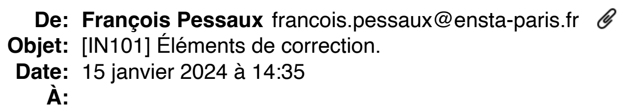

De: **François Pessaux** francois.pessaux@ensta-paris.fr
Objet: [IN101] Éléments de correction.
Date: 15 janvier 2024 à 14:35
À:

Cpu

Q1.1
Notre processeur a 4 registres, donc « cases mémoire » ne pouvant contenir que des entiers. Je regarde ses instructions, les registres s'appellent R0, R1, R2, R3. Donc je vais avoir envie d'accéder rapidement et facilement à n'importe quel registre en fonction de s'il est 0, 1, 2 ou 3. Un tableau sera pratique.

Si à la place je choisis de représenter l'état du processeur par une `struct { int r0 ; int r1 …}` , ça semble plus joli mais je n'aurai aucun moyen d'accéder de manière uniforme, générique, simple à un registre en fonction de son « numéro ». Il faudra que je teste à chaque fois quel est le nom du registre pour choisir explicitement le champ `r0`, `r1` etc. Hum, vraiment pas pratique 😕

Réponse: un tableau de 4 int.

Q1.2
Une instruction comporte 3 éléments : son « nom » et ses deux opérandes. Donc, clairement on a envie de regrouper tout ça. Une `struct` fera l'affaire.

Maintenant, qu'est-ce que je mets dans cette damnée `struct`, comment j'encode ? Je pourrais choisir de mémoriser simplement les chaînes. Mais bon, ça va être pénible car je sais qu'en C il va falloir que je fasse des `malloc()` et des `strcpy()` partout, puis qu'il faudra que je libère tout ça 😱 Et en plus, je ne ferais que repousser le problème de vraiment savoir « quelle est l'instruction » au niveau de la fonction d'exécution qui devra, elle, décortiquer la ou les chaînes. On va s'épargner cette peine.

Essayons de trouver plus compact. Des instructions, il y en a 2, donc je peux les encoder par un entier valant 0 ou 1. Mieux, je peux faire un `enum` ce qui sera plus lisible. Des registres, il y en a 4, donc je peux les encoder par un entier entre 0 et 3. Ce sera la fonction de chargement qui fera une fois pour toute la traduction chaîne « *r-truc* » vers l'indice qui correspond au registre dans mon tableau de Q1.1. Pour terminer, la constante entière du `movi`, ben je vais en faire bêtement un entier.

Réponse : une struct comportant un champ « opcode » étant un enum représentant `movi` ou `add`, 2 champs entiers représentant les registres (le second ne sera pas utilisé dans le cas `movi`, mais on n'a pas vu mieux en C), 1 champ entier représentant la constante dans le cas de `movi` (qui ne sera pas utilisée dans le cas `add`).

Q1.3
On veut décrire à haut niveau les grandes étapes du programme. Il n'est donc pas nécessaire de décrire dès maintenant le fonctionnement détaillé de chaque traitement. D'ailleurs comme, en accord avec les consignes, j'ai lu tout le sujet avant de me jeter à corps perdu dedans, je vois bien que les questions suivantes vont me demander de rentrer dans le détail de chaque sous-problème. Les différentes questions me guident pour faire un programme structuré et modulaire et éviter de tout coller pêle-mêle dans mon `main()`. Et vu que c'est la première question d'algorithmique, c'est normal que l'on s'intéresse au programme dans ses grandes lignes, on découpera chaque étape plus tard.

Donc, globalement je dois vérifier s'il y a bien des arguments, afin de ne pas planter s'ils ne sont pas là, puis charger le programme, puis l'exécuter, puis libérer la mémoire que j'aurai alloué.

Réponse :
```
prog (args) =
  Vérifier présence arguments
  Charger le programme à partir du fichier
  Vérifier succès
  Lancer l'exécution
  Libérer la mémoire
```

Q1.4
Le texte du programme est dans un fichier, il va donc falloir que je connaisse son nom pour aller piocher dedans.

Je vais sortir quoi ? Moralement, une sorte de « liste » d'instructions. Mais les listes, je n'ai pas encore vu, c'est compliqué et par chance, je vois que dans le format du fichier, le nombre d'instructions à lire est indiqué en premier. Je peux donc facilement faire un tableau que je retournerai en résultat. Ah oui, c'est vrai, dans ce satané

C, les tableaux ne mémorisent pas leur taille, donc il va falloir que je la retourne aussi pour que la fonction d'exécution sache plus tard combien il y a d'instructions dans ce tableau.

