Automating the pipeline of arithmetic datapaths

Matei Istoan, Florent De Dinechin

Automating the pipeline of arithmetic datapaths

Matei Iştoan
Univ Lyon, Inria, INSA Lyon, CITI
F-69621 Villeurbanne, France

Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France

***Abstract*—This article presents the new framework for semi-automatic circuit pipelining that will be used in future releases of the FloPoCo generator. From a single description of an operator or datapath, optimized implementations are obtained automatically for a wide range of FPGA targets and a wide range of frequency/latency trade-offs. Compared to previous versions of FloPoCo, the level of abstraction has been raised, enabling easier development, shorter generator code, and better pipeline optimization. The proposed approach is also more flexible than fully automatic pipelining approaches based on retiming: In the proposed technique, the incremental construction of the pipeline along with the circuit graph enables architectural design decisions that depend on the pipeline.**

I. Introduction

The FloPoCo project is an open-source generator of arithmetic cores for FPGAs. Its main purpose is to study the opportunities of tailoring arithmetic components to their application context. This includes parameterizing the operators in size, and offering them in a range of cost/performance trade-offs (Figure 1). It also requires flexibility in the latency/frequency trade-off, which can be achieved by pipelining. FloPoCo pioneered frequency-directed pipelining: the parameter controlling the pipeline depth is a user-provided target frequency. This enables the construction of complex pipelined operators out of smaller ones, all designed to work at the same frequency. FloPoCo is also vendor-neutral, supporting a range of FPGAs from Altera and Xilinx. Although its main point is to offer original operators, it must also achieve performance on par with vendor-provided tools.

The contribution of this article is a framework that addresses these needs. It enables the construction of high-quality pipelines for a wide range of frequencies on a wide range of targets. For this, it requires very limited design overhead on top of the description of the combinatorial operator.

This work is motivated by a review of existing pipelining techniques in Section II. Section III details the proposed pipelining construction framework. Section IV presents some relevant design examples. Section V shows how the timing capabilities of the various supported FPGAs are modelled to enable pipeline optimization to a range of targets from a single code. Section VI evaluates this framework, and Section VII concludes.

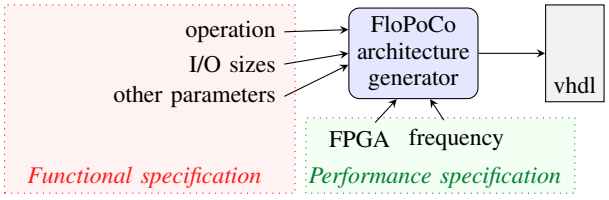

Fig. 1: Interface to FloPoCo operators

The case study we use throughout the paper is the floating-point adder. It is a well-understood benchmark, and it requires several sub-components (shifters, leading zero counters, several integer adders of various sizes) and a long pipeline with the need to synchronize many signals.

Space prevents providing all the details, but the interested reader is encouraged to obtain the source code from `gforge.inria.fr`. This framework was developed in the `newPipelineFramework` branch of the git repository, and will be used in FloPoCo 5.0 and following.

II. Background

Automatic circuit pipelining was formalized by Leiserson and Saxe [1], who introduced the notion of *retiming*: moving a register from the output of a gate to its inputs (or conversely) doesn't change the function computed by the circuit, but may change its performance. This technique is complex to apply in practice (because of initialization values, multiple clocks, etc), and it took many years before commercial tools like Synplify Pro offered it. Its use in mainstream FPGA design tools is currently still limited to pushing registers inside DSP blocks or memory.

It is possible to pipeline a design purely by retiming, but if the goal is to achieve a desired frequency, one first has to determine how many pipeline levels must be inserted. One option is to obtain (by synthesis) the critical path delay of the combinatorial operator. Then, a lower bound on the the needed number of pipeline levels is obtained by dividing the critical path delay by the target period.

