A Reactive System-specific Compilation Chain from Synchronous Dataflow Models to FPGA Netlist

Inès Winandy, Arnaud Dion, Pierre-Loïc Garoche, Florent Manni

A Reactive System-specific Compilation Chain from Synchronous Dataflow Models to FPGA Netlist

Inès Winandy*, Arnaud Dion†, Pierre-Loïc Garoche*
Fédération ENAC ISAE-SUPAERO ONERA
Université de Toulouse, Toulouse, France
* first.last@enac.fr, †first.last@isae-supaero.fr

Florent Manni
CNES - Centre national d'études spatiales
Toulouse, France
florent.manni@cnes.fr

***Abstract*—Modern Field Programmable Gate Arrays (FPGAs) offer a solution to several issues related to real-time on-board systems, such as guaranteed execution time. They are currently considered as target platforms for space applications. However, the complexity of producing circuits on these components poses a challenge to their widespread adoption. To address this issue, high-level synthesis tools provide another layer of abstraction above the logic circuit design process, for example compiling C code into Hardware Description Languages such as VHDL or Verilog. However, high-level synthesis results are poorly predictable and do not guarantee the efficient use of recent FPGA capabilities provided by new primitives like digital signal processor or random-access memory. In this paper we propose a compilation chain dedicated to reactive systems, ie. controllers, providing a more predictable synthesis process for critical embedded control applications. The implemented solution demonstrates timing performance equivalent to the traditional synthesis process with a more predictable result.**

***Index Terms*—High Level Synthesis (HLS), Embedded systems, Guidance, Navigation and Control (GNC), Dataflow Programming, Computer-aided Design (CAD)**

I. Introduction

One can compare the different hardware families with respect to flexibility, predictability and computation speed. Flexibility denotes the ability to change the function performed by the hardware. Predictability is associated to execution time and the need in numerous embedded system to guarantee strict real-time deadlines. Both predictability and computation speed are mandatory requirements for embedded aerospace applications such as Guidance Navigation and Control systems. We can associate these properties to the main families of hardware. *Application Specific Integrated Circuits* (ASIC) has predictability and computation speed but their functionality cannot be changed. *Non-recurrent engineering* (NRE) costs are typically very high. *Central Processing Units* (CPU) are very flexible but are less predictable and typically slower at application level because of the layers of software piled to support application execution (kernel, OSX, etc). Last, *Field Programmable Gate Array* (FPGA) is a third family of hardware that combines all these property. Recently FPGA have become very large components and are now considered as an interesting alternative to regular CPUs. In the context of space applications, radiation tolerance has been an obstacle to FPGA adoption for a long time, but nowadays space grade FPGA exist [13], providing a great opportunity for the space sector.

Embedded guidance navigation and control (GNC) systems are safety critical systems that would benefit from FPGA usage. For example, researchers showed that model predictive control (MPC) computation can be executed at Megahertz rate on an FPGA [9]. However, the design of such a high-performance circuit is the result of a hand made optimized design, by both control engineers and FPGA experts. FPGA programming and configuration are a tedious process which requires knowledge in digital circuit design. Some tools aim to facilitate FPGA circuit generation for non-specialized developers. However the resulting performance of automatically generated circuits is often below hand made design. Another issue related with these toolchains is the lack of traceability in the produced circuits. Indeed, FPGA design involves complex algorithms translating input description into circuits. To obtain an efficient circuit, the process is typically very costly, both in time and resources, and it doesn't provide traceability information as required by certification authorities in the aeronautics industry. These issues are mainly due to the uncertainty in primitive inference process. In addition, recent FPGAs contain high-performance primitives that cannot be inferred without using specific annotation.

In this paper, we aim at providing another FPGA synthesis toolchain, focused on control systems for space applications. Similarly to the generation of code for aircraft, following the DO178-C regulation [10], our toolchain emphasizes on the traceability between the input description and the produced circuit.

We made the following contributions:

- a toolchain translating input models in Lustre [5] into Reticle [12], a hardware description language instantiated on a specific FPGA architecture (cf. Section III);
- the approach guarantees the use of specific primitives and provides more fine grain control on the produced circuit (cf. Sections III-B and III-C), ensuring high performances, while preserving traceability information for certification purpose (cf. Section III-D);
- we extended the tool Reticle to target Xilinx Artix 7 FPGA (cf. Section III-E)
- the compilation toolchain performs a quicker synthesis than proprietary tools (cf. Section IV-B).

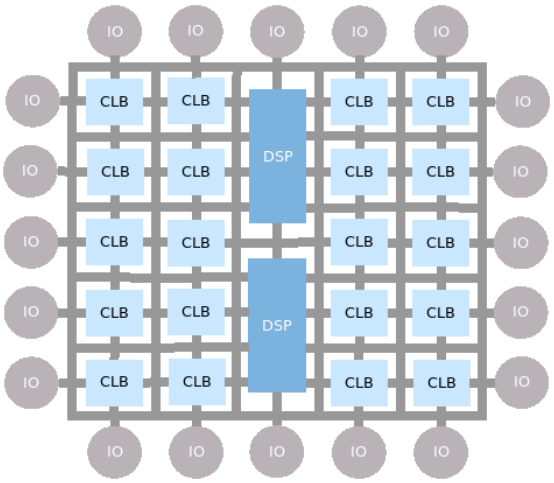


Fig. 1: FPGA Architecture

The paper is structured as follows. In the next section we recall basic design process of FPGA and introduce the reader to both the Reticle hardware description language and Lustre, a synchronous dataflow language. Section III presents our compilation scheme from Lustre to Reticle. Experiments are presented in Section IV. Section V presents the positioning of our contribution with respect to related works, while Section VI concludes.

II. Background

The design of an FPGA circuit is a specific process. We first introduce the reader to the architecture of the FPGA before describing the classical design process, typically from Hardware Description Languages (HDL) such as VHDL or Verilog. We then present the recent new synthesis approach of Reticle and give a short introduction to the Lustre language.

