Ressource R2.04
Communication et fonctionnement bas niveau

Cours n°2
Le jeu d'instruction x86-64

version du 4 février 2024

Plan du cours

le jeu d'instruction x86-64

x86-64

- Version 64 bits de l'ISA x86, proposée par AMD en 1999, compatible avec les précédentes incarnations 8, 16 et 32 bits de l'ISA Intel.
  - De son côté, Intel et HP avaient proposées l'ISA IA-64, incompatible, ce qui a limité fortement sa diffusion.
- Fait passer un cap à l'ISA x86 : adresses sur 64 bits, augmentation du nombre de registres, meilleur support du calcul flottant, ajout d'instructions vectorielles, etc.

Aujourd'hui, c'est le jeu d'instruction principal des processeurs Intel et AMD.

À noter que l'ISA la plus utilisée dans le monde aujourd'hui est sans doute l'ISA A64 de ARM.

Anatomie d'une instruction

Comme dans tous les jeux d'instruction, une instruction x86-64est composée (entre autre)

- d'un opcode : identifie l'opération
  Ex : push, mov, add, call, ret
- d'un format : indique la taille des données manipulées
  Pour les types entiers : b, w, l, q
- d'opérandes sources et/ou destinations : constantes, registres, emplacements en mémoire
  Ex : %rbp, $16, 8(%rsp)

Plan du cours

Plan du cours

Fichiers de registres

Mémoire de quelques centaines d'octets, située dans le processeur, et connectée aux unités fonctionnelles. Adressable par mot mémoire, chaque mot disposant d'un nom spécifié dans le jeu d'instruction.

L'ISA x86-64 spécifie :

- 16 registres de 64 bits pour le calcul sur les entiers
- 16 pseudos-registres de 32 bits pour le calcul sur les entiers
  → 4 octets de poids faible des registres 64 bits
- 8 registres de 8 bits pour la compatibilité 8086 →
  2 octets de poids faible de certains pseudo-registres 32 bits
- des registres de contrôle ou description de l'état de la machine ne peuvent être utilisés comme source ou cible des instructions

Noms des registres x86-64

- `%rax` (`%eax`, `%ah`, `%al`)
- `%rbx` (`%ebx`, `%bh`, `%bl`)
- `%rcx` (`%ecx`, `%ch`, `%cl`)
- `%rdx` (`%edx`, `%dh`, `%dl`)
- `%rsi` (`%esi`)
- `%rdi` (`%edi`)
- `%rsp` (`%esp`, pointeur de pile)
- `%rbp` (`%ebp`, pointeur de base)
- `%r8` (`%r8d`)
- `%r9` (`%r9d`)
- `%r10` (`%r10d`)
- `%r11` (`%r11d`)
- `%r12` (`%r12d`)
- `%r13` (`%r13d`)
- `%r14` (`%r14d`)
- `%r15` (`%r15d`)

Les micro-architectures modernes ont bien plus de registres et incluent un étage de renommage pour que les programmes accèdent aux registres physiques.

Plan du cours

L’espace d’adresage

L’espace d’adressage fait 2^{64} octets : 2^{61} mots de 64 bits (8 octets). C’est un espace virtuel : la machine n’a pas besoin d’avoir autant de mémoire.

Les micro-architectures actuelles utilisent seulement les 48 bits de poids faibles, soit 256 tébioctets adressables.

Les adresses utilisées dans le programme binaire sont virtuelles. L’association entre adresses virtuelles et adresses phyisiques est faite à l’exécution par le système de mémoire virtuelle (collaboration système d’exploitation / micro-architecture).

Modes d'adressage

Une adresse mémoire peut être utilisée comme opérande source ou cible d'une instruction.

Plusieurs options possibles :

- adresse absolue : objets alloués statiquements (code, globales, constantes)
- adresse relative au pointeur de pile : objets alloués automatiquement (variables locales)
- adresse relative à une adresse contenue en mémoire : objets allouées dynamiquements (variables dynamiques, certaines variables locales)

Adresse contenue en mémoire = pointeur !

