Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2014, Article ID 564924, 14 pages
http://dx.doi.org/10.1155/2014/564924

Research Article

Architecture and Application-Aware Management of Complexity of Mapping Multiplication to FPGA DSP Blocks in High Level Synthesis

Sharad Sinha[1,2] and Thambipillai Srikanthan[1]

[1] *School of Computer Engineering, Nanyang Technological University, Singapore 639798*
[2] *The Hong Kong University of Science and Technology, Hong Kong*

Correspondence should be addressed to Sharad Sinha; sharad_sinha@pmail.ntu.edu.sg

Received 9 May 2014; Revised 22 September 2014; Accepted 7 October 2014; Published 21 October 2014

Academic Editor: Michael Hübner

Multiplication is a common operation in many applications and there exist various types of multiplication operations. Current high level synthesis (HLS) flows generally treat all multiplication operations equally and indistinguishable from each other leading to inefficient mapping to resources. This paper proposes algorithms for automatically identifying the different types of multiplication operations and investigates the ensemble of these different types of multiplication operations. This distinguishes it from previous works where mapping strategies for an individual type of multiplication operation have been investigated and the type of multiplication operation is assumed to be known *a priori*. A new cost model, independent of device and synthesis tools, for establishing priority among different types of multiplication operations for mapping to on-chip DSP blocks is also proposed. This cost model is used by a proposed analysis and priority ordering based mapping strategy targeted at making efficient use of hard DSP blocks on FPGAs while maximizing the operating frequency of designs. Results show that the proposed methodology could result in designs which were at least 2× faster in performance than those generated by commercial HLS tool: Vivado-HLS.

1. Introduction

FPGAs are now available with on-chip DSP blocks for high-performance multiplication and allied operations like multiplication followed by an addition/subtraction. For instance, multiplication followed by an addition is typically found in finite impulse response (FIR) filters. The work presented in this paper looks at mapping different kinds of multiplications and allied operations, which are automatically identified in an application, efficiently to on-chip DSP blocks based on a cost measure in order to improve performance as well as save FPGA logic for other uses.

Frequently executed code blocks in an application, that have already been identified through established methods like code profiling, can be further analyzed for multiplication and allied operations for mapping to available DSP blocks on the FPGA based on a priority ordering of these operations in order to maximize performance.

This main contributions of the work presented in this paper are as follows.

(i) An Algorithm 1 called "*autoidentify*" which automatically identifies the different kinds of multiplication and allied operations in an application described in C/C++.

(ii) An Algorithm 2 which then maps the operations identified by Algorithm 1 to DSP blocks in a prioritized order based on a cost measure to maximize performance. This automatic mapping with the aim to maximize performance is crucial as this exercise pipelines the DSP blocks to maximize performance. This is what a hardware designer would typically do instead of relying on inference by downstream physical synthesis tools which have limited ability to do pipelining. Algorithm 2 abstracts this hardware design viewpoint and automates it for HLS.

Input: A wordlength optimized dataflow graph (DFG)
Output: Consolidated list of multiplication operations in three categories
(1) Perform *Breadth First Search* (BFS) on DFG
(2) At each level of DFG
(3) ***If*** a constant value node is found
(4) Perform a *Depth First Search* (DFS) with constant value node as root node to find its successor node
(5) ***If*** the second input to successor node is labeled a variable, store the variable identity and the constant in a hash table: H(MUL)
(6) ***EndIf***
(7) ***EndIf***
(8) **End** BFS
(9) Identity (ID) = 0
(10) **For** each multiplication node
(11) Check if successor node, SN is addition or subtraction else Step 9
(12) Check if successor node SNN of SN is addition or subtraction
(13) **If** (11) is ***TRUE***
(14) Check if second operand source of SNN is a MUL node
(15) **If** (13) is ***TRUE***
(16) Collect all nodes in 9, 10, 11, and 13 as a partial graph
(17) SN = SNN
(18) Repeat 11 to 16 until (11) or (13) returns ***FALSE***
(19) Chain Detected = Yes
(20) Length of chain = Number of ADD/SUB nodes
(21) Chain ID = ID + 1
(22) **EndFor**
(23) Remove from H(MUL) all MUL nodes collected in Step 15.

Algorithm 1: Autoidentify algorithm for identification of multiplication and allied operations.

Input: A collection of multiplication operations of three different types obtained from Algorithm A
Output: Ordered of multiplication operation instances identified for mapping to DSP blocks.
(1) Calculate the number of FAs required using Tables 4, 6, and 8 and (4)
(2) Calculate the number of AND gates required using (3)
(Note that no AND gate is required when multiplying with constants)
(3) Calculate "*Equivalent LUT Cost*" for all SCM and variable-variable multiplication operations using (5)
(4) Calculate "*Equivalent LUT Cost*" for different sets of MCM operations using (5)
(4.1) In each set, select the operation with the maximum "*Equivalent LUT Cost*" that is "*MAX(Equivalent LUT Cost)*".
(5) Calculate "Equivalent LUT Cost" for different MULT-(ADD/SUBTRACT) Chains using (5)
(5.1) In each set, select the multiplication operation with the maximum "*Equivalent LUT Cost*" that is "*MAX(Equivalent LUT Cost)*".
(6) Remove multiplication operations part of MULT-(ADD/SUBTRACT) chains from the H(MULT) hash table
(7) Arrange multiplication operations from 5.1 and 6 in descending order of "*Equivalent LUT Cost*" as per Steps (3), (4.1) and (5.1) to form list *D*
(8) Map multiplication operations from the top in list *D* to DSP blocks until DSP blocks are exhausted
(9) Map remaining multiplication operations to LUTs

Algorithm 2: Joint comparison and selection (JCS) algorithm.

(iii) A new cost model for evaluating different types of multiplication operations for the purpose of comparison in order to select which ones to prioritize to mapped to on-chip DSP resources. This cost model is independent of device and synthesis tools and is hence applicable to current as well as future devices as well as future version of synthesis tools.

The remainder of the paper is organized as follows. Section 2 discusses the different types of multiplication operations and the automatic identification Algorithm 1; Section 3 discusses prior work related to mapping of different kinds of multiplication operations, introduces on-chip DSP blocks in modern FPGAs, provides the cost model for prioritizing these operations, and discusses the proposed priority based

selection and mapping Algorithm 2. Experiments and results are discussed in Section 4 while Section 5 concludes the paper.

2. Automatic Identification of Multiplication and Allied Operations

In this section, the different kinds of multiplication and allied operations that could exist in an application are discussed. Also, the algorithm to identify these operations in an application without designer intervention is presented.

2.1. Types of Multiplication and Allied Operations in Applications. Multiplication on FPGAs is a costly operation in terms of area and requires careful timing considerations. Adding to this complexity is the fact that there are three major kinds of multiplication operations, namely, (1) two variables multiplied together, (2) a variable multiplied by one constant (single constant multiplication or SCM), and (3) a variable multiplied by more than one constant (multiple constant multiplication or MCM). This paper does not consider the case of two constants being multiplied together because in a HLS flow this multiplication will be replaced by the actual result of multiplication during the compilation phase. Each of these types has been studied in detail individually and various mapping strategies have been proposed for each of these by assuming a priori information about the existence of such types in an application. Modern FPGA devices have dedicated on-chip resources for multiplication for speeding up the multiplication process. Examples of such resource include DSP48 [1] and DSP48E [2] blocks in Xilinx Virtex-4 and Virtex-5/6/7 devices, respectively. Similarly, the 25 × 18 DSP block on Altera FPGAs [3] serves the same purpose. With the presence of a large number of on-chip multiplication resources like these, it is a challenge to map operations to them in a way that is most effective with respect to area and timing performance when the number of multiplication operations exceeds the number of resources and in the presence of different types of multiplication operations. Current high level synthesis flows identify multiplication operations by the presence of multiplication operator in the program. The multiplication operator is "$*$" in C, C++, Verilog, and VHDL. Every instance of "$n*$" in the program is treated equally without any distinction as to whether a particular instance belongs to any of the three categories of multiplication operations mentioned earlier. In contrast, for hand-crafted digital designs, a designer is aware of the different kinds of multiplication operations present in the application and hence can manually choose a particular mapping strategy for each of them. However, this is a time consuming process and dependent on designer expertise.

