

**TSFPGA: A Time-Switched
Field-Programmable Gate Array**

by

Derrick Chen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

© Derrick Chen, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis document in whole or in part, and to grant
others the right to do so.

Author.....
Department of Electrical Engineering and Computer Science
May 28, 1996

Certified by.....
Dr. Thomas F. Knight
Senior Research Scientist
Thesis Supervisor

Accepted by.....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 11 1996

EngA

TSFPGA: A Time-Switched Field-Programmable Gate Array

by

Derrick Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 1996, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Field-Programmable Gate Arrays (FPGAs) are computational devices containing uncommitted logic and interconnect resources which users configure for the applications they wish to run. Because the potential applications for FPGAs are not known at the time that they are fabricated, manufacturers of FPGAs should ideally provide abundant amounts of both logic capacity and interconnect flexibility to suit the widest possible range of applications. Current FPGAs, however, are lacking in interconnect flexibility because they neglect the temporal dimension of flexibility and exploit only the spatial aspect.

Unlike conventional FPGAs, the Time-Switched Field-Programmable Gate Array (TSFPGA) capitalizes on both the spatial and temporal aspects of flexibility by utilizing a time-switched interconnect architecture. The TSFPGA applies its time-switched interconnect to multiplex the physical switch and wire resources so that any particular switch or wire can replicate the functionality of n switches or wires over $O(n)$ amount of time units. Under this scheme, time serves as an additional dimension for augmenting spatial interconnect resources because it allows the TSFPGA to emulate connectivity between logic blocks temporally instead of providing that connectivity physically. By reusing the limited number of physical switches and wires, the TSFPGA promises greater interconnect flexibility than that possible in existing FPGA architectures.

Thesis Supervisor: Dr. Thomas F. Knight
Title: Senior Research Scientist

Acknowledgments

Foremost, I would like to thank my supervisors, André DeHon and Dr. Thomas F. Knight, for providing me with this opportunity to realize my dreams of VLSI design. I would also like to thank my officemate, Edward Tau, for his company and spirited humour during my many long nights of work on the TSFPGA. Lastly, I would like to thank my parents for their constant support over the past five years of my education at MIT.

This research is supported by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

Contents

1	Introduction	13
1.1	The Benefits of Gate Array Technology	13
1.2	The Advent of Field-Programmable Gate Arrays	14
1.3	The Problems With Field-Programmable Gate Arrays	14
2	The Time-Switched Field-Programmable Gate Array	17
2.1	Solution	17
2.2	Prior Work	18
3	Architecture	21
3.1	Control	21
3.2	Logic	22
3.2.1	Function Generator	22
3.2.2	Array Element	24
3.2.3	Subarray	24
3.3	Interconnect	24
3.3.1	Complete Network	26
3.3.2	Manhattan Network	27
3.3.3	Multi-Stage Network	27
3.3.4	Context Memory	28
3.4	Operational Interface	28
3.5	Configuration Interface	29
4	Implementation	31
4.1	Strategy	31
4.2	Environment	32
4.3	Look-Up Table	32
4.4	Input Registers	35

4.5	Comparators	38
4.6	Function Generator	41
4.7	Array Element	41
4.8	Crossbar	46
4.9	Multiplexer Interface	49
4.10	Context Memory	49
4.11	Subarray	54
5	Evaluation	59
5.1	Logic Area Utilization	59
5.1.1	Context Memory	61
5.1.2	Array Element	61
5.2	Interconnect Area Utilization	62
5.2.1	Configuration Interface	65
5.2.2	Routing Channels	65
6	Conclusion	67

List of Figures

3-1	TSFPGA Logic Architecture	23
3-2	TSFPGA Interconnect Architecture	25
4-1	LUT Schematic	33
4-2	LUT Layout	34
4-3	Input Registers Schematic	36
4-4	Input Registers Layout	37
4-5	Comparators Schematic	39
4-6	Comparators Layout	40
4-7	Function Generator Schematic	42
4-8	Function Generator Layout	43
4-9	Array Element Schematic	44
4-10	Array Element Layout	45
4-11	Crossbar Schematic	47
4-12	Crossbar Layout	48
4-13	Multiplexer Interface Schematic	50
4-14	Context Memory Schematic	51
4-15	Context Memory Layout	52
4-16	Subarray Schematic	55
4-17	Subarray Layout	56

List of Tables

5.1	TSFPGA Logic Area Utilization	60
5.2	TSFPGA Interconnect Area Utilization (Part 1)	63
5.3	TSFPGA Interconnect Area Utilization (Part 2)	64

Chapter 1

Introduction

The application of gate array technology in the production of integrated circuits permits a number of cost reductions over custom-design methods. Section 1.1 summarizes the major cost benefits of utilizing this technology. Section 1.2 points out that the consequent development of field-programmable gate arrays (FPGAs) to replace traditional gate arrays has broadened the appeal of this technology to larger audiences while maintaining its advantages. However, as Section 1.3 explains, FPGAs do not currently make highly effective use of the resources available in integrated circuits. This thesis presents a new FPGA architecture which promises greater utility than that provided by conventional FPGAs.

1.1 The Benefits of Gate Array Technology

In their continual search for cheaper means of production, manufacturers of integrated circuits have increasingly converted from custom-design methods to gate array technology. Through gate array technology, manufacturers can achieve a combination of material and labor savings not possible with previous techniques [WE92].

While custom-design methods generally require different integrated circuits to be uniquely specified at all levels, leaving little opportunity for reuse, gate array technology permits the sharing of a single base wafer across multiple circuits for material savings. Not only does reusing a single wafer enable manufacturers to mass-produce those wafers for high availability while maintaining low amortized cost per wafer, it also allows the standardization of bonding patterns and associated packaging requirements across different integrated circuits, both of which contribute to further lowering the material costs of production.

In addition to material savings, manufacturers can also achieve labor savings by using gate array technology to reduce the amount of human work needed to produce different integrated circuits in comparison with custom-design methods. This reduction in work comes from two optimizations:

one, since the base wafer of gate arrays is predetermined, personalizing it for different integrated circuits requires manufacturers to generate only the subset of layers needed to describe the upper-level metalization of each circuit, and two, gate array technology allows automated tools to take advantage of the predictability of standardized components to manage the majority of the time-consuming work in performing placement, routing, and testing of integrated circuits.

1.2 The Advent of Field-Programmable Gate Arrays

While the combination of material and labor savings made possible by gate array technology has allowed manufacturers to produce integrated circuits at lower costs relative to custom-design methods, the nature of traditional gate arrays as a manufacturing technology limits its accessibility to only those individuals directly involved in the production of integrated circuits. Hence, only a small number of designers working at the lowest levels of hardware are able to reap the lower-cost benefits of gate arrays.

The more-recent development of FPGAs seeks to maintain the material and labor savings of traditional gate arrays while extending these advantages to a broader range of users, including system designers and application developers. Like traditional gate arrays, FPGAs are standard parts which can be mass-produced for material savings and manipulated through automated tools for labor savings. But rather than having their function predetermined during the manufacturing process as in the case of traditional gate arrays, FPGAs provide uncommitted logic and interconnect resources which system designers or application developers can configure at the time of use to personalize the integrated circuits for particular applications. By providing end-user programmability, FPGAs have made material and labor savings accessible to a larger audience than that of traditional gate arrays.

1.3 The Problems With Field-Programmable Gate Arrays

Because the potential uses for FPGAs are not known at the point of their manufacture, FPGAs need to provide abundant configurable resources in order to achieve general utility across the widest possible range of applications. At one extreme, systolic-style applications require high logic capacity for optimal pipelining but only limited interconnect flexibility for passing data between neighboring pipeline stages. At the other extreme, reduction-style applications require expansive interconnect flexibility to facilitate communication between distant processing elements but do not necessarily require high logic capacity. To accommodate both sets of potential applications as well as the majority of applications between the two extremes, FPGAs must incorporate both ample logic capacity and interconnect flexibility.

However, despite the apparent need for both logic capacity and interconnect flexibility, major

manufacturers of FPGAs, such as Xilinx and Altera, have historically promoted their products primarily on the basis of the former requirement while deemphasizing the latter necessity, and as a result, interconnect flexibility has become the limiting factor in the general utility of FPGAs today. For example, due to their limited interconnect flexibility for accommodating typical applications, Xilinx XC2000 and XC3000 series FPGAs are only able to achieve 60% utilization of peak logic capacity [Xil93]. In general, FPGAs with limited interconnect flexibility cannot capitalize on their full logic capacity to effectively serve a wide range of potential applications.

To compound the problem of low logic utilization, limited interconnect flexibility also has the undesirable effect of forcing mapping programs which compile applications into configuration information to perform more extensive searches over the architectural space of FPGAs in order to generate potential translations. Because the mapping process for conventional FPGAs is a NP-complete problem, broader and more in-depth searches necessarily require exponentially increasing amounts of compile time, thus further reducing the utility of current FPGAs.

While practical experience suggests that expanding interconnect flexibility is desirable, current FPGAs already dedicate 75%-90% of their available area to interconnect so that applying more area to improve interconnect flexibility is not generally feasible [CD96]. As a consequence, FPGAs must attempt to provide greater interconnect flexibility within their existing interconnect area.

This thesis introduces a new FPGA architecture, named the Time-Switched Field-Programmable Gate Array (TSFPGA), which incorporates a different interconnect scheme than that provided by conventional FPGAs to address the disparity between desiring greater interconnect flexibility for improved utility and interactivity and desiring more efficient use of available interconnect resources.

Chapter 2

The Time-Switched Field-Programmable Gate Array

Section 2.1 presents one solution to the problem of improving the interconnect utilization of conventional FPGAs. It hinges upon the observation that FPGAs currently exploit only one dimensional of capacity, and thus, their existing interconnect structures do not provide nearly the flexibility that is possible when taking full advantage of available capacity as in the case of the TSFPGA. Section 2.2 provides an alternative development of the ideas behind the TSFPGA in terms of prior work on the Dynamically-Programmable Gate Array (DPGA).

2.1 Solution

Conventional FPGAs require inordinate amounts of interconnect resources to adequately handle typical applications possibly because of a misunderstanding on the part of their manufacturers of how to characterize the interconnect capacity of FPGAs. While traditional definitions of capacity have focused on measures of density, in actuality, capacity should be measured in terms of bandwidth, which encompasses both spatial and temporal dimensions. Because the design of conventional FPGAs considers only the spatial aspect, existing FPGAs under-utilize the interconnection bandwidth that is available and, as a result, require excessive resources to accommodate practical applications.

Without provisions for temporal use, conventional FPGAs must employ a static assignment of signals to specific switches and wires in their interconnect. For general applications, this static assignment results in only a few percent of the interconnect at any point in time actively performing useful functionality in transmitting data from one logic element to another. The majority of the switches and wires in the interconnect are idle because they are either not being used at all in the application, or not yet used and waiting to take part in the transmission of data, or already used in

the current computation and waiting for the next set of data to arrive [CD96].

Unlike conventional FPGAs, the TSFPGA capitalizes on the temporal aspect of capacity by introducing a time-switched interconnect architecture. The TSFPGA uses time-switched interconnect to multiplex the physical switch and wire resources so that any particular switch or wire can replicate the functionality of n switches or wires over $O(n)$ amount of time units. Under this scheme, time serves as an additional dimension for augmenting spatial interconnect resources because it allows the TSFPGA to emulate connectivity between logic blocks temporally instead of providing that connectivity physically. By reusing the limited number of physical switches and wires, the TSFPGA extracts greater routing capacity from its interconnect than that possible in existing FPGA architectures [Chen96].

As a result of capitalizing on both the spatial and temporal aspects of capacity in its interconnect architecture, the TSFPGA provides a number of advantages over conventional FPGAs:

- **High Interconnect Utilization:** While conventional FPGAs must use a static assignment of signals in which any given switch or wire serves at most one signal for the duration of an application, time-switched interconnect in the TSFPGA can allocate multiple signals to each switch or wire over the running time of an application for higher interconnect utilization.
- **High Logic Utilization:** Whereas conventional FPGAs can suffer from poor logic utilization due to their limited interconnect flexibility, the TSFPGA can map applications which require greater connectivity than that provided in its physical resources by extending its translation of those applications into the time dimension, so that the interconnect requirements of the applications do not constrain their logic utilization.
- **Fast Mapping Compilation:** While conventional FPGAs can suffer long delays in their mapping process because of their limited interconnect flexibility, the TSFPGA takes advantage of the greater connectivity provided by its time-switched interconnect to produce faster translations of applications.
- **Flexible Optimization:** By using both spatial and temporal aspects of capacity, the TSFPGA, unlike conventional FPGAs, allows the user to trade-off one dimension for the other. When time is at a premium, the TSFPGA can pursue fast mapping compilation, but when space is the critical resource, the TSFPGA can attempt to provide a denser mapping at the cost of longer compute times.

2.2 Prior Work

The inspiration for pursuing the TSFPGA was based upon previous research on the DPGA. The goal of the DPGA project was to improve the functional diversity of conventional FPGAs by incorporating

an incremental amount of memory for storing multiple configurations simultaneously. The original assumption was that the multiple configurations would allow the DPGA to reuse its logic elements to manage a larger variety of functions than that possible with existing FPGAs. However, the unexpected conclusion of the DPGA project showed that the true advantage of the DPGA approach was not in the reuse of the logic elements, but rather, in the reuse of the interconnect resources because the interconnect, not the logic, dominates the area of conventional FPGAs [Tau96]. Building upon the experience gained in the design and construction of the DPGA, the TSFPGA seeks to improve existing FPGAs by providing a new architecture which focuses on extracting the maximum utility possible from the interconnect.

Chapter 3

Architecture

Each section in this chapter presents a particular facet of the overall architecture of the TSFPGA. Section 3.1 explains the control system which governs the global operation of the TSFPGA. Section 3.2 describes the organization of the logic elements which perform the processing needed by applications. Section 3.3 covers the interconnect structures which transmit data between different logic elements. Section 3.4 discusses the operational interface for communication between the TSFPGA and external devices. Section 3.5 presents the programming interface through which the user configures the TSFPGA for applications.

3.1 Control

The control architecture of the TSFPGA is based upon a mechanism borrowed from SIMD architectures [BDK93]. The defining characteristic of SIMD architectures is the incorporation of some means for generating and broadcasting a single instruction stream to all processing elements so that all elements perform the same operations in-step. The TSFPGA uses a modified version of this global signal generation and distribution model to broadcast two values: a time and a context, instead of a SIMD instruction. The time is an eight-bit quantity used to control the operation of the logic elements, while the context is a six-bit quantity used to control the operation of the interconnect. The details of how the time and context quantities interact with the logic elements and interconnect respectively are explained in Section 3.2 and Section 3.3.

Rather than broadcast a single value as in SIMD architectures, the TSFPGA maintains a distinction between the time and context quantities to separate the amount of time that the logic elements need to compute the results of functions that they implement from the number of contexts that the interconnect needs to transfer new values or computed results between different logic elements. Ultimately, this distinction gives the architecture the flexibility of supporting a greater or fewer number of time-steps independent of adjustments to the number of contexts.

3.2 Logic

The logic architecture of the TSFPGA is organized in a three-level hierarchy shown pictorially in Figure 3-1. At the lowest level are individual function generators which serve as the fundamental processing units of the TSFPGA by each being able to implement any arbitrary four-input, one-output function. At the intermediate level, groups of function generators form array elements, and at the top of the hierarchy, groups of array elements combine into subarrays.

3.2.1 Function Generator

Each function generator is composed of three components: a look-up table (LUT), input registers, and comparators.

Look-Up Table

The core of the function generator is a LUT which computes the result of the four-input, one-output function. The decision to use a four-input, one-output LUT in the TSFPGA was based on prior work at the University of Toronto which concluded that such LUTs produced the most area efficient designs across a number of circuit benchmarks [RFLC90]. Although later work revised the conclusion to suggest that a larger five- or six-input LUT was a better choice, the TSFPGA maintains the original four-input, one-output LUT to leverage existing software tools, such as Chortle [Fra92], which already provide mapping capabilities for such LUTs.

Input Registers

Every function generator includes four registers to supply input values to its LUT. Contrary to the typical function generators found in some existing FPGAs, the generators in the TSFPGA contain input registers, rather than output ones. Input registers are needed by the time-switched interconnect in the TSFPGA because they allow the individual function generators to latch-in data locally, thereby freeing the switches and wires which delivered the data for use in transmitting other signals. Although input registers force the TSFPGA architecture to use four times as many registers as comparable output register architectures, making the TSFPGA less area efficient than conventional FPGAs in this regard, the additional logic area cost of the extra registers is offset by the interconnect area savings of time-switched interconnect.