Les modes d'adressage

Soient $v \in [0, 2^{64}], r, r1, r2 \in Regs, e \in \{1, 2, 4, 8\}$. M (resp. R) est un tableau qui représente la mémoire (resp. le fichier de registres).

Syntaxe	Sémantique	Mode d'adressage
`v`	$M[v]$	Absolu
`(r)`	$M[R[r]]$	Indirect
`v(r)`	$M[v + R[r]]$	Base + déplacement
`(r1, r2)`	$M[R[r_1] + R[r_2]]$	Indexé
`v(r1, r2)`	$M[v + R[r_1] + R[r_2]]$	Indexé
`(,r,e)`	$M[R[r] \times e]$	Indexé avec échelle
`v(,r,e)`	$M[v + (R[r] \times e)]$	Indexé avec échelle
`(r1,r2,e)`	$M[R[r_1] + (R[r_2] \times e)]$	Indexé avec échelle
`v(r1,r2,e)`	$M[v + R[r_1] + (R[r_2] \times e)]$	Indexé avec échelle

Note : `$v` désigne la constante entière `v`.

Modes d'adressage : exemples

Soit l'état partiel de la machine :

R

`%rax`	`0xff …ffab10`
`%rdx`	`0x4`

M

`0xff …ffab10`	`0x1`
`0xff …ffab18`	`0x10`
`0xff …ffab20`	`0x100`
`0xff …ffab28`	`0x1000`
`0xff …ffab30`	`0x2`
`0xff …ffab38`	`0x20`

Complétez le tableau :

Expression	Sémantique	Valeur
`(%rax)`	$M[R[\%rax]]$	1
`0x10(%rax)`	$M[R[\%rax] + 0x10]$	
`(%rax, %rdx)`		
	$M[R[\%rax] + 2 \times R[\%rdx]]$	

Plan du cours

Placement des données en mémoire

1) Les objets du programmes sont en mémoire.

Objet = variables, fonctions.

Le nom donné à l'objet dans le code source est appelé symbole. Chaque symbole est associée à une adresse pour toute sa durée de vie :

- à l'édition des liens : symboles statiquements,
- à l'exécution : symboles automatiques et dynamiques.

2) Même avec une ISA CISC, la majorité des opérations agissent uniquement sur les registres.

Les opérations de copie

1) + 2) $\Longrightarrow$ les programmes doivent sans cesse copier les données entre mémoire et registres

- un compilateur optimisant minimiser ces mouvements : propagation de constante, association de symbole à des registres, etc.

- `mov src, dst` entre mémoire et registre
- `push reg` ou `pop reg` entre pile et registres
- `push reg` correspond à `mov reg, (%rbp)` suivi de `sub $8, %rbp`

Les combinaisons source / destination

Toutes les combinaisons ne sont pas autorisées.

	src	dest	exemple	en C
`movl`	Constante	Registre	`movq $0xA, %rax`	
		Mémoire	`movq $0xB, (%rax)`	
	Registre	Registre	`movq $rax, %rdx`	
		Mémoire	`movq $rdx, (%rax)`	
	Mémoire	Registre	`movq ($rax), %rdx`	

Quand source et destination n'ont pas la même taille

Quand source et destination n'ont pas la même taille, deux variantes :

- `movs` complète la destination en copiant le bit de poids fort de la source,
- `movz` complète avec des 0.

Notes :

- marche seulement si la source est plus petite que la destination
- deux suffixes de taille sont utilisées, p.ex. `movzbl` ou `movslq`.

