Code Machine
Code Assembleur
Processeurs RISC
(Byte code)

Luc Maranget
http://pauillac.inria.fr/~maranget

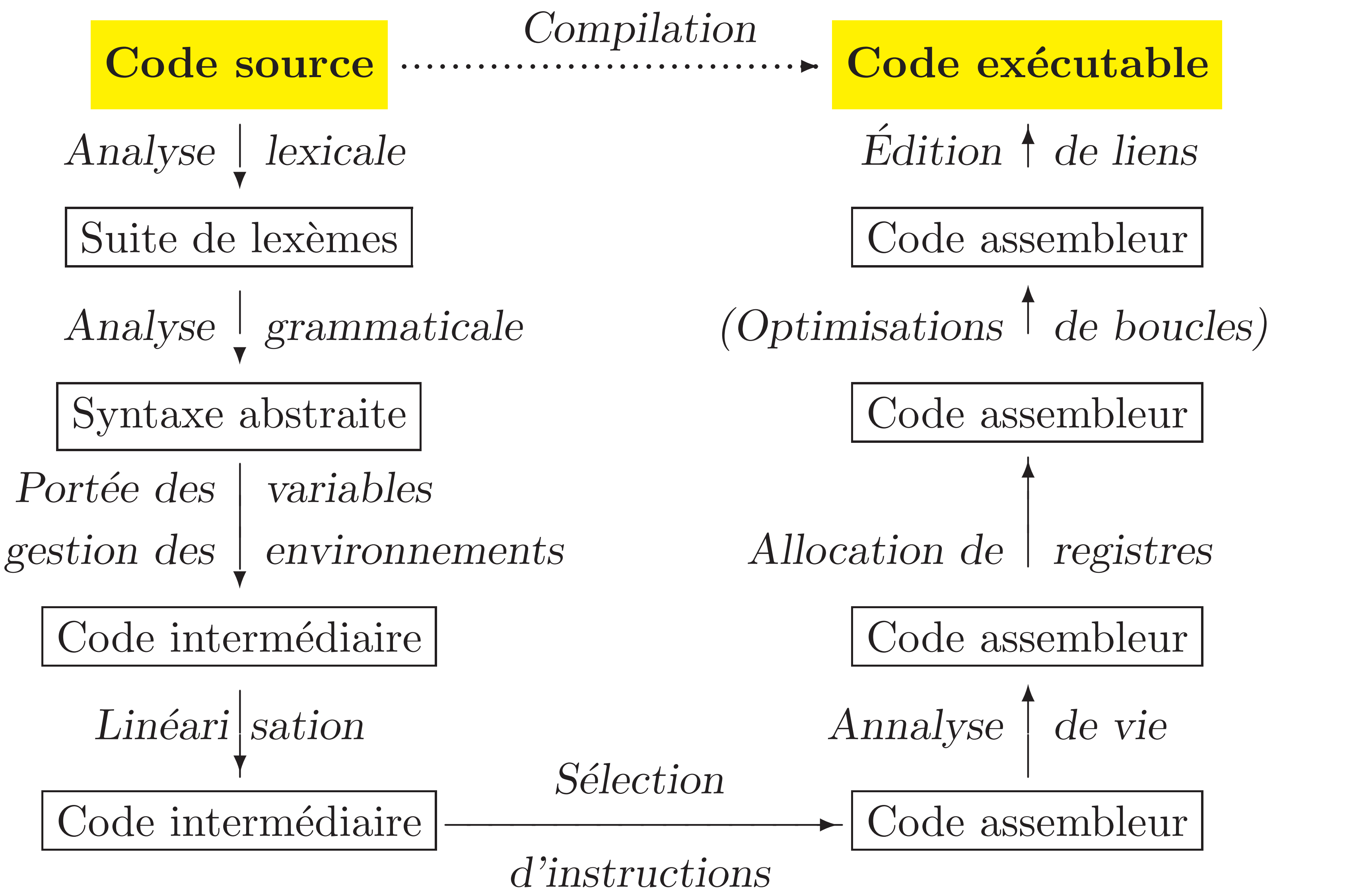
2005

Qu'est-ce qu'un compilateur

Une réponse détaillée est dans le poly.

