sort par unité de temps - ou de son inverse, le nombre de cycle par instruction (*CPI*);
- *latence*: nombre de cycle pour l'exécution complète de l'instructions.

Mise en œuvre

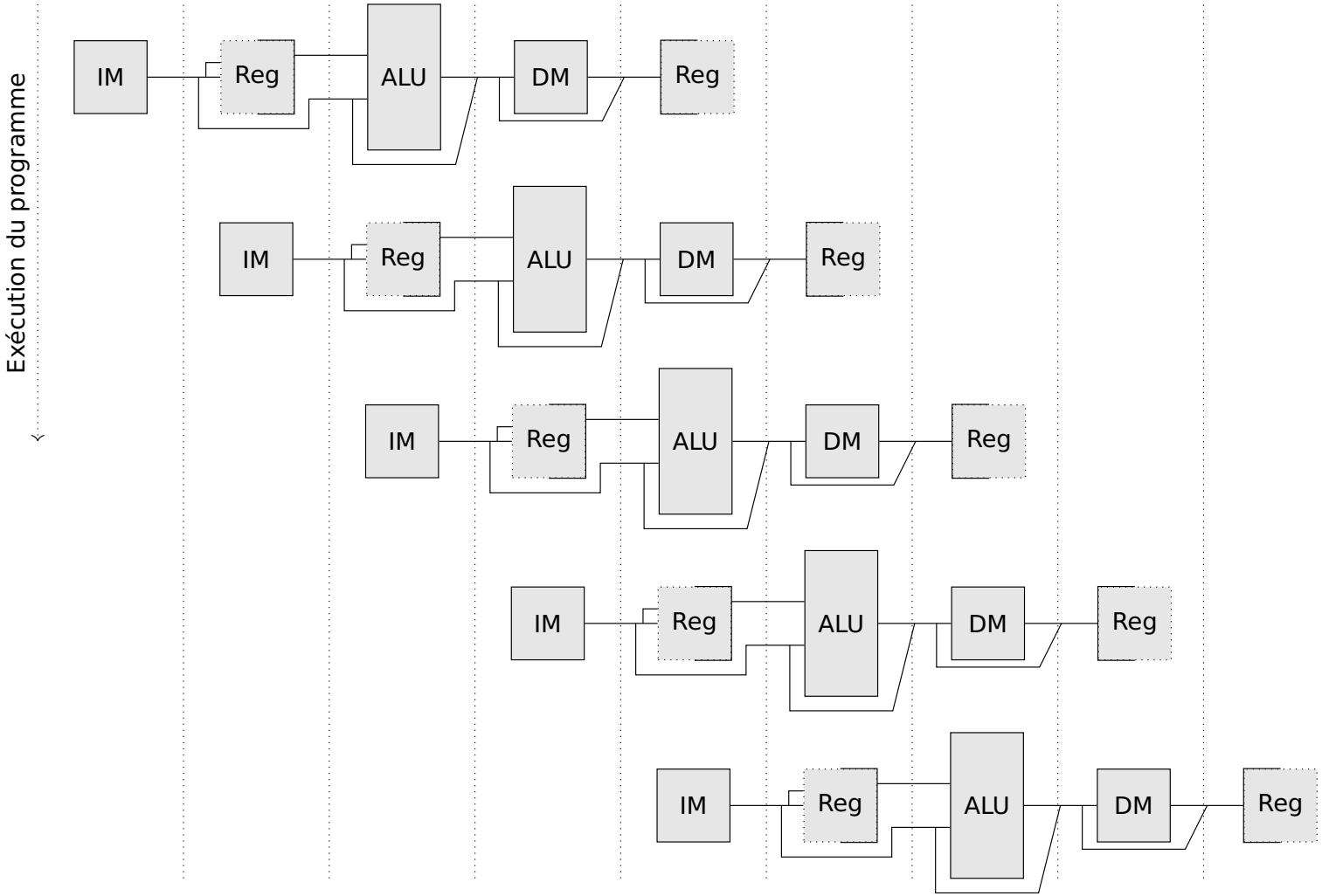
Si on s'intéresse au *flot de données* d'une instruction:



- `IM`: Instruction Memory
- `DM`: Data Memory
- Reg: banc de registres (lecture/écriture)

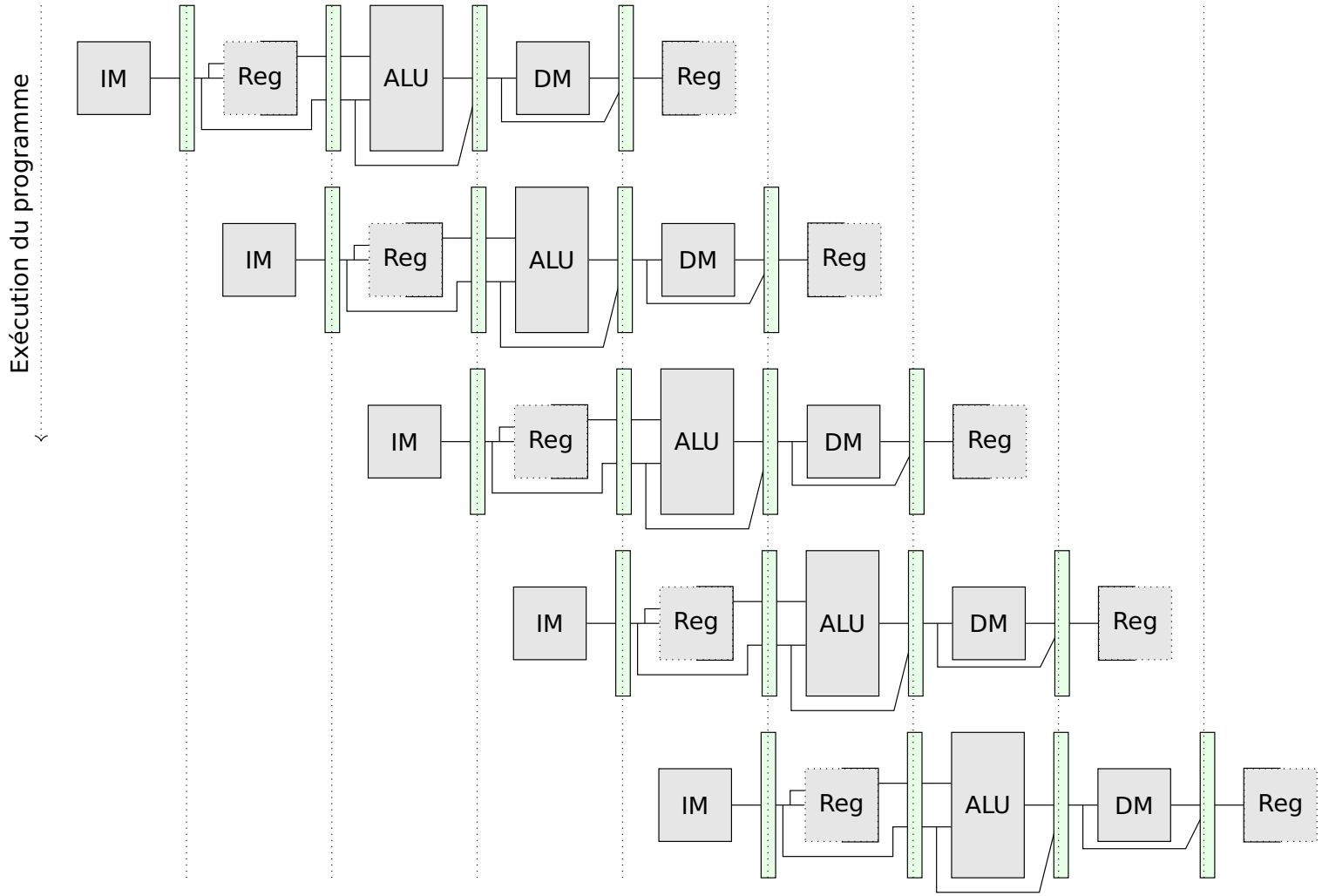
Mise en œuvre

Avec le pipeline, ceci devient alors:

