Langages de Programmation, Interprétation, Compilation

Christine Paulin
Christine.Paulin@universite-paris-saclay.fr

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

LDD3 Informatique, Mathématiques - Magistère Informatique

2025–26

Traduction en assembleur d'un langage impératif

Introduction

- La compilation est un processus en plusieurs étapes.
- Etape d'analyse :
  - extraire la structure et le sens du programme source reçu en entrée.
  - identifier des programmes "mal formés"
- Etape de synthèse :
  - transforme l'arbre de syntaxe abstraite d'un programme *source* en un programme *cible* "équivalent"
  - le programme cible est écrit dans un fichier
- Le langage cible peut être par exemple :
  - un autre langage de haut niveau (C, javascript . . .), on utilise alors la plateforme d'exécution du langage cible ;
  - du bytecode : code bas niveau pour une machine virtuelle (java, caml ou python) qui peut être ensuite interprété ;
  - un langage assembleur, exécuté sur une architecture matérielle donnée (Intel x86, ARM, MIPS, RISC-V. . .) après assemblage et édition de liens.

Nous allons étudier comment transformer un programme IMP en code pour l'architecture MIPS.

Partie 1 : Introduction

1. Traduction en assembleur d'un langage impératif
   - Architecture cible
   - Langage assembleur MIPS
   - Traduction complète pour IMP

Architecture MIPS

- Architecture simple mais réelle appelée MIPS (Microprocessor without Interlocked Pipeline Stages)
- Processeur RISC (Reduced Instruction Set Computer), 32 bits qui est un précurseur de l'architecture moderne RISC-V.
- Utilisé dans les systèmes embarqués, imprimantes, routeurs. . .

Les deux principaux composants de l'architecture sont :

- un processeur, avec un petit nombre de registres et des unités de calcul qui opèrent sur les registres
- une grande quantité de mémoire, où sont stockées à la fois des données et le programme à exécuter lui-même.

Architecture MIPS : mémoire

- l'octet, une unité universelle comportant 8 bits, représentée en code hexadécimal par deux caractères $0-9, a-f$ chacun désignant un chiffre de 0 à 15.
- le mot mémoire : espace alloué à une donnée « ordinaire », sur une architecture 32-bits un mot mémoire est formé de 4 octets soit 32 bits ou encore 8 caractères hexadécimaux.

Codage des entiers

Un entier machine est représenté sur 32 bits

- Entier non signé : entre 0 et $2^{32} - 1$
- Entier signé : -2^{31} et $2^{31} - 1$
  un nombre négatif p est représenté en “complément à deux” par la valeur $2^{32} + p$, le bit de poids fort (bit de signe) d'un tel nombre est toujours 1
  Revien à calculer module 2^{32}

Mémoire

- 32 registres “généraux” accessibles directement par l’unité de calcul, chacun stocke un mot mémoire
- Une mémoire principale de 2^{32} octets (4Go)
  - chaque octet est associé à une adresse (entier entre 0 et $2^{32} - 1$
  - les adresses des données “ordinaires” sont (en générale) des multiples de 4
  - l’accès à la mémoire est relativement coûteux

Boucle d'exécution

- Le programme à exécuter est une séquence d'instructions stockée de manière contiguë dans la mémoire
- Chaque instruction est codée sur 4 octets
- Un registre spécial pc (program counter, ou pointeur de code) contient l'adresse de l'instruction courante

L'exécution d'un programme procède en répétant le cycle suivant :

1. lecture d'un mot mémoire à l'adresse pc (fetch),
2. interprétation des octets lus comme une instruction (decode),
3. exécution de l'instruction reconnue (execute),
4. accès mémoire pour transfert entre registre(s) et mémoire (memory)
5. mise à jour des registres (dont pc pour passer à l'instruction suivante)

C'est le pipeline de traitement des instructions

Instructions

Trois catégories principales d'instructions :

- instructions arithmétiques
- accès à la mémoire : transfert des valeurs de la mémoire vers les registres ou inversement,
- instructions de contrôle : modifier le pointeur de code pc.

