

## Architectures des ordinateurs (une introduction)

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



## Bibliographie

- *Architectures logicielles et matérielles*, Amblard, Fernandez, Lagnier, Maraninchi, Sicard, Waille, Dunod 2000
- *Architecture des ordinateurs*, Cazes, Delacroix, Dunod 2003.
- *Computer Organization and Design : The Hardware/Software Interface*, Patterson and Hennessy, Dunod 2003.
- *Processeurs ARM*, Jorda. DUNOD 2010.
- <https://im2ag-moodle.univ-grenoble-alpes.fr/course/view.php?id=336>
- <https://moodle.caseine.org/course/view.php?id=716>

## Organisation

(1/2)

- **Cours** : bruno.ferres@univ-grenoble-alpes.fr & kevin.marquet@univ-grenoble-alpes.fr
- **TD et TDE**
  - INM-01 : pierre.marrec@inria.fr;
  - INM-02 : philippe.waille@univ-grenoble-alpes.fr (TD) & ernest.foussard@grenoble-inp.fr (TP);
  - INM-03 : denis.bouhineau@imag.fr;
  - INM-04 : david.rios.uga@gmail.com;
  - MIN-01 : kevin.marquet@univ-grenoble-alpes.fr;
  - MIN-02 : gomezбай@univ-grenoble-alpes.fr;
  - MIN-03 : bruno.ferres@univ-grenoble-alpes.fr;
  - MIN-04 : philippe.waille@univ-grenoble-alpes.fr
  - MIN-Int : david.rios.uga@gmail.com;
- **Note** : 0.25 CC1 (partiel) + 0.25 CC2 (note de TP) + 0.5 examen
- **Partiel** : mi-mars
- **Examen** : mi-mai

## Modèle de von Neumann : qu'est ce qu'un ordinateur ?

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



## Description du modèle de von Neumann

(2/5)

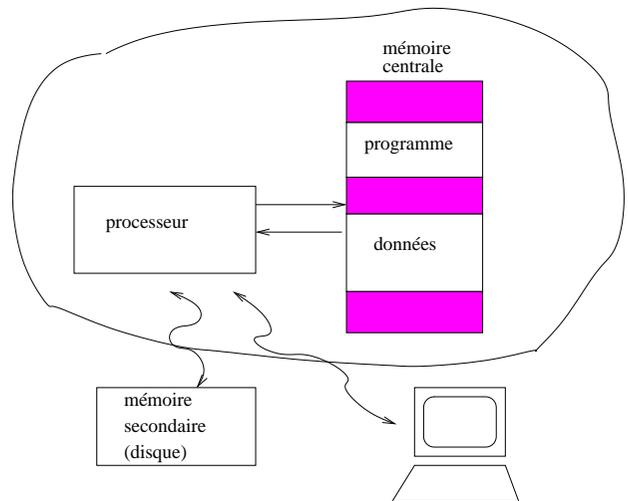


Figure – Processeur, mémoire et périphériques

## Mémoire centrale (vision abstraite)

La mémoire contient des **informations** prises dans un certain domaine

La mémoire contient un certain nombre (fini) d'**informations**

Les informations sont **codées** par des vecteurs binaires d'une certaine taille

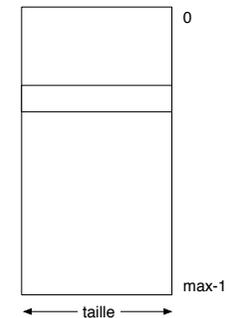


Figure – Mémoire abstraite

## Résumé : processeur/mémoire

**Processeur** : circuit relié à la **mémoire** (bus adresses, données et contrôle)

La mémoire contient des informations de nature différentes :

- des données : représentation binaire d'une couleur, d'un entier, d'une date, etc.
- des instructions : représentation binaire d'une ou plusieurs actions à réaliser.

Le processeur, relié à une mémoire, peut :

- **lire** un mot : le processeur fournit une adresse, un signal de commande de lecture et reçoit le mot.
- **écrire** un mot : le processeur fournit une adresse ET une donnée et un signal de commande d'écriture.
- ne pas accéder à la mémoire.
- **exécuter** des instructions, ces instructions étant des informations lues en mémoire.

## Entrées/Sorties : définitions

On appelle **périphériques d'entrées/sortie** les composants qui permettent :

- L'interaction de l'ordinateur (mémoire et processeur) avec l'**utilisateur** (clavier, écran, ...)
- L'interaction de l'ordinateur avec le **réseau** (carte réseau, carte WIFI, ...)
- L'accès aux **mémoires secondaires** (disque dur, clé USB...)

L'accès aux périphériques se fait par le biais de **ports** (usb, serie, pci, ...).

**Sortie** : ordinateur  $\longrightarrow$  extérieur

**Entrée** : extérieur  $\longrightarrow$  ordinateur

**Entrée/Sortie** : ordinateur  $\longleftrightarrow$  extérieur

# Les bus

Un **bus** informatique désigne l'ensemble des lignes de communication (câbles, pistes de circuits imprimés, ...) connectant les différents composants d'un ordinateur.

- **Le bus de données** permet la circulation des données.
- **Le bus d'adresse** permet au processeur de **désigner à chaque instant la case mémoire ou le périphérique** auquel il veut faire appel.
- **Le bus de contrôle** indique quelle est l'**opération que le processeur veut exécuter**, par exemple, s'il veut faire une écriture ou une lecture dans une case mémoire.  
On trouve également, dans le bus de contrôle, une ou plusieurs lignes qui permettent aux périphériques d'effectuer des demandes au processeur ; ces lignes sont appelées **lignes d'interruptions matérielles (IRQ)**.

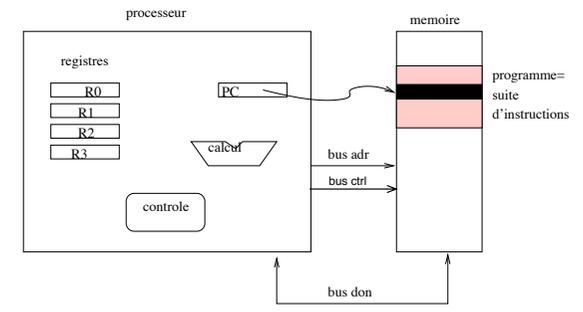
# Codage des instructions : langage machine

- Représentation d'une instruction en mémoire : **un vecteur de bits**
- **Programme** : **suite de vecteurs binaires** qui codent les instructions qui doivent être exécutées.
- Le codage des instructions constitue le **Langage machine** (ou *code machine*).
- Chaque modèle de processeur a son propre langage machine (on dit que le langage machine est **natif**)

# Composition du processeur

Le processeur est composé d'unités (ressources matérielles internes) :

- **des registres** : cases de mémoire interne  
Caractéristiques : désignation, lecture et écriture "simultanées"
- **des unités de calcul (UAL)**
- **une unité de contrôle** : (UC, *Central Processing Unit*)
- **un compteur ordinal** ou **compteur programme** : PC



# Codage des informations et représentation des nombres par des vecteurs binaires

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



## Exemples (3/3) : Code ASCII (Ensemble des caractères affichables)

ASCII = « American Standard Code for Information Interchange »

On obtient le tableau ci-dessous par la commande Unix `man ascii`

32	␣	33	!	34	"	35	#	36	\$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124		125	}	126	~	127	del

Code.ascii (q) = 113; Decode.ascii (51) = 3.

## Application en langage C

```
printf("%c", 'A' + 'b' - 'a');
printf("%c", 'A' + 3);
```

```
char c;
scanf("%c",&c);
if(c>='a' && c<='z') {
    printf("%c est une lettre minuscule",c);
}
```

'A'+'b'-'a'

→ affichage de la lettre B

'A'+3

→ affichage de la lettre D

char c;

→ entier sur 5 bits

if (c>='a' && c<='z')

→ comparaison entier et code ascii

## Conclusion sur le codage : Où est le code ?

- Le code n'est pas dans l'information codée.  
Par exemple : 14 est :
  - le code du jaune dans le code des couleurs du PC ...
  - ou le code du couple (2,4) ...
  - ou le code du bleu pâle dans le code du commodore 64 ...
- Pour interpréter, comprendre une information codée il faut connaître la règle de codage.  
Le code seul de l'information ne donne rien, c'est le **système de traitement de l'information (logiciel ou matériel)** qui « connaît » la règle de codage, sans elle il ne peut pas traiter l'information.

## Exercice : Enumérer les nombres représentables sur 3 chiffres binaires.

```
0 : 0 0 0
1 : 0 0 1
2 : 0 1 0
3 : 0 1 1
4 : 1 0 0
5 : 1 0 1
6 : 1 1 0
7 : 1 1 1
```

## Quelques valeurs à connaître

X	$2^X$
0	1
1	2
2	4
3	8
4	16
8	256
10	1 024 ( $\approx 1\ 000$ , 1 Kilo)
16	65 536
20	1 048 576 ( $\approx 1\ 000\ 000$ , 1 Méga)
30	1 073 741 824 ( $\approx 1\ 000\ 000\ 000$ , 1 Giga)
31	2 147 483 648
32	4 294 967 296

## Conversion base 10 vers base 2 : Troisième méthode

169	1	(169 = 84 × 2 + 1)
84	0	(84 = 42 × 2 + 0)
42	0	(42 = 21 × 2 + 0)
21	1	
10	0	
5	1	
2	0	
1	1	
0		

On a ainsi  $169_{10} = 10101001_2$

## Conversion base 2 vers base 10

Soit  $a_{n-1}a_{n-2}\dots a_1a_0$  un nombre entier en base 2

En utilisant les puissances de 2 :

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
128	64	32	16	8	4	2	1

$$(a_{n-1}a_{n-2}\dots a_1a_0)_2 \text{ vaut } (a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_12^1 + a_02^0)_{10}$$

### Exemple

$$\begin{aligned}
 1010 \text{ vaut } & 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
 & = 2^3 + 2^1 \\
 & = 8 + 2 = 10
 \end{aligned}$$

## Représentation des relatifs, solution : Complément à deux

Sur  $n$  bits, en choisissant  $00\dots000$  pour le codage de zéro, il reste  $2^n - 1$  possibilités de codage : la moitié pour les positifs, la moitié pour les négatifs.

