UNIVERSITE Joseph FOURIER, Grenoble
U.F.R. d' Informatique et Maths. Appliquées

Licence Sciences et Technologie
Parcours MIN, INF, IAG, BIN
UE INF241

Introduction aux Architectures Logicielles et Matérielles

Notes de COURS - 2008/2009

Table des matières

Introduction

Nous allons étudier comment les `PROGAMMES` sont `exécutés` par un `PROCESSEUR`.

Du point de vue physique un processeur est un ensemble de circuits, un circuit étant un assemblage de transistors. Un transistor est un dispositif électronique fonctionnant comme un interrupteur (mais actif : il pompe de l'énergie sur l'alim du circuit, il a besoin d'être alimenté et il produit de l'énergie, suffisamment pour faire basculer l'interrupteur suivant ...)

Nous étudierons plusieurs types de processeurs, comme on étudie, parfois en informatique, des algorithmes. En TP nous travaillerons avec un processeur appelé `ARM`. On le trouve entre autres dans des téléphones portables, des consoles de jeu.

On ne va pas parler dans ce cours de la conception des circuits (cours ALM en L3). On va se préoccuper du fonctionnement logique d'un processeur.

Quand on parle de processeur, on parle de *circuit intégré* (sur un même support) de *traitement numérique de l'information* (les informations sont représentées par des nombres).

Rem : plusieurs types de représentation de l'information :

- représentation d'une tension électrique sur le cadran d'un voltmètre, info rep. par la position de l'aiguille : représentation analogique
- représentation de la tension par une valeur entière : représentation numérique, on dit aussi digitale.

Bibliographie :

- Architectures logicielles et matérielles, Amblard, Fernandez, Lagnier, Maraninchi, Sicard, Waille, Dunod 2000 et web (TIMA editions)
- Architecture des ordinateurs, Cazes, Delacroix, Dunod 2003

Chapitre 1

Codage des informations et représentation des naturels par des vecteurs binaires

1.1 Codage

Un codage est une correspondance un-à-un (bijection) entre un ensemble d'informations et un ensemble de naturels. Le plus souvent, pour N informations, on choisit l'intervalle [0, N-1]. Fonction de codage dans un sens, de décodage dans l'autre. La fonction de codage a un nom car on peut inventer plusieurs fonctions.

Exemples On donne ces codes sur papier aux étudiants plutôt que de les faire copier au tableau ! !

Ex1 : Codage des couleurs du Commodore 64

propriétés intéressantes : pas d'ordre apparent, pas de structure clair/foncé,..

B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$	
0	0 0 0 0	noir	5	0 1 0 1	vert	10	1 0 1 0	rose
1	0 0 0 1	blanc	6	0 1 1 0	bleu	11	1 0 1 1	gris foncé
2	0 0 1 0	rouge	7	0 1 1 1	jaune	12	1 1 0 0	gris moyen
3	0 0 1 1	cyan	8	1 0 0 0	orange	13	1 1 0 1	vert pâle
4	0 1 0 0	violet	9	1 0 0 1	brun	14	1 1 1 0	bleu pâle
						15	1 1 1 1	gris pâle

Code_C64 (violet) = 4 ; Decode_C64 (12) = gris moyen.

Ex2 : Codage des 16 couleurs sur les premiers PC couleurs

En regardant le code en base 2, on voit un bit de rouge (b2), un bit de vert (b1), un bit de bleu (b0) et un bit de clair (b3). Le codage du violet est fait de bleu et de rouge...

B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$		B	$b_3b_2b_1b_0$	
0	0 0 0 0	noir	5	0 1 0 1	violet	10	1 0 1 0	vert pâle
1	0 0 0 1	bleu	6	0 1 1 0	brun	11	1 0 1 1	cobalt
2	0 0 1 0	vert	7	0 1 1 1	gris	12	1 1 0 0	rose
3	0 0 1 1	cyan	8	1 0 0 0	noir pâle	13	1 1 0 1	mauve
4	0 1 0 0	rouge	9	1 0 0 1	bleu pâle	14	1 1 1 0	jaune
						15	1 1 1 1	blanc

Ex3 : L'ensemble des caractères affichables

code ASCII : "American Standard Code for Information Interchange"

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 .... chiffre nombre...

On obtient ce tableau par la commande Unix `man ascii`.

Propriétés : ordre, passage min/maj.

1.2 Codage d'ensemble structuré d'informations

Ensemble ordonné ↔ intervalle de naturels en respectant l'ordre (ex : lettres de l'aphabet).

Ensemble muni de **relations et d'opérations** : codage qui "respecte" les relations et opérations. (Exemple ordre alphabétique dans ASCII, majuscule + 32 = minuscule).

Ensemble **produit cartésien** : codage par des n_uplets (positions sur terre, lattitude, longitude ; ensemble des jours de l'année et jour de la semaine, numéro de semaine sur les tickets de stationnement).

Ensemble **"hiérarchisé"** d'éléments "regroupables" (critères à discuter) exemple villes olympiques. (généralisation : feuilles d'un arbre)

Ex4 : Code des villes olympiques

Soit l'ensemble suivant de villes Olympiques. E= {Albertville, Athènes, Atlanta, Chamonix, Grenoble, Los Angeles, Melbourne, Mexico, Montréal, Paris, Pékin, Rome, Sydney, Tokyo}

Ce sont les feuilles d'un arbre où la hiérarchie est donnée :

- par les continents (niveau 1) Amérique : 0 ; Asie : 1 ; Europe : 2.
- par les pays (niveau 2) en Amérique : Canada : 0 ; E.U. 1 ; Mexique : 2
- par les régions (niveau 3) en France : Ile de France : 0 ; Rhône-Alpes : 1

On code chaque continent, pays, région, ville par un entier, le code d'une ville est alors un quadruplet. Pour les objets de même niveau, on utilise l'ordre alphabétique.

continent	pays	région	ville	code quadruplet
Amérique	Canada		Montréal	(0, 0, 0, 0)
	Etats-Unis	Californie	Los Angeles	(0, 1, 0, 0)
		Géorgie	Atlanta	(0, 1, 1, 0)
	Mexique		Mexico	(0, 2, 0, 0)
Asie	Chine		Pékin	(1, 0, 0, 0)
	Japon		Tokyo	(1, 1, 0, 0)
Europe	France	Ile de France	Paris	(2, 0, 0, 0)
		Rhône-Alpes	Albertville	(2, 0, 1, 0)
			Chamonix	(2, 0, 1, 1)
			Grenoble	(2, 0, 1, 2)
	Grece		Athenes	(2, 1, 0, 0)
	Italie		Rome	(2, 2, 0, 0)
Océanie	Australie	New South Wales	Sydney	(3, 0, 0, 0)
		Victoria	Melbourne	(3, 0, 1, 0)

Si ce code vous parait "inventé", essayez donc de comprendre le code des Unités d'enseignement du DSU, il y a quelques ressemblances...

Codage d'instructions, de commandes, d'ordres,... On peut imaginer des codes par de tels N_iplets pour des ordres, des commandes à un robot, des instructions d'ordinateur,...

numéro du bras	direction	angle	vitesse de déplacement	...
3	Nord	135	haute	..
2	Sud	45	très faible	..

numéro de case mémoire	action	...
12345	remise à zéro	...
47	incrémentation	..

Correspondance entre n_uplet et naturel

Ex5 : code COUPLE3_4 {0, 1, 2, 3} × {0, 1, 2, 3, 4} est codé par {0, 1, 2, .., 19}
il y a 4*5=20 informations
COD_COUPLE3_4 ((0, 3)) = 0 x 5 + 3 = 3
DECOD_COUPLE3_4 (11) = (2, 1)
Compléter le tableau ci-après :

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

2 formules à savoir : (soit par coeur soit à savoir retrouver vite et sans erreur)
COD_COUPLE3_4 ((a, b)) = a x 5 + b
DECOD_COUPLE3_4 (n) = (n div 5, n reste 5)
La simplicité des deux dernières formules justifie le fait que l'on numérote à partir de 0 plutôt que de 1.

Où est le code ? Le code n'est pas dans l'information codée : 14 est le code du jaune dans le code des couleurs du PC (Ex2) et 14 est le code du couple (2,4) dans le code de l'Ex5.

Pour interpréter, comprendre une info codée il fautr connaitre le code. Le code de l'info ne donne rien, c'est le **système de traitement de l'info (logiciel ou matériel)** qui "connait" le code sinon il ne peut pas la traiter.

1.3 Représentation des naturels par des vecteurs binaires

Avec N chiffres en base b on peut représenter les b^N naturels de l'intervalle [0, b^N-1] en numération de position.
Sur X chiffres binaires on peut écrire les 2^X naturels de l'intervalle $[0, .., 2^X - 1]$

Ex : en base 10 avec 3 chiffres onn peut rep. les 10^3 naturels de l'intervalle $[0, 999]$.

Apprendre par coeur le tableau des puissances (arrondies !) de 2 et l'ordre de grandeur $2^{10} \approx 10^3$.

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

On s'intéresse au codage en base 2. Sur N positions, 2^N naturels, $[0, 2^N - 1]$. ex sur 4 positions.

Logarithme à base 2, nombre de bits L'opération réciproque de l'élévation à la puissance s'appelle le logarithme. Ici, le logarithme à base 2. Ainsi si $Y = 2^X$, on a $X = \log_2 Y$ Par exemple $\log_2 512 = 9$ Pour représenter Y naturels différents, il faut $\log_2 Y$ chiffres. (Attention aux arrondis à l'entier supérieur) Pour représenter les naturels de l'intervalle [0,. . .,9] il faut 4 bits, de même pour les naturels de l'intervalle [0,. . .,15]. Pour représenter les naturels de l'intervalle [0,. . .,255] il faut 8 bits. Pour coder les naturels de [0, 127], il faut 7 bits.

Comment trouve-t-on les chiffres binaires de l'écriture d'un naturel ? On parlera de chiffres des unités, DEUZAINES, QUATRAINES, huitaines, seizaines. (bien que ces mots ne soient pas tous dans le dictionnaire)
La méthode des restes successifs donnent les chiffres en commençant par celui des unités.
$169 = 84 \times 2 + \boxed{1}$ (chiffre des unités = 1)
$169 = (42 \times 2 + \boxed{0}) \times 2 + 1$ (chiffre suivant = 0)
$169 = ((21 \times 2 + 0) \times 2 + 0) \times 2 + 1$
$169 = (((10 \times 2 + 1) \times 2 + 0) \times 2 + 0) \times 2 + 1$
$169 = ((((5 \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 0) \times 2 + 1$
$169 = (((((2 \times 2 + 1) \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 0) \times 2 + 1$
$169 = ((((((1 \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 0) \times 2 + 1$

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

Utilisation des puissances de 2 :

```
169 = 128 + 42
    = 128 + 32 + 9
    = 128 + 32 + 8 + 1
    = 1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0
```

1.4 Ecriture en base 16 ou base hexadécimale

Les 16 chiffres sont 0, 1, .., 8, 9, A, B, C, D, E, F. Ils représentent respectivement les naturels qui s'écrivent 0, 1, .. 8, 9, 10, .., 15 en décimal. On apprend par coeur le tableau suivant qui donne pour les nombres de 0 à 15 leurs écritures en base 10, 2 et 16 :

0	0000	0	4	0100	4	8	1000	8	12	1100	C
1	0001	1	5	0101	5	9	1001	9	13	1101	D
2	0010	2	6	0110	6	10	1010	A	14	1110	E
3	0011	3	7	0111	7	11	1011	B	15	1111	F

Passage d'une base à une autre (en relation de puissance) Technique pour passer de la base 2 à la base 8 (ou octale) ou 16. Technique pour passer de la base 16 ou 8 à la base 2.

$169 = 10 \times 16 + 9$
$169_{10} = A9_{16}$
$169_{10} = A_{16} \times 16_{10} + 9_{16}$
$169_{10} = (((1\times2+0)\times2+1)\times2+0)\times2^4 + (((1\times2+0)\times2+0)\times2+1)$
$169_{10} = 1\times2^7+0\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0$
On retrouve $169_{10} = 10101001_2$
On obtient cette écriture en "remplaçant", dans l'écriture en base 16, A par 1010 et 9 par 1001.

base	x	y	z	t
16	4...9...D..	8...F...0..	0...0...8..	5...5...A..
2	010010011101	100011110000	000000001000	010101011010
8	2..2..3..5.	4..3..6..0.	0..0..1..0.	2..5..3..2.

1.5 Représentation des relatifs par des vecteurs binaires

1.5.1 une première idée : signe et valeur absolue

esquisser l'algo de l'addition : compliqué...

1.5.2 complément à deux

même algo pour add/sub naturels/relatifs. note : même algo = même circuit.

idée complément à 2 sur 8 bits : on peut représenter $2^8 = 256$ infos différentes ; la moitié pour les positifs, la moitié pour les négatifs. On code les relatifs de l'intervalle $[-128, +127]$, pas symétrique : le zéro... et surtout x+(-x)=0!!!

principe :
- $x \geq 0$ $x \in [0, +127]$ CodeC2(x)=x
- $x < 0$ $x \in [-128, -1]$ CodeC2(x)=x+256

Exemple : CodeC2(3)=3, CodeC2(127)=127, CodeC2(0)=0, CodeC2(-128)=128+256=128, CodeC2(-1)=-1+256=255

En binaire, ca donne

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	1000 0010
...		
12	12	0000 1100
...		
127	127	0111 1111

Résumé : CodeC2(x)=(x+256) mod 256

Décodage :

- $c < 128$ decodeC2(c)=c
- $c \geq 128$ decodeC2(c)=c-256

Exercice : établir le tableau des codes des relatifs de l'intervalle [-8,+7] et de leur code représenté en base 10 et en base 2.

Où est le code ? Soit le vecteur binaire 1001. Que représente-t-il ? Si je ne vous donne pas la règle de codage, vous ne savez pas répondre. Il code une couleur (bleu pale) ou un naturel sur 4 bits (9) ou un relatif sur 4 bits (-7).

1.6 Opérations

1.6.1 Addition

rappel + en base 10 : retenue, propagation, taille résultat.

Un processeur fait ses calculs sur une taille donnée : c'est physique, il a pas prévu la place...

Interprétation du résultat : Si naturels 5+2=7, 5+3=8, 7+9=16, sur 4 bits $n \in [0, +15]$, 16 pas représentable, retenue C

Si relatifs 5+2=7, 5+3=-8 ? ? ?, 7+(-7)=0, sur 4 bits $n \in [-8, +7]$, 5+3 donne un résultat qui n'a pas de sens, il n'est pas représentable, dans le processeur indicateur pour donner cette info, indicateur V

Chapitre 2

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

Dans tout notre travail : double vision :
• **vision abstraite** : des fonctions, des "entités", des modèles, des communications, des informations. Le modèle de définition de l'ordinateur a été élaboré entre 30 et 45 par Turing, Von Neumann, d'autres.
• **vision concrète** : où ça se passe, combien il y a de fils électriques, combien ça met de temps en secondes (ou en 10^{-9} secondes).

Les premiers ordinateurs ont été construits entre 40 et 45, principalement pour des besoins militaires (calcul de balistique, décryptage,..)

L'ordinateur (figure 2.1 exécute des programmes au moyen de 3 entités qui interagissent :
- processeur ;
- mémoire centrale ;
- système d'entrées-sorties (et liaison avec la mémoire secondaire) On parlera peu d'entrées-sorties en L2. C'est pour les grands de L3.

Le processeur exécute les programmes contenus dans la mémoire (centrale). Cette exécution a pour effet des actions vis à vis du monde extérieur via les périphériques. (disque, écran, imprimante..)

Les programmes traitent des REPRÉSENTATIONS des informations. Et les programmes SONT des informations. Dans les ordinateurs les informations sont représentées par des vecteurs binaires (vision abstraite), ce qui permet des connexions par des nappes de fils (vision physique) électriques au potentiel Haut ou Bas, selon l'instant.

2.1 Mémoire (sous-entendu centrale)

Les informations représentées par des vecteurs binaires peuvent être mémorisées. (Pour être consultées plusieurs fois)

2.1.1 Définition

Vision abstraite, logique

La mémoire contient des informations (parmi un certain domaine) codées (vecteurs binaires d'une certaine taille), en certaine quantité, pouvant (peut-être) être modifiées et désignables par une certaine "adresse".

```
mémoire abstraite

   --------
0  |      |
```

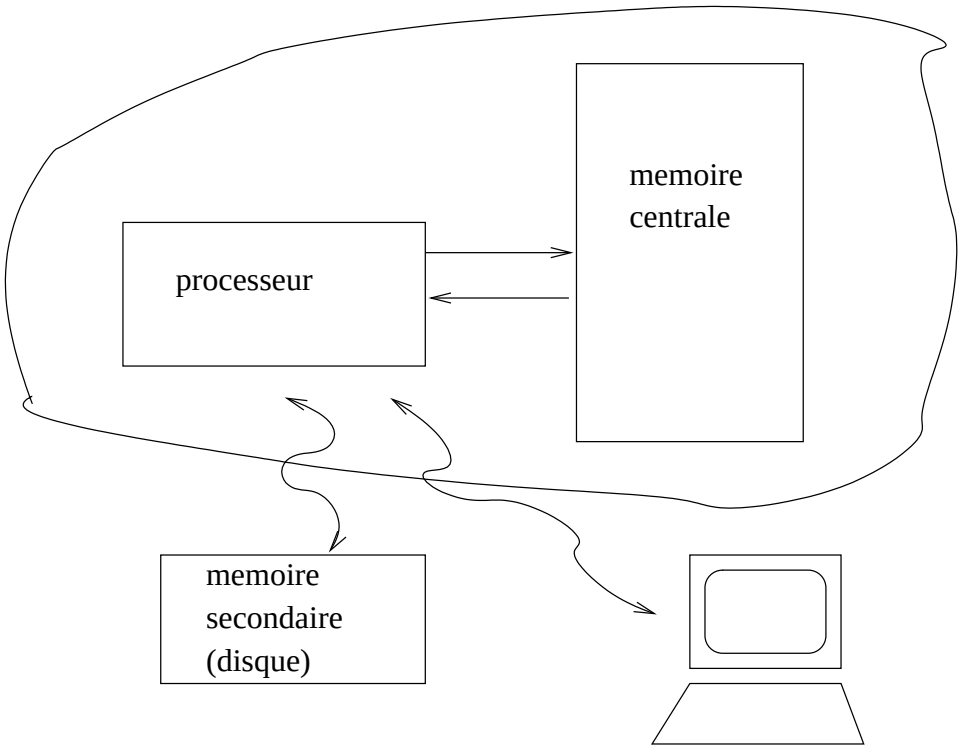


FIG. 2.1 – Processeur, mémoire et périphériques

```
      |      |
      |______|
      |______|
      |      |
      |      |
      |      |
      |      |
      |      |
      |      |
max-1 |______|
       taille
      <------>
```

Adresse = moyen de désigner une info dans la mémoire Adresse parmi [0, max-1], numéros d'emplacements. On parle d'emplacement précédent, suivant.

Une info en mémoire est représentée par un vecteur binaire d'une certaine `taille` (nombre de bits).

Nous verrons que l'on représente des infos :

– de taille 8 bits : octet (byte)

– de taille 16 bits : demi-mot (hword)

– de taille 32 bits : mot (word)

Si une info est à l'adresse X, l'octet suivant est à l'adresse X+1, le mot suivant est à l'adresse X+4. Les adresses sont exprimées en nombre d'octets.

Une adresse est aussi représentée sur un certain nombre de bits. Le numéro d'emplacement est codé par un vecteur binaire de (Log_2 max) bits. Par exemple 32 bits : cela permet de désigner 2^{32} octets différents.

