Architectures des ordinateurs (une introduction)

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

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`

Modèle de Von Neumann : qu'est ce qu'un ordinateur ?

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Description du modèle de Von Neumann (2/3)

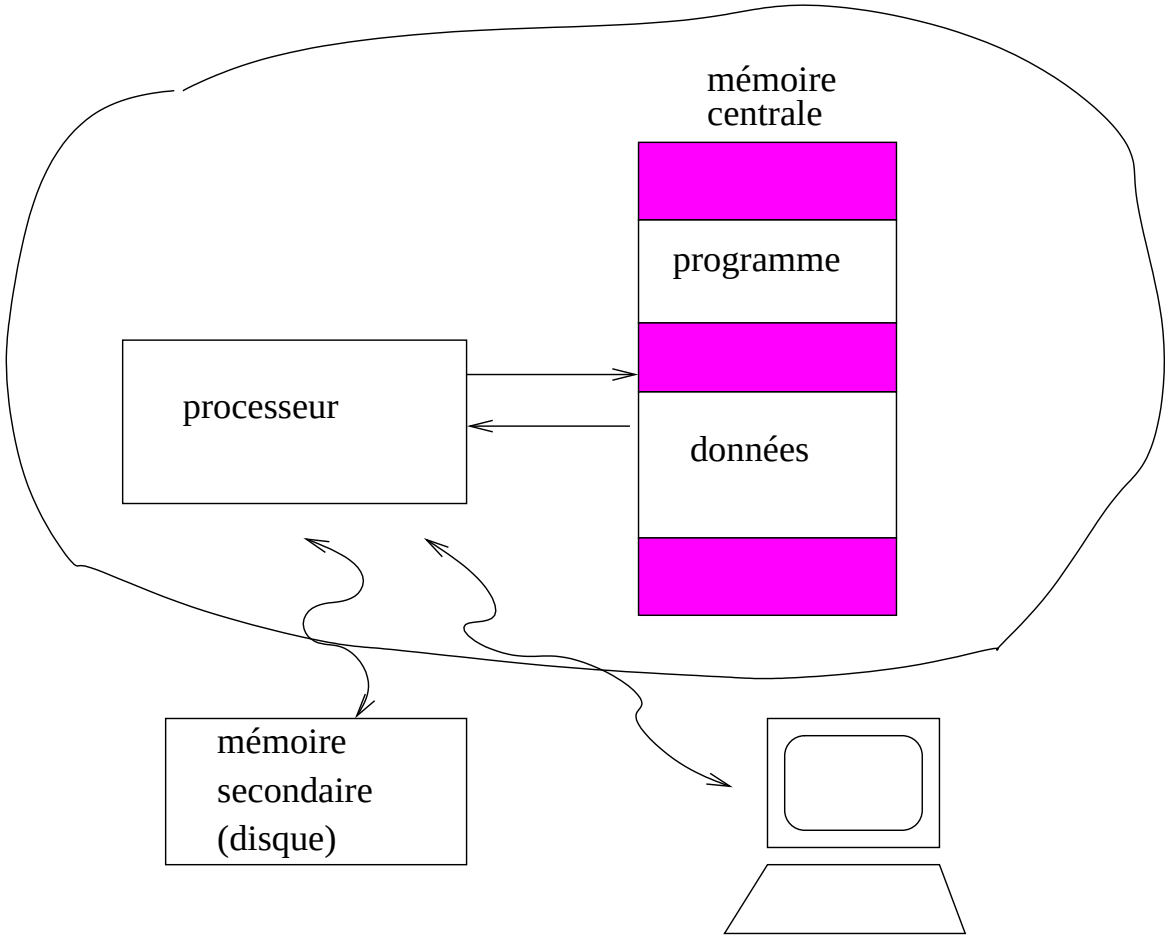


FIGURE – Processeur, mémoire et périphériques

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

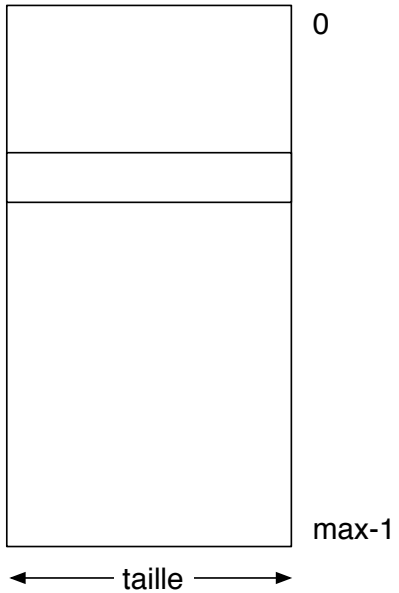


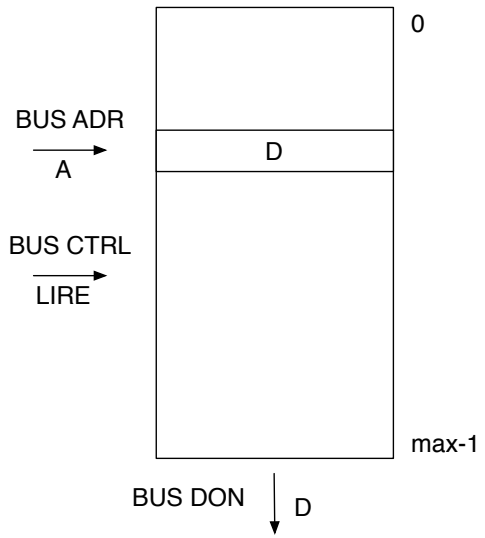
FIGURE – Mémoire abstraite

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 <-- mem[A]`

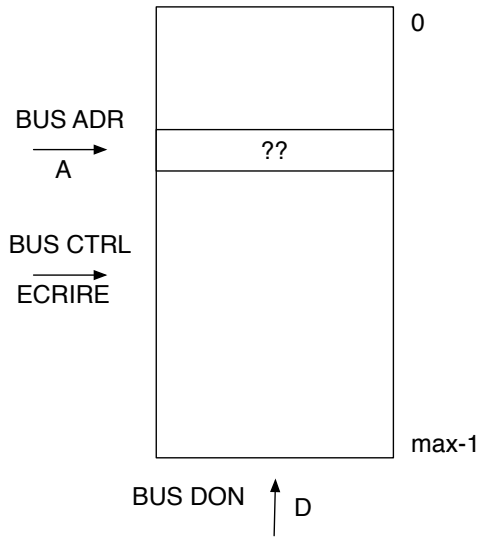
`mem[A]` : emplacement mémoire dont l'adresse est A

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 A



On écrit : `mem[A] <-- 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.

Entrées/Sorties : définitions

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

- L'intéraction de l'ordinateur (mémoire et processeur) avec l'**utilisateur** (clavier, écran, ...)
- L'intéraction 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.

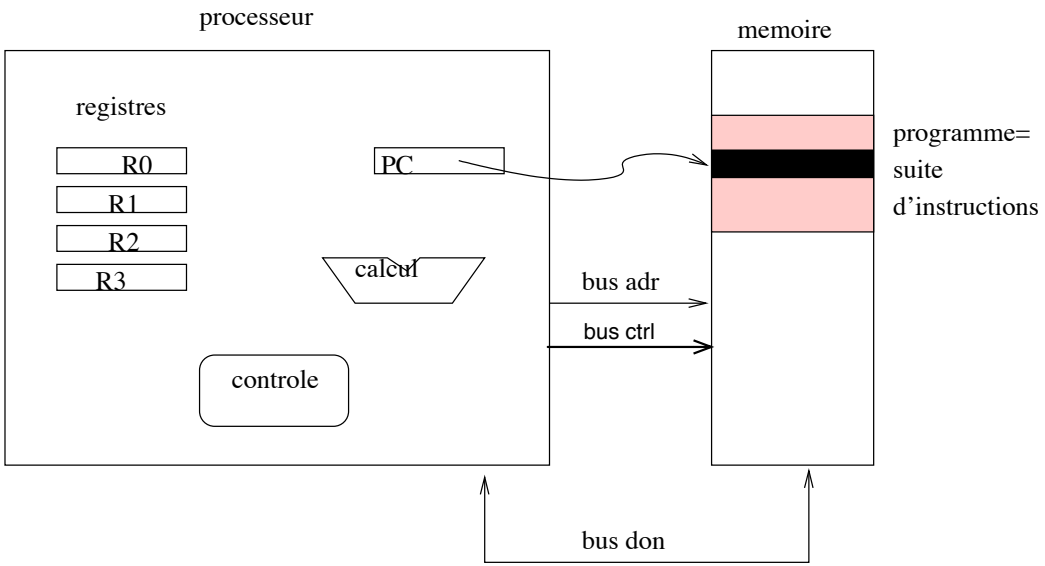
- Le bus de données permet la circulation des données.
- Le bus d'adresse permet au processeur de **désigner à chaque instant la case mémoire ou le périphérique** auquel il veut faire appel.
- Le bus de contrôle indique quelle est l'**opération que le processeur veut exécuter**, par exemple, s'il veut 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).

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 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)

Codage des informations et représentation des nombres par des vecteurs binaires

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

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.

UTF-8

- Codage extensible, compatible avec ASCII
- Permet de représenter plus d'un million de caractères

Caractères codés	Représentation binaire UTF-8	Signification
U+0000 à U+007F	0xxxxxxx	1 octet codant 1 à 7 bits
U+0080 à U+07FF	110xxxxx 10xxxxxx	2 octets codant 8 à 11 bits
U+0800 à U+0FFF	1110 0000 10 1xxxxx 10xxxxxx	3 octets codant 12 à 16 bits

Source wikipédia.

Correspondance entre n_uplet et naturel (2/2)

	0	1	2	3	4
0	(0,0) 0	(0,1) 1	(0,2) 2	(0,3) 3	(0,4) 4
1	(1,0) 5	(1,1) 6	(1,2) 7	(1,3) 8	(1,4) 9
...	...	...	...	...	
3	(3,0) 15	(3,1) 16	(3,2) 17	(3,3) 18	(3,4) 19

2 formules à savoir :

COD_COUPLE4_5 ((a, b)) = a x 5 + b

DECOD_COUPLE4_5 (n) = (n div 5, n reste 5)

Remarque : Quelles seraient ces formules si nous avions numéroté à partir de 1 au lieu de 0 ?

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 ≪ connait ≫ la règle de codage, sans elle il ne peut pas traiter l'information.

Numération de position

En numération de position, avec N chiffres en base b on peut représenter les b^N naturels de l'intervalle $[0, b^N\text{-}1]$

Exemple : en base 10 avec 3 chiffres on peut représenter les 10^3 naturels de l'intervalle $[0, 999]$.

Avec N chiffres binaires (base 2) on peut écrire les 2^N naturels de l'intervalle $[0, 2^N - 1]$

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

Logarithme et taille de donnée (1 sur 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.