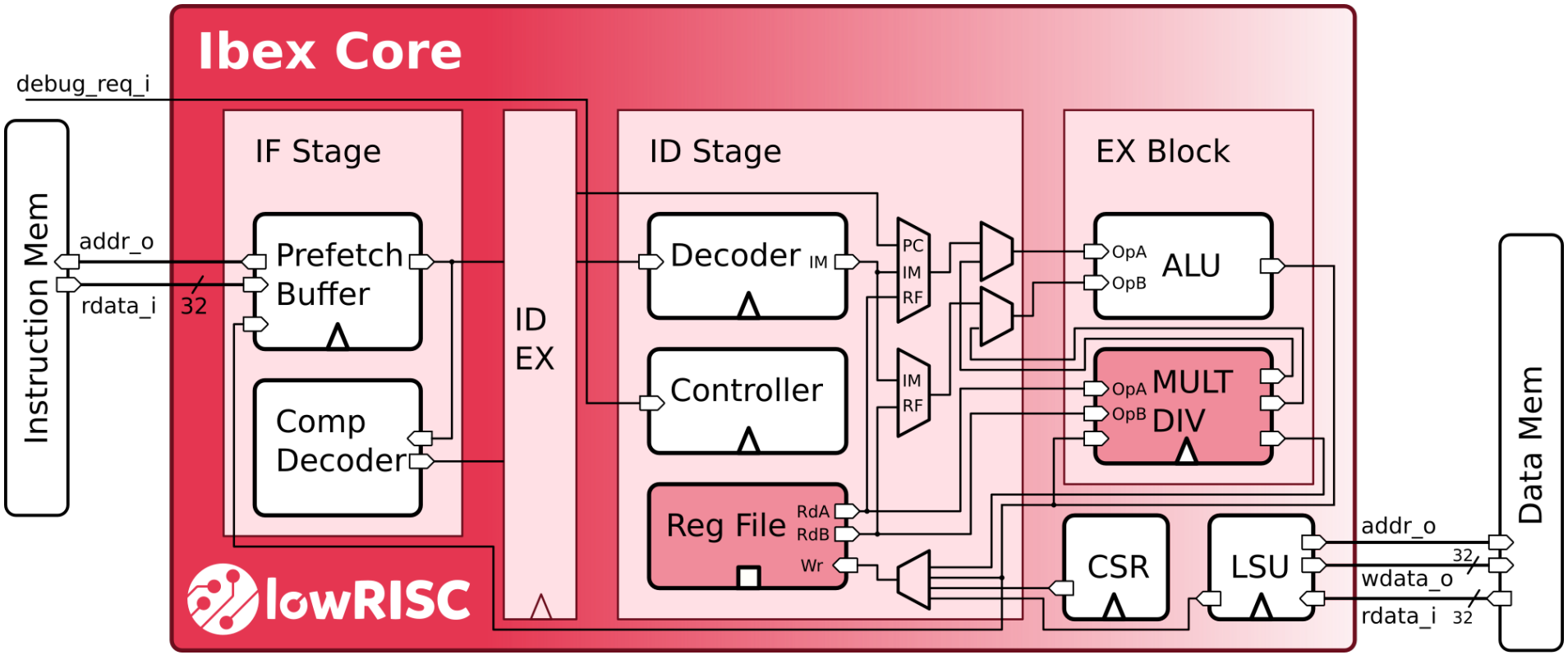


Mise en œuvre

On utilise alors des *registres de pipeline* pour mémoriser l'état des données entre chaque étage:

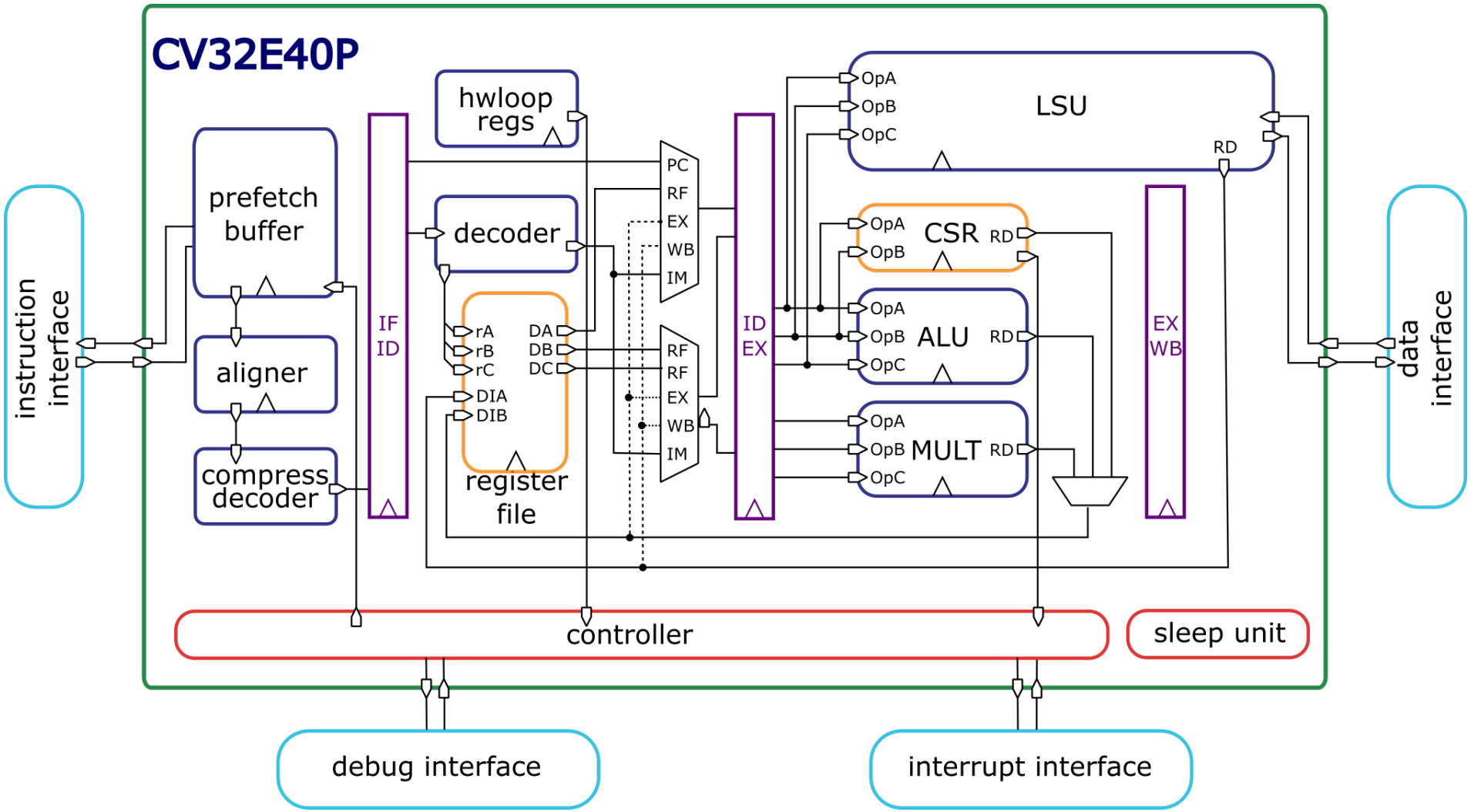


Exemple d'architecture RISC-V à 2 étages



source: `https://ibex-core.readthedocs.io/en/latest/index.html/`

Exemple d'architecture RISC-V à 4 étages



source: `https://cv32e40p.readthedocs.io/en/latest/intro/`

Exemple 1

Soit un pipeline de 5 étages (FDEMR). Quel est la latence et les CPI de la séquence suivante:

```
add  x5, x3, x1
or   x4, x2, x8
sub  x6, x7, x9
add x10, x11, x12
sub x13, x1, x2
```

- Si on considère que la version non pipelinée exécute une instruction en 3 cycles, à partir de quelle longueur de séquence l'utilisation du pipeline sera rentable? Quel sera la limite de l'accélération dûe au pipeline?

(les instructions sont toutes indépendantes les unes des autres...)

Exemple 2

On considère un processeur de fréquence 100MHz (temps de cycle = 10ns).
Les instructions arithmétiques et les branchements requièrent 4 cycles, les instructions mémoire 5 cycles. Leurs fréquences relatives sont respectivement de 40%, 20% et 40%.

- Quel est le gain apporté par l'utilisation d'un pipeline (pas d'aléas pour le moment). On néglige la latence initiale.

Aléas

Dans la pratique, certaines situations conduisent à ne pas pouvoir superposer l'exécution des instructions de manière optimale, et on réduit donc le débit. Ces *aléas* sont définis en 3 classes:

- aléa structurel
- aléa de donnée
- aléa de contrôle

Dans ces cas, on va être obligé de bloquer le pipeline. On parle alors de *bulle* (*stall*, *bubble*).

