Compilation

Introduction à l'architecture x86-64

Adrien Guatto

Master 1 Informatique
2022–2023

Plan

1 Introduction

2 Les bases du jeu d'instructions

3 Convention d'appel et gestion de la pile

4 Conclusion

Plan

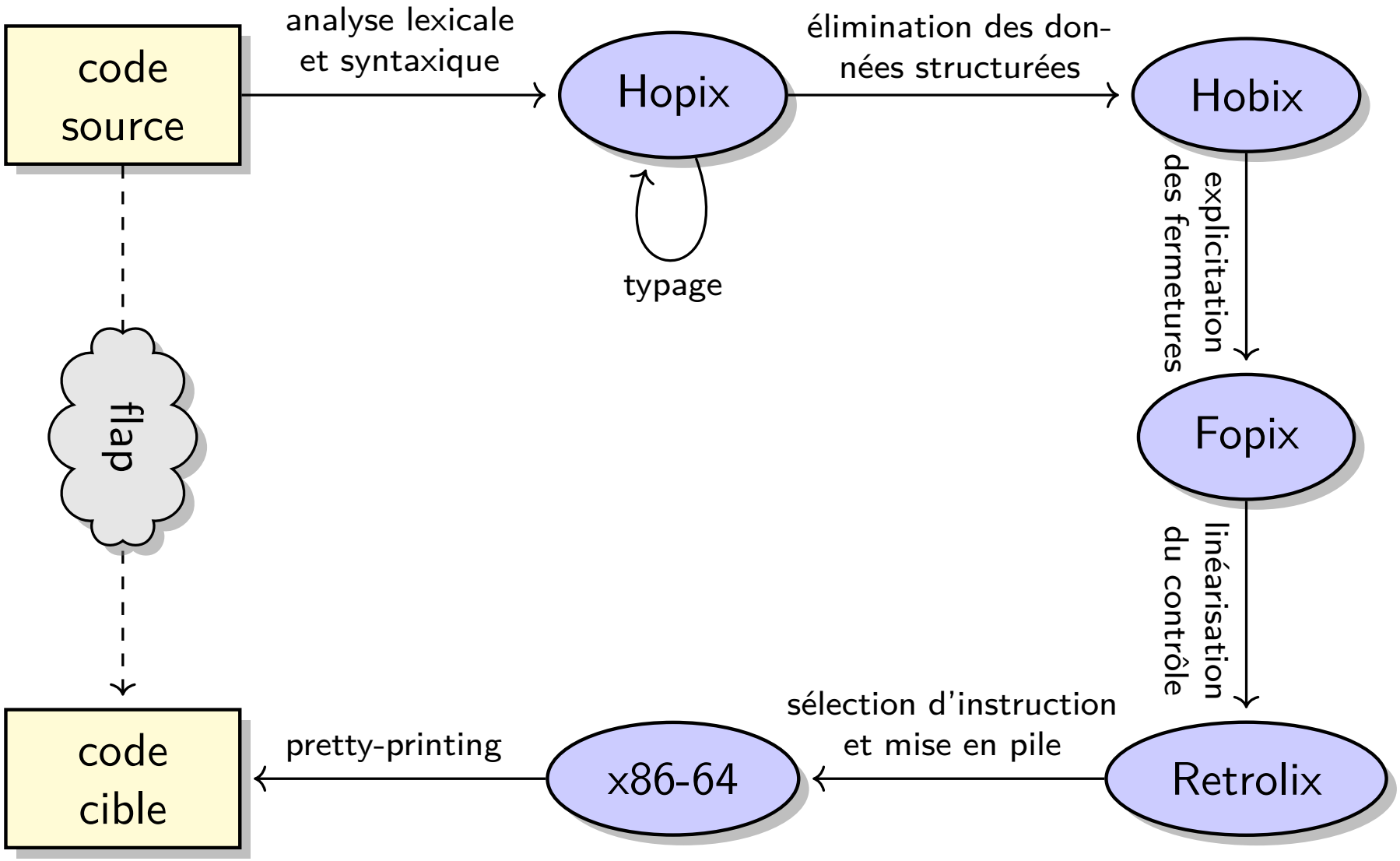
1 Introduction

2 Les bases du jeu d'instructions

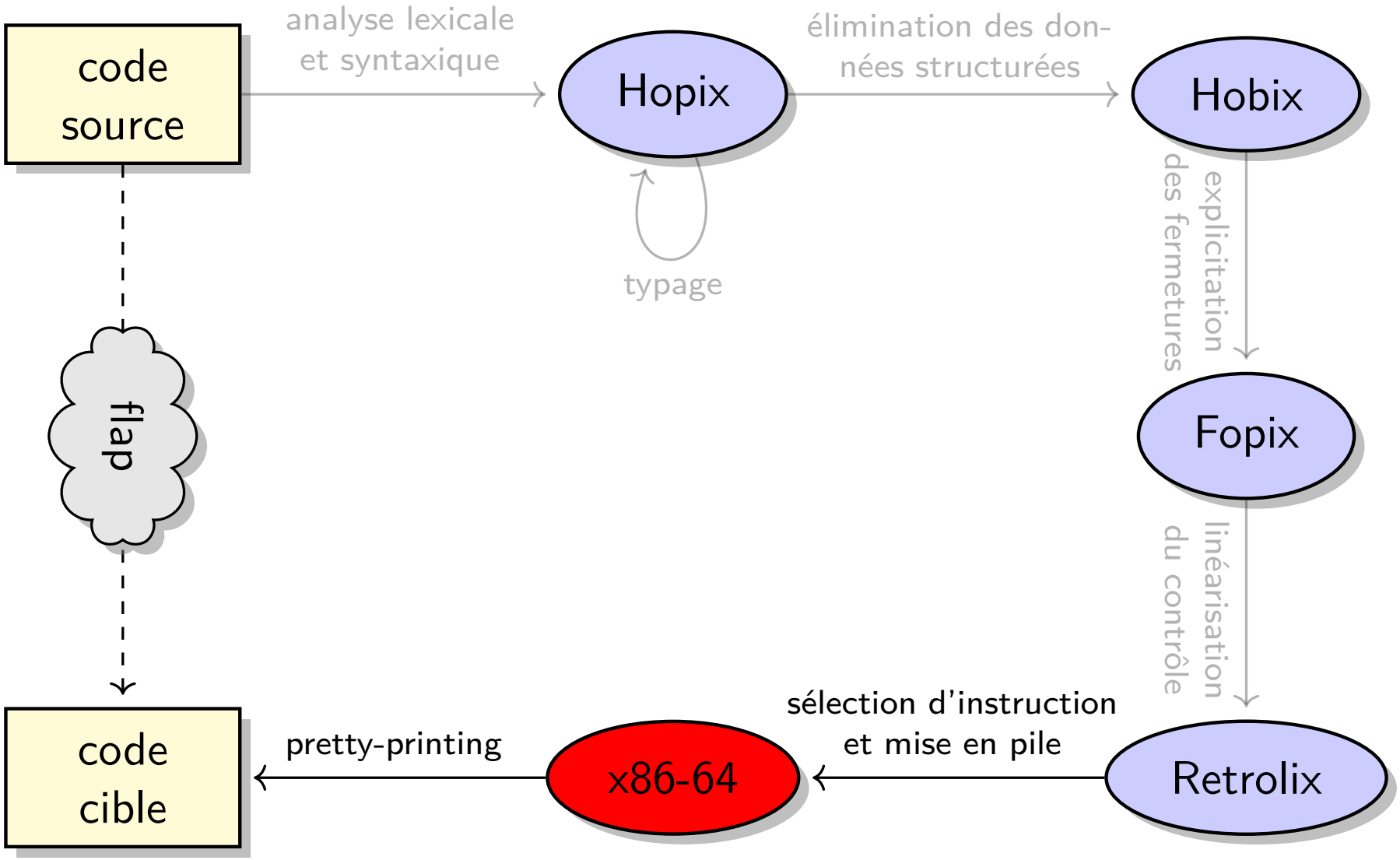
3 Convention d'appel et gestion de la pile

4 Conclusion

Le flot de compilation de flap



Le flot de compilation de flap



Introduction à x86-64

Pourquoi cette introduction ?

- Vous donner les armes pour faire la traduction Retrolix vers x86-64.
- Enrichir votre culture dans un pan omniprésent de l'informatique
- Mieux comprendre le système. Utile pour déboguer, optimiser.

Dans un cours d'architecture des ordinateurs, on aborde le sujet avec des lunettes de concepteurs et implémenteurs d'architecture. Aujourd'hui, nous sommes de purs utilisateurs, c'est-à-dire des programmeurs bas-niveau !

La méthode

- Je décris les grandes lignes et quelques détails de l'architecture.
- On illustre par des programmes en assembleur, sous Linux.

De quoi parle-t-on ?

Ne pas confondre architecture et micro-architecture.

De quoi parle-t-on ?

Ne pas confondre architecture et micro-architecture.

Architecture

- Interface de programmation de bas niveau.
- Spécifie un jeu d'instruction au format binaire (*langage machine*).
- Exemples : x86-64, ARMv8, RISC-V.

De quoi parle-t-on ?

Ne pas confondre architecture et micro-architecture.

