Improved Carry Chain Mapping for the VTR Flow

Ana Petkovska*, Grace Zgheib*, David Novo*, Muhsen Owaida*, Alan Mishchenko† and Paolo Ienne*

*Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
{ana.petkovska, grace.zgheib, david.novobruna, mohsen.ewaida, paolo.ienne}@epfl.ch

†University of California, Berkeley, Department of EECS
alanmi@berkeley.edu

***Abstract*—Carry chains facilitate the implementation of adders and improve the performance of arithmetic circuits in FPGAs. The last version of the commonly used open-source *Verilog-to-Routing (VTR)* CAD flow now enables modelling carry chains in FPGA architectures. However, one of the shortcomings of the existing flow lies in its inability to identify arithmetic operations when described as gate-level circuits. Moreover, the VTR flow squanders most of the LUTs preceding the chain logic. This paper focuses on these two problems and proposes preprocessing the circuit before technology mapping to allow for a more efficient use of carry chains. The first proposed method maps logic on the carry chains for circuits expressed using a gate-level description. On average, it identifies about 30% more meaningful full adders than the existing tool flow operating on the RTL descriptions. Area is thus improved by up to 15% with an average of 6% for almost no delay penalty. Secondly, we increase the use of the LUTs preceding the chain logic by a factor 2 on average. This reduces delay (up to 9%) and area (up to 2%), compared to the existing VTR flow. The new approach is independent of the specific carry-chain architecture and can be generically adapted to any FPGA with built-in hardened adders.**

I. Introduction

Modern *Field-Programmable Gate Array (FPGA)* architectures have dedicated circuitry to boost the performance of arithmetic operations. Such circuitry includes *Digital Signal Processing (DSP)* blocks, multipliers, and carry chains. The latter, which are the focus of this paper, are designed to reduce the critical path and area of adders and subtractors. In typical FPGAs, each logic block includes one or more components of chain logic whose architecture depends on the manufacturer. For example, some carry chains have a ripple-carry structure [1], while others have a carry-lookahead structure [12]. Altogether, carry chains exhibit two main properties: (1) they offer almost null routing delay between two adjacent carry-chain nodes, as illustrated in Figure 1, and (2) they include dedicated full-adder circuitry to be able to implement adders with minimum utilisation of *Look-Up Tables (LUTs)*.

How to better use available carry chains is still an open research question. The standard approach is to use carry chains only when the arithmetic operations are defined by high-level primitives (e.g., '+' operator) in a *Register-Transfer Level (RTL)* language. For example, *Verilog-to-Routing (VTR)*, the de facto academic tool flow for FPGA architectural and CAD research [5], uses such an approach. However, this is not always optimal nor possible. In some cases, the RTL code is not available because the design comes as a gate-level netlist. Additionally, suboptimal mapping decision can be made by assigning adders to carry chains before technology mapping (i.e., design step that transforms the gate-level description of the input into a network of LUTs). Luu et al. [6] have recently shown that using carry chains provides an average speed up of approximately 15% for an area penalty of approximately 5%. However, in some benchmarks using carry chains led to slower and bigger circuits. Accordingly, intelligent heuristics for when the carry chains should be used is certainly necessary, and it should consider the interaction between the chain and the configurable logic of the FPGA. Furthermore, carry chains may be useful in other contexts that have no equivalent operators at RTL level, such as compressor trees [9] and perhaps XOR-rich cryptographic functions. Such opportunities are missed by standard tools and can only be addressed by a tedious and careful manual restructuring of the circuit.

In this paper, we address some of the shortcomings of RTL-based carry-chain detection and propose an alternative approach to improve the current support of carry chains in VTR. Firstly, we present an automatic mapping to carry chains for circuits expressed with a gate-level description. On average, we identify about 30% more adders than the original

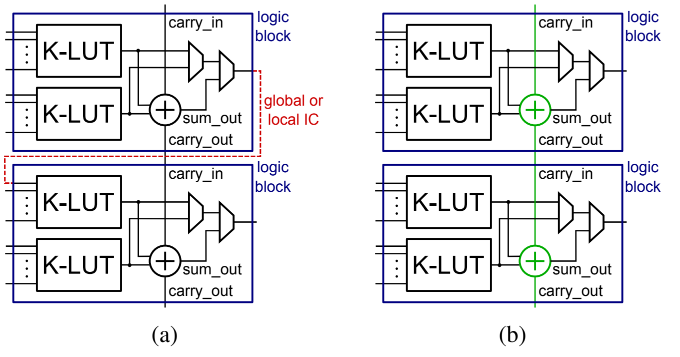
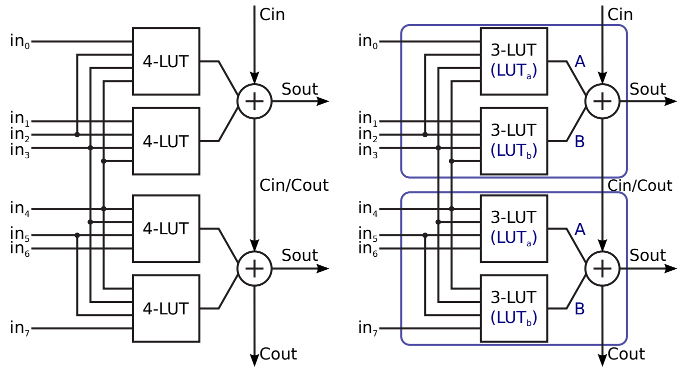


Fig. 1: Two types of interconnect between FPGA logic blocks. (a) Typically, logic blocks are interconnected by the local or global programmable interconnect. (b) A fast hardwired connection of the chain logic introduced for an efficient implementation of arithmetic circuits.