The FloPoCo framework [2] automated this task. The typical design flow was to first write a generator of combinatorial operators, then enrich it with constructs that 1/ estimate the global critical path of a circuit, 2/ automatically insert pipeline registers accordingly, and 3/ semi-automatically synchronize the resulting signals [2]. However, in this approach, synchronization as well as critical path construction were assisted but not automated. This required the designer to explicitely manage a global view of the whole datapath.

Altera's solution, briefly described in [3], improved on this. Their method consists of first building a graph representing the circuit, annotated with local delay information. Then,

a scheduler aggregates this local information to determine critical paths and automatically inserts pipeline registers in the graph in a greedy fashion, while keeping the graph synchronized. The authors identify three main issues. The first is the management of *functional registers* such as those delaying signals in a digital filter. These are part of the function of the filter and should be preserved, although they can be retimed. In [3], they are ignored during the construction of the schedule, and only taken into account when generating the VHDL code describing the circuit. The second challenge is to manage loops in the circuit graph. Their approach is to break the loops during scheduling and reconnect them once the process is over. The third challenge is sub-components: they are pipelined separately.

Xilinx developed comparable tools at the same time, but contrary to Altera, they work with placed and routed circuits [4]. This offers the best precision, but has a high computational cost. More recently, Xilinx introduced a new tool (`report_pipeline_analysis`) for automatically determining the pipelining potential of a circuit [5]. It determines the maximum achievable frequency for the circuit, taking into account that circuit loops will limit it. This analysis tool currently only modifies the circuit in experimental versions of the Vivado design suite.

Matlab and Simulink [6] also have an approach based on retiming [1]. Their major contribution consists in removing the need to ensure the equivalence of the circuit states to the initial state. This allows them to considerably speed up the algorithm, while imposing certain limitations on the initial design.

The approach introduced in the present article inherits from FloPoCo's [2] and Altera's [3]. It works at the HDL level. It requires a designer to add local timing information to the combinatorial circuit. The aggregation of this local information for synchronization and timing construction is then fully automated. Functional registers are also supported.

An originality of the present work is to allow for incremental scheduling. This is more flexible than all the previous approaches which start with the full circuit graph. Computing the circuit pipelining during the construction of the architecture, and not afterwards, opens new opportunities: the construction of the circuit graph itself may be optimized by taking into account its partial pipeline. Examples of applications will be reviewed in Section IV. Of course, it is still possible to schedule (or re-schedule) a complete graph in order to benefit from optimal scheduling algorithms, e.g. based on integer linear programming.

Another originality of the present work, demonstrated in the sequel, is that it works for a range of FPGAs and a range of vendor tools.

Finally, the proposed approach doesn't forbid post-place-and-route retiming of the circuits it produces for optimal performance.

III. Pipeline construction in FloPoCo

An overview of the pipelining methodology is presented on Fig. 2 and detailed in Sections III-A, III-B and III-D.

