

Architectures des ordinateurs (une introduction)

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

Université Grenoble Alpes

15 décembre 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 : augustin.bonnel@univ-grenoble-alpes.fr
 - INM-02 : denis.bouhineau@imag.fr (TD) & neven.villani@univ-grenoble-alpes.fr (TP);
 - INM-03 : denis.bouhineau@imag.fr;
 - INM-04 : gomezbaj@univ-grenoble-alpes.fr;
 - MIN-01 : kevin.marquet@univ-grenoble-alpes.fr;
 - MIN-02 : david.rios.uga@gmail.com;
 - MIN-03 : david.rios.uga@gmail.com;
 - MIN-04 : bruno.ferres@univ-grenoble-alpes.fr;
 - MIN-Int : olivier.romane@univ-grenoble-alpes.fr.
- **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



15 décembre 2025

Description du modèle de von Neumann

(2/5)

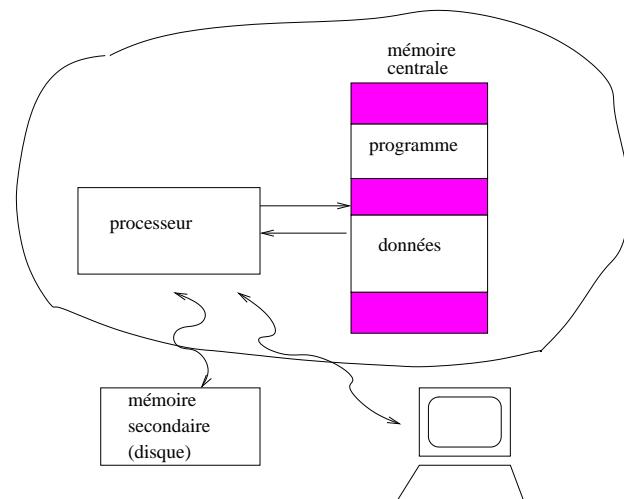


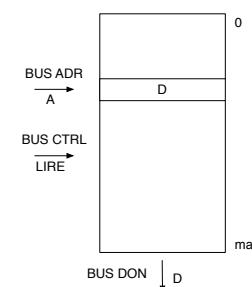
Figure – Processeur, mémoire et périphériques

Actions sur la mémoire : LIRE

La mémoire reçoit :

- un vecteur binaire (représentant une adresse A) sur le bus adresses,
- un signal de commande de lecture sur le bus de contrôle.

Elle délivre un vecteur binaire représentant la donnée D sur le bus données.

On note : $D \leftarrow \text{mem}[A]$ $\text{mem}[A]$: emplacement mémoire dont l'adresse est A

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

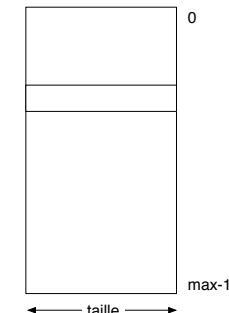
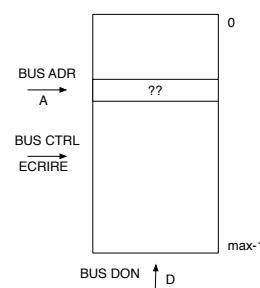


Figure – Mémoire abstraite

Actions sur la mémoire : ECRIRE

La mémoire reçoit :

- un vecteur binaire (représentant une adresse A) sur le bus adresses,
- un vecteur binaire (représentant la donnée D) sur le bus données,
- un signal de commande d'écriture sur le bus de contrôle.

Elle inscrit (*peut-être*, voir tableau ci-après) la donnée D comme contenu de l'emplacement mémoire dont l'adresse est AOn écrit : $\text{mem}[A] \leftarrow D$ **Remarque :** le bus de données est bidirectionnel

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.

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)**.

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 → extérieur

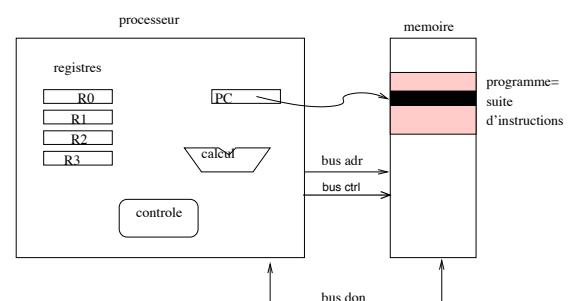
Entrée : extérieur → ordinateur

Entrée/Sortie : ordinateur ↔ extérieur

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



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

Logarithme et taille de donnée

(1/2)

On ne s'intéresse qu'à la base 2 : un chiffre binaire est appelé **bit**.

Logarithme : opération réciproque de l'élévation à la puissance

Si $Y = 2^X$, on a $X = \log_2 Y$

Pour représenter en base 2, K naturels différents

↪ il faut $\lceil \log_2 K \rceil$ chiffres en base 2

- si K est une puissance de 2 ($K = 2^N$), il faut N bits.
- si K n'est pas une puissance de 2, soit P la plus petite puissance de 2 telle que $P > K$, il faut $\log_2 P$ bits.

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 $169_{10} = 10101001_2$

Complément à deux : autre version

Comment retrouver l'opposé d'un entier A ?

① prendre $A = a_{n-1}a_{n-2}\dots a_1a_0$

② remarquer que

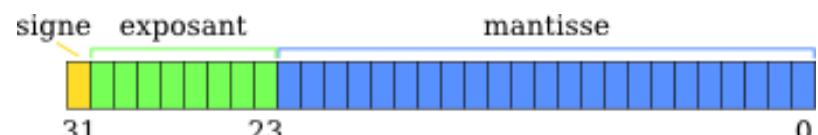
$$A + \bar{A} = 11\dots 11 = -1$$

③ en déduire que

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

Les nombres à virgule flottante

(1/2)



source Wikipedia.

Les nombres à virgule flottante (2/2)

- Norme IEEE 754
- Codage par champ (exemple sur 32 bits) :
 - Signe (1 bit), Exposant (8 bits), Mantisse (23 Bits)
- Valeur = $(-1)^{\text{signe}} * 1.\text{Mantisse} * 2^{\text{Exposant}-127}$
- Exceptions : 0, +Infini, -Infini, NaN, nombres proches de 0 ...
- Intervalle : $[-3.4 \cdot 10^{38}; 3.4 \cdot 10^{38}]$
avec la moitié des nombres entre $[-2; 2]$

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)

Récapitulatif : Pour 3 bits,

- il y a 8 vecteurs de bits possibles,
- comme entiers naturels : 0 ... 7,
- comme entiers relatifs : -4 ... 3,
- mais une seule addition.

+	000	001	010	011	100	101	110	111
000								
001								
010								
011								
100								
101								
110								
111								

+ 000 001 010 011 100 101 110 111

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

Question

À votre avis : Pour 3 bits,

- A) il n'y a pas d'erreur d'addition ni pour les naturels ni les relatifs
- B) il y a autant d'erreurs d'addition pour les naturels que pour les relatifs
- C) il y a plus d'erreurs d'addition pour les naturels que pour les relatifs
- D) il y a moins d'erreurs d'addition pour les naturels que pour les relatifs

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

Equations

Question : d'après ce qui précède, vous sauriez résoudre les équations suivantes (sur 3 bits) ?

$$x + x = 0$$

$$y + y = 1$$

$$z + z = 2$$

$$a + a = -2$$

Equations

Question : d'après ce qui précède, vous sauriez résoudre les équations suivantes (sur 3 bits) ?

$$x + x = 0$$

Réponse : 2 solutions, 000+000 et 100+100

$$y + y = 1$$

Réponse : pas de solution

$$z + z = 2$$

Réponse : 2 solutions, 001+001 et 101+101

$$a + a = -2$$

Réponse : 2 solutions (seulement C2), 011+011 et 111+111

Etapes de compilation

Langage d'assemblage, langage machine

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

Université Grenoble Alpes

15 décembre 2025

- **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
- **Édition de liens :** arm-eabi-gcc monprog.o malib.o -o monprog
un ou plusieurs fichiers objets → binaire exécutable : monprog

Précompilation (*pre-processing*)

(1/2)

```
arm-eabi-gcc -E monprog.c > monprog.i
```

produit **monprog.i**

La **précompilation** réalise plusieurs opérations de substitution sur le code, notamment :

- suppression des commentaires.
- inclusion des profils des fonctions des bibliothèques dans le fichier source.
- traitement des directives de compilation.
- remplacement des macros

Compilation

(1/2)

```
arm-eabi-gcc -S monprog.i
```

produit **monprog.s**

Le code source « enrichi » est transformé en langage d'assemblage (lisible)

→ il contient des **instructions** et des **données**.

Assemblage

(1/2)

```
arm-eabi-gcc -c monprog.s
```

produit **monprog.o**

Le code en langage d'assemblage (lisible) est transformé en **code machine**.

Le code machine se présente comme une succession de vecteurs binaires.

Le code machine ne peut pas être directement édité et lu. On peut le rendre lisible en utilisant une commande *od -t x4z monprog.o*.

Le fichier **monprog.o** contient des instructions en langage machine et des données mais il n'est pas **exécutable**.

→ On parle de binaire **translatable**.

