

Chapter 2

A Fabric Component Based Approach to the Architecture and Design Automation of High-Performance Integer Arithmetic Circuits on FPGA

Ayan Palchaudhuri and Rajat Subhra Chakraborty

Abstract FPGA-specific primitive instantiation is an efficient approach for design optimization to effectively utilize the native hardware primitives as building blocks. Placement steps also need to be constrained and controlled to improve the circuit critical path delay. Here, the authors present optimized implementations of certain arithmetic circuits and pseudorandom sequence generator circuits to indicate the superior performance scalability achieved using the proposed design methodology in comparison to circuits of identical functionality realized using other existing FPGA CAD tools or design methodologies. The Hardware Description Language specifications as well as the placement constraints can be automatically generated. A GUI-based CAD tool has been developed that is integrated with the Xilinx Integrated Software Environment for design automation of circuits from user specifications.

2.1 Introduction

With a significant increase in circuit complexity for Field Programmable Gate Array (FPGA)-based designs, even the most sophisticated Computer Aided Design (CAD) tools often result in circuit implementations with unsatisfactory performance and resource requirements owing to their inability to optimally exploit the underlying FPGA architecture. Also, the CAD tool is unable to place the technology mapped sub-circuits at the desired locations on the FPGA fabric. To overcome these shortcomings, a designer needs to identify the basic building blocks of the circuit, and make an effort to optimally construct them from the hardware primitives available on the FPGA and ensure their proper placement. Hence, implementations derived through the standard automatic logic synthesis-based design flow starting with the behavioral or Register Transfer Level (RTL) Hardware Description Language (HDL) of the circuit can be

A. Palchaudhuri (✉) · R.S. Chakraborty
Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur, Kharagpur 721302, West Bengal, India
e-mail: ayanpalchaudhuri@gmail.com

R.S. Chakraborty
e-mail: rschakraborty@cse.iitkgp.ernet.in

outperformed by more low-level “custom” design techniques. This chapter discusses certain FPGA-specific design techniques which needs to be adopted for optimal realization of high performance circuits and presents a relevant case study.

2.1.1 Overview of FPGA Design Philosophy

Modern FPGA CAD tools facilitate the automatic mapping of binary addition logic and homogeneous wide AND and OR gates to carry-chain structures [1] to accelerate signal propagation. It can also infer *wide function multiplexers* native to an FPGA slice, thereby achieving speedup by avoiding switch-box routing to the extent possible. However, it fails to do all of this, as soon as the final carry outputs of individual slices [2] or Lookup Table (LUT) outputs are tapped out or registered to facilitate pipelining of the architecture. In such a scenario, the designer has to spell out special directives in the HDL of the design or in the associated “constraints files,” so that in the *packing* or *clustering* step (as known in the FPGA CAD literature [3]) of the FPGA design flow, the desired technology mapped circuit is efficiently “packed” into the available hardware resources.

Most current FPGA vendors allow direct instantiation of the available primitives in the HDL code [4], and also a “mixed” style of HDL coding, whereby high-level behavioral code is intermingled with relatively “low-level” structural code. Placement steps also need to be constrained and controlled, otherwise the technology mapped logic elements get unevenly distributed across the FPGA fabric, resulting in greater routing and critical path delay. Although most modern FPGA vendors provide special hardware IPs for common integer arithmetic operations that can be directly instantiated in the HDL code, we would demonstrate that we can do better by adopting careful design techniques.

With sufficient modularity in the circuit architecture, it becomes easy to automate the generation of the HDL and constraint files, which are themselves very regular in their grammar. Target FPGA-specific *primitive instantiation* is an effective approach for design optimization [5], and is often the only approach, or is simpler than rewriting the RTL code to coax the logic synthesis tool to infer the desired architectural components. In case the entire circuit is not amenable to primitive instantiation, the philosophy can be adopted to design those sub-circuits that are amenable, and contribute significantly to the critical path. The only disadvantage of using such a design methodology is that the design becomes less portable and harder to maintain. In spite of this, the methodology is very effective in practice, considering that (a) often the target FPGA platform is known before the circuit is designed, and, (b) FPGAs from a related family from the same vendor are often backward compatible regarding the design elements supported. For example, newer versions of FPGAs of the “Virtex” family from Xilinx are expected to support primitives supported in some older Virtex versions. Thus, the HDL code for instantiating primitives targeting the older versions, and the constraints file to control the placement, can be reused in the newer version after small tweaks, if necessary.

2.1.2 Existing FPGA CAD Tools

Xilinx IP Core Generator and *FloPoCo* are the most common FPGA CAD tools available. We shall now present the features, advantages, and limitations of each of these tools.

2.1.2.1 Xilinx IP Core Generator

The standard *Xilinx IP Core Generator* has a GUI-based utility through which synthesizable HDL code for common integer arithmetic circuits (both combinational and pipelined versions) can be automatically generated. User gives the parameter *latency* as input for pipelined architecture realizations. Although such HDL generated is functionally correct, it fails to give high performance when implemented, because the synthesis tool performs inefficient technology mapping and the inferred logic elements are scattered in an apparently random fashion across the FPGA fabric, thereby causing large routing delays. Xilinx also allows the direct instantiation of “Digital Signal Processing” (DSP) hardware macros in the HDL targeted for FPGAs, to implement certain common arithmetic operations, but they can be outperformed by the proposed circuits through maximal forward path pipelining at the cost of higher latency.

2.1.2.2 FloPoCo (Floating-Point Cores)

FloPoCo is an open-source C++ framework for generating arithmetic cores for FPGAs [6]. *FloPoCo* provides a command-line interface through which the user can input operator specifications, and the program generates the corresponding synthesizable HDL. The main features of *FloPoCo* as listed in [7] are as follows:

- Supports integer, fixed point, floating point, and *Logarithmic Number System* (LNS) arithmetic.
- Supports pipelining by allowing the user to specify the desired operating frequency.
- Allows the user to specify the target FPGA implementation platform, and generates synthesizable HDL code optimized for that target platform since *FloPoCo* can perform target platform specific pipelining. Its frequency-directed pipelining paradigm takes into consideration the timing information about the target FPGA [7].

However, detailed experimentation with the latest released version of *FloPoCo* (v 2.5.0) [6], and implementation and characterization of the integer arithmetic circuit descriptions generated by it indicate certain potential drawbacks:

- *FloPoCo* only generates pure behavioral HDL code which cannot correctly infer the desired hardware primitives of the target FPGA platform and also has no control over the inference and placement of logic blocks on the FPGA fabric. This makes the post-synthesis performance of the circuit worse than the user-specified

target frequency. Thus, *FloPoCo* provides no guarantee that the target frequency specified would be met in the final implementation.

- *FloPoCo* at times create very deep pipelines apparently to meet input frequency constraints, but *post place and route* implementations do not guarantee that the delay constraints are met.
- Pipelining behavior of *FloPoCo* is very inconsistent. It was observed that for integer adder circuit implementations, *FloPoCo* creates very deep pipelines, whereas it creates fairly unbalanced and irregular pipelines for integer dual subtractor implementations, where each of the pipeline stages have different complexities. At the same time, *FloPoCo* completely fails to pipeline integer multiplier circuits. Our observations about these inconsistencies in the pipelining behavior of the current version of *FloPoCo* have been concurred with by the creators of *FloPoCo* through personal correspondence. They have acknowledged that a bug exists in their program, which they have filed and would probably be taken care of in future releases.

The important approach to note is that most or all of the existing options for automatic generation of arithmetic circuit cores targeting FPGAs are agnostic of the “low-level” architecture of the target FPGA platform. Consequently, they can be predicted to be unable to take advantage of the hardware primitives, and to generate the most optimal circuit descriptions for the target FPGA.

2.2 Architecture of Target FPGA Platform

The Configurable Logic Blocks (CLBs) of FPGAs are the main logic resources for implementing sequential as well as combinatorial circuits. A typical CLB of Virtex-5 FPGA contains 2 “slices,” with each slice (called a “SLICEL” or “SLICEM”) comprising of four 6-input LUTs, four flip-flops (FFs), three wide function multiplexers, and a length-4 carry chain comprising of multiplexers and XOR gates [8, 9] as shown in Fig. 2.1.

For both Virtex-5 and Virtex-6 FPGAs, the 6-input LUT can also be configured as a dual output LUT which can configure a 5 (or less)-input, 2-output logic function with shared inputs, thereby reducing the requirement in the number of LUTs from two to one for certain logic expressions. The LUTs present in SLICEL can implement arbitrary combinational logic, whereas the LUTs in SLICEM can additionally implement *distributed RAM* elements [9]. The carry chain represents the fast carry propagation logic and the LUTs in the slice can be optionally connected to the carry chain via dedicated routes to implement complex logic functionality [4].

Each storage element or FF present in the slice can be controlled using the set, reset, clock, and clock enable signals. Virtex-6 slice architecture is quite similar to Virtex-5 slice architecture, other than the fact that it offers four additional FFs per slice but each FF has one control signal less, i.e., it does not have independent set and reset pins compared to Virtex-5 architecture [10].

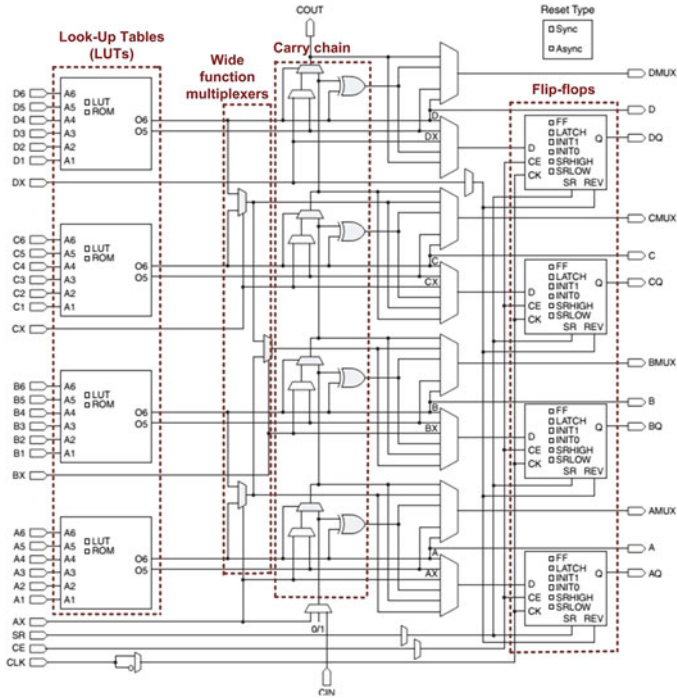


Fig. 2.1 Xilinx Virtex-5 slice architecture [9]

2.3 A Fabric Component-Based Design Approach for High-Performance Integer Arithmetic Circuits

Implementation of highly optimized arithmetic circuits targeted toward a specific FPGA family continue to remain a challenging problem as many fast arithmetic circuits proposed over the decades may not be amenable to a very optimized implementation. Often, the logic synthesis tools are unable to infer the desired native circuit components from the input HDL, as they explore only a small design space close to the input architectural description [11]. The logic synthesis algorithms are also unable to apply the logic identities and perform appropriate algebraic factoring and sub-expression sharing in many cases, especially when intermediate signals are tapped out [2] or registered to facilitate pipelining of the architecture. It is in general a nontrivial computational problem to decompose the Boolean equations describing the implemented circuit, to forms such that the sub-expressions can be mapped easily and efficiently to the fabric primitives on the target FPGA. It helps if the designer manipulates the Boolean equations *a priori* in the HDL, to forms that can be optimally mapped to the native target architecture by the CAD software.

