Assembleur MIPS

David Delahaye

David.Delahaye@lirmm.fr

Faculté des Sciences

Master M1 2020-2021

Objectifs du chapitre

1. Manipuler un langage de programmation de bas niveau ;
2. Utiliser des jeux d'instructions pour écrire des mini-programmes ;
3. Comprendre la segmentation logique de la mémoire (pile, tas, etc.) ;
4. Comprendre le mécanismes d'appel de procédure (sous-programme) ;
5. Percevoir les principes de programmation universel.

Pourquoi ? [Strandth & Durand, 2005]

Une petite citation

Le rôle d'un informaticien n'est pas de concevoir des architectures, en revanche il a besoin d'un modèle de fonctionnement de l'ordinateur qui lui donne une bonne idée de la performance de son programme et de l'impact que chaque modification du programme aura sur sa performance.

Assimiler un tel modèle suppose un certain nombre de connaissances sur le fonctionnement d'un ordinateur, notamment le mécanisme d'appel de fonction, la transmission des paramètres d'une fonction à l'autre, l'allocation ou la libération d'espace mémoire, etc.

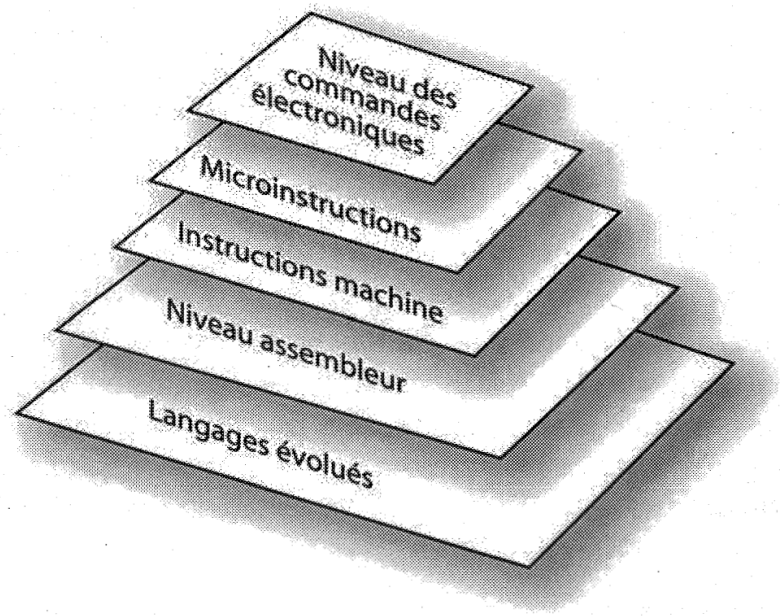
Conclusion

$\Rightarrow$ Apprendre l'architecture et un langage machine permet cela.

Niveaux de programmation

Langage machine

Le programmeur a le choix entre différents langages (assembleur, Ada, Java, C, etc.). La machine ne comprends que le langage machine (i.e., instructions binaires) !



- L'assembleur (langage d'assemblage) est le premier langage non binaire accessible au programmeur ;
- Code mnémoniques et symboles ;
- L'assembleur (programme traducteur) convertit le langage d'assemblage en langage machine ;
- Permet d'exploiter au maximum les ressources de la machine ;
- Dépend de la machine, de son architecture.

Interprétation et compilation

Interprétation

Conversion et exécution d'un programme en une seule étape : les instructions sont lues les unes après les autres et sont exécutées immédiatement par la machine.

- Pas de programme objet intermédiaire ;
- Répétition du travail de traduction à chaque exécution.

Compilation

Génération d'un programme équivalent au code source appelé code objet.

- Traduction réalisée une seule fois ;
- Exécution rapide et efficace.

Interprétation et compilation

Interprétation

Conversion et exécution d'un programme en une seule étape : les instructions sont lues les unes après les autres et sont exécutées immédiatement par la machine.

