Macle : un langage dédié à l'accélération de programmes OCaml sur circuits FPGA

Loïc Sylvestre, Jocelyn Sérot, Emmanuel Chailloux

Macle : un langage dédié à l'accélération de programmes OCaml sur circuits FPGA

Loïc Sylvestre[1], Jocelyn Sérot[2] et Emmanuel Chailloux[1]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
loic.sylvestre@lip6.fr
emmanuel.chailloux@lip6.fr
[2] Institut Pascal, UMR 6602 UCA/CNRS/SIGMA
jocelyn.serot@uca.fr

Résumé

Les circuits reprogrammables de types FPGA constituent un matériel de choix pour la réalisation d'accélérateurs de calculs. L'implémentation O2B de la machine virtuelle OCaml permet d'appeler des circuits externes réalisés sur FPGA depuis un langage de haut niveau (OCaml) compilé en bytecode. La conception de circuits s'appuie sur des langages de description de matériel (HDL) souvent fort éloignés des langages algorithmiques. C'est pourquoi l'on présente Macle, un langage applicatif dédié à la programmation de calculs séquentiels et parallèles synthétisables en circuits. On décrit la chaîne de compilation de Macle vers un HDL et son intégration à O2B.

1 Introduction

Contexte Les FPGA (*Field-Programmable Gate Array*) sont des circuits composés d'un grand nombre [1] de cellules logiques configurables dont les interconnexions peuvent être elles-mêmes re-programmées. Ce type de matériel fait l'objet d'un intérêt croissant de la part des programmeurs, notamment d'applications embarquées à performance ou sûreté critique, puisqu'il permet la réalisation d'architectures spécifiques modifiables. Mais programmer le matériel est difficile : les détails d'implantation rendent les codes sources complexes et sujets aux erreurs de programmation ou de spécification [2]. La programmation efficace de ce type de circuit passe par l'usage de langages de description de matériel (*Hardware Description Languages*, HDL) comme VHDL ou Verilog [13]. Ces langages décrivent les fonctions logiques à réaliser au niveau dit transfert de registre (*Register Transfer*, RT), c.à.d., essentiellement, des opérations élémentaires sur un ensemble de registres constituant l'état du programme. En somme, la description de circuits efficaces dans un HDL nécessite non seulement des connaissances en électronique numérique mais aussi l'expérience des outils de synthèse [3].

La programmation impérative dans des langages comme C, C++ ou Python, projetés en matériel via des outils de *synthèse haut niveau* (*High Level Synthesis*) comme OpenCL [5] permet de pallier en partie ces difficultés. Mais ces outils laissent peu de contrôle au programmeur quant à l'efficacité et la fiabilité du code ainsi engendré au niveau RT.

Les langages de programmation fonctionnels ont parfois été présentés comme une solution à ces limitations. Outre le niveau d'abstraction offert, la sémantique de ces langages s'accorde en

1. De quelques dizaines à plusieurs centaines de milliers.
2. Un circuit peut être synthétisable sans pour autant implémenter correctement la fonctionnalité souhaitée.
3. Chaque outil de synthèse supporte un sous-ensemble des descriptions exprimables dans un HDL. Maîtriser la syntaxe et la sémantique (complexe) d'un HDL ne suffit pas : il faut comprendre de quelle manière une description est synthétisée.

effet mieux au modèle d'exécution intrinsèquement concurrent du niveau RT [6]. En pratique, plusieurs approches ont été suivies :

1. Embarquer, dans un langage fonctionnel généraliste comme Haskell, un langage dédié (*Domain Specific Language*, DSL) permettant la description de matériel. On peut citer Lava [2], Hawk [9] et Cλash [1] en Haskell, HardCaml en OCaml [7], Kami [3] en Coq. Le programmeur bénéficie alors des traits de haut niveau du langage hôte pour spécifier des circuits.
2. Compiler un langage fonctionnel éventuellement restreint vers des FPGA. C'est l'approche suivie, par exemple, par le langage SAFL [10](*Statically Allocated Parallel Functional language*), le langage Hume [15](*Higher-order Unified Meta-Environment*), le compilateur SHard [12] pour un sous-ensemble de Scheme et le projet FHW [16,19] pour Haskell.
3. Implémenter une machine virtuelle pour un langage fonctionnel ciblant des FPGA. C'est l'approche suivie par O2B [14] (OCaml *on Board*), une adaptation d'OMicroB [17] sur le processeur *softcore* Nios2 d'Intel (ex-Altera). Pour « tailler sur mesure » le matériel et ainsi programmer des applications hautement optimisées, O2B propose une approche *hybride* via l'ajout de primitives permettant l'appel des fonctions externes directement implémentées au niveau RT, depuis des programmes OCaml.

Cet article présente Macle (*autoMata-based AppliCative LanguagE*), un langage dédié à l'accélération de programmes OCaml sur FPGA. Macle permet de programmer des circuits utilisables comme fonctions externes en OCaml via O2B.

Plan Après une présentation de l'approche « machine virtuelle » suivie par O2B (section 2), cet article définit le langage Macle (section 3), donne une vue d'ensemble de la chaîne de compilation Macle vers un HDL (section 4) puis décrit ses langages intermédiaires (section 5) ainsi que l'interopérabilité entre Macle et OCaml (section 6). Suivent des tests de performance pour valider notre démarche (section 7), une présentation de travaux connexes (section 8) et une conclusion qui suggère des pistes de travaux futurs (section 9).

2 La plateforme O2B

La plateforme expérimentale O2B (OCaml *on Board* [14]) est une adaptation de l'implémentation OMicroB [17] de la machine virtuelle OCaml pour cibler des processeurs *softcore* réalisés sur circuits FPGA. Écrite en C, O2B est étendue de manière à pouvoir exécuter des fonctions externes implantées directement en matériel sous forme de cellules logiques. De telles fonctions – que nous appelons *circuits*[4] – ont alors accès à l'ensemble des dispositifs d'entrée-sortie d'un FPGA[5] et peuvent réaliser des calculs de manière très efficace en tirant parti du parallélisme massif disponible sur celui-ci et du fait que ces calculs peuvent être cadencés directement par l'horloge de base du FPGA.

La communication entre les circuits et le bytecode OCaml interprété par O2B se fait via un ensemble de registres dédiés associés au FPGA et *mappés* dans la mémoire du processeur *softcore*. Il est aussi possible de partager directement de la mémoire entre le processeur et les circuits.

4. Ce sont des *custom components* dans la terminologie des outils de développement Intel.
5. Elles peuvent ainsi lire la valeur d'un capteur ou activer une LED par exemple.

Le flot de compilation permettant d'engendrer la configuration complète du FPGA correspondant à un programme source est décrit sur la figure 1 dans le cas d'un FPGA de la famille Intel [6]. Le bytecode d'un programme OCaml – engendré par le compilateur `ocamlc` – est transformé en tableau C via l'outil `bc2c` de l'environnement OMicroB. Ce tableau est embarqué dans le programme C implémentant l'interpréteur de bytecode et la bibliothèque d'exécution (*runtime*) d'OMicroB. Ce programme est associé aux fonctions du *Board Support Package* donnant accès aux ressources matérielles de la carte cible ; l'ensemble étant compilé en un code exécutable par le processeur *softcore* (Nios2 dans le cas des FPGA Intel).

