

Le langage assembleur intel 64 bits

©Sovanna Tan

Novembre 2013

Plan

- 1 Introduction
- 2 Les registres
- 3 Les instructions
- 4 Les structures de contrôle
- 5 Les tableaux
- 6 Les sous programmes
- 7 Les calculs flottants

Sources

- *Introduction to 64 Bit Intel Assembly Language Programming for Linux*, RAY SEYFARTH, CreateSpace Independent Publishing Platform, 2nd edition , 2012.
- *La documentation Yasm*, <http://yasm.tortall.net/Guide.html>.
- *Le langage assembleur, Maîtrisez le code des processeurs de la famille X86*, OLIVIER CAUET, Editions ENI, 2011.
- *Introduction à la programmation en assembleur 64 bits*, PIERRE JOURLIN, <http://www.jourlin.com>, 2010.
- *Langage Assembleur PC (32 bits)* , PAUL A. CARTER traduction SÉBASTIEN LE RAY, 2005, <http://www.drpaulcarter.com/pcasm>.
- *Initiation à l'assembleur*, Pierre Marchand, 2000, <http://www.ift.ulaval.ca/~marchand/ift17583/Supplement2.pdf>.
- *Linux, Assembly Language Programming*, BOB NEVELN, Prentice Hall , 2000.

Le microprocesseur

Microprocesseur

- Circuit électronique complexe : cœur de l'ordinateur
- Identifie et exécute les instructions

Instruction

- Code opérateur ou *opcode* : action à effectuer
- Opérande(s)

Registre

- Petite zone de mémoire d'accès très rapide située dans le microprocesseur

Architecture simplifiée d'un processeur

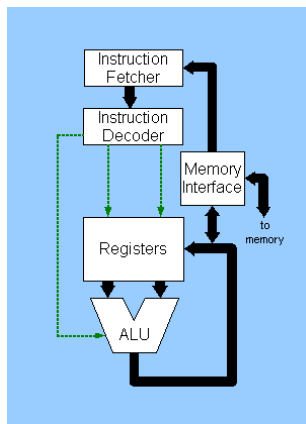


Image provenant de : <http://www.lions-wing.net/lessons/hardware/hard.html#cpu>

Le langage assembleur

Langage de bas niveau

- Chaque instruction se traduit directement en une instruction binaire pour le processeur.
- Il existe différents dialects :
 - **Yasm, the modular assembler** : utilisé dans ce cours, en TD et en TP, issu de
 - **Netwide asm (nasm)** : utilisé les années précédentes
 - **Gas** : assembleur GNU utilisé avec gcc
 - **Macro asm** : assembleur Microsoft
 - **Turbo asm** : assembleur Borland

Premier programme en assembleur (1)

```

; fonctions externes pour les entrees/sortie ; appel printf("%s",prompt1)
extern printf
extern scanf
segment .data ; memoire globale
; donnees initialisees
prompt1 db "Entrez_un_entier_::" ,0 ; appel scanf("%ld",entier1)
prompt2 db "Entrez_un_deuxieme_entier_::" ,0
formatSortie db "La_somme_des_deux_entiers_::%ld" ,10,0
stringFormat db "%s" ,0
longIntFormat db "%ld" ,0
newLine db 10,0

; appel printf("%s",prompt2)
segment .bss ; memoire globale
; donnees non itialisees
entier1 resq 1
entier2 resq 1
resultat resq 1

; appel printf("%s",prompt1)
mov rdi,stringFormat
mov rsi,prompt1
mov rax,0
call printf

; appel scanf("%ld",entier1)
mov rdi,longIntFormat
mov rsi,entier1
mov rax,0
call scanf

; appel printf("%s",prompt2)
mov rbx,[entier1]
mov rdi,stringFormat
mov rsi,prompt2
mov rax,0 ; rax contient le nombre de
; parametres de type double dans l'appel de
; printf
call printf

; appel scanf("%ld",entier2)
segment .text ; code du programme
global asm_main
asm_main: ; fonction appelee par le programme C
; sauvegarde des registres sur la pile
push rbp
; parametres de type double dans l'appel de
; scanf
call scanf
mov rcx,[entier2]
; calcul de la somme et sauvegarde du resultat
add rbx,rcx
mov [resultat],rbx

```

Premier programme en assembleur et code C qui appelle le programme

```

; appel printf(formatSortie,*resultat)
mov rdi,formatSortie
mov rsi,[resultat]
mov rax,0
call printf
; restauration des registres
pop rbp
; envoi de 0 au programme C
mov rax, 0
ret

```

```

extern unsigned long int asm_main();

int main(){
    unsigned long int ret_status;
    ret_status=asm_main();
    return ret_status;
}

```

Commandes de compilation

```

gcc -c -g -std=c99 -m64 driver.c
yasm -g dwarf2 -f elf64 first.asm
gcc -m64 -g -std=c99 -o first driver.o first.o

```

Exécution du programme

```

./first
Entrez un entier : 7
Entrez un deuxieme entier : 34
La somme des eux entiers : 41

```


- 1 Introduction
- 2 Les registres**
- 3 Les instructions
- 4 Les structures de contrôle
- 5 Les tableaux
- 6 Les sous programmes
- 7 Les calculs flottants

Les principaux registres 64 bits

rax	registre général, accumulateur, contient la valeur de retour des fonctions
rbx	registre général
rcx	registre général, compteur de boucle
rdx	registre général, partie haute d'une valeur 128 bits
rsi	registre général, adresse source pour déplacement ou comparaison
rdi	registre général, adresse destination pour déplacement ou comparaison
rsp	registre général, pointeur de pile (stack pointer)
rbp	registre général, pointeur de base (base pointer)
r8	registre général
r9	registre général
:	
r15	registre général
rip	compteur de programme (instruction pointer)

Accès à une partie du registre

Certaines parties des registres sont accessibles à partir des identifiants suivants :

rax, rbx, rcx, rdx, rdi, rsi, rbp, rsp, r8, r9, ... ,r15	64 bits
eax, ebx, ecx, edx, edi, esi, ebp, esp, r8d, r9d, ... ,r15d	32 bits
ax, bx, cx, dx, di, si, bp, sp, r8w, r9w, ... ,r15w	16 bits (15:0)
ah, bh, ch, dh	8 bits high (15:8)
al, bl, cl, dl, dil, sil, bpl, spl, r8b, r9b, ... ,r15b	8 bits low (7:0)

Le registre RFLAGS

- Le registre **rflags** contient des informations concernant le résultat de l'exécution d'une instruction.
- Certains bits du registre appelés drapeaux ont une signification particulière.
- Seuls les 32 bits de la partie **eflags** sont utilisés.

