# Rapport final du projet de microprocesseur

Florian Bourse, Joris Giovannangeli, Théo Zimmermann

Janvier 2012

# 1 Langage de description de circuits

## 1.1 Les spécifications

### 1.1.1 Principes généraux

Le langage que nous avons créé est un langage hiérarchisé. Il permet de définir des portes logiques complexes et de les combiner entre elles dans la définition d'autres portes. Le circuit lui-même est vu comme une porte.

Le programme d'un circuit consiste en la description successive des portes.

Le circuit est donné par une porte de nom "Start".

### 1.1.2 Définition d'une porte

Une porte est la donnée de son nom, de la liste de ses variables d'entrée, d'un ensemble d'instructions décrivant la manière dont on combine les variables, et de la liste de ses variables de sorties.

**Contenu** Sont disponibles les portes logiques élémentaires *not*, *and*, *or*, *xor* et *mux*, les primitives *reg* pour les registres (1 bit) et *lw* pour lire dans la ROM, les constantes *true* et *false*.

**Utilisation** Il est possible d'assigner des expressions complexes à des variables ou dans des tableaux.

Les variables sont définies à l'occasion des assignations.

En revanche, les tableaux doivent être déclarés avant d'être utilisés.

Les variables définies à l'intérieur d'une porte ainsi que les variables d'entrée sont utilisables dans les expressions qui se trouvent dans cette porte. L'ordre n'importe pas. Cependant, pour que le circuit compile, il convient de ne pas créer de boucle combinatoire.

Les variables définies à l'intérieur d'une porte sont utilisables pour en décrire les sorties.

Il est possible de fournir un sous-tableau partout où est attendue une liste de variables.

A contrario, il est indispensable de donner un sous-tableau comme sortie de certaines instructions (appel d'une porte et de la primitive *lw*).

Enfin, une fonction *for* est disponible et elle utilise des indices entiers sur lesquels sont permis les calculs de l'arithmétique élémentaire.

**Syntaxe** Le code peut être émaillé de commentaires, "à la C", entre les caractères "/*" et "*/".

Le nom de la porte doit commencer par une majuscule. Il est suivit des caractères "<-" puis, entre parenthèses, de la liste des entrées séparées par des virgules.

Les entrées sont soit des variables dont on donne le nom, soit des tableaux dont on donne le nom puis la taille entre crochets.

Les instructions sont soit des déclarations de tableaux avec le mot clé *array* suivi du nom du tableau et de sa taille entre crochets, soit des boucles *for* avec, entre parenthèses et séparés par des virgules, le nom de l'indice, sa valeur initiale et sa valeur finale puis, entre accolades, le corps de la boucle, soit des assignations de la valeur d'une expression à une variable ou un sous-tableau avec l'opérateur "=".

Un sous-tableau est donné par son nom, suivi, entre crochets, d'un intervalle, c'est-à-dire de ses deux extrémités séparées par les caractères ".." .

Une expression est la combinaison des portes logiques et primitives de base, soit infixes, soit préfixes, prenant leurs arguments entre parenthèses quand il y en a plusieurs. On peut aussi faire appel à des portes précédemment définies en passant les entrées avec les caractères "<-" suivis d'une liste de variables entre parenthèses.

## 1.2 Le compilateur

Le compilateur traduit le programme dans un langage intermédiaire de description de net-list topologiquement triée.

Après avoir parsé le code, le compilateur effectue de nombreuses vérifications sémantiques. Cette étape est l'occasion de lister les variables qui sont définies à l'intérieur de chaque porte. Puis il crée un graphe représentant le circuit. Ceci se fait en deux étapes : on crée les noeuds internes aux portes (correspondants aux variables locales) et on relie les différentes portes qu'on aura instancié en créant des arêtes supplémentaires. Afin de pouvoir trier topologiquement le graphe ainsi obtenu avec un algorithme usuel, les registres sont préalablement "éclatés" entre une entrée et une sortie. On mémorise dans des tableaux la correspondance qu'il y a entre entrées et sorties de registres. A l'occasion du tri, une erreur est déclenchée en cas de circuit combinatoire. La dernière étape consiste à générer la net-list, notamment en faisant correspondre à chaque noeud du graphe une variable du langage intermédiaire, qui sera représentée par un numéro.