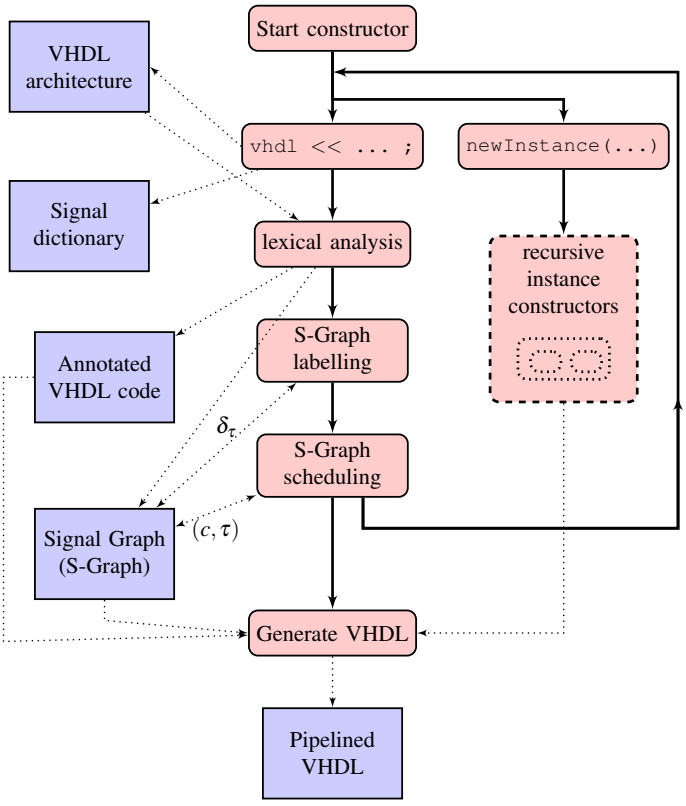

Fig. 2: Constructor flow overview

```
vhdl << declare("signX")  << "<= newX("<<wE+wF<<");";
vhdl << declare("signY")  << "<= newY("<<wE+wF<<");";
vhdl << declare(target->logicDelay(), "effSub")
<< "<= signX xor signY;"  ;
(...)
vhdl << declare(target->logicDelay(2), "excR", 2)
<< " (...) when effSub ='1' (...)"
(...)
vhdl<< declare(target->adderDelay(wE+1),"eXmeY", wE)
<< " <= (X" << range(wE+wF-1,wF)<<") - (Y"
<<range(wE+wF-1,wF)<<");";
```

Fig. 3: Example C++ constructor code

A. Design entry

A FloPoCo operator is created by the corresponding C++ constructor, as shown in the top part of Fig. 2. The architecture is built by either adding VHDL statements to the `vhdl` stream, or by instantiating sub-components (described

```
signX<= newX(31);
signY<= newY(31);
effSub <= signX_d1 xor signY_d1;
(...)
excR <= (...) when effSub_d2='1' (...);
(...)
eXmeY <= (X(30 downto 23)) - (Y(30 downto 23));
```

Fig. 4: VHDL code generated by the code of Fig. 3

in Section III-F). . Fig. 3 shows an example extracted from the architecture description of a parametric floating-point adder. The resulting pipelined VHDL code is presented in Fig. 4 (a signal name ending in '_d*xx*' represents a signal delayed by *xx* cycles).

Fig. 3 shows how certain design parameters (here exponent and mantissa sizes `wE` and `wF`) are held in C++ variables: this is far more convenient than using VHDL generics.

Signals are declared using the `declare()` method. Its main purpose is to add a signal to a signal dictionary. Its parameters are a signal name, and an optional signal bit width (e.g. signal `excR`). In addition, a first optional argument is a delay (in seconds) that estimates the delay contribution δ_τ of the right-hand side expression.

As Fig. 3 shows, this delay is typically captured by high-level methods of the Target class (here `target->logicDelay()` and `target->adderDelay()`. For instance, `target->adderDelay(wE+1)` returns an estimation of the delay of an addition of `wE+1` bits. These methods will be detailed and evaluated in Section V.

B. The signal graph

A lexical analyzer (middle of Fig. 2) extracts the data dependencies between VHDL signal names in the left-hand side and right-hand side of each statement of the `vhdl` stream. It builds a signal graph (S-Graph): its nodes are the signal names and its edges correspond to such data dependencies. Fig. 5, produced by the tool, shows the S-Graph for a single-precision floating-point adder, with a zoom on the part that corresponds to Fig. 4. For example, in Fig. 4, `effSub` depends on signals `signX` and `signY`. Thus, the signal graph contains edges from the nodes `signX` and `signY` to the node `effSub`.

Moreover, each signal node is labelled with its contribution to the critical path (which is the delay passed to `declare()`). This is the number on the second line of each box on Fig. 5.

For the sake of readability, Fig. 5 is a compact representation of the graph, where the graphs of the sub-components are omitted. FloPoCo can also output the full S-Graph with all sub-components flattened.

The lexer also annotates the architecture's VHDL code to facilitate further passes over the architecture which will replace signals with their registered version (those whose name ends with `_dxx` on Fig. 4).

The next step on Fig. 2 is to compute the scheduling of the S-graph so that it corresponds to a circuit pipelined for the target frequency. This consists in assigning to each signal the timing at which it will be computed by the circuit. Let us now describe how this timing is determined.