Format des instructions sur 32 bits :

- les 6 premiers désignent l'opération à effectuer (opcode)
- les suivants donnent les paramètres ou des précisions éventuelles sur l'opération (afin de distringuer plus de $2^6 = 64$ opérations).

Instructions : exemples

- instruction 9 (`addiu`) : $r_t \leftarrow r_s + n$ avec n entier non signé de 16 bits.
  $r_{17} \leftarrow r_5 + 42$

op	r_s	r_t	n
001001	00101	10001	0000000000101010

- instruction 15 (`lui`) : place un nombre de 16 bits dans les deux octets supérieurs d'un registre.
  Pour mettre 42 dans les deux octets supérieurs du registre numéro 17 :

op		r_t	n
001111	00000	10001	0000000000101010

Opérations binaires

- Un opcode spécifique (`SPECIAL`) : 000000
- Les 6 derniers bits (appelés la fonction) précisent l'opération : addition, multiplication, décalage...
- Ces instructions prennent trois paramètres : deux registres r_s et r_t pour les opérandes, et un registre de destination r_d.
- Exemple $r_3 \leftarrow r_1 + r_2$: (code de fonction de l'addition : 32)

op	r_s	r_t	r_d		*f*
000000	00001	00010	00011	00000	100000

- Certaines opérations ignorent le paramètre r_s et utilisent une donnée constante sur les 5 bits libres.
- Exemple : décalage vers la gauche (code de fonction 0) : $r_d \leftarrow r_t << n$ avec n un entier sur 5 bits (<32)
  Le code pour $r_6 \leftarrow r_4 << 5$ est

op		r_t	r_d	*k*	*f*
000000	00000	00100	00110	00101	000000

Accès à la mémoire

- Paramètres : deux registres r_s et r_t et un entier signé n sur 16 bits
- Lecture d'un mot mémoire (code 35) : r_s contient une adresse a, n est un entier signé sur 16 bits qui représente un décalage, on transfère le contenu de la mémoire à l'adresse $a + n$ dans le registre r_t
- Exemple : placer dans le registre 3 la donnée trouvée à l'adresse stockée dans le regitre 5 + 8 octets :

op	r_s	r_t	n
100011	00101	00011	0000000000001000

- Ecriture d'un mot mémoire : r_s contient une adresse a, n est un entier signé sur 16 bits qui représente un décalage, on transfère le contenu de la mémoire à l'adresse $a + n$ dans le registre r_t

Ce schéma général de découpage vaut encore pour les instructions d'accès à la mémoire. L'instruction de lecture d'un mot mémoire par exemple prend en paramètres . Le registre r_s est supposé contenir une adresse, et l'entier n est appelé décalage. L'instruction de lecture calcule une adresse a en additionnant l'adresse de base r_s et le décalage n, et transfère vers le registre r_t le mot mémoire lu à l'adresse a. Le code de cette opération est 35. Ainsi, pour placer dans le registre numéro 3 la donnée trouvée à l'adresse obtenue en ajoutant 8 octets à l'adresse donnée par le registre numéro 5, on a la

Opérations de contrôle

- saut conditionnel (opcode 7) :
  - paramètres un registre r_s et un entier n signé sur 16 bits,
  - avance de n instructions dans le code du programme seulement si la valeur de r_s est strictement positive
  - Exemple : reculer de 5 instructions lorsque la valeur du registre 3 est strictement positive

op	r_s	r_t	*n*
000111	00011	00000	1111111111111011

- saut absolu (opcode 2)
  - paramètre : un entier sur 26 bits interprété comme l'adresse a d'une instruction
  - positionne le pointeur pc à l'adresse a
  - Exemple : aller à l'instruction d'adresse 0x00efface

op	*a*
000010	00111011111111101011001110

Partie 1 : Introduction

1. Traduction en assembleur d'un langage impératif
   - Architecture cible
   - Langage assembleur MIPS
   - Traduction complète pour IMP

