2 Background

This chapter provides the background of the two main parts of this thesis. The circuit and graph representations of SCM, MCM and CMM operations are given and the notation and related work is introduced which is common for the later chapters.

2.1 Single Constant Multiplication

The multiplication with a single constant is introduced in the following, starting from the standard binary multiplication.

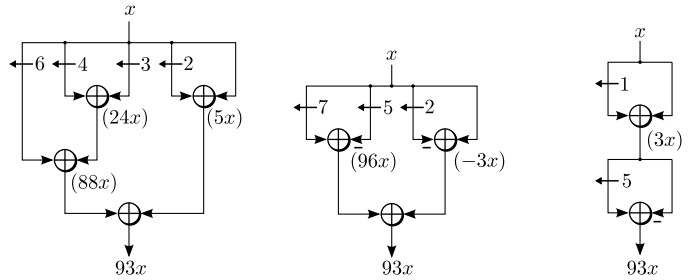
2.1.1 Binary Multiplication

A generic multiplication of two unsigned binary B-bit integer numbers $x = (x_{B-1} \dots x_1 x_0)$ and $y = (y_{B-1} \dots y_1 y_0)$, with $x_i, y_i \in \{0, 1\}$ can be written as

$$x \cdot y = x \cdot \left(\sum_{i=0}^{B-1} 2^i y_i \right) = \sum_{i=0}^{B-1} 2^i \underbrace{x \cdot y_i}_{\text{partial product}} \quad . \tag{2.1}$$

In a generic multiplier, the partial products $x \cdot y_i$ can be obtained by a bitwise AND-operation. The final product is then obtained by adding the bit-shifted partial products. Now, if y is a constant bit vector, all bits y_i which are zero lead to zero partial products and the corresponding adders can be removed. Thus, the number of required adders for the constant multiplication using this representation is equal to the number of non-zero elements (Hamming weight) in the binary representation of y minus one. To illustrate this, consider a multiplier with $y = 93$. Its binary representation $y = 1011101_2$ has five ones and requires four adders in the corresponding add-and-shift-realization $93x = (2^6 + 2^4 + 2^3 + 2^2 + 1)x$ as illustrated in Figure 2.1(a). Note that bit shifts to the left are indicated by left arrows in Figure 2.1.

M. Kumm, *Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays*, DOI 10.1007/978-3-658-13323-8_2



(a) Binary realization (b) CSD realization (c) Minimum adder realization

Figure 2.1: Different adder circuits to realize a multiplication with 93

2.1.2 Multiplication using Signed Digit Representation

The total number of operations can be further reduced by allowing subtract operations in the add-and-shift network. As the subtract operation is nearly equivalent in hardware cost (in terms of area, speed and energy), both are referred to as adders in the following. The adder reduction can be realized by converting the constant to the signed digit (SD) number system [26–28] (sometimes also called "ternary balanced notation" [29]), in which each digit is represented by one of the values of $\{-1, 0, 1\}$. In the example, the coefficient can be represented by $93 = 10\overline{1}00\overline{1}01_{\mathrm{SD}}$ where digit $\overline{1}$ corresponds to -1, i. e., $93 = 2^7 - 2^5 - 2^2 + 2^0$. Now, the corresponding circuit uses one adder less compared to the binary representation as illustrated in Figure 2.1(b).

The signed digit number system is not unique as there may exist several alternative representations for the same number. For example, 93 could also be represented by $93 = 1\overline{1}0\overline{1}000\overline{1}\,\overline{1}_{\mathrm{SD}}$, which has the same number of non-zero digits as the binary representation. A unique representation with minimal number of non-zero digits is given by the canonic signed digit (CSD) representation. A binary number can be converted to the CSD with the following simple algorithm [30, 31]:

Algorithm 1 (CSD Conversion Algorithm)**.** *Starting with the least significant bit (LSB), search a bit string of the form '*$011\ldots11$*' with length* ≥ 2 *and replace it with '*$10\ldots0\overline{1}$*' of the same length. This procedure is repeated until no further bit string can be found.*

For example, the binary representation of $93 = 1011101_2$ is transformed to $93 = 1100\bar{1}01_{\text{SD}}$ in the first iteration which is transformed to $93 = 10\bar{1}00\bar{1}01_{\text{CSD}}$ in the second and final iteration. The CSD representation is unique and guarantees a minimal number of non-zeros. However, there may still be several SD representations with a minimal number of non-zeros. The representations with a minimal number of non-zeros are called minimum signed digit (MSD). The SD representation after the first iteration for 93 is an example of an MSD number which is not a CSD number.

Starting from the CSD representation, valid MSD representations can be constructed by replacing the bit patterns '$10\bar{1}$' with '011' or '$\bar{1}01$' with '$0\bar{1}\bar{1}$'. Doing this for all combinations results in a set of MSD numbers. For the example '93', four different MSD representations can be constructed:

$$\begin{aligned} 93 &= 10\bar{1}00\bar{1}01_{\text{CSD}} \\ &= 01100\bar{1}01_{\text{MSD}} \\ &= 10\bar{1}000\bar{1}\bar{1}_{\text{MSD}} \\ &= 011000\bar{1}\bar{1}_{\text{MSD}} \end{aligned} \tag{2.2}$$

2.1.3 Multiplication using Generic Adder Graphs

Although the arithmetic complexity of the constant multiplier can be reduced by using an MSD representation of the constant, it is not guaranteed to be minimal. Consider again the example number 93 which can be factored into $93 = 3 \cdot 31$. With $3 = 11_{\text{MSD}}$ and $31 = 10000\bar{1}_{\text{MSD}}$, the cascade of these two constant multipliers reduces the required adders to only two as shown in Figure 2.1(c).

This solution can be obtained by recognizing that two patterns in the CSD representation of 93 are related to each other:

$$10\bar{1}_{\text{SD}} = -\bar{1}01_{\text{SD}} \ . \tag{2.3}$$

With that, 93 can be represented as $93 = 10\bar{1}00\bar{1}01_{\text{CSD}} = 3 \cdot 2^5 - 3 = 3 \cdot (2^5 - 1) = 3 \cdot 31$. This concept is known as sub-expression sharing and the corresponding optimization method is called common subexpression elimination (CSE) [12,32,33]. However, there is no guarantee to find a solution with the minimal number of adders as some common patterns may be hidden by other patterns due to overlaps [34]. Take for example, $25 = 10\bar{1}001_{\text{CSD}}$ which can be factored into $25 = 5 \cdot 5 = 101_2 \cdot 101_2$. Due to the fact that two ones

in the pattern 101_2 overlap when computing $25 = 101_2 + 2^2 \cdot 101_2 = 11001_2$, the corresponding 101_2 pattern is not visible in the CSD representation.

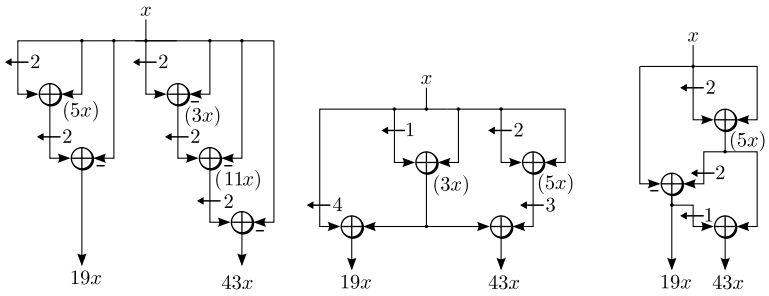
Finding an adder circuit that multiplies with a given single constant using a minimum number of adders is an optimization problem which is known as the SCM problem.

