Ph.D. DISSERTATION

Design Methodology of Adaptable Hybrid Adders

적응 가능한 이종 가산기 설계 방법론

BY

YONGHWAN KIM

FEBRUARY 2012

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Design Methodology of Adaptable Hybrid Adders

적응 가능한 이종 가산기 설계 방법론

BY

YONGHWAN KIM

FEBRUARY 2012

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Design Methodology of Adaptable Hybrid Adders

적응 가능한 이종 가산기 설계 방법론

지도교수 김 태 환

이 논문을 공학박사 학위논문으로 제출함

2011년 11월

서울대학교 대학원

전기컴퓨터 공학부

김 용 환

김용환의 공학박사 학위 논문을 인준함

2011년 12월

위 원 장: _____

부위원장: _____

위 원: _____

위 원: _____

위 원: _____

Abstract

As the CMOS processing technology scales down, saytisfiying timing constraints is becoming more important in the integrated circuit design, and most critical timing paths in a circuit involve one or more arithmetic components such as adder, subtractor, and multiplier. Subtractor and multiplier can be implemented with adder, there have been many researches regarding the enhancement of the speed of the adder.

This dissertation provides the method of redesigning the addition logic on a critical timing path to meet the timing constraint while minimally allocating the area of adders using hybrid structure bf AHA(adaptable hybrid adder). The previous hybrid adder structures assumed uniform or specific patterns of input arrival times to the adder or used very simplified method to estimate the delay. But, the proposed method extracts the required time as well as the input arrival time from the real circuit implementation. With these timing constraints, the proposed method uses a systematic approach of hybrid adder design exploration, based on dynamic programming with well-controlled pruning techniques. The proposed method can cope with various timing constraints which were extracted from real circuits by satisfying given timing constraints with minimal area. Various experimental data are provided to show the applicability of the proposed method.

keywords: Hybrid adder, RTL resynthesis, arithmetic optimization, timing optimization

student number: 2007-30216

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern VLSI designs including that of digital signal processing applications perform very intensive arithmetic operations repeatedly under tight timing requirements. Consequently, synthesizing fast arithmetic circuits for the operations under area and/or power constraints has been an important research topic.

Among the arithmetic operations, addition is the most common operation component and thus, a considerable work has been devoted to designing fast adders or area efficient adders under a tight timing constraint. The adders can be classified into two groups: *pure* adders and *hybrid* adders.

1.1 *Pure* adders

The pure adders are the ones generated by applying a specific addition scheme to groups of bit addends uniformly, such that different pure adders have different characteristics of performance, power, and area. Ripple-carry adder (RCA) is a typical example of pure adder with small area but long carry generation time. Fig. 1.1 shows 32bit structure of RCA

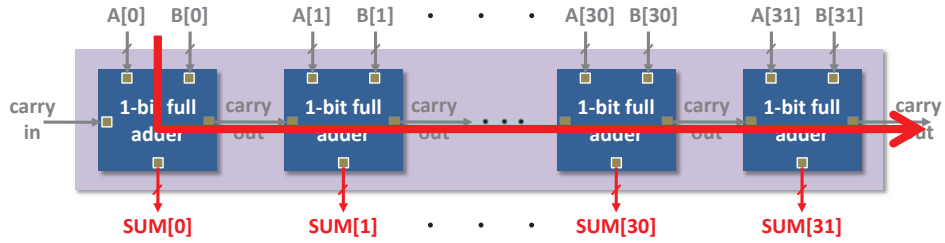


Figure 1.1: The ripple-carry adder(RCA).

On the other hand, carry look-ahead adder (CLA), can reduce critical delay caused by carry propagation of the RCA by fast carry calculation. We divide the addition in groups and each group does carry lookahead addition as shown in Fig. 1.2 because of limited faninout of the gate.

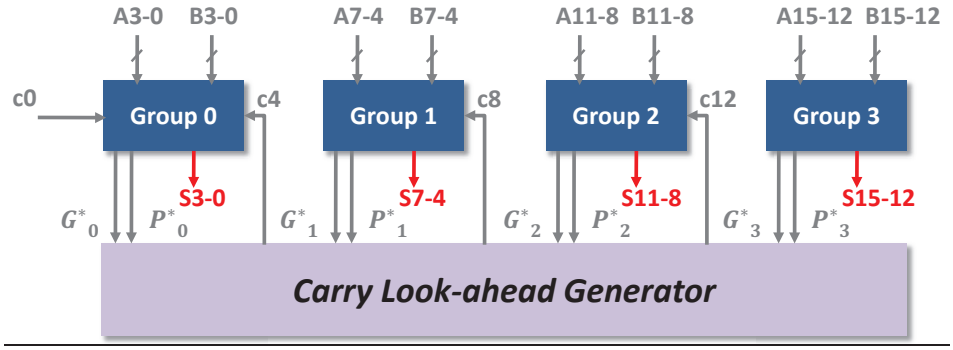


Figure 1.2: The carry look-ahead adder(CLA).

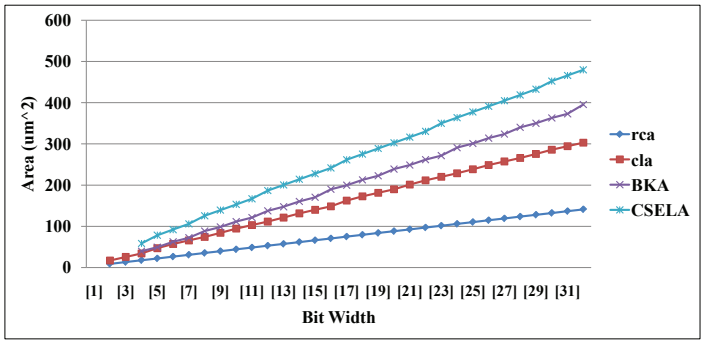
carry-skip adder (CSKA) [4], and carry-select adder (CSLA) [5] are another examples of pure adder designed for fast carry generation which utilize fast carry propagation or carry computation as carry look-ahead adder.

1.2 Parallel prefix adders

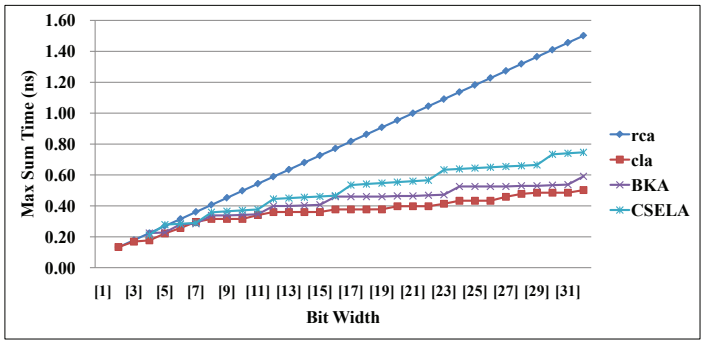
Parallel prefix adders, as the name implies, segmented addition is done in a parallel manner. So, faster addition can be achieved. In particular, various styles of parallel prefix adder scheme are a class of widely used fast pure adders; Brent-Kung adder (BKA) [6] is the simplest prefix adder, having a minimum number of prefix nodes but a maximum number of prefix depth in its prefix graph; Ladner-Fischer adder [7] has a low depth but a high fanout; Kogge-Stone adder (KSA) [8] has a low fanout but a large number of prefix nodes; Ling adder [9] is a variant of CLA, but can also be translated into prefix adder. (Refer to [10] for the details on the properties of the various styles of prefix adders.)

Fig 1.3 shows area, maximum sum output time and maximum carry output time of the adders with respect to bit width of the adders which are synthesized with TSMC $40nm$ standard cell library using Synopsys Design Compiler.

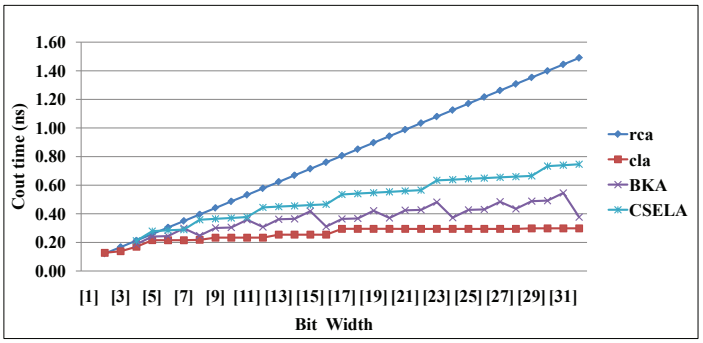
Some of adder design techniques placed their primary importance on minimizing power consumption rather than timing. For example, Mathew *et al.* [11] proposed a technique of synthesizing a low power adder to be used for address generation unit (AGU) in cores by utilizing sparse-tree adder structure. Contrary to Kogge-Stone adder [8], they divided the carry-merge tree into the critical and noncritical sections to reduce the size of node fanout and the length of inter-stage interconnect. Then, they allocated single-rail dynamic logic and high-V_T logic on the critical section, and static logic and low-V_T on the noncritical section to save power. Zlatanovici *et al.* [12] explored the energy and delay trade-off in carry look-ahead tree structure with 90nm technology. Zeydel *et al.* [13] also explored the energy and delay trade-off by taking into account adder topology, addition algorithm, gate sizing, and CMOS logic styles.



(a) Area of the adders with respect to bit width.



(b) Maximum sum output time of the adders with respect to bit width.



(c) Maximum carry output time of the adders with respect to bit width.

Figure 1.3: Comparison of area, maximum sum output time and maximum carry output time of the adders with respect to bit width of the adders.

1.3 *Hybrid* adders

On the other hand, the *hybrid* adders are those generated by applying more than one addition scheme used in the pure adder generations to the bit addends. The hybrid adders are mostly designed for adding the bit addends of non-uniform arrival times while the pure adders are designed assuming uniform arrival times of addends. Thus, for designs in which the inputs to an addition operation come from the result of a combinational logic computation rather than directly from the primary inputs or the outputs of flip-flops (FFs), a hybrid adder implementation could be more efficient than pure adders. This work also focuses on the problem of designing a (single or multiple) hybrid adders of minimal area under timing constraint, but it is unique and suitable for special applications that most of the previous works have never been addressed.

There have been proposed several schemes of designing fast hybrid adders. Han and Carlson [14] improved the parallel prefix scheme by combining the Brent-Kung and Kogge-Stone structures into a hybrid structure. Lynch and Swartzlander [15] proposed a new hybrid carry look-ahead - carry-select adder structure to reduce the number of carries that are to be derived in the carry look-ahead tree, in which they used (4,3) manchester carry chain (Mcc) carry look-ahead modules. Kantabutra [16] improved the hybrid adder in [15] by using Manchester carry chains of various lengths instead of chains of all the same length to speed up the addition. Wang, Pai, and Song [17] proposed a generalized architecture of hybrid carry look-ahead - carry-select adder in which their idea was to implement the group carry propagates and carry generators without individual carry propagate/generator signals and complement the group carry propagate/generator signals to gain speed. In addition, Dimitrakopoulos and Nikolos [18] proposed a hybrid structure composed of Kogge-Stone parallel prefix structure for carry generation and carry-select structure for sum calculation to further reduce timing over the pure parallel prefix adders. Lee *et al.* [19] also proposed a

method of synthesizing hybrid adders composed of two (RCA, CLA) or three (RCA, CSKA, CLA) pure sub-adders with the objective of minimizing timing under area constraint. They formulated the problem into an integer linear programming (ILP) and solved it optimally. All of the previously mentioned works [15–19] assumed that the arrival times of input bits are all identical (i.e., all zeros). Thus, it is unsure that such a hybrid adder can fully contribute to synthesizing arithmetic intensive circuits with tight timing budget if it were mapped to an addition operation on the timing critical path with severely uneven input arrival times.

1.4 *Hybrid* adders with timing constraints

A number of works have addressed the problem of designing hybrid adders with uneven input bit arrival times. Oklobdzija and Velleger [1] attempted to improve the timing of parallel multiplier [20] by proposing a critical path based column compression tree generation followed by creating a hybrid adder by analyzing the profile of bit arrival times of the two output vectors of compression tree to produce the final sum. They divided the arrival times into three regions from the least significant bit to the most significant bit, and attempted the first, second, and third regions of the bit intervals to implement with a ripple carry adder, a carry-select adder, and a carry look-ahead adder, respectively, so that the timings of all output bits are to be almost even. Fig. 1.4 shows input arrival timing to the final adder and adder allocation.

Stelling and Oklobdzija [21] generalized the work of hybrid adder design in [1] under the cases where the bit arrival times are convex such that the times gradually increases and then decreases from the least significant input bit to the most. In [22] they extended the hybrid adder design in [21], so that it can be used in the addition of the structure of Multiply-Accumulate operation (i.e., $A \times B + C$). On the other hand, Zimmermann [23] (also in [24]) proposed a parallel-prefix structure based synthesis

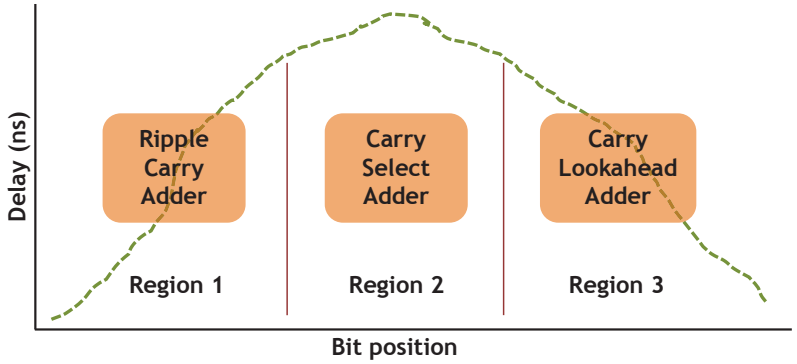


Figure 1.4: Input arrival timing to the final adder and adder allocation of [1]

flow of hybrid adder that consists of the following steps:

(1) translating timing constraints into prefix graph constraints

(2) generating a serial-prefix graph (3) compressing prefix graph

(4) performing depth-controlled prefix graph expansion

(5) mapping the prefix graph to prefix adder logic, using either

carry look-ahead or carry-select scheme. In summary, the approach generates a hybrid adder (in step 5) according to the pattern of prefix level structure of sub-ranges of bit addends. Here, the approach estimated timing by counting the structure depth in the prefix graph as shown in 1.5. So, it's not exact in timing estimation because the prefix graph can't represent the timing exactly.

Finally, Das and Khatri [2] proposed an approach to partitioning the input bits of the final addition in the parallel multiplier into three disjoint bit intervals, like the work done in [1], so that the first interval is implemented with a ripple carry adder, the second a Brent-Kung, and the third a carry-select adder. The two common limitations of the previous works [1,2,21,22] are that (limitation-1) the hybrid adder design is dedicated to the adders associated with special architecture: parallel multiplier and Multiply-Accumulator, and (limitation-2) the bit-level output required times are not considered

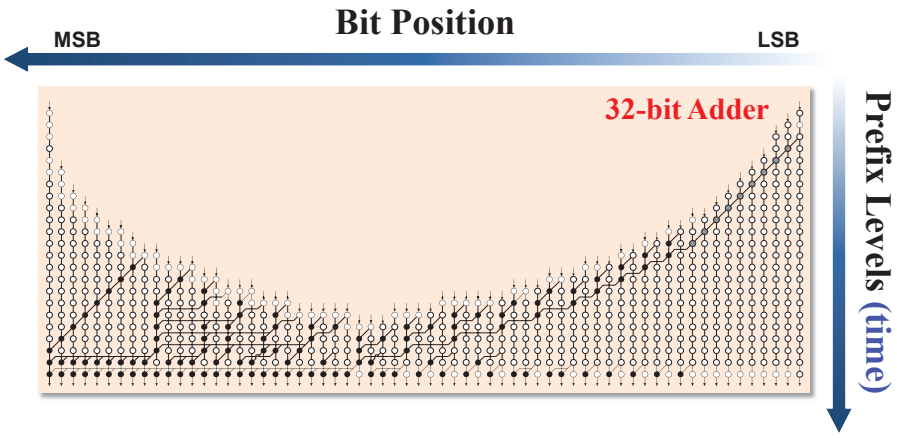


Figure 1.5: The prefix graph example.

at all. Limitation-1 inhibits a wide application of hybrid adders to arithmetic circuits and limitation-2 reduces the suitability of re-optimizing adder in a circuit to further improve the overall circuit timing.

1.5 Contribution of this dissertation

This dissertation overcomes previous limitations. The contributions of this dissertation are summarized as:

- This dissertation propose a hybrid adder design scheme that takes into account the required time of output bits of the addition as well as the input bit arrival time. This means that the synthesized hybrid adder will be the one that satisfies the output required timing while minimizing the implementation area. For some design stages at which the entire system has been almost implemented and the critical timing is on a path containing arithmetic logic with addition, resynthesizing solely the adder in the logic using our scheme would be greatly useful.

- Proposed adder design method is not confined to specific architectures. Proposed design scheme covers all the applications in which an addition operation is involved. In addition, proposed strategy of synthesizing a hybrid adder is theoretically optimal (under a certain timing property of pure adders), as formally described in Theorem 4.5, in that the selection and usage of various pure adders are exhaustively explored. In practice, the quality is well-controlled with reasonably acceptable running time through the invention of a systematic exploration of design alternatives and pruning techniques.

