Université de Strasbourg

Département d'Informatique
Licence 2 Semestre 3 informatique
Architecture des Ordinateurs

Projet RISC-V : *Space Invaders*

Ce document décrit le projet à réaliser dans le cadre de l'UE architecture des ordinateurs. L'objectif de ce projet est d'implémenter les algorithmes permettant de réaliser un jeu de *space invaders*, dont le principe et les règles sont décrits sur cette page. Le programme sera à réaliser en assembleur RISC-V (code source `.s` ou `.asm`) et utilisera l'émulateur `Rars1_6` pour l'exécution. Le projet est à réaliser en binôme et devra être rendu sur la plateforme *Moodle* avant le dimanche 9 novembre à 23h55. Ce projet fera l'objet d'une soutenance de 10 minutes pendant la dernière séance de TP pendant la semaine du 10 novembre. Il est inutile de préparer un rapport ou un exposé. Pendant cette soutenance, l'examinateur vous demandera de commenter, de modifier, d'expliquer vos choix et de tester ou d'extraire des éléments de votre code, montrant ainsi que vous savez le maîtriser.

1 Spécifications

1.1 Le principe du jeu

Nous visons une version simplifiée du jeu *space invaders*.

Arène : Le jeu *space invaders* se déroule dans un espace 2D de forme carrée ou rectangulaire qu'on appellera l'*arène* (cf figure 1).

Envahisseurs : Au départ du jeu, une rangée de rectangles (en rouge sur la figure 1) représente des envahisseurs. Si vous le voulez, vous pouvez faire plusieurs rangées (4 rangées sur la figure), mais cela n'influencera pas l'évaluation. Les envahisseurs sont tous animés par le même mouvement. Ils commencent par aller de gauche à droite puis, avant qu'il y ait collision avec le bord droit de l'arène, ils descendent quelque peu vers le bas, puis vont de droite à gauche. De même, avant qu'il y ait collision avec le bord gauche de l'arène, ils descendent encore et vont de gauche à droite et ainsi de suite.

Sol : Le sol est situé à 1/5 de la hauteur de l'arène. Il est matérialisé par le bord supérieur des rectangles jaunes. Si les envahisseurs atteignent le sol, c'est la fin du jeu.

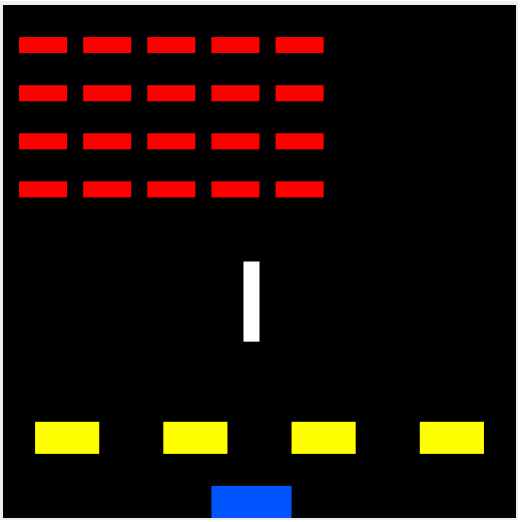

FIGURE 1 – Une arène avec 20 envahisseurs (rouge) un joueur (bleu) 4 obstacles (jaunes) et un missile (blanc)

Joueur : Sous la ligne du sol, le canon du joueur, représenté aussi par un rectangle (en bleu sur la figure 1), a un mouvement exclusivement horizontal. L'utilisateur déplace le joueur vers la droite (touche `'p'`) ou vers la gauche (touche `'i'`). En pressant la touche `'o'`, il peut tirer un missile.

missiles : Les missiles sont représentés par des lignes verticales (blanc sur la figure 1). Ils ont un mouvement exclusivement vertical. Les missiles tirés par le joueur vont vers le haut, les missiles tirés par les envahisseurs vont vers le bas. Le joueur tire en pressant la touche `'o'`. À chaque intervalle de temps régulier (réglable) un envahisseur choisi aléatoirement, tire un missile vers le bas. Lorsqu'un envahisseur est atteint par un missile tiré par le joueur, il disparaît. Un envahisseur est insensible aux missiles tirés par ses congénères. Lorsque le joueur est atteint par un missile, il perd une vie. Si le joueur n'a plus de vie ou si tous les envahisseurs ont été éliminés, c'est la fin du jeu.