Alors, je sais qu'en C on ne peut retourner qu'une seule valeur. Or là, j'en ai deux. J'ai deux solutions, soit je fais une `struct` pour grouper le pointeur représentant le tableau et sa taille, soit je retourne la taille via un passage par adresse. Je fais mon choix et je m'en souviendrai lorsque je vais commencer à coder.

Réponse :
Entrée : nom fichier, donc chaîne de caractères.
Sortie : tableau d'instructions + taille de ce tableau.

Q1.5
Il va falloir lire dans un fichier, donc il faut commencer par l'ouvrir et vérifier que ça a marché.

Le format du fichier est sympa car il me dit à l'avance combien il va y avoir d'instructions à lire. Je n'ai qu'à lire la première valeur dans le fichier, sous la forme d'un entier et allouer un tableau d'instructions de cette taille. Et bien sûr vérifier que ça a marché.

Ensuite, il y a 2 cas d'instruction. Comment savoir dans lequel je suis pour savoir combien d'opérandes je dois lire et quelle sera leur forme (constante entière ou nom de registre) ? Il me suffit de lire la première chaîne et en fonction de si c'est `"add"` ou `"movi"`, je lirai ensuite ce qui convient (un entier et une chaîne ou deux chaînes) et construirai l'instruction courante.

Il va falloir transformer un nom de registre en son index qui le représente dans le microprocesseur (tableau). Il n'y a qu'à se faire une fonction qui discrimine sur le nom du registre passé en paramètre et renvoie 0, 1, 2 ou 3 selon ce nom. On pourra l'écrire avec une cascade de `if else` qui comparent le nom avec les 4 chaînes possibles (`strcmp()` sera nécessaire en C pour la comparaison de chaînes).
On pourra sinon exploiter le fait que le second caractère de la chaîne représentant un nom de registre est directement l'indice, en bidouillant un peu dans la chaîne (mais attention à vérifier que la chaîne fait bien 2 caractères et commence par `'r'` — on peut bidouiller, mais ça doit rester correct et robuste quand même).
Chacun sa solution.

Bien entendu, on répétera ce processus autant de fois qu'il y a d'instructions et on logera chaque instruction dans le tableau alloué à cet effet.

S'il l'on n'a pas rencontré d'erreur, on ferme le fichier et on retourne le tableau d'instructions ainsi que sa taille.

Réponse :
```
load (nom fichier) =
  Ouvrir fichier
  Vérifier succès
  long_prg <- lire un entier
  prg <- alllouer mémoire long_prg instructions
  Vérifier succès
  Faire long_prg fois (compteur i)
    s <- Lire une chaîne dans le fichier
    Si s est "add"
      prg[i].opcode <- ADD
      op1 <- lire une chaîne dans le fichier
      prg[i].op1 <- transformer op1 en entier
      op2 <- lire une chaîne dans le fichier
      prg[i].op2 <- transformer op2 en entier
    Sinon si s est "movi"
      prg[i].opcode <- MOVI
      prg[i].op1 <- lire un entier dans le fichier
      op2 <- lire une chaîne dans le fichier
      prg[i].op2 <- transformer op2 en entier
    Sinon fermier fichier, libérer la mémoire, retourner erreur
  Fermer le fichier
 Retourner prg et long_prg
```

Q1.6
Cette fonction a besoin des instructions et de savoir combien il y en a, donc il va lui falloir en entrée le tableau et sa taille. L'énoncé me dit qu'à l'issue de l'exécution, je dois afficher l'état des registres. Bon, ma fonction peut très bien se charger de cet affichage, pas besoin de retourner l'état pour le faire afficher après, ça ne rentre pas en conflit avec l'énoncé.
Sinon, je pourrais lui faire prendre en argument le tableau représentant le processeur, elle bidouillerait dedans et

ce serait mon `main()` qui afficherait.
Bon, faisons simple (mais pas crado) on affiche, donc on ne retourne rien.

Réponse :
Entrée : Tableau d'instructions + taille
Sortie : `void` car affiche.