**Attention**, ce n'est pas un nombre pair, l'intervalle des entiers relatifs codés ne sera pas symétrique.

Principe :

- Les entiers positifs sont codés par leur code en base 2
- Les entiers négatifs sont codés de façon à ce que  $\text{code}(a) + \text{code}(-a) = 0$

D'où **sur 8 bits**, intervalle représenté  $[-128, +127] = [-2^7, 2^7 - 1]$

- $x \geq 0$   $x \in [0, +127]$  :  $\text{CodeC2}(x) = x$
- $x < 0$   $x \in [-128, -1]$  :  $\text{CodeC2}(x) = x + 256 = x + 2^8$   
( $x$  étant négatif et  $\geq -128$ ,  $x + 2^8$  est « codable » sur 8 bits)  
( $x + 2^8 > 127$ , donc pas d'ambiguïté)

$$\text{CodeC2}(a) + \text{CodeC2}(-a) = a - a + 2^8 = 0 \text{ (sur 8 bits)}$$

# Complément à deux : trouver le code d'un entier négatif

Soit un entier relatif positif  $a$  codé par les  $n$  chiffres binaires :

$$(a_{n-1} a_{n-2} \dots a_1 a_0)_2$$

$$\begin{aligned} \text{valeur}(-a) &= 2^n - \text{valeur}(a) \\ &= 2^n - (a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_12 + a_0) \\ &= (2^{n-1} + 2^{n-1}) - (a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_12 + a_0) \\ &= (1 - a_{n-1})2^{n-1} + 2^{n-1} - (a_{n-2}2^{n-2} + \dots + a_12 + a_0) \\ &= \dots \\ &= (1 - a_{n-1})2^{n-1} + (1 - a_{n-2})2^{n-2} + \dots + (1 - a_0) + 1 \end{aligned}$$

## Règle pour un entier négatif

- 1 écrire le code de la valeur absolue
- 2 inverser tous les bits
- 3 ajouter 1

# Complément à deux : autre version

Comment retrouver l'opposé d'un entier  $A$  ?

1 prendre  $A = a_{n-1} a_{n-2} \dots a_1 a_0$

2 remarquer que

$$A + \bar{A} = 11\dots11 = -1$$

3 en déduire que

$$-A = \bar{A} + 1$$

# Indicateurs

	naturel	relatif
débordement addition	$C = 1$	$V = 1$
débordement soustraction	$C = 0$	$V = 1$

2 autres indicateurs (flags) :

- $N$  : bit de signe (1 si négatif)
- $Z$  : test si nulle ( $Z = 1$  si nulle)

Les indicateurs permettent aussi d'évaluer les conditions ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $=$ ,  $\neq$ ).

Pour évaluer une condition entre  $A$  et  $B$ , le processeur positionne les indicateurs en fonction du résultat de  $A - B$ .

**Exemple :** Supposons que  $A$  et  $B$  sont des entiers naturels. Alors,  $A - B$  provoque un débordement (c'est-à-dire,  $C = 0$ ) si et seulement si  $A < B$ .

# Table d'addition (3 bits, naturels)

Récapitulatif :

Pour 3 bits et les entiers naturels :

- il y a 8 entiers naturels : 0 ... 7,
- et l'addition suivante

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

## Table d'addition (3 bits, relatifs)

### Récapitulatif :

Pour 3 bits et les entiers relatifs codés en complément à 2 :

- il y a 8 entiers relatifs : -4 ... 3,
- et l'addition suivante

+	-4	-3	-2	-1	0	1	2	3
-4	0	1	2	3	-4	-3	-2	-1
-3	1	2	3	-4	-3	-2	-1	0
-2	2	3	-4	-3	-2	-1	0	1
-1	3	-4	-3	-2	-1	0	1	2
0	-4	-3	-2	-1	0	1	2	3
1	-3	-2	-1	0	1	2	3	-4
2	-2	-1	0	1	2	3	-4	-3
3	-1	0	1	2	3	-4	-3	-2

## Langage d'assemblage, langage machine

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



## Etapes de compilation

- **Précompilation** : `arm-eabi-gcc -E monprog.c > monprog.i`  
source : `monprog.c` → source « enrichi » `monprog.i`
- **Compilation** : `arm-eabi-gcc -S monprog.i`  
source « enrichi » → langage d'assemblage : `monprog.s`
- **Assemblage** : `arm-eabi-gcc -c monprog.s`  
langage d'assemblage → binaire translatable : `monprog.o` (fichier objet)  
même processus pour `malib.c` → `malib.o`
- **Edition de liens** : `arm-eabi-gcc monprog.o malib.o -o monprog`  
un ou plusieurs fichiers objets → binaire exécutable : `monprog`

## Instruction de calcul entre des informations mémorisées

L'instruction désigne la(les) **source(s)** et le **destinataire**. Les *sources* sont des cases mémoires, registres ou des valeurs. Le *destinataire* est un élément de mémorisation.

L'instruction code : destinataire, source1, source2 et l'opération.

désignation du destinataire	←	désignation de source1	oper	désignation de source2
mém, reg		mém, reg		mém, reg, valIMM

- mém** signifie que l'instruction fait référence à un mot dans la mémoire
- reg** signifie que l'instruction fait référence à un registre (nom ou numéro)
- valIMM** signifie que l'information source est contenue dans l'instruction

## Instruction de rupture de séquence

- **Fonctionnement standard** : Une instruction est écrite à l'adresse  $X$  ; l'instruction suivante (dans le temps) est l'instruction écrite à l'adresse  $X+t$  (où  $t$  est la taille de l'instruction). C'est implicite pour toutes les instructions de calcul.
- **Rupture de séquence** : Une instruction de *rupture de séquence* peut désigner la prochaine instruction à exécuter (à une autre adresse).

## Désignation des objets

(1/7)

On parle parfois, improprement, de **modes d'adressage**. Il s'agit de dire comment on écrit, par exemple, la valeur contenue dans le registre numéro 5, la valeur  $-8$ , la valeur rangée dans la mémoire à l'adresse  $0xff, \dots$

Il n'y a pas de **standard de notations**, mais des **standards de signification** d'un constructeur à l'autre.

L'**objet** désigné peut être **une instruction** ou **une donnée**.

## Désignation des objets : par registre

(2/7)

### Désignation registre/registre.

L'objet désigné (une donnée) est le contenu d'un registre. L'instruction contient le nom ou le numéro du registre.

- **En 6502 (MOS Technology)** : 2 registres  $A$  et  $X$  (entre autres)  
**TAX** signifie transfert de  $A$  dans  $X$   
 $X \leftarrow$  contenu de  $A$  (on écrira  $X \leftarrow A$ ).
- **ARM** : `mov r4 , r5` signifie  $r4 \leftarrow r5$ .

## Désignation des objets : immédiate

(3/7)

### Désignation registre/valeur immédiate.

La donnée dont on parle est littéralement **écrite dans l'instruction**

- **En ARM** : `mov r4 , #5` ; signifie  $r4 \leftarrow 5$ .

**Remarque** : la valeur immédiate qui peut être codée dépend de la place disponible dans le codage de l'instruction.

## Désignation des objets : indirect par registre (5/7)

### Désignation registre/indirect par registre

L'objet désigné est dans une case mémoire dont l'adresse est dans un registre précisé dans l'instruction.

- **En ARM** : `ldr r3, [r5]` signifie  $r3 \leftarrow$  (le mot mémoire dont l'adresse est contenue dans le registre 5)  
On note  $r3 \leftarrow \text{mem}[r5]$ .

## Etiquettes : définition (1/4)

**Etiquette** : nom choisi librement (quelques règles lexicales quand même) qui désigne une case mémoire. Cette case peut contenir une donnée ou une instruction.

Une **étiquette** correspond à une **adresse**.

Pourquoi ?

- L'emplacement des programmes et des données n'est à priori pas connu  
la directive `ORG` ne peut pas toujours être utilisée
- Plus facile à manipuler

## Séparation données/instructions

Le texte du programme est organisé en **zones** (ou **segments**) :

- **zone TEXT** : code, programme, instructions
- **zone DATA** : données initialisées
- **zone BSS** : données non initialisées, réservation de place en mémoire

On peut préciser où chaque zone doit être placée en mémoire : la directive `ORG` permet de donner l'adresse de début de la zone (ne fonctionne pas toujours!).

## Programmation des structures de contrôles

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



## Exécution séquentielle vs. rupture de séquence : rôle du PC

registre *PC* : Compteur de programme, repère l'instruction à exécuter

A chaque cycle :

- 1  $bus\ d'adresse \leftarrow PC ; bus\ de\ contrôle \leftarrow lecture$
- 2  $bus\ de\ donnée \leftarrow Mem[PC] = instruction\ courante$
- 3 Décodage et exécution
- 4 Mise à jour de *PC* (par défaut, incrémentation)

Les instructions sont exécutées séquentiellement  
sauf **ruptures de séquence !**

## Séquencement (3/7)

### Rupture inconditionnelle

Une instruction de **branchement inconditionnel** force une adresse *adr* dans *PC*.