Architecture

- Interface de programmation de bas niveau.
- Spécifie un jeu d'instruction au format binaire (*langage machine*).
- Exemples : x86-64, ARMv8, RISC-V.

Micro-architecture

- Implémentation (ou famille d'implémentations) d'une architecture.
- Peut exécuter tous les programmes écrits pour l'architecture.
- Exemples : Zen 3 (AMD) implémente x86-64, Vortex/Tempest (Apple) implémente ARMv8, U8 (SiFive) implémente RISC-V.

Contexte

On distingue traditionnellement deux types d'architecture : RISC et CISC.

Contexte

On distingue traditionnellement deux types d'architecture : RISC et CISC.

Reduced Instruction Set Computer

Un petit nombre d'instructions simples et orthogonales, ce qui permet de simplifier la micro-architecture. Exemple : RISC-V, ARM (historiquement).

Contexte

On distingue traditionnellement deux types d'architecture : RISC et CISC.

Reduced Instruction Set Computer

Un petit nombre d'instructions simples et orthogonales, ce qui permet de simplifier la micro-architecture. Exemple : RISC-V, ARM (historiquement).

Complex Instruction Set Computer

Beaucoup d'instructions baroques, des redondances. Micro-architecture complexe au niveau du décodage. Exemple paradigmatique : x86-32, x86-64 ; les ARM modernes s'en rapprochent.

Contexte

On distingue traditionnellement deux types d'architecture : RISC et CISC.

Reduced Instruction Set Computer

Un petit nombre d'instructions simples et orthogonales, ce qui permet de simplifier la micro-architecture. Exemple : RISC-V, ARM (historiquement).

Complex Instruction Set Computer

Beaucoup d'instructions baroques, des redondances. Micro-architecture complexe au niveau du décodage. Exemple paradigmatique : x86-32, x86-64 ; les ARM modernes s'en rapprochent.

Nous nous intéressons à l'architecture x86-64.

L'architecture x86-64/AMD64/EM64T/Intel64

Le fruit d'une longue évolution.

- Proposé par AMD en 2000, implémenté en 2003 (AMD K8).
- Évolution de x86-32, implémenté en 1985 (Intel 80386).
- Évolution de x86-16, implémenté en 1978 (Intel 8086).
- Évolution de l'Intel 8008, produit en 1972. Processeur 8 bits !

Vous imaginez l'accumulation de détails ! Certains ont pu être simplifiés.

L'architecture x86-64/AMD64/EM64T/Intel64

Le fruit d'une longue évolution.

- Proposé par AMD en 2000, implémenté en 2003 (AMD K8).
- Évolution de x86-32, implémenté en 1985 (Intel 80386).
- Évolution de x86-16, implémenté en 1978 (Intel 8086).
- Évolution de l'Intel 8008, produit en 1972. Processeur 8 bits !

Vous imaginez l'accumulation de détails ! Certains ont pu être simplifiés.

Produire du code x86-64 dans un compilateur pédagogique ?

- Inconvénients : laid, baroque, complexe.
- Avantages : réaliste ; fonctionne sur tous les PC.

L'architecture x86-64/AMD64/EM64T/Intel64

Le fruit d'une longue évolution.

- Proposé par AMD en 2000, implémenté en 2003 (AMD K8).
- Évolution de x86-32, implémenté en 1985 (Intel 80386).
- Évolution de x86-16, implémenté en 1978 (Intel 8086).
- Évolution de l'Intel 8008, produit en 1972. Processeur 8 bits !

Vous imaginez l'accumulation de détails ! Certains ont pu être simplifiés.

Produire du code x86-64 dans un compilateur pédagogique ?

- Inconvénients : laid, baroque, complexe.
- Avantages : réaliste ; fonctionne sur tous les PC.

Réferences

- Manuels officiels x86-64 (Intel Corporation 2023).
- *Notes on x86-64 Programming*, Tolmach (2012). **À lire**.

Décrire abstraitement une architecture

On peut identifier une architecture à un système de transitions

$$(\mathbb{S} : \mathbf{Set}, T : \mathbb{S} \nrightarrow \mathbb{S})$$

formé d'un ensemble d'états $\mathbb{S}$ et d'une relation T dite "de transition".

- Un élément de $\mathbb{S}$ est un état de la machine complète.
  - L'état du processeur et des périphériques, le contenu de la mémoire...
- La relation de transition T décrit l'évolution de la machine.
  - C'est une *relation* car un état peut avoir zéro ou plusieurs successeurs.

Décrire abstraitement une architecture

On peut identifier une architecture à un système de transitions

$$(\mathbb{S} : \mathbf{Set}, T : \mathbb{S} \nrightarrow \mathbb{S})$$

formé d'un ensemble d'états $\mathbb{S}$ et d'une relation T dite "de transition".