A previous related work [12] had reported area requirements (for LUT-based FPGAs) in terms of the total number of k -input LUTs required to map a function of x variables, as

$$lut(x) = \begin{cases} 0 & \text{if } x \leq 1 \\ 1 & \text{if } 1 < x \leq k \\ \lfloor \frac{x-k}{k-1} \rfloor + 2 & \text{if } x > k \text{ and } (k-1) \nmid (x-k) \\ \frac{x-k}{k-1} + 1 & \text{if } x > k \text{ and } (k-1) \mid (x-k) \end{cases} \quad (2.1)$$

The LUT estimation is based on the fact that LUTs are perhaps the principal components for implementing combinational logic in FPGAs, where combinational logic blocks with higher number of inputs are expected to be implemented by a cascade of LUTs, with permitted amount of parallel processing. However, for efficient designs, the designer must explore the additional logical capabilities that the LUTs of modern day FPGAs provide. In addition, the wide-function multiplexers and the carry chains can significantly reduce LUT requirements. The above closed-form expression in (2.1) for hardware estimation thereby suffers from the following limitations:

- It assumes that all LUTs provide single outputs, whereas modern FPGAs from Xilinx provide dual-output LUTs that can significantly reduce hardware cost, provided the logic functions to be mapped satisfy certain criteria.
- Certain logic expressions can be factored appropriately to form sub-expressions or manipulated such that they can be compactly realized using LUTs, wide function multiplexers, and carry chains, which can provide multiple outputs out of a single slice. The closed-form expressions in (2.1) possibly hint at an approximate upper bound on the number of LUTs required.
- The number of slices spanned by the logic elements give a more accurate estimate of the hardware overhead in comparison to LUT count. The philosophy behind estimating the LUT requirements [12] may not reflect its actual implementation on FPGA. For example, let us consider an 8:1 multiplexer, which is essentially an 11-input 1-output function.

$$f(s_2, s_1, s_0, a, b, c, d, e, f, g, h) = s'_2 s'_1 s'_0 a + s'_2 s'_1 s_0 b + s'_2 s_1 s'_0 c + s'_2 s_1 s_0 d + s_2 s'_1 s'_0 e \\ + s_2 s'_1 s_0 f + s_2 s_1 s'_0 g + s_2 s_1 s_0 h \quad (2.2)$$

Going by (2.1), $lut(x) = 2$ for $k = 6$. This information indicates that the first six variables go as input to the first LUT and the remaining five variables along with the output of the first LUT go as input to the second LUT. However, on examining (2.2) closely, it can be observed that there is no possible way to decompose it to the following form comprising of two functions f_1 and f_2 , which could have actually realized it using two LUTs:

$$\begin{aligned}
 & f(s_2, s_1, s_0, a, b, c, d, e, f, g, h) \\
 & \quad \text{implemented using 1 LUT} \\
 & = f_2 \left(\underbrace{f_1(x_1, x_2, x_3, x_4, x_5, x_6)}_{\text{implemented using 1 LUT}}, x_7, x_8, x_9, x_{10}, x_{11} \right)
 \end{aligned}$$

where each x_i is distinct and can be any one of the variables of the function f .

2.3.1 Guidelines for High-Performance Realization

We list certain simple but useful guidelines for compact and high-performance realization of circuits on modern high-end FPGAs from Xilinx:

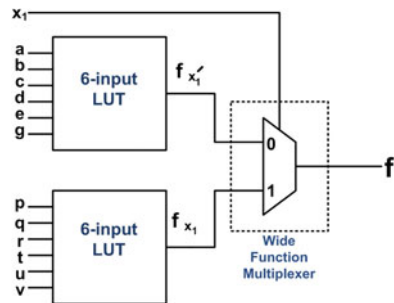
1. A 6-input LUT can implement any arbitrary combinational logic function y , having a maximum of six inputs and a single output, like $y = f(x_1, \dots, x_n)$, where $2 \leq n \leq 6$.
2. A 6-input LUT can implement any arbitrary five (or less)-input two-output function where each of the single-output functions may or may not have shared inputs. For example, consider two functions g and h , where $g = f(x_1, \dots, x_n)$ with $X = \{x_1, \dots, x_n\}$, and $h = f(y_1, \dots, y_m)$ with $Y = \{y_1, \dots, y_m\}$. Here, the sets X and Y are called the *support* [13] of the functions g and h . For packing g and h into a single LUT, either of the conditions must be satisfied:
 - $4 \leq |X| + |Y| \leq 5$; if $X \cap Y = \emptyset$ (i.e., g and h are *orthogonal* functions)
 - $2 \leq |X| + |Y| \leq 10$; if $X \cap Y \neq \emptyset$

where $|X|$ and $|Y|$ are the cardinality of the support of the functions X and Y .

3. Let f be a Boolean function of n variables ($8 \leq n \leq 13$) which can be represented in the following form (see Fig. 2.2):

$$f(i_1, i_2, \dots, i_n) = x'_1 f_{x'_1} + x_1 f_{x_1}$$

Fig. 2.2 Architecture mapping for Boolean logic that can be decomposed with respect to a single variable



Here, f_{x_1} and $f_{x'_1}$ are each six (or less)-input combinational functions that can be individually realized using one LUT each. The wide function multiplexer present in the same slice as that of the LUTs computes the final expression, as shown in Fig. 2.2. Equation (2.1) however evaluates to $lut(x) = 3$, where $x = x_{max} = 13$ (6×2 (two 6-input LUTs) + 1 (select line)) and $k = 6$. If there exists p functions of the form as in f , the design requires $\lceil p/2 \rceil$ slices, and $2p$ LUTs. An 8:1 multiplexer can be realized using this logic where the 6-input LUTs of Fig. 2.2 are configured as 4:1 multiplexers each (sharing the same select lines), and the wide function multiplexer selecting one of the LUT outputs.

4. Let f be a function of n variables ($17 \leq n \leq 26$) such that we can apply recursive decomposition twice on it as shown below:

$$\begin{aligned} f(i_1, i_2, \dots, i_n) &= x'_1 f_{x'_1} + x_1 f_{x_1} \\ &= x'_1 (x'_2 f_{x'_1 x'_2} + x_2 f_{x'_1 x_2}) + x_1 (x'_3 f_{x_1 x'_3} + x_3 f_{x_1 x_3}) \\ &= x'_1 x'_2 f_{x'_1 x'_2} + x'_1 x_2 f_{x'_1 x_2} + x_1 x'_3 f_{x_1 x'_3} + x_1 x_3 f_{x_1 x_3} \end{aligned}$$

Here, $f_{x'_1 x'_2}$, $f_{x'_1 x_2}$, $f_{x_1 x'_3}$ and $f_{x_1 x_3}$ are each 6 (or less)-input combinational functions that can individually be realized using one LUT each. Three wide function multiplexers present in the same slice as that of the LUTs computes the final expression as shown in Fig. 2.3. Equation (2.1) however evaluates to $lut(x) = 6$, where $x = x_{max} = 27$ (6×4 (four 6-input LUTs) + 3 (select lines)) and $k = 6$. If there exists p functions of the form as in f , the design requires p slices and $4p$ LUTs. A 16:1 multiplexer can thus be mapped in a single slice using four LUTs, and three wide function multiplexers.

5. Consider any expression R of the following form:

$$\begin{aligned} R &= a'b + a[X] = a'b + a[c'd + c(Y)] = a'b + a[c'd + c(e'f + e\{Z\})] \\ &= a'b + a[c'd + c(e'f + e\{g'h + gi\})] \end{aligned} \quad (2.3)$$

where $X = c'd + cY$, $Y = e'f + eZ$ and $Z = g'h + gi$. Here i is a single input variable, and for every member of the pair (a, b) , (c, d) , (e, f) and (g, h) , there can be either of the following possibilities: either both the members satisfy the criteria of being packed into a single LUT or the first member can be a six (or less)-input function and the second member can be a single variable. The above expression can be realized within a single slice using four LUTs and a carry chain, as shown in Fig. 2.4.

The Boolean logic (2.3) essentially represents a cascade of 2:1 multiplexer functions. Here, R can have a maximum of 29 (6×4 (four 6-input LUTs) + 4 inputs $v_{4:1}$ external to the logic slice + 1 external input to the bottom MUXCY of the carry chain) input variables. If the Boolean logic function to be realized is of the form such that the variable i in (2.3) can be substituted by another expression bearing a similar resemblance to (2.3), and in such a way, if a total of n such substitutions can be carried out only at the position of variable i , then the entire

expression can be realized using $(n + 1)$ slices and a maximum of $4(n + 1)$ LUTs. From (2.1), we obtain $lut(x) = 6$, where $x = 29$ and $k = 6$ for which a minimum FPGA area of two slices are required. However, with the help of carry-chain fabric, the entire architecture can be compacted within a single slice. For example, a wide 24-input AND gate can be realized by the function R to fit the form of (2.3) as shown below:

$$\begin{aligned} R &= a_1 a_2 \dots a_{23} a_{24} = \overline{a_{19} \dots a_{24}}.0 + (a_{19} \dots a_{24})[a_{18} a_{17} \dots a_1 a_0] \\ &= \overline{a_{19} \dots a_{24}}.0 + (a_{19} \dots a_{24})[\overline{a_{13} \dots a_{18}}.0 + (a_{13} \dots a_{18}) \\ &\quad [\overline{a_7 \dots a_{12}}.0 + (a_7 \dots a_{12})[\overline{a_1 \dots a_6}.0 + (a_1 \dots a_6).1]]]] \end{aligned}$$

Thus, going by Fig. 2.4, $a = a_{19} \dots a_{24}$, $c = a_{13} \dots a_{18}$, $e = a_7 \dots a_{12}$ and $g = a_1 \dots a_6$, $b = d = f = h = 0$, and $i = 1$. Hence each 6-input LUT realizes a 6-input AND gate and the outputs of the 6-input LUTs are AND-ed using the carry chain.

Example of a wide 24-input OR gate where the logic equation can be manipulated to fit the form of (2.3) as shown below:

$$\begin{aligned} R &= a_1 + a_2 + \dots + a_{23} + a_{24} = (a_{19} + \dots + a_{24}).1 + (\overline{a_{19} + \dots + a_{24}})[a_{18} + \dots + a_1] \\ &= (a_{19} + \dots + a_{24}).1 + (\overline{a_{19} + \dots + a_{24}})[(a_{13} + \dots + a_{18}).1 + (\overline{a_{13} + \dots + a_{18}}) \\ &\quad [(a_7 + \dots + a_{12}).1 + (\overline{a_7 + \dots + a_{12}})[(a_1 + \dots + a_6).1 + (\overline{a_1 + \dots + a_6}).0]]]] \end{aligned}$$

Thus, going by Fig. 2.4, $a = \overline{a_{19} + \dots + a_{24}}$, $c = \overline{a_{13} + \dots + a_{18}}$, $e = \overline{a_7 + \dots + a_{12}}$ and $g = \overline{a_1 + \dots + a_6}$, $b = d = f = h = 1$, and $i = 0$. Hence, each 6-input LUT realizes a 6-input NOR gate and the outputs of the 6-input LUTs are fed to the carry chain and a wide input OR gate is realized by following the absorption law $a + \overline{a}b = a + b$.

We discuss certain practical circuits where a wide input AND and OR gate are necessary for realization.

