Rapport de projet – Système digital

Théophile Bastian, Noémie Cartier, Nathanaël Courant

24 janvier 2016

Résumé

Notre objectif dans ce projet a été principalement d'avoir un code *correct* et *rapide à l'exécution* ; nous pensons avoir atteint ces deux objectifs. Dans ce but, nous avons commencé par implémenter non pas un simulateur, mais un *compilateur* de netlists ; puis avons produit un code assembleur pour l'horloge extrêmement optimisé, tout en conservant un nombre constant de cycles processeur par incrément de seconde. Notre processeur en lui-même s'inspire fortement d'une architecture ARM, sans toutefois rechercher l'exhaustivité des opérations, ce qui nous a permis d'obtenir un processeur fonctionnel et composé de peu de portes, donc efficace. Pour finir, nous avons implémenté une GUI affichant un écran composé de blocs 7 segments, contrôlés directement par la sortie du processeur, optimisée jusqu'à ce que le programme limitant la vitesse d'exécution soit le processeur lui-même.

En définitive, les programmes obtenus, une fois combinés correctement, répondent bien au cahier des charges imposé et permettent d'atteindre une vitesse d'exécution en mode *fast* de l'ordre de **9.25 jours simulées par seconde réelle**.

Table des matières

1 Utilisation globale

1.1 Utilisation

La compilation de toutes les composantes du projet s'effectue simplement par un `make` à la racine du projet. Les dépendances suivantes sont toutefois nécessaires :

- OCaml : `ocamlbuild`, `ocamlopt`, `ocamllex`, `menhir` ;
- C : `gcc` ;
- C++ : `g++` ;
- Python : `python3` ;
- Qt version ≥ 4, `qmake` ;
- un environnement bash.

Le lancement de l'horloge peut se faire dans trois modes :

- temps réel : `make run` ou `./run_realtime.sh`, horloge en temps réel synchronisée sur le temps système à l'initialisation ;
- rapide : `make run-fast` ou `./run_fast.sh`, horloge en accéléré initialisée sur le temps système ;
- rapide, initialisé à 0 : `./run_fast_0.sh`, horloge en accéléré initialisée au 1er janvier 0000, `00:00:00`, utile pour évaluer la performance.

1.2 Organisation du projet

Les éléments du projet sont répartis dans des dossiers comme il suit :

- `assembler` : l'assembleur pour notre processeur (§4) ;
- `clock` : les codes assembleur de nos programmes d'horloge et nos quartz (§5) ;
- `display_gui` : l'interface d'affichage (§6) ;
- `processor` : le code produisant la netlist de notre processeur (§3) ;
- `simulator` : le compilateur de netlists (§2).

2 Compilateur de netlists

Pour des raisons de performances, nous avons préféré *compiler* (et non interpréter) nos netlists. Notre compilateur compile les netlists vers du C, qui peuvent ensuite être compilées avec `gcc` par exemple (optimisant encore le code produit avec `-O2` ou autre option d'optimisation).

2.1 Utilisation

Pour compiler le compilateur, il suffit de faire `ocamlbuild compiler.native` dans le dossier du simulateur (`simulator`), ou un simple `make` dans le dossier du projet.

L'utilisation du compilateur est la suivante :

```
./compiler.native [options] fichier.net
```

Le compilateur produira un fichier C `fichier.c`, et un fichier exécutable `fichier`. Les options acceptées sont les suivantes :

- `-print` : n'écrit que le code C, en particulier, ne produit pas l'exécutable `fichier` ;
- `-iomode` n : choisit le mode d'entrée-sortie de la netlist compilée. Les modes d'entrée-sortie sont les suivants :
  - 0 : entrées-sorties interactives, les nappes de fils étant considérées comme des entiers 64 bits non signés, qui sont affichés et demandés à l'utilisateur en décimal. C'est la valeur par défaut ;
  - 1 : chaque bit d'entrée-sortie est représenté par un caractère `'0'` ou `'1'` ;
  - 2 : chaque nappe d'entrée-sortie est représentée de la manière la plus compacte possible sur un nombre entier de caractères, dans l'ordre *big-endian* ;
- `-n` n : nombre de cycles à simuler. Si n est omis, il est pris comme égal à $+\infty$.