Obstacles : Entre les envahisseurs et le joueur, il y a des obstacles ou des bâtiments (en jaune sur la figure 1) qui arrêtent les missiles tirés par les envahisseurs et par le joueur.

Vous l'aurez noté : par rapport à la description donnée sur la page wikipedia, nous introduisons plusieurs changements :

— Il y a un seul type d'envahisseurs ;
— Les envahisseurs, le joueur, et les bâtiments seront représentés par des rectangles.
— On peut se contenter d'une seule rangée d'envahisseurs.

— Il y a une seule vague d'envahisseurs et le jeu se termine lorsque tous les envahisseurs de cette vague sont éliminés, lorsque le joueur a perdu ses 3 vies ou lorsque les envahisseurs atteignent le sol.
— Les obstacles restent entiers (alors que, dans le vrai jeu, ils sont endommagés par les missiles et peuvent finir par disparaître) ;
— On ne demande pas qu'il y ait d'accélération au cours du jeu.
— On ne peut pas tirer sur un missile avec un autre missile.

1.2 La structure du projet : six parties

Ce projet est divisé en six parties. Les trois premières sont quasiment indépendantes. Les trois dernières parties, consistent essentiellement à assembler les parties précédentes pour réaliser le jeu complet. Chaque partie nécessite la réalisation d'un certain nombre de fonctionnalités imposées et non négociables. Pour l'implémentation de ces fonctionnalités, les différentes parties sont plus ou moins cadrées. Lorsqu'une procédure a été spécifiée dans le sujet, vous devez obligatoirement avoir écrit cette procédure. Dans le cas contraire, vous ne pourrez pas avoir tous les points de la partie, même si elle fonctionne parfaitement. Si aucune procédure n'est spécifiée pour une fonctionnalité, vous pouvez la réaliser comme vous le souhaitez. Si vous parvenez à finir le projet, vous pourrez rendre un seul fichier code. Sinon, vous pouvez rendre un fichier par partie.

2 Partie 1 : la pause

Un des appels système permet de mettre le processeur en sommeil pendant un certain nombre de millisecondes. Je vous laisse chercher cet appel sur cette page : `https://github.com/TheThirdOne/rars/wiki/Environment-Calls`. Grâce à cet appel système, écrire un programme qui écrit sur la sortie standard les entiers de 1 à 10 à raison d'un entier toutes les demie-secondes (500 ms). Si tout se passe bien, l'exécution de votre programme devra prendre exactement 5 secondes.

3 Partie 2 : l'entrée synchrone au clavier

3.1 Énoncé du problème

Vous connaissiez sans doute déjà un appel système correspondant au `scanf` en langage C. Cet appel système met le processus en sommeil en attendant que l'utilisateur fasse une saisie suivie de la touche *Entree*. Or dans le cas de notre jeu, l'utilisateur a besoin de guider le canon alors même que le jeu suit son cours. Autrement dit, le jeu ne doit pas se mettre en sommeil en attendant que le joueur choisisse la touche et presse la touche *Entree*.

3.2 Keyboard and Display MMIO Simulator

Pour cela, nous aurons recours à un nouvel outil de *Rars1_6*. Dans le menu *Tools*, choisir *Keyboard and Display MMIO Simulator* (MMIO : *Memory Mapped Input-Output*). Vous devriez voir apparaître quelque chose de semblable à la figure 2.

