



Interprétation et compilation

Chapitre 3 Production d'assembleur MIPS depuis OCaml



Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/ic

Production d'assembleur MIPS depuis OCaml

- ▶ Un programme assembleur est une suite d'instructions.
- ▶ Il n'y a pas de structure de contrôle, ni d'expression.
- ▶ Quelle structure de données permet de représenter cela ?


- ▶ Un programme assembleur est une suite d'instructions.
- ▶ Il n'y a pas de structure de contrôle, ni d'expression.
- ▶ Quelle structure de données permet de représenter cela ?
 - Une liste, tout simplement.

- ▶ En pratique on va avoir
 - une liste d'instructions dans le segment `.text`, et
 - une liste de déclarations dans le segment `.data`.

```
1 type asm = { text: instr list ; data: decl list }
```

- ▶ Il nous reste bien sûr à définir les types `instr` et `decl`.

Exemple

 Version assembleur :

```
1 .text
2 .globl main
3 # ...
4 .data
5 # ...
```

 Version OCaml :

```
1 let asm =
2   { text = [ (* ... *) ]
3     ; data = [ (* ... *) ]
4   }
```

Le segment de données

- ▶ Commençons par le segment de données.
- ▶ Dans ce segment on a des *déclarations* qui prennent la forme d'un *label* qui donne un nom à un espace mémoire réservé par une *directive*.
- ▶ Un label est une chaîne.
- ▶ Une directive est généralement un type de donnée qui prend en argument une valeur d'initialisation.

```
1 type label = string
2
3 type directive =
4   | Asciiiz of string
5
6 type decl = label * directive
```

Exemple

▶ Version assembleur :

```
1 .text
2 .globl main
3 # ...
4 .data
5 num: .asciiz "Please enter the value of n: "
6 sum: .asciiz "The sum of the numbers from 1 to n is: "
7 nl: .asciiz "\n"
```

▶ Version OCaml :

```
1 let asm =
2   { text = [ (* ... *) ]
3     ; data =
4       [ "num", Asciiiz "Please enter the value of n: "
5         ; "sum", Asciiiz "The sum of the numbers from 1 to n is: "
6         ; "nl", Asciiiz "\n"
7       ]
8   }
```


Le segment de code

- ▶ Dans le segment de code, on retrouve des déclarations de label et des *instructions* qui prennent en argument des *registres* et des *adresses*.
- ▶ Une adresse peut être un label, un registre, ou un emplacement mémoire.

```

1  type reg =
2  | Zero
3  | SP
4  | RA
5  | V0
6  | A0
7  | A1
8  | T0
9  | T1
10 | T2
11
12 type label = string
13
14 type loc =
15 | Lbl of label
16 | Reg of reg
17 | Mem of reg * int

```

```

18 type instr =
19 | Label of label
20 | Li of reg * int
21 | La of reg * loc
22 | Sw of reg * loc
23 | Lw of reg * loc
24 | Sb of reg * loc
25 | Lb of reg * loc
26 | Move of reg * reg
27 | Addi of reg * reg * int
28 | Add of reg * reg * reg
29 | Syscall
30 | B of label
31 | Beq of reg * reg * label
32 | Bne of reg * reg * label
33 | Jal of label
34 | Jr of reg

```

Exemple

▶ Version assembleur :

```

1 .text
2 .globl main
3 main:
4     li $v0, 4
5     la $a0, num
6     syscall
7     # ...
8 loop:
9     beq $t0, $0, end_loop
10    add $t1, $t1, $t0
11    addi $t0, $t0, -1
12    b loop
13 end_loop:
14    # ...
15    jr $ra
16 .data
17 num: .asciiz "Please enter the value of n: "
18 # ...

```

▶ Version OCaml :

```

1 let asm =
2   { text =
3     [ Label "main"
4       ; Li (V0, Syscall.print_str)
5       ; La (A0, Lbl "num")
6       ; Syscall
7       ; Label "loop"
8       ; Beq (T0, Zero, Lbl "end_loop")
9       ; Add (T1, T1, T0)
10      ; Addi (T0, T0, -1)
11      ; B "loop"
12      ; Label "end_loop"
13      ; Jr RA
14    ]
15   ; data =
16     [ "num", Asciiiz "Please enter the value of n: "
17     ]
18   }

```

Numéro des appels systèmes

- ▶ Dans l'extrait de code OCaml on a pu voir `Syscall.print_str`.
- ▶ C'est tout simplement une valeur qu'on a défini en OCaml parce que c'est plus lisible que 4 et que ça nous évite d'avoir à retenir par cœur les numéros des appels systèmes.

```
1 module Syscall = struct
2   let print_int = 1
3   let print_str = 4
4   let read_int = 5
5   let read_str = 8
6 end
```

- ▶ Il suffit d'écrire des fonctions capables de transformer en chaîne de caractères les valeurs de chaque type de données OCaml que l'on a défini.
- ▶ On peut ensuite les utiliser dans une fonction `print_asm` par exemple :

```
1 let print_asm oc asm =  
2   Printf.fprintf oc ".text\n.globl main\n" ;  
3   List.iter (fun i -> Printf.fprintf oc "%s\n" (fmt_instr i)) asm.text ;  
4   Printf.fprintf oc "\n.data\n" ;  
5   List.iter (fun (l, d) -> Printf.fprintf oc "%s: %s\n" l (fmt_dir d)) asm.data
```