Drapeaux du registre RFLAGS

CF Carry Flag (bit 0)	retenue
PF Parity Flag (bit 2)	
AF Auxiliary Carry Flag (bit 4)	
ZF Zero Flag (bit 6)	vaut 1 lorsque le résultat est 0
SF Sign Flag (bit 7)	bit de signe du résultat
OF Overflow Flag (bit 11)	dépassement, le résultat contient trop de bits
DF Direction Flag (bit 10)	sens d'incrémement de ESI et EDI
TF Task Flag (bit 8)	active la gestion de tâche en mode protégé
IF Interrupt Flag (bit 9)	interruption
IOPL I/O Privilege Level (bits 12 et 13)	
NT Nested Task (bit 14)	
RF Resume Flag (bit 16)	active le mode debug
VM Virtual 8086 Mode (bit 17)	
AC Aligment Check (bit 18)	
VIF Virtual Interrupt Flag (bit 19)	
VIP Virtual Interrupt Pending (bit 20)	
ID Identification Flag (bit 21)	

- 1 Introduction
- 2 Les registres
- 3 Les instructions**
- 4 Les structures de contrôle
- 5 Les tableaux
- 6 Les sous programmes
- 7 Les calculs flottants

Le langage assembleur

Langage de bas niveau

- Chaque instruction se traduit directement en une instruction binaire pour le processeur.
- Syntaxe : *instruction source* ou *instruction destination source*

Types de l'opérande source

- registre
- mémoire
- immédiat, valeur codée dans l'instruction
- implicite, valeur qui n'apparaît pas dans l'instruction

Types de l'opérande destination

- registre
- mémoire

ATTENTION : Une instruction ne peut pas avoir deux opérandes de type mémoire.

Accès à la mémoire avec l'opérateur []

- [*cetteAdresse*] représente la valeur stockée à l'adresse *cetteAdresse*.
- [*ceRegistre*] représente la valeur stockée à l'adresse contenue dans le registre *ceRegistre*.
- On peut associer une étiquette *cetteEtiquette* à une adresse mémoire et utiliser [*cetteEtiquette*].

Les directives de données

- Permettent de réserver de l'espace de mémoire dans les segments de données.

dx	dans le segment .data données initialisées
resx	dans le segment .bss (Block Started by Symbol) données non initialisées

- La valeur du caractère *x* dépend de la taille des données.

b	1 octet (byte)
w	1 mot (word)
d	2 mots (double word)
q	4 mots (quadruple word)
t	10 octets

Les instructions de déplacement de données

mov <i>dest,src</i>	$dest \leftarrow src$
mov <i>taille dest,src</i>	
movzx <i>dest,src</i>	extension avec des 0 dans <i>dest</i>
movsx <i>dest,src</i>	extension avec le bit de signe dans <i>dest</i>

- Tailles possibles :

byte	1 octet
word	2 octets ou 1 mot
dword	2 mots
qword	4 mots
tword	10 octets

On utilise la taille pour lever l'ambiguïté dans les instructions quand c'est nécessaire. ¹

lea <i>dest,[op]</i>	$dest \leftarrow$ adresse de <i>op</i> (load effective address)
push <i>op</i>	décrémente <i>rsp</i> et empile <i>op</i>
pop <i>op</i>	dépile dans <i>op</i> et incrémente <i>rsp</i>

1. Ce n'est pas réservé à l'instruction **mov**.

La pile

- De nombreux processeurs ont un support intégré pour une pile.
- Une pile est une liste Last-In First-Out (LIFO) : dernier entré, premier sorti.
- **push** ajoute une donnée sur la pile de taille **qword** (64 bits). Cette donnée se trouve au *sommet* de la pile.
- **pop** retire la donnée de taille **qword** qui se trouve au sommet de la pile.
- Le registre **rsp** contient l'adresse de la donnée qui se trouve au sommet de la pile.
- **push** décrémente **rsp** de 8.
- **pop** incrémente **rsp** de 8.

Rôle la pile

- Sauvegarde des registres
- Passage des paramètres lors de l'appel de sous programme
- Stockage de variables locales
- Stockage des adresses de retour

Exemple :

```
push qword 1 ; 1 est stocke en 0x0FFC, RSP = 0FFC  
push qword 2 ; 2 est stocke en 0x0FF4, RSP = 0FF4  
push qword 3 ; 3 est stocke en 0x0FEC, RSP = 0FEC  
pop rax ; RAX = 3, RSP = 0x0FF4  
pop rbx ; RBX = 2, RSP = 0x0FFC  
pop rcx ; RCX = 1, RSP = 0x1004
```

Les instructions arithmétiques

add <i>op1,op2</i>	$op1 \leftarrow op1 + op2$
sub <i>op1,op2</i>	$op1 \leftarrow op1 - op2$
neg <i>reg</i>	$reg \leftarrow -reg$
inc <i>reg</i>	$reg \leftarrow reg + 1$
dec <i>reg</i>	$reg \leftarrow reg - 1$
imul <i>op</i> (signé ou mul non signé)	rdx:rax \leftarrow rax \times <i>op</i>
imul <i>dest,op</i>	<i>dest</i> \leftarrow <i>dest</i> \times <i>op</i>
imul <i>dest,op,immédiat</i>	<i>dest</i> \leftarrow <i>op</i> \times <i>immédiat</i>
idiv <i>op</i> (div non signé)	rax \leftarrow rdx:rax / <i>op</i> , rdx \leftarrow rdx:rax mod <i>op</i> ²

