Systèmes Numériques : de l'algorithme au circuit
Processeur Pomme

Samuel Coulomb, Théotime Le Hellard, Vincent Peth

25 janvier 2022

Notre projet est commenté au travers des différents README dédiés à chaque parties, comme le précise le README principal. Ce rapport vise plus à expliquer les difficultés rencontrées et les choix effectués.

1 L'assembleur pomme

Nous avons créé notre langage assembleur exotique pomme, écrit en Ocaml. Toute sa syntaxe, ainsi que l'explication du code binaire de sortie est largement détaillé dans le README dédié. Avant de lire la liste des difficultés qui suits, il est indispensable de lire le README.

— Nous avons 8 registres, initialement nous avions prévu 4 registres sur 8 bits et 4 sur 16 bits signés. Ce qui amenait moult difficulté, nous avons finalement opté pour 8 registres sur 16 bits signés.

— Initialement nous avions une unique RAM et l'instruction sept_batons, écrivait dans la RAM de travail. Mais pour se faciliter la vie pour l'envoie des informations à l'afficheur (cf la section dédiée à l'affichage), nous avons opté pour deux RAM séparées.

— Nous avons tâtonné avant d'aboutir à l'instruction qui accélère considérablement le programme : `incrz <int> <reg>`. Finalement, elle augmente la valeur du registre de 1, et si celle-ci atteint <int>, elle retombe à zero. Typiquement `incrz 24 r1` avec r1 de valeur 23, le fait passer à 0. Toutefois, si r1 vaut 120, il va simplement passer à 121. C'est à l'utilisateur de faire attention.

— Nous avons eu des difficultés avec les adresses.
Premièrement, on ne peut pas enregistrer un entier dans la RAM. En effet une adresse peut s'écrire de la forme `($r1 + 5)`, en utilisant déjà un entier, or notre microprocesseur ne peut prendre qu'un seul entier brut en entrée. Ainsi il faut penser à faire `set 31 r4` puis `save r4 (5)`.
Deuxièmement, on préfère toujours que ce soit le "premier" registre qui parte dans l'Alu. Ainsi, même si de manière générale le deuxième registre est celui où sera stocké la valeur finale (typiquement pour `add $r1 r2`). Dans le cas d'adresse, que ce soit pour `save r1 ($r2 + 5)` ou `load ($r1 + 5) r2`, dans la transcription binaire, le premier registre sera toujours celui d'adresse ($r2 pour save, $r1 pour load) et le deuxième registre s'adapte.

2 Le microprocesseur

Notre idée phare dans la création du microprocesseur était la simplicité, nous voulions limiter le nombre de portes au maximum pour pouvoir avoir une meilleure efficacité, et nous avons fait

plusieurs choix en ce sens :

Dès le début, nous voulions une architecture sur peu de bits : nous avions pensé à utiliser 8 registres, dont 4 signés sur 16 bits et 4 non signés sur 8 bits (nous espérions que ces registres seraient plus rapide, et que cela accélère notre microprocesseur sur les petits calculs répétés, comme par exemple l'incrément des secondes). Finalement, ce choix a été abandonné suite à la prise de conscience des problèmes qu'il suscitait : il nous aurait fallu une alu sur 8 bits non signés et une alu sur 16 bits signés, et comment aurions géré les opérations entre deux registres de tailles différentes ? C'est pourquoi nous avons finalement choisi d'utiliser simplement 8 registres signés sur 16 bits.

Le second problème rencontré venait de carotte.py : alors que l'outil était pratique et fonctionnel, il s'est mis d'un coup (lors d'utilisation des modules) a ne pas respecter ses conventions sur les noms : les noms à droite des équations n'étaient pas les mêmes que les noms à gauche, d'où des problèmes à la compilation de la netlist. Merci à Théophile Wallez de l'avoir prestement réparé (si des bogues surgissent, installer assignhooks, puis la dernière version de carotte.py).

