Un Mini-Javascript sur circuit FPGA (version 0)

Loïc Sylvestre *

25 mars 2024

Javascript, le langage de programmation WEB bien connu, repose sur un noyau fonctionnel-impératif inspiré du langage Scheme. L'ancien cours de Compilation de L3 3I018 [1] avait pour objet la compilation de ce noyau, appelé MicroJS, vers du code octet (bytecode) pour une machine à pile appelée VM3I018.

Par exemple, 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
```

*loic.sylvestre@lip6.fr

1. Des supports utiles sont disponibles à cette adresse `http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2015/ue/3I018-2016fev/Cours/`

VM3I018 est une machine virtuelle relativement simple : elle ne comporte que 12 instructions. Par exemple :

- `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'assigner à la i-ème variable globale,
- `GFETCH` i lit la i-ème variable globale,
- `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 ; si v est une primitive (par exemple `+`), alors n valeur sont dépilées et passées en arguments de la primitive, puis le résultat est empilé.

Les autres instructions (`PUSH FUN` ℓ, `STORE`, `FETCH` et `RETURN`, et `CALL` n avec en sommet de pile une valeur fonctionnelle) seront présentées ultérieurement.

Une implémentation C de la machine VM3I018 a été présentée au cours 8. Vous avez aussi utilisé C en TME et dans le projet Mark&Compact pour implanter des traits de programmation de haut-niveau (des structures de données dynamiquement allouées, par exemple).

L'objet de ce projet est l'implantation de la machine VM3I018 dans le langage Eclat, présenté au cours 7. Eclat est un noyau de langage déclaratif inspiré d'OCaml et compilé vers des descriptions de matériel VHDL pour reconfigurer des circuits FPGA.

Ce projet sera noté et couvrira les dernières séances de TME (8, 9 et 10).

Pour réaliser ce projet, vous trouverez sur Moodle une archive avec :

- un dossier `Eclat` qui contient un sous-dossier `eclat-compiler` (c'est le compilateur Eclat ainsi qu'un sous-dossier `vm` (c'est un squelette d'implémentation 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.

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.

On propose d'implanter la VM VM3I018 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;;
```

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** $A_1\ p_1 \rightarrow e_1$ | $\cdots$ $A_n\ p_n \rightarrow e_n$ (| _ $\rightarrow e$)?
		|	**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

Pour représenter les instructions de la machine VM3I018, on utilise un type somme ECLAT [2]. 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   (* et ceteta *)
9
10 type value = Bool of bool
11           | Int of long
12           | Nil of unit
13           | Prim of prim   (* et ceteta *)
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.

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

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, passé en paramètre de `vm_run_instr` comporte plusieurs éléments :

- un bloc d'activation (ou *stack frame*) : c'est un quadruplet `(sp,env,pc,fp)` où `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` est un *buffer d'écriture postée* qui pourra être utilisé pour optimiser l'écriture dans le segment des variables globales.
- `finished` est un registre qui vaut `true` si l'exécution se termine (cas d'une instruction `POP` avec une pile de 1 élément, notamment).

Voici les commandes pour utiliser les différents outils :

```
$ 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.
```

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 VM) 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 !