Fast Arbitration in Dilated Routers

by

Matthew E. Becker

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering

at the Massachusetts Institute of Technology

May 1996

© Massachusetts Institute of Technology 1996

Author_____
Department of Electrical Engineering and Computer Science
May, 1995

Certified by_____
Thomas F. Knight, Jr.
Thesis Supervisor

Accepted by_____
Professor Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

AUG 10 1995

LIBRARIES

Fast Arbitration in Dilated Routers

by

Matthew E. Becker

Submitted to the
Department of Electrical Engineering and Computer Science

May, 1995

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering

Abstract

In multibutterfly networks the primary limitation on performance is the speed of the dilated routing component. This thesis studies a technique, dynamic dilation, which reduces latency in dilated routing components without greatly affecting flexibility. Initially in order to achieve lower latency we limit dilation flexibility, extract parallelism from the arbitration process and pipeline the process into two cycles [1].

For implementation reasons the allocation cycle is required to have some unused latency. Dynamic dilation takes advantage of this latency to allow the dilation of a component to be reconfigured each cycle based on incoming messages. This effectively recovers the flexibility lost by limiting dilation, while retaining extremely low latency.

Further we utilize a modified version of conventional domino style logic. This new type of logic decreases critical path latency by a factor of two.

Thesis Supervisor: Thomas F. Knight, Jr.
Title: Principal Research Scientist, MIT AI Lab

Acknowledgments

Thanks to Tom Simon, Fred Chong, Mike Bolotski, André DeHon, Raj Amirtharajah and Weip Chen for their support, helpful ideas, and friendship. Special thanks to Tom Knight for being there to listen to my ideas, shape my understanding of the material and provide guidance.

Thanks to my parents for their patience, understanding and support.

This research is supported in part by the Advanced Research Projects Agency under contracts N00014-91-J-1698 and N00014-91-J-4038.

Contents

List of Figures

Chapter 1

Introduction

All massively parallel computers need some kind of interconnection network to allow communication between processors. For a large scale system with n processors, it would be impossible to connect each processor to every other processor with a wire. That would require n^2 number of wires. Similarly a single bus proves impractical as the number of processors increases, because of bandwidth problems. A network using routing components proves more effective at combatting these problems. This thesis optimizes one type of routing component, the dilated router, in order to improve overall network performance.

Basically, the routing component can connect any of its input ports to any of its output ports. To form a network, one could simply use a single component to connect up all the processors, one to each port. The number of wires becomes linear with the number of processors. However, a single routing component has i/o and reliability limitations. A better solution is to stage smaller components by connecting the output ports of the first stage to other components and then to processors as shown in Figure 2-1. This allows many more processors to be connected with components of a reasonable size. In order to provide fault-tolerance, multiple paths through the network must exist. This is effectively implemented with dilated routers through the use of a multibutterfly organization, which is described later.

Within these types of interconnection networks, the latency and bandwidth swiftly become an important issue because as distributed systems get larger and faster, the amount of network traffic increases dramatically. Unless network latency and bandwidth keep pace, overall system performance degrades. However, latency cannot be reduced and bandwidth increased, at the

expense of too much flexibility. Examples of desirable flexibility include randomized path selection for fault tolerance or variable dilation for network reconfiguration.

In the case of dilated circuit-switched crossbars, my results from optimizing between latency, bandwidth and flexibility have been promising. The central compromise of flexibility is a limitation on dilation during a single cycle of allocation. This makes path allocation faster, and allows the switch to operate at speeds close to 400 MHz. These results and any that follow come from simulations of a 0.8 micron, 3-metal CMOS process under nominal process corners.

Chapter 2 of this paper refines the concept of dilated routers and their importance. Chapter 3 outlines the different choices of dilation style and how each style affects latency and flexibility. In Chapter 4, I describe the basic logic form used to improve performance by a factor of two over conventional forms. Finally the topics presented throughout the thesis are reviewed in Chapter 5.

Chapter 2

Routing Components

The dilated routing component is described in Section 1. In section 2 some benefits of using dilation in interconnection networks are outlined. Section 3 provides a brief comparison of a butterfly style network with and without dilation. Finally in section 4 I explain the basic structure of a routing component which is fleshed out in later chapters and the appendices.

2.1 Dilation in Routing Components

A dilated routing switch is a crossbar routing component used in multistage interconnection networks with the important feature of allowing multiple simultaneous messages to be routed in each logical output direction. An n input switch, for example, having $n/2$ outputs in each of two logical direction, can route any of the n input messages intended for either logical direction to any of the available $n/2$ physical outputs corresponding to that direction. This switch is said to have a dilation of $n/2$. Normal crossbar components thus have a dilation of 1.

Dilation refers to the number of ports on a particular component which a message can be routed through to get to the *same* place. It is important to realize that the *same* place could either be a single end point or it could be a set of multiple endpoints which provide the same functionality. An example of multiple endpoints being equivalent logically would be a set of endpoints that provide i/o to the outside world. If a message is willing to pass through any i/o port, then all the i/o endpoints would be the *same*. This distinction of *same* is relevant

to the issue of dilation flexibility, because *sameness* can change from message to message.

The use of dilated routing switches is a very effective technique for improving performance of multistage interconnection switches for three important reasons:

- They improve the probability of successful message routing through the switch.
- They allow the construction of multibutterfly networks which further improve statistics and avoid hot spot contention.
- They enable high reliability networks with inherent fault tolerance through source-responsible routing.

2.2 Dilation Benefits

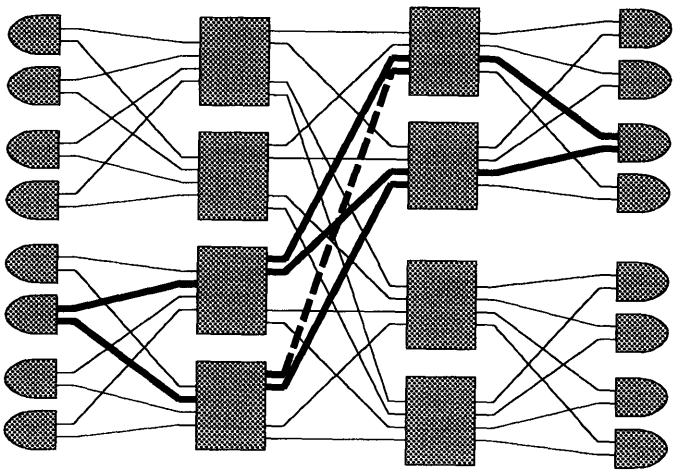
2.2.1 Routing Statistics

Details of the statistical improvements in message routing are more fully analyzed in [2], but the key insight can be given here. Imagine a switch with n inputs, where each input is attempting to route to either of $n/2$ outputs. As long as there are less than $n/2$ inputs wanting to route in a given direction, all of the messages will be successfully routed. If more than $n/2$ so desire to route, then any in excess of $n/2$ will fail to route.

Consider now the statistics of the router inputs. It is extremely unlikely that all inputs together wish to route in only one direction. The most probable set of input routings desired is an even split, where half of the inputs route one way while the rest route the other way. The distribution, in fact, is a binomial distribution, with mean $n/2$ and standard deviation $\sqrt{n/2}$. Statistically, almost all messages are successfully routed through such a component, for large n.

Unlike a dilation 1 crossbar component, where the statistics for blocking impose an expected output routing probability of $1 - 1/e$ for a fully loaded set of inputs, a large dilation router can successfully route almost all messages.

The dilation we can use in real routers is limited by the available pin count, and by the necessity, in the final stages of the multistage network, to eventually route to a single final destination or a small set of final destinations – requiring a low dilation router in the final



An 8x8 multibutterfly network with a stage of dilation two routers and a stage of dilation one routers. In bold are the equivalent paths from the sixth endpoint to the third endpoint. The dotted line is a blocked connection.

Figure 2-1: Multibutterfly with path redundancy

stage.

Statistics further improve dramatically, when compared to other types of networks, for loading even slightly below full loading.

2.2.2 Multibutterfly Techniques

A significant problem with conventional multistage interconnection networks is the hot spot behavior induced by relatively simple, regular, interconnect patterns such as transpose. In the worst case, the straightforward butterfly network allows only a single message to successfully route due to contention within the switch.

Dilation partially solves this hot spot problem, but the use of the *multibutterfly* interconnection techniques, see Figure 2-1, essentially eliminates hot spot behavior within the switch. To develop the multibutterfly, we note that in the first stage of an interconnection switch, it makes no difference how we order the outputs which are routed in a logical direction. All of these outputs are logically equivalent. By forming the wiring between the first and second router stages as an expander, we can assure that there are no worst case routing problems. The construction of such multibutterfly networks depends on our ability to construct dilated

routers.

2.2.3 Fault Tolerance

The availability of multiple router outputs in a single logical direction provides the key idea in construction of wiring and router fault tolerant interconnection networks [3] [4].

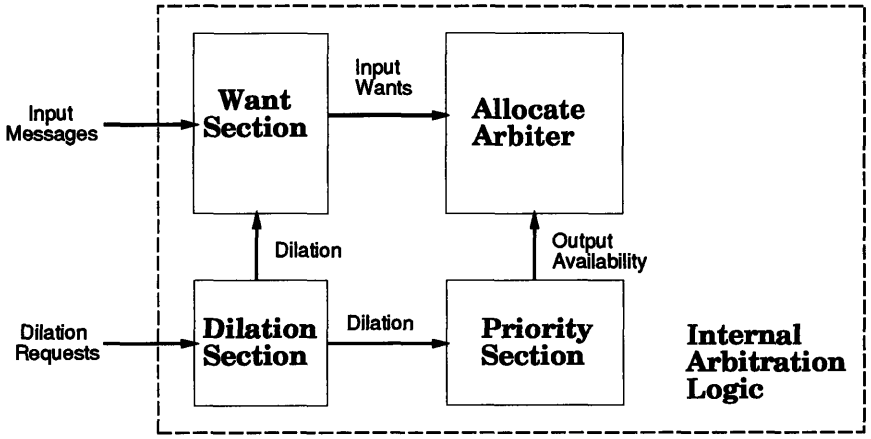
If a particular wire or routing component fails, there is always an alternative path which avoids the wire or components, but arrives at the same destination. By randomly choosing the physical output out of available channels in the specified logical direction, we will route a message on a second attempt through a different path with high probability. Again, the possibility of performing this rerouting relies on the dilation of the switch component [5].

2.3 Comparison of Network with and without Dilation

Some designers achieve a portion of the gain associated with dilation through the use of one or more additional stages of routing at the input to a multistage switch. By randomly choosing the path through these extra switch stages, some of the properties associated with dilation can be achieved, primarily a degree of fault tolerance and hot spot avoidance [6]. The difficulty with such extra stage approaches is that the extra stage route is usually chosen prior to message routing, providing no opportunity to avoid busy ports in the switch components. By making the decision of physical output port dynamically during the route setup process, we can route around otherwise occupied or faulty output ports in the switch.

In a dilated switch, then, a portion of the path selection is specified by the logical route carried with the message, and the details of the physical route are decided with a combination of random choice and busy path avoidance in each of the switch components.

The delay associated with route setup in such dilated switch components is an important contributor to end to end latency of interprocessor and cache miss traffic in parallel computer networks.



The *want* section interprets the input allocation requests. The *priority* section stores back port availability. The *dilation* section arbitrates between different dilation request in dynamic dilation. The *allocate arbiter* makes the decision on new allocates.

Figure 2-2: Internal format for allocator

2.4 Component Format

Before we begin describing the different arbitration techniques in detail, we need to discuss the format of a component as seen in Figure 2-2. It is made up of four sections:

- The dilation section is nonexistent in general and fixed dilation, but in dynamic dilation it arbitrates between input port request for different dilations as described in Appendix A.
- The want section interprets allocation requests made by the input ports. It provides information to the allocate arbiter about whether each input port would like to be connected with each output port. It is described in Appendix B.
- The priority section, described in Appendix C, stores the availability of the output ports. It updates and passes this information to the allocate arbiter each cycle.
- The allocate arbiter uses the input requests and the output availability to decide on the allocation of new connections. This information combined with previously allocated connections is used for the passing of data through the appropriate ports of the component.

It is described in Appendix D.

- The clock section creates the necesary clocking signals for the new domino-like logic style which is described later. The section is described in Appendix E.
- The test section, briefly described in Appendix F, provides the user interface. One can load in a set of test vectors, allow the internal circuits to evaluate at full speed, and then load out the result for comparison with expected values.

Chapter 3

Dilation Strategies

I consider three types of dilated components: general dilation, fixed dilation and dynamic dilation. General dilation allows each input port to specify its own dilation on any given cycle. This translates into network flexibility, but entails a large latency cost. Fixed dilation is the most conservative dilation technique with a single dilation being statically set for all ports. This technique significantly reduces the complexity and hence latency of arbitration. Because of implementation requirements, the fastest version of fixed dilation can be generalized to dynamic dilation without any penalty in latency. In dynamic dilation, the dilation is configured on each cycle for the entire component based on incoming messages. The degree of flexibility for dynamic dilation is the same as the flexibility of general dilation for the most common case.

The importance of dilation flexibility is outlined in Section 1. In the following two section, 2 and 3, the properties and implementations of general dilation and then fixed dilation are described. In section 4 I introduce dynamic dilation and compare its flexibility and latency to the other dilation strategies.

3.1 Dilation Flexibility

General and dynamic dilation allow messages to specify dilation, which may be useful for:

- Fault Tolerance - checking any path through the network

- Load Balancing - spawning messages to random places
- Research Tool - reconfiguring the network
- Separating Multiple Users - dividing the network
- Explicit Traffic Control - statically controlling traffic patterns
- Improved Routing Statistics - described in Chapter 2

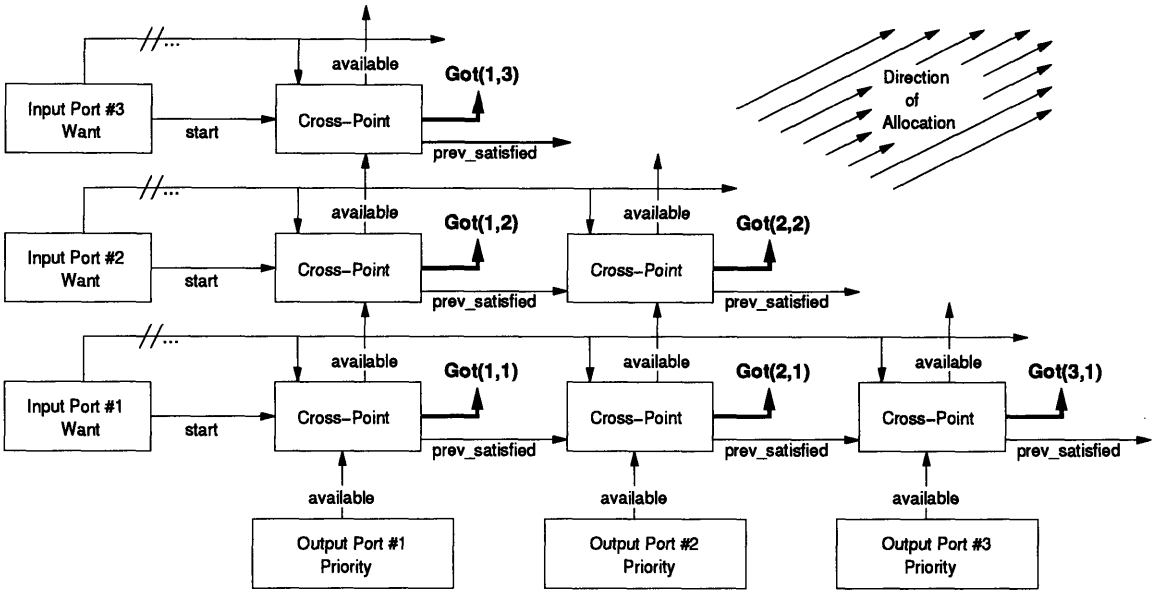
In terms of fault tolerance a message can be forced to route through a specific path by having the message set the dilation to one at each stage. In this way a faulty part of the network could be located at run time. It can similarly be used as a start-up test to check the entire network very quickly.

In deterministic dilation the initial processor specifies a set of locations for a spawned process, which could be in a highly used area of the network. If instead the initial processor could set the dilation as wide as possible, then the spawning process would naturally tend to flow away from highly congested areas. This is an extremely simple method of load balancing.

As a research tool, networks with flexible dilation can be used to test and evaluate many different types of network configurations. All the messages can be setup to request the appropriate dilation at each stage of the network.

In most systems it would be useful to insure that certain users do not affect each other. Dividing a system usually entails separating only processors, with one common network. In order to truly separate the users, the network must also be divided. A simple method of doing this is to give each message access to only a certain set of paths by limiting its dilation. For example specifying a dilation of two would divide the network in half and then one user could use the upper half while the second could only use the lower half.

Run-time flexibility also allows explicit control of path selection. This could be used when a particular application requires a specific traffic pattern to operate efficiently. The application designer could use the control to specify the particular pattern needed.



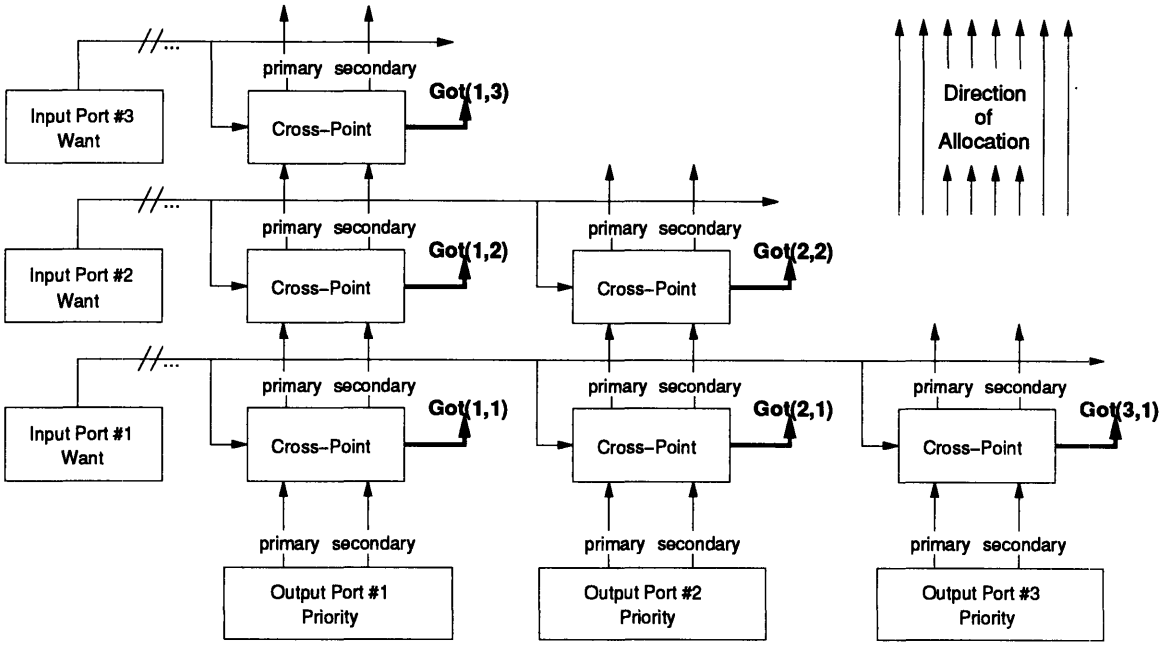
The *want* signals tell each cross-point if its input port wants to be connected to its output port. Intermediary signals, *prev_satisfied* and *available*, contain information on allocates to higher priority cross-points, which prevents multiple allocations to the same port.

