Conception and realization of the VIVACE architecture

Projet de Système digital

A.Auvolat, E.Enguehard, J.Laurent

16 janvier 2014

The VIVACE[1] architecture is a minimalistic 16 bits RISC microprocessor architecture, largely inspired by the MIPS microprocessor.

The principal characteristics of the architecture are :

— *8 general-purpose registers*, which can hold 16 bits integers : `Z, A, B, C, D, E, F, G`

— *16 bit memory addressing*, enabling the CPU to use up to $64kb$ of memory.

In order to implement and run the architecture, the following programs have been written :

— *Netlist simulator* and *netlist optimizer*

— An *OCaml library* for generating netlists from Caml code

— The code for the *CPU implementation* (written in Caml)

— A *monitor* which is used to interact dynamically with the netlist simulator

— An *assembler* which can be used to produce the ROM files run by the CPU.

1 How to run the VIVACE cpu

1.1 Preparation

All the tools described in the introduction must first be compiled :

```
$ cd csim; make; cd ..
$ cd sched; make; cd ..
$ cd monitor; make; cd ..
$ cd asm; make; cd ..
```

To run the VIVACE CPU, type the following :

```
$ cd cpu; make
```

1.2 Monitor commands

You are now running the VIVACE CPU. The monitor accepts a few commands to control the simulation. First, you must configure the monitor to communicate with the CPU :

```
t 0
s 1 19 18
d7 20 21 22 23 24 25 26 27
```

The first command sets up the tick input (a tick is sent once every second on this input by the monitor). The second command sets up the serial input/output. The third command sets up the 7-segment display (8 digits displayed). Now, use the following commands to control the simulation :

1. Virtually Infaillible VIVACE Automated Computing Environment

— `a` run the simulation at full speed
— `m` run the simulation step by step (enter an empty command to run a step)
— `f <freq>` run the simulation at fixed frequency (frequency is dynamically ajusted so this is not very accurate)
— `q` exit simulation

The CPU recieves commands on the serial input. To send a command to the CPU, use the following syntax :

```
:<cpu_command>
```

For instance :

```
:Y2014
```

These commands are essentially used to set one of the six variables `YMDhms` ; the syntax is similar to the example command given above. An empty CPU command tells the CPU to just tell us what time and what day it is.

2 Program details

2.1 Generating netlists from Caml code

We have developped a library that enables us to easily generate netlists from Caml code. The Caml code we write has the same abstraction level that MiniJazz has, but it is more comfortable to write circuits like this than with MiniJazz code.

The library functions are defined in `cpu/netlist_gen.mli`. Basically, we have created functions that build the graph of logical operations. The abstract type `t` is actually a closure to a function that adds the required equation to a netlist program being built, therefore the generation of a netlist consists in two steps : the generation of a closure graph that describes the graph of logical operations, and the execution of these closures on a program which, at the beginning, has only the circuit inputs. The equations

2.2 Format des fichiers ROM

Un fichier ROM est composé d'une suite de blocs. Un bloc est composé d'un entête de la forme suivante

$$\texttt{ROM id ADDRSIZE } n_1 \texttt{ WORDSIZE } n_2$$

où les deux arguments numériques sont fournis en écriture décimale. Le contenu de la mémoire correspondant à un entête est décrit par une suite de valeurs ordonnées dans le sens des adresses croissantes. Il est possible de sauter directement à une adresse spécifiée avec la directive

```
ADDR addr :
```

Un exemple est fourni dans le dossier `samples` du programme.

2.3 Le language de scripts d'interaction

Après affichage de l'invite de commande `>`, l'utilisateur peut saisir plusieurs directives, séparées par des points ou des sauts de lignes. La directive la plus simple consiste à entrer une séquence de valeurs `val` séparées par des espaces, et correspondant à chaque entrée du circuit, dans l'ordre de leur déclaration au sein du fichier net-list. Le logiciel exécute alors un cycle machine à partir de ces entrées et affiche les nouvelles valeurs des sorties. Il est également possible de saisir une liste d'affectations suivant l'exemple suivant :

`a = 0, b = 0110, c = 10`

Les entrées su circuit qui n'auront pas été mentionnées dans la liste garderont leur valeur précédente. Il est également possible d'utiliser une des directives suivantes :

- `%inputs` $\texttt{id}_1$ `...` $\texttt{id}_n$ spécifie une nouvelle liste ordonnée des entrées du circuit
- `%outputs` $\texttt{cell}_1$ `...` $\texttt{cell}_n$ spécifie la liste des sorties du circuit
- `%show` $\texttt{cell}_1$ `...` $\texttt{cell}_n$ affiche la valeur des variables et emplacements RAM indiqués. En l'absence d'arguments, la valeur des sorties est affichée
- `%run n` exécute n cycles machine
- `%echo off` désactive l'affichage automatique des sorties après exécution d'un cycle. Pour réactiver cette fonctionnalité, remplacer `off` par `on`
- `%echo on` active l'affichage des sorties
- `%undo n` annule les n dernières commandes ayant eu pour effet d'exécuter des cycles machine. Si l'argument n'est pas spécifié, $n = 1$
- `%redo n` annule un appel anterieur de `%undo n`, si aucune opération d'écriture n'a été excutée d'ici là
- `%history n` redéfinit la valeur de la capacité de l'historique permettant d'effectuer des `undo`

Des exemples sont disponnibles dans le dossier `samples` du programme.

3 Carcatéristiques techniques

Une exécution du programme se déroule schématiquement de la manière suivante :

Au chargement d'un fichier net-list :
- Les équations sont triées dans l'ordre topologique et tout cycle d'exécution est signalé
- Une vérification de la cohérence des types est effectuée
- La liste d'équations est précompilée (voire suite)

A chaque cycle :
- Les valeurs des mémoires RAM sont mises à jour en fonction de l'état de la machine à la fin du cycle *précédent*. Le membre gauche d'une équation de type RAM ne dépend ainsi dans l'ordre topologique que du paramètre de l'adresse de lecture.
- Chaque équation modifie l'état courant sur la base de l'état de la machine à la fin du cycle précédent (calcul des valeurs des registres) et de l'état obtenu après exécution des équations antérieures (par rapport à l'ordre topologique)

Pour plus de détails, le lecteur est invité à consulter le fichier *netlist_ simulator.ml*.

3.1 Optimisations

Afin de rendre le programme raisonnablement efficace, deux optimisations ont été principalement mises en oeuvre :
- Chaque variable se voit affecter un identifiant numérique et l'état de toutes les variables de la machine est stockée dans un tableau. Cependant, le type `state` exporté par `Netlist_simulator` est persistent (le tableau est recopié à la fin de toutes les fonctions exportées par la signature du module), ce qui permet de gérer plutôt simplement un historique.
- La liste des équations est précompilée en une liste de fonctions du type `state -> state -> state` de manière à ce que la structure de ses éléments ne soit pas réexplorée à chaque cycle. Pour plus d'explications, consulter *netlist_ simulator.ml*.

Ces deux améliorations représentent un gain de temps d'un facteur 12 environ environ. A titre indicatif, la simulation de 84000 cycles du circuit à 300 variables `clockHMS` prend $3s$.

3.2 Améliorations possibles

Le système actuel génère des erreurs très imprécises et ne vérifie pas la sémantique des fichiers de ROM et des directives d'entrées.