- The proposed method also propose an extended adder design scheme that simultaneously synthesizes two hybrid adders of parent-child dependency relation where the conventional addend compression techniques such as CSA-tree (carry-save adder tree) and FA-tree (full-adder tree) constructions are not adequately applicable, to fully exploit the combined benefit of hybrid adders on minimizing area under tight timing constraint.

- Synthesis results on diverse arithmetic expressions are given in experiments to show the usefulness and feasibility of our proposed scheme, confirming that the previous pure adder and hybrid adder schemes never meet the timing constraint or meet timing constraint with substantially large addition logic whereas our scheme creates hybrid adders that are well customized to the whole designs, satisfying the timing constraint with much smaller addition logic.

The two main tasks that are performed in the RTL/logic synthesis of arithmetic circuits are the implementation (or adder structure) selection and cell (or gate) sizing, in which due to the run time problem the two tasks are practically performed sequentially, the implementation selection first, then cell sizing. We target our hybrid adder scheme to enhancing the task of implementation selection where we assumed to use a single implementation with moderate timing/area for each pure adder scheme that is

used to form a hybrid adder. However, it should be noted that our scheme can also partially consider the task of cell sizing by including in our pure adder (implementation) library multiple (cell sized) implementations for each pure adder scheme. Throughout our presentation, we simply use a single implementation of moderate timing/area for each pure adder scheme, but we also use one more differently cell sized (fast timing but large area) implementation for each pure adder scheme.[1] Note that theoretically our adder scheme can perfectly perform the two tasks simultaneously by considering all possible cell sized implementations, but practically our scheme rather focuses on the implementation selection using typical (or representative) timing/area numbers for each pure adder, and a fine-grained cell sizing will resort to a subsequent step under the selected implementation.

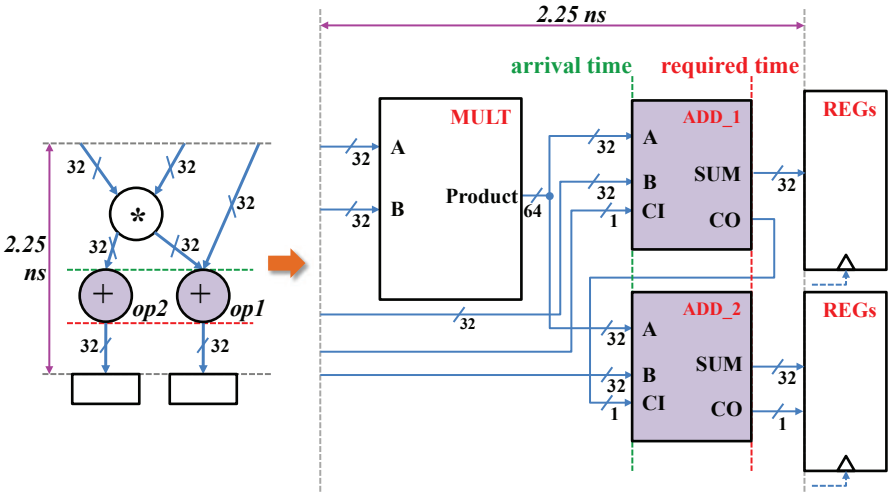
[1]The differently cell sized implementations were obtained by using Synopsys Design Compiler with the use of `set_max_delay 0 all_outputs()` command.

Chapter 2

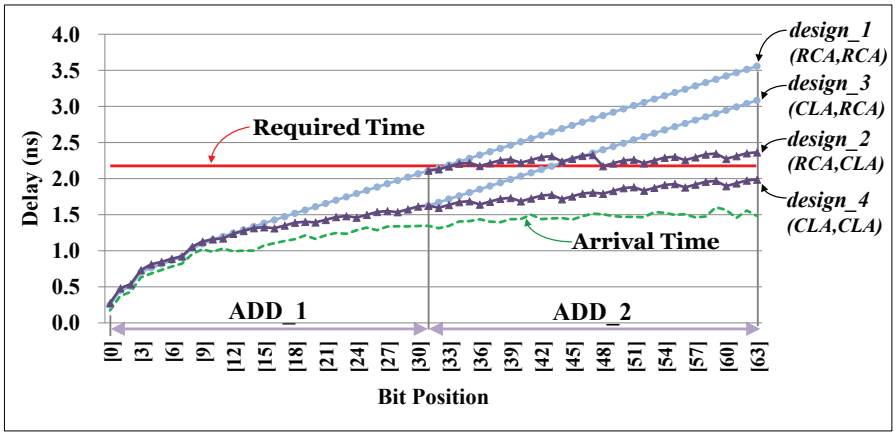
Motivational Examples

This section illustrates, using examples, the usefulness of hybrid adders and the limitations.

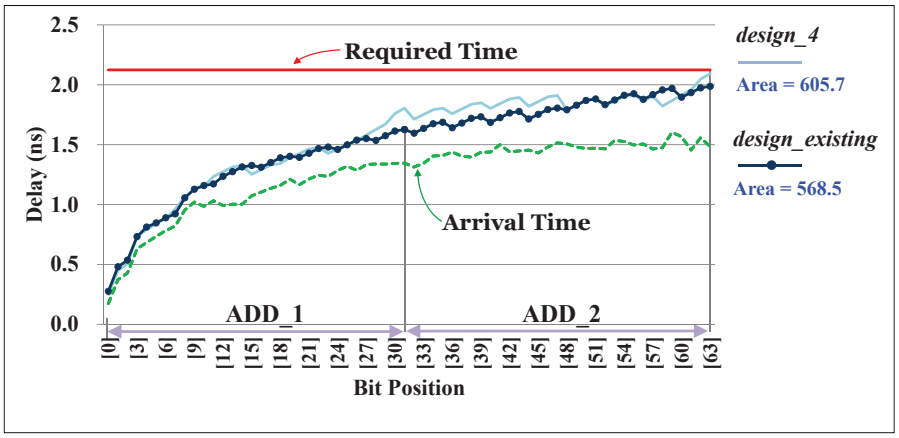
• *Synthesizing an adder with non-uniform input bit arrival times but uniform output bit required times*: Fig. 2.0(a) shows the dataflow graph (DFG) of a simple arithmetic expression $a * b + c$ and its architectural implementation diagram. The diagram (and DFG) shows a chained multiplication-and-addition where the lower 32-bit of the 64-bit output of multiplication is fed to addition *op1* and the upper 32-bit is fed to addition *op2*. The *op1* and *op2* are bound to adders ADD_1 and ADD_2, respectively. Suppose that the arithmetic expression has the required timing constraint of 2.25ns. The four curves labeled *design_1*, *design_2*, *design_3*, and *design_4* in Fig. 2.0(b) show the output timing profiles of the data path in Fig. 2.0(a) when (ADD_1, ADD_2) is implemented by adders (RCA, RCA), (RCA, CLA), (CLA, RCA), and (CLA, CLA), respectively. (To extract timing of the logical and physical implementations we used Synopsys Design Compiler and Prime Time with TSMC 40 nm standard cell library.) The dotted green curve in Fig. 2.0(b) represents the arrival times to the input ports of ADD_1 and ADD_2 with respect to the bit positions, and the red line indicates the



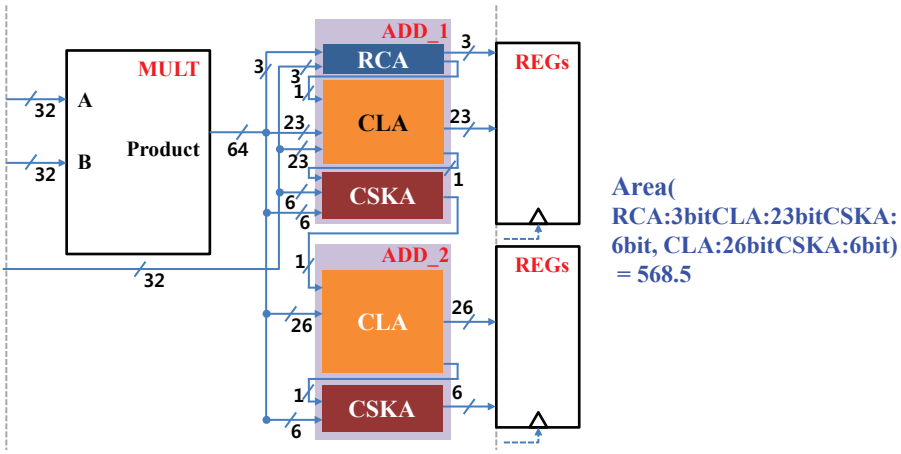
(a) A DFG representation for $a * b + c$ and its architectural block diagram.



(b) Output timing profiles of four possible adder implementations. Only *design_4* satisfies the timing requirement of 2.25ns.



(c) Output timing profiles of the reimplemented design (*design_existing* by replacing the CLAs for ADD_1 and ADD_2 of *design_4* in (b) with hybrid adders created by the hybrid adder scheme in [2].



(d) Architectural block diagram of the reimplemented design *design_existing* in (c). It is seen that the two adders are customized into hybrid adders, reducing area under the timing constraint.

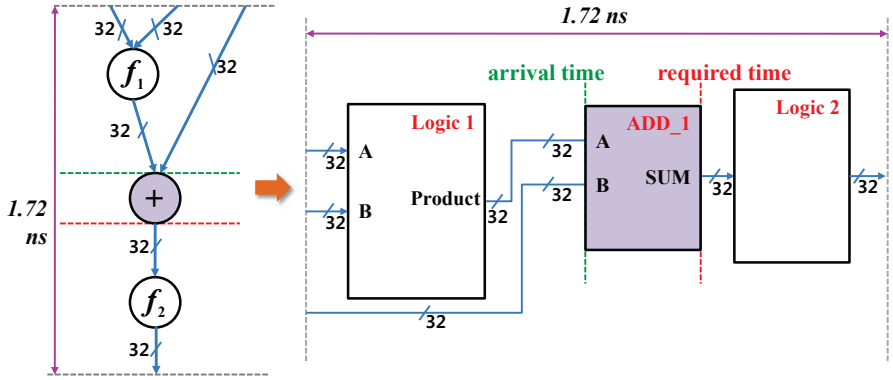
Figure 2.0: Synthesizing a hybrid adder in the context of designing $a * b + c$ where the arrival times of input bits to the additions are not even (due to the multiplication) while the required times of the output bits of the additions are all equal, setting to $2.25ns$. (In extracting the timing of logical and physical implementations, Synopsys Design Compiler and Prime Time with TSMC $40nm$ standard cell library are used.)

output required timing of ADD_1 and ADD_2 to meet the timing constraint of design. Since only *design_4* meets the timing constraint, both of ADD_1 and ADD_2 should be implemented with CLAs, resulting in the area of 605.7μm^2. Let us now consider to reimplement ADD_1 and ADD_2, which were mapped to CLAs, with hybrid adders produced by applying the scheme in [2]. The dark blue curve labeled *design_existing* in Fig. 2.0(c) shows the output timing profile of the data path in Fig. 2.0(a) when the addends of bit positions 0 to 2 are added by RCA, the addends of bit positions 3 to 25 are added by CLA, and the addends of bit positions 26 to 31 are added by carry-skip adder(CSKA) for ADD_1. For ADD_2, the addends of bit positions 0 to 25 are added by CLA, and the addends of bit positions 26 to 31 are added by CSKA. (The reason why we replaced Brent-Kung adder (BKA) with CLA and carry-select adder (CSLA) with CSKA was that BKA and CSLA needed much bigger area cost for the implementation in our experiments.) Fig. 2.0(d) shows the resulting implementation, which still satisfies the timing constraint, but reduces the logic area from 605.7μm^2 to 568.5μm^2, which is about 18.2% reduction. This example shows that a proper use of a hybrid adder scheme can reduce the circuit timing or area further. However, all the existing hybrid adder schemes always assume *uniform required times* of all output bits, as shown the straight red lines in Fig. 2.0.

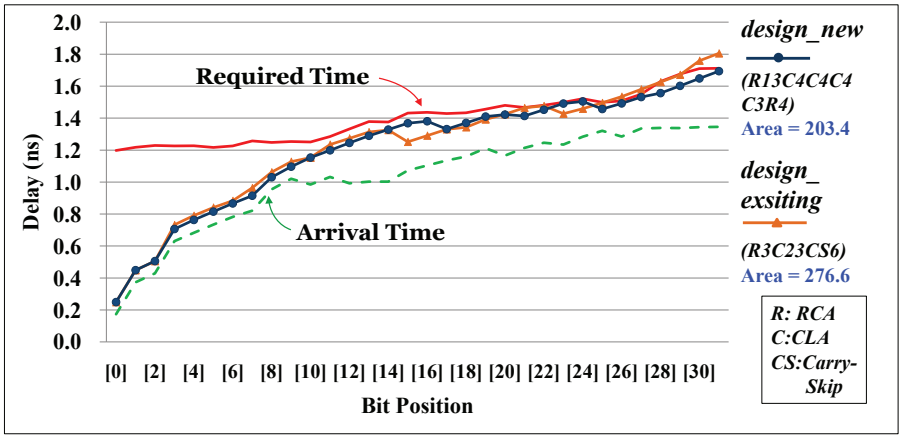
• *Synthesizing an adder with both non-uniform output required and input arrival times*: Let us consider another data path in Fig. 2.1(a) with timing constraint of 1.72ns, the structure of which is slightly different from that in Fig. 2.0(a); besides some logic f_1 connected to the inputs of ADD_1, there is another, possibly irregular structure of combinational logic f_2 connected to the outputs of ADD_1. Thus, the required times of the output bits of ADD_1 will not be the same, as indicated by the red curve in Fig. 2.1(b). Since existing schemes of hybrid adder design including that used in *design_existing* in Fig. 2.0(d) simply assume uniform required times of all output bits, it is hard to find a best (area-minimal) implementation that meets the timing constraint, in most

cases failing in meeting the timing constraint. The curve labeled *design_existing* in Fig. 2.1(b) shows the output timing profile. On the other hand, the curve labeled *design_new* in Fig. 2.1(b) shows the output timing profile of ADD_1, where ADD_1 is reimplemented with a hybrid adder consisting of RCA for bits 0 to 12, CLA for bits 13 to 16, CLA for bits 17 to 20, CLA for bits 21 to 24, CLA for bits 25 to 27 and RCA for 28 to 31. As a result, the area is reduced from 276.6μm^2 to 203.4μm^2, which is about 26.5% reduction while satisfying the timing requirement of all output bits. This example clearly shows that in order to produce an optimal hybrid adder, it is very important to take into account the bit-level output required times as well as the input arrival times.

• *Synthesizing multiple adders with both non-uniform output bit required and input bit arrival times*: Fig. 2.1(a) shows two chained additions *op1* and *op2* surrounded by another logic components f_1, f_2 and f_3, and the corresponding architectural block diagram. Thus, for both ADD_1 and ADD_2, their output required times are not even. The four curves in Fig. 2.1(b) show the output timing profiles of the implementations of (ADD_1, ADD_2) with (RCA, RCA), (RCA, CLA), (CLA, RCA), and (CLA, CLA). We can see that only the (CLA, CLA) implementation satisfies the required timing, resulting in the area of 605.7μm^2. Now, we reimplement the (CLA, CLA) in Fig. 2.1(b) by using the hybrid adder scheme in [2], where ADD_1 is reimplemented first and ADD_2 is implemented later using the input arrival times from the sum output bits of ADD_1. The output timing profile of the resulting implementation is shown in the curve labeled *design_existing* in Fig. 2.1(c). It does not meet the required timing constraint. On the other hand, the curve labeled *design_new* shows the output timing profile of another optimized hybrid implementation, generated *while considering the timing inter-dependency between the two hybrid adders*. Note that the implementation satisfies the output required time and even reduces the area by 42.4% further compared to (CLA, CLA). This comparison illustrates that simultaneously synthesizing multiple

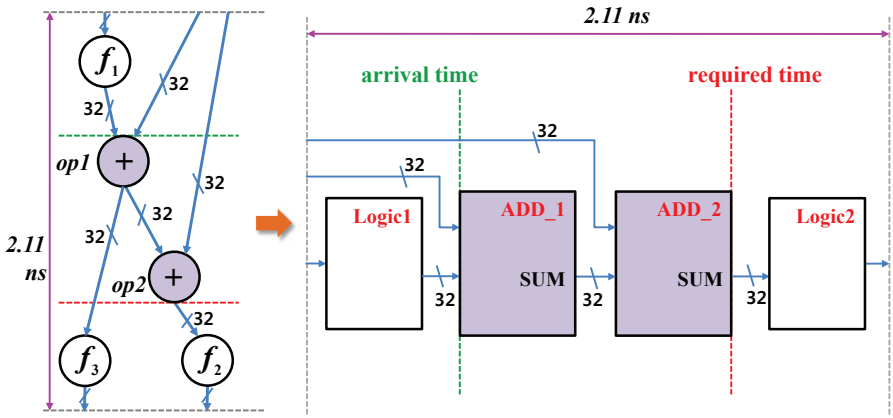


(a) Addition operation in between two logic clusters f_1, f_2, and the architectural block diagram.

