Documentation pour le développeur

Ce projet a pour objectif de simuler en langage C le fonctionnement d'un processeur fictif. Tout au long de notre code, nous utilisons les conventions suivantes :

- les noms de variables et de fonctions sont en français
- les commentaires sont en français
- les noms de variables et de fonctions sont de la forme nom_variable_ou_fonction
- le code est indenté de façon conventionnelle

Afin d'implémenter ce simulateur, nous avons divisé les tâches en deux grandes parties. Dans la 1ère partie, le programme lit un fichier texte qui contient toutes les instructions en langage assembleur que le processeur doit exécuter. Puis le programme, après avoir vérifié qu'elles ne comportent pas d'erreurs de syntaxe, transforme les instructions en hexadécimal (ie en langage machine) et les stocke dans un nouveau fichier texte généré nommé *hexa.txt*. Dans la 2ème partie, le programme doit lire *hexa.txt* et exécuter les instructions lues comme le ferait un processeur.

I. Génération du fichier d'instructions en hexadécimal (langage machine)

Tout d'abord, le programme lit le fichier en assembleur ligne par ligne, tout en utilisant un compteur afin de garder une trace du numéro de l'instruction lue. On souhaite pouvoir ignorer d'éventuelles lignes vides dans le fichier d'entrée en assembleur. Par conséquent, on incrémentera ce compteur seulement si la ligne lue n'est pas vide car on ne stocke en mémoire que les instructions, pas les lignes vides. Il est en effet important de ne donner aux étiquettes que des adresses valides dans la mémoire, et les lignes vides peuvent fausser le dénombrement des instructions. Afin de déterminer si une ligne est vide, on utilise la fonction **est_ligne_vide**.

A. Première lecture du fichier et gestion des étiquettes (Estelle Rohan)

La première tâche que nous avons effectuée est la lecture d'éventuelles étiquettes en début de chaque ligne. Pour stocker les étiquettes, nous avons créé une structure **etiquette,** contenant le nom de l'étiquette ainsi que le numéro de la ligne où elle apparaît (deux informations qui seront utiles par la suite lors de la lecture d'une instruction de saut); de plus, on utilise un tableau afin de stocker les pointeurs sur ces étiquettes. Depuis le fichier texte en assembleur, on récupère (dans une chaîne de caractères nommée **ligne)** et on traite une par une chaque instruction en assembleur (si elle n'est pas une ligne vide), en se servant de la fonction **gerer_etiquette**. Pour chaque instruction, cette fonction parcourt **ligne** afin de détecter un ':' . S'il y en a un, une structure **etiquette** est créée, dans laquelle on stocke le nom de l'étiquette et le numéro de ligne/instruction à laquelle elle se rapporte.

Cette première lecture du fichier pour repérer les étiquettes était indispensable. En effet, on est obligé de lire 2 fois le fichier en assembleur. Si on avait décidé de lire toutes les informations de l'instruction en un seul passage, on aurait pu rencontrer lors d'un saut un nom d'étiquette encore inconnu (typiquement un saut vers une étiquette **fin,** qui ne se trouve qu'après le moment où elle est appelée) : on aurait alors été dans l'incapacité de remplacer l'étiquette par l'adresse mémoire associée. Après ce constat, nous avions l'intention de repérer et stocker l'éventuelle

étiquette de l'instruction, puis de la supprimer de l'instruction. Enfin nous voulions stocker les instructions modifiées dans un tableau que nous exploiterions dans un deuxième temps afin de récupérer les champs code op., Dest, Src1, Imm et Src2. Nous avons cependant modifié l'idée initiale et avons décidé, pour économiser de la mémoire, de ne pas stocker nos instructions modifiées et de lire une deuxième fois le fichier. Ainsi, la première lecture sert à repérer et stocker les étiquettes, mais sans les supprimer des instructions. Puis dans la 2ème lecture du fichier, on va récupérer ligne par ligne les instructions, et si l'instruction lue n'est pas une ligne vide, supprimer sur le moment l'éventuelle étiquette du début de l'instruction avec la fonction **supp_etiquette**, et transmettre l'instruction modifiée à une fonction **lire_instructions**. On crée également un compteur de lignes vides afin d'indiquer le bon numéro de ligne à l'utilisateur lorsqu'une erreur est détectée.