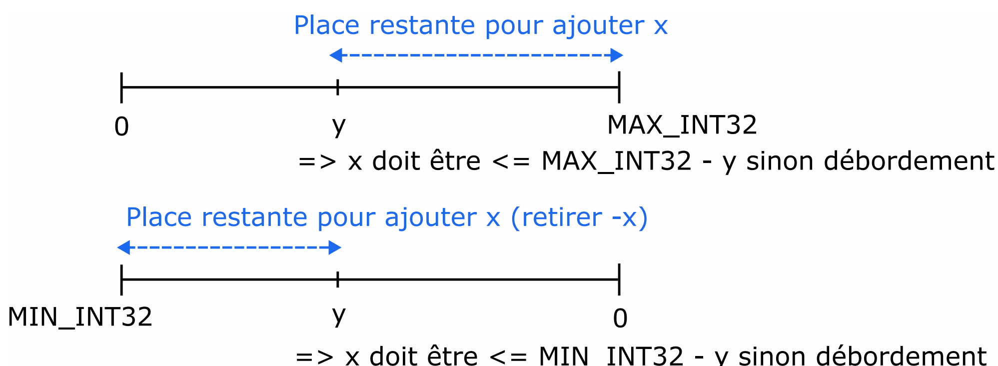
Q1.7
Cette fonction va simplement itérer sur le tableau d'instructions. Pour chacune, elle va regarder quel est le type de l'instruction.
Si c'est un `add` alors c'est entre deux registres et on va mettre à jour le registre destination dans le tableau, à partir des valeurs que ces deux registres ont dans ce même tableau.
Si c'est un `movi` on va directement mettre la valeur constante dans le tableau, à l'indice qui représente le registre utilisé par l'instruction.
À la fin de la boucle, on affiche les 4 valeurs du tableau.

Réponse :
```
Boucle i 0 -> longueur programme (exclue)
  Si prg[i].opcode = ADD
    cpu.regs[prg[i].op2] <- cpu.regs[prg[i].op1] + cpu.regs[prg[i].op2]
  Sinon si prg[i].opcode = MOVI
    cpu.regs[prg[i].op2] <- prg[i].op1
  Sinon erreur
Afficher le contenu des 4 registres
```

Q1.9
La contrainte importante est qu'il n'est pas possible d'effectuer des opérations sur plus de 32 bits. Donc, la solution d'utiliser un `long int` n'est pas viable et donc non pertinente. On cherche donc comment détecter que $r = x + y$ **a ou va** provoquer un débordement.

Première solution, incorrecte, mais avec de l'idée :
Si *x* et *y* sont de signes différents, il n'y aura pas de débordement. S'ils sont de signes identiques, qu'après l'addition le résultat est de signe différent, alors il y a eu débordement. On vérifie donc après-coup.
Cette solution ne marche en fait pas car la norme de C indique qu'un débordement arithmétique provoque un résultat non spécifié. Dit autrement, le résultat n'est pas défini et l'on ne peut rien en déduire, ça peur donner n'importe quoi (et avec des options d'optimisation, le compilateur ne s'en prive pas 😱). Certes, il faut connaître ce détail du langage C.

Seconde solution, correcte :
L'idée est de vérifier, **a priori**, s'il va y avoir débordement. Une fois encore, si *x* et *y* sont de signes différents, il n'y aura pas de débordement.
Si *y* est positif alors il y aura débordement s'il n'y a pas « assez de place » entre `MAX_INT32` et *y* pour ajouter *x*. Donc, si `(y > 0 && x > MAX_INT32 - y)` : débordement « par le haut ».
Si *y* est négatif alors il y aura débordement s'il n'y a pas « assez de place » entre `MIN_INT32` et *y* pour ajouter *x* (qui est négatif, donc c'est bien une soustraction). Donc, si `(y < 0 && x < MIN_INT32 - y)` : débordement « par le bas ».
Allez, un schéma valant mieux qu'un long discours, le PDF attaché montre ça visuellement.



IP

Il faut que j'extraie 4 champs séparés par des points, puis que je vérifie à chaque fois qu'ils représentent bien une valeur entière entre 0 et 255 inclus.

Est-ce que je commence par vérifier que la longueur de la chaîne est bien inférieure à la longueur maximale d'une IP correcte (i.e xxx.yyy.zzz.ttt —> 1(caractères) ? Bof, non car de toute façon il faudra que je repasse encore après sur la chaîne pour récupérer les entiers. Donc ça va me compliquer les choses inutilement.

Est-ce que je commence par vérifier s'il y a le bon nombre de points ? Bof, non, pareil qu'au-dessus, il faudra que je repasse de toute façon sur ma chaîne.

Ah tiens, je vois au passage dans les tests que les champs ne sont pas forcément constitués de 3 chiffres, il peut y en avoir moins. Donc ça me dissuade de toute idée s'appuyant sur « 3 chiffres à chaque fois » : il va falloir être plus souple.