Langage assembleur

- Langage binaire
  - un modèle de mémoire/calcul bas niveau particulier
  - un encodage des instructions spécifiques
- Langage d'assemblage ou assembleur
  - forme textuelle des instructions, plus facile à mémoriser
  - traduction ensuite automatique vers le langage binaire
  - moins dépendant de l'architecture cible

Langage assembleur

En langage assembleur on a en particulier :

- des écritures textuelles pour les instructions,
- la possibilité d'utiliser des étiquettes symboliques plutôt que des adresses explicites,
- une allocation statique des données globales,
- quelques pseudo-instructions, qui correspondent à des combinaisons simples d'instructions réelles.

Assembleur MIPS

- Ecriture de programmes en assembleur MIPS
- Utilisation du simulateur MARS (Pete Sanderson et Ken Vollmar à Missouri State University)

  `https://computerscience.missouristate.edu/mars-mips-simulator.htm`

  - mode commande en ligne
  - mode graphique pour le débogage et simulation pas à pas
- Usage : `java -jar Mars4_5.jar`

Registres

Dans l'assembleur MIPS, les 32 registres sont désignés par leur numéro, de `$0` jusqu'à `$31` ou par leur nom.

n	*nom*
$0	$zero
$1	$at
$2	$v0
$3	$v1
$4	$a0
$5	$a1
$6	$a2
$7	$a3

n	*nom*
$8	$t0
$9	$t1
$10	$t2
$11	$t3
$12	$t4
$13	$t5
$14	$t6
$15	$t7

n	*nom*
$16	$s0
$17	$s1
$18	$s2
$19	$s3
$20	$s4
$21	$s5
$22	$s6
$23	$s7

n	*nom*
$24	$t8
$25	$t9
$26	$k0
$27	$k1
$28	$gp
$29	$sp
$30	$fp
$31	$ra

Registres

- Les 24 registres `$v*`, `$a*` (à l'exception de `at`), `$t*` et `$s*` sont des registres ordinaires, on peut les utiliser librement pour des calculs
- Les 8 autres registres ont des rôles particuliers :
  - `$gp`, `$sp`, `$fp` et `$ra` contiennent des adresses utiles pour les appels de fonction
  - `$zero` contient toujours la valeur 0 et ne peut pas être modifié,
  - les 3 registres `$at` et `$k0` et `$k1` sont réservés respectivement pour l'assembleur et pour le système (il ne faut donc pas les utiliser).
- Il y a aussi des registres spécialisés :
  - `$pc` le pointeur de code qui désigne l'instruction à exécuter
  - `$hi`, `$lo` pour la multiplication ou division d'entiers 32 bits

Programme MIPS

- Une zone d'instructions, introduite par `.text`
- Une zone de données "statiques" , introduite par `.data`
- Une instruction peut être précédée d'une étiquette `etiq :` qui permet de faire référence à l'adresse du code de cette instruction
- un commentaire dans le fichier assembleur commence par \# et se termine au retour à la ligne
- Une instruction commence par un mot clé appelé mnémonique, désignant l'opération à effectuer, qui est suivi d'un certain nombre de paramètres séparés par des virgules.
- Exemple d'instructions (mettre l'entier 42 dans le registre `$t0`, puis transférer la valeur de `$t0` dans `$t1`

```
li  $t0 , 42
move $t1 , $t0
```

Pseudo-instructions

- Une pseudo-instruction assembleur peut être traduite en plusieurs instructions binaires
- `li $t0, n` avec n un entier de 32 bits qui se décompose en $n_h n_l$ se traduit en

```
lui $at, n_h
ori $t0, n_l
```