Aléa structurel

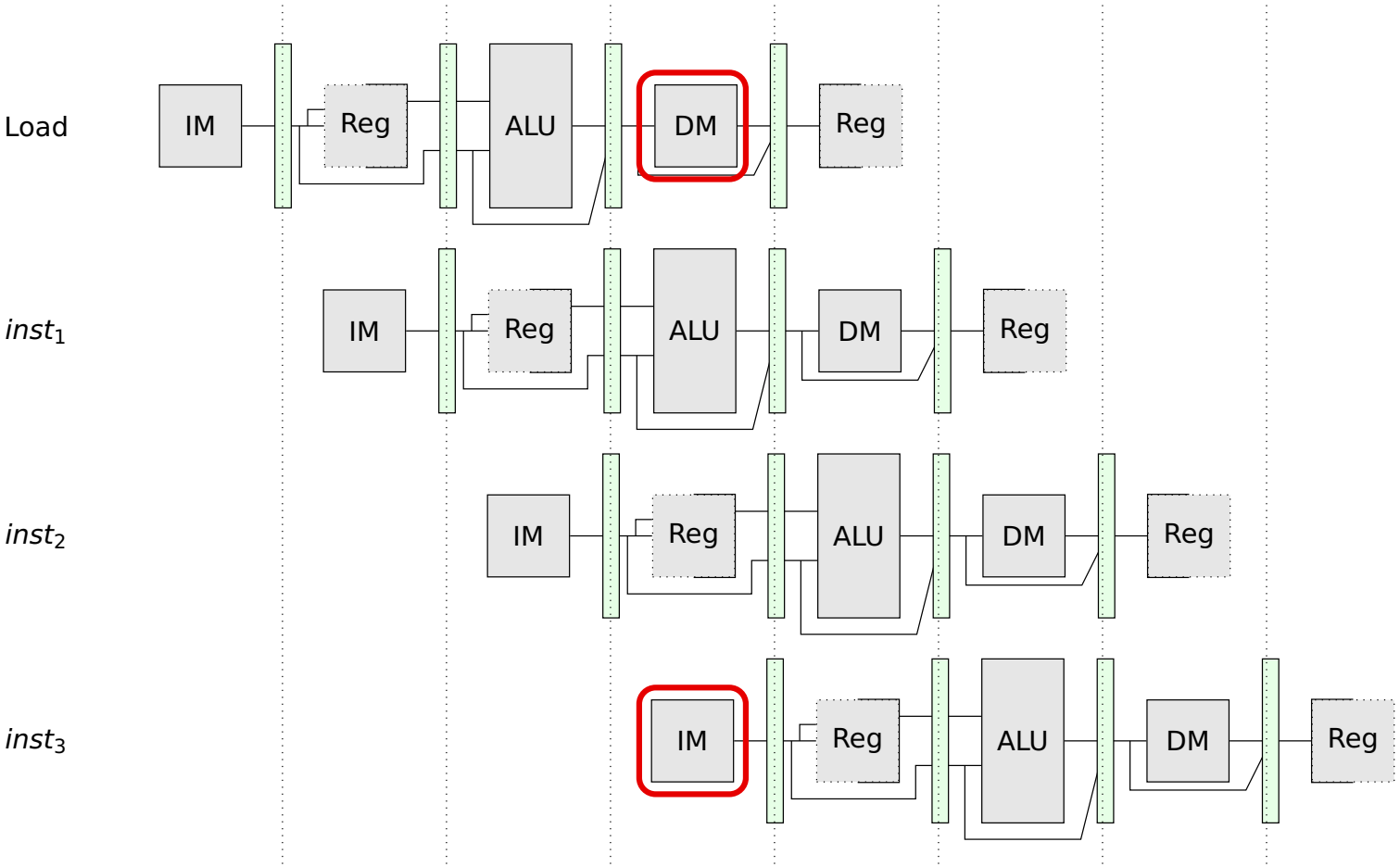
Il intervient lors d'un conflit de ressources: une ressource matérielle ne peut pas être utilisée par 2 instructions simultanément:

- une unité fonctionnelle n'est pas complètement *pipelinée*
- une unité n'a pas été suffisamment dupliquée.

Par exemple, avec une mémoire commune (instructions/données) avec un seul accès à la fois possible.

Aléa structurel

Exemple:



Performance moyenne avec les aléas

L'accélération (acc) avec le pipeline est définie:

$$acc = \frac{\textit{temps moyen des instructions sans pipeline}}{\textit{temps moyen des instructions avec pipeline}}$$

$$acc = \frac{CPI_{unpipelined}}{CPI_{pipelined}} \times \frac{\textit{temps de cycle sans pipeline}}{\textit{temps de cycle avec pipeline}}$$

et $CPI_{pipelined} = CPI\ parfait + cycles\ perdus\ avec\ bulles\ par\ inst$

Exemple

On considère un processeur avec un aléa structurel qui intervient dans 40% des instructions (1 cycle de pénalité). La version sans aléa (CPI=1) a une horloge 5% plus lente.

- quelle version est préférable?

Compromis

Si un aléa structurel intervient rarement, le coût (matériel) pour le supprimer peut ne pas être rentable.

Aléa de données

L'aléa de donnée est la conséquence d'une dépendance entre les instructions. Exemple

```
add  x1, x2, x3
or   x4, x1, x4
sub  x5, x1, x4
```

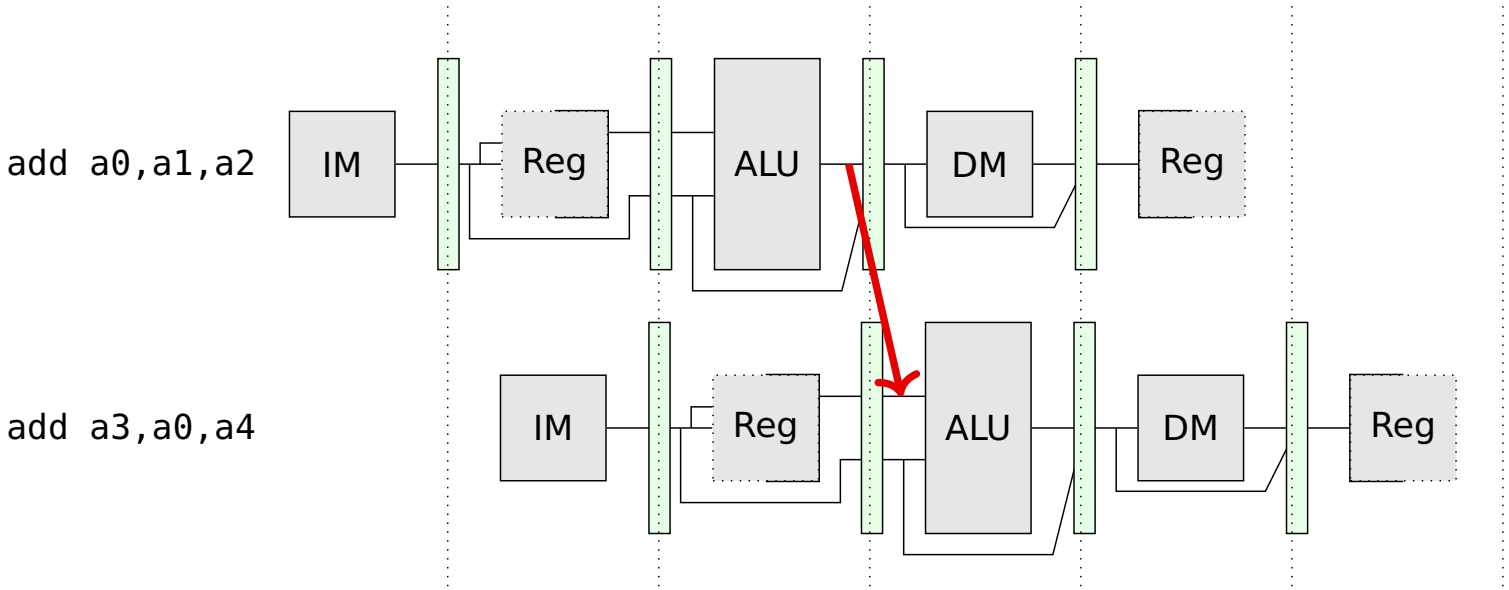
Combien de cycles sont nécessaires pour l'exécution de ce morceau de code? Combien de bulles sont rajoutées?