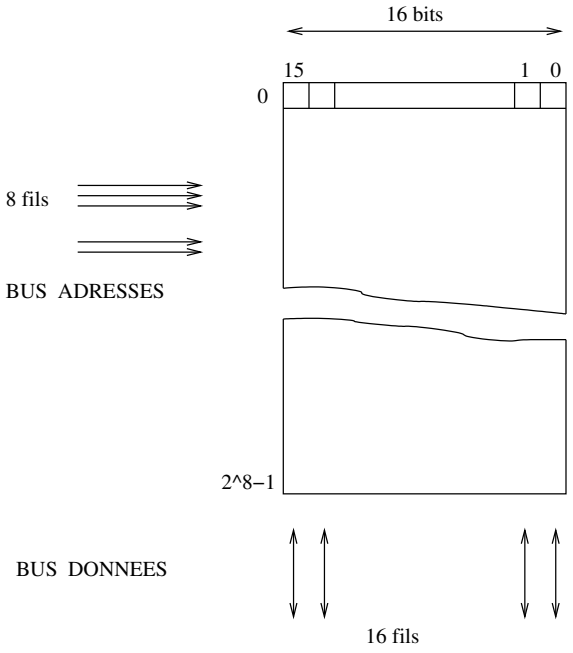


FIG. 2.2 – Mémoire physique d'info sur 16 bits avec addresses sur 8 bits

Note : remarquons que $2^10 = 1024 \approx 10^3$
D'où $2^{32} = 2^{10} \times 2^{10} \times 2^{10} \times 4 \approx 10^3 \times 10^3 \times 10^3 \times 4 = 4 \times 10^9$

Vocabulaire :
- 1 kilo (octet) = $10^3 \approx 2^{10}$
- 1 méga (octet) = $10^6 \approx 2^{20}$
- 1 giga (octet) = $10^9 \approx 2^{30}$

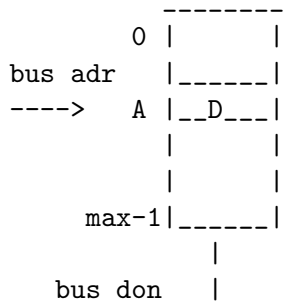
Vision physique

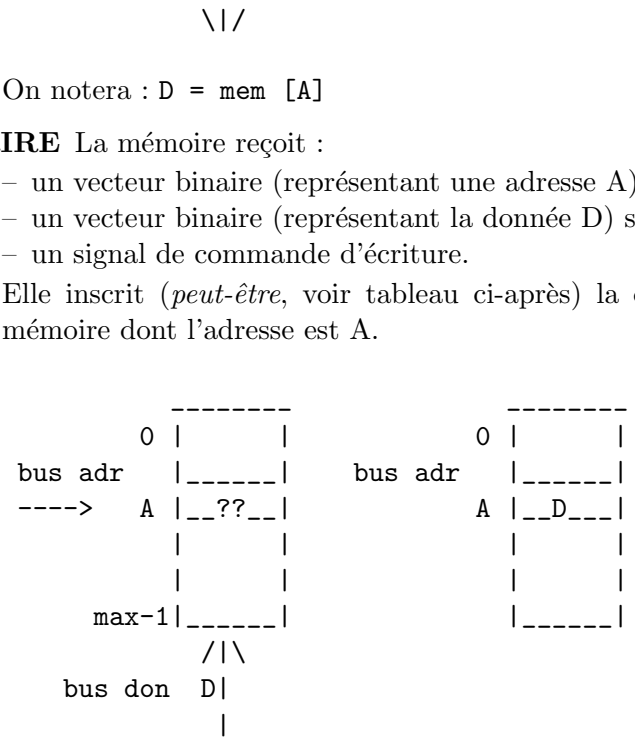
Toutes les infos sont représentées par des vecteurs binaires ce qui permet des connexions par des nappes de fils.

2.1.2 Actions sur la mémoire

LIRE La mémoire reçoit un vecteur binaire représentant une adresse (notons A) sur le bus adresses et un signal de commande de lecture.

Elle délivre un vecteur binaire représentant la donnée (notons D) sur le bus données. Il s'agit du contenu de l'emplacement mémoire dont l'adresse est A.





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.

Elle inscrit (*peut-être*, voir tableau ci-après) la donnée D comme contenu de l'emplacement mémoire dont l'adresse est A.

NE RIEN FAIRE : ni lecture, ni écriture. La mémoire garde indéfiniment (*peut-être*, voir tableau ci-après) les informations.

La lecture et l'écriture ne peuvent pas être simultanées. On dit que le bus données est "bidirectionnel" mais c'est dans le temps. A un instant donné ce n'est pas vrai.

2.1.3 Différents types ; vision très concrète

nom du type de mémoire	propriété de sauvegarde si coupure de courant	propriété de modifiabilité selon type d'utilisateur
ROM mémoire morte	garde l'info	inscrite à la fabrication usine de fabrication semi-conducteurs
Eprom (+ variantes)	garde l'info	modifiable (100 fois) atelier PMIPME électronique
mémoire "flash"	garde l'info	modifiable 100 000 fois utilisateur dans appareil carte bancaire, téléphone, vitale
RAM mémoire vive	perd l'info si coupure de courant	modifiable autant de fois que le signal écriture sera activé
mémoire cache	rien à voir	dans cette classification

2.2 Entrées sorties

Le processeur peut accéder à divers organes physiques, clavier, disque, réseau, écran, d'interaction avec l'utilisateur, de communication ou de stockage. Des informations peuvent être échangées (dans un ou deux sens) entre ces organes et la mémoire.

2.3 Processeur

2.3.1 Actions, processeur/mémoire

Le processeur est un circuit relié à la mémoire par les bus adresses et données.

La mémoire contient des informations de nature différentes :

- des données : représentation binaire d'une couleur, d'un entier, d'une date, etc.
- des instructions : spécification écrites en binaire d'une ou plusieurs actions que le processeur peut *exécuter*; par exemple, ajouter le contenu de deux mots mémoire et ranger le résultat à une adresse précise.

Le processeur, lié 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.

2.3.2 Composition du processeur

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

des registres cases de mémoire interne désignées par des numéros ou des noms particuliers. A la différence des emplacements mémoire (centrale), peuvent être lus et écrits simultanément (et même, peuvent être écrits avec une valeur qui est fonction de la valeur qui y était précédemment). L'action REG $<-$ REG $+ 1$ est atomique sur les registres.

des unités de calcul élaborent un vecteur binaire à partir d'un (ou plusieurs) vecteur(s) binaire(s) présenté(s) en entrée (représentant une "valeur" et, peut-être, une commande).

exemple : additionneur 8 bits. dessin 2 nappes de 8 fils en entrées, 1 nappe de 8 fils en sortie, + des sorties donnat des infos sir la correction du résultat.

une unité de contrôle elle aussi contient registres et unités de calcul. Mais ses "résultats" (sorties) sont des ordres, ou successions d'ordres, émis vers les registres, unités de calcul, et mémoire. Son rôle est d'enchaîner des actions de base, par ex faire faire 2 additions de suite. L'**exécution** des instructions consiste précisemment à émettre de telles successions d'ordres.

un compteur ordinal ou compteur programme (PC) ou pointeur d'instructions Ce registre particulier repère l'adresse en mémoire de l'instruction en cours d'exécution.

2.3.3 Notion d'exécution séquentielle des instructions

Pour *exécuter* une suite d'instructions (on appelle une telle suite un programme) le processeur effectue de façon cyclique :

- lire une instruction à l'adresse contenue dans PC
- exécuter cette instruction
- calculer l'adresse de l'instruction suivante et mettre à jour PC

Dans un ordinateur cette boucle démarre par un signal physique (RESET) qui a pour effet de forcer une valeur initiale dans PC et ne s'arrête jamais (sauf coupure de courant...). Il peut dans les machines modernes être suspendu pour économiser de l'énergie (machines portables, embarquées,..)

Normalement les instructions sont exécutées dans l'ordre où elles sont écrites en mémoire. Certaines instructions spécifiques donnent l'adresse de l'instruction suivante (rupture de séquence). On y reviendra.

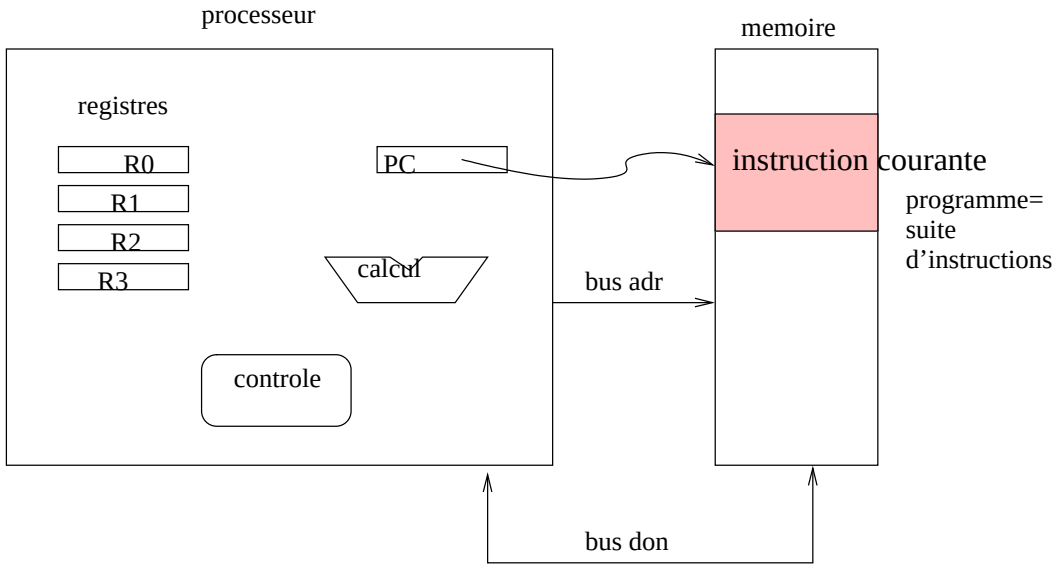


FIG. 2.3 – Processeur relié à de la mémoire

2.3.4 Notion élémentaire de langage machine : première définition

La nature de l'instruction c'est-à-dire des actions à réaliser par le processeur est CODÉE dans un (ou plusieurs) mots de l'instruction.

Le codage des instructions par des vecteurs binaires constitue le *langage machine du processeur*.

On parle :

- D'ARCHITECTURE LOGICIELLE pour décrire l'organisation du langage machine, ISA=Instruction Set Architecture
- D'ARCHITECTURE MATÉRIELLE pour décrire la façon dont il est exécuté par du matériel.

2.3.5 C'est quoi les processeurs ?

Sur les PC : des pentium, ou des compatibles. Sur les Macintosh : 68000 (très anciens) ou PowerPC (anciens) ou Pentium (récents), Sur certaines consoles de jeux : des ARM (le même qu'on étudiera en TP)

Chapitre 3

Langage d'assemblage, langage machine

3.1 Vie d'un programme

timing : Ce chapitre prend un peu plus d'une séance d'1h30. Le suivant par contre (rupture de séq et sous-programme) est un peu plus court.

Etapes de compilation

1. `monprog.c` contient des instructions du langage C + données par exemple les chaînes de caractères (cf tp avec la chaîne charlot)
2. `gcc -S monprog.c` ou `arm-elf-gcc -S monprog.c` produit `monprog.s` qui contient des instructions du lg d'assemblage + données : on va y revenir ; lisible par un éditeur de texte (mais non exécutable)
3. `gcc monprog.s -o monprog` produit `monprog` qui contient des instructions en langage machine + données ; non lisible mais exécutable. On dit que le fichier 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. remarque : il y a une étape supplémentaire, la production du fichier `monprog.o` qui contient du binaire dit translatable, non directement exécutable ; on y reviendra dans un chapitre ultérieur.

Le binaire exécutable contient des instructions codées en binaire et pouvant être traitées, exécutées par un processeur.

Comment exécuter monprog ? Plusieurs classes d'ordinateurs.

- On est sur un ordinateur (PC, station Solaris) dont le binaire est connu du processeur (ex on a produit du "binaire Intel", on est sur un PC dont la carte mère est équipée d'un processeur Intel, avec une commande comme `gcc monprog.c`). On tape `monprog` ou on "double clique" dessus. Une partie du Syst.Exploit. copie le contenu du fichier exécutable du disque dur vers la mémoire centrale, puis met l'adresse de début dans le registre PC et ensuite le matériel (le processeur) fait son boulot...
  C'est le Syst.Exploit. qui choisit l'emplacement en mémoire centrale et NOUS ne le connaissons pas (on va négliger l'aspect mémoire virtuelle...)
  Connaissance pas utile de l'architecture matérielle fine. Le programme sera mis n'importe où, et çà marche quand même.
- On est sur un ordinateur qui ne connait pas le binaire que l'on produit (ex. `arm-elf-gcc monprog.c` produit du binaire arm et on est sur un Intel). On utilise un simulateur, c'est-à-dire un programme qui fait semblant d'exécuter ce binaire (`arm-elf-run monprog`). Le simulateur réalise des actions équivalentes aux instructions du programme dans une mémoire à lui.

– Cas des automatismes (lecteur de carte électronique, etc.) Quand on écrit le programme on connait précisément l'architecture et on peut produire directement une image du programme dans la mémoire à une adresse fixe et ranger le binaire dans une EPROM (une mémoire qui garde l'info quand on coupe le courant).

3.2 Langage machine

Pour tout processeur il existe UN langage machine caractéristique. (compatibilité entre processeurs dérivés, famille,..)

Comme tout langage (informatique ou non) le LM a

– un lexique : les mots que l'on peut écrire
– une syntaxe : l'organisation des mots qui est permise
– une sémantique : le sens des phrases, cad des succession de mots

Un peu plus en "détail" :

– **un lexique et une morphologie** : la liste des mots du langage (les instructions), la façon dont ils s'écrivent (leur codage) , éventuellement les variantes, les groupements d'instructions d'une même famille, etc Pour le LM les mots sont des suites de 0 et de 1. On remarque au passage que si votre numéro de sécurité sociale qui traine qqpart en mémoire est AUSSI le code d'une certaine instruction, si le compteur ordinal du processeur, à la suite d'une erreur, se met à "pointer" là, il se peut que votre numéro de sécu soit "exécuté". sinon... voir ci-après, cas de fautes.
– **une syntaxe** quels sont les groupements valides d'instructions en programmes ? C'est, en général, très souple : on peut écrire ce qu'on veut... Pour le LM, la syntaxe dit que les suites de 0 et de 1 sont organisées par champs pour former des instructions.
– **une sémantique (signification)** Un programme est une suite (ou séquence, liste) d'instructions, il y a un ORDRE. Les instructions sont exécutées dans l'ordre où elles sont écrites. Vu de l'extérieur si on considère un processeur dans lequel chaque instruction est codée sur 32 bits, un programme est une séquence de mots de 32 bits. Un mot de 32 bits représente 1 instruction c'est-à-dire un ordre à faire exécuter par le processeur. Par exemple, l'instruction 36842 précise qu'il faut ajouter le nombre actuellement dans le registre 17 avec celui qui est dans le registre AB et ranger le résultat en mémoire à l'adresse 27.
– **un dictionnaire des fautes !** Si il y a 237 instructions dans le langage machine d'un certain processeur, elles peuvent être codées par un vecteur binaire de 8 bits ($2^7 = 128$; $2^8 = 256$) et donc il y a 19 vecteurs invalides. Que doit alors faire le processeur s'il rencontre un code invalide ?

3.3 Instructions et programmes en langage machine

Nous sommes débutants : nous nous plaçons dans l'hypothèse du cas où en écrivant l programme, on sait à quelle adresse le programme sera rangé en mémoire pour être exécuté (instructions et données) **mais attention, en TP avec le processeur ARM et le simulateur utilisé, ça ne sera pas le cas**.

Par ailleurs nos instructions, nous les écrivons en français, plus lisible que le binaire. les données, pareil, nous écrivons des caractères ASCII, des nombres,..

2 types principaux : instructions de calcul (ou d'échanges de place) entre des informations mémorisées et instructions de rupture de séquence.

3.3.1 Instructions de calcul (ou d'échanges de place) entre des informations mémorisées.

L'instruction désigne alors la *source* et le *destinataire*. Les "sources" sont des éléments de mémorisations : cases mémoires ou registres du processeur. Le destinataire sera pareil.

L'instruction désigne destinataire, source1, source2, opération. on a les combinaisons suivantes

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

mém signifie que l'instruction fait référence à une case dans la mémoire

reg signifie que l'instr fait référence à un registre (par son nom ou son numéro)

valIMM signifie que l'information source est contenue dans l'instruction du langage machine.

Exemples de signification d'instructions, peu importe le codage :

- reg12 ← reg14 + reg1
- registre4 ← le mot mémoire d'adresse 36000 PLUS le registre A
- reg5 ← reg5 - 1
- le mot mémoire d'adresse 564 ← registre7 (opération : rien du tout)
- CONVENTIONS de NOMS : move, load, store

3.3.2 Instructions de rupture de séquence

On y reviendra en détail.

Fonctionnement standard : une instruction est écrite à l'adresse X ; l'instruction suivante (dans le temps) est l'instruction écrite à l'adresse X+(taille de l'instruction). C'est implicite pour toutes les instructions de calcul.

Une instruction de *rupture de séquence* peut désigner la prochaine instruction à exécuter (à une autre adresse).

Exemples :

- L'instr. suivante est celle d'adresse 5012. La suivante de l'instruction II peut même être II, mais alors le processeur part en boucle non contrôlée. Il faut utiliser Reset.
- L'instr. suivante est celle rangée à l'adresse "mon adresse" MOINS 40 (ou "mon adresse" PLUS 10). Note : cela peut servir à "faire" des boucles, répéter le même paquet d'instructions plusieurs fois (dessins...)
- Si le résultat du calcul précédent est ZERO, alors la prochaine instr. à exécuter est celle d'adresse "mon adresse+10", sinon la prochaine instr. à exécuter est la suivante dans l'ordre d'écriture, cad à l'adresse "mon adresse"+1. Utilisation : codage de choix.

3.3.3 Remarques :

1. une instr. du LM est une chaine de bits. Pour en parler on a écrit en "francais". On aurait besoin d'une correspondance : instr. en francais ⇜⇝ chaine de bits (LM écrit en binaire)
2. lorsque l'on considère une chaine de bits (pax ex 1 mot de la mémoire) on ne sait pas si elle représente une donnée (un naturel, un relatif, une couleur, un caractère, ...) ou une instruction. Le code n'est pas dans l'info codée !!!
3. on a vu plusieurs instructions qui tiennent compte de la place réelle en mémoire des données ou instructions. Quand on ne connaît pas ces emplacements (les adresses), que faire ?

3.4 Langage d'assemblage

On introduit un niveau de langage plus facile à manipuler, à lire :
- langage textuel (au lieu de binaire) mais correspondance 1 instruction Lg Ass ↔ 1 instruction Lg machine
- claire séparation données, instructions
- on ne connait pas les adresses réelles, on ne peut donc pas en tenir compte.

3.4.1 Langage textuel, notation des instructions

1 instruction = mot conventionnel (mnémonique) + désignation des objets (instr ou données) manipulés. A chaque notation est associée une convention d'interprétation).

Exemples :
- machine1 (ARM) : `add r4, r5, r6` signifie r4 ← r5 + r6. `r4` désigne le contenu du registre, on parle bien sur du *contenu* des registres, on n'ajoute pas des ressources physiques
- machine 2 (SPARC) : `add %g4, %g5, %g6` signifie g6 ← g4 + g5
- machine 3 : `addA 5000` signifie regA ← regA + Mem[5000]
- machine 3 : `addA ♯50` signifie regA ← regA + 50
- machine 4 : `add r3, r3, [5000]` signifie reg3 ← reg3 + Mem[5000]

Remarque : Il n'y a pas de règle (notation) générale, cela dépend des fabricants. Il y a juste qq habitudes concernant les mnémoniques (add, sub, load, store, jump, branch, clear, inc, dec) ou la notation des opérandes (`#`, `[xxx]`) mais rien de général...

3.4.2 Désignation des objets

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 `-5`, la valeur rangée dans la mémoire à l'adresse 0xff, etc.

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.

Définitions (à connaître :)

désignation par registre L'objet désigné, une donnée, est le contenu d'un registre. L'instruction contient le nom ou le numéro du registre.
- 6502 : 2 registres A et X (entre autres) `TAX` = transfert de X dans A = contenu de X ← contenu de A (on écrira X $leftarrow$ A). Désignations registre, registre.
- ARM : `mov r4 , r5` ; signifie r4 ← r5. Désignations registre, registre.

désignation immédiate La donnée dont on parle est "carrément" incluse dans l'instruction (tjrs des données)
- ARM : `mov r4 , #5` ; signifie r4 ← 5. Désignations registre, immédiate.

désignation directe ou absolue L'objet désigné est dans une case mémoire dont on donne l'adresse dans l'instruction. L'objet désigné peut être une instr ou une donnée.
- machine 1 : `store R4, [5000]` = Mem[5000] ← R4. Désignations registre, directe ou absolue. le deuxième opérande (ici une donnée) est désigné par son adresse en mémoire.
- machine 2 : jump 0x2000 : l'instruction suivante (qui est l'instruction que l'on veut désigner) est celle d'adresse 0x2000.

désignation indirecte par registre L'objet désigné est dans une case mémoire dont l'adresse est dans un registre précisé dans l'instruction.
- ARM : `add r3, [r5]` = r3 ← r3 + le mot mémoire dont l'adresse est contenue dans le registre 5 ; on note souvent r3 ← r3 + mem[r5]. Dessin... Désignations registre, indirect par registre

désignation indirecte 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.
- ARM : `add r3, [r5, #4]` = r3 ← r3 + mem[r5 + 4]. Dessin... la notation `[r5 + #4]` désigne le mot mémoire (une donnée ici) d'adresse `r5 + #4]`.
- machine 3 `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. Quand le registre est PC on parle de désignation relative au compteur de programme PC.

