Weikang Qian, Marc D. Riedel, Kia Bazargan,
and David J. Lilja

Synthesizing Combinational Logic to Generate Probabilities: Theories and Algorithms*

September 29, 2010

Springer

*This work is supported by a grant from the Semiconductor Research Corporation's Focus Center Research Program on Functional Engineered Nano-Architectonics, contract No. 2003-NT-1107.

Contents

List of Tables

List of Figures

Chapter 1
Synthesizing Combinational Logic to Generate Probabilities: Theories and Algorithms

1.1 Chapter Overview

As CMOS devices are scaled down into the nanometer regime, concerns about reliability are mounting. Instead of viewing nano-scale characteristics as an impediment, technologies such as PCMOS exploit them as a source of randomness. The technology generates random numbers that are used in probabilistic algorithms. With the PCMOS approach, different voltage levels are used to generate different probability values. If many different probability values are required, this approach becomes prohibitively expensive.

In this chapter, we demonstrate a novel technique for synthesizing logic that generates new probabilities from a given set of probabilities. We focus on synthesizing combinational logic to generate arbitrary *decimal* probabilities from a given set of input probabilities. We demonstrate how to generate arbitrary decimal probabilities from small sets – a single probability or a pair of probabilities – through combinational logic.

The remainder of this chapter is organized as follows: Section 1.2 introduces the problem of synthesizing combinational logic to generate decimal probabilities. Section 1.3 describes related work. Sections 1.4 and 1.5 show the existence of a pair of probabilities and of a single probability, respectively, that can be used as input sources to generate arbitrary decimal probabilities. Section 1.6 describes our implementation and presents algorithms for optimizing the resulting circuits. Section 1.7 demonstrates the effectiveness of the proposed algorithms. Finally, Section 1.8 summarizes this chapter.

1.2 Introduction and Background

It can be argued that the entire success of the semiconductor industry has been predicated on a single, fundamental abstraction, namely, that digital computation consists of a deterministic sequence of zeros and ones. From the logic level up, the precise Boolean functionality of a circuit is prescribed; it is up to the physical layer to produce voltage values that can be interpreted as the exact logical values that are called for. This abstraction delivers all the benefits of the digital paradigm: precision, modularity, extensibility. And yet, as circuits are scaled down into the nanometer regime, delivering the physical circuits underpinning the abstraction is increasingly costly and challenging. Power consumption is a major concern [6]. Also, soft errors caused by ionizing radiation are a problem, particularly for circuits operating in harsh environments [1].

We advocate a novel view for digital computation: instead of transforming definite inputs into definite outputs – say, Boolean, integer, or real values into the same – we design circuits that transform probability values into probability values; so, conceptually, real-valued probabilities are both the inputs and the outputs. The circuits process random bit streams; these are digital, consisting of zeros and ones; they are processed by ordinary logic gates, such as AND and OR. The inputs and outputs are encoded through the statistical distribution of the signals instead of specific values. When cast in terms of probabilities, the computation is robust [10].

The topic of computing probabilistically dates back to von Neumann [9]. Many flavors of probabilistic design have been proposed for circuit-level constructs. For instance, [8] presents a design methodology based on Markov random fields, geared toward nanotechnology. Recent work on *probabilistic* CMOS (PCMOS) is a promising approach. Instead of viewing variable circuit characteristics as an impediment, PCMOS exploits them as a source of randomness. The technology generates random numbers that are used in probabilistic algorithms [3].

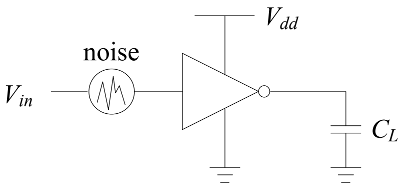

Fig. 1.1: A PCMOS switch. It consists of an inverter with its input coupled to a noise source.

A PCMOS switch is an inverter with the input coupled to a noise source, as shown in Figure 1.1. With the input V_{in} set to 0 volts, the output of the inverter has a certain probability p $(0 \leq p \leq 1)$ of being at logical one. Suppose that the probability density function of the noise voltage V is $f(V)$ and that the trip point of the inverter is $V_{dd}/2$, where V_{dd} is the supply voltage. Then, the probability that the output is one equals the probability that the input to the inverter is below $V_{dd}/2$, or

$$p = \int_{-\infty}^{V_{dd}/2} f(V)\,\mathrm{d}V,$$

which corresponds to the shaded area in Figure 1.2. Thus, with a given noise distribution, p can be modulated by changing V_{dd}.

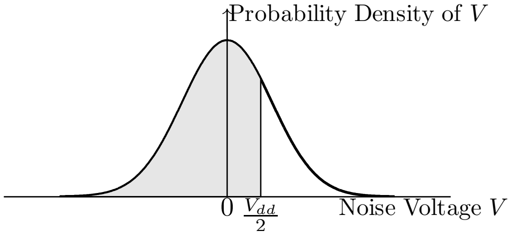

Fig. 1.2: Probability density function of the noise source. The probability that the output of the PCMOS switch is one equals the shaded area in the figure. Changing V_{dd} will change this probability.

In [2], PCMOS switches are applied to form a probabilistic system-on-a-chip (PSOC) architecture that is used to execute probabilistic algorithms. In essence, the PSOC architecture consists of a host processor that executes the deterministic part of the algorithm, and a coprocessor built with PCMOS switches that executes the probabilistic part of the algorithm. The PCMOS switches in the coprocessor are configured to realize the set of probabilities needed by the algorithm. This approach achieves an energy-performance-product improvement over conventional architectures for some probabilistic algorithms.

However, as is pointed out in [2], a serious problem must be overcome before PCMOS can become a viable design strategy for many applications: since the probability p for each PCMOS switch is controlled by a specific voltage level, different voltage levels are required to generate different probability values. For an application that requires many different probability values, many voltage regulators are required; this is costly in terms of area as well as energy.

In this chapter, we present a synthesis strategy to mitigate this issue: we describe a method for transforming probability values from a small set to many different probability values entirely through combinational logic. For what follows, when we say "with probability p," we mean "with a probability p of being at logical one." When we say "a circuit," we mean a combinational circuit built with logic gates.

