Systèmes digitaux

Ecole Normale Supérieure - Département Informatique - L3 S1

Groupe : Paul-Nicolas MADELAINE, Axel KUGELMANN, Antonin REITZ

1 Organisation du projet

1.1 Structure globale

Le projet contient cinq exécutables. Dans l'ordre, voici le nom du fichier exécutable, le fichier Haskell qui le produit et sa description :

- `simulator`, `Simulator.hs`, un simulateur de netlists ;
- `generator`, `Cpu.hs`, le programme qui génère la netlist du CPU ;
- `optimizer`, `Optimizer.hs`, un optimisateur de netliists ;
- `compiler`, `Compiler.hs`, le compilateur de netlisits vers C ;
- `assembly`, `Assembly.hs`, le compilateur d'assembleur MIPS vers une ROM.

Il y a quelques changements de convention pour la netlist, d'un point de vue syntaxique :

- `slice` i j x contient les bits de x d'indices $>= i$ et $< j$;
- `ram` et `rom` ne prennent plus d'entiers en paramètre en ce qui concerne la taille de la RAM et la taille des mots, ceux-ci sont hardcodés.

L'arborescence du projet est la suivante :

- `src`
  - `Assembly`, qui contient le code du compilateur MIPS vers un fichier qui représente une ROM qui sera chargé en ROM et exécuté par le processeur
    - `Ast.hs`
    - `Compiler.hs`
    - `Parser.hs`
  - `Assembly.hs`
  - `Compiler.hs`
  - `Cpu`, le code Haskell qui va générer la netlist du CPU
    - `Adder.hs`
    - `Alu.hs`
    - `Branch.hs`
    - `Control.hs`, multiplexage à partir de l'opcode
    - `Instr.hs`
    - `Memory.hs`
    - `Misc.hs`
    - `Mult.hs`, multiplication et division sur plusieurs cycles
    - `Nalu.hs`
  - `Cpu.hs`
  - `Netlist`
    - `Ast.hs`
    - `Compiler.hs`, le compilateur depuis la représentation Haskell d'une Netlist vers du C
    - `Graph.hs`
    - `Jazz.hs`, le code de la monade Jazz qui permet de générer des netlists
    - `Opt.hs`, le code de l'optimisateur de netlists
    - `Parser.hs`
    - `Scheduler.hs`

    - `Show.hs`
    - `Simulator.hs`
    - `Typer.hs`
  - `Optimizer.hs`
  - `Simulator.hs`

1.2 Compilation

Lancez la commande `make`. Elle génère les différents executables du projet, notamment `clock_exec` et `fast_clock_exec` qui sont les deux horloges.

1.3 Jazz

Jazz est notre monade pour générer des netlist, elle remplace MiniJazz. On implémente deux types importants : Bit et Wire, qui représentent les fils et les nappes de fils. Ainsi, on peut écrire le type du combinateur qui génère une porte AND :
$(/\backslash) :: \text{Bit} \to \text{Bit} \to \text{Jazz Bit}$
On a en fait enrichi ce modèle en créant des type class Bt et Wr associées aux deux types :
$(/\backslash) :: (\text{Bt } a, \text{Bt } b) \implies a \to b \to \text{Jazz Bit}$
Ce qui permet d'écrire des choses telles que :
$x \leftarrow a\ /\backslash (b\ /\backslash c)$

Le module Cpu.Control illustre bien la puissance de Jazz : on implémente une structure qui contient un champ pour chaque instruction cpu, à chacune de ces insrtructions on associe un fil et on écrit une fonction opcode_mux qui à partir de cette structure construit la netlist d'un multiplexer, qui sélectionnera un fil à partir de l'instruction décodée. On peut ainsi spécifier très simplement le comportement du CPU. Il est alors très facile d'ajouter, de retirer et de modifier des instructions au CPU.

1.4 CPU

Nous avons implémenté un sous-ensemble de MIPS. Une liste des instructions se trouve dans Cpu.Control. Nous avons notamment implémenté :

- les opérations binaires "élémentaires" (sauf XOR, qui n'était pas dans le standard) ;
- l'addition et la soustraction ;
- les jumps ;
- des branchements conditionnels ;
- la division et la multiplication sur plusieurs cycles.