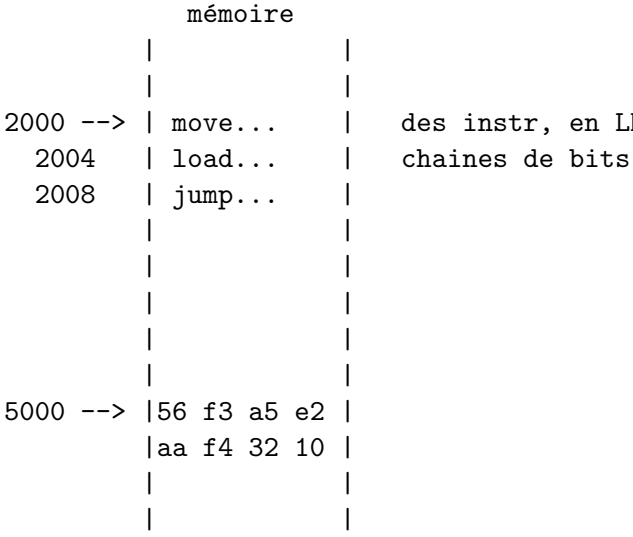
3.4.3 Séparation données instructions

Le texte du programme est organisé en zones :
- zone TEXT : code, programme, instructions, constantes
- zone DATA : données initialisées
- zone BSS : données non initialisées, reservation de place en mémoire

Parfois on peut préciser où chaque zone doit être placée en mémoire : directive ORG pour donner l'adresse de début de la zone.

On prend cette hypothèse pour l'exemple suivant, dans lequel on suppose que les instructions sont toutes codées sur 4 octets :

```
zone TEXT, org 0x2000
move r4, #5004
load r5, [r4]
jump 0x2000
.......
zone DATA,  org 0x5000
entier sur 4 octets : 0x56F3A5E2
entier sur 4 octets : 0xAAF43210
....
```



effet de ce programme : mettre 0xAAF43210 dans r5 et recommencer indéfiniment. Ce qui est très utile..

3.4.4 Etiquettes

2 raisons :
- si on ne sait pas l'emplacement de chargement des programmes (la directive ORG ne peut pas toujorus être utilisée)
- si on veut se faciliter la vie

Définition : Etiquette = nom choisi librement (il y a des règles lexicales quand même) qui désigne une case mémoire. cette case peut contenir une donnée ou une instruction, D'où une étiquette correspond à une adresse.

Exemple :

```
zone TEXT
DD: move r4, #5004
    load r5, [YY]
 jump DD

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

Avantages :
- Le programmeur n'a pas besoin de connaitre ces VRAIES adresses pour comprendre ce que fait son programme...
- Le programme est plus facile à lire, à écrire

Quand la correspondance étiquette/adresse est-elle faite ?
- Le remplacement de DD par 2000 et de YY par 5004 est fait au moment de l'assemblage sur les machines où l'adresse de chargement est connue au moment de l'assemblage (si les adresses de chargement des zones TEXT et DATA sont respectivement 2000 et 5000). De plus en plus rare. L'adresse de chargement peut être connue de l'outil ou passée en paramètre lors de l'appel de l'assemblage.
- la correspondance peut aussi être faite lors de la phase d'implantation en mémoire (on parle de chargement) si l'adresse de chargement N'est PAS connue au moment de l'assemblage.

Chapitre 4

Instructions de rupture de séquence et Programmation des structures de contrôle

4.1 Exécution séquentielle et ruptures de séquences

On se souvient du rôle du compteur programme (PC).

4.1.1 Exécution séquentielle et ruptures de séquences

Il existe un registre spécial du processeur : le compteur programme PC (ou pointeur d'instructions IP).

A chaque instruction le proc émet une copie du contenu de PC sur le bus adresses, récupère l'instruction fournie sur le bus données. Il l'exécute (on verra comment plus tard). Il met à jour le PC pour préparer l'accès à l'instruction suivante. Par défaut en l'incrémentant.

Et ça recommence...

Ainsi les instructions écrites dans la mémoire sont exécutées dans l'ordre d'écriture. SAUF ... instructions de rupture de séquence.

Regardons les 8 cas de séquencement :

1. A l'**initialisation**, PC est forcé à une valeur pré-établie (0)
2. **Séquencement normal** Après chaque instruction le PC est incrémenté. De 1 si l'instruction est codée sur UN seul mot, parfois sur plusieurs. Ca dépend des machines et des instructions et de la taille des mots.
   - REMARQUE sur ARM, qui est une machine où toutes les instructions ont la même longueur, le compteur programme progresse d'instructions en instructions. Comme les adresses sont des adresses d'octets et que les instructions sont sur 4 octets, le PC progresse de 4 en 4
   - à contrario sur certaines machines où les instructions sont de longueur variable, par exemple, 1, 2, ou 3 octets, le PC donne successivement les adresses des différents octets de l'instruction. On y reviendra quand on étudiera l'exécution détaillée.
3. **Rupture inconditionnelle :** L'exécution d'une instruction consiste à forcer une nouvelle valeur dans le PC.

   Cas TRES particulier : certains des premiers RISC (dont le Sparc, le mips, ...) exécutent quand même l'instruction qui suit le saut. C'est bizarre, c'est comme ça... Il y a des raisons,
4. **Rupture conditionnelle :** si la condition [1] est vérifiée, le PC est modifié, sinon il progresse de 1 comme normalement. (ou de 4...)

[1] Condition en général interne au processeur, parfois externe

5. **Appels retour de sous programmes :** On va y revenir tout de suite.
6. **Interruptions** Un mécanisme essentiel, mais pour les grands.. Si un certain signal PHYSIQUE externe est actif, PC prend une valeur lue en mémoire à une adresse conventionnelle.
7. **Machine superscalaire** Exécute plusieurs instructions "en même temps" cours d'architecture avancée
8. **Instructions de séquencement exotiques** cours d'architecture avancée.

Désignation de l'instruction suivante

- Désignation `directe` : l'adresse de l'instruction suivante est donnée dans l'instruction. Exemple dans le processeur utilisé pour le TD automate il y a des JUMP qui sont comme ça
- Désignation `relative` : l'adresse de l'instruction suivante est obtenue en ajoutant un certain `déplacement` (peut être signé) au compteur Programme.

Remarque : On est souvent dérouté par le fait que le déplacement est ajouté par rapport à l'adresse de l'instruction qui suit la rupture. On verra pourquoi en étudiant en détail le véritable séquencement (Chapitre 7).

C'est le cas sur ARM.

Exemple : machine à instructions sur 4 octets

etiquette	adresse correspondant a l'etiquette	instruction ASS	codage effectif de l'instruction
	0x1028	CLR Reg6	0x********
e1 :	0x102C	BRANCH si cond e2	0x**** +0x0C
	0x1030	xxx	xxx
	0x1034	xxx	xxx
	0x1038	xxx	xxx
e2 :	0x103C	yyy	0x********
	0x1040	xxx	xxx
	0x1044	xxx	xxx

(0x102C + 4 + 0x0C = 0x103C)

Utilisation typique des sauts ou branchements

On se place du point de vue de la programmation en langage d'assemblage, et de façon SYSTÉMATIQUE.

On va considérer la programmation de 3 types de construction algorithmiques :

- programmation systématique des structures de contrôle de la programmation classique (type if then else)
- programmation systématique des solutions modélisées par des automates avec actions (automates étendus de INF232) (on dira aussi machines séquentielles avec actions)
- programmation systématique des sous-programmes (procédures et fonctions).

On commence ici par les structures de contrôle classiques de la programmation impérative ?

- Action0 ; if (cond) then (Action1) ; Action2
- Action0 ; if (cond) then (Action1) else Action2 ; Action3
- Action0 ; while (cond) (Action1) ; Action2 ; Action3
- Action0 ; repeat (Action1) until (cond) ; Action2 ; Action3
- Action0 ; for (i=0 ; i<N ; i++) Action2 ; Action3 ; ; Action4
- Action0 ; for (i=N ; i>=0 ; i– Action2 ; Action3 ; ; Action4

– Action0 ; if (cond1 ET cond2) then (Action1) ; Action2
– Action0 ; if (cond1 ETpuis cond2) then (Action1) ; Action2
– Action0 ; if (cond1 OU cond2) then (Action1) ; Action2
– Action0 ; if (cond1 OUalors cond2) then (Action1) ; Action2

On dispose seulement des sauts et des sauts conditionnels : `branch etiq` et `branch si cond etiq`.

4.2 Conditionnelles

4.2.1 Version Paul

`Action0 ; si cond alors {Action1 ; Action2} ; Action3`

etiquette	adresse correspondant a l'etiquette	instruction ASS	codage de l'instruction
	0x1020	fin de Action0...	0x********
	0x1024	calcul...	0x********
	0x1028	Evaluer cond dans FLAGS	0x********
	0x102C	BRANCH si cond debutdeact1	0x****
	0x1030	JMP debutdeaction3	cible : 0x106C
debutdeact1 :	0x1034	premiere instruction de Action1	0x********
	0x1040	xxx	xxx
	0x1044	xxx	xxx
findeact1 :	0x1048	derniere instruction de Action1	...
debutdeact2 :	0x1050	premiere instruction de Action2	..
	...	..... il pourrait y en avoir d'autres	...
findeact2 :	0x1054	derniere instruction de Action2	...
debutdeact3 :	0x106C	premiere instruction de Action3	...
	0x1070	suite de Action3	
findeact2 :			

`Action0 ; si cond alors Action1 sinon Action2 ; Action3`

etiquette	adresse correspondant a l'etiquette	instruction ASS	codage de l'instruction
	0x1020	fin de Action0...	0x********
	0x1024	calcul...	0x********
	0x1028	Evaluer cond dans FLAGS	0x********
	0x102C	BRANCH si cond debutdeact1	0x****
	0x1030	premiere instruction de Action2	..
	...	..... il pourrait y en avoir d'autres	...
	0x1034	derniere instruction de Action2	...
	0x1038	JMP debutdeact3	0x...0x104C
debutdeact1 :	0x103C	premiere instruction de Action1	0x********
	0x1040	xxx	xxx
	0x1044	xxx	xxx
findeact1 :	0x1048	derniere instruction de Action1	...
debutdeact3 :	0x104C	premiere instruction de Action3	...
	0x1050	suite de Action3	

`VERSION 2 Action0 ; si cond alors Action1 sinon Action2 ; Action3`
(En effet les conditions sur les codes de conditions arithmétiques vont souvent 2 par 2. L'une porte sur une condition et l'autre sur la condition complémentaire. cf doc du processeur quelqu'il soit.

etiquette	adresse correspondant a l'etiquette	instruction ASS	codage de l'instruction
	0x1020	fin de Action0...	0x********
	0x1024	calcul...	0x********
	0x1028	Evaluer cond dans FLAGS	0x********
	0x102C	Branch si NON cond debutdeact2	0x****
	0x1030	premiere instruction de Action1	..
	...	..... il pourrait y en avoir d'autres	...
	0x1034	derniere instruction de Action1	...
	0x1038	JMP debutdeact3	0x...0x104C
debutdeact2 :	0x103C	premiere instruction de Action2	0x********
	0x1040	xxx	xxx
	0x1044	xxx	xxx
findeact2 :	0x1048	derniere instruction de Action2	...
debutdeact3 :	0x104C	premiere instruction de Action3	...
	0x1050	suite de Action3	

4.2.2 Version Fabienne

On note `Ik` une instruction ; pour fixer les idées on peut penser que cela peut être une affectation mais cela peut être n'importe quelle instruction complexe ou séquence d'instructions.

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

Par exemple `ExpCond est A=B` avec A et B deux entiers rangés respectivement dans les registres R1 et R2.

```
            I1
            calcul de A-B + positionnement de ZNCV
            branch si non égal à 0 = non(Z) à etiq_suite
            I2
            I3
            I4
etiq_suite: I5
```

Avec des adresses et des instructions de machines existantes codées sur 32 bits.

```
0x1020 I1               I1
0x1024 CMP R1, R2       CMP R1, R2
0x1028 JUMP_ABS_1038    BRANCH_RELATIF+12octets
0x102c I2               I2
0x1030 I3               I3
0x1034 I4               I4
0x1038 I5               I5
```

Note : les 12 octets sont à moduler selon la valeur de PC au moment de l'exécution du branchement.

Autre codage :

```
            I1
            calcul de A-B + positionnement de ZNCV
            branch si égal à 0 = Z à etiq_alors
            branch à etiq_suite
etiq_alors: I2
            I3
            I4
etiq_suite: I5
```

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

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

Autre solution :

```
            I1
            évaluer ExpCond + ZNCV
            branch si vrai à etiq_alors
            I4
            I5
            I6
            branch inconditionnel etiq_finsi
etiq_alors: I2
            I3
etiq_finsi: I7
```

Quand et comment l'adresse cible du branchement ou le déplacement correspondant sont-ils calculés ? Par le traducteur (assembleur) lors du passage `prog.s` à `prog.o`. Soit il connait l'adresse de début et il associe une adresse à chaque instruction, soit il ne la connait pas et il calcule une différence (il connait la taille nécessaire au codage de chaque instruction en langage machine).

4.3 boucles

4.3.1 `I1 ; while ExpCond do {I2 ; I3} ; I4`

```
       I1
debut: evaluer ExpCond
       branch si faux fintq
       I2
```

```
       I3
       branch toujours debut
fintq: I4
```

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

4.3.2 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 tantque avec test à la fin.

4.3.3 I1 ; for (i=0 ; i<N ; i++) {I2 ; I3 ;I4} ; I5

Traduire en tantque puis traduire en ARM (en exo...)

```
i=0                    mov R0, #0
tantque i<N       tq:  cmp R0, R1    @ N dans R1
  I2                   bge ftq
  I3                      I3
  I4                      I3
  i=i+1                   I4
I5                        add R0, R0, #1
                       bal tq
                  ftq: I5
```

Exercice : deux boucles for imbriquées

```
for (i=0; i<N; i++)
   for (j=0; i<K; j++)
        I2;I3
```

4.4 Conditions non élémentaires

4.4.1 si C1 ou C2 ou C3 alors I1 ;I2 sinon I3

Solution avec évaluation minimale des conditions, dite `oualors` en algorithmique.

```
           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 toujours etiq_fin
etiq_sinon: I3
etiq_fin:
```

ou

```
            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 toujours etiq_fin
etiq_alors: I1
            I2
etiq_fin:
```

Si on veut tout évaluer : évaluer chaque Ci dans un registre et utiliser l'instruction OR du processeur.

4.4.2 `si C1 et C2 et C3 alors I1 ;I2 sinon I3`

Solution avec évaluation minimale des conditions, dite `etpuis` en algorithmique.

```
            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 toujours etiq_fin
etiq_sinon: I3
etiq_fin:
```

4.5 Codage d'un choix

```
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  (le test a>b est inutile...)
```

Spécifique ARM : instructions ARM conditionnelle. Dans le codage d'une instruction il est prévu un champ condition (bits 31 à 28) contenant un des mnémoniques des conditions. La sémantique d'une instruction est est la suivante : si la condition est vraie exécuter l'instruction sinon passer à l'instruction suivante (dans l"ordre d'écriture).

```
selon a,b:           a dans R0, b dans R1, x dans R2
  a<b : x=x+5           cmp R0,R1
  a=b : x=x+1           addlt R2, R2, #5
  a>b : x=x+9           addeq R2, R2, #1
                        addlt R2, R2, #9
```

Ces instructions conditionnelles peuvent aussi être utilisées pour supprimer les branchements dans des if then else à corps-alors et corps-sinon de petite taille.

Réfléchir : il existe `addeqs`...

4.6 Exercice de synthèse : nombre de 1

```
x, nb : des entiers >= 0
nb=0
tantque x<>0
  si x mod 2 <> = alors nb=nb+1
  x = x div 2
```

```
    .text
    .global main

@ nb : r0
@ x : r1
main:
    mov r1, #0xE2

    mov r0, #0
tq: cmp r1, #0
    beq ftq
      tst r1, #1
      beq suite
        add r0, r0, #1
suite:
    mov r1, r1, lsr #1
    bal tq
ftq:

    @ ecrire le contenu de r0
    mov r1, r0
    bl EcrNdecimal8

fin: bal exit
```

Chapitre 5

Programmation à partir des automates reconnaisseurs

5.1 Automate avec actions à nombre fini d'états : Définition commentée

Un automate avec actions à nombre fini d'états (on dit aussi un automate d'état fini étendu) est la donnée de :

- Un ensemble E (non vide, fini) nommé le **vocabulaire d'entrée** dont les éléments sont les **symboles** d'entrées. On appelle parfois E le "vocabulaire" terminal V_T. Les entrées peuvent aussi être des conditions portant sur les valeurs de variables.
- Un ensemble S (non vide, fini) dont les éléments sont appelés les **états**.
- Un élément particulier s_0 de S qui est identifié comme l'état initial.
- Un ensemble O (non vide, fini) appelé vocabulaire de sortie. Les éléments sont des actions sur les variables considérées (affectation, incrémentation, etc).
- Une fonction f dite **fonction de transition** : f : E X S $\rightarrow$ S
- Une fonction g dite **fonction de sortie** : g : S $\rightarrow$ O. Cette sorte d'automate est appelé automate de Moore.

Si f (e_i, s_j) = s_k on dit que l'état s_k est le **successeur, ou état suivant** de s_j pour l'entrée e_i. Parfois au lieu de parler de la fonction f, on parle d'une relation R comportant, notamment, le triplet (e_i, s_j, s_k). Cela revient au même si l'on précise des conditions sur les triplets permis. Parfois au lieu d'automate, on parle de machines séquentielles (Finite State Machine).

Il faut retrouver ses petits.

A chaque état correspond une ou plusieurs actions. Quand l'automate est dans cet état, les actions sont exécutées (simultanémant ou dans l'ordre, il faut le préciser, ça donne deux variantes). On peut aussi trouver des interprétations où les actions sont exécutées "à l'entrée" dans l'état, ou bien "pendant" l'état, mais on ne voit pas encore bien ce que ces mots peuvent vouloir dire...

ATTENTION : il y a une différence par rapport aux automates étendus vus au premier semestre dans le cours de Langage et Automates. Dans ce cours les actions etaient associees aux TRANSITIONS. ICI aux états. C'est la meme transformation que le passage de Mealy (fct de sortie : g : E X S $\rightarrow$ O) à Moore. C'est anecdotique.

Quand le "vocabulaire d'entrée" comporte des "conditions" booléennes portant sur les variables, si la condition est vraie, le changement d'état peut avoir lieu (comme en langage et automate).

Intérêt du modèle : permet de décrire de fonctionnement d'un système séquentiel pour le simuler, faire des preuves (équivalence, état atteignable) avant de passer à une réalisation logicielle OU

matérielle.

Pour nous c'est l'occasion de montrer une programmation systématique avec l'organisation du programme en mémoire et de revenir sur toutes les notions déjà traitées.

On retrouvera le modèle automate avec actions pour décrire le fonctionnement d'un processeur.

5.2 Exemple

On s'intéresse à la reconnaissance et à l'évaluation d'un nombre à virgule. L'algorithme est exprimé sous forme d'un **automate avec actions**, les actions étant attachées aux états (automate de Moore).

On travaille avec un **processeur** relié à de la **mémoire**.

On considère des nombres à virgule, écrits en binaire (c'est plus facile à évaluer en langage d'assemblage). Les nombres sont écrits en caractères, par exemple au clavier. On peut par exemple ne s'intéresser que à des nombres avec 4 chiffres avant la virgule et 4 après.

L'ensemble des caractères possibles et {0, 1, •, ⊔ }. • est la virgule (un point en anglo-saxon), ⊔ est l'espace final (on pourrait mettre tout autre séparateur).

Pour simplifier on supposera qu'il n'y a pas d'erreurs.

5.2.1 Evaluation

Exemples d'évaluation : des nombres et leurs valeurs

nombre	valeur
⊔	0
011⊔	3
•1001⊔	$\frac{1}{2} + \frac{1}{16}$
101•01⊔	$5 + \frac{1}{4}$
101100•01⊔	erreur de codage

On évalue de la façon suivante :

- E est évalué, c'est le naturel partie entière du nombre représenté. On pourrait détecter si E > 15. (ce serait facile, il y aurait alors plus de 4 chiffres avant la virgule)
- D est évalué, c'est le naturel partie après la virgule du nombre représenté.
- N est évalué, c'est le nombre de chiffres après la virgule du nombre représenté. On pourrait détecter les erreurs si, par exemple, N > 4.

Par exemple 0110•1010⊔ représente 6,625. L'évaluation donne E = 6 ; D = 10 ; N = 4.

5.2.2 Modélisation de l'évaluation par un Automate d'état fini avec actions

L'ensemble d'états est donné par : Etats = { Ini, E0, E1, Poi, D1, D0, Fin }.

L'état initial est Ini.

la figure 5.1 décrit l'automate. Les tableaux ci-dessous donnent la fonction de transition et les actions associées à chaque état :

étatdedépart \ *symbole*	0	1	•	⊔
Ini	E0	E1	Poi	Fin
E0	E0	E1	Poi	Fin
E1	E0	E1	Poi	Fin
Poi	D0	D1		
D0	D0	D1		Fin
D1	D0	D1		Fin
Fin	Fin	Fin	Fin	Fin

Etat	action
Ini	E ← 0, D ← 0, N ← 0
E0	E ← 2 * E
E1	E ← 2 * E + 1
Poi	rien
D0	D ← 2 * D ; N ← N + 1
D1	D ← 2 * D + 1 ; N ← N + 1
Fin	rendre la main

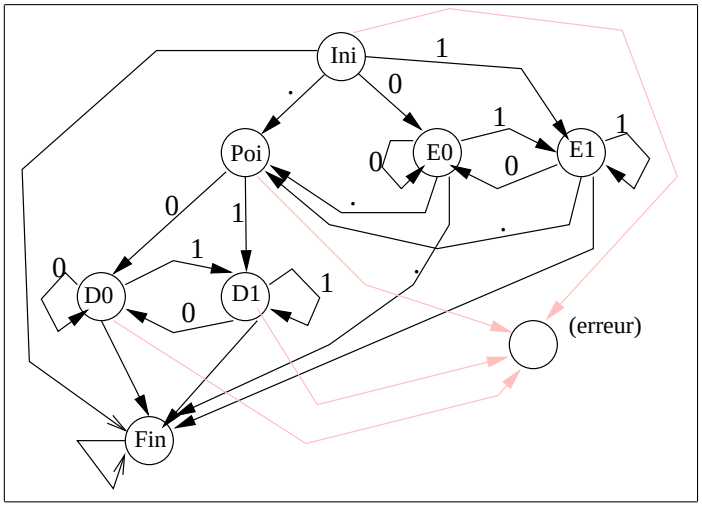


FIG. 5.1 – Automate évaluateur

5.3 Mise en œuvre logicielle

5.3.1 Idée de la mise en œuvre par un langage évolué

Caractères lus au clavier (ou dans un fichier). Le programme est boucle de calcul et changement d'état. Deux fins possibles : état fin ou état erreur

L'ordre d'écriture des états n'a pas d'importance dans le texte ci-dessous.

Etat ← Ini			
while (Etat	différent de	(si il y a lieu Erreur ou)	de Fin)
{case Etat parmi			
	E0 :	E ← 2XE	
		lire entrée	
		case entrée parmi	
			"•" Etat ← Poi
			'0' Etat ← E0
			'1' Etat ← E1
			séparateur Etat ← Fin
	Poi ..		
		lire entrée	
		case entrée parmi	
			'0' Etat ← D0
			'1' Etat ← D1
	In ..	E, D, N ← 0	
}			
Message	de fin	Bonne	(ou mauvaise si il y a lieu)

Il y a une autre methode à base de table de la fonction...
a voir (on n'en parle pas ici...)

5.3.2 Idée de la mise en œuvre en langage d'assemblage

