Copyright © 1988, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

DESIGN AND IMPLEMENTATION OF INTEGRATED CIRCUITS FOR A REAL-TIME FLEXIBLE CHANNEL EMULATOR APPLYING SILICON ASSEMBLY TOOLS

by

Jane S. Sun

Memorandum No. UCB/ERL M88/43

14 June 1988

COVER PAGE

DESIGN AND IMPLEMENTATION OF INTEGRATED CIRCUITS FOR A REAL-TIME FLEXIBLE CHANNEL EMULATOR APPLYING SILICON ASSEMBLY TOOLS

by

Jane S. Sun

Memorandum No. UCB/ERL M88/43

14 June 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

DESIGN AND IMPLEMENTATION OF INTEGRATED CIRCUITS FOR A REAL-TIME FLEXIBLE CHANNEL EMULATOR APPLYING SILICON ASSEMBLY TOOLS

by

Jane S. Sun

Memorandum No. UCB/ERL M88/43

14 June 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Acknowledgements

Many thanks to my sister for her spirit, to my mother for her concern, and to my father for giving me a sense of independence, and to all of them for encouraging me to do my best.

I would like to thank Professor Brodersen for his advice and academic guidance, and also to Rajeev Jain for his sustaining counsel and technical support thoughout this project. The suggestions and circuit designs from S. Khalid Azim, Alex Lee, and Robert Neff were valuable to this research.

This research was sponsored by DARPA under the contract N00039-87-C-0182 .

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In designing a computer network, two critical design issues are how the computers should be connected and how to organize the transfer of information. The first issue is an architectural or network topology problem. The second concern deals with the communications protocol. In the end, developing or studying networks then involves network architecture performance testing and protocol verification. As an alternative to conventional methods, the UCB Protocol Workroom examines network performance and operation through a digital hardware facility that emulates network features and properties. The hardware facility consists of node emulators and a channel emulator, as shown in Figure 1.1. The node emulators imitate the protocol behavior of modeled network node stations. The channel emulator emulates the physical characteristics of the transmission medium that links the network nodes, including the network topology. A previous channel emulator implementation using discrete components occupied the areas of several boards but linked just a few nodes. And, herein, lies the motivation for the topic of this report: a VLSI approach toward implementing the channel emulator, which should be a more viable method than through discrete components. A channel emulator, implemented through dedicated hardware, can be applied as an accurate research tool in studying a variety of communications network linking a large number of nodes.

This report describes the ICs for the integrated 10MHz channel emulator implementation, in a scalable cmos process. This includes the chip designs and the

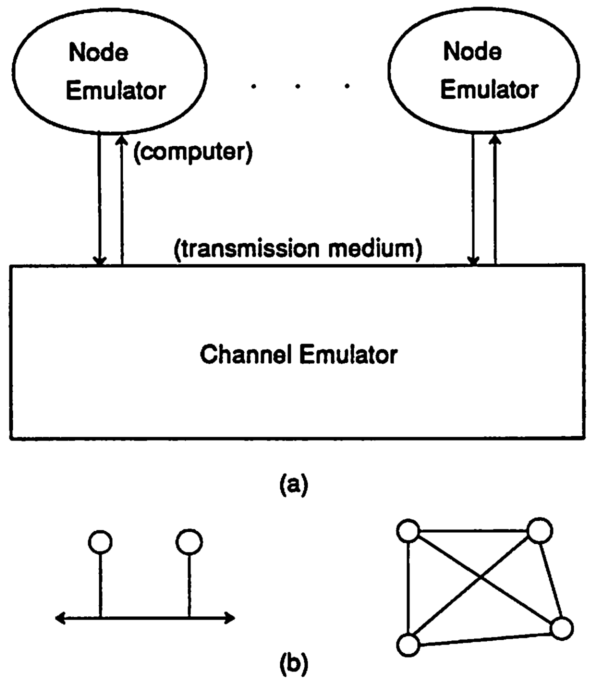


Figure 1.1: (a) Hardware Facility Organization, (b) conceptual networks

design approach. The report focuses on eight areas.

First, it provides an overview of the channel emulator system. At the top level, the system architecture was developed by Amanda Kao and Lester Ludwig [1]. Previously, this architecture was realized with a design geared toward TTL implementation. In this next implementation, the same system architecture is used, but it is realized with VLSI design.

Second, this report places an emphasis on the design approach used to develop the channel emulator hardware. The approach is based on macrocell development, and it applies LAGERIII layout generation tools to assemble macrocells and perform routing.

Next, it describes the four chips developed for the channel emulator. This covers from chip architectural to logic description, and circuit to layout design. Also included are simulation and test procedures, and results from the silicon. This paper pays particular attention to design considerations influenced by design tools and the LAGERIII tools.

The report concludes with comments and suggestions regarding the project and the tools, and areas for future work. It includes a full design cycle example for a chip in the channel emulator system, and a macrocell library listing.

Chapter 2

Overview of the Channel Emulator

A computer network has node stations and the channel, which physically links the stations and transports data. From the moment a node presents data to the channel, the channel has three major features. These are the physical interface between the node and the transmission medium, the network interconnection among the nodes or network point-to-point requirements, and the propagation delay that a signal experiences while traveling through the transmission medium between nodes. The channel emulator emulates those three physical characteristics of the channel in real time at 10MHz.

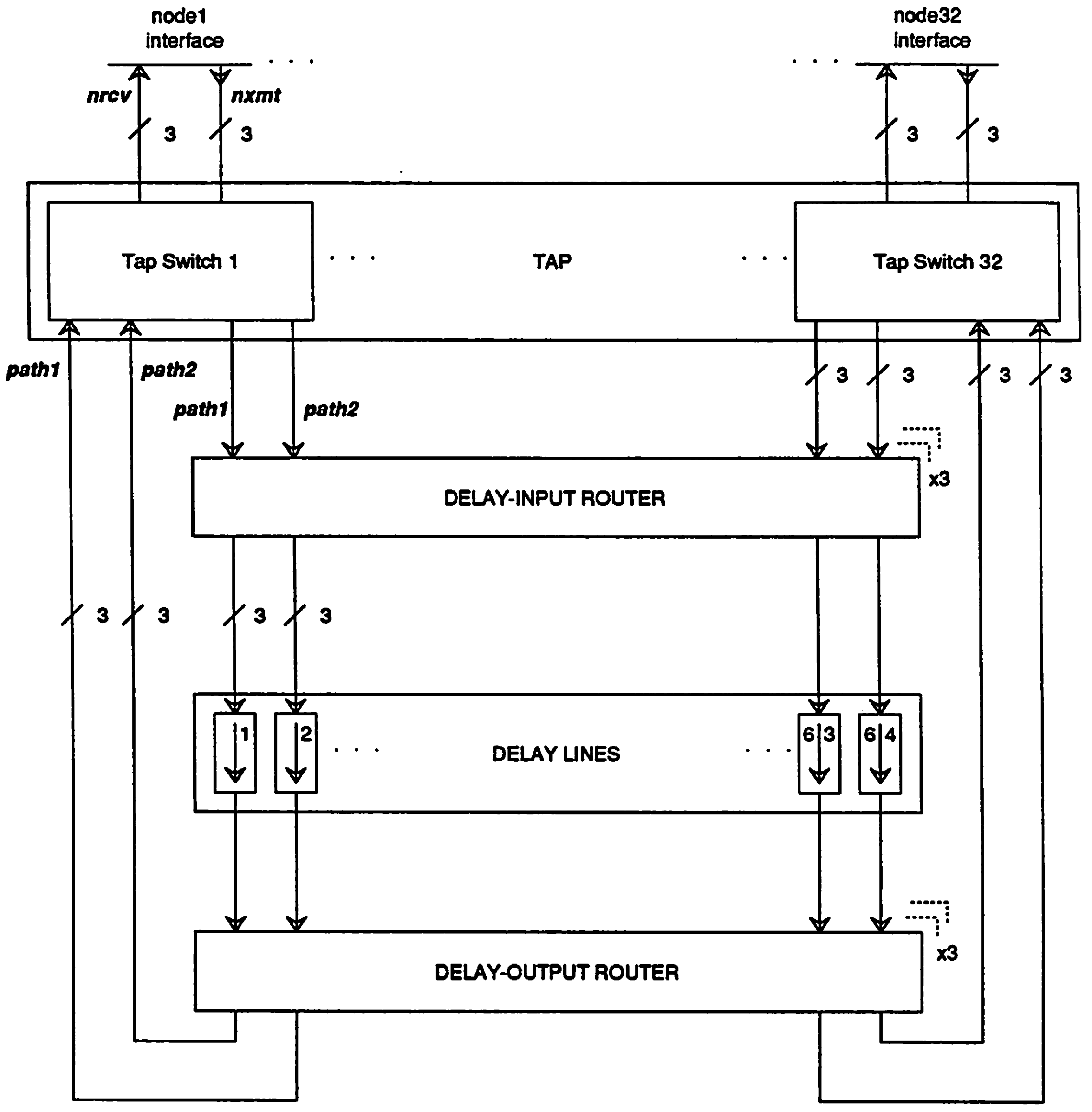
Although all computer networks have nodes and a channel as a common trait, different networks can use different types of transmission mediums, such as coaxial cable or atmosphere, and they can have different network topologies, such as a ring or bidirectional bus. Their point-to point requirements are also different. Since these networks will behave differently, the channel emulator is programmable so that network interface, delay and connections are reconfigurable, thus avoiding the need to change hardware to study different networks. The emulation is performed digitally, assuming that data emerging from the node emulators to the channel emulator is a bit stream. In this manner, the channel emulator is accomplished entirely with digital logic and is capable of emulating the behavior of various transmission mediums and network topologies.

2.1 System Architecture

The channel emulator architecture is shown in Figure 2.1. The system transports a 3-bit wide signal in parallel between the nodes; the three bits are known as *data*, *carrier* and *code violation*. The system is partitioned into four sections: the Tap block, Delay-Input Router, Delay Block and the Delay-Output Router. Each block emulates one of the three channel properties discussed above.

The Tap block acts as the node to channel interface in a network. It emulates how the nodes access or "tap" the network transmission medium. More specifically, it takes signals emerging from network nodes and from the Delay-Output Router to derive the signals transmitted to the nodes and the Delay-Input Router. The block contains Tap Switches placed in parallel, one for each node that the channel must interface with. The Tap Switch is programmable, permitting a reconfigurable interface. It has three ports, and each has input and output lines which are 3-bits wide. There is the node emulator port for the transmit and receive signals of the node emulator, known as *nxmt* and *nrcv*, respectively. The two other ports are called *path1* and *path2* ports. These two are used only in the channel emulator. Extending throughout the emulator, *path1* and *path2* lines carry the 3-bit wide signal from the nodes through the next three blocks for distribution and processing, terminating at their Tap block ports. Figure 2.1 shows these lines and the signal flow through the system.

The Delay block emulates the propagation delay introduced by a transmission medium. The block is a collection of individual Delay Lines which insert programmable delays into the 3-bit wide data stream traveling between nodes. In this system, the propagation time is quantized into clock cycle units, even though transmission times are continuous on a real network channel. The range of delay times extends from 0 to 1023 clock cycles. At a 10MHz clock rate, this translates to a 102.3μs maximum propagation delay in 100ns increments. The outputs of the Delay-Input Router are connected to the inputs of the Delay block with a one-to-



• d,c,cv signals form a 3-bit wide signal

d-data c-carrier cv-code violation

012888

Figure 2.1: Channel Emulator Architecture

one correspondence. After performing its delay operation, the Delay block passes the signals to the Delay-Output Router.

The Delay-Input and Delay-Output Routers realize the point-to-point requirements of the network. Both blocks are programmable, allowing for reconfigurable network connections. The Delay-Input Router takes the signals from the Tap block *path1* and *path2* ports and delivers each signal to an assigned delay line, hence its name as a router. Functionally, an input to the Delay-Input Router is routed to one or more of the Router output terminals. This allows for broadcast or point-to-point networks where a source node connects to several destination nodes.

To complete the node connections, the Delay-Output Router relays the 3-bit wide delayed signal from the Delay block output terminal to the specified Tap Switch interface. Functionally, the Delay-Output Router behaves just like the Delay-Input Router except for an additional "masked OR" feature. This means that the Delay-Output Router can connect several input signals to one of its output terminals. Through the "masked OR" mechanism, the channel emulator can imitate collisions on a network channel such as broadcast or bidirectional bus, where data from several sources on the network may collide at one node. When the output signal from the Delay-Output Router reaches the Tap block, the Tap Switch interface passes the signal to the node emulator or returns it back to the channel emulator. Data from a node may circulate through the channel emulator several times until reaches the destination node.

Overall, the Channel Emulator architecture has a modular structure. For every node that interfaces with the channel emulator, there are the same number of Tap Switches in the Tap block. Processing a 3-bit signal, the Tap Switch is implemented on one chip, and all Tap Switches are identical with input and output ports as described previously. Since the channel emulator has a *path1* and *path2* line for each Tap Switch, the Delay block incorporates two Delay lines for every node the system must interact with. A Delay Line is implemented on one chip, and all Delay lines share the same chip design. Unlike the Tap Switch and Delay Line, the two Router Blocks must gather and distribute data from and to all the *path1* and *path2* lines, rather than processing them individually. Because of this, there is one router for each of the signal types in the 3-bit wide signal. This is shown as three router planes in Figure 2.1. Each Router is implemented on one chip, so a set of three identical chips completes the Delay-Input or the Delay-Output portions. If the number of nodes interfacing with the channel emulator is considered as a parameter n, then implementing an n node channel emulator system with the four IC designs requires n+2n+3+3=3n+6 chips. The ultimate goal is a system that supports a 32 node network. The current work on this project supports an eight node system.

The system architecture discussed provides a brief and adequate description for the purposes of this paper. A detailed treatment is covered in a report authored by Kao and Ludwig [1]. The following sections in this paper describes the Tap Switch, Delay Line and the two Router circuits, which are Crossbar Switches, and the chip design approach.

The designs share common specifications. 10MHz data from the nodes are assumed to be synchronized. The IC designs use a 10MHz two-phase non-overlapping clock. They are implemented in 3μm or 2μm scalable cmos process, and use a +5V power supply.

Chapter 3

IC Design Approach

As functionally unlike the chips may be, all the ICs for the channel emulator were developed with the same design approach. This approach integrates the use of CAD tools into VLSI design. For these circuits, their design can be separated into functional design and physical design. Functional design involves developing the architecture and the logic which carry out the chip's purpose. Here, logic level tools aided in designing the Tap Switch. Physical design includes circuit and layout design. For all the ICs, this design phase heavily applied CAD tools, especially the LAGERIII layout generation tools. Altogether, from architecture to final layout, IC design for the channel emulator was a multistage process. The guideline discussed in this chapter describes the design methodology developed for the chips while clarifying the terminology associated to the CAD tools.

Upon defining the purpose of the chip and its specifications, the designer develops an architecture. Functional blocks called macrocells and their interconnections form the architectural structure, such as a pointer macrocell or memory unit. Smaller functional blocks assembled together make up the macrocell. These blocks may be other macrocells or some unit cells called leafcells. Leafcells are at the bottommost level of the hierarchical structure. For example, a memory macrocell is composed of single bit memory leafcells and periphery circuit leafcells. The hierarchical approach provides a methodical way of translating the the structure into a high level description in *sdl* language or in *.c* routine format, both of which will be explained later.

At this point in the design cycle, a set of necessary macrocells has been defined. Designing the macrocells is the next task. In developing the Crossbar Switches and Delay Line, a bulk of the design effort lies in designing reliable macrocells, since they implement the function of the IC design. In the Tap case, most of the macrocells came from another designer. For all cases, the macrocells are physically constructed by TimLager, a parametric module generator program in LAGERIII. TimLager creates the macrocell layout by tiling predesigned leafcells together. It uses a *.c* routine for specific tiling instructions and reads a *.pdl* file for parameter values that describe the macrocell. The *.c* routine describes how leafcells are oriented and abutted to form the macrocell. For example, an n-stage shift register macrocell is essentially n register leafcells serially abutted. The abutment is described in the *.c* routine. The shift register design applies to a, say, 4 or 8 stage register by assigning the parameter value n=4 or n=8, respectively, in the *pdl* description.

Before applying TimLager, the first step in macrocell design is defining and designing the leafcells. This includes logic and circuit design. The circuit simulators, spice and spice3, are used to choose proper transistor sizes based on estimated load capacitances and speed or timing requirements. Quality of the design using spice depends on the accuracy of the designer's circuit and device model. Leafcell layout is designed manually with magic, an UCB layout editor. The layout is extracted with the magic extractor, and the output file is converted into a format acceptable to esim or spice. Esim is a switch level simulator and is next applied to confirm the leafcell's functionality at the logic level. With input test patterns provided by the user, esim applies the patterns to the extracted layout and observes the output node vectors produced during esim evaluation. The output should match predicted results. Spice can be applied again to verify the circuit performance. In the experience of the writer, the leafcell design methodology discussed here is an iterative process. Normally, none or a few iterations were performed. Though it may seem time consuming, it is worth the gain in a reliable leafcell design.

To complete the macrocell design, the designer provides a *.c* routine and a parametric description. At this design stage, macrocell design may be optimized if its logic design is optimized. CAD tool that perform logic minimization, such as eqntott and espresso programs, are useful here. TimLager generates the final macrocell layout in magic format. For added confidence in the macrocell design, the layout is often extracted, followed by logic level simulation to verify the layout and logic design. Errors detected at the point are normally due to high level logic design faults or connectivity errors at boundaries of abutted leafcells. Again, redesign may be necessary at the leafcell or the macrocell level, reiterating the design methods described so far.

All the macrocells developed for the channel emulator chips reside in a library created by the writer except for those from other designers. Appendix A lists all the macrocells and provides information about their directory location and describing parameters. Detailed discussions about the macrocell logic, circuit and layout appear in the chapter that describes the chip which contains the macrocell.

Since a group of macrocells must communicate, the next step is placing the macrocells and routing the signals between them. Flint is an interactive placement and route tool, requiring information about the macrocells and there interconnectivity from *hdl* files. These files contain macrocell dimension, terminal location along the macrocell boundary, and interconnect information. Fortunately, the *hdl* descriptions files are created by TimLager. The layout generated by Flint can be considered as another macrocell. Flint also creates a *hdl* description for the module it produced. Sometimes, manual place and route can produce better results than this tool sometimes. If manual layout does not require a huge effort or if the tools give results which compromise design reliability, then the manual method is chosen.

To generate the layout for silicon realization, Flint or Padroute are two applicable layout generation techniques. Padroute is an automatic placement and route tool that is dedicated to routing from the functional circuitry in the core of the chip to the pad circuitry along the chip boundary, also known as a pad frame. Treating each side of the pad frame as macrocells and the functional circuitry as another macrocell, Flint or Padroute assembles the final layout, given the macrocell *hdl* descriptions. Preceding this, TimLager assembles the four pad groups from leafcells in an existing LAGERIII cell library.

As the macrocells are designed, the designer must write the input files and individually call the layout generators, whether TimLager or Flint or Padroute, to produce the modules. Design manager(DM), another LAGERIII program, coordinates the efforts if the various layout generators to create the modules in one session. The input to DM is a *sdl* description which describes the architecture of the module to be generated. This is a list of the lower level macrocells, parameter values, and a net list describing the macrocell terminal connectivity. With just the *sdl* description, DM produces all the *pdl* input files necessary to create the macrocells and invokes the appropriate layout generator. so, once a macrocell library with leafcells and *.c* routines has been created, all the separate efforts accorded to each macrocell can be reduced into a single effort by using DM. Macrocells should be completely debugged before applying DM.

The layout produced in the last stage(with Flint or Padroute) is hopefully the final layout. But, errors can creep in here also. These include design rule violations in the layout or functional design errors which originate at the designer's end. For instance, specifying the wrong terminal in a netlist would cause two terminals to be mistakenly connected. Layout errors are normally detected through magic design rule checks and manual connectivity checks using magic. To test the logic of the entire chip design, the layout should be extracted and simulated with esim. Even if simulation results match predicted results this type of simulation does not detect all error possibilities or simulate the circuit performance. But, importantly, it does give the designer confidence in the logic design. This is at least reassuring when beginning tests of the silicon after fabrication.

The approach discussed applies CAD tools as a valuable resource in designing and simulation. It uses an hierarchical layout design based on leafcells and macrocells. As a matter of design reliability, it is wise to monitor the CAD tool results because of bugs in some tools and in the designer's input or faulty design. Although the guideline provided is step by step in nature, the writer emphasizes that designing the ICs was an iterative process. Within any major phase, the design was corrected or adjusted to achieve satisfactory results.

This chapter has stressed the design method. The following chapters describe the chip designs which were developed with the discussed approach. The circuits use minimum size transistors, unless otherwise noted. In the diagrams, Wp denotes the channel width of a pmos device and Wn denotes the width of a nmos device. Lp or Ln signifies channel length.

Chapter 4

Tap Switch

As the node emulators exchange information through the channel emulator, the Tap Switch emulates the interface between the node and the transmission medium, and it transports the 3-bit wide signal (*data*, *carrier* and *code violation*) through its input and output ports at a 10MHz rate. The Tap is programmable so that a variety of interface structures, called tap configurations, can be realized from the same chip design. For example, the bidirectional bus and ring network do not share the same type of interface, as shown in Figure 4.3.

Simulating the node/channel signal interface, the tap switch essentially transforms the signals transmitted from the node or received from the channel into signals passed onto the channel or to the node. At a high level, this transformation can be viewed as the black box of Figure 4.1 which has three outputs - *nrcv*, *p1xt*, *p2xt* - and each output is a combination of the three inputs - *nxmt*, *p1rv*, *p2rv*. Depending on what type of interface is being realized, the specific combination is specified by the 9-bit configuration address, *D*.

4.1 Architecture

Since the *data*, *carrier* and *code violation* signals propagate between node stations in parallel, the node links to the channel with identical tap configurations for all three signals. Accordingly, the Tap Switch architecture, shown in Figure

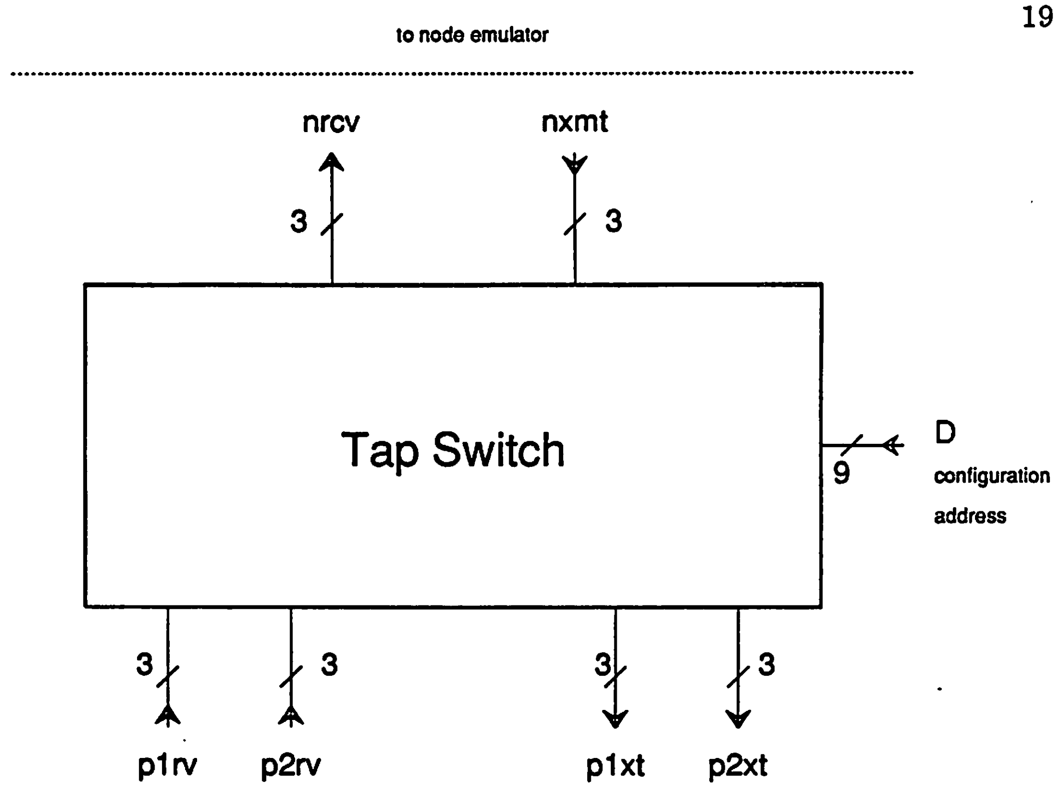


Figure 4.1: High Level View of the Tap Switch

4.2 contains three identical macrocells that separately realize the interface for each signal in the 3-bit wide data stream. The macrocell is referred to as a tap switch and the three instances are called the data, carrier, and the code violation tap switch. In addition to the tap switches, a 9-bit latch stores the common configuration address which controls all three switches. The latch accepts a new address when the signal, *ld*, is true. The address lines are the only interconnections necessary between the macrocells. All other signal lines are purely input and output lines to and from the chip. Figure 4.2 can be considered an expanded view of the block box representation in Figure 4.1.

To digitally implement the tap switch function, the transformation mentioned above can be represented as a boolean expression for each output signal. The *nrcv*, *p1xt*, and *p2xt* output signals are a logical OR combination of the *nrcv*, *p1xt*, and *p2rv* inputs. Different tap configurations use different input combinations. For example, as shown in Figure 4.3, the bidirectional bus interface uses the input combinations, expressed in boolean logic: $nrcv = nxmt{+}p1rv{+}p2rv$, $p1xt = nxmt{+}p1rv$,

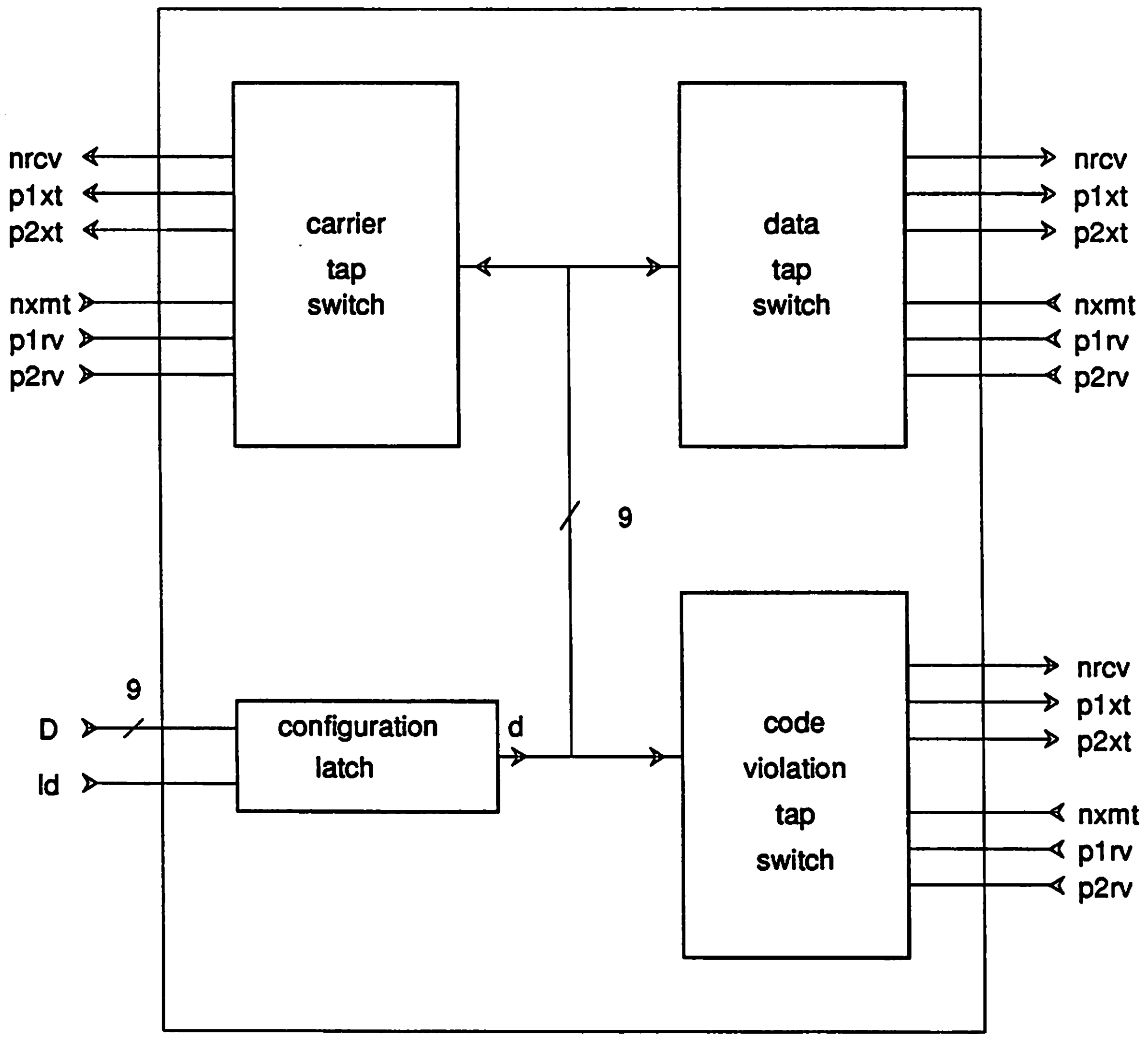
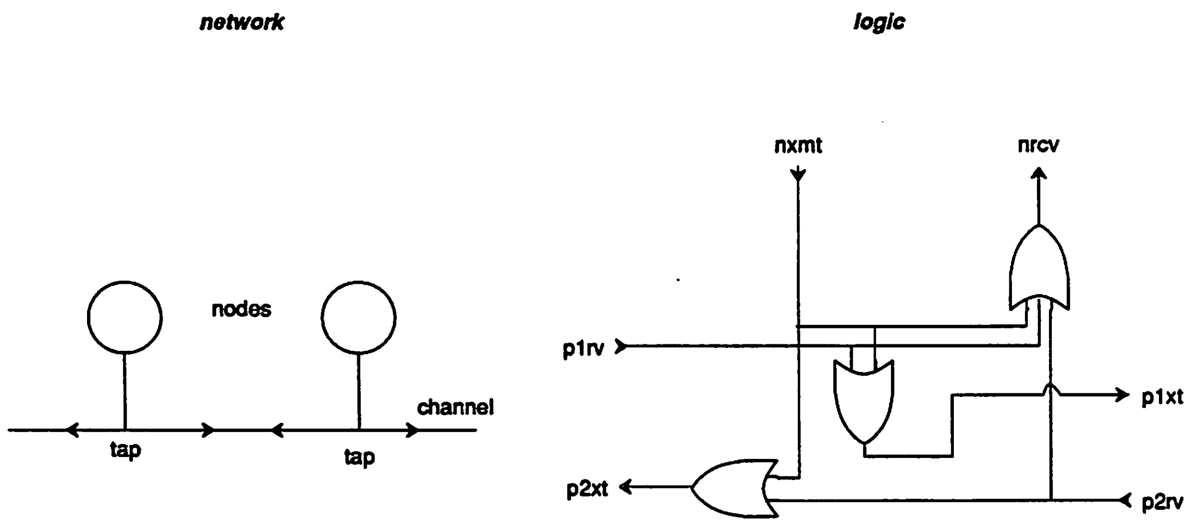
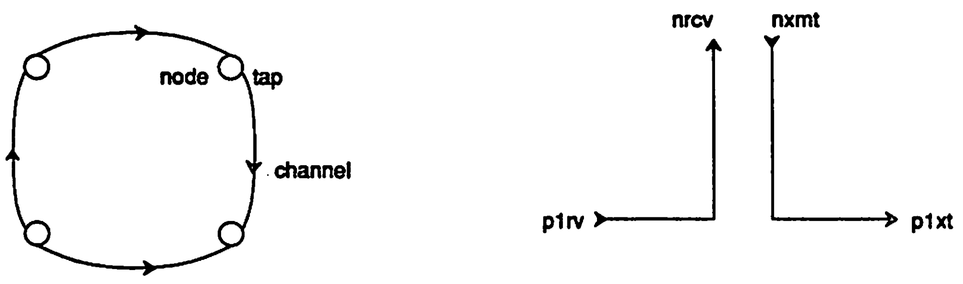


Figure 4.2: Tap Switch Architecture



Bidirectional bus tap configuration



Ring tap configuration

Figure 4.3: Examples of Tap Configurations

and $p2xt = nxmt + p2rv$. While, the input combinations that represent the ring interface are: $nrcv = p1rv$, $p1xt = nxmt$, and $p2xt = don'tcare(not\ used)$. To make the tap switch applicable over a wide range of network architectures, the tap switch contains the unique and necessary input combinations to cover the input-to-output transformations from all network types.

The *nrcv* output needs only five input combinations and two default values to choose from, depending on the tap configuration. These are:

$$p1rv,\ p2rv,\ p1rv + nxmt,\ p2rv + nxmt,\ p1rv + p2rv + nxmt,\ 0, \text{ and } 1.$$

The p2xt and p2xt outputs also select from only five input combinations and two defaults to account for all the existing network types. These combinations are:

$$p1rv,\ p2rv,\ p1rv + nxmt,\ p2rv + nxmt,\ nxmt,\ 0, \text{ and } 1.$$

For each of the three output signals, a 3-bit configuration address is sufficient to determine which input combination the output selects. These three sets of 3-bit signals collectively form the 9-bit configuration address, $D = d_8d_7d_6d_5d_4d_3d_2d_1d_0$, which specifies the tap configuration for a particular network type. The three least significant bits in the configuration address, $d_2d_1d_0$, control selection for the *p1xt* output; the $d_6d_5d_4$ address bits selects for the p2xt output; the three most significant bits, $d_9d_8d_7$, correspond to the *nrcv* output. Table 4.1 shows the set of input signal combinations supported by the tap switch, and it shows the address assigned to the each input combination. To realize the tap interface, the complete boolean equations which describes the tap switch are listed in Table 4.2.

The tap switch logic is implemented with PLA circuitry. There are then three identical PLAs in the chip. This choice of implementation is explained in the next section.

input combination	config. address $d_k d_j d_i$
0	000
1	111
p1rv	001
p2rv	010
nxmt	011
p1rv+nxmt	100
p2rv+nxmt	101
p1rv+p2rv+nxmt	110

Table 4.1: Configuration Address Assignment

output	boolean function
nrcv=	$d_8 d_7 d_6 + \bar{d}_8 \bar{d}_7 d_6 p1rv + \bar{d}_8 d_7 \bar{d}_6 p2rv +$ $d_8 \bar{d}_7 \bar{d}_6 (p1rv + nxmt) + d_8 \bar{d}_7 d_6 (p2rv + nxmt) +$ $d_8 d_7 \bar{d}_6 (p1rv + p2rv + nxmt)$
p2xt=	$d_5 d_4 d_3 + \bar{d}_5 \bar{d}_4 d_3 p1rv + \bar{d}_5 d_4 \bar{d}_3 p2rv +$ $d_5 \bar{d}_4 \bar{d}_3 (p1rv + nxmt) + d_5 \bar{d}_4 d_3 (p2rv + nxmt) +$ $\bar{d}_5 d_4 d_3 nxmt$
p1xt=	$d_2 d_1 d_0 + \bar{d}_2 \bar{d}_1 d_0 p1rv + \bar{d}_2 d_1 \bar{d}_0 p2rv +$ $d_2 \bar{d}_1 \bar{d}_1 (p1rv + nxmt) + d_2 \bar{d}_1 d_0 (p2rv + nxmt) +$ $\bar{d}_2 d_1 d_0 nxmt$

Table 4.2: Tap Switch Boolean Equations

4.2 Macrocell Design and Descriptions

Tap Switch

Realizing the tap switch logic uses PLA design techniques and circuits. This choice of approach over other approaches, such as random logic or standard cell design, was motivated by two reasons. First, a PLA design integrates the tap switch control logic and interface logic into one compact structure, avoiding configuration address decoding in a separate macrocell.

