TD Mips (fonctions et mémoire)

Exercice 1 (Fonctions) Traduire en Mips les fonctions suivantes, en respectant cette convention d'appel : les deux premiers arguments sont passés respectivement via les registres `$a0` et `$a1` et le résultat renvoyé via le registre `$v0`. *Indication : il vous faudra identifier les registres qui ont besoin ou non d'être sauvegardés. Le cas échéant, n'hésitez pas à adapter la stratégie générale du cours à ces cas particuliers. Autre indication : cela vaut le coup de tester votre code dans le simulateur MARS !*

1. Comptage de bits.

```
let rec compte n =
  if n = 0
  then 0
  else n mod 2 + compte (n/2)
```

2. Variante récursive terminale.

```
let rec aux n acc =
  if n = 0
  then acc
  else compte (n/2) (acc + n mod 2)

let compte n = aux n 0
```

3. Fonction 91 de McCarthy.

```
let rec f91 n =
  if n > 100
  then n - 10
  else f91(f91(n+11))
```

Correction :

1. *Les registres* `$fp` *et* `$ra` *sont sauvegardés sur la pile au début de l'appel et restaurés à la fin. Dans la branche* **else**, *la valeur* `n mod 2` *calculée et placée dans le registre* `$t0` *avant l'appel récursif est sauvegardée sur la pile. Elle est restaurée après l'appel récursif pour être ajoutée à son résultat.*

```
    .text
main:
    li     $a0, -81
    jal    compte
    move   $a0, $v0
    li     $v0, 1
    syscall
    li     $v0, 10
    syscall

compte:
    subi   $sp, $sp, 8    # Sauvegarde fp et ra
    sw     $fp, 8($sp)
    sw     $ra, 4($sp)
    addi   $fp, $sp, 8    # Init nouveau fp
    bnez   $a0, compte_rec # Vers branche else
    li     $v0, 0         # Cas de base : 0
    b      compte_fin
compte_rec: # Cas récursif (branche else)
    rem    $t0, $a0, 2    # Calcul n mod 2 dans t0
    sw     $t0, 0($sp)    # puis sauvegarde sur la pile
    subi   $sp, $sp, 4
    sra    $a0, $a0, 1    # Appel récursif sur n/2
    jal    compte
    addi   $sp, $sp, 4    # Restauration de t0
    lw     $t0, 0($sp)
    add    $v0, $t0, $v0  # Rés. : n mod 2 + résultat appel réc.
compte_fin: # Nettoyage du tableau d'activation
    lw     $ra, 4($sp)    # Restauration ra et fp
    lw     $fp, 8($sp)
    addi   $sp, $sp, 8
    jr     $ra            # Fin
```

Note : `$fp` *ne sert jamais ici. On pourrait donc économiser sa sauvegarde sur la pile, sa mise à jour et sa restauration pour obtenir le code suivant.*

```
compte:
    subi   $sp, $sp, 4    # Sauvegarde ra
    sw     $ra, 4($sp)
    bnez   $a0, compte_rec # Vers branche else
    li     $v0, 0         # Cas de base : 0
    b      compte_fin
compte_rec: # Cas récursif (branche else)
    rem    $t0, $a0, 2    # Calcul n mod 2 dans t0
    sw     $t0, 0($sp)    # puis sauvegarde sur la pile
    subi   $sp, $sp, 4
    sra    $a0, $a0, 1    # Appel récursif sur n/2
    jal    compte
    addi   $sp, $sp, 4    # Restauration de t0
```

```
    lw     $t0, 0($sp)
    add    $v0, $t0, $v0 # Rés. : n mod 2 + résultat appel réc.
compte_fin: # Nettoyage du tableau d'activation
    lw     $ra, 4($sp)   # Restauration ra et fp
    addi   $sp, $sp, 4
    jr     $ra           # Fin
```

2. *La fonction auxiliaire est récursive terminale, et est utilisée de manière terminale par la fonction principale. On n'a jamais besoin ici de modifier (ni donc de sauvegarder)* `$ra`*. En outre, rien dans le contenu potentiel du tableau d'activation n'est utilisé, on peut se passer de sa création. D'où une traduction possible particulièrement économe.*

```
compte_tr:
    li     $a1, 0        # Initialisation de acc
    j      aux           # Appel terminal
aux:
    bnez   $a0, aux_rec  # Si non nul, branche else
    move   $v0, $a1      # Cas de base : rés = acc
    jr     $ra           # Fin
aux_rec:
    rem    $t0, $a0, 2
    add    $a1, $a1, $t0 # acc + n mod 2
    sra    $a0, $a0, 1   # n-1
    j      aux           # Appel terminal
```