B. Deuxième lecture du fichier et lecture des instructions (Julie Pibouteau)

La fonction **lire_instructions** a plusieurs rôles : elle doit récupérer tous les paramètres de l'instruction en assembleur en vérifiant que les instructions lues ne comportent pas d'erreurs. Pour cela, on crée une structure **instruction** qui contiendra le code-opération de l'instruction, ses valeurs dest, src1, imm et src2, ainsi qu'un code d'erreur et un message d'erreur. L'idée ici est de créer une variable de type instruction dans la fonction **main**. On ne l'initialise pas dans **lire_instructions** mais dans le **main** pour n'en créer qu'une seule au lieu de recréer une structure à chaque nouvelle instruction assembleur lue. Puis on fait modifier cette structure par pointeurs pour chaque ligne en assembleur par la fonction **lire_instructions**. Celle-ci renvoie un pointeur sur la structure d'instruction après avoir rempli ses champs. Si au cours de la lecture, une erreur est détectée, le code-erreur et le message d'erreur sont mis à jour, et le pointeur sur la structure d'instruction est immédiatement renvoyé. Sinon on remplit tous les champs de la structure avec les valeurs lues et on renvoie le pointeur sur cette structure d'instruction.

Les différents codes-erreurs prévus sont :

0 : pas d'erreur
1 : mot-clé de l'opération non reconnu
2 : nombre d'opérandes insuffisant
3 : nombre d'opérandes trop important
4 : opérande avec une syntaxe incorrecte

Tout d'abord, on transforme la ligne d'instruction assembleur en retirant les virgules et les parenthèses (afin de simplifier la lecture avec le sscanf) puis on étudie le code-opération. Dans la fonction **main,** nous avons créé un tableau servant à recenser les mots-clés d'opérations pour pouvoir les comparer à l'opération lue. Chaque opération est stockée dans la case d'indice égal à son code-opération (par exemple, "add" est stockée dans la case d'indice 0). S'il n'y a pas d'erreur, la lecture continue selon le type d'instruction lue. Nous avons choisi de regrouper les instructions selon le nombre d'opérandes à lire. Ainsi, si l'instruction est arithmétique, logique, aléatoire ou de transfert, elle est lue par la fonction **lire_IAL_IT**. Une instruction de saut est lue par la fonction **lire_IS**. Une instruction d'entrée/sortie est lue par la fonction **lire_IES**. Chacune de ces fonctions annexes continue la lecture selon son type d'instruction et vérifie les erreurs potentielles par le biais des fonctions **registre_incorrect**, **src1_incorrecte** et **src2_incorrecte**. Le cas de l'opération "hlt" est mis à part car aucune information autre que le mot-clé opération n'a besoin d'être lue.

C. Gestion des erreurs dans les instructions lues (Julie Pibouteau)

Dans tous les cas, après avoir vérifié que le mot-clé opération est correct (via le tableau d'opérations), on vérifie que le nombre d'opérandes dans le reste de l'instruction assembleur est correct. Pour cela, on utilise la valeur de retour du sscanf (qui indique le nombre de valeurs qui ont pu être lues selon le format défini) et on peut ainsi éventuellement renvoyer une erreur (si la valeur de retour du sscanf est plus petite que prévue, alors il manque des opérandes). Pour déterminer s'il y a trop d'arguments, on doit essayer de lire une valeur en plus (une chaîne de caractères initialisée à l'octet nul). En effet, si on demande à sscanf de lire 3 opérandes et que la chaîne lue en contient une de trop, la valeur sera simplement ignorée et on ne détectera pas l'erreur. Si la chaîne surnuméraire n'est plus réduite à l'octet nul après la lecture, c'est qu'il y a trop d'opérandes.

