DESIGN OF VLSI CIRCUITS USING RETIMING

GUY EVEN

DESIGN OF VLSI CIRCUITS USING RETIMING

RESEARCH THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF SCIENCE

GUY EVEN

SUBMITTED TO THE SENATE OF THE TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY

Kislev, 5755 Haifa November, 1994

THIS RESEARCH THESIS WAS SUPERVISED BY DR. AMI LITMAN AND PROF. BENNY CHOR UNDER THE AUSPICES OF THE COMPUTER SCIENCE DEPARTMENT

ACKNOWLEDGEMENT

I would like to thank Ami Litman for countless discussions which were crucial for completing this work and Benny Chor for his encouragement and support.

Ilan Spillinger introduced me to the "the world of testing", and arranged a summer invitation from IBM-Watson. The summer in IBM turned out to be a big success, and apart from working with Ilan I had a chance to work with Baruch Schieber and Madhu Sudan. The work done jointly with Ilan appears in Chapter 3, and was published in [8].

My father (Shimon Even) and Ami Litman shared their knowledge about systolic arrays and retiming with me. This subject fascinated me so much, because it involved using formal techniques (mainly graph definitions and algorithms) for designing and explaining hardware circuits. The numerous conversations I had with them helped me crystallize my thoughts and determine what subjects I wanted to research. Their support was invaluable, and I thank them for it. The work done jointly with Ami Litman appears in Chapter 5.

Chapter 5 evolved from a question raised by Mike Paterson. The discussion with Mike Paterson spawned the research, and I thank him for it.

A partial list of people who helped, listened, and supported me includes: Ran Canetti, Oded Goldreich, Amir Herzberg, Oded Lempel, Seffi Naor, Erez Petrank, Baruch Schieber, Hadas Shachnai, and Madhu Sudan. I thank them for their assistance.

Last, but definitely not least, I would like to thank Daria. She was always available when I needed her support, shared my discourages, and was happy with my successes.

THE GENEROUS FINANCIAL HELP OF THE MIRIAM AND AARON GUTWIRTH MEMORIAL FELLOWSHIP IS GRATEFULLY ACKNOWLEDGED

Contents

Abstract

This work deals with issues related to the design of VLSI circuits. An emphasis is given to the design of systolic algorithms which are easily realized by VLSI technology. Systolic algorithms satisfy strict design rules which enable simple implementation and analysis. Moreover, expansion and modifications of systolic circuits is natural, and these properties are essential since VLSI design are continuously and rapidly revised. However, the design of systolic circuits is very difficult. Transformations, such as retiming, have been proposed for facilitating the design of systolic algorithms and for improving the performance of synchronous circuits. The use of transformations sets a systematic methodology for designing and evaluating circuits. According to this methodology, "inferior" circuits are improved by transforming them into faster circuits, for example, non-systolic circuits are mapped to systolic circuits. A fundamental requirement of such transformations is that they preserve functionality. Namely, the correctness of the circuit obtained by a transformation should be inferred from the correctness of the original circuit (before the transformation). Non-systolic circuits, which do not adhere to the strict design rules of systolic circuits, are easier to design, and their correctness is simpler to prove. Hence, the transformation of non-systolic circuits into systolic circuits establishes a design methodology of systolic algorithms.

This thesis consists of four parts. In the first part, we discuss the foundational basis of retiming. We provide a new proof of the Retiming Lemma which states that, under certain restrictions, retiming preserves functionality. Previous proofs of the Retiming Lemma are very complicated and lack intuition. Our proof is based on space-time transformations, and, hopefully, succeeds in giving intuition which has been lacking to-date. We also provide two applications of the retiming lemma which deal with initial states of circuits and with the effect of retiming on the detection capabilities of test sequences.

In the second part, we deal with the problem of finding an initialization of retimed circuit so that the retimed circuit is equivalent to the original circuit. The Retiming Lemma does not resolve this issue, since it only provides a characterization of initializations of the original circuit which have equivalent initializations in the retimed circuit. However, finding the equivalent initialization in the retimed task is PSPACE-hard. We seek retiming functions which yield fast clock rates, and for which facilitate the problem of finding equivalent initializations after retiming. We define two heuristic cost functions which estimate the complexity of finding initializations in the retimed circuit. Efficient algorithms are presented

which find retimings which minimize the value of these heuristic cost functions, and which induce circuits capable of operating with short clock periods.

In the third part, we demonstrate the benefits of using design methodologies, such as retiming, by presenting two new real-time, bit serial, systolic integer multipliers. One multiplier is pipelined and the other is not. Our non-pipelined multiplier requires roughly half the hardware as compared to Atrubin's multiplier, which is the best bit-serial, real-time, systolic integer multiplier known to-date. We start with a well known multiplier, called a serial/parallel multiplier. Its drawbacks are eliminated by a sequence of transformations. We use retiming and two new transformations: replacing the parallel inputs with broadcast and a method for pipelining. Our systematic approach enables the design of an improved circuit, the correctness of which is easily understood.

In the fourth part, we present a method for overcoming chip-to-chip delays and clock skews. We consider a setting in which one wants to realize a systolic array (either a linear array or a two-dimensional array) which is too large to fit on a single chip. Chip-to-chip interconnections incur a physical setting which suffers from long delays and large clock skews. We consider a setting in which the delay and clock skew may equal several clock periods. Our solution enables operation with clock periods which are very close to the clock periods of the systolic array when it fits on a single chip. The solution consists of three stages: First, we present an implementation of an interconnection which provides a "logical" interconnection similar to a shift register with many stages. This implementation functions properly even in the presence of long delays and large clock skews. Second, we duplicate logic in the vicinity of the chip-to-chip interconnections. Third, we retime the duplicated logic so that two objectives a re met: (a) The chip-to-chip interconnections have enough registers along them so that the implementation proposed in the first stage is applicable. The number of registers required along the interconnection between chips is determined by the delay and clock skew. (b) The retimed array can operate with a clock period which almost equals the feasible clock period which is obtainable when a small version of the array is mapped onto a single chip.

All four parts deal with retiming, and describe different aspects of this transformation, such as: properties (preserving functionality), algorithms which find good retimings, and implementations for systolic algorithms. This broad treatment can be helpful in clarifying the power and potential benefits of retiming.

List of terms and symbols

List of Figures

List of Tables

Chapter 1

Introduction

This chapter is organized as follows: In Section 1.1 we describe the background relevant to this work. The works of Leiserson and Saxe and the works of S. Even and Litman are overviewed. In Section 1.2 basic definitions and concepts used throughout the work are presented. Section 1.3 overviews chapters 2-5.

1.1 Background

The ability to produce inexpensive special-purpose complex VLSI chips raises the question of to how to design VLSI algorithms which utilize the capabilities this technology. VLSI circuits have short life cycles, and designs are updated and expanded even before they are produced. Ad-hoc designs are very difficult to expand, and hence, modifications to such designs have the side effect of deteriorating circuit performance. Another crucial issue is proving that a design is correct. It is unfortunate that many suggestions for improving circuit performance also ruin the correctness of the design. The need for a general methodology which enables systematic circuit design and evaluation, and which enables automation of circuit design has been recognized by many [18, 19, 21, 23, 27].

This work focuses on two related suggestions which provide partial solutions to the problem of designing VLSI circuits. One suggestion is to use systolic algorithms [18]. Systolic algorithms are special purpose designs which are easily realized as VLSI circuits. Moreover, systolic designs have the following advantages: Fast clocks - the circuits can operate with very fast clock rates. Modularity - the circuits are constructed from a small set of cells. Scalability - the designs are easily modified for longer input lengths. Small number of primary inputs/outputs - facilitates scalability and routing. Simple layouts - facilitates designing the VLSI realization of the circuit. Systolic algorithms have been studied extensively [18, 21, 19], and have been proven to be extremely useful in various signal processing applications and matrix manipulation problems.

However, systolic algorithms have the drawback of being very complicated, and the correctness of systolic algorithms is difficult to understand. Two drastic examples of this phenomenon are circuits for palindrome recognition and integer multipliers. Cole [4] and Sieferas [28]) presented intricate circuits capable of recognizing strings which are palindromes. Atrubin [1] presented an ingenious systolic, bit-serial, real-time integer multiplier but was never able to prove its correctness.

Leiserson and Saxe [19, 21, 27, 23] recognized that systolic algorithms are desirable one one hand, but very difficult to design on the other hand. They suggested using transformations, mainly retiming, for solving this problem. The transformation maps a non-systolic algorithm, which is relatively easy to design, into a systolic one, and guarantees that functionality is kept intact. Hence, if the non-systolic algorithm is correct, then so is the systolic algorithm obtained by the transformation. In addition, they showed that these transformations could be utilized for improving the performance of non systolic circuits, without necessarily aiming for systolic algorithms.

The main components of Leiserson and Saxe work are:

1. An abstraction of synchronous circuits. This abstraction is called a communication graph, and is crucial for defining the transformations and the optimization goals.
2. A definition of a transformation called retiming which enables improving the design of synchronous circuits.
3. A proof that retiming preserves functionality (under certain restrictions).
4. Examples which demonstrate the power of retiming; they presented a simple systolic palindrome recognizer.
5. A general method of transforming synchronous circuits into systolic circuits.
6. Algorithms which find retimings which optimize the performance of synchronous circuits. The two main examples are finding a retiming which minimizes the clock period, and finding a retiming which minimizes the number of registers in a circuit.

The work of Leiserson and Saxe was followed by the work of S. Even and Litman [12, 13]. Even and Litman [13] introduced a better formalism of the extent to which retiming preserves functionality, discussed the problem of circuit initialization in the context of retiming, presented a new utilization of retiming called the tiling technique, presented a new technique called the bypassing technique, and introduced some new algorithms for making a circuit systolic and for utilizing these techniques. Even and Litman [12] also demonstrated that the tiling technique and retiming can be used to prove the correctness of Atrubin's multiplier. First, they suggest a non-systolic version, the correctness of which is easy to prove, and then they transform it into a systolic circuit.

1.2 Preliminaries

In this section we present the basic definitions and concepts which are used throughout this work. The material presented in this chapter is based on the works of Leiserson et. al. [19, 21, 20, 22, 23] and on the works of S. Even and Litman [12, 13].

1.2.1 The communication graph model

Synchronous circuits are modeled by *communication graphs.* A communication graph, $Y = \langle V, E, w, d \rangle$, consists of a directed graph (V, E) with non-negative integer weights, $w(e)$, on the edges and non-negative real delays, $d(v)$, on the vertices. The vertices model combinational functional units. The edges model interconnects between functional units, and therefore, only point to point interconnections are allowed between components in the circuits (e.g. buses are not allowed). The weight of an edge models the number of registers along the corresponding interconnect. Each register has exactly one input and one output; at the end of each clock cycle, a register samples its input, and the sampled value is output during the next clock cycle. Therefore, a signal sent from vertex u along a positive weighted edge $u \to v$ during the i'th clock cycle arrives at vertex v in the beginning of clock cycle $i + w(u \to v)$. The delays of vertices model the propagation delays of the functional units in the vertex. A special vertex, called the *host*, is designated for modeling the external world which feeds the inputs and reads the outputs. We usually coalesce all the input and output ports into the host; every output enters the host, and all inputs emanate from the host. The delay of the host is defined to be zero. The weight of a path, $p = v_0 \stackrel{e_0}{\to} v_1 \stackrel{e_1}{\to} \cdots \stackrel{e_{k-1}}{\to} v_k$, is defined as $w(p) \triangleq \sum_{i=0}^{k-1} w(e_i)$. The delay of a path is defined as $d(p) \triangleq \sum_{i=0}^{k} d(v_i)$ (in section 3.6.1 we refine this definition). (See Example 3.7 on page 58). Henceforth, reference to a synchronous circuit means the communication graph which models this circuit.

In a communication graph, every cycle has at least one edge with a positive weight. Without this restriction, values of the signals carried along such zero-weight cycles might not be well defined, due to instability problems. Moreover, this condition is sufficient for uniquely defining the computation performed by a synchronous circuit: First, note that removing the edges containing registers partitions the circuit into disjoint acyclic combinational components. The inputs of each such combinational component are given by registers and by the host. The outputs of each combinational component enter the host or the registers. The computation in a synchronous circuit is defined as follows: During every clock cycle, the registers output the values they sampled and stored at the end of the previous clock cycle, and the input hosts send values according to the input sequence. Therefore, the inputs of each combinational component are well defined. The outputs of each combinational component are computed according to the input values, and the output values either enter the output host or a register. Values entering registers are sampled at the end of the clock cycle, and stored for output during the next clock cycle.

One of the advantages of using this graph model is the possibility of estimating the minimum clock period a given circuit can operate with. In each clock cycle, the signals propagate through registerless paths, and should be stable by the end of the clock cycle. If the delay of a vertex is an upper bound on the propagation delay of the functional unit which corresponds to the vertex, then the maximum delay of a zero weight path constitutes an upper bound on the minimum clock period. Hence, the minimum feasible clock period of G is defined as: $\Phi(G) \triangleq \max\{d(p) : w(p) = 0\}$.

1.2.2 Systolic circuits

Systolic circuits are are special purpose circuits which are especially suited for realization by VLSI technology. Consider the problem of multiplying n-bit integers. We would like to design a small "library" of cells, from which multipliers for every multiplicands' lengths can be constructed. We would like all wires connecting cells to have at least one register along them. Moreover, we would like all wires interconnecting neighboring cells to have a constant length (which is independent of n). This guarantees that the clock period of the multiplier is independent of the operands' length, and depends only on the propagation delay of a single cell. Such a design can help simplify the task of realizing the multiplier on a chip, since problems such as placement, routing, and circuit analysis are simplified. To summarize, the following design goals are sought and obtained in systolic designs:

Scalability A systolic design constitutes a "family" of circuits. We define a parameter called the circuit's size, or the inputs' length. This parameter refers to the number of vertices (also called cells) in the communication graph, or to the length of the inputs. For example, a systolic multiplier may be constructed from a single type of cell. Multiplication of n-bit integers requires a graph which contains $n/4$ cells.

A systolic design easily extends to every input length. Moreover, the clock rate does not increase with expansion of the circuit, and circuit area grows linearly with the length of the inputs.

Fast clocks Systolic circuits can operate with very fast clock rates. The clock rate depends only on the complexity of a single cell (i.e. a vertex in the communication graph). Hence, increasing the number of bits of the multiplicands does not slow down the clock rate of the multiplier, and does not effect other circuits in the system.

Modularity Systolic circuits are constructed from a small variety of types of cells. Hence, one can optimize and analyze these few cells thoroughly, and the complexity of these tasks does not depend on the size of the circuit.

Simple layouts Apart from having compact mappings onto chips, the interconnections between neighboring cells have constant lengths (which are independent of the length of the inputs). This is obtained by using communication graphs such as linear arrays and two-dimensional arrays.

We define a circuit to be systolic if all weights in its communication graph have positive weight. This definition is useful provided that the circuit is partitioned into vertices so that the complexity and propagation delays of the vertices in the communication graph is small. Synchronous circuits which have zero weighted edges are also called semi-systolic.

1.2.3 Retiming

The design of systolic arrays is complicated and the correctness of these circuits is difficult to understand. For example, Cole [4] and Sieferas [28] studied and designed systolic circuits which recognize palindromes. However, these designs are very complicated to understand, and proving their correctness is intricate. Leiserson and Saxe [21] suggested retiming as a design methodology for simplifying and improving the design of systolic arrays and synchronous circuits. They exemplified the power of retiming by presenting a simple and easy to understand systolic palindrome recognizer.

The design methodology suggested by Leiserson and Saxe is to first design a semi-systolic circuit which solves the problem. The semi-systolic circuit should be easy to understand, and the proof of its correctness should be easy as well. The systolic circuit is obtained by transforming the semi-systolic circuit into a systolic circuit using retiming. The transformation (retiming) keeps functionality intact, and hence, a systolic circuit is obtained. (Chapter 2 is devoted to proving that retiming keeps functionality intact).

Retiming is a transformation which maps a synchronous circuit into a synchronous circuit having the same topology, but with different weights on the edges (in other words, retiming can only add or delete registers from existing edges). Retiming is carried out by assigning each vertex a lag. This lag specifies a number of clock cycles the vertex is "shifted in the time axis". From the point of view of a single vertex, retiming amounts to performing the same computation either earlier or later. A vertex with a positive lag performs the computations later than it does in the original circuit (and is said to *advanced*), and a vertex with a negative lag performs the computations earlier than it originally does (and is said to *lagged*).

Retiming is defined using the communication graph abstraction as follows: Let $\langle V, E, w\rangle$ be a synchronous circuit. A retiming is specified by an integer valued function $lag : V \to \mathsf{Z}$, where $lag(v)$ denotes the lag of vertex v. When a vertex is lagged by k, then we remove k registers from each edge emanating it, and add k registers to each edge entering it. This implies that after performing the retiming, an edge $u \to v$ will have $w(u \to v) - lag(u) + lag(v)$ registers along it. Since the number of registers along an edge must be non-negative, we require that the function $lag()$ satisfy the following equation:

$$\forall u \to v \; : \quad w(u \to v) \geq lag(u) - lag(v) \tag{1.1}$$

A function which satisfies equation 1.1 is called a *retiming function.*

The weights of the retimed circuit, $\langle V, E, w'\rangle$, resulting from retiming the circuit $\langle V, E, w\rangle$ according to the retiming function $lag()$ are defined by:

$$w'(u \to v) \triangleq w(u \to v) - lag(u) + lag(v)$$

The proof of the following claim is straightforward.

Claim 1.1: [23] Cycle weights are not modified by retiming. Namely, if c is a directed cycle in (V, E) then $w(c) = w'(c)$.

1.2.4 Eliminating broadcast in linear arrays

One of the most useful applications of retiming is for eliminating broadcast from a semi-systolic circuit. Broadcast is a mechanism by which the host sends a signal every clock cycle which reaches all vertices in the same clock cycle. This mechanism is very powerful, but, in practice, incurs long delays, and therefore, slows down the feasible clock rates. We demonstrate the elimination of broadcast using retiming on linear arrays.

Consider a semi-systolic circuit which is systolic apart from having a broadcast mechanism. Two techniques have been suggested for eliminating the broadcast and obtaining a functionally equivalent circuit which is fully systolic:

2-slowdown

Leiserson and Saxe [21] suggested eliminating the broadcast in two stages:

1. Multiply the number of registers along each edge by 2. This results in a new circuit which is a 2-slowdown of the original circuit. This means that if one feeds the inputs and observes the outputs only during the even clock cycles, then the circuit behaves identically to the original circuit. The new circuit can be utilized to perform two different (unrelated) computations during the even clock cycles and during the odd clock cycles, and therefore, this transformation does not reduce the throughput of the circuit, although latency is increased by a factor of 2.

2. Retime the circuit so that every edge has positive weight.

Note, that this technique does not preserve functionality. The new circuit is a 2-slowdown of the original one.

This technique is demonstrated in the following example. Consider the linear array with broadcast depicted in figure 1.1. Registers are depicted by short segments perpendicular to the edges. Self loops attached to the vertices serve as the memory of the vertex. The broadcast network is implemented by registerless edges going along the linear array. First, we double the number of registers along each edge. The retiming of this array which obtains a systolic array is depicted in figure 1.2. The lag of each vertex is written inside the vertex. After retiming, all edges have positive weight, and we obtain a systolic circuit.

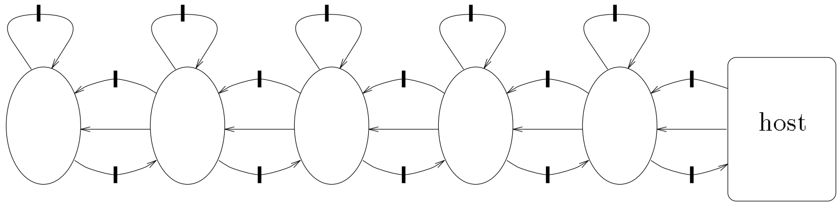


Figure 1.1: Linear Array with broadcast
איור 1.1: מערך ליניארי עם כריזה.

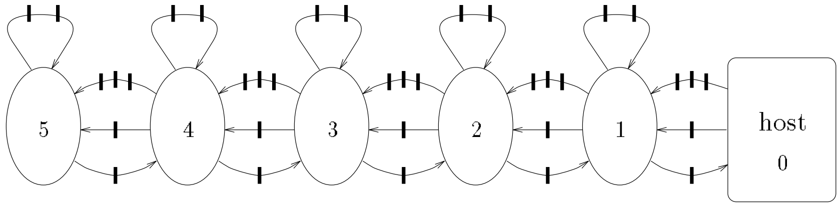


Figure 1.2: 2-slowed down linear array after retiming.
איור 1.2: 2-האטה של מערך ליניארי לאחר רה-תזמון.

The tiling technique

Even and Litman [12, 13] suggested a different technique for eliminating broadcast, called the tiling technique. First, they observed that modeling the broadcast by only adding edges to the communication graph sets an unnecessary limitation. The source of this limitation is that retiming does not modify the weight of cycles (Claim 1.1). Consider an edge $u \to v$ which has weight 1, and suppose we add a zero weighted edge $v \to u$ for the broadcast network. Since the weight of the cycle $v \to u \to v$ equals 1, retiming can not change both edge weights to positive weights, and only increasing the number of registers can solve this limitation. Therefore, Even and Litman suggested separating the broadcast network from the graph. Instead of multiplying the number of registers, they suggested that one contract small groups of vertices into a single vertex (called a *super-cell*), and only require that edges connecting these super-cells have positive weight. Edges and vertices within each super-cell are regarded as an implementation of the functional unit, and no requirements are set on edge weights within the "super-cells". Subsequently, their goal is to find small as possible "super-cells" for which there exists a retiming which causes all edges interconnecting "super-cells" to have positive weights. The partitioning of the graph into super-cells is called tiling by Even and Litman.

We demonstrate the tiling technique on the linear array with broadcast depicted in figure 1.3. The broadcast network is separated from the linear array, and is depicted by the bottom row of vertices which simply split the signal into two; one part goes to the vertex above, and the second part goes to the rest of the linear array. Figure 1.4 depicts the linear array after retiming using the tiling technique. The tiles (super-cells) are drawn as dotted boxes. The lags of the vertices are written inside each vertex. After retiming, all edges interconnecting super-cells have positive weight.

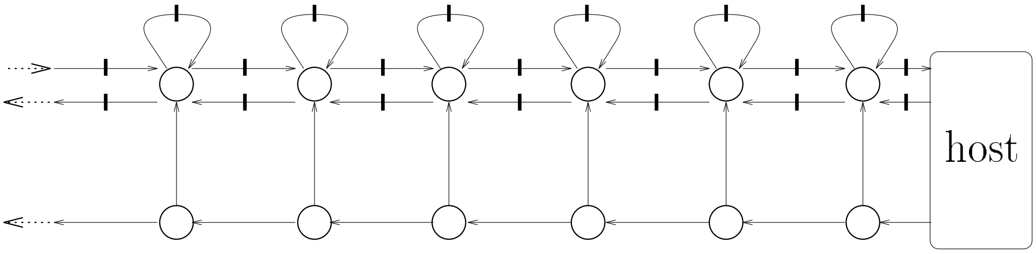


Figure 1.3: Linear array with separate broadcast network.
איור 1.3: מערך ליניארי עם רשת כריזה מופרדת.

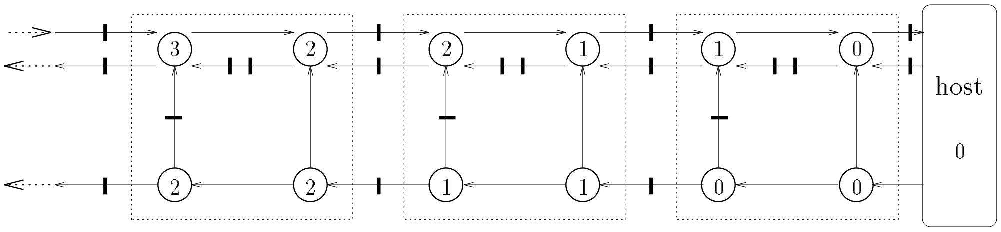


Figure 1.4: Applying the tiling technique to the linear array.
איור 1.4: יישום טכניקת הריצוף על מערך ליניארי.

Note, that the tiling technique keeps functionality intact. However, the minimum feasible clock period is greater that the delay of a cell due to registerless paths within the super-cells. In the case of the linear array, clock period equals at most the sum of delays of two neighboring vertices, because the longest registerless path passes through two vertices.

1.3 Overview

This work deals with issues related to the design of VLSI circuits using retiming. In this section we give a brief overview of chapters 2-5.

In Chapter 2 we discuss the foundational basis of retiming. We provide a new proof of the retiming lemma which states that, under certain restrictions, retiming preserves functionality. The exact extent to which retiming preserves functionality was given by Even and Litman [13] without proof. Our proof is based on space-time transformations, and, hopefully, succeeds in giving intuition which has been lacking to-date. We also provide two applications of the retiming lemma which extend the discussion of Even and Litman [13] about initializations in light of retiming. These applications deal with instantly initializable circuits and with test patterns. Instantly initializable circuits have a special reset signal which is asserted in concert with new computations. This reset signal has the property that it causes the circuit to behave as if it is initialized to a predefined state. We show that a special class of retimings, called non-negative retimings, preserves the property of being instantly initializable and keeps functionality intact. We also show that the detection capabilities of test sequences are preserved by non-negative retimings.

In Chapter 3 we deal with the problem of finding an initialization of retimed circuit so that the retimed circuit is equivalent to the original circuit. The retiming lemma does not resolve this issue, since it only provides a characterization of initializations of the original circuit which have equivalent initializations in the retimed circuit. However, finding the equivalent initialization in the retimed task is generally very hard (PSPACE-hard to be exact). We seek retiming functions which attain identical benefits (namely, fast clock rates), and for which the problem of finding equivalent initializations after retiming is as easy as possible. Our approach consists of three parts: First, we consider a restricted problem, called the problem of finding corresponding initializations, which is only NP-hard. Second, we define two heuristic cost functions which estimate the complexity of finding a corresponding initialization. Third, we present efficient algorithms which find retiming functions which minimize these cost functions and induce circuits capable of operating with short clock periods.

In Chapter 4 two new real-time, bit serial, systolic integer multiplier are introduced. One multiplier is pipelined and the other is not. Our non-pipelined multiplier requires roughly half the hardware as compared to Atrubin's multiplier [1]. Our construction exemplifies that using design methodologies, such as retiming, leads to the design of better circuits the correctness of which is easily understood. We start with a well known multiplier, called a serial/parallel multiplier. Its drawbacks are eliminated by retiming and by replacing the parallel inputs with broadcast. This broadcast is eliminated using retiming again. We proceed by presenting a method for pipelining this multiplier. This method is applicable in situations where a circuit performs a simplified operation during a significant portion of the computation, and is applicable to Atrubin's multiplier as well.

In Chapter 5 we present a method for overcoming chip-to-chip delays and clock skews.

We consider a setting in which one wants to realize a systolic array (either a linear array or a two-dimensional array) which is too large to fit on a single chip. Chip-to-chip interconnections incur a physical setting which suffers from long delays and large clock skews. We consider a setting in which the delay and clock skew may equal several clock periods. Our solution enables operation with clock periods which are very close to the clock periods of the systolic array when it fits a single chip. The solution consists of three stages: First, we present an implementation of an interconnection which provides a "logical" interconnection similar to a shift register with many stages. This implementation functions properly even in the presence of long delays and large clock skews. Second, we duplicate logic in the vicinity of the chip-to-chip interconnections. Third, we retime the duplicated logic so that two objectives are met: (a) The chip-to-chip interconnections have enough registers along them so that the implementation proposed in the first stage is applicable. The number of registers required along the interconnection between chips is determined by the delay and clock skew. (b) The retimed array can operate with a clock period which almost equals the feasible clock period which is obtainable when a small version of the array is mapped onto a single chip.

