

# Chapitre 7

## Production de code assembleur

### Sommaire

|            |  |          |
|------------|--|----------|
| <b>7.1</b> | <b>Du code intermédiaire à l'assembleur abstrait : la sélection d'instructions . . . . .</b> | <b>1</b> |
| 7.1.1      | L'assembleur abstrait . . . . .  | 1        |
| 7.1.2      | Sélection des instructions . . . . .   | 3        |
| <b>7.2</b> | <b>Filtrage de l'arbre de code intermédiaire . . . . .</b>                                   | <b>4</b> |
| 7.2.1      | Algorithme simple . . . . .  | 5        |
| 7.2.2      | Algorithme de programmation dynamique . . . . .  | 5        |
| 7.2.3      | Schéma d'implantation de MaximalMunch . . . . .  | 6        |
| 7.2.4      | Place de la sélection d'instructions dans le compilateur . . . . .                           | 6        |
| <b>7.3</b> | <b>Allocation des temporaires . . . . .</b>  | <b>7</b> |
| 7.3.1      | Stratégie naïve . . . . .  | 7        |
| 7.3.2      | Stratégie naïve pour les processeurs RISC . . . . .  | 8        |
| 7.3.3      | Stratégie naïve pour les processeurs CISC . . . . .  | 9        |

## 7.1 Du code intermédiaire à l'assembleur abstrait : la sélection d'instructions

### 7.1.1 L'assembleur abstrait

#### Assembleur abstrait

Nous introduisons une version abstraite (une variante du code à trois adresses que l'on trouve en littérature) de langage assembleur. Il ne s'agit plus d'une représentation arborescente : le programme est bel et bien devenu une liste d'instructions machine, définies comme suit :

```
type 'reg instr =
| OP of binop * 'reg * 'reg * 'reg (** Arithmetical operation on registers *)
| OPI of binop * 'reg * 'reg * int (** Same, with immediate 2nd operand *)
| BR of relop * 'reg * 'reg * lbl (** Conditional branching *)
| BRI of Ir.relop * 'reg * int * lbl (** Same, with immediate 2nd operand *)
| LW of 'reg * int * 'reg (** lw reg1, int(reg2) *)
| SW of 'reg * int * 'reg (** sw reg1, int(reg2) *)
```

```

| LA of 'reg * lbl
| LI of 'reg * int
| J of lbl
| JAL of lbl
| JR of 'reg
| MOV of 'reg * 'reg
| LBL of lbl
| COMM of string

```

```

type 'reg instrs = 'reg instr list

```

Notez que cette partie dépend du processeur cible.

### Le fichier mipsAsmPrint.ml

```

open Printf
open Temp
open Ir

```

```

type instr = ... (* cf ci-dessus *)

```

```

type instrs = instr list

```

```

(** [instr_map] : Given a modification function for sources and

```

```

let asm_of_relop = function
| EQ -> "beq"
| NE -> "bne"
| LT -> "blt"
| GT -> "bgt"
| LE -> "ble"
| GE -> "bge"
| ULT -> "bltu"
| ULE -> "bleu"
| UGT -> "bgtu"
| UGE -> "bgeu"

(** [format] : display of an assembly instruction, given a printing
    function for temps. *)

let format saytemp = function
| OP(o,d,s1,s2) -> sprintf "%s %s, %s, %s"
    (asm_of_binop o) (saytemp d) (saytemp s1) (saytemp s2)
| OPI(o,d,s,i) -> sprintf "%s %s, %s, %d"
    (asm_of_binop o) (saytemp d) (saytemp s) i
| BR(r,s1,s2,l) -> sprintf "%s %s, %s, %s"
    (asm_of_relop r) (saytemp s1) (saytemp s2) l
| BRI(r,s,i,l) -> sprintf "%s %s, %d, %s" (asm_of_relop r) (saytemp s) i l
| LW(d,i,s) -> sprintf "lw %s, %d(%s)" (saytemp d) i (saytemp s)
| SW(s1,i,s2) -> sprintf "sw %s, %d(%s)" (saytemp s1) i (saytemp s2)
| LA (d,l) -> sprintf "la %s, %s" (saytemp d) l
| LI (d,i) -> sprintf "li %s, %d" (saytemp d) i
| J l -> "j ^l"
| JAL l -> "jal ^l"
| JR s -> "jr ^(saytemp s)"
| MOV (d,s) -> sprintf "move %s, %s" (saytemp d) (saytemp s)
| LBL l -> l^":\n"
| COMM s -> s

