Mixing interaction and computation on FPGA

Loïc Sylvestre – Sorbonne Université, Lip6, IRILL

30 november, 2023 – Kiel, SYNCHRON 2023

Outline

Design and implementation of Eclat

- a declarative (OCaml-like) language compiled to circuits
- for programming reactive hardware applications on FPGA
  - based on a logical time = the global clock of the circuit
  - able to simply express long-running computations
- with an extension for interacting with the FPGA environment

A (typed) call-by-value λ-calculus with let-polymorphism

expression $e ::= c$
$\mid x$
$\mid \texttt{fun}\ p\ \texttt{->}\ e$
$\mid e\ e$
$\mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
$\mid (e, e)$
$\mid \texttt{let}\ p = e\ \texttt{in}\ e$

pattern $p ::= x \mid (p, p)$

constant $c ::= \text{true} \mid \text{false} \mid N \mid + \mid - \mid < \mid > \mid =$

value $v ::= c \mid \texttt{fun}\ p\ \texttt{->}\ e \mid (v, v)$

abbreviation $(\texttt{let}\ f\ p = e\ \texttt{in}\ e') \equiv (\texttt{let}\ f = \texttt{fun}\ p\ \texttt{->}\ e\ \texttt{in}\ e')$

Reduction semantics

reduction relation $e \longrightarrow e'$

context $E ::= \square\ e \mid v\ \square \mid \texttt{if}\ \square\ \texttt{then}\ e\ \texttt{else}\ e$
$\mid (\square, e) \mid (v, \square) \mid \texttt{let}\ p = \square\ \texttt{in}\ e$

reduction rules:

$E[e] \longrightarrow E[e']$ if $e \longrightarrow e'$ (Context)

$(\texttt{fun}\ p\ \texttt{->}\ e)\ v \longrightarrow e[p \mapsto v]$ (App)

$\texttt{if}\ \text{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_1$ (If-true)

$\texttt{if}\ \text{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2$ (If-false)

$\texttt{let}\ p = v\ \texttt{in}\ e \longrightarrow e[p \mapsto v]$ (Let)

$c\ v \longrightarrow \text{call}(c, v)$ (Call)

Exemple: a combinational circuit

```
let not a = if a then false else true in
let xor (a,b) = if a then not b else b in
let or (a,b) = if a then true else b in
let land (a,b) = if a then b else false in

let half_add(a,b) = (xor(a,b), land(a,b)) in

let full_add(a,b,ci) =
  let (s1,c1) = half_add(a,b) in
  let (s,c2) = half_add(ci,s1) in
  let co = or(c1,c2) in
  (s, co)
in
full_add (* entry point, applied to input values at each clock tick *)
```

Expressing long-running computations

- e ::= $\cdots$ | `pause` e
- new reduction rule: $E[\texttt{pause}\ e] \longrightarrow \texttt{pause}\ E[e]$ (Pause)
- but: $\texttt{pause}\ e \not\longrightarrow$
- $e \longrightarrow^* e'$: a suite of zero or more reductions

- A close expression e is either **instantaneous** ($e \longrightarrow^* v$) or **non-instantaneous** ($e \longrightarrow^* \texttt{pause}\ e'$)
- Behavioral semantics: $e \Longrightarrow e'$ (reduction in one clock cycle)

$$\textsc{Tick-val}\ \frac{e \longrightarrow^* v}{e \Longrightarrow v} \qquad \textsc{Tick-pause}\ \frac{e \longrightarrow^* \texttt{pause}\ e'}{e \Longrightarrow e'}$$

Expressing long-running computations

- e ::= $\cdots$ | $\texttt{pause}\ e$
- new reduction rule: $E[\texttt{pause}\ e] \longrightarrow \texttt{pause}\ E[e]$ (Pause)
- but: $\texttt{pause}\ e \not\longrightarrow$
- $e \longrightarrow^* e'$: a suite of zero or more reductions

- A close expression e is either **instantaneous** ($e \longrightarrow^* v$) or **non-instantaneous** ($e \longrightarrow^* \texttt{pause}\ e'$)
- Behavioral semantics: $e \Longrightarrow e'$ (reduction in one clock cycle)

$$\textsc{Tick-val}\ \frac{e \longrightarrow^* v}{e \Longrightarrow v} \qquad \textsc{Tick-pause}\ \frac{e \longrightarrow^* \texttt{pause}\ e'}{e \Longrightarrow e'}$$

