Système digital : Rapport sur le microprocesseur

Josselin Giet
Jian Qian
Elie Studnia
Lucas Willems

17 janvier 2017

1 Mode d'emploi de la montre

En se placant de le fichier principal, on peut générer la net-list du microprocessur ainsi que sa version compilée avec la commande :

```
make
```

De plus, pour lancer la montre, on tape la commande :

```
python3 clock/main.py
```

Par défaut, la montre fonctionne en mode asynchrone, pour lancer le mode synchrone, il faut saisir :

```
python3 micro/main.py -s
```

Pour compiler un fichier quelconque écrit en assembleur, on peut taper la commande :

```
./assembly.byte <nom du fichier>.s
./simulator.byte micro/main.net -rom <nom du fichier>.byte -rom <nom de la ram>.byte
```

Enfin, pour supprimer les fichier générés par le `make`, il faut taper `make clean`.

2 Description du code

2.1 Répertoire simulator

Le dossier comprend le simulateur de netlist compilé `main.byte`, un fichier `README.md` (ou `README.html` pour une lecture plus claire) expliquant comment utiliser `main.byte`, un dossier `test` contenant une liste de tests (fichiers `.net`), et une liste de fichiers sources OCaml :

- `graph.ml` : fournit une structure de graphe et implémente la détection de cycles et un tri topologique
- `main.ml` : gère tous les composants du simulateur i.e. le `scheduler` et le `simulator` et propose une interface en ligne de commande
- `netlist.ml` : permet de passer d'une netlist en syntaxe concrète en une netlist en syntaxe abstraite et inversement
- `netlist_ast.ml` : fournit la syntaxe abstraite d'une netlist
- `netlist_lexer.ml` et `netlist_parser.ml` : permettent de lexer et de parser une netlist i.e. de passer d'une netlist en syntaxe concrète en une netlist en syntaxe abstraite
- `netlist_printer.ml` : permet de printer une netlist i.e. de passer d'une netlist en syntaxe abstraite en une netlist en syntaxe concrète
- `scheduler.ml` : permet de détecter les cycles combinatoires présents dans une netlist et d'ordonnancer les instructions

— `simulator.ml` : permet de simuler une netlist en syntaxe abstraite. Le simulateur prend en paramètre `p` (une netlist en syntaxe abstraite), `number_steps` (le nombre d'étape dans la simulation) et `rom` (la ROM i.e. un tableau de bouléens) et fonctionne comme suit :

1. Initialisation d'une variable `e` qui va stocker, tout au long de la simulation, l'environnement de la netlist i.e. les valeurs des variables
2. Exécution `number_steps` fois du cycle suivant :
   (a) La saisie des entrées (avec vérification de leur correction)
   (b) L'interprétation des équations à partir des entrées saisies, des registres, de la RAM et de la ROM. Lorsqu'une instruction RAM est recontrée, la lecture se fait à ce moment-là et l'écriture se fait lorsque toutes les instructions ont été parcourues.
   (c) L'affichage des sorties

2.2 Répertoire assembly

Ce réprtoire est le compilateur de notre code assembleur. Une fois compilé le fichier `main.ml` renommé `asembly.byte` dans le `make`, prend en entré un code assembleur (extension `".s"`) est génère un code executable pour notre simulateur qui est donnée en entrée comme une rom.

2.3 Répertoire micro

Ce fichier contient l'ALU de notre microprocesseur dans le fichier `main.mj`. Chaque autre fichier `.mj` contient le code MiniJazz pour une insruction de notre microprocesseur (*cf.* ci-dessous). Ces fichiers sont automatiquement ajoutés dans le fichier `main.mj` grâce au fichier `main.mmj`.

2.4 Répertoire clock

Ce répertoire contient le code assembleure de notre montre `clock-without-mod.s` ainsi que le fichier gérant l'affichage de la montre : `main.py`.

3 Généralités sur le simulateur

Le simulateur reprend celui écrit par Lucas Willems. On notera toutefois certaines différences dans l'affichage des outputs et dans la réalisation du tri topologioque optimisé pour les simulations sans inputs.

4 Généralités sur le processeur

4.1 Architecture du microprocesseur

Notre microprocesseur a une architecture 32 bits. Il possède en tout 16 registres chacun codé sur 4 bits (notés de 0 à 15 en code assembleur). On utilise aussi 9 instructions codées en tout sur 4 bits.

4.2 Jeu d'instructions du microprocesseur

Dans cette section les instructions sont celles d'un microprocesseur Mips, dont nous avons changé le format (*cf.* Fig. 1).

Remarque : `and` et `or` sont des opérations bits-par-bits

Remarque : l'`opcode` d'une instruction est donnée par le numéro du fichier `.mj` qui la code dans le répertoire micro.

Nom	Format	Syntaxe	fonction
`li`	L	`li r, immediate`	Remplace la valeur dans le registre `r` par `immediate`
`addiu`	I	`addiu rd, rs, immediate`	Met dans le registre `rd` la valeur R[`rs`]+`immediate`
`bne`	I	`bne rs, rt, offset`	Si R[`rs`] != R[`rt`], alors on se déplace de offset vers le bas
`sltu`	R	`sltu rd, rs, rt`	Si R[`rs`]<R[`rt`], alors R[`rd`] prend la valeur 1, 0 sinon
`srlv`	R	`srlv rd, rs, rt`	décale de R[`rt`] la valeur de R[`rs`] et met le tout dans `rd`
`and`	R	`and rd, rs, rt`	Met dans `rd`, R[`rs`] `and` R[`rt`]
`or`	R	`or rd, rs, rt`	Met dans `rd`, R[`rs`] `or` R[`rt`]
`j`	J	`j adre`	Va à l'adresse `adre`

Figure 1 – Liste des instructions

bits	31 28	27 24	23 20	19 16	15 0
L	opcode	`0`	`0`	`r`	`immediate`
R	opcode	`rs`	`rt`	`rd`	`0`
I	opcode	`rs`	`0`	`rd`	`immediate`
J	opcode	`0`	`0`	`0`	`immediate`

Figure 2 – Format des instructions

4.3 Gestion de la mémoire

Le processeur ayant une architecture Mips, il ne pourra gérer que la Ram dont on utilise que l'écriture. Les instructions suivantes sont donc pour un Ram. Mais la lecture des instructions se fait comme une lecture en Rom dans le langage MiniJazz et dans le simulateur.

Nom	Format	Syntaxe	fonction
`lw`	I	`lw rs, rt, adre`	R[`rs`] = RAM[R[`rt`] + `adre`]

Figure 3 – Instruction pour la gestion de mémoire

4.4 Execution d'une instruction

Quand on lit une instruction dans la Rom, le microproicesseur simule toutes les instructions, puis séléctionne la bonne nappe de fils en sortie par une série de `Mux` sur l'`opcode` pour avoir l'adresse suivante et les valeurs de registre.