A. FPGA architecture

A *Field Programmable Gate Array* (FPGA) is a type of reconfigurable hardware component. The architecture of an FPGA consists of a collection of primitives placed onto an interconnection matrix. These primitives are grouped in rows and columns, as shown in Figure 1. In addition to coordinate, Basic Element Location (BEL) annotations are used to identify individual primitive. Configuration of an FPGA consists in activating connections between targeted primitives to create a circuit achieving an expected behaviour. The basic components of an FPGA are its *Configurable Logic Blocks* (CLBs), These blocks contain configurable *Lookup Tables* (LUTs), multiplexers (MUX), carry primitives (CARRY4/CARRY8), and flip-flops (FDRE). Some FPGAs also includes specific primitives such as *Digital Signal Processors* (DSPs) and *random-access memory* (RAM). While these primitives can theoretically be implemented using regular CLBs, their implementation is time-consuming, resource-intensive, less efficient, and less predictable than using the predefined one. This is why it is recommended to use dedicated primitives for specific applications, such as arithmetic computation and data storage.

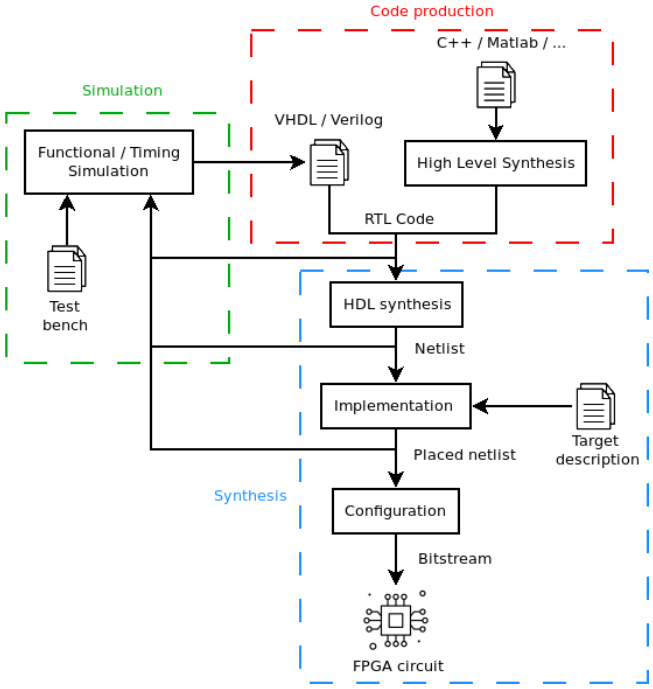


Fig. 2: FPGA circuit design workflow

B. Classical development flow

The design flow for FPGA circuits is an iterative process that involves several steps, as shown in Figure 2. The main difference between FPGA programming and software programming lies in the final usage of the source code. In software programming, the source code is compiled to produce executable code that can be read by the host machine, whereas in FPGA programming, the source code is synthesised to create the machine on a targeted device.

The initial stage of FPGA circuit design involves producing a code using a *Hardware Description Language* (HDL). This process typically begins by creating a behavioural description of the circuit to define and simulate its expected behaviour. After validation, the description needs to be refined to produce a *register transfer level* (RTL) representation that details the data exchanges between registers in the future design. The synthesis process begins at this point, where the RTL code is transformed into a list of components and their interconnections, known as a *netlist*. This netlist is then placed on the targeted FPGA design during implementation, and a bit file can be generated to configure it. Simulations are performed at each stage for validation or code refinement in case of test failure.

The production of RTL code is a crucial step in generating an efficient design and is particularly challenging for non-specialist developers: it requires adapting the system description while considering hardware design constraints. A synthesizable code must contain sufficient information to infer expected components; otherwise, the synthesis result may be an inefficient or non-functional circuit. Listing 1 provides an example of well-formed RTL code. The module's header

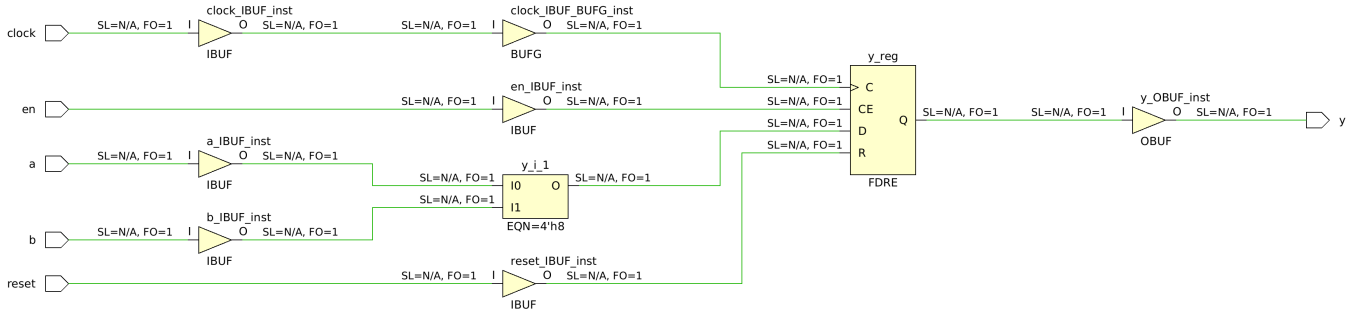


Fig. 3: Example of a Netlist

```
1
2  module main ( input wire clock,
3                input wire reset,
4                input wire [7:0] a,
5                input wire [7:0] b,
6                input wire en,
7                output reg [7:0] y);
8      wire [7:0] t0;
9      assign t0 = a & b;
10     always @(posedge clock) begin
11         if(reset) begin
12             y <= 0;
13         end else if(en) begin
14             y <= t0;
15         end
16     end
17 endmodule
```

Listing 1: RTL code example (in Verilog)

describes its inputs and outputs, which include a `clock` (to ensure timings and validity of other signals in a sampled time), the inputs (`a` and `b`), the output (`y`), a `reset` signal (for register reset), and a clock enabler (`en`). The program declares and computes the intermediate signal `t0` in its body. At each clock cycle, `t0` is stored in the `y` register if the clock is enabled. This example illustrates how a simple AND operation can become complex to describe accurately in HDL. To address this challenge, high-level synthesis (HLS) can be performed to produce RTL representation from higher-level language code, typically C code. Regardless of the code production method, the performance of the resulting circuit may vary depending on the synthesis tool chain.