‘

Exemple

```
func swap (p1 *int32, p2 *int64) {
    tmp1 := *p1
    tmp2 := *p2
    *p2 = int64(tmp1)
    *p1 = int32(tmp2)
}
```

`go build -gcflags='-l'` puis `objdump -d mov > mov.d`

```
main.swap:
  movslq (%rax),%rcx
  mov    (%rbx),%rdx
  mov    %rcx,(%rbx)
  mov    %edx,(%rax)
  ret
```

Plan du cours

Appel de procédures, pile, cadre

Fonction/procédure du code source → procédure du code machine

- attention, selon le langage, les deux objets peuvent avoir des noms $\neq$

Appel et exécution de procédure dans le code machine utilisent

Pile et cadres (2)

`main` appelle `f1`, qui appelle `f2`, qui appelle `f3`.

```
-------- <- base de la pile
| main |
--------
|  f1  |
--------
|  f2  |
-------- <- %rbp (début cadre f3)
|  f3  |
-------- <- %rsp (sommet)
```

Une même procédure peut-elle avoir plusieurs cadres vivants au même instant dans la pile ?

Principe général de l'appel de procédure

Dans l'ABI x86-64, l'appel de procédure repose sur deux opérations :

Appel : `call symbole`

- empile l'adresse de l'insruction suivante
- charge `%rip` avec l'adresse associée à `symbole`

Retour : `ret`

- dépile vers `%rip`
- l'instruction exécutée juste après est donc celle dont l'adresse a été empilée par `call`

Pourquoi le retour de procédure utilise la pile plutôt qu'un « branchement » vers un symbole ?

Illustration : call

```
47aec4: e8 97 ff ff ff          call   47ae60 <main.swap>
```

