Compte-rendu de développement de Sim

Note : Ce compte-rendu s'intéresse aux aspects techniques du développement du programme Sim. Pour les informations pratiques (installation et utilisation), se référer au fichier `README.txt`.

Ici seront décrit l'organisation des fichiers du programme ainsi que les choix techniques faits lors de son développement.

Organisation du projet

Arborescence

Le fichier principal (`main.ml`) gère les paramètres donnés à la fonction (fichiers d'entrée, de sortie, de ROM, etc. cf `README.txt` pour plus d'informations) à l'aide du module `Arg`. Il initialise ensuite la simulation en appelant les différents parseurs (celui des netlists et celui des fichiers .sim) puis en appelant le scheduler pour ordonner le fichier. Une fois l'initialisation terminée, il lance cycle par cycle la simulation et enregistre son résultat dans le fichier de sortie.

L'export de la sortie vers le fichier `output.sim` n'a pas été séparé du fichier principal car ne prend pas beaucoup de place mais il aurait peut-être été plus rigoureux de le gérer dans un module.

Le module `Netlist` appelé pour l'ouverture et l'analyse syntaxique du fichier netlist est celui fourni par Tim Bourke.

Le module `Scheduler` (ainsi que le module `Graph` dont il dépend) sont ceux écrit lors du TP.

Le fichier `core.ml` gère le cœur du simulateur, la fonction `tic`, dont le rôle est d'exécuter un cycle du circuit sur un jeu d'entrées donné.

Le lexeur `sim_lexer.mll` est un analyseur syntaxique de fichier `.sim` (dont la syntaxe est décrite dans le fichier `README.txt`). J'ai choisi une syntaxe suffisamment simple pour éviter d'utiliser un parseur et permettre une rédaction rapide mais contenant tout de même des commentaires afin d'être clair et notamment de pouvoir indiquer dans le fichier à quoi correspondent les colonnes.

Compilation

La compilation se fait à l'aide de la commande `make`. Je n'ai pas utilisé `ocamlbuild` car je maîtrise moins son fonctionnement et que l'apprentissage de la syntaxe des `Makefile` est de toute façon nécessaire pour de nombreuses situations.

Pour ne conserver que les fichiers de développement, utiliser la commande `make clean`.

Fonctionnement et implémentation

Fonctionnement général

La simulation s'effectue cycle par cycle. Chaque cycle nécessite de connaître :

- La valeur des entrées à ce cycle
- La valeur au cycle précédent des entrées des registres
- L'état courant de la RAM
- La ROM

La valeur des entrées est directement donnée par l'analyse du fichier d'input. Pour avoir la valeur des entrées des registres, on passe en argument de la fonction `tic` l'ancienne valeur de toutes les variables, ce qui évite de faire un tri inutile.

RAM et ROM sont passées en paramètres sous la forme d'une table de hachage. J'ai choisi pour la ROM une table de hachage car les valeurs sont lues bien plus souvent qu'elles sont écrite et c'est donc plus efficace. J'ai alors fait le même choix pour la RAM afin de pouvoir utiliser les même fonctions. De plus, d'un point de vue sémantique ce choix est plus cohérent : une table de hachage est modifiée par effet de bord à l'instar de la RAM dans une circuit logique.

Au fur et à mesure de la simulation d'un cycle, on lit les différentes équations dans l'ordre dans lequel `Scheduler` les a placées. Si ce dernier a correctement remplit sa fonction, lorsque l'on lit une équation, toutes les variables du membre de droite ont déjà été évaluées pour ce cycle et on peut donc calculer la variable du membre de gauche. Ces valeurs sont ajoutées à une table d'association étiquetée par des identifiants (type `Env` donné dans le module `Netlist_ast`).

Les erreurs d'identifiant non évalué ne sont pas prises en charge dans cette partie car sont déjà gérées par le scheduler.

Gestion des registres

Dans le graphe de dépendance, il faut bien ajouter les registres comme les autres (pour définir les variables au bon moment) mais ne pas ajouter de liens de dépendance (`reg n` ne nécessite pas de connaître `n`).

Pour gérer les registres, l'ancien environnement est passé en argument de la commande `tic`. Si `n` n'est pas trouvé dans l'ancien environnement (ce qui arrive à la première itération, lorsque celui-ci est encore vide, une valeur par défaut est appliquée (`VBit false` pour un simple bit et `VBitArray [|false; ...; false|]` pour un groupe de bits).

Gestion de la mémoire

Pour la lecture, on recherche l'adresse dans la table de hachage correspondante (`rom` ou `ram`). Si la valeur n'est pas trouvée, on retourne tous les bits à 0. Ce choix de retourner des bits nuls est totalement arbitraire. Pour être plus proche de la réalité, j'aurais pu décider de choisir la valeur de façon aléatoire, mais un tel choix aurait grandement compliqué le débogage. En effet, un même programme peut alors retourner des valeurs différentes selon les exécutions et on ne sait donc pas toujours d'où elle viennent, ni même si elles surviennent.