```
adresse 1000 : (état In)
mettre 0 dans la case mémoire représentant la variable E
mettre 0 dans la case mémoire représentant la variable N
mettre 0 dans la case mémoire représentant la variable D
```

```
lire entrée
comparer entrée, symbole'1'
si égal aller à 3000 (état E1)
comparer entrée, symbole'0'
si égal aller à 8000 (état E0)
...
adresse 3000 : (état E1)
lire la case mémoire représentant la variable E
multiplier son contenu par 2
ajouter 1
écrire le résultat dans la case mémoire représentant la variable E
lire entrée
comparer entrée, symbole'separateur'
si égal aller à 7000 (état Fin)
............
adresse 7000 : (état final)
afficher sortie
rendre la main...
```

On peut aussi faire un dessin, j'ai pas le temps qu'il soit beau.

adresse	code des instructions
1000	(E0) mettre 0 dans la case mémoire représentant la variable E
1001	mettre 0 dans la case mémoire représentant la variable N
1002	mettre 0 dans la case mémoire représentant la variable D
1003	...
....	...
1145	lire entrée
1146	comparer avec '0' (code ascii 0x30)
1147	si egal aller a 8000
1148	

5.3.3 Mise en œuvre en langage d'assemblage d'un processeur particulier

Un processeur et son langage machine

Le processeur a 1 registre accumulateur `Acc` de taille 8 bits. Les adresses mémoires sont des mots de 16 bits. La mémoire est organisée en octets (mots de 8 bits).

On peut charger l'accumulateur avec une valeur immédiate ou un mot de la mémoire ((instructions `load#` et `load`). On peut ajouter une valeur immédiate à l'accumulateur (instructions `add#`) ou décaler son contenu de 1 position binaire (instruction `lsl`). On peut ranger le contenu de l'accumulateur dans la mémoire (instruction `store`).

Il y a un registre d'état qui comporte un bit : `Z`. La comparaison de l'accumulateur avec une valeur immédiate positionne le bit du mot d'état.

Le compteur de programme (`pc`) repère l'instruction qui suit celle qui est en cours d'exécution.

Il existe deux types d'instructions de rupture de séquence :

– branchement conditionnel : `bne` ou `beq`

– branchement inconditionnel : `jmp`

L'instruction `bne` (ou `beq`) a un opérande qui est une valeur de déplacement sur 8 bits ; il s'agit d'un branchement relatif. L'instruction `jmp` a un opérande sur 16 bits qui est une adresse ; il s'agit d'un branchement absolu.

Ce processeur est du style de ceux des années 75 (6800, 6502, Z80, 8088).

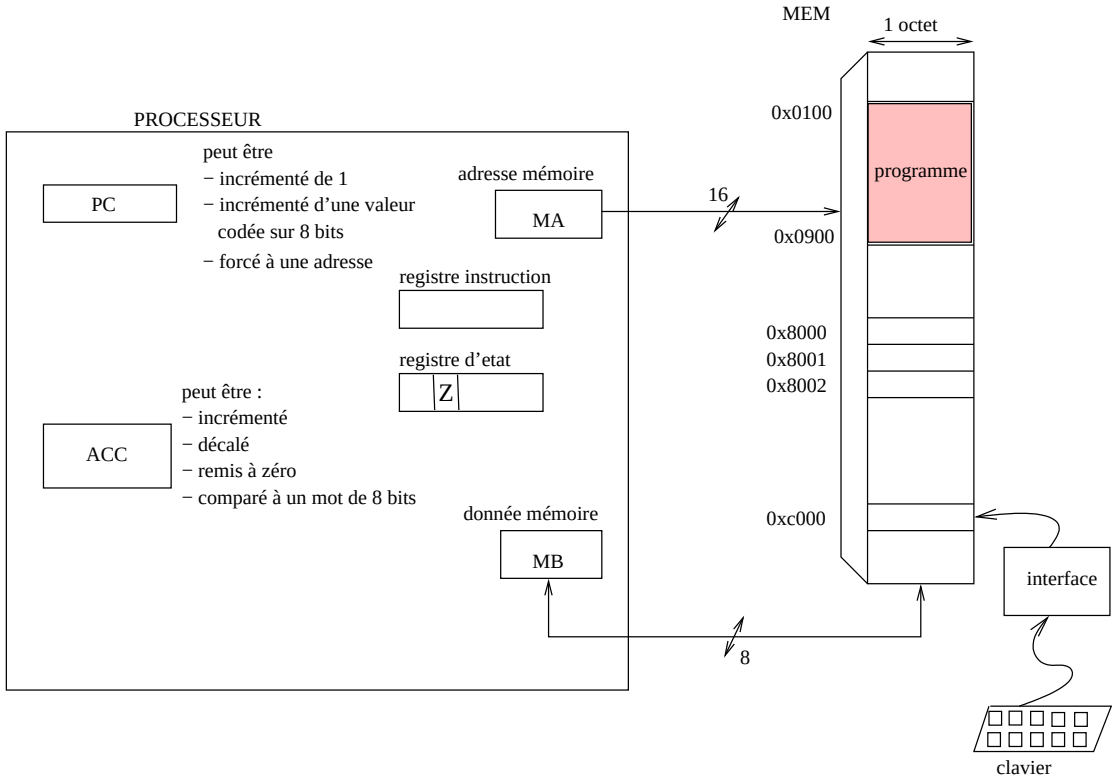


FIG. 5.2 – Processeur et mémoire

La figure 5.2 donne un schéma de l'organisation processeur, mémoire, périphérique d'entrée (clavier ici) et le tableau 5.3 donne l'ensemble des instructions avec leur signification et les règles de codage ; le code opération est toujours sur 1 octet (le premier), l'opérande quand il existe est sur 1 octet (`vi`, `depl`) ou 2 octets (`adr`, le premier des deux octets représente les poids forts de l'adresse).

On convient que la lecture à l'adresse `0xC000` fournit un octet qui est le code ASCII du caractère courant.

On range les valeurs `E`, `D` et `N` respectivement aux adresses `0x8000`, `0x8001` et `0x8002`.

Après un certain temps, le système d'exploitation passe la main à notre programme qui débute à l'adresse `0x0100`.

instruction	signification	codage
`load adr`	`Acc <-- Mem[adr]`	3 octets
`load# vi`	`Acc <-- vi`	2 octets
`store adr`	`Mem[adr] <-- Acc`	3 octets
`add# vi`	`Acc <-- Acc + vi`	2 octets
`lsl`	`Acc <-- Acc * 2`	1 octet
`cmp vi`	`Acc - vi`	2 octets
`beq depl`	si Z=0 `pc <-- pc + depl`	2 octets
`bne depl`	si Z=1 `pc <-- pc + depl`	2 octets
`jmp adr`	`pc <-- adr`	3 octets

FIG. 5.3 – Jeu d'instructions

Programmation de l'évaluateur

Le codage est systématique. On pourrait très facilement faire un compilateur pour un langage à automate et générer de tels programmes en langage machine.

Le code de chaque état est installé en mémoire à une adresse différente pour chaque état. Le passage d'un état à un autre est fait par une rupture de séquence.

En supposant qu'on implante le code des états Ini, E0, E1, D0, D1 et Poi aux adresses respectives 100, 200, 300, 400, 500 et 800, le dessin ci-dessous représente la mémoire avec les différents "tronçons" du programme, correspondants aux différents états de l'automate.

```
adresse      données
0000         système
0100         état Ini
...          ...
0123         fin de état Ini
0200         état E0
...
0223         fin de état E0
0300         état E1
...
0311         fin de état E1
0400         état D0
...
???          fin de état D0
0500         état D1
...
???          fin de état D1
0800         Poi
...
???          fin de Poi
0900         Fin
...
0902         fin de Fin
```

Les figures 5.4 et 5.5 montre le programme. On peut distribuer la première et faire trouver la seconde...

Remarques :

- Les parties calcul de l'état suivant sont identiques sauf en ce qui concerne les adresses.
- Pour programmer l'état Fin avec un branchement relatif il faut une condition `toujours` pour l'instruction de branchement (genre `bal`) et un déplacement de `-2`.

Exercices : Ecrire les parties de programmes correspondants aux états Poi, D0 et D1.

Observation du comportement au niveau des bus, du processeur

On voudrait observer les informations (signaux) qui passent sur les bus données (8 bits) et adresses (16 bits) lors d'une exécution de notre programme.

Pour cela, on fait semblant d'avoir un oscilloscope qu'on installe entre le processeur et la mémoire. On commence à observer à partir du moment où c'est le programme automate qui s'exécute (adresse 100). On regarde sur les bus quand il y a un échange, et dans quel sens. Cet oscilloscope est particulier,

0	système	de lancement	du programme	saut à 100
....	..	.....	.....	.....
adresse	codeop	2ème octet éventuel	3ème octet	commentaire
Etat Ini				
100	load#	00		`acc <-- 0`
102	store	80	00	`E = Mem[0x8000] <-- 0`
105	store	80	01	`D = Mem[0x8001] <-- 0`
108	store	80	02	`N = Mem[0x8002] <-- 0`
10B	load	C0	00	lecture
10E	cmp	30		code ASCII de '0' = 0x30 = 48
110	bne	3		si (caractère != '0') alors aller à 3 octets en avant
112	jmp	02	00	car=='0' : aller à addresse de l'état E0
115	cmp	31		code du '1' = 0x31
117	bne	3		
119	jmp	03	00	adresse de l'état E1
11C	cmp	2E		code de '.' = 0x2E
11E	bne	3		
120	jmp	08	00	adresse de l'état Poi
123	jmp	09	00	adresse de l'état Fin
Etat E0				
200	load	80	00	`Acc <-- E = Mem[0x8000]`
203	lsl			`Acc <-- Acc * 2`
204	store	80	00	`E = Mem[0x8000] <-- Acc`
207	load	C0	00	lecture
...	calcul de	l'état suivant		
Etat Fin				
900	jmp	09	00	boucle infinie...

FIG. 5.4 – Le programme en langage machine de l'automate. Les codes opérations ne sont pas donnés en hexadécimal pour faciliter la lisibilité. A la ligne 10E se trouve la mise en oeuvre du "si cond alors aller a E0" par un "si non cond alors ne pas aller a E0"

on ne "voit" QUE les cycles où il y a échange entre le processeur et la mémoire. Il y a peut-être d'autres cycles d'horloge mais on n'en voit pas la trace.

La figure 5.6 montre le comportement du programme comportant le code pour les états Ini, E0, E1 et Fin (Cf figures 5.4 et 5.5). Les séparations sont entre les instructions.

Dans le tableau de la figure 5.6 le caractère lu à l'instruction de la ligne 0x10B est un '0'.

Si on lit un '1', on obtient le comportement donné dans la figure 5.7. Lors de l'exécution de l'instruction de la ligne 0x110, le compteur de programme repère l'instruction suivante : `pc==0x112`, la comparaison donne faux donc la prochaine instruction à exécuter est à l'adresse : 0x112+3 = 0x115.

Sujets de réflexion

1. Que faudrait-il modifier si on ajoutait un état d'erreur ? Indiquer les modifications de l'automate et du programme.
2. Indiquer comment on pourrait modifier l'automate pour n'accepter QUE des octets avec 4 bits avant le • et 4 bits après.

Réponses :

1. – Les modifications de l'automate sont dans la figure 5.1.
   – Modification du programme de l'évaluateur :

123	cmp y..	code du ⊔
125	bne ????	erreur

   – Programme pour D0 :

```
ld 0x80 01
sla
st  0x80 01
ld 0x80 02
inc
st 0x80 02
ld 0xC0 00
cmp 0x30
bne +3
jmp D0
cmp 0x31
bne +3
jmp D1
cmp 0x... code de espace
bne +3
jmp fin
jmp erreur
```

2. Pour compter les bits : soit on ajoute un compteur (comme N) et on teste ; soit on quadriplique les états (après quatre transitions sur '0' on a une transition vers erreur). Les automates d'etat fini ca compte mal...

adresse	codeop	2ème octet éventuel	3ème octet	commentaire
Etat E0				
200	load	80	00	`Acc <-- E = Mem[0x8000]`
203	lsl			`Acc <-- Acc * 2`
204	store	80	00	`E = Mem[0x8000] <-- Acc`
207	load	C0	00	lecture
20E	cmp	30		code ASCII de '0' = 0x30 = 48
210	bne	3		si (caractère != '0') alors aller à
				3 octets en avant
212	jmp	02	00	car=='0' : aller à addresse de l'état E0
215	cmp	31		code du '1' = 0x31
217	bne	3		
219	jmp	03	00	adresse de l'état E1
21C	cmp	2E		code de '.' = 0x2E
21E	bne	3		
220	jmp	08	00	adresse de l'état Poi
223	jmp	09	00	adresse de l'état Fin
Etat E1				
300	load	80	00	`Acc <-- E = Mem[0x8000]`
303	lsl			`Acc <-- Acc * 2`
304	add#	1		`Acc <-- Acc + 1`
306	store	80	00	`E = Mem[0x8000] <-- Acc`
309	load	C0	00	lecture
30C	cmp	30		code ASCII de '0' = 0x30 = 48
30E	bne	3		si (caractère != '0') alors aller à
				3 octets en avant
310	jmp	02	00	car=='0' : aller à addresse de l'état E0
313	cmp	31		code du '1' = 0x31
315	bne	3		
317	jmp	03	00	adresse de l'état E1
31A	cmp	2E		code de '.' = 0x2E
31C	bne	3		
31E	jmp	08	00	adresse de l'état Poi
311	jmp	09	00	adresse de l'état Fin

FIG. 5.5 – Etats E0 et E1

adresse	lire/écrire	données	destinataire		source
100	lire	code de load#	proc	←	mem
101	lire	00	proc	←	mem
102	lire	code de store	proc	←	mem
103	lire	80	proc	←	mem
104	lire	00	proc	←	mem
8000	ecrire	00	mem	←	proc
105	lire	code de store	proc	←	mem
106	lire	80	proc	←	mem
107	lire	01	proc	←	mem
8001	ecrire	00	mem	←	proc
108	lire	code de store	proc	←	mem
109	lire	80	proc	←	mem
10A	lire	02	proc	←	mem
8002	ecrire	00	mem	←	proc
10B	lire	code de load	proc	←	mem
10C	lire	C0	proc	←	mem
10D	lire	00	proc	←	mem
C000	lire	30	proc	←	périph
10E	lire	code de cmp	proc	←	mem
10F	lire	30	proc	←	mem
110	lire	code de bne	proc	←	mem
111	lire	3	proc	←	mem
112	lire	code de jmp	proc	←	mem
113	lire	02	proc	←	mem
114	lire	00	proc	←	mem
200	lire	code de ld	proc	←	mem

FIG. 5.6 – Comportement du programme

adresse	lire/écrire	données	destinataire		source
100	lire	code de load#	proc	←	mem
101	lire	00	proc	←	mem
102	lire	code de store	proc	←	mem
103	lire	80	proc	←	mem
104	lire	00	proc	←	mem
8000	ecrire	00	mem	←	proc
105	lire	code de store	proc	←	mem
106	lire	80	proc	←	mem
107	lire	01	proc	←	mem
8001	ecrire	00	mem	←	proc
108	lire	code de store	proc	←	mem
109	lire	80	proc	←	mem
10A	lire	02	proc	←	mem
8002	ecrire	00	mem	←	proc
10B	lire	code de load	proc	←	mem
10C	lire	C0	proc	←	mem
10D	lire	00	proc	←	mem
C000	lire	**31**	proc	←	périph
10E	lire	code de cmp	proc	←	mem
10F	lire	30	proc	←	mem
110	lire	code de bne	proc	←	mem
111	lire	3	proc	←	mem
115	lire	code de cmp	proc	←	mem
116	lire	31	proc	←	mem
117	lire	code de bne	proc	←	mem
118	lire	3	proc	←	mem
119	lire	code de JMP	proc	←	mem
11A	lire	03	proc	←	mem
11B	lire	00	proc	←	mem
300	lire	code de load	proc	←	mem
301	lire	80	proc	←	mem
302	lire	00	proc	←	mem
8000	lire	0	proc	←	mem
303	lire	code de lsl	proc	←	mem
304	lire	code de add#	proc	←	mem
305	lire	1	proc	←	mem
306	lire	code de store	proc	←	mem
307	lire	80	proc	←	mem
308	lire	00	proc	←	mem
8000	ecrire	01	mem	←	proc

FIG. 5.7 – Comportement du programme en cas de lecture d'un '1'

Chapitre 6

Programmation des appels de procédure et fonction (3 Séances)

6.1 Objectifs

Comprendre les notions d'appel/retour de fonction/procédure. Comprendre comment gérer les paramètres lors de l'exécution d'un programme. On traite le passage par valeur, le résultat d'une fonction et le passage par adresse. On traite le passage des paramètres dans des registres et/ou dans la pile.

Il me semble qu'il faut parler en même temps des variables locales et aussi de la sauvegarde des temporaires (registres utilisés localement à une fonction). En effet il est bien rare qu'une procédure ne comporte pas de variables locales...

L'accès aux objets stockés dans la pile (variables locales et/ou paramètres) se fera par un registre dédié à cette tâche : `fp` pour "frame pointer" dans le processeur ARM (et la plupart des processeurs actuels). Ce registre est en fait un repère sur la base de l'environnement courant, repère qui reste fixe pendant la durée d'exécution de la procédure/fonction. Cette méthode est celle utilisée de façon standard par un compilateur et qui permet la prise en compte de la possibilité d'imbriquer des procédures.

On commence par traiter des fonctions avec des paramètres données et un résultat. On ne parle de passage de paramètre permettant de rendre une valeur que dans un deuxième temps.

6.2 Introduction à travers un exemple : programmation d'une fonction

On définit une fonction `PP` et on l'appelle deux fois avec des arguments différents. Voilà l'écriture en C et en ADA.

```
en C                          en ADA
====                          ======
int PP (int x) {              function PP (x : in integer)
int z, p;                              return integer is
   z = x  + 1;                z, p : integer;
   p = z + 2;                 begin
   return (p);                   z := x + 1;
}                                p := z + 2;
                                 return p;
main () {                     end
```

```
int i, j, k;
   i = 0;                    i, j, k : integer;
   j = i + 3;                begin
   j = PP (i + 1);              i := 0; j = i + 3;
      /* point a */             j := PP (i + 1);
   k= PP(2 * (i + 5));          k := PP(2 * (i + 5));
      /* point b */          end
}
```

Quelques rappels de vocabulaire : Ici le `main` est nommé `appelant`, la fonction est nommée `appelé` Une fonction a un (ou des) paramètre(s) et calcule une valeur que l'on appellera *résultat de la fonction*. La fonction PP a un `paramètre x` qui constitue une `donnée` pour la fonction, on parle de *paramètre formel*. La fonction PP calcule une valeur de type entier, le `résultat de 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)`.

Les variables z et p sont appelées variables locales à la fonction.

Le premier appel revient à exécuter le corps de la fonction en remplaçant `x` par `i+1` ; le deuxième appel consiste en l'exécution du corps de la fonction en remplaçant `x` par `2*(i+5)`. Ça vaut le coup de rappeler ça ; ce qui est derrière c'est la notion de subtitution qui n'est pas toujours acquise...

6.2.1 Traduction langage d'assemblage ARM en plaçant toutes les valeurs dans des registres ?

Chaque valeur représentée par une variable ou un paramètre doit être rangée quelque part en mémoire : mémoire centrale ou registres. Pour commencer on place **tout dans des registres**.

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

- i,j,k dans r0,r1,r2
- z dans r3, p dans r4
- la valeur x dans r5
- la valeur rendue par la fonction dans r6
- si on a besoin d'un registre pour faire des calculs on utilisera r7

Pour l'appel à proprement parler on verra après, pour l'instant on écrit appeler/retourner.

Faire remarquer les séquences de code pour la préparation des paramètres avant l'appel et pour la récupération du résultat après.