C. Lexicographic time

In a combinatorial circuit, the timing of a signal *s* can be computed by accumulating the delays on the longest simple path from the earliest input to *s* (critical path delay).

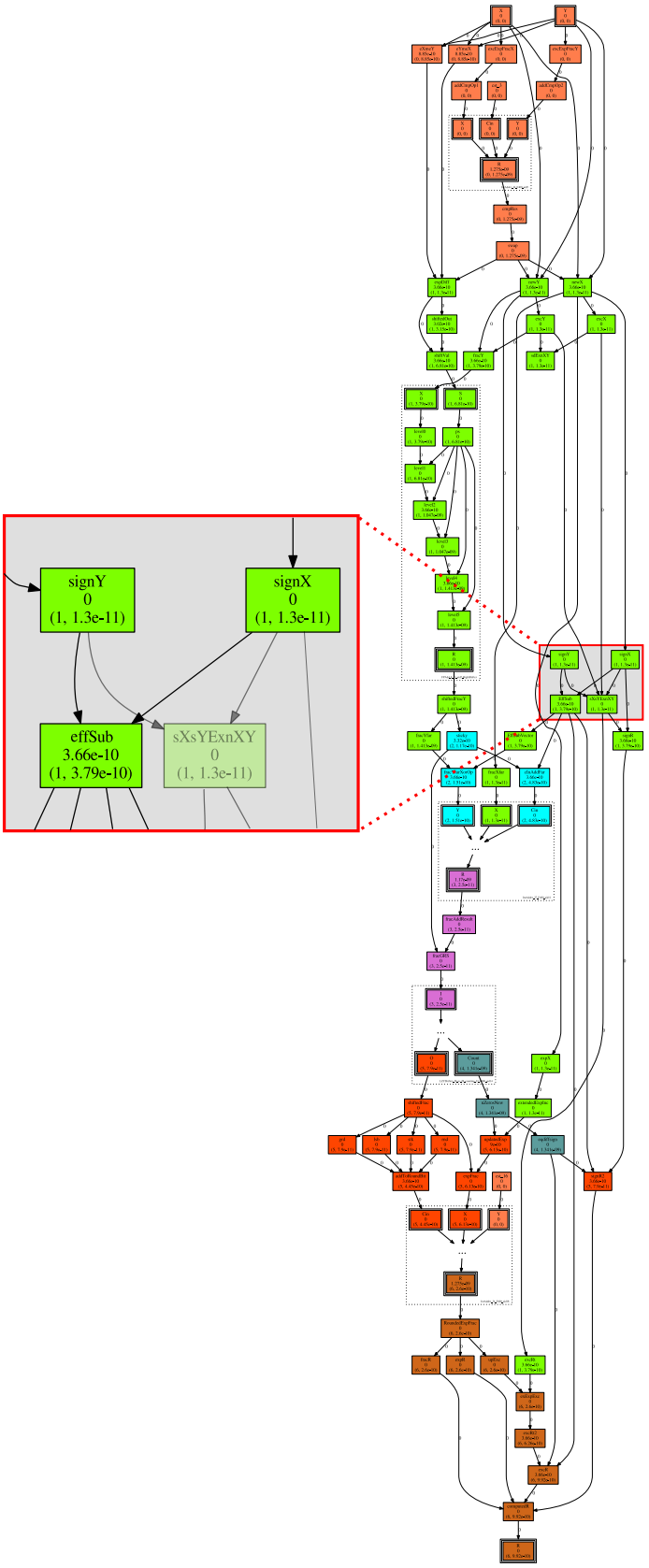

Fig. 5: S-Graph for a single-precision floating-point adder

Inside a pipelined circuit, however, a signal s may be delayed by a number of cycles c. The timing of s is then expressed as a pair (c, τ), where

- c is an integer that counts the number of registers on the longest path from an input to s.
- τ is a real number that represents the critical path delay (in seconds) from the last register or earliest input to s.

