Custom Arithmetic Datapath Design for FPGAs using the FloPoCo Core Generator

Florent de Dinechin, Bogdan Pasca
LIP (ENSL-CNRS-Inria-UCBL), École Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France
Email: florent.de.dinechin@ens-lyon.fr, bogdan.pasca@ens-lyon.org

***Abstract*—Reconfigurable circuits have a strong potential as acceleration engines. However, using them efficiently requires much design effort compared to classical software programming. The FloPoCo open-source core generator project addresses this issue for a restricted class of circuits that is central to reconfigurable computing: arithmetic datapaths.**

The FloPoCo framework clearly isolates the two main design issues for such datapaths: implementing the correct mathematical function, and pipelining it to an arbitrary frequency.

The function is expressed in FloPoCo as a combinatorial VHDL circuit. The design of this circuit is assisted by a powerful C++ framework for VHDL generation, allowing a designer to program complex optimizations around the VHDL itself. It also provides high-level, function-based testbench generation.

The issue of pipelining is then completely automated. FloPoCo automatically builds correct-by-construction pipelines optimized for a wide range of target FPGAs and target operating frequency.

FloPoCo is shown to be useful for a wide spectrum of productivity/efficiency trade-offs. At one end, it automatically converts C-like straight-line code into a parameterized and pipelined floating-point datapath. At the other end, it assists expert designers in building complex FPGA-specific operators.

***Keywords*-FloPoCo; core generator; arithmetic circuit; pipelining; datapath**

I. Introduction

FPGAs are increasingly being considered as application accelerators. They are especially relevant for applications which 1/ expose parallelism, and 2/ require arithmetic operations not well supported in hardware by mainstream processors. Examples include novel cryptography algorithms, Monte Carlo simulations requiring massive amount of random numbers, digital signal processing, and many others.

Translating an application into an optimized FPGA design has always been a tedious task. Emerging high-level synthesis approaches [1] ease this task but often restrict the class of applications and trade efficiency for productivity.

A. Arithmetic datapath design

Here we address the design of *parameterized, pipelined arithmetic datapaths* for FPGAs.

An arithmetic datapath is the implementation of some function, where the word "function" is used in its mathematical sense: $f(X) = Y$ where $X = x_0, ..., x_{i-1}$ is a set of inputs and $Y = y_0, ..., y_{j-1}$ is a set of outputs. Examples of such functions include the basic operations, elementary functions such as sine or exponential, complex multiplication $x + iy = (a + ib)(c + id)$, or even a Fourier transform.

This mathematical function defines a reference value for each value computed by a datapath implementing it. For functions on integer or finite-field, the datapath should return the exact same value as the mathematical function. For functions over the reals, the datapath must provide an *approximation*. This approximation is constrained by the format chosen for the inputs and output, typically some fixed-point or floating-point format. The availability of a mathematical reference enables *data-efficient* implementations, as discussed in Sidebar 1. It also enables *testing* the operators against this reference, as will be detailed in Section IV.

Sidebar 1 Data efficiency for numerical datapaths

For FPGAs, an often overlooked measure of efficiency is whether each of the bits that is carried along an application holds useful information. Let us take an example.
A single-precision exponential function with a relative error of 2^{-8} may make sense in a software. The processor imposes the single-precision floating-point format, that is a 32-bit data granularity. Due to the 23-bit significand, no single-precision exponential may offer a relative accuracy better than 2^{-24}. However, some applications will be contented with a much worse accuracy, say 2^{-8}. If this saves resources or time, it makes sense to provide such a degraded implementation. Half of the 32 output bits will contain noise instead of useful information, but removing them from the processor datapath is not an option.
In an FPGA, it is an option, as data formats are much more flexible. Using a single precision exponential core accurate to 2^{-8} means that, out of the 32 bits passed along the datapath, 16 bits hold useless noise. Obviously, this entails a waste of precious routing resources and registers. Besides, the subsequent operators in the datapath will compute on this noise, meaning more wasted resources and useless power consumption.
Therefore we claim that no operator should be designed that is not accurate to its last bit. If an application, at some point, requires a relative accuracy of 2^{-8}, then the floating-point format it uses at this point should have only 8 bits of significand.
Design optimally data-efficient datapaths, i.e. defining which accuracy is required at each point of a datapath (possibly including *guard bits* to absorb rounding errors), may take considerable design effort. FloPoCo helps a designer to tackle this challenge in two ways. Firstly, all the supplied operators are last-bit accurate. Secondly, they are all parameterized by the precision, enabling design-space exploration that includes precision tuning.