Nous avons rajouté une option permettant de faire une réallocation automatique des numéros de variables afin de minimiser, lors de la simulation, la taille

de l'environnement. Il s'agit d'une coloration de graphe d'interférence, résolue avec un algorithme glouton.

Le compilateur est sans aucun doute la partie qui nous aura demandé le plus de travail vu les larges possibilités dont nous avons muni notre langage. Il a fallu sans cesse le retoucher et y ajouter des fonctionnalités. Malgré les nombreux tests que nous avions créés, l'écriture du processeur nous a conduit à trouver de nouveaux bugs, étant donné que nous n'avions jamais pu tester le compilateur sur un circuit de cette ampleur.

### 1.3 Le simulateur

**Principe de fonctionnement** Le simulateur exécute linéairement les instructions fournies dans un fichier binaire produit par le compilateur. Il prend en entrées les entrées du circuit et différentes options d'affichage (en console, en décimal et en mode graphique) ainsi que le nombre de cycles à effectuer.

Un environnement est maintenu avec les valeurs des différentes variables intermédiaires et les valeurs des registres sont transmises d'un cycle sur le suivant.

La ROM est représentée en mémoire par un tableau de listes de booléens. Les adresses sont fournies en binaires sous forme de liste de booléens.

**Options supplémentaires** Une option permet de précharger la ROM avec le contenu d'un fichier. Nous avons aussi rajouté un mode de fonctionnement "horloge", pour des circuits à une entrée, qui met la valeur de cette entrée à 1 toutes les secondes (et 0 le reste du temps). Ce mode permet d'exécuter la montre en temps réel. Inversement, il suffit de passer une entrée constante égale à 1 pour exécuter la montre en accéléré.

## 2 Le microprocesseur

### 2.1 Description générale

Le microprocesseur est largement inspiré d'un microprocesseur MIPS mais nous avons gardé un minimum d'instructions, utiles pour réaliser le programme de la montre.

L'ALU calcule l'addition, la soustraction, les tests de comparaison et les opérations logiques bit à bit.

Les instructions sont chargées depuis la ROM à partir de l'adresse 0.

Nous avons rajouté des instructions de branchement et de saut inconditionnel.

L'unité de contrôle décode les instructions suivant les mêmes conventions que MIPS.

Le microprocesseur possède 32 registres ; le registre *$31* est réservé pour enregistrer la valeur du timestamp. Le microprocesseur prend une entrée, un

top d'horloge, et met à jour le timestamp automatiquement. Il renvoie systématiquement en sortie les valeurs de 14 registres déterminés qui correspondent aux unités, dizaines, etc des heures, minutes, secondes, jour, mois, années.

## 2.2 Assembleur

Nous disposons d'un assembleur qui est inspiré de MIPS. Les instructions fournies sont : *add*, *sub*, *and*, *or*, *li*, *beq*, *j* et il est possible de définir des étiquettes dans le code. Le programme assembleur se charge de traduire cela en binaire, et génère un fichier "a.out" dans le format demandé par le simulateur pour précharger la ROM. Ainsi, un programme exécuté sur le microprocesseur sera systématiquement préchargé dans la ROM de cette manière.

**Conventions détaillés de l'assembleur** Les instructions sont codées sur 32bits, décomposées comme suit :

```
[ 3bits ] [ 29 bits ]
```

Les 3 premiers bits contiennent l'opcode de l'instruction, c'est à dire son type. Au vu du jeu d'instruction réduit, ces bits permettent de donner directement les valeurs de contrôle sans se préoccuper de mettre en place une table de vérité. On obtient les valeurs de contrôle suivantes :

**datasrc** : choisit l'entrée du bloc de registre contenant les données à écrire. 0 : sortie de l'ALU. 1 : la constante étendue à 32 bits contenue dans les 16 bits bas de l'instruction. Défini comme le 3ème bit de l'opcode.