- Consider the design of a *priority encoder* which arbitrates among N units that are all requesting access to a shared resource. Access is to be granted to a single unit with highest priority decided by the LSB of the input. The corresponding logic equations can be described as

$$\begin{aligned} Y_0 &= N_0 \\ Y_i &= \underbrace{N_i \cdot \overline{N_{i-1}} \cdot \overline{N_{i-2}} \cdot \dots \cdot \overline{N_2} \cdot \overline{N_1}}_{\text{wide input AND gate}} \quad \text{if } i \geq 1 \end{aligned}$$

- An *incrementer* that adds 1 to an input word N can be described as

$$Y_0 = \overline{N_0}$$

$$Y_i = N_i \oplus \underbrace{(N_{i-1} \cdot N_{i-2} \cdot \dots \cdot N_1 \cdot N_0)}_{\text{wide input AND gate}} \quad \text{if } i \geq 1$$

- A *decrementer* that subtracts 1 from an input word N can be described as

$$Y_0 = \overline{N_0}$$

$$Y_i = N_i \odot \underbrace{(N_{i-1} + N_{i-2} + \dots + N_1 + N_0)}_{\text{wide input OR gate}} \quad \text{if } i \geq 1$$

- $K = A + B$ *Comparator* [15]

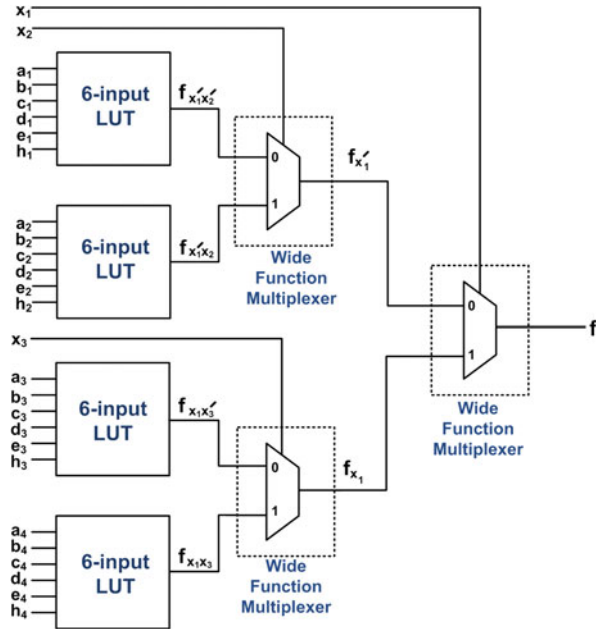
To design a circuit to detect $A + B = K$, the usual approach followed by the FPGA CAD tool is to infer an adder that adds inputs A and B , and feed the sum and input K to an equality comparator. However, to significantly reduce hardware and computational overhead, a methodology was proposed in [15]. The key observation is the fact that if A and B are known, the carry into each bit to make $K = A + B$ can be determined. Thus, it is sufficient to check adjacent pairs of bits to verify that the carry-out produced by the previous bit and the carry-in required by the current bit are both the same. The truth Table 2.1 shows the *required* and *generated* carries. The required carry-in cr_{i-1} for bit i and the generated carry-out cp_{i-1} for bit $i-1$ are obtained as $cr_{i-1} = A_i \oplus B_i \oplus K_i$ and $cp_{i-1} = (A_{i-1} \oplus B_{i-1})\overline{K_{i-1}} + A_{i-1} \cdot B_{i-1}$. Equality check for the i th bit position is performed using a single LUT as it can be computed using six distinct variables— A_i , B_i , K_i , A_{i-1} , B_{i-1} and K_{i-1} , by evaluating $EQ_i = cr_{i-1} \odot cp_{i-1}$. Final equality check is done using carry chain where all the outputs corresponding to equality checks at every bit position are AND-ed together.

$$EQ = \underbrace{EQ_j \cdot EQ_{j-1} \cdot \dots \cdot EQ_i \cdot \dots \cdot EQ_1 \cdot EQ_0}_{\text{wide input AND gate}}$$

Additionally, we can obtain the following outputs from the XORCY gates of the carry chain: $L = a \oplus X$, $M = c \oplus Y$, $N = e \oplus Z$ and $O = g \oplus i$. The XORCY gates can compute the sum bits of an adder $s_i = p_i \oplus c_i$ where p_i is computed using LUT; $p_i = a_i \oplus b_i$. The carry-out bit of each stage is computed using MUXCY of carry chain; $c_{i+1} = \overline{p_i}a_i + p_i c_i$ (see Sect. 2.4.1.1 for details). Thus, an n -bit adder can be realized using $\lceil n/4 \rceil$ slices and a maximum of n LUTs.

6. LUTs of SLICEM can be configured as shift registers which are typically implemented for *Linear Feedback Shift Register* (LFSR) circuits. Each SLICEM LUT can be configured as a variable 1–32 clock cycle shift register [4] whose length

Fig. 2.3 Architecture mapping for Boolean logic that can be decomposed with respect to two variables



can be fixed, static, or dynamically adjusted by controlling $A[4:0]$ as shown in Fig. 2.5. The LUT can be described as a 32:1 multiplexer with the five inputs serving as binary select lines, and the values programmed into the LUT serves as the data being selected. Such LUTs can be cascaded with FFs and LUTs of other SLICEMs to realize greater shift lengths. Presence of these special LUTs reduce FPGA resource utilization compared to implementations using FFs only. Since each SLICEM in a Virtex-5 FPGA contains four LUTs and four FFs, a shift register of length 1–132 can be realized in a single slice; and a 1–136 clock cycle register can be realized in a single slice for Virtex-6 FPGAs as it contains four additional FFs. For realization of a shift register of length n , we require $\lceil n/132 \rceil$ slices with a maximum of $4\lceil n/132 \rceil$ LUTs and $4\lceil n/132 \rceil$ FFs for Virtex-5 FPGA, and $\lceil n/136 \rceil$ slices with a maximum of $4\lceil n/136 \rceil$ LUTs and $8\lceil n/136 \rceil$ FFs for Virtex-6 FPGA.

2.4 Architecture of Arithmetic Circuits

We present the pipelined implementations of a few important and widely used circuits with controlled placement on the fabric logic such that the critical path delay can be significantly reduced and the throughput increased.

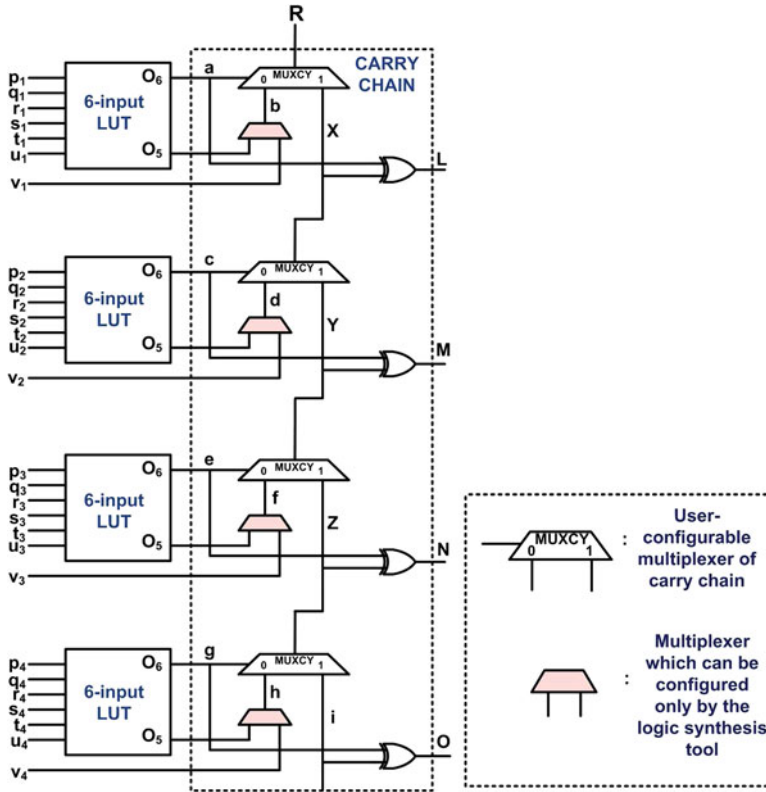


Fig. 2.4 Architecture mapping for Boolean logic that can exploit the carry chain

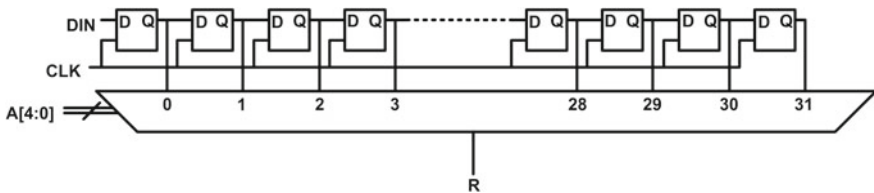


Fig. 2.5 Configuration of an LUT (of SLICEM) as a shift register

2.4.1 Integer Adder Architecture

2.4.1.1 Hybrid Ripple Carry Adder

A pipelined “hybrid ripple carry adder” (hybrid RCA), using the carry chain, LUTs and FFs available in a Xilinx slice, can be realized as shown in Fig. 2.6. The outputs of the “XORCY” gates provide the sum bits, whereas the output of each MUXCY

Table 2.1 Required and generated carries [14]

A_i	B_i	K_i	cr_{i-1} (required)	cp_i (produced)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	1

calculates the intermediate carries [16]. Latches can be inserted on the carry propagation path for pipelining the design. Thus, for an n -bit adder, $\lceil n/4 \rceil - 1$ pipeline stages are required. The LUTs compute the *propagate* function $p_i = a_i \oplus b_i$. Let $g_i = a_i b_i$ be the corresponding *generate* function. If c_i is the i th carry-in bit, the i -th sum bit can be calculated by XOR-ing the LUT and MUXCY outputs as

$$s_i = p_i \oplus c_i = a_i \oplus b_i \oplus c_i$$

The output of each MUXCY gate computes the i th carry-out as

$$\begin{aligned} c_{oi} &= g_i + p_i c_i = a_i b_i + (a_i \oplus b_i) c_i \\ &= (\overline{a_i} \overline{b_i} + a_i b_i) a_i + (a_i \oplus b_i) c_i = \overline{p_i} a_i + p_i c_i \end{aligned}$$

2.4.1.2 Xilinx DSP Slice-Based Adder

Adders can also be realized using embedded DSP48E slices in Virtex-5 FPGAs. Each slice can accept operands of width 48 bits. To realize adders with larger operand width $n(> 48)$, we require $\lceil n/48 \rceil$ DSP48E slices. Such designs can be pipelined by activating the pipeline registers internal to the slice. An $n(> 48)$ -bit pipelined DSP slice-based adder requires $\lceil n/48 \rceil - 1$ pipeline stages. For addition, the attributes “ALUMODE” and “OPMODE” have to be set to “0000” and “0001111” respectively [17]. Figure 2.7 illustrates the DSP adder architecture.

2.4.1.3 FloPoCo-based Adder

The behavioral synthesizable HDL generated by *FloPoCo* for Virtex-5 adders with the user-specified constraints of operating frequency remaining the same as obtained through our approach of constrained placement, shows that the circuit description

Fig. 2.6 Basic building block for pipelined implementation of hybrid ripple carry adder (RCA)

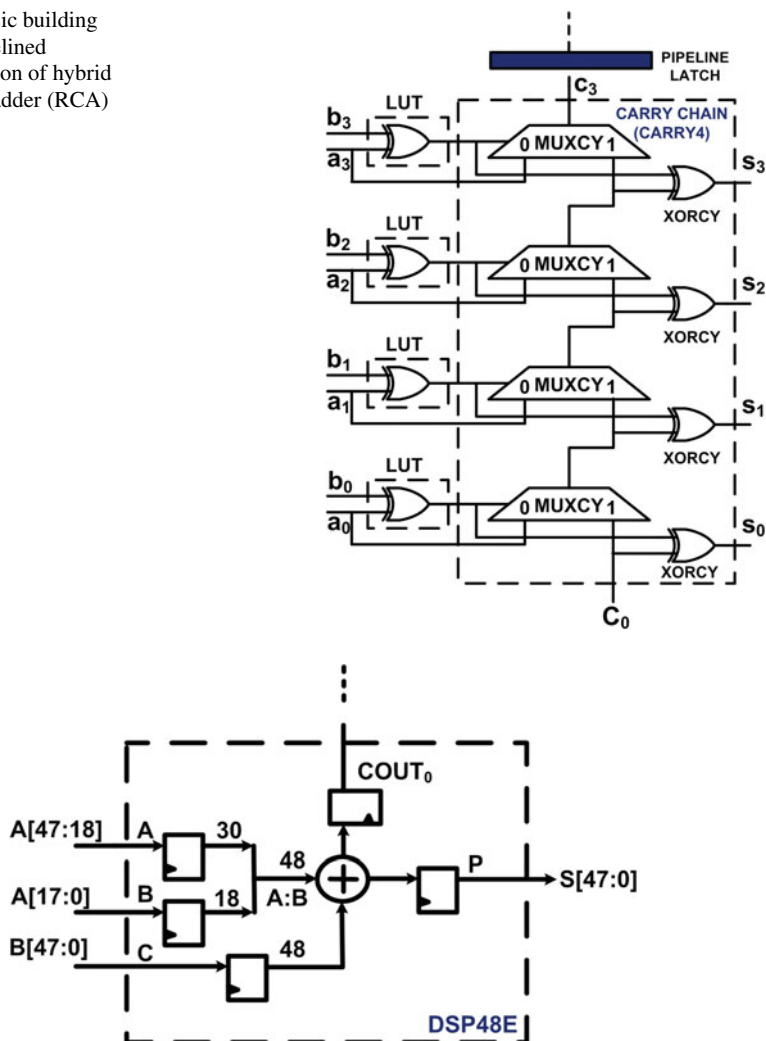


Fig. 2.7 Xilinx DSP slice-based adder [17]

generated is pipelined after every 2-bit addition as shown in Fig. 2.8. This in itself proves that the architecture cannot enjoy the benefit of utilizing a complete length-4 carry chain natively available in the Virtex-5 family. This results in a complete LUT-based implementation and the frequency constraints are not met.