The colors on Fig. 5 indicate the cycle, and the complete lexicographic time of each signal is given by the third line of each signal box.

There is a lexicographic order on such timings: $(c_1, \tau_1) > (c_2, \tau_2)$ if $c_1 > c_2$ or if $c_1 = c_2$ and $\tau_1 > \tau_2$.

D. Incremental scheduling

The scheduling of a circuit is a well-studied problem, which also bears resemblance to many other similar computer science problems. In our case it consists in assigning to each signal of the S-graph a lexicographic time that respects causality along the dependencies of the S-Graph. There are several possible objective functions, essentially minimizing the output timings (i.e. optimizing for performance), or minimizing the number of registers (i.e. optimizing for resources). Our implementation currently focusses on the former.

The difficulty is that we consider an incremental scheduling problem, where signals arrive over time. The total number of signals is not known in advance. A signal's critical path contribution becomes known upon its insertion in the S-graph. However, it may happen that some of of the dependency information concerning a signal is still unknown upon its arrival.

The approach chosen is a greedy, as-soon-as-possible solution. Signals are scheduled as soon as their predecessors are scheduled. They are then assigned the earliest possible timing. This scheduling process is outlined in Algorithm 1.

Algorithm 1 S-Graph Scheduling

```
 1: while new VHDL instruction exists do
 2:     POPULATE_SIGNALS_TO_SCHEDULE( )
 3:     if new subcomponent instance then
 4:         SCHEDULE_INSTANCE( )
 5:     end if
 6:     for all signals_to_schedule do
 7:         SCHEDULE_SIGNAL(current_signal)
 8:     end for
 9: end while
10:
11: procedure SCHEDULE_SIGNAL(signal)
12:     pred ← GET_LATEST_PREDECESSOR(signal)
13:     timing ← GET_SIGNAL_TIMING(pred, δ_t(signal))
14:     SET_SIGNAL_TIMING(signal, timing)
15:     for all GET_SUCCESSORS(signal) do
16:         SCHEDULE_SIGNAL(current_successor)
17:     end for
18: end procedure
```

The `populate_signals_to_schedule()` function on line 2 selects from the signal dictionary the signals that have been affected by the latest `vhdl` stream operation. These are the signals that need to be scheduled, together with their direct and transitive successors. Note that a signal might be rescheduled. This is due to the incremental nature of the design process, and it ensures compatibility with VHDL's way of specifying concurrent instructions.

The calls to `schedule_signal()` on line 7 recursively trigger the scheduling process. It starts with the signals selected on line 2 and propagates to their dependences, as per the S-Graph, and depicted in lines 15-17. Lines 12-14 describe how a signal's timing is chosen: it is the smallest (lexicographically) pair (c, τ) which satisfies the following constraints:

$$(c, \tau) \geq \delta_\tau + (c_{\text{pred}}, \tau_{\text{pred}}), \forall \text{pred} \in \text{predecessors(signal)}.$$

This is where new pipeline levels are added: the addition in the previous equation is a lexicographic addition for a cycle of latency $1/f$, where f is the target frequency. Defining $\delta_{\text{obj}} = 1/f - \delta_{\text{ff}}$ where δ_{ff} is the delay of a register, the lexicographic time addition is:

$$(c, \tau) + \delta = \left(c + \left\lfloor \frac{\tau + \delta}{\delta_{\text{obj}}} \right\rfloor, \quad \frac{\tau + \delta}{\delta_{\text{obj}}} - \left\lfloor \frac{\tau + \delta}{\delta_{\text{obj}}} \right\rfloor \right)$$

When rescheduling a signal, in addition to the original constraint, a similar constraint must be met with respect to the successors: $(c, \tau) \leq (c_{\text{succ}}, \tau_{\text{succ}}) - \delta_\tau(\text{succ}), \forall \text{succ} \in$ successors(signal) .

E. Sub-components

An operator constructor can also launch a chain of constructor calls (top left part of Fig. 2. Each subcomponent constructor builds its S-Graph, which is then integrated into the S-Graph of the parent operator. The result is that a hierarchy of components can also be managed as a fully flattened S-Graph.