Allied operations refer to (1) multiplication followed by addition, (2) multiplication followed by subtraction, and (3) multiplication and accumulation. These operations are supported by the DSP blocks provided by all major FPGA vendors and each such operation can be executed on one single DSP block. These operations are typically found in implementations of digital filters like finite impulse response (FIR) filter, polynomial expressions, matrix multiplication, molecular dynamics simulation [4], and so forth. Such operations can also form a chain like in FIR filter design, matrix multiplication, and so forth. Equation (1) shows the calculation of an element of a 3 by 3 product matrix formed by multiplying two 3 by 3 matrices. The calculation is done using a chain of multiplication and addition operations. Mapping those chains to DSP blocks using internal cascaded connections can improve the timing performance of the chain. Therefore, chains of such operations are also automatically identified from the DFG of an application:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} aj + bm + cp & ak + bn + cq & al + bo + cr \\ dj + em + fp & dk + en + fq & dl + eo + fr \\ gj + hm + ip & gk + hn + iq & gl + ho + ir \end{bmatrix}. \tag{1}$$

Multiplication, when implemented using LUTs in FPGA, consumes resources based on the size of operation: bigger multiplications consume more LUTs. Due to the associated logic depth with large multiplications, their timing performance is also poor. Therefore, larger multiplication operations are critical to the timing performance of a design whether they exist independently or as part of a chain.

2.2. Algorithm for Identification of Multiplication and Allied Operations. Since an application can be represented by its control and dataflow graph (CDFG) which is extracted from its C/C++ description, no assumption is made regarding prior information about the type of multiplication operations available in the application. It is also an error prone and time consuming exercise to manually analyze the application for the presence or absence of different types of multiplication operations. Hence, their automatic identification is important. In existing design methodologies, it is left to the designer to manually identify different categories of multiplication operations and select a hardware implementation strategy for them and implement that strategy in the design. While it is possible to do so at the RTL level, it is not always possible at C/C++ level because there can be instances where there is no appropriate C/C++ based description possible for a given hardware implementation strategy. Besides it goes against the spirit of HLS if a hardware structure is explicitly coded (where possible) in C/C++.

This entire manual process is also less amenable to optimization as it is extremely time consuming to generate the results of different implementation strategies and their combinations. Besides, it cannot be assumed that a designer will have all the *domain knowledge* to decide which implementation strategies and their combinations to choose as it is humanly impossible to do so in a large application. Automatic identification of chains of allied operations helps in ensuring that they can be mapped to DSP blocks using internal cascaded signals. This is achieved in the work presented in this paper by instantiating DSP blocks with cascaded connections during RTL code generation phase of

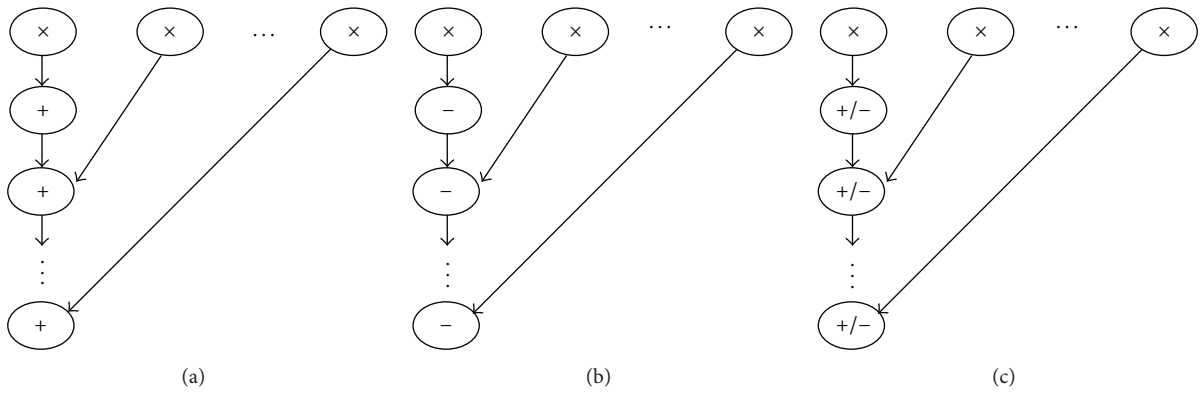


Figure 1: (a) Chain of multiply-add, (b) chain of multiply-subtract, and (c) chain of multiply with a mix of add and subtract.

HLS. If these chains are not identified and RTL code is not generated appropriately, leaving such chains to be handled by inference (as is the case with existing HLS methodologies) by downstream physical synthesis cannot ensure such a mapping. Automatic identification of multiplication and allied operations existing in an application enables applying a joint implementation strategy to three categories and allied operations with respect to mapping them to DSP blocks on FPGAs for better area and timing performance.

The joint implementation strategy is discussed in Section 3.3. The automatic identification algorithm, *autoidentify*, for multiplication categories and allied operations is given in Algorithm 1. The patterns related to chains of allied operations shown in Figure 1 are automatically identified. It should be noted here that the different multiplication operations shown in Figure 1 could be scheduled in different c-steps and not necessarily in the same c-step though they are shown like that in Figure 1 for the sake of clarity. These patterns show that an addition or a subtraction operation should be immediately followed in the next c-step by another addition or subtraction operation which will take another input as the result of some other multiplication operations. The breadth first search (BFS) returns constant value nodes at each level of the DFG (line 1). DFS search (lines 4-5) at each of these nodes then returns their target multiplication nodes (if any) with other inputs to such nodes being variables. These variables and the corresponding constants are then stored in a hash table: H(MUL) with variable identities as hash keys (line 5).

Clearly, there can be more than one constant associated with a variable in the hash table. This way the hash table stores all instances of single constant multiplication as well as multiple constant multiplication. Hash table is a useful data structure to query the different constants that a variable is multiplied with because the variable name can be used as the key. Removing all such multiplication nodes from the DFG would leave us with those multiplication nodes where both the operands are variables. In this way all the three different kinds of multiplication operations are identified automatically.

It should be noted here that the DFG is obtained from the internal representation that a compiler generates for a C-based description as a result of compilation. Hence, the effect of optimizations like replacing multiplication by a multiplier which is a power of two (an example of integer multiplication) by left shifting the multiplicand variable has already been considered. For instance, multiplication by 4 is equivalent to left shifting the multiplicand by 2 bits. Therefore, *autoidentify* algorithm is applied to a DFG which has already undergone such transformations.

For the MCM case, the DFG might have undergone a transformation. Such a transformation could have been brought about by the compiler or it could be another postprocessing step on the DFG. Such transformations could be related to merging of operations or reducing operations to a series of add and shift with adder operator for addition operation being shared, and so forth. For the work presented in this paper, a DFG that has already undergone such transformations is considered. However, it is not necessary that a DFG *should* have undergone such transformations. Thus, the work presented in this paper is independent of such transformations and is to be considered as the next processing step if such transformations have taken place.

The DFG is also traversed in lines 10–22 to locate the patterns identified in Figures 1(a), 1(b), and 1(c). Once these patterns have been identified, multiplication nodes belonging to them are removed from the hash table H(MUL) to nullify duplicate identification of such nodes. The length of the identified chains is also calculated (line 20) and it equals the number of ADD or SUB nodes.

