FAST CIRCUIT TOPOLOGIES FOR FINDING THE MAXIMUM OF n k-BIT NUMBERS

by

Bilgiday Yüce

B.S., Electrical & Electronics Engineering,

TOBB University of Economics and Technology, 2010

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical & Electronics Engineering
Boğaziçi University
2013

FAST CIRCUIT TOPOLOGIES
FOR FINDING THE MAXIMUM OF n k-BIT NUMBERS

APPROVED BY:

Prof. Günhan Dündar (Thesis Supervisor)	....
Assoc. Prof. H. Fatih Uğurdağ (Thesis Co-supervisor)	....
Assoc. Prof. Şenol Mutlu	....
Assoc. Prof. Sezer Gören Uğurdağ	....
Assist. Prof. İ. Faik Başkaya	....

DATE OF APPROVAL: 11.06.2013

ACKNOWLEDGEMENTS

It would not have been possible to bring about this thesis without the support and help of several individuals around me, to only some of whom it is possible to give particular mention here.

First, I would like to thank my thesis advisors Assoc. Prof. H. Fatih Uğurdağ and Prof. Günhan Dündar for their invaluable guidance and support throughout the development of this thesis. I would like to express my deepest gratitude to Assoc. Prof. H. Fatih Uğurdağ for his guidance, care, and patience. He also provided me an excellent atmosphere throughout my research. His continuous help with all kind of difficulties, which I encountered during my M.S. years, and his trust in me always kept me motivated. I am also really thankful to Prof. Günhan Dündar for dedicating his valuable time for my thesis and for his suggestions that have enhanced this thesis.

I would like to thank Assoc. Prof. Sezer Gören, Assoc. Prof. Şenol Mutlu and Asst. Prof. İ. Faik Başkaya for taking part in my thesis committee and for their time. In particular, Assoc. Prof. Sezer Gören's helpful comments, insightful discussions, and constructive feedback have greatly improved the quality of this work.

Also, I want to thank all of my friends who made my life enjoyable throughout my graduate education. Especially, I want to mention Abdullah Sarıduman, Barış Esen, Berkan Yaman, Doğan Ulus, Erdem Ulusoy, Fatih Temizkan, Furkan Şahin, Gökhan Güner, İsmail Kara, İsmail Terkeşli, Kenan Türksoy, Mert Kaya, and Waqas Hussain.

In addition, I would also like to thank TÜBİTAK for supporting me through BİDEB scholarship program between 2010-2012 and through ARDEB in 2013.

Finally, I am most grateful to my family members for their endless support and love throughout my life.

ABSTRACT

FAST CIRCUIT TOPOLOGIES FOR FINDING THE MAXIMUM OF n k-BIT NUMBERS

Finding the value and/or address (position) of the maximum (or similarly minimum) element of a set of binary numbers is a fundamental arithmetic operation. Numerous systems, which are used in various application areas, require fast (low-latency) circuits to carry out this operation. In this thesis, we present a detailed literature survey of previous works and propose three circuit topologies that determine both value and address of the maximum (or similarly minimum) element within an n-element set of k-bit binary numbers. Our proposed topologies are Array-based Topology (AbT), Hybrid Binary tree Topology (HBT), and Quad tree Topology (QT). The timing complexity of the fastest proposed architecture (AbT) is $O(\log_2 n + \log_2 k)$, whereas the timing complexity of the fastest topology in previous work is $O(\log_2 n \log_2 k)$. We wrote RTL code generators for the proposed topologies as well as their competitors. These automated generators are scalable to any value of n and k. Then, we applied a standard-cell based iterative synthesis flow, which finds the optimum timing through binary search. We obtained area, power consumption, and timing results for the proposed topologies as well as their competitors. Using these results, we also compute some combined performance metrics such that area-timing product (ATP), area-timing-square product (AT2P), power-timing product (PTP), and energy-timing product (ETP). The synthesis results showed that on the average, AbT is 1.61 times, QT is 1.28 times, and HBT is 1.01 times faster than the fastest in the literature.

ÖZET

n ADET *k*-BİT SAYININ EN BÜYÜĞÜNÜ BULMAK İÇİN HIZLI DEVRE TOPOLOJİLERİ

Bir sayı kümesinin maksimum (ya da minimum) elemanının değerini ve/veya adresini (pozisyonunu) bulmak en temel aritmetik işlemlerden biridir. Çeşitli uygulama alanlarındaki birçok sistem, bu işlemi yerine getiren hızlı devrelere ihtiyaç duyar. Bu tezde, n tane k-bit uzunluğunda sayının en büyüğünün (benzer biçimde en küçüğünün) hem değerini hem de adresini bulan devreler için detaylı bir literatür taraması ve üç tane yeni devre topolojosi sunuyoruz. Önerdiğimiz topolojileri şu şekilde adlandırıyoruz: "Array-based Topology (AbT)", "Hybrid Binary tree Topology (HBT)" ve "Quad tree Topology (QT)". Önerilen topolojilerden en hızlısı (AbT) işini $O(\log_2 n + \log_2 k)$ sürede tamamlarken, literatürdeki en hızlı topolojinin işini tamamlaması için $O(\log_2 n \log_2 k)$ süre gerekir. Hem literatürdeki topolojiler hem de önerilen topolojiler için HDL kod üreteçleri yazdık. Bu otomatik kod üreteçleri herhangi bir n ve k değeri için ilgili topolojinin HDL kodunu üretebilecek yetenektedirler. Daha sonra ise, en iyi zaman kısıtını ikilik arama algoritmasına benzer bir yaklaşım ile bulan, yinelemeli ve standard-devre tabanlı bir sentez süreci uyguladık. Böylece, hem önerilen topolojiler hem de literatürdeki topolojiler için alan, güç tüketimi ve zaman sonuçlarını elde ettik. Ayrıca, bu sonuçları kullanarak şu birleşik başarım kıstaslarını da hesapladık: alan-zaman çarpımı (AZÇ), alan-zaman-kare çarpımı (AZ2Ç), güç-zaman çarpımı (GZÇ) ve enerji-zaman çarpımı (EZÇ). Sentez sonuçları, literatürdeki en hızlı devrenin, ortalamada AbT'den 1.61 kat, QT'den 1.28 kat ve HBT'den 1.01 kat daha yavaş olduğunu gösterdi.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

A	Area
$\mathbf{D}$	Set of n k-bit numbers
D_i	i-th element of $\mathbf{D}$
$d_{n-1:0,j}$	bit-slice j of the n input elements
D_W	Maximum (Winner) element of $\mathbf{D}$
f	Frequency
k	Bitwidth of an input element
n	Number of input elements
O	Asymptotic O-notation
P	Power Consumption
t	Time
V_{DD}	Supply voltage
W	The address (index) of the maximum element
α	Switching activity (Node transition activity factor)

LIST OF ACRONYMS/ABBREVIATIONS

AbT	Array based Topology
AGU	Address Generation Unit
AT	Array Topology
AT2P	Area-Timing-Square Product
ATP	Area-Timing Product
BKT	Brent-Kung Topology
BM	Binary Merger
CAU	Comparator Array Unit
CET	Carry-lookahead Equation Topology
CLT	Carry Lookahead parallel binary tree Topology
CST	Carry Select parallel binary tree Topology
DSU	Data Selection Unit
FF	Flip-Flop
FPGA	Field Programmable Gate Array
HBT	Hybrid Binary tree Topology
HCT	Han-Carlson Topology
HDL	Hardware Description Language
KST	Kogge-Stone Topology
LCT	Leading-zero Counting Topology
LFT	Ladner-Fischer Topology
LSB	Least Significant Bit
MF	Maximum Finder
MLT	Multi-Level Topology
MSB	Most Significant Bit
PBT	Parallel Binary tree Topology
PPN	Parallel Prefix Network
PT2P	Power-Timing-Square Product
PTP	Power-Timing Product
QT	Quad tree Topology

RCT	Ripple Carry parallel binary tree Topology
RT	Row Topology
RTL	Register-Transfer Level
ST	Selection Topology
TBT	Traditional Binary tree Topology
WLM	Wire-Load Model

1. INTRODUCTION

This chapter introduces the motivation behind this thesis and familiarizes the readers with the subject. Summary of contributions and outline of the thesis also appear in this chapter.

1.1. Motivation

Fast (low-latency) maximum (or similarly minimum) finder (MF) circuits, which determine the maximum (or minimum) number in an unsorted (arbitrary) list of unsigned binary numbers and/or its address (position) in the list, play an important role in a wide range of application areas such as communication systems [6], video processing [7], computer networks [8], real-time systems [9], and sorting networks [10]. Since any minimum finder circuit can be easily converted into a maximum finder circuit with minor modifications (such as swapping operands in comparisons), and vice versa, we include minimum as well as maximum finder circuits in the set of possible applications. In other words, any MF topology is both a maximum and minimum finder topology.

Depending on the application, MF circuits are required to produce the value and/or address of the maximum element. For instance, the motion estimation hardware proposed in [7] requires only the value of the maximum element. On the other hand, the schedulers in [8] need only address of the maximum. Other examples are the real-time task managers in [9], and the channel estimation architecture in [6], which require both the value and address of the maximum. Hence, fast computation of both the value and address of the maximum element become equally important.

Moreover, the number of elements (n) to be compared and bit-width (k) of each element are also dependent upon the application. For the circuits in [6–10], n and k are in the range of 8 to 64. However, we cannot constrain the range and combinations of parameters (n,k) with only these values, since finding the maximum is a fundamental problem, and thus, it can be used in numerous applications.

1.2. Contributions of the Thesis

In this thesis, we analyze existing MF circuit topologies and then propose three new fast MF circuit topologies, which can determine both the value and the address of the maximum element within an n-element set of k-bit binary numbers. The address may be one-hot (n bits) and/or binary ($\log_2 n$ bits).

Additionally, to evaluate the performance of existing and proposed topologies, we wrote Perl scripts for HDL code generation of all topologies. Our scripts also generate self-checking testbench code for functional verification besides RTL code (which will be synthesized). In addition, we also wrote synthesis (batch) scripts for a standard-cell based iterative flow that finds the optimum timing through binary search [11–13]. In other words, we automated not only design and functional verification but also synthesis of both existing and proposed MF circuits.

Our contributions can be summarized as follows:

- We provide a detailed literature survey of existing MF topologies, including *Array Topology (AT)* [1], *Row Topolgy (RT)* [2], *Selection Topology (ST)* [3], *Traditional Binary tree Topology (TBT)*, *Parallel Binary tree Topology (PBT)* [4, 14], *Multi-Level Topology (MLT)* [5], and *Leading-zero Counting Topology (LCT)* [5].
- While implementing Carry Lookahead PBT (CLT), we employed four pioneering *Parallel Prefix Network (PPN)* topologies such that Brent-Kung (BK) [15], Han-Carlson (HC) [16], Kogge-Stone (KS) [17], and Ladner-Fischer (LF) [18]. In addition, we implemented another version of CLT, in which we directly wrote the carry lookahead equations in Verilog.
- We propose three MF circuit topologies, namely, *Array based Topology (AbT)*, *Hybrid Binary tree Topology (HBT)*, and *Quad tree Topology (QT)*.
- We provide a comparison of existing and proposed topologies in terms of theoretical time, area and power consumption costs.
- The contribution of AbT is providing the fastest MF topology of which timing complexity is $O(\log_2 n + \log_2 k)$. On the contrary, state-of-the-art has a timing

complexity of $O(\log_2 n \, \log_2 k)$.

- QT is an hybrid topology, which combines the speed advantage of AbT and area advantage of quad tree configuration. Its timing complexity ($O(\log_4 n \, \log_2 k)$) is better than any existing topology as well as its area complexity ($O(\log_2 n \, \log_2 k)$) is same as area complexity of the most area-efficient existing topology.
- HBT is another hybrid topology, which provides a generalization for PBT by thinking of an $(n{,}k)$-HBT as a combination of $\lceil k/p \rceil$ cascaded $(n{,}p)$-TBTs. The timing complexity of HBT ($O\left((k/p + \log_2 n) \, \log_2 p\right)$) depends on the value of the partition (p) parameter, while its area complexity is ($O(\log_2 n \, \log_2 k)$) is same as area complexity of the most area-efficient existing topology.
- We wrote an estimator script in Perl to find the optimum partition (p) value of HBT that leads to the maximum clock frequency for a given $(n{,}k)$ combination.
- We developed HDL code generators (in Perl) for almost all of the MF topologies mentioned above. Due to their high theoretical time complexities, we did not write generators for AT, RT, and ST. The code generators produces a Verilog RTL to be synthesized, an automatic testbench for pre- and post-synthesis functional verification, and a wrapper for proper timing computation after synthesis.
- We obtained a rich set of timing, power consumption and area results using an automated standard-cell based logic synthesis flow. The flow includes wire-load models and uses an iterative synthesis strategy [11–13].
- We also provide a comparison of proposed topologies and their competitors (existing topologies) in terms of empirical time, power consumption, and area results.

1.3. Outline of the Thesis

Rest of the report is organized as follows. In Chapter 2, we introduce the basic concepts and notations used in this thesis. Chapter 3 gives an overview of the related work. Chapter 4 presents the details of the proposed topologies. In Chapter 5, we explain our synthesis methodology and RTL code generators. Chapter 6 provides experimental results for the proposed topologies and their competitors. And Chapter 7 gives the conclusions and future work.

2. PRELIMINARIES

Since finding the maximum (or similarly minimum) element of a set of binary numbers and/or its address is a fundamental arithmetic operation, it can take part in numerous applications in different formats. Therefore, we first give a formal definition of the problem, and we introduce the notations as well as definitions we use in the thesis (Section 2.1). Next, we give circuit complexity and performance models, which are used for theoretical comparison of MF circuit topologies (Section 2.2).

2.1. Problem Definition

The problem of finding the maximum of n k-bit unsigned binary numbers is formally defined as follows:

Given a set $\mathbf{D} = \{D_0, D_1, \ldots, D_{n-1}\}$ of n unsigned binary numbers, an MF circuit produces output(s) D_W and/or W, where D_W denotes the maximum (winner) element of $\mathbf{D}$ ($D_W \geq D_i, \forall D_i \in \mathbf{D}$) and W denotes its address (position).

We use the following notations throughout this thesis report:

- We will use (n,k)-*name* notation to denote a circuit/topology, which is called *name* and has n k-bit inputs.
- We will represent the binary version of the address as $W = (w_{\lceil \log_2 n \rceil -1} w_{\lceil \log_2 n \rceil -2} \ldots w_0)$.
- We will represent the one-hot version of the address as $W = (m_{n-1} m_{n-2} \ldots m_0)$.
- We will represent each element D_i as a k-bit unsigned number such that $D_i = (d_{i,k-1} d_{i,k-2} \ldots d_{i,1} d_{i,0})$.
- We will also use $d_{n-1:0,j}$ notation to denote bit-slice (i.e., collection) j $(d_{n-1,j} d_{n-2,j} \ldots d_{0,j})$ of n elements.

Note also that the maximum element does not necessarily have to be unique. In the case of multiple maximum elements, MF circuits choose one of those maximum elements as the winner element applying a priority scheme. Hence, we assumed as if the maximum element was unique in our definition and the assumption will continue to be valid throughout this report.

In this thesis, we will examine several MF circuits and topologies. The *topology* of an MF circuit specifies how basic building blocks of the circuit are connected to each other (independent of n and k parameters). Topology does not deal with the exact internal implementation of building blocks; it deals with how the building blocks are interconnected. In other words, changing the implementation of some or all instances of a building block does not cause any topological change. On the other hand, we will use the term *circuit* to denote any specific hardware implementation of a topology.

2.2. Circuit Complexity and Performance Modeling

We use simple complexity and performance models for a rough characterization of gate-level MF circuits. Given a circuit, specified by a logic equation or a netlist built from basic logic gates, we need estimations of the expected area, time (speed), and power consumption for a synthesized circuit as a function of the parameters n and k.

2.2.1. Area Modeling

The silicon real estate a VLSI circuit occupies is the sum of its logic cells as well as their interconnections. Total *logic cell area* can be estimated from logic equations and gate-level netlists. In addition, wiring area is proportional to total wire length, which can be estimated from the sum of cell fanout. Since total cell fanout is also proportional to total circuit size, we can claim that total *wiring area* is also proportional to the number of transistors [19].

Taking into account the above criteria, we use *unit-gate area model* [20] in this thesis to estimate the area of a circuit from logical equations and netlists consisting of simple logic gates. Unit-gate area model can be summarized as follows:

- A unit-gate is an elementary 2-input gate, such as AND, NAND, OR, and NOR, except XOR gate.
- An XOR gate is counted as two unit gates.
- Areas of inverters and buffers are ignored.
- Area of an elementary m-input gate is $m-1$ because the m-input gate is thought of as a binary tree configuration of $m-1$ 2-input simple gates.

2.2.2. Delay Modeling

Delay of a VLSI circuit is determined by logic cells and wires on the critical path of the circuit, which can be defined as the path between an input and an output with the maximum delay. Critical path evaluation is done by “static timing analysis”, which is a method of computing expected timing of digital circuits utilizing graph algorithms. Although timing is also dependent on temperature, voltage, and process variations, we assume they affect all circuits equally.

The *total critical path delay* is the sum of *logic cell delays*, *output transition delays*, and *wire delays* on the respective path. Logic cell delay is determined by transistor-level implementation and complexity of the cell. Output transition delay is the time it takes for a logic cell output to charge/discharge capacitances, which consists of interconnect and cell input loads and are proportional to the fanout of the cell. The wire delay is the RC-delay of interconnects [19].

Considering the above factors, we use *unit-gate delay model*, similar to the unit-gate area model in this thesis to estimate the delay of a circuit from logical equations and netlists consisting of simple logic gates. Unit-gate delay model can be summarized as follows:

- A unit-gate is an elementary 2-input gate, such as AND, NAND, OR, and NOR, except XOR gate.

- An XOR gate is counted as two unit gates.
- Delays of inverters and buffers are ignored.
- Delay of an elementary m-input gate is $\lceil \log_2 m \rceil$ because the m-input gate is thought of as a binary tree configuration of $m - 1$ 2-input simple gates.

2.2.3. Power Consumption Modeling

The power dissipation of a CMOS circuit is dominated by the dynamic dissipation resulting from charging and discharging capacitances. If we ignore the dynamic short-circuit currents and static leakage power, the total power consumption can be written as:

$$P_{total} = \sum_i C_{total}\, V_{DD}^2\, f_{clk}\, \alpha$$

Here, C_{total} is the total load capacitance at node i, V_{DD} is the supply voltage, f_{clk} is the clock frequency, and α is the power consuming switching activity (or node transition activity factor) at node i. Switching activity of a node can be defined as the probability that a clock event results in a power consuming transition at this node. Total capacitance is roughly proportional to the circuit size. Supply voltage and clock frequency can be considered as constant throughout the circuit and therefore is not relevant in circuit comparisons. Switching activity is a function of nature and statistics of input signals as well as overall circuit topology and function to be implemented.

Hence, if we assume supply voltage, clock frequency, and input switching activity values are the same for all circuits, power consumption of the circuit becomes approximately proportional to circuit size. Therefore, *unit-gate area* model can be used for estimation of power consumption of a circuit.

2.2.4. Combined Circuit Performance Measures

Depending on the requirements of the design, circuit performance can be measured in terms of either area, time, power consumption, or combination of those.

Some of the most common combined performance measures are *area-timing product (ATP)*, *power-timing product (PTP)*, *area-timing-square product (AT2P)*, and *area-square-timing product (A2TP)*. ATP roughly indicates the circuit size per data items processed per time unit. AT2P tries to stress time objective than area while A2TP stresses area objective more than time. PTP can be regarded as the power dissipation per computation. The common property of all those combined performance metrics is that the smaller the used combined metric is, the higher the performance of the corresponding circuit is. Also, we can use aforementioned unit-gate models to calculate these combined performance metrics.

2.2.5. Asymptotic O-notation

Using the circuit complexity and performance models given in Sections 2.2.1 up to 2.2.4, we obtain a *complexity function*, mathematical function of input parameters, for the circuit. For MF circuits examined in this thesis, input parameters are the count of the numbers to be compared (n) and the bitwidth of each number (k). Hence, we obtain complexity functions such that $g_{complexity} = g(n, k)$.

In comparing the complexities of different circuits, not all terms of a circuit's complexity function is essential. Instead, only the *rate of growth* or the *order of magnitude* of the complexity function is typically of the most concern [21]. *Asymptotic notations* are symbols used in computational complexity theory [22] to express the efficiency of algorithms with a focus on their orders of growth. In this thesis, we use one of the most widely-used asymptotic notations to compare efficiency of our circuits' complexity functions: O-notation.

O-notation denotes the *asymptotic upper bounds* of the complexity functions. Given to functions $g(n)$ and $h(n)$ that map a natural number n to some positive value, it is said that $g(n) = O(h(n))$ (pronounced as "$g(n)$ is big-oh of $h(n)$"or as "$g(n)$ is of order at most $h(n)$) if two constants c and n_0 exist such that the following proposition is true:

$$\forall n \geq n_0 : g(n) \leq c\, g(n)$$

Note also that $O(h(n))$ denotes a set of functions and the equality sign in $g(n) = O(h(n))$ denotes the set membership, namely $g(n) \in O(h(n))$. The following are legal expressions in computational theory:

$$\begin{aligned}
\frac{1}{3}n^2 &= O(n^2) \\
n^3 + 1000n^2 + n &= O(n^3) \\
4n \log n + n &= O(n \log n) \\
2n^3 &= O\left(\frac{1}{7}n^3\right) \\
2n^3 &= O\left(n^3 + n^2\right)
\end{aligned}$$

Although the last three expressions are valid, we do not prefer them while describing complexities of our circuits. We choose the asymptotic notation to be as simple and as informative as possible. Table 2.1 lists some of the most common O-notations and their aliases in ascending complexity order.

Table 2.1. Frequently used orders of functions and their aliases.

***O*-notation**	**Alias**
$O(1)$	Constant complexity
$O(\log n)$	Logarithmic complexity
$O(n \log n)$	linearithmic, loglinear, or quasi linear
$O(n^2)$	Quadratic complexity
$O(n^3)$	Cubic complexity
$O(n^c)$, c >1	Polynomial complexity
$O(c^n)$, c >1	Exponential complexity
$O(n!)$	Factorial complexity

3. PREVIOUS WORK

In this chapter we shall examine the existing MF circuit topologies: *Array Topology* (AT) [1], *Row Topolgy* (RT) [2], *Selection Topology* (ST) [3], *Traditional Binary tree Topology* (TBT), *Parallel Binary tree Topology* (PBT) [4,14], *Multi-Level Topology* (MLT) [5], and *Leading-zero Counting Topology* (LCT) [5].

3.1. Array Topology (AT)

AT [1] is best described as a filter. It starts with all elements being candidates for the maximum, looks at each bit from most significant bit (MSB) to least significant bit (LSB), and narrows down the candidates in each iteration until only one is left. An (n,k)-AT circuit is organized as a k-by-n array of AT-*cells* as shown in Figure 3.1. Row r of the array consists of n AT-cells that form bit-slice r of n elements. Each column c of the array consists of k AT-cells corresponding to D_c. AT first initializes the enable signals ($en_{0:n-1,k-1}$) of the AT-cells residing in $(k-1)$-th row of AT, and then applies the following steps for each bit-slice (row) r ($r = k-1$ DOWNTO 0) of AT, as shown in the second row of Figure 3.1:

(i) Each cell c in the bit-slice performs an AND operation on its input $d_{c,r}$ and its corresponding enable signal ($en_{c,r}$) using a 2-input AND gate (labelled with i in Figure 3.1).

(ii) An n-input OR gate (labeled with ii in Figure 3.1) computes the $d_{W,r}$ (r-th bit of the winner element D_W) using the 1-bit signals produced in Step (i).

(iii) Computed $d_{W,r}$ is inverted by an inverter (labeled with iii in Figure 3.1).

(iv) Each cell c in the bit-slice performs an AND operation on its input $en_{c,r}$ with the inverted signal in Step (iii) using a 2-input AND gate (labeled with iv in Figure 3.1).

(v) Each cell c in the bit-slice computes the enable signal $en_{c,r-1}$ of the corresponding AT-cell in the next bit-slice $r-1$ with an 2-input OR gate(labeled with v in Figure 3.1).