2.2 Multiple Constant Multiplication

A frequent application in signal processing is the multiplication of a variable by multiple constants which is commonly called multiple constant multiplication (MCM). Here, intermediate adder results can be shared between different coefficients such that the overall complexity is reduced. Take, for example, the multiplication with the constants 19 and 43. Their CSD based multipliers as introduced in the last section are shown in Figure 2.2(a). Redundancies may be found using CSE like in the SCM case but not only within a single coefficient but also between different coefficients. One MSD representation of the example constants is $19 = 10011_{\text{MSD}}$ and $43 = 101011_{\text{MSD}}$ (in which case the binary representations are MSD representations themselves). In both representations, the bit pattern 11_{MSD} is found which can be shared to reduce one adder. The resulting adder circuit is shown in Figure 2.2(b). However, as demonstrated in Figure 2.2(c) another adder configuration can be found which does the same multiplications with one adder less. It can not be obtained by the CSE approach as no bit pattern of the factor $5 = 101_{\text{MSD}}$ can be found in both MSD representations of the constants. The problem in finding the adder circuit with minimum adders is known as the MCM problem.

2.3 Constant Matrix Multiplication

An extension of the MCM operation (and the corresponding optimization problem) is the multiplication of a constant matrix with a vector of variables, which is called constant matrix multiplication (CMM) in the following. Its application can be found in digital filters (parallel 2D filters or polyphase filters [35, 36]), transformations like the FFT [37] or DCT [38], color space conversion in video processing [39] as well as in the multiplication of complex numbers by complex constants [40]. Again, the operation count can be reduced by sharing intermediate results. Take, for example, the complex



(a) CSD realization (b) CSE realization (c) Minimum adder realization

Figure 2.2: Different adder circuits to realize a multiplication with coefficients 19 and 43

multiplication

$$(x_r + jx_i) \cdot (19 + j43) = \underbrace{19x_r - 43x_i}_{=y_r} + j(\underbrace{43x_r + 19x_i}_{=y_i}) \tag{2.4}$$

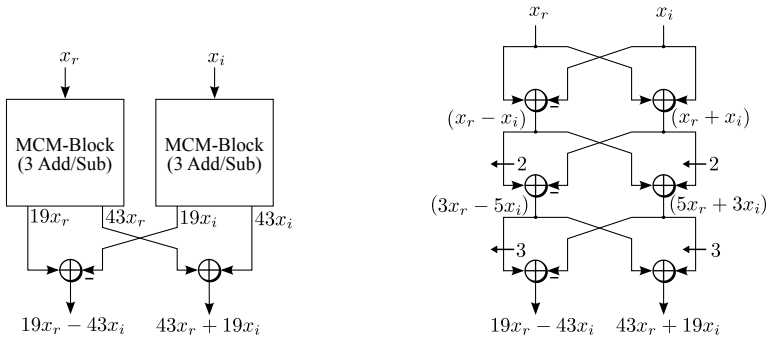
which can be represented in matrix form as

$$\begin{pmatrix} y_r \\ y_i \end{pmatrix} = \begin{pmatrix} 19 & -43 \\ 43 & 19 \end{pmatrix} \cdot \begin{pmatrix} x_r \\ x_i \end{pmatrix} = \begin{pmatrix} 19x_r - 43x_i \\ 43x_r + 19x_i \end{pmatrix} . \tag{2.5}$$

One solution to realize this matrix multiplication is to use the MCM circuit of Figure 2.2(c) two times for real and imaginary input and to add/subtract the results as illustrated in Figure 2.3(a). Even if the MCM solutions are optimal in terms of number of adders, the CMM circuit in Figure 2.3(a) is not optimal as there exist a CMM circuit with two adders less as shown in Figure 2.3(b). Hence, redundancies between different inputs were used to reduce the complexity. Finding an adder circuit with minimum adders for a given constant matrix is known as the CMM problem.

2.4 Definitions

Before starting to formulate the optimization problems and discussing the previous work on how to solve these, some common concepts and notations



(a) CMM with two MCM operations (b) Optimized CMM operation

Figure 2.3: Example CMM operation with constant $19 + j43$

that are used throughout this work are introduced. The notation is close to the framework introduced by Voronenko and Püschel [41] which unifies many concepts from previous work.

2.4.1 Adder Graph Representation of Constant Multiplication

The adder circuits for constant multiplication as introduced above can be represented as a directed acyclic graph (DAG) [42] which is called *adder graph* in the following. Each node except the input node corresponds to an adder and has an in-degree of two. Each edge weight corresponds to a bit shift. An integer number can be assigned to each node of the graph which corresponds to the multiple of the input. This value is commonly called a *fundamental* [42]. The operation of each node can be fused into a single generalized add/subtract operation which is called $\mathcal{A}$-operation [41]:

Definition 1 ($\mathcal{A}$-operation)**.** *An $\mathcal{A}$-operation has two input fundamentals $u, v \in \mathbb{N}$ and produces an output fundamental*

$$\mathcal{A}_q(u, v) = |2^{l_u} u + (-1)^{s_v} 2^{l_v} v| 2^{-r} \tag{2.6}$$

where $q = (l_u, l_v, r, s_v)$ is a configuration vector which determines the left shifts $l_u,\ l_v \in \mathbb{N}_0$ of the inputs, an output right-shift $r \in \mathbb{N}_0$ and a sign bit $s_v \in \{0, 1\}$ which denotes whether an addition or subtraction is performed.

Note that output right shifts ($r > 0$) can alternatively be represented by allowing negative values for the input shifts, e. g., $l'_{u/v} = l_{u/v} - r$ ($l'_{u/v} \in \mathbb{Z}_0$), although it will be implemented differently in hardware. The adder graph of the circuit in Figure 2.2(c) using this representation is illustrated in Figure 2.4. The node values correspond to the fundamentals $w = \mathcal{A}_q(u, v)$, where u and v are the nodes connected at the input, the sign (s_v) is indicated by '+' or '−' and $l'_{u/v}$ are used as edge weights. Node '1' corresponds to the input terminal. For example, node '5' is realized by left shifting the input x by 2 bits and adding the unshifted input, leading to $2^2x + 2^0x = 5x$. Note that no explicit right shift r is used in this representation as it can be easily derived from negative edge values.

It was shown by Dempster and Macleod that any adder graph can be transformed to an unique adder graph with identical topology (and hence identical adder count) where each fundamental is represented by an odd fundamental, which they called an *odd fundamental graph* [42]. In addition, the MCM search can be typically restricted to only positive fundamentals without limiting the search space as they can be negated by changing adders to subtractors (of the adder graph or succeeding adders) and vice versa [42, 43]. Hence, given a set of target constants T to be optimized, the first step in any MCM algorithm is to compute the unique odd representation of their magnitudes

$$T_{\text{odd}} := \{\text{odd}(|t|) \mid t \in T\} \setminus \{0\} \ , \tag{2.7}$$

denoted as the target set, where odd(t) is equal to t divided by 2 until it is odd. Throughout this work, adder graphs are always assumed to be odd fundamental graphs.

2.4.2 Properties of the $\mathcal{A}$-Operation

In the following, some useful properties of the $\mathcal{A}$-operation are derived which are used in this work and follow the derivations from [41].

