3- Programmation assembleur

L'ensemble des instructions = **Instruction Set Architecture**

Un modèle de machine:

- Unités fonctionnelles
- Registres
- Mémoire

Un modèle d'exécution

- Parallélisme d'instruction
- VLIW, Superscalaire

Un modèle d'instructions

- CISC/RISC, microcode
- Modes d'adressage
- Format d'instructions

3- Programmation assembleur

Mémoire

- Unité de stockage pour mémoriser le programme et les données
- La mémoire est divisée en des **cellules mémoire (1 octet généralement)**
  - Chacune possède un numéro unique, son **adresse**. Adressage sur M bits = 2^M cellules. Actuellement, M=64
  - Contiennent toutes le même nombre de bits. N bits = 2^N valeurs possibles. Pas de typage (instruction, entier, flottant, ...)
- Taille mémoire: 2^M*N/8 octets.

3- Programmation assembleur

- Deux adresses consécutives sont adjacentes.
- Cellule mémoire: plus petite unité mémoire manipulable par les instructions
- Différents types de mémoire (vu de l'ISA)
  - **Random Access Memory** (RAM): on accède à n'importe quelle adresse, dans n'importe quel ordre. Contraire d'un accès séquentiel. Perd les données quand pas alimenté
  - **Read-Only Memory** (ROM): comme RAM, mais seulement en lecture. D'habitude pour programmes, systèmes d'exploitation...Non volatile.
  - Registres: sert de stockage temporaire. Accès très rapide

3- Programmation assembleur

- Utilisation particulière mémoire: **pile**
  - Zone mémoire accédée que par un registre contenant l'adresse du sommet de pile
  - Accès en empilant/dépilant des valeurs de la pile vers registres (push/pop instructions sur x86)
  - Utilisé pour des appels de fonction (voir cours compilation), pour stocker variables locales à une fonction.
    - Début de fonction: met sur la pile toutes les variables locales
    - Fin de fonction: on dépile toutes les valeurs locales

3- Programmation assembleur

Unités de traitement

- **Unités de contrôle**: responsable décodage instruction à partir de la mémoire
- **Unités fonctionnelles**
  - La plus simple: unité arithmétique et logique (UAL ou ALU), fait tous les calculs arithmétiques et logiques
  - Spécialisés: unité flottante (calcul flottant), voire spécialisés pour certains types d'instructions
  - Il peut y avoir plusieurs unités d'un même type pour calcul parallèle (voir apres)

3- Programmation assembleur

Registres

- “Variables”, zone mémoire rapide pour stockage valeurs temporaires
- Nombre limité, taille en bits fixée. Ensemble des registres: b**anc des registres**
- **Registres de calcul:** généralistes (stocke d'importe quoi), entier ou flottant, vectoriels

3- Programmation assembleur

Registres

- **Registres spéciaux:**
  - Compteur ordinal (program counter): stocke l'adresse de la prochaine instruction à exécuter
  - Registre d'instruction (instruction register): stocke l'instruction en cours
  - Registre de pile: indique le sommet d'une pile en mémoire pour des instructions de manipulation de pile.
  - Registres d'état: divers flags booléens pour indiquer le résultats d'opérations (overflow, resultat d'un test, ...)

3- Programmation assembleur

© Matrix

Instructions

Pas de programmation en binaire directement !

- Le langage assembleur: des instructions **très simples**
- **Pas de variables**: des registres et des cellules mémoire
- Des « fonctions », pas de boucle, pas de if..then..else

Un outil **traduit** ces instructions dans le langage binaire

- C'est l' 'assembleur'
- Une syntaxe pour aider les programmeurs, propre au traducteur

3- Programmation assembleur

© Terminator

Instructions

RISC

reduced instruction set computer

- Jeu limité d'instructions
- Unité de controle simple
- Une instruction est traitée directement par l'unité de controle
- Programmes asm plus longs

CISC

complex instruction set computer

- Jeu d'instruction très étendu
- Unité de controle très compliquée
- Les instructions correspondent à du microcode (un petit programme)
- Programmes courts

3- Programmation assembleur

© Terminator

RISC

Architectures

- MIPS, Sparc, ARM, Power/ PowerPC

S'appuie sur compilateur pour optimisations

Seuls les LOAD/STORE peuvent accéder à la mémoire, les autres opérandes sont des registres

Beaucoup de registres

CISC

Architectures

- X86, 68x, VAX

Maintenant, CISC microcodé comme du RISC.