(a) (b)

Fig. 2: An example of arithmetic mode configurations. In arithmetic mode, hard adders with a dedicated carry chain are used along with some LUTs to compute their inputs. Different configurations of the LUTs are possible: (a) A typical configuration where the LUTs share multiple inputs and (b) a particular view of the previous configuration where part of the flexibility is relinquished to make each logic cell (adder with two LUTs) completely independent from the other.

VTR flow, which leads to 6% reduction in area for no change in delay. Secondly, we better exploit the LUTs that are in front of the hardened carry chain by premapping useful logic onto them instead of just bypassing signals. We show that our method increases the use of these LUTs by a factor of 2, on average (excluding inverters and buffers); this achieves up to 9% reduction in delay and 2% reduction in area compared to the original VTR flow. Our code is open-source (available at `http://lap.epfl.ch/downloads`) and can be easily integrated in the VTR flow for further research.

The rest of the paper is organised as follows. Section II summarises limitations of the tools we have studied and outlines our goals. Next, we describe our heuristics for automated mapping on carry chains for gate-level circuits in Section III and for mapping logic to the carry-chain LUTs in Section IV. After presenting our experimental setup in Section V, we present and discuss our experimental results in Section VI. Finally, we describe the state-of-the-art in Section VII before we conclude and present ideas for future work in Section VIII.

II. Making Good Use of Hardened Adders

In this section, we describe typical carry chains and the ordinary CAD flow that maps designs unto them. Then, we review the limitations we observed in commercial tools as well as VTR, and outline how we proceed about them in the following sections.

A. Carry Chains and Hard Adders

Practically all commercial implementations of carry chains rely on a special configuration mode of the basic logic block that involves hardened adders. As mentioned in the introduction, various implementations of the same idea exist, ranging from chains of full adders (the very reason for the term *carry chain*) to moderate size carry-lookahead blocks. All these architectures have in common (1) the existence of a hardened carry logic, (2) the presence of some LUTs in front of it, and (3) some connectivity constraints on the inputs of the LUTs. For instance, Figure 2a shows a logic block in arithmetic mode mimicking the *Adaptive Logic Module (ALM)* of an Altera Stratix V FPGA [1]. The two hardened full adders are connected through the carry signal, implementing a complete 2-bit ripple-carry adder, and receive their inputs, each, from two 4-LUTs. In this mode of operation, all four 4-LUTs share some inputs, which constrains the packing of logic functions onto them.

B. Discovering Carry Chains

To the best of our knowledge, confirmed by our experiments with various vendors tools, commercial CAD flows infer adders only or almost only when the arithmetic operators *plus* ('$+$') and *minus* ('$-$') are used in RTL. VTR supported hard adders and carry chains for the first time in version 7.0 and, for that purpose, the front-end ODIN-II was modified to detect addition and subtraction operations in the Verilog description of the circuits. In this simple heuristic to infer the use of hardened adders, one can typically specify a minimum bit width for which such adders are generated, because, in general, they are not beneficial for small bit widths (e.g., 2–3 bits). In this approach, identified hard adders are then provided as black boxes to the synthesizer and the technology mapper, and as such, they will never be modified or optimised after this point in the tool flow. We argue that this is not the best approach to discover carry chains: mainly, (1) it is impossible to discover chains in circuits where adders are implemented at the gate level, (2) misses opportunities of using full adders or small-bit-width adders in other situations not immediately derived from explicit additions, and (3) removes these components from the sight of logic synthesis, thus preventing even some basic logic optimisations such as constant propagation. Instantiating hardened adders before logic optimisation may easily lead to redundant or unnecessary logic not being eliminated [6]. Additionally, in the VTR flow, (4) the adders presented as black boxes prevent the synthesizer ABC from correctly modelling the critical path of the circuit. For all these reasons, we will present in Section III a technique to discover opportunities for hardened adders at the level of logic synthesis.

C. Using the Carry-Chain LUTs

Once the rest of the circuit is synthesised with the hardened adders as black boxes, technology mapping rewrites the gate-level netlist as a netlist of LUTs. The successive packing process decides which LUTs to place in the same cluster and, among others, if any logic can be placed in the LUTs that are immediately in front of the hardend full adders, which we call *carry-chain LUTs* for brevity. In this case, we have observed that commercial tools do a reasonable job in efficiently filling such LUTs; yet, the packer of VTR seldom or never makes any other use of such LUTs than propagating one particular input to the LUT output (that is, to the adder input). Part of the problem relates to the highly constrained connectivity of such carry-chain LUTs for which hardly the packer has enough flexibility to do the right choice; again, we argue that logic synthesis is the right time to prepare logic to fill such LUTs. Yet, in doing that, we need to simplify our problem by assuming that, whatever the architecture of the logic block is in arithmetic mode, there is a possibility of using the block in a way that there is no input constraint across independent bits of an adder. The example of Figure 2b clarifies this point: some of the block flexibility is relinquished by attributing inputs only to one or the other half of it, so that each bit of the hardened adder and all the LUTs in front of the inputs are completely independent. With this simplification, in Section IV, we will introduce a heuristic technique to try and make the best use of these carry-chain LUTs: it might be not fundamentally different from what some commercial tools appear to do, but we are unaware of published algorithms that perform this task.