Figure 3-1: Section of a general dilation allocator

3.2 General Dilation: Wave-style arbitration

In general dilation a message on any input port may request any dilation and any set of output ports, which maximizes the probability that it will be connected. The availability of an output port is based on whether the port has been allocated on a previous allocate cycle *and* on whether the output port is allocated to another input port during the same cycle. Similarly, a message's allocate request for an output port depends not only on the original request, but also on whether the request is satisfied by another output port. Consequently the decision about making a connection between a particular input-output port pair requires information about the decision of ports being allocated in the same cycle. This makes allocation in general dilation a sequential process requiring time linear with the total number of ports.

This sequential process is most easily represented by a two-dimensional array of cross-points as shown in Figure 3-1. Each cross-point is associated with an input port on the y-axis and an output port on the x-axis. A cross-point is simply a set of logic that controls whether



The *want* signals tell each cross-point if its input port wants to be connected to its output port. The *primary* and *secondary* signals contain information on the priority of the output ports. When a secondary port passes a want signal it is promoted to a primary port. When a primary port passes a want signal it creates a connection. Unavailable ports have neither priority line asserted.

Figure 3-2: Section of wave-style allocator with a fixed dilation of two

its associated ports are connected. The allocation begins at the lower left cross-point and proceeds like a wave to the upper right cross-point. A connection is made at a cross-point when the input port wants to connect, the input port request is not previously satisfied, and the output port is available.

Due to the sequential nature of the process, the fastest general dilation implementation is a factor of five times slower than the fixed and dynamic dilation techniques. The latency for arbitration in general dilation is approximately 14 ns as included in Table 3.1. The general dilation techniques and performance figures are very similar to separate work described in [8].

3.3 Fixed Dilation

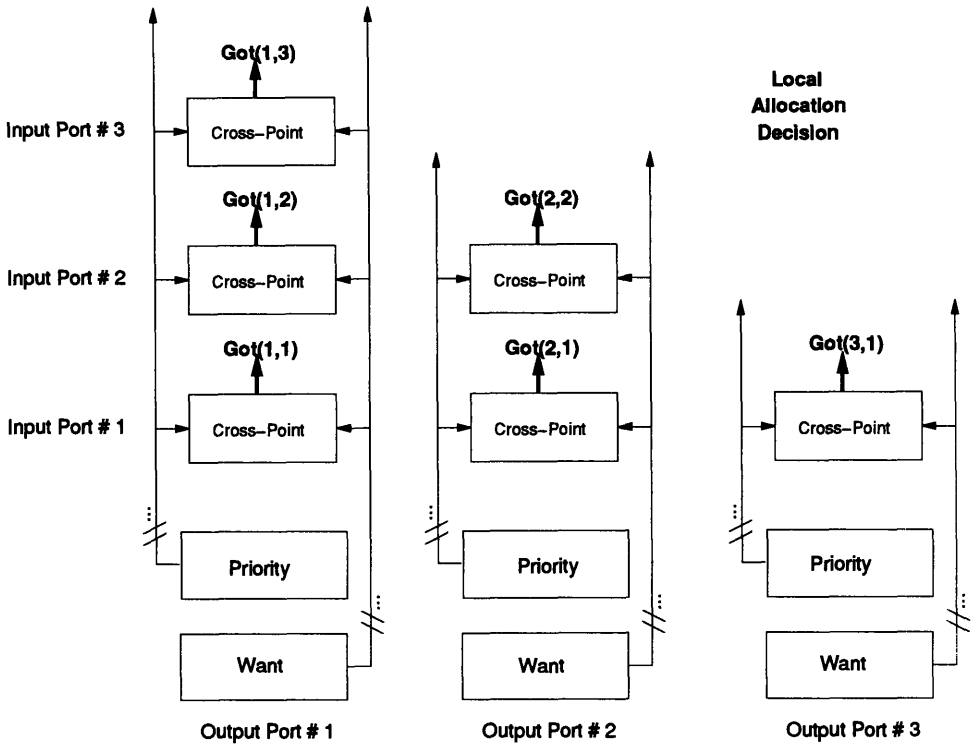
Fixed dilation schemes produce significantly lower allocation latency, because they restrict the distribution of want signals and the sequential effects of previous allocations. This results from the dilation being configured statically for all input ports. The order in which the output ports are allocated is statically set, which removes the need for the *prev_satisfied* signal of general dilation schemes. There are two styles of implementation: wave-style arbitration and parallel-style arbitration.

3.3.1 Wave-Style Arbitration

Similar to the general dilation implementations, fixed dilation wave-style arbitration proceeds like a wave as shown in Figure 3-2 [7]. However, the signaling propagates along only one axis and therefore passes through half the number of cross-points. At each cross-point an input-output port pair is allocated only if its associated output port is *primary* and the input port is requesting it. Every cross-point above the allocated one receives an unavailable signal on the priority lines. If the priority of an output port is not primary it is promoted to a higher priority whenever it passes a cross-point with a input port request. This method lends itself to a simple pass gate implementation with a latency of 5.2 ns which is included in Table 3.1.

3.3.2 Parallel-Style Arbitration

The second implementation of fixed dilation is fully parallelized as shown in Figure 3-3. As before, the allocation cycle begins with grouping and ordering, but this time of both input and output ports. The cross-point receives the randomized order in which its associated ports are to be allocated. The priorities at each cross point are compared and if they are equal, the pair is connected. For example if a cross-point has a second priority input port and the output port is the second to be allocated then a connection is made. Because the decision of each cross-point can be completed in parallel, the allocation process has extremely low latency. The fixed dilation scheme implemented with parallel techniques minimizes single cycle latency at 4.3 ns as shown in Table 3.1.



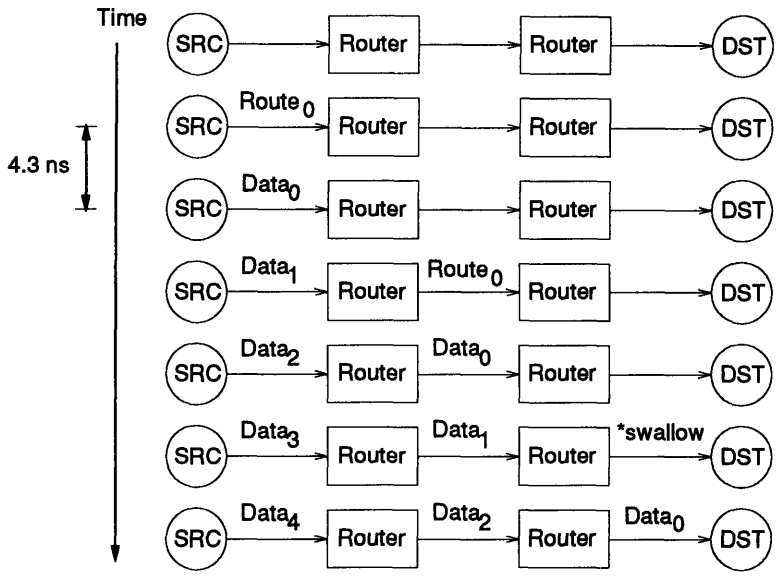
The ouput *priority* signals tell each cross-point which order the output ports will be allocated. The input *want* signals for a specific output port inform the cross-points of the input port ordering. The cross-point allocates when the output priority matches the input priority.

Figure 3-3: Section of parallel-style allocator

3.3.3 Message Format and Pipelining

Messages begin with routing words followed by data words. The routing words which set up new paths, require more processing than the data words. For this reason the allocation logic can be pipelined to increase network performance.

The routing words contain destination information, which allows a message to route through a stage of the network. Because of extra word width, often a routing word can be used to traverse more than one stage. However in large multi-stage networks it often cannot be used to cross the entire network, and must be *swallowed* at some point. During a swallow the original routing word is discarded and a new routing word with information for subsequent stages propagates to the front of the message as shown in Figure 3-4.



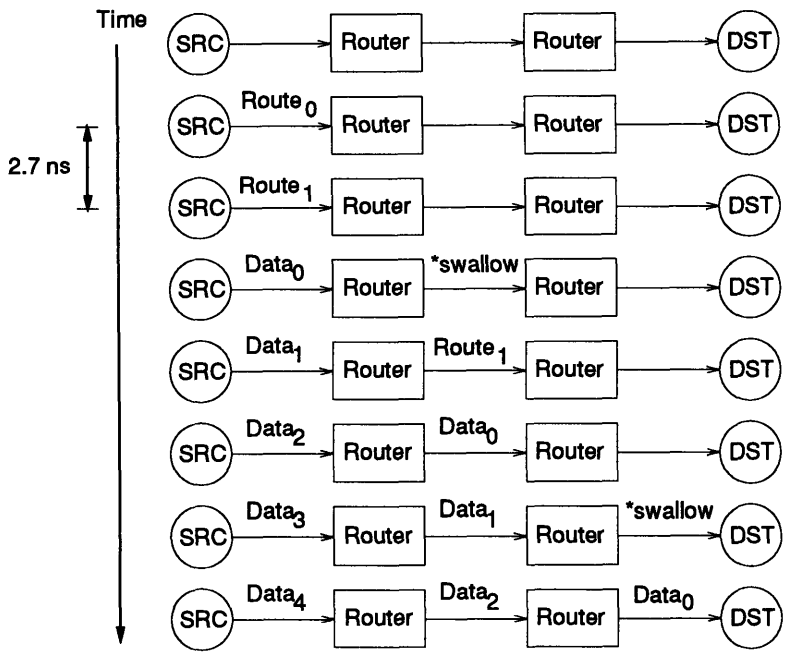
The network is idle when SRC starts to input a message to be routed to DST. The message requires one cycle to cross every wire and go through a component. Because of the limited width of the routing words, in the last stage the routing word has been used up and is *swallowed*. In the fastest single cycle implementation each cycle lasts for 4.3 ns.

Figure 3-4: Message Routed Through a Network with Single-Cycle Allocate

Because arbitration latency is high, the allocation of initial routing words limits the clock frequency of the network. The latency of a single cycle allocate which allows a routing word to be used for multiple stages requires the full 4.3 ns. The case of a small message, containing only eight data words, requires 25.8 ns to make a initial connection through the network seen in Figure 3-4. For the entire message to completely traverse this network requires 55.9 ns.

Compared to routing words, data words require much less processing time, only about 1.8 ns. One obvious way to improve network performance is to pipeline the initial allocation process into two cycles of 9 gate delays or 2.7 ns. This cycle time is limited by a feedback path associated with port availability, which will be described later. From the system viewpoint the network appears to be swallowing a routing word at each stage.

For the example network shown in Figure 3-5, this double-cycle allocate increases the number of routing words from one to two, and the total message length to ten words when compared to the single-cycle allocate. However, the increased clock frequency on all words



In this case the allocation process occurs over two cycles which forces a routing words to be swallowed at each stage. The data words of the message still requires only one cycle to cross every wire and go through each component. Each cycle lasts for only 2.7 ns.

Figure 3-5: Message Routed Through a Network with Double-Cycle Allocate

offsets the cost of requiring a new routing word for each network stage. An initial connection requires only 18.9 ns and the total transit time reduces to 35.1 ns. When compared to similar single-cycle allocate, the pipelining increases bandwidth, decreases total latency, and actually *decreases* initial connection latency.

3.4 Dynamic Dilation

By reducing flexibility given to input port requests, we were able to increase bandwidth and reduce latency. This was the move from general dilation to fixed dilation. Fixed dilation does not allow a message to decide its dilation. Instead, this is statically set at design time. In the process of *reducing* allocation latency we pipelined the allocation decision into two unequal cycles. The reason for this is explained in the previous section. Dynamic dilation uses the

extra time in one cycle to recover the lost dilation flexibility. As with fixed dilation, the entire chip operates in one dilation during each allocate cycle. But, similar to general dilation, dilation is chosen based on the input port allocate requests at the beginning of each cycle.

Because the component must handle multiple requests for allocation in a single cycle, two input ports might request different dilations. This requires some mediation to choose a single dilation to be used for all the ports. To insure that an input port is not forced to take a port it does not consider equivalent, the lowest dilation must always be chosen.

A simple example would be if two input ports simultaneously requested dilation two and dilation four. If the chip dilation were incorrectly set to four then the input port requesting dilation two could be connected to any of four output ports even though it considers two ports to not be equivalent to the ones it is requesting. However if dilation two is chosen then the input port requesting dilation four is only forced to take a subset of the requested ports, but each one is still acceptable.

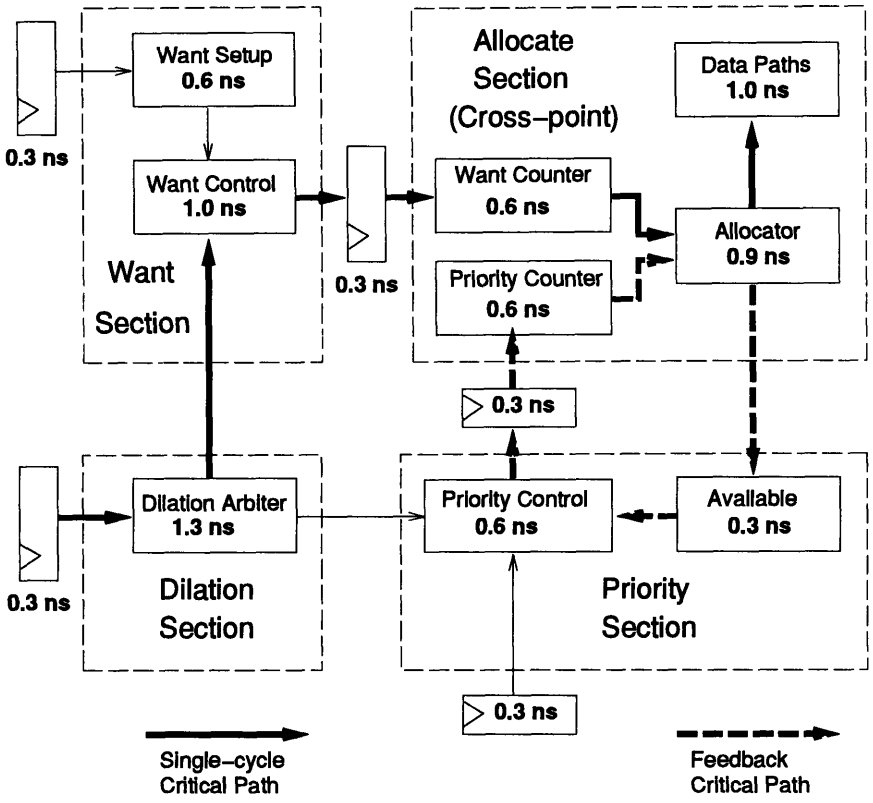
3.4.1 Timing Issues

The breakdown of delay in the final version of the dynamic dilation allocator is shown in Figure 3-6. The most important part of the diagram is the feedback path in which the new allocates of one cycle are fed back into the availability paths of the next cycle. This must occur on a cycle by cycle basis or it would allow an allocation of an output port on one cycle and then another allocation of the same port on the next cycle. The feedback path which fills the second allocate stage limits the cycle time to 2.7 ns.

The latency of the first cycle turns out to be 2.6 ns, even with the added logic to support dynamic dilation. Therefore the addition of dynamic dilation comes at simply the cost of filling up the unused part of the first cycle.

3.4.2 Flexibility Comparison

Dynamic dilation provides as much flexibility as general dilation in all instances except for multiple allocations requesting different dilations in one cycle. Normally when there is only one requesting port, the port would get whichever dilation it requested. Because there is only one allowed dilation each cycle, if two ports request different dilations, one will be forced to



This diagram includes all the major functional units of the allocate logic with their associated delays. Included are the pipeline stages used to reduce overall network latency. The dotted bold path shows the feedback loop which limits cycle time to 2.7 ns. The plain bold path marks the critical path of an unpipelined version of the allocator which requires 4.8 ns.

Figure 3-6: Timing/Circuit Diagram of Dynamic Dilation

take a dilation it did not request.

However, the likelihood of such an event is rare. $T_{message}$ represents the average number of cycles for a node to spawn a message. Therefore, the reciprocal of $T_{message}$ becomes the frequency at which a new route will begin or the probability that a route will start on a given cycle. If N_{nodes} is the number of nodes, then the number routing components in each stage is half that. From this, $P_{2messages}$, the probability that two messages arrive on a given cycle, where N_{ports} is the number of stages, can be calculated:

$$P_{2messages} = \left[\frac{1}{T_{message}} * N_{nodes}\right]^2 * \left[\frac{2}{N_{nodes}} * N_{stages}\right] \quad (3.1)$$

The first term in the above equation represents the probability of getting exactly two messages entering the network on the same cycle. The second term is the chance of the messages colliding in the same routing component.

The probability of getting a collision between two ports requesting *different* dilations, $P_{collision}$, relies on the distribution of dilation requests. If one assumes an even distribution of these probabilities, $P_{different}$, then $P_{collision}$ follows easily:

$$P_{collision} = [1 - P_{different}] * P_{2messages} \quad (3.2)$$

The first term is equal to the probability of the second message not being the same dilation as the first. These back-of-the-envelope calculations do not include the second order effect of more than two messages coming in at the same time. The probability of getting three messages at once is small compared to the probability of getting two, by a factor of $t_{message}$.