Lemma 1. *If there exists a configuration q with $w = \mathcal{A}_q(u, v)$ then there also exists a valid configuration q' for which $w = \mathcal{A}_{q'}(v, u)$.*

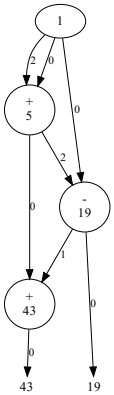


Figure 2.4: Adder graph realizing the multiplication with the coefficients $\{19, 43\}$

Proof. The relation $|a \pm b| = |b \pm a|$ can be used to rearrange (2.6) as follows

$$\begin{aligned} w &= \mathcal{A}_q(u, v) \\ &= |2^{l_u} u + (-1)^{s_v} 2^{l_v} v| 2^{-r} \\ &= |2^{l_v} v + (-1)^{s_v} 2^{l_u} u| 2^{-r} \\ &= \mathcal{A}_{q'}(v, u) \end{aligned} \tag{2.8}$$

with $q' = (l_v, l_u, r, s_v)$. □

Lemma 2. *If there exists a configuration q with $w = \mathcal{A}_q(u, v)$ then there also exists a valid configuration q'' for which $u = \mathcal{A}_{q''}(w, v)$.*

Proof. Rearranging (2.6) to u results in

$$u = \begin{cases} (2^r w + (-1)^{\overline{s_v}} 2^{l_v} v) 2^{-l_u} & \text{when } 2^{l_u} u + (-1)^{s_v} 2^{l_v} v \geq 0 \\ -(2^r w + (-1)^{s_v} 2^{l_v} v) 2^{-l_u} & \text{otherwise} \end{cases} \tag{2.9}$$

leading to

$$|u| = A_{q''}(w, v) \tag{2.10}$$

with $q'' = (r, l_v, l_u, s'_v)$ and

$$s'_v = \begin{cases} \overline{s_v} & \text{when } 2^{l_u} u + (-1)^{s_v} 2^{l_v} v \geq 0 \\ s_v & \text{otherwise} \end{cases} \tag{2.11}$$

□

2.4.3 $\mathcal{A}$-Sets

Besides the $\mathcal{A}$-operation, it is useful to define a set which contains all possible numbers that can be obtained from u and v by using one $\mathcal{A}$-operation:

Definition 2 ($\mathcal{A}_*(u,v)$-set)**.** *The $\mathcal{A}_*(u,v)$-set contains all possible fundamentals which can be obtained from u and v by using exactly one $\mathcal{A}$-operation:*

$$\mathcal{A}_*(u,v) := \{\mathcal{A}_q(u,v) \mid q \text{ is a valid configuration}\} \tag{2.12}$$

A valid configuration is a combination of l_u, l_v, r and s_v such that the result is a positive odd integer $\mathcal{A}_q(u,v) \leq c_{\max}$.

The limit $c_{\max}$ has to be introduced to keep the set finite. It is usually chosen as power-of-two value which is set to the maximum bit width of the target values plus one [41, 44],

$$c_{\max} := 2^{B_T+1} \tag{2.13}$$

with

$$B_T = \max_{t \in T_{\text{odd}}} \lceil \log_2(t) \rceil \ . \tag{2.14}$$

For convenience, the $\mathcal{A}_*$-set is also defined for two input sets $U, V \subseteq \mathbb{N}$, which contains all fundamentals that can be computed from the sets U and V by a single $\mathcal{A}$-operation:

$$\mathcal{A}_*(U,V) := \bigcup_{u \in U,\ v \in V} \mathcal{A}_*(u,v) \tag{2.15}$$

In addition, the $\mathcal{A}_*$-set is also defined for single input set $X \subseteq \mathbb{N}$ as follows:

$$\mathcal{A}_*(X) := \bigcup_{u,v \in X} \mathcal{A}_*(u,v) \tag{2.16}$$

Note that $\mathcal{A}_*(\{u,v\})$ is different from $\mathcal{A}_*(u,v)$ as the first one contains all combinations of u and v: $\mathcal{A}_*(\{u,v\}) = \mathcal{A}_*(u,u) \cup \mathcal{A}_*(u,v) \cup \mathcal{A}_*(v,v)$.

2.4.4 The MCM Problem

Using the graph representation above, the MCM problem can be stated as follows:

Definition 3 (MCM Problem)**.** *Given a set of positive odd target constants T, find a valid adder graph with minimum adders that realizes the multiplication with each constant of T.*

A *valid* adder graph is given if there exists a path from the input node '1' to each of the target nodes in T_{odd}. With the definition of the $\mathcal{A}$-operation in (2.6) and the odd representation (2.7) we can separate the core of the MCM problem formally as follows (according to [41]):

Definition 4 (Core MCM Problem)**.** *Given a set of positive odd target constants $T_{\text{odd}} = \{t_1, \ldots, t_M\}$, find the smallest set $R = \{r_0, r_1, \ldots, r_K\}$ with $T_{\text{odd}} \subseteq R$ and $r_0 = 1$ for which the elements in R can be sorted in such a way that for all triplets $r_i, r_j, r_k \in R$ with $r_k \neq r_i$, $r_k \neq r_j$ and $0 \leq i, j < k$, there is an $\mathcal{A}$-configuration p_k that fulfills:*

$$r_k = \mathcal{A}_{p_k}(r_i, r_j) \ . \tag{2.17}$$

In other words, the core of the MCM problem is to find the smallest intermediate set I such that a *valid* adder graph can be constructed from the nodes in $R = I \cup T_{\text{odd}}$. The elements from set I are denoted as non-output fundamentals (NOFs). When this set is found, it is a non-complex task to find the corresponding $\mathcal{A}$-configurations to build the graph. This can be done with the optimal part of the heuristics explained later in Section 2.6.2. Note that besides the minimum node count (i. e., number of adders), there also exist other cost metrics which employ the actual hardware costs in terms of full-adder cells or gate count [45–48].

2.4.5 MCM with Adder Depth Constraints

An important property of an adder graph is the so-called *adder depth (AD)* (sometimes referred to as *logic depth* or *adder-steps*). The *adder depth of node c*, denoted $\text{AD}(c)$, is defined as the maximum number of cascaded adders on each path from input node '1' to node c. The minimum possible adder depth of node c, denoted $\text{AD}_{\min}(c)$, can be obtained by using a binary tree of adders from the CSD representation of c [49]. It can be directly computed from the constant by

$$\text{AD}_{\min}(c) = \lceil \log_2(\text{nz}(c)) \rceil \tag{2.18}$$

where $\text{nz}(c)$ denotes the number of non-zeros of the CSD representation of c. Obviously, an adder graph with minimal AD may need more adders than the solution with minimal number of adders.

The adder depth plays an important role for the delay and the energy consumption of an MCM circuit [34, 50–55]. Thus, it is useful for many applications to limit the adder depth besides the minimization of adders. To reduce the overall delay it is sufficient to limit the worst-case adder depth of all nodes. The lowest possible adder depth is limited by the node(s) with maximum adder depth

$$D_{\max} = \max_{t \in T} \mathrm{AD}_{\min}(t) \ . \tag{2.19}$$

Limiting the adder depth to $D_{\max}$ leads to the lowest possible delay and to the corresponding optimization problem:

Definition 5 (MCM with Bounded Adder Depth ($\mathrm{MCM_{BAD}}$) Problem)**.** *Given a set of positive target constants $T = \{t_1, \ldots, t_M\}$, find an adder graph with minimum cost such that $\mathrm{AD}(t) \leq D_{\max}$ for all $t \in T$.*

The adder depth has an impact on the power consumption [34, 50–55]. It is based on the insight that a transition which is generated at the input or an adder output produces more transitions (glitches) in the following stages. If the adder depth is larger than needed, this produces more transitions than needed and, thus, a higher dynamic power consumption. Therefore, it is desirable for low power applications that *each* output node in the graph is obtained with minimal AD, which is defined as follows:

Definition 6 (MCM with Minimal Adder Depth ($\mathrm{MCM_{MAD}}$) Problem)**.** *Given a set of positive target constants $T = \{t_1, \ldots, t_M\}$, find an adder graph with minimum cost such that $\mathrm{AD}(t) = \mathrm{AD}_{\min}(t)$ for all $t \in T$.*

2.5 Field Programmable Gate Arrays

In this section, a brief overview of the architecture of FPGAs is given, as needed to introduce the mapping of the aforementioned adder graphs. More details about FPGA fundamentals can be found in textbooks (e. g., [56,57]), an introduction to the arithmetic features of more recent FPGAs is given in the thesis of Bogdan Pasca [58].