Le fichier exécutable produit prend autant d'arguments que le nombre de ROMs du circuit ; chacun étant un fichier binaire contenant l'état initial de la ROM.

2.2 Fonctionnement

Le compilateur commence par ordonner les équations afin que celles-ci s'exécutent dans le bon ordre (on n'utilise pas une variable avant de la mettre à jour, avec un traitement spécial des registres : ceux-ci sont en effet modifiés à la fin). Ensuite, la traduction de chaque équation est ajoutée dans le corps de la boucle principale du programme.

Les nappes de fils sont stockées dans des entiers 64 bits ; en particulier, cela implique que les nappes de fils de plus de 64 bits ne sont pas supportées facilement. Par souci de simplicité, ces nappes ne sont pas supportées du tout. Cela n'affecte pas notre processeur, étant donné que celui-ci n'utilise que des nappes d'au plus 64 bits.

2.3 Optimisations

On utilise bien sûr des opérations bit-à-bit lors d'opérations comme `AND`, `OR` ou `MUX` sur des nappes de fils.

De plus, on garde une table permettant de savoir où trouver chaque nappe de fils ou fil (i.e. dans quelle variable, et sa position interne à cette variable), ce qui permet de drastiquement diminuer le nombre d'opérations `SLICE`, `SELECT` et `CONCAT`. Le compilateur optimise finalement l'appel d'opérateurs sur chaque fil d'une nappe (ou de deux nappes) en un seul opérateur bit-à-bit, afin de simplifier encore plus le code produit. Ainsi, dans un additionneur complet, les portes `XOR` et `AND` se retrouvant entre les deux entrées sur chaque fil se retrouvent transformées en une unique opération bit-à-bit sur les nappes.

Finalement, lorsqu'un unique fil est étendu en une nappe complète (par le biais d'opérations `CONCAT`), on optimise cela en utilisant un `-` unaire : `sortie = (-(entrée&1))&`*masque*. Ceci permet d'éliminer un grand nombre d'opérations `CONCAT` dues à l'extension d'un fil en une nappe, par exemple pour des `MUX` ou pour l'ALU.

On effectue donc une simplification assez importante de la netlist de cette manière.

3 Processeur

Nous avons fait le choix d'écrire notre processeur en Python : le code, une fois exécuté, produit un fichier de netlist qui peut être compilé avec le compilateur de circuits (§2). Le code a été conçu modulairement, avec un fichier Python par sous-unité du processeur (ALU, ...). Le fichier `main.py` se charge donc exclusivement de faire le lien entre ces unités, « branchant » des fils d'une unité à l'autre ; tandis que `netlist.py` définit les fonctions à appeler pour construire la netlist en mémoire, affichable à la fin.

Au total, le processeur est composé de 1699 portes logiques (dont certaines sur des nappes de fils) ; dont seulement 698 en excluant les `CONCAT`, `SELECT`, `SLICE` (ce qui est pertinent puisque ces portes sont optimisées par le compilateur).

3.1 Architecture du processeur

3.1.1 Mémoires et registres

Notre processeur comporte trois types de mémoire :

- la ROM, dans laquelle est stocké exclusivement et intégralement le programme généré par notre assembleur, sous forme d'*opcodes* (§3.1.2) ;
- la RAM, initialement vide et accessible via la *memory unit* (§3.2.3) ;
- les registres, au nombre de 16, nommés `%r00` ... `%r15`.

Plusieurs registres ont des fonctions spéciales dans le processeur :

- `%r00` est le *program counter* (PC) : il contient l'adresse dans la ROM de l'instruction actuelle. S'il n'est pas modifié au cours d'un cycle processeur, il est incrémenté. En particulier, un saut dans le programme n'est rien d'autre qu'une modification de ce registre.
- `%r01` est le *registre d'entrée* : à chaque cycle, le processeur lit sur l'entrée standard une valeur 64 bits et l'y stocke.

