Architecture des Ordinateurs

Assembleur et adressage mémoire dans MIPS

2ème Année Licence Informatique

Travaux Dirigés #4

1. Contrôle préalable

Cette partie est conçue comme une vérification pour vous permettre de déterminer si vous comprenez les concepts abordés en cours ou non. Veuillez répondre par « Vrai » ou « Faux » aux questions suivantes et inclure une explication :

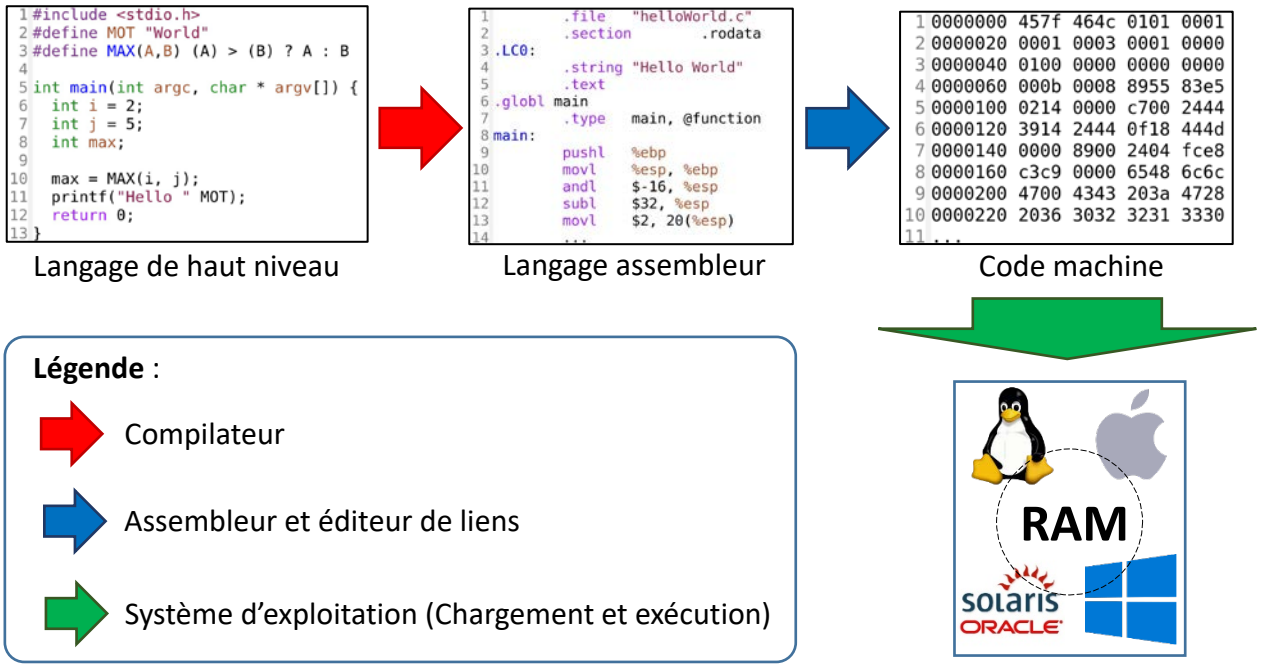
1.1. Le compilateur de langage évolué (C/C++, Fortran, …) peut générer des pseudo-instructions assembleur.
Vrai. C'est l'outil Assembleur qui doit transformer les pseudo-instructions en des instructions assembleur conformes à l'ISA de la machine cible.

1.2. L'objectif principal de l'outil Assembleur est de générer un code machine optimisé.
Faux. Cela est la fonction du Compilateur. L'Assembleur est principalement responsable de la traduction des pseudo-instructions par des instructions assembleur réelles et du calcul des valeurs et déplacements relatifs pour les instructions de sauts et de branchement.

1.3. Les adresses de destination de toutes les instructions de saut est complètement déterminée après l'édition de lien.
Faux. Les sauts relatifs aux registres (c.-à-d. à partir des instructions comme `jr` et `jalr`) ne sont connus qu'au moment de l'exécution du programme.