The aim of the synthesis process is to create a circuit that corresponds to the RTL representation by generating a netlist and placing its components on the targeted device. This process involves several steps, which often include optimization strategies to increase the performance of the circuit. These optimizations involve both the choice of primitives to foster parallelism and their placement to reduce propagation times. Because of the wide variety of FPGA boards and their contained primitives, HDL synthesizers require specific writing of an operation to enable the inference of a specific primitive such as DSPs. As a result, these synthesizers are

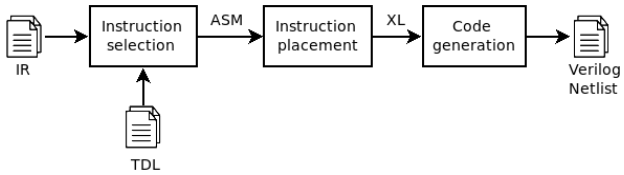


Fig. 4: Reticle compilation pipeline

typically not able to infer these DSPs without proper writing by the developer. Figure 3 displays the netlist generated after synthesizing the previous example, which exclusively contains look up tables and flip-flops. When performing a single bit operation, this implementation may be efficient. However, as the number of bits and operations increases, this efficiency may not be maintained. Therefore, circuits that process a large number of operations between larger data types may benefit from using DSP.

C. The Reticle approach

Reticle [12] provides a solution for generating a placed netlist from an intermediate representation (IR). Therefore, Reticle serves as a synthesis tool chain that uses an alternative to classical RTL representation. Figure 4 details its translation flow: Reticle IR is combined with a target specific file describing patterns of implementation choices. The Reticle IR is first transformed into an parametrized assembly code (ASM). The choice of the parameters amounts to *place* the components on the FPGA board. The resulting circuit is generated as a Verilog netlist that can then be sent to the FPGA.

Reticle IR is a three address code using grammar presented in Figure 5. We recall that three address expression do not allow nesting of operators but instead require each subterm to be associated with a dedicated variable. A Reticle program consists in a main function composed of a list of instructions. Instruction can either be a ”wire” or a ”primitive” instruction. Wire instructions denote simple wiring signals (eg: assignment, extraction, concatenation) it uses wire operators $\oplus$ (eg: id, cat, ext ...) while primitive instructions denote signal processing with primitive operators such as arithmetic, bitwise or memory operations. In both cases, a sequence of integer parameters i^* allows to specify bit indices for these operations.

$fun \in Function ::= n(v:\tau)^* \rightarrow (v:\tau)^+\{ins^+\}$
$ins \in Instruction$ ::=wire | prim
$wire \in Wire ::= v:\tau = \oplus[i^*](v^*)$
$prim \in Primitive ::= v:\tau = \boxplus[i^*](v^+)@res$
$res \in Resource$ ::=?? | ρ
$\rho \in ResourceType$ ::= lut | dsp | lram | bram | uram
$\oplus \in WireOp$
$\boxplus \in PrimOp$
$i \in \mathbb{Z}$
$v \in Variable$
$\tau \in bool, int, \overrightarrow{int}$
$?? \in Wildcard$

Fig. 5: IR Grammar

```
def main(a: bool, b: bool, en:bool) -> (y: bool)
    {
    t0:bool = and(a, b);
    y:bool = reg[0](t0, en);
}
```

Listing 2: IR code example

Primitive instructions $\boxplus$ can be explicitly associated with a set of hardware resources using the suffix notation `@: ??` denotes a wildcard while a LUT, a DSP or different kinds of RAM can be requested. Last, Reticle variables are of type boolean, integer or vector of integers. The *instruction selection* phase amounts to replace these primitives instructions in Reticle by their FPGA-specific components.

A *target description language* (TDL) file describes pairs of *patterns* (pat) and *implementation* (imp). These pairs of patterns and implementations are used to match IR instructions, or sequences of such, and match them with corresponding FPGA primitives and configuration parameters using Reticle assembly syntax. As an example, one can provide different TDLs with different modeling choices: a first one relying mainly on LUTs to perform the computation versus a second one relying more on DSP. This approach gives Reticle the ability to control the inference process and to guarantee the usage of specific primitives. The matched primitives constitute the assembly code that will be placed. To place the primitives, Reticle uses the Z3 [7] SMT solver. Each free integer parameter corresponds to the position of the component on the FPGA (in the slices presenting on Figure 1). Last, translation between the placed assembly code and Verilog input format is direct.

Listings 2, 3, 4 and 5 illustrate IR processing to ASM using TDL. This program performs the same AND operation than the previous Verilog code (cf. Listing 1).

```
def main(a:bool, b:bool, en:bool) -> (y:bool) {
    y:bool = lut_and_b(a, b, en) @lut(0, 0);
}
```

Listing 3: ASM code example

```
pat lut_and_b[lut, 1, 2](a: bool, b: bool, en:
    bool) -> (y: bool) {
    t0:bool = and(a, b);
    y:bool = reg[0](t0, en);
}

imp lut_and_b[x, y](a: bool, b: bool, en:bool) ->
     (y: bool) {
    t0:bool = lut2[tbl=8](a, b) @a6lut(x, y);
    y:bool = fdre(t0, en) @aff(x, y);
}

pat dsp_and[dsp, 1, 1](a: bool, b: bool) -> (y:
    bool){
   y:bool = and(a, b);
}

imp dsp_and[x,y](a: bool, b: bool) -> (y: bool){
   y: bool = dsp[op=and](a,b,en,en,en) @alu(x,y);
}
```

Listing 4: TDL example

```
module main (
    input wire clock,
    input wire reset,
    input wire a,
    input wire b,
    input wire en,
    output wire y
);
    wire gnd;
    wire vcc;
    wire n0;
    GND GND (
        .G(gnd)
    );
    VCC VCC (
        .P(vcc)
    );
    (*LOC = "SLICE_X0Y0", BEL = "A6LUT"*)
    LUT2 # (
        .INIT(4'h8)
    ) __n0 (
        .I0(a),
        .I1(b),
        .O(n0)
    );
    (*LOC = "SLICE_X0Y0", BEL = "AFF"*)
    FDRE __y (
        .C(clock),
        .CE(en),
        .D(n0),
        .Q(y),
        .R(reset)
    );
endmodule
```

