Designing Custom Arithmetic Data Paths with FloPoCo

Florent de Dinechin and Bogdan Pasca
École Normale Supérieure de Lyon

Editor's note:
Efficient implementation of basic, data-path circuit elements is of fundamental importance to achieving high performance in FPGA-based acceleration of scientific computing. This work presents a leading effort to automate the production of pipelined data-path circuits for implementing numerical functions.
—George A. Constantinides (Imperial College London) and Nicola Nicolici (McMaster University)

FPGAs ARE INCREASINGLY being considered as application accelerators. They are especially relevant for applications that expose parallelism and require arithmetic operations that are not well-supported in hardware by mainstream processors. Examples include novel cryptography algorithms, Monte Carlo simulations requiring massive amounts of random numbers, digital-signal processing (DSP), and many others.

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

In this article, we address the design of parameterized, pipelined arithmetic data paths for FPGAs. An arithmetic data path is the implementation of some function; here, we use the word *function* in its mathematical sense: $f(X) = Y$, where $X = x_0, \ldots, x_{i-1}$ is a set of inputs and $Y = y_0, \ldots, y_{j-1}$ is a set of outputs. Examples of such functions include basic operations, elementary functions such as sine or exponential, complex multiplication $[x + iy = (a + ib)(c + id)]$, and even Fourier transforms.

This mathematical function defines a reference for each value computed by a data path implementing it. For functions over integers or finite fields, the data path should return the exact same value as the mathematical function. For functions over real numbers, the data path must provide an approximation. The accuracy of this approximation is constrained by the formats chosen for the inputs and outputs, typically some fixed-point or floating-point format. The "Data Efficiency for Numerical Data Paths" sidebar explains why accurate data paths on minimal formats are key to efficient FPGA implementations.

FloPoCo project overview

FloPoCo (Floating-Point Cores) is an open-source C++ framework for generating arithmetic data paths; it can be downloaded from http://flopoco.gforge.inria.fr. FloPoCo provides a command-line interface that inputs operator specifications, and outputs synthesizable VHDL.

Each data-path generator in FloPoCo is a C++ class. At the lowest level, the task of such a class involves 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.

Assisted pipeline design

Arithmetic data paths, as we've defined them, can be implemented as combinatorial functions. One way to exploit the inherent parallelism of FPGAs for better performance is to pipeline these combinatorial implementations. Pipelining is conceptually easy, but in practice is tedious and error prone. The FloPoCo framework lets programmers focus on the high-level aspects of this task, and

Data Efficiency for Numerical Data Paths

For FPGAs, an often overlooked measure of efficiency is whether each of the bits carried along an application holds useful information.

For example, consider an application that can be content with an exponential accurate to 0.4%, or 2^{-8}. In software, the smallest floating-point format available is single precision, a 32-bit format with a 23-bit significand. The default single-precision exponential offers a relative accuracy of 2^{-24}, matching this format. This is much too accurate for this application; therefore, it makes perfect sense here to design a faster single-precision exponential, accurate to 2^{-8} only. Half of the 32 bits of the single-precision result will contain noise instead of useful information, but removing them from the processor data path is not an option.

In an FPGA, however, removing output bits is an option, because data formats are far more flexible. Using a single-precision exponential core accurate to 2^{-8} means that, out of the 32 bits passed along the data path, 16 bits hold useless noise. Obviously, this entails a waste of precious routing resources and registers. Besides, the subsequent operators in the data path will compute on this noise, meaning more wasted resources and needless power consumption.

Therefore, 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 an 8-bit significand.

However, designing optimally data-efficient data paths—that is, defining how much accuracy is required at each point of a data path (possibly including guard bits to absorb rounding errors)—could take considerable design effort. FloPoCo helps tackle this challenge in two ways. First, it provides an expanding library of coarse operators, which are state of the art in terms of data efficiency. Second, all its operators are parameterized by the precision and are last-bit accurate, thus enabling design-space exploration that includes precision tuning.

automates the rest. The main features of pipeline generation in FloPoCo are frequency-directed pipelining, target-specific pipeline tuning, and close designer control.