Ensuite, selon le type d'instruction, on teste si les valeurs lues sont bien des registres ou des valeurs immédiates.
Pour les registres dest et src1, on vérifie que la valeur lue est bien un registre (débute par un 'r'), est suivie immédiatement d'une valeur numérique, et que cette valeur numérique est comprise entre 0 et 31.
Pour la valeur src2, on détermine si c'est un registre, une valeur immédiate en base 10 (débute uniquement par '#') ou une valeur immédiate en base 16 (débute par "#h"). Les vérifications d'erreurs pour les registres sont les mêmes que pour dest et src1. Pour les valeurs en base 10, on vérifie qu'elles ne sont composées que de chiffres (et éventuellement un signe moins au début), et si ce nombre n'excède pas la valeur de 16 bits signés (il doit appartenir à [-32768, 32767]). Pour les valeurs en hexadécimal, on vérifie qu'elles ne sont composées que de symboles hexadécimaux (les chiffres de 0 à 9 et les lettres de A à F) et que la valeur n'excède pas 16 bits signés (taille des registres). Il y a toutefois le cas particulier de la valeur src2 lorsque c'est une adresse pour un saut. Elle doit être comprise entre 0 et 65535, pour correspondre à des adresses mémoire valides.
Si la valeur n'est ni un registre, ni une valeur immédiate, il y a deux possibilités : une erreur est présente, ou la valeur lue est un nom d'étiquette. Dans la fonction **lire_IS** on vérifie si la valeur lue appartient au tableau d'étiquettes créé au début. Si c'est le cas, on met la valeur imm à 1, et la valeur de src2 de la structure d'instruction à l'adresse de l'étiquette. L'adresse de l'étiquette est égale à 4 * (numéro de ligne) de l'instruction car chaque instruction fait une taille de 32 bits et la mémoire est composée de cases de 1 octet, donc une instruction est stockée sur 4 cases mémoire.

D. Traduction des instructions en hexadécimal et écriture dans le fichier en hexadécimal (Estelle Rohan)

Lorsque **lire_instructions** s'arrête, le pointeur sur la structure d'instruction est renvoyé à la fonction **main.** On utilise alors le code-erreur pour déterminer si l'instruction est correctement écrite. Si ce n'est pas le cas, on affiche le message d'erreur associé à la structure d'instruction, on libère la mémoire allouée jusqu'ici et la simulation s'arrête, on ne génère pas de fichier contenant les instructions en hexadécimal. Pour cela, on utilise **remove()** pour supprimer le fichier hexadécimal.
Sinon, on va créer un tel fichier et y écrire les instructions en hexadécimal. Pour chaque instruction, après l'avoir lue et vérifiée, on utilise la fonction **generateur_hexa**. Cette fonction prend en argument le code opération et les opérandes de l'instruction mais aussi une chaîne de caractère (*hexa)* qui recevra l'écriture hexadécimale de l'instruction.

L'instruction en binaire devant être de la forme :

code op (5 bits) | dest (5 bits) | src1 (5 bits) | imm (1 bit) | src2 (16 bits)

L'idée initiale était donc de créer un tableau de 32 int (des 0 et des 1) pour traduire l'instruction en binaire par le biais de divisions euclidiennes par 2. Puis on aurait traduit l'instruction en hexadécimal en regroupant les symboles binaires par 4 et en les identifiant via un switch dont les cas sont les nombres de 0 à 15. Néanmoins, il nous est venu une autre méthode plus simple à implémenter. Pour passer directement des opérandes en décimal à un grand nombre décimal (un int qui correspond au grand nombre binaire à 32 bits), il faut multiplier les valeurs des opérandes par les bonnes puissances de 2. Ainsi on dit que la valeur décimale de notre instruction est :

val_decimale = src2 + imm*(2^16) + rs*(2^17) + rd*(2^22) + code_op*(2^27)

Cependant, si src2 est négative, cela fausse la valeur décimale de l'instruction et l'hexadécimal ensuite généré sera alors faux. Il faut donc calculer son complément à deux avant de combiner toutes les opérandes. En effet, cela permet de récupérer un nombre positif ayant la même représentation binaire, donc hexadécimale. (Il suffira alors de faire l'opération inverse pour retrouver la valeur d'origine dans la partie lecture de l'hexadécimal.)
Pour récupérer le complément à deux si et seulement si src2 est négative, on fait :
src2 = (src2+65536)%65536.