Un compilateur est un traducteur d'un langage (de programmation) vers l'assembleur.

Un compilateur se décompose en *phases* :


Compilation
Code source
Code exécutable
Analyse lexicale
Suite de lexèmes
Analyse grammaticale
Syntaxe abstraite
Portée des variables
gestion des environnements
Code intermédiaire
Linéarisation
Code intermédiaire
Sélection d'instructions
Code assembleur
Annalyse de vie
Code assembleur
Allocation de registres
Code assembleur
(Optimisations de boucles)
Code assembleur
Édition de liens

Programme du cours

1. Assembleur
2. Sémantique (description de Pseudo-Pascal)
3. Analyse lexicale
4. Analyse grammaticale
5. Code intermédiaire I (compilation)
6. Code intermédiaire II (optimisations)
7. Sélection ($\longrightarrow$ code assembleur)
8. Analyse de durée de vie
9. Allocation de registres

Examen

1. Un écrit

Les processeurs

L'architecture des ordinateurs varie peu du point de vue de l'utilisateur qui les programme « *directement* ».

Ils comportent principalement une mémoire et un processeur.

- La mémoire est un grand tableau de cases.
- Les registres sont quelques cases très rapidement accessibles (elle sont « *dans* » le processeur).
- Le processeur exécute les instructions selon un schéma dont le principe varie peu. Un registre interne `pc` indique où se trouve l'instruction à exécuter.
  - Lire l'instruction pointé par `pc`.
  - La décoder et l'exécuter (par ex. ranger la somme de deux registres dans un troisième).
  - À la fin, `pc` pointe sur une nouvelle instruction, ce peut être l'instruction qui suit, mais pas toujours (transferts).

Risc et Cisc

Les différences entre les divers (micro-)processeurs se voient surtout dans leur jeu d'instructions.

- Les (vieux) processeurs CISC (*Complex Instruction Set*)
  - Leurs instructions sont de taille variable et beaucoup réalisent des transferts avec la mémoire ; ils possèdent en général peu de registres (et pas uniformes)
  - Conçus avant 1985. Typiquement : Intel 8086 et Motorola 68000.
- Les (nouveaux) processeurs RISC (*Reduced Instruction Set*)
  - Instructions de taille fixe, régulières (*trois adresses*) dont peu font des transfert avec la mémoire. Ils possèdent en général beaucoup de registres (uniformes).
  - Conçus après 1985. Typiquement : Alpha, Sparc et Mips. G4.
- Le pentium d'Intel est un peu à part : il mélange tout.

La mémoire

Tous les processeurs modernes comportent une unité mémoire (MMU) et un système d'exploitation qui fait croire à chaque programme qu'il dispose de toute la mémoire.

Cela permet à chaque programme de choisir ses adresses indépendamment des autres programmes (qui peuvent être exécutés en même temps sur la même machine avec les mêmes adresses virtuelles mais des adresses réelles différentes).

Du point de vue de l'utilisateur, la mémoire est un (grand) tableau dont les indices sont les adresses.

Les tailles

- Le *byte* est la taille du contenu de ces cases, c'est souvent 8 bits (*octet*).
- Le *mot* est (plus ou moins) la taille des adresses, c'est souvent 32 ou 64 bits.

Groupage des cases de la mémoire

Si par exemple chaque case mémoire est un octet et que les adresses sont « 32 bits »

- Pour ranger une adresse en mémoire, il faut : 4 cases.
- L'adresse d'une adresse designe alors la première des quatre cases.
- Il est généralement préférable (voire obligatoire) que l'adresse d'une adresse soit *alignée*, c'est à dire multiple de quatre.
- L'alignement signifie que l'adresse des objets tenant sur 2^k bytes est un multiple de 2^k.