Comparators

Each of the four input registers in the function generator is controlled by an associated comparator. At every time-step, each comparator checks the globally-broadcast time quantity against a locally-stored value to determine whether or not the two values match. When the comparator detects a

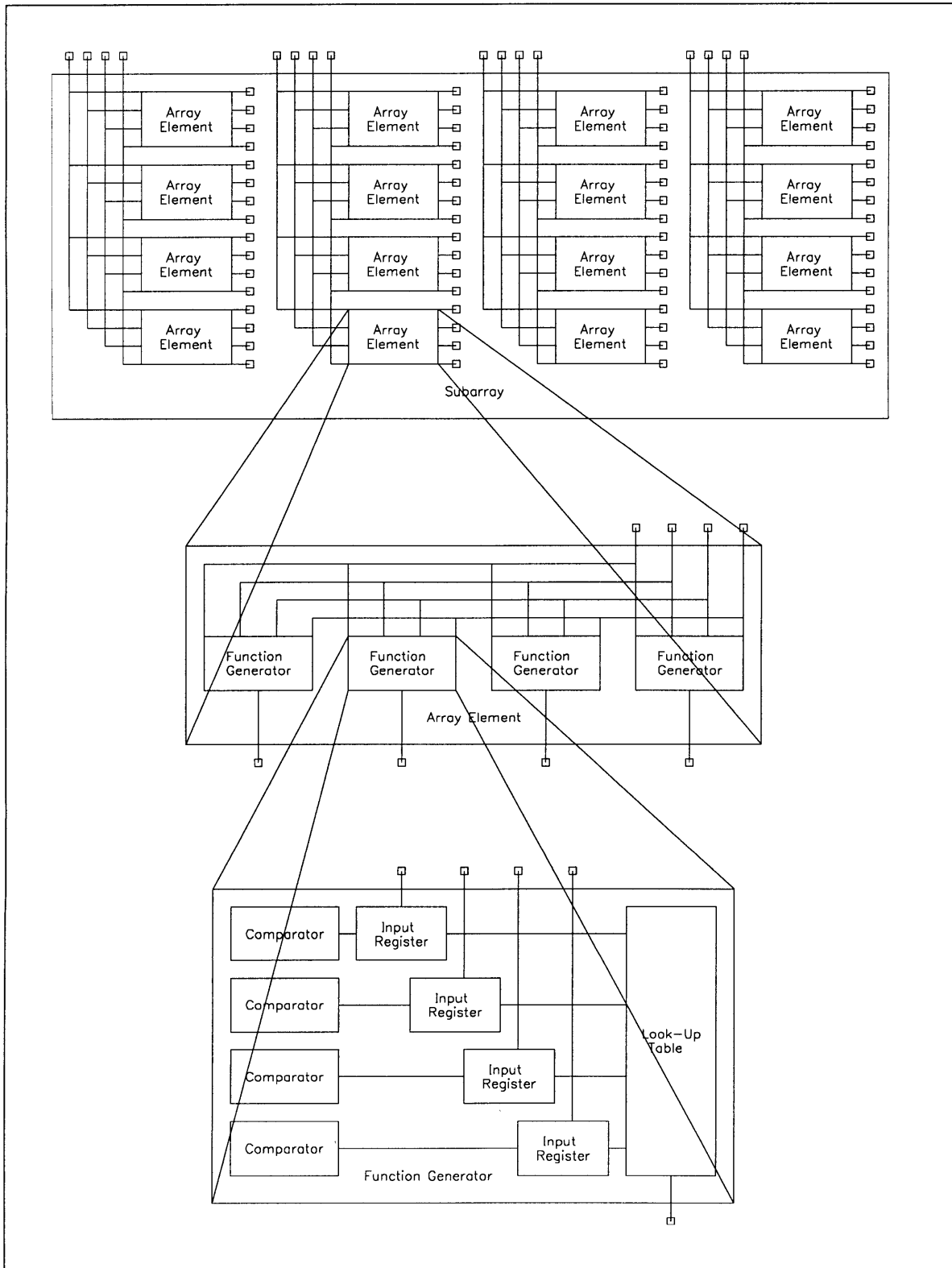


Figure 3-1: TSFPGA Logic Architecture

match, it enables its associated register so that the register can latch-in new data from the interconnect. When the two values mismatch, the comparator disables its associated register to prevent it from receiving data. By enabling and disabling the input registers, the comparators, operating in response to the global time, are able to selectively determine which data its function generator should process.

3.2.2 Array Element

Each array element is a four-input, four-output logic block formed from groups of four function generators. Each generator of an array element receives a copy of the four incoming values and computes one of the outgoing results.

The principal motivation for grouping generators together is to allow the TSFPGA to mimic the organization of the DPGA which uses array elements, instead of individual function generators, as its fundamental processing units [BCE⁺94]. The sharing of input values, however, does also allow an optimization in which multiple function generators receive common input values simultaneously to exploit the inherent locality and hierarchy of computation.

3.2.3 Subarray

Subarrays, each of which contains sixteen array elements, are a further level of abstraction in the logic hierarchy. Each subarray receives sixteen inputs and produces sixty-four outputs. The sixteen input bits of a subarray are distributed in four buses of four bits each, with each bus connecting to groups of four array elements so that each array element of a group receives the same input signals as the other array elements in that group. The sixty-four output bits of a subarray are the aggregate outputs of each of the sixty-four function generators in that subarray.

The primary purpose of this final level of abstraction in the logic hierarchy is to form the fundamental tiling block of the TSFPGA. To accommodate large applications requiring more logic than that provided by a single subarray, the TSFPGA simply uses a matrix of subarrays instead of introducing a fourth logic level. Although the matrix can actually be arbitrarily large, for the purposes of explanation, references to multiple subarrays within the context of this thesis assume a four-by-four tiling of subarrays.

3.3 Interconnect

The interconnect architecture of the TSFPGA is organized in a three-level hierarchy diagrammed in Figure 3-2. At the lowest level is a complete network embedded within each of the subarrays of the TSFPGA. Each complete network interconnects the sixteen array elements of its associated subarray. At the intermediate level is a manhattan network partially connecting the multiple subarrays which

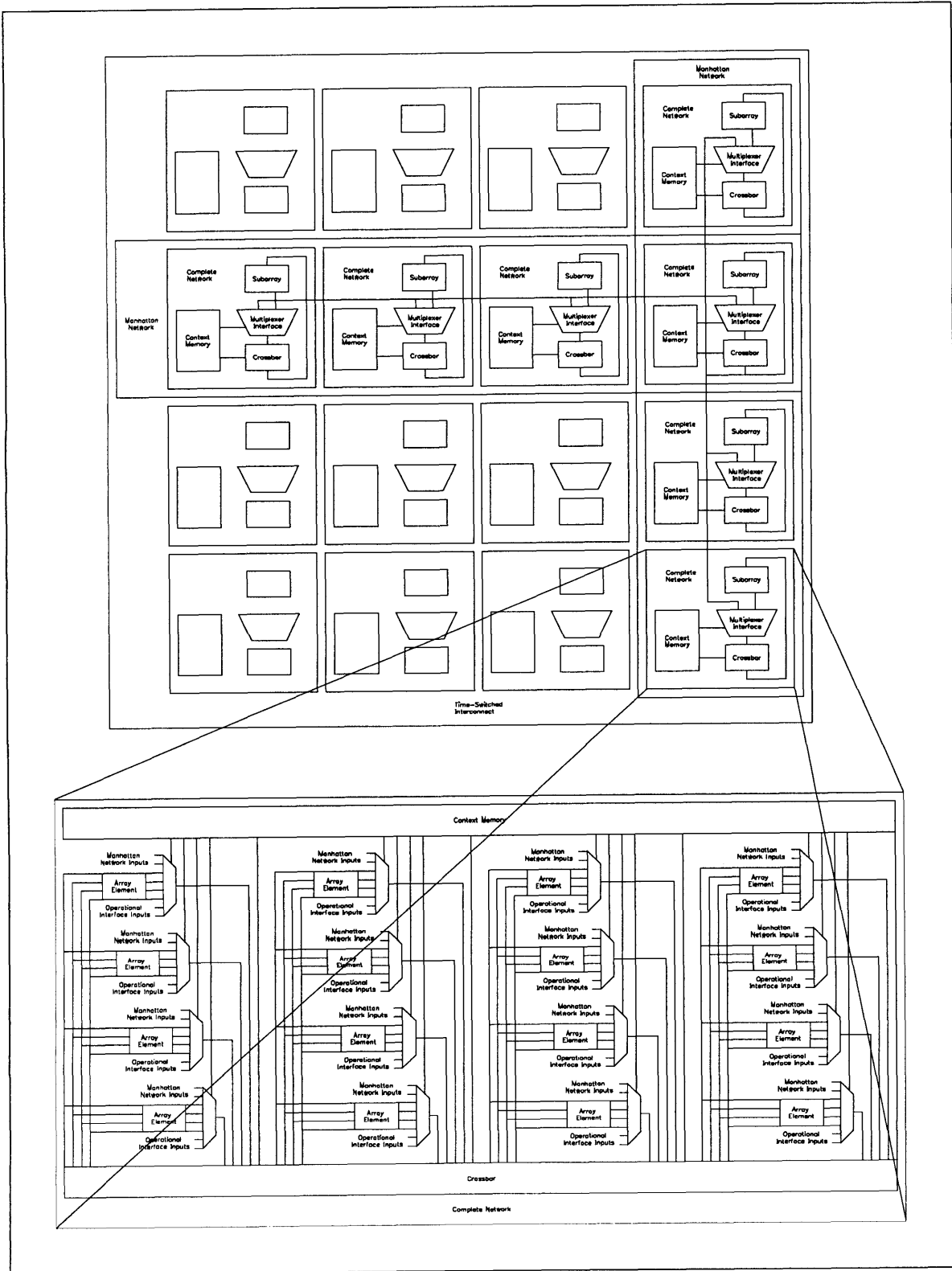


Figure 3-2: TSFPGA Interconnect Architecture

constitute the full TSFPGA. The combination of the two levels forms a multi-stage network at the top of the interconnect hierarchy linking all logic elements and allowing time-switched interconnect.

3.3.1 Complete Network

The complete network within each subarray is composed of two components: a sixteen-by-sixteen crossbar and a multiplexer interface.

Crossbar

Given that the goal of the TSFPGA is to extract greater interconnect flexibility from fewer interconnect resources than that needed by conventional FPGAs, the use of a crossbar to connect the array elements of a subarray may appear to be counterproductive at first glance. While crossbars are an ideal choice for maximum interconnect flexibility because they allow full connectivity between each of their inputs and outputs, they theoretically provides such connectivity inefficiently by, in general, requiring $O(n^2)$ switches and wires to link any given n logic elements.

This observation, however, accentuates the difference in the orders of growth of the interconnect area versus the logic area by abstracting away the constant factors associated with the two quantities being compared. Within a subarray, the constant factors do make a significant difference and should be considered. Because a single array element is typically a hundred to a thousand times the size of a switch or wire [BCE⁺94], the constant factors overshadow the differences in the two orders of growth in situations where the number of array elements being interconnected is small. Thus, crossbars can be feasible means of interconnect when used conservatively. While using a sixty-four-by-sixty-four crossbar to directly interconnect all sixty-four function generators of a subarray is probably unrealistic, a sixteen-by-sixteen crossbar to interconnect the sixteen array elements is practical.

To actually link the array elements of a subarray, two sets of connections are necessary. On one side, the sixteen outputs of the crossbar connect directly to the sixteen inputs of the subarray. On the other side, the sixteen inputs of the crossbar connect to the sixty-four outputs of the subarray via a multiplexer interface.

Multiplexer Interface

To interface between the sixteen inputs of the crossbar and the sixty-four outputs of the array elements in a subarray, the complete network utilizes sixteen eight-to-one multiplexers. Two of the eight inputs to each multiplexer are needed by the manhattan network and their use is explained in Section 3.3.2. Two other inputs are used by the operational interface of the TSFPGA described in Section 3.4. The remaining four inputs of each multiplexer attach to the four outputs of each array element in the subarray so that every output of a subarray connects to one multiplexer input.

In turn, the outputs of the sixteen multiplexers connect to the sixteen inputs of the crossbar to complete the interface.

A multi-context memory, explained in Section 3.3.4, regulates the operation of both the crossbar and the multiplexers to determine which of the outputs of the array elements can pass their values via the multiplexers to the crossbar inputs and what permutation of inputs arrives at the crossbar outputs.

3.3.2 Manhattan Network

The TSFPGA utilizes a manhattan network to allow array elements in a matrix of subarrays to communicate across subarray boundaries. The manhattan network is a series of wires linking the outputs of the crossbar in each complete network to the multiplexer interface inputs of every complete network in the crossbar's row or column. More specifically, using a four-by-four matrix of complete networks as an example, the manhattan network divides the sixteen outputs of each crossbar into four sets of four values and associates two complete networks, one in the same row as the crossbar and the other in the same column as the crossbar, with each of the four sets of values. Each complete network in this example, then, is the sender of sixteen outputs and the recipient of thirty-two inputs from the manhattan network.

3.3.3 Multi-Stage Network

Together, the interconnect resources of the complete and manhattan networks form the multi-stage network providing time-switched interconnect in the TSFPGA. The crossbar and multiplexer interface of the complete network act as the switches for time-switched interconnect, while the manhattan network serves as the wires.

Because the multi-stage network is composed of both the complete and manhattan networks, framing its explanation in terms of the functionality provided by the two hierarchically lower networks aids in clarifying its purpose. Conceptually, the complete network is a zero-dimensional interconnect because it transfers data within one location, the subarray. In contrast, the manhattan network is a one-dimensional interconnect because it transfers data along a line, either the vertical or horizontal axis in a matrix of subarrays. Continuing to expand upon the dimensionality, the multi-stage network is a two-dimensional interconnect allowing any subarray of a matrix to communicate with any other subarray in that matrix.

It follows naturally then that transmitting data across the two-dimensional multi-stage network is performed through an extension of the means used by the complete and manhattan networks to communicate information. In the complete network, data moves from the output of an array element, through the multiplexer interface, to the input of the crossbar, which then switches the data to the appropriate output path and passes it to the receiving array element. In the manhattan network,

data moves from the output of a array element, through the multiplexer interface, to the input of the crossbar, which then switches the data to the appropriate manhattan wire. The manhattan wire transmits the data to the destination subarray where it passes through the multiplexer interface of that subarray and enters the input of the crossbar within that subarray. That crossbar then performs the final switching needed to permute the data to the correct destination array element. Extending this pattern, the multi-stage network passes data from the sender function generator through three phases of switching and two phases of interconnect traversal, one phase along the vertical axis and the other phase along the horizontal axis, to deliver the data to the recipient function generator.

The multi-stage network acts as a time-switched interconnect due to its reuse of common resources, most notably the crossbar. Over the course of multi-stage network operation, the crossbar performs three different functions. One, it acts as an output device transmitting data from array element outputs to intermediate switching points. Two, it behaves as a routing device moving data between source and destination subarrays. Three, it also serves as an input device receiving data from intermediate switching points and forwarding it to the appropriate destination array element.

3.3.4 Context Memory

To regulate the operation of its crossbars and multiplexer interfaces between the three different interconnect levels and across multiple phases of operation, the TSFPGA relies on a sixty-four word context memory. At each phase of operation, the six-bit, globally-broadcast context value described in Section 3.1 selects one of the sixty-four words stored in the context memory and applies that word to configure each of the crossbars and multiplexers in the TSFPGA. For each crossbar, the configuration in that word determines whether the crossbar will operate as a input crossbar, routing crossbar, or output crossbar as well as precisely how the input values to the crossbar will be permuted to arrive at the output values. For each multiplexer, the configuration determines whether the multiplexer will pass a value from its array element inputs, or from its manhattan network inputs, or from its operational interface inputs to its output.

3.4 Operational Interface

The operational interface of the TSFPGA is a series of time-switched input and output connections linking the internal logic elements to external circuitry. Input values are received via the two dedicated inputs in the multiplexer interface associated with each subarray, while output values are taken from the output of the crossbar in each subarray.

To control the input and output connections, the TSFPGA relies upon time-multiplexing techniques similar to the ones used in its internal logic architecture. Each connection is enabled and disabled by the result of a comparison between the globally-broadcast time value and a locally-stored

time value in a manner parallel to the comparison process in function generators.

Just as the comparators in the function generators allow the switches and wires which deliver data values to the generators to be reused temporally, the application of time-multiplexed input and output connections permits the operational interface to reuse its limited number of physical connections over time to extract greater utility from available resources.

3.5 Configuration Interface

Because the TSFPGA, like all FPGAs, is intended to be user-programmed, the configuration interface is as simple as possible. Rather than following the example of commercial FPGAs which employ proprietary interfaces [Xil93], the TSFPGA utilizes the simple and widely-understood configuration interface of random-access memories. Users can program the TSFPGA from scratch by writing values to a range of addresses, and once a configuration is installed, they can make incremental changes to select portions of the configuration without affecting the contents of other areas by using the random-access capabilities of the interface.

Besides providing random-access to configurations for ease of use, the TSFPGA also includes facilities for background programming of its configurations to accelerate interactive use. Conventional FPGAs which are not capable of background programming typically require several milliseconds to perform a complete programming of their configuration [Xil93]. During that time, the user is not able to make use of the FPGA. In contrast, the TSFPGA uses background programming to hide the delays incurred during programming by allowing the process to take place while the TSFPGA is actively running an application.

Chapter 4

Implementation

This chapter describes the implementations of the logic and interconnect architectures of the TSFPGA. The details of the global control and the operational and configuration interfaces are explained only as far as they affect the implementations of the two primary concerns.