Second, CAD tools applicable toward PLA generation from logic representation to final layout were readily available. At the logic level, the eqntott and espresso programs can convert a set of boolean equations into truth table format and minimize the truth table, optimizing the PLA design. Furthermore, given the truth table description as input, the layout generator tool TimLager can synthesize the final PLA layout from an existing tiling routine and leafcells in a cell library, both developed by Robert Neff [2]. A circuit generated from that cell library had already been fabricated and tested to confirm its circuit operation, adding to confidence in the reliability of the chosen cell library. Overall, the designer needs only to provide the boolean expressions, and the aforementioned CAD tools accomplish PLA design optimization and layout generation, passing a bulk of the design effort to the tools. In the long run, this approach minimizes the redesign effort in case the boolean equations are redefined in the future. This advantage owes to the parametric nature of the applied layout generator. The immediate motivation, however, is drastically reducing chip design time from logic definition to layout phases without compromising circuit performance.

Implementing the boolean expressions in Table 4.2, the tap switch PLA has twelve inputs, since there are nine configuration address bits coming from the 9-bit latch and three tap inputs. It also has three PLA outputs, one for each of the tap outputs. The original boolean equations collectively contained 26 product terms. The PLA truth table was reduced to 20 product terms after transforming the equations into a truth table and minimizing the logic with eqntott and espresso.

Macrocell Description

In the next step toward generating the three PLA macrocells, the 0s and 1s in the minimized truth table are treated as parameters in the TimLager *pdl* description file. TimLager selects the leafcells which corresponds to a 0 or 1 in the truth table, and then tiles them to create the final macrocell layout. Overhead circuitry includes the precharge and evaluate transistors, and the input and output drivers. The PLA circuit is clocked with a two phase non-overlap clock signal. Its AND and OR planes are implemented with NOR circuitry in the PLA.

This completes the tap switch physical layout, resulting in a tap switch with physical dimensions 348λ x 435λ. The 12 inputs and 3 output terminals appear on the same side of the macrocell. For this tap switch macrocell, the parameters in the TimLager *pdl* description are *in*(number of inputs), *out*(number of outputs), *minterm*(number of rows in truth table), *out-plane*(contents of output plane in truth table), and *in-plane*(contents of input plane in truth table). The *tapcore.parval* file, shown in Table 4.3 lists the assigned parameter values describing this tap switch. The PLA layout generated by TimLager is shown in Figure 4.4. The input and output terminals appear at the top edge of the PLA structure. Just below these terminals are the input and output drivers. The truth table contents are in the bottom half of the layout. The transistor locations can be seen from inspection.

Address Latch

The address latch stores the 9-bit tap configuration address. It consists of 9 individual latches stacked so that the address bits can be accepted in parallel, as instructed by a control signal, and read out in parallel.

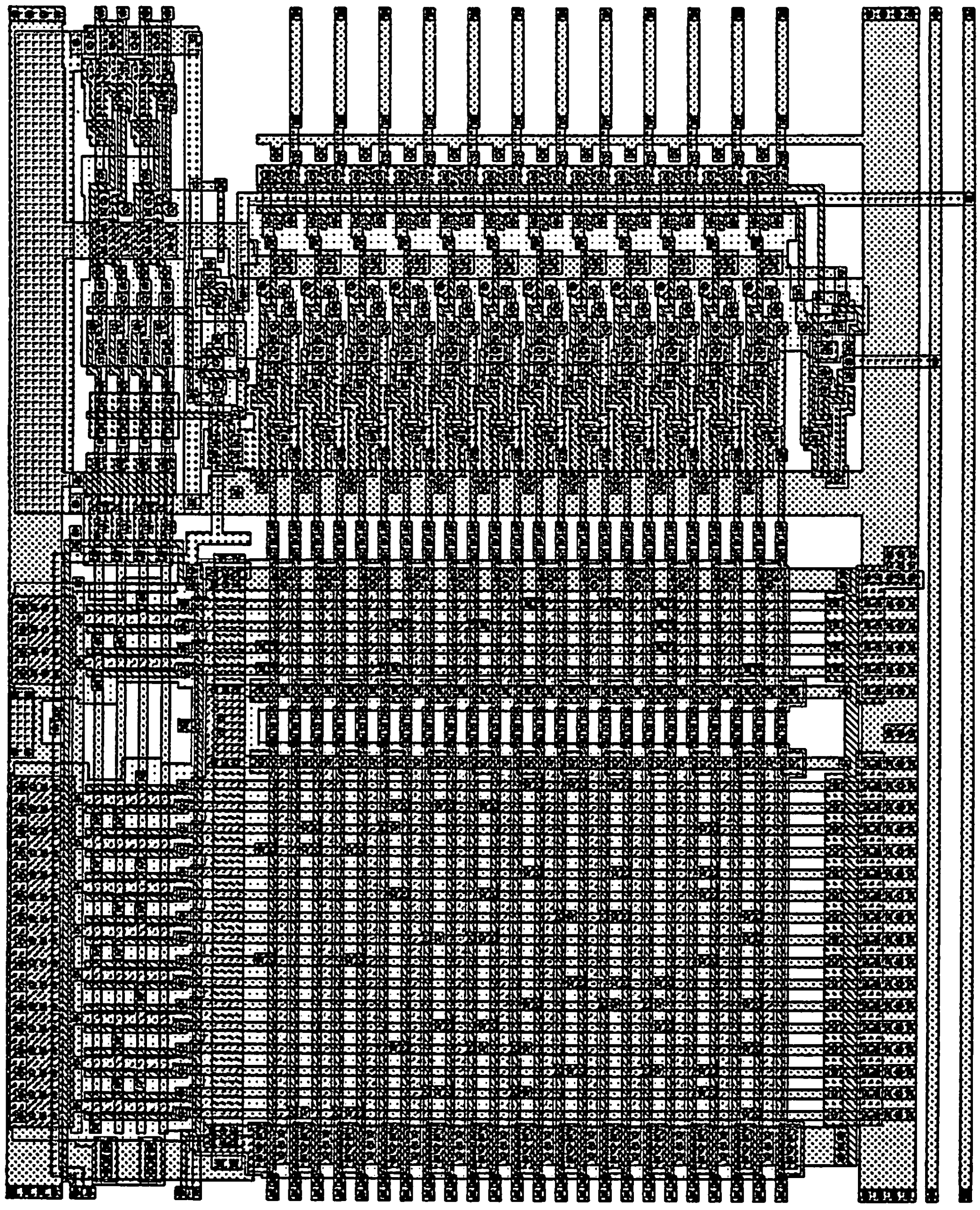
Leafcell Design

The circuit for the one bit latch is shown in Figure 4.5 It is a two stage dynamic register with two extra cmos transmission gates. The dynamic register is clocked by a two phase non-overlapping clock signal at the input of inverters A and

Table 4.3: tap.parval

```
;tap switch paramaters and values
;
(in 12)
(out 3)
(minterm 20)
(nbits 9)
(output-plane
;  array: nrcv p2xt p1xt
       ((array |100| )
        (array |010| )
        (array |001| )
        (array |010| )
        (array |010| )
        (array |001| )
        (array |001| )
        (array |100| )
        (array |010| )
        (array |001| )
        (array |010| )
        (array |001| )
        (array |100| )
        (array |100| )
        (array |010| )
        (array |001| )
        (array |100| )
        (array |100| )
        (array |010| )
        (array |001| ))
)
(input-plane
;  array: d8 d7 d6 d5 d4 d3 d2 d1 d0 nxmt p2rv p1rv
       ((array |001xxxxxxxx1| )
        (array |xxx010xxxx1x| )
        (array |xxxxxx010x1x| )
        (array |xxx100xxxxx1| )
        (array |xxxx11xxx1xx| )
        (array |xxxxxx100xx1| )
        (array |xxxxxxx111xx| )
        (array |1x1xxxxxxx1x| )
        (array |xxx001xxxxx1| )
        (array |xxxxxx001xx1| )
        (array |xxx1x1xxxx1x| )
        (array |xxxxxx1x1x1x| )
        (array |111xxxxxxxxx| )
        (array |x10xxxxxxx1x| )
        (array |xxx111xxxxxx| )
        (array |xxxxxx111xxx| )
        (array |1x0xxxxxxxx1| )
        (array |1xxxxxxxx1xx| )
        (array |xxx10xxxx1xx| )
        (array |xxxxxx10x1xx| ))
        )
```

Figure 4.4: Tap Switch PLA Layout



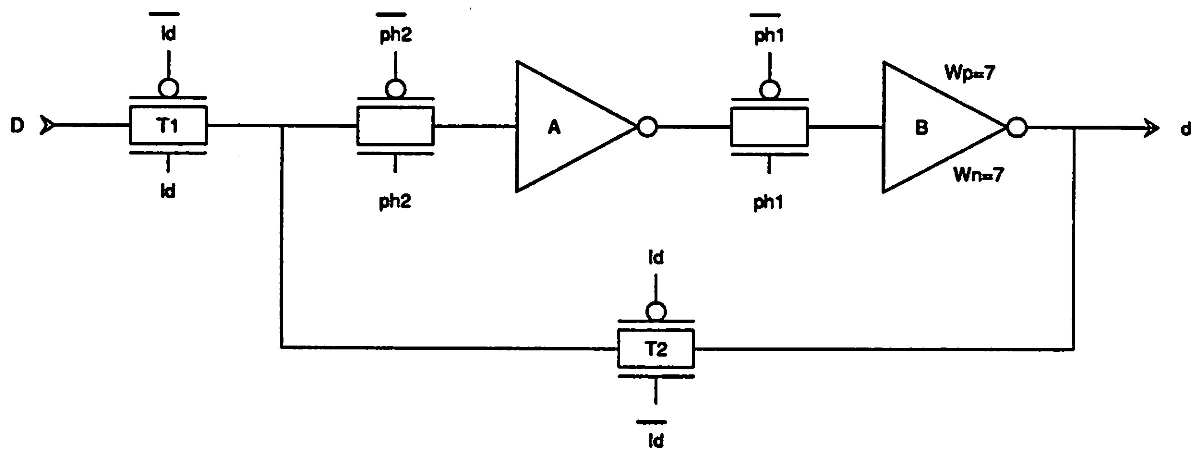


Figure 4.5: Latch Leafcell Circuitry

B. The two transmission gates T1 and T2 load a new address bit into the register from the input *D* during the high of the *ld* signal, which must cover a *ph2* phase pulse; the feedback loop is disconnected when the latch loads a bit. At the falling edge of the *ld* signal, the new address bit is fully latched in and the output at the *d* terminal is valid; the feedback loop is connected during the low of *ld*, preventing charge leakage at the dynamic nodes within the register. Latching in a new address bit requires one clock cycle. In addition to the latch cell, there is an extra cell which produces the *ph1-*, *ph2-* and *ld-* signals from the true signals.

The layout for the latch leafcell appears in Figure 4.6. Transistors are located in the center. The clock and *ld* or *ld-* control lines run vertically along the sides while the power and ground lines run vertically through the center. These lines will naturally join with the matching lines in the cell above and below it in the final multibit latch layout, forming straight and continuous clock, control, and power lines.

Macrocell Description

The address latch is physically laid out as instances of the latch leafcell

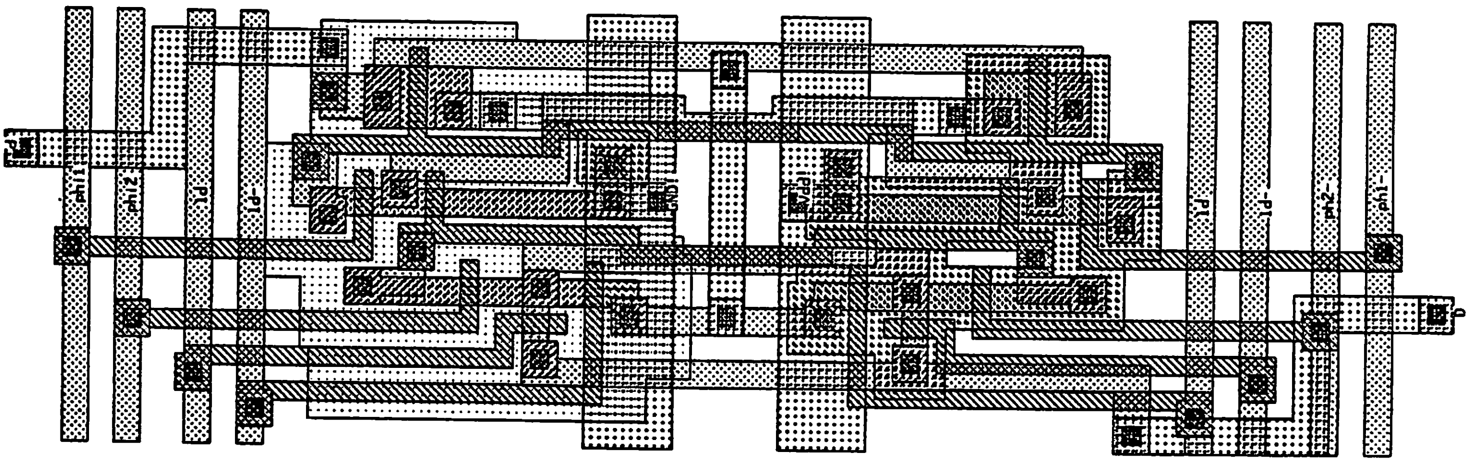
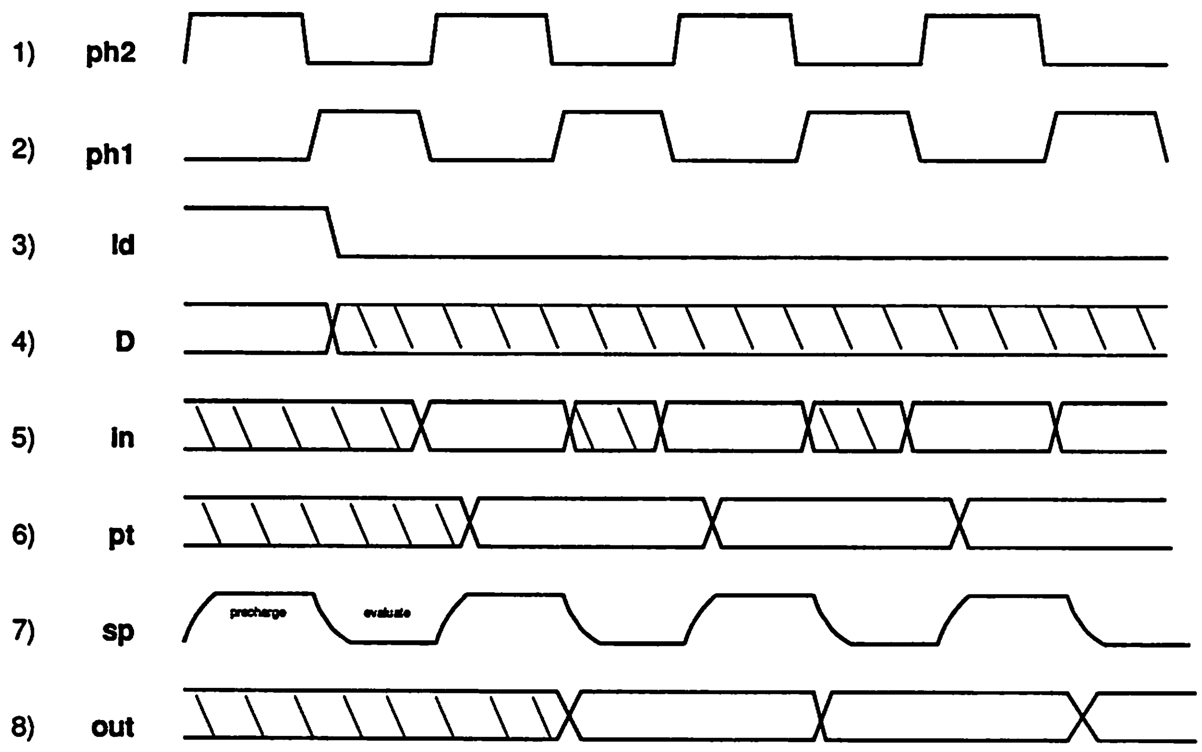


Figure 4.6: Latch Leafcell Layout

stacked on top of each other. To create this structure, TimLager uses the parameter *nbits* which is the number of one-bit latches in the macrocell. The parameter is assigned a value of nine in this case.

4.3 Timing Requirements

The tap switch and latch macrocells have been described. For these circuits to work together they must follow the requirements shown in the timing diagram of Figure 4.7. An external source supplies the two clock phases, *ph1* and *ph2*. At the time of this design, no circuits were included to generate these phases locally. According to the convention set for the channel emulator hardware, data appearing at a clocked circuit input must be valid over the entire *ph2* pulse; data appearing at a clocked circuit output changes within the *ph1* pulse. In this way, clocked outputs are compatible as inputs to other clocked cells. This convention applies to the Tap Switch inputs, which are *ld* for programming, the nine address bits, and the nine data bearing signals. Programming the address into the latch requires one clock cycle. The Tap Switch PLA also requires one full clock cycle to evaluate the boolean functions for each set of sampled inputs.



in	input to tap switch
pt	product terms
sp	sum of products
out	output from tap switch

Figure 4.7: Timing Diagram for the Tap Switch

4.4 Chip Synthesis from Macrocells

So far, generating the address latch and the three tap switches only partially realizes the Tap architecture. Placing and interconnecting the four macrocells together and routing the chip input and output signals to the proper input or output pads completes the chip layout.

In the first Tap chip generated by the designer, these last two phases were performed manually with the magic layout editor. In the second chip version, the chip layout was completely automated with the LAGERIII tools; the last two phases were performed by Flint and Padroute, respectively. The macrocell design remained unchanged. As a comparison, Figure 4.8 and Figure 4.9 shows the fabricated chip for both versions. The core circuitry of the first version has dimensions 945λ x 1090λ after interconnections. The core circuitry of the second version has dimensions 945λ x 1115λ, comparable to the first version. Both versions have the same floorplans, but their macrocells are oriented differently. Routing in the second version was hand edited in magic to eliminate design rule violations and unnecessary jogs created by the automatic layout tools. The routing between macrocells looks neater in the first chip than the second chip, and the same observations hold true for routing between the core and the I/O pads. However, with the full aid of the LAGERIII tools, chip design time is significantly reduced compared to the time required for a semi-automated design approach, which makes the automated approach preferable from a development time standpoint.

A list of the pads in each side of the pad frame appears in Tables 4.4, 4.5, 4.6, and 4.7. This list applies to the second version of the Tap Switch.

4.5 Simulations

Verifying the Tap design with esim, a switch level simulator, was a two stage process. First, the macrocell layouts and latch leafcell were extracted with the magic extractor and converted into a format acceptable to esim. Results from esim confirmed the functionality of the address latch and the Tap Switch PLA. Spice

Figure 4.8: Photograph of Tap Switch, Version 1

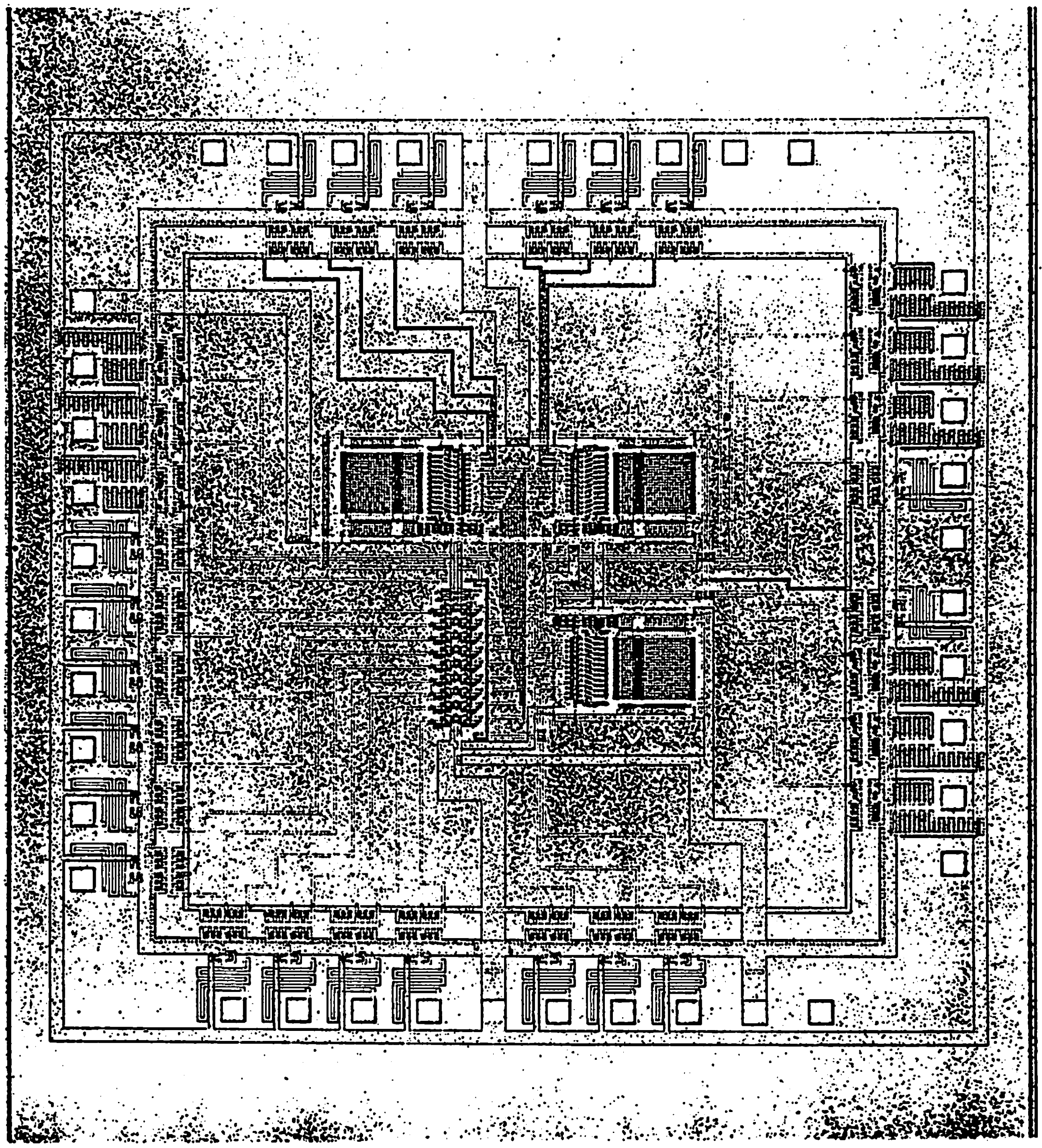
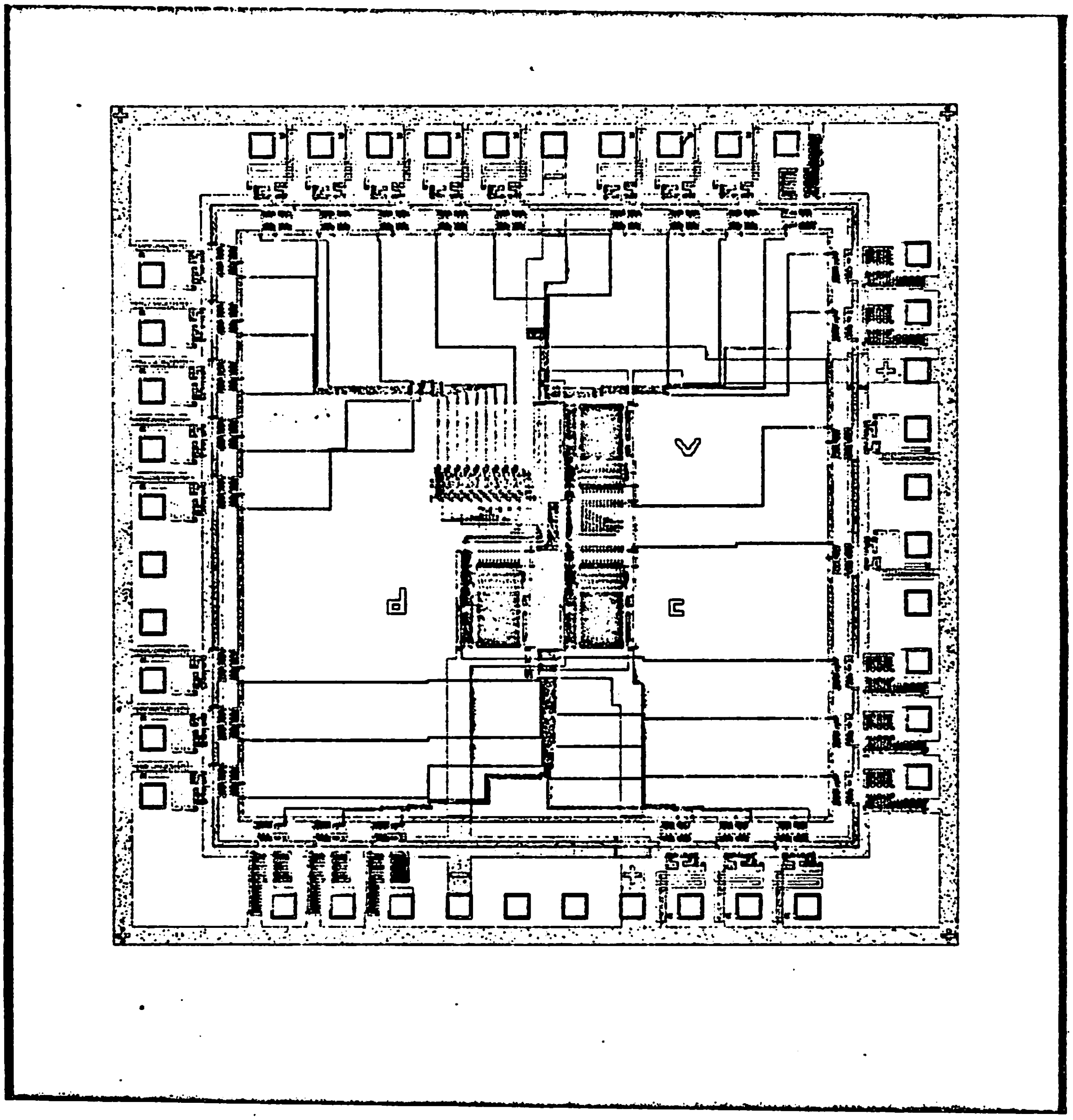


Figure 4.9: Photograph of Tap Switch, Version 2



North Side Pads from Left to Right		
name	type	description
D5	input	d_5 bit of configuration address
D6	input	d_6 bit of configuration address
D7	input	d_7 bit of configuration address
D8	input	d_8 bit of configuration address
ld	input	load signal, resets address latch
GND	GND	
p1rv.v	input	p1rv signal to code violation tap switch
p2rv.v	input	p2rv signal to code violation tap switch
nxmt.v	input	nxmt signal to code violation tap switch
p1xt.v	output	p1xt signal from code violation tap switch

Table 4.4: Pads on North Side of Pad Frame

East Side Pads from Top to Bottom		
name	type	description
p2xt.v	output	p2xt signal from code violation tap switch
nrcv.v	output	nrcv signal from code violation tap switch
Vdd	Vdd	
phi1	input	ph1 clock phase
substrate	dummy	chip substrate contact
phi2	input	ph2 clock phase
nrcv.d	output	nrcv signal from data tap switch
p2xt.d	output	p2xt signal from data tap switch
p1xt.d	output	p1xt signal from data tap switch

Table 4.5: Pads on East Side of Pad Frame

South Side Pads from left to right		
name	type	description
p1xt.c	output	p1xt signal from carrier tap switch
p2xt.c	output	p2xt signal from carrier tap switch
nrcv.c	output	nrcv signal from carrier tap switch
GND	GND	
Vdd	Vdd	
p1rv.d	input	p1rv signal to data tap switch
p2rv.d	input	p2rv signal to data tap switch
nxmt.d	input	nxmt signal to data tap switch

Table 4.6: Pads on South Side of Pad Frame

West Side Pads from Top to Bottom		
name	type	description
D4	input	d_4 bit of configuration address
D3	input	d_3 bit of configuration address
D2	input	d_2 bit of configuration address
D1	input	d_1 bit of configuration address
D0	input	d_0 bit of configuration address
p1rv.c	input	p1rv signal to carrier tap switch
p2rv.c	input	p2rv signal to carrier tap switch
nxmt.c	input	nxmt signal to carrier tap switch

Table 4.7: Pads on West Side of Pad Frame

confirmed the circuit performance of the latch. Second, the entire chip was extracted and simulated from the I/O pads. To emulate real node to channel interfaces, tap configurations were specifically set to the bidirectional bus and ring configurations. Random input test vectors were applied and the output nodes were monitored for discrepancies between expected results and simulated results.

4.6 Testing

The first Tap circuit was fabricated in $3\mu m$ pwell scmos technology, returned in April, 1987, and was fully functional. Testing the silicon followed the same philosophy as for logic simulation on the extracted layout. Copies of the latch and a tap switch PLA were placed on a separate test chip. Although latch outputs connect to the PLA inputs in the Tap design, the test chip maintained independent input and output lines. After fabrication, the latch and PLA were tested individually and their performance and functionality confirmed. With confidence in the macrocells, the fabricated Tap circuit was tested next using random input test vectors and various tap configurations.

The chip test pattern consists of test vectors for the data bearing input signals - *nxmt*, *p2rv* and *p1rv* and the programming signals - the latch *ld* signal and configuration address *D*. An objective of testing is to simulate the real working environment that the IC functions in. A sufficient test pattern, which was not exhaustive, was designed with this in mind. In a normal operating environment, the data bearing signals are random. To simulate the randomness, the three bit wide pattern which represents *nxmt*, *p2rv* and *p1rv* signals cycles through all eight possibilities. One cycle is applied for various tap configurations. When the configuration changes, the load and address vectors take on the proper bit pattern for one clock cycle and then become a pattern of 0s until the Tap circuit is reprogrammed again. This tests if the latch accepts and stores the address properly. The test patterns cover eight tap configurations including the bidirectional bus and ring.

Testing the chip begins at low clock rates, say 1MHz. The purpose is to verify functionality. Then, to test circuit performance, the test pattern described is repeatedly applied as the clock rate is adjusted toward 10MHz. From test results, the first Tap circuit worked up to 7MHz. Beyond that speed, two of the three identical tap switches worked up to 9MHz while one fails, exhibiting wrong output bits . Reexamining the layout did not provide a reason. It is known that the larger ROM designed by Robert Neff using the same leafcells as the PLA tap switch had worked up to 11MHz. So, the writer suspects that the first tap switch was fabricated on a slow run.

The second Tap circuit, which was fully laid out with CAD tools, was submitted for fabrication in 2μm pwell scmos technology. It was returned in October of 1987. Using the same test methods and test patterns as in the previous Tap Switch, the second Tap Switch was verified to be fully functional at speeds up to 20MHz. The increase in operational speed compared to the first Tap is attributed largely to the down scaled technology used for the second Tap.

Chapter 5

Variable Register Delay Line

To simulate the propagation delay introduced by the transmission medium, the Variable Register delay Line inserts an artificial delay into an input signal. From a DSP viewpoint, it transforms and input signal X into the output Y using the relation $Y = z^{-N}$, where N is the amount of delay. The delay operation is implemented with shift registers, hence, its name. Figure 5.1 shows the Delay Line at at the top level. The input signal is three bits wide, consisting of the *data*, *carrier* and *code violation* bits. The amount of delay the signal experiences is variable, ranging from 0 to 1023 clock cycles of delay in increments of one clock cycle. The delay is derived from a 10MHz two phase non-overlapping clock signal. A 10 bit address, Q, called the delay value address, specifies the amount of delay.

5.1 Architecture

Just as in the Tap Switch design, the Delay Line architecture contains four structures, as shown in Figure 5.2. In this partition, The Delay Value Address Latch loads and stores the ten bit address, as instructed by the *reset* signal. The address is directly distributed to the delay units. A delay unit introduces the programmable delay into the input bit stream and sends the signal out. It contains the shift registers and control circuitry. Since the input signal to the Delay Line carries three parallel bits, the delay unit is repeated three times for the *data*, *carrier*, and

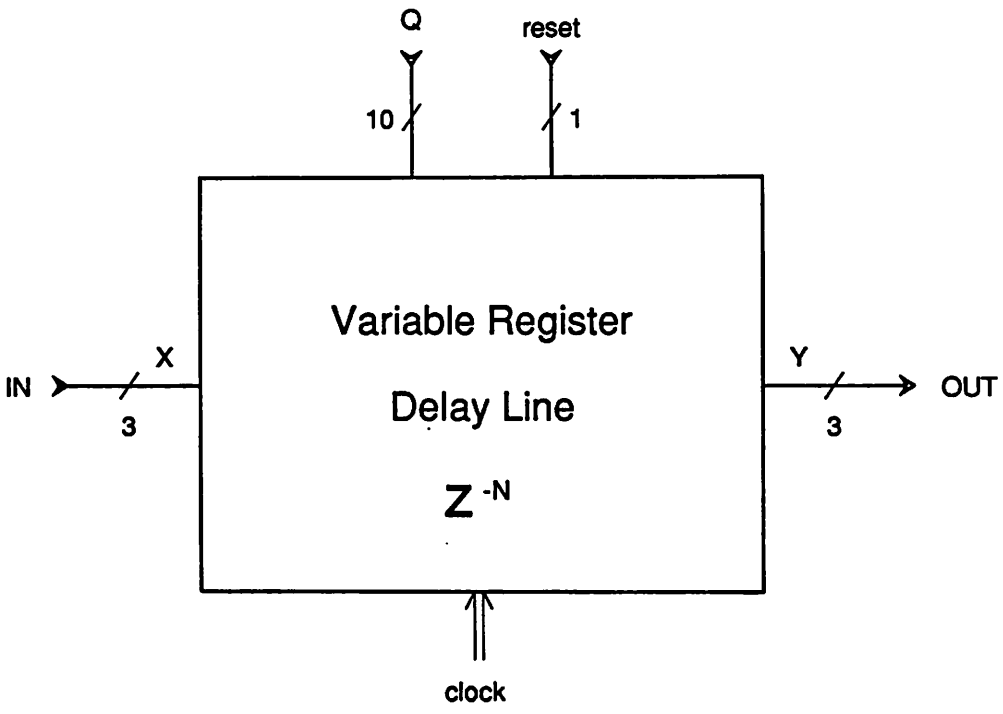


Figure 5.1: High Level View of Delay Line

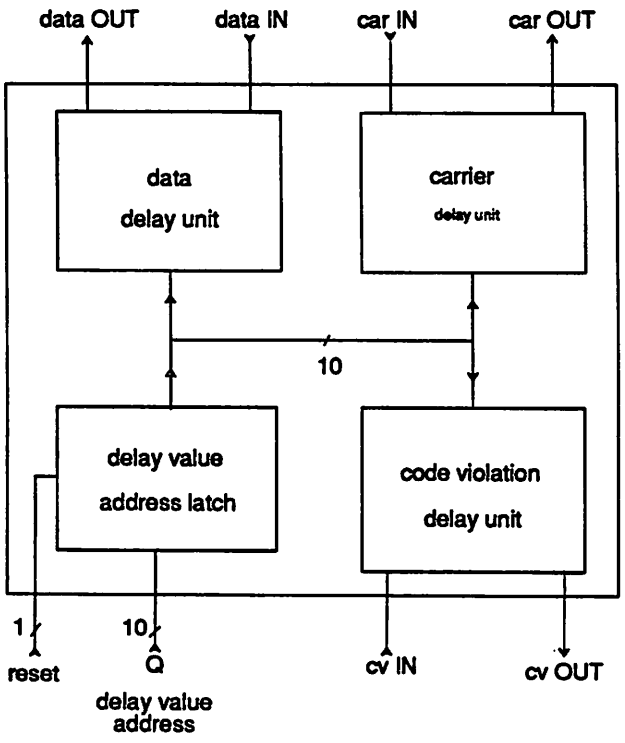


Figure 5.2: Variable Register Delay Line Architecture

code violation bit, so each unit has identical circuitry dedicated to delaying its bit stream. For simplicity, only one delay unit needs to be described in detail.