Beaucoup d'instructions (difficile à toutes utiliser par le compilateur)

3- Programmation assembleur

© Terminator

RISC

Ma,Mb: adresses d'une valeur

r0,r1,r2,r3 registres

```
ld r1 , Ma
ld r2, Mb
ld r3, Mc
mul  r0, r1, r2
add  r0, r0, r3
st Ma, r0
```

CISC

```
mv r1, Ma
mv r2, Mb
fma  r3,r1, r2,  (Mc)
mv  Ma, r3
```

3- Programmation assembleur

© Terminator

L'assembleur de l'Intel x86

- On ne voit qu'une petite partie des instructions
- Instructions 64 bits: l'accès à la mémoire se fait par 64 bits (8 octets), les registres ont 64 bits de large.
- Le manuel de référence:

http://www.intel.com/products/processor/manuals/

Quand a-t-on besoin de l'assembleur ?

- Programmation système d'exploitation, drivers
- Optimisation de performances (vectorisation)
- Vérification de code

3-a Jeu d'instructions x86

- Des registres
- Des instructions d'accès à la mémoire
- Des instructions de calcul
- Des instructions de branchements
- Un peu de syntaxe pour l'outil d'assemblage

3-a Jeu d'instructions x86

Les registres:

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)

3-a Jeu d'instructions x86

Le registre d'état

- Contient l'état du processeur, le résultat des précédents opérations, comparaisons
- Pas directement manipulable

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'incrémentation 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 Vitrual 8086 Mode (bit 17)	
AC Alignement Check (bit 18)	
VIF Virtual Interrupt Flag (bit 19)	
VIP Virtual Interrupt Pending (bit 20)	
ID Identification Flag (bit 21)	

3-a Jeu d'instructions x86

Instructions mémoire

mov *dest,src* **mov** *taille dest,src*	*dest* ← *src*
movzx *dest,src*	extension avec des 0 dans *dest*
movsx *dest,src*	extension avec le bit de signe dans *dest*

Exemples

```
mov r8, 0
```

Met à 0 le registre r8

```
mov r9, 1000
mov r8, [r9]
```

Met 1000 dans r9, puis met le contenu à l'adresse 1000 dans r8

3-a Jeu d'instructions x86

Instructions de déplacement des données

Plusieurs combinaisons:

- **mov** registre, memoire
- **mov** memoire, registre
- **mov** memoire, valeur numérique
- **mov** registre, valeur numérique

3-a Jeu d'instructions x86

Instructions de déplacement des données

Opérations sur la pile

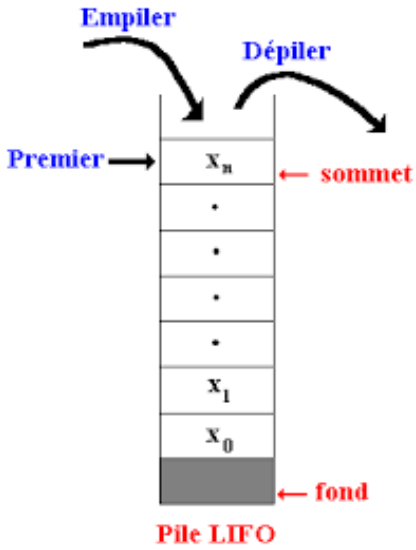
push op: met sur la pile la valeur op (valeur numérique, registre)

pop registre: dépile la valeur du sommet de pile, la met dans le registre.

Registres spéciaux:

rsp: le sommet de la pile. Quand on lance un programme ce compteur est initialisé correctement à un adresse servant de pile.

rbp: sert à marquer un endroit dans la pile



3-a Jeu d'instructions x86

Instructions arithmétiques

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

3-a Jeu d'instructions x86

Instructions arithmétiques

add *op1,op2*	$op1 \leftarrow op1 + op2$

Les opérandes op1 et op2 peuvent être

- Des registres
- Des valeurs immédiates
- Une adresse mémoire

(comme le mov)

3-a Jeu d'instructions x86

Exemple d'instructions arithmétiques

```
mov   rax, 10
mov   rbx, 5
add   rax, rbx
mul 2
dec   rdx
```

Quelles sont les valeurs de rax, rbx, rdx ?