Édition de liens

(1/2)

```
arm-eabi-gcc monprog.o malib.o -o monprog
```

produit **monprog**

L'édition de liens permet de rassembler le code de différents fichiers.

A l'issue de cette phase le fichier produit contient du **binaire exécutable**.

Remarque : ne pas confondre **exécutable**, lié à la nature du fichier, et « muni du droit d'être exécuté », lié au système d'exploitation.

Instruction de calcul entre des informations mémorisées

L'instruction désigne la(s) **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	\leftarrow	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).

Exemples

- $\text{reg12} \leftarrow \text{reg14} + \text{reg1}$
- $\text{registre4} \leftarrow \text{le mot mémoire d'adresse } 36000 + \text{le registre A}$
- $\text{reg5} \leftarrow \text{reg5} - 1$
- $\text{le mot mémoire d'adresse } 564 \leftarrow \text{registre7}$

Convention de noms

mov, ldr, str, add, sub, and, orr

Exemples

- Branch #125 : l'instruction suivante est désignée par une **adresse fixe**.
- Branch -40 : l'instruction suivante est une **adresse calculée**.
- Branch S1Zero +10 : si le résultat du calcul précédent est ZERO, alors la prochaine instruction à exécuter est celle d'adresse « **adresse courante+10** », sinon la prochaine instruction à exécuter est la suivante dans l'ordre d'écriture, c'est-à-dire à l'adresse « **adresse courante** » +t.

Exemples

En ARM :

- **add r4, r5, r6** signifie $r4 \leftarrow r5 + r6$.

$r5$ désigne le contenu du registre, on parle bien sûr du **contenu** des registres, on n'ajoute pas des ressources physiques !

En X86 (Intel) :

- **add eax, 10** signifie $eax \leftarrow eax + 10$.

En 6800 ou 68000 (Motorola) :

- **addA 5000** signifie $regA \leftarrow regA + Mem[5000]$
- **MOVE.W #500,D0** signifie $regD0 \leftarrow 500$

Remarque : pas de règle générale, interprétations différentes selon les fabricants, quelques habitudes cependant concernant les mnémoniques (add, sub, load, store, jump, branch, clear, inc, dec) ou la notation des opérandes (#, [xxx])

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

(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, ...

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 : 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 : directe ou absolue

(4/7)

Désignations registre/directe ou absolue.

On donne dans l'instruction l'adresse de l'objet désigné. L'objet désigné peut être une instruction ou une donnée.

- En **68000 (Motorola)** : `move.l D3, $FF9002` signifie

`Mem[FF9002] ← D3.`

la deuxième opérande (ici une donnée) est désigné par son adresse en mémoire.

- En **SPARC** : `jump 0x2000` signifie l'instruction suivante (qui est l'instruction que l'on veut désigner) est celle d'adresse 0x2000.

Désignation des objets : indirect par registre & déplacement
(6/7)

Désignation registre/indirect par registre et déplacement

L'adresse de l'objet désigné est obtenue en ajoutant le contenu d'un registre précisé dans l'instruction et d'une valeur (ou d'un autre registre) précisé aussi dans l'instruction.

- En **ARM** : `ldr r3, [r5, #4]` signifie $r3 \leftarrow \text{mem}[r5 + 4]$. La notation `[r5, #4]` désigne le mot mémoire (une donnée ici) d'adresse $r5 + 4$.
- En **6800** : `jump [PC - 12]` = le registre est PC, le déplacement -12.

L'instruction suivante (qui est l'instruction que l'on veut désigner) est celle à l'adresse obtenue en calculant, au moment de l'exécution, $PC - 12$.

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 $r3 \leftarrow (\text{le mot mémoire dont l'adresse est contenue dans le registre } 5)$
On note $r3 \leftarrow \text{mem}[r5]$.

Désignation des objets : relative au compteur programme
(7/7)

Désignation relative au compteur programme

L'adresse de l'objet désigné (en général une instruction) est obtenue en ajoutant le contenu du compteur de programme et une valeur précisée aussi dans l'instruction.

En **ARM** : `b + 20` signifie $pc \leftarrow pc + 20$

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 !).

Etiquettes : exemple

(2/4)

```
zone TEXT
DD: move r4, #42
    load r5, [YY]
    jump DD
```

```
zone DATA
XX: entier sur 4 octets : 0x56F3A5E2
YY: entier sur 4 octets : 0xAAFF43210
```

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

Etiquettes : correspondance étiquette/adresse

(4/4)

Supposons les adresses de début des zones TEXT et DATA respectivement 2000 et 5000

Il faut remplacer DD par 2000 et YY par 5004.

zone TEXT	contenu de Mem[2000], ...
DD: move r4, #42	move r4, #42
load r5, [YY]	load r5, [5004]
jump DD	jump 2000

zone DATA	
XX: entier sur 4 octets : 0x56F3A5E2	
YY: entier sur 4 octets : 0xAAFF43210	

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 \leftarrow 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;

Une première solution

```
I1
si a=b alors {I2; I3; I4}
I5
```

```
I1
calcul de a-b + positionnement de ZNCV
branch si (egal a 0) a etiq_alors
branch a etiq_suite
etiq_alors: I2
I3
I4
etiq_suite: I5
```

Instruction *Si < simple >*

```
I1
si a=b alors {I2; I3; I4}
I5
```

a et b deux entiers dont les valeurs sont rangées respectivement dans les registres r1 et r2.

Codage en ARM

```
x $\leftarrow$ 0;a $\leftarrow$ 5;b $\leftarrow$ 6
si a=b alors {x $\leftarrow$ 1;}
x $\leftarrow$ x+10;
a et b dans r0, r2, x dans r1
    mov r1, #0
    mov r0, #5
    mov r2, #6
    cmp r0,r2          @ ou subs r3, r0, r2
    beq alors          @ égal à 0
                      b finsi          @ always
alors:   mov r1, #1
finsi:   add r1, r1, #10
```

Remarque : égal à 0 équivalent à Z

Une autre solution

```
I1
si a=b alors {I2; I3; I4}
```

```
I5
I1
calcul de a-b + positionnement de ZNCV
branch si (non égal à 0) à etiq_suite
I2
I3
I4
etiq_suite: I5
```

Question

Combien d'instructions nécessaires pour la traduction de
 $a \leftarrow 5; b \leftarrow 6;$ si $a=b$ alors $\{x \leftarrow 1;\}$ sinon $\{x \leftarrow 0;\}$
 a et b dans r0, r2, x dans r1
 traduction avec respect de l'ordre alors/sinon (condition opposée).

- A) 5 instructions
- B) 6 instructions
- C) 7 instructions
- D) 10 instructions

Instruction Si alors sinon : Une solution

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

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

Codage en ARM

Traduction :

```
a←5;b←6; si a=b alors {x←1;} sinon {x←0;}
a et b dans r0, r2, x dans r1
```

```
mov r0, #5
mov r2, #6
cmp r0,r2
bne sinon
mov r1, #1      @ alors
b finsi
sinon: mov r1,#0
finsi:
```

Question

Combien d'instructions nécessaires pour la traduction de
 $a \leftarrow 5; b \leftarrow 6; \text{ si } a=b \text{ alors } \{x \leftarrow 1;\} \text{ sinon } \{x \leftarrow 0;\}$
 a et b dans r0, r2, x dans r1
 traduction avec respect de l'ordre alors/sinon (condition opposée).

- A) 5 instructions
- B) 6 instructions
- C) 7 instructions
- D) 10 instructions

Réponse attendue : C) 7 instructions (et une étiquette)

Exécution

		Ligne	r0	r2	?=?	r1	proch	Ligne
I.0	mov r0 , #5	-1	?	?	?	?	0	
I.1	mov r2, #6	0	5	?	?	?	1	
I.2	cmp r0,r2	1		6	?	?	2	
I.3	bne sinon	2			faux	?	3	
I.4 alors:	mov r1, #1	3				?	6	
I.5	b finsi	6				0	7	
I.6 sinon:	mov r1, #0	7						
I.7 finsi:	nop							

Question

Combien d'instructions **effectuées** lors de l'**exécution** de
 $a \leftarrow 5; b \leftarrow 6; \text{ si } a=b \text{ alors } \{x \leftarrow 1;\} \text{ sinon } \{x \leftarrow 0;\}$
 a et b dans r0, r2, x dans r1
 traduction avec respect de l'ordre alors/sinon (condition opposée).

- A) 5 instructions
- B) 6 instructions
- C) 7 instructions
- D) 10 instructions

Question

Combien d'instructions **effectuées** lors de l'**exécution** de
 $a \leftarrow 5; b \leftarrow 6; \text{ si } a=b \text{ alors } \{x \leftarrow 1;\} \text{ sinon } \{x \leftarrow 0;\}$
 a et b dans r0, r2, x dans r1
 traduction avec respect de l'ordre alors/sinon (condition opposée).