Let's consider a reasonable example:

- 64 nodes ($N_{nodes} = 64$)
- 4 stages ($N_{stages} = 4$)
- 3 dilations ($P_{different} = \frac{1}{3}$)
- 100 cycles between messages per processor ($T_{message} = 100$ cycles)

$P_{2messages}$ would be 0.08%. $P_{collision}$ comes out to about 0.05%. Furthermore, it is likely that most messages will be of the same dilation at a particular stage. In this light the fact that dynamic dilation is as flexible as general dilation for 99.9% of the time is quite impressive.

3.4.3 Latency Comparison

Table 3.1 summarizes the performance of various implementations of the three different dilation schemes. General dilation is by a factor of five the slowest scheme, but provides the greatest flexibility. The wave-style and parallel-style implementations of fixed dilation greatly

Type	Allocate Latency (ns)	Initial Connection Latency (ns)	Total Connection Latency (ns)
general	14	84	182
fixed wave	5.2	31.2	67.6
fixed parallel	4.3	25.8	55.9
piped fixed	2.7 (x2)	18.9	35.1
piped dynamic	2.7 (x2)	18.9	35.1

These results are for an 8 word message traveling the network shown in Figure 3-4 and 3-5. The *allocate latency* sets the system clock. The *initial connection latency* is for one hop across the network, while the *total connection latency* is for the entire message.

Table 3.1: Dilation Performance Results

reduce latency, but severely limit dilation flexibility. The pipelined versions of dynamic and fixed dilation provide the lowest network latencies of every other technique, and dynamic dilation recovers most of the flexibility.

Chapter 4

High Performance Logic Style

The primary design concerns in this project are the latency and bandwidth of the routing component. In order to improve both of these paramters I needed to minimize the latency of a single critical path. I found that by altering the conventional style of high speed logic, domino logic, I could increase speed by almost a factor of two. These findings are especially interesting because they are not limited to this singular application. They could be applied to almost any clocked design, where logic latency affects performance.

This chapter reviews conventional domino-style logic, which is generally one of the fastest logic forms in clocked digital systems. Improvements to this logic form and their effect on latency as well as noise and area are discussed. Finally implementation difficulties and solutions to these difficulties are outlined.

4.1 Conventional Domino Logic

A single stage of the conventional style of domino logic is shown in Figure 4-1. This stage implements the general function:

$$OUT = [input(1,1) * input(1,2) * ... * input(1,m)] + ...$$
$$+[input(n,1) * input(n,2) * ... * input(n,m)]$$

By chaining a number of these stages together, as in Figure 4-2, we can obtain any arbitrary positive function. One important feature to note is the use of a cutoff device at the bottom

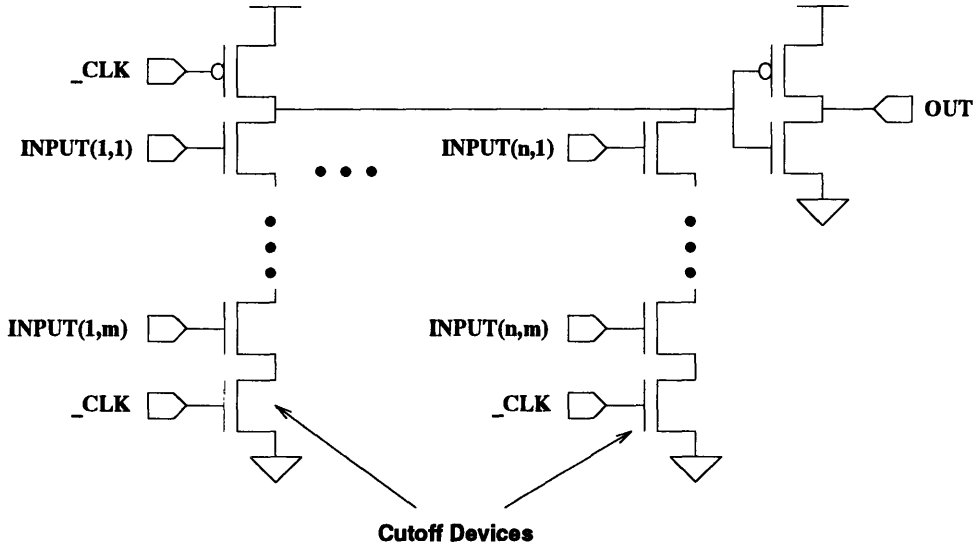


Figure 4-1: A single stage of domino style logic

of the pull-down chain. This cut-off device serves two purposes. First it minimizes power spiking caused by the precharge p-device turning on while the pull-down path is still asserted. Secondly, it allows all domino stages to precharge simultaneously, reducing the total precharge time. Typically at the beginning of every new cycle the entire chain of domino logic will precharge in order to prepare for an evaluation as seen in Figure 4-2.

To calculate the total clock cycle required by conventional domino logic, we need to add

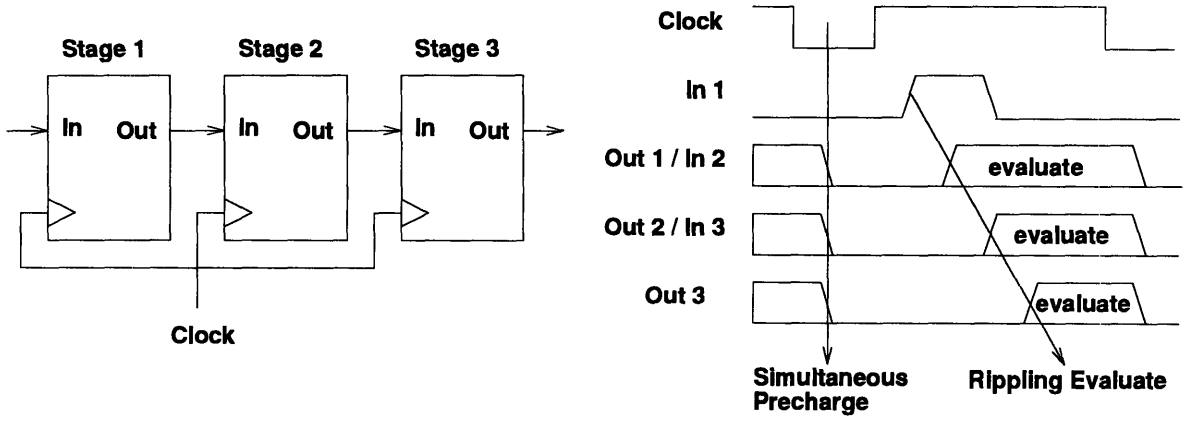


Figure 4-2: Timing diagram for a three stage conventional domino path

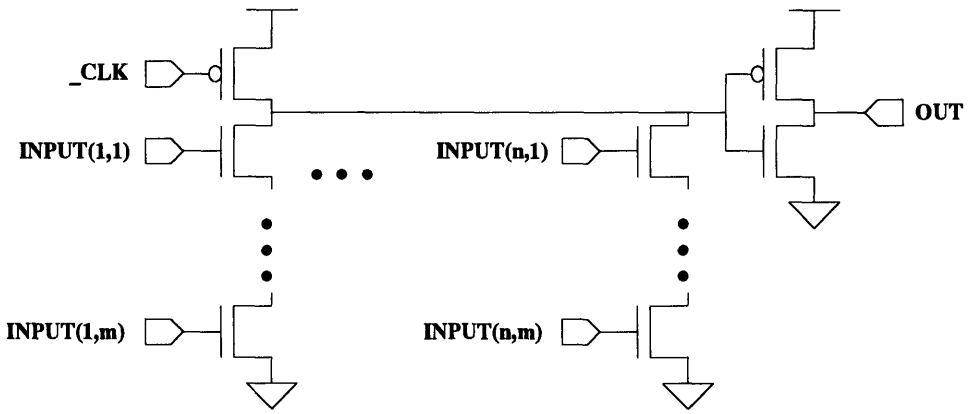


Figure 4-3: Domino without cutoff devices

the precharge time, the evaluate time and whatever setup time constraints we have on our registers or latches. The precharge latency equals the amount of time required to charge the internal capacitance of the domino stage to V_{dd} plus the delay of the inverter. However, because this time is short, the precharge time is more likely set by the rise and fall times of the precharge clock plus the minimum allowable pulse width. The evaluate latency consists of a signal rippling through a chain of precharged logic stages as shown in Figure 4-2. The cut-off devices slow the rippling evaluate by adding one more serial transitor/resistance to each pull-down path.

Secondary effects of cut-off devices include an increase in capacitive loading on the clock. This requires a larger driver, which creates di/dt noise on the power lines which could feed possibly sensitive logic. Also because more buffering is required, minimum pulse widths are slightly increased, resulting in slightly longer precharge times. Finally, the cut-off devices eat up valuable area within circuit sections. This requires longer interconnect and larger wire capacitance which slows down the circuitry.

4.2 Improvements on Conventional Domino

4.2.1 Removal of Cut-off Devices

Removing all the cutoff devices from conventional domino logic, as shown in Figure 4-3, allows up to a fifty percent gain in the speed of the rippling evaluate. The theoretical

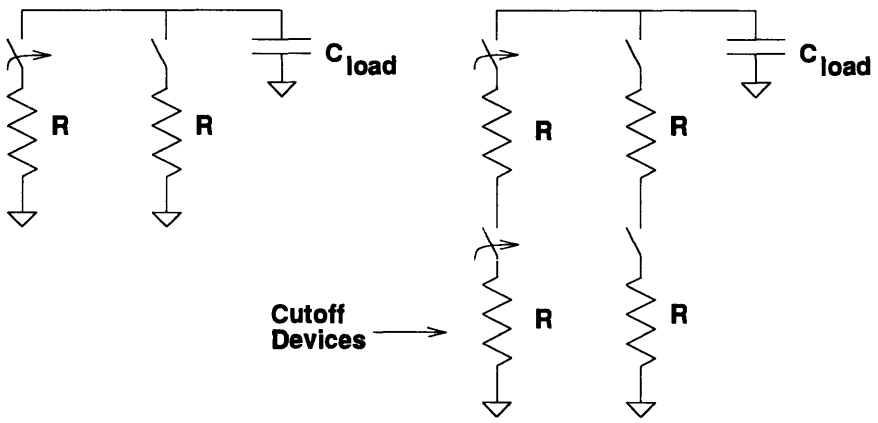


Figure 4-4: Resistor model of a 2-input nor with and without cutoff devices

limit occurs in the n-input nor gate. If we model the pull-down path as a resistor chain as in Figure 4-4, the limit is easily calculated. For the n-input nor, the worst case, pull-down resistance,R_{conv}, is $2R$. In a style without cutoff devices the resistance, $R_{no-cuts}$, is R. For the same C_{load} and ignoring self-loading, the percent increase in speed is given by:

$$\%increase = 100\%(\frac{R_{conv}C_{load} - R_{no-cuts}C_{load}}{R_{conv}C_{load}})$$

$$= 100\%(\frac{2RC_{load} - RC_{load}}{2RC_{load}}) = 100\%(\frac{1}{2}) = \mathbf{50\%}$$

Unfortunately in domino logic this effect is limited by the near constant speed of the inverter at the end of every stage. As it turns out the increase hovers between 15% and 30%. The types of gates that fit in this range include all nor gates, buffers, and up to 4-input nand gates. Complex gates, which use combinations of the above types, can be shown to work in this range as well. Other logic forms like, NORA [9], which alternate pull-up stages with pull-down stages, do not require a standard inverter. Their performance would not be limited by the constant speed inverter, but may have dificulties with noise immunity.

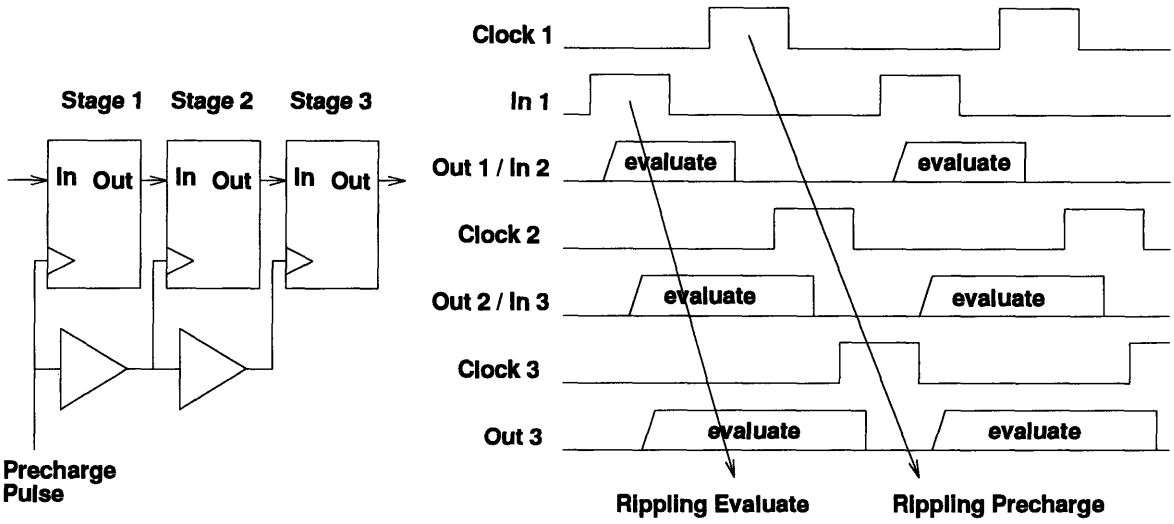


Figure 4-5: Timing diagram of new logic

4.2.2 Rippling of Precharge

As described earlier one purpose of the cutoff device is to prevent power spiking. To prevent spiking without using cut-off devices, the inputs to a stage must be low before that stage begins to precharge. Rippling the precharge through each stage as shown in Figure 4-5 accomplishes this. The first stage precharges before the next stage will start to precharge. The evaluate devices, themselves, act as the cut-off devices.

Such a technique requires a matched delay chain to delay the precharge pulse to each stage. If we make the precharge assertion only long enough to precharge a few stages, then while later stages are still precharging initial stages can begin evaluating. The need for a precharge time no longer exists because of the "wave pipelined" precharge and evaluate times. The logic for precharge pulse generation is described in Appendix E. One concern is that the precharge time will have to be long enough to withstand buffering and distibution. However, this concern does not directly affect path latency.

In total we have removed between 15% and 30% off the evaluate latency and we have completely removed the precharge period. Also the area is smaller, thus reducing interconnect capacitance and decreasing latency. In the specific application I looked at, the critical path of the routing component, these effects dropped the latency from 5 ns to 2.5 ns, a factor of

two. Another positive aspect is lower di/dt noise, since precharging is distributed across the entire cycle.

4.2.3 Summary of Benefits

- Speed increase in evaluation time (typically 30%)
- Speed increase by removing precharge time
- Smaller area (no cut-off devices)
- Reduced di/dt noise
- Adaptable to other clocked logic forms (eg. NORA)

4.3 Implementation Concerns

In this style of logic we are faced with two central problems each of which can be handled. The first is the race conditions which exist between the evaluate rippling through the logic and the rippling precharge pulse. The question to be answered is what happens if one or the other of these ripples overtakes the other and if there is anything that can be done to prevent such an event. The second problem to be dealt with is asymmetric fan-in from different locations on the logic chain.

4.3.1 Race Condition: Fast Precharge Ripple

The first race condition occurs if the precharge pulse overtakes the evaluate rippling through the logic. Should this happen, the value being calculated is lost and the circuit will not function properly. Realize this requires that the precharge pulse overtakes the **front** edge of the evaluate ripple. One solution is to delay the precharge pulse at the beginning of every cycle. This is partially taken care of by the delay associated with generating the short precharge pulse. Further delay can be achieved by inserting a simple buffer. Also the delay chain that ripples the precharge pulse can be designed to be slightly slower than the worst case delay of the evaluate logic which it feeds. Together these two techniques can effectively prevent the race condtion in which the precharge pulse overtakes the evaluate ripple.

4.3.2 Race Condition: Fast Evaluate Ripple

The second race condition is the converse in which the front edge of the evaluate ripple overtakes the back edge of the rippling precharge pulse. In this case a logic stage is still trying to precharge when its inputs change to evaluate values. This will cause some temporary power spiking, which will waste power, possibly reduce the lifetime of our transistors and introduce some di/dt noise on our supply lines. The possibility of this race condition can be minimized by shortening the precharge pulse and by closely matching the delay line with evaluate times. However, as will be seen when we later analyze asymmetric data paths, this kind of condition still exists unless we also throw some area at the problem.

4.3.3 Asymmetric Fan-in

The central problem associated with fan-in of asymmetric path lengths can be traced back to the race conditions mentioned earlier. If one evaluate path is very short or if there is a data dependent fast path, the evaluate can overtake the precharge pulse. For the short data path case the intoduction of extra stages will not affect whole path latency but can more closely match up the two ripples. Not much can be done for data dependent differences in speed. As stated earlier this race condition will not affect the logical correctness of the circuit. It will simply affect the amount of power wasted. However, with respect to assymetric path lengths and the second race condition logical correctness is an issue. In this case, an early stage that feeds a later stage may begin to precharge before the other inputs to the later stage have evaluated. This could lead to an incorrect evaluation in the stage with fan-in. Luckily there really is no uncontrollable data dependence to the precharge timing. Therefore the addition of extra stages into the short path will insure logical correctness. The extra area offsets the area gained by removal of the cutoff devices.

4.4 Flip-flops

Registering values from this type of logic is reasonably straight foward. The only problem to overcome is that the input is not necesarily stable at the rising edge of the next clock cycle. The input could conceivably precharge sometime before then, if it were early in the logic chain.

