On teaching fast adder designs: revisiting Ladner & Fischer*

Guy Even †

February 1, 2006

Abstract

We present a self-contained and detailed description of the parallel-prefix adder of Ladner and Fischer. Very little background is assumed in digital hardware design. The goal is to understand the rational behind the design of this adder and view the parallel-prefix adder as an outcome of a general method.

*This essay is from the book: Shimon Even Festschrift, edited by Goldreich, Rosenberg, and Selman, LNCS 3895, pp. 313-347,2006 .

†Department of Electrical-Engineering, Tel-Aviv University, Israel. E-mail:guy@eng.tau.ac.il

Contents

1 Introduction

This essay is about how to teach adder designs for undergraduate Computer Science (CS) and Electrical Engineering (EE) students. For the past eight years I have been teaching the second hardware course in Tel-Aviv University's EE school. Although the goal is to teach how to build a simple computer from basic gates, the part I enjoy teaching the most is about addition. At first I thought I felt so comfortable with teaching about adders because it is a well defined, very basic question, and the solutions are elegant and can be proved rigorously without exhausting the students. After a few years, I was able to summarize all these nice properties simply by saying that this is the most algorithmic topic in my course. Teaching has helped me realize that appreciation of algorithms is not straightforward; I was lucky to have been influenced by my father. In fact, while writing this essay, I constantly asked myself how he would have presented this topic.
When writing this essay I had three types of readers in mind.

- Lecturers of undergraduate CS hardware courses. Typically, CS students have a good background in discrete math, data structures, algorithms, and finite automata. However, CS students often lack enthusiasm for hardware, and the only concrete machine they are comfortable with is a Turing machine. To make teaching easier, I added a rather long preliminaries section that defines the hardware model and presents some hardware terminology. Even combinational gates and flip-flops are briefly described.

- Lecturers of undergraduate EE hardware students. In contrast to CS students, EE students are often enthusiastic about hardware (including devices, components, and commercial products), but are usually indifferent to formal specification, proofs, and asymptotic bounds. These students are eager to learn about the latest buzzwords in VLSI and microprocessors. My challenge, when I teach about adders, is to convince the students that learning about an old and solved topic is useful.

- General curious readers (especially students). Course material is often presented in the shortest possible way that is still clear enough to follow. I am not aware of a text that tells a story that sacrifices conciseness for insight about hardware design. I hope that this essay could provide such insight to students interested in learning more about what happens "behind the screen".

1.1 On teaching hardware

Most hardware textbooks avoid abstractions, definitions, and formal claims and proofs (Müller and Paul [MüllerPaul00] is an exception). Since I regard hardware design as a subbranch of algorithms, I think it is a disadvantage not to follow the format of algorithm books. I tried to follow this rule when I prepared lecture notes for my course [Even04]. However, in this essay I am lax in following this rule. First, I assumed that the readers could easily fill in the missing formalities. Second, my impression of my father's teaching was that he preferred clarity over formality. On the other hand, I tried to present the development of a fast adder in a systematic fashion and avoid ad-hoc solutions. In particular, a distinction is made between concepts and

representation (e.g., we interpret the celebrated "generate-carry" and "propagate-carry" bits as a representation of functions, and introduce them rather late in a specific design in Sec. 6.7).

I believe the communication skills of most computer engineers would greatly benefit if they acquired a richer language. Perhaps one should start by modeling circuits by graphs and using graph terminology (e.g., out-degree vs. fanout, depth vs. delay). I decided to stick to hardware terminology since I suspect that people with a background in graphs are more flexible. Nevertheless, whenever possible, I tried to use graphs to model circuits (e.g., netlists, communication graphs).

1.2 A brief summary

This essay mainly deals with the presentation of one of the parallel-prefix adders of Ladner and Fischer [LadnerFischer80]. This adder was popularized by Brent and Kung [BrentKung82] who presented a regular layout for it as well as a reduction of its fanout. In fact, it is often referred to as the "Brent-Kung" adder. Our focus is on a detailed and self-contained explanation of this adder.

The presentation of the parallel-prefix adder in many texts is short but lacks intuition (see [BrentKung82, MüllerPaul00, ErceLang04]). For example, the carry-generate (g_i) and carry-propagate (p_i) signals are introduced as a way to compute the carry bits without explaining their origin[1]. In addition, an associative operator is defined over pairs (g_i, p_i) as a way to reduce the task of computing the carry-bits to a prefix problem. However, this operator is introduced without explaining how it is derived.

Ladner and Fischer's presentation does not suffer from these drawbacks. The parallel-prefix adder is systematically obtained by "parallelizing" the "bit-serial adder" (i.e., the trivial finite state machine with two states, see Sec. 4.1). According to this explanation the pair of carry-generate and carry-propagate signals represent three functions defined over the two states of the bit-serial adder. The associative operator is simply a composition of these functions. The mystery is unravelled and one can see the role of each part.

Ladner and Fischer's explanation is not long or complicated, yet it does not appear in textbooks. Perhaps a detailed and self-contained presentation of the parallel-prefix adder will influence the way parallel-prefix adders are taught. I believe that students can gain much more by understanding the rational behind such an important design. Topics taught so that the students can add some items to their "bag of tricks" often end up in the "bag of obscure and forgotten tricks". I believe that the parallel-prefix adder belongs to the collection of fundamental algorithmic paradigms and can be presented as such.

1.3 Confusing terminology (a note for experts)

The terms "parallel-prefix adder" and "carry-lookahead adders" are used inconsistently in the literature. Our usage of these terms refers to the algorithmic method employed in obtaining the design rather than the specifics of each adder. We use the term "parallel-prefix adder" to refer to an adder that is based on a reduction of the task of computing the carry-bits to a prefix problem (defined in Sec. 6.3). In particular, parallel-prefix adders in this essay are based on the parallel prefix circuits of Ladner and Fischer. The term "carry-lookahead adder" refers to an adder in which special gates (called carry-lookahead gates) are organized in a tree-like structure.

[1] The signals g_i and p_i are defined in Sec. 6.7

The topology is not precisely a tree for two reasons. First, often connections are made between nodes in the same layer. Second, information flows both up and down the tree

One can argue justifiably that, according to this definition, a carry-lookahead is a special case of a parallel-prefix adder. To help the readers, we prefer to make the distinction between the two types of adders.

1.4 Questions

Questions for the students appear in the text. The purpose of these questions is to help students check their understanding, consider alternatives to the text, or just think about related issues that we do not focus on. Before presenting a topic, I usually try to convince the students that they have something to learn. The next question is a good example for such an attempt.

Question 1.1 *1. What is the definition of an adder? (Note that this is a question about hardware design, not about Zoology.)*

2. Can you prove the correctness of the addition algorithm taught in elementary school?

3. (Assuming students are familiar with the definition of the delay (i.e., depth) of a combinational circuit) What is the smallest possible delay of an adder? Do you know of an adder that achieves this delay?

4. Suppose you are given the task of adding very long numbers. Could you share this work with friends so that you could work on it simultaneously to speed up the computation?

1.5 Organization

We begin with a preliminaries in Section 2. This section is a brief review of digital hardware design. In Section 3, we define two types of binary adders: a combinational adder and a bit-serial adder. In Section 4, we present trivial designs for each type of adder. The synchronous adder is an implementation of a finite state machine with two states. The combinational adder is a "ripple-carry adder". In Section 5, we prove lower bounds on the cost and delay of a combinational adder. Section 6 is the heart of the essay. In it, we present the parallel-prefix circuit of Ladner and Fischer as well as the parallel-prefix adder. In Section 7, we discuss various issues related to adders and their implementation. In Section 8, we briefly outline the history of adder designs. We close with a discussion that attempts to speculate why the insightful explanation in [LadnerFischer80] has not made it into textbooks.

2 Preliminaries

2.1 Digital operation

We assume that inputs and outputs of devices are always either zero or one. This assumption is unrealistic due to the fact that the digital value is obtained by rounding an analog value that changes continuously (e.g., voltage). There is a gap between analog values that are rounded to zero and analog values that are rounded to one. When the analog value is in this gap, its digital value is neither zero or one.

The advantage of this assumption is that it simplifies the task of designing digital hardware. We do need to take precautions to make sure that this unrealistic assumption will not render

our designs useless. We set strict design rules for designing circuits to guarantee well defined functionality.

We often use the term *signal*. A signal is simply zero or one value that is output by a gate, input to a gate, or delivered by a wire.

2.2 Building blocks

The first issue we need to address is: what are our building blocks? The building blocks are combinational gates, flip-flops, and wires. We briefly describe these objects.

Combinational gates. A *combinational gate* (or gate, in short) is a device that implements a Boolean function. What does this mean? Consider a Boolean function $f : \{0,1\}^k \to \{0,1\}^\ell$. Now consider a device G with k inputs and ℓ outputs. We say that G *implements* f if the outputs of G equal $f(\alpha) \in \{0,1\}^\ell$ when the input equals $\alpha \in \{0,1\}^\ell$. Of course, the evaluation of $f(\alpha)$ requires time and cannot occur instantaneously. This is formalized by requiring that the inputs of G remain stable with the value α for at least d units of time. After d units of time elapse, the output of G stabilizes on $f(\alpha)$. The amount of time d that is required for the output of G to stabilize on the correct value (assuming that the inputs are stable during this period) is called the *delay* of a gate.

Typical gates are inverters and gates that compute the Boolean OR/AND/XOR of two bits. We depict gates by boxes; the functionality is written in the box. We use the convention that information flows rightwards or downwards. Namely, The inputs of a box are on the right side and the outputs are on the left side (or inputs on the top side and outputs on the bottom side).

We also consider a particular gate, called a *full-adder*, that is useful for addition, defined as follows.

Definition 2.1 (Full-Adder) *A* full-adder *is a combinational gate with* 3 *inputs* $x, y, z \in \{0,1\}$ *and* 2 *outputs* $c, s \in \{0,1\}$ *that satisfies:* $2c + s = x + y + z$. *(Note that each bit is viewed as a zero/one integer.)*

The output s of a full-adder is called the *sum output*, while the output c of a full-adder is called the *carry-out output*. We denote a full-adder by FA. A *half-adder* is a degenerate full-adder with only two inputs (i.e., the third input z of the full-adder is always input a zero).

We do not discuss here how to build a full-adder from basic gates. Since a full-adder has a constant number of inputs and outputs, every (reasonable) implementation has constant cost and delay.

Flip-Flops. A flip-flop is a memory device. Here we use only a special type of flip-flops called *edge triggered D-flip-flops*. What does this mean? We assume that time is divided into intervals, each interval is called a clock cycle. Namely, the ith clock cycle is the interval $(t_i, t_{i+1}]$. A flip-flop has one input (denoted by D), and one output (denoted by Q). The output Q during clock cycle $i+1$ equals the value of the input D at time t_i (end of clock cycle i). We denote a flip-flop by FF.

This functionality is considered as memory because the input D is sampled at the end of clock cycle i. The sampled value is stored and output during the next clock cycle (i.e., clock cycle $i+1$). Note that D may change during clock cycle $i+1$, but the output Q must stay fixed.

We remark that three issues are ignored in this description: (i) Initialization. What does a flip-flop output during the first clock cycle (i.e., clock cycle zero)? We assume that it outputs a zero. (ii) Timing - we ignore timing issues such as setup time and hold time. (iii) The clock signal is missing. (The role of the clock signal is to mark the beginning of each clock cycle.) In fact, every flip-flop has an additional input for a global signal called the clock signal. We assume that the clock signal is used only for feeding these special inputs, and that all the clock inputs of all the flip-flops are fed by the same clock signal. Hence, we may ignore the clock signal.