Before explaining the logic and network implementations, Section 4.1 provides an overview of the basic principles guiding the approach to the implementation tasks and Section 4.2 explains the tools used to create the implementations. Section 4.3, Section 4.4, and Section 4.5 cover the fundamental LUT, register, and comparator building blocks of the logic implementation leading up to Section 4.6 and Section 4.7 which describe the construction of the function generator and the array element respectively. Section 4.8, Section 4.9, and Section 4.10 document the crossbar, multiplexer interface, and context memory components making up the interconnect implementation, which combined with the logic implementation, constitutes the subarray depicted in Section 4.11.

4.1 Strategy

The particular implementations of the logic and interconnect architectures described in this chapter are the product of several guiding principles derived from both engineering ideals and practical experience.

The foremost principle is the application of regularity. Each implementation of the components described in this chapter exploits the internal regularity of the structures within those components to allow reusability and to create predictability. Reusability is advantageous in minimizing the work needed to produce the individual components while predictability is beneficial for facilitating the integration of the components into complete logic and interconnect implementations of the TSFPGA.

A second principle is the preference for smaller implementations. Rather than pursuing futile attempts to characterize the entire space of potential implementations according to size, speed, and power criteria, the particular implementations described in this chapter were, to first order,

chosen on the basis of size considerations. In practice, smaller size implementations for a particular design style have a tendency, as a by-product, to also be among the higher-speed and lower-power implementations of that design style.

One last principle guiding the implementation of the TSFPGA is the avoidance of global circuitry. Previous experience in the construction of the DPGA showed that circuits at the top level of an implementation produce excessive organizational and verification difficulties which can render an entire implementation unworkable. To avoid repeating those difficulties in the TSFPGA, the implementation bias was to follow approaches which simplify the global circuitry as much as possible and which place the necessary remaining circuitry into lower levels of the implementation hierarchy when possible. In particular, portions of the control and configuration architectures are implemented inside the logic and interconnect components described in this chapter.

4.2 Environment

The logic and network implementations of the TSFPGA are intended to be manufactured in Hewlett-Packard's CMOS14 half-micron, three-layer metal VLSI fabrication process [HP94]. They were constructed using Cadence Design Systems EDA software running on a Sun workstation. Composer was used for schematic entry, Virtuoso for layout, and Diva for verification. All simulations were performed using Metaware's HSPICE software via the Cadence HSPICE interface and the STL language.

4.3 Look-Up Table

Because the four-input, one-output LUT used in the function generator of the TSFPGA is, in essence, a sixteen-bit memory, the implementation of the LUT is based on approaches to memory implementations. Memories can generally be categorized into one of two forms, either a decoder-style memory or a multiplexer-style memory [WE92]. Although the former style is the more typical approach to building memories, the LUT in the TSFPGA is actually implemented using the latter method which, by merging the decode and data paths, promises to be more compact. Additionally, rather than requiring one driver for each word of memory as in the case of the decoder-style memory, the multiplexer-style implementation can, in the extreme situation, factor all driver requirements into one driver at the output of the multiplexer to achieve further space savings.

A schematic view of the TSFPGA LUT is shown in Figure 4-1. The two parts of the diagram to note are the sixteen memory cells which determine the function of the LUT and the multiplexer which selects the output of the LUT.

The sixteen memory locations containing the LUT function are implemented in the form of

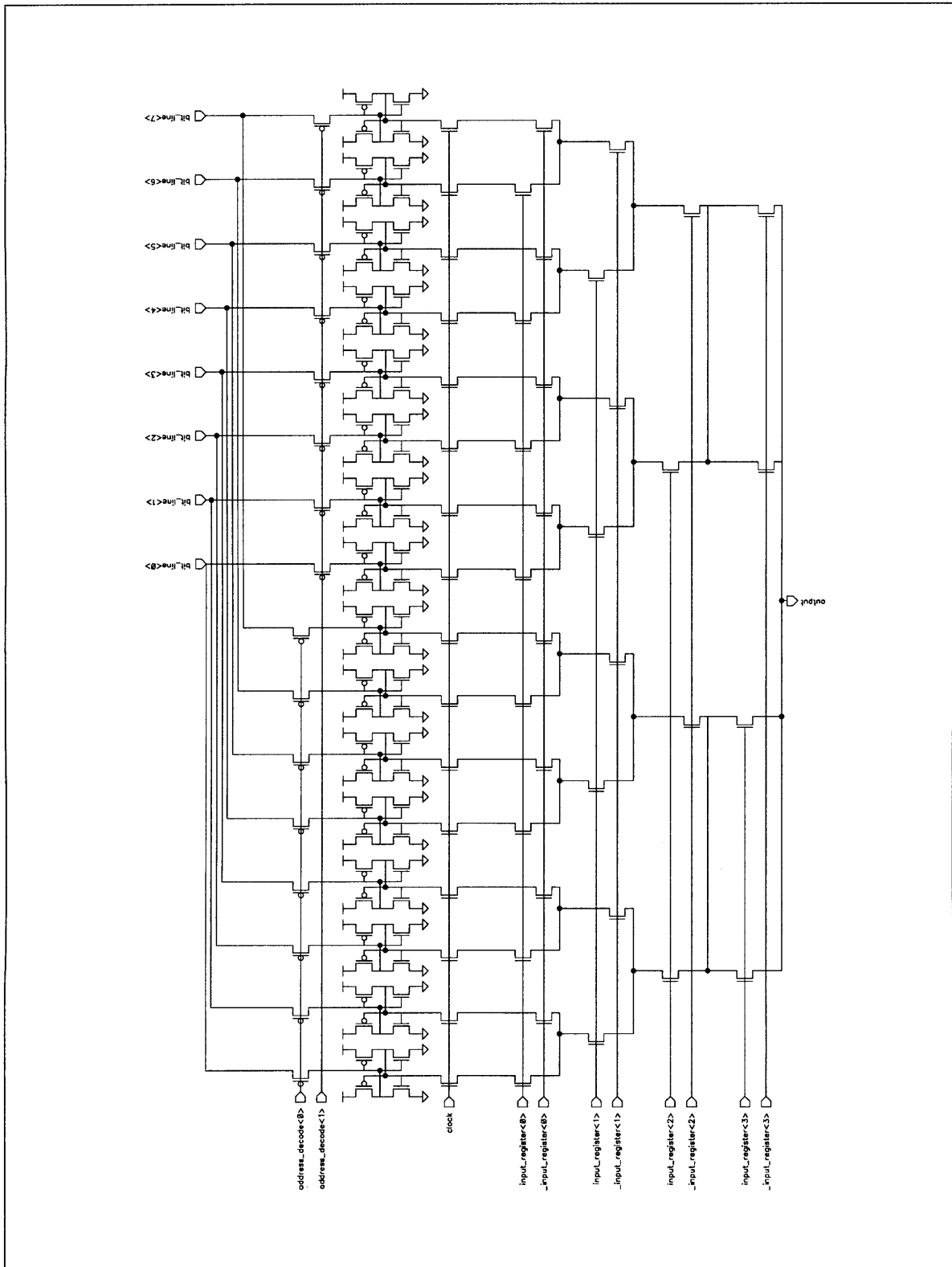


Figure 4-1: LUT Schematic

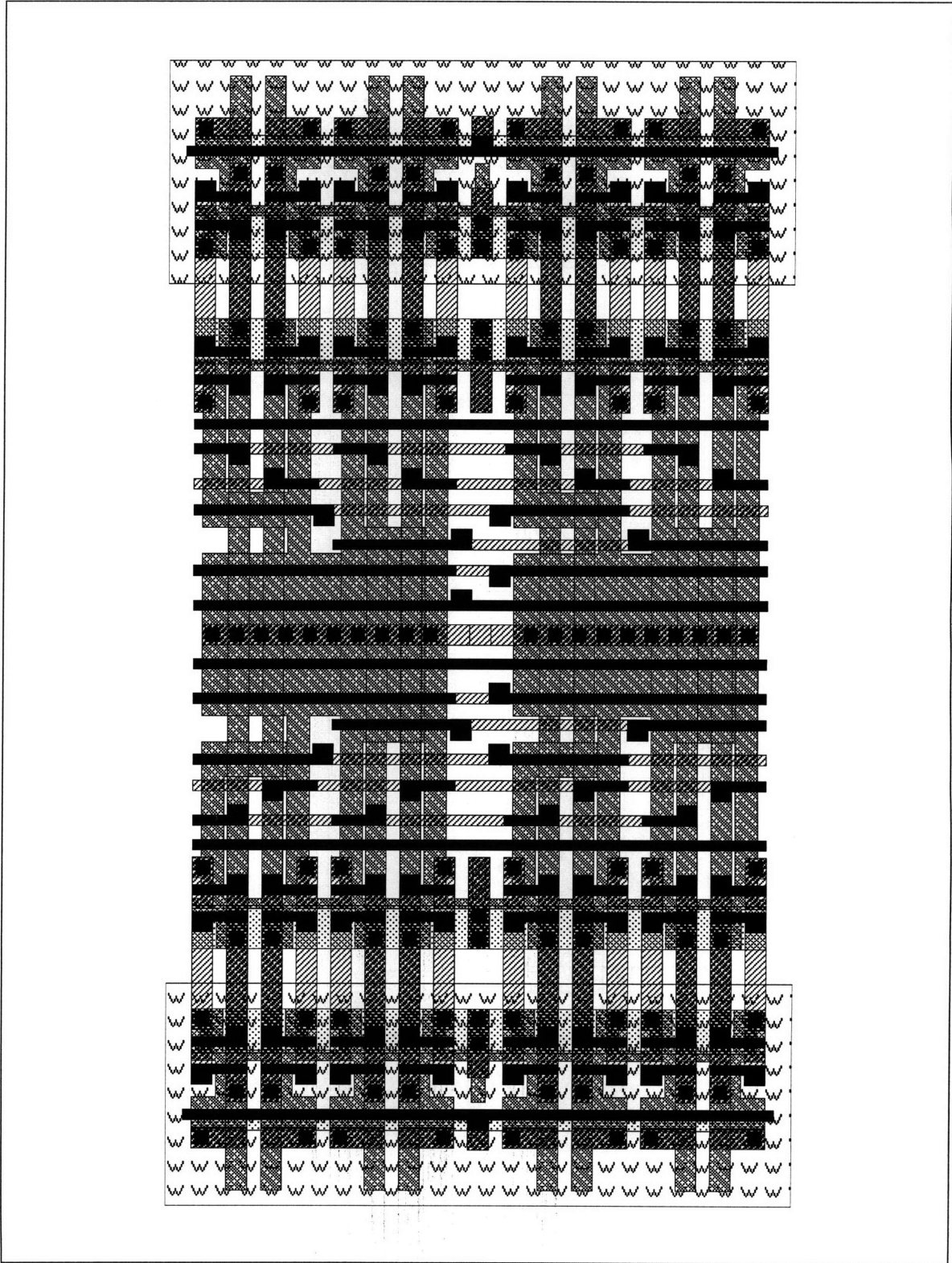


Figure 4-2: LUT Layout

sixteen SRAM cells. Taking the basic approach to SRAM cell implementation, each cell is a cross-coupled pair of inverters. Each SRAM cell has two pass-gate connections for reading and writing. The read connection attaches the SRAM cell to the inputs of the multiplexer using a NMOS transistor while the write connection links the SRAM cell to the bit-lines of the configuration interface through a PMOS gate. A global clock signal enables and disables the read connection to determine the operation of the LUT, and decoders of the configuration interface enable and disable the write connection to control the programming of the SRAM cell.

The multiplexer which selects the output of the LUT is a binary tree of NMOS pass-gates with a slight modification to split the root of the tree in order to facilitate a compact layout. Each of the four levels of the tree is controlled by a corresponding input register described separately in Section 4.4. In a modular design, the output side of the multiplexer would ideally end with a buffer to isolate the internal resistance and capacitance of the multiplexer, limit the extent of charge sharing, and build up the output drive capacity, but in this particular schematic, the output buffer has been converted into an input buffer for the next circuit stage and is shown as part of that schematic diagram in Figure 4-13.

The layout of the TSFPGA LUT is shown in Figure 4-2. Instead of being a direct translation of the schematic, the layout exploits the symmetry of the LUT by mirroring half of the SRAM cells and half of the multiplexer on either side of the LUT output, located in the center. An alternative approach to this layout of the LUT is to organize the SRAM cells and the multiplexer into an H-tree pattern. Given sufficient technology resources, the H-tree approach can produce a more compact layout than the current approach, but requires one metal layer for each level of the multiplexer in order to achieve maximum layout density. In contrast, the mirrored layout of the LUT uses only one layer of metal for the entire layout. Because the successive levels of the multiplexer do not overlap, the current approach is easily scalable and does not ever need more than one interconnect layer to be effective.

4.4 Input Registers

The input registers of the function generator are implemented, not as full flip-flops, but rather as latches to save space. Unlike a flip-flop that has two feedback loops, a latch is composed of a single loop which, when slightly modified, reduces the latch into a SRAM cell. The implementation of the input registers capitalizes on this observation to reuse the memory cells in the LUT for the input registers of the function generator as well.

Figure 4-3 shows a schematic diagram of the input registers in relation to the schematic of the LUT. The input registers connect directly to the multiplexer in the LUT without any intermediate buffering. Like the SRAM cells in the LUT, each of the input registers has two pass-gate connections