- Un élément de $\mathbb{S}$ est un état de la machine complète.
  - L'état du processeur et des périphériques, le contenu de la mémoire…
- La relation de transition T décrit l'évolution de la machine.
  - C'est une *relation* car un état peut avoir zéro ou plusieurs successeurs.

L'architecture x86-64, pour nous

- $\mathbb{S}$ = registres + mémoire = registres + tas + pile + code.
- La mémoire est un tableau dont certaines cases sont inaccessibles.
- Modèle de von Neumann : compteur d'instruction (registre `%rip`).
- La relation T fait évoluer l'état selon l'instruction courante.

Programmer en assembleur

- L'architecture ne spécifie que le format binaire des instructions.
- Deux grands formats textuels : Intel et AT&T. On utilisera AT&T.
  - Le format Intel est utilisé par la documentation Intel, sans surprise.
  - Le format AT&T est celui des outils GNU et notes d'Andrew Tolmach.

Programmer en assembleur

- L'architecture ne spécifie que le format binaire des instructions.
- Deux grands formats textuels : Intel et AT&T. On utilisera AT&T.
  - Le format Intel est utilisé par la documentation Intel, sans surprise.
  - Le format AT&T est celui des outils GNU et notes d'Andrew Tolmach.

```
# Mon premier programme assembleur
.global _start
_start: mov $60, %rax
        mov $42, %rdi
        syscall
```

Programmer en assembleur

- L'architecture ne spécifie que le format binaire des instructions.
- Deux grands formats textuels : Intel et AT&T. On utilisera AT&T.
  - Le format Intel est utilisé par la documentation Intel, sans surprise.
  - Le format AT&T est celui des outils GNU et notes d'Andrew Tolmach.

```
# Mon premier programme assembleur
.global _start
_start: mov $60, %rax
        mov $42, %rdi
        syscall
```

```
$ as -o test.o test.S # Traduit du langage d'assemblage en langage machine
$ ld test.o -o test   # Produit un exécutable autonome
$ ./test; echo $?     # Lance l'exécutable et affiche son code de retour
42
```

Utiliser GCC pour assembler

Le code assembleur produit par Flap utilise la bibliothèque standard C.

```
.global _start
_start: mov $42, %rdi
        call exit
```

Utiliser GCC pour assembler

Le code assembleur produit par Flap utilise la bibliothèque standard C.

```
.global _start
_start: mov $42, %rdi
        call exit
```

```
$ as -o test2.o test2.S
$ ld test2.o -o test2
ld: test2.o : dans la fonction « _start » :
(.text+0x8): undefined reference to `exit'
$ ld test2.o -o test2 -lc
$ ./test2
bash: ./test2 : ne peut exécuter : le fichier requis n'a pas été trouvé
$ ld test2.o -o test2 -lc --dynamic-linker /lib/ld-linux-x86-64.so.2 # OK !
$ gcc -o test2 test2.S
/usr/bin/ld : /tmp/cchqTU0V.o : dans la fonction « _start » :
(.text+0x0) : définitions multiples de « _start »; /usr/lib/Scrt1.o:(.text+0x0) : défini
/usr/bin/ld: /usr/lib/Scrt1.o : dans la fonction « _start » :
(.text+0x1b): undefined reference to `main'
collect2: erreur: ld a retourné le statut de sortie 1
$ sed -ie 's/_start/main/' test2.S
$ gcc -o test2 test2.S # Nettement plus simple !
```

Plan

1 Introduction

2 Les bases du jeu d'instructions

3 Convention d'appel et gestion de la pile

4 Conclusion

Les registres, l'instruction mov

- Petites zones de stockage situées directement sur le processeur.
- En x86-64, on dispose de seize *registres généraux* de 64 bits.
  - RAX, RBX, RCX, RDX, RBP, RSP, RDI, RSI, R8-R15.
  - En syntaxe AT&T, on préfixe leur nom d'un signe % (e.g. `%rdi`).
- On dispose également d'*alias* de registres 32, 16 et 8 bits.
  - E.g., %al désigne les 8 bits de poids faibles de %ax.
  - E.g., %ax désigne les 16 bits de poids faibles de %eax.
  - E.g., %eax désigne les 32 bits de poids faibles de %rax.
- Il existe de plus des *registres spéciaux* en lecture seule. Notamment :
  - `%rip` contient l'adresse de la prochaine instruction à exécuter ;
  - `%rflags` est modifié implicitement par les instructions arithmétiques.

La copie

L'instruction `mov SRC, DST` copie `SRC` dans `DST`.

- `mov %rdi, %r8` : copie le contenu de RDI dans R8.
- `mov %ebx, %rsp` : copie les 32 bits de poids faible de RBX dans RSP.