Wires. The idea behind connecting a wire between two components is to take the output of one component and use it as input to another component. We refer to an input or an output of a component as a *port*. In this essay, a wire can connect exactly two ports; one port is an output port and the other port is an input port. We assume also that a wire can deliver only a single bit.

We will later see that there are strict rules regarding wires. To give an idea of these rules, note that it makes little sense to connect two outputs ports to each other. However, it does make sense to feed multiple input ports by the same output port. We, therefore, allow different wires to be connected to the same output port. However, we do not allow more than one wire to be connected to the same input port. The reason is that multiple wires feeding the same input port could cause an ambiguity in the definition of the input value.

The number of inputs ports that are fed by the same output port is called the *fanout* of the output. In the hardware design community, the fanout is often defined as the number of inputs minus one. We prefer to define the fanout as the out-degree.

We remark that very often connections are depicted by *nets*. A net is a set of input and output ports that are connected by wires. In graph terminology, a net is a hyperedge. Since we are not concerned in this essay with the detailed physical design, and do not consider complicated connections such as buses, we consider only point-to-point connections. An output port that feeds multiple input ports can be modelled by a star of directed edges that emanate from the output port. Hence, one can avoid using nets. Another option is to add trivial gates in branching points, as described in the next section.

2.3 Combinational circuits

One can build complex circuits from gates by connecting wires between gates. However, only a small subset of such circuits are combinational. (In the theory community combinational circuits are known as Boolean circuits.) To guarantee well defined functionality, strict design rules are defined regarding the connections between gates. Only circuits that abide these rules are called combinational circuits. We now describe these rules.

The first rule is the "output-to-input" rule which says: *the starting point of every connection must be an output port and the end point must be an input port.* There is a problem with this rule; namely, how do we feed the external inputs to input ports? We encounter a similar problem with external outputs. Instead of setting exceptions for external inputs and outputs, perhaps the best way to solve this problem is by defining special gates for external inputs and outputs. We define an input gate as a gate with a single output port and no input port. An input gate simply models an external signal that is fed to the circuit. Similarly, we define an output gate as a gate with a single input port and no output port.

The second rule says: *feed every input port exactly once*. This means that there must be exactly one connection to every input port of every gate. We do not allow input ports that are not fed by some signal. The reason is that an unconnected input may violate the digital abstraction (namely, we cannot decide whether it feeds a zero or a one). We do not allow multiple connections to the same input port. The reason is that different connections to the same input port may deliver different values, and then the input value is not well defined. (Note that we do allow the same output port to be connected to multiple inputs and we also allow unconnected output ports.)

The third rule is the "no-cycles" rule which says: *a connection is forbidden if it closes a directed cycle*. The idea is that we can model a circuit C using a directed graph $G(C)$. This graph is called the *netlist* of C. We assign a vertex for every gate and a directed edge for every wire. The orientation of the edge is from the output port to the input port. (Note that we can always orient the edges thanks to the output-to-input rule). The no-cycles rule simply does not allow directed cycles in the directed graph $G(C)$.

Finally, we add a word about nets. Very often connections in circuits are not depicted only by point-to-point wires. Instead, one often draws nets that form a connection between more than two ports (see, for example, the schematic on the right side of Fig. 1). A net is depicted as a tree whose leaves are the ports of the net. We refer to the interior vertices of these trees as branching points. We interpret every branching point in a drawing of a net as a trivial combinational gate with one input port and one output port (the output port may be connected to many input ports). In this trivial gate, the output value simply equals the input value. Hence one should feel comfortable with branching points as long as nets contain exactly one output port.

Question 2.2 *Consider the circuits depicted in Figure 1. Can you explain why these are not valid combinational circuits?*

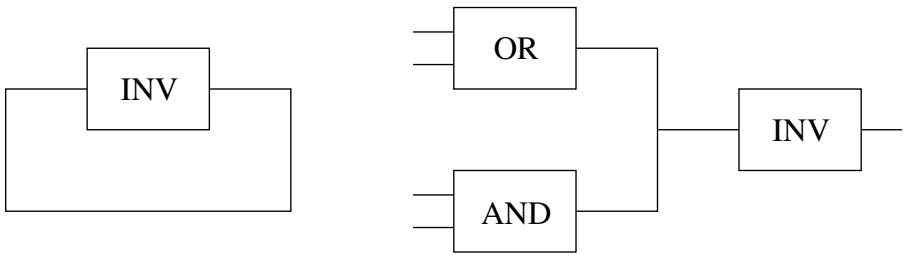

Figure 1: Two examples of non-combinational circuits.

The definition we use for combinational circuits is syntactic; namely, we only require that the connections between the components follow some simple rules. Our focus was syntax and we did not say anything about functionality. This, of course, does not mean that we are not interested in functionality! In natural languages (like English), the meaning of a sentence with correct syntax may not be well defined. The syntactic definition of combinational circuits has two main advantages: (i) It is easy to check if a circuit is indeed combinational. (ii) The functionality of every combinational circuit is well defined. The task of determining the output values given the input values is referred to as *logical simulation*; it can performed as follows. Given the input values of the circuit, one can scan the circuit starting from the inputs and determine

all the values of the inputs and outputs of gates. When this scan ends, the output values of the circuit are known. The order in which the gates should be scanned is called *topological order*, and this order can can be computed in linear time [Even79, Sec. 6.5]. It follows that combinational circuits implement Boolean functions just as gates do. The difference is that functions implemented by combinational circuits are bigger (i.e., have more inputs/outputs).

2.4 Synchronous circuits

Synchronous circuits are built from gates and flip-flops. Obviously, not every collection of gates and flip-flops connected by wires constitutes a "legal" synchronous circuit. Perhaps the simplest way to define a synchronous circuit is by a reduction that maps synchronous circuits to combinational circuits.

Consider a circuit C that is simply a set of gates and flip-flops connected by wires. We assume that the first two rules of combinational circuit are satisfied: Namely, wires connect outputs ports to inputs ports and every input port is fed exactly once.

We are now ready to decide whether C is a synchronous circuit. The decision is based on a reduction that replaces every flip-flop by fictive input and output gates as follows. For every flip-flop in C, we remove the flip-flop and add an output-gate instead of the input D of the flip-flop. Similarly, we add an input-gate instead of the output Q of the flip-flop. Now, we say that C is a synchronous circuit if C' is a combinational circuit. Figure 2 depicts a circuit C and the circuit C' obtained by removing the flip-flops in C.

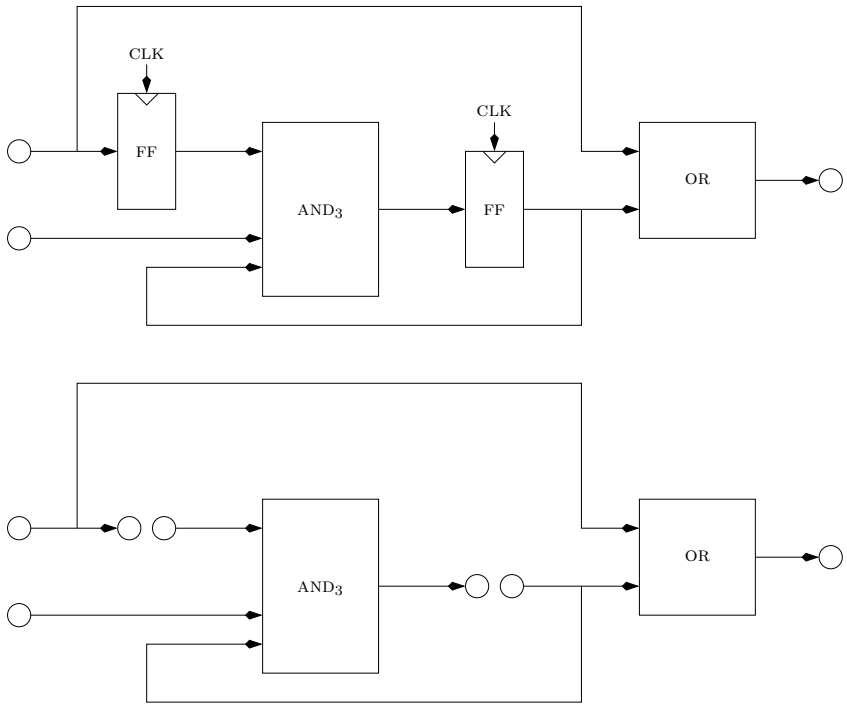

Figure 2: A synchronous circuit C and the corresponding combinational circuit C' after the flip-flops are removed.

Question 2.3 *Prove that every directed cycle in a synchronous circuit contains at least one*

flip-flop. (By cycle we mean a closed walk that obeys the output-to-input orientation.)

As in the case of combinational circuits, the definition of synchronous circuits is syntactic. The functionality of a synchronous circuit can be modeled by a finite state machine, defined below.

2.5 Finite state machines

A finite state machine (also known as a finite automaton with outputs or a transducer) is an abstract machine that is described by a 6-tuple: $\langle Q, q_0, \Sigma, \Delta, \delta, \gamma \rangle$ as follows. (1) Q is a finite set of states. (2) $q_0 \in Q$ is an initial state, namely, we assume that in clock cycle 0 the machine is in state q_0. (3) Σ is the input alphabet and Δ is the output alphabet. In each cycle, a symbol $\sigma \in \Sigma$ is fed as input to the machine, and the machine outputs a symbol $y \in \Delta$. (4) The *transition function* $\delta : Q \times \Sigma \to Q$ specifies the next state: If in cycle i the machine is in state q and the input equals σ, then, in cycle $i+1$, the state of the machine equals $\delta(q, \sigma)$. (5) The *output function* $\gamma : Q \times \Sigma \to \Delta$ specifies the output symbol as follows: when the state is q and the input is σ, then the machine outputs the symbol $\gamma(q, \sigma)$.

One often depicts a finite state machine using a directed graph called a *state diagram*. The vertices of the state diagrams are the set of states Q. We draw a directed edge (q', q'') and label the edge with a pair of symbols (σ, y), if $\delta(q', \sigma) = q''$ and $\gamma(q', \sigma) = y$. Note that every vertex in the state diagram has $|\Sigma|$ edges emanating from it.

We now explain how the functionality of a synchronous circuit C can be modeled by a finite state machine. Let F denote the set of flip-flops in C. For convenience, we index the flip-flops in F, so $F = \{f_1, \ldots, f_k\}$, where $k = |F|$. Let $f_j(i)$ denote the bit that is output by flip-flop $f_j \in F$ during the ith clock cycle. The set of states is $Q = \{0,1\}^k$. The state $q \in Q$ during cycle i is simply the binary string $f_1(i) \cdots f_k(i)$. The initial state q_0 is the binary string whose jth bit equals the value of $f_j(0)$ (recall that we assumed that each flip-flop is initialized so that it outputs a predetermined value in cycle 0). The input alphabet Σ is $\{0,1\}^{|I|}$, where I denotes the set of input gates in C. The jth bit of $\sigma \in \Sigma$ is the value of fed by the jth input gate. Similarly, the output alphabet Δ is $\{0,1\}^{|Y|}$, where Y denotes the set of output gates in C. The transition function is uniquely determined by C since C is a synchronous circuit. Namely, given a state $q \in Q$ and an input $\sigma \in \Sigma$, we apply logical simulation to the reduced combinational circuit C' to determine the values fed to each flip-flop. These values are well defined since C' is combinational. The vector of values input to the flip-flops by the end of clock cycle i is the state $q' \in Q$ during the next clock cycle. The same argument determines the output function.

We note that the definition given above for a finite state machine is often called a *Mealy machine*. There is another definition, called a *Moore machine*, that is a slightly more restricted [HopUll79]. In a Moore machine, the domain of the output function γ is Q rather than $Q \times \Sigma$. Namely, the output is determined by the current state, regardless of the current input symbol.