Example 1.1. Suppose that we have a set of probabilities $S = \{0.4, 0.5\}$. As illustrated in Figure 1.3, we can generate new probabilities from this set:

1. An inverter with an input x with probability 0.4 will have output z with probability 0.6 since for an inverter,

$$P(z = 1) = P(x = 0) = 1 - P(x = 1). \tag{1.1}$$

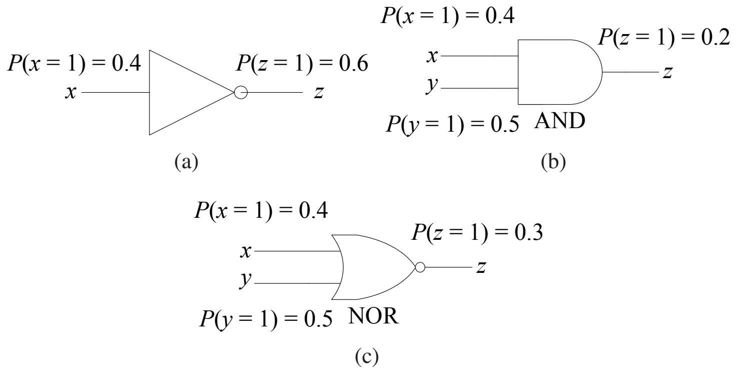

Fig. 1.3: An illustration of generating new probabilities from a given set of probabilities through logic. (a): An inverter implementing $p_z = 1 - p_x$. (b): An AND gate implementing $p_z = p_x \cdot p_y$. (c): A NOR gate implementing $p_z = (1 - p_x) \cdot (1 - p_y)$.

2. An AND gate with inputs x and y with *independent* probabilities 0.4 and 0.5, respectively, will have an output z with probability 0.2 since for an AND gate,

$$P(z = 1) = P(x = 1, y = 1) = P(x = 1)P(y = 1). \tag{1.2}$$

3. A NOR gate with inputs x and y with *independent* probabilities 0.4 and 0.5, respectively, will have output z with probability 0.3 since for a NOR gate,

$$\begin{aligned} P(z = 1) &= P(x = 0, y = 0) = P(x = 0)P(y = 0) \\ &= (1 - P(x = 1))(1 - P(y = 1)). \end{aligned} \tag{1.3}$$

Thus, using only combinational logic, we can get the additional set of probabilities $\{0.2, 0.3, 0.6\}$. □

Motivated by this example, we consider the problem of how to synthesize combinational logic to generate a required probability q from a given set of probabilities $S = \{p_1, p_2, \ldots, p_n\}$. Specifically, we focus on synthesizing arbitrary decimal probabilities (i.e., q is a decimal number). We assume that the probabilities in a set S can be freely chosen and each element in S can be used as the input probability any number of times. (We say that the probability is *duplicable*.) The problem is to find a good set S such that, for an arbitrary decimal probability, we can construct a circuit to generate it.

As a result, in Section 1.4, we will show that there exist sets consisting of two elements that can be used to generate arbitrary decimal probabilities. In fact, in Section 1.4.1, we will first show that we can generate arbitrary decimal probabilities from the set $S = \{0.4, 0.5\}$. The proof is constructive: we will show a procedure for synthesizing logic that generates such probabilities. Next, in Section 1.4.2, we will show that we can generate arbitrary decimal probabilities from the set

$S = \{0.5, 0.8\}$. We will show that with this set of input probabilities, for an output probability of n decimal digits, we can synthesize combinational logic with $2n$ inputs.

Further, in Section 1.5, we will show that there exist sets consisting of a single element that can be used to generate arbitrary decimal probabilities. This is essentially a mathematical result: we will show that the single probability value cannot be a rational value; it must be an irrational root of a polynomial.

In Section 1.6, we will show a practical algorithm based on fraction factorization to synthesize circuits that generate decimal probabilities from the set $S = \{0.4, 0.5\}$. The proposed algorithm optimizes the depth of the circuit.

1.3 Related Work

We point to three related pieces of research:

- In an early set of papers, Gill discussed the problem of generating a new set of probabilities from a given set of probabilities [4, 5]. He focused on synthesizing a sequential state machine to generate the required probabilities.
- In recent work, the proponents of PCMOS discussed the problem of synthesizing combinational logic to generate probability values [2]. These authors suggest a tree-based circuit. Their objective is to realize a set of required probabilities with minimal additional logic. This is positioned as future work; no details are given.
- Wilhelm and Bruck [11] proposed a general method for synthesizing *switching circuits* to achieve a desired probability. Their designs consist of relay switches that are open or closed with specified probabilities. They proposed an algorithm that generates circuits of optimal size for any binary fraction.

In contrast to Gill's work and Wilhelm and Bruck's work, we focus on combinational circuits built with logic gates. Our approach dovetails nicely with the circuit-level PCMOS constructs. It is complementary and orthogonal to the switch-based approach of Wilhelm and Bruck. Our scheme can generate arbitrary decimal probabilities, whereas the method of Wilhelm and Bruck only generates binary fractions.

1.4 Sets with Two Elements that Can Generate Arbitrary Decimal Probabilities

In this section, we will show two input probability sets that contain only two elements and can generate arbitrary decimal probabilities. The first one is the set $S = \{0.4, 0.5\}$ and the second one is the set $S = \{0.5, 0.8\}$.

1.4.1 Generating Decimal Probabilities from the Input Probability Set $S = \{0.4, 0.5\}$

We will first show that we can generate arbitrary decimal probabilities from the input probability set $S = \{0.4, 0.5\}$. Then, we will show an algorithm to synthesize circuits that generate arbitrary decimal probabilities from the set of input probabilities.

Theorem 1.1. *With circuits consisting of fanin-two AND gates and inverters, we can generate arbitrary decimal fractions as output probabilities from the input probability set* $S = \{0.4, 0.5\}$.

Proof. First, we note that an inverter with a probabilistic input gives an output probability equal to one minus the input probability, as was shown in Equation (1.1). An AND gate with two probabilistic inputs performs a multiplication on the two input probabilities, as was shown in Equation (1.2). Thus, we need to prove that with the two operations $1 - x$ and $x \cdot y$, we can generate arbitrary decimal fractions as output probabilities from the input probability set $S = \{0.4, 0.5\}$. We prove this statement by induction on the number of digits n after the decimal point.

Base case:

1. $n = 0$. It is trivial to generate 0 and 1.
2. $n = 1$. We can generate 0.1, 0.2 and 0.3 as follows:

$$\begin{aligned} 0.1 &= 0.4 \times 0.5 \times 0.5, \\ 0.2 &= 0.4 \times 0.5, \\ 0.3 &= (1 - 0.4) \times 0.5. \end{aligned}$$

   Since we can generate the decimal fractions $0.1, 0.2, 0.3$ and 0.4, we can generate $0.6, 0.7, 0.8$ and 0.9 with an extra $1 - x$ operation. Together with the given value 0.5, we can generate any decimal fraction with one digit after the decimal point.

Inductive step:
Assume that the statement holds for all $m \leq (n-1)$. Consider an arbitrary decimal fraction z with n digits after the decimal point. Let $u = 10^n \cdot z$. Here u is an integer.

Consider the following four cases.

