Programmation 1

TD n°2

Amrita Suresh

22/09/2020

1 Assembler code exercises

Exercise 1 :

Which sample contains

— a `for` loop ?

— only a single `if-else` construct ?

— a `while` loop ?

— an `if-else-else` construct ?

1.
```
func1:
movl $0, %eax
jmp .L2
.L3:
addl $1, %eax
.L2:
movslq %eax, %rdx
cmpb $0, (%rdi,%rdx)
jne .L3
ret
```

2.
```
func2:
cmpl $-1, %edi
je .L6
cmpl $-1, %esi
je .L6
testl %edi, %edi
jle .L5
cmpl %esi, %edi
jle .L5
addl $1, %edi
movl %edi, (%rdx)
ret
.L5:
movl $0, (%rdx)
ret
.L6:
subl $1, %esi
movl %esi, (%rdx)
ret
```

3.
```
func3:
movl (%rdi), %eax
movl $1, %edx
```

```
jmp .L2
.L4:
movslq %edx, %rcx
movl (%rdi,%rcx,4), %ecx
cmpl %ecx, %eax
jle .L3
movl %ecx, %eax
.L3:
addl $1, %edx
.L2:
cmpl %esi, %edx
jl .L4
ret
```

4.
```
func4:
cmpl %esi, %edi
jge .L2
movl %edi, (%rdx)
ret
.L2:
movl %esi, (%rdx)
ret
```

Exercise 2 :

Justify why `pushl` ⟨source⟩ is equivalent to :

```
subl $4, %esp
movl ⟨source⟩, (%esp)
```

Propose an equivalent of `popl` ⟨dest⟩.

Exercise 3 :

The instruction `leal` ("load effective address") resembles `movl`. But where `movl` ⟨source⟩, ⟨dest⟩ copies ⟨source⟩ to ⟨dest⟩, `leal` ⟨source⟩, ⟨dest⟩ copies the *address* where ⟨source⟩ is stored to ⟨dest⟩. So, for example :

— `leal 0x8f90ea48, %eax` (absolute) is equivalent to
`movl $0x8f90ea48, %eax` (immediate),

— `leal 4(%ebx), %eax` puts the content of `%ebx` plus 4 in `%eax`, instead of reading what is stored at the address `%ebx` plus 4.

1. What does `leal (%eax, %esi, 4), %ebx` do ?
2. Why do the immediate and register addressing modes make no sense for the case of `leal` ?
3. Is the instruction `jmp` ⟨source⟩ equivalent to `movl` ⟨source⟩`, %eip` or to `leal` ⟨source⟩`, %eip` ? (Except that these last two instructions do not exist, because you cannot use the `%eip` register as an argument to any instruction. . .)

Exercise 4 :

The only difference between `call` and `jmp` is that `call` first stacks the contents of `%eip`, that is, the address that is immediately after the `call` statement.

1. I wrote the following code to address `0x8aa0e018` : `call 0x7fe976a0`. What could I have written instead that has the same effect, and that doesn't use `call` ?
2. Symmetrically, what can I use instead of `ret` ?
3. Why do we never do it ?

Exercise 5 :

Here is some assembly code.

```
f:
movl a, %eax
movl b, %edx
andl $255, %edx
subl %edx, %eax
movl %eax, a
retq
```

Write valid C code that could have compiled into this assembly (i.e., write a C definition of function `f`), given the global variable declarations `extern unsigned a, b;`.

Exercise 6 :

Indicate for what values of the register contents will the jump be taken.

1. ```
   xorl %eax, %eax
   jge LABEL
   ```
2. ```
   testb $1, %eax
   jne LABEL
   ```
3. ```
   cmpl %edx, %eax
   jl LABEL
   ```

Exercise 7 :

Consider the C program :

```
int x = 5;
int y = 3;
int z;

z = 5*(x+y);
```

1. Convert the program to three value code, where all the instructions are of the form $x = y$ op z or x, y, z are three variables (not necessarily distinct). Feel free to add new variables.
2. Now write the program in MIPS assembly.

Exercise 8 : Compilation à la main.

We consider the C program :

```
int main(){
  int i;
  i = 2*4+3;
  printf("%d\n", i*i);
  return 0;
}
```

1. Convert the program to three value code, where all the instructions are of the form $x = y$ op z or x, y, z are three variables (not necessarily distinct). Feel free to add new variables.
2. Assign a temporary MIPS register to each variable obtained.
3. Now write the program in MIPS assembly.
```

2 Guide de référence rapide de l'assembleur x86 32 bits

Les registres : `%eax %ebx %ecx %edx %esi %edi %ebp %esp`. Ils contiennent tous un entier de 32 bits (4 octets), qui peut aussi être vu comme une adresse. Le registre `%esp` est spécial, et pointe sur le sommet de pile ; il est modifié par les instructions `pushl`, `popl`, `call`, `ret` notamment.