2.6 Cost and Delay

Every combinational circuit has a cost and a delay. The cost of a combinational circuit is the sum of the costs of the gates in the circuit. We use only gates with a constant number of inputs and outputs, and such gates have a constant cost. Since we are not interested in the constants, we simply attach a unit cost to every gate, and the cost of a combinational circuit equals the number of gates in the circuit.

The delay of a combinational circuit is defined similarly to the delay of a gate. Namely, the delay is the smallest amount of time required for the outputs to stabilize, assuming that the inputs are stable.

Let C denote a combinational circuit. The delay of a path p in the netlist $G(C)$ is the sum of the delays of the gates in p.

Theorem 2.4 *The delay of combinational circuit C is not greater than the maximum delay of a path in $G(C)$.*

Proof: Focus on a single gate g in C. Let $d(g)$ denote the delay of g. If all the inputs of g stabilize at time t, then we are guaranteed that g's outputs are stable at time $t + d(g)$. Note that $t + d(g)$ is an upper bound; in reality, g's output may stabilize much sooner.

We assume that all the inputs of C are stable at time $t = 0$. We now describe an algorithm that labels each net with a delay t that specifies when we are guaranteed that the signal on the net stabilizes. We do so by topologically sorting the gates in the netlist $G(C)$ (this is possible since $G(C)$ is acyclic). Now, we visit the gates according to the topological order. If g is an input gate, then we label the net that it feeds by zero. If g is not an input gate, then the topological order implies that all the nets that feed g have been already labeled. We compute the maximum label $t_{\max}$ appearing on nets that feed g (so that we are guaranteed that all the inputs of g stabilize by t_{max}). We now attach the label $t_{\max} + d(g)$ to all the nets that are fed by g. The statement in the beginning of the proof assures us that every output of g is indeed stable by time $t_{\max} + d(g)$.

It is easy to show by induction on the topological order that the label attached to nets fed by gate g equals the maximum delay of a path from an input gate to g. Hence, the theorem follows. □

To simplify the discussion, we attach a unit delay to every gate. With this simplification, the delay of a circuit C is the length of the longest path in the netlist graph $G(C)$.

We note that a synchronous circuit also has a cost that is the sum of the gate costs and flip-flop costs. Instead of a delay, a synchronous circuit has a minimum clock period. The minimum clock period equals the delay of the combinational circuit obtained after the flip-flops are removed (in practice, one should also add the propagation delay and the hold time of the flip-flops).

2.7 Notation

A binary string with n bits is often represented by an indexed vector $X[n-1:0]$. Note that an uppercase letter is used to denote the bit-vector. In this essay we use indexed vectors only with combinational circuits. So in a combinational circuit $X[i]$ always means the ith bit of the vector $X[n-1:0]$.

Notation of signals (i.e., external inputs and outputs, and interior values output by gates) in synchronous circuits requires referring to the clock cycle. We denote the value of a signal x during clock cycle t in a synchronous circuit by $x[t]$. Note that a lowercase letter is used to denote a signal in a synchronous circuit.

We use lowercase letter for synchronous circuits and uppercase letters for combinational circuits. Since we deal with very simple synchronous circuits, this distinction suffices to avoid confusion between $x[i]$ and $X[i]$: The symbol $x[i]$ means the value of the signal x in the ith clock cycle. The symbol $X[i]$ means the ith bit in the vector $X[n-1:0]$.

Referring to the value of a signal (especially in a synchronous circuit) introduces some confusion due to the fact that signals change all the time from zero to one and vice-versa. Moreover, during these transitions, there is a (short) period of time during which the value of the signal is neither zero or one. The guiding principle is that we are interested in the stable value of a signal. In the case of a combinational circuit this means that we wait sufficiently long after the inputs are stable. In the case of a synchronous circuit, the functionality of flip-flops implies that the outputs of each flip-flop are stable shortly after a clock cycle begins and remain stable till the end of the clock cycle. Since the clock period is sufficiently long, all other nets stabilize before the end of the clock cycle.

2.8 Representation of numbers

Our goal is to design fast adders. For this purpose we must agree on how nonnegative integers are represented. Throughout this essay we use binary representation. Namely, a binary string $X[n-1:0] \in \{0,1\}^n$ represents the number $\sum_{i=0}^{n-1} X[i] \cdot 2^i$.

In a synchronous circuit a single bit signal $x[t]$ can be also used to represent a nonnegative integer. The ith bit equals $x[i]$, and therefore, the number represented by $x[t]$ equals $\sum_{t=0}^{n-1} x[t]$.

Although students are accustomed to binary representation, it is useful to bear in mind that this is not the only useful representation. Negative numbers require a special representation, the most common is known as two's complement. In two's complement representation, the binary string $X[n-1:0]$ represents the integer $-2^{n-1} \cdot X[n-1] + \sum_{i=0}^{n-2} X[i] \cdot 2^i$.

There are representations in which the same number may be have more than one representation. Such representations are called *redundant representations*. Interestingly, redundant representations are very useful. One important example is carry-save representation. In carry-save representation, a nonnegative integer is represented by two binary strings (i.e., two bits are used for each digit). Each binary string represents an integer in binary representation, and the number represented by two such binary strings is the sum of the numbers represented by the two strings. An important property of carry-save representation is that addition in carry-save representation can be computed with constant delay (this can be done by using only full-adders). Addition with constant delay is vital for the design of fast multipliers.

3 Definition of a binary adder

Everybody knows that computers compute arithmetic operations; even a calculator can do it! So it is hardly a surprise that every computer contains a hardware device that adds numbers. We refer to such a device as an *adder*. Suppose we wish to design an adder. Before we start discussing how to design an adder, it is useful to specify or define exactly what we mean by this term.

3.1 Importance of specification

Unfortunately, the importance of a formal specification is not immediately understood by many students. This is especially true when it comes to seemingly obvious tasks such as addition. However, there are a few issues that the specification of an adder must address.

Representation: How are addends represented? The designer must know how numbers are represented. For example, an adder of numbers represented in unary representation is

completely different than an adder of numbers represented in binary representation. The issue of representation is much more important when we consider representations of signed numbers (e.g., two's complement and one's complement) or redundant representations (e.g., carry-save representation).

Another important issue is how to represent the computed sum? After all, the addends already represent the sum, but this is usually not satisfactory. A reasonable and useful assumption is to require the same representation for the addends and the sum (i.e., binary representation in our case). The main advantage of this assumption is that one could later use the sum as an addend for subsequent additions.

In this essay we consider only binary representation.

Model: How are the inputs fed to the circuit and how is the output obtained? We consider two extremes: a combinational circuit and a synchronous circuit. In a combinational circuit, we assume that there is a dedicated port for every bit of the addends and the sum. For example, if we are adding two 32-bit numbers, then there are 32×3 ports; 32×2 ports are input ports and 32 ports are output ports. (We are ignoring in this example the carry-in and the carry-out ports.)

In a synchronous circuit, we consider the bit-serial model in which there are exactly two input ports and one output port. Namely, there are exactly three ports regardless of the length of the addends. The synchronous circuit is very easy to design and will serve as a starting point for combinational designs. In Section 3.3 we present a bit-serial adder.

We are now ready to specify (or define) a combinational binary adder and a serial binary adder. We specify the combinational adder first, but design a serial adder first. The reason is that we will use the implementation of the serial adder to design a simple combinational adder.

3.2 Combinational adder

Definition 3.1 *A* combinational binary adder *with input length n is a combinational circuit specified as follows.*

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$.

Output: $S[n-1:0] \in \{0,1\}^n$ *and* $C[n] \in \{0,1\}$.

Functionality:

$$C[n] \cdot 2^n + \sum_{i=0}^{n-1} S[i] \cdot 2^i = \sum_{i=0}^{n-1} A[i] \cdot 2^i + \sum_{i=0}^{n-1} B[i] \cdot 2^i. \tag{1}$$

We denote a combinational binary adder with input length n by ADDER(n). The inputs $A[n-1:0]$ and $B[n-1:0]$ are the binary representations of the addends. Often an additional input $C[0]$, called the *carry-in bit*, is used. To simplify the presentation we omit this bit at this stage (we return to it in Section 7.1). The output $S[n-1:0]$ is the binary representation of the sum modulo 2^n. The output $C[n]$ is called the *carry-out bit* and is set to 1 if the sum is at least 2^n.

Question 3.2 *Verify that the functionality of* ADDER(n) *is well defined. Show that, for every $A[n-1:0]$ and $B[n-1:0]$ there exist unique $S[n-1:0]$ and $C[n]$ that satisfy Equation 1.*

Hint: Show that the set of integers that can be represented by sums $A[n-1:0]+B[n-1:0]$ is contained in the set of integers that can be represented by sums $S[n-1:0] + 2^n \cdot C[n]$.

There are many ways to implement an ADDER(n). Our goal is to present a design of an ADDER(n) with optimal asymptotic delay and cost. In Sec. 5 we prove that every design of an ADDER(n) must have at least logarithmic delay and linear cost.

3.3 Bit-serial adder

We now define a synchronous adder that has two inputs and a single output.

Definition 3.3 *A* bit-serial binary adder *is a synchronous circuit specified as follows.*

Input ports: $a, b \in \{0, 1\}$.

Output ports: $s \in \{0, 1\}$.

Functionality: *For every clock cycle* $n \geq 0$,

$$\sum_{i=0}^{n} s[i] \cdot 2^i = \sum_{i=0}^{n} (a[i] + b[i]) \cdot 2^i \pmod{2^n}. \tag{2}$$

We refer to a bit-serial binary adder by s-adder. One can easily see the relation between $a[i]$ (e.g., the bit input in clock cycle i via port a) and $A[i]$ (e.g., the ith bit of the addend A). Note the lack of a carry-out in the specification of a s-adder.

4 Trivial designs

In this section we present trivial designs for an s-adder and an ADDER(n). The combinational adder is obtained from the synchronous one by applying a time-space transformation.

4.1 A bit-serial adder

In this section we present a design of a bit-serial adder. The design performs addition in the same way we are taught to add in school (i.e., from the least significant digit to the most significant digit). Figure 3 depicts a design that uses one flip-flop and one full-adder. The output of the flip-flop that is input to the full-adder is called the carry-in signal and is denoted by c_{in}. The carry output of the full-adder is input to the flip-flop, and is called the carry-out signal. It is denoted by c_{out}. We ignore the issue of initialization and assume that, in clock cycle zero, $c_{in}[0] = 0$.

We now present a correctness proof of the s-adder design. The proof is by induction, but is not totally straightforward. (Definitely not for undergraduate hardware design students!) The problem is that we need to strengthen Eq. 2 (we point out the difficulty within the proof). Another reason to insist on a complete proof is that, for most students, this is the first time they prove the correctness of the addition algorithm taught in school.

Claim 4.1 *The circuit depicted in Fig. 3 is a correct implementation of a s-adder.*

Proof: We prove that the circuit satisfies Eq. 2. The proof is by induction on the clock cycle. The induction basis, for the clock cycle zero (i.e., $n = 0$), is easy to prove. In clock cycle zero, the inputs are $a[0]$ and $b[0]$. In addition $c_{in}[0] = 0$. By the definition of a full-adder, the output $s[0]$ equals XOR$(a[0], b[0], c_{in}[0])$. It follows that $s[0] = \text{mod}(a[0] + b[0], 2)$, as required.

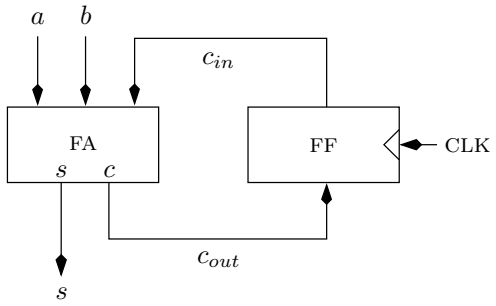

Figure 3: A schematic of a serial adder.