Instructions et opérandes

Un opérande d'instruction est soit :

- un registre général ;
- une *valeur immédiate*, constante littérale sur 32 bits maximum ;
  - (Préfixée par `$` en syntxe AT&T, e.g., `mov $42, %rdi`.)
- une référence au contenu de la mémoire.

Beaucoup d'instructions ont des variantes selon la taille des données :

- `mov`**`b`** `SRC, DST` : copie un ***b****yte* (8 bits) ;
- `mov`**`w`** `SRC, DST` : copie un ***w****ord* (16 bits) ;
- `mov`**`l`** `SRC, DST` : copie un ***l****ong-word* (32 bits) ;
- `mov`**`q`** `SRC, DST` : copie un ***q****uad-word* (64 bits) ;

En l'absence de suffixe, GNU as infère en fonction des opérandes.

La mémoire et les références mémoires

- Grande zone de mémoire située à l'extérieur du processeur.
- Un tableau adressable à l'octet par des adresses 64 bits.
- (De la mémoire *virtuelle* gérée par le SE. Voir un cours de système.)

On désigne une case via une référence mémoire de la forme

$$O(B, I, S) \quad \text{qui désigne l'adresse} \quad O + B + I \cdot S$$

où B et I sont des registres, O et S des immédiats et S une puissance de 2. On peut omettre O, I et S.

Que font les instructions suivantes, à votre avis ?

- `movq $42, (%rax)`
- `movq %rax, -8(%rbp)`
- `movq %rbx, (%rcx, %rdx, 4)`
- `movq (%rax), (%rbx)`

La mémoire et les références mémoires

- Grande zone de mémoire située à l'extérieur du processeur.
- Un tableau adressable à l'octet par des adresses 64 bits.
- (De la mémoire *virtuelle* gérée par le SE. Voir un cours de système.)

On désigne une case via une référence mémoire de la forme

$$O(B, I, S) \quad \text{qui désigne l'adresse} \quad O + B + I \cdot S$$

où B et I sont des registres, O et S des immédiats et S une puissance de 2. On peut omettre O, I et S.

Que font les instructions suivantes, à votre avis ?

- `movq $42, (%rax)` : $Mem[\%\texttt{rax}] \leftarrow 42$
- `movq %rax, -8(%rbp)`
- `movq %rbx, (%rcx, %rdx, 4)`
- `movq (%rax), (%rbx)`

La mémoire et les références mémoires

- Grande zone de mémoire située à l'extérieur du processeur.
- Un tableau adressable à l'octet par des adresses 64 bits.
- (De la mémoire *virtuelle* gérée par le SE. Voir un cours de système.)

On désigne une case via une référence mémoire de la forme

$$O(B, I, S) \quad \text{qui désigne l'adresse} \quad O + B + I \cdot S$$

où B et I sont des registres, O et S des immédiats et S une puissance de 2. On peut omettre O, I et S.

Que font les instructions suivantes, à votre avis ?

- `movq $42, (%rax)` : $Mem[\%\texttt{rax}] \leftarrow 42$
- `movq %rax, -8(%rbp)` : $Mem[\%\texttt{rbp} - 8] \leftarrow \%\texttt{rax}$
- `movq %rbx, (%rcx, %rdx, 4)`
- `movq (%rax), (%rbx)`

La mémoire et les références mémoires

- Grande zone de mémoire située à l'extérieur du processeur.
- Un tableau adressable à l'octet par des adresses 64 bits.
- (De la mémoire *virtuelle* gérée par le SE. Voir un cours de système.)

On désigne une case via une référence mémoire de la forme

$$O(B, I, S) \quad \text{qui désigne l'adresse} \quad O + B + I \cdot S$$

où B et I sont des registres, O et S des immédiats et S une puissance de 2. On peut omettre O, I et S.

Que font les instructions suivantes, à votre avis ?

- `movq $42, (%rax)` : $Mem[\%\texttt{rax}] \leftarrow 42$
- `movq %rax, -8(%rbp)` : $Mem[\%\texttt{rbp} - 8] \leftarrow \%\texttt{rax}$
- `movq %rbx, (%rcx, %rdx, 4)` : $Mem[\%\texttt{rcx} + 4 \cdot \%\texttt{rdx}] \leftarrow \%\texttt{rbx}$
- `movq (%rax), (%rbx)`

La mémoire et les références mémoires

- Grande zone de mémoire située à l'extérieur du processeur.
- Un tableau adressable à l'octet par des adresses 64 bits.
- (De la mémoire *virtuelle* gérée par le SE. Voir un cours de système.)

On désigne une case via une référence mémoire de la forme

$$O(B, I, S) \quad \text{qui désigne l'adresse} \quad O + B + I \cdot S$$

où B et I sont des registres, O et S des immédiats et S une puissance de 2. On peut omettre O, I et S.