3. Mapping Multiplication and Allied Operations to DSP Blocks

Multiplication has been a widely researched topic. In this section, an overview of work relevant to mapping of the different kinds of multiplication operations identified in Section 2.2 is presented. In [5], the authors have investigated the single constant multiplication (SCM) problem with respect to implementing such operations in FPGA look up tables (LUTs). More work on LUT based SCM mapping has been carried out in [6, 7], and so forth. Their efforts have focused on replacing a multiplication operation by a series of shift and add or subtract operations because multiplication is a costly operation in hardware compared to addition and subtraction

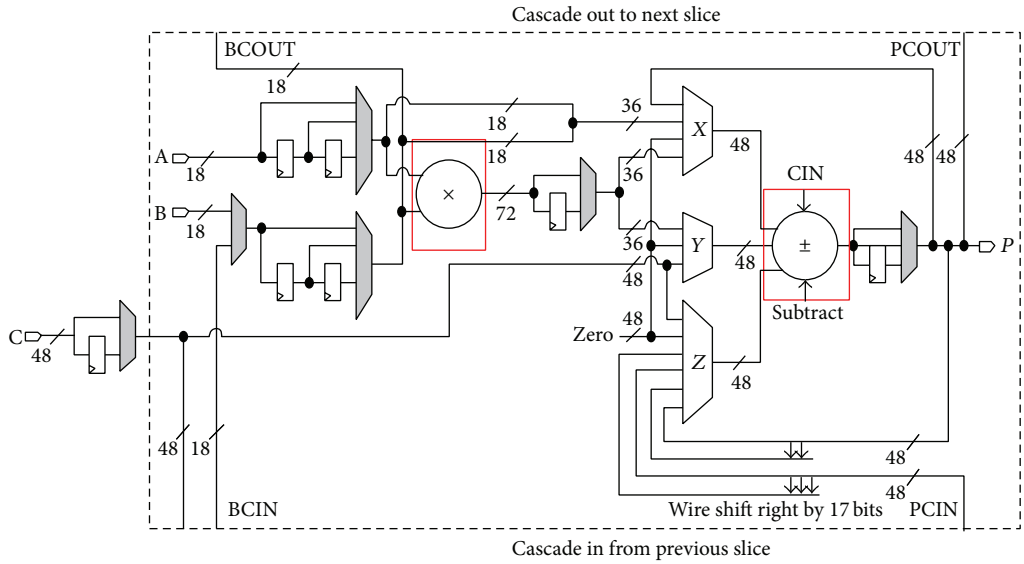
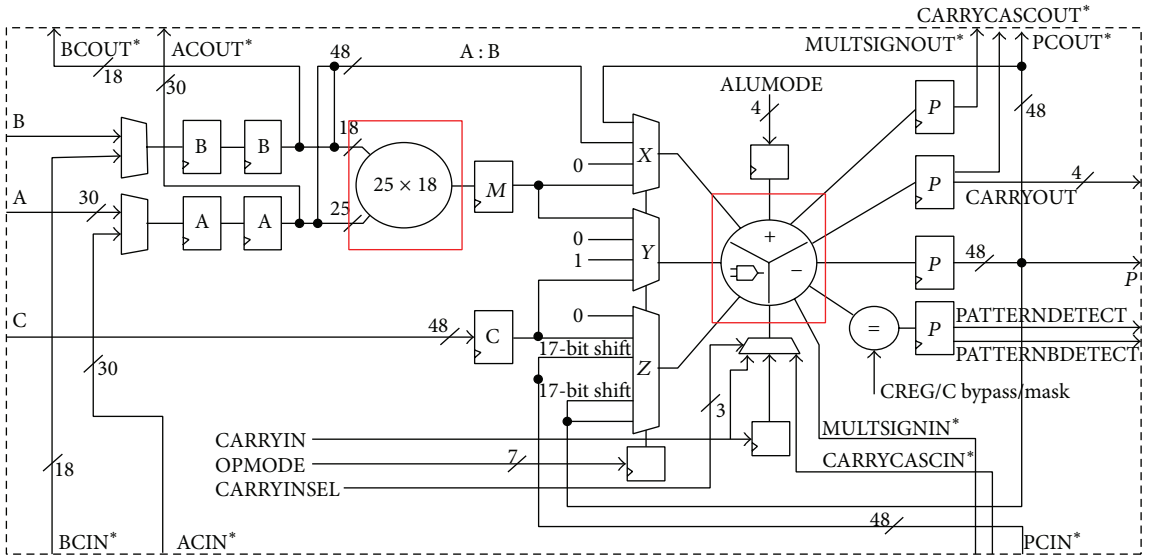


Figure 2: DSP48 architecture in Xilinx Virtex-4 [1].

whereas shift operation requires only wires as resources and no other computational resources. Multiple constant multiplication (MCM), where a variable is multiplied by more than one constant, has been studied in [8, 9]. Again, the effort is directed toward replacing multiplication by a series of shift and add or subtract operations. Besides, these two works have also focused on time multiplexing the operations so that resources can be shared within the multiplier block resulting in overall reduced resource utilization. In [10], the authors have focused on using carry save arithmetic for time multiplexed MCM. Reconfigurable multiplier blocks called ReMB for time multiplexed MCM have been studied in [11, 12] where the authors present the fundamentals in [11] and the algorithm itself in [12]. Mapping to DSP blocks on Xilinx FPGAs has been investigated in [13] and on both Xilinx and Altera DSP blocks in [14] with the aim of reducing the number of DSP blocks needed when the basic multiplication would involve more than one DSP resource. Their work has relied on the Karatsuba-Ofman algorithm [15] for large multiplication. However, they have restricted the size of operands for multiplication to a maximum of 68 × 65 bits. The FPGA vendor Altera also has a design guideline on implementing multipliers in FPGAs [16]. The work that comes closest to the work presented here is [17] where the authors have worked on automatic exploration of arithmetic-circuit architectures. However, their work uses the Boolean functions to represent the arithmetic function. Common subexpression elimination is applied on the Boolean network to select a set of expressions for implementation. However, they do not investigate the role of existing architectural advancements (*the body of knowledge*) in selecting the set of expressions and hence is very different from the work presented here in this sense. Their method does explore a restricted set of architectures, but it is unclear from the paper how it is done. Partitioning of multiplication operations into smaller multiplications and additions has also been studied in [18] as a presynthesis optimization step in high level synthesis with a view to decrease the circuit delay. However, it does not make use of Karatsuba-Ofman algorithm for partitioning the multiplication operations.