As it can be seen, the two 2-input AND gates (*i* and *iv* in Figure 3.1) filters out the elements, whose enable signal is zero. If the enable signal of an element is 1, the 2-input AND gate (*iv* in Figure 3.1) checks whether the corresponding bit of the element is the same as the corresponding bit of the winner element, which is computed by the n-input OR gate (*ii* in Figure 3.1). If these two bits are equal, the enable signal of the corresponding element becomes 1 for the next row and becomes zero otherwise. In AT,

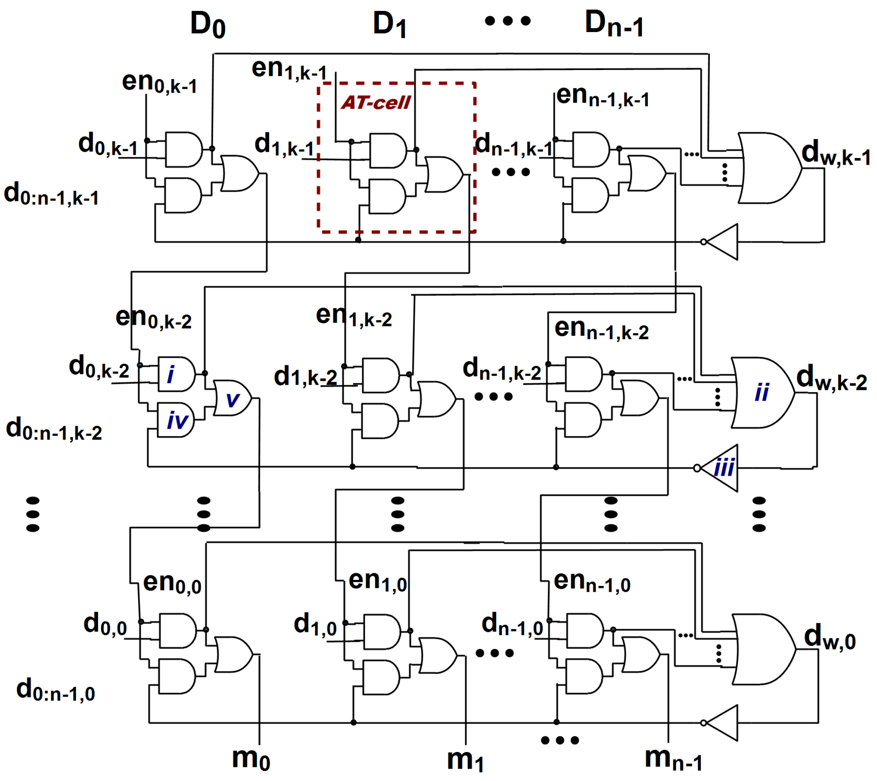


Figure 3.1. Array Topology [1].

enable (en) signals ripple down from MSB to LSB. Thus, $d_{W,j}$ cannot be determined until the $en_{n-1:0,j+1}$ signals settle down.

As it can be seen in Figure 3.1, AT produces the value and one-hot address of the maximum (winner) element.

3.1.1. Theoretical Complexity of AT

An AT-cell consists of two 2-input AND gates and one 2-input OR gate, and thus, we can write the following complexity expressions for an AT-cell, using the complexity models discussed in Section 2.2:

$$t_{AT\text{–}cell} = t_{AND2} + t_{OR2} = 1 + 1 = 2 \tag{3.1}$$

$$A_{AT\text{–}cell} = 2 \times A_{AND2} + A_{OR2} = 2 \times 1 + 1 = 3 \tag{3.2}$$

$$P_{AT\text{–}cell} = A_{AT\text{–}cell} = 3 \tag{3.3}$$

Complexity functions of each n-input OR gate that resides in the rows of AT:

$$t_{ORn} = \log_2 n \times t_{OR2} = \log_2 n \tag{3.4}$$

$$A_{ORn} = (n-1) \times A_{OR2} = n - 1 \tag{3.5}$$

$$P_{ORn} = A_{ORn} = n - 1 \tag{3.6}$$

The total complexity of AT:

$$\begin{aligned} t_{AT} &= k \times (1 + t_{ORn} + t_{AT\text{–}cell}) = k \times (1 + \log_2 n + 2) \\ &= k \times (\log_2 n + 3) \\ &= O(k \times \log_2 n + k) \end{aligned} \tag{3.7}$$

$$\begin{aligned} A_{AT} &= k \times n \times A_{AT\text{–}cell} + k \times A_{ORn} \\ &= k \times n \times 3 + k \times (n-1) = 4 \times k \times n - k \\ &= O(k \times n) \end{aligned} \tag{3.8}$$

$$P_{AT} = A_{AT} = 4 \times k \times n - k = O(k \times n) \tag{3.9}$$

3.2. Row Topology (RT)

RT [2] determines the maximum of n elements in k cycles, where k is the width of each element. RT can be thought of as a row of AT with additional storage elements within *RT-cells* as seen in Figure 3.2. RT also uses a shift register to store the value

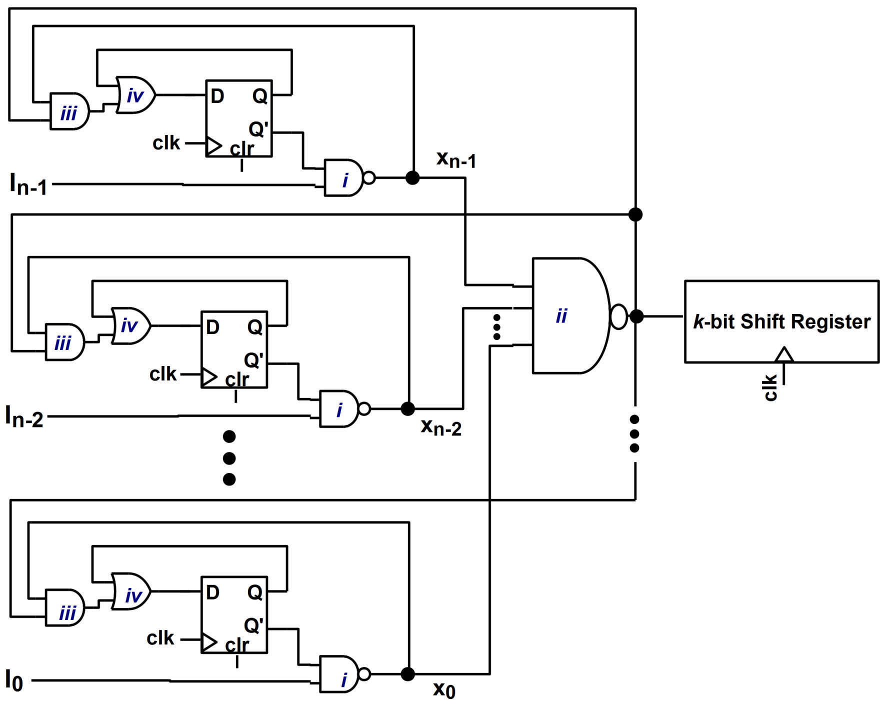


Figure 3.2. Row Topology [2].

of the maximum element. RT computes one bit of the maximum value in each cycle, from MSB to LSB. After k cycles, the maximum value becomes available in the shift register. One-hot address of the maximum element is also available in cycle k. RT first clears all the flip-flops with a *start pulse* and then applies the following steps for each clock cycle c ($c = k - 1$ DOWNTO 0), as shown in Figure 3.2:

(i) The bit-slice $d_{n-1:0,c}$ is made available at the input ports $I_{n-1:0}$ respectively.

(ii) $x_{n-1:0}$ signals are concurrently computed using 2-input NAND gates (labeled with i in Figure 3.2).

(iii) An n-input NAND gate (labeled with ii in Figure 3.2) computes the $d_{W,c}$ (c-th most significant bit of the winner element D_W) using $x_{n-1:0}$ signals produced in

Step (i).

(iv) When an input x of the n-input NAND gate equals to the $d_{W,c}$ computed in Step (iii), the corresponding D flip-flop changes its state (Q signal in Figure 3.2) in the next clock cycle $c-1$. This operation is concurrently carried out by 2-input AND gates (labeled with *iii* in Figure 3.2) and 2-input OR gates (labeled with *iv* in Figure 3.2).

(v) Finally, computed $d_{W,c}$ value is moved into the shift register with the next clock pulse.

Note also that $x_{n-1:0}$ signals at the final cycle give the one-hot address of the maximum element. In addition, RT is not our competitor as it computes the maximum element in multiple clock cycles.

3.2.1. Theoretical Complexity of RT

An RT-cell consists of a 2-input AND gate, a 2-input OR gate,a 2-input NAND gate, and a D flip-flop. Hence, we can write the following complexity expressions for an RT-cell, using the complexity models discussed in Section 2.2:

$$t_{RT\text{-}cell} = t_{NAND2} + t_{AND2} + t_{OR2} = 1 + 1 + 1 = 3 \tag{3.10}$$

$$A_{RT\text{-}cell} = A_{NAND2} + A_{AND2} + A_{OR2} + A_{DFF} = 1 + 1 + 1 + 4 = 7 \tag{3.11}$$

$$P_{RT\text{-}cell} = A_{RT\text{-}cell} = 7 \tag{3.12}$$

In above calculations, we ignore clock-to-q delay and setup time of the D flip-flop. Also, we take the area of the flip-flop as 4 unit gates as indicated in [2].

Complexity expressions for the n-input NAND gate:

$$t_{NANDn} = \log_2 n \times t_{NAND2} = \log_2 n \tag{3.13}$$

$$A_{NANDn} = (n-1) \times A_{NAND2} = n - 1 \tag{3.14}$$

$$P_{NANDn} = A_{NAND2} = n - 1 \tag{3.15}$$

The total complexity of RT:

$$\begin{aligned} t_{RT} &= k \times (t_{NANDn} + t_{RT\text{–}cell}) = k \times (\log_2 n + 3) \\ &= O(k \times \log_2 n + k) \qquad (3.16) \\ A_{RT} &= n \times A_{RT\text{–}cell} + A_{NANDn} + AShiftReg \\ &= n \times 7 + (n-1) + (4 \times k) = 8 \times n + 4 \times k - 1 \\ &= O(n+k) \qquad (3.17) \\ P_{RT} &= A_{RT} = O(n+k) \qquad (3.18) \end{aligned}$$

3.3. Selection Topology (ST)

In [3], finding the maximum is thought of as a subset of selection operation, which refers to finding an element of given order from an ordered set of elements. And an MF circuit topology, which we call Selection Topology (ST), was proposed. ST computes the position(s) of the maximum element(s) of a set of n k-bit numbers in k cycles, and outputs the result as an n-bit signal. In Figure 3.3, ST with the signals at the j-th cycle is given. Similar to AT and RT, it successively operates on bits of all numbers. Initially, each element is a candidate for the maximum element. Hence, flip-flops of Figure 3.3 are initially set to 1. In each cycle j, ST observes the j-th bit-slice $d_{n:0,j}$ of the input elements and decides if a particular element should remain as a candidate, or if it should be permanently rejected. Thus, we obtain a subset of the initial set that does not include the rejected elements, after each cycle j. After k cycles, we obtain the set of the maximum element(s) and outputs of the flip-flops show the positions of the maximum element(s). In order to select the maximum value we need an additional priority encoder followed by a multiplexer circuit. In addition, ST is not our competitor as it computes the maximum element in multiple clock cycles.

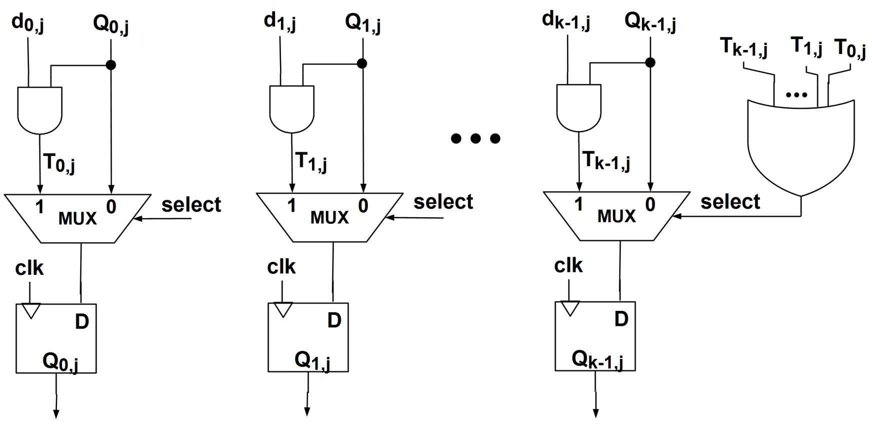


Figure 3.3. Selection Topology [3].

3.3.1. Theoretical Complexity of ST

Total complexity of ST:

$$\begin{aligned} t_{ST} &= k \times (t_{AND2} + t_{ORn} + t_{mux2}) = k \times (1 + \log_2 n + 2) \\ &= k \times (\log_2 n + 3) = O(k \times \log_2 n + k) \qquad (3.19) \\ A_{ST} &= n \times (A_{AND2} + A_{mux2} + A_{DFF}) + A_{ORn} \\ &= n \times 7 + (n-1) = 8 \times n - 1 \\ &= O(n) \qquad (3.20) \\ P_{ST} &= A_{ST} = O(n) \qquad (3.21) \end{aligned}$$

3.4. Traditional Binary Tree Topology (TBT)

The most widely used MF circuit topology in the literature is traditional binary tree topology (TBT), in which (2,k)-*cmp-mux* blocks are connected in a binary tree configuration. In Figure 3.4, an (8,k)-TBT is given as an example.

A (2,k)-*cmp-mux* in ℓ-th level of the binary tree, given in Figure 3.5, has two $(k+\ell)$-bit inputs (A and B in Figure 3.5) and a $(k+\ell+1)$-bit output. The most

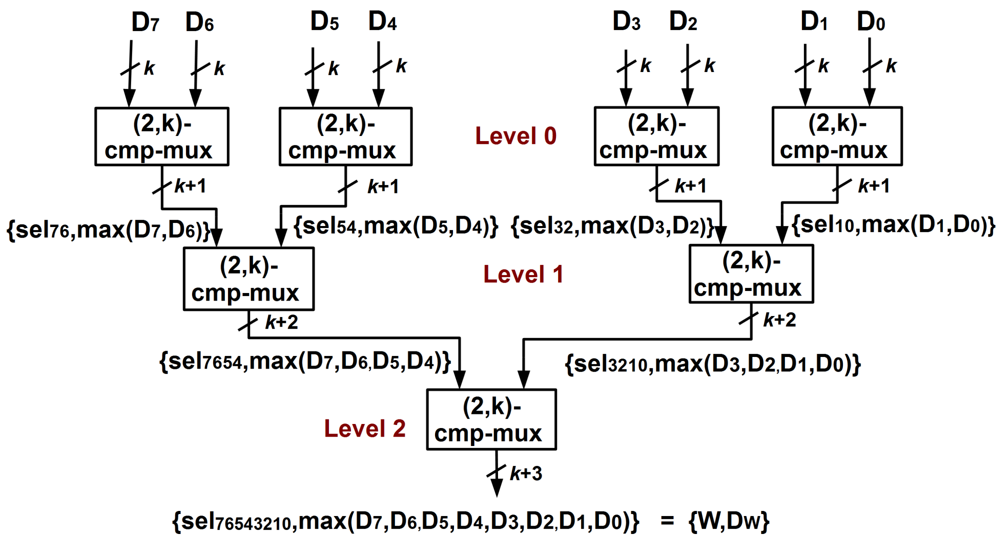


Figure 3.4. Traditional Binary Tree Topology ($n = 8$).

significant ℓ bits of an input signal represent the address of winner element determined by the preceding sub-tree rooted at this input port. And, the remaining k bits of an input signal is the value of the aforementioned winner element. For instance, in Figure 3.4, sel_{7654} is the address of the maximum of elements D_7, D_6, D_5, and D_4, while $max(D_7, D_6, D_5, D_4)$ is the value of the maximum one. Each (2,k)-*cmp-mux* in ℓ-th level of the binary tree consists of a (2,k)-*comparator (cmp)* (denoted as $A>B$ in Figure 3.5) to determine which one of its two inputs is greater, and a $(k + \ell)$-bit 2-to-1 *multiplexer (mux)* to select the greater input. The multiplexer uses the output of the comparator (sel_{AB} in Figure 3.5) to make the selection. Data inputs of the multiplexers in the ℓ-th level of the tree are $(k + \ell)$-bit signals in order to transfer the binary addresses and the values of the maximum elements so far computed to the next level of the binary tree.

In TBT, efficient hardware implementation of (2,k)-*comparators* is crucial to reduce the latency. To determine the greater one of two k-bit binary numbers A and B, we only need to apply a subtraction operation, $(A - B)$ or $(B - A)$, and then to check only the final carry bit of the operation. Hence, we can utilize parallel prefix adders to implement delay- and area-efficient (2,k)-*comparators*, as in [23]. Parallel prefix adders concurrently compute generate (g_i) and propagate (p_i) signals as a preprocessing step,

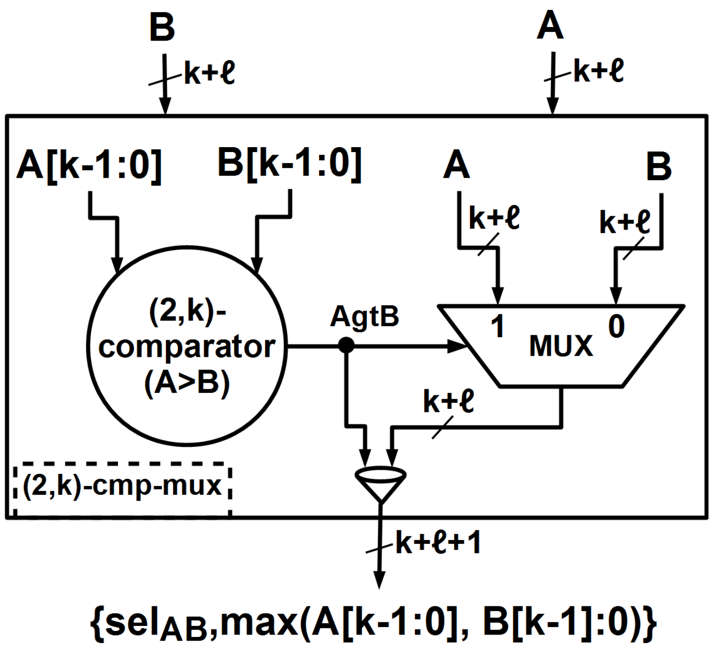
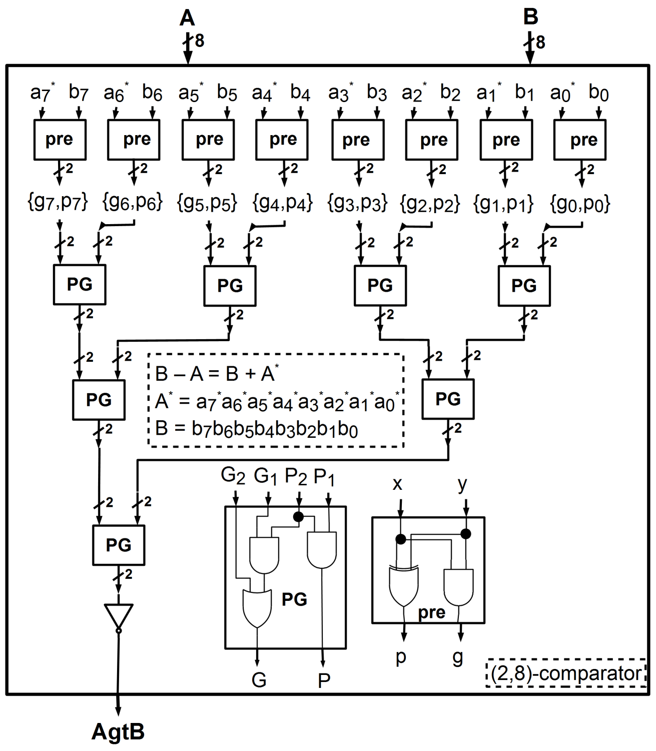


Figure 3.5. Internal structure of a (2,k)-*cmp-mux*.

which takes $O(1)$ time. Then, they compute group generate ($G_{i:j}$) and group propagate ($P_{i:j}$) signals between bit positions i and j using parallel prefix structures, which can compute the final carry in $O(log_2 k)$ time. Finally, they calculate the sum in $O(1)$ time. Since we only need to compute the final carry bit, we do not need to compute the sum and to determine carry bit for each bit position. Thus, we only use preprocessing logic and a binary tree part of the parallel prefix structure, which produces the final carry in order to implement a (2,k)-*comparator* circuit. A (2,8)-*comparator* is shown in Figure 3.6 as an example and as a reference for theoretical complexity calculations in Section 3.4.1.



A
8
B
8
pre
PG
{g7,p7} {g6,p6} {g5,p5} {g4,p4} {g3,p3} {g2,p2} {g1,p1} {g0,p0}
B – A = B + A*
A* = a7*a6*a5*a4*a3*a2*a1*a0*
B = b7b6b5b4b3b2b1b0
G2 G1 P2 P1
G P
x y
p g
(2,8)-comparator
AgtB

Figure 3.6. Internal structure of a (2,k)-*comparator*.

3.4.1. Theoretical Complexity of TBT

Complexity expressions for (2,k)-*comparator*:

$$\begin{aligned}
t_{comparator} &= (\log_2 k) \times t_{PG} + t_{pre} \\
&= (\log_2 k) \times (t_{OR2} + t_{AND2}) + t_{pre} = (\log_2 k) \times (1+1) + 2 \\
&= 2 \times \log_2 k + 2 = O((\log_2 k)) \qquad (3.22)
\end{aligned}$$

$$\begin{aligned}
A_{comparator} &= k \times A_{pre} + (k-1) \times A_{PG} \\
&= k \times (A_{XOR} + A_{AND2}) + (k-1) \times (A_{OR2} + 2 \times A_{AND2}) \\
&= k \times (2+1) + (k-1) \times (1 + 2 \times 1) = 6 \times k - 1 = O(k) \qquad (3.23)
\end{aligned}$$

$$P_{comparator} = A_{comparator} = 6 \times k - 1 = O(k) \qquad (3.24)$$

Complexity expressions for (2,k)-*cmp-mux*:

$$\begin{aligned}
t_{cmp-mux} &= t_{comparator} + t_{mux2} \\
&= (2 \times \log_2 k + 2) + (t_{AND2} + t_{OR2}) = 2 \times \log_2 k + 4 \\
&= O(\log_2 k) \qquad (3.25) \\
A_{cmp-mux} &= A_{comparator} + k \times A_{mux2} \\
&= (6 \times k - 1) + k \times (2 \times A_{AND2} + A_{OR2}) = 6 \times k - 1 + 3 \times k \\
&= 9 \times k - 1 = O(k) \qquad (3.26) \\
P_{cmp-mux} &= A_{cmp-mux} = 9 \times k - 1 = O(k) \qquad (3.27)
\end{aligned}$$

Complexity functions for (n,k)-TBT:

$$\begin{aligned}
t_{TBT} &= \log_2 n \times t_{cmp-mux} = \log_2 n \times (2 \times \log_2 k + 4) \\
&= 2 \times \log_2 n \times \log_2 k + 4 \times \log_2 n = O(\log_2 n \times \log_2 k + \log_2 n) \qquad (3.28) \\
A_{TBT} &= (n-1) \times A_{cmp-mux} + A_{extra} \\
&= (n-1) \times (9 \times k - 1) + (\log_2 n - 1) \times A_{mux2} \\
&= (9 \times k \times n - n - 9 \times k + 1) + (\log_2 n - 1) \times 3 \\
&= O(k \times n) \qquad (3.29) \\
P_{TBT} &= A_{TBT} = O(k \times n) \qquad (3.30)
\end{aligned}$$

The A_{extra} term of Equation 3.29 denotes the area overhead caused by the fact that we convey both the value and the address of the maximum element to the next level of TBT.