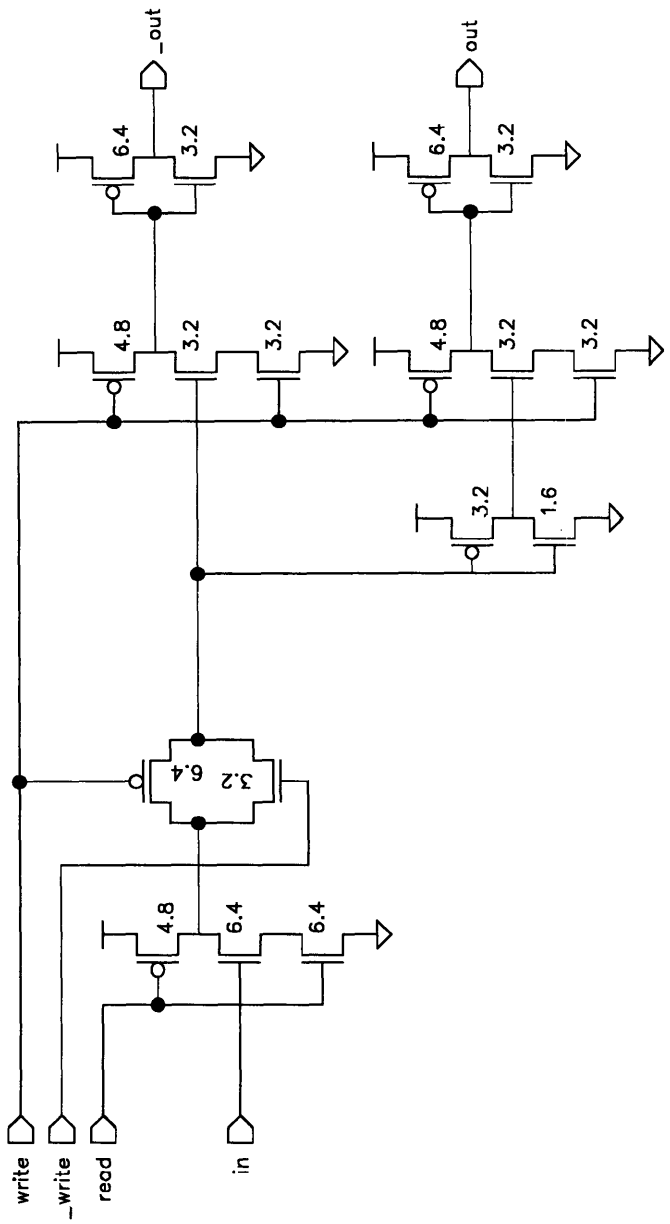


Figure 4-6: Flip-Flop for Modified Dynamic Logic Style

For this reason dynamic latches that have a regular domino stage as the master latch provide the correct functionality as in Figure 4-6. They look for a transition high on the input line and a transition back low will have no effect because the value will already be latched.

In order to hook up to the front end of this type of logic the output of the flip-flop must be stable at its final evaluated value or it must be in some precharge state. In order for this to work the slave latch of the flip-flop must also act as a domino stage which can precharge once the evaluate ripple has been started. It is actually the first stage in the domino chain to receive the precharge pulse. The configuration is shown in Figure 4-6.

Chapter 5

Conclusions

This thesis optimizes dilated self-routing crossbars, in order to increase the bandwidth of a multi-path interconnection network, without losing flexibility. Fixing the dilation minimizes the allocation latency of a single routing component. However, cross network latency can be further reduced by pipelining the allocate process into two cycles. The pipelining gives us the opportunity to regain some flexibility which was lost in the single cycle fixed dilation schemes. Pipelined routing components with dynamic dilation therefore provide routing flexibility as well as very low latency.

In order to further decrease the latency of the routing components, a new type of dynamic logic was employed. This logic style looks like conventional domino without the cutoff devices. Also, the local clocking methodology must be more complex. The advantage is a factor of two in speed over conventional domino logic styles.

Even though each of these techniques is used for a specific application, it is hoped that the concept of dynamic dilation and especially the new dynamic logic style can be applied to other systems and other networks with similar success.

Appendix A

Dilation Section

This appendix describes the final implementation of the dilation section in the pipelined version of the dynamic dilation routing component. The purpose of this section is to choose an appropriate dilation for the whole chip from among those requested by the input ports. The lowest dilation requested must always be chosen when there are conflicting requests. This limits one input port, but ensures that the other does not have to choose an output port which it does not consider equivalent. Ports which are not allocating are simply ignored.

As shown in Figure A-1 the section is broken into 4 stages. The first stage, *latch-static*, latches the values from the input ports and is shown in Figure A-2. The second stage, *dilation-buffer*, simply buffers the input signals so that they can be distributed. The last two stages handle the arbitration process. The third stage, *dilation-setup*, masks the dilation requests of non-allocating ports and combines the input requests. The final stage, *dilation-ctl*, translates the requested dilation into a form more easily used by the other sections. Transistor level diagrams are included for each block in Figures A-3 to A-5.

There are 3 different types of inputs. The first is the *_all-fwd* ⟨...⟩ signals which tell if an input is not attempting to allocate. The *_dil1-fwd* ⟨...⟩ and *_dil2-fwd* ⟨...⟩ encode the dilation being requested by each port. If *_dil1-fwd* ⟨1⟩ is low, the first port is requesting dilation 1. If *_dil2-fwd* ⟨1⟩ is low, the first port is requesting dilation 2. If neither are low, the first port is requesting dilation 4.

The output signals, *dilation1* through *dilation4*, tell the other sections what the dilation will be for the entire chip on that cycle. The appropriate output is enabled high.

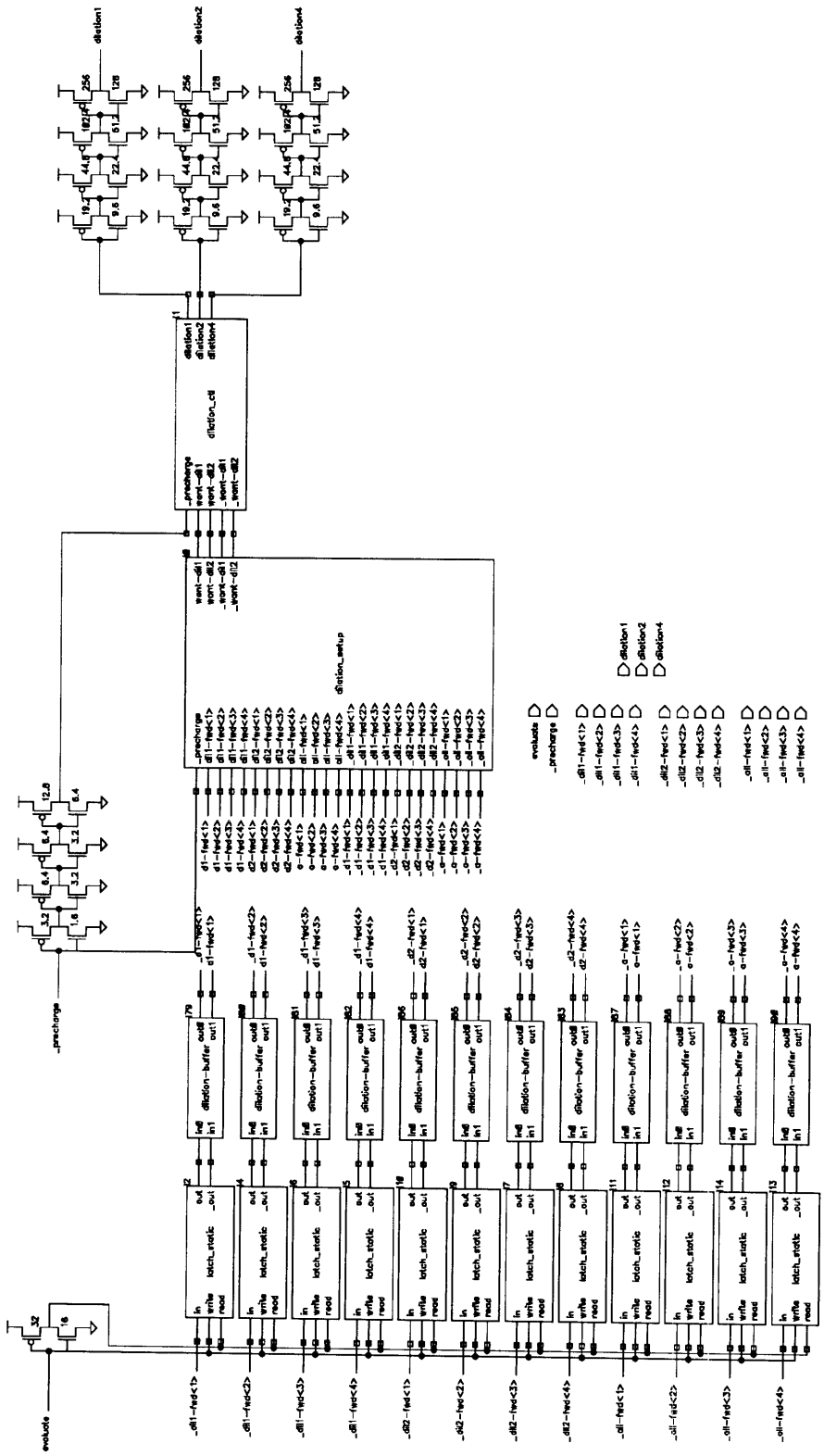


Figure A-1: Block Diagram of Dilation Section

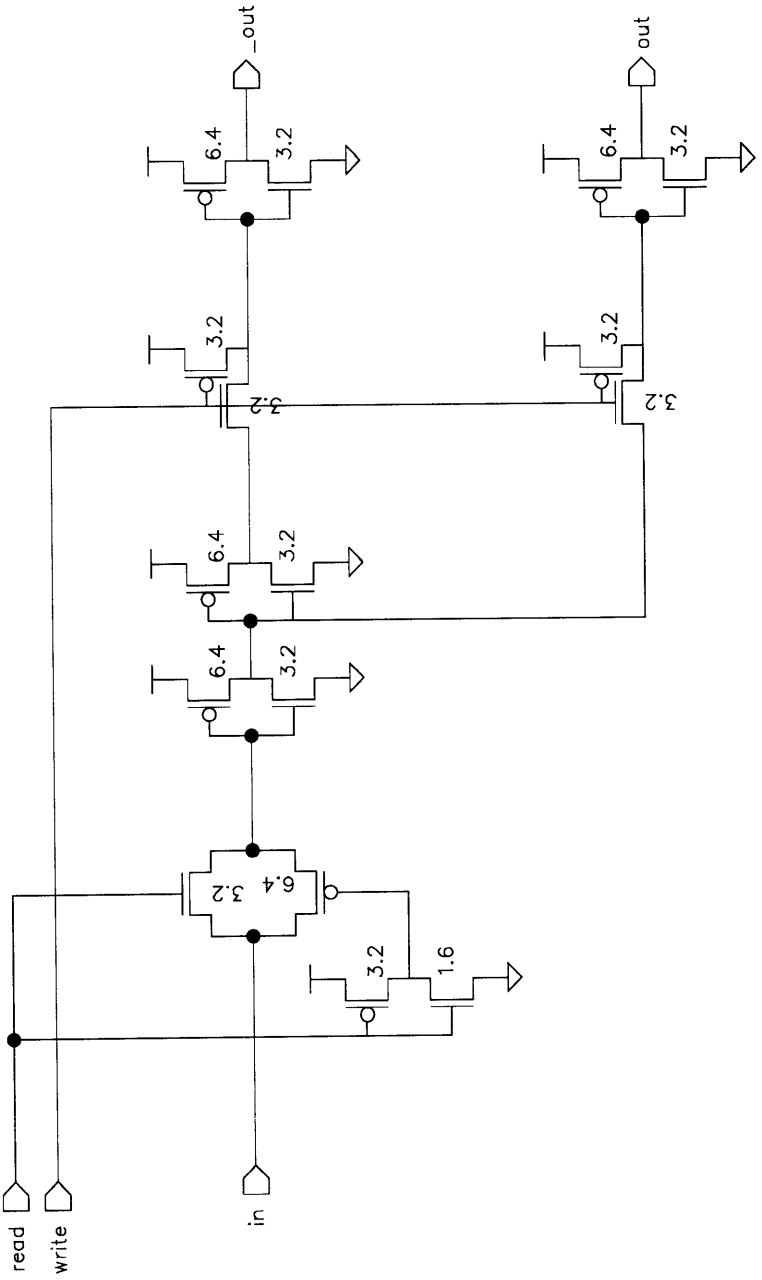


Figure A-2: Transistor Schematic of Latch Static

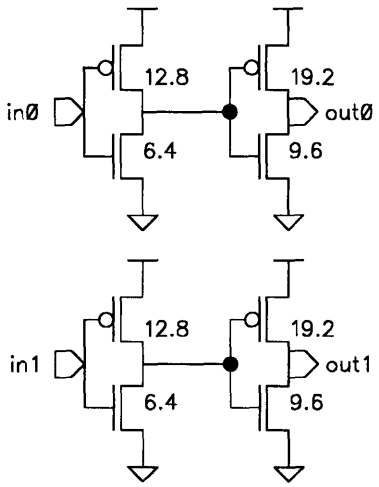


Figure A-3: Transistor Schematic of Dilation Buffer (2nd stage)

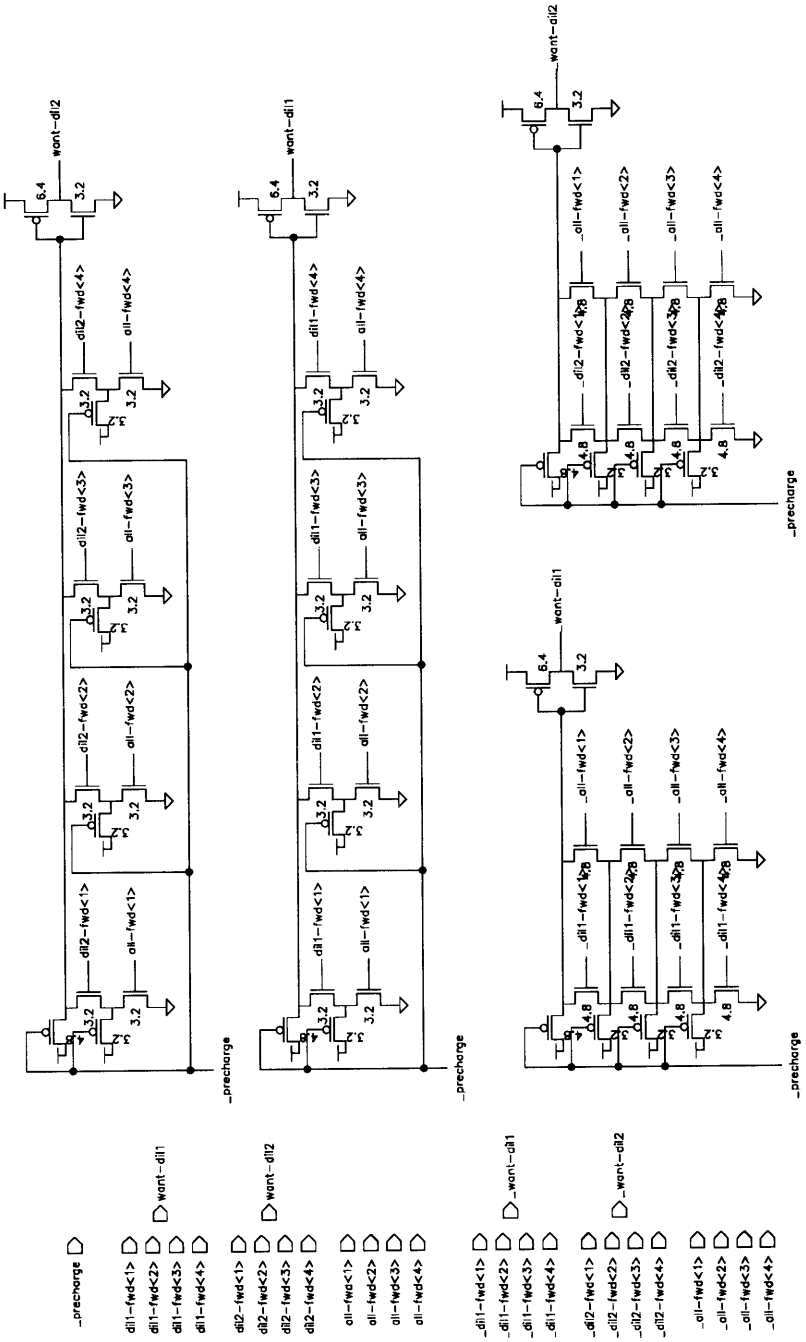


Figure A-4: Transistor Schematic of Dilation Setup (3rd stage)

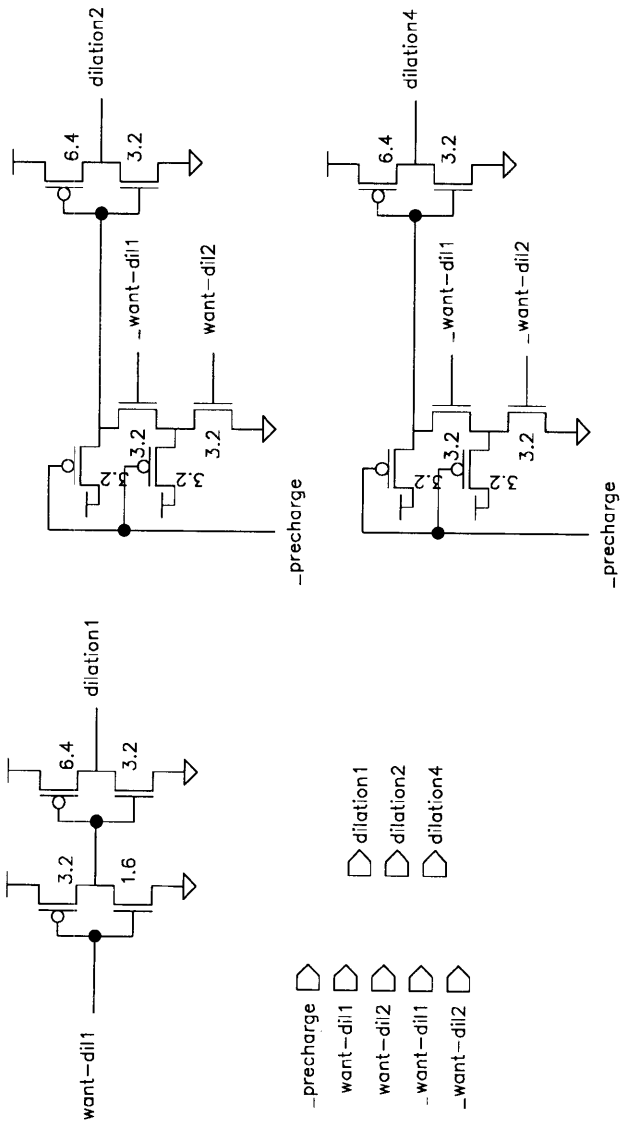


Figure A-5: Transistor Schematic of Dilation Ctl (4th stage)

Appendix B

Want Section

This appendix describes the final implementation of the want section in the pipelined version of the dynamic dilation routing component. The want section serves two purposes. First, it randomly assigns the order, or priority, in which each of the input ports is to be allocated with respect to each other. For any input port, its ordering is then distributed to its associated cross-points. Ports which are not allocating are excluded from this ordering and a special signal is sent instead. Secondly, this section sends a signal to each cross-point on whether the cross-point's associated input port wants a connection with its output port. This requires the translation of the input request and inclusion of the dilation. Since this entire section is not in the critical time path it uses conventional domino logic. However, to insure the reasonable operating speeds the inclusion of dilation is left for the last step.