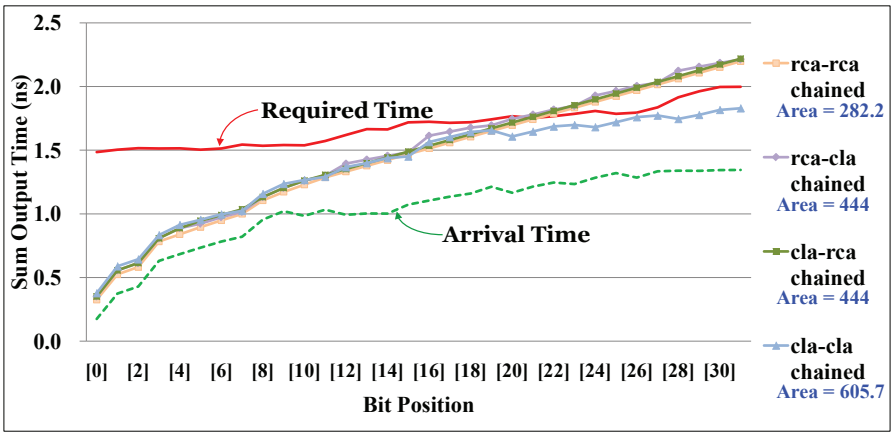


(b) Output timing of *design_existing* produced by the hybrid adder scheme in [2] and *design_new* produced by a further elaborated hybrid adder design method.

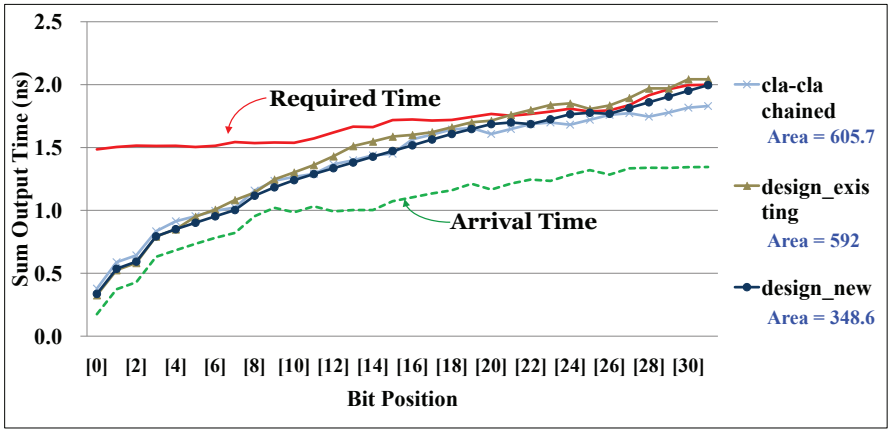
Figure 2.1: Synthesizing a hybrid adder in the context of both non-uniform output bit required and input bit arrival times. It is observed that when the output required times are not uniform a highly sophisticated hybrid adder design is expected in order to make the timing constraint be met.



(a) Arithmetic expression for $a + b + c$ surrounded by logic clusters f_1, f_2, f_3 and the architectural block diagram.



(b) Output timing of four possible adder implementations. Only *CLA-CLA chained* satisfies the timing requirement.



(c) Output timing of *design_4* when ADD_1 followed by ADD_2 are reimplemented by the adder scheme in [2] and ADD_1 and ADD_2 are simultaneously reimplemented by proposed scheme.

Figure 2.1: Synthesizing multiple hybrid adders.

hybrid adders is essential to fully exploit the benefit of hybrid adders on optimizing timing or minimizing area under tight timing constraint.

Chapter 3

Definitions and Design Flow

3.1 Notations and Definitions

Our hybrid design scheme is to partition the bit addends, so that each partitioned bit addends are added by a pure adder. That is, an n-bit hybrid adder, which we call *adaptable hybrid adder* (AHA), produced by our scheme is a concatenation of sub-adders and each sub-adder is implemented by an adder scheme in $\mathcal{L}$, where $\mathcal{L}$ is the set of pure adder implementation schemes. (A pure adder scheme can have multiple implementations with different cell sizing. In our presentation we simply assume to use a single implementation of moderate timing/area for each pure adder scheme, but in the experiments we include not only the results produced by using the single implementations only but also the results by using multiple implementations for each adder scheme. The details will be explained in the experimentation section.)

Adaptable hybrid adder is recursively defined as:

Definition 3.1.1 (Adaptable hybrid adder AHA) *For a set $\mathcal{L}$ of pure adder implementation schemes, an n-bit adder A is called* **adaptable hybrid adder** $AHA(0, n -$

1) *if one of the following two conditions is satisfied.*

1. A is a pure adder PA_{λ_i} implemented by some scheme $\lambda_i \in \mathcal{L}$;
2. A is decomposable into two sub-adders $A_1(0, k-1)$ and $A_2(k, n-1)$ such that the carry out of A_1 is connected to the carry in of A_2, and $A_1(0, k-1)$ is a k-bit adaptable hybrid adder (which is recursively defined) and A_2 is a pure adder that is implementable by a scheme $\lambda_i \in \mathcal{L}$, for some k, $1 \leq k \leq n-1$.

We call the AHA A_1 in Definition 3.1.1 *head* of AHA A and the pure adder A_2 *tail* of A. In addition, we use notations A_{head} and A_{tail} to indicate the head and tail sub-adders of AHA A, respectively, and $A_{head} \parallel A_{tail}$ to indicate A. A_{head} is empty if A is a pure adder.

Table 3.1 lists the notations to be used in our presentation. The visual description of the notations is shown in Fig. 3.1, where a 16-bit addition of X and Y is implemented with a hybrid adder $AHA(0, 15)$ that is composed of a hybrid adder $AHA(0, 11)$ which is the head of $AHA(0, 15)$ and a pure adder $PA_{\lambda_1}(12, 15)$ which is the tail of $AHA(0, 15)$. $AHA(0, 11)$ is further decomposed into head $AHA(0, 8)$ and tail $PA_{\lambda_2}(9, 11)$ of $AHA(0, 11)$.

The problem we want to solve can be described as:

Problem 3.1.2 *[Generating an adaptable hybrid adder] Given a library $\mathcal{L}$ of pure adder schemes, two n-bit operands $X[0 : n-1]$ and $Y[0 : n-1]$ to be added with their arrival times $a(\cdot)$s, and the required timing constraint $\Gamma[0 : n-1]$ of sum outputs, find a structure of $AHA(0, n-1)$ that minimizes the value of $Cost(\cdot)$[1] of the AHA while satisfying the timing constraint Γ.*

[1] $Cost(\cdot)$ is the total area of the implemented AHA in this work, but can be any design parameters such as power consumption.

Table 3.1: Description of notations.

Notation	Description
$\mathcal{L} = \{\lambda_1, \lambda_2, \cdots, \lambda_M\}$	The set of pure adder schemes (e.g., $\lambda_1 = RCA$, $\lambda_2 = CLA$, ...)
PA_{λ_i}	An adder implemented by a pure adder scheme λ_i.
$X[i:j] = [x_i, x_{i+1}, \cdots, x_j]$	Input operand X of addition.
$Y[i:j] = [y_i, y_{i+1}, \cdots, y_j]$	Input operand Y of addition.
$S[i:j] = [s_i, s_{i+1}, \cdots, s_j]$	Sum output S of $X[i:j] + Y[i:j]$.
$cin(i)$	carry in to be added at bit position i.
$cout(j)$	carry out generated from bit position j.
$a(e)$	Arrival time of signal e. (e is cin, $cout$, or a signal in X, Y, S.)
$\Gamma[i:j] =$ $[r(s_i), r(s_{i+1}), \cdots, r(s_j)]$	Required times of sum output bits.
$AHA(i,j)$	An AHA for $X[i:j] + Y[i:j]$, under $a(\cdot)$, $\Gamma[i:j]$, and $\mathcal{L}$.
$Cost(A)$	The implementation cost of adder A.
A_{head}	The head sub-adder of AHA A.
A_{tail}	The tail sub-adder of AHA A.
$\|A\|$	The bit-width of an addend of adder A.[a]

[a] We assume that the two input addends in an adder have the same bit widths.

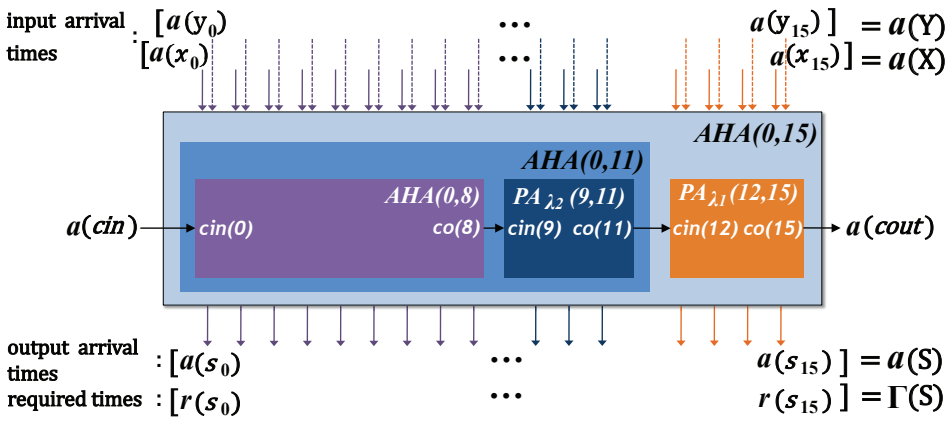


Figure 3.1: Visual description of the notations in Table 3.1 for an AHA(0,15) and its internal structure.

Definition 3.1.3 (Feasible AHA) *Given a library $\mathcal{L}$ of adder schemes and required timing $\Gamma[0:n-1]$, an adder that performs $X[0:n-1]+Y[0:n-1]+cin$ with the bit-level arrival times $a(X[0:n-1])$, $a(Y[0:n-1])$, and $a(cin)$[2] of carry in cin is a* **feasible** *AHA if it is an AHA implementation using $\mathcal{L}$ and satisfies $a(S[0:n-1]) \leq \Gamma[0:n-1]$ where $a(S[0:n-1])$ is the bit-level arrival times of sum output vector $S[0:n-1]$ of the adder.*

Then, Problem 3.1.2 is to find a minimum-cost feasible AHA under $\mathcal{L}$, the input arrival times $a(\cdot)$s, and the output required times Γ.

[2]For adding $X+Y$, we can assume $cin = 0$ and $a(cin) = 0$.

Chapter 4

Synthesis of Adaptable Hybrid Adders

The basic idea of generating an n-bit AHA of minimum cost is, starting from a set of feasible AHAs of smallest bit-width, to incrementally update a set of current candidates of AHAs until the bit-width of the updated AHAs reaches n.[1] Our AHA synthesis is targeted to two scopes of addition: (case 1) resynthesizing an isolated (single) adder on the critical timing path of circuit; (case 2) resynthesizing chained (multiple) adders together on the critical timing path. Our AHA synthesis approach to case 1 is a special case of case 2. Fig. 4.1 shows the flow of design methodology which utilizes our AHA synthesis.

The first pass of synthesis for the initial HDL code returns the timing information on the input arrival times and the output required times of an adder(s). Then, our adder scheme produces an area-optimized hybrid adder structure(s) that satisfies the timing constraint extracted in the first pass of synthesis. The second pass of synthesis is then performed for the initial HDL code again while preserving the hybrid adder structure(s) obtained by our AHA scheme.

[1] A discussion about theoretical background on our AHA synthesis approach is given at the end of subsection 4.1.

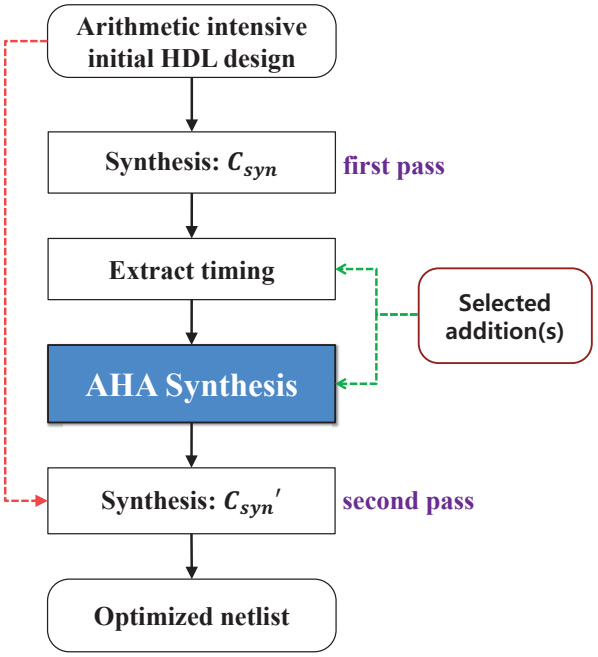


Figure 4.1: The flow of design methodology using our AHA synthesis. Both of the first and second passes of synthesis use the initial HDL design code as input, but the second pass will preserve the hybrid adder structure(s) produced by the AHA synthesis. The selection of addition(s) to be optimized will be controlled by designer.

In the following two subsections, we proposes our AHA synthesis schemes for the two scopes of addition.

4.1 Synthesizing Single Adaptable Hybrid Adder

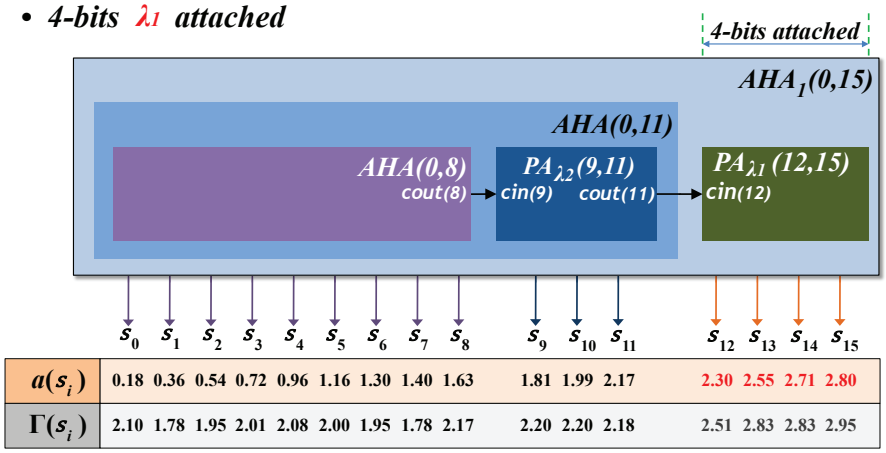
Our AHA synthesis scheme is an incremental approach. At each iteration, with a set of all feasible AHA implementations of a certain size k of input bits, we want to generate a set of all feasible AHA implementations of an input size greater than k. The generation is based on the notions of $\Delta d\text{-}attachable$ and $\Delta d\text{-}extendable$.