Listing 5: Netlist example

$$
\begin{array}{rcl}
bt & ::= & real \mid bool \mid int \\
d & ::= & \textbf{node}\ f\ (p)\ \textbf{returns}\ (p);\ \textbf{vars}\ p\ \textbf{let}\ D\ \textbf{tel} \\
p & ::= & x : bt;\ ...;\ x{:}bt \\
D & ::= & pat = e; D \mid pat = e; \\
pat & ::= & x \mid (pat, \ldots, pat) \\
e & ::= & v \mid x \mid (e,...,)e \mid e \rightarrow e \mid op(e,...,e) \\
 & & \mid \textbf{pre}\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
v & ::= & i{::}int \mid r{::}real \mid true{::}bool \mid false{::}bool
\end{array}
$$

Fig. 6: Subset of Lustre

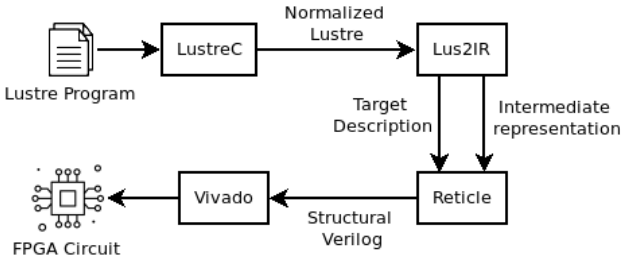


Fig. 7: Lustre to FPGA compilation toolchain

```
1 node main(a,b:bool) returns(y: bool);
2 var t0: bool;
3 let
4    t0 =  a and b;
5    y = 0 -> pre t0;
6 tel
```

Listing 6: Lustre code example

$$
\begin{array}{rcl}
d & ::= & \textbf{node}\ f\ (p)\ \textbf{returns}\ (p);\ \textbf{vars}\ p\ \textbf{let}\ D\ \textbf{tel} \\
\tilde{D} & ::= & pat = \tilde{e}; D \mid pat = \tilde{e}\ ; \\
l & ::= & v \mid x \\
\tilde{e} & ::= & l \mid true \rightarrow false \mid op(l,...,l) \mid \textbf{pre}\ l \\
 & & \mid \textbf{if}\ l\ \textbf{then}\ l\ \textbf{else}\ l
\end{array}
$$

Fig. 8: Normalized Lustre

D. Lustre: a synchronous dataflow language

Lustre [5] is a synchronous data flow programming language used in aerospace industry for the development of safety critical application software. In this programming paradigm for real-time systems, the function part is separated from the timing aspect. In Lustre, a node denotes the processing of discrete time input signal into discrete time output ones. Computation is assumed to be instantaneous. When used on the real-time platform, it is the duty of the engineer to ensure that the worst-case execution time of the node is compatible with the execution rate of the function.

A Lustre program is composed of nodes and follows the grammar given in Figure 6. The body of a node contains (unsorted) equations dedicated to data processing. They rely on expressions that use arithmetic and comparison operators, functional construct (**if-then-else**), temporal operators (**pre**, follow by ($\rightarrow$)). Temporal operators allow to access value of an expression at the previous time instant and to initialize these signals. Lustre also includes a notion of clocked expressions, node restarts or construction to describe automata, but it is omitted here to keep the presentation simple. A valid Lustre node shall not define a cyclic dependency between its signals without an explicit call to the temporal operator **pre**.

Listing 6 shows an example of a simple Lustre program. The internal signal `t0` is the conjunction of input flows `a` and `b`. The output flow returns this value `t0` at the previous time instant.

The similarities between the synchronous dataflow programming paradigm and the register transfer level representation of a system is attractive. Instead of relying on the existing HLS approach that compile these dataflow models into C and rely on the classical toolchain to produce the final circuit, we aim at building a more predictable toolchain, maintaining the structure of the Lustre model, supporting traceability of the compilation process.

III. Compiling Lustre into Reticle

Here, we describe the compilation scheme from Lustre to Reticle. Figure 7 shows the different steps we propose to produce netlist from Lustre program. When considering the examples of Listings 2 and 6, we can observe similarities between Lustre and Reticle IR. A first step of the flow is the production of preprocessed Lustre model, called *normalized Lustre*, to support a more direct translation to Reticle IR. The second step is the translation of this normalized Lustre into Reticle IR, while producing the associated TDL. We can then rely on Reticle to generate the netlist, placing the components on the board, before configuring the FPGA with its vendor tool Vivado.

A. Lustre preprocessing

Lustre compilation into C code can be performed efficiently while preserving the structure of the initial model. This is the modular compilation of synchronous dataflow language [2], used in most state of the art compilers.

This compilation is performed in three steps. The first one translates the input language into a smaller subset, called *normalized Lustre*. New variables are introduced to decompose automaton in pure dataflow code, sub-expressions involving node calls are bound to fresh variables, etc. The next two steps process the produced Lustre subset into an imperative data-structure (step 2) and produce the final imperative code, typically C (step 3).

In [4] a more specific preprocessing was proposed to support the translation of Lustre into Simulink model. Here, we further extend this preprocessing to inline all nodes into a single one and only allow three address expression. However, we bypass the next phases of the compilation, avoiding the translation as an imperative program. The restricted syntax for normalized Lustre is described in Figure 8

```
node AssignExample (x: bool) returns (y: bool);
let
y = x ;
tel
```

(a) Lustre assignment

```
def main(x: bool) -> (y: bool) {
y = id(x);
}
```

(b) IR assignment

Fig. 9: Assignment

B. Optimization choices

Previously, we advocate for the use of advanced primitives such as DSPs. Indeed the use of DSPs is relevant for circuits that require a significant number of arithmetic operations, because this kind of resources was created and optimized specifically for this kind of operation.

Therefore, to ensure high-speed performance of the circuit, we have prioritised the implementation of DSP for arithmetic logic units. This choice is modelled in the generated Reticle TDL file, associating DSP uses in the TDL patterns. Since DSPs are generally less common on FPGAs, it should still be possible to implement small operations using LUTs.