Les zones mémoire

Adresses hautes	Stack
	↓ ↑
	Données dynamiques Allouée dynamiquement
	Données statiques Modifiable
	Texte (programme) Non écrivable
Adresses basses	Réservé au système

Les registres du MIPS

Le MIPS comporte 32 registres généraux interchangeables, sauf

- le registre `zero` qui vaut toujours `0`, même après une écriture.
- Le registre `ra`, utilisé implicitement par certaines instructions pour sauver l'adresse de retour avant un saut.

Les autres registres ont des utilisations préférentielles, mais cela n'est strict que pour communiquer avec d'autres programmes (par exemple fournir ou utiliser des programmes en librairies) :

- passage (`a0`, ... `a3`) et retour (`v0`, `v1`) des arguments ;
- registres temporaires sauvegardés (`s0`, .. `s7`) ou non (`t0`, ... `t9`) par les appels de fonction.
- pointeurs de pile `sp`, `fp` ou de donnée `gp` ;
- réservés par l'assembleur `at` et le système `k0`, `k1`.

Les tailles Le *mot* est aussi la taille des registres, ici 32 bits. Ici, c'est aussi la taille d'une instruction !

Le jeu d'instructions

n	une constante entière
ℓ	une étiquette (adresse)

r	nom de registre
a	absolu (n ou ℓ)
o	opérande (r ou a)

La plupart des instructions suivent le modèle

- **add** r_1, r_2, o qui place dans r_1 la valeur $r_2 + o$.

Les instructions qui interagissent avec la mémoire sont uniquement les instructions `load` et `store`.

- **lw** $r_1, n(r_2)$ place dans r_1 le mot contenu à l'adresse $r_2 + n$.
- **sw** $r_1, n(r_2)$ place r_1 dans le mot contenu à l'adresse $r_2 + n$.

Les adresses *doivent* être alignées (il y a des instructions spécifique [plus lentes] pour les accès généraux).

Les instructions de contrôle conditionnel ou inconditionnel :

- **bne** r, a, ℓ saute à l'adresse ℓ si r et a sont différents,
- **jal** o qui sauve $pc + 4$ dans `ra` et saute à l'étiquette o.

Liste abrégée des instructions

Syntaxe	Effet
move r_1, r_2	$r_1 \leftarrow r_2$
add r_1, r_2, o	$r_1 \leftarrow o + r_2$
sub r_1, r_2, o	$r_1 \leftarrow r_2 - o$
mul r_1, r_2, o	$r_1 \leftarrow r_2 \times o$
div r_1, r_2, o	$r_1 \leftarrow r_2 \div o$
and r_1, r_2, o	$r_1 \leftarrow r_2$ land o
or r_1, r_2, o	$r_1 \leftarrow r_2$ lor o
xor r_1, r_2, o	$r_1 \leftarrow r_2$ lxor o
sll r_1, r_2, o	$r_1 \leftarrow r_2$ lsl o
srl r_1, r_2, o	$r_1 \leftarrow r_2$ lsr o
li r_1, n	$r_1 \leftarrow n$
la r_1, a	$r_1 \leftarrow a$

Syntaxe	Effet
lw $r_1, o(r_2)$	$r_1 \leftarrow tas.(r_2 + o)$
sw $r_1, o(r_2)$	$r_1 \rightarrow tas.(r_2 + o)$
slt r_1, r_2, o	$r_1 \leftarrow r_2 < o$
sle r_1, r_2, o	$r_1 \leftarrow r_2 \leq o$
seq r_1, r_2, o	$r_1 \leftarrow r_2 = o$
sne r_1, r_2, o	$r_1 \leftarrow r_2 \neq o$
j o	$pc \leftarrow o$
jal o	$ra \leftarrow pc + 4 \wedge pc \leftarrow o$
beq r, o, a	$pc \leftarrow a$ si $r = o$
bne r, o, a	$pc \leftarrow a$ si $r \neq o$
syscall	appel système
nop	ne fait rien

Les appels systèmes

Ils permettent l'interaction avec le système d'exploitation, et en dépendent. Le numéro de l'appel système est lu dans `v0` (attention, ce n'est pas la convention standard). Selon l'appel, un argument supplémentaire peut être passé dans `a0`.

