Architectures des ordinateurs (une introduction)
Année 1, l'exécution des programmes en langage machine.
(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Bibliographie

- *Architectures logicielles et matérielles*, Amblard, Fernandez, Lagnier, Maraninchi, Sicard, Waille, Dunod 2000

- *Architecture des ordinateurs*, Cazes, Delacroix, Dunod 2003.
- *Computer Organization and Design : The Hardware/Software Interface*, Patterson and Hennessy, Dunod 2003.
- *Processeurs ARM*, Jorda. DUNOD 2010.

- `https://im2ag-moodle.univ-grenoble-alpes.fr/course/view.php?id=336`

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

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

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau   Fabienne Carrier   Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Description du modèle de Von Neumann (2/3)

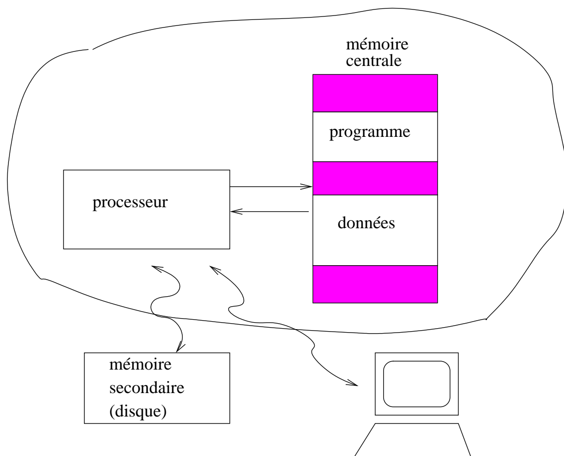


FIGURE – Processeur, mémoire et périphériques

Mémoire centrale (vision abstraite)

La mémoire contient des informations prises dans un certain domaine

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

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

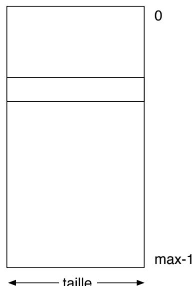


FIGURE – Mémoire abstraite

Résumé : processeur/mémoire

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

La mémoire contient des informations de nature différentes :

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

Le processeur, relié à une mémoire, peut :

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

Entrées/Sorties : définitions

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

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

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

Sortie : ordinateur $\longrightarrow$ extérieur

Entrée : extérieur $\longrightarrow$ ordinateur

Entrée/Sortie : ordinateur $\longleftrightarrow$ extérieur

Les bus

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

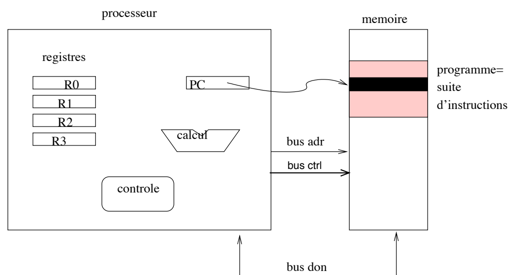
- Le bus de données permet la circulation des données.
- Le bus d'adresse permet au processeur de **désigner à chaque instant la case mémoire ou le périphérique** auquel il veut faire appel.
- Le bus de contrôle indique quelle est l'**opération que le processeur veut exécuter**, par exemple, s'il veut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

  On trouve également, dans le bus de contrôle, une ou plusieurs lignes qui permettent aux périphériques d'effectuer des demandes au processeur ; ces lignes sont appelées lignes d'interruptions matérielles (IRQ).

Composition du processeur

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

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



Codage des instructions : langage machine

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

Codage des informations et représentation des nombres par des vecteurs binaires

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

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau   Fabienne Carrier   Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

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

ASCII = ≪ American Standard Code for Information Interchange ≫

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

32	␣	33	!	34	”	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	ˆ	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.

Conclusion sur le codage : Où est le code ?

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

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

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

Quelques valeurs à connaître

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

Conversion base 10 vers base 2 : Troisième méthode

169	1	$(169 = 84 \times 2 + 1)$
84	0	$(84 = 42 \times 2 + 0)$
42	0	$(42 = 21 \times 2 + 0)$
21	1	
10	0	
5	1	
2	0	
1	1	
0		

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

Conversion base 2 vers base 10

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

En utilisant les puissances de 2 :

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

$a_{n-1}a_{n-2}\ldots a_1a_0$ vaut $a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_12^1 + a_02^0$ en base 10

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

Représentation des relatifs, solution : Complément à deux

Sur n bits, en choisissant 00...000 pour le codage de zéro, il reste $2^n - 1$ possibilités de codage : la moitié pour les positifs, la moitié pour les négatifs.

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

Principe :

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

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

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

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

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

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

$a_{n-1}a_{n-2}...a_1a_0$

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

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

Indicateurs

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

2 autres indicateurs (flags) :

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

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

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

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

Table d'addition (3 bits, naturels)

Récapitulatif : Pour 3 bits et les entiers naturels :

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

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

Table d'addition (3 bits, relatifs)

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

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

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

Langage d'assemblage, langage machine

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

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Etapes de compilation

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

Instruction de calcul entre des informations mémorisées

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

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

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

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

Instruction de rupture de séquence

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

Désignation des objets (1/7)

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

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

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

Désignation des objets (2/7) : par registre

Désignation registre/registre.

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

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

Désignation des objets (3/7) : immédiate

Désignation registre/valeur immédiate.

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

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

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

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

Désignation registre/indirect par registre

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

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

Séparation données/instructions

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

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

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

Etiquettes (1/4) : définition

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

Une étiquette correspond à une adresse.

