TD3

Entrées/sorties ; interruptions

1- Entreés/Sorties

Un processeur a besoin de dispositifs d'entrées/sorties pour communiquer avec l'extérieur.

« craps » et son simulateur disposent d'entrées/sorties minimalistes :

- Entrées : 16 switches (interrupteurs permettant d'entrer 0 ou 1) : La lecture des 16 switches se fait via l'adresse 0x90000000 avec l'instruction ld.
- Sorties : 16 leds permettant d'afficher 16 bits en parallèle : L'écriture sur les leds se fait via l'adresse 0xB0000000 avec l'instruction st

Le code suivant permet de lire les switches en continu et d'afficher la valeur lue sur les leds :

```
SWITCHES = 0x90000000
LEDS = 0xB0000000
            set SWITCHES, %r19
            set LEDS, %r20
boucle:     ld [%r19], %r1
            st %r1, [%r20]
            ba boucle
```

Ecrire le sous-programme afficher_leds_7_0 qui affiche une valeur (8 bits), passée en paramètre dans %r1, sur les 8 leds de faible poids et met à 0 les 8 leds de fort poids.
Ecrire le sous-programme afficher_leds_15_8 qui fait l'affichage sur les 8 bits de fort poids.

2- Interruptions

Une interruption est un front d'un signal qui arrive sur une ligne distincte (qui ne fait pas partie des lignes d'entrées/sorties), et qui nécessite un traitement prioritaire. Un processeur dispose généralement de plusieurs lignes d'interruption, chacune étant associée à l'occurrence d'un type d'événement particulier : événement d'un périphérique d'entrée (clavier, souris, etc.), disque dur, réseau, horloge de séquencement.

Une interruption est dite asynchrone, car elle peut intervenir à tout moment de l'exécution d'un programme. Lors de son occurrence, un mécanisme spécifique déclenche l'appel à un sous-programme d'interruption sans que cela n'affecte l'exécution du programme interrompu.

CRAPS dispose d'une seule ligne d'interruption appelée IT, dont le mécanisme de prise en compte est le suivant :

- Un front montant sur IT déclenche la mémorisation de l'interruption dans une bascule « pendingIT »
- L'instruction en cours d'exécution est terminée, jusqu'au retour du séquenceur à l'état FETCH

- Lorsque le séquenceur est dans l'état FETCH et que pendingIT= 1, le séquenceur ne va pas dans l'état DECODE ; il suit une autre séquence d'états durant laquelle :
  - opendingIT est remis à 0
  - la valeur courante de PC est empilée dans la pile
  - les flags (N, Z, V, et C) sont empilés
  - PC <- 1
  - Retour à l'état FETCH

Ainsi, l'occurrence d'une interruption provoque un branchement à l'adresse 1. À cette adresse, un sous-programme (handler) d'interruption doit être implanté, terminé par l'instruction reti.
L'exécution de reti a l'effet suivant :

- Le sommet de la pile est dépilé dans les flags
- Le sommet de la pile est dépilé dans PC

L'exécution de reti provoque donc le retour dans le programme interrompu, juste après l'instruction durant laquelle l'interruption a eu lieu.

Le code suivant, éclaté exprès sur deux colonnes, compte le nombre d'interruptions IT. A chaque appui sur le bouton IT, une variable « compteur_IT » est incrémenté et est affichée sur les leds.

```
LEDS = 0xB0000000
PILE = 0x200
@0        ba progp

// forcément après la fin du handler d'IT
progp:    set PILE, %sp
          set compteur_IT, %r1
          st %r0, [%r1]
pbcle:    nop
          nop
          nop
          ba pbcle
compteur_IT:  .word 0
```

```
@0          ba progp
@1  handler_IT:   push %r1
                  push %r2
                  push %r20
                  set LEDS, %r20
                  set compteur_IT, %r1
                  ld [%r1], %r2
                  inc %r2
                  st %r2, [%r20]
                  st %r2, [%r1]
                  pop %r20
                  pop %r2
                  pop %r1
                  reti
```