Expressing long-running computations

- e ::= $\cdots \mid$ `pause` e
- new reduction rule: $E[\texttt{pause}\ e] \longrightarrow \texttt{pause}\ E[e]$ (Pause)
- but: `pause` $e \not\longrightarrow$
- $e \longrightarrow^* e'$: a suite of zero or more reductions
- A close expression e is either **instantaneous** ($e \longrightarrow^* v$) or **non-instantaneous** ($e \longrightarrow^*$ `pause` e')
- Behavioral semantics: $e \Longrightarrow e'$ (reduction in one clock cycle)

$$\textsc{Tick-val}\quad \frac{e \longrightarrow^* v}{e \Longrightarrow v} \qquad \textsc{Tick-pause}\quad \frac{e \longrightarrow^* \texttt{pause}\ e'}{e \Longrightarrow e'}$$

Expressing long-running computations

- e ::= $\cdots \mid$ $\texttt{pause}\ e$
- new reduction rule: $E[\texttt{pause}\ e] \longrightarrow \texttt{pause}\ E[e] \quad (\textsc{Pause})$
- but: $\texttt{pause}\ e \not\longrightarrow$
- $e \longrightarrow^* e'$: a suite of zero or more reductions

- A close expression e is either **instantaneous** ($e \longrightarrow^* v$) or **non-instantaneous** ($e \longrightarrow^* \texttt{pause}\ e'$)
- Behavioral semantics: $e \Longrightarrow e'$ (reduction in one clock cycle)

$$\textsc{Tick-val}\ \frac{e \longrightarrow^* v}{e \Longrightarrow v} \qquad \textsc{Tick-pause}\ \frac{e \longrightarrow^* \texttt{pause}\ e'}{e \Longrightarrow e'}$$

Example

if (pause true) then (let x = true in (pause x)) else false

$\longrightarrow$ pause (if true then (let x = true in (pause x)) else false)

$\Longrightarrow$ (if true then (let x = true in (pause x)) else false)

$\longrightarrow$ let x = true in (pause x)

$\longrightarrow$ pause true

$\Longrightarrow$ true

Example

`if` (`pause` true) `then` (`let` x = true `in` (`pause` x)) `else` false

$\longrightarrow$ `pause` (`if` true `then` (`let` x = true `in` (`pause` x)) `else` false)

$\Longrightarrow$ (`if` true `then` (`let` x = true `in` (`pause` x)) `else` false)

$\longrightarrow$ `let` x = true `in` (`pause` x)

$\longrightarrow$ `pause` true

$\Longrightarrow$ true

Example

if (pause true) then (let x = true in (pause x)) else false

$\longrightarrow$ pause (if true then (let x = true in (pause x)) else false)

$\Longrightarrow$ (if true then (let x = true in (pause x)) else false)

$\longrightarrow$ let x = true in (pause x)

$\longrightarrow$ pause true

$\Longrightarrow$ true

Example

$\texttt{if}$ (pause true) then (let x = true in (pause x)) else false

$\longrightarrow$ pause (if true then (let x = true in (pause x)) else false)

$\Longrightarrow$ (if true then (let x = true in (pause x)) else false)

$\longrightarrow$ let x = true in (pause x)

$\longrightarrow$ pause true

$\Longrightarrow$ true

Example

```
if (pause true) then (let x = true in (pause x)) else false

⟶ pause (if true then (let x = true in (pause x)) else false)

⟹ (if true then (let x = true in (pause x)) else false)

⟶ let x = true in (pause x)

⟶ pause true

⟹ true
```

Example

$\texttt{if}$ ($\texttt{pause}$ true) $\texttt{then}$ ($\texttt{let}$ x = true $\texttt{in}$ ($\texttt{pause}$ x)) $\texttt{else}$ false

$\longrightarrow$ $\texttt{pause}$ ($\texttt{if}$ true $\texttt{then}$ ($\texttt{let}$ x = true $\texttt{in}$ ($\texttt{pause}$ x)) $\texttt{else}$ false)

$\Longrightarrow$ ($\texttt{if}$ true $\texttt{then}$ ($\texttt{let}$ x = true $\texttt{in}$ ($\texttt{pause}$ x)) $\texttt{else}$ false)

$\longrightarrow$ $\texttt{let}$ x = true $\texttt{in}$ ($\texttt{pause}$ x)