Aléa de données

Une solution matérielle

rajouter des court-circuits! (*bypass*, *forwarding*, *short-circuitry*).

il "suffit" de modifier le flot de contrôle afin de proposer le résultat de l'opération "au plus tôt".



Aléa de données

- comment peut-on "facilement" implémenter ce bypass au niveau matériel?

Aléa de données

- comment peut-on "facilement" implémenter ce bypass au niveau matériel?

il suffit de:

- comparer le numéro de registre r_d en sortie d'ALU avec ceux des 2 sources r_s1 et rs_2;
- mettre un multiplexeur pour chacune des entrées avec choix r_d ou rs_i;
- le résultat de la comparaison contrôle le multiplexeur;

Aléa de données

Ça ne suffit pas forcément tout le temps. Exemple:

```
li   a1, 10      ; a1 <= 10
lw   a0, 10(a2)  ; a0 <= mem[a2+10]
add  a2, a0, a1  ; a2 <= a0 + a1
```

Aléa de données

Une solution logicielle

Modifier l'ordre des instructions (si possible) pour diminuer les dépendances.

```
lw   a0, 10(a2) ; a0 <= mem[a2+10]
li   a1, 10     ; a1 <= 10
add  a2, a0, a1 ; a2 <= a0 + a1
```

Aléa de données

On distingue 3 types d’aléas de données:

RAW : Read After Write

WAW : Write After Write

WAR : Write After Read

Aléa de données: RAW

C’est le cas "classique" de l’attente d’une donnée produite précédemment:

```
add  a2, a0, a1
add  a3, a2, a4
```

C’est une *vraie dépendance* de données.

Aléa de données: WAW

```
add  a2, a0, a1
add  a2, a3, a4
```

C’est une dépendance de sortie. Ça peut arriver s’il y a plus qu’un étage pour l’écriture.

Aléa de données: WAR

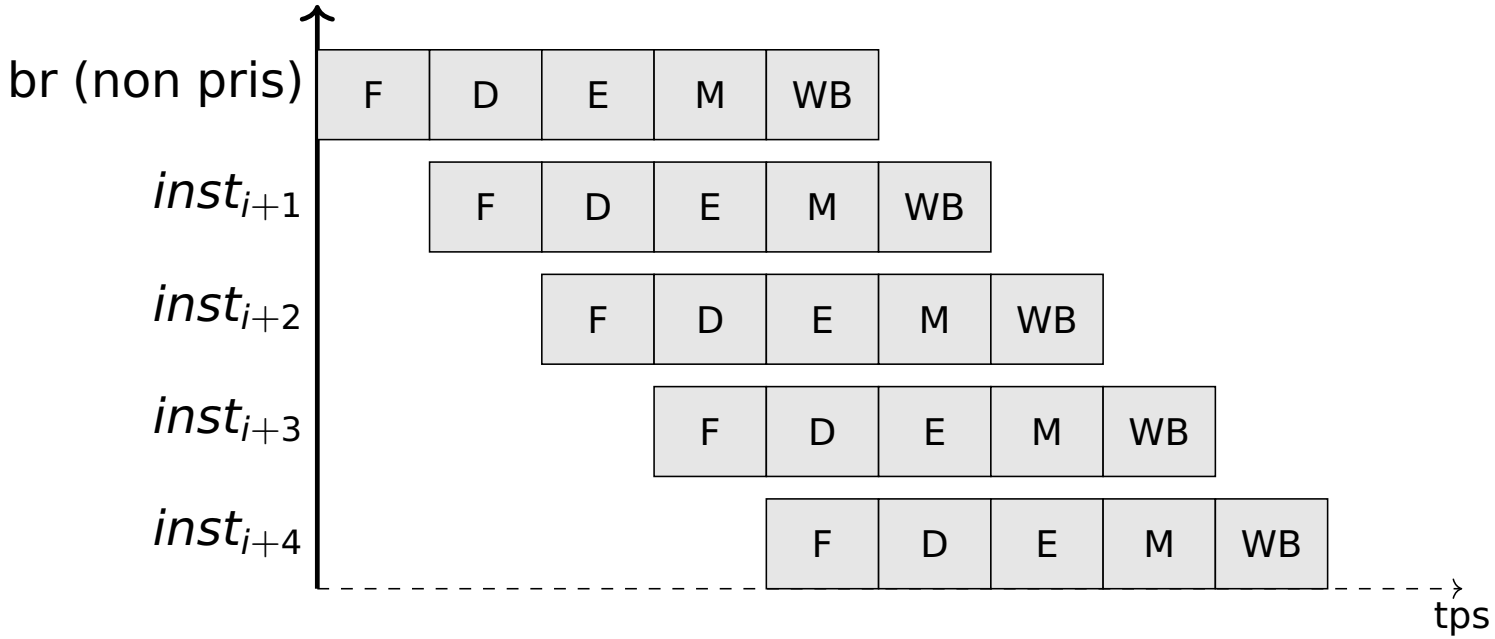
```
add  a2, a0, a1
add  a0, a3, a4
```

Dans ce cas, il faut que l'écriture se fasse bien avant la lecture de l'instruction précédente. (ça ne peut pas arriver sur un pipeline *statique* simple...)

Aléa de contrôle

Ils sont liés au instructions de saut dans le programme.

Premier cas: le saut n'est pas pris.



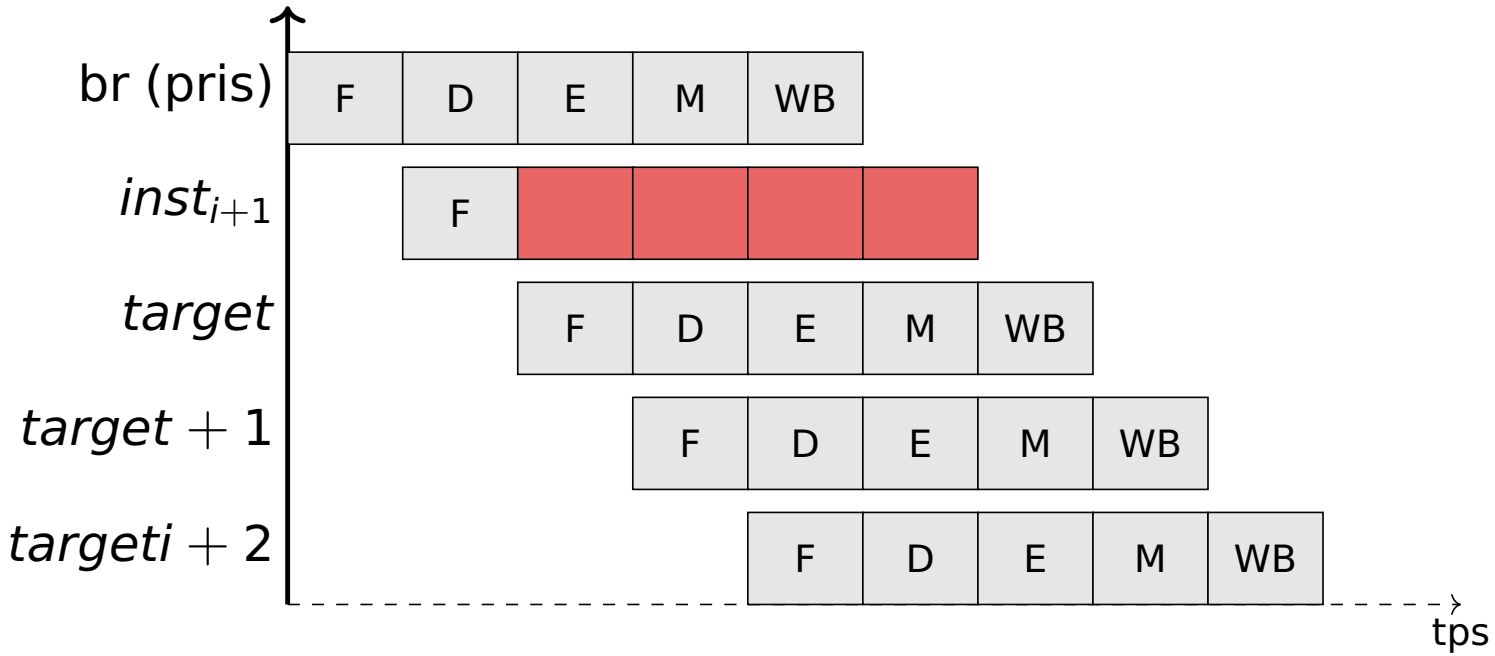
Aléa de contrôle

Ils sont liés au instructions de saut dans le programme.