Chapter 2

The Retiming Lemma

2.1 Introduction

Leiserson and Saxe utilized retiming for improving the performance of synchronous circuits. In particular, they used retiming for transforming synchronous circuits into systolic circuits. They state the Retiming Lemma, which claims that under “appropriate initializations”, retiming does not change the *functionality* of the circuit. The main difficulty in the proof is to show, that every initial state of the original circuit, which is “sufficiently old”, has a corresponding initial state in the retimed circuit, so that the input/output patterns of both (initialized) circuits are identical. This correspondence should not depend on the inputs which are fed after the initialization. The proof of Leiserson and Saxe is “not enlightening”, and the exact quantification of the extent to which retiming preserves functionality is vague. An exact definition and quantification is given by Even and Litman [13] without proof. Litman [24] proves the Retiming Lemma using predicate calculus. Neither proof gives one an intuition of the exact characterization of “appropriate initializations”. The paper of German and Wang [15] and the paper of Brookes [3] prove the correctness of specific retimings of specific circuits which were given as examples by Leiserson and Saxe to demonstrate the benefits of retiming (in fact, for these specific retimings, all initializations are “appropriate”). The book of Megson [25] misrepresents the Retiming Lemma, and mixes issues such as slow-down and algorithms which find retiming functions. The proof presented by Megson ignores the issue of functional equivalence and fails to mention that retiming *may* effect functionality. We believe that the proof of the general case presented in this chapter succeeds in giving intuition and in explaining the extent to which functionality is preserved after retiming.

The proof presented here is based on space-time transformations. Space-time transformations of synchronous circuits are a convenient tool for analyzing the behavior of such circuits [16]. For example, Vergis and Steiglitz [32] utilized space-time transformations for testing bidirectional systolic arrays. The main advantage of a space-time diagram representation of a synchronous circuit is that it is combinational (i.e. it is acyclic and does not contain registers). This representation facilitates the analysis of the communication and the relative delays of signals in the circuit. We believe that analyzing synchronous circuits according to the methods of this chapter can help establish a better understanding of communication issues in synchronous circuits, and in particular provide a model which describes "what goes on" in the retiming lemma.

We also describe two applications of the Retiming Lemma. These applications demonstrate the advantages of using non-negative retiming functions (i.e. no vertices are advanced). The first application examines the effect of retiming on the functionality of circuits designed to function properly regardless of their initialization. We call such circuits *instantly-initializable*. In instantly-initializable circuits, beginnings of computations are marked by a special input (called a *reset* signal) which is fed by a designated input host, and consequently causes the circuit to function as if it is initialized to a fixed state. The beginning of the computation is marked by inputting a "1" through the reset host during the first clock cycle of the computation. We show that applying a non-negative retiming on an instantly-initializable circuit results in an instantly-initializable circuit. Moreover, the original circuit and the retimed one are equivalent with respect to input patterns which "mark" the beginnings of computations. This application was used implicitly in [21, 12, 11] when retiming was utilized for eliminating broadcast.

The second application deals with the effect of non-negative retimings to the detection capabilities of functional test patterns. A functional test pattern is an input pattern which when given to the circuit can verify that the circuit was in a correct initial state and that each of the vertices is functioning correctly. We show that non-negative retiming does not make testing harder. In particular, a test pattern for a synchronous circuit is also a test pattern for a non-negative retiming of it. The motivation for this question is that finding test patterns for circuits before they are retimed (e.g. for semi-systolic circuits) seems to be easier than finding test patterns after retiming.

This chapter is organized as follows: In Section 2.2 we define space-time transformations and space-time circuits. In Section 2.3 we define inputs, outputs, states, initializations and computations of synchronous circuits and space-time circuits. In Section 2.4 we define retiming on space-time circuits. In Section 2.5 we prove the Retiming Lemma. In Section 2.6 we describe two applications of the Retiming Lemma.

2.2 Space-Time Transformations

We define space-time circuits and space-time transformations which transform a synchronous circuit into a space-time circuit. We also show that this transformation can be inverted. We begin by reviewing the graph model used for modeling synchronous circuits.

A synchronous circuit is modeled by a directed graph with non-negative integer weights on the edges, and is denoted by a 4-tuple $Y = \langle V, E, w, F \rangle$. The vertices of the circuit are denoted by V, the edges by E and the weight function by $w : E \to \mathsf{N}$, where N denotes the non-negative integers. For the sake of simplicity, we model a circuit as a graph instead of a multi-graph. One can easily extend the discussion to multi-graphs. The functionality of the circuit is defined by the mapping F, and will be described shortly. (We omit the mapping F when we are only interested in the topology of the circuit). Vertices having only incoming edges are called *output hosts*. Vertices having only outgoing edges are called *input hosts*. We denote the set of input hosts by V_{in} and the set of output hosts by V_{out}. The rest of the vertices correspond to functional elements which compute functions and send the results along the edges emanating them. Each vertex may compute a different function for every edge emanating from it. The functions computed by vertices, which are not hosts, are specified by the mapping F. Namely, the function computed at vertex u, the result of which is sent along the edge $u \to v$, is denoted by $F_{u \to v}$. The weight of an edge is the number of unit registers along it. The set of registers of the circuit Y is denoted by $Regs(Y)$.

We do not restrict the values carried by the edges to binary values. In practice, values carried by edges are encoded in binary, and each wire carries a single bit. Let Σ denote the alphabet of possible values carried by the edges. Retiming does not change the topology of the circuit, nor does it change the set of values carried by each edge. Therefore, there is no risk of changing "edge-capacities" when the same alphabet is used for all edges, and we do so only for simplifying the discussion.

The next definition defines space-time circuits. These are infinite combinational circuits having an iterative property. These circuits are useful for analyzing synchronous circuits.

Definition 2.1 : A *space-time circuit* is a 3-tuple $\langle TV, TE, TF\rangle$, where $\langle TV, TE\rangle$ is a directed graph, and TF is a mapping which assigns a function to each edge. A space-time circuit satisfies the following conditions:

C1: The vertex set is a Cartesian product of a set V and the integers, in other words, $TV = V \times \mathbb{Z}$.

C2: (no "backward edges") If $((u,i) \to (v,j)) \in TE$ then $i \leq j$.

C3: The graph $\langle TV, TE\rangle$ is acyclic.

C4: (repetitive structure and functionality) If $(u,i) \to (v,j) \in TE$ then $(u,i+k) \to (v,j+k) \in TE$ for every $k \in \mathbb{Z}$. Moreover, for every edge $(u,i) \to (v,j) \in TE$, and every integer k, the mapping TF assigns identical functions to the edges $(u,i) \to (v,j)$ and $(u,i+k) \to (v,j+k)$.

The input and output hosts of space-time circuits are defined in the same fashion as they are in synchronous circuits, and are denoted by TV_{in} and TV_{out}, respectively.

The next definition defines space-time transformations. This transformation maps synchronous circuits into space-time circuits, and thus trades time with space.

Definition 2.2 : The space-time transformation maps a synchronous circuit $Y = \langle V, E, w, F\rangle$ into the space-time circuit $TY = \langle TV, TE, TF\rangle$ defined as follows:

1. $TV \triangleq V \times \mathsf{Z}$
2. $TE \triangleq \{(u,i) \rightarrow (v,j) : u \rightarrow v \in E, w(u \rightarrow v) = j - i\}$
3. $TF((u,i) \rightarrow (v,j)) \triangleq F(u \rightarrow v)$

Note, that the space-time circuit contains an infinite number of copies of each vertex of the synchronous circuit. Each copy has the same input ports, output ports and functionality as the vertex of the synchronous circuit.

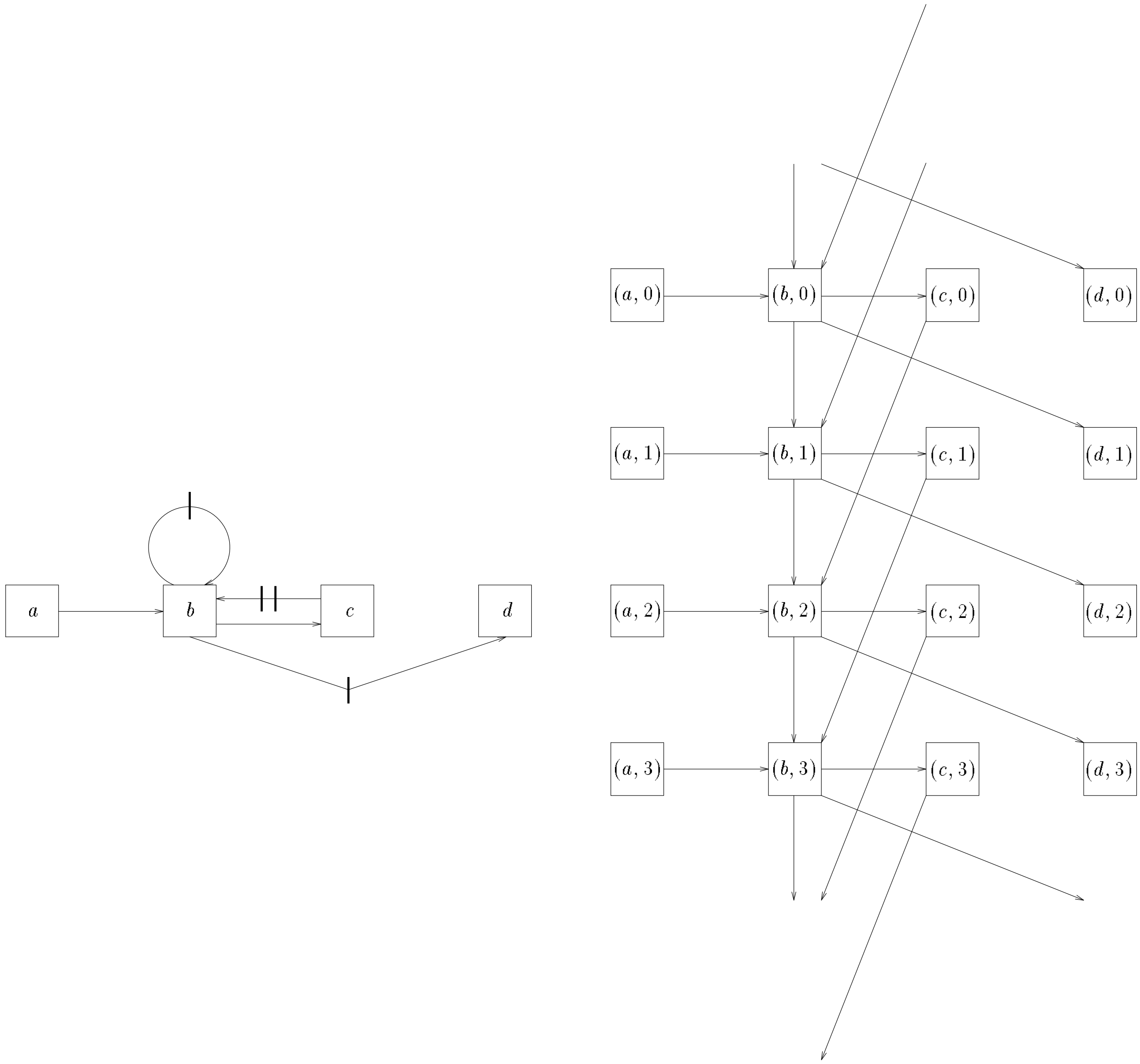
Figure 2.1 depicts a synchronous circuit and its space-time transformation. We define the vertex set of the space-time transformation to be $V \times \mathsf{Z}$ rather than $V \times \mathsf{N}$ (where N denotes the natural numbers) because we want to model this circuit as a graph. Cutting the vertex set on one side causes the edges which emanate the discarded part and enter the remaining part to become "rays" (like the edges entering the top vertices in part (B) of figure 2.1).

The above space-time transformation also has an inverse. Given a space-time circuit $\langle TV, TE, TF\rangle$, We construct the corresponding synchronous circuit $\langle V, E, w, F\rangle$ as following:

1. V is the set such that $TV = V \times \mathsf{Z}$.
2. $E = \{u \rightarrow v : \exists i, j \text{ such that } ((u,i) \rightarrow (v,j)) \in TE\}$.
3. Let $(u,i) \rightarrow (v,j) \in TE$ denote the edge which the edge $u \rightarrow v$ originates from. Then the weight of the edge $u \rightarrow v$ is $j - i$, and the function $F(u \rightarrow v)$ is equal to the function $TF((u,i) \rightarrow (v,j))$. Note that condition C4 guarantees that the weight of edges are well defined, and that the mapping F is well defined.

2.3 Inputs, Outputs, States, Initializations and Computations

Synchronous systems are usually described as Mealy machines. In the context of retiming, it is not convenient to model a synchronous circuit as a Mealy machine, since this model ignores the implementation (a circuit) and focuses only on the functionality. However, retiming is a transformation which modifies the circuit and is used to improve the performance of a circuit without changing its functionality. Since our goal is to show that retiming hardly changes the functionality, we need to define the terms related to functionality such as: inputs, outputs, initialization, states and computations in the circuits which we use (either in realizations or for analysis purposes). In this section we define these terms both for synchronous circuits and for space-time circuits.



(A) (B)

Figure 2.1: (A) a synchronous circuit (B) a segment of its space-time transformation

איור 2.1: (א) מעגל סינכרוני (ב) קטע של מעגל מקום-זמן המתאים למעגל הסינכרוני.

Since a space-time circuit is acyclic and does not contain any clock controlled devices (in our model, unit registers), it is a combinational circuit. The analogue of clock cycles in space-time circuits is the integer component of the vertices. For example, the set of vertices performing a computation during the i'th clock cycle is the set $V \times \{i\}$. We refer to the set $V \times \{i\}$ as the i'th layer of the circuit, and denote it by TV_i. This time notion in space-time circuits is the foundation of the functional correspondence between the synchronous and the space-time circuits: The value entering vertex $v \in V$ at time t from the edge $e = u \to v$ corresponds with the value entering vertex (v, t) from the edge $(u, t - w(e)) \to (v, t)$.

Let $TY = \langle TV, TE, TF \rangle$ denote the space-time circuit obtained by applying the space-time transformation to the synchronous circuit $Y = \langle G, E, w, F \rangle$.

The next definition refers to subsets of vertices and edges of a space-time circuit.

Definition 2.3: The *cut* of TY at time t is the set of edges which emanate from vertices belonging to layers preceding layer number t and which enter vertices of layers numbered t or higher. Formally, the cut of TY at time t is defined as (see figure 2.2):

$$Cut_t(TY) \triangleq \{(u, i) \to (v, j) \in TE : i < t \leq j\}$$

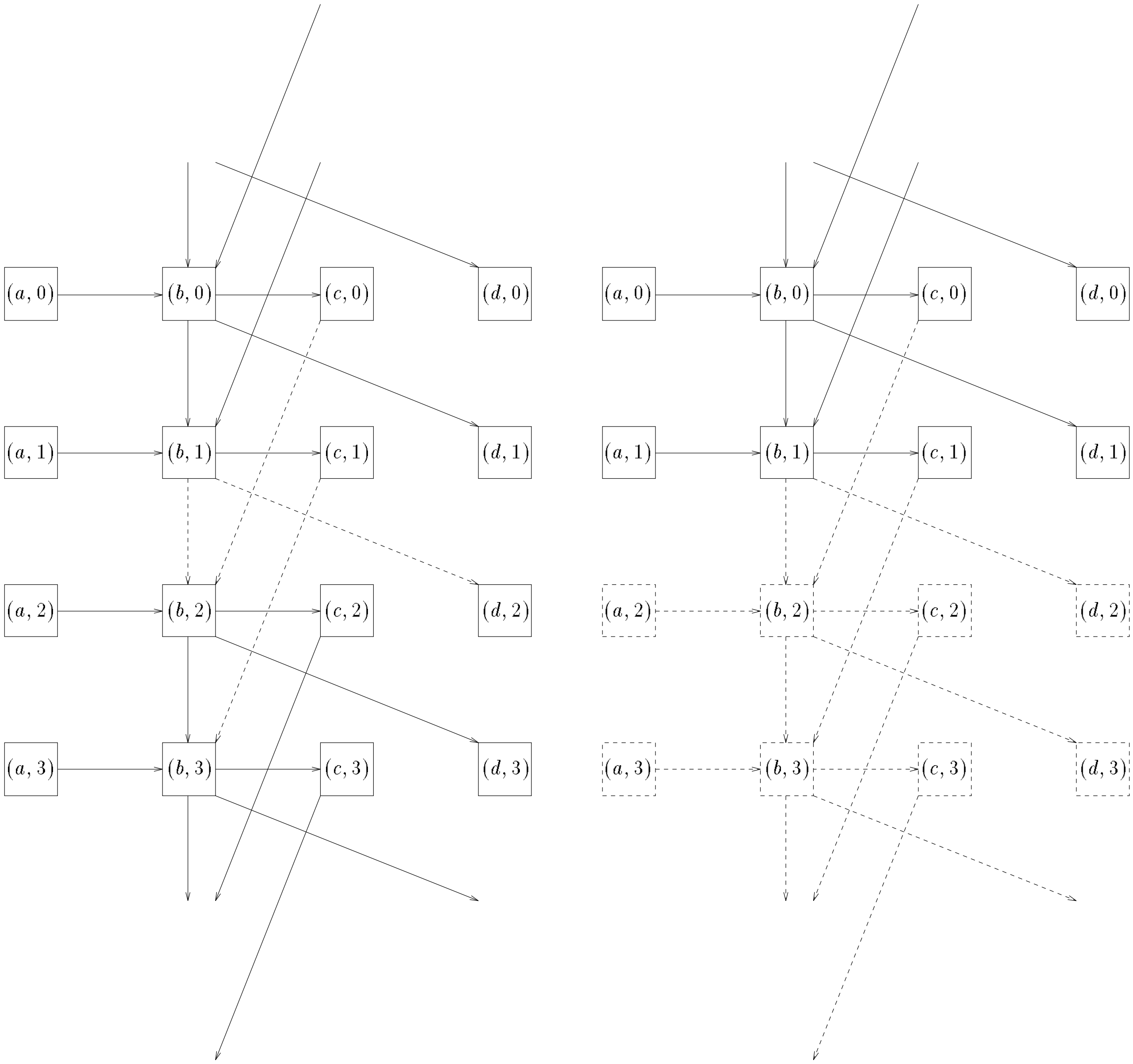
The *t-suffix* of TV is the set of vertices belonging to layers with a number greater than or equal to t, and is denoted by $TV_{j \geq t}$. The *t-suffix* of TE is the set of edges which enter vertices of the t-suffix $TV_{j \geq t}$. Formally, the t-suffix of TE is defined as (see figure 2.2):

$$TE_{j \geq t} \triangleq \{(u, i) \to (v, j) \in TE : j \geq t\}$$

Before formally defining input and output patterns, we describe the "mechanism" used for input and output. We allow multiple input hosts, where each input host may have several edges emanating it. In each clock cycle the inputs are "pumped into" the circuit by assigning a string to each input host. The length of a string assigned to an input host equals the out-degree of the input host. Each string is broken into letters, and each letter is sent along a single edge. The position of a letter in a string determines the edge along which it is sent. Output is similarly defined: We allow multiple hosts, each output host may have several incoming edges. In each clock cycle, each output host concatenates the incoming letters into strings. The position of an incoming letter in the output string is determined by the edge along which it is sent. The outputs in each clock cycle consist of the (ordered) set of strings created by the output hosts.

Definition 2.4: An input pattern of a synchronous circuit $Y = \langle G, E, w \rangle$ is a function $In : V_{in} \times \mathsf{N} \to \Sigma^*$. The number of letters in the string $In(u, t)$ equals the out degree of the host u. At time t, the host u breaks up the string $In(u, t)$ into single letters, and sends along each edge emanating it a single letter.

An input pattern of a space-time circuit $\langle TV, TE \rangle$ is similarly defined as a function $In : TV_{in} \to \Sigma^*$. The host (u, t) sends each letter of $In(u, t)$ along one edge which emanates it.



(A) (B)

Figure 2.2: (A) $Cut_t(TY)$ - cut at time $t = 2$ is denoted by the dashed edges (B) edges of the t-suffix $TE_{j\geq 2}$ and vertices of the t-suffix $TV(j \geq 2)$ are dashed.

איור 2.2: (א) החתך $Cut_t(TY)$ עבור $t = 2$ מוצג בקוים מקווקוים. (ב) קשתות הסיפא $TE_{j\geq 2}$ וצמתי הסיפא $TV_{j\geq 2}$ מקווקוים.

Although the output pattern is determined by the input pattern, by the initialization and by the functionality, we define it independently. Later, when we define computations we will show how the output pattern is determined.

Definition 2.5: An output pattern of a synchronous circuit $Y = \langle G, E, w \rangle$ is a function $Out : V_{out} \times \mathsf{N} \rightarrow \Sigma^*$. The letters of the string $Out(v,t)$ are the concatenation of the values which enter the output host v at time t.

An output pattern of a space-time circuit $\langle TV, TE \rangle$ is similarly defined as a function $Out : TV_{out} \rightarrow \Sigma^*$. The letters of the string $Out(v,t)$ are the concatenation of the values which enter the output host (v,t).

In synchronous circuits, the state describes the values stored by the circuit's memory elements. The state of a space-time circuit describes the values sent along edges of a cut of the form Cut_t.

Definition 2.6: A state of a synchronous circuit $Y = \langle G, E, w \rangle$ is a function $State : Regs(Y) \rightarrow \Sigma$. A state function, $State$, is the state of a circuit Y at time t if for every register, d, the value $State(d)$ equals the value that register d outputs at time t.

The state of a space-time circuit $\langle TV, TE \rangle$ is a function $State : Cut \rightarrow \Sigma$, where Cut equals Cut_t for some time t. Let $(u,i) \rightarrow (v,j)$ denote an edge of Cut. The value $State((u,i) \rightarrow (v,j))$ denotes the value carried along the edge $((u,i) \rightarrow (v,j))$. A state function is the state of a space-time circuit at time t if its domain is Cut_t.

Definition 2.7: An initialization of a synchronous circuit $Y = \langle G, E, w \rangle$ at time $t = 0$ is a function $Init : Regs(Y) \rightarrow \Sigma$.

An initialization of a space-time circuit $\langle TV, TE \rangle$ at time $t = 0$ is a function $Init : Cut_0 \rightarrow \Sigma$.

An initialization determines the state of the circuit at time $t = 0$. The state at time 0 equals the function $Init$, regardless of the the state of the circuit at time -1.

Given an initialization, $Init$, and an input pattern, In, a synchronous circuit performs a uniquely defined computation. Following [24], we sketch the definition of a computation of a synchronous circuit. A computation is a labeling of the edges with symbols of Σ, and is defined by a function $C : E \times \{in, out\} \times \mathsf{N} \rightarrow \Sigma$. The value entering the edge e in the i'th clock cycle is equal to $C(e, in, i)$, and the value exiting the edge e in the i'th clock cycle is equal to $C(e, out, i)$. This function C should satisfy the following properties:

1. The value output by an edge, e, at a clock cycle $i > w(e)$ is equal to the value input to the edge in clock cycle $i - w(e)$.

2. The values output by an edge, e, during the first $w(e)$ clock cycles comply with the initialization values stored in the registers along it.

3. The values entering edges which emanate from input hosts are the values specified by the input pattern.

4. The values entering edges which emanate from functional elements are compatible with the functionality of the vertex and with the values entering the vertex during the same clock cycle.

The computation, C, depends on the circuit's topology and functionality, on the initialization and on the input pattern. Whenever we want to emphasize this point we write $C_{Y,Init,In}$ instead of C. With this notation, $C_{Y,Init,In}(e, in, i)$ denotes the value entering edge e in the i'th clock cycle in circuit Y when initialized by $Init$ and given input pattern In. The output pattern is determined by the computation C as follows: Let u denote an output host, $Out(u, t)$ equals the concatenation of the values $C(e, out, t)$ for edges e which enter the output host u.

One of the main advantages of space-time circuits is that one can easily deal with computations taking place in various subsets of the circuit which are not necessarily t-suffixes of TV. For the proof of the Retiming Lemma, we need to define computations taking place in sub-circuits having a "closure property" that a path starting inside a sub-circuit does not exit it.

Definition 2.8 : Let $W \subseteq TV$ denote a subset of the vertices. Let $TY(W)$ denote the sub-circuit induced by the vertex set W. The sub-circuit $TY(W)$ is *closed,* if every edge which emanates from a vertex of W enters a vertex of W.

We would like to define computations performed by closed sub-circuits. Such a definition requires that we extend the definition of initializations and input/output patterns of space-time circuits.

Definition 2.9 : Let $TY(W)$ denote a closed sub-circuit induced by a vertex set W. Let $Cut(W^c, W)$ denote the subset of edges of TE which emanate vertices outside W and enter vertices of W.

An initialization of $TY(W)$ is a function $Init : Cut(W^c, W) \rightarrow \Sigma$.

An input pattern of $TY(W)$ is a function $In : TV_{in} \cap W \rightarrow \Sigma^*$.

An output pattern of $TY(W)$ is a function $Out : TV_{out} \cap W \rightarrow \Sigma^*$.

Given a closed sub-circuit, $TY(W)$, an initialization, $Init$, an input pattern, In, the sub-circuit $TY(W)$ performs a uniquely defined computation, since it is an acyclic combinational circuit in which values have been assigned to all the incoming edges and to all input hosts. We denote this computation by the function $C : Cut(W^c, W) \cup TE(W) \rightarrow \Sigma$. This function satisfies the following properties:

1. The computation function C is compatible with the initialization function.
2. The computation function C is compatible with the input pattern.
3. Values carried by edges emanating from a functional element are determined by the functionality of the vertex and by the values entering it.

The computation depends on the circuit TY, the subset of vertices W, the initialization $Init$ and on the input pattern In. Whenever we want to emphasize this point we denote the computation by the function $C_{TY,W,Init,In}$.

When $W = TV_{j\geq 0}$, a computation of the sub-circuit $TY(W)$ can be easily translated to a computation of the synchronous circuit Y. This is done by identifying the value entering vertex (v, t_2) from the edge $(u, t_1) \rightarrow (v, t_2)$ with the value entering vertex $v \in V$ at time t_2 from the edge $u \rightarrow v$.

Two states which have identical output patterns for all input patterns are *equivalent.* It is important to note, that all properties defined for states should be properties of equivalence classes of states. The reason for this is that one can not distinguish between equivalence states by applying input patterns and comparing the output patterns.

This work focuses on determining which initial states of a synchronous circuit have an equivalent initial state in the retimed circuit, and in this way determining the extent to which the retimed circuit can simulate the original circuit. Such a distinction requires being able to distinguish between states which are obtainable during the course of computation and states which are obtainable either only in the initialization or shortly after it. The following definition defined by Even and Litman [13] quantifies this property.

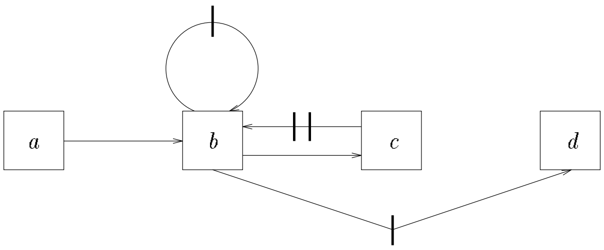
Definition 2.10 : A state function, $State$, (either of a synchronous circuit or of a space-time circuit) is m-*old,* if there exists an initialization and an input pattern such that the state of the circuit at time $t = m$ is equivalent to the function $State$.

Note that every state is 0-old, and that every $(m + 1)$-old state is also m-old.

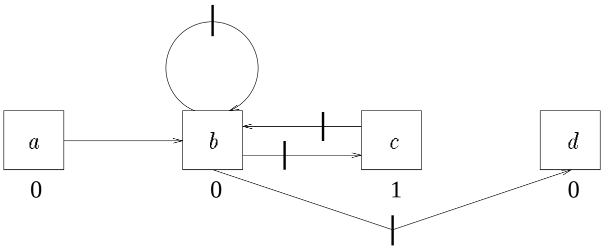
2.4 Retiming

In this section we define retiming in the domain of space-time circuits. Retiming of space-time circuits is an isomorphism of (infinite) graphs which maps a space-time circuit into an isomorphic space-time circuit. We call this isomorphism a retiming because one can retime a synchronous circuit as follows: perform a space-time transformation, retime the space-time circuit and perform the inverse of the space-time transformation on the retimed space-time circuit. This property is formally stated in Lemma 2.2. Performing retimings of synchronous circuits via retimings of space-time circuit will enable us to picture more clearly the properties of retiming.