$\longrightarrow$ $\texttt{pause}$ true

$\Longrightarrow$ true

Parallel composition (returns a pair of values)

$$e ::= \cdots \mid (e \parallel e)$$

$$E ::= \cdots \mid (\square \parallel e) \mid (v \parallel \square) \mid (\texttt{pause}\ e \parallel \square)$$

$$(v \| v') \longrightarrow (v, v') \quad (\textsc{Par-val})$$

$$E[\texttt{pause}\ e] \longrightarrow \texttt{pause}\ E[e] \quad \text{if } E \not\equiv (\square \parallel \texttt{pause}\ e') \quad (\textsc{Pause})$$

$$(\texttt{pause}\ e \parallel \texttt{pause}\ e') \longrightarrow \texttt{pause}\ (e \parallel e') \quad (\textsc{Par-pause})$$

Recursive functions

expression $e ::= \cdots \mid \texttt{fix}\ f\ (\texttt{fun}\ p\ \texttt{->}\ e)$

value $v ::= \cdots \mid \texttt{fix}\ f\ (\texttt{fun}\ p\ \texttt{->}\ e)$

- $$(\underbrace{\texttt{fix}\ f\ (\texttt{fun}\ p\ \texttt{->}\ e)}_{\phi})\ v \longrightarrow \texttt{pause}\ (e[f \mapsto \phi]\ v) \qquad (\textsc{Fix})$$

Derivated construct: $(\texttt{let rec}\ f\ p = e\ \texttt{in}\ e')$
$$\equiv \texttt{let}\ f = \texttt{fix}\ f\ (\texttt{fun}\ p\ \texttt{->}\ e)\ \texttt{in}\ e'$$

Limitation: the compiler supports only tail-recursion.

Example: a long-running computation

```
let rec gcd(a,b) = (* does not always terminate, e.g., gcd(1,-1) *)
  if a < b then gcd(a,b-a)
  else if a > b then gcd(a-b,b)
  else a in
let x = gcd(2,2) in
let (x1,x2) = (gcd(18,12) ‖ gcd(5,10)) in
let s = x1 + x2 in
(x,s)  (* see on slide 15 how to run such a computation
          in reactive programs *)
```

clock ticks	t_0	t_1	t_2	t_3	t_4
x	ε	2	2	2	2
x1		ε	ε	ε	6
x2		ε	ε	5	5
s					11

Stateful computation (for interaction)

$e ::= \cdots \mid$ `reg`$_\ell$ e `last` e (*e.g.*, `reg`$_\ell$ (`fun c -> c + 1`) `last 0`)

This construct instantaneously returns a value.
It uses a local memory for saving values between instantaneous execution of the program.

Labels (ℓ) are omitted in source programs.

Reduction relation extended with states μ: $e/\mu \longrightarrow e'/\mu'$
Non-standard substitution: $e[x \mapsto v]$ means "replace each occurrence of x in e by an instance of value v".
Instantiation renames labels in the same way at each execution.

Stateful computation (for interaction)

$$e ::= \cdots \mid \texttt{reg}_\ell\ e\ \texttt{last}\ e \qquad (\textit{e.g.}, \texttt{reg}_\ell\ (\texttt{fun}\ \texttt{c}\ \texttt{->}\ \texttt{c}\ \texttt{+}\ \texttt{1})\ \texttt{last}\ \texttt{0})$$

$$E ::= \cdots \mid \texttt{reg}_\ell\ \square\ \texttt{last}\ e \mid \texttt{reg}_\ell\ v\ \texttt{last}\ \square$$

$$\overbrace{\texttt{reg}_\ell\ w\ \texttt{last}\ v}^{e}/\mu \longrightarrow e/\mu[\ell \mapsto w\ v] \quad \text{if } \ell \notin \mathrm{dom}(\mu) \quad (\textsc{Reg-init})$$

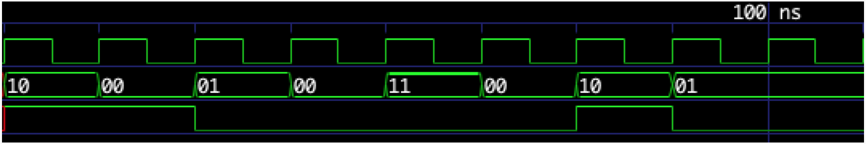
$$\texttt{reg}_\ell\ w\ \texttt{last}\ v/\mu \longrightarrow v'/\mu[\ell \mapsto w\ v'] \quad \text{if } \mu(\ell) = v' \quad (\textsc{Reg-next})$$

$$\textsc{Reg-context}\quad \frac{\mu(\ell) = e \qquad e/\mu \longrightarrow e'/\mu'}{\texttt{reg}_\ell\ w\ \texttt{last}\ v/\mu \longrightarrow \texttt{reg}_\ell\ w\ \texttt{last}\ v/\mu'[\ell \mapsto e']}$$

