TP2 : Simulation d'une C.P.U. et d'un système

CE TP DOIT ÊTRE RENDU.
Les modalités seront précisées dans le TP4.

1 Présentation de la machine

Pour suivre ce TP vous devez récupérer le prototype du simulateur qui est mis à votre disposition.

1.1 Mémoire centrale

La mémoire centrale de notre machine est composée de mots. Un mot mémoire est un entier de 32 bits (la taille des entiers sur une architecture PC classique). La mémoire contient quelques centaines de mots simulés par un simple tableau. Les adresses physiques sont contiguës et varient de zéro à 1023.

```
typedef int WORD; /* un mot est un entier 32 bits */

WORD mem[1024];   /* memoire                      */
```

1.2 Instructions de la C.P.U.

Notre C.P.U. exécute des instructions à taille fixe (un mot de 32 bits). Une instruction est composée d'un code opération, de deux numéros de registre et d'un argument. Vous pouvez voir les détails ci-dessous :

```
typedef struct {
    unsigned OP: 10;   /* code operation (10 bits)  */
    unsigned i:   3;   /* nu 1er registre (3 bits)  */
    unsigned j:   3;   /* nu 2eme registre (3 bits) */
    short    ARG;      /* argument (16 bits)        */
} INST;
```

Une instruction va ressembler à ceci : ≪ `ADD R1, R2, 1000` ≫. Elle va effectuer l'affectation `R1=R1+R2+1000`. Afin de ne pas rendre la création d'un programme trop pénible, une fonction est fournie pour implanter une instruction en mémoire.

```
void make_inst(int adr, unsigned code, unsigned i, unsigned j, short arg) {
    union { WORD word; INST fields; } inst;
    inst.fields.OP  = code;
    inst.fields.i   = i;
    inst.fields.j   = j;
    inst.fields.ARG = arg;
    mem[adr]        = inst.word;
}
```

1.3 Structure de la C.P.U.

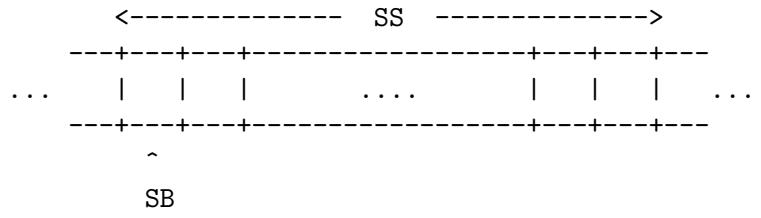
Le mot d'état du processeur est défini comme suit :

```
typedef struct PSW { /* Processor Status Word */
    WORD PC;         /* Program Counter       */
    WORD SB;         /* Segment Base          */
    WORD SS;         /* Segment Size          */
    WORD IN;         /* Interrupt number      */
    WORD DR[8];      /* Data Registers        */
    WORD AC;         /* Accumulateur          */
    INST RI;         /* Registre instruction  */
} PSW;
```

— `PC` : Le *Program Counter* (compteur ordinal) est un pointeur sur la prochaine instruction à exécuter (en fait un entier) ;
— `SB` et `SS` : voir section suivante ;
— `IN` : En cas d'interruption, la C.P.U. range dans ce registre la cause de cette interruption. Cette information peut être exploitée par le système d'exploitation ;
— `DR[i]` : La C.P.U. dispose de huit registres généraux pouvant contenir chacun un entier signé de 32 bits ;
— `AC` : Le registre accumulateur est un relais entre d'une part les instructions de test et d'autre part les instructions de branchement conditionnel. Le registre `AC` est également affecté par les instructions qui modifient le contenu d'un registre (chargement, calcul, incrémentation, ...).