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	$(169 = 84 \times 2 + 1)$
84	0	$(84 = 42 \times 2 + 0)$
42	0	$(42 = 21 \times 2 + 0)$
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

$a_{n-1}a_{n-2}\dots a_1a_0$ vaut $a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_1 2^1 + a_0 2^0$ en base 10

Exemple : 1010 vaut
$1 \times 2^3 + 0 \times 2^2 + 1^1 + 0 \times 2^0 = 2^3 + 2^1 = 8 + 2 = 10$

Représentation des relatifs, solution : Complément à deux

Sur n bits, en choisissant 00…000 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 code(a) + code(-a) = 0

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

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

CodeC2(a)+CodeC2(-a) = a-a+$2^8 = 0$ (sur 8 bits)

Complément à deux sur 8 bits : tous les entiers relatifs

entier relatif	Code(base10)	CodeC2(base2)
-128	128	1000 0000
-127	129	1000 0001
-126	130	1000 0010
...		
-1	255	1111 1111
0	0	0000 0000
1	1	0000 0001
2	2	0000 0010
...		
12	12	0000 1100
...		
127	127	0111 1111

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

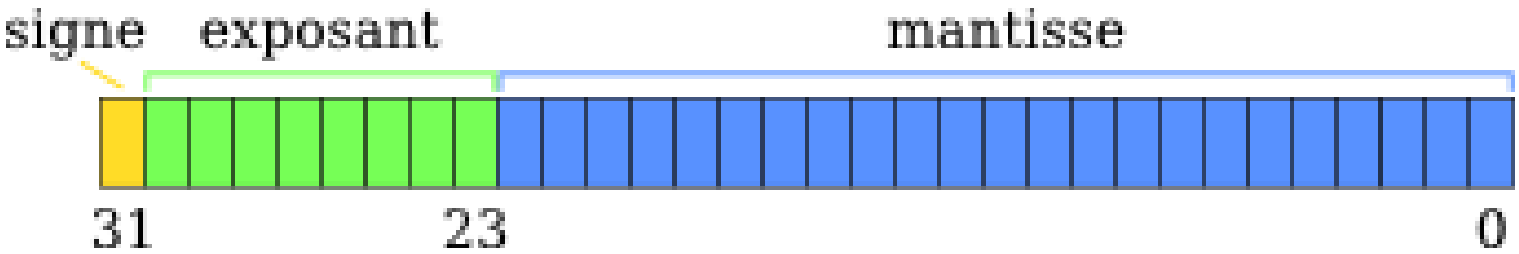
Soit un entier relatif positif a codé par les n chiffres binaires :

$a_{n-1}a_{n-2}\ldots a_1 a_0$

$$\begin{aligned}
\text{valeur(-a)} &= 2^n - \text{valeur(a)} \\
&= 2^n - (a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2 + a_0) \\
&= (2^{n-1} + 2^{n-1}) - (a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2 + a_0) \\
&= (1 - a_{n-1})2^{n-1} + 2^{n-1} - (a_{n-2}2^{n-2} + \ldots + a_1 2 + a_0) \\
&= \ldots\ldots\ldots\ldots\ldots\ldots \\
&= (1 - a_{n-1})2^{n-1} + (1 - a_{n-2})2^{n-2} + \ldots + (1 - a_0) + 1
\end{aligned}$$

Règle : écrire le code de la valeur absolue, inverser tous les bits, ajouter 1

Les nombres à virgule flottante



source Wikipedia.

Les nombres à virgule flottante

- Norme IEEE 754
- Codage par champ (exemple sur 32 bits) : Signe (1 bit), Exposant (8 bits), Mantisse (23 Bits)
- Valeur = $(-1)^{signe}$ * 1,Mantisse * $2^{Exposant-127}$
- Exceptions : 0, +Infini, -Infini, NaN, nombres proches de 0 ...
- Intervalle : [-3.4 10^{38} ;3.4 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 relatif : -4 ... 3,
- mais une seule addition.

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

Table d'addition (3 bits)

Récapitulatif : Pour 3 bits,

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

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

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

Equations

Question : d'après ce qui précède, vous sauriez résoudre les équations suivantes ?

$$x + x = 0$$

$$y + y = 1$$

$$z + z = 2$$

$$a + a = -2$$

Langage d'assemblage, langage machine

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

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`

Précompilation (*pre-processing*)

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

arm-eabi-gcc -S monprog.i

produit **monprog.s**

Le code source ≪ enrichi ≫ est transformé en langage d'assemblage (lisible)

instructions et données.

Assemblage

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 -x 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.

Edition de liens

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(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

Exemples

- reg12 ← reg14 + reg1
- registre4 ← le mot mémoire d'adresse 36000 + le registre A
- reg5 ← reg5 - 1
- le mot mémoire d'adresse 564 ← registre7

Convention de noms

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

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

- `Branch 125` : l'instruction suivante est désignée par une adresse « fixe ».
- `Branch -40` : l'instruction suivante est une adresse calculée.
- `Branch SiZero +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 ← 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 ← eax + 10.

En 6800 ou 68000 (Motorola) :

- addA 5000 signifie regA ← regA + Mem[5000]
- MOVE.W ♯500,D0 signifie regD0 ← 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 (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 (2/7) : par registre

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 (3/7) : immédiate

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 (4/7) : directe ou absolue

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] $\leftarrow$ 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 (5/7) : indirect par registre

Désignation registre/indirect par registre

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

- add r3, r3, [r5] signifie r3 $\leftarrow$ r3 + (le mot mémoire dont l'adresse est contenue dans le registre 5)
  On note r3 $\leftarrow$ r3 + mem[r5].

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

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.

- add r3, r3, [r5, #4] signifie r3 $\leftarrow$ r3 + 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 (7/7) : relatif au compteur programme

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 (1/4) : définition

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 (2/4) : exemple

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

zone DATA
XX: entier sur 4 octets : 0x56F3A5E2
YY: entier sur 4 octets : 0xAAF43210
```

Etiquettes (4/4) : correspondance étiquette/adresse

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 : 0xAAF43210
```

Programmation des structures de contrôles

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

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 !

Différents types de séquencement

- initialisation ou lancement d'un programme
- séquencement ≪ normal ≫
- rupture de séquence inconditionnelle
- rupture de séquence conditionnelle
- appels et retours de procédure/fonction
- interruptions
- exécution ≪ parallèle ≫

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 (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 (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).

Désignation de l'instruction suivante

- Désignation directe : l'adresse de l'instruction suivante est donnée dans l'instruction.
- Désignation relative : l'adresse de l'instruction suivante est obtenue en ajoutant un certain déplacement (peut être signé) au compteur programme.

Remarques :

- le mode de désignation en **ARM** est uniquement **relatif**.
- en général, le déplacement est ajouté à l'adresse de l'instruction qui suit la rupture. C'est-à-dire, $PC + 4 +$ déplacement.
  En ARM, $PC + 8 +$ déplacement.

Codage des structures de contrôle : notations

On dispose de sauts et de sauts conditionnels notés :

- branch etiquette et
- branch si cond etiquette.

cond est une expression booléenne portant sur Z, N, C, V

ATTENTION : les conditions dépendent du **type**. Par exemple, la condition $<$ à utiliser est différente selon qu'un entier est un naturel ou un relatif (l'interprétation du bit de poid fort est différente !).

Toute autre instruction (affectation, addition, ...) est notée `Ik`

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* ≪ *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.

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
```

Codage en ARM

`x←0;a←5;b←6; si a=b alors {x←1;} x←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 egal a 0) a etiq_suite
                I2
                I3
                I4
 etiq_suite:    I5
```

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
```

Codage en ARM

`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:
```

Exécution

```
I.0          mov r0 , #5
I.1          mov r2, #6
I.2          cmp r0,r2
I.3          bne sinon
I.4 alors:   mov r1, #1
I.5          b finsi
I.6 sinon:   mov r1, #0
I.7 finsi:   nop
```

Ligne	r0	r2	?=?	r1	proch Ligne
-1	?	?	?	?	0
0	5	?	?	?	1
1		6	?	?	2
2			faux	?	3
3				?	6
6				0	7
7					

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
```

Instruction *Tant que* : 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
```

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
```

Exécution

```
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
2			faux		3
3					4
4				1	5
5	2				6
6					2
...					

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.

Instruction *Pour* : Solution

```
I1; pour (i←0 à N) {I2; I3;I4;} I5;
    I1
    i<-0
    tant que i<=N
                    I2
                    I3
                    I4
                    i<-i+1
    I5
```

Exercice

Deux boucles imbriquées

```
pour (i=0 a N)
                    pour (j=0 a K)
                                        I2 ;I3
```

Expression conditionnelle complexe avec des *ou* : Solution I

```
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:
```

Solution II

```
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.

Expression conditionnelle complexe avec des *et* : solution

```
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:
```

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<b : x<-x+5
  a=b : x<-x+1
  a>b : x<-x+9
```

```
a dans r0, b dans r1, x dans r2
    cmp r0,r1
    addlt r2, r2, #5
    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

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Utilité-Nécessité des fonctions et procédures

A quoi servent les fonctions et procédures :

- **Structurer** le code (nommer un bloc d'instruction)
- Eviter de dupliquer du code
- Eviter les structures de contrôles imbriquées
- Permettre l'utilisation de variables **locales**
- Permettre la définition de bibliothèques
- Programmer avec de la **récursivité**
- Préparer la programmation orientée objet

Rappel : en C, et dans beaucoup de langage, tout ou presque est fonction. Il n'y a pas de script C (i.e. code hors fonction). Par contre, il peut y avoir des variables globales (!)

Un exemple en langage de ≪ haut niveau ≫ (1 /2)

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)

- 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 1er appel revient à exécuter le corps de la fonction en remplaçant `x` par `i+1` ; le 2ème appel consiste en l'exécution du corps de la fonction en remplaçant `x` par `2*(i+5)`

Tentative de traduction en ARM

Tentative de traduction en ARM

Utilisation de registres

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 *r*0,*r*1,*r*2
- *z* dans *r*3, *p* dans *r*4
- la valeur *x* dans *r*5
- le résultat de la fonction dans *r*6
- si on a besoin d'un registre pour faire des calculs on utilisera *r*7 (variable temporaire)

Remarque :

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

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-1er-appel——
        add r5, r0, #1      @ x ← i + 1
        appeler PP
        mov r1, r6          @ j ← ...
@ ——Fin-1er-appel——
@ ——Début-2ème-appel——
        add r7, r0, #5      @ r7 ← i + 5
        mov r5, r7, lsl #1  @ x ← 2 * r7
        appeler PP
        mov r2, r6          @ k ← ...
@ ——Fin-2ème-appel——
```

Problème :

appeler et retourner ?

Quel est le problème ?

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.

Adresse de retour

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.

Où est gardée cette adresse ?

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.