Example: await

```
let await (i, reset) =
  let step s = (s or i) & not reset in
  reg step last false
in
await   (* entry point, applied to input values at each clock tick *)
```

Simulation of the VHDL generated code:



Example: ABCRO

```
let fby (a,b) =
  let step (pre,cur) = (cur,b) in
  let (cur,_) = reg step last (a,a) in cur in

let edge i = not (fby( false , i)) & i in

let abro ((a,b),r) = edge (await (a,r) & await(b,r)) in

let abcro (((a,b),c),r) = abro ((abro((a,b),r),c),r)
in abcro   (* entry point, applied to input values at each clock tick *)
```

Mix interaction and computation

Accept registers with non-instantaneous update functions:

$$\texttt{reg}_\ell\ w\ \texttt{last}\ v/\mu \longrightarrow v/\mu[\ell \mapsto e] \quad \text{if } \mu(\ell) = \texttt{pause}\ e \quad (\textsc{Default})$$

derivated construct:

$$\texttt{exec}\ e_1\ \texttt{default}\ e_2 \equiv \texttt{reg}\ (\texttt{fun \_ ->}\ (e_1,\text{true}))\ \texttt{last}\ (e_2,\text{false})$$

Example

```
let main (((a,b),r),((x,y),suspend)) =
  if suspend then 0
  else (let (v,rdy) = exec gcd(x,y) default 0 in
        if abcro(((a,b),rdy),r) then v
        else 42)
in
main  (* entry point, applied to input values at each clock tick *)
```

- when suspend, returns 0 without activating the *else* branch
- otherwise,
  - run step-by-step the computation gcd(x,y)
  - use the *rdy* output of construct `exec` as input c for abcro

Type-based reactivity analysis (1/3)

$$
\begin{array}{ll}
\textbf{type} & \tau ::= \texttt{bool} \mid \mathsf{int} \mid \tau \times \tau \mid \tau \xrightarrow{\delta} \tau \mid \alpha \\
\textbf{type scheme} & \sigma ::= \forall \alpha . \tau \\
\\
\textbf{response time} & \delta ::= N \mid \max(\delta, \delta) \mid \delta + \delta
\end{array}
$$

Judgement $\boxed{\Gamma \vdash e : \tau | \delta}$, means "in the typing environment Γ, expression e has type τ and worst-case response time δ".

A close program e is reactive if $\varnothing \vdash e : \tau | \mathbf{0}$.

Type-based reactivity analysis (2/3)

Ty-Const

$$\frac{\Delta(c) = \sigma}{\Gamma \vdash c : \text{instance}(\sigma, \Gamma) | \mathbf{0}}$$

Ty-Var

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \text{instance}(\sigma, \Gamma) | \mathbf{0}}$$

Ty-Pause

$$\frac{\Gamma \vdash e : \tau | \delta}{\Gamma \vdash \texttt{pause}\ e : \tau | \mathbf{1} + \delta}$$

Ty-Let

$$\frac{\Gamma \vdash e : \tau | \delta \qquad \Gamma[p \mapsto \text{gen}(\tau, \Gamma)] \vdash e' : \tau' | \delta'}{\Gamma \vdash \texttt{let}\ p = e\ \texttt{in}\ e' : \tau' | \delta + \delta'}$$

Ty-If

$$\frac{\Gamma \vdash e : \texttt{bool} | \delta \qquad \Gamma \vdash e_i : \tau | \delta_i \qquad i \in \{1, 2\}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau | \delta + \max(\delta_1, \delta_2)}$$

Ty-Sub

$$\frac{\Gamma \vdash e : \tau | \delta}{\Gamma \vdash e : \tau | \delta + \boldsymbol{N}}$$

Type-based reactivity analysis (3/3)

Ty-Pair

$$\frac{\Gamma \vdash e_i : \tau_i | \delta_i \qquad i \in \{1,2\}}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 | \delta_1 + \delta_2}$$

Ty-par

$$\frac{\Gamma \vdash e_i : \tau_i | \delta_i \qquad i \in \{1,2\}}{\Gamma \vdash (e_2 \| e_1) : \tau_1 \times \tau_2 | \max(\delta_1, \delta_2)}$$