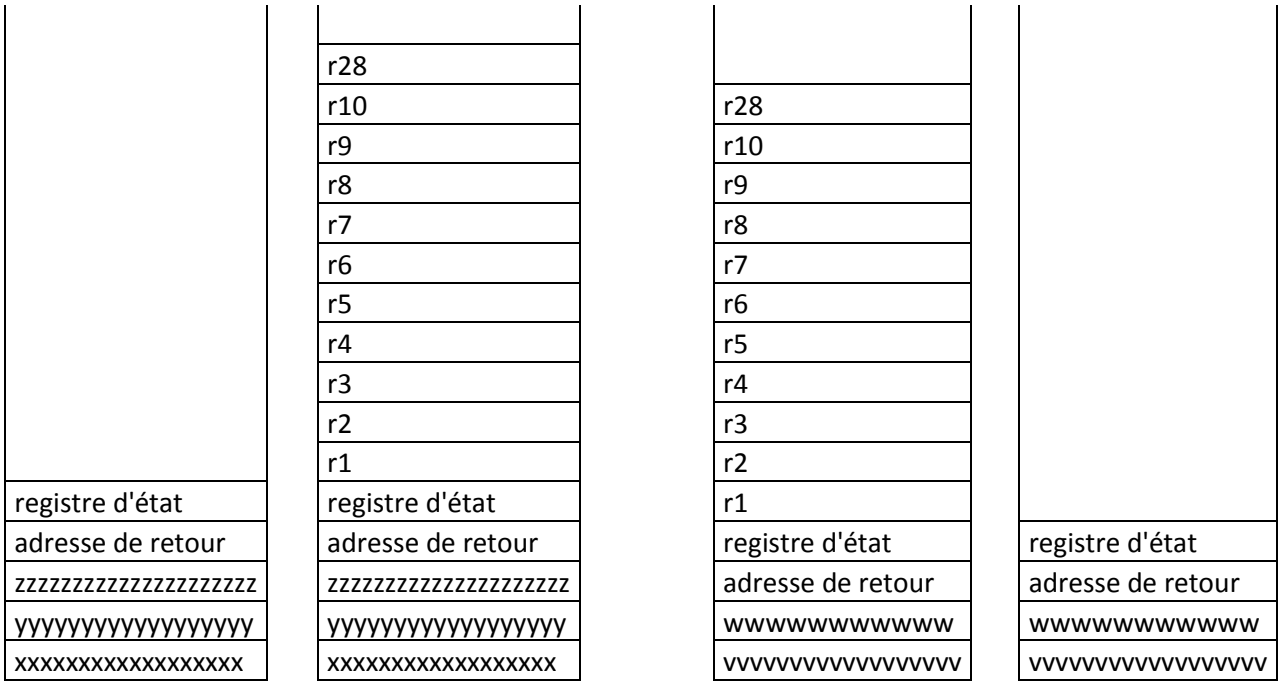
2- Commutation de tâches

Nous allons utiliser l'interruption IT pour implanter une version simplifiée de la commutation de tâches. C'est le travail que fait le système d'exploitation pour suspendre, à intervalle de temps régulier, le processus en cours et donner la main au processus élu.

On suppose disposer de deux programmes susceptibles d'utiliser chacun tous les registres compris entre r1 et r10 (pour alléger le contexte sauvegardé). On verra en TP ce que font ces programmes.

Lorsque le programme1 est suspendu, tout son contexte doit être sauvegardé pour qu'il puisse être récupéré une fois il sera activé de nouveau, et reprendre son exécution comme si aucune interruption n'a eu lieu. Ce contexte est composé de tous les registres qu'il est supposé utiliser (on ne sait pas ce qu'il fait et ce qu'il utilise), y compris le registre %r28 (car il a le droit d'appeler des sous-programmes).

Mais où ce contexte doit-il être sauvegardé ? dans une pile générale commune à toutes les tâches, ou dans une pile spécifique à chaque tâche. La réponse est que chaque tâche doit disposer de sa propre pile. On peut ci-dessous l'évolution de la pile du processus suspendu et celle du processus élu, de l'entrée dans le commutateur jusqu'à la sortie.



On peut donc résumer l'algorithme du commutateur de tâches comme suit :

Sauvegarder tous les registres de travail dans la pile du processus suspendu (qui est la pile courante)
Sauvegarder le pointeur de pile dans un emplacement qui lui est associé
Incrémenter le numéro du processus courant (stratégie simple)
Récupérer le pointeur de pile du processus élu
Récupérer tous les registres de travail du processus élu
Reprendre l'exécution du processus élu

On utilisera donc :

- Une variable Proc_courant qui contient le numéro du processus courant
- Un tableau de pointeurs de pile (Tab_sp) pour sauvegarder le pointeur de pile des processus après leur suspension

Ecrire le code du handler commutateur de tâches. On suppose les déclarations suivantes faites :

Variable Nombre_proc : indique le nombre de processus
Variable Proc_courant : indique le numéro du processus courant (de 1 à Nbre_proc)
Tableau Tab_sp : tableau pour sauvegarder les pointeurs de piles des processus.