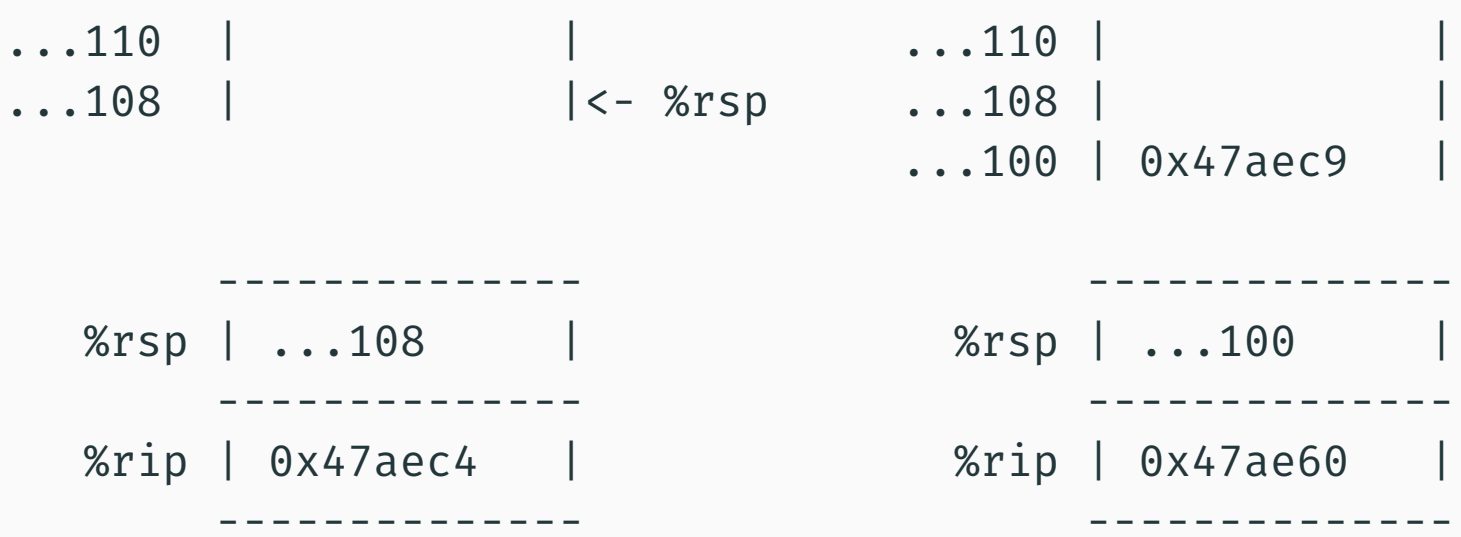
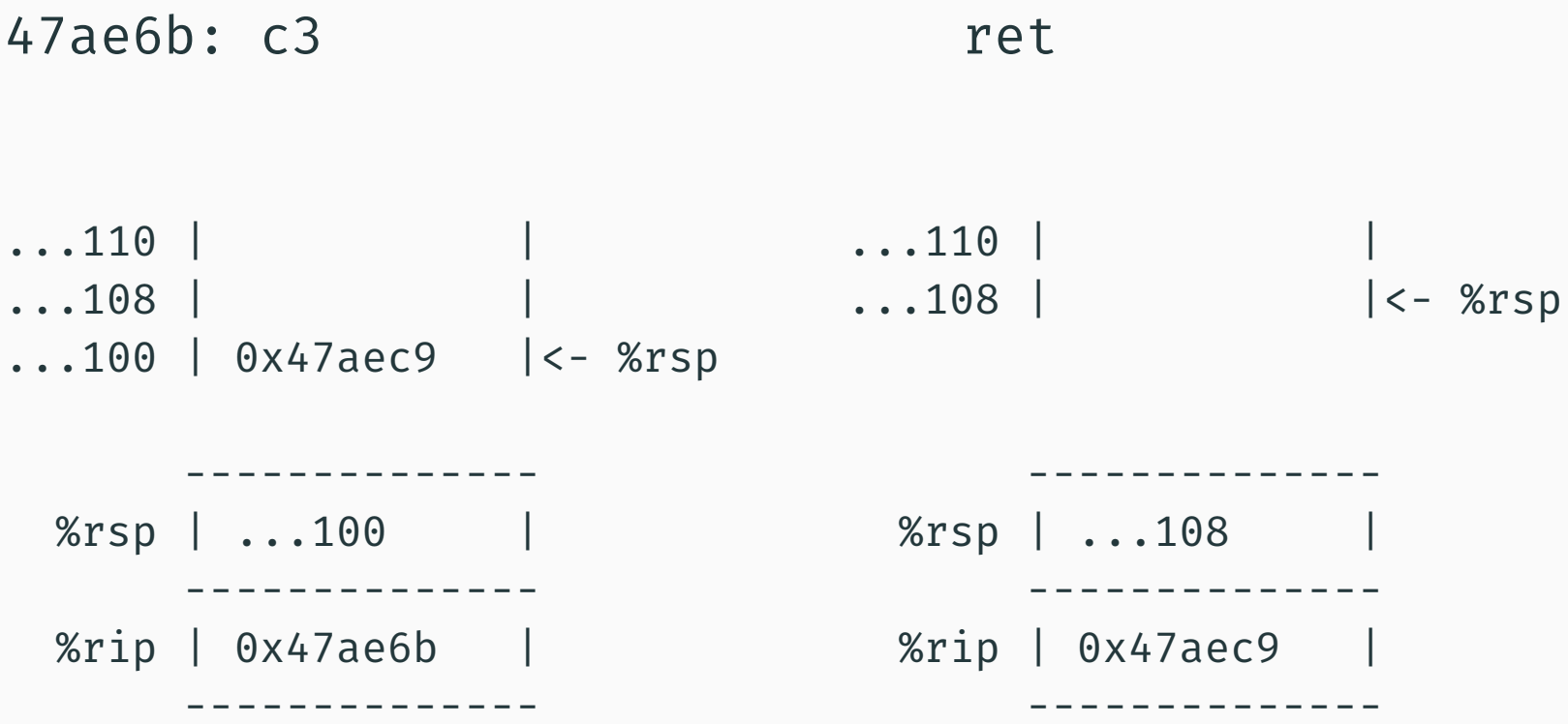


Illustration : ret



Allocation du cadre

Premières instructions d'une procédure (= prologue) : allocation du cadre

1. empile la valeur courante de `%rbp`
2. fait pointer `%rbp` sur le nouveau sommet de la pile
3. soustrait à `%rsp` la taille du cadre à allouer (calculée par le compilateur)

Dernières instructions (épilogue) : libération du cadre

1. ajoute à `%rsp` la taille du cadre alloué
2. dépile `%rbp`

Note : le code optimisé simplifie les choses en n'utilisant pas de pointeur de base.