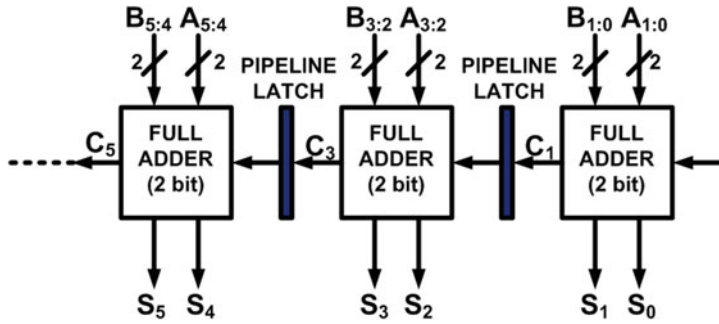


Fig. 2.8 Architecture for *FloPoCo* generated adder

2.4.1.4 Fast Carry Adder Using Carry-Lookahead Mechanism

The novel adder proposed in [16] had been designed using carry-lookahead mechanism by splitting an n -bit adder into two independent, identical portions L-RCA and H-RCA, each of which calculates $n/2$ sum bits (assuming n to be even). The H-RCA receives its carry input from a *fast carry generator* circuit. Both the L-RCA and H-RCA are architecturally identical to the pipelined implementation of the hybrid RCA shown in Fig. 2.6. The architecture of the fast adder architecture proposed in [16] for 64-bit operands is shown in Fig. 2.9. The reformulation of the carry-chain computation [18] has been addressed in the fast carry generator, where $P_{i:j}$ and

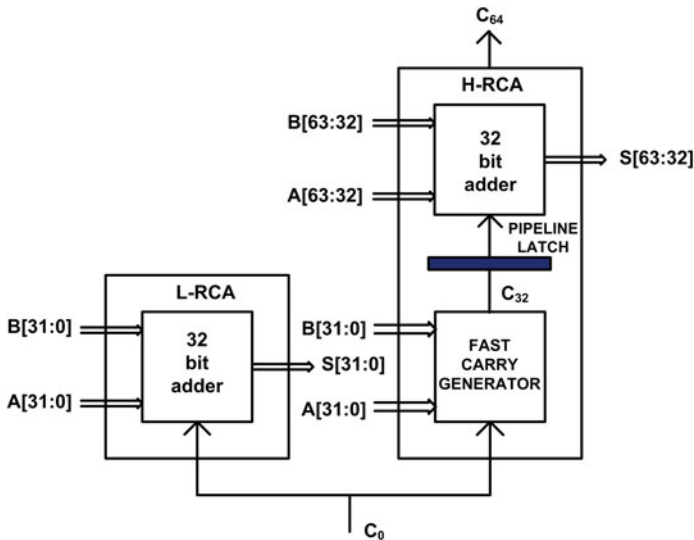


Fig. 2.9 Fast adder architecture proposed in [16]

$G_{i:j}$ denote the *group-propagated carry* and the *group-generated carry* functions, respectively, for a group of bit positions $i, i-1, \dots, j$ (with $i \geq j$) [19].

$$P_{i:j} = \begin{cases} P_i, & \text{if } i = j \\ P_i P_{i-1:j} & \text{if } i > j \end{cases} \quad (2.4)$$

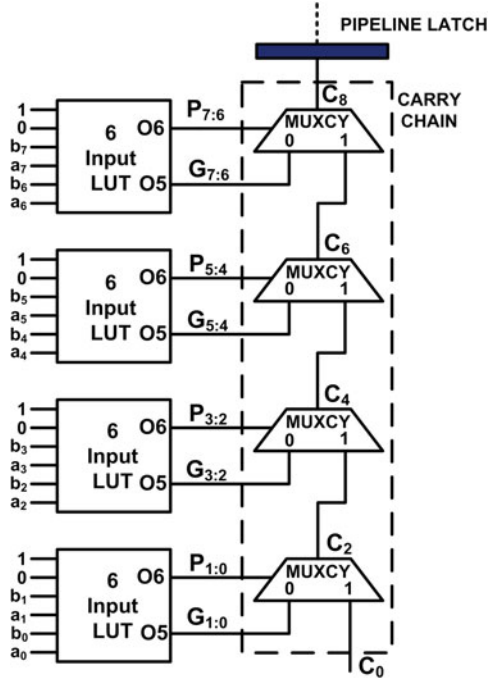
$$G_{i:j} = \begin{cases} G_i, & \text{if } i = j \\ G_i + P_i G_{i-1:j} & \text{if } i > j \end{cases} \quad (2.5)$$

where $P_i = a_i \oplus b_i$ and $G_i = a_i b_i$.

The recursive Eqs. (2.4)–(2.5) can be further generalized to $P_{i:j} = P_{i:m} P_{m-1:j}$ and $G_{i:j} = G_{i:m} + P_{i:m} G_{m-1:j}$ where $i \geq m \geq j+1$. For the m th bit position, ($i \geq m \geq j$), we have $c_m = G_{m-1:j} + P_{m-1:j} c_j$. In Fig. 2.10, $G_{i:j}$ and $P_{i:j}$ are calculated using 6-input LUTs, where $i = j+1$ and $m = i+1$. c_m are calculated using the carry chain.

Thus,

Fig. 2.10 Architecture for fast carry generator [16]



$$\begin{aligned}
P_{i:j} &= P_i P_j = (a_i \oplus b_i)(a_j \oplus b_j) \\
G_{i:j} &= G_i + P_i G_{i-1:j} = a_i b_i + (a_i \oplus b_i) a_j b_j \\
c_m &= G_{m-1:j} + P_{m-1:j} c_j = G_{m-1:m-2} + P_{m-1:m-2} c_{m-2} \\
&= G_{i:j} + P_{i:j} c_{m-2} = \overline{P_{i:j}} G_{i:j} + P_{i:j} c_{m-2}
\end{aligned}$$

Hence, c_m can be obtained from c_{m-2} using only a single multiplexer in the fast carry generator, which is in contrast to the hybrid RCA that computes c_m from c_{m-2} using two multiplexers of the carry chain. Hence, c_8 can be obtained from c_0 within a single slice, which is shown as follows where (2.6) assumes the same form as (2.3):

$$\begin{aligned}
c_8 &= \overline{P_{7:6}} G_{7:6} + P_{7:6} c_6 = \overline{P_{7:6}} G_{7:6} + P_{7:6} [\overline{P_{5:4}} G_{5:4} + P_{5:4} c_4] \\
&= \overline{P_{7:6}} G_{7:6} + P_{7:6} [\overline{P_{5:4}} G_{5:4} + P_{5:4} [\overline{P_{3:2}} G_{3:2} + P_{3:2} [\overline{P_{1:0}} G_{1:0} + P_{1:0} c_0]]] \quad (2.6)
\end{aligned}$$

The $n/2$ -bit H-RCA requires $\lceil n/8 \rceil - 1$ pipeline stages, while the $n/2$ -bit fast carry generator requires $\lceil n/16 \rceil - 1$ pipeline stages. Overall, an n -bit fast carry adder requires $\lceil 3n/16 \rceil - 1$ pipeline stages, including the pipeline stage between the fast carry generator and the H-RCA.

2.4.1.5 Adder Implementation Results

The adder circuit has been compared for five design styles—FPGA fabric-based adder (IP Core) generated by the GUI utility in ISE, DSP slice-based adder, the *FloPoCo* generated adder, the fast carry generator-based adder [16] and hybrid RCA. The circuits were implemented on a Xilinx Virtex-5 FPGA, device family XC5VLX330T, package FF1738 and speed grade-2 using the *Xilinx ISE* (v 12.4) design environment and all the *post place and route* implementation results are tabulated and compared with [16]. The dashed entries in Table 2.2 indicate that the equivalent results are either not applicable or not reported in [16]. The speed of operation, resource utilization, and power-delay product (PDP) of the architectures have been compared with those reported in the existing literature (if any) for different modes of implementation. Power-delay product has been calculated as the product of the power dissipation, the (minimum) clock-period (toggle rate of 12.5 %), and the latency (in terms of the number of clock cycles required to complete the computation). Although not explicitly mentioned in [16], the authors informed us through personal correspondence that they inserted register banks in the Fast Carry Adder only at the input and output ports of the circuit to estimate the frequency. The important trend to note here is that in all cases, constrained placement adders give substantially better performance than the corresponding unconstrained placed adders of the same operand width.

A partial floorplan view for a 128-bit adder is shown in Fig. 2.11 for two different implementation modes: fabric IP Core-based pipelined 128-bit adder with unconstrained placement, and a 128-bit pipelined hybrid RCA using proposed design

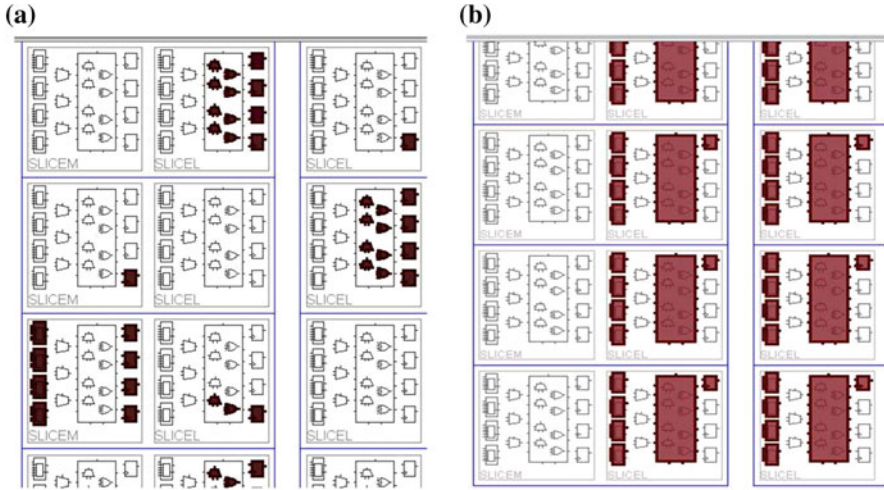


Fig. 2.11 Partial floorplan views for 128-bit adder for fabric ISE GUI-based adder and hybrid RCA with constrained placement. **a** Partial floorplan of IP core-based fabric adder with unconstrained placement. **b** Partial floorplan of hybrid RCA with constrained placement

methodology. From this figure, it is clearly visible that synthesis tools perform an unoptimized inference of logic elements and their random and haphazard placement.