3. *Pour enchaîner les deux appels récursifs, on commence par modifier* `$a0` *pour le faire passer de* `n` *à* `n+11`*, on déclenche le premier appel, puis on transfère le résultat obtenu (dans* `$v0`*) vers* `$a0` *pour le passer comme paramètre au deuxième appel.*

```
f91:
    subi   $sp, $sp, 8   # Sauvegarde fp et ra
    sw     $fp, 8($sp)
    sw     $ra, 4($sp)
    addi   $fp, $sp, 8   # Init nouveau fp
    ble    $a0, 100, f91_rec # Branche else
    subi   $v0, $a0, 10  # Cas de base : n-10
    b      f91_fin
f91_rec:
    addi   $a0, $a0, 11
       jal f91           # Premier appel
    move   $a0, $v0
    jal    f91           # Deuxième appel
f91_fin:
    lw     $ra, 4($sp)   # Restauration ra et fp
    lw     $fp, 8($sp)
    addi   $sp, $sp, 8
    jr     $ra           # Fin
```

Améliorations possibles : comme pour la première version de compte, le registre `$fp` *n'est jamais utilisé et pourrait donc être ignoré. En outre, l'un des deux appels récursif est terminal, on peut songer à l'optimiser. Pour cela cependant, il faut écrire deux versions Mips de* `f91` *: une à utiliser lors des appels terminaux et une à utiliser lors des appels non terminaux.*

□

Exercice 2 (Valeurs optionnelles) On s'intéresse à des expressions admettant une forme de valeur optionnelle similaire au type option de Caml. Ainsi, Some e désigne une valeur optionnelle calculée par e, et None désigne une valeur optionnelle absente. Pour accéder à la valeur d'une option e, on se donne une construction ?e : d, qui évalue l'expression e et teste la valeur obtenue :

- si e s'évalue en Some v alors le résultat est v,
- si e s'évalue en None alors le résultat est la valeur de l'expression d (qu'on appelle le résultat par défaut).

Pour représenter de telles valeurs optionnelles en mémoire, on propose de procéder ainsi :

- La valeur `None` est représentée comme l'entier `0`.
- La valeur `Some v` est représentée par un pointeur vers un bloc alloué dans le tas, formé d'une part d'un entête contenant le nombre d'éléments dans le bloc (en l'occurrence, 1), et d'autre part d'un champ par élément du bloc (en l'occurrence, un unique champ pour l'unique élément v).

Note : la valeur `0` n'est jamais reconnue comme un pointeur valide.

Questions

1. Décrire l'état des registres et du tas après l'exécution du code suivant, et préciser la valeur contenue dans le registre `$v0`.

```
li  $a0, 8
li  $v0, 9
```

```
    syscall
    li   $t0, 1
    sw   $t0, 0($v0)
    li   $t0, 2
    sw   $t0, 4($v0)
    move $t0, $v0
    li   $v0, 9
    syscall
    sw   $t0, 4($v0)
    li   $t0, 1
    sw   $t0, 0($v0)
```

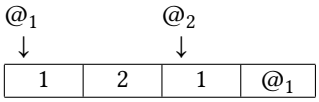
2. Supposons que le registre `$a0` contienne la valeur `Some (Some (Some 3))`. Donner une configuration possible du tas, en précisant les adresses des blocs représentés et leur contenu, sachant que la première adresse du tas est `0x10040000`.
3. Supposons que le registre `$a0` contienne une valeur de la forme `Some (Some` n). Écrire un fragment de code MIPS qui place la valeur n dans le registre `$v0`.
4. Donner le code MIPS pour une fonction f qui prend en entrée une valeur de type `int option` et est telle que :

$$\begin{aligned} f(\mathsf{None}) &= -1 \\ f(\mathsf{Some}\ n) &= n+1 \end{aligned}$$

   Le paramètre est passé par le registre `$a0` et le résultat est passé par le registre `$v0`.
5. Supposons que les registres `$a0` et `$a1` contiennent chacun une valeur de type `int option`. Écrire un fragment de code MIPS qui écrit `1` dans le registre `$v0` si les deux valeurs sont égales, et `0` sinon.

