Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Pipelined FPGA Adders

Florent de Dinechin,
Hong Diep Nguyen,
Bogdan Pasca

LIP, Arénaire
CNRS/ENSL/INRIA/UCBL/Université de Lyon
46, allée d'Italie, 69364 Lyon Cedex 07, France
Florent.de.Dinechin@ens-lyon.fr,Hong.Diep.Nguyen@ens-lyon.fr,Bogdan.Pasca@ens-lyon.fr

April 2010

Research Report N° 2010-16

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

Pipelined FPGA Adders

Florent de Dinechin, Hong Diep Nguyen, Bogdan Pasca

LIP, Arénaire

CNRS/ENSL/INRIA/UCBL/Université de Lyon

46, allée d'Italie, 69364 Lyon Cedex 07, France

Florent.de.Dinechin@ens-lyon.fr,Hong.Diep.Nguyen@ens-lyon.fr,Bogdan.Pasca@ens-lyon.fr

April 2010

Abstract

Integer addition is a universal building block, and applications such as quad-precision floating-point or elliptic curve cryptography now demand precisions well beyond 64 bits. This study explores the trade-offs between size, latency and frequency for pipelined large-precision adders on FPGA. It compares three pipelined adder architectures: the classical pipelined ripple-carry adder, a variation that reduces register count, and an FPGA-specific implementation of the carry-select adder capable of providing lower latency additions at a comparable price. For each of these architectures, resource estimation models are defined, and used in an adder generator that selects the best architecture considering the target FPGA, the target operating frequency, and the addition bit width.

Keywords: adder, pipeline, FPGA

1 Introduction

Integer addition is used as a building block in many coarser operators. Examples which require large adders include integer multipliers, most floating-point operators, and modular adders used in some cryptographic applications. In floating-point, the demand in precision is now moving from double (64-bit) to the recently standardized quadruple precision (128-bit format, including 112 bits for the significand) [1]. In elliptic-curve cryptography, the size of modular additions is currently above 150 bits for acceptable security.

This study presents an operator generator for binary integer addition that is based on resource estimation models of possible implementations. Given a specification including a target frequency, the generator queries the implementation models in order to select the one matching this frequency at minimal cost. Once found, the VHDL code of the selected implementation is generated.

Adders differ in the way they propagate carries. Modern FPGAs include special hardware dedicated to carry propagation [2, 3, 4, 5]. Sending a carry to a neihbouring cell through the dedicated carry line is much faster than sending a bit to the same cell through the general reconfigurable routing fabric. Therefore, proven solutions for VLSI designs [6] bring little speed improvement on FPGAs over the ripple carry adder (RCA) excecpt for addition size exceeding 64 bits [7]. These speed improvements are small, and they come at a cost penalty exceeding a factor 2 over the RCA. Therefore, a binary addition is expressed in VHDL as a + and is implemented by default as an RCA.

This article re-evaluates this situation when a *pipelined* adder is needed. Pipelining is used for cutting the critical path in order to increase operator frequency. To the best of our knowledge, there is no IP core generator nor VHDL/Verilog library which provide high-performance pipelined binary adders for FPGAs. This work introduces the adder generator used in the FloPoCo project[1] as a building block of most other operators.

The main contributions of this work are:

- an alternative pipelining of ripple-carry adder;
- a novel short-latency pipelined adder;
- resource estimation models including slice, register and LUT count for three adder architectures;
- integration of these models into an addition operator generator that takes as input a list of user specifications, and returns the VHDL code of the best operator.

1.1 Related Work

The simplest pipelining of binary addition [8, 9, 6] consists in buffering the carry-out of each full-adder (FA) along the carry propagation path, and inserting synchronization registers for I/O. The previous technique is wasteful when the objective period is larger than the delay of a 1-bit carry propagation. For these cases, a better version [10, 6, 11] consists in registering carries only every α FA cell. This technique will be detailed in section 2.1, and is referred to as the *classical* RCA pipelining technique.

[1] http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

Faster techniques than the previous classical architecture have been developed for VLSI. A first idea is to speed up the logic on the carry propagation path [12, 9]. Other, more algorithmic approaches include carry-select, carry-skip, and the family of prefix adders [6]. These designs map poorly on FPGAs, however they have served as an initial source of inspiration for the proposed pipelining techniques from section 2.3.