The first commercial FPGA in its current form was introduced by the XC2000 series of Xilinx Inc. in 1985 [56]. Its main difference compared to its programmable logic predecessors is the array-like organization of programmable logic elements (LEs) which are embedded in a programmable

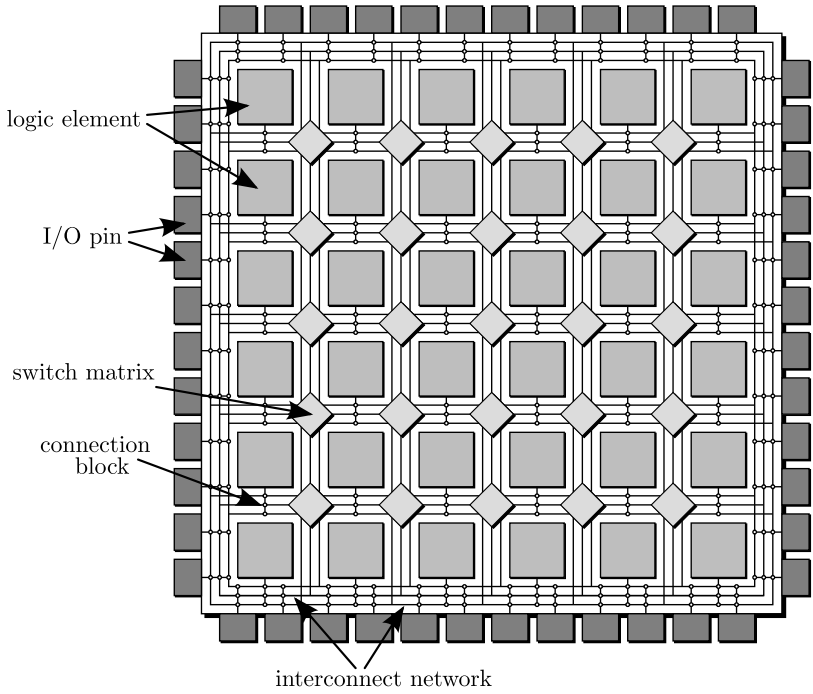


Figure 2.5: Simplified architecture of a generic FPGA layout

interconnect network consisting of switch matrices and connection blocks as indicated in Figure 2.5. The smallest common piece of logic of todays FPGAs typically consists of a look-up table (LUT), carry-chain logic and flip flops which is denoted as basic logic element (BLE) in the following. Each LE is typically built from a cluster of BLEs, i. e., several BLEs are combined with some local interconnect, to limit the input count. The interconnect or routing network allows the combination of several LEs to build much more complex logical circuits whose size is only limited by the resources of the FPGA. As both LEs and the interconnect network, are configurable at runtime, the logical circuit can be defined after manufacturing the chip. Due to this flexibility and low non-recurring engineering (NRE) cost, FPGAs became very popular as an alternative to ASICs. The huge overhead due to the programmability is partly compensated by the smallest and fastest technologies which are only affordable in very high volume productions. Hence,

todays FPGAs are reaching a complexity in the order of millions of BLEs and a speed close to one GHz.

While all FPGAs have an array-like structure, they have many differences in the detailed routing and LE structure. In the next two subsections, the LE structure is further detailed for recent commercial high-end FPGA devices from the two largest vendors: Xilinx and Altera. Of course, this limited number of FPGA architectures can not be comprehensive but gives a reasonable overview about the state-of-the-art.

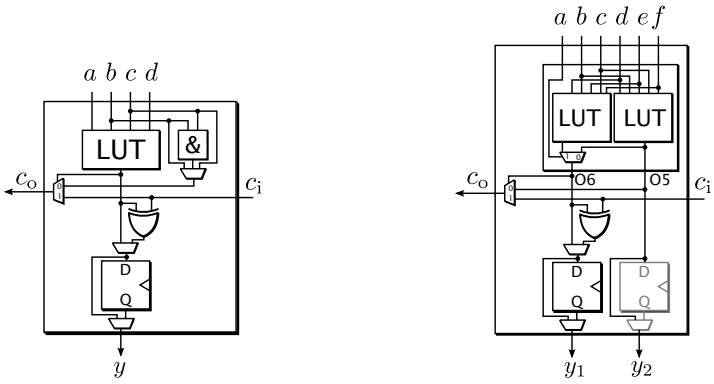
2.5.1 Basic Logic Elements of Xilinx FPGAs

On Xilinx FPGAs, the LE corresponds to a complex logic block (CLB) which contains several sub-units which are called slices. The Virtex 4 architecture contains four slices per CLB. Each Virtex 4 slice contains two BLEs consisting of a 4-input LUT, a carry-chain logic and a single flip flop (FF) [59]. The simplified structure of a Virtex 4 BLE is shown in Figure 2.6(a). It can realize any Boolean function of four input variables (*a*-*d*), followed by an optional register (selected by the multiplexer (MUX) close to the output y). Alternatively, the dedicated carry logic consisting of an exclusive-or (XOR) gate and a MUX can be used to build a fast full adder with dedicated carry-in (c_i) and carry-out (c_o) connections to other BLEs to build a ripple carry adder (RCA). For this, the LUT has to be configured as XOR gate between b and c and one of these ports has to be connected to the zero-input of the MUX. The AND gate can be used for generating partial products as needed in generic multiplications.

Starting from the Virtex 5 architecture, the LUT was extended to a 6-input LUT per BLE which can be configured into two independent 5-input LUTs that share the same inputs as shown in Figure 2.6(b) [60]. Each slice now contains four identical BLEs. In the Virtex 6, the Spartan 6 and the latest 7 series FPGA architectures, an additional FF was added (shown gray in Figure 2.6(b)) [61–63]. They also contain the same carry logic as the Virtex 4 but the LUT bypass and the AND gate are now realized using the second 5-input LUT.

2.5.2 Basic Logic Elements of Altera FPGAs

Recent Altera Stratix FPGAs are organized in logic array blocks (LABs) which contain sub-units called ALM. The Stratix III to Stratix V FPGAs



(a) Virtex 4 BLE (half slice)

(b) Virtex 5/6, Spartan 6 and 7 Series BLE (quater slice)

Figure 2.6: Simplified BLE of Xilinx FPGA families

contain ten ALMs per LAB [64–66]. The functionality of a BLE as defined above is roughly covered by one half of an ALM. The structure of this Altera BLE mainly consists of one 4-input LUT, two 3-input LUTs, a dedicated FA and two FFs as shown in Figure 2.7. The second FF (shown gray in Figure 2.7) was introduced with the Stratix V architecture and is missing in older Stratix FPGAs. Each ALM has eight input lines which have to be shared between two BLEs. Multiplexers between both BLEs in the same ALM can be used to adapt the logic granularity to different effective LUT sizes (which is called adaptive LUT (ALUT)). The possible configurations of a complete ALM are two independent 4-input LUTs, two independent 3-input and 5-input LUTs, one independent 6-input LUT, various combinations of LUTs up to six inputs where some of the inputs are shared and some specific 7-input LUTs [64–66].