Correction :

1. *Après exécution de ce code :*
   - *le registre* `$a0` *contient l'entier 8,*
   - *le registre* `$t0` *contient l'entier 1,*
   - *le registre* `$v0` *contient une adresse* $@_2$ *(celle allouée par le deuxième appel système* `sbrk`*)*
   - *le tas contient deux blocs consécutifs de 8 octets*



   La valeur correspondant à `$v0` *est Some (Some 2)*
2. *On place dans le tas trois blocs de deux mots (ici, ils sont consécutifs). Les deux premiers ont chacun comme contenu un pointeur vers le bloc suivant, et le troisième a la constant* `3`.

0x10040000	0x10040004	0x10040008	0x1004000c	0x10040010	0x10040014
1	0x10040008	1	0x10040010	1	3

3. *Le registre* `$a0` *contient un pointeur vers un bloc* B_1*, dont le deuxième champ contient un pointeur vers un bloc* B_2*, donc le deuxième champ contient la valeur* n *cherchée. On accède d'abord au deuxième champ de* B_1*, c'est-à-dire à l'adresse de* B_2*, et on la stocke temporairement dans* `$v0`*. Puis on va lire le deuxième champs de* B_2*.*

```
    lw   $v0, 4($a0)
    lw   $v0, 4($v0)
```

4. *Si le paramètre vaut* `0`*, c'est-à-dire représente None, on saute à l'étiquette* `L1` *et on charge la valeur* `-1` *dans le registre de résultat. Sinon, on va lire le contenu du premier champ (qu'on place directement dans le registre de résultat) et on l'incrémente de* `1`*. Dans tous les cas, on aboutit ensuite à* `L2` *et on rend la main à l'appelant.*

```
    beqz $a0, L1
    lw   $v0, 4($a0)
    addi $v0, $v0, +1
    b    L2
L1:
    li   $v0, -1
L2:
    jr   $ra
```

   Note : cette fonction ne fait pas appel à d'autres fonctions, et ne nécessite donc pas de sauvegarder `$ra`*. Elle n'utilise pas non plus de tableau d'activation : pas besoin de redéfinir (et donc pas besoin de sauvegarder) le pointeur* `$fp`*.*
5. *On teste la forme (None ou Some) de chacun des deux arguments, ce qui fait quatre possibilités au total :*
   - *Cas None/None : on charge le résultat* `1`
   - *Cas None/Some et Some/None : on charge le résultat* `0` *(cas regroupés derrière l'étiquette* `incompatible`*).*

— *Cas Some/Some : on charge le contenu de chacun des deux blocs, et on prend comme résultat la comparaison de ces contenus.*

```
    bnez  $a0, some

    # a0 = None
    bnez  $a1, incompatible

    # a0 = None ; a1 = None
    li    $v0, 1
    b     Fin

some:
    beqz  $a1, incompatible

    # a0 = Some x ; a1 = Some y
    lw    $t0, 4($a0)
    lw    $t1, 4($a1)
    seq   $v0, $t0, $t1
    b     Fin

incompatible:
    # un None et un Some
    li    $v0, 0

Fin:
```

□

Exercice 3 (Exceptions) On s'intéresse à un petit langage impératif avec des mécanismes pour déclencher et rattraper des exceptions, similaires à raise et try/with en Caml ou à throw et try/catch en Java. Outre quelques formes d'instructions standards, ce langage propose donc des exceptions, chacune identifiée par un nombre entier positif. La construction

try { i }
catch k { i' }

permet d'exécuter l'instruction i tout en permettant de rattraper une éventuelle exception déclenchée pendant cette exécution.

- Si l'exécution de l'instruction i se termine sans déclencher d'exception, il ne se passe rien d'autre et on passe à l'instruction suivante.
- Si l'exécution de l'instruction i produit l'exception numéro n, alors :
  - si $n = k$ alors on exécute i',
  - sinon on propage l'exception n.

L'instruction throw(n) déclenche l'exception numéro n : les instructions suivantes sont ignorées et l'exception n est propagée au bloc try/catch englobant. Si une exception est déclenchée ou se propage en dehors de tout bloc try/catch alors l'exécution du programme est interrompue.