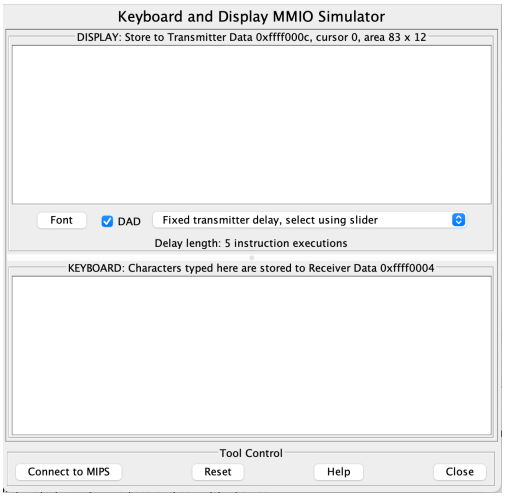

FIGURE 2 – Memory-mapped Input-output simulator

Presser le bouton *Connect to Program* (en bas à gauche), puis cliquer dans la fenêtre inférieure *Keyboard*. Grâce à cet outil, dès que vous pressez une touche du clavier (tout au moins celles qui sont associées à un code Ascii) l'événement est détecté et envoyé à la mémoire. Plus précisément, *Rars1_6* gère ces entrées-sorties au moyen de deux cases mémoire appelées, à tort, des *registres* :

— Le registre de contrôle du récepteur (RCR : adresse `0xffff0000`)
— Le registre de données du récepteur (RDR : adresse `0xffff0004`)

1. Par défaut (lorsqu'aucune touche n'est pressée), la valeur du RCR est 0 ;
2. Lorsqu'une touche associée à un code Ascii est pressée, la valeur du RCR passe à 1 et la valeur du RDR est mise au code Ascii de la touche pressée. Les deux registres gardent ces valeurs même si l'utilisateur cesse de presser la touche.
3. lorsque la donnée du RDR est lue, le RCR reprend la valeur de 0.

3.3 Tâche à réaliser

Écrire un programme qui affiche la valeur du registre `t0` toutes les demie-secondes (cf section 2).
— Le fait de presser la touche `'i'` doit diminuer la valeur de `t0`.
— Le fait de presser la touche `'p'` doit augmenter la valeur de `t0`.
— Le fait de presser la touche `'o'` doit arrêter le programme.
— Les autres touches ne doivent avoir aucun effet.

4 Partie 3 : les images

4.1 Les couleurs

Rars1_6 permet de visualiser certaines plages de la mémoire par une image *(bitmap)*. Chaque entier (4 octets) de cette plage représente une couleur. Voici la correspondance entre les entiers et les couleurs.

— L'octet de poids fort n'est pas pris en compte.
— L'octet suivant donne la composante rouge.
— L'octet suivant donne la composante verte.
— L'octet suivant donne la composante bleue.

Par exemple l'entier `0x0057ff03` (figure 3) représente la couleur dont les composantes rouge, verte et bleue sont respectivement 87, 255 et 3.

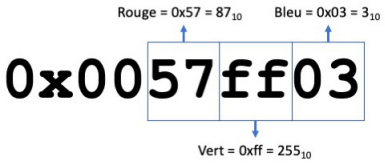

FIGURE 3 – Un entier exprimé en hexadecimal et la couleur associée

Autres exemples :
* L'entier `0x00ff0000` représente le rouge (rvb=(255,0,0)).
* L'entier `0x0000ff00` représente le vert (rvb=(0,255,0)).
* L'entier `0x00000000` représente le noir (rvb=(0,0,0)).
* L'entier `0x000055ff` représente une teinte de bleu (rvb=(0,85,255)).

4.2 *Bitmap Display*

Dans le menu *Tools*, choisir *Bitmap Display*. Vous devriez obtenir quelque chose de semblable à la figure 4.