2. Ne pas oublier de mettre 0 dans **rdx** avant d'appeler **idiv**.

Les opérations sur les bits

and <i>op1,op2</i>	$op1 \leftarrow op1 \& op2$
or <i>op1,op2</i>	$op1 \leftarrow op1 op2$
xor <i>op1,op2</i>	$op1 \leftarrow op1 \wedge op2$
not <i>reg</i>	$reg \leftarrow \sim reg$
shl <i>reg,immédiat</i>	$reg \leftarrow reg \ll \text{immédiat}$
shr <i>reg,immédiat</i>	$reg \leftarrow reg \gg \text{immédiat}$
sal <i>reg,immédiat</i>	$reg \leftarrow reg \ll \text{immédiat}$
sar <i>reg,immédiat</i>	$reg \leftarrow reg \gg \text{immédiat}$ signé
rol <i>reg,immédiat</i>	$reg \leftarrow reg$ décalage Circulaire Gauche De <i>imm</i>
ror <i>reg,immédiat</i>	$reg \leftarrow reg$ décalage Circulaire Droite De <i>imm</i>
rcl <i>reg,immédiat</i>	$reg : CF \leftarrow reg : CF$ décalage Circ Gauche de <i>imm</i>
rcr <i>reg,immédiat</i>	$reg : CF \leftarrow reg : CF$ décalage Circ Droite de <i>imm</i>

Les instructions de comparaison et de branchement

cmp <i>op1,op2</i>	calcul de $op1 - op2$ et de ZF,CF et OF
jmp <i>op</i>	branchement inconditionnel à l'adresse <i>op</i>
jz <i>op</i>	branchement à l'adresse <i>op</i> si ZF=1
jnz <i>op</i>	branchement à l'adresse <i>op</i> si ZF=0
jo <i>op</i>	branchement à l'adresse <i>op</i> si OF=1
jno <i>op</i>	branchement à l'adresse <i>op</i> si OF=0
js <i>op</i>	branchement à l'adresse <i>op</i> si SF=1
jns <i>op</i>	branchement à l'adresse <i>op</i> si SF=0
jc <i>op</i>	branchement à l'adresse <i>op</i> si CF=1
jnc <i>op</i>	branchement à l'adresse <i>op</i> si CF=0
jp <i>op</i>	branchement à l'adresse <i>op</i> si PF=1
jnp <i>op</i>	branchement à l'adresse <i>op</i> si PF=0

Les instructions de branchement après `cmp op1,op2`

Signées :

je <i>op</i>	branchement à l'adresse <i>op</i> si $op1 = op2$
jne <i>op</i>	branchement à l'adresse <i>op</i> si $op1 \neq op2$
jl <i>op</i> (jnge)	branchement à l'adresse <i>op</i> si $op1 < op2$
jle <i>op</i> (jng)	branchement à l'adresse <i>op</i> si $op1 \leq op2$
jg <i>op</i> (jnle)	branchement à l'adresse <i>op</i> si $op1 > op2$
jge <i>op</i> (jnl)	branchement à l'adresse <i>op</i> si $op1 \geq op2$

Non signées :

je <i>op</i>	branchement à l'adresse <i>op</i> si $op1 = op2$
jne <i>op</i>	branchement à l'adresse <i>op</i> si $op1 \neq op2$
jb <i>op</i> (jnae)	branchement à l'adresse <i>op</i> si $op1 < op2$
jbe <i>op</i> (jna)	branchement à l'adresse <i>op</i> si $op1 \leq op2$
ja <i>op</i> (jnbe)	branchement à l'adresse <i>op</i> si $op1 > op2$
jae <i>op</i> (jnb)	branchement à l'adresse <i>op</i> si $op1 \geq op2$

Exemple de branchement

```
if (a>b){  
  
    printf("%d" ,a);  
  
}  
else {  
  
    printf("%d" ,b);  
  
}
```

```
if:    cmp    r12 , r13  
       jng   else  
       mov   qword rdi , format  
       mov   qword rsi , r12  
       mov   qword rax , 0  
       call printf  
       jmp  endif  
else:  
       mov   qword rdi , format  
       mov   qword rsi , r13  
       mov   qword rax , 0  
       call printf  
endif:
```

Les boucles

Les instructions suivantes sont déconseillées car elles ne permettent qu'un branchement à une distance inférieure à 127 octets.

loop <i>op</i>	décrémente rcx et saut à <i>op</i> si rcx ≠ 0
loope <i>op</i> (loopz)	rcx -- et saut à <i>op</i> si rcx ≠ 0 et ZF=1
loopne <i>op</i> (loopnz)	rcx -- et saut à <i>op</i> si rcx ≠ 0 et ZF=0

Exemple :

```
for (int a=10;a>0;a--){
```

```
    for1 :
```

```
        mov rcx,10
```

```
        push rcx
```

```
        mov qword rdi,format
```

```
        mov qword rsi,rcx
```

```
        mov qword rax,0
```

```
        call printf
```

```
        pop rcx
```

```
        loop for1
```

```
    printf("%d",a)
```

```
}
```

- 1 Introduction
- 2 Les registres
- 3 Les instructions
- 4 Les structures de contrôle**
- 5 Les tableaux
- 6 Les sous programmes
- 7 Les calculs flottants

if

```

if (a>b){
    ...
}
else{
    ...
}

if ((a > b) && (c <= d)) {
    ...
}

if1 :    cmp rax , rax
          jng  else1
          ...
          jmp endif1
else1 :
    ...
endif1 :

if2 :    cmp rax , rax
          jng  endif2
          cmp rcx , rdx
          jnle endif2
          ...
endif2 :
```

switch

```

switch (i) {
    case 1:
        ...
        break;
    case 2:
        ...
        break;
    default:
        ...
}
...

switcha:    cmp    rcx,1
           jne    casea2
           ...
           jmp    endswitcha
casea2:    cmp    rcx,2
           jne    defaulta
           ...
           jmp    endswitcha
defaulta:
           ...
endswitcha:
           ...

```

while, do while

```
while (a>0){
```

```
...
```

```
a--;
```

```
}
```

```
do {
```

```
...
```

```
a--;
```

```
} while (a>0);
```

```
while1: cmp rax,0  
        jle endwhile1
```

```
...
```

```
dec rax
```

```
jmp while1
```

```
endwhilei1: ...
```

```
do2:
```

```
...
```

```
dec rax;
```

```
cmp rax,0
```

```
jg do2
```

for

```
for (int i=1; i < 10; i++){          for1:    mov rcx, 1
                                     jmp test1
                                     next1:
...                                     ...
                                     inc rcx
                                     test1:  cmp rcx, 10
                                     jl next1
...                                     ...
```

- 1 Introduction
- 2 Les registres
- 3 Les instructions
- 4 Les structures de contrôle
- 5 Les tableaux**
- 6 Les sous programmes
- 7 Les calculs flottants

Les tableaux

- **Bloc continu** de données en mémoire
- Tous les éléments sont de **même type** et de **même taille**.
- L'adresse de chaque élément du tableau se calcule à partir de
 - l'adresse du premier élément du tableau ;
 - le nombre d'octets de chaque élément ;
 - l'indice de l'élément.