Definition 4.1.1 (Δd**-**$attachable$ **AHA)** *Let $A(0, k-1)$ be a feasible AHA under the required timing of the first k bits in $\Gamma[0 : n]$ ($k + \Delta d \leq n$) and $B(0, \Delta d - 1)$ be a Δd-bit ($\Delta d > 0$) pure adder. Then, it is said that B is $\Delta d\text{-}attachable$ to A if the $(k+\Delta d)$-bit $AHA(0, k+\Delta d-1)$ constructed by connecting the carry out of A to the carry in of B is feasible under the required timing of the first $(k + \Delta d)$-bit in $\Gamma[0 : n]$.*

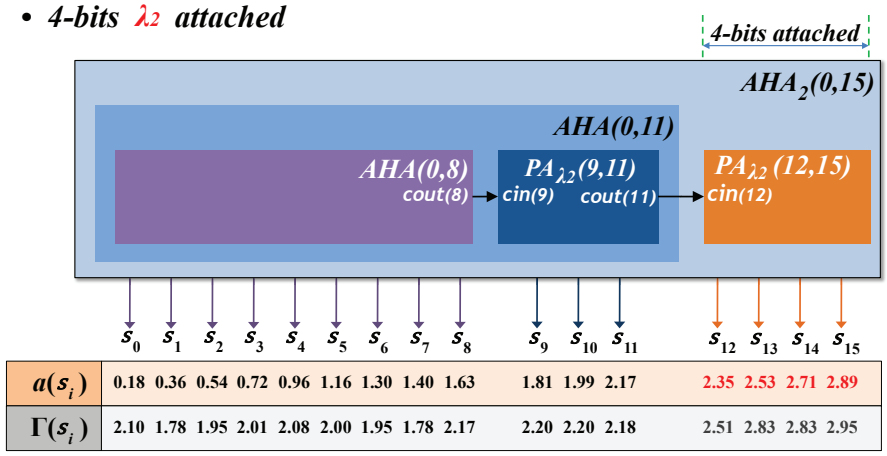
Definition 4.1.2 (Δd**-**$extendable$ **AHA)** *Let $A(0, k+s-1)$ be a $(k+s)$-bit feasible AHA under the required timing of the first $(k + s)$-bit in $\Gamma[0 : n]$ ($k + s < n$) and $B(k, k + s - 1)$ be the tail sub-adder of $A(0, k + s - 1)$ implemented by scheme λ_i. Then, it is said that B is $\Delta d\text{-}extendable$ from A if the $(k + s + \Delta d)$-bit $AHA(0, k + s + \Delta d - 1)$ $(k + s + \Delta d \leq n)$ constructed by replacing the s-bit $B(k, k + s - 1)$ with the $(s + \Delta d)$-bit adder implemented by λ_i is feasible under the required timing of the first $(k + s + \Delta d)$-bit in $\Gamma[0 : n]$.*

Figs. 4.1(a) and (b) show examples of 4-attachable and 4-extendable pure adders to and from AHA(0,11), respectively. As seen from the examples, for Δd-attachment any pure scheme of sub-adder can be considered whereas for Δd-extension the scheme of sub-adder should be the matched with the scheme of tail adder. Note that each of

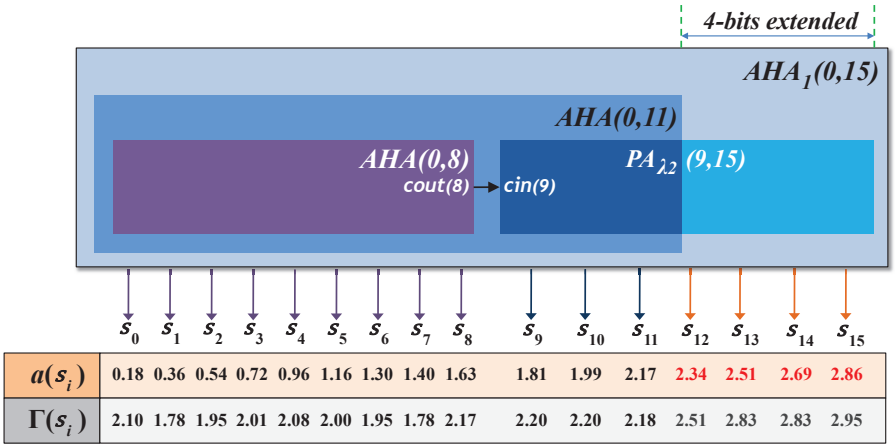
- *4-bits λ_1 attached*



- *4-bits λ_2 attached*



(a) Feasible AHA(0,15)s which are formed by connecting the carry out of the tail (i.e., $PA_{\lambda 2}(9, 11)$ of a feasible AHA(0,11) to carry in of a 4-bit pure adder.



(b) A feasible AHA(0,15) which is formed by extending the bit-width of the tail (i.e., $PA_{\lambda 2}(9, 11)$) of AHA(0,11) 4-bit more to become $PA_{\lambda 2}(9, 15)$.

Figure 4.1: Examples of (a) 4-attachable AHA pure adder to an AHA and (b) 4-extendable pure adder from an AHA.

the resulting AHA(0,15)s is feasible, which means that its output bits all satisfy the required timing constraint.

Fig. 4.2 shows the iteration flow of synthesizing an n-bit AHA, starting from a set of all feasible l_0-bit (pure) adders. At each iteration, from a set of all feasible l-bit AHA adders, a set of $(l + \Delta d)$-bit feasible AHAs is derived by applying all possible Δd-attachments and Δd-extensions to/from each of the l-bit AHAs.

(Note that the pure adder library $\mathcal{L}$ contains implementation netlist for every implementation, and the netlist is attempted to be attached and extended for every iteration of the AHA synthesis to extract the resulting bit-level output timing.)

Let S_i be the set of feasible AHAs produced in the ith iteration. Then, in the iteration since every adder in S_i is tested one time for the extension, but tested for attachment on all types of pure adder in $\mathcal{L}$, we have $|S_{i+1}| = O(|S_i| \cdot (|\mathcal{L}| + \infty))$. Thus, the total number of AHA adders considered for attachment and extension dur-

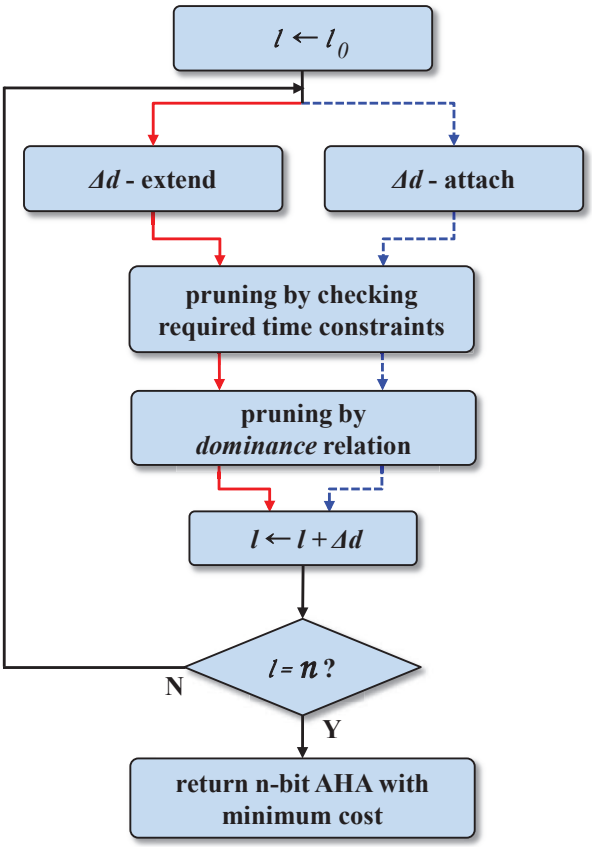


Figure 4.2: The iteration flow of synthesizing an n-bit single AHA.

ing iterations is exponentially bounded as shown in Fig. 4.3 . The following notion of dominating/dominated AHA helps drastically prune the unnecessary partial AHAs generated during the iterations while retaining the quality of final (n-bit) AHA.

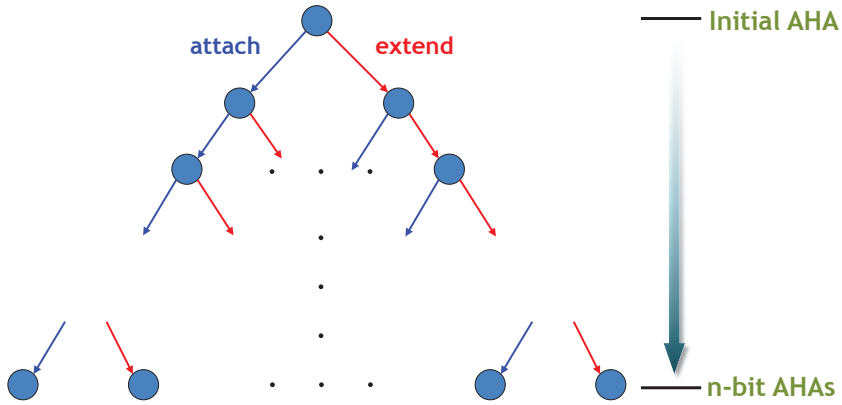


Figure 4.3: Exponential growth of serach space.

Definition 4.1.3 (Dominating/dominated AHA) *Let A and B be two feasible AHAs such that $|A| = |B|$ under $\mathcal{L}$, Γ, and input arrival times $a(\cdot)$. Then, it is said that A is* dominating *B (or B is* dominated *by A) if (1) $Cost(A) \leq Cost(B)$, (2) $|A| = |B|$, and (3) $a(cout_{A_{tail}}) \leq a(cout_{B_{tail}})$.*

The three conditions (1-3) ensure that the dominated AHAs can be removed from the set S_i of partial AHAs before performing the next iteration. For example, Fig. 4.4 shows the generation of set S_{i+1} of partial AHAs by attachment and extension from/to the partial AHAs in S_i and the process of removing the dominated AHAs in S_{i+1}. In this example, there are four 12-bit intermediate AHAs A_1, A_2, A_3 and A_4. A_1 dominates A_2 and A_4 in terms of area cost. A_1 also dominates A_3 and A_4 in terms of $a(cout_{A_{tail}})$ time; Thus, A_4 is removed from the consideration of the attachment and extension on the next iteration.

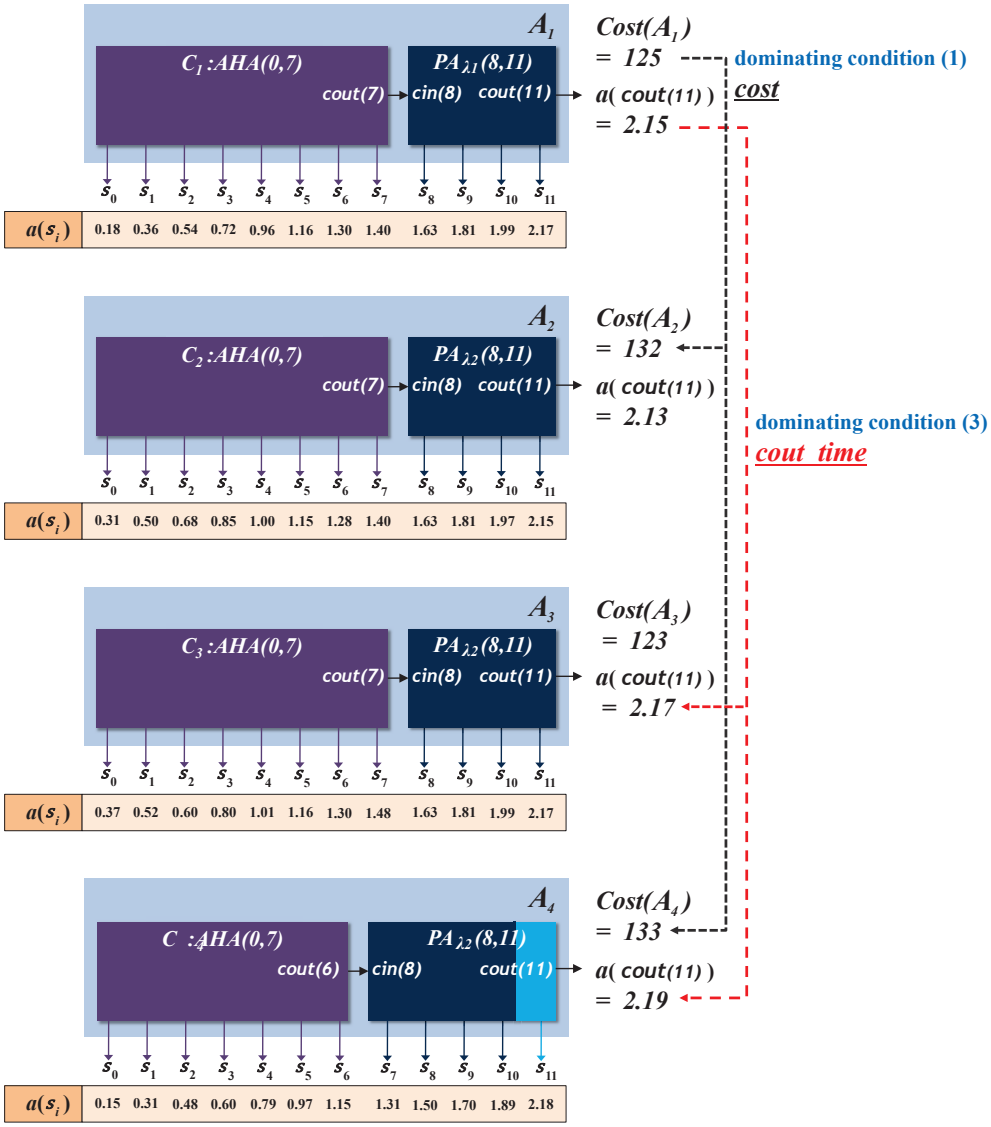


Figure 4.4: An example of pruning dominated AHAs. The AHA A_1 dominates the AHA A_4 since A_1 has earlier timing of carry out indicated by red dotted arrow and smaller area indicated by the black dotted arrow, thus A_4 can be removed safely.

The number of iterations is $\lceil \frac{n}{\Delta d} \rceil$. The value of Δd (usually $3 \sim 6$) is given by designer. The smaller the value of Δd is, the more likely the resulting n-bit AHA satisfies the timing constraint. Note that the final AHA generated is the one of minimum cost among all implementations of hybrid adders which have the form of the concatenations of *any number* of pure adders of *any bit-width of multiple* Δd, and satisfy the required (possibly non-uniform) output times.

Discussion about theoretical background: Theoretically, problem 3.1.2, which is to generate an optimal structure of adaptable hybrid adder for given input arrival times and output required times, can be formulated into dynamic programming, by which the optimal structures of various smaller bit-widths are examined and used to construct an optimal structure of a bigger bit-width. Thus, apparently, the total number of computation is $O(n^2)$. (Basically, our AHA scheme can also accomplish this by performing only the tail attachment for every value of Δd (i.e., $\Delta d = 1, 2, \cdots, n$). However, such a simple construction cannot measure the change of output bit times due to the unknown output load capacitance of the carry out signal when performing the attachments. It can only be measured accurately by really resynthesizing the corresponding addition logic. Since each resynthesis is computationally expensive, our AHA scheme devises a number of simplification and pruning techniques. Those are using a fixed value of parameter Δd for attachment, introducing Δd-extension in addition to the Δd-attachment, and utilizing dominance relation. The optimality of our AHA synthesis is stated below.

Definition 4.1.4 (Excess-O(n) property) *Let $A(i,j)$ and $A(i,k)$, and $B'(k+1,j)$, $i \leq k < j$ be three pure adders implemented by the same scheme in $\mathcal{L}$ which use the same input addends and carry in, except that $B'(k+1,j)$ uses, as carry in, a cin' with $a(cin') \leq a(cout_{A(i,k)})$. Then, it is called that $\mathcal{L}$ satisfies* **Excess-O(n) perperty** *if for every scheme in $\mathcal{L}$, the corresponding three adders can satisfy the inequali-*

ties of $Cost(B'(k+1,j) \leq Cost(A(i,j) - Cost(A(i,k)$ *and* $a(cout_{B'(k+1,j)}) \leq a(cout_{A(i,j)})$.

The *area cost inequality* of *Excess-O(n)* property holds for nearly all adder schemes since RCA, which is known to be the most area efficient adder scheme for all values of bit-width n, follows a consistent linear curve on area. On the other hand, the *timing inequality* of *Excess-O(n)* does not hold if the carry propagation delay curve is below a linear line, for example, $log_b n$ curve for large values of bit-width n for CLA where b is the blocking factor. However, since the size of partial pure adders used to form a hybrid adder is quite small, which is usually less than 8 bits in practice, the timing property will be satisfied for most of adder schemes. In particular, RCA satisfies the property for *all* sizes.

Theorem 4.1.5 *If the pure adder set* $\mathcal{L}$ *satisfies Excess-O(n) property, the proposed AHA generation scheme in Fig. 4.2 using* $\Delta d = 1$ *will always find, if there exists, an optimal AHA.*

Proof. Let $A(0,n)$ be an optimal AHA. Then, it suffices to show: *every sub-AHA* $A(0,k)$, $k = 0, \cdots, n$ *of* $A(0,A)$ *will never be pruned by the dominating/dominated relation.*

Suppose $B(0,k)$ is not a partial AHA of an optimal $AHA(0,n)$ and dominates $A(0,k)$. (It means $Cost(B(0,k)) \leq Cost(A(0,k))$ and $a(cout_{B(0,k)}) \leq a(cout_{A(0,k)})$.) Then, $A(0,k+1)$ of optimal $A(0,n)$ belongs to one of the two cases: (case 1) $A(0,k+1)$ has $1 - attachment$ relation with $A(0,k)$; (case 2) $A(0,k+1)$ has $1 - extension$ relation with $A(0,k)$.

For case 1, we replace $A(0,k)$ in $A(0,n)$ with $B(0,k)$ to form a new AHA $B(0,k)||A(k+1,n)$. Since $a(cout_{B(0,k)}) \leq a(cout_{A(0,k)})$, $A(k+1,n)$ which uses $cout_{B(0,k)}$ as carry in is still feasible. But, since $Cost(B(0,k)) \leq Cost(A(0,k))$,

$Cost(B(0,k)||A(k+1,n)) \leq Cost(A(0,n))$, which contradicts the assumption that $B(0,k)$ is not a partial AHA of an optimal $AHA(0,n)$.

For case 2, let $A(i,j)$ be the pure adder in $A(0,n)$ such that $i \leq k < j$. We create a pure adder $B'(k+1,j)$ by using $cout_{B(0,k)}$ as its carry in. Then, by the *Excess-O(n)* property of the adder schemes in $\mathcal{L}$, $Cost(B'(k+1,j)) \leq Cost(A(i,j)) - Cost(A(i,k))$ since $a(cin_{B'(k+1,j)}) \leq a(cout_{A(i,k)})$. Thus, if we replace $A(k+1,j)$ in $A(0,n)$ with $B'(k+1,j)$, the resulting AHA $A(0,k)||B'(k+1,j)||A(j+1,n)$ is still feasible because $a(cout_{B'(k+1,j)}) \leq a(cout_{A(i,j)})$, and satisfies $Cost(A(0,k)||B'(k+1,j)||A(j+1,n)) \leq Cost(A(0,k)) + (Cost(A(i,j)) - Cost(A(i,k))) + Cost(A(j+1,n))$ = $(Cost(A(0,i-1)) + Cost(A(i,k)) + Cost(A(i,j)) - Cost(A(i,k)) + Cost(A(j+1,n))$ = $Cost(A(0,i-1)) + Cost(A(i,j)) + Cost(A(j+1,n)) = Cost(A(0,n))$, which contradicts the assumption that $B(0,k)$ is not a partial AHA of an optimal $AHA(0,n)$.