- move $t0, $t1 est transformé en addu $t0, $0, $t1

Arithmétique

Les opérations arithmétiques et logiques MIPS suivent le format « trois adresses » :

```
<mnemo>   <dest>, <r1>, <r2>
```

- Opérations arithmétiques : add, sub, mul, div, rem (remainder : reste), ...
- Opérations logiques : `and`, `or`, xor (exclusive or), ...
- Comparaisons : seq (equal : égalité), sne (not equal : inégalité) slt (less than : $<$), sle (less or equal : $\leqslant$), sgt (greater than : $>$), sge (greater or equal : $\geqslant$), ...
- opérations unaires classiques avec un seul opérande : abs (valeur absolue), neg (opposé), not (négation logique).
- variantes pour les opérations non signées (addu, subu...)
- variantes pour une seconde opérande constante (addi, ...)
- Décalages et rotations : sll (shift left logical), sra (shift right arithmetic), srl (shift right logical), rol (rotation left), ror (rotation right), ... par défaut, le décalage est une valeur immédiate.

Exemple

Séquence d'instructions pour évaluer la comparaison 3*4 + 5 < 2*9

		$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7	$
`li`	`$t0, 3`	3								
`li`	`$t1, 4`	3	4							
`li`	`$t2, 5`	3	4	5						
`li`	`$t3, 2`	3	4	5	2					
`li`	`$t4, 9`	3	4	5	2	9				
`mul`	`$t5, $t0, $t1`	3	4	5	2	9	12			
`add`	`$t6, $t5, $t2`	3	4	5	2	9	12	17		
`mul`	`$t7, $t3, $t4`	3	4	5	2	9	12	17	18	
`slt`	`$t8, $t6, $t7`	3	4	5	2	9	12	17	18	

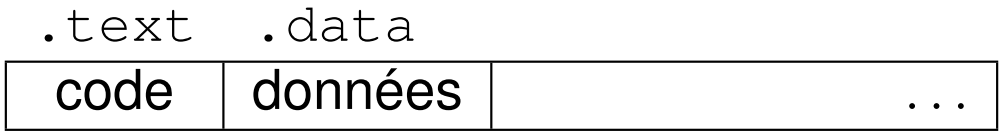
Exemple : version optimisée

		$t0	$t1
`li`	`$t0, 3`	3	
`li`	`$t1, 4`	3	4
`mul`	`$t0, $t0, $t1`	12	4
`addi`	`$t0, $t0, 5`	17	4
`li`	`$t1, 9`	17	9
`sll`	`$t1, $t1, 1`	17	18
`slt`	`$t0, $t0, $t1`	1	18

Données statiques

- Le programme peut réserver de l'espace pour des données globales
  - variables globales
  - chaînes de caractères
  - ...

En mémoire, les données ainsi déclarées sont placées dans une zone située après la zone réservée aux instructions du programme.



Format des données

La définition d'une donnée ou d'un groupe de données combine :

- la déclaration d'une étiquette symbolique, un nom qui désigne l'adresse où la donnée est stockée
- une ou plusieurs valeurs initiales.
- un mot-clé précise la nature des données fournies :
  - .word pour une valeur 32 bits (4 octets),
  - .byte pour une valeur d'un unique octet (8 bits),
  - .asciiz pour une chaîne de caractères au format ASCII (un caractère par octet, la chaîne étant terminée par l'octet zéro (0x00).

Format des données : exemples

- donnée désignée par le nom reponse et valant 42 :

```
reponse:  .word 42
```

- Tableau comportant une suite d'entiers, l'adresse `prems` est celle de la première entrée

```
prems:  .word 2 3 5 7 11 13 17 19
```

L'adresse réservée pour chaque donnée est calculée au moment de l'assemblage et remplacée en dur dans le programme.

Accès à la mémoire

Le format standard des adresses mémoire en MIPS contient deux composantes :

- une adresse de base, donnée par la valeur d'un registre $r,
- un décalage, donné par une constante d (16 bits, signée).