2.4.2 Loadable Bidirectional Binary Counter Architecture

2.4.2.1 Proposed Counter Architecture

An up/down counter can be realized as a combination of a D-FF-based Parallel-In Parallel-Out (PIPO) register and an incrementer/decrementer, which accepts the output of the register as its input, and feeds back its outputs to the input of the register, as shown in Fig. 2.12. Thus, counters have higher routing complexity in comparison to adders. The carry chain has been configured as wide AND gate for up-counter and wide OR gate for down counter as shown in Fig. 2.12, where larger counters can be realized by successive cascading of the “Stage 1” block. The PIPO register has been realized using the “FDRSE” Xilinx primitive which is a D-FF with synchronous reset, set, and clock enable. Pipeline latency cannot be tolerated in a counter design, as the inputs to the PIPO register come at a specific instant of time and outputs are expected to be obtained in the following clock cycle. Hence, the pipelined latches are realized using the “FDCPE” Xilinx primitive [4] which is a D-FF with clock enable and asynchronous preset and clear. These FFs are presetted if the output from the previous carry chain of the adjacent slice is high and cleared if low. For an n -bit counter, $\lceil n/4 \rceil - 1$ asynchronous pipeline stages are required.

Table 2.2 Integer adder implementation results

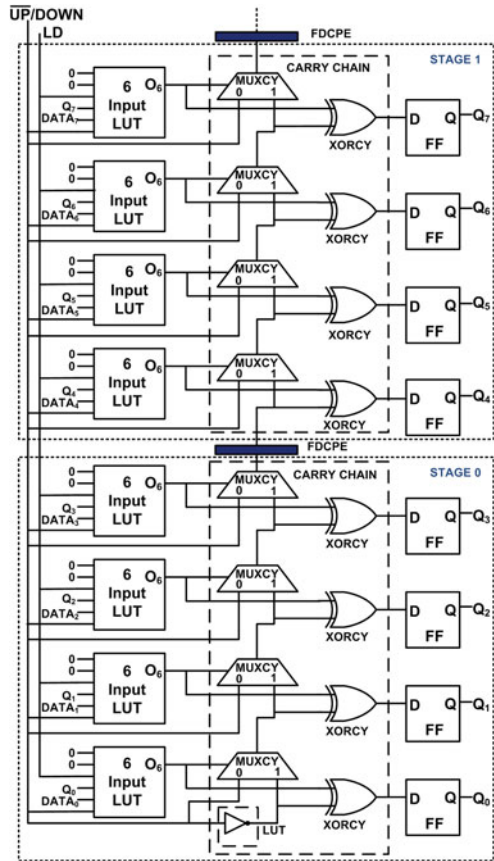
Operand width	Adder circuit	Design with unconstrained placement						Pipelined design with constrained placement							
		Freq (MHz)	Latency (#clk cycles)	Power delay product (nj)	#FF	#LUT	#DSP	#Slice	Freq (MHz)	Latency (#clk cycles)	Power delay product (nj)	#FF	#LUT	#DSP	#Slice
32	Fabric adder (IP core)	378.36	7	0.67	98	116	0	44	—	—	—	—	—	—	—
	DSP slice adder	550.00	2	0.08	0	0	1	0	—	—	—	—	—	—	—
	<i>FloPoCo</i> adder	714.29	16	1.56	138	133	0	51	—	—	—	—	—	—	—
	Fast carry adder [16]	521.00	—	—	98	41	0	—	808.41	5	0.18	9	40	0	10
	Hybrid RCA	—	—	—	—	—	—	—	809.06	7	0.26	8	32	0	8
48	Fabric adder (IP Core)	348.43	11	1.47	157	191	0	78	—	—	—	—	—	—	—
	DSP slice adder	550.00	2	0.10	0	0	1	0	—	—	—	—	—	—	—
	<i>FloPoCo</i> adder	664.45	24	3.66	210	205	0	112	—	—	—	—	—	—	—
	Fast carry adder [16]	472	—	—	146	61	0	—	789.27	8	0.53	14	60	0	15
	Hybrid RCA	—	—	—	—	—	—	—	806.45	11	0.76	12	48	0	12
64	Fabric adder (IP core)	468.60	15	2.48	217	263	0	117	—	—	—	—	—	—	—
	DSP slice adder	500.00	3	0.23	0	0	2	0	—	—	—	—	—	—	—
	<i>FloPoCo</i> adder	595.59	32	6.43	282	277	0	166	—	—	—	—	—	—	—
	Fast carry adder [16]	397.00	—	—	194	81	0	—	788.02	11	1.08	19	80	0	20
	Hybrid RCA	—	—	—	—	—	—	—	806.45	15	1.32	16	64	0	16

(continued)

Table 2.2 (continued)

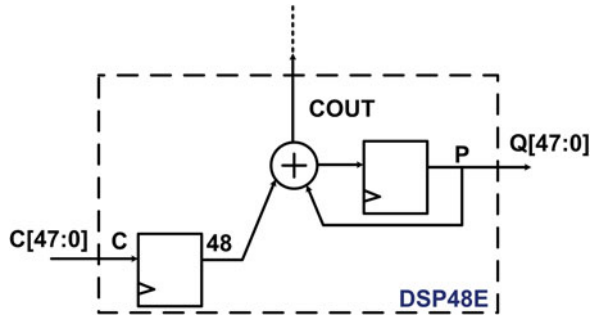
Operand width	Adder circuit	Design with unconstrained placement							Pipelined design with constrained placement						
		Freq (MHz)	Latency (#clk cycles)	Power delay product (nj)	#FF	#LUT	#DSP	#Slice	Freq (MHz)	Latency (#clk cycles)	Power delay product (nj)	#FF	#LUT	#DSP	#Slice
96	Fabric adder (IP core)	413.22	23	6.05	337	407	0	182	–	–	–	–	–	–	–
	DSP slice adder	500.00	3	0.26	0	0	2	0	–	–	–	–	–	–	–
	<i>FloPoCo</i> adder	471.48	48	15.05	426	507	0	208	–	–	–	–	–	–	–
	Fast carry adder [16]	303.00	–	–	290	121	0	–	787.40	17	2.45	29	120	0	30
	Hybrid RCA	–	–	–	–	–	–	–	754.15	23	3.30	24	96	0	24
128	Fabric adder (IP core)	414.94	31	11.80	457	551	0	271	–	–	–	–	–	–	–
	DSP slice adder	500.00	4	0.47	0	0	3	0	–	–	–	–	–	–	–
	<i>FloPoCo</i> adder	522.19	64	25.91	570	747	0	305	–	–	–	–	–	–	–
	Fast carry adder [16]	–	–	–	–	–	–	–	787.40	23	4.42	39	160	0	40
	Hybrid RCA	–	–	–	–	–	–	–	760.46	31	8.16	32	128	0	32

Fig. 2.12 Architecture for loadable, *up/down* counter targeted towards Xilinx Virtex-5 FPGA



The basic building block of this architecture is a 4-bit counter realized within a single slice. The logic functionality of accepting a new data $DATA_i$, when the “load control” signal LD to load external data to the FFs is high, and accepting the output from the FFs when LD is low, along with the XOR operation, is taken care of by the 6-input LUT configured as $O_6 = (\overline{LD} \cdot Q_i + LD \cdot DATA_i) \oplus \overline{UP/DOWN}$, where the counter counts up if $\overline{UP/DOWN} = 0$, and counts down if $\overline{UP/DOWN} = 1$. The terminal count is detected by the carry output of the most significant carry chain. As the external data to be loaded into the register is not supplied directly to the input of the FFs, but comes from the output of the incrementer/decrementer logic, the user must send the value $(x - 1)$ to load an up-counter with the value x , or send $(y + 1)$ to load a down-counter with the value y .

Fig. 2.13 Xilinx Virtex-5
DSP slice-based counter [17]



2.4.2.2 DSP Slice-Based Counter

Counters can also be designed using DSP48E slices, where n -bit counters can be realized by cascading $\lceil n/48 \rceil$ DSP48E slices along a column, as shown in Fig. 2.13. To achieve the desired functionality, the slices have been configured as a 48-bit accumulator each by setting the attributes “OPMODE” as 0101100, and “ALUMODE” as 0000 for addition and 0011 for subtraction [17]. The additional usage information to be taken note of is that while the counter is operating as a down-counter and the user wants to load the registers with the value x , the two’s complement of x must be given as input. Currently, *FloPoCo* (v 2.5.0) does not generate HDL for counters.

2.4.2.3 Counter Implementation Results

The circuits were implemented on a Xilinx Virtex-5 FPGA, device family XC5VLX330T, package FF1738 and speed grade-2 using the *Xilinx ISE* (v 12.4) design environment and all the *post place and route* implementation results are tabulated. Operating frequencies for both the counter synthesized from behavioral description and that generated through the GUI utility deteriorate steadily with increase in the number of output bits. In contrast, the proposed design shows better operand width scalability with respect to frequency. *FloPoCo*, however, does not support counter implementations till its latest release (Table 2.3).

2.5 Compact FPGA Implementation of Cellular Automata Circuits

Cellular Automata (CA) circuits are useful for test pattern generation and construction of Built-In Self Test (BIST) structures within VLSI chips [20]. The regular, modular, and cascadable structure of CA with only local neighborhood dependence of the cells makes it suitable for VLSI implementation [20, 21]. The perceived

Table 2.3 Counter implementation results

Operand width	Counter circuit	Freq (MHz)	Power delay product (pJ)	#FF	#LUT	#DSP	#Slice
32	Behavioral counter	504.80	48.67	32	49	0	15
	Fabric counter (ISE GUI)	536.19	50.52	32	47	0	14
	DSP slice counter	550.00	35.65	0	0	1	0
	Proposed counter	587.89	37.30	39	41	0	17
48	Behavioral counter	400.32	43.39	48	70	0	29
	Fabric counter (ISE GUI)	427.35	59.09	48	69	0	30
	DSP slice counter	550.00	40.80	0	0	1	0
	Proposed counter	567.21	57.10	59	61	0	25
64	Behavioral counter	367.51	53.33	64	94	0	32
	Fabric counter (ISE GUI)	387.59	67.47	64	93	0	39
	DSP slice counter	500.00	50.46	0	0	2	0
	Proposed counter	565.61	59.85	79	81	0	33
96	Behavioral counter	292.65	89.08	96	139	0	46
	Fabric counter (ISE GUI)	298.95	84.39	96	137	0	43
	DSP slice counter	500.00	61.64	0	0	2	0
	Proposed counter	562.75	72.59	119	121	0	48

(continued)

Table 2.3 (continued)

Operand width	Counter circuit	Freq (MHz)	Power delay product (pJ)	#FF	#LUT	#DSP	#Slice
128	Behavioral counter	231.70	121.02	128	186	0	64
	Fabric counter (ISE GUI)	246.49	119.03	128	184	0	67
	DSP slice counter	500.00	79.30	0	0	3	0
	Proposed counter	563.70	111.12	159	161	0	64

regularity and locality of interconnects in a CA are often *logical* rather than *physical*, and difficult to achieve in practical implementations. Implementation of CA on FPGAs often turns out to be inefficient, because usually the user has limited control on the inference of logic elements, along with their placement and routing.

In [22], authors had reported faster implementation of CA on FPGA hardware, compared to optimized software implementation by achieving a speedup in the range of 14–19. A methodology for VLSI implementation of CA algorithms, where an automatic translation scheme from CA algorithms to the corresponding VHDL was proposed in [23]. FPGA-based CA implementation was also reported in [24]. However, to the best of our knowledge, there has been no reported work regarding the principles and design philosophy for efficient low-level implementation of CA on modern families of actual FPGAs, aiming to map the CA structures optimally to the native architecture of the FPGA.

