Projet 2 de CA (2023-2024)
Un Mini-Javascript sur circuit FPGA (version 1)

Loïc Sylvestre *

28 avril 2024

Le langage Eclat présenté en cours est un langage fonctionnel-impératif, parallèle et synchrone pour programmer des circuits. Les programmes Eclat sont compilés vers des descriptions de matériel VHDL pour être synthétisées sur une carte FPGA.

Comme exemple d'application Eclat réaliste, une version simplifiée de la machine virtuelle OCaml a été implémentée en Eclat [1], ce qui permet non seulement d'exécuter des programmes OCaml directement par un circuit spécialisé sur FPGA, mais aussi d'appeler depuis OCaml des fonctions externes Eclat (c'est-à-dire, des accélérateurs matériels) depuis les programmes OCaml.

Dans ce projet, on vise à reproduire cette expérience avec le langage microJS, un mini-langage fonctionnel-impératif à la Javascript. Le travail demandé est de réimplémenter en Eclat la machine virtuelle de l'ancien cours de Compilation de 3I018 [1]. Il s'agit d'une machine à pile appelée VM3I018, qui exécute du bytecode obtenu par compilation de MicroJS.

1 Modalité de rendu

Le rendu du projet se fera sur Moodle. La date limite est fixée au **dimanche 12 mai à 23h59**. Il est demandé un rendu par étudiant (rapport personnel + solution au projet) mais vous pouvez developper votre solution en binôme (en l'indiquant dans les deux rapports). Le rapport comportera entre 3 à 6 pages. Il devra indiquer ce qui est fonctionnel, ce qui ne l'est pas, ainsi que les éventuels choix d'implantation et optimisations [2].

2 Code fourni

Une archive avec le squelette du projet et disponible sur Moodle. Il contient :

- un dossier `Eclat` avec d'une part un sous-dossier `eclat-compiler` (c'est le compilateur Eclat) et d'autre part un sous-dossier `vm` (c'est le squelette de la machine VM3I018 à compléter),
- un dossier `MicroJS` qui contient le compilateur MicroJS reciblé pour embarqué du bytecode dans un tableau Eclat. L'executable (construit avec `make`) comporte une option `-eclat` pour embarqué le bytecode VM3I018 dans un tableau Eclat,
- à toute fin utile, un dossier `NativeVM` contenant les sources originaux de l'implantation C de la machine VM3I018.

*loic.sylvestre@lip6.fr

1. Des supports utiles sont disponibles à cette adresse (cours 8, 9 et 10) :
`http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2015/ue/3I018-2016fev/Cours/`

2. Le but étant que la machine fonctionne, mais aussi qu'elle exécute les instructions de bytecode le plus rapidement possible (en nombre de cycles d'horloge).

3 Sécification de la machine

VM3I018 est une machine virtuelle relativement simple. Elle comporte des registres et seulement 12 instructions. Les registres principaux sont :

- `sp` est le pointeur de pile,
- `env` est l'environnement lexical courant,
- `pc` est le pointeur de code,
- `fp` est un pointeur vers le bloc d'activation suivant.
- `gp` est le pointeur sur la dernière variable globale allouée avec `GALLOC`
- `hp` (*heap pointer*) et un pointeur vers la fin du tas
- `write_buf` (qui est *optionnel*) est un *buffer d'écriture postée* pouvant être utilisé pour optimiser l'écriture dans le segment des variables globales.
- `finished` est un registre qui vaut `true` si l'exécution se termine. Pour ce projet, on propose que lorsque la pile est vide après un `POP`, la valeur dépilée est affichée et l'exécution s'arrête.

Pour implanter l'appel de fonction, l'état de la machine comprend une structure de données appelée bloc d'activation (ou *stack frame*). C'est un quadruplet `(sp,env,pc,fp)` qui contient les valeurs courantes des registres `sp`, `env`, `pc`, et un pointeur `fp` vers le bloc d'activation suivant (c'est à dire, celui de la fonction appelante).