3.5. Parallel Binary Tree Topologies (PBTs)

Since the (2,k)-*comparator*s of TBT propagates signals from LSB to MSB, a node ((2,k)-*cmp-mux*) at the level ($\ell + 1$) of TBT cannot start comparison before its two descendant nodes at level ℓ computes all k bits of their results. Authors of [4, 14] propose an MF topology, which we call *Parallel Binary Tree (PBT)*, to reduce the overall propagation delay of TBT by using (2,k)-*cmp-mux* circuits that propagate the signals from MSB to LSB. Using this kind of (2,k)-*cmp-mux* circuits, more significant result bits of a (2,k)-*cmp-mux* circuit in the tree become available to its descendant (2,k)-*cmp-mux* circuit and the descendant can start comparison operation before the end of the comparison in the predecessor. In other words, each level of the binary tree concurrently operates on the different bit positions of its inputs. Hence, we call this topology as PBT to distinguish it from TBT. To illustrate the operation of PBT, a conceptual (8,4)-*PBT* example is given in Figure 3.7. In this conceptual PBT circuit, each node of the tree computes its result bit-by-bit from MSB to LSB. The sequence of the events and propagation of the results through (8,4)-*PBT* of Figure 3.7 can be summarized as follows:

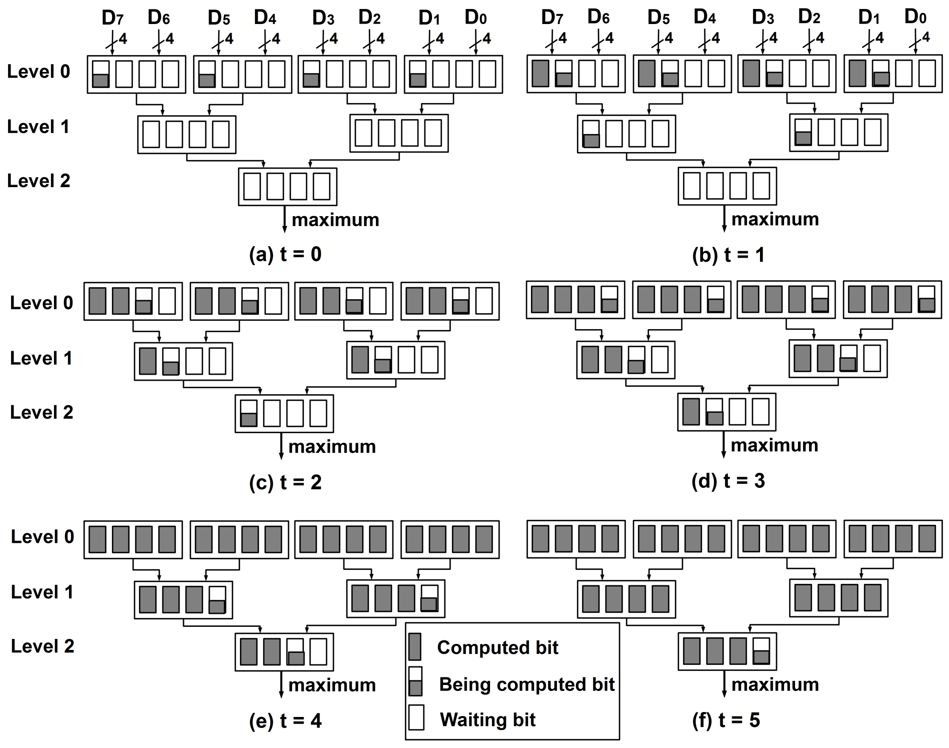


Figure 3.7. An example (8,4)-*PBT* circuit.

- ($t = 0$). (2,4)-*comparator*s at *Level*-0 start to compare MSBs (4th bits) of their corresponding inputs.
- ($t = 1$). The winners of MSB comparisons at *Level*-0 conveyed to corresponding

(2,4)-*comparator*s at *Level*-1. At the same time, (2,4)-*comparator*s at *Level*-0 start to compare 3rd bits of their inputs.

- ($t = 2$). (2,4)-*comparator*s at *Level*-0 start to compare 2nd bits of their inputs and (2,4)-*comparator*s at *Level*-1 start to compare 3rd bits of their inputs. Concurrently, (2,4)-*comparator*s at *Level*-2 start to compare MSBs of their inputs.
- ($t = 3$). (2,4)-*comparator*s at *Level*-0 start to compare LSBs (1st bits) of their inputs and (2,4)-*comparator*s at *Level*-1 start to compare 2nd bits of their inputs. Concurrently, (2,4)-*comparator*s at *Level*-2 start to compare 3rd bits of their inputs.
- ($t = 4$). (2,4)-*comparator*s at *Level*-0 complete their jobs. (2,4)-*comparator*s at *Level*-1 start to compare 1st bits of their inputs. Concurrently, (2,4)-*comparator*s at *Level*-2 start to compare 2nd bits of their inputs.
- ($t = 5$). (2,4)-*comparator*s at *Level*-1 finish their jobs. Concurrently, (2,4)-*comparator*s at *Level*-2 start to compare 1st bits of their inputs. And all bits of the result will be available at $t = 6$.

In [4, 14], three PBT variants are proposed, each of which utilizes a different well-known adder topology: *Ripple Carry parallel binary tree Topology* (RCT), *Carry Lookahead parallel binary tree Topology* (CLT), and *Carry Select parallel binary tree Topology* (CST). They define a 2-bit *carry* signal ($\{choose, found\}$) propagating from MSB to LSB. The *found* bit of the carry signal at bit-position i indicates if the winner is determined so far by comparators between MSB and the bit-position i. The *choose* bit of the carry signal at bit-position i indicates which one of the inputs is the winner. For instance, a (2,k)-*cmp-mux* of RCT consists of k cascaded (2,1)-*RC-cells* (a kind of (2,1)-*cmp-mux*). Each RC-cell at bit-position i computes the carry input of bit-position $i - 1$ and *select* signal for the multiplexer that selects bit i of the result. We shall examine each of the three variants in a separate subsection (in Sections 3.5.1 –3.5.3) to show more clearly the differences between them.

3.5.1. Ripple Carry Parallel Binary Tree Topology (RCT)

In RCT, each node of the tree is a (2,k)-*RC-cell*, which is functionally equivalent to a (2,k)-*cmp-mux* and consists of serially connected k (2,1)-*RC-cell*s as shown in Figure 3.8. A (2,1)-*RC-cell* at the bit position i produces bit i of the winner element (m_i in Figure 3.8) as well as carry signal for the bit position $i-1$. The (2,1)-*RC-cell*

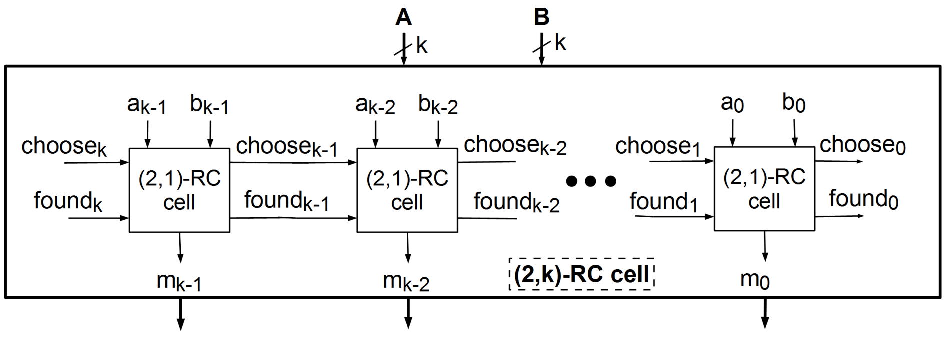


Figure 3.8. A node ((2,k)-*RC-cell*) of Ripple Carry PBT [4].

at the bit position i computes its outputs using the following equations [4]:

$$choose_i = choose_{i+1} + \overline{found_{i+1}}.(a_i.\overline{b_i}) \tag{3.31}$$

$$found_i = found_{i+1} + (a_i \oplus b_i) \tag{3.32}$$

$$m_i = \overline{choose_i}.b_i + choose_i.a_i \tag{3.33}$$

3.5.2. Carry Select Parallel Binary Tree Topology (CST)

In a (2,k)-*RC-cell*, each (2,1)-*RC-cell* has to wait for an incoming carry signal to compute its result bit. To reduce this linear dependency, CST adapts the idea of carry select adders to the comparison operation, and uses (2,k)-*CS-cell*s (see Figure 3.9) at the nodes of PBT. Internal structure of the (2,k)-*CS-cell* is similar to that of carry select adders. A (2,k)-*CS-cell* consists of cascaded (2,p_i)-*CS-cell*s. A (2,p_i)-*CS-cell* consists of three blocks: A (2,p_i)-*ripple-block*, (2,p_i)-*mux-block*, and a (2,p_i)-*CS-block*. The ripple-block is the carry generator part of the RC-cell and the mux-block is a p_i

bit 2-to-1 multiplexer. The CS-block takes carry signals from the previous CS-cell as well as from the current ripple-block, and then computes select signals of the current mux-block besides carry signals for the next CS-cell. At the beginning, each (2,p_i)-*CS-cells* starts to compute corresponding p_i bits of the winner as if higher order bits of the inputs were equal and then the CS-block selects the correct result when the true value of the incoming carry signal is determined.

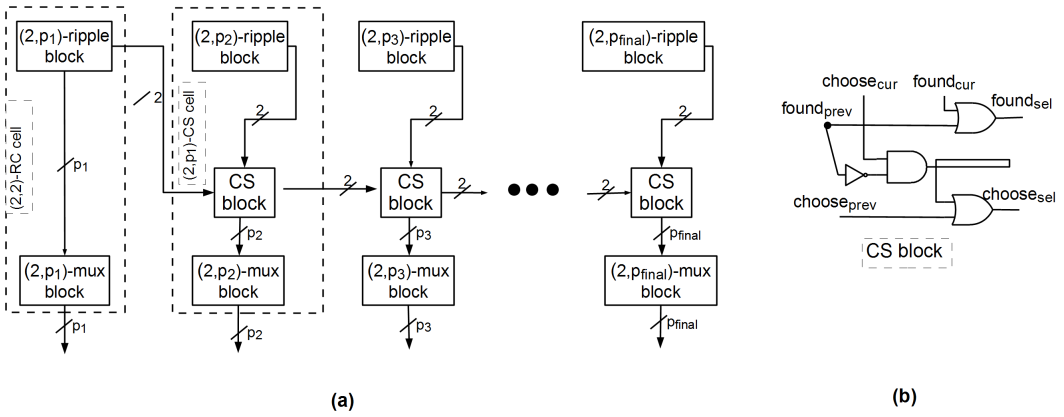


Figure 3.9. (a) Linear configuration of a (2,k)-*CS-cell* [4]. (b) Internal structure of the CS-block.

In CST, a mismatch between the arrival time of the input carry signals and preparation time of the out carry signals: The closer a (2,p_i)-*CS-cell* to LSB is, the earlier it prepares the output carry signal and the later its input carry arrives. In order to get around the mismatch, a square–root configuration was designed in [4]. In the square–root configuration the p_i values are chosen as $\{p_1 = \sqrt{2k}$, $p_2 = p_1 + 1, \ldots,$ $p_{final} = p_{final-1} + 1\ \}$ to obtain a timing complexity of $O(\sqrt{k})$. We shall use this version in Chapter 6 while we are obtaining synthesis results. In Figure 3.10, the authors also introduces a new type of block, namely *half carry-select block (HCS-block)*, which is a simplified version of CS-block. The purpose of HCS-block is reducing the fanout on the critical path of (2,k)-*CS-cell*. Further details can be found in [4].

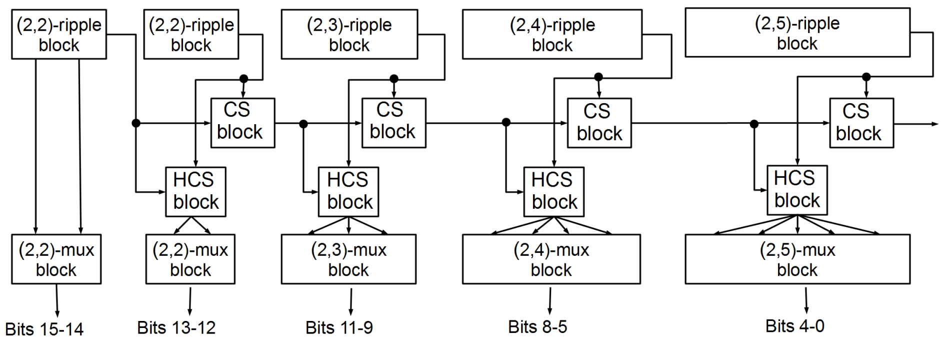


Figure 3.10. Square-root configuration of a (2,k)-*CS-cell* [4].

3.5.3. Carry Look-ahead Parallel Binary Tree Topology (CLT)

CLT eliminates the ripple effect in RCT and CST using the following Boolean equations:

$$gc_i = a_i.\overline{b_i} \tag{3.34}$$

$$gf_i = a_i \oplus b_i \tag{3.35}$$

$$choose_i = \sum_{x=k-1}^{i} \left(\prod_{y=x+1}^{i+1} \overline{gf_y} \right) .gc_x \tag{3.36}$$

$$found_i = \sum_{x=k-1}^{i} gf_x \tag{3.37}$$

Using this equations, carry signals for each bit position can be computed as soon as the inputs A and B becomes available to the corresponding node of the PBT.

Although it was not mentioned in [4, 14], we can use parallel prefix networks (PPNs) to compute $choose_i$ and $found_i$ signals in Equations 3.36 and 3.37. Hence, we also implemented four different PPN variants of CLT, each of which uses a different PPN topology: *Brent-Kung PBT (BKT)* [15], *Kogge-Stone PBT (KST)* [17], *Han-Carlson PBT (HCT)* [16], and *Ladner-Fischer PBT (LFT)* [18]. Note that we shall name the MF topology, which directly implements the Equations 3.34 –3.37 in Verilog, as *Carry-lookahead Equations Topology (CET)*. In Figure 3.11, we show the PPNs for

$n = 8$ as an example.

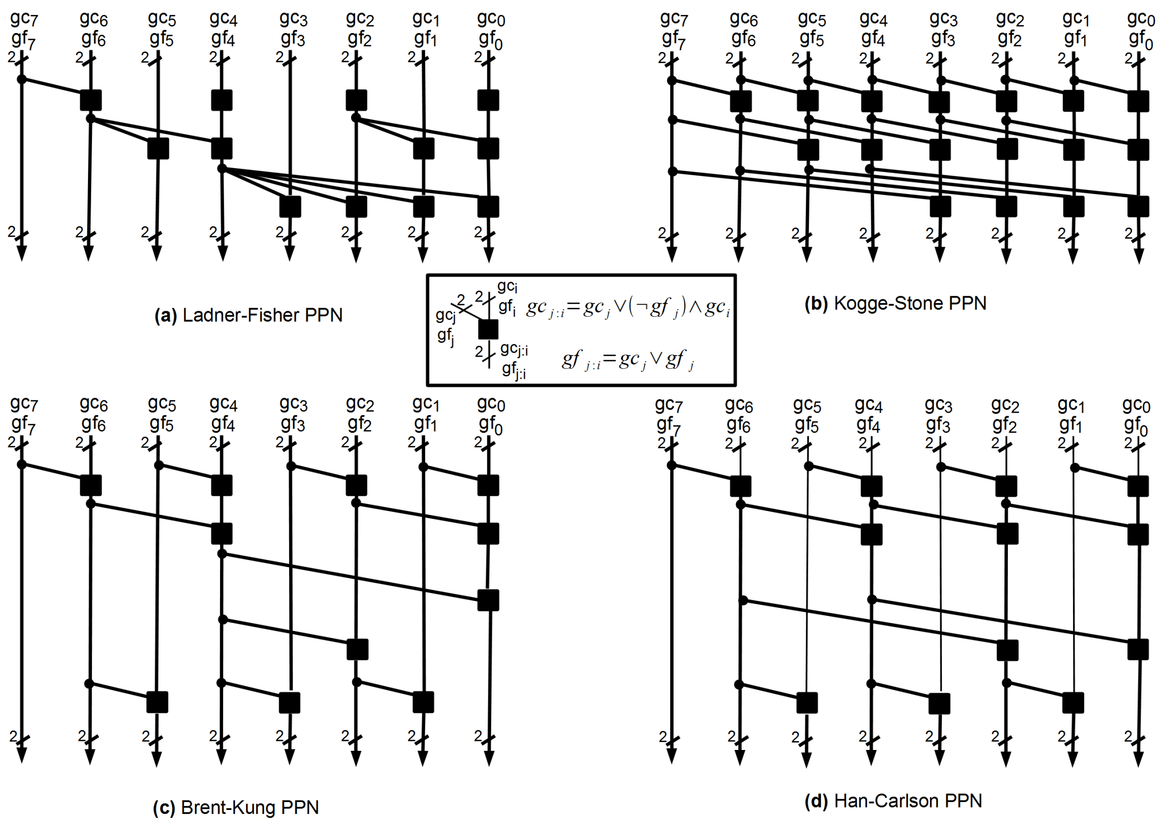


Figure 3.11. Four baseline PPN topologies for CLT: (a) Ladner-Fisher (LF). (b) Kogge-Stone (KS). (c) Brent-Kung (BK). (d) Han-Carlson (HC).

3.5.4. Theoretical Complexity of PBTs

In Table 3.1, we give the theoretical complexities of seven different PBT variants: RCT, CST, CLT, LFT, BKT, KST, and HCT. As the detailed analysis of RCT, CST, and CLT can be found in [4], we do not give detailed complexity equations for them. Since area and power consumption are the same according to our complexity models explained in Section 2.2, we do not list the power consumption complexity functions in Table 3.1.

Table 3.1. Theoretical complexity expressions of PBTs.

Topology	**Timing Complexity**	**Area Complexity**
RCT	$O(\log_2 n + k)$	$O(n \times k)$
CST	$O(\sqrt{k}) + O((\sqrt{k} - \log_2 n) \times log_2 n)$	$O(n \times k)$
CET	$O(\log_2 n \times \log_2 k)$	$O(n \times k)$
LFT	$O(\log_2 n \times \log_2 k)$	$O(n \times k \times \log_2 k)$
BKT	$O(\log_2 n \times \log_2 k)$	$O(n \times k)$
KST	$O(\log_2 n \times \log_2 k)$	$O(n \times k \times \log_2 k)$
HCT	$O(\log_2 n \times \log_2 k)$	$O(n \times k \times \log_2 k)$

3.6. Leading-zero Counting Topology (LCT)

Leading-zero Counting Topology (LCT) [5] consists of two cascade stages. In the first stage, value of the winner (i.e. maximum) element is determined. And in the second stage, binary address of the winner is computed. In the first stage, each of the n elements is converted into a one-hot code and then a 2^k-bit vector B is obtained by applying OR operation on the one-hot elements. Finally, the k-bit value of the winner element is computed using an a leading-zero counting (LZC) algorithm (a priority encoder). In the second stage, each element is first compared with the winner element to obtain n-bit vector D. Finally, the LZC is used to determine $\log_2 n$-bit address of the winner element from D. As a result, LCT outputs both value and binary address of the winner. In Figure 3.12, we present a $(8,k)$-LCT circuit as an example.

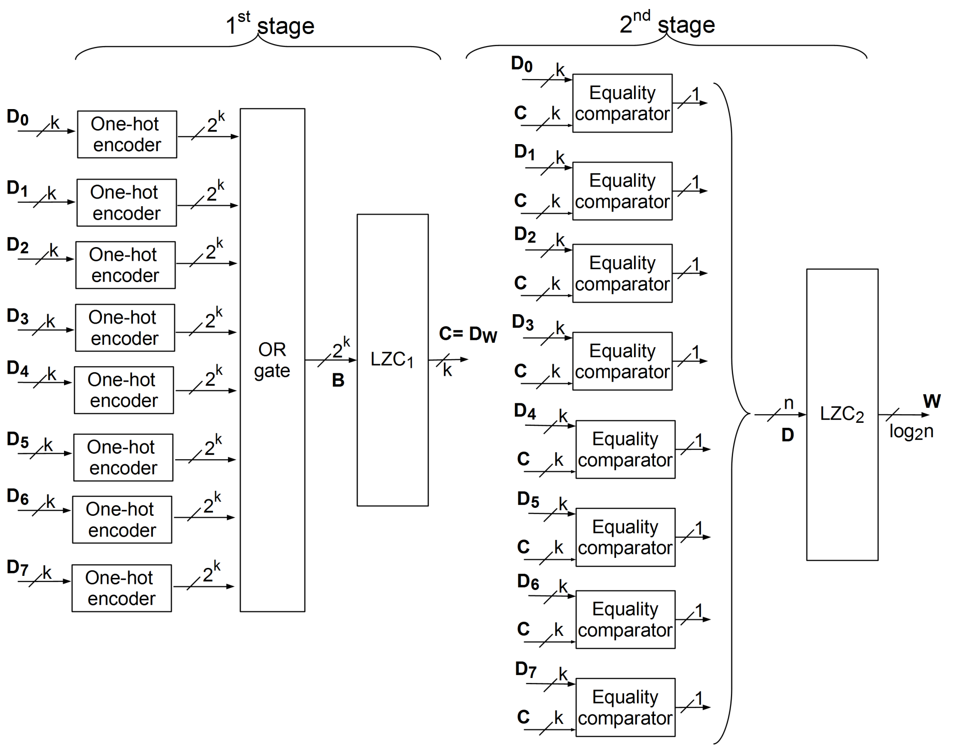


Figure 3.12. Leading-zero Counting Topology [5] ($n = 8$).

3.6.1. Theoretical Complexity of LCT

Complexity formulas for a (n,k)-LCT :

$$\begin{aligned}
t_{LCT} &= t_{one-hot} + t_{ORn} + t_{LZC_1} + t_{Equality-comparator} + t_{LZC_2} \\
&= (k+1) + 2 \times \log_2 k + 2 \times \log_2 n + 3 = O(k + \log_2 k + \log_2 n) \qquad (3.38)
\end{aligned}$$

$$\begin{aligned}
A_{LCT} &= A_{one-hot} + A_{ORn} + A_{LZC_1} + A_{Equality-comparator} + A_{LZC_2} \\
&= (2^k \times (k-1)) + (2^k \times (k-1)) + (\frac{2^k}{2} \times k) \\
&+ (k \times 2 + k - 1) + (\frac{n}{2} \times \log_2 n) \\
&= 2^k \times (\frac{3 \times k}{2} + n - 2) + 2k + 1 + \frac{n}{2} \times \log_2 n \\
&+ (\frac{n^2 - 2 \times n + 8}{8}) + (\frac{n-2}{2}) \times k \times 3 \\
&= O(2^k \times (n+k)) \qquad (3.39)
\end{aligned}$$

$$P_{LCT} = A_{LCT} = O(2^k \times (n+k) \qquad (3.40)$$

3.7. Multi-Level Topology (MLT)

In Multi-Level Topology (MLT) [5], inputs are first divided into 2-element groups and the winner of each group is determined by a (2,k)-*cmp-mux* circuit. Then, a parallel comparison (i.e., every pair is compared each other) is applied between groups using (2,k)-*comparator*s. Finally, the value of the winner is transferred to the output through a serial configuration of the multiplexers. A (8,k)-*MLT* is given in Figure 3.13 as an example. Second stage of MLT topology has the same architecture as *comparator*

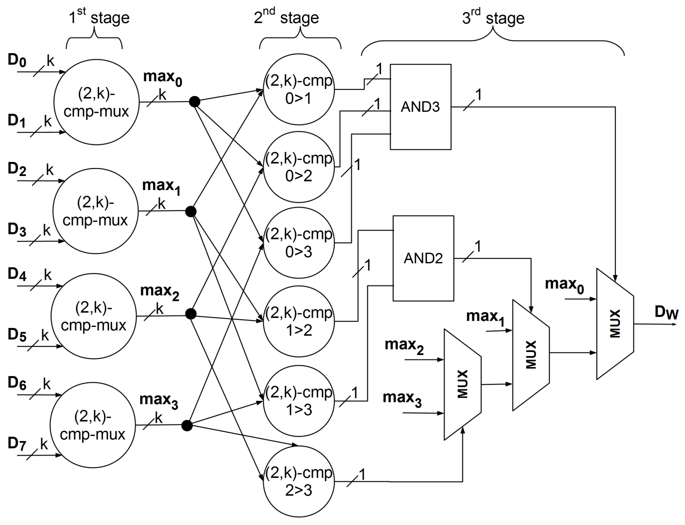


Figure 3.13. Multi-Level Topology [5] (n = 8).

array unit of our AbT, which we shall explain in Section 4.1. However, MLT does not utilize the concurrency between comparison and selection operations (3^{rd} stage of MLT) in contrast to our approach. Also, MLT outputs only the value of the winner element. We can serially connect the second stage of LCT to the output of MLT to obtain the binary address of the maximum element.

3.7.1. Theoretical Complexity of MLT

Complexity formulas for a (n,k)-MLT :

$$\begin{aligned}
t_{MLT} &= t_{1^{st}stage} + t_{2^{nd}stage} + t_{3^{rd}stage} \\
&= t_{cmp-mux} + t_{comparator} + (\frac{n}{2} - 1) \times t_{mux2} \\
&= (\log_2 k + 2) + \log_2 k + (\frac{n}{2} - 1) \times 2 \\
&= \frac{n}{2} + 2 \times \log_2 k = O(n + \log_2 k) \qquad (3.41)
\end{aligned}$$

$$\begin{aligned}
A_{MLT} &= A_{1^{st}stage} + A_{2^{nd}stage} + A_{3^{rd}stage} \\
&= \frac{n}{2} \times A_{cmp-mux} + (\frac{n^2 - 2 \times n}{8}) \times A_{comparator} \\
&+ (\frac{n^2 - 2 \times n + 8}{8}) \times A_{AND2} + (\frac{n-2}{2}) \times k \times A_{mux2} \\
&= \frac{n}{2} \times (9 \times k - 1) + (\frac{n^2 - 2 \times n}{8}) \times (6 \times k - 1) \\
&+ (\frac{n^2 - 2 \times n + 8}{8}) + (\frac{n-2}{2}) \times k \times 3 \\
&= O(k \times n^2) \qquad (3.42)
\end{aligned}$$

$$P_{MLT} = A_{MLT} = O(k \times n^2) \qquad (3.43)$$

4. PROPOSED MF CIRCUIT TOPOLOGIES

In this chapter, we present the details of our proposed topologies: *Array based Topology (AbT)*, *Hybrid Binary tree Topology (HBT)*, and *Quad tree Topology (QT)*.