Pourquoi ?

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

Programmation des structures de contrôles

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

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau   Fabienne Carrier   Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

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

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

A chaque cycle :

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

Les instructions sont exécutées séquentiellement
sauf ruptures de séquence !

Séquencement (2/7)

Séquencement « normal »

Après chaque instruction le registre *PC* est incrémenté.

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

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

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

Séquencement (3/7)

Rupture inconditionnelle

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

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

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

Séquencement (4/7)

Rupture conditionnelle

Si une condition est vérifiée, **alors**
PC est modifié
sinon
PC est incrémenté normalement.

la condition est interne au processeur :
expression booléenne portant sur les *codes de conditions arithmétiques*

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

Codage des structures de contrôle : exemples traités

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

```
    a<b : I1;
    a=b : I2;
    a>b : I3;
```

Instruction *Si alors sinon* : Une solution

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

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

Instruction *Tant que* : Une première solution

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

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

Construction *selon*

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

Une solution consiste à traduire en si alors sinon.

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

ARM offre (ou offrait) une autre possibilité…

Programmation des appels et retours de procédures simples

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

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Quel est le problème ?

Appel = branchement
instruction de rupture de séquence inconditionnelle (`B`) ?

MAIS Comment revenir ensuite ?

Le problème du retour : comment à la fin de l'exécution du corps de la fonction, indiquer au processeur l'adresse à laquelle il doit se brancher ?
Point de vigilance : garantir le bon usage des registres.

Adresse de retour

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

Cette adresse est appelée adresse de retour.

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

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

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

Où est gardée cette adresse ?

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

`BL` signifie *branch and link*

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

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

Conclusion

Conclusions : Il est possible d'avoir un ensemble d'instructions géré comme un bloc indépendant sous certaines conditions très limitatives (un seul appel `bl ma_proc`, convention commune à l'appel, si main==appel, retour `bx lr`, ...), pour s'affranchir de ces conditions :

- Paramètres : il faut une zone de stockage dynamique **commune** à l'*appelant* et à l'*appelé*. L'appelant y range les valeurs **avant** l'appel et l'appelé y prend ces valeurs et les utilise
- Variables locales : il faut une zone de mémoire dynamique **privée** pour chaque procédure pour y stocker ses variables locales : il ne faut pas que cette zone interfère les variables globales ou locales à l'appelant
- Variables temporaires : elles ne doivent pas interférer avec les autres variables
- Généralisation : il faut que la méthode choisie soit généralisable afin de pouvoir générer du code

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

Programmation de procédures (suite)
Utilisation de la pile

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

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Mécanisme de pile

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

Deux opérations possibles :

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

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

Comment réaliser une pile ? (1 /4)

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

Comment réaliser une pile ? (2 /4)

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

sens évolution	croissant	croissant	décroissant	décroissant
repère	1 ^{er} vide	der^{er} plein	1 ^{er} vide	der^{er} plein
empiler reg	Mem[sp]←reg sp←sp+1	sp←sp+1 Mem[sp]←reg	Mem[sp]←reg sp←sp-1	sp ←sp-1 Mem[sp]←reg
dépiler reg	sp←sp-1 reg←Mem[sp]	reg←Mem[sp] sp←sp-1	sp←sp+1 reg←Mem[sp]	reg←Mem[sp] sp←sp+1

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

Appel/retour : solution utilisée avec le processeur ARM

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

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

si nécessaire ...

Programmation des appels de procédure et fonction (fin)

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

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

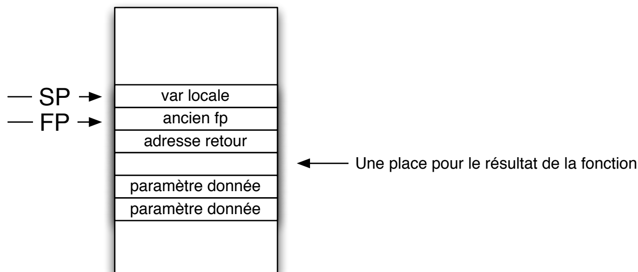
Université Grenoble Alpes

13 janvier 2021

Résultat dans la pile (3/3)

Lors de l'exécution du corps de la fonction.

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



Variables temporaires

Problème :

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

Solution :

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

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

appelant `P` :

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

appelée `Q` :

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

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

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

(comprendre pour programmer efficacement et sans bug)

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

13 janvier 2021

Les instructions

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

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

Codage des instructions

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

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

Le codage est le suivant :

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

Algorithme d'interprétation

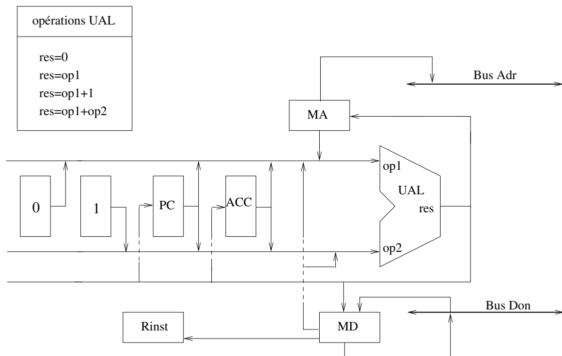
En adoptant un point de vue fonctionnel, en considérant les ressources du processeur comme les variables d'un programme, l'algorithme d'interprétation des instructions peut être décrit de la façon suivante :

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

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

Partie opérative

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



Micro-actions et micro-conditions

On fait des hypothèses FORTES sur les transferts possibles :

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

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

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

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