— `addl` ⟨source⟩, ⟨dest⟩ . . . . . ⟨dest⟩= ⟨dest⟩+ ⟨source⟩ (addition)
Ex : `addl $1, %eax` ajoute 1 au registre `%eax`.
Ex : `addl $4, %esp` dépile un élément de 4 octets de la pile.
Ex : `addl %eax, (%ebx, %edi, 4)` ajoute le contenu de `%eax` à la case mémoire à l'adresse `%ebx` $+4\times$`%edi`. (Imaginez que `%ebx` est l'adresse de début d'un tableau `a`, `%edi` est un index `i`, ceci stocke `%eax` dans `a[i]`.)

— `andl` ⟨source⟩, ⟨dest⟩ . . . . . ⟨dest⟩= ⟨dest⟩& ⟨source⟩ (et bit à bit)

— `call` ⟨dest⟩ . . . . . appel de procédure à l'adresse ⟨dest⟩
Équivalent à `pushl $`a, où a est l'adresse juste après l'instruction `call` (l'adresse *de retour*), suivi de `jmp` ⟨dest⟩.
Ex : `call printf` appelle la fonction `printf`.
Ex : `call *%eax` (appel indirect) appelle la fonction dont l'adresse est dans le registre `%eax`. Noter qu'il y a une irrégularité dans la syntaxe, on écrit `call *%eax` et non `call (%eax)`.

— `cltd` . . . . . conversion 32 bits → 64 bits
Convertit le nombre 32 bits dans `%eax` en un nombre sur 64 bits stocké à cheval entre `%edx` et `%eax`.
Note : `%eax` n'est pas modifié ; `%edx` est mis à 0 si `%eax` est positif ou nul, à -1 sinon.
À utiliser notamment avant l'instruction `idivl`.

— `cmpl` ⟨source⟩, ⟨dest⟩ . . . . . comparaison
Compare les valeurs de ⟨source⟩ et ⟨dest⟩. Utile juste avant un saut conditionnel (`je`, `jge`, etc.).
À noter que la comparaison est faite dans le sens inverse de celui qu'on attendrait. Par exemple, `cmp` ⟨source⟩, ⟨dest⟩ suivi d'un `jge` ("jump if greater than or equal to"), va effectuer le saut si ⟨dest⟩≥⟨source⟩ : on compare ⟨dest⟩ à ⟨source⟩, et non le contraire.

— `idivl` ⟨dest⟩ . . . . . division entière et reste
Divise le nombre 64 bits stocké en `%edx` et `%eax` (cf. `cltd`) par le nombre 32 bits ⟨dest⟩. Retourne le quotient en `%eax`, le reste en `%edx`.

— `imull` ⟨source⟩, ⟨dest⟩ . . . . . multiplie ⟨dest⟩ par ⟨source⟩, résultat dans ⟨dest⟩

— `jmp` ⟨dest⟩ . . . . . saut inconditionnel : `%eip`=⟨dest⟩

— `je` ⟨dest⟩ . . . . . saut conditionnel
Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩=⟨source⟩, continue avec le flot normal du programme sinon.

— `jg` ⟨dest⟩ . . . . . saut conditionnel
Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩>⟨source⟩, continue avec le flot normal du programme sinon.

— `jge` ⟨dest⟩ . . . . . saut conditionnel
Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩≥⟨source⟩, continue avec le flot normal du programme sinon.

— `jl` ⟨dest⟩ ..... saut conditionnel
Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩<⟨source⟩, continue avec le flot normal du programme sinon.

— `jle` ⟨dest⟩ ..... saut conditionnel
Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩≤⟨source⟩, continue avec le flot normal du programme sinon.

— `leal` ⟨source⟩, ⟨dest⟩ ..... chargement d'adresse effective
Au lieu de charger le contenu de ⟨source⟩ dans ⟨dest⟩, charge l'*adresse* de ⟨source⟩.
Équivalent C : ⟨dest⟩=`&`⟨source⟩.

— `movl` ⟨source⟩, ⟨dest⟩ ..... transfert
Met le contenu de ⟨source⟩ dans ⟨dest⟩. Équivalent C : ⟨dest⟩=⟨source⟩.
Ex : `movl %esp, %ebp` sauvegarde le pointeur de pile `%esp` dans le registre `%ebp`.
Ex : `movl %eax, 12(%ebp)` stocke le contenu de `%eax` dans les quatre octets commençant à `%ebp`+12.
Ex : `movl (%ebx, %edi, 4), %eax` lit le contenu de la case mémoire à l'adresse `%ebx` +4×`%edi`, et le met dans `%eax`. (Imaginez que `%ebx` est l'adresse de début d'un tableau `a`, `%edi` est un index `i`, ceci stocke `a[i]` dans `%eax`.)