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

Target-specific pipeline tuning. The frequency of a given design strongly depends on the target FPGA. Therefore, frequency-directed pipelining must be based on an abstract model of the FPGA's capabilities, including timing information. FloPoCo comes with such models for main FPGA families from both Xilinx and Altera.

Pipeline under close designer control. Ideally, a designer would write only the combinatorial version of an operator (focusing on its functionality) and let the tools pipeline this design for a given target FPGA and frequency[2]—for instance, by using retiming.[3-5] However, the frequency might dictate fundamental changes in the architecture, such as imposing fast adders[6] or tables of precomputed values. FloPoCo lets a designer program such architectural choices, which entails also programming the construction of the pipeline. However, the framework makes this task easy and safe through high-level notions such as cycles, synchronization, and critical path.

FloPoCo operator library

FloPoCo provides an ever-increasing library of arithmetic cores, each parameterized in size and following the frequency-directed pipeline paradigm. This library includes integer, fixed-point, and floating-point basic operators, sometimes in several variants (e.g., large pipelined adders,[6] and 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, with more to come. Finally, the FloPoCo library also includes meta-operators:

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

The project website provides a full list, as well as pointers to research articles describing most of these components.

A motivating example

Consider a floating-point sum of squares: this data path inputs three floating-point numbers (X, Y, and Z), and outputs a floating-point number for $X^2 + Y^2 + Z^2$.

FloPoCo command line

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

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

will generate synthesizable VHDL for a floating-point adder pipelined to run at 200 MHz on a Xilinx Virtex-4, using a custom floating-point format with 10 bits for the exponent and 36 bits for the significand; this format is intermediate between standard single and double precisions.

For design exploration, the four parameters in this example (target FPGA, frequency, exponent size, and significand size) can be changed within sensible range. The frequency and precision are orthogonal parameters, as they should be. The pipeline depth is computed and reported to the user when this command is run.

More complex data paths 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 data path.

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 that we are used to seeing 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 because they don’t occur often enough in general code. We’ve already mentioned squarers: writing the pseudo-program as

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

lets us obtain a data path that consumes less resources and has a slightly shorter latency.

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

- recovers the intrinsic parallelism and symmetry of this expression;
- fuses the data paths of the two additions;
- disposes of the logic that, in standard floating-point adders, manages the addition of numbers of different signs;[10]
- avoids intermediate roundings and normalizations, as in Langhammer and VanCourt’s floating-point compiler;[11] and
- returns a last-bit accurate result.

Figure 1 presents this data path. Compared to the one obtained by Langhammer and VanCourt,[11] it is better specified and embeds more optimizations, but it took several days to write.

Tables 1 through 3 compare these three approaches and illustrate the versatility of FloPoCo.

FloPoCo VHDL generation framework

Here, we describe the framework that enables the construction of such versatile operators. We assume the reader has a basic knowledge of object-oriented concepts with the C++ terminology. Figure 2 provides a simplified overview of the FloPoCo class hierarchy.

Operators and VHDL generation

The core class is `Operator`: every data path 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 subcomponents), and then generates the VHDL for them.

Figure 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 information. Then, the `outputVHDL()` method combines the `vhdl` stream and the pipeline information to form the VHDL code of the pipelined data path. It also declares all the needed VHDL signals, entities, components, and so forth; that way, the designer only needs to focus on the architectural part of the VHDL code.

Figure 4 shows an example of basic FloPoCo code, and Figure 5 gives the corresponding generated VHDL code. These figures describe the upper left part of Figure 1.

Pipelining basics

A pipeline of depth n consists of $n + 1$ pipeline stages (numbered from 0 to 6 on the right of Figure 1), separated by synchronization barriers (dashed lines). In essence, pipelining involves associating, with every signal, the number of the cycle in which it's defined (its `cycle` attribute in FloPoCo), then using this information to insert the proper number of registers between this signal definition and any of its later uses in the architecture.