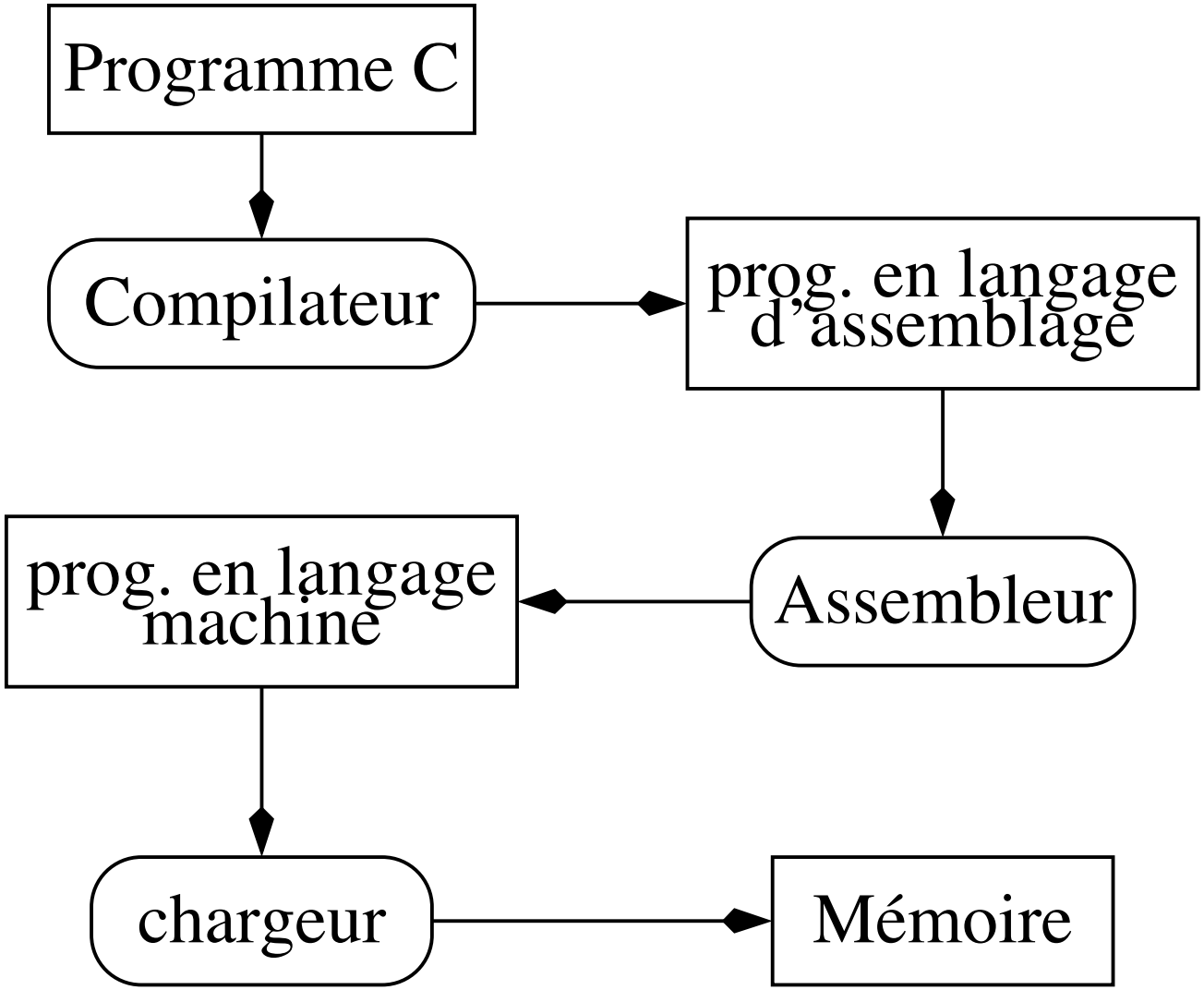
- Pas de programme objet intermédiaire ;
- Répétition du travail de traduction à chaque exécution.

Compilation

Deux types de compilation :

- Native : le code objet est de l'assembleur et sera traduit en langage machine. Cette compilation est donc dépendante de l'architecture, mais est la plus efficace ;
- « Bytecode » : le code objet est exécuté par une machine virtuelle. Cette compilation est donc indépendante de l'architecture, mais est en moyenne 5 fois moins efficace qu'une compilation native.