Ty-Fun

$$\frac{\Gamma[p : \tau] \vdash e' : \tau' | \delta}{\Gamma \vdash (\texttt{fun}\ p \to e) : \tau \xrightarrow{\delta} \tau' | \mathbf{0}}$$

Ty-App

$$\frac{\Gamma \vdash e_1 : \tau \xrightarrow{\delta'} \tau' | \mathbf{0} \qquad \Gamma \vdash e_2 : \tau | \delta}{\Gamma \vdash e_1\ e_2 : \tau' | \delta + \delta'}$$

Ty-Fix

$$\frac{\Gamma[f : \tau \xrightarrow{\delta+1} \tau'][p : \tau] \vdash e : \tau' | \delta}{\Gamma \vdash \texttt{fix}\ f\ (\texttt{fun}\ p \to e) : \tau \xrightarrow{\delta+1} \tau' | \mathbf{0}}$$

Ty-Reg

$$\frac{\Gamma \vdash e : \tau \xrightarrow{\delta} \tau | \mathbf{0} \qquad \Gamma \vdash e_0 : \tau | \mathbf{0}}{\Gamma \vdash \texttt{reg}_\ell\ e\ \texttt{last}\ e_0 : \tau | \mathbf{0}}$$

Type-based reactivity analysis (3/3)

$$\textsc{Ty-Pair}\quad \frac{\Gamma \vdash e_i : \tau_i | \delta_i \qquad i \in \{1,2\}}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 | \delta_1 + \delta_2}$$

$$\textsc{Ty-par}\quad \frac{\Gamma \vdash e_i : \tau_i | \delta_i \qquad i \in \{1,2\}}{\Gamma \vdash (e_2 \| e_1) : \tau_1 \times \tau_2 | \max(\delta_1, \delta_2)}$$

$$\textsc{Ty-Fun}\quad \frac{\Gamma[p : \tau] \vdash e' : \tau' | \delta}{\Gamma \vdash (\texttt{fun}\ p \to e) : \tau \xrightarrow{\delta} \tau' | \mathbf{0}}$$

$$\textsc{Ty-App}\quad \frac{\Gamma \vdash e_1 : \tau \xrightarrow{\delta'} \tau' | \mathbf{0} \qquad \Gamma \vdash e_2 : \tau | \delta}{\Gamma \vdash e_1\ e_2 : \tau' | \delta + \delta'}$$

$$\textsc{Ty-Fix}\quad \frac{\Gamma[f : \tau \xrightarrow{\delta+1} \tau'][p : \tau] \vdash e : \tau' | \delta}{\Gamma \vdash \texttt{fix}\ f\ (\texttt{fun}\ p \to e) : \tau \xrightarrow{\delta+1} \tau' | \mathbf{0}}$$

$$\textsc{Ty-Reg}\quad \frac{\Gamma \vdash e : \tau \xrightarrow{\delta} \tau | \mathbf{0} \qquad \Gamma \vdash e_0 : \tau | \mathbf{0}}{\Gamma \vdash \texttt{reg}_\ell\ e\ \texttt{last}\ e_0 : \tau | \mathbf{0}}$$

Conclusion

A core language (Eclat) for programming reactive hardware applications involving long-running computations *[Sylvestre, Chailloux, Sérot – WIP: mixing computation and interaction on FPGA, EMSOFT 23]*

- global clock = logical time (no need to compute a WCET)
- features sized integers, simulation primitives
  and global arrays implemented using RAM memory blocks

with an attempt to trade time and space:

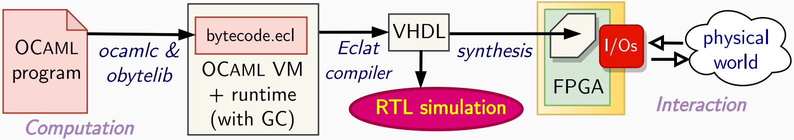
- by making function environments explicit (by λ-lifting),
- then, inlining non-recursive functions
  (= augmenting the size, diminishing the throughput)
- and sharing tail-recursive functions calls
  (= pausing for one tick, diminishing the size)

Conclusion

A core language (ECLAT) for programming reactive hardware applications involving long-running computations *[Sylvestre, Chailloux, Sérot – WIP: mixing computation and interaction on FPGA, EMSOFT 23]*

- global clock = logical time (no need to compute a WCET)
- features sized integers, simulation primitives
  and global arrays implemented using RAM memory blocks

A realistic application: the OCAML VM and runtime in ECLAT



[Sylvestre, Sérot, Chailloux – Hardware implementation of OCAML using a synchronous functional language, PADL 24]