B. The FloPoCo project

FloPoCo (**Flo**ating-**Po**int **Co**res, but not only) is an open-source[1] C++ framework for the generation of arithmetic datapaths. It provides a command-line interface that inputs operator specifications, and outputs synthesizable VHDL.

Each datapath generator in FloPoCo is a C++ class. At the lowest level, the task of such a class consists in printing VHDL code to a specific C++ stream. Therefore, FloPoCo embeds the full expressive power of VHDL, and is relatively easy to get started with for the VHDL-literate.

C. Assisted pipeline design

Arithmetic datapaths as we defined them can be implemented as combinatorial functions. Then, a way to exploit the inherent parallelism of FPGAs for better performance is to pipeline these combinatorial implementations. Pipelining is conceptually easy, but in practice it is a tedious and error-prone task. The FloPoCo framework has been designed to let the programmer focus on the high-level aspects of this task, and automate the rest. Here are the main features of pipeline generation in FloPoCo, further discussed in Section III.

1) Frequency-directed pipeline: Pipelining involves a trade-off between *latency* (number of pipeline levels, or number of clock cycles needed for the computation) and *frequency* (or *throughput*). Most core generators let the user specify the latency. In FloPoCo, on the contrary, the user specifies a frequency, and the datapath is pipelined for this frequency. This approach (also used in recent work by Perry [2]) is preferred because enables *composition*: a large component operating at frequency f can be built by assembling smaller components designed to operate at frequency f.

2) Target-specific pipeline tuning: As the frequency of a given design is strongly dependent on the target FPGA, frequency-directed pipelining must be based on an abstract model of the capabilities of an FPGA, including timing information. FloPoCo comes with such models for main FPGA families from both Xilinx and Altera.

3) Pipeline in the hands of the designer: Ideally, a designer would write only the combinatorial version of an operator (focussing on its functionality), and let the tools pipeline this design for a given target FPGA and frequency [2], for instance using retiming [3], [4], [5]. However, the frequency may dictate fundamental changes in the architecture, for instance impose fast adders [6] or tables of precomputed values. The choice in FloPoCo is therefore that the designer has to program the pipeline construction. The framework makes this easy and safe through high-level notions such as cycles, synchronization, critical path, etc.

D. FloPoCo operator library

Besides, pipeline tuning has been done already for a wide range of useful components: FloPoCo provides an ever increasing *library of arithmetic cores*, each parameterized in size and following the frequency-directed pipeline paradigm. It includes integer, fixed-point and floating-point basic operators, sometimes in several variants (e.g. large pipelined adders [6], DSP-saving Karatsuba or truncated multipliers [7]). Some of these operators are specialized: a squarer, for instance, is a specialized multiplier that saves resources. This library also provides state-of-the art architectures for elementary functions, currently exponential [8], logarithm and power, and more to come. Finally, the FloPoCo library also includes meta-operators:

- several generators of multipliers by a constant (another example of specialization),
- `HOTBM` and `FunctionEvaluator`, two generators of polynomial evaluators for fixed-point functions [9],
- `FPPipeline`, a floating-point datapath generator that assembles a full floating-point datapath out of pseudo-C straight-line code.

The project web site provides a full list, and pointers to research articles describing most of these components.

[1]http://flopoco.gforge.inria.fr/

II. A MOTIVATING EXAMPLE

Consider a floating-point sum of squares: This datapath inputs three floating-point numbers X, Y and Z, and outputs a floating-point number for $X^2 + Y^2 + Z^2$.