A complete study on unpipelined binary FPGA addition is presented in [7]. The authors present FPGA-specific optimization opportunities for carry-skip and carry-select adders and show that optimized versions of these adders can be faster than the RCA for large addition sizes. However, these faster versions come at at a significant size penalty, which recommends them only for delay-critical applications. Moreover, pipelining is not covered. The present article extends this previous study to pipelined addition.

1.2 FPGA addition in the FloPoCo context

FloPoCo is a generator of arithmetic cores (**Flo**ating-**Po**int **Co**res, but not only) for FPGAs. FloPoCo also provides a framework for arithmetic operator development that is, to our knowledge, the easiest way to design complex operators with flexible pipelines [13]. The operators presented in this paper have been developed using the FloPoCo framework and are essential building blocks of most complex FloPoCo operators.

FloPoCo generates arithmetic operators in human-readable synthesizable VHDL starting from a list of user specifications (see Figure 1). These specifications include: operator parameters (operand width for binary addition), deployment FPGA target, target frequency and others. One of the original features of FloPoCo is that operator generation is *frequency-driven*. Instead of generating the fastest possible operator, the FloPoCo philosophy is to provide the smallest operator meeting a frequency constraint. This approach has the advantage of being compositional: a larger operator working at frequency f may be assembled out of sub-components working at frequency f. This study formalizes frequency-driven addition pipelining.

1.3 Design-space exploration by resource estimation

Modern FPGA resources are heterogeneous, including LUT-based logic, embedded memories, embedded DSP blocks, and others. For addition, we only need to estimate logic and registers. This study gives resource estimation formulae for these resources for several Xilinx FPGAs. Altera targets are currently only partially supported. This doesn't mean that FloPoCo operators do not work on Altera, just that they are not optimized accurately.

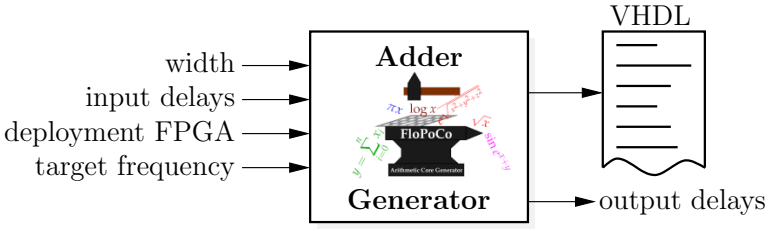

Figure 1: FloPoCo adder generator

The formulae allow for a fast and exhaustive design-space exploration, where only the selected architecture will be generated and synthesized. For this method to be valid, we will check in 3.1 that these formulae effectively predict the performance and resource consumption of the operator after synthesis and technology mapping. Addition and register mapping is simple enough for these formulae to be accurate to about one percent in all cases.

1.4 FPGA targets

In the FloPoCo framework, each FPGA is abstracted to a list of essential attributes: LUT features, routing delays, DSP configurations, on-chip memory, etc..

The Xlinx VirtexII-Pro[2], Spartan3 [3] and Virtex-4 [4] FPGAs have very similar slice structure (Figure 2): two 4-input LUTs with corresponding flip-flops and arithmetic logic for carry-bit computation and propagation. Carry-bit propagation is accomplished by means of dedicated carry-chains running vertically through the FPGA layout.

This is the default slice type and is denoted by `sliceL`. In addition, a secondary slice type featuring a superset of functionalities is available. The `sliceM`s allow the LUT to be configured as a variable-length shift-register (SRL16). When this configuration is used, shift registers of up-to 16 bits can be absorbed in one half-slice. This feature, when available, allows minimizing input/output synchronization cost.

The Virtex-5 and Virtex-6 slices [5] are similar with respect to addition. However, they allow independent use of the LUTs and registers, which means that estimation formulae have to count them separately.

2 Pipelined addition on FPGA

Let X, Y be two integers on w bits (in the range $\{0, ..., 2^w - 1\}$) and c_{in} a carry-in bit. The sum of X, Y and c_{in} is noted $R = X + Y + c_{in}$. It is in $[0, 2^{w+1} - 1]$ and is representable on $w + 1$ bits. Note that all the following also applies to signed integers in 2's complement notation.