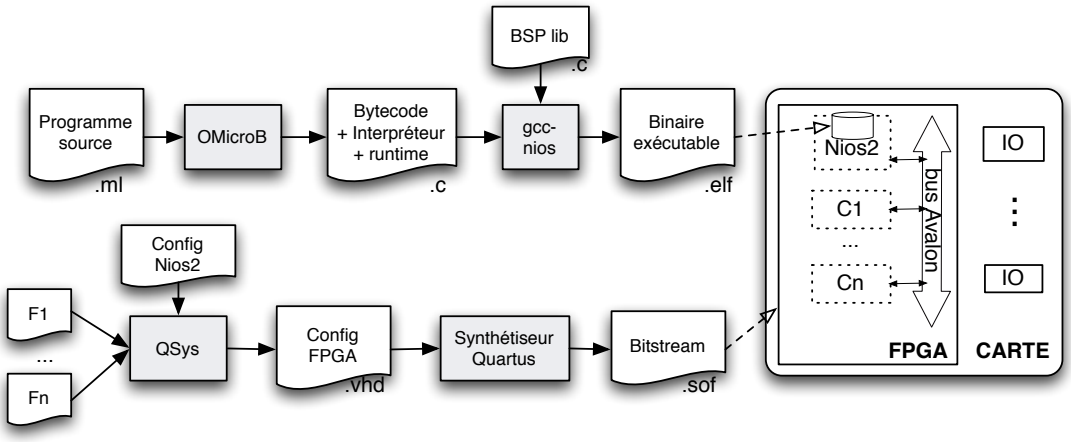


Figure 1 – Flot de compilation O2B vers une cible Intel

La configuration complète du FPGA comprend l'architecture exacte du processeur *softcore* utilisé, la description des fonctions externes F1 ··· Fn devant être implantées sous la forme de circuits C1 ··· Cn ainsi que, le cas échéant, le nombre et la nature des périphériques associés. Techniquement, cette étape de configuration est assurée par l'outil QSys de la chaîne Quartus d'Intel. Elle engendre un ensemble de fichiers VHDL qui constituent la description de la plateforme matérielle. C'est cette description – une fois synthétisée par les outils de la chaîne Quartus pour produire un fichier de configuration (ou *bitstream*) – qui est utilisée pour re-configurer le FPGA.

La version d'O2B présentée dans [14] laissait au programmeur le soin de spécifier lui-même des circuits dans un langage de description de matériel (VHDL). L'étape de configuration sous QSys, mais aussi la réalisation d'une interface de communication entre ces circuits et l'environnement d'exécution OCaml étaient effectuées manuellement. Dans la version courante d'O2B, les descriptions des circuits en VHDL ainsi que l'interface de communication sont engendrées depuis un langage de programmation dédié (Macle). L'étape de configuration est elle aussi automatisée par un ensemble de scripts : la génération du *bitstream* étant devenue entièrement automatique, la programmation FPGA est ainsi mise à la portée des programmeurs sans expérience préalable en description de matériel, en vérification voire en synthèse de circuits.

6. Seule famille actuellement supportée par O2B – mais l'approche suivie permet de cibler facilement d'autre familles de circuits (Xilinx par exemple).

3 Le langage Macle

Macle est un langage dédié à la programmation de circuits FPGA. Il permet d'enrichir la bibliothèque d'exécution d'un langage algorithmique comme OCaml en y ajoutant des fonctions externes automatiquement accélérées en matériel. Ces fonctions, comme le circuit qui calcule le n-ième terme de la suite de Fibonacci ci-dessous, sont définies par le programmeur dans un style applicatif.

```
circuit fibonacci n =
  let rec f a b n =
    if n < 1 then a else f b (a + b) (n - 1)
  in f 0 1 n
```

3.1 Motivation

Les fonctions Macle sont compilées sous forme de circuits spécifiés dans un langage de description de matériel classique [7]. Nous souhaitons que ces circuits soient efficaces et facilement utilisables comme fonctions externes depuis des programmes OCaml exécutés par O2B. Naturellement, cela conduit à des choix dans la conception du langage.

Macle est un langage strict en appel par valeur avec une syntaxe *à la ML*. Par construction, les circuits engendrés à partir de Macle n'allouent pas de mémoire et n'utilisent pas de pile d'appels. Schématiquement, la compilation de Macle vers VHDL expanse (*i.e.* intègre) les applications de fonctions, transforme en automates les définitions de fonctions mutuellement récursives terminales et implémente de façon parallèle les liaisons locales (*let ... and ...*) ainsi que les applications d'opérateurs, sous forme de barrières de synchronisation.

Ce schéma de compilation, visant à produire des circuits efficaces, restreint cependant l'expressivité du langage. D'une part, les récursions non terminales sont rejetées statiquement. D'autre part, les fonctions Macle ne sont pas des valeurs à part entière ; elles peuvent seulement être appliquées ou passées en paramètres à d'autres fonctions. Le système de types de Macle est monomorphe. Il garantit que la compilation d'un circuit bien typé produit une description VHDL synthétisable.

3.2 Syntaxe

La syntaxe du noyau de Macle explicitement typé [8] est définie figure 2. Nous appelons $\mathcal{X}$ un ensemble dénombrable de noms dénotés par les lettres x et f. Un *circuit* est une définition de fonction globale dont le corps est une expression. Une expression est une variable x, une constante c, une application d'opérateur ou de fonction, une conditionnelle ou une définition locale. Chaque valeur est typée avec un type de base τ. Un type σ est soit un type de base, soit un type fonctionnel $\sigma \to \sigma$. Chaque fonction est typée avec un type fonctionnel. Chaque définition de fonction B comporte un nom x, une séquence d'arguments w, un type de retour τ et un corps e. Les fonctions peuvent prendre en paramètres des fonctions. Une analyse statique sur la syntaxe des expressions empêche la définition de fonctions récursives non terminales. La classe des opérateurs binaires (*resp.* unaires) comporte uniquement des fonctions combinatoires.

7. VHDL dans notre cas, mais rien ne s'oppose à la génération de Verilog.
8. Il correspond à l'arbre de syntaxe engendré par inférence des types à partir des programmes sources Macle non typés.

circuit	$\phi ::= \textsf{circuit}\ B$	*variable*	$x, f \in \mathcal{X}$
fonction	$B ::= f\ w : \tau = e$	*opérateur unaire*	$\ominus ::= - \mid \textsf{not}$
arguments	$w ::= (x_1 : \sigma_1) \cdots (x_n : \sigma_n)$	*opérateur binaire*	$\oplus ::= + \mid = \mid < \mid \cdots$
expression	$e ::= x$	*constante*	$c ::= \textsf{true} \mid \textsf{false} \mid n$
	$\mid c$	*type de base*	$\tau ::= \texttt{bool} \mid \texttt{int}$
	$\mid \ominus e$	*type*	$\sigma ::= \tau \mid \sigma \to \sigma$
	$\mid e_1 \oplus e_2$		
	$\mid f\ e_1 \cdots e_n$		
	$\mid \textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3$		
	$\mid \textsf{let}\ x_1 : \tau_1 = e_1\ \textsf{and} \cdots x_n : \tau_n = e_n\ \textsf{in}\ e$		
	$\mid \textsf{let rec}\ B_1\ \textsf{and} \cdots B_n\ \textsf{in}\ e$		
	$\mid \textsf{let}\ B\ \textsf{in}\ e$		

Figure 2 – Syntaxe de Macle explicitement typé

3.3 Typage

Le typage de Macle est défini figures 3 et 4 par un ensemble de règles. L'environnement de typage Γ (*resp.* l'environnement de typage des opérateurs Δ) comporte des liaisons de la forme $\Gamma(x) = \sigma$ (*resp.* $\Delta(op) = \sigma$). Le jugement $\Gamma \vdash e : \tau$ signifie « dans l'environnement de typage Γ, l'expression e est bien typée, de type τ ». Le jugement $\vdash_{\textsf{circuit}} \phi : \tau$ signifie « le circuit ϕ est bien typé, de type τ ». Le jugement $\Gamma \vdash_{\textsf{arg}} e : \sigma$ signifie « dans l'environnement de typage Γ, l'argument e est bien typé, de type σ ».