EcrNdecim32 dans `es.s`

Rappel procedures d'affichage (es.s) :

```
.global EcrNdecim32

@ EcrNdecim32 : ecriture en decimal de l'entier dans
EcrNdecim32 : mov ip, sp
stmfd 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

```
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——
```

Exécution

```
l.0 PP :   add r3, r5, #1
l.1        add r4, r3, #2
l.2        mov r6, r4
l.3        bx lr
l.4 main : mov r0, #0
l.5        add r1, r0, #3
l.6        add r5, r0, #1
l.7        bl PP l
l.8        mov r1, r6
l.9        add r7, r0, #5
l.10       mov r5, r7, lsl #19
l.11       bl PP
l.12       mov r2, r6
```

l.	r0	r1	r3	r4	r5	r6	lr	> l.
-1	?	?	?	?	?	?	?	4
4	0	?	?	?	?	?	?	5
5		3	?	?	?	?	?	6
6			?	?	1	?	?	7
7			?	?		?	8	0
0			2	?		?		1
1				4		?		2
2						4		3
3								8
8		4						9
19								10
10					10			11
11							12	0
...								

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 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`, ...)

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
```

Fonctions récursives (2/2)

Même chose avec les variables locales !

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

Conclusion : fonctions récursives

Conclusion 1

On ne peut pas travailler avec une seule zone de paramètres, il en faut une pour chaque appel et pas pour chaque fonction.
Les paramètres effectifs (ou arguments) sont attachés à l'appel d'une fonction et pas à l'objet fonction lui-même

Conclusion 2

On ne peut pas travailler avec une seule zone pour les variables locales, il en faut une pour chaque appel et pas pour chaque fonction.
Les variables locales sont attachées à l'appel d'une fonction et pas à l'objet fonction lui-même

Programmation de procédures (suite)
Utilisation de la pile

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Zones de mémoire dynamique

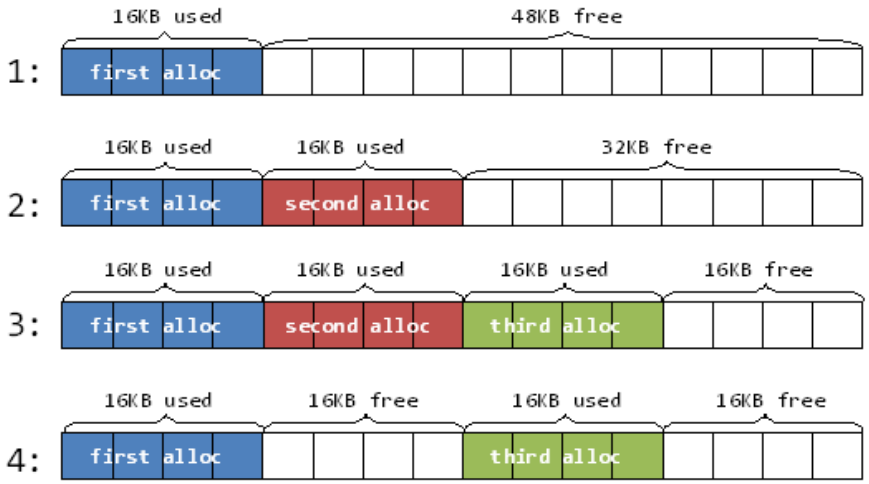
Parmi les zones de mémoire dynamique :

- le tas (heap) (malloc, free ; new, delete),
- la file mécanisme dit FIFO : *First In First Out* (Premier entré, premier sorti) (enfiler, défiler)
- la pile mécanisme dit LIFO : *Last In First Out* (Dernier entré, premier sorti) (empiler, dépiler)

Attention, le tas (heap) est aussi une structure de données qui permet de représenter un arbre dans un tableau (ex. : tri par tas), mais cela n'a que peu de rapport avec la zone de mémoire dynamique.

Notion de tas

Exemple : malloc(first) ; malloc(second) ; malloc(third) ; free(second) ;



(source Qualcomm)

Notions associées

- fragmentation (et défragmentation), ramasse miette (garbage collecting),
- realloc.

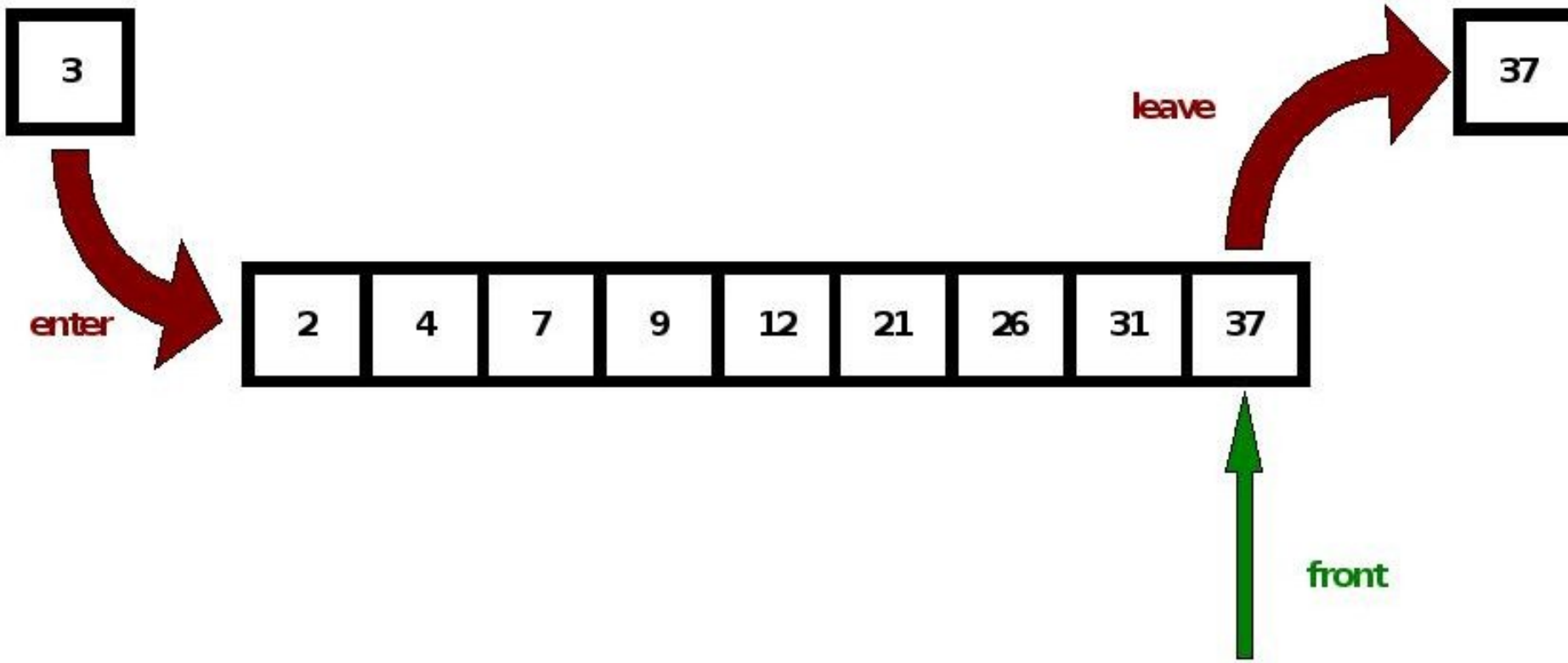
défragmentation, realloc dans le tas

voir animation défragmentation
cours06_Pile/Strip-Defragmentation-Windows-95-650-final.gif
(source CommitStrip)

voir animation realloc cours06_Pile/post_1_sj_realloc_std_small.gif
(source Dmitry Frank)

Notion de file

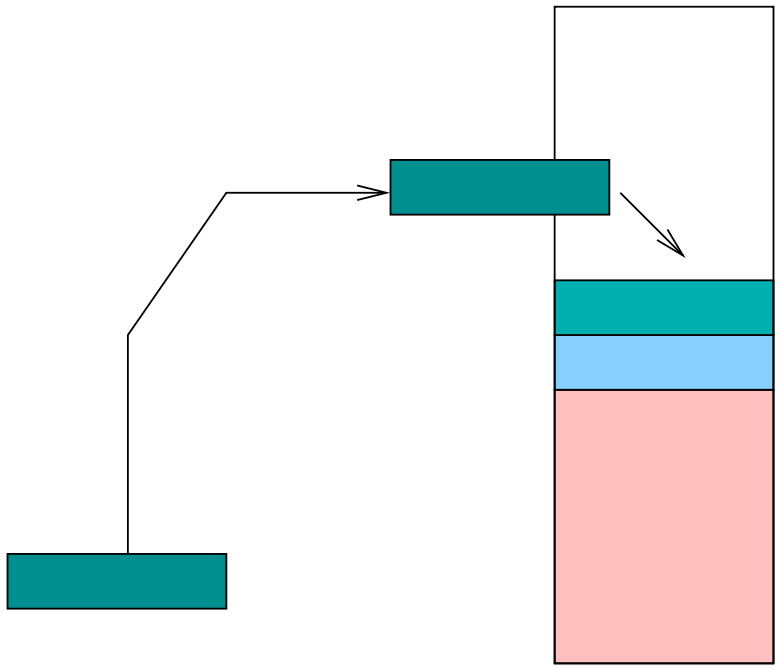
Exemple : enfiler(3) ; défiler(X) ;



(source wikipedia)

Notion de pile

Exemple : empiler(X), ...(autres instruction hors pile) ..., dépiler(Y)



Meilleur choix

Parmi les zones de mémoire dynamique :

- le tas (heap) (malloc, free ; new, delete),
- la file mécanisme dit FIFO : *First In First Out* (Premier entré, premier sorti) (enfiler, défiler)
- la pile mécanisme dit LIFO : *Last In First Out* (Dernier entré, premier sorti) (empiler, dépiler)

Meilleur choix : la pile.