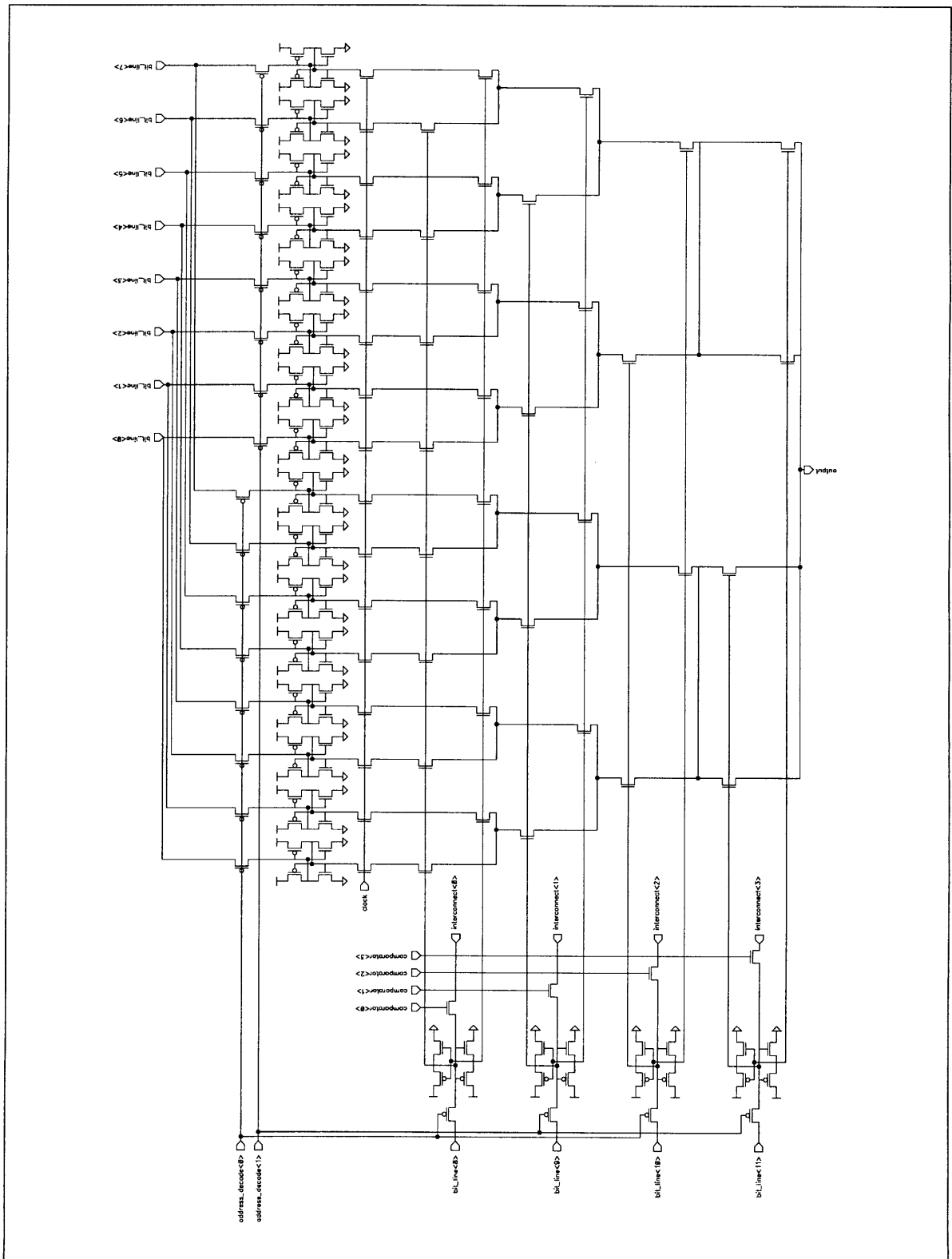


Figure 4-3: Input Registers Schematic

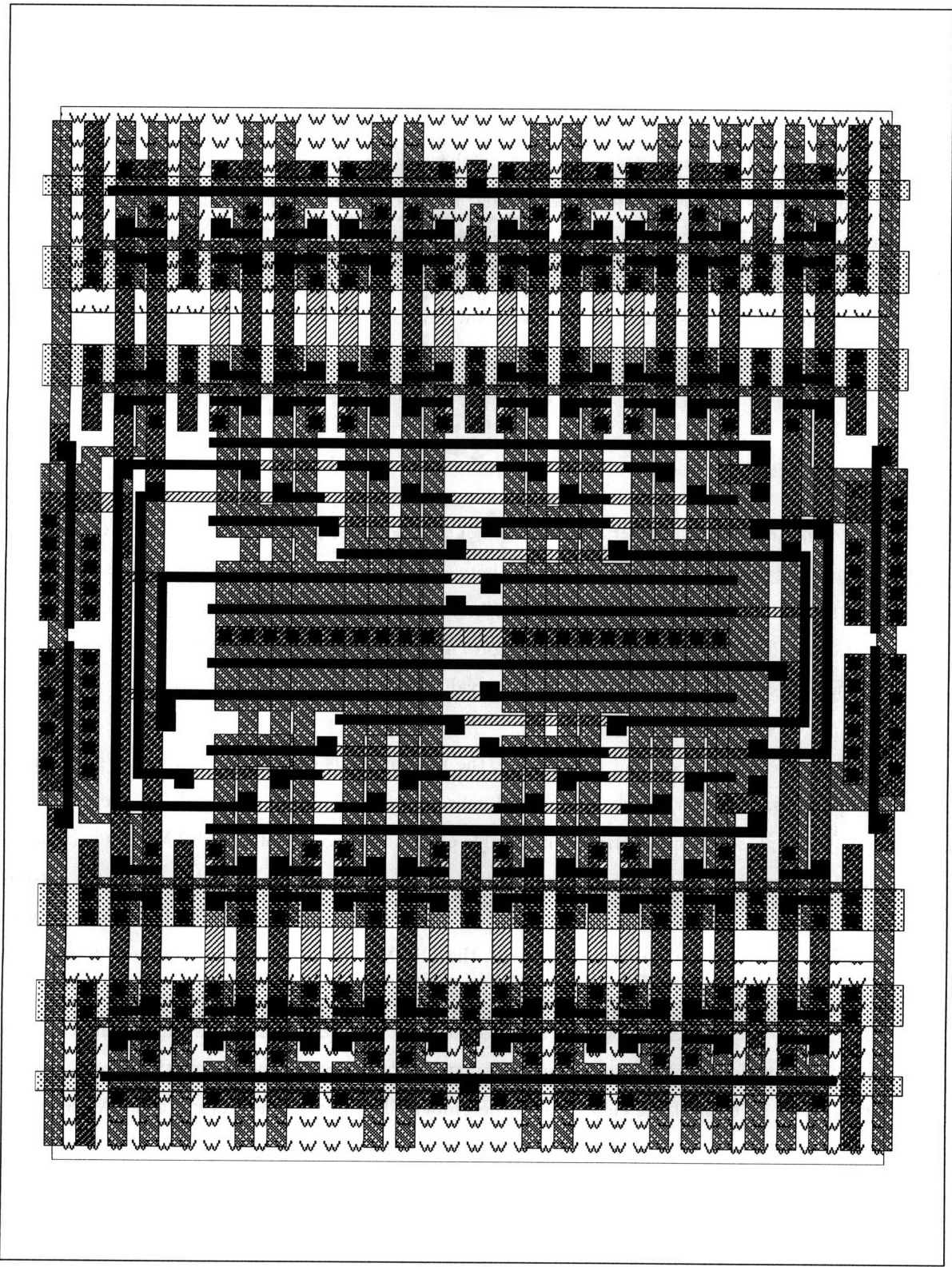


Figure 4-4: Input Registers Layout

for reading and writing. The PMOS pass-gate, controlled by the decoders of the configuration interface, links the input register to the bit-lines. By making every input register accessible through the configuration interface, the TSFPGA allows users to preset initial conditions for the operation of an application by writing to the registers and to check the status of an application by reading those registers. The NMOS pass-gate, controlled by the comparators described in Section 4.5, connects the input registers to the outputs of the crossbar in the complete network of the subarray and allows the registers to receive data sent from other function generators.

Because the input registers are simply SRAM cells, their layout, shown in Figure 4-4 alongside the layout of the LUT, is the same as that of the SRAM cells in the LUT. The four input registers sit at each of the four corners of the LUT.

4.5 Comparators

The comparators in the function generators of the TSFPGA take a two-step XOR-NOR approach to determining whether or not a match exists between the globally-distributed eight-bit time quantity and the locally-stored time quantity inside each of the comparators. A schematic diagram of a single comparator in the TSFPGA is shown in Figure 4-5. The two key features in the diagram are the eight memory cells which store the local time value and the comparison and reduction logic which detects the existence of a match.

Like the memory cells found in the LUT, the memory cells of the comparator are implemented as SRAM cells. However, unlike the SRAM cells in the LUT, the SRAM cells of the comparator have only one pass-gate connection. That single connection in each of the SRAM cells is controlled by the decoders of the configuration interface and links the individual SRAM cells to the bit-lines. No pass-gate connections are needed for the SRAM cells to communicate with the comparison and reduction portion of the comparator. Instead, the SRAM cells directly input their eight values into the comparison logic.

To conserve space, the two logical XOR-NOR stages of the comparator are implemented schematically in a single domino-style gate. Four pass-gates, two controlled by each of the SRAM cells and two controlled by each of the eight bits in the globally-broadcast time, compute the XOR value of each of the eight bits of time. The parallel arrangement of the eight XORs forms the NOR portion of the gate which combines the eight XOR values into a single output, buffered by an AND gate, for controlling the input register in a function generator.

For robustness, the schematic implementation assumes a two-phase clock. One phase of the clock precharges and discharges the domino gate to control the actual comparison operation. Once the result of the comparison has stabilized, the other phase of the clock enables and disables the AND buffer to determine when that result should be applied to the input registers of the function

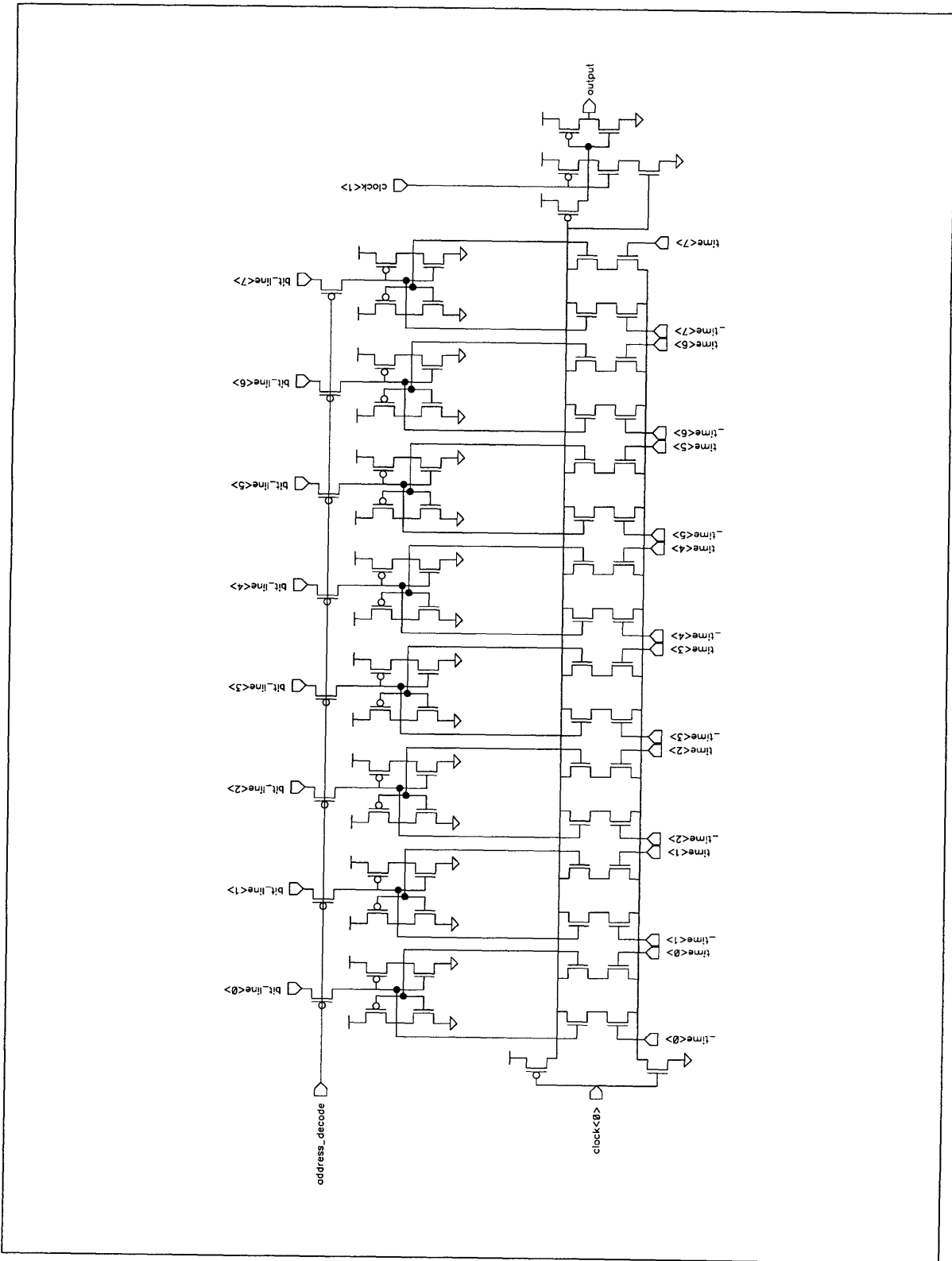


Figure 4-5: Comparators Schematic

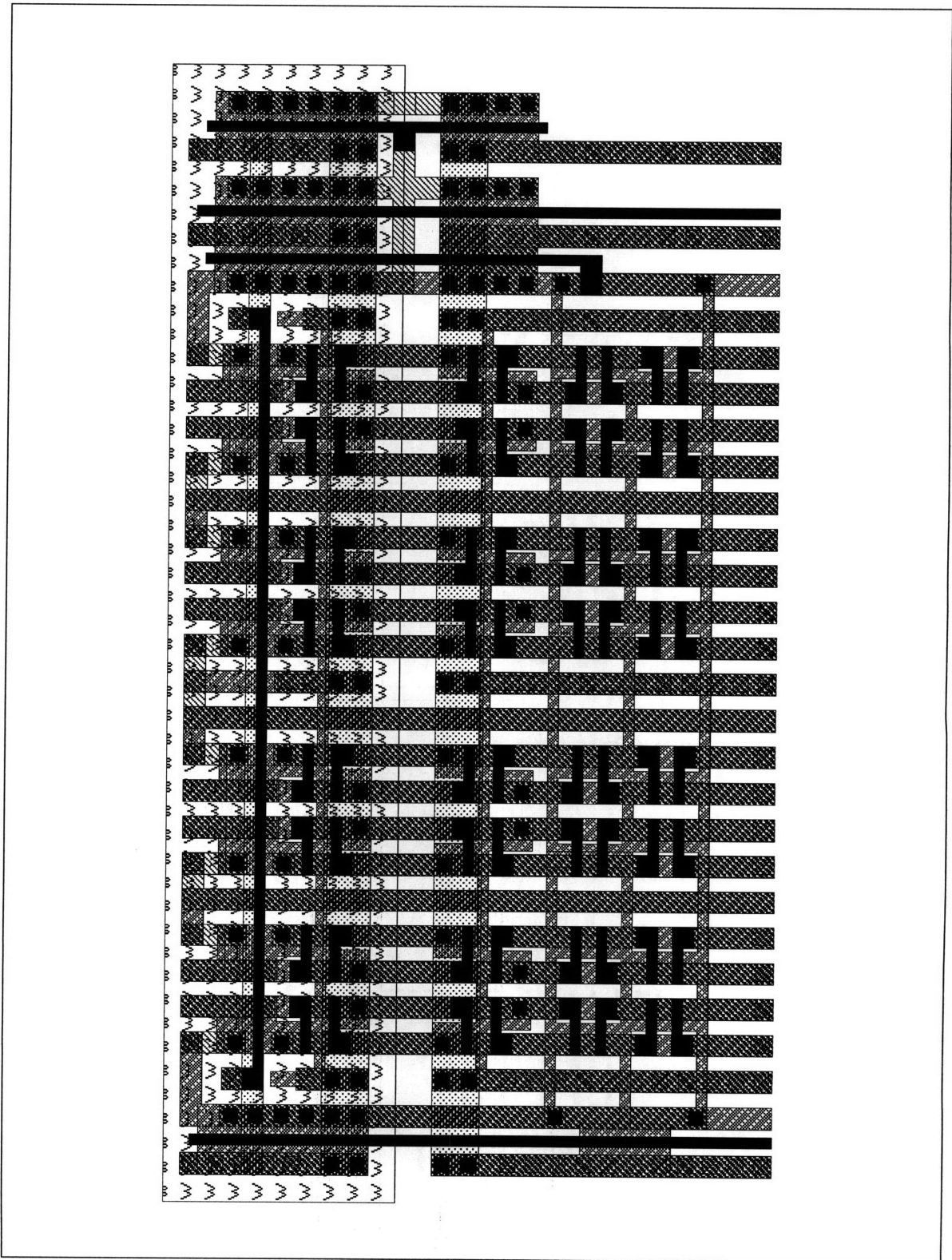


Figure 4-6: Comparators Layout

generator. By segregating the operation into comparison and application stages, the comparator provides the freedom of adjusting the clocks of the two stages independently of each other to resolve potential timing difficulties.

The layout of the comparator is shown in Figure 4-6. It is essentially a direct translation of the schematic.

4.6 Function Generator

The function generator combines the implementations of a LUT, four input registers, and four comparators. Figure 4-7 diagrams the connectivity between each of these subcomponents. One important detail is that the fifty-two total pass-gate connections linking the SRAM cells inside the LUT, input registers, and comparators to the bit-lines of the configuration interface are divided into two groups of twenty-six. A single twenty-six-bit bus connects the two groups together. To differentiate between the two groups, two address decode signals are provided by the configuration interface to the function generator.

The layout of the function generator is shown in Figure 4-8. Two comparators flank each side of the central LUT and input registers shown previously in the layout of Figure 4-4.

4.7 Array Element

The schematic implementation of each array element, shown in Figure 4-9, consists of an array of four function generators, as well as decoders and amplifiers from the configuration interface.

Collectively, a total of eight decoders are necessary to control the eight words of configuration information stored inside the four function generators. Each decoder is a four-input AND gate which combines a unique combination of the lowest three bits of the configuration address with a CHIP-SELECT signal for controlling the timing of the address decode. CHIP-SELECT is enabled only when the address inputs have stabilized so as to prevent multiple words, connected to the common twenty-six bit-lines, from interfering with each other.

Although the eight decoders only examine the lowest three bits of the configuration address, a total of thirteen bits are actually needed to fully distinguish between each of the words of configuration information controlling the programmable elements of the TSFPGA. The upper ten bits, along with a READ/WRITE signal, are decoded by two four-input AND gates and four three-input AND gates shown in the lower left-hand corner of Figure 4-9. The output of these decoders drive the amplifiers linking the bit-lines of the array element to the global programming lines of the configuration interface.