To demonstrate our proposed design philosophy, consider a *null boundary*, maximal length linear CA [25] which is a collection of a discrete lattice of cells, where each cell has a D-FF with associated combinational logic. If the CA has n cells, then the state at any instant may be expressed as $Y_t = \{q_0(t), q_1(t), \dots, q_{n-1}(t)\}$, where $q_i(t)$ denotes the state of the i th cell at the t th instant of time. The state of the i th cell at the $(t + 1)$ th instant of time is denoted by $q_i(t + 1)$, where $q_i(t + 1) = f(q_{i-1}(t), q_i(t), q_{i+1}(t))$, which determines the combinational logic for each stage. Here, ' $f()$ ' is known as the *rule of the CA* [25], which, if expressed in the form of a truth-table, the equivalent decimal output is called *rule number of the CA*. For example, the next state logic equations for *rule-90* and *rule-150* CAs are given as $q_i(t + 1) = q_{i-1}(t) \oplus q_{i+1}(t)$ and $q_i(t + 1) = q_{i-1}(t) \oplus q_i(t) \oplus q_{i+1}(t)$ respectively [26] with their circuit representations as depicted in Fig. 2.14.

If the CA is *linear*, the combinational logic functions $f()$ involves only XOR logic. A CA having a combination of XOR and XNOR logic is called an *additive* CA, whereas for *non-linear* or *non-additive* CA, $f()$ involves AND/OR logic [27]. If all CA cells obey the same rule, then it is termed as *uniform* CA, else it is a *hybrid*

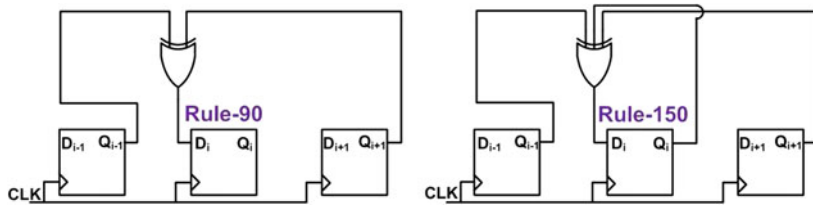


Fig. 2.14 Combinational logic for cells corresponding to *rule-90* and *rule-150*

CA. By convention, CA is usually described by a string of 0's and 1's, where, for example, '1' refers to rule-150 and '0' refers to rule-90. Our proposed methodology can efficiently implement two-rule *linear*, *additive*, *uniform* and *hybrid* CAs.

2.5.1 Adapting CA to the Native FPGA Architecture

Packing is a key step in the FPGA tool flow that is tightly integrated with the boundaries between *logic synthesis*, *technology mapping* and *placement* [3]. For Virtex-5 FPGAs, the packing technique targets the dual-output LUTs to achieve area efficiency by exploring the feasibility of packing two logic functions into a single LUT [3]. This is possible whenever the two logic functions have no more than five distinct input variables. In such cases, a more efficient mapping of the design is expected, culminating into shorter interconnect wirelength, which in turn results in lesser critical path delay. However, our implementation results for Virtex-6 family of FPGAs, which is an advanced and modified version of Virtex-5, clearly show that in spite of the methodology adopted by the common FPGA CAD tools, the packing behavior is highly unpredictable and the tool fails to configure the LUTs in the dual output mode. This doubles the overall LUT and slice requirement.

Consider a 1-D CA where the next state of a particular cell depends only on itself or on one or both of its two immediate neighbors. It is easy to deduce that in such cases, any two adjacent cells can have a maximum of four distinct inputs. In such a situation, it is possible to pack the next state logic for any two adjacent cells of a CA into a single LUT. Since Virtex-6 architectures facilitate registering of both the LUT outputs using two FFs present in the same slice as that of the LUT, we can achieve a compact FPGA realization of the architecture [28]. The architecture for a 16 cell 1-D linear maximal length CA for the (primitive) polynomial $x^{16} + x^5 + x^3 + x^2 + 1$ (or the equivalent *hybrid* rule <0001111001001000>) [29] is shown in Fig. 2.15. Thus, for an n -cell maximal length CA architecture, $\lceil n/8 \rceil$ Virtex-6 FPGA slices are required.

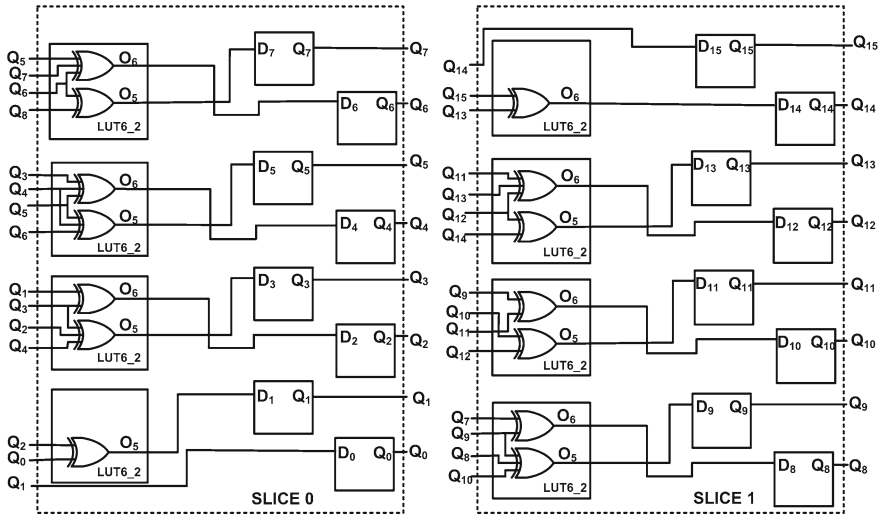


Fig. 2.15 Optimized architecture for a high-performance 16-bit 1-D linear CA for the (primitive) polynomial $x^{16} + x^5 + x^3 + x^2 + 1$ (or the equivalent hybrid rule <0001111001001000>) mapped on Xilinx Virtex-6 FPGAs [28]

2.5.2 CA Implementation Results

The CA circuits were implemented on Xilinx Virtex-6 FPGA, device family XC6VLX550T, package FF1760 and speed grade -2 using the *Xilinx ISE* (v 12.4) design environment. Polynomials of CAs of the order 32, 48, 64, 80, and 96 were implemented using two different techniques—RTL coding followed by unconstrained automatic logic synthesis by ISE and the custom design technique using the proposed design methodology. It was observed that for an RTL description of the CA circuit, the Xilinx *post place and route* results indicate that double the FPGA area gets consumed than what a compact realization should have taken. The operating speed for the CA circuits also drastically reduces as the order of the polynomial is steadily increased, which is undesirable from the point of view of hardware acceleration. The implementation results were compared with respect to their frequency of operation, PDP, and hardware resource requirement (FFs, LUTs and slices), and are tabulated in Table 2.4. The polynomials are from [30] and, for example, the entry in the polynomial field of Table 2.4, 32 28 27 1 0, represents the polynomial $x^{32} + x^{28} + x^{27} + x + 1$.

Table 2.4 CA implementation results

Polynomial	Mode of implemen- tation	Freq (MHz)	Power delay product (pJ)	#FF	#LUT	#Slice
32, 28, 27, 1, 0	RTL design	1014.20	31.61	32	30	8
	Proposed design	1103.75	31.15	32	16	4
48, 28, 27, 1, 0	RTL design	320.41	37.36	48	46	12
	Proposed design	1089.32	40.26	48	24	6
64, 4, 3, 1, 0	RTL design	361.40	43.80	64	64	16
	Proposed design	1083.42	52.92	64	32	8
80, 38, 37, 1, 0	RTL design	414.08	64.05	80	78	20
	Proposed design	976.56	59.08	80	40	10
96, 49, 47, 2, 0	RTL design	361.79	70.92	96	96	24
	Proposed design	908.27	62.59	96	48	12

2.6 Design Automation

The design of all the circuits has been automated using a CAD tool developed by us. We call the CAD tool *FlexiCore*, short for “**Flexible** Arithmetic Soft **Core** Generator.” It is flexible in a sense that the operand widths for the mapped circuits can be varied, and the CAD tool allows partial control to the user over the placement of the circuits on the FPGA fabric. The tool is developed in JAVA, and includes a simple GUI. The CAD software executable is invoked from the TCL command—prompt in-built in Xilinx ISE using a top-level TCL script. Currently, *FlexiCore* can generate synthesizable HDL and placement constraints for adders/subtractors, absolute difference circuits, multipliers, squarers, universal shift registers, comparators, counters, and CA-based pseudorandom binary sequence generators.

The *FlexiCore* design flow is depicted in Fig. 2.16. Here, the top-level script invokes a GUI which displays the list of circuits currently supported by *FlexiCore*, and prompts the user to enter (in the GUI entry fields) the circuits (along with their operand widths and whether the user wants pipelined/non-pipelined version), for which the user wants constrained placement-based high-performance design. The user can also optionally enter the starting coordinate for the entire constrained placement exercise. If this is not provided, *FlexiCore* determines the feasible starting coordinate from the existing project constraints file called User Constraints File (.ucf).

After the user enters her options, *FlexiCore* examines the feasibility of placement of the selected building blocks on the FPGA fabric, with the starting coordinate

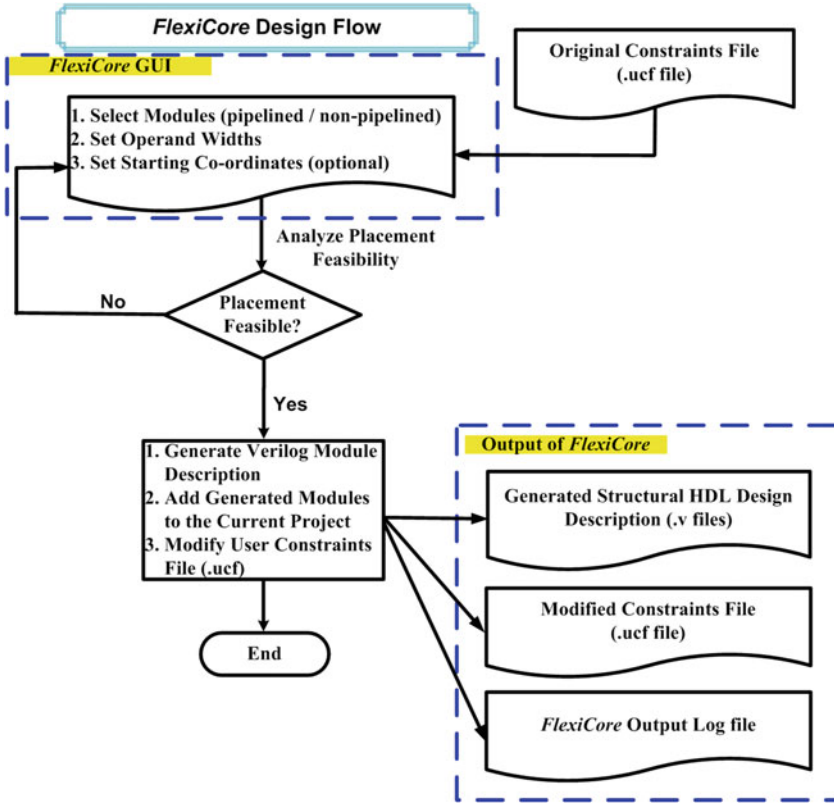


Fig. 2.16 The *FlexiCore* design flow for arithmetic circuits

entered by the user as origin, or the starting coordinate inferred. It takes into consideration the existing placement constraints, if any, in the project constraints file. If the placement is deemed feasible, *FlexiCore* performs the following:

- Generates the Verilog module descriptions for the selected circuit building blocks, and adds the files to the current project. Care is taken to ensure that no hardware primitive on the FPGA is used more than once in building the high-performance building blocks. Pipeline registers as required, are automatically inserted. At present, *FlexiCore* supports two options—either a maximally pipelined implementation (optimized for Virtex-5 and Virtex-6 platform), or a purely combinational circuit. We expect to support variable latency circuits in future.
- Modifies the project User Constraints File (.ucf), by adding the placement constraints for the generated high-performance circuit building blocks.
- Creates a log file to provide the user with all the necessary information about the generated modules.