The first two stages of the want section, Figure B-1, capture the input allocation request and hold the values for an appropriate period of time. The first stage, *latch-static*, latches the values from the input ports. The second stage, *alloc-precharge*, evaluates at the beginning of every cycle and holds its value until after the rest of the logic stages have begun to evaluate. Then it precharges and waits for the next clock cycle. The schematics for these stages are shown in Figure A-2 and in Figure D-8.

The rest of the logic is most easily described section by section. *Want-setup*, shown in Figure B-2, translates the raw input port allocation requests to a simple form covering every possible dilation. Table B.1 shows the appropriate translations of the raw requests

The next section, *want-ctl*, combines the data from *want-setup* with the dilation to produce

raw1	raw2	$\overline{w1}$	$\overline{w2}$	$\overline{w3}$	$\overline{w4}$	$\overline{w1-2}$	$\overline{w3-4}$
0	0	0	1	1	1	0	1
0	1	1	0	1	1	0	1
1	0	1	1	0	1	1	0
1	1	1	1	1	0	1	0

Table B.1: Table of raw, input request translations

the information on whether a cross-point's associated input port wants a connection with its output port. A transistor level schematic is shown in Figure B-3.

Another section which aids in the priority calculation, is *want-priority-setup* shown in Figure B-4. This logic block searches for possible collsions, or matches, of two input ports requesting the same set of output ports. It disregards the dilation. This match information is passed onto the next section, *want-priority-dilinc.*

Want-priority-dilinc, shown in Figure B-5, uses the dilation to combine the matches for the final stage. The last stage, *want-priority-ctl*, then uses the random data inputs to assign a priority and simultaneously includes the collisions between input port requests. Also, the priority sends a special signal if a port is not allocating that cycle. This output form is specified in Table B.2. The transistor level implementation is seen in Figure B-6.