The RCA delay is proportional to the addition size. It has three components. First, the LUT delay, δ_{LUT}, used to precompute the carry multiplexer select signal. Then there is a worst-case delay of $(w-1)\delta_{\text{carry}}$ for carry propagation. Finally, δ_{xor}, the delay of the xor gate used to compute the MSB sum bit.

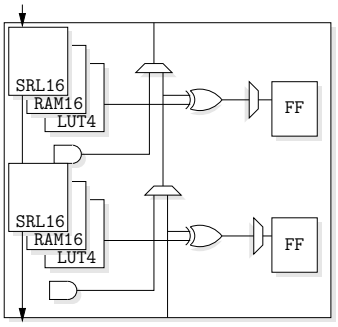

Figure 2: `sliceM` (VirtexII-Pro, Spartan3 and Virtex-4)

$$\delta_w = \delta_{\text{LUT}} + (w-1)\delta_{\text{carry}} + \delta_{\text{xor}}$$

As w increases the addition frequency decreases as illustrated in Figure 3 for three FPGAs.

In the context of frequency-driven pipelining, a pair (w, f) which is under the corresponding curve in Figure 3 meets the frequency constraint. There are two solutions for additions not meeting this constraint. We can choose a different addition architecture that is able to reach the frequency without too much of a cost penalty [7]. This solution is unable to cover the entire (w, f) space. Another solution is to pipeline the adder design such that the critical path of the circuit is less than the target period $T = 1/f$. This study focuses on the second solution, because it is more scalable and often consumes less resources.

2.1 Classical RCA Pipelining

A tight frequency-driven pipelining is obtained by first determining the maximal addition size α for which the critical path delay is less than the target period T:

$$\alpha = 1 + \left\lfloor \frac{T - \delta_{\text{LUT}} - \delta_{\text{xor}}}{\delta_{\text{carry}}} \right\rfloor \quad .$$

Next, the addition is split into k chunks of α bits (except the last chunk denoted by β, $\beta \leq \alpha$) such that $w = (k-1)\alpha + \beta$.

An instantiation of this architecture highlighting the previously discussed parameters is presented in Figure 4 for $k = 4$. As k decreases, the number of registers used for synchronization decreases. When the critical path of the w-bit addition is $\leq T$, no pipelining is required $(k = 1)$ and the addition may be expressed as a simple + in VHDL.

The column labelled Classical in Table 2 presents the resource estimation formulae function of α, β, w, k, respectively with and without allowing shift-register packing in LUTs (SRL). Let us now explain how such formulae were built.

2.2 Resource estimation techniques

Let us take as a running example the previous classical architecture, annotated on Figure 5.

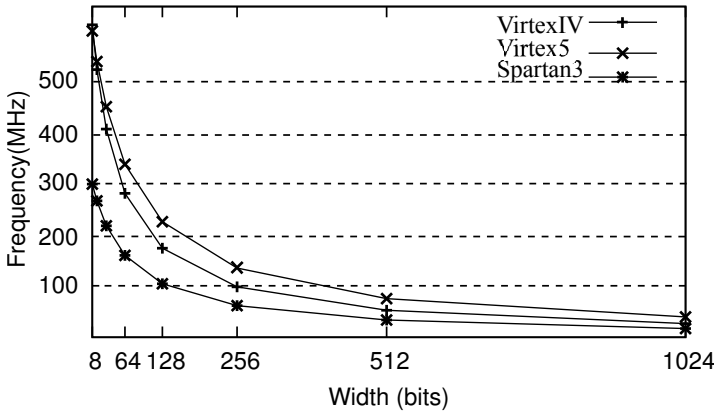

Figure 3: Ripple-Carry Addition Frequency for VirtexIV, Virtex5 and Spartan3E

The LUTs of the Xilinx FPGAs can be be used either as a function generator or as a variable length shift-register, as previously presented in Section 1.4.

For classical architecture, the addition diagonal uses w LUTs configured as function generators (Figure 5, σ). The LUT SRL configuration is used wherever two or more flip-flops are cascaded to form a shift register. This is the case of the $(k-2)\alpha$ SRLs under the addition diagonal (Figure 5, ξ), together with the 2β SRLs corresponding to the last column of width β (Figure 5, μ) and of the $2(k-3)\alpha$ SRLs above the diagonal (Figure 5, θ). These sum up to $w + (3k-8)\alpha + 2\beta = (4k-9)\alpha + 3\beta$, which is the value reported in Table 2.