Each amplifier attached to each of the twenty-six bit-lines is composed of two inverters. One

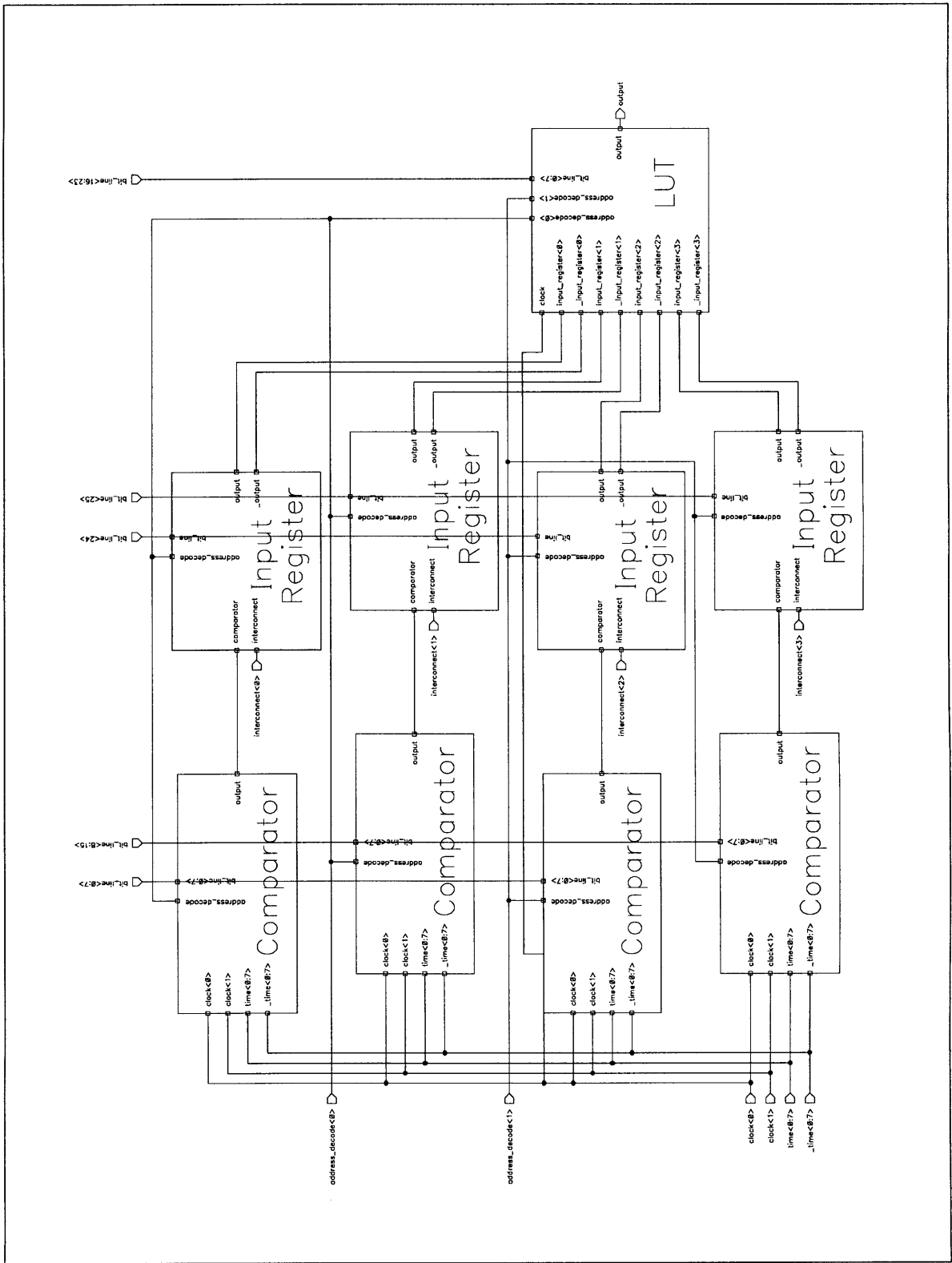


Figure 4-7: Function Generator Schematic

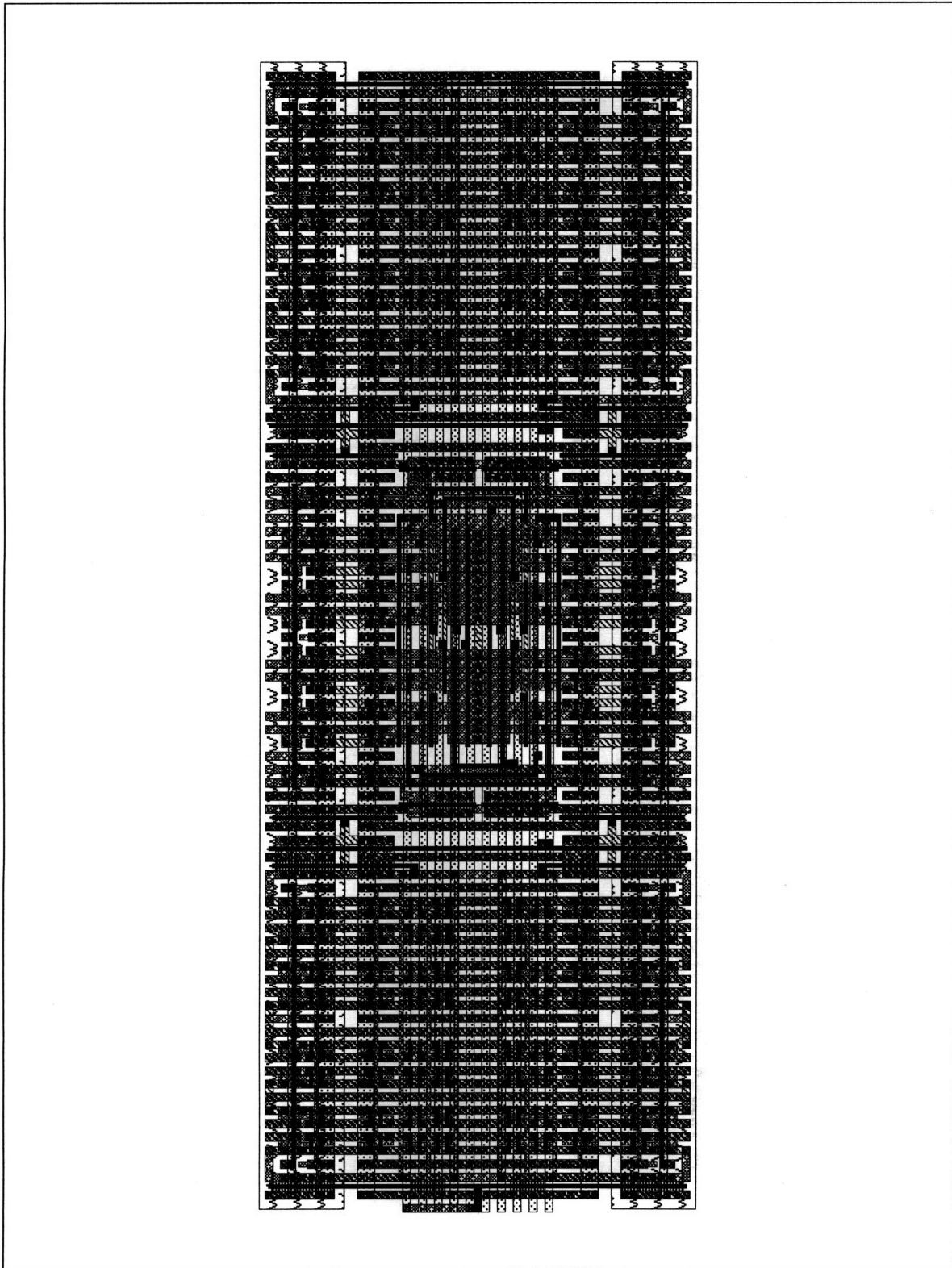


Figure 4-8: Function Generator Layout

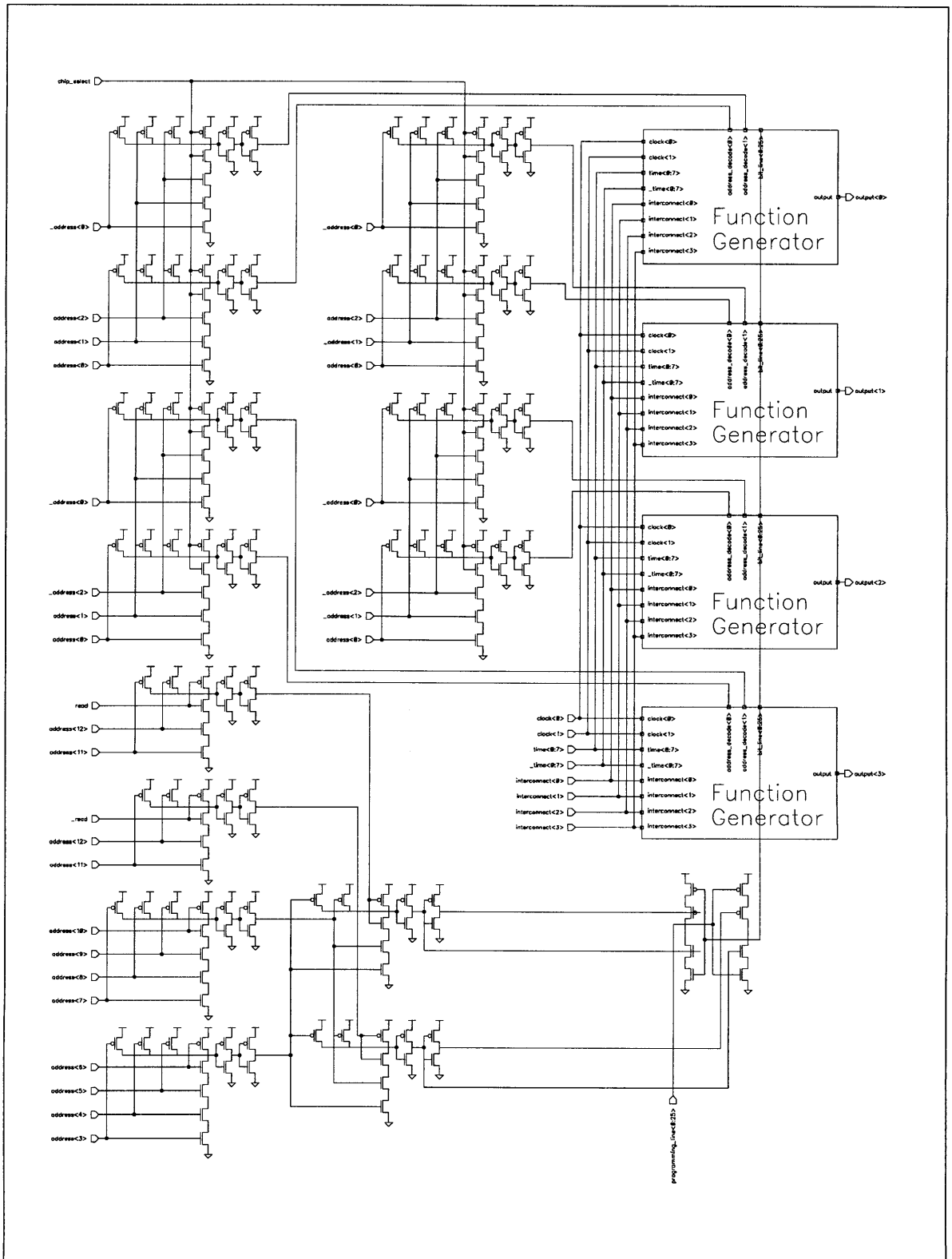


Figure 4-9: Array Element Schematic

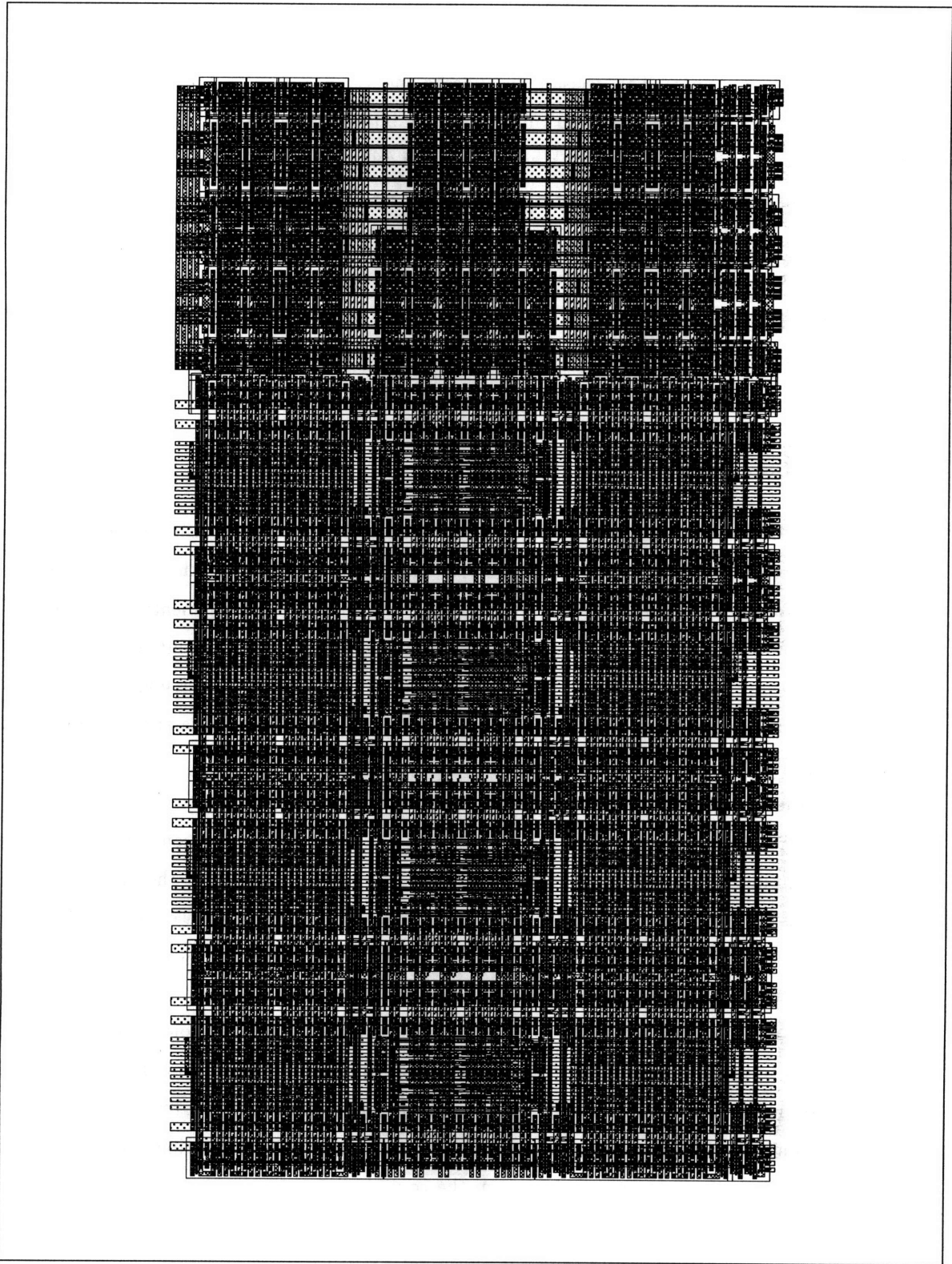


Figure 4-10: Array Element Layout

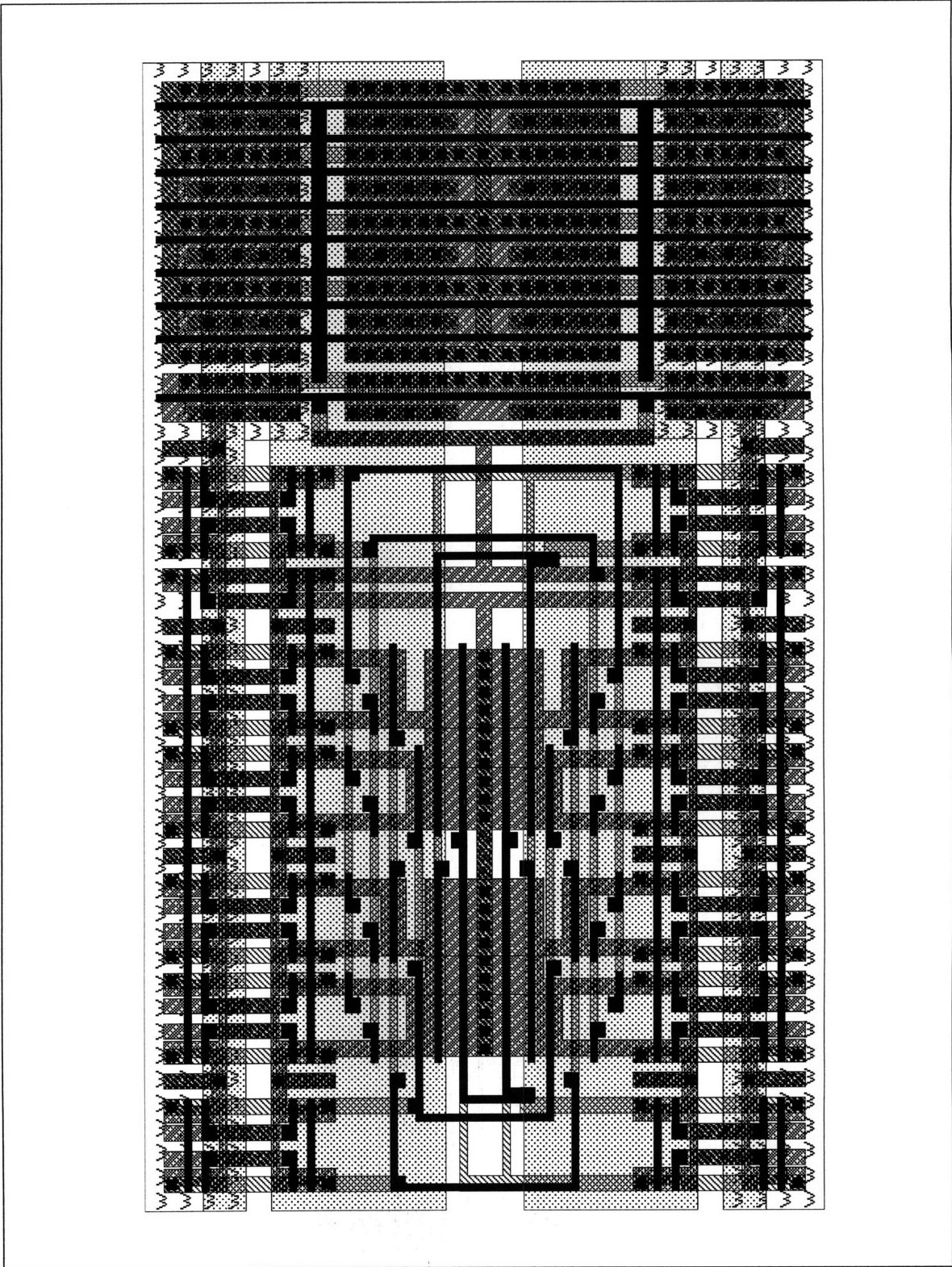


Figure 4-12: Crossbar Layout

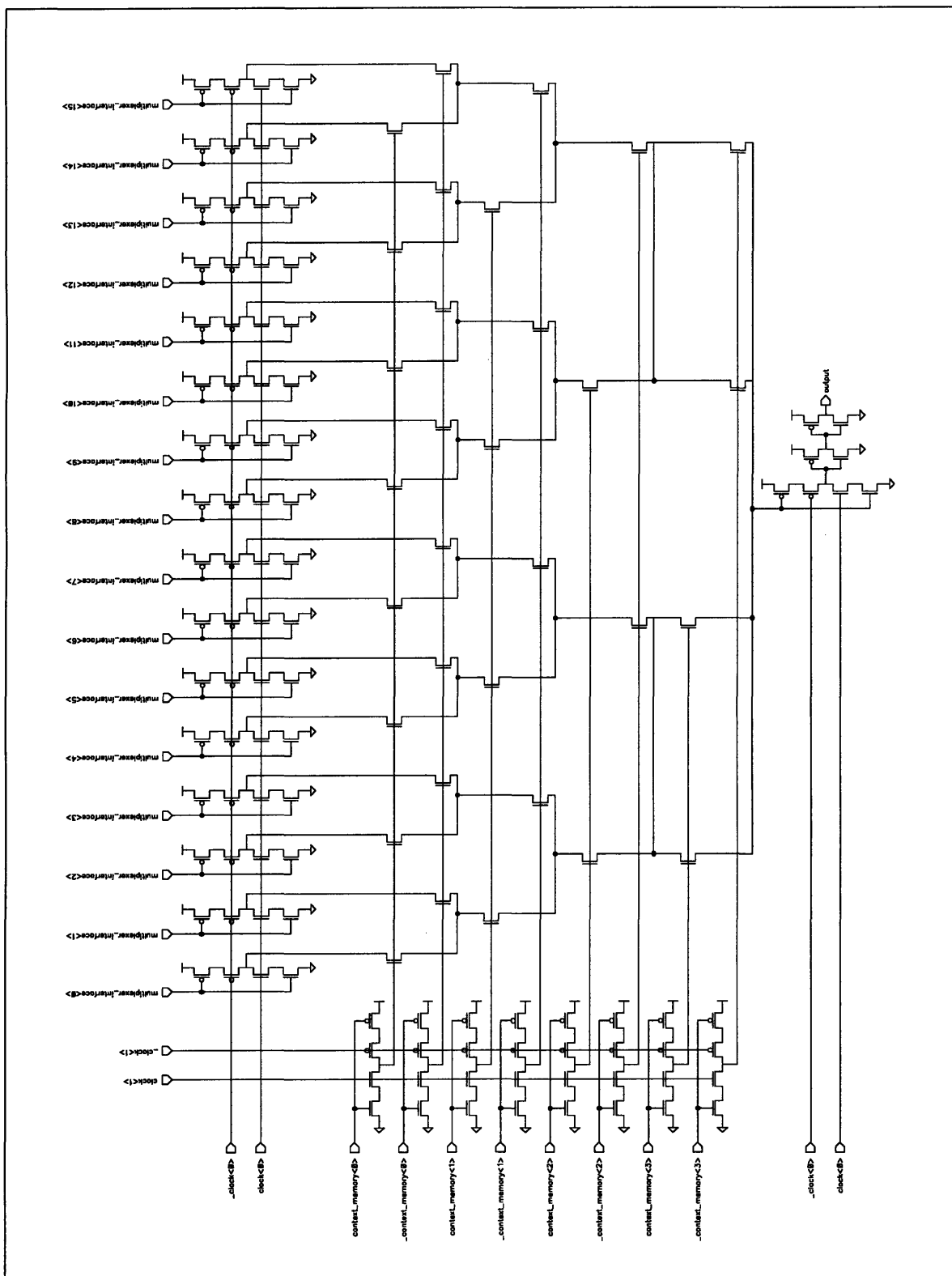


Figure 4-11: Crossbar Schematic

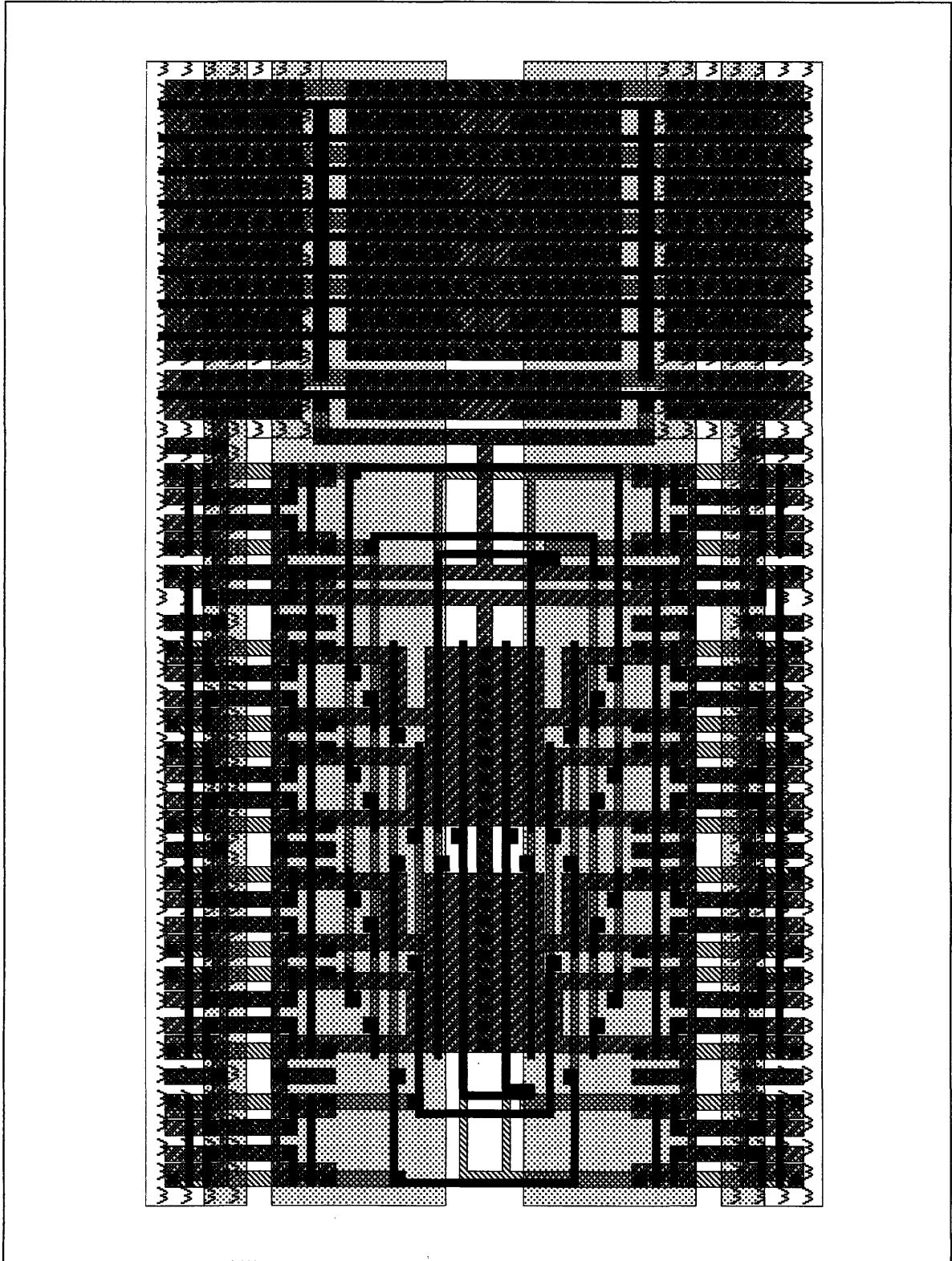


Figure 4-12: Crossbar Layout

phase of a two-phase clock. The output buffer serves two purposes: one, it isolates the internals of the multiplexer tree from further circuitry, and two, it separates the two pipeline stages of the TSFPGA. The first pipeline stage consists of the logic evaluation of the array element, multiplexer interface, and crossbar circuitry. The second stage is strictly dedicated for the interconnect delays encountered in the transmission of the crossbar outputs to the array element inputs in the same subarray via the complete network, to the multiplexer interface inputs of other complete networks via the manhattan network, and to the output connections to circuitry external to the TSFPGA via the operational interface.

The multiplexers of the crossbar, being implemented in a manner similar to that of the LUT, utilizes a layout derived from the layout of the LUT. Figure 4-12 shows the mirrored layout of a single multiplexer bounded on two sides with the sixteen half-registers of the input buffer and surrounded in the four corners with the eight half-registers of the select buffer. The output buffer attaches to the rest of the multiplexer at the top of the figure.

