Génération du pré-code machine

Alexis Nasr
Carlos Ramisch
Manon Scholivet
Franck Dary

Compilation – L3 Informatique
Département Informatique et Interactions
Aix Marseille Université

Génération de pré-code

- Il ne s'agit pas encore tout à fait du code final.
- On considère que l'on dispose d'un nombre illimité de registres : `r1`, `r2`, ...
- L'affectation aux registres du processeur `eax`, `ebx`, `ecx` et `edx` sera réalisée à la prochaine étape : l'allocation de registres.
- Tout le reste est identique au code final.

Code trois adresses

```
1  main fbegin
2
3
4        t0 = 3 + 10
5
6        write t0
7
8        fend
```

Pré assembleur

```
1  main : push ebp
2         mov  ebp, esp
3         sub  esp, 0
4         mov  r0,  3
5         add  r0,  10
6         mov  eax, r0
7         call iprintLF
8         add  esp, 0
9         pop  ebp
10        ret
```

Idée générale

- Parcourir la séquence des instructions du code trois adresses
- Chaque instruction trois adresses devient une suite d'instructions assembleur x86
- Les opérandes du code trois adresses deviennent des registres, positions mémoire ou constantes

Plan

- Opérandes
  - temporaires
  - constantes
  - variables
  - étiquettes
- Opérations artithmétiques
- Sauts
- Affectation
- Appel de fonction
  - Paramètres
  - Appel
  - Entrée dans une fonction
  - Valeur de retour
  - Sortie d'une fonction
- Entrées / Sorties

Opérandes

- Les opérandes des instructions du code trois adresses sont :
  1. des variables (pointeur vers la table des symboles)
  2. des temporaires
  3. des étiquettes
  4. des fonctions (pointeur vers la table des symboles)
  5. des constantes
- Les opérandes des instructions x86 sont :
  1. des registres
  2. des constantes
  3. des adresses mémoire spécifiées par des étiquettes
  4. des adresses mémoire spécifiées par des calculs

Conversions

- variable $\rightarrow$ adresse, étiquette
- temporaires $\rightarrow$ registre
- étiquette $\rightarrow$ étiquette
- fonctions $\rightarrow$ étiquette
- constante $\rightarrow$ constante

Variables

- En code trois adresses, on ne distingue pas les variables globales, des variables locales, des paramètres de fonctions.
- En assembleur, chaque type de variable est stocké à des endroits spécifiques de la mémoire.
- Variables globales :
  - Dans la région `.data`.
  - Chaque variable est associée à une étiquette.
- Variables locales et paramètres :
  - Dans la pile
  - Pas d'étiquettes
  - On y accède par un calcul à partir de la base de la trame de pile (`ebp`)
  - Le calcul dépend du format de la trame de pile
  - Exemples :
    - `glob` $\rightarrow$ `[glob]`
    - `local_i` $\rightarrow$ `[ebp - 4 * i]`
    - `arg_i` $\rightarrow$ `[ebp + 8 + 4 * nb_args - 4 * i]`

Temporaires

- Les temporaires du code trois adresses sont des registres du pré-assembleur
- `t1` $\rightarrow$ `r1`
- On peut garder les mêmes numéros, ça aide pour débugger.
- Attention aux collisions pour les registres créés lors de la création du pré-assembleur.

Étiquettes

- Noms symboliques associés à des adresses
- Trois types :
  - Les fonctions (adresse de la première instruction de la fonction)
  - Les étiquettes automatiques (existantes dans le code trois adresses)
  - Les variables globales

Opérations arithmétiques : additions, soustractions et multiplications

Code trois adresses

```
1  t0 = 3 + 10
```

Pré assembleur

```
1  mov r0, 3
2  add r0, 10
```

- L'instruction :

  `add destination source`

- Effectue :

  `destination = destination + source`

- Avant de réaliser l'addition, on copie la première opérande dans la destination.

Opérations arithmétiques : divisions

Code trois adresses

```
1  t0 = 10 / t1
2
3
4  t0 = 10 / 2
```

Pré assembleur

```
1  mov  eax, 10
2  idiv t1
3
4  mov  eax, 10
5  mov  ebx, 2
6  idiv ebx
```

- L'instruction :

  `imul source`

- Effectue : `eax = eax / source`.
- La première opérande **doit** être mise dans `eax`.
- `source` ne peut pas être une constante.