Compilation native



Langage d'assemblage

- Utilisés par les spécialistes $\Rightarrow$ optimisation ;
- Pour valoriser l'architecture spécifique de la machine ;
- Diagnostic d'erreurs (i.e., examen du contenu de la mémoire) ;
- L'assembleur est une variante symbolique du langage machine $\Rightarrow$ même jeu d'instructions ;
- Propre à chaque type de machine ;
- Permet d'accéder aux ressources de la machine (i.e., registres) ;
- Permet d'accéder aux facilités de traitement (e.g., décalage) ;

Le programmeur peut utiliser :

- Codes mnémoniques (jeu d'instruction) ;
- Étiquettes (adresse symboliques) ;
- Littéraux (constante numériques) ;
- Directives (pseudo instruction).

Processeurs CISC et RISC

CISC

- « Complex Instruction Set Computer » ;
- Jeu étendu d'instructions complexes ;
- Chaque instruction peut effectuer plusieurs opérations élémentaires ;
- Jeu d'instructions comportant beaucoup d'exceptions ;
- Instructions codées sur une taille variable.

RISC

- « Reduced Instruction Set Computer » ;
- Jeu d'instructions réduit ;
- Chaque instruction effectue une seule opération élémentaire ;
- Jeu d'instructions plus uniforme ;
- Instructions codées sur la même taille et s'exécutant dans le même temps (un cycle d'horloge en général).

Processeurs CISC et RISC

Répartition des principaux processeurs

CISC (pré-1985)	RISC (post-1985)
S/360 (IBM)	Alpha (DEC)
VAX (DEC)	PowerPC (Motorola)
68xx, 680x0 (Motorola)	MIPS
x86, Pentium (Intel)	PA-RISC (Hewlett-Packard)
	SPARC (Sun)

Les machines MIPS

- Langage MIPS : assembleur des processeurs MIPS (MIPS I à V, MIPS32, MIPS64) ;
- Introduit au début des années 1980 ;
- Inventé par John Henessy, inventeur de MIPS et président de Stanford aujourd'hui ;
- Développé par MIPS Technologies ;
- Processeur RISC, en 32 et 64 bits ;
- Souvent utilisé comme support pédagogique dans le milieu académique.
- Processeur des machines NEC, SGI (super-calculateurs), Sony PS, PS2, PSP, Nintendo (console), FreeBox, NeufBox (routeur) ;
- Nous allons nous intéresser à la norme du langage MIPS32 (32 bits).

Les registres du MIPS

Le MIPS comporte 32 registres généraux, nommés $r0$ à $r31$. Le registre $r0$, appelé *zero*, contient toujours la valeur 0, même après une écriture.
Les 31 autres registres sont interchangeables.
Néanmoins, ces 31 registres peuvent être, par convention :

- Réservés pour le passage d'arguments ($a0$-$a3$, ra) ou le renvoi de résultats ($v0$-$v1$) ;
- Considérés comme sauvegardés par l'appelé ($s0$-$s7$) ou non ($t0$-$t9$) lors des appels de fonctions ;
- Réservés pour contenir des pointeurs vers la pile (sp, fp) ou vers les données (gp) ;
- Réservés par le noyau ($k0$-$k1$) ;
- Réservés par l'assembleur (at).

Les registres du MIPS

Le MIPS propose trois types principaux d'instructions :

- Les instructions de transfert entre registres et mémoire ;
- Les instructions de calcul ;
- Les instructions de saut.

Seules les premières permettent d'accéder à la mémoire ; les autres opèrent uniquement sur les registres.

Instructions de transfert

Lecture et écriture

- Lecture (« load word ») :

  lw dest, offset(base) On ajoute la constante (de 16 bits) offset à

  l'adresse contenue dans le registre base pour obtenir une nouvelle adresse ; le mot stocké à cette adresse est alors transféré vers le registre dest.

- Écriture (« store word ») :

  sw source, offset(base) On ajoute la constante (de 16 bits) offset à

  l'adresse contenue dans le registre base pour obtenir une nouvelle adresse ; le mot stocké dans le registre source est alors transféré vers cette adresse.

Instructions de calcul

Principe

Ces instructions lisent la valeur de 0, 1 ou 2 registres dits arguments, effectuent un calcul, puis écrivent le résultat dans un registre dit destination.

Un même registre peut figurer plusieurs fois parmi les arguments et destination.

Instructions de calcul

Instructions nullaires

- Lecture d'une constante (« load immediate ») :

  li dest, constant

  La constante constant est transféré vers le registre dest.