4.2 Synthesizing Multiple Adaptable Hybrid Adders

This section describes a scheme of (simultaneously) synthesizing AHAs for multiple additions that are directly cascaded through the critical path of circuit. Obviously, applying the single AHA synthesis scheme proposed in the previous section to the adders sequentially one by one would not be sufficiently effective because resynthesizing an adder on the critical path will be greatly affected by the timing result of the previously resynthesized adders on the critical path. To be more effective, we propose a technique of simultaneous synthesis of two AHAs for two connected additions. For more than two additions, we can use a repeated application of the proposed technique to the additions. First, we suggest a set of structures of addition expression that are suitable for the multiple AHA synthesis application. Then, we propose an efficient procedure of

synthesizing (simultaneous) multiple AHAs.

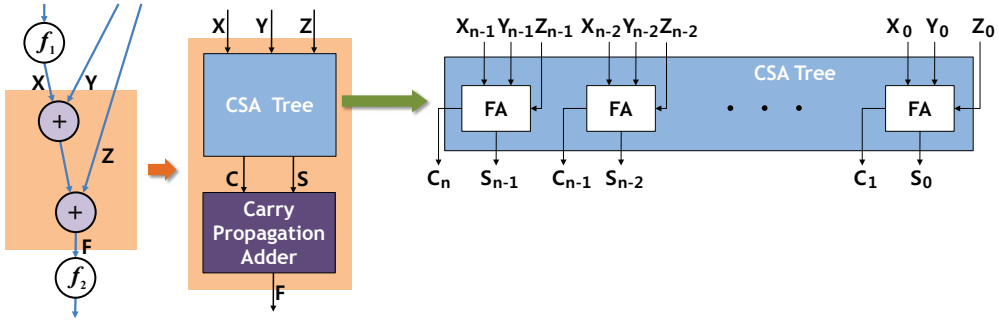
Identifying addition expressions for multiple AHAs: It is traditionally known that generating a CSA-tree (carry-save adder tree)[2] (e.g., [25,26]) or FA-tree (full-adder tree)[3] (e.g., [27,28]) followed by a final adder implementation is the most effective approach to the synthesis of fast arithmetic circuit for a cluster of addition operations. For example, Fig. 4.5(a) shows the CSA-tree consisting of one CSA and a carry propagating (final) adder for implementing $F = X + Y + Z$. Note that the final adder can be implemented with a single AHA if we want to reduce area while meeting the same timing requirement as before. Since there is no carry propagation in the CSA-tree or FA-tree, the timing of the transformed circuit will be the fastest. (See the internal structure of a CSA in Fig.4.5(a).) However, there are a number of cases where such a CSA-tree or FA-tree transformation is not adequate or causes a high design penalty if applied, but our multiple AHA synthesis scheme can be applied safely and effectively. The cases are illustrated in Figs. 4.5(b), (c), and (d) which explain *addition with multiple fanout*, *multiplexor between additions*, and *additions across design boundary*, respectively. (Note that our AHA scheme can also be applied to every addition expression with any mixture of the three cases in Figs. 4.5(b), (c), and (d).)

- Case 1 (*addition with multiple fanout*): When the output of an addition is used as input to another logic, as shown in in Fig. 4.5(b) where the outcome of the upper addition is used as input to logic f_3 as well as the lower addition. In this case, CSA-tree (or FA-tree) implementation is not possible because there is no way to produce the output of the upper addition, if operation duplication is not allowed due to area limitation.

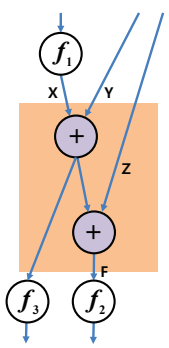
- Case 2 (*multiplexor between additions*): Similar to case 1, when there is a multi-

[2]It corresponds to the word-level compression tree for additions.

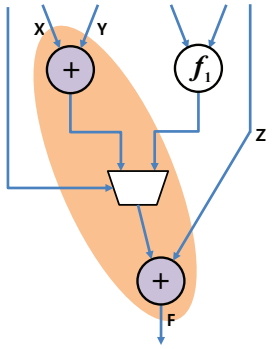
[3]It corresponds to the bit-level compression tree for additions.



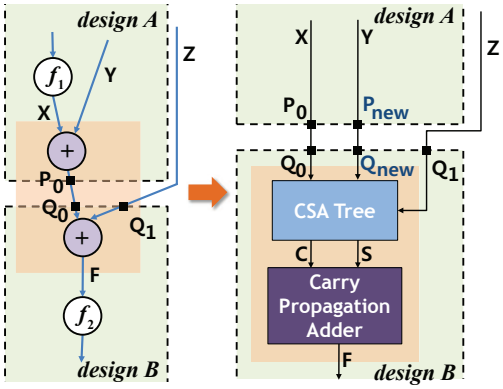
(a) The conventional CSA-tree transformation is shown on the left side. The internal structure of a CSA is shown on the right side.



(b) Case 1: addition with multiple fanout.



(c) Case 2: multiplexer between additions.



(d) Case 3: additions across design boundary

Figure 4.5: The sweet spots of CSA-tree (or FA-tree) and multiple AHA implementations. (a) Case where CSA-tree implementation is effective. (b), (c), (d) Cases where simultaneous multiple AHA implementation is effective.

plexor between two addition operations as shown in Fig. 4.5(c), the simultaneous AHA implementation can be used for optimizing timing through the additions, which otherwise an expensive operation duplication over the multiplexor shall be required to enable CSA-tree or FA-tree transformation.

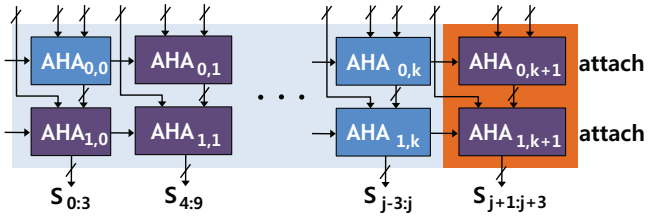
- Case 3 (*additions across design boundary*): When two additions are placed across a design boundary in the hierarchical design, applying CAS-tree (or FA-tree) transformation to the additions will change the number of input/output ports on the design boundary. This means that the meaning of some of the original ports is no longer valid. For example, the right side in Fig. 4.5(d) shows the transformed CSA-tree with final adder, in which new ports P_{new} and Q_{new} are created with the additional checking of port constraints $v(P_{new})$ = $v(Q_{new})$ in addition to the original checking of $v(P_0) = v(Q_0)$ where $v(\cdot)$ represents the data value that passes on the port. Furthermore, the original meaning of the data on ports P_0 and Q_0 is lost. To reduce the burden on checking design validation environment, it is desirable to avoid such a port change. For this reason, in a safe hierarchical design testing AHA implementation can be another alternative for timing optimization across design boundary.

Simultaneous synthesis of multiple AHAs: The idea of simultaneously synthesizing two chained additions into AHAs is considering the AHAs collectively as one *super* AHA. Thus, a super AHA consists of two AHAs where one is called *top* AHA and the other is called *bottom* AHA, and there is data connection from the top AHA to the bottom AHA. The generation procedure of a super AHA is basically the same as that of the generation of a single AHA described in the previous section. At each iteration, four possible combinations of Δd-attachment and Δd-extension are applied to the tail of the partial (super) AHA, as shown in Fig. 4.6, where the partial super AHA is composed of top partial AHA and bottom partial AHA. For example, Fig. 4.6(b)

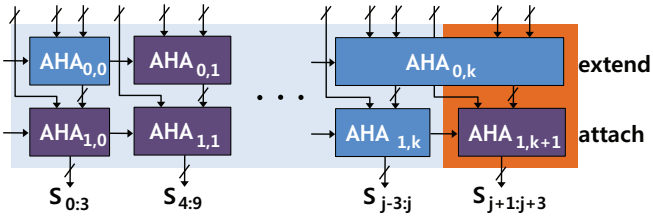
shows the expansion of super AHA by applying Δd-extension to the top AHA and Δd-attachment to the bottom AHA, and Fig. 4.6(d) shows the expansion by applying Δd-extension to both of the top and bottom AHAs. Note that the pruning based on dominance relation is also applied to the set of expanded partial supper AHAs.

In addition, at each iteration we perform the following input refining technique to optimize timing further. *Input reordering*: This technique makes use of the uneven input bit arrival times of three vector addends. This input refining technique is very effective when there is a high variation on the bit-level arrival times of inputs. For example, as shown on the left side in Fig. 4.6(b), the initial input bit segments to be added are rearranged according to the ir arrival times, and the late inputs are connected to the upper AHA, as indicated in the right side in Fig. 4.6(b). Furthermore, the two carry outs of the (partial) super AHA are also involved in the input reordering together with the three input bits, as shown the example of Fig. 4.6(c), where carry out $cout_2$ is now used as input to the upper AHA.

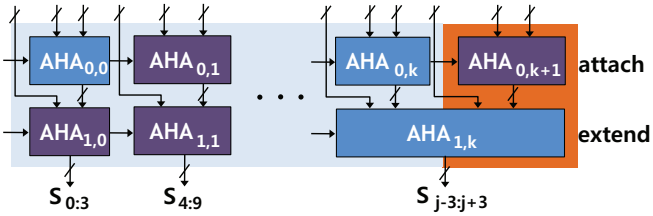
The iteration flow of synthesizing a super AHA is summarized in Fig. 4.7. Since the four combinations of Δd-attachment and Δd-extension are considered at each iteration, the total number of partial AHAs generated will be substantially large. However, due to the limited size of bit-width, which is no more than 64 in practice and the help of pruning by dominance relation and a proper control of Δd value for attachment and extension, an exhaustive exploration of design space is possible within a reasonably small run time ($\leq$ 15 minutes) as verified from our experiments.



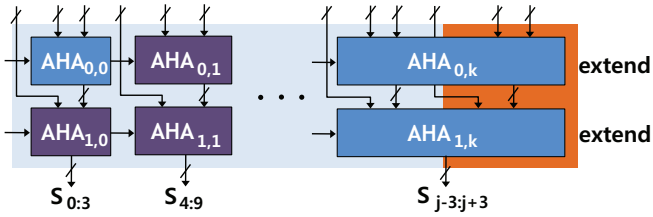
(a) Δd-attach for both top and bottom AHAs.



(b) Δd-extend for top AHA and Δd-attach for bottom AHA.



(c) Δd-attach for top AHA and Δd-extend for bottom AHA.



(d) Δd-extend for both top and bottom AHAs.

Figure 4.6: Four possible combinations of extending a (partial) super AHA. (a) Combination 1: Δd-attach for both top and bottom AHAs. (b) Combination 2: Δd-extend for top AHA and Δd-attach for bottom AHA. (c) Combination 3: Δd-attach for top AHA and Δd-extend for bottom AHA. (d) Combination 4: Δd-extend for both top and bottom AHAs.

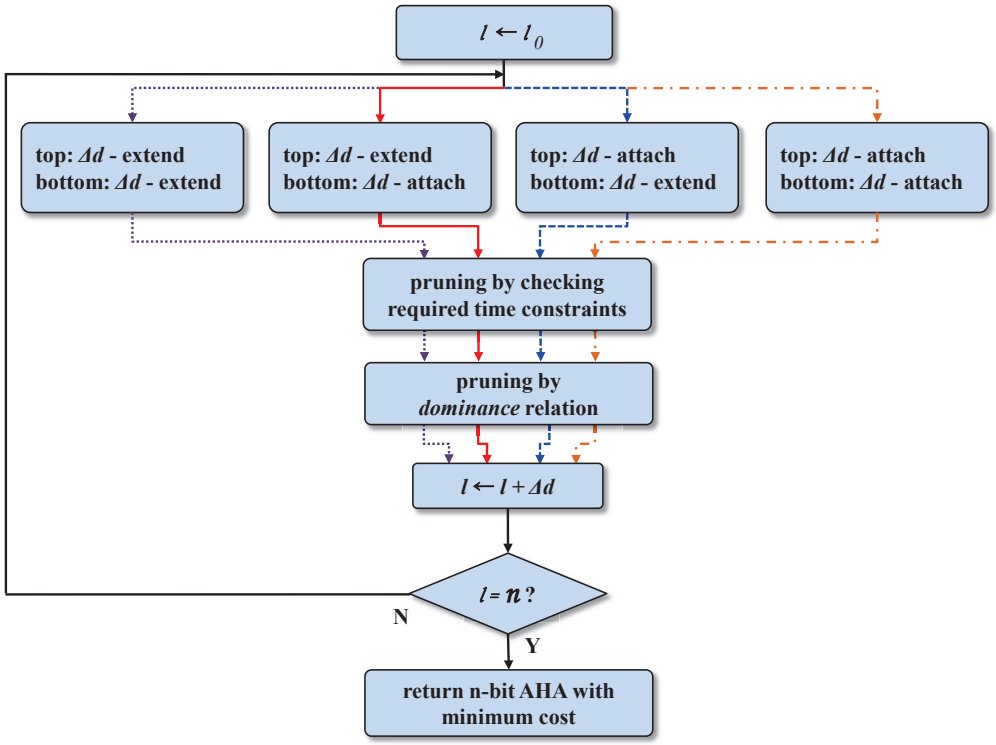


Figure 4.7: The iteration flow of synthesizing an n-bit super AHA.

Chapter 5

Experimental Results

We implemented our proposed AHA generation scheme with TCL (Tool Command Language) script to link to Synopsys synthesis tools on a linux server with octa-core 2.0GHz Intel zeon processor and 6GB RAM. We used Synopsys Design Compiler and Prime Time with TSMC 40 nm library for the logic optimization and timing estimation, respectively. We evaluate our proposed synthesis scheme on a set of typical arithmetic expressions with additions that are commonly appeared in DSP applications. We compare the results produced by our scheme with that produced by the adder optimization scheme proposed in [2] as well as several pure adders under the following settings of AHA generation mode and input and output timings: (1) generating a single adder with non-uniform input arrival times; (2) generating a single adder considering non-uniform output required time constraint; (3) generating a single adder considering both non-uniform input arrival and output required times; (4) generating multiple (super) adders; In addition, (5) we compare our results with that produced by the commercial synthesis tool; (6) we provide a set of comparisons of results to show how much our AHA scheme is effective when more than one differently cell sized implementation for each pure adder scheme are used in the AHA synthesis. Finally,

(7) we also provide data that shows how well our AHA scheme controls the synthesis quality and running time.

5.1 Generating a Single Adder with Non-uniform Input Arrival Times

The previous works in [1, 2, 21, 22] also proposed schemes of synthesizing single hybrid adders considering uneven input arrival times. Unfortunately, however, the schemes are a sort of 'specialized' hybrid adder schemes in that they utilize (fixed) patterns or trend of arrival times of input bits in designing a hybrid adder of the final addition on the partial products reduction tree (PPRT) in the multiplier design; they observed that as input bits are close to the least significant bit (LSB) the bit operands (or signals) tend to arrive early, validating the use of a slow adder for a segment of input operands near LSB, and as they are close to the middle bit, the signals arrive late, validating the use of a fast adder for a segment of input operands near the middle bit or the most significant bit (MSB). On the other hand, our hybrid adder scheme is 'general' in that it accepts any arbitrary arrival times of input bits. That is, our scheme explores, for a particular segment of input bits, all possible implementations of pure adder being used as adding the input segment. We compare our AHA synthesis results with that produced by the most recent hybrid adder synthesis work in [2]. We also compare our results with the several pure adder implementations that meet the output required timing constraint.

Table 5.1 shows the comparison of the performance of our AHA synthesis with that of pure adder implementations: RCA (ripple carry adder), CLA (carry look-ahead adder), BKA (brent-kung adder), and CSLA (carry-select adder). The first column shows the tested arithmetic expressions where the additions marked with red color are the target additions and the second column shows the implementation area of AHA for

Table 5.1: Comparison of AHA scheme with pure adder schemes under uneven input arrival times.

Expression	Timing (ns)					Area (μm^2)	Area red. (req. timing)
	RCA	CLA	BKA	CSLA	AHA	AHA	
(A$+_1$B)+C	**1.58**	1.60	1.65	1.62	1.58	141.1	0% (1.58)
(A$+_2$B)+C	1.60	**0.82**	0.86	0.92	0.81	240.6	21% (0.82)
(A$+_3$B)+C	1.60	**0.88**	0.95	0.95	0.85	236.6	22% (0.88)
(A$+_4$B)+C	1.62	**0.99**	1.09	1.09	0.96	224.0	26% (0.99)
(A$*_L$B)+C	2.11	**1.63**	1.68	1.67	1.61	207.4	32% (1.63)
(A/B)+C	30.25	**29.20**	29.31	29.50	29.16	291.9	11% (29.20)
(A$\ll$B)+C	1.77	**0.78**	0.86	1.02	0.70	268.8	11% (0.78)
A^2+B	**1.54**	1.55	1.61	1.58	1.54	141.1	0% (1.54)
(A-B)+C	1.61	**0.83**	0.95	0.93	0.82	240.6	21% (0.83)
Average							16%

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits of 64-bit output.