1. The case where $0 \leq z \leq 0.2$.
   a. The integer u is divisible by 2. Let $w = 5z$. Then $0 \leq w \leq 1$ and $w = (u/2) \cdot 10^{-n+1}$, having at most $(n-1)$ digits after the decimal point. Thus, based on the induction hypothesis, we can generate w. It follows that z can also be generated as $z = 0.4 \times 0.5 \times w$.

   b. The integer u is not divisible by 2 and $0 \leq z \leq 0.1$. Let $w = 10z$. Then $0 \leq w \leq 1$ and $w = u \cdot 10^{-n+1}$, having at most $(n-1)$ digits after the decimal point. Thus, based on the induction hypothesis, we can generate w. It follows that z can also be generated as $z = 0.4 \times 0.5 \times 0.5 \times w$.
   c. The integer u is not divisible by 2 and $0.1 < z \leq 0.2$. Let $w = 2 - 10z$. Then $0 \leq w < 1$ and $w = 2 - u \cdot 10^{-n+1}$, having at most $(n-1)$ digits after the decimal point. Thus, based on the induction hypothesis, we can generate w. It follows that z can also be generated as $z = (1 - 0.5 \times w) \times 0.4 \times 0.5$.

2. The case where $0.2 < z \leq 0.4$.

   a. The integer u is divisible by 4. Let $w = 2.5z$. Then $0 < w \leq 1$ and $w = (u/4) \cdot 10^{-n+1}$, having at most $(n-1)$ digits after the decimal point. Thus, based on the induction hypothesis, we can generate w. It follows that z can be generated as $z = 0.4 \times w$.
   b. The integer u is not divisible by 4 but is divisible by 2. Let $w = 2 - 5z$. Then $0 \leq w < 1$ and $w = 2 - (u/2) \cdot 10^{-n+1}$, having at most $(n-1)$ digits after the decimal point. Thus, based on the induction hypothesis, we can generate w. It follows that z can be generated as $z = (1 - 0.5 \times w) \times 0.4$.
   c. The integer u is not divisible by 2 and $0.2 < u \leq 0.3$. Let $w = 10z - 2$. Then $0 < w \leq 1$ and $w = u \cdot 10^{-n+1} - 2$, having at most $(n-1)$ digits after the decimal point. Thus, based on the induction hypothesis, we can generate w. It follows that z can also be generated as $z = (1 - (1 - 0.5 \times w) \times 0.5) \times 0.4$.
   d. The integer u is not divisible by 2 and $0.3 < u \leq 0.4$. Let $w = 4 - 10z$. Then $0 \leq w < 1$ and $w = 4 - u \cdot 10^{-n+1}$, having at most $(n-1)$ digits after the decimal point. Thus, based on the induction hypothesis, we can generate w. It follows that z can be generated as $z = (1 - 0.5 \times 0.5 \times w) \times 0.4$.

3. The case where $0.4 < z \leq 0.5$. Let $w = 1 - 2z$. Then $0 \leq w < 0.2$ and w falls into case 1. Thus, we can generate w. It follows that z can be generated as $z = 0.5 \times (1 - w)$.
4. The case where $0.5 < z \leq 1$. Let $w = 1 - z$. Then $0 \leq w < 0.5$ and w falls into one of the above three cases. Thus, we can generate w. It follows that z can be generated as $z = 1 - w$.

For all of the above cases, we proved that z can be generated with the two operations $1 - x$ and $x \cdot y$ on the input probability set $S = \{0.4, 0.5\}$. Thus, we proved the statement for all $m \leq n$. Thus, the statement holds for all integers n. □

Based on the proof above, we derive an algorithm to synthesize a circuit that generates an arbitrary decimal fraction output probability z from the input probability set $S = \{0.4, 0.5\}$. See Algorithm 1.

The function GetDigits(z) in Algorithm 1 returns the number of digits after the decimal point of z. The while loop continues until z has at most one digit after the decimal point. During the loop, it calls the function ReduceDigit(ckt, z), which synthesizes a partial circuit such that the number of digits after the decimal point of z is reduced, which corresponds to the inductive step in the proof. Finally, Algorithm 1

Algorithm 1 Synthesize a circuit consisting of AND gates and inverters that generates a required decimal fraction probability from the given probability set $S = \{0.4, 0.5\}$.

```
{Given an arbitrary decimal fraction $0 \le z \le 1$.}
Initialize $ckt$;
while GetDigits($z$) $> 1$ do
    $(ckt, z) \Leftarrow$ ReduceDigit($ckt, z$);
AddBaseCkt($ckt, z$); {Base case: $z$ has at most one digit after the decimal point.}
return $ckt$;
```

calls the function AddBaseCkt(ckt, z) to synthesize a circuit that realizes a number having at most one digit after the decimal point; this corresponds to the base case of the proof.

Algorithm 2 ReduceDigit(ckt, z)

```
{Given a partial circuit $ckt$ and an arbitrary decimal fraction $0 \le z \le 1$.}
$n \Leftarrow$ GetDigits($z$);
if $z > 0.5$ then {Case 4}
    $z \Leftarrow 1 - z$; AddInverter($ckt$);
if $0.4 < z \le 0.5$ then {Case 3}
    $z \Leftarrow z/0.5$; AddAND($ckt, 0.5$);
    $z \Leftarrow 1 - z$; AddInverter($ckt$);
if $z \le 0.2$ then {Case 1}
    $z \Leftarrow z/0.4$; AddAND($ckt, 0.4$);
    $z \Leftarrow z/0.5$; AddAND($ckt, 0.5$);
    if GetDigits($z$) $< n$ then
        go to END;
    if $z > 0.5$ then
        $z \Leftarrow 1 - z$; AddInverter($ckt$);
    $z = z/0.5$; AddAND($ckt, 0.5$);
else {Case 2: $0.2 < z \le 0.4$}
    $z \Leftarrow z/0.4$; AddAND($ckt, 0.4$);
    if GetDigits($z$) $< n$ then
        go to END;
    $z \Leftarrow 1 - z$; AddInverter($ckt$);
    $z \Leftarrow z/0.5$; AddAND($ckt, 0.5$);
    if GetDigits($z$) $< n$ then
        go to END;
    if $z > 0.5$ then
        $z \Leftarrow 1 - z$; AddInverter($ckt$);
    $z = z/0.5$; AddAND($ckt, 0.5$);
END: return $ckt, z$;
```

Algorithm 1 builds the circuit from the output back to the inputs. The circuit is built up gate by gate when calling the function ReduceDigit(ckt, z), shown in Algorithm 2. Here the function AddInverter(ckt) attaches an inverter to the input of the circuit ckt and then changes the input of the circuit to the input of the inverter.

The function AddAND(ckt, p) attaches a fanin-two AND gate to the input of the circuit and then changes the input of the circuit to one of the inputs of the AND gate. The other input of the AND gate is connected to a random input source of probability p. In Algorithm 2, Lines 3–4 correspond to Case 4 in the proof; Lines 5–7 correspond to Case 3 in the proof; Lines 8–15 correspond to Case 1 in the proof; Lines 16–26 correspond to Case 2 in the proof.

The synthesized circuit has a number of gates that is linear in the number of digits after the required value's decimal point, since at most 3 AND gates and 3 inverters are needed to generate a value with n digits after the decimal point from a value with $(n-1)$ digits after the decimal point.[1] The number of primary inputs of the synthesized circuit is at most $3n+1$.