We now try to prove the induction step for $n+1$. Surprisingly, we are stuck. If we try to apply the induction hypothesis to the first n cycles, then we cannot claim anything about $c_{in}[n+1]$. (We do know that $c_{in}[n+1] = c_{out}[n]$, but Eq. 2 does not tell us anything about the value of $c_{out}[n]$.) On the other hand we cannot apply the induction hypothesis to the last n cycles (by decreasing the indexes of clock cycles by one) because $c_{in}[1]$ might equal 1.

The way to overcome this difficulty is to strengthen the statement we are trying to prove. Instead of Eq. 2, we prove the following stronger statement: For every clock cycle $n \geq 0$,

$$2^{n+1} \cdot c_{out}[n] + \sum_{i=0}^{n} s[i] \cdot 2^i = \sum_{i=0}^{n} (a[i] + b[i]) \cdot 2^i. \tag{3}$$

We prove Equation 3 by induction. When $n = 0$, Equation 3 follows from the functionality of a full-adder and the assumption that $c_{in}[0] = 0$. So now we turn again to the induction step, namely, we prove that Eq. 3 holds for $n+1$.

The functionality of a full-adder states that

$$2 \cdot c_{out}[n+1] + s[n+1] = a[n+1] + b[n+1] + c_{in}[n+1]. \tag{4}$$

We multiply both sides of Eq. 4 by 2^{n+1} and add it to Eq. 3 to obtain

$$2^{n+2} \cdot c_{out}[n+1] + 2^{n+1} \cdot c_{out}[n] + \sum_{i=0}^{n+1} s[i] \cdot 2^i = 2^{n+1} \cdot c_{in}[n+1] + \sum_{i=0}^{n+1} (a[i] + b[i]) \cdot 2^i. \tag{5}$$

The functionality of the flip-flop implies that $c_{out}[n] = c_{in}[n+1]$, and hence, Eq. 3 holds also for $n+1$, and the claim follows. □

4.2 Ripple-carry adder

In this section we present a design of a combinational adder ADDER(n). The design we present is called a *ripple-carry adder*. We abbreviate and refer to a ripple-carry adder for binary numbers of length n as RCA(n). Although designing an RCA(n) from scratch is easy, we obtain it by applying a transformation, called a *time-space transformation*, to the bit-serial adder.

Ever since Ami Litman and my father introduced me to retiming [LeiSaxe81, LeiSaxe91, EvenLitman91, EvenLitman94], I thought it is best to describe designs by functionality preserving transformations. Namely, instead of obtaining a new design from scratch, obtain it

by transforming a known design. Correctness of the new design follows immediately if the transformation preserves functionality. In this way a simple design evolves into a sophisticated design with much better performance. Students are rarely exposed to this concept, so I chose to present the ripple-carry adder as the outcome of a time-space transformation applied to the serial adder.

Time-space transformation. We apply a transformation called a *time-space transformation*. This is a transformation that maps a directed graph (possibly with cycles) to an acyclic directed graph. In the language of circuits, this transformation maps synchronous circuits to combinational circuits.

Given a synchronous circuit C, construct a directed multi-graph $G = (V, E)$ with non-negative integer weights $w(e)$ defined over the edges as follows. This graph is called the *communication graph* of C [LeiSaxe91, EvenLitman94]. The vertices are the combinational gates in C (including branching points). An edge (u, v) in a communication graph models a p path in the netlist, all the interior nodes of which correspond to flip-flops. Namely, we draw an edge from u to v if there is a path in C from an output of u to an input v that traverses only flip-flops. Note that a direct wire from u to v also counts as a path with zero flip-flops. The weight of the edge (u, v) is set to the number of flip-flops along the path. Note that there might be several paths from u to v, each traversing a different number of flip-flops. For each such path, we add a parallel edge with the correct weight. Finally, recall that branching points are considered to be combinational gates, so such a path may not traverse a branching point. In Figure 4 a synchronous circuit and its communication graph are depicted. Following [LeiSaxe81], we depict the weight of an edge by segments across an edge (e.g., two segments mean that the weight is two).

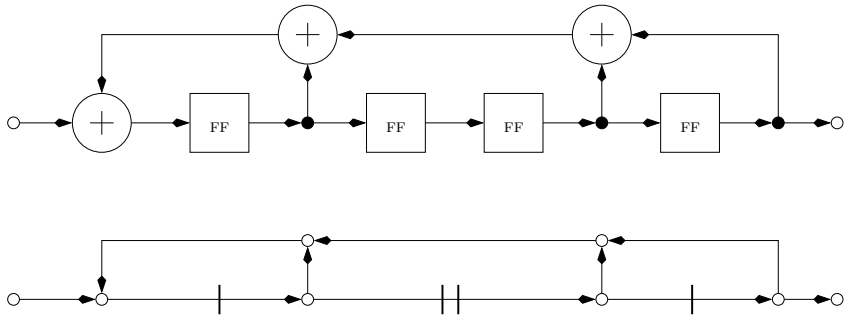

Figure 4: A synchronous circuits and the corresponding communication graph (the gates are XOR gates).

The weight of a path in the communication graph is the sum of the edge weights along the path. The following claim follows from the definition of a synchronous circuit.

Claim 4.2 *The weight of every cycle in the communication graph of a synchronous circuit is greater than zero.*

Question 4.3 *Prove Claim 4.2*

Let n denote a parameter that specifies the number cycles used in the time-space transformation. The time-space transformation of a communication graph $G = (V, E)$ is the directed graph

$st_n(G) = (V \times \{0, \ldots, n-1\}, E')$ defined as follows. The vertex set is the Cartesian product of V and $\{0, \ldots, n-1\}$. Vertex (v, i) is a copy of v that corresponds to clock cycle i. There is an edge from (u, i) to (v, j) if: (i) $(u, v) \in E$, and (ii) $w(u, v) = j - i$. The edge $((u, i), (v, j))$ corresponds to the data transmitted from u to v in clock cycle i. Since there are $w(u, v)$ flip-flops along the path, the data arrives to v only in clock cycle $j = i + w(u, v)$.

Since the weight of every directed cycle in G is greater than zero, it follows that $st_n(G)$ is acyclic. We now build a combinational circuit C_n whose communication graph is $st_n(G)$. This is done by placing a gate of type v for each vertex (v, i). The connections use the same ports as they do in C.

There is a boundary problem that we should address. Namely, what feeds input ports of vertices (v, j) that "should" be fed by a vertex (u, i) with a negative index i? We encounter a similar problem with output ports of a vertex (u, i) that should feed a vertex (v, j) with an index $j \geq n$. We solve this problem by adding input gates that feed vertices (v, j) where $j < w(u, v)$. Similarly, we add output gates that are fed by vertices (u, i) where $i + w(u, v) > n$.

We are now ready to apply the time-space transformation to the bit-serial adder. We apply it for n clock cycles. The input gate a has now n instances that feed the signals $A[0], \ldots, A[n-1]$. The same holds for the other input b and the output s. The full-adder has now n instances denoted by $\text{FA}_0, \ldots, \text{FA}_{n-1}$. We are now ready to describe the connections. Since the input a feeds the full-adder in the bit-serial adder, we now use input $A[i]$ to feed the full-adder FA_i. Similarly, the input $B[i]$ feeds the full-adder FA_i. The carry-out signal c_{out} of the full-adder in the bit-serial adder is connected via a flip-flop to one of the inputs of the full-adder. Hence, we connect the carry-out port of full-adder FA_i to one of the inputs of FA_{i+1}. Finally, the output $S[i]$ is fed by the sum output of full-adder FA_i. Note that full-adder FA_0 is also fed by a "new" input gate that carries the carry-in signal $C[0]$. The carry-in signal is the initial state of the serial adder. The full-adder FA_{n-1} feeds a "new" output gate with the carry-out signal $C[n]$. Figure 5 depicts the resulting combinational circuit known as a ripple-carry adder. The netlist of the ripple-carry adder is simple and regular, and therefore, one can easily see that the outputs of FA_i depend only on the inputs $A[i:0]$ and $B[i:0]$.

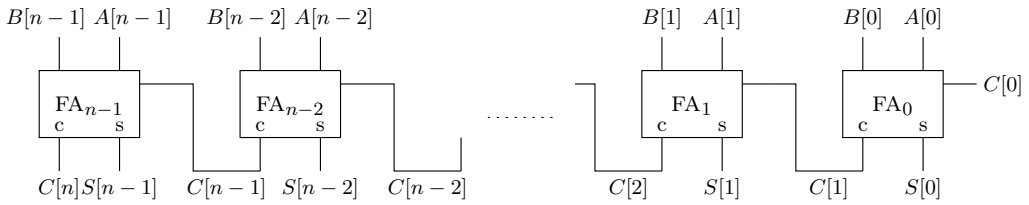

Figure 5: A ripple-carry adder.

The main advantage of using the time-space transformation is that this transformation preserves functionality. Namely, there is a value-preserving mapping between the value of a signal $x[i]$ in clock cycle i in the synchronous circuit and the value of the corresponding signal in the combinational circuit. The main consequence of preserving functionality is that the correctness of $\text{RCA}(n)$ follows.

Question 4.4 *Write a direct proof of the correctness of a ripple-carry adder* $\text{RCA}(n)$*. (That is, do not rely on the time-space transformation and the correctness of the bit-serial adder.)*

4.3 Cost and Delay

The bit-serial adder consists of a single full-adder and a single bit flip-flop. It follows that the cost of the bit-serial adder is constant. The addition of n-bit numbers requires n clock cycles.

The ripple-carry adder RCA(n) consists of n full-adders. Hence, its cost is linear in n. The delay is also linear since the path from $A[0]$ to $S[n-1]$ traverses all the n full-adders.

5 Lower bounds

We saw that RCA(n) has a linear cost and a linear delay. Before we look for better adder designs, we address the question of whether there exist cheaper or faster adders. For this purpose we need to prove lower bounds.

Lower bounds are rather mysterious to many students. One reason is mathematical in nature; students are not used to arguing about unknown abstract objects. The only lower bound that they probably encountered so far is the lower bound on the number of comparisons needed for sorting. Another reason is that they have been noticing a continuous improvement in computer performance and an ongoing reduction in computer costs. So many students are under the impression that there is no limit to faster and cheaper circuits. This impression might be even "well founded" if they heard about "Moore's Law". Finally, they are accustomed to thinking that better ways are awaiting the ingenious or lucky inventor.

We first state the lower bounds for the delay and cost of binary adders.

Theorem 5.1 *Assume that the number of inputs of every combinational gate is bounded by c. Then, for every combinational circuit G that implements an* ADDER(n)*, the following hold: $c(G) \geq n/c$ and $d(G) \geq \log_c n$.*

How does one prove this theorem? The main difficulty is that we are trying prove something about an unknown circuit. We have no idea whether there exist better ways to add. Perhaps some strange yet very simple Boolean function of certain bits of the addends can help us compute the sum faster or cheaper? Instead of trying to consider all possible methods for designing adders, we rely on the simplest properties that every adder must have. In fact, the proof is based on topological properties common to every adder. There are two topological properties that we use: (i) There must be a path from every input to the output $S[n-1]$. (ii) The number of inputs of every combinational gate is bounded by c. Hence the proof of Theorem 5.1 reveals an inherent limitation of combinational circuits rather than incompetence of designers.

Question 5.2 *Prove Theorem 5.1.*

Hint: Show that the output bit $S[n]$ depends on all the inputs. This means that one cannot determine the value of $S[n]$ without knowing the values of all the input bits. Prove that in every combinational circuit in which the output depends on all the inputs the delay is at least logarithmic and the cost is at least linear in the number of inputs. Rely on the fact that the number of inputs of each gate is at most c.