On note d($r) une telle adresse, dont la valeur est donc $r + d. Transferts possibles :

- transférer une valeur depuis une adresse mémoire vers un registre (lecture, load)
- depuis un registre vers une adresse mémoire (écriture, store).
- Exemples

```
lw $t0, 8($a1)
sw $t0, 0($t1)
```

- Les mnémoniques lw et sw signifient respectivement Load Word et Store Word.

Variantes

- Des variantes existent pour lire ou écrire seulement un octet ou un demi-mot ou encore un double mot
- On peut omettre un décalage de 0

```
sw $t0, ($t1)
```

- on peut utiliser une adresse désignée par une étiquette

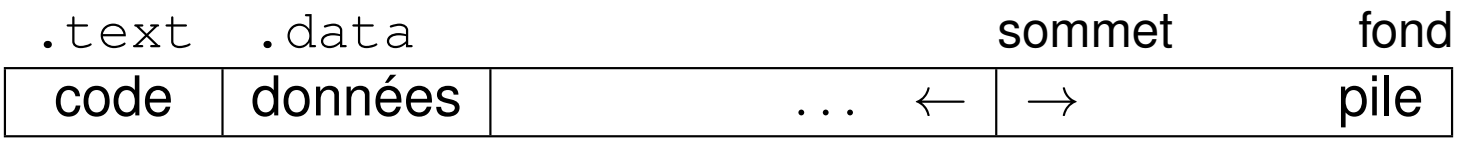
```
lw $t0, prems
lw $t1, prems+4
lw $t2, prems+12
```

- une instruction la (Load Address) permet de récupérer ou calculer une adresse, sans lire son contenu.

```
la $t0, prems
```

Pile

- Le mécanisme des données statiques n'est pas suffisant dans un programme ordinaire
- On a besoin de pouvoir stocker des données en mémoire
- La quantité d'espace nécessaire n'est pas forcément prévisible, on parle d'une allocation dynamique de la mémoire, réalisée à l'exécution
- Le programme doit néanmoins autant que possible libérer l'espace qui n'est plus utilisé
- La partie haute de la mémoire est organisée sous forme de pile :
  - Structure linéaire dont une extrémité est appelée le fond et l'autre le sommet
  - l'ajout et le retrait se font uniquement au sommet
  - le premier élément retiré est le dernier arrivé



- le fond de la pile est calé sur l'adresse la plus haute,
- la pile croît en s'étendant vers les adresses inférieures.
- le sommet de la pile est l'adresse mémoire la plus basse utilisée par la pile : stockée dans le registre $sp.

Opérations sur la pile

- Consulter la valeur au sommet de la pile (peek) :

```
lw    $t0, 0($sp)
```

- Consulter des éléments plus profonds dans la pile : +4 octets par élément à sauter.
  4ème élément : décalage de 12 octets.

```
lw    $t0, 12($sp)
```

- Ajouter un élément au sommet de la pile (push)

```
addi $sp, $sp, -4
sw    $t0, 0($sp)
```

- Retirer l'élément au sommet de la pile

```
addi $sp, $sp, 4
```

- L'opération usuelle pop, consiste à récupérer l'élément au sommet tout en le retirant de la pile

```
lw    $t0, 0($sp)
addi $sp, $sp, 4
```

Sauts et branchements

- les instructions, les données statiques ont toutes une adresse
- les adresses peuvent avoir des étiquettes
- les étiquettes servent de cible aux instructions de saut
- exemple : si les registres $t0 et $t1 ont même valeur, aller à l'instruction d'étiquette lab sinon continuer à l'instruction suivante.

```
beq  $t0, $t1, lab
```

- variantes de comparaison

$=$	$\neq$	$<$	$\leqslant$	$>$	$\geqslant$
`beq`	`bne`	`blt`	`ble`	`bgt`	`bge`

- variante test à 0 : beqz, bnez, bltz, ...
- possibilité d'un second paramètre immédiat
- une pseudo-instruction b lab provoque un branchement inconditionnel vers l'instruction d'étiquette lab.