Subcomponents are, by default, inlined, each instance of the same component being scheduled independently (line 4). This provides the best performance, as the schedule of an instance may depend on the schedule of its inputs. However, for components reused many times, there is also the option to schedule only once each component and reuse this schedule for all instances. This prevents a size explosion of the S-DAG (and the generated VHDL) in such cases, but imposes that pipeline stages inside such shared-schedule instances will be identical. This option is essentially used for elementary (atomic) components such as compressors which are too small to be pipelined anyway.

F. Functional registers

Using the `delay()` method, designers may also explicitely insert functional registers. These will be inserted during the final VHDL generation in addition to the pipeline registers: functional registers don't affect scheduling.

IV. Circuit Design with Timing Information

The ability to use timing information during circuit construction is the major motivation for the incremental scheduling approach presented here. The bit heap framework, introduced in [7] and improved in [8], illustrates this need, because bits will arrive in the bit heap at various lexicographic times. For instance, when a multiplier is built out of DSP blocks and logic, it must add bits coming from DSP blocks after two or three cycles, and bits that arrive from LUT-based multipliers after one LUT latency only. Another example would be a FIR where each constant product comes from a highly constant-dependent shift-and-add architecture, with a correspondingly constant-dependent timing. In all such cases, an optimal compressor tree should group together bits of similar lexicographic time: the construction of the circuit graph itself depends of the schedule of a part of the graph.

Circuits with loops, such as IIR filters, are another class of circuits that benefit from incremental scheduling. The chosen solution is to assume that a designer knows that there is a loop in the circuit: loops must be tagged as such by the designer in the constructor code. This results in immediate feedback on the maximum achievable frequency, which can be further used to constraint the target frequency of the rest of the circuit.

V. Target models

The `Target` virtual class tries to encapsulate the performance capabilities of the various target FPGAs in a way that is as generic as possible. It defines virtual methods that can be used in operator constructor code. These methods are implemented in the actual instances of the `Target` class.

In the new framework, the design choices were motivated by genericity to three different targets, each with a different design suite: Virtex-6 with ISE, Kintex-7 with Vivado, StratixV with Quartus II. We also tried to pave the way for ASIC target classes.

Here are some methods used in this work, by order of importance. Each of them returns a delay in seconds.

- `ffDelay()` returns the delay of a flip-flop. So far, this method just returns a constant.
- `logicDelay(n)` returns an estimation of the delay of an arbitrary logic function of *n* arguments. Its implementation is architecture dependent, reflecting for instance the hierachy of muxes that can be used inside a Xilinx slice before having to go to the general routing. This method supersedes the less generic `lutDelay()` of previous versions.
- `adderDelay(n)` returns the delay of an addition of size *n*. Again the implementation is very different for our three targets: Fast carry logic has a granularity of 1 on Virtex-6, 4 on Kintex-7, and 10 on StratixV.
- `wideOrDelay(n)`, `eqComparatorDelay(n)`, `eqConstComparatorDelay(n)` attempt to capture the use of fast-carry logic to implement wide OR and wide AND operations.

For each of these methods, the returned delay now also includes an estimation of the local routing delay. In previous versions, such delays had to be added in constructor code, which turned out to be very repetitive.

Target	Operator	estim. delay	measured delay
Virtex6	IntAdder 32	1.23 ns	1.54 ns
(xc6vhx380T-3)	Shifter 63	2.2 ns	2.0 ns
/ ISE	FPAdd 8 23	19.2 ns	11.4 ns
Kintex7	IntAdder 32	1.4 ns	1.4 ns
(xc7k70tfbv484-3)	Shifter 63	6.28 ns	6.8 ns
/ Vivado	FPAdd 8 23	19.8 ns	17.2 ns
StratixV	IntAdder 32	1.26 ns	1.39 ns
(5SGXEA3K1F35C1)	Shifter 63	2.68 ns	2.88 ns
/ Quartus	FPAdd 8 23	17.6 ns	9.8 ns