Mécanisme de pile

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

Deux opérations possibles :

Dépiler : suppression de l'élément en tête de la pile

Empiler : ajout d'un élément en tête de la pile

Comment réaliser une pile ? (1 /4)

- Une zone de mémoire,
- Un repère sur la tête de la pile
  SP : pointeur de pile, *stack pointer*
- Deux choix indépendants :
  - Comment progresse 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*)

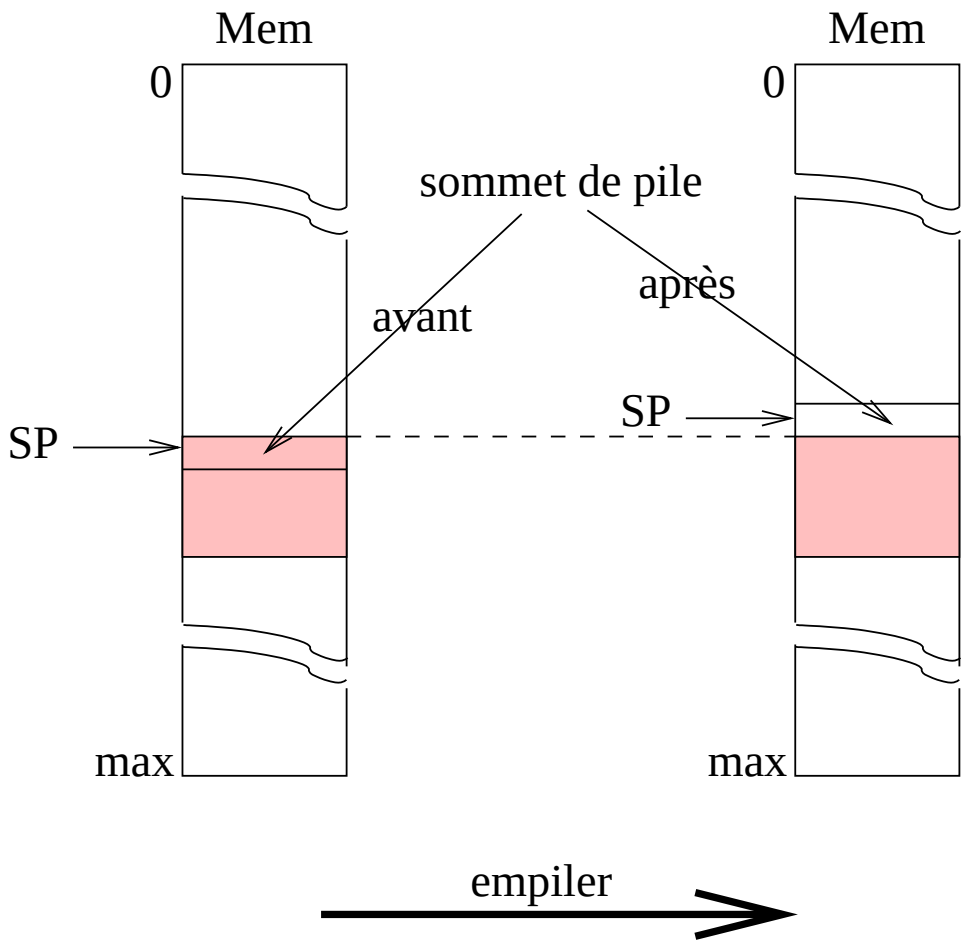
Comment réaliser une pile ? (2 /4)

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

sens évolution	croissant	croissant	décroissant	décroissant
repère	1^{er} vide	der^{er} plein	1^{er} vide	der^{er} 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)

Comment réaliser une pile ? (3 /4)



Comment réaliser une pile ? (3 /4)

En Arm, empiler R3 (convention full descending) :

- push {R3}
- stmfd 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

Appel de procédure, deux actions exécutées par le processeur :

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

Au retour, *PC* prend pour valeur l'adresse en sommet de pile puis le sommet est dépilé : ***PC*** $\leftarrow$ **`depiler()`**.

Remarque : Ce n'est pas la solution utilisée par le processeur ARM.

Application sur l'exemple

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

PC	instructions			état de la pile
10	*A*1			{}
11	*A*2			{}
12	`saut` 20 (*B*)			`empile` 13
20		*B*1		{13}
21		*B*2		{13}
22		*B*3		{13}
23		retour		`sommet` = 13
13	*A*3			{}
14	`saut` 30 (*C*)			`empile` 15
30		*C*1		{15}
31		`saut` 20 (*B*)		`empile` 32
20			*B*1	{32 ; 15}
21			*B*2	{32 ; 15}
22			*B*3	{32 ; 15}
23			retour	`sommet` = 32
32		*C*2		{15}

Trace d'exécution

33		`cond` :`saut` 30 (*C*)			`empile` 34
30			*C*1		{34 ; 15}
31			`saut` 20 (*B*)		`empile` 32
20				*B*1	{32 ; 34 ; 15}
21				*B*2	{32 ; 34 ; 15}
22				*B*3	{32 ; 34 ; 15}
23				retour	`sommet` = 32
32			*C*2		{34 ; 15}
33			`cond` :`saut` 30		(pas d'appel à *C*)
34			*C*3		{34 ; 15}
35			*C*4		{34 ; 15}
36			retour		`sommet` = 34
34		*C*3			{15}
35		*C*4			{15}
36		retour			`sommet` = 15
15	*A*4				{ }

Appel/retour : solution utilisée avec le processeur ARM

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

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 BL d'ARM)

En utilisant l'instuction BL (Branch and Link) d'ARM :

```
10   A1
11   A2
12   BL B     (appel)
13   A3
14   BL C     (appel)
15   A4
```

```
20   empiler LR
21   B1
22   B2
23   B3
24   retour: dépiler dans PC
```

```
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
```

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
```

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.

Exemple

```
procedure A ** procedure principale, sans parametre
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

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

Il faut un emplacement mémoire pour la variable locale u $u \leftarrow 2$:

```
mov r0, #2
str r0, [adr_u]
```

Appel de $B(u+3)$:

Il faut un emplacement mémoire pour le paramètre x

et on y range la valeur de $u+3=5$

```
ldr r0, [adr_u]
add r0, r0, #3
str r0, [adr_x]
```

Le flot d'exécution est en début de la procédure B

Il faut deux emplacements mémoire pour les variables locales s et v

$s \leftarrow x+4$

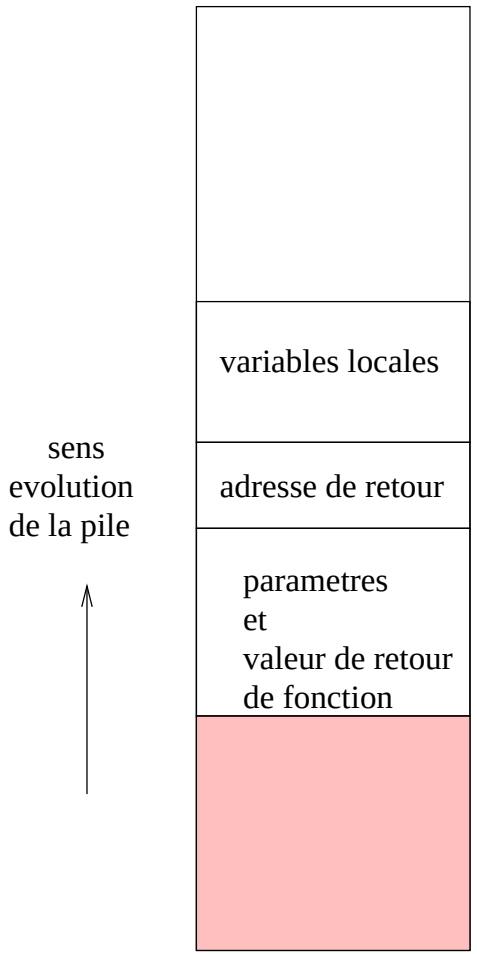
Remarques

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)

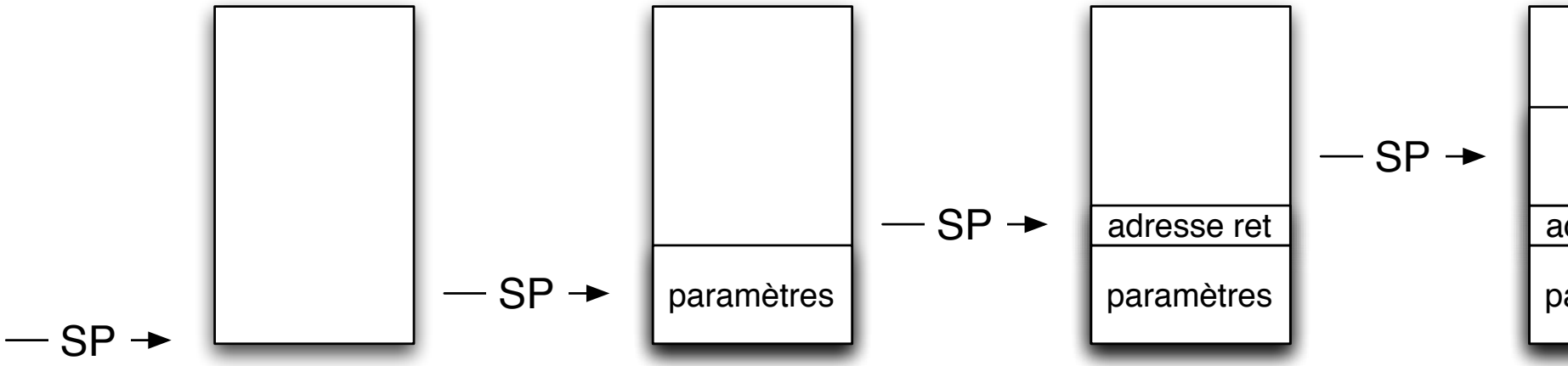
Organisation des informations dans la pile lors de l'exécution d'une procédure



Organisation du code

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 variables 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 : $[SP, \sharp dpl]$

$dpl >= 0$

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

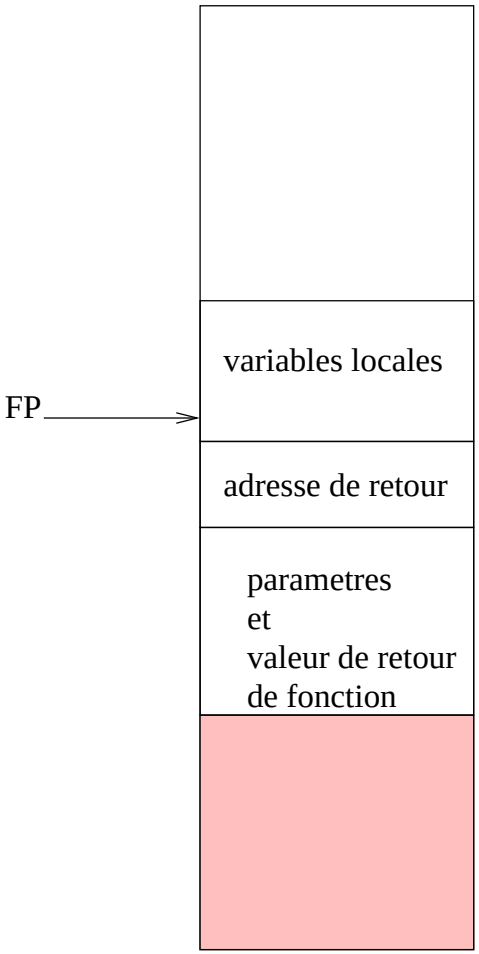
Accès aux variables et paramètres : *frame pointer* (1/2)

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**.