4.1. Array Based Topology (AbT)

AbT is a fast MF circuit topology, where the main idea is first producing an $n \times n$ array of 1-bit comparison results by comparing every pair of input elements in parallel, and then concurrently generating the address as well as the value of the maximum element via output logic. Carrying out all possible comparisons in parallel allows us to avoid $\log_2 n$ stages of the binary tree topologies, and thus, AbT ends up with just one comparator on the critical path followed by some selection logic.

The conceptual block diagram of AbT is given in Figure 4.1. AbT consists of three blocks in the top-level: *Comparator Array Unit* (CAU), *Address Generation Unit* (AGU), and *Data Selection Unit* (DSU). CAU computes all magnitude relations

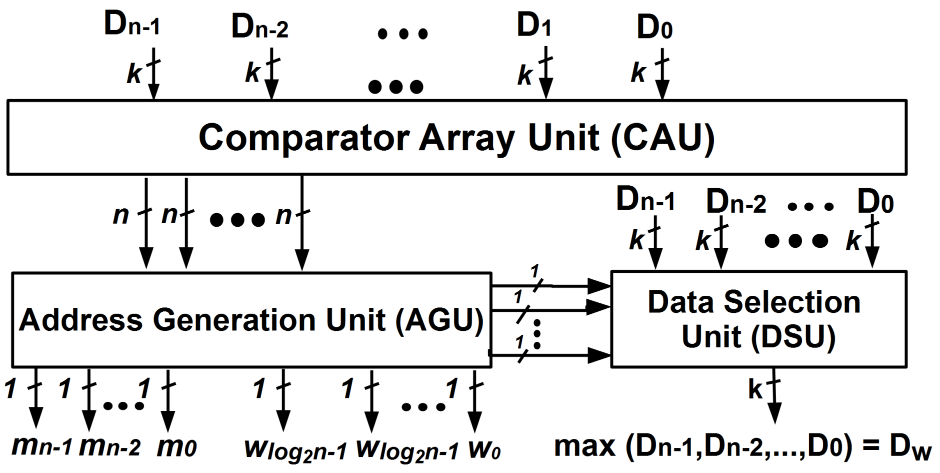

D n-1
D n-2
D 1
D 0
k
Comparator Array Unit (CAU)
n
Address Generation Unit (AGU)
Data Selection Unit (DSU)
1
m n-1
m n-2
m 0
w log2n-1
w 0
max (D n-1, D n-2,...,D 0) = D w

Figure 4.1. High-level block diagram of AbT.

in parallel, which are necessary to compute one-hot address of the maximum. AGU rapidly generates one-hot and/or binary address of the maximum by processing the results of CAU. DSU operates concurrently with AGU and conveys the maximum element to the output. DSU uses the signals produced by intermediate stages of AGU, and hence, it does not bring any (theoretical) delay overhead on AGU.

Detailed structure of AbT will be described in the following subsections.

4.1.1. Comparator Array Unit (CAU)

CAU can be thought of as a preprocessing unit that constructs an $n \times n$ array (see Figure 4.2) of 1-bit comparison results g_{xy} ($g_{xy} = D_y > D_x$, where $\forall x, y \in [0, n-1]$) by concurrently computing all required magnitude relations between the inputs (D_i) to produce the one-hot address of maximum. Note that x and y denote the corresponding column and row of the array, respectively, throughout this subsection.

In order to determine if a specific element D_i is the maximum (D_W) of the n elements, we need to compare the element D_i with all of the other elements (D_j, where $j \neq i$) and thus we need $(n-1)$ comparison operations. For instance, the following logic equation gives us if D_0 is the maximum element of our input set $\mathbf{D}$.

$$\begin{aligned} m_0 &= (D_0 \geq D_1) \cdot (D_0 \geq D_2) \cdot (D_0 \geq D_3) \cdot \ldots \cdot (D_0 \geq D_{n-1}) \\ &= \overline{(D_1 < D_0) + (D_2 < D_0) + (D_2 < D_0) + \ldots (D_{n-1} < D_0)} \end{aligned} \tag{4.1}$$

If we apply the same operations to each element of the set $\mathbf{D}$, we obtain Equation 4.2. In Equation 4.2, each bit m_i specifies whether the element D_i has the maximum value or not. In regards to Equation 4.2, we need $n \times (n-1)$ comparison operations to determine the position(s) of the maximum element(s). We can do all of these operations in parallel as there is no data dependency between the required comparison operations.

However, Equation 4.2 cannot handle the cases in which more than one element has the maximum value. In such cases, the computed n-bit vector $(m_{n-1}m_{n-2} \ldots m_1m_0)$ shall not be one-hot. Thus, we need a priority encoder circuit to obtain the one-hot address of the maximum element. This priority encoder brings an extra area and delay overhead to the AGU. Fortunately, this problem can be solved by embedding the priority information into Equation 4.2. As a result, we obtain Equation 4.3.

$$\begin{aligned}
m_0 &= (D_0 \geq D_1) \cdot (D_0 \geq D_2) \cdot (D_0 \geq D_3) \cdot \ldots \cdot (D_0 \geq D_{n-1}) \\
&= \overline{(D_0 < D_1) + (D_0 < D_2) + (D_0 < D_3) + \ldots + (D_0 < D_{n-1})} \\
m_1 &= (D_1 \geq D_0) \cdot (D_1 \geq D_2) \cdot (D_1 \geq D_3) \cdot \ldots \cdot (D_1 \geq D_{n-1}) \\
&= \overline{(D_1 < D_0) + (D_1 < D_2) + (D_1 < D_3) + \ldots + (D_1 < D_{n-1})} \\
&\vdots \\
m_{n-1} &= (D_{n-1} \geq D_0) \cdot (D_{n-1} \geq D_2) \cdot (D_{n-1} \geq D_3) \cdot \ldots \cdot (D_{n-1} \geq D_{n-2}) \\
&= \overline{(D_{n-1} < D_0) + (D_{n-1} < D_2) + (D_{n-1} < D_3) + \ldots + (D_{n-1} < D_{n-2})}
\end{aligned} \tag{4.2}$$

$$\begin{aligned}
m_0 &= (D_0 > D_1) \cdot (D_0 > D_2) \cdot (D_0 > D_3) \cdot \ldots \cdot (D_0 > D_{n-1}) \\
&= \overline{(D_0 \leq D_1) + (D_0 \leq D_2) + (D_0 \leq D_3) + \ldots + (D_0 \leq D_{n-1})} \\
&= \overline{\sum_{y=1}^{n-1} (D_0 \leq D_y)} \\
m_1 &= (D_1 \geq D_0) \cdot (D_1 > D_2) \cdot (D_1 > D_3) \cdot \ldots \cdot (D_1 > D_{n-1}) \\
&= \overline{(D_1 < D_0) + (D_1 \leq D_2) + (D_1 \leq D_3) + \ldots + (D_1 \leq D_{n-1})} \\
&= \overline{(D_1 < D_0) + \sum_{y=2}^{n-1} (D_0 \leq D_y)} \\
&\vdots \\
m_{n-1} &= (D_{n-1} \geq D_0) \cdot (D_{n-1} \geq D_2) \cdot (D_{n-1} \geq D_3) \cdot \ldots \cdot (D_{n-1} \geq D_{n-2}) \\
&= \overline{(D_{n-1} < D_0) + (D_{n-1} < D_1) + (D_{n-1} < D_2) + \ldots + (D_{n-1} < D_{n-2})} \\
&= \overline{\sum_{y=0}^{n-2} (D_2 < D_y)}
\end{aligned} \tag{4.3}$$

We can also rewrite Equation 4.3 as in Equation 4.4:

$$\begin{aligned} m_x &= \overline{\left(\sum_{y=0}^{x-1}(D_x < D_y)\right) + \left(\sum_{y=x+1}^{n-1}(D_x \leq D_y)\right)} \\ &= \overline{\left(\sum_{y=0}^{x-1}(g_{xy})\right) + \left(\sum_{y=x+1}^{n-1}(\overline{g_{yx}})\right)} \end{aligned} \tag{4.4}$$

In this embedded priority scheme, the greater the address of an element D_i is, the greater its priority is. Using this priority scheme, we break the tie when multiple inputs possess the maximum value, and we obtain a one-hot n-bit address vector $(m_{n-1}m_{n-2}\dots m_1m_0)$ without any delay and area overhead. The priority scheme also leads to an area optimization in CAU, which shall be clear in the following paragraphs.

As seen in Equation 4.4, in order to find any m_x, we first compute n magnitude relations and then apply a NOR operation on the computed relations. Hence, we should compute n^2 magnitude relations and should apply n NOR operations to obtain the whole one-hot address. The required relations are concurrently computed by CAU, while NOR operations are applied by AGU.

Fortunately, we can reduce the number of (2,k)-comparators required to obtain g_{xy} values. First, we do not need to compute g_{xy} values on the diagonal of the array because they are all zero. Second, we can compute the lower triangular part $(x < y)$ of the array by transposing and inverting the upper triangular part ($x > y$) owing to our priority scheme. Hence, we can find g_{xy} value residing in column x and row y of the array using Equation 4.5.

$$g_{xy} = \begin{cases} 0 & if\ y = x, \\ D_y > D_x & if\ y < x \quad \forall x, y \in [0, n-1] \\ \overline{g_{yx}} & if\ y > x \end{cases} \tag{4.5}$$

Based on Equation 4.5, we compute $n\,(n-1)/2$ g_{xy} values within the upper triangular part with $n\,(n-1)/2$ (2,k)-*comparator*s. Then, we use $n\,(n-1)/2$ inverters to compute $n\,(n-1)/2$ g_{xy} values within the lower triangular part, as illustrated in Figure 4.2. In this figure, we use gray dotted lines to show the array cells computed by inverters, while we use gray solid lines for inverters and their connections. On the other hand, we use black solid lines for the cells computed directly by (2,k)-*comparator*s. Each (2,k)-*comparator* of Figure 4.2 determines whether its first input (D_y) is greater than its second input (D_x) and produces a 1-bit output ($g_x y$). We utilize PPN adder topologies to implement (2,k)-*comparator*s of CAU as mentioned in Section 3.4. As a result, CAU can compute the required n^2 relations, in Equation 4.5, in $O(\log_2 k)$ time.

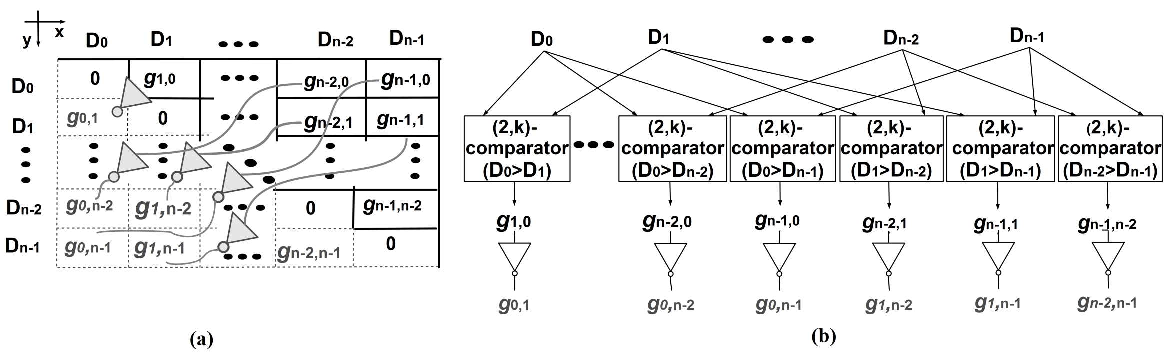


Figure 4.2. Block diagram of CAU: (a) Conceptual array view. (b) Interconnection of CAU blocks.

4.1.2. Address Generation Unit (AGU)

AGU computes one-hot and/or binary address of the maximum element using $n\,(n-1)$ magnitude relations computed by CAU.

AGU uses an n-input NOR gate to compute each bit of one-hot address and an encoder circuit to compute binary address, as shown in 4.3b. The n-element NOR gate, which produces m_x of Equation 4.4, uses (n–1) g_{xy} values within the column x of CAU. We implement an n-input NOR gate as a binary tree of 2-input OR gates with a final inverter. Similarly, we implement the encoder circuit as a binary tree of 2-input OR gates. Hence, AGU can generate one-hot and/or binary address of the maximum element in $O(\log_2 n)$ time. CAU and AGU are connected to each other as illustrated in 4.3a.

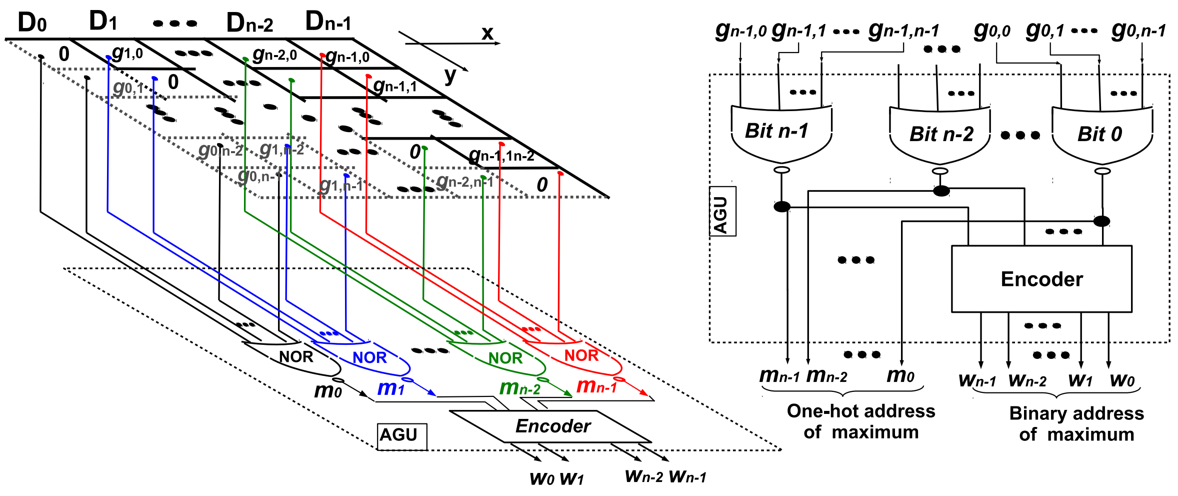
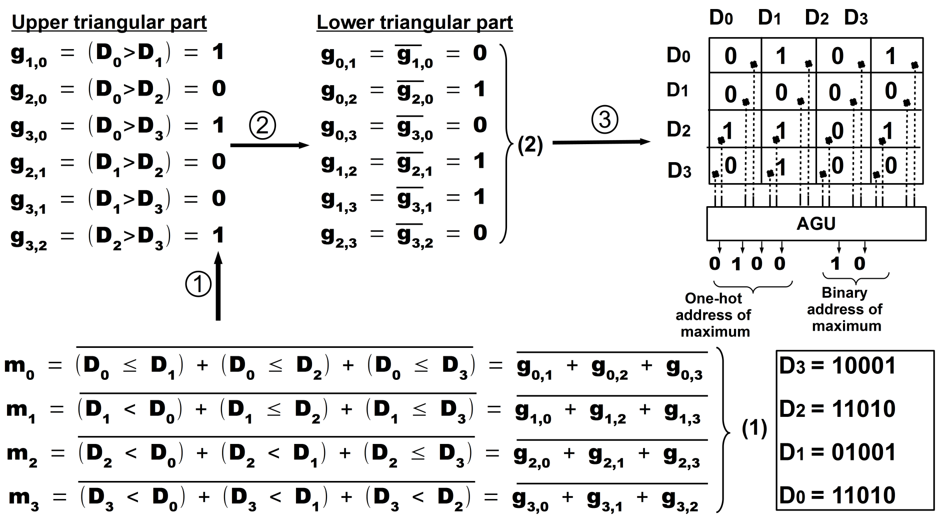


Figure 4.3. (a) Interconnection of CAU and AGU. (b) Block diagram of AGU.

The following example, which is also shown in Figure 4.4 is useful to understand how CAU and AGU function together. Let $\mathbf{D}$ be $\{D_3, D_2, D_1, D_0\} = \{10001, 11010, 01001, 11010\}$. In this case, we have $n = 4$ and $k = 5$. In Figure 4.4, we explicitly give equation sets obtained from Equations 4.3 and 4.4. As it can be seen in the figure, we first compute each row of the upper triangular part of the array via (2,5)-comparators. Then, we compute the values within the lower triangular part of the array using only inverters. In this example, the maximum element is D_2, and our (4,5)-AbT example outputs one-hot address as "0100" and binary address as "10" in Figure 4.4.

4.1.3. Data Selection Unit (DSU)

DSU is a binary tree of k-bit 2-to-1 multiplexers and operates concurrently with AGU. Ideally, DSU does not cause any extra delay overhead to AGU. This can be proven by a simple timing analysis: In order to determine the *select* input of a 2-to-1 multiplexer at level j of the DSU, a 2^{j-1}-input NAND gate is needed. Each input of



Upper triangular part
$g_{1,0} = (D_0>D_1) = 1$
$g_{2,0} = (D_0>D_2) = 0$
$g_{3,0} = (D_0>D_3) = 1$
$g_{2,1} = (D_1>D_2) = 0$
$g_{3,1} = (D_1>D_3) = 0$
$g_{3,2} = (D_2>D_3) = 1$
Lower triangular part
$g_{0,1} = \overline{g_{1,0}} = 0$
$g_{0,2} = \overline{g_{2,0}} = 1$
$g_{0,3} = \overline{g_{3,0}} = 0$
$g_{1,2} = \overline{g_{2,1}} = 1$
$g_{1,3} = \overline{g_{3,1}} = 1$
$g_{2,3} = \overline{g_{3,2}} = 0$
①
②
③
(2)
(1)
D0 D1 D2 D3
AGU
0 1 0 0
1 0
One-hot address of maximum
Binary address of maximum
$m_0 = \overline{(D_0 \le D_1) + (D_0 \le D_2) + (D_0 \le D_3)} = \overline{g_{0,1} + g_{0,2} + g_{0,3}}$
$m_1 = \overline{(D_1 < D_0) + (D_1 \le D_2) + (D_1 \le D_3)} = \overline{g_{1,0} + g_{1,2} + g_{1,3}}$
$m_2 = \overline{(D_2 < D_0) + (D_2 < D_1) + (D_2 \le D_3)} = \overline{g_{2,0} + g_{2,1} + g_{2,3}}$
$m_3 = \overline{(D_3 < D_0) + (D_3 < D_1) + (D_3 < D_2)} = \overline{g_{3,0} + g_{3,1} + g_{3,2}}$
D3 = 10001
D2 = 11010
D1 = 01001
D0 = 11010

	D0	D1	D2	D3
D0	0	1	0	1
D1	0	0	0	0
D2	1	1	0	1
D3	0	1	0	0

Figure 4.4. An example for AbT without DSU ($n = 4$, $k = 5$).

this NAND gate comes from the level j–1 of associated OR trees of AGU. Assuming that all g_{xy} values of CAU is ready at the time ($t = 0$), arrival times for *data* (t_{data}) and *select* (t_{select}) input signal of the a 2-to-1 multiplexer at the level j of DSU are given in Equations 4.6 and 4.7.

$$t_{select} = t_{ORtree} + t_{NAND} = \log_2 2^{j-1} + (j-1) = 2 \times j - 2 \quad (4.6)$$

$$t_{data} = (j-1) \times t_{mux} = 2 \times j - 2 \quad (4.7)$$

Equations 4.6 and 4.7 prove that *select* and *data* signals arrive at the same time. An example for $n = 8$ is given in Figure 4.5. In Figure 4.5, OR trees of AGU are shown as gray solid lines. Also, *data* and *select* signals are shown as black solid and dotted lines, respectively. Arrival times of these signals are also given in Figure 4.5.

In reality, however, the situation is a little different. Adding DSU increases the fanout where it taps into AGU, causing some extra delay overhead to AGU. We shall examine the delay overhead of DSU in Section 6.2.1 with an empirical approach.

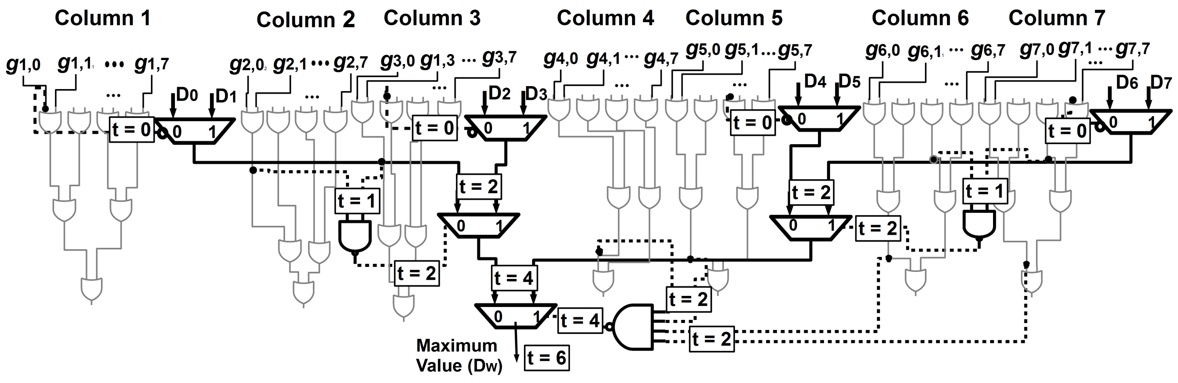


Figure 4.5. Data selection unit (DSU) with OR trees of AGU.

4.1.4. Theoretical Complexity of AbT

We can estimate the theoretical complexity of an (n,k)-CAU as follows:

$$\begin{aligned} t_{CAU} &= t_{comparator} \\ &= (2 \times \log_2 k + 2) = O(\log_2 k) \quad (4.8) \\ A_{CAU} &= \frac{n \times (n-1)}{2} \times A_{comparator} \\ &= \frac{n \times (n-1)}{2} \times (6 \times k - 1) = O(k \times n^2) \quad (4.9) \\ P_{CAU} &= A_{CAU} = O(k \times n^2) \quad (4.10) \end{aligned}$$

Complexity formulas for a (n,k)-AGU:

$$\begin{aligned} t_{AGU} &= t_{NORn} \\ &= \log_2 n = O(\log_2 n) \quad (4.11) \\ A_{AGU} &= n \times + A_{NORn} \\ &= n \times (n-1) = O(n^2) \quad (4.12) \\ P_{AGU} &= A_{AGU} = O(n^2) \quad (4.13) \end{aligned}$$

We can write the following formulas that compute the theoretical complexity of (n,k)-DSU:

$$\begin{aligned}
t_{DSU} &= \log_2 n \times t_{mux2} \\
&= \log_2 n \times (t_{OR2} + t_{AND2}) \\
&= \log_2 n \times 2 = O(\log_2 n) \qquad (4.14) \\
A_{DSU} &= A_{MUXes} + A_{NANDs} \\
&= ((n-1) \times A_{mux2}) + \left(\sum_{l=1}^{\log_2 n} (l-1) \times \frac{n}{2^l} \right) \\
&= ((n-1) \times 3 \times k) + \left(\frac{(\log_2 n - 1) \times \log_2 n}{2} \right) \\
&= O(k \times n) \qquad (4.15) \\
P_{DSU} &= A_{DSU} = O(k \times n) \qquad (4.16)
\end{aligned}$$

Using the theoretical complexity formulas above, which we wrote for CAU, AGU and DSU, we obtain the following theoretical complexity values for AbT:

$$\begin{aligned}
t_{AbT} &= t_{CAU} + max\,(t_{AGU}, t_{DSU}) \\
&= O(\log_2 k) + O(\log_2 k) = O(\log_2 k + \log_2 n) \qquad (4.17) \\
A_{AbT} &= A_{CAU} + A_{AGU} + A_{DSU} \\
&= O(k \times n^2) + O(n^2) + O(k \times n) \\
&= O(k \times n^2) \qquad (4.18) \\
P_{AbT} &= A_{AbT} = O(k \times n^2) \qquad (4.19)
\end{aligned}$$

4.1.5. Other Useful Features of AbT

In this subsection, we shall talk about some possible applications, in which AbT can be used, and also make a generalization for AbT.

First of all, we can use AbT to solve the selection problem, which can be defined as finding the i-th smallest element (i-th order statistic) of a set $\mathbf{D}$ of n binary numbers [24]. For instance, the maximum element (D_W) is the n-th order statistic ($i = n$) of the set $\mathbf{D}$. AGU naturally determines the rank of input elements. The number of ones in column j of CAU gives the rank of input D_j. For example, in Figure 4.4 columns 0, 1, 2, and 3 contain 1, 3, 0, and 2 ones, respectively. This implies the following ranking: D2, D0, D3, D1. Hence, we can compute the i-th order statistic of the set $\mathbf{D}$ in four steps: First, producing the CAU results, and then computing the sum of the elements ($g_{j,0:n-1}$) in each column j of CAU to obtain the rank of each element D_j. The third step is comparing each computed sum s_j with the order i to obtain the one-hot address of the i-th smallest element. Finally, we select the value of the i-th smallest element using the one-hot address. From a hardware point of view, we need adder circuits and equality comparator circuits in addition to CAU. For the example in Figure 4.4, a selection circuit is given in Figure 4.6.