Le simulateur SPIM implémente les appels suivants :

Nom	N^o	Effet
`print_int`	1	imprime l'entier contenu dans `a0`
`print_string`	4	imprime la chaîne en `a0` jusqu'à `'\000'`
`read_int`	5	lit un entier et le place dans `v0`
sbrk	9	alloue `a0` bytes dans le tas, retourne l'adresse du début dans `v0`.
exit	10	arrêt du programme en cours d'exécution

Langage d'assembleur et langage machine

Le langage d'assembleur est un langage symbolique qui donne des noms aux instructions (plus lisibles que des suites de bits). Il permet aussi l'utilisation d'étiquettes symboliques et de pseudo-instructions et de modes d'adressage surchargés.

Le langage machine est une suite d'instructions codées sur des mots (de 32 bits pour le MIPS). Les instructions de l'assembleur sont expansées en instructions de la machine à l'édition de lien. Les étiquettes sont donc résolues et les pseudo-instructions remplacées par une ou plusieurs instructions machine.

L'assemblage est la traduction du langage d'assembleur en langage machine. Le résultat est un fichier object qui contient, en plus du code, des informations de relocation qui permettent de lier *(linker)* le code de plusieurs programmes ensemble.

Pseudo-instructions

La décompilation du langage machine en langage assembleur est facile. Elle permet de présenter les instructions machine (mots de 32 bits) sous une forme plus lisible.

Exemple d'expansion (présentée sous une forme décompilée).

Assembleur	Langage machine	Commentaire
`blt` r, o, a	`slt $1,` r, o `bne $1, $0,` a	Justifie le registre `at ($1)` réservé par l'assembleur.
`li $t0, 400020`	`lui $1, 6` `ori $8, $1, 6804`	charge les 16 bits de poids fort puis les 16 bits de poids faible
`add $t0, $t1, 1`	`addi $8, $9, 1`	addition avec une constante
`move $t0, $t1`	`addu $8, $0, $9`	addition "unsigned" avec zéro

Exemple de programme assembleur

hello.spi

```
        .data
hello:  .asciiz "hello\n"    # hello pointe vers "hello\n\0"
        .text
        .globl __start
__start:
        li    $v0, 4         # la primitive print_string
        la    $a0, hello     # a0 l'adresse de hello
        syscall
```

Le programme est assemblé, chargé puis exécuté par :

```
spim -notrap -file hello.spi
```

Par convention le programme commence à l'étiquette __start.

Si on retire l'option −notrap, le chargeur ajoute un prélude qui se branche à l'étiquette main (remplacer alors __start par main).

if-then-else

On utilise des sauts conditionnels et inconditionnels :

Pascal la fonction minimun

```
if  t1 < t2 then t3 := t1 else  t3 := t2
```

Assembleur Mips

```
        blt   $t1, $t2, Then      # si t1 >= t2 saut à Then
        move  $t3, $t2            # t3 := t1
        j     End                 # saut à Fi
Then:   move  $t3, $t1            # t3 := t2
End:                              # suite du programme
```

Boucles

Pascal : calcule dans t2 = 0 la somme des entiers de 1 à t1

```
while t1 > 0 do begin t2 := t2 + t1; t1 := t1 −1 end
```

Programme équivalent

```
While:
    if t1 <= 0 then goto End
    else
      begin
        t2 := t2 + t1;
        t1 := t1 − 1;
        goto While
      end;
End:
```

Code Mips

```
While:
    ble  $t1, $0, End


    add  $t2, $t2, $t1
    sub  $t1, $t1, 1
    j    While

End:
```

Appel de procédure

Une procédure s'appelle et ne renvoie pas de résultat.