Que font les instructions suivantes, à votre avis ?

- `movq $42, (%rax)` : $Mem[\%\texttt{rax}] \leftarrow 42$
- `movq %rax, -8(%rbp)` : $Mem[\%\texttt{rbp} - 8] \leftarrow \%\texttt{rax}$
- `movq %rbx, (%rcx, %rdx, 4)` : $Mem[\%\texttt{rcx} + 4 \cdot \%\texttt{rdx}] \leftarrow \%\texttt{rbx}$
- `movq (%rax), (%rbx)` : **erreur d'assemblage** (2 opér. mémoire)

Les opérations arithmétiques et logiques

Instruction	Sémantique	Remarques
`not*` d	$d \leftarrow \neg d$	bit à bit
`and*` s, d	$d \leftarrow d \wedge s$	bit à bit
`or*` s, d	$d \leftarrow d \vee s$	bit à bit
`xor*` s, d	$d \leftarrow d \oplus s$	bit à bit
`sal*` i, d	$d \leftarrow d \ll i$	i immédiat
`sar*` i, d	$d \leftarrow d \gg i$	i immédiat ; décalage arithmétique
`shr*` i, d	$d \leftarrow d \gg i$	i immédiat ; décalage logique
`neg*` d	$d \leftarrow -d$	
`add*` s, d	$d \leftarrow d + s$	
`sub*` s, d	$d \leftarrow d - s$	
`inc*` d	$d \leftarrow d + 1$	
`dec*` d	$d \leftarrow d - 1$	
`imul*` d, s	$d \leftarrow (d \times s) \wedge 2^{k/2}$	d doit être un registre de taille k
`idivq` s	$(\%\texttt{rax}, \%\texttt{rdx}) \leftarrow (\%\texttt{rdx} :: \%\texttt{rax}) \div s$	division signée
`divq` s	$(\%\texttt{rax}, \%\texttt{rdx}) \leftarrow (\%\texttt{rdx} :: \%\texttt{rax}) \div s$	division non signée
`cqto`	$\%\texttt{rax} :: \%\texttt{rdx} \leftarrow SEXT_{128}(\%\texttt{rax})$	
`lea` s, d	$d \leftarrow \&s$	s doit être un opérande mémoire

- Les instructions suffixées * existent en variantes b/w/l/q, sauf `imulb`.
- Il y en a des centaines d'autres ! Mais ce petit jeu d'instruction suffit.

Drapeaux et instructions arithmétiques et logiques

- L'état du processeur contient quatre *drapeaux* booléens.

Drapeau	Propriété du résultat
ZF	$= 0$
CF	retenue sortante
SF	< 0
OF	dépassement de capacité

Drapeaux et instructions arithmétiques et logiques

- L'état du processeur contient quatre *drapeaux* booléens.

Drapeau	Propriété du résultat
ZF	$= 0$
CF	retenue sortante
SF	< 0
OF	dépassement de capacité

- Les instructions arithmétiques et logiques positionnent les drapeaux.

Drapeaux et instructions arithmétiques et logiques

- L'état du processeur contient quatre *drapeaux* booléens.

Drapeau	Propriété du résultat
ZF	$= 0$
CF	retenue sortante
SF	< 0
OF	dépassement de capacité

- Les instructions arithmétiques et logiques positionnent les drapeaux.
- Utiliser `cmp` et `test` positionne les drapeaux sans modifier de registre.

Instruction	Sémantique	Remarques
`cmp*` s_2, s_1	$\_ \leftarrow s_1 - s_2$	
`test*` s_2, s_1	$\_ \leftarrow s_1 \wedge s_2$	bit à bit

Drapeaux et instructions arithmétiques et logiques

- L'état du processeur contient quatre *drapeaux* booléens.

Drapeau	Propriété du résultat
ZF	$= 0$
CF	retenue sortante
SF	< 0
OF	dépassement de capacité

- Les instructions arithmétiques et logiques positionnent les drapeaux.
- Utiliser `cmp` et `test` positionne les drapeaux sans modifier de registre.

Instruction	Sémantique	Remarques
`cmp*` s_2, s_1	$\_ \leftarrow s_1 - s_2$	
`test*` s_2, s_1	$\_ \leftarrow s_1 \wedge s_2$	bit à bit

- Les drapeaux influencent la valeur des *codes de condition*.

Sauts et codes de condition

- `jmp label` : continue l'exécution à `label`.
- `jmp *%reg` : continue l'exécution à l'adresse contenue dans `%reg`.
- `j`cc `label` : continue l'exécution à `label` si cc.