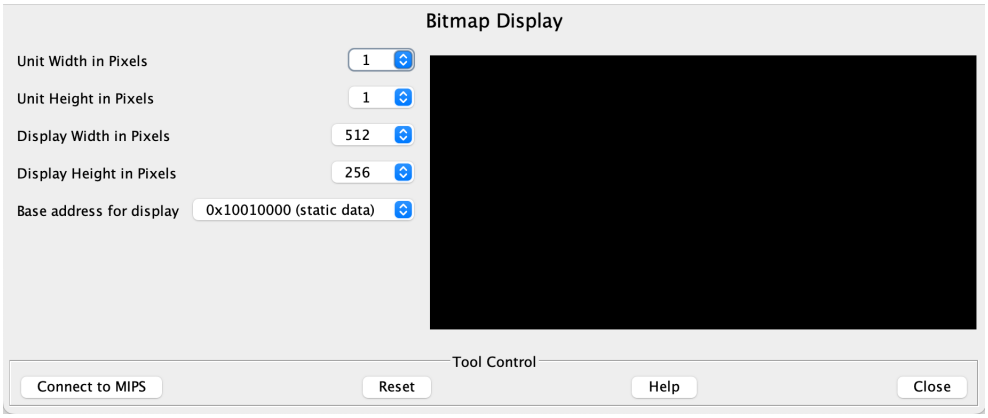

FIGURE 4 – L'interface *Bitmap Display* de *Rars1_6*

Cette interface possède 5 paramètres.
— ***Unit width in pixels*** et ***Unit height in pixels*** sont à 1. Cela signifie que chaque entier de la plage mémoire est associé à un seul pixel. Nous verrons (en 4.3) que ce n'est pas toujours le cas.
— ***Display width in pixels*** et ***Display height in pixels*** représentent respectivement la largeur et la hauteur de l'image exprimées en pixels. Remarquons que cela représente 131072 pixels, donc 131072 entiers, donc $131072 \times 4 = 524288_{10}$ octets. En hexadecimal, cela fait `0x80000`.
— ***Base address for display*** représente l'adresse du début de la plage de mémoire qui est représentée sur l'image. Ici, cette adresse vaut `0x10010000`. Nous avons vu (point précédent) que le volume nécessaire pour contenir les données de l'image était de `0x80000` octets. Donc la plage mémoire représentée ici est la plage comprise entre l'adresse `0x10010000` et l'adresse `0x10090000`. La première adresse de cette plage (`0x10010000`) correspond au pixel en haut à gauche de l'image. Les 512 entiers suivants (de `0x10010000` à `0x100107ff`) représentent les autres pixels du bord supérieur de l'image. Les 512 entiers suivants

(`0x10010800` à `0x10010fff`) représentent les pixels de la deuxième ligne. Et ainsi de suite. Dans la suite, cette plage de mémoire sera appelée la *mémoire image*.

Application : Pour vous assurer que vous avez compris le fonctionnement, écrire un programme qui écrit l'entier `0x00ff0000` (rouge) sur la première moitié des entiers de la mémoire image. (de `0x10010000` à `0x10050000`). Vous devriez obtenir une image rouge sur la moitié supérieure et noire sur la moitié inférieure. Avant d'exécuter ce programme, assurez-vous d'avoir pressé le bouton *Connect to Program* qui établit la connexion entre l'image et la mémoire. Attention ! Lorsque vous relancez le programme, la plage mémoire n'est pas réinitialisée à 0. Pour repartir d'une image noire, presser le bouton *Reset*.

4.3 Les *Units* : de gros pixels

Pour faire une image comme celle de la figure 1, il n'est pas nécessaire d'avoir $512 \times 256 = 131072$ pixels. Au contraire, le fait d'avoir à rafraîchir tant de pixels à chaque frame peut grandement ralentir le jeu. Heureusement, *Rars1_ 6* permet d'associer chaque entier de la plage non plus à un seul pixel, mais à de petits carrés ou rectangles appelés *Units*. Ainsi, si vous choisissez une valeur de 8 pour *Unit width in pixels* et *Unit height in pixels*, alors chaque entier de la plage mémoire contrôlera la couleur d'un petit carré de 8×8 pixels.

Il en résulte que chaque ligne sera représentée non plus par 512 entiers, mais par $512/8 = 64$ entiers. L'image sera constituée de 64 colonnes et 32 lignes, ce qui représente $64 \times 32 = 2048$ entiers. L'image de la figure 1 a été obtenue avec une image de 256×256 pixels et des *Units* de 8×8 pixels.