Formatage des directives

 La fonction `fmt_dir` :

```
1 let ps = Printf.sprintf (* alias raccourci *)  
2  
3 let fmt_dir = function  
4   | Asciiiz (s) -> ps ".asciiiz \"%s\"" s
```

Formatage des registres et emplacements

▶ Les fonctions `fmt_reg` et `fmt_loc` :

```
1 let ps = Printf.sprintf (* alias raccourci *)
2
3 let fmt_reg = function
4   | Zero -> "$zero"
5   | SP  -> "$sp"
6   | RA  -> "$ra"
7   | V0  -> "$v0"
8   | A0  -> "$a0"
9   | A1  -> "$a1"
10  | T0  -> "$t0"
11  | T1  -> "$t1"
12  | T2  -> "$t2"
13
14 let fmt_loc = function
15   | Lbl (l)   -> l
16   | Reg (r)   -> fmt_reg r
17   | Mem (r, o) -> ps "%d(%s)" o (fmt_reg r)
```

Formatage des instructions

► La fonction `fmt_instr` :

```

1 let ps = Printf.sprintf (* alias raccourci *)
2
3 let fmt_instr = function
4 | Label (l)      -> ps "%s:" l
5 | Li (r, i)      -> ps " li %s, %d" (fmt_reg r) i
6 | La (r, a)      -> ps " la %s, %s" (fmt_reg r) (fmt_loc a)
7 | Sw (r, a)      -> ps " sw %s, %s" (fmt_reg r) (fmt_loc a)
8 | Lw (r, a)      -> ps " lw %s, %s" (fmt_reg r) (fmt_loc a)
9 | Sb (r, a)      -> ps " sb %s, %s" (fmt_reg r) (fmt_loc a)
10 | Lb (r, a)      -> ps " lb %s, %s" (fmt_reg r) (fmt_loc a)
11 | Move (rd, rs)  -> ps " move %s, %s" (fmt_reg rd) (fmt_reg rs)
12 | Addi (rd, rs, i) -> ps " addi %s, %s, %d" (fmt_reg rd) (fmt_reg rs) i
13 | Add (rd, rs, rt) -> ps " add %s, %s, %s" (fmt_reg rd) (fmt_reg rs) (fmt_reg rt)
14 | Syscall        -> ps " syscall"
15 | B (l)           -> ps " b %s" l
16 | Beq (rs, rt, l) -> ps " beq %s, %s, %s" (fmt_reg rs) (fmt_reg rt) l
17 | Bne (rs, rt, l) -> ps " bne %s, %s, %s" (fmt_reg rs) (fmt_reg rt) l
18 | Jal (l)         -> ps " jal %s" l
19 | Jr (r)          -> ps " jr %s" (fmt_reg r)

```

- ▶ Pourquoi ne pas manipuler directement des chaînes de caractères en OCaml ?

- ▶ Pourquoi ne pas manipuler directement des chaînes de caractères en OCaml ?
 - facilité de création par du code OCaml (utile dans nos futurs compilateurs),
 - souplesse de manipulation (optimisation),
 - bénéfice du typage !

En attendant la compilation

- ▶ Notre prochain objectif va être de représenter en OCaml des programmes de plus haut niveau, puis d'apprendre à les *compiler*, c'est à dire à transformer les structures de contrôle de haut niveau en instructions assembleurs.
- ▶ En attendant, on peut déjà utiliser OCaml pour écrire des fonctions d'aide qui produisent des instructions assembleurs.

Exemple

▶ Par exemple pour les appels systèmes :

```
1 let print_int r =  
2   [ Li (V0, Syscall.print_int)  
3     ; Move (A0, r)  
4     ; Syscall ]  
5  
6 let read_int r =  
7   [ Li (V0, Syscall.read_int)  
8     ; Syscall  
9     ; Move (r, V0) ]
```

▶ ou pour définir une fonction :

```
1 let push r =  
2   [ Addi (SP, SP, -4)  
3     ; Sw (r, Mem (SP, 0)) ]  
4  
5 let pop r =  
6   [ Lw (r, Mem (SP, 0))  
7     ; Addi (SP, SP, 4) ]  
8  
9 let def name body =  
10  [ Label name ]  
11  @ push RA  
12  @ body  
13  @ pop RA  
14  @ [ Jr RA ]
```

- Codons ensemble ce programme en assembleur directement, puis en passant par OCaml :

```
1 #include <stdio.h>
2
3 void countdown (int n)
4 {
5     if (n != 0) {
6         printf("%d\n", n);
7         countdown(n - 1);
8     }
9     else {
10        printf("BOUM!\n");
11    }
12 }
13
14 int main ()
15 {
16     int n;
17     printf("Count from? ");
18     scanf("%d", &n);
19     countdown(n);
20     return 0;
21 }
```