A. FloPoCo command line

A first option is to assemble standard floating-point multipliers and adders. For this, the command line

```
flopoco -target=Virtex4 -frequency=200
         FPAdder 10 36
```

will generate synthesizable VHDL for a floating-point adder, pipelined to run at 200MHz on a Xilinx Virtex4, using a custom floating-point format with 10 bits of exponent and 36 bits of significand (this format is intermediate between the standard single- and double-precisions).

For design exploration, the 4 parameters we have in this example (target FPGA, frequency, exponent size and significand size) can be changed within sensible range. The frequency and the precision are orthogonal parameters, as they should be, and the pipeline depth is reported.

More complex datapaths can be obtained in seconds using the `FPPipeline` meta-operator of FloPoCo. Assume the file `SumOfSquares.txt` contains the following pseudo-program:

```
R = X*X + Y*Y + Z*Z;
output R;
```

then the command line

```
flopoco -target=Virtex4 -frequency=300
         FPPipeline SumOfSquares.txt 9 31
```

will generate the VHDL for a complete floating-point pipelined datapath.

B. Reconfiguring arithmetic

One of the main goals of the FloPoCo project is to encourage FPGA designers to use arithmetic operators beyond the “one size fits all” operators we are used to see in microprocessors. Exploring non-standard precisions is a start, but FloPoCo also offers FPGA-specific operators, operators that will probably never make sense in a general purpose processor

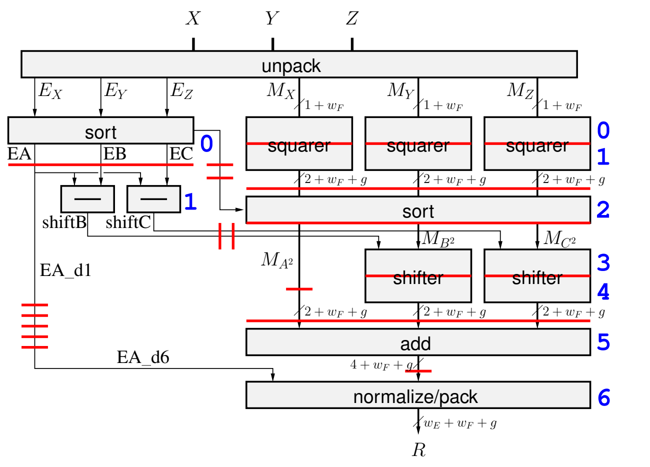


Fig. 1. A FloPoCo pipeline for the floating-point sum-of-square. E_X, E_Y and E_Z are the exponents (of size w_E bits) of the three inputs, M_X, M_Y and M_Z are their significands (of size $1 + w_F$ bits). Internal precision is extended with g guard bits, and $g = 3$ ensures last-bit accuracy of the result. The red lines are synchronization barriers for one example of pipeline.

because they do not occur often enough in general code. We already mentioned squarers: writing the pseudo-program as

```
R = sqr(X) + sqr(Y) + sqr(Z);
output R;
```

we may obtain a datapath which consumes less resources and has a slightly shorter latency (option 2 in Table I).

A third option is to design from scratch, for the function $x^2 + y^2 + z^2$, a fused datapath that

- recovers the intrinsic parallelism and symmetry of this expression,
- fuses the datapaths of the two additions,
- disposes of the logic that, in standard floating-point adders, manages addition of number of different signs [10], and
- avoids all intermediate roundings and normalizations, a task that Langhammer's floating-point compiler [11] automates,
- and returns a last-bit accurate result.

This datapath is presented in Fig. 1. Compared to one obtained by [11], it is better specified and embeds more optimizations, but took several days to write.

III. The FloPoCo VHDL generation framework

We here assume basic knowledge of object-oriented concepts with the C++ terminology. Fig 2 provides a very simplified overview of the FloPoCo class hierarchy.

A. Operators and VHDL generation

The core class is `Operator`: every datapath we design is an `Operator` (i.e. inherits this class). An `Operator` corresponds to a VHDL entity. Running FloPoCo constructs a list of `Operators` (those specified on the command-line, and all their sub-components), then generates the VHDL for them.