— `%r02`, `%r03` sont les *registres de sortie* : à chaque fin de cycle, le processeur écrit sur la sortie standard les valeurs 64 bits contenues dans ces registres.

À chaque cycle, il est possible d'écrire sur *un* registre et d'en lire *deux* (indiqués par leur adresse), à l'exception du PC qui est systématiquement donné à l'*opcode unit* (§3.2.6).

3.1.2 Opcodes

Les instructions de l'assembleur sont traduites en *opcodes* par l'assembleur (§4), des valeurs sur 64 bits décrivant intégralement une opération (*ie.* un cycle) du processeur. Cet opcode est séparé en plusieurs morceaux ayant des significations différentes, aisément *slicé* pour qu'une unité du processeur ne reçoive que la partie de l'*opcode* pertinente.

Bits	Contenu
1 – 4	Condition d'exécution
5 – 8	Code de l'instruction
9	Écrire le résultat ?
10	Utiliser le *carry bit* ?
11	Écrire les flags ?
12 – 15	Registre de destination
16 – 19	op_1
20	Remplacer op_1 par $0 \cdots 0$?
21 – 46	op_2

TABLE 1 – Contenu d'un opcode

Bits	Contenu
1	Registre (`1`) ou constante (`0`) ?
2 – 17	Constante (16 bits)
18 – 19	Opération de shift
20 – 25	Valeur du shift

TABLE 2 – Contenu d'op_2 (positions relatives)

Nom	Opération
`LSL`	Shift à gauche
`LSR`	Shift logique à droite
`ASR`	Shift arithmétique à droite

TABLE 3 – Opérations de shift

3.1.3 Opérations assembleur supportées

Les opérations assembleur supportées sont :

— `ADD` : addition
— `ADC` : addition avec ajout du *carry bit* de la précédente opération
— `SUB` : soustraction
— `SBC` : soustraction + carry - 1
— `RSB` : soustraction inversée ($op_2 - op_1$)
— `RSC` : soustraction inversée + carry - 1
— `AND` : et logique
— `EOR` : ou exclusif logique
— `ORR` : ou logique
— `BIC` : op_1 AND NOT op_2
— `CMP` : soustraction sans conserver le résultat (lève les flags)
— `CMN` : addition sans conserver le résultat
— `TST` : et logique sans conserver le résultat
— `TEQ` : ou exclusif logique sans conserver le résultat
— `MOV` : déplacement d'op_2 vers un registre
— `MVN` : déplacement de NOT op_2 vers un registre
— `LDR` : accès RAM en lecture
— `STR` : accès RAM en écriture
— `JMP` : saut vers un label (sucre syntaxique : est assemblé en une opération sur `%r00`)

Chaque opération peut être assortie d'une conditionnelle dépendant des flags placés par la dernière opération.

Pour plus de détails sur l'assembleur utilisé, consultez le rapport précédent mis à jour : `https://frama.link/PwqfabCE`.

3.2 Détail des unités

3.2.1 ALU

Grâce à un choix adapté des opcodes pour les bonnes opérations, on limite ici le nombre de multiplexeurs utilisés.

L'ALU commence par déterminer les op_1 et op_2 adaptés à l'opération considérée :

— si le dernier bit de l'instruction est un 1 (vrai dans le cas de SUB et BIC), l'op_2 utilisé sera la négation de celui fournit par l'op_2 unit (en prenant en compte la retenue dans le cas de la soustraction, on retrouve bien l'opposé de l'op_2 donné à la base) ;

— si le deuxième bit de l'instruction est un 1, et uniquement dans le cas des opérations arithmétiques (donc seulement das le cas de l'opération RSB), l'op_1 utilisé sera la négation de celui fourni par l'op_1 unit.

L'ALU calcule ensuite séparément les résultats de toutes les opérations booléennes ainsi qu'un unique résultat arithmétique, qui sera une addition ou une soustraction (dans un sens ou dans l'autre) en fonction des op_1 et op_2 sélectionnés plus tôt. Nous justifions cette séparation par le fait qu'une structure d'additionneur est suffisamment importante pour la limiter à une unique occurrence, alors que les différentes opérations booléennes, en plus d'être difficiles à combiner (à l'exception de AND et BIC, qui l'ont été), sont nettement moins coûteuses en terme de nombre de portes logiques.