Returning the ripple-carry adder RCA(n), we see that its cost is optimal (upto a constant). However, its delay is linear. The lower bound is logarithmic, so much faster adders might exist. We point out that, in commercial microprocessors, 32-bit numbers are easily added within a

single clock cycle. (In fact, in some floating point units numbers over 100 bits long are added within a clock cycle.) Clock periods in contemporary microprocessors are rather short; they are shorter than 10 times the delay of a full-adder. This means that even the addition of 32-bit numbers within one clock cycle requires faster adders.

6 The adder of Ladner and Fischer

In this section we present an adder design whose delay and cost are asymptotically optimal (i.e., logarithmic delay and linear cost).

6.1 Motivation

Let us return to the bit-serial adder. The bit-serial adder is fed one bit of each addend in each clock cycle. Consider the finite-state diagram of the bit-serial adder depicted in Fig. 6. In this diagram there are two states: 0 and 1 (the state is simply the value stored in the flip-flop, namely, the carry-bit from the previous position). The computation of the bit-serial adder can be described as a walk in this diagram that starts in the initial state 0. If we know the sequence of the states in this walk, then we can easily compute the sum bits. The reason for this is that the output from state $q \in \{0,1\}$ when fed inputs $A[i]$ and $B[i]$ is $\textsc{xor}(q, A[i], B[i])$.

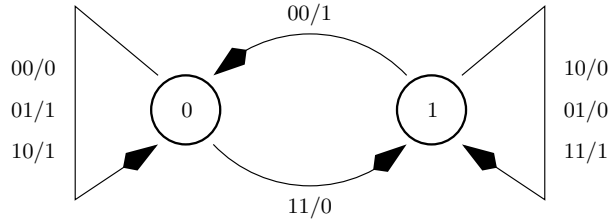

Figure 6: The state diagram of the bit-serial adder.

We now consider the goal of *parallelizing* the bit-serial adder. By paralleling we mean the following. We know how to compute the sum bits if the addends are input one bit at a time. Could we compute the sum faster if we knew all the bits of the addends from the beginning? Another way to state this question is to ask whether we could quickly reconstruct the sequence of states that are traversed.

Perhaps it is easier to explain our goal if we consider a state-diagram with several states (rather than just two states). Let $\sigma_i \in \Sigma$ denote the symbol input in cycle i. We assume that in cycle zero the machine is in state q_0. When fed by the input sequence $\sigma_0, \ldots, \sigma_{n-1}$, the machine walks through the sequence of states $q_0, \ldots, q_n$ defined by transition function δ, as follows: $q_{i+1} = \delta(q_i, \sigma_i)$. The output sequence $y_0, y_1, \ldots, y_{n-1}$ is defined by output function γ, as follows: $y_i = \gamma(q_i, \sigma_i)$. We can interpret the inputs as "driving instructions" in the state diagram. Namely, in each cycle, the inputs instruct us how to proceed in the walk (i.e., "turn left", "turn right", or "keep straight"). Now, we wish to quickly reconstruct the walk from the sequence of n instructions.

Suppose we try to split this task among n players. The ith player is given the input symbol σ_i and performs some computation based on σ_i. After this, the players meet, and try to quickly glue together their pieces. Each player is confronted with the problem of simplifying the task of

gluing. The main obstacle is that each player does not know the state q_i of the machine when the input symbol σ_i is input. At first, it seems that knowing q_i is vital if, for example, players i and $i+1$ want to combine forces.

Ladner and Fisher proposed the following approach. Each player i computes (or, more precisely chooses) a restricted transition function $\delta_i : S \to S$, defined by $\delta_i(q) = \delta(q, \sigma_i)$. One can obtain a "graph" of δ_i if one considers only the edges in the state diagram that are labeled with the input symbol σ_i. By definition, if the initial state is q_0, then $q_1 = \delta_0(q_0)$. In general, $q_i = \delta_{i-1}(q_{i-1})$, and hence, $q_i = \delta_{i-1}(\delta_{i-2}(\cdots(\delta_0(q_0))\cdots)$. It follows that player i is satisfied if she computes the composition of the functions $\delta_{i-1}, \ldots, \delta_0$. Note that the function δ_i is determined by the input symbol σ_i, and the functions δ_i are selected from a fixed collection of $|\Sigma|$ functions.

Before we proceed, there is a somewhat confusing issue that we try to clarify. We usually think of an input σ_i as a parameter that is given to a function (say, f), and the goal is to evaluate $f(\sigma)$. Here, the input σ_i is used to select a function δ_i. We do not evaluate the function δ_i; instead, we compose it with other functions.

We denote the composition of two function f and g by $f \circ g$. Note that $(f \circ g)(q) \stackrel{\triangle}{=} f(g(q))$. Namely, g is applied first, and f is applied second.

We denote the composition of the functions $\delta_i, \ldots, \delta_0$ by the function π_i. Namely, $\pi_0 = \delta_0$, and $\pi_{i+1}(q) = \delta_{i+1}(\pi_i(q))$. Assume that, given representations of two functions f and g, the representation of the composition $f \circ g$ can be computed in constant time. This implies that the function π_{n-1} can be, of course, computed with linear delay. The goal in parallelizing the computation is to compute π_{n-1} with logarithmic delay. Moreover, we wish to compute all the functions $\pi_0, \ldots, \pi_{n-1}$ with logarithmic delay and with overall linear cost.

Recall that our goal is to compute the output sequence rather than the sequence of states traversed by the state machine. Obviously, if we compute the walk $q_0, \ldots, q_{n-1}$, then we can easily compute the output sequence simply by $y_i = \gamma(q_i, \sigma_i)$. Hence, each output symbol y_i can be computed with constant delay and cost after q_i is computed. This means that we have reduced the problem of parallelizing the computation of a finite state machine to the problem of parallelizing the computation of compositions of functions.

Remember that our goal is to design an optimal adder. So the finite state machine we are interested in is the bit-serial adder. There are four possible input symbols corresponding to the values of the bits $a[i]$ and $b[i]$. The definition of full-adder (and the state diagram), imply that the function δ_i satisfies the following condition for $q \in \{0, 1\}$:

$$\delta_i(q) = \begin{cases} 0 & \text{if } q + a[i] + b[i] < 2 \\ 1 & \text{if } q + a[i] + b[i] \geq 2. \end{cases}$$

$$= \begin{cases} 0 & \text{if } a[i] + b[i] = 0 \\ q & \text{if } a[i] + b[i] = 1. \\ 1 & \text{if } a[i] + b[i] = 2. \end{cases} \tag{6}$$

In the literature the sum $a[i]+b[i]$ is often represented using the carry "kill/propagate/generate" signals [CLR90]. This terminology is justified by the following explanation.

- When $a[i] + b[i] = 0$, it is called "kill", because the carry-out is zero. In Eq. 6, we see that when $a[i] + b[i] = 0$, the value of the function δ_i is always zero.

- When $a[i] + b[i] = 1$, it is called "propagate", because the carry-out equals the carry-in. In Eq. 6, we see that when $a[i] + b[i] = 1$, the function δ_i is the identity function.
- When $a[i] + b[i] = 2$, it is called "generate", because the carry-out is one. In Eq. 6, we see that when $a[i] + b[i] = 2$, the value of the function δ_i is always one.

From this discussion, it follows that the "kill/propagate/generate" jargon (also known as k, p, g) is simply a representation of $a[i] + b[i]$. We prefer to abandon this tradition and use a different notation described below.

Equation 6 implies that, in a bit-serial adder, δ_i can be one of three functions: the zero function (i.e., value is always zero), the one function (i.e., value is always one), and the identity function (i.e., value equals the parameter). We denote the zero function by f_0, the one function by f_1, and the identity function by f_{id}.

The fact that these three functions are closed under composition can be easily verified since for every $x \in \{0, id, 1\}$:

$$f_0 \circ f_x = f_0$$
$$f_{id} \circ f_x = f_x$$
$$f_1 \circ f_x = f_1.$$

Finally, we point out that the "multiplication" table presented for the operator defined over the alphabet $\{k, p, g\}$ is simply the table of composing the functions f_0, f_{id}, f_1 (see, for example, [CLR90]).

6.2 Associativity of composition

Before we continue, we point out an important property of compositions of functions whose domain and range are identical (e.g., Q is both the domain and the range of all the functions δ_i).

Consider a set Q and the set $\mathcal{F}$ of all functions from Q to Q. An *operator* is a function $\star : \mathcal{F} \times \mathcal{F} \to \mathcal{F}$. (One could define operators $\star : A \times A \to A$ with respect to any set A. Here we need only operators with respect to $\mathcal{F}$.) Since $\star$ is a dyadic function, we denote by $f \star g$ the image of $\star$ when applied to f and g.

Composition of functions is perhaps the most natural operator. Given two functions, $f, g \in \mathcal{F}$, the composition of f and g is the function h defined by $h(q) = f(g(q))$. We denote the composition of f and g by $f \circ g$.

Definition 6.1 *An operator $\star : \mathcal{F} \times \mathcal{F} \to \mathcal{F}$ is* associative *if, for every three functions $f, g, h \in \mathcal{F}$, the following holds:*

$$(f \star g) \star h = f \star (g \star h).$$

We note that associativity is usually defined for dyadic functions, namely, $f(a, f(b, c)) = f(f(a, b), c)$. Here, we are interested in operators (i.e., functions of functions), so we consider associativity of operators. Of course, associativity means the same in both cases, and one should not be confused by the fact that the value of an operator is a function.

We wish to prove that composition is an associative operator. Although this is an easy claim, it is often hard to convince the students that it is true. One could prove this by a reduction to multiplication of zero-one matrices. Instead, we provide a "proof by diagram".

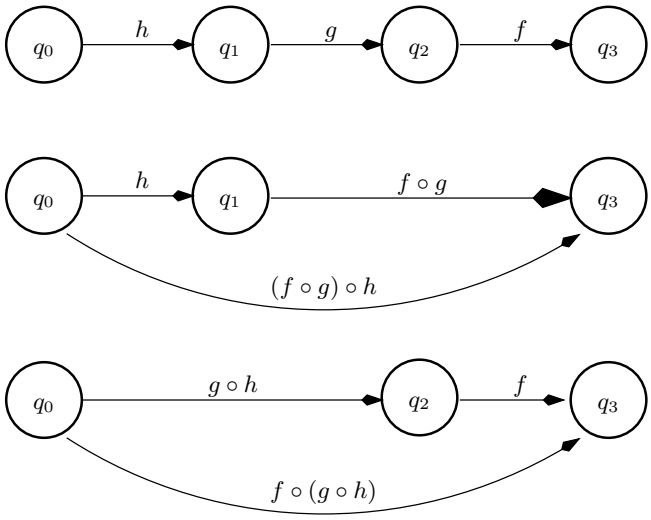

Figure 7: Composition of functions is associative.

Claim 6.2 *Composition of functions is associative.*

Proof: Consider three functions $f, g, h \in \mathcal{F}$ and an element $q_0 \in Q$. Let $q_1 = h(q_0)$, $q_2 = g(q_1)$, and $q_3 = f(q_2)$. In Figure 7 we depict the compositions $(f \circ g) \circ h$ and $f \circ (g \circ h)$. In the second line of the figure we depict the following.

$$\begin{aligned}((f \circ g) \circ h)(q_0) &= (f \circ g)(h(q_0)) \\ &= f(g(h(q_0))).\end{aligned}$$

In third line of the figure we depict the following.

$$\begin{aligned}f \circ (g \circ h)(q_0) &= f((g \circ h)(q_0)) \\ &= f(g(h(q_0))).\end{aligned}$$

Hence ,the claim follows. □

The associativity of an operator allows us to write expressions without parenthesis. Namely, $f_1 \circ f_2 \circ \cdots \circ f_n$ is well defined.