Example 1.2. We show how to generate the probability value 0.757. Based on Algorithm 1, we can derive a sequence of operations that transform 0.757 to 0.7:

$$0.757 \xRightarrow{1-} 0.243 \xRightarrow{/0.4} 0.6075 \xRightarrow{1-} 0.3925 \xRightarrow{/0.5} 0.785 \xRightarrow{1-} 0.215 \xRightarrow{/0.5} 0.43,$$
$$0.43 \xRightarrow{/0.5} 0.86 \xRightarrow{1-} 0.14 \xRightarrow{/0.4} 0.35 \xRightarrow{/0.5} 0.7.$$

Since 0.7 can be realized as $0.7 = 1-(1-0.4)\times 0.5$, we obtain the circuit shown in Figure 1.4. (Note that here we use a black dot to represent an inverter.) □

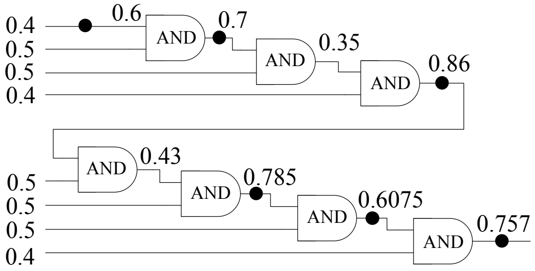

Fig. 1.4: A circuit taking input probabilities from the set $S = \{0.4, 0.5\}$ generating a decimal output probability of 0.757.

[1] In Case 3, z is transformed into $w = 1-2z$ where w is in Case 1(a). Thus, we actually need only 3 AND gates and 1 inverter for Case 3. For the other cases, it is not hard to see that we need at most 3 AND gates and 3 inverters.

1.4.2 Generating Decimal Probabilities from the Input Probability Set $S = \{0.5, 0.8\}$

Given a probability set $S = \{0.4, 0.5\}$, the algorithm in the previous section produces a circuit with at most $3n + 1$ inputs to generate a decimal probability of n digits. If we use the set $S = \{0.5, 0.8\}$, then we can do better in terms of the number of inputs. With this set, we can synthesize a circuit with at most $2n$ inputs that generates a decimal probability of n digits. To prove this, we need the following lemma.

Lemma 1.1. *Given an integer* $n \geq 2$*, for any integer* $0 \leq m \leq 10^n$*, there exist integers* $0 \leq a_i \leq 2^n \binom{n}{i}$*,* $i = 0, 1, \ldots, n$*, such that* $m = \sum_{i=0}^{n} a_i 4^i$*.*

Proof. Define $s_k = \sum_{i=0}^{k} 2^n \begin{pmatrix} n \\ i \end{pmatrix} 4^i$. We first prove the following statement:

Given $0 \leq k \leq n$, for any integer $0 \leq m \leq s_k$, there exist integers $0 \leq a_i \leq 2^n \binom{n}{i}$, $i = 0, 1, \ldots, k$, such that $m = \sum_{i=0}^{k} a_i 4^i$.

We prove the above statement by induction on k.

Base case: When $k = 0$, we have $s_0 = 2^n$. For any integer $0 \leq m \leq 2^n$, let $a_0 = m$. Then $0 \leq a_0 \leq 2^n \binom{n}{0}$. The statement is true for $k = 0$.

Inductive step: Assume the statement holds for $k - 1$ $(k \leq n)$. Consider the statement for k. There are two cases for $0 \leq m \leq s_k$.

1. $0 \leq m \leq 2^n \binom{n}{k} 4^k$. Let $a_k = \left\lfloor \frac{m}{4^k} \right\rfloor$. Then,

$$0 \leq a_k \leq \frac{m}{4^k} \leq 2^n \begin{pmatrix} n \\ k \end{pmatrix}$$

and

$$0 \leq m - a_k 4^k < 4^k \leq 2^n 4^{k-1} \leq \sum_{i=0}^{k-1} 2^n \begin{pmatrix} n \\ i \end{pmatrix} 4^i = s_{k-1}.$$

Based on the induction hypothesis, there exist integers $0 \leq a_i \leq 2^n \binom{n}{i}$, $i = 0, 1, \ldots, k - 1$, such that

$$m - a_k 4^k = \sum_{i=0}^{k-1} a_i 4^i.$$

Therefore, $m = \sum_{i=0}^{k} a_i 4^i$, where $0 \leq a_i \leq 2^n \binom{n}{i}$, for $i = 0, 1, \ldots, k$.

2. $2^n \binom{n}{k} 4^k < m \leq s_k$. Let $a_k = 2^n \binom{n}{k}$. Then,

$$0 < m - a_k 4^k \leq s_k - 2^n \begin{pmatrix} n \\ k \end{pmatrix} 4^k = s_{k-1}.$$

Based on the induction hypothesis, there exist integers $0 \leq a_i \leq 2^n \binom{n}{i}$, $i = 0, 1, \ldots, k - 1$, such that

$$m - a_k 4^k = \sum_{i=0}^{k-1} a_i 4^i.$$

Therefore, $m = \sum_{i=0}^{k} a_i 4^i$, where $0 \le a_i \le 2^n \binom{n}{i}$, for $i = 0, 1, \ldots, k$.

Thus, the statement is true for all $0 \le k \le n$.
Note that when $k = n$,

$$s_k = \sum_{i=0}^{n} 2^n \binom{n}{i} 4^i = 2^n (4+1)^n = 10^n.$$

Thus, for any integer $0 \le m \le 10^n = s_n$, there exist integers $0 \le a_i \le 2^n \binom{n}{i}$, $i = 0, 1, \ldots, n$, such that $m = \sum_{i=0}^{n} a_i 4^i$. □

With the above lemma, we can prove the following theorem.

Theorem 1.2. *For any decimal fraction of n $(n \ge 2)$ digits, there exists a combinational circuit with 2n inputs generating that decimal probability with input probabilities taken from the set $S = \{0.5, 0.8\}$.*

Proof. Consider combination logic with $2n$ inputs $x_1, x_2, \ldots, x_{2n}$ with input probabilities set as

$$P(x_i = 1) = \begin{cases} 0.8, & i = 1, \ldots, n, \\ 0.5, & i = n+1, \ldots, 2n. \end{cases}$$

For $n+1 \le i \le 2n$, since $P(x_i = 1) = 0.5$, we have $P(x_i = 1) = P(x_i = 0) = 0.5$. Therefore, the probability of a certain input combination occurring only depends on the values of the first n inputs or, more precisely, only depends on the number of ones in the first n inputs. Thus, there are in total $2^n \binom{n}{i}$ $(0 \le i \le n)$ input combinations whose probability of occurring is $0.8^i \cdot 0.2^{n-i} \cdot 0.5^n$.