There is one consideration to be made before counting registers: each time an SRL is used, the corresponding slice flip-flop is also used. In other words, for a p-level shift-register, $p-1$ levels are pushed into the SRL and one into the flip-flop. Hence, we count $(3k-8)\alpha + 2\beta$ registers for the same number of SRL, and, in addition, α registers (Figure 5, ϕ) under, 2α registers (Figure 5, ρ) above the diagonal plus the $k-1$ registers for the carry-bit propagation. These total $(3k-5)\alpha + 2\beta + k - 1$, the value reported in Table 2.

The next task is to count slices. We choose to count half-slices and divide this number by 2 rounding upwards. This corresponds to a dense placement of the pipelined adder, which the tools are expected to favor. Experimental results given in section 3.1 will validate this assumption.

The number of half-slices used by the classical implementation is: w for the diagonal addition, $(3k-8)\alpha + 2\beta$ for the SRL and corresponding flip-flops, and $3\alpha + k - 1$ for the independent registers. However, we subtract α as the left-most addition of α bits includes the registers in the same slice as the LUT. The number totals $(4k-7)\alpha + 3\beta + k - 1$, which is reported in Table 2.

All the formulae presented in this paper were deduced using these techniques. Relative errors of these estimation formulae are given in Table 3. The worst case relative error is of the order of 10^{-2} (one percent) which makes them sufficiently accurate for estimation formulae.

2.3 Alternative RCA Pipelining

The classical pipelining technique requires a significant amount of registers for input synchronization. This number may be lowered by performing the chunk additions at the first pipeline level and then propagating these sums instead. When no SRL are allowed, the number of registers propagated above the diagonal will be approximatively halved, and may still be packed in shift registers. An instantiation of this architecture for $k = 4$ is presented in Figure 6.

Each adder on the addition diagonal takes as input an operand on $\alpha + 1$ bits and a 1-bit carry in and returns a $\alpha + 1$-bit wide result. This addition does not overflow, as the $\alpha + 1$-bit input was the result of an addition of two α-bit numbers with a carry-in of 0.

The resource estimation formulae for this architecture are presented in Table 2.

2.4 Short-Latency Addition Architecture

Given a target frequency f, the pipeline depth of the previously presented architectures increases linearly with addition size. In this section we propose a scalable low-latency addition architecture based on the textbook carry-select architecture, whose novel feature is to make efficient use of the fast-carry chains.

The algorithm first determines the chunk size α as per section 2.1. Next, two sums are computed for each pair of chunks: $X_i + Y_i$ and $X_i + Y_i + 1$. The final result R is a combination

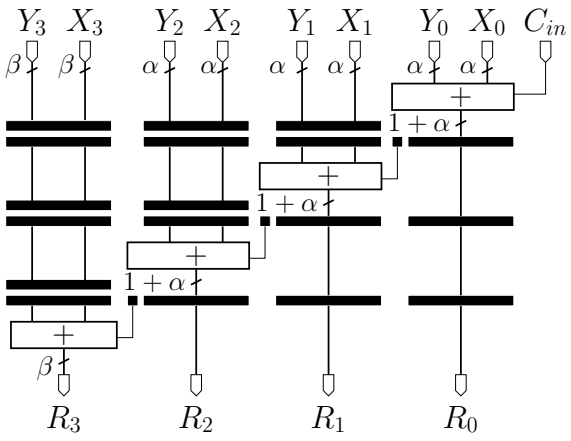

Figure 4: Classical addition architecture [6]