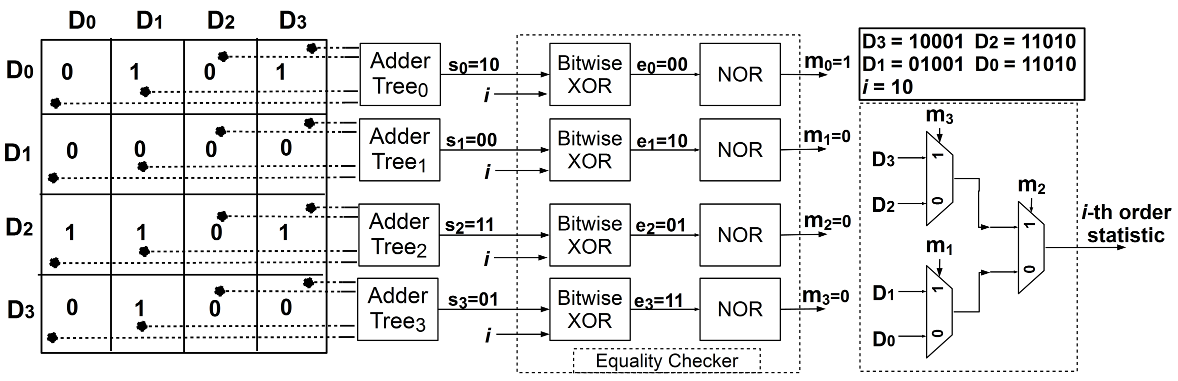


Figure 4.6. An example AbT-based selection circuit.

The selection problem can also be solved by sorting all elements of the set and then simply choosing the i-th element of the sorted list. However, a circuit tailored to solve a larger problem (e.g., sorting) may be more complex than required to solve a smaller problem (e.g., selection). For example, a bitonic sorter [25] circuit, which is one of the well-known parallel sorting methods, sorts all elements in $O((\log_2 n)^2 \times \log_2 k)$ time whereas the circuit in Figure 4.6 determines the address of i-th element in $O((\log_2 n)^2 + \log_2 k)$ time. However, the bitonic sorter circuit consumes $O(k\,n\,\log_2 n)$ area while AbT has an area complexity of $O(k\,n^2)$. As seen from these complexity expressions, for some specific values of parameters n and k, using AbT can be more convenient than using a bitonic sorter circuit in terms of time and/or area.

Another important feature of AbT is that we can use any type of comparison circuit ($>$, $<$, $\geq$, $\leq$) to implement (2,k)-*comparators* of CAU. Hence, we can rewrite 4.5 in a more general form such that:

$$g_{xy} = \begin{cases} 0 & if\ y = x, \\ D_y\, op\, D_x & if\ y < x \quad \forall x, y \in [0, n-1] \\ \overline{g_{yx}} & if\ y > x \end{cases} \tag{4.20}$$

In Equation 4.20, *op* specifies the type ($>$, $<$, $\geq$, $\leq$) of the (2,k)-*comparators* of CAU. Depending on *op*, there are four possible configurations for AbT. For each configuration, AGU uses different g_{xy} values of CAU to compute a bit m_i of one-hot address. Also, resulting priority scheme depends on the chosen *op*. For each *op* type, g_{xy} values used to compute m_i and the resulting priority schemes are listed in Table 4.1 for maximum as well as minimum finder circuits. In ascending (descending) priority scheme, priority

Table 4.1. Implementation options for CAU.

op	**Finding Maximum**		**Finding Minimum**	
	AGU Inputs	**Priority Scheme**	**AGU Inputs**	**Priority Scheme**
$>$	g_{ix}	Descending	g_{yi}	Ascending
$\leq$	g_{ix}	Ascending	g_{yi}	Descending
$<$	g_{yi}	Ascending	g_{ix}	Descending
$\leq$	g_{yi}	Descending	g_{ix}	Ascending

of an input element increases (decreases) as its address increases (decreases). Note that, we have chosen *op* as $>$ in Figure 4.2 and 4.3, and for experimental work which we shall present in Chapter 6.

4.2. Hybrid Binary Tree Topology (HBT)

In Sections 3.4 and 3.5, we explained TBT and PBTs, respectively. In TBT, signals are propagated from LSB to MSB, while signals in PBTs are propagated from MSB to LSB. Thus, PBTs are superior than TBT in terms of latency (speed). On the other hand, area complexity of (*2*,*k*)-*cmp-mux* circuit of TBT is less than the area complexities of PBTs' (*2*,*k*)-*cmp-mux* circuits. In this section, we propose a hybrid MF circuit topology, which combines the speed advantage of PBTs and area advantage of TBT. We call this topology *Hybrid Binary tree Topology (HBT)* since HBT is a kind of combination of PBT and TBT. Owing to HBT, we obtain an MF topology, which is faster than TBT and smaller than PBTs. In the following paragraphs we shall explain the details of HBT.

For simplicity, let us first consider RCT, which is a variant of PBT. RCT can be thought of as consisting of cascaded instances of TBT that operate in parallel as shown in Figure 4.7. In Figure 3.7, we show the (8,4)-*RCT* as a binary tree configuration of

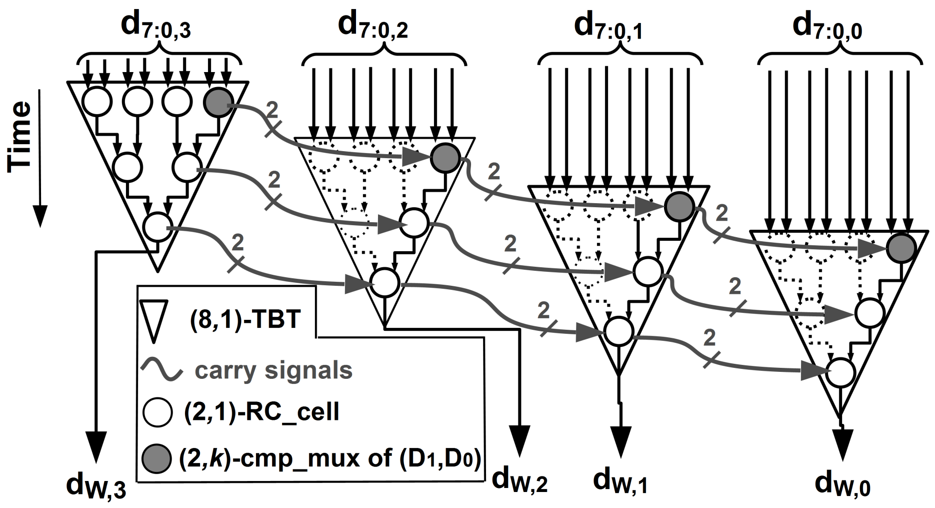


Figure 4.7. An example (8,4)-HBT circuit.

$(n-1)$ (2,4)-*cmp-mux* circuits. On the other hand, in Figure 4.7, we present the (8,4)-RCT as a ripple configuration of four (8,1)-TBTs, in which each TBT takes a bit-slice of n input elements and computes the corresponding bit of the maximum value. These

TBTs operate in an interleaved manner. That is, they operate in parallel, except a delay between subsequent bit positions that is equal to the delay of a (2,1)-*RC-cell.* The carry signal flows between (2,1)-*RC-cells* in the same positions in different TBTs in a ripple style. This ripple connection is shown for three out of the seven (2,4)-*cmp-mux*es in Figure 4.7 so that the figure does not look crowded. Each group of four (2,1)-*RC-cells* connected in a ripple forms a (2,4)-*cmp-mux* of PBT, which is topologically identical to a ripple carry adder.

Hence, we can make the following generalization for an (n,k)-*PBT* topology, which is also the main idea behind HBT: Any (n,k)-*PBT* circuit can be implemented as a ripple configuration of $\lceil k/p \rceil$ (n,p)-*TBT*s, in which each TBT computes the corresponding p bits of the maximum. We obtain (n,k)-*RCT* for $p = 1$, while we get (n,k)-*CLT* when p equals to k. Similarly, we can obtain the block diagram for HBT by generalizing Figure 4.7 to Figure 4.8.

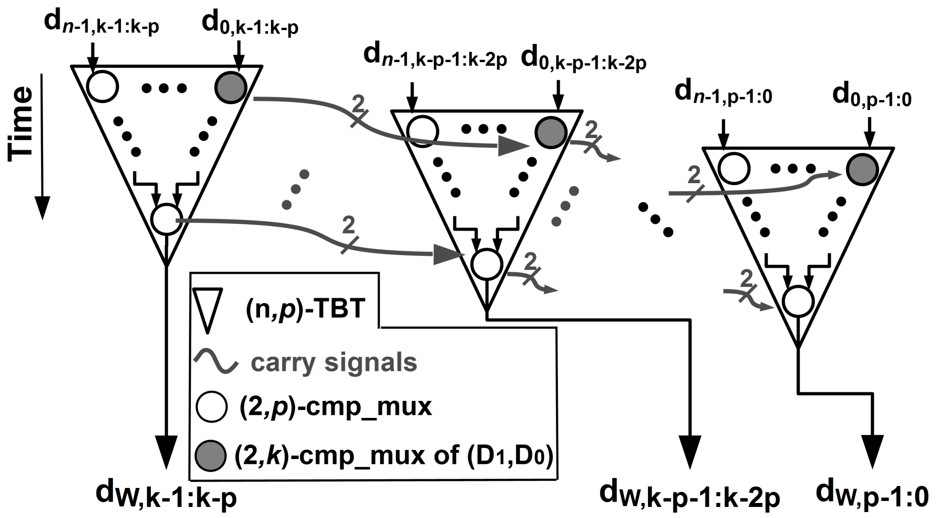


Figure 4.8. Block diagram for an (n,k)-*HBT* consisting of cascaded (n,p)-*TBT*s.

The internal structure of each $(2,p)$-*cmp-mux* of Figure 4.8 is given in Figure 4.9. Here, we made some modifications on the cmp-mux internal structure explained in Section 3.4 in order to incorporate the carry signals (*found* and *choose*) of PBT into TBT. We put an additional circuitry consisting of an AND2 and an OR2 gate

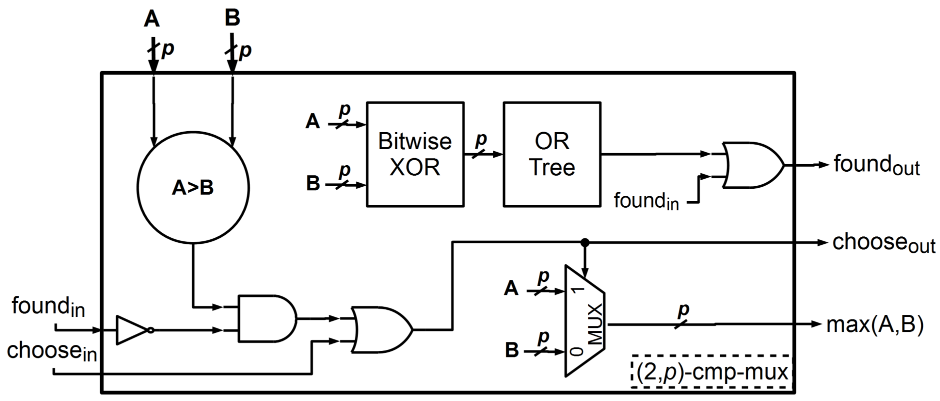


Figure 4.9. Internal structure for a (2,p)-*cmp-mux* of HBT.

between the output of comparator circuit and the select input of multiplexer in order to compute output *choose* signal. We also added an XOR and an OR gate block to produce output *found* signal.

As it shall be seen in Section 4.2.1, HBT has a different theoretical complexity value for each different value of p.

4.2.1. Theoretical Complexity of HBT

We should first obtain theoretical complexity expressions of our modified (2,p)-*cmp-mux*:

$$\begin{aligned} t_{(2,p)-cmp-mux} &= max\left(t_{A>B}, t_{choose_{out}}, t_{found_{out}}, t_{value}\right) \\ &= max\left(\log_2 p, \log_2 p + 2, \log_2 p + 3, \log_2 p + 4\right) \\ &= \log_2 p + 4 = O(\log_2 p) \qquad (4.21) \end{aligned}$$

$$\begin{aligned} A_{(2,p)-cmp-mux} &= A_{A>B} + A_{choose_{out}} + A_{found_{out}} + A_{value} \\ &= p + 2 + 3 \times p + 3 \times p \\ &= 7 \times p + 1 = O(p) \qquad (4.22) \end{aligned}$$

$$P_{(2,p)-cmp-mux} = A_{(2,p)-cmp-mux} = O(p) \qquad (4.23)$$

Theoretical complexity expressions of HBT:

$$\begin{aligned}
t_{HBT} &= \left(\frac{k}{p} + \log_2 p - 1\right) \times t_{(2,p)-cmp-mux} \\
&= \left(\frac{k}{p} + \log_2 n - 1\right) \times (\log_2 p + 4) \\
&= O\left(\left(\frac{k}{p} + \log_2 n\right) \times \log_2 p\right) \qquad (4.24) \\
A_{HBT} &= \frac{k}{p} \times t_{(n,p)-cmp-mux} \\
&= \frac{k}{p} \times (n-1) \times t_{(2,p)-cmp-mux} \\
&= \frac{k}{p} \times (n-1) \times (7 \times p + 2) = O(k \times n) \qquad (4.25) \\
P_{HBT} &= A_{HBT} = O(k \times n) \qquad (4.26)
\end{aligned}$$

4.2.2. Delay Estimator for HBT

As it can be seen in Equation 4.24, HBT has a different timing complexity for each value of p. That is, there are multiple HBT solutions for a specific (n,k) combination and the performance of each solution is different. Hence, we should choose the best partition value p to implement the fastest (n,k)-HBT circuit.

The most obvious way of finding the fastest HBT implementation for a certain combination of n and k is doing synthesis for all possible values of p and choosing the fastest implementation from among the synthesized circuits. However, this approach has several disadvantages. First, it is very time consuming, especially for large values of k. Second, it is an expensive method taking into account the licensing costs of CAD tools. Since companies generally pay high amounts of money for CAD tool licenses for a limited time, their purpose is completing each design as soon as possible. That is, licensing cost per design should be as small as possible. In our case, we can define a design as implementing the fastest (n,k)-HBT for a specific (n,k) case. For our brute force approach, however, licensing cost per design is the maximum since we try all possible solutions in our design space.

In order to overcome the disadvantages of the obvious method, we can use an estimation software, which estimates the fastest HBT implementation for a given (n,k) combination before synthesis. For this purpose, we wrote an estimation tool in Perl, which estimates the performance of different HBT implementations based on Equation 4.24. As Equation 4.24 roughly estimates the timing complexity of an HBT implementation, the fastest HBT implementation chosen by the estimator might not be the fastest HBT implementation in reality. Hence, we should synthesize a group of HBT implementations rather than synthesizing only the first choice of the estimator. In this work, we shall synthesize the first four choices of the estimator in Chapter 6 to evaluate the accuracy and precision of the estimator. We can summarize how our estimator works as follows:

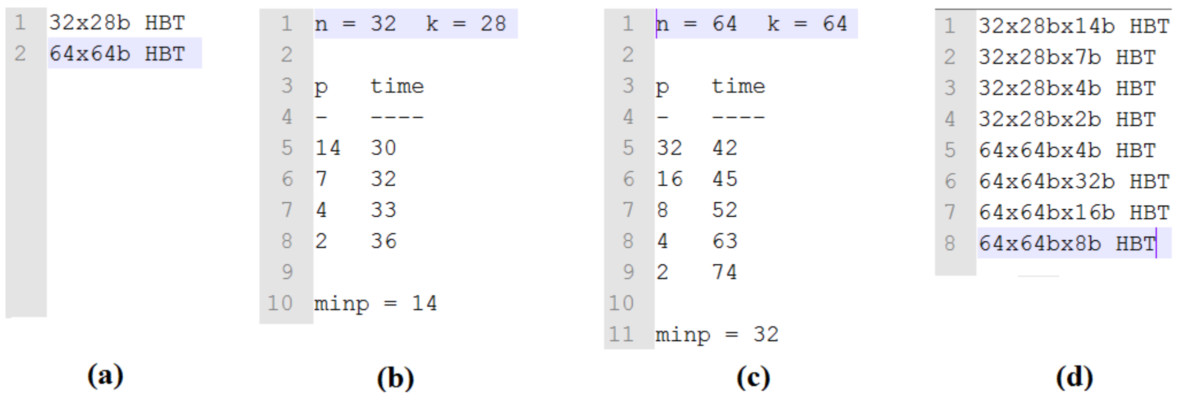
(i) The first step is creating the input file of the estimator, which we call *job-list*. Each line of the job-list specifies the values of n and k as well as the name of the topology to be synthesized. The format of the job-list is as follows (see Figure 4.10a): nxkb *topology*. In this case, the name of the topology is obviously HBT.

(ii) Then, the estimator script is executed with the input file. The estimator reads the each line of the job-list, computes the timing complexity values of each line for all possible partition (p) values, and writes the results into a separate output file (see Figure 4.10b and c), which belongs to the current line.

(iii) Finally, it chooses the fastest four partition values of each file created in the previous step and writes them into the results file (see Figure 4.10d), which shall be given as an input file to our main synthesis script described in Chapter 5.

4.3. Quad Tree Topology (QT)

In this section, we shall propose another hybrid topology, which we call Quad Tree Topology (QT). In QT, an (n,k)-*MF* circuit is implemented as a quad tree configuration of (4,k)-AbT circuits. In Figure 4.11, we show a (16,k)-QT as an example. Our main purpose for coming up with QT is to overcome two potential drawbacks of AbT. First one of these handicaps is that AbT has a quadratic area complexity (O($k\,n^2$)) and this could prevent AbT to scale well for large values of parameter n. Second, every input of



(a)

```
32x28b HBT
64x64b HBT
```

(b)

```
n = 32   k = 28

p    time
-    ----
14   30
7    32
4    33
2    36

minp = 14
```

(c)

```
n = 64   k = 64

p    time
-    ----
32   42
16   45
8    52
4    63
2    74

minp = 32
```

(d)

```
32x28bx14b HBT
32x28bx7b HBT
32x28bx4b HBT
32x28bx2b HBT
64x64bx4b HBT
64x64bx32b HBT
64x64bx16b HBT
64x64bx8b HBT
```

Figure 4.10. (a) Input file of the estimator (job-list). (b) Output for the first line of the job-list. (c) Output for the second line of the job-list. (d) Result file of the estimator.

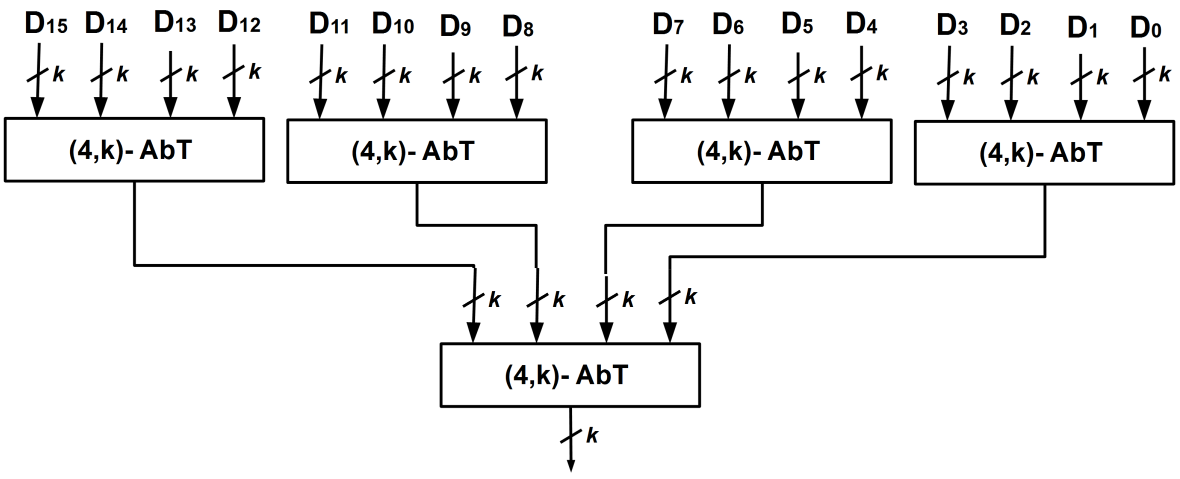


Figure 4.11. Block diagram of QT ($n = 16$).

AbT has a fanout of $n-1$ and this could cause degradation in expected performance and necessitate buffer insertion for large values of n. In order to overcome these drawbacks, we can utilize hybrid topologies that use AbT as a building block. QT is a promising candidate for such topologies because both the area and fanout disadvantages of AbT become inconsiderable for $n = 4$. QT utilizes the speed advantage of AbT and area advantage of quad tree configuration. As a result, we obtain an MF circuit topology, which is significantly faster than any MF topology in the previous work as well as brings a little area overhead on the most area-efficient MF circuit topology in the previous work.

4.3.1. Theoretical Complexity of QT

We can estimate the theoretical complexity of (n,k)-QT as follows.

$$\begin{aligned} t_{QT} &= \log_4 n \times t_{(4,k)-AbT} \\ &= \log_4 n \times O(\log_2 k + 2) = O(\log_4 n \times \log_2 k) \end{aligned} \tag{4.27}$$

$$\begin{aligned} A_{QT} &= \frac{n-1}{3} \times A_{(4,k)-AbT} \\ &= \frac{n-1}{3} \times O(k \times 16) = O(k \times n) \end{aligned} \tag{4.28}$$

$$P_{QT} = A_{QT} = O(k \times n) \tag{4.29}$$

As it can be seen from Equations 4.27 and 4.28, our area complexity reduces from $O(k\,n^2)$ to $O(k\,n)$ in comparison to AbT. Besides, fanout of each input becomes $O(1)$ as opposed to $O(n)$. We shall validate our claims in Chapter 6.

5. RTL GENERATION AND SYNTHESIS METHODOLOGY

In Chapters 3 and 4, we explained the details of several MF circuit topologies and derived their theoretical computational complexity expressions based on the simple complexity models given in Section 2.2. Our next step shall be to validate our theoretical claims by synthesizing the proposed architectures as well as competitive existing structures. Hence, we need to write Register-Transfer Level (RTL) Verilog code of each topology to be synthesized and testbench code for verification of the RTL design. Besides, we should configure our synthesis tool for each specific synthesis job to obtain the optimum synthesis results. If we try to handle the whole process manually, it shall be excessively time-consuming and error-prone as we want to synthesize the MF topologies for many (n,k) combinations. Thus, we automated the synthesis process. In this chapter, we shall give details of our automated synthesis methodology.

Our methodology starts by creating the *input file*, each line of which contains a synthesis job to be done. A synthesis job is specified by the parameters n, k, and name of the MF topology to be synthesized. Then, we call our *main synthesis script* (written in Perl), which carries out the synthesis jobs in the input file one by one. For each job, main synthesis script synthesizes the specified topology and produces area, timing, and power consumption results. After the main synthesis script completes all of the jobs in the input file, we execute our result extraction script (written in Perl) to collect the synthesis results into a single output file. In Figure 5.1, we give a high-level flow diagram of our methodology. In Figure 5.1, we use black dashed lines to show manual steps, while we use black solid lines for the steps carried out by our main synthesis script. In addition, we use gray solid lines to represent the step executed by the *result extraction script*. In the following sections, we shall examine each step of Figure 5.1 in detail.

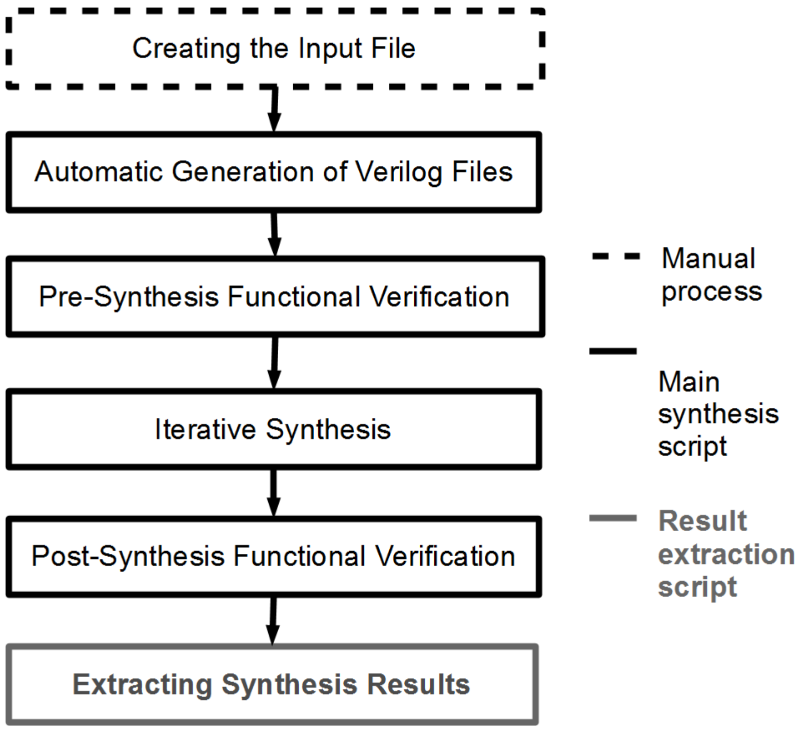

Creating the Input File
Automatic Generation of Verilog Files
Pre-Synthesis Functional Verification
Iterative Synthesis
Post-Synthesis Functional Verification
Extracting Synthesis Results
Manual process
Main synthesis script
Result extraction script

Figure 5.1. Flow diagram of the automated synthesis methodology.