Enfin, l'ALU sélectionne la bonne sortie parmi toutes celles calculées. En particulier, un troisième bit à 0 indique une opération arithmétique, tandis qu'un 1 indique une opération booléenne.

3.2.2 Flags unit

Son but est de stocker les flags et de donner la valeur logique d'une conditionnelle (l'une des 16, *eg.* GE). Elle contient donc un registre de nappe de taille 4 et un arbre de MUX.

3.2.3 Memory unit

Gère l'accès à la RAM, se contente de renvoyer le contenu de la mémoire à l'adresse indiquée, et d'écrire une valeur dans celle-ci à cette adresse si l'instruction est une instruction d'écriture.

3.2.4 op_1 unit

Essentiellement un fil entre la sortie op_1 des registres et l'entrée op_1 de l'ALU, sauf lorsqu'on demande à op_1 d'être nul (utile pour les sauts).

3.2.5 op_2 unit

Choisit entre la sortie op_2 des registres et la constante littérale qui lui est donnée dans l'opcode ; contient le barrel shifter de l'op_2.

3.2.6 Opcode unit

Accède à la ROM à chaque cycle pour récupérer l'*opcode* (§3.1.2) sous le *program counter* (fourni par les registres). Communique avec la *flags unit* pour déterminer si l'éventuelle conditionnelle passée est vérifiée ou non. Si ce n'est pas le cas, désactive l'écriture dans les registres, dans la RAM et dans les flags (l'instruction est quand même exécutée, mais sans aucun effet).

3.2.7 Registers unit

Crée 16 registres, dans lesquels on peut lire et écrire ; certains sont particuliers car sont le *program counter*, le registre d'entrée ou un des registres de sortie. Le *program counter* est incrémenté de 1 si on ne lui met pas une nouvelle valeur, contrairement aux autres registres qui restent inchangés. Des arbres de MUX/DEMUX permettent d'indexer les registres par une adresse sur 4 bits.

3.2.8 Result selector

Choisit simplement la valeur écrite dans les registres entre celle donnée par la *memory unit* (§3.2.3) et celle donnée par l'ALU (§3.2.1) en fonction de l'opcode.

Il ne s'agit ici que d'un multiplexeur : si l'instruction a son premier et son quatrième bit égaux à 1, on sélectionne le résultat d'un accès mémoire ; sinon, on sélectionne le résultat donné par l'ALU.

4 Assembleur

L'assembleur est très probablement la partie la plus simple de notre projet : les opcodes (§3.1.2) utilisés sont très simples à construire, ainsi notre assembleur (programmé en OCaml) est constitué d'un lexer/parser, d'un traducteur d'AST en opcodes, d'un « linker » gérant les sauts (`JMP`) et d'un module écrivant la ROM (§3.1.1) finale.

4.1 Gestion des sauts

Pour gérer les instructions `JMP`, on se contente de commencer par calculer l'adresse de chaque label par un simple parcours du code, un label n'occupant aucune adresse et toutes les autres instructions en occupant une. Il suffit ensuite de remplacer une instruction `JMP` *l* en `MOV %r00,` *addr*, où *addr* est l'adresse de *l*.

5 Programme de l'horloge

Le programme de l'horloge doit gérer l'heure et la date, mais également le décodage de celles-ci vers un affichage 7 segments, et pour la version temps réel, la synchronisation sur l'entrée qui lui donne un *tick* à chaque seconde. Les programmes sont dans les fichiers `clock/clock.s` pour la version rapide, et `clock/clock_fast.s` pour la version temps réel.

5.1 Lecture de l'entrée

Pendant les 7 premiers cycles, le processeur reçoit l'heure actuelle dans son registre d'entrée, sous forme secondes, minutes, heures, jours, mois, années, siècles (dans l'ordre). On stocke ces valeurs, puis si l'entrée du processeur était `/dev/null`, on initialise au 1er janvier 0000, `00:00:00` à la place. Pour cela, il suffit de vérifier si les valeurs qui ont été stockées étaient négatives.