La prochaine instruction exécutée est celle située en  $Mem[adr]$

**Cas TRES particulier : les premiers RISC (Sparc, MIPS)** exécutaient quand même l'instruction qui suivait le branchement.

## Séquencement (2/7)

### Séquencement « normal »

Après chaque instruction le registre *PC* est incrémenté.

Si l'instruction est codée sur *k* octets :  $PC \leftarrow PC + k$

Cela dépend des processeurs, des instructions et de la taille des mots.

- En **ARM**, toutes les instructions sont codées sur 4 octets. Les adresses sont des adresses d'octets. **PC progresse de 4 en 4**
- Sur certaines machines (ex. Intel), les instructions sont de longueur variable (1, 2 ou 3 octets). **PC prend successivement les adresses des différents octets de l'instruction**

## Séquencement (4/7)

### Rupture conditionnelle

**Si** une condition est vérifiée, **alors**

*PC* est modifié

**sinon**

*PC* est incrémenté normalement.

la condition est **interne** au processeur :

expression booléenne portant sur les *codes de conditions arithmétiques*

- *Z* : nullité,
- *N* : bit de signe,
- *C* : débordement (naturel) et
- *V* : débordement (relatif).

## Codage des structures de contrôle : exemples traités

- I1; **si** ExpCondSimple **alors** {I2; I3; I4;} I5;
- I1; **si** ExpCondSimple **alors** {I2; I3;} **sinon** {I4; I5; I6;} I7;
- I1; **tant que** ExpCond **faire** {I2; I3;} I4;
- I1; **répéter** {I2; I3;} **jusqu'à** ExpCond; I4;
- I1; **pour** (i←0 à N) {I2; I3; I4;} I5;
- **si** C1 **ou** C2 **ou** C3 **alors** {I1;I2;} **sinon** {I3;};
- **si** C1 **et** C2 **et** C3 **alors** {I1;I2;} **sinon** {I3;};
- **selon** a,b  
     a<b : I1;  
     a=b : I2;  
     a>b : I3;

Instruction *Si alors sinon* : Une solution

```
I1
si ExpCond alors {I2; I3} sinon {I4; I5; I6}
I7;
```

```
I1
evaluer ExpCond + ZNCV
branch si faux a etiq_sinon
I2
I3
branch etiq_finsi
etiq_sinon: I4
I5
I6
etiq_finsi: I7
```

## Une première solution

```
I1
tant que ExpCond faire {I2; I3;}
I4;

I1
debut: evaluer ExpCond + ZNCV
branch si faux fintq
I2
I3
branch debut
fintq: I4
```

Construction *selon*

```
selon a,b:
a<b : I1
a=b : I2
a>b : I3
```

Une solution consiste à traduire en **si alors sinon**.

```
si a<b alors I1
sinon si a=b alors I2
      sinon si a>b alors I3
```

ARM offre (ou offrait) une autre possibilité

# Programmation des appels et retours de procédures simples

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



## Utilisation de registres

$\mathcal{CT}_0$

Chaque valeur représentée par **une variable ou un paramètre** doit être rangée quelque part en **mémoire** : mémoire centrale ou registres.

Dans un premier temps, utilisons **des registres**.

On fait un choix (pour l'instant complètement arbitraire) :

- $i, j, k$  dans  $r0, r1, r2$
- $z$  dans  $r3$ ,  $p$  dans  $r4$
- la valeur  $x$  dans  $r5$
- le résultat de la fonction dans  $r6$
- si on a besoin d'un registre pour faire des calculs on utilisera  $r7$  (**variable temporaire**)

### Remarque :

Une fois, ces conventions fixées, on peut écrire le code de **la fonction indépendamment du code correspondant à l'appel**, mais cela demande beaucoup de registres.

## Quelle convention d'appel choisir ?

### Objectif du module

Prise en main de la convention utilisée par `gcc` :

- passage des arguments :  $\leftrightarrow$  les 4 premiers dans  $r0$  à  $r3$   
 $\leftrightarrow$  le reste par la pile
- valeur de retour : stockée dans  $r0$
- gestion du contexte : certains registres sont sauvegardés par l'**appelante**, d'autres par l'**appelée** (voir la documentation technique dans le poly du cours)

Mais la convention de `gcc` manipule des **concepts complexes**...

- nous allons progressivement étudier différentes propositions de *conventions temporaires* ( $\mathcal{CT}_0, \mathcal{CT}_1, \dots$ ), et leurs limites
- pour aboutir à la convention utilisée par un compilateur récent.

## Quel est le problème ?

$\mathcal{CT}_0$

Appel = branchement

**instruction de rupture de séquence inconditionnelle** ( $\mathbb{B}$ ) ?

MAIS **Comment revenir ensuite ?**

**Le problème du retour** : comment, à la fin de l'exécution du corps de la fonction, indiquer au processeur l'adresse à laquelle il doit se brancher ?

**Point de vigilance** : garantir le bon usage des registres.

# Adresse de retour $\mathcal{CT}_0$

Il existe une instruction de rupture de séquence **particulière** qui permet au processeur de **garder** l'adresse de l'instruction qui suit le branchement avant qu'il ne réalise le branchement, *i.e.*, avant qu'il ne transfère le contrôle.

Cette adresse est appelée **adresse de retour**.

On peut simuler cette instruction et la notion d'adresse de retour :

- Ajout d'une étiquette de retour (mais avec une utilisation très limitée, à un seul endroit d'appel/retour)
- Calcul de l'adresse de retour avant l'appel (mais attention : le PC avance au cours de l'exécution, PC vaut PC+8 à la fin de B)

L'instruction de rupture de séquence **particulière** recherchée est une facilité justifiée pour des raisons d'efficacité et de garantie de respect des conventions.

# Conclusion

## Conclusions :

- Il est possible d'avoir un ensemble d'instructions géré comme un bloc indépendant sous certaines conditions très limitatives : un seul appel bl ma\_proc, convention commune à l'appel, si main==appel, retour bx lr, ...)
- Pour s'affranchir de ces conditions :
  - **Paramètres** : il faut une zone de stockage dynamique **commune** à l'appelant et à l'appelé. L'appelant y range les valeurs **avant** l'appel, et l'appelé y prend ces valeurs et les utilise
  - **Variables locales** : il faut une zone de mémoire dynamique **privée** pour chaque procédure *appelée* pour y stocker ses variables locales : il ne faut pas que cette zone interfère les variables globales ou locales à l'appelant
  - **Variables temporaires** : elles ne doivent pas interférer avec les autres variables
  - **Généralisation** : il faut que la méthode choisie soit généralisable afin de pouvoir générer du code

**Remarque** : on a généralement peu de registre à notre disposition (16 en ARM, mais plusieurs sont dédiés à des tâches spécifiques, *i.e.* PC, LR, ...)

# Où est gardée cette adresse ? $\mathcal{CT}_0$

Dans le processeur **ARM**, l'instruction **BL** réalise un branchement inconditionnel avec **sauvegarde de l'adresse de retour** dans le registre nommé **lr** (*i.e.*, r14).

BL signifie *branch and link*

**Attention** : ne pas confondre BL et B

**Attention** : il ne faut pas modifier le registre lr pendant l'exécution de la fonction.

# Programmation de procédures (suite) Utilisation de la pile

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



# Mécanisme de pile

Notion de **haut de pile** : dernier élément entré  
 L'élément en haut de la pile est appelé *sommet*.

Deux opérations possibles :

**Dépiler** : suppression de l'élément en haut de la pile

**Empiler** : ajout d'un élément en haut de la pile

# Comment réaliser une pile ? (1/4)

- Une **zone de mémoire**,
- Un **repère** sur le haut de la pile  
*SP* : pointeur de pile, *stack pointer*
- Deux choix indépendants :
  - Comment **progresser** la pile :  
 ⇔ le sommet est **en direction des adresses croissantes (ascending) ou décroissantes (descending)**
  - Le pointeur de pile **pointe vers une case vide (empty) ou pleine (full)**

# Comment réaliser une pile ? (2/4)

**Mem** désigne la mémoire  
**sp** désigne le pointeur de pile  
**reg** désigne un registre quelconque

sens évolution	croissant	croissant	décroissant	décroissant
repère	1 <sup>er</sup> vide	der <sup>er</sup> plein	1 <sup>er</sup> vide	der <sup>er</sup> plein
empiler reg	Mem[sp]←reg sp←sp+1	sp←sp+1 Mem[sp]←reg	Mem[sp]←reg sp←sp-1	sp←sp-1 Mem[sp]←reg
dépiler reg	sp←sp-1 reg←Mem[sp]	reg←Mem[sp] sp←sp-1	sp←sp+1 reg←Mem[sp]	reg←Mem[sp] sp←sp+1

**Remarque**  
 Il existe des instructions **Arm** dédiées à l'utilisation de la pile  
 (exemple : pour la gestion **full descending** on utilise *stmfd* ou *push* pour empiler et *ldmfd* ou *pop* pour dépiler)

# Appel/retour : solution utilisée avec le processeur Arm $\mathcal{CT}_2$

Lors de l'appel, l'instruction **BL** réalise un branchement inconditionnel **avec sauvegarde de l'adresse de retour** dans le registre nommé **lr** (*i.e.*, r14).