As code is written to the `vhdl` stream, a variable `currentCycle` is updated, thanks to the `manageCriticalPath()` calls in Figure 4. Sometimes, `manageCriticalPath()` increases `currentCycle`, and sometimes

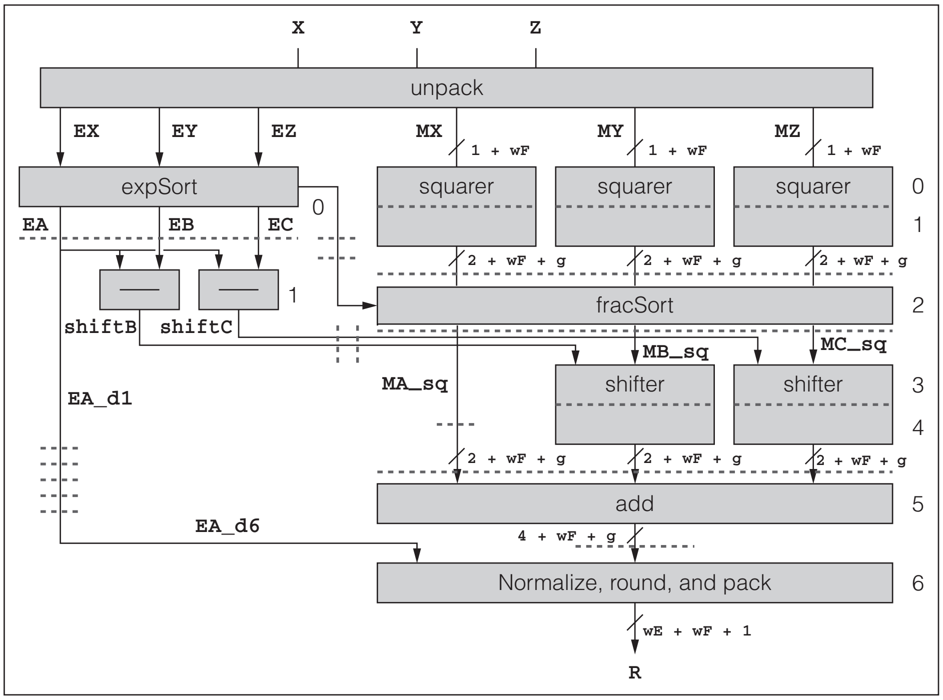

Figure 1. A fused data path pipeline for the floating-point sum of squares. `EX`, `EY`, and `EZ` are the exponents (of width `wE` bits) of the three inputs; `MX`, `MY`, and `MZ` are their significands (of width 1 + `wF` bits). `EA`, `EB`, and `EC` represent `EX`, `EY`, and `EZ` after sorting, and `MA_sq`, `MB_sq`, and `MC_sq` represent the squares of `MX`, `MY`, `MZ`, also sorted according to `EX`, `EY`, and `EZ`. Internal precision is extended with *g* guard bits, and *g* = 3 ensures last-bit accuracy of the result—a detailed error analysis proving this is given in the source code. The dashed lines are synchronization barriers for one example of the pipeline.

Table 1. Productivity versus performance for $x^2 + y^2 + z^2$ on a Virtex-4 (xc4vfx100-12), with a target frequency of $f = 350$ MHz.

Format*	Approach** (development time)**	Performance***	Cost***
(8, 23)	Xilinx LogiCore (1 hour)	34 cycles, 482 MHz	1,356 slices, 12 DSPs
	Option 1 (1 minute)	35 cycles, 327 MHz	1,279 slices, 12 DSPs
	Option 2 (1 minute)	35 cycles, 333 MHz	1,043 slices, 9 DSPs
	Option 3 (3 days)	11 cycles, 369 MHz	470 slices, 9 DSPs
(11, 52)	Xilinx LogiCore (1 hour)	50 cycles, 354 MHz	3,074 slices, 48 DSPs
	Option 1 (1 minute)	47 cycles, 319 MHz	3,859 slices, 48 DSPs
	Option 2 (1 minute)	45 cycles, 322 MHz	3,137 slices, 18 DSPs
	Option 3 (3 days)	16 cycles, 368 MHz	1,866 slices, 18 DSPs