La deuxième partie de la lecture de l'entrée se déroule dans le mode temps réel : l'entrée vaut alors `0`, sauf une fois par seconde réelle, où elle vaut `1`. Pour attendre ce moment, on utilise l'instruction suivante : `ADD %r00, %r00, %r01`. En effet, `%r01` étant le registre d'entrée, et `%r00` le compteur d'instructions, le résultat de cette instruction sera de ne passer à l'instruction suivante que si l'entrée était `1`. Le seul problème de cette instruction est qu'elle n'est pas très robuste : si l'entrée n'est ni `0` ni `1`, on effectue un saut à un endroit incontrôlé du code... On pourrait la remplacer par `ADD %r00, %r00, %r01, LSR #63`, qui est robuste et ne passe à l'instruction suivante que si l'entrée est strictement négative.

5.2 La boucle principale

La fréquence de l'horloge est de deux cycles processeur par seconde simulée, ce qui est optimal si on veut que ce nombre soit constant : en effet, il y a deux registres de sortie qu'il faut mettre à jour, et le processeur ne peut écrire que dans un seul registre à chaque cycle...

Pour atteindre cette fréquence optimale, il a fallu dérouler complètement la boucle incrémentant les secondes. Ainsi, dans le code final, chaque itération de la boucle principale correspond à une minute, avec une mise à jour tous les deux cycles de l'affichage contenant les secondes. Cette dernière est faite en effectuant un `EOR` avec une constante bien choisie, qui est le `XOR` de l'affichage 7-segments du nombre de secondes actuel à cet instant de la boucle, et de celui de ce nombre plus 1.

L'autre instruction qui est éxécutée une fois sur deux calcule l'état de l'affichage à la minute suivante et le stocke dans deux registres temporaires, `%r12` et `%r13`. Pour cela, on commence par calculer le nombre de jours du mois actuel, en vérifiant si l'année est bissextile (en prenant en compte les exceptions pour les années multiples de 100 et de 400), puis on incrémente les minutes, on vérifie si celles-ci valent à présent 60. Si c'est le cas, on remet à jour les minutes, on incrémente les heures, et on fait de même avec successivement les heures, les jours, les mois, les années et les siècles. Une fois que le nouvel état est calculé, on utilise une table en mémoire pour décoder ces nombres vers l'affichage 7-segments, et on stocke cet affichage dans `%r12` et `%r13`. On remplit ensuite les espaces restants pour des instructions jusqu'à la fin de la boucle par des `NOP` (`ADD %r15, %r15, #0`), pour garder deux cycles d'horloge par seconde simulée et que la fréquence de l'horloge reste stable. Enfin, à la toute fin de la boucle, on met à jour la nouvelle sortie avec les résultats qu'on avait stockés, et on recommence une nouvelle itération (le saut à la fin de la boucle comptant pour une instruction).

Dans la version temps réel, on insère de plus une instruction d'attente (`ADD %r00, %r00, %r01`) avant chaque mise à jour des secondes, pour attendre le signal de passage à la seconde suivante.

5.3 Initialisation

L'initialisation de l'horloge consiste à initialiser la mémoire RAM avec les informations de décodage 7-segments, et le nombre de jours dans chaque mois. On effectue ensuite un saut dans une version spécialisée de la boucle pour la première itération, qui initialise la sortie, qui calcule l'état de l'itération suivante, puis qui effectue un saut indexé afin de se retrouver au bon nombre de secondes à initialiser. La partie dans laquelle on effectue le saut est comme la boucle principale, excepté qu'il n'y a pas d'autre calcul à effectuer et que les instructions intermédiaires sont donc uniquement des `NOP` cette fois-ci. On effectue enfin un saut dans la boucle principale une fois que le nombre de secondes est revenu à 0, ce qui permet à celle-ci de fonctionner correctement.

5.4 Génération de l'entrée standard

Pour générer une entrée standard adéquate pour le processeur, nous avons commencé par utiliser un programme Python (`clock/quartz.py`) donnant au processeur la date et l'heure actuelle, puis, selon le mode souhaité, soit contrôle ses temps de `sleep` pour avoir exactement un incrément par seconde réelle, soit se termine pour que le programme reçoive `/dev/null` comme entrée standard.