2. Compilation de programme

Voici un diagramme qui schématise le processus de création et d'exécution d'un programme à partir du code source de langage évolué comme le C.



2.1. Combien de passages dans le code l'outil Assembleur doit effectuer et pourquoi cela ?
Vrai. C'est l'outil Assembleur qui doit transformer les pseudo-instructions en des instructions assembleur conformes à l'ISA de la machine cible.

2.2. Décrivez les six parties principales des fichiers objets générés par l'assembleur (En-tête, texte, données, table de relocalisation, table de symboles, informations de débogage).

- Entête : Tailles et positions des autres parties du fichier objet.
- Texte : Le code exécutable
- Données : La représentation binaire des données statiques présentes dans le fichier source.
- Table de relocalisation : Identifie les lignes de code qui doivent être gérées par l'éditeur de liens (saute vers des étiquettes externes (par exemple, fichiers lib), références à des données statiques).
- Table de Symboles : Liste des étiquettes et des données pouvant être des références dans les fichiers sources.
- Informations de débogage : Informations supplémentaires utilisées par les débogueurs.

3. Assemblage de code source MIPS

Soit un programme C qui contient une seule fonction `sum` qui calcule la somme des éléments dans un tableau. Voici ci-dessous une version compilée de ce programme en MIPS.

```
1 .import print.s       # print.s est un fichier différent
2 .data
3  array:  .word 1,2,3,4,5
4 .text
5  sum:    la $t0, array
6          li $t1, 4
7          move $t2, $0
8  loop:   blt $t1, $0, end
9          sll $t3, $t1, 2
10         add $t3, $t0, $t3
11         lw $t3, 0($t3)
12         add $t2, $t2, $t3
13         addi $t1, $t1, -1
14         j loop
15 end:    move $a0, $t2
16         jal print_int # fonction définie dans print.s
17         ...
```

3.1. Quelles lignes contiennent des pseudo-instructions que l'assembleur doit convertir en des instructions réelles de l'ISA MIPS ?
5, 6, 7, 15. La pseudo instruction « `la` » est convertie en deux instructions « `lui` » et « `ori` » . « `li` » devient ici une instruction « `addiu` », et « `move` » devient « `addu` ».

3.2. Pour les instructions de branchement / saut, quelles étiquettes seront résolues lors de la première passe de l'assembleur ? et quelles autres dans la deuxième passe ?
Le saut vers `loop` sera résolu dans la première passe de l'Assembleur parce que l'étiquette est définie (à la ligne 8) avant qu'elle soit référencée (à la ligne 14). L'étiquette `end` en revanche, sera résolue (à la ligne 8) dans une deuxième passe de l'Assembleur parce qu'elle est définie plus tard dans le code (ligne 15). La première passe aura permis d'identifier où se trouve cette référence tardive. L'étiquette `print_int` (à la ligne 16) fait référence à un symbole dans un autre fichier, et sera dans ce cas résolu par l'éditeur de liens.

3.3. Supposons que le code de ce programme, une fois charger en mémoire, commence à l'adresse `0x00400000` (voir ci-dessous). Pourquoi y a-t-il un saut de 8 entre la première et la deuxième ligne ?

```
0x00400000:  sum:    la $t0, array
0x00400008:          li $t1, 4
0x0040000C:          move $t2, $0
0x00400010:  loop:   blt $t1, $0, end
0x00400014:          sll $t3, $t1, 2
0x00400018:          add $t3, $t0, $t3
0x0040001C:          lw $t3, 0($t3)
0x00400020:          add $t2, $t2, $t3
0x00400024:          addi $t1, $t1, -1
0x00400028:          j loop
0x0040002C: end:     move $a0, $t2
0x00400030:          jal print_int
```

Il y'a un saut de 8 car « `la` » est une pseudo-instruction qui se traduit par deux instructions MIPS régulières (voir réponse de la question 3.1) !