4.9 Multiplexer Interface

Each of the sixteen multiplexers in the multiplexer interface of the TSFPGA is implemented as shown in Figure 4-13. To save design effort, the implementation of the multiplexer interface borrows heavily from that of the crossbar and the schematic shown in the figure is essentially a scaled-down version of the schematic of Figure 4-11. The only notable difference is that the multiplexers of the multiplexer interface lack output buffers. These buffers are implemented instead as the input buffers of the crossbar.

The layout of the multiplexer interface is constructed in entirely the same fashion as that of the crossbar and is essentially a repeat of the crossbar layout.

4.10 Context Memory

The sixty-four-word context memory, indexed by the globally-broadcast, six-bit, context quantity, supplies the inputs to the select buffers of the crossbar and multiplexer interface. Because each of the sixteen multiplexers in the crossbar requires four select bits and each of the sixteen multiplexers in the multiplexer interface requires three select bits, a total of one hundred twelve bits are needed to satisfy both elements of the complete network. Rather than supplying all one hundred twelve bits of configuration information from a single, unwieldy, one hundred twelve-bit wide, sixty-four-word, monolithic block of memory, the TSFPGA divides the implementation of the context memory into thirty-two more manageable twenty-eight-bit wide, eight-word blocks which collectively provide the necessary configuration information.