3-a Jeu d'instructions x86

Instructions manipulant des bits

and *op1,op2*	*op1* ← *op1* & *op2*
or *op1,op2*	*op1* ← *op1* | *op2*
xor *op1,op2*	*op1* ← *op1* ^ *op2*
not *reg*	*reg* ← ~*reg*
shl *reg,immédiat*	*reg* ← *reg* << *immédiat*
shr *reg,immédiat*	*reg* ← *reg* >> *immédiat*
sal *reg,immédiat*	*reg* ← *reg* << *immédiat*
sar *reg,immédiat*	*reg* ← *reg* >> *immédiat* signé
rol *reg,immédiat*	*reg* ← *reg* decalageCirculaireGaucheDe *imm*
ror *reg,immédiat*	*reg* ← *reg* decalageCirculaireDroiteDe *imm*

3-a Jeu d'instructions x86

Labels et branchements

Un label sert à désigner une instruction,

Un branchement permet de continuer l'exécution à partir d'une instruction donnée par son label

Exemple

```
        mov rax, 10
L:      dec rax
        jnz L
```

jnz dit qu'il faut réexecuter l'instruction **dec** si rax n'est pas nul.

3-a Jeu d'instructions x86

Instructions de branchement et comparaison

cmp *op1*,*op2*	calcul de *op*1 − *op*2 et de ZF,CF et OF
jmp *op*	branchement inconditionnel à l'adresse *op*
jz *op*	branchement à l'adresse *op* si ZF=1
jnz *op*	branchement à l'adresse *op* si ZF=0
je *op*	branchement à l'adresse *op* si $op1 = op2$
jne *op*	branchement à l'adresse *op* si $op1 \neq op2$
jl *op* (**jnge**)	branchement à l'adresse *op* si $op1 < op2$
jle *op* (**jng**)	branchement à l'adresse *op* si $op1 \leq op2$
jg *op* (**jnle**)	branchement à l'adresse *op* si $op1 > op2$
jge *op* (**jnl**)	branchement à l'adresse *op* si $op1 \geq op2$

3-a Jeu d'instructions x86

Exemple de boucle:

```
        mov rax, 0
        mov rbx, 33
loop:   add rbx, rax
        inc rax
        cmp rax, 100
        jl loop
endloop:
        add rbx, rax
```

3-a Jeu d'instructions x86

Exemple de boucle:

```
        mov rax, 0
        mov rbx, 33
loop:   add rbx, rax
        inc rax
        cmp rax, 100
        jl loop
endloop:
        add rbx, rax
```

```
int rax,rbx;
rbx = 33;
for (rax=0; rax<100; rax++)
     rbx = rbx + rax;
rbx = rbx + rax;
```

3-a Jeu d'instructions x86

A faire: écrire un programme qui calcule la factorielle de n, lorsque n est placé dans le registre rax.

Le résultat sera dans le registre rax.

3-a Jeu d'instructions x86

Les appels de fonction:

call label

appelle une fonction. Met sur la pile la valeur courant de IP.

Fait un

push rip

jmp label

ret : return. Retourne d'une fonction, restaure la valeur de IP qui est sur la pile. Fait un

pop rip

La pile est un élément essentiel pour les appels de fonctions !

3-a Jeu d'instructions x86

Les appels de fonction:

Comment passer les paramètres d'un appel de fonction mafonction(2,3) ?

Comment retrouver les paramètres une fois dans la fonction ?

Et si c'est une fonction que vous n'avez pas écrite ? (ex: printf)

Pour la valeur de retour ?

3-a Jeu d'instructions x86

Les appels de fonction:

Il y a un **protocole**

- Les 6 premiers paramètres doivent être dans `rdi, rsi, rdx, rcx, r8, r9`
- Les suivants doivent être stockés sur la pile

Pour la valeur de retour ? Elle doit être dans `rax`.