- A) 5 instructions
- B) 6 instructions
- C) 7 instructions
- D) 10 instructions

Réponse attendue : A) 5 instructions

Une autre solution

```
I1
si ExpCond alors {I2; I3;} sinon {I4; I5; I6;}
I7;
I1
evaluer ExpCond + ZNCV
branch si vrai a etiq_alors
I4
I5
I6
branch etiq_finsi
etiq_alors: I2
I3
etiq_finsi: I7
```

Codage en ARM

```
a←0; b←5; tant que a<b faire {x←a; a←a+1;} ; x←b;
```

a, b dans r0, r2, x dans r1

```
    mov r0, #0
    mov r2, #5
tq:   cmp r0,r2
      bge fintq @ ou bhs
      mov r1,r0 @ corps de boucle
      add r0,r0,#1
      b tq
fintq: mov r1,r2
```

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
```

Question (1/2)

Quelles seront les valeurs prises par r0 ?

- A) 0, 1, 2, 3, 4
- B) 0, 1, 2, 3, 4, 5
- C) 0, 1, 2, 3, 4, 5, 6

Exécution

(1/2)

```

I.0    mov r0, #0
I.1    mov r2, #5
I.2 tq: cmp r0,r2
I.3    bge fintq
I.4    mov r1,r0
I.5    add r0,r0,#1
I.6    b tq
I.7 fintq: mov r1,r2

```

	Ligne	r0	r2	?>=?	r1	proch Ligne
	-1	?	?	?	?	0
	0	0	?	?	?	1
	1		5	?	?	2
	2			faux	?	3
	3				?	4
	4				0	5
	5	1				6
	6				2	
	7					
	8					
	9					
	10					
	11					
	12					
	13					
	14					
	15					
	16					
	17					
	18					
	19					
	20					
	21					
	22					
	23					
	24					
	25					
	26					
	27					
	28					
	29					
	30					
	31					
	32					
	33					
	34					
	35					
	36					
	37					
	38					
	39					
	40					
	41					
	42					
	43					
	44					
	45					
	46					
	47					
	48					
	49					
	50					
	51					
	52					
	53					
	54					
	55					
	56					
	57					
	58					
	59					
	60					
	61					
	62					
	63					
	64					
	65					
	66					
	67					
	68					
	69					
	70					
	71					
	72					
	73					
	74					
	75					
	76					
	77					
	78					
	79					
	80					
	81					
	82					
	83					
	84					
	85					
	86					
	87					
	88					
	89					
	90					
	91					
	92					
	93					
	94					
	95					
	96					
	97					
	98					
	99					
	100					
	101					
	102					
	103					
	104					
	105					
	106					
	107					
	108					
	109					
	110					
	111					
	112					
	113					
	114					
	115					
	116					
	117					
	118					
	119					
	120					
	121					
	122					
	123					
	124					
	125					
	126					
	127					
	128					
	129					
	130					
	131					
	132					
	133					
	134					
	135					
	136					
	137					
	138					
	139					
	140					
	141					
	142					
	143					
	144					
	145					
	146					
	147					
	148					
	149					
	150					
	151					
	152					
	153					
	154					
	155					
	156					
	157					
	158					
	159					
	160					
	161					
	162					
	163					
	164					
	165					
	166					
	167					
	168					
	169					
	170					
	171					
	172					
	173					
	174					
	175					
	176					
	177					
	178					
	179					
	180					
	181					
	182					
	183					
	184					
	185					
	186					
	187					
	188					
	189					
	190					
	191					
	192					
	193					
	194					
	195					
	196					
	197					
	198					
	199					
	200					
	201					
	202					
	203					
	204					
	205					
	206					
	207					
	208					
	209					
	210					
	211					
	212					
	213					
	214					
	215					
	216					
	217					
	218					
	219					
	220					
	221					
	222					
	223					
	224					
	225					
	226					
	227					
	228					
	229					
	230					
	231					
	232					
	233					
	234					
	235					
	236					
	237					
	238					
	239					
	240					
	241					
	242					
	243					
	244					
	245					
	246					
	247					
	248					
	249					
	250					
	251					
	252					
	253					
	254					
	255					
	256					
	257					
	258					
	259					
	260					
	261					
	262					
	263					
	264					
	265					
	266					
	267					
	268					
	269					
	270					
	271					
	272					
	273					
	274					
	275					
	276					
	277					
	278					
	279					
	280					
	281					
	282					
	283					
	284					
	285					
	286					
	287					
	288					
	289					
	290					
	291					
	292					

Une autre solution

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

```
I1
branch etiqcond
debutbcle: I2
I3
etiqcond: evaluer ExpCond
branch si vrai debutbcle
fintq: I4
```

Solution

```
I1
répéter {I2; I3;} jusqu'à ExpCond;
I4;
```

```
I1
debutbcle: I2
I3
evaluer ExpCond
branch si faux debutbcle
I4
```

Observer les différences entre ce codage et la solution du tant que avec test à la fin.

Exercice

Deux boucles imbriquées
pour (i=0 a N)

pour (j=0 a K)

I2 ; I3

Expression conditionnelle complexe avec des et : solution 1

```
si C1 et C2 et C3 alors { I1; I2 } sinon I3
evaluer C1
branch si faux etiq_sinon
evaluer C2
branch si faux etiq_sinon
evaluer C3
branch si faux etiq_sinon
etiq_alors: I1
I2
branch etiq_fin
etiq_sinon: I3
etiq_fin:
```

Expression conditionnelle complexe avec des *et* : solution 2

```

si C1 et C2 et C3 alors { I1; I2 } sinon I3
    evaluer C1
    branch si faux etiq_sinon
    evaluer C2
    branch si faux etiq_sinon
    evaluer C3
    branch si vrai etiq_alors
etiq_sinon: I3
    branch etiq_fin
etiq_alors: I1
    I2
etiq_fin:

```

Expression conditionnelle complexe avec des *ou* : Solution 1

```

si C1 ou C2 ou C3 alors { I1;I2 } sinon I3
    evaluer C1
    branch si vrai etiq_alors
    evaluer C2
    branch si vrai etiq_alors
    evaluer C3
    branch si faux etiq_sinon
etiq_alors: I1
    I2
    branch etiq_fin
etiq_sinon: I3
etiq_fin:

```

Expression conditionnelle complexe avec des *et*

si C1 et C2 et C3 alors { I1; I2 } sinon I3

Solution avec évaluation **complète** des conditions

- Evaluer chaque **Ci** dans un registre
- Utiliser l'instruction **AND** du processeur.

Expression conditionnelle complexe avec des *ou* : Solution 2

```

si C1 ou C2 ou C3 alors { I1;I2 } sinon I3
    evaluer C1
    branch si vrai etiq_alors
    evaluer C2
    branch si vrai etiq_alors
    evaluer C3
    branch si vrai etiq_alors
etiq_sinon: I3
    branch etiq_fin
etiq_alors: I1
    I2
etiq_fin:

```

Expression conditionnelle complexe avec des *ou*

si C1 ou C2 ou C3 alors { I1; I2 } sinon I3

Solution avec évaluation **complète** des conditions

- Evaluer chaque **Ci** dans un registre
- Utiliser l'instruction **ORR** du processeur.

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

Solution

Instructions ARM conditionnelle.

Dans le codage d'une instruction, champ condition (bits 28 à 31).

Sémantique d'une instruction : si la condition est vraie, exécuter l'instruction sinon passer à l'instruction suivante.

selon a,b:	a dans r0, b dans r1, x dans r2
a<b : x<-x+5	cmp r0,r1
a=b : x<-x+1	addt r2, r2, #5
a>b : x<-x+9	addeq r2, r2, #1
	addgt r2, r2, #9

Que se passe-t-il si on remplace le **addeq** par un **addeqs** ?

Enoncé : le nombre de 1

Traduisez l'algorithme suivant en ARM :

```
x, nb : entiers >= 0
nb<-0
tant que x<>0 faire
    si x mod 2 <> 0 alors nb<-nb+1
    x<-x div 2
fin tant que
afficher nb
```

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

15 décembre 2025



Un exemple en langage de « haut niveau » (1/2)

```
int PP(int x) {
    int z, p;
    z = x + 1;
    p = z + 2;
    return p;
}
```

```
int main() {
    int i, j, k;
    i = 0;
    j = i + 3;
    j = PP(i + 1);
    k = PP(2 * (i + 5));
    return 0;
}
```

Analyse

- Le **main**, nommé **appelant** fait appel à la fonction **PP**, nommée **appelée**
- La fonction **PP** a un **paramètre** qui constitue une **donnée**, on parle de **paramètre formel**
- La fonction **PP** calcule une valeur de type entier, le **résultat de la fonction**
- Les variables **z** et **p** sont appelées **variables locales** à la fonction **PP**

Un exemple en langage de « haut niveau » (2/2)

```
int PP(int x) {
    int z, p;
    z = x + 1;
    p = z + 2;
    return p;
}
```