5.1. Creating the Input File

The first step of our methodology is constructing the input file for the main synthesis script. Each line of the input file is a specific synthesis job and it specifies the values of parameters n and k as well as the MF topology to be synthesized. This is the only step in our synthesis methodology that we carry out manually. Its format is the same as the given format in Figure 4.10a. The main synthesis script sequentially reads the each line of the input file and produces synthesis results for the MF circuit, which is defined in this specific line, by performing the steps given in Figure 5.1.

5.2. Automatic Generation of Verilog Files

After reading a specific line of the input file, our main synthesis script automatically generates self-checking testbench and RTL design codes (in Verilog) for the (n,k)-*MF* circuit specified by this line. To achieve this, the main synthesis script uses our code generator scripts, which were written in Perl scripting language and are scalable to any value of k and n. The code generators let us to get rid of intensive Verilog code writing process when we compare the MF topologies against each other for various combinations of n ($n \in 8, 16, 32, 64$) and k ($n \in 8, 16, 24, 32, 64$).

We wrote Verilog code generators for all of the proposed and existing topologies except AT, RT, and ST. Because of their high theoretical timing complexity values, we did not write generators for those three topologies. Our code generators take n and k as input arguments and generate the required Verilog files: *maxNk.v*, *wrappermaxNk.v*, and *TB _maxNk.v*. The structure of our Verilog files is given in Figure 5.2. Note also that HBT has a third argument p, which specifies the bit-widths of the TBTs used in HBT. Thus HBT's code generator takes p as an argument as well. *maxNk.v*

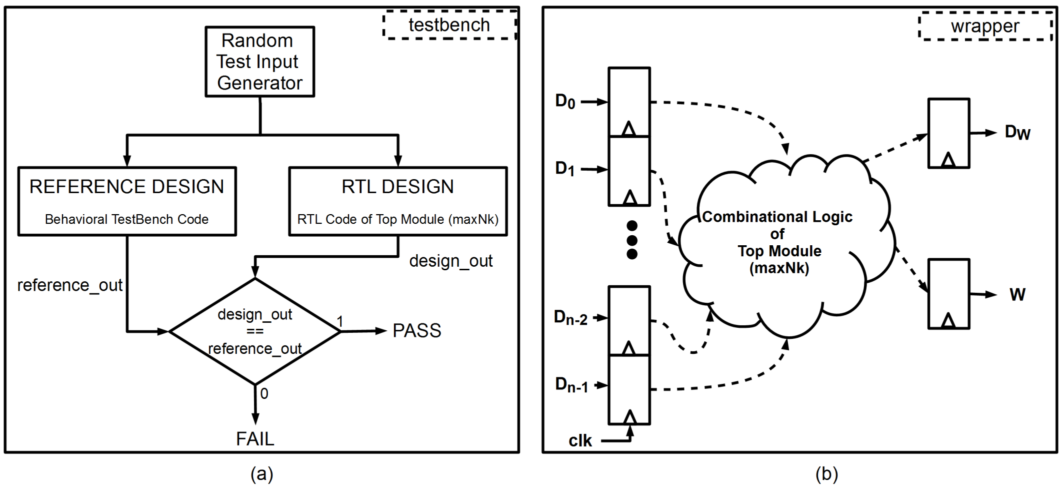

testbench
Random Test Input Generator
REFERENCE DESIGN
Behavioral TestBench Code
RTL DESIGN
RTL Code of Top Module (maxNk)
design_out
reference_out
design_out == reference_out
1
PASS
0
FAIL
(a)
wrapper
D0
D1
Dn-2
Dn-1
clk
Combinational Logic of Top Module (maxNk)
DW
W
(b)

Figure 5.2. (a) Structure defined by the testbench file. (b) Structure defined by the wrapper file.

is RTL Verilog code of the specified (n,k)-*MF* circuit. Its inputs are n k-bit binary numbers and outputs are the value and binary ($\log_2 n$ -bit) address of the maximum number. It includes the top module, called *maxNk*, as well as the definitions of other modules instantiated by the top module. Note also that the maximum element does not necessarily have to be unique. In the case of multiple maximum elements, our top module chooses one of those maximum elements as the winner element applying a priority scheme. In our priority scheme, priority of an input element increases as its address increases.

TB_maxNk.v functionally verifies the top module. It instantiates the top module and checks its correctness applying random test inputs. It is just used in verification steps of Figure 5.1.

wrappermaxNk.v is just for synthesis purposes. It has n k-bit flip-flops right before the inputs. On the output, it has a k-bit flip-flop for the maximum value, and a $\log_2 n$-bit flip-flop for the binary address of the maximum. It instantiates the top module between this flip-flop set so that all top module inputs are fed from flip-flops and all top module outputs are sunk by flip-flops. Hence, our timing results shall be flop-to-flop and include flop clock-to-Q delay as well as setup time.

5.3. Pre-Synthesis Verification

Before synthesis, we should verify functional correctness of our RTL code. Our main synthesis script uses *maxNk.v* and *TB_maxNk.v* files, which are created by code generators explained in Section 5.2, for functional verification. Our testbench can be divided into three blocks as shown in Figure 5.2a. *RTL Design* block is an instantiation of our top module produced in the first step. *Reference Design* is a high-level (behavioral) description of our top module. *Random Test Input Generator* block randomly produces test inputs and applies them to the RTL design as well as the reference design. Finally, the testbench compares the outputs of two design blocks. If those two outputs are equal, our RTL design passes the verification and it can be synthesized. Otherwise, it fails and we should solve the problems before proceeding to iterative synthesis step of Figure 5.1.

5.4. Iterative Synthesis

In the synthesis process, our aim is to obtain the smallest clock period for each synthesis job. One of the most important parameters that affects the achieved clock period is the clock period target, which is supplied to the synthesis tool. Hence, we need to give the optimum clock period target to our synthesis tool in order to obtain the smallest clock period possible. However, we cannot know what the optimum clock period target is beforehand. Fortunately, we can feed the synthesis tool with a clock period target, which is close to the optimum one, via our *iterative synthesis script* [11–13].

Our iterative synthesis script (written in Perl) synthesizes each circuit four times and searches for the smallest clock period with a kind of binary search. In each iteration we have an upper bound value and a lower bound value for the clock period. Lower bound is the largest clock target that was not achieved so far. Upper bound is the smallest clock target that was met so far. Our initial target clock period is 0.1ns (impossible to be met) for the first iteration. For other iterations except the last iteration, our clock period target is the average of the upper and lower bound values. For the last iteration, the target is the 90% of the best achieved clock period so far. As a result, we happen to force the synthesis tool as much as possible to obtain the smallest clock period. Timing, area, and power consumption results are saved into separate text files after each iteration.

5.4.1. Obtaining Timing Results

In order to obtain timing results, we use static timing analysis, which validates the performance of a design by checking all possible timing paths and without requiring simulation. Each timing path has a start-point and an end-point. The data is launched from the start-point by a clock edge and then captured at the end-point by another clock edge after it is propagated through combinational logic between the start-point and end-point. As we use a wrapper consisting of input and output flip-flops, we can define the start-point and end-point as follows: The start-point of a path is clock input of a flip-flop and the end-point is a data input of a flip-flop.

Each path in the design has an associated slack, which is the difference between arrival time and required time of the path. Arrival time of a path is the elapsed time from triggering clock edge at the start-point to arrival of the data signal at the end-point of the path. Required time is the latest arrival time for the path, which does not make the clock cycle any longer than targeted. The minimum clock period, at which the design operates properly, is difference between the target clock period and the slack of the critical path. Critical path of a design is the path that has the smallest algebraic slack value. Our iterative synthesis script outputs the minimum clock period value for its each iteration.

5.4.2. Obtaining Area Results

Since we used the wrapper module only for timing purposes, we do not report the areas of the wrapper's input and output flip-flops. For each iteration, our iterative synthesis script only reports the area of our top module (*maxNk*) in square microns.

5.4.3. Obtaining Power Consumption Results

Our iterative synthesis script reports the average dynamic power dissipation of the synthesized gate-level design for each iteration in Watts. We use a toggle rate based (statistical) power analysis method, which does not require simulation. Toggle is a transition from 0 to 1 or vice versa. Toggle rate is the number of toggles per unit time. In the power estimation method, we supply the toggle rates at the primary inputs of our design and then these values are propagated into the circuit. Finally, power analysis tool calculates the average dynamic power dissipation using the toggle rates propagated.

5.5. Post-Synthesis Verification

After synthesis, we obtain a gate-level netlist for our design. We have to verify the functionality of this gate level synthesis. We use the same method explained in Section 5.3 for this purpose. The only difference is that we have a gate-level netlist consisting of cells of our standard cell library instead of a RTL Verilog code.

5.6. Extraction of Synthesis Results

We should collect all timing, area, and power results, which are obtained after synthesis, into a single file to analyze the results. We wrote a result extraction script in Perl that automatically collects all of the results and writes them into a *comma seperated values* (CSV) file.

6. EXPERIMENTAL WORK

In this chapter, we present our experimental setup as well as synthesis area, timing, and power consumption results, which are obtained using the methodology explained in Chapter 5, for our proposed MF topologies and their competitors.

6.1. Experimental Setup

Synthesis results highly depend on the version of the synthesis tool, synthesis parameters, targeted technology, and standard cell library. In our experimental work, we used 2010.12 version of Synopsys Design Compiler (DC) as the synthesis tool and the worst case version of UMC Faraday 0.18 μm standard cell library with a *wire-load model (WLM)*. Moreover, we used a server class workstation with a Xeon-X5650 processor running at 2.67GHz with 8GB DRAM.

Delay due to wires (wire load and wire propagation delay) is an extremely important contributor to overall delay in a circuit. It is not too hard to compute delay due to wires after physical placement and routing. However, we did not apply a place and route process in this work. Hence, we used a set of WLMs, which is supplied in the technology library, in order to take the wire delay into account during the synthesis process. A WLM consists of lookup tables that map the fanout of a net to estimated wirelength, thereby estimated capacitance and resistance [26]. A net with a larger fanout is assumed to have more wires and therefore more delay (more resistance and capacitance). Furthermore, as the design size increases, standard cells may happen to be placed physically farther apart within that design, which typically results in longer wires. Hence, our WLM set includes several WLMs to be used for different design sizes. In our synthesis process, DC automatically selects an appropriate WLM based on the area of the design. In addition, we set the wire-load mode to *enclosed*, which means that the wire delay of each net is calculated using the WLM corresponding to the smallest subcircuit that completely encloses that net.

In this work, synthesis was carried out with a main batch script which uses an input (job-list) file and calls other scripts when required, as we explained in Chapter 5. Every row of the job-list file specifies k and n parameters and the circuit topology to be synthesized. The main synthesis script first reads the job and then performs a three-stage process: As the first step, it generates the RTL and testbench based on k, n, and the specified topology. Then, the main script functionally verifies the design with the self-checking testbench produced in the first step. If the design passes the test, an iterative synthesis script [11–13], which calls DC in batch mode, is executed. Otherwise, the process is aborted, and the next job in the job-list file is read. The iterative synthesis script synthesizes the circuit *four times* and searches for the smallest clock period with a kind of binary search. In each synthesis run, area, power consumption, and timing results are written into a file. Then, the target clock period is updated.

6.2. Experimental Results

Before we give the results, we present the theoretical area and time complexities in Table 6.1 for all MF topologies. Since area and power consumption are of the same order of magnitude according to our complexity models explained in Section 2.2, we do not list the power consumption complexities in Table 6.1. Based on the complexities in Table 6.1, we chose the competitors (TBT, RCT, CLT, CST, MLT, and LCT) of our proposed topologies (AbT, HBT, and QT) for implementation and excluded the rest due to their high theoretical complexities. Hence, we wrote Verilog code generators for all of the topologies except AT,RT, and ST. As stated in Chapter 5, our scripts generate self-checking testbench code for functional verification and RTL (in Verilog) for logic synthesis. Also, our code generators are scalable to any value of k and n.

All of the implemented topologies output both the value and binary address of the maximum element. We implemented two different versions (AbT_1 and AbT_2) of AbT. AbT_1 outputs the value and one-hot address, while AbT_2 outputs the value and binary address of the maximum element. The purpose of implementing AbT_1 is not to compare it with the competitors, but to prove that AbT can produce binary and/or one-hot address in $O(\log_2 k + \log_2 n)$. On the other hand, AbT_2 is compared with the

Table 6.1. Theoretical comparison of all MF topologies.

Topology	**Timing Complexity**	**Area Complexity**
AT	$O(k \times \log_2 n + k)$	$O(k \times n)$
RT	$O(k \times \log_2 n + k)$	$O(n)$
ST	$O(k \times \log_2 n + k)$	$O(n)$
TBT	$O(\log_2 k \times \log_2 n + \log_2 n)$	$O(k \times n)$
RCT	$O(k + \log_2 n)$	$O(k \times n)$
CST	$O(\sqrt{k})$ + $O((\sqrt{k} - \log_2 n) \times log_2 n)$	$O(k \times n)$
CET	$O(\log_2 k \times \log_2 n)$	$O(k \times n)$
LFT	$O(\log_2 n \times \log_2 k)$	$O(n \times k \times \log_2 k)$
BKT	$O(\log_2 n \times \log_2 k)$	$O(n \times k)$
KST	$O(\log_2 n \times \log_2 k)$	$O(n \times k \times \log_2 k)$
HCT	$O(\log_2 n \times \log_2 k)$	$O(n \times k \times \log_2 k)$
MLT	$O(\log_2 k \times \log_2 n + n)$	$O(k \times 2^k)$
LCT	$O(\log_2 k \times \log_2 n + k)$	$O((k + n) \times 2^k)$
AbT	$O(\log_2 k + \log_2 n)$	$O(k \times n^2)$
HBT	$O((k/p + \log_2 n) \times \log_2 p)$	$O(k \times n)$
QT	$O(\log_4 n \times \log_2 k)$	$O(k \times n)$

competitors topologies as well as other proposed topologies.

In the following subsections we give the synthesis results for several values of k and n such that $k \in \{8, 16, 24, 32, 64\}$ and $n \in \{4, 8, 16, 32, 64\}$. In the result tables, bold numbers shall show the best result in each case (each row of the tables).

Before proceeding further, we should note that we shall not give the detailed synthesis results of LCT although we wrote code generators and obtained synthesis results for this topology as well. The main reason of not giving the results of LCT is that our workstation runs out of memory when LCT is synthesized for $k > 8$. The second reason is that obtained synthesis results of LCT for $k \leq 8$ are only better than the results of MLT. Hence, we shall not show the results for LCT in the Sections 6.2.1–6.2.7.

6.2.1. Timing Results

In this subsection, we shall give the timing results (minimum achievable clock period) in nanoseconds. All the inputs are fed from flip-flops (FFs) and all outputs are also sunk by FFs. The timing results given in this subsection are FF-to-FF and thus include clock-to-Q delays and setup times of the FFs.

Table 6.2 shows the timing results for the two variants of AbT (AbT_1 and AbT_2) in nanoseconds. AbT computes the one-hot and binary addresses of the maximum element with the same timing complexity as seen in the Table 6.2.

In Table 6.3, we list the timing results of the proposed topologies and their competitors. Although we implemented five variants (CET, LFT, HCT, KST, and BKT of Section 3.5.3) of CLT, we only give the best timing results of the five variants as a column (CLT) in Table 6.3 for brevity. We present all of the timing results for the five variants of CLT in Table 6.4. Similarly, we put only the best timing results of HBT into Table 6.3, although we obtained up to four different timing results (for each different partition value) for each specific combination of n and k. All of the obtained

Table 6.2. Comparison of two AbT variants in terms of timing (in nanoseconds): AbT_1 (value + one-hot address) and AbT_2 (value + binary address).

$\boldsymbol{n}\times\boldsymbol{k}$	**AbT$_1$ (one-hot address)**	**AbT$_2$ (binary address)**
4×8	1.65	1.66
4×16	1.91	1.87
4×24	2.02	2.00
4×32	2.09	2.07
4×64	2.28	2.30
8×8	1.94	1.97
8×16	2.17	2.15
8×24	2.39	2.38
8×32	2.49	2.52
8×64	2.71	2.76
16×8	2.27	2.28
16×16	2.57	2.64
16×24	2.72	2.84
16×32	2.87	2.87
16×64	3.15	3.18
32×8	2.73	2.82
32×16	3.07	3.06
32×24	3.26	3.30
32×32	3.48	3.49
32×64	5.34	5.24
64×8	3.26	3.34
64×16	4.56	5.33
64×24	5.77	5.71
64×32	5.63	5.70
64×64	8.23	7.25
Average	3.22	3.23

timing results of HBT can be found in Table 6.6.

As seen in Table 6.3, AbT offers superior performance than any competitor in all cases. It is the fastest topology in 22 out of 25 test cases as well. AbT is 1.21 –2.17 times faster than the fastest competitor. On the average, AbT is 1.61 times faster than the fastest competitor. Besides, the performance superiority of AbT becomes more apparent as k increases. For instance, AbT is 1.41 times faster on the average than its closest competitor for $k = 8$, while it is 1.72 times faster for $k = 64$.

The second fastest topology of Table 6.3 is QT, which is 1.26 times slower than AbT on the average. On the other hand, it is slightly faster than AbT for ($n = 64$, $k = 16$) and ($n = 64$, $k = 64$). It is 1.10 –1.41 times faster than the fastest competitor. On the average, QT is 1.28 times faster than the fastest competitor in the literature.

Our third proposed topology, HBT, is 0.88 –1.06 times faster than the fastest competitor. In addition, it is 1.01 times faster than the fastest competitor on the average. HBT is 0.94 times faster (1.06 times slower) than the fastest competitor for $k = 8$, while it is 1.04 times faster for $k = 64$ on the average. Hence, it becomes more preferable to the fastest competitor as the parameter k increases. Furthermore, HBT is faster than the fastest competitor in 19 out of 25 test cases of Table 6.3, and it is only 1.06 times slower, on the average, than the fastest competitor in the remaining six cases. HBT is also 1.25 times faster than RCT, 1.07 times faster than CST as well as 1.01 times faster than CLT on the average. On the other hand, HBT is 1.59 and 1.27 times slower than AbT and QT, on the average, respectively. In addition to timing results of HBT in Table 6.3, we list the timing results of the fastest four HBT implementations, in regards to our estimator explained in Section 4.2.2, in Table 6.6 of Section 6.2.3 in order to evaluate our estimator.

Considering the competitors, PBT is the fastest topology in all cases except the cases of ($n = 4$, $k = 64$), ($n = 8$, $k = 64$), and ($n = 32$, $k = 64$). For ($n = 4$, $k = 64$) and ($n = 8$, $k = 64$), the fastest competitor is TBT, while for ($n = 32$, $k = 64$) MLT is the fastest competitor. Except for the cases of ($n = 16$, $k = 32$), ($n = 64$,

Table 6.3. Timing results for the proposed and competitor topologies in nanoseconds.

	Proposed Topologies			**Competitor Topologies**				
$n \times k$	AbT_2	QT	HBT[1]	TBT	RCT	CST	CLT[2]	MLT
4×8	1.66	**1.64**	2.17	2.23	2.37	2.24	2.19	2.80
4×16	**1.87**	**1.87**	2.62	2.74	3.26	2.72	2.63	3.46
4×24	**2.00**	2.04	2.79	3.01	4.12	2.98	2.85	3.95
4×32	**2.07**	2.11	2.99	3.14	4.86	3.26	3.02	4.14
4×64	**2.30**	2.27	3.45	3.51	6.37	3.97	3.57	4.53
8×8	**1.97**	2.76	2.97	3.09	3.03	3.07	2.84	3.94
8×16	**2.15**	3.08	3.58	3.89	4.03	3.67	3.65	4.43
8×24	**2.38**	3.40	3.87	4.18	4.93	4.14	4.05	4.95
8×32	**2.52**	3.67	4.16	4.52	5.82	4.54	4.36	5.16
8×64	**2.76**	4.10	4.91	5.19	9.45	5.60	5.21	5.78
16×8	**2.28**	2.97	3.61	4.09	3.59	3.76	3.46	4.80
16×16	**2.64**	3.44	4.46	5.06	4.77	4.72	4.61	5.65
16×24	**2.84**	3.77	5.02	5.62	5.85	5.33	5.23	6.16
16×32	**2.87**	3.97	5.51	6.00	6.73	5.80	5.82	6.37
16×64	**3.18**	4.45	6.65	7.23	10.97	7.48	6.91	6.99
32×8	**2.82**	4.15	4.39	4.95	4.10	4.33	3.99	6.02
32×16	**3.06**	4.94	5.54	6.19	5.52	5.57	5.32	6.73
32×24	**3.30**	5.49	6.48	7.22	6.95	6.67	6.50	7.31
32×32	**3.49**	5.74	6.89	7.71	7.93	7.26	7.04	7.52
32×64	**5.24**	6.42	8.19	8.88	12.39	9.45	8.59	8.23
64×8	**3.34**	4.53	5.23	6.15	4.61	5.27	4.70	6.85
64×16	5.33	**5.20**	6.43	7.77	6.44	6.67	6.49	7.81
64×24	**5.71**	5.86	7.47	8.78	7.70	7.83	7.35	8.87
64×32	**5.70**	6.14	8.03	9.37	9.03	8.62	8.24	8.82
64×64	7.25	**6.71**	9.73	10.67	13.61	11.33	10.31	14.28

$k = 8$), and ($n = 64$, $k = 16$), CLT is the fastest PBT variant. We list the timing results of all five variants of CLT in Table 6.4. When we look at Table 6.4, we can see that our PPN implementations (BKT, LFT, HCT, and KST) are faster than the direct implementation (CET) of Equations 3.36 and 3.37 in 16 out of 25 cases.

6.2.2. Delay Overhead of DSU

In Section 4.1.3, we stated that although DSU does not increase the overall delay of AbT in theory, it brings an extra delay overhead on AGU in practice because it increases the fanout of AGU nodes to which it is connected. To show the effect of DSU on the delay of AGU, we listed the timing results for AbT_2 with/without DSU in Table 6.5 in nanoseconds. As seen in Table 6.5, DSU increases the overall delay 13.69% on the average. In other words, we can compute the value of the maximum element in addition to its address with only a 13.69% delay overhead thanks to our parallel DSU architecture explained in Section 4.1.3. Note also that the AbT variant with DSU is faster than the AbT variant without DSU in the case of ($n = 64$, $k = 64$).

6.2.3. Evaluation of HBT Estimator

In this subsection, we shall present the synthesis results of HBT for our 25 example cases. For each different (n,k) combination in Table 6.6, we give the results of four (if possible) theoretically fastest HBT implementations in regards to our estimator. We list the results within each row of the table, in descending order in terms of their theoretical timing complexity values computed by the estimator. In other words, the second column (1^{st}) includes the synthesis results of the theoretically fastest HBT implementations while the last column (4^{th}) includes the results of the fourth fastest HBT implementation. We also show the corresponding partition (p) values in parentheses.

As it can be seen from Table 6.6, the first or the second choice of our estimator has the best timing results in all cases. Furthermore, the last choice of our estimator

[1]Results of the fastest topology in Table 6.6.

[2]Results of the fastest topology in Table 6.4.

Table 6.4. Timing results for the five variants of CLT in nanoseconds.

$n{\times}k$	**CET**	**BKT**	**LFT**	**HCT**	**KST**
4×8	**2.19**	2.25	2.30	2.25	2.22
4×16	2.75	2.71	2.68	**2.63**	2.63
4×24	3.09	3.00	2.91	**2.85**	2.92
4×32	3.26	3.23	**3.02**	3.07	3.07
4×64	3.94	3.76	**3.57**	3.58	**3.57**
8×8	**2.84**	3.02	3.11	3.06	3.04
8×16	3.77	3.74	**3.65**	3.66	3.68
8×24	4.34	4.08	4.10	**4.05**	4.05
8×32	4.68	4.49	**4.36**	4.43	4.38
8×64	5.93	5.44	5.38	**5.21**	**5.21**
16×8	**3.46**	3.78	3.81	3.83	3.80
16×16	**4.61**	4.73	4.69	4.74	4.72
16×24	5.45	5.30	**5.23**	5.48	5.53
16×32	6.03	5.87	**5.82**	5.88	5.85
16×64	7.50	**6.91**	6.93	7.05	7.02
32×8	**3.99**	4.42	4.58	4.61	4.64
32×16	**5.32**	5.75	5.92	5.97	6.00
32×24	6.57	6.51	**6.50**	6.70	6.83
32×32	7.22	**7.04**	7.12	7.21	7.37
32×64	9.34	**8.59**	8.78	8.70	8.65
64×8	**4.70**	5.50	5.48	5.61	5.56
64×16	**6.49**	6.81	6.77	7.11	7.01
64×24	**7.35**	7.95	7.83	8.22	8.21
64×32	8.43	**8.24**	8.46	8.66	8.74
64×64	11.24	**10.31**	10.64	11.15	11.22

Table 6.5. Delay overhead of DSU.