- `PUSH` v empile la valeur v,
- `POP` dépile une valeur et l'afficher si la pile obtenue est vide,
- `GALLOC` réserve de l'espace pour une nouvelle variable globale,
- `GSTORE` i dépile une valeur et l'assigne à la i-ème variable globale,
- `GFETCH` i lit la i-ème variable globale et l'empile,
- `JUMP` ℓ réalise un branchement inconditionnel à l'adresse ℓ
- `JTRUE` ℓ dépile une valeur v, puis réalise un branchement conditionnelle à l'adresse ℓ si v est le booléen `true`,
- `JFALSE` ℓ dépile une valeur v, puis réalise un branchement conditionnelle à l'adresse ℓ si v est le booléen `false`,
- `CALL` n dépile une valeur v ; puis :
  - si v est une primitive (par exemple `+`), alors n valeur sont dépilées et passées en arguments de la primitive, puis le résultat est empilé,
  - si v est une fermeture, alloue dans le tas un bloc de taille $n + 1$ dans lequel stocker l'environnement (`env`) et les valeurs des n arguments dépilés, puis sauvegarder les valeurs des registres dans le bloc d'activation courant, puis allouer un nouveau bloc d'activation, puis sauter à l'adresse du code de la fermeture avec comme environnement courant, l'environment de la fermeture.
- `PUSH FUN` ℓ crée une fermeture $\langle \ell | \texttt{env} \rangle$ qui enclôt un pointeur sur le code ℓ et l'environnement courant `env`,
- `STORE` n dépile une valeur et l'assigne à la i-ème variable locale (dans l'environnement)
- `GFETCH` i lit la i-ème variable locale et l'empile,
- `RETURN` dépile une valeur v, restore le bloc d'activation de l'appelant (*i.e.,* les valeurs des registres `sp`, `env`, `pc` et `fp`), et empile v.

4 Description du code fourni

On propose d'implanter la VM VM3I018 en Eclat de la façon suivante (d'autres choix sont possibles). Le bytecode est représenté comme un tableau d'instructions de taille fixe n (vous pouvez choisir la valeur de n).

```
let code = array_create n;;
```

Pour représenter les instructions de la machine VM3I018, on utilise un type somme Eclat [3]. Cela permet d'avoir "gratuitement" une union discriminante comme dans la version C. Notez qu'en Eclat, tous les constructeurs doivent être paramétrés.

```
1  type ptr = int<32>
2  type long = int<32>
3
4  type prim = P_ADD of unit
5         | P_SUB of unit
6         | P_MUL of unit
7         | P_EQ of unit
8         | P_LT of unit    (* à compléter *)
9
10 type value = Bool of bool
11            | Int of long
12            | Nil of unit
13            | Prim of prim    (* à compléter *)
14
15 type instr = I_GALLOC of unit
16            | I_GSTORE of ptr
17            | I_GFETCH of ptr
18            | I_STORE of ptr
19            | I_FETCH of ptr
20            | I_PUSH of value
21            | I_PUSH_FUN of ptr
22            | I_POP of unit
23            | I_CALL of long
24            | I_RETURN of unit
25            | I_JUMP of ptr
26            | I_JTRUE of ptr
27            | I_JFALSE of ptr
```

Le type des valeurs devra être complété, notamment pour représenter les fermetures. L'exécution du bytecode est réalisée par une fonction récursive terminale `vm_run_code` qui est fournie. Cette fonction exécute les instructions de bytecode une à une en appelant la fonction `vm_run_instr`. L'état de la VM (notamment les registres) est passé en paramètre de `vm_run_instr`.

3. En Eclat, les types sommes ne sont pas récursifs et doivent être monomorphes. On ne peut pas définir un type liste par exemple, mais on peut définir un type `int_option = Some of int | None of unit`

Références

[1] Loïc Sylvestre, Jocelyn Sérot, and Emmanuel Chailloux. Hardware implementation of ocaml using a synchronous functional language. In *International Symposium on Practical Aspects of Declarative Languages*, pages 151–168. Springer, 2024.

Annexe A : Exemple de compilation de MicroJS

Le programme MicroJS suivant définit une fonction `inc` puis appel cette fonction avec l'argument 42.

```
1 var s = 0;
2 var i = 0;
3 while (i < 10) {
4   i = i + 1;
5   s = i + s;
6 }
```

Le bytecode obtenu par compilation de ce programme est :

```
  GALLOC
  PUSH INT 0
  GSTORE 0
  GALLOC
  PUSH INT 0
  GSTORE 1
  JUMP L1
L2:
  PUSH INT 1
  GFETCH 1
  PUSH PRIM 0
  CALL 2
  GSTORE 1
  GFETCH 0
  GFETCH 1
  PUSH PRIM 0
  CALL 2
  GSTORE 0
L1:
  PUSH INT 10
  GFETCH 1
  PUSH PRIM 6
  CALL 2
  JTRUE L2
```

Annexe B : Syntaxe d'Eclat

La syntaxe d'ECLAT est définie sur la figure 1. Les fonctions doivent être récursives terminales. La construction **let** $p_1 = e_1$ **and** $p_1 = e_2$ **in** e calcule e_1 et e_2 en parallèle, puis calcule e. La construction `array_create` n créée un tableau de taille n non initialisé. Chaque accès mémoire (x`.(`e`)` et x`.(`e`)` $\leftarrow e$) prend deux cycles. Un verrou associé à chaque tableau empêche l'accès simultanée au tableau : un seul prend le verrou et les autres attendent. L'ordre des accès suit l'ordre d'évaluation d'ECLAT qui va de la gauche vers la droite. Le langage est en appel par valeurs.