2.5.3 Mapping of Adder Graphs to FPGAs

As mentioned above, BLEs of modern FPGAs directly contain dedicated logic and routing resources to build fast ripple carry adders (RCAs). An example configuration of a small 4-bit RCA using the CLBs of a Xilinx Virtex 4 FPGA is given in Figure 2.8(a). Each BLE computes the Boolean

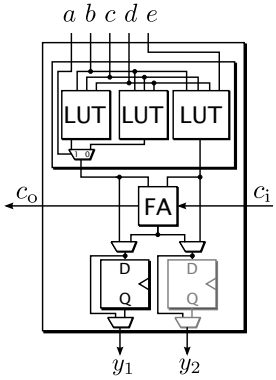


Figure 2.7: Simplified BLE (half ALM) of Altera Stratix III to Stratix IV FPGAs

relations

$$s_k = a_k \oplus b_k \oplus c_k \tag{2.20}$$

$$c_{k+1} = (a_k \oplus b_k)c_k + \overline{a_k \oplus b_k}a_k \tag{2.21}$$

$$= a_k c_k + b_k c_k + a_k b_k \tag{2.22}$$

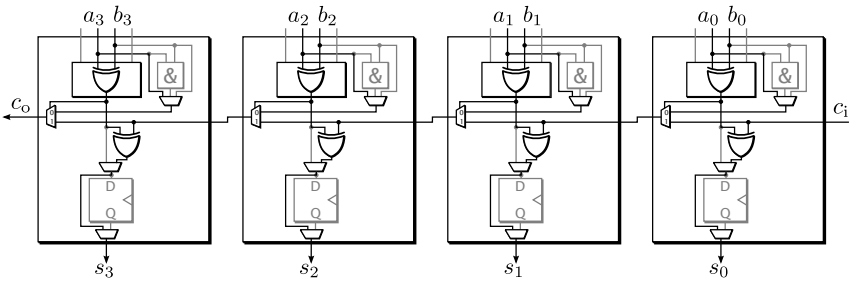
$$\tag{2.23}$$

which correspond to the function of a full adder (FA).

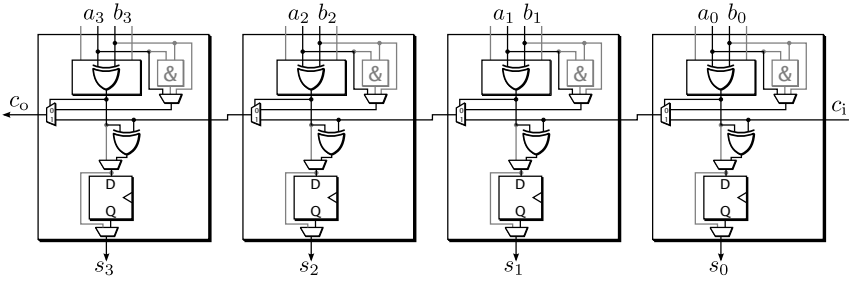
Each RCA is pretty fast due to the fast carry chain multiplexers. An RCA with B_o output bits is compact and its critical path, which is typically from the LSB input to the most significant bit (MSB) output, consists of the delay of one LUT (denoted τ_{LUT}), $B_o - 1$ times the carry-chain of a single BLE (denoted τ_{CC}), and one XOR gate (denoted τ_{XOR}) [67]:

$$\tau_{RCA}(B_o) = \tau_{LUT} + (B_o - 1)\tau_{CC} + \tau_{XOR} \tag{2.24}$$

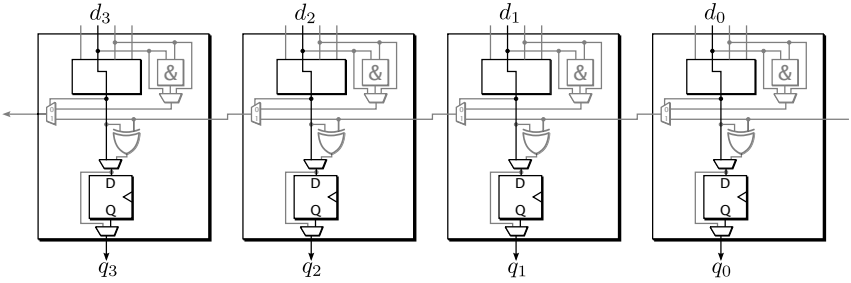
To give some example values, consider the Virtex 6 FPGA architecture. They are available with different performance grades which are called *speed grades*. The carry-in to carry-out delay of a complete slice with four BLEs for the fastest speed grade of 3 is specified to be at most 0.06 ns [68]. Hence, each BLE contributes with $\tau_{CC} = 0.015$ ns to the carry delay. The relevant combinational delay of a LUT ranges between $\tau_{LUT} = 0.14 \ldots 0.32$ ns depending which input influences which output [68]. The delay of the XOR (from carry-in to one of the BLE outputs within a slice) is $\tau_{XOR} = 0.21 \ldots 0.25$ ns. For this example, the worst-case delays of 8, 16 and a 32 bit adders are about 1 ns, 1.5 ns and 2.5 ns, respectively.



(a) 4-bit adder



(b) 4-bit registered adder



(c) 4-bit register

Figure 2.8: BLE configurations for pipelined adder graphs on Xilinx Virtex 4 FPGAs

Note that it is well known that in general VLSI designs the RCA is the most compact adder structure but also the slowest one [69, 70]. Much faster adder structures are known but most of them are counter productive on FPGAs due to the dedicated fast carry logic. The only exception are large word lengths of at least 32 bit and more [67, 71, 72].

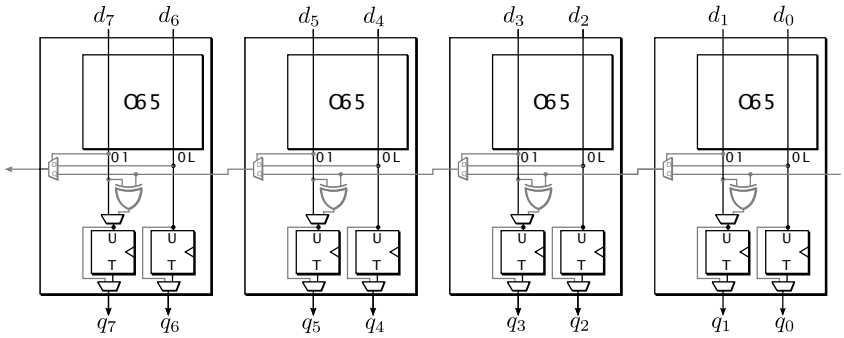


Figure 2.9: BLE configuration for an 8 bit register on Xilinx Virtex 6 FPGAs

Clearly, each node of an adder graph can be mapped to such an RCA structure. While in VLSI realizations the delay of local wires can often be neglected, a connection between two LEs on an FPGA typically includes several routing elements which significantly affects the overall delay of the circuit. A typical local routing delay on a Virtex 6 FPGA is in the order of $0.3 \ldots 0.5$ ns (obtained from timing analyses of representative designs). Thus, a one-to-one mapping of an adder graph to BLEs accumulates the adder and routing delays with the consequence that the speed potential of the FPGA is by far not exhausted. The common countermeasure to break down the critical part is pipelining [73]. To give an example of the potential speed, carefully pipelined designs can run at 600 MHz and above on a Virtex 6 FPGA. Here, even a single local routing connection in the critical path consumes about one third of the clock period of 1.66 ns.