	# of priority bits asserted low
priority 1	1
priority 2	2
priority 3	3
priority 4	4
not allocating	0

Table B.2: Table of priority output

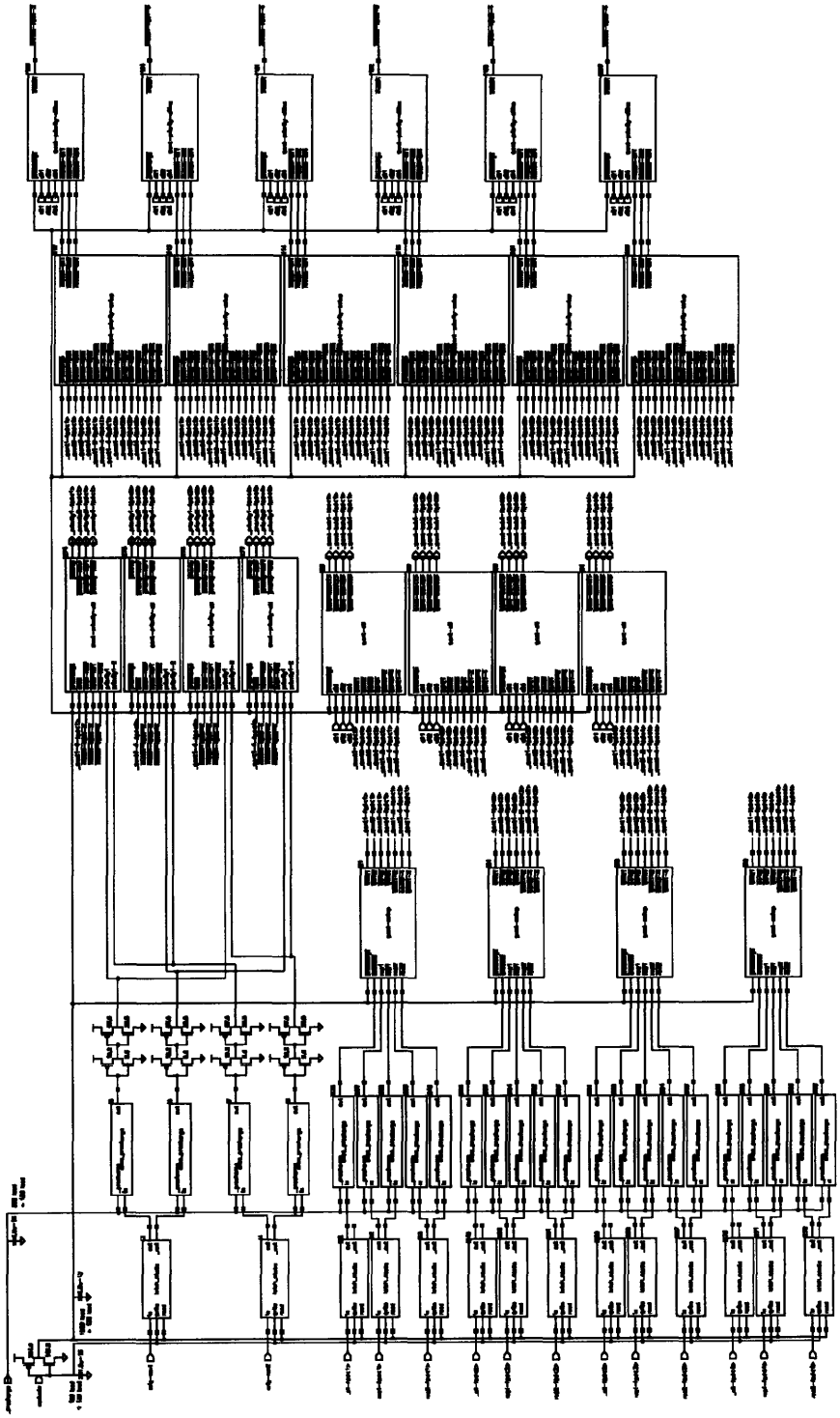


Figure B-1: Block Diagram of Want Section

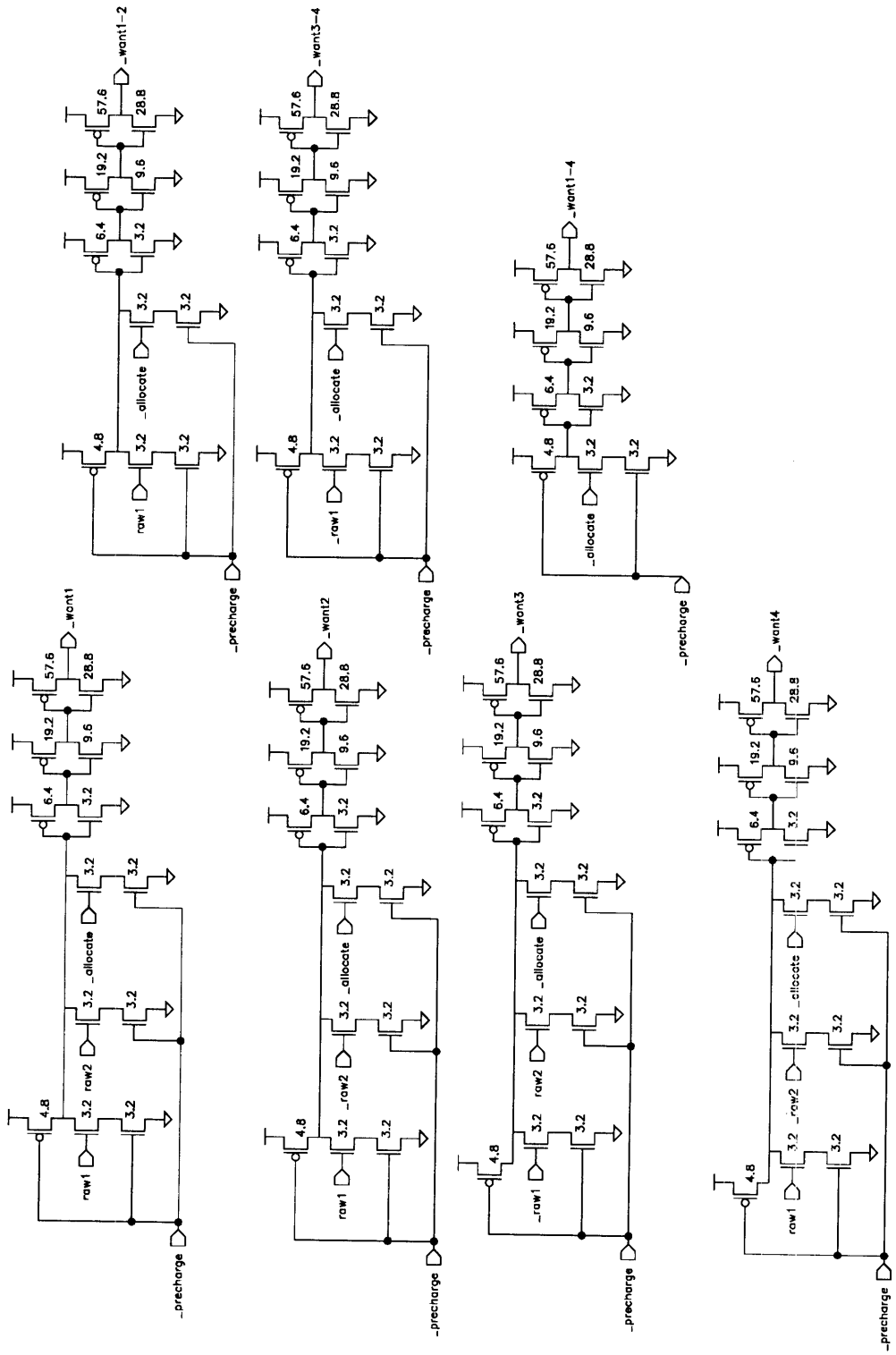


Figure B-2: Transistor Schematic of Want Setup

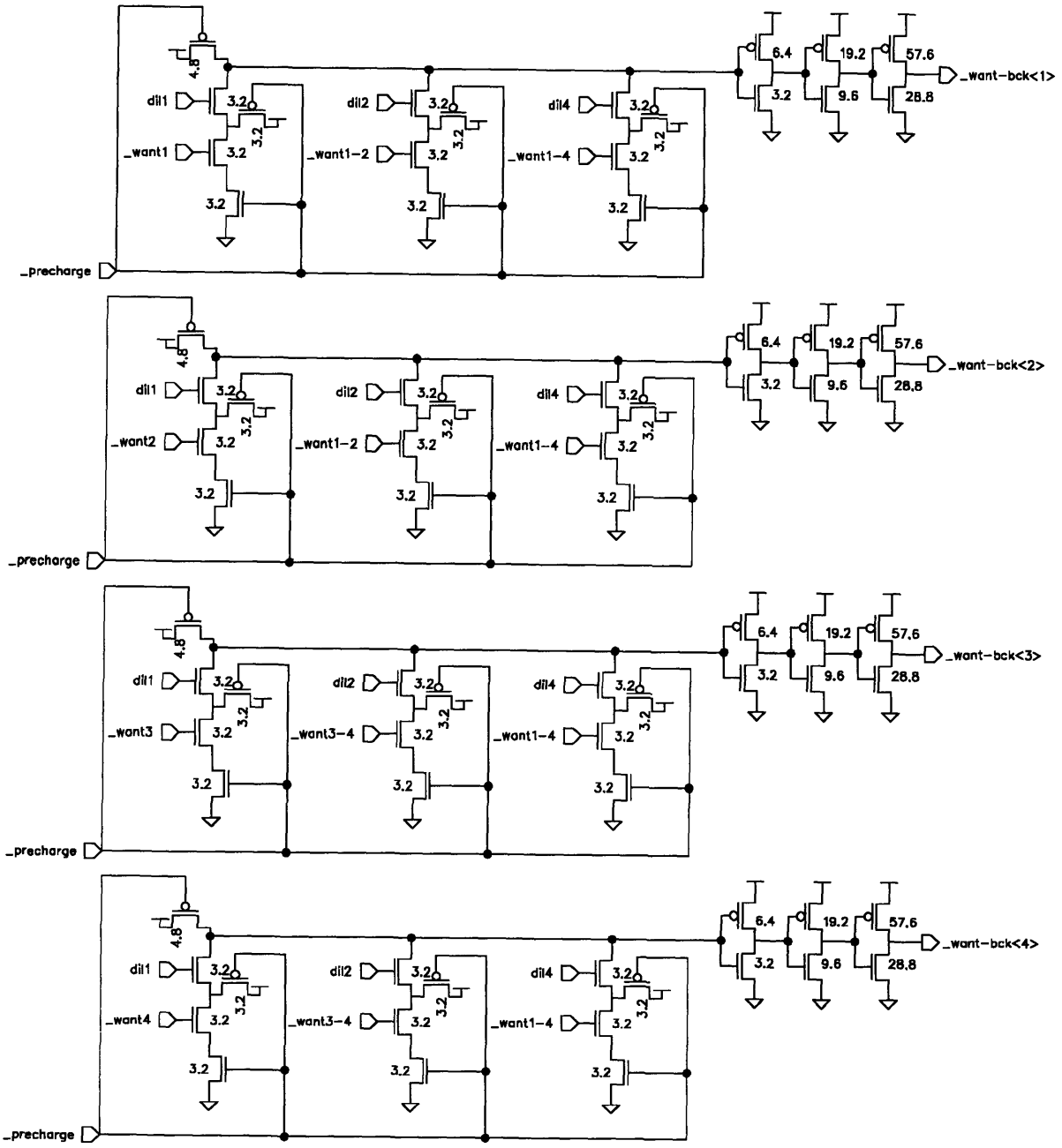


Figure B-3: Transistor Schematic of Want Ctl

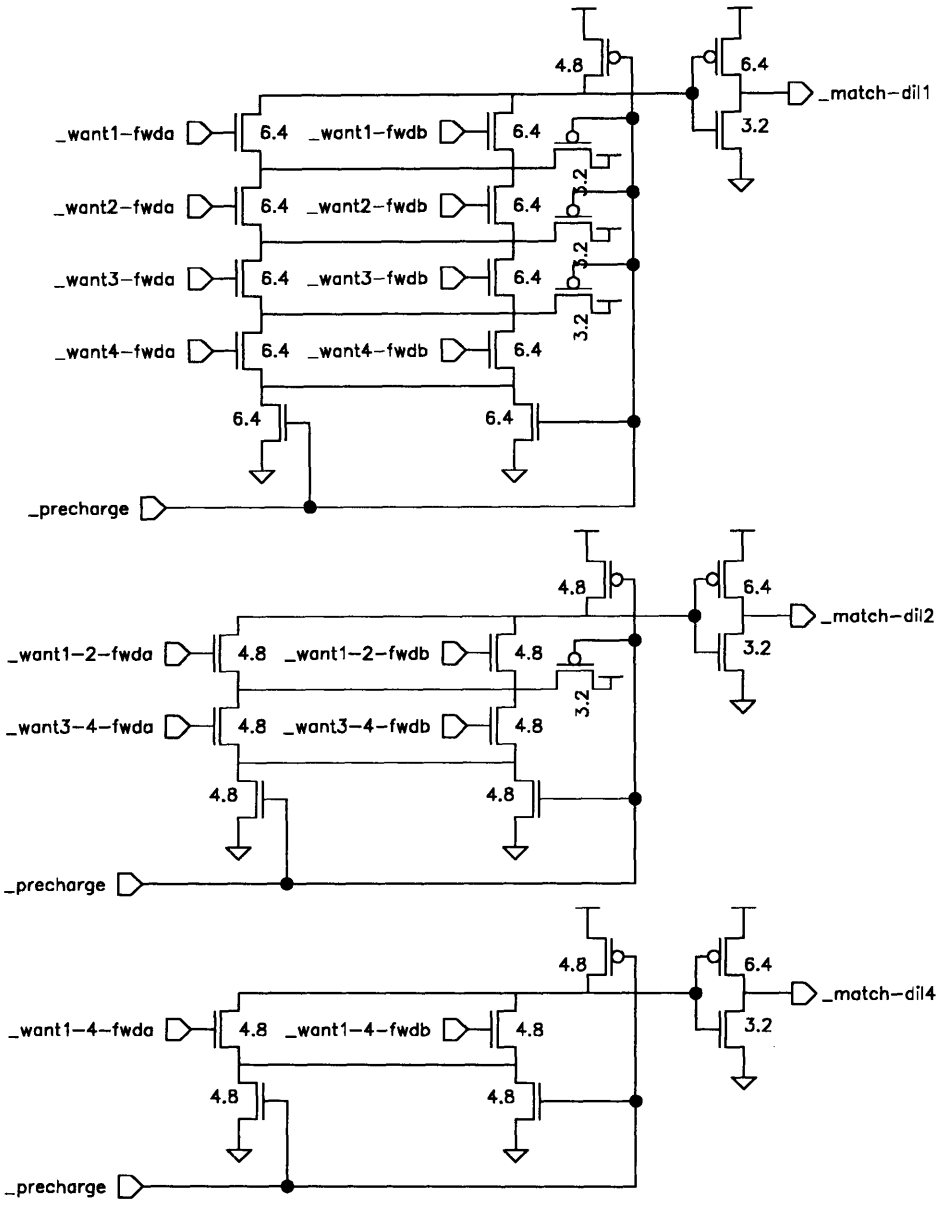


Figure B-4: Transistor Schematic of Want Priority Setup

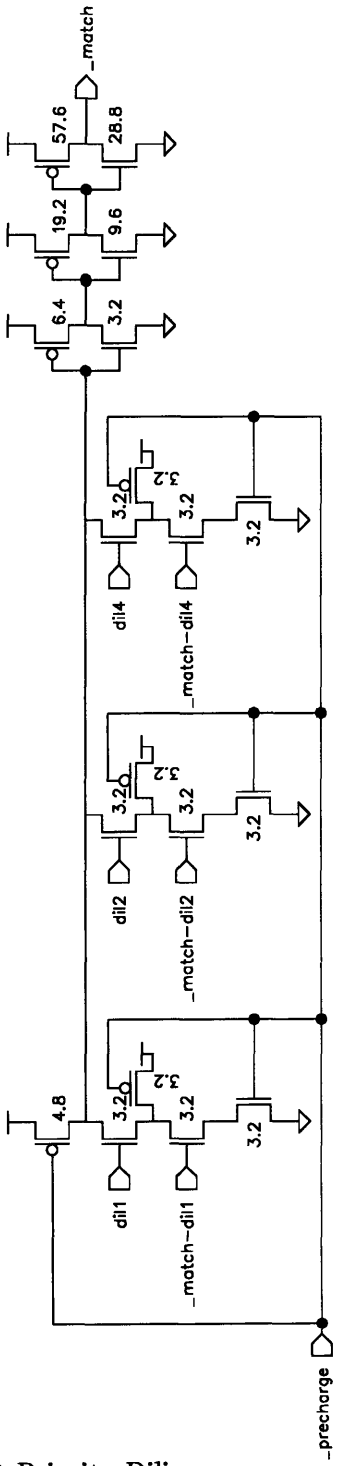


Figure B-5: Transistor Schematic of Want Priority Dilinc

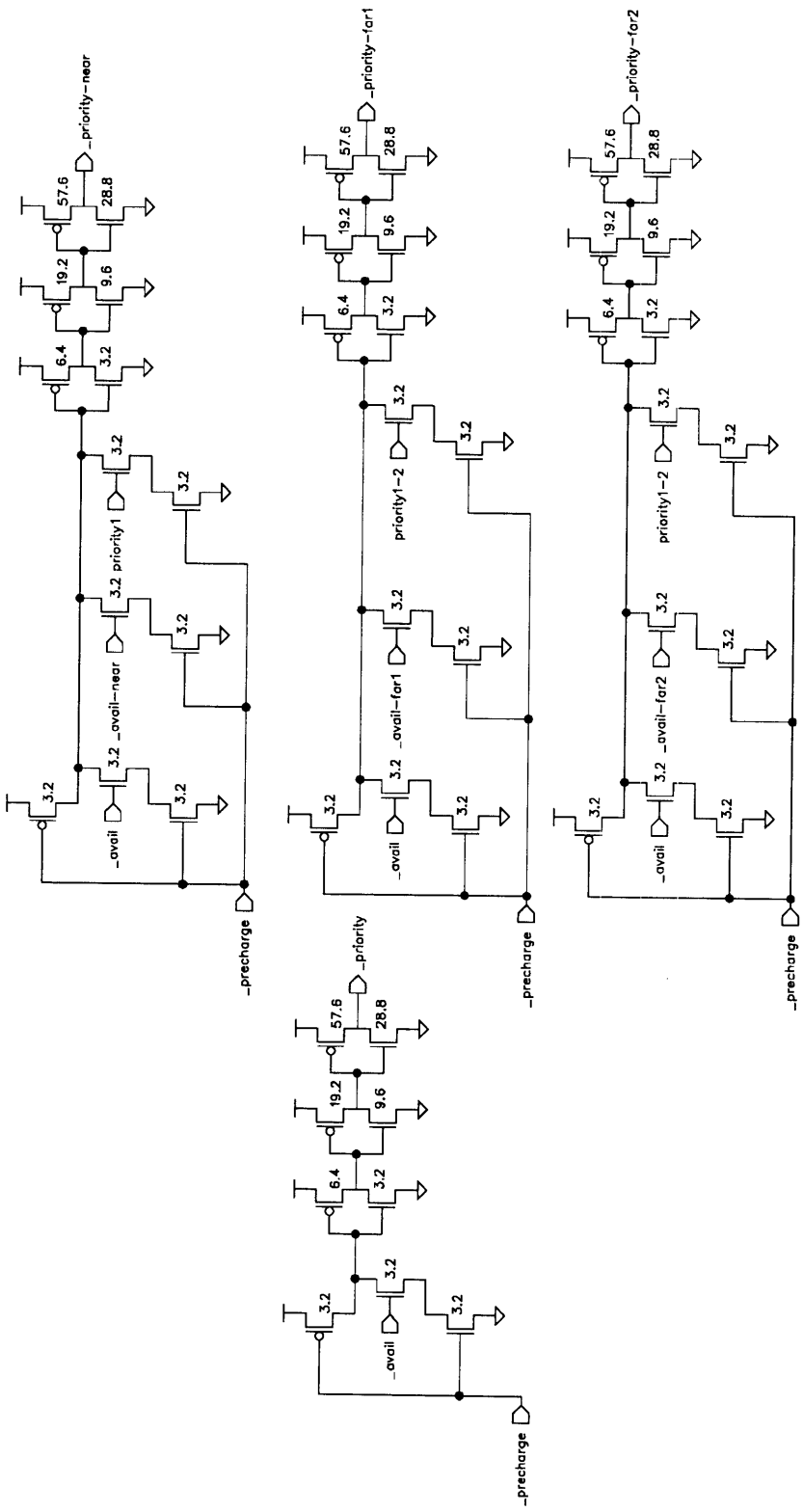


Figure B-6: Transistor Schematic of Want Priority Ctl

Appendix C

Priority Section

This appendix describes the final implementation of the priority section in the pipelined version of the dynamic dilation routing component. The priority section randomly assigns the order, or *priority*, in which each of the output ports is to be allocated with respect to eachother. For any output port, its ordering is then distributed to its associated cross-points. Ports which are not available are excluded from this ordering and a special signal is sent instead. Since only one stage is part of the critical path, this section uses conventional domino logic.

As described in Appendix B, the first two stages of the priority section, Figure C-1, capture the input allocation request and hold the values for an appropriate period of time. The schematic for these stages is shown in Figure A-2 and in Figure D-8.

The information on cross-point connections is held locally at each cross-point. In order to prioritize the output ports, the priority section must collect the information on output port availability every cycle. *Priority-available*, shown in Figure C-2, uses this information to produce a signal for each output port on whether it is available.

The next logic block, *priority-dilinc*, assigns the order with respect to the dilation and random data inputs. It produces to output signals. If the first signal, called priority1, is asserted, the output port will be either 1st or 3rd. Priority1-2 is asserted when the output port is to be 1st or 2nd. The next section handles the change in priority caused by higher priority ports being unavailable. The schematic for this block is shown in Figure C-3. Table C.1 showing the transformation between random data and output port priority is shown below. A lower dilation would simply force the priority lines to be asserted regardless of the random

ran1	ran2	port 1: p1	port 1: p1-2	port 2: p1	port 2: p1-2	port 3: p1	port 3: p1-2	port 4: p1	port 4: p1-2
0	0	0	0	1	0	0	1	1	1
0	1	0	1	1	1	0	0	1	0
1	0	1	0	0	0	1	1	0	1
1	1	1	1	0	1	1	0	0	0

Table C.1: Table of random input translations for dilation 4

input. For example in dilation 2, priority1-2 would be asserted for all ports.

The last stage, *priority-ctl*, uses the priority and availability to assign an order to be distributed to the cross-points. If the port is unavailable for allocation a special signal is sent. This output form is specified below in Table C.2. The transistor level implementation can be seen in Figure C-4.

order	# of $\overline{priority}$ bits asserted low
1st to allocate	1
2nd to allocate	2
3rd to allocate	3
4th to allocate	4
not allocating	0

Table C.2: Table of priority output

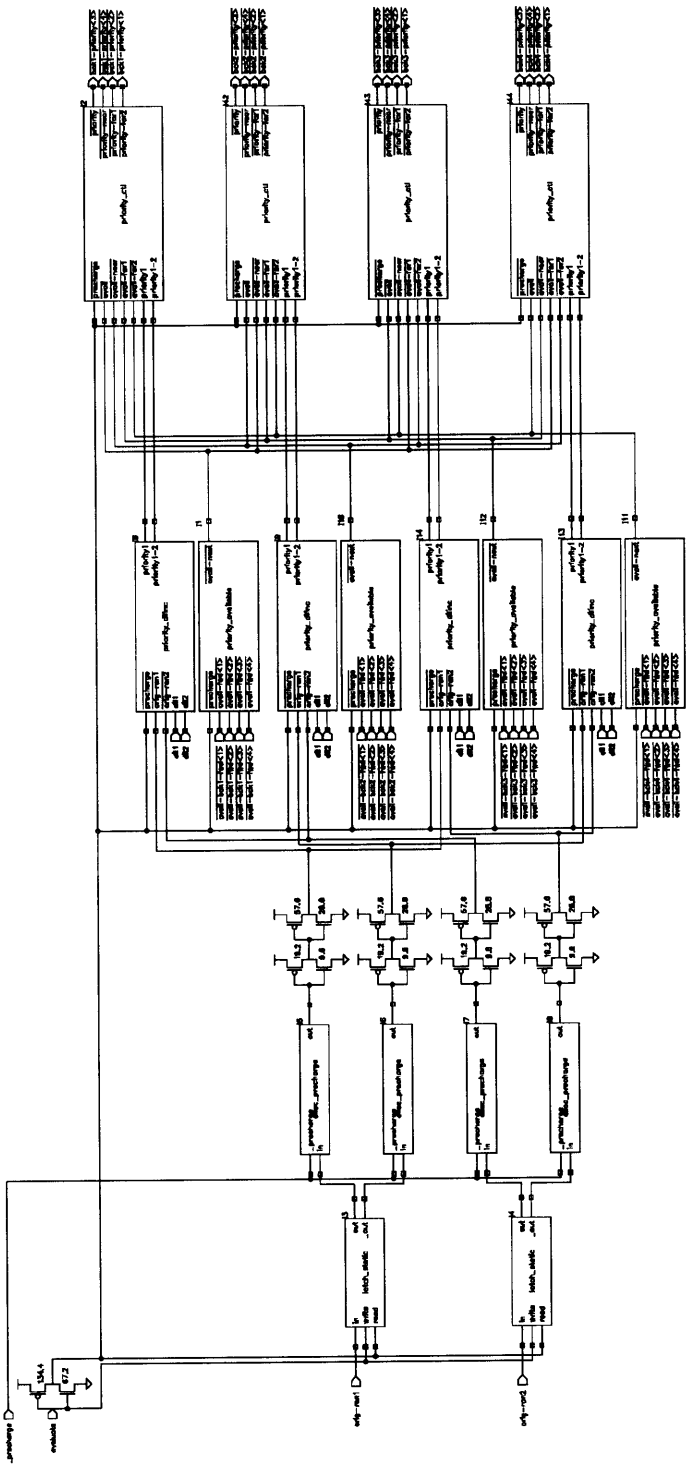


Figure C-1: Block Diagram of Priority Section

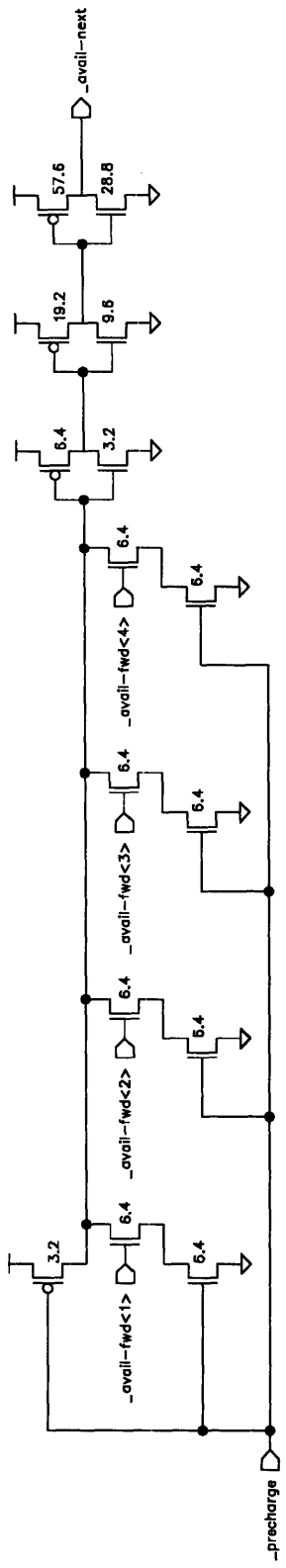


Figure C-2: Transistor Schematic of Priority Available

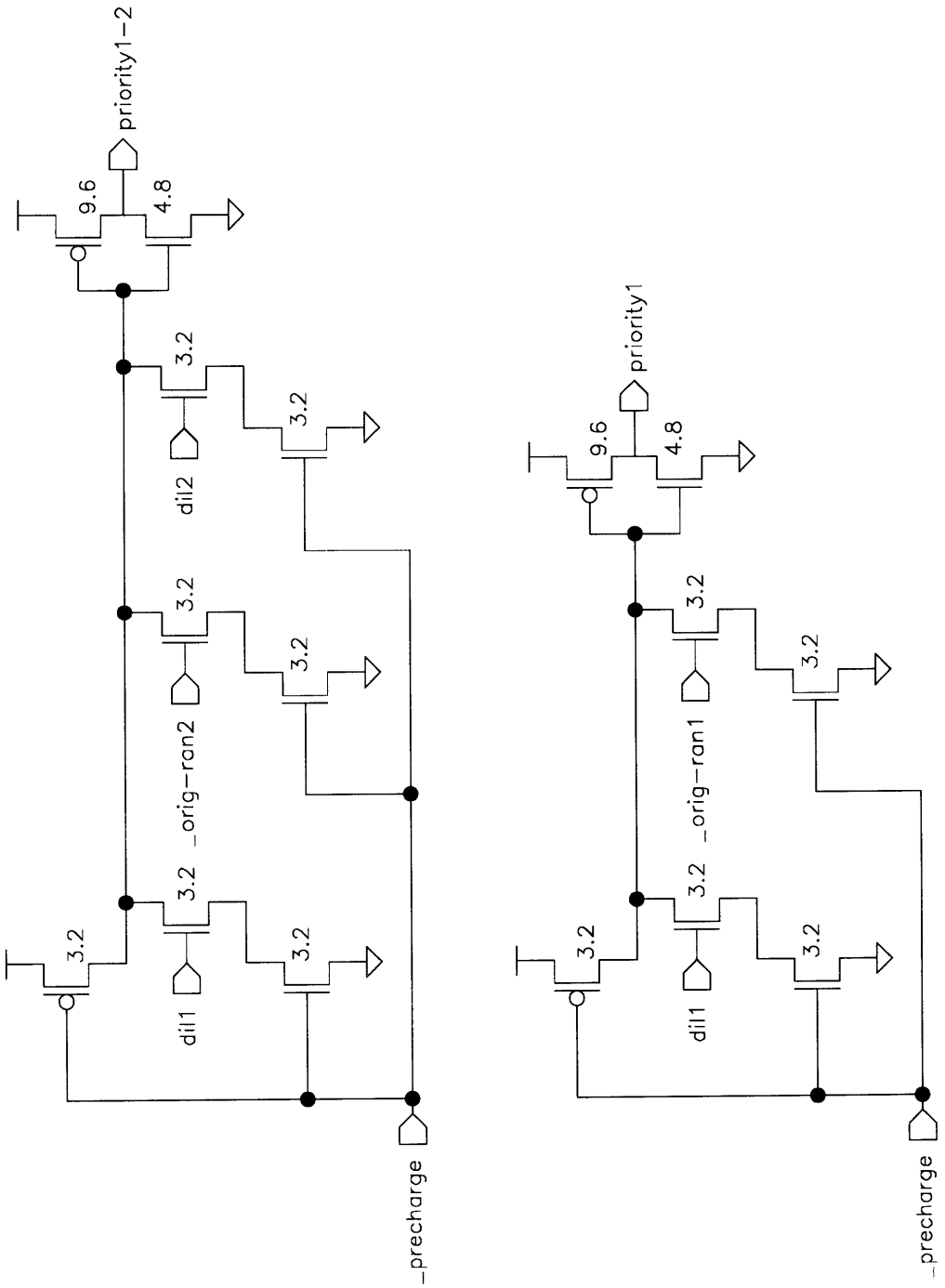


Figure C-3: Transistor Schematic of Priority Dilinc

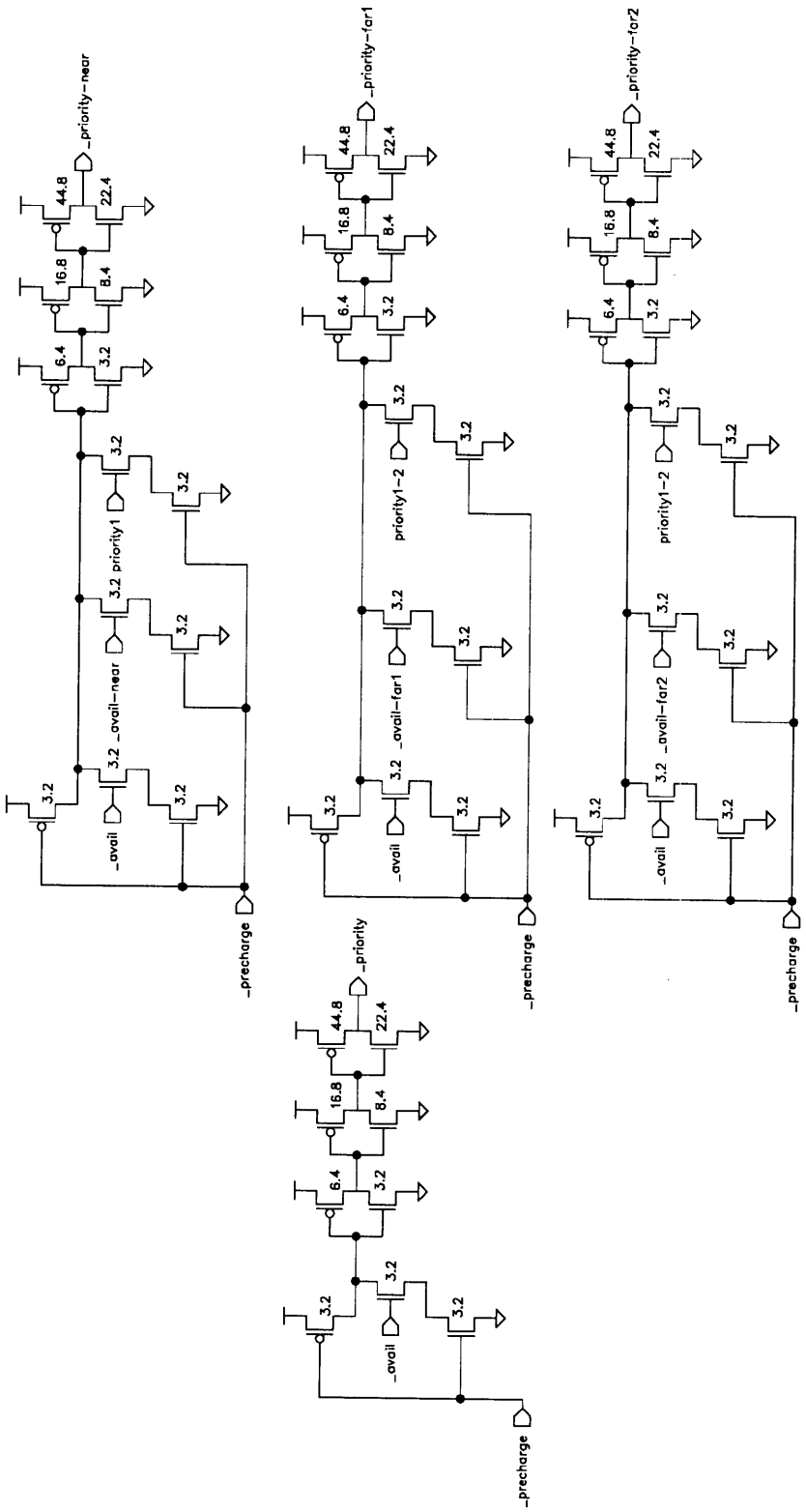


Figure C-4: Transistor Schematic of Priority Ctl

Appendix D

Allocate Section

The first block diagram shown in Figure D-1 is the array of cross-points or *alloc-block*'s. In this block diagram the want section would fit on the left. It feeds the array information on priority of input ports and on specifying port pairs with the possibility of allocatiion. The priority section would be located below the array and would feed in priorities and receive availability signals for each cross-point. The drop signals allow the cross-point that have previously allocated to drop their connection. Finally, the data lines for each port criss-cross the array.