To understand how the delay unit works, we can compare it to the following basic algorithm. Suppose the amount of delay N inserted into the data stream is represented by a 10 bit delay value $Q = q_9 \ldots q_0$. These 10 bits specifies delays up to $2^{10} - 1 = 1023$, or the range $0 \leq N \leq 1023$. Arithmetically, the relationship between N and the 10 bit number is:

$$N = q_9 \cdot 2^9 + q_8 \cdot 2^8 + \ldots + q_2 \cdot 2^2 + q_1 \cdot 2^1 + q_0 \cdot 2^0.$$

where q_9 is the most significant bit.

Simply stated, the value N is the sum of the appropriate powers of two. For example, the delay value of 9 is specified by the address, $Q = 0000001001$ since $9 = 1 \cdot 2^3 + 1 \cdot 2^1$. The delay unit in a similar fashion converts the 10 bit address into the assigned delay of N.

Figure 5.3 shows an expanded view of the architecture and illustrates the delay unit. The delay unit has 1023 registers partitioned into successive powers of two. Each partition is called a delay line and is implemented with a shift register of length 2^m. So, the delay unit contains 10 shift registers of lengths 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. To introduce the desired amount of delay, the input data stream is successively directed through the appropriate delay lines as specified by the delay value address, starting at the least significant delay line. For the aforementioned example, the input signal passes through the delay line of lengths 1 and 8.

Directing data to the proper delay lines is managed by the 2:1 multiplexers in a ten stage control section. A multiplexer is assigned to each delay line. The inputs to the multiplexer are the output of the assigned delay line and the output from the previous multiplexer. The multiplexer select signal, s, is the address bit, q_i, corresponding to the delay line with the appropriate power of two. As a specific example, the fourth delay line which has length $8 = 2^3$ is assigned a multiplexer which is controlled by the address bit q_3. For the first stage, the signal IN arrives at an input to the first multiplexer and also propagates through the first delay line.

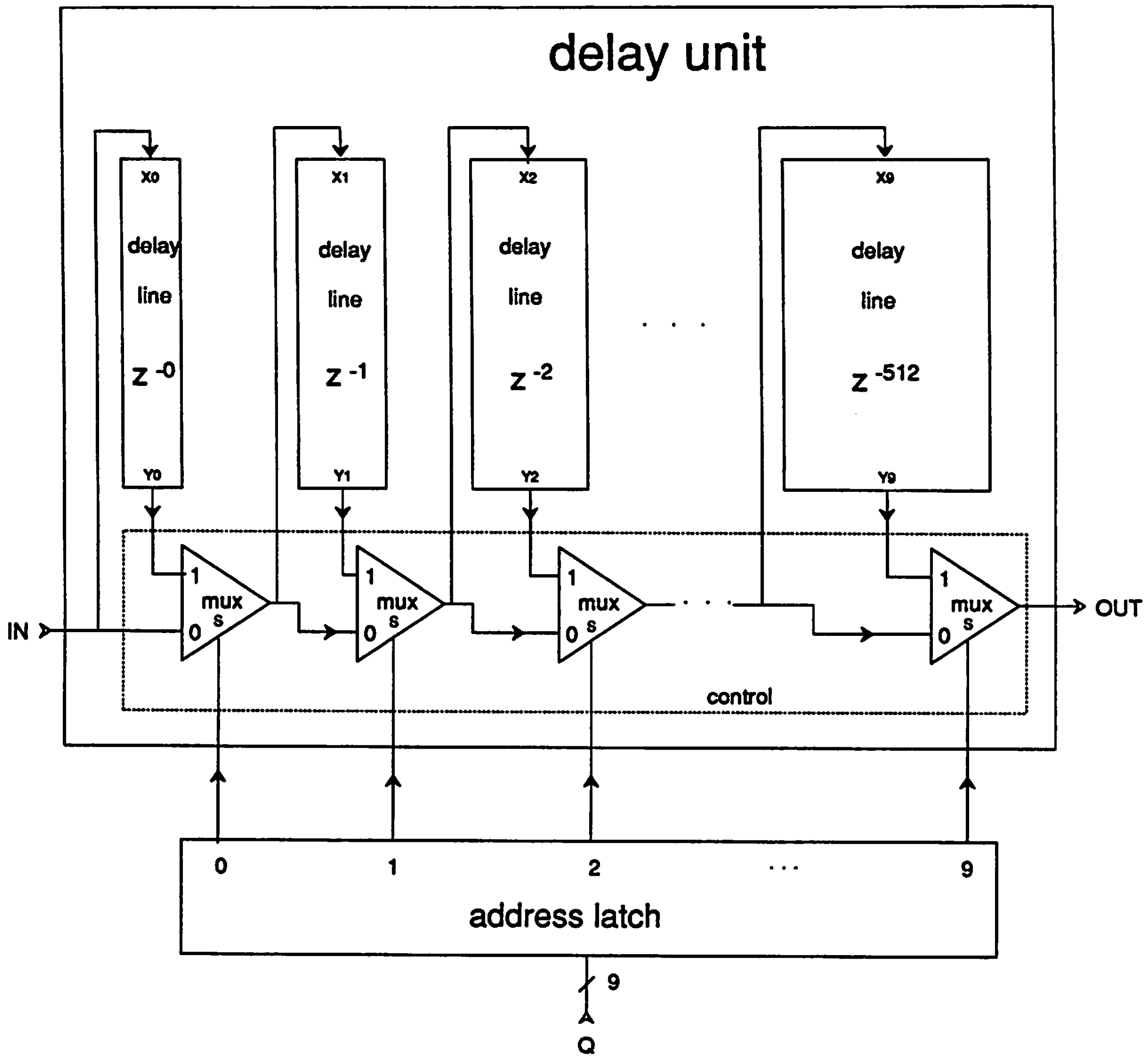


Figure 5.3: Delay Unit

The multiplexer selects the incoming signal if the least significant address bit is a 0, or it selects the delayed version if the address bit is a 1. The multiplexer output appears at the input of the second stage. This stage makes the same decision based on the value of q_2, and the same applies to the subsequent stages based on their address bit. In this manner, the multiplexers force the input data stream to bypass delay lines or to pass through them depending on their assigned address bit. The signal accumulates the appropriate delays until it is delayed by N clock cycles. The fully delayed signal OUT is delivered at the output of the tenth multiplexer.

Albeit 1023 registers occupy more area than say a RAM based implementation, the architecture described is easy to implement since a delay unit requires only a shift register macrocell, of which there are 10 instances, and a multiplexer control macrocell to carry out the delay operation. It also avoids decoding the 10 bit delay value address, considerably simplifying control circuitry.

5.2 Macrocell Design and Description

Delay Line

The delay line macrocell is a shift register formed from a chain of register leafcells. A register delays its input by one clock cycle, and its output is the input to the following register. At the last register in the chain, the output signal is a delayed version of the input to the first.

Register Leafcell

The leafcell circuit of Figure 5.4 is a two staged dynamic register which stores the signal level at the inverter inputs. Rather than using conventional cmos pass gates, data is clocked through nmos pass gates. In the long run, this choice over the full cmos choice reduces cell area since two pmos transistors, the *ph1-* and *ph2-* clock signals, and associated contacts are eliminated. Of course, this design compromises noise immunity because it suffers from body effect at the nmos drains. But, the tradeoff is well worth the gain in layout compactness, especially when the

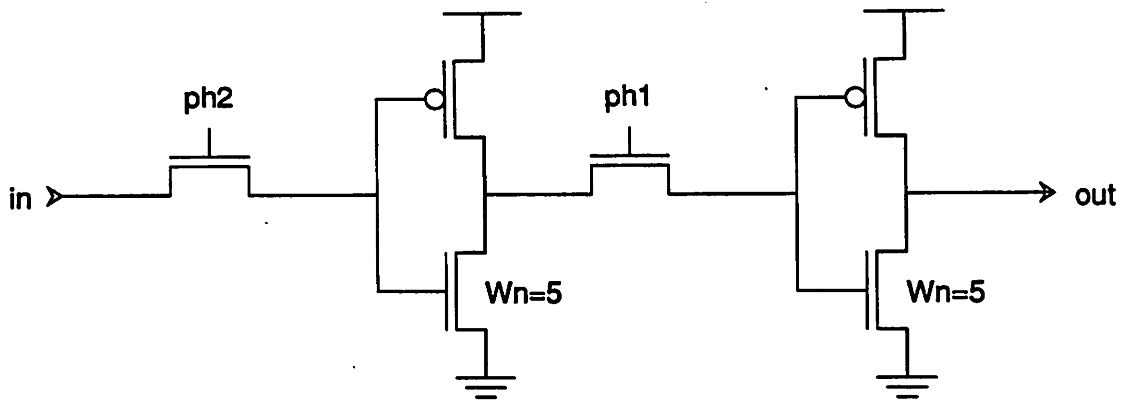


Figure 5.4: Register Leafcell Circuitry

architecture requires 1023 registers in each of the three delay units.

Focusing on circuit design, the nmos pass gate transmits a low level perfectly, so the inverter output can fully achieve a 5V level. According to spice simulations, body effect degrades a 5V signal level at the nmos pass gate input to 2.7V at the output which appears at the input to the inverter. To maintain logic integrity, the inverter nmos device must sink more current than the pmos device so that the inverter pulls its output node sufficiently low. Fortunately, with nmos W/L=5/2 and pmos W/L=3/2(minimum size), the inverter pulls its output node down to .2V, safely below the threshold voltage of the next nmos transistor.

The register leafcell layout is shown in Figure 5.5. Clock lines run in metal2 perpendicular to the metal1 ground and power lines. Input and output terminals are naturally at opposite ends of the cell. All these lines are placed so that they meet the same signal lines in neighboring cells.

Macrocell Description

For the delay line macrocells of longer lengths, strictly laying the cells in series would be dimensionally ridiculous. Instead, the leafcells are placed in series

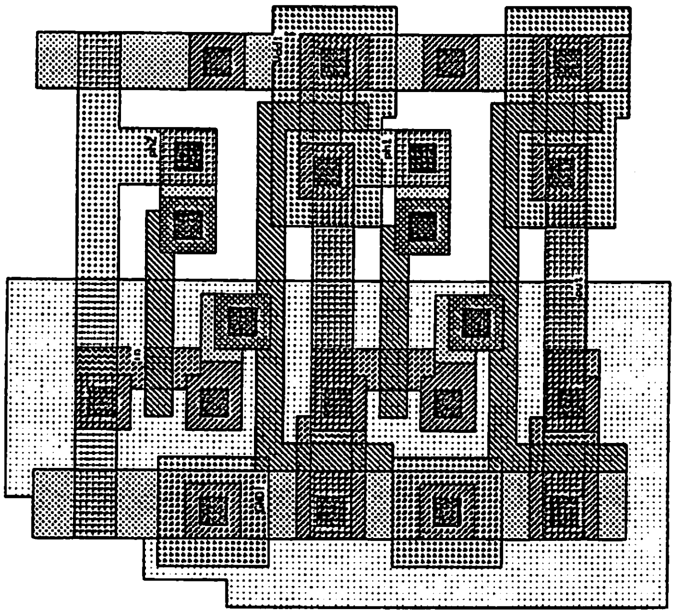


Figure 5.5: Register Leafcell Layout

and "snaked" into multiple rows to physically achieve a rectangular macrocell. The parameters in the TimLager *pdl* description are *rows*, *cols*, and *indx*. TimLager generates a macrocell with 2^m leafcell instances arranged in a *rows*x*cols* array, where 2^m = *rows*x*cols* and m = *indx*. The *indx* parameter serves labeling purposes. TimLager labels the input of the first register as `X[m]` and the output of the last register as `Y[m]`. The delay line of length 1 and 2 are manually created. For example, the delay line of length 32 has 32 registers cascaded into 4 rows and 8 columns, and `X[5]` and `Y[5]` designate the input and output terminals, respectively.

Table 5.1 shows the *sdl* description for the Delay Line core. It lists the 10 delay line macrocells, referred as dm which identifies their length, in the delay unit and the specific parameter values that describe their dimensions.

Clock Driver

Inside the delay unit, the clock driver macrocell takes an incoming two phase clock signal and generates the buffered version from it for the delay line

Table 5.1: delay line *sdl* description

```
;dcore.sdl
; use Flint to call TimLager from macrocell .sdl files.
(layout-generator Flint)
(parent-cell dcore)
(sub-cells
        (ckdriver ckdriver (parameters (cols 32)))
;       * dline d1 and d2 are manually created *
        (dline d4 (parameters (indx 2) (cols 2) (rows 2)))
        (dline d8 (parameters (indx 3) (cols 2) (rows 4)))
        (dline d16 (parameters (indx 4) (cols 4) (rows 4)))
        (dline d32 (parameters (indx 5) (cols 8) (rows 4)))
        (dline d64 (parameters (indx 6) (cols 32) (rows 2)))
        (dline d128 (parameters (indx 7) (cols 32) (rows 4)))
        (dline d256 (parameters (indx 8) (cols 32) (rows 8)))
        (dline d512 (parameters (indx 9) (cols 32) (rows 16)))
        (pgmux muxctl (parameters (m 10)))
)
;
;--delay unit internal and external connectivity specifications
;give net_name parameter connectivity: -parent means connection
;                                          to outside world.
;                                       -subcell means internal connection.
;
(net D 10 ((parent D) (dvalue_latch D)))
(net d 10 ((dvalue_latch d) (muxctl S)))
; * X0 and X1 are manually created
(net X2 ((muxctl X 2) (d4 X 2)))
(net Y2 ((muxctl Y 2) (d4 Y 2)))
(net X3 ((muxctl X 3) (d8 X 2)))
(net Y3 ((muxctl Y 3) (d8 Y 2)))
(net X4 ((muxctl X 4) (d16 X 4)))
(net Y4 ((muxctl Y 4) (d16 Y 4)))
(net X5 ((muxctl X 5) (d32 X 5)))
(net Y5 ((muxctl Y 5) (d32 Y 5)))
(net X6 ((muxctl X 6) (d64 X 6)))
(net Y6 ((muxctl Y 6) (d64 Y 6)))
(net X7 ((muxctl X 7) (d128 X 7)))
(net Y7 ((muxctl Y 7) (d128 Y 7)))
(net X8 ((muxctl X 8) (d256 X 8)))
(net Y8 ((muxctl Y 8) (d256 Y 8)))
(net X9 ((muxctl X 9) (d512 X 9)))
(net Y9 ((muxctl Y 9) (d512 Y 9)))
(net IN ((parent IN) (muxctl IN)))
(net OUT ((parent OUT) (muxctl OUT)))
```

registers. So far, the clock drivers that have been described produced one set of buffered ph1 and ph2 signals for distribution to their macrocell. Unlike the previous design, the delay unit clock driver produces several sets of the buffered clock signals in parallel, and it distributes the sets to the 1023 registers in a branched style. This clock distribution and clock driver layout scheme is shown in Figure 5.6. In the lower portion, the rows and columns in the delay line are graphically represented as a snaked line. The clock driver is at the top. The figure shows two levels in generating the local *ph1* and *ph2* signals. At the first level, the clock driver redundantly inverts and buffers the input *ph1* signal in parallel. This is done for the input *ph2* signal also. Then, at the second level, each inverted *ph1* signal branches to two subsequent inverters. Here, the clock driver recovers the true state of the clock signal, producing several instances of buffered *ph1* signals for the registers. Instances of the *ph2* signal are generated through its own branch. In the clock driver layout, the *ph1* and *ph2* branches are interleaved. In this manner, a pair of *ph1* and *ph2* signals emanating from the clock driver conveniently meets the *ph1* and *ph2* lines that run vertically through a column of the registers.

The clock driver uses a odd and even leafcell to implement the first level of buffering. The odd cell inverts the *ph1* input signal. The even cell inverts the *ph2* signal. At the second level, the clock driver circuit uses one leafcell which produces a pair of *ph1* and *ph2* signals for a column of registers. *Ph1*- and *ph2*- lines internal to the leafcells abut with the corresponding terminal of an adjacent cell to complete the interleaving. The parameter *cols* defines the number of columns the clock driver must span. In this case, *cols* is 32.

If all 1023 registers shared one clock line, the clock path will have a large load capacitance and long path length. This may be sufficient to cause clock skew. In addition, data that propagates between delay lines, especially at extreme ends of the 1023 register chain, accumulates delay introduced by several multiplexers. Because of this, the data arriving at a delay line register may not be synchronized relative to a skewed clock signal that arrives. The purpose of dedicating clock drivers to a column of registers is to prevent clock skew among the delay lines by reducing the load capacitance and length of the clock lines. The buffered *ph1*

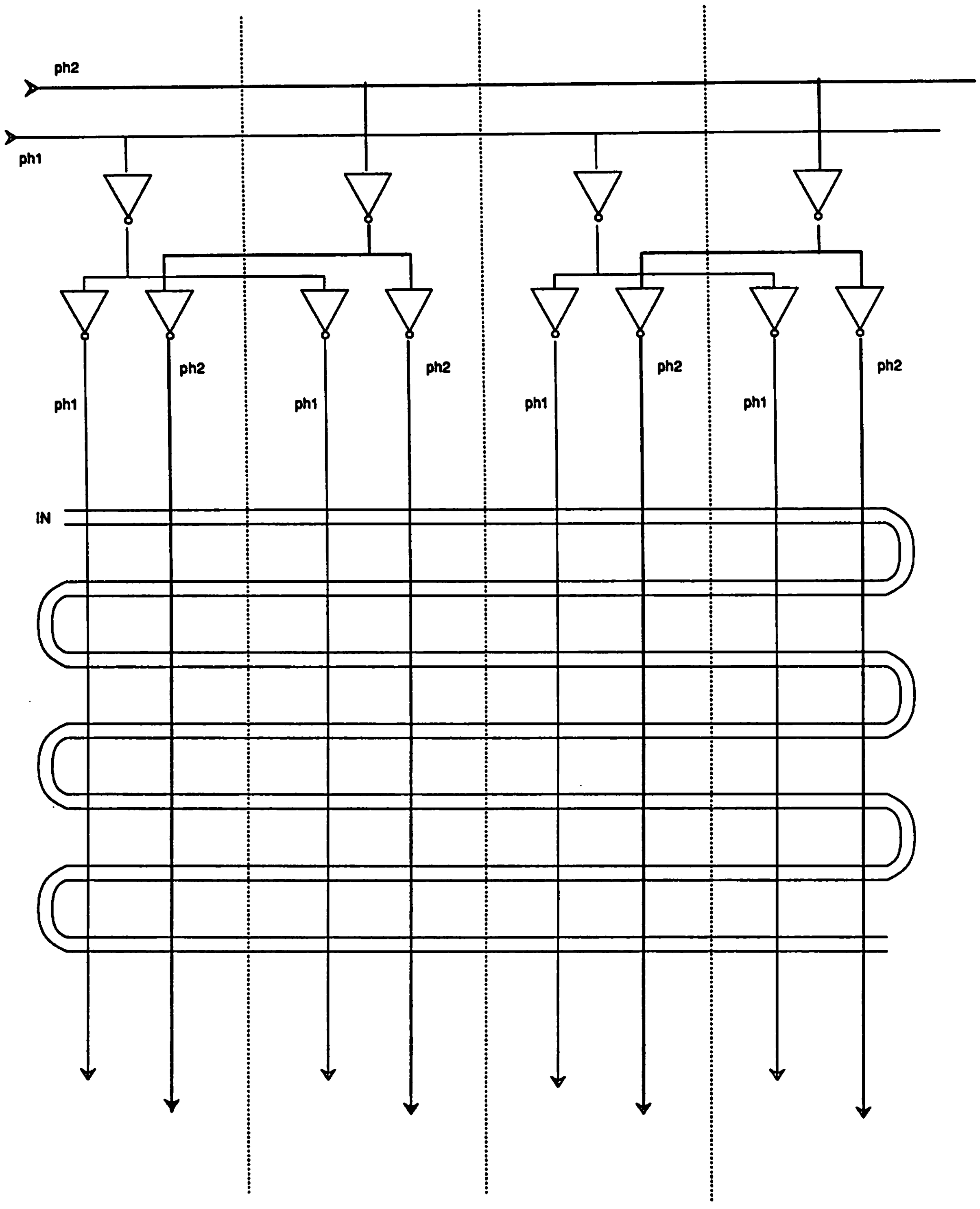


Figure 5.6: Clock Distribution in Delay Unit

and *ph2* signals sent to the registers are in phase with each other, since they are generated in parallel.

Multiplexer Control

The multiplexer control section in each delay unit selects the proper delay lines, as specified by the delay value address, and guides the input data through them. As it was previously discussed, there are ten multiplexers in the delay unit, one assigned to each of the ten delay units. For implementation, the multiplexers are considered as leafcells and are cascaded into one macrocell. The delay unit illustration in Figure 5.3 shows exactly how the multiplexers are connected.

Multiplexer Leafcell

The 2:1 multiplexer circuit of Figure 5.7 uses two cmos pass gates to select one of its two inputs, A or B. Input A corresponds to the "1" terminal and input B corresponds to the "0" terminal. The pass gates are controlled by a select signal, *sel*, which is complementary to the select signal of the second gate. The pass gates output nodes are both connected to a stage of two inverters, which buffers the selected input signal and restores the waveform at the output node C. The select signal is complemented within this leafcell.

Although the circuit is not clocked, it does receive data clocked out from a delay line register on the rising edge of *ph1*, and it sends its output signal to a register input which is sampled on the rising edge of the following *ph2* pulse. If there is a long string of 0s in the delay value address, data passing from a delay line to the next one must propagate through several multiplexers before it is resynchronized by a clocked register. To satisfy timing requirements, data has to propagate through the series of multiplexers within the time of a *ph1* pulse. For the worst case situation, the address is Q=1000000000(N=512), and the delay unit input signal propagates through nine multiplexers before reaching the 512 length delay line input. With the aid of spice and using conservative values for the spice model parameters, the multiplexer circuit is designed to have a 4ns propagation delay from input to output,

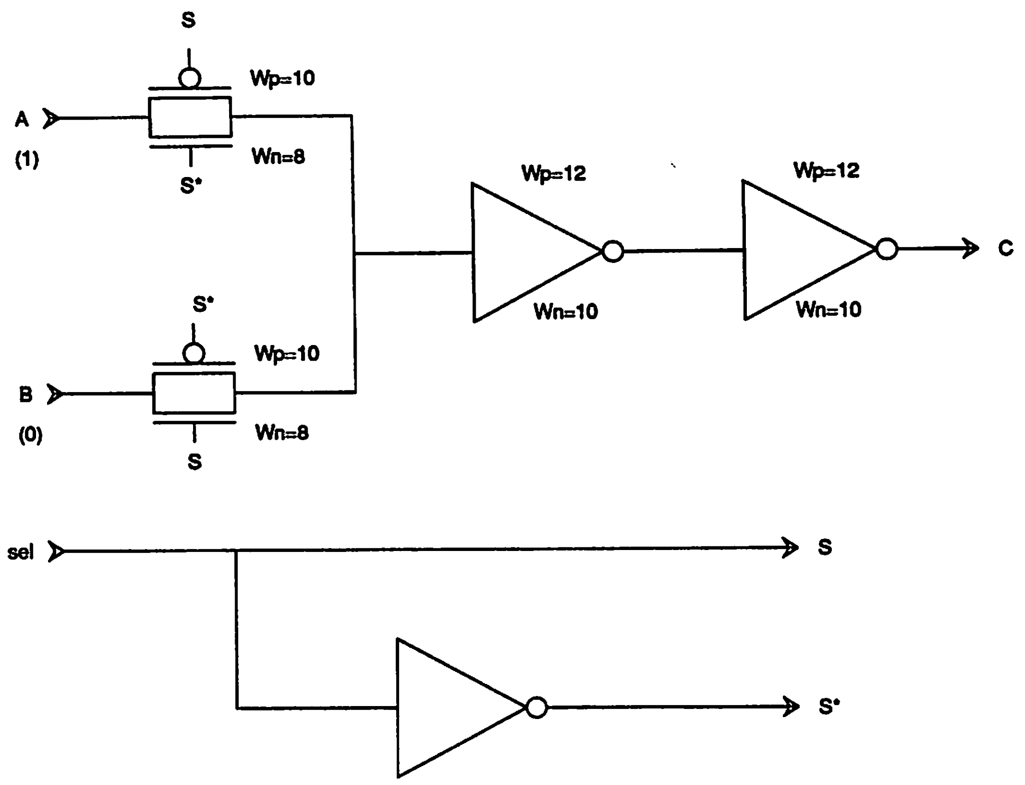


Figure 5.7: Multiplexer Leafcell Circuitry

which explains the large transistor widths for the pass gates. For the worst case, the propagation delay time through nine multiplexers extends to 36ns. In the special case where the delay unit is programmed to insert no delay(N=0), the propagation time through all 10 multiplexers is 40ns.

The multiplexer leafcell layout is shown in Figure 5.8. The *B* and *C* terminals are placed such that the *C* terminal will naturally abut against the *B* terminal in the subsequent multiplexer. The *A*, *B*, and *C* signal lines are extended to the cell boundary for eventual routing to the delay lines.

Macrocell Description

The 10 stage multiplexer macrocell is physically laid out as 10 instances of the multiplexer leafcells stacked on top of each other. The select signal terminals all appear on one side of the macrocell; the input signal terminals which connect to the delay lines are on the opposite side. The delay unit *IN* and *OUT* terminals are

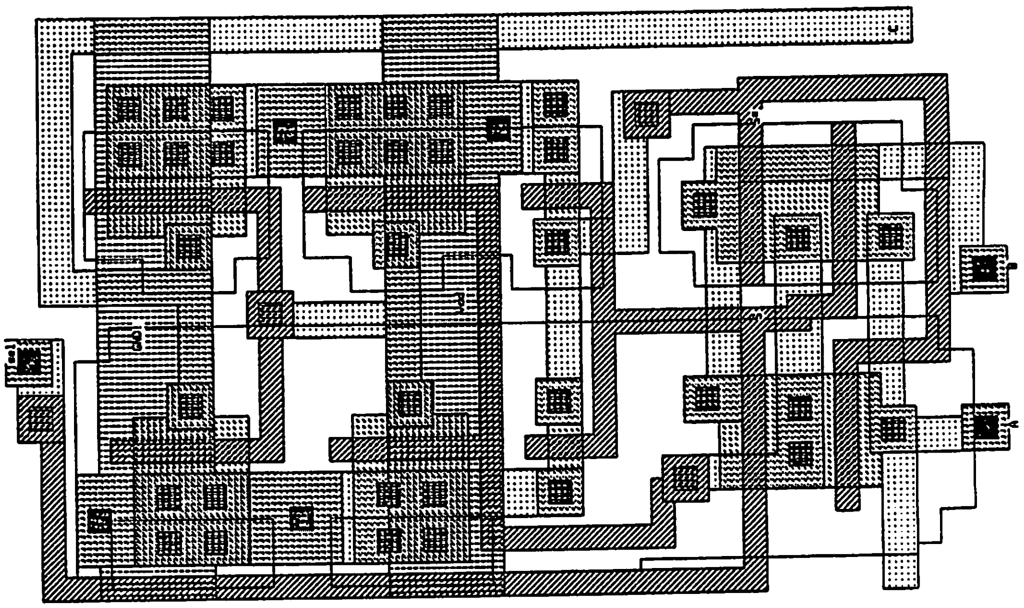


Figure 5.8: Multiplexer Leafcell Layout

located at the bottom and top side of the macrocell, respectively. All the control circuitry in the delay unit resides in this macrocell.

The describing parameter in the TimLager *pdl* description is m. The value of m is the number of multiplexers in the control macrocell, and m is 10 for this Delay Line. For variable Delay Lines realizing maximum delays other than 1023, the parameterized multiplexer design can be used to generate control macrocells with the appropriate number of stages.

Address Latch

The delay value address latch uses the same leafcell and macrocell design as the address latch described in the Tap Switch chapter. The parameter *nbits* is set to 10, so TimLager can generate a 10-bit latch. At the macrocell level, the Delay Line signal reset is the load signal *ld* for the latch.

Clock Generator

The registers in the delay units and the address latch requires a 10MHz two phase non-overlapping clock. A clock generator cell is incorporated into the Delay Line to supply the two phases, *ph1* and *ph2*. This cell was designed by Alex Lee [3] and was known to have been working on silicon at the time of inclusion. As seen in Figure 5.9 the cell consists of a chain of inverters which progressively delays the input signal, clk, a 10MHz symmetric square wave. One NAND circuit creates the *phi2* phase from the *clk* signal and a logically true version, while another NAND gate creates the *phi1* phase from versions which have experienced an odd number of inversions. Both NAND circuits use the inherent propagation delay introduced by the inverters to create the non-overlap property in the two phases.

5.3 Synthesis from Macrocells

Delay Unit

Physically creating the delay unit involves generating the ten delay lines, the multiplexer control, and the clock driver, followed by placing and routing the twelve macrocells together. The delay unit *sdl* description, shown in Table 5.1 is a list of these macrocells, their parameter values, and a net list. From this description, the Design Manager tool creates the parameter descriptions and invokes TimLager to construct each macrocell. With DM, one initial effort from the designer is required to create the twelve macrocells.

Instead of applying Flint, the macrocells were placed and routed manually with magic. Here, the designer felt that manual work could produce a more optimally placed and tightly packed structure than what Flint could produce. Routing the macrocells input or output signals is relatively easy, since a macrocell terminal is connected to at most one other macrocell terminal.

Figure 5.10 shows the physical layout of the delay unit. All the delay lines and the clock driver are abutted together into a rectangular area, sharing contacts to *ph1* and *ph2* terminals at the abutting edges. This abutment style is

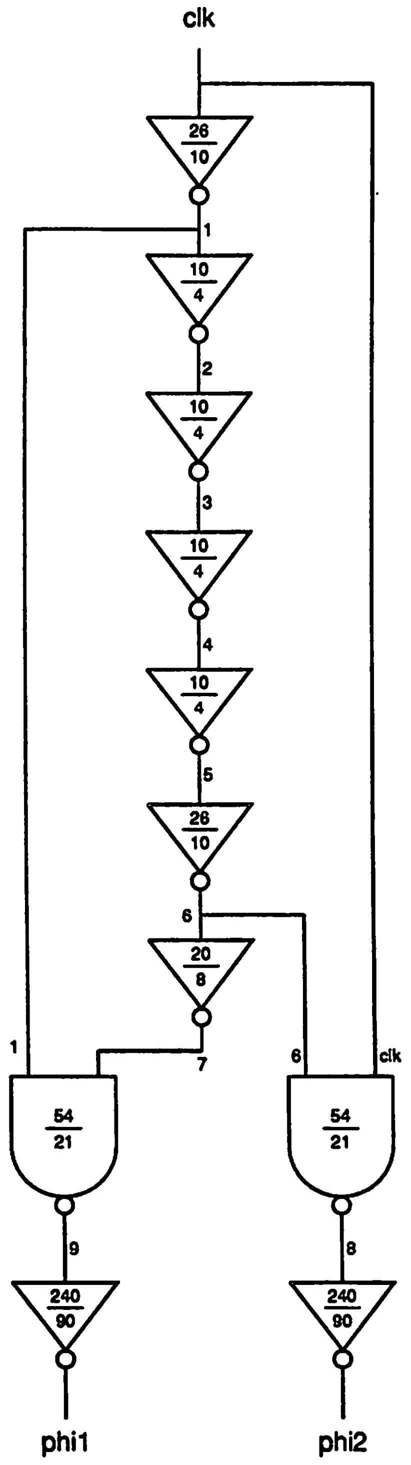
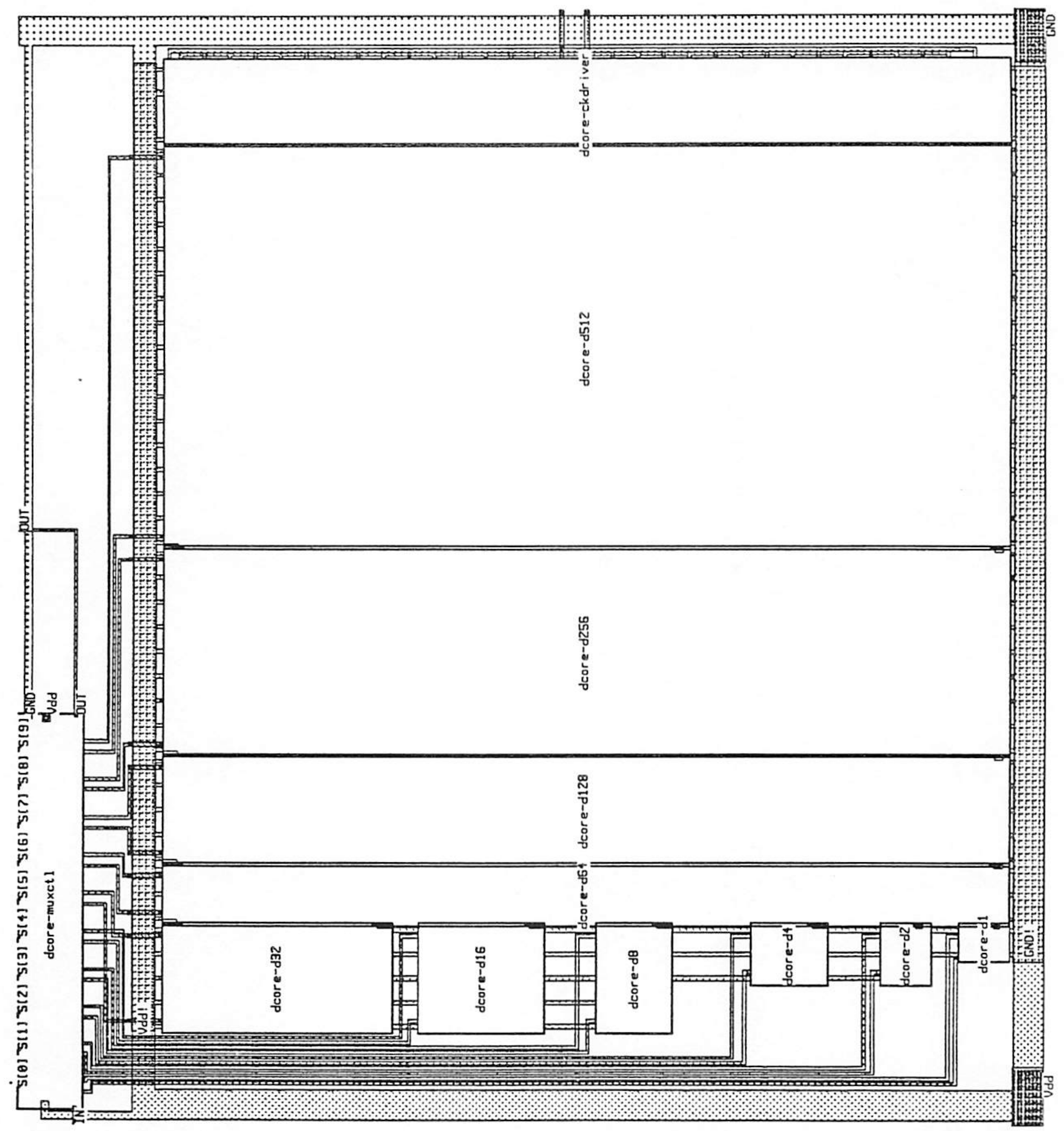


Figure 5.9: Two Phase Non-Overlapping Clock Generator

Figure 5.10: Delay Unit Layout



not permitted by Flint because of its nature as a channel router. If the macrocells were expanded, the clock lines would run vertically, while the ground and power lines run horizontally contacting the ground and power busses at the left and right edges of the larger delay line macrocells. The multiplexer control is placed at the lower left corner. Overall, the interconnect lines occupy a small space in the entire delay unit area. The delay unit layout has dimensions 1498λx1360 λ, almost square in shape.