* AHA: { RCA, CLA, CSKA }

* Area (32-bit addition): RCA:141.1, CLA:302.9, CSKA:220.1, BKA:395.5, CSLA:479.8.

the target additions when AHA scheme uses pure adder set $\mathcal{L} = \{RCA, CLA, CSKA\}$. The third thru seventh columns show the latest output times of the corresponding adder implementations. For the AHA synthesis results in the second and seventh columns, we assume a uniform required output timing constraint (shown in the parentheses of the last column) that is set to the fastest output time of the pure adders, as indicated by the blue colored numbers in the third thru sixth columns. (The area of pure adder implementations is specified at the bottom of the table.)

Note that for some designs, which is usually a monotonically increasing input arrival times, a pure RCA implementation will suffice to produce a minimal area while meeting the required timing. (See the results for test cases (A$+_1$B)+C and A^2+B in Table 5.1.).

The last column summarizes the area reduction by our AHA implementation over the least-area pure adder implementation that meets the same output timing constraint. Overall, under uneven input arrival times our scheme is able to adaptably generate hybrid adders with 16% reduced area.

Table 5.2 shows the comparison of our AHA synthesis results with that of the scheme in [2] which uses RCA, CLA, and CSKA adders for low-, middle-, and upper-bit input operands, respectively. We used $\mathcal{L}$ = {RCA, CLA, CSKA} for the AHA implementation. We set the required output timing, specified in the second column, to the output time of the hybrid adder produced by the scheme in [2]. From the table, we can see that our AHA scheme reduces the area by 27% under the same output timing constraint, revealing that our scheme performs very well in optimizing area over the conventional hybrid adder design scheme as well as pure adder schemes without any timing increase. The fifth and sixth columns show how the pure adders are combined in the hybrid adder implementation, together with their bit-width information.

Note that the scheme in [2] does not work well on test cases (A$+_1$B)+C and A^2+B. This is because the input arrival times of the two test cases follow a monotonically

Table 5.2: Comparison of AHA scheme with [2] under uneven input arrival times.

Expression	Req. output time	Adder composition (bit-widths)		Area (μm^2)		Area red.
		DAS	AHA	DAS	AHA	
(A+$_1$B)+C	1.59	(0 20 12)	[r] (32)	272.4	141.1	48%
(A+$_2$B)+C	0.91	(0 26 6)	[r||c||c||c||c||r] (9 4 4 4 4 4 3)	291.9	224.0	23%
(A+$_3$B)+C	1.03	(0 26 6)	[r||c||c||c||c||r] (12 4 4 4 5 3)	291.9	215.7	26%
(A+$_4$B)+C	1.16	(0 26 6)	[r||c||c||c||c||r] (16 4 4 4 3 1)	291.9	203.4	30%
(A$*_L$B)+C	1.65	(3 23 6)	[r||c||c||c||c||r] (16 4 4 4 3 1)	276.6	203.4	26%
(A/B)+C	29.39	(0 26 6)	[r||c||c||c||r] (1 16 4 3 8)	291.9	248.2	15%
(A$\ll$B)+C	0.89	(0 26 6)	[r||c||c||c||r] (2 4 15 4 7)	291.9	248.2	15%
A^2+B	1.54	(0 20 12)	[r] (32)	272.4	141.1	48%
(A-B)+C	1.01	(0 26 6)	[r||c||c||c||c||r] (12 4 4 4 4 3 1)	291.9	220.0	25%
Average						**27%**

* DAS indicates the implementation scheme in [2], with [RCA||CLA|| CSKA]

* AHA: { RCA, CLA, CSKA }

* 'r' indicates RCA; 'c' indicates CLA.

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits 64-bit output.

increasing curve, but the scheme in [2] mainly targets U-shape pattern of input arrival times.

5.2 Generating a Single Adder Considering Non-uniform Output Required Time Constraint

To facilitate the optimization of adders under uneven output required times, we change the target addition to be optimized in the expressions in Table 5.1, as shown in the first column in Table 5.3. Except the change of target operation, all other conventions in Table 5.3 are exactly the same as that in Table 5.1. As indicated by the last column in Table 5.3, our AHA synthesis is able to use 20% less area compared to the pure adder of least-area while meeting the output timing constraint. In addition, Table 5.4 shows the comparison of our synthesis results with that of the scheme in [2]. We can see that our AHA finds more fine-grained combination of pure adders to adapt the uneven output required timing while the work in [2] dose not, which is even worse than the pure implementations in Table 5.3 in some test cases.

5.3 Generating a Single Adder Considering Both Non-uniform Input Arrival and Output Required Times

Unlike the tested data in Tables 5.1 through 5.4, we update the expressions, so that each target addition has uneven input arrival times as well as uneven required output times. Tables 5.5 and 5.6 show the comparisons of the results by AHA scheme with pure adder schemes and the scheme in [2], respectively. It is seen that the area saving by AHA is consistent in the range of $9\% \sim 31\%$, which implies that AHA can be useful in customizing adders with both uneven bit-level input arrival times and output required times.

Table 5.3: Comparison of AHA scheme with pure adder schemes under the constraint of uneven required output times.

Expression	Timing (ns)					Area (μm^2)	Area red. (req. timing)
	RCA	CLA	BKA	CSLA	AHA	AHA	
(A+B)$+_1$C	**1.58**	1.60	1.60	1.62	1.58	141.1	0% (1.58)
(A+B)$+_2$C	1.60	**0.82**	0.88	0.99	0.82	248.2	18% (0.82)
(A+B)$+_3$C	1.65	**0.86**	0.95	1.09	0.86	269.0	11% (0.86)
(A+B)$+_4$C	1.62	**0.92**	0.95	1.09	0.91	231.6	24% (0.92)
(A+B)$*_L$C	2.75	**1.76**	1.84	2.03	1.72	268.8	11% (1.76)
(A+B)/C	29.44	28.82	**28.75**	28.75	28.75	224.0	43% (28.75)
(A+B)$\ll$C	1.78	**0.78**	0.87	1.02	0.71	268.8	11% (0.78)
(A+B)^2	3.61	**3.20**	3.22	3.26	3.19	196.7	35% (3.20)
(A+B)-C	1.61	**0.92**	0.94	1.00	0.91	224.0	26% (0.92)
Average							20%

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits of 64-bit output.

* AHA: { RCA, CLA, CSKA }

* Area (32-bit addition): RCA:141.1, CLA:302.9, CSKA:220.1, BKA:395.5, CSLA:479.8.

Table 5.4: Comparison of AHA scheme with [2] under the constraint of uneven required output times.

Expression	Req. output time	Adder composition (bit-widths)		Area (μm^2)		Area red.
		DAS	AHA	DAS	AHA	
(A+B)$+_1$C	1.60	(0 26 6)	[r] (32)	291.9	141.1	52%
(A+B)$+_2$C	0.82		$[r\|\|c\|\|c\|\|c\|\|r]$ (2 4 4 15 7)		248.2	15%
(A+B)$+_3$C	0.86		$[r\|\|c\|\|c\|\|r]$ (2 4 15 11)		231.6	21%
(A+B)$+_4$C	0.92		$[r\|\|c\|\|r]$ (2 4 26)		157.7	46%
(A+B)$*_L$C	1.86		$[r\|\|c\|\|c\|\|c\|\|r]$ (2 4 16 4 6)		252.3	14%
(A+B)/C	28.75		$[r\|\|c\|\|c\|\|c\|\|c\|\|c\|\|r]$ (2 4 4 4 4 4 10)		224.0	23%
(A+B)$\ll$C	0.88		$[r\|\|c\|\|c\|\|c\|\|r]$ (2 4 16 4 6)		252.3	14%
$(A+B)^2$	3.20		$[r\|\|c\|\|c\|\|r]$ (2 4 8 18)		196.7	33%
(A+B)-C	0.92		$[r\|\|c\|\|c\|\|c\|\|c\|\|c\|\|r]$ (2 4 4 4 4 4 10)		224.0	23%
Average						**27%**

* DAS indicates the implementation scheme in [2], with [RCA||CLA|| CSKA]

* AHA: { RCA, CLA, CSKA }

* 'r' indicates RCA; 'c' indicates CLA.

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits 64-bit output.

Table 5.5: Comparison of AHA scheme with pure adder schemes under both uneven input arrival and required output times.

Expression	Timing (ns)					Area (μm^2)	Area red. (req. timing)
	RCA	CLA	BKA	CSLA	AHA	AHA	
((A$*_L$B)+C)-D	2.22	**1.85**	1.88	1.85	1.84	190.9	37% (1.85)
((A-B)+C)$*_L$D	2.87	**2.09**	2.20	2.22	2.07	253.7	16% (2.09)
((A/B)+C)$\ll$D	30.53	**29.47**	29.59	29.77	29.44	268.8	11% (29.47)
((A$\ll$B)+C)/D	29.64	**28.75**	28.75	28.89	28.75	264.8	13% (28.75)
((A$\ll$B)+C)-D	1.88	**1.19**	1.21	1.28	1.19	224.0	26% (1.19)
((A-B)+C)$\ll$D	1.89	**1.11**	1.22	1.21	1.07	240.6	21% (1.11)
Average							21%

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits of 64-bit output.

* AHA: { RCA, CLA, CSKA }

* Area (32-bit addition): RCA:141.1, CLA:302.9, CSKA:220.1, BKA:395.5, CSLA:479.8.

Table 5.6: Comparison of AHA scheme with [2] under both uneven input arrival and required output times.

Expression	Req. output time	Adder composition (bit-widths)		Area (μm^2)		Area red.
		DAS	AHA	DAS	AHA	
((A$*_L$B)+C)-D	1.91	(3 23 6)	$[r\|\|c\|\|c\|\|c\|\|r]$ (16 4 4 3 5)	272.4	186.8	31%
((A-B)+C)$*_L$D	2.24	(0 26 6)	$[r\|\|c\|\|c\|\|c\|\|c\|\|c\|\|r]$ (9 4 4 4 4 4 3)	291.9	224.0	23%
((A/B)+C)$\ll$D	29.51	(0 26 6)	$[r\|\|c\|\|c\|\|c\|\|r\|\|c\|\|r]$ (1 16 4 4 1 4 2)	291.9	268.8	8%
((A$\ll$B)+C)/D	28.75	(0 26 6)	$[r\|\|c\|\|c\|\|c\|\|c\|\|r]$ (2 4 16 4 3 3)	291.9	264.8	9%
((A$\ll$B)+C)-D	1.19	(0 26 6)	$[r\|\|c\|\|c\|\|c\|\|c\|\|c\|\|r]$ (2 4 4 4 4 4 10)	291.9	224.0	23%
((A-B)+C)$\ll$D	1.24	(0 26 6)	$[r\|\|c\|\|c\|\|c\|\|c\|\|c\|\|r]$ (9 4 4 4 4 4 3)	291.9	224.0	23%
Average						**20%**

* DAS indicates the implementation scheme in [2], with [RCA||CLA|| CSKA]

* AHA: { RCA, CLA, CSKA }

* 'r' indicates RCA; 'c' indicates CLA.

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits 64-bit output.

5.4 Generating Multiple (Super) Adders

If two or more addition operations are chained, implementing the adders with FA-tree or CSA-tree might reduce the overall delay. However, as stated in subsection 4.2 there are cases where FA-tree or CSA-tree implementation is not appropriate. We assume those cases in the experiments. We test our AHA scheme with the implementation of RCA-RCA (two chained RCAs), CLA-CLA (two chained CLA), BKA-BKA (two chained BKA) and CSLA-CSLA (two chained CSLA). Table 5.7 shows the results of AHA and those combinations of pure adders. The results indicate that the multiple AHA synthesis outperforms the pure adder implementations, but in some design there is no improvement by AHA at all. Table 5.8 shows the comparison of AHA synthesis results with that in [2]. The area improvement is $18\% \sim 31\%$, which clearly indicates that our proposed multiple AHA synthesis scheme works well.

5.5 Comparison with Commercial Synthesis Tool

It might be interesting to see how well a commercial synthesis tool selects pure adders to optimize area under a given timing constraint. We compare our AHA synthesis results with that produced by Synopsys Design Compiler [3]. Tables 5.9 and 5.10 show the comparison of results for various arithmetic expressions. In this experiments the output timing constraints are set to the earliest times used in the experiments in Tables 5.1, 5.3, 5.5 and 5.7. Our design methodology reoptimizes, in the second pass of synthesis in the design flow shown in Fig. 4.1, the entire arithmetic logic by using Design Compiler while retaining the implementation structure of AHA we obtained. We have observed that Design Compiler initially uses CLA in most of designs to meet the tight timing constraint and then gradually reduces area at the expense of increasing timing [3]. For some test cases, Design Compiler uses smaller implementation area, but for most of test cases our hybrid design scheme is able to find more efficient

Table 5.7: Comparison of AHA scheme with pure adders for synthesizing two chained additions.

Expression	Timing (ns)					Area (μm^2)	Area red. (req. timing)
	RCA	CLA	BKA	CSLA	AHA	AHA	
(((A$*_L$B) +C)+D)-E	2.31	**2.07**	2.17	2.15	2.07	381.7	37% (2.07)
(((A-B) +C)+D)$*_L$E	2.96	**2.35**	2.49	2.56	2.35	464.6	23% (2.35)
(((A/B) +C)+D)$\ll$E	30.61	**29.80**	29.94	30.12	29.80	514.4	15% (29.80)
(((A$\ll$B) +C)+D)/E	29.73	**28.97**	29.96	29.23	28.97	605.8	0% (28.97)
(((A$\ll$B) +C)+D)-E	1.97	**1.44**	1.47	1.59	1.43	443.1	27% (1.44)
(((A-B) +C)+D)$\ll$E	1.98	**1.37**	1.52	1.55	1.34	464.6	23% (1.37)
Average							21%

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits of 64-bit output.

* AHA: { RCA, CLA, CSKA }

* Area (32-bit addition): RCA:141.1, CLA:302.9, CSKA:220.1, BKA:395.5, CSLA:479.8.

Table 5.8: Comparison of AHA scheme with [2] for two chained additions.

Expression	Req. output time	Adder composition (bit-widths)		Area (μm^2)		Area red.
		DAS	AHA	DAS	AHA	
(((A$*_L$B) +C)+D)-E	2.09	(3 23 6) (3 23 6)	[r||c||c||c||r||r] (13 4 4 4 4 3) [r||r||c||c||c||r] (13 4 4 4 4 3)	553.2	381.7	31%
(((A-B) +C)+D)$*_L$E	2.52	(0 26 6) (0 26 6)	[r||c||c||c||c||r||r] (9 4 4 4 4 4 3) [r||r||c||c||c||c||r] (9 4 4 4 4 4 3)	583.9	414.9	29%
(((A/B) +C)+D)$\ll$E	29.92	(0 26 6) (0 26 6)	[r||c||c||c||c||c||c||r||r] (3 4 4 4 4 4 4 4 4 1) [r||r||c||c||c||c||c||c||r] (3 4 4 4 4 4 4 4 4 1)	583.9	481.2	18%
(((A$\ll$B) +C)+D)/E	29.09	(0 26 6) (0 26 6)	[r||c||c||c||c||c||c||r||r] (3 4 4 4 4 4 4 4 4 1) [r||r||c||c||c||c||c||c||r] (3 4 4 4 4 4 4 4 4 1)	583.9	481.2	18%
(((A$\ll$B) +C)+D)-E	1.44	(0 26 6) (0 26 6)	[r||c||c||c||c||r||r] (3 4 6 4 4 4 7) [r||r||c||c||c||c||r] (3 4 6 4 4 4 7)	583.9	443.1	24%
(((A-B) +C)+D)$\ll$E	1.54	(0 26 6) (0 26 6)	[r||c||c||c||c||r||r] (9 4 4 4 4 4 3) [r||r||c||c||c||c||r] (9 4 4 4 4 4 3)	583.9	414.9	29%
Average						**25%**

* DAS indicates the implementation scheme in [2], with [RCA||CLA|| CSKA]

* AHA: { RCA, CLA, CSKA }

* 'r' indicates RCA; 'c' indicates CLA.

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits 64-bit output.

implementations.

Table 5.9: Comparison of AHA scheme with Synopsys Design Compiler [3] for simple arithmetic expressions.