À la fin de la procédure, le contrôle doit être donné à l'instruction qui suit l'appel.

En pratique

Dans le MIPS.

- Pour appeler la procédure p, on utilise l'instruction **jal** p, qui transfère le contrôle en p, et sauve l'adresse de l'instruction suivante dans `ra`.
- À la fin de la procédure, on transfère le contrôle à l'adresse sauvée par **j $ra**.

Un exemple de procédure

Par exemple on définit une procédure writeln qui imprime un entier puis un retour à la ligne.

```
        .data
nl:     .asciiz "\n"
        .text
writeln:                # l'argument est dans a0
        li  $v0, 1      # le numéro de print_int
        syscall         # appel système
        li  $v0, 4      # la primitive print_string
        la  $a0, nl     # la chaîne "\n"
        syscall
        j   $ra         # saut à l'adresse ra
```

Le programme complet

```
        .data                # le tas
nl:     .asciiz  "\n"        # la chaîne "\n"
        .text                # la zône code
        .globl  __start
__start:
        li   $a0, 1          # a0 <- 1
        jal  writeln         # ra <- pc+1; saut à writeln
        li   $a0, 2          # on recommence avec 2
        jal  writeln
        j    Exit            # saut à la fin du programme
writeln:
        ...
Exit:                        # fin du programme
```

Noter la différence entre les instructions **j** et **jal**.

Une fonction

```
let succ x = x+1
```

C'est comme la procédure, il faut en plus rendre le résultat. On fixe un registre conventionel pour cet usage :

Code de la fonction

```
succ:
   add $v0, $a0, 1   # calcul
   j $ra             # retour
```

Usage de la fonction

```
# Afficher succ(a1)
   move $a0, $a1
   jal succ
   move $a0, $v0
   jal writeln
```

Procédures–fonctions récursives

```
let rec fact n = if n <= 0 then 1 else n * fact (n-1)
```

Compilation « naive »

```
fact:
  blez $a0, fact_0    # si a0 <= 0, saut
  sub $a0, $a0, 1     # décrément a0
  jal fact            # appel (récursif)
  mul $v0, $v0, $a0 # v0 <- a0 * v0
  j $ra
fact_0:
  li $v0, 1
  j $ra
```

Ou est le problème ?

Procédure récursive II

Lorsqu'une procédure est récursive plusieurs appels imbriqués peuvent être actifs simultanément :

$$f_n \begin{cases} a_0 \leftarrow a_0 - 1 \\ f_{n-1} \begin{cases} a_0 \leftarrow a_0 - 1 \\ \ldots \\ v_0 \leftarrow v_0 \times a_0 \end{cases} \\ v_0 \leftarrow v_0 \times a_0 \end{cases}$$

C'est bien sûr le même code qui est utilisé pour tous les appels. L'appel récursif va donc modifier `a0`. Or, `a0` est utile après le retour de l'appel.

Il faut donc sauver le contenu de `a0` avant l'appel quelque-part. Mais où ?

On utilise la pile.

La pile

Par convention, la pile grossit vers les adresses décroissantes. Le registre *sp* pointe vers le dernier mot utilisé.

Pour sauver un registre *r* sur la pile

```
sub $sp, $sp, 4   # alloue un mot sur la pile
sw  r, 0($sp)     # écrit r sur le sommet de la pile
```

Pour restaurer un mot de la pile dans un registre *r*

```
lw  r, 0($sp)     # lit le sommet de la pile dans r
add $sp, $sp, 4   # désalloue un mot sur la pile
```

En général, on alloue (**sub**) et désalloue (**add**) l'espace en pile par blocs pour plusieurs registres à la fois.

Conventions d'appel

En général :

- les arguments sont passés dans les registres `a0` à `a3` puis dans la pile.
- la ou les valeurs de retour dans `v0` et `v1`.