Note:

On considère ici qu'on sait si le saut est pris ou non à la fin de l'étage de décodage.

Deuxième cas: le saut est pris.



Aléa de contrôle

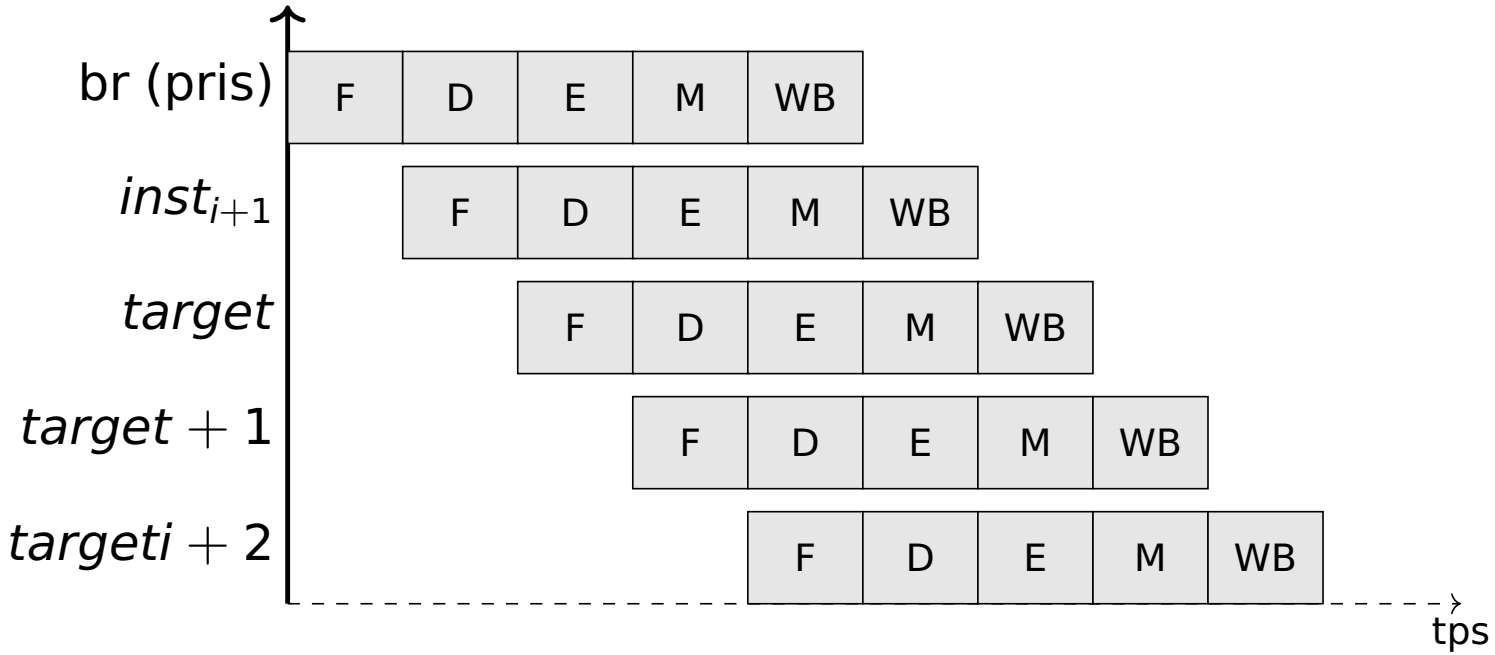
Dans la pratique:

- nécessité de détecter le saut au plus tôt dans le pipeline: généralement au décodage.
- la pénalité peut être bien plus importante si la mémoire à la cible du saut n'est pas accessible en 1 cycle (*pre-fetch buffer*).

Aléa de contrôle: *Delayed branch*

Quelle solution?

Une première approche: les sauts retardés (processeur SPARC). Dans ce cas, les n instructions suivantes ($n = 1\ldots$) sont exécutées dans tous les cas:



L'instruction $inst_{i+1}$ est systématiquement exécutée, plus besoin de bulle...

Aléa de contrôle: *Delayed branch*

Exemple: Implémenter une boucle `for` en assembleur RISC-V en considérant un *delay slot* de 1 instruction.

Ainsi: la tâche revient au compilateur d'utiliser ce *slot* pour ajouter une instruction (pour le prochain tour de boucle par exemple). Mais. . .

Aléa de contrôle: *Delayed branch*

c’est une fausse bonne idée, car on intègre dans l’ISA une contrainte d’architecture:

- le compilateur sera plus complexe (mais c’est faisable. . .)
- si l’architecture évolue, la taille du slot risque de ne pas être conforme avec la nouvelle architecture (trop court. . .): 31 étages pour le pipeline du Pentium 4

Contents

Prédiction de branchement

Les aléas de *contrôle* sont liés à une *instruction de branchement*.
Pour réduire l'impact, on essaie de déduire le plus tôt possible dans le pipeline:

- si le branchement est pris
- si oui, quelle est la cible

Délai important

un branchement conduit à modifier invalider toutes les instructions suivantes.

Sur notre architecture à 5 étages, le branchement est déduit dès le 2^e étage, mais il faut tout de même rajouter 1 bulle!

Prédiction de branchement

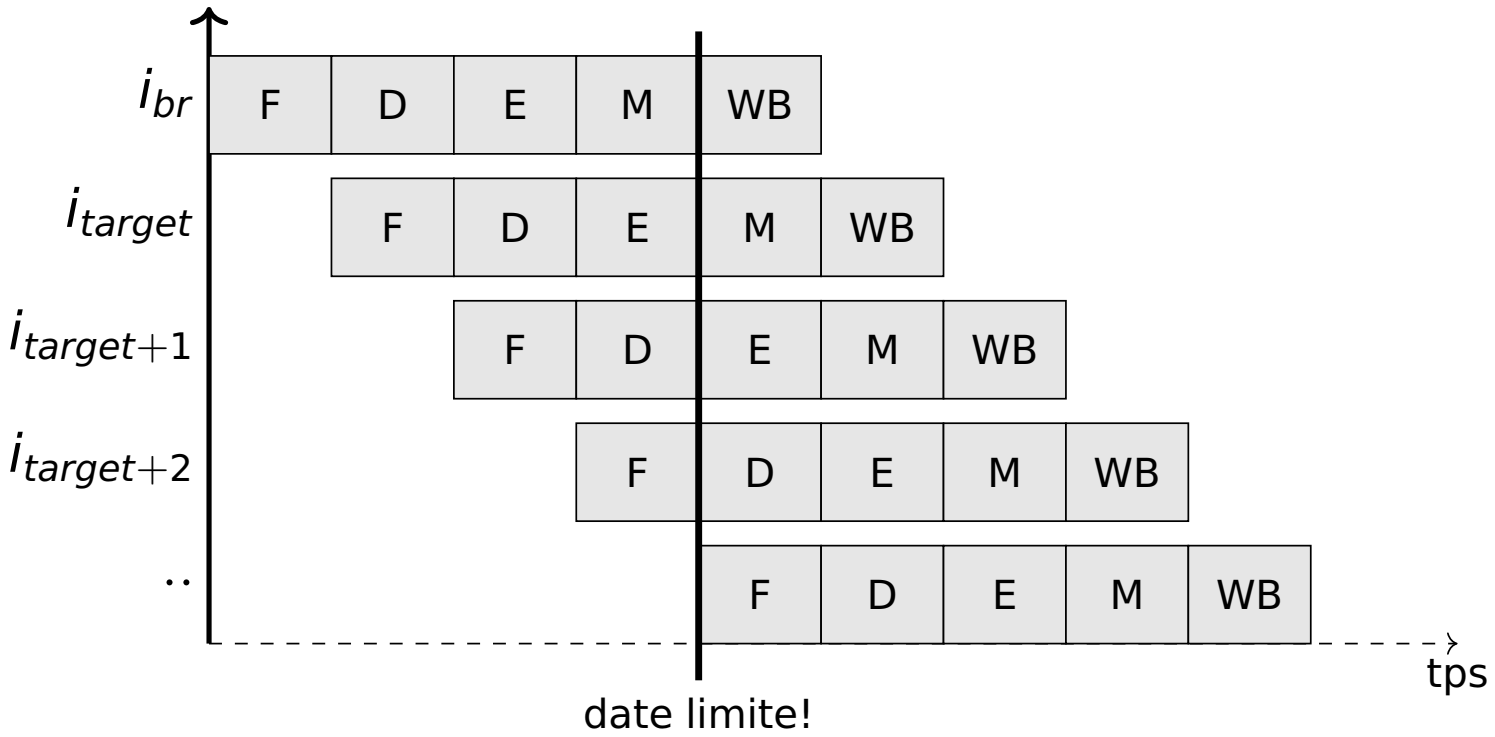
Et si on *essayait* de prédire le branchement?