On s'intéresse à la compilation des mécanismes d'exceptions en suivant la stratégie dite de *stack cutting*, dans laquelle on stocke en mémoire un ensemble de gestionnaires d'exception correspondant chacun à un try/catch en cours d'exécution. Les différents gestionnaires sont chaînés entre eux, du plus récent vers le plus ancien. On maintient également un pointeur vers le gestionnaire courant (c'est-à-dire le plus récent encore actif).

Un gestionnaire d'exceptions est un bloc contenant, au minimum :

- l'adresse du gestionnaire précédent,
- le numéro de l'exception rattrapée par ce gestionnaire,
- l'adresse du code de rattrapage,
- les sauvegardes des valeurs des registres $fp et $sp au moment de la création du gestionnaire d'exceptions.

En l'absence de gestionnaire précédent, on indique par convention l'adresse 0. Les gestionnaires d'exceptions sont manipulés comme suit.

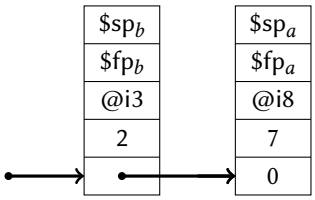
- À l'entrée dans un bloc try/catch on crée un nouveau gestionnaire g' contenant les informations de rattrapage de ce bloc et les sauvegardes de $fp et $sp. Le gestionnaire g' pointe sur l'ancien gestionnaire courant g et on met à jour le pointeur vers le gestionnaire courant.
- À la sortie d'un bloc try/catch on défausse le gestionnaire courant g'. Cela se fait par une mise à jour du pointeur vers le gestionnaire courant, qui va maintenant pointer vers le gestionnaire qui précédait g.
- Lorsqu'une exception est déclenchée on compare son numéro avec le numéro de l'exception rattrapée par le gestionnaire courant.
  - si les numéros sont égaux on restaure les valeurs sauvegardées dans le gestionnaire courant pour $fp et $sp, puis on exécute le code pointé par ce gestionnaire,
  - sinon on propage l'exception au gestionnaire précédent.

  Dans ces deux cas, on défausse de plus le gestionnaire courant.

Considérons le programme suivant.

```
try { try { i1 }
      catch 2 { i3 }
      try { i4 }
      catch 5 { i6 } }
catch 7 { i8 }
```

Voici une représentation des gestionnaires d'exceptions lors de l'exécution de l'instruction i1. On y note respectivement @i3 et @i8 les adresses des blocs de code i3 et i8, et $\$sp_a$, $\$sp_b$, $\$fp_a$ et $\$fp_b$ les valeurs sauvegardées de $sp et $fp à différents moments.

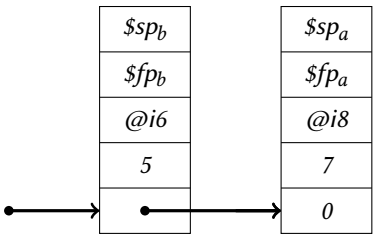


Questions

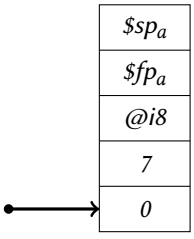
1. Parmi les candidats suivants, dire lesquels sont adaptés ou non pour stocker les gestionnaires d'exceptions, et lesquels sont adaptés ou non pour stocker le pointeur vers le gestionnaire courant. Justifier, et éventuellement préciser les particularités de certains choix possibles.
   (a) les données statiques (b) la pile (c) le tas (d) les registres
2. Dessiner des représentations des gestionnaires d'exceptions aux moments suivants :
   (a) Pendant l'exécution de i4, (b) Pendant l'exécution de i6, (c) Pendant l'exécution de i8.
3. Écrire du code MIPS réalisant les actions suivantes. Précisez quelle stratégie vous choisissez parmi celles citées à la question 1.
   (a) Défausser le gestionnaire d'exception courant.
   (b) Exécuter le code de rattrapage du gestionnaire d'exception courant (sans tester le numéro de l'exception).
   (c) Créer un nouveau gestionnaire d'exception rattrapant l'exception numéro 3 avec un code de rattrapage désigné par l'étiquette rat.
   (d) En supposant que le registre $t0 contient le numéro d'une exception, restaurer $fp et $sp et exécuter le code de rattrapage du gestionnaire d'exception courant si celui-ci porte le bon numéro (sans propager aux gestionnaires suivants si ce n'est pas le cas).
   (e) Le mécanisme complet de throw 1 (avec propagation aux gestionnaires d'exceptions aussi loin que nécessaire).
4. Décrire une stratégie permettant d'assurer qu'une exception non rattrapée interrompt proprement le programme, avec un appel système exit. *On ne demande pas de code MIPS.*