Figure D-2 shows the block diagram for the cross-point, *alloc-block*. The first stage of latches described in Chapter 4, retime the priorities signals from the want and priority sections. The schematic for *latch-dynamic* is shown in Figure 4-6.

The following two stages, *alloc-2-counter* and *alloc-4-counter*, sum both sets of 4 priority signals so that *alloc-got* can compare the results to provide a connection desired signal. This process simplifies to comparing, for an input-output port pair, the order in which they are to allocate. If the two ports have the same order (e.g. input is 3rd and output is 3rd) then a connection is made and got is asserted. The transistor schematics are shown in Figure D-3, Figure D-4 and Figure D-5.

There are two stages to process the drop requests which would come from the input and output port control FSM's (not needed in this test chip). The first is a normal static latch described earlier and shown in Figure A-2. The second block is *alloc-drop*, which simply combines the two drop signals into one drop signal. Its transistor level diagram is shown in

Figure D-6.

The next set of logic blocks calculate whether a connection should be made and keep track of connection from cycle to cycle. The *alloc-final* will assign a connection if there is no drop input and there was an allocation on the previous cycle or there was a got assertion and the port pair wants to allocate. The transistor level schematic is included in Figure D-7. The allocate signal is fed to the pass gates which explicitly make the connection between the input port's and output port's data lines. Figure D-8 shows *alloc-precharge*, which retimes the asymetric logic paths as described in Chapter 4.