Suppose the given decimal fraction of n digits is $q = \frac{m}{10^n}$, where $0 \le m \le 10^n$ is an integer. Then, based on Lemma 1.1, there exist integers $0 \le a_i \le 2^n \binom{n}{i}$, $i = 0, 1, \ldots, n$, such that $m = \sum_{i=0}^{n} a_i 4^i$.

For each $0 \le i \le n$, since $0 \le a_i \le 2^n \binom{n}{i}$, we are able to choose a_i out of $2^n \binom{n}{i}$ input combinations whose probability of occurring is $0.8^i \cdot 0.2^{n-i} \cdot 0.5^n$; let the combinational logic evaluate to one for these a_i input combinations. Thus, the probability of the output being one is the sum of the probability of occurrence of all input combinations for which the logic evaluates to one, which is,

$$\sum_{i=0}^{n} a_i 0.8^i \cdot 0.2^{n-i} \cdot 0.5^n = \sum_{i=0}^{n} a_i 4^i \cdot 0.1^n = \frac{m}{10^n} = q. \qquad \square$$

Remarks: Like Theorem 1.1, Theorem 1.2 implies a procedure for synthesizing combinational logic to generate a required decimal fraction. Although this procedure will synthesize a circuit with fewer inputs than that synthesized through Algorithm 1, the number of two-input logic gates in this circuit may be greater. Moreover, for this procedure, we must judiciously choose a_i out of $2^n \binom{n}{i}$ input combinations with probability $0.8^i \cdot 0.2^{n-i} \cdot 0.5^n$ of occurring as the minterms of the Boolean function, in order to minimize the gate count. In contrast, Algorithm 1 produces a circuit directly.

1.5 Sets with A Single Element that Can Generate Arbitrary Decimal Probabilities

In Section 1.4, we showed that there exist input probability sets of two elements that can be used to generate arbitrary decimal fractions. A stronger question is whether we can further reduce the size of the set down to one, i.e., whether there exists a real number $0 \leq p \leq 1$ such that any decimal fraction can be generated from p with combinational logic.

The first answer to this question is that there is no *rational* number p such that an arbitrary decimal fraction can be generated from that p through combinational logic. To prove this, we first need the following lemma.

Lemma 1.2. *If the probability $\frac{1}{2}$ can be generated from a rational probability p through combinational logic, then $p = \frac{1}{2}$.*

Proof. Obviously, $0 < p < 1$. Thus, we can assume that

$$p = \frac{a}{b}, \tag{1.4}$$

where both a and b are positive integers, satisfying that $a < b$ and $(a,b) = 1$.

Moreover, we can assume that $a \geq b - a$. Otherwise, suppose that $a < b - a$. Since we can generate $\frac{1}{2}$ from p, we can also generate $\frac{1}{2}$ from $p^* = 1 - p$ by using an inverter to convert p^* into p. Note that $p^* = \frac{a^*}{b^*}$, where $a^* = b - a$ and $b^* = b$, satisfying that $a^* > b^* - a^*$. Thus, we can assume that $a \geq b - a$.

Suppose that the combinational logic generating $\frac{1}{2}$ from p has n inputs. Let l_k $(k = 0, 1, \ldots, n)$ be the number of input combinations that evaluate to one and have exactly k ones. Note that $0 \leq l_k \leq \binom{n}{k}$, for $k = 0, 1, \ldots, n$.

Since each input of the combinational logic has probability p of being 1, we have

$$\frac{1}{2} = \sum_{k=0}^{n} l_k (1-p)^{n-k} p^k. \tag{1.5}$$

Let $c = b - a$. Based on Equation (1.4), we can rewrite Equation (1.5) as

$$b^n = 2 \sum_{k=0}^{n} l_k a^k c^{n-k}. \tag{1.6}$$

From Equation (1.6), we can show that $a = 1$, which we prove by contradiction.

Suppose that $a > 1$. Since $0 \leq l_0 \leq \binom{n}{0} = 1$, l_0 is either 0 or 1. If $l_0 = 0$, then from Equation (1.6), we have

$$b^n = 2\sum_{k=1}^{n} l_k a^k c^{n-k} = 2a\sum_{k=1}^{n} l_k a^{k-1} c^{n-k}.$$

Thus, $a|b^n$. Since $(a,b) = 1$, the only possibility is that $a = 1$ which is contradictory to our hypothesis that $a > 1$. Therefore, we have $l_0 = 1$. Together with binomial expansion

$$b^n = \sum_{k=0}^{n} \binom{n}{k} a^k c^{n-k},$$

we can rewrite Equation (1.6) as

$$c^n + \sum_{k=1}^{n} \binom{n}{k} a^k c^{n-k} = 2c^n + 2\sum_{k=1}^{n} l_k a^k c^{n-k},$$

or

$$c^n = a\sum_{k=1}^{n} \left(\binom{n}{k} - 2l_k\right) a^{k-1} c^{n-k}. \tag{1.7}$$

Thus, $a|c^n$. Since $(a,b) = 1$ and $c = b - a$, we have $(a,c) = 1$. Thus, the only possibility is that $a = 1$, which is contradictory to our hypothesis that $a > 1$.

Therefore, we proved that $a = 1$. Together with the assumption that $b - a \leq a < b$, we get $b = 2$. Thus, p can only be $\frac{1}{2}$. □

Now, we can prove the original statement:

Theorem 1.3. *There is no rational number p such that an arbitrary decimal fraction can be generated from that p with combinational logic.*

Proof. We prove the above statement by contradiction. Suppose that there exists a rational number p such that an arbitrary decimal fraction can be generated from it through combinational logic.

Since an arbitrary decimal fraction can be generated from p, $0.5 = \frac{1}{2}$ can be generated. Thus, based on Lemma 1.2, we have $p = \frac{1}{2}$.

Note that $0.2 = \frac{1}{5}$ is also a decimal number. Thus, there exists combinational logic which can generate the decimal fraction $\frac{1}{5}$ from $p = \frac{1}{2}$. Suppose that the combinational logic has n inputs. Let m_k $(k = 0, 1, \ldots, n)$ be the number of input combinations that evaluate to one and that have exactly k ones.

Since each input of the combinational logic has probability $p = \frac{1}{2}$ of being 1, we have

$$\frac{1}{5} = \sum_{k=0}^{n} m_k \left(1 - \frac{1}{2}\right)^{n-k} \left(\frac{1}{2}\right)^k,$$

or

$$2^n = 5 \sum_{k=0}^{n} m_k,$$

which is impossible since the right-hand side is a multiple of 5.

Therefore, we proved the statement in the theorem. □

Thus, based on Theorem 1.3, we have the conclusion that if such a p exists, it must be an irrational number.

On the one hand, we note that if such a value p exists, then 0.4 and 0.5 can be generated from it. On the other hand, if p can generate 0.4 and 0.5, then p can generate arbitrary decimal numbers, as was shown in Theorem 1.1. The following lemma shows that such a value p that could generate 0.4 and 0.5 does, in fact, exist.