Le second et principal problème que nous a posé la création du microprocesseur peut paraître insignifiant, mais a joué en réalité un grand rôle dans 4 des 5 bogues que nous avons chassé lors de la phase de test du microprocesseur (pour information, le cinquième bogue était une erreur sur le côté du bit de poids faible dans l'incrément des lignes de lecture) : il s'agit de l'ordre des opérandes dans l'alu.
Les conventions que nous nous étions fixées étaient :

— Le registre 1 est toujours utilisé pour les calculs d'adresse
— Le registre 2 est toujours le registre d'écriture s'il y en a un
— L'alu possède une opération identité qui renvoie le premier argument

Au début, le registre 1 était le premier argument de l'alu et l'entier ou le registre 2 le second (au choix selon le dernier bit de la commande). Cette façon de faire avait l'avantage d'être simple à exécuter, mais le désavantage d'être très fausse : les cas qui pouvaient se présenter étaient :

— Calcul d'adresse : on veut additionner l'entier et le registre 1, peut importe l'ordre
— Move : on veut renvoyer la valeur du registre 1, r1 doit donc être l'opérande de gauche
— Set : on veut soit renvoyer la valeur du registre 1, soit celle de l'entier, selon un bit d'indication
— Opération : on veut renvoyer le résultat de l'opération, mais attention : les deux opérandes sont soit r1 et r2 ou r2 et l'entier, et non pas r1 et l'entier, comme c'est le cas pour les calculs d'adresse. De plus, l'ordre des opérandes est très important, car `sub 5 r2` $= r_2 - 5$. r2 doit donc être le premier opérande.

Finalement, la solution retenue consistait à laisser deux possibilités pour le premier opérande : r2 pour les opérations, et r1 pour les calculs d'addresse et le move (réalisé rapidement grâce à l'opération identité). Le deuxième opérande peut être au choix r1 ou l'entier (selon qu'on a besoin d'un entier ou pas).

En particulier, le set ne peut pas utiliser l'opération identité, car l'entier ne peut pas être le premier argument. Pour réaliser un set, on utilisera donc l'addition entre le registre r1 et l'entier : Soit le premier est le registre nul, et on obtient l'entier, soit l'entier est nul et on obtient la valeur du registre voulu. C'est pour cette raison (durée de l'addition) que `set $rax rbx` est moins efficace que `move rax rbx`

Grâce à ces choix, nous avons un microprocesseur de taille assez petite : seulement 3250 portes, dont 2700 sont utilisée par l'alu (en particulier la multiplication qui prend beaucoup de portes).

Le fait que les calculs soient paresseux fait qu'un tour d'horloge n'exécutera généralement pas plus de 300 portes. La longueur du chemin critique est de 528 portes, ce qui est plutôt grand, et nous soupçonnons fortement la multiplication de n'être pas pour rien dans cette valeur de chemin critique.

3 Compilateur de Netlists

Nous avons choisi de compiler les netlist vers le format `.ml`, pour pouvoir gagner en vitesse et en flexibilité par rapport à l'interprétation de netlist.

Pour être le plus efficace possible, nous avons fait le choix d'une évaluation complètement paresseuse des portes : les portes sont des fonctions mémoïzées, et on ne calcule que les portes dont on a besoin (en particulier cela économise des portes sur les multiplexeurs, certains ou et certains and). Si la porte s (numéro 256) a besoin de la porte t (numéro 192) elle appelle var_t (), si on a déjà calculé une fois la porte t, alors on renvoie t_var.(192), sinon on calcule la porte t et on enregistre le résultat dans t_var.(192).

Nous avons fait le choix de représenter les bus par des entiers, ce qui nous limite à des bus de 61 bits au plus, mais qui nous permet de réaliser les opérations directement sur les bus grâce aux fonctions d'opérations logique sur les entiers d'Ocaml. Cela prend également considérablement moins de place que les VBit_Array.

Le principal cauchemar de cette partie était le sens des bus, à savoir faire la différence entre petit boutisme et gros boutisme. En Ocaml, les entiers ont les bits de poids faible à droite, et en netlist il n'y a pas de sens fixé à priori, mais les opérations select, slice et concat font appel à une notion de sens des bus qui n'est pas forcément la même intuitivement que celle d'Ocaml. Il était important de respecter ce sens pour les netlist (les netlists ne pouvait par exemple pas être un miroir de Ocaml, ce qui, en y regardant de loin n'aurait pas eu d'effet indésirable) car les constantes devaient correspondre entre Ocaml et les fichiers `.net` dans lesquelles elles sont harcodées.