Application : Pour vous assurer que vous avez compris le fonctionnement, reprendre l'exercice d'application de la section précédente, mais pour une image de 256×256 pixels et des *Units* de 8×8 pixels et colorier en rouge la moitié supérieure de l'image.

Dans toute la suite, quand on parlera de *pixels*, il faudra entendre *Units*, c'est à dire un gros pixel dont la couleur est déterminée par un entier.

4.4 Tâches à réaliser

1. Définir quatre variables globales représentant la largeur et la hauteur de l'image en pixels et la largeur et la hauteur des *Units*. Ces variables doivent être conformes aux valeurs que vous aurez choisies dans l'interface de *Bitmap Display*. Dans toute la suite, pour changer la taille de l'image et des *Units*, il devra suffire de modifier la valeur de ces quatre variables (sans apporter d'autres changements par ailleurs dans le programme).
2. Calculer deux variables globales ou deux fonctions `I_largeur` et `I_hauteur` représentant respectivement la largeur et la hauteur de l'images en nombre de *Units* et non en pixels. Ces valeurs devront être calculées en fonction des 4 variables mentionnées en 1.
3. Écrire une fonction `I_creer` qui ne prend aucun argument et qui alloue la mémoire image (en fonction des valeurs des 4 variables mentionnées en 1.). Le calcul des variables `I_largeur` et `I_hauteur` peut se faire dans le cadre de cette fonction. Au retour de la fonction, une variable globale (que j'ai appelé `I_buff`) devra contenir l'adresse de la mémoire allouée. Cette variable pourra être utilisée par toutes les fonctions ci-dessous.
4. Écrire une fonction `I_xy_to_addr` qui, à partir de deux arguments entiers (dans `a0` et `a1`), représentant l'abscisse et l'ordonnée d'un pixel (*Unit*) retourne, dans `a0`, l'adresse de l'entier dans la mémoire image associé à ce pixel.
5. Écrire une fonction `I_addr_to_xy` qui, à partir d'un argument entier (dans `a0`) représentant l'adresse d'un entier dans la mémoire image retourne (dans `a0` et `a1`) l'abscisse et l'ordonnée du pixel correspondant.
6. Écrire une fonction `I_plot` qui, à partir de trois arguments entiers (dans `a0`, `a1` et `a2`) représentant l'abscisse et l'ordonnée d'un pixel et d'une couleur, colorie le pixel correspondant avec cette couleur. On demande expressément à ce que cette procédure fasse appel à `I_xy_to_addr`. Arrivé à ce point, vous devez décider si vous préférez représenter un pixel par une abscisse et une ordonnée ou plutôt par un seul entier représentant son adresse dans la mémoire image. Je vous demande d'expliciter ce choix sous la forme d'un commentaire dans votre code.
7. Écrire une fonction `I_effacer` qui ne prend pas d'arguments mais initialise tous les pixels de l'image avec la couleur noire.
8. Écrire une fonction `I_rectangle` qui prend 5 arguments entiers (`a0` à `a4`) dessine un rectangle dont le coin supérieur gauche est au pixel `(a0,a1)` avec une largeur de `a2` pixels, une hauteur de `a3` pixels et une couleur donnée par `a4`. On demande expressément à ce que cette procédure fasse appel à `I_plot`.
9. Réaliser une animation de rectangles. Pour cela, une première idée consiste à dessiner un rectangle, attendre 50ms, l'effacer, redessiner le rectangle plus loin, attendre 50ms, effacer de nouveau et redessiner plus loin etc. Est-ce que le mouvement est fluide ?

10. Vous l'aurez compris, pour rendre le mouvement un tant soit peu fluide, il ne faut jamais complètement effacer l'image, mais la remplacer par une autre image. Dans la plupart des applications, pour afficher une animation, on utilise deux mémoires image (un double buffer). Soient deux variables `I_visu` et `I_buff` qui représentent l'adresse de ces mémoires d'image. Seules les données dans `I_visu` sont visualisées. À chaque frame, on efface le contenu de `I_buff` et on dessine les rectangles dans `I_buff`. Lorsque le dessin est terminé, on transfère toutes les données dans `I_visu`. Écrire une fonction `I_buff_to_visu` qui transfère les données de `I_buff` dans `I_visu`.
11. Réaliser une animation représentant un rectangle qui se déplace d'un point à un autre sur l'image.