$n \times k$	AB$_2$ (without DSU)	AB$_2$ (with DSU)	Delay Overhead (%)
4×8	1.46	1.66	13.89
4×16	1.63	1.87	14.88
4×24	1.67	2.00	19.82
4×32	1.74	2.07	18.97
4×64	1.82	2.30	26.45
8×8	1.77	1.97	11.30
8×16	1.96	2.15	9.44
8×24	2.11	2.38	12.56
8×32	2.12	2.52	18.63
8×64	2.27	2.76	21.63
16×8	2.13	2.28	6.92
16×16	2.40	2.64	10.10
16×24	2.52	2.84	12.70
16×32	2.65	2.87	8.11
16×64	2.79	3.18	13.80
32×8	2.55	2.82	10.61
32×16	2.90	3.06	5.53
32×24	2.96	3.30	11.57
32×32	3.18	3.49	9.75
32×64	4.80	5.24	9.01
64×8	3.13	3.34	6.88
64×16	3.53	5.33	51.06
64×24	5.04	5.71	13.31
64×32	5.37	5.70	6.15
64×64	7.30	7.25	-0.75

Table 6.6. Timing results of different HBT implementations.

$\boldsymbol{n \times k}$	**HBT Implementations**			
	1^{st}	2^{nd}	3^{rd}	4^{th}
4×8	**2.17** $(p = 4)$	2.21 $(p = 2)$	-	-
4×16	**2.62** $(p = 8)$	2.63 $(p = 4)$	2.72 $(p = 2)$	-
4×24	**2.79** $(p = 8)$	2.83 $(p = 12)$	2.85 $(p = 6)$	2.88 $(p = 4)$
4×32	3.05 $(p = 16)$	**2.99** $(p = 8)$	3.15 $(p = 4)$	3.54 $(p = 2)$
4×64	**3.45** $(p = 32)$	3.47 $(p = 16)$	3.56 $(p = 8)$	4.13 $(p = 4)$
8×8	**2.97** $(p = 4)$	**2.97** $(p = 2)$	-	-
8×16	3.62 $(p = 8)$	**3.58** $(p = 4)$	3.65 $(p = 2)$	-
8×24	**3.87** $(p = 8)$	3.98 $(p = 6)$	4.01 $(p = 4)$	4.12 $(p = 12)$
8×32	**4.16** $(p = 8)$	4.33 $(p = 16)$	4.30 $(p = 4)$	4.66 $(p = 2)$
8×64	5.08 $(p = 32)$	**4.91** $(p = 16)$	4.98 $(p = 8)$	5.36 $(p = 4)$
16×8	**3.61** $(p = 2)$	3.83 $(p = 4)$	-	-
16×16	**4.46** $(p = 4)$	4.69 $(p = 8)$	4.59 $(p = 2)$	-
16×24	5.09 $(p = 8)$	**5.02** $(p = 4)$	5.04 $(p = 6)$	5.42 $(p = 3)$
16×32	**5.46** $(p = 8)$	5.51 $(p = 4)$	5.75 $(p = 16)$	5.74 $(p = 2)$
16×64	**6.65** $(p = 16)$	6.67 $(p = 8)$	6.86 $(p = 32)$	7.03 $(p = 4)$
32×8	**4.39** $(p = 2)$	4.60 $(p = 4)$	-	-
32×16	5.57 $(p = 4)$	**5.54** $(p = 2)$	5.94 $(p = 8)$	-
32×24	6.49 $(p = 4)$	6.67 $(p = 8)$	**6.48** $(p = 6)$	6.94 $(p = 3)$
32×32	6.99 $(p = 8)$	**6.89** $(p = 4)$	8.38 $(p = 16)$	7.14 $(p = 2)$
32×64	**8.19** $(p = 8)$	8.38 $(p = 16)$	8.63 $(p = 32)$	8.59 $(p = 4)$
64×8	**5.23** $(p = 2)$	5.47 $(p = 4)$	-	-
64×16	**6.43** $(p = 2)$	6.78 $(p = 4)$	7.32 $(p = 8)$	-
64×24	7.55 $(p = 4)$	**7.47** $(p = 2)$	7.59 $(p = 8)$	8.28 $(p = 3)$
64×32	**8.03** $(p = 4)$	8.34 $(p = 8)$	8.36 $(p = 2)$	9.10 $(p = 16)$
64×64	**9.73** $(p = 8)$	10.06 $(p = 16)$	10.16 $(p = 4)$	10.55 $(p = 32)$

has the worst timing results in 21 out of 25 cases. In the remaining four cases, the estimator's last choice has the second worst timing. Hence, we can use our estimator to avoid synthesizing an (n,k)-*HBT* for each possible partition of k. We can synthesize the first three choices of the estimator and then pick the fastest one among them. So that, we can implement the fastest (n,k)-*HBT* circuit by only synthesizing three different possible implementations instead of k different possible implementations.

6.2.4. Area Results

Table 6.7 shows the normalized area results. In each case, we take the area of TBT, which is the most area-efficient MF topology as it can be seen in Table 6.7, as unity and scale other values in regards to this value. In addition to individual area results of proposed and competitor topologies, we also give the area results of the fastest competitor of each case within the last column of Table 6.7.

In spite of its superior timing results, AbT is the least area-efficient one of our proposed topologies as it can be seen in Table 6.7. The area results of AbT is 4.82 times worse, on the average, than the fastest competitor's results and 7.53 times worse than TBT's results. In contrast, AbT is 1.61 and 1.76 times faster than its fastest competitor and TBT, respectively, as shown in Table 6.3. Despite this overall area overhead, AbT has very promising area results for $n = 4$. AbT is 1.20 times smaller as well as 1.43 times faster than the fastest competitor for $n = 4$. Furthermore, it is only 1.30 times larger than TBT, while it is 1.47 times faster than TBT for $n = 4$.

QT, another proposed topology, is 1.37 times larger and 1.40 times faster than the most area-efficient topology (TBT) on the average. Moreover, QT is 1.25 times more area-efficient as well as 1.28 times faster than its fastest competitor as stated in Section 6.2.1. On the other hand, QT is 5.50 times more area-efficient than AbT although it is 1.26 times slower than AbT on the average.

In regards to Table 6.7, HBT is the most area-efficient proposed topology. The area of HBT is 1.19 times larger than TBT, which is the best MF topology in terms of

Table 6.7. Normalized area results.

	Proposed Topologies			**Competitor Topologies**					
$n{\times}k$	AbT$_2$	QT	HBT[3]	TBT	RCT	CST	CLT[4]	MLT	Fastest
4×8	1.29	1.29	1.23	**1.00**	1.26	1.33	1.40	1.64	1.40
4×16	1.27	1.36	1.24	**1.00**	1.23	1.24	2.16	1.59	2.16
4×24	1.25	1.17	1.11	**1.00**	1.14	1.24	2.10	1.30	2.10
4×32	1.30	1.15	1.11	**1.00**	1.31	1.29	2.07	1.44	2.07
4×64	1.41	1.45	1.26	**1.00**	2.05	1.52	4.33	1.74	1.00
8×8	2.04	1.28	1.12	**1.00**	1.16	1.35	1.08	1.71	1.08
8×16	2.56	1.58	1.24	**1.00**	1.26	1.49	1.94	1.93	1.94
8×24	2.27	1.45	1.25	**1.00**	1.18	1.32	2.94	1.61	2.94
8×32	2.25	1.32	1.10	**1.00**	1.18	1.32	2.01	1.72	2.01
8×64	2.64	1.34	1.22	**1.00**	1.32	1.34	3.01	2.00	1.00
16×8	4.39	1.34	1.26	**1.00**	1.13	1.31	1.27	2.14	1.27
16×16	5.28	1.58	1.23	**1.00**	1.21	1.44	1.19	2.17	1.19
16×24	4.36	1.38	1.22	**1.00**	1.22	1.42	2.04	2.42	2.04
16×32	5.14	1.25	1.13	**1.00**	1.27	1.40	2.04	2.42	1.40
16×64	5.16	1.49	1.28	**1.00**	1.44	1.45	1.98	2.17	1.98
32×8	7.23	1.21	1.06	**1.00**	1.03	1.29	1.13	3.01	1.13
32×16	8.53	1.35	1.05	**1.00**	1.17	1.35	1.14	3.18	1.14
32×24	7.94	1.32	1.12	**1.00**	1.18	1.32	1.89	3.25	1.89
32×32	8.88	1.26	1.08	**1.00**	1.28	1.36	1.65	3.06	1.65
32×64	13.17	1.54	1.32	**1.00**	1.50	1.50	2.04	3.38	3.38
64×8	16.21	1.55	1.21	**1.00**	1.13	1.42	1.23	5.08	1.13
64×16	20.03	1.48	1.23	**1.00**	1.23	1.34	1.20	4.68	1.23
64×24	17.77	1.26	1.19	**1.00**	1.23	1.32	1.99	4.98	1.99
64×32	21.01	1.40	1.13	**1.00**	1.30	1.37	1.66	5.53	1.66
64×64	24.74	1.46	1.28	**1.00**	1.42	1.36	1.94	6.68	1.94

area-efficiency. However, HBT is 1.10 times faster than TBT on the average. HBT is also is 1.43 times more area-efficient than its fastest competitor. Moreover, HBT is 6.33 and 1.15 times more area-efficient, on the average, than AbT and QT, respectively.

6.2.5. Area-Timing Product (ATP) Results

Synthesis tools sacrifice area to achieve shorter clock periods, and hence, area and timing costs are somewhat inversely proportional. Thus, we used area-timing product (ATP), also called area-delay product, as a figure of merit for area consumption per unit delay in order to make a reasonable comparison between competitors. Thus, we list (normalized) ATP results in Table 6.8, in addition to area results in Table 6.7. Similar to the area results given in Section 6.2.4, we take the ATP result of TBT as unity in each row of Table 6.8.

Regarding to ATP results in Table 6.8, QT is the best proposed topology and TBT is the the best existing topology on the average. QT is also the most ATP-efficient MF topology on the average. Besides, our proposed MF topologies have better ATP results than the competitors in 16 out of 25 cases of Table 6.8.

QT is 1.01 times more ATP-efficient than TBT as well as it is 1.58 times more ATP-efficient than the fastest competitor, on the average. Since QT is topologically equivalent to AbT for $n = 4$, we can claim that QT has the most ATP-efficient MF topology for $n = 4$. Similarly, QT has superior ATP results for $n = 64$. QT has also 4.48 and 1.09 times better ATP results, on the average, than AbT and HBT, respectively.

Moreover, HBT is the our second most ATP-efficient topology and it is 1.09 times less ATP-efficient than the TBT, which is the best competitor from the ATP point of view. On the other hand, HBT is 1.46 times more efficient than the fastest competitor in terms of ATP results. As HBT provides a generalization for PBTs, we can compare

[3]Results of the fastest topology in Table 6.6.

[4]Results of the fastest topology in Table 6.4.

Table 6.8. Normalized ATP results.

	Proposed Topologies			Competitor Topologies					
$n{\times}k$	AbT$_2$	QT	HBT[5]	TBT	RCT	CST	CLT[6]	MLT	Fastest
4×8	**0.96**	**0.96**	1.20	1.00	1.34	1.34	1.37	2.07	1.37
4×16	**0.87**	0.93	1.19	1.00	1.46	1.23	2.07	2.00	2.07
4×24	0.83	**0.78**	1.02	1.00	1.55	1.22	1.98	1.71	1.98
4×32	0.85	**0.76**	1.06	1.00	2.02	1.34	1.99	1.90	1.99
4×64	**0.92**	0.95	1.24	1.00	3.73	1.72	4.40	2.24	1.00
8×8	1.30	1.14	1.08	**1.00**	1.13	1.34	**1.00**	2.18	1.00
8×16	1.41	1.25	1.14	**1.00**	1.31	1.40	1.82	2.19	1.82
8×24	1.29	1.18	1.16	**1.00**	1.40	1.31	2.85	1.91	2.85
8×32	1.25	1.07	1.01	**1.00**	1.52	1.33	1.94	1.96	1.94
8×64	1.40	1.06	1.16	**1.00**	2.41	1.44	3.02	2.23	1.00
16×8	2.44	**0.97**	1.11	1.00	0.99	1.21	1.07	2.51	1.07
16×16	2.76	1.07	1.08	**1.00**	1.14	1.34	1.08	2.42	1.08
16×24	2.21	**0.92**	1.09	1.00	1.27	1.35	1.90	2.66	1.90
16×32	2.45	**0.83**	1.04	1.00	1.42	1.36	1.90	2.57	1.36
16×64	2.27	**0.92**	1.18	1.00	2.18	1.50	1.89	2.10	1.89
32×8	4.11	1.01	0.94	1.00	**0.85**	1.13	0.91	3.66	0.91
32×16	4.21	1.08	**0.94**	1.00	1.04	1.21	0.98	3.46	0.98
32×24	3.63	**1.00**	1.01	**1.00**	1.13	1.22	1.70	3.29	1.70
32×32	4.02	**0.94**	0.96	1.00	1.32	1.28	1.49	2.99	1.49
32×64	7.76	1.11	1.22	**1.00**	2.09	1.59	1.98	3.13	3.13
64×8	8.80	1.14	1.02	1.00	**0.85**	1.22	0.94	5.65	**0.85**
64×16	13.74	**0.99**	1.02	1.00	1.02	1.15	**1.00**	4.70	1.02
64×24	11.55	**0.84**	1.01	1.00	1.08	1.18	1.77	5.03	1.77
64×32	12.77	**0.91**	0.97	1.00	1.25	1.26	1.50	5.20	1.50
64×64	16.80	**0.92**	1.17	1.00	1.81	1.45	1.87	8.94	1.87

its ATP results with the ATP results of PBTs. In comparison to CST, which is the best PBT variant in terms of ATP, QT has 1.04 - 1.30 times (1.23 times on the average) better ATP-results. Besides, it is 1.36 and 1.62 times more ATP-efficient than RCT and CLT, respectively.

AbT is the least ATP-efficient MF topology as seen in Table 6.8. The ATP results of AbT are 2.80 times worse, on the average, than the fastest competitor's results and 4.43 times worse than TBT's results. On the contrary, ATP results of AbT are 1.88 and 1.12 times better than the ATP results of the fastest competitor and TBT, respectively, for $n = 4$.

6.2.6. Area-Timing-Square Product (AT2P) Results

Area-timing-square product (AT2P), also called area-delay-square product, is similar to ATP, but it stresses the timing objective more than the area. Since the performance of an MF circuit is more important than its area in our case, we can use AT2P as a figure of merit for area evaluation of the MF topologies examined in this thesis. Thus, we list (normalized) AT2P results in Table 6.9. Similar to the area results given in Section 6.2.4, we take the AT2P result of TBT as unity in each row of Table 6.9.

Regarding to AT2P results in Table 6.9, QT is the best proposed topology and TBT is the the best existing topology on the average. QT is also the most AT2P-efficient MF topology on the average. Besides, our proposed MF topologies have better AT2P results than the competitors in 22 out of 25 cases of Table 6.9.

QT is 1.39 times more AT2P-efficient than TBT as well as it is 2.03 times more AT2P-efficient than the fastest competitor, on the average. Since QT is topologically equivalent to AbT for $n = 4$, we can claim that QT has the most AT2P-efficient MF topology for $n = 4$ and $n = 16$. Furthermore, QT has superior AT2P results for 17

[5]Results of the fastest topology in Table 6.6.

[6]Results of the fastest topology in Table 6.4.

Table 6.9. Normalized AT2P results.

	Proposed Topologies			**Competitor Topologies**					
$n \times k$	AbT_2	QT	HBT^7	TBT	RCT	CST	CLT^8	MLT	Fastest
4×8	0.72	**0.70**	1.17	1.00	1.42	1.35	1.35	2.61	1.35
4×16	**0.59**	0.64	1.13	1.00	1.74	1.22	1.99	2.53	1.99
4×24	0.55	**0.54**	0.95	1.00	2.13	1.21	1.87	2.23	1.87
4×32	0.56	**0.52**	1.01	1.00	3.12	1.39	1.92	2.50	1.92
4×64	**0.60**	0.61	1.22	1.00	6.78	1.95	4.48	2.90	1.00
8×8	**0.83**	1.02	1.03	1.00	1.11	1.33	0.92	2.78	0.92
8×16	**0.78**	0.99	1.05	1.00	1.35	1.32	1.70	2.50	1.70
8×24	**0.74**	0.96	1.07	1.00	1.65	1.30	2.76	2.27	2.76
8×32	**0.70**	0.87	0.93	1.00	1.95	1.33	1.86	2.24	1.86
8×64	**0.74**	0.83	1.10	1.00	4.39	1.55	3.03	2.48	1.00
16×8	1.36	**0.71**	0.98	1.00	0.87	1.11	0.91	2.95	0.91
16×16	1.44	**0.73**	0.95	1.00	1.07	1.25	0.99	2.70	0.99
16×24	1.11	**0.62**	0.97	1.00	1.32	1.28	1.76	2.91	1.76
16×32	1.17	**0.55**	0.95	1.00	1.59	1.31	1.84	2.73	1.31
16×64	1.00	**0.57**	1.08	1.00	3.31	1.55	1.80	2.03	1.80
32×8	2.34	0.85	0.84	1.00	**0.70**	0.99	0.73	4.46	0.73
32×16	2.08	0.86	**0.84**	1.00	0.93	1.09	**0.84**	3.76	**0.84**
32×24	1.66	**0.76**	0.90	1.00	1.09	1.12	1.53	3.33	1.53
32×32	1.82	**0.70**	0.86	1.00	1.35	1.21	1.36	2.91	1.36
32×64	4.58	**0.81**	1.12	1.00	2.91	1.69	1.91	2.90	2.90
64×8	4.78	0.84	0.87	1.00	**0.64**	1.04	0.72	6.29	**0.64**
64×16	9.42	**0.66**	0.84	1.00	0.84	0.99	0.84	4.72	0.84
64×24	7.51	**0.56**	0.86	1.00	0.94	1.05	1.49	5.09	1.49
64×32	7.76	**0.60**	0.83	1.00	1.21	1.16	1.32	4.90	1.32
64×64	11.41	**0.58**	1.07	1.00	2.31	1.54	1.81	11.96	1.81

out of 25 cases. QT has also 3.68 and 1.37 times better AT2P results, on the average, than AbT and HBT, respectively.

Moreover, HBT is the our second most ATP-efficient topology and it is 1.01 times more AT2P-efficient than the TBT, which is the best competitor from the AT2P point of view. On the other hand, HBT is 1.47 times more efficient than the fastest competitor in terms of AT2P results. As HBT provides a generalization for PBTs, we can compare its AT2P results with the AT2P results of PBTs. In comparison to CST, which is the best PBT variant in terms of AT2P, QT has 1.07 - 1.59 times (1.32 times on the average) better AT2P results. Besides, it is 1.82 and 1.65 times more AT2P-efficient than RCT and CLT, respectively.

AbT is the least AT2P-efficient MF topology as seen in Table 6.9. The AT2P results of AbT are 1.82 times worse, on the average, than the fastest competitor's results and 2.65 times worse than TBT's results. On the contrary, AT2P results of AbT are 2.48 and 1.46 times better than the AT2P results of the fastest competitor and TBT, respectively, for $n < 16$.

6.2.7. Power Consumption Results

Table 6.10 shows the normalized dynamic power consumption results of our proposed topologies as well as their competitors. Although we also obtained the leakage power consumption results, we do not show them as they are negligible in comparison to dynamic power consumption results. We take the results of TBT as unity and scale other values in regards to this value.

As seen in Table 6.8, TBT has the best power consumption results in all cases, except the case of ($n = 4$,$k = 24$), in which AbT has the best power consumption value. On the average, HBT is the best proposed topology and TBT is the best competitor topology in terms of power consumption.

[7]Results of the fastest topology in Table 6.6.

[8]Results of the fastest topology in Table 6.4.

Table 6.10. Normalized power consumption results.

	Proposed Topologies			**Competitor Topologies**					
$n \times k$	AbT_2	QT	HBT[9]	TBT	RCT	CST	CLT[10]	MLT	Fastest
4×8	1.11	1.12	1.22	**1.00**	1.30	1.53	1.40	1.76	1.40
4×16	1.08	1.20	1.33	**1.00**	1.14	1.56	2.49	1.75	2.49
4×24	**0.99**	1.07	1.28	1.00	1.09	1.41	2.52	1.59	2.52
4×32	1.02	1.07	1.32	**1.00**	1.28	1.49	2.59	1.83	2.59
4×64	1.16	1.17	1.30	**1.00**	1.79	1.48	4.87	1.82	**1.00**
8×8	1.78	1.21	1.20	**1.00**	1.29	1.37	1.11	2.19	1.11
8×16	1.78	1.41	1.26	**1.00**	1.25	1.44	2.17	2.33	2.17
8×24	1.96	1.39	1.43	**1.00**	1.17	1.42	3.89	2.06	3.89
8×32	1.89	1.31	1.28	**1.00**	1.18	1.39	2.50	2.36	2.50
8×64	2.26	1.14	1.27	**1.00**	1.12	1.28	3.17	2.58	**1.00**
16×8	4.09	1.41	1.48	**1.00**	1.36	1.54	1.37	2.74	1.37
16×16	4.43	1.67	1.34	**1.00**	1.19	1.46	1.13	2.85	1.13
16×24	4.05	1.47	1.46	**1.00**	1.28	1.57	2.58	3.59	2.58
16×32	3.99	1.23	1.24	**1.00**	1.19	1.52	2.58	3.30	1.52
16×64	4.19	1.37	1.34	**1.00**	1.25	1.38	1.97	2.57	1.97
32×8	7.83	1.36	1.33	**1.00**	1.29	1.59	1.15	4.54	1.15
32×16	8.05	1.44	1.21	**1.00**	1.22	1.34	1.07	4.46	1.07
32×24	8.01	1.65	1.55	**1.00**	1.49	1.56	2.69	5.16	2.69
32×32	8.14	1.32	1.29	**1.00**	1.26	1.41	2.30	4.17	2.30
32×64	16.48	1.49	1.43	**1.00**	1.36	1.47	2.10	4.16	4.16
64×8	10.93	1.72	1.34	**1.00**	1.24	1.63	1.08	7.01	1.24
64×16	22.79	1.52	1.32	**1.00**	1.22	1.17	1.01	5.86	1.22
64×24	23.20	1.31	1.29	**1.00**	1.18	1.19	2.41	6.92	2.41
64×32	31.28	1.69	1.46	**1.00**	1.27	1.53	1.96	8.61	1.96
64×64	37.72	1.40	1.35	**1.00**	1.27	1.29	1.96	12.67	1.96

HBT, the best proposed topology in terms of power, consumes 1.33 times more power than TBT, while it consumes 1.49 times less power than the fastest competitor. In addition, it is 6.33 and 1.02 times more power-efficient on the average than AbT and QT, respectively.

QT is the second most power-efficient proposed topology and consumes 1.36 times more power than TBT on the average. QT is the most power-efficient topology up to $n = 8$ as well. QT is also 1.46 times less power-consuming than the fastest competitor.

AbT is the most power-consuming MF topology on the average, as seen in Table 6.8. AbT consumes 4.25 and 8.41 times more power than the fastest competitor and TBT, respectively. On the other hand, AbT is 1.86 times better than the fastest competitor in terms of power consumption, while it is only 1.07 times worse than TBT for $n = 4$. Moreover, AbT is the most power-efficient proposed topology on the average for $n = 4$.

6.2.8. Power-Timing Product (PTP) Results

Power-timing product (PTP), also called power-delay product, is another combined performance metric like ATP and AT2P. PTP is a balanced measure that considers both the speed and power consumption factors of a design. We can use PTP as a figure of merit for power efficiency of a circuit. Thus, we list (normalized) PTP results in Table 6.11. Similar to the area results given in Section 6.2.4, we take the PTP result of TBT as unity in each row of Table 6.11.

Regarding to PTP results in Table 6.11, QT is the best proposed topology and TBT is the the best existing topology on the average. QT is also the most PTP-efficient MF topology on the average.

QT is 1.01 times more PTP-efficient than TBT as well as it is 1.84 times more

[9]Results of the fastest topology in Table 6.6.
[10]Results of the fastest topology in Table 6.4.

