

Assembleur

Rappels d'architecture

Un ordinateur se compose principalement

- d'un processeur,
- de mémoire.

On y attache ensuite des périphériques, mais ils sont optionnels.

données : disque dur, etc

entrée utilisateur : clavier, souris

sortie utilisateur : écran, imprimante

processeur supplémentaire : GPU

Le processeur

- Le processeur lit et écrit des informations en **mémoire**
- Il peut de plus effectuer des **opérations** arithmétiques et logiques
- Chaque action qu'il peut effectuer est appelée **instruction**
- Les instructions effectuées par le processeur sont stockées dans la mémoire.
- Il dispose d'un petit nombre d'emplacements mémoire d'accès plus rapide, les **registres**.
- Un registre spécial nommé pc (program counter) (ou ip (instruction counter)) contient l'adresse de la prochaine instruction à exécuter
- De façon répétée le processeur :
 - 1 lit l'instruction stockée à l'adresse contenue dans pc
 - 2 l'interprète ce qui peut modifier certains registres (dont pc) et la mémoire

CISC / RISC

C'est principalement le jeu d'instruction qui distingue les processeurs

- Les processeurs CISC (Complex Instruction Set Computer)
 - Nombre d'instruction élevé
 - Les instructions réalisent souvent les transferts vers et depuis la mémoire
 - peu de registres
 - Exemples : Intel 8068, Motorola 68000
- Les processeurs RISC (Reduced Instruction Set Computer)
 - Peu d'instructions
 - Les instructions opèrent sur des registres
 - Registres nombreux
 - Exemples : Alpha, Sparc, MIPS, PowerPC

X86

- X86 est un jeu d'instruction commun à plusieurs processeurs
- Le nom X86 provient des processeurs Intel utilisant ce jeu d'instructions (8086, 80186, 80286, 80386, 80486)
- jeu d'instruction des processeurs équipant les ordinateurs personnels

Registres

- Les processeurs X86 (à partir du 386) ont huit registres de quatre octets chacun

eax	ax	ah	al
ebx	bx	bh	bl
ecx	cx	ch	cl
edx	dx	dh	dl
esi			
edi			
esp			
ebp			

- les registres `eax`, `ebx`, `ecx` et `edx` peuvent être découpés en registres plus petits
- `eax` peut être découpé en trois registres : un registre de deux octets : `ax` et deux registres d'un octet : `ah` et `al`.
- `esp` pointe sur le sommet de la pile
- `ebp` pointe sur l'adresse de base de l'espace global

Segmentation de la mémoire

- La mémoire est divisée en segments indépendants.
- L'adresse de début de chaque segment est stockée dans un registre.
- Chaque segment contient un type particulier de données.
- On s'intéressera plus particulièrement à trois segments :
 - le **segment de données** où sont stockés les variables globales et les constantes. La taille de ce segment n'évolue pas au cours de l'exécution du programme (il est statique).
 - le **segment de code** où sont stockées les instructions qui composent le programme
 - la **pile** où sont stockés les variables locales, paramètres de fonctions et certains résultats intermédiaires de calcul
- L'organisation de la mémoire en segments est conventionnelle
- En théorie tous les segments sont accessibles de la même manière

Registres liés aux segments

■ Segment de code

- **cs** (Code Segment) adresse de début du segment de code
- **eip** (Instruction Pointer) adresse relative de la prochaine instruction à effectuer
 - cs + eip** est l'adresse absolue de la prochaine instruction à effectuer

■ Segment de données

- **ds** (Data Segment) adresse de début du segment de données

■ Pile

- **ss** (Stack Segment) adresse de la base de la pile
- **esp** (Stack Pointer) adresse relative du sommet de pile
 - ss + esp** est l'adresse absolue du sommet de pile
- **ebp** (Base Pointer) registre utilisé pour le calcul d'adresses de variables locales et de paramètres

Segmentation de la mémoire

stack segment ss →	pile	paramètres de fonctions et variables locales
base pointer ebp →		
stack pointer esp →		
	espace non alloué	
	tas	objets alloués dynamiquement
	données	variables globales et constantes
data segment ds →	code	instructions
instr. pointer eip →		
code segment cs →		