C. Direct translation

Each signal definition in the normalized Lustre shall now be translated to a set of Reticle IR instructions. In addition, a set of TDL patterns will be produced to drive the generation of the final circuit for a specific FPGA architecture.

1) Assignment: Assignment corresponds to a wire instruction in Reticle. It does not require a specific inference pattern. See Listings 9a and 9b

*2) Memory with **pre**:* Let us consider a variable `D` at time `t`, the output of the Lustre `pre D` statement is the value of `D` at time `t-1` as illustrated as follows:

-	t0	t1	t2	t3	t4
D	1	2	3	4	5
pre D	x	1	2	3	4
Q	**0**	1	2	3	4

Here, the element x at time $t0$ for the signal `pre D` denotes an undefined value. In terms of hardware representation, this behaviour corresponds to a *Parallel-In Parallel-Out* (PIPO) shift register. PIPO registers consist of a cascade of flip-flops, meaning that, in Reticle, the detected pattern must be a register and its implementation the flip-flop cascade. To avoid data uncertainty at t0, all the flip-flops of the PIPO must be set to 0, as illustrated on the line `Q` is the above table. Listing 10a, 10b and 10c denote respectively the Lustre expression, the IR associated construct and the TDL pattern/implementation pair. Figure 11 illustrates the PIPO register, each flip flop store a bit of the four bit integer describe in the TDL file.

3) true → false: After normalization (preprocessing), the resulting Lustre node contains at most one of such expression. This signal allows to distinguish initial state from later ones. A variable is first set to `true` and then switch to `false`, as illustrated in the following table:

```
let
Q = pre D ;
tel
```

(a) Lustre pre

```
    Q:i4 = reg[0](D,en);
}
```

(b) IR reg

```
    y:i4 = reg[0](a, en);
}

imp regi4[x, y](a: i4, en: bool) -> (y: i4) {
    D0:bool = ext[0](a);
    D1:bool = ext[1](a);
    D2:bool = ext[2](a);
    D3:bool = ext[3](a);
    Q0:bool = fdre(t0, en) @aff(x, y);
    Q1:bool = fdre(t1, en) @bff(x, y);
    Q2:bool = fdre(t2, en) @cff(x, y);
    Q3:bool = fdre(t3, en) @dff(x, y);
    y:i4 = cat(t4, t5, t6, t7);
}
```

(c) TDL reg

Fig. 10: Pre

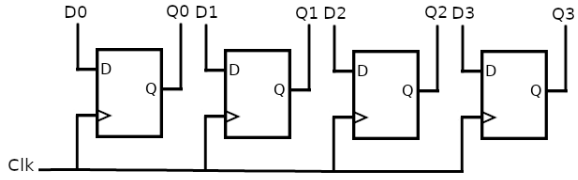


Fig. 11: 4-bit PIPO shift register

-	t0	t1	t2	...	tn
x	true	false	false	false	false

At circuit level, this behaviour can be obtained with a single D flip-flop initialized to `true` while its input is connected to the `false` constant. Listings 12a, 12b and 13b represent respectively the Lustre model, the IR definition and the associated TDL pattern, producing a flip-flop gate in the resulting circuit.

4) op(l,...,l) : This statement in Lustre enables standards arithmetic (addition, subtraction, multiplication) and boolean operations (and, or, xor, not, is equal to, is greater than ...), an example is given in listing 13a. Hardware components that perform this kind of operations are arithmetic and logic unit (ALU). As explained previously, for performances and predictability reason we decided to prioritize DSP implementation (see Figure 1)for this kind of circuit. With Reticle IR operators are expressed with their corresponding primitive operator. See Listings **??** and 13c for both the IR and TDL, supporting DSP inference.

5) if l then l else l: Conditional expression in Lustre is a functional operator. Both branches are eventually computed while the proper value is propagated. This acts as a multiplexer. The statement presented in Listing 14a is therefore mapped in IR to an expression involving `mux` (Listing 14b).

```
let
Q = true -> false ;
tel
```

(a) Lustre tfby

```
    t0:bool const[0]
    Q:bool = reg[1](t0,en);
}
```

(b) IR tfby

```
    t0:bool = const[0];
    y:bool = reg[1](t0, en);
}

imp regi8[x, y](en: bool) -> (y: i8) {
    t0:bool = const[0];
    y:bool = fdre(t0, en) @aff(x, y);
}
```

(c) TDL tfby

Fig. 12: true → false

```
let
    y = a + b ;
tel
```

(a) Lustre tfby

```
    t0:bool = const[0];
    y:bool = reg[1](t0, en);
}

imp regi8[x, y](en: bool) -> (y: i8) {
    t0:bool = const[0];
    y:bool = fdre(t0, en) @aff(x, y);
}
```

(b) IR op

```
   y:i16 = add(a, b);
}

addi16[x,y](a: i16, b: i16) -> (y: i16){
   y: i16 = dsp[op=add](a,b,en,en,en) @alu(x,y);
}
```

(c) TDL op

Fig. 13: true → false

This can be obtained in Reticle with a *two-to-one multiplexer* (MUX) (see Listing 14b), or a cascade of MUX for integer datatypes as we can see on Figure 15. This kind of function can be created with LUTs primitives (see Listing 14c for an if-then-else of `int4`), the first eight lines (lines 7-14) extract the four bits of `a` and `b`, these extracted bits are then given to LUT3 components (lines 15-18). LUT3 are lookup tables configured to takes three input bits and returning one output bit. These LUT3 select a bit from `a` and `b` according to `sel`. Finally selected bits are stored in registers and concatenated.