— `negl` ⟨dest⟩ ..... ⟨dest⟩=-⟨dest⟩(opposé)

— `notl` ⟨dest⟩ ..... ⟨dest⟩=`~`⟨dest⟩(non bit à bit)

— `orl` ⟨source⟩, ⟨dest⟩ ..... ⟨dest⟩= ⟨dest⟩`|` ⟨source⟩ (ou bit à bit)

— `popl` ⟨dest⟩ ..... dépilement
Dépile un entier 32 bits de la pile et le stocke en ⟨dest⟩.
Équivalent à `movl (%esp),`⟨dest⟩ suivi de `addl $4, %esp`.
Ex : `popl %ebp` récupère une ancienne valeur de `%ebp` sauvegardée sur la pile, typiquement, par `pushl`.

— `pushl` ⟨source⟩ ..... empilement
Empile l'entier 32 bits ⟨source⟩ au sommet de la pile.
Équivalent à `movl` ⟨source⟩`, -4(%esp)` suivi de `subl $4, %esp`.
Ex : `pushl %ebp` sauvegarde la valeur de `%ebp`, qui sera rechargée plus tard par `popl`.
Ex : `pushl` ⟨source⟩ permet aussi d'empiler les arguments successifs d'une fonction. (Note : pour appeler une fonction C comme `printf` par exemple, il faut empiler les arguments en commençant par celui de droite.)

— `ret` ..... retour de procédure
Dépile une adresse de retour a, et s'y branche. Lorsque la pile est remise dans l'état à l'entrée d'une procédure f, ceci a pour effet de retourner de f et de continuer l'exécution de la procédure appelante.
Équivalent à `popl %eip`... si cette instruction existait (il n'y a pas de mode d'adressage permettant de manipuler `%eip` directement).

— `subl` ⟨source⟩, ⟨dest⟩ ..... ⟨dest⟩= ⟨dest⟩- ⟨source⟩(soustraction)
Ex : `subl $1, %eax` retire 1 du registre `%eax`.
Ex : `subl $4, %esp` alloue de la place pour un nouvel élément de 4 octets dans la pile.

— `xorl` ⟨source⟩, ⟨dest⟩ . . . . . ⟨dest⟩= ⟨dest⟩^ ⟨source⟩ (ou exclusif bit à bit)

La gestion de la pile standard est la suivante. En entrée d'une fonction, on trouve en `(%esp)` l'adresse de retour, en `4(%esp)` le premier paramètre, en `8(%esp)` le deuxième, et ainsi de suite. La fonction doit commencer par exécuter `pushl %ebp` puis `movl %esp, %ebp` pour sauvegarder `%ebp` et récupérer l'adresse de base dans `%ebp`. A ce stade et dans tout le reste de la fonction l'adresse de retour sera en `4(%ebp)`, le premier paramètre en `8(%ebp)`, le second en `12(%ebp)`, etc.

La fonction empile et dépile ensuite à volonté, les variables locales et les paramètres étant repérées par rapport à `%ebp`, qui ne bouge pas.

A la fin de la fonction, la fonction rétablit le pointeur de pile et `%ebp` en exécutant `movl %ebp, %esp` puis `popl %ebp` (et enfin `ret`).

3 Guide de référence rapide de l'assembleur MIPS

On compte 32 registres d'usage général. Par convention, on leur réserve l'usage indiqué :

— `zero` : contient toujours 0, même après une écriture ;

— `v0`, `v1` : servent à retourner des valeurs de fonctions ;

— `a0`, `a1`, `a2`, `a3` : servent à passer les quatre premiers arguments des fonctions (les autres sont empilés) ;

— `t0`, `t1`, . . . , `t9` : registres temporaires, non sauvegardés lors des appels de fonctions (donc à sauvegarder par la fonction appelante, si elle compte dessus au retour de la fonction appelée) ;

— `s0`, `s1`, . . . , `s7` : registres temporaires, que les fonctions appelées doivent sauvegarder si elles les utilisent ;

— `gp` : Global Pointer, pointe vers le début de la zone des variables globales ;

— `sp` : Stack Pointer, pointe vers le sommet de pile (comme `%esp` sur x86) ;

— `fp` : Frame Pointer (comme `%ebp` sur x86) ;

— `ra` : Return Address, contient l'adresse de retour à la fonction appelante (un `jmp $ra` est donc ce qui remplace `ret` sur un MIPS ; mais ce registre doit être sauvegardé par la fonction appelante !)