Let $TY = \langle TV, TE \rangle$ be a space-time circuit, and let $lag : V \rightarrow \mathsf{Z}$ be a function. The retimed circuit, $TV' = \langle TV, TE' \rangle$, can be visualized as following: Suppose that the vertices TV are placed on a two dimensional grid, so that vertices of the same layer are on the same row, and vertices of the set $\{v\} \times \mathsf{Z}$ are on the same column. Suppose the edges TE are elastic. The interpretation of the retiming function is a description of how much each column should be "moved" vertically: if $lag(v) = \ell$, then the column $\{v\} \times \mathsf{Z}$ is pushed ℓ units downwards (if $\ell < 0$ then it is pulled $|\ell|$ units upwards). The retimed circuit corresponds to the original circuit after experiencing a "geological shear". Figure 2.3 depicts a retimed synchronous circuit, and figure 2.4 depicts the space-time representations of the circuits of figure 2.3.



(A)



(B)

Figure 2.3: (A) synchronous circuit
(B) retimed circuit: lags are written next to each vertex
איור 2.3: (א) מעגל סינכרוני. (ב) מעגל אחרי רה-תזמון: ערכי התזמון רשומים ליד הצמתים.

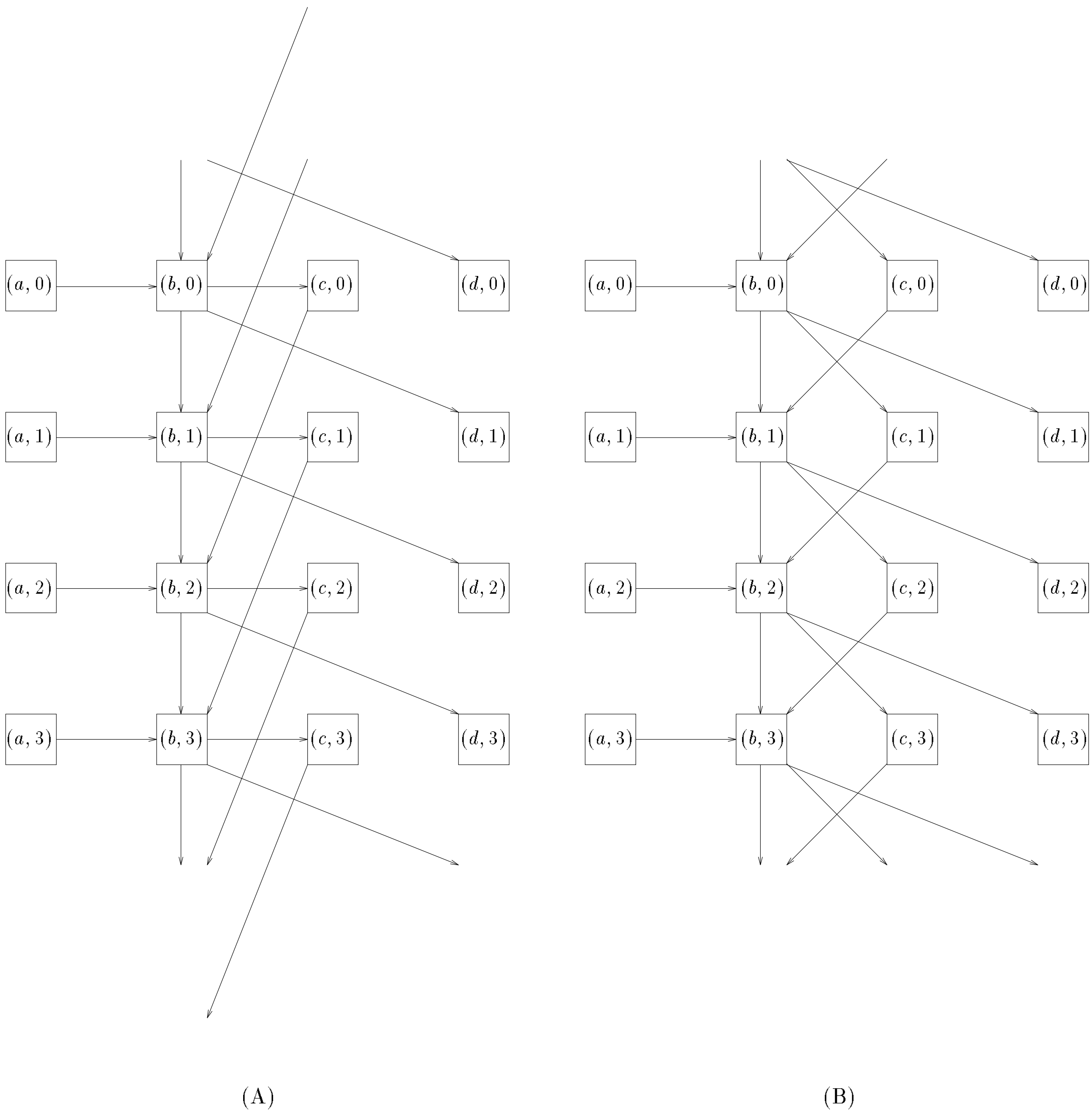


Figure 2.4: (A) original space-time circuit (B) retimed space-time circuit

איור 2.4: (א) מעגל מקום-זמן מקורי (ב) מעגל מקום-זמן אחרי רה-תזמון.

Formally, the edge set of the retimed circuit, TE', is defined as:

$$TE' \triangleq \{(u, i + lag(u)) \rightarrow (v, j + lag(v)) : (u,i) \rightarrow (v,j) \in TE\}$$

Since we want the circuit obtained by retiming a space-time circuit to be also a space-time circuit we require that the function $lag()$ satisfy:

$$\text{for every edge } (u,i) \rightarrow (v,j) \in TE: \quad i + lag(u) \leq j + lag(v) \tag{2.1}$$

A function which satisfies equation 2.1 is called a *retiming function* of space-time circuits. A retimed space-time circuit does not contain any "backward-edges", and we obtain the following lemma.

Lemma 2.1: Retiming a space-time circuit yields a space-time circuit. In other words, circuit $\langle TV, TE', TF \rangle$ satisfies all the conditions (C1-C4) of a space-time circuit.

The following lemma summarizes the relationship between retiming synchronous circuits, retiming space-time circuits, and space-time transformations. The proof is straightforward.

Lemma 2.2: The retiming and the space-time transformations commute. In other words, suppose that the synchronous circuit $\langle V, E, w', F \rangle$ is obtained by retiming the synchronous circuit $\langle V, E, w, F \rangle$ according to the function $lag()$. Suppose that the circuit $\langle TV, TE, TF \rangle$ is the space-time transformation of the synchronous circuit $\langle V, E, w, F \rangle$. Suppose that the circuit $\langle TV, TE', TF \rangle$ is obtained by retiming the circuit $\langle TV, TE, TF \rangle$ according to the function $lag()$. Then the circuit $\langle TV, TE', TF \rangle$ is the space-time transformation of the synchronous circuit $\langle V, E, w', F \rangle$. (see figure 2.5.)

$$\begin{tikzcd}
\langle V, E, w, F \rangle \arrow[r] \arrow[d, "lag()"'] & \langle TV, TE, TF \rangle \arrow[d, "lag()"] \\
\langle V, E, w', F \rangle \arrow[r] & \langle TV, TE', TF \rangle
\end{tikzcd}$$

Figure 2.5: Retiming and Space-Time Transformations Commute

איור 2.5: דיאגרמה קומוטטיבית של רה-תיזמון וטרנספרומצית מקום-זמן.

2.5 The New Proof

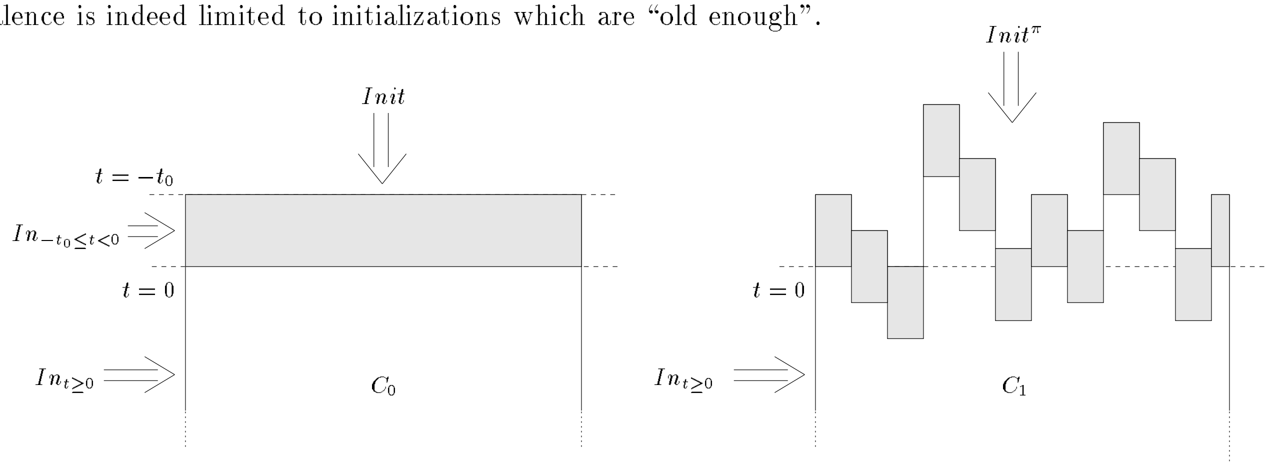
In this section we present our new proof of the Retiming Lemma. This version of the Retiming Lemma is due to Even and Litman [13].

Lemma 2.3:[retiming-lemma] Let $Y = \langle V, E, w\rangle$ be a synchronous circuit, and let $lag : V \rightarrow \mathbb{Z}$ be a retiming function. Let $Y' = \langle V, E, w'\rangle$ be the synchronous circuit resulting from retiming the circuit Y according to the function $lag()$. Let $t_0 \triangleq \max_{v \in V} lag(v)$. Suppose that $lag(v) = 0$ for every v which is an input host or an output host. Under these conditions, there exists a mapping:

$$\tau \ : \ \{t_0\text{-old states of } Y\} \rightarrow \{\text{initializations of } Y'\}$$

which satisfies: For every t_0-old state, S_Y of Y, if Y is initialized according to S_Y and Y' is initialized according to $\tau(S_Y)$, then for every input pattern both circuits have identical output patterns.

Not every initialization of the original circuit has, necessarily, a corresponding initialization in the retimed circuit. For example, consider a node with two edges emanating it e_1 and e_2, each edge having a single register. Suppose that the functionality of the node is such that it always outputs the same values along edges e_1 and e_2. One could lag this node by one clock cycle and cause edges e_1 and e_2 to become registerless. However, an initialization assigning different values to the registers of edges e_1 and e_2 does not have a corresponding initialization in the retimed circuit. This example shows that the functional equivalence is indeed limited to initializations which are "old enough".



(A) (B)

Figure 2.6: (A) computation C_0 in TY (B) computation C_1 in TY'.
איור 2.6: (א) חישוב C_0 ב-TY (ב) חישוב C_1 ב-TY'.

Proof: Let TY and TY' denote the space-time transformations of Y and Y', respectively. Since we have a correspondence between computations of a synchronous circuit and computations of its space-time transformation (for the vertex subset $TV_{j\geq 0}$), we describe the mapping τ in terms of the space-time transformations.

Since S_Y is t_0-old, there exist an input pattern $In_{-t_0 \leq t < 0}$ (defined for the time interval $-t_0 \leq t < 0$) and an initialization $Init$ (applied at time $-t_0$) which lead circuit Y into state S_0 at time $t = 0$, such that the states S_Y and S_0 are equivalent. It is sufficient to prove the lemma for the case in which S_Y equals S_0, because otherwise we can define $\tau(S_Y) = \tau(S_0)$.

Let $In_{t\geq 0}$ denote any input pattern defined in the interval $t \geq 0$. Denote the computation performed by sub-circuit $TY(TV_{j\geq -t_0})$ by the function $C_0 \triangleq C_{TY,TV_{j\geq -t_0},Init,In_{-t_0\leq t<0}\cup In_{t\geq 0}}$.

Consider the isomorphism defined by the retiming $lag()$ of the circuits TY and TY'. This isomorphism is defined by: $\pi(v,i) \triangleq (v, i+lag(v))$. We want to use this isomorphism to map computations of circuit TY to computations of circuit TY'. Such a mapping requires extending the isomorphism to initializations and input/output patterns. Since the retiming $lag()$ satisfies $lag(v) = 0$, for every vertex v which is an input host or an output host, the mapping of input/output patterns is the identity mapping. The initialization of TY' obtained by mapping the initialization $Init$ is denoted by the function $Init^\pi : \pi(Cut_{-t_0}(TY)) \rightarrow \Sigma$, and is defined by $Init^\pi(e') \triangleq Init(\pi^{-1}(e'))$. Using this mapping we obtain a computation of circuit TY' denoted by: $C_1 \triangleq C_{TY',\pi(TV_{j\geq -t_0}),Init^\pi,In_{-t_0\leq t<0}\cup In_{t\geq 0}}$.

Define $\tau(S_Y)$ to be the values carried by edges of the cut $Cut_0(TY')$ in the computation C_1.

We show the following two facts: (a) $\tau(S_Y)$ is well defined for every input pattern $In_{t\geq 0}$. (b) $\tau(S_Y)$ does not depend on the input pattern $In_{t\geq 0}$.

Fact (a) follows from the fact that $lag(v) \leq t_0$ for every vertex v. Therefore, $Cut_0(TY') \subseteq \pi(TE_{j\geq -t_0})$.

We prove fact (b) as following: If $e' = (u,i) \rightarrow (v,j) \in Cut_0(TY')$, then $i < 0$. We want to show that $\tau(S_Y)(e')$ does not depend on the pattern $In_{t\geq 0}$. This can be shown by showing that TY' does not contain a path starting at an input host (s,k) where $k \geq 0$ and which contains the edge e'. (Recall that for every input host $lag(s) = 0$). Condition C2 of space-time circuits insures that a path cannot go "backwards", therefore, such a path does not exist, and fact (b) follows.

Let $C_2 \triangleq C_{TY',TV_{j\geq 0},\tau(S_Y),In_{t\geq 0}}$ denote the computation in subcircuit $TY'(TV_{j\geq 0})$ when initialized by $\tau(S_Y)$ and given input pattern $In_{t\geq 0}$.

By the definitions of computation C_2 and initialization $\tau(S_Y)$, it follows that computations C_1 and C_2 agree on all the edges of $TE'_{j\geq 0}$. Since the retiming $lag()$ satisfies $lag(v) = 0$ for every vertex v which is an output host, computation C_0 of circuit TY and computation C_1 of circuit TY' have identical output patterns. Combining these two statements implies that for every input pattern $In_{t\geq 0}$, computations C_0 and C_2 have identical output patterns.

Translate this conclusion to the synchronous circuits Y and Y' using the inverse of the space-time transformation, and the lemma follows. $\square$

2.6 Two applications of the Retiming Lemma

In this section we discuss two applications of the Retiming Lemma. These applications are applicable only when one uses a retiming function which is *non-negative* (in other words, $lag(v) \geq 0$ for every vertex v). The first application shows that non-negative retimings fully preserve the functionality of *instantly initializable* circuits. The second application shows that the detection capabilities of test patterns are preserved after non-negative retimings.

2.6.1 Retimings of Instantly-Initializable Circuits

The first application shows that non-negative retimings preserve the functionality of circuits designed to function properly regardless of their initialization. We call such circuits *instantly-initializable*. Instantly-initializable circuits have a designated input host which serves for "initializing" the circuit. We name this input host the *reset input host*. Whenever a computation begins, the value input through the reset input host is '1' (otherwise it is '0'). Using the reset input host for marking the beginnings of computations enables the circuit to function properly as if it is initialized to a fixed state. Examples of such circuits can be found in [1, 12, 11]. Most instantly-initializable circuits contain a reset signal which sets all the registers to an "initial state". However, a circuit may be instantly-initializable even without such an explicit setting of register values.

The effect of retiming on initializations was dealt with by Even and Litman [13], where the following setting was considered: Initially, the circuit is in an arbitrary state. The circuit is brought to an initial "legal" state by an initializing input pattern which is input before the "real" inputs are given (this initializing input pattern is called a *decisive word* in [13]). After the initializing input pattern is given, the state of the circuit is equivalent to a fixed state, and thereafter the circuit behaves as desired. In this setting, the retimed circuit also has an initializing input pattern, but it is longer than the original initializing input pattern by $max_v(-lag(v))$ clock cycles. However, in the special case of non-negative retimings, an initializing input pattern of the original circuit is also an initializing input pattern of the retimed circuit. Our setting differs from that of [13] in the sense that it is not necessary to "wait" before starting inputing the "real" inputs. The initializing input host inputs a '1' *in concert* with the beginning of the computation.

In the following definition we define instantly-initializable circuits.

Definition 2.11 : A synchronous circuit Y is *instantly-initializable* with respect to an input host v if for every input pattern In which satisfies: (a) $In(v,0) = 1$ and (b) $In(v,t) = 0$ for every $t > 0$, the output pattern after the first clock cycle does not depend on the initialization of Y.

We consider the output pattern only after the first clock cycle, because in the first clock cycle the output pattern might depend on the initialization, and we do not want to exclude this possibility. In fact, an instantly-initializable circuit has at most one register along edges entering output hosts.

This sub-section shows that retiming an instantly-initializable circuit with a non-negative retiming results in an instantly-initializable circuit. Furthermore, the retimed circuit is equivalent to the original circuit with respect to input patterns which "mark" the beginnings of computations. This property is implicitly claimed in [1, 12, 11]. The next lemma formalizes this property.

Lemma 2.4: Let Y be an instantly-initializable synchronous circuit with respect to an input host v. Let Y' be a synchronous circuit obtained by retiming circuit Y by a non-negative retiming function lag. Suppose that $lag(v) = 0$ for every vertex which is an input host or an output host. Let $Init$ be any initialization of circuit Y and let $Init'$ be any initialization of circuit Y'. If In is an input pattern which satisfies (a) $In(v,0) = 1$ and (b) $In(v,t) = 0$ for every $t > 0$, then the following output patterns are identical after the first clock cycle: The output pattern of circuit Y when initialized with $Init$ and given the input pattern In and the output pattern of circuit Y' when initialized with $Init'$ and given the input pattern In.

Note that we do not claim that the original circuit and the retimed circuit have identical output patterns for every input pattern. We restrict this equivalence to input patterns which "mark" the beginnings of computations.

Proof: The circuit Y' is obtained by retiming the circuit Y according to the retiming function lag. Define a retiming function $r(v) \triangleq -lag(v)$. The function $r()$ is a non-positive retiming function which maps the circuit Y' to the circuit Y. The function $r()$ satisfies all the assumptions of the Retiming Lemma, and it also satisfies: $max_v r(v) = 0$. Since every state is 0-old, then according to the Retiming Lemma, every initialization $Init'$ of circuit Y' has a corresponding initialization $Init_1$ of circuit Y, such that Y and Y' are equivalent when initialized by $Init_1$ and $Init'$ respectively.

Since circuit Y is instantly-initializable with respect to input host v and since the input pattern In satisfies (a) $In(v,0) = 1$ and (b) $In(v,t) = 0$ for every $t > 0$, the output pattern of Y does not depend on the initialization. This means that the following output patterns are identical after the first clock cycle: The output pattern of circuit Y when initialized with $Init_1$ and given the input pattern In, and the output pattern of circuit Y when initialized with $Init$ and given the input pattern In. Since Y and Y' are equivalent when initialized by $Init_1$ and $Init'$ respectively, the lemma follows. □

2.6.2 Retiming and Test Patterns

The second application deals with the effect of non-negative retimings to the detection capabilities of test patterns. A test pattern is an input pattern which when given to the circuit can verify that the circuit was in a correct initial state and that each of the vertices is functioning correctly. In this subsection we show that non-negative retiming does not make testing harder. In particular, a test pattern for a synchronous circuit is also a test pattern for a non-negative retiming of it. The motivation for this question is that finding test patterns for circuits before they are retimed (in other words, for semi-systolic circuits) seems to be easier than finding test patterns after retiming.

The test patterns we are referring to in this subsection are test patterns which detect faults belonging to a fault model which consists of two types of faults:

faulty initialization The initialization of the circuit does not belong to a predefined set of initializations, called the "correct" initializations. (Note that a test pattern does not necessarily enable one to find out the exact initialization of the circuit, it only verifies that the initialization belongs to the set of "correct" initializations).

faulty vertices Some of the vertices compute erroneous functions. (Recall that the vertices are combinational elements which simply compute functions).

Note that (a) Faulty registers (e.g. registers in which the output value does not equal the input value at the end of the previous clock cycle) can be "mapped" to faulty vertices. (b) In an instantly-initializable circuit, all initializations are "correct", and therefore, a test pattern only checks for faulty vertices.

In the next definition we define test patterns.

Definition 2.12 : Let $Y = \langle V, E, w, F \rangle$ be a synchronous circuit. An input pattern In_{test} is a test pattern for Y with respect to a set of initializations *Correct* if the following holds: for every synchronous circuit $Y_1 = \langle V, E, w, F_1 \rangle$, the output pattern of circuit Y with initialization $Init$ and input pattern In_{test} does not equal the output pattern of circuit Y_1 with initialization $Init_1$ and input pattern In_{test}, whenever at least one of the following conditions holds:

(a) $Init \in$ *Correct* and $Init_1 \notin$ *Correct.*

(b) $F \neq F_1$.

Let Y be a synchronous circuit. Let Y' be a synchronous circuit obtained by retiming circuit Y by a non-negative retiming function lag. Suppose that $lag(v) = 0$ for every vertex which is an input host or an output host. Let $t_0 \triangleq \max_{v \in V} lag(v)$. Suppose that In_{test} is a test pattern for Y with respect to a set of initializations *Correct*. Before we prove anything about the detection capabilities of the test pattern In_{test} for the retimed circuit Y', we must check the the retimed circuit also has "correct" initializations. Otherwise, one can not consider the retimed circuit to be equivalent to the original circuit. The set of "correct" initializations of Y' (denoted by *Correct'*) is the set of states obtained by mapping the t_0-old states in the set *Correct* to initializations of Y' according to the Retiming Lemma. The set *Correct'* will be non-empty if and only if the set *Correct* contains t_o-old states. Using this notation we obtain:

Lemma 2.5: Suppose that the retiming function $lag()$ is non-negative. If the set *Correct'* is non-empty, then In_{test} is a testing pattern of Y' with respect to the set of initialization *Correct'*.

Proof: Let $Y_1' = \langle V, E, w', F_1 \rangle$ be a candidate for a "faulty" instance of $Y = \langle V, E, w', F \rangle$. Let $Init' \in$ *Correct'* and let $Init_1'$ be an initialization of Y_1'. We consider two cases: (a) $Init_1' \notin$ *Correct'* and (b) $F \neq F_1$.

We use the retiming $t() \triangleq -lag()$ for analyzing the computations of Y' and Y_1'. The circuit $Y = \langle V, E, w, F \rangle$ is obtained by retiming Y' according to $t()$, which is a non-positive retiming function. Let $Y_1 = \langle V, E, w, F_1 \rangle$ denote the circuit obtained by retiming Y_1' according to $t()$. Let $Init$ be the initialization of Y obtained by mapping the initialization $Init'$ to an initialization of Y. The definition of *Correct'* implies that, if $Init' \in$ *Correct'* then $Init \in$ *Correct*. Let $Init_1$ be the initialization of Y_1 obtained by mapping the initialization $Init_1'$ to an initialization of Y_1. By the definition of *Correct'*, $Init_1' \in$ *Correct'* if and only if $Init_1 \in$ *Correct*.

We now consider cases (a) $Init' \notin$ *Correct'*. If $Init_1' \notin$ *Correct'*, then $Init_1 \notin$ *Correct*. Since In_{test} is a test pattern of Y with respect to *Correct*, we conclude that the output pattern of Y with initialization $Init$ and input pattern In_{test} does not equal the output pattern of Y_1 with initialization $Init_1$ and input pattern In_{test}. Since Y and Y' are equivalent when initialized by $Init$ and $Init'$ respectively, and since Y_1 and Y_1' are equivalent when initialized by $Init_1$ and $Init_1'$ respectively, we conclude that In_{test} detects faulty initializations of Y'.

Consider case (b) $F \neq F_1$. Since In_{test} is a test pattern of Y, we use the same reasoning to conclude that In_{test} detects faulty vertices of Y'.

We have shown that In_{test} detects faulty initializations and faulty vertices of Y', and therefore, the lemma follows. □

Chapter 3

Finding Initializations After Retiming

3.1 Introduction

The Retiming Lemma 2.3 characterizes the initializations (i.e. the values stored in the registers of the circuit when the computation starts) of the original circuit which have equivalent initializations in the retimed circuit [21, 12, 7]. This equivalence ensures that if the retimed circuit is initialized by an initialization which is equivalent to the initialization of the original circuits, then the outputs of both circuits will be identical, for every input pattern. However, the problem of finding an equivalent initialization is very difficult (in Claim 3.1 we show that it is PSPACE-hard). This chapter focuses on the problem of how to initialize a circuit after is has been retimed. We consider settings in which one wishes to retime a circuit so that its clock period will not exceed a given period. A retiming satisfying this goal is called a feasible retiming.

Touati and Brayton [31] deal with the problem of finding an equivalent initialization after retiming. The starting point of their work is a circuit and a given retiming of it. They describe how to find an equivalent initialization, if the retiming function is non-positive. In such a case, a solution can always be found with a simple and efficient procedure. When a retiming has positive values, they propose adding logic and a reset mechanism, which enables one to easily find a precomputation consisting of any number of transitions which leads the original circuit to the initialization. Finding an equivalent initialization in the retimed circuit is simple when such a precomputation is known. Apart from introducing a hardware overhead, the added logic degrades the performance of the retimed circuit and ruins the gains obtained by the retiming. Our approach is antipodal: we seek retiming functions which attain identical benefits and for which the problem of finding equivalent initializations is as easy as possible.

First, we restrict the problem of finding an equivalent initialization to a problem of finding a corresponding initialization. Every corresponding initialization is an equivalent initialization, but the converse is not true. The advantage of trying to find a corresponding initialization instead of an equivalent one, is that corresponding initializations have short proofs of their validity (i.e. the problem of validating that an initialization is a corresponding initialization is in NP). We present a procedure for finding a corresponding initialization in Subsection 3.3.3.

Second, we define two heuristic cost functions, which estimate the cost of the search for a corresponding initialization. The first heuristic cost function is the maximum positive lag. The second heuristic cost function is the weighted sum of the positive lags. We explain the significance of these heuristic cost functions in light of the procedure for finding a corresponding initialization. In the extreme case, in which there exists a feasible retiming for which finding a corresponding initialization is trivial, the cost of optimal feasible retimings with respect to both cost functions is zero.

Third, we deal with the problem of finding optimal feasible retimings with respect to each of the heuristic cost functions. Algorithm Rev-FEAS, which is based on the FEAS algorithm of Leiserson and Saxe [23], is presented. Algorithm Rev-FEAS finds an optimal feasible retiming with respect to the first cost function, if feasible retimings exist. The overhead of running Rev-FEAS over FEAS is linear. In addition, we present a reduction of the problem of finding an optimal feasible retiming with respect to the second cost function to the problem of finding a feasible retiming with a minimum total number of registers (also called a state-minimization problem). The costliest computation involved in executing the reduction is solving an all-pairs shortest-paths problem. The state-minimization problem can be solved efficiently using previous algorithms [23, 5], and basically requires solving a min-cost flow problem.

Organization: Section 3.2 defines retiming lightest paths and critical paths. Section 3.3 deals with finding initializations after retiming. First, the problem of finding an equivalent initialization is presented. We continue by restricting the problem to finding a corresponding initialization, and discuss how a corresponding initialization is found. Examples are given which demonstrate these issues. Section 3.4 presents two heuristic cost functions which estimate the cost of the search for a corresponding initialization. Section 3.5 contains an overview of previous work of Leiserson and Saxe; the problems they dealt with are described, and the algorithms they presented for solving these problems are discussed briefly. Section 3.6 presents algorithm Rev-FEAS which finds an optimal feasible retiming with respect to the cost function defined by the maximum lag. Algorithm Rev-FEAS is based on two algorithms presented by Leiserson and Saxe: The first algorithm, CP, computes the clock period of a synchronous circuit, and we present a correction of algorithm CP. The second algorithm, FEAS, finds a feasible retiming, and we review algorithm FEAS and provide a detailed proof of its properties and correctness. Section 3.7 presents a reduction from the problem of finding an optimal feasible retiming with respect to the second cost function to the problem of finding a feasible retiming with a minimum total number of registers.