L'indice du premier élément du tableau est 0.

- Tableau à deux dimensions : tableau de tableaux à une dimension.
Se généralise pour les dimensions supérieures.
- Chaîne de caractères : tableau d'octets dont le dernier élément est 0.

Exemples de définition de tableau

```
segment .data
```

```
; definit un tableau de 10 doubles mots initialises
```

```
; a 1,2,...,10
```

```
a1      dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
; definit un tableau de 10 quadruples mots initialises a
```

```
a2      dq 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

```
; idem mais en utilisant times
```

```
a3      times 10 dq 0
```

```
; definit un tableau d'octets avec 200 0 puis 100 1
```

```
a4      times 200 db 0
```

```
        times 100 db 1
```

```
segment .bss
```

```
; definit un tableau de 10 doubles mots non initialises
```

```
a5      resd 10
```

```
; definit un tableau de 100 mots non initialises
```

```
a6      resw 100
```

Exemple d'accès aux éléments d'un tableau

```
array1 db 50, 40, 30, 200, 100 ; tableau d'octets
array2 dw 5, 4, 3, 2, 1 ; tableau de mots
```

```
mov al,[array1] ; al=array1[0]
mov al,[array1 +1] ; al=array1[1]
mov [array1 + 3],al ; array1[3]=al
mov ax,[array2] ; ax=array2[0]
mov ax,[array2 + 2] ; ax=array2[1]
mov [array2 + 6],ax ; array2[3]=ax
mov ax,[array2 +1] ; ax=??
```

Calcul de la somme des éléments du tableau array1

```
mov rbx,array1 ; rbx = adresse de array1
mov rdx,0 ; dx contiendra la somme
mov rcx,5 ; initialisation compteur boucle

next1:
add dl,[rbx] ; dl += *ebx
jnc suite1 ; si pas de retenue goto suite1
inc dh ; incremente dh quand retenue

suite1:
inc rbx ; bx++
dec rcx
cmp rcx,0
jg next1
```

Adressage indirect

L'adressage indirect facilite l'accès aux éléments d'un tableau ou de la pile.

```
[deplacement+registre_base+facteur*registre_index]
```

- `registre_base` et `registre_index` sont des registres généraux
- `facteur` vaut 1, 2, 4 ou 8 (omis lorsqu'il vaut 1).
- `deplacement` est une étiquette.

Exemple : tri par sélection (1)

```

extern printf
segment .data
longIntFormat db "%ld_" , 0
stringFormat db "%s" , 0
newline db 10, 0
a dq 7, 5, 2, 18, 14, 8
segment .bss
; tri par selection du tableau b
; elements de a numerotes de 0 a n-1
; pour i de 0 a n-2
;   min ← i
;   pour j de i + 1 a n-1
;     si a[j] < a[min] min ← j
;     si min != i echanger a[i] et a[min]
segment .text
global asm_main
asm_main:
; sauvegarde pointeur pile
push rbp
mov rbp, rsp
push r12

for1: mov rcx, 0
      jmp test1
; rbx contient a[rcx]
next1: mov rbx, [a+8*rcx]
       mov rdx, rcx
       push rcx
       inc rdx

if1:  cmp rbx, [a+8*rdx]
      jng endif1
; valeur min dans rbx
      mov rbx, [a+8*rdx]
; sauvegarde de indice min sur pile
      push rdx
endif1: inc rdx
test2: cmp rdx, 6
      jl if1
; min dans rbx, echange
      mov rax, [a+8*rcx]
      mov [a+8*rcx], rbx
      pop rdx
      mov [a+8*rdx], rax
      inc rcx

test1: cmp rcx, 5
      jl next1

```

Exemple : tri par sélection (2)

Exécution du programme

```

;affichage du tableau
mov r12,0
jmp test3

next3: mov rdi, longIntFormat
      mov rsi, [a+8*r12]
      mov rax, 0
      call printf
      inc r12
test3: cmp r12, 6
      jl next3
      mov rdi, stringFormat
      mov rsi, newLine
      mov rax, 0
      call printf
;restauration pointeur pile
pop r12
mov rsp, rbp
pop rbp

mov rax, 0
ret

```

```

./trise1
2 5 7 8 14 18

```

- 1 Introduction
- 2 Les registres
- 3 Les instructions
- 4 Les structures de contrôle
- 5 Les tableaux
- 6 Les sous programmes**
- 7 Les calculs flottants

Appel de sous programme

Sous programme

- Partie du code écrite une seule fois que l'on peut exécuter à partir de différents endroits du programme.
- Implémentation des procédures et des fonctions des langages de haut niveau.
- A la fin de l'exécution d'un sous programme, l'exécution doit se poursuivre avec l'instruction qui suit l'appel.
- Impossible à implémenter avec une étiquette.
- Il faut mémoriser l'adresse de retour et faire un saut à cette adresse en utilisant l'adressage indirect.

On dispose des instructions suivantes :

call <i>op</i>	saut inconditionnel à <i>op</i> et empile l'adresse de l'instruction suivante
ret	dépile une adresse et saute à cette adresse

Attention à la gestion de la pile!

La valeur de retour des fonctions

Elles sont passées par des registres :

- **rax** pour un pointeur ou un type entier de taille inférieure ou égale à 64 bits (Les valeurs sont étendues sur 64 bits.) ;
- **rdx:rax** pour une valeur 128 bits ;
- **xmm0** pour une valeur flottante ou **xmm1:xmm0** si besoin.

Exemple : calcul de $n^2 - 1$

```

extern printf
segment .data
formatSortie1 db "6^2-1\vaut\_:_%ld",10,0
formatSortie2 db "8^2-1\vaut\_:_%ld",10,0

segment .bss

segment .text
global asm_main
asm_main:
; sauvegarde des registres sur la pile
push rbp
; 1er appel de la fonction
mov rdi,6
call fonc1
; appel printf
mov rdi,formatSortie1
mov rsi,rax
mov rax,0
call printf
; 2eme appel de la fonction
mov rdi,8
call fonc1