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



Accès à un paramètre :

$[fp, \sharp dpl\_param]$

$dpl\_param > 0$

Accès à une variable locale :

$[fp, \sharp dpl\_varloc]$

$dpl\_varloc < 0$

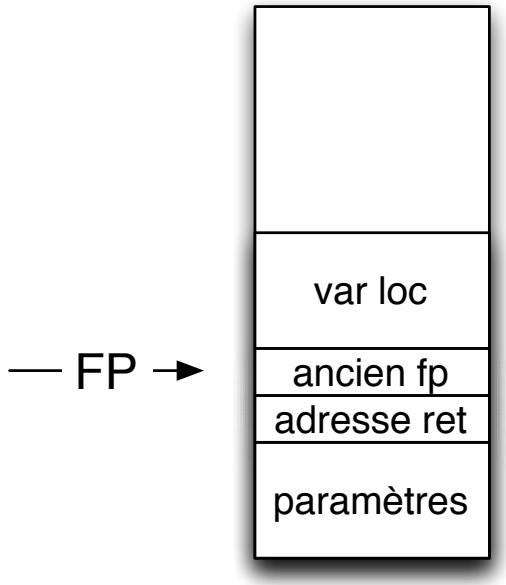
Organisation du code en utilisant le registre *frame pointer*

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 variables locales
restaurer `fp`
récupérer adresse de retour
retour

Organisation de la pile lors de l'exécution avec *frame pointer*



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*

En ARM : code de *B*

```
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 @ rÃⓒservation 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 Ã  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 Ã  C @@@

@ v<-2
mov r1,#2
str r1,[fp,#-8]

@ passe de s+v en parametre de C
ldr r1, [fp,#-4]
ldr r2, [fp,#-8]
add r1,r1,r2
push {r1}

bl C @ appel C

add sp,sp,#4 @ depile parametre

@@@ fin du corps de B @@@
add sp,sp,#8 @ depile s,v

pop {fp} @ retour a l'ancien fp

@ recuperation adresse retour
pop {lr} @ ldr lr, [sp]
          @ add sp,sp,#4

bx lr @ retour
```

Programmation des appels de procédure et fonction (fin)

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

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

1. Le résultat d'une fonction est calculé par l'appelée
2. 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 l'exemple, **la pile**.

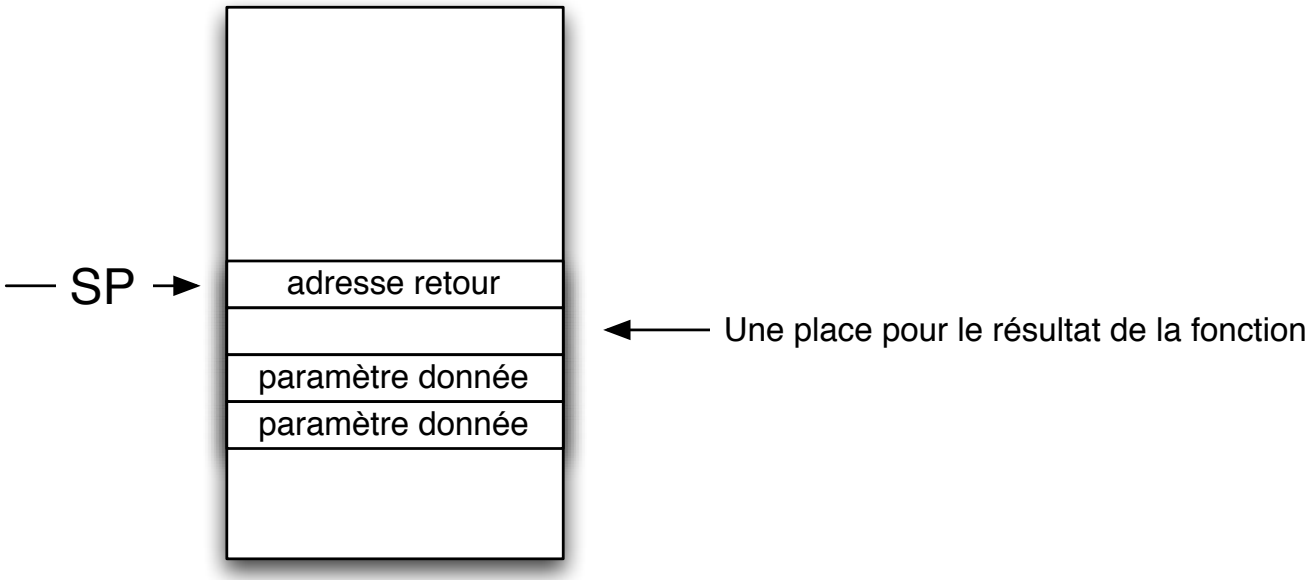
Résultat dans la pile (1/3)

1. avant l'appel, **L'appelant** réserve une place pour le résultat dans la pile
2. **L'appelée** rangera son résultat dans cette case dont le contenu sera récupéré par l'appelant après le retour

Résultat dans la pile (2/3)

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

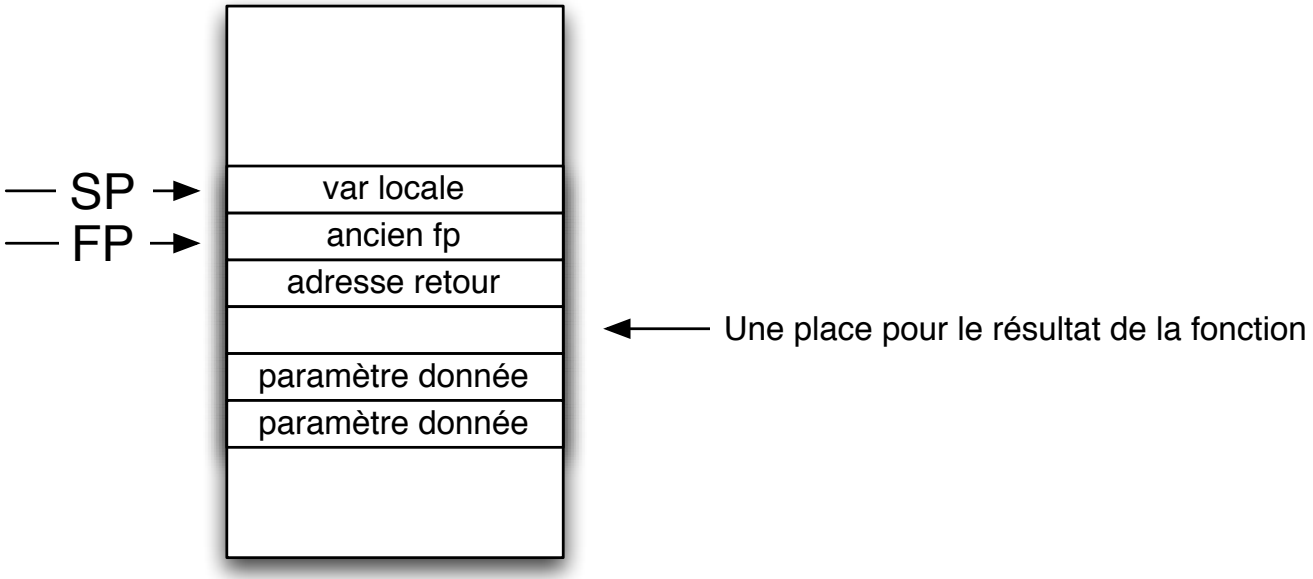
- Les valeurs des deux paramètres sont empilés
- Une case est réservée pour le résultat de la fonction



Résultat dans la pile (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 données par les adresses : $fp + 8 + 4$ et $fp + 8 + 8$ et
3. La case résultat par l'adresse $fp + 8$.



Structure du code de l'appel de la fonction et du corps de la fonction

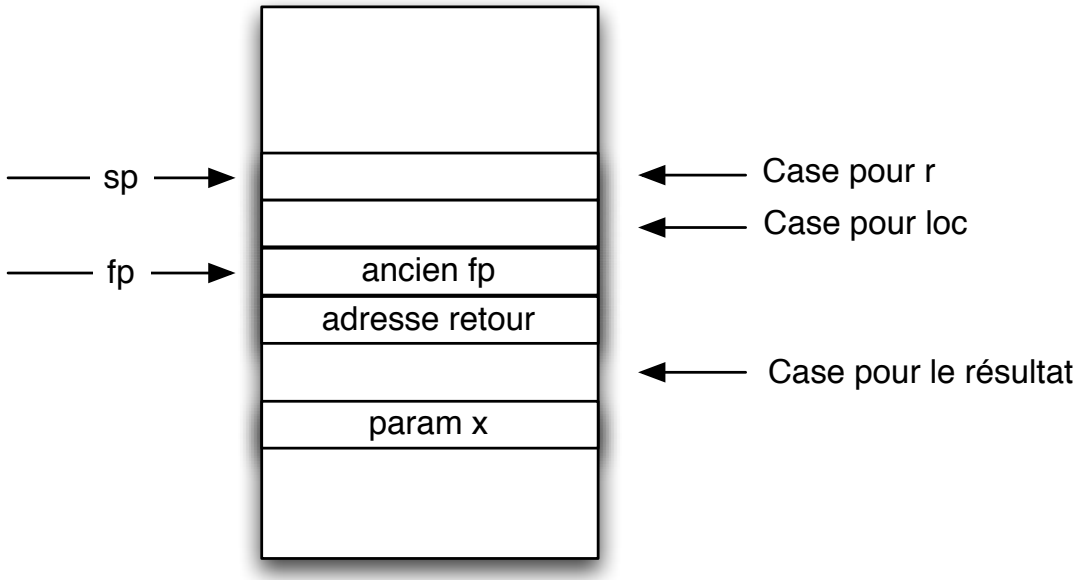
appelant `P` :
préparer et empiler les paramètres
réserver la place du résultat dans la pile
appeler `Q` : `BL Q`
récupérer le résultat
libérer la place allouée aux paramètres
libérer la place allouée au résultat

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`

Application : codage d'une fonction factorielle avec des variables locales

```
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 du corps de factorielle juste après l'appel dans main



Nouvelle version de la fonction `fact`

```
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          @ r1=x-1
        str r1, [sp]
        sub sp, sp, #4          @ case resultat
        bl fact                 @ appel
        ldr r1, [sp]            @ recuperer resultat
        add sp, sp, #8          @ desallouer param et res
        @ apres l'appel
        str r1, [fp, #-4]       @ loc=fact(x-1)
        ldr r0, [fp, #+12]      @ r0=x
        ldr r1, [fp, #-4]       @ r1=loc
        mul r2, r0, r1          @ x*loc
        str r2, [fp, #-8]       @ r=x*loc
finsi:
        ldr r2, [fp, #-8]
        str r2, [fp, #+8]       @ return r
        @ recuperer place var loc
        add sp, sp, #8
        pop {fp}                @ recuperer fp
        @ retour
        pop {lr}                @ recuperer lr
        bx lr
```

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.

Application à l'exemple de la fonction `fact`

Le code de la fonction `fact` utilise les registres `r0, r1, r2`.