**C'est le programmeur qui doit gérer les sauvegardes dans la pile!**

si nécessaire ...

# Programmation des appels de procédure et fonction (fin)

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



## Application : fonction fact avec des variables locales $\mathcal{CT}_5$

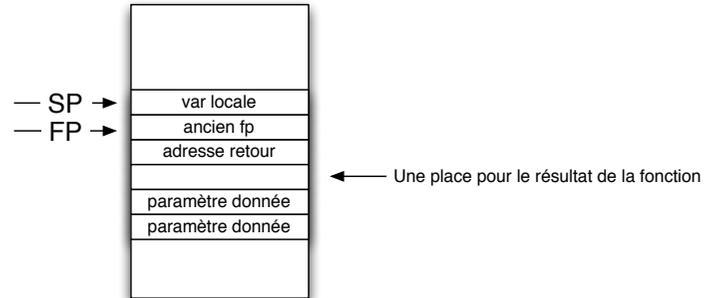
```
int fact(int x) {
    int loc, r;
    if (x==0) { r = 1; }
    else { loc = fact(x-1); r = x * loc; }
    return r;
}

main() {
    int n, y;
    ...
    y = fact(n);
    ...
}
```

## Résultat dans la pile ( $\mathcal{CT}_5$ ) (3/3)

Lors de l'exécution du corps de la fonction.

- 1 Les variables locales sont accessibles par une adresse de la forme :  $fp - 4 - depl$  avec  $depl \geq 0$ ,
- 2 Les paramètres donnés par les adresses :  $fp + 8 + 4$  et  $fp + 8 + 8$  et
- 3 La case résultat par l'adresse  $fp + 8$ .



## Variables temporaires

**Problème :**

- Les registres utilisés par une procédure ou une fonction pour des calculs intermédiaires locaux sont modifiés
- Or il serait sain de les retrouver inchangés après un appel de procédure ou fonction

**Solution :**

- Sauvegarder les registres utilisés : r0, r1, r2... **dans la pile.**
- Et cela doit être fait **avant** de les modifier donc en tout début du code de la procédure ou fonction.

## Convention générique $CG$ à retenir à l'issue du cours

On aboutit enfin à une convention **générique** permettant de générer du code de **façon systématique**.

↪ la **convention  $CG$** , à utiliser par la suite.

## Structure générale du code d'un appel et du corps de la fonction ou procédure $CG$

**appelant P :**

- 1) préparer et empiler les paramètres (valeurs et/ou adresses)
- 2) si fonction, réserver une place dans la pile pour le résultat
- 3) appeler Q : BL Q
- 4) si fonction, récupérer le résultat
- 5) libérer la place allouée aux paramètres
- 6) si fonction, libérer la place allouée au résultat

**appelée Q :**

- 1) empiler l'adresse de retour (lr)
- 2) empiler la valeur fp de l'appelant
- 3) placer fp pour repérer les variables de l'appelée
- 4) allouer la place pour les variables locales
- 5) empiler les variables temporaires (registres) utilisées
- 6) **corps de la fonction**
- 7) si fonction, le résultat est rangé en **fp+8**
- 8) dépiler les variables temporaires (registres) utilisées
- 9) libérer la place allouée aux variables locales
- 10) dépiler fp
- 11) dépiler l'adresse de retour (lr)
- 12) retour à l'appelant : BX lr

## Comparaison entre $CG$ et $C_{gcc}$

### Notion d'ABI (*Application Binary Interface*)

La convention utilisée pour les appels de fonction fait partie de l'ABI :

- un contrat entre **appelante** et **appelée**
  - ↪ qui est responsable de sauvegarder les registres ?
  - ↪ de réserver l'espace ?
- un compilateur doit respecter l'ABI pour que le code produit fonctionne
  - ↪ en particulier si on utilise des fonctions de bibliothèques externes !

La convention  $CG$  est différente de la convention  $C_{gcc}$  que l'on manipulera en TP.

↪ voir plus loin dans le cours pour un résumé de la convention  $C_{gcc}$

↪ en TD, et aux examens, on manipulera la convention  $CG$ , **à moins qu'on ne précise autre chose**

## Remarque : des fois, ça marche ou pas ?

Comment faire +1 sur le premier élément d'un tableau ?

- Par procédure :

```
procedure inc (t : tableau d'entiers)
    t [0] = t [0] +1;
```

```
Ns : tableau d'entiers
inc(Ns);
```

- Cette fois, ça marche !
- Ns (ou t) sont des références ...
- C'est la suite du drame du passage de paramètre par valeur

# La vie des programmes

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



# Plan

- 1 Introduction
- 2 Synthèse
- 3 Compilation haut niveau
- 4 Compilation assembleur
- 5 Editeur de liens

# Aujourd'hui

Nous allons étudier en détail **les différentes étapes de compilation** permettant de produire un exécutable à partir d'un ou plusieurs fichiers sources.

**Remarque** : lorsque l'on compile plusieurs fichiers sources en un seul exécutable, on parle de **compilation séparée**.

# Analyse et synthèse

La compilation comporte deux phases :

- 1 Phase d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
- 2 Phase de synthèse de code
  - Génération de code intermédiaire
  - Optimisation de code intermédiaire
  - Génération de code cible

Dans ce cours, nous nous préoccupons surtout de la seconde phase.

# Compilation et interprétation

L'exécution d'un programme peut être effectuée via :

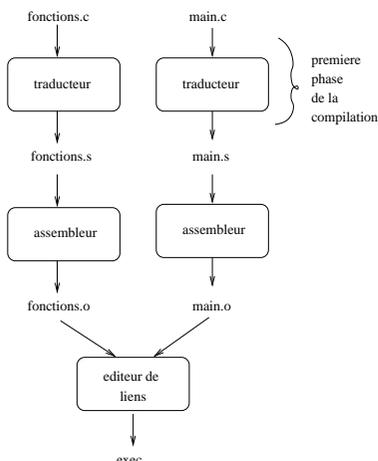
- un compilateur (le programme est transformé en langage machine par le compilateur, puis chargé en mémoire vive par le chargeur et exécuté par la machine)
- un interpréteur (le programme est transformé et interprété par l'interpréteur [qui s'exécute sur la machine])

Compilateurs et interpréteurs partagent la première phase de travail (phase d'analyse).

Compilateurs et interpréteurs se distinguent au moment de l'exécution :

- le code cible produit par un compilateur est exécuté directement par la machine cible
- la structure intermédiaire obtenue par l'interpréteur est exécutée par l'interpréteur lui-même (comme sur une machine virtuelle)

# Compilation haut niveau



Remarque : la phase de traduction comporte une phase de pré-traitement dite phase de « pré-compilation » où le code source est transformé en code source « enrichi » (les directives de pré-compilations — # — sont traitées)

- Pour « compiler », produire un exécutable, on enchaîne les commandes :
  - gcc -c fonctions.c
  - gcc -c main.c
  - gcc -o exec main.o fonctions.o
- La commande gcc -c main.c produit un fichier appelé main.o.
- La commande gcc -c fonctions.c produit un fichier fonctions.o.
- Les fichiers fonctions.o et main.o contiennent du **binaire translatable**, c'est-à-dire, du code binaire qui ne peut pas directement être exécuté en mémoire.
- La commande gcc -o exec main.o fonctions.o produit le fichier exec qui contient du **binaire exécutable**. Ce fichier résulte de la liaison des deux fichiers objets (.o). On parle d'**édition de liens**.
- **Remarque** : gcc cache l'appel à différents outils (logiciels).

# Un exemple en langage C

```

/* fichier fonctions.c */
int somme (int *t, int n) {
    int i, s;
    s = 0;
    for (i=0;i<n;i++) s = s + t[i];
    return (s); }

int max (int *t, int n) {
    int i, m;
    m = t[0];
    for (i=1;i<n;i++) if (m < t[i]) m = t[i];
    return (m); }

=====

/* fichier main.c */
extern int somme (int *t, int n);
extern int max (int *t, int n);

#define TAILLE 10
static int TAB [TAILLE];

main () {
    int i,u,v;
    for (i=0;i<TAILLE;i++) scanf ("%d", &TAB[i]);
    u = somme (TAB, TAILLE);
    v = max (TAB, TAILLE); }
    
```

- Dans le fichier main.c les fonctions somme et max sont dites **importées** : elles sont définies dans un autre fichier.
- Dans le fichier fonctions.c, somme et max sont dites **exportées** : elles sont utilisables dans un autre fichier.

# Exemple avec ARM : essai.s et lib.s

essai.s

```

.text
.global main
main:
    mov r0, #0
bcle: cmp r0, #10
    beq fin
    ldr r3, LD_xx
    ldr r2, [r3]
    bl add1
    str r2, [r3]
    add r0, r0, #1
    b bcle
fin:  bx lr
LD_xx: .word xx
.data
.word 99
xx: .word 3
    
```

lib.s

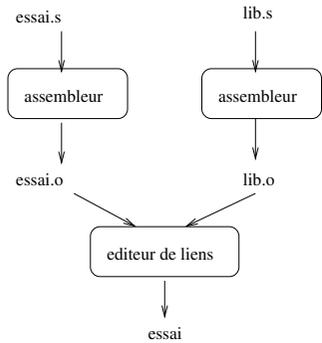
```

.text

.global add1

add1 : add r2, r2, #1
    bx lr
    
```

## Compilation en assembleur



- Pour « compiler », on enchaîne les commandes :
  - `arm-eabi-gcc -c essai.s`
  - `arm-eabi-gcc -c lib.s`
  - `arm-eabi-gcc -o essai essai.o lib.o`
- Les commandes `arm-eabi-gcc -c essai.s` et `arm-eabi-gcc -c lib.s` produisent les fichiers `essai.o` et `lib.o`.
- Les fichiers `essai.o` et `lib.o` contiennent du **binaire translatable**.
- La commande `arm-eabi-gcc -o essai essai.o lib.o` produit le fichier `essai` qui contient du **binaire exécutable**. Ce fichier résulte de la liaison des deux fichiers **objets** (.o). On parle d'**édition de liens**.
- **Remarque** : `arm-eabi-gcc` cache différents outils.
  - La commande `arm-eabi-gcc` appliqué à un fichier `.s` avec l'option `-c` correspond à la commande `arm-eabi-as`.
  - La commande `arm-eabi-gcc` avec l'option utilisée avec `-o` correspond à la commande d'édition de liens `arm-eabi-ld`.