A special TimLager *.c* routine instructs TimLager to place one instance of the delay unit and to label its terminals, formally transforming the delay unit into a macrocell from the perspective of the LagerIII tools. More significantly, an *hdl* description is also produced, containing the delay unit dimensions and terminal location information. This is the real motive behind this step in layout design, since manually extracting that information is time consuming and tedious.

From this point on in the design process, CAD tools assemble the layout for silicon realization.

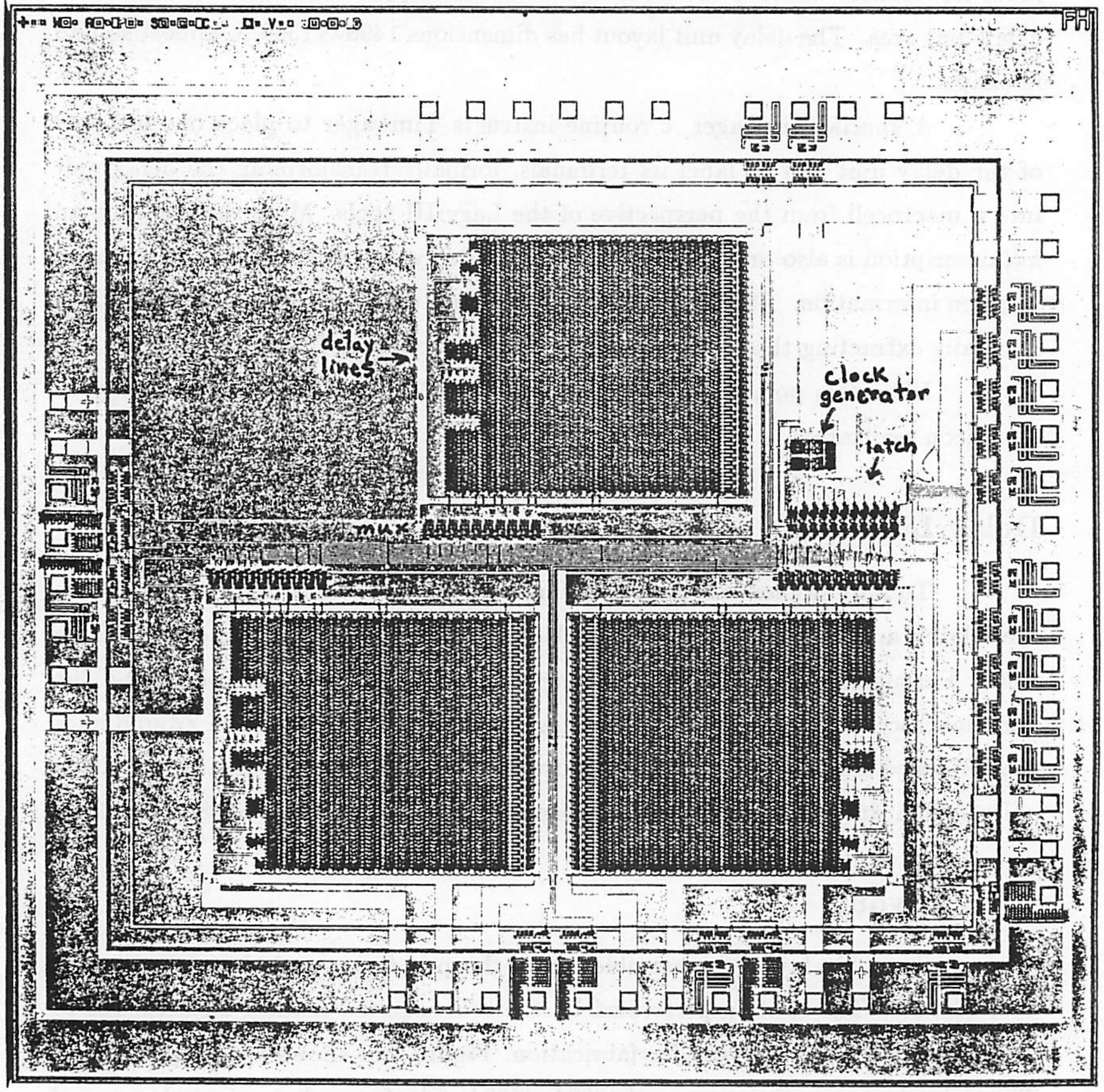
Delay Line Core

To fully implement the Delay Line architecture, three delay units, an address latch and a clock generator are placed and routed, forming the Delay Line Core. Through Design Manager, TimLager creates delay unit macrocells and the latch macrocell. Following, Flint generates the core layout. Power and ground lines were widened manually to carry more current, since all 1023 registers in three delay units switch simultaneously. The dimensions of the core circuitry is 2878λx3055λ.

Chip Layout

The chip layout is assembled from the pad frame and the core circuitry by Padroute. The routing performed by the tool required manual adjustments to produce an acceptable layout for fabrication. Figure 5.11 shows the resulting chip. The Delay Line occupies an area of 4062λx4462λ. The frame design includes several ground and power pads. This reduces the effective inductance introduced by metal

Figure 5.11: Photograph of Variable Register Delay Line



North Side Pads from Left to Right		
name	type	description
Vdd	Vdd	
Vdd	Vdd	
Vdd	Vdd	
GND	GND	
GND	GND	
GND	GND	
reset	input	resets address latch
clock	input	single square wave clock
GND	GND	
Vdd	Vdd	

Table 5.2: Pads on North Side of Pad Frame

bonding wires and metal traces in the packaging which lead the external power and ground supply to the IC power and ground pads. Tables 5.2, 5.3, 5.4 and 5.5 describe the pads appearing on each side of the pad frame.

5.4 Simulations

After extracting the layout, logic simulations with esim were performed throughout the layout hierarchy to confirm that the various cells were logically functional. Circuit simulation was carried out during the circuit design phase, but not after the cell layout, relying on the switch level logic simulation to detect any bugs in the layout. Leafcell logic simulations were simple and straight forward, as was simulating the delay line and multiplexer control macrocells.

Simulating the delay unit was only slightly more complex. The input test patterns for the delay lines and the multiplexer control were applied, and the output vectors from the output nodes were observed and then verified to be a delayed version of the input test vectors.

Instead of performing logic simulation at the next level in the layout hierarchy, the subsequent simulation occurred at the top level. Since the delay unit is the heart and majority of the Delay Line design, and since delay unit simulations

East Side Pads from Top to Bottom		
name	type	description
GND	GND	
Vdd	Vdd	
q9	input	q_9 bit of delay value address
q8	input	q_8 bit of delay value address
q7	input	q_7 bit of delay value address
q6	input	q_6 bit of delay value address
q5	input	q_5 bit of delay value address
substrate	dummy	chip substrate contact
q4	input	q_4 bit of delay value address
q3	input	q_3 bit of delay value address
q2	input	q_2 bit of delay value address
q1	input	q_1 bit of delay value address
q0	input	q_0 bit of delay value address
GND	GND	
Vdd	Vdd	
out.c	output	output from carrier delay unit

Table 5.3: Pads on East Side of Pad Frame

South Side Pads from left to right		
name	type	description
Vdd	Vdd	
GND	GND	
GND	GND	
Phi2	output	ph2 phase from clock generator
Phi1	output	ph1 phase from clock generator
GND	GND	
Vdd	Vdd	
TestIn	input	for testing
TestOut	output	for testing
GND	GND	
Vdd	Vdd	
in.c	input	input to carrier delay unit

Table 5.4: Pads on South Side of Pad Frame

West Side Pads from Top to Bottom		
name	type	description
Vdd	Vdd	
GND	GND	
in.v	input	input to code violation delay unit
out.d	output	output from data delay unit
out.v	output	output from code violation delay unit
in.d	input	input to data delay unit
GND	GND	
Vdd	Vdd	

Table 5.5: Pads on West Side of Pad Frame

and latch simulations were favorable, the writer felt that another simulation to confirm logic validity could be postponed until the final layout was generated. After extracting the layout, logic simulation with esim did not converge. Esim delivered "don't care" states from all the output nodes of the delay units. Regressing step by step in the layout hierarchy, the bug was discovered at the core layout level. The problem was not due to faulty circuit or layout design, but rather due to oversight on the designer's part. In simulating the delay units and the address latch, the *ph1* and *ph2* clock patterns were explicitly provided as *ph1*=0010 and *ph2*=1000 for one clock cycle. The non-overlap property is observed in the second and fourth position. However, at the core level, the clock generator provides the two phases to the delay unit and address latch. As discussed, the clock generator relies on the inherent propagation delay of a transistor to produce the non- overlap. But esim models transistors as devices which respond instantaneously to its stimulating input, causing the extracted clock generator to deliver *ph1* as 0011 and *ph2* as 1100 during simulation. Driven by two phases whose non-overlap is not well defined, the simulations on the extracted delay line and latch fail to converge at the chip level. Logic simulations do converge to the predicted results by forcing the *ph1* and *ph2* signals to take on the proper pattern in the simulation.

The lesson learned from this simulation exercise is two-fold. First, it demonstrates how the hierarchical layout style aids debugging a design. By simulating from the low level leafcells up to the top level, this approach facilitates locating the cause of a problem as compared to locating the bug in a flattened layout which can be an exhaustive effort. Second, it serves as a warning that CAD tools, as powerful as they may be, should not be applied with blind faith which can lead to misuse and misinterpretation of results. In this example, familiarity with the limitations of esim could have prevented the oversight on the designer's part.

The extracted Delay Line layout was simulated for delays in the range of N=0 to N=50 clock cycles of delay. The fabricated chip went through more exhaustive testing as will be described in the next section.

5.5 Testing

The Delay Line circuit was fabricated in 3μm pwell scmos technology and the finished silicon was tested for functionality and performance in October of 1987. This section describes the test strategy, test patterns and results.

Delay Line operation obeys the timing diagram shown in Figure 5.12. It follows the clocking convention set for the Channel Emulator. The Delay Line timing requirements are identical to the Tap timing requirements, except that the delay line generates its two phase clock from a square wave rather than using external *ph1* and *ph2* sources.

Testing the Silicon

To aid in troubleshooting the logic, circuit and layout design, subcells within the Delay Line are duplicated in a separate chip created for testing purposes only. The philosophy is that testing the smaller cells separately simplifies locating the source and determining the cause of a design error as compared to testing at the higher level. After all, high level cells will not work if their subcells do not work. This is similar to the logic simulation strategy. Register and clock buffer leafcells as well as a clock generator, address latch and multiplexer control macrocells reside in the test chip. These circuits have their input and output nodes connected to Input or Output pads for observation. The *ph1* and *ph2* clock phases needed by

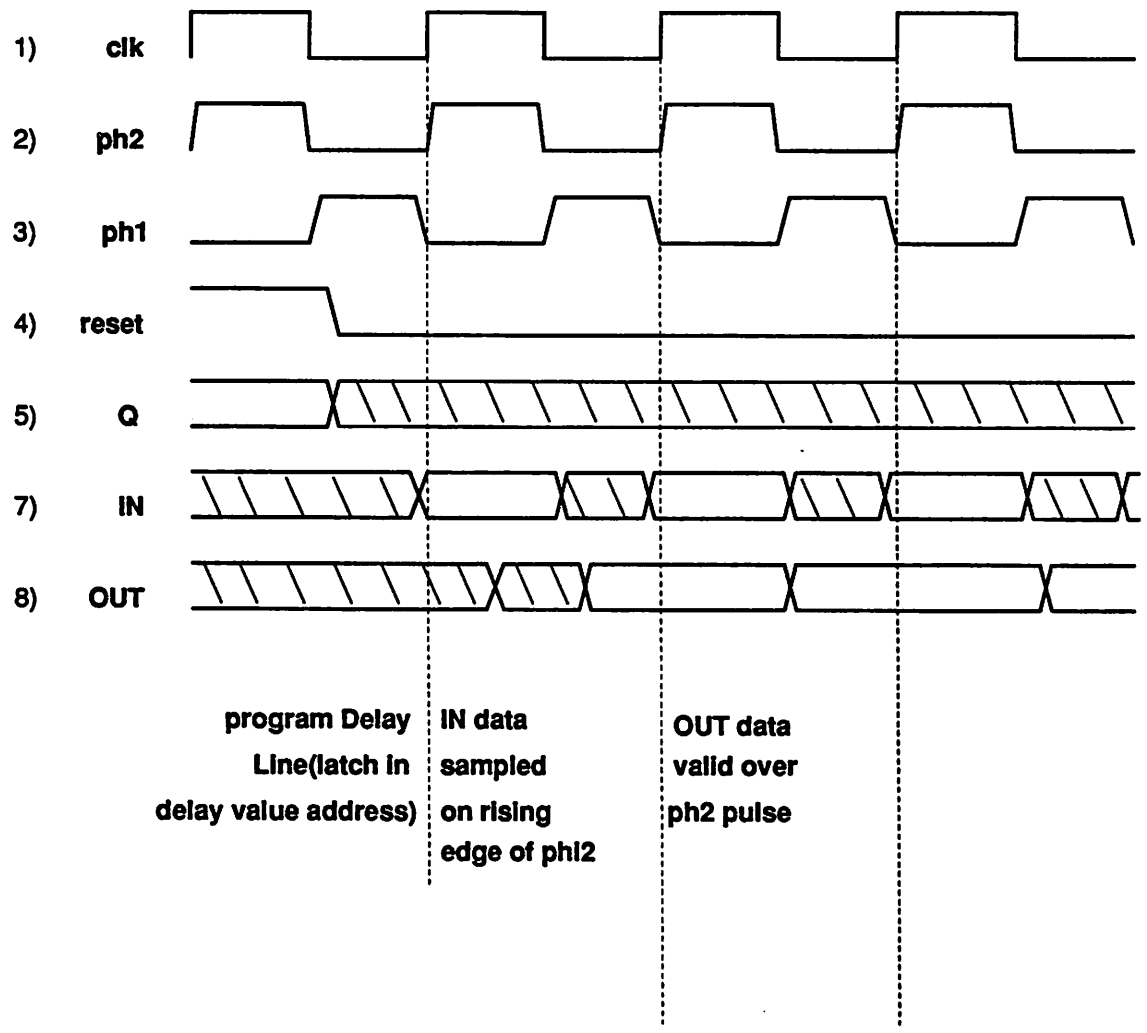

1) clk
2) ph2
3) ph1
4) reset
5) Q
7) IN
8) OUT
program Delay Line(latch in delay value address)
IN data sampled on rising edge of phi2
OUT data valid over ph2 pulse

Figure 5.12: Timing of Delay Line Signals

the cells are provided by an external source. This ensures that the circuits are tested independently of the clock generator circuit.

For the Delay Line circuit in the finished silicon, testing is essentially simulating the real working environment the chip operates in. The procedure involves applying input test patterns and observing the output nodes from I/O pads. Internal nodes are not probed, since this is accounted for in the separate test circuit. Hopefully and fortunately, the observed outputs are as expected, and the silicon testing determined that the circuit was fully functional.

The Delay Line has a straight-forward purpose: to delay the 3-bit input signal by N clock cycles. Testing is also straight- forward. An 10-bit address is sent to the address input pins, specifying the amount of delay. Then, a pattern is applied to the *data*, *carrier* and *code violation* input pins. For each of the 3 bits in the signal, the input pattern is a short stream of 0s and then a burst of 1s followed by a long trail of 0s. The burst of 1s in the long stream of 0s serves as a time "marker." Comparing the observed output to the applied input, the input and output signals are identical in sequence but the first occurring 1's are spaced N clock cycles apart in time. This test is repeated for various values of N. As in the Tap chip, testing begins at low clock rates to confirm functionality, and continues to 10MHz to confirm that the circuit satisfies the speed specification.

Performance Results

Besides verifying the design functionality, measurements were collected to assess the chip performance. Performance results depend on the particular run the chip was fabricated from. So, the test results to be discussed are influenced by the run properties as well as the design.

An input pad terminal is tied to an output pad terminal. With this setup, the input and output pad propagation delay combined was measured at 20ns. This figure is useful in determining the macrocell propagation delays.

The *ph2* and *ph1* output nodes of the clock generator are connected to an output pad. There is a 6ns delay from the *clk* node to the *ph2* node. The two phases have an non-overlap time of 5ns and a 40ns pulse width. This was measured from the output pads which buffer the internal *ph1* and *ph2* signals that is actually distributed to the circuitry.

Programming N=0 into the circuit, a signal propagating through all 10 stages in the multiplexer control exhibited a 35ns delay. This gives 3.5 ns propagation delay per multiplexer cell. At operating frequencies above 10MHz the 35ns delay becomes comparable to the pulse width of a phase. This propagation delay will eventually limit the speed at which the circuit can operate while conforming to timing requirements at the same time.

Out of all the dies in the batch returned, half of the ICs worked properly at 10MHz. Among the others, the clock generators always tested properly, and one or two delay units in the die worked while the other demonstrated a "stuck at 0" or "1" condition. The cause is suspected to be associated to the run, or it could be in the design. Overall, the results were quite satisfying for a first pass design. It is confident to conclude from direct testing that the Delay Line worked at speeds up to 15MHz for any delay value. The limiting frequency is at least 20MHz. Tests were not carried out beyond this rate due to equipment limitations.

Chapter 6

Crossbar Switch

The Crossbar Switch has a set of 16 input channels and a set of 16 output channels. It routes 10MHz data received at each input channel to selected output channels. The single constraint is that each output can be connected to at most one input. But, data from one source can be transported to more than one destination. Essentially, the crossbar switch is functionally a set of sixteen 16:1 multiplexers that have their inputs tied.

Figure 6.1 shows only one of many ways to interconnect or route data from the input channels to the output channels. The interconnection pattern is referred to as the crossbar switch configuration. So, the interconnection arrangement, shown in the illustration, is just a particular crossbar configuration. The crossbar switch is programmable in the sense that its configuration can be changed by the user; in other words, the user can reconfigure the current point-to-point requirements of the application.

The crossbar switch implements the Delay-Input Routing block of the channel emulator. The 16x16 switch can carry signals from eight tap switch outputs to the Delay Lines. The crossbar design is parameterized so that it can be expanded into an nxn crossbar switch, where n is an even value. The ultimate goal is a crossbar switch that accommodates 64 input and 64 output channels, fitting into a 32 node channel emulator system.

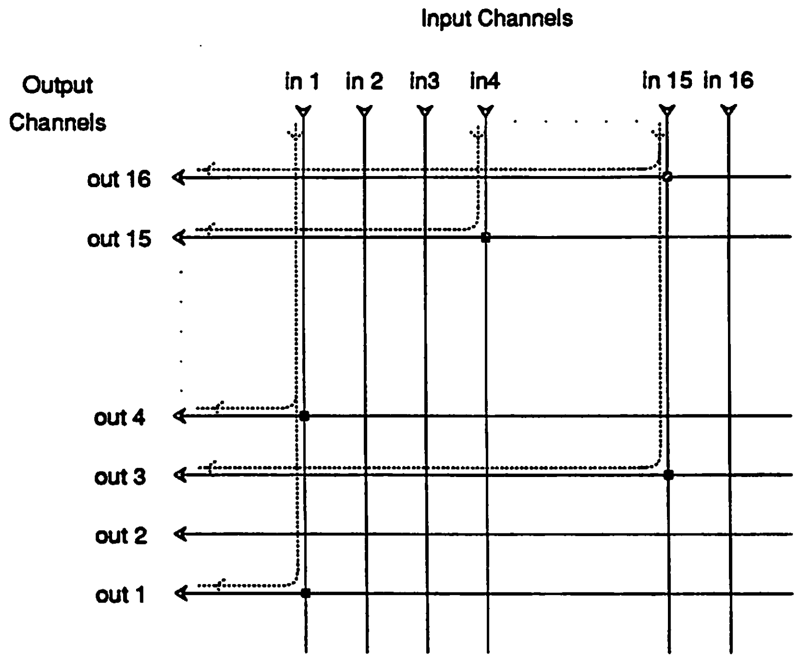


Figure 6.1: Conceptual View of the Crossbar Switch

6.1 Architecture

The crossbar switch performs a routing function and its configuration must be programmable. Accordingly, the chip architecture is is divided into blocks that provide these functions separately, as illustrated in Figure 6.2. The physical arrangement of the blocks is also the floor plan of the chip. The architecture consists of two main sections, the Routing section which transmits data from multiple sources to their assigned destinations and stores the input-to-output configuration, and the Programming section which programs the input-to-output configuration.

A more detailed illustration of the architecture is shown in Figure 6.3, which is actually an expansion of the block diagram in Figure 6.2. Figure 6.3 shows the flow of information within and among each block. It also illustrates the design approach for the crossbar switch which takes advantage of what VLSI technology can best produce: a regular structure of identical cells, with a minimum of communication lines required between them. This architecture design allows

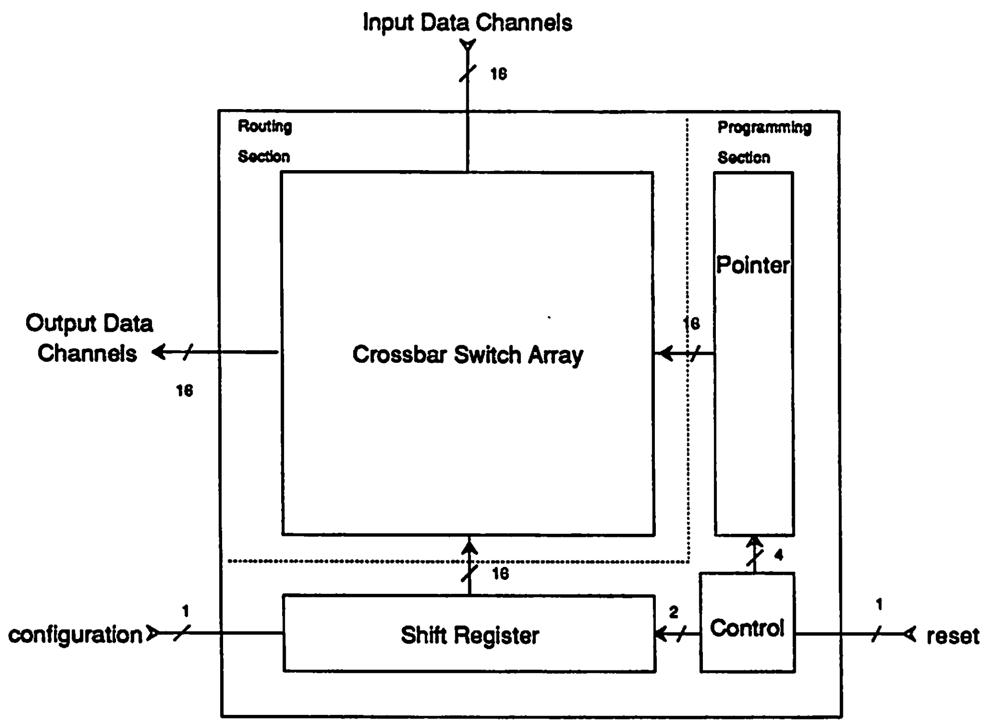


Figure 6.2: Crossbar Switch Architecture

for modular increases in routing capacity and is well suited toward parameterized layout generation using TimLager. The following discussion explains further details of the architecture and how routing and programming is implemented.

As Figures 6.2 and 6.3 indicate, the crossbar switch is composed of two main sections. First, the routing section interconnects the input and output data channels, shown as *Din* and *Dout*. This is implemented by the crossbar switch Array. The Array consists of interconnection cells; each cell has a memory element and a switch element. The switch is in either the ON state(connect=1) or the OFF state(no connect=0). This state is stored in the memory element by a one-bit RAM cell. The array of 1's and 0's stored in the RAM cells represents the crossbar configuration. A 1 bit held in the (i,j) interconnection cell causes the switch to connect the ith output channel to the jth input channel, and a 0 bit in the (i,j) cell disconnects them. For example, if the interconnection cells in the bottom row contained all 0s except a 1 in the last cell, then the first output channel selects data from the 16th input channel. By using the RAM circuit, the state of the switch

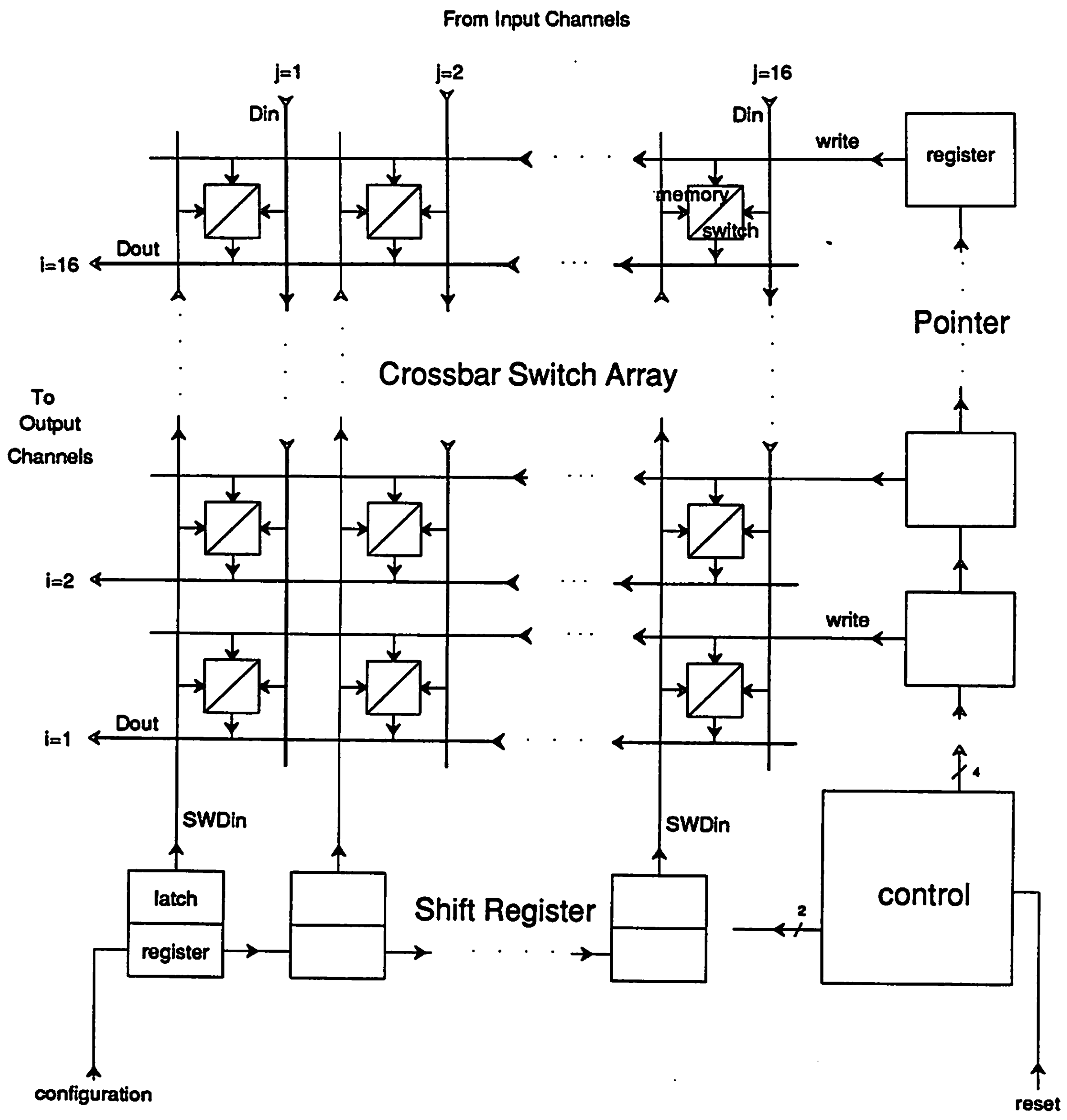


Figure 6.3: Crossbar Switch Architecture in Detail

can be changed and, hence, the crossbar configuration becomes programmable. The crossbar switch Array contains 16x16=256 interconnection cells.

The second section programs the the crossbar configuration into the Array. To carry out its function, this section uses a Shift Register of length 16, a 16 stage Pointer and a Programming Control unit. The Shift Register serially accepts the crossbar configuration bits, representing the switch states, and then transfers them in parallel to the RAM. The bit lines are shown as *SWDin* in Figure 6.3. The Shift Register serially brings in the configuration bits for the switches in row 1 first, followed by the bits for row 2, then row 3 and so on. When the *ith/* row of configuration bits has been shifted in, latches within the Shift Register capture the row of states and then write the signals onto the RAM bit lines . The row of RAM cells is addressed by the Pointer, and all of its cells receive their bit signals in parallel. While the latches write the states into the ith row of memory elements, the register serially shifts in the sixteen (i+1)*th* row of configuration bits over a time of 16 clock cycles.

The Pointer simultaneously addresses all the memory elements in an row. It delivers a *write*(=1) signal to row 1, then to row 2, until row 16. Thereby, the crossbar configuration is programmed into the RAM structure in a row by row fashion. The Pointer is basically a 16 stage shift register that shifts every 16th clock cycle. The first register passes a 1 bit to the next stage followed by a stream of 0s. This 1 bit is the signal that "points" to the rows sequentially. Since it takes 16 clock cycles to bring in a row of states into the Register, the Register and the Pointer writes the previous row of states into memory in that amount of time.

The Control block uses a 4 bit counter and logic gates to generate the proper control signals required to program the configuration into the Array. To initiate programming, the *reset* signal initializes the control block which, in turn, provides the initial 1 bit to the Pointer.

Altogether, the crossbar switch needs one macrocell to implement routing and three macrocells to perform programming. In this design, the input signals on the individual channels may be asynchronous among each other, because the switch in the crossbar Array macrocell is not clocked. The macrocells for programming, however, use clocked circuitry and require a two phase non-overlapping clock signal.

6.2 Macrocell Design and Description

Crossbar Switch Array

This macrocell contains the interconnection cells tiled into an array. These cells do not communicate with each other, but they do share common data and control lines.

Leafcell Design

The interconnection leafcell integrates the memory circuit and the switch into one structure. Figure 6.4 shows the leafcell circuit. For its application, the writing speed of the RAM is not a critical issue, but storing the configuration bit for an indefinite amount of time is a design requirement. Taking these into consideration, the memory cell uses a five transistor static RAM design, rather than the typical six transistor circuit. The design has two cross coupled inverters and one write transistor, and brings data into the memory from one bit line. This choice consumes less layout space and simplifies the routing task compared to using a bit and bit* line. The leafcell application also does not require reading out the stored data. This introduces more freedom in sizing the transistors, and makes it possible to exclude any circuitry for a read operation. The impact of this on testing the fabricated circuit is discussed in the testing section. In designing the memory circuit, the emphasis is in determining the ratio of transistor N1 channel width to N2 channel width. To write from 1 to 0, Transistor N1 can discharges a high level at node X quicker with a larger W. In writing from 0 to a 1, transistor N1 charges node X to an intermediate level because of body effect and the the tendency of the N2 device to sink charge. With the aid of Spice simulations, the circuit writes a 0 in 70ns and a 1 in 100ns using the transistor sizes of Figure 6.4. This falls well within the time limit imposed by the programming scheme.

The switch in the interconnection cell is a cmos pass gate. Minimum

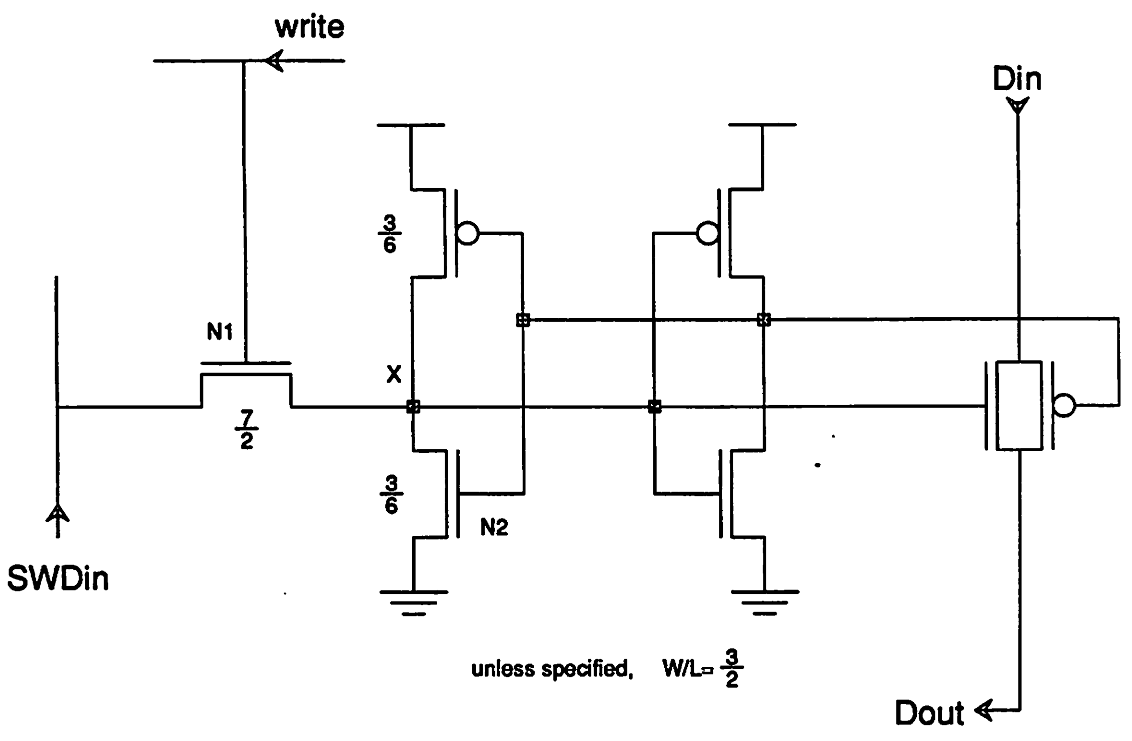


Figure 6.4: Interconnection Leafcell Circuitry

transistor sizes minimize the parasitic drain and source capacitances on the data input and data output lines. A high level stored at node X in the memory circuit turns the pass transistors on, forming an electrical connection between the data input and the data output line. The pass gate disconnects the electrical path if node X is at a low state.