L'environnement vide est noté $\varnothing$. L'extension $\Gamma[x : \sigma]$ de l'environnement Γ avec le nom x de type σ est définie par $\Gamma[x : \sigma](x) = \sigma$ et $\Gamma[x : \sigma](y) = \Gamma(x)$ si $x \neq y$. L'abréviation $\Gamma[x_1 : \sigma_1,\ \cdots\ x_n : \sigma_n]$ désigne l'environnement $\Gamma[x_1 : \sigma_1] \cdots [x_n : \sigma_n]$. Étant donnée une suite d'arguments $w \triangleq (x_1 : \sigma_1) \cdots (x_n : \sigma_n)$, les abréviations $\Gamma[w]$ et $\texttt{fun-ty}(w, \tau)$ désignent respectivement l'environnement $\Gamma[x_1 : \sigma_1,\ \cdots x_n : \sigma_n]$ et le type $\sigma_1 \to \cdots \sigma_n \to \tau$.

Let
$$\frac{i \in \{1, \cdots n\} \qquad \Gamma \vdash e_i : \tau_i \qquad \Gamma[x_1 : \tau_1, \cdots x_n : \tau_n] \vdash e : \tau}{\Gamma \vdash \textsf{let}\ x_1 : \tau_1 = e_1\ \textsf{and} \cdots x_n : \tau_n = e_n\ \textsf{in}\ e : \tau}$$

Arg-Val
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{\textsf{arg}} e : \tau}$$

Arg-Fun
$$\frac{\Gamma(f) = \sigma_1 \to \sigma_2}{\Gamma \vdash_{\textsf{arg}} f : \sigma_1 \to \sigma_2}$$

Let-Fun
$$\frac{\Gamma[w] \vdash e : \tau \qquad \Gamma[f : \texttt{fun-ty}(w, \tau)] \vdash e' : \tau'}{\Gamma \vdash \textsf{let}\ f\ w : \tau = e\ \textsf{in}\ e' : \tau'}$$

App
$$\frac{\Gamma(x) = \sigma_1 \to \cdots \sigma_n \to \tau \qquad i \in \{1, \cdots n\} \qquad \Gamma \vdash_{\textsf{arg}} e_i : \sigma_i}{\Gamma \vdash x\ e_1 \cdots e_n : \tau}$$

LetRec
$$\frac{\begin{array}{c}\textsf{first-order}(\texttt{fun-ty}(w_i, \tau_i)) \\ \Gamma' = \Gamma[f_1 : \texttt{fun-ty}(w_1, \tau_1), \cdots f_n : \texttt{fun-ty}(w_n, \tau_n)] \\ i \in \{1, \cdots n\} \qquad \Gamma'[w_i] \vdash e_i : \tau_i \qquad \Gamma' \vdash e : \tau\end{array}}{\Gamma \vdash \textsf{let rec}\ f_1\ w_1 : \tau_1 = e_1\ \textsf{and} \cdots f_n\ w_n : \tau_n = e_n\ \textsf{in}\ e : \tau}$$

Circuit
$$\frac{\textsf{first-order}(\texttt{fun-ty}(w, \tau)) \qquad \varnothing[w] \vdash e : \tau}{\vdash_{\textsf{circuit}}\ \textsf{circuit}\ f\ w : \tau = e : \texttt{fun-ty}(w, \tau)}$$

Figure 3 – Typage de Macle (1)

Les règles Arg-Val, Arg-Fun et App vérifient le bon typage des applications de fonctions ; les arguments étant soit des expressions, soit des noms de fonctions. Un circuit est une fonction

globale typée dans un environnement de typage vide (règle CIRCUIT). La condition de bord $\mathsf{first\text{-}order}(\sigma_1 \to \cdots \sigma_n)$ dans les règles LETREC et CIRCUIT impose que chaque type σ_i soit un type de base. Il en résulte que les définitions bien typées de circuits et de fonctions récursives terminales ne prennent pas de fonctions en paramètres. Les autres règles sont standard.

$$\textsc{True}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathtt{bool}} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathtt{bool}} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Unop}\ \frac{\Delta(\ominus) = \tau_1 \to \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash \ominus\, e : \tau_2}$$

$$\textsc{If}\ \frac{\Gamma \vdash e : \mathtt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad \textsc{Binop}\ \frac{\Delta(\oplus) = \tau_1 \to \tau_2 \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

Figure 4 – Typage de MACLE (2)

Grâce à cette discipline de types, et en particulier les restrictions qu'elle impose quant à l'utilisation des fonctions, nous pouvons garantir que les techniques de compilation présentées dans les sections suivantes s'appliquent correctement à tout programme MACLE bien typé.

4 Compilation de MACLE

La chaîne de compilation MACLE est schématisée figure 5. Elle vise à produire, à partir d'une spécification haut niveau, la description d'un circuit en VHDL portable et synthétisable sur FPGA. Les fonctions MACLE sont d'abord typées, puis traduites dans un langage intermédiaire (VSML) dans lequel sont réalisées des optimisations. Suit un second langage intermédiaire (ESML) à partir duquel la description VHDL est engendrée puis synthétisée pour obtenir le fichier de configuration du circuit FPGA. Une interface de communication (FFI O2B) – fondée sur la couche d'interopérabilité existante entre C et OCAML – permet l'appel des circuits MACLE depuis du bytecode OCAML exécutable par O2B sur le même FPGA cible. À des fins de simulation sur PC, MACLE et VSML sont traduits en OCAML tandis qu'ESML est interprété par un évaluateur écrit en OCAML suivant la sémantique décrite en sous-section 5.2.

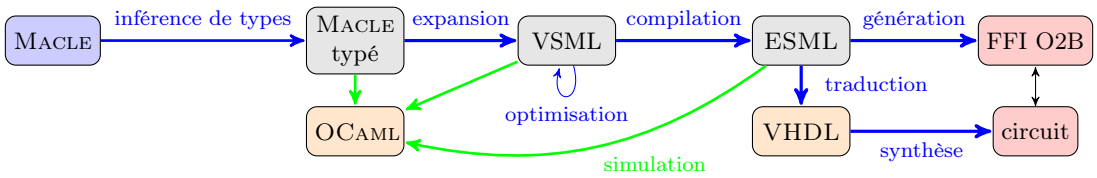


Figure 5 – Compilation de MACLE vers circuit

4.1 Exemple de compilation

La figure 6 illustre les étapes de traduction menant d'une fonction MACLE très simple (somme des carrés de 1 à n) à une description VHDL. Les langages intermédiaires VSML et ESML sont présentés plus en détails à la section 5. L'encodage de MACLE dans VSML transforme la fonction récursive locale S en une construction d'automate à états paramétrés,