## Exemple : traduction d'une conditionnelle

Prenons une conditionnelle :

```
si Condition alors { ListeInstructions }
```

elle sera traduite selon le schéma (récurusif) :

```

etiq_debut:
    traduction(Condition) avec positionnement de ZNCV
    branch si (non vérifiée) a etiq_suite
    traduction(ListeInstruction)
etiq_suite:
  
```

## Du langage C vers l'assembleur (ou un code intermédiaire)

- L'objectif d'un langage de haut niveau type langage C est de permettre au programmeur de **s'abstraire** des détails inhérents au fonctionnement de la machine.
- Il permet de manipuler des concepts bien plus **élaborés**.
- Mais il empêche la gestion de certains de ces détails.
- La première phase de la compilation consiste en la **traduction systématique** d'une syntaxe complexe en un langage plus simple et plus proche de la machine (langage machine ou code intermédiaire).

## Les schémas de traduction

Il y a ainsi, des schémas (récuratifs) de traduction prévus pour toutes les règles de grammaire décrivant les concepts du langage de programmation. Ces schémas sont définis pour un type de machine (large).

Exemples de schémas :

- pour l'évaluation d'opérateurs arithmétiques
- pour l'évaluation d'opérateurs relationnels
- pour l'affectation
- pour les structures de contrôle
- pour la définition de fonctions
- etc.



## Que contient un fichier .s ?

```
.text
.global main
main:
    mov r0, #0
bcle: cmp r0, #10
    beq fin
        ldr r3, LD_xx
        ldr r2, [r3]
        bl add1
            str r2, [r3]
            add r0, r0, #1
        b cble
fin:    bx lr
LD_xx: .word xx
        .data
        .word 99
xx:    .word 3
```

- **des directives** : .data, .bss, .text, .word, .hword, .byte, .skip, .asciz, .align
- **des étiquettes** appelées aussi symboles
- **des instructions** du processeur
- **des commentaires** :@ blabla

**Note** : Parfois une directive (.org) permet de fixer l'adresse où sera logé le programme en mémoire. Cette facilité permet alors de calculer certaines adresses dès la phase d'assemblage.

## Exemple : essai.o, entête

On obtient l'entête avec la commande `arm-eabi-readelf -a essai.o`.

```
ELF Header:
Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
Class:                   ELF32
Data:                     2's complement, little endian
Version:                  1 (current)
OS/ABI:                   ARM
ABI Version:              0
Type:                     REL (Relocatable file)
Machine:                  ARM
Version:                  0x1
Entry point address:      0x0
Start of program headers: 0 (bytes into file)
Start of section headers: 184 (bytes into file)
Flags:                    0x0
Size of this header:      52 (bytes)
Size of program headers:  0 (bytes)
Number of program headers: 0
Size of section headers:  40 (bytes)
Number of section headers: 9
Section header string table index: 6
```

## Que contient un fichier .o ?

- **un entête** contenant la taille du fichier, les adresses des différentes tables, la taille de la zone de données non initialisées (bss), etc.
- **la zone de données** (data) parfois incomplète
- **la zone des instructions** (text) parfois incomplète
- les informations associées à chaque symbole rangées dans une section appelée : **table des symboles**.
- les informations permettant de compléter ce qui n'a pu être calculé... On les appelle **informations de translation** et l'ensemble de ces informations est rangé dans une section particulière appelée **table de translation**.
- une **table des chaînes** à laquelle la table des symboles fait référence.

## Exemple : essai.o, organisation des tables

On obtient l'entête avec la commande `arm-eabi-readelf -a essai.o (suite)`.

```
Section Headers:
[Nr] Name           Type           Addr      Off      Size    ES Flg Lk  Inf Al
[ 0]                NULL          00000000 000000 000000 00   0  0  0
[ 1] .text            PROGBITS      00000000 000034 00002c 00  AX  0  0  1
[ 2] .rel.text       REL           00000000 00033c 000018 08   7  1  4
[ 3] .data           PROGBITS      00000000 000060 000008 00  WA  0  0  1
[ 4] .bss            NOBITS       00000000 000068 000000 00  WA  0  0  1
[ 5] .ARM.attributes ARM_ATTRIBUTES 00000000 000068 000010 00   0  0  1
[ 6] .shstrtab       STRTAB        00000000 000078 000040 00   0  0  1
[ 7] .symtab         SYMTAB        00000000 000220 0000f0 10   8 12  4
[ 8] .strtab         STRTAB        00000000 000310 000029 00   0  0  1
```

Key to Flags:  
W (write), A (alloc), X (execute), M (merge), S (strings)  
I (info), L (link order), G (group), x (unknown)  
0 (extra OS processing required) o (OS specific), p (processor specific)

## Exemple : essai.o, zone data

On obtient la zone `.data` avec la commande `arm-eabi-objdump -s -j .data essai.o`.

```
essai.o:      format de fichier elf32-littlearm
```

```
Contenu de la section .data:
0000 63000000 03000000
```

## Exemple : essai.o, zone text

La zone `.text` avec désassemblage avec la commande `arm-eabi-objdump -j .text -d essai.o`.

Disassembly of section `.text`:

```
00000000 <main>:
  0: e3a00000  mov r0, #0

00000004 <bcle>:
  4: e350000a  cmp r0, #10
  8: 0a000005  beq 24 <fin>
  c: e59f3014  ldr r3, [pc, #20] ; 28 <LD_xx>
 10: e5932000  ldr r2, [r3]
 14: ebfffffe  bl 0 <add1>
 18: e5832000  str r2, [r3]
 1c: e2800001  add r0, r0, #1
 20: eafffff7  b 4 <bcle>

00000024 <fin>:
 24: eafffffe  b 0 <exit>

00000028 <LD_xx>:
 28: 00000004  .word 0x00000004
```

## Exemple : essai.o, zone text

On obtient la zone `.text` avec la commande `arm-eabi-objdump -j .text -s essai.o`.

```
essai.o:      format de fichier elf32-littlearm
```

```
Contenu de la section .text:
0000 0000a0e3 0a0050e3 0500000a 14309fe5
0010 002093e5 feffffeb 002083e5 010080e2
0020 f7ffffea feffffea 04000000
```

## Rappel : essai.o, organisation des tables

On obtient l'entête avec la commande `arm-eabi-readelf -a essai.o (suite)`.

```
Section Headers:
[Nr] Name           Type           Addr           Off           Size           ES Flg Lk  Inf Al
[ 0]                 NULL          00000000      000000      000000 00      0  0  0
[ 1] .text             PROGBITS      00000000      000034      00002c 00  AX  0  0  1
[ 2] .rel.text         REL           00000000      00033c      000018 08      7  1  4
[ 3] .data             PROGBITS      00000000      000060      000008 00  WA  0  0  1
[ 4] .bss              NOBITS        00000000      000068      000000 00  WA  0  0  1
[ 5] .ARM.attributes   ARM_ATTRIBUTES 00000000      000068      000010 00      0  0  1
[ 6] .shstrtab         STRTAB        00000000      000078      000040 00      0  0  1
[ 7] .symtab           SYMTAB        00000000      000220      0000f0 10      8 12  4
[ 8] .strtab           STRTAB        00000000      000310      000029 00      0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
```





## Que contient un fichier exécutable ? organisation des tables

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	000000	000000	00		0	0	0
[ 1]	.init	PROGBITS	00008000	008000	000020	00	AX	0	0	4
[ 2]	.text	PROGBITS	00008020	008020	002500	00	AX	0	0	4
[ 3]	.fini	PROGBITS	0000a520	00a520	00001c	00	AX	0	0	4
[ 4]	.rodata	PROGBITS	0000a53c	00a53c	00000c	00	A	0	0	4
[ 5]	.eh_frame	PROGBITS	0000a548	00a548	00083c	00	A	0	0	4
[ 6]	.ctors	PROGBITS	00012d84	00ad84	000008	00	WA	0	0	4
[ 7]	.dtors	PROGBITS	00012d8c	00ad8c	000008	00	WA	0	0	4
[ 8]	.jcr	PROGBITS	00012d94	00ad94	000004	00	WA	0	0	4
[ 9]	.data	PROGBITS	00012d98	00ad98	00095c	00	WA	0	0	4
[10]	.bss	NOBITS	000136f4	00b6f4	000108	00	WA	0	0	4
[11]	.comment	PROGBITS	00000000	00b6f4	0001e6	00		0	0	1
[12]	.debug_aranges	PROGBITS	00000000	00b8e0	000350	00		0	0	8
[13]	.debug_pubnames	PROGBITS	00000000	00bc30	00069c	00		0	0	1
[14]	.debug_info	PROGBITS	00000000	00c2cc	00ea53	00		0	0	1
[15]	.debug_abbrev	PROGBITS	00000000	01ad1f	002956	00		0	0	1
[16]	.debug_line	PROGBITS	00000000	01d675	002444	00		0	0	1
[17]	.debug_str	PROGBITS	00000000	01fab9	001439	01	MS	0	0	1
[18]	.debug_loc	PROGBITS	00000000	020ef2	002163	00		0	0	1
[19]	.debug_ranges	PROGBITS	00000000	023058	0002e8	00		0	0	8
[20]	.ARM.attributes	ARM_ATTRIBUTES	00000000	023340	000010	00		0	0	1
[21]	.shstrtab	STRTAB	00000000	023350	0000df	00		0	0	1
[22]	.symtab	SYMTAB	00000000	0237f0	0013e0	10		23	213	4
[23]	.strtab	STRTAB	00000000	024bd0	0007d1	00		0	0	1