3.2 Lightest paths and critical delays

Consider the communication graph $Y = (V, E, w)$ of a circuit with non-negative vertex delays, $d(v)$. Leiserson and Saxe [23] define the following two quantities for every pair of vertices. The *weight of a lightest path* between vertex a and vertex b is denoted by $W(a,b)$. The *critical delay* between vertex a and b, $D(a,b)$, is the maximum delay along a lightest path between a and b. Formally, let $a \overset{p}{\rightsquigarrow} b$ denote a path p starting from vertex a and ending in vertex b, then $W(a,b)$ and $D(a,b)$ are defined as:

$$\begin{aligned} W(a,b) &\triangleq \min\left\{w(p) : a \overset{p}{\rightsquigarrow} b\right\} \\ D(a,b) &\triangleq \max\left\{d(p) : a \overset{p}{\rightsquigarrow} b \text{ and } w(p) = W(a,b)\right\} \end{aligned}$$

The importance of these quantities is summarized in the following lemma.

Lemma 3.1: Let $Y = \langle V, E, w, d \rangle$ be a communication graph.

1. The clock period, $\Phi(Y)$, is greater than c if and only if there exists a pair of vertices for which $W(a,b) = 0$ and $D(a,b) > c$.
2. Let $r()$ be a retiming function of Y, let $W_r()$ denote the weight of lightest paths in Y_r, let $D_r()$ denote the critical delays in Y_r. Then $W_r(a,b) = W(a,b) - r(a) + r(b)$ and $D_r(a,b) = D(a,b)$, for every $a, b \in V$.

Lemma 3.1 shows that a retiming of Y enables a clock period c in the retimed communication graph if and only if for every pair of vertices a, b for which $D(a,b) > c$ we have $W_r(a,b) > 0$. Leiserson and Saxe show how to compute these quantities by reducing the problem of computing these quantities to an all-pairs shortest-paths problem [23].

3.3 Finding initializations after retiming

The motivation for retiming is that it is a transformation which enables improving the performance of circuits without modifying their functionality. Consequently, retiming is a methodology for designing circuits: First, design a circuit the correctness of which is easily understood, but which may have a poor performance, and then retime it to improve its performance. The correctness of the retimed circuit is guaranteed from the correctness of the original circuit. However, the correctness of the original circuit depends on its initialization (i.e. the initial values stored in the registers), and one must find a corresponding initialization of the retimed circuit.

The Retiming Lemma 2.3 characterizes the initializations of the original circuit which have equivalent initializations in the retimed circuit. In practice, determining whether an initialization has an equivalent initialization in the retimed circuit (using the criteria of the Retiming Lemma) is as hard as finding such an initialization. Therefore, this characterization of "good" initializations might not be helpful in practice. We propose a restriction of the problem of finding an equivalent initialization in the retimed circuit called the problem of finding a corresponding initialization. The restricted problem is easier to solve, although a solvable instance of the problem of finding an equivalent initialization might not be solvable under this restriction.

The discussion in this section is prerequisite for understanding the significance of the heuristic cost functions defined in Section 3.4. These cost functions measure the cost of searching for a corresponding initialization.

This section is organized as follows: In Subsection 3.3.1 we define the problem of finding equivalent initializations. In Subsection 3.3.2 we define the problem of finding a corresponding initialization after retiming. In subsection 3.3.3 we discuss how the search for a corresponding initialization is conducted. We end this section with examples which demonstrate the discussed issues.

3.3.1 Finding equivalent initializations

Given a retiming of a circuit, a solution to the following problem is sought:

Finding an equivalent initialization

input: A circuit Y, an initialization S, and a retiming $r()$.

output: Let Y_r denote the circuit obtained by retiming Y by $r()$. Find an initialization S_r, so that Y initialized by S and Y_r initialized by S_r are functionally equivalent. (Namely, if the same values are input in all clock cycles to both circuits, then identical values are output during all the clock cycles). If no such S_r exists, then output *"fail"*.

Note that functionally equivalent circuits may have different clock periods. The definition of functional equivalence only guarantees that if the same values are input in all clock cycles to both circuits, then identical values are output during all the clock cycles.

The following claim shows that finding an equivalent initialization is PSPACE-hard.

Claim 3.1: The problem of finding an equivalent initialization is PSPACE-hard.

Proof: We reduce the QBF problem to the problem of finding an equivalent initialization. In the QBF problem the input consists of a quantified boolean formula with no free variables, and the output is truth value of the formula (namely, "true" or "false"). The QBF problem is PSPACE-hard (the language consisting, TQBF, of all quantified boolean formulas with a "true" value is PSPACE-complete) [17].

Since QBF in is PSPACE, there exists a Turing Machine, M, which computes the truth value of a QBF, and which requires n^c space where n is the length of the formula, and c is a constant. Given a QBF, φ, we construct a circuit, Y_φ, of polynomial size which "runs" M on the formula φ. Construct circuit Y_φ so that it has a reset mechanism which enables resetting the value of every register in circuit Y_φ. This reset mechanism works by a signal $Init$, and is depicted in figure 3.1 (an explanation of the symbols in the figure is given in the next paragraph). The output of circuit Y_φ starts with value 1, and changes to 0 when the circuit first finds that φ has a "false" value.

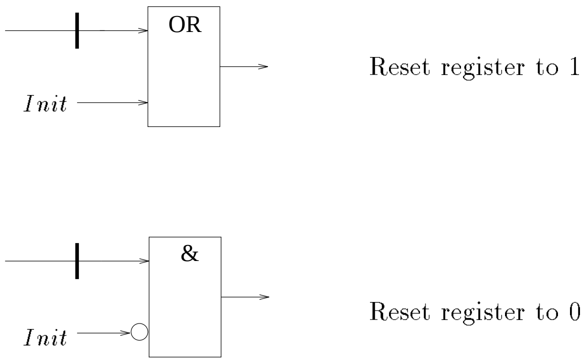


Figure 3.1: Resetting the registers by $Init$.
איור 3.1: איתחול רגיסטרים על-ידי $Init$.

Figure 3.2 shows a schematic of circuit Y which we use in the reduction. The circuit has a single input called In and a single output called Out. The meaning of the schematic is as follows: A box with an "&" symbol is an AND gate. A circle on an input to a gate means that the input is inverted before entering the gate. A box with an "OR" in it is an OR gate. Registers are denoted by thick segments perpendicular to a wire. The box with a "MUX" in it denotes a multiplexor which routes the input marked by "1" to the output when the value of input s equals 1, and when the value of input s is 0 the input marked by "0" is routed to the output. The counter outputs a 1 every T cycles, where T is a bound on the number of configurations that Turing machine, M, has on input φ. (Since M runs with polynomial space, T is exponential in the space of M, and the counter requires only $\log T$ registers and hardware, which is polynomial in the length of φ).

The initialization, S, of circuit Y which we consider is defined by: (a) $R_1 = 0$ (b) R_2=1 (c) The registers of Y_φ are initialized so that the state of Y_φ corresponds to the initial state of the Turing machine M with input φ. (d) The counter is initialized to zero.

We consider two cases: (a) the value of formula φ is "true" (b) the value of formula φ is "false". In case (a), the inputs of the MUX marked by "1" and "0" always have value 1, and therefore, for every input sequence, circuit Y outputs a 1 forever. In case (b), the output of circuit Y_φ equals 1, changes to 0 when it is detected that the truth value of φ is "false", and returns to 1 when Y_φ is initialized by $Init$.

Consider a retiming in which the combinational units in the dotted ellipse marked by v are lagged by 1 , and all other vertices are left unchanged (namely, they are lagged by zero). This retiming causes register R_1 "to move" from the output of v to the input of v. Does the retimed circuit, Y_r, have an initialization, S_r, so that Y initialized by S is functionally equivalent to Y_r initialized by S_r?

We consider cases (a) the truth value of φ is "true" and (b) the truth value of φ is "false":

In case (a), consider an initialization, S_r, which agrees with S on the values of the registers of Y_φ and the registers of the counter. Circuit Y_r outputs a 1 forever, because the inputs to the MUX are both 1. Hence, Y initialized y S is functionally equivalent to Y_r initialized by S_r.

In case (b), we consider an input sequence in which the value of In equals 1 during the first clock cycle, and remains 0 for all other clock cycles. The value of input s of the MUX in circuit Y for this input sequence is always 1. Hence the output of Y initialized by S is always 1. However, the value of input s of the MUX in circuit Y_r for this input sequence is always 0. Since Y_φ is initialized periodically by the counter, eventually it will detect that the truth value of φ is "false" and output a 0 (which is input to the MUX). This 0 value is output by the MUX (since the s input has value 0), and therefore, circuit Y_r outputs eventually a 0. Note that Y_r outputs a 0 regardless of its initialization. We conclude that, in case (b), Y_r does not have an initialization, S_r, for which Y initialized by S is functionally equivalent to Y_r initialized by S_r.

This discussion implies that solving the problem of finding an equivalent initialization with circuit Y after this retiming also resolves the question of whether the truth value of φ is "true" or "false". Hence, the problem of finding an equivalent initialization is PSPACE-hard, and the claim follows. $\square$

3.3.2 Corresponding initializations

In this subsection we define a restriction of the equivalent initialization problem which we call the corresponding initialization problem. This problem is defined as follows:

The Corresponding Initialization Problem

Input: A circuit Y, an initialization S, and a retiming $r()$.

Output: Let TY denote the space-time circuit corresponding to to circuit Y, and let TY_r denote the space-time circuit corresponding to circuit Y_r. Let $C_{TY,S,In}$ denote a

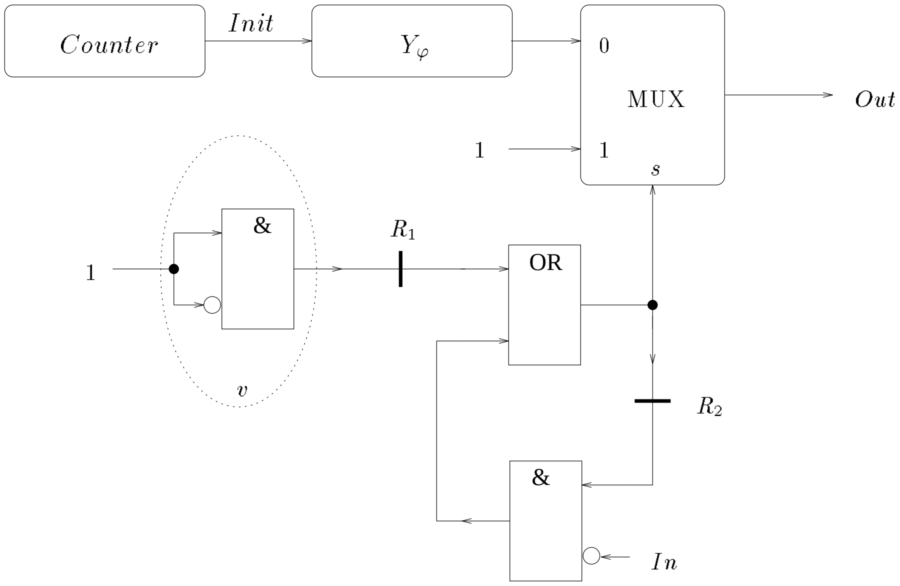


Figure 3.2: A schematic of circuit Y.
איור 3.2: תרשים סכמטי של מעגל Y.

computation (labeling of the edges) of TY obtained by the initialization, S, and by an input sequence, In. Find an initialization, S_r, of the retimed circuit,Y_r, so that

$$C_{Y,S,In}((u,i) \to (v,j)) = C_{Y_r,S_r,In}((u,i+r(u)) \to (v,j+r(v)))$$

for every edge $(u,i) \to (v,j)$ for which $j \geq \max\{0, r(v)\}$, and for every input sequence In.

If no such initialization exists output "fail".

A solution to the equivalent initialization problem might not be a valid solution to the corresponding initialization problem. Therefore, given the same input, a solution to the corresponding initialization problem can be "fail", whereas the equivalent initialization problem has a valid solution. The difference between the equivalent initialization problem and the corresponding initialization problem is best explained by considering the time-space circuits, TY and TY_r corresponding to circuit Y (before retiming) and to circuit Y_r (after retiming). A solution to the equivalent initialization problem must provide identical output values, which translate to equal labels on edges which enter the host in circuits TY and TY'. However, a solution to the corresponding initialization problem must have equal labels on many edges, most of which do not enter the host (i.e. are not outputs). Therefore, the corresponding initialization problem is a restriction of the equivalent initialization problem.

This restriction guarantees that short proofs of the validity of solutions to the corresponding initialization problem always exist (namely, the corresponding initialization is in NP), whereas the equivalent initialization problem is PSPACE-hard.

3.3.3 Finding a corresponding initialization

The corresponding initialization in the retimed circuit is found using two primitives: (a) A simulation primitive in which the labels of the edges emanating from a vertex are determined by the labels of the incoming edges. (b) A back-tracing primitive in which the labels of edges incoming a vertex are guessed from the labels of the edges emanating from the vertex. Note that a simulation primitive is straightforward and always succeeds, whereas a back-tracing primitive might be hard to compute, and it might have several feasible solutions or even no feasible solutions.

An initialization, S_r, is found in two phases: (a) A simulation phase in which simulation primitives are applied to the set of vertices $\{(v,j) \in TY_r : r(v) \leq j < 0\}$ in topological order (i.e. order the vertices so that if $(u,i) \rightarrow (v,j) \in TE_r$, then (u,i) comes before (v,j)). (b) A back-tracing phase in which back-tracing primitives are applied to the set of vertices $\{(v,j) \in TY_r : 0 \leq j < r(v)\}$ in reversed topological order. Only the back-tracing phase might require large computation resources (and it might even fail). When the normalized retiming function is non-positive no back-tracing primitives are required, and an initialization, S_r, is easily found. Touati and Brayton [31] restricted their discussion to finding a corresponding initialization for non-positive retimings, which is the trivial case.

3.3.4 Examples

Example 3.1: Figure 3.3 depicts a circuit, and figure 3.4 depicts a communication graph of this circuits. The functional units corresponding to the vertices of the communication graph are shown as dotted boxes in figure 3.3. Note, that the vertices of the communication graph can be defined in various ways.

Example 3.2: Consider the communication graph depicted in figure 3.4. Consider a retiming function defined by: $r(host) = 0, r(a) = 0, r(b) = 1$. Figure 3.5 depicts the communication graph after retiming. The retimed circuit has two registers, R_1' and R_2'.

Example 3.3: Figure 3.6 depicts a segment of the space-time diagram of the communication graph shown in figure 3.4.

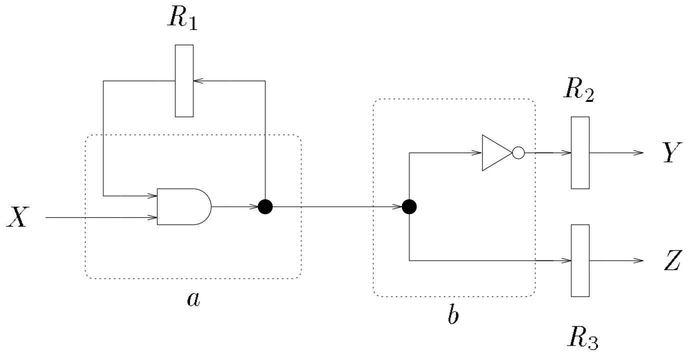


Figure 3.3: A synchronous circuit.
איור 3.3: מעגל סינכרוני.

Example 3.4: Figure 3.7 depicts a computation in the segment of the space-time diagram shown in figure 3.6. Labels are written adjacent to the edges. Note, that there are two edges from vertex b to the host, the top one corresponds to the edge with register R_2, and the bottom one to the edge with register R_3. The initialization in figure 3.7 is $R_1 = 1, R_2 = 1, R_3 = 1$, and the input pattern is $1, 1, 0, 1$ (refer to the labels on the edges $(H, i) \to (a, i)$). The output pattern is $(1, 1), (0, 1), (0, 1), (1, 0)$, where in each pair the output of R_2 appears first, and the output of R_3 appears second.

Example 3.5: Figure 3.8 depicts a segment of the space-time diagram of the retimed communication graph shown in figure 3.5. The labels induced from the labels of the space-time diagram shown in figure 3.7 are shown as well. Note that no labeling is induced on the edge entering vertex $(b, 0)$, although an initialization of the retimed circuit should specify a label of this edge.

Example 3.6: Solving the corresponding initialization problem on the circuit depicted in figure 3.3, initialized as in figure 3.7, and retimed as depicted in figure 3.5, involves back-tracing the computation of vertex $(b, 0)$ when the output values of the vertex are $(1, 1)$. However, the output values of vertex $(b, 0)$ are always complementary, and therefore, no solution to this back-tracing primitive exists.

3.4 Two heuristic cost functions

Given a communication graph and a required clock period, many feasible retimings may exist. A feasible retiming is a retiming which induces a communication graph which achieves

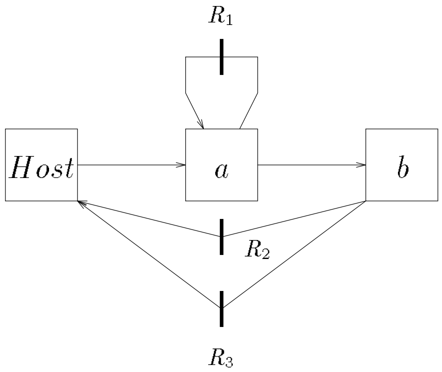


Figure 3.4: A communication graph of the circuit of figure 3.3.
איור 3.4: גרף התקשורת של המעגל מאיור 3.3.

the required clock period. In Subsection 3.3.3, we showed how a corresponding initial state in the retimed circuit can be found. The crucial part in this search is the back-tracing phase, and therefore, one would prefer a retiming which minimizes the complexity of searching for an initialization. Quantifying the cost of this search is a difficult task since the cost is data-dependent. In this section, we discuss the difficulty of estimating the cost of the search for a corresponding initialization, and we propose two heuristic cost functions.

The difficulty of the back-tracing phase is caused by two problems: (a) A back-tracing primitive is an inversion of a function. Inversion of some functions is hard, and even determining whether the back-tracing primitive has a non-empty solution might be hard. (b) A back-tracing primitive may have more than one solution or none at all. During the search, whenever one encounters a back-tracing primitive which has no solution, one needs to backtrack the search to a back-tracing primitive which had more than one solution, and try the other solutions. Therefore, even if in typical circuits (e.g. gate level descriptions of circuits) the first problem does not impose a significant problem, then the second problem may impose a search of exponential complexity.

Given a circuit and a retiming of the circuit, it is difficult to estimate the cost of the search for a corresponding initialization. The reason for this is that the solutions of the back-tracing primitives depend very strongly on the values emanating from the given vertex. Therefore, it is impossible to assess (without knowing, S, the original initialization) the total number of possibilities which might be required to scan before one can determine that a corresponding initialization does not exist.

We propose two heuristic cost functions which estimate the cost of the back-tracing phase. The first cost is the maximum positive lag (assuming that the retiming is normalized, that is, $lag(Host) = 0$). This cost function measures the depth of the sub-graph (of the space-time diagram) in which the back-tracing phase takes place. When a retiming function

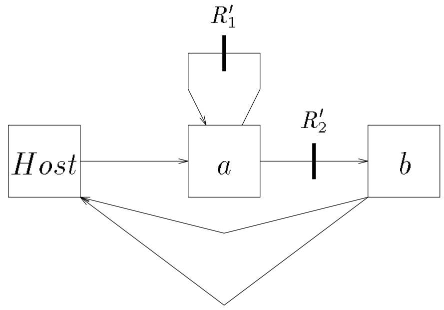


Figure 3.5: The communication graph of the circuit after retiming.
איור 3.5: גרף התקשורת של המעגל לאחר רה-תזמון.

is non-positive, then this cost function is zero, and the back-tracing phase is null. In particular, minimizing the maximum lag guarantees, that if a non-positive feasible retiming exists, then such a retiming will be found.

The second cost function, called the weighted sum of positive lags, is based on attaching a "price" to back-tracing each vertex, denoted by $p(v)$. The cost function is defined by $\sum_{\{v:r(v)>0\}} p(v) \cdot r(v)$ (again, we assume that the retiming is normalized). This cost function measures the total "price" of the back-tracing phase. For example, the price $p(v)$ can be defined to be the maximum (or average) number of solutions of a back-tracing primitive applied to vertex v. Alternatively, one may prefer to assign unit prices to all back-tracing primitives, in which case, this cost function counts the total number of back-tracing primitives required if all back-tracing primitives are successful.

3.5 Overview of Previous Algorithms

In this section we overview the algorithms proposed by Leiserson and Saxe [23] which deal with the following problems:

The clock feasibility problem Given a communication graph $Y = \langle V, E, w, d \rangle$ and a required clock period c, find a retiming $r()$, such that $\Phi(Y_r) \leq c$ or answer that no such retiming exists.

The state minimization problem Consider a communication graph, $Y = \langle V, E, w, d \rangle$, in which a breadth is assigned to each edge. The breadth of an edge signifies the number of bits sent in parallel along the edge. An edge with weight $w(e)$ and breadth $\beta(e)$ is implemented by $\beta(e) \cdot w(e)$ registers, and the total number of registers in a circuit is $\sum_{e \in E} \beta(e) \cdot w(e)$.

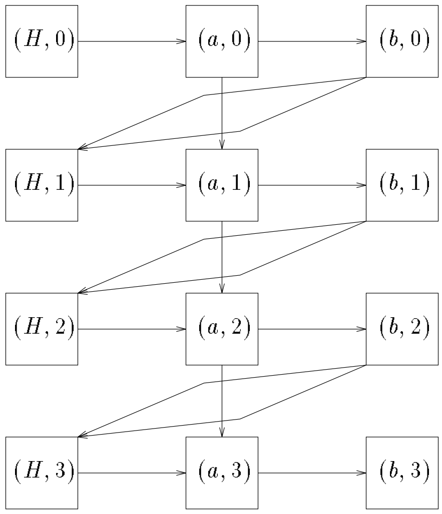


Figure 3.6: Four layers of the space-time diagram of the communication graph from figure 3.4.

איור 3.6: ארבע שכבות של דיאגרמת מקום-זמן של גרף התקשורת מאיור 3.4.

The inputs to a the state minimization problem are a communication graph, $Y = \langle V, E, w, d\rangle$, and the breadth of the edges, $\beta(e)$. The problem is to find a retiming, $r()$, which minimizes the total number of registers in the retimed circuit. In other words, find a retiming function, $r()$, which minimizes the sum $\sum_{e\in E} \beta(e)\cdot w_r(e)$.

The state minimization problem has a form in which a required clock period is part of the input. The problem, in this form, is to find a feasible retiming function, $r()$, which minimizes the sum $\sum_{e\in E} \beta(e)\cdot w_r(e)$.

Leiserson and Saxe propose three algorithms for solving the clock feasibility problem (let n denote the number of vertices, and m denote the number of edges in the graph):

1. A reduction of the clock feasibility problem to a set of Bellman-Ford inequalities [23]. The set of inequalities is over the unknowns $\{r(v)\}_{v\in V}$:

$$\begin{aligned} r(a) - r(b) &\leq w(a,b) && \text{for every edge } (a,b)\in E \\ r(a) - r(b) &\leq W(a,b) - 1 && \text{for all vertices } a,b \text{ such that } D(a,b) > c \end{aligned}$$

From lemma 3.1 it follows that $r()$ is a solution to this set of inequalities if and only if it is a retiming function which induces a retimed communication graph with a clock period not greater than c. Therefore, if the Ford-Bellman algorithm finds a solution, then a required retiming is found. Otherwise, no solution exists, and the

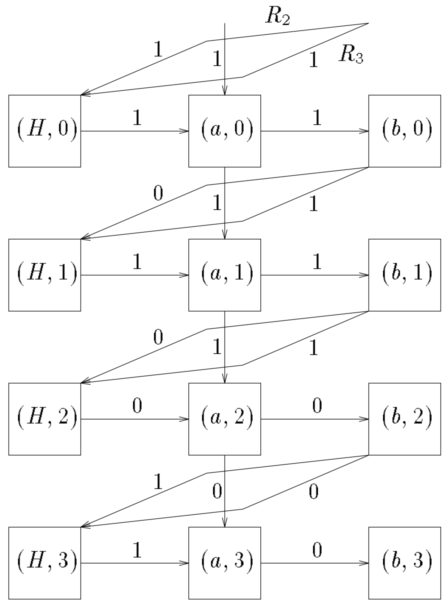


Figure 3.7: A computation in the segment of the space-time diagram from figure 3.6.
איור 3.7: חישוב בקטע של דיאגרמת מקום-זמן מאיור 3.6.

communication graph cannot be retimed as required. The complexity of this algorithm is $O(n^3)$ because there are $O(n^2)$ inequalities. In addition, finding the constraints requires computing the critical delays and the lightest paths in the communication graph. This is done by solving an all-pairs shortest-paths problem.

2. A reduction to a special mixed integer linear programming algorithm, with complexity $O(n \cdot m \cdot \log n + n^2 \cdot log^2 n)$ [22].

3. Algorithm FEAS, with complexity $O(n \cdot m)$ [23]. A detailed review of algorithm FEAS is given in Section 3.6.2.

The state minimization problem, including the form in which a required clock period is given, is solved by using a primal-dual linear programming technique. The dual of the linear program which corresponds to the state minimization problem is a min-cost flow problem, and therefore, an integral optimal solution can be found efficiently [23]. The same algorithm for solving the same problem in a different setting is presented in [5].

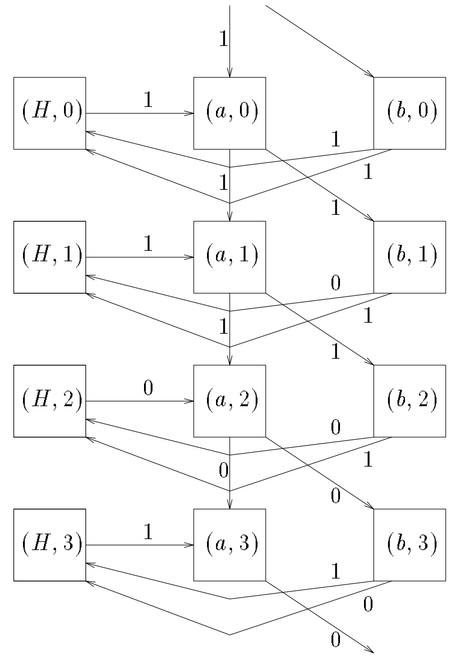


Figure 3.8: The labeling induced on the retimed communication graph by the labels of the space-time diagram shown in figure 3.7.

איור 3.8: התוויות המושרות על גרף התקשורת לאחר רה-תזמון על-ידי התוויות של דיאגרמת מקום-זמן המוצגת באיור 3.7.

3.6 Minimizing the maximum lag

In this section we present algorithm Rev-FEAS which finds a feasible retiming, which after normalization (i.e. reducing the lag of the host from the lag of all vertices) is a retiming with a minimum maximum lag. In Section 3.4, we defined the maximum lag of a normalized retiming as a heuristic cost function which measures the cost of finding a corresponding initialization. Algorithm Rev-FEAS is based on the FEAS algorithm of Leiserson and Saxe: First, we reverse the directions of the edges of the communication graph. Second, we compute the feasible retiming on the reversed graph using the FEAS algorithm. Third, we negate the retiming function, and obtain a feasible retiming with a minimum maximum lag. Consequently, one can find a feasible retiming which minimizes the maximum lag with only a linear time overhead over the FEAS algorithm. Since the correctness of our algorithm is based on the properties of the FEAS algorithm, we present a detailed review of the FEAS algorithm and its properties.