Sauts et codes de condition

- `jmp label` : continue l'exécution à `label`.
- `jmp *%reg` : continue l'exécution à l'adresse contenue dans `%reg`.
- `j`cc `label` : continue l'exécution à `label` si cc.

Code de condition cc	Test réalisé	Sémantique après cmp s_2, s_1
e	ZF	$s_1 = s_2$
ne	$\neg ZF$	$s_1 \neq s_2$
l	$SF \oplus OF$	$s_1 < s_2$
le	$(SF \oplus OF) \vee ZF$	$s_1 \leq s_2$
g	$\neg(SF \oplus OF) \wedge \neg ZF$	$s_1 > s_2$
ge	$\neg(SF \oplus OF)$	$s_1 \geq s_2$

- Toutes les comparaisons arithmétiques ci-dessus sont signées.
- Voir Tolmach (2012) pour les cas non signés (codes a, ae, b, be).

Plan

1 Introduction

2 Les bases du jeu d'instructions

3 Convention d'appel et gestion de la pile

4 Conclusion

La pile

- Une zone de mémoire gérée suivant la discipline *last in, first out*.
- La dernière adresse de pile utilisée est dans `%rsp` (*stack pointer*).
- Elle croît vers le bas, donc on décrémente `%rsp` pour allouer.
- Quelques instructions permettent de la modifier facilement.

La pile

- Une zone de mémoire gérée suivant la discipline *last in, first out*.
- La dernière adresse de pile utilisée est dans $\%\texttt{rsp}$ (*stack pointer*).
- Elle croît vers le bas, donc on décrémente $\%\texttt{rsp}$ pour allouer.
- Quelques instructions permettent de la modifier facilement.

Instruction	Sémantique
push* *s*	$\%\texttt{rsp} \leftarrow \%\texttt{rsp} - 2^k; Mem[\%\texttt{rsp}] \leftarrow s$
pop* *d*	$d \leftarrow Mem[\%\texttt{rsp}]; \%\texttt{rsp} \leftarrow \%\texttt{rsp} + 2^k$
call *s*	$\%\texttt{rsp} \leftarrow \%\texttt{rsp} - 8; Mem[\%\texttt{rsp}] \leftarrow \%\texttt{rip}; \%\texttt{rip} \leftarrow s$
ret	$\%\texttt{rip} \leftarrow Mem[\%\texttt{rsp}]; \%\texttt{rsp} \leftarrow \%\texttt{rsp} + 8$

(Ci-dessus, k désigne la taille du suffixe, par exemple $k = 3$ pour pushq.)

Respect des bonnes mœurs lors des appels fonctions

- Le duo `call`/`ret` permet d'appeler/revenir d'un appelant.
- Reste à gérer : passage arguments et résultat ; registres préservés.

Respect des bonnes mœurs lors des appels fonctions

- Le duo `call`/`ret` permet d'appeler/revenir d'un appelant.
- Reste à gérer : passage arguments et résultat ; registres préservés.

La convention d'appel POSIX System-V AMD64, pour nous

- Les six premiers arguments et le résultat sont passés par les registres.
  - Arguments : `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` ; résultat : `%rax`.
  - Les éventuels arguments restants sont empilés, du dernier au septième.
- Seuls `%rbx`, `%rbp`, `%r12`–`%r15` doivent être préservés par les appels.
  - Une garantie pour les appelants, une contrainte pour les appelés.
- `%rsp` doit être aligné sur 16 octets à l'entrée d'une fonction.
  - Autrement dit, `%rsp` $+ 8$ doit être aligné sur 16 octets à chaque `call`.
  - Si vous violez cette règle, tout fonctionne… jusqu'à `call printf`.

Respect des bonnes mœurs lors des appels fonctions

- Le duo `call`/`ret` permet d'appeler/revenir d'un appelant.
- Reste à gérer : passage arguments et résultat ; registres préservés.

La convention d'appel POSIX System-V AMD64, pour nous

- Les six premiers arguments et le résultat sont passés par les registres.
  - Arguments : `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` ; résultat : `%rax`.
  - Les éventuels arguments restants sont empilés, du dernier au septième.
- Seuls `%rbx`, `%rbp`, `%r12`–`%r15` doivent être préservés par les appels.
  - Une garantie pour les appelants, une contrainte pour les appelés.
- `%rsp` doit être aligné sur 16 octets à l'entrée d'une fonction.
  - Autrement dit, `%rsp` $+ 8$ doit être aligné sur 16 octets à chaque `call`.
  - Si vous violez cette règle, tout fonctionne… jusqu'à `call printf`.

Tout cela ne nous dit pas comment gérer les *variables locales*…

Le cadre de pile et les variables locales

Chaque appel de fonction va avoir son *cadre de pile*, portion privée.