## Que contient un fichier exécutable ? table des symboles

Symbol table '.symtab' contains 318 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00008000	0	SECTION	LOCAL	DEFAULT	1	
2:	00008020	0	SECTION	LOCAL	DEFAULT	2	
3:	0000a520	0	SECTION	LOCAL	DEFAULT	3	
.....							
9:	00012ea8	0	SECTION	LOCAL	DEFAULT	9	
.....							
74:	0000821c	0	NOTYPE	LOCAL	DEFAULT	2	bcle
75:	0000823c	0	NOTYPE	LOCAL	DEFAULT	2	fin
76:	00008240	0	NOTYPE	LOCAL	DEFAULT	2	LD_xx
77:	00008240	0	NOTYPE	LOCAL	DEFAULT	2	\$d
78:	00012eb4	0	NOTYPE	LOCAL	DEFAULT	9	xx
.....							
230:	00008244	0	NOTYPE	GLOBAL	DEFAULT	2	add1
.....							

## Que contient un fichier exécutable ? section data

Contents of section .data:

```
.....
12ea8 00000000 00000000 63000000 03000000
.....
```

## Que contient un fichier exécutable ? section text

```
00008218 <main>:
8218: e3a00000 mov r0, #0

0000821c <bcle>:s
821c: e350000a cmp r0, #10
8220: 0a000005 beq 823c <fin>
8224: e59f3014 ldr r3, [pc, #20] ; 8240 <LD_xx>
8228: e5932000 ldr r2, [r3]
822c: eb000004 bl 8244 <add1>
8230: e5832000 str r2, [r3]
8234: e2800001 add r0, r0, #1
8238: eafffff7 b 821c <bcle>

0000823c <fin>:
823c: ea000007 b 8260 <exit>

00008240 <LD_xx>:
8240: 00012eb4 .word 0x00012eb4

00008244 <add1>:
8244: e2822001 add r2, r2, #1
8248: e1a0f00e bx lr
```

## Organisation interne d'un ordinateur

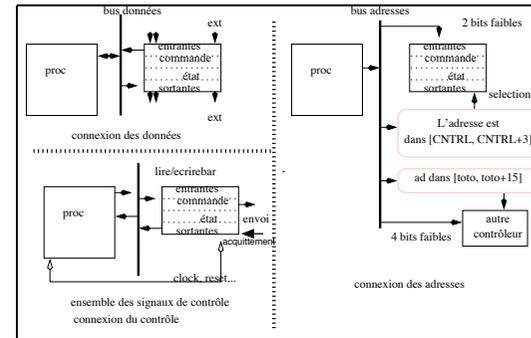
Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025

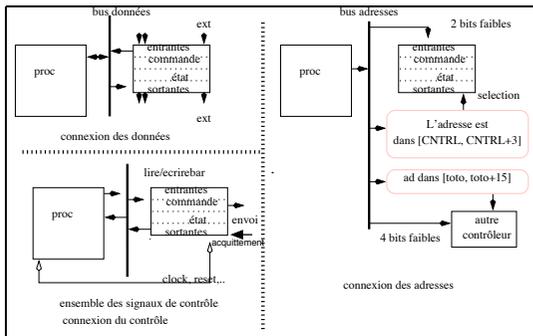


## Etude du matériel d'entrées-sorties : les entrées



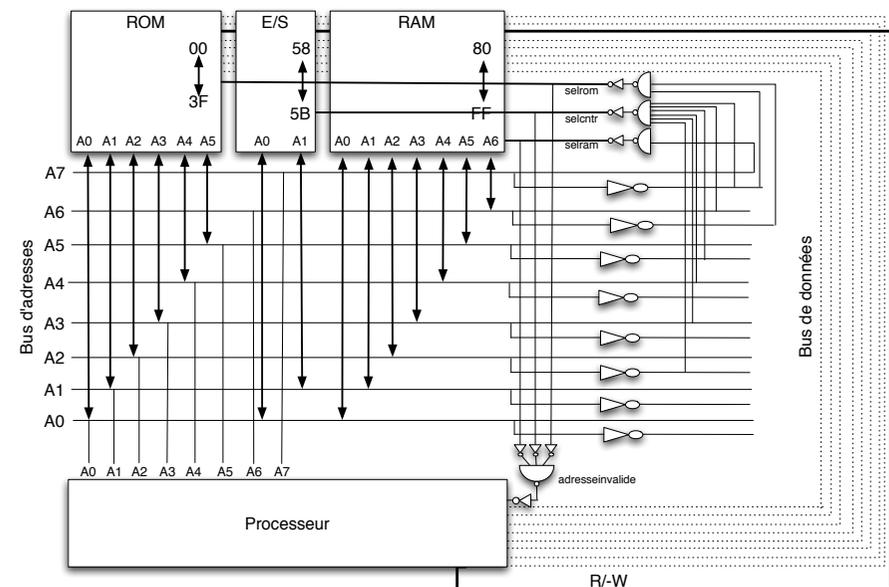
- bus données (lié au processeur)
- deux bits de bus adresses (pour sélectionner l'un des 4 mots CNTRL +0, +1, +2 ou +3)
- un signal de sélection provenant du **décodeur d'adresses**
- le signal *Read/Write* du processeur
- un paquet de données (8 fils) venant du monde extérieur. Disons pour simplifier 8 interrupteurs
- le signal d'horloge (par exemple le même que le processeur). On peut raisonner comme si, à chaque front de l'horloge la valeur venant des interrupteurs était échantillonnée dans le registre  $M_{donnéesent}$ .
- une entrée **ACQUITTEMENT** si c'est un contrôleur de sortie.

## Etude du matériel d'entrées-sorties : les sorties



- Il délivre sur le bus données du processeur le contenu du registre  $M_{donnéesent}$  si il y a **sélection, lecture et adressage** de  $M_{donnéesent}$ , c'est-à-dire si le processeur exécute une instruction LOAD à l'adresse CNTRL +3
- Il délivre sur le bus données du processeur le contenu du registre Métat si il y a **sélection, lecture et adressage** de Métat, c'est-à-dire si le processeur exécute une instruction LOAD à l'adresse CNTRL +1.
- On peut raisonner comme si le contenu du registre  $M_{donnéesort}$  était affiché en permanence sur 8 pattes de sorties vers l'extérieur (8 diodes, par exemple).
- Une sortie **ENVOI** si c'est un contrôleur de sortie.

## Connexions processeur/contrôleur/mémoires/décodeur



# Introduction à la structure interne des processeurs : une machine à 5 instructions

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



## Codage des instructions

Les instructions sont codées sur **1 ou 2 mots de 4 bits** chacuns :

- le premier mot représente le code de l'opération : `clr`, `ld`, `st`, `jmp`, `add`);
- le deuxième mot, s'il existe, contient une **opérande** (une adresse ou bien une constante).

Le codage est le suivant :

<code>clr</code>	1	
<code>ld #vi</code>	2	<code>vi</code>
<code>st ad</code>	3	<code>ad</code>
<code>jmp ad</code>	4	<code>ad</code>
<code>add ad</code>	5	<code>ad</code>

## Les instructions

Les instructions sont décrites ci-dessous. On donne pour chacune une **syntaxe de langage d'assemblage**, ainsi que l'effet (la **sémantique**) de l'instruction.

- `clr` : mise à zéro du registre ACC.
- `ld #vi` : chargement de la valeur immédiate `vi` dans ACC.
- `st ad` : rangement en mémoire à l'adresse `ad` du contenu de ACC.
- `jmp ad` : saut à l'adresse `ad`.
- `add ad` : mise à jour de ACC avec la somme du contenu de ACC et du mot mémoire d'adresse `ad`.

