Programmation 1 – TD 2

Certaines des questions sont dues à Juliusz Chroboczek, que je remercie.

1 Assembleur, assembleur, assembleur...

1. Que calculent les codes assembleurs suivants ? Les deux premiers sont en assembleur x86 (32 bits) : voir la section 2 pour un guide rapide de ce langage. Le troisième est en assembleur MIPS : voir la section 3.

```
.start:
  cmpl %eax, 0
  je .end
  pushl %eax
  subl $1, %eax
  call .start
  popl %ebx
  imull %ebx, %eax
  ret
.end
  movl $1, %eax
  ret
```

```
  pushl %edx
  xorl %ebx, %ebx
.loop:
  cmpl %eax, 0
  je .end
  movl %eax, %edx
  imull %edx, %edx
  addl %edx, %ebx
  subl $1, %eax
  jmp .loop
.end
  movl %ebx, %eax
  popl %edx
  ret
```

```
.f:
  blez $a0, .f_0
  sub  $sp, $sp, 8
  sw   $ra, 0($sp)
  sw   $a0, 4($sp)
  sub  $a0, $a0, 1
  jal  .f
  lw   $a0, 4($sp)
  mul  $v0, $v0, $a0
  lw   $ra, 0($sp)
  add  $sp, $sp, 8
  j    $ra
.f_0:
  li   $v0, 1
  j    $ra
```

   Quel est le but de `pushl %edx` dans le code du milieu ?

2. Supposons que `%eax` contient l'entier 4 et `%ebx` contient l'entier -3 (en signé) $= 4\ 294\ 967\ 293$ (en non signé). Noter que le branchement `jge` sera pris après une instruction `cmpl %ebx, %eax`. Qu'en est-il dans les cas de : (i) `jg`, (ii) `jl`, (iii) `jle`, (iv) `ja`, (v) `jna`, (vi) `jb`, (vii) `jnb` ?

3. Justifier pourquoi `pushl` ⟨source⟩ est équivalente à :

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

   Proposer un équivalent de `popl` ⟨dest⟩.

4. L'instruction `leal` ("load effective address") ressemble à `movl`. Mais, là où `movl` ⟨source⟩, ⟨dest⟩ recopie ⟨source⟩ dans ⟨dest⟩, `leal` ⟨source⟩, ⟨dest⟩ recopie l'*adresse* où est stocké ⟨source⟩ dans ⟨dest⟩. Donc, par exemple :
   — `leal 0x8f90ea48, %eax` (adressage absolu) est équivalent à `movl $0x8f90ea48, %eax` (adressage immédiat),
   — `leal 4(%ebx), %eax` met le contenu de `%ebx` plus 4 dans `%eax`, au lieu de lire ce qui est stocké à l'adresse qui vaut le contenu de `%ebx` plus 4.

   (a) Que fait `leal (%eax, %esi, 4), %ebx` ?

   (b) Pourquoi les modes d'adressage immédiat et registre n'ont-ils aucun sens pour la source de `leal` ?

   (c) L'instruction `jmp` ⟨source⟩ est-elle équivalente à `movl` ⟨source⟩`, %eip` ou bien à `leal` ⟨source⟩`, %eip` ? (À ceci près que ces deux dernières instructions n'existent pas, car il est impossible d'utiliser le registre `%eip` comme argument d'aucune instruction…)

5. La seule différence entre `call` et `jmp`, c'est que `call` empile d'abord le contenu de `%eip`, c'est-à-dire l'adresse qui est juste après l'instruction `call`.

   (a) J'ai écrit le code suivant à l'adresse `0x8aa0e018` : `call 0x7fe976a0`. Qu'aurais-je pu écrire à la place qui ait le même effet, et qui n'utilise pas `call` ?

   (b) Symétriquement, que puis-je utiliser à la place de `ret` ?

   (c) Pourquoi ne le fait-on jamais ?

6. (Compilation à la main.) On considère le programme C suivant :

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

   (a) Convertir de programme en code à trois valeurs, c'est-à-dire où toute instruction est soit de la forme $x = y$ op z où x, y, z sont trois variables (pas nécessairement distinctes), soit sous la forme d'une fonction appliquée à des variables. (N'hésitez pas à introduire de nouvelles variables.)

   (b) Affecter un registre temporaire MIPS à chacune des variables du programme ainsi obtenu.

   (c) Maintenant, traduire votre programme en assembleur MIPS.

   (d) Même exercice, mais pour l'assembleur x86 : on commencera par convertir le programme en code à *deux* valeurs, c'est-à-dire où toute instruction est soit de la forme $x = x$ op y où x, y sont deux variables (pas nécessairement distinctes), soit sous la forme d'une fonction appliquée à des variables.

7. On considère le programme C, un peu plus compliqué, suivant :

```
int main()
{
  int i, j;
  for (i=0; i<10; i++)
    {
      j = 2*i+5;
      printf ("%d\n", j*j);   /* (1) */
    }
  return j;
}
```

On appelle *branchement conditionnel* une conditionnelle de la forme `if (`e`) goto` lab, c'est-à-dire sans clause `else` et donc la branche à exécuter si la condition e est vraie est un saut.

(a) Éliminer les boucles de ce programme en convertissant les `for` en `while`, puis les `while` en branchements.

(b) Convertir le programme ainsi obtenu en code à trois valeurs, et traduire en assembleur MIPS.

(c) Convertir le programme en code à deux valeurs plutôt, et traduire en assembleur x86.

(d) Même exercice après avoir remplacé la ligne `(1)` par :

```
if (i % 2 != 0) printf ("%d\n", j*j);
```

8. Le processeur MIPS (multiple interlocked pipeline stages) a certaines particularités déroutantes, notamment celle du « delay slot »... que nous avons ignorées jusqu'ici. Voici ce qui se passe : à chaque cycle d'horloge, le processeur décode une instruction et en même temps exécute l'instruction décodée au cycle précédent. L'instruction décodée sera exécutée au cycle suivant. Ceci permet une exécution, dite en *pipeline*, qui rend le processeur très efficace.

(a) En tenant compte de ceci, que fait le code suivant sur MIPS ?

```
    lw v0, 1
    j .plus_loin
    add v0, v0, 3
.plus_loin:
    move v1, v0
```

(b) Tous les codes MIPS que nous avons lus ou écrits depuis le début sont donc faux... les corriger.

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` $+ 4 *$ `%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 $\to$ 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⟩$\geq$⟨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⟩$\geq$⟨arg_2⟩.

— `bgez` ⟨source⟩, a
Branche à 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)