Autres :

— Quand et comment traiter les REG, nous avons débattu en TP pour savoir s'il était possible de traiter les registres au bon moment pour ne pas utiliser une tb_reg, ce qui n'est pas le cas. Dans la version compilé on procède ainsi :
On calcule les portes des outputs (qui vont faire appel aux portes dont elles ont besoin). Si on voit `a = REG b`, on a une fonction variable triviale : `let var_a () = t_val.(<num de a>)`.
Une fois ceci fait, on va actualiser tous les registres, on commence par calculer toutes les portes, mais sans modifier t_val cette fois-ci, avec : `t_reg.(<num de a>) <- var_b ()`. Idem on met à jour les RAM.
Puis, une fois ceci fait pour toutes les RAM, on actualise t_val : `t_val.(<num de a>) <- t_reg.(<num de a>)`

— Nous sommes très souples sur les ROM, on peut préciser un dossier où chercher les ROM (avec leur nom.txt) avec `-romdir`, ou les donner dans l'ordre avec `-roms`, par exemple `./microproc.exe -s -roms rom_input.txt ../pomme/clock_quick.txt`, ou enfin les donner manuellement quand le programme les demande.

4 Pour l'affichage sept segments / sept bâtons

Explication des choix faits pour le contact avec l'extérieur, que ce soit en entrée pour recevoir la date exacte actuelle et une pulsation au rythme des secondes, mais aussi en sortie, pour faire afficher le résultat.
Idée : la netlist du microprocesseur possède une unique entrée `real_clock` et une unique sortie `maj_ecran`, les deux sont des mots clés spéciaux, repéré par le compilateur de Netlists. Pour l'entrée, au lieu de demander (quand voulu) un input à l'utilisateur (du style let valeur = ast_insput x 4), `real_clock` demande la parité du nombre de secondes écoulées depuis le début du processus (du style `let valeur = int_of_float (Sys.time ()) mod 2`). De même en sortie, `maj_ecran` n'est pas traité comme un output classique, au contraire sa présence nous signifie qu'il faut ouvrir un écran, et si `maj_ecran > 0`, on actualise l'affichage (via `Affiche.affiche_batons`).

Au niveau du microprocesseur, en plus de la ROM d'entrée contenant le programme à exécuter, on rajoute une deuxième ROM, qui nous sert d'input. C'est par cette ROM qu'on fait passer l'heure de départ, récupérée via `Unix.time ()` et avec l'instruction pomme `load_rom`. D'autre part, en plus de la RAM de travaille, il y a une deuxième RAM dédié à écrire l'heure au format sept batons. Ainsi pour l'affichage, `maj_ecran` étant le *write enable* de la RAM sept_batons, on peut récupérer son numéro, et ensuite appeler l'afficheur avec les données contenues dans cette RAM. Du genre :

```
let ram = t_rams.(2722) in
Affiche.affiche_batons ram.(0) ram.(1) ram.(2) ram.(3) ram.(4) ram.(5) ram.(6)
```

Difficultés

- Apprendre à utiliser le module Graphics d'OCaml.
- Minimiser les opérations d'affichage (assez coûteuses) : on met à jour uniquement les bâtons modifiés en retenant ceux affichés en permanence.
- Convertir efficacement un chiffre en 7 bâtons (nombre de portes et chemin critique).

5 Le main

Le README et le Makefile sont plutôt clairs dans le cas où tout ce passe bien pour compiler le microprocesseur et l'exécuter. On déplace le programme dans le dossier carotte, on appelle carotte.py pour obtenir une netlist, puis on la compile en .ml et enfin on l'exécute. En parallèle on compile le .pomme en .txt pour créer la ROM_prgm, enfin on initialise la rom_input avec la date exacte de départ.

Mais tout ceci se passe bien à condition d'avoir un module Graphics coopérant. Si jamais le module n'est pas dans le dossier de défaut de Ocaml, nous avons eu des gros soucis. En effet, nous avons mis beaucoup de labeur à trouver comment le forcer à le trouver, avec ocamlfind, il faut préciser `ocamlfind ocamlopt -o $@ -linkpkg -package graphics graphics.cmxa affiche.cmx mon_prgm.cmx`. MAIS si jamais le module était bien présent, ocamlfind se mélange les pinceaux, et plante. Ainsi il a fallu faire deux versions du Makefile, et donc apprendre à faire des if en Makefile, via des variables.