Flags

- Les flags sont des variables booléennes (stockées sur un bit) qui donnent des informations sur le déroulement d'une opération et sur l'état du processeur.
- 32 flags sont définis, ils sont stockés dans le registre `eflags`, appelé registre d'état.
- Valeur de quelques flags après une opération :
 - **CF** : Carry Flag.
Indique une retenue (**CF**=1) sur les entiers non signés.
 - **PF** : Parity Flag.
Indique que le résultat est pair (**PF**=1) ou impair (**PF**=0).
 - **ZF** : Zero Flag.
Indique si le résultat est nul (**ZF**=1) ou non nul (**ZF**=0).
 - **SF** : Sign Flag.
Indique si le résultat est positif (**SF**=0) ou négatif (**SF**=1).
 - **OF** : Overflow Flag.
Indique un débordement (**OF**=1) sur les entiers signés.

Langage machine

Une instruction de langage machine correspond à une instruction possible du processeur.

Elle contient :

- un code correspondant à opération à réaliser,
- les arguments de l'opération : valeurs directes, numéros de registres, adresses mémoire.

code op	opérandes
---------	-----------

Langage machine

Si on ouvre un fichier exécutable avec un éditeur (hexadécimal), on obtient

...

```
01ebe814063727473747566662e6305f5f43544f525f4c
5f05f5f44544f525f4c4953545f5f05f5f4a43525f4c49
53545f5f05f5f646f5f676c6f62616c5f64746f72735f6
75780636f6d706c657465642e36353331064746f725f69
```

...

Langage machine lisible

Si on ouvre un fichier exécutable avec un éditeur (hexadécimal), on obtient

...

```
01ebe814063727473747566662e6305f5f43544f525f4c
5f05f5f44544f525f4c4953545f5f05f5f4a43525f4c49
53545f5f05f5f646f5f676c6f62616c5f64746f72735f6
75780636f6d706c657465642e36353331064746f725f69
```

...

C'est une suite d'instructions comme 01ebe814, que l'on peut traduire directement de façon plus lisible :

```
mov     eax, ebx
```

C'est ce qu'on appelle l'*assembleur*.

- L'assembleur est donc une *représentation* du langage machine.
- Il y a autant d'assembleurs que de type de processeurs différents.

NASM : Netwide Assembler

- Langage assembleur pour processeurs 80x86 et x86-64
- conçu pour la portabilité et la modularité
- permet de générer de nombreux formats exécutables (Linux, BSD, ELF ...)
- syntaxe intel (`instr dest,source`) simplifiée
- système de macros et pré compilateur

NASM : exemple

```
section .data
const      dw      123

section     .bss
var        resw     1

section     .text
global _start
_start:
    call    main
    mov     eax, 1
    int     0x80
main:
    push    ebp
    mov     ebp, esp
    mov     word [var], const
    pop     ebp
    ret
```

Sections

Un programme NASM est composé de trois sections :

- `.data`
Déclaration de constantes (leur valeur ne changera pas durant l'exécution)
- `.bss`
Déclaration de variables
- `.text`
Instructions qui composent le programme

La section data

- La section data permet de définir des constantes
- Elle commence par

```
section .data
```

- Elle est constituée de lignes de la forme
etiquette pseudo-instruction valeur
- Les pseudo instructions sont les suivantes :

db	define byte	déclare un octet
dw	define word	déclare deux octets
dd	define doubleword	déclare quatre octets
dq	define quadword	déclare huit octets
dt	define tenbytes	déclare dix octets

- Exemples :

```
const db 1  
const dw 123
```

- les variables déclarées en séquence sont disposées les unes à côté des autres en mémoire

La section bss

- La section bss permet de définir des variables
- Elle commence par

```
section .bss
```

- Elle est constituée de lignes de la forme
etiquette pseudo-instruction nb
- Les pseudo instructions sont les suivantes :

resb	reserve byte	déclare un octet
resw	reserve word	déclare deux octets
resd	reserve doubleword	déclare quatre octets
resq	reserve quadword	déclare huit octets
rest	reserve tenbytes	déclare dix octets

- nb représente le nombre d'octets (pour resb) de mots (pour resw)
...à réserver
- Exemples :

```
buffer      resb    64    ; reserve 64 octets
wordvar     resw    1     ; reserve un mot (deux octets)
realarray   resq    10    ; reserve 10 * 8 octets
```