Lemma 1.3. *The polynomial* $g_1(t) = 10t - 20t^2 + 20t^3 - 10t^4 - 1$ *has a real root* $0 < p < 0.5$. *This value p can generate both* 0.4 *and* 0.5 *through combinational logic.*

Proof. First, note that $g_1(0) = -1 < 0$ and that $g_1(0.5) = 0.875 > 0$. Based on the continuity of the function $g_1(t)$, there exists a $0 < p < 0.5$ such that $g_1(p) = 0$. Let polynomial $g_2(t) = t - 2t^2 + 2t^3 - t^4$. Thus, $g_2(p) = 0.1$.

Note that the Boolean function

$$f_1(x_1,x_2,x_3,x_4,x_5) = (x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4 \vee \neg x_5)$$

has 30 minterms, $m_1, m_2, \ldots, m_{30}$. It is not hard to verify that with $P(x_i = 1) = p$ for $i = 1,2,3,4,5$, the output probability of f_1 is

$$\begin{aligned} p_1 &= 5(1-p)^4 p + 10(1-p)^3 p^2 + 10(1-p)^2 p^3 + 5(1-p)p^4 \\ &= 5g_2(p) = 0.5. \end{aligned}$$

Thus, the probability value 0.5 can be generated. The Boolean function

$$\begin{aligned} f_2(x_1,x_2,x_3,x_4,x_5) = {} & (x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_3 \vee \neg x_5) \\ & \wedge (\neg x_2 \vee x_3 \vee \neg x_5) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4 \vee \neg x_5) \end{aligned}$$

has 24 minterms, $m_2, m_4, m_5, \ldots, m_8, m_{10}$, $m_{12}, m_{13}, \ldots, m_{24}, m_{26}, m_{28}, m_{29}, m_{30}$. It is not hard to verify that with $P(x_i = 1) = p$ for $i = 1,2,3,4,5$, the output probability of f_2 is

$$\begin{aligned} p_2 &= 4(1-p)^4 p + 8(1-p)^3 p^2 + 8(1-p)^2 p^3 + 4(1-p)p^4 \\ &= 4g_2(p) = 0.4. \end{aligned}$$

Thus, the probability value 0.4 can be generated. □

Based on Theorem 1.1 and Lemma 1.3, we have the following theorem.

Theorem 1.4. *With the set $S = \{p\}$, where p is the root of the polynomial $g_1(t) = 10t - 20t^2 + 20t^3 - 10t^4 - 1$ in the unit interval, we can generate arbitrary decimal fractions with combinational logic.*

1.6 Implementation

In this section, we will discuss algorithms to optimize circuits that generate decimal probabilities from the input probability set $S = \{0.4, 0.5\}$.

As shown in Example 1.2, the circuit synthesized by Algorithm 1 is in a linear style (i.e., each gate adds to the depth of the circuit). For practical purposes, we want circuits with shallower depth. We explore two kinds of optimizations to reduce the depth.

The first kind of optimization is at the logic level. The circuit synthesized by Algorithm 1 is composed of inverters and AND gates. We can reduce its depth by properly repositioning certain AND gates, as illustrated in Figure 1.5.

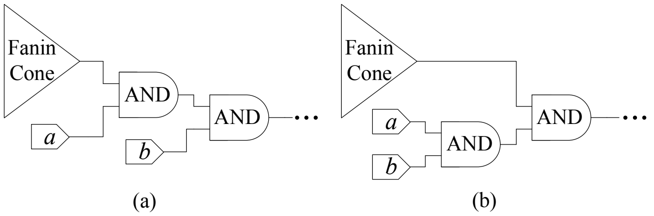

Fig. 1.5: An illustration of balancing to reduce the depth of the circuit. Here *a* and *b* are primary inputs. (a): The circuit before balancing. (b): The circuit after balancing.

The second kind of optimization is at a higher level, based on the factorization of the decimal fraction. We use the following example to illustrate the basic idea.

Example 1.3. Suppose we want to generate the decimal fraction probability value 0.49.

Method based on Algorithm 1: We can derive the following transformation sequence:

$$0.49 \overset{/0.5}{\Longrightarrow} 0.98 \overset{1-}{\Longrightarrow} 0.02 \overset{/0.4}{\Longrightarrow} 0.05 \overset{/0.5}{\Longrightarrow} 0.1.$$

The synthesized circuit is shown in Figure 1.6(a). Notice that the circuit is balanced and it still has 5 AND gates and depth 4.

Method based on factorization: Notice that $0.49 = 0.7 \times 0.7$. Thus, we can generate the probability 0.7 twice and feed these values into an AND gate. The synthesized circuit is shown in Figure 1.6(b). Compared to the circuit in Figure 1.6(a), both the number of AND gates and the depth of the circuit are reduced. □

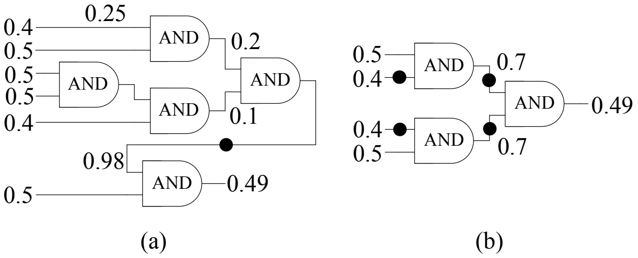

Fig. 1.6: Synthesizing combinational logic to generate probability 0.49. (a): The circuit synthesized through Algorithm 1. (b): The circuit synthesized based on fraction factorization.

Algorithm 3 shows the procedure that synthesizes the circuit based on the factorization of the decimal fraction. The factorization is actually carried out on the numerator. A crucial function is PairCmp(a_l, a_r, b_l, b_r), which compares the integer factor pair (a_l, a_r) with the pair (b_l, b_r) and returns a positive (negative) value if the pair (a_l, a_r) is better (worse) than the pair (b_l, b_r). Algorithm 4 shows how the function PairCmp(a_l, a_r, b_l, b_r) is implemented.

The quality of a factor pair (a_l, a_r) should reflect the quality of the circuit that generates the original probability based on that factorization. For this purpose, we define a function EstDepth(x) to estimate the depth of the circuit that generates the decimal fraction of a numerator x. If $1 \le x \le 9$, the corresponding fraction is $x/10$. EstDepth(x) is set as the depth of the circuit that generates the fraction $x/10$, which is

$$\mathrm{EstDepth}(x) = \begin{cases} 0, & x = 4,5,6, \\ 1, & x = 2,3,7,8, \\ 2, & x = 1,9. \end{cases}$$

When $x \ge 10$, we use a simple heuristic to estimate the depth: we let $\mathrm{EstDepth}(x) = \lceil \log_{10}(x) \rceil + 1$. The intuition behind this is that the depth of the circuit is a monotonically increasing function of the number of digits of x. The estimated depth of the circuit that generates the original fraction based on the factor pair (a_l, a_r) is

$$\max\{\mathrm{EstDepth}(a_l), \mathrm{EstDepth}(a_r)\} + 1. \tag{1.8}$$

The function PairCmp(a_l, a_r, b_l, b_r) essentially compares the quality of pair (a_l, a_r) and pair (b_l, b_r) based on Equation (1.8). Further details are given in Algorithm 4.