Elles ne sont cependant pas encore toutes implémentées, ou tout du moins elles ne vérifient pas encore toutes la sémantique de MIPS. Les instructions de lecture/écriture en RAM notamment ont un comportement assez mal défini pour le moment. De même, la division et la multiplication ne fonctionnent que pour des entiers non signés.

2 Horloges

L'horloge est divisée en plusieurs parties.

1. Tout d'abord, on charge le timestamp et on convertit les années pour commencer à compter le temps (approximativement) au bon moment.

   Cette fonction est realisée par le code suivant (le timestamp est chargé en RAM à l'adresse 28) :

```
  lw $at,$zero,28
  li $t0,0
  li $t1,0
  li $t2,0
  li $t3,1
  li $t4,3
  li $t5,1
  li $t6,1970

  lui $ra, 1926
  addiu $ra,$ra,8064
  divu $at,$ra
  mflo $a1
  sll $ra,$a1,2
  addu $t6,$t6,$ra

  mfhi $s0
  lui $ra, 481
  addiu $ra,$ra,13184
  divu $s0,$ra
  mflo $a2
  addu $t6,$t6,$a2

  mfhi $ra
  subu $fp,$at,$ra


  li $ra,5
  multu $ra,$a1
  mflo $ra
  addu $t4,$t4,$ra
  addu $t4,$t4,$a2

  li $ra,7
  divu $t4,$ra
  mfhi $t4

  j init
```

où les instructions

```
  mfhi $ra
  subu $fp,$at,$ra
```

chargent dans fp le nombre de secondes précalculées

2. Puis, on effectue le calcul du temps à proprement parler dans le code suivant :

```
second:
  lw $at,$zero,28
  beq $at,$fp,second

  addiu $fp,$fp,1
```

```
addiu $t0,$t0,1
beq $t0,$a0,minute
sw $t0,$zero,0
j second
```

On fait appel à minute lorsque 60 seconde sont atteintes ($a0 contient 60), puis à heure lorsque 60 minutes sont atteintes, et ainsi de suite.

Les instructions

```
lw $at,$zero,28
beq $at,$fp,second
```

empêchent le processus d'incrémenter les secondes tant que le timestamp ne change pas (at contient le nouveau timestamp, et fp le timestamp calculé). Elles permettent cependant de rattraper l'heure actuelle dans le processus en temps réel.

L'horloge rapide consiste à enlever ces 2 instructions.

3. Enfin, les données importantes sont sauvegardées en RAM seulement si elles sont modifiées, ce qui permet d'accélérer l'horloge.

3 Compilateur

Le but d'avoir un compilateur du format netlist vers le C est de bénéficier de toute la rapidité du C, notamment en compilant ensuite avec GCC avec le degré d'optimisation maximal (Ofast).

On pourrait d'ailleurs davantage parler de « traducteur » que de compilateur de netlists vers du C, la compilation étant réalisée d'un langage de bas niveau à un autre langage de bas niveau, assez semblables.

Deux classes d'optimisations différentes sont réalisées :

— des optimisations liées à l'efficacité du code à exécuter ;
— des optimisations liées à l'affichage de ce que produit ce code.

3.1 Optimisations du code

3.1.1 Optimisations réalisées

Les constantes sont remplacées autant que possible, il s'agit d'un simple *inlining*. Les opérations entre constantes ne sont pas effectuées en Haskell pour que le résultat soit remplacé dans le fichier C généré, ce qui pourrait être une source supplémentaire d'optimisation.

Toutes les valeurs manipulables et accessibles par la netlist sont encodées par des entiers de 64 bits. Ce qui permet ceci et ce qui permet les optimisations détaillées plus bas est que les opérations de sélection (`select` ou `slice`) ont déjà été vérifiées pour que ne soit accessible que ce qui doit l'être.