```
int main() {
    int i, j, k;
    i = 0;
    j = i + 3;
    j = PP(i + 1);
    k = PP(2 * (i + 5));
    return 0;
}
```

- Il y a deux **appels** à la fonction **PP**
- Lors de l'appel **PP(i + 1)**, la valeur de l'expression **i+1** est passée à la fonction, c'est le **paramètre effectif** que l'on appelle aussi **argument**
- Après l'appel le résultat de la fonction est rangé dans la variable **j**: **j = PP(i+1)**
- Le 1^{er} appel revient à exécuter le corps de la fonction en remplaçant **x** par **i+1**; le 2^{ème} appel consiste en l'exécution du corps de la fonction en remplaçant **x** par **2*(i+5)**

Quelle convention d'appel choisir ?

Objectif du module

Prise en main de la convention utilisée par gcc :

- passage des arguments : ↗ les 4 premiers dans r0 à r3
↗ 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 **conceptis complexes**...

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

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.

Tentative de traduction en ARM

Ce qui précède fonctionne mais ce n'est pas la solution préconisé par ARM.

Nouvelle tentative de traduction en ARM : convention \mathcal{CT}_0

Code en langage d'assemblage

PP :	add r3, r5, #1	@ z ← x + 1
	add r4, r3, #2	@ p ← z + 2
	mov r6, r4	@ rendre p
	retourner	
main :	mov r0, #0	@ i ← 0
	add r1, r0, #3	@ j ← i + 3
@ —Début-1 ^{er} -appel—		
	add r5, r0, #1	@ x ← i + 1
	appeler PP	
	mov r1, r6	@ j ← ...
@ —Fin-1 ^{er} -appel—		
@ —Début-2 ^{ème} -appel—		
	add r7, r0, #5	@ r7 ← i + 5
	mov r5, r7, lsl #1	@ x ← 2 * r7
	appeler PP	
	mov r2, r6	@ k ← ...
@ —Fin-2 ^{ème} -appel—		

Problème :
appeler et retourner ?

Quel est le problème ?

 \mathcal{CT}_0

Appel = branchement
instruction de rupture de séquence inconditionnelle (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.

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.

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.

EcrNdecim32 dans es.s

Rappel procedures d'affichage (es.s) :

```
.global EcrNdecim32
@ EcrNdecim32 : écriture en decimal de l'entier dans r1
EcrNdecim32 : mov ip, sp
                stmdf sp!, {r0, r1, r2, r3, fp, ip, lr, pc}
                sub fp, ip, #4
                ldr r0, LD_fe_na32
                bl printf
                ldmea fp, {r0, r1, r2, r3, fp, sp, pc}

LD_fe_na32 : .word fe_na32

fe_na32 : .asciz "%u"

(extrait de es.s)
```

Codage complet de l'exemple

 \mathcal{CT}_0

```

PP : add r3, r5, #1      @ z ← x + 1
      add r4, r3, #2      @ p ← z + 2
      mov r6, r4          @ rendre p
      bx lr               retour

main : mov r0, #0          @ i ← 0
       add r1, r0, #3      @ j ← i + 3
@ —Début-1er-appel—
       add r5, r0, #1      @ x ← i + 1
       bl PP               appel
       mov r1, r6          @ j ← PP(x)
@ —Fin-1er-appel—
@ —Début-2ème-appel—
       add r7, r0, #5      @ r7 ← i + 5
       mov r5, r7, lsl #1   @ x ← 2 * r7
       bl PP               appel
       mov r2, r6          @ k ← PP(x)
@ —Fin-2ème-appel—
  
```

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, ...)

Exécution

 \mathcal{CT}_0

I.	r0	r1	r3	r4	r5	r6	lr	> I.
I.0 PP :								
I.1	4	0	?	?	?	?	?	5
I.2	5	3	?	?	?	?	?	6
I.3	6		?	?	1	?	?	7
I.4 main :	7		?	?		?	8	0
I.5	0	2	?	?		?	1	
I.6	1		4			?	2	
I.7	2			4			3	
I.8	3				8		8	
I.9	8	4					9	
I.10	9						10	
I.11	10						10	11
I.12	11						12	0
	...							

Un deuxième problème : fonctions récursives (1/2)

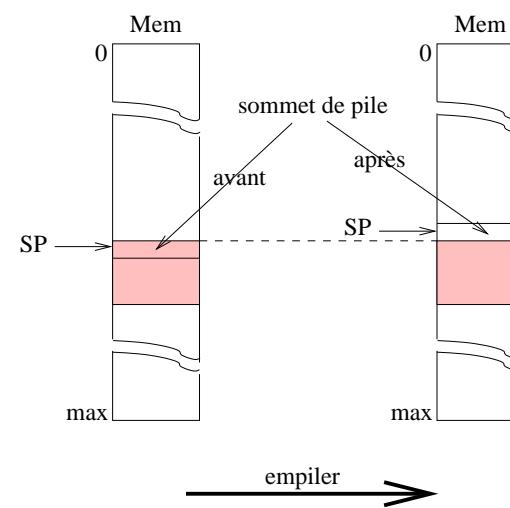
```

int fact (int x)
  if (x==0) then return 1
  else return x * fact(x-1);

// appel principal
int n, y;
  .... lecture d'un entier dans n
  y = fact(n);
  .... utilisation de la valeur de y
  
```


Comment réaliser une pile ?

(3/4)



Comment réaliser une pile ?

(4/4)

En Arm, empiler R3 (convention *full descending*) :

- push {R3}
- stmdfd SP!, {R3}
- str R3, [SP, #-4]!
- add SP, SP, #-4
- str R3, [SP]

En Arm, dépiler R3 (convention *full descending*) :

- pop {R3}
- ldmfd SP!, {R3}
- ldr R3, [SP], #4
- ldr R3, [SP]
- add SP, SP, #4

Appel/retour : utilisation d'une pile

 \mathcal{CT}_1

Appel de procédure, deux actions exécutées par le processeur :

- sauvegarde de l'adresse de retour dans une pile
c'est-à-dire **empiler(PC + taille)**
- modification du compteur programme (rupture de séquence)
c'est-à-dire **$PC \leftarrow \text{adresse de la procédure}$**

Au retour, PC prend pour valeur l'adresse en sommet de pile puis le sommet est déplié : **$PC \leftarrow \text{dépiler}()$** .

Remarque

Ce n'est pas la solution utilisée par le processeur Arm.
→ mais c'est presque le cas.

Application sur l'exemple

 \mathcal{CT}_1

La taille de codage d'une instruction est supposée être égale à 1

10	A1		20	B1
11	A2		21	B2
12	empiler 13; sauter à 20 (B)		22	B3
13	A3		23	retour: dépiler PC
14	empiler 15; sauter à 30 (C)			
15	A4			

30	C1
31	empiler 32; sauter à 20 (B)
32	C2
33	si X alors empiler 34; sauter à 30
34	C3
35	C4
36	retour: dépiler PC

Trace d'exécution (\mathcal{CT}_1)

(1/2)

PC	instructions	état de la pile
10	A1	{}
11	A2	{}
12	saut 20 (B)	empile 13
20	B1	{13}
21	B2	{13}
22	B3	{13}
23	retour	sommet = 13
13	A3	{}
14	saut 30 (C)	empile 15
30	C1	{15}
31	saut 20 (B)	empile 32
20	B1	{32; 15}
21	B2	{32; 15}
22	B3	{32; 15}
23	retour	sommet = 32
32	C2	{15}

Trace d'exécution (\mathcal{CT}_1)

(2/2)

33	cond :saut 30 (C)	empile 34 {34; 15}
30		empile 32 {32; 34; 15}
31		B1 {32; 34; 15}
20		B2 {32; 34; 15}
21		B3 {32; 34; 15}
22		retour sommet = 32
23		{34; 15}
32		(pas d'appel à C) {34; 15}
33	cond :saut 30	{34; 15}
34		C3 {34; 15}
35		C4 {34; 15}
36		retour sommet = 34
34	C3	{15}
35	C4	{15}
36	retour	sommet = 15
15	A4	{ }

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 ...

Application sur l'exemple \mathcal{CT}_2

La taille de codage d'une instruction est supposée être égale à 1

10	A1	20	empiler LR
11	A2	21	B1
12	sauver 13 dans LR; sauter à 20 (B)	22	B2
13	A3	23	B3
14	sauver 15 dans LR; sauter à 30 (C)	24	retour: dépiler dans PC
15	A4		

30	empiler LR
31	C1
32	sauver 33 dans LR ; sauter à 20 (B)
33	C2
34	si X alors sauver 35 dans LR; sauter à 30 (C)
35	C3
36	C4
37	retour: dépiler dans PC

Application sur l'exemple (version avec le BI d'Arm)

 $C\mathcal{T}_2$

En utilisant l'instruction BL (Branch and Link) d'Arm :

10	A1		20	empiler LR
11	A2		21	B1
12	BL B (appel)		22	B2
13	A3		23	B3
14	BL C (appel)		24	retour: dépiler dans PC
15	A4			
			30	empiler LR
			31	C1
			32	BL B
			33	C2
			34	si X alors BL C
			35	C3
			36	C4
			37	retour: dépiler dans PC

Gestion des variables, des paramètres : généralisation

La gestion des appels en cascade nous a montré que les adresses de retour nécessitent une gestion « en pile »

En fait, c'est le fonctionnement général des appels de procédure qui a cette structure : **chaque variable locale et/ou paramètre est rangé dans la pile** et la case mémoire associée est repérée par son adresse.

Remarque

Lorsqu'une procédure n'en appelle pas d'autres, on parle de procédure **feuille**

→ la sauvegarde dans la pile n'est pas nécessaire.

C'est le cas de la procédure B dans l'exemple.

10	A1 (idem prec.)		20	B1
11	A2		21	B2
12	BL B		22	B3
13	A3		23	BX LR
14	BL C			
15	A4			
	30 empiler LR (idem prec.)			
	31 C1			
	32 BL B			
	33 C2			
	34 si X alors BL C			
	35 C3			
	36 C4			
	37 retour: dépiler dans PC			

Exemple