Fig. 3 describes this VHDL generation flow. The constructor method of each `Operator` places combinatorial VHDL code in the `vhdl` stream. At the same time, it builds up pipeline

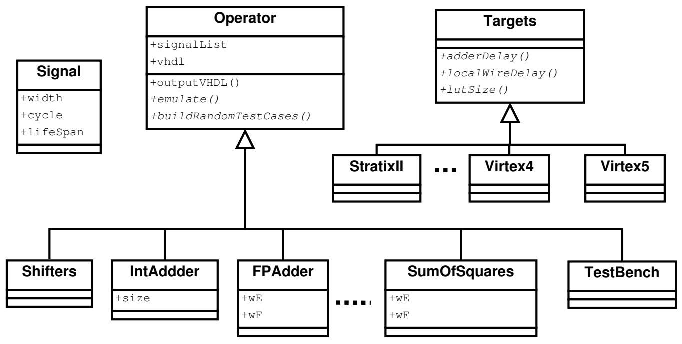
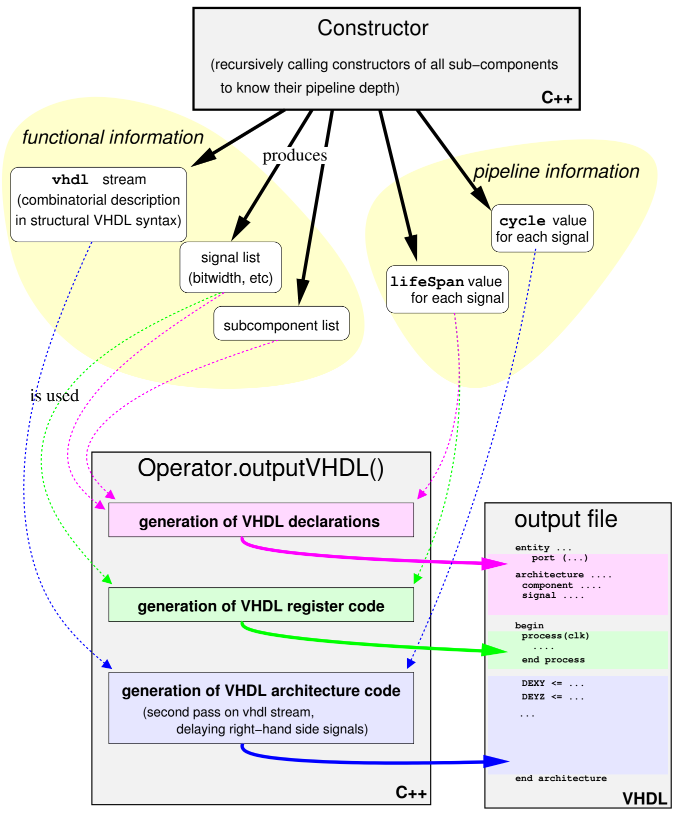


Fig. 2. FloPoCo class hierarchy (very simplified overview)



Constructor
(recursively calling constructors of all sub-components to know their pipeline depth)
C++
functional information
produces
pipeline information
vhdl stream (combinatorial description in structural VHDL syntax)
signal list (bitwidth, etc)
subcomponent list
lifeSpan value for each signal
cycle value for each signal
is used
Operator.outputVHDL()
generation of VHDL declarations
generation of VHDL register code
generation of VHDL architecture code (second pass on vhdl stream, delaying right-hand side signals)
C++
output file
entity ... port (...)
architecture .... component .... signal ....
begin process(clk) .... end process
DEXY <= ...
DEYZ <= ...
end architecture
VHDL

Fig. 3. Simplified overview of VHDL generation flow

information as described below. Then the `outputVHDL()` method combines the `vhdl` stream and the pipeline information to form the VHDL code of the pipelined datapath. It also declares all the needed VHDL signals, entities, components, etc, so that a designer only has to focus on the architectural part of the VHDL code.

An example of basic FloPoCo code is given by Listing 1. The corresponding generated VHDL is given by Listing 2. These listings describe the upper left part of Fig. 1 and will be explained below.