The interconnect leafcell layout is shown in Figure 6.5. The *write*, *Dout* data output and ground lines run horizontally in metal2, while the lines carrying the *SWDin* switch configuration and power run vertically in metal1. Ground and power contacts are shared with the adjacent cell in the macrocell, as are the *Din* data in and *write* signal contacts. This approach compacts the layout and reduces parasitic drain capacitances on the signal lines.

There is also another leafcell which contains a weak pmos pullup. This device is attached to the end of each *Dout* output line. In case an output is not connected to any input line, this device maintains a definite high signal at the output, preventing the output node from floating.

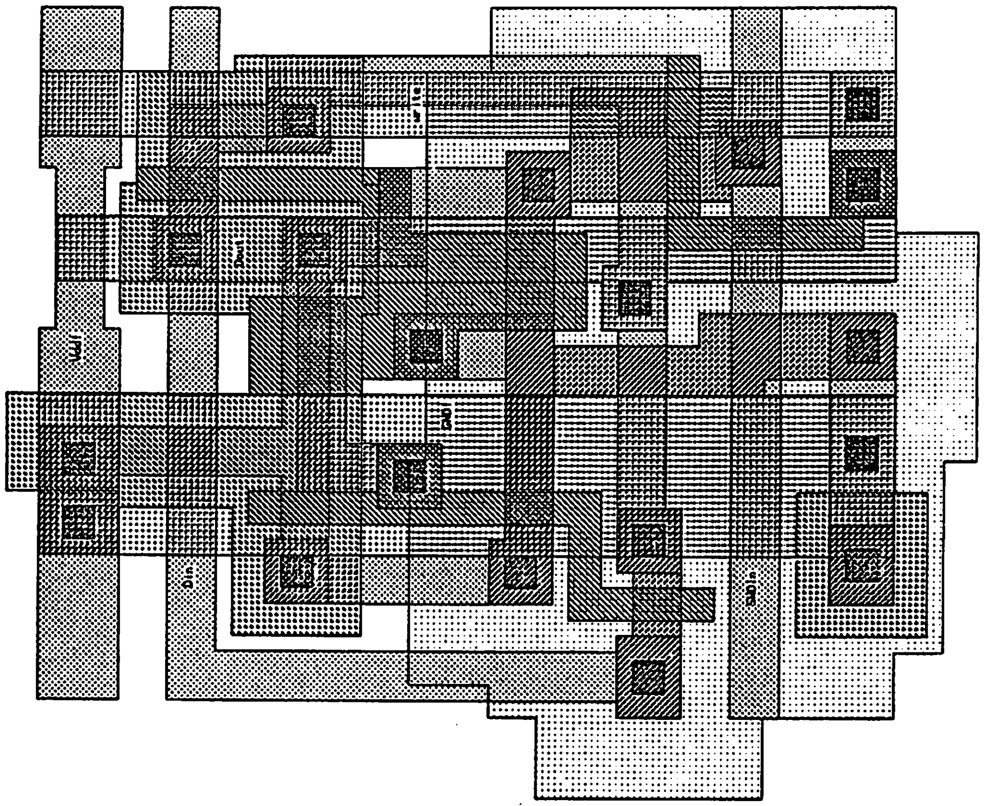


Figure 6.5: Interconnection Leafcell Layout

Macrocell Description

The Array macrocell is laid out as instances of interconnect leafcells arranged in a 16x16 matrix structure. Also in the macrocell, a column of pullup leafcells abuts against the right edge of the matrix. To form the the interconnect array from one leafcell layout, leafcells in even numbered rows or columns are mirrored about the X or Y axis, respectively, So, as they are placed by TimLager, the leafcells in the first row are alternately mirrored about the Y axis. In the second row, all the leafcells are oriented upside down, or mirrored about the X axis, and alternately mirrored about the Y axis. This placement pattern is repeated for the remaining rows. Cells share contacts and their I/O lines meet properly with those in adjacent cells. At the macrocell edges, the data input terminals lie on the top edge and the data output terminals appear at the left edge; the write signal from the Pointer enters at the right edge, and the switch state terminals are located on the bottom edge. This placement evenly distributes the macrocell *Din* and *Dout* signal lines around its sides.

The describing parameter used by TimLager is *nodes*. To generate a crossbar Array for an n node channel emulator, the parameter is assigned a value of *nodes*=$2n$. For this case, nodes is specified as 16.

Shift Register

The configuration bits enter the Crossbar Switch serially. To program these into the crossbar Array by rows, the Shift Register macrocell converts the bit stream into parallel format and drives all the bit lines. The register consists of 16 stages containing the same circuit. A two phase non-overlap clock signal controls the shifting.

Leafcell Design

Figure 6.6 shows the circuit for a single stage. The basic register at the bottom of the figure stores input data for one clock cycle and shifts it to the input of the next stage. Meanwhile, the next configuration bit is brought into the register. The latch at the top is the same circuit seen in the address latches discussed in previous chapters. Its input taps the data stored in the register, accepting it when the ld signal is high. Supplied by the Programming Control cell, the *ld* control signal pulses high on every 16th clock cycle and covers the *ph2* pulse. After one clock cycle, the particular configuration bit is latched in. The latch output is connected to a bit line in the Array and the top inverter writes the data into the addressed RAM cell.

The leafcell's layout, in Figure 6.7 has clock, *ld*, and *ld-* lines running horizontally in metal2, and the power and ground lines running vertically in metal1. The register *srin* input and *srout output* terminals are placed so that the output terminal of one cell naturally joins with the input terminal of the adjacent cell. Special effort was made to ensure that the width of this leafcells matches the interconnect cell width. For the higher level layout, this allows systematic routing between the Shift Register and Array macrocells and avoids awkward macrocell sizes relative to one another.

is implemented with a 16 stage shift register and extra write operation logic. A 1 bit moves through the registers to select the assigned row of RAM cells for writing. All other rows remain unselected. At the beginning of the programming process, the Control cell supplies the 1 bit to the first register at a terminal called *Pin*.

Leafcell Design

A Pointer stage contains the circuit shown in Figure 6.8. The familiar latch circuit appears in this cell. With the same *ld* signal used in the Shift Register, the circuit shifts in a new bit when the *ld* signal is high. This occurs once every 16th clock cycle. The output of the latch, *Pout*, is connected to the latch input, *Pin*, in the next stage and to the input of an AND gate. The second input to the AND gate is the *wrctl* signal. This is the actual write enabling signal that specifies the exact write time period, and comes from the Control cell. A *wrctl* signal pulse falls between the *ld* pulses without overlapping them. The inverter output of the gate drives the *write* line in the Crossbar Array. The AND logic is implemented in its own leafcell, so that two leafcells realize one Pointer stage.

The layout for the latch and the AND gate have the same height. As in the Shift Register leafcell, this height matches the vertical pitch of the interconnection cell. Figure 6.9 shows the layout for a stage. Clock and control lines run vertically, and the *write* terminal appears at the left edge. The clock signal drivers for the Pointer are placed in a separate leafcell.

Macrocell Design

The Pointer macrocell is laid out as 16 stages stacked on top of one another. The *ld*, *wrctl* and *Pin* control signals enter from terminals at the bottom of the macrocell. The describing parameter for TimLager is again *nodes*. For this case, nodes is assigned the value 16 since there are 16 stages.

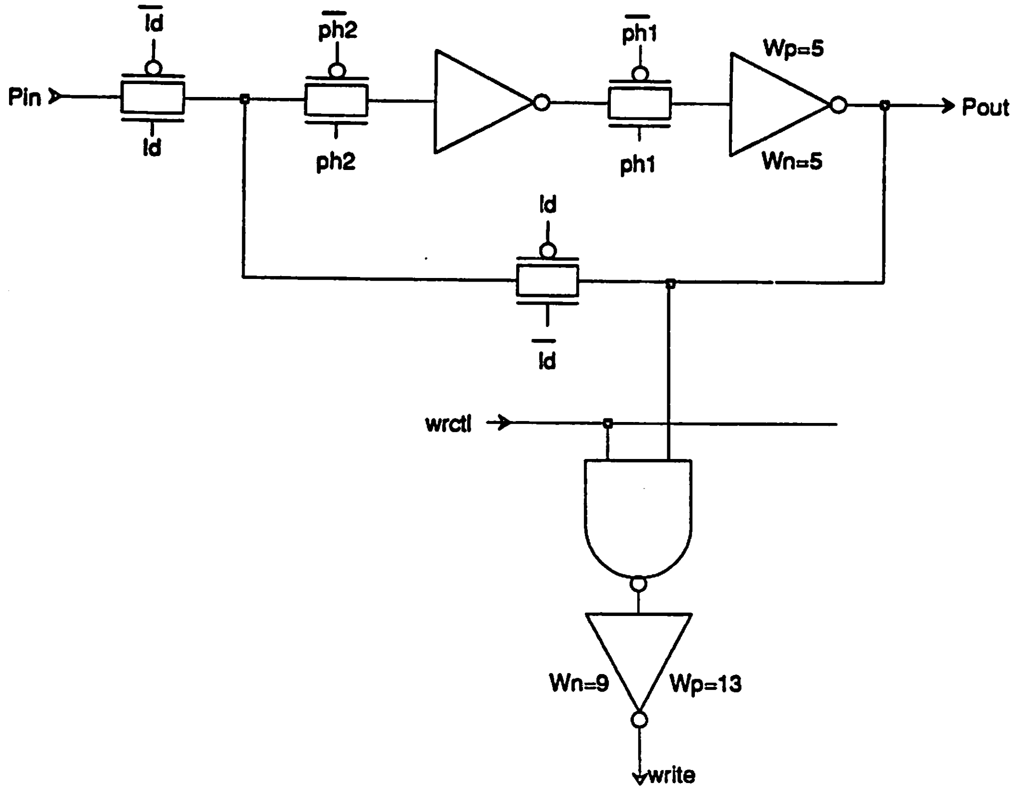


Figure 6.8: Pointer Leafcell Circuitry

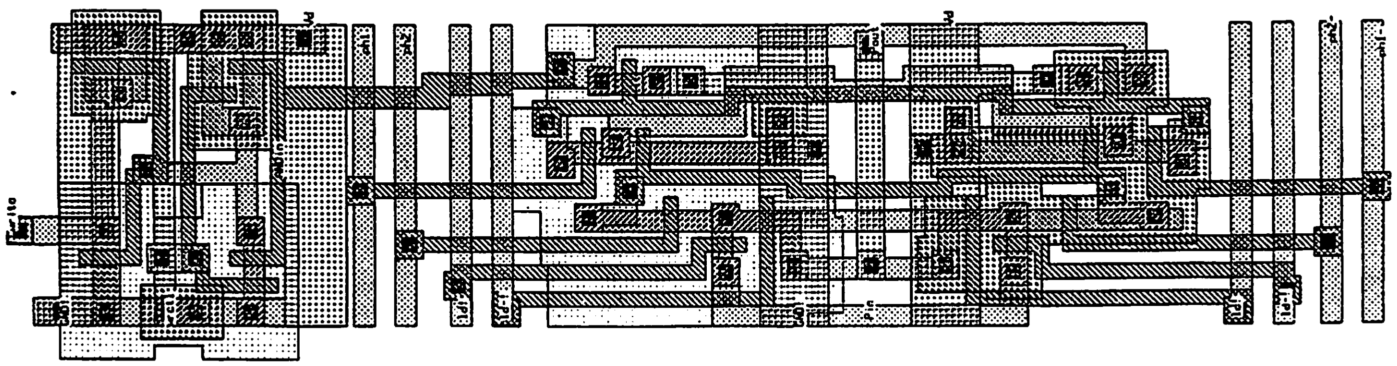


Figure 6.9: Pointer Leafcell Layout

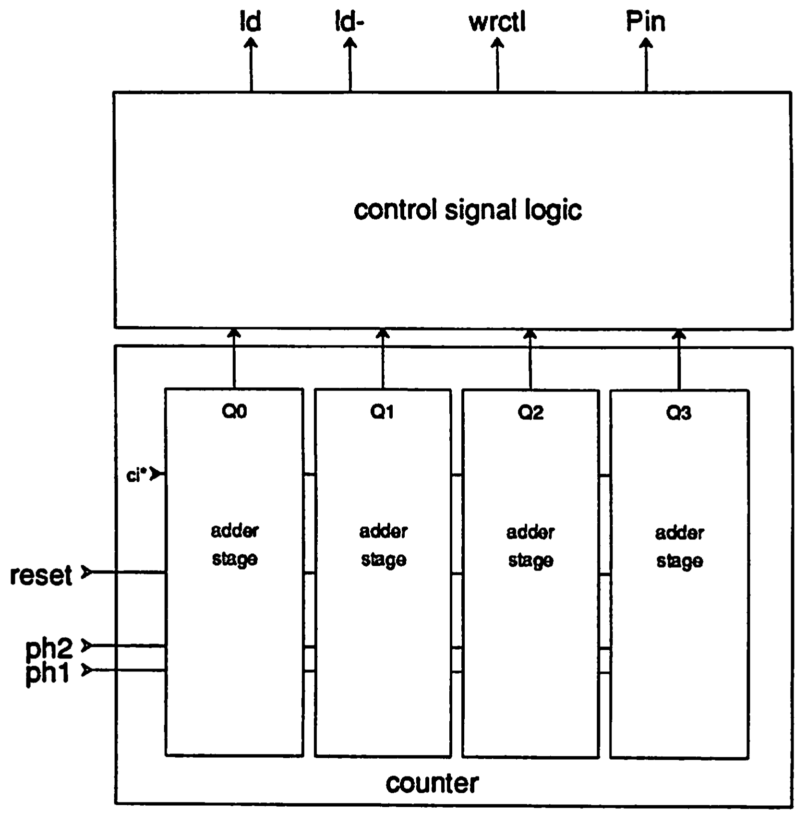


Figure 6.10: Floorplan for Programming Control Macrocell

Programming Control

The Programming Control section produces the control signals that the Shift Register and Pointer needs to program the crossbar configuration into the Array. It derives the signals, *ld*, *ld-*, *wrctl*, and *Pin*, from a two phase non-overlapping clock and the *reset* pulse. These functions are implemented with a 4-bit counter and combinational as well as sequential logic. The floor plan for this macrocell is shown in Figure 6.10.

When the counter is initiated by the *reset* pulse, it counts from 0 to 15, cyclically. The counter design uses a 4-bit carry ripple adder that increments its sum every clock cycle. This sum is the 4-bit binary count, $Q_3Q_2Q_1Q_0$, appearing at the outputs of the counter. The counter is based on a design made by Alex Lee [3] and was used to save development time. The logic and layout design was modified to fit the particular application. Four slices make up the full counter. Each stage is a half adder followed by a clocked output register whose output is the sum. The sum signal is fed back to the addend of the adder. To increment by one, the first half adder always carries in a one. This signal ripples through all four adders for the new sum. There is an odd and even slice to minimize the logic that produces the carry out signal. The only difference between them is a NOR gate in the even slice to produce the carry out while the odd slice uses a NAND gate. Each bit of the counter output is delivered from the registers to logic that generates the control signals.

The generating logic is shown in Figure 6.11, and the timing diagram for configuration programming is in Figure 6.12. Both the Shift Register and the Pointer use *ld* and *ld-* to control their latching action. These two signals are derived from the four counter outputs resulting in a pulse every 16th clock cycle. The four input AND gate is a conventional cmos AND circuit using four pmos and four nmos devices. The transistor sizes are indicated in the figure. Only the Pointer requires *wrctl* and *Pin* to control memory writing. The *wrctl* signal covers 10 clock cycles, allowing 1000ns to write the memory cell. Sequential logic similar to parity checking derives *Pin* from the *ld* and *reset* signals. The high of *Pin* covers only the first *ld* pulse, placing a 1 in the first Pointer stage.

Overall, the programming circuitry completes configuration programming over 17x16=272 clock periods.

Test Cell

For debugging purposes during testing, leafcells from the Array, Pointer and Shift Register are placed together into a test structure which is included in the final Crossbar Switch submitted for fabrication.

6.3 Synthesis from Macrocells

Again, the crossbar switch architecture contains the four macrocells discussed. Of these four, three are completely described by one parameter, *nodes*. Providing this parameter, TimLager constructs the all the macrocell layouts. Like

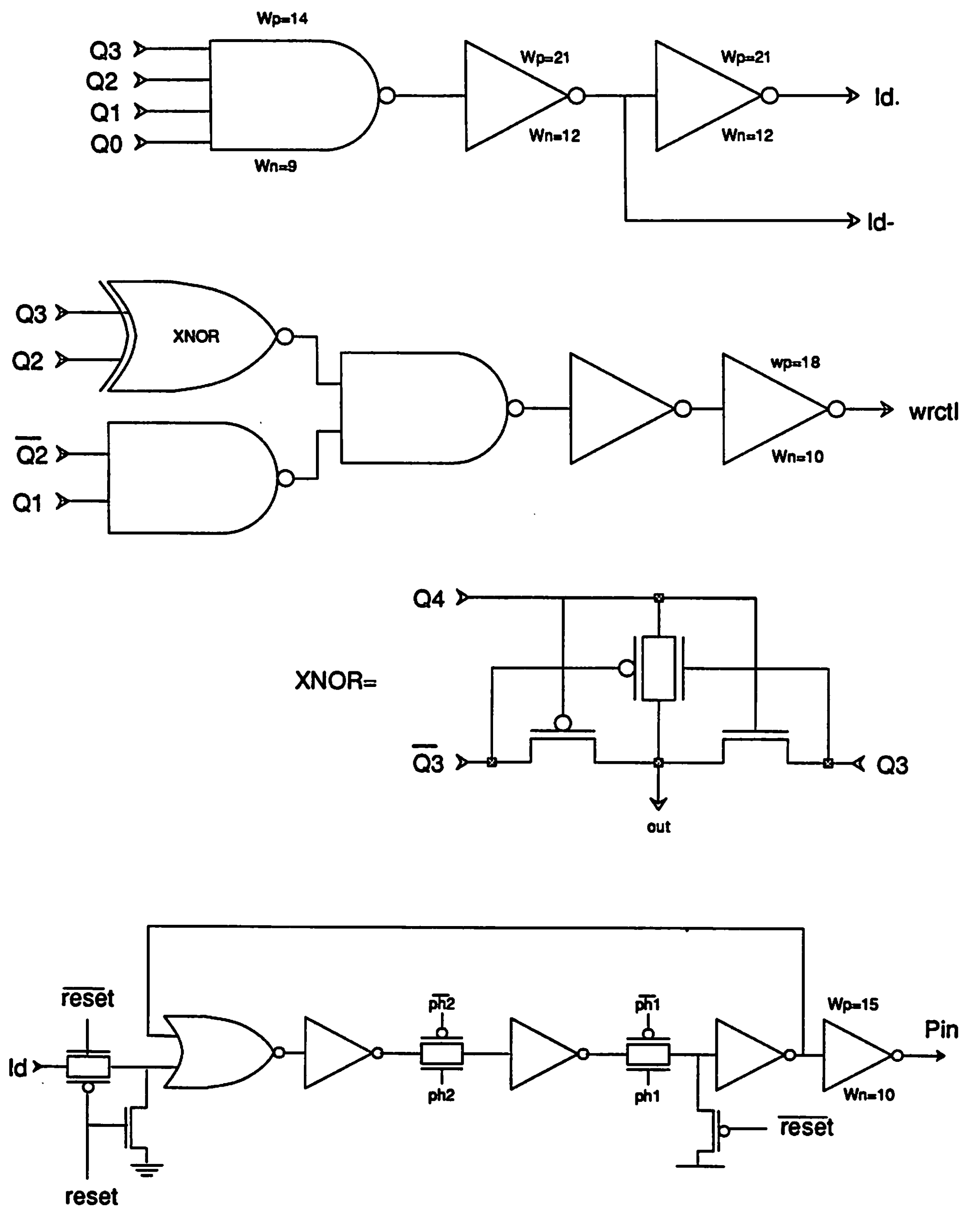
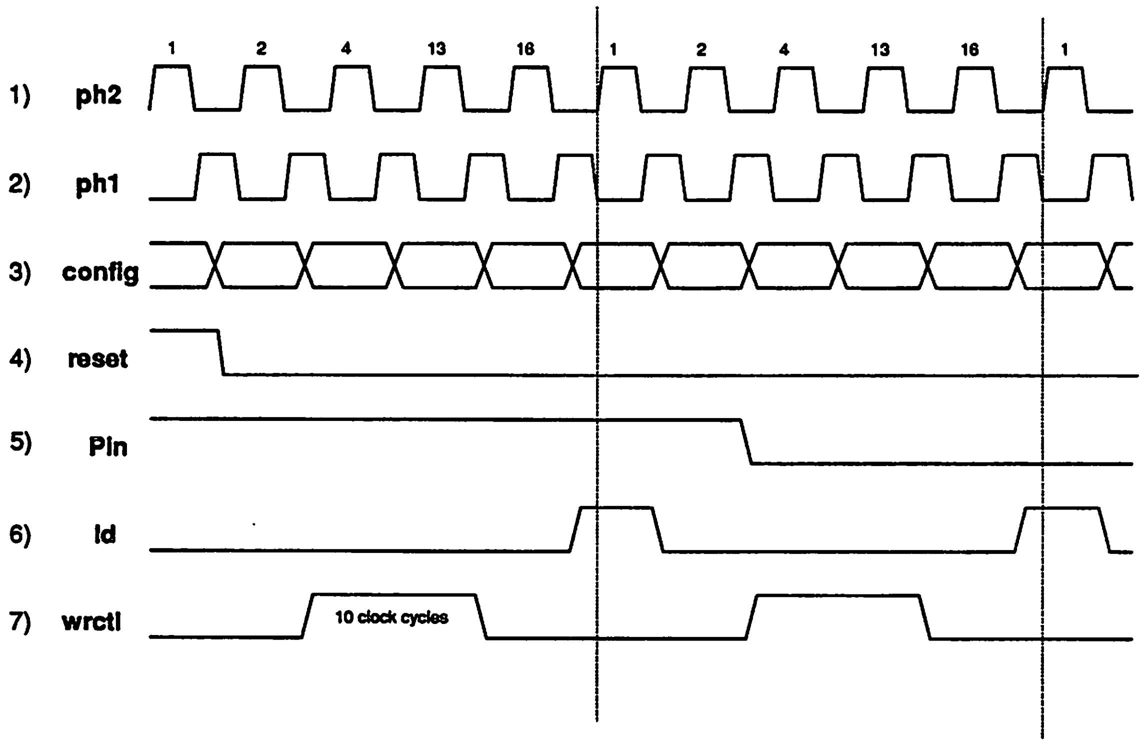


Figure 6.11: Control Signal Logic



config	configuration bit input to Control
reset	input to Control
Pin	output from Control
ld	output from Control
wrctl	output from Control

Figure 6.12: Timing Diagram for Configuration Programming

the Tap and Delay designs, a *nxn* crossbar switch can be generated from the same architecture using this parameterized design and, conveniently, one parameter.

Crossbar Switch Core

For the core layout, the macrocells were manually placed and routed according to the floorplan shown in the architecture of Figure 6.2. In designing the macrocells, the I/O terminals were positioned along particular sides, giving consideration to the floor plan and future routing tasks. The decisions were made to ensure that signal lines connecting macrocells will cross the channel space naturally during routing, minimizing amount of crossovers, metal contacts and signal line lengths. For instance, the Pointer *write* signals wire directly to the corresponding Array *write* signal in metal1, since the macrocell terminals are positioned across from each other. Similarly, in metal1, the Shift Register output lines directly connect to the Array bit lines within a narrow channel between the two macrocells. The Pointer control signal terminals are located at the bottom edge, and the Shift Register control terminals are at its right edge. They connect to the matching Control cell signals appearing at the cell's top edge. Clock lines run along the bottom and left edge of the floor plan. Besides these internal signals, the input channel data *Din*, output channel data *Dout*, *reset* and *configuration* input terminals reside along the outer boundaries of the core. Overall, signal lines tend to run across ground or power lines rather than other switching signal lines. The neatness and compactness of the routing is mostly attributed to forethought put into the leafcell layout. The crossbar switch serves as a good example of the benefits of careful leafcell design.

Chip Layout

Manual Routing from the core to the pad frame, created by TimLager, completes the final layout. Tables 6.1, 6.2, 6.3, and 6.4 describe the pads and their assigned signal for each side of the frame. A picture of the fabricated circuit appears in Figure 6.13. Because the Crossbar Switch has a large number of inputs and outputs, the chip area is determined by the perimeter of pad frame rather

North Side Pads from Left to Right		
name	type	description
Din1	input	channel1 input
Din2	input	channel2 input
Din3	input	channel3 input
Din4	input	channel4 input
Din5	input	channel5 input
Din6	input	channel6 input
Din7	input	channel7 input
Din8	input	channel8 input
Din9	input	channel9 input
Din10	input	channel10 input
Din11	input	channel11 input
Din12	input	channel12 input
Din13	input	channel13 input
Din14	input	channel14 input
Din15	input	channel15 input
Din16	input	channel16 input

Table 6.1: Pads on North Side of Pad Frame

than the area of the core circuitry. The 16x16 Crossbar Switch, fabricated in 3μm technology, occupies a 61mm x 61mm area.

6.4 Simulations

While creating the crossbar switch, the leafcell and macrocell designs were simulated to verify the logic and circuit operation, and the layout connectivity.

The leafcells were first simulated with Spice during their design phase. To meet timing requirements, Spice aided the designer in chosing transistor sizes for driving estimated capacitive loads in sufficient time. In the Pointer and Shift Register cases, the circuits were modeled as two connecting leafcells to simulate the circuit operation in a macrocell environment. Such simulations are more realistic, since the input of one cell depends on the output of another. In the case of the crossbar Array, the interconnect cells are essentially independent of one another. So, it suffices to model the effects of neighboring cells in the macrocell as capacitive

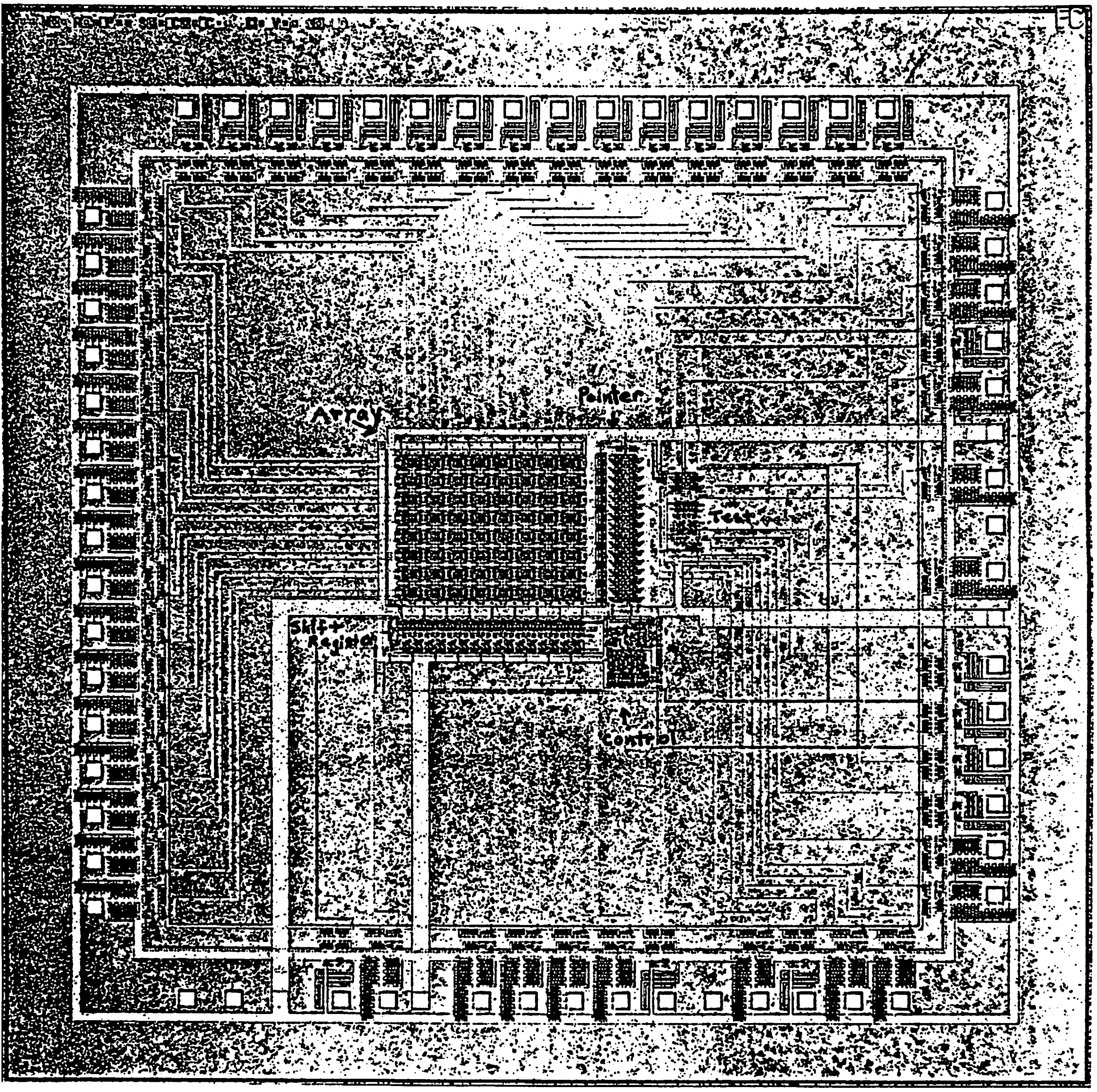
East Side Pads from Top to Bottom		
name	type	description
Pout	output	output from Pointer, for test
wr	output	from test structure
PoutT	output	from test structure
ld	input	to test structure
swd1	output	from test structure
GND	GND	
swd2	output	from test structure
substrate	dummy	chip substrate contact
sroutT	output	from test structure
Vdd	Vdd	
Testin	input	to test structure
ph1	input	ph1 clock phase
ph2	input	ph2 clock phase
Din	input	to test structure
sw-	output	from test structure
Dout	output	from test structure

Table 6.2: Pads on East Side of Pad Frame

South Side Pads from left to right		
name	type	description
Vdd	Vdd	
srin	input	configuration, serial input
SWDin1	output	from Shift Register, for test
GND	GND	
srout	output	from Shift Register, for test
ph2-.s	output	from Shift Register, for test
ph1-.s	output	from Shift Register, for test
write1	output	from Pointer, for test
reset	input	initiates configuration programming
Pout2	output	from test structure
wrctl	input	to test structure
inv	output	from test structure
passout	output	from test structure

Table 6.3: Pads on South Side of Pad Frame

Figure 6.13: 16x16 Crossbar Switch Chip



West Side Pads from Top to Bottom		
name	type	description
Dout16	output	channel16 output
Dout15	output	channel15 output
Dout14	output	channel14 output
Dout13	output	channel13 output
Dout12	output	channel12 output
Dout11	output	channel11 output
Dout10	output	channel10 output
Dout9	output	channel9 output
Dout8	output	channel8 output
Dout7	output	channel7 output
Dout6	output	channel6 output
Dout5	output	channel5 output
Dout4	output	channel4 output
Dout3	output	channel3 output
Dout2	output	channel2 output
Dout1	output	channel1 output

Table 6.4: Pads on West Side of Pad Frame

loads.

After leafcell layout and extraction, only logic simulation was performed with esim to confirm functionality. For the most part, the logic simulations were simple. The only difficulty encountered was in steering the esim simulator toward a convergence for the RAM circuit and an EX-OR circuit. In the static RAM, a pmos device pulls an internal node up while a nmos device pulls down when writing from a 0 to a 1. Since esim prefers only one pulling direction on a node during an event, the contention condition in the RAM causes convergence problems. The EX-OR design has four devices used as pass gates which share a common node. Though the input and output nodes of the pass devices are evident to the designer, esim does not have such a clear concept and treated an intended input node as an output node. Both RAM and EX-OR problems were solved by having esim initialize internal nodes or by applying kesim as the logic simulator. Kesim is a version of esim modified especially for cmos technology.

After the macrocells were designed and assembled with TimLager, esim simulations were performed to verify the logic design and connectivity between leafcells. To keep the input and output test vectors at a tractable length, the ld signal pattern did not pulse every 16 clock cycles, but rather every say 3 cycles. The Pointer, Shift Register and Control macrocells were easily simulated. The chip layout, however, was not extracted and simulated because input test vectors in esim format would be unmanageably long for the designer to produce.

6.5 Testing

A 16x16 Crossbar Switch using the described design was fabricated in $3\mu m$ pwell scmos technology. The fabricated circuit was received and tested in April, 1987. The design was fully functional and met speed requirements.

To locate design faults and to fully verifying the macrocell design, duplicates of the Shift Register, Pointer and Control circuits were placed on separate silicon for testing. As an additional debugging measure, leafcells were also duplicated in a test structure included in the Crossbar Switch silicon. Test results from silicon applying both these test methods confirmed correct leafcell and macrocell performance.

The Crossbar Switch operation conforms to the given timing diagram. The two clock phases are supplied from a source external to the silicon.

Testing the Silicon

Testing the fabricated Crossbar Switch involves brief leafcell and macrocell tests, followed by a thorough test of the Array. After these macrocells, the Crossbar Switch is tested with various network configurations, and results are drawn from measurements and observations. This section describes the test patterns accompanying this procedure and the results from silicon.

Preliminary Tests

In the programming section, selected macrocell output terminals were connected to output pads. This precaution provides a means for quickly checking the programming macrocells in the Crossbar Switch and electrical connectivity between the macrocells. It also helps to isolate errors to the programming section if a failure were to occur there, otherwise the designer may be clueless as to where a potential failure has occurred in the Crossbar Switch.

Checking the Pointer involves observing the register output of the last stage, *Pout*. Since the Control unit sends a 1 bit and subsequently 0 bits to the first stage input, the output exhibits the same sequence occurring 256 clock cycles after the *reset* pulse. This test confirms that the cascade of registers circuits works and that the Pointer is receiving the proper *Pin* and *ld* and *ld-* signals from the Control unit.

The register output of the Shift Register's last stage is also observable from an output pad. The input test pattern is a bit stream applied at the configuration input. To confirm its operation as a shift register, the *srout* output signal should follow the applied input signal, *srin*, 16 clock periods later. The latch output of the first stage is the 16th bit in the input stream, but held for 16 clock periods, verifying proper operation and connectivity between the Control cell *ld* terminal and the Shift Register *ld* terminal.

Observing the *wrctl* signal relative to the *reset* pulse from an output pad completes the Control cell check.

Before testing the Array, the interconnect cell operation was confirmed. An instance of this leafcell appears in the test structure and its internal memory node as well as its terminals were connected to I/O pads for input or observation. Testing this circuit follows the same method as for the Array, which is discussed next.

Crossbar Array Test

In testing the Crossbar Switch Array, the main objective is verifying the full programmability of the Array. This means testing each of the 256 interconnect cells to confirm the writing and switching operation. The cells are independent of one another, but they do share common input, output and control signal lines. The test configuration and input test patterns are designed to verify cell functionality and to detect interference between cells.