## Algorithme d'interprétation

En adoptant un point de vue fonctionnel, en considérant les ressources du processeur comme les variables d'un programme, l'algorithme d'interprétation des instructions peut être décrit de la façon suivante :

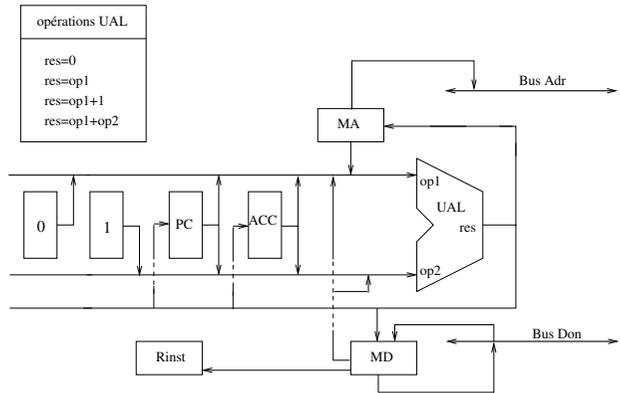
```

pc ← 0
tantque vrai
    selon mem[pc]
        mem[pc]=1 {clr} : acc ← 0                pc ← pc+1
        mem[pc]=2 {ld} : acc ← mem[pc+1]         pc ← pc+2
        mem[pc]=3 {st} : mem[mem[pc+1]] ← acc   pc ← pc+2
        mem[pc]=4 {jmp} :                       pc ← mem[pc+1]
        mem[pc]=5 {add} : acc ← acc + mem[mem[pc+1]] pc ← pc+2
    
```

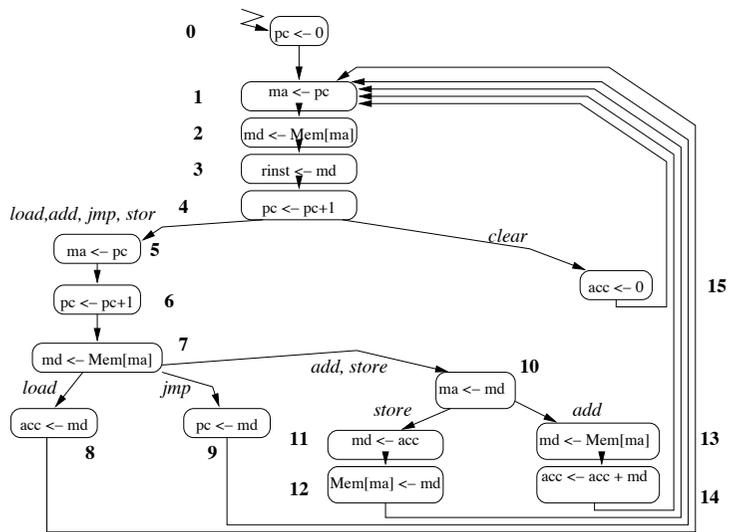
**Exercice** : Dérouler l'exécution du programme précédent en utilisant cet algorithme.

# Partie opérative

Le processeur comporte une partie qui permet de stocker des informations dans des registres (visibles ou non du programmeur), de faire des calculs (+, -, &, ...). Cette partie est reliée à la mémoire par **les bus adresses et données**.  
On l'appelle **Partie Opérative** (ou PO).



# Version amélioré de l'automate



Version améliorée de l'automate d'interprétation du langage machine (**Partie Contrôle**)

# Micro-actions et micro-conditions

On fait des hypothèses **FORTES** sur les transferts possibles :

$md \leftarrow mem[ma]$	lecture d'un mot mémoire.	C'est la seule possibilité en lecture!
$mem[ma] \leftarrow md$	écriture d'un mot mémoire	C'est la seule possibilité en écriture!
$rinst \leftarrow md$	affectation	C'est la seule affectation possible dans $rinst$
$reg_0 \leftarrow 0$	affectation	$reg_0$ est pc, acc, ma, ou md
$reg_0 \leftarrow reg_1$	affectation	$reg_0$ est pc, acc, ma, ou md
$reg_0 \leftarrow reg_1 + 1$	incrémementation	$reg_0$ est pc, acc, ma, ou md $reg_1$ est pc, acc, ma, ou md
$reg_0 \leftarrow reg_1 + reg_2$	opération	$reg_0$ est pc, acc, ma, ou md $reg_1$ est pc, acc, ma, ou md $reg_2$ est pc, acc, ou md

On fait aussi des hypothèses sur les tests : ( $rinst = entier$ )

Ces types de transferts et les tests constituent **le langage des micro-actions et des micro-conditions**.

# Optimisations

Bruno Ferres Kevin Marquet Denis Bouhineau  
basé sur un cours de Fabienne Carrier & Stéphane Devismes

Université Grenoble Alpes

25 avril 2025



# Plan

- 1 Arithmétique
- 2 Structures de contrôle
- 3 Fonctions
- 4 Pipeline
- 5 Mémoire cache
- 6 Conclusions

# Résultats connus

(2/2)

- Calcul avec des constantes (version optimisée)

```
#include <stdio.h>

int main () {
    int i=4, j=5;
    printf("4+5=%d\n",i+j);
    return;}

.file "progConstantes.c"
.text
.align 2
.global main
.type main, %function
main:
stmfd sp!, {r3, lr}
ldr r0, .L2
mov r1, #9
bl printf
ldmfd sp!, {r3, lr}
bx lr
.L3:
.align 2
.L2:
.word .LC0
.size main, .-main
.section .rodata.str1.4,"aMS",%progbits,1
.align 2
.LC0:
.ascii "4+5=%d\n000"
.ident "GCC: (GNU) 4.5.3"
```

# Résultats connus

(1/2)

- Calcul avec des constantes (version attendue)

```
#include <stdio.h>

int main () {
    int i=4, j=5;
    printf("4+5=%d\n",i+j);
    return;}

.file "progConstantes.c"
.section .rodata
.align 2
.LC0:
.ascii "4+5=%d\n012\000"
.text
.align 2
.global main
.type main, %function
main:
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #8
mov r3, #4
str r3, [fp, #-8]
mov r3, #5
str r3, [fp, #-12]
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
add r3, r2, r3
ldr r0, .L2
mov r1, r3
bl printf
mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
bx lr
```

# Opérations équivalentes

- Divisions équivalentes

## Division arbitraire

- Division arbitraire (version attendue)

```
#include <stdio.h>

int main () {
    int i,j;
    scanf("%d",&i);
    scanf("%d",&j);
    printf("%d / %d=%d\n",i,j,i/j);
    return;}

main:
stmfd sp!, {r4, r5, fp, lr}
add fp, sp, #12
sub sp, sp, #8
sub r3, fp, #16
ldr r0, .L2
mov r1, r3
bl scanf
sub r3, fp, #20
ldr r0, .L2
mov r1, r3
bl scanf
ldr r5, [fp, #-16]
ldr r4, [fp, #-20]
ldr r2, [fp, #-16]
ldr r3, [fp, #-20]
mov r0, r2
mov r1, r3
bl __aeabi_idiv
mov r3, r0
ldr r0, .L2+4
mov r1, r5
mov r2, r4
bl printf
mov r0, r3
sub sp, fp, #12
ldmfd sp!, {r4, r5, fp, lr}
```

## Arithmétique (conclusions)

- Les expressions calculées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les expressions sous forme logiques / littérales
- Le compilateur optimise !
- Gain espéré : temps de calcul

## Division par 2

- Division par 2 (version optimisée)

```
#include <stdio.h>

int main () {
    int i;
    scanf("%d",&i);
    printf("%d / 2 =%d\n",i,i/2);
    return;}

main:
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #8
sub r3, fp, #8
ldr r0, .L2
mov r1, r3
bl scanf
ldr r2, [fp, #-8]
ldr r3, [fp, #-8]
mov r1, r3, lsr #31
add r3, r1, r3
mov r3, r3, asr #1
ldr r0, .L2+4
mov r1, r2
mov r2, r3
bl printf
mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
bx lr
.L3:
.align 2
.L2:
.word .LC0
.word .LC1
.size main, .-main
.ident "GCC: (GNU) 4.5.2"
```

## Conditionnelles

(1/2)

- Conditionnelle (version attendue)
- (version assembleur X86)

```
#include <stdio.h>

int main () {
    int i=4;
    if (i&1) {
        printf("4 est impair\n");}
    else {
        printf("4 est pair\n");}
    return 0;}

main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rsp
movl $4, -4(%rbp)
movl -4(%rbp), %eax
andl $1, %eax
testl %eax, %eax
je .L2
movl $.LC0, %edi
call puts
jmp .L3
.L2:
movl $.LC1, %edi
call puts
.L3:
movl $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```



## Fonctions en ligne (inline)

(1/2)

## ● Fonction (version attendue)

```
#include <stdio.h>
```

```
int somme(int a,int b) {
    return a+b;}

```

```
int main () {
    int i, j;
    scanf("%d",&i);
    scanf("%d",&j);
    printf("4+5=%d\n",somme(i,j));
    return;}

```

```
somme:
[...] movl -8(%rbp), %eax
movl -4(%rbp), %edx
addl %edx, %eax
popq %rbp
ret
[...] main:
[...] movl $0, %eax
call __isoc99_scanf
leaq -4(%rbp), %rax
movq %rax, %rsi
movl $.LC0, %edi
movl $0, %eax
call __isoc99_scanf
movl -4(%rbp), %edx
movl -8(%rbp), %eax
movl %edx, %esi
movl %eax, %edi
call somme
movl %eax, %esi
movl $.LC1, %edi
movl $0, %eax
call printf
nop
leave
ret
```

## Fonctions en ligne (inline)

(2/2)

## ● Fonction (version optimisée)

```
#include <stdio.h>
```

```
int somme(int a,int b) {
    return a+b;}

```

```
int main () {
    int i, j;
    scanf("%d",&i);
    scanf("%d",&j);
    printf("4+5=%d\n",somme(i,j));
    return;}

```

```
somme:
[...]
leal (%rdi,%rsi), %eax
ret
main:
[...]
movl $.LC0, %edi
movl $0, %eax
call __isoc99_scanf
leaq 12(%rsp), %rsi
movl $.LC0, %edi
movl $0, %eax
call __isoc99_scanf
movl 12(%rsp), %edx
addl 8(%rsp), %edx
movl $.LC1, %esi
movl $1, %edi
movl $0, %eax
call __printf_chk
addq $24, %rsp
ret
```

## Fonctions récursives

(1/6)

- Fonction récursive (version attendue)
- Somme (version attendue)