La section text

- La section text contient les instructions correspondant au programme
- Elle commence par

```
section .text
```
- Elle est constituée de lignes de la forme

```
[étiquette] nom_d_instruction [opérandes]
```

les parties entre crochets sont optionnelles
- une étiquette correspond à une adresse (l'adresse dans laquelle est stockée l'instruction)
- une opérande peut être :
 - un registre,
 - une adresse mémoire,
 - une constante
 - une expression

Accès à la mémoire

- Si `adr` est une adresse mémoire, alors `[adr]` représente le contenu de l'adresse `adr`
- C'est comme l'opérateur de déréférencement `*` du langage C
- La taille de l'objet référencé peut être spécifié si nécessaire
 - `byte [adr]` un octet
 - `word [adr]` deux octets
 - `dword [adr]` quatre octets
- `adr` peut être :
 - une constante `[123]`
 - une étiquette `[var]`
 - un registre `[eax]`
 - une expression `[2*eax + var + 1]`

constantes

- t_t

Instructions

- instructions de transfert : registres $\leftrightarrow$ mémoire
 - Copie : mov
 - Gestion de la pile : push, pop
- instructions de calcul
 - Arithmétique : add, sub, mul, div
 - Logique : and, or
 - Comparaison : cmp
- instructions de saut
 - sauts inconditionnels : jmp
 - sauts conditionnels : je, jne, jg, jl
 - appel et retour de procédure : call, ret
- appels système

Copie - mov

- Syntaxe :

`mov destination source`

- Copie `source` vers `destination`

- `source` : un registre, une adresse ou une constante

- `destination` : un registre ou une adresse

- Les copies registre - registre sont possibles, mais pas les copies mémoire -mémoire

- Exemples :

```
mov eax, ebx           ; reg reg
mov eax, [var]          ; reg mem
mov ebx, 12             ; reg constante
mov [var], eax          ; mem reg
mov [var], 1            ; mem constante
```

Nombre d'octets copiés

- Lorsqu'on copie vers un registre ou depuis un registre , c'est la taille du registre qui indique le nombre d'octets copiés
- lorsqu'on copie une constante en mémoire, il faut préciser le nombre d'octets à copier, à l'aide des mots clefs
 - byte un octet
 - word deux octets
 - dword quatre octets
- Exemples :

```
mov eax, ebx           ; reg reg
mov eax, [var]         ; reg mem
mov ebx, 12            ; reg constante
mov [var], eax         ; mem reg
mov word [var], 1      ; mem constante
```

Empile – push

- Syntaxe :

push source

- Copie le contenu de source au sommet de la pile.
- Commence par décrémenter esp de 4 puis effectue la copie
- source : adresse, constante ou registre
- Exemples

```
push 1      ; empile la constante 1
push eax    ; empile le contenu de eax
push [var]   ; empile la valeur se trouvant
              ; a l'adresse var
```

Dépile – pop

- Syntaxe :

`pop destination`

- Copie les 4 octets qui se trouvent au sommet de la pile dans `destination`.
- Commence par effectuer la copie puis incrémente `esp` de 4.
- `destination` est une adresse ou un registre
- Exemples :

```
pop eax    ; depile dans le registre eax  
pop [var]  ; depile a l'adresse var
```

Addition - add

- Syntaxe :

add destination source

- Effectue $\text{destination} = \text{destination} + \text{source}$

- **source** : un registre, une adresse ou une constante

- **destination** : un registre ou une adresse

- modifie éventuellement les flags overflow (**OF**) et carry (**CF**)

- Les opérations registre - registre sont possibles, mais pas les opérations mémoire - mémoire

- Exemples :

```
add eax, ebx      ; reg reg
add eax, [var]     ; reg mem
add eax, 12        ; reg const
add [var], eax     ; mem reg
add [var], 1       ; mem const
```

Soustraction - sub

- Syntaxe :

sub destination source

- Effectue $\text{destination} = \text{destination} - \text{source}$
- **source** : un registre, une adresse ou une constante
- **destination** : un registre ou une adresse
- modifie éventuellement les flags overflow (**OF**) et carry (**CF**)
- Les opérations registre - registre sont possibles, mais pas les opérations mémoire - mémoire
- Exemples :

```
sub eax, ebx      ; reg reg
sub eax, [var]     ; reg mem
sub eax, 12        ; reg const
sub [var], eax     ; mem reg
sub [var], 1       ; mem const
```

Multiplication – mul

- Syntaxe :

`mul source`

- Effectue : `eax = eax * source`

- La multiplication de deux entiers codés sur 32 bits peut nécessiter 64 bits.