B. Pipelining basics

A pipeline of depth n is composed of $n + 1$ pipeline stages (numbered from 0 to n, in blue on Fig. 1), separated by synchronization barriers (red lines). In essence, pipelining consists in associating to every signal the number of the cycle

TABLE I
SOME SYNTHESIS RESULTS FOR $x^2 + y^2 + z^2$.

Productivity versus performance on Virtex4, target frequency $f = 350$ MHz			
format	approach	performance	cost
(8,23)	LogiCore	34 cycles @ 482 MHz	1356 slices, 12 DSP
	option 1	35 cycles @ 327 MHz	1279 slices, 12 DSP
	option 2	35 cycles @ 333 MHz	1043 slices, 9 DSP
	option 3	11 cycles @ 369 MHz	470 slices, 9 DSP
(11,52)	LogiCore	50 cycles @ 354 MHz	3074 slices, 48 DSP
	option 1	47 cycles @ 319 MHz	3859 slices, 48 DSP
	option 2	45 cycles @ 322 MHz	3137 slices, 18 DPS
	option 3	16 cycles @ 368 MHz	1866 slices, 18 DSP

Performance versus cost on Virtex4, option 3, varying target frequency			
format	target f	performance	cost
(10,36)	200 MHz	6 cycles @ 203 MHz	874 slices, 9 DSP
	100 MHz	2 cycles @ 109 MHz	809 slices, 9 DSP
	50 MHz	0 cycles @ 51 MHz	751 slices, 9 DSP
(11,52)	200 MHz	7 cycles @ 187 MHz	1285 slices, 18 DSP
	100 MHz	3 cycles @ 102 MHz	1272 slices, 18 DSP
	50 MHz	2 cycles @ 64 MHz	1130 slices, 18 DSP

Portability to different FPGAs, , target frequency $f = 200$ MHz			
format	FPGA	performance	cost
(10,36)	Virtex 5	5 cycles @ 196 MHz	1444L, 762 R, 9 DSP48E
	Stratix II	8 cycles @ 179 MHz	1395L, 1295 R, 18 9-bit elem
	Stratix IV	4 cycles @ 213 MHz	1529L, 792 R., 18 9-bit elem

Format is given as (exponent size, significand size). We provide a reference as LogiCore operators assembled by hand. Option 1 is FPPipeline, using multipliers. Option 2 is FPPipeline, using squarers. Option 3 is the fused datapath of of Fig.1. All these numbers were obtained in empty FPGAs using ISE 11.5 for Xilinx and QuartusII 9.1 for Altera.

where it is defined (its `cycle` attribute in FloPoCo), then using this information to insert the proper number of registers between this definition of a signal and its later use.

As code is written to the `vhdl` stream, a variable *currentCycle* is updated thanks to the `manageCriticalPath()` calls in Listing 1. Their detailed behaviour will be explained below, for now, one just has to understand that `manageCriticalPath()` sometimes increases *currentCycle*, and sometimes does not: the pipeline information is built dynamically in function of frequency, target FPGA, etc.

This *currentCycle* variable is used for two things:

- it defines the `cycle` of signals appearing on the left-hand side of <=;
- it is compared to the `cycle` of any signal appearing on the right-hand side of <=: the difference is the number of registers that should be inserted between the declaration of the signal and its use.

For an example, consider again Listing 1, and assume the `manageCriticalPath()` of line 27 has increased *currentCycle*. Arriving line 30, we have an `EA` that is used on the right-hand side one cycle after its declaration line 19. To use the synchronized version, `outputVHDL()` simply replaces the right-hand side `EA` of line 30 with `EA_d1` in the generated code (see line 15 in Listing 2). Here the `1` in `EA_d1` is computed as *currentCycle*-`cycle(EA)`. Each signal also has a `lifeSpan` attribute which holds its maximum delay, and will be used to create, in the generated VHDL code, the needed number of new signals (here `EA_d1` to `EA_d6`) with registers between them.

Similar techniques enable managing sub-components, such as the shifters or squarers on Fig. 1. Effective inputs are managed as right-hand side signals, and effective outputs as left-hand side signal declarations: their `cycle` is defined as *currentCycle*, plus the sub-component's pipeline depth.