* Format is given as (exponent width, significand width).
** Xilinx LogiCore operators, assembled by hand, are provided as a reference. Option 1 is FPPipeline, using multipliers. Option 2 is FPPipeline, using squarers. Option 3 is the fused data path of Figure 1.
*** All Xilinx numbers are postsynthesis results using ISE 11.5.

Table 2. Performance versus cost (resource consumption) for $x^2 + y^2 + z^2$ on a Virtex-4, option 3, with a varying target frequency.

Format	Target frequency (MHz)	Performance	Cost
(10, 36)	200	6 cycles, 203 MHz	874 slices, 9 DSPs
	100	2 cycles, 109 MHz	809 slices, 9 DSPs
	50	0 cycles, 51 MHz	751 slices, 9 DSPs
(11, 52)	200	7 cycles, 187 MHz	1,285 slices, 18 DSPs
	100	3 cycles, 102 MHz	1,272 slices, 18 DSPs
	50	2 cycles, 64 MHz	1,130 slices, 18 DSPs

Table 3. Portability to different FPGAs, option 3, with a target frequency of f = 200 MHz, for $x^2 + y^2 + z^2$.

Format	FPGA*	Performance	Cost**
(10, 36)	Virtex-5 (xc5vfx100T-3)	5 cycles, 196 MHz	1,444 L, 762 R, 9 DSP48E
	Stratix II (EP2S15F484C3)	8 cycles, 179 MHz	1,395 L, 1,295 R, 18 9-bit multipliers
	Stratix IV (EP4S40G2F40I1)	4 cycles, 213 MHz	1,529 L, 792 R, 18 18-bit multipliers***

* The Altera Stratix II and Stratix IV numbers are post-place-and-route results in empty FPGAs using Quartus II 9.1.
** L: look-up tables; R: registers.
*** Only 9 are actually used; 9 are lost due to DSP-block I/O constraints.