3-a Jeu d'instructions x86

Dernier effort pour écrire un programme assembleur: utiliser un assembleur

- **nasm** par exemple

Structure du code minimal:

```
  section .text
global _main
_main:
    Mon code assembleur
    ret
```

3-a Jeu d'instructions x86

Déclarer un tableau, une chaine, des constantes:

```
 section .data
D:  db « hello world », 10, 0 #pour des octets
N:  dq 3 # pour des valeurs de 64 bits
```

Pour les utiliser:

```
 mov rax, [N]        # met 3 dans rax
 mov rbx, D  # met l'adresse de 'hello world' dans rbx
```

3-a Jeu d'instructions x86

Un exemple ! Calcul les puissances de 2.

```
        section .text
global _main
_main:
        mov rax, 1
        mov rbx, 0
        mov rcx, 2
loop:   mul rcx
        inc rbx
        cmp rbx, 8
        jl loop
        ret
```

3-b Schémas de programmation

Pour programmer en assembleur:

- Se baser sur des exemples, le compilateur fournit des codes !

  gcc -S toto.c

- S'appuyer sur des schémas de traduction, du C vers l'assembleur

3-b Schémas de programmation

If..then..else:

```
if (a == b) {
   S1
} else {
   S2
}
```

```
    cmp rax,rbx
    jne Else
    S1
    jmp Endif
Else: S2
Endif:
```

3-b Schémas de programmation

Expressions complexes:

a = (a*b)+(c-d)*2

Décomposition avec des temporaires

```
mul rbx
add rcx,rdx
shl rcx, 2
add rax, rcx
```

3-b Schémas de programmation

while (..) { .. }:

```
while (p) {
  S
}
```

```
Entry: cmp rax, 0
       jeq exit
       S
       jmp Entry
Exit:
```

3-b Schémas de programmation

Tableau d'entiers (4 octets/ élément)

A: dq 0, 0 , 0

A[0] = 1

A[1] = 2

A[2] = 3

```
mov rax, A
mov [rax], 1
add rax, 8
mov [rax], 2
add rax, 8
mov [rax], 3
```

3-b Schémas de programmation

Tableau d'entiers (8 octets/élément) A: dq 0, 0 , 0

A[0] = 1

A[1] = 2

A[2] = 3

```
mov rax, A
mov [rax], 1
mov [rax+8], 2
mov [rax+16], 3
```

Tous les formats d'adresse:

[nombre] ou [label]

[registre]

[registre + nombre]

[registre + registre * scale]

[registre + registre * scale + nombre]

3-b Schémas de programmation

Parcours de tableau

```
for (i=0; i<100; i++) {
  A[i] = 0;
}
```

```
    mov rax, 0
    mov rbx, 100
    mov rcx,A
L:  mov [rcx+rax*8], 0
    inc rax
    cmp rax, rbx
    jl L
```

Calcul avec arithmétique sur pointeurs ! (rcx)

3-b Schémas de programmation

Appel de fonction

```
int f(int a) {
    …
    return …
}
…
  int b
   …
   a = f(b);
…
```

```
f: push rbp
   …# rsi contient a
   pop rbp
   mov rax,…
   ret
   …
   # si rbx contient b, rax  a
   push rbx # sauve rbx
   mov  rsi, rax
   call f
   # rax contient la valeur de retour
   pop rbx # restaure rbx
```

3-b Schémas de programmation

Fonction factorielle en récursif ?

```
int fact(int n) {
  int p;
  if (n<=1) return 1;
  p=n*fact(n-1);
  return p;
}
```

3- Programmation assembleur

Proposez une traduction assembleur
pour le programme

```
for (i=100; i>0; i--) {
   if (x!=0) x = x*2-10;
  else x=x+3;
  A[i+3] = B[i] +x;
}
```

3- Conclusion

A retenir sur la programmation assembleur

- Pas de structures de contrôle évoluées (for, while, if..then..else), que des branchements
  - La programmation assembleur est difficile
  - Programmes assembleurs difficiles à lire
- Pas de structures de données (tableaux, structures)

Nécessaire de s'imposer des règles de programmation (schémas de traduction par ex comme utilisé par compilateur)

Programmation assembleur nécessaire pour

- Développement système d'exploitation, drivers matériels
- Quand compilateur pas assez mur