Exemple

Calcul de la factorielle de la valeur de $a0 dans le registre $v0.

```
1           move $t0, $a0
2           li   $t1, 1
3           b    test
        loop:
4           mul  $t1, $t1, $t0
5           addi $t0, $t0, -1
        test:
6           bgtz $t0, loop
7           move $v0, $t1
```

Saut

- Branchement limité à des déplacements "locaux" (limité à un décalage de 2^{15} octets)
- Les instructions de saut définissent la prochaine instruction de manière absolue
- version immédiate

```
j       label
```

- adresse calculée dans un registre

```
jr      $t0
```

- utilisées pour un saut non local, comme un appel de fonction.
- instructions spécifiques qui sauvegardent une adresse de retour dans le registre $ra, qui permet plus tard de revenir à l'exécution de la séquence en cours.

```
jal   label
jalr  $t0
```

Appels système

- Certaines fonctionnalités dépendant du système d'exploitation
- L'assembleur fait appel aux bibliothèques système
- Fonctionnalités du simulateur MARS
  - instruction syscall,
  - le registre $v0 contient le code du service
  - le registre $a0 sert éventuellement de paramètre

service	code	arg.	rés.
affichage	1 (entier) 4 (chaîne) 11 (ascii)	`$a0`	
lecture	5 (entier)		`$v0`
arrêt	10		
extension mémoire	9	`$a0`	`$v0`

Hello, world

```
        .text
main:   li    $v0, 4
        la    $a0, hw
        syscall
        li    $v0, 10
        syscall

        .data
hw:     .asciiz "hello␣world\n"
```

Partie 1 : Introduction

1. Traduction en assembleur d'un langage impératif
   - Architecture cible
   - Langage assembleur MIPS
   - Traduction complète pour IMP

Module auxiliaire MIPS

Représentation efficace des concaténations de chaînes de caractères.

```
type asm =
  | Nop
  | S of string
  | C of asm * asm

let (@@) x y = C (x, y)

type program = { text: asm; data: asm; }
```

Concaténer deux fragments a_1 et a_2 de code assembleur :
`C(`a_1`,` a_2`)` qui s'écrit a_1 `@@` a_2

Abbréviations pour les registres

Pour faciliter la production de code MIPS sous cette représentation, on définit en plus de cela quelques constantes pour les registres utilisés

```
let t0 = "$t0"
let t1 = "$t1"
let a0 = "$a0"
let v0 = "$v0"
let sp = "$sp"
let ra = "$ra"
```

Commandes pour la génération d'instructions

L'expression Caml add t0 t0 t1 produit un élément de type asm correspondant au code assembleur "␣␣add␣$t0,␣$t0,␣$t1\n".

```
open Printf
let li   r1 i       = S(sprintf "␣␣li␣␣␣%s,␣%i"      r1 i)
let la   r1 x       = S(sprintf "␣␣la␣␣␣%s,␣%s"      r1 x)
let move r1 r2      = S(sprintf "␣␣move␣%s,␣%s"      r1 r2)

let add  r1 r2 r3   = S(sprintf "␣␣add␣␣%s,␣%s,␣%s" r1 r2
let addi r1 r2 i    = S(sprintf "␣␣addi␣%s,␣%s,␣%d" r1 r2
let mul  r1 r2 r3   = S(sprintf "␣␣mul␣␣%s,␣%s,␣%s" r1 r2
let slt  r1 r2 r3   = S(sprintf "␣␣slt␣␣%s,␣%s,␣%s" r1 r2
let and_ r1 r2 r3   = S(sprintf "␣␣and␣␣%s,␣%s,␣%s" r1 r2

let j    l          = S(sprintf "␣␣j␣␣␣␣%s"          l)
let jal  l          = S(sprintf "␣␣jal␣␣%s"          l)
let jr   r1         = S(sprintf "␣␣jr␣␣␣%s"          r1)
let beqz r1 l       = S(sprintf "␣␣beqz␣%s,␣%s"      r1 l)
let bnez r1 l       = S(sprintf "␣␣bnez␣%s,␣%s"      r1 l)
let bltz r1 l       = S(sprintf "␣␣bltz␣%s,␣%s"      r1 l)
```