— `at`, `k0`, `k1` : réservés ;

Les seules instructions qui interagissent avec la mémoire sont `lw` (load word) et `sw` (store word). Les seules instructions permettant de charger des constantes sont `li` et `la`. Les instructions de test et de saut peuvent aussi manipuler des adresses constantes (les adresses où sauter), mais toutes les autres ne manipulent que des registres.

Toutes les instructions ne sont pas listées, et leurs mnémoniques sont souvent simplifiés par rapport aux assembleurs existants.

— `add` ⟨dest⟩, ⟨$\arg_1$⟩, ⟨$\arg_2$⟩ . . . . . ⟨dest⟩= ⟨$\arg_1$⟩+ ⟨$\arg_2$⟩ (addition)

— `and` ⟨dest⟩, ⟨$\arg_1$⟩, ⟨$\arg_2$⟩ . . . . . ⟨dest⟩= ⟨$\arg_1$⟩& ⟨$\arg_2$⟩ (et bit-à-bit)

— `beq` ⟨$\arg_1$⟩, ⟨$\arg_2$⟩, a
Branche à l'adresse a si ⟨$\arg_1$⟩=⟨$\arg_2$⟩.

— `bge` ⟨$\arg_1$⟩, ⟨$\arg_2$⟩, a
Branche à l'adresse a si ⟨$\arg_1$⟩≥⟨$\arg_2$⟩.

— `bgez` ⟨source⟩, a
Branche á l'adresse a si ⟨source⟩≥ 0.

— `ble` ⟨arg_1⟩, ⟨arg_2⟩, a
Branche à l'adresse a si ⟨arg_1⟩$\leq$⟨arg_2⟩.

— `blez` ⟨source⟩, a
Branche à l'adresse a si ⟨source⟩≤ 0.

— `bne` ⟨arg_1⟩, ⟨arg_2⟩, a
Branche à l'adresse a si ⟨arg_1⟩$\neq$⟨arg_2⟩.

— `div` ⟨dest⟩, ⟨arg_1⟩, ⟨arg_2⟩ ..... ⟨dest⟩= ⟨arg_1⟩/ ⟨arg_2⟩ (quotient)

— `j` a
Saute à l'adresse a. (Note : a peut être un registre ou une constante d'adresse.)

— `jal` a
Saute à l'adresse a après avoir sauvegardé l'adresse de l'instruction qui suit dans `ra`.

— `la` ⟨dest⟩, a ..... ⟨dest⟩= a (chargement de constante adresse)

— `li` ⟨dest⟩, n ..... ⟨dest⟩= n (chargement de constante entière)

— `lw` ⟨dest⟩, c(⟨source⟩)
Charge le contenu de l'adresse $s + c$ dans le registre ⟨dest⟩, où s est le contenu du registre ⟨source⟩ et c est une constante.

— `move` ⟨dest⟩, ⟨source⟩ ..... ⟨dest⟩= ⟨source⟩

— `mul` ⟨dest⟩, ⟨arg_1⟩, ⟨arg_2⟩ ..... ⟨dest⟩= ⟨arg_1⟩$\times$ ⟨arg_2⟩ (multiplication)

— `nop`
Ne fait rien.

— `or` ⟨dest⟩, ⟨arg_1⟩, ⟨arg_2⟩ ..... ⟨dest⟩= ⟨arg_1⟩`|` ⟨arg_2⟩ (ou bit-à-bit)

— `sll` ⟨dest⟩, ⟨arg_1⟩, ⟨arg_2⟩ ..... ⟨dest⟩= ⟨arg_1⟩`<<` ⟨arg_2⟩ (décalage à gauche)

— `slr` ⟨dest⟩, ⟨arg_1⟩, ⟨arg_2⟩ ..... ⟨dest⟩= ⟨arg_1⟩`>>` ⟨arg_2⟩ (décalage à droite)

— `sub` ⟨dest⟩, ⟨arg_1⟩, ⟨arg_2⟩ ..... ⟨dest⟩= ⟨arg_1⟩`-` ⟨arg_2⟩ (addition)

— `sw` ⟨source⟩, c(⟨dest⟩)
Écrit le contenu du registre ⟨source⟩ à l'adresse $d + c$, où d est le contenu du registre ⟨dest⟩ et c est une constante.

— `xor` ⟨dest⟩, ⟨arg_1⟩, ⟨arg_2⟩ ..... ⟨dest⟩= ⟨arg_1⟩`^` ⟨arg_2⟩ (ou exclusif bit-à-bit)