- si on a juste: parfait, on n'a pas perdu de temps!
- si on a faux... il faut s'assurer que l'instruction ne va pas modifier l'état du système tant qu'elle est *spéculative*!

Prédiction de branchement

Concrètement, une *instruction spéculative* peut

- passer dans les étages F, D et E;
- par contre, pour M et R, il faut avoir levé le doute!



Prédiction statique

L'approche la plus simple est la prédiction statique. On peut avoir différents cas:

- tous les branchements pris ou non pris (sans exception)
- prendre en compte la direction de branchement...

exemple:

```
/* code compilé en -O0..*/
for(int i = 0; i<10; i++)
{
    val += i;
}
```

```
    sw      zero,-20(s0) #i
    j       .L5
.L6:
    lw      a4,-24(s0)
    lw      a5,-20(s0)
    add     a5,a4,a5
    sw      a5,-24(s0)
    lw      a5,-20(s0)
    addi    a5,a5,1
    sw      a5,-20(s0)
.L5:
    lw      a4,-20(s0)
    li      a5,9
    ble     a4,a5,.L6
```

CENTRALE NANTES

Prédiction statique

L'approche la plus simple est la prédiction statique. On peut avoir différents cas:

- tous les branchements pris ou non pris (sans exception)
- prendre en compte la direction de branchement...

exemple:

```
if(cond)
{
    truc;
} else {
    troc;
}
fin
```

```
eval condition
beq L1
troc
j L2
L1: truc
L2: fin
```

Prédiction statique

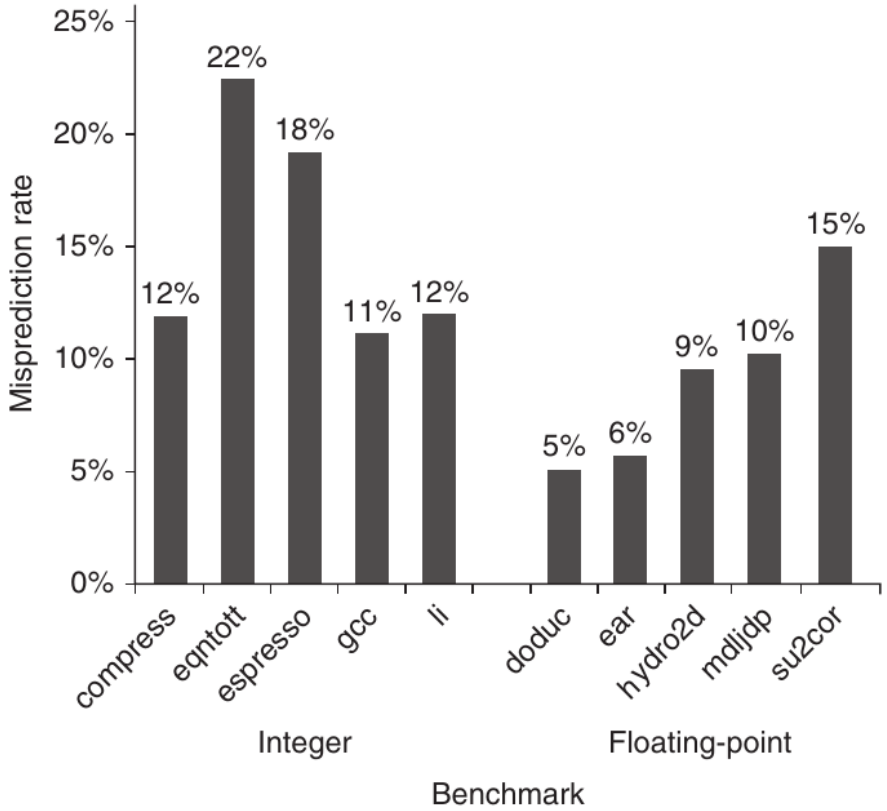
Les compilateurs peuvent prendre en compte cette prédiction:

- Dans une boucle, le test est réalisé à la fin, et c'est un branchement arrière qui est effectué
  - si on prédit le branchement arrière pris, alors pour une boucle de n tours, on ne se trompe que la première fois!
- pour les conditions... c'est plus compliqué...

certains jeux d'instructions (PowerPC, Pentium 4) avait dans leur jeu d'instruction la possibilité de rajouter des *branch hint* pour indiquer le comportement du branchement par défaut (prédit pris/non pris).
On peut évaluer la prédiction à partir d'un profil (scenrios de test).

Prédiction statique

Évaluation:



taux de mauvaises prédictions sur un ensemble de *benchmark* SPEC92.

source: Computer Architecture A Quantitative Approach 5^{th} ed. - Hennessy & Patterson

Prédiction dynamique

La prédiction statique apporte un gain interessant. . . mais montre des limites.
On pourrait peut-être se baser sur le comportement *en ligne* pour avoir des prédictions plus précises. . .
Le plus simple utilise un *branch-prediction buffer*

- zone mémoire dédiée;
- indexée par la partie basse de PC;
- contient 1 bit pour indiquer la nature du précédent saut

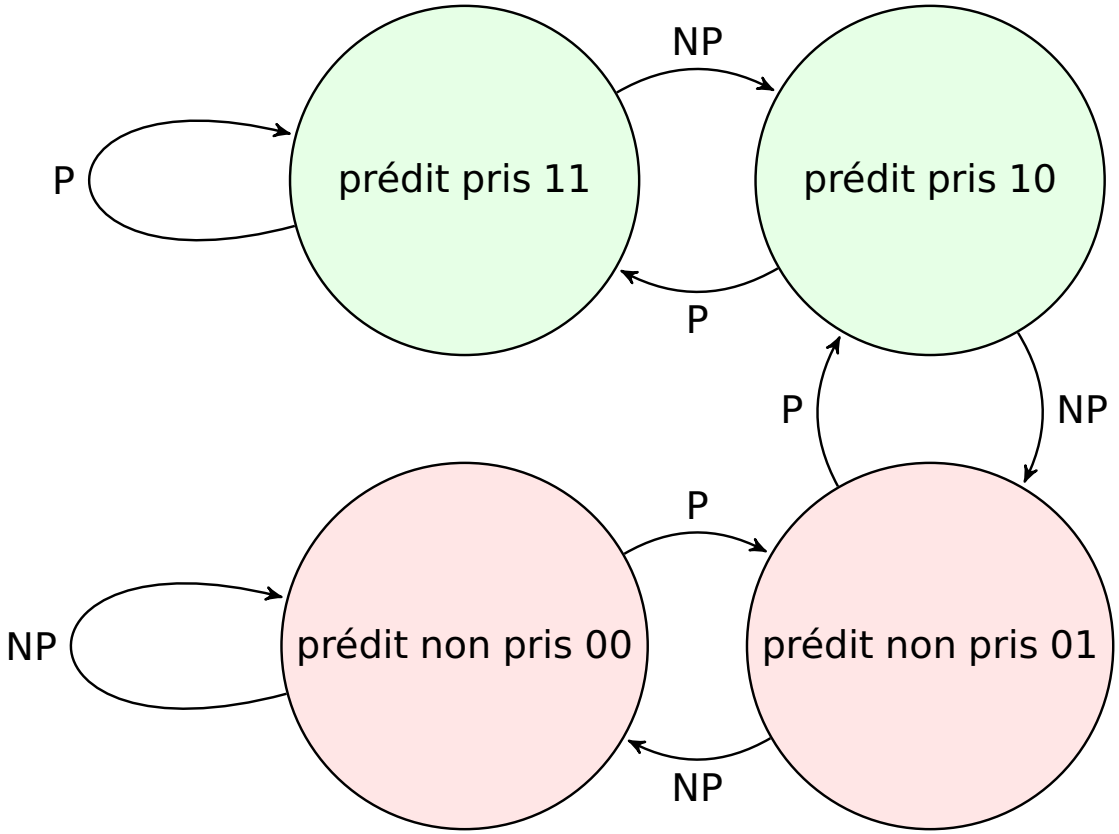
Prédiction dynamique

C’est mieux. . . mais

- on n’utilise que le poids faible de PC
- une sortie de boucle conduit à 2 mauvaises prédictions