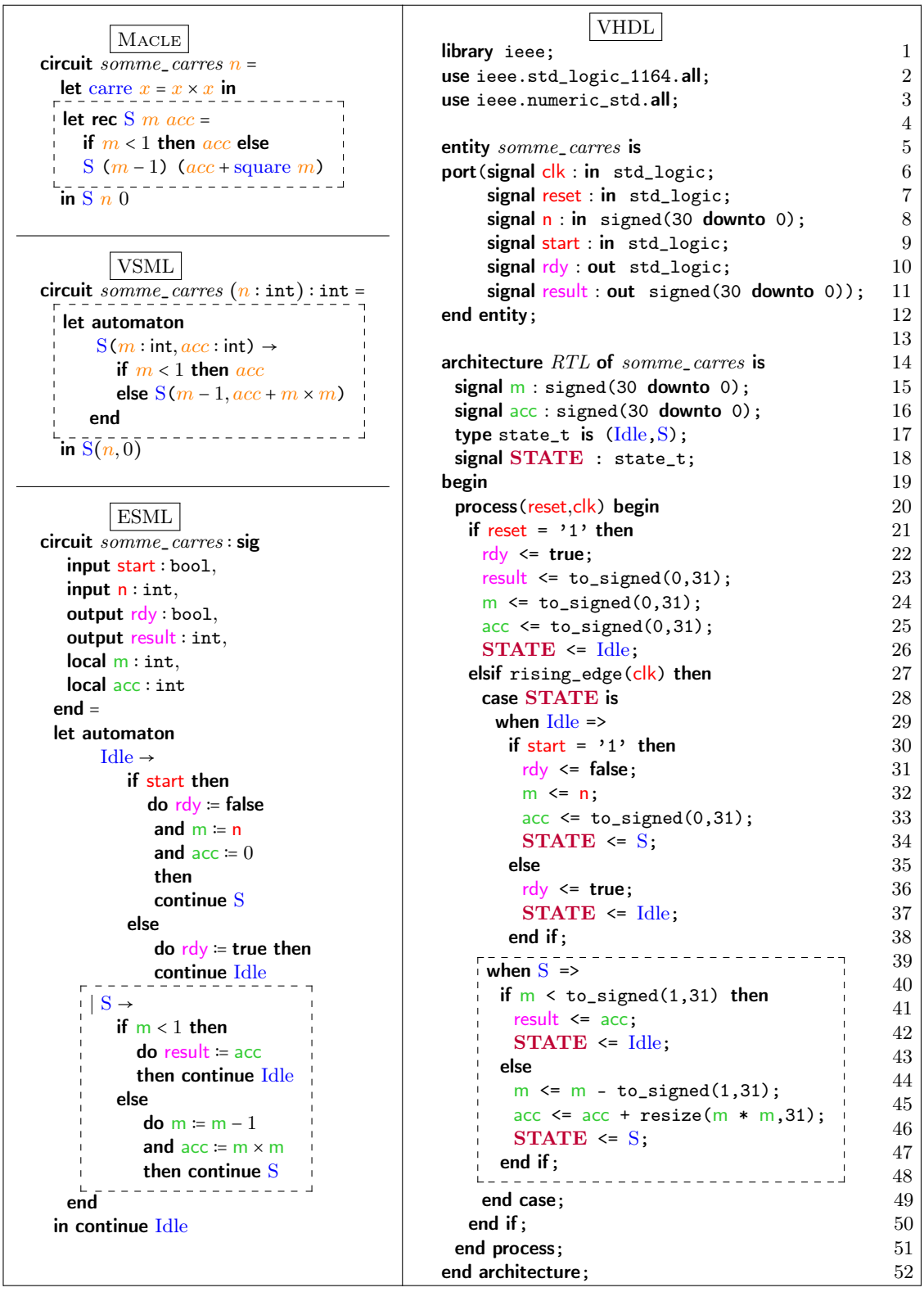


Figure 6 – Étapes de compilation d'un circuit MACLE vers VHDL

pouvant potentiellement se terminer et retourner une valeur. La compilation de VSML vers ESML remplace les états paramétrés par des variables « globales », explicite les terminaisons des calculs via un nouvel état « puit » Idle et ajoute des variables de synchronisation permettant l'appel (start) du circuit engendré et l'attente du résultat (rdy, result). ESML est finalement traduit en un sous-ensemble de VHDL synthétisable.

4.2 Codage en VHDL

L'encodage de l'automate en VHDL (dans le cadre droit) se fait, classiquement, sous la forme d'un *process* synchrone contrôlé par un signal d'horloge clk pour cadencer l'exécution du circuit et un signal rst pour initialiser l'automate (de manière asynchrone). Ce *process* manipule des signaux représentant d'une part les entrées, sorties et variables locales de l'automate ESML et d'autre part un signal **STATE** mémorisant l'état courant du système. Dans l'exemple de la figure 6, ce signal peut prendre deux valeurs : Idle et S, correspondant aux deux états de l'automate ESML. Les lignes 22 à 26 sur la figure décrivent ce qui se passe lors d'un *reset* asynchrone : les signaux rdy, result, m et acc sont initialisés et l'état de l'automate est positionné à la valeur Idle. Les lignes 28 à 49 décrivent ce qui se passe lors d'un front montant de l'horloge clk, front montant qui est la réalisation matérielle du « top » cadençant implicitement l'automate ESML. A chaque fois, en fonction de la valeur de l'état courant, un certain nombre de signaux sont modifiés [9], y compris, possiblement, celui codant l'état (*e.g.* **STATE** <= Idle). Un point fondamental est que ces modifications sont effectuées de manière *parallèle synchrone*. Lorsque plusieurs signaux sont modifiés lors d'une même activation du *process* (comme les signaux rdy, m, acc et **STATE** aux lignes 31 à 34, la mise à jour de ces signaux s'effectue non pas de manière séquentielle, mais simultanément pour l'ensemble des signaux concernés et à la fin de l'activation du *process*. Cela signifie en particulier que la valeur d'un signal apparaissant à droite du signe <= est toujours celle qu'a ce signal au début de l'activation, la nouvelle valeur éventuellement affectée à ce signal n'étant visible qu'à la *prochaine* activation du *process*. Dans l'exemple sus-cité, nous pouvons voir :

```
m <= m - to_signed(1,31);
acc <= acc + resize(m * m,31);
STATE <= S;
```

Ici, la valeur de m utilisée dans l'appel à `resize` ne dépend pas de la soustraction de la ligne précédente [10]. Cette sémantique particulière est liée à la manière dont les machines à états sont synthétisées en matériel [11]. C'est elle qui justifiera en particulier la sémantique opérationnelle donnée au langage ESML à la sous-section 5.2, notamment quant aux notions d'*environnement courant* et d'*environnement de sortie*.

5 Langages intermédiaires

La section précédente a donné une vue d'ensemble de la chaîne de compilation de Macle vers VHDL, illustrant le comportement des circuits au niveau RT à partir d'un exemple de fonction Macle qui – par étapes de traduction successives à travers deux langages intermédiaires (VSML et ESML) – est implémentée par un automate synchrone encodé en VHDL. Ces deux langages de

9. Via la syntaxe `<signal> <= <expression>`.
10. Et, de fait, l'ordre dans lequel ces deux lignes sont écrites est parfaitement indifférent.
11. Sous la forme d'un machine dite de Moore dans lequel l'état est codé dans un registre mis à jour à chaque front montant avec une valeur calculée à partir de sa sortie et des entrées.

machines à états sont des abstractions de MACLE pour VSML et de VHDL pour ESML. Après une présentation de la syntaxe de VSML on s'intéresse principalement à la définition d'une sémantique d'évaluation d'ESML pour comprendre le modèle d'exécution des circuits définis en MACLE. Par contre la traduction de VSML vers ESML est indiquée de manière succinte.

5.1 VSML

VSML (*Valued State Machine Language*) est un sous-ensemble de MACLE dans lequel toutes les applications de fonctions sont en position terminale. Les circuits MACLE sont mis sous cette forme par expansion complète des appels de fonctions à l'exception des fonctions mutuellement récursives. Les boucles récursives résultantes s'apparentent à des automates hiérarchiques avec états paramétrés *à la* LUCID SYNCHRONE [4].

La syntaxe de VSML est définie figure 7. Un circuit ϕ est une déclaration de fonction globale. Une expression e est soit un atome a, soit une structure de contrôle. La classe des atomes comprend les constantes, variables et applications d'opérateurs. VSML comporte quatre structures de contrôle : *application d'état*, conditionnelle, liaisons locales ou automate à états paramétrés. Les états sont dénotés par la lettre q.

circuit	$\phi ::= \textsf{circuit}\ f\ w : \tau = e$	*variable*	$x, q, f \in \mathcal{X}$
transition	$t ::= q\ w \to e$	*opérateur unaire*	$\ominus ::= - \mid \textsf{not}$
arguments	$w ::= (x_1 : \tau_1,\ \cdots\ x_n : \tau_n)$	*opérateur binaire*	$\oplus ::= + \mid\ =\ \mid\ <\ \mid \cdots$
expression	$e ::= a$	*constante*	$c ::= \textsf{true} \mid \textsf{false} \mid n$
	$\mid q(a_1, \cdots a_n)$	*type de base*	$\tau ::= \texttt{bool} \mid \texttt{int}$
	$\mid \textsf{if}\ a\ \textsf{then}\ e_1\ \textsf{else}\ e_2$		
	$\mid \textsf{let}\ x_1 : \tau_1 = e_1\ \textsf{and} \cdots x_n : \tau_n = e_n\ \textsf{in}\ e$		
	$\mid \textsf{let automaton}\ t_1 \mid \cdots t_n\ \textsf{end in}\ e$		
atome	$a ::= x \mid c \mid\ \ominus a \mid a_1 \oplus a_2$		

Figure 7 – Syntaxe de VSML

En VSML, les constructions de langage sont simples et référentiellement transparentes, facilitant la mise en œuvre d'optimisations de code. Le compilateur MACLE implémente d'une part une passe de propagation des atomes et d'autre part une forme de *let-floating* [11] visant à maximiser le degré de parallélisme.