III. Automated Mapping to Carry Chains for Gate-Level Circuits

As mentioned, our techniques rely on logic synthesis to improve the current state of the art. Since we use ABC [2] as our synthesizer, we define here the AIG structure, which is the fundamental representation of ABC. Then, we describe our method for automated discovery of carry chains for circuits with gate-level description.

An *And-Inverter Graph (AIG)* is a multilevel logic network composed of two-input AND gates and optional inverters. AIGs enable short runtime and high-quality results in synthesis, mapping, and verification due to their simplicity and flexibility [7], [8]. The AIG representation enables us to easily analyse the designs, detect and form carry chains, as well as premap logic for the carry-chain LUTs.

The intuitive way to recover full adders, which can be mapped on a carry chain, from the gate-level description is by searching for full adders that are connected through the `cin`/`cout` signals. For an AIG with bit-blasted adder logic, the proposed approach detects full-adder chains using structural matching. To detect chains, each full adder should exist in the AIG in terms of its structural cut-points. In other words, inputs and outputs of each full adder should correspond to some specific AIG nodes. We start by enumerating all three-input cuts of all internal AND nodes in the AIG along with their truth tables. This allows detecting full adders as pairs of nodes having two cuts with the same three inputs and with logic functions equal to XOR3 and Majority. The XOR3 and Majority cuts implement the `sum_out` and the `cout` output, respectively. Next, full-adder chains are detected by finding full adders connected to each other via carry-in/carry-out connections. To avoid adding combinational loops, we create a topologically ordered sequence of full-adder chains. Then, we reconstruct the AIG by replacing full-adder chains that are longer than some user-given size with chains of boxes. Each chain box has three inputs and two outputs, which represent the inputs and outputs of the corresponding full adder, respectively. In the resulting AIG, the inputs/outputs of the chain boxes are represented as additional primary outputs/inputs. For verification, the resulting AIG with boxes can be collapsed into a regular AIG, which is used to check equivalence with the original AIG. Figure 3 shows how the network is transformed from an AIG to an AIG with full-adder boxes. However, restrictions exist on the structural representation of the internal logic of the full adders. In general, the carry-chain architecture imposes no external fan-out and no inverters along the carry-in/carry-out paths. The generated chain boxes are guaranteed to satisfy these requirements by considering only carry connections with a single fan-out and by propagating negation of each `cout` to the inputs of the full adder, given the fact that $\overline{\texttt{cout}} = \text{Majority}(\texttt{A}, \texttt{B}, \texttt{cin})$ is equivalent to $\texttt{cout} = \text{Majority}(\overline{\texttt{A}}, \overline{\texttt{B}}, \overline{\texttt{cin}})$.

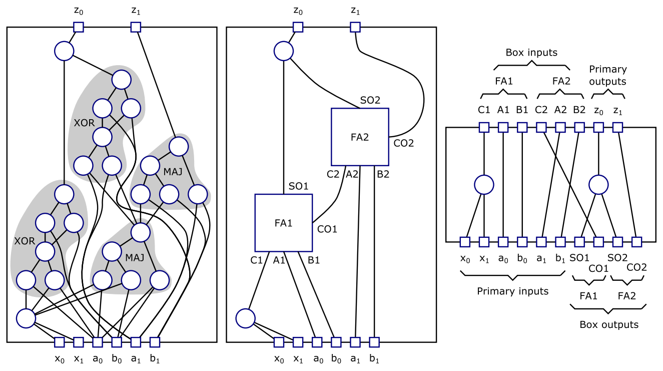


Fig. 3: Transformation from an AIG to an AIG with full-adder boxes. The input AIG (left) with the matched XOR3 and Majority cuts. The full adders are replaced with boxes and a chain of two full adders is discovered (middle). The resulting AIG (right) in which the inputs/outputs of the chain boxes are represented as additional primary outputs/inputs.

Additionally, to allow for an easy integration with the VTR flow, we provide a method to output the circuit with the found full-adder chains into a BLIF file. We explicitly instantiate these full adders using VTR hard-adder primitive to enable placing them on the FPGA carry chains. Similar to ODIN-II, this heuristic only extracts the adders, but does not define mapping for the LUTs connected to the carry chains.

Our algorithm is general because we ultimately detect complete adders that can be mapped as required to any hardened adder structure irrespective of its architecture. In other words, we detect and generate adders in ripple-carry form, but once this is done, they can be trivially transformed to fit any destination architecture by configuring properly the carry-chain LUTs. For example, for the architecture of Altera's Stratix V FPGA devices [1] shown in Figure 2a, it is enough to configure the carry-chain LUTs to propagate the adder inputs in a way that each logic block is literally a full adder of the identified ripple-carry adder. On the other hand, to implement a detected adder on the carry-lookahead structure of Xilinx's Virtex-7 FPGA devices [12], we should configure the carry-chain LUTs to implement the propagate and generate signals required for this type of hardened adder structure.