This section is organized as follows: In Subsection 3.6.1, we present algorithm MCP for computing the minimum clock period of a communication graph. This algorithm corrects a flaw in algorithm CP of Leiserson and Saxe [23]. In Subsection 3.6.2, we present a detailed review of the FEAS algorithm and the proof of its correctness. In Subsection 3.6.3, we present algorithm Rev-FEAS, and prove that it finds a feasible retiming with a minimum maximum lag.

3.6.1 Computing the minimum clock period

In this subsection we present algorithm MCP which computes the minimum clock period of a given communication graph. Algorithm MCP is a correction of algorithm CP presented by Leiserson and Saxe [23]. The flaw in algorithm CP comes from the fact that the host is treated as a regular vertex. Therefore, if the host is an interior vertex in all critical paths, then the computed minimum clock is too large. We amend this point by redefining the delay of a simple path as follows:

Definition 3.1 : Let $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} v_k$ denote a simple path. Assume that $d(host) = 0$, the delay, $d(p)$ is defined as follows:

$$d(p) \triangleq \begin{cases} \sum_{i=0}^{k} d(v_i) & \text{if } host \notin \{v_i\}_{i=1}^{k-1} \\ \max\{\sum_{i=0}^{\ell-1} d(v_i), \sum_{i=\ell+1}^{k} d(v_i)\} & \text{if } v_\ell = host \text{ and } 0 < \ell < k \end{cases}$$

As before, the minimum clock period of a circuit is defined as the maximum delay of a registerless path. Algorithm MCP, which is described in figure 3.9, computes the minimum clock period. In addition, algorithm MCP computes for each vertex, v, the maximum delay of a registerless path ending in v; this value is denoted by $\Delta(v)$. The computation of $\Delta_1(v)$ for all vertices in the algorithm is done by a time evaluation procedure on PERT graphs in $O(m+n)$ time [9, p. 138-139] (in the time evaluation procedure, the delay of an edge is the delay of the vertex from which it emanates). The computation of the vertices which are reachable from *host* in G_0 can be performed by a DFS on the graph G_0 starting from the host. Therefore, the complexity of running algorithm MCP is $O(n+m)$.

3.6.2 The FEAS algorithm

Leiserson and Saxe presented algorithm FEAS for finding a feasible retiming, if such a retiming exists. Algorithm FEAS is described in figure 3.10. After running Algorithm FEAS(Y, c), one should run MCP(Y_r); if one finds that $\Phi(Y_r) \leq c$ then FEAS found a feasible retiming, otherwise, no such retiming exists.

Leiserson and Saxe suggest a shrewd analysis of the FEAS algorithm. They prove the correctness of the FEAS algorithm (which has complexity $O(n \cdot m)$) by claiming that its execution is equivalent to an execution of a variation of Ford-Bellman's algorithm on the constraints graph, defined below. The complexity of Ford's algorithm on the constraints

```
MCP (Y = ⟨V, E, w, d⟩)
    Let E_0 ≜ {e ∈ E : w(e) = 0}
    Let G_0 be the sub-graph with vertex set V and edge set E_0.
    Let E_1 ≜ {(host, v) ∈ E}
    Let G_1 be the sub-graph with vertex set V and edge set E_0 − E_1.
    Compute Δ_1(v) ≜ max{d(p) : p is a path in G_1 which ends in v} for every vertex v.
    Compute the set of vertices which are reachable from host in G_0.
    For every v ∈ V do
        If v is reachable from host in G_0 then
            Δ(v) = max {Δ_1(v), Δ_1(host)}
        else
            Δ(v) = Δ_1(v)
    Φ(Y) = max_{v∈V} Δ(v)
```

Figure 3.9: Compute The Minimum Feasible Clock Period.
איור 3.9: חישוב מחזור השעון החוקי הקצר ביותר.

graph is $O(n^3)$. This analysis is distinctive because the correctness of the FEAS algorithm executed on the communication graph is proven by comparing it to Ford's algorithm on a different, although related, constraints graph. We give a detailed review of this proof, since we rely on the properties of the FEAS algorithm for the correctness of algorithm Rev-FEAS.

The following definition of a constraints graph is similar to the definitions given in [22, 26].

Definition 3.2: The constraints graph, $Y_c = \langle V, E_c, w_c \rangle$, of a communication graph, $Y = \langle V, E, w, d \rangle$, and a required clock period, c, is defined as follows:

$$\begin{aligned} E_c &\triangleq E \cup \{(u,v) : D(u,v) > c\} \\ w_c(u,v) &\triangleq \begin{cases} w(u,v) & \text{if } (u,v) \in E \\ W(u,v) - 1 & \text{if } D(u,v) > c \end{cases} \end{aligned}$$

If the definition of a weight is ambiguous for an edge, $(u,v) \in E_c$, then define $w_c(u,v) \triangleq \min\{w(u,v), W(u,v) - 1\}$.

The importance of the constraints graph is summarized in the following lemma ([26]):

Lemma 3.2: Let Y be a communication graph, and let c be a required clock period. The circuit Y has a retiming $\ell()$ such that $\Phi(Y_\ell) \leq c$ if and only if the graph Y_c does not have negative weighted cycles.

```
FEAS (Y = ⟨V, E, w, d⟩, c)
    initialize: for each vertex v ∈ V set r_0(v) = 0.
    For i = 1 to |V| − 1 do
        Compute the weight function, w_{r_{i−1}}(), of the retimed circuit Y_{r_{i−1}}.
        Run algorithm MCP(Y_{r_{i−1}}), to determine the values Δ(v) for every vertex v ∈ V.
        For every vertex v ∈ V do
            if Δ(v) > c then r_i(v) = r_{i−1}(v) + 1
            else r_i(v) = r_{i−1}(v)
    Return r() ≜ r_{|V|−1}().
```

Figure 3.10: The FEAS Algorithm.
איור 3.10: אלגוריתם $FEAS$.

In figure 3.11, we describe a version of the Ford-Bellman algorithm which is similar to the one of Saxe [27]. This algorithm is presented for the purpose of analyzing algorithm FEAS. In fact, its complexity is $O(n^3)$ compared with the $O(n \cdot m)$ complexity of the FEAS algorithm, because the constraints graph has $O(n^2)$ edges. Note, that this version of the Ford-Bellman algorithm differs from the standard one [9, p. 15-18], since the tentative distances are all initialized to zero (instead of ∞).

```
FORD (Y_c = ⟨V, E_c, w_c⟩)
    initialize: for each vertex v ∈ V set t_0(v) = 0.
    For i = 1 to |V| − 1 do
        For every vertex v ∈ V do t_i(v) = t_{i−1}(v)
        For every edge (u, v) ∈ E_c do
            If t_i(v) > t_{i−1}(u) + w_c(u, v) then t_i(v) = t_{i−1}(u) + w_c(u, v)
    Return t() ≜ t_{|V|−1}().
```

Figure 3.11: The Version of The Ford-Bellman Algorithm.
איור 3.11: הגרסה לאלגוריתם פורד-בלמן.

The following claim shows that executions of FEAS(Y, c) and FORD(Y_c) satisfy $r() = -t()$, and that $r()$ is a feasible retiming, if one exists. Notation in the claim is based on the notation used in the descriptions of these algorithms.

Claim 3.2: Let $Y = \langle V, E, w, d\rangle$ be a communication graph, and let c be a required clock period. Let Y_c be the constraints graph corresponding to Y and c.

1. $r_i()$ is a legal retiming function, for $0 \leq i \leq |V| - 1$.

2. Let Y_i denote the circuit obtained by retiming Y by the function $r_i()$, and let $w_i()$ denote the weight function of edges in Y_i. Define $W_i(u,v)$ as the weight of the shortest path in Y_i from u to v, and define $\Delta_i(v)$ as the value $\Delta(v)$ computed by MCP(Y_i). When we refer to the values, $t_i(v)$, computed by algorithm FORD(Y_c), we refer to the values of $t_i(v)$ at the *end* of the i'th iteration of the loop in algorithm FORD(Y_c). Define,

$$\begin{aligned} R_i &\triangleq \{v \in V : \Delta_{i-1}(v) > c\} \\ T_i &\triangleq \{(u,v) \in E_c : t_{i-1}(v) > t_{i-1}(u) + w_c(u,v)\} \end{aligned}$$

If the functions $r_{i-1}(v) = -t_{i-1}(v)$ for every vertex $v \in V$, then

$$R_i = \{v \in V : \exists u \in V \text{ such that } (u,v) \in T_i\}.$$

3. The functions $r_i()$ and $t_i()$ satisfy $r_i(v) = -t_i(v)$, for every vertex $v \in V$ and for every i such that $0 \leq i \leq |V| - 1$.

4. If Y_c does not have negative weighted cycles, then for every vertex, v, at the end of FORD(Y_c), the value of $t(v)$ equals the weight of a minimum weight path in Y_c which ends in v. Formally,

$$t(v) = \min\{w_c(p) : u \overset{p}{\rightsquigarrow} v \text{ in the graph } Y_c\}. \tag{3.1}$$

5. Retiming circuit, Y, by the retiming function, $r()$, returned by FEAS(Y, c) yields a circuit with a clock period not greater than c, if there exists a retiming ℓ such that $\Phi(Y_\ell) \leq c$.

Proof:

Proof of part 1: The proof is by induction on i. The induction's basis holds because $r_0()$ is a constant function. Suppose $r_{i-1}()$ is a retiming function, we show that $r_i()$ is also a retiming function: We need to show that $w(u,v) + r_i(v) - r_i(u) \geq 0$ for every $(u,v) \in E$. If $r_i(v) - r_i(u) \geq r_{i-1}(v) - r_{i-1}(u)$, then the required inequality holds because of the induction's hypothesis. Otherwise, since the values of $r(v)$ and $r(u)$ do not decrease, it must be the case that $r_i(u) = r_{i-1}(u) + 1$ and $r_i(v) = r_{i-1}(v)$. This implies that in $Y_{r_{i-1}}$ we have $\Delta(v) \leq c$ and $\Delta(u) > c$, which implies that $w(u,v) + r_{i-1}(v) - r_{i-1}(u) > 0$, and part 1 of the claim follows.

Proof of part 2: First we show that $R_i \subseteq \{v : \exists u \text{ such that } (u,v) \in T_i\}$:

$$\begin{aligned} v \in R_i \quad &\Rightarrow \quad \Delta_{i-1}(v) > c \\ &\Rightarrow \quad \exists u \in V : W_{i-1}(u,v) = 0 \text{ and } D(u,v) > c \\ \text{But} \quad & \quad\;\; W_{i-1}(u,v) = W(u,v) - r_{i-1}(u) + r_{i-1}(v), \\ &\Rightarrow \quad r_{i-1}(v) = r_{i-1}(u) - W(u,v) \\ &\Rightarrow \quad r_{i-1}(v) < r_{i-1}(u) - (W(u,v) - 1) \end{aligned}$$

$$\begin{aligned}
D(u,v) > c &\Rightarrow (u,v) \in E_c \text{ with } w_c(u,v) \le W(u,v) - 1 \\
&\Rightarrow r_{i-1}(v) < r_{i-1}(u) - w_c(u,v) \\
t_{i-1}() = -r_{i-1}() &\Rightarrow t_{i-1}(v) > t_{i-1}(u) + w_c(u,v) \\
&\Rightarrow \exists u \text{ such that } (u,v) \in T_i
\end{aligned}$$

We now show that $\{v : \exists u \text{ such that } (u,v) \in T_i\} \subseteq R_i$: Suppose that $(u,v) \in T_i$. If $(u,v) \in E$ then since $t_{i-1}() = -r_{i-1}()$ we conclude that $r_{i-1}()$ is not a legal retiming, which contradicts part 1. Therefore, $w_c(u,v) = W(u,v) - 1$ and $D(u,v) > c$. Let p denote a path in Y from u to v with $w(p) = W(u,v)$ and $d(p) > c$. Since $(u,v) \in T_i$, we have $t_{i-1}(v) - t_{i-1}(u) > W(u,v) - 1$. This implies that $w_{i-1}(p) = w(p) + t_{i-1}(u) - t_{i-1}(v) < 1$. Since $r_{i-1}()$ is a legal retiming, $w_{i-1}(p)$ is non-negative, and therefore, $w_{i-1}(p) = 0$. Since $D(u,v) > c$, we have $\Delta_{i-1}(v) > c$. Therefore, $v \in R_i$, and part 2 follows.

Proof of part 3: The proof is by induction on i. The induction basis for $i = 0$ is trivial because both functions are constant zero. Suppose that $r_{i-1}(v) = -t_{i-1}(v)$ for every vertex $v \in V$. If $v \notin R_i$ then $r_i(v) = r_{i-1}(v)$ and by part 2 we have $t_i(v) = t_{i-1}(v)$, and we are done. If $v \in R_i$, then $r_i(v) = r_{i-1}(v) + 1$. By part 2 we know that there exists a vertex u such that $(u,v) \in T_i$. In the proof of part 2, we showed that $(u,v) \in T_i$ implies that $W(u,v) + t_{i-1}(u) - t_{i-1}(v) = 0$, and that $t_i(v) = t_{i-1}(u) + W(u,v) - 1$. The last two equations imply that $t_i(v) = t_{i-1}(v) - 1$, and part 3 follows. [1]

Proof of part 4: This version of Ford's algorithm differs from the standard version [9, p. 15-18] by the fact that all tentative distances are initialized to zero instead of (a) initializing the tentative distance of the source to zero, and (b) initializing all other tentative distances to infinity. We show that this variation causes the algorithm to compute the shortest path entering each vertex, instead of computing the shortest path from the source to each vertex. Note, that the lack of negative cycles implies that shortest paths are well defined. The proof follows [11].

This can be shown by considering the following modification of Ford's algorithm (denoted by FORD-0): (1) Add a new vertex (the source) s, connect s to all the vertices with edges of weight 0. (2) Initialize the tentative distances: $t(s) = 0$ and $t(v) = \infty$ for every $v \in V$. 3) Execute $|V|$ relaxation iterations. Since Ford's algorithm computes shortest paths from the source s to all vertices, we know that $t(v)$ equals the shortest path from s to v. Since s is connected with every vertex with a zero weight edge, equation 3.1 is trivially satisfied for algorithm FORD-0. However, after the first relaxation iteration of FORD-0, if the edges outgoing s are relaxed last, then the tentative distance function $t()$ is a constant zero, and the edges outgoing s are not relaxed again. Therefore, the presented version of FORD's algorithm and algorithm FORD-0 have identical outputs, and equation 3.1 follows.

Proof of part 5: If there exists a retiming ℓ such that $\Phi(Y_\ell) \le c$, then by lemma 3.2, the graph Y_c does not have negative cycles, and therefore, by part 4, the value of $t(v)$

[1] Note, that from part 3 it follows that $t_i(v)$ does not depend on the order in which the edges of E_c are visited in each iteration of algorithm FORD(Y_c).

equals the shortest path entering v. We show that $t()$ satisfies $t(u) - t(v) + w_c(u, v) \geq 0$ for every edge $(u, v) \in E_c$. Let p_u and p_v denote the shortest paths entering vertices u and v respectively in the graph Y_c, so that $t(u) = w_c(p_u)$ and $t(v) = w_c(p_v)$. Equation 3.1 implies that $w_c(p_v) \leq w_c(p_u) + w_c(u, v)$, and therefore, $t(u) - t(v) + w_c(u, v) \geq 0$. According to part 3, $r(v) = -t(v)$ for every vertex $v \in V$. Therefore, we have $w_c(u, v) - r(u) + r(v) \geq 0$ for every edge $(u, v) \in E_c$. Satisfying all these constraints implies that $\Phi(Y_r) \leq c$, and part 5 follows. □

3.6.3 The Rev-FEAS algorithm

In this section we present algorithm Rev-FEAS which finds a feasible retiming. After normalization, this retiming has a minimum maximum lag. We end this section with an example which demonstrates that there is a linear gap between the maximum lag found by algorithm FEAS and the maximum lag found by algorithm Rev-FEAS.

Let $Y = \langle V, E, w, d \rangle$ denote a communication graph, and c a required clock period. Algorithm Rev-FEAS is described in figure 3.12, and it consists of 3 phases: (a) Construct a communication graph whose edges are the reversed edges of the given communication graph. (b) Execute FEAS on the reversed communication graph with the required clock period. (c) Negate the returned retiming function.

Claim 3.3 summarizes the properties of algorithm Rev-FEAS. The first part shows that Rev-FEAS finds a feasible solution, if one exists. The second part shows that any normalized feasible retiming has a maximum lag which is not smaller than the maximum lag of the retiming found by Rev-FEAS after normalization.

```
Rev-FEAS (Y, c)
    Let Y'(V, E', w', d) be the reversed circuit defined by:
        E' ≜ {(v, u) : (u, v) ∈ E}
        w'(v, u) ≜ w(u, v)

    Let r'() denote the retiming function returned by FEAS(Y', c)
    Return the function lag() defined by lag(v) ≜ −r'(v)
```

Figure 3.12: The Rev-FEAS Algorithm.
איור 3.12: אלגוריתם $Rev - FEAS$.

Claim 3.3: Let $Y = \langle V, E, w, d \rangle$ be a communication graph and c a required clock period. Let $lag()$ denote the retiming computed by algorithm Rev-FEAS(Y, c). Then,

1. Algorithm Rev-FEAS(Y, c) finds a retiming $lag()$ such that $\Phi(Y_{lag}) \leq c$, if such a retiming exists.

2. If the all vertices are reachable from the $host$ in the graph (V, E), and if $\Phi(Y_{lag}) \leq c$, then for every retiming $lag'()$ for which $\Phi(Y_{lag'}) \leq c$ the following holds:

$$\max_{v \in V}\{lag(v) - lag(host)\} \leq \max_{v \in V}\{lag'(v) - lag'(host)\}$$

Proof: Part 1 follows from three observations: (a) If $r'()$ is a legal retiming of Y', then $-r'()$ is a legal retiming of Y, and $\Phi(Y'_{r'}) = \Phi(Y_{-r'})$. (b) FEAS$(Y', c)$ returns a retiming $r'()$ for which $\Phi(Y'_{r'}) \leq c$ if there exists such a retiming. (c) There exists a retiming $r'()$ such that $\Phi(Y'_{r'}) \leq c$ if and only if there exists a retiming $r()$ such that $\Phi(Y_r) \leq c$.

Proof of Part 2: By lemma 3.2, since $\Phi(Y_{lag}) \leq c$, the constraints graph of Y and c does not have negative weighted cycles. Since Y' is the reversed graph of Y, it follows that the constraints graph of Y' and c, denoted by Y'_c, also lacks negative cycles. In claim 3.2 we showed that under this assumption, $-r'(v) = \min\{w'_c(p') : u \overset{p'}{\leadsto} v \text{ in } Y'_c\}$, where $w'_c()$ is the weight function of Y'_c. By reversing edges back again we obtain:

$$lag(v) = \min\{w_c(p) : v \overset{p}{\leadsto} u \text{ in } Y_c\} \tag{3.2}$$

Note, that $lag(v)$ equals the minimum weight of a path *leaving* v. Since lag' is also a retiming of Y which satisfies $\Phi(lag') \leq c$, we have,

$$w_c(u, v) - lag'(u) + lag'(v) \geq 0 \text{ for every edge } (u, v) \in E_c \tag{3.3}$$

Consider a path $host \overset{p}{\leadsto} v$ in Y_c. Summing up the inequalities of equation 3.3 corresponding to the edges along the path p, we obtain $w_c(p) - lag'(host) + lag'(v) \geq 0$. We can choose the shortest path from $host$ to v, and obtain $lag'(v) - lag'(host) \geq -\min\{w_c(p) : host \overset{p}{\leadsto} v \text{ in } Y_c\}$. From equation 3.2 it follows that $lag'(v) - lag'(host) \geq -lag(host)$. By maximizing over all vertices we obtain:

$$\max_{v \in V}\{lag'(v) - lag'(host)\} \geq -lag(host)$$

From equation 3.2 it follows that $lag(v) \leq 0$ for every vertex $v \in V$, since an empty path has zero weight. Therefore, $\max_{v \in V}\{lag'(v) - lag'(host)\} \geq \max_{v \in V}\{lag(v) - lag(host)\}$, and part 2 follows. □

Example 3.7: Figure 3.13 depicts a communication graph, Y, which consists of a ring of $n + 1$ vertices. The delay of each vertex v_i is 1, and the delay of the *host* is zero. All edges have weight zero, except for the edges which are incident to the *host*, which have weight $n/2 - 1$ (assume that n and $n/2$ are even). Suppose we wish to retime the communication graph so that its clock period equals 2.

Figure 3.14 depicts the communication graph obtained by retiming Y according to the retiming function computed by FEAS($Y, 2$). Lags of the vertices are shown next to the vertices, and the weight of the edges after retiming are shown on the edges. All lags are non-negative and range from zero to $n/2 - 1$.

Figure 3.15 depicts the communication graph obtained by retiming Y according to the retiming function computed by Rev-FEAS($Y, 2$). Lags of the vertices are shown next to the vertices, and the weight of the edges after retiming are shown on the edges. All lags are non-positive and range from $-n/2 + 1$ to zero.

The maximum lag of the retiming found by FEAS($Y, 2$) is $n/2-1$, whereas the maximum lag of the retiming found by Rev-FEAS($Y, 2$) is zero.

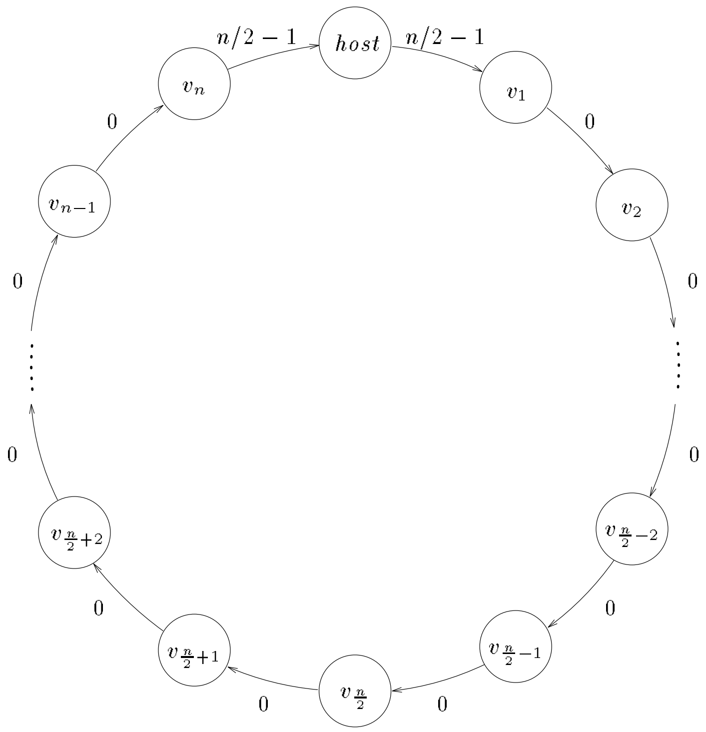


Figure 3.13: The ring before retiming.
איור 3.13: הטבעת לפני רה-תזמון.

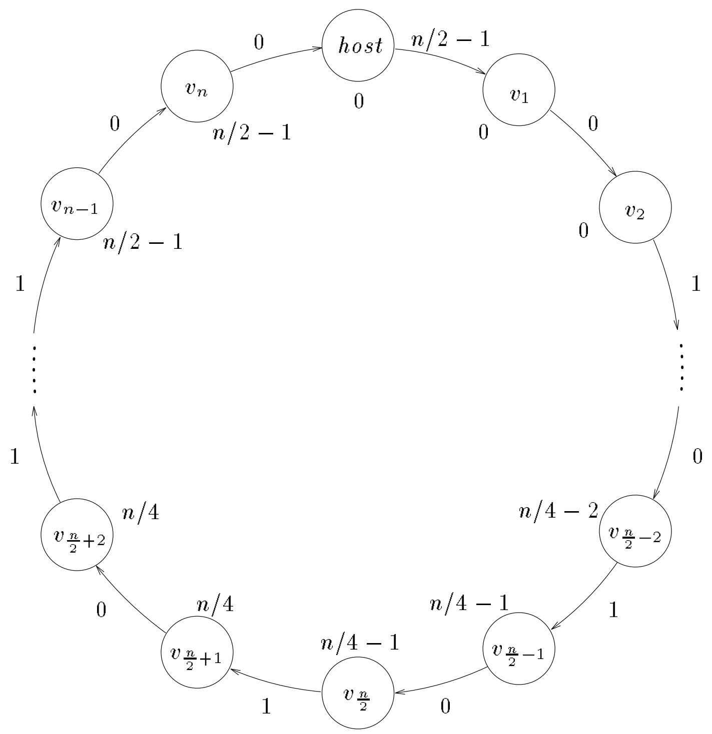


Figure 3.14: The ring after retiming with the retiming function found by FEAS(Y, 2).
איור 3.14: הטבעת לאחר רה-תזמון על-ידי הפונקציה שנמצאה על-ידי $FEAS(Y, 2)$.

3.7 Minimizing the sum of positive lags

In this section we present a reduction which reduces the problem of finding a normalized feasible retiming with a minimum weighted sum of positive lags to a problem of finding a feasible retiming with a minimum total number of registers (also called a state-minimization problem). The state minimization problem can be efficiently solved using a min-cost flow algorithm, using the primal-dual technique suggested by [5, 23]. In section 3.4 we defined the (weighted) sum of positive lags of a feasible retiming as a heuristic cost function which measures the cost of finding a corresponding initialization. Therefore, this section presents a method of finding a feasible retiming which is optimal with respect to this cost function.

The reduction is done in two steps. In the first step, the problem of finding a feasible retiming with a minimum weighted sum of positive lags is written as a linear program. In the second step, the obtained linear program is reduced to a problem of finding a feasible retiming with a minimum total number of registers. In practice, solution of these problems does not involve linear programming, and linear programming is used solely for analysis

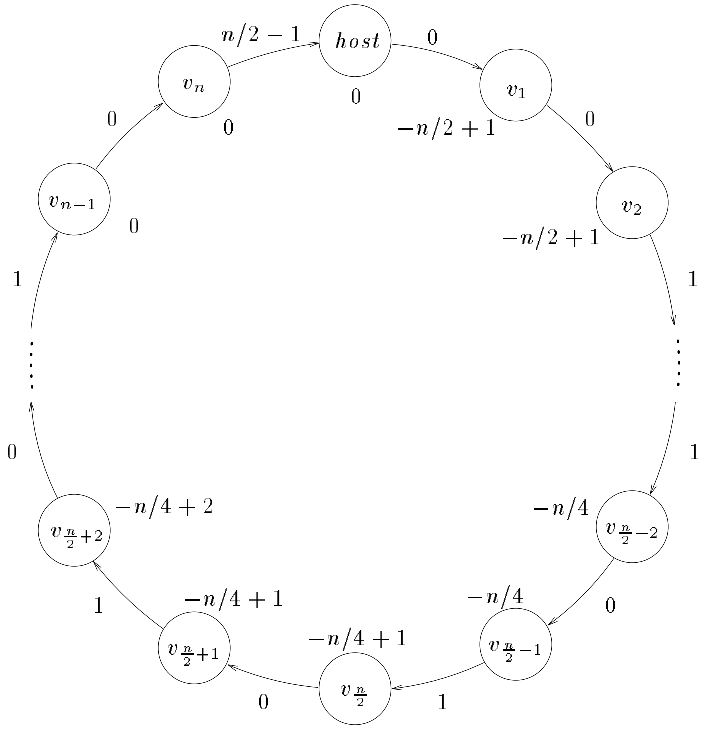


Figure 3.15: The ring after retiming with the retiming function found by Rev-FEAS$(Y, 2)$.

איור 3.15: הטבעת לאחר רה-תזמון על-ידי הפונקציה שנמצאה על-ידי $Rev - FEAS(Y, 2)$.

purposes.

3.7.1 Writing the problem as a linear program

The problem of finding a feasible retiming with a minimum weighted sum of positive lags is defined as follows:

Problem Normalized Feasible Retiming With Minimum Sum of Positive Lags

input: A communication graph $Y = \langle V, E, w, d \rangle$, a non-negative price function $p()$ defined on the vertices, and a required clock period c. (Assume that a feasible retiming exists).