Illustration : prologue

```
080484c4 <main.swap>:
  47ae60: 55                    push   %rbp
  47ae61: 48 89 e5              mov    %rsp,%rbp
  47ae64: 48 83 ec 10           sub    $0x10,%rsp
```

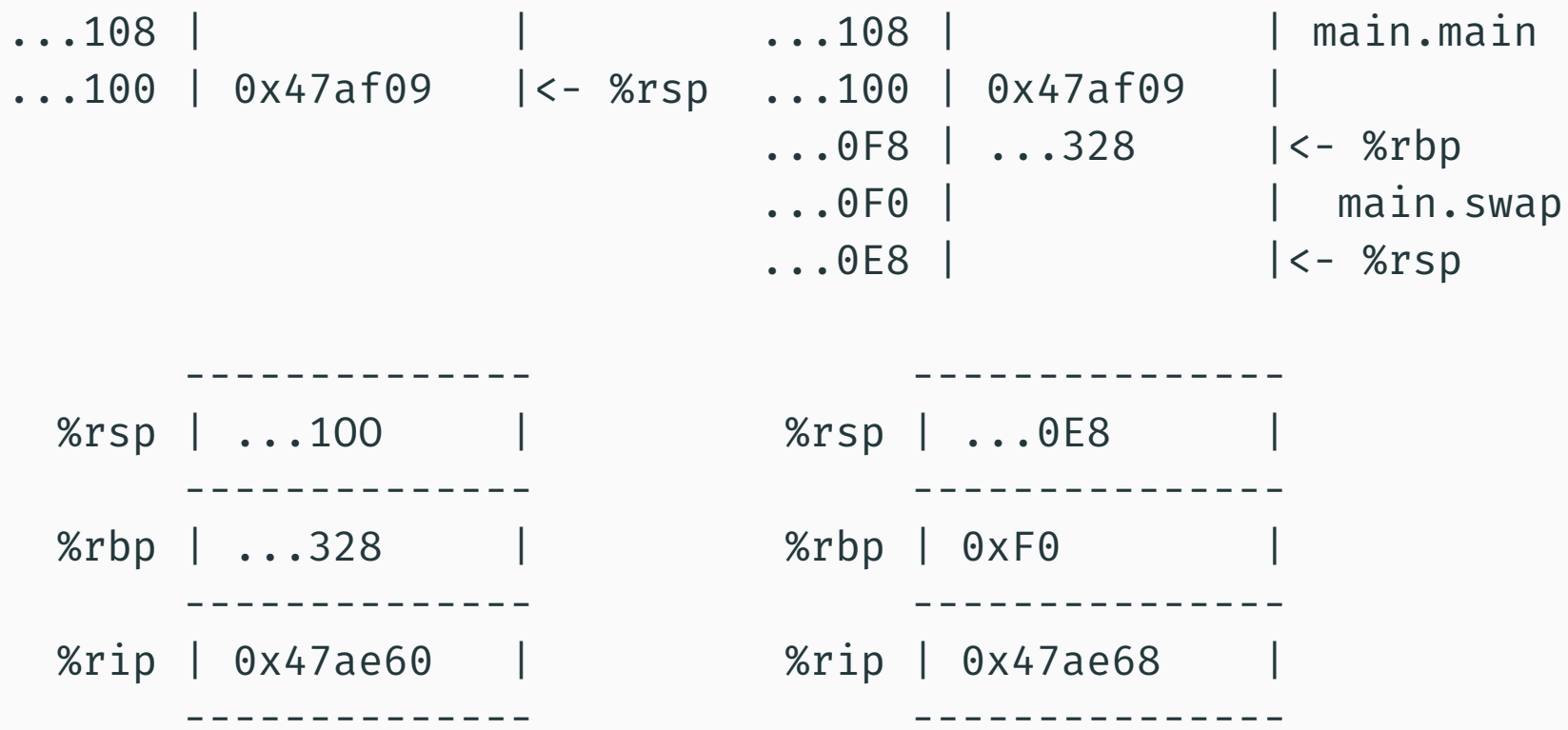


Illustration : épilogue

```
47ae9c: 48 83 c4 10             add    $0x10,%rsp
47aea0: 5d                      pop    %rbp
47aea1: c3                      ret
```

```
...108 |           |        ...108 |           |
...100 | 0x47af09  |        ...100 | 0x47af09  |<- %rsp
...0F8 | ...328    |<- %rbp
...0F0 | 0x120     |
...0E8 |           |<- %rsp
```

```
     --------------              --------------
 %rsp | ...0E8    |          %rsp | ...100    |
     --------------              --------------
 %rbp | ...0F8    |          %rbp | ...328    |
     --------------              --------------
 %rip | 0x47ae9c  |          %rip | 0x47aea1  |
     --------------              --------------
```

Passage des paramètres et codes de retour

Dans l'ABI x86-64, les paramètre et les codes de retour sont passés

- via les registres, en commençant par `%rax` puis `%rbx` puis ...
- au-delà de 10, ils sont passés par la pile

En plus de cette convention, plusieurs langage (dont go) assimilent les paramètres à des variables locales

- allouées par l'appelant
- mais initialisées et utilisées par l'appelé

En go, les codes de retour sont aussi associés à des locales, allouées, initialisées et utilisées par l'appelé.

Sauvegarde et restauration des registres

L'appel de procédure ne doit pas interférer sur l'exécution de la fonction appelante → en dehors des codes de retour, l'état de la machine doit être restauré.

Si l'appelant à des valeurs utiles dans les registres porteurs des codes de retour → empile avant l'appel, dépile (dans l'ordre inverse) après l'appel