Prédiction dynamique

Une prédiction à 2 bits, avec saturation:

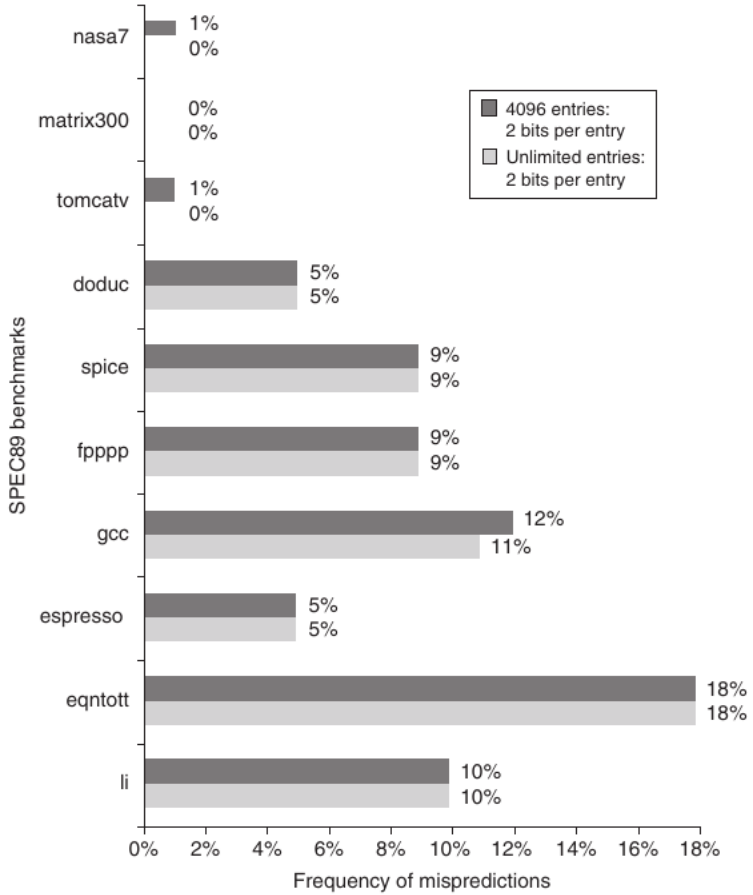


P: pris NP: non pris

Le résultat de la prédiction dépend du bit de poids fort du predicteur.

Prédiction dynamique

Évaluation:



taux de mauvaises prédictions sur un ensemble de *benchmark* SPEC89 avec un buffer de 4096 entrées et prédicteur 2 bits.

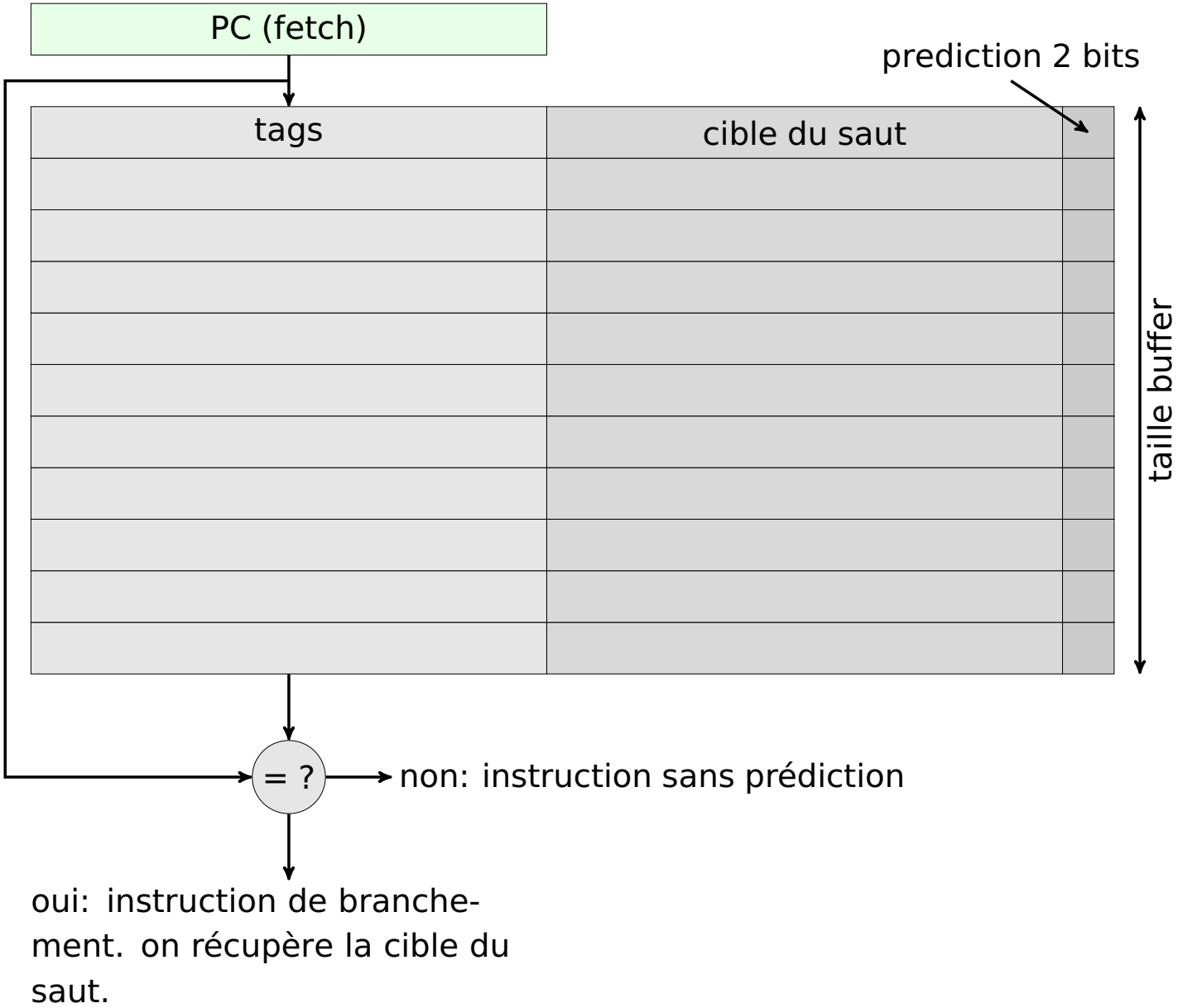
source: Computer Architecture A Quantitative Approach 5^{th} ed. - Hennessy & Patterson

Branch Target Buffer BTB

Le BTB contient plus d'information. Pour chaque ligne, on a:

- un *tag* avec la valeur de PC
- l'adresse destination
- le prédicteur 2 bits.

Branch Target Buffer BTB



Branch Target Buffer BTB

Étapes:

- l'évaluation se fait le plus tôt possible (étage FETCH)
- si l'instruction n'est pas dans le BTB, mais que c'est un saut, elle est rajoutée
- si la prédiction est fausse... on invalide les instructions spéculatives
- on ne stocke que les branchement prédit pris...

Branch Target Buffer BTB

instruction dans le *buffer*	prédiction	réalisation	pénalité
oui	pris	pris	non
oui	pris	non pris	oui
non		pris	oui
non			non

Exemple: En considérant

- une pénalité de 2 cycles
- une prédiction correcte à 90% (pour les instructions dans le buffer)
- un taux de *hit* de 90% dans le BTB (pour les branchement prédits pris)

Quel est la pénalité moyenne des sauts?

Note: pour un processeur haute performance, la pénalité peut atteindre 15 cycles!

Branch Target Buffer BTB

Variante: On peut mettre aussi dans le cache, en plus de l'adresse cible, l'instruction cible

conséquence: pour un saut inconditionnel, le BTB substitue directement le code de l'instruction entre Fetch et Decode, et on peut avoir une instruction en 0 cycles!

Pour les architecture qui peuvent exécuter plusieurs instructions par cycle (*superscalaires*), on peut avoir aussi des instructions conditionnelles résolues en temps nul.

Branchement à corrélation

Le prédicteur de branchement a 2 bits traite chaque branchement de manière indépendante, et est moins efficace si les branchement sont liés:

```
if(aa==2) aa=0;
if(bb==2) bb=0;
if(aa!=bb) {.}
```

extrait de code du benchmark `eqntott` de la suite SPEC

La 3^e condition dépend des 2 précédentes.