Les sauts I

Code trois adresses

```
1  t1 == 0 goto l2
2
```

Pré assembleur

```
1  cmp r1, 0
2  je l2
```

- L'instruction :

  `cmp destination, source`

- Effectue l'opération `destination` - `source`
- si `destination = source`, `ZF` (Zero Flag) vaut `VRAI`
- L'instruction :

  `je adr`

- Va à l'adresse `adr` si `ZF` vaut `VRAI`

Les sauts II

Code trois adresses

```
1 1 == 2 goto l2
2
```

Pré assembleur

```
1 mov r4, 1
2 cmp r4, 2
3 je  l2
```

- L'instruction : `cmp destination, source`
- Ne peut avoir une constante pour valeur de `destination`, il faut passer par un registre

Affectations

Code trois adresses

```
1  glob = 123
2  t0   = 123
3  loc  = 123
4  glob = loc
```

Pré assembleur

```
1  mov dword [glob], 123
2  mov r0, 123
3  mov dword [ebp -4*1], 123
4  mov r1, dword [ebp -4*1]
5  mov glob, r1
```

- La traduction est directe
- Dans l'exemple, la variable locale `loc` a pour adresse 0
- L'instruction `mov` ne peut avoir pour opérandes deux adresses, il faut passer par un registre.

Paramètre

Code trois adresses

```
1  param 1
2  param t1
3  param glob
```

- Taduction directe

Pré assembleur

```
1  push 1
2  push r1
3  push [glob]
```

Appel de fonction

Code trois adresses

```
1  t1 = call fonc
```

Pré assembleur

```
1  sub  esp, 4
2  call fonc
3  pop  r1
4  add  esp, 8
```

- Allocation de quatre octets dans la pile pour stocker la valeur de retour
- Appel à la fonction
- Récupération de la valeur de retour
- Désallocation de l'espace occupé dans la pile par les paramètres (ici `fonc` a deux paramètres)

Entrée dans une fonction

Code trois adresses

```
1  fonc fbegin
```

Pré assembleur

```
1  fonc : push ebp
2         mov  ebp, esp
3         sub  esp, 4
```

- Sauvegarde dans la pile de l'ancienne valeur de `ebp`
- Nouvelle valeur de `ebp`
- Allocation de mémoire dans la pile pour les variables locales (ici une)

Valeur de retour

Code trois adresses

```
1  ret t0
```

Pré assembleur

```
1  mov dword [ebp+4*2], r0
```

Sortie de fonction

Code trois adresses

```
1 fend
```

Pré assembleur

```
1 add esp, 4
2 pop ebp
3 ret
```

- Désallocation de la mémoire occupée par les variables locales (ici une)
- Restauration de l'ancienne valeur de `ebp` (d'avant l'appel)
- Sortie de la fonction (récupération de la valeur de `eip` empilée par l'instruction `call`)

Un exemple complet d'appel de fonction

```
1  fonc(entier a)
2  entier b;
3  {
4  b = a;
5  retour(b);
6  }
7
8  main()
9  {
10 fonc(1);
11 }
12
```

```
1  fonc fbegin
2
3
4        b = a
5        ret b
6        fend
7
8  main fbegin
9        param 1
10       t0 = call fonc
11       fend
```

Un exemple complet d'appel de fonction

```
1  fonc fbegin
2
3
4        b = a
5
6
7        ret b
8        fend
9
10
11 main fbegin
12
13
14       param 1
15
16       t0 = call fonc
17
18       fend
```

```
1  fonc : push ebp
2         mov ebp, esp
3         sub esp, 4
4         mov r3, dword [ebp+4*3]
5         mov dword [ebp-4*1], r3
6         mov r4, dword [ebp-4*1]
7         mov dword [ebp+4*2], r4
8         add esp, 4
9         pop ebp
10        ret
11 main : push ebp
12        mov ebp, esp
13        sub esp, 0
14        push 1
15        sub esp, 4
16        call fonc
17        pop r0
18        add esp, 4
19        add esp, 0
20        pop ebp
21        ret
```

Entrées Sorties

- On delègue la gestion des E/S au système d'exploitation
- **Rappel :** les appels au système sont des interruptions `int 0x80`, avec le code de l'opération à effectuer dans `eax`

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

- La bibliothèque assembleur `io.asm` encapsule ces appels.
- Elle définit trois fonctions :
  - `iprintLF` : affiche l'entier contenu dans `eax`
  - `readline` : lit et stocke une ligne à l'adresse pointée par `eax`
  - `atoi` : met dans `eax` l'entier obtenu de la chaîne pointée par `eax`
- Les opérations `read` et `write` sont implémentées de cette façon

Lecture / Ecriture

Code trois adresses

```
1  write 1
2
3
4  t1 = read
```

Pré assembleur

```
1  mov eax, 1
2  call iprintLF
3
4  mov  eax, sinput
5  call readline
6  call atoi
7  mov  r1, eax
```