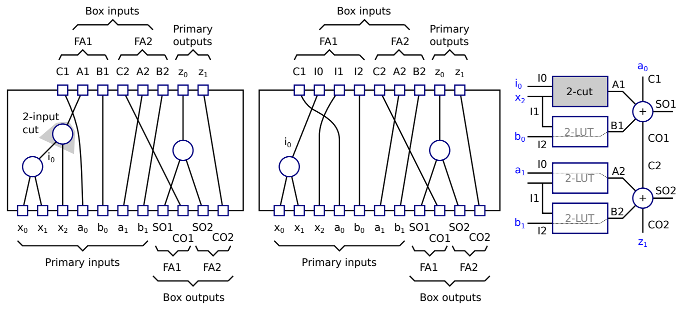


Fig. 4: Transforming an AIG with boxes after selecting cuts for the carry-chain LUTs. If the LUTs are 2-input with one shared input, in the original network (left) only one cut with inputs i_0 and x_2 is selected for the $A1$ input of the first full adder. The remaining network (middle) is copied and has the inputs of the selected cut, i_0 and x_2, as outputs. The selected cut is implemented with one of the LUTs from the pair, while the other is used as buffer for the signal b_0 (right). The carry-chain LUTs of the second logic block are also used as buffers.

IV. Mapping Logic to Carry-Chain LUTs

In this section, we present the heuristic for premapping logic into the carry-chain LUTs. The mapping for these LUTs is challenging due to the input sharing constraints presented in Section II-C. As mentioned there, we simplify these constraints by considering only input sharing between LUTs in the same block; that is, we restrict the flexibility of more general blocks, such as the one shown in Figure 2a, and use them in a simplified configuration such as the one of Figure 2b. This way, the mapping decision can be made for each pair of carry-chain LUTs with shared inputs, independently from all others. Our heuristic works for different architectures and configurations of the LUTs, since the user should provide as parameters the size for the carry-chain LUTs, $\texttt{LUT}_\texttt{a}$ and $\texttt{LUT}_\texttt{b}$, as well as the number of shared inputs between them.

The method gets as an input a circuit description in BLIF format in which the full-adder logic of the carry chains is defined using hard-adder primitives. Since these primitives are represented as black boxes in the AIG, the full-adder inputs/outputs represent additional primary outputs/inputs. However, (1) the design might have additional modules that are also represented with black boxes (such as multipliers and RAM blocks), and (2) each signal appears only once in the list of AIG outputs even if it is an input to multiple full adders. Thus, first, we identify, out of all the AIG outputs, the ones that are full-adder inputs. To do so, we parse the input file to match and create pairs of AIG primary outputs, such that the two outputs from the pair are the $\texttt{A}$ and $\texttt{B}$ inputs of one full adder. Next, we compute all k-input cuts for the pairs, where $k = max(a, b)$, and a and b are the user-defined number of inputs for $\texttt{LUT}_\texttt{a}$ and $\texttt{LUT}_\texttt{b}$, respectively. For each pair, the two cuts with the highest cumulative gain are selected per output. The *gain* of a cut is the number of nodes covered by the cut that do not need to be duplicated. Duplication may happen because the outputs of $\texttt{LUT}_\texttt{a}$ and $\texttt{LUT}_\texttt{b}$ are only available to the chain logic: thus, if a node x, covered by a selected cut, has a fan-out to another node y that is not covered by the cut, then x should be duplicated into the remaining network together with the covered nodes from its fanin cone. For each selected pair of cuts, $\texttt{cut}_\texttt{a}$ and $\texttt{cut}_\texttt{b}$, we guarantee that it can be mapped on the logic block by verifying that

$$nPI(\texttt{cut}_\texttt{a}) + nPI(\texttt{cut}_\texttt{b}) - nShared(\texttt{cut}_\texttt{a}, \texttt{cut}_\texttt{b}) \leq$$
$$nPI(\texttt{LUT}_\texttt{a}) + nPI(\texttt{LUT}_\texttt{b}) - nShared(\texttt{LUT}_\texttt{a}, \texttt{LUT}_\texttt{b}),$$

where $nPI(x)$ presents the number of inputs of x and $nShared(x, y)$ the number of shared inputs of x and y. Once the cuts are selected, we should generate a mapping for the nodes that are not mapped on the carry-chain LUTs. Thus, a new AIG is created, representing the remaining network that consists of the original AIG without the nodes covered by the selected cuts. In this AIG, the inputs of the selected cuts become primary outputs. Then, since the new AIG differs from the original, we can optionally optimise it. Finally, we perform regular mapping on LUTs in normal mode. Figure 4 shows the original network with one selected 2-input cut (left) which is duplicated without the cut into a remaining network (middle) for regular mapping on LUTs in normal mode, while the cut is implemented with the logic block (right).

The method outputs a BLIF file with the new mapping of the circuit, which consists of the selected cuts for the pairs and the cuts selected by the regular mapping, together with the modules from the original circuit. This BLIF file can be used to check equivalence to the initial circuit description.

V. Experimental Setup

In this section we discuss the architecture used in our experiments as well as the modifications made to VTR in order to implement our algorithms.

A. Architecture

The architecture used for all our experiments is based on the FPGA architecture provided with the VTR tool and modelled in a 40nm CMOS technology. The cluster architecture that introduces carry chains to the VTR flow is a simplistic version of the Altera Stratix V FPGA that facilitates the description of the hard adders and the carry chain. However, such simplification uses inefficiently the logic and, by overconstraining the inputs, underutilises the LUTs.

Our modelled cluster consists of 10 ALMs, where each ALM has eight inputs and two outputs. An input crossbar distributes the 52 global inputs and the 20 feedback connections to the ALMs. To simplify the packing process, the crossbar is fully populated (a *full crossbar*). The ALM can operate in two different modes: normal and arithmetic. Usually, the ALM is used in *normal mode*, and consists of a 6-input fracturable LUT with optional registers. However, to implement arithmetic operations, the ALM can also operate in *arithmetic mode*, using the existing full adders along with its dedicated carry-chain interconnect. The ALM's arithmetic mode is modelled as already explained in Section II-C and presented in Figure 2b.