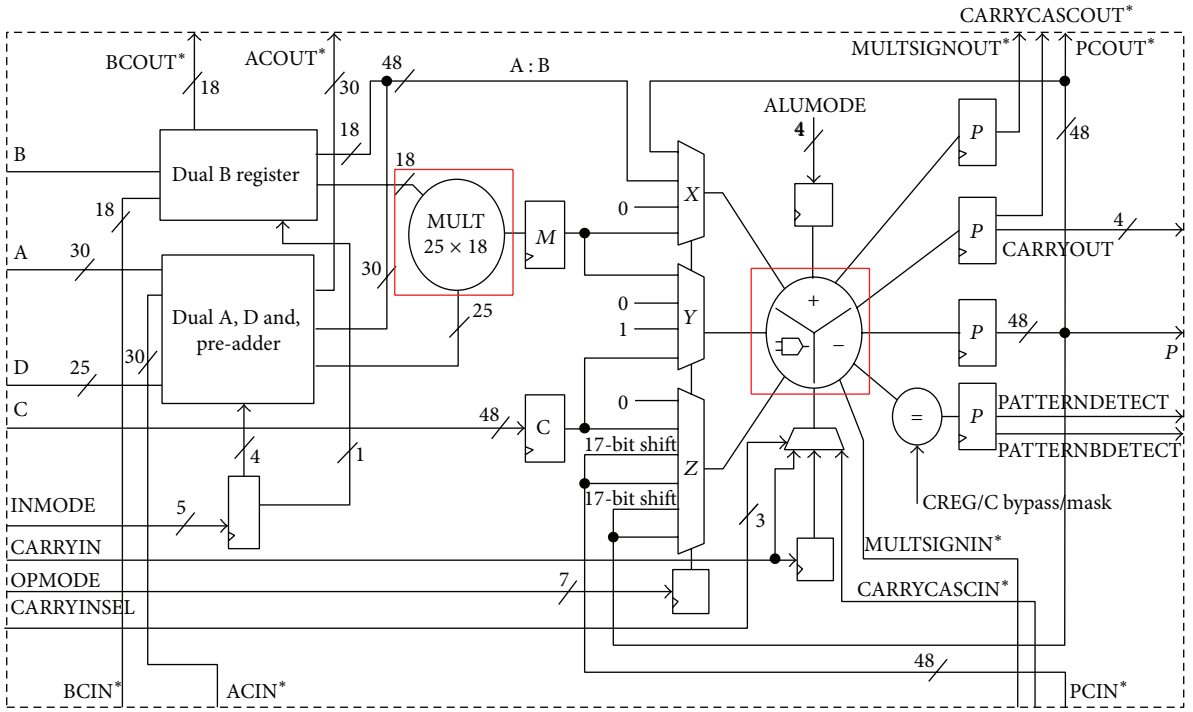
3.1. DSP Block Architecture and Functionality. All major FPGA vendors, that is, Xilinx, Altera, Actel, and so forth, provide on-chip DSP blocks on their FPGA devices. This paper presents work that is applicable to FPGA devices from any vendor and is therefore not restricted to any particular vendor though for the sake of discussion Xilinx FPGAs are considered. These DSP blocks are "hard" blocks and are implemented as ASIC-like blocks on the FPGA. They are distinct from the configurable fabric comprising LUTs, registers, multiplexers, and so forth. All registers and multiplexers within a DSP block are also part of this hard block and are distinct from their counterparts in the soft configurable space of the FPGA. On Xilinx devices, DSP blocks are referred to as "DSP48," "DSP48E," and "DSP48E1" depending on the device family. On Altera devices, they are referred to simply as "DSP blocks" while on Actel devices they are referred to as "mathblocks." Functionally they are all the same with minor variations and enhancements in architecture in newer device families. They were all introduced to provide high performance on-chip resources for multiplication, multiply and accumulate, multiply and add, and multiply and subtract. Figure 2 shows the DSP48 architecture in Xilinx Virtex-4 [1], Figure 3 shows the DSP48E1 architecture in Xilinx Virtex-5 [19], and Figure 4 shows the DSP48E1 architecture in Xilinx Virtex-6 [2]. Their details are easily available in the user guides [1, 2, 19].

These DSP blocks have configurable pipeline registers: at the input A (2 registers), B (2 registers), a register for multiplier output, and another register for output of ALU or add/subtract unit as can be seen in Figures 2, 3, and 4. Ensuring that all these registers are made use of during



*These signals are dedicated routing paths internal to the DSP48E column. They are not accessible via fabric routing resources.

Figure 3: DSP48E1 architecture in Xilinx Virtex-5 [19].



*These signals are dedicated routing paths internal to the DSP48E1 column. They are not accessible via fabric routing resources.

Figure 4: DSP48E1 architecture in Xilinx Virtex-6 [2].

hardware binding leads to maximum operating frequencies of the DSP blocks [1, 2, 19], reaching operating frequencies of more than 450 MHz. Therefore, if fully pipelined DSP blocks are used to implement multiplication and allied operations, the operating frequency of a design increases compared to implementing those operations in FPGA fabric, that is, LUTs. Though this increase in operating frequency comes at the cost of increased latency, it is essential when a design is expected to operate at higher clock frequencies. In the work presented in this paper, DSP blocks are fully pipelined during the hardware binding stage to achieve maximum increase in operating frequency compared to implementation in LUTs of multiplication and allied operations. As can be seen in Figures 2, 3, and 4, each DSP block has a multiplier and

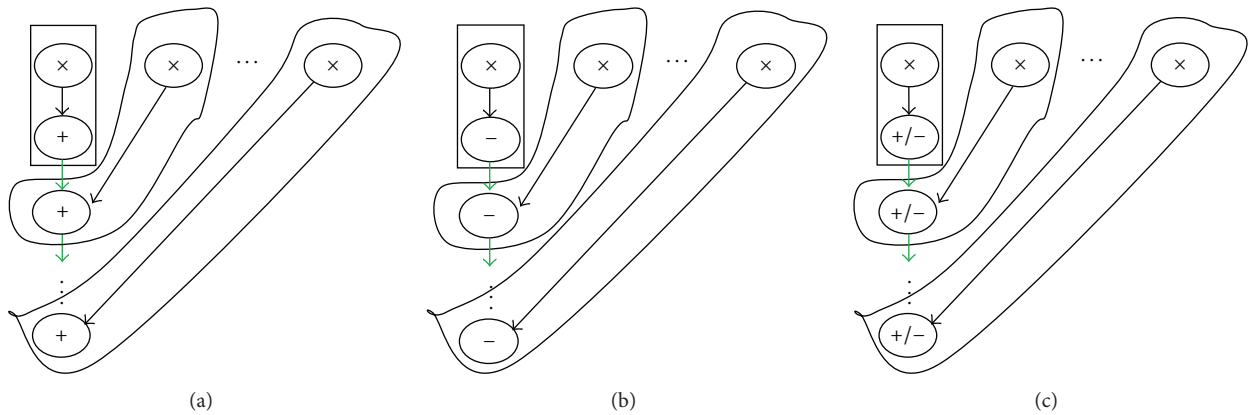


Figure 5: Mapping of operations in a chain of allied operations to DSP blocks.

an adder/subtractor block. The multiplier is 18 bits wide in Virtex-4 while it supports 25-bit by 18-bit multiplication in Virtex-5 and Virtex-6. The adder/subtractor block is enhanced with a logic unit in Virtex-5 and Virtex-6.

Figures 5(a), 5(b), and 5(c) show how the chain of allied operations is mapped to DSP slices.

Each curve represents a DSP block and the green arrows indicate the internal cascaded routing from one DSP block to another in a column of DSP blocks on a FPGA. This leads to greater performance as such signals do not come out to FPGA fabric which would result in performance degradation. The largest estimated LUT utilization of any multiplication operation, $E_{\max}$(LUT), in a chain is taken as a measure of its criticality: chains with larger values of $E_{\max}$(LUT) are more critical from timing perspective.

3.2. LUT Utilization Model for Multiplication Operations. The binary multiplication of two binary numbers "a" and "b", each 3 bits wide can be represented as shown in Table 1, where b_i is the ith bit of binary number "b," a_j is the jth bit of binary number "a," and "×" is Boolean multiplication. As can be seen, there is a circuit size C (number of gates) and a circuit depth D (maximum distance from an input to output) associated with the Boolean circuit implementing this multiplication. Clearly, the greater the number of bits in the operands, the bigger the circuit size and the higher the depth with the latter equal to the number of rows of partial products or the number of rows of partial products plus one when there is a carry-in from some column to another column. When one of the two operands is a constant, all the partial products are reduced to constants and only the logical summation remains. This reduces the circuit size. Therefore, LUT utilization and timing performance depend on C and D.

For the multiplication in Table 1, the partial products "p_{ij}" are given by

$$p_{ij} = b_i \times a_j, \quad \text{where } i, j \in [0, 2]. \tag{2}$$

Table 1: A 3-bit by 3-bit binary multiplication.

			a_2	a_1	a_0
		x	b_2	b_1	b_0
			p_{02}	p_{01}	p_{00}
		p_{12}	p_{11}	p_{10}	
	p_{22}	p_{21}	p_{20}		
PB_5	PB_4	PB_3	PB_2	PB_1	PB_0

The product bits "PB_i" are given by the sum of the partial products in the respective columns and can be calculated using

$$PB_i = C_{i-1} + \sum p_i, \quad \text{where } i \in [0, 5], \quad C_{i-1} = 0 \quad \text{for } i = 0, \tag{3}$$

where p_i is the partial product belonging to ith column, C_{i-1} being the carry-out from the summation of partial products in $(i-1)$th column with its value being 0 for $i = 0$. Here summation refers to Boolean summation performed by an adder. Note that a m-bit by n-bit multiplication will result in a total of $(m+n)$ product bits and $(m+n-1)$ columns of partial products with the carry-out from $(m+n-1)$th column forming the $(m+n)$th product bit.