Pipelining of non-recursive circuits like the adder circuits under consideration is done by breaking the combinational paths and placing additional FFs in between such that each resulting path from an input to an output has the same number of pipeline FFs [73]. The number of FFs in each path from input(s) to output(s) is called the *pipeline depth* of the circuit. Pipelining only increases the latency by the pipeline depth and keeps the functionality of the circuit. Pipelining an RCA on an FPGA is cost neutral as the otherwise unused FFs are simply used as shown in Figure 2.8(b).

However, to guarantee that each path has the same number of pipeline FFs, additional registers have to be placed to balance the pipeline. The BLE configuration of a pure register with 4 bit is shown in Figure 2.8(c). Clearly, on this FPGA architecture, a register consumes exactly the same resources in terms of BLE resources as an adder of the same word size (combinational

or registered). On modern FPGAs each BLE can realize two FF leading to twice the number of bits for registers as illustrated in Figure 2.9.

However, as many of these pipeline balancing registers may be necessary, it is important to consider them in the adder graph optimization. To give a first motivating example, consider again the adder graph of the MCM solution for multiplying with the constants $\{19, 43\}$ of Figure 2.4 which is repeated in Figure 2.10(a). It was obtained by the $\mathrm{H}_{\mathrm{cub}}$ algorithm [41] and is optimal in terms of the number of required adders (as its adder count is equal to the lower-bound given in [74]). To pipeline this adder graph, each adder is augmented by a pipeline register and three additional registers are required to balance the pipeline. This is illustrated as pipelined adder graph (PAG) in Figure 2.10(b). Here, each node with a rectangular shape includes a pipeline register, i. e., two-input nodes marked with '+' or '–' are pipelined adders and subtractors, respectively, while single input nodes are pure registers. Each node in the PAG corresponds to similar BLE costs. As illustrated in Figure 2.10(c), there obviously exists a solution with less nodes. This PAG will typically use less BLEs although more adders are used.

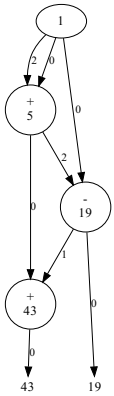
Note that optimizing PAGs is complementary to methods that increase the speed of a single adder, e. g., deeply pipelined adders [67]. In contrast to this, some registers can be saved and the latency can be reduced at the cost of a reduced throughput by skipping some of the stages. Both cases can be considered by adjusting the cost of the registers and adders in each stage, the core optimization problem remains the same (the details are elaborated in Section 5.4).

2.5.4 Cost Models for Pipelined Adders and Registers

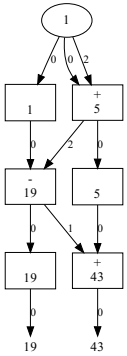
Assuming that each BLE is able to realize a single FA, a single FF or a combination of both, a pipelined RCA with B_{o} output bits requires exactly B_{o} BLEs. The output word size of a node computing a factor w of the input requires $\lceil \log_2(w) \rceil$ bits in addition to the input word size of the MCM block. Thus, the cost in terms of the number of BLEs for a *pipelined* $\mathcal{A}$-operation can be obtained by

$$\mathrm{cost}_{\mathrm{A}}(u, v, w) = B_{\mathrm{o}} = B_{\mathrm{i}} + \lceil \log_2(w) \rceil + r \tag{2.25}$$

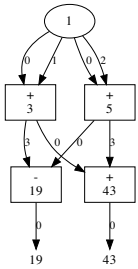
where B_{i} is the input word size of the MCM block and $r \in q$ is the right shift required to obtain $w = \mathcal{A}_q(u, v)$ according to (2.6). Note that there may be a significant difference in the cost compared to the *non-pipelined* $\mathcal{A}$-operation as FFs can be replaced by wires.



(a) Optimal adder graph



(b) Optimal adder graph after pipelining



(c) Optimal pipelined adder graph

Figure 2.10: Adder graphs realizing the multiplication with the coefficients $\{19, 43\}$

The right shift r has to be considered in the cost as additional half adders for the carry propagation of LSBs are required if right shifts are used. Take for example the computation of $5x = (3x + 7x)2^{-1} = 10x2^{-1}$ using $r = 1$. It will require $\lceil \log_2(10) \rceil = 4$ additional bits (on top of B_i) for its computation but only $\lceil \log_2(5) \rceil = 3$ additional bits are used to store the right shifted result. As right shifts occur relatively seldom and it takes effort to compute the right shift value for the cost evaluation, a good approximation is obtained by ignoring possible right shifts:

$$\mathrm{cost_A}(w) = B_\mathrm{i} + \lceil \log_2(w) \rceil \tag{2.26}$$

The register costs are modeled in a similar way as again $B_\mathrm{i} + \lceil \log_2(w) \rceil$ FF are used for storage:

$$\mathrm{cost_R}(w) = \left(B_\mathrm{i} + \lceil \log_2(w) \rceil\right) / K_\mathrm{FF} \tag{2.27}$$

Here, K_FF represents the number of FFs per BLE. Throughout the thesis, this cost model at BLE level is called the *low-level* cost model. Simply counting the adders and pure registers (i. e., $\mathrm{cost_A}(u, v, w) = 1$ and $\mathrm{cost_R}(w) = 1$) is called the *high-level* cost model. Note that the low-level model described above is different from the number of FAs compared to previous work on non-pipelined adder graphs [45–48] as additional BLEs are required for FFs in the

pipelined case. The cost of the total adder graph, denoted as cost_{PAG}, corresponds to the sum of the cost values of each node. Note that some FPGA vendors provide efficient register chains that use the flip flops that usually store the LUT content in a very efficient way (SRL16/SRLC32E primitives on Xilinx FPGAs [75]). This will be considered in detail in Section 5.4.

For ASICs, a good low-level approximation for RCA-based PAG implementations is given by

$$\text{cost}_{\text{R}}(w) = c_{\text{FF}}(B_{\text{i}} + \lceil \log_2(w) \rceil) \tag{2.28}$$

$$\text{cost}_{\text{A}}(w) = (c_{\text{FA}} + c_{\text{FF}})(B_{\text{i}} + \lceil \log_2(w) \rceil) \tag{2.29}$$

where c_{FF} and c_{FA} correspond to the area cost of a single FF and FA, respectively. The number of FAs might be slightly larger compared to the more sophisticated models [45–48] but gives a good approximation.

2.6 Related Work

The related work of SCM and MCM is given in the following. Related work to specific topics is discussed in the corresponding section when necessary.

2.6.1 Single Constant Multiplication

The idea to perform the multiplication as a sum of power-of-two values is pretty old. The first known documentation of such a scheme is the mathematical Papyri Rhind of Ahmes which dates back to the seventeenth century B.C. The method is known as the Egyptian multiplication (sometimes also referred to as Russian multiplication) [76]. One operand is decomposed into a sum of power-of-two values by repeated subtraction of the largest power-of-two. The other operand is duplicated multiple times and the corresponding duplicates are selected and added. Take, for example, the multiplication of 77 times 89. The largest power-of-two term less than 77 is 64, the next one less than $77 - 64 = 13$ is 8, etc., leading to $77 = 64 + 8 + 4 + 1 = 2^6 + 2^3 + 2^2 + 2^0$. Then, the other operand is duplicated by multiple additions, e. g., $2 \cdot 89 = 89 + 89 = 178$, $4 \cdot 89 = 178 + 178 = 356$, $8 \cdot 89 = 356 + 356 = 712$, etc. The multiplication result is obtained by adding the multiples corresponding to the power-of-two decomposition of the first operand, i. e., $77 \cdot 89 = (64 + 8 + 4 + 1) \cdot 89 = 5696 + 712 + 356 + 89 = 6853$.