Instructions de calcul

Instructions unaires

- Addition d'une constante (« add immediate ») :

  addi dest, source, constant

  Produit la somme de la constante (de 16 bits) constant et du contenu du registre source.

- Déplacement (« move ») :

  move dest, source

  Produit le contenu du registre source. Cas particulier de **addi**.

- Négation (« negate ») :

  neg dest, source

  Produit l'opposé du contenu du registre source. Cas particulier de **sub**.

Instructions de calcul

Instructions binaires

- Addition (« add ») :

  add dest, source1, source2

  Produit des contenus des registres source1 et source2.
- On a également **sub**, **mul**, **div**.
- Comparaison (« set on less than ») :

  slt dest, source1, source2

  Produit 1 si le contenu du registre source1 est inférieur à celui du registre source2 ; produit 0 sinon.
- On a également sle, sgt, sge, seq, sne.

Instructions de saut

Principe

On distingue les instructions de saut selon que :

- Leurs destinations possibles sont au nombre de 1 (saut inconditionnel) ou bien 2 (saut conditionnel) ;
- Leur adresse de destination est constante ou bien lue dans un registre ;
- Une adresse de retour est sauvegardée ou non.

Instructions de saut

Saut inconditionnel

- Saut (« jump ») :

  j address

  Saute à l'adresse constante address. Celle-ci est en général donnée sous forme symbolique par une étiquette que l'assembleur traduira en une constante numérique.

Instructions de saut

Saut conditionnel

- Saut conditionnel unaire (« branch on greater than zero ») :

  bgtz source, address

  Si le contenu du registre source est supérieur à zéro, saute à l'adresse constante address.
- On a également **bgez**, **blez**, **bltz**.
- Saut conditionnel binaire (« branch on equal ») :

  beq source1, source2, address

  Si les contenus des registres source1 et source2 sont égaux, saute à l'adresse constante address.
- On a également **blt**, **bne**.

Instructions de saut

Saut avec retour

- Saut avec retour (« jump and link ») :

  jal address

  Sauvegarde l'adresse de l'instruction suivante dans le registre ra, puis saute à l'adresse constante address.

Instructions de saut

Saut vers adresse variable

- Saut vers adresse variable (« jump register ») :

  jr target

  Saute à l'adresse contenue dans le registre target.

- L'instruction **jr** $*ra* est typiquement employée pour rendre la main à l'appelant à la fin d'une fonction ou procédure.

Instructions de saut

Une instruction spéciale

- Appel système (« system call ») :

  syscall

  Provoque un appel au noyau. Par convention, la nature du service souhaité est spécifiée par un code entier stocké dans le registre $v0$. Des arguments supplémentaires peuvent être passés dans les registres $a0$-$a3$. Un résultat peut être renvoyé dans le registre $v0$.

Pseudo-instructions

Certaines des instructions précédentes ne sont en fait pas implantées par le processeur, mais traduites par l'assembleur en séquences d'instructions plus simples.

Par exemple :

blt $t0, $t1, address

est expansée en :

slt $at, $t0, $t1
bne $at, $zero, address

Pour nous, la distinction entre instructions et pseudo-instructions n'aura pas d'importance.

Programme « Hello World »

Code

```
# hello_world.asm

.data
hello: .asciiz "hello world\n"

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

Un autre programme

Code

Demander la saisie d'un entier et afficher s'il est positif ou non.

```
.data
positive: .asciiz "positive\n"
negative: .asciiz "negative\n"

.text
main:    li $v0, 5
         syscall
         li $t0, 0
         blt $v0, $t0, neg
         li $v0, 4
         la $a0, positive
         syscall
         j end
neg:     li $v0, 4
         la $a0, negative
         syscall
end:
```

Même programme avec une routine

Code

```
.data
positive: .asciiz "positive\n"
negative: .asciiz "negative\n"

.text
main:    li $v0, 5
         syscall
         move $a0, $v0
         jal pos
         li $v0, 10
         syscall

pos:     li $t0, 0
         blt $a0, $t0, neg
         li $v0, 4
         la $a0, positive
         syscall
         j end
neg:     li $v0, 4
         la $a0, negative
         syscall
end:     jr $ra
```