Pour les autres registres, c'est à l'appelé de faire le travail : empile juste après le prologue, dépile (dans l'ordre inverse) juste après l'épilogue.

Optimisations

Un compilateur optimisant simplifie souvent beaucoup le protocole d'appel :

- pointeur de base inutile (le compilateur connaît la taille du cadre)
- allocations de locales pour les paramètres / codes de retour le plus souvent inutiles
- si aucune locale, aucun cadre alloué
- appel supprimé par *inlining* des fonctions dans le code de l'appelant
- instructions supprimées quand c'est possible par propagation des constantes

L'analyse d'un code optimisé nécessite d'avoir un esprit « ouvert »

Plan du cours

Registre de code de conditions

Les unités fonctionnelles récoltent des informations sur les résultats.

Par exemple, pour l'UAL :

- résultat nul
- retenue sortante
- débordement
- parité
- ...

Ces informations sont accessibles depuis le registre de codes de condition (ou CCR pour *Conditiion Code Register*)

Le registre RFLAGS

Le CCR de l'ISA x86-64 se nomme **RFLAGS**.

22 bits utilisés pour stocker des codes de condition (appelées *flag*).

Principaux flags : **CF**, **ZF**, **PF**, **OF**, **SF**

Mise à jour de RFLAGS

Les bits de RFLAGS sont mis à jour à chaque instruction arithmétique et logique.

Les bits non touchés par une opération sont laissés à la valeur précédente.

x86-64 propose également des opérations « sans destination » qui mettent à jour RFLAGS, par exemple :

- `cmp a, b` : calcule $b - a$
- `test a, b` : calcule $a \wedge b$ (bit à bit)
- `bt r, n` : copie le nième bit de `r` dans CF

Les conditions de l'ISA x86-64

Le jeu d'instruction propose un ensemble de conditions évaluées par lecture de RFLAGS

Symbole	Nom	Évaluation
`a`	above	$\neg(cf \vee zf)$
`b`	below	cf
`g`	greater	$\neg(sf \oplus of) \wedge \neg zf$
`l`	lower	$sf \oplus of$
`e`, `z`	equal, zero	zf
`ge`	greater or equal	$\neg(sf \oplus of)$
`ne`	not equal	$\neg zf$
...		