Providing the *reset* pulse, a crossbar configuration is programmed into the Array by the programming macrocells. The test configuration pattern allows only one switch, say the (i,j) switch, in the Array to be enabled at a time. So, only one output channel is connected to any input using this test configuration. The configuration input into the Shift Register is then a stream of 0 bits with a one bit inserted in the proper position. For example, an input pattern with a one bit in the first position followed by 0 bits results in a configuration where the first output channel is connected to the 16th input channel. This is seen from inspection of the architecture illustration in Figure 6.2.

Next the test patterns for the 16 input channels are applied to the corresponding input *Pin*, and the signals from the 16 output channels are observed. Since the crossbar design has no read mechanism, the (i,j) RAM cell and switch connection are both tested by applying a toggling pattern to the *Din(j)* input channel and watching the *Dout(i)* output channel for a toggling output signal. Meanwhile, all the other input channel test patterns are constant 0 signals. A toggling *Dout(i)* output indicates that the RAM cell can store a one bit and the switch can connect the data input and output line. Because of the chosen input patterns, it should also suggest that the active output channel *Dout(i)* depends only on the selected input *Din(j)*. Since only one output channel is active in this test, the other output channels should exhibit a constant 1 signal due to the action of the pullups on the output data busses. This observation shows that these outputs are independent of any input, confirming that the other 255 RAM cells have stored a 0 and the switches are disabled. Using the example above, the test pattern applied to the *Din(1)* through *Din(15)* input channels are constanly 0 bits. The *Din(16)* input signal toggles every clock cycle. Watching the output channels, *Dout(2)* through *Dout(16)* exhibit constantly high levels, while the *Dout(1)* signal follows the toggling pattern of the 16th input signal.

The test method and patterns described above are applied another 255 times, enabling a different switch in the Array each time. If all 256 test runs produce output patterns which match the expected results, this confirms that each RAM cell is completely writable and controls its attached switch, and that the switch can connect or disconnect its input and output. Hopefully, the devised patterns thoroughly test the Array for errors such as shorts or opens and faulty interconnection cells.

Crossbar Switch Test

To complete functional verification, the Crossbar Switch is programmed with more realistic crossbar configurations and tested with differing input patterns. Here, the purpose was to test the silicon performance in true environments. The programmed configurations covered cases where several switches were enabled implementing various point-to-point connections. This included input signals routed to multiple outputs. Again, output channel signals should exactly match the selected input signals.

Performance Results

During testing, circuit performance was measured in addition to the functional verification. As mentioned before, circuit performance is dependent on the particular run the chip was fabricated from. The following test results are representative for its particular 3μm pwell run.

The programming circuitry functions at a clock rate of up to 12.5MHz, which is safely beyond the 10MHz requirement. The test chip which contained the separate programming test cells were fabricated on a 2μ run. Upon testing, they functioned up to 15MHz. Based on these results, it is expected that the circuit design will support programming rates in excess of 12.5MHz, depending on the run's feature length. While testing the crossbar Array at 10MHz, the *Din(j)* input was toggled during RAM programming. As an observation, the *Dout(i)* signal starts to follow the input signal within one clock period after the write line is selected, indicating that the RAM cell is written within one clock cycle time. This is well within the 1000ns allotted write time, since the RAM circuit was designed with conservative guidelines.

Performance related to the switch connection was also considered. The propagation delay through the input and output buffer pads were approximately 30ns. The measured delay between the input channel pin and the output channel pin was 40ns. So, the switch, which is a minimum sized cmos pass gate, introduces a 10ns propagation delay into the data stream. The minimum size devices of the switch limits the current available to charge or discharge the input and output data busses, which area loaded with large capacitances. These factors contribute to the 10ns delay of the switch. Although programming was achieved at a 12.5MHz rate, data passing through a switch connection maintains its integrity up to a 15MHz rate, at least. The throughput was not tested beyond this rate due to limitations in the test equipment. In the Crossbar Switch design, the realizable switch data rate and the realizable programming rate are practically independent of one another, because the switch pass gate is an asynchronous circuit design, whereas programming cells use clocked circuitry.

For broadcast network configurations, one data input signal is connected to all 16 output channels. In this case, the signal experienced a 45ns propagation delay through the I/O pads and switch. The input pad here must drive 16 switch outputs. The accumulated capacitive load on the 16 output busses contributes the additional 5ns delay.

Chapter 7

Masked OR Crossbar Switch

Functionally, the Masked OR Crossbar Switch routes information from 16 source channels to 16 destination channels. Compared to the Crossbar Switch described in Chapter 6, this version has the same features and one extra capability. As before, it connects an input to multiple output ports, and, in addition, it allows an output to be connected to more than one input port. This relieves the constraint imposed on the Crossbar Switch of Chapter 6. These concepts are illustrated in Figure 7.1.

To implement multiple input selections, each output signal, *Dout(i)*, can be represented as a logically "ORed" combination of the *Din* inputs, and hence the name for the design. In the previous Crossbar design, the output *Dout(i)* was logically one or none of the inputs *Din(j)*.

The design to be described in this chapter is an inplementation of the Delay-Output Router in the channel emulator. It routes signals from the Delay Lines to the Tap Switches, and the input to output interconnection requirements are reconfigurable. Because of the masked OR feature, this Crossbar Switch supports emulation of multi-source broadcast reception and collision events, where several data signals collide at one receiving node of a network. A parameterized design once again implements the 16x16 Switch, anticipating its expansion into a 64 input/64 output router.

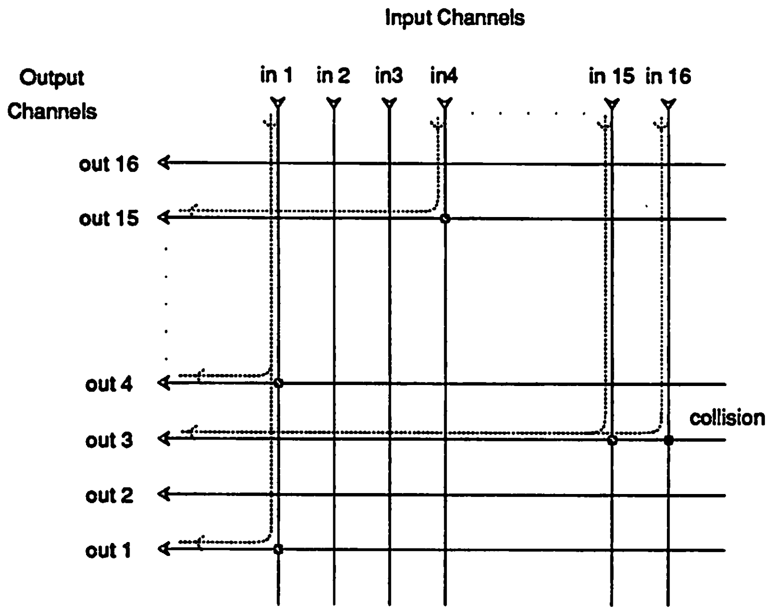


Figure 7.1: Conceptual View of the Masked OR Crossbar Switch

7.1 Architecture

At a high level, this Masked OR crossbar works on the same principles as the previous Crossbar Switch. The architecture consists of the Crossbar Array, Pointer, Shift Register and Programming Control. Each is made up of their respective leafcells placed into a structure of repeated cells, as shown Figure 7.2. The Array interconnection cells has a memory and switch element. The memory circuit is a static RAM circuit, just as before. But, the switch circuit uses the "wired or" circuit technique to implement the logical OR connection. Along the periphery of the Array, input buffers and output buffers support the "wired or" circuitry inside the Array. The input data bus that is internal to the Array is referred to as *din*, and the output data bus in the Array is called *dout*.

In summary, each row of interconnection cells has switch elements which selects data from one of many input channels and directs it to its output channel. The selection of a particular input channel is controlled by the memory elements.

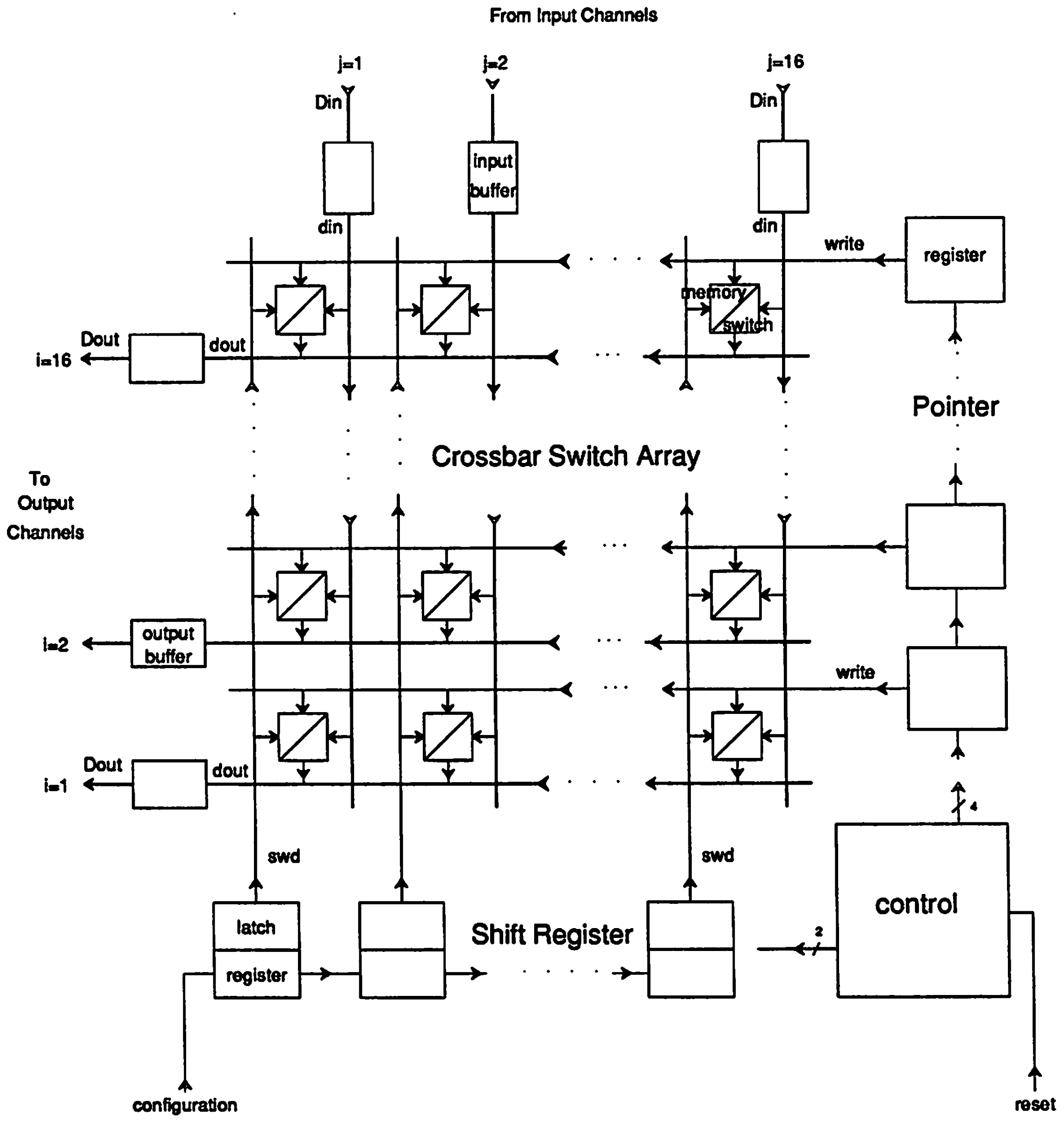


Figure 7.2: Architecture of Masked OR Crossbar Switch

The (i,j)th memory element contains a "1" to select data from the jth input channel for the ith output line and, "0" otherwise. The configuration is programmed in a row by row fashion, as directed by the programming macrocells.

7.2 Macrocell Design and Description

Crossbar Switch Array

To implement the Array's routing function, the switch design uses the "wired or" circuit technique. The switch in the interconnection cell is two nmos transistors connected in series, as shown in Figure 7.5. Device N4 constantly monitors a *din* input bus at its gate and tries to drive the data onto the output bus *dout*. Transistor N3 conditionally passes the input data to its output node, which is the device drain, depending on the state of the switch stored at its gate. Within in Array, all the switches in a row have there output node attached to the same output bus. The N4 sources are connected to a ground path. This is the "wired or" circuit which implements the logical OR of the selected inputs. Figure 7.3 illustrates the array circuitry, and Figure 7.4 shows the Array timing diagram. Each row of switches share an output bus, and each column of switches shares an input bus. To meet speed requirements, the routing design uses precharge and evaluate circuit techniques. Accordingly, the output bus is precharged for during half a clock period and evaluated during the next half. The input buffer synchronizes the *Din* data and drives the *din* input bus, and the output buffer releases the valid *dout* signal and inverts it to recover the true data at the *Dout* node. The clocking scheme inserts a delay of one clock period between the input occurrence and the valid output occurrence. A more detailed description of the Array circuit appears in the following discussion.

In the previous Crossbar Switch design, the switch was a cmos pass gate. There, the input was electrically connected to the switch output. The driving input directly charged or discharged the output bus. For this reason, multiple inputs connected to one output interfered with other output channels. In comparison,

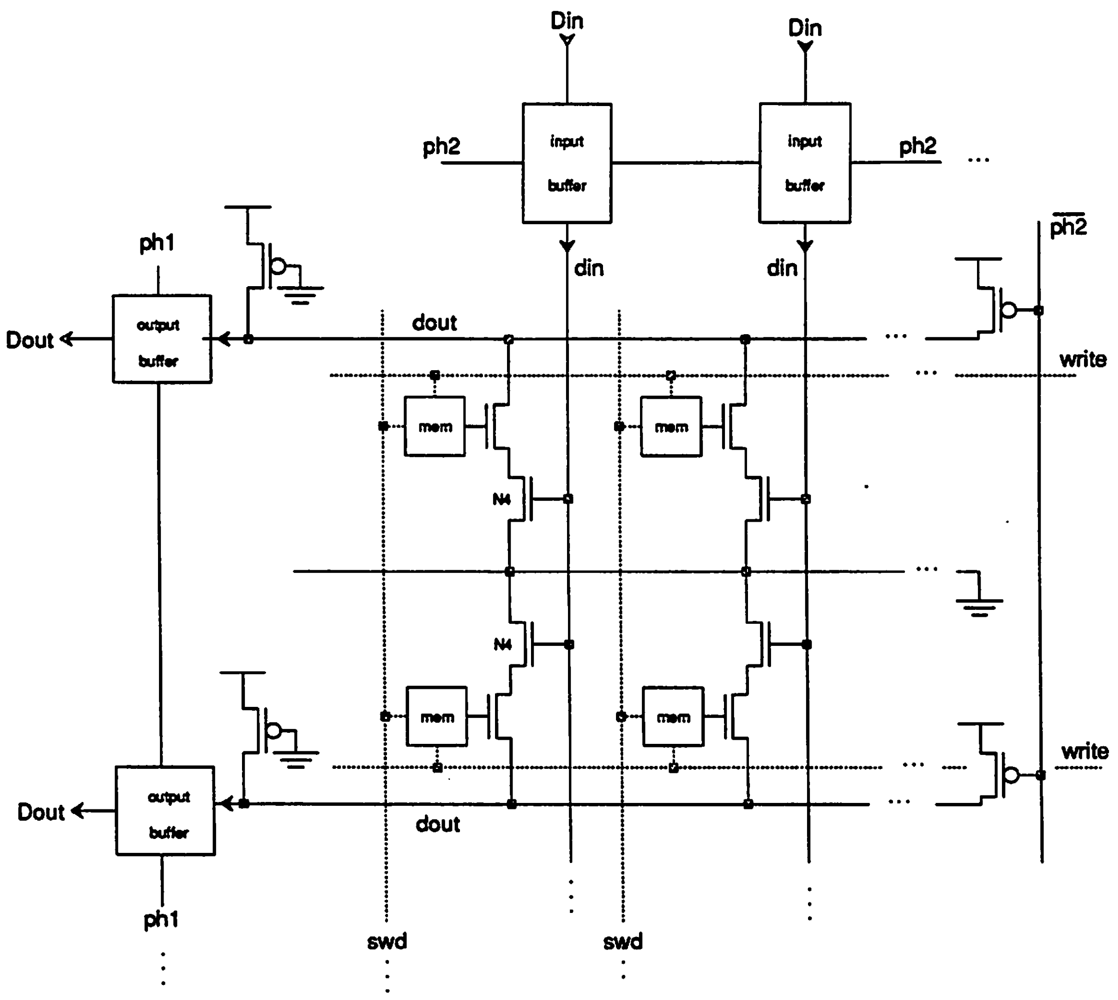


Figure 7.3: "Wired Or" Array Circuitry

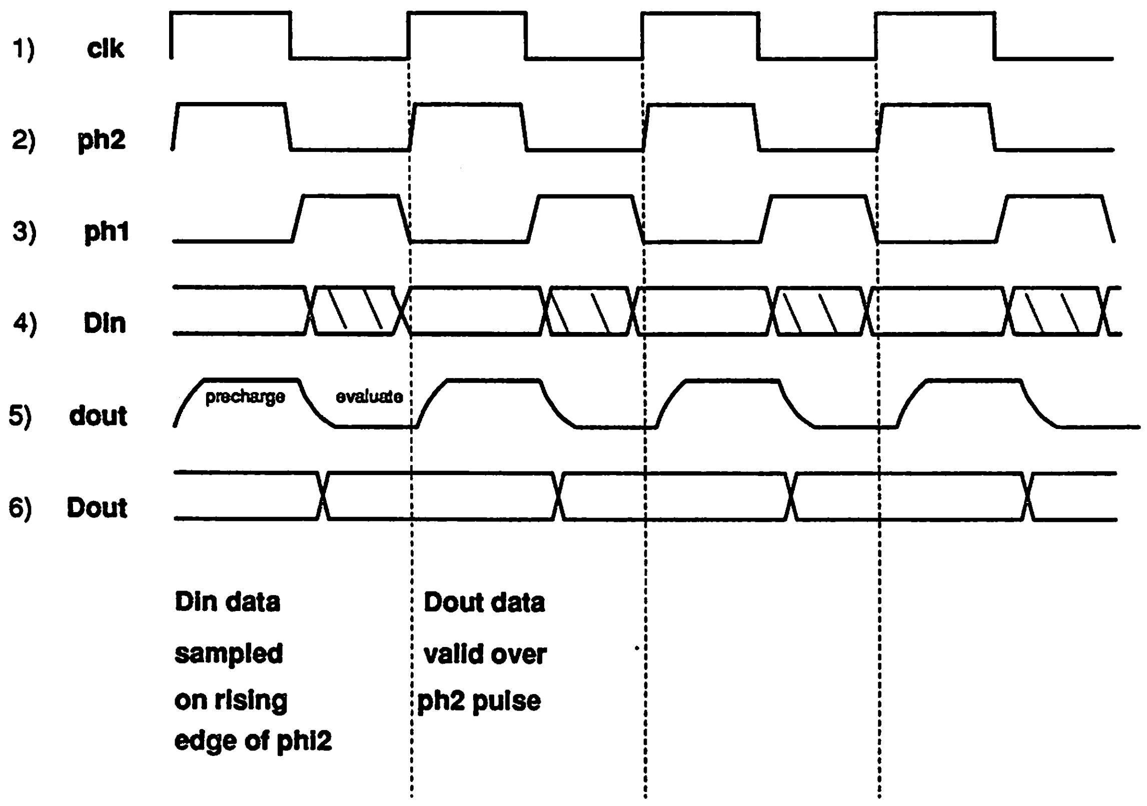

1) clk
2) ph2
3) ph1
4) Din
5) dout
6) Dout
precharge
evaluate
Din data sampled on rising edge of phi2
Dout data valid over ph2 pulse

Figure 7.4: Timing of Array Signals

the "wired or" design "isolates" the switch input from the output bus because the input is at the device gate. The output bus is charged or discharged by the switch itself, rather than the input driver. For this reason, one output channel can select several input signals without interference from other unwanted signals. For example, *Din(1)* and *Din(2)* inputs can be routed to the *Dout(1)* output channel; at the same time, the *Dout(2)* channel can select data from the *Din(2)* input and remain independent of the *Din(1)* input. In the pass gate Crossbar Switch, *Dout(2)* would suffer interference from the *Din(2)* input.

Unlike the previous design, the routing operation in this design is clocked, so all the inputs are synchronous. The circuits which synchronize the input and output signals and clock the switch operation lie along the Array periphery.

Leafcell Design

For the masked OR design, the interconnection leafcell circuit is shown in Figure 7.5. The leafcell contains a five transistor static RAM and the masked OR switch, incorporating the memory and switching functions into one structure. The RAM circuit works on the same principles as that described in Chapter 6. The RAM's internal node, *sw*, controls the gate of N3, turning the switch on or off. The output bus, *dout* is precharged during the high of *ph2*, and a low *din* signal during this time prevents loss of charge through N4. The evaluation phase occurs when the *ph2* phase is low. Valid input data appears at the gate of N4, and the switch conditionally discharges the output bus. If only one input channel is selected, then an *dout* signal is the complement of the chosen *din* signal.

The RAM circuit for this Crossbar Switch has different transistor sizes compared to the RAM discussed in the previous chapter. With the new sizes, the RAM layout area is more compact than before, while write time is decreased. The switch transistors' dimensions were chosen to minimize the output bus capacitance, due mainly to drain diffusion, while maximizing the available current through the switch to discharge the bus within half a clock cycle. With the aid of spice3 simulations which includes the bit line load capacitance for a 16x16 array, the RAM circuit writes a 0 in 10ns and a 1 in 31 ns. For a 64x64 array, the write times are

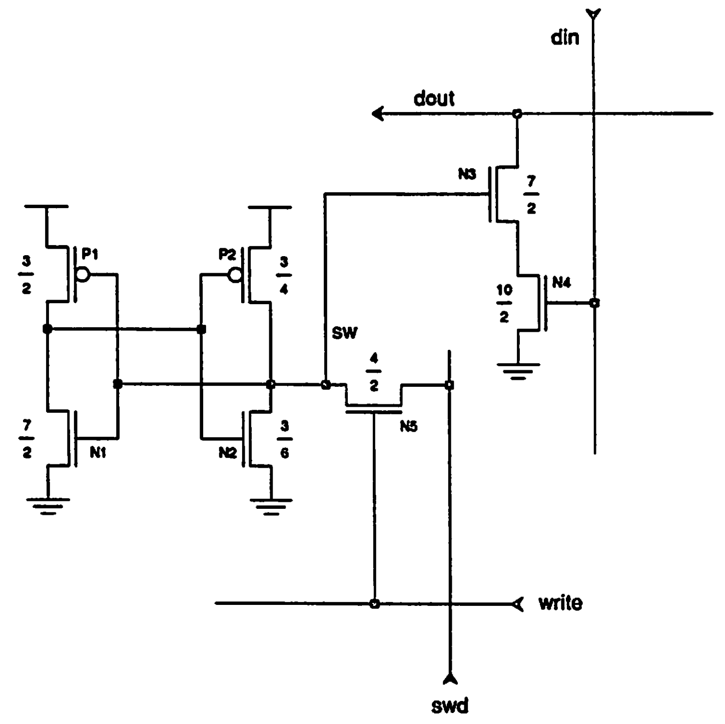


Figure 7.5: Interconnection Leafcell Circuitry

28ns and 50ns, respectively. The two switch devices discharge the 5V precharged output bus within 25ns.

The layout for the interconnection cell is shown in Figure 7.6. The dimensions of this cell are the same as that for the Crossbar Switch in Chapter 6. Also, like the interconnection cell for the first Crossbar Switch , the *write*, *dout* output, and ground lines run horizontally, while the *swd* bit, *din* input, and power lines run vertically. These dimensional and routing features are preserved in the masked OR interconnection cell so that the Pointer and Shift Register designs from the original Crossbar Switch can be re-used, reducing the design effort put into the masked OR Crossbar Switch. For the Array macrocell, adjacent interconnection cells share power contacts and data out contacts.

The periphery circuits that support the "wired or" circuitry include an data input buffer, a data output buffer, and a precharge cell. Figure 7.7 shows the input buffer circuit. There is one of these cells for every input channel. The right-hand cmos pass gate samples the input signal, *Din*, on the rising edge of the

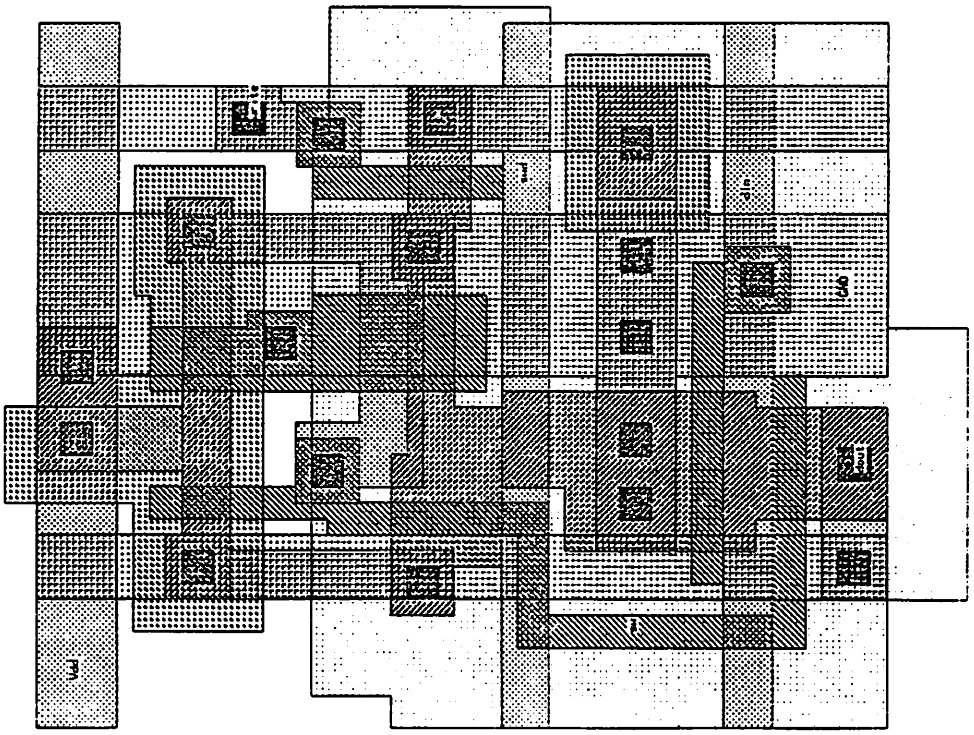


Figure 7.6: Interconnection Leafcell Layout

ph2 phase. On this edge, the circuitry to the left pulls node Y high and, hence, discharges the internal *din* bus low, irrespective of the external input signal. While *ph2* is low, the sampled data is stored at node X and the signal propagates through the right pass gate to the internal input bus *din*. The pmos pullup is turned off. Transistors for the left inverter are sized to drive the input bus between high and low within half a clock period. The parasitic capacitance present on this bus is modeled as cumulative gate oxide capacitances from the 16 switches connected to a bus. The layout for this cell is shown in Figure 7.8. The terminal for the external input, *Din*, appears at the top, while the data terminal *din* resides at the bottom. The placement of the second terminal aligns with the *din* line of the interconnection cell placed just below it. Power lines run vertically in metal1, and they are perpendicular to the *ph2*, *ph2-* and ground lines which are metal2.

Data on the output bus in the Array is evaluated during the low of *ph2*. So, the output buffer cell, shown in Figure 7.9, samples the bus signal at the *dout* node during the high of the *ph1* phase and inverts that signal. If the output bus

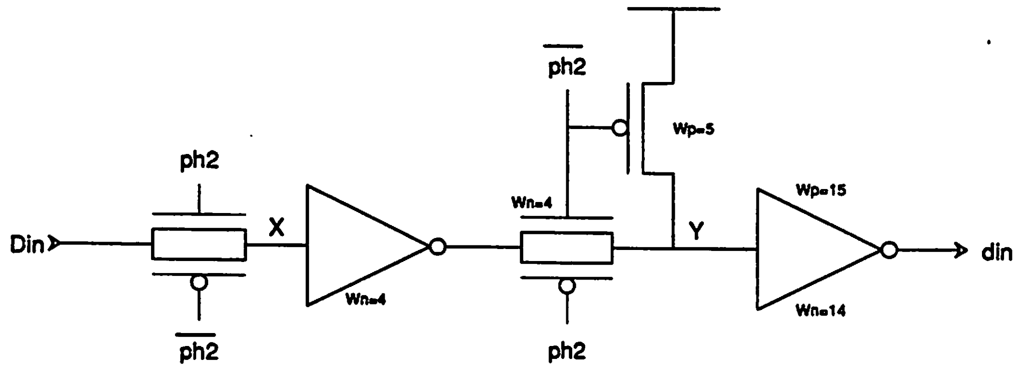


Figure 7.7: Input Buffer Circuitry

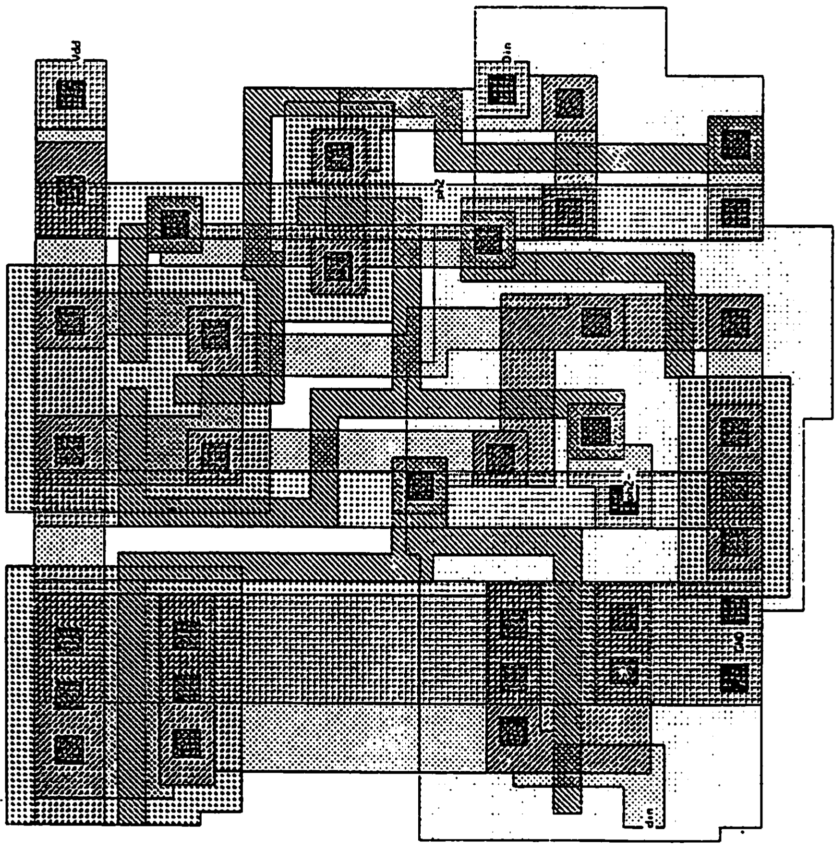


Figure 7.8: Input Buffer Layout

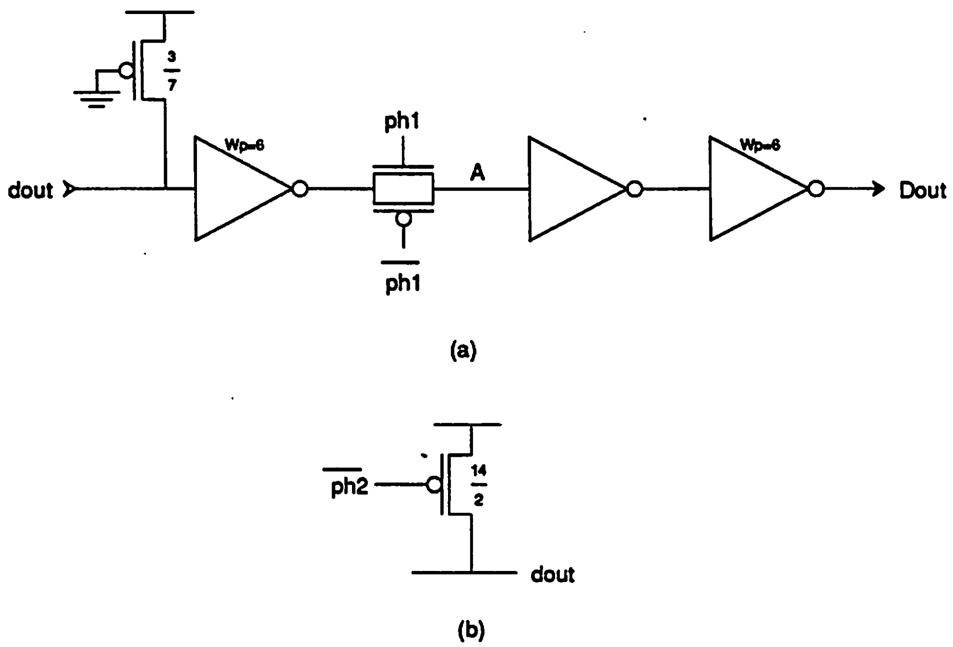


Figure 7.9: (a)Output Buffer Circuitry (b)Precharge Circuitry

signal should be high, the weak pullup attached to the *dout* node helps to maintain this state. Otherwise, the bus is floating and may drift from its high level due to capacitive noise in the Array. Upon the falling edge of *ph1*, the buffer holds the data at node A and the output signal is valid at the *Dout* node. The layout for the output cell is shown in Figure 7.10. There is one output buffer for each output bus, and adjacently placed cells share contacts to power and ground. The *dout* terminal aligns with the output bus line of the interconnection cell placed to the right. Based on spice3 simulations using estimated load capacitances on the output bus, output data switches from high to low within 13ns after the rising edge of ph1, and from low to high in 11ns.

The Array macrocell also includes a precharge cell which is a single pmos device with W/L=14/2 that precharges the output bus to 5V during the ph2 pulse. In addition, there are two cells that drive the clock signals to the input and output circuits, and the precharge circuits.

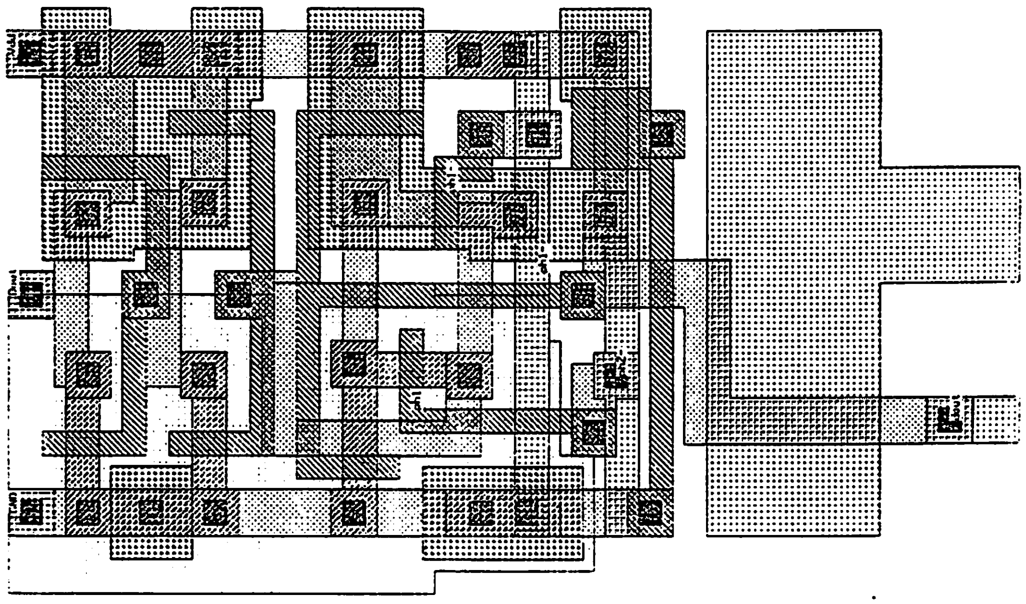


Figure 7.10: Output Buffer Layout

Macrocell Design

The Array macrocell is layouted out as a 16x16 matrix of interconnection cells. As in the previous Array, the leafcells are mirrored about the X and/or Y axis and abutted against neighboring cells to share contacts. Along the macrocell periphery, 16 input and 16 output buffers are set along the top and left edge, respectively. The precharge cells are located at the right edge. The two clock signal drivers are placed at the upper left and right corners. Figure 7.11 illustrates the cell placement to construct a 4x4 Array macrocell. The data input terminals and data output terminals are distributed along the top and left edge, respectively; the write signal terminals, included in the precharge cell layout, appear at the right and the configuration signals, *swd*, enter the macrocell from the bottom.