; appel printf
mov rdi,formatSortie2
mov rsi,rax
mov rax,0
call printf
; restauration des registres
pop rbp
; envoi de 0 au programme C
mov rax,0
ret
; code de la fonction
fonc1:
push rbp
mov rbp,rsb
imul rdi,rdi
sub rdi,1
mov rax,rdi
mov rsp,rbp
pop rbp
ret

```

Code réentrant

Sous programme réentrant

Un sous programme réentrant peut être utilisé simultanément par plusieurs tâches. Une seule copie du code en mémoire peut être utilisée par plusieurs utilisateurs en même temps. Pour cela, le sous programme réentrant ne doit pas :

- modifier son code ;
- modifier les données globales comme celles des segments **.data** et **.bss**. Toutes les variables locales doivent être stockées sur la pile.

Calcul récursif

Sous programme récursif

Un sous programme récursif est un sous programme qui s'appelle lui même :

- directement ;
- ou indirectement par l'intermédiaire d'autres appels de fonctions.

Un sous programme récursif doit avoir une condition de terminaison pour que le programme s'arrête, sinon il engendre une erreur à l'exécution lorsque les ressources sont épuisées.

Un sous programme réentrant peut s'appeler de manière récursive.

Les variables locales

- On alloue l'espace requis par les variables locales en diminuant **rsp**. La taille des blocs doit être un multiple de 16 pour Windows et Linux.
- On peut alors mettre les valeurs des variables locales dans **[rsp]**, **[rsp+8]**,...
- A la fin du sous programme, on libère l'espace mémoire correspondant avec **mov rsp,rbp**

Les instructions `enter` et `leave`

- Simplifient l'écriture des sous programmes.

```
enter TAILLE_VAR_LOC,0           push rbp  
                                     mov rbp , rsp  
                                     sub rsp ,TAILLE_VAR_LOC  
  
... ; code du ss prog           ... ; code du ss prog  
  
leave                             mov rsp , rbp  
                                     pop rbp  
  
ret                               ret
```

- `TAILLE_VAR_LOC` doit être un multiple de 16. Le pointeur `rsp` doit contenir la valeur 0 en hexadécimal pour le chiffre des unités.
- Les systèmes d'exploitation Windows et Linux demandent le maintien du pointeur de pile aligné sur un bloc de 16 octets. Sa valeur en hexadécimal doit contenir 0 pour le chiffre des unités. Un appel de fonction avec la valeur 8 en hexadécimal pour le chiffre des unités de `rsp` conduit à une segmentation fault.

La pile des appels de fonctions (stack frame)

Comme chaque **call** empile, l'adresse de l'instruction à exécuter après le retour de l'appel, le fait de commencer un appel de fonction par

```
enter TAILLE_VAR_LOC,0
```

ou

```
push rbp
```

```
mov rbp , rsp
```

```
sub rsp , TAILLE_VAR_LOC
```

et de le terminer par **leave** permet au débogueur de remonter la pile des appels de fonctions.

Exemple de sous programme récursif : calcul de $n!$

```
extern printf
segment .data
formatSortie1 db "5! vaut : %ld" ,10,0
segment .bss
segment .text
global asm_main
asm_main:
; sauvegarde des registres sur la pile
push rbp
; appel de la fonction pour n=5
mov rdi,5
call fact
; appel printf
mov rdi,formatSortie1
mov rsi,rax
mov rax,0
call printf
; restauration des registres
pop rbp
mov rax,0
ret
```

```
fact:
; une variable locale entiere
enter 16,0
; sauvegarde du parametre n
mov [rsp],rdi
; cas n==1
cmp rdi,1
je term_cond
; cas <math>n > 1</math> calcul de fact(n-1)
dec rdi
call fact
; fact(n)=fact(n-1)*n
imul rax,[rsp]
jmp end_fact
term_cond:
; fact(1)=1
mov rax,1
end_fact:
leave
ret
```


Conventions d'appel d'un sous programme

- Lors de l'appel d'un sous programme, le code appelant et le sous programme doivent se coordonner pour le passage des paramètres.
- Ce processus est normalisé dans les langages de haut niveau. Ce sont les *conventions d'appel*.
- Au niveau de l'assembleur, il n'y a pas de norme. Les conventions diffèrent d'un compilateur à l'autre.
- Pour interfacer, l'assembleur avec un langage de haut niveau l'assembleur doit suivre les conventions d'appel du langage de haut niveau.

Contenu de la convention d'appel

- Où le code appelant place-t-il les paramètres : sur la pile ou dans des registres ?
- Quand les paramètres sont placés sur la pile, dans quel ordre y sont-ils placés : du premier au dernier ou du dernier au premier ?
- Quand les paramètres sont placés sur la pile, qui doit nettoyer la pile suite à l'appel : le code appelant ou le code du sous programme ?
- Quels sont les *registres préservés (callee-save)* que le sous programme doit sauver avant d'utiliser et restaurer à la fin et *les registres de travail (caller-save)* dont l'appelant a sauvé le contenu et qui peuvent être utilisés par le sous programme à volonté ?
- Où se trouve la valeur de retour d'une fonction ?

Le passage des paramètres

- Le passage des paramètres peut se faire par les registres comme dans les fonctions `fonc1` et `fact`.
- S'il y a trop de paramètres, il faut en passer certains par la pile.
 - On empile les paramètres avant le **call**.
 - Si le sous programme doit modifier le paramètre, il faut passer l'adresse de la donnée à modifier.
 - Si la taille du paramètre est inférieure à celle d'un **qword**, il faut l'étendre avant de l'empiler.
- Les paramètres ne sont pas dépilés par le sous programme mais on y accède depuis la pile.
 - L'adresse de retour empilée par **call** doit être dépilée en premier.
 - Les paramètres peuvent être utilisés plusieurs fois dans le sous programme.

Les conventions d'appel du C (cdecl)

- Les paramètres sont empilés dans l'ordre inverse de celui dans lequel ils sont écrits dans la fonction C.
- Utilisation du registre **rbp** pour accéder aux paramètres passés par la pile (écriture simplifiée avec **enter** et **leave**).
 - Le sous programme doit empiler **rbp** avec **push rbp** puis copier **rsp** dans **rbp** avec **mov rbp,rsp**.
 - On accède aux paramètres en utilisant **rbp**. Les paramètres sont **[rbp+16]**, **[rbp+24]**...
Cela permet de continuer à utiliser la pile dans le sous programme pour stocker les variables locales afin d'écrire du code réentrant.
 - On alloue l'espace requis par les variables locales en diminuant **rsp**.
 - A la fin du sous programme, on libère l'espace mémoire correspondant avec **mov rsp,rbp**
 - A la fin de l'appel, le sous programme doit restaurer la valeur de **rbp** avec **pop rbp**.
- Le code appelant doit retirer les paramètres de la pile.

Différences entre Linux et Windows

● Linux

- Les six premiers paramètres entiers sont passés dans **rdi**, **rsi**, **rdx**, **rcx**, **r8** et **r9** dans cet ordre. Les autres sont passés par la pile.
- Ces registres, ainsi que **rax**, **r10** et **r11** sont détruits par les appels de fonctions.
- Les registres callee-save sont **rbx**, **r12**, ... , **r15**.
- Les paramètres flottants sont passés dans **xmm0**, **xmm1**, ..., **xmm7**.

● Windows

- Les quatre premiers paramètres entiers sont passés dans **rcx**, **rdx**, **r8** et **r9** dans cet ordre. Les autres sont passés par la pile.
- Ces registres, ainsi que **rax**, **r10** et **r11** sont détruits par les appels de fonctions.
- La valeur de retour entière est contenue seulement dans **rax**.
- Les paramètres flottants sont passés dans **xmm0**, **xmm1**, **xmm2** et **xmm3**.
- La valeur de retour flottante est contenue seulement dans **xmm0**.

D'autres conventions

- `stdcall` :
 - Le sous programme doit retirer les paramètres de la pile. Cela empêche l'écriture de fonctions à nombre variable d'arguments comme `printf` ou `scanf`.
 - Utilisée dans l'API Microsoft.
 - Avec `gcc`, on peut préciser `cdecl` ou `stdcall` dans le code C.
- Pascal :
 - Les paramètres sont empilés dans l'ordre où ils apparaissent dans la fonction, de gauche à droite.
 - Le sous programme doit retirer les paramètres de la pile.
 - Utilisée dans Delphi de Borland.

Fonction qui affiche un tableau d'entier