```
int somme(int a,int s) {
    if (!a) {
        return s;}
    else {
        return somme(a-1,a+s);}}
```

```
int main () {
    int i;
    scanf("%d",&i);
    printf("1+2+...+%d=%d\n",i,somme(i,0));
    return 0;}

```

```
somme:
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #8
str r0, [fp, #-8]
str r1, [fp, #-12]
ldr r3, [fp, #-8]
cmp r3, #0
bne .L2
ldr r3, [fp, #-12]
b .L3
.L2:
ldr r3, [fp, #-8]
sub r2, r3, #1
ldr r1, [fp, #-8]
ldr r3, [fp, #-12]
add r3, r1, r3
mov r0, r2
mov r1, r3
bl somme
mov r3, r0
.L3:
mov r0, r3
sub sp, fp, #4
ldmfd sp!, {fp, lr}
bx lr
```

## Fonctions récursives

(2/6)

- Fonction récursive (version attendue)
- Main (version attendue)

```
int somme(int a,int s) {
    if (!a) {
        return s;}
    else {
        return somme(a-1,a+s);}}
```

```
int main () {
    int i;
    scanf("%d",&i);
    printf("1+2+...+%d=%d\n",i,somme(i,0));
    return 0;}

```

```
main:
stmfd sp!, {r4, fp, lr}
add fp, sp, #8
sub sp, sp, #12
sub r3, fp, #16
ldr r0, .L5
mov r1, r3
bl scanf
ldr r4, [fp, #-16]
ldr r3, [fp, #-16]
mov r0, r3
mov r1, #0
bl somme
mov r3, r0
ldr r0, .L5+4
mov r1, r4
mov r2, r3
bl printf
mov r3, #0
mov r0, r3
sub sp, fp, #8
ldmfd sp!, {r4, fp, lr}
bx lr
```

## Fonctions récursives

(3/6)

- Fonction récursive (version optimisée)
- Somme (version optimisée)

```
int somme(int a,int s) {
    if (!a) {
        return s;
    }
    else {
        return somme(a-1,a+s);
    }
}

int main () {
    int i;
    scanf("%d",&i);
    printf("1+2+...+%d=%d\n",i,somme(i,0));
    return 0;
}
```

```
somme:
stmfd sp!, {r3, lr}
subs r3, r0, #0
beq .L2
sub r0, r3, #1
add r1, r1, r3
bl somme
mov r1, r0
.L2:
mov r0, r1
ldmfd sp!, {r3, lr}
bx lr
```

## Fonctions récursives

(5/6)

- Fonction récursive (version + optimisée)
- Somme (version + optimisée)

```
int somme(int a,int s) {
    if (!a) {
        return s;
    }
    else {
        return somme(a-1,a+s);
    }
}

int main () {
    int i;
    scanf("%d",&i);
    printf("1+2+...+%d=%d\n",i,somme(i,0));
    return 0;
}
```

```
somme:
subs r3, r0, #0
bne .L5
b .L2
.L4:
mov r3, r2
.L5:
subs r2, r3, #1
add r1, r1, r3
bne .L4
.L2:
mov r0, r1
bx lr
```

## Fonctions récursives

(4/6)

- Fonction récursive (version optimisée)
- Main (version optimisée)

```
int somme(int a,int s) {
    if (!a) {
        return s;
    }
    else {
        return somme(a-1,a+s);
    }
}

int main () {
    int i;
    scanf("%d",&i);
    printf("1+2+...+%d=%d\n",i,somme(i,0));
    return 0;
}
```

```
main:
stmfd sp!, {r4, lr}
sub sp, sp, #8
ldr r0, .L4
add r1, sp, #4
bl scanf
ldr r4, [sp, #4]
mov r0, r4
mov r1, #0
bl somme
mov r2, r0
ldr r0, .L4+4
mov r1, r4
bl printf
mov r0, #0
add sp, sp, #8
ldmfd sp!, {r4, lr}
bx lr
```

## Fonctions récursives

(6/6)

- Fonction récursive (version + optimisée)
- Main (version + optimisée)

```
int somme(int a,int s) {
    if (!a) {
        return s;
    }
    else {
        return somme(a-1,a+s);
    }
}

int main () {
    int i;
    scanf("%d",&i);
    printf("1+2+...+%d=%d\n",i,somme(i,0));
    return 0;
}
```

```
main:
str lr, [sp, #-4]!
sub sp, sp, #12
add r1, sp, #4
ldr r0, .L14
bl scanf
ldr r1, [sp, #4]
cmp r1, #0
movne r3, r1
movne r2, #0
bne .L9
b .L13
.L11:
mov r3, r0
.L9:
subs r0, r3, #1
add r2, r2, r3
bne .L11
.L8:
ldr r0, .L14+4
bl printf
mov r0, #0
add sp, sp, #12
ldr lr, [sp], #4
bx lr
.L13:
mov r2, r1
```

## Fonctions récursives : exemples d'exécutions (1/2)

- Somme récursive non terminale

```
int somme(int a) {
    if (!a) {
        return 0;}
    else {
        return a+somme(a-1);}}
```

```
int main () {
    int i;
    scanf("%d",&i);
    printf("1+2+...+%d=%d\n",i,somme(i));
    return 0;}
```

- Somme récursive terminale

```
int somme(int a,int s) {
    if (!a) {
        return s;}
    else {
        return somme(a-1,a+s);}}
```

```
int main () {
    int i;
    scanf("%d",&i);
    printf("1+2+...+%d=%d\n",i,somme(i,0));
    return 0;}
```

## Fonctions (conclusions)

- Les fonctions utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les fonctions les plus claires
- Le compilateur optimise !
- Gain espéré : temps d'exécution des appels (branchement+gestion paramètres)
- Gain espéré : place dans la pile (et cela peut fonctionner !)
- Gain espéré : et plus ? [branchement ... voir la suite]

## Fonctions récursives : exemples d'exécutions (2/2)

- Version récursive non terminale : plante à 300 000 (segmentation fault)
- Version récursive non terminale optimisée par le compilateur : plante à 600 000 (segmentation fault)
- Version récursive terminale : plante à 300 000 (segmentation fault)
- Version récursive terminale optimisée par le compilateur : plante à 600 000 (segmentation fault)
- Version + optimisée par le compilateur : 3 000 000 000 termes calculés en 1s

## Pipeline (rappel)

Hypothèse :

La chaîne de traitement d'une instruction comporte 5 étapes :

- IF : récupération de l'instruction
- ID : récupération des opérandes
- EX : exécution de l'opération
- MEM : écriture ou lecture mémoire
- WB : écriture registre

## Exécution de trois instructions

Exécution "simple" de trois instructions :



(source wikipedia)

- temps d'exécution : 15  $\Delta$

## Rupture de pipeline

Mais !

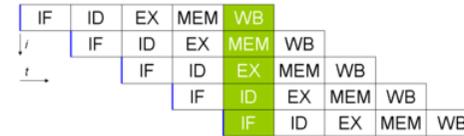
les pipelines peuvent être rompus :

- Branchement
- Dépendance entre données
- ...

## Exécution de cinq instructions

Avec pipeline

Exécution pipelinée de cinq instructions, les unes à la suite des autres :



(source wikipedia)

- temps d'exécution : 9  $\Delta$

## Pipeline et branchement

Ruptures de pipeline et branchement, cela concerne :

- Conditionnelles
- Boucles
- Appels de fonction

Le compilateur peut s'en occuper

↔ Le programmeur peut aider

## Pipeline (Conclusions)

Les processeurs modernes fonctionnent avec un **pipeline d'instructions**, afin d'optimiser les performances.

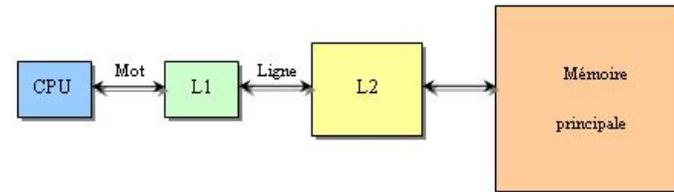
Ce fonctionnement en pipeline peut être altéré :

- rupture de séquence
  - ↔ quelle instruction ajouter dans le pipeline après un branchement conditionnel ?
- dépendance de données
  - ↔ besoin d'attendre un résultat avant d'avancer dans le pipeline

Pour gérer cela, des "bulles" sont insérées dans le pipeline, lors de l'exécution du programme.

## Organisation mémoire avec cache (rappels)

Structuration en caches de plusieurs niveaux



source wikipedia.

Pour le processeur, il n'y a qu'une mémoire : la Mémoire.

## Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :
  - ↔ l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.
- Principe de localité temporelle :
  - ↔ l'accès à une donnée X à un instant donné va probablement être suivi d'autres accès à cette même donnée
- Le cache répond aux demandes du processeur :
  - si la donnée est disponible dans le cache : ok
  - si la donnée n'est pas disponible
    - ↔ il faut aller la chercher en mémoire
    - ↔ et faire de la place dans le cache

## Optimisations (conclusions)

- Le compilateur peut optimiser
- Le programmeur peut aussi optimiser !
  - ↔ il faut travailler l'**algorithmique** et, plus tard, le **parallélisme**
- Comprendre l'exécution en regardant le code machine
  - ↔ voire, contrôler le code machine ...
- Utiliser des *profilers* pour déterminer où se trouvent les coûts
  - ↔ n'optimiser que ce qui est bloquant ...
- **Mais surtout, avant d'optimiser : programmer juste !**