**regWrite** : choisit si on va écrire ou non dans les registres 0 : on garde les valeurs des registres, 1 : on écrit dans 1 registre. Défini comme l'opposé du premier bit de l'opcode.

**jump** : choisit la valeur du prochain PC : soit celle venant de l'ALU, soit celle venant des 29 bits de la partie basse de l'instruction. 1 : on prend la valeur de l'instruction. 0 : on prend la valeur de l'ALU. Défini comme le 2ème bit de l'opcode.

**branch** (br) : choisit la valeur du prochain PC 0 : on prend PC + 1 , 1 : on prend l'adresse donnée par PC +1 auquel on ajoute la partie basse de l'instruction de branchement étendue à 32bits. Défini comme le premier bit d'opcode.

**writeReg** : choisit la source de l'adresse du registre à écrire, sur 5 bits. 0 : partie 16-20 de l'instruction, 1 : partie 11-15 de l'instruction. Défini comme l'opposé du 3ème bit d'opcode .

**R type** Les instrucions arithmétiques et logiques.

```
[ 3bits ] [3bits] [ 5 bits ] [ 5 bits ] [ 5 bits ] [ 5 bits] [ 6bits ]
opcode      vide     rs          rt         rd        vide    ALUfunc
```

Les 6 bits d'ALUfunc choisissent l'opération que l'ALU doit effectuer. Seuls les 3 bits de poids faible comptent ALUopérations :

**000** : AND

**100** : OR

**010** : ADD

L'opcode est 000, donc :

- datasrc = 0, résultat de l'instrution en entrée des registres
- regWrite = 1 : on écrit le résultat dans le registre,
- jump = 0 : on ne fait pas de saut,
- branch = 0 : on prend l'instruction suivante,
- writeReg = 1 : on prend la partie rd 11-15 de l'instruction comme registre de destination.

**Load** load immediate

```
[ 3bits ] [ 3bits ] [ 5bits] [ 5 bits ] [ 16 bits ]
opcode     vide      vide      rd         entier
```

L'opcode est : 001, donc :

- datasrc = 1, partie basse de l'instruction en entrée des registres,
- regWrite = 1, on écrit dans le registre
- jump = 0,
- branch = 0, on prend l'instruction suivante pour PC
- writeReg = 0, on prend la partie rd 26-20 comme destination

**Jump** Les instructions de saut, avec juste jump dans les faits.

```
[ 3bits ] [       29 bits          ]
  opcode     addresse du futur PC
```

l'opcode est 110, donc

- datasrc = 0,
- regWrite = 0,
- jump = 1,
- branch = 1 (Don't care, puisque jump = 1),
- writeReg = 1 (Don't care, car regWrite = 0)

**Branch** L'instruction *beq*.

```
[ 3 bits ] [ 3bits ] [ 5bits ] [ 5bits ] [ 16 bits ]
  opcode    vide       rs        rt        offset
```

L'opcode est 100, donc :

- datasrc = 0
- regWrite = 0
- jump = 0
- branch = 1
- writeReg = 1 (Don't care)

# 3 Le programme de la montre

Le programme de la montre est assez simple:

il garde dans des registres les valeurs des chiffres à retenir: unités et dizaines des secondes, minutes, heures, jours, mois, années, ainsi que les unités et les dizaines des siècles. Elle comprend également un registre qui permet de compter modulo 4 pour savoir si l'année est bissextile ou non, ainsi que des registres contenant le nombre d'heures, de mois et de jours, afin de faciliter la remise à zéro du compteur en temps voulu. Pour connaître le nombre de jours dans un mois, on se donne aussi un registre dans lequel on stocke cette information, il est mis à jour à chaque changement de mois.

Ensuite, on effectue une simple boucle, qui incrémente le chiffre des unités des secondes à chaque fois que le timestamp a changé, celui-ci étant lié aux entrées fournies par le simulateur, et qui transmet la retenue s'il dépasse le maximum autorisé (9), qui va se propager jusqu'à actualisation complète de la date.