Shift Register and Pointer

In programming the crossbar configuration, the Array at the macrocell level behaves just as that for the previous Crossbar Switch. This makes it possible

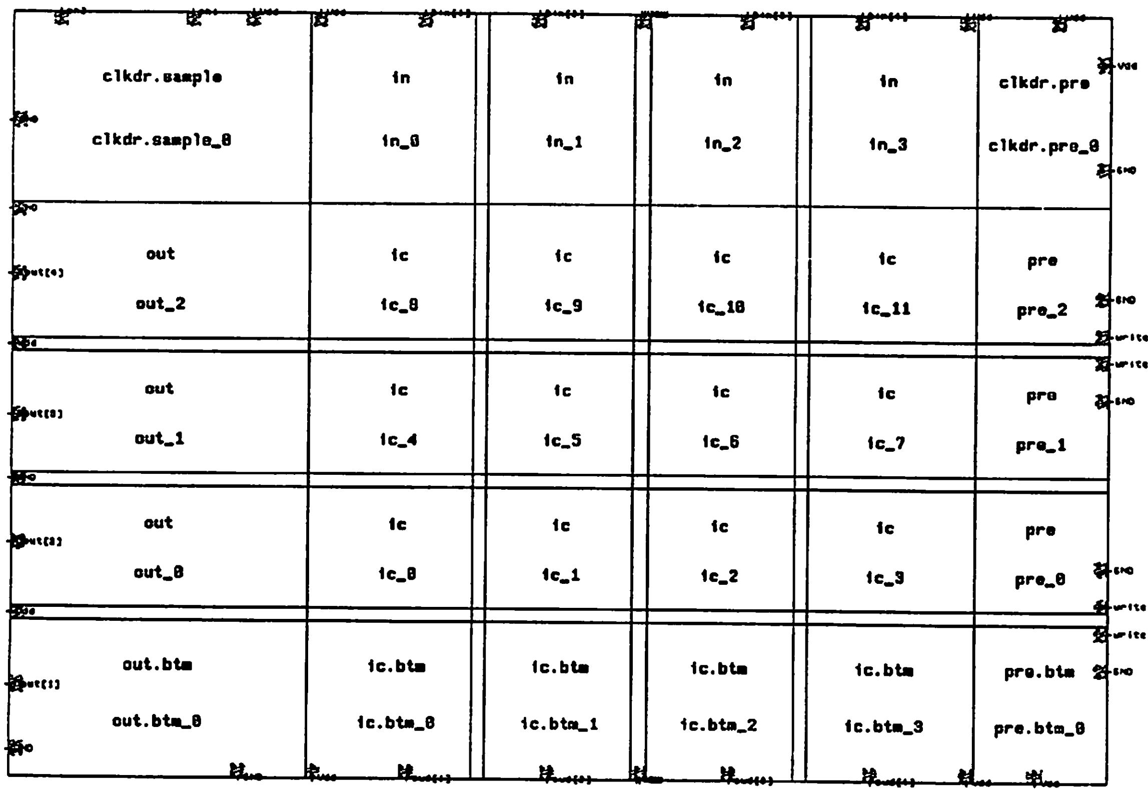


Figure 7.11: 4x4 Array Layout

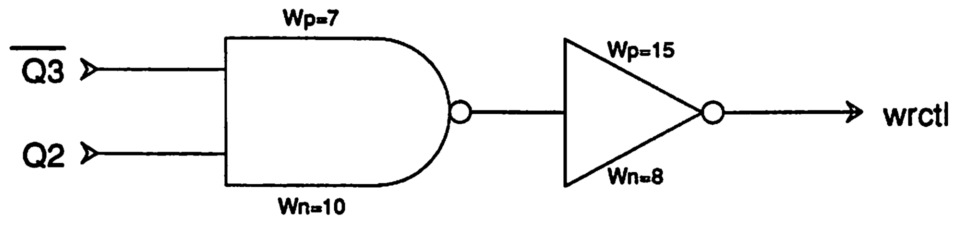


Figure 7.12: *Wrctl* Control Signal Logic

to reuse the same Shift Register and Pointer circuit and layout design, discussed in Chapter 6, for the programming section of the masked OR Crossbar Switch. Also the RAM design for this Array presents an estimated bit and write line load capacitance that is less in value than those for the previous Array. So, for the masked OR Array, the Register and Pointer circuits' driving capability should be more than adequate to charge and discharge these line capacitances in the required time. There is practically full confidence in their circuit and layout design reliability, since a fabricated original Crossbar Switch demonstrated a Shift Register and a Pointer macrocells that performed within requirements.

Programming Control

The Programming Control macrocell generates the control signals to program the configuration into the Array memory. It derives the *ld*, *ld-*, *Pin*, and *wrctl* signals from a two phase non-overlapping clock and the *reset* pulse. This is implemented with the 4-bit counter and the logic that generates the *ld*, *ld-* and *Pin* signals described in Chapter 6. The *wrctl* signal logic was redesigned for the masked OR Crossbar Switch. Here, the *wrctl* circuit is a NAND gate followed by and inverter to produce the logical AND of the *Q2* and *Q3-* output signal from the counter. This is shown in Figure 7.12, and the resulting *wrctl* pulse covers four clock cycles, allowing 400ns to write a memory cell. The previous Crossbar memory write time lasted 1000ns. According to spice3 simulations of the RAM circuit, the 400ns time is more than sufficient to write the circuit. Tests results for the fabricated

Crossbar Switch demonstrated that the memory cell content could change within one clock cycle, reinforcing the choice for a shorter write time. Figure 7.13 shows the timing requirements for configuration programming.

Clock Generator

The Array, Shift Register, Pointer and Control macrocells operate from a two phase non-overlapping clock signal. The clock generator described in Chapter 5 is placed into the masked OR Crossbar design to supply the two phases. The four macrocells include local clock drivers to generate their own *ph1* and *ph2* phase complements.

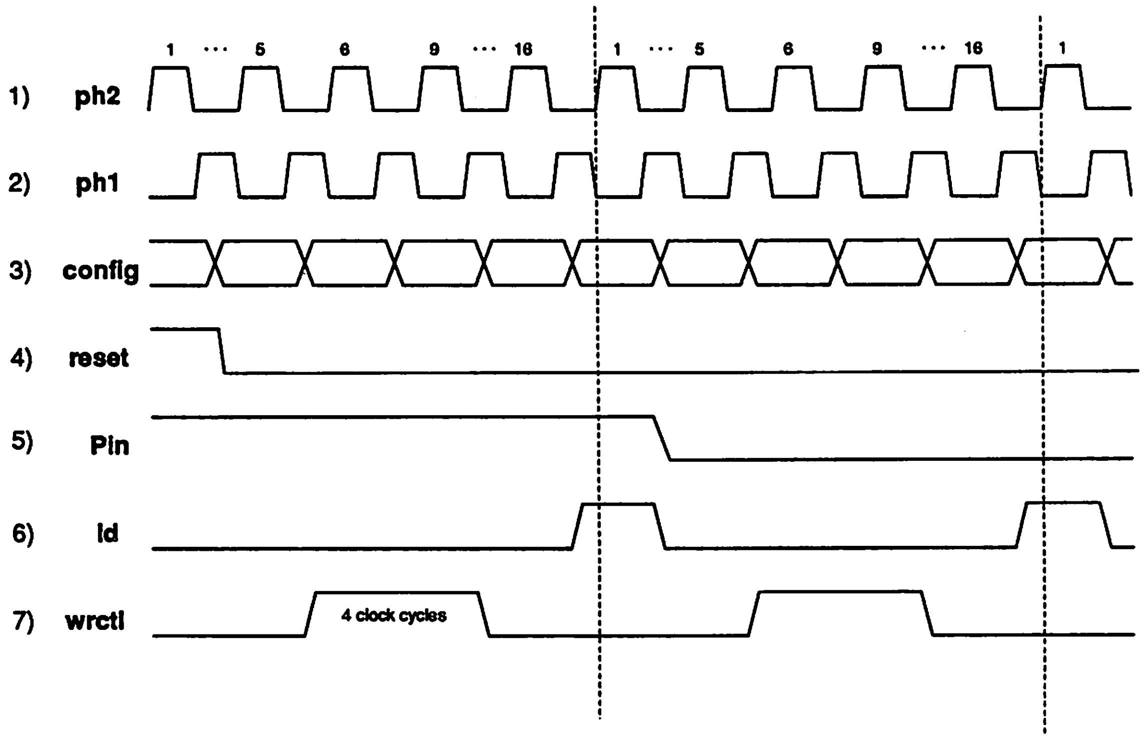
Test Cell

Anticipating debugging during future testing, the five leafcells that comprise the Array are placed into a test structure. This test block is included in the final masked OR Crossbar Switch submitted for fabrication.

7.3 Synthesis from Macrocells

Crossbar Switch Core

Other than the test structure, the functional circuitry realizing the masked OR architecture consists of the five macrocells discussed above. Through Design Manager, these macrocell layouts are generated automatically by TimLager given the one parameter, *nodes*. Including the test cell, they are placed and routed manually to form the Core circuitry, which is treated at a higher level as a macrocell also. The core layout is shown in Figure 7.14. The floor plan follows the same placement as in the previous Crossbar Switch. The clock generator and test cell is placed along the left side of the core. The layout produced uses the same routing strategy as described for the core circuitry of the previous Crossbar. The *Din* input terminals are at the top edge of Figure 7.14, while the *Dout* output terminals are



config	configuration bit input to Control
reset	input to Control
Pin	output from Control
ld	output from Control
wrctl	output from Control

Figure 7.13: Timing Diagram for Configuration Programming

located along the right side of the core boundary. The *configuration*, *reset* and clock signal terminals are distributed along the bottom and right edges. Several terminals included for test purposes reside along these edges also.

Chip Layout

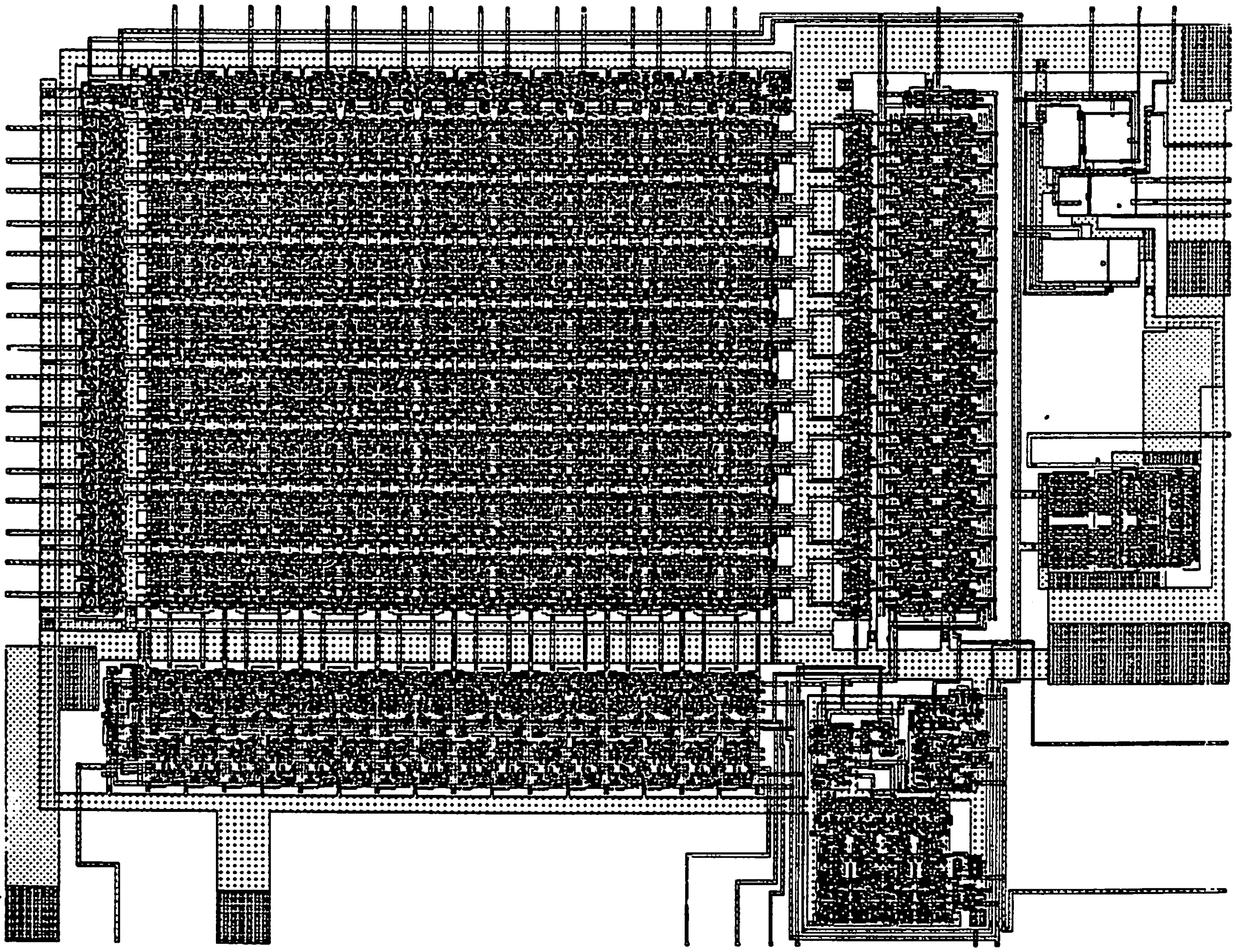
The chip layout is generated with Flint. Since the place and route from core to pads was performed manually for the previous chip, this chip was generated at one higher level of automation. Using an *sdl* description that treats the core and each side of the pad frame as macrocells, Design Manager invokes TimLager which generates the macrocells. Then it calls Flint which places the four pad groups into a frame structure surrounding the core and routes from the core terminals to the corresponding pads. For each pad group in the frame, Table 7.1, 7.2, 7.3, and 7.4 describes the pad types residing in that group and the signal assigned to the pad. To reduce inductive voltage drops during current switching, the pad frame includes several ground and power pads distributed evenly through the frame. The area consumed by the Crossbar Switch is pad limited for the 16x16 case. A fabricated masked OR Crossbar switch, in 3μm scmos technology, occupies 61mmx61mm of silicon area. The original Crossbar switch occupied the same amount of area.

7.4 Simulations

Circuit simulation aided in designing the leafcell circuits. Logic simulations verified the leafcell logic and layout connectivity. The simulations simulations used the same input test patterns and encountered the same difficulties as in the original Crossbar Switch.

At the macrocell level, layouts were extracted and then logically simulated to confirm their functionality. For the Array, Shift Register and Pointer macrocells, TimLager generates the layout by tiling the leafcells according to a repeated pattern. Specifically, leafcells are stacked vertically to create the Pointer, odd and even leafcells are alternately abutted to form the Shift Register, and interconnec-

Figure 7.14: Core Layout



North Side Pads from Left to Right		
name	type	description
Vdd	Vdd	
Din1	input	channel1 input
Din2	input	channel2 input
Din3	input	channel3 input
Din4	input	channel4 input
Din5	input	channel5 input
Din6	input	channel6 input
Din7	input	channel7 input
Din8	input	channel8 input
Din9	input	channel9 input
Din10	input	channel10 input
Din11	input	channel11 input
Din12	input	channel12 input
Din13	input	channel13 input
Din14	input	channel14 input
Din15	input	channel15 input

Table 7.1: Pads on North Side of Pad Frame

East Side Pads from Top to Bottom		
name	type	description
Din16	input	channel16 input
Pout	output	output from Pointer, for test
in	input	to test structure
out.ip	output	from test structure
out.op	output	from test structure
GND	GND	
pre.ic	input	to test structure
substrate	space	chip substrate contact
wr.ic	input	to test structure
Vdd	Vdd	
out.ic	output	from test structure
swd.ic	input	to test structure
clk	input	single square wave clock
Pin	output	control signal from Control macrocell
reset	input	initiates configuration programming
testout	output	for test

Table 7.2: Pads on East Side of Pad Frame

South Side Pads from left to right		
name	type	description
Doutl	output	channell output
Vdd	Vdd	
config	input	configuration, serial input
GND	GND	
srout	output	from Shift Register, for test
ld-	output	control signal from Control macrocell
ld	output	control signal from Control macrocell
wrctl	output	control signal from Control macrocell
phi2	output	from clock generator
phil	output	from clock generator
Vdd	Vdd	
GND	GND	
testout*	output	for test
testin	input	for test

Table 7.3: Pads on South Side of Pad Frame

West Side Pads from Top to Bottom		
name	type	description
GND	GND	
Dout15	output	channel15 output
Dout14	output	channel14 output
Dout13	output	channel13 output
Dout12	output	channel12 output
Dout11	output	channel11 output
Dout10	output	channel10 output
Dout9	output	channel9 output
Dout8	output	channel8 output
Dout7	output	channel7 output
Dout6	output	channel6 output
Dout5	output	channel5 output
Dout4	output	channel4 output
Dout3	output	channel3 output
Dout2	output	channel2 output

Table 7.4: Pads on West Side of Pad Frame

tion leafcells are alternately mirrored and abutted to generate the Array. Actual simulation used macrocells formed by tiling with the prescribed pattern once. This should suffice as a substitute for the full sized macrocell, since simulating an extracted 16x16 array or 16 cell Pointer/Shift Register with the esim or kesim tool is practically unmanageable. A 4x4 Array was extracted and simulated with kesim. The simulation verified that an output could be connected to none, one, two or three inputs after the memory cells were programmed. To ensure that mirroring of the array's leafcells did not cause errors at the macrocell level, logic simulations were performed with various configurations. A two cell Shift Register and Pointer was created by their TimLager routines and then extracted and simulated. These simulations were straight forward.

7.5 Testing

A 16x16 Masked OR Crossbar Switch implemented with the described design has been placed on a chip, and submitted in November, 1987 for fabrication in 3μm pwell scmos technology. After return from fabrication in March, 1988, the silicon was tested and performed properly at 20MHz. Tests beyond this rate give unreliable results due to equipment limitations. On chip, an input pad terminal was connected to an output pad terminal. Testing this structure, a 14ns propagation delay through the two pads combined was measured.

The submitted chip provides output pins for quick macrocell checks. The output of the last register in the Shift Register is connected to an output pad. During testing, this output signal should be the configuration signal delayed by 16 clock periods. The Pointer also has its last register's output brought off chip for observation. All four control signals produced by the programming control unit have their terminals connected to output pads also. These signals should match those shown in the timing diagram of Figure 7.13. The test strategy is the same as that described in Chapter 6. This type of testing will help isolate errors to their respective macrocell if any were to occur during testing, although it may not help in finding the cause of the problem to a great extent. The test structure has its input and output terminals connected to I/O pads. The Array leafcells can be directly tested by applying test patterns to the input pads and watching the leafcells response at the output pad. After verifying that the programming macrocells and the Array leafcells work, the next step is testing Array operation.

The purpose in testing the Crossbar Switch Array is to check that each RAM cell is fully programmable and that each switch can be turned on to pass the selected input signal or turned off. The test method and test vectors discussed for the Crossbar Array test in Chapter 6 are also applicable for this case. The input signal applied to the selected channel is a stream of alternating 1 and 0 bits. Because signals processed by the Array switches are clocked, the toggling signal observed from an output pad is delayed by one clock cycle relative to the toggling input signal. The outputs *Dout* which are independent of any data input exhibit a constant 0 signal due to action of the pullups on the output bus inside the Array.

To complete functional testing, the fabricated chip should operate with realistic configurations. Importantly, to test the masked OR feature, configurations where an output is dependent on several inputs are used, as well as the case where an output is connected to one or no input channel. The observed output data is expected to be the logical OR of the selected input data. Strictly speaking, the Masked OR Crossbar specifications do not require that the output be the logical OR. Since, in an collision event where data from multiple sources collide at one destination node, the received data would be considered as invalid anyway. But, for this Crossbar Switch, the "wired or" circuitry preserves the logical integrity of the output signal during a collision. During testing, this helps to show that unselected inputs are not interfering at the observed output.

Chapter 8

Impact of LAGERIII Tools and Suggestions

Throughout this project, IC design tools were applied to logic minimization, simulation, leafcell abutment and tiling, and macrocell placment and routing. These tools saved development, redesign, and debugging time throughout all levels of the design hierarchy, and this savings was their greatest benefit. Specifically, the LAGERIII system reduced the time devoted to layout generation, allowing the designer to place more effort on architectural, logic and circuit design, and simulations.

Design Manager provided a structured and tractable implementation of the design process. For the Tap Switch design, the numerous parameters that describe the PLA contents and the connectivity specifications between PLAs, the address latch, and the I/O pads were systematically listed in the *sdl* description. This provided an organized way to manage the design information. The same holds for the Delay line and the Crossbar Switches. Because the described chips have highly modular architectures, the LAGERIII tools, TimLager and Flint, were well suited to generate the macrocells.

TimLager is the most frequently used of all the tools. The flexibility of the chip designs is extended by TimLager's notion of a parameter. In the Tap Switch example, a change in the PLA contents and the size of the address latch can be easily reflected in layout by reassigning the parameter values. For the Delay design, delay lines of different lengths are generated from predesigned leafcells and the same module *.c* routine. The Crossbar Switch macrocells are described by the one parameter, *nodes*. Given the parametric descriptions of the modules, the LAGERIII tool can generate Crossbar Switches, that accommodate different numbers of channels, from the same architectural and circuit design.

TimLager also reduces redesign time of macrocell layouts. In a few cases, during macrocell development, the designer modified a leafcell layout that meant changing the amount of overlap during abutment. To generate the new macrocell layout, the designer needs only to make minor modifications to the *.c* routine placement instructions and invoking TimLager. Currently, the designer must check that terminals along leafcell boundaries, whether signal, power or ground, abut properly after TimLager tiles the cells. This is done by manual inspection while in magic, or extraction and simulation. If TimLager supports a program that looks at a terminal netlist and extracted terminal positions, and then checks that terminals meet corresponding ones along neighboring cells, it could be an even more powerful tool and could save macrocell development time.

For placing and routing macrocells including core and pad groups, Flint was also useful. Many times during first passes, it produced results that contained design rule violations or awkward routing that compromised design reliability. Nevertheless, it at least gave a good initial place and route solution. From there, the designer manually edited the initial result into an acceptable final layout. This saved development time in floor planning and in connecting large amounts of signal terminals, since first pass placement and route performed manually is error prone and time consuming. Overall, custom effort was most frequently applied to distributing clock lines and widening power and ground lines to ensure design reliability.

As a suggestion, a tool that performs electrical connectivity checks would enhance LAGERIII system capabilities. Before logic simulation of the layout, connectivity between macrocell signals was visually verified during a magic session. Similar in concept to the leafcell terminal check, the tool could use the netlist provided in the *sdl* or *hdl* descriptions to check signal connectivity in an extracted layout. Connectivity verification will give the designer added confidence in the design prior to logic simulation, and it will save design time in the long run by detecting connectivity errors at an earlier stage.

Chapter 9

Future Work

To be truly useful and meaningful, the set of chips needs to be incorporated into a channel emulator system. After all, this application is their intended purpose. Future work is in system level design, and further work with these ICs should be directed toward adjusting the chip design to support a 32 node system.

System level design will include development of the interface between the channel emulator chips and the node emulator according to specifications. Behavioral modeling and system level simulation will aid the design. At a higher level, the Input and Output Router address is up to six bits wide and provided in parallel. So, to program the channel emulator, system design also requires developing software that converts the address into a decoded, bit serial configuration address for the Crossbar Switches. Board design issues include definition of hardware specifications, hardware design, distribution of global clock signals, and testing and debugging the fabricated board with and without the chips.

To implement a 32 node system, a 64x64 crossbar switch could be generated from the described design. The parameter nodes is reassigned from 16 to 64. The Array, Pointer, and Shift Register design for the 16x16 case is generic and applies to a 64x64 case. But, since parasitic capacitances that load the write, bit, and data lines increase linearly with the parameter value *nodes* the line drivers in the Shift Register, Pointer and Array I/O buffers may need their W/L ratios to be increased. Presently, the drivers are designed to surpass speed requirements for a 16x16 operating environment. Nevertheless, to implement a 64x64 Crossbar Switch, re-evaluating the driver designs and increasing the driving capability will ensure that the Crossbar Switches still function at 10MHz. This consequently means modifying existing leafcells.

In addition, the program control macrocell needs to be redesigned for the 64x64 Crossbar Switches. The *ld*, *ld-*, *Pin*, and *wrctl* control signals will be derived from new logic and a 6-bit counter. The same leafcell circuits provided by Alex Lee's counter design can be used again for the new counter.

Interestingly, the Masked OR Crossbar Switch can also be substituted for the regular Crossbar Switch, without violating system specifications. The difference is that the Masked OR Crossbar synchronizes its data inputs and outputs, whereas the regular Crossbar Switch can handle inputs and outputs that are asynchronous.

Unlike the Crossbar Switch, the Tap Switch and Delay Line design is independent of the number of nodes in the channel emulator system. So, their circuit design is completely applicable to any nxn emulator. Of the four chips designed, the Delay Line and Masked OR Crossbar Switch have local two-phase non-overlapping clock generators. A Tap Switch that includes a clock generator could be placed on a chip and submitted for fabrication. Since the current Tap Switch area is pad limited, an additional clock generator would occupy area that is now unused. With clock generators placed in every chip, the clock signal generation and distribution problem at the board level will be simplified, since a 10MHz symmetric waveform needs to be routed to the ICs rather than two waveforms.

Chapter 10

Conclusion

This report has described the chip designs for a reconfigurable channel emulator system, and it has discussed the design methodology which integrates the use of CAD design tools.

First, all the ICs are programmable, and they can implement a variety of network topologies. The Tap Switches interface the node emulator to the channel emulator. The Delay Lines insert variable delays into a bit stream, and the Crossbar Switches provide reconfigurable network connections. The modular architecture of each design realizes a channel emulator which can be incremented to any number of nodes, n, and exploits the parametric nature of the LAGERIII tools. This minimizes the chip redesign effort for implementing an n node system. The four chip designs have been fabricated and tested, and are fully functional at the required 10MHz operating frequency. The results of the chip designs are summarized in Table 10.1.

Second, generating the chips with automatic layout tools used a hierarchical design approach for organization, design flexibility, and development time savings. This project demonstrated the importance and usefulness of CAD tools in IC design, as the chips evolved from the conceptual stage into layout. It has shown that fast, efficient and reliable chip design is made possible by applying the LAGERIII tools in a structured approach.

circuit	comment	feature	area[1]	# of xtors[2]	speed
Tap Switch	version 1 version 2	3μm 2μm	2.3mm^2 1.1mm^2	1,110 1,110	9MHz 20MHz
Delay Line	$0 \leq N \leq 1023$	3μm	19.8mm^2	19,744	15MHz
Crossbar Switch	16 inputs/ 16 outputs	3μm	3.3mm^2	3,220	12.5MHz
Masked OR Crossbar Switch	16 inputs/ 16 outputs	3μm	4.2mm^2	3,466	20MHz

[1] area of core circuitry
[2] in entire chip area

Table 10.1: Circuit Information Summary

Bibliography

[1] A. M. Kao, L. F. Ludwig, "A Flexible Channel Emulator for Communications Protocol and Architecture Research", ERL M85/83, November 1985, U.C. Berkeley EECS Department.

[2] Neff, Robert, "Design of a Custom IIR Filter using LAGER", M.S. Report, May 1987, U.C. Berkeley EECS Department.

[3] Lee, Chang-Chuan Alex, Unpublished Work.

[4] R. Jain, *TimLager USERS MANUAL*, U.C. Berkeley, September 1986.

[5] C. S. Shung, R. Jain, *Design Manager for Lager-III*, U.C. Berkeley, January 1987.

[6] Mukherjee, Amar, *Introduction to nMOS and CMOS VLSI Systems Design*, Prentice-Hall, Englewood Cliffs, NJ, 1986, pp. 211-223.

[7] J. Rabaey, S. Pope and R. Brodersen, "An Integrated Automatic Layout Generation System for DSP Circuits", IEEE Trans. Computer-aided Design, CAD-4(3):285-296, July 1985.

[8] P. Ruetz, et al, "Automatic Layout Generation of Real-Time Digital Image Processing Circuits", CICC 1986.

Appendix A

Macrocell Library

The macrocells related to the Tap Switch, the Delay Line, the Crossbar Switch, and the Masked OR Crossbar Switch are listed below for each design. The list is arranged in hierarchical order, showing how the cells are composed of macrocells down to the lowest level macrocell. Each macrocell entry starts with the instance name, which is usually shown as it appears in the brodersuns file system, and it is then accompanied by at most four descriptive pieces of information. First, the generic macrocell name, if one exists, follows the instance name. Second, if the macrocell was developed by the author, parameters associated to the generic cell are given. Then a brief explanation about the macrocell appears in brackets []. Finally, the entry may give names of *.sdl* or *.parval* files that contain important netlist information and assigned parameter values.

All the files related to the macrocells reside in subdirectories of the ~sun or ~lager directories. After the hierarchy of macrocells, a list of paths to the related directories are given. This includes paths to the *.sdl*, *.parval*, and TimLager *.c* and *.o* files. Paths to magic layout files throughout the layout hierarchy appear last. All the magic files for leafcells contained in a macrocell are located together in the same directory. These leafcell directories are under their respective TimLager directories.

Leafcell layout designs follow scmos design rules. Their circuits are described in preceding chapters of this report.

```
Tap Switch
    tapcore [core circuitry]: tapcore.sdl tapcore.parval
        data_tap fsm [data PLA]
        carrier_tap fsm [carrier PLA]
        violation_tap fsm [code violation PLA]
        config_latch latch(parameter nbits) [configuration address latch]
    pad frame
        north40 scpads2 [top group of 2u pads]
        south40 scpads2 [bottom group of 2u pads]
        east40 scpads2 [right group of 2u pads]
        west40 scpads2 [left group of 2u pads]

(sdl-parval ~sun/tap/tap_rev2/core)
(TimLager ~sun/Tim/fsm ~sun/Tim/latch ~lager/LagerIII/lib/TimLager/scpads2)
(magic ~sun/tap/tap_rev2/chip ~sun/tap/tap_rev2/frame/layout
~sun/Tim/scpads2/leafcells ~sun/tap/tap_rev2/core/layout
~sun/Tim/fsm/leafcells ~sun/Tim/latch/leafcells)
```

```
Variable Register Delay Line
    core: dlay.sdl
        data_du du [data Delay Unit]: dcore.sdl
            d1 dline(manually created) [shift register of length 1]
            d2 dline(manually created) [shift register of length 2]
            d4 dline(parameter indx cols rows)[shift register of length4]
            d8 dline(see above) [shift register of length 8]
            d16 dline(see above) [shift register of length 16]
            d32 dline(see above) [shift register of length 32]
            d64 dline(see above) [shift register of length 64]
            d128 dline(see above) [shift register of length 128]
            d256 dline(see above) [shift register of length 256]
            d512 dline(see above) [shift register of length 512]
            muxctl pgmux(parameter m) [multiplexer control]
        carrier_du [carrier Delay Unit]: dcore.sdl
                see data_du
        violation_du [code violation Delay Unit]: dcore.sdl
                see data_du
        dvalue_latch latch [Delay Line address latch]: dcore.sdl
                see data_du
        cu clock [two phase non-overlapping clock generator]
    pad frame
        north64 scpads3 [top group of 2u pads]
        south64 scpads3 [bottom group of 2u pads]
        east64 scpads3 [right group of 2u pads]
        west64 scpads3 [left group of 2u pads]

(sdl ~sun/delay/core ~sun/delay/core/cells)
(TimLager ~sun/delay/core/cells ~sun/Tim/dline ~sun/Tim/pgmux ~sun/Tim/latch
(magic ~sun/delay/chip ~sun/delay/frame/layout
~lager/LagerIII/lib/TimLager/scpads3 ~sun/delay/core/layout
~sun/delay/core/cells/layout ~sun/Tim/dline/leafcells ~sun/Tim/pgmux/leafcell
~sun/Tim/latch/leafcells ~sun/Tim/clock/leafcells)
```

```
Crossbar Switch
    core
        cbarray16 cbarray(parameter nodes) [array of interconnect cells]
        shiftreg16 shiftreg(parameter nodes) [shift register that transfers
                configuration bits onto the bit lines in array]
        pointer16 pointer(parameter nodes) [pointer that controls write
                lines in the array]
        ctrl [programming control unit, contains control logic and counter]
    pad frame [created before scpads library existence]
        in [buffered input pad]
        out [buffered output pad]
        gnd [GND pad]
        vdd [Vdd pad]
        dummy [dummy pad]
        corner [cornerpiece for pad frame]

 (TimLager ~sun/Tim/cbarray ~sun/Tim/shiftreg ~sun/Tim/pointer)
 (magic ~sun/cb2lay/3u_lib ~lager/LagerIII/lib/TimLager/scpads3(may use))
```

```
Masked OR Crossbar Switch*
Crossbar Switch [highest level in hierarchy]: crossbar.sdl crossbar.parval
    CBcore cbcore [core circuitry]: cbcore.sdl
        array cbar3(parameter nodes) [array of interconnect cells]
        shiftreg shiftreg(parameter nodes) [shift register that writes
                configuration bits onto the bit lines in array]
        pointer pointer3(parameter nodes) [pointer that controls write
                lines in the array]
        pcu cbpcu [programming control unit]
            ctlog [logic that generates programming control signals]
            prcount pc(parameter width) [counter]
        clkgen clock [two phase non-overlapping clock generator]
    pad frame
        north64 scpads3 [top group of pads]
        south64 scpads3 [bottom group of 2u pads]
        east64 scpads3 [right group of 2u pads]
        west64 scpads3 [left group of 2u pads]

(sdl-parval ~sun/crossbar/chip ~sun/crossbar/core)
(TimLager ~sun/crossbar/core/CBcore ~lager/LagerIII/lib/TimLager/scpads3
~sun/Tim/cbar ~sun/Tim/shiftreg ~sun/Tim/pointer ~sun/Tim/cbpcu ~sun/Tim/cloc
(magic ~sun/crossbar/chip/layout ~sun/crossbar/core/layout
~sun/Tim/cbar/leafcells ~sun/Tim/shiftreg/leafcells ~sun/Tim/pointer/leafcell
~sun/Tim/cbpcu/leafcells ~sun/Tim/clock/leafcells
~lager/LagerIII/lib/TimLager/scpads3/leafcells)

* the .c routines are used with TimLager3
```

Appendix B

Tap Switch Design Example

In Chapter 4, a hierarchical design approach applying macrocell generation techniques was discussed. The Tap Switch design provides a good example of the design process. This appendix contains a tutorial on generating the Tap layout with the LAGERIII tools. These tools construct the layout given a high level architectural description. this section illustrates how the architecture is described, and it specifies the particular LAGERIII tool employed at each stage in the layout design.