Algorithm 3 ProbFactor(ckt, z)

```
{Given a partial circuit $ckt$ and an arbitrary decimal fraction $0 \le z \le 1$.}
$n \Leftarrow$ GetDigits$(z)$;
if $n \le 1$ then
    AddBaseCkt$(ckt, z)$;
    return $ckt$;
$u \Leftarrow 10^n z$; $(u_l, u_r) \Leftarrow (1, u)$; {$u$ is the numerator of the fraction $z$}
for each factor pair $(a, b)$ of integer $u$ do
    if PairCmp$(u_l, u_r, a, b) < 0$ then
        $(u_l, u_r) \Leftarrow (a, b)$; {Choose the best factor pair for $z$}
$w \Leftarrow 10^n - u$; $(w_l, w_r) \Leftarrow (1, w)$;
for each factor pair $(a, b)$ of integer $w$ do
    if PairCmp$(w_l, w_r, a, b) < 0$ then
        $(w_l, w_r) \Leftarrow (a, b)$; {Choose the best factor pair for $1 - z$}
if PairCmp$(u_l, u_r, w_l, w_r) < 0$ then
    $(u_l, u_r) \Leftarrow (w_l, w_r)$; $z \Leftarrow w/10^n$;
    AddInverter$(ckt)$;
if IsTrivialPair$(u_l, u_r)$ then {$u_l = 1$ or $u_r = u$}
    ReduceDigit$(ckt, z)$; ProbFactor$(ckt, z)$;
    return $ckt$;
$n_l \Leftarrow \lceil \log_{10}(u_l) \rceil$; $n_r \Leftarrow \lceil \log_{10}(u_r) \rceil$;
if $n_l + n_r > n$ then {Unable to factor $z$ into two decimal fractions in the unit interval}
    ReduceDigit$(ckt, z)$; ProbFactor$(ckt, z)$;
    return $ckt$;
$z_l \Leftarrow u_l/10^{n_l}$; $z_r \Leftarrow u_r/10^{n_r}$;
ProbFactor$(ckt_l, z_l)$; ProbFactor$(ckt_r, z_r)$;
Add an AND gate with output as $ckt$ and two inputs as $ckt_l$ and $ckt_r$;
if $n_l + n_r < n$ then
    AddExtraLogic$(ckt, n - n_l - n_r)$;
return $ckt$;
```

Algorithm 4 PairCmp(a_l, a_r, b_l, b_r)

```
{Given two integer factor pairs $(a_l, a_r)$ and $(b_l, b_r)$}
$c_l \Leftarrow$ EstDepth$(a_l)$; $c_r \Leftarrow$ EstDepth$(a_r)$;
$d_l \Leftarrow$ EstDepth$(b_l)$; $d_r \Leftarrow$ EstDepth$(b_r)$;
Order$(c_l, c_r)$; {Order $c_l$ and $c_r$, so that $c_l \le c_r$}
Order$(d_l, d_r)$; {Order $d_l$ and $d_r$, so that $d_l \le d_r$}
if $c_r < d_r$ then {The circuit w.r.t. the first pair has smaller depth}
    return 1;
else if $c_r > d_r$ then {The circuit w.r.t. the first pair has larger depth}
    return -1;
else
    if $c_l < d_l$ then {The circuit w.r.t. the first pair has fewer ANDs}
        return 1;
    else if $c_l > d_l$ then {The circuit w.r.t. the first pair has more ANDs}
        return -1;
    else
        return 0;
```

In Algorithm 3, Lines 2–5 corresponds to the trivial fractions. If the fraction z is non-trivial, Lines 6–9 choose the best factor pair (u_l, u_r) of integer u, where u is the numerator of the fraction z. Lines 10–13 choose the best factor pair (w_l, w_r) of integer w, where w is the numerator of the fraction $1-z$. Finally, the better factor pair of (u_l, u_r) and (w_l, w_r) is chosen. Here, we consider the factorization on both z and $1-z$, since in some cases the latter might be better than the former. An example is $z = 0.37$. Note that $1-z = 0.63 = 0.7 \times 0.9$; this has a better factor pair than z itself.

After obtaining the best factor pair, we check whether we can utilize it. Lines 17–19 check whether the factor pair (u_l, u_r) is trivial. A factor pair is considered trivial if $u_l = 1$ or $u_r = 1$. If the best factor pair is trivial, we call the function ReduceDigit(ckt, z) (shown in Algorithm 2) to reduce the number of digits after the decimal point of z. Then we perform factorization on the new z.

If the best factor pair is non-trivial, Lines 20–23 continue to check whether the factor pair can be transformed into two decimal fractions in the unit interval. Let n_l be the number of digits of the integer u_l and n_r be the number of digits of the integer u_r. If $n_l + n_r > n$, where n is the number of digits after the decimal point of z, then it is impossible to utilize the factor pair (u_l, u_r) to factorize z. For example, consider $z = 0.143$. Although we could factorize $u = 143$ as 11×13, we could not utilize the factor pair $(11, 13)$ for the factorization of 0.143. The reason is that either the factorization 0.11×1.3 or the factorization 1.1×0.13 contains a fraction larger than 1, which cannot be a probability value.

Finally, if it is possible to utilize the best factor pair, Lines 24–26 synthesize two circuits for fractions $u_l/10^{n_l}$ and $u_r/10^{n_r}$, respectively, and then combine these two circuits with an AND gate. Lines 27–28 check whether $n > n_l + n_r$. If this is the case, we have

$$z = u/10^n = u_l/10^{n_l} \cdot u_r/10^{n_r} \cdot 0.1^{n-n_l-n_r}.$$

We need to add an extra AND gate with one input probability $0.1^{n-n_l-n_r}$ and the other input probability $u_l/10^{n_l} \cdot u_r/10^{n_r}$. The extra logic is added through the function AddExtraLogic(ckt, m).

1.7 Empirical Validation

We empirically validate the effectiveness of the synthesis scheme that was presented in Section 1.6. For logic-level optimization, we use the “balance” command of the logic synthesis tool ABC [7], which we find very effective in reducing the depth of a tree-style circuit.[2]

Table 1.1 compares the quality of the circuits generated by three different schemes. The first scheme is called “Basic,” which is based on Algorithm 1. It generates a linear-style circuit. The second scheme is called “Basic+Balance,” which

[2] We find that the other combinational synthesis commands of ABC such as “rewrite” do not affect the depth or the number of AND gates of a tree-style AND-inverter graph.

Table 1.1: A comparison of the basic synthesis scheme, the basic synthesis scheme with balancing, and the factorization-based synthesis scheme with balancing.