This simple technique has many advantages:

- It is simple to implement, as it involves only comparisons and subtractions of integers.
- It clearly separates two very different issues: building a functional combinatorial datapath (on the left of Fig. 3), and pipelining it (on the right). From a combinatorial datapath, we are guaranteed to obtain a correctly synchronized pipeline with the same functionality, without touching any of the lines that define this datapath (the lines starting with `vhdl <<`).
- Its complexity is linear in the size of the generated code. Two passes are necessary: The first one writes the combinatorial VHDL code in the `vhdl` stream, and builds a dictionary of signals with their `cycle` and `lifeSpan`. The second one delays right-hand side signals.
- It adapts to arbitrary, dynamical placement of synchronization barriers, which is what we need for frequency-directed pipeline. It also gracefully degrades to a unpipelined, combinatorial implementation.
- The overhead of pipeline management in the generated code is minimal (some signal names posfixed by `_dxxx`), and this code remains as easy to read as the unpipelined version, especially compared to a pipeline of similar flexibility that would be written using VHDL `GENERATE` constructs.
- Finally, since this technique only involves post-processing signal names, it works for arbitrary VHDL.

For the designer, the construction of the pipeline boils down to managing the value of *currentCycle* at each point of its C++ code. Let us now study this.

Listing 1. Exponent difference and sorting in Fig. 1

```
1  // The expSort box
2  manageCriticalPath(  // evaluate the delay
3     target->adderDelay(wE+1)    // exp. diff.
4   + target->localWireDelay(wE) // wE is the fanout
5   + target->lutDelay()     );   // mux
6
7  // determine the max of the exponents
8  vhdl << declare("DEXY", wE+1) <<
9    " <=   ('0' & EX) - ('0' & EY);" << endl;
10 vhdl << declare("DEYZ", wE+1) <<
11   " <=   ('0' & EY) - ('0' & EZ);" << endl;
12 vhdl << declare("DEXZ", wE+1) <<
13   " <=   ('0' & EX) - ('0' & EZ);" << endl;
14 vhdl << declare("XltY") << "<= DEXY(wE);" << endl;
15 vhdl << declare("YltZ") << "<= DEYZ(wE);" << endl;
16 vhdl << declare("XltZ") << "<= DEXZ(wE);" << endl;
17
18 // rename  exponents  to A,B,C with A>=(B,C)
19 vhdl << declare("EA", wE)  << " <= "
20   << "EZ when (XltZ='1') and (YltZ='1')  else "
21   << "EY when (XltY='1') and (YltZ='0')  else "
22      "EX;"   << endl;
23 vhdl << declare("EB", wE)  << " <= " << (...);
24 vhdl << declare("EC", wE)  << " <= " << (...)
25
26 // the parallel subtractions
27 manageCriticalPath(  target->adderDelay(wE-1) );
28
29 vhdl << declare("shiftB", wE-1) <<
30   " <=  EA(wE-2 downto 0) - EB (wE-2 downto 0);";
31 vhdl << declare("shiftC", wE-1) <<
32   " <=  EA(wE-2 downto 0) - EC (wE-2 downto 0);";
```

Listing 2. VHDL generated by Listing 1 for wE

```
DEXY <=   ('0' & EX) - ('0' & EY);
DEYZ <=   ('0' & EY) - ('0' & EZ);
DEXZ <=   ('0' & EX) - ('0' & EZ);
XltY <=   DEXY(8);
YltZ <=   DEYZ(8);
XltZ <=   DEXZ(8);
EA <=
   EZ when (XltZ='1') and (YltZ='1')  else
   EY when (XltY='1') and (YltZ='0')  else
   EX;
EB <= (...)
EC <= (...)
--Synchro barrier, entering cycle 1--
shiftB <=
  EA_d1(6 downto 0) - EB_d1(6 downto 0) ;
shiftC <=
  EA_d1(6 downto 0) - EC_d1(6 downto 0) ;
```

C. Cycle management and synchronization

We want to pipeline our datapath for a given frequency f. When done by hand, this task consists in identifying the critical path of the combinatorial circuit, then inserting enough synchronization barriers to split it into sub-paths, each of delay smaller than $1/f$.

