Mixing software abstractions for high-level FPGA programming

Loïc Sylvestre[1] – Sorbonne Université, Lip6, IRILL

July 10, 2023 – FSiC 2023

[1] Doctoral candidate under the supervision of Pr. Emmanuel Chailloux (Sorbonne Université) & Pr. Jocelyn Sérot (Université Clermont Auvergne)

Experiments in FPGA programming

Field Programmable Gate Array (FPGA)

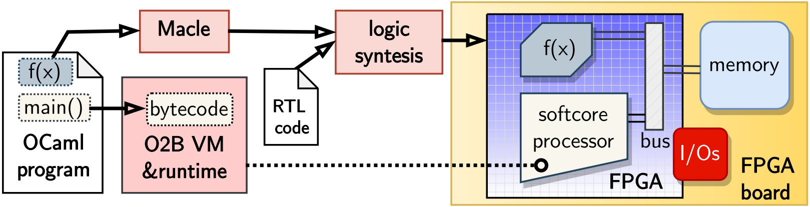
- reconfigurable architecture
- to emulate custom hardware designs

Design and implementation of programming languages

- on FPGA targets
- by compilation to hardware descriptions
  - at the Register Transfer Level (RTL)
- implementation of high level programming features
  - like dynamic data structures
    with automatic memory management
- dedicated language constructs
  - to exploit fine-grained parallelism
  - to interact with the FPGA environment

OCaml on FPGA

- **OCaml** (https://ocaml.org): multi-paradigm programming language, free & open-source, developed by INRIA, *2023 ACM SIGPLAN Programming Language Software Award*
- **O2B**: implementation of the OCaml Virtual Machine on a soft processor (based on OMicroB[2] which targets microcontrollers)
- **Macle**: compiler for a subset of OCaml to RTL



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

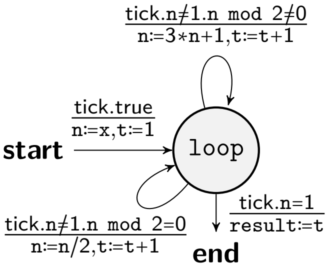
[2] https://github.com/stevenvar/OMicroB

An OCaml program on FPGA

```
(* each "circuit" is an OCaml function to be compiled to RTL with Macle *)
circuit max(a,b) =
  if a > b then a else b ;;

circuit collatz(n) =
  let rec loop(n,t) = (* inner tail-recursive function "loop" *)
    if n == 1 then t else
    if n mod 2 == 0 then
      loop(n/2,t+1)
    else loop(3*n+1,t+1)
  in loop(n,1) ;;

(* "host" code compiled to bytecode and executed by O2B *)
let main() =
  let x = ref 0 in
  for i = 1 to 100 do (* sequential execution *)
    x := max(!x, collatz(i))
  done ;;

main() ;;
```

Compiling tail-recursion

```
circuit collatz(n) =
  let rec loop(n,t) =
    if n == 1 then t else
    if n mod 2 == 0 then
      loop(n/2,t+1)
    else loop(3*n+1,t+1)
  in loop(n,1)
```



- no need for a call stack
- translation to Finite State Machine (FSM) at the RT level
- each tail-call is a "pause" until the next clock tick
- parameter passing corresponds to variable assignment
- current work: sharing of non simultaneous functions calls, like:

```
let x = collatz(n) in collatz(x)
```

A faster OCaml program on FPGA

```
(* OCaml functions compiled to RTL with Macle *)
circuit max(a,b) = ... ;;

circuit collatz(n) = ... ;;

circuit max_collatz(n,m) =
  let a = collatz(n)   (* runs collatz(n) and collatz(m) in parallel *)
  and b = collatz(m) in
  (* synchronization *)
  max(a,b)

(* "host" code executed by O2B *)
let main() =
  let x = ref 0 in
  for i = 1 to 50 do (* sequential execution *)
    x := max (!x, max_collatz (i*2, i*2+1))
  done ;;

main() ;;
```