```

## 7.1.2 Sélection des instructions

### Du code intermédiaire à l'assembleur

Une instruction assembleur peut recouvrir plusieurs noeuds d'un arbydby cuds intermé(arb.-735(191 Tf 16.93

```
LMOVE(LTEMP 5,
      LMEM(LBINOP(PLUS,
                  LTEMP fp,
                  LCONST 10)))
```

peut être recouvert aussi par la séquence d'instructions assembleur

```
ddi $15,$fp,10
lw $5, ($15)
```

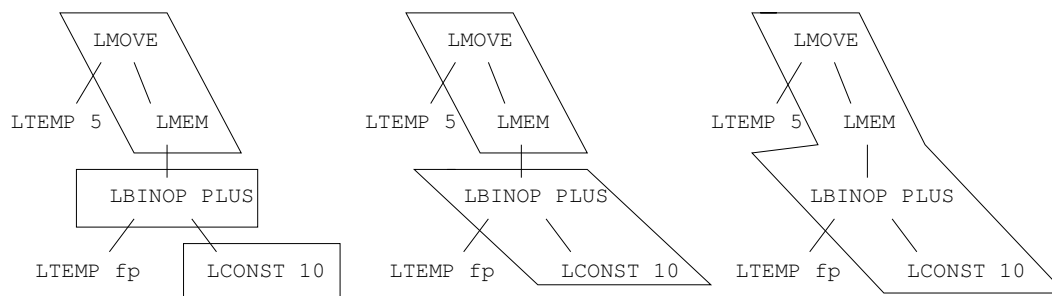
ou même par

```
ddi $18,$0,10 # $0 contient toujours zero sur MIPS
dd $15,$fp,$18
lw $5, ($15)
```

## 7.2 Filtrage de l'arbre de code intermédiaire

### Tuiles

On appellera *tuile* (*tile* en anglais) un fragment d'arbre de code intermédiaire correspondant à une instruction du processeur. C'est un arbre, avec une racine et des feuilles.



Une instruction assembleur correspond (en général) à une tuile, et un arbre de code intermédiaire doit pouvoir être recouvert entièrement par des tuiles *disjointes*, qui correspondent à des instructions assembleur.

### Tuiles et temporaires

Il n'est pas nécessaire de prévoir des temporaires pour les noeuds *internes* d'une tuile (dans le cas de `lw $5, 10($fp)`, il ne nous intéresse pas de savoir par quel moyen la machine assembleur fait la somme entre `FP` et `10` et lit la mémoire). Par contre, on doit allouer des temporaires pour les noeuds racine (le résultat de l'instruction) et les feuilles (les paramètres de l'instruction). Voir exemple fait au tableau pour `x := [10]`.

## Algorithmes

Quelle séquence d'instruction assembleur doit-on utiliser pour recouvrir un arbre de code intermédiaire ? Un critère raisonnable est la minimisation du *coût* de la séquence d'instructions assembleur produite (où le coût est souvent le temps d'exécution).

Vis à vis d'une notion de coût fixée, on peut obtenir une séquence assembleur :

**optimale** si on ne peut jamais remplacer deux tuiles adjacentes par une tuile (ayant le même effet) de coût inférieur

**optimum** s'il n'existe pas d'autre séquence d'instruction assembleur (ayant le même effet) de coût inférieur

### 7.2.1 Algorithme simple

#### Algorithme : MaximalMunch

Il est facile d'obtenir une séquence optimale :

```
Algorithme MaximalMunch( arbre )
```

```
  Prendre toutes les tuiles qui peuvent recouvrir une partie de l'arbre
  en partant de la racine, choisir la plus grosse.
```

```
  Émettre l'instruction correspondante
```

```
  Se répéter récursivement sur les sous-arbres
  correspondants aux feuilles de la tuile choisie
```

Attention : cet algorithme émet les instructions dans l'ordre inverse !

#### Un mot sur le filtrage de tuiles

Le filtrage de motifs (pattern matching) de Ocaml est très bien adapté à la sélection de tuiles opérée par MaximalMunch.

Il faut mettre les tuiles les plus grosses *d'abord*, et le compilateur Ocaml produira un arbre de décision qui permet d'opérer le filtrage en temps linéaire (on ne reviendra pas sur une comparaison déjà effectuée).