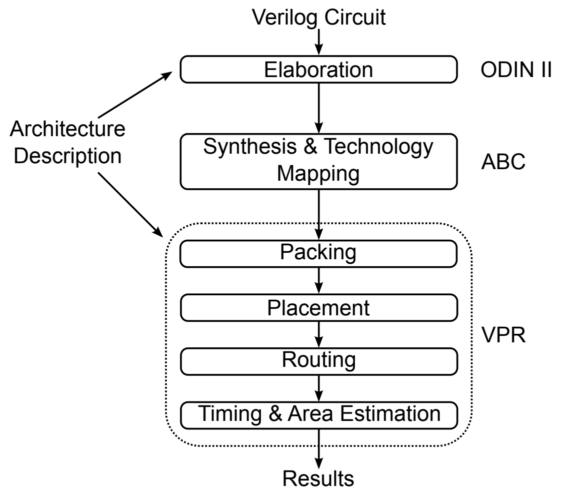


Fig. 5: The VTR tool flow used in our experimental setup.

The routing parameters of the architecture are kept the same as the ones provided with the VTR tool and used in previous research on similar architectures [5]. The DSP blocks and single/dual-port memory blocks are also included and placed using intervals of eight columns of logic clusters.

B. CAD Flow

We use the VTR 7.0 CAD flow [5] with some modifications to improve its performance on the targeted architecture. The flow, shown in Figure 5, takes as input a Verilog description of the circuit along with a description of the FPGA architecture and starts by performing elaboration and logic optimisation using ODIN-II. Then ABC [2] performs technology mapping, providing the packer with a netlist of atoms to pack into clusters, and then place and route them before estimating the critical path delay and area of the circuit.

In the baseline experiments, ODIN-II infers adders from the RTL operators used, as explained in Section II-B. ABC is an open-source tool, designed for logic synthesis, technology mapping, and formal verification for logic circuits. It is part of the VTR flow, used mainly for technology mapping while performing some logic optimisations. However, in VTR, the provided version of ABC is old and not maintained. ABC, as an independent tool, has come a long way since its VTR version: it consists of new and more efficient algorithms for both logic optimisation and mapping. Because our implemented mapping algorithms are based on the latest version of ABC, we modified the flow to use the same version of ABC for both the reference experiments and ours. Furthermore, our algorithms are implemented as additional commands in ABC, so they were easily integrated in the flow and used right before the technology mapping phase.

VPR, used for packing, placement, and routing, takes, as input, the mapped netlist from ABC and the description file of the architecture such as the one described in Section V-A. It takes particular care of the chains and ensures that all the adders of one chain are connected in the right order, using the dedicated carry channels. All carry-chain related connections and blocks are annotated in the architecture file with a special *packing pattern* to help the packer identify and group these blocks together. However, despite this special treatment, we realised during our experiments that the packer still under-utilises the carry-chain LUTs. Even if the opportunity of using these LUTs exists, the packer misses it in most cases and ends up using these LUTs as buffers.

To evaluate the efficiency of the new techniques, we use the benchmarks from the VTR 7.0 release, which are known to have arithmetic operation. Although our algorithms have reasonable run times, which range from 50 milliseconds up to 3 minutes depending on the benchmark's size, we omit the benchmarks that are too large to finish the complete flow in a reasonable delay. In all experiments, we first run placement and routing without any constraint on the channel width, and then we augment the used channel width by 30% and repeat the experiment. All reported results are averaged over three runs with different placement seeds.

VI. Experimental Results

To test each of the algorithms presented in the previous sections, we run both the reference flow and proposed flow on different VTR benchmarks. We present, in this section, the results of these experiments while analysing the strengths and shortcomings of each approach.

A. Gate-Level Arithmetic Detection

As discussed in Section II-B, ODIN-II instantiates hard adder primitives for VTR only if the adders are visible as high-level operators. We also verified that ODIN-II overlooks adders for gate-level circuits by processing various gate-level implementations of 8-, 16- and 32-bit adders, and indeed for such gate-level structures no hard adders are instantiated.

On the other hand, with the method described in Section III we discover 6% more full adders than ODIN-II; however, the structure of the chains among the two versions differs. In the chains, we can distinguish three types of nodes: First, there are full adders for starting and ending chains, which use only the carry and sum output, respectively—that is a particularity of how adders are built with chains, and is common to both us and high-level detection methods. Among the full adders which are in the middle of chains, one can distinguish (1) true 3-input full adders, where the two carry-chain LUTs either implement some useful logic function or simply propagate a signal, and (2) full adders with at least one constant input implemented in the carry-chain LUTs. ODIN-II, by translating high-level addition-like operands into hard adders, misses opportunities for elementary logic optimisations (constant propagation, logic simplification), and thus 17% of the generated full adders have constant inputs—that is, the implemented functionality should be simpler than that of full adder. On the contrary, since our method works with a gate-level circuit, which is not constrained by the adders' logic, basic logic optimisations can be performed on the adders, but also across the adder. Moreover, to truly benefit from the chain logic, the composed