D. Toolchain properties

a) Efficiency: The toolchain translates a Lustre program into an implementable FPGA circuit, in an automated fashion. It simplifies the code production task without requiring hardware design expertise.

```
let
    y = if s then a else b ;
tel
```

(a) Lustre Lustre if-then-else

```
    y:i4 = mux(s,a,b);
}
```

(b) IR mux

```
    t0:i4 = mux(sel, a, b);
    y:i4 = reg[0](t0, en);
}

imp muxi4[x, y](S: bool, a: i4, b: i4, en: bool)
    -> (y: i4){
    A0:bool = ext[0](a);
    B0:bool = ext[0](b);
    A1:bool = ext[1](a);
    B1:bool = ext[1](b);
    A2:bool = ext[2](a);
    B2:bool = ext[2](b);
    A3:bool = ext[3](a);
    B3:bool = ext[3](b);
    Y0:bool = lut3[tbl=0xac](t0, t1, S) @a6lut(x,
        y);
    Y1:bool = lut3[tbl=0xac](t2, t3, S) @b6lut(x,
        y);
    Y2:bool = lut3[tbl=0xac](t4, t5, S) @c6lut(x,
        y);
    Y3:bool = lut3[tbl=0xac](t6, t7, S) @d6lut(x,
        y);
    t12:bool = fdre(t8, en) @aff(x, y);
    t13:bool = fdre(t9, en) @bff(x, y);
    t14:bool = fdre(t10, en) @cff(x, y);
    t15:bool = fdre(t11, en) @dff(x, y);
    y:i4 = cat(t12, t13, t14, t15);
}
```

(c) TDL mux

Fig. 14: if-then-else

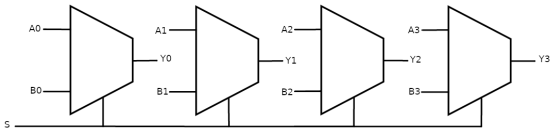


Fig. 15: cascade of Mux Primitive

b) Predictability: Pattern matching with TDL controls the inference process. Resulting complexity in terms of DSP/-LUT usage is known *a priori*.

c) Computation speed: Usage of dedicated components ensures high-speed computation for arithmetic operations.

d) Traceability: Inference steps are known, supporting the traceability from hardware instruction up to Lustre model expressions.

e) Fast design process: With the explicit choice of FPGA components to be used to implement Lustre and IR operators, the synthesis step is faster than regular HLS, but can be potentially less optimized.

```
node pi (fb , kp, ki, cmd: int; ) returns (u:int)
var eps: int;
let
    eps = 0 -> cmd - fb;
    u = 0 -> kp * eps + kp * pre eps + ki * eps +
         pre u;
tel
```

Listing 7: Lustre Code

E. Reticle extension for Xilinx 7 series FPGA

The Reticle version used [1] enables the production of placed structural Verilog for Xilinx Ultrascale+ [14] FPGA family. Our target is a Xilinx Artix 7 FPGA [15] on an Arty A7 100T development board. Differences between these two families lie on their primitive types, quantity of slices and organization of such. The main differences between these FPGA families are their slices architecture, Figure 16a shows the architecture of an Ultrascale+ slice while Figure 16b shows an Artix 7 slice. Ultrascale+ slices contain more LUT and Flipflop Primitives than Artix 7 slices. This difference leads to different implementation of Artix 7 TDL file. Artix 7 LUT BELannotations are from `a5lut` to `d5lut` and `a6lut` to `d6lut` instead of `a5lut` to `h5lut` and `a6lut` to `h6lut`. Same for Flip Flop BEL, they go from `aff` to `dff` and `a5ff` to `d5ff` instead of `aff` to `hff` and `aff2` to `hff2`. Also the Carry primitive are not the same in an Artix7 and an Ultrascale slice: Artix 7 contains different carry primitive (`carry4` instead of `carry8`). In addition to slice architecture differences, Slice and DSP matrix doesn't have the same size. there is 3*80 Slices and DSP on an Artix 7 FPGA while there is 3*72 Slices and 5*72 DSP on an Ultrascale+ FPGA. We implemented these changes into the Reticle placer source code. Finally, the DSP primitives existing in Artix 7 FPGA (`DSP48E1`) differ from the ones of the Ultrascale (`DSP48E2`). The next board configuration has been implemented in Recticle.

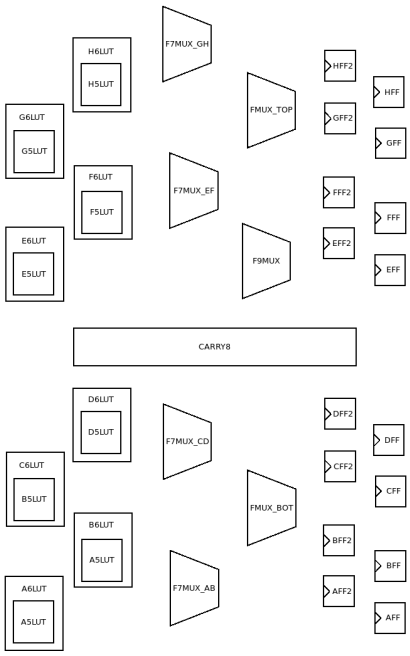
IV. Experimentation

The proposed process has been applied on a simple representative example: a proportional integral (PI) controller. We report here the comparison of three different methods to produce the final circuit: (1) traditional RTL implementation, (2) our proposal (Lus2IR), and (3) high-level synthesis from Vitis. The current implementation is straightforward without any pipelining nor optimized placement.

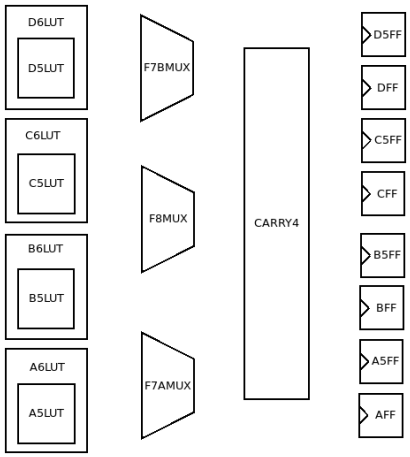
A. Lustre to FPGA

Listing 7 represents the Lustre model performing the PI function. The control signal `u` is computed according to command signal `cmd` and the feedback loop with proportional gain `kp` and integral gain `ki`. This program relies on a stateful variable `eps` containing the error between the command and the feedback loop. Output variable `u` is also stateful since it depends on its previous state.

[1]Available at https://github.com/vegaluisjose/reticle-evaluation



(a) Ultrascale slice



(b) Artix 7 slice

Fig. 16: Comparison of Ultrascale (a) and Artix 7 (b) slices.