The binary multiplication method in Table 1 is adopted to model the LUT resource utilization of different kinds of multiplication operations. Boolean multiplication is performed by an AND gate while Boolean summation is performed by an adder circuit. Therefore, the number of partial products affects the number of required AND gates and the number of summations affects the number of required adders. The subsections ahead model the different kinds of multiplication operations to give an estimate of the number of required AND operations and adders. Ripple carry adder is assumed for modeling the number of full adders.

3.2.1. Variable Multiplied with Another Variable. An example of this kind of multiplication is variable $V1$[16:0] multiplied with another variable $V2$[17:0] resulting in product

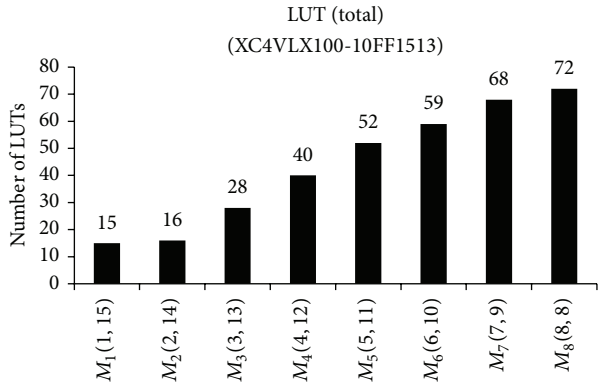


Figure 6: Actual LUT utilization corresponding to 8 different multiplications with the same product bit-width (16).

$P1[34:0]$. Since the multiplication operator when mapped to the basic mapping element in a FPGA, that is, LUTs, involves decomposing the multiplication operation into logical AND operations and use of adders, the utilization increases when the number of AND operations and the number of adders is high. This means that one can order two or more such instances of multiplication based on the estimated LUT utilization. As can be seen in Figure 6, 8 different instances of multiplication where the product bit width remains constant (16 bits wide) result in different LUT utilization values and different clock periods. This is due to the effect of number of logical AND operations and number of full adders (FAs) for summation.

For a *m-bit* variable M multiplied with a *n-bit* variable N, the number of required AND operations, that is, Count(AND), is given by (4) and the number of required Fas, that is, Count(FA), is given by (5). Some of these FAs are actually half adders (HA) because their carry-ins are tied to logical "0." Hence, no distinction is made between complete FAs and those used as HAs for the purpose of adder resource estimation:

$$\text{Count (AND)} = m \times n \tag{4}$$

$$\begin{aligned}\text{Count (FA)} &= \{[(m+n-1) - ((\min(m,n)-1)\times 2)] \times \min(m,n)\} \\ &+ \sum_{q=1}^{q=\min(m,n)-1} q + \sum_{t=1,t\neq 2}^{t=\min(m,n)-1} t.\end{aligned} \tag{5}$$

In Figure 6, $M_1(1, 15)$ means multiplier design 1 with operand bit widths 1 and 15 resulting in a product bit width of 16 (= 1+15). A 1-bit logical AND operation requires "a" LUTs while a FA requires "b" LUTs. For Xilinx FPGAs, $a = 1$ and $b = 1$. Hence, the total number of LUTs estimated, Count(LUT) to be required is given by

$$\text{Count (LUT)} = a \times \text{Count (AND)} + b \times \text{Count (FA)}. \tag{6}$$

This estimated value of Count(LUT) will always be higher than the actual number of LUTs used, that is, Actual(LUT), because this estimate does not take into account packing of logic functions at the Boolean logic level where both logical AND operation and FA operations will be considered. The difference between Count(LUT) and Actual(LUT) is of no significance in the present work because Count(LUT) will be used to order the multiplication operations and it is evident from Table 2 that the inequality in (7) holds:

$$\begin{aligned}&\text{If Count}\left(\text{LUT}, M_1\right) \geq \text{Count}\left(\text{LUT}, M_2\right), \\ &\text{then Actual LUT}\left(M_1\right) \geq \text{Actual LUT}\left(M_2\right) \quad \forall M_i,\end{aligned} \tag{7}$$

where M_i are different multiplication instances and M_1 and M_2 are any two cases out of M_i.

Table 2: Variation of actual (LUT), count (LUT), and DSP implementation.

$V1$	$V2$	Actual (LUT)	Count (LUT) (2)	CLK (ns) (LUT Imp.)	DSP Imp.
16	8	128	254	6.307	1
24	8	184	392	6.883	2
32	8	240	510	7.734	2
32	32	1113	2750	12.511	4

Table 2 shows the actual LUT utilization in ascending order with the maximum LUT utilization being for $V1[31:0]$ multiplied with $V2[31:0]$ (device: XC4VLX100-10FF1513). Column *DSP Imp.* shows the number of DSP blocks required if these multiplications were to be mapped to them and not to LUTs. Thus, when compared with the number of DSP blocks used, we can say that the "*equivalent LUT cost*," that is, Count(LUT) of mapping, a multiplication operation of this type is the number of LUTs used when the multiplication operator "$*$" is used and mapped to LUTs. Since the DSP blocks are very fast with maximum clock frequency being 450 MHz for DSP 48 and 550 MHz for DSP48E, mapping these multiplication operations to the DSP blocks gives better timing performance compared to mapping them to LUTs which is evident from the high clock periods in Table 2 for mapping to LUT.

3.2.2. Multiple Constant Multiplication (MCM). There can be multiple instances of MCM in an application. For instance, a variable $V1[7:0]$ can be multiplied with $\{33, 117, 221, 45, 98, 171\}$ while another variable $V2[7:0]$ can be multiplied with $\{3, 5, 6, 7, 9, 10, 11\}$. If there is limited availability of DSP blocks, then those sets of MCM operations should be mapped to DSP blocks which would benefit most in terms of performance. Figure 7 shows the LUT utilization result of multiplying an 8-bit variable $A[7:0]$ with seven different 8-bit constants. Figure 8 shows the clock period (ns) for the same set of designs. These figures clearly demonstrate that the actual value of a constant affects the LUT utilization as well as the minimum clock period (maximum clock frequency). This is in contrast to DSP blocks where the size or the actual value of operands does not affect the maximum clock frequency as long as the size of operands can be accommodated within the maximum input

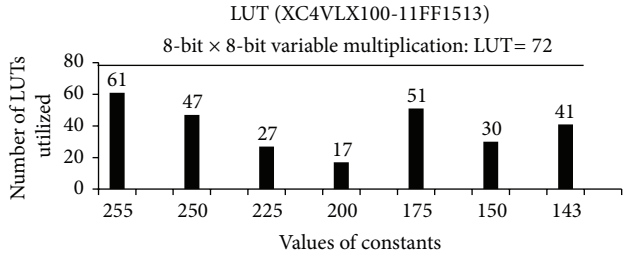


Figure 7: Actual LUT utilization for $A[7:0] \times 8$-bit constant.

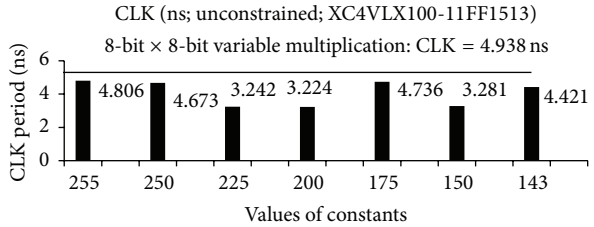


Figure 8: CLK period (ns) for $A[7:0] \times 8$-bit constant.

size available for one DSP block. For all multiplication cases in Figure 7, one (1) DSP block was used and the clock period achieved was 2.39 ns.

It can be clearly seen from Figure 7 that a variable (A, 8 bits wide) multiplied by another variable (B, 8 bits wide) consumes more LUT resources compared to A being multiplied by a constant with the same bit width as variable B. Similarly, Figure 8 shows that a variable (A, 8 bits wide) multiplied by another variable (B, 8 bits wide) suffers from higher clock period compared to variable A multiplied by a constant with the same bit width as B in its binary representation. It can be seen easily seen from Figures 7 and 8 that the value of the constant has no direct relationship with LUT utilization; that is, LUT utilization does not always increase with an increase in the value of the constant or vice versa. The "*Equivalent LUT cost*" for each multiplication operation in a set of MCM is calculated in order to make comparison between sets of MCM as well as to compare with variable multiplied by variable. The formulas required to compute this cost for a variable multiplied by a constant are presented in Section 3.2.4.