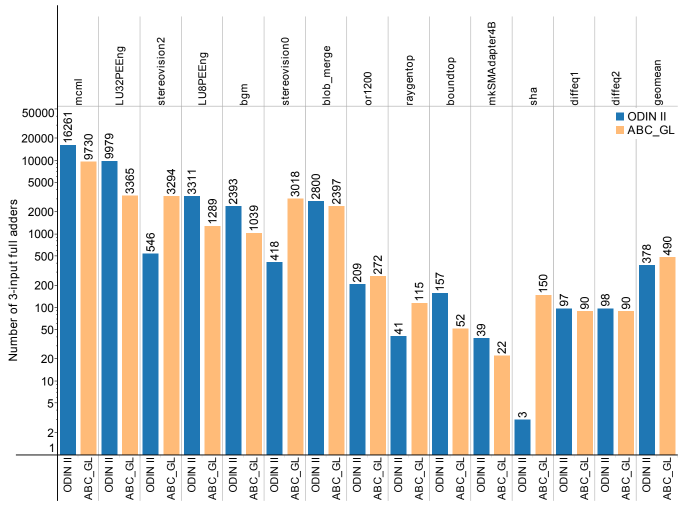


Fig. 6: Number of 3-input full adders shown on logarithmic scale when the minimum chain length is four. By recovering full adders from the gate level description, we increase the number of 3-input full adders by 30% on average.

TABLE I: Geomean of the different types of full adders generated with different methods. For the gate-level circuits, before performing the chain detection, we run three different logic optimisations in ABC: LS1 which performs a basic logic optimisation while generating an AIG with `strash`, LS2 and LS3 which are the well-known optimisation techniques `strash; dc2` or `resyn; resyn2`, respectively.

	ODIN-II	ABC (gate-level)		
	(RTL)	LS1	LS2	LS3
Start/end of chains	51	60	42	36
Middle real 3-input	378	489	238	187
Middle with a constant	98	0	0	0

chains contain full adders without any constant input. Thus, as Figure 6 shows, we generate 30% more true 3-input full adders on average compared to ODIN-II, when the minimum chain length for both methods is set to four. Consequently, as Figure 7 shows, this mainly improves area (up to 15%) for almost no change in delay on average, when the complete VTR flow is executed.

The gate-level descriptions of the benchmarks used for the experiments are obtained by elaborating the Verilog files with ODIN-II while disabling the generation of hard adders. Then, before calling the algorithm for the chain detection, we only run the `strash` command from ABC to create the AIG and perform basic logic optimisations. However, since our algorithm is based on structural matching, to remove the dependency of the structure generated by ODIN-II, we also compare the number of found full adders when the circuit is preprocessed by some heavier logic synthesis techniques. From the results we can conclude that even if nontrivial logic optimisation commands are run in ABC before chain detection (for instance, `strash; dc2` or `resyn; resyn2`), a significant fraction of the full adders that would be found on

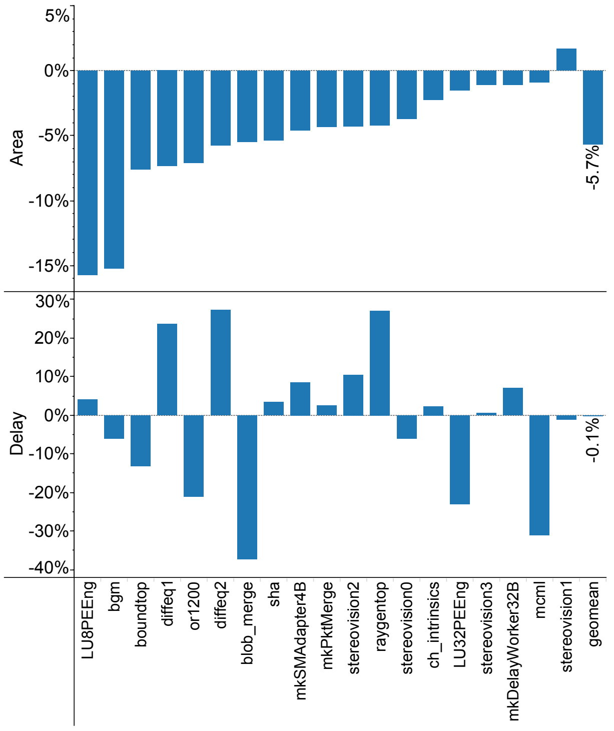


Fig. 7: Comparison of area and delay of the implementation with carry chains recovered from the gate-level description versus the one with carry chains generated by ODIN-II. By generating carry chains composed only of 3-input full adders, we improve the area by about 6%.

the unprocessed input netlist are still found after tangible logic restructuring: the strength of detecting adders in the synthesis engine is that, as shown in Table I, up to 60% of the adders are still detected despite ABC's heavy optimisations.

B. Mapping on the Carry Chain LUTs

Starting from the observation that the carry-chain LUTs are underutilised, mainly as buffers, the technique presented in Section IV increases the chances of using these LUTs to implement logic. Considering only the 2-input and 3-input functions selected for the carry-chain LUTs by the premapping algorithm (i.e. excluding the inverters and buffers which, being single-input single-out functions can, by default, be placed on the LUTs), the new mapping technique packs twice the number of functions on the carry-chain LUTs than the reference flow. Theoretically, this special mapping should (1) reduce the logic delay, since the signal would traverse one less LUT, and at the same time (2) reduce the area, because the carry-chain LUT implements what was previously implemented outside. To test this approach, the reference VTR tool flow, which identifies arithmetic operations through ODIN-II, is augmented with the premapping selection technique. Figure 8 shows the relative delay and area, with respect to the reference flow. The results differ from one benchmark to another but most of those benchmarks land in the second and fourth quadrant of the diagram, presenting a trade off of delay and area with Pareto