5.2 ESML

ESML (*Extended State Machine Language*) est un langage parallèle synchrone qui intègre un nombre réduit de constructions de VHDL, en particulier la compositions d'automates synchrones et l'affectation.

La syntaxe d'ESML est définie figure 8. Un circuit Φ comprend une signature S et un corps P. La signature explicite les entrées, les sorties et les variables locales du circuit ainsi que leurs types respectifs. Le corps du circuit est une composition parallèle d'automates $A_1 \parallel \cdots A_n$ [8] tous cadencés par une même horloge globale. Chaque automate A est formé d'un ensemble de transitions et d'une instruction d'initialisation. Chaque transition t associe à un état source q une instruction.

L'instruction **continue** q suspend l'exécution de l'automate englobant jusqu'au top d'horloge suivant et fait passer celui-ci dans l'état q. L'instruction **do** $x_1 \coloneqq a_1$ **and** $\cdots x_n \coloneqq a_n$ **then** s lie les

circuit	$\Phi ::= \textsf{circuit}\ f : S = P$	*variable*	$x, q, f \in \mathcal{X}$
produit	$P ::= A \parallel P \mid A$	*opérateur unaire*	$\ominus ::= - \mid \textsf{not}$
automate	$A ::= \textsf{let automaton}\ t_1 \mid \cdots t_n\ \textsf{end in}\ s$	*opérateur binaire*	$\oplus ::= + \mid\ = \mid\ < \mid \cdots$
transition	$t ::= q \to e$	*constante*	$c ::= \textsf{true} \mid \textsf{false} \mid n$
instruction	$s ::= \textsf{continue}\ q \mid \textsf{if}\ a\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \mid \textsf{do}\ x_1 := a_1\ \textsf{and} \cdots x_n := a_n\ \textsf{then}\ s$	*signature*	$S ::= \textsf{sig}\ d_1, \cdots d_n\ \textsf{end}$
		déclaration	$d ::= m\ x : \tau$
atome	$a ::= x \mid c \mid \ominus a \mid a_1 \oplus a_2$	*mode*	$m ::= \textsf{input} \mid \textsf{output} \mid \textsf{local}$
		type de base	$\tau ::= \texttt{bool} \mid \texttt{int}$

Figure 8 – Syntaxe des circuits ESML

valeurs v_i des atomes a_i aux variables x_i dans l'instruction s (à la manière d'un *let ... and ...*), puis écrit $x_i := v_i$ dans l'environnement [12] au top d'horloge suivant. Un exemple de circuit ESML a été donné figure 6.

Les figures 9, 10 et 11 définissent la sémantique opérationnelle à réduction [18] d'ESML. Toute valeur ESML est soit une constante c, soit un glaçon $\#\langle A \rangle$ représentant l'état d'un automate A à la fin d'un instant synchrone. Un environnement d'exécution ρ comporte des liaisons de la forme $\rho(x) = v$. L'extension d'un environnement ρ est notée $\rho[x_1 \mapsto v_1, \cdots x_n \mapsto v_n]$ de même que précédemment. Un contexte d'évaluation E (*resp.* F) est un atome (*resp.* une instruction) avec un unique trou $[]$ dans lequel s'insère un atome. Un contexte d'évaluation M est un automate avec un unique trou $\langle\rangle$ dans lequel s'insère une instruction.

$$
\begin{array}{l}
v ::= c \mid \#\langle A \rangle \\
E ::= \ominus [] \mid [] \oplus a \mid v \oplus [] \\
F ::= \textsf{if}\ []\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \\
\quad \mid \textsf{do}\ x_1 := []\ \textsf{and}\ x_2 := a_2\ \textsf{and} \cdots x_n := a_n\ \textsf{then}\ s \\
\quad \mid \cdots \\
\quad \mid \textsf{do}\ x_1 := v_1\ \textsf{and} \cdots x_{n-1} := v_{n-1}\ \textsf{and}\ x_n := []\ \textsf{then}\ s \\
M ::= \textsf{let automaton}\ ts\ \textsf{end in}\ \langle\rangle
\end{array}
$$

Figure 9 – Valeurs et contextes d'évaluation ESML

La relation $\rho_0 \vdash a \longrightarrow a'$ signifie « dans l'environnement ρ_0, l'atome a se réduit en a' ». Elle est définie par la règle d'inférence Context-Atom (ou réduction sous un contexte d'évaluation E) ainsi que trois règles de réécriture : R-Unop, R-Binop et R-Var. L'application d'un opérateur $(\ominus\, v)$ (*resp.* $v_1 \oplus v_2$) se réduit *toujours* en une valeur, notée $[\![\ominus]\!](v)$ (*resp.* $[\![\oplus]\!](v_1, v_2)$).

La réduction d'une instruction s ou d'un automate A dans un environnement courant ρ_0 met à jour l'environnement ρ de l'instant synchrone suivant (ou *environnement de sortie*). La relation $\rho_0 \vdash s; \rho \longrightarrow s'; \rho'$ signifie « dans l'environnement courant ρ_0 l'instruction s et l'environnement de sortie ρ, se réduisent en s' dans l'environnement de sortie ρ' ». Elle est définie par la règle d'inférence Context-Instruction et trois règles de réécriture : R-If-True, R-If-False et R-Do. Intuitivement, l'instruction $\textsf{do}\ x_1 := v_1\ \textsf{and} \cdots x_n := v_n\ \textsf{then}\ s$ écrit $x_i := v_i$ dans

12. L'environnement est partagé en lecture. Chaque sortie ou variable locale ne peut être écrite que par un seul et même automate au cours de l'exécution du circuit. Les entrées ne peuvent pas être écrites. Le traduction de VSML en ESML supporte ces contraintes par construction.

l'environnement de sortie, puis calcule l'instruction s en substituant chaque occurrence de x_i par v_i (l'environnement courant étant laissé inchangé).

$$\textsc{Context-Atom}\ \frac{\rho_0 \vdash a \longrightarrow a'}{\rho_0 \vdash E[a] \longrightarrow E[a']} \qquad \textsc{Context-Instruction}\ \frac{\rho_0 \vdash a \longrightarrow a'}{\rho_0 \vdash F[a];\rho \longrightarrow F[a'];\rho} \qquad \textsc{Context-Automaton}\ \frac{\rho_0 \vdash s/\rho \longrightarrow s'/\rho'}{\rho_0 \vdash M\langle s\rangle;\ \rho \longrightarrow M\langle s'\rangle;\ \rho'}$$

$$\boxed{\rho_0 \vdash a \longrightarrow a'} \quad \rho_0 \vdash \ominus\ v \longrightarrow [\![\ominus]\!](v) \qquad \text{(R-Unop)}$$
$$\rho_0 \vdash v_1 \oplus v_2 \longrightarrow [\![\oplus]\!](v_1, v_2) \qquad \text{(R-Binop)}$$
$$\rho_0 \vdash x \longrightarrow \rho_0(x) \text{ si } x \in \mathsf{dom}(\rho_0) \qquad \text{(R-Var)}$$

$$\boxed{\rho_0 \vdash s;\rho \longrightarrow s';\rho'} \quad \rho_0 \vdash \textbf{if true then } s_1 \textbf{ else } s_2;\rho \longrightarrow s_1;\rho \qquad \text{(R-If-True)}$$
$$\rho_0 \vdash \textbf{if false then } s_1 \textbf{ else } s_2;\rho \longrightarrow s_2;\rho \qquad \text{(R-If-False)}$$
$$\rho_0 \vdash \textbf{do } x_1 \coloneqq v_1 \textbf{ and } \cdots x_n \coloneqq v_n \textbf{ then } s;\rho \qquad \text{(R-Do)}$$
$$\longrightarrow \{x_1 \mapsto v_1, \cdots x_n \mapsto v_n\}(s);\rho[x_1 \mapsto v_1, \cdots x_n \mapsto v_n]$$

$$\boxed{\rho_0 \vdash A;\rho \longrightarrow A';\rho'} \quad \rho_0 \vdash \textbf{let automaton } ts \textbf{ end in } (\textbf{continue } q);\rho \qquad \text{(R-Continue)}$$
$$\longrightarrow \#\langle \textbf{let automaton } ts \textbf{ end in } ts(q)\rangle;\rho$$