- les quatre octets de poids de plus faible sont mis dans `eax` et les quatre octets de poids le plus fort dans `edx` (`edx:eax`).

- `source` : adresse, constante ou registre

- Exemples :

```
mul ebx      ; eax = eax * ebx
mul [var]    ; eax = eax * var
mul 12       ; eax = eax * 12
```

Division – div

- Syntaxe :

div source

- Effectue la division entière : `edx:eax` / `source`
- Le quotient est mis dans `eax`
- Le reste est mis dans `edx`
- `source` : adresse, constante ou registre

Opérations logiques

```
and  destination  source
or   destination  source
xor  destination  source
not  destination
```

- Effectue les opérations logiques correspondantes bit à bit
- Le résultat se trouve dans `destination`
- opérandes :
 - `source` peut être : une adresse, un registre ou une constante
 - `destination` peut être : une adresse ou un registre

Comparaisons – cmp

- Syntaxe :

cmp destination, source

- Effectue l'opération destination - source
- le résultat n'est pas stocké
- destination : registre ou adresse
- source : constante, registre ou adresse
- les valeurs des flags ZF (zero flag), SF (sign flag) et PF (parity flag) sont affectés
- si destination = source, ZF vaut 1
- si destination > source, SF vaut 1,

Saut inconditionnel – jmp

- Syntaxe :

jmp adr

- va à l'adresse adr

Saut conditionnel – je

- Syntaxe :

je adr

- je veut dire *jump equal*
- Si ZF vaut 1 va à l'adresse adr

Autres sauts conditionnels – jne, jg, jl

Instruction	Description	Flags testés
jne	jump not equal	ZF
jg	jump greater	OF, SF, ZF
jl	jump less	OF, SF

Appel de procédure - call

- Syntaxe :

call adr

- empile eip (instruction pointer)
- va à l'adresse adr
- utilisé dans les appel de procédure : va à l'adresse où se trouve les instructions de la procédure et sauvegarde la prochaine instruction à effectuer au retour de l'appel.

Retour de procédure - ret

- Syntaxe :

`ret`

- dépile `eip`
- utilisé en fin de procédure
- à utiliser avec `call`

Appels système

- Syntaxe :

`int 0x80`

- NASM permet de communiquer avec le système de façon simple par la commande `int 0x80`.
- La fonction réalisée est déterminée par la valeur de `eax`

<code>eax</code>	Name	<code>ebx</code>	<code>ecx</code>	<code>edx</code>
1	<code>sys_exit</code>	<code>int</code>		
3	<code>sys_read</code>	<code>unsigned int</code>	<code>char *</code>	<code>size_t</code>
4	<code>sys_write</code>	<code>unsigned int</code>	<code>const char *</code>	<code>size_t</code>

Compilation directe

L'assembleur est une version "lisible" du langage machine, mais on pourrait en écrire directement.

```
add    $t7, $t3, $sp
```

- Le registre \$t0 est en fait le numéro 8, et \$sp le numéro 29.
- Le code de l'opération est ici coupé en deux : le premier (0) indique c'est une opération arithmétique, le deuxième de quelle opération arithmétique précisément.

op1	\$t7	\$t3	\$sp	(inutilisé)	op2
000000	01111	01011	11101	00000	010100

Compilation directe

L'assembleur est une version "lisible" du langage machine, mais on pourrait en écrire directement.

```
add    $t7, $t3, $sp
```

- Le registre \$t0 est en fait le numéro 8, et \$sp le numéro 29.
- Le code de l'opération est ici coupé en deux : le premier (0) indique c'est une opération arithmétique, le deuxième de quelle opération arithmétique précisément.

op1	\$t7	\$t3	\$sp	(inutilisé)	op2
000000	01111	01011	11101	00000	010100

Soit, en hexadécimal, 01ebe814.

On pourrait donc compiler directement un exécutable MIPS.

Références

- Cours d'architecture de Peter Niebert :

<http://www.cmi.univ-mrs.fr/~niebert/archi2012.php>

- Introduction au MIPS :

<http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm>

- Table de référence du MIPS :

<http://pageperso.lif.univ-mrs.fr/~alexis.nasr/Ens/Compilation/mipsref.pdf>

- Cours de compilation de François Pottier :

<http://www.enseignement.polytechnique.fr/informatique/INF564/>