Quelle est la différence entre `a` et `g` ? Entre `b` et `l` ?

Les opérations conditionnelles

Les conditions sont utilisées par des opérations conditionnelles.

Opération conditionnelle
Opération dont le comportement dépend de la valeur d'une condition.

x86-64 propose 2 groupes d'opérations conditionnelles :

- `setxxx r` : $r \leftarrow 1$ si `xxx` et 0 sinon.
- `jxxx adr` : saute à l'adresse `adr` si `xxx`, ne fait rien sinon.
- `cmovxxx a, b` : $b \leftarrow a$ si `xxx`.

Plan du cours

Principe des sauts

Par défaut, le processeur exécute les instructions dans l'ordre où elles sont rangées en mémoire.

Certaines instructions permettent de changer cet ordre, en modifiant arbitrairement la valeur du pointeur d'instruction. On parle de branchement, ou saut (vocabulaire x86-64).

Déjà rencontrées : `call adr` et `ret`, utilisées pour l'appel de procédure.

`jmp` et `jxxx`, où `xxx` est une condition. Utilisées pour traduire les structures de contrôle : conditionnelles, boucles.

Traduction d'une conditionnelle simple

```
func f1(a, b int) (r int) {
    if a >= b {
        r = a-b
    } else {
        r = b-a
    }
    return
}
```

Note : le code optimisé utilise `cmovxxx` et aucun saut. Pourquoi ?

```
push   %rbp
mov    %rsp,%rbp
sub    $0x8,%rsp
mov    %rax,0x18(%rsp)
mov    %rbx,0x20(%rsp)
movq   $0x0,(%rsp)
cmp    %rbx,%rax
jge    47c5e2
jmp    47c5eb
sub    %rbx,%rax  # 47c5e2
mov    %rax,(%rsp)
jmp    47c5f4
sub    %rax,%rbx  # 47c5eb
mov    %rbx,(%rsp)
jmp    47c5f4
mov    (%rsp),%rax # 47c5f4
add    $0x8,%rsp
pop    %rbp
ret
```

Traduction d'une boucle simple

```
func f2(x  int) (r int) {
    r = 1
    for cpt := 1; cpt < x;
                  cpt+=2 {
        r *= cpt
    }
    return
}
```

```
mov    $0x1,%ecx
mov    $0x1,%edx
jmp    47c5f7
lea    0x2(%rcx),%rbx  # 47c5ec
imul   %rcx,%rdx
mov    %rbx,%rcx
cmp    %rcx,%rax  # 47c5f7
jg     47c5ec
mov    %rdx,%rax
nop
ret
```

Traduction d'une conditionnelle complexe

```
func f3(x int) (r int) {
    x = x % 6
    switch(x) {
    case 0:
        r = 2*x
    case 1:
        r = 1/x
    case 2:
        r = x*x
    case 3:
        r = x/2
    case 4:
        r = x+3
    case 5:
        r = x-7
    }
    return
}
```

Cf. fichier.

Plan du cours

Bilan

Ce qu'on a vu et qui doit maintenant être connu :

- Rappel sur la compilation et la création d'exécutable
- Principes de base de l'organisation d'un ordinateur
- Zoom sur l'unité centrale, notion de registre, d'instruction, d'opérations
- Organisation de la mémoire d'un processus (sections)
- Faire le lien entre code source et programme binaire

Conclusions

Quelques leçons pour la suite de vos études et votre carrière :

- Pour optimiser votre code, en premier lieu, faites confiance au compilateur
- Pour bien comprendre la sémantique des langages, il faut penser à la traduction en code machine
- L'étude approfondie des architectures matérielles et de l'interface matérielle-logicielle est au centre de plusieurs sujets :
  - performances « avancées »
  - cybersécurité
  - systèmes d'exploitation
  - informatique temps réel
  - …

Si vous souhaitez approfondir le sujet, il y a quelques ouvrages recommandables à la bibliothèque.