Figure 10 – Sémantique des instructions ESML

La relation $\rho_0 \vdash A;\rho \longrightarrow A';\rho'$ signifie « dans l'environnement courant ρ_0, l'automate A et l'environnement de sortie ρ se réduisent en A' dans l'environnement de sortie ρ' ». Elle est définie par la règle d'inférence Context-Automaton et la règle de réécriture R-Continue réalisant un branchement à l'état suivant de l'automate est l'attente (l'automate est gelé) jusqu'à l'instant synchrone suivant. Dans cette dernière règle, la notation $ts(q_i)$ représente l'instruction s_i telle que $ts \triangleq q_1 \to e_1 \mid \cdots q_n \to e_n$ et $i \in \{1, \cdots n\}$.

$$\textsc{Par-left}\ \frac{\rho_0 \vdash A;\rho \longrightarrow A';\rho'}{A \| P/\rho_0;\rho \longrightarrow A' \| P/\rho_0;\rho'} \qquad \textsc{Par-right}\ \frac{P/\rho_0;\rho \longrightarrow P'/\rho_0;\rho'}{v \| P/\rho_0;\rho \longrightarrow v \| P'/\rho_0;\rho'}$$

$$\boxed{P/\rho_0;\rho \longrightarrow P'/\rho_0';\rho'} \quad \#\langle A_1\rangle \| \cdots \#\langle A_n\rangle/\rho_0;\rho' \longrightarrow A_1 \| \cdots A_n \| P'/\rho';\rho' \qquad \text{(R-Clock)}$$

$$\boxed{\rho \vdash \Phi \longrightarrow P/\rho;\rho} \quad \textbf{circuit } f : \textbf{sig } m_1\ x_1 : \tau_1, \cdots m_n\ x_n : \tau_n \textbf{ end} = P/\rho_0 \longrightarrow P/\rho_0, \varnothing \qquad \text{(R-Circuit)}$$
$$\text{si } x_1, \cdots x_n \in \mathsf{dom}(\rho_0)$$

Figure 11 – Sémantique des circuits ESML

La relation $P/\rho_0;\rho \longrightarrow P'/\rho_0';\rho'$ signifie « le produit P dans l'environnement courant ρ_0 et l'environnement de sortie ρ se réduit en P' dans l'environnement courant ρ_0' et l'environnement de sortie ρ' ». Elle est définie par deux règles d'inférence Par-Left et Par-Right (qui réduisent le produit en laissant intact l'environnement courant) ainsi qu'une règle de réécriture R-Clock (qui réduit un produit totalement évalué $\#\langle A_1\rangle \| \cdots \#\langle A_1\rangle/\rho_0;\rho$ en un produit $A_1 \| \cdots A_n/\rho;\rho$) caractérisant le passage à l'instant synchrone suivant.

La relation $\rho \vdash \Phi \longrightarrow P/\rho;\rho$ signifie « dans l'environnement initial ρ, le circuit Φ se réduit en P dans l'environnement courant et de sortie ρ ». Elle est définie par la règle de réécriture R-Circuit qui vérifie que la signature S du circuit à réduire est compatible avec l'environnement initial ρ.

La compilation de VSML vers ESML nécessite d'aplatir la hiérarchie d'automates, de globaliser les paramètres des états et de transformer les liaisons locales en composition parallèle d'automates afin d'exploiter le parallélisme implicite des programmes Macle.

6 Interopérabilité entre Macle et OCaml

Pour faciliter le développement d'applications OCaml sur FPGA utilisant O2B, le compilateur Macle engendre non seulement une description de matériel, mais aussi l'ensemble des fichiers nécessaires à la synthèse du circuit et à son utilisation depuis du bytecode OCaml. Le format d'entrée du compilateur Macle supporte ainsi des applications mêlant déclarations Macle et OCaml, tels les deux exemples figure 12. Les déclarations de circuits Macle précèdent le programme OCaml. L'appel *print_int* en OCaml affiche un entier sur la sortie standard en utilisant le lien de communication série (UART) reliant le PC et le FPGA. Le programme OCaml figure 12a compose deux circuits Macle. Le circuit *somme_temps_de_vol4* figure 12b calcule la somme des temps de vol des suites de Syracuse comprises entre deux entiers. Ce calcul s'effectue en parallèle par paquets de 4 éléments à la fois.

```
circuit fact n =
  let rec f acc n =
    if n < 1 then acc else
    f (n × acc) (n − 1)
  in f 1 n

circuit gcd a b =
  let rec g a b =
    if a < b then g a (b − a) else
    if a > b then g (a − b) a
             else a
  in g a b

let n = 10;;

for i = 1 to n do
  print_int (gcd i (fact n))
done;;
```

(a)

```
circuit somme_temps_de_vol4 a b =
  let temps_de_vol n =
    let rec syracuse t n =
      if n ≤ 1 then t else
      if n mod 2 = 0 then syracuse (t + 1) (n/2)
                     else syracuse (t + 1) (3 × n + 1)
    in syracuse 0 n
  in
  let rec somme acc i =
      if i ≥ b then acc else
      if i > b − 4 then somme(acc + temps_de_vol i) (i + 1)
                   else somme (acc + temps_de_vol i
                                   + temps_de_vol (i + 1)
                                   + temps_de_vol (i + 2)
                                   + temps_de_vol (i + 3)) (i + 4)
  in somme 0 a

print_int (somme_temps_de_vol4 1 1024);;
```

(b)

Figure 12 – Programme OCaml étendu avec deux circuits Macle

Les circuits Macle sont paramétrés par des valeurs extérieures (entiers ou booléens) définies au niveau du programme OCaml. Dès lors, il est naturel d'étendre Macle avec de nouvelles constructions syntaxiques pour que ces circuits puissent déréférencer voire modifier physiquement des valeurs allouées dans le tas OCaml.

La figure 13 définit une extension de Macle avec filtrage de listes OCaml, accès et modification de références OCaml. Trois constructeurs de types (`unit`, τ `list` et τ `ref`) sont introduits. Les règles de typage MatchList, Get et Set sont standard.

