Rapport architecture microprocesseur

HORNERO Baptiste
HERZLICH Raphaël
HELARY Axel

Comment compiler ?

Compilation de l'horloge

Pour compiler l'horloge, il faut se placer dans le dossier principal et executer un `make run` si l'on veut avoir un fonctionnement en temps normal, c'est à dire une seconde toutes les secondes. Si l'on veut avoir un fonctionnement rapide, il suffit d'executer un `make runfast`.

Ces deux commandes font les actions suivantes :
- compilation de la netlist
- compilation du code assembleur de l'horloge
- création du fichier `main.net` avec carotte
- appel du simulateur de netlist sur `main.net`

Structure de l'assembleur

Le dossier netlist contient le simulateur de netlist, le dossier assembly contient la définition de l'assembleur, le dossier roms contient la rom obtenue par execution de l'assembleur sur clock.lv (resp. clockfast.lv) pour une utilisation normale (resp. rapide).

Dans Utils, on trouvera les codes de l'ALU et de quelques fonctions utilitaires pour notre fichier principal `main.py` dont la netlist associée (après compilation par carotte) est `main.net`

Options de *netlist_simulator*

Les options sont les suivantes:
- -help : affiche les options
- -n : nombre de tours
- -ten : affiche les nombres en base 10
- -clock : affiche sous forme compatible avec une horloge
- -sec : affiche seulement lorsque la variable `sec` est actualisée

Netlist

Compilation

Afin de compiler le projet on utilisera : `make`

et pour compiler uniquement le scheduler on pourra utiliser : `make schedule`

Utilisation

Afin de simuler la netlist sur le fichier `/test/*.net` on va utiliser la commande : sh

```
./netlist_simulator.byte test/*.net
```

Pour pouvoir utiliser des roms préinitialisée, on va pour chaque rom ident déclarée dans la netlist, on va fournir dans le dossier test (l'emplacement des roms peut être changé dans le `netlist_simulator.ml`) un fichier `ident.rom` selon la norme suivante:

Chaque ligne correspond à un **VBitArray**, ou 0 correspond à `false` et 1 à `true`, Autrement dit pour encoder la **ROM** r :

true	false
false	false
false	true
false	true
true	true
false	false
false	false
false	false

on fournit le fichier `r.rom` :

```
10
00
01
01
11
00
00
00
```

De plus si aucun fichier n'est fourni, la ROM est initialisée entièrement remplie de false. De même si toutes les lignes ne sont pas spécifiées dans le fichier elles sont également initialisées à false. Ainsi pour initialiser la ROM précédente on peut également fournir le fichier `r.rom` :

```
10
00
01
01
11
```

De même pour rentrer dans un input un **VBitArray** [false;true;false;false—]— on rentrera `Valeur de a (array de taille 4):`

Spécifications

Scheduler

Le Scheduler commence par construire un graphe de dépendance entre toutes les variables. Puis ordonne les opérations selon un ordre topologique (s'il y en a un, sinon il renvoit une erreur `Combinatorial_Cycle`).

Il y a cependant 2 choses notables à remarquer lors du passage du scheduler :

- On ne considère par l'opération `REG` comme une opération de dépendance, puisque `y = REG x` donne à y la valeur de x à l'étape précédente (et donc n'est pas impactée par l'ordre de réassignation de x) - Lors d'un appel `y = RAM _ _ ra we wa data`, seule `ra` est considérée comme une dépendance, puisque comme l'on va faire tout les write de **RAM** à la fin de chaque étape, toutes dépendance relative à l'écriture n'impacte pas l'ordre de simulation de l'opération.

Simulateur

Specifications particulières

REG :

- L'assignation d'un registre utilise l'environnement final de la passe précédente (du pre-process s'il s'agit de la première passe). Ainsi la lecture de `y = REG x` à la passe *n* assignera à **y** la valeur de **x** à la fin de la passe **n-1**.

NOT:

- L'application de **NOT** sur un array est effectuée bit à bit.

BINOP:

- Une binop entre deux **VBit** est faite de façon normale
- Une binop entre un **VBit b** et un **VBitArray a** est faite bit à bit sur les éléments de **a** et **b**
- Une binop entre deux **VBitArray a1,a2** renvoie un tableau de la taille minimum entre les deux arrays rempli des opérations bit à bit.

CONCAT:

- Une concat entre deux **VBitArray** agit de façon intuitive
- Le reste des opérations possibles considèrent les **VBit** commes des **VBitArray** de taille 1

SELECT:

- Une select sur un **VBit** n'est légale seulement si on sélectionne l'élément 0, dans ce cas elle renvoie le **VBit**
- Une select sur des **VBitArray** agit de façon intuitive

SLICE:
- Un slice n'est légal que sur un **VBitArray**

MUX:
- On va considérer que l'argument de choix pointe vers le premier élément dans deux cas :
- C'est un **VBit true**
- C'est un **VBitArray** entièrement remplit de true
- Dans tout les autres cas on renverra le deuxième élement

ROM:
- Une addresse est forcément un **VBitArray** de taille inférieure ou égale à `address_size`

RAM:
- Pareil que pour `ROM`

Processeur

Architecture du processeur

L'architecture du processeur est une architecture proche de l'architecture x86, les instructions sont codées sur 32 bits. On a 16 registres accessibles, numérotés de 0 à 15, chacun de taille 32. On ajoute un register de 32 bits correspondant à la position du pointeur sur la ROM, qui correspond au programme lu en cours, qu'on appelle P. On se munit de trois flags : ZF (dernière opération a renvoyé 0 ou non), SF (positivité de la dernière opération), OF (overflow de la dernière opération arithmétique) tous de taille 1.

Instructions

Le set d'instructions est le suivant:

Instruction	Encodage	Description	Arguments
NOP	0000 0000	No opération	
ADD	0001 0001	Addition	rs1 rs2
SUB	0001 0010	Soustraction	rs1 rs2
MUL	0001 0011	Multiplication	rs1 rs2
AND	0010 0001	Et logique	rs1 rs2
OR	0010 0010	Ou logique	rs1 rs2
XOR	0010 0011	Xor logique	rs1 rs2
NOT	0011 0001	Non	rs1
SLL	0011 0010	Décalage gauche logique	rs1
SRL	0011 0011	Décalage droite logique	rs1

et :

Instruction	Encodage	Description	Arguments	Description formelle
MOV	0110 0001	Met un registre à la valeur d'un autre registre	rs1 rs2	rs1 ← rs2
MOVI	0111 0001	Met une valeur immédiate dans un registre	rs1 immediate	rs1 ← immediate
CMP	1000 0001	Compare deux valeurs	rs1 rs2	Mise à jour des flags
JMP	0100 0001	Jump à une valeur	immediate	P ← immediate
JNE	0100 0010	Jump si non égal (ZF = 0)	immediate	Si ZF =0 alors P← immediate
JE	0100 0011	Jump si égal (ZF = 1)	immediate	Si ZF = 1 alors P ← immediate
JGE	0100 0100	Jump si plus grand ou égal	immediate	Si OF=SF alors P ← immediate
LOAD	0101 0001	Lit dans la ram	rs1 rs2	rs1 ← R[rs2]
STORE	0101 0010	Stocke dans la ram	rs1 rs2	R[rs2] ← rs1
LOADFIX	0101 0011	Lit dans la ram	rs1 immediate	rs1 ← R[immediate]
STOREFIX	0101 0100	Stocke dans la ram	rs1 immediate	R[immediate] ← rs1