```
//procedure principale, sans parametre
procedure A
var u : entier
    u=2; B(u+3); u=5+u; B(u)

procedure B (donnee x : entier)
var s, v : entier
    s=x+4 ; C(s+1); v=2; C(s+v)

procedure C (donnee y : entier)
var t : entier
    t=5; ecrire(t*4); t=t+1
```

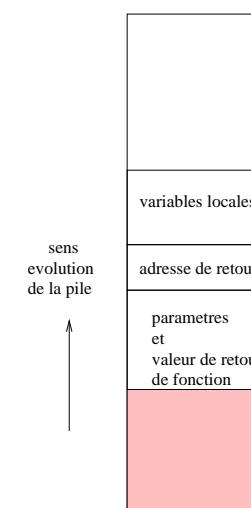
Flot d'exécution en partant de A

 \mathcal{CT}_3

Remarque : On supposera que **écrire** est une procédure qui demande son paramètre dans le registre **r1** (comme en TP)



Organisation de la pile lors de l'exécution d'une procédure

 \mathcal{CT}_3 

Remarques

 \mathcal{CT}_3

Dans l'exemple précédent, nous observons une gestion des zones de mémoire nécessaires pour les paramètres et les variables en pile !

L'approche est identique pour tout : résultats de fonction, paramètres, etc.

Et il faut, dans la même pile, sauvegarder les adresses de retour (cf. problème des appels en cascade)

Organisation du code

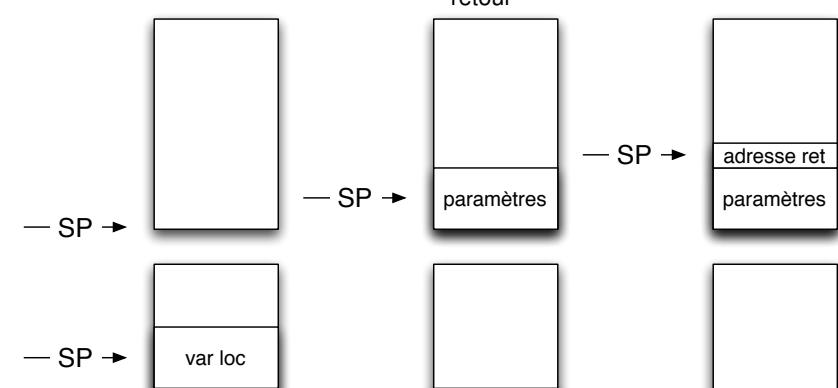
 \mathcal{CT}_3

appelant P :
préparer les paramètres
BL Q

libérer la place allouée aux paramètres

appelé Q :
sauver l'adresse de retour
allouer la place pour les variables locales

corps de la fonction
libérer la place réservée pour les var. locales
récupérer adresse de retour
retour



Comment accéder aux variables locales et aux paramètres ?

On pourrait utiliser le pointeur de pile *SP* :

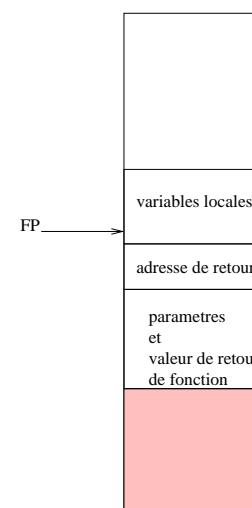
accès indirect avec déplacement : $[SP, \#dpl]$
 $dpl >= 0$

Mais si on utilise la pile, par exemple pour sauvegarder la valeur d'un registre que l'on souhaite utiliser, il faut re-calculer les déplacements.

Pas pratique !

Pose des problèmes de généralisation

Accès aux variables et paramètres : *frame pointer* (2/2)



Accès à un paramètre :
 $[fp, \#dpl_param]$
 $dpl_param > 0$

Accès à une variable locale :
 $[fp, \#dpl_varloc]$
 $dpl_varloc < 0$

Accès aux variables et paramètres : *frame pointer* (1/2)

Utiliser un repère sur l'environnement courant (paramètres et variables locales) qui reste **fixe** pendant toute la durée d'exécution de la procédure.

Ce repère est traditionnellement appelé ***frame pointer*** en compilation

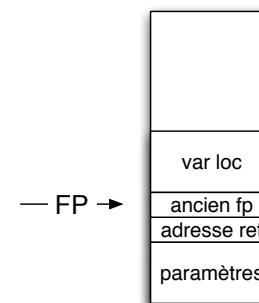
Un registre ***frame pointer*** existe dans la plupart des architectures de processeur : il est noté **fp** dans le processeur **Arm**.

Organisation du code en utilisant le registre *frame pointer* \mathcal{CT}_4

Comme pour le registre mémorisant l'adresse de retour, le registre **fp** doit être sauvegardé avant d'être utilisé.

appelant P :
 préparer les paramètres
 $BL\ Q$
 libérer la place allouée aux paramètres

appelé Q :
 sauver l'adresse de retour
 sauver l'ancienne valeur de **fp**
 placer **fp** pour repérer les nouvelles variables
 allouer la place pour les variables locales
corps de la fonction
 libérer la place réservée pour les var. locales
 restaurer **fp**
 récupérer adresse de retour
 retour

Organisation de la pile lors de l'exécution avec *frame pointer**CT4*Si les adresses sont sur **4 octets** :

- Accès aux variables locales :
adresse de la forme *fp - 4 - déplacement*
- Accès aux paramètres :
adresse de la forme *fp + 8 + déplacement*

Question

CT4

Pour le code suivant :

```
procedure B(donnee x : entier)
var s, v : entier
    s=x+4 ; C(s+1); v=2; C(s+v)
```

- A.) x se trouve en fp-8 et s et v se trouvent en fp-4 et fp-8
- B.) x se trouve en fp-8 et s et v se trouvent en fp+4 et fp+8
- C.) x se trouve en fp+8 et s et v se trouvent en fp-4 et fp-8
- D.) x se trouve en fp+8 et s et v se trouvent en fp+4 et fp+8
- E.) Je ne sais pas

Réponse :

C.) x se trouve en fp+8 et s et v se trouvent en fp-4 et fp-8

En Arm : code de B

CT4

```
B:
@ sauvegarde adresse retour
push {lr}   @ sub sp,sp,#4
            @ str lr,[sp]

push {fp}   @ sauvegarde ancien fp

mov fp,sp  @ mise en place nouveau fp

sub sp,sp,#8 @ reservation variables locales s,v

@@@ debut du corps de B @@
ldr r1, [fp,#+8] @ s <- x+4
add r1,r1,#4
str r1,[fp,#-4]

@@@ debut de l'appel a C @@
ldr r1, [fp,#-4] @ passage de s+1 en parametre de C
add r1,r1,#1
push {r1}

bl C @ appel C

add sp,sp,#4 @ depile le parametre
@@@ fin de l'appel a C @@
bx lr @ retour
```

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

15 décembre 2025

Résultat d'une fonction (Qui ? Quand ? Où ?)

- ➊ Le résultat d'une fonction est calculé **par l'appelée**
- ➋ Le résultat doit être rangé à un emplacement **accessible par l'appelante** de façon à ce que cette dernière puisse le récupérer.

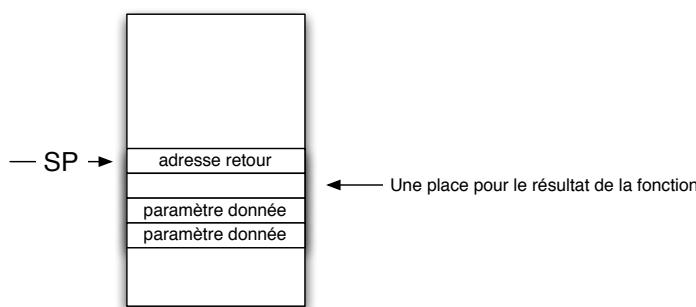
Il faut donc utiliser une zone mémoire **commune** à l'appelante et l'appelée.

Par exemple, **la pile**.

Résultat dans la pile (\mathcal{CT}_5) (2/3)

Avant l'appel d'une fonction qui a deux paramètres données

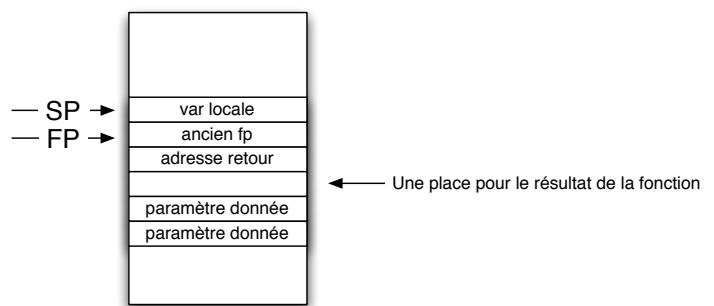
- Les valeurs des deux paramètres sont empilés
- Une case est réservée pour le résultat de la fonction



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

Lors de l'exécution du corps de la fonction.

- ➊ Les variables locales sont accessibles par une adresse de la forme : $fp - 4 - dep$ avec $dep \geq 0$,
- ➋ Les paramètres données par les adresses : $fp + 8 + 4$ et $fp + 8 + 8$ et
- ➌ La case résultat par l'adresse $fp + 8$.

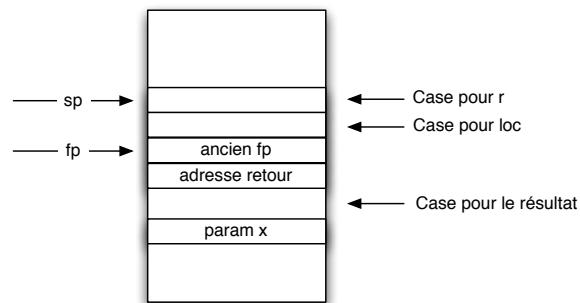


Structure du code de l'appel et du corps de la fonction CT₅**appelant P :**

préparer et empiler les paramètres
réserver la place du résultat dans la pile
appeler Q : BL Q
récupérer le résultat
libérer la place allouée aux paramètres
libérer la place allouée au résultat

appelé Q :

empiler l'adresse de retour
empiler la valeur de fp
placer fp pour repérer les nouvelles variables
allouer la place pour les variables locales
corps de la fonction Q
le résultat est rangé en fp+8
libérer la place allouée aux variables locales
dépiler fp
dépiler l'adresse de retour
retour à l'appelant (P) : BX lr

Etat de la pile lors de l'exécution de fact juste après l'appel dans main CT₅Application : fonction fact avec des variables locales CT₅