Expression	Area (μm^2)		Area red.	Expression	Area (μm^2)		Area red.
	AHA	DC			AHA	DC	
(A+$_1$B)+C	141.1	261.2	46%	(A+B)+$_1$C	141.1	263.7	46%
(A+$_2$B)+C	216.6	260.9	17%	(A+B)+$_2$C	225.4	260.9	14%
(A+$_3$B)+C	214.5	261.8	18%	(A+B)+$_3$C	269.0	260.9	-3%
(A+$_4$B)+C	224.0	260.9	14%	(A+B)+$_4$C	231.6	260.9	11%
(A$*_L$B)+C	199.0	260.9	24%	(A+B)$*_L$C	236.4	260.9	9%
(A/B)+C	236.4	260.9	9%	(A+B)/C	268.8	260.9	-3%
(A$\ll$B)+C	236.4	260.9	9%	(A+B)$\ll$C	236.4	260.9	9%
A^2+B	141.1	260.9	46%	(A+B)^2	196.7	260.9	25%
(A-B)+C	216.6	260.9	17%	(A+B)-C	224.0	260.9	14%
Average			**22%**	Average			**14%**

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits of 64-bit output.

* DC: Synopsys Design Compiler®

5.6 AHA Synthesis Combined with Cell Sizing

To consider the effect of cell sizing on the AHA synthesis, besides the area-efficient implementations RCA, CLA, BKA, and CSLA used previously, we have produced

Table 5.10: Comparison of AHA scheme with Synopsys Design Compiler [3] for complex arithmetic expressions.

Expression	Area (μm^2)		Area red.	Expression	Area (μm^2)		Area red.
	AHA	DC			AHA	DC	
((A$*_L$B)+C)-D	190.9	260.9	27%	(((A$*_L$B)+C)+D)-E	380.3	521.8	27%
((A-B)+C)$*_L$D	253.7	260.9	3%	(((A-B)+C)+D)$*_L$E	464.6	523.9	11%
((A/B)+C)$\ll$D	236.4	260.9	9%	(((A/B)+C)+D)$\ll$E	450.9	522.3	14%
((A$\ll$B)+C)/D	234.3	260.9	10%	(((A$\ll$B)+C)+D)/E	605.8	527.8	-15%
((A$\ll$B)+C)-D	224.0	260.9	14%	(((A$\ll$B)+C)+D)-E	443.1	521.8	15%
((A-B)+C)$\ll$D	240.6	260.9	8%	(((A-B)+C)+D)$\ll$E	424.4	521.8	19%
Average			**12%**	Average			**12%**

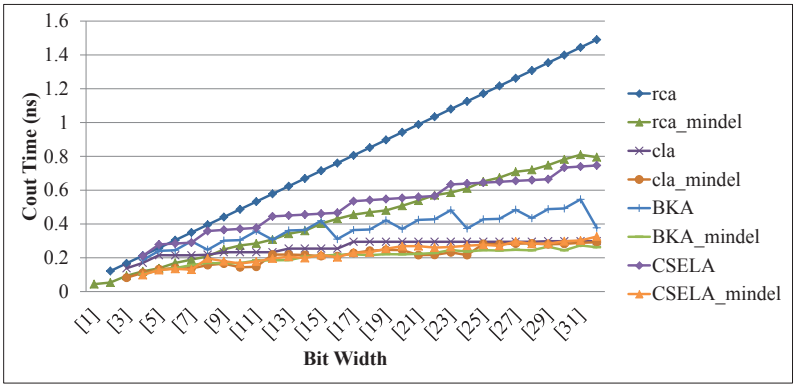
* DC: Synopsys Design Compiler®

four additional fast implementations labeled RCA$_m$, CLA$_m$, BKA$_m$, and CSLA$_m$ by applying Synopsys Design Compiler [3] with `set_max_delay 0 all_outputs()` command. Carry out time, maximum sum out time and area of the adders with respect to bit width of each adders are depicted 5.1. We set the pure adder library $\mathcal{L}$ = {RCA, RCA$_m$, CLA, CLA$_m$, BKA$_m$} in our AHA synthesis. The comparisons of results are shown in Tables 5.11, 5.12, and 5.13. As expected, for some test cases, considering the cell sizing effect on the AHA synthesis is very effective in reducing area while meeting timing, but for some test cases, there is no improvement. This is the case where due to the tight timing constraint (blue numbers in tables) our scheme was not able to produce an AHA composed of multiple pure adders and thus used, as the AHA, only the single (i.e., the whole n-bit) fast pure adder implementation that had been created by cell sizing.

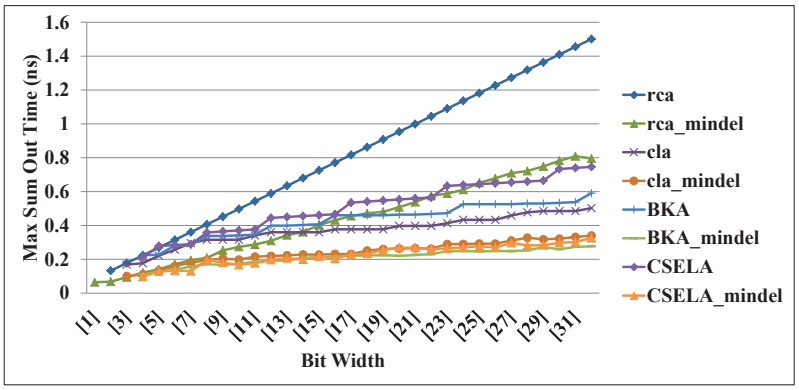
5.7 Synthesis for power minimization

AHA synthesis can be used for power consumption minimization while satisfying given timing constraints. Table 5.14 shows optimization results targeting minimizing power consumption while satisfying given timing constraints. The first column shows the tested arithmetic expression and addition marked with red color are the addition to be optimized. Timing constraints are given by using the addition with pure adders {RCA, RCA$_m$, CLA, CLA$_m$, BKA, BKA$_m$, CSLA, CSLA$_m$ }, then selecting the adders which gives fastest timing. The last column represents power reduction compared to the pure adders which gives best timing.

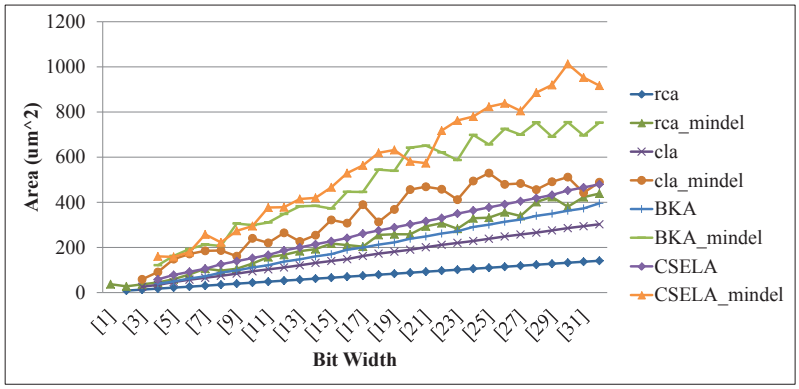
Fig. 5.2 shows area and power consumption relationship of the optimal solution which targets minimizing power consumption. As shown, the adder with larger area consumes more power and this have almost linear relationship.



(a) Carry output time



(b) Maximum sum output time



(c) Area

Figure 5.1: Cout time, maximum sum out time and Area of the adders with respect to bit width

Table 5.11: Comparison of AHA scheme using multiply cell sized pure adder implementations under uneven input arrival times.

Exp.	Timing (ns)								Area (μm^2)	Area red.
	RCA	RCA$_m$	CLA	CLA$_m$	BKA	BKA$_m$	CSLA	CSLA$_m$	AHA	
(A+$_1$B)+C	**1.58**	1.59	1.60	1.69	1.65	1.76	1.62	1.65	141.1	0%
(A+$_2$B)+C	1.60	1.01	0.82	**0.72**	0.86	0.76	0.92	0.75	489.2	0%
(A+$_3$B)+C	1.60	1.02	0.88	**0.78**	0.95	0.85	0.95	0.84	389.5	20%
(A+$_4$B)+C	1.62	1.06	0.99	**0.96**	1.09	1.01	1.09	0.99	281.0	43%
(A$*_L$B)+C	2.11	1.66	1.63	**1.57**	1.68	1.61	1.67	1.59	489.2	0%
(A/B)+C	30.25	29.54	29.20	29.08	29.31	**29.02**	29.50	29.07	753.1	0%
(A$\ll$B)+C	1.77	1.07	0.78	0.62	0.86	**0.55**	1.02	0.60	753.1	0%
A^2+B	1.54	**1.54**	1.55	1.64	1.61	1.71	1.58	1.60	186.8	57%
(A-B)+C	1.61	1.00	0.83	**0.82**	0.95	0.88	0.93	0.84	245.4	50%
Average										**16%**

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits of 64-bit output.

* Area (32-bit addition): RCA:141.1, RCA$_m$:438.9, CLA:302.9, CLA$_m$:489.2, BKA:395.5, BKA$_m$:753.1, CSLA:479.8, CSLA$_m$:917.1.

Table 5.12: Comparison of AHA scheme using multiply cell sized pure adder implementations under the constraint of uneven required output times.

Exp.	Timing (ns)								Area (μm^2)	Area red.
	RCA	RCA$_m$	CLA	CLA$_m$	BKA	BKA$_m$	CSLA	CSLA$_m$	AHA	
(A+B)$+_1$C	1.58	1.61	1.60	1.60	1.60	**1.57**	1.62	1.64	172.5	77%
(A+B)$+_2$C	1.60	1.06	0.82	0.77	0.88	**0.73**	0.99	0.78	404.8	46%
(A+B)$+_3$C	1.65	1.10	0.86	0.81	0.95	**0.78**	1.09	0.83	753.1	0%
(A+B)$+_4$C	1.62	1.09	0.92	0.87	0.95	**0.85**	1.09	0.92	753.1	0%
(A+B)$*_L$C	2.75	2.06	1.76	1.62	1.84	**1.60**	2.03	1.61	264.8	65%
(A+B)/C	29.44	28.82	28.82	28.82	28.75	28.82	**28.75**	28.82	224.0	76%
(A+B)$\ll$C	1.78	1.07	0.78	0.62	0.87	**0.55**	1.02	0.60	753.1	0%
(A+B)^2	3.61	3.21	3.20	3.15	3.22	**3.13**	3.26	3.18	248.9	67%
(A+B)-C	1.61	1.08	0.92	0.87	0.94	**0.82**	1.00	0.87	753.1	0%
Average										**37%**

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits of 64-bit output.

* Area (32-bit addition): RCA:141.1, RCA$_m$:438.9, CLA:302.9, CLA$_m$:489.2, BKA:395.5, BKA$_m$:753.1, CSLA:479.8, CSLA$_m$:917.1.

Table 5.13: Comparison of AHA scheme using multiply cell sized pure adder implementations under both uneven input arrival and required output times.

Exp.	Timing (ns)								Area (μm^2)	Area red.
	RCA	RCA$_m$	CLA	CLA$_m$	BKA	BKA$_m$	CSLA	CSLA$_m$	AHA	
((A$*_L$B)+C)-D	2.22	1.95	1.85	**1.82**	1.88	1.83	1.85	1.84	190.9	0%
((A-B)+C)$*_L$D	2.87	2.27	2.09	**2.07**	2.20	2.11	2.22	2.13	253.7	48%
((A/B)+C)$\ll$D	30.53	29.82	29.47	29.36	29.59	**29.30**	29.77	29.35	268.8	60%
((A$\ll$B)+C)/D	29.64	29.01	28.75	28.82	**28.75**	28.82	28.89	28.82	264.8	34%
((A$\ll$B)+C)-D	1.88	1.36	1.19	1.14	1.21	**1.10**	1.28	1.15	224.0	0%
((A-B)+C)$\ll$D	1.89	1.28	1.11	**1.07**	1.22	1.10	1.21	1.11	240.6	51%
Average										**32%**

* Implementation: $+_1$:RCA, $+_2$:CLA, $+_3$:BKA, $+_4$:CSLA; $*_L$: low order 32 bits of 64-bit output.

* Area (32-bit addition): RCA:141.1, RCA$_m$:438.9, CLA:302.9, CLA$_m$:489.2, BKA:395.5, BKA$_m$:753.1, CSLA:479.8, CSLA$_m$:917.1.

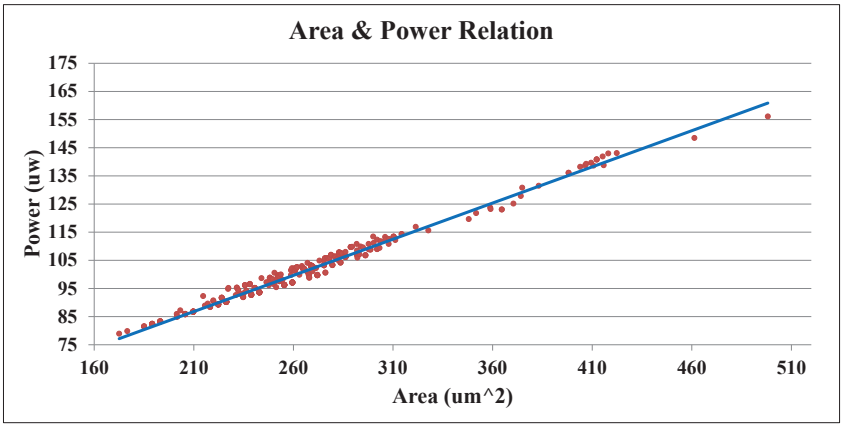


Figure 5.2: Area and power relation of the optimal solution targeting power minimization.

Table 5.14: Comparison of AHA scheme with [2] under both uneven input arrival and required output times.

Expression	Adder composition	Power (μw^2)	Power red.
((A-B)+C)$*_L$D	$[r\|\|c\|\|c\|\|c\|\|c\|\|c]$ (7 4 4 4 10 3)	100.0	41%
((A/B)+C)$\ll$D	$[r\|\|c\|\|c\|\|r]$ (1 26 4 1)	110.8	51%
((A$\ll$B)+C)/D	$[r\|\|c\|\|c\|\|c\|\|r\|\|c\|\|r]$ (2 4 4 16 2 3 1)	102.2	27%
((A-B)+C)$\ll$D	$[r\|\|c\|\|c\|\|c\|\|c\|\|c\|\|c\|\|r]$ (7 4 4 4 4 4 4 1)	95.1	44%
Average			**27%**

* AHA: { $\mathcal{L}$ = {RCA, RCA$_m$, CLA, CLA$_m$, BKA, BKA$_m$, CSLA, CSLA$_m$ } }

* 'r' indicates RCA; 'c' indicates CLA.

* Power (32-bit addition): RCA:74.4, RCA$_m$:160.7, CLA:112.5, CLA$_m$:168.3, BKA:140.8, BKA$_m$:223.9420, CSLA:180.0, CSLA$_m$:327.0.

5.8 Design Quality and Running time.

• *Design quality*: Fig. 5.3 shows the implementation area of AHAs with respect to various pure adder library $\mathcal{L}$ that AHA scheme uses. The tested 16 circuits are those arithmetic expressions used in the previous tables. It is shown that $\mathcal{L} = \{RCA, CLA\}$, $\{RCA, CLA, CSKA\}$ and $\{RCA, CLA, CSLA\}$ produce the most efficient adders. Consequently, we can trim the library to $\mathcal{L} = \{RCA, CLA\}$ to reduce the ANA synthesis complexity while creating area-efficient adders under timing constraint. However, depending on the technology and pure adder implementation details to be used, the best library might be different.

• *Running time*: Fig. 5.3 summarizes the run time of AHA synthesis with respect to various Δd values, bit-width of addends, and target library $\mathcal{L}$. We measured average run time of AHA synthesis for the expressions in Tables 5.1, 5.3, 5.5, and 5.7. The slowest run time is about 15 minutes, which happens when $\Delta d = 1$ and $\mathcal{L} = \{RCA, CLA, CSKA\}$ for 32-bit addition, as shown in Fig. 5.3(d). If the size of $\mathcal{L}$ is over 4 or the bit-width of addends exceeds 32, the running time can be controlled by increasing the value of parameter Δd.