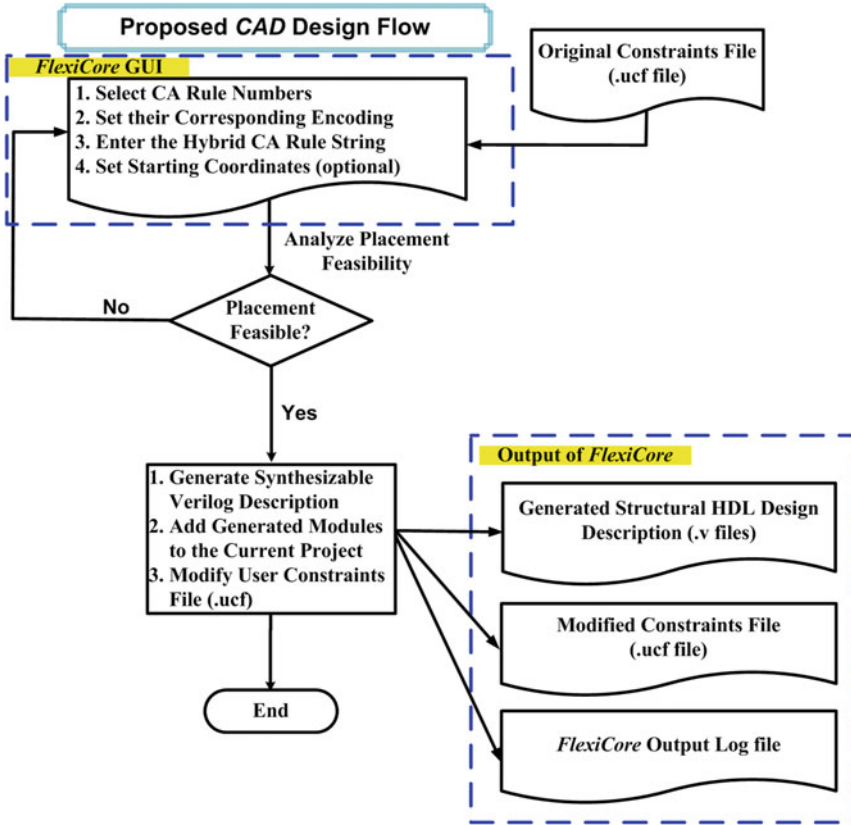


Fig. 2.17 The *FlexiCore* design flow for CA circuits [28]

If *FlexiCore* fails to find a feasible placement configuration, it reports it to the user and again prompts her to enter a (reduced) number of building blocks, or a different starting coordinate. Note that the situation where *FlexiCore* fails to find a feasible placement rarely arises, given the large availability of resources on a Virtex family FPGA. We did not find any such scenario with our real-life design testcases.

To accommodate the CA circuits into the CAD tool for their automatic generation, provision has been kept for the user to invoke the GUI, which displays all the list of rules corresponding to which equivalent CA circuits can be generated, and prompts the user to enter the following fields: the two CA rule numbers, their corresponding encoding of 0 and 1, and the hybrid CA rule comprising of a string of 0's and 1's. The CAD tool interprets the string by reading two bits at a time, calculates the *truth table* of the dual output LUTs appropriately for realizing the next state logic for the CA cells, and instantiates the required FPGA logic elements in the HDL code. The remaining CAD tool flow remains to be the same as for arithmetic circuits. The design flow, particularly for CA circuits, is shown in Fig. 2.17.

2.7 Case Study—a Greatest Common Divisor (GCD) Calculator

We shall present a case study of the hardware implementation of a Greatest Common Divisor (GCD) calculator. The architecture uses several of the arithmetic building blocks supported by *FlexiCore* such as the absolute difference circuit, counter, and a barrel shifter. The architecture has been derived from the Binary GCD algorithm [31] which has been explained in Algorithm 1. This algorithm uses simpler arithmetic operations than the conventional Euclidean GCD algorithm as it replaces complex operations such as division and multiplication with division and multiplications by powers of two (implemented using only shift operations), comparisons and subtractions [32], thereby making it suitable for hardware implementation.

The architecture for the algorithm at the block diagram level is shown in Fig. 2.18. We present two *multi-function* registers P and Q which are loaded in accordance with the control signals: active low load control signal \overline{INIT} , which accepts two unsigned integers as inputs whose GCD has to be computed, and LSBs of registers P and Q as depicted in Table 2.5. The multifunction registers and its associated combinational logic, which is a nonstandard representation of a 4:1 multiplexer, has been mapped intelligently to the 6-input LUTs and wide function multiplexers $MUXF7$ as shown

Algorithm 1: GCD Calculation Algorithm

Input: 2 unsigned integers: P and Q .

Output: S : GCD of P and Q

```

1   $Rem(P, Q)$ : Remainder when  $P$  is divided by  $Q$ 
2   $abs(P - Q)$ : Absolute difference of  $P$  and  $Q$ 
3   $min(P, Q)$ : Minimum of  $P$  and  $Q$ 
4   $Computation\_Over\_Flag \leftarrow 0, R \leftarrow 0$ 
5  begin
6      while  $P \neq Q$  do
7          if  $(Rem(P, 2) == 0)$  then
8               $P \leftarrow P/2;$ 
9              if  $(Rem(Q, 2) == 0)$  then
10                  $Q \leftarrow Q/2;$ 
11                  $R \leftarrow R + 1;$ 
12             else
13                 if  $(Rem(Q, 2) == 0)$  then
14                      $Q \leftarrow Q/2;$ 
15                 else
16                      $P \leftarrow abs(P - Q);$ 
17                      $Q \leftarrow min(P, Q);$ 
18 end
19  $S \leftarrow P * (2^R);$ 
20  $Computation\_Over\_Flag \leftarrow 1;$ 

```

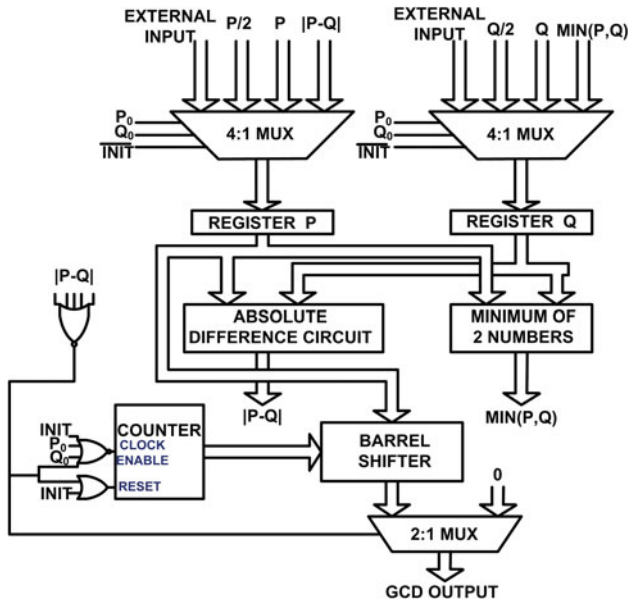


Fig. 2.18 Overall architecture of the GCD computation circuit

Table 2.5 Function table for the multi-function registers and counter

Control/Select signals			Registers		Counter
\overline{INIT}	P_0	Q_0	P	Q	R
0	X	X	LOAD	LOAD	0
1	0	0	$P/2$	$Q/2$	$R + 1$
1	0	1	$P/2$	Q	R
1	1	0	P	$Q/2$	R
1	1	1	$ P - Q $	$\min(P, Q)$	R

in Fig. 2.19 to ensure compact implementation. The absolute difference circuit has been realized as was proposed in [33] which comprises of a less-than comparator (Fig. 2.20) and a subtractor (Fig. 2.21) as its sub-circuits. If A and B are two inputs, the n -bit less-than comparator generates a high signal if $A < B$. Each LUT accepts 2-bit sub-words $A_{i:i-1}$ and $B_{i:i-1}$, each of which has no more than four distinct inputs, and outputs two signals $AeqB_{i:i-1}$ and $AlessB_{i:i-1}$. $AeqB_{i:i-1} = 1$ if $A_{i:i-1} = B_{i:i-1}$ and $AlessB_{i:i-1} = 1$ if $A_{i:i-1} < B_{i:i-1}$. $AeqB_{i:i-1}$ drives the select line of the

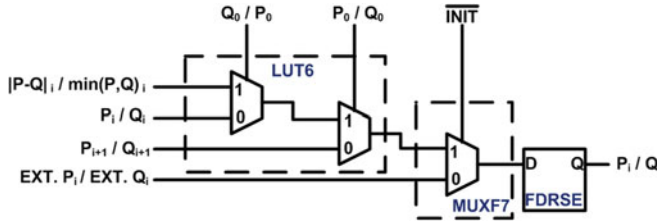


Fig. 2.19 Multifunction register

multiplexer of the carry chain and $Aless B_{i:i-1}$ is an input to the multiplexer which is selected if $Aeq B_{i:i-1}=0$. If $x_i = A_i \odot B_i$ and $x_{i-1} = A_{i-1} \odot B_{i-1}$, then

$$Aless B_{i:i-1} = \overline{A_i} B_i + x_i \overline{A_{i-1}} B_{i-1}$$

$$Aeq B_{i:i-1} = x_i x_{i-1}$$

The output of the less-than comparator $A_l_B_n$ decides upon the operation $A - B$ or $B - A$. For an n -bit less than comparator, its output $A_l_B_n$ is obtained using the following Boolean logic recurrence relation:

$$A_l_B_n = \overline{Aeq B_{n:n-1}} Aless B_{n:n-1} + Aeq B_{n:n-1} A_l_B_{n-2}$$

where the *base* condition is $A_l_B_0 = 0$. This recurrence relation bears exact resemblance to (2.3) making it an ideal candidate for carry-chain implementation. When $A_l_B_n = 1$, $B + \overline{A} + 1$ is computed, else $A + \overline{B} + 1$ is computed, as shown in Fig. 2.21, where \overline{A} and \overline{B} are the 1's complement of A and B respectively.

Fig. 2.20 Module computing if $A < B$ [33]

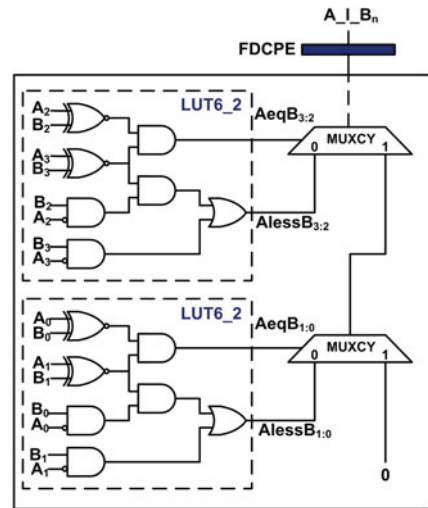
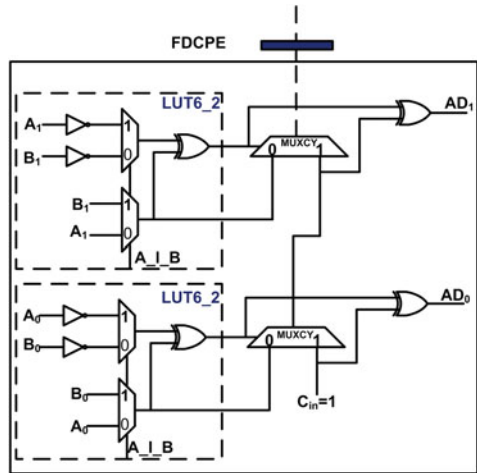


Fig. 2.21 Module computing absolute difference [33]



The absolute difference circuit has been pipelined using the “FDCPE” Xilinx primitive [4] where these FFs are presetted if the output from the previous carry chain of the adjacent slice is high and cleared if low. An intermediate output A_I_B (which decided whether to compute $A - B$ or $B - A$) of the absolute difference circuit serves as a select line to the multiplexer which outputs the minimum of two numbers. This architecture to compute the minimum of two numbers has been realized using dual output LUTs as shown in Fig. 2.22. The counter keeps track of the number of left shifts to be applied to the contents of the P register after the final iteration. The contents of the register P are left-shifted using a barrel shifter which is implemented using dual output LUTs as shown in Fig. 2.23 where stage i of the barrel shifter ($i \geq 0$) can implement a $2^i/0$ bit shift. Thus, the data to be shifted is given to the data inputs of the multiplexers, whereas the amount of left shift is given as input to the select lines of the multiplexers in the barrel shifter. The final output gives the GCD of two numbers.