La compilation de ces nouvelles constructions (listes et références) consiste d'une part à ajouter celles-ci à la syntaxe de VSML et d'autre part à introduire des opérateurs supplémentaires en ESML (sans effet de bord) pour manipuler les mots mémoire OCaml (e.g. consulter la taille d'un bloc). Les accès mémoire sont alors encodés en ESML à l'aide de variables de synchronisation réalisant des requêtes séquentielles sur un bus mémoire.

$$
\begin{array}{l}
e ::= \cdots \\
\quad \mid \textsf{match}\ e\ \textsf{with}\ [] \to e_1 \mid x_1 :: x_2 \to e_2 \\
\quad \mid\ !e \\
\quad \mid\ e := e' \\
\tau ::= \cdots \\
\quad \mid \texttt{unit} \\
\quad \mid \tau\ \texttt{list} \\
\quad \mid \tau\ \texttt{ref}
\end{array}
$$

(a) Syntaxe

$$
\textsc{MatchList}\quad \frac{\Gamma \vdash e : \tau\ \texttt{list} \qquad \Gamma \vdash e_1 : \tau' \qquad \Gamma[x_1 : \tau, x_2 : \tau\ \texttt{list}] \vdash e_2 : \tau'}{\Gamma \vdash \textsf{match}\ e\ \textsf{with}\ [] \to e_1 \mid x_1 :: x_2 \to e_2 : \tau'}
$$

$$
\textsc{Get}\quad \frac{\Gamma \vdash e : \tau\ \texttt{ref}}{\Gamma \vdash\ !e : \tau} \qquad \textsc{Set}\quad \frac{\Gamma \vdash e : \tau\ \texttt{ref} \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e' : \texttt{unit}}
$$

(b) Typage

Figure 13 – Extension de Macle avec accès mémoire

La figure 14 donne un exemple de circuit Macle calculant la longueur d'une liste OCaml.

```
circuit longueur ℓ =
  let rec aux ℓ acc =
    match ℓ with
      [] → acc
    | _ :: ℓ' → aux ℓ' acc
  in aux ℓ 0
```

Figure 14 – Calcul de la longueur d'une liste en Macle

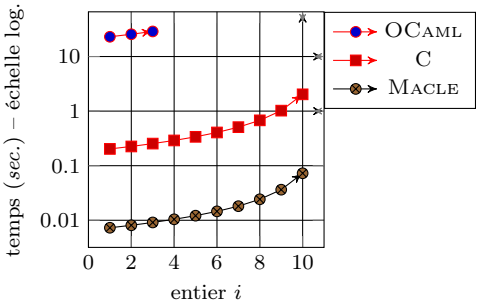
7 Tests de performance

Cette section compare les temps d'exécution de fonctions OCaml, C et Macle incorporées à la bibliothèque d'exécution d'O2B et appelées par des programmes de test écrits en OCaml. Le facteur d'accélération de Macle vis-à-vis d'OCaml et C est mesuré dans un même environnement d'exécution, O2B, compilé par gcc avec optimisation activée (-Os) à travers le backend Nios2. Le code assembleur produit est exécuté par un processeur softcore Nios2 réalisé sur le FPGA Intel Max 10 embarqué sur une carte DE10-Lite de Terasic. Ce modèle comprend 50K cellules logiques et 1638 Kbits de mémoire pour une fréquence d'horloge de 50 MHz. Les programmes OCaml sont compilés par le compilateur ocamlc. Le programme en bytecode engendré est modifié par l'outil ocamlclean. Les circuits Macle sont synthétisés à travers la chaîne de développement Quartus Lite. La taille du tas O2B est de 11756 mots. La taille de la pile O2B est de 512 mots. La taille d'un mot est de 4 octets. Le temps d'exécution d'une fonction externe C (*resp.* OCaml) est mesuré aux extrémités du corps de celle-ci (*resp.* avant et après avoir été appliquée) par la différence de deux appels à la fonction nios_timer_get_us() définie dans O2B. Le temps d'exécution d'un circuit Macle est défini comme le temps d'exécution de la fonction C qui appelle ce circuit.

7.1 Comparaison du temps d'exécution entre OCaml, C et Macle

Nous implémentons en OCaml, C et Macle l'algorithme d'Euclide $\gcd(a, b)$ calculant le plus grand diviseur commun de deux entiers a et b. La fonction OCaml est récursive terminale, la version C est itérative ; la version Macle est donnée figure 12a. Nous comparons le temps

d'exécution de $\gcd(i, n!)$ pour i allant de 1 à n avec n fixé à la valeur 10. Sur la figure 15b, Nous observons que le facteur d'accélération de la version MACLE atteint 3000 avec le bytecode OCAML (sous sa forme OMICROB) et 28 avec C, ce qui confirme l'hypothèse d'efficacité des circuits MACLE.



(a)

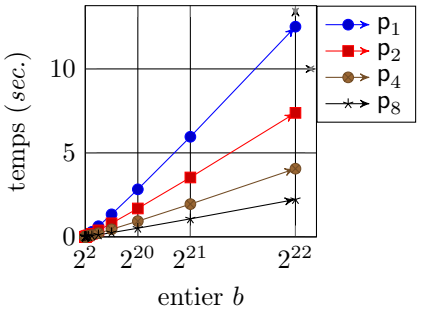
i	ratio C/MACLE	ratio OCAML/MACLE	OCAML (sec.)
1	27.85	3155	23
2	27.81	3164	25.6
3	27.86	3159	28.8
4	27.85		
5	27.84		
6	27,89		
7	28,04		
8	27.96		
9	28.07		
10	27.95		

(b)

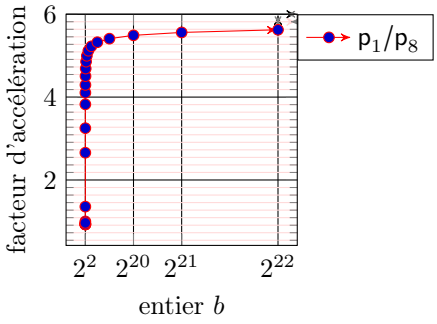
Figure 15 – Temps d'exécution des versions OCAML, C et MACLE du PGCD

7.2 Parallélisme

Pour illustrer le parallélisme des circuits programmés en MACLE, nous considérons le circuit *somme_temps_de_vol4*(a, b) présenté à la figure 12b. Ce circuit comporte une boucle récursive terminale effectuant la somme des expressions (temps_de_vol $(i+k)$) pour k variant de 0 à 3, en traitant les éléments $(i+k)$ en parallèle par paquet de 4. La fonction temps_de_vol est une boucle non bornée qui calcule le *temps de vol* de la suite de Syracuse i. Par exemple, (temps_de_vol 10) vaut 6, (temps_de_vol 11) vaut 14, (temps_de_vol 12) vaut 9. Nous généralisons cette fonction à des paquets de 1, 2, 4 et 8 liaisons locales parallèles. La figure 16a mesure le temps d'exécution de ces quatre versions pour a égal à 1 et b variant de 2^1 à 2^{22}.



(a)



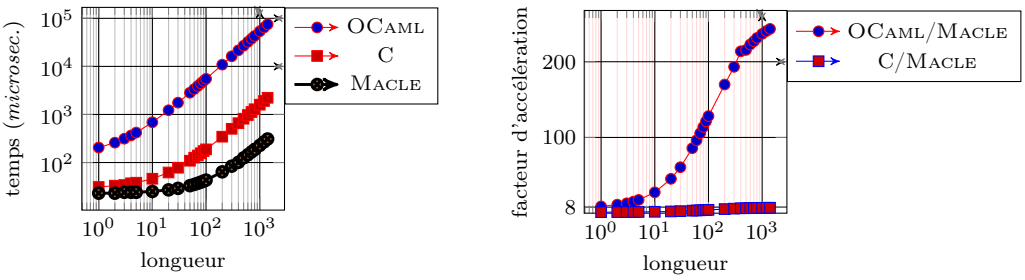
(b)

Figure 16 – Calculs irréguliers exécutés en parallèle par groupes de 1 à 8

La figure 16b met en évidence un facteur d'accélération de 5,6 pour le circuit p_8 vis-à-vis de p_1, soit de 70% par rapport à l'optimum (un facteur d'accélération pour p8 vis-à-vis de p1).

7.3 Calcul de la longueur d'une liste

Nous implémentons une fonction OCaml, une fonction C et un circuit Macle calculant la longueur d'une liste OCaml. La version OCaml est récursive terminale (c'est `List.length`) ; La version C est itérative ; la version Macle est donnée en figure 14. On compare le temps d'exécution de ces deux versions en fonction de la longueur de la liste d'entrée. Nous observons le facteur d'accélération de la version Macle en fonction de la longueur de la liste. La figure 17a donne les valeurs mesurées. La figure 17b donne le facteurs d'accélération de Macle. Celui-ci atteint seulement 7 vis-à-vis de C pour une liste de 1400 éléments, et 270 vis-à-vis d'OCaml. Cela met en évidence un goulet d'étranglement dû au transfert de données. En effet, notre utilisation du bus Avalon ne permet pas de paralléliser les lectures en mémoire.