Branchement à corrélation

Le code pourrait ressembler à:

```
    la      a5,aa           # a5 <- @ de aa
    lw      a3,0(a5)        # a3 <- aa
    li      a4,2
    bne     a3,a4,cond1     # aa!=2 ?
    sw      x0,a5
cond1:
    la      a5,bb           # a5 <- @ de bb
    lw      a2,0(a5)        # a2 <- bb
    bne     a2,a4,cond2     # bb!=2 ?
    sw      x0,a5
cond2:
    beq     a2,a3,cond3     # dépendence avec les 2 précédents.
    #...
cond3: #fin condition
```

Branchement à corrélation

Le *prédicteur de branchement à corrélation*, ou *prédicteur à 2 niveaux* (m, n):

- utilise les m branchements précédents, pour choisir dans 2^m prédicteurs
- chacun de ces préditeurs est un prédicteur n bits.

L'implémentation est "relativement" simple car on peut utiliser un *registre à décalage* (à m bits) pour enregistrer l'historique (1 si le branchement a été pris, 0 sinon).
Ce registre permet de retrouver le prédicteur n bits associé.

Branchement à corrélation

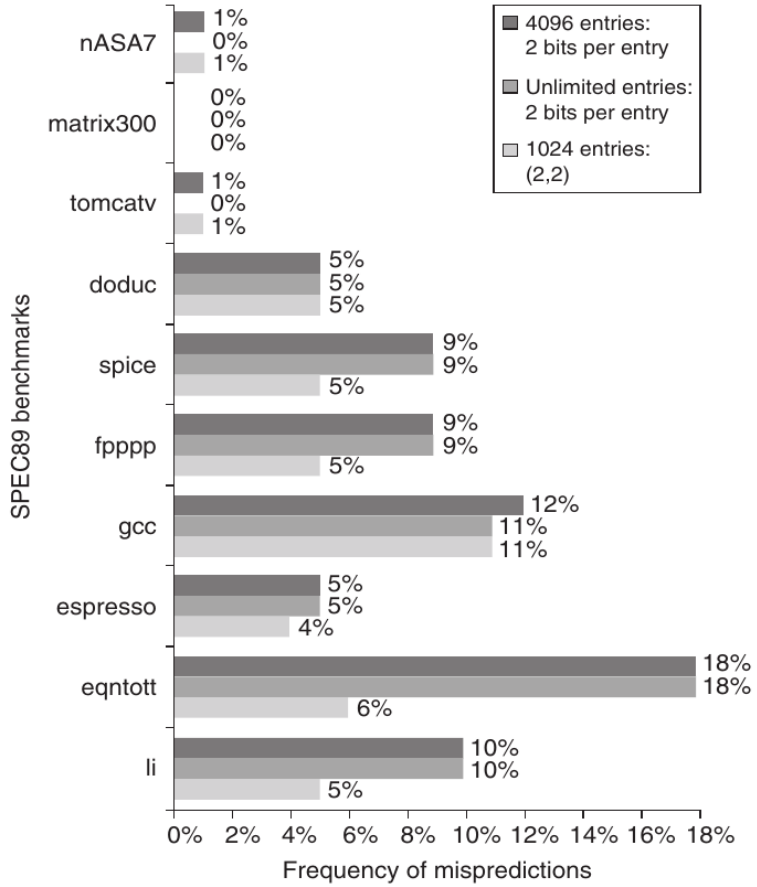
Exemple: table de 64 entrées, avec un prédicteur (2,2).

- on utilise les 4 bits de poids faible de l'adresse de branchement;
- on concatène avec les 2 bits du registre à décalage, qui indique le comportement des 2 derniers sauts
- on obtient une valeur sur 6 bits $\Rightarrow$ on obtient le compteur à saturation du prédicteur 2 bits (parmi les $2^6 = 64$ entrées).

Par conséquent, un prédicteur à 2 bits classique est un cas particulier sans historique: prédicteur (0,2).

Branchement à corrélation

Évaluation:



taux de mauvaises prédictions sur un ensemble de *benchmark* SPEC89 avec un buffer de 4096 entrées et prédicteur 2 bits.

source: Computer Architecture A Quantitative Approach 5^{th} ed. - Hennessy & Patterson

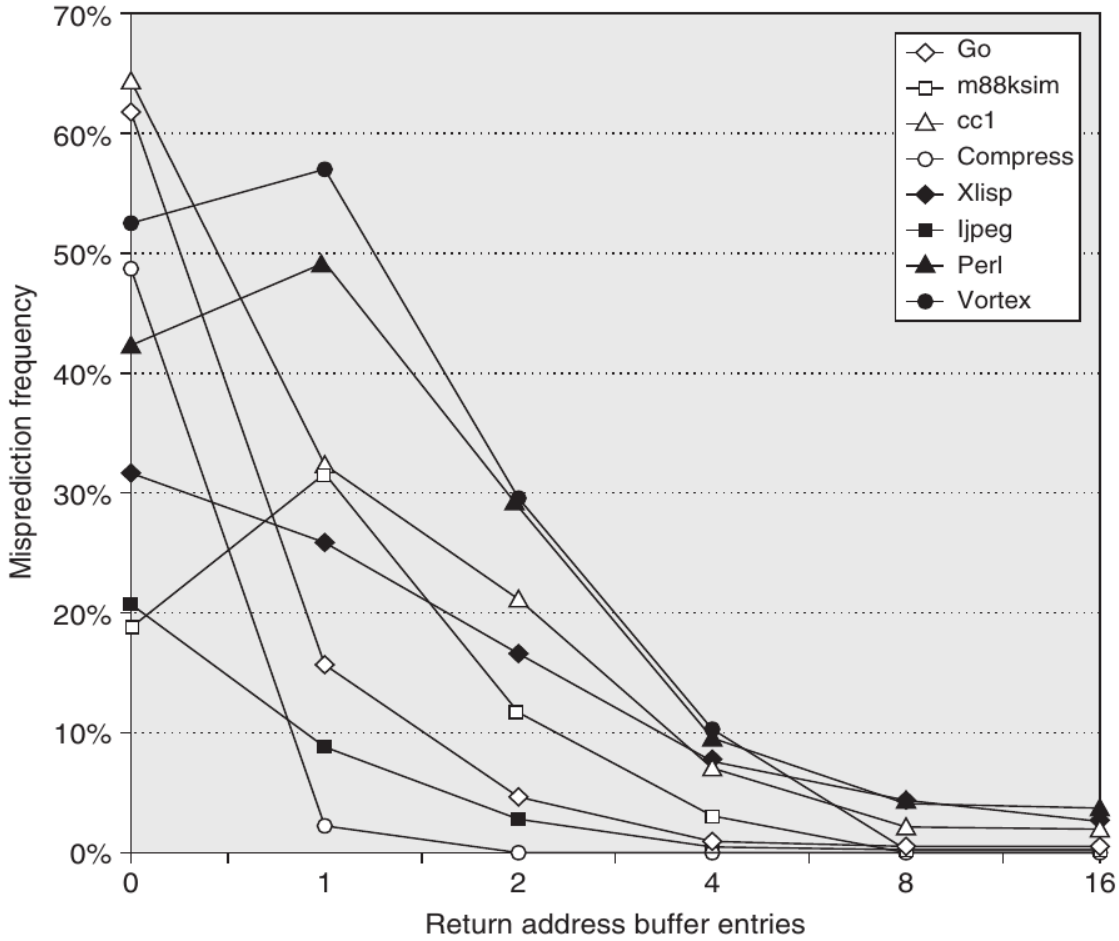
Return address prediction

Le problème des retours de fonction est que l'adresse de retour... n'est pas toujours la même! C'est un saut indirect. Et ceci représente beaucoup de sauts (15% dans les benchmark SPEC95).
On peut uriliser une petite zone mémoire avec un fonctionnement comme une pile:

- lors d'un appel `jalr`, on insère la donnée sur la pile
- lors du retour `ret`, on récupère l'adresse de retour

Bien sûr, les performances dépendandent de la taille de cette pile matérielle.

Return address prediction



taux de mauvaises prédictions avec une pile de retours de fonctions sur un ensemble de *benchmark* SPEC95

source: Computer Architecture A Quantitative Approach 5th ed. - Hennessy & Patterson