5 Partie 4 : Gestion des données

Vous avez déjà créé des variables globales représentant les dimensions de l'image et l'adresse des mémoires image.

1. Créer d'autres variables globales représentant vos choix :
   — Pour le joueur : la position, la largeur, la hauteur, la couleur du rectangle représentant le joueur ;
   — Pour les envahisseurs : leur nombre, les dimensions des rectangles, leur couleur, l'espacement horizontal, l'augmentation de l'ordonnée quand la rangée atteint le bord droit ou gauche, le rythme des tirs de missile etc ;
   — Pour les obstacles : leur nombre, les dimensions des rectangles, leur couleur, l'espacement horizontal (l'ordonnée du bord supérieur des rectangles doit etre au 1/5 de la hauteur de l'image) ;
   — Pour les missiles : la couleur, la vitesse, la longueur, éventuellement l'épaisseur, le nombre maximal de missiles présent simultanément etc.
2. Les données ci-dessus caractérisent chaque type d'objet dans leur globalité. Il reste aussi des attributs qui caractérisent chaque individu : position, direction, mort/vivant etc. En déduire les données nécessaires pour caractériser chaque envahisseur, chaque obstacle, chaque missile. En déduire des structures de données pour chaque type d'objet. Par exemple vous pourriez décider qu'un objet qui serait caractérisé par trois entiers pourrait être représenté par une suite de 3×4 octets en mémoire. Ainsi pour désigner cet objet il suffirait de fournir l'adresse du premier entier. Je vous demande d'expliciter dans votre code, sous la forme de commentaires, la structure que vous avez choisi pour représenter chaque martien, chaque obstacle, chaque missile.
3. Étant données ces variables, écrire les fonctions permettant de créer les différents objets : joueur, envahisseurs, obstacles, missiles. Ces fonctions (qu'on peut appeler respectivement `J_creer`, `E_creer`, `O_creer`, `M_creer`) devront, en particulier, initialiser la valeur des variables (par exemple la position du joueur, des envahisseurs et des obstacles. Elles devront aussi, dans le cas des envahisseurs, des obstacles et des missiles, organiser toutes les données en tableaux et allouer dynamiquement l'espace mémoire nécessaire dans le tas en fonction du nombre d'exemplaires souhaités pour ces objets.
4. Écrire les fonctions d'affichage correspondantes : `J_afficher`, `E_afficher`, `O_afficher`, `M_afficher` qui devront faire appel à `I_rectangle`.
5. Afficher, en statique, toute une scène (comme sur la figure 1). On devra pouvoir contrôler le nombre d'envahisseurs, de missiles et d'obstacles, leurs dimensions, leur couleur en modifiant les variables globales de la question 1 de cette partie.

6 Partie 5 : Le mouvement

1. Écrire une fonction `J_deplacer` qui, en fonction des entrées au clavier, permet de déplacer le joueur horizontalement.
2. Écrire une fonction `M_deplacer` qui, en fonction des données relatives à un missile (en particulier sa direction : vers le haut ou vers le bas) et de sa position courante permet de déterminer sa position à la frame suivante et de la mettre à jour.
3. Écrire une fonction `E_deplacer` qui, en fonction des données relatives aux envahisseurs et de leur position courante, permet de déterminer leur position à la frame suivante et de la mettre à jour. Ainsi, une des données à mémoriser est celle de la direction courante du mouvement : de gauche à droite ? ou de droite à gauche ? Cette fonction devra détecter le fait que certains envahisseurs vont sortir du cadre. Elle devra alors les descendre d'un étage et les diriger dans l'autre direction.
4. Arrivés à ce stade, vous ne savez pas encore détecter le contact entre un missile et un joueur/obstacle/envahisseur. Mais vous avez tous les éléments pour faire afficher et se déplacer tous les éléments (Joueur, envahisseurs, missiles) frame après frame à raison de 50ms par frame.