### 7.2.2 Algorithme de programmation dynamique

#### Optimum par programmation dynamique

Il est plus complexe d'obtenir une séquence optimum :

```
Algorithme MoindreCoût(arbre) // On suppose connus les coûts
                                // des tous les sous-arbres
  Parmi toutes les tuiles qui peuvent recouvrir l'arbre
  en partant de la racine, choisir celle telle que la
  somme de son coût avec les coûts des sous-arbres correspondant
  aux feuilles est minimale.
```

Émettre le code correspondant à la tuile, puis émettre le code correspondant aux sous-arbres en position de feuille pour la tuile choisie.

Dans la pratique, MaximalMunch se conduit plutôt bien.

### 7.2.3 Schéma d'implantation de MaximalMunch

```
open Ast
open Assem
open Ir

let emit, get_emitted_instrs = Util.mkfifo ()

(* NB: we can skip some special cases (for example BINOP(PLUS, CONST k, e))
   thanks to some ad-hoc transformations in Canon.do_exp *)
let rec munchExp = function
| LTEMP t -> t
| a ->
    let res = Temp.new_temp () in
    emit (munchExp_result res a); res

and munchExp_result res = function
| LMEM(LBINOP(PLUS, e, LCONST k)) -> LW(res, k, munchExp e)
| ...

and munchStm = function
| LMOVE(LMEM(LBINOP(PLUS, e1, LCONST k)), e2) ->
    emit (SW(munchExp e2, k, munchExp e1))
| ...
```

#### Mise en garde

Le coût d'un programme (en temps) n'est pas toujours exprimable facilement en terme de coût des instructions isolées, surtout dans le cadre des architectures récentes, superscalaires et « pipelined ». Les algorithmes que l'on vient de voir fonctionnent dans le cas simpliste d'une architecture séquentielle traditionnelle.

### 7.2.4 Place de la sélection d'instructions dans le compilateur

#### Les étapes de la transformation

```

Ast.r wexp = (Symbol.symbol, Symbol.symbol, Symbol.symbol) Ast.exp
                ↓calcul de level-offest
Ast. ttexp = (Ast. ttrv r, Ast. ttrfunc ll, Ast. ttrfundef) Ast.exp
                ↓Génération de code intermédiaire
                Ir.ir = Ir.exp Ir.prog
                ↓Canonisation
                Lin.lin = Lin.funbody Ir.prog
                ↓Sélection d'instructions
                MipsAsm. bstr_instrs Ir.prog
                ↓Allocation des registres
                MipsAsm.simpl_instrs Ir.prog

```

L'émission des prologues et épilogues se fait en dernier, lorsque l'on connaît le nombre de temporaires sur la pile.

## 7.3 Allocation des temporaires

### 7.3.1 Stratégie naïve

#### Traitement des temporaires

Maintenant, il nous faut nous tourner vers le seul objet que l'on a oublié jusqu'ici : les temporaires.

Pour plein d'opérations, et maintenant pour chaque tuile, il nous faut des temporaires pour contenir les résultats intermédiaires ou les paramètres de chaque instruction.

#### Exemple

La traduction de l'expression  $1 + 2 + 3 + (4 + 5) + (6 + 7)$  ressemble à

L1:

```

ddi t102 $0 6
ddi t101 t102 7
ddi t105 $0 4
ddi t104 t105 5
ddi t108 $0 1
ddi t107 t108 2
ddi t106 t107 3
dd t103 t106 t104
dd t100 t103 t101

```

#### Traitement simple des temporaires

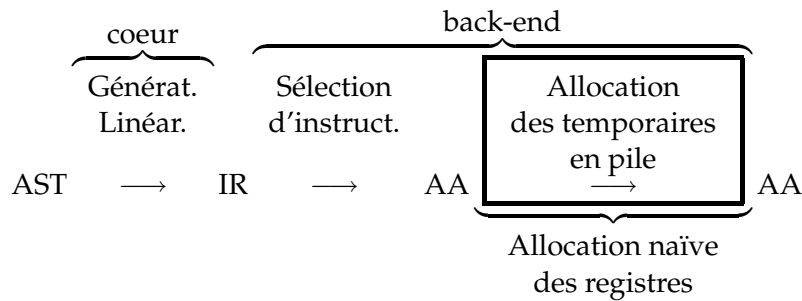
Qu'est-ce qu'un temporaire ? Nous proposons dans ce chapitre un traitement simple, mais inefficace, qui se résume ainsi :

Un temporaire est traité exactement comme une variable locale d'une fonction, donc il est alloué en mémoire, sur la pile.