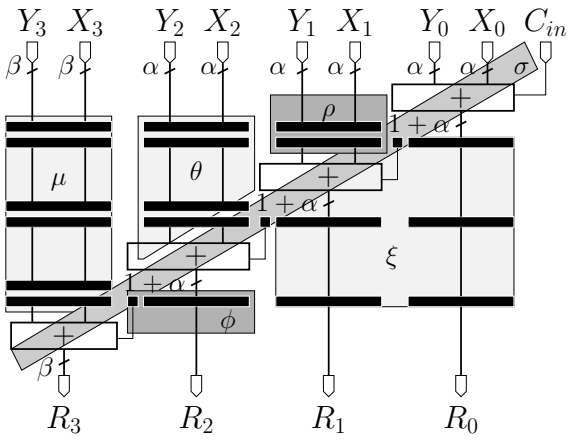

Figure 5: Annotated classical architecture

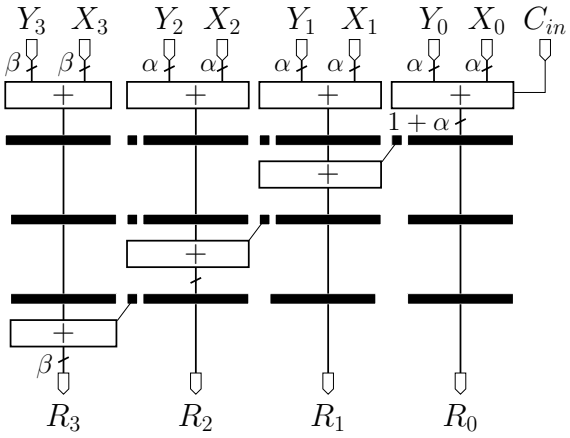

Figure 6: Proposed FPGA architecture

of the corresponding sub-sums and is found in a space of 2^k combinations. Selecting the appropriate sub-sum is done by using a carry-in bit. The novel idea in this algorithm is the use of the dedicated fast-carry chains to compute the carry-bits for the result selection.

Actually, for each chunk, a pair (sum, carry-out) is computed for both possible values of the carry-in. We use the following notations to denote the concatenation of the sub-sums and their corresponding carry-out bits.

$$c_i{}^0 S_i{}^0 = X_i + Y_i$$
$$c_i{}^1 S_i{}^1 = X_i + Y_i + 1$$

We denote by R_i the ith sub-result such that $R = R_{k-1} \dots R_1 R_0$. The value of R_i can be expressed in the following way knowing $S_i{}^0, S_i{}^1$ and c_{i-1}.

$$\text{if } (c_{i-1} = 0) \text{ then } R_i \leftarrow S_i{}^0$$
$$\text{else } R_i \leftarrow S_i{}^1$$

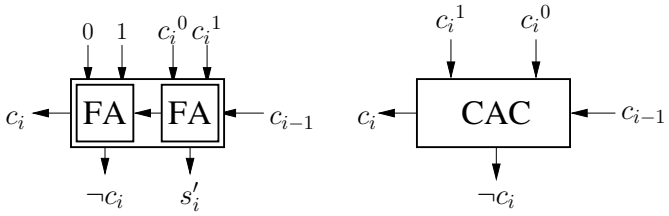

Figure 7: Carry-Add-Cell (CAC) implementation and representation

The carry-out bit for a chunk c_i is computed from its carry-in c_{i-1} and the two precomputed carries $c_i{}^0$ and $c_i{}^1$. The circuit used to compute them is particularly designed to take advantage of the fast carry chains of the FPGA by expressing the carry-out computation under the form of an addition (Figure 7):

$$c_i \neg c_i s'_i = c_{i-1} + c_i{}^0 + c_i{}^1 + 2$$

One can verify the correctness of the carry generation by checking the truth table presented in Table 1. Note that the greyed-out rows of the table will never be needed, as $c_i{}^0 = 1$ implies $c_i{}^1 = 1$ (it is not possible that $X_i + Y_i$ overflows and $X_i + Y_i + 1$ doesn't). The value of s'_i is not used further but is necessary for correct inference and mapping of the addition on the fast-carry chains of the FPGA.

It should be noted that a strong point of this approach is that this carry propagation is expressed as an addition, and therefore portable (no need for vendor-specific low-level LUT-filling primitives). For instance, porting it to Altera chips should resume to getting the parameters right.

The formulae presented in Table 2 are deduced for $k \geq 3$. To use them we thus have to ensure $w \geq 2\alpha + 1$, possibly by reducing α with respect to the optimal α deduced from the target frequency.

The short-latency architecture depicted on Figure 8 has a constant latency of two cycles. In addition, for lower frequency operators, the second register levels can be discarded. However, choosing the correct splitting for the inputs is not trivial as we have to ensure that the critical path length is smaller than the target period T. Considering that the first sums are registered, we have to find the correct sizes for splitting the inputs, such that the critical path length that includes the carry generation circuit and the final additions is less than T.