7 Partie 6 : Space Invaders

1. Écrire une fonction appelée : `M_intersecteRectangle` qui prend en argument l'adresse d'un missile et un rectangle (quelque soit la façon que vous avez choisie pour représenter un missile et un rectangle). Cette fonction devra retourner (dans `a0`) la valeur 1 si l'arrière du missile est hors du rectangle et qu'un des points du missle est dans le rectangle. Elle devra retourner la valeur 0 dans tous les autres cas.
2. Faire tourner le jeu en détectant les collisions entre les missiles et les différents éléments du jeu.
3. En tenir compte pour faire disparaître les envahisseurs touchés et diminuer le nombre de vies du joueur et, enfin, détecter la fin du jeu.

8 Structure et lisibilité

Votre programme sera évalué non seulement sur les résultats qu'il produit et son efficacité, mais aussi sur sa structuration en procédures et sur sa lisibilité, notamment au moyen de commentaires qui indiquent la signification de vos opérations en termes algorithmiques. Ci-dessous, un style à éviter absolument (figure 5) et un style correct (figure 6).

```
mv a0,s0     # a0 <- le contenu de s0
jal toto     # appelle toto
mv s2,a0     # s2 <- le contenu de a0
```

FIGURE 5 – Un code illisible et des commentaires inutiles : À PROSCRIRE!!!

```
mv a0,s0         # a0 <- l'adresse des rectangles envahisseurs
jal rech_pixel   # recherche le pixel suivant
mv s2,a0         # s2 <- l'indice du pixel choisi
```

FIGURE 6 – Des procédures avec des noms porteurs de sens et des commentaires qui rendent le code lisible et qui associent le programme à l'algorithme sous-jacent.

9 Barème indicatif

Ce barème est *indicatif*. Cela signifie qu'il est susceptible de subir des modifications mineures.

2 points La pause (section 2)
2 points L'entrée synchrone au clavier (section 3)
6 points Les images (section 4)
4 points La gestion des données (section 5)
3 points Le mouvement (section 6)
2 points Space invaders (section 7)
1 point Lisibilité et structuration

L'obtention de ces points est soumise au fait que, pendant la soutenance, vous puissiez, à la demande de l'examinateur :

- localiser l'endroit, dans votre programme, où vous effectuez un traitement donné ;
- expliquer la façon dont vous avez mis en œuvre un traitement donné.
- modifier un traitement donné, en place, ou après l'avoir extrait du programme.

Il est également soumis au fait que vous sachiez gérer la pile des appels de fonction. Ainsi, le fait d'avoir un programme qui marche parfaitement n'est ni nécessaire ni suffisant pour avoir une bonne note. Pas suffisant : un programme qui marche sans que vous puissiez le modifier ou l'expliquer vous donne 0/20. Pas nécessaire : si vous savez maîtriser les 4 premières parties avec un code lisible et structuré, vous aurez 15/20.
Les deux dernières parties (le mouvement et, surtout, space invaders) sont les plus difficiles. Le petit nombre de points qui leur sont attribués ne rendent pas compte de cette difficulté. Nous avons fait ce choix car le fait d'avoir fini les quatre premières parties représente déjà un travail conséquent. Ainsi celles et ceux qui n'ont pas réalisé les deux dernières parties pourront avoir de bonnes notes et que celles et ceux qui auront fini le projet pourront avoir de meilleures notes.
Dans l'éventualité d'un projet non fini, il est très important de pouvoir montrer le fonctionnement de certaines parties pendant la soutenance. Il vaut mieux, de loin, montrer une ou deux parties qui fonctionnent qu'un jeu complet mais qui ne compile pas. C'est la raison pour laquelle, je vous invite vivement à versionner votre travail. Chaque version doit correspondre à un programme plus ou moins limité mais opérationnel. Ainsi, vous serez certains qu'au moment de la soutenance, dans le pire des cas, il vous suffira de revenir à la version précédente pour avoir quelque chose de montrable.