Number	Basic		Basic+Balance			Factor+Balance				
of Digits	#AND	Depth	#AND	Depth	Runtime	#AND	Depth	Runtime	#AND	Depth
					(ms)			(ms)	Imprv. (%)	Imprv. (%)
n			a_1	d_1		a_2	d_2		$100\frac{a_1-a_2}{a_1}$	$100\frac{d_1-d_2}{d_1}$
2	3.67	3.67	3.67	2.98	0.22	3.22	2.62	0.22	12.1	11.9
3	6.54	6.54	6.54	4.54	0.46	5.91	3.97	0.66	9.65	12.5
4	9.47	9.47	9.47	6.04	1.13	8.57	4.86	1.34	9.45	19.4
5	12.43	12.43	12.43	7.52	0.77	11.28	5.60	0.94	9.21	25.6
6	15.40	15.40	15.40	9.01	1.09	13.96	6.17	1.48	9.36	31.5
7	18.39	18.39	18.39	10.50	0.91	16.66	6.72	1.28	9.42	35.9
8	21.38	21.38	21.38	11.99	0.89	19.34	7.16	1.35	9.55	40.3
9	24.37	24.37	24.37	13.49	0.75	22.05	7.62	1.34	9.54	43.6
10	27.37	27.37	27.37	14.98	1.09	24.74	7.98	2.41	9.61	46.7
11	30.36	30.36	30.36	16.49	0.92	27.44	8.36	2.93	9.61	49.3
12	33.35	33.35	33.35	17.98	0.73	30.13	8.66	4.13	9.65	51.8

combines Algorithm 1 and the logic-level balancing algorithm. The third scheme is called "Factor+Balance," which combines Algorithm 3 and the logic-level balancing algorithm. We perform experiments on a set of target decimal probabilities that have n digits after the decimal point and average the results. The table shows the results for n ranging from 2 to 12. When $n \leq 5$, we synthesize circuits for all possible decimal fractions with n digits after the decimal point. When $n \geq 6$, we randomly choose 100000 decimal fractions with n digits after the decimal point as the synthesis targets. We show the average number of AND gates, average depth and average CPU runtime in columns "#AND," "Depth," and "Runtime," respectively.

From Table 1.1, we can see that both the "Basic+Balance" and the "Factor+Balance" synthesis schemes have only millisecond-order CPU runtimes. Compared to the "Basic+Balance" scheme, the "Factor+Balance" scheme reduces by 10% the number of AND gates and by more than 10% the depth of the circuit for all n. The percentage of reduction on the depth increases with increasing n. For $n = 12$, the average depth of the circuit is reduced by more than 50%.

In Figure 1.7, we plot the average number of AND gates and depth of the circuit versus n for both the "Basic+Balance" scheme and the "Factor+Balance" scheme. Clearly, the figure shows that the "Factor+Balance" scheme is superior to the "Basic+Balance" scheme. As shown in the figure, the average number of AND gates in the circuits synthesized by both the "Basic+Balance" scheme and the "Factor+Balance" scheme increases linearly with n. The average depth of the circuit synthesized by the "Basic+Balance" scheme also increases linearly with n. In contrast, the average depth of the circuit synthesized by the "Factor+Balance" scheme increases logarithmically with n.

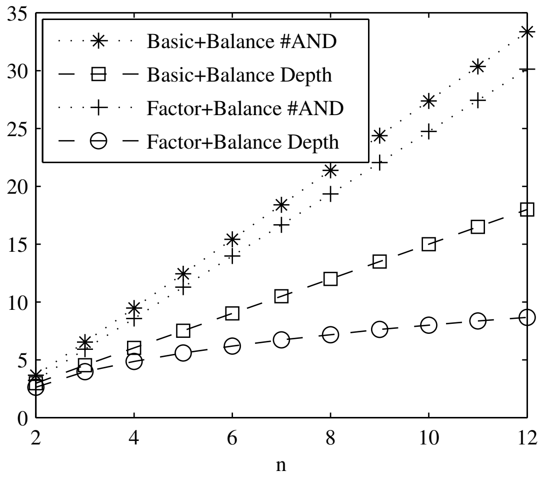

Fig. 1.7: Average number of AND gates and depth of the circuit versus n.

1.8 Chapter Summary

In this chapter, we introduced the problem of synthesizing combinational logic to generate specified output probabilities from a given set of input probabilities. We focused on generating decimal output probabilities and aimed at finding a small set of input probabilities. We first showed that input probability sets consisting of two elements can be used to generate arbitrary decimal output probabilities. Next, as a mathematical result, we demonstrated that there exists a single input probability that can be used to generate any output decimal probability. That input probability cannot be rational; it must be an irrational root of a polynomial. We proposed algorithms to synthesize circuits that generate output decimal probabilities from the input probability set $S = \{0.4, 0.5\}$. To optimize the depth of the circuits, we proposed a method based on fraction factorization. We demonstrated the effectiveness of our algorithm with experimental results. The average depth of the circuits synthesized by our method is logarithmic in the number of digits of the output decimal probability.

References

1. Borkar, S., Karnik, T., De, V.: Design and reliability challenges in nanometer technologies. In: Design Automation Conference, p. 75 (2004)
2. Chakrapani, L., Korkmaz, P., Akgul, B., Palem, K.: Probabilistic system-on-a-chip architecture. ACM Transactions on Design Automation of Electronic Systems **12**(3), 1–28 (2007)

3. Cheemalavagu, S., Korkmaz, P., Palem, K., Akgul, B., Chakrapani, L.: A probabilistic CMOS switch and its realization by exploiting noise. In: IFIP International Conference on VLSI, pp. 535–541 (2005)
4. Gill, A.: Synthesis of probability transformers. Journal of the Franklin Institute **274**(1), 1–19 (1962)
5. Gill, A.: On a weight distribution problem, with application to the design of stochastic generators. Journal of the ACM **10**(1), 110–121 (1963)
6. Karnik, T., Borkar, S., De, V.: Sub-90nm technologies: Challenges and opportunities for CAD. In: International Conference on Computer-Aided Design, pp. 203–206 (2002)
7. Mishchenko, A., et al.: ABC: A system for sequential synthesis and verification (2007). URL http://www.eecs.berkeley.edu/ alanmi/abc/
8. Nepal, K., Bahar, R., Mundy, J., Patterson, W., Zaslavsky, A.: Designing logic circuits for probabilistic computation in the presence of noise. In: Design Automation Conference, pp. 485–490 (2005)
9. von Neumann, J.: Probabilistic logics and the synthesis of reliable organisms from unreliable components. In: Automata Studies, pp. 43–98. Princeton University Press (1956)
10. Qian, W., Riedel, M.D.: The synthesis of robust polynomial arithmetic with stochastic logic. In: Design Automation Conference, pp. 648–653 (2008)
11. Wilhelm, D., Bruck, J.: Stochastic switching circuit synthesis. In: International Symposium on Information Theory, pp. 1388–1392 (2008)