3.4. Donnez la table des symboles une fois que l'outil Assembleur a effectué toutes ses passes

Étiquette	Adresse
`sum`	`0x00400000`
`loop`	`0x00400010`
`end`	`0x0040002C`

L'inclusion des étiquettes `loop` et `end` dans la table des symboles est une réponse parfaitement valide si on utilisait un assembleur seul, car il n'aurait aucun moyen de faire la différence entre les trois étiquettes. Dans la réalité, cependant, on utilise très souvent des outils qui intègrent au même temps un compilateur, un assembleur et un éditeur de lien pour produire des fichiers objets / exécutables (par exemple `gcc`). Ce genre d'outil saura déterminer à partir de la phase de compilation quelles étiquettes sont pour les fonctions et lesquelles ne le sont pas. En tant que tel, il ne mettra que les étiquettes de fonction dans la table des symboles puisque ce sont les seules que d'autres fichiers peuvent référencer.

4. Adressage MIPS

Il existe plusieurs modes d'adressage pour accéder à la mémoire dans MIPS :

- **Registre** : l'adresse est contenue dans le registre. e.g. « `jr $ra` »
- **Indexé** : l'opérande est une adresse de base contenu d'un registre à laquelle est ajouté une valeur de déplacement. e.g. « `lw $a1, 0($s0)` » , « `sb $t1, 3($a0)` » .
- **Indexé sur le PC** : Utilise le PC (en fait le PC actuel plus quatre) et ajoute l'immédiat de l'instruction (multiplié par 4) pour créer une adresse. Ce mode est utilisé par des instructions comme `beq`, `bne`. L'étiquette est remplacée par un immédiat qui indique le nombre d'instructions pour le branchement.
- **Pseudo-direct** : adresse obtenue par concaténation des quatre bits de poids fort du PC et les 26 bits [25:0] de l'instruction avec des bits LSB implicites de 00 (pouvez-vous dire pourquoi deux bits implicites de 00 ?). Ce mode est utilisé pas les instructions de type J.

4.1. Vous devez sauter à une adresse de $2^{28} + 4$ octets de plus que le PC actuel. Comment faire ? Pour cette question, l'adresse de destination exacte est supposée connue au moment de la compilation. (Indication : vous avez besoin de plusieurs instructions)

Les instructions de saut de type J ne peuvent atteindre que des adresses qui possèdent les même quatre bits de poids fort que le PC. Un saut de $2^{28} + 4$ octets nécessiterait de changer le quatrième bit de poids fort. Par conséquent, une instruction de saut de type J n'est pas adaptée. Nous devons charger l'adresse dans un registre et utiliser l'instruction `jr`.

```
lui $at , {les 16 bits de poids fort de l'adresse}
ori $at , $at , {les 16 bits de poids faible de l'adresse}
jr $at
```

4.2. Vous devez maintenant sauter à une instruction $2^{17} + 4$ octets plus élevée que le PC actuel lorsque `$t0` vaut `0`. Vous pouvez supposer pour cette question que le saut ne s'effectue pas à une adresse appartenant un nouveau bloc de 2^{28} octets. Donnez les instructions MIPS ?

La plus grande adresse qu'une instruction de branchement peut atteindre est PC + 4 + SgnExt(Imm). Comme que le champ immédiat pour ce type d'instructions est sur 16 bits et signé. Donc le plus grand déplacement possible qu'on puisse faire est de $2^{15} - 1$ mots (c.-à-d. $2^{17} - 4$ octets). De ce fait, nous ne pouvons pas utiliser une instruction simple de pour se brancher à notre objectif, mais selon l'hypothèse du problème, nous pouvons utiliser une instruction de saut (type-J). En supposant l'instruction est indiquée par l'étiquette `Foo`,

```
                bne $t0 , $0, NePasSauter
                j  Foo
NePasSauter:    ...
```

4.3. Soit les instructions MIPS suivantes (et leurs adresses associées), remplissez les champs vides pour les instructions indiquées (vous aurez besoin de votre fiche mips!)

```
0x002cff00:  loop:  addu $t0, $t0, $t0   | 0  |  8  |  8  |  8  |  0  |0x21|
0x002cff04:         jal foo              | 3  |          0xc0001          |
0x002cff08:         bne $t0, $0, loop    | 5  | 8  | 0  | -3 = 0xfffd  |
      ⋮                        ⋮
0x00300004:  foo:   jr $ra               # $ra =   0x002cff08
```