(a) **(b)**

Figure 17 – Performance d'un calcul sur des listes en fonction de leurs longueurs

7.4 Synthèse des tests de performance

Ces premiers résultats sont encourageants et confortent l'approche suivie. Sur le calcul du PGCD, les gains, de l'ordre de 30, de Macle vis-à-vis de C semblent optimaux.

Sur le calcul parallèle du temps de vol de la suite de Syracuse, et même si les calculs ne sont pas réguliers, nous obtenons un gain de 70% de la taille des paquets traités en parallèle, soit environ 5,6 pour des paquets de taille 8. Ces deux gains se multiplient si le calcul du PGCD est effectué par paquet en parallèle vis-à-vis du C séquentiel.

Le calcul de la longueur d'une liste du tas OCaml met en évidence un goulet d'étranglement du au transfert de données. Notre utilisation du bus Avalon ne permet pas de paralléliser les lectures en mémoire. Le gain de 7 par rapport à C est moindre qu'escompté, mais non négligeable tout de même. Une meilleure localisation mémoire et un contrôle plus fin de la taille des blocs transférés sur le bus depuis la mémoire devrait permettre d'améliorer les performances.

8 Travaux connexes

Cette section présente les travaux en compilation de langages fonctionnels sur FPGA les plus proches des nôtres.

SAFL (*Statically Allocated Parallel Functional Language*) [10] est un langage strict avec une syntaxe *à la ML*. Il est compilé en Verilog. Les choix de conception du langage suivent deux principes : les sous-expressions indépendantes sont exécutées en parallèle et les circuits

engendrés n'allouent pas de mémoire (il n'y a ni pile, ni tas). Chaque fonction est compilée en un circuit indépendant. Un mécanisme d'exclusion mutuelle (arbitre) est inséré dans le code engendré pour séquentialiser les accès concurrents à une même fonction (e.g. dans $f(1)+f(2)$).

Le compilateur SHARD (*a Scheme to Hardware Compiler*) [12] pour Scheme cible VHDL. Les sous-expressions indépendantes sont calculées en parallèle. Le langage supporte les entiers, les tableaux globaux, l'allocation de fermetures et la récursion. Il est possible d'encoder des listes avec des fermetures (et ainsi implanter, par exemple, un tri fusion). La gestion mémoire est élémentaire : les fermetures sont désallouées dès qu'elles sont appliquées, ce qui (d'après les auteurs) est source d'erreurs et difficilement utilisable en pratique.

Le langage FLOH (*Functional Language On Hardware*) [16] est une langage non-strict proche du langage *noyau* du compilateur GHC. FLOH est compilé vers SystemVerilog par une succession de transformations préservant la sémantique. La compilation des appels de fonctions s'inspire du mécanisme de SAFL (arbitres). Des extensions de FLOH supportent la récursion (par introduction de piles d'appels explicites) et le polymorphisme (en monomorphisant) [19].

9 Conclusion et travaux futurs

Cet article a présenté Macle : langage dédié à la programmation FPGA en style applicatif. Les *circuits* Macle sont compilés vers des automates synchrones parallèles en VHDL pour reconfigurer un FPGA. L'implémentation O2B de la machine virtuelle OCaml ciblant un processeur *softcore* réalisé sur le même FPGA permet l'appel de ces circuits Macle (vus comme des fonctions externes C) depuis des programmes OCaml. La reconfiguration du FPGA est entièrement assurée par le compilateur Macle. Le programmeur a ainsi accès, de façon transparente, à la puissance de calcul du FPGA. Macle permet par ailleurs le parcours et la modification physique de valeurs allouées dans le tas OCaml, vu comme une mémoire partagée entre les circuits et le *softcore*.

Les premiers résultats obtenus sont encourageants. Nous observons des facteurs d'accélérations importants (de l'ordre de 30) des circuits Macle vis-à-vis de fonctions C exécutées sur le *softcore*. Par contre, les accès au tas OCaml depuis les circuits Macle – via des requêtes sur le bus assurant la communication entre le *softcore* et le reste du FPGA – sont moins efficaces (7 fois plus rapide que C). En exploitant le parallélisme présent implicitement dans les circuits (e.g. liaisons locales, applications de fonctions et d'opérateurs), Macle peut cependant accélérer significativement certains calculs, réguliers ou irréguliers.

Macle et O2B sont des logiciels libres [13] disponibles aux adresses suivantes :

https://github.com/lsylvestre/macle
https://github.com/jserot/O2B

Travaux futurs L'approche portable d'O2B et de Macle devrait faciliter leur portage sur d'autres familles de FPGA ainsi que la construction d'un mode simulateur enrichi pour la mise au point des programmes. L'introduction de squelettes de parallélisme sur des structures de données allouées, en particuliers les vecteurs, aidera à l'écriture d'algorithmes parallèles tout en optimisant l'accès mémoire par paquets. L'ajout d'un mécanisme de gestion d'erreurs et d'un allocateur de mémoire devrait aussi permettre d'implémenter en Macle une machine virtuelle OCaml simplifiée.

13. Les travaux sur Macle et O2B sont soutenus par l'Initiative de Recherche et Innovation sur le Logiciel Libre (IRILL).

Références

[1] C. Baaij and J. Kuper. Using Rewriting to Synthesize Functional Languages to Digital Circuits. In *International Symposium on Trends in Functional Programming*, pages 17–33. Springer, 2013.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava : hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1) :174–184, 1998.

[3] J. Choi, M. Vijayaraghavan, B. Sherman, and A. Chlipala. Kami : A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP) :1–30, 2017.

[4] J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, pages 73–82, 2006.

[5] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to high-performance hardware on FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534. IEEE, 2012.

[6] P. Gammie. Synchronous Digital Circuits as Functional Programs. *ACM Computing Surveys (CSUR)*, 46(2) :1–27, 2013.

[7] The HardCaml OCaml Library. https://github.com/janestreet/hardcaml.

[8] F. Maraninchi and Y. Rémond. Mode-automata : About Modes and States for Reactive Systems. In *European Symposium On Programming*, pages 185–199. Springer, 1998.

[9] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in hawk. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225)*, pages 90–101. IEEE, 1998.

[10] A. Mycroft and R. Sharp. A Statically Allocated Parallel Functional Language. In *International Colloquium on Automata, Languages, and Programming*, pages 37–48. Springer, 2000.

[11] S. Peyton Jones, W. Partain, and A. Santos. Let-floating : moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN International Conference on Functional programming*, pages 1–12, 1996.

[12] X. Saint-Mleux, M. Feeley, and J.-P. David. SHard : a Scheme to Hardware Compiler. In *Workshop on Scheme and Functional Programming*, 2006.

[13] J. Sérot. *La programmation des circuits FPGA et le langage VHDL. Une introduction pour les programmeurs et par l'exemple.* Ellipse, 2019.

[14] J. Sérot and E. Chailloux. OCaml sur circuit FPGA. *32 ème Journées Francophones des Langages Applicatifs*, pages 72–74, 2021.

[15] J. Sérot and G. Michaelson. Compiling Hume down to gates. In *Draft Proceedings of 11th International Symposium on Trends in Functional Programming*, pages 191–226, 2011.

[16] R. Townsend, M. A. Kim, and S. A. Edwards. From Functional Programs to Pipelined Dataflow Circuits. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 76–86, 2017.

[17] S. Varoumas, B. Vaugon, and E. Chailloux. A Generic Virtual Machine Approach for Programming Microcontrollers : the OMicroB Project. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France, Jan. 2018.

[18] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and computation*, 115(1) :38–94, 1994.

[19] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards. Hardware Synthesis from a Recursive Functional Language. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 83–93. IEEE, 2015.