Goal: Find a normalized feasible retiming which minimizes the cost $\sum_{\{v \in V : r(v) > 0\}} p(v) \cdot r(v)$.

First, we rewrite the problem as an optimization problem over unknowns $\{r(v)\}_{v\in V}$ as follows:

$$\text{minimize} \sum_{\{v\in V: r(v)>0\}} p(v)\cdot r(v) \quad (\forall v\in V: p(v)\geq 0)$$

subject to the constraints,

$$\begin{array}{rl} & r(host)=0 \\ \forall (u,v)\in E & w(u,v)-r(u)+r(v)\geq 0 \\ \forall u,v \text{ with } D(u,v)>c & W(u,v)-r(u)+r(v)\geq 1 \end{array}$$

The above optimization problem is not a linear program because the cost function is not a linear function. We add unknowns $\{z(v)\}_{v\in V}$, and constrains on the new variables to obtain the following linear problem:

The Corresponding Linear Program

$$\text{minimize} \sum_{v\in V} p(v)\cdot z(v) \quad (\forall v\in V: p(v)\geq 0)$$

subject to the constraints,

$$\begin{array}{rl} & r(host)=0 \\ \forall (u,v)\in E & w(u,v)-r(u)+r(v)\geq 0 \\ \forall u,v\in V \text{ with } D(u,v)>c & W(u,v)-r(u)+r(v)\geq 1 \\ \forall v\in V & z(v)\geq r(v) \\ \forall v\in V & z(v)\geq 0 \end{array}$$

Claim 3.4 shows that optimal solutions to the problem of finding a normalized feasible retiming with a minimum sum of positive lags and the corresponding linear program have equal costs. Moreover, it is easy to translate an optimal solution of one problem to an optimal solution of the other problem. The proof is of claim 3.4 is straightforward.

Claim 3.4: The problem of finding a normalized feasible retiming with a minimum weighted sum of positive lags and the corresponding linear program have identical optimal costs. In particular, if $r()$ is a feasible solution of the first problem, then $r()$ and $z(v)\triangleq\max\{0,r(v)\}$ constitute a feasible solution of the second problem with an identical cost. If $r()$ and $z()$ constitute a feasible solution of the second problem, then $r()$ is a feasible solution of the first problem with an identical cost.

3.7.2 Reducing the linear program to a state minimization problem

Given a linear program of the form described above, we construct the communication graph $\tilde{Y} = \langle \tilde{V}, \tilde{E}, \tilde{w}, \tilde{d} \rangle$, defined by,

$$\begin{aligned}
\tilde{V} &= V \cup \{z_v : v \in V\} \\
\tilde{E} &= E \cup \{(v, z_v)\}_{v \in V} \cup \{(host, z_v)\}_{v \in V} \\
\tilde{w}(x, y) &= \begin{cases} w(x, y) & \text{if } (x, y) \in E \\ 0 & \text{otherwise} \end{cases} \\
\tilde{d}(x) &= \begin{cases} d(x) & \text{if } x \in V \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Define real edge breadths, $\beta : \tilde{E} \to \mathsf{R}$, by

$$\beta(x, y) = \begin{cases} p(v) & \text{if } (x, y) = (host, z_v) \\ 0 & \text{otherwise} \end{cases}$$

The instance of the state-minimization problem obtained is:

Instance of State Minimization Problem

input: A communication graph $\tilde{Y} = \langle \tilde{V}, \tilde{E}, \tilde{w}, \tilde{d} \rangle$ with edge breadths $\beta()$, and a required clock period c. (Assume that a feasible retiming exists).

Goal: Find a feasible retiming $\tilde{r}()$ which minimizes the cost $\sum_{\{e \in \tilde{E}\}} \beta(e) \cdot \tilde{w}_{\tilde{r}}(e)$.

Claim 3.5 shows how to translate a feasible retiming of $\tilde{Y}$ to a feasible solution of the corresponding linear program. The costs of both feasible solutions are equal.

Claim 3.5: If $\tilde{r}()$ is a normalized feasible retiming of $\tilde{Y}$, then $r(v) \triangleq \tilde{r}(v)$ and $z(v) \triangleq \tilde{r}(z_v)$ is a feasible solution of the linear program. Moreover, both solutions have the same cost.

Proof: Since $\tilde{r}()$ is a normalized feasible retiming, we only need to show that: (a) $z(v) \geq r(v)$, for every $v \in V$ and (b) $z(v) \geq 0$, for every $v \in V$. The retimed weight of the edge (v, z_v) is non-negative, meaning that $w(v, z_v) - \tilde{r}(v) + \tilde{r}(z_v) \geq 0$. Since $w(v, z_v) = 0$, it follows that (a) holds. Similarly, the retimed weight of the edge $(host, z_v)$ is non-negative, meaning that $w(host, z_v) - \tilde{r}(host) + \tilde{r}(z_v) \geq 0$. Since $w(host, z_v) = 0$, and since $\tilde{r}(host) = 0$, it follows that (b) holds.

The retimed weight of the edge $(host, z_v)$ equals $\tilde{r}(z_v)$, which equals $z(v)$. The breadth of the edges $(host, z_v)$ equals $p(v)$, and all other edges have breadth which equals zero. Therefore, both solutions have equal costs, and the claim follows. □

Claim 3.6 shows how to translate a feasible solution of the linear program to a feasible retiming of $\tilde{Y}$. The costs of both feasible solutions are equal.

Claim 3.6: If $r()$ and $z()$ constitute a feasible solution of the linear program, then $\tilde{r}()$ defined by,

$$\tilde{r}(x) \triangleq \begin{cases} r(v) & \text{if } x = v \in V \\ z(v) & \text{if } x = z_v \end{cases}$$

is a feasible retiming of $\tilde{Y}$. Moreover, both solutions have equal costs.

Proof: Note, that $r()$ is a feasible retiming of Y with respect to clock period c. The critical paths of $\tilde{Y}_{\tilde{r}}$ are identical to those of Y_r, since all the nodes $\{z_v\}_{v \in V}$ are sinks with zero delay. Therefore, if $\Phi(Y_r) \leq c$, then $\Phi(\tilde{Y}_{\tilde{r}}) \leq c$.

The cost of the solution $r()$ and $z()$ is $\sum_v p(v) \cdot z(v)$. The cost of the solution $\tilde{r}()$ is $\sum_v \beta(host, z_v) \cdot \tilde{w}_{\tilde{r}}(host, z_v)$. Since $\beta(host, z_v) = p(v)$, and $\tilde{w}_{\tilde{r}}(host, z_v) = z(v)$, for every vertex v, these costs are equal, and the claim follows. □

Claims 3.5 and 3.6 show that optimal solutions of the linear program and the state minimization have equal costs. Moreover, it is easy to translate an optimal solution of one problem to an optimal solution of the other problem.

Chapter 4

A Real-Time Systolic Integer Multiplier

4.1 Introduction

In this chapter we describe the construction of two practical, real-time, bit-serial, systolic integer multipliers. One of the multipliers is pipelined and the other is not. Our non-pipelined multiplier requires roughly half the hardware as compared to Atrubin's multiplier, which is the best systolic integer multiplier known to-date. Our multipliers receive as input two n-bit multiplicands, and output serially the $2n$ bits of the product. One of the inputs, $\mathbf{a}$, is input one bit at a time, and the other multiplicand, $\mathbf{b}$, is input two bits at a time. The i'th bit of the product is output one clock cycle after the i'th bit of multiplicand $\mathbf{a}$ is input. Our multipliers are constructed from a linear array of identical cells, which do not depend on the length of the multiplicands. The number of cells required to multiply n-bit integers is $n/4$, and therefore, our circuits require linear area. Wires connect only neighboring cells, and all wires have at least one register. Therefore, the clock period depends only the cells of the linear array, and do not depend on the length of the multiplicands.

The starting point of our design is a non-systolic multiplier known as a serial/parallel multiplier [30, 6]. We improve this circuit step by step using systematic methodologies until our design is obtained. The methodologies used are: broadcast elimination using retiming, replacing parallel inputs with broadcast and adding a simplified version of the circuit in order to enable pipelining. Our method of replacing parallel inputs, which input constant values, with broadcast is new, and can be used for decreasing the number of primary inputs in other circuits as well. Our method of enabling pipelining increases the throughput by a factor of two, yet the increase in the hardware is by a factor less than two. The same method can be applied to Atrubin's multiplier.

The organization of this chapter is as follows: In Section 4.2 we present the serial/parallel multiplier and discuss its disadvantages. In Section 4.3 we show how to eliminate broadcast using retiming. In Section 4.4 we show how to replace parallel inputs with broadcast. In Section 4.5 we present our non-pipelined multiplier. In Section 4.6 we present a pipelined version of the serial/parallel multiplier and discuss how our pipelined multiplier is obtained. In Section 4.8 we summarize the comparison of our multiplier and Atrubin's multiplier.

4.2 Serial/Parallel multiplier

In this section we present a multiplication circuit known as a serial/parallel multiplier [30, 6]. This multiplier is a basic design from which we finally obtain a systolic multiplier. The serial/parallel multiplier is a synchronous circuit consisting of a linear array of identical combinational elements and of registers. The serial/parallel multiplier is designed to multiply two n-bit positive integers represented by the binary vectors $\mathbf{a} = a_{n-1} \cdots a_0$ and $\mathbf{b} = b_{n-1} \cdots b_0$. The communication graph of a serial/parallel multiplier of 4-bit integers is depicted in figure 4.1. The boxes denote combinational units called *summand-adders* the structure and functionality of which is depicted in figure 4.2.

The circuit has $n + 2$ primary inputs and one primary output. The bits of the multiplicand $\mathbf{a}$ are input one bit per clock cycle, least significant bit first, through the primary input a_{in}. After the bits of $\mathbf{a}$ have been input, zeros are input through a_{in} for n clock cycles. The bits of the multiplicand $\mathbf{b}$ are input in parallel; the i'th bit, b_i, is input through the primary input b_i. The primary input *Start* is used to mark the beginning of computations: A '1' is input by *Start* simultaneously with the least significant bit of $\mathbf{a}$. During all other clock cycles a '0' is input by *Start*. The product is output through the primary output p_{out}. The bits of the product are output one bit per clock cycle, least significant bit first, starting from one clock cycle after the computation begins. Since the product is $2n$-bits long, it is necessary to wait n clock cycles after the most significant bit of $\mathbf{a}$ has been input before a new computation can be started.

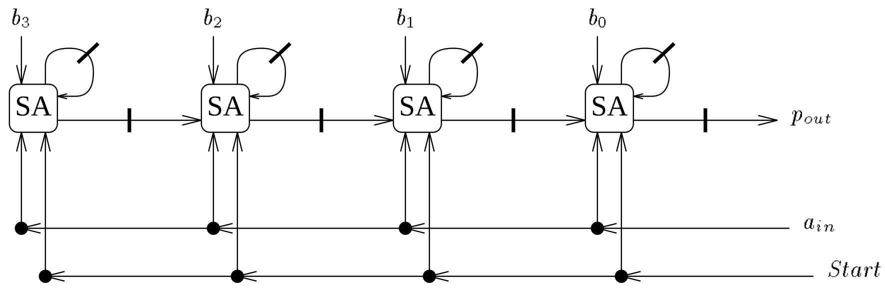
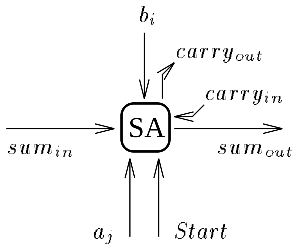


Figure 4.1: Serial/parallel multiplier.
איור 4.1: כופל סדרתי/מקבילי.



if *Start* = 0 then
  begin
  $sum_{out} := \mathrm{XOR}(a_j \cdot b_i, sum_{in}, carry_{in})$
  $carry_{out} := \begin{cases} 0 & \text{if } a_j \cdot b_i + sum_{in} + carry_{in} < 2 \\ 1 & \text{otherwise} \end{cases}$
  end
else
  begin
  $sum_{out} := a_j \cdot b_i$
  $carry_{out} := 0$
  end

Figure 4.2: Summand-adder: inputs/outputs and functionality.
איור 4.2: מחבר: קלטים/פלטים ופונקציונליות.

The serial/parallel multiplier is a bit-serial, real-time modular design which can be easily implemented in VLSI. It requires n summand-adders for multiplying n-bit positive integers. However, it has three disadvantages:

1. The bits of the operand **a** and the *Start* signal are sent along registerless paths to all the summand-adders in the circuit. The disadvantage of this feature, called 'broadcast' in [21, 13], is that it sets a lower bound on the clock period which is linear with n.

2. The bits of the operand **b** are input in parallel to all the summand-adders in the circuit. The disadvantage of this feature is that the number of primary inputs required for the multiplication circuit is linear with n.

3. The elapsed time between successive computations is $2n$ clock cycles although the inputs are n bits long. In section 4.6 we show a pipelined version of the serial/parallel multiplier in which the elapsed time between computations is n clock cycles that requires less than twice the hardware of a single non-pipelined serial/parallel multiplier.

4.3 Eliminating the broadcast by retiming

In this section we show how to eliminate the broadcast from the serial/parallel multiplier. In Section 1.2.4 we described two methods for eliminating broadcast which use retiming. We choose to use the tiling technique of Even and Litman [13, Section 5].

Figure 4.3 depicts the serial/parallel multiplier after eliminating the broadcast. The retimed circuit does not have a broadcast mechanism but still has the disadvantage of too many primary inputs. We deal with this in the next section.

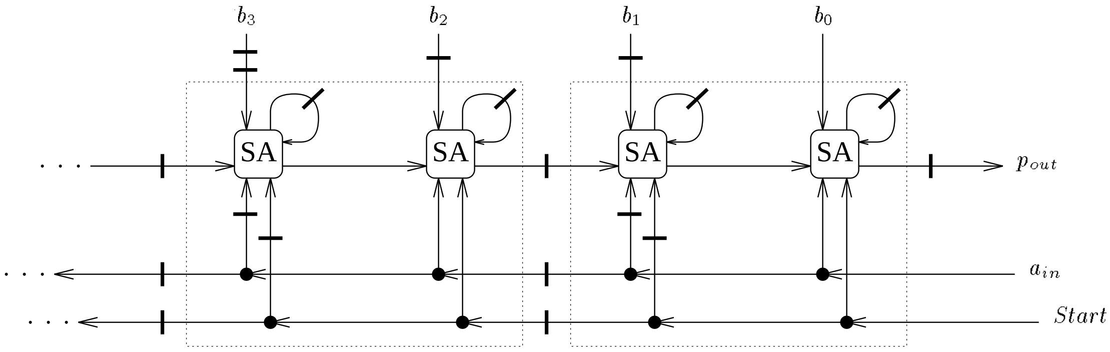


Figure 4.3: Retimed circuit - broadcast of multiplicand **a** and of *Start* signals eliminated.
איור 4.3: ביטול כריזה על-ידי רה-תזמון.

4.4 Replacing parallel inputs with broadcast

In this section we show how one can replace the parallel inputs of the retimed linear array with a broadcast mechanism. This change involves a modification of the topology of the circuit and it also involves minor adjustments of the functionality of the circuit. This new transformation can be helpful when one wishes to eliminate parallel inputs, which input constant values, from a linear array. We apply this replacement to the retimed parallel/serial multiplier depicted in figure 4.3, and obtain a circuit which has only four primary inputs but which has a broadcast mechanism. The broadcast can then be eliminated by using the transformation presented in the previous section.

Consider the retimed linear array depicted in figure 4.3. Figure 4.4 depicts the communication graph of the retimed linear array; the joined elements are shown as boxes, and the hosts as ellipses. Suppose that the top row of input hosts, named $H_0, H_1, ...$, input constant values throughout the computation (which lasts $2n$ clock cycles). We show how to replace these input hosts with two hosts and two broadcast nets. Let b_i denote the value input by H_i during the computation. Notice, that due to the increasing amount of registers along the edges which emanate from the hosts $H_0, H_1, ...$, these hosts input at most two new values to the linear array at every clock cycle. In figure 4.5, we replace these hosts with two hosts: H_{odd} and H_{even}, which are connected to two broadcast nets. The values $b_0, b_2, ...$ are input by host H_{even} starting from the first clock cycle of the computation, and the values $b_1, b_3, ...$ are input by the host H_{odd} starting from the second clock cycle of the computation. Each host inputs one value at a time, and therefore, these input values reach their destinations in the correct clock cycle.

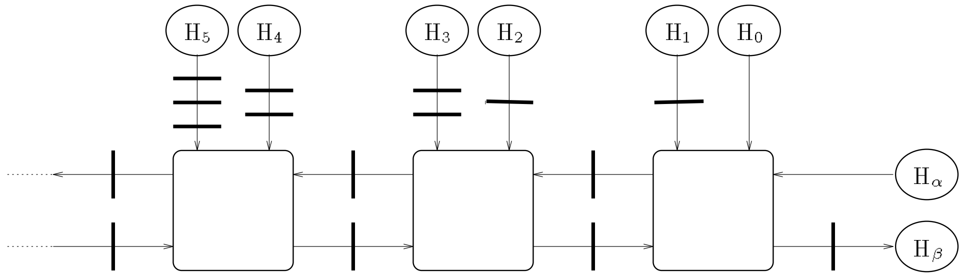


Figure 4.4: The communication graph of the retimed linear array.
איור 4.4: גרף התקשורת של המערך הליניארי אחרי רה-תזמון.

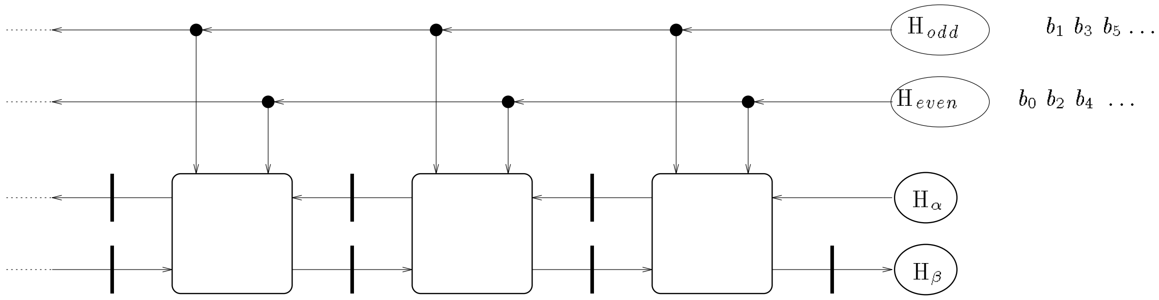
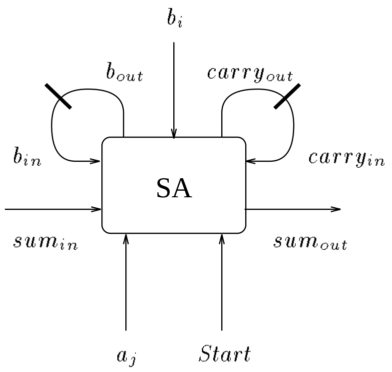


Figure 4.5: A linear array with two broadcast nets instead of parallel inputs.
איור 4.5: החלפת כניסות מקביליות בשתי רשתות כריזה.

To insure proper functioning of the array, we slightly modify the circuit. The reason for this is that the functional elements may sample the values input by the parallel inputs during the whole computation rather than only during the first time they engage in the computation. We overcome this problem by using an initialization signal and a register for each input (which used to be a parallel input): The initialization signal notifies the functional element when the value input is valid, and at that clock cycle the input is sampled and stored by the register for future clock cycles. It is convenient to assume that the original linear array with broadcast had an initialization signal, which was broadcast simultaneously to all the linear array. This initialization signal is no longer broadcast in the retimed circuit, but it arrives at each element “at the right time”.

We apply this replacement to the serial/parallel multiplier of figure 4.3, and obtain a circuit which has only four primary inputs. The multiplier circuit has already incorporated an initialization signal, *Start*, so we only have to add to each summand-adder a register which holds the sampled input. The modified summand-adder with the register for holding the sampled value of the input b_i is illustrated and described in figure 4.6. We hereafter refer to the modified summand-adder as the summand-adder.



if $Start = 0$ then
    begin
    $sum_{out} := \mathrm{XOR}(a_j \cdot b_{in}, sum_{in}, carry_{in})$
    $carry_{out} := \begin{cases} 0 & \text{if } a_j \cdot b_{in} + sum_{in} + carry_{in} < 2 \\ 1 & \text{otherwise} \end{cases}$
    $b_{out} := b_{in}$
    end
else
    begin
    $sum_{out} := a_j \cdot b_i$
    $carry_{out} := 0$
    $b_{out} := b_i$
    end

Figure 4.6: Modified summand-adder: inputs/outputs and functionality.
איור 4.6: שינוי במחבר: קלטים/פלטים ופונקציונליות.

4.5 A non-pipelined version of our multiplier

The transformation described in section 4.4 reduced the number of primary inputs to a constant number, regardless of the length of the numbers multiplied. However, this transformation replaces parallel inputs with a broadcast mechanism. Our final circuit is obtained by eliminating the broadcast nets by using again the tiling technique of Even and Litman, as done in Section 4.3.

Figure 4.7 depicts a systolic, real-time, bit-serial multiplier, which is obtained by retiming the multiplier obtained in the previous section. The filled boxes denote summand adders, the dotted boxes which contain two summand adders denote the "joined-elements" of figure 4.3, and the dotted boxes which contain four summand adders denote the new cells one obtains after the last retiming.

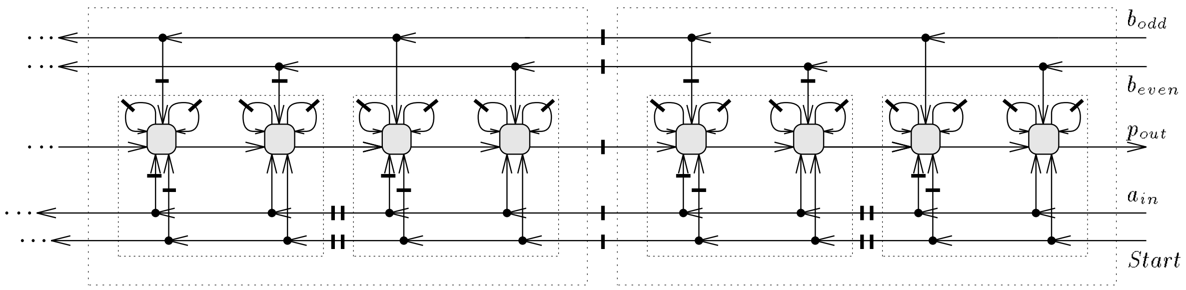


Figure 4.7: A non-pipelined version of our multiplier.
איור 4.7: גרסא בלי *pipelining* של הכופל.

4.6 A pipelined version of our multiplier

The non-pipelined version of our multiplier deals only with one computation at a time. This restriction forces one to wait idle for n clock cycles before starting to feed two new multiplicands. This characteristic may seem to be natural due to the fact that the product is $2n$ bits long. We present a pipelined version of our multiplier which requires less than twice the amount of hardware of a single non-pipelined multiplier. This technique can be applied to Atrubin's multiplier as well, and is especially suited for single precision multiplications, in which one is only interested in the n most significant bits of the product. The non-pipelined multiplier has a throughput of one computation every $2n$ clock cycles even though the output is only n bits long in single precision multiplications.

In this section we show a pipelined version of the serial/parallel multiplier which can handle two computations simultaneously. This multiplier outputs two bits per clock cycle; one bit belongs to the n most-significant bits of the product of the previous inputs, and one bit belongs to the n least-significant bits of the product of the current inputs. The gain obtained by this modification is that the elapsed time between the start of two consecutive multiplication operations is n clock cycles instead of $2n$ clock cycles. The pipelined multiplier has, therefore, a throughput rate which is twice as large as the throughput of the non-pipelined multiplier. However, the amount of hardware needed for implementing the pipelined multiplier is less than the hardware needed for implementing two non-pipelined multipliers. The transformations described in Sections 4.3-4.5 can also be applied to the pipelined serial/parallel multiplier, yielding a pipelined, systolic, real-time, bit-serial multiplier.

Consider the serial/parallel multiplier. Note, that zeros are input through a_{in} during the last n clock cycles of each computation, causing the summand adders to function as half-adders: they add the carry bit, $carry_{in}$, with the sum bit, sum_{in}. Let us add a new array of half-adders which is capable of simulating the computation performed by the serial/parallel multiplier during the last n clock cycles of each computation. Both arrays (the summand-adders' array and the half-adders' array) are utilized according to the following schedule:

Every n clock cycles, new multiplicands are input in concert with *Start* rising to '1'. The values stored in the summand-adders' array are passed to the half-adders' array which finishes computing the previous product. This frees the summand-adders' array which starts computing the current product.

Figure 4.8 depicts the pipelined serial/parallel multiplier for 4-bit integers. Note, that the bits of multiplicand **a** and the *Start* signal are broadcast to all the summand-adders, whereas only the *Start* signal is broadcast to the half-adders. The n least-significant bits of each product are output by the summand-adders' array (the top output), and the n most-significant bits of the product are output by the half-adders' array (the bottom output).

The input/output ports of a half-adder are named and shown in figure 4.9 along with the functionality. The value of the $carry_{val}$ input equals the value of the $carry_{in}$ input of the summand-adder, and the value of the sum_{val} input equals the sum_{in} input of the summand-adder. The summand-adders of the pipelined serial/parallel multiplier are identical to the summand-adders of the serial/parallel multiplier, except for an increased fanout of the $carry_{in}$ and sum_{in} signals.

If one joins together the i'th summand-adder and the i'th half-adder into a single element, then one obtains a circuit the communication graph of which equals the communication graph of the serial/parallel multiplier. It is, therefore, straightforward to apply the same transformations to the pipelined serial/parallel multiplier in order to obtain a systolic pipelined multiplier.

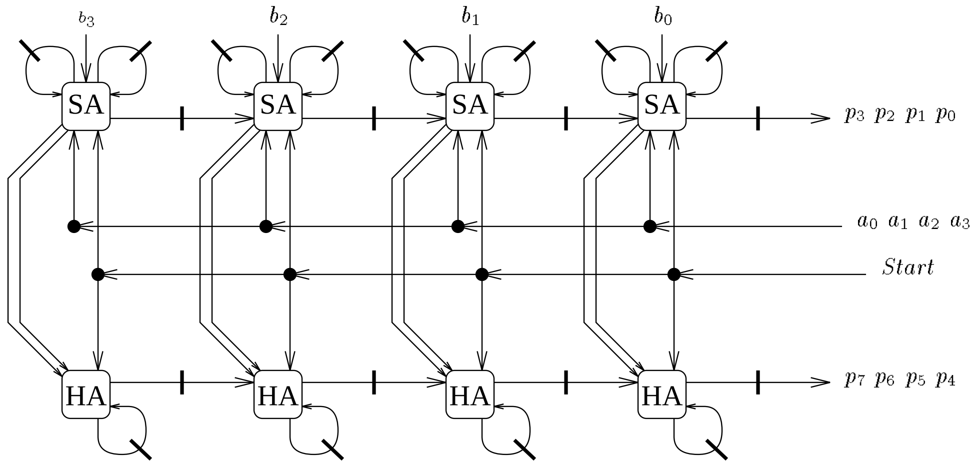
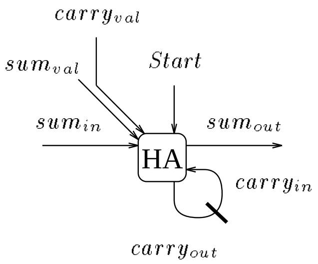


Figure 4.8: 4-bit pipelined serial/parallel multiplier.
איור 4.8: כופל סידרתי/מקבילי לארבעה ביטים.




```
if Start = 0 then
      begin
      sum_out := XOR(sum_in, carry_in)
      carry_out := AND(sum_in, carry_in)
      end
else
      begin
      sum_out := XOR(sum_val, carry_val)
      carry_out := AND(sum_val, carry_val)
      end
```

Figure 4.9: Half-adder: inputs/outputs and functionality.
איור 4.9: חצי-מחבר: קלטים/פלטים ופונקציונליות.