Intuitively, as the index of the chunks added is higher, the length of the corresponding carry bit propagation is longer and thus the length of the final addition has to be smaller. We

Table 1: CAC Truth table. Greyed-out rows are not needed

c_{i-1}	$c_i{}^0$	$c_i{}^1$	c_i	$\neg c_i$	s'_i
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	1	0	0
1	1	1	1	0	1

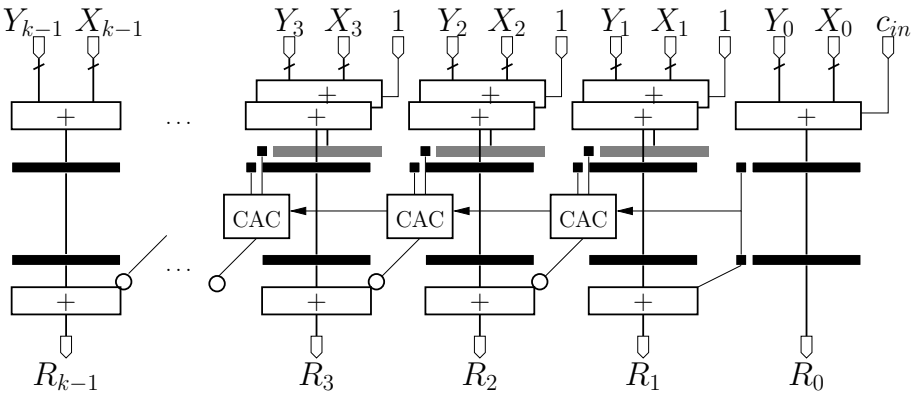

Figure 8: Short-Latency Addition architecture

use a greedy algorithm that, at index i finds the maximum addition size such that the carry propagation for index i and the final addition for this index is smaller than T. However, it is possible that for a given input size w and a target frequency f, such a solution does not exist. In this case the second register level is inserted, and the chunk size becomes α.

In addition to latency reduction, this optimization brings the following gains: the number of registers is reduced by the carry propagation size (which now needs no registering), the LUT count is reduced by approximatively w, and the number of slices by approximatively $w/2$.

Finally, the scalability of this architecture may be ensured by pipelining the carry propagation circuit. Once $k > \alpha/2$ the length of the carry propagation becomes greater than the target period and violates the constraints. In this case we pipeline this addition with the best pipelining algorithm function on its size. The increase in resources of the obtained architecture only equals the increase in size of the carry-propagation operator, as the possible delay introduced by this operator will be transparently absorbed by the shift registers.

2.5 No packing of shift registers in LUTs (SRL)

The addition architectures presented so far make extensive use of the shift-registers available in the SLICEMs. However, this resource is getting rarer over the years. All the slices in a

Table 2: Resource estimation formulae for the tree pipelined adder architectures with and without shift-register extraction (SRL)

		Classical	Alternative	Short-Latency
SRL	**REG**	$(3k-5)\alpha + 2\beta + k - 1$	$(2k-3)\alpha + \beta + 2k - 3$	$(k-1)\alpha + \beta + 4k - 7$
	LUT	$\begin{cases} \alpha+\beta & : k=2 \\ (4k-9)\alpha+3\beta & : k \geq 3 \end{cases}$	$\begin{cases} \alpha+2\beta & : k=2 \\ (4k-8)\alpha+3\beta+k-3 & : k \geq 3 \end{cases}$	$(4k-6)\alpha + 3\beta + 2k - 4$
	SLICE	$\lceil((4k-7)\alpha + 3\beta + k - 1)/2\rceil$	$\begin{cases} \lceil(\alpha+2\beta+1)/2\rceil & : k=2 \\ \lceil((4k-8)\alpha+3\beta+2k-5)/2\rceil & : k \geq 3 \end{cases}$	$\lceil((4k-6)\alpha + 3\beta + 2k - 4)/2\rceil$
No SRL	**REG**	$\frac{3k^2-7k+4}{2}\alpha + 2(k-1)\beta + k - 1$	$(k-1)w + k^2 - 2k + 1$	$2w + 3k - 5$
	LUT	w	$2w - \alpha$	$3w - 2\alpha - \beta + 2(k-2)$
	SLICE	$\left\lceil \left(w + \frac{3(k^2-3k+2)}{2}\alpha + 2(k-1)\beta \right) /2 \right\rceil$	$\left\lceil ((k-1)w + \beta + k^2 - 2k + 1)/2 \right\rceil$	$\lceil(4w - 2\alpha - \beta + 2k - 4)/2\rceil$