1.4 Adressage logique et physique

Durant l'exécution, la C.P.U. adresse la mémoire en utilisant des *adresses logiques* c'est à dire des entiers compris entre 0 et la taille de la zone mémoire allouée à ce processus (*Segment*). Le début de cette zone mémoire est pointé par le registre `SB` (*Segment Base*) tandis que le registre `SS` en donne la taille (*Segment Size*).



pour chaque adresse logique ≪ a ≫ la C.P.U. calcule l'adresse physique ≪ p ≫ en appliquant l'algorithme suivant :

```
lire(a) :
   | si (a < 0) ou (a >= SS) <erreur adressage>
   | p = (SB + a)
   | renvoyer mem[p]
```

1.5 Simulation de la machine

Nous avons vu en cours que la C.P.U. passe son temps à alterner des cycles ou elle exécute du code utilisateur et des cycles ou elle exécute du code système. Elle passe du code utilisateur au code système

par une interruption et du code système au code utilisateur par un chargement du mot d'état processeur (ou *Processor Status Word*). On peut donc simuler ce comportement par le code ci-dessous :

```
mep.IN = INT_INIT; /* interruption INIT */
while (1) {
    mep = systeme(mep);
    mep = cpu(mep);
}
```

La fonction `cpu()` simule l'exécution du code utilisateur jusqu'à l'apparition d'une interruption. La fonction `systeme()` reprends la main, traite l'interruption et redonne la main au code utilisateur.

2 Nouvelles fonctions à réaliser

2.1 Tracer les interruptions

1. Pour l'instant le simulateur boucle sans rien afficher. Faites en sorte que le système indique les numéros d'interruption reçus ;
2. Faites en sorte que les interruptions d'erreur (instruction inconnue et erreur d'adressage) provoque l'arrêt du simulateur. Modifiez le programme exécuté pour faire apparaître ces deux interruptions ;
3. Modifiez le code du système pour que l'interruption `TRACE` provoque l'affichage des registres (pas tous, seulement `PC` et les `DR`).

2.2 Améliorer la C.P.U.

1. Nous allons maintenant ajouter de nouvelles instructions à la C.P.U. afin de pouvoir exécuter la simple boucle ci-dessous :

```
 0 : SUB R1, R1, 0          |  R1 = 0
 1 : SUB R2, R2, -1000      |  R2 = 1000
 2 : SUB R3, R3, -10        |  R3 = 10
 3 : SUB R4, R4, 0          |  R4 = 0
 4 : CMP R1, R2, 0          |  AC = (R1 - R2)
 5 : IFGT R4, 11            |  if (AC > 0) PC = R4 + 11
 6 : NOP                    |  no operation
 7 : NOP                    |  no operation
 8 : NOP                    |  no operation
 9 : ADD R1, R3, 0          |  R1 += R3
10 : JUMP R4, 4             |  PC = R4 + 4
11 : HALT                   |  HALT
```

   Pour pouvoir coder ce programme, il faut ajouter quatre instructions à notre C.P.U. (`IFGT`, `NOP`, `JUMP` et `HALT`).

2. Pour diminuer le nombre d'interruptions (et donc le nombre d'affichages réalisés par le système), faites en sorte que la C.P.U. exécute trois instructions avant de générer une **nouvelle** interruption de fin de tranche de temps (`INT_CLOCK`). L'interruption de trace n'est donc plus nécessaire.

2.3 Appels au système

Pour l'instant les affichages de notre simulateur sont réalisés par les traces du *PSW* faites par le système. Il est temps maintenant d'ajouter une nouvelle instruction, que nous appellerons `SYSC`, dont le but est de générer une interruption afin de donner la main au système. La partie argument de cette instruction indiquera au système l'action voulue et les registres indiqueront les éventuels paramètres de cette action.

1. Commencez par implanter l'appel système `SYSC_EXIT` qui provoque l'arrêt du processus demandeur et donc du système puisque nous avons, pour l'instant, un seul processus ;
2. Continuez avec l'appel `SYSC_PUTI` qui affiche l'entier stocké dans le premier registre de l'instruction `SYSC`. Vous pouvez maintenant placer cette instruction au coeur de la boucle et voir le déroulement de la boucle ;
3. Pour implanter le calcul de la taille mémoire alloué à un processus, il nous manque l'instruction de lecture mémoire que nous appellerons `LOAD` :

```
instruction       | action réalisée
------------------|-----------------------------------------------
LOAD Ri, Rj, k    | AC = Rj + k
                  | si (AC < 0) ou (AC >= SS) <erreur adressage>
                  | AC = mem[SB + AC]
                  | Ri = AC
                  | PC += 1
```