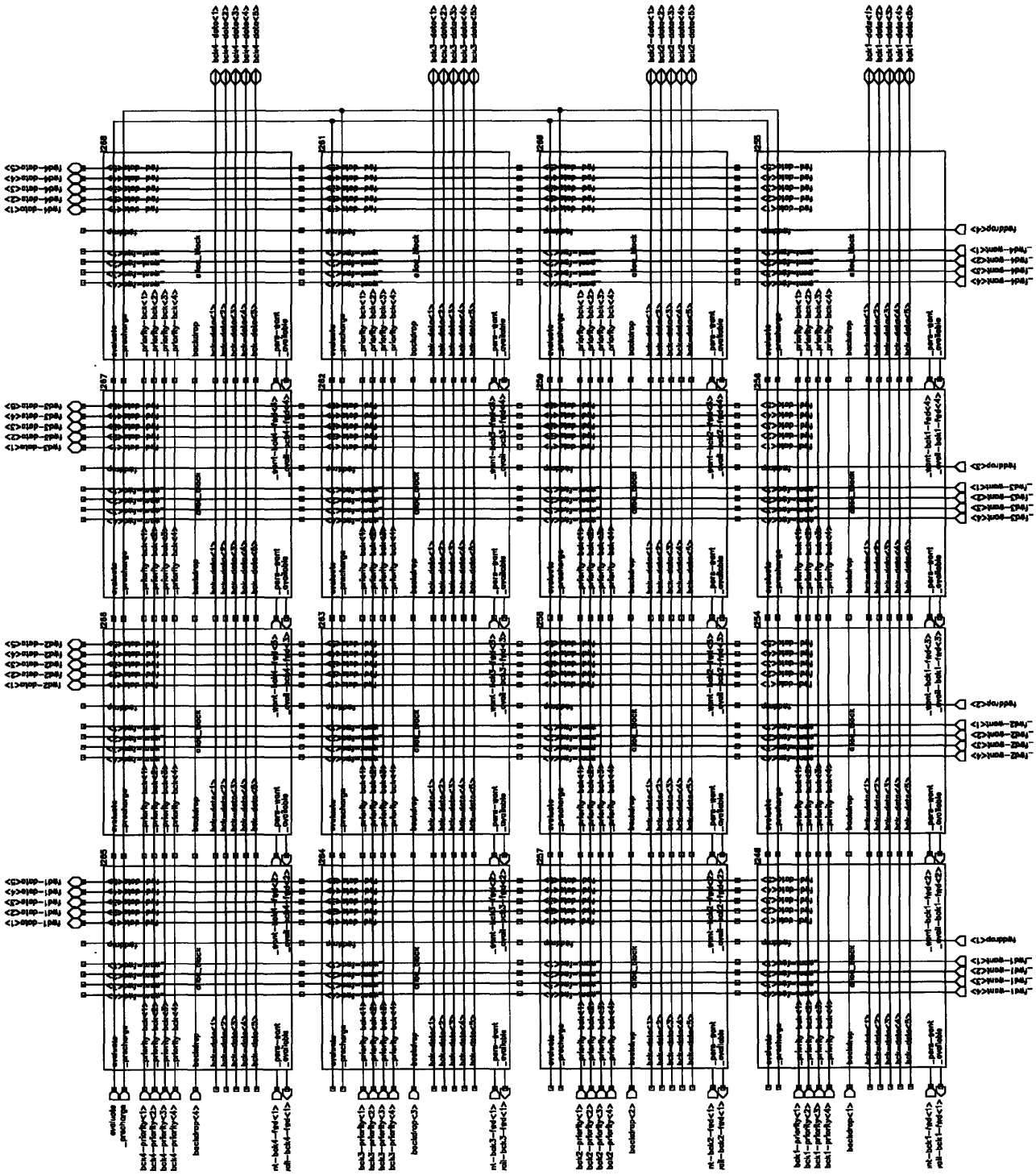


Figure D-1: Block Diagram of Allocate Section

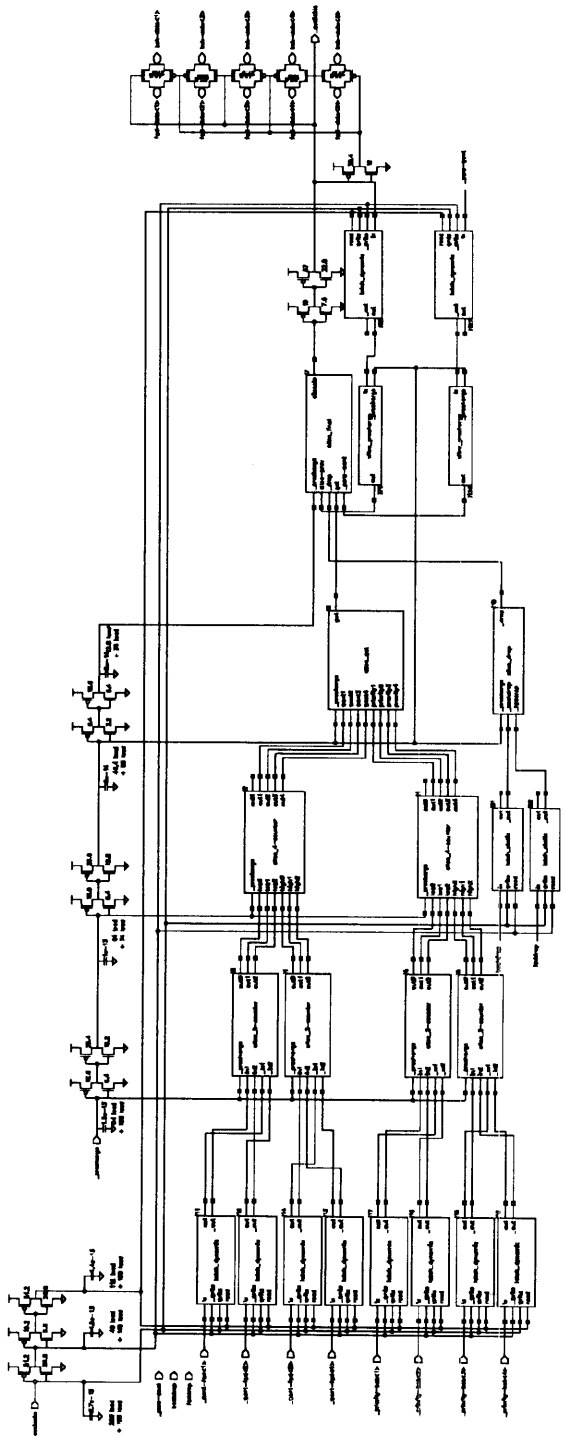


Figure D-2: Block Diagram of the Cross-Point, Allocate Block

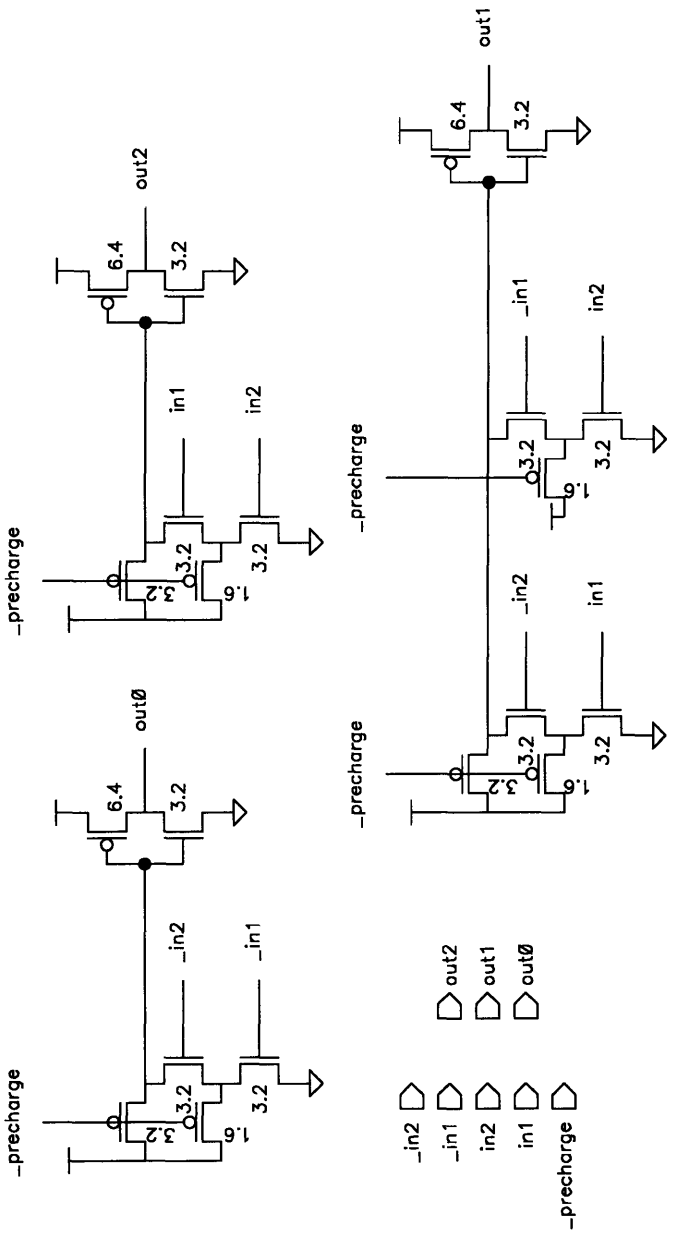


Figure D-3: Transistor Schematic of Allocate 2-Counter

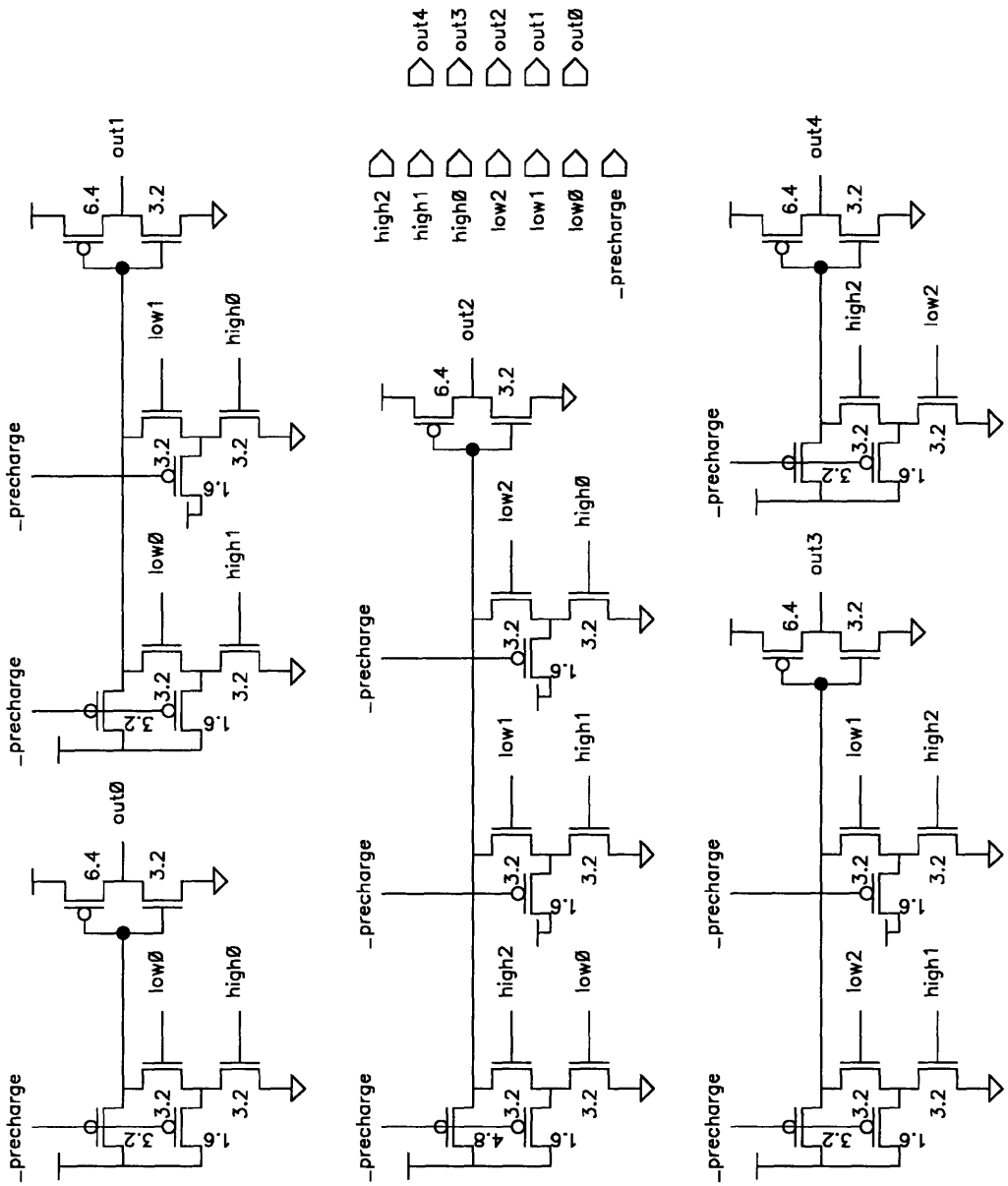


Figure D-4: Transistor Schematic of Allocate 4-Counter

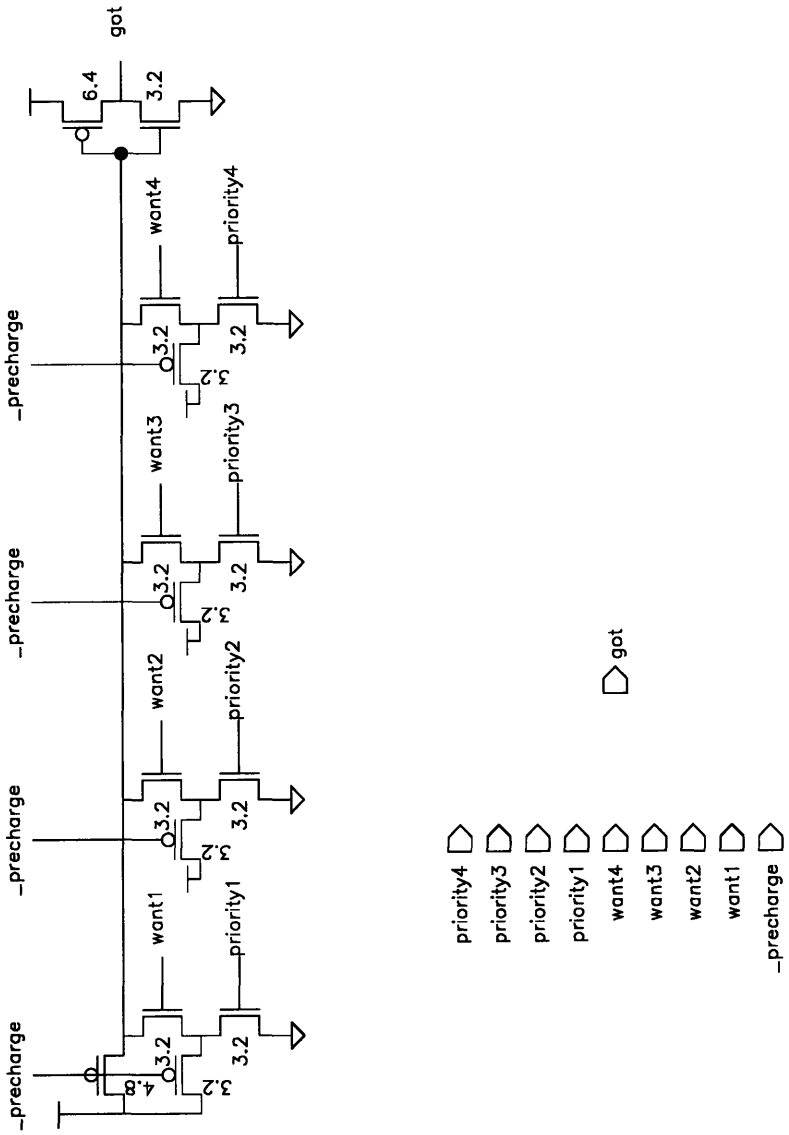


Figure D-5: Transistor Schematic of Allocate Got

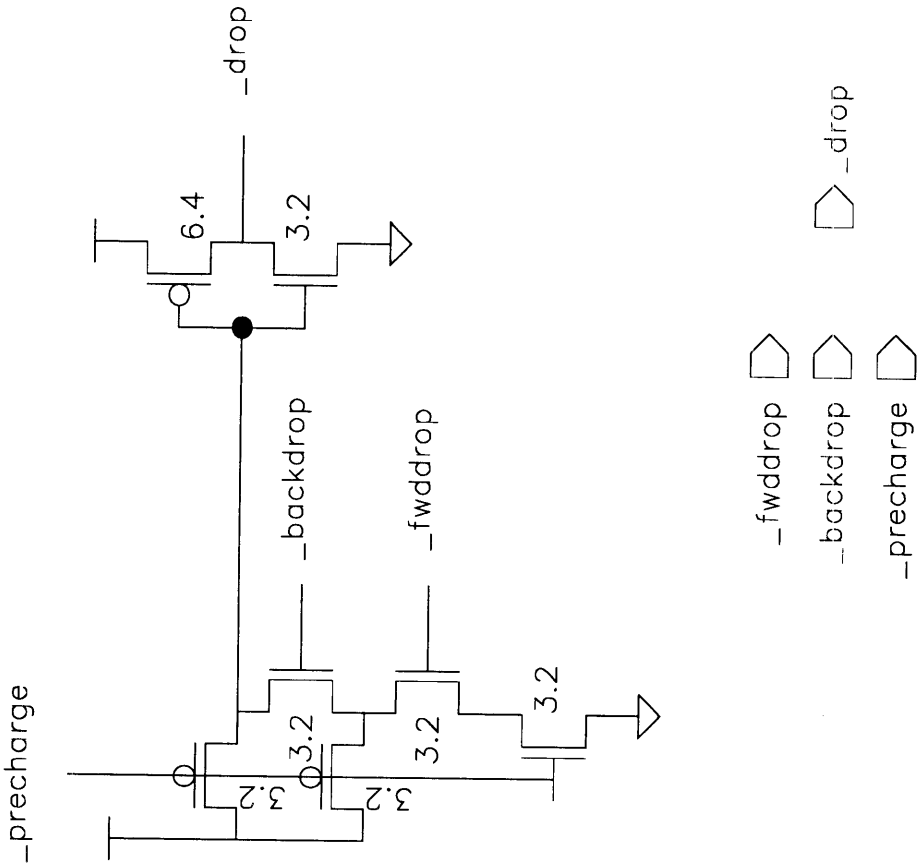


Figure D-6: Transistor Schematic of Allocate Drop

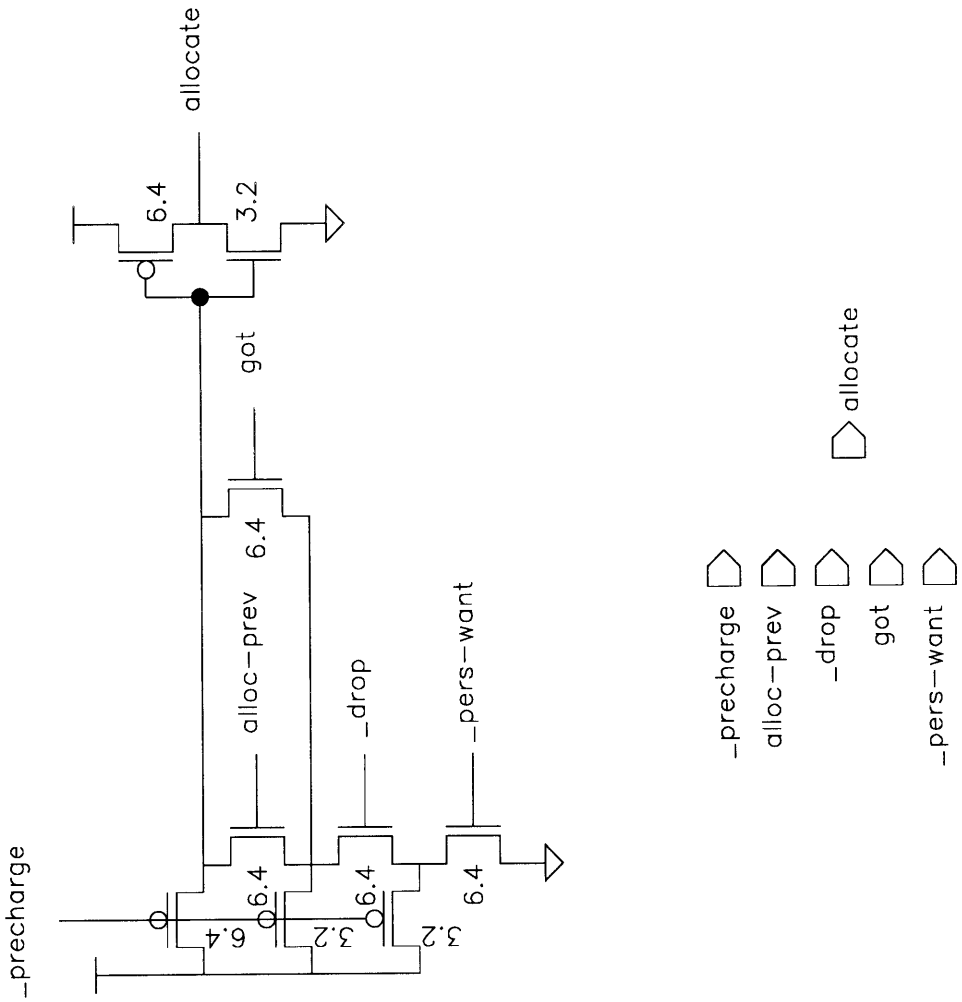


Figure D-7: Transistor Schematic of Allocate Final

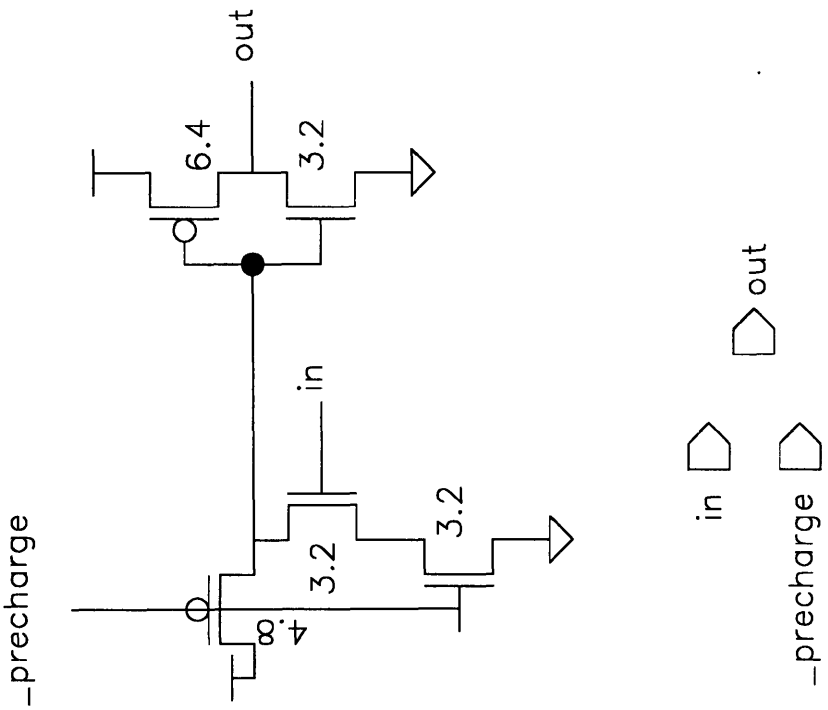


Figure D-8: Transistor Schematic of Allocate Precharge

Appendix E

Clock Section

Figure E-1 shows the block diagram for the clock section. This section creates the clock signals which are fed to all the latches and logic segments. In order to test the new logic style mentioned earlier, the control of these signals is very important. Nine input signals control the frequency, pulse widths, and separation between pulses of these special clock signals.

The leftmost block, *clock-gen*, generates a half-duty cycle clock signal at the desired frequency. The transistor schematic is shown in Figure E-2. It consists of a ring oscilator, control muxes and a reset circuit. The muxes, *clock-mux*, allow the user to set the number of inverters in the ring and hence the rate of oscillation. A diagram for the mux is shown Figure E-3.

Even though the internal logic can run around 300 MHz, the pads and packaging cannot. In order to determine the maximum clocking speed, the internal clock frequency remains high. For measurement purposes, the clock is divided by 1024 and run off-chip. *Clock-divider*, shown in Figure E-4, does this. Each stage, flip-flop and inverter, divides the frequency by a factor of two. The schematic for the flip-flop (it is labeled *latch-rising-ff*) is shown in Figure E-5.

The next stage, *clock-pulse-gen*, shown in Figure E-6, creates two pulsed clock signals from the half duty cycle clock. The block creates four versions of the first clock signal, evaluate, which pulses on the rising edge of the half duty cycle clock. The second set of pulse signals, $\overline{precharge}$, are delayed versions of evaluate. The desired pulse width is selected from among the four by selecting it with the final set of muxes.

If further delay between pulses proves necesary, the final block, *clock-delay*, can delay the two clock signals with respect to eachother. As shown in Figure E-7, the muxes select the

number of inverter delays to be added to each signal's path.

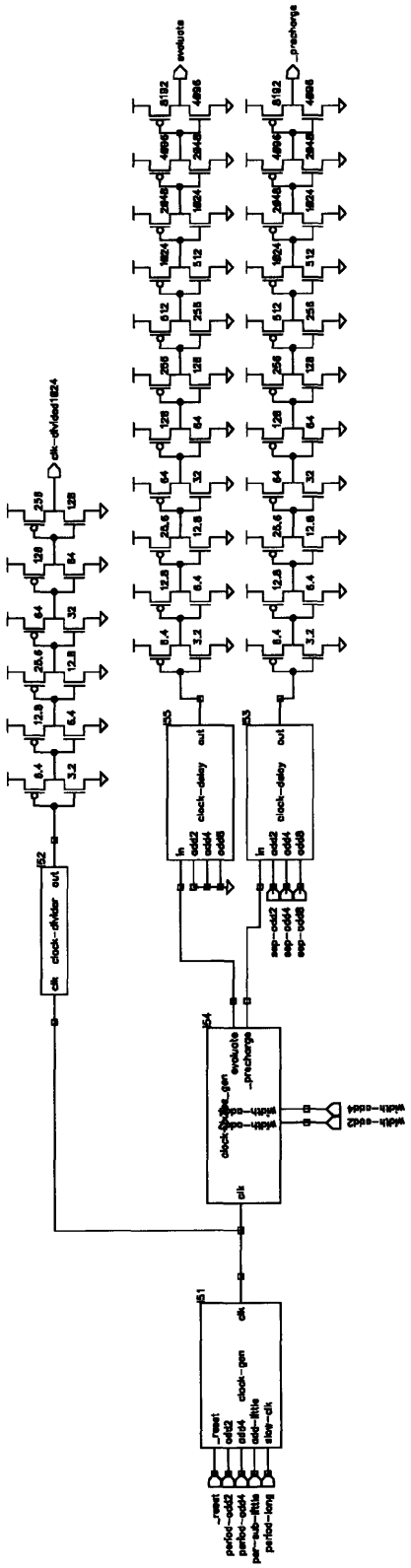


Figure E-1: Block Diagram of the Clock Section

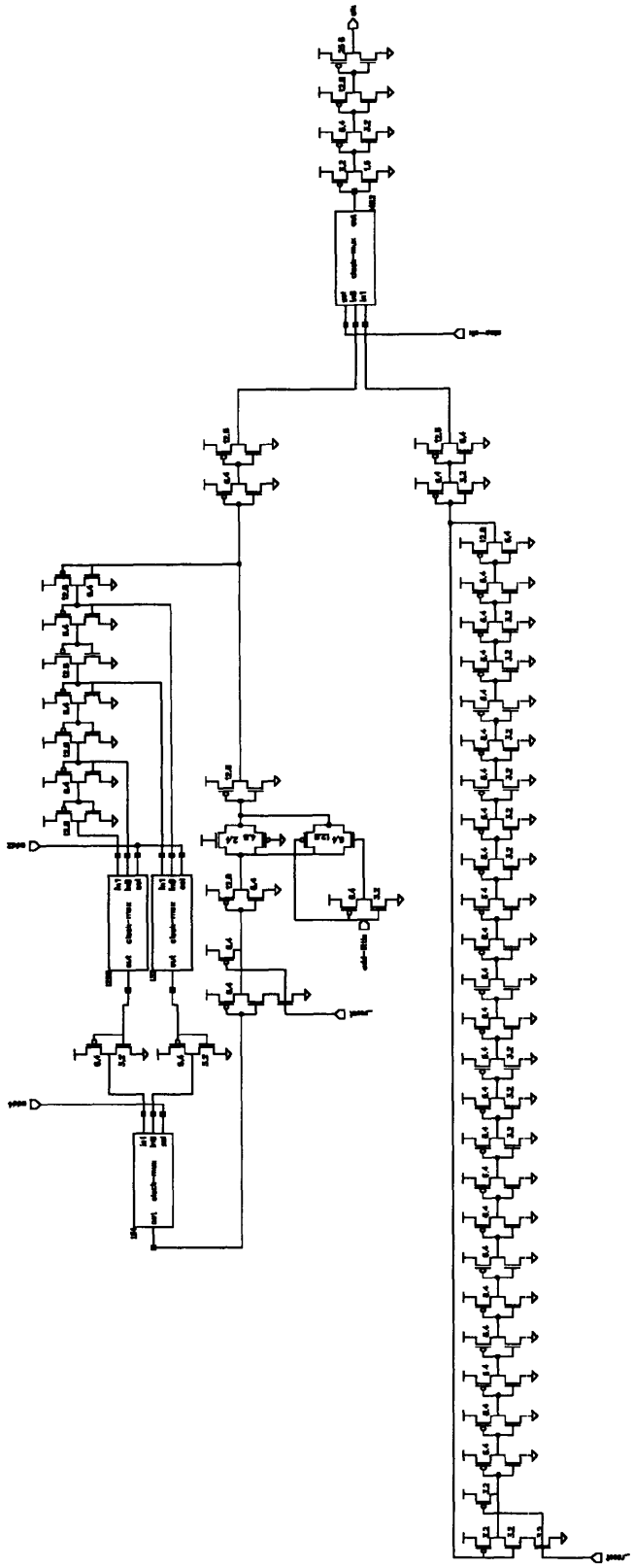


Figure E-2: Transistor Schematic of Clock Gen

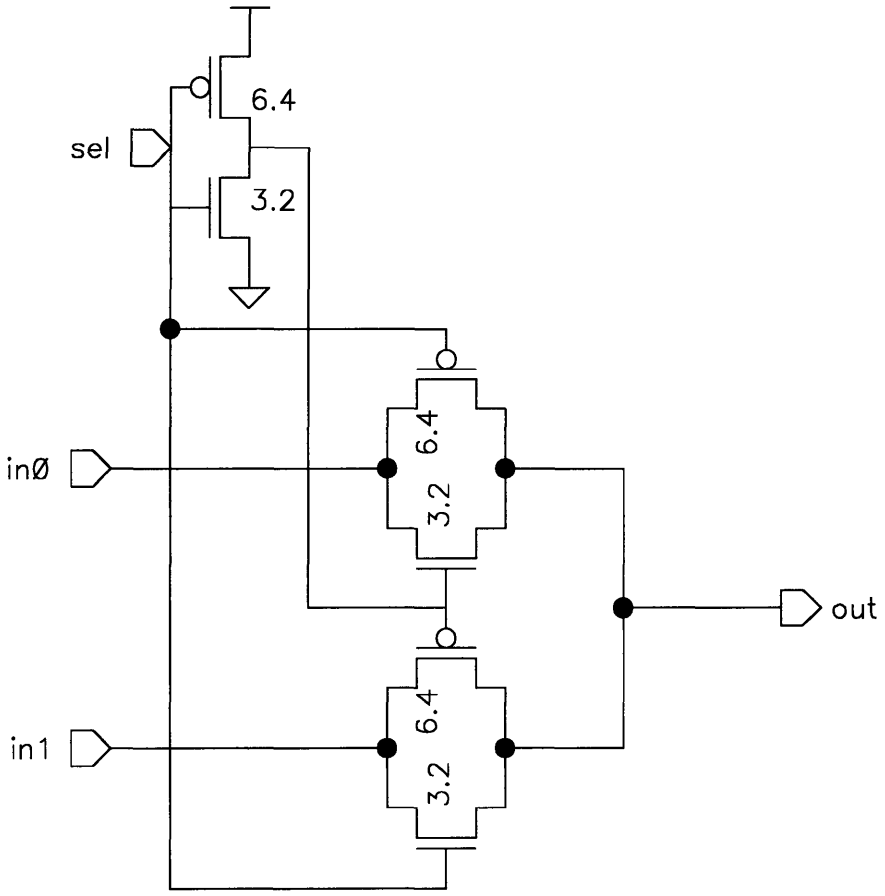


Figure E-3: Transistor Schematic of Clock Mux