Question 6.3 *Let $Q = \{0, 1\}$. Find an operator in $\mathcal{F}$ that is not associative.*

Interestingly, in most textbooks that describe parallel-prefix adders, an associative dyadic function is defined over the alphabet $\{k, p, g\}$ (or over its representation by the two bits p and g). This associative function is usually presented without any justification or motivation. As mentioned at the end of Sec. 6.1, this function is simply the composition of f_0, f_{id}, f_1. The associativity of this operator is usually presented as a special property of addition. In this section we showed this is not the case. The special property of addition is that the functions $\delta(\cdot, \sigma)$ are closed under composition. Associativity, on the other hand, is simply the associativity of composition.

6.3 The parallel prefix problem

In Section 6.1, we reduced the problem of designing a fast adder to the problem of computing compositions of functions. In Section 6.2, we showed that composition of functions is an associative operator. This motivates the definition of the prefix problem.

Definition 6.4 *Consider a set $\mathcal{A}$ and an associative operator $\star : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$. The* prefix problem *is defined as follows.*

Input: $\delta_0, \ldots, \delta_{n-1} \in \mathcal{A}$.

Output: $\pi_0, \ldots, \pi_{n-1} \in \mathcal{A}$ *defined by* $\pi_0 = \delta_0$ *and* $\pi_i = \delta_i \star \cdots \star \delta_0$, *for* $0 < i < n$. *(Note that* $\pi_{i+1} = \delta_{i+1} \star \pi_i$.*)*

Assume that we have picked a (binary) representation for elements in $\mathcal{A}$. Moreover, assume that we have an implementation of the associative operator $\star$ with respect to this implementation. Namely, a $\star$-gate is a combinational gate that when fed two representations of elements $f, g \in \mathcal{A}$, outputs a representation of $f \star g$. Our goal now is to design a fast circuit for solving the prefix problem using $\star$-gates. Namely, our goal is to design a parallel prefix circuit.

Note that the operator $\star$ need not be commutative, so the inputs of the $\star$-gate cannot be interchanged. This means that there is a difference between the left input and the right input of a $\star$-gate.

6.4 The parallel prefix circuit

We are now ready to describe a combinational circuit for the prefix problem. We will use only one building block, namely, a $\star$-gate. We assume that the cost and the delay of a $\star$-gate are constant (i.e., do not depend on n).

We begin by considering two circuits; one with optimal cost and the second with optimal delay. We then present a circuit with optimal cost and delay.

Linear cost but linear delay. Figure 8 depicts a circuit for the prefix problem with linear cost. The circuit contains $(n-1)$ copies of $\star$-gates, but its delay is also linear. In fact, this circuit is very similar to the ripple carry adder.

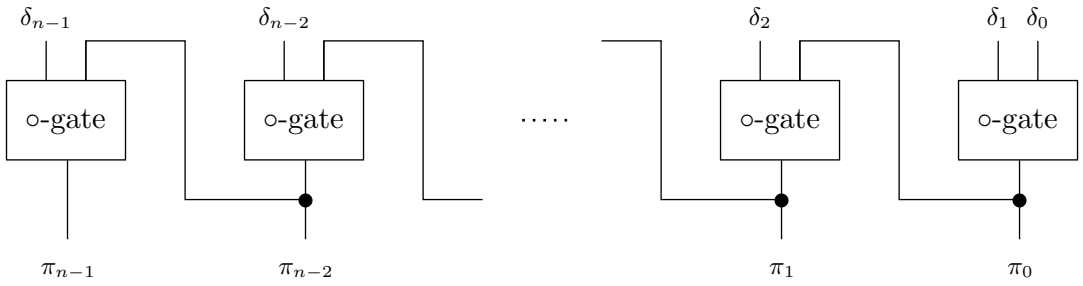

Figure 8: A prefix computation circuit with linear delay.

Logarithmic delay but quadratic cost. The ith output π_i can be computed by circuit with the topology of a balanced binary tree, where the inputs are fed from the leaves, a $\star$-gate is placed in each node, and π_i is output by the root. The circuit contains $(i-1)$ copies of $\star$-gates and its delay is logarithmic in i. We could construct a separate tree for each π_i to obtain n circuits with logarithmic delay but quadratic cost.

Our goal now is to design a circuit with logarithmic delay and linear cost. Intuitively, the design based on n separate trees is wasteful because the same computations are repeated in different trees. Hence, we need to find an efficient way to "combine" the trees so that computations are not repeated.

Parallel prefix computation. We now present a circuit called PPC(n) for the prefix problem. The design we present is a recursive design. For simplicity, we assume that n is a power of 2. The design for $n = 2$ simply outputs $\pi_0 \leftarrow \delta_0$ and $\pi_1 \leftarrow \delta_1 \star \delta_0$. The recursion step is depicted in Figure 9. Adjacent inputs are paired and fed to a $\star$-gate. The $n/2$ outputs of the $\star$-gates are fed to a PPC($n/2$) circuit. The outputs $\pi'_0, \ldots, \pi'_{n/2-1}$ of the PPC($n/2$) circuit are directly connected to the odd indexed outputs, namely, $\pi_{2i+1} \leftarrow \pi'_i$. Observe that wires carrying the inputs with even indexes are drawn (or routed) over the PPC($n/2$) box; these "even indexed" wires are not part of the PPC($n/2$) design. The even indexed outputs (for $i > 0$) are obtained as follows: $\pi_{2i} \leftarrow \delta_{2i} \star \pi'_{i-1}$.

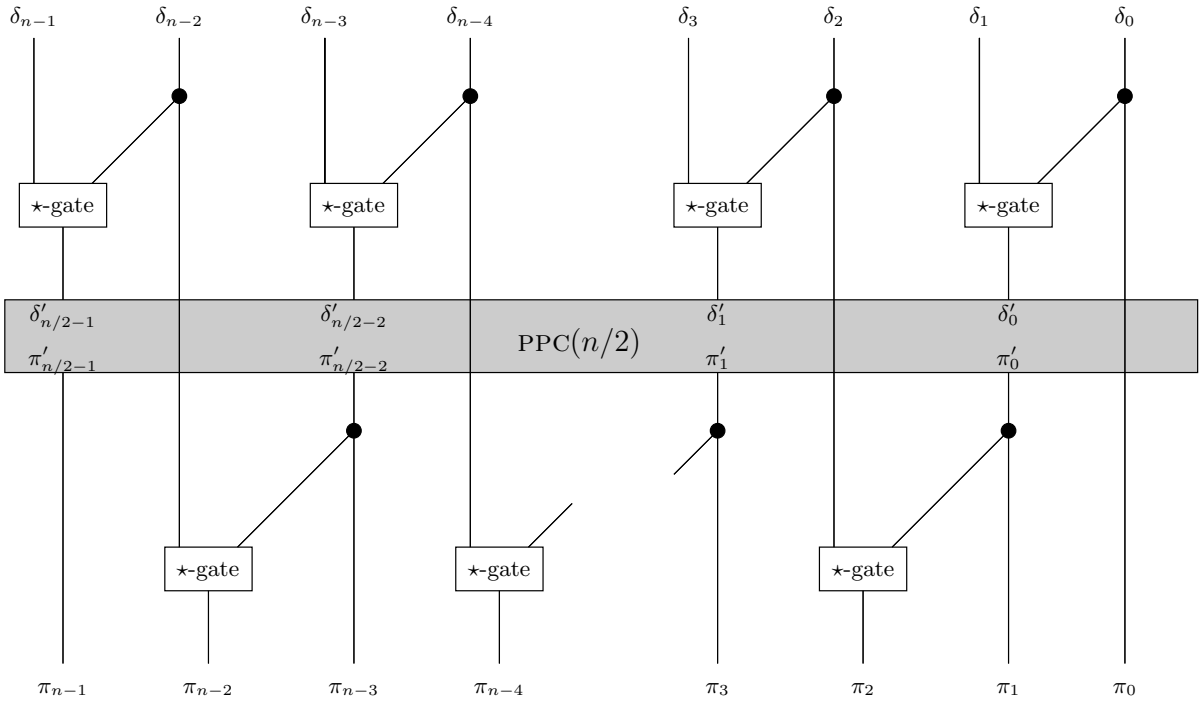

Figure 9: A recursive design of PPC(n). (The wires that pass over the PPC($n/2$) box carry the even indexed inputs $\delta_0, \delta_2, \ldots, \delta_{n-2}$. These signals are not part of the PPC($n/2$) circuit. The wires are drawn in this fashion only to simplify the drawing.)

6.5 Correctness

Claim 6.5 *The design depicted in Fig. 9 is correct.*

Proof: The proof of the claim is by induction. The induction basis holds trivially for $n = 2$. We now prove the induction step. Consider the PPC$(n/2)$ used in a PPC(n). Let δ'_i and π'_i denote the inputs and outputs of the PPC$(n/2)$, respectively. The ith input $\delta'[i]$ equals $\delta_{2i+1} \star \delta_{2i}$. By associativity and the induction hypothesis, the ith output π'_i satisfies:

$$\begin{aligned}\pi'_i &= \delta'_i \star \cdots \delta'_0 \\ &= (\delta_{2i+1} \star \delta_{2i}) \star \cdots \star (\delta_1 \star \delta_0)\end{aligned}$$

It follows that the output π_{2i+1} equals the composition $\delta_{2i+1} \star \cdots \star \delta_0$, as required. Hence, the odd indexed outputs $\pi_1, \pi_3, \ldots, \pi_{n-1}$ are correct.

Finally, output in position $2i$ equals $\delta_{2i} \star \pi'_{i-1} = \delta_{2i} \star \pi_{2i-1} = \delta_{2i} \star \cdots \star \delta_0$. It follows that the even indexed outputs are also correct, and the claim follows. □

6.6 Delay and cost analysis

The delay of the PPC(n) circuit satisfies the following recurrence:

$$d(\text{PPC}(n)) = \begin{cases} d(\star\text{-gate}) & \text{if } n = 2 \\ d(\text{PPC}(n/2)) + 2 \cdot d(\star\text{-gate}) & \text{otherwise.} \end{cases}$$

If follows that

$$d(\text{PPC}(n)) = (2 \log n - 1) \cdot d(\star\text{-gate}).$$

The cost of the PPC(n) circuit satisfies the following recurrence:

$$c(\text{PPC}(n)) = \begin{cases} c(\star\text{-gate}) & \text{if } n = 2 \\ c(\text{PPC}(n/2)) + (n-1) \cdot c(\star\text{-gate}) & \text{otherwise.} \end{cases}$$

Let $n = 2^k$, it follows that

$$\begin{aligned}c(\text{PPC}(n)) &= \sum_{i=2}^{k} (2^i - 1) \cdot c(\star\text{-gate}) + c(\star\text{-gate}) \\ &= (2n - \log n - 2) \cdot c(\star\text{-gate}).\end{aligned}$$

We conclude with the following corollary.

Corollary 6.6 *If the delay and cost of an $\star$-gate is constant, then*

$$\begin{aligned}d(\text{PPC}(n)) &= \Theta(\log n) \\ c(\text{PPC}(n)) &= \Theta(n).\end{aligned}$$

Question 6.7 *This question deals with the implementation of $\circ$-gates for general finite state machines. (Recall that a $\circ$-gate implements composition of restricted transition functions.)*

1. *Suggest a representation (using bits) for the functions δ_i.*
2. *Design a $\circ$-gate with respect to your representation, namely, explain how to compute composition of functions in this representation.*
3. *What is the size and delay of the $\circ$-circuit with this representation? How does it depend on Q and Σ?*

6.7 The parallel-prefix adder

So far, the description of the parallel-prefix adder has been rather abstract. We started with the state diagram of the serial adder, attached a function δ_i to each input symbol σ_i, and computed the composition of the functions. In essence, this leads to a fast and cheap adder design. In this section we present a concrete design based on this construction. For this design we choose a specific representation of the functions f_0, f_1, f_{id}. This representation appears in Ladner and Fischer [LadnerFischer80], and in many subsequent descriptions (see [BrentKung82, ErceLang04, MüllerPaul00]). To facilitate comparison with these descriptions, we follow the notation of [BrentKung82]. In Fig. 10, a block diagram of this parallel-prefix adder is presented.