One can say that the ancient Egyptians implicitly used the binary number system which is nowadays the base of most digital arithmetic.

The use of signed digits to simplify multiplication as well as division can be dated back to 1726 [77]. Its efficient use in digital systems became popular in the 1950'th [26–28]. Later, the complexity was further reduced by the use of common subexpression elimination (CSE) of repeated patterns in the number representation [12,33]. An additional step towards fewer complexity was the graph based representation of constant multiplications as used in digital filters [78]. Adder graph based representations yield to optimal SCM circuits (in terms of the number of additions) which were exploited by the minimized adder graph (MAG) algorithm of Dempster and Macleod for up to four adders covering all 12 bit coefficients [42]. Later, these results were enhanced by introducing simplifications by Gustafsson et al. to five adders covering all coefficients up to 19 bit [79, 80]. Further extensions for up to six adders resulted in optimal SCM circuits for coefficients up to 32 bit [81, 82]. In the latter work, not all graphs are generated exhaustively but the minimal representation of a given constant is obtained by the graph topology tests introduced by Voronenko and Püschel [41]. Although it was shown by Cappello and Steiglitz that the SCM problem is NP-complete [83], the SCM problem can be regarded as solved for most of the practical applications (which typically include coefficients with less than 32 bit).

2.6.2 Multiple Constant Multiplication

The MCM problem is a generalization of the SCM problem and has been an active research topic for the last two decades. Many different optimization methods have been proposed [7, 9, 12, 33, 41, 43–45, 48, 49, 78, 84–101]. There are two fundamentally different concepts for solving the MCM problem, in the first concept, intermediate values are obtained from common subexpressions [12, 33, 89, 92–94], in the second concept, graph based methods are used [7, 9, 41, 43, 44, 49, 78, 84–91, 95–101]. CSE methods search for identical patterns in the binary or signed digit number representation of the constant(s). As demonstrated in Section 2.1.3, this method may fail to find a minimal solution due to hidden non-zeros [34, 102]. Some of these hidden non-zeros can be extracted using the method proposed by Thong and Nicolici but some of them may remain hidden [102].

The hidden non-zero problem is inherently avoided in graph based methods as they are not restricted to the number representation. A detailed comparison between optimal graph based methods and CSE with different

number representations (without considering hidden non-zeros) can be found in the work of Aksoy et al. [99]. On average, 10.8 additional adders were required for an exact CSE-based solution compared to the exact graph based solution in an experiment with 30 MCM instances with 10 to 100 random coefficients with 14 bit each. However, as the search space is much larger for graph based methods, it can still be advantageous to use CSE methods for large problems [103].

As the MCM problem is a generalization of the SCM problem it is also NP-complete. Hence, even if there exist optimal methods, there is a need of good heuristics for larger problem sizes. The n-dimensional reduced adder graph (RAG-n) algorithm, introduced by Dempster and Macleod [44], was one of the leading MCM heuristics for many years. This situation changed significantly with the introduction of the $\mathrm{H_{cub}}$ algorithm by Voronenko and Püschel [41] and the introduction of the difference adder graph (DiffAG) algorithm by Gustafsson [43], both in 2007. Both state-of-the-art MCM heuristics are briefly introduced in the following.

The $\mathrm{H_{cub}}$ algorithm starts, like most other MCM algorithms, with an algorithm that is called the *optimal part* of the MCM optimization. A pseudocode of this part is given in Algorithm 2.1. The input to the algorithm is the target set T containing all coefficients. First, the unique odd representation is computed (line 2). Then, the so-called *realized set* R is initialized to a set containing only the element '1'. Next, the so-called *successor set* S is computed, which contains all odd fundamentals which can be computed by one addition from the realized set (line 5). It is now checked if new target elements can be realized and the corresponding target elements are stored in set T' (line 6). These elements (if any) are inserted into R and removed from T (lines 7 and 8). This procedure is repeated until no further target element can be realized (then, T' is empty). If the target set T is also empty, it is known that the solution is optimal (each target can be computed by one adder or subtractor). If elements in T remain, at least one additional NOF from the successor set has to be inserted into R to compute the remaining elements of T. For that, adder graph topologies with up to three adders are evaluated in the $\mathrm{H_{cub}}$ algorithm to obtain or estimate (in case that more than three additional adders are necessary) the so-called $\mathcal{A}$-distance of all possible successors in S. The $\mathcal{A}$-distance (which is equivalent to the adder distance [44]), denoted as $\mathrm{dist}(R, c)$, is defined as the minimum number of $\mathcal{A}$-operations necessary to compute c from R. The main idea is to select the

Algorithm 2.1: The optimal part of the MCM optimization

```
1 R = MCM_opt(T)
2   T ← {odd(|t|) | t ∈ T} \ {0}
3   R := {1}
4   do
5     S ← A_*(R)
6     T' ← T ∩ S
7     R ← R ∪ T'
8     T ← T \ T'
9   while T' ≠ ∅
```

successor s leading to the best benefit

$$B(R, s, t) = \text{dist}(R, t) - \text{dist}(R \cup \{s\}, t) \tag{2.30}$$

for all remaining target coefficients $t \in T$ (in fact, a slightly modified *weighted* benefit function is used [41]). The best successor s is included in R and the optimal part of the algorithm is evaluated again which moves elements from T to R. This procedure is continued until T is empty. The implementation of the H_{cub} algorithm is available online as source code [104]. This implementation includes a switch to solve the MCM problem with bounded adder depth, denoted as $\text{H}_{\text{cub,BAD}}$ in the following. The idea of H_{cub} was later extended in the H3 and H4 heuristics by Thong and Nicolici [100]. The key improvements were obtained by computing the adder distance exactly for H4 and using better estimators for larger distances in both heuristics. With that they obtained better results and less runtime (due to a restriction of $c_{\max}$) compared to H_{cub} for problems with larger word length and low coefficient count.

The DiffAG algorithm [43] follows a different strategy. It basically starts with the optimal part as given in Algorithm 2.1. Then, an undirected complete graph is formed where each node corresponds to a set of coefficients, initially one node for the realized set $N_0 = R$ and for each remaining target element, one set containing the target element $N_i = \{t_i\}$ for all $t_i \in T_{\text{odd}} \setminus R$. Now, the so-called *differences* are computed between each pair of nodes in the graph and are inserted into the difference sets D_{ij}. A difference $d_{ij} \in D_{ij}$ is a fundamental that allows – when inserted into R – to compute the elements in N_i from the elements in N_j and vice versa. Each edge (i, j) corresponds to one set of differences D_{ij}. Formally, for each $n_i \in N_i$ and $n_j \in N_j$, there exists an $\mathcal{A}$-operation such that $n_i = \mathcal{A}_q(n_j, d_{ij})$ and, due to Lemma 2 (see

page 16), $n_j = \mathcal{A}_{q'}(n_i, d_{ij})$. The difference set can be directly computed from the node values by $D_{ij} = \mathcal{A}_*(N_i, N_j)$. If there exists a difference which is included in the realized set ($d_{ij} \in R$), the corresponding nodes are merged into a single node $N_k = N_i \cup N_j$, N_i and N_j are removed and the corresponding edges and difference sets are merged. If no difference remains that is included in the realized set, the idea is to realize the least-cost and most frequent difference. This process is continued until all nodes in the graph are merged into a single node which set corresponds to the final solution. The results in [43] show that the DiffAG algorithm is advantageous for large target sets with low coefficient word size compared to the $\mathrm{H}_{\mathrm{cub}}$ algorithm.