Commandes de lecture/écriture

Dans l'ordre : le registre sur lequel on agit, le décalage, et le registre donnant l'adresse de base.
On écrit donc lw t0 4 t1 pour générer l'instruction lw $t0, 4($t1)

```
let lw   r1 o r2 = S(sprintf "  lw   %s, %d(%s)" r1 o r
let sw   r1 o r2 = S(sprintf "  sw   %s, %d(%s)" r1 o r
let lbu  r1 o r2 = S(sprintf "  lbu  %s, %d(%s)" r1 o r
```

Données statiques

```
let ilist = function
  | []     -> ""
  | [i]    -> sprintf "%d" i
  | i :: l -> sprintf "%d, %s" i (ilist l)
let dword l  = S(sprintf "  .word %s" (ilist l))
let asciiz s = S(sprintf "  .asciiz %s" s)
```

Exemple

```
  # built-in atoi
atoi:
  li    $v0, 0
  li    $t1, 10

atoi_loop:
  lbu   $t0, 0($a0)
  beqz  $t0, atoi_end
  addi  $t0, $t0, -48
  bltz  $t0, atoi_error
  bge   $t0, $t1 atoi_error
  mul   $v0, $v0, $t1
  add   $v0, $v0, $t0
  addi  $a0, $a0, 1
  j     atoi_loop

atoi_error:
  li    $v0, 10
  syscall

atoi_end:
  jr    $ra
```

```
let built_ins =
  comment "built-in atoi"
  @@ label "atoi"
  @@ li    v0 0
  @@ li    t1 10

  @@ label "atoi_loop"
  @@ lbu   t0 0(a0)
  @@ beqz  t0 "atoi_end"
  @@ addi  t0 t0 (-48)
  @@ bltz  t0 "atoi_error"
  @@ bge   t0 t1 "atoi_error"
  @@ mul   v0 v0 t1
  @@ add   v0 v0 t0
  @@ addi  a0 a0 1
  @@ j     "atoi_loop"

  @@ label "atoi_error"
  @@ li    v0 10
  @@ syscall

  @@ label "atoi_end"
  @@ jr    ra
```

Pile

```
let push r =
  addi sp sp (-4) @@ sw r 0(sp)
let pop r =
  lw r 0(sp) @@ addi sp sp 4
```

Génération de l'assembleur

```
let rec print_asm fmt a =
  match a with
  | Nop          -> ()
  | S s          -> fprintf fmt "%s\n" s
  | C (a1, a2) ->
     let () = print_asm fmt a1 in
     print_asm fmt a2

let print_program fmt p =
  fprintf fmt ".text\n";
  print_asm fmt p.text;
  fprintf fmt ".data\n";
  print_asm fmt p.data
```

Traduction de IMP vers MIPS

- Choisir une stratégie d'évaluation
- On commence par un modèle de calcul des expressions à l'aide d'une pile
  - la fonction tr_expr, appliquée à une expression e, produit un code assembleur qui calcule la valeur de e et place le résultat dans le registre $t0,
  - toutes les valeurs intermédiaires sont enregistrées sur la pile.