```
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);
    ...
}
```

Etat de la pile lors de l'exécution de fact juste après l'appel dans main CT₅Nouvelle version de la fonction fact CT₅

<pre>fact: @ empiler adr retour push {lr} @ mise en place fp @ place pour loc et r push {fp} mov fp, sp sub sp, sp, #8 @ if x==0 ... ldr r0, [fp, #+12] @ r0=x cmp r0, #0 bne sinon alors: mov r2, #1 str r2, [fp, #-8] @ r = 1 b finsi sinon: @ appel fact(x-1) @ preparer param et resultat sub sp, sp #4 sub r1, r0, #1 str r1, [sp]</pre>	<pre>sub sp, sp, #4 bl fact ldr r1, [sp] add sp, sp, #8 @ apres l'appel str r1, [fp, #-4] ldr r0, [fp, #+12] ldr r1, [fp, #-4] mul r2, r0, r1 str r2, [fp, #-8] finsi: ldr r2, [fp, #-8] str r2, [fp, #+8] @ recuperer place var loc add sp, sp, #8 pop {fp} @ retour pop {lr} bx lr</pre>	<pre>@ case resultat @ appel @ recuperer resultat @ desallouer param et res @ loc=fact(x-1) @ r0=x @ r1=loc @ x*loc @ r=x*loc</pre>
		<pre>@ recuperer fp @ recuperer lr</pre>

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

C_G

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

Question

(2/2)

Quelle place sur la pile est nécessaire pour l'exécution d'une fonction ayant 2 paramètres, 3 variables locales, 4 registres temporaires ?

- A.) 40 octets
- B.) 44 octets
- C.) 48 octets
- D.) 52 octets
- E.) Je ne sais pas

Réponse : $(2+3+4+1 \text{ (LR)}+1 \text{ (FP)}+1 \text{ (résultat fonction)})*4 = 48 \text{ octets}$

Question

(1/2)

Quelle place sur la pile est nécessaire pour l'exécution d'une fonction ayant 2 paramètres, 3 variables locales, 4 registres temporaires ?

- A.) une vingtaine d'octets
- B.) entre 24 et 44 octets
- C.) au moins 48 octets
- D.) Je ne sais pas

pour être plus précis, continuer...

Précisions sur la convention C_{gcc}

Pour des raisons d'optimisation et de facilités de programmation, gcc utilise la convention suivante :

- les 4 premiers paramètres sont passés dans les registres R0 à R3
 → s'il y a plus de 4 paramètres, les suivants sont stockés dans la pile
- la valeur de retour de la fonction est stockée dans R0
 (et R1 si besoin)
- certains registres à sauvegarder par l'**appelante**, d'autres par l'**appelée**

Cette convention est assez similaire à la convention C_G, mais utilise moins la pile

→ elle est donc un peu moins "générique"
 (traitement différent des registres)

Question

Les codes suivants fonctionnent-ils ?

```
procedure incr(x:entier)      procedure minus(str:chaîne)
x = x+1;                      str[0] = str[0] ou 32
```

```
n : entier                  ch : "Bonjour"
incr(n);                   minus(ch)
...
...
```

- A.) oui (pour les deux)
- B.) oui pour incrémenté, non pour minuscule
- C.) non pour incrémenté, oui pour minuscule
- D.) non (pour les deux)
- E.) Je connais la réponse, mais je ne sais pas dire pourquoi
- F.) Je ne sais pas

Réponse : C.) non pour incrémenté, oui pour minuscule

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

Situation : comment faire +1 via une fonction ?

- Directe : n : entier
n = n+1;

```
procedure inc (x : entier)
x = x+1;
```

- Par procédure :

```
n : entier
inc(n);
```

- Catastrophe, cela ne marche pas
- Le +1 s'effectue pour l'élément situé sur la pile, pas sur l'original !
- C'est le drame du passage de paramètre par valeur
- Solution : passage de paramètre par référence, ou par adresse
(paramètre donnée vs paramètre résultat)

Autre solution

Si on ne peut pas accéder à une référence...

- Par fonction (et confier l'affectation à l'appelant) :

```
fonction inc (x : entier)
retourne x+1;
```

n : entier
n=inc(n);

- Par macro (si disponible)

Réalisation, vocabulaire

On se place maintenant dans le cas d'une procédure ayant **des paramètres de type donnée** et **des paramètres de type résultat**.

```
procedure XX (données x, y : entier; résultat z : entier)
u,v : entier
```

```
...
u=x;
v=y+2;
...
z=u+v;
...
```

- Les paramètres données **ne doivent pas être modifiés par l'exécution de la procédure** : les paramètres effectifs associés à x et y sont des expressions qui sont évaluées avant l'appel, leurs valeurs étant substituées aux paramètres formels avant de l'exécution du corps de la procédure.
- Le paramètre effectif associé au paramètre formel résultat z est une variable **dont la valeur n'est significative qu'après l'appel de la procédure** ; cette valeur est calculée dans le corps de la procédure et affectée à la variable z passée en argument.

L'exemple d'appel traité

```
a,b,c : entier
b=3;
...
XX (b, 7, adresse de c);
```

Notations

Il existe différentes façons de gérer le paramètre z.

Nous n'en étudions qu'une seule : la méthode dite du **passage par adresse**.

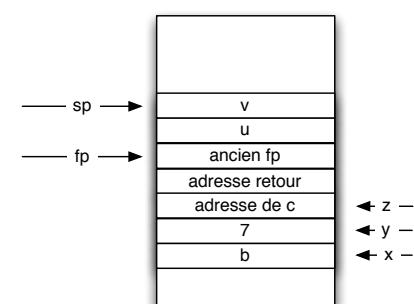
Nous utilisons la notation suivante :

```
procedure XX (données x, y : entier ; adresse z : entier)
u,v : entier
```

```
...
u=x;
v=y+2;
...
mem[z]=u+v; @ mem[z] désigne le contenu de la mémoire d'adresse z
...
```

Solution : état de la pile lors de l'exécution de la procédure XX

CG



Plan

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

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 préoccuperons surtout de la seconde phase.

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**.

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)

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); }

```

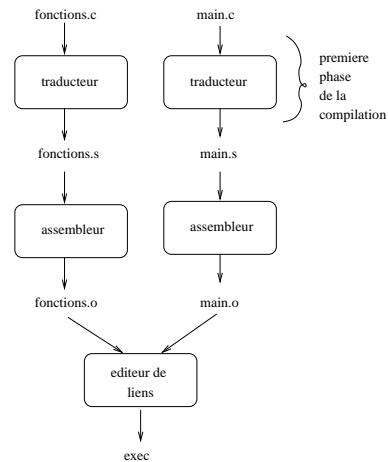
- Dans le fichier main.c les fonctions somme et max sont dites **importées** : elles sont définies dans un autre fichier.
 - Dans le fichier fonctions.c, somme et max sont dites **exportées** : elles sont utilisables dans un autre fichier.

```
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, #
    b bcle
fin: bx lr
LD_xx: .word xx
.data
.word 99
xx: .word 3
```

lib.

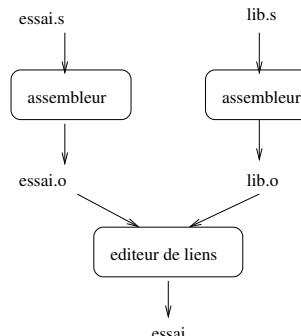
```
.text  
.global add1  
  
add1 : add r2, r2, #  
       bx lr
```



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).