In FloPoCo, code generation progresses from input to output, so the idea is to maintain an estimation of the current critical path delay, and insert synchronization barriers when needed. This is what `manageCriticalPath()` does. This function takes as argument an estimation of the critical path delay of the logic generated by the C++ code that follows it (up to the next `manageCriticalPath()`). It adds this argument to a variable *currentCriticalPath*, and if the resulting delay is larger than $1/f$, it inserts a synchronization barrier: it increments *currentCycle*, and resets the critical path delay to its argument.

In Listing 1 we have defined two atomic blocks that correspond respectively, on Fig. 1, to the expSort box (lines 7 to 24), and to the two parallel subtraction boxes (lines 29 to 32). Depending on the target frequency, the code of Listing 1 will fuse these two blocks in a single cycle, or will insert a synchronization barrier between them.

The designer has the freedom to chose the granularity of these atomic boxes. This is a matter of expertise. Here, for instance, we know that we are subtracting exponents, which will therefore remain relatively small (even the 128-bit quadruple precision format has only `wE=15` exponent bits), so it makes sense to consider the expSort box as atomic.

Many other high-level functions help a designer managing *currentCycle*. For instance, synchronization of several paths (as is needed at the input of the normalize/pack box of Fig. 1) means advancing *currentCycle* to the max of the `cycle`s of the signals to be synchronized.

D. The Target class hierarchy

The delays passed to `manageCriticalPath()` are evaluated thanks to methods of a `target` object. This object holds the current target FPGA (which can be specified by the `-target` option of the FloPoCo command line). Thus for instance, `adderDelay(16)` will return different values for a Spartan3 or a Virtex5, and eventually the pipeline will be deeper for a slower FPGA.

As seen on Fig. 2, a Target object provides methods for delay estimation (including routing), but also methods that can be used for architecture tuning. For instance, Chapman's constant multiplication algorithm is based on FPGA look-up tables [12]. Its generic FloPoCo implementation queries `lutSize()` for the input size of LUTs in the target FPGA. Other methods describe the capabilities of DSP blocks and embedded memories.

Modeling FPGAs is an endless effort, all the more as new models appear each year, but the reliance on the virtual Target class ensures that FloPoCo datapaths are designed in a reasonably future-proof way.

E. Towards optimal pipelines

The general philosophy of FloPoCo's approach to pipelining is "best effort". While it gets the pipeline almost right for small operators in empty FPGAs, the actual performance in a real application may depend on subsequent optimizations by the synthesizer, and on unpredictable effects such as placement within a larger design, and routing congestion. These effects are out of control of the datapath designer, so it is impossible to guarantee that the final circuit will run at the desired frequency. However, targeting a higher frequency will improve the actual frequency, and targeting a lower frequency will save resources. This is enough for design exploration.

FloPoCo pipelines should also be a good starting point for automatic retiming algorithms introduced by Leiserson and Saxe [3], [4], which are slowly being integrated in synthesis tools (after the technology mapping and related optimizations, but before place and route). As these algorithms work by local modifications of the circuit, they will converge much faster and avoid being trapped in local extrema if they start with a good approximation of the global optimal.

To sum up, we view Leiserson and Saxe retiming as a back-end to FloPoCo, and we view FloPoCo as a back-end to global application-level retiming approaches such as Perry's [2]. The abstraction level offered by FloPoCo (cycles and approximate critical path) is just right for this context.

IV. Arithmetic-based testing

The underlying mathematical nature of an arithmetic datapath can be usefully exploited towards test-bench generation: we have a reference function ($x^2 + y^2 + z^2$ for the sum of squares), and the datapath should behave as the composition of this function with some rounding to the target format.

A. *Specification-based testing*

In FloPoCo, one may define what a datapath is supposed to compute by overloading the virtual method `emulate()`. Thanks to the bit-accurate MPFR library[2], this typically takes about ten lines (due to lack of space we refer again to the FloPoCo distribution for actual examples).