Traduction des expressions

```
let rec tr_expr = function
  | Cst n -> li t0 n
  | Var x -> la t0 x @@ lw t0 0(t0)
  | Bop(bop, e1, e2) -> let op = match bop with
                          | Add -> add
                          | Mul -> mul
                          | Lt  -> slt
                          | And -> and_
                        in
                        tr_expr e1
                      @@ push t0
                      @@ tr_expr e2
                      @@ pop t1
                      @@ op t0 t1 t0
```

Traduction des instructions élémentaires

Traduction d'une instruction

```
let rec tr_instr = function
  | Print e ->
      tr_expr e
    @@ move a0 t0
    @@ li v0 11
    @@ syscall
  | Set(x, e) ->
      tr_expr e
    @@ la t1 x
    @@ sw t0 0(t1)
```

Traduction des boucles

Fonction auxiliaire new_label qui crée de nouvelles étiquettes.

```
let new_label =
  let cpt = ref (-1) in
  fun s -> incr cpt; Printf.sprintf "__%s_%d" s !cpt
```

Traduction des boucles :

- Une étiquette pour chaque cible de saut
  - un saut conditionnel (beqz) pour sortir de la boucle si le test est négatif, en ciblant une étiquette end_label placée après le corps la boucle,
  - un saut inconditionnel (j) pour revenir au test (étiquette loop_label après chaque exécution du corps de la boucle.

Traduction des boucles : code

```
| While(e, s) ->
    let loop_label = new_label "loop"
    and end_label = new_label "endloop"
    in
    label loop_label
    @@ tr_expr e                    (* test *)
    @@ beqz t0 end_label
    @@ tr_seq s                     (* loop body *)
    @@ j loop_label
    @@ label end_label
```

Traduction des tests

- Une étiquette pour la branche **else**,
- un saut inconditionnel à la fin de la branche **then** vers une étiquette end_label pour ne pas passer par la branche **else**.

```
| If(e, s1, s2) ->
    let else_label = new_label "else"
    and end_label = new_label "endif"
    in
    tr_expr e                          (* test *)
  @@ beqz t0 else_label
  @@ tr_seq s1                         (* then branch *)
  @@ j end_label
  @@ label else_label
  @@ tr_seq s2                         (* else branch *)
  @@ label end_label
```

Traduction d'une suite d'instructions

```
and tr_seq = function
  | []    -> nop
  | [i]   -> tr_instr i
  | i::s -> tr_instr i @@ tr_seq s
```

Identification des données statiques

```
module VSet = Set.Make(String)
let rec vars_expr = function
  | Cst _ -> VSet.empty
  | Var x -> VSet.singleton x
  | Bop(_, e1, e2) ->
     VSet.union (vars_expr e1) (vars_expr e2)
let rec vars_instr = function
  | Print e -> vars_expr e
  | Set(x, e) ->
     VSet.add x (vars_expr e)
  | While(e, s) ->
     VSet.union (vars_expr e) (vars_seq s)
  | If(e, s1, s2) ->
     VSet.union (vars_expr e)
                (VSet.union (vars_seq s1) (vars_seq s2))
and vars_seq = function
  | []   -> VSet.empty
  | i::s -> VSet.union (vars_instr i) (vars_seq s)
```

Traduction du programme

- traduction des instructions ;
- allocation de ses variables globales (initialisées à 0).

```
let tr_prog p =
  let text = tr_seq p in
  let vars = vars_seq p in
  let data = VSet.fold
    (fun id code -> label id @@ dword [0] @@ code)
    vars nop
  in
  { text; data }
```

Synthèse

A retenir

- Les principes généraux d'un code binaire MIPS
  - modèle de calcul (opérations sur les registres, transferts mémoire)
  - modèle de codage des instructions
- Les éléments de base de l'assembleur MIPS
  - organisation de la mémoire (registres, programme, données, pile)
  - instructions élémentaires (arithmétique, comparaison, sauts)
  - utilisation des registres et de la pile pour calculer des expressions
  - schéma de compilation d'une conditionelle et d'une boucle

Savoir faire

- Lire et écrire un programme simple en assembleur MIPS
- Ecrire les bases d'un compilateur en utilisant l'interface Caml MIPS