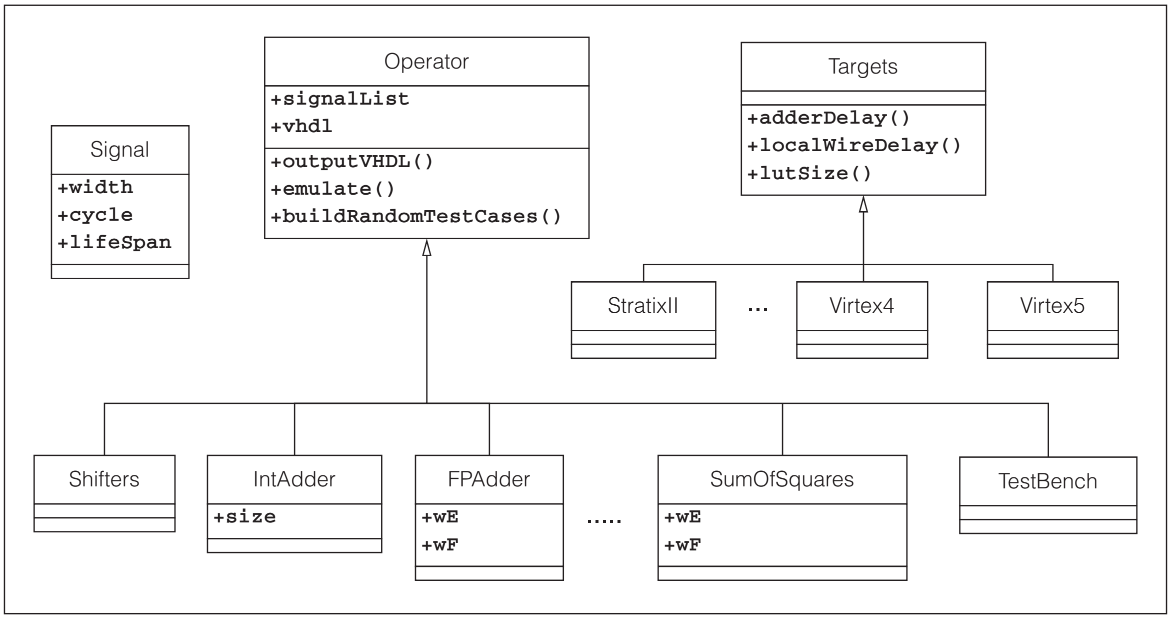

Figure 2. Simplified overview of the FloPoCo class hierarchy.

it does not: the pipeline information is built dynamically, depending on frequency, target FPGA, and so on.

This `currentCycle` variable serves two purposes:

- 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 <=, and the difference is the number of registers that should be inserted between the declaration of the signal and its use.

For example, consider again Figure 4, and assume the `manageCriticalPath()` of line 27 has

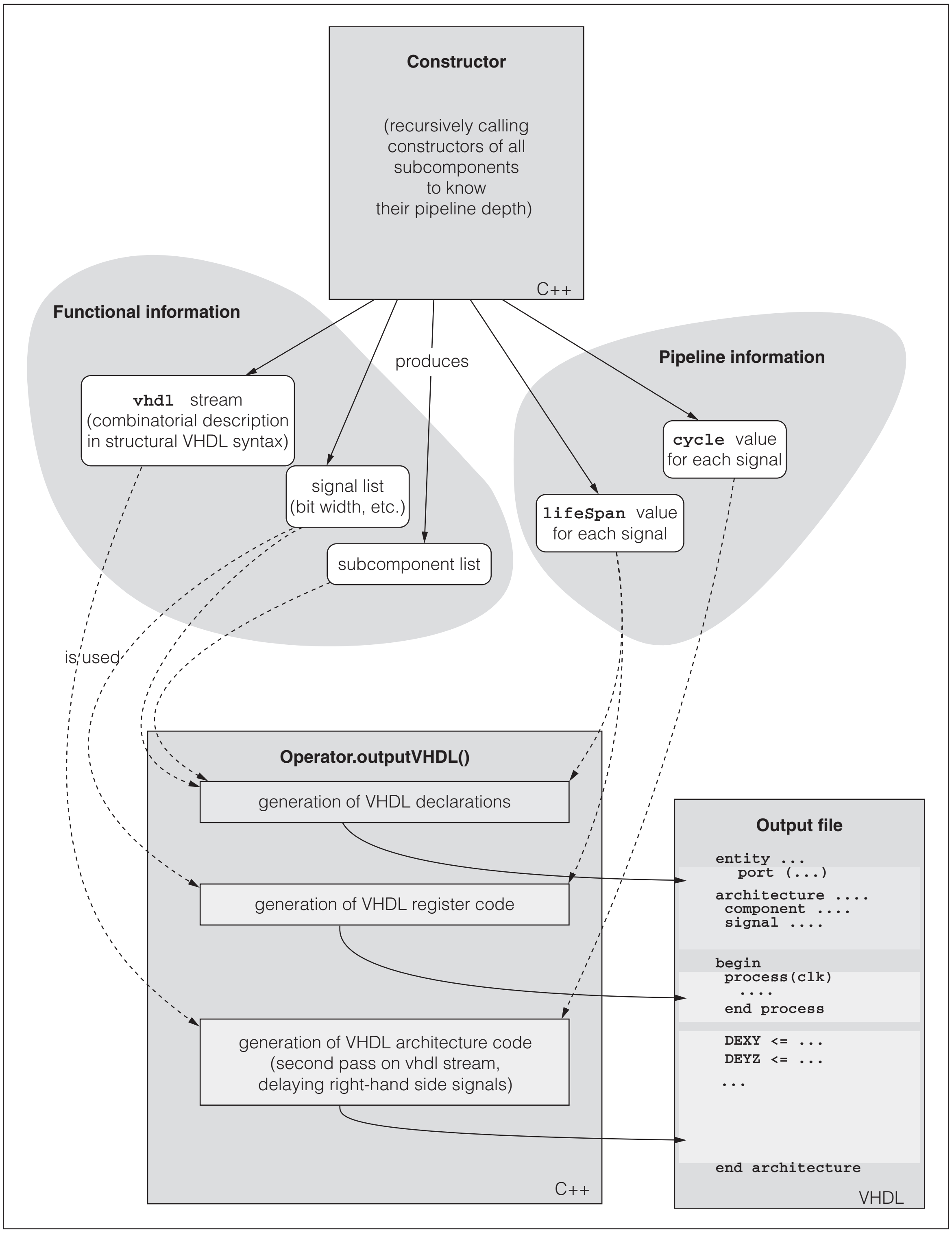
Constructor
(recursively calling constructors of all subcomponents to know their pipeline depth)
C++
Functional information
produces
Pipeline information
vhdl stream (combinatorial description in structural VHDL syntax)
signal list (bit width, etc.)
subcomponent list
cycle value for each signal
lifeSpan 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