VirtexII-Pro device were similar to SLICEMs, but they were reduced to half the total number of slices for Virtex4 and Spartan3, and about a quarter in Virtex5 and Virtex6 devices (with higher density at the input of the DSP48E blocks). There is an ISE option that prevents using this resource. It may therefore be relevant to be able to generate adders with this in view.

Out of the presented architectures, the low-latency one will behave better when no shift registers are allowed. This is due to the fact that it requires less registers for synchronization. When $k = 2$, the alternative implementation behaves better than the classical one, as it propagates approximatively half as many signals on the upper part of the addition diagonal. Resource estimations for the three architectures when not allowing SRLs are presented in Table 2.

2.6 Managing partial cycle delays

By assembling two pipelined components A and B working at frequency f with registers between them, one obtains an operator $A|B$ that also works at frequency f, whose latency is the sum of those of A and B, plus one. However, one may sometimes save the registers between A and B if this doesn't introduce a critical path longer than the target period. The FloPoCo framework includes experimental support for this possibility. In general, a component may input a vector of input delays, and will report the delays on each of its input (see Figure 1). It could also work from output to input, this is an arbitrary choice.

Back to adders, for the classical architecture, in the presence of an input delay, the upper-rightmost addition now needs to use a γ chunk size, $\gamma < alpha$ so that the period of the γ addition is less than T minus the input delay. The rest of the chunks are split as before, as they are registered anyway. We now have $w = \beta + (k-2)\alpha + \gamma$. The cost impact on the architecture is dictated by $\beta_{old} > \gamma$ and the use of SRLs. The $\beta_{old} > \gamma$ leads to an increment in pipeline depth. This is absorbed by the shift-registers if available at no extra cost, or costs as much as $w/2$ slices.

For both alternative and low-latency architectures, there are two options: either perform all additions in using chunk size γ, or buffer the inputs and perform computations using chunk size α. For the alternative architecture, lower values for γ will increase the latency of the operator. When SRLs are available, the cost is maintained under control, otherwise the synchronization cost greatly increases. For the low-latency operator, a smaller γ may require pipelining the carry generation circuit. However, the size of this circuit remains small with respect to the total size.

All this shows that the best adder really depends on the context. Work is under way to exploit these new possibilities in FloPoCo.

3 Reality check

3.1 Estimation formulae

We have checked our estimation formulae against synthesis results using Xilinx ISE 11.1. Results presenting the resource usage estimations, obtained results and relative errors for both with and without SRLs are presented in Table 3 for a 128-bit addition synthesised on a Virtex4 (speedgrade -12) with a required frequency of 400MHz.

Table 3: Relative Error for the estimation formulae on a 128-bit adder Virtex4 (4vlx15sf363-12) for a requested frequency of 400MHz.

	Architecture	SRL	Results			Estimations			Relative Error		
			LUTs	regs	slices	LUTs	regs	slices	LUTs	regs	slices
128bit Virtex4(-12) 400MHz	Classical	N	128	573	309	128	573	300	0	0	$2 \cdot 10^{-2}$
		Y	318	292	198	318	292	194	0	0	$2 \cdot 10^{-2}$
	Alternative	N	222	392	216	223	393	207	$4 \cdot 10^{-3}$	$2 \cdot 10^{-3}$	$4 \cdot 10^{-2}$
		Y	352	199	183	352	199	177	0	0	$3 \cdot 10^{-2}$
	Short-Latency	N	288	264	216	293	263	214	10^{-2}	$3 \cdot 10^{-3}$	$9 \cdot 10^{-3}$
		Y	416	136	216	421	137	211	10^{-2}	$7 \cdot 10^{-3}$	$2 \cdot 10^{-2}$

Table 4: Synthesis results on Xilinx FPGAs (obtained using ISE 11.1)