```
fact:   @ empiler adr retour
        push {lr}
        @ mise en place fp et allocation loc et r
        push {fp}
        mov fp, sp
        sub sp, sp, #8
        @ sauvegarde de r0, r1, et r2 (empiler)
        push {r2}
        push {r1}
        push {r0}
        @ if x==0 ...
        ...
```

```
        @ restaurer les registres r0, r1, r2 (depiler)
        pop {r0}
        pop {r1}
        pop {r2}
        @ desallouer var locales
        add sp, sp, #8
        pop {fp} @ ancien fp
        @ depiler adr retour dans lr
        pop {lr}
        bx lr @ retour
```

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

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`

Situation : comment faire +1 par programme ?

- 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)

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 cela marche :-)
- Ns (ou t) sont des références ...
- C'est la suite du drame du passage de paramètre par valeur nom

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 (donnees x, y : entier ; resultat 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, les valeurs étant substituées aux paramètres formels lors de l'exécution du corps de la procédure.
- Le paramètre effectif associé au paramètre formel résultat 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 passée en argument.

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 (donnees x, y : entier ; adresse z : entier)
u,v : entier

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

L'exemple d'appel traité

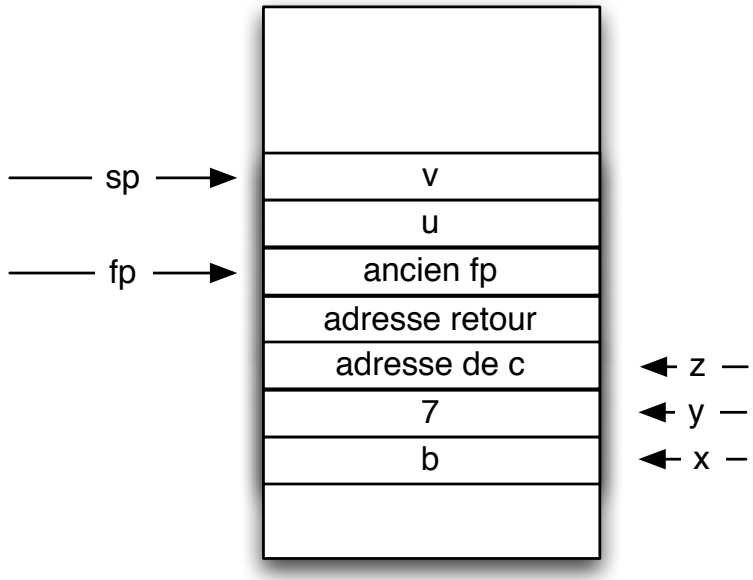
a,b,c : entier

b=3 ;

....

XX (b, 7, adresse de c) ;

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



main

```
        .bss
a:      .skip 4
b:      .skip 4
c:      .skip 4
        .text
main:
        ...
        ldr r0, LD_c        @ r0 ← adresse de c
        sub sp, sp, #4
        push {r0}           @ empiler adresse de c

        mov r0, #7          @ r0 ← 7
        push {r7}           @ empiler 7

        ldr r0, LD_b
        ldr r0, [r0]        @ r0 ← valeur de b
        push {r0}           @ empiler b
        bl XX               ...

        ...
        bx lr
LD_a:   .word a
LD_b:   .word b
LD_c:   .word c
```

Procédure XX

```
XX:
        ...
        ldr r0, [fp, #+16]    @ u ← x
        str r0, [fp, #-4]

        ldr r0, [fp, #+12]    @ v ← y + 2
        add r0, r0, #2
        str r0, [fp, #-8]
        ...
        ldr r0, [fp, #-4]
        ldr r1, [fp, #-8]
        add r0, r0, r1        @ calcul de u + v

        ldr r2, [fp, #+8]     @ r2 ← z, i.e., adresse c
        str r0, [r2]          @ mem[z] ← u + v, i.e., mem[adresse c] ← u + v
        ...
```

Conclusion / Rappel : Structure générale du code d'un appel et du corps de la fonction ou procédure

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`

La vie des programmes

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Plan

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 :

- Phase d'analyse
  - Pré-traitement
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
- 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.

Compilation et interprétation

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

- un compilateur
- un interptéteur

Compilateurs et interpréteurs partagent la première phase de travail (phase d'analyse).
Compilateurs et interpréteurs se distinguent au moment de l'éxécution :

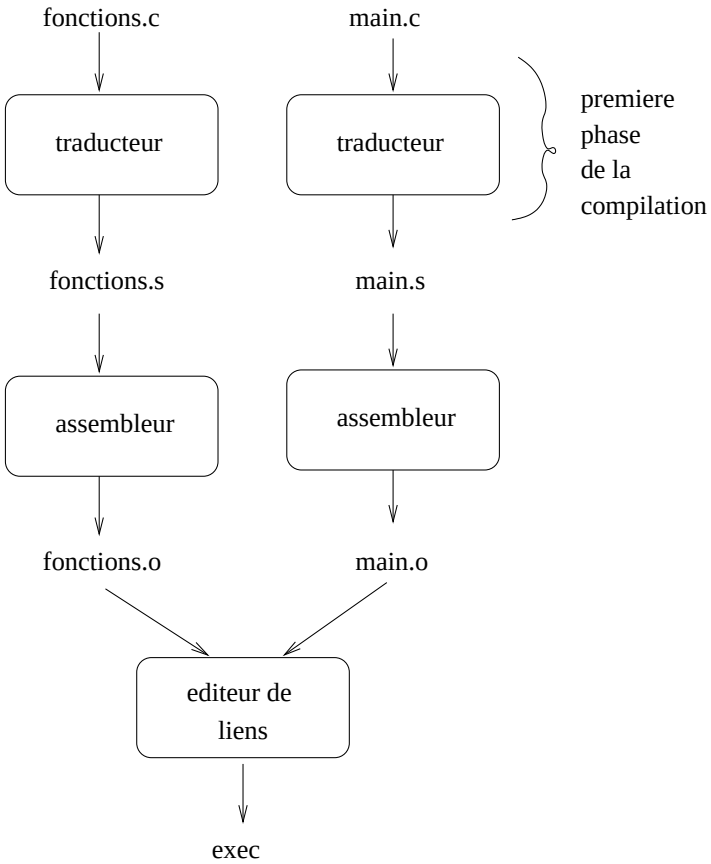
- le code cible produit par un compilateur est exécuté directement par la machine cible
- la structure intermédiaire obtenue par l'interpréteur est exécutée par l'interpréteur lui-même (comme sur une machine virtuelle)

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.

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 enchaine 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).

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

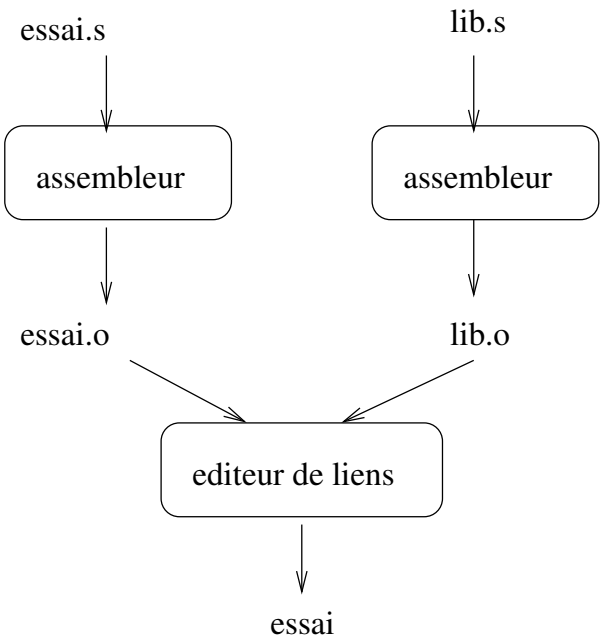
essai.s

```
    .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 enchaine 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`.

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).

Exemple : traduction d'une conditionnelle

Prenons une conditionnelle :

`si Condition alors { ListeInstructions }`

elle sera traduite selon le schéma (récursif) :

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

Les schéma de traduction

Il y a ainsi, des schémas (récursifs) 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.

Problématique

L'objectif de l'assembleur est de produire du code binaire à partir du langage d'assemblage.

Il n'est pas toujours possible de produire du binaire qui puisse être directement copié en mémoire pour deux raisons principalement :

Cas 1 On ne connaît pas en général l'adresse à laquelle les zones **text** et **data** doivent être rangées en mémoire.

Cas 2 Le programme peut faire référence à des noms qui ne sont pas définis dans le fichier en cours de traduction.

- Dans le premier cas, on peut produire une **image** du binaire à partir de l'adresse 0, à charge du matériel de translater l'image à l'adresse de chargement pour l'exécution (il faut garder les informations permettant de savoir quelles sont les adresses à translater)
- Dans le deuxième cas on ne peut rien faire.

Premier cas

```
    .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
```

L'adresse associée au symbole `xx` est : adresse de début de la zone `data` + 4 mais encore faut-il connaître l'adresse de début de la zone `data` !

Si on considère que la zone `data` est chargée à l'adresse **0**, l'adresse associée à `xx` est alors **4**. Si on doit translater le programme à l'adresse **2000**, il faut se rappeler que à l'adresse `LD_xx` on doit modifier la valeur **4** en **2000 + 4**.

Cette information à mémoriser est appelée une donnée de translation (**relocation** en anglais).

Deuxième cas

```
    .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
```

- Dans le fichier `essai.o` il n'est pas possible de calculer le déplacement de l'instruction `bl add1` puisque l'on ne sait pas où est l'étiquette `add1` quand l'assembleur traite le fichier `essai.s`. En effet l'étiquette est dans un autre fichier : `lib.s`
- **Reprenons l'exemple en langage C.** Suite à la traduction en langage d'assemblage, dans le fichier `main.s` les références aux fonctions `somme` et `max` ne peuvent être complétées car les fonctions en question ne sont pas définies dans le fichier `main.c` mais dans `fonctions.c`.

Deuxième cas

Que faire pour trouver la(/les) adresse(s), le(s) déplacement(s) ?

- Dans le deuxième cas on ne peut rien faire
- Pour l'instant, la traduction va être incomplète

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 bcle
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.

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 chaines à laquelle la table des symboles fait référence.

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
```

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)
  O (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

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 <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
```

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)
```

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 bcle
     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            00000000 000000 000000 00      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)
```

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 assembleur

1. Reconnaissance de la syntaxe (lexicographie et syntaxe)
2. repérage des symboles. Fabrication de la table des symboles utilisée par la suite dès qu'une référence à un symbole apparait.
3. Traduction = production du binaire.

Rôle de l'éditeur de liens

Le travail de l'éditeur de liens consiste à :

- Identifier les symboles définis et exportés d'un côté et les symboles non définis de l'autre (importés).
- Rassembler les zones de même type et effectuer les corrections nécessaires.

Remarque : L'édition de liens rassemble des fichiers objets. L'assembleur ne peut pas produire du binaire exécutable, il produit un binaire incomplet dans lequel il conserve des informations permettant de le compléter plus tard.
La phase d'édition de liens bien qu'elle permette de résoudre les problèmes de noms globaux produit elle aussi du binaire incomplet car les adresses d'implantation des zones `text` et `data` ne sont pas connues.