4.7 Gate realizations of our multiplier and Atrubin's multiplier

We claim that our multiplier requires roughly half the amount of hardware required by Atrubin's multiplier. One cannot give an exact comparison, since the amount of hardware required depends on the technology used and on various decisions which lead to specific designs. In this section we suggest gate level realizations of both multipliers, using similar gates and techniques. This example supports our claim, and that it shows that our multiplier does not have any 'hidden' properties which might complicate its realization. Moreover, it is likely that the ratios of the gate count in our multiplier and in Atrubin's multiplier given in this section will carry on to other technologies and designs. This belief stems from the similarity between the functionality of the "basic-cells" of these multipliers.

4.7.1 Symbols and notation

In this subsection we describe the hardware components which are the building blocks of the gate-level realizations given in the next subsections. Table 4.1 gives a description of the gates and memory components we use in these realizations.

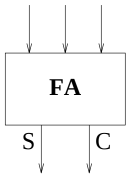
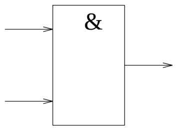
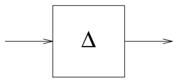
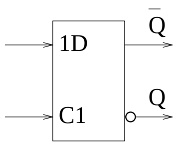
Name	Symbol	Remarks
Full Adder		S := XOR of inputs C := $\begin{cases} 0 & \text{if sum of inputs} < 2 \\ 1 & \text{otherwise} \end{cases}$
AND Gate		
Delay Element		$out(t+1) := in(t)$
D latch		Q := last value input in D while the C input was high.

Table 4.1: Gates and memory components.
טבלה 4.1: שערים ורכיבי זכרון.

4.7.2 A gate level realization of our multiplier

In this subsection we suggest a realization of our non-pipelined multiplier. A schematic of the summand adder from figures 4.6 is shown in figure 4.10.

According to figure 4.7, one needs 19 delay-elements and 4 summand-adders for every 4 bits of the multiplicands' length. (We do not assign delay-elements for storing the bits of because we use D-latches for storing them).

4.7.3 A gate level realization of Atrubin's multiplier

A description and an explanation of Atrubin's multiplier appears in [12]. Therefore, we limit the discussion in this subsection to a suggested hardware realization of Atrubin's multiplier.

Atrubin's multiplier can be obtained by retiming a linear array with broadcast. In figure 4.11 we depict Atrubin's multiplier after the retiming. The dotted boxes denote the cells of the multiplier after joining together the basic cells, which are denoted by filled boxes.

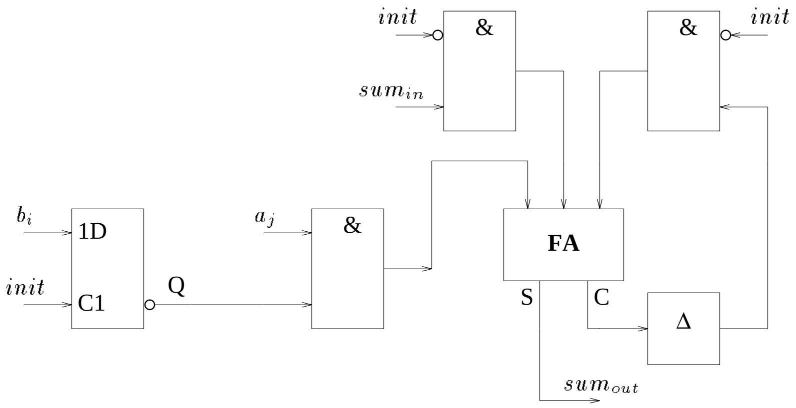


Figure 4.10: Realization of summand-adder.
.איור 4.10: מימוש מחבר

In figure **??** we name the input/output ports of the basic cells. In Atrubin's multiplier, multiplication of n bit integers requires n basic cells. A suggested realization of the basic cell is shown in figure 4.13.

The dotted boxes in figure 4.11 are constructed from 2 basic cells and 8 delay-elements (note, that the lines for a_i and b_i are two bits wide). The suggested realization of the basic cells in Atrubin's multiplier shows that each dotted box can be constructed from: $2 \times 2 = 4$ full adders, $2 \times 2 = 4$ D latches, $2 \times 3 = 6$ AND gates, $2 \times 2 = 4$ AND gates with 3-inputs, and $2 \times 3 + 8 = 14$ delay elements (3 per basic cell, and 8 for lines in the dotted box). (It is possible to reduce the number of delay-elements in each basic-cell to 2 by deleting the delay-element used to delay $Init$, and by taking an output of a delay-element which delays the outgoing $Init$ signal).

4.7.4 Summary of comparison results

In table 4.2 we compare the number of hardware elements needed for realizing our multiplier and Atrubin's multiplier using the realizations suggested in Sections 4.7.3 and 4.7.2. We count the number of hardware elements needed for every 4 bits of the multiplicands' length. This comparison shows that our non-pipelined multiplier requires roughly half the hardware as compared to Atrubin's multiplier.

4.8 Summary

We presented a which non-pipelined multiplier requires roughly half the hardware as compared to Atrubin's multiplier. The speed of a synchronous circuit (the fastest clock it can

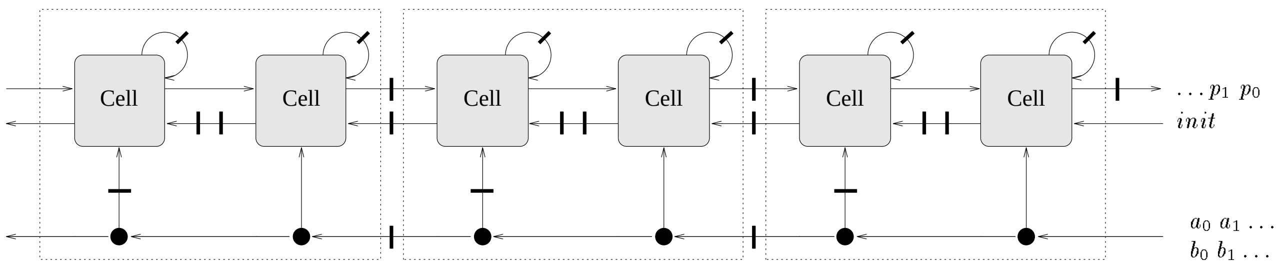


Figure 4.11: Atrubin's multiplier.
איור 4.11: הכופל של אטרובין.

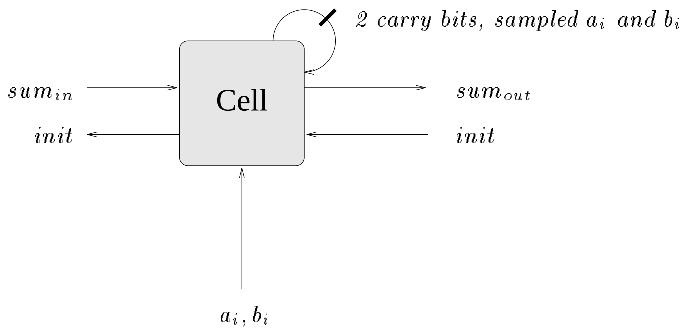


Figure 4.12: Ports of a cell in Atrubin's multiplier
איור 4.12: כניסות ויציאות של תא בכופל של אטרובין.

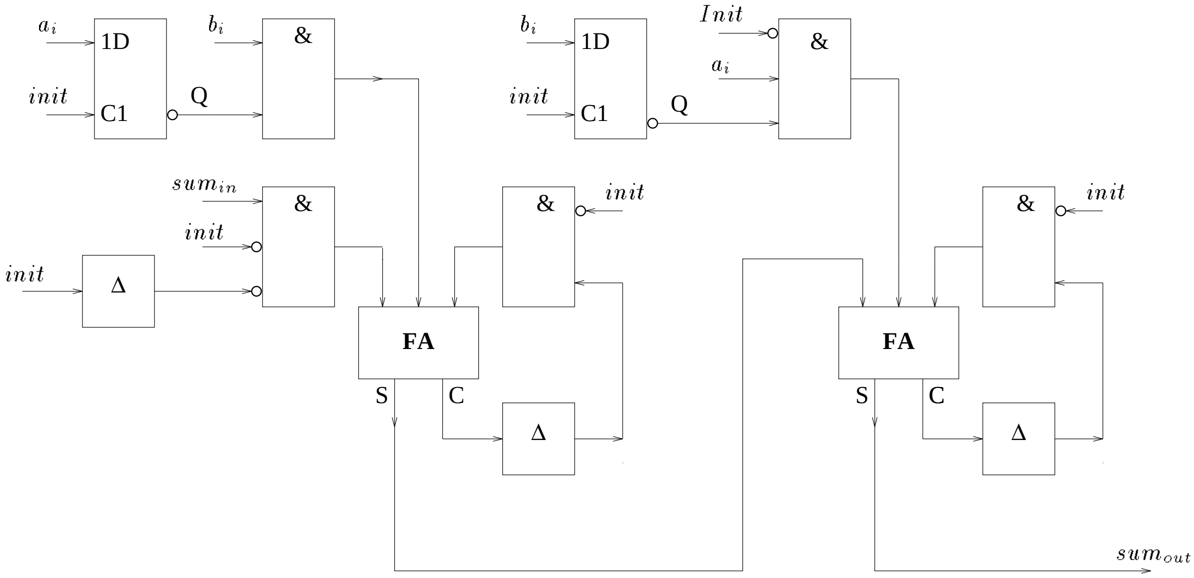


Figure 4.13: Realization of the basic cell of Atrubin's multiplier.
איור 4.13: מימוש תא בסיסי בכופל של אטרובין.

operate with) is determined by the longest path of combinational elements a signal has to propagate through during a single clock cycle. In the suggested realizations both multipliers have critical paths of similar length: In our multiplier the longest path goes through 4 AND gates and 4 full-adders. In Atrubin's multiplier, the longest path goes through 2 AND gates (each with 3 inputs) and through 4 full-adders. Although our multiplier requires less hardware than Atrubin's, the fact that the multiplicands are input at different rates might complicate integrating it with other circuits.

In section 4.6 we introduced a technique for transforming a non-pipelined multiplier into a pipelined one. This technique can also be applied to Atrubin's multiplier, and thus obtaining a pipelined version of Atrubin's multiplier.

The circuits introduced in this paper are particularly suited for throughput oriented environments which operate with very fast clock rates. Cryptographic uses are examples of such environments in which very long integers are multiplied, therefore, rendering multipliers with quadratic area impractical. For example, the modular multiplier described in [10] includes an integer multiplier, and Atrubin's multiplier was suggested for that purpose.

Hardware Element	Our Multiplier	Atrubin's Multiplier
Full Adder	4	8
D-Latch	4	8
AND gates	12	12
3-input AND gates	0	8
Delay Elements	19	28

Table 4.2: Hardware Elements Needed For Every 4 Bits.
טבלה 4.2: חומרה נידרשת לכל ארבעה ביטים.

Chapter 5

Overcoming chip-to-chip delays and clock skews

5.1 Introduction

This chapter proposes a solution which enables mapping large systolic linear arrays and systolic two-dimensional arrays onto several chips while almost maintaining the clock rates which are obtainable when these circuits are small enough to fit into a single chip. In general, mapping a circuit onto several chips incurs a physical setting which differs from those within a chip. Specifically, the delay of chip-to-chip interconnections is much longer than the delay of wires and gates in a chip. This delay also effects the bandwidth of the wires. In addition, the clock skew between chips is larger than the clock skew within a chip. One may mistakenly conclude that the feasible clock period of a systolic array cannot be smaller than the maximal delay of an interconnection in a realization of the circuit.

We focus on systolic linear arrays and systolic two-dimensional arrays. These systolic arrays have a very simple physical layout, which have short interconnections between functional units. The clock period of such systolic arrays does not depend on the size of the systolic arrays. Namely, the minimum feasible clock period of a systolic array is determined by the propagation delay of its cells, and not by the number of cells in the array. However, how does one maintain such fast clock rates when the systolic array contains too many cells to fit into a single chip? We present a solution which enables a feasible clock period which equals twice the clock period of a small systolic array which fits into one chip. This solution does not require any modifications to the cells of the array; all modifications are performed along the wires between cells. We also present a solution which enables a feasible clock period which equals $(\Phi + d_{max} + \varepsilon)$ time units, where Φ denotes the clock period of a small systolic array which fits into one chip, d_{max} denotes the maximum propagation delay of a gate in a realization of a cell of the systolic array, and $\varepsilon > 0$ is arbitrary. This better solution requires modifying some of the cells of the systolic array. Our solutions do not rely on special analogue techniques such as sense amplifiers. Each solution described as a sequence of transformations (logic duplication and retiming), reductions, and an implementation of interconnections which have a required behavior in a given physical setting. It is shown that each step preserves functionality, and subsequently, the correctness of the proposed solutions is implied.

We formalize the parameters which characterize the physical properties of chip-to-chip interconnections as follows: The *delay* of an interconnection is the amount of time it takes for a signal to arrive to the other end of the interconnection. The delay also effects the bandwidth, since the input to the interconnection must be kept stable until the signal arrives to the other end. Delays of chip-to-chip interconnections are much larger than the delays of wires and gates within a chip. Utilizing fast clock rates in presence of long delay interconnections is an important issue, which we solve for systolic linear arrays and systolic two dimensional arrays. Our implementation allows delays of chip-to-chip interconnections which equal several clock periods. In addition, we consider *clock skews* between the clocks of the chips. Our implementation is "loosely" synchronous in the sense that it allows clock skews of several clock periods. We believe that this is the first work which deals which such large clock skews, and which shows how to implement "purely" synchronous circuits in "loosely" synchronous environments.

Our solution consists of three stages:

1. We consider an interconnection between chips whose behavior is given and which is supposed to be realized in a given physical setting which is characterized by long delays and clock skews. This implementation uses two ideas: First, we trade the delay of the wires connecting chips with the latency of the interconnection (which is measured as the number of clock cycles it should take a signal to propagate through the interconnection). Second, we increase the bandwidth of the implementation of the interconnection by connecting the chips with several parallel wires. This implementation uses two special hardware components which we call an n-demultiplexer and an n-multiplexer. A fine tuning between the required behavior of the interconnection and the parameters of the physical setting enables correct functionality of the implementation with very fast clock rates.

2. We duplicate logic in the vicinity of the interconnections between chips. The same computations are performed by the duplicated logic at both ends of an interconnection between chips.

3. The systolic array (after duplicating logic) is retimed so that two objectives are met: (a) The chip-to-chip interconnections have enough registers along them so that the implementation proposed in the first stage is applicable. The number of registers required along the interconnection between chips is determined by the delay and clock skew. (b) The retimed array can operate with a clock period which almost equals the feasible clock period which is obtainable when a small version of the array is mapped onto a single chip.

Our solution is applicable also in situations in which a systolic array is partitioned among remote areas of the same chip. In general, this chapter presents a method of overcoming the naive lower bound on clock rates imposed by sparse slow interconnections in systolic arrays.

This chapter is organized as follows: Section 5.2 discusses previous works on related issues. Section 5.3 presents an implementation, in a given physical setting, of an interconnection whose behavior is specified. Section 5.4 presents our solution for systolic linear arrays. Section 5.5 presents our solution for systolic two-dimensional arrays. Section 5.6 discusses the hardware overheads required by our solutions.

5.2 Previous works

Previous works which deal with problems incurred by chip-to-chip interconnections focus on reducing the delay and clock skew. Our starting point is a given physical setting after measures for decreasing delay and clock skew have been taken. Hence, our work seeks solutions after other techniques have been utilized as much as possible. A brief summary of previous works follows.

One solution is to use a pair of wires for each interconnection and a sense amplifier to determine the value of the signal. This solution reduces the delay, and is most common for on-chip interconnections in memory chips [2, p. 147-155].

Solutions which reduce the clock skew are divided into two categories: (1) Use special control circuitry (phased locked loops) which guarantees bounded clock skews. [2, p. 358-360] (2) Use H-trees for clock distribution to reduce the differences in path lengths from the clock driver to neighboring terminals which use the clock [2, Sec. 8.8], [14]. This approach was examined by Fisher and Kung [14] for systolic arrays which fit into a single chip. They show that it can solve the problem of clock skews for linear arrays but not for two-dimensional arrays. (Note that we consider a setting in which the troublesome interconnections are the chip-to-chip interconnections, and that these interconnections are few compared to the number of on-chip interconnections).

A third approach is to use self-timed logic and asynchronous circuitry [2, Sec 8.8], [29]. The system is partitioned into locally synchronous regions, each region has its own local clock. Communication between the regions can be either asynchronous (using a self-timed discipline) or synchronous using a slow global clock.

5.3 Trading delay with latency

In this section we describe a mechanism which we use in the proposed solutions in Sections 5.4 and 5.5. This tool describes how to implement an interconnection whose behavior is given in a physical setting which is characterized by long delays and clock skews.

The required behavior of the interconnection between chips (namely, the way we would like the interconnection to function) is specified as follows:

1. The interconnection is synchronous. A new value is input on every clock cycle, and it outputs a new value on every clock cycle. The required clock period of the interconnection is specified.

2. Latency - We measure the latency in time units which equal the clock period. The latency equals the number of clock cycles which elapse from the time a signal is input to the interconnection until the signal is received at the interconnection's output. However, new values are sent along the interconnection even before the old value is received at the other end. We also refer to the latency as "the number of registers along the interconnection". The reason for this terminology comes from the fact that one can regard the required behavior of an interconnection as if it is a k-stage shift register. Every clock cycle, a new value is input, and a value is output from the the shift register. It takes a value k clock cycles to get to the output of the shift register, and therefore, the latency is k clock cycles.

   We require that one may input new signals to the interconnection on every clock cycle, regardless of the number of registers along the interconnection. In other words, the number of registers along a wire effects the latency, but does not effect on the throughput.

3. The breadth of the interconnection. This parameter specifies the number of bits input simultaneously to the interconnection. To simplify the discussion, we first deal with interconnections with single-bit breadths.

The physical setting (namely, the parameters which characterize the actual behavior of the interconnection) is characterized by the following parameters:

1. The delay of the wires. The delay is the time that elapses from the moment a signal is input to a wire until the signal arrives at the other end of the wire. The delay is denoted by d. Moreover, the signal is supposed to remain valid for at least d time units at the input to insure correct arrival at the other end of the wire. Therefore, a delay of a wire also determines the bandwidth of the wire.

2. The clock skew between the chips. We consider clock skews which can be much larger than the clock period, and thus constitute a "loosely" synchronous implementation. Let s bound the clock skew, i.e. the absolute value of the difference between the times of the i'th pulses of both clocks is at most s time units.

Henceforth, we normalize the values of the delay and the clock skew by dividing these values by the clock period. In other words, d and s are measured in units of clock periods, where the clock period is part of the specification of the behavior of the interconnection. This normalization frees us from referring to the value of the clock period.

We propose implementations of the following combinations of behavioral specification of the interconnection and physical settings:

1. An interconnection with a latency of d clock cycles, in a physical setting which has a delay $d > 0$ and a clock skew $s = 0$.
2. An interconnection with a latency of s clock cycles, in a physical setting which has a delay $d = 0$ and a clock skew $s > 0$.
3. An interconnection with a latency of $(d + s)$ clock cycles,in a physical setting which has a delay $d > 0$ and a clock skew $s > 0$.

The proposed implementation uses two components, an n-demultiplexer and an n-multiplexer, which are defined as follows: An n-demultiplexer is clock controlled and has a single input and n outputs. It is a device which has memory and it routes the input to the outputs, one by one, in cyclic order. In each clock cycle, the input is routed to one output, and the values of the other outputs are kept intact. The n-multiplexer is also clock controlled and has n inputs and a single output. The n-multiplexer routes its inputs to its output, one by one, in cyclic order.

Figure 5.1 depicts an implementation of an interconnection constructed with an n-demultiplexer and an n-multiplexer. The implementation uses n wires in order to implement a single "required" interconnection. We name the wires $e_0, e_1, ..., e_{n-1}$. Wire e_i emanates from the i'th output of the demultiplexer, and enters the i'th input of the multiplexer.

Assume, for the time being, that the propagation delay of an n-demultiplexer and an n-multiplexer is zero. We consider these propagation delays at the end of this section.

Correct functionality of an implementation of an interconnection requires synchronizing between the n-multiplexer and the n-demultiplexer. This synchronization is quantified by a *shift*. When the shift is set to δ, the components function as follows: The n-multiplexer, in its i'th clock cycle, sets the the value of output $(i \bmod n)$ to the value of the input, all other outputs are kept intact. The n-demultiplexer, in its $(i + \delta)$ clock cycle, routes its $(i \bmod n)$ input to its output. Synchronizing the shift between the multiplexer and the demultiplexer is part of initializing the circuit, and is done during power up. The suggested implementation resembles circular buffers which are common in software.

The following claim summarizes the properties of such an interconnection:

Claim 5.1: Consider an implementation of an interconnection depicted in figure 5.1. Assume that the delay and clock skew are measured in clock periods. The following statements hold:

1. Assume that the clock skew is zero. Let $n = d + 1$ and let the shift be d. The implementation realizes an interconnection with a latency equal to d, even if the delay of each wire is upto d.

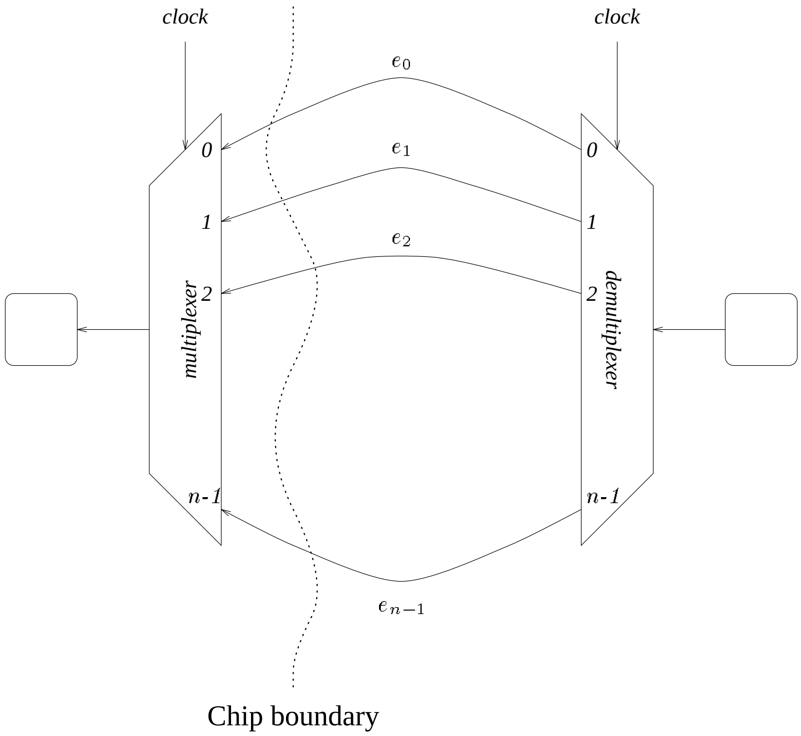


Figure 5.1: Implementation of a wire interconnecting chips.
איור 5.1: מימוש חיבור בין שבבים.

2. Assume that the wire delay is zero. Let $n = 2 \cdot s + 1$ and let the shift be s. The implementation realizes an interconnection with a latency equal to s, even if the the clock skew is upto s.

3. Let $n = d + 2 \cdot s + 1$ and let the shift be $(d + s)$. The implementation realizes an interconnection with a latency equal to $(d + s)$, even if the the delay of each wire is upto d and the clock skew is upto s.

Proof:

proof of part 1: A value written to a wire is kept valid for $d+1$ clock cycles, and arrives at the input of the multiplexer after at most d clock cycles. The multiplexer reads a value from a wire exactly d clock cycles after it is written to it. Therefore, the value read from a wire equals the value written to it, and the signal is delivered through the interconnection with a latency equal to d clock cycles as required.

proof of part 2: We differentiate between the clock cycles of the multiplexer and the demultiplexer. In the i'th clock cycle of the demultiplexer, a value is written on wire

$e_{i \bmod (2\cdot s+1)}$. The multiplexer reads a value on wire $e_{i \bmod (2\cdot s+1)}$ in cycle $i+s$ of *its* clock. Since the clock skew is at most s clock cycles, the time elapsed between writing and reading the value to the wire is non-negative and it is not greater than $2 \cdot s$ clock cycles. Since the value written on the wire remains valid for $2\cdot s+1$ clock cycles, it follows that the value read from the wire equals the value written to it, and the implementation behaves as required.

proof of part 3: Consider a value written to wire $e_{i \bmod (d+2\cdot s+1)}$ in the i'th clock cycle of the demultiplexer. The multiplexer reads a value from this wire in its $i+d+s$ clock cycle. The written value equals the read value provided that the elapsed time between writing it and reading it is at least d clock cycles and less than $d+2\cdot s+1$ clock cycles. Since the skew is at most s clock cycles, it follows that the implementation behaves as required. □

Note, that if the breadth of an interconnection is k (i.e. k bits are sent in parallel every clock cycle), then the proposed implementation requires $k \cdot n$ "real" wires. The area of an n-multiplexer and an n-demultiplexer for breadth k is $O(k \cdot n)$.

The propagation delay of the n-demultiplexer and the n-multiplexer can be accounted for in the delay of the wires. It is reasonable to assume that the propagation delays of the n-demultiplexer and the n-multiplexer are $O(\log n)$. Therefore, it is sufficient to use a value of n which satisfies: $n \geq d+2\cdot s+1+O(\log n)$. This does not increase n significantly, and we ignore this increase, for the sake of simplicity, in the sequel.

5.4 Linear Arrays

In this section we consider a systolic linear array which consists of cells which are Mealy finite state machines. Every edge connecting two cells has a register along it. For the sake of simplicity, we assume that all the cells are identical, however, the proposed implementation holds even when the cells are not identical. Suppose that one wishes to implement such a systolic linear array which is so long, that it does not fit into one chip. When one maps this linear array onto more than one chip, the interconnections between chips incur a large delay or there might be a large clock skew between the clocks of connected chips. The proposed solution reduces the problem caused by chip-to-chip interconnections to the situations solved in Section 5.3, which can be implemented so that they can operate with short clock periods. In particular, this reduction transforms the linear array into an equivalent circuit, in which all chip-to-chip interconnections have several registers along them. The functionality of the linear array is kept intact. Assume that the feasible clock period of the linear array when mapped onto a single chip equals 1 time unit. First, we show a solution which enables a clock period of 2 time units. Then, we show how one can obtain clock periods which are close to 1 time units, when the gate level description of the cells is considered.

Let m denote the number of registers along the interconnection required for the implementation suggested in Section 5.3 for overcoming the delay of wires interconnecting chips and the clock skew between connected chips. Namely, $m = d + s$, where d denotes the delay and s denotes the clock skew. The reduction consists of two stages: In the first stage, a segment of $O(m)$ cells is duplicated, so that a copy of the duplicated segment appears on both sides of the interconnection. The duplicated segments perform, at this stage, the same computations at the same time, and therefore, functionality is kept intact. In the second stage, the duplicated segments are retimed. This retiming causes all troublesome interconnections (which cross from one chip to another) to have m registers along them. Retiming preserves functionality, and therefore, the original linear array and the obtained circuit are functionally equivalent. The interconnections between chips are implemented as suggested in Section 5.3.

5.4.1 Duplicating Segments

Duplicate a segment of $O(m)$ cells, so that a copy of the segment appears on each side of the boundary. The exact length of the duplicated segment is discussed in Subsection 5.4.2. Connect the segments as shown in figure 5.2. Note that: (a) The rightmost output of the left duplicated segment and the leftmost output of the right duplicated segment are not connected. (b) The output of the right most cell to the left of the duplicated segments is split and sent to two cells. The same holds for the output of the left most cell to the right of the duplicated segments.

Clearly, the cells which are connected to the duplicated segments do not "notice" any change in functionality, and therefore, this step keeps functionality intact.