```
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
      @ ------------
      add r5, r0, #1  @ x <-- i+1    preparation des param
      appeler PP
      mov r1, r6      @ j <-- ...    recuperation du resultat
      @ ------------
      add r7, r0, #5       @ r7 <-- i+5
```

```
      mov r5, r7, lsl #1   @ r5 <-- 2*r7
      appeler PP
      mov r2, r6           @ k <-- ...
      @ ------------
```

Une fois les conventions fixées, on peut écrire le code de la fonction indépendamment du code correspondant à l'appel.

6.2.2 "appeler" et "retourner" en langage d'assemblage ARM ?

Quel est le problème : l'appel pourrait très bien se faire par une instruction de rupture de séquence inconditionnelle, par exemple un `BAL` en ARM. MAIS comment revenir ensuite, le problème est le retour : comment à la fin de l'exécution du corps de la fonction, dire au processeur l'adresse à laquelle il doit se brancher ?

Il existe une instruction de rupture de séquence particulière qui permet au processeur avant qu'il ne réalise le branchement (c.a.d. avant qu'il ne transfère le contrôle) de garder l'adresse de l'instruction qui suit le branchement. Cette adresse est appelée *adresse de retour*.

Nouvelle question : où est gardée cette adresse ?

Dans le processeur ARM, l'instruction `BL` (attention cela n'a rien à voir avec `BAL`...) qui s'appelle *branch and link* réalise un branchement inconditionnel avec sauvegarde de l'adresse de retour dans le registre nommé `LR` (i.e. `r14`).

D'où la programmation complète de notre exemple :

```
PP: add r3, r5, #1   @ z <-- x + 1
    add r4, r3, #2   @ p <-- z + 2
    mov r6, r4       @ rendre p
    mov pc, lr       @ retour

main: mov r0, #0      @ i <-- 0
      add r1, r0, #3  @ j <-- i + 3
      @ ------------
      add r5, r0, #1  @ x <-- i+1
      bl PP           @ appel de PP
      mov r1, r6      @ j <-- ...
      @ ------------
      add r7, r0, #5       @ r7 <-- i+5
      mov r5, r7, lsl #1   @ r5 <-- 2*r7
      bl PP                @ appel de PP
      mov r2, r6           @ k <-- ...
      @ ------------
```

Exercice : on met des adresses, arbitrairement pour la première, devant chaque instruction. Suivre le flot de contrôle et l'évolution de PC au fur et à mesure de l'exécution du programme.

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

On a déjà vu en TP des appels de fonctions/procédures :

```
LDR r1, adresse de début d'une chaîne de caractère
BL EcrChaine
```

```
LDR r1, un nombre
BL EcrHexa32
```

6.2.3 Questions de cohérence, valeurs temporaires

Supposons que dans la fonction `PP` on ait un calcul compliqué à faire et que l'on utilise pour cela le registre r7. Supposons qu'après l'appel `k= PP(2 * (i + 5))` on ait besoin de la valeur `i+5` et qu'on veuille la prendre dans r7 ...

```
PP: ...
    modifie r7
    ...

    add r7, r0, #5       @ r7 <-- i+5
    mov r5, r7, lsl #1   @ r5 <-- 2*r7
    bl PP     @ modification de r7
    mov r2, r6
    ...
    utilisation de r7 qui contient la valeur i+5 @ c'est incorrect !!!
```

6.2.4 Conclusions

Le codage se passe plutôt bien (insister sur l'aspect systématique) ; il suffit de bien respecter les conventions choisies. En particulier il ne faut pas utiliser les registres pour autre chose ; par exemple si on voulait utiliser le registre r0 pour z... la valeur de i serait modifiée entre les points a et b, ce qui serait incorrect.

Paramètres : il faut une zone de stockage **commune** à l'appelant et à l'appelé. L'appelant (la procédure qui appelle) y range les valeurs **avant** l'appel et l'appelé (la procédure qui est appelée) y prend ces valeurs et les utilise. On reviendra sur les résultats plus tard.

Variables locales : il faut une zone de mémoire **privée** pour chaque procédure pour y ranger ses variables locales. En effet, il ne faut pas que cette zone interfère, écrase par exemple des valeurs de variables globales ou locales à l'appelant. Dans notre exemple, la zone utilisée pour `z` et `p` ne doit pas modifier la zone utilisée pour `i, j` et `k`.

Valeurs temporaires : ne doivent pas interférer avec les autres.

Généralisation : On peut déjà se douter que la méthode utilisée ne va pas être facile à généraliser... On n'a en général pas assez de registres (même si ce n'est pas vrai pour les petits exemples que l'on écrit, on fera comme si).

Mais il y a des problèmes plus graves.

6.2.5 Problèmes à résoudre

Appels de procédure en cascade : un programme appelle une procédure `P` qui elle-même appelle une procédure `Q`.

programme	...	procédure P	...	procédure Q	...
	...		...		...
	appel P		appel Q		...
α	...	β	...		retour
	...		...		
	...		retour		

Lors de l'appel de P, l'adresse α est rangée dans `lr` et lors de l'appel de Q, l'adresse β est rangée dans `lr` et écrase la valeur précédemment sauvegardée.

Procédure récursives : On considère la fonction qui calcule la factorielle d'un entier :
$0! = 1 \qquad n! = n * (n-1)!$

```
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
```

Fixons des conventions pour la mise en oeuvre de la fonction `fact` : le paramètre x dans le registre `r0`, le résulta de la fonction dans le registre `r1`.

Considérons l'appel `fact(3)` : `x=r0<-3`, il est différent de 1, le calcul se poursuit avec l'appel `fact(2)` tout en gardant la valeur 3 dans la variable `x` de façon à pouvoir effectuer par la suite le produit de `x` par le résultat de `fact(2)`. Pour l'appel de `fact(2)` on doit placer `x=r0<-2` et c'est là que ça coince : on perd la valeur 3 qu'il fallait garder.

Conclusion : 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**.

C'est la même chose pour les variables locales.

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

Imaginons que nous ayions réservé une case pour la variable `loc`, on veut y ranger la valeur de `fact(x-1)` mais pour effectuer le travail nécessaire à cet appel, il va falloir aussi une case `loc` et en fait il en faut autant que d'appels récursifs...

6.3 Appels en cascade

6.3.1 Le problème et une solution à travers un exemple

En langage évolué : A_i, B_i, C_i sont des actions élémentaires. X est une expression booléenne (vraie ou fausse). A, B, C sont des procédures, A est la principale !

procédure A	A1	procédure B	B1	procédure C	C1
	A2		B2		B
	B		B3		C2
	A3				si condX C
	C				C3
	A4				C4

Représentation des exécutions possibles :

A1 A2 (B1 B2 B3) A3 (C1 (B1 B2 B3) C2 nonX C3 C4) A4

A1 A2 (B) A3 (C1 (B) C2 ouiX (C1 (B) C2 nonX C3 C4) C3 C4) A4

Comment mettre en place de tels enchaînements en langage machine ?

On a vu qu'en ARM, l'adresse de retour est gardée dans le registre `lr`. Pour résoudre le problème précédent, il faudrait plusieurs registres ou cases mémoire. Le problème est que l'on ne sait pas toujours à l'avance combien il en faudrait (par exemple dans le cas des procédures récursives).

6.3.2 Mécanisme de pile

analogie pile de chemises :

- dépiler : on en prend une sur le sommet
- empiler : on en pose une sur le sommet

Comment réaliser une pile ?

- une zone de mémoire,
- un repère sur le "sommet" de la pile (SP, registre pointeur de pile)
- deux choix indépendants :
  - comment progresse la pile : on dit que le sommet est en direction des adresses croissantes ou décroissantes
  - on dit que le pointeur de pile pointe vers une case vide ou pleine

On peut ainsi avoir 4 types de réalisation possibles :

sens pointage	croissant 1^{er} vide	croissant der^{er} plein	décroissant 1^{er} vide	**décroissant der^{er} plein**
empiler reg	M[sp]←reg sp←sp+1	sp←sp+1 M[sp]←reg	M[sp]←reg sp←sp-1	**sp ←sp-1** **M[sp]←reg**
dépiler reg	sp←sp-1 reg←M[sp]	reg←M[sp] sp←sp-1	sp←sp+1 reg←M[sp]	**reg←M[sp]** **sp←sp+1**

6.3.3 Une solution : l'instruction d'appel sauve l'adresse de retour dans une pile

remarque : dans ce paragraphe et le suivant on se concentre sur l'évolution du PC lors de l'exécution des appels/retours. On ne veut pas se préoccuper des problèmes de codage et plus particulièrement des tailles de codage des instructions. On considère ainsi que toutes les instructions sont codées sur la même taille à savoir 1.

Ce n'est pas la solution mise en oeuvre dans le processeur ARM. A l'appel de sous-programme, il y a deux actions :

- sauvegarde de l'adresse de retour (PC + 1) dans une pile (=empiler PC+1)
- modification du compteur programme (rupture), PC=adresse de la procédure

Au retour, PC prend pour valeur l'adresse de retour empilée. Le retour devient depiler PC.

On reprend l'exemple en mettant des adresses.

10	A1	20	B1	30	C1
11	A2	21	B2	31	empiler 32 ; sauter à 20 (B)
12	empiler 13 ; sauter à 20 (B)	22	B3	32	C2
13	A3	23	retour : dépiler PC	33	si condX empiler 34 ; sauter à 30
14	empiler 15 ; sauter à 30(C)			34	C3
15	A4			35	C4
				36	retour : dépiler PC

Trace d'exécution (voir figure 6.1) : dans ce diagramme le "sommet" est à gauche. On n'a pas besoin de savoir comment c'est réalisé pour comprendre. les colonnes d'instructions, c'est juste pour faire joli.

PC	instructions				état de la pile
10	A1				{}
11	A2				{}
12	saut 20				empile 13
20		B1			{13}
21		B2			{13}
22		B3			{13}
23		retour			sommet = 13
13	A3				{}
14	saut 30				empile 15
30		C1			{15}
31		saut 20			empile 32
20			B1		{32 ; 15}
21			B2		{32 ; 15}
22			B3		{32 ; 15}
23			retour		sommet = 32
32		C2			{15}
33		cond :saut 30			empile 34
30			C1		{34 ; 15}
31			saut 20		empile 32
20				B1	{32 ; 34 ; 15}
21				B2	{32 ; 34 ; 15}
22				B3	{32 ; 34 ; 15}
23				retour	sommet = 32
32			C2		{34 ; 15}
33			cond :saut 30		faux
34			C3		{34 ; 15}
35			C4		{34 ; 15}
36			retour		sommet = 34
34		C3			{15}
35		C4			{15}
36		retour			sommet = 15
15	A4				{}

FIG. 6.1 – Trace d'exécution du programme donné en exemple.

6.3.4 Une autre solution : l'instruction d'appel sauve l'adresse de retour dans un registre

C'est le cas du ARM. Un registre particulier `lr (=r14)`. `bl adr_proc` fait sauvegarde adresse retour dans lr et pc= adr_proc.

C'est le programmeur qui doit gérer les sauvegardes dans la pile (si nécessaire)

Ca devient :

```
10 A1
11 A2
12 B = (sauver 13 dans LR; sauter à 20)
13 A3
14 C = (sauver 15 dans LR; sauter à 30)
15 A5

20.0 Empiler lr
20.1 B1
21   B2
22   B3
23.0 depiler dans lr
23.1 mov pc, lr (restaure lr dans le compteur programme)

30.0 empiler lr
30.1 C1
31 B = (sauver 32 dans lr; sauter à 20)
32 ..
...
36.0 dépiler vers lr
36.1 mov pc, lr (restaure lr dans le compteur programme)
```

Remarque : quand une procédure n'en appelle pas d'autres (on parle de proc. feuille) la sauvegarde dans la pile n'est pas nécessaire. C'est le cas de B dans l'exemple.

```
20.1 B1
21   B2
22   B3
23.1 mov pc, lr
```

6.4 Gestion des variables et paramètres en pile

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.

Reprenons un exemple comme le précédent en y ajoutant des paramètres. Chaque variable et/ou paramètre est rangé dans la mémoire et la case mémoire associée est repérée par son adresse.

```
procedure A {procédure principale, sans paramètres}
var u : entier
   u=2; B(u+3); u=5+u; B(u)
```

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

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

On supposera que ecrire est une procédure qui demande son paramètre dans le registre `r1` comme en TP.

Etudions le flot d'exécution de ces procédures en partant de l'exécution de `A` :

- u est rangé dans la mémoire à une adresse `adr_u`
- la valeur `u+3` est calculée puis rangée à une adresse `adr_param_B`
- le flot d'exécution est maintenant en début du corps de la procédure `B`, les variables `s` et `v` sont à des adresses `adr_s` et `adr_v`, le paramètre formel `x` doit être "lié" au paramètre effectif `u+3` et est donc à l'adresse `adr_param_B`.
- le calcul de `s=x+4` est :
  ```
  ldr r0, adr_param_B
  ldr r0, [r0]
  add r0, r0, #4
  ldr r1, adr_s
  str r0, [r1]
  ```
- la valeur `s+1` est calculée puis rangée à une adresse `adr_param_C`
- le flot d'exécution est maintenant en début du corps de la procédure `C`. La variable `t` est à l'adresse `adr_t`, le paramètre formel `y` doit être "lié" au paramètre effectif `s+1` et est donc à l'adresse `adr_param_C`.
- mise à jour de `t`, appel de `ecrire`, mise à jour de `t`, puis retour dans le corps de la procédure `B` pour calculer `v=2`, puis `s+v`, etc. Les cases mémoire d'adresse `adr_t` et `adr_param_C` n'ont plus de rason d'être accessibles. Par contre, il faut que les adresses des variables accessibles soient visibles...

Si on veut gérer ces adresses en mémoire au fur et à mesure des besoins, on s'aperçoit que l'on gère des zones de mémoire en pile. On a besoin d'une case mémoire pour `u` puis d'une autre pour `u+3` puis de deux autres pour `s` et `v` puis d'une autre pour `s+1` puis on n'a plus besoin de celles correspondant à `s+1=y`, etc.

Dessin.

L'idée consiste donc à utiliser une zone de mémoire que l'on va gérer en pile : on va en fait gérer une pile de blocs de la taille nécessaire pour ranger les paramètres et les variables locales. Et au milieu de tout ça il faudra gérer les adresses de retour. On pourrait avoir plusieurs piles mais la tradition et la structure des processeurs est telle qu'on place tout dans la même.

On aura la même approche pour tout : résultats de fonctions, paramètres autres que données, etc.

6.4.1 Programmation des appels de procédures avec les paramètres dans la pile

La structure des programmes appelants et appelés est :

```
appelant P                      appelé Q
                                 sauver adresse de retour
préparer les paramètres          réserver de la place pour les variables locales
et les empiler                   /\
BL Q                             |
recupérer la place               |   travail sur variables locales
```

```
alouée aux paramètres            |   et paramètres
                                 |
                                 \/
                                 libérer place variables locales
                                 récupérer adresse de retour
                                 retour
```

Dessin de la pile

```
       |----------------|
SP-->  | var loc        |
       |                |
       |----------------|
       | adresse retour |
       |----------------|
       |                |
       | paramètres     |
       |----------------|
       |                |
```

La question qui se pose est comment accéder aux valeurs des paramètres et des variables locales. On pourrait utiliser le pointeur de pile mais dès que l'on va empiler quelque chose (par exemple pour sauvegarder une valeur car on a besoin d'un registre) il faudra re-calculer les déplacements.

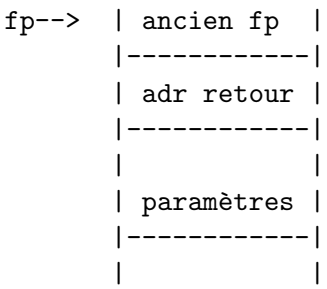
Une solution plus simple consiste à utiliser un repère sur l'environnement courant de l'appel (paramètres et variables locales) et qui reste fixe pendant toute la durée d'exécution de la procédure. Ce repère est traditionnellement appelé "frame pointer" en compilation et la plupart des processeurs l'ont pris en compte dans leur organisation : il est noté `fp` dans le processeur ARM.

Comme pour le registre mémorisant l'adresse de retour, ce registre fp doit être sauvegardé avant d'être à nouveau utilisé. D'ou la structure de code :

```
appelant P                      appelé Q
                                 sauver adresse de retour
préparer les paramètres          sauver l'ancienne valeur de fp
BL Q                             et placer fp pour repérer les
récupérer place                  nouvelles variables
                                 réserver de la place pour les variables locales
                                 /\
                                 |   travail sur variables locales
                                 |   et paramètres
                                 |
                                 \/
                                 libérer place variables locales et restaurer fp
                                 récupérer adresse de retour
                                 retour
```

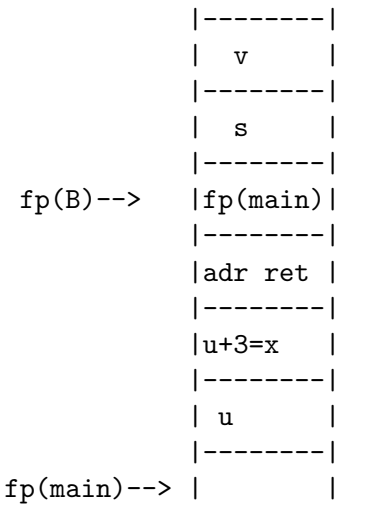
Structure de la pile

```
       |------------|
       | var loc    |
       |            |
       |------------|
```



D'ou si les adresses sont sur 4 octets, l'accès aux variables locales se fait avec une adresse de la forme : fp - déplacement et l'accès aux paramètres avec une adresse de la forme : fp + 8 + déplacement

Application à l'exemple :




```
B: @ empiler adr ret = lr
   sub sp, sp, #4
   str lr, [sp]
   @ empiler ancien fp
   sub sp, sp, #4
   str fp, [sp]
   @ mettre en place fp
   mov fp, sp
   @ reserver 2*4 octets pour 2 variables
   sub sp, sp, #8
   @ s=x+4
   ldr r1, [fp, #+8]
   add r1, r1, #4
str r1, [fp, #-4]
@ C(s+1)
appelC1:@ empiler parametre s+1
        ldr r1, [fp, #-4]
        add r1, r1, #1
        sub sp, sp, #4
        str r1, [sp]
        @ appel
        bl C
```

```
  @ récuperer la memoire associee au parametre
  add sp, sp, #4
        @ la pile est dans le meme etat que lorsque le controle etait en appelC1
   @ v=2
mov r1, #2
str r1, [fp, #-8]
   @ C(s+v)
appelC2:@ empiler parametre s+v
        ldr r1, [fp, #-4]
        ldr r2, [fp, #-8]
        add r1, r1, r2
        sub sp, sp, #4
        str r1, [sp]
        @ appel
        bl C
  @ récuperer la memoire associee au parametre
  add sp, sp, #4
        @ la pile est dans le meme etat que lorsque le controle etait en appelC2
   @ liberer  la place occupee par les variables locales s et v
add sp, sp, #8
@ recuperer le fp de l'appelant, ici fp(main)
ldr fp, [sp]
add sp, sp, #4
@ recuperer l'adresse de retour
ldr lr, [sp]
add sp, sp, #4
   @ la pile est dans le meme etat que lorsque le controle etait en B
@ retour
mov pc, lr
```

Exercices :

- écrire en ARM le code de main et de la procédure C
- choisir des valeurs d'adresses arbitraires pour chaque instruction, choisir une valeur de sp et imaginez l'évolution de pc, sp, fp, r1, r2 au cours de l'exécution du programme proposé. Il peut être utile de dessiner le contenu de la pile.

6.4.2 Appel de fonction avec le résultat de la fonction dans la pile

Nous avons vu que les paramètres sont empilés avant l'appel ; en effet, c'est la procédure appelante qui connait les informations, par conséquent elle les prépare (calcul) puis les empile avant de réaliser l'appel (bl).

Par contre, le résultat d'une fonction est calculé par l'appelée puis doit être rangé à un emplacement accessible par l'appelante de façon à ce que cette dernière puisse le récupérer. Si l'on veut utiliser la pile, l'appelant, avant l'appel, peut réserver une case pour ce résultat, l'appelé rangera son résultat dans cette case dont le contenu sera récupéré après le retour par l'appelant.

Cela nous donne la configuration suivante pour la pile avant l'appel d'une fonction qui a deux paramètres données et un résultat.