Phase de chargement : production de binaire exécutable

Le calcul des adresses définitives peut avoir lieu de façon statique ou de façon dynamique au moment ou on en a besoin.
Deux solutions possibles :

- édition de liens au moment du chargement en mémoire (au lieu de rassembler le contenu de deux fichiers complets, on ne charge que le code des fonctions utilisées, par ex. pour les bibliothèques) ou
- édition de liens au moment de l'exécution (appel de la fonction) ce qui permet de partager des fonctions et de ne pas charger en mémoire plusieurs fois le même code.

Que contient un fichier exécutable ? entête

```
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:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x810c
  Start of program headers:          52 (bytes into file)
  Start of section headers:          144432 (bytes into file)
  Flags:                             0x2, has entry point, GNU EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         24
  Section header string table index: 21
```

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

Année 1, l'exécution des programmes en langage machine.

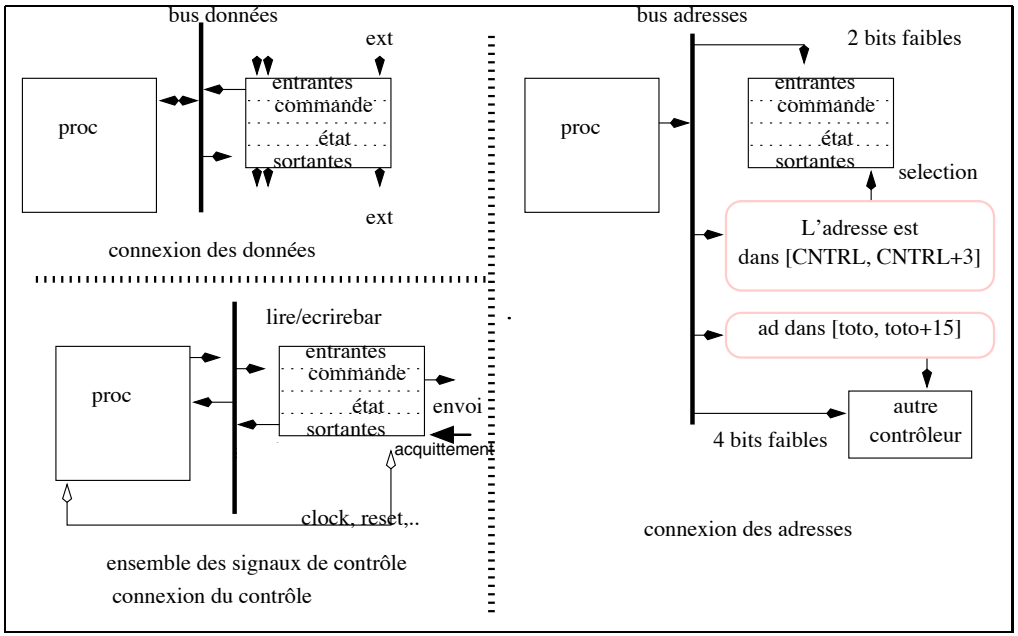
(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

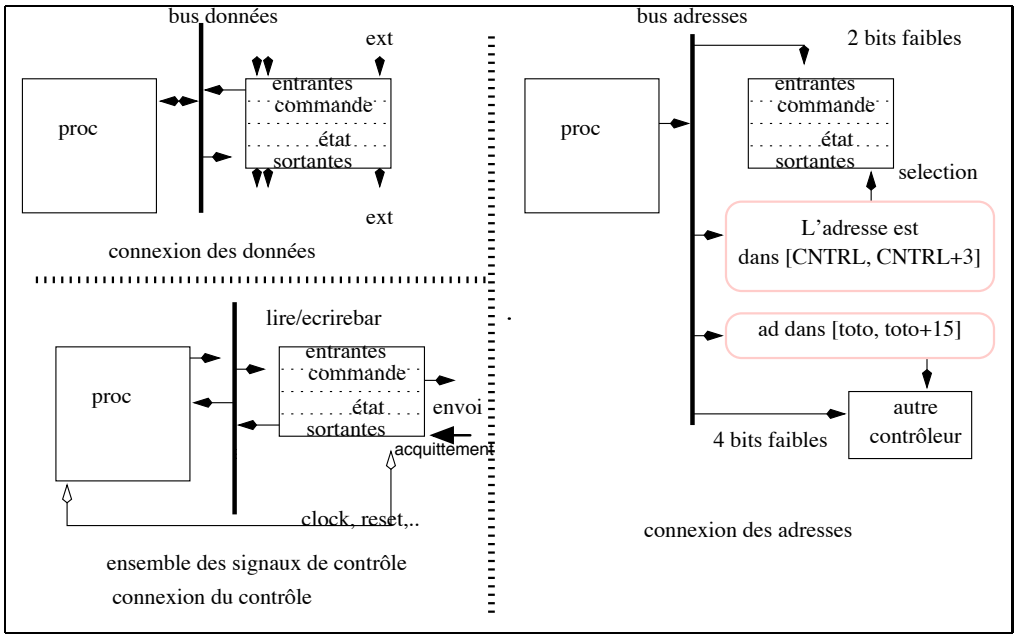
13 janvier 2021

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



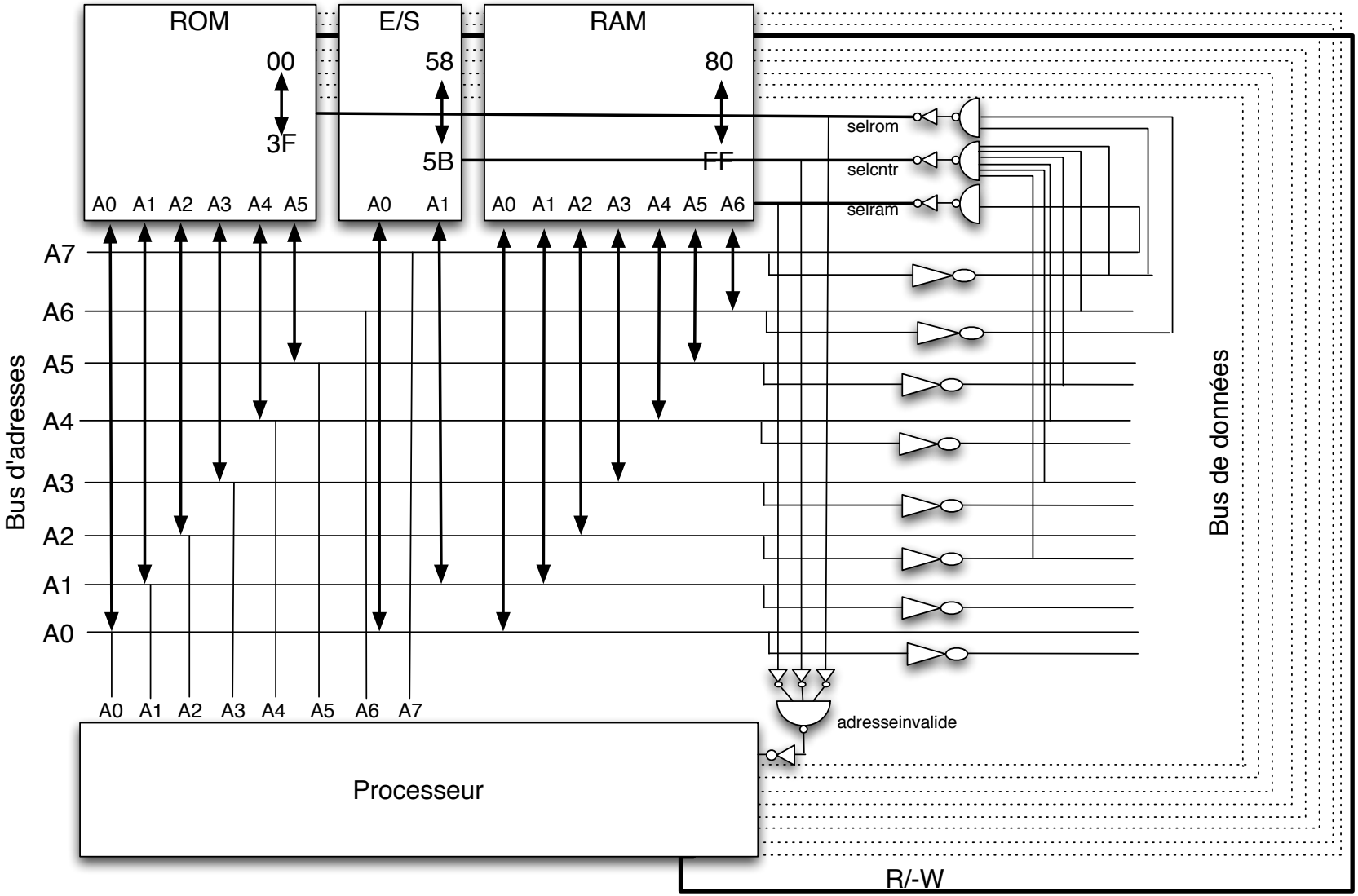
- bus données (lié au processeur)
- deux bits de bus adresses (pour sélectionner l'un des 4 mots `CNTRL` +0, +1, +2 ou +3)
- un signal de sélection provenant du décodeur d'adresses
- le signal *Read*/$\overline{\textit{Write}}$ du processeur
- un paquet de données (8 fils) venant du monde extérieur. 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 `Mdonnéessort` 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

Année 1, l'exécution des programmes en langage machine.

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Les instructions

Les instructions sont décrites ci-dessous. On donne pour chacune une syntaxe de langage d'assemblage et l'effet 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`.

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 adresse ou bien une constante.

Le codage est le suivant :

`clr`	1	
`ld #vi`	2	vi
`st ad`	3	ad
`jmp ad`	4	ad
`add ad`	5	ad

Exemple de programme (1/2)

```
        ld #3
        st 8
et :    add 8
        jmp et
```

Que contient la mémoire après assemblage (traduction en binaire) et chargement en mémoire ? On suppose que l'adresse de chargement est `0`.

