

Mixing software abstractions for high-level FPGA programming

Loïc Sylvestre¹ – Sorbonne Université, Lip6, IRILL

July 10, 2023 – FSiC 2023

¹Doctoral 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)

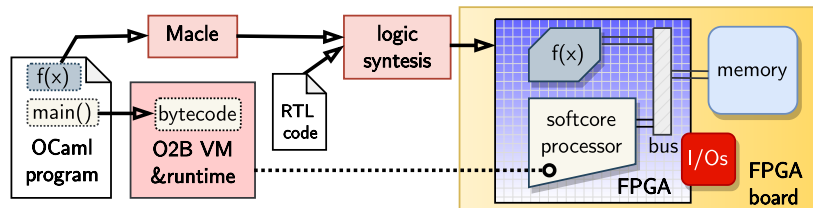
- ▶ reconfigurable architecture
- ▶ to emulate custom hardware designs

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 OMicroB² which targets microcontrollers)
- ▶ **Macle**: compiler for a subset of OCaml to RTL



<https://github.com/jserot/O2B>

<https://github.com/lsylvestre/macle>

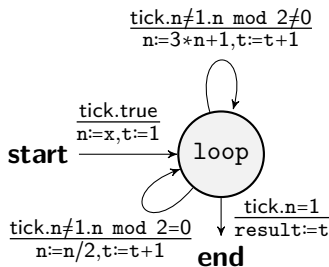
²<https://github.com/stevenvar/OMicroB>

An OCaml program on FPGA

```
1  (* each "circuit" is an OCaml function to be compiled to RTL with Macle *)
2  circuit max(a,b) =
3      if a > b then a else b ;;
4
5  circuit collatz(n) =
6      let rec loop(n,t) = (* inner tail-recursive function "loop" *)
7          if n == 1 then t else
8          if n mod 2 == 0 then
9              loop(n/2,t+1)
10         else loop(3*n+1,t+1)
11         in loop(n,1) ;;
12
13 (* "host" code compiled to bytecode and executed by O2B *)
14 let main() =
15     let x = ref 0 in
16     for i = 1 to 100 do (* sequential execution *)
17         x := max(!x, collatz(i))
18     done ;;
19
20 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

```
1  (* OCaml functions compiled to RTL with Macle *)
2  circuit max(a,b) = ... ;;
3
4  circuit collatz(n) = ... ;;
5
6  circuit max_collatz(n,m) =
7    let a = collatz(n)  (* runs collatz(n) and collatz(m) in parallel *)
8    and b = collatz(m) in
9    (* synchronization *)
10   max(a,b)
11
12  (* "host" code executed by O2B *)
13  let main() =
14    let x = ref 0 in
15    for i = 1 to 50 do (* sequential execution *)
16      x := max (!x, max_collatz (i*2, i*2+1))
17    done ;;
18
19  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

```
1  (* OCaml functions compiled to RTL with Mace *)
2  circuit collatz(n) = ... ;;
3
4  circuit map_collatz(a) = (* "circuit" accessing shared memory *)
5    for i = 0 to array_length a - 1 do (* sequential execution *)
6      a.(i) <- collatz(a.(i)) (* uses only one instance of "collatz" *)
7    done ;;
8
9  (* "host" code executed by O2B *)
10 let main() =
11   let a = Array.init 1024 (fun i -> i+1) in (* dynamic allocation *)
12   map_collatz(a) ;;
13
14 main() ;;
```

- ▶ currently, no dynamic allocation from the accelerated code

Parallel skeletons

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

```
1  (* OCaml functions compiled to RTL with Macle *)
2  circuit collatz(n) = ... ;;
3
4  circuit map_collatz(a) = (* "circuit" accessing shared memory *)
5    (* uses 32 instances of "collatz" in parallel *)
6    (* optimizes bus transfers using a 32-place buffer *)
7    array_map<32> collatz a
8
9  (* "host" code executed by O2B *)
10 let main() =
11   let a = Array.init 1024 (fun i -> i+1) in (* dynamic allocation *)
12   map_collatz(a) ;;
13
14 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 (pauses for one clock tick)
 - asynchronous primitive calls (responds after 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 $\tau \Rightarrow \tau'$)

```
1 | let half_add(a,b) =  
2 |   (a xor b, a & b)  
3 | val half_add : bool * bool  $\Rightarrow$  bool  
4 |  
5 | let full_add(a,b,c) =  
6 |   let (s1, c1) = half_add(a,b) in  
7 |   let (s, c2) = half_add(c, s1) b in  
8 |   (s, c1 or c2)  
9 | val full_add : bool * bool * bool  $\Rightarrow$  bool
```

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

```
1 | let collatz(n) =  
2 |   let rec loop(n,t) = (* inner tail-recursive function "loop" *)  
3 |     if n == 1 then t else  
4 |     if n mod 2 == 0 then loop(n/2,t+1)  
5 |     else loop(3*n+1,t+1) (* each call to loop pauses for one tick *)  
6 |   in loop(n,1) ;;  
7 | val collatz : int  $\rightarrow$  int
```

Mixing interaction and computation

Stateful instantaneous functions (*à la Lustre*)

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

Asynchronous calls from instantaneous functions

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

Conclusion

- ▶ using FPGAs to implement programming languages
- ▶ “programming language” approach to better program FPGAs
 - formal synchronous semantics \leadsto 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:
 - OrangeCrab³ with the Yosys open synthesis suite⁴
 - Intel Max 10 on the DE10-Lite board
- ▶ simulation with GHDL⁵ & GTKWave⁶

³<https://orangecrab-fpga.github.io/orangecrab-hardware>

⁴<https://github.com/YosysHQ/yosys>

⁵<https://github.com/ghdl>

⁶<https://github.com/gtkwave>