```
      |-------------------|
sp--> | adresse de retour |
```

```
      |------------------|
      |                  | une place pour le résultat de la fonction
      |------------------|
      | paramètre donnée |
      |------------------|
      | paramètre donnée |
      |------------------|
```

Et voici la pile lors de l'exécution du corps de la fonction ; les variables locales sonr accessibles par une adresse de la forme : fp-depl avec depl ≥ 4, les paramètres données par les adresses : fp+8 et fp+8+4 et la case résultat par l'adresse fp+8+8.

```
      |------------------|
     | var locale       |
      |------------------|
fp-->| ancien fp        |
      |------------------|
      | adresse de retour |
      |------------------|
      |                  | une place pour le résultat de la fonction
      |------------------|
      | paramètre donnée |
      |------------------|
      | paramètre donnée |
      |------------------|
```

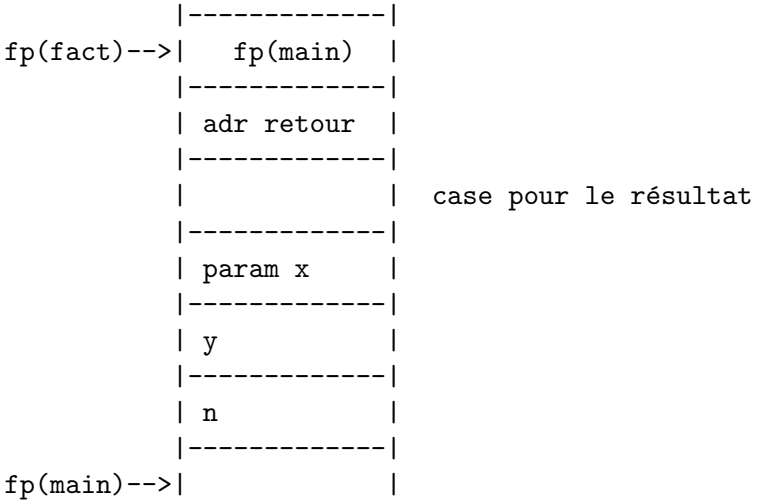
La structure du code de l'appel de la fonction et du corps de la fonction sont les suivantes :

```
appelant: ...
          ...
       ** ...
          empiler les valeurs des paramètres données
          réserver de la place sur la pile pour ranger le résultat de la fonction
          appel (adr retour dans lr)
          récupérer le résultat
          défaire la partie de la pile correspondant aux paramètres données et au réssultat
          la pile est dans son état initial (comme en **)
          ...
          ...

appelé: empiler lr
        empiler la valeur de fp
  mettre le nouveau fp en place
  reserver de la place pour les variables locales
        ...
        ... accès parametres et resultat de fonction par [sp, #8+déplacement]
        ...
  recuperer place allouee aux variables locales
  recuperer l'ancienne valeur de fp
        recuperer lr
        pc<--lr
```

Application : codage de la fonction factorielle

```
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
```

Dans cet exemple comme il n'y a pas de variables locales on pourrait se passer de mettre en place fp pour la fonction appelée. On va tout de même l'utiliser de façon à adresser les paramètres toujours de la même façon. Voici la pile lors de l'exécution du corps de fact :




```
   .text
main:
   ldr r1, [fp, #-4]  @ n
   @ séquence d'appel de fact(n)
      @ empiler la valeur de n, paramètre donnée
      sub sp, sp, #4
      str r1, [sp]      @ valeur de n rangée dans la pile
      @ réserver la place pour le résultat de fact
      sub sp, sp, #4
      bl fact           @ appel
      ldr r1, [sp]      @ récupération du résultat dans r1
str r1, [fp, #-8] @ et leranger à sa place
      add sp, sp, #8    @ remise de la pile dans son état initial
...

fact: @ empiler lr car la fonction en appelle une autre
      sub sp, sp, #4
      str lr, [sp]
@ sauvegarde ancien fp et mise en place fp(fact)
sub sp, sp, #4
```

```
str fp, [sp]
mov fp, sp
      @ récupérer la valeur de x
      ldr r0, [fp, #+12] @r0=x
      cmp r0, #0
      bne sinon
alors:   @ ranger le résultat, qui est 1 dans ce cas
         mov r1, #1
         str r1, [fp, #+8]
      bal fsi
sinon:   @ r0 contient x
         @ appel de fact(x-1)
             @ ==> préparer la zone paramètre donnée et résultat de fonct
             sub r1, r0, #1  @ r1 = x-1
             sub sp, sp, #4
             str r1, [sp]    @ empiler x-1, param donnée
             sub sp, sp, #4  @ placepour résultat
             @ appel
             bl fact
             @ récupérer le résultat de l'appel fact(x-1) et remettre pile en état
             ldr r1, [sp]      @ r1=fact(x-1)
             ldr r0, [fp, #+12] @ r0=x
             add sp, sp, #8
             @ r0 * r1 = x * fact(x-1)
             multiplier r3, r0, r1 @ cette instruction existe en ARM
                                   @ mais elle n'est pas dans la doc
  @ résumée fournie
         @ ranger le résultat
         str r3, [fp, #+8]
fsi:
      @ recuperer fp
ldr fp, {sp]
add sp, sp, #4
      @ retour
      ldr lr, [sp]
      add sp, sp, #4
      mov pc, lr
```

Exercice : dérouler l'exécution avec n=3 en dessinant les différents état de la pile, c'est-à-dire la zone de mémoire repérée par `sp`.

Application à la 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;
   .... lecture d'un entier dans n
   y = fact(n);
   .... utilisation de la valeur de y
}
```

Pile lors de l'exécution du corps de factorielle juste après l'appel dans main :

```
      |-------------|
      |             |  case pour r
      |-------------|
      |             |  case pour loc
      |-------------|
fp--> | ancien fp   |
      |-------------|
      | adr retour  |
      |-------------|
      |             |  case pour le résultat
      |-------------|
      | param x     |
      |-------------|
```

Et un codage systématique de la nouvelle version de `fact` :

```
fact: @ empiler adr retour
         sub sp, sp, #4
         str lr, [sp]
      @ réserver place pour loc et r et mise en place de fp
   sub sp, sp, #4
str fp, [sp]
         sub sp, sp, #8
      @ if x==0 ...
         ldr r0, [fp, #+12]   @ r0=x
         cmp r0, #0
         beq sinon
alors:     mov r2, #1
           str r2, [fp, #-8]       @ r = 1
         bal finsi
sinon:   @ appel fact(x-1)
            @ preparer parametre et resultat
            sub sp, sp #4
            ldr r0, [fp, #+12]   @ r0=x
            sub r1, r0, #1       @ r1=x-1
            str r1, [sp]
            sub sp, sp #4
```

```
            bl fact

            @ recuperer resultat et place param
            ldr r0, [sp]      @ recuperer resultat
            add sp, sp, #8    @ recuperer place param et resultat
         @ apres l'appel
         str r0, [fp, #-4]          @ loc=fact(x-1)
         ldr r0, [fp, #+12]    @ r0=x
         ldr r1, [fp, #-4]          @ r1=loc
         multiplier r2, r1, r0 @ x*loc instruction n'existant pas sur ARM...
         str r2, [fp, #-8]      @ r=x*loc
finsi:
         @ return r
str r2, [fp, #+8]
      @ recuperer place var loc et fp
      add sp, sp, #8
ldr fp, [sp]
add sp, sp, #4
      @ depiler adr retour dans lr
      ldr lr, [sp]
      add sp, sp, #4
      @ retour
      mov pc, lr
```

6.4.3 Problème des temporaires :

Il faut sauvegarder les registres utilisés : `r0, r1, r2`... dans la pile. Et cela doit être fait avant de les utiliser donc en tout début du code de la fonction. On choisit de le faire de telle façon que cela ne remettre pas en cause les déplacements des variables locales déjà utilisés mais on pourrait faire autrement...

```
fact: @ empiler adr retour
         sub sp, sp, #4
         str lr, [sp]
      @ réserver place pour loc et r et mise en place de fp
   sub sp, sp, #4
str fp, [sp]
         sub sp, sp, #8
      @ sauvegarde de r0, r1, et r2 (empiler)
         sub sp, sp, #4
         str r0, [sp]
         sub sp, sp, #4
         str r1, [sp]
         sub sp, sp, #4
         str r2, [sp]
      @ if x==0 ...
.........................
.........................
      @ restaurer les registres r0, r1, r2 (depiler)
      ldr r2, [sp]
```

```
      add sp, sp, #4
      ldr r1, [sp]
      add sp, sp, #4
      ldr r0, [sp]
      add sp, sp, #4
      @ recuperer place var loc et fp
      add sp, sp, #8
ldr fp, [sp]
add sp, sp, #4
      @ depiler adr retour dans lr
      ldr lr, [sp]
      add sp, sp, #4
      @ retour
      mov pc, lr
```

6.5 Passage de paramètre par *adresse*

6.5.1 Vocabulaire, notations

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.

Il existe différentes façons de gérer le paramètre `z`. Nous n'en étudions qu'une seule ; la méthode dite du *passage par adresse* et nous utiliserons 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] désigne le contenu de la mémoire d'adresse z
   ...
```

L'appel à la fonction XX est alors par exemple :

```
a,b,c : entier

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

6.5.2 Programmation en langage d'assemblage

Paramètres et variables locales dans des registres

On prend les conventions suivantes : `x dans r0, y dans r1, adresse z dans r2`. Les variables `u` et `v` sont rangées dans `r3` et `r4` respectivement.

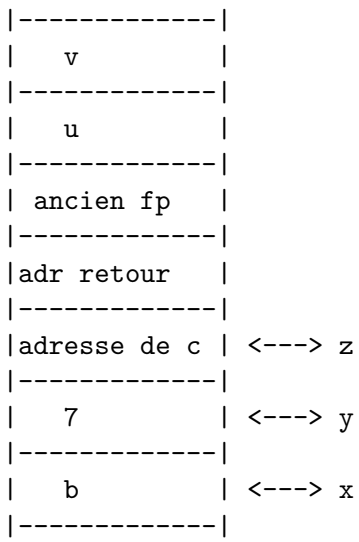
```
   .data
a: skip 4
b: skip 4
c: skip 4

   .text

main:
   ldr r0, ptr_a
   mov r1, #3
   str r1, [r0]
   ...
   ldr r0, ptr_b
   ldr r0, [r0]   @ r0= valeur de b
   mov r1, #7     @ r1= 7
   ldr r2, ptr_c  @ r2= adresse de c
   bl XX
   ...
XX:
   mov r3, r0      @ u=x
   add r4, r1, #2  @ v=y+2
   ...
   add r7, r3, r4
   str r7, [r2]    @ mem[z]=u+v
```

Paramètres et variables locales dans la pile

```
   .data
a: skip 4
b: skip 4
c: skip 4

   .text

main:
   ...
   ldr r0, ptr_b
```

```
   ldr r0, [r0]   @ r0= valeur de b
sub sp, sp, #4
str r0, [sp]   @ empiler b
   mov r0, #7     @ r1= 7
sub sp, sp, #4
str r0, [sp]   @ empiler 7
   ldr r0, ptr_c  @ r2= adresse de c
sub sp, sp, #4
str r0, [sp]   @ empiler adresse de c
   bl XX
   ...
```

Pile lors de l'exécution du corps de la procédure XX :




```
XX:
   ldr r0, [fp, #+16] @ u=x
   str r0, [fp, #-4]
   ldr r0, [fp, #+16] @ 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]
   str r0, [r2]    @ mem[z]=u+v
```

Chapitre 7

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

On illustre le propos avec un exemple de machine simpliste mais suffisant pour comprendre les principes des base et permettre de garder une taille raisonnable à la description du processeur. Référence : livre ALM fig 14.3 page 351.

7.1 Description du processeur vu du programmeur

Le processeur comporte un seul registre de données, directement visible par le programmeur, appelé `ACC` (pour accumulateur).

La taille nécessaire au codage d'une adresse ou d'une donnée est 1 mot. la taille d'un mot est 4 bits. D'où la taille d ela mémoire : 16 mots de 4 bits (adresses de 0000 à 1111... on ne rit pas, c'est juste pour comprendre).

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`.

Les instructions sont codées sur 1 ou 2 mots : 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 des instructions est le suivant :

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

7.2 Un exemple de programme

```
ld# 3
st [8]
```

```
et: add [8]
    jmp et
```

Que calcule ce programme ?

```
ACC <-- 3
mem[8] <-- ACC c'est-à-dire 3
ACC <-- mem[8] + ACC  c'est-à-dire 6
et recommencer
ACC <-- mem[8] + ACC c'est-à-dire 6+3 = 9
etc.
```

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		
9		

7.3 Description de l'exécution du programme = interprétation des différentes instructions

Le mot interprétation ou exécution est utilisé ici comme en musique : l'orchestre interprète un morceau, l'exécute. Le processeur interprète, exécute les instructions

7.3.1 Interpréter... l'exemple

Dans cet exemple, l'exécution du programme commence par l'interprétation de la première instruction, dont le code est en mémoire à l'adresse 0. Ce code étant celui de l'instruction `ld`, l'interprète lit une information supplémentaire dans le mot d'adresse 1. Cette valeur est alors chargée dans le registre `ACC`. Finalement, le compteur programme (`PC`) est modifié de façon à traiter l'instruction suivante.

7.3.2 Vision algorithmique de cette 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

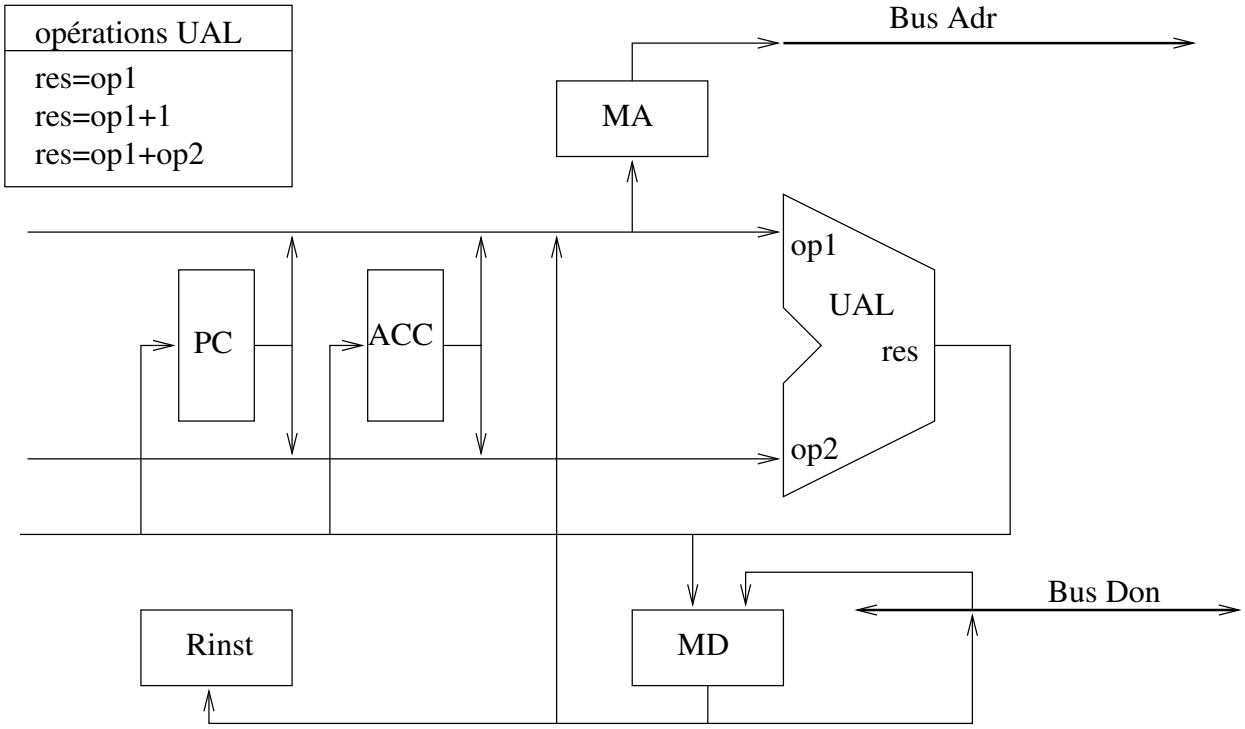


FIG. 7.1 – Partie Opérative

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

7.4 Organisation du processeur

Le processeur comporte une partie qui permet de stocker des informations dans des registres (visibles ou non du programmeur, de faire des calculs (+, -, and,... UAL). Cette partie est reliée à la mémoire par les bus adresses et données. On l'appelle *Partie Opérative*. Une possible est décrite dans la figure 7.1.

Selon l'organisation de cette partie opérative, un certain nombre d'actions de base sont possibles : on les appelle des micro-actions.

Dans la partie opérative on a des registres : `pc`, `acc`, `rinst`, `ma` (memory address), `md` (memory data),.. (plus des flags si on voulait ?).

La partie opérative donne aussi des infos qui servent à élaborer les conditions.

On fait des hypothèses FORTES sur les transferts possibles :

md ← mem[ma]	lecture d'un mot mémoire.	C'est la seule possibilité en lecture de la mémoire.
mem[ma] ← md	écriture d'un mot mémoire	
reg ← reg1 op reg2	opérations	un destinataire, deux (ou 1) sources

Hypothèses sur les tests : (rinst = entier)

Ces 3 types de transferts et les tests constituent le langage des micro-actions et des micro-conditions (analogie avec microZ de procesim...)

Le processeur comporte une partie qui permet d'enchainer des calculs et/ou des manipulations de registres et/ou des accès à la mémoire, c'est la *Partie Contrôle* aussi appelée algo d'interprétation des instructions du processeur. C'est une *machine séquentielle (automate) avec actions*. Cette machine décrit *Comment interpréter, en vrai ?*.

7.5 Automate d'interprétation des instructions du processeur

On parle aussi de contrôleur, de partie contrôle, de graphe de contrôle.

La machine algo avec actions est un automate.

Les entrées sont des conditions booléennes portant sur la valeur de `Rinst` (par ex Rinst = 7) ou sur la valeur d'un bit d'un registre (par ex bit 3 de MA = 1).

Les sorties sont des actions. Pour des raisons qu'on verra plus tard quand l'automate passe dans l'état, l'action est exécutée une fois.

Les états sont.. des états. L'un d'eux est initial.

Le graphe d'état s'appelle graphe de contrôle ou automate de contrôle

7.5.1 Une première version

Si on veut on peut sauter cette version mais je trouve que c'est plus clair de commencer par présenter la séquence des micro-actions nécessaires à l'interprétation de chaque instruction machine et à part la lecture du code op et des opérandes la mise en graphe, le fait d'incrémenter en avance le pc ne me semble pas évident quand on ne l'a jamais vu...

Sans trop optmiser sauf la partie lecture de l'instruction et lecture de l'opérande, on peut décrire le graphe de contrôle par la figure 7.2.

La notation de la condition `clr` doit être comprise comme le booléen `rinst = 1`. Mais c'est plus facile à lire.

L'automate est ensuite réalisé avec du matériel (=des transistors). Il existe une méthode systématique. On essaie de le rendre petit, performant.

- minimiser le nombre d'états : taille de l'automate,
- minimiser le temps pour exécuter une instruction. En effet, en gros 1 état = 1 période d'horloge (cycle du processeur). Nombre d'états traversés pour exécuter une instruction = temps exec. pour cette instruction.

Quelques pistes :

- `pc = pc +1` peut être fait en avance. Et `pc` repère alors le mot suivant...
- `ma = md` commun à plusieurs chemins

7.5.2 Version améliorée

Voir figure 7.3.

7.6 Déroulement d'un programme, au cycle près

On donne un (autre) exemple de programme en langage d'assemblage figure 7.4.

On donne/cherche ? le déroulement du programme étape par étape (Cf figure 7.5. Une étape correspond à un coup d'horloge, ou, naturellement au passage dans un état de l'automate de contrôle. Ca peut être un exercice, ou un travail dirigé...

Durée d'exécution du programme = 14 cycles + NN × 28
où NN est le nombre de fois où on laisse s'exécuter la boucle. En effet il n'y a pas de moyen de stopper ce programme : reset.. Sur la base d'une période d'horloge de 1 ns (fréquence 1 GigaHz) il peut y avoir $10^9/28$ boucles par seconde, soit environ 35 millions.