0	2	ld #3
1	3	
2	3	st 8
3	8	
et=4	5	add 8
5	8	
6	4	jmp et = jmp 4
7	4	
8		

Algorithme d'interprétation

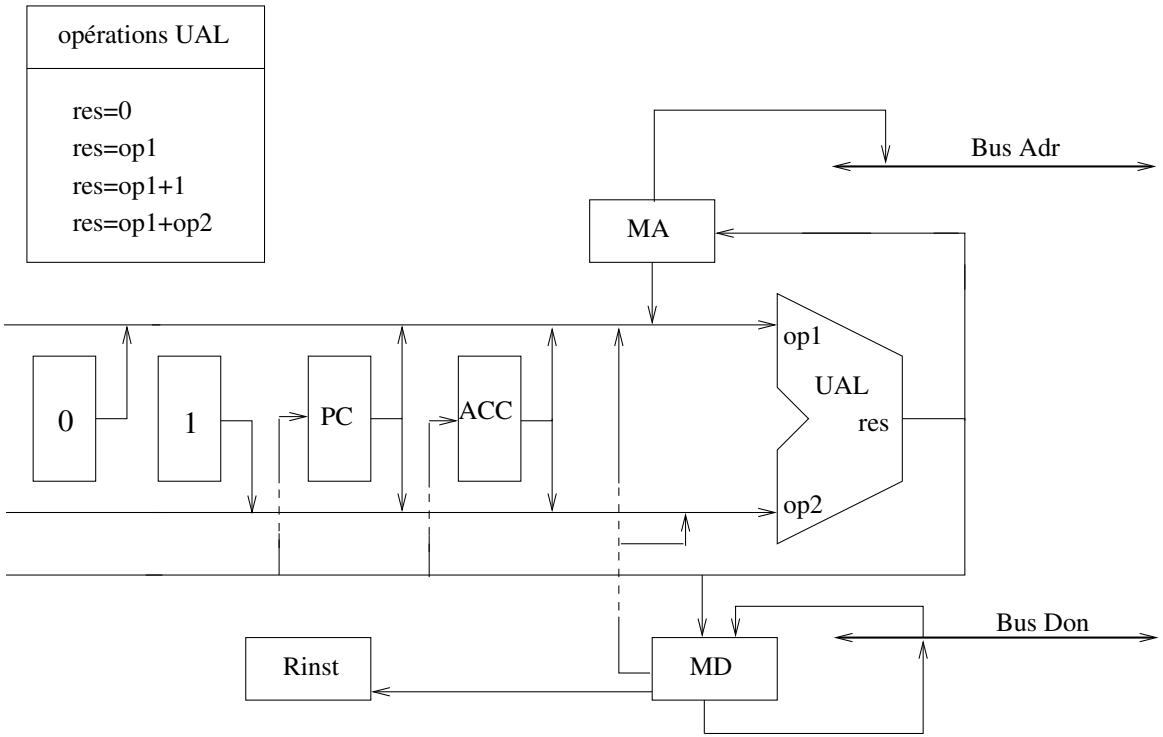
En adoptant un point de vue fonctionnel, en considérant les ressources du processeur comme les variables d'un programme, l'algorithme d'interprétation des instructions peut être décrit de la façon suivante :

```
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 (+, -, and,...). Cette partie est reliée à la mémoire par **les bus adresses et données**. On l'appelle Partie Opérative.



Micro-actions et micro-conditions

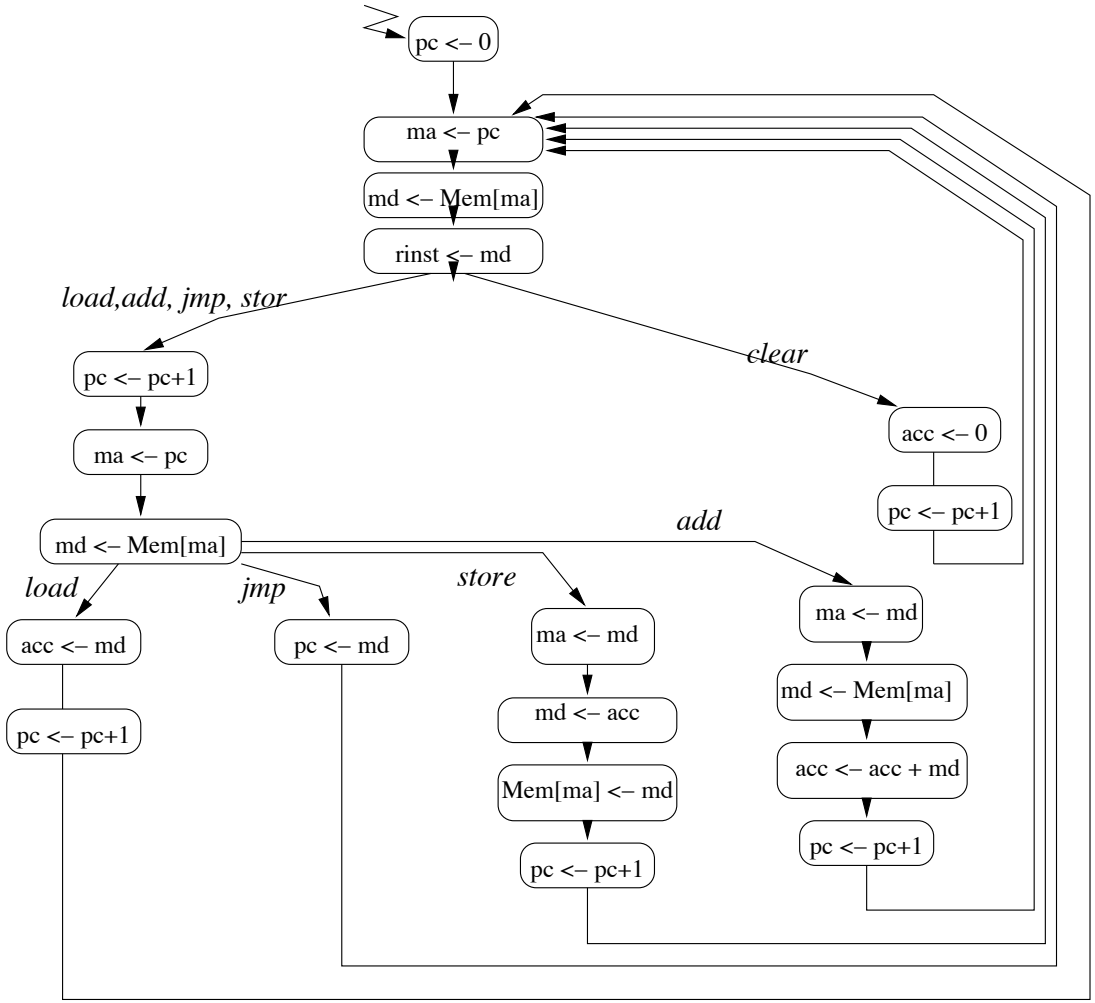
On fait des hypothèses FORTES sur les transferts possibles :

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`
$\mathbf{reg_0} \leftarrow \mathbf{0}$	affectation	reg_0 est `pc`, `acc`, `ma`, ou `md`
$\mathbf{reg_0} \leftarrow \mathbf{reg_1}$	affectation	reg_0 est `pc`, `acc`, `ma`, ou `md` reg_1 est `pc`, `acc`, `ma`, ou `md`
$\mathbf{reg_0} \leftarrow \mathbf{reg_1} + 1$	incrémentation	reg_0 est `pc`, `acc`, `ma`, ou `md` reg_1 est `pc`, `acc`, `ma`, ou `md`
$\mathbf{reg_0} \leftarrow \mathbf{reg_1} + \mathbf{reg_2}$	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)

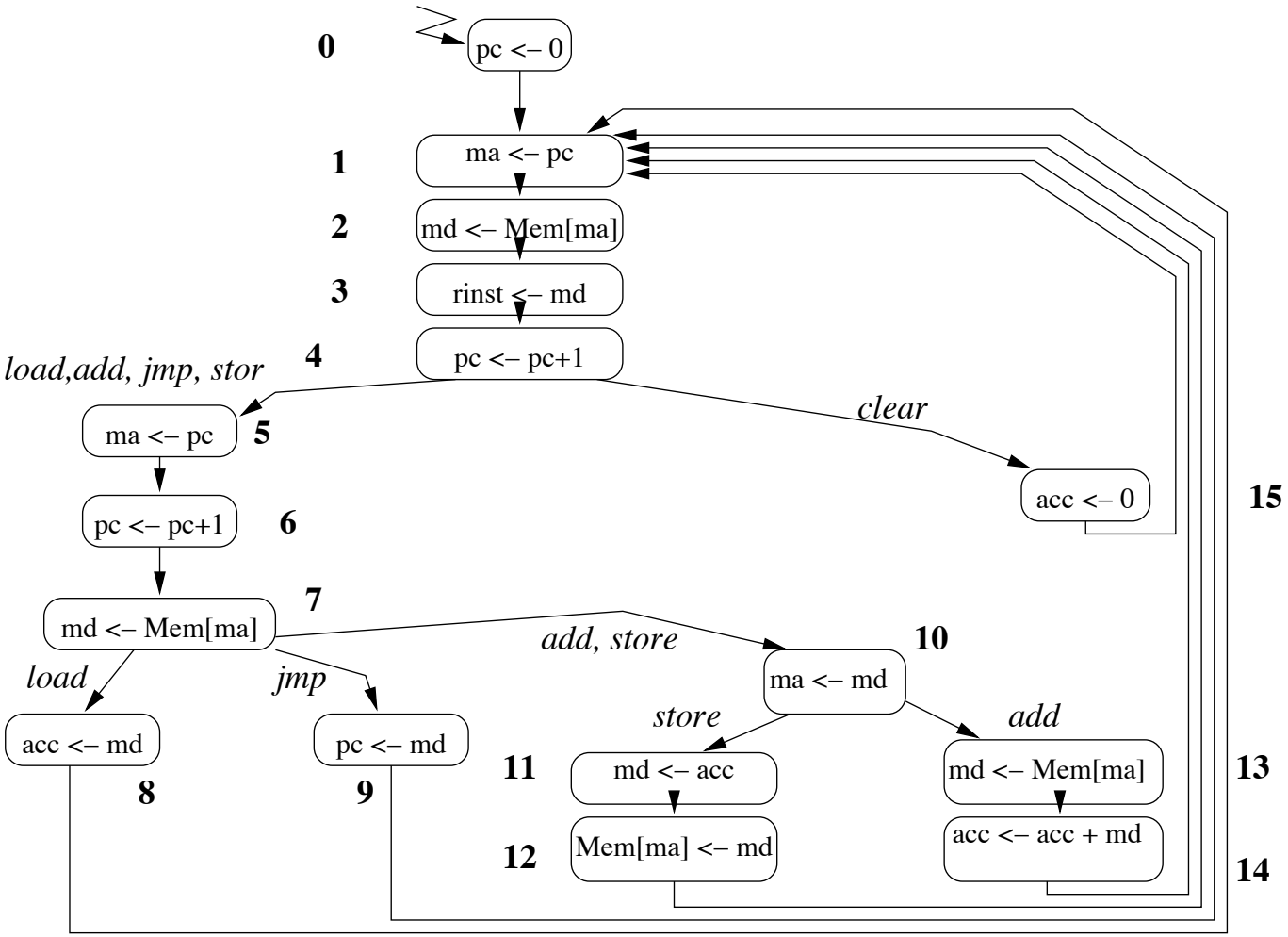
Ces types de transferts et les tests constituent le langage des micro-actions et des micro-conditions.

Une première version



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

Version amélioré de l'automate d'interprétation du langage machine : Partie Contrôle.



Exemple de code

étiquette	mnémonique ou directive	référence	mode adressage
	.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

état	*pc*	*ma*	*md*	*rinst*	*acc*	*mem*[15]
0	0					
1		0				
2			1			
3				1		
4	1					
15					0	
1		1				
2			2			
3				2		
4	2					
5		2				
6	3					
7			8			
8					8	
1		3				
2			3			
3				3		
4	4					
5		4				
6	5					
7			15			
10		15				

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