Figure 3. Simplified overview of the VHDL generation flow.

increased `currentCycle`. In line 30, signal `EA` is used on the right-hand side one cycle after its declaration in 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 Figure 5). Here, the 1 in `EA_d1` is computed as `currentCycle – cycle(EA)`. Each signal also has a `lifeSpan` attribute, which holds its

```
// The expSort box
manageCriticalPath( // evaluate the delay
    target->adderDelay(wE+1) // exp. diff.
  + target->localWireDelay(wE) // wE is the fanout
  + target->lutDelay() ); // mux

// determine the max of the exponents
vhdl ≪ declare("DEXY", wE+1) ≪
  " <= ('0' & EX) - ('0' & EY);" ≪ endl;
vhdl ≪ declare("DEYZ", wE+1) ≪
  " <= ('0' & EY) - ('0' & EZ);" ≪ endl;
vhdl ≪ declare("DEXZ", wE+1) ≪
  " <= ('0' & EX) - ('0' & EZ);" ≪ endl;
vhdl ≪ declare("XltY") ≪ "<= DEXY(wE);" ≪ endl;
vhdl ≪ declare("YltZ") ≪ "<= DEYZ(wE);" ≪ endl;
vhdl ≪ declare("XltZ") ≪ "<= DEXZ(wE);" ≪ endl;

// rename exponents to A,B,C with A>=(B,C)
vhdl ≪ declare("EA", wE) ≪ " <= "
  ≪ "EZ when (XltZ='1') and (YltZ='1') else "
  ≪ "EY when (XltY='1') and (YltZ='0') else "
  ≪ "EX;" ≪ endl;
vhdl ≪ declare("EB", wE) ≪ " <= " ≪ (...);
vhdl ≪ declare("EC", wE) ≪ " <= " ≪ (...)

// the parallel subtractions
manageCriticalPath( target->adderDelay(wE-1) );

vhdl ≪ declare("shiftB", wE-1) ≪
  " <= EA(wE-2 downto 0) - EB (wE-2 downto 0);";
vhdl ≪ declare("shiftC", wE-1) ≪
  " <= EA(wE-2 downto 0) - EC (wE-2 downto 0);";
```

Figure 4. Basic FloPoCo Code for exponent difference and sorting in Figure 1.

```
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);
```

Figure 5. VHDL code generated by the FloPoCo code given in Figure 4 for $w_E = 8$.

maximum delay and will be used to create, in the generated VHDL code, the required number of new signals (here, `EA_d1` to `EA_d6`) with registers between them.

Similar techniques enable managing subcomponents, such as the shifters and squarers in Figure 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 subcomponent's pipeline depth.