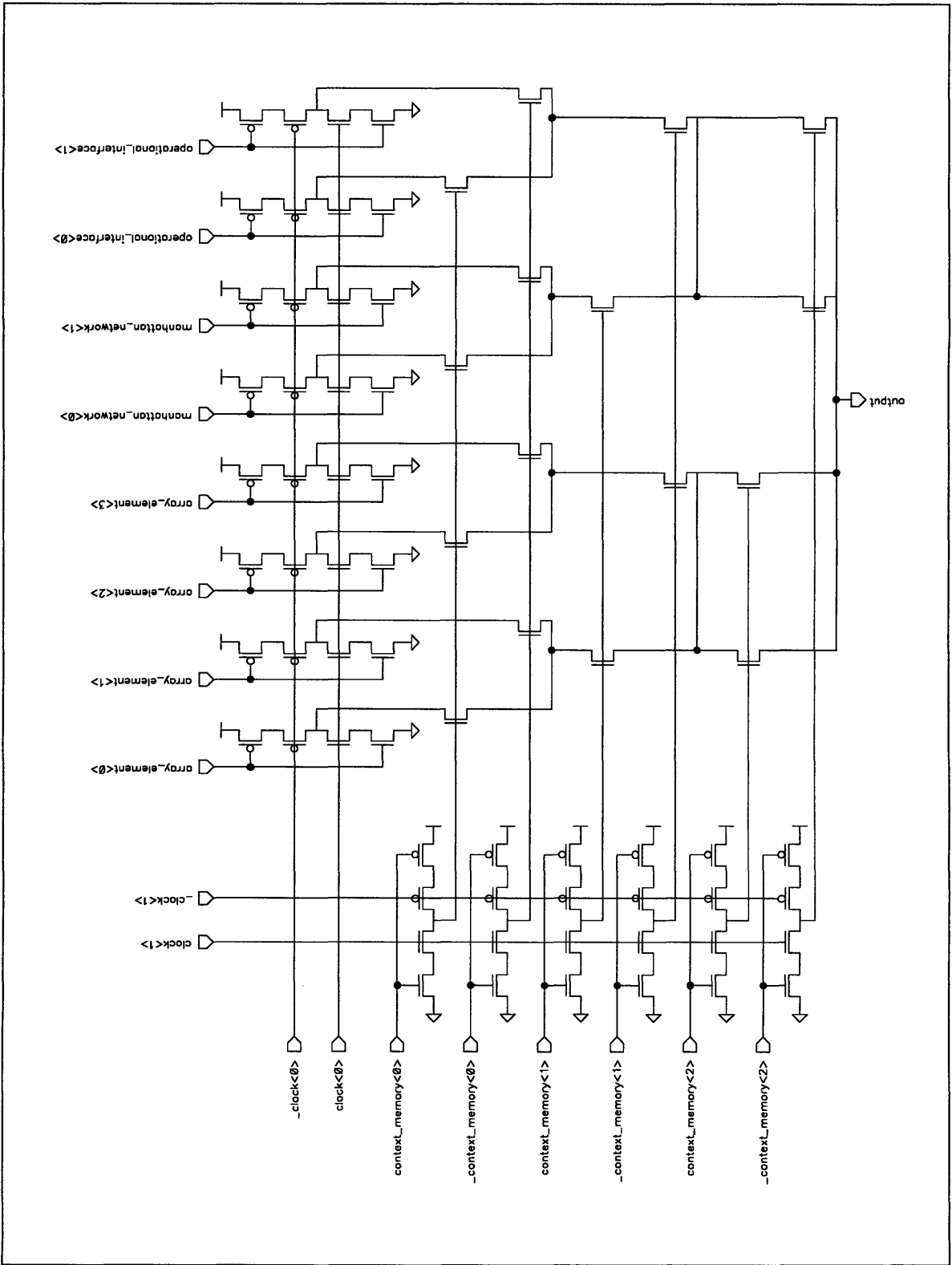


Figure 4-13: Multiplexer Interface Schematic

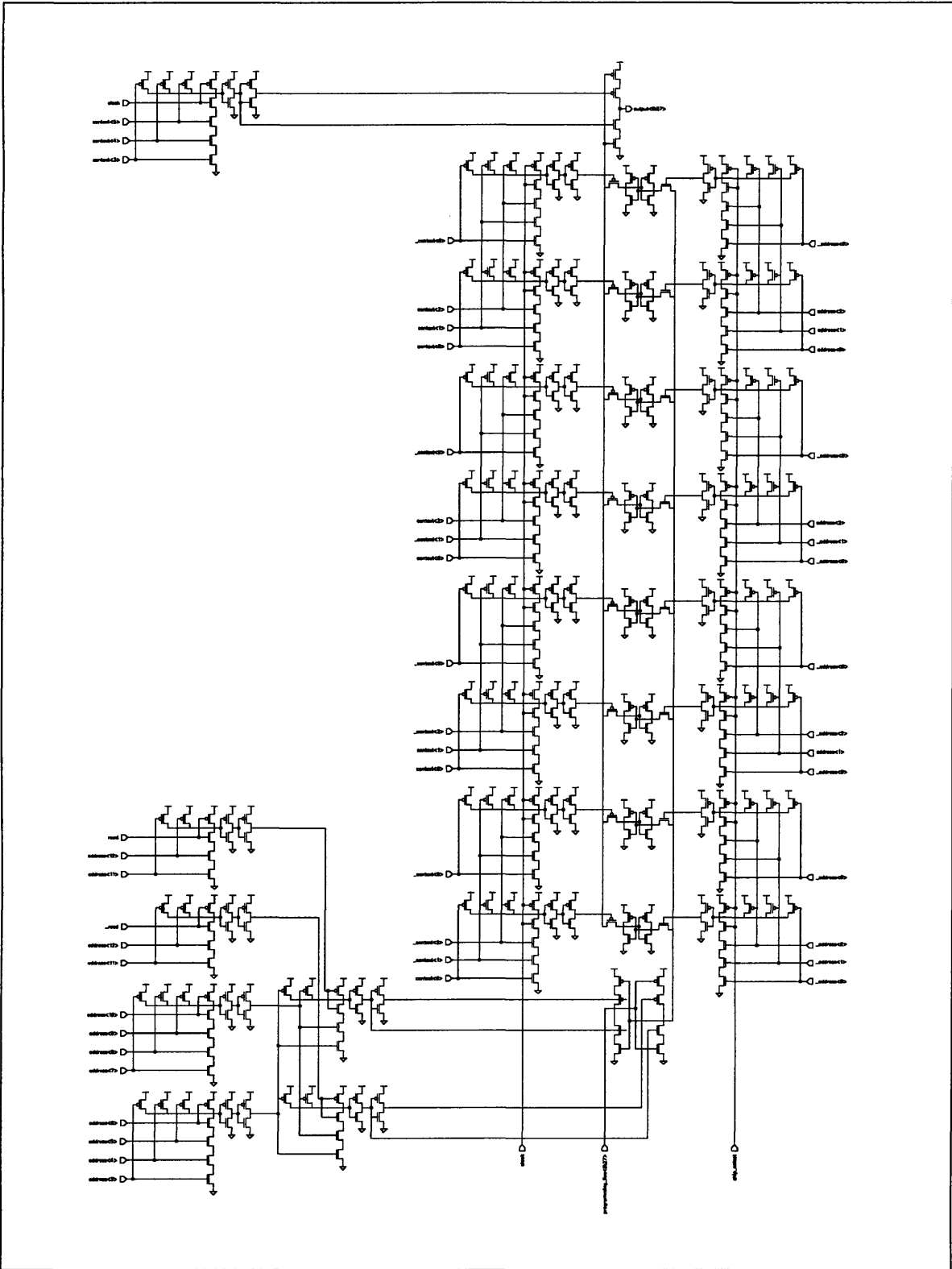


Figure 4-14: Context Memory Schematic

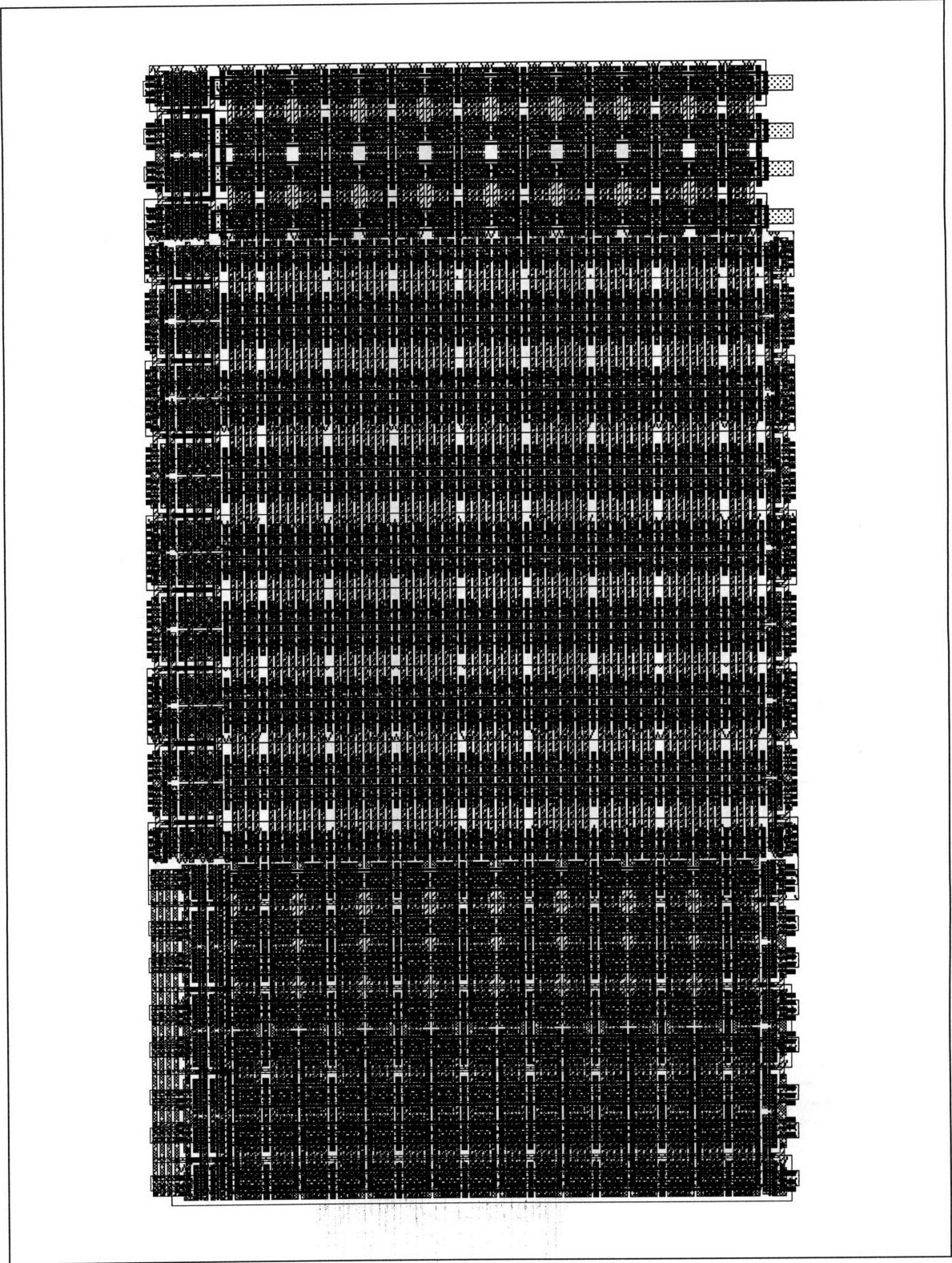


Figure 4-15: Context Memory Layout

The schematic of Figure 4-14 diagrams one of those thirty-two blocks of context memory, abstracting all of the identical columns of that memory into a single column. The circuitry shown in the schematic is organized into five functional groups. Central to the schematic and forming the first group is the column of eight memory cells. To the left of that column and forming the second group is the column of context decoders. Associated with those decoders are the output amplifiers at the top of the schematic constituting the third group. To the right of the column of memory cells is the column of address decoders belonging to the configuration interface and making up the fourth group. Associated with these decoders are the configuration amplifiers at the bottom of the schematic belonging to the fifth group.

The column of memory cells storing the configuration information for the crossbar and multiplexer interface is implemented in the form of standard SRAM cells with two pass-gate connections per cell, one connection to each set of decoders on either side of the memory column.

The PMOS connections of the SRAM cells are controlled by the context decoders determining which of the eight words of memory corresponds to the globally-broadcast context quantity. Each decoder is a four-input NAND gate merging a different combination of the lowest three bits of the globally-broadcast context with the configuration phase of a two-phase clock. The three bits differentiate between the eight words of memory while the clock input controls the timing of the memory read operation. By altering the state of the PMOS gates only on configuration clock phase edges after the three context inputs have stabilized, the context decoders ensure that the SRAM cells do not contend with each other when driving their common bit-lines.

The PMOS gates of the SRAM cells link the eight words of memory to a bus of twenty-eight bit-lines, each bit-line ending in an output amplifier implemented in the form of a half-register. All of the output amplifiers are controlled by a single four-input AND gate which decodes the top three bits of the globally-broadcast context quantity, combining them with the configuration phase of a two-phase clock for proper timing. During the configuration clock phase, if the AND gate enables the memory block outputs, then the half-registers amplify the chosen word of memory and drive the values from the twenty-eight bit-lines to the crossbar and multiplexer interface.

The NMOS connections of the SRAM cells are controlled by a group of eight address decoders which function analogously to the context decoders controlling the PMOS gates and work in exactly the same way as the address decoders used in the array element. The four-input decoders merge the eight different combinations of the lowest three bits of the configuration address with a CHIP-SELECT signal to enable and disable the each of the eight words of configuration memory during programming.

Like the address decoders, the configuration amplifiers of the context memory also the same as those found in the array element. The NMOS gates connect the SRAM cells to an array of twenty-eight configuration amplifiers interfacing between the bit-lines of the context memory and

the programming lines of the configuration interface. These amplifiers are controlled by a set of two four-input and four three-input AND gates decoding the top ten bits of the configuration address, along with the READ/WRITE signal.

The separation of the context decoders from the configuration address decoders and the incorporation of two sets of amplifiers is essential for allowing the TSFPGA to provide users with background programming capabilities. While the TSFPGA uses the context decoders to operate the crossbar and multiplexer interface, the user can simultaneously use the configuration decoders to access the programming stored in the context memory without conflict.

The layout of the context memory is shown in Figure 4-15. The SRAM cells are located in a rectangular matrix at the center of the layout and are bounded on both sides by decoders, on the left by the context decoders and on the right by the address decoders, just as in the schematic. At the top of the layout are the output amplifiers, implemented in two rows so that each half-register can span the width of two columns of SRAM cells for more effective use of transistor area in comparison to a single row of tall, one-column-wide, amplifiers. The configuration amplifiers are similarly implemented in two rows at bottom of the layout. To the right of both sets of amplifiers are the decoders enabling their operation.

4.11 Subarray

The subarray combines the logic and interconnect implementations described in the previous sections of this chapter to form the fundamental tiling block needed for the construction of the complete TSFPGA. The schematic of the subarray, composed using the array element, multiplexer interface, crossbar, and context memory components, is shown in Figure 4-16.

Figure 4-17 diagrams the layout of the subarray. Leveraging the benefits of symmetry to avoid potential complications in the tiling of subarrays, the layout is organized so that the left and right halves of the subarray, as well as the top and bottom halves, are mirror images of one another. The sixteen array elements forming the logic portion of the subarray are arranged in two rows of eight array elements each, one row at the top of the subarray and the other at the bottom. Separating the two rows of array elements are the components of the interconnect implementation, most prominently the context memory occupying the four center rows with eight blocks in each row for a total of thirty-two blocks. The sixteen multiplexers of the multiplexer interface form the boundary between the two rows of array elements and the two outer rows of the context memory, while the sixteen multiplexers of the crossbar, located at the center of the subarray, separate the two inner rows of the context memory. Besides being advantageous for tiling, this particular symmetric arrangement of logic and interconnect components produces an uniform, predictable flow of data from the boundaries of the subarray inward to its center during each time-step of computation.

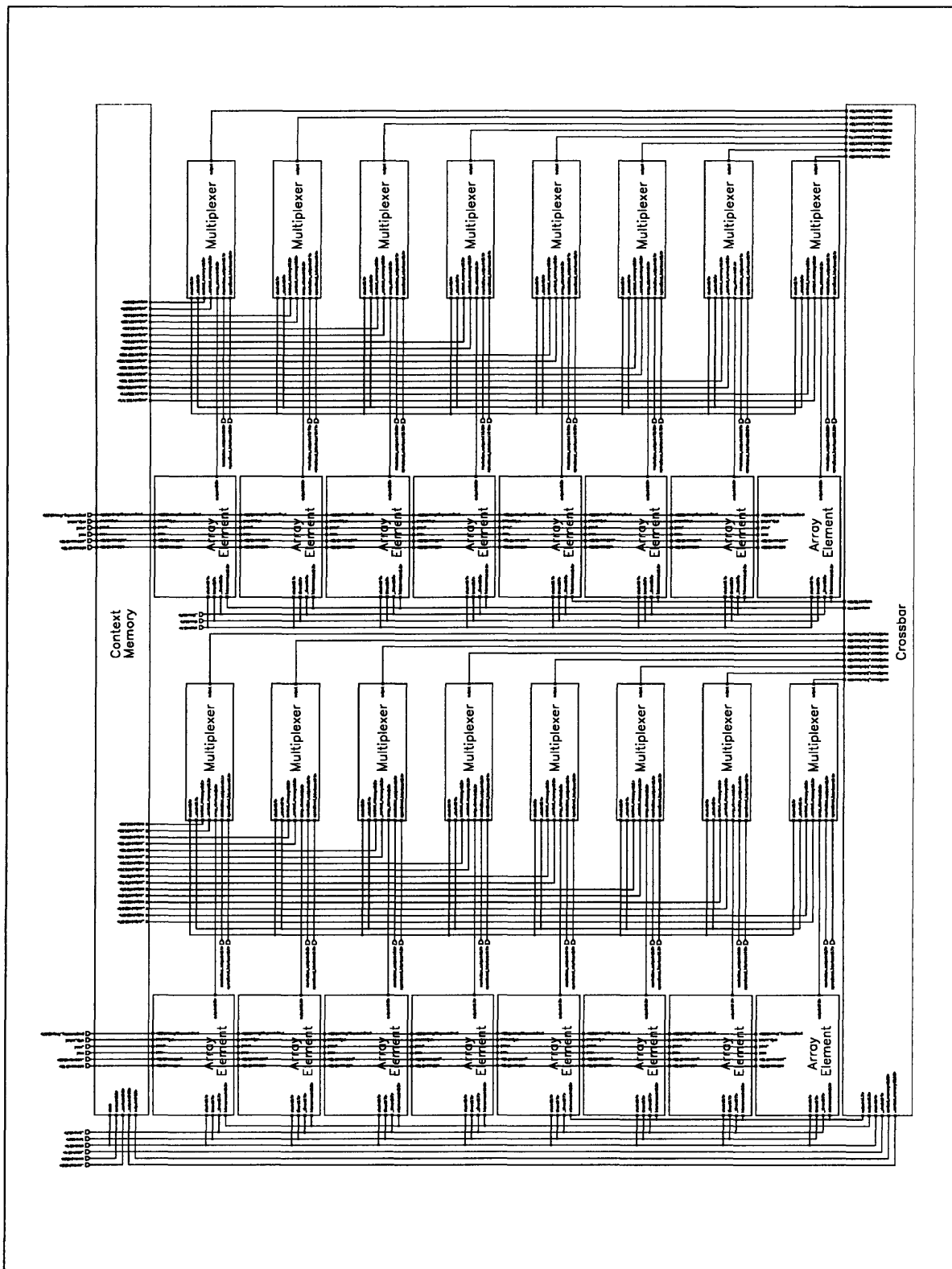


Figure 4-16: Subarray Schematic

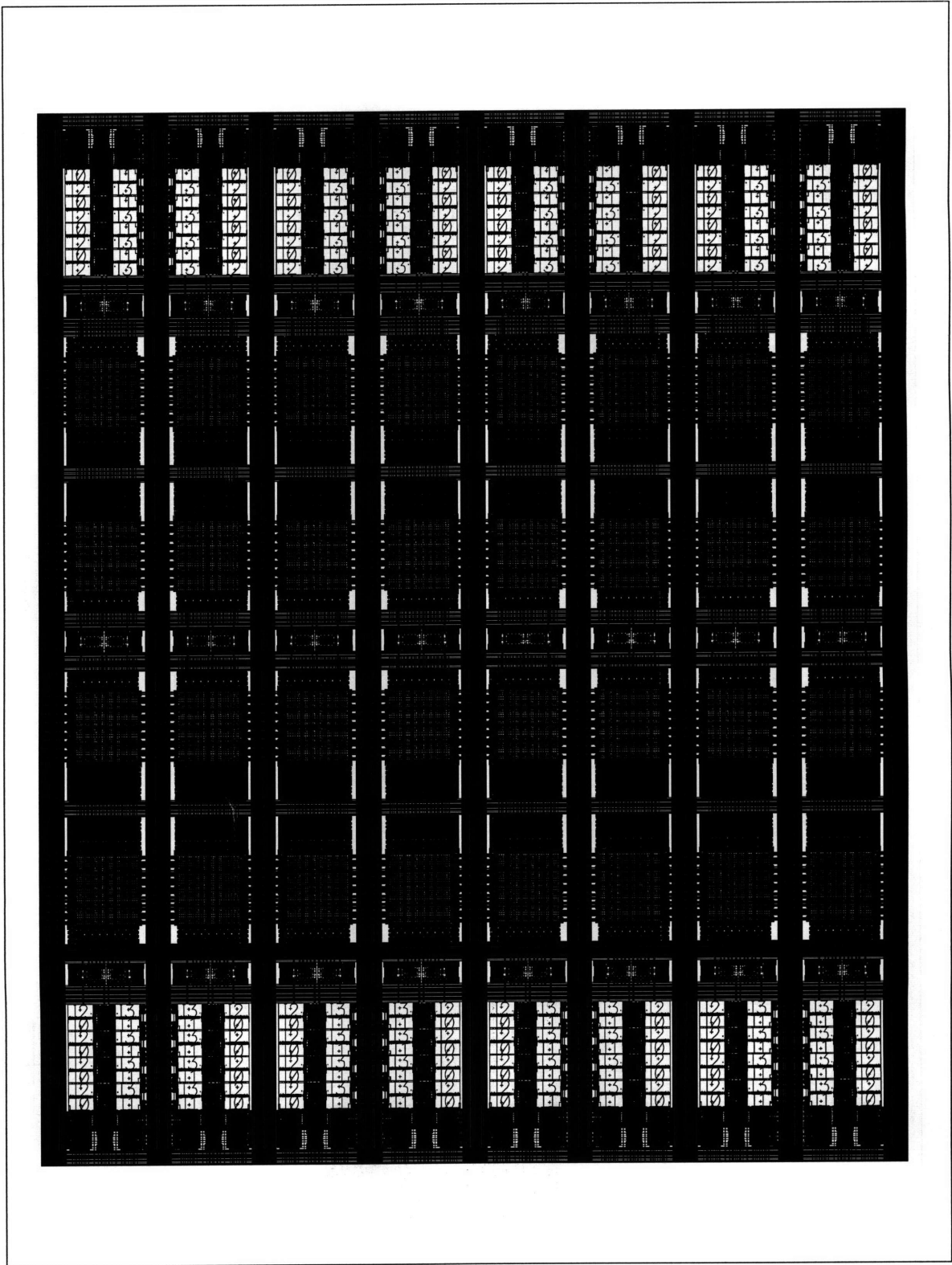


Figure 4-17: Subarray Layout

The sole additions to the layout of the subarray not shown previously in the layout figures of the individual logic and interconnect components are the series of routing channels running in both vertical and horizontal directions separating each of the components. The vertical channels are primarily used for the global distribution of the time and context quantities and for the distribution of the configuration address to the context memory. The horizontal channels carry the programming lines to the array elements and context memory as well as connect each of the thirty-two blocks of memory to the crossbar and multiplexer interface.

Chapter 5

Evaluation

Because the TSFPGA architecture promises to offer greater utility than conventional FPGA architectures while using fewer physical resources, an implementation of the TSFPGA should ideally provide improved flexibility beyond the capability of existing FPGA products while being more compact. In consideration of these two goals, this chapter presents an evaluation of the particular TSFPGA implementation presented in this thesis.