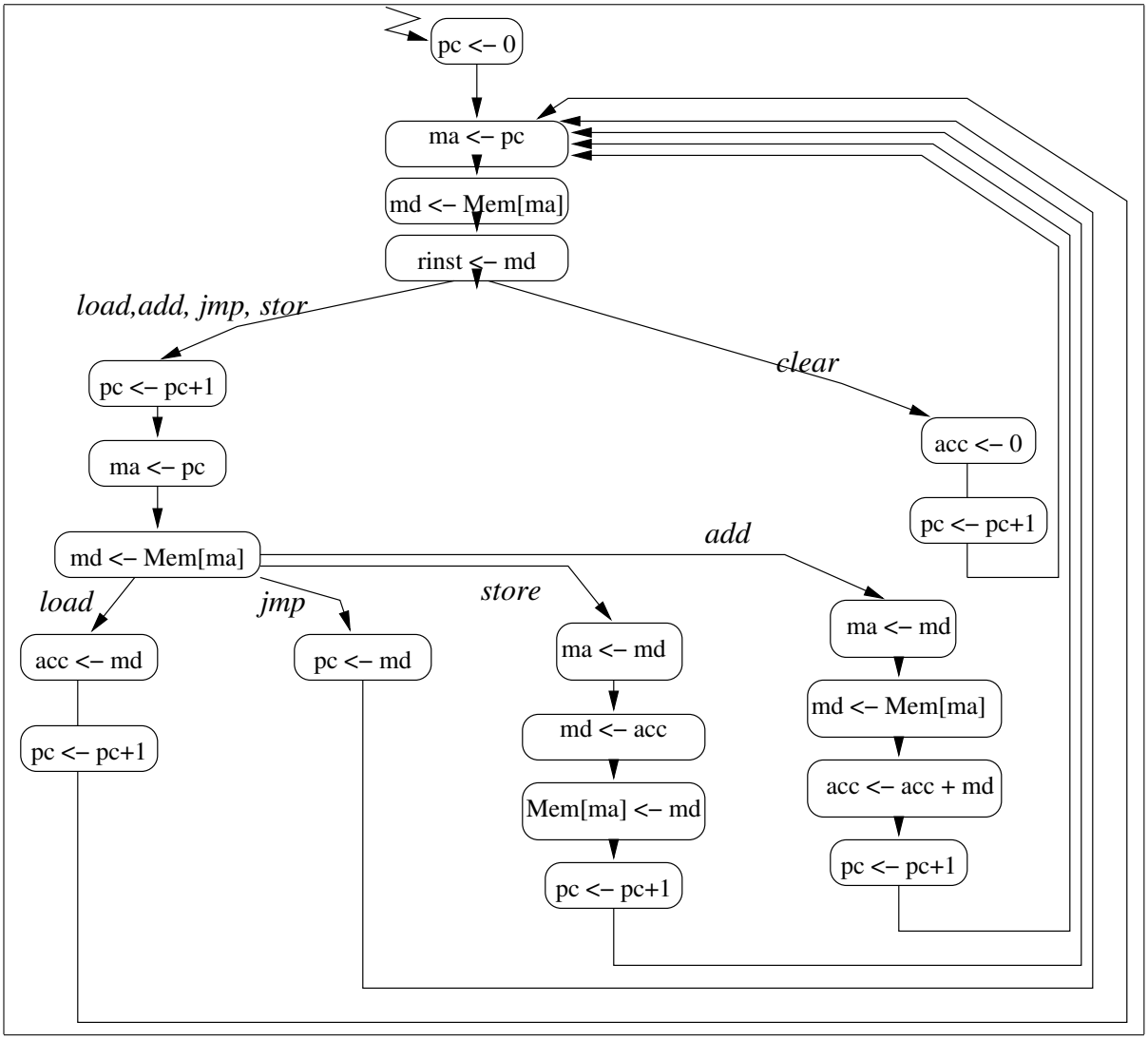


FIG. 7.2 – Une première version de graphe de contrôle

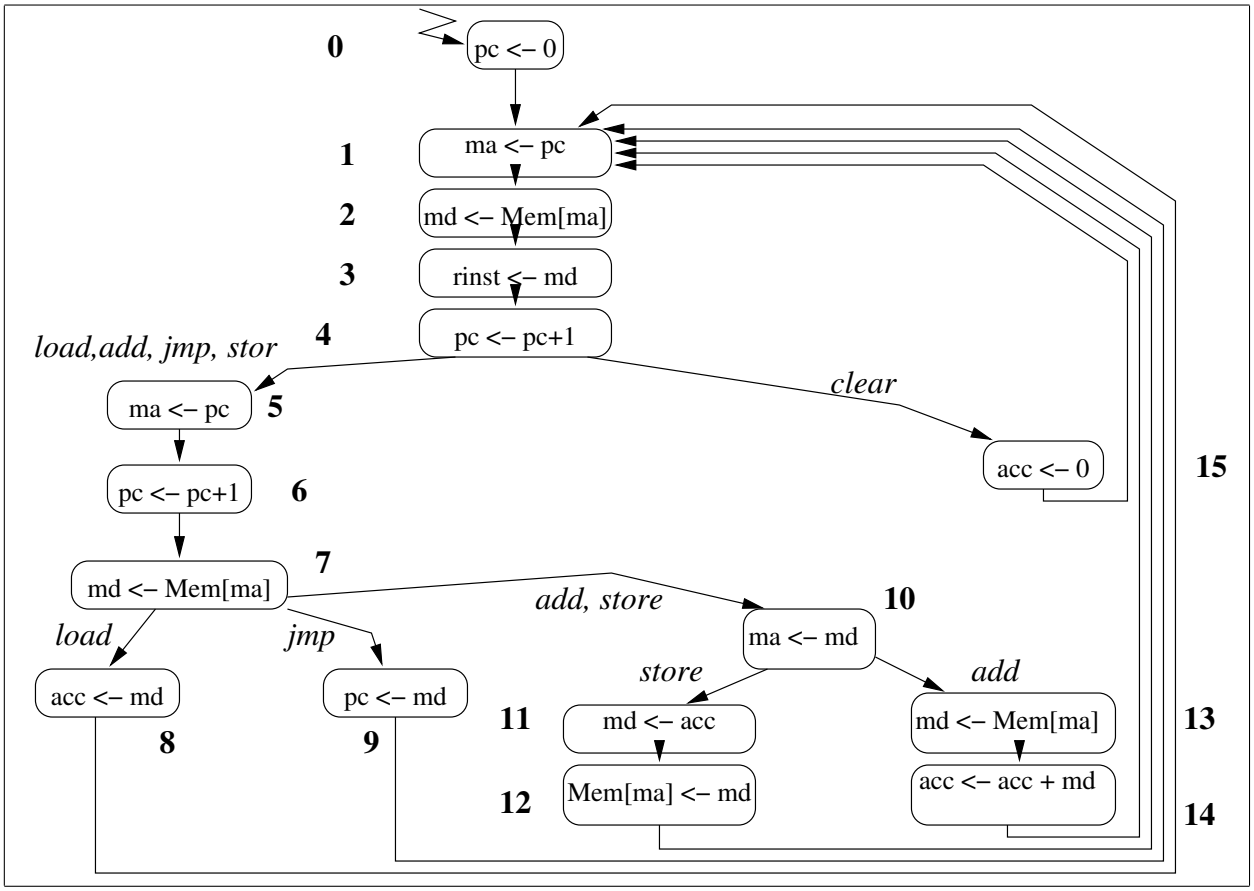


FIG. 7.3 – Graphe de contrôle. L'automate est de Moore. Les commandes associées aux états sont émises vers la partie opérative.

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

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	

FIG. 7.4 – Un programme en langage d'assemblage du jouet

		clr	
0	pc	←	0
1	ma	←	pc
2	md	←	Mem[ma = 0]
3	rinst	←	md
4	pc	←	pc + 1
15	acc	←	0
		load	
1	ma	←	pc
2	md	←	Mem[ma = 1]
3	rinst	←	md
4	pc	←	pc + 1
5	ma	←	pc
6	pc	←	pc + 1
7	md	←	Mem[ma = 2]
8	acc	←	md = 8
		store	
1	ma	←	pc
2	md	←	Mem[ma = 3]
3	rinst	←	md
4	pc	←	pc + 1
5	ma	←	pc
6	pc	←	pc + 1
7	md	←	Mem[ma = 4]
10	ma	←	md = 15
11	md	←	acc = 8
12	Mem[ma = 15]	←	md
		add	
1	ma	←	pc
2	md	←	Mem[ma = 5]
3	rinst	←	md
4	pc	←	pc + 1
5	ma	←	pc
6	pc	←	pc + 1
7	md	←	Mem[ma = 6]
10	ma	←	md
13	md	←	Mem[ma = 15]
14	acc	←	acc + md
		jump	
1	ma	←	pc
2	md	←	Mem[ma = 7]
3	rinst	←	md
4	pc	←	pc + 1
5	ma	←	pc
6	pc	←	pc + 1
7	md	←	Mem[ma = 8]
9	pc	←	md = 3
		store	
1	ma	←	pc

FIG. 7.5 – Déroulement du programme. Les mnémoniques ne sont là que comme points de repère. Les nombres de la colonne de gauche correspondent aux états du graphe de contrôle.

Chapitre 8

Vie des programmes

8.1 Etapes permettant de produire un exécutable

8.1.1 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]);
    }
    for (i=0;i<TAILLE;i++) {
       printf ("%d\n", TAB[i]);
    }
    u = somme (TAB, TAILLE);
 v = max (TAB, TAILLE);
 printf ("somme= %d, max= %d\n", u, v);
}
```

Un peu de vocabulaire. Dans le fichier `main.c` les fonctions `somme` et `max` sont dites **importées** : on peut les utiliser mais elles sont définies ailleurs. Le fait de déclarer l'en-tête des fonctions permet au compilateur de vérifier que leur utilisation est correcte. Dans le fichier `fonctions.c`, `somme` et `max` sont dites **exportées** : on a le droit de les utiliser ailleurs.

Pour "compiler", produire un exécutable, on enchaine les commandes :

```
gcc -c fonctions.c
gcc -c main.c
gcc -o exec main.o fonctions.o
```

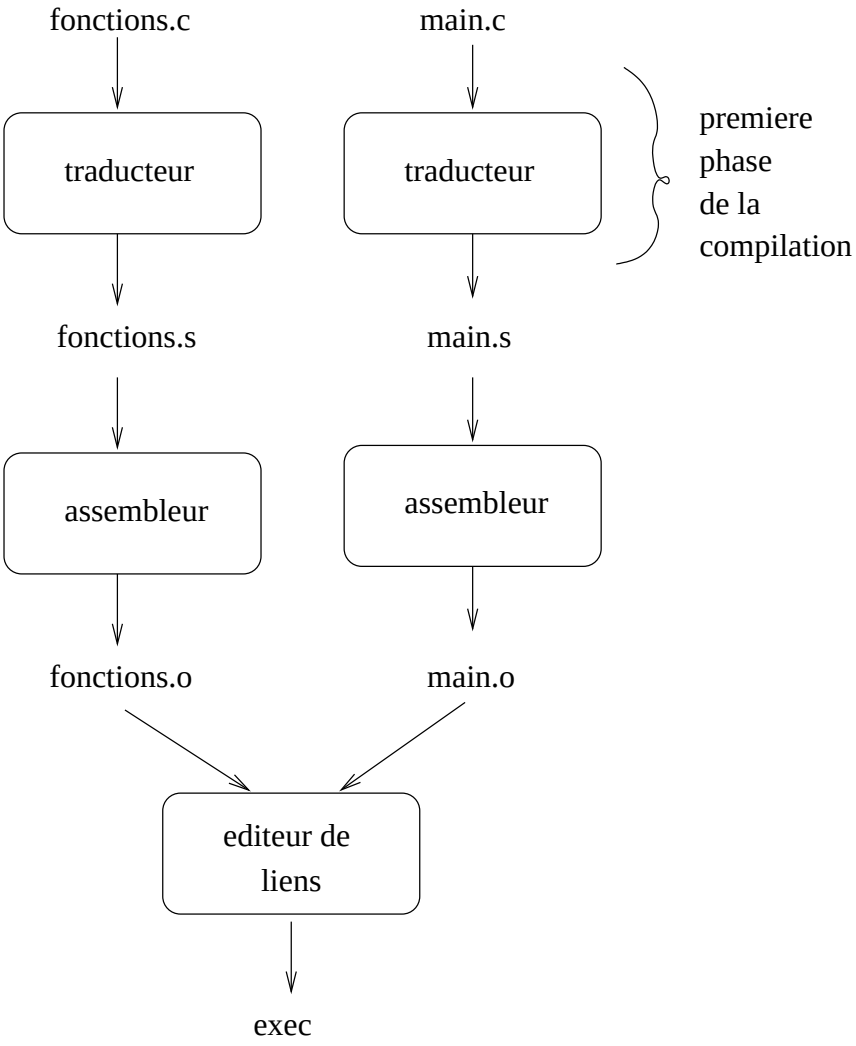


FIG. 8.1 – Compilation des fichiers `fonctions.c` et `main.c`

La commande `gcc -c main.c` a pour effet de produire un fichier appelé `main.o`. La commande `gcc -c fonctions.c` a pour effet de produire un fichier `fonctions.o`. Les fichiers `fonctions.o` et `main.o` contiennent du **binaire transtable**, c'est-à-dire du binaire qui ne peut être copié tel quel dans la mémoire.

La commande `gcc -o exec main.o fonctions.o` a pour effet de produire `exec` qui contient du `binaire exécutable`. Ce fichier résulte de la liaison des deux `.o`. On parle d'**édition de liens**.

En fait `gcc` cache l'appel à différents outils (logiciels).

8.1.2 Un exemple en langage d'assemblage

Cet exemple est pris dans le TP sur le codage des données en mémoire. Dans le fichier `accesmem.s`, la fonction `EcrHexa32` est importée, dans le fichier `es.s` cette fonction est exportée.

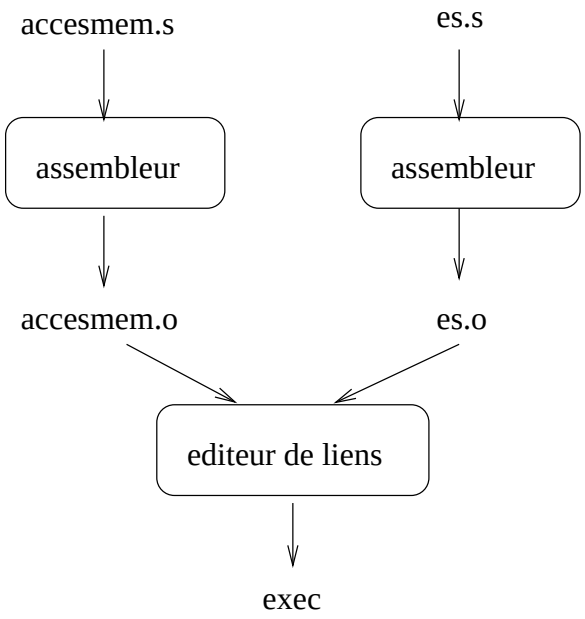


FIG. 8.2 – Compilation des fichiers `es.s` et `accesmem.s`

```
@ fichier accesmem.s

    .text

........

    MOV r1, r5
    BL EcrHexa32

........
```

```
@ fichier es.s

    .global EcrHexa32

.......

    .text

.......

EcrHexa32:
   mov ip, sp
   stmfd sp!, {r0, r1, r2, r3, fp, ip, lr, pc}
   sub     fp, ip, #4
........
   ldmea fp, {r0, r1, r2, r3, fp, sp, pc}

.......
```

Pour "compiler", produire un exécutable, on enchaine les commandes :

```
arm-elf-gcc -c es.s
arm-elf-gcc -c accesmem.s
arm-elf-gcc -o exec es.o accesmem.o
```

La commande `arm-elf-gcc -c es.s` a pour effet de produire un fichier appelé `es.o`. La commande `arm-elf-gcc -c accesmem.s` a pour effet de produire un fichier appelé `accesmem.o`. Les fichiers `es.o` et `accesmem.o` contiennent du **binaire transtable**. La commande `arm-elf-gcc -o exec accesmem.o es.o` a pour effet de produire `exec` qui contient du `binaire exécutable`.

La commande `arm-elf-gcc` utilisée avec l'option `-c` correspond à la commande `arm-elf-as` et utilisée avec `-o ...` correspond à la commande d'édition de liens (`arm-elf-ld`).

8.2 Traduction du langage C en langage d'assemblage

On n'en parle pas dans ce cours. L'idée est de faire faire par un programme de façon systématique ce que l'on fait à la main...

8.3 Assembleur

L'objectif de l'assembleur est de produire du 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 raison principalement :

1. On ne connaît pas en général l'adresse à laquelle les zones text et data doivent être tangées en mémoire.
2. Le programme peut faire référence à des noms qui ne sont pas définis dans le fichier en cours de traduction.

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