Besides the heuristic approaches discussed above, a lot of research effort has been taken towards optimal MCM methods in the last years [9, 96–100]. An ILP-based method for finding the common subexpressions in an MCM problem was proposed by Aksoy et al. [96]. However, as mentioned above, this will not necessarily lead to the least adder solution due to the number representation dependency. Another approach for an optimal MCM method using ILP was proposed by Gustafsson [97]. Here, the MCM problem is formulated as the problem of finding a Steiner hypertree in a directed hypergraph. A hypergraph is an extended graph, where each edge, called hyperedge, can connect more than two nodes. A Steiner hypertree is a straight forward extension of the Steiner tree, which is defined as a minimum weight tree spanning a given set of vertices using some additional vertices (if required), to hypergraphs [105]. The details of this approach will be discussed in Section 5.1. It finds optimal solutions and can be adapted to several additional MCM constraints like minimal adder depth or fanout restrictions. However, due to its computational complexity it is only suitable for small MCM instances or to find better lower bounds (e. g., compared to [74, 106]) by relaxing the model to a continuous LP problem. Optimal breadth-first search and depth-first search algorithms based on branch-and-bound for the graph-based MCM problem were proposed by Aksoy et al. [9, 98, 99], leading to the first optimal MCM method for real-world FIR filter applications. A similar approach that can handle problems with larger word lengths but less coefficients was used by the bounded depth-first search (algorithm) (BDFS) algorithm introduced by Thong and Nicolici [100].

2.6.3 Modified Optimization Goals in SCM and MCM Optimization

Besides the reduction of adders in the SCM/MCM problem, attention was paid to different optimization goals related to the circuit implementation in more recent work. As already mentioned above, the power consumption is related to the adder depth of the adder graph, so many optimization algorithms solving the MCM_{MAD} problem were proposed [34, 50–55, 107–109]. Besides the adder depth, the glitch path count (GPC) was proposed as a more accurate measure to estimate a power equivalent at high level [50]. Here, different paths from the input to each adder are counted which is approximately proportional to the dynamic power. The GPC power estimation was further refined to the glitch path score (GPS) [52].

Another important optimization goal that considers the bit-level cost was first proposed by Johansson et al. [45, 46] and later used by Aksoy et al. [47] and Brisebarre et al. [48]. They observed that in computations like $w = 2^{l_u}u + v$, the l_u lowest bits of w are identical to the l_u lowest bits of v and, hence, a lower number of full adders is sufficient. Their models consider the bit shifts of all the cases that can occur in an adder graph leading to an accurate number of full adders, half adders and even inverters [47]. A bit-level model can also be used to reduce the critical path delay as demonstrated recently by Lou et al. [110, 111].

Besides carry propagate adders (CPAs) like the ripple-carry adders, carry-save adders (CSAs) can be used to reduce the delay as it is usually done in parallel multipliers. But the number of CSAs is different to the number of CPAs for the same adder graph as there is no 1:1 mapping possible (some adders can be replaced by wires while others require two CSAs). Hence, another modified optimization goal is to reduce the number of CSAs in MCM [112–115] or digital filters [116, 117].

2.6.4 Pipelined Multiple Constant Multiplication

Another method to increase the speed of a combinational circuit is pipelining [73]. It was mentioned in an early work of Hartley that *"it is important to consider not only the number of adders used, but also the number of pipeline latches, which is more difficult to determine"* [33]. As discussed earlier, this is in particular important on FPGAs as the routing delay is much larger compared to ASICs. Hence, Macpherson and Stewart proposed

the reduced slice graph (RSG) algorithm [118] which is an FPGA-specific MCM optimization algorithm based on ideas of the RAG-n [44] and MAG [42] algorithms. Their aim is to reduce the adder depth as this reduces the number of additional registers to balance the pipeline. For that, they obtained optimal SCM solutions from the MAG algorithm but used the adder depth as primary selection metric and the adder count only as a secondary metric. Then, in contrast to RAG-n, non-trivial coefficients with highest cost are realized using the known SCM solution. Next, the optimal part of RAG-n (see Algorithm 2.1) is applied to find out if other coefficients can be realized using the new NOF. This procedure is repeated until all coefficients are realized. The solution typically has a larger adder count but a lower adder depth, leading to less registers after pipelining. They compared their results with parallel distributed arithmetic (DA) implementations which are often called to be perfectly suited for FPGAs as they are realized using LUTs and adder arithmetic only. They showed that considerable logic reductions are possible by using their method compared to DA.

The impact of adder graph pipelining was further quantified by Meyer-Baese et al. [119] who showed that average speedups of 111% can be achieved by a very moderate area increase of 6%. They used adder graphs obtained by the RAG-n algorithm [44] with a hand-optimized pipeline. They also compared their designs to parallel DA implementations and quantified the resource reductions to 71% on average.

Another optimization method that targets pipelined MCM circuits for FPGAs is the 'add and shift' method proposed by Mirzaei et al. [5, 6, 120]. They used a CSE algorithm to extract non-output fundamentals while keeping the AD minimal. Again, this reduces additional pipeline registers as a side effect. They reported average slices reductions of 50% compared to parallel DA.

An algorithm that already considers the cost of the pipeline registers during the optimization was first proposed by Aksoy et al. [7]. They used an iterative method, called HCUB-DC+ILP, where each iteration works as follows: First, an MCM solution is obtained by using the $\mathrm{H}_{\mathrm{cub,BAD}}$ algorithm with an iteration-specific random seed. The resulting coefficients (target and intermediate values) are inserted into the *realized set* R. In addition, all cost-1 coefficients of the form $2^k \pm 1$ up to the maximum word size of the target coefficients are also inserted into R. Then, all possible $\mathcal{A}$-operations to realize the target coefficients in R are computed and the best combination (in terms of gate-level area [121]) is searched using a binary integer linear programming (BILP) method which also considers the cost for pipeline reg-

isters. Finally, the pipeline method of [118] is applied and the costs are obtained and stored if they are better than any solution before. The next iteration starts with a different random seed to obtain a (possibly) different MCM solution and the results are added to R. The optimization stops when a fixed size of R is reached or if $\mathrm{H}_{\mathrm{cub,BAD}}$ does not find new solutions for a fixed number of iterations. Their target was a 0.18μm standard cell library for an ASIC design. They achieved a 5.6% area reduction on average compared to $\mathrm{H}_{\mathrm{cub,BAD}}$ solutions, which were automatically pipelined by the retiming of the logic synthesis tool (Cadence Encounter RTL Compiler). Compared to the non-pipelined solution (by removing the registers), the speedup is 34.0% which comes along with an area increase of 55.1% for the pipeline registers.

As a side effect, pipelining also reduces the logic depth and the power consumption of a circuit [122]. The pipeline registers "isolate" power producing glitches from one logic stage to the next yielding low power designs similar to designs obtained by the operand isolation technique introduced by Correale [123]. In MCM circuits, power consumption reductions of 43.0% due to pipelining were reported even though the area was increased [7]. The effect of the power reduction due to pipelining is expected to be even larger on FPGAs due to the programmable routing [124]. Wilton et al. [124] reported reductions between 40% and 90% in generic logic circuits. Hence, pipelining is a method to improve both figure of merit axes "performance" and "energy".

http://www.springer.com/978-3-658-13322-1

Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays

Kumm, M.

2016, XXXIII, 206 p. 47 illus., Softcover
ISBN: 978-3-658-13322-1