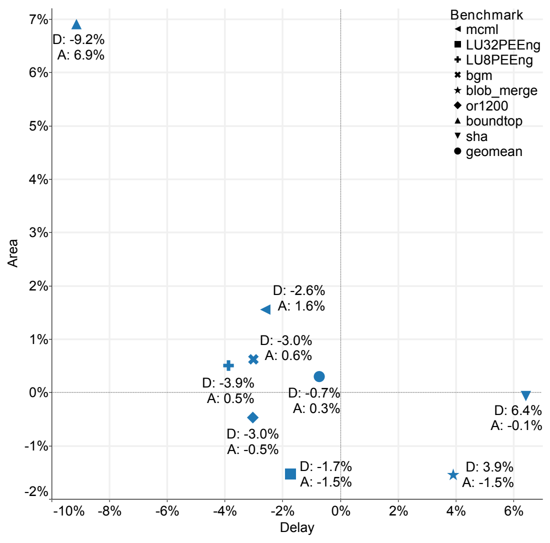


Fig. 8: Area and delay results after premapping logic in the carry-chain LUTs, compared to the regular VTR flow.

optimal solutions. A couple of benchmarks land in the third quadrant, improving both delay and area, even if marginally.

Although encouraging, the results remain minimalistic, far from the intuitive improvement expected at first. A closer look at the packed netlist helped us identify a potential reason behind these results. Table II shows the number of cuts/LUTs selected using the premapping phase along with the percentage of correctly packed, out of all found carry-chain LUTs. So, despite the identification and selection of the carry-chain LUTs at the mapping stage, the packer fails to associate them with their respective adders and, as such, ends up wasting the carry-chain LUTs as buffers while placing their targeted functions in different logic blocks. Due to this limitation of the packer only 54% of the selected cuts, on average, are correctly packed, with a minimum of 14%. This hits on both the delay and area results of the circuits, but no direct association can be easily concluded.

VII. Related Work

Carry-chain logic has been widely studied since early 90s when first introduced by Hsieh et al. [4] in the Xilinx 4000 FPGAs. Carry-chain logic was introduced in the FPGA fabric to facilitate the realisation of fast ripple-carry adders. Many research efforts studied variations on the carry-chain architecture to help in the implementation of arithmetic operators other than adders. On the other hand, fewer researchers addressed the problem of automatic recognition of carry-chain logic in the RTL specification. As mentioned, tools rely on users explicitly writing additions/subtractions using RTL primitives (‘$+$’ and ‘$-$’ operators) or using predefined adder macro to recognise carry-chain logic. However, when the arithmetic operations are described at gate-level, state-of-the-art tools often miss opportunities to use carry chains, even for circuits rich in adder logic composed of 3-input majority functions and 3-input XOR gates, for which using the carry chains could reduce both area and delay. Frederick and Somani [3] proposed a mapping algorithm that exploits carry chains for general circuits, proving that carry chains have the potential to improve the delay of general-purpose circuits and not only arithmetic circuits. However, this technique is only applicable to carry-select chains, which are no longer present in modern FPGAs, whereas ours detects only ripple-carry chains but can map the resulting adders on any hardened adder structure, as discussed in Section III.

TABLE II: Percentage of the packed LUTs out of the ones specially mapped to be placed in front of the hard adders. Although more opportunities are found to map logic functions on the adders’ LUTs, the packer often fails to associate them with their respective adders.

Benchmark	#Mapped LUTs	%Packed LUTs
mcml	525	14%
LU32PEEng	512	91%
LU8PEEng	128	79%
bgm	72	74%
blob_merge	22	100%
or1200	4	25%
boundtop	26	92%
sha	32	44%
Geomean		54%

Another aspect overlooked in carry-chain mapping is the utilisation of the LUTs which are in front of the hardened adders and which, at least in open source tools and published literature, are mostly used just to pass the inputs of the carry chain without implementing any useful logic. Luu et al. [5] modified the VTR packer to pack cells (e.g., flip flops, LUTs) that have transitive connectivity (i.e., connected through the carry chain) into the same logic clusters to avoid undesirable packing results. While they do try to fill the carry-chain LUTs, their approach rarely finds such opportunities since due to the position in the flow where they look for them (packing as opposed to mapping). Our technique discussed in Section VI-B is able to double these opportunities by searching for it at an earlier stage. In the same paper, the authors also evaluated the necessity to map adders on carry chains by computing an empirical threshold of the adder width. They found that adders with operands smaller than 12 bits are better implemented using LUTs rather than carry chains, but this is orthogonal to our goals and our techniques can equally well implement detected chains in hardened adders only above some length (in fact, by taking the decision during the synthesis process, we could have a better ability to take a decision based on the circuit topology rather than on a coarse heuristic, but we have not yet pursued this venue of optimisation).

Preußer and Spallek [11] proposed an approach that uses carry chains for general logic implementation. It uses a carry chain node to map a $(k+1)$-cut to a k-LUT. In their carry-chain mapping algorithm, they search for cuts that cover the LUT and the carry-chain node in the same logic block, and this places a tighter constraint than in our approach on the parts of the circuit which can be mapped there. The benefit of their approach is that it naturally utilises the carry-chain LUTs.

Our approach is superior, for adder-like structures, in two aspects; firstly, we decouple carry-chain mapping from carry-chain LUT filling, and this relaxes the constraints and increases the mapping possibilities. Secondly, we only map on the hardened adders cuts that use both the sum and carry-out pins in the carry chain to address, among others, adder structures, whereas, due to their focus on general logic, Preußer and Spallek miss this opportunity.