```
   .data
    .word 24
AA: .word 42
   .text
...
ldr r0, adrAA
ldr r0, [r0]
...
adrAA: .word AA
```

Exemples :

1. Dans l'exemple ci-dessous, l'adresse associée au symbole AA 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 AA est alors 4. Si on doit translater le programme à l'adresse 2000 par exemple, il faut se rappeler que à l'adresse adrAA on doit modifier la valeur 4 en 2000+4. Cette information à mémoriser est appelée une donnée de translation (relocation en anglais).
2. Dans l'exemple en langage d'assemblage du premier paragraphe, dans fichier `accesmem.o` il n'est pas possible de calculer le déplacement de l'instruction `BL EcrHexa32` puisque l'on ne sait pas où est l'étiquette `EcrHexa32` quand l'assembleur traite le fichier `accesmem.s`. En effet étiquette est dans un autre fichier `es.s`

   Dans l'exemple en langage C, on imagine que dans le fichier en langage d'assemblage `main.s` les références aux fonctions `somme` et `max` ne peuvent être complètes car les fonctions en question ne sont pas définies dans le fichier `main.c` mais dans `fonctions.c`.

8.3.1 Que contient un fichier .s ?

- des directives : `.data`, `.bss`, `.text`, `.word`, `.hword`, `.byte`, `.skip`, `.asciz`, `.align`
- des étiquettes appelés aussi symboles
- des instructions du processeur

Note : Parfois une directive (`.org`) permet de fixer l'adresse où sera logé le programme en mémoire. Cette facilité permet alors de calculer certaines adresse dès la phase d'assemblage. Nous avons utilisé cette facilité lors du TD4.

8.3.2 Que contient un fichier .o ?

Je simplifie... par exemple il ne me semble pas très utile de leur dire qu'il y a une table des chaines à laquelle la table des symboles fait référence, c'est une question de forme de mise en oeuvre.

- la zone de données, `data` (les octets) parfois incomplète
- la zone des instructions, `text` parfois incomplète
- la taille de la zone de données non initialisées, `bss`
- les informations associées à chaque symbole rangées dans une section appelée : table des symboles.
- des 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.

8.3.3 Etapes d'un assembleur

- reconnaissance de la syntaxe (lexicographie et syntaxe) et repérage des symboles. Fabrication de la table des symboles utilisée par la suite dès qu'une référence à un symbole apparait.
- traduction = production du binaire

Table des symboles : pour chaque symbole, son nom (chaine de caractères), sa valeur (déplacement par rapport au début du fichier), sa zone de définition (text, data ou bss) et sa portée (défini dans le fichier ou non, utilisable à l'extérieur ou non), symbole local, exporté, importé.

Par exemple, dans le fichier `es.s` le symbole `EcrHexa32` est défini dans la section text et peut être utilisé à l'extérieur (directive .global).

Dans le fichier `accesmem.s` le symbole `EcrHexa32` n'est pas défini, il est utilisé. On intuite bien que la traduction de l'instruction `bl EcrHexa32` ne va pas être possible car on devrait calculer le déplacement entre l'instruction `bl` et l'étiquette `EcrHexa32` qui n'est pas dans le même fichier.

L'information gardée dans `accesmem.o` est du style : il y a quelque chose à modifier à l'adresse ou il y a l'instruction `bl EcrHexa32`, et il faudra corriger avec les infos liées au symbole `EcrHexa32` en mettant le bon déplacement.

8.4 Editeur 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.

On peut reprendre l'exemple en langage C du début pour imaginer ce que l'on trouve dans les fichiers `.s`, ce que l'on ne sait pas traduire dans les fichiers `.o`. Il n'y a rien de fondamentalement différent de l'exemple en langage d'assemblage mnais ca me semble intéresant qu'il sache mettre quelque chose derrière le mot `extern`. De plus la convention C est : si je ne dis rien c'est exporté, pour dire importé il faut mettre extern alors qu'en assembleur la convention est si je ne dis rien c'est local (pas exporté), pour dire importé je ne dis rien...

On peut aussi parler de `static` en C : devant une fonction ca veut dire locale.

Nous avons décrit l'édition de liens comme le rassemblement de deux (ou plus) fichier de type `.o`. En fait la question peut être vue de façon plus générale. L'assembleur ne peut pas produire du binaire exécutable, il produit donc du binaire incomplet dans lequel il conserve des informations permettant

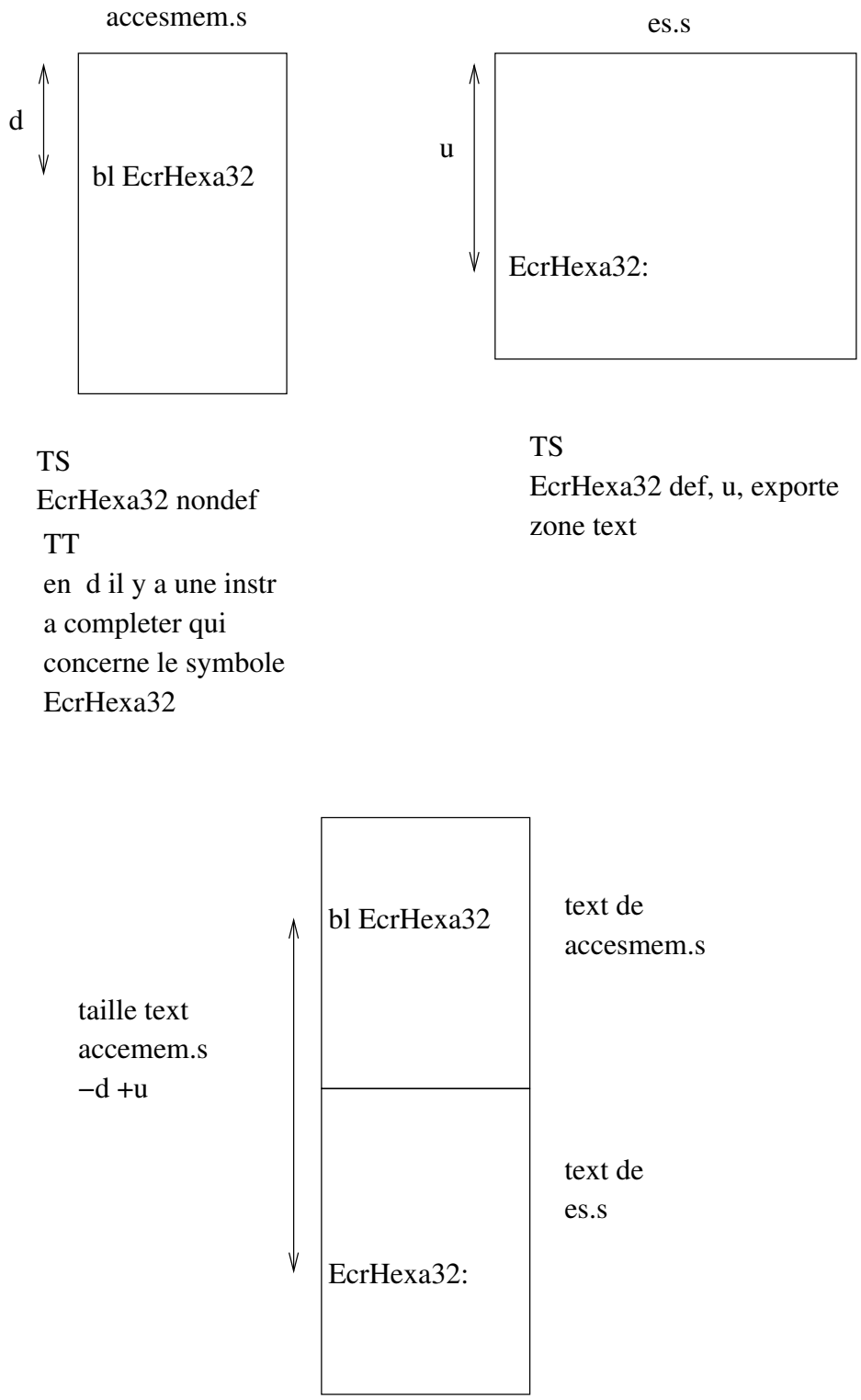


FIG. 8.3 – Contenu des fichiers `accesmem.o` et `es.o`. L'éditeur de liens rassemble les deux zones text et fait le cacul de la valeur du déplacement entre l'instruction bl EcrHexa32 et l'étiquette EcrHexa32. Pour cela, il est nécessaire de connaitre, dans accesmem.s pour traiter l'instruction à l'adresse d l'information liée à EcrHexa32 et dans le fichier es.s l'adresse liée à Ecrhexa32 qui est u.

de compléter celui-ci plus tard. Le “plus tard” peut avoir lieu de façon statique comme on l'a décrit précédemment mais peut aussi avoir lieu de façon dynamique au moment ou on en a besoin. Et là encore deux solutions sont 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 em mémoire plusieurs fois le même code.

8.5 Jusqu'où aller ? ? ? ? ?

Il me semble inutile à ce niveau de rentrer dans le détail du contenu des tables de symboles et de translation. Il suffit de dire que les informations dont on a besoin sont dans ces tables.

On peut prendre aussi un exemple avec des symboles de la zone data. En assembleur, c'est assez facile.

Faut-il prendre des exemples de données globales (partagées entre deux fichiers) en C ? ? ? ?

On peut expliquer que les variables globales sont rangées dans une zone data, les variables locales à une fonction sont dans la pile...

`static` devant une variable globale signifie qu'elle est locale au module (fichier), c'est-à-dire pas utilisable dans un autre fichier.

Par contre, il me semble qu'il ne faut pas parler du static associé à une variable locale à une fonction.

Chapitre 9

Organisation d'un ordinateur

1 Organisation de l'ordinateur

Il y a plusieurs sortes d'ordinateur... On trouve les consoles de jeu, les PC, les téléphones portables, les machines à laver... Ou des ordinateurs serveurs (thalès,..) On parle souvent de systèmes embarqués quand ça ne ressemble pas à un ordinateur. (contrôle de l'arrivée d'essence dans une voiture : pas de clavier, pas de souris, pas de disque, pas d'écran, mais c'est peut-être un processeur avec un programme)

1.0.1 Quelques critères pour classifier des ordinateurs de types différents

- peut-on ajouter des programmes et les lancer ? (cartouche de jeu, par exemple ..)
- même question : peut-on ajouter des programmes en LG machine, qu'on a écrits, compilés, assemblés soi-même ?
- existence de mémoire secondaire (disque dur, disquette, CD-Rom ? ?)

Il y a des points communs : on a toujours processeur, mémoire, bus adresses, données, signaux de contrôle (Read/$\overline{Write}$, autres),
ET DEUX choses nouvelles : décodage, contrôleurs ES

1.1 Notion de contrôleur d'Entrées-Sorties

1.1.1 Vision externe du contrôleur d'Entrées-Sorties

- un simple circuit avec 4 registres
- le circuit est vu par le processeur (espace d'adressage) comme 4 mots
== Mcommande (un mot de commande)
== Métat (un mot d'état)
== Mdonnéessort (données sortantes)
== Mdonnéesentr (données entrantes)
Quand il y a écriture (par une instruction STORE exécutée par le processeur) dans un des trois registres, Mcommande, Métat, Mdonnéessort ça écrit dans le registre du circuit contrôleur d'Entrées-Sorties ce que le processeur envoie sur le bus données.
Quand il y a lecture (par une instruction LOAD exécutée par le processeur) dans le registre Mdonnéesentr, ou dans le registre Métat, le processeur récupère le contenu du registre Mdonnéesentr (resp Métat).

Pour simplifier les 4 mots sont aux adresses respectives CNTRL, CNTRL+1, +2, +3. Mais attention CNTRL est une adresse constante, pas déterminée par l'assembleur. Elle a un sens physique qu'on va voir.
Notations Pour comprendre ce qui se passe, on écrira simplement
LOAD reg, Métat
pour signifier qqchose comme
METTRE l'adresse CNTRL dans regad
LOAD [Regad + 1], reg
de même, on écrira qqchose comme
STORE reg, Mdonnéessort
pour
STORE reg, [Regad + 2]

1.1.2 Utilisation du contrôleur d'Entrées-Sorties

Dans la documentation technique du contrôleur, certaines valeurs sont prédéfinies. Lire cette doc dans les cas réels.... Ici on écrira ces valeurs prédéfinies en italique. Exemples de valeurs prédéfinies pour

un contrôleur graphique (simpliste).
– *libre* : cette valeur dans le mot d'état signifie que le contrôleur est prêt à accepter une commande. (Le contrôleur est "libre", il n'est pas occupé...) Le contrôleur tient à jour son état de disponibilité pour que le processeur puisse savoir si il est occupé ou non.
– *fond_écran* : cette valeur, envoyée vers le mot de commande affiche un fond d'écran de la couleur contenue dans le registre de données sortantes ;
– *rouge* code de la coleur rouge, peut être contenu dans le registre de données sortantes ;
Exemple de programme simple d'action de sorties, affichage de la couleur du fond d'écran via un contrôleur graphique.

LOAD reg, Métat
CMP reg, *libre*
BNE ailleurs
STORE *rouge*, Mdonnéessort
STORE *fond_écran*, Mcommande
`pouf` c'est magique le fond de l'écran devient rouge ! !

On aurait de même des actions d'envoi d'un caractère, de positionnement de la tête de lecture d'un disque dur,.. Les contrôleurs réels sont parfois très simples, parfois très complexes (simple affichage comme ci-dessus, contrôleur réseau..) parfois pour une seule commande, plusieurs données...

1.1.3 Etude matérielle du contrôleur d'entrées-sorties

On voit les connexions entre le contrôleur, le processeur et le monde extérieur sur la figure 1. On reste avec l'exemple simple précédent.

ENTREES Il reçoit,
- bus données (lié au processeur)
- deux bits de bus adresses (pour sélectionner l'un des 4 mots CA SUFFIT) Ca revient à coder si l'écriture se fait a CNTRL +0, +1, +2 ou +3
- un signal de sélection provenant du décodeur d'adresses (On va y revenir)
- le signal Lire_ecrirebar du processeur,
- un paquet de données (8 fils) venant du monde extérieur. Disons pour simplifier 8 interrupteurs (plus une résistance de rappel !)
- 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.

— Dans la suite on ajoutera une entrée ACQUITTEMENT si c'est un contrôleur de sortie.

SORTIES Il délivre sur le bus données du processeur (via une sortie 3 états) 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 (via une sortie 3 états) 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, pourquoi pas)

— Dans la suite on ajoutera une sortie ENVOI si c'est un contrôleur de sortie.

1.2 Pourquoi et comment le décodage d'adresses ?

On va raisonner comme si les adresses étaient sur 8 bits au lieu de 32, c'est plus facile. Les adresses A sont codées sur 8 bits A_7,..,A_0. On représente les valeurs portées pas ces 8 fils par deux chiffres

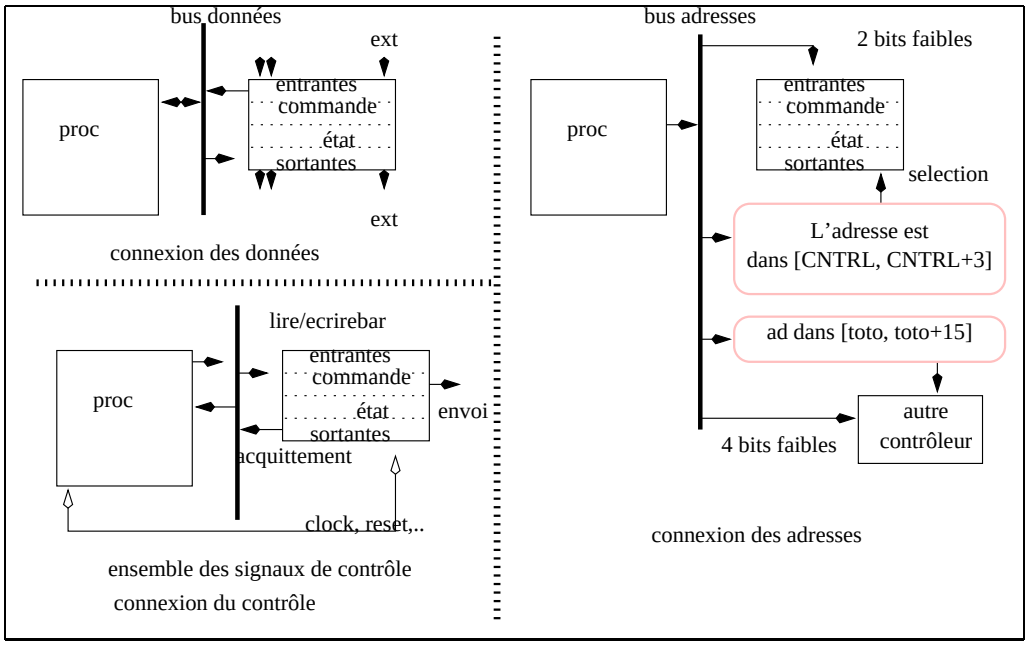


FIG. 1 – Connexions entre processeur, contrôleur d'entrées sorties et monde extérieur ; le dessin est en trois parties pour mettre en évidence les trois types d'informations véhiculées par les fils, mais bien sûr les fils sont tous présents dans l'assemblage.

hexadécimaux. On note val_x une valeur représentée en hexa. On dira, pour simplifier, que l'entrée A "vaut" 67_x pour dire que les 8 fils sont dans les états 0110 0111.

L'ordinateur est organisé ainsi : (Voir figure 2

1. Le mot (lu ou écrit) est pris dans une mémoire morte (ROM) ssi l'entrée A est telle que 0_x $<= A <=$ 3F_x
   c'est à dire A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 sont compris entre
   0 0 0 0 0 0 0 0 et
   0 0 1 1 1 1 1 1
   Ceci couvre les 64 monômes canoniques pour lesquels A_7=0 et A_6=0.
   Le boîtier physique de ROM, qui contient 64 mots reçoit deux types d'information
   – les 6 fils A_5 A_4 A_3 A_2 A_1 A_0 qui permettent d'adresser un des 64 mots
   – UN FIL de sélection `selrom` qui vaut 1 ssi A_7=0 et A_6=0. On voit que `selrom`=A_7'.A_6'.
2. Le mot (lu ou écrit) est pris dans le contrôleur d'entrées sorties (vu précédemment) ssi l'entrée A est telle que 58_x $<= A <=$ 5B_x
   (On a choisi que CNTRL = 58_x)
   c'est à dire ssi A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 sont compris entre
   0 1 0 1 1 0 0 0 et
   0 1 0 1 1 0 1 1
   Ceci couvre les 4 monômes canoniques pour lesquels A_7=0, A_6=1, A_5=0, A_4=1, A_3=1, A_2=0.
   Le boîtier physique de contrôleur, qui contient 4 mots, reçoit deux types d'information
   - les 2 fils A_1 A_0 qui permettent d'adresser un des 4 mots
   – UN FIL de sélection `selcntr` qui vaut 1 ssi A_7=0, A_6=1, A_5=0, A_4=1, A_3=1, A_2=0.
   On voit que `selcntr`=A_7'.A_6.A_5'.A_4.A_3.A_2'
3. Le mot (lu ou écrit) est pris dans le boîtier de mémoire vive ssi l'entrée A est telle que 80_x $<= A <=$ FF_x c'est à dire ssi A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 sont compris entre
   1 0 0 0 0 0 0 0 et

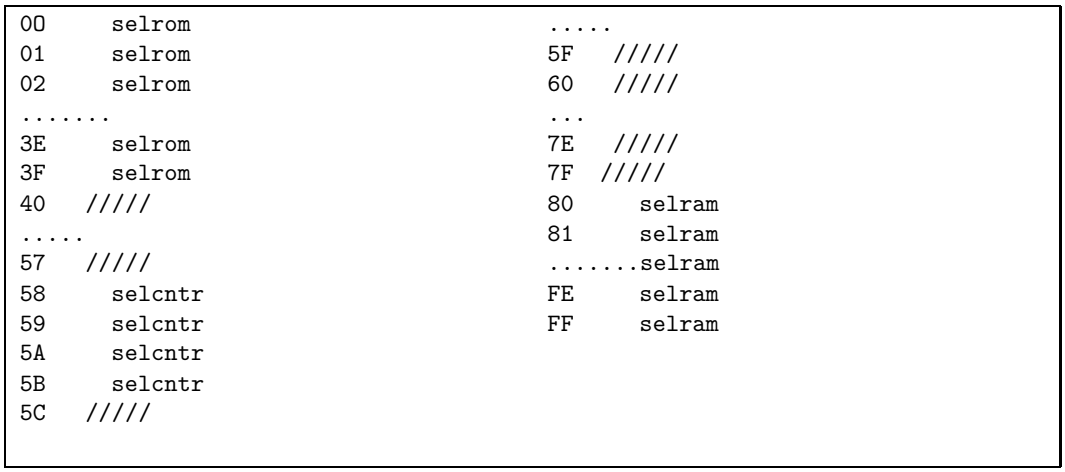


FIG. 2 – Organisation de l'espace d'adressage en ROM, contrôle d'entrées sorties et RAM

1 1 1 1 1 1 1 1
Ceci couvre les 128 monômes canoniques pour lesquels A_7=1.
Le boîtier physique de RAM, qui contient 128 mots, reçoit deux types d'information
- les 7 fils A_6 A_5 A_4 A_3 A_2 A_1 A_0 qui permettent d'adresser un des 128 mots
– UN FIL de sélection `selram` qui vaut 1 ssi A_7=1.
On voit que `selram`=A_7.

Circuit décodeur d'adresses

Un circuit C reçoit 8 fils d'entrées A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 Le circuit C a 4 sorties booléennes : `selram`, `selcntr`, `selrom`, `adresseinvalide` (si l'adresse ne "tombe" ni en RAM ni en ROM ni dans le contrôleur d'écran, elle est invalide) Le cablage global est donné figure 3. Il nous reste une question *bf angoissante* : Si l'adresse est invalide que se passe-t-il ?

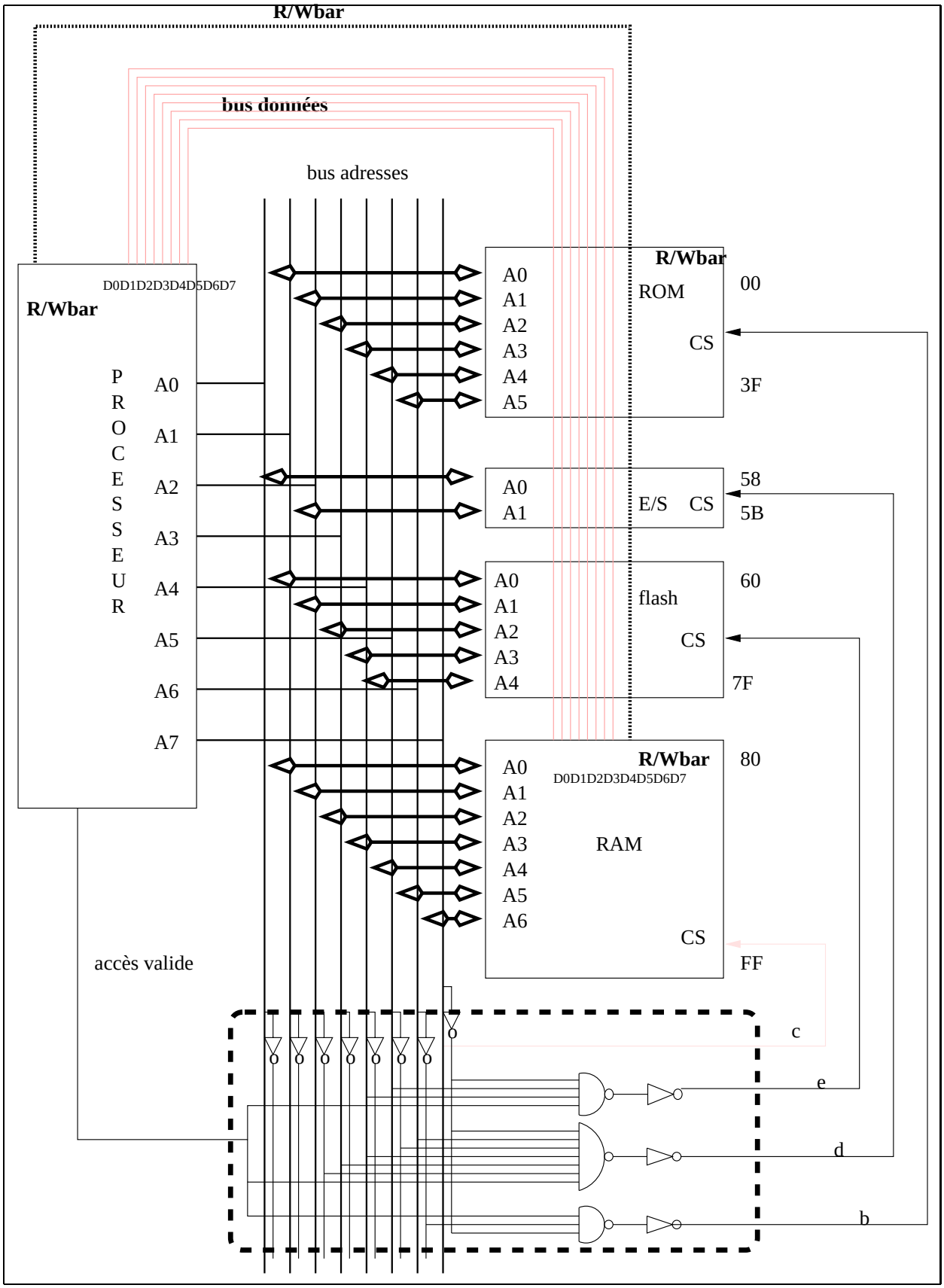


FIG. 3 – Connexions entre le processeur, le contrôleur d'entrées sorties, les deux mémoires et le décodeur d'adresses.