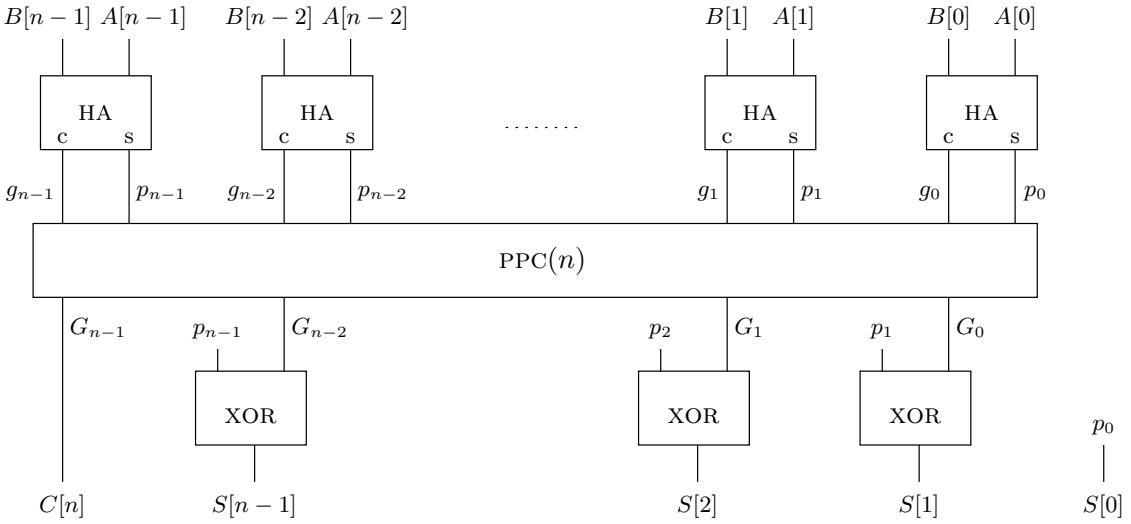

Figure 10: A parallel-prefix adder. (HA denotes a half-adder and the PPC(n) circuit consists of $\circ$-gates described below.)

The carry-generate and carry-propagate signals. We decide to represent the functions $\delta_i \in \{f_0, f_1, f_{id}\}$ by two bits: g_i - the carry-generate signal, and p_i - the carry propagate signal. Recall that the ith input symbol σ_i is the pair of bits $A[i], B[i]$. We simply use the binary representation of $A[i] + B[i]$. The binary representation of the sum $A[i] + B[i]$ requires two bits: one for units and the second for twos. We denote the units bit by p_i and the twos bit by g_i. The computation of p_i and g_i is done by a half-adder that is input $A[i]$ and $B[i]$.

Implementation of the $\circ$-gate. Now that we have selected a representation of the functions f_0, f_1, f_{id}, we need to design the $\circ$-gate that implements composition. This is an easy task: if $(g,p) = (g_1,p_1) \circ (g_2,p_2)$, then $g = g_1$ OR (p_1 AND g_2) and $p = p_1$ AND p_2.

Putting things together. The pairs of signals (g_0,p_0), (g_1,p_1), $\ldots (g_{n-1},p_{n-1})$ are input to a PPC(n) circuit that contains only $\circ$-gates. The output of the PPC(n) is a representation of the functions π_i, for $0 \leq i < n$. We denote the pair of bits used to represent π_i by (G_i,P_i). We do not need the outputs $P_0,\ldots,P_{n-1}$, so we only depict the outputs $G_0,\ldots,G_{n-1}$.

Recall that state q_{i+1} equals $\pi_i(q_0)$. Since $q_0 = 0$, it follows that $\pi_i(q_0) = 1$ if and only if $\pi_i = f_1$, namely, if and only if $G_i = 1$. Hence $q_{i+1} = G_i$.

We are now ready to compute the sum bits: Since $S[0] = \text{XOR}(A[0],B[0])$, we may reuse $p[0]$, and output $S[0] = p[0]$. For $0 < i < n$, $S[i] = \text{XOR}(A[i],B[i],q_i) = \text{XOR}(p_i,G_{i-1})$. Finally, for those interested in the carry-out bit $C[n]$, simply note that $C[n] = q_n = G_{n-1}$.

Question 6.8 *Compute the number of gates of each type in the parallel prefix adder presented in this chapter as a function of n.*

In fact it is possible to save some hardware by using degenerate $\circ$-gates when only the carry-generate bit of the output of a $\circ$-gate is used. This case occurs for the $n/2-1$ $\circ$-gates whose outputs only feed outputs of the PPC(n) circuit. More generally, one could define such degenerate $\circ$-gates recursively, as follows. A $\circ$-gate is degenerate if: (i) Its output feeds only an output of the PPC(n) circuit, or (ii) Its output is connected only to ports that are the right input ports of degenerate $\circ$-gates or an output port of the PPC(n)-circuit.

Question 6.9 *How many degenerate $\circ$-gates are there?*

7 Further topics

In this section we discuss various topics related to adders,

7.1 The carry-in bit

The role of the carry-in bit is somewhat mysterious at this stage. To my knowledge, no programming language contains an instruction that uses a carry-in bit. For example, we write $S := A + B$, and we do not have an extra single bit variable for the carry-in. So why is the carry-in included in the specification of an adder?

There are two justifications for the carry-in bit. The first justification is that one can build adders of numbers of length $k+\ell$ by serially connecting an adder of length k and an adder of length ℓ. The carry-out output of the adder of the lower bits is fed to the carry-in inputs of the adder of the higher bits.

A more important justification for the carry-in bit is that it is needed for a constant time reduction from subtraction of two's complement numbers to binary addition. A definition of the two's complement representation and a proof of the reduction appear in [MüllerPaul00, Even04].

In this essay, we do not define two's complement representation or deal with subtraction. However, most hardware design courses deal with these issues, and hence, they need the carry-in input. The carry-in input creates very little complications, so we do not mind considering it, even if its usefulness is not clear at this point.

Formally, the carry-in bit is part of the input and is denoted by $C[0]$. The goal is to compute $A + B + C[0]$.

There are three main ways to compute $A + B + C[0]$ within the framework of this essay:

1. The carry-in bit $C[0]$ can be viewed as a setting of the initial state q_0. Namely, $q_0 = C[0]$. This change has two effects on the parallel-prefix adder design: (i) The sum bit $S[0]$ equals $\gamma(q_0, \sigma_0)$. Hence $S[0] = \text{XOR}(C[0], p[0])$. (ii) For $i > 0$, the sum bit $S[i]$ equals $\text{XOR}(\pi_{i-1}(q_0), p_i)$. Hence $S[i]$ can be computed from 4 bits: p_i, $C[0]$, and G_{i-1}, P_{i-1}. The disadvantage of this solution is that we need an additional gate for the computation of each sum bit. The advantage of this solution is that it suggests a simple way to design a compound adder (see Sec. 7.2).

2. The carry-in bit can be viewed as two additional bits of inputs, namely $A[-1] = B[-1] = C[0]$, and then $C[0] = 2^{-1} \cdot (A[-1] + B[-1])$. This means that we reduce the task of computing $A + B + C[0]$ to the task of adding numbers that are longer by one bit. The disadvantage of this solution is that the $\text{PPC}(n)$ design is particularly suited for n that is a power of two. Increasing n (that is a power of two) by one incurs a high overhead in cost.

3. Use δ'_0 instead of δ_0, where δ'_0 is defined as follows:

$$\delta'_0 \triangleq \begin{cases} f_0 & \text{if } A[0] + B[0] + C[0] < 2 \\ f_1 & \text{if } A[0] + B[0] + C[0] \geq 2. \end{cases}$$

   This setting avoids the assignment $\delta_0 = f_{id}$, but still satisfies: $\delta'_0(q_0) = q_1$ since $q_0 = C[0]$. Hence, the $\text{PPC}(n)$ circuit outputs the correct functions even when δ_0 is replaced by δ'_0. The sum bit $S[0]$ is computed directly by $\text{XOR}(A[0], B[0], C[0])$.

   Note that the implementation simply replaces the half-adder used to compute p_0 and g_0 by a full-adder that is also input $C[0]$. Hence, the overhead for dealing with the carry-in bit $C[0]$ is constant.

7.2 Compound adder

A compound adder is an adder that computes both $A + B$ and $A + B + 1$. Compound adders are used in floating point units to implement rounding. Interestingly, one does not need two complete adders to implement a compound adder since hardware can be shared. On the other hand, this method does not allow using a $\text{PPC}(n)$ circuit with degenerate $\circ$-gates.

The idea behind sharing is that, in the first method way for computing $A + B + C[0]$, we do not rely on the carry-in $C[0]$ to compute the functions π_i. Only after the functions π_i are computed, the carry-in bit $C[0]$ is used to determine the initial state q_0. The sum bits then satisfy $S[i] = \text{XOR}(\pi_{i-1}(q_0), p_i)$, for $i > 0$. This means that we can share the circuitry that computes π_i for the computation of the sum $A + B$ and the incremented sum $A + B + 1$.

7.3 Fanout

The fanout of a circuit is the maximum fanout of an output port in the circuit. Recall that the fanout of an output port is the number of input ports that are connected to it. In reality, a large fanout slows down the circuit. The main reason for this in CMOS technology is that

each input port has a capacity. An output port has to charge (or discharge) all the capacitors corresponding to the input ports that it feeds. As the capacitance increases linearly with the fanout, the delay associated with stabilizing the output signal also increases linearly with the fanout. So it is desirable to keep the fanout low. (Delay grows even faster if resistance is taken into account.)

Ladner and Fischer provided a complete analysis of the fanout. In the PPC(n) circuit, the only cause for fanout greater than 2 are the branchings after the output of PPC($n/2$). Namely, only nets that feed outputs have a large fanout. Let $fo(i, n)$ denote the fanout of the net that feeds π_i. It follows that $fo(2i, 2n) = 1$ and $fo(2i + 1, 2n) = 1 + fo(i, n)$. Hence, the fanout is logarithmic. Brent and Kung [BrentKung82] reduced the fanout to two but increased the cost to $O(n \log n)$. One could achieve the same fanout while keeping the cost linear (see question below for details); however, the focus in [BrentKung82] was area and a regular layout, not cost.

Question 7.1 *In this question we consider fanout in the* PPC(n) *design and suggest a way to reduce the fanout so that it is at most two.*

- *What is the maximum fanout in the* PPC(n) *design?*
- *Find the output port with the largest fanout in the* PPC(n) *circuit.*
- *The fanout can be made constant if buffers are inserted according to the following recursive rule. Insert a buffer in every branching point of the* PPC(n) *that is fed by an output of the* PPC($n/2$) *design (such branching points are depicted in Fig. 9 by filled circles below the* PPC($n/2$) *circuit). (A buffer is a combinational circuit that implements the identity function. A buffer can be implemented by cascading two inverters.)*
- *By how much does the insertion of buffers increase the cost and delay?*
- *The rule for inserting buffers to reduce the fanout can be further refined to save hardware without increasing the fanout. Can you suggest how?*

7.4 Tradeoffs between cost and delay

Ladner and Fischer [LadnerFischer80] present an additional type of "recursive step" for constructing a parallel prefix circuit. This additional recursive step reduces the delay at the price of increasing the cost and the fanout. For input length n, Ladner and Fischer suggested $\log n$ different circuits. Loosely speaking, in the kth circuit, one performs k recursive steps of the type we have presented and then $\log n - k$ recursive steps of the second type. We refer the reader to [LadnerFischer80] for the details and the analysis.