This technique has many advantages:

- It is simple to implement because it involves only comparisons and subtractions of integers.
- It clearly separates two very different issues: building a functional combinatorial data path (on the left of Figure 3) and pipelining it (on the right of Figure 3). From a combinatorial data path, we are guaranteed to obtain a correctly synchronized pipeline with the same functionality, without touching any of the lines that define this data path (the lines starting with `vhdl ≪` in Figure 4).
- Its complexity is linear in the size of the generated code. Two passes are necessary. The first pass writes the combinatorial VHDL code in the `vhdl` stream, and builds a dictionary of signals with their cycle and `lifeSpan`. The second pass delays the right-hand-side signals.
- It adapts to arbitrary, dynamical placement of synchronization barriers, which we need for frequency-directed pipelines. It also gracefully degrades to an unpipelined, combinatorial implementation.
- The overhead of pipeline management in the generated code is minimal (with some signal names postfixed 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, because this technique involves only post-processing signal names, it works for arbitrary VHDL.

For the designer, the construction of the pipeline resumes managing the value of `currentCycle` at each point of its C++ code, which we address next.

Cycle management and synchronization

We'd like to pipeline our data path for a given frequency *f*. When done by hand, this task involves identifying the critical path of the combinatorial circuit, then inserting enough synchronization barriers to split it into subpaths, each of a 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 then adds this argument to a variable, `currentCriticalPath`, and if the resulting delay is larger than 1/*f*, inserts a synchronization barrier: it increments `currentCycle`, and resets the critical-path delay to its argument.

For instance, Figure 4 defines two atomic blocks that correspond, respectively, in Figure 1, to the `expSort` box (lines 7 to 24 in Figure 4), and the two parallel subtraction boxes (lines 29 to 32 in Figure 4). Depending on the target frequency, the code of Figure 4 will fuse these two blocks in a single cycle, or will insert a synchronization barrier between them.

The designer can choose 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 manage `currentCycle`. For instance, synchronization of several paths (as is needed at the input of the normalize-and-pack box of Figure 1) means advancing `currentCycle` to the max of the `cycle` attributes of the signals to be synchronized.

Target class hierarchy

The delays passed to `manageCriticalPath()` are evaluated via methods of a `target` object, describing the current target FPGA as specified by the `-target` option of the FloPoCo command line. Thus, for example, `adderDelay(16)` will return different values for a Spartan-3 or a Virtex-5, and eventually the pipeline will be deeper for a slower FPGA.

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

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

Toward optimal pipelines

The general philosophy of FloPoCo's approach to pipelining is "best effort." Although it gets the pipeline almost right for small operators in empty FPGAs, the actual performance in a real application might depend on subsequent optimizations by the synthesizer and unpredictable effects, such as placement within a larger design and routing congestion. These effects are out of the data-path designer's control, so it's 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 the automatic retiming algorithms introduced by Leiserson and Saxe,[3,4] which are slowly being integrated into synthesis tools (after the technology mapping and related optimizations, but before place and route). Because these algorithms work by making local modifications to the circuit, they will converge much faster and avoid being trapped in local extrema if they start with a good approximation of the global optimum.

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

Arithmetic-based testing

The underlying mathematical nature of an arithmetic data path can be usefully exploited toward test-bench generation. The reference function, composed with some rounding to the target format, specifies the expected behavior of the data path.

Specification-based testing

In FloPoCo, a designer can define what an `Operator` is supposed to compute by overloading its virtual `emulate()` method. Thanks to the bit-accurate MPFR (multiple-precision floating-point correctly rounded) library (http://www.mpfr.org), this typically takes about 10 lines. (Because of space limitations, we refer to the FloPoCo source code for actual examples.)

For any FloPoCo `Operator` with an `emulate()` method, the `TestBench` operator will then test its generated VHDL against its expected behavior. Millions of test vectors can 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.

Function-specific random testing

`TestBench` can generate exhaustive tests when practical. Otherwise, it must resort to testing on random inputs. In this case, it's often desirable to use a function-specific random test-case generator. Let's just consider two examples.