Size	Freq	Target	Optimisation	Classical		Alternative		Short-Latency		Gain w/r classical
				Cost	Depth	Cost	Depth	Cost	Depth	
32bit	200	Spartan3	SLICE/SRL	62	4	62	4	76	2	0%
		3s200pq208-5	SLICE/-	110		84		64		41%
64bit	450	Virtex4	SLICE/SRL	96	2	81	2	109	2	15%
		4vlx15sf363-12	SLICE/-	113		82		110		27%
128bit	450	Virtex4	SLICE/SRL	247	5	230	5	258	2	6%
		4vlx15sf363-12	SLICE/-	516		369		258		50%
128bit	450	Virtex5	REG/SRL	322	4	232	4	143	2	56%
		5vlx30ff324-3	REG/-	718		525		267		63%

First, it should be mentioned that all the synthesized adders met the frequency target. In addition, one may observe that the resource estimations are accurate for all criteria. The best estimations are obtained as expected for LUTs and registers. The slice estimations represent the lowest bound obtainable leading to underestimation of the result. Nevertheless, the relative error of the estimation remains small, of the order 10^{-2}, or one percent.

3.2 Synthesis results

Synthesis results for some combinations of the input specifications are presented in Table 4. We choose different target FPGA and different operating frequencies. For each architecture and set of specifications we present the costs reported by Xilinx ISE 11.1 and its pipeline depth. The last column shows the gain of using the generated addition operator against using the classical implementation.

The grey cells in Table 4 highlight the lowest costs for the given specifications. We can observe that for different addition sizes the lowest cost is obtained by different architectures. The accurate estimation formulae help choosing the best architecture given the specifications and obtain the reported gain.

4 Conclusions

This article addresses the construction of pipelined adders for large operands working at high frequencies, from specifications including operand size, deployment target, running frequency,

and optimization directives.

When the FloPoCo project was initiated, it was not expected that we would need to dedicate so much work to something as seemingly simple as integer addition on FPGAs. The reason why it became important is that addition is so pervasive. The presented adder generator provides subcomponents for integer multipliers and constant multipliers, and for most floating-point cores, including addition, multiplication, division and square root, and elementary functions. If we want these cores to work at a high frequency for double precision and beyond, we need high-performance adders, but we also need them to consume as little resources as possible. Therefore, the adder generation described here is frequency-driven (possibly inheriting the frequency from the wider context) and minimizes resource consumption, based on accurate resource estimation formulae for three alternative pipelined adder architectures.

Work is under way to integrate the proposed adders in all the coarser cores of the FloPoCo project, and to support more FPGA targets. Future work also includes extending the optimization options to include operator latency, and possibly combinations such as "LUTs and latency"

References

[1] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1 –58, 29 2008.

[2] *Virtex-II Platform FPGA Handbook*, Xilinx Corporation, 2000.

[3] *Spartan-3 Generation FPGA User Guide*, Xilinx Corporation, 2009.

[4] *Virtex-4 Virtex-4 FPGA User Guide*, Xilinx Corporation, 2008.

[5] *Virtex-5 FPGA User Guide*, Xilinx Corporation, 2009.

[6] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[7] S. Xing and W. W. Yu, "FPGA Adders: Performance Evaluation and Optimal Design," *IEEE Design and Test of Computers*, vol. 15, pp. 24–29, 1998.

[8] I. Unwala and E. Swartzlander, "Superpipelined Adder Designs," in *Circuits and Systems, 1993., ISCAS '93, 1993 IEEE International Symposium on*, May 1993, pp. 1841–1844.

[9] L. Dadda and V. Piuri, "Pipelined Adders," *Computers, IEEE Transactions on*, vol. 45, no. 3, pp. 348–356, Mar 1996.

[10] P. M. Martinez, V. Javier, and B. Eduardo, "On the design of FPGA-based Multioperand Pipeline Adders," in *XII Design of Circuits and Integrated System Conference*, 1997.

[11] R. Beguenane, J.-L. Beuchat, J.-M. Muller, and S. Simard, "Modular multiplication of large integers on FPGA," in *in Proceedings of the Thirty Ninth Asilomar Conference on Signals, Circuits and Systems*, 2005, pp. 1361–1365.

[12] J. M. Muller, *Arithmétique des Ordinateurs*. Masson, Paris, 1989.

[13] F. de Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *Field Programmable Logic and Applications*. IEEE, Aug. 2009.