L'écriture (qui n'a lieu que dans le cas de la RAM) se fait systématiquement en fin de cycle, une fois que toutes les autres valeurs ont été calculées. On évite ici d'une part les cycles (si la valeur à écrire dépend de la valeur lue en mémoire par la même équation) et les problèmes de chronologie (lorsqu'une équation lit la mémoire qui vient d'être réécrite.

Lorsque la valeur stockée dans la mémoire ou envoyée en entrée ne correspond pas à la valeur de `wordSize` indiquée, on tronque ou complète arbitrairement (on suppose les poids faibles à droite) les entrées et sorties de la mémoire. Ces situations ne sont censées arriver en pratique que lorsque la valeur de la mémoire n'est pas initialisée si on conserve dans tout le circuit la même `wordSize`.

Fichiers d'entrée et de sortie

Les valeurs doivent toujours bien correspondre au nombre d'entrées du circuit ainsi qu'à leur type. Une erreur de type génère seulement un avertissement mais conduit généralement à une erreur par la suite (les opérations sur les nappes ne peuvent être appliquées aux bits et inversement). Une erreur sur le nombre d'entrée arrête immédiatement la simulation.

Ce choix de gérer les entrées et sorties dans des fichiers a été fait en perspective de l'organisation modulaire du projet. En effet, ses différentes composantes (Simulateur, Microprocesseur, Assembleur, Programme d'horloge) devront pouvoir communiquer entre elles et pour cela un fichier est plus simple à gérer que les sortie et entrée standards. Ainsi, l'assembleur (programme) convertira le fichier .asm en fichier d'entrée de `sim`, en .sim.

Le fichier `output.sim` ne possède pas de commentaire de légende indiquant la variable à laquelle correspondent les colonnes car cela nécessitait d'aligner les colonnes, ce que je n'ai pas pris le temps de faire. C'est cependant une idée relativement simple d'amélioration.

La possibilité d'exporter la netlist ordonnée `sch.net` est essentiellement un outil de débogage, c'est pourquoi je n'ai pas pris la peine d'ajouter la possibilité de modifier le nom de ce fichier.

Difficultés rencontrées

Fichiers .net

Une ancienne version du compilateur `mjc` semblait préférer `SELECT 1 foo` à `SLICE 1 1 foo`. J'en ai alors conclu que `SELECT` pouvait se faire sur un bit simple et pas uniquement sur un groupe de bits et que `[|b|]` et `b` devaient être identifiés malgré le manque de rigueur que cela impliquait.

Cette même ancienne version avait créé une boucle dans le fichier ram.net : Ligne 11 et 14 dans ram.net : il fallait remplacer `_l_14` par `o` pour éviter les boucles. Un erreur dans mon scheduler l'empêchait de plus de détecter les boucles de dépendances (ex: `_l_14` -> `_l_7_21` -> `_l_9_19`).

Les fichiers .net donnés ne correspondaient en fait pas aux compilés des fichiers .mj et il suffisant donc de les recompiler.

J'ai également remplacé `and` par `or` dans `or_n<n>`, d'une part pour que le circuit corresponde à son

nom et également parce que, initialisant la mémoire à 0, le circuit résultant était inintéressant (on ne pouvait enregistrer d'autres valeurs que 0).

Fichiers .sim

La gestion de l'ordre des valeurs a été une source d'erreur lors de la lecture des fichiers et l'enregistrement des données dans les listes à envoyer à la fonction `tic`. La présence des `List.rev` qui en découle peut peut-être être optimisée.

Un autre erreur a été l'oubli de réinitialisation des références de listes lors de l'import des fichiers .sim qui donnait les mêmes valeurs d'entrée à chaque itération.

Un dernier problème a été la gestion des commentaires sur une ligne complète. En effet, ils ne fonctionnent pas exactement comme les commentaires en fin de ligne puisqu'ils empêchent en plus la ligne de compter comme une ligne d'entrée. Une solution plus simple aurait été d'utiliser un caractère explicite pour passer au cycle suivant mais cela aurait surchargé le fichier. J'ai donc fait du bricolage à l'aide de la variable `!empty` qui regarde si une ligne est vide ou non lorsque le commentaire commence. Il était important que la ligne `#blabla` soit distincte de la ligne `#blabla` pour pouvoir gérer le cas des fichiers sans entrées comme `clock_div.net` par exemple.

Gestion de la mémoire

Les principales difficultés ont déjà été décrites. On peut citer par exemple la définition du graphe de dépendance mais aussi les problèmes de chronologie : on ne doit pas écrire la nouvelle valeur de la mémoire avant que toutes les équation qui lisent cette adresse aient été exécutées.