```

extern printf
segment .data
longIntFormat db "%ld_" ,0
stringFormat db "%s" ,0
newLine db 10,0
a dq 7, 5, 2, 18, 14, 8
segment .bss
segment .text
global asm_main
asm_main:
    push rbp
; affiche les 6 elements du tableau
    mov rdi ,a
    mov rsi ,6
    call affiche
; affiche 3 elements a partir du 2eme
    mov rdi ,a
    add rdi ,16
    mov rsi ,3
    call affiche

    pop rbp
    mov rax ,0
    ret

;affichage du tableau
affiche:
    push rbp
    mov rbp ,rsp

    mov rbx ,rdi
    mov r13 ,rsi
    mov r12 ,0
    jmp test3
next3:  mov rdi ,longIntFormat
    mov rsi ,[rbx+8*r12]
    mov rax ,0
    call printf
    inc r12
test3: cmp r12 ,r13
    jl next3
    mov rdi , stringFormat
    mov rsi , newLine
    mov rax , 0
    call printf

    leave
    mov rax , 0
    ret

./afftab
7 5 2 18 14 8
2 18 14

```

Tri par sélection avec fonctions (1)

```

extern printf
segment .data
longIntFormat db "%ld_" ,0
stringFormat db "%s" ,0
newline db 10,0
a dq 7, 5, 2, 18, 14, 8
b dq 27, 35, 12, 25, 19, 12, 34
segment .bss
; tri par selection du tableau t
; elements de a numerotes de 0 a n-1
; pour i de 0 a n-2
;   min ← i
;   pour j de i + 1 a n-1
;     si t[j] < t[min] min ← j
;     si min != i echanger t[i] et t[min]
segment .text
global asm_main
asm_main:
    push rbp
; appel de la fonction trisel
; le premier parametre, adresse du tableau,
; et le deuxieme parametre, taille du tableau
; sont passes dans les registres rdi et rsi,
; le troisieme parametre, nombre d'elements
; a trier, est passe par la pile
; tri de a
    mov rdi ,a
    mov rsi ,6
    push 6
    call trisel
    add rsp ,8
; affichage du tableau
    mov rdi ,a
    mov rsi ,6
    call affichage
; tri des 5 premiers elements de b
    mov rdi ,b
    mov rsi ,7
    push 5
    call trisel
    add rsp ,8
; affichage du tableau
    mov rdi ,b
    mov rsi ,7
    call affichage
    pop rbp
    mov rax ,0
    ret

```


Tri par sélection avec fonctions (2)

```

; fonction de tri
trisel:
    enter 0,0
    mov r12,rdi
    mov r14,rsi
    mov r13,[rbp+16]
    dec r13

for1:   mov rcx,0
        jmp test1
;rbx contient t[rcx]
next1:
; sauvegarde des registres
; et affichage du tableau
    push rcx
    push rdx
    mov rdi,r12
    mov rsi,r14
    call affichage
    pop rdx
    pop rcx

        mov rbx,[r12+8*rcx]
        mov rdx,rcx
        push rcx
        inc rdx
if1:   cmp rbx,[r12+8*rdx]
        jng endif1
; valeur min dans rbx
        mov rbx,[r12+8*rdx]
; sauvegarde de indice min sur pile
        push rdx
endif1: inc rdx
test2: cmp rdx,[rbp+16]
        jl if1
; min dans rbx, echange
        mov rax,[r12+8*rcx]
        mov [r12+8*rcx],rbx
        pop rdx
        mov [r12+8*rdx],rax
        inc rcx
test1: cmp rcx,r13
        jl next1

        leave
        ret

```

Tri par sélection avec fonctions (3)