Processing this Lustre model with the steps presented in the previous sections produces the IR code in Listing 8. We can observe the decomposition of Lustre code in three address code. To enable register control, clock enabling signal `en` has been inferred during the code production step.

This IR code is given to Reticle with a target description given in Listing 9. Reticle is then able to produce the circuit presented on Figure 17. This synthesized circuit is then given to Vivado for placement and bitstream production and importation. Since Reticle only places the DSPs, LUTs and FDREs but not the IOs so we let Vivado do the placement, bypassing the Reticle Z3-based placement algorithm.

```
def main(cmd:i16, fb:i16, ki:i16, kp:i16, en:
    bool) -> (u:i16) {
  pi_3:i16 = reg[0](u, en) ;
  pi_5:i16 = reg[0](eps, en) ;
  u:i16 = id (pi_11) ;
  pi_11:i16 = mux(pi_1, pi_2, pi_10) ;
  pi_10:i16 = add (pi_9, pi_3) ;
  pi_9:i16 = add (pi_8, pi_4) ;
  pi_4:i16 = mul (ki, eps) ;
  pi_8:i16 = sub (pi_7, pi_6) ;
  pi_6:i16 = mul (kp, pi_5) ;
  pi_7:i16 = mul (kp, eps) ;
  eps:i16 = id (pi_13) ;
  pi_13:i16 = mux(pi_1, pi_2, pi_12) ;
  cst0:bool = const[0] ;
  pi_1:bool = reg[1](cst0, en) ;
  pi_12:i16 = sub (cmd, fb) ;
  pi_2:i16 = const[0] ;
  }
```

Listing 8: IR Code

```

pat dsp_add_i16[dsp, 1, 1](a: i16, b: i16) -> (y:
     i16){
   y:i16 = add(a, b);
}

pat dsp_sub_i16[dsp, 1, 1](a: i16, b: i16) -> (y:
     i16){
   y:i16 = sub(a, b);
}

pat dsp_mul_i16[dsp, 1, 1](a: i16, b: i16) -> (y:
     i16){
   y:i16 = mul(a, b);
}

pat lut_mux_i16[lut, 1, 2](sel: bool, a: i16, b:
    i16) -> (y: i16) {
    y:i16 = mux(sel, a, b);
}

pat lut_reg_0_bool[lut, 1, 2](a: bool, en: bool)
    -> (y: bool) {
    y:bool = reg[1](a, en);
}

pat lut_reg_0_i16[lut, 1, 2](a: i16, en: bool) ->
     (y: i16) {
    y:i16 = reg[0](a, en);
}
```

Listing 9: Target description pattern

B. Results

Figures 18 and 19 show resource analysis and timing analysis of the circuit obtained with RTL synthesis, proposed compilation tool chain and Vitis High-level synthesis. Resource analysis highlights the resource economy realized by Reticle, while timing analysis reveals comparable computation speed between the RTL inferred circuit and Reticle circuit. As expected the number of DSP, on Reticle inferred circuit, correspond to the number of arithmetic operations in the PI controller. The quantity of flipflop and lookup tables are also proportional to the number of registers and multiplexer of the

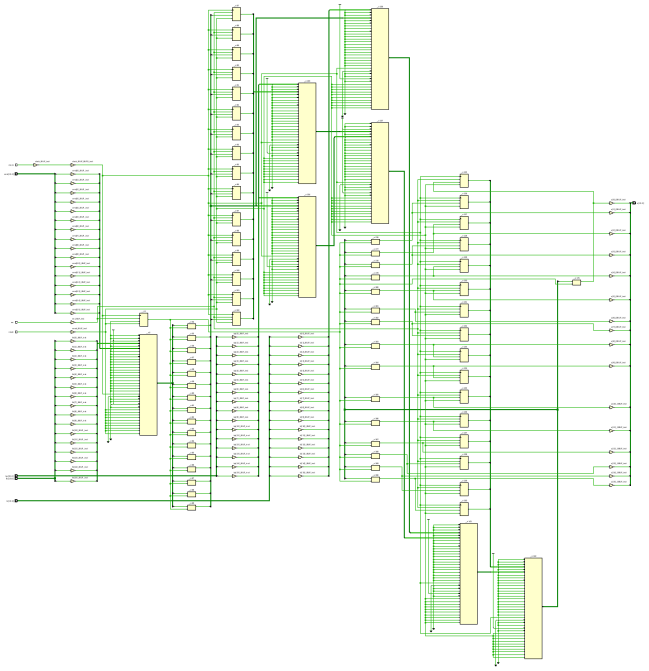


Fig. 17: PI controller circuit

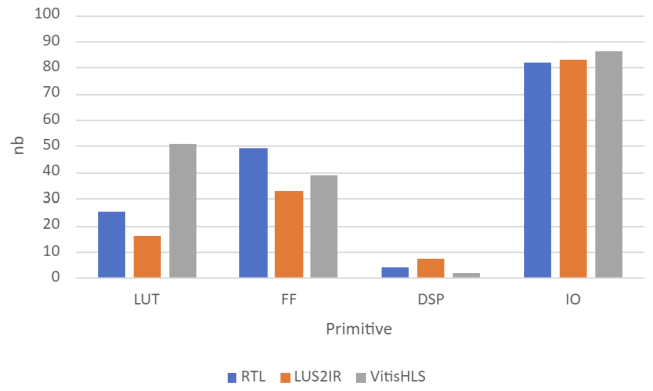
intermediate representation. This observation let us conclude that Reticle synthesis is highly predictable. The results of the timing analysis must be interpreted with caution. Indeed, the maximum frequency of the circuit can be increased at the cost of adding latency, which we have not measured.

One of the unanticipated results is the duration of the synthesis process. While legacy tools could produce optimized code by exploring a large variety of implementation choice, our process is more direct and more traceable. It is also much faster since Reticle only applies the pattern specified in the TDL produced.

As an example on this very simple PI example, Vivado takes 44 seconds to compile an input RTL into the netlist, without the placement, while our Lustre-Reticle based approach takes 0.31 second.