3.2.3. Single Constant Multiplication (SCM). SCM can be considered as a special case of MCM where a given variable is multiplied by just one constant. Therefore all the analytical arguments presented by way of Figures 7 and 8 are applicable to SCM also.

3.2.4. Analysis of Multiplication with a Constant

Symmetric Multiplication. For symmetric multiplication, the number of partial product terms $\mathrm{PP}(C_i)$ in each column C_i can be easily derived from the set of equations in (8). For an

Table 3: Contributing rows for different columns for 4-bit variable multiplied by 4-bit constant.

Column	Contributing rows (each row contributes 1 partial product element to the column)
C_0	R_0
C_1	R_0, R_1
C_2	R_0, R_1, R_2
C_3	R_0, R_1, R_2, R_3
C_4	R_3, R_2, R_1
C_5	R_3, R_2
C_6	R_3

Table 4: Contributing rows for different columns for $m * m$ multiplication.

Column	Condition	Contributing rows (each row contributes 1 partial product element to the column)
C_i	$i \in [0, m-1]$	R_0 to R_i
C_i	$i \in [m, 2m-2]$	R_{m-1} to $R_{m-\mathrm{PP}(C_i)}$

m-bit by *m-bit* multiplication, which will have $2m$ product bits and $2m - 1$ columns:

$$\begin{aligned} \mathrm{PP}(C_i) &= i + 1 \quad \text{for } i \in [0, m-1], \\ \mathrm{PP}(C_i) &= C_{i-1} - 1 \quad \text{for } i \in [m, 2m-2] \end{aligned} \tag{8}$$

each column C_i up to $i = m - 1$ corresponds to the bit location b_i in the binary representation of the constant. Also, the number of rows of partial products equals the number of bits in the binary representation of the constant. Therefore if any bit $b_i = 0$, then all partial products in row R_i are zeros. The number of terms in each row R_i is equal m. Table 3 shows which rows contribute partial products to which columns for a 4-bit variable multiplied by a 4-bit constant. The contributing rows for different columns for $m * m$ multiplication are given in Table 4. Thus, the number of FAs can be calculated based on which rows are all zeroes; that is, their partial products are all zeroes given the fact that row R_i corresponds to bit b_i in the binary representation of the constant. All AND operations are just wires when a variable is multiplied with a constant and hence they are ignored. For instance, for $A[3:0]$ multiplied by $4'b1010$, Table 5 shows calculation of the number of FAs.

A C_{out} in a column results in a FA in the next column. Hence, for the example in Table 5, the total number of FAs is 5.

Asymmetric Multiplication. For asymmetric *m-bit* by *n-bit* multiplication, there will be $(m + n)$ product bits and $(m + n - 1)$ columns with $(m+n)$th column being just the $(m+n)$th product bit with a value of either 0 or 1. The number of

Table 5: Example number of FAs for $m = 4$ (constant = $4'b1010$).

Column	Contributing rows	Number of nonzero partial products	Number of FAs as a result of column addition
C_0	R_0	0	0
C_1	R_0, R_1	0, 1 = 1	0
C_2	R_0, R_1, R_2	0, 1, 0 = 1	0
C_3	R_0, R_1, R_2, R_3	0, 1, 0, 1 = 2	$1 + C_{out} = 2$
C_4	R_3, R_2, R_1	1, 0, 1 = 2	$1 + C_{out} = 2$
C_5	R_3, R_2	1, 0 = 1	C_{out}
C_6	R_3	1	0

Table 6: Contributing rows for $m > n$.

Column	Condition	Contributing rows (each row contributes 1 partial product element to the column)
C_i	$i \in [0, C_{begin})$	R_0 to R_i
C_i	$i \in [C_{begin}, C_{end}]$	R_0 to R_{m-1}
C_i	$i \in [C_{end+1}, m+n-2]$	R_{n-1} to $R_{(n-1)-PP(C_i)+1}$

columns, NC_{equal}, with equal number of partial products is given by

$$NC_{equal} = \max(m, n) - \min(m, n) + 1. \quad (9)$$

The beginning column number for such a column is given by

$$C_{begin} = \min(m, n) - 1. \quad (10)$$

The end column number for such a column is given by

$$C_{end} = C_{begin} + NC_{equal} - 1. \quad (11)$$

The number of partial product terms $PP(C_i)$ in each column C_i can be easily derived from the set of equations in

$$PP(C_i) = i + 1 \quad \text{for } i \in [0, C_{begin}),$$
$$PP(C_i) = C_{i-1} - 1 \quad \text{for } i \in (C_{end} + 1, m + n - 2]. \quad (12)$$

The rows which contribute to a particular column can be obtained from the expressions in Table 6 for $m > n$, where m is the number of bits in variable and n is the number of bits in the constant.

Table 7 shows which rows contribute partial products to which columns for a 5-bit variable multiplied by a 3-bit constant. The rows which contribute to a particular column can be obtained from the expressions in Table 8 for $m < n$, where m is the number of bits in variable and n is the number of bits in the constant. Table 9 shows which rows contribute partial products to which columns for 3-bit variable multiplied by a 5-bit constant.

Each column C_i, where $i \in [1, m + n - 3]$, contributes a maximum of $PP(C_i)$ number of FAs which includes the FA required to add the carry-out from C_i to C_{i+1}. No FA is contributed by C_0 and C_{m+n-2} under this analysis. Let $R_k = 0$ mean that all elements (partial products) in row k are zeros and it will correspond to bit b_k in the binary representation of the constant. Therefore, the number of FAs contributed by a column will decrease by as many rows that are zeros. The contribution of the adder for carry-out from column C_i *will* also vanish if all rows or all rows except the one that contributes to C_i are zeros.

Table 7: Contributing rows for $m = 5$ and $n = 3$.

Column	Contributing rows (each row contributes 1 partial product element to the column)
C_0	R_0
C_1	R_0, R_1
C_2	R_0, R_1, R_2
C_3	R_0, R_1, R_2
C_4	R_0, R_1, R_2
C_5	R_2, R_1
C_6	R_2

Table 8: Contributing rows for $m < n$.

Column	Condition	Contributing rows (each row contributes 1 partial product element to the column)
C_i	$i \in [0, C_{begin})$	R_0 to R_i
C_i	$i \in [C_{begin}, C_{end}]$	$R_{i-(m-1)}$ to $R_{i-(m-1)+NC_{equal}-1}$
C_i	$i \in [C_{end+1}, m+n-2]$	R_{n-1} to $R_{(n-1)-PP(C_i)+1}$

Table 9: Contributing rows for $m = 3$ and $n = 5$.

Column	Contributing rows (each row contributes 1 partial product element to the column)
C_0	R_0
C_1	R_0, R_1
C_2	R_0, R_1, R_2
C_3	R_1, R_2, R_3
C_4	R_2, R_3, R_4
C_5	R_4, R_3
C_6	R_4

3.2.5. Analysis of Chain of Allied Operations. For each chain of allied operations identified in the DFG, the multiplication operation with the highest estimated LUT consumption is taken into consideration by the selection algorithm (Section 3.3). If a multiplication operation belonging to a chain of allied operations is selected for mapping to DSP block, then the entire chain is mapped to a series of DSP blocks as shown in Figure 5 as this makes use of fast internal routing within the DSP column and splitting a chain between DSP blocks and FPGA general fabric will result in degraded performance.

Table 10: Variation in LUT consumption for different 9-bit constants with the same number of 1′s in binary representation.