Donc, si dans la compilation du corps de la fonction  $f$  nous utilisons les temporaires entre  $t100$  et  $t114$ , il y aura 15 emplacements supplémentaires alloués sur la pile, pour le contenir, et chaque référence à un de ces temporaires se traduira par un accès en mémoire à la case correspondante.

Nous verrons dans le chapitre suivant une meilleure technique d'allocation des temporaires.

### Où en est-on dans la chaîne de compilation ?



L'assembleur abstrait modifié et réécrit en fin de chaîne peut être enfin converti en fichier assembleur pour SPIM.

### Stratégie naïve pour RISC et CISC

La stratégie que l'on vient de voir génère beaucoup de trafic mémoire et n'exploite pas les registres machines disponibles sur des processeurs RISC.

Sur des machines RISC et CISC, sa réalisation demande des approches légèrement différentes.

### 7.3.2 Stratégie naïve pour les processeurs RISC

Les processeurs RISC ont beaucoup de registres, sur lesquels on peut effectuer normalement toutes les opérations. Dans la plupart des cas, comme celui du MIPS émulé par SPIM, l'accès à la mémoire est disponible exclusivement à travers des instructions de chargement et mémorisation explicite (architecture LOAD/STORE). Cela signifie que si l'on souhaite réaliser une addition entre deux temporaires, disons  $t110$  et  $t120$  qui se trouvent sur la pile, pour mettre le résultat dans  $t115$ , on ne peut pas écrire

```
dd M[t115] M[t110] M[t120]
```

Il faudra séparer la séquence en une suite « chargement, opération, mémorisation », comme dans les pseudo-instructions

```
lw $3 M[t110]
# ch rge d ns le registre 3
# l c se mémoire pour t110
lw $4 M[t120]
# ch rge d ns le registre 4
# l c se mémoire pour t120
```



```

dd $5 $3 $4
sw $5 M[t115]
# mémorise le registre 5
# dans la case mémoire pour t110

```

Bien entendu, pour obtenir de l'assembleur correct, il faudra remplacer les `M[t115]` etc. par l'adresse de la case mémoire du temporaire correspondant. Typiquement, cela deviendra du `k($fp)` où *k* est l'offset par rapport à `$fp` de la case mémoire sur la pile contenant `t115`.

Voilà donc notre approche simpliste pour un processeur RISC :

- on alloue tous les temporaires en pile
- on réserve 2 registres pour les opérations, par exemple :

**\$10** opérande 1

**\$11** opérande 2

**\$10** résultat

- avant de chaque opération, on charge le contenu des temporaires opérandes dans les 2 registres opérande
- on effectue l'opération
- on sauvegarde le registre résultat à l'adresse sur la pile contenant la case du temporaire destination

### 7.3.3 Stratégie naïve pour les processeurs CISC

#### Stratégie simpliste pour CISC

Les processeurs CISC proposent (à différence des RISC), peu de registres (6 sur le Pentium), mais sur lesquels on peut faire plein de choses (par exemple, les utiliser pour indexer la mémoire lors d'opérations arithmétiques).

Dans le cas du Pentium, on a aussi une autre restriction : les opérations sont toujours à 2 adresses et pas 3.

Pour ces processeurs, l'approche simpliste est même plus facile à mettre en place, parce que les `load` et `store` peuvent devenir implicites grâce aux modes d'adressage, et on peut se réduire à :

- on alloue tous les temporaires en pile
- on effectue une partie des opérations directement en mémoire en exploitant les adressages plus complexes.

Par exemple, une pseudo-instruction à 2 opérandes du style

```
dd M[t110] M[t110] M[t120]
```

pourrait devenir

```

mov edx, [ebp-12]< 11>
dd [ebp-4], edx

```

plutôt que la suite « à la RISC »

```
mov edx, [ebp-12]< 11>
mov e x, [ebp-4]< 11>
  dd e x, edx< 11>
mov [ebp-4], e x
```

### Défauts de l'approche simpliste

Avec cette approche simpliste, et sans optimisations successives, il est fort probable que l'on retrouve des séquences de LOAD et STORE complètement inutiles, parce que la valeur chargée ou mémorisée dans ou depuis un registre est la même que la précédente.

Pour obtenir du code beaucoup plus efficace, il est nécessaire de procéder à une analyse plus fine du programme source, et à une allocation des registres machines qui ne gaspille pas des ressources.

Ceci est le but de la « liveness analysis » et de l'allocation de registres par coloriage de graphes utilisés dans les compilateurs modernes (chapitre suivant).