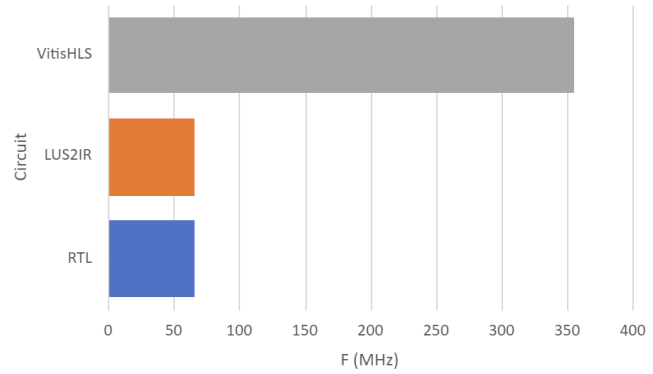
V. Related works

While FPGAs were introducing in the 80s, their programming is still an actual concern. Del Sozzo *et al* [11] recently published a thorough report reviewing the different means to facilitate FPGA design process. One of the main approaches to raise the abstraction level and allow developers to write FPGA program is high-level synthesis. Nowadays, several HLS toolchains exist and are able to produce efficient RTL code. Among these we can mention Vitis HLS [1], the Xilinx high-level toolchain that translates C++ into RTL and HDL-Coder [6] a Matlab library that translates Simulink models into RTL. In both cases, the input model is first translated into RTL, mainly through a first C code or imperative code translation. These approaches do not specifically target netlists and lose the initial structure of the input program. In addition, the



This figure presents the number of LUTs, Flip-flops (FF), DSP and input/output ports (IO) used by each generated circuits. As requested and expected, our toolchain inferred less LUTs and FF than other circuits and uses more DSP instead.

Fig. 18: Resource analysis



On this PI example, VitisHLS circuit is seven times more efficient than a direct synthesis from RTL or using our Lustre/Reticle toolchain.

Fig. 19: Timing analysis

input program shall not be an arbitrary C code. To summarize all these HLS frameworks still requires deep expertise in hardware design to be usable.

The programming of FPGA from Lustre model has already been addressed, about 15 years ago, by the Gencod project [8] and, with a similar approach, with the Lava project [3]. In these approaches, the resulting HDL program is using RTL and, therefore, shares the same feature as the HLS framework: the RTL encoding requires Vivado or similar tools to translate the RTL into netlist and lose program structure.

Our work provides a more predictable compilation process that takes in account recent evolution of FPGA architecture for the translation of Lustre into FPGA circuit. We also use an alternative to RTL representation targeting directly netlists.

VI. Conclusion

We focused on the current need to support the design of FPGAs. of FPGAs. FPGAs are now seen as a serious alternative to CPUs for space applications, but their design process still requires deep hardware design expertise. With our toolchain, we propose to support the design of such FPGA for GNC applications written in dataflow languages such as Simulink or Lustre. Our toolchain translates the input model in Lustre directly into a netlist, a circuit description. Our method preserves the structure of the input model in the final circuit, and is therefore best suited for embedded critical applications where traceability is paramount to ensure confidence in the system. We have evaluated our toolchain on a simple proportional integral controller. Our experiment showed comparable circuit timing performance between classical RTL synthesis and our Lustre/Reticle synthesis, at a reduced resource cost, while providing both traceability information and predictability in the resulting circuit components. Timing performance remains lower than that of the resulting Vitis HLS circuit, but on par with other methods based on RTL synthesis. This first framework opens up numerous perspectives for modelling and analysis into the hardware design world. We plan to use static analysis methods at the model level to optimise at bit level the fixed-point format to be used in the circuit. We also plan to rely on methods typically used to improve numerical accuracy, but applied here to optimise the FPGA design process. Another perspective concerns the extension of the input language to a larger subset of Lustre, e.g. including logical clocks. Finally, we are currently working on a more realistic use case, representative of space applications.

References

[1] Soujanya Bhowmick. Optimizing Transformer Inference on FPGA: A Study on Hardware Acceleration using Vitis HLS, 2023. Master thesis. Aalto University. Master's Programme in Electronics and Nanotechnology (TS2013).

[2] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In Krisztián Flautner and John Regehr, editors, *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*, pages 121–130. ACM, 2008.

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, page 174–184, New York, NY, USA, 1998. ACM.

[4] Hamza Bourbouh, Pierre-Loïc Garoche, Christophe Garion, and Xavier Thirioux. From Lustre to Simulink: Reverse Compilation for Embedded Systems Applications. *ACM Transactions on Cyber-Physical Systems*, 5(3):1–20, July 2021.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 178–188, New York, NY, USA, 1987. ACM.

[6] Serena Curzel, Michele Fiorito, Patricia Lopez Cueva, Tiago Jorge, Thanassis Tsiodras, and Fabrizio Ferrandi. Exploration of Synthesis Methods from Simulink Models to FPGA for Aerospace Applications. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 243–249, Bologna Italy, May 2023. ACM.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Adrien Guatto and Marc Pouzet. Rapport d'étude sur la traduction de SCADE/Lustre vers VHDL. Technical report, Ecole normale ´ supérieure, 2010. Projet GENCOD.

[9] Juan L. Jerez, Paul J. Goulart, Stefan Richter, George A. Constantinides, Eric C. Kerrigan, and Manfred Morari. Embedded Online Optimization for Model Predictive Control at Megahertz Rates. *IEEE Transactions on Automatic Control*, 59(12):3238–3251, December 2014.

[10] Special C. RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011.

[11] Emanuele Del Sozzo, Davide Conficconi, Alberto Zeni, Mirko Salaris, Donatella Sciuto, and Marco D. Santambrogio. Pushing the Level of Abstraction of Digital System Design: A Survey on How to Program FPGAs. *ACM Computing Surveys*, 55(5):1–48, May 2023.

[12] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. Reticle: a virtual machine for programming modern FPGAs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 756–771, Virtual Canada, June 2021. ACM.

[13] Michael Wirthlin. High-Reliability FPGA-Based Systems: Space, High-Energy Physics, and Beyond. *Proceedings of the IEEE*, 103(3):379–389, March 2015.

[14] Xilinx. UltraScale Architecture Libraries Guide, 2023.

[15] Xilinx. Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide, 2023.