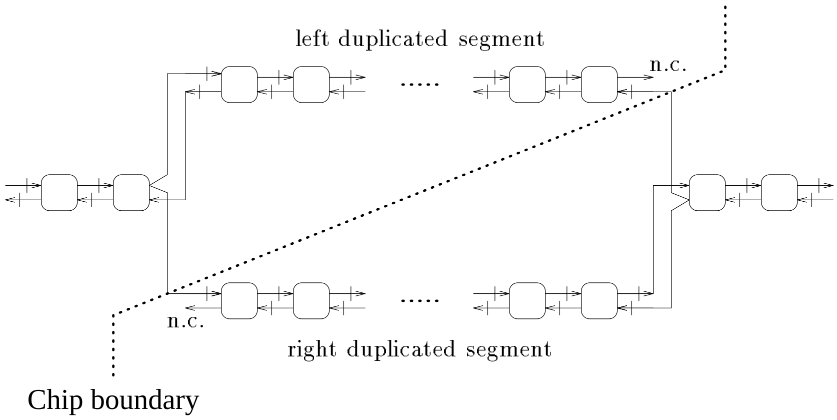


Figure 5.2: Duplicating segments.
איור 5.2: שיכפול קטעים.

5.4.2 Retiming The Linear Array

Our goal is to retime the linear array after duplicating segments so that the interconnecting between different chips have at least m registers along them. When this goal is achieved, the implementation suggested in Section 5.3 is applicable. Functionality of the circuit is kept intact due to the Retiming Lemma 2.3.

First, we present a retiming which enables a clock period of 2 time units (where the feasible clock period of the array when mapped onto a single chip equals 1 time unit). The advantage of this retiming is that it does not require partitioning of the cells into "smaller" sub-components. A solution which enables a clock period equal to 2 time units is depicted in figure 5.3. Each segment consists of $2m-3$ cells. The lags of the cells are written in the cells, and the number of registers along the wires are depicted in the figure.

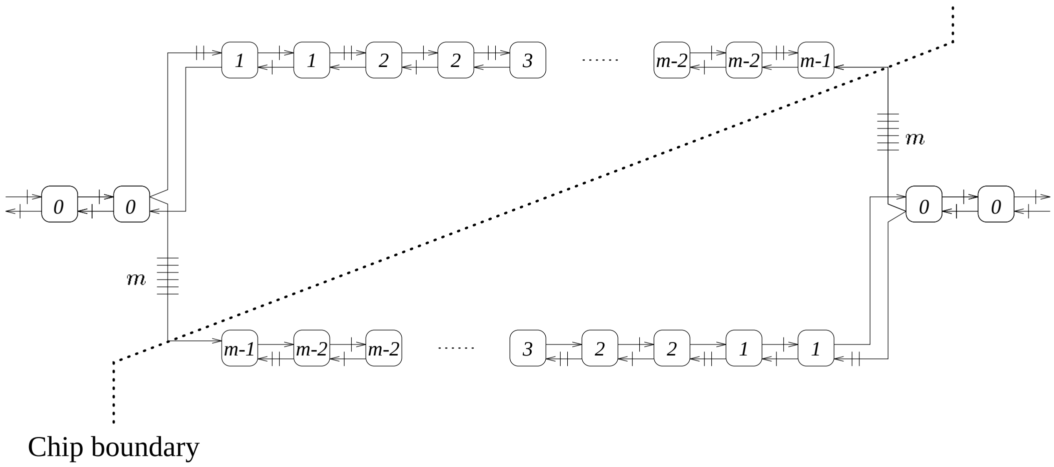


Figure 5.3: Retiming the linear array.
איור 5.3: רה-תזמון של המערך הליניארי.

Shorter clock periods can be obtained by considering the gate-level description of the cells, i.e., replace each cell with a network of combinational gates and registers. Theorem 5.1 states the tradeoff between the length of the duplicated segment and the clock period which can be obtained by retiming. The retiming, the existence of which is guaranteed by the theorem, can be found by solving a single-source shortest-paths problem.

Theorem 5.1: Let ℓ denote the number of cells in the duplicated segment of the linear array. Let m denote the number of registers required along interconnections between chips. Let d_{max} denote the maximum propagation delay of a combinational gate after partitioning a cell into combinational gates and registers. Let $\varepsilon > 0$. If $\ell \geq m \cdot (1 + 1/\varepsilon) - 1$, then the duplicated segments can be retimed so that (a) they can operate with a clock period of $1 + \varepsilon + d_{max}$ time units, and (b) interconnections which enter the segments from the neighboring chip have at least m registers along them.

Proof: The proof is based on an algorithm presented in work of Even and Litman [13]. The theorem can also be proven using the technique and algorithm of Papaefthymiou [26].

Figure 5.2 depicts the communication graph of the linear array in the vicinity of the chip boundary. Note, that each cell is now considered as communication graph by itself, since the cells are partitioned into a network of combinational gates and registers. Define a weight function, $w()$, on the edges, so that the weight of edge, e, equals the number of registers along it.

A "lower-bound" weight function, $\underline{w}()$, is defined on the edges of the communication graph as follows: (1) The weight of edges, e, crossing between chips is $\underline{w}(e) = m$. (2) The weight of all other edges, $e = u \rightarrow v$, is defined by $\underline{w}(e) = d(u)/(1+\varepsilon)$, where $d(u)$ denotes the propagation delay of vertex u.

Let p denote a path in the communication graph, and let $w(p)$ denote the sum of the weights of the edges along p. Similarly, $\underline{w}(p)$ denotes the sum of the lower-bound weights of the edges along p. Let $w_r(e)$ denote the number of registers along edge e after retiming the circuit by a retiming function $r()$. Similarly, $w_r(p)$ denotes the number of registers along path p after retiming the circuit by a retiming function $r()$. Let $d(p)$ denote the sum of the propagation delays of the vertices along the path p.

First, we show that our goal is to find a retiming function, $r()$, so that $w_r(p) \geq \lfloor \underline{w}(p) \rfloor$, for every path p. When this goal is achieved, the clock period is at most $1 + \varepsilon + d_{max}$ time units, and all interconnections which cross the chip boundary have at least m registers along them. The following claim summarizes this property.

Claim 5.2: If $r()$ is a retiming function of the linear array after duplicating segments which satisfies: $w_r(p) \geq \lfloor \underline{w}(p) \rfloor$, for every path p, then:

1. Let e denote an edge which crosses between chips. $w_r(e) \geq m$, and therefore, after retiming, the interconnection which enters the segment from the neighboring chip has at least m registers along it.

2. If $w_r(p) = 0$, then the propagation delay, $d(p)$, of the path p is at most $1 + \varepsilon + d_{max}$ time units. Therefore, the retimed circuit can operate with a clock period of $1 + \varepsilon + d_{max}$ time units.

Proof: Part (1) follows immediately from the definition of $\underline{w}(e)$.

Let p be a path for which $w_r(p) = 0$ which ends in vertex v. Since $w_r(p) \geq \lfloor \underline{w}(p) \rfloor$, it follows that $\underline{w}(p) < 1$. By the definition of $\underline{w}(p)$, it follows that $d(p) - d(v) = \underline{w}(p) \cdot (1+\varepsilon)$. Since $\underline{w}(p) < 1$, and since $d(v) \leq d_{max}$, it follows that $d(p) \leq 1 + \varepsilon + d_{max}$, and the claim follows. □

According to [13, lemma 2.1], a retiming which satisfies $w_r(p) \geq \lfloor \underline{w}(p) \rfloor$, for every path p, exists if and only if $w(c) \geq \underline{w}(c)$, for every simple cycle c. Therefore, it is sufficient to prove, that if $\ell \geq m \cdot (1 + 1/\varepsilon) - 1$, then $w(c) \geq \underline{w}(c)$, for every cycle c.

Claim 5.3: Let c denote a simple cycle in the communication graph depicted in figure 5.2. Call an edge which interconnects cells on different chips a *jump*. If $\ell \geq m \cdot (1 + 1/\varepsilon) - 1$, then:

$$w(c) \geq \frac{w(c)}{1+\varepsilon} + m \cdot \|\{e \in c : e \text{ is a jump }\}\|.$$

Proof: Every path from a jump to a jump passes through at least $\ell + 1$ cells. Therefore, $\|\{e \in c : e \text{ is a jump }\}\| \leq w(c)/(\ell + 1)$. It is sufficient to show, that if $\ell \geq m \cdot (1 + 1/\varepsilon) - 1$, then:

$$w(c) \geq \frac{w(c)}{1+\varepsilon} + m \cdot \frac{w(c)}{(\ell+1)}.$$

This can be easily verified, and the claim follows. □

Given a cycle, c, partition cycle c into jump edges and non-jump edges. The sum of the weights, $\underline{w}(e)$, of jump edges equals $m \cdot \|\{e \in c : e \text{ is a jump }\}\|$. The sum of the weights, $\underline{w}(e)$, of non-jump edges is bounded by $d(c)/(1 + \varepsilon)$. The delay between every two registers along c is bounded by 1 time unit. (Recall, that the circuit originally operated with a clock period of 1 time unit). Therefore, $d(c) \leq w(c)$. Combining these observations, we obtain:

$$\underline{w}(c) \leq \frac{w(c)}{(1+\varepsilon)} + m \cdot \|\{e \in c : e \text{ is a jump }\}\|. \tag{5.1}$$

Using claim 5.3 and equation 5.1, it follows that, if $\ell \geq m \cdot (1 + 1/\varepsilon) - 1$, then $w(c) \geq \underline{w}(c)$, for every simple cycle c, and the theorem follows. □

The retiming which satisfies $w(p) \geq \lfloor \underline{w}(p) \rfloor$, for every path p, can be found by solving a single-source shortest-paths problem on the communication graph with the weight function defined by $\widehat{w}(e) \triangleq w(e) - \underline{w}(e)$.

5.5 Two dimensional arrays

In this section we consider a systolic two dimensional array which does not fit into one chip. Figure 5.4 depicts a segment of a systolic mesh-connected two dimensional array. The segment shows the cells in the vicinity of a boundary between four chips. As in Section 5.4, a reduction is shown, which reduces the problem caused by interconnections between chips to the situations solved in Section 5.3. Namely, the array is transformed into a functionally equivalent array, in which all interconnections between chips have several registers along them. Assume that the feasible clock period of the two-dimensional array when mapped onto a single chip equals 1 time unit. First, we show a solution which enables a clock period

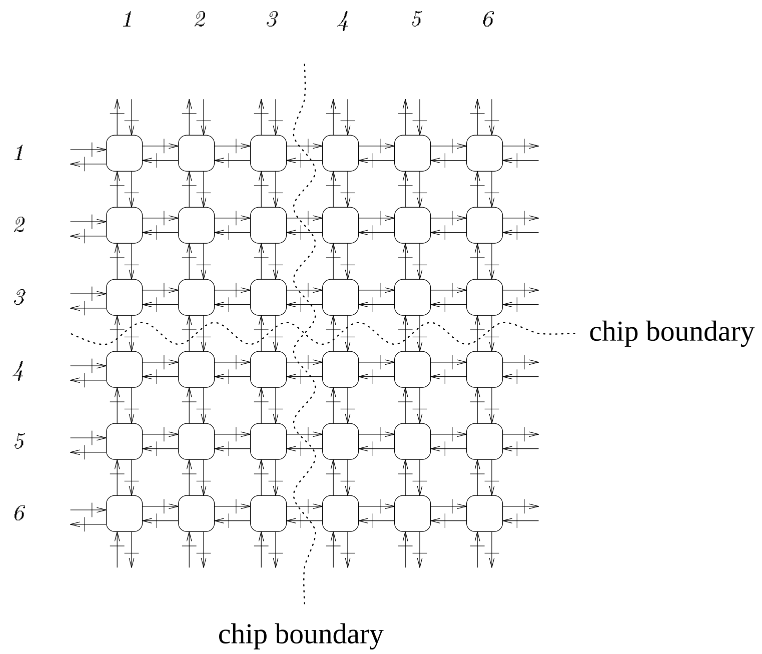


Figure 5.4: A boundary between 4 chips in a two-dimensional array.
איור 5.4:   מפגש גבולות בין 4 שבבים במערך דו-מימדי.

of 2 time units. Then, we show how one can obtain clock periods which are close to 1 time units, when the gate level description of the cells is considered.

The reduction consists of two stages (as in Section 5.4): duplicating hardware and retiming. Let $m = d + s$, where d denotes the delay of wires interconnecting chips and s denotes the clock skew between connected chips. In the first stage, $O(m)$ rows and columns are duplicated along the chip boundaries. The interconnections between chips are set so that cells along the chip boundaries only receive inputs from other chips. The outputs of the cells along the chip boundaries which originally emanate from the chip are not connected. Figure 5.5 provides a diagrammatic description of the duplication stage: Indices of rows and columns are shown at the left and at the top. Six rows and six columns are duplicated along the chip borders. Therefore, rows $4-9$ appear both in the top two chips and in the bottom two chips. Similarly, columns $4-9$ appear twice. The interconnections within each chip are not modified, and are not shown in the figure. Interconnections between chips are modified. Figure 5.5 depicts only a small portion of interconnections between chips. However, all other interconnections between chips follow the same principle. Namely, an input to a cell on a chip boundary comes from the neighbor of the cell (taking into account the duplication) in the neighboring chip. For example, the right outputs of cells in the 3rd column are connected to the left inputs of the cells in the 4th column both in the same chip and in the chip to the right. At this stage, all interconnections have exactly one register along them. It is important to note, that the functionality of the array is kept intact after this stage.

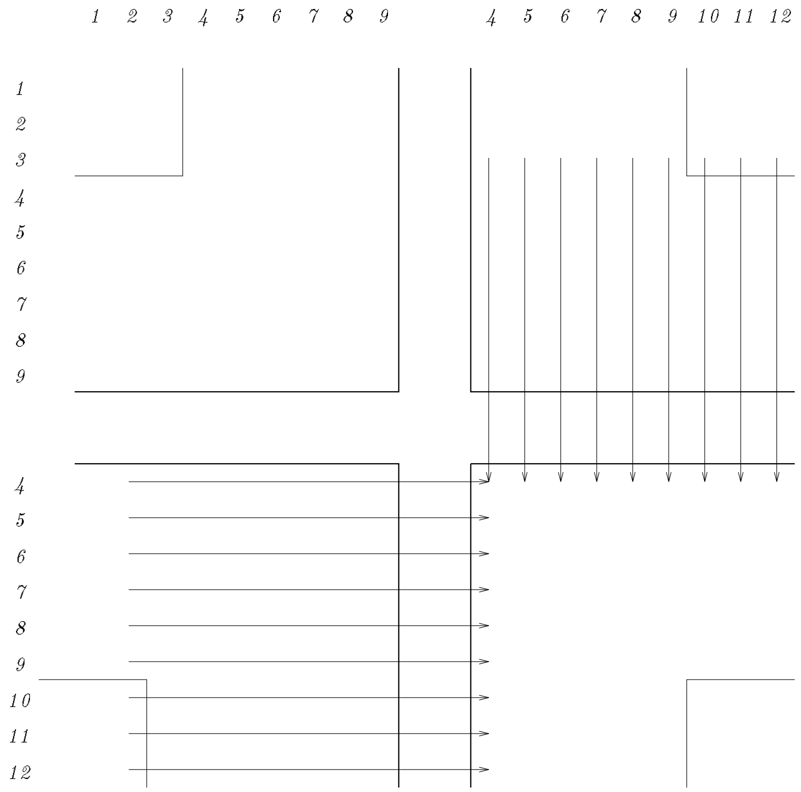


Figure 5.5: Duplicating rows and columns and some interconnections.
איור 5.5: שיכפול שורות ועמודות ומספר חיבורים.

In the second stage, the duplicated rows and columns are retimed. After the retiming, all interconnections between chips have m registers along them. The retiming is a "two-dimensional" version of the retiming described in Subsection 5.4.2. Figure 5.6 depicts a retiming for the case $m = 4$ which enables a clock cycle of 2 time units (where the feasible clock period of the array when mapped onto a single chip equals 1 time unit). The duplication stage involved duplicating 6 rows and columns along the chip boundaries. In figure 5.6, each chip is divided into areas with identical lags, and the lags are shown in these areas. This retiming has two important properties: (a) There are no paths of length 2 which start at a cell with lag $i+2$ and end in a cell of lag i. Therefore, the longest combinational (registerless) path consists of two cells, and each chip can operate with a clock cycle of 2 time units. (b) If an interconnection between chips emanates from a cell with lag i, then it enters a cell with lag $i+3$. Therefore, 3 registers are added to all interconnections between chips, and they have 4 registers along them after the retiming.

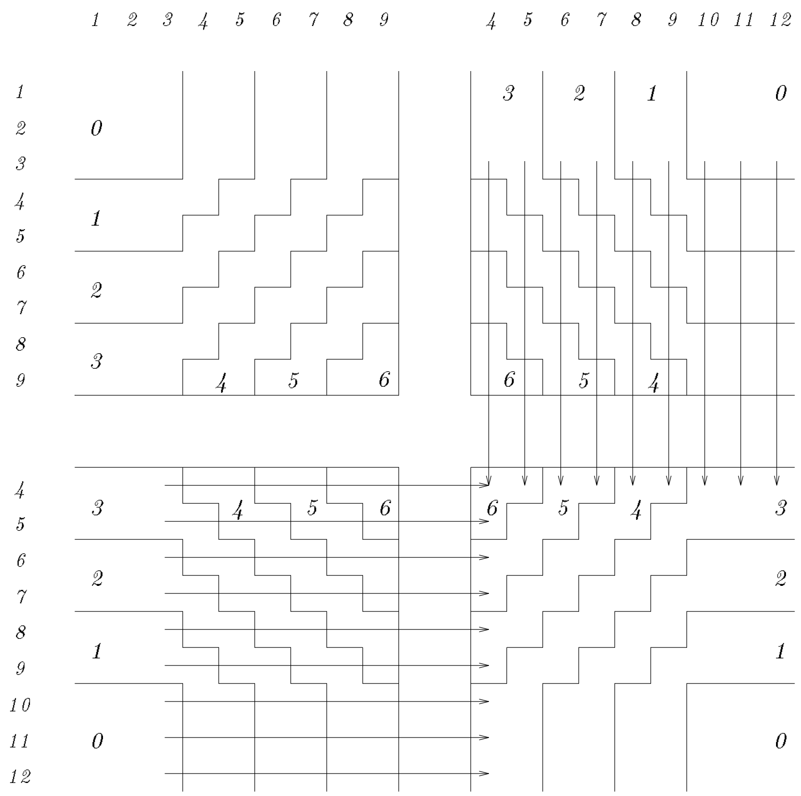


Figure 5.6: Retiming the duplicated rows and columns ($m = 4$) to obtain a clock period equal to 2.

איור 5.6: רה-תזמון של השורות והעמודות המשוכפלות (עבור $m = 4$) לקבלת מחזור שעון השווה ל-2.

Shorter clock periods can be obtained by considering the gate-level description of the cells, i.e., replace each cell with a network of combinational gates and registers. Theorem 5.2 states the tradeoff between the length of the duplicated segment and the clock period which can be obtained by retiming. The retiming, the existence of which is guaranteed by the theorem, can be found by solving a single-source shortest-paths problem.

Theorem 5.2: Let ℓ denote the number of duplicated rows and columns along the chip boundaries. Let m denote the number of registers required along interconnections between chips. Let d_{max} denote the maximum propagation delay of a combinational gate after partitioning a cell into combinational gates and registers. Let $\varepsilon > 0$. If $\ell \geq m \cdot (1 + 1/\varepsilon) - 1$, then the two dimensional array (after duplicating rows and columns) can be retimed so that (a) the array can operate with a clock period of $1+\varepsilon+d_{max}$ time units, and (b) interconnections which enter a chip from a neighboring chip have at least m registers along them.

Proof: The proof follows the proof of theorem 5.1, but is complicated due to the two-dimensional topology.

As in the proof of theorem 5.1, we consider the communication graph derived from the array after duplicating rows and columns and the weight function associated with it. The communication graph is derived from the circuit depicted in figure 5.5. The weight of all the edges interconnecting cells equals 1. Each cell is partitioned into a network of combinational gates and registers, and the weight of an edge within a cell equals the number of registers along it..

A "lower-bound" weight function, $\underline{w}()$, is defined on the edges of the communication graph as follows: (1) The weight, $\underline{w}(e)$, of edges which cross from one chip to another equals m. (2) The weight of all other edges, $e = u \rightarrow v$, is defined by $\underline{w}(e) = d(u)/(1+\varepsilon)$, where $d(u)$ denotes the propagation delay of vertex u.

The proof of claim 5.4 is similar to the proof of claim 5.2.

Claim 5.4: If a retiming function, $r()$, of the two-dimensional array after duplicating rows and column satisfies: $w_r(p) \geq \lfloor \underline{w}(p) \rfloor$, for every path p, then:

1. For every edge, e, which crosses from one chip to another, $w_r(e) \geq m$. Therefore, after retiming, the interconnections which connect neighboring chips have at least m registers along them.
2. If $w_r(p) = 0$, then the propagation delay, $d(p)$, of the path p is at most $1+\varepsilon+d_{max}$ time units. Therefore, the retimed circuit can operate with a clock period of $1+\varepsilon+d_{max}$ time units.

According to [13, lemma 2.1], a retiming which satisfies $w_r(p) \geq \lfloor \underline{w}(p) \rfloor$, for every path p, exists if and only if $w(c) \geq \underline{w}(c)$, for every simple cycle c. It is, therefore, sufficient to show, that if $\ell \geq m \cdot (1+1/\varepsilon) - 1$, then $w(c) \geq \underline{w}(c)$, for every simple cycle c.

Claim 5.5: Let c denote a simple cycle in the two-dimensional array after duplicating rows and columns. If $\ell \geq m \cdot (1+1/\varepsilon) - 1$, then:

$$w(c) \geq \frac{w(c)}{1+\varepsilon} + m \cdot \|\{e \in c : e \text{ is a jump }\}\|.$$

Proof: We differentiate between vertical jumps and horizontal jumps. Similarly, we differentiate between horizontal and vertical edges which connect cells on the same chip.

Consider a horizontal jump in cycle c and the set of horizontal edges connecting cells which follow this jump until the next horizontal jump in c. This set of horizontal edges contains at least ℓ edges. Therefore the weight (with respect to $w(e)$) of this set of edges combined with the weight of the jump is at least $\ell+1$. A similar argument holds for vertical jumps. Therefore,

$$w(c) \geq (\ell+1) \cdot \|\{e \in c : e \text{ is a jump }\}\|.$$

The rest of the proof follows the proof of claim 5.3. □

Equation 5.1 holds for cycles in the two-dimensional case, and the rest of the proof of the theorem follows the proof of theorem 5.1. □

5.6 Hardware overheads of the proposed solution

In this section we discuss the hardware overhead required by the solution proposed in this paper. The hardware overhead consists of three parts:

1. A segment of $O(d+s)$ cells is duplicated for every interconnection between cells, where d denotes the delay of the interconnection and s denotes the clock skew (these parameters are measured in clock periods).
2. An addition of an n-demultiplexer and n-multiplexer is required for every interconnection between chips, where $n = d + 2 \cdot s + 1$. These components can be realized with $O(k \cdot n)$ area, where k denotes the breadth of the interconnection between chips.
3. An increase in the number of pins used for interconnections between chips is required. Our solution increases the number of pins needed for every interconnection between chips by a factor of $n+1$, where $n = d + 2 \cdot s + 1$.

Although the number of pins is usually a scarce resource, it is definitely not the case with systolic linear arrays. If longer clock periods are good enough, then the number of pins can be decreased by increasing the clock period. The reason for this is that the delay and clock skew are measured in clock periods.

References

[1] A. J. Atrubin. A one-dimensional real-time iterative multiplier. *IEEE Trans. on Electronic Computers*, EC-14(3):394–399, Oct 1965. See also "Serial Digital Multiplier," United States Patent, Patent Number 3,407,290 , Oct. 1968.

[2] H. B. Bakoglu. *Circuits, interconnections, and packaging for VLSI.* Addison-Wesley, Reading, Mass., 1990.

[3] S. Brookes. Using fixed-point semantics to prove retiming lemmas. *Formal Methods in System Design*, 2:73–91, 1993.

[4] S. N. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Trans. on Computers*, C-18:349–365, Apr. 1969.

[5] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5, 1971.

[6] P. E. Danielsson. Serial/parallel convolvers. In Jr. E. E. Swartzlander, editor, *Systolic Signal Processing Systems.* Dekker, 1987.

[7] Guy Even. A new proof of the retiming lemma and applications (preliminary version). Technical report, #762, Computer Science Dept., Technion, 1993.

[8] Guy Even and Ilan Spillinger. Facilitating the problem of finding initializations after retiming: definitions, heuristic cost functions, and algorithms. Technical Report RC 19525, IBM Research Division, T.J. Watson Research Center, March, 1994.

[9] Shimon Even. *Graph Algorithms.* Computer Science Press, Rockville, Md., 1979.

[10] Shimon Even. Systolic modular multiplication. In A. J. Menezes and S. A. Vanstone, editors, *Advances in Cryptology - Crypto '90 Proceedings*, Lecture Notes in Computer Science 537. Springer, 1990. See also "Systolic Array For Modular Multiplication," United States Patent, Patent Number 5,101,431 , Mar. 31, 1992.

[11] Shimon Even, April 1992. Mid-term exam in Graph Algorithms, question 4, T.A.'s I. Ben-Aroya, R. Canetti, M. Frances, A. Orni, L. Zosin, (in Hebrew), Technion, Haifa.

[12] Shimon Even and Ami Litman. A systematic design and explanation of the atrubin multiplier. June 1991. presented in Sequences, Positano, Italy, June 1991. To appear in the proceedings (Springer-Verlag).

[13] Shimon Even and Ami Litman. On the capabilities of systolic systems. *Mathematical Systems Theory*, 27:3–28, 1994. a preliminary version appeared in *Proc. 3rd Annual ACM Symp. on Parallel Algorithms and Architectures,* Hilton Head, South Carolina, pp. 357-367, July 21-24, 1991.

[14] Allen L. Fischer and H. T. Kung. Synchronizing large VLSI processor arrays. *IEEE trans. on Computers*, C-34(8):734–740, Aug. 1985.

[15] S. M. German and Y. Wang. Formal verification of parameterized hardware designs. In *Proc. of the IEEE International Conference on Computer Design: VLSI in Computers*, pages 549–552, 1985. Reprinted in, Yoeli, M., "Formal Verification of Hardware Design," IEEE Computer Society Press, pp. 85-88, 1990.

[16] F. C. Hennie III. *Iterative Arrays of Logical Circuits.* MIT Press Research Monographs. MIT Press and John Wiley Sons, Inc., 1961.

[17] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading Mass., 1979.

[18] H. T. Kung. Why systolic architectures? *IEEE Computer Magazine*, 1982.

[19] C. E. Leiserson. *Area-Efficient VLSI Computation.* ACM Doctoral Dissertation Award Series, 1982. MIT Press, Cambridge, Mass., 1983.

[20] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In R. Bryant, editor, *Third Caltech Conference on Very Large Scale Integration*, pages 87–116. Computer Science Press, 1983.

[21] 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.

[22] C. E. Leiserson and J. B. Saxe. A mixed-integer linear programming problem which is efficiently solvable. *Journal Of Algorithms*, 9:114–128, 1988.

[23] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[24] A. Litman, 1992. private communication.

[25] G. M. Megson. *An Introduction to Systolic Algorithm Design.* Oxford Univ. Press, Oxford, 1992.

[26] M. C. Papaefthymiou. Understanding retiming through maximum average-delay cycles. In *3rd ACM Symposium on Parallel Algorithms and Architectures*, 1991. to appear in Mathematical Systems Theory.

[27] J. B. Saxe. *Decomposable Searching Problems and Circuit Optimization by Retiming: Two Studies in General Transformations of Computational Structures.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, Aug. 1985.

[28] J. I. Seiferas. Iterative arrays with direct central control. *Acta Informatica*, 8:177–192, 1977.

[29] C. Seitz. *System timing*, chapter 7. Addison-Wesley, Reading, Mass., 1980. In *Introduction to VLSI systems* by C. Mead and L. Conway.

[30] L. A. Taylor. Multipurpose serial/parallel multiplier. United States Patent, Patent Number 3,610,907, Oct. 1971.

[31] H. J. Touati and Brayton R. K. Computing the initial states of retimed circuits. *IEEE Trans on CAD*, 12(1):157–162, Jan 1993.

[32] A. Vergis and K. Steiglitz. Testability conditions for bilateral arrays of combinational cells. *IEEE Trans. on Computers*, C-35(1):13–22, Jan 1986.