TABLE I: Accuracy of our target models

target		performance	resources
StartixV	FloPoCo	7+2 cycles @ 421 MHz	492R + 231L
/ Quartus	ALTFP 16.0	7+2 cycles @ 333 MHz	477R + 358L
Virtex6	FloPoCo	8+2 cycles @ 408 MHz	470R + 461L
/ ISE	FloatingPoint6.1	8+2 cycles @ 427 MHz	500R + 416L
Kintex7	FloPoCo	8+2 cycles @ 401 MHz	463R + 339L
/ Vivado	FloatingPoint7.1	8+2 cycles @ 476 MHz	499R + 334L
	FloatingPoint7.1	6+2 cycles @ 417 MHz	384R + 344L

TABLE II: Pipelining a floating-point adder with registers on the I/O, for three different targets using three different vendor tools. For each target, the FloPoCo operator is compared to the equivalent vendor IP.

Capturing the delay of embedded memories and DSP blocks is more complex, since these blocks have internal registers, various chaining possibilities, dual-porting, etc. These features are best used through FloPoCo operators such as `Table` and `IntMult`, which take care of this complexity. Inside these operators, we are not ashamed to have target-specific ad-hoc implementations.

Table I evaluates the accuracy of these models. The first line of each target shows that prediction of small logic units (here one single addition) is very accurate. A 63-bit barrel shifter consists of 6 logic levels, but the constructor code of the corresponding operator attempts to pack two or three of them in a LUT, depending on the number of LUT inputs reported by the `lutInputs()` method of `Target`. As the second line of each target shows, this prediction remains relatively accurate. Finally, for a complete floating-point adder, the tools find more opportunities to fuse logic in large LUTs, and our predictions become very pessimistic. However, as Table II shows, in a pipelined operator, there are few logic delays in each level, and the models, again, work well.

Let us end this section with something that doesn't work well: taking into account large fanouts. We do have a `fanoutDelay(n)` method, but its current implementation is very arbitrary. The problem is that there are many, many ways, in modern FPGAs, to route large fanout signals, and the tools do exploit this freedom when routing for a prescribed clock constraint. As a consequence, although the tools do report fanout information and the associated delays in the detailed critical path timing reports, it seems impossible to turn this information into an accurate model: firstly, the fanout mentioned there have very little relationship with the logical

fanout. For instance, in the 63-bit shifter, we expect several large fanout signals which are increasing powers of two, but the largest fanout reported is 5. Secondly, there is no strict relationship between the fanout and the reported delay, as Figure 6 illustrates: we have there fo=8, delay=2.17ns as well as fo=143, delay=1.086ns. The current choice is an ad-hoc linearization of this figure, but it will obviously be very inaccurate. The good news is that the tools seem to be able to compensate if we underestimate the fanout delay, so this is the sensible thing to do.

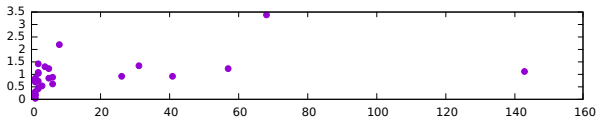

Fig. 6: Plots of net delay versus fanout in the floating-point adder on Kintex-7 (placed and routed design).

specification			performance	resources
wE=8 wF=23	400 MHz	old:	9+2 cycles @ 423 MHz	604R + 233L
		new:	7+2 cycles @ 421 MHz	492R + 231L
	300 MHz	old:	7+2 cycles @ 305 MHz	505R + 208L
		new:	5+2 cycles @ 354 MHz	375R + 225L
	200 MHz	old:	3+2 cycles @ 232 MHz	281R + 248L
		new:	3+2 cycles @ 226 MHz	281R + 250L
	100 MHz	old:	2+2 cycles @ 132 MHz	234R + 264L
		new:	1+2 cycle @ 126 MHz	149R + 233L
	comb	both:	0+2 cycle @ 102 MHz	102R + 242L
wE=11 wF=52	400 MHz	old:	14+2 cycles @ 414 MHz	1690R + 628L
		new:	10+2 cycles @ 330 MHz	1077R + 531L
	300 MHz	old:	7+2 cycles @ 270 MHz	976R + 498L
		new:	7+2 cycles @ 299 MHz	906R + 502L
	200 MHz	old:	5+2 cycles @ 220 MHz	653R + 532L
		new:	4+2 cycles @ 217 MHz	592R + 478L
	100 MHz	old:	2+2 cycles @ 130 MHz	450R + 509L
		new:	2+2 cycles @ 127 MHz	345R + 503L
	comb	both:	0+2 cycles @ 82 MHz	198R + 514L