The exponential function e^x very quickly overflows 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 ($-1.8 \times 10^{308} < x < 1.8 \times 10^{308}$), we will statistically test mainly the underflow and overflow logic. What is needed here is a random generator that is biased toward 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 largest of the two, and again this is the most probable situation when taking two random operands. Here, we must bias the random generator toward cases in which 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 of finding.

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

We plan to use FloPoCo for ever-coarser data paths, such as signal-processing filters. We also hope it can be used as a back end for high-level synthesis tools. We will develop both the library and framework to address the needs of these application fields. Potential future work also includes enhancing the framework with floorplanning 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,* vol. 26, no. 4, 2009, pp. 18-24.
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," *Proc. Design, Automation and Test in Europe Conf.* (DATE 09), European Design and Automation Assoc., 2009, pp. 1202-1207.
3. C.E. Leiserson and J.B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica,* vol. 6, nos. 1-6, 1991, pp. 5-35.
4. K.N. Lalgudi and M.C. Papaefthymiou, "DELAY: An Efficient Tool for Retiming with Realistic Delay Modeling," *Proc. 32nd Design Automation Conf.,* ACM Press, 1995, pp. 304-309.
5. K. Eguro and S. Hauck, "Simultaneous Retiming and Placement for Pipelined Netlists," *Proc. 16th Int'l Symp. Field-Programmable Custom Computing Machines* (FCCM 08), IEEE CS Press, 2008, pp. 139-148.
6. F. de Dinechin, H.D. Nguyen, and B. Pasca, "Pipelined FPGA Adders," *Proc. Int'l Conf. Field Programmable Logic and Applications* (FPL 10), IEEE Press, 2010, pp. 422-427.
7. S. Banescu et al., "Multipliers for Floating-Point Double Precision and Beyond on FPGAs," *ACM SIGARCH*

Computer Architecture News, vol. 38, no. 4, 2010, pp. 73-79.

8. F. de Dinechin and B. Pasca, “Floating-Point Exponential Functions for DSP-Enabled FPGAs,” *Proc. Int’l Conf. Field-Programmable Technology* (FPT 10), IEEE Press, 2010, pp. 110-117.
9. F. de Dinechin, M. Joldes, and B. Pasca, “Automatic Generation of Polynomial-Based Hardware Architectures for Function Evaluation,” *Proc. 21st IEEE Int’l Conf. Application-specific Systems, Architectures and Processors* (ASAP 10), IEEE Press, 2010, pp. 216-222.
10. J. Liang, R. Tessier, and O. Mencer, “Floating Point Unit Generation and Evaluation for FPGAs,” *Proc. 11th Ann. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 03), IEEE CS Press, 2003, pp. 185-194.
11. M. Langhammer and T. Van Court, “FPGA Floating Point Datapath Compiler,” *Proc. 17th IEEE Symp. Field Programmable Custom Computing Machines* (FCCM 09), IEEE CS Press, 2009, pp. 259-262.
12. K.D. Chapman, “Fast Integer Multipliers Fit in FPGAs,” *EDN Magazine,* 12 May 1994; http://www.edn.com/archives/1994/051294/10di2.htm.

Florent de Dinechin is an assistant professor at École Normale Supérieure de Lyon. His research interests include software and hardware computer arithmetic, elementary-function evaluation, and FPGA computing. He has a PhD in computer science from Université of Rennes-1.

Bogdan Pasca is pursuing a PhD in the Department of Computer Science at École Normale Supérieure de Lyon. His research interests include FPGA computing and hardware computer arithmetic. He has an MSc in computer science from École Normale Supérieure de Lyon.

■ Direct questions and comments about this article to Florent de Dinechin and Bogdan Pasca, LIP (ENSL-CNRS-Inria-UCBL), École Normale Supérieure de Lyon, 46 allée d’Italie, 69364 Lyon Cedex 07, France; florent.de.dinechin@ens-lyon.org and bogdan.pasca@ens-lyon.org.