The tutorial was prepared in July, 1987 as part of a comprehensive document on the LAGERIII system.

1. TAP

The Tap Switch — J. Sun

A. Introduction

A computer network has node stations and the channel which physically links the nodes. The Tap Switch emulates the interface between the node and the tansmission medium, simulating how the nodes access or "tap" the physical channel. At the interface, the node can transmit or receive data, and data can be transmitted to or received from the channel. This is shown in the high level view of the Tap Switch in Fig. A.1. To connect the node and the channel, each of the Tap output port signals -- nrcv, p1xt, p2xt -- can be set to 0, 1 or some collection of the input port signals -- nxmt, p1rv, p2rv -- ORed together. A particular collection is the tap configuration, and it is specified with a 9 bit configuration address. The signals are 3 bits wide and are individually known as data, carrier, and code violation.

Since the three bits propogate between nodes in parallel, the node "taps" into the physical channel at the interface with identical tap configurations for all three signals. This means that the Tap Switch interface can be composed of three identical structures, one for each signal in the 3-bit wide data stream. Each structure is also referred to as a tap switch and is implemented through a PLA circuit. All the PLAs contain the same logic. In addition to the tap switches, a 9 bit latch stores the configuration address which controls the tap switches. The "ld" signal resets the latch for loading a new address.

Generating the chip layout with the aid of automatic layout tools uses a hierarchical layout design for organization, tractability and to introduce flexibility. Among the LAGER tools, the Tap Switch layout process applies **TimLager, Flint** and **Padroute. Design Manager (DM)** coordinates the flow of layout generation to realize the chip architecture throughout the hierarchy.

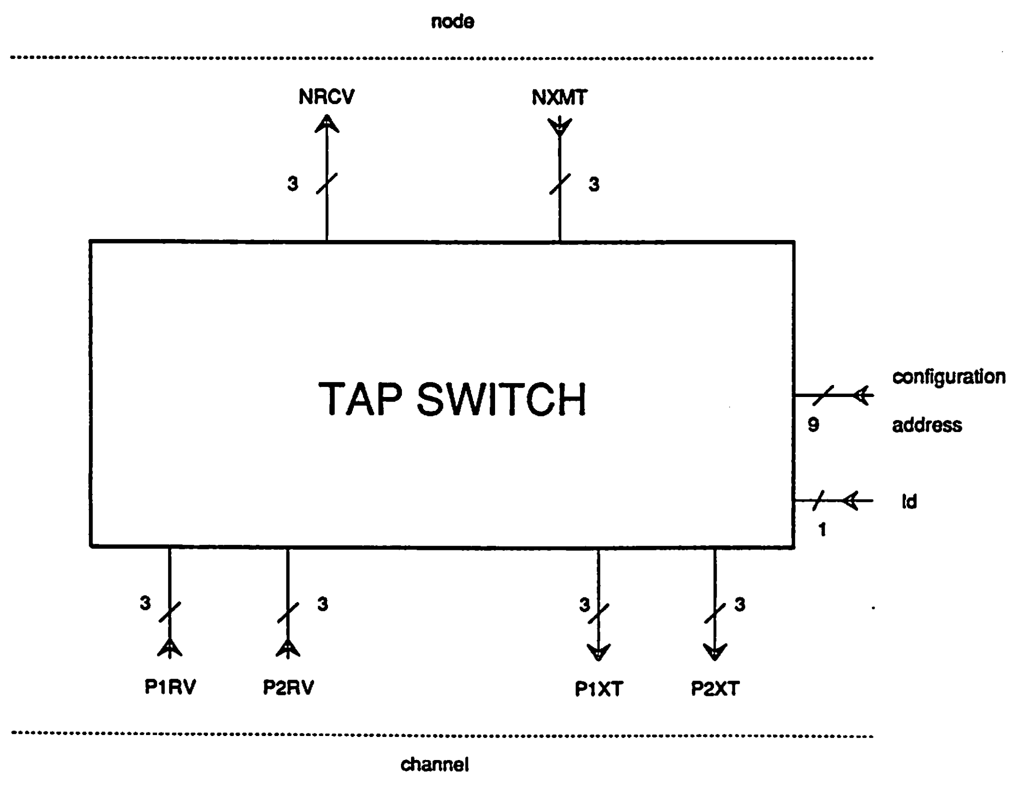


Fig. A.1

B. Describing the Chip Architecture Using *sdl*

A description of the chip architecture precedes the generation of any layout.

Using the hierarchical design approach for the Tap, the chip architecture is split into three levels. Each level is viewed as a macrocell, or parent-cell, composed of interconnecting sub-cells. Smaller cells can make up the sub-cell, qualifying it as a macrocell also. The highest level is known as the root level. As the hierarchy propagates down, the architecture is further contained in the second and third levels. This section illustrates how the chip architecture is hierarchically described using *sdl* (structural design language). At each level, the *sdl* description contains a list of the sub-cells, a net-list specifying connectivity between sub-cell terminals, and the layout generator that physically creates the macrocell. This desciption also includes parameter information that is passed onto Tim-Lager.

For clarity, Fig. B.0 shows the hierarchical breakdown of the chip design and the *sdl* descriptions which complete the architectural description. All the *sdl* files mentioned in this example appear at the end of partB. The *parameter* file is appended to partC.

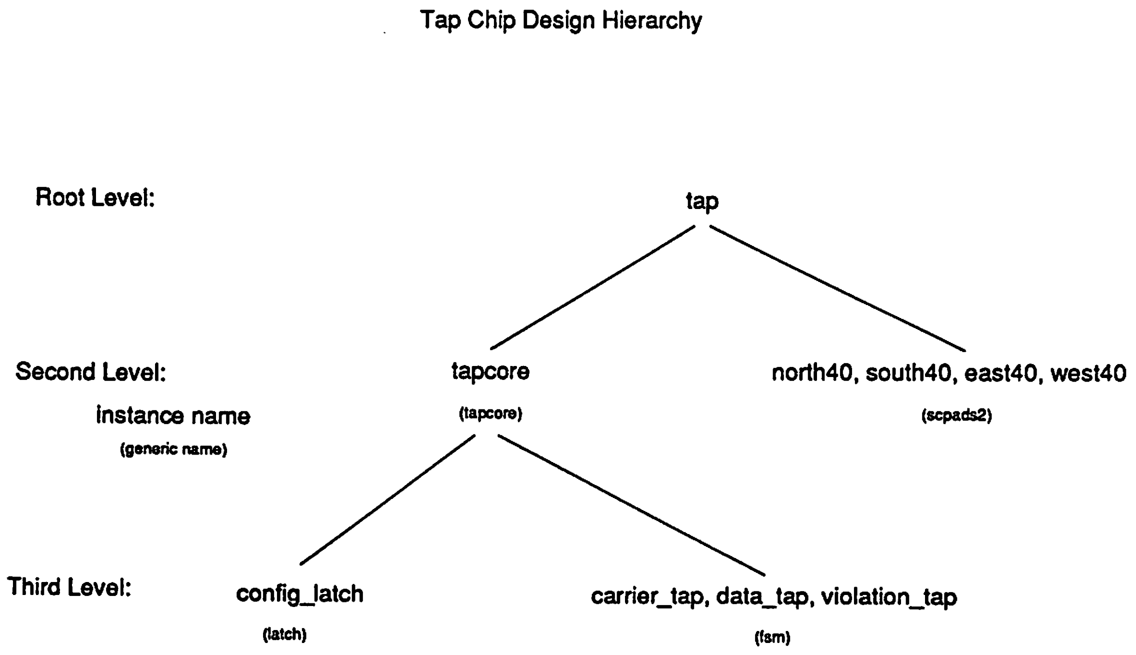


Fig. B.0

sdl files describing chip architecture:	tap.sdl scpads2.sdl tapcore.sdl latch.sdl fsm.sdl
parameter file:	tap.parval

Root Level-

At the root level, the Tap Switch is basically seen as four pad groups, which buffer the the chip input and output signals, and the internal core circuitry which is actually the functional part of the chip. So, a set of five subcells form the parent, known as tap, at the top level of the architecture. The subcells are shown in the Root Level Floor Plan of Fig. B.1, and listed in the table of Root Level Subcells in Table B.1. At the root level, the applied layout generator is **Padroute**, which assembles the pad frame from the four groups, places the core and connects corresponding signals between the pads and the core. With **Padroute**, each subcell has a special parameter called "fplan," and its assigned value is shown in the fplan values Table B.2.

Using the floor plan and the tables, the root level architecture is transformed into its *sdl* description as shown in the tap.sdl file. Since this is the root level description, all parameter names appearing throughout all hierarchical levels are listed under parent-cell. These parameter names are grouped under their proper subcells as shown in the sub-cells section of the *sdl* description. In this section, the parameter names are also matched with the corresponding generic parameter name. Instances of generic subcells have their names assigned in this section also. Parameter values are specified with the *sdl* description, or in a separate *parameters* list file, or interactively when running **Design Manager.** Besides the subcells, *sdl* also describes the signal connections between the I/O pads and the core as shown in the netlist section.

Root Level Floor Plan

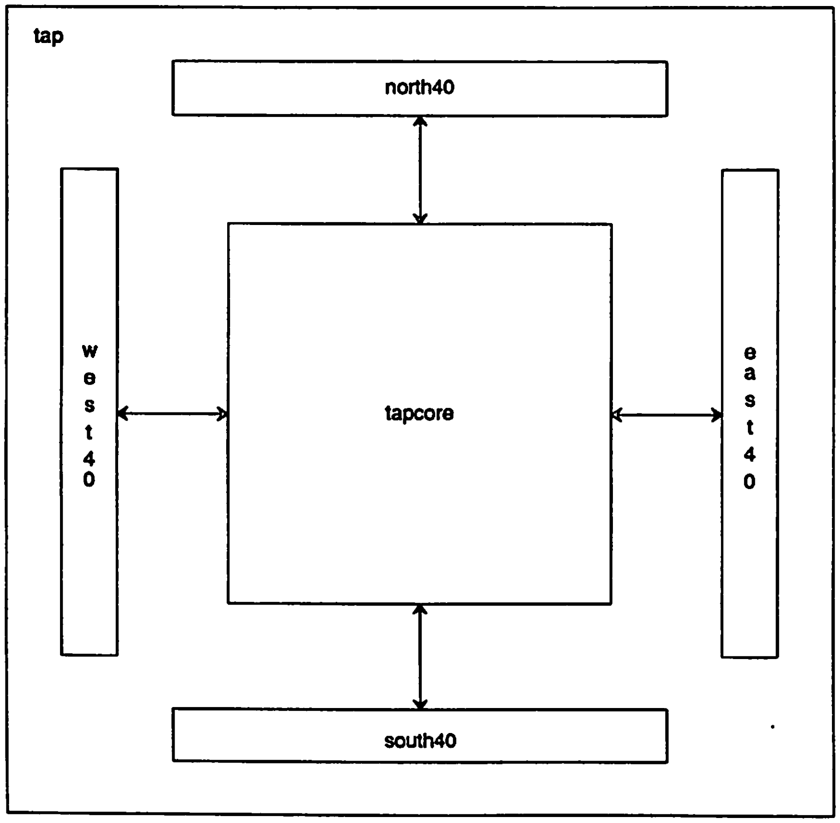


Fig. B.1

Root Level Subcells	
generic-name	instance-name
tapcore	tapcore
scpads2	north40
scpads2	south40
scpads2	east40
scpads2	west40

Table B.1

fplan values	
subcell	fplan value
core	'middle
top pad group	'top
bottom pad group	'bottom
left pad group	'left
right pad group	'right

Table B.2

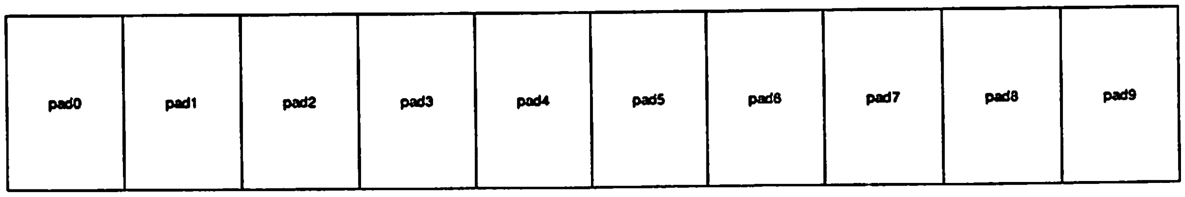
Second Level-

By expanding the subcells in the parent, tap, we reach the next level below root within the chip architecture. At this second level of hierarchy, the four pad groups and tapcore are considered as parents(they were subcells at the root level).

The pad groups are instances of the generic parent, scpads2, and there is a *sdl* description for it also. Scpads2 is composed of 10 individual pads as shown in Fig. B.2, and the pads are abutted using the layout generator **TimLager**. Since TimLager deals with leafcells, there are really no subcells within the parent cell scpads2, and there are no more levels of hierarchy under this parent. Thus, the *sdl* description for the pad groups contain only the parent cell data with the list of parameter names. The *sdl* description is shown in the scpads2.sdl file, and the assigned parameter values appear in the tap.parval file at the end of partC.

Scpads Pad Group Plan

(in second level of design hierarchy)



parameters:

number = number of outputs

pad0 = type of pad in position 0

.

.

.

pad9 = type of pad in position 9

Fig. B.2

Also in the second level of design hierarchy, the parent tapcore consists of four subcells which are three tap switches and a configuration latch. The tap switches direct and control data flow. They are basically three identical instances of a generic PLA circuit under the name of "fsm." The latch stores the 9-bit configuration address and is an instance of a general n-bit latch by the name of "latch." The Tapcore Floor Plan and the accompanying list of subcells is shown in Fig. B.3 and Table B.3. With this floor plan and subcell list, the tapcore architecture is described with *sdl* as shown in the tapcore.sdl file. Layout generation at this level applies **Flint**, which interactively places the subcells, routes signals between subcells and connects internal signals to the boundary of the parent cell tapcore. Using Fig.s B.4 and B.5 which illustrate the internal structure of the subcells, fsm and latch, the signal connection information that **Flint** needs to perform routing is described in the netlist. As in the root level *sdl* description, all parameters used in the chip architecture extending from this level to the level below are listed under the parent cell section. These parameters are grouped under their respective subcell in the subcell section of the *sdl* description.

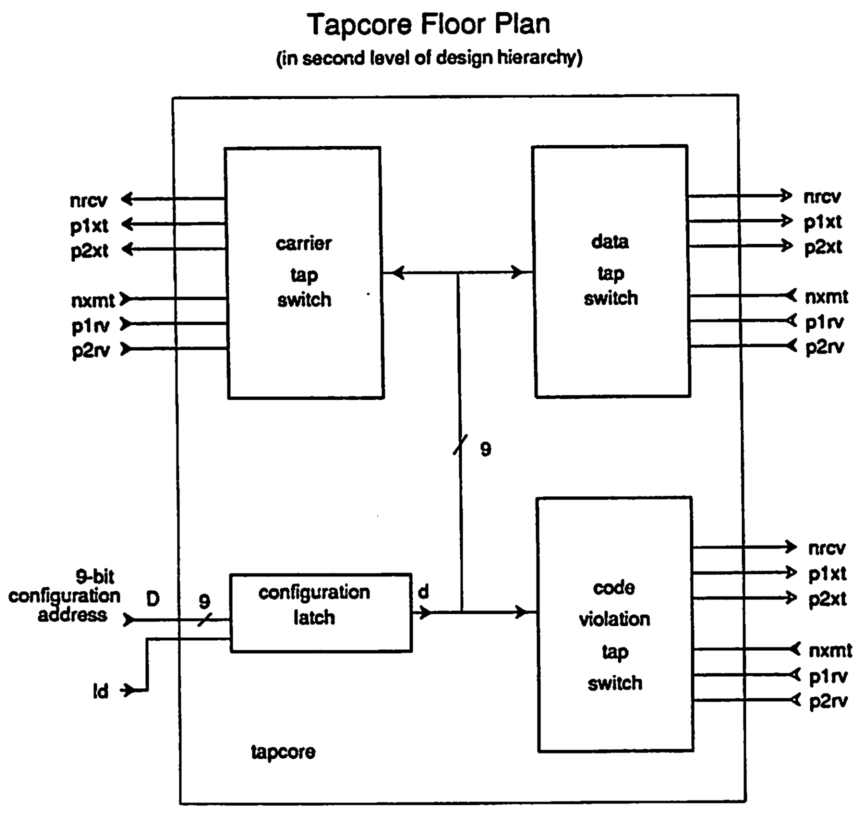


Fig. B.3

Tapcore Subcells	
generic-name	instance-name
fsm	carrier_tap
fsm	data_tap
fsm	violation_tap
latch	config_latch

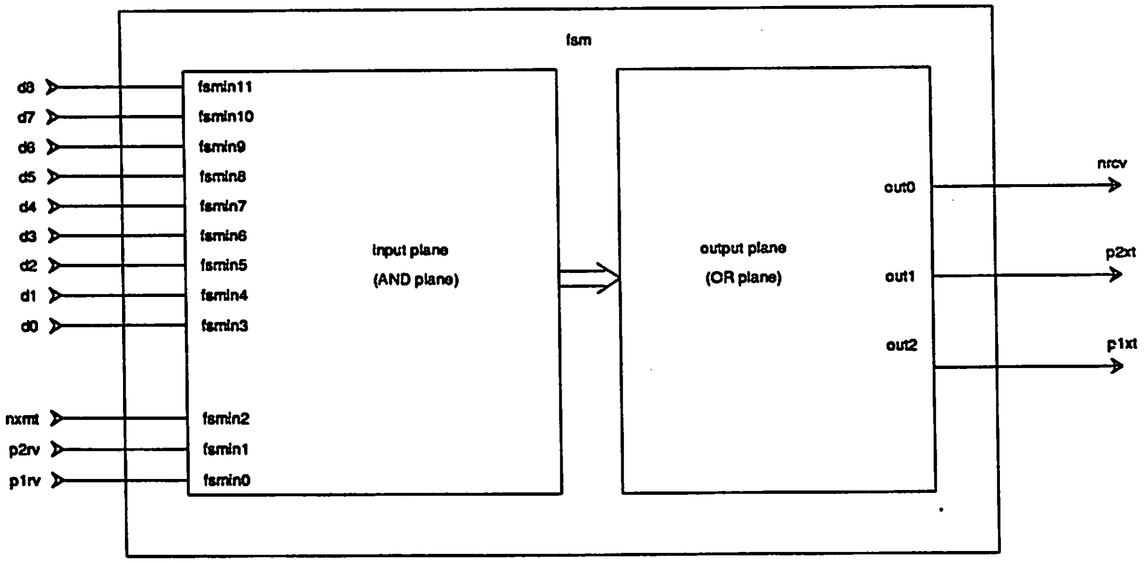
Table B.3

Third Level-

At the second level, fsm and latch were considered as subcells within tapcore. At the third and lowest level in the design hierarchy, fsm and latch are treated as parent cells. Since there are two parents -- latch and fsm, of which three instances occur, there is accordingly two *sdl* descriptions -- latch.sdl and fsm.sdl. The 9-bit configuration address latch and the tap switch PLAs are both generated through application of **TimLager**, and are illustrated in Fig.s B.4 and B.5. Because there are no more levels of hierarchy under these parents, it suffices to just specify the parent cell information and parameter names as the *sdl* description. These two *sdl* files complete the *sdl* description of the entire Tap Switch architecture.

For **TimLager** to generate the address latch and the PLA, the number of bits stored in the latch and the contents of the PLA(logic 1s and 0s) are treated as parameters. The parameter values are shown in the tap.parval file.

Tap Switch FSM(PLA) Plan



parameters:

in = number of inputs

out = number of outputs

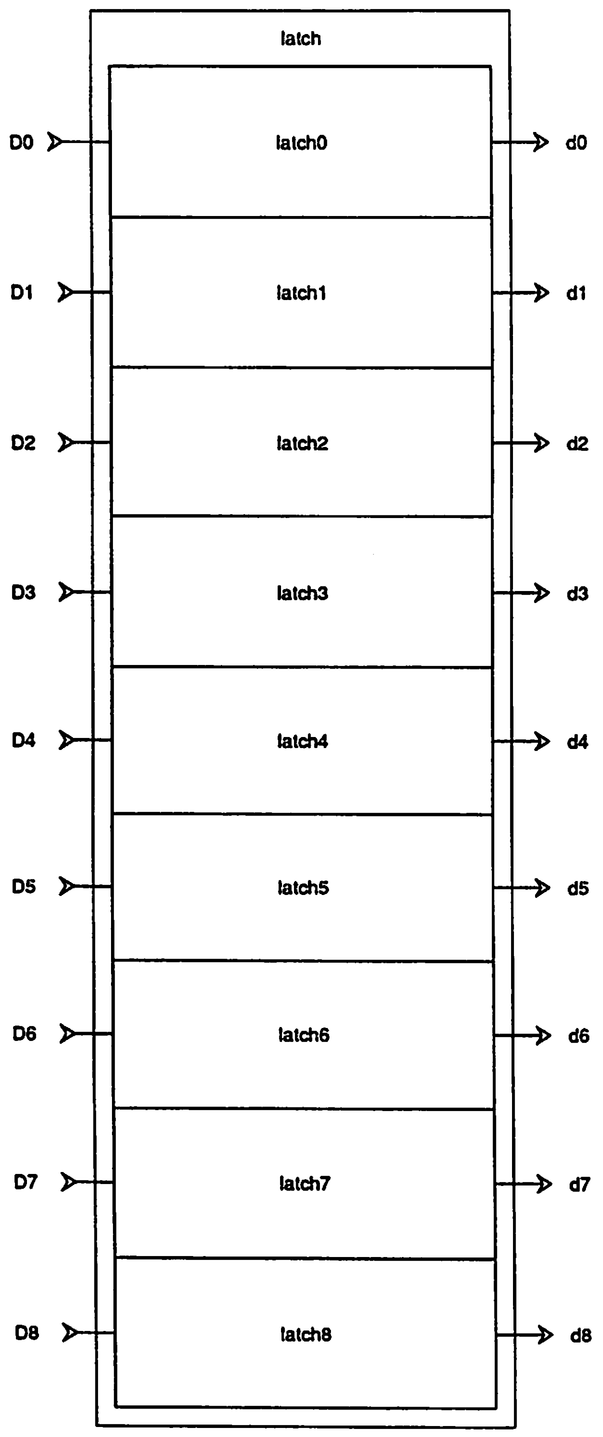
minterm = number of minterms

input-plane = contents of input plane

output-plane = contents of output plane

Fig. B.4

9-Bit Latch Plan



parameters:

nbits = # of stored bits

Fig. B.5

```
(layout-generator Padroute)
(parent-cell tap
  (parameters in out minterm nbits output-plane input-plane
  east-pad0 east-pad1 east-pad2 east-pad3  east-pad4  east-pad5  east-pad6
  east-pad7  east-pad8  east-pad9 east-number
  west-pad0 west-pad1 west-pad2 west-pad3  west-pad4  west-pad5  west-pad6
  west-pad7  west-pad8  west-pad9 west-number
  south-pad0 south-pad1 south-pad2 south-pad3  south-pad4  south-pad5
  south-pad6 south-pad7  south-pad8  south-pad9 south-number
  north-pad0 north-pad1 north-pad2 north-pad3  north-pad4  north-pad5
  north-pad6 north-pad7  north-pad8  north-pad9 north-number)
)
(sub-cells
  (tapcore tapcore
   (parameters (fplan 'middle) (in in) (out out) (minterm minterm) (nbits nbits)
   (output-plane output-plane) (input-plane input-plane)))
  (scpads2 north40 (parameters  (fplan 'top) (number north-number)
      (pad0 north-pad0) (pad1 north-pad1) (pad2 north-pad2) (pad3 north-pad3)
      (pad4 north-pad4) (pad5 north-pad5) (pad6 north-pad6) (pad7 north-pad7)
      (pad8 north-pad8) (pad9 north-pad9)))
  (scpads2 east40 (parameters  (fplan 'right) (number east-number)
      (pad0 east-pad0) (pad1 east-pad1) (pad2 east-pad2) (pad3 east-pad3)
      (pad4 east-pad4) (pad5 east-pad5) (pad6 east-pad6) (pad7 east-pad7)
      (pad8 east-pad8) (pad9 east-pad9)))
  (scpads2 west40 (parameters  (fplan 'left) (number west-number)
      (pad0 west-pad0) (pad1 west-pad1) (pad2 west-pad2) (pad3 west-pad3)
      (pad4 west-pad4) (pad5 west-pad5) (pad6 west-pad6) (pad7 west-pad7)
      (pad8 west-pad8) (pad9 west-pad9)))
  (scpads2 south40 (parameters  (fplan 'bottom) (number south-number)
      (pad0 south-pad0) (pad1 south-pad1) (pad2 south-pad2) (pad3 south-pad3)
      (pad4 south-pad4) (pad5 south-pad5) (pad6 south-pad6) (pad7 south-pad7)
      (pad8 south-pad8) (pad9 south-pad9)))
)
;
;------tap chip pads to core connectivity specifications----------
;
; nbits is implicitly assumed to be 9.
(net D0 ((tapcore D 0) (west40 D0)))
(net D1 ((tapcore D 1) (west40 D1)))
(net D2 ((tapcore D 2) (west40 D2)))
(net D3 ((tapcore D 3) (west40 D3)))
(net D4 ((tapcore D 4) (west40 D4)))
(net nxmt_c ((tapcore nxmt_c) (west40 nxmt_c)))
(net p2rv_c ((tapcore p2rv_c) (west40 p2rv_c)))
(net p1rv_c ((tapcore p1rv_c) (west40 p1rv_c)))

(net D5 ((tapcore D 5) (north40 D5)))
(net D6 ((tapcore D 6) (north40 D6)))
(net D7 ((tapcore D 7) (north40 D7)))
(net D8 ((tapcore D 8) (north40 D8)))
(net ld ((tapcore ld) (north40 ld)))
(net nxmt_v ((tapcore nxmt_v) (north40 nxmt_v)))
(net p2rv_v ((tapcore p2rv_v) (north40 p2rv_v)))
(net p1rv_v ((tapcore p1rv_v) (north40 p1rv_v)))
(net p1xt_v ((tapcore p1xt_v) (north40 p1xt_v)))
;
```

```
(net nrcv_d ((tapcore nrcv_d) (east40 nrcv_d)))
(net p2xt_d ((tapcore p2xt_d) (east40 p2xt_d)))
'net p1xt_d ((tapcore p1xt_d) (east40 p1xt_d)))
  et nrcv_v ((tapcore nrcv_v) (east40 nrcv_v)))
(net p2xt_v ((tapcore p2xt_v) (east40 p2xt_v)))
;
(net nxmt_d ((tapcore nxmt_d) (south40 nxmt_d)))
(net p2rv_d ((tapcore p2rv_d) (south40 p2rv_d )))
(net p1rv_d ((tapcore p1rv_d) (south40 p1rv_d )))
(net nrcv_c ((tapcore nrcv_c) (south40 nrcv_c)))
(net p2xt_c ((tapcore p2xt_c) (south40 p2xt_c)))
(net p1xt_c ((tapcore p1xt_c) (south40 p1xt_c)))
;
```

```
(layout-generator TimLager)
(parent-cell scpads2
(parameters
        number pad0 pad1 pad2 pad3 pad4 pad5 pad6 pad7 pad8 pad9)
(layout-generator Flint)
(parent-cell tapcore (parameters in out minterm
        nbits output-plane input-plane))
  ub-cells
        (fsm data_tap (parameters (in in) (out out)
                (minterm minterm) (output-plane output-plane)
                (input-plane input-plane)))
        (fsm carrier_tap (parameters (in in) (out out)
                (minterm minterm) (output-plane output-plane)
                (input-plane input-plane)))
        (fsm violation_tap (parameters (in in) (out out)
                (minterm minterm) (output-plane output-plane)
                (input-plane input-plane)))
        (latch config_latch (parameters (nbits nbits)))
)
;
;------tapcore internal and external connectivity specifications----------
;give net_name parameter connectivity:-parent signifies connection to
;                                          outside world.
;                                       -subcell signifies internal connection.
;
(net d nbits ((config_latch d) (data_tap fsmin 3) (carrier_tap fsmin 3)
(violation_tap fsmin 3)))
(net D nbits ((parent D) (config_latch D)))
(net ld ((parent ld) (config_latch ld)))
;
(net nxmt_d ((parent nxmt_d) (data_tap fsmin 2)))
(net p2rv_d ((parent p2rv_d) (data_tap fsmin 1)))
(net plrv_d ((parent plrv_d) (data_tap fsmin 0)))
(net nrcv_d ((parent nrcv_d) (data_tap out 0)))
  et p2xt_d ((parent p2xt_d) (data_tap out 1)))
 .et plxt_d ((parent plxt_d) (data_tap out 2)))
;
(net nxmt_c ((parent nxmt_c) (carrier_tap fsmin 2)))
(net p2rv_c ((parent p2rv_c) (carrier_tap fsmin 1)))
(net plrv_c ((parent plrv_c) (carrier_tap fsmin 0)))
(net nrcv_c ((parent nrcv_c) (carrier_tap out 0)))
(net p2xt_c ((parent p2xt_c) (carrier_tap out 1)))
(net plxt_c ((parent plxt_c) (carrier_tap out 2)))
;
(net nxmt_v ((parent nxmt_v) (violation_tap fsmin 2)))
(net p2rv_v ((parent p2rv_v) (violation_tap fsmin 1)))
(net plrv_v ((parent plrv_v) (violation_tap fsmin 0)))
(net nrcv_v ((parent nrcv_v) (violation_tap out 0)))
(net p2xt_v ((parent p2xt_v) (violation_tap out 1)))
(net plxt_v ((parent plxt_v) (violation_tap out 2)))
```

```
(layout-generator TimLager)
(parent-cell fsm (parameters in out minterm output-plane input-plane))
```

```
(layout-generator TimLager)
(parent-cell latch (parameter nbits))
```

C. Generating Layout With Design Manager

The *sdl* description embodies the Tap Switch architecture. DM essentially takes the *sdl* description and the parameter values to form a complete hierarchical description of the design and directs the Lager tools to generate the layout.

The *sdl* files are set up before invoking DM, however the assigned parameter values need not be specified before that. The parameter values are either given in the *sdl* files directly, or listed in a *parameter* file which DM is instructed to call, or entered interactively while in DM. For this Tap Switch example, the parameters are given with the second option above. All the values are specified as shown in the tap.parval file. The parameter names were explained in part B. To generate this layout, the flag options *c m l* were entered into DM to specify the scmos, magic and TimLager labelling options. Given the *sdl* descriptions and upon reading the assigned parameter values, DM can invoke **TimLager** to generate the layout for the 9-bit latch and the three PLAs at the lowest level of the design hierarchy. To realize tapcore, DM calls **Flint** and gives that layout generator the signal terminal information necessary for routing. Finally, after invoking **TimLager** to layout the four pad groups, DM calls **Padroute** to complete the chip layout. DM produces a *dm.log* and *lg.log* file where it dumps its diagnostic messages.

The use of parameters introduces flexibility into the Tap architecture. With the *sdl* descriptions set up, only the parameter values need to be modified to generate Tap Switches with different logic functions. Specifically, for the case of the Tap Switch, generating the layout for a new design using a different number of address bits and different PLA contents is conveniently handled by entering the appropriate new parameter values. The *sdl* description does not change except for those associated with scpads2 and **Padroute**. DM takes the new values and coordinates the same layout process as before with the Lager tools to create a new layout.

```
(in 12)
(out 3)
(minterm 20)
'nbits 9)
  utput-plane
;  array: nrcv p2xt p1xt
        ((array |100| )
         (array |010| )
         (array |001| )
         (array |010| )
         (array |010| )
         (array |001| )
         (array |001| )
         (array |100| )
         (array |010| )
         (array |001| )
         (array |010| )
         (array |001| )
         (array |100| )
         (array |100| )
         (array |010| )
         (array |001| )
         (array |100| )
         (array |100| )
         (array |010| )
         (array |001| ))
)
(input-plane
;  array: d8 d7 d6 d5 d4 d3 d2 d1 d0 nxmt p2rv p1rv
        ((array |001xxxxxxxx1| )
         (array |xxx010xxxx1x| )
         (array |xxxxxx010x1x| )
         (array |xxx100xxxxx1| )
         (array |xxxx11xxx1xx| )
         (array |xxxxxx100xx1| )
         (array |xxxxxxx111xx| `)
         (array |1x1xxxxxxx1x| )
         (array |xxx001xxxxx1| )
         (array |xxxxxx001xx1| )
         (array |xxx1x1xxxx1x| )
         (array |xxxxxx1x1x1x| )
         (array |111xxxxxxxxx| )
         (array |x10xxxxxxx1x| )
         (array |xxx111xxxxxx| )
         (array |xxxxxx111xxx| )
         (array |1x0xxxxxxxx1| )
         (array |1xxxxxxxx1xx| )
         (array |xxx10xxxx1xx| )
         (array |xxxxxx10x1xx| ))
         )
         (east-number (10))
         (east-pad0 (out p2xt_v ))
         (east-pad1 (out nrcv_v ))
         (east-pad2 (Vdd))
         (east-pad3 (in phi1 phi1*))
         (east-pad4 (dummy))
        (east-pad5 (in phi2 phi2*))
        (east-pad6 (dummy))
        (east-pad7 (out nrcv_d ))
        (east-pad8 (out p2xt_d ))
        (east-pad9 (out p1xt_d ))
        (west-number (10))
        (west-pad0 (in nxmt_c nxmt_c*))
        (west-pad1 (in p2rv_c p2rv_c*))
        (west-pad2 (in p1rv_c p1rv_c*))
        (west-pad3 (dummy))
        (west-pad4 (dummy))
        (west-pad5 (in D0 D0*))
        (west-pad6 (in D1 D1*))
        (west-pad7 (in D2 D2*))
        (west-pad8 (in D3 D3*))
        (west-pad9 (in D4 D4*))
        (north-number (10))
        (north-pad0 (in D5 D5*))
        (north-pad1 (in D6 D6*))
        (north-pad2 (in D7 D7*))
        (north-pad3 (in D8 D8*))
        (north-pad4 (in ld ld*))
        (north-pad5 (GND))
        (north-pad6 (in p1rv_v p1rv_v*))
        (north-pad7 (in p2rv_v p2rv_v*))
        (north-pad8 (in nxmt_v nxmt_v*))
        (north-pad9 (out p1xt_v ))
        (south-number (10))
        (south-pad0 (in nxmt_d nxmt_d*))
        (south-pad1 (in p2rv_d p2rv_d*))
        (south-pad2 (in p1rv_d p1rv_d*))
        (south-pad3 (Vdd))
        (south-pad4 (dummy))
        (south-pad5 (dummy))
        (south-pad6 (GND))
        (south-pad7 (out nrcv_c ))
        (south-pad8 (out p2xt_c ))
        (south-pad9 (out p1xt_c ))
```