TABLE III: Comparison between old and new FloPoCo on FPAdd (single and double precision) with registers on the I/Os.

VI. Results

The main positive result of this work is the simplification of the design of complex operators. `FPAddSinglePath.cpp`, which describes our floating-point adder case study, was reduced from 557 down to 472 lines of code. Many of these lines were dedicated to explicit synchronization management, which is no longer required. Having to worry only about *local* timing information makes life much simpler.

Generation time is still very fast, with all operators in this article generated in a fraction of a second. Working at the level of VHDL signals, we have much fewer signals to manage than working at the level of the gate or the bit. Of course, this aggregated timing information is less accurate. This probably explains the slightly better results of Xilinx IP in Table I.

Still, Table III shows that in most cases, the new pipeline framework improves the generated pipelines by reducing the latency, increasing the frequency and reducing resource consumption.

This data was obtained using the following command line:

```
flopoco target=StratixV frequency=400 \
        FPAdd we=8 wF=23   Wrapper
```

The `Wrapper` operator simply adds registers on the inputs and outputs of the previous `FPAdd`. The synthesis results themselves were obtained for StratixV (5SGXEA3K1F35C1) using Quartus 16.0 and the `tools/quartus_runsyn.py` utility of FloPoCo. They should be reproducible.

Since the only difference between both versions is in the pipeline, the combinatorial operator is identical between them. Its clock constraint was set to 400 MHz.

VII. Conclusion

By raising the abstraction level offered to the designer, this work allows her to write, with very little effort, generic code that produces complex pipelines of high quality for a range of targets and a range of frequencies. The quality of the generated IP is on par with that of vendor tools, but of course the purpose of FloPoCo is to keep researching arithmetic operators not provided by vendor tools.

Working with the new generation of back-end tools from Altera and Xilinx has proven very challenging. However, as these tools require a clock constraint to implement a design, there is a clear convergence here with the frequency-directed optimization philosophy of FloPoCo.

Current effort focuses on refining the target models further, completing the upgrade of the bit heap into this framework, and porting all the operators that depend on it. This includes large multipliers and squarers, constant multipliers, various elementary functions and function generators, and filters.

With a solid management of pipelines with loops, we also plan to take FloPoCo beyond its initial domain into signal processing.

Acknowledgements

This work was supported by the French National Research Agency through the INS program *MetaLibm*.

References

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

[2] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[3] A. J. Chung, K. Cobden, M. Jervis, M. Langhammer, and B. Pasca, "Tools and techniques for efficient high-level system design on FPGAs," in *First International Workshop on FPGAs for Software Programmers*, 2014.

[4] B. Gaide, "Methods of pipelining a data path in an integrated circuit," Nov. 18 2014, US Patent 8,893,071.

[5] I. Ganusov, H. Fraisse, A. N. Ng, R. T. Possignolo, and S. Das, "Automated extra pipeline analysis of applications mapped to Xilinx UltraScale+ FPGAs," in *Field Programmable Logic and Applications*, August 2016.

[6] G. Venkataramani and Y. Gu, "System-level retiming and pipelining," in *Field-Programmable Custom Computing Machines*, May 2014, pp. 80–87.

[7] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *Field-Programmable Logic and Applications*, Sep. 2013.

[8] M. Kumm and P. Zipf, "Pipelined compressor tree optimization using integer linear programming," in *Field Programmable Logic and Applications*. IEEE, 2014.