```
./trisel2
7 5 2 18 14 8
2 5 7 18 14 8
2 5 7 18 14 8
2 5 7 18 14 8
2 5 7 8 14 18
2 5 7 8 14 18
27 35 12 25 19 12 34
12 35 27 25 19 12 34
12 19 27 25 35 12 34
12 19 25 27 35 12 34
12 19 25 27 35 12 34
```

- 1 Introduction
- 2 Les registres
- 3 Les instructions
- 4 Les structures de contrôle
- 5 Les tableaux
- 6 Les sous programmes
- 7 Les calculs flottants**

L'unité de calcul flottant (FPU : Floating Point Unit)

- Au début, elle était un circuit séparé, le coprocesseur mathématique.
- Puis, elle a été intégrée au microprocesseur, mais se program-
mait toujours de façon distincte.
- Avec les architectures 64 bits, apparaît un nouveau jeu d'ins-
tructions SSE (Streaming SIMD (Single Instruction Multiple
Data) Extensions) pour des registres de taille 128 ou 256 bits
selon les processeurs.

Les types de nombres flottants

- Simple précision IEEE 754-2008 : 1 bit signe, 8 bits exposant, 23 bits mantisse
- Double précision IEEE 754-2008 : 1 bit signe, 11 bits exposant, 52 bits mantisse
- Precision étendue intel 80 bits : 1 bit signe, 15 bits exposant, 64 bits mantisse
 - Ce format est différent des formats IEEE 754-2008.
 - Dans le processeur Intel 32 bits, tous les calculs étaient effectués en précision étendue puis convertis en simple ou double précision IEEE 754-2008.
- Quadruple précision IEEE 754-2008 : 1 bit signe, 15 bits exposant, 112 bits mantisse
- 256 bits (extrapolation de IEEE 754-2008) : 1 bit signe, 19 bits exposant, 236 bits mantisse

Les registres

- 16 registres généraux **xmm0**, **xmm1**,... et **xmm15**.
- Ces registres peuvent être utilisés avec des instructions
 - pour une seule valeur
 - ou pour un vecteur de valeurs.

Ce vecteur comporte alors soit 4 **floats** soit 2 **doubles**.

- Les processeurs de la série i-core ont 16 registres 256 bits **yymm0**, **yymm1**,... et **yymm15**. Les registres **xmm0**, **xmm1**,... et **xmm15** correspondent aux premiers 128 bits des registres **yymm0**, **yymm1**,... et **yymm15**. Ces derniers peuvent être vu comme des vecteurs de 8 **floats** ou 4 **doubles**.

Les instructions de déplacement de données

- Déplacement d'une seule valeur

movsd <i>dest,src</i>	déplacement d'un flottant double
movss <i>dest,src</i>	déplacement d'un flottant simple (float)

- Déplacement de paquets (plusieurs valeurs simultanément)

movapd <i>dest,src</i>	déplacement de 2 doubles alignés ³
movaps <i>dest,src</i>	déplacement de 4 floats alignés
movdqa <i>dest,src</i>	déplacement d'un double qword aligné
movupd <i>dest,src</i>	déplacement de 2 doubles non alignés
movups <i>dest,src</i>	déplacement de 4 floats non alignés
movdqu <i>dest,src</i>	déplacement d'un double qword non aligné

- Les noms des instructions pour les registres 256 bits commencent par **v**, **vmovapd**, **vmovaps**,...

3. alignés sur 1 bloc de 16 octets

Calculs flottants

addsd <i>dest,src</i>	addition de 2 doubles
addss <i>dest,src</i>	addition de 2 floats
addpd <i>dest,src</i>	2 doubles + 2 doubles
addps <i>dest,src</i>	4 floats + 4 floats
subsd <i>dest,src</i>	soustraction de 2 doubles
subss <i>dest,src</i>	soustraction de 2 floats
subpd <i>dest,src</i>	2 doubles - 2 doubles
subps <i>dest,src</i>	4 floats - 4 floats
mulsd <i>dest,src</i>	multiplication de 2 doubles
mulss <i>dest,src</i>	multiplication de 2 floats
mulpd <i>dest,src</i>	2 doubles * 2 doubles
mulps <i>dest,src</i>	4 floats * 4 floats
divsd <i>dest,src</i>	division de 2 doubles
divss <i>dest,src</i>	division de 2 floats
divpd <i>dest,src</i>	2 doubles / 2 doubles
divps <i>dest,src</i>	4 floats / 4 floats

- *dest* doit être un registre flottant.

Les instructions de conversion

cvtss2sd <i>dest,src</i>	convertit un float en double
cvtps2pd <i>dest,src</i>	convertit un paquet de 2 floats en un paquet de 2 doubles
cvtsd2ss <i>dest,src</i>	convertit un double en float
cvtpd2ps <i>dest,src</i>	convertit un paquet de 2 doubles en un paquet de 2 floats
cvtss2si <i>dest,src</i>	convertit un float en dword ou qword
cvtsd2si <i>dest,src</i>	convertit un double en dword ou qword
cvtsi2ss <i>dest,src</i>	convertit un entier en float
cvtsi2sd <i>dest,src</i>	convertit un entier en double

- Lorsque l'opérande source est dans la mémoire, il faut préciser la taille de la donnée à lire.

Les instructions de comparaison

comisd <i>op1,op2</i>	comparaison ordonnée de deux doubles
comiss <i>op1,op2</i>	comparaison ordonnée de deux floats
ucomisd <i>op1,op2</i>	comparaison non ordonnée de deux doubles
ucomiss <i>op1,op2</i>	comparaison non ordonnée de deux floats

- *op1* doit être un registre, *op2* peut être un registre ou une valeur en mémoire.
- Les instructions mettent à jour les drapeaux ZF, PF et CF.
- Les comparaisons non ordonnées lèvent une exception si l'un des opérandes vaut SNaN (Signaling Not a Number i.e. que des 1 dans l'exposant et mantisse de la forme 0x) et les comparaisons ordonnées lèvent une exception si l'un des deux opérandes vaut SNaN ou QNaN (Quiet Not a Number i.e. que des 1 dans l'exposant et mantisse de la forme 1x).

Les branchements conditionnels pour les flottants

- Après une instruction de comparaison de flottants (**comisd**, **comiss**, **ucomisd** ou **ucomiss**) entre *op1* et *op2* :

jb <i>op</i>	branchement à <i>op</i> si $op1 < op2$	CF=1
jbe <i>op</i>	branchement à <i>op</i> si $op1 \leq op2$	CF=1 ou ZF=1
ja <i>op</i>	branchement à <i>op</i> si $op1 > op2$	CF=1
jae <i>op</i>	branchement à <i>op</i> si $op1 \geq op2$	CF=1 ou ZF=1

Arrondis

roundsd <i>dest,src,mode</i>	arrondi <i>src</i> de type double dans <i>dest</i>
roundss <i>dest,src,mode</i>	arrondi <i>src</i> de type float dans <i>dest</i>
roundpd <i>dest,src,mode</i>	arrondi des 2 doubles de <i>src</i> dans <i>dest</i>
roundps <i>dest,src,mode</i>	arrondis des 4 floats de <i>src</i> dans <i>dest</i>

- *dest* doit être un registre flottant.
- *mode* est un opérande immédiat :

0	arrondi à la valeur entière paire la plus proche
1	arrondi à la valeur entière inférieure
2	arrondi à la valeur supérieure
3	arrondi à la valeur absolue inférieure (à la valeur entière la plus proche de 0)

Instructions diverses

minsd <i>dest,src</i>	minimum de 2 doubles (<i>src</i> et <i>dest</i>) dans <i>dest</i>
minss <i>dest,src</i>	minimum de 2 floats (<i>src</i> et <i>dest</i>) dans <i>dest</i>
minpd <i>dest,src</i>	2 calculs de minimum de 2 doubles dans <i>dest</i>
minps <i>dest,src</i>	4 calculs de minimum de 2 floats dans <i>dest</i>
maxsd <i>dest,src</i>	maximum de 2 doubles (<i>src</i> et <i>dest</i>) dans <i>dest</i>
maxss <i>dest,src</i>	maximum de 2 floats (<i>src</i> et <i>dest</i>) dans <i>dest</i>
maxpd <i>dest,src</i>	2 calculs de maximum de 2 doubles dans <i>dest</i>
maxps <i>dest,src</i>	4 calculs de maximum de 2 floats dans <i>dest</i>
sqrtsd <i>dest,src</i>	racine carrée de <i>src</i> de type double dans <i>dest</i>
sqrts <i>dest,src</i>	racine carrée de <i>src</i> de type float dans <i>dest</i>
sqrtpd <i>dest,src</i>	2 racines carrées de <i>src</i> (2 doubles) dans <i>dest</i>
sqrtps <i>dest,src</i>	4 racines carrées de <i>src</i> (4 floats) dans <i>dest</i>

- *dest* doit être un registre flottant.

Exemple : calcul de la valeur d'un polynôme (1)