Constant	Binary	Number of 1′s	LUT	CLK (ns)	Number of FAs
260	100000100	2	9	2.077	9
259	100000011	3	16	2.509	22
261	100000101	3	15	2.479	21
383	101111111	8	63,47	5.490	62
447	110111111	8	63,48	5.638	62
479	111011111	8	62,50	5.781	62
495	111101111	8	64,51	5.855	64
503	111110111	8	61,50	5.556	63
510	111111110	8	56,45	5.687	62
511	111111111	9	60,49	5.501	70

3.2.6. Why the Proposed Estimation Models Are Better? The expected LUT utilization for multiplication of two variables of bit widths m and n can be also approximated as $m \times n$ when calculated to full precision, that is, $(m + n)$ bits in product. However, this leads to inconsistent estimation for symmetric and asymmetric multiplication. For instance, from Table 2, 16-bit multiplied by 8-bit uses 128 LUTs (actual) which equals $m \times n = 128$. However, 24-bit multiplied by 8-bit uses 184 LUTs (actual) while $m \times n = 192$. In this case, there is overestimation. Similarly, 32-bit multiplied by 32-bit uses 1113 LUTs (actual) while $m \times n = 1124$ resulting in overestimation. Thus, it can be seen that there is no consistency in result when estimated using the approximate formula $m \times n$. Also, the biggest drawback with this estimation method is that it disregards the effect of one of the multiplicands being a constant. It also does not work when the result of a multiplication is not calculated to full precision. Therefore, using this formula, an *m-bit* variable multiplied by a *n-bit* variable would be estimated to use the same number of LUTs as an *m-bit* variable multiplied by an *n-bit* constant. On the other hand, the proposed model also works when the product is not calculated to full precision. In this case, $(m + n)$ take the value of the number of bits in the product instead of their full precision value and all relevant columns, partial products, and full adders can be easily identified from the formulas in the previously mentioned tables.

Quadratic curve fitting based estimation proposed in [20, 21] also suffers from the drawback that it cannot take into account the effect of one of the multiplicands being a constant. Besides, curve fitting is based on using physical synthesis data for a set of multiplication operations, thus exposing the estimation method to variance in physical synthesis approaches. The resource utilization model proposed in this paper is independent of physical synthesis approach and relies on the basic multiplication method to provide consistent approximation results. It should be noted and emphasized here that the approximation method proposed in this paper is meant for the purpose of ordering different multiplication operations in an application. It is not its objective to give an estimate which is close to the actual synthesis result as such an objective is not necessary when ordering multiplication operations. The proposed resource utilization model also enables estimating LUT utilization for any LUT based device either existing or the ones to appear in the future as it is independent of both device architecture and synthesis algorithms and is based entirely on mathematical reasoning.

Table 10 shows the variation in LUT consumption for an 8-bit variable multiplied by different 9-bit constants. It can be seen that the number of LUTs actually consumed is proportional to the number of 1′s in the binary representation of the constant.

Therefore, LUT consumption could also have been estimated by assuming a proportionality relationship between LUT consumption and number of 1′s in the binary representation of a number. However, when the number of 1′s is the same, there is still variation in LUT consumption. This information cannot be captured by the proportionality logic but is captured by the number of FAs (the proposed model in this paper) though with some deviations as evident in row for "511." Thus, the count of FA and AND operations (no AND gates for multiplication with a constant) is a more reliable measure to make comparisons between the three classes of multiplication operations.

3.3. Selection Algorithm for Mapping of Multiplication and Allied Operations. This section presents the joint (J) comparison (C) and selection (S) algorithm (Algorithm 2) for joint comparison between the three classes of multiplication operations leading to the selection of operations to be prioritized for mapping to DSP blocks on FPGA.

In the proposed algorithm, the numbers of FAs, AND gates, and LUTs are calculated first (lines 1–4) for SCM, MCM, and other multiplication operations. Thereafter for the identified MUL-(ADD/SUBTRACT) chains, the maximum LUT cost is estimated (lines 5-5.1). Once these are done, multiplication operations common to H(MULT) hash table and MULT-(ADD/SUBTRACT) chains are removed from H(MULT). Thus, H(MULT) only has multiplication operations which are not part of MULT-(ADD/SUBTRACT) chains (line 6). Thereafter multiplication operations are arranged in descending order of estimated LUT consumption

Table 11: Result comparison between Vivado-HLS and proposed approach for 7-tap FIR filter.

Design	Vivado-HLS			Proposed approach	
	CLK period required (ns)	CLK period achieved (ns)	DSP48	CLK period achieved (ns)	DSP48
7-tap FIR filter	3	5.819	9	2.413	7
	4	5.837	9		
	5	6.083	9		
	6	5.902	9		

to form list D (line 7). Operations beginning from the top in list D are mapped to DSP blocks.

Since list D may also contain a multiplication operation belonging to a MULT-(ADD/SUBTRACT) chain, all operations in such a chain are mapped to DSP blocks as explained in Section 3.1. Once all DSP blocks are exhausted this way, remaining multiplication operations and MULT-(ADD/SUBTRACT) chains are mapped to LUTs. During the automatic RTL datapath generation phase, DSP blocks are instantiated automatically in the RTL description and internal cascaded connections between the DSP blocks are utilized where necessary (as in Figure 5) to ensure that such routing does not use low speed FPGA fabric. The internal cascaded connections being made from high speed routing lead to increased operating frequency.

In order to maintain the generality of the proposed design methodology, it has been assumed that bit-width optimization techniques [22–24] and so forth have already been applied to the DFG before the algorithms presented in this paper process the DFG.

4. Experiments and Results

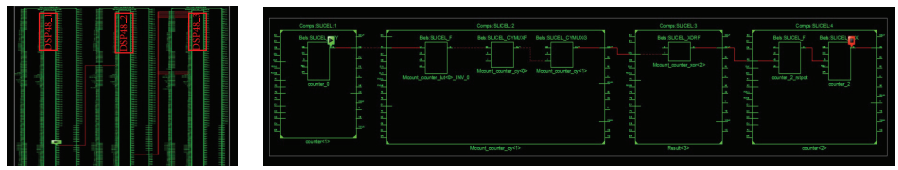
In this section, case studies are presented to demonstrate the better quality of results obtained using the methodology presented in this paper. The proposed methodology was implemented within a C-to-RTL framework and experiments were carried out using Xilinx XC4VLX100 device, Vivado-HLS 2012.1 [25], and Xilinx ISE 14.2 [26]. The experiments were carried out for two different scenarios: (1) when the number of multiplication operations (M) in an application exceeds the number of available DSP resources (D) and (2) when the number of multiplication operations (M) in an application is less than the number of available DSP resources (D). While it would appear that the methodology proposed in this paper is more applicable for the case $M > D$, the experiments demonstrate that it is also applicable for the case $M < D$ when maximizing performance is an objective. For the selected Xilinx device, $D = 32$. Vivado-HLS was chosen for comparison because it is a tool representative of all the steps in a HLS flow and hence a comparative analysis with the proposed methodology is a reflection on the limitations of existing HLS approaches.

4.1. FIR Filter. Different FIR filters were implemented, with the number of taps being more than 32. Direct form filter structure was used. Similar results were obtained for them. For instance, a 53-tap filter was implemented using Vivado-HLS and the methodology proposed in this paper. In this case $M = 53$. Vivado-HLS could not meet 7 ns, 8 ns, and 10 ns clock period requirements and hence could not implement the design on the chosen device. It could implement the design only at 20 ns clock period requirement. On the other hand, using the proposed methodology, the fastest possible implementation could be clocked at a clock period of 6.97 ns resulting in more than 3× faster design and it made use of all the 32 DSP blocks before mapping the remaining multiplication operations to LUTs. Both Vivado and the proposed approach used 32 DSPs, but the proposed approach was superior because it chose which operations to map to DSP based on the filter coefficients. Out of the 53 filter coefficients (which lead to 53 constant multiplications), based on the LUT cost model (Section 3.2), the proposed JCS algorithm chose those constant multiplications which had higher LUT cost. Clearly, implementing them in LUT would degrade the performance and hence the JCS algorithm selected them for mapping to DSP blocks.