Toutes les opérations binaires (ainsi que le `mux` sont traduites vers les opérations binaires équivalentes. Trois cas sont plus particuliers : les `select`, `slice` et `concat` :

— l'opération `select` i a consiste à effectuer un shift vers la droite de i, ce qui est correct d'après ce qui a été dit dans le paragraphe précédent ;
— l'opération `slice` i j a consiste à effectuer également un shift vers la droite de i, ce qui est correct par les mêmes arguments ;
— l'opération de concaténation est réalisée en utilisant un *bitmask* et un shift vers la gauche adaptés.

3.1.2 D'autres pistes

Parmi les optimisations non réalisées, il y a le fait de choisir une représentation mémoire adaptée à la taille du nombre : stocker 1 bit sur 64 bits n'est pas le plus efficace qui soit.
Néanmoins, une telle optimisation amène à beaucoup d'autres, notamment à la concaténation des valeurs de faible longueur pour effectuer plusieurs instructions sur un seul cycle (on peut imaginer l'exécution de 64 instructions de même nature concernant des booléens effectuées en un seul cycle).

On peut également imaginer, encore à un tout autre niveau de complexité, l'exécution en parallèle de certaines instructions. En somme, on peut imaginer d'implémenter toutes les optimisations « classiques » présentes sur les processeurs modernes.

3.2 Optimisations de l'affichage

L'optimisation décrite n'est utile que dans le cas de l'horloge dont la vitesse de défilement doit être maximale. Afficher une valeur à chaque seconde simulée est coûteux. La programmation parallèle a donc été utilisée, un *thread* étant chargé de la simulation de l'horloge, un autre *thread* étant chargé de l'afficher à intervalles réguliers. La période choisie empiriquement est de 20 ms, correspondant à une fréquence de 50 Hz, ce qui paraît raisonnable compte tenu de la fréquence de rafraîchissement de la plupart des moniteurs actuellement sur le marché.

3.3 Limites

Actuellement, la compilation n'est pas très générale, au sens où le format de l'affichage est fixé - calqué sur celui d'une horloge - et ne dépend pas d'un quelconque paramètre. Ainsi, on pourrait imaginer que les netlists spécifient la forme de l'affichage (au-delà de ce qui doit être affiché ou non), ou que des options adaptées à la compilation rendent le tout flexible.

4 Répartition des tâches

Chacun s'est essentiellement concentré sur une tâche particulière.

— Paul-Nicolas Madelaine a beaucoup travaillé sur la réalisation d'un bon cadre en Haskell, permettant de travailler ensuite beaucoup plus efficacement ;
— Axel Kugelmann a notamment réalisé l'horloge, la version rapide celle-ci étant bien optimisée ;
— Antonin Reitz a réalisé le compilateur et les différentes optimisations en C pour le calcul et l'affichage.

5 Résultats

[Les résultats en eux-mêmes sont susceptibles d'être modifiées compte tenu de l'actuelle mise à jour des fichiers horloge, lors de la démonstration tout était fonctionnel, mais suite à des modifications ultérieures :

— l'horloge à vitesse de défilement normale fonctionne mais ne charge plus le timestamp actuel ;
— l'horloge à vitesse de défilement maximale fonctionne mais moins vite que lors de la présentation - environ 300 000 secondes simulées par seconde.

Les deux sont liées à la volonté d'avoir des horloges respectant toutes les vitesses de défilement. Le nombre d'instructions exécutées par seconde reste de l'ordre du MHz.]

Au début de ce projet, aucun d'entre nous n'était à l'aise en Haskell, malgré de bonnes bases en OCaml, autre langage fonctionnel mais impur. (La pureté d'Haskell cause parfois quelques difficultés avec la monade IO pour afficher des valeurs, notamment lors du déboguage.) Nous sommes désormais tous les trois plutôt à l'aise.

De plus, la structure de ce projet est telle que refaire ce projet avec une autre architecture ne nécessiterait que très peu de modifications, et celles-ci pourraient être réalisées relativement facilement. Simuler le fonctionnement entier d'une Game Boy par exemple constitue un beau projet, la fréquence du CPU d'une Game Boy étant d'environ 4 MHz (quelques centaines de milliers d'instructions par seconde étant effectivement exécutées, beaucoup d'instructions utilisant plusieurs cycles), celle du rafraîchissement du moniteur étant d'environ 60 Hz.