Correction :

1. *Les gestionnaires sont créés dynamiquement, et leur nombre n'est pas connu à l'avance. Ils ne peuvent donc être placés que dans la pile ou le tas. Le pointeur vers le gestionnaire courant est en revanche unique. On peut lui dédier un registre ou un mot dans les données statiques.*
2. *Pendans l'execution de i4 :*



Pendant l'exécution de i6 :



Pendant l'exécution de i8 :

0

3. *On place les gestionnaires dans le tas, et l'adresse du gestionnaire courant dans le registre* `$s7`.

(a) *Défausse d'un gestionnaire. Le registre* `$s7` *nous donne l'adresse du gestionnaire courant, dont la première case contient l'adresse du précédent. Il suffit de remplacer* `$s7` *par la valeur pointée par* `$s7`.

```
lw $s7, 0($s7)
```

(b) *Ne pas oublier de d'abord restaurer les registres. On récupère ensuite l'adresse du code de rattrapage dans le registre* `$t0`*, on défausse le gestionnaire à l'aide du code de la question précédente, puis on saute à l'adresse de rattrapage avec* `jr`.

```
lw $fp, 12($s7) # $fp <- $fp sauvegardé
lw $sp, 16($s7) # $sp <- $sp sauvegardé
lw $t0, 8($s7) # $t0 <- adresse du code de rattrapage
lw $s7, 0($s7) # défausse du gestionnaire courant
jr $t0
```

(c) *Il faut d'abord allouer de l'espace pour le gestionnaire, par exemple ici avec sbrk pour simplifier. On suppose que le registre* `$t0` *est disponible. On initialise l'ensemble du bloc avec les bonnes valeurs et on met* `$s7` *à jour.*

```
li $a0, 20
li $v0, 9
syscall        # $v0 <- adresse du nouveau gestionnaire
sw $s7, 0($v0) # initialisation pointeur précédent
li $t0, 3
sw $t0, 4($v0) # initialisation numéro d'erreur
la $t0, rat
sw $t0, 8($v0) # initialisation code de rattrapage
sw $fp, 12($v0)
sw $sp, 16($v0) # sauvegarde $fp et $sp
move $s7, $v0  # mise à jour gestionnaire courant
```

(d) *Si le numéro n'est pas le bon, on saute à la fin de ce bloc de code sans rien faire. On suppose que le registre* `$t1` *est disponible.*

```
lw $t1, 4($s7)
bne $t0, $t1, next
# inclure ici le code de la question (b)
next:
# la suite (rien n'apparaît ici pour l'instant)
```

(e) *On combine les éléments précédents, après avoir chargé dans le registre* `$t0` *le numéro de l'exception lancée.*

```
   # numéro de l'exception
0  li $t0, 1
   # Boucle de recherche du bon gestionnaire d'exception
  search:
   # test du numéro, question (d)
1  lw $t1, 4($s7)
2  bne $t0, $t1, next
   # ici, on a trouvé le bon gestionnaire : restauration de
   # $fp et $sp et saut au code de rattrapage, question (b)
3  lw $fp, 12($s7)
4  lw $sp, 16($s7)
5  lw $t0, 8($s7)
6  lw $s7, 0($s7)
7  jr $t0
  next:
   # défausse du gestionnaire courant, question (a)
8  lw $s7, 0($s7)
9  b search
```

Note : si l'exception n'est pas rattrapée, le pointeur `$s7` *vers le gestionnaire courant finira par prendre la valeur* `0`*. L'exécution du programme va alors s'interrompre brutalement à la ligne* `8`*, car on ne peut pas lire à l'adresse* `0`.

4. *Plutôt que de démarrer avec un pointeur* `$s7` *valant* `0`*, on peut créer au début de l'exécution du programme un premier gestionnaire d'exceptions dans lequel :*
   - *le code de rattrapage gère l'arrêt propre du programme, et*
   - *on a à la place du numéro d'exception une valeur spéciale rattrapant n'importe quelle exception.*

   Il faut alors mettre à jour la boucle de recherche du bon gestionnaire d'exception pour tester la présence de cette valeur spéciale.

□