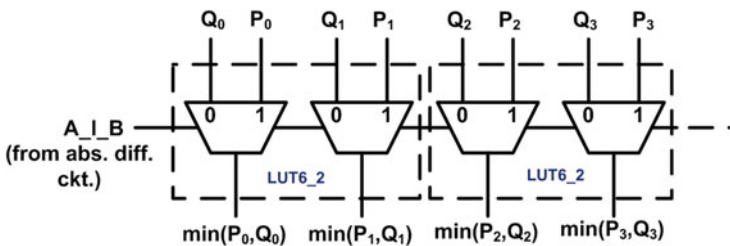


Fig. 2.22 Circuit to compute minimum of two numbers

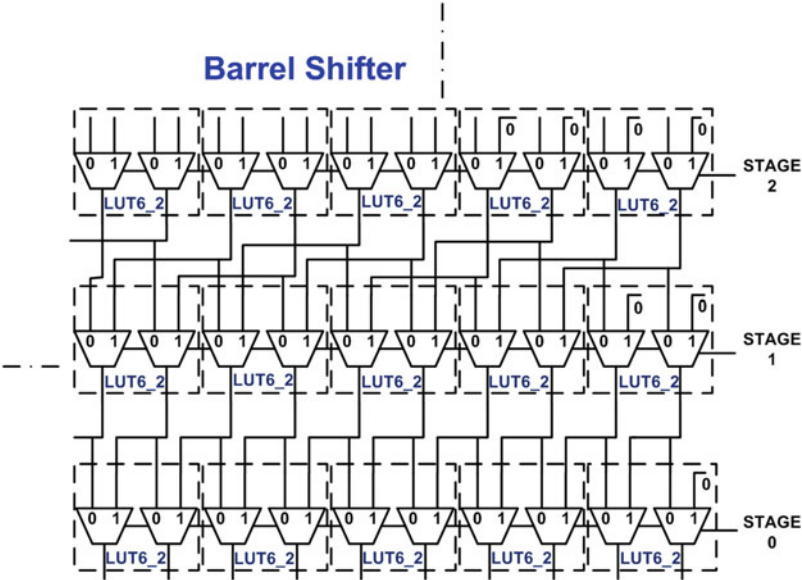


Fig. 2.23 LUT level implementation of the barrel shifter

2.7.1 GCD Implementation Results

The GCD computation circuit for 32-bit operands was implemented on the Xilinx Virtex-5 FPGA using two approaches: behavioral Verilog modelling, and second, using constrained arithmetic circuit descriptions generated by *FlexiCore*. Results are tabulated in Table 2.6, where the two input operands are 70 and 100. The results clearly indicate that using the second approach, the designer can achieve a higher frequency and lower PDP value with considerable lesser amount of hardware resources.

Table 2.6 Implementation results for a 32-bit GCD Circuit (Operands: 100 and 70)

Implementation mode	Freq (MHz)	Power delay product (pJ)	#FF	#LUT	#Slice
Behavioral modelling	160.49	745.85	69	356	208
Primitive instantiation	214.73	508.87	87	298	93

2.8 Conclusion

Manual instantiation of hardware primitives and macros, and their careful, constrained placement on the FPGA fabric leads to very promising performances in terms of speed. We have considered some arithmetic circuits and pseudorandom sequence generators which are very regular in their architectures and have shown how to configure them using the *bit-sliced* design paradigm where an entire architecture has been constructed using identical sub-circuits. Designs that are pipelined and have a very regular data flow, like those considered in our work, usually lend themselves to regular floorplanning. Since each slice of an FPGA are register-rich, pipelined implementations can be done with ease without consuming additional number of slices. The regular architectures also facilitate their design automation which is taken care of by the *FlexiCore* CAD tool. The tool is not only capable of generating the synthesizable HDL and placement directives for the designs, but can also examine the feasibility of placement of a circuit by ensuring that the area spanned by it on the FPGA fabric is not occupied by any other logic.

Acknowledgments The authors would like to thank Prof. Anindya Sundar Dhar, Department of Electronics and Electrical Communication Engineering, IIT Kharagpur, and Dr. Debdeep Mukhopadhyay, Department of Computer Science and Engineering, IIT Kharagpur, for their valuable insights into the work. The authors would also like to acknowledge two undergraduate students of the Department of Computer Science and Engineering, IIT Kharagpur, Mohammad Salman and Sreemukh Kardas, for their contributions in developing the proposed CAD tool, *FlexiCore*.

References

1. Preußner, T.B., Zabel, M., Spallek, R.G.: Accelerating computations on FPGA carry chains by operand compaction. In: 20th IEEE Symposium on Computer Arithmetic (ARITH), pp. 95–102 (2011)
2. Preußner, T.B., Spallek, R.G.: Mapping basic prefix computations to fast carry-chain structures. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 604–608 (2009)
3. Ahmed, T., Kundarewich, P.D., Anderson, J.H.: Packing techniques for Virtex-5 FPGAs. ACM Trans. Reconfig. Technol. Syst. (TRETs), 2(18), 18:1–18:24 (2009)
4. Xilinx Inc., Virtex-5 Libraries Guide for HDL Designs, UG621 (v 11.3) (2009). Cited 16 September 2009, http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/Virtex5_hdl.pdf
5. Ehliar, A.: Optimizing Xilinx designs through primitive instantiation. In: Proceedings of the 7th FPGAWorld Conference, pp. 20–27 (2010)
6. FloPoCo: Arithmetic core generator (2014). Cited 14 June 2014, <http://flopoco.gforge.inria.fr/>
7. Dinechin, F.de, Pasca, B.: Designing custom arithmetic data paths with FloPoCo. IEEE Des. Test Comput. 28(3), 18–27 (2009)
8. Cosoroaba, A., Rivoallon, F.: Xilinx Inc., White paper: Virtex-5 family of FPGAs. Achieving Higher System Performance with the Virtex-5 Family of FPGAs WP245 (v1.1.1) (2006). Cited 7 July 2006, http://www.origin.xilinx.com/support/documentation/white_papers/wp245.pdf
9. Xilinx Inc., Virtex-5 FPGA user guide, UG190 (v 5.4) (2012). Cited 16 March 2012, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

10. Xilinx Inc., Virtex-6 FPGA configurable logic block, UG364 (v 1.2) (2012). Cited 24 June 2009, http://www.xilinx.com/support/documentation/user_guides/ug364.pdf
11. Verma, A.K., Brisk, P., Jenne, J.P.: Challenges in automatic optimization of arithmetic circuits. In: 19th IEEE Symposium on Computer Arithmetic (ARITH), pp. 213–218 (2009)
12. Roy, S.S., Rebeiro, C., Mukhopadhyay, D.: Theoretical modeling of the Itoh-Tsujii inversion algorithm for enhanced performance on k -LUT based FPGAs. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–6 (2011)
13. Hachtel, G.D., Somenzi, F.: Logic Synthesis and Verification Algorithms. Kluwer Academic Publisher, Dordrecht (1996)
14. Weste, N.H.E., Harris, D., Banerjee, A.: CMOS VLSI Design: A Circuits and Systems Perspective. 3rd edn. Pearson Publisher, New York (2011)
15. Cortadella, J., Llabería, J.: Evaluation of $A + B = K$ conditions without carry propagation. IEEE Trans. Comput. **41**(11), 1484–1487 (1992)
16. Zicari, P., Perri, S.: A fast carry-chain adder for Virtex-5 FPGAs. In: 15th IEEE Mediterranean Electrotechnical Conference (MELECON), pp. 304–308 (2010)
17. Xilinx Inc., Virtex-5 FPGA XtremeDSP design considerations user guide, UG193 (v 3.5) (2012). Cited 26 January 2012, http://www.xilinx.com/support/documentation/user_guides/ug193.pdf
18. Koren, I.: Computer Arithmetic Algorithms, 2nd edn. A.K.Peters Ltd, Natick (2002)
19. Brent, R.P., Kung, H.T.: A Regular layout for parallel adders. IEEE Trans. Comput. **C-31**(3), 260–264 (1982)
20. Sarkar, P.: A brief history of cellular automata. ACM Comput. Surv. (CSUR) **32**(1), 80–107 (2000)
21. Chowdhury, D.R., Chaudhuri, P.P.: Architecture for VLSI design of CA based byte error correcting code decoders. In: Proceedings of the 7th International Conference on VLSI Design, pp. 283–286 (1994)
22. Halbach, M., Hoffmann, R.: Improving cellular automata in FPGA logic. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium, pp. 258–262 (2004)
23. Sirakoulis, G.C., Karafyllidis, I., Thanailakis, A., Mardiris, V.: A methodology for VLSI implementation of cellular automata algorithms using VHDL. Adv. Eng. Softw. **32**(3), 189–202 (2000)
24. Torres-Huitzil, C., Delgadillo-Escobar, M., Nuno-Maganda, M.: Comparison between 2D cellular automata based pseudorandom number generators. IEICE Electron. Express **9**(17), 1391–1396 (2012)
25. Das, A.K., Ganguly, A., Dasgupta, A., Bhawmik, S., Chaudhuri, P.P.: Efficient characterization of cellular automata. IEEE Proc. Comput. Digital Tech. **137**(1), 81–87 (1990)
26. Chaudhuri, P.P., Chowdhury, D.R., Nandi, S., Chattopadhyay, S.: Additive Cellular Automata Theory and its Application. vol. 1. IEEE Computer Society Press (1997)
27. Mukhopadhyay, D.: Group properties of non-linear cellular automata. J. Cell. Autom. **5**(1–2), 139–155 (2010)
28. Palchadhuri, A., Chakraborty, R.S., Salman, M., Kardas, S., Mukhopadhyay, D.: Highly compact automated implementation of linear CA on FPGAs. In: Cellular Automata—11th International Conference on Cellular Automata for Research and Industry, pp. 388–397 (2014)
29. Cattell, K., Muzio, J.: Technical Report: Tables of linear cellular automata for minimal weight primitive polynomials of degrees up to 300. Issue: 163. University of Victoria (BC), Department of Computer Science (1991)
30. Bardell, P.H., McAnney, W.H., Savir, J.: Built-In Test for VLSI: Pseudorandom Techniques, Wiley, London (1987)
31. Stehlé, D., Zimmermann, P.: A binary recursive GCD algorithm. In: Proceedings of ANTS'04, Lecture Notes in Computer Science, vol. 3076, pp. 411–425. Springer, New York (2004)
32. Brent, R.P., Kung, H.T.: A systolic algorithm for integer GCD computation. In: IEEE 7th Symposium on Computer Arithmetic (ARITH), pp. 118–125 (1985)
33. Perri, S., Zicari, P., Corsonello, P.: Efficient absolute difference circuits in Virtex-5 FPGAs. In: 15th IEEE Mediterranean Electrotechnical Conference (MELECON), pp. 309–313 (2010)

Computational Intelligence in Digital and Network
Designs and Applications

Fakhfakh, M.; Tlelo-Cuautle, E.; Siarry, P. (Eds.)

2015, XIX, 350 p. 151 illus., 69 illus. in color., Hardcover

ISBN: 978-3-319-20070-5