programme	π	::=	d_1`;;` $\cdots$ d_n`;;`
déclaration globale	d	::=	**let** $x = e$
		|	**let** f $p = e$
		|	**let rec** f $p = e$
		|	`type` $x = \tau$
		|	`type` $x =$ A_1 **of** τ_1 `|` $\cdots$ A_n **of** τ_n
type	τ	::=	`int<`n`>` | `bool` | `unit` | `string<`n`>` | τ `array<`n`>` | `(`τ `*` τ`)`
expression	e	::=	c | x | e e | `(`e`,` $\cdots$ e`)` | op | **let** $p = e$ **in** e
		|	**if** e **then** e **else** e | **let** f $p = e$ **in** e
		|	A e | **match** e **with** A_1 $p_1 \rightarrow e_1$ `|` $\cdots$ A_n $p_n \rightarrow e_n$ (`|` `_` $\rightarrow e$)? **end**
		|	**let rec** f $p = e$ **in** e | **let** $p = e$ **and** $p = e$ **in** e
		|	**reg** (**fun** $p \rightarrow$ e) **last** e
		|	`array_create` n | x`.(`e`)` | x`.(`e`)` $\leftarrow e$ | x`.length`
motif	p	::=	x | `()` | `(`p`,` $\cdots$ p`)`
constante	c	::=	`true` | `false` | n | `()` | `(`c`,`c`)` | *"str"*
fonction primitive	op	::=	`+` | `-` | `*` | `mod` | `/` | `abs` | `not` | `or` | `&` | `xor`
		|	`=` | `<` | `<=` | `>` | `>=`
		|	`print_int` | `print_string` | `print_newline`

FIGURE 1 – Syntaxe d'ECLAT

Annexe C : lignes de commandes de MicroJS à VHDL

```
$ cd MicroJS
$ make
$ ./microjs tests/inc.js -compile
...
  GALLOC
  JUMP L2
L1:
  PUSH INT 1
  FETCH 0
  PUSH PRIM 0
  CALL 2
  RETURN
  PUSH UNIT
  RETURN
...
$ ./microjs tests/inc.js -eclat
...
(* code généré par le compilateur MicroJS *)
let load_bytecode () =
  code.(0) <- I_GALLOC();
```

```
  code.(1) <- I_JUMP 9;
  code.(2) <- I_PUSH (Int 1);
  code.(3) <- I_FETCH 0;
  code.(4) <- I_PUSH (Prim (P_ADD()));
  code.(5) <- I_CALL (2);
  code.(6) <- I_RETURN();
  code.(7) <- I_PUSH (Nil());
  code.(8) <- I_RETURN();
  code.(9) <- I_PUSH_FUN (2);
  code.(10) <- I_GSTORE 0;
  code.(11) <- I_PUSH (Int 42);
  code.(12) <- I_GFETCH 0;
  code.(13) <- I_CALL (1);
  code.(14) <- I_POP();
  () ;;
(* ====================================== *)

$ cd ../Eclat/
$ cd eclat-compiler
$ make
$ cd ../vm
$ make DEBUG=true
vhdl code generated in ../target/main.vhdl
testbench generated in ../target/tb_main.vhdl for software RTL simulation using GHDL.
$ make simul NS=4000    # NS est le nombre de nanosecondes
START (cy=0)
[sp:0|env:0|pc:0|fp:0]|gp:0|hp:0
todo :-)
(exit) bye bye.
```

Annexe D : Remarques diverses

- Le Makefile du compilateur ECLAT (`Eclat/eclat-compiler/Makefile` doit être modifier légèrement pour fonctionner sur les machines de la PPTI : il suffit de remplacer l'exécutable `menhir` par `/users/nfs/Enseignants/chaillou/.opam/4.13.1/bin/menhir`.
- l'initialisation du bytecode se fait par l'appel `load load_bytecode` dans la fonction `main` (fichier `main.ecl`).
- Il y a un GC dans l'implémentation C de la machine VM3I018. Ce projet 2 se concentre sur l'implantation d'un interprète de bytecode. On peut se contenter d'allouer des valeurs dans le tas sans jamais les récupérer. Ecrire un GC en ECLAT serait encore mieux !
- en ECLAT, chaque accès à un tableau prend 2 cycles d'horloge. Cependant, les accès peuvent se faire en parallèle dans des tableaux différents. C'est plus efficace... En particulier :
  - c'est à cela que servira le *buffer d'écriture postée* (dans l'état de la machine) pour l'assignation de variables globales,
  - la machine fait de nombreux dépilements ; on pourrait faire un dépilement *spéculatif* en parallèle de la lecture de l'instruction dans le tableau de byteocde, ce qui améliorait sensiblement les performances – attention toutefois à bien gérer le cas où la pile est vide !