```

extern printf, scanf
segment .data
coeff dq 1.0, 2.0, 3.0, 4.0
x dq 2.0
promptDouble db "%lf", 0
prompt1DoubleIn db "4x3+3x2+2x+1_(2) _=%lf", 10, 0
prompt2DoubleIn db "4x3+3x2+2x+1_(%lf) _=%lf", 10, 0
promptEntrer db "Entrez x: ", 0
segment .bss
y resq 1
segment .text
global asm_main, horner

asm_main:
    push rbp
    mov rbp, rsp
    lea rdi, [coeff]
    movsd xmm0, [x]
    mov esi, 3
    call horner
    mov rdi, prompt1DoubleIn
    mov rax, 1
    call printf
    mov rdi, promptEntrer
    mov rax, 0
    call printf
    mov rdi, promptDouble
    mov rsi, y
    mov rax, 0
    call scanf

    lea rdi, [coeff]
    movsd xmm0, [y]
    mov esi, 3
    call horner
    mov rdi, prompt2DoubleIn
    movsd xmm1, xmm0
    movsd xmm0, [y]
    mov rax, 2
    call printf
    xor eax, eax
    leave
    ret
; extrait du livre de Ray Seyfarth
horner: movsd xmm1, xmm0
; use xmm1 as x
    movsd xmm0, [rdi+rsi*8]
; accumulator for b_k
    cmp esi, 0
; is the degree 0?
    jz done
    more: sub esi, 1
    mulsd xmm0, xmm1
; b_k * x
    addsd xmm0, [rdi+rsi*8]
; add p_k
    jnz more
done: ret

```

Exemple : calcul de la valeur d'un polynôme (2)

```
./horner
```

```
4x3+3x2+2x+1 (2) = 49.000000
```

```
Entrez x : 3
```

```
4x3+3x2+2x+1 (3.000000) = 142.000000
```

Exemple : calcul de la somme de deux vecteurs (1)

```

extern printf
segment .data
a      dq  1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0
b      dq  2.0, 0.5, 4.0, 0.2, 3.0, 1.5, 0.5, 0.4
floatFormat db "%lf_", 0
promptRes db "a*b=", 0
promptln db 10,0
segment .bss
c      resq 8
segment .text
global asm_main, horner
asm_main:
    push    rbp
    mov     rbp, rsp
    mov     rcx, 0
    mov     rdx, 8
    jmp     test1
next1:  movapd xmm0, [a+8*rcx]
        movapd xmm1, [b+8*rcx]
        addpd  xmm1, xmm0
        movapd [c+8*rcx], xmm1
        add   rcx, 2
test1:  cmp     rcx, rdx
        jl    next1

        mov     rdi, a
        mov     rsi, 8
        call    affiche

        mov     rdi, b
        mov     rsi, 8
        call    affiche
        mov     rdi, c
        mov     rsi, 8
        call    affiche
        mov     rax, 0
        leave
        ret

affiche:
        enter 0,0
        push  r12
        push  r13
        push  r14
        push  rbx

        mov  rbx, rdi
        mov  r13, rsi
        mov  r12, 0
        jmp  test3
next3:  mov  rdi, floatFormat
        movsd xmm0, [rbx+8*r12]
        mov  rax, 1
        call printf
        inc  r12
test3:  cmp  r12, r13
        jl  next3

```


Exemple : calcul de la somme de deux vecteurs (2)

```
mov rdi , promptln
mov rax , 0
call    printf

pop rbx
pop r14
pop r13
pop r12
mov rax , 0
leave
ret
```

./addvect

1.000000	2.000000	3.000000	4.000000	5.000000	6.000000	7.000000	8.000000
2.000000	0.500000	4.000000	0.200000	3.000000	1.500000	0.500000	0.400000
3.000000	2.500000	7.000000	4.200000	8.000000	7.500000	7.500000	8.400000