Table 6.11. Normalized PTP results.

	Proposed Topologies			**Competitor Topologies**					
$n \times k$	AbT$_2$	QT	HBT[11]	TBT	RCT	CST	CLT[12]	MLT	Fastest
4×8	0.83	**0.82**	1.19	1.00	1.38	1.54	1.38	2.22	1.38
4×16	**0.74**	0.82	1.27	1.00	1.35	1.54	2.38	2.21	2.38
4×24	**0.66**	0.72	1.19	1.00	1.49	1.40	2.38	2.08	2.38
4×32	**0.67**	0.72	1.26	1.00	1.97	1.54	2.49	2.41	2.49
4×64	**0.76**	0.76	1.28	1.00	3.26	1.67	4.95	2.35	1.00
8×8	1.13	1.07	1.15	**1.00**	1.26	1.36	1.02	2.78	1.02
8×16	**0.98**	1.12	1.16	1.00	1.29	1.35	2.03	2.65	2.03
8×24	1.11	1.14	1.32	**1.00**	1.38	1.40	3.77	2.44	3.77
8×32	1.05	1.06	1.18	**1.00**	1.52	1.40	2.41	2.70	2.41
8×64	1.20	**0.90**	1.20	1.00	2.04	1.38	3.17	2.87	1.00
16×8	2.28	1.02	1.30	**1.00**	1.20	1.42	1.15	3.21	1.15
16×16	2.31	1.13	1.18	**1.00**	1.12	1.36	1.03	3.18	1.03
16×24	2.05	**0.98**	1.31	1.00	1.33	1.48	2.40	3.94	2.40
16×32	1.91	**0.82**	1.14	1.00	1.34	1.47	2.50	3.51	1.47
16×64	1.84	**0.84**	1.24	1.00	1.90	1.43	1.88	2.49	1.88
32×8	4.46	1.14	1.18	1.00	1.07	1.39	**0.93**	5.53	**0.93**
32×16	3.97	1.15	1.09	1.00	1.08	1.20	**0.92**	4.85	**0.92**
32×24	3.66	1.25	1.39	**1.00**	1.43	1.44	2.42	5.22	2.42
32×32	3.68	**0.98**	1.15	1.00	1.30	1.32	2.10	4.07	2.10
32×64	9.72	1.07	1.32	**1.00**	1.89	1.56	2.04	3.86	3.86
64×8	5.94	1.27	1.14	1.00	0.93	1.40	**0.82**	7.80	0.93
64×16	15.63	1.02	1.09	1.00	1.01	1.01	**0.84**	5.89	1.01
64×24	15.08	**0.87**	1.10	1.00	1.04	1.06	2.02	6.99	2.02
64×32	19.01	1.10	1.25	**1.00**	1.22	1.41	1.73	8.10	1.73
64×64	25.61	**0.88**	1.23	1.00	1.62	1.37	1.90	16.95	1.90

PTP-efficient than the fastest competitor, on the average. Since QT is topologically equivalent to AbT for $n = 4$, we can claim that QT has the most PTP-efficient MF topology for $n = 4$. QT has also 5.10 and 1.22 times better PTP results, on the average, than AbT and HBT, respectively.

Moreover, HBT is the our second most ATP-efficient topology and it is 1.21 times less PTP-efficient than the TBT, which is the best competitor from the PTP point of view. On the other hand, HBT is 1.50 times more efficient than the fastest competitor in terms of PTP results. As HBT provides a generalization for PBTs, we can compare its PTP results with the PTP results of PBTs. In comparison to CST, which is the best PBT variant in terms of PTP, QT has 1.15 times on the average better PTP results. Besides, it is 1.20 and 1.65 times more PTP-efficient than RCT and CLT, respectively.

AbT is the least PTP-efficient MF topology as seen in Table 6.11. The PTP results of AbT are 2.78 times worse, on the average, than the fastest competitor's results and 5.05 times worse than TBT's results. On the contrary, PTP results of AbT are 2.70 and 1.37 times better than the PTP results of the fastest competitor and TBT, respectively, for $n = 4$.

6.2.9. Energy-Timing Product (ETP) Results

Energy-timing product (ETP), also called energy-delay product, considers both the timing and energy consumption of a VLSI circuit simultaneously. Thus, we can use ETP as a figure of merit for energy efficiency of an MF circuit. We give (normalized) ETP results in Table 6.12. Similar to the area results given in Section 6.2.4, we take the PTP result of TBT as unity in each row of Table 6.12.

Regarding to ETP results in Table 6.12, QT is the best proposed topology and TBT is the the best existing topology on the average. QT is also the most ETP-efficient MF topology on the average. Besides, our proposed MF topologies have better ETP

[11]Results of the fastest topology in Table 6.6.

[12]Results of the fastest topology in Table 6.4.

Table 6.12. Normalized ETP results.

	Proposed Topologies			**Competitor Topologies**					
$n \times k$	AbT$_2$	QT	HBT[13]	TBT	RCT	CST	CLT[14]	MLT	Fastest
4×8	0.62	**0.61**	1.16	1.00	1.47	1.55	1.35	2.79	1.35
4×16	**0.50**	0.56	1.21	1.00	1.61	1.53	2.29	2.79	2.29
4×24	**0.44**	0.49	1.10	1.00	2.04	1.38	2.25	2.72	2.25
4×32	**0.44**	0.48	1.20	1.00	3.05	1.60	2.39	3.17	2.39
4×64	0.50	**0.49**	1.26	1.00	5.92	1.89	5.03	3.04	1.00
8×8	**0.72**	0.96	1.11	1.00	1.24	1.35	0.94	3.55	0.94
8×16	**0.54**	0.89	1.06	1.00	1.34	1.28	1.90	3.02	1.90
8×24	**0.63**	0.92	1.23	1.00	1.63	1.39	3.66	2.89	3.66
8×32	**0.59**	0.86	1.09	1.00	1.96	1.40	2.32	3.08	2.32
8×64	**0.64**	0.71	1.14	1.00	3.71	1.48	3.18	3.20	1.00
16×8	1.27	**0.74**	1.15	1.00	1.05	1.30	0.98	3.76	0.98
16×16	1.21	**0.77**	1.04	1.00	1.06	1.27	0.94	3.54	0.94
16×24	1.03	**0.66**	1.17	1.00	1.39	1.41	2.23	4.32	2.23
16×32	0.91	**0.54**	1.04	1.00	1.50	1.42	2.42	3.72	1.42
16×64	0.81	**0.52**	1.14	1.00	2.89	1.48	1.80	2.40	1.80
32×8	2.54	0.96	1.04	1.00	0.89	1.22	**0.75**	6.73	**0.75**
32×16	1.96	0.92	0.97	1.00	0.97	1.08	**0.79**	5.27	**0.79**
32×24	1.68	**0.95**	1.25	1.00	1.38	1.33	2.18	5.28	2.18
32×32	1.67	**0.73**	1.03	1.00	1.33	1.25	1.92	3.97	1.92
32×64	5.73	**0.78**	1.22	1.00	2.64	1.66	1.97	3.57	3.57
64×8	3.22	0.93	0.97	1.00	0.70	1.20	**0.63**	8.69	0.70
64×16	10.72	**0.68**	0.90	1.00	0.84	0.87	0.70	5.92	0.84
64×24	9.80	**0.58**	0.94	1.00	0.91	0.95	1.69	7.07	1.69
64×32	11.55	**0.72**	1.07	1.00	1.18	1.30	1.52	7.63	1.52
64×64	17.39	**0.55**	1.12	1.00	2.06	1.45	1.83	22.68	1.83

results than the competitors in 22 out of 25 cases of Table 6.8.

QT is 1.39 times more ETP-efficient than TBT as well as it is 2.35 times more ETP-efficient than the fastest competitor, on the average. Since QT is topologically equivalent to AbT for $n = 4$, we can claim that QT has the most ETP-efficient MF topology for $n = 4$ and $n = 16$. QT has also 4.28 and 1.53 times better ATP results, on the average, than AbT and HBT, respectively.

Moreover, HBT is the our second most ETP-efficient topology and it is 1.10 times less ETP-efficient than the TBT, which is the best competitor from the ETP point of view. On the other hand, HBT is 1.54 times more efficient than the fastest competitor in terms of ETP results. As HBT provides a generalization for PBTs, we can compare its ETP results with the ETP results of PBTs. In comparison to CST, which is the best PBT variant in terms of ETP, QT has 1.23 times, on the average, better ETP results. Besides, it is 1.58 and 1.69 times more ETP-efficient than RCT and CLT, respectively.

AbT is the least ETP-efficient MF topology as seen in Table 6.12. The ETP results of AbT are 1.82 times worse, on the average, than the fastest competitor's results and 3.08 times worse than TBT's results. On the contrary, ETP results of AbT are 2.92 and 1.79 times better than the ETP results of the fastest competitor and TBT, respectively, for $n < 16$.

[13]Results of the fastest topology in Table 6.6.

[14]Results of the fastest topology in Table 6.4.

7. CONCLUSIONS AND FUTURE WORK

In this thesis, we examined the existing circuit topologies (AT, RT, ST, TBT, RCT, CST, CLT, MLT, and LCT), while we proposed three circuit topologies (AbT, HBT, QT), which compute the value and address of the maximum of n k-bit numbers.

We compared the proposed topologies (AbT, HBT, QT) with their competitors (TBT, RCT, CST, CLT, MLT, and LCT) based on timing, area, ATP, and power dissipation. We use an automated synthesis methodology (Chapter 5), which consists of a main synthesis script carrying out the synthesis process, and a set of Verilog code generators (written in Perl) to implement all of these MF circuit topologies. Our code generators, which are scalable to any value of n and k, produce RTL and self-checking testbench code for the proposed topologies as well as their competitors.

In regards to the synthesis results in Section 6.2.1, AbT is the best topology in terms of timing results (latency) for all cases. Besides, AbT can produce the value of maximum element in addition to its address with little delay overhead due to our parallel DSU structure. On the average, the second and third fastest topologies are QT and HBT, respectively. In addition, the fastest competitor is CLT on the average.

As mentioned in Section 6.2.3, we wrote an estimator script in Perl to estimate the timing complexities of all possible implementations (with different p values) of an (n,k)-*HBT* for a specific (n,k) value. Using the estimator, we eliminate the need to synthesize all possible HBT implementations to find the fastest implementation for a specific (n,k) value.

The most area-efficient topology on the average is TBT, which is also the most obvious and widely used competitor topology. On the other hand, the best proposed topology is HBT and the worst proposed topology is AbT in terms of area. Although it has the best timing results, AbT has a significant area overhead for $n > 8$. Fortunately, we can overcome this disadvantage by using QT, which is the most ATP-efficient MF topology. Using QT for $n > 8$, we can save 9.06 times more area with the cost of 1.28 times lower speed (i.e., more latency) in comparison to AbT. On the other hand, QT is still 1.21 times faster than the fastest competitor on the average. Moreover, HBT has better area than all of the existing PBT variants on the average.

In terms of power consumption, TBT is the best MF topology, while HBT is the best proposed topology on the average. Our second least power-consuming proposed topology is QT, whose power consumption is very close to power consumption of HBT. The most power-consuming topology is AbT, which is the fastest topology.

Taking into account of the ATP and PTP results, QT, a proposed topology, is the best MF topology on the average. The second best topology is TBT from the ATP as well as PTP point of view. On the other hand, the least ATP-efficient and the least PTP-efficient topology is AbT, which is the fastest topology, on the average. Interestingly, for $n = 4$, AbT is the best MF topology in terms of both ATP and PTP results. Our second most ATP-efficient and PTP-efficient proposed topology, HBT, has better ATP and PTP results than all of the existing PBT variants on the average as well.

As it can be seen from Sections 6.2.6 and 6.2.9, AT2P and ETP results are similar to each other. Considering AT2P and ETP, QT is the best MF topology on the average while TBT is the second best MF topology. Although AbT is the worst MF topology on the average in terms of both AT2P and ETP, it is the best MF topology for $n < 16$. Our HBT is the third best MF topology on the average in regards to AT2P as well as ETP.

Our future work will possibly involve synthesis of AbT with other configurations given in Table 4.1. Moreover, looking for more area-efficient implementations, such as using a hierarchical approach to reduce size of AbT's CAU or using simpler comparators, is another possible work. Obviously, we should work on our estimator tool to improve its accuracy and precision. We may also write an estimator tool, which estimates the best MF topology for a specific (n,k) case. Proposed architecture and its competitors may also be synthesized on FPGAs.

Another important future work is examining the usage of proposed MF topologies in selection and sorting topologies. Here, we shall look over the possible usage of AbT as an example. In Section 4.1.5, we claimed that AbT can be used to compute i-th order statistic of a set $\boldsymbol{D}$ of n k-bit numbers, and we gave an example topology for this purpose in Figure 4.6. AbT can also produce multiple order statistics of the set $\boldsymbol{D}$ without any extra delay, as shown in Figure 7.1. Adding an extra order statistic brings an area overhead of $\mathrm{O}(n\,(k+\log_2 n))$. Thanks to its ability of computing

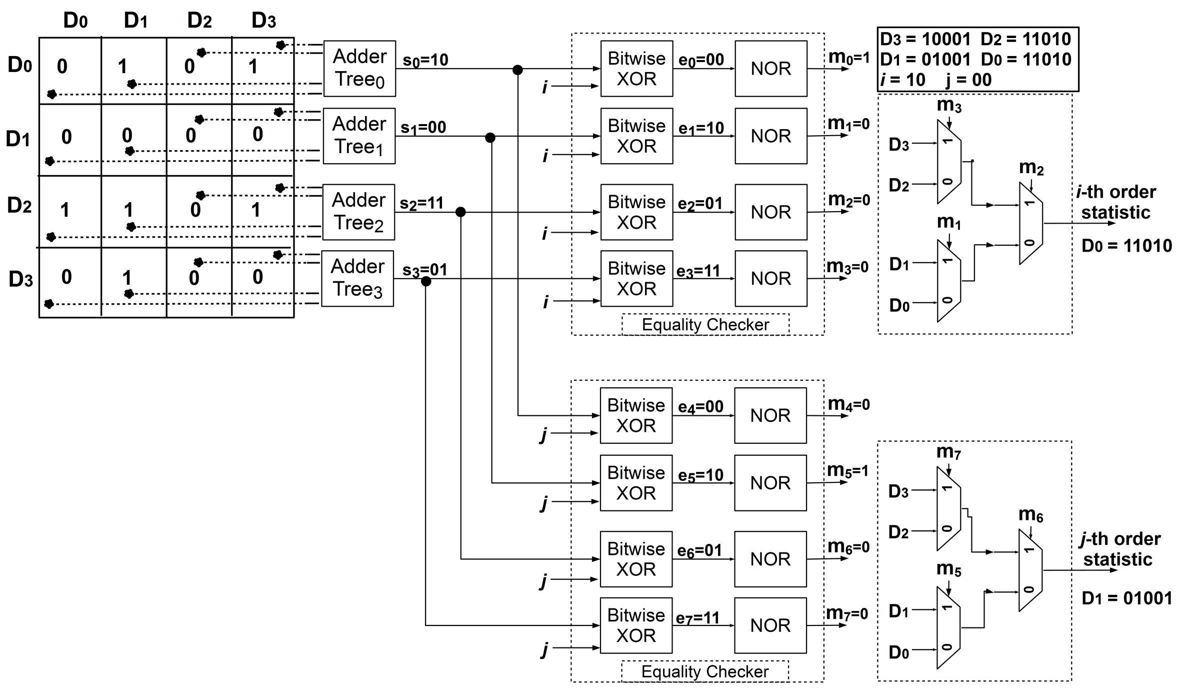


Figure 7.1. Computing multiple order statistics with AbT.

multiple order statistics, we can also use AbT as a subcircuit in sorting networks such as bitonic sorter [25], which is one of the fastest sorting networks. A bitonic sequence of numbers can be defined as a concatenation of a non-decreasing monotonic sequence and a non-increasing monotonic sequence. And if we apply a circular rotation operation on a bitonic sequence, the result is also a bitonic sequence. An (n,k)-bitonic sorter repeatedly merges shorter bitonic sequences to obtain larger bitonic sequences. It starts with one-element bitonic sequences (n inputs) and finally comes up with an

n-element (sorted) bitonic sequence. As an example, we give a bitonic sorter, which sorts 8 k-bit numbers in Figure 7.2. An (n,k)-bitonic sorter consists of $\log_2 n$ stages of

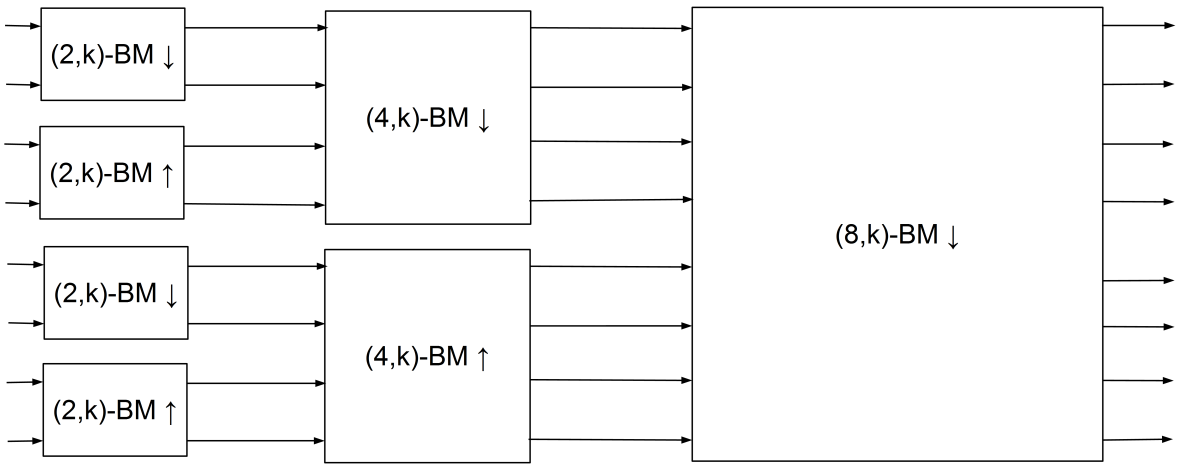


Figure 7.2. A (8,k)-bitonic sorter circuit.

binary merger (BM) blocks. The i-th stage of the sorter consists of $n/2^i$ BM blocks with the size of 2^i. As you can see from Figure 7.2, we have two kinds of BM blocks: *increasing* BM (with up arrow) and *decreasing* BM (with down arrow). Increasing BM sorts the inputs in ascending order while decreasing BM sorts the inputs in decreasing order. The vertical arrows in Figure 7.2 point toward to larger number.

The traditional implementation of a (p,k)-BM block consists of $\log_2 p$ stages of p (2,k)-cmp-mux circuits. Hence, total timing and area complexities of a traditional (n,k)-bitonic sorter are $O((\log_2 n)^2 \log_2 k)$ and $O(k\, n\, (\log_2 n)^2)$ respectively. If we implement a (p,k)-BM with the AbT topology shown in Figure 7.1, we obtain an (n,k)-bitonic sorter with the timing and area complexities of $O((\log_2 n)^2 + \log_2 n \log_2 k)$ and $O(k\, n^2)$, respectively. Note also that we need only $p^2/8$ comparators in CAU of a (p,k)-BM, instead of $p\,(p+1)/2$ comparators, since our input is a bitonic sequence. Furthermore, we can use hybrid BM blocks, which are combination of the traditional and AbT-based implementations, to reduce the area overhead of using AbT.

REFERENCES

1. Vai, M. and M. Moy, “Real-Time Maximum Value Determination on an Easily Testable VLSI Architecture”, *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, Vol. 40, No. 4, pp. 283–285, 1993.

2. Vinnakota, B. and V. Bapeswara Rao, “A New Circuit for Maximum Value Determination”, *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, Vol. 41, No. 12, pp. 929–930, 1994.

3. Daneshgaran, F. and K. Yao, “Efficient Parallel Pipelinable VLSI Architecture for Finding the Maximum Binary Number”, *IEE Proceedings on Circuits, Devices and Systems*, Vol. 141, No. 6, pp. 527–534, 1994.

4. Harteros, K., *Fast Parallel Comparison Circuits for Scheduling*, Technical Report 304, University of Crete, FORTH-ICS, 2002, http://archvlsi.ics.forth.gr/muqpro/cmpTree.html, accessed at May 2013.

5. Huang, X.-P., X.-Y. Fan, S.-B. Zhang and F. Zhang, “An Optimized Tag Sorting Circuit in WFQ Scheduler Based on Leading Zero Counting”, *Proceedings of the IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pp. 533–535, Shanghai, China, 2010.

6. del Campo, F. M., R. Cumplido, R. Perez-Andrade and A. Orozco-Lugo, “A System on a Programmable Chip Architecture for Data-Dependent Superimposed Training Channel Estimation”, *International Journal of Reconfigurable Computing*, Vol. 2009, Article ID 912301, 10 pages, 2009.

7. Huang, Y.-W., S.-Y. Chien, B.-Y. Hsieh and L.-G. Chen, “Global Elimination Algorithm and Architecture Design for Fast Block Matching Motion Estimation”, *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 14, No. 6, pp. 898–907, 2004.

8. Yang, M., H. Selvaraj, E. Lu, J. Wang, S. Q. Zheng and Y. Jiang, "Scheduling Architectures for DiffServ Networks with Input Queuing Switches", *Electronics and Telecommunications Quarterly*, Vol. 55, No. 1, pp. 9–30, 2009.

9. Kohout, P., B. Ganesh and B. Jacob, "Hardware Support for Real-Time Operating Systems", *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 45–51, California, USA, 2003.

10. Azuma, S., T. Sakuma, T. Nakano, T. Ando and K. Shirai, "High-Performance Sort Chip", *Proceedings of the Annual International Symposium on High Performance Chips (Hot Chips)*, California, USA, 1999.

11. Ugurdag, H. F. and O. Baskirt, "Fast Parallel Prefix Logic Circuits for n2n Round-Robin Arbitration", *Microelectronic Journal*, Vol. 43, No. 8, pp. 573–581, 2012.

12. Ugurdag, H. F., F. Temizkan, O. Baskirt and B. Yuce, "Fast two-pick n2n round-robin arbiter circuit", *Electronics Letters*, Vol. 48, No. 13, pp. 759–760, 2012.

13. Yuce, B., H. F. Ugurdag, S. Goren and G. Dundar, "A fast circuit topology for finding the maximum of n k-bit numbers", *Proceedings of the IEEE International Symposium on Computer Arithmetic (ARITH)*, pp. 59–66, Texas, USA, 2013.

14. Hendry, D., "Comparator trees for winner-take-all Circuits", *Elsevier Neurocomputing*, Vol. 62, pp. 389–403, 2004.

15. Brent, R. P. and H. T. Kung, "A Regular Layout for Parallel Adders", *IEEE Transactions on Computers*, Vol. C-31, No. 3, pp. 260–264, 1982.

16. Han, T. and D. Carlson, "Fast Area-efficient VLSI Adders", *Proceedings of the IEEE International Symposium on Computer Arithmetic (ARITH)*, pp. 49–56, Como, Italy, 1987.

17. Kogge, P. M. and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a

General Class of Recurrence Equations", *IEEE Transactions on Computers*, Vol. C-22, No. 8, pp. 786–793, 1973.

18. Ladner, R. E. and M. J. Fischer, "Parallel Prefix Computation", *Journal of the ACM*, Vol. 27, No. 4, pp. 831–838, oct 1980.

19. Zimmermann, R., *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*, Ph.D. Thesis, Swiss Federal Institute of Technology, 1997.

20. Tyagi, A., "A Reduced-Area Scheme for Carry-Select Adders", *IEEE Transactions on Computers*, Vol. 42, No. 10, pp. 1163–1170, 1993.

21. Wang, L.-T., Y.-W. Chang and K.-T. Cheng, *Electronic Design Automation : Synthesis, verification, and test*, Morgan Kaufmann/Elsevier, Boston, MA, USA, 2009.

22. Ullman, J. D., *Computational Aspects of VLSI*, W. H. Freeman & Co., New York, NY, USA, 1984.

23. Veeramachaneni, S., M. Krishna, L. Avinash, P. Reddy and M. Srinivas, "Efficient Design of 32-bit Comparator Using Carry Look-ahead Logic", *IEEE Northeast Workshop on Circuits and Systems (NEWCAS)*, pp. 867–870, Montreal, Canada, 2007.

24. Cormen, T. H., C. Stein, R. L. Rivest and C. E. Leiserson, *Introduction to Algorithms*, McGraw-Hill Higher Education, 2nd edn., 2001.

25. Batcher, K. E., "Sorting Networks and Their Applications", *Proceedings of the Spring Joint Computer Conference*, pp. 307–314, New Jersey, USA, 1968.

26. Boese, K. D., A. B. Kahng and S. Mantik, "On the Relevance of Wire Load Models", *Proceedings of the ACM/IEEE International Workshop on System Level Interconnect Prediction*, pp. 91–98, California, USA, 2001.