Puis on traduit la valeur de l'instruction en hexadécimal et on la stocke en même temps dans la chaîne *hexa* passée en argument avec : sprintf(hexa, "%.8X", val_decimale).
"X" permet d'écrire l'hexadécimal en majuscules et ".8" permet de forcer l'écriture à 8 caractères, ainsi des zéros non significatifs seront ajoutés si nécessaire.
Puis dans le **main**, on écrit cette valeur dans le fichier en hexadécimal, *hexa.txt*.

Lorsqu'on a lu toutes les instructions en assembleur et écrit *hexa.txt,* on libère la mémoire allouée dynamiquement dans cette première partie.

II. Exécution des instructions en hexadécimal (Estelle Rohan)

Maintenant que les instructions ont été traduites en langage machine, on va pouvoir les exécuter.

A. Initialisation des éléments de la machine fictive

Afin de simuler le fonctionnement d'une machine fictive composée d'un processeur et d'une mémoire, on crée un tableau **Mem** de 65536 char représentant la mémoire de la machine, une case du tableau correspondant alors à un octet. Chaque instruction à exécuter faisant 32 bits, elle occupe 4 cases de la mémoire.
A première vue, nous voulions représenter la mémoire avec des unsigned char pour pouvoir facilement y mettre l'instruction en hexadécimal. Cependant nous nous sommes rendues compte qu'il fallait aussi pouvoir stocker des nombres négatifs en mémoire. Donc nous avons choisi de modéliser la mémoire avec un tableau de char (donc dans l'intervalle de représentation [-128,127]) puis à chaque fois de repasser ces char en leur valeur positive dans [0,255] pour pouvoir lire l'instruction, en faisant leur complément à 2 lorsqu'ils sont négatifs.
On crée également un tableau **R** de 32 shorts (représentant les 32 registres généraux de 16 bits du processeur), une variable PC initialisée à 0 (représentant le registre PC qui contient l'adresse

de la prochaine instruction à effectuer) et 3 variables Z, C, N (représentant les 3 bits du registre d'état). Dans un processeur, les registres ne sont jamais initialisés mais ici, nous les initialisons à zéro afin d'éviter toute erreur.

B. Lecture de hexa.txt et stockage des instructions dans la mémoire

Il s'agit de lire le fichier en hexadécimal instruction par instruction, tout en comptant le nombre de cases mémoires remplies, et de stocker chaque instruction dans la mémoire (4 octets = 4 cases mémoires par instruction). Pour ce faire, comme on a 32 bits donc 8 symboles hexadécimaux, on va lire les symboles hexadécimaux deux par deux grâce au format "**%2s**" (permet de récupérer une chaîne de maximum 2 caractères). Puis on récupère le nombre représenté par cette nouvelle chaîne, et on le stocke dans une case mémoire.

C. Exécution du programme

Tout d'abord on vérifie que l'adresse contenue dans PC est valide, c'est-à-dire positive, et inférieure au nombre de cases mémoires remplies, afin d'ajouter une sécurité supplémentaire pour éviter d'accéder à des cases mémoires interdites si l'utilisateur oublie d'arrêter son programme par exemple. Et tant que PC est valide, on exécute l'instruction à l'adresse correspondante.
Tout d'abord, on la transforme en binaire avec la fonction **conv_instr_memoire_vers_tab_bin** qui traduit octet par octet, puis assemble tout dans un tableau de 32 int (composé de 0 et de 1).
On peut ensuite utiliser la fonction **conv_bin_vers_operandes** qui prend en argument le tableau de 32 int représentant l'instruction et des pointeurs vers les variables stockant le code opératoire et les opérandes afin de les modifier directement. La fonction va lire les 5 premiers bits du tableau et les traduire en décimal pour identifier le code opératoire, et ainsi de suite avec les autres opérandes. Pour cela, elle fait appel à une autre fonction **conv_bin_vers_int(*int nb_bits, int bin[]*)** qui renvoie l'entier correspondant aux *nb_bits* premiers bits stockés dans *bin*[]. Ainsi, pour récupérer les opérandes il suffira d'appeler cette fonction avec le bon nombre de bits et le bon pointeur sur le tableau de 32 int.
Toutefois, pour src2, on rappelle que la valeur d'origine était peut être négative. Donc si le code opératoire ne correspond pas à un saut (car src2 doit pouvoir dans ce cas valoir entre 0 et 65535) et que src2 > 32767, c'est que src2 est en fait négative. Il faut donc ensuite calculer son complément à deux pour retrouver la bonne valeur (src2 = src2 - 65536).