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 appliquée 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 : 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
```

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 <bclie>:
 4: e350000a cmp r0, #10
 8: 0a000005 beg 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 <bclie>

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

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

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	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)

Exemple : essai.o, table des symboles

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

```
Symbol table '.symtab' contains 15 entries:
 Num: Value Size Type Bind Vis Ndx Name
 0: 00000000 0 NOTYPE LOCAL DEFAULT UND
 1: 00000000 0 SECTION LOCAL DEFAULT 1
 2: 00000000 0 SECTION LOCAL DEFAULT 3
 3: 00000000 0 SECTION LOCAL DEFAULT 4
 4: 00000000 0 NOTYPE LOCAL DEFAULT 1 $a
 5: 00000004 0 NOTYPE LOCAL DEFAULT 1 bclie
 6: 00000024 0 NOTYPE LOCAL DEFAULT 1 fin
 7: 00000028 0 NOTYPE LOCAL DEFAULT 1 LD_xx
 8: 00000028 0 NOTYPE LOCAL DEFAULT 1 $d
 9: 00000004 0 NOTYPE LOCAL DEFAULT 3 xx
10: 00000000 0 NOTYPE LOCAL DEFAULT 3 $d
11: 00000000 0 SECTION LOCAL DEFAULT 5
12: 00000000 0 NOTYPE GLOBAL DEFAULT 1 main
13: 00000000 0 NOTYPE GLOBAL DEFAULT UND add1
14: 00000000 0 NOTYPE GLOBAL DEFAULT UND exit
```

Exemple : essai.o, table de translation

On obtient la table de translation avec la commande `arm-eabi-readelf -a essai.o`.

Relocation section '.rel.text' at offset 0x33c contains 3 entries:

Offset	Info	Type	Sym.	Value	Sym.	Name
00000014	00000d01	R_ARM_PC24		00000000	add1	
00000024	00000e01	R_ARM_PC24		00000000	exit	
00000028	00000202	R_ARM_ABS32		00000000		.data

Suite exemple : lib.o, organisation des tables

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]	NULL	PROGBITS	00000000	000000	000000 00		0	0	0	0
[1]	.text	PROGBITS	00000000	000034	000008 00	AX	0	0	1	
[2]	.data	PROGBITS	00000000	00003c	000000 00	WA	0	0	1	
[3]	.bss	NOBITS	00000000	00003c	000000 00	WA	0	0	1	
[4]	.ARM.attributes	ARM_ATTRIBUTES	00000000	00003c	000010 00		0	0	1	
[5]	.shstrtab	STRTAB	00000000	00004c	00003c 00		0	0	1	
[6]	.symtab	SYMTAB	00000000	0001c8	000070 10		7	6	4	
[7]	.strtab	STRTAB	00000000	000238	000009 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)

Suite exemple : lib.o, tables des symboles

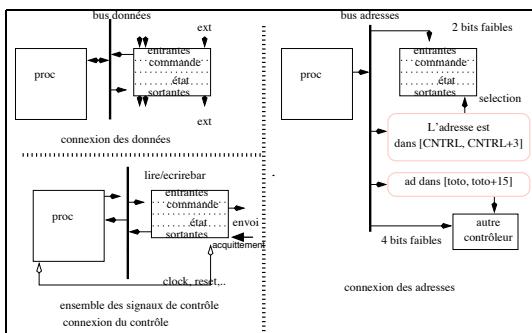
Symbol table '.symtab' contains 7 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	SECTION	LOCAL	DEFAULT	1	
2:	00000000	0	SECTION	LOCAL	DEFAULT	2	
3:	00000000	0	SECTION	LOCAL	DEFAULT	3	
4:	00000000	0	NOTYPE	LOCAL	DEFAULT	1	\$a
5:	00000000	0	SECTION	LOCAL	DEFAULT	4	
6:	00000000	0	NOTYPE	GLOBAL	DEFAULT	1	add1

Etapes d'un assebleur

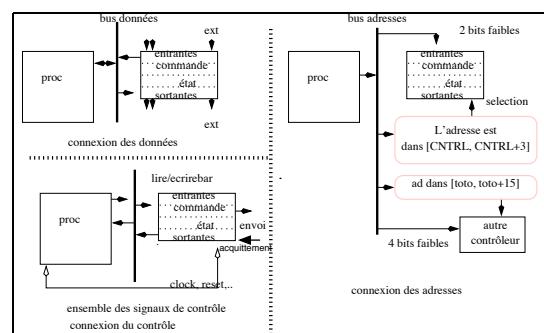
- ➊ Reconnaissance de la syntaxe (lexicographie et syntaxe)
- ➋ Repérage des symboles. Fabrication de la table des symboles utilisée par la suite dès qu'une référence à un symbole apparaît.
- ➌ Traduction = production du binaire.

Etude du matériel d'entrées-sorties : les entrées



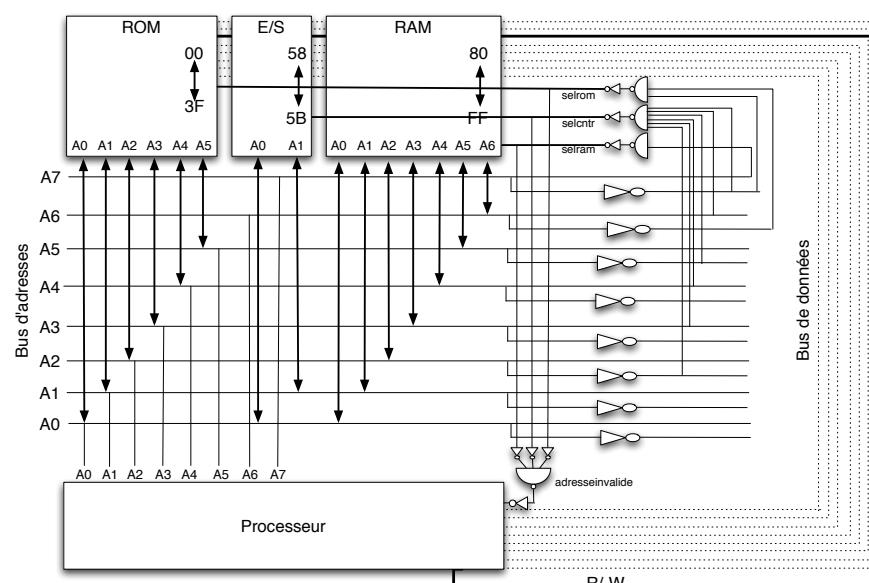
- bus données (lié au processeur)
- deux bits de bus adresses (pour sélectionner l'un des 4 mots CNTRL +0, +1, +2 ou +3)
- un signal de sélection provenant du **décodeur d'adresses**
- le signal *Read / Write* du processeur
- un paquet de données (8 fils) venant du monde extérieur. Disons pour simplifier 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 Mdonnéesentr.
- 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 Mdonnéesentr si il y a **sélection, lecture et adressage** de Mdonnéesentr, 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 Mdonnéeessort 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

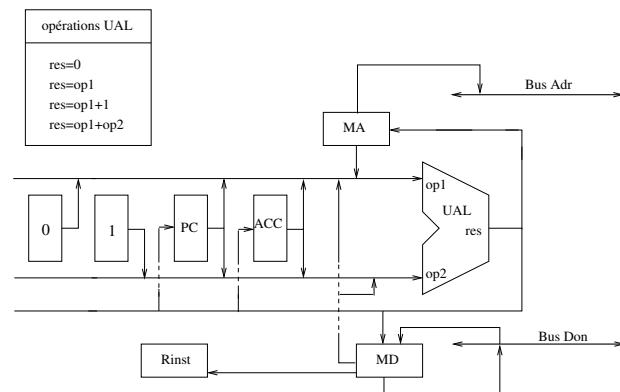
Université Grenoble Alpes

15 décembre 2025

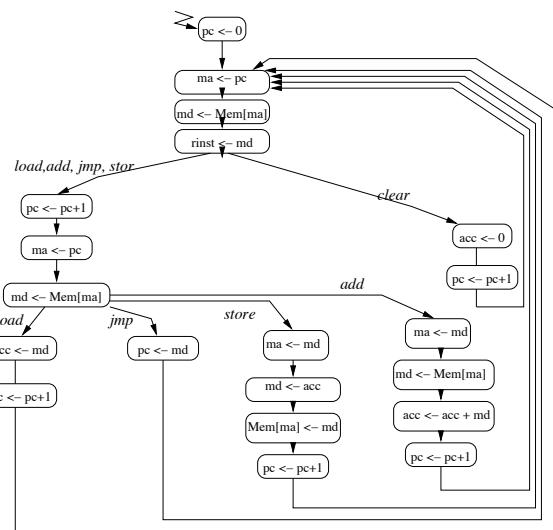
Partie opérative

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).



Une première version



Remarque : La notation de la condition clear doit être comprise comme le booléen rinst = 1.

Micro-actions et micro-conditions

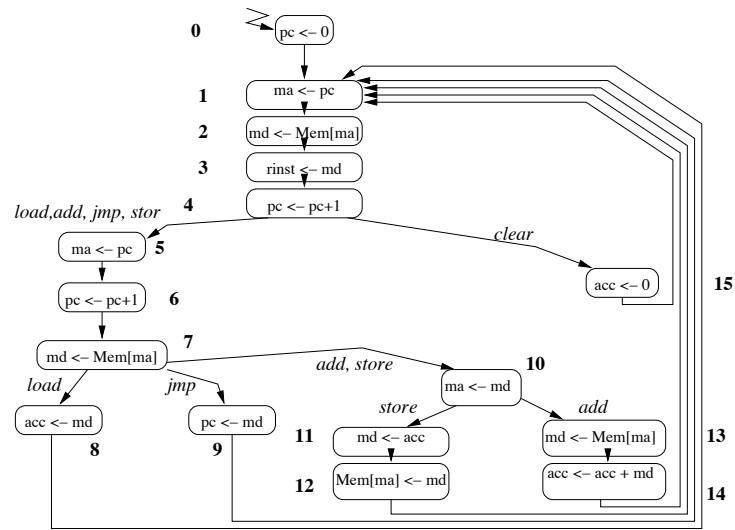
On fait des hypothèses FORTES sur les transferts possibles :

$md \leftarrow \text{mem}[ma]$	lecture d'un mot mémoire.	C'est la seule possibilité en lecture !
$\text{mem}[ma] \leftarrow md$	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
$reg_0 \leftarrow reg_1 + 1$	incrémentation	reg_0 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**.

Version améliorée de l'automate



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

Question

(1/2)

Combien de "temps" pour un `ld`? (en considérant que les lectures prennent 2 "temps" et les écritures 3 "temps")

- A.) 9 "temps" ou moins
- B.) 10-11 "temps"
- C.) 12-13 "temps"
- D.) 14 "temps" ou plus
- E.) Je ne sais pas

Réponse : B.) 10 "temps"

Question

(2/2)

Combien de "temps" pour un `st`? (en considérant que les lectures prennent 2 "temps" et les écritures 3 "temps")

- A.) 9 "temps" ou moins
- B.) 10-11 "temps"
- C.) 12-13 "temps"
- D.) 14 "temps" ou plus
- E.) Je ne sais pas

Réponse : D.) 14 "temps"

Exemple de code

	étiquette	mnémonique	référence	mode adressage
	ou directive			
	.text			
debut :	clr			
	ld	#8		immédiat
ici :	st	xx		absolu ou direct
	add	xx		absolu ou direct
	jmp	ici		absolu ou direct
	.data			
xx :				

Exercice : Que contient la mémoire après chargement en supposant que l'adresse de chargement est 0 et que xx est l'adresse 15.

Contenu en mémoire

adresse	valeur	origine
0	1	clr
1	2	load
2	8	val immédiate
3	3	store
4	15	adresse zone data
5	5	add
6	15	adresse zone data
7	4	jump
8	3	adresse de "ici"
...
15	variable	non initialisée

Exercice : Donnez le déroulement au cycle près du programme.

Déroulement

(2/2)

état	pc	ma	md	rinst	acc	mem[15]	état	pc	ma	md	rinst	acc	mem[15]
0	0						11		8				8
1		0					12						
2			1				1		5				
3				1			2			5			
4	1						3				5		
15					0		4	6					
1		1					5		6				
2			2				6	7					
3				2			7			15			
4	2						10		15				
5		2					13		8				
6	3						14			16			
7			8				1	7					
8				8			2		4				
1	3						3			4			
2		3					4	8					
3			3				5		8				
4	4						6	9					
5		4					7			3			
6	5						9	3					
7			15				1	etc.					
10		15											

Optimisations

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

Université Grenoble Alpes

15 décembre 2025



Plan

1 Arithmétique

2 Structures de contrôle

3 Fonctions

4 Pipeline

5 Mémoire cache

6 Conclusions

Résultats connus

• Calcul avec des constantes (version attendue)