A more general technique for optimising the routing wire utilisation of a given circuit by replacing some programmable interconnect with nonroutable carry-chain connections was proposed by Parandeh-Afshar et al. [10]. However, this is a post technology mapping technique and, thus, the possibility to form a chain is limited by the availability of mapped logic functions that, per chance, can be re-mapped on the chain. As a result, some opportunities for using chains will be lost due to inappropriate partitioning of the circuit logic into mapped logic functions. Moreover, the predefined mapping does not allow the authors to use the second output of the chain that is freely available. The heuristic achieves a 9% average reduction of the routing wires but the circuit delay is increased by 3%. Zgheib and Ouaiss [13] extended this heuristic with Boolean matching and decomposition techniques to achieve up to 24% reduction in routing wires. In contrast, our premapping approach discovers additional opportunities to use complete carry-chain logic by restructuring the gate-level description. Furthermore, our technique achieves, in most benchmarks, a sizable reduction in delay on top of the reduction in area (and not only of routing resources, then).

VIII. Conclusion

Carry chains are a standard feature of today's FPGAs: they are very effective in improving speed and area of important arithmetic components (most notably adders), to the extent that they even contradict classic tenets of arithmetic logic design (e.g., on an FPGA, a ripple carry adder is the fastest adder). However, existing tools are limited to inferring carry chains only when high-level descriptions are available. In this paper we extract carry chains from gate-level circuits, ignoring high-level information about the implemented arithmetic operators. Differently from prior attempts, we chose a generic approach that does not depend on the specificities of one implementation of carry chains and we have tackled the problem earlier in the flow than others did—that is, during synthesis and prior to LUT mapping. Additionally, we have looked into a way to use as proficiently as possible the LUTs that invariably practical architectures have in front of carry chains.

On the carry-chain detection front, we are quite successful: On most benchmarks we identify a decent fraction of the full-adder structures that others identify at high-level and, arguably, we identify those that are most relevant (that is, those which would not be significantly improved by logically restructuring the function). More interestingly, in a number of cases, we identify very significantly more opportunities than available at the high-level; this is due to XOR-based structures that are not immediately derived from additions and subtractions, such as cryptographic primitives or compressors. We are less able to show the advantages of exploiting the LUTs before the carry chains, which a tool like VTR almost completely ignores: although our results indicate that using these LUTs is never a bad choice in a Pareto sense (that is, none of our results is Pareto dominated by the reference implementation), only in a few cases our result is better in both area and timing as we would generally expect. It appears that we are victims of a shortcoming in the VTR packer which most often disobeys our indications on what functionality to place in these LUTs.

It is high time to look into techniques to use hardened logic such as carry chains at the level of logic synthesis: results could be much better than by "dumbly" detecting plus signs in RTL (as our results show in a number of cases) and the techniques could be readily extended to other hardened structures (such as logic-chains that some authors have proposed in the past). Our improved version of ABC is available to other researchers so as to enable further research on this topic.

References

[1] Altera Corporation. *Stratix V Device Handbook, vols. 1 and 2*, 2014. `http://www.altera.com/literature/`.

[2] Berkeley Logic Synthesis and Verification Group, Berkeley, Calif. *ABC: A System for Sequential Synthesis and Verification*, June 2015. Release 50630, `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[3] M. T. Frederick and A. K. Somani. Beyond the arithmetic constraint: Depth-optimal mapping of logic chains in LUT-based FPGAs. In *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 37–46, New York, Feb. 2008. ACM.

[4] H.-C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin, L. Tinkey, and R. Kanazawa. Third-generation architecture boosts speed and density of field-programmable gate arrays. In *Proceedings of the IEEE Conference on Custom Integrated Circuits*, pages 31–8, Boston, Mass., May 1990.

[5] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6:1–6:30, June 2014.

[6] J. Luu, C. McCullough, S. Wang, S. Huda, B. Yan, C. Chiasson, K. B. Kent, J. Anderson, J. Rose, and V. Betz. On hard adders and carry chains in FPGAs. In *Proceedings of the 22nd IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 52–9, Boston, Mass., May 2014.

[7] A. Mishchenko, S. Chatterjee, and R. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Erl technical report, EECS Dept., UC Berkeley, Berkeley, Calif., 2005.

[8] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Proceedings of the 43rd Design Automation Conference*, pages 532–36, San Francisco, Calif., July 2006.

[9] H. Parandeh-Afshar, P. Brisk, and P. Ienne. Exploiting fast carry-chains of FPGAs for designing compressor trees. In *Proceedings of the 19th International Conference on Field-Programmable Logic and Applications*, pages 242–49, Prague, Aug. 2009.

[10] H. Parandeh-Afshar, G. Zgheib, P. Brisk, and P. Ienne. Routing wire optimization through generic synthesis on FPGA carry chains. In *Proceedings of the 20th International Workshop on Logic and Synthesis*, San Diego, Calif., June 2011.

[11] T. B. Preußer and R. G. Spallek. Enhancing FPGA device capabilities by the automatic logic mapping to additive carry chains. In *Proceedings of the 20th International Conference on Field-Programmable Logic and Applications*, pages 318–25, Milano, Aug. 2010.

[12] Xilinx Inc. *7 Series FPGAs Configurable Logic Block*, 2014. Version 1.7, `http://www.xilinx.com/`.

[13] G. Zgheib and I. Ouaiss. Enhanced technology mapping for FPGAs with exploration of cell configurations. *Journal of Circuits, Systems and Computers*, 24(3), Mar. 2015.