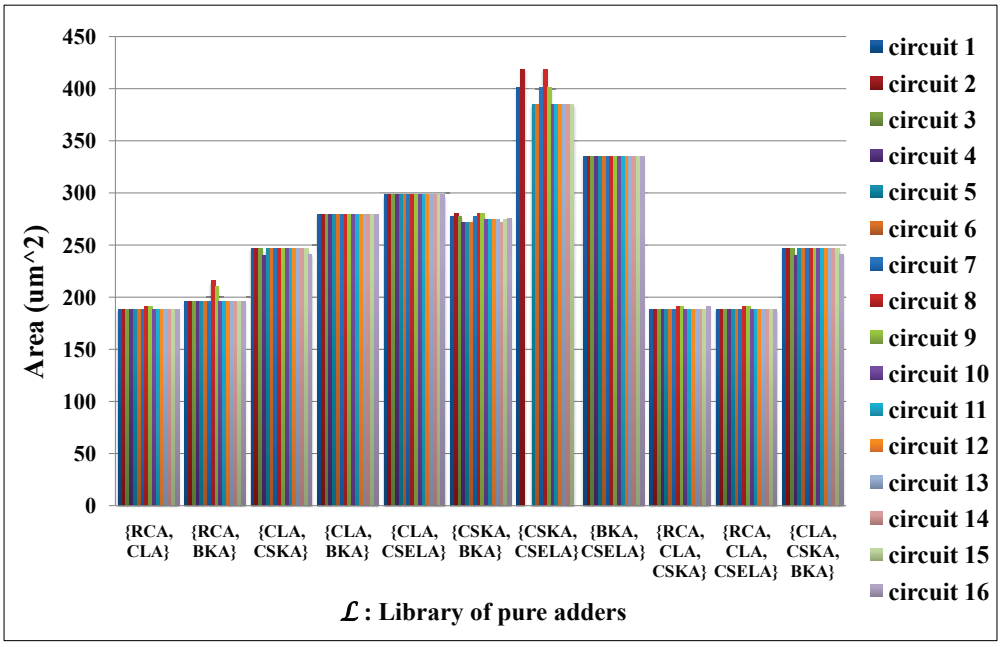
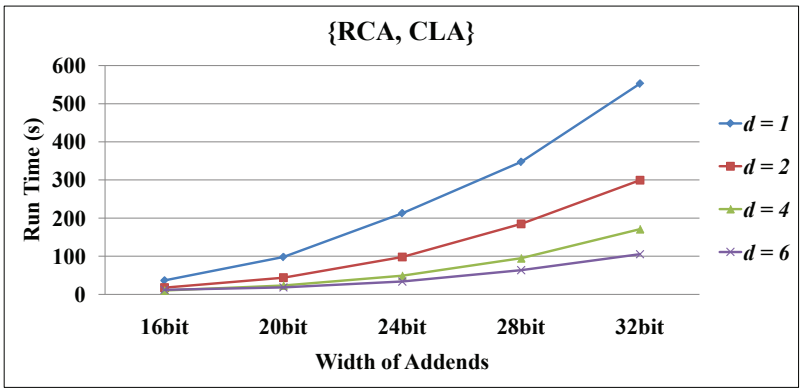
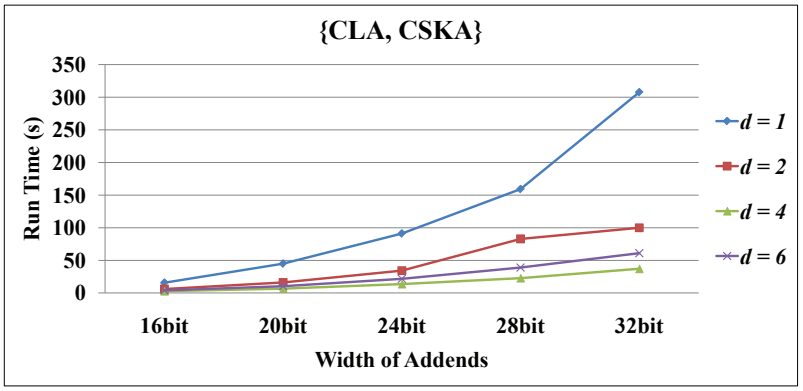


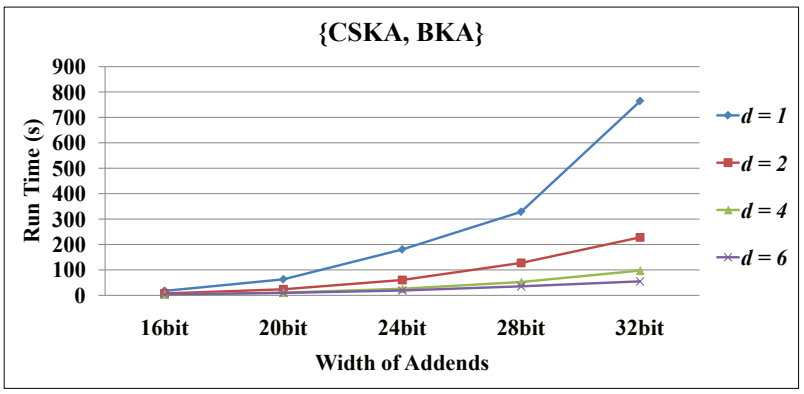
Figure 5.3: Comparison of implementation area produced by using various libraries of pure adders.



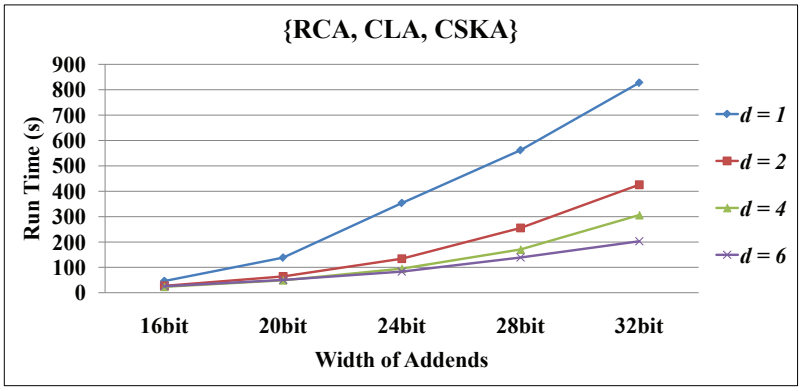
(a) $\mathcal{L} = \{\mathcal{RCA}, \mathcal{CLA}\}$



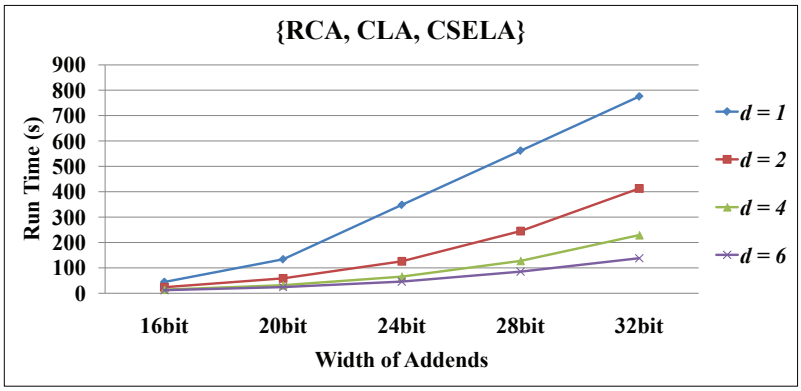
(b) $\mathcal{L} = \{\mathcal{CLA}, \mathcal{CSKA}\}$



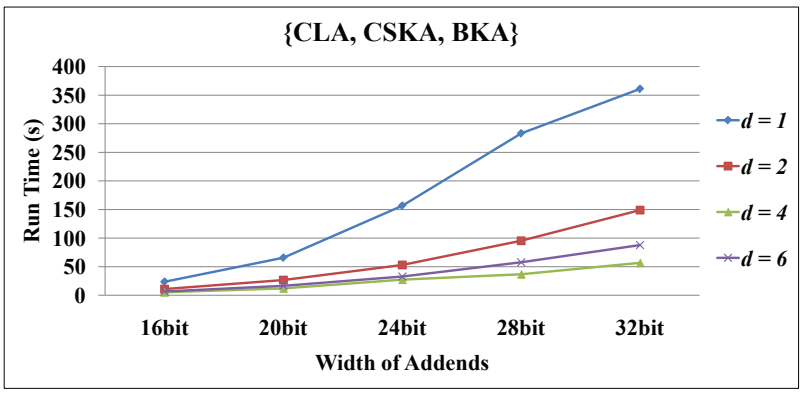
(c) $\mathcal{L} = \{\mathcal{CSKA}, \mathcal{BKA}\}$



(d) $\mathcal{L} = \{\mathcal{RCA}, \mathcal{CLA}, \mathcal{CSKA}\}$



(e) $\mathcal{L} = \{\mathcal{RCA}, \mathcal{CLA}, \mathcal{CSLA}\}$



(f) $\mathcal{L} = \{\mathcal{CLA}, \mathcal{CSKA}, \mathcal{BKA}\}$

Figure 5.3: Run times of AHA scheme for various values of parameter d, library $\mathcal{L}$, and bit-width of addends.

Chapter 6

Conclusion

This dissertation proposed a new hybrid adder design scheme. Contrary to the conventional hybrid adder scheme in which the target application was confined to the synthesis of final adder in the fast multiplier design, our scheme targeted re-optimization strategy where the design was under a stringent timing violation and an addition logic was on a critical delay path of the design. This dissertation proposed a new systematic hybrid adder design scheme, called adaptable hybrid adder scheme, to customize the addition structure by combining pure sub-adders effectively to meet the timing constraint. The proposed adder design scheme will be practically very useful in finding a new adder structure or resynthesizing an existing adder under a tight timing budget, which otherwise, a complete restructuring or reoptimizing of the entire design shall be needed.

Bibliography

[1] V. Oklobdzija and D. Villeger, "Improving multiplier design by using improved column compressiontree and optimized final adder in cmos technology," *IEEE Transactions on VLSI Systems*, vol. 3, no. 2, pp. 292–301, 1995.

[2] S. Das and S. P. Khatri, "Generation of the optimal bit-width topology of the fast hybrid adder in a parallel multiplier," in *Proceedings of the 11th International Conference on IC Design and Technology (ICICDT'07)*. Los Alamitos, CA: IEEE, 2007, pp. 1–6.

[3] "Synopsys timing constraints and optimization user guide," 2010, http://www.synopsys.com.

[4] B. Parhami, *Computer Arithmetic Algorithms And Hardware Designs*. New York, NY: Oxford University Press, 1999.

[5] O. J. Bedrij, "Carry-select adder," *IRE Transactions on Electron Computers*, no. 3, pp. 340–346, 1962.

[6] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, no. 3, pp. 260–264, 1982.

[7] R. Ladner and M. Fischer, "Parallel prefix computation," *Journal of ACM*, vol. 27, pp. 831–838, 1980.

[8] P. M. Kogge and H. S. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence equations,” *IEEE Transactions on Computers*, no. 8, pp. 786–793, 1973.

[9] H. Ling, “High-speed binary adder,” *IBM Journal of Research and Development*, vol. 5, no. 3, pp. 156–166, 1981.

[10] S. Knowles, “A family of adders,” in *Proceedings of 14th IEEE Symp. Computer Arithmetic*, 1999, pp. 14–16.

[11] S. Mathew, M. Anders, R. Krishnamurthy, and S. Borkar, “A 4 ghz 130 nm address generation unit with 32-bit sparse-tree adder core,” in *Symp. VLSI Circuits Digest of Technical Papers*, 2002, pp. 126–127.

[12] R. Zlatanovic, S. Kao, and B. Nikolic, “Energy-delay optimization of 64-bit carry-lookahead adders with a 240ps 90nm cmos design example,” *IEEE Journal of Solid-State Circuits*, vol. 44, no. 2, pp. 569–583, 2009.

[13] B. Zeydel, D. Baran, and V. Oklobdzija, “Energy-efficient design methodologies: high-performance vlsi adders,” *IEEE Journal of Solid-State Circuits*, vol. 45, no. 6, pp. 1220–1233, 2010.

[14] T. Han and D. Carlson, “Fast area-efficient vlsi adders,” in *Proceedings of 8th IEEE Symp. on Computer Arithmetic*, 1987, pp. 49–56.

[15] T. Lynch and E. E. Swartzlander, “A spanning tree carry lookahead adder,” *IEEE Transactions on Computers*, vol. 41, no. 8, pp. 931–939, 1992.

[16] V. Kantabutra, “A recursive carry lookahead - carry select hybrid adder,” *IEEE Transactions on Computers*, vol. 42, no. 12, pp. 1495–1499, 1993.

[17] Y. Wang, C. Pai, and X. Song, "The design of hybrid carry lookahead - carry select sdders," *IEEE Transactions on Circuit and Systems*, vol. 49, no. 1, pp. 16–24, 2002.

[18] G. Dimitrakopoulos and D. Nikolos, "High-speed parallel-prefix ling adders," *IEEE Transactions on Computers*, vol. 54, no. 2, pp. 225–231, 2005.

[19] J. Lee, J. Lee, B. Lee, and M. Ercegovac, "A design method for heterogeneous adders," in *Proceedings of the 3rd International Conference on Embedded Software and Systems (ICESS'07)*. Berlin Heidelberg: Springer-Verlag, 2007, pp. 121–132.

[20] I. Koren, *Computer arithmetic algorithms*. Natick, MA: A. K. Peters, 2001.

[21] P. F. Stelling and V. G. Oklobdzija, "Design strategies for optimal hybrid final adders in a parallel multiplier," *Journal of VLSI Signal Processing*, vol. 14, no. 3, pp. 321–331, 1996.

[22] P. Stelling and V. G. Oklobdzija, "Implementing multiply-accumulate operation in multiplication time," in *Proceedings of the 13th Symposium on Computer Arithmetic (ARITH'97)*. Washington D.C., DC: IEEE Computer Society, 1997, pp. 99–106.

[23] R. Zimmermann, "Non-heuristic optimization and synthesis of parallel-prefix adders," in *Proceedings of Int. Workshop on Logic and Architecture Synthesis*, 1996, pp. 123–132.

[24] R. Zimmermann and D. Tran, "Optimized synthesis of sum-of-products," in *Proceedings of 37th Asilomar Conference on Signals, Systems, and Computers*, 2003, pp. 9–12.

[25] T. Kim, W. Jao, and S. Tjiang, "Circuit optimization using carry-save-adder cells," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 10, pp. 974–984, 1998.

[26] J. Um and T. Kim, "An optimal allocation of carry-save-adders in arithmetic circuits," *IEEE Transactions on Computers*, vol. 50, no. 3, pp. 215–233, 2001.

[27] V. G. Oklobdzija, D. Villeger, and S. S. Lin, "A method for speed optimized partial product reduction and generation of fast parallel multiplier using an algorithmic approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–306, 1996.

[28] P. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi, "Design strategies for optimal multipler circuits," *IEEE Transactions on Computers*, vol. 47, no. 3, pp. 273–285, 1998.

국문 초록

CMOS 반도체 소자의 공정이 미세 공정으로 변화하면서, 회로의 시간제약을 만족시키는 것이 집적회로 설계에 있어서 점점 더 중요해지고 있으며, 집적회로에서 가장 중요하게 시간에 영향을 끼치는 경로에는 가산기, 감산기, 그리고 곰셈기와 같은 연산 요소들이 포함되어 있다. 감산기와 곰셈기는 덧셈기로 구현될 수 있기 때문에, 곰셈기에 대한 동작 속도를 향상시키기 위한 많은 연구들이 있어왔다. 본 논문은 가장 중요하게 시간에 영향을 끼치는 회로의 경로 상의 덧셈기에 대해 혼성 덧셈기 구조를 사용하여 시간 제약을 만족시키면서 동시에 덧셈기의 면적을 줄이는 방법을 제안한다. 이전의 혼성 덧셈기의 구조는 균일하거나 특정한 형태의 입력 시간을 가정하였다. 하지만 본 논문에서 제안되어지는 방법은 실재의 회로에서 입력 시간 뿐만 아니라 출력 단에서의 필요시간을 추출하여 이를 덧셈기의 최적화에 사요한다. 특히 본 논문에서는 효율적인 혼성 덧셈기의 제거 방법을 사용하여, 동적 프로그래밍에 기반한 혼성 덧셈기의 설계를 위한 체계적인 방법을 제시한다. 본 논문에서 제안되는 방법은 시간 제약이 심한 상황에서 연산 집중적인 회로의 시간을 최적화 하는데 사용될 수 있다는 데 있어서, 실질적이다고 할 수 있다. 본 논문에서 이와 관련한 여러 상황에 대하여 본 논문에서 제안되어지는 방법이 순수한 덧셈기나 이전 연구에 비해 얼만큼 시간과 면적에 대하여 효율적으로 최적화 할 수 있는 지에 대한 다양한 실험 자료들을 제공한다.

주요어: Hybrid adder, RTL resynthesis, arithmetic optimization, timing optimization
학번: 2007-30216

ACKNOWLEDGMENT

어느덧 7년이라는 시간이 지났습니다. 돌이켜 보면 지난 시간들 동안 많은 일들이 있었고 그 와중에 과분할 정도로 많은 사랑을 받아왔던 것 같습니다.

먼저, 지난 7년여의 대학원생활 동안 부족한 저를 이끌어주시고 지도해주신 김태환 교수님께 진심으로 감사 드립니다. 막상 연구실을 떠나려 하니 제가 얼마나 든든한 그늘 밑에 있었는지 새삼 깨닫게 됩니다. 연구자로서 갖추어야 할 덕목과 태도뿐만 아니라 한 명의 사람으로서 갖추어야 하는 것이 무엇인지 교수님을 통하여 배울 수 있었습니다. 교수님의 가르침을 가슴 깊이 새기며 살아가도록 하겠습니다.

더불어 대학원생활을 하는 동안 가족처럼 의지하고 힘이 되어준 시스템합성 연구실 선후배님들께도 감사의 말씀을 전합니다. 여러분들과 연구실에서 함께 한 추억들은 평생 큰 힘이 되어줄 것 같습니다.

또한 가까이서 멀리서 저와 함께해준 친구들, 동아리 선후배님들, 동문회 선후배님들과 친척분들께도 감사 드립니다.

마지막으로 학위 과정을 무사히 마칠 수 있도록 저를 응원해주시고 든든한 버팀목이 되어주신, 세상에서 가장 사랑하고 존경하는 아버지, 어머니와 형님, 그리고 형수님. 진심으로...감사 드립니다.