- Alloué par le *prologue* de la fonction, au début de l'appel.
- Désalloué par l'*épilogue* de la fonction, juste avant le retour.
- Invariant : %rbp contient l'adresse du cadre de la fonction courante.

Le cadre de pile et les variables locales

Chaque appel de fonction va avoir son *cadre de pile*, portion privée.

- Alloué par le *prologue* de la fonction, au début de l'appel.
- Désalloué par l'*épilogue* de la fonction, juste avant le retour.
- Invariant : %rbp contient l'adresse du cadre de la fonction courante.

```
# Prologue générique
pushq %rbp
movq %rsp, %rbp
subq $FRAME_SIZE, %rsp
```

Le cadre de pile et les variables locales

Chaque appel de fonction va avoir son *cadre de pile*, portion privée.

- Alloué par le *prologue* de la fonction, au début de l'appel.
- Désalloué par l'*épilogue* de la fonction, juste avant le retour.
- Invariant : %rbp contient l'adresse du cadre de la fonction courante.

```
# Prologue générique
pushq %rbp
movq %rsp, %rbp
subq $FRAME_SIZE, %rsp
```

```
# Épilogue générique
addq $FRAME_SIZE, %rsp
popq %rbp
ret
```

Le cadre de pile et les variables locales

Chaque appel de fonction va avoir son *cadre de pile*, portion privée.

- Alloué par le *prologue* de la fonction, au début de l'appel.
- Désalloué par l'*épilogue* de la fonction, juste avant le retour.
- Invariant : %rbp contient l'adresse du cadre de la fonction courante.

```
# Prologue générique
pushq %rbp
movq %rsp, %rbp
subq $FRAME_SIZE, %rsp
```

```
# Épilogue générique
addq $FRAME_SIZE, %rsp
popq %rbp
ret
```

L'idée clef

- On choisit, pour chaque variable locale x, un emplacement $slot(x)$ dans le cadre de pile de la fonction.
- Comme %rbp est fixe durant l'exécution de la fonction, on peut toujours faire référence à x via l'opérande mémoire $-slot(x)(\%\text{rbp})$.

Une vue schématique de la pile


```
f() { ... g(..., arg7, arg8); }
g(...) { int x, y, z; ... }
```

Un exemple réaliste

```c
#include <stdint.h>
int64_t twice(int64_t num) { int64_t x = num; return x + x; }
```

Un exemple réaliste

```c
#include <stdint.h>
int64_t twice(int64_t num) { int64_t x = num; return x + x; }
```

```asm
# Produit par gcc -O0 -fno-omit-frame-pointer
twice:  pushq   %rbp            # }
        movq    %rsp, %rbp      # } PROLOGUE
        movq    %rdi, -8(%rbp)  # stack(x) <- %rdi
        movq    -8(%rbp), %rax  # %rax <- stack(x)
        addq    %rax, %rax      # %rax <- %rax + %rax
        popq    %rbp            # } ÉPILOGUE
        ret                     # }
```

Un exemple réaliste

```c
#include <stdint.h>
int64_t twice(int64_t num) { int64_t x = num; return x + x; }
```

```asm
# Produit par gcc -O0 -fno-omit-frame-pointer
twice:  pushq   %rbp                # }
        movq    %rsp, %rbp          # } PROLOGUE
        movq    %rdi, -8(%rbp)      # stack(x) <- %rdi
        movq    -8(%rbp), %rax      # %rax <- stack(x)
        addq    %rax, %rax          # %rax <- %rax + %rax
        popq    %rbp                # } ÉPILOGUE
        ret                         # }

# Produit par gcc -O1 -fno-omit-frame-pointer
twice:  leaq (%rdi, %rdi), %rax
        ret
```

Plan

1 Introduction

2 Les bases du jeu d'instructions

3 Convention d'appel et gestion de la pile

4 Conclusion

Conclusion

Les découvertes de cette séance

- Le langage machine x86-64.
  - Baroque mais pas *si* complexe pour un compilateur simple.
- Les bases de la programmation système des PC sous Linux.

Conclusion

Les découvertes de cette séance

- Le langage machine x86-64.
  - Baroque mais pas *si* complexe pour un compilateur simple.
- Les bases de la programmation système des PC sous Linux.

Ces informations sont essentielles pour écrire un compilateur :

- *correct* et capable d'interopérer avec le système (comme flap) ;
- *optimisant*, comme GCC ; notamment, pour l'*allocation de registre*.

En route pour le dernier jalon : Retrolix vers x86-64 !

Bibliographie


Intel Corporation (2023). *x86-64 Programming Manual*. URL : https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.


Tolmach, Andrew (2012). *Notes on x86-64 programming*. URL : http://web.cecs.pdx.edu/~apt/cs491/x86-64.pdf.