In the computer arithmetic literature several circuits have been suggested for fast adders. For example, Kogge and Stone suggested a circuit with logarithmic delay but its cost is $O(n \log n)$. Its asymptotic layout area is the same as the area of the layout suggested by Brent and Kung. More details about other variations and hybrids can be found in [Knowles99, Zimmerman98].

7.5 VLSI area

This essay is about hardware designs of adders. Since such adders are implemented as VLSI circuits, it makes sense to consider criteria that are relevant to VLSI circuits. With all other factors remaining the same, area is more important in VLSI than counting the number of gates.

By area one means the area required to draw the circuit on a chip. (Note, however, that more gates usually consume more power.)

More about the model for VLSI area can be found in [Shiloach76, Thompson80]. In fact, the first formal research on the area of hardware circuits was done by Yossi Shiloach under the supervision of my father during the mid 70's in the Weizmann institute. This research was motivated by discussions with the people who were building the second Golem computer using printed circuit boards.

In addition to reducing fanout, Brent and Kung "unrolled" the recursive construction depicted in Fig. 9 and presented a regular layout for adders. The area of this layout is $O(n \log n)$.

8 An opinionated history of adder designs

The search for hardware algorithms for addition with short delay started in the fifties. Ercegovac and Lang [ErceLang04] attribute the earliest reference for an adder with a logarithmic delay to Weinberger and Smith [WeinSmith58]. This adder is often referred to as a *carry-lookahead adder*. On the other hand, Ladner and Fischer cite Ofman [Ofman63] for a carry-lookahead adder. In Cormen et. al., it is said that the circuit of Weinberger and Smith required large gates as opposed to Ofman's design that required constant size gates. So it seems that in modern terms, the delay of the design of Weinberger and Smith was not logarithmic.

I failed to locate these original papers. However, one can find descriptions of carry-lookahead adders in [CLR90, ErceLang04, Koren93, Kornerup97]). The topology of a carry-lookahead adder is a complete tree of degree 4 with $\log_4 n$ levels (any constant is fine, and 4 is the common choice). The data traverses the tree from the leaves to the root and then back to the leaves. The carry-lookahead adder is rather complicated and seems to be very specific to addition. Even though the delay is logarithmic and the cost is linear, this adder was not the "final word" on adders. In fact, due to its complexity, only the first two layers of the carry-lookahead adder are described in detail in most textbooks (see [Koren93, ErceLang04]).

The conditional-sum adder [Sklansky60] is a simple adder with logarithmic delay. However, its cost is $O(n \log n)$. Even worse, it has a linear fanout, so in practice the delay is actually $O(\log^2 n)$. The main justification for the conditional-sum adder is that it is simple.

There was not much progress in the area of adder design till Ladner and Fischer [LadnerFischer80] presented the *parallel-prefix adder*. Their idea was to design small combinational circuits that simulate finite-state machines (i.e., transducers). They reduced this task to a *prefix problem* over an associative operator. They call a circuit that computes the prefix problem a *parallel prefix circuit*. When applied to the trivial bit-serial adder (implemented by a two-state finite state machine), their method yielded a combinational adder with logarithmic delay, linear size, and logarithmic fanout.

In fact, for every n, Ladner and Fischer presented $\log n$ different parallel prefix circuits; these designs are denoted by $\mathcal{P}_k(n)$, for $k = 0, \ldots, \log n$. The description of the circuits uses recursion. There are two types of recursion steps, the combination of which yields $\log n$ different circuits. (For simplicity, we assume throughout that n is a power of 2.) All these designs share a logarithmic delay and a linear cost, however, the constants vary; reduced delay results in increased size. Ladner and Fischer saw a similarity between $\mathcal{P}_{\log n}(n)$ and the carry-lookahead adder. However, they did not formalize this similarity and resorted to the statement that 'we believe that our circuit is essentially the same as the "carry-lookahead" adder'.

Theory of VLSI was at a peak when Ladner and Fischer published their paper. Brent

and Kung [BrentKung82] popularized the parallel-prefix adder by presenting a regular layout for the parallel-prefix adder $\mathcal{P}_{\log n}(n)$. The layout was obtained by "unrolling" the recursion in [LadnerFischer80]. The area of the layout is $O(n \log n)$. In addition, Brent and Kung reduced the fanout of $\mathcal{P}_{\log n}(n)$ to two by inserting buffers. Since they were concerned with area and were not sensitive to cost, they actually suggested increasing the cost to $O(n \log n)$ (see Question 7.1 for guidelines regarding the reduction of the fanout while keeping the cost linear.)

The adder presented in Brent and Kung's paper is specific for addition. The ideas of parallelizing the computation of a finite-state machine and the prefix problem are not mentioned in [BrentKung82]. This meant that when teaching the Brent-Kung adder, one introduces the carry-generate and carry-propagate signals (i.e., g_i, p_i signals) without any motivation (i.e., it is a way to compute the carry bits, but where does this originate from?). Even worse, the associative operator is magically pulled out of the hat. Using simple and short algebraic arguments, it is shown that applying this operator to a prefix problem leads to the computation of the carry bits. Indeed, the proofs are short, but most students find this explanation hard to follow. I suspect that the cause of this difficulty is that no meaning is attached to the associative operator and to the g_i, p_i signals.

The prefix problem gained a lot of success in the area of parallel computation. Leighton [Leighton91] presented a parallel prefix algorithm on a binary tree. The presentation is for a synchronous parallel architecture, and hence the computation proceeds in steps. I think that the simplest way to explain the parallel prefix algorithm on a binary tree is as a simulation of Ladner and Fischer's $\mathcal{P}_{\log n}(n)$ circuit on a binary tree. In this simulation, every node in the binary tree is in charge of simulating two nodes in $\mathcal{P}_{\log n}$ (one before the recursive call and one after the recursive call).

In Cormen et. al. [CLR90], a carry-lookahead adder is presented. This adder is a hybrid design that combines the presentation of Brent and Kung and the presentation of Leighton. The main problem with the presentation in [CLR90] is that the topology is a binary tree and data traverses the tree in both directions (i.e., from the leaves to the root and back to the leaves). Hence, it is not obvious that the design is combinational (although it is). So the presentation in Cormen et. al. introduces an additional cause for confusion. On the other hand, this might be the key to explaining Ofman's adder. Namely, perhaps Ofman's adder can be viewed as a simulation of $\mathcal{P}_{\log n}(n)$ on a binary tree.

9 Discussion

When I wrote this essay, I was surprised to find that in many senses the presentation in [LadnerFischer80] is better than subsequent textbooks on adders (including my own class notes). Perhaps the main reason for not adopting the presentation in [LadnerFischer80] is that Ladner and Fischer presented two recursion steps. The combination of these steps led to $\log n$ designs for parallel prefix circuits with n arguments (and hence for n-bit adders). Only one of these designs has gained much attention (i.e., the circuit they called $\mathcal{P}_{\log n}(n)$). There are probably two reasons for the success of this circuit: fanout and area. Brent and Kung were successful in reducing the fanout and presenting a regular layout only for $\mathcal{P}_{\log n}(n)$. It seems that these multiple circuits confused many who preferred then only to consider the presentation of Brent and Kung. In this essay only one type of recursion is described for constructing only one parallel prefix circuit, namely, the $\mathcal{P}_{\log n}(n)$ circuit.

I can speculate that the arithmetic community was accustomed to the carry-propagate and

carry-generate signals from the carry-lookahead adder. They did not need motivation for it and the explanation of Ladner and Fischer was forgotten in favor of the presentation of Brent and Kung.

Another speculation is that the parallel computation community was interested in describing parallel algorithms on "canonic" graphs such as binary trees and hypercubes (see [Leighton91]). The special graph of Ladner and Fischer did not belong to this family. In hardware design, however, one has usually freedom in selecting the topology of the circuit, so canonic graphs are not important.

Acknowledgments

I thank Oded Goldreich for motivating me to write this essay. Thanks to his continuous encouragement, this essay was finally completed. Helpful remarks were provided by Oded Goldreich, Ami Litman and Peter-Michael Seidel. I am grateful for their positive feedback and constructive suggestions.

References

[BrentKung82] R. P. Brent and H. T. Kung, "Regular Layout for Parallel Adders", IEEE Trans. Comp., Vol. C-31, no. 3, pp. 260-264. 1982. Available online at `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pd/rpb060.pdf`

[CLR90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[ErceLang04] M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, 2004.

[Even79] Shimon Even, *Graph Algorithms*, Computer Science Press, 1979.

[Even04] G. Even, "Lecture Notes in Computer Structure", manuscript, 2004.

[EvenLitman91] Shimon Even and Ami Litman, "A systematic design and explanation of the Atrubin multiplier", *Sequences II, Methods in Communication, Security and Computer Sciences*, R. Capocelli et.al. (Editors), pp. 189-202, Springer-Verlag, 1993.

[EvenLitman94] Shimon Even and Ami Litman. "On the capabilities of systolic systems", *Mathematical Systems Theory*, 27:3-28, 1994.

[HopUll79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to automata theory, languages, and computation*, Addison Wesley, 1979.

[Knowles99] S. Knowles, "A Family of Adders" Proceedings of the 14th IEEE Symposium on Computer Arithmetic, pp 30-34, 1999. (Note that the figures in the published proceedings are wrong.)

[Koren93] Israel Koren, *Computer Arithmetic Algorithms*, Prentice Hall, 1993.

[Kornerup97] Peter Kornerup, "Chapter 2 on Radix Integer Addition", manuscript, 1997.

[LadnerFischer80] R. Ladner and M. Fischer, "Parallel prefix computation", *J. Assoc. Comput. Mach.*, 27, pp. 831–838, 1980.

[Leighton91] F. Thomson Leighton, *Introduction to parallel algorithms and architectures: array, trees, hypercubes*, Morgan Kaufmann, 1991.

[LeiSaxe81] C. E. Leiserson and J. B. Saxe, "Optimizing synchronous systems", *Journal of VLSI and Computer Systems*, 1:41-67, 1983. (Also appeared in Twenty-Second Annual Symposium on Foundations of Computer Science, pp. 23-36, 1981.)

[LeiSaxe91] C. E. Leiserson and J. B. Saxe. "Retiming synchronous circuitry". *Algorithmica*, 6(1)5-35, 1991.

[MüllerPaul00] S. M. Müller and W. J. Paul, *Computer Architecture: complexity and correctness*, Springer, 2000.

[Ofman63] Yu Ofman, "On the algorithmic complexity of discrete functions", *Sov Phys Dokl*, 7, pp. 589-591, 1963.

[Shiloach76] Yossi Shiloach, *Linear and planar arrangements of graphs*, Ph.D. thesis, Dept. of Applied Mathematics, Weizmann Institute, 1976.

[Sklansky60] J. Sklansky, "An evaluation of several two-summand binary adders", *IRE Trans. on Electronic Computers*, EC-9:213-26, 1960.

[Thompson80] Clark David Thompson, *A Complexity Theory For VLSI*, PhD Thesis, Carnegie Mellon University, 1980.

[WeinSmith58] A. Weinberger and J. L. Smith, "A logic for high-speed addition", *Nat. Bur. Stand. Circ.*, 591:3-12, 1958.

[Zimmerman98] R. Zimmermann, *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*, PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, Hartung-Gorre Verlag, 1998. Available online at `http://www.iis.ee.ethz.ch/~zimmi/`