```
.file "progConstantes.c"
.section .rodata
.align 2
.LC0:
.ascii "4+5=%d\012\000"
.text
.align 2
.global main
.type main, %function
main:
    stmf 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
```

Résultats connus

(2/2)

• Calcul avec des constantes (version optimisée)

```
.file "progConstantes.c"
.text
.align 2
.global main
.type main, %function
main:
    stmdfd 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\n",i+j
.ident "GCC: (GNU) 4.5.3"
```

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;}
```

Opérations équivalentes

• Divisions équivalentes

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;}
```

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

Conditionnelles (2/2)

- Conditionnelle (version optimisée)
- (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;
}
```

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 %rbp, -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
```

Boucles (1/2)

- Boucle (version attendue)
- (version clang)

```
#include <stdio.h>

int main () {
    int i, s=0, a;
    for(i=0;i<10;i++) {
        scanf("%d",&a);
        s=s+a;
    }
    printf("Somme : %d\n",s);
    return 0;
}
```

```
main: # @main
[...] subq $32, %rsp
movl $0, -4(%rbp)
movl $0, -12(%rbp)
movl $0, -8(%rbp)
.LBB0_1: # =>This Inner Loop
    cmpl $10, -8(%rbp)
    jge .LBB0_4
# BB#2: # in Loop: Header
    movabsq $.L.str, %rdi
    leaq -16(%rbp), %rsi
    movb $0, %al
    callq __isoc99_scanf
    movl -12(%rbp), %ecx
    addl -16(%rbp), %ecx
    movl %ecx, -12(%rbp)
    movl %eax, -20(%rbp) # 4-byte Spill
# BB#3: # in Loop: Header
    movl -8(%rbp), %eax
    addl $1, %eax
    movl %eax, -8(%rbp)
    jmp .LBB0_1
.LBB0_4:
    movabsq $.L.str1, %rdi
    movl -12(%rbp), %esi
    movb $0, %al
    callq printf
    movl $0, %esi
    movl %eax, -24(%rbp) # 4-byte Spill
    movl %esi, %eax
```

Boucles

(2/2)

- Boucle (version optimisée)
- (version clang)

```
#include <stdio.h>

int main () {
    int i, s=0, a;
    for(i=0;i<10;i++) {
        scanf("%d",&a);
        s+=a;
    }
    printf("Somme : %d\n",s);
    return 0;
}
```

```
main:
[...]
leaq 4(%rsp), %r14
movl $.L.str, %edi
xorl %eax, %eax
movq %r14, %rsi
callq __isoc99_scanf
movl 4(%rsp), %ebx
movl $.L.str, %edi
xorl %eax, %eax
movq %r14, %rsi
callq __isoc99_scanf
addl 4(%rsp), %ebx
movl $.L.str, %edi
xorl %eax, %eax
movq %r14, %rsi
callq __isoc99_scanf
addl 4(%rsp), %ebx
movl $.L.str, %edi
xorl %eax, %eax
movq %r14, %rsi
callq __isoc99_scanf
addl 4(%rsp), %ebx
movl $.L.str, %edi
xorl %eax, %eax
movq %r14, %rsi
callq __isoc99_scanf
addl 4(%rsp), %ebx
movl $.L.str, %edi
xorl %eax, %eax
movq %r14, %rsi
callq __isoc99_scanf
addl 4(%rsp), %ebx
movl $.L.str, %edi
xorl %eax, %eax
movq %r14, %rsi
callq __isoc99_scanf
addl 4(%rsp), %ebx
movl $.L.str, %edi
xorl %eax, %eax
```

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
```

Structures de contrôle (conclusions)

- Les structures de contrôle utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les structures de contrôle les plus claires
- Le compilateur optimise ! (tous les compilateurs, pour tous les asm)
- Mais attention : dérouler une boucle peut prendre de la place dans le fichier exécutable et du temps à la compilation ...
- Gain espéré : temps d'exécution des branchements et de calcul des compteurs d'itération
- Gain espéré : et plus ? [branchement ... voir la suite]

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)

```
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:
    stmf 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
    ldmd sp!, {fp, lr}
    bx lr
```

Fonctions récursives

(3/6)

- Fonction récursive (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:
    stmf sp!, {r3, lr}
    subs r3, r0, #0
    beg .L2
    sub r0, r3, #1
    add r1, r1, r3
    bl somme
    mov r1, r0
.L2:
    mov r0, r1
    ldmd sp!, {r3, lr}
    bx lr
```

Fonctions récursives

(2/6)

- Fonction récursive (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:
    stmf 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
    ldmd sp!, {r4, fp, lr}
    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:
    stmf 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
    ldmd sp!, {r4, 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

(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));
```

- 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

Rupture de pipeline

Mais !

les pipelines peuvent être rompus :

- Branchement
- Dépendance entre données
- ...

Pipeline (Conclusions)

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

Ce fonctionnement en pipeline peut être alteré :

- 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.

Pipeline et branchemet

Ruptures de pipeline et branchemet, cela concerne :

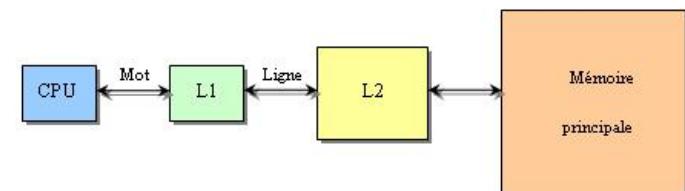
- Conditionnelles
- Boucles
- Appels de fonction

Le compilateur peut s'en occuper

→ Le programmeur peut aider

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 **parallelisme**
- Comprendre l'exécution en regardant le code machine
 - voire, contrôler le code machine ...
- Utiliser des *profiler* pour déterminer où se trouvent les coûts
 - n'optimiser que ce qui est bloquant ...
- **Mais surtout, avant d'optimiser : programmer juste !**