Memory accesses from the accelerated code

- the generated RTL code can perform bus requests to access the external memory, in which OCaml values are allocated

```
(* OCaml functions compiled to RTL with Macle *)
circuit collatz(n) = ... ;;

circuit map_collatz(a) = (* "circuit" accessing shared memory *)
  for i = 0 to array_length a - 1 do (* sequential execution *)
    a.(i) <- collatz(a.(i)) (* uses only one instance of "collatz" *)
  done ;;

(* "host" code executed by O2B *)
let main() =
  let a = Array.init 1024 (fun i -> i+1) in  (* dynamic allocation *)
  map_collatz(a) ;;

main() ;;
```

- currently, no dynamic allocation from the accelerated code

Parallel skeletons

- exploit fine-grained parallelism
- concisely express (simple) parallel algorithms
- optimize memory transfer

```
(* OCaml functions compiled to RTL with Macle *)
circuit collatz(n) = ... ;;

circuit map_collatz(a) = (* "circuit" accessing shared memory *)
  (* uses 32 instances of "collatz" in parallel *)
  (* optimizes bus transfers using a 32-place buffer *)
  array_map<32> collatz a

(* "host" code executed by O2B *)
let main() =
  let a = Array.init 1024 (fun i -> i+1) in  (* dynamic allocation *)
  map_collatz(a) ;;

main() ;;
```

Current approach: reversing the roles

- compiling a cycle-accurate language to RTL
  - following a synchronous reactive approach (*à la* Lustre)
  - execution as sequence of logic steps (or clock ticks)
  - to program interaction with I/Os as instantaneous functions (i.e., functions responding before the next tick)
- all language constructs react instantaneously, except :
  - tail-recursive function call (one clock tick)
  - asynchronous primitive call (several ticks)
- allows expressing both instantaneous and non-instantaneous functions, *i.e.*, interaction and computation
- providing (asynchronous) memory primitives
- could use a softcore processor
  with automatic memory management

Instantaneous vs non-instantaneous functions

- Instantaneous functions (of type $\tau \Rightarrow \tau'$)

```
let half_add(a,b) =
  (a xor b, a & b)
val half_add : bool * bool ⇒ bool

let full_add(a,b,c) =
  let (s1, c1) = half_add(a,b) in
  let (s, c2) = half_add(c, s1) b in
  (s, c1 or c2)
val full_add : bool * bool * bool ⇒ bool
```

- Non-instantaneous functions (of type $\tau \rightarrow \tau'$)

```
let collatz(n) =
  let rec loop(n,t) = (* inner tail-recursive function “loop” *)
    if n == 1 then t else
    if n mod 2 == 0 then loop(n/2,t+1)
    else loop(3*n+1,t+1) (* each call to loop pauses for on tick *)
  in loop(n,1) ;;
val collatz : int → int
```

Mixing interaction and computation

Stateful instantaneous functions (*à la Lustre*)

```
(* sustains value true as soon as input a is true until reset *)
let aro(a,reset) =
  let step(s) = (s or a) & not reset in
  reg step last false
val edge : bool * bool ⇒ bool
```

Asynchronous calls from instantaneous functions

```
(* sustains value true as soon as input a is true
   until collatz(n) returns a value v higher than tmax *)
let main(a,n,tmax) =
  let v,rdy = exec collatz(n) default 0 in
  let reset = rdy & (v > tmax) in
  aro(a,reset)
val main : bool * int * int ⇒ bool
```

Conclusion

- using FPGAs to implement programming languages
- "programming language" approach to better program FPGAs
  - formal synchronous semantics $\rightsquigarrow$ cycle accuracy
  - general-purpose programming: asynchronous calls + shared memory + runtime system
- mixing interaction and computation
- to program reactive embedded applications on FPGA
- current experiment on small FPGAs:
  - OrangeCrab[3] with the Yosys open synthesis suite[4]
  - Intel Max 10 on the DE10-Lite board
- simulation with GHDL[5] & GTKWave[6]

---

[3] https://orangecrab-fpga.github.io/orangecrab-hardware
[4] https://github.com/YosysHQ/yosys
[5] https://github.com/ghdl
[6] https://github.com/gtkwave