For any FloPoCo `Operator` with an `emulate()` method, the `TestBench` operator tests its generated VHDL against its expected behaviour. Millions of test vectors may be generated automatically in seconds, and this high-level approach minimizes the possibility of making the same mistake in both the operator and its testbench.

B. *Function-specific random testing*

`TestBench` may generate exhaustive tests when it is practical. Otherwise, it has to resort to testing on random inputs. In this case, it is often desirable to use a function-specific random test-case generator. Let us just take two examples.

- The exponential function e^x very quickly over- and underflows. For instance, in double precision it overflows for $x > 710$ and underflows to 0 for $x < -746$. If we test it on random inputs in the full floating-point range ($-1.8 \cdot 10^{308} < x < 1.8 \cdot 10^{308}$) we will statistically mostly test the under/overflow logic. What is needed here is a random generator that is biased towards the useful interval $-746 < x < 710$. We also want to bias it against all the small inputs ($|x| < 2^{-55}$) in double precision) for which the exponential returns 1.0.
- In a floating-point adder, if the difference between the exponents of the two operands is larger than the significand size, the adder will simply return the biggest of the two, and again this is the most probable situation when taking two random operands. Here we must bias the random generator towards cases where the two operands have close exponents.

Such cases are managed by overloading the `Operator` method `buildRandomTestCases()`. In addition, `buildStandardTestCases()` specifically tests corner cases which even focused random testing has little chance to find.

[2]www.mpfr.org

V. Conclusion and future work

The FloPoCo framework improves the productivity of designing flexible and efficient arithmetic datapaths for FPGAs. Its main features are a state-of-the-art arithmetic operator library, a novel methodology for the generation of correct-by-construction pipelines matching a given frequency on a given FPGA target, and arithmetic-oriented test-bench generation.

The most complex operator currently in FloPoCo is the floating-point exponential operator detailed in [8]. It makes use of shifter and adders, but also constant multipliers, truncated multipliers, table generators and polynomial evaluators available only in FloPoCo, and a lot of glue logic. FloPoCo enabled us to manage this complexity, to provide an operator that is parameterized in exponent and significand size, last-bit accurate for all its input, automatically optimized for a range of Altera and Xilinx targets, and pipelined to frequencies close to the maximum practical on these FPGA.

We plan to use FloPoCo for ever coarser datapaths such as signal-processing filters. We also hope it will be used as a back-end for high-level synthesis tools. Both the library and the framework will be developed to address the needs of these application fields. Potential future work also includes adding to the framework resource estimation, floorplaning support, fixed-point support, support of sequential circuits, and ASIC targets.

References

[1] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 6, no. 4, pp. 18–24, 2009.

[2] S. Perry, "Model based design needs high level synthesis: a collection of high level synthesis techniques to improve productivity and quality of results for model based electronic design," in *Design, Automation and Test in Europe*. EDAA, 2009, pp. 1202–1207.

[3] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5 – 35, 1991.

[4] K. N. Lalgudi and M. C. Papaefthymiou, "DELAY: an efficient tool for retiming with realistic delay modeling," in *ACM/IEEE Design Automation Conference*, 1995, pp. 304–309.

[5] K. Eguro and S. Hauck, "Simultaneous retiming and placement for pipelined netlists," in *Field-Programmable Custom Computing Machines*. IEEE, 2008, pp. 139–148.

[6] F. de Dinechin, H. D. Nguyen, and B. Pasca, "Pipelined FPGA adders," in *Field Programmable Logic and Applications*. IEEE, 2010.

[7] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for floating-point double precision and beyond on FPGAs," in *Higly-Efficient Accelerators and Reconfigurable Technologies*. ACM, 2010.

[8] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Field-Programmable Technologies*. IEEE, 2010.

[9] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2010.

[10] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for FPGAs," in *Field-Programmable Custom Computing Machines*. IEEE, 2003.

[11] M. Langhammer and T. VanCourt, "FPGA floating point datapath compiler," in *Field-Programmable Custom Computing Machines*. IEEE, 2009, pp. 259–262.

[12] K. Chapman, "Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner)," *EDN magazine*, May 1994.