Comment je ferais à la main ?
Ben, je lirais caractère par caractère. Tant que je ne tombe pas sur un point, je retiens les caractères (et je vérifie qu'il sont bien entre `'0'` et `'9'`). Et quand je tombe sur un point, j'ai isolé un champ. Je peux transformer ces caractères en un entier pour vérifier qu'ils correspondent bien à un nombre entre 0 et 255 inclus.
Ah ben tiens, plutôt que de mémoriser les caractères pour ensuite en faire un entier (genre, avec `atoi()`), je pourrais peut-être être plus efficace en calculant au fur et à mesure la valeur entière que valent les chiffres que j'ai trouvés. Et pour faire ça, j'utilise le schéma habituel à coup de ** 10 + chiffre*.

Bon, c'est bien tout ça, mais là, je n'ai trouvé qu'un champ et il m'en faut 4. Ben en fait, il me suffit de faire la même chose sur le reste de la chaîne. Donc, faisons les choses simplement en découpant le problème : on va faire une fonction qui cherche à isoler un champ et qui retourne sa valeur si elle en trouve un. Et puis, ben on l'appellera 4 fois pour qu'elle aille bouffer la suite de la chaîne.
Ah oui, mais « *la suite* », ça veut dire qu'il faut que je sache où elle s'est arrêtée la fois d'avant. Bon, ben elle n'aura qu'à retourner aussi cette position. Et par conséquent, elle prendra aussi en argument la position à partir de laquelle commencer à rechercher, au lieu de toujours commencer depuis le début de la chaîne.

Au fait, quand la chaîne est mal formée et que l'on échoue à trouver un champ, on retourne quoi ? Bof, on n'a qu'à retourner -1 puisqu'un champ c'est entre 0 et 255, on verra clairement que c'est un cas d'erreur.

Bon, on récapitule.

On commence par la fonction principale. Elle va appeler 4 fois la fonction de découpe en vérifiant à chaque fois si la détection d'un champ a été un succès. En cas de succès, la fonction principale va débuter sa prochaine recherche à l'indice retourné au coup précédent par la fonction d'analyse. Sinon elle émet le message d'erreur en fonction du champ fautif et termine.

```
prog (args) =
  Vérifier présence d'un seul argument, arg
  start <- 0
  Si analyse de arg depuis start est incorrecte alors message pour champ 1 et fin
  champ1 <- valeur du champ retournée
  start <- nouvel indice de début retourné
  Si analyse de arg depuis start est incorrecte alors message pour champ 2 et fin
  Etc, idem pour champs 3 et 4
  Si on est arrivé à la fin de la chaîne alors afficher les 4 valeurs de champs
  Sinon afficher un message d'erreur
```

Ah oui, petite subtilité, une fois que l'on a bien trouvé 4 champs, il faut penser à s'assurer qu'il ne reste pas des caractères superflus à la suite. Sinon je vais dire que "`1.1.1.1.1.1.1.1.1.1.1.1`" est une IP correcte.

Maintenant, notre fonction de découpe, on a dit qu'elle va avancer caractère par caractère dans la chaîne depuis la position indiquée initialement. Tant qu'elle n'est pas arrivée à la fin de la chaîne, elle inspecte le caractère courant `c`. Si `c` est un chiffre, elle récupère sa « valeur » et s'en sert pour construire la valeur du champ (méthode de Horner). Sinon, si `c` est un point, il marque la fin du champ en suspens et la fonction doit retourner sa valeur et l'indice courant *plus 1* pour signaler où continuer la recherche. Et dans tous les autres cas c'est une erreur.

Arrivé en fin de chaîne (on tombe sur `'\0'`), on a sans doute une valeur de champ en suspens qu'il faut retourner (c'est toujours le cas si la chaîne représente une IP correcte). C'est le cas si l'indice courant est différent de l'indice de départ initialement transmis.

À noter qu'à chaque fois, avant de retourner la valeur du champ, il faut s'assurer qu'elle est bien <= 255, sinon il faut retourner une erreur.

```
cut (str, start) =
  i <- start
  accu <- 0
  Tant que str[i] <> '\0'
```

```
    Si str[i] est un chiffre alors
      c <- récupérer sa valeur
      accu <- accu * 10 + c
      i <- i + 1
    Sinon si str[i] = '.' et i > start alors
      Si accu < 256 alors
        Retourner accu, i + 1, ok
      Sinon retourner erreur
    Sinon retourner erreur
  Si i <> start et accu < 256 retourner accu, i, ok
  Sinon retourner erreur
```

Petite subtilité… Eh oui, si la chaîne commence par un point, genre `".1.1.1"`, il ne faut pas que je considère que je viens de réussir à isoler un champ ! Et plus généralement, si la recherche commence sur un point le problème est le même, comme pour `"1..1.1"`. Donc, je n'ai vraiment terminé un champ que si `i` est strictement supérieur à l'endroit où j'ai commencé ma recherche (i.e. `start`).