Pour traiter cette instruction, nous avons ensuite créé un switch dont les cas sont les numéros de code opératoire. Pour chaque cas, il faut effectuer l'instruction correspondante avec les opérandes récupérées plus tôt.
Cependant, afin de ne pas distinguer à chaque fois les cas où src2 est un registre ou une valeur immédiate, on effectue avant le switch l'opération suivante :

```
if (imm == 0)
      src2 = R[src2];
```

Cela permet de faire en sorte que la variable src2 contienne bien directement soit la valeur immédiate, soit la valeur contenue dans le registre.
Il faut également mettre à jour PC avant le switch. En effet, si on le faisait après le switch, les sauts ne pourraient jamais être effectués. On incrémente donc PC de 4, car une instruction prend 4 octets en mémoire.

Puis dans le switch, pour chaque instruction effectuée, on met à jour les registres d'état avec les fonctions **registres_etat** et **registres_etat_ZN** (cette dernière permettant de gérer C à part dans les cas particuliers). Dans ces fonctions :
Le bit Z est mis à jour en testant si la valeur obtenue est nulle ou non, et le bit N, qui doit stocker le bit de poids fort, en testant si la valeur est négative ou non.
Lorsqu'un débordement est possible, on utilise une variable *result* qui stocke le résultat normalement attendu (sans débordement) et la fonction **registres_etat** teste s'il dépasse l'intervalle de représentation des 16 bits signés [-32768,32767]. Si c'est le cas, on met le bit C à 1, sinon à 0. Lors de certaines opérations le débordement est impossible, on appelle donc la fonction **registres_etat** mais avec 0 à la place de *result*. Ainsi, 0 ne dépasse jamais l'intervalle et C est toujours mis à 0.

Quelques cas particuliers :

- Dans le cas d'une instruction de saut, si la valeur lue dans le registre src2 est négative en raison de la représentation signée [-32768, 32767], on la transforme en la valeur positive correspondante dans [0, 65535] grâce au complément à deux.
- Dans le cas de la multiplication, pour récupérer seulement les 8 bits de poids faibles, on convertit explicitement les valeurs des 2 opérandes en char avant de les multiplier.
- Dans le cas du décalage bit à bit "`shl`", afin de récupérer le dernier bit qui disparaît pour le stocker dans le registre C procède au décalage en 2 étapes : on décale à gauche de src2-1 bits, puis on stocke le bit de poids fort dans C, puis on décale de 1 bit. Si src2 est négative, on fait de même mais en décalant à droite et en stockant dans C le bit de poids faible. Pour calculer le bit de poids fort, il suffit de déterminer le signe (1 si négatif et 0 si positif ou nul). Et pour le bit de poids faible, on calcule le reste de la division entière par 2.
- Lorsqu'on rencontre "`hlt`" , PC prend la valeur -1. De cette manière, l'adresse n'est plus valide et le programme s'arrête.

Enfin si la valeur contenue dans R[0] a été modifiée, on la remet à 0.
Le programme s'arrête lorsque le test de la boucle while détecte que PC n'est plus une adresse valide.

III. Quelques améliorations possibles…

Pour améliorer ce programme, on pourrait par exemple dans la 2e partie changer la méthode pour passer de l'instruction en mémoire aux opérandes. En effet, au lieu de convertir "à la main" l'instruction en binaire, puis en opérandes explicites, on pourrait procéder plus facilement avec une méthode de décalage bit à bit comme ce que nous avons fait dans la 1ere partie pour assembler les données de toutes les opérandes pour obtenir la valeur finale de l'instruction (puisque multiplier par 2^n revient à décaler de n bits à gauche).
Cela permettrait une meilleure lisibilité du code et sûrement aussi d'améliorer la complexité en temps de l'algorithme.