4.2. 40 by 40 Matrix Multiplication. In this case, a large matrix A with 40 elements in each row was multiplied with another large matrix B with 40 elements in each column. The total number of multiplications and additions required to get one element in the product matrix is 40 and 39, respectively. As a result $M = 40$. In this case, data for the elements in one row and data for the elements in one column for matrices A and B, respectively, were input to the function to be implemented. Vivado-HLS could not meet 7 ns, 8 ns, and 10 ns clock period requirements and hence could not implement the design on the chosen device. It could implement the design only at 15 ns clock period requirement. On the other hand, using the proposed methodology, the fastest possible implementation could be clocked at a clock period of 6.512 ns resulting in more than 2× faster design. It made use of all the 32 DSP blocks before mapping the remaining multiplication operations to LUTs.

4.3. 7-Tap FIR Filter. In this case $M (= 7) < D (= 32)$. The results are shown in Table 11.

When implemented using Vivado-HLS, it was found that Vivado-HLS could not meet 3 ns, 4 ns, and 5 ns clock period requirements. For 3 ns, it reported utilization of 9 DSP blocks with an achievable clock of 5.819 ns. For 4 ns, it reported utilization of 9 DSP blocks with an achievable clock of 5.837 ns while for 5 ns, the figures were 9 DSP blocks



(a) (b)

Figure 9: (a) Critical path in Vivado implementation. (b) Critical path in proposed implementation.

and 6.083 ns. It succeeded for a clock period requirement of 6 ns reporting 9 DSP blocks and achievable clock of 5.902 ns. Using the methodology proposed in this paper, only 7 DSP blocks were utilized with a much smaller clock period of 2.413 ns.

It should be noted that due to clever assignment of multiplication operations based on the proposed approach, the critical paths (in red) are different between Vivado implementation and the proposed implementation. In case of Vivado implementation, the critical path runs across three consecutive DSP blocks as shown in Figure 9(a) whereas for the proposed implementation, the critical path lies in the state machine as seen in Figure 9(b). These figures were taken using the technology viewer in Xilinx ISE.

4.4. Quadratic Polynomial. The quadratic polynomial: $49199x^2 + 27133x + 41237$ was implemented using Vivado-HLS as well as the proposed methodology. The Vivado-HLS generated result failed the design rule check (DRC) during implementation and hence though it did generate RTL description of the design, it could not be implemented. On the other hand, using the proposed methodology, 4 DSP blocks were used and the smallest clock period achieved was 2.567 ns.

4.5. MCM-1. This consisted of multiplying a 16-bit variable A with 8 different randomly selected constants (33, 117, 221, 45, 98, and 171). The proposed methodology used one DSP48 achieving a maximum clock frequency of 2.2 ns while Vivado-HLS could implement it only at a clock period constraint of 4 ns using 2 DSP blocks. Thus the proposed method resulted in a design which was nearly 2× faster in performance.

5. Conclusion

This paper has presented a methodology to automatically identify and map multiplication and allied operations to DSP blocks on FPGAs, thus bringing in both application and architecture awareness in HLS. It has been shown that the proposed LUT consumption model serves the purpose of ordering these operations with respect to their criticality (in terms of resource utilization) and thus enables to map them in order to maximize performance. The automated methodology is relevant for all types of multiplication and allied operations. The result for 7-tap FIR filter also demonstrates that applying compiler optimizations like replacing a multiplication with a series of shift and add operations does not always result in better timing performance. It can be seen that increase in operating frequencies of more than 3 times was obtained (53-tap FIR filter). Figures 9(a) and 9(b) clearly show that the critical path can also change when the proposed methodology is applied. While results have been reported for Xilinx FPGAs, the methodology is equally applicable to FPGAs from any vendor that support on-chip DSP blocks.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This work was carried out when the first author was at Nanyang Technological University, Singapore.

References

[1] "Xtreme DSP for Virtex-4 FPGAs User Guide," UG073(v2.7), 2008.

[2] Virtex-6 FPGA DSP48E1 Slice User Guide, UG369(v1.3), 2011.

[3] 2013, http://www.altera.com/devices/fpga/stratix-fpgas/about/dsp/stx-dsp-block.html.

[4] 2014, http://en.wikipedia.org/wiki/Molecular_dynamics.

[5] M. J. Wirthlin, "Constant coefficient multiplication using look-up tables," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 36, no. 1, pp. 7–15, 2004.

[6] N. Boullis and A. Tisserand, "Some optimizations of hardware multiplication by constant matrices," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1271–1282, 2005.

[7] P. K. Meher, "LUT optimization for memory-based computation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 57, no. 4, pp. 285–289, 2010.

[8] Y. Voronenko and M. Puschel, "Multiplierless multiple constant multiplication," *ACM Transactions on Algorithms*, vol. 3, no. 2, article 11, 2007.

[9] P. Tummeltshammer, J. C. Hoe, and M. Püschel, "Time-multiplexed multiple-constant multiplication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 9, pp. 1551–1563, 2007.

[10] R. Gutierrez, J. Valls, and A. Perez-Pascual, "FPGA-implementation of time-multiplexed multiple constant multiplication based on carry-save arithmetic," in *Proceedings of the International Conference on Field Programmable Logic and Applications*

(FPL '09), pp. 609–612, Prague, Czech Republic, September 2009.

[11] S. S. Demirsoy, I. Kale, and A. G. Dempster, "Synthesis of reconfigurable multiplier blocks: part I—fundamentals," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'05)*, vol. 1, pp. 536–539, May 2005.

[12] S. S. Demirsoy, I. Kale, and A. G. Dempster, "Synthesis of reconfigurable multiplier blocks: part I—algorithm," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '05)*, pp. 540–543, May 2005.

[13] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 250–255, September 2009.

[14] F. de Dinechin and B. Pasca, "Large multipliers with less DSP blocks," LIP Research Report RR2009-03, 2009.

[15] A. Karatsuba and O. Yu, "Multiplication of multidigit numbers on automata," *Soviet Physics Doklady*, vol. 7, no. 7, pp. 595–596, 1963.

[16] Implementing Multipliers in FPGA Devices, Altera Application Note, AN 306(ver 3.0), July 2004.

[17] A. K. Verma and P. Ienne, "Towards the automatic exploration of arithmetic-circuit architectures," in *Proceedings of the Design Automation Conference (DAC '06)*, pp. 445–450, 2006.

[18] R. Ruiz-Sautua, M. C. Molina, J. M. Mendías, and R. Hermida, "Pre-synthesis optimization of multiplications to improve circuit performance," in *Proceedings of the Design, Automation and Test in Europe (DATE '06)*, vol. 1, pp. 1–6, Munich, March 2006.

[19] *Virtex-5 FPGA XtremeDSP Design Considerations User Guide*, UG193 (v3.5), 2012.

[20] D. Kulkarni, W. A. Najjar, R. Rinker, and F. J. Kurdahi, "Compile-time area estimation for LUT-based FPGAs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 1, pp. 104–122, 2006.

[21] T. Jiang, X. Tang, and P. Banerjee, "Macro-models for high level area and power estimation on FPGAs," in *Proceedings of the 14th ACM Great lakes Symposium on VLSI (GLSVLSI '04)*, pp. 162–165, April 2004.

[22] G. A. Constantinides and G. J. Woeginger, "The complexity of multiple wordlength assignment," *Applied Mathematics Letters*, vol. 15, no. 2, pp. 137–140, 2002.

[23] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie, "A metric for automatic word-length determination of hardware datapaths," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2228–2231, 2006.

[24] L. Zhang, Y. Zhang, and W. Zhou, "Fast trade-off evaluation for digital signal processing systems during wordlength optimization," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '09)*, pp. 731–738, San Jose, Calif, USA, November 2009.

[25] Vivado-HLS User Guide, UG 910 (v 2013.2), June 2013.

[26] *XST User Guide for Virtex-4, Virtex-5, Spartan-3 and Newer CPLD Devices*, UG627(v 14.5), 2013.