Nous avons remarqué que dans le second cas, cette procédure de remplacement de l'entrée était *très* lente. En effet, après avoir reprogrammé le quartz en C (`clock/quartz.c`) pour initialiser l'horloge, puis faire des `putchar(0);` en boucle infinie, nous avons constaté une amélioration des performances de l'ordre de 300%.

Le modèle final adopté est donc :

— `clock/quartz.py` pour le mode temps réel (les performances ne sont pas si importantes) ;
— `clock/quartz.c` pour le mode rapide ;
— `clock/quartz0.c` pour le benchmarking, qui initialise l'heure à `00:00:00 01-01-0000`, ce qui est plus simple à lire dans le cadre du benchmarking.

6 Interface de sortie

L'interface, programmée en C++ avec Qt, est composée de 14 afficheurs 7 segments, pilotés par l'entrée standard (détaillé ci-dessous §6.1). Elle supporte sans ralentissement les quelques 25Mo d'entrée par seconde en mode *fast*.

6.1 Utilisation

La compilation s'effectue par un simple `qmake && make` dans le dossier de l'interface (`display_gui`), ou un `make` dans le dossier du projet ; le binaire produit est `display_gui/display_gui`.

Le programme ne prend aucun argument. Il attend sur son entrée standard des caractères formatés comme la sortie du processeur, c'est-à-dire 16 caractères à la suite. Les caractères contrôlent chacun un afficheur, dans l'ordre `--HHMMSSYYYYMMDD` respectivement heure à deux chiffres, minutes, secondes, année à 4 chiffres, mois, jour.

Les bits du caractère contrôlent les segments d'un afficheur, à savoir en commençant par les poids forts `-gfedcba` (figure 1 ci-contre).

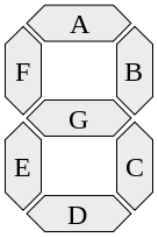

FIGURE 1 – Ordre des segments. Crédit : H2g2bob @ Wikipedia

6.2 Fonctionnement et optimisations

La lecture de l'entrée standard est gérée dans un thread séparé du thread principal, se chargeant de l'affichage ; la communication entre threads est aisée puisque l'un est en écriture seule sur la mémoire partagée quand l'autre est en lecture seule.

Le thread d'affichage est volontairement limité à 30 FPS : 30 fois par seconde, il demande au thread de lecture des valeurs pour chacun de ses segments et met à jour son interface. Ceci fut une première amélioration notable des performances, par rapport à un modèle où les segments sont rafraîchis aussi vite que possible.

Le thread de lecture, quant à lui, a été bien plus optimisé. Les modèles suivants ont été considérés, successivement :

— lorsque des valeurs sont demandées, on lit 16 caractères sur `stdin` ; lorsque le rafraîchissement est terminé, on `fflush` l'entrée standard afin d'ignorer ce qui est arrivé pendant ce temps. Cette option n'a jamais fonctionné, d'autant qu'elle ferait perdre tout repère de « début de bloc de 16 caractères » ;

— on lit en continu des blocs de 16 caractères et, à chaque demande de rafraîchissement, on renvoie les 16 derniers acquis. Cette option marchait bien, mais était trop lente ;

— La « boucle infinie » de lecture étant gérée avec Qt, le modèle signal/slot impose de faire en réalité une fonction de lecture se terminant par l'empilement d'un événement « appeler la fonction de lecture ». Cette procédure est très coûteuse (beaucoup plus que `getchar` !). Ainsi nous ignorons systématiquement 12 blocs sur 13 : la fonction de lecture lit 13×16 caractères et ne conserve que les 16 derniers. La fréquence de notre processeur nous permet (et nous oblige) à procéder de la sorte pour avoir une vitesse d'affichage raisonnable.

Avec ce fonctionnement, nous arrivons à une horloge capable de supporter l'entrée massive qui lui est fournie chaque seconde, sans ralentir le système global.