De même :

- les registres t_i sont sauvés par l'appelant (l'appelé peut les écraser)
- les registres s_i sont sauvés par l'appelé (qui doit les remettre dans l'état où il les a pris)

Mais ce n'est qu'une convention ! (celle proposé par le fabriquant)

La respecter permet de comminiquer avec d'autres programmes (qui la respectent également).

Choisir une autre convention est possible tant que l'on n'intéragit pas avec le monde extérieur.

Exemple : calcul de la factorielle

L'argument est dans `a0`, le résultat dans `v0`.

```
fact:   blez $a0, fact_0      # si a0 <= 0 saut à fact_0
        sub  $sp, $sp, 8      # réserve deux mots en pile
        sw   $ra, 0($sp)      # sauve l'adresse de retour
        sw   $a0, 4($sp)      # et la valeur de a0
        sub  $a0, $a0, 1      # décrément a0
        jal  fact             # v0 <- appel récursif (a0-1)
        lw   $a0, 4($sp)      # récupére a0
        mul  $v0, $v0, $a0    # v0 <- a0 * v0
        lw   $ra, 0($sp)      # récupére l'adresse de retour
        add  $sp, $sp, 8      # libère la pile
        j    $ra              # retour à l'appelant

fact_0: li   $v0, 1           # v0 <- 1
        j    $ra              # retour à l'appelant
```

Allocation dans le tas

Allocation statique Un espace est réservé au chargement.

```
Tableau:                  # adresse symbolique sur le début
        .align 2          # aligner sur un mot (2^2 bytes)
        .space 4000       # taille en bytes
```

Allocation dynamique En cours de calcul on peut demander au système d'agrandir le tas, à l'aide de l'appel système spécifique.

```
malloc:                   # procédure d'allocation dynamique
        li $v0, 9         # appel système n. 9
        syscall           # alloue une taille a0 et
        j  $ra            # retourne le pointeur dans v0
```

Un autre gestion de l'allocation

En pratique, pour éviter les appels système couteux, on peut allouer un gros tableau `Tas` (statiquement ou dynamiquement) dans lequel on s'alloue des petits blocs.

Par exemple on réserve un registre pour pointer sur le premier emplacement libre du tas.

```
__start :
        la   $t8, Tas          # on réserve le registre t8
         ...
array:
        sw   $a0, ($t8)        # écrit la taille dans l'entête
        add  $v0, $t8, 4       # v0 <-- adresse de la case 0
        add  $t8, $v0, $a0     # augmente t8 de la taille+1
        j    $ra
```

Le byte-code

C'est un jeu d'intructions inventé pour une machine virtuelle (ou abstraite) qui est alors interprété par un programme, lui même compilé dans l'assembleur de la machine réelle.

Avantages : le byte-code est indépendant de l'architecture, il est portable (fonctionne sur différentes machines).

Le jeu d'instructions peut être mieux adpapté au langage compilé.

Inconvénient : l'interprétation du byte-code ajoute au moins un facteur 5 en temps d'exécution (pour de très bons byte-codes).

Exemples : Java, Ocaml.

Micro-code : Dans une machine réelle, les instructions sont en fait interprétées par du micro-code (qui lui est vraiment câblé).

Compilation vers le byte-code

Les machines virtuelles possèdent généralement peu de registres.

Elles utlisent abondamment la pile.

- Pour les calculs.
- Pour les appels de fonction.

Ainsi si nous considérons la fonction `twice` en Caml :

```
let twice x = x+x
```

Exemple de compilation en bytecode

On obtient le bytecode suivant (`ocamlc -dinstr`).

```
twice:
        acc 0
        push
        acc 1
        addint
        return 1
```

Ce code se comprend si on sait que :

- La machine possède un registre dit *accumulateur*.
- L'instruction `acc` n copie le contenu de la n-ième position en pile dans l'accumulateur.
- L'instruction `push` empile le contenu de l'accumulateur.
- Les opérations arithmétiques (cf. `addint`) sont en pile. Le

premier argument est l'accumulateur, le second est dépilé, le résultat est rangé dans l'accumulateur.