Because the degree of improved flexibility in the TSFPGA cannot be fully determined until the TSFPGA has been completely assembled and tested, this chapter focuses on evaluating the area utilization of the implementation. Section 5.1 gives an overview of the logic area utilization of the TSFPGA while Section 5.2 presents a summary of the interconnect area utilization.

5.1 Logic Area Utilization

An evaluation of the area requirements of each of the logic components incorporated into the subarray of the TSFPGA is given in Table 5.1. The table lists the components and their constituents in the order that they were presented in Chapter 4. Although the vertical and horizontal routing channels of the subarray are technically part of the interconnect, not the logic, they are included as part of this analysis because their existence prevents the silicon area underneath them from being utilized, and as a result, they affect the logic area statistics of the subarray.

According to the table, in this implementation of the TSFPGA, the context memory, routing channels, and array elements consume the vast majority of the logic area, totalling over ninety-five percent of the subarray area, while the crossbar and multiplexer interface account for an almost negligible amount. Ignoring possible optimizations to the implementation of the routing channels, which will be discussed in Section 5.2, the information given in the table implies that the greatest benefit in reducing the logic area utilization of future generations of TSFPGAs will come from optimizing the implementations of the context memory and array elements.

Component	Quantity	Unit Dimensions	Total Area	Percent Area
Look-Up Table	1	34.0um x 60.2um	2046.8um²	100.0%
- SRAM Cells	16	4.0um x 18.7um	1196.8um ²	58.5%
- Multiplexer	1	34.0um x 23.6um	802.4um ²	39.2%
Input Registers	1	8.6um x 29.6um	254.6um²	100.0%
Comparators	1	60.4um x 33.5um	2023.4um²	100.0%
- SRAM Cells	8	4.0um x 17.7um	566.4um ²	28.0%
- XOR-NOR Reduction	1	60.4um x 13.4um	809.4um ²	40.0%
Function Generator	1	172.8um x 65.0um	11232.0um²	100.0%
- Look-Up Table	1	34.0um x 60.2um	2046.8um ²	18.2%
- Input Registers	4	8.6um x 29.6um	1018.2um ²	9.1%
- Comparators	4	60.4um x 33.5um	8093.6um ²	72.1%
Array Element	1	192.4um x 346.4um	66647.4um²	100.0%
- Function Generators	4	172.8um x 65.0um	44928.0um ²	67.4%
- Decoders	1	15.0um x 254.0um	3810.0um ²	5.7%
- Amplifiers	1	192.4um x 92.4um	17777.8um ²	26.7%
Crossbar	1	1409.6um x 51.4um	72453.4um²	100.0%
Multiplexer Interface	1	1409.6um x 51.4um	72453.4um²	100.0%
Context Memory	1	177.8um x 307.8um	54726.8um²	100.0%
- SRAM Cells	256	4.5um x 20.9um	24076.8um ²	44.0%
- Context Decoders	1	21.1um x 169.2um	3570.1um ²	6.5%
- Context Amplifiers	1	177.8um x 46.2um	8214.4um ²	15.0%
- Address Decoders	1	8.7um x 169.2um	1472.0um ²	2.7%
- Address Amplifiers	1	177.8um x 92.4um	16428.7um ²	30.0%
Subarray	1	2018.4um x 2479.4um	5004421.0um²	100.0%
- Array Elements	16	192.4um x 346.4um	1066357.8um ²	21.3%
- Crossbar	1	1409.6um x 51.4um	72453.4um ²	1.4%
- Multiplexer Interface	1	1409.6um x 51.4um	72453.4um ²	1.4%
- Context Memory	32	177.8um x 307.8um	1751258.9um ²	35.0%
- Vertical Channels	8	60.0um x 2479.4um	1190112.0um ²	23.8%
- Horizontal Channels	8	2018.4um x 47.6um	768606.7um ²	15.4%

Table 5.1: TSFPGA Logic Area Utilization

5.1.1 Context Memory

Because the context memory of the TSFPGA is implemented in thirty-two blocks of eight, twenty-eight-bit words instead of a single monolithic block containing all configuration information, each block of memory in the TSFPGA devotes approximately half of its logic area to SRAM cells and half to context and configuration address decoders and amplifiers.

The amount of area required by the memory cells would have been greater had the TSFPGA architecture not made use of a dense encoding of configuration information for the crossbar and multiplexer interface to minimize their number of configuration bits. The four bits needed for configuring each multiplexer of the crossbar and the three bits needed for each multiplexer of the multiplexer interface are the result of storing a logarithmic encoding of the multiplexer function in the context memory.

However, despite taking advantage of the area benefits of dense encoding, the implementation of the SRAM cells in the context memory should have been completed in even less area. During the construction of the memory, the layout of the SRAM cells was slightly enlarged to accommodate the vertical layout height of the decoders because the additional size increase was judged to be minimal. In retrospect, this decision was a mistake. The minimization of the SRAM cell area should have taken priority over layout conveniences for the decoders. Reimplementing the context memory to follow this precedence will reduce the area of the memory layout by approximately five percent.

Further reductions in the area of the context memory can come from implementing two additional changes. The first is reorganizing the memory into fewer than thirty-two blocks, with each block having more than eight words. Fewer blocks not only imply the need for fewer decoders and amplifiers, but also allow the area overhead of the remaining decoders and amplifiers to be amortized over more words of memory. The major tradeoff in taking this approach is a performance decrease in the operation of the memory due to the larger size of each block. The second possible change to allow further area reductions is eliminating the capability of background programming and using only one set of decoders and amplifiers for both the programming and operation of the TSFPGA. This modification can ultimately save up to a third of the area needed to implement the context memory, but with the negative result of eliminating the possibility of simultaneous programming and operation.

5.1.2 Array Element

The majority of the area used in the implementation of the array element can be attributed to two groups of circuitry. Approximately two-thirds of the area in the array element is occupied by the four function generators, while the remaining third is consumed by the configuration address decoders and amplifiers. Within the function generators, almost three-quarters of the logic area is used by the

comparators, with just over a quarter of the area allocated for the LUT and input registers. This area distribution suggests that optimizations to the area usage of the array element will benefit most from size reductions in the address decoders and amplifiers and from reductions in the comparators of the function generators.

Reductions in the size of the address decoders and amplifiers in the array element can be achieved by applying the same reorganization techniques previously suggested for the context memory. In the current implementation, one set of decoders and amplifiers is implemented per array element for every four function generators in order to parallel the architectural design of the TSFPGA in the interest of simplicity. However, the implementation of the TSFPGA does not necessarily have to follow the partitions of the logic hierarchy set by the architecture, and in fact, choosing to group eight function generators, or more, for each set of decoders and amplifiers in future implementations of the TSFPGA will result in area reductions, just as grouping more than eight words of configuration in each block of context memory improves space utilization.

Area reductions in the implementation of the comparators can come from reorganizing the layout of the function generator. In the current layout, the comparators, placed at the four corners of the function generator, are restricted to matching the vertical height set by the LUT. The integration of the comparators with the LUT using this organization results in inefficient layout because the comparators, being vertically smaller than the LUT, leave slack space in the function generator. By attempting other layout strategies which may eliminate the unused logic area, the size of the function generator, and consequently, the array element, can potentially be reduced by as much as ten percent.

5.2 Interconnect Area Utilization

Table 5.2 and Table 5.3 provide an overview of the interconnect requirements of the primary signals implemented in the subarray of the TSFPGA. These two tables list the components of the TSFPGA in the order that they appear in the logic area utilization table, and for each component, the tables provide a decomposition of the interconnect area utilization of the primary signals in that component by metal layer. In every component with the exception of the subarray, the lowest metal layer is always completely used by the local interconnect for connections between the PMOS and NMOS devices of the implementation and for connections between adjacent components. If a different design style deemphasizing the use of complementary devices had been selected to implement the TSFPGA, a portion of the first metal layer could have been made available for use by the signals currently relegated to the higher metal layers.

The tables show that, in terms of functionality, the dominant consumers by far of interconnect area for this implementation of the TSFPGA are the bit-lines, programming lines, and address distri-

Component	Layer	Function	Percent Area
Look-Up Table	Metal 1	Local Interconnect	100%
	Metal 2	Bit-Lines	50%
		Unused	50%
Metal 3	Power/Ground	20%	
	Time Distribution	20%	
	Clock Distribution	5%	
	Unused	55%	
Input Registers	Metal 1	Local Interconnect	100%
	Metal 2	Data	60%
		Bit-Lines	40%
Metal 3	Power/Ground	20%	
	Time Distribution	20%	
	Clock Distribution	5%	
	Unused	55%	
Comparators	Metal 1	Local Interconnect	100%
	Metal 2	Time Distribution	60%
		Bit-Lines	40%
Metal 3	Power/Ground	20%	
	Time Distribution	20%	
	Clock Distribution	5%	
	Unused	55%	
Function Generator	Metal 1	Local Interconnect	100%
	Metal 2	Bit-Lines	45%
		Time Distribution	40%
Data		5%	
Unused		10%	
Metal 3	Power/Ground	20%	
	Time Distribution	20%	
	Clock Distribution	5%	
	Unused	55%	
Array Element	Metal 1	Local Interconnect	100%
	Metal 2	Bit-Lines	45%
		Time Distribution	30%
Programming Lines		10%	
Data		5%	
Metal 3	Unused	10%	
	Power/Ground	20%	
	Time Distribution	15%	
	Clock Distribution	5%	
Unused	60%		

Table 5.2: TSFPGA Interconnect Area Utilization (Part 1)

Component	Layer	Function	Percent Area
Crossbar	Metal 1	Local Interconnect	100%
	Metal 2	Data	40%
		Select	10%
		Unused	50%
	Metal 3	Power/Ground	40%
		Clock Distribution	10%
Unused		50%	
Multiplexer Interface	Metal 1	Local Interconnect	100%
	Metal 2	Data	40%
		Select	10%
		Unused	50%
	Metal 3	Power/Ground	40%
		Clock Distribution	10%
Unused		50%	
Context Memory	Metal 1	Local Interconnect	100%
	Metal 2	Bit-Lines	40%
		Programming Lines	40%
		Unused	20%
	Metal 3	Power/Ground	40%
		Clock Distribution	5%
Unused		55%	
Subarray	Metal 1	Local Interconnect	65%
		Data	12%
		Address Distribution	10%
		Context Distribution	7%
		Programming Lines	6%
	Metal 2	Bit-Lines	25%
		Programming Lines	22%
		Address Distribution	12%
		Data	10%
		Time Distribution	8%
		Context Distribution	6%
		Clock Distribution	4%
	Select	3%	
	Unused	10%	
	Metal 3	Power/Ground	35%
Data		12%	
Time Distribution		9%	
Clock Distribution		8%	
Programming Lines		6%	
Unused		30%	

Table 5.3: TSFPGA Interconnect Area Utilization (Part 2)

bution of the configuration interface, together accounting for almost sixty percent of the interconnect on the second metal layer as well as portions of the interconnect on the first and third layers. Optimizing the implementation of these three facets of the configuration interface should provide future generations of TSFPGAs with the greatest reduction in interconnect area. Additionally, optimizing the routing channels, as pointed out in the summary of Section 5.1, should also provide significant interconnect benefits.

5.2.1 Configuration Interface

The TSFPGA architecture specifies a configuration interface based upon the concept of a random-access memory because of its simplicity and flexibility. However, as the current implementation of the architecture shows, the penalty for that flexibility is the investment of significant amounts of interconnect resources in constructing the configuration interface.

Nevertheless, the resource requirements of the configuration interface can be moderated with an informed implementation. One potential optimization over the current implementation makes use of the observation that the decoders of the TSFPGA require a relatively small amount of logic area. This observation suggests that instead of using twenty-eight-bit-wide words, as is done by the current configuration interface, future implementations of the TSFPGA should divide the configuration information into smaller-width words, trading the incremental logic area cost of additional decoders for greater interconnect area savings resulting from minimizing the bit-line and programming line resources needed by the interface. The main disadvantage of using smaller-width words is the longer initial programming time needed to configure the TSFPGA. Device configuration time can be an important factor determining the performance of the TSFPGA, and FPGA-style devices in general, in rapid-reuse situations where the user makes heavy use of configuration capabilities.

Future implementations of the TSFPGA can reduce the interconnect requirements of the configuration interface even further by forgoing the random-access capability of the interface. By adopting a serial, shift-register scheme for loading configuration information, like that used by Xilinx FPGAs, the interconnect requirements of the configuration interface can potentially be satisfied using only the local interconnect of metal 1.

5.2.2 Routing Channels

As mentioned in the discussion of the subarray in Section 4.11, the TSFPGA uses the vertical and horizontal routing channels to distribute a number of signals, including the time and context quantities, the configuration address, and the programming lines. For the sake of simplicity and in the interest of eliminating control logic at the top level of the TSFPGA, the subarray distributes all signals in an unencoded format. This approach eliminates the need for decoders, but requires the distribution of 2^n lines for each n -bit quantity.

Because the optimization of interconnect area is a concern for the TSFPGA, some interconnect area can be recovered through the distribution of encoded quantities instead, at the cost of introducing global logic complexity. Although the implementation of additional circuitry for decoding the encoded signals does not intrinsically pose a significant challenge, the simulation and verification of that circuitry in combination with all other circuit elements at the global scale can require significant amounts of computational resources and may not be accurate, or, in general, feasible.

Assuming that encoding signals is not always an available implementation option, another possible technique for reducing the area utilization of the routing channels is to attempt to implement as many of those wires as possible in the available interconnect space above the array element, crossbar, multiplexer interface, and context memory components of the subarray. Because only ten percent of the metal 2 layer is available for this optimization in the current implementation of the subarray, making this optimization requires moving the majority of the signals to metal 3. Although some signals can make the transition from the routing channels to the metal 3 layer above the logic circuitry, most signals, unfortunately, have to remain in the routing channels because of their need to make turns. With only limited metal 2 interconnect, turning corners in metal 3 alone is not generally possible when routing over existing circuitry. However, if a four-layer metal fabrication process were available, all signals could pass over the existing components of the subarray, without requiring any redesign of the underlying circuitry, and the routing channels could be completely eliminated.

Chapter 6

Conclusion

Field-Programmable Gate Arrays (FPGAs) are computational devices containing uncommitted logic and interconnect resources which users configure for the applications they wish to run. Because the potential applications for FPGAs are not known at the time that they are fabricated, manufacturers of FPGAs should ideally provide abundant amounts of both logic capacity and interconnect flexibility to suit the widest possible range of applications. Current FPGAs, however, are lacking in interconnect flexibility because they neglect the temporal dimension of flexibility and exploit only the spatial aspect.

Unlike conventional FPGAs, the Time-Switched Field-Programmable Gate Array (TSFPGA) capitalizes on both the spatial and temporal aspects of flexibility by utilizing a time-switched interconnect architecture. The TSFPGA applies its time-switched interconnect to multiplex the physical switch and wire resources so that any particular switch or wire can replicate the functionality of n switches or wires over $O(n)$ amount of time units. Under this scheme, time serves as an additional dimension for augmenting spatial interconnect resources because it allows the TSFPGA to emulate connectivity between logic blocks temporally instead of providing that connectivity physically. By reusing the limited number of physical switches and wires, the TSFPGA promises greater interconnect flexibility than that possible in existing FPGA architectures.

Bibliography

- [BCE⁺94] Jeremy Brown, Derrick Chen, Ian Eslick, Edward Tau, and André DeHon. A 1μ CMOS Dynamically Programmable Gate Array. Transit Note 112, MIT Artificial Intelligence Laboratory, November 1994. Anonymous FTP `transit.ai.mit.edu:transit-notes/tn112.ps.Z`.
- [BDK93] Michael Bolotski, André DeHon, and Thomas F. Knight Jr. *Unifying FPGAs and SIMD Arrays*. Transit Note 95, MIT Artificial Intelligence Laboratory, September 1993. Anonymous FTP `transit.ai.mit.edu:transit-notes/tn95.ps.Z`.
- [CD96] Derrick Chen and André DeHon. TSFPGA: A Fine-Grain Reconfigurable Architecture with Time-Switched Interconnect Transit Note 134, MIT Artificial Intelligence Laboratory, 1996. Anonymous FTP `transit.ai.mit.edu:transit-notes/tn134.ps.Z`.
- [Chen96] Derrick Chen. An Introduction to the TSFPGA. Transit Note 133, MIT Artificial Intelligence Laboratory, 1996. Anonymous FTP `transit.ai.mit.edu:transit-notes/tn133.ps.Z`.
- [Fra92] Robert Francis. *Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays*. PhD thesis, University of Toronto, 1992.
- [HP94] Hewlett-Packard Co., Integrated Circuit Business Div., Fort Collins, CO. *CMOS14TB Design Reference Manual, 1994*.
- [RFLC90] Jonathan Rose, Robert Francis, David Lewis, and Paul Chow. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency. *IEEE Journal of Solid-State Circuits*, 25(5):1217-1225, October 1990.
- [Tau96] Edward Tau. *Testing of a First-Generation Dynamically Programmable Gate Array*. Masters thesis, Massachusetts Institute of Technology, 1996.
- [WE92] Neil H.E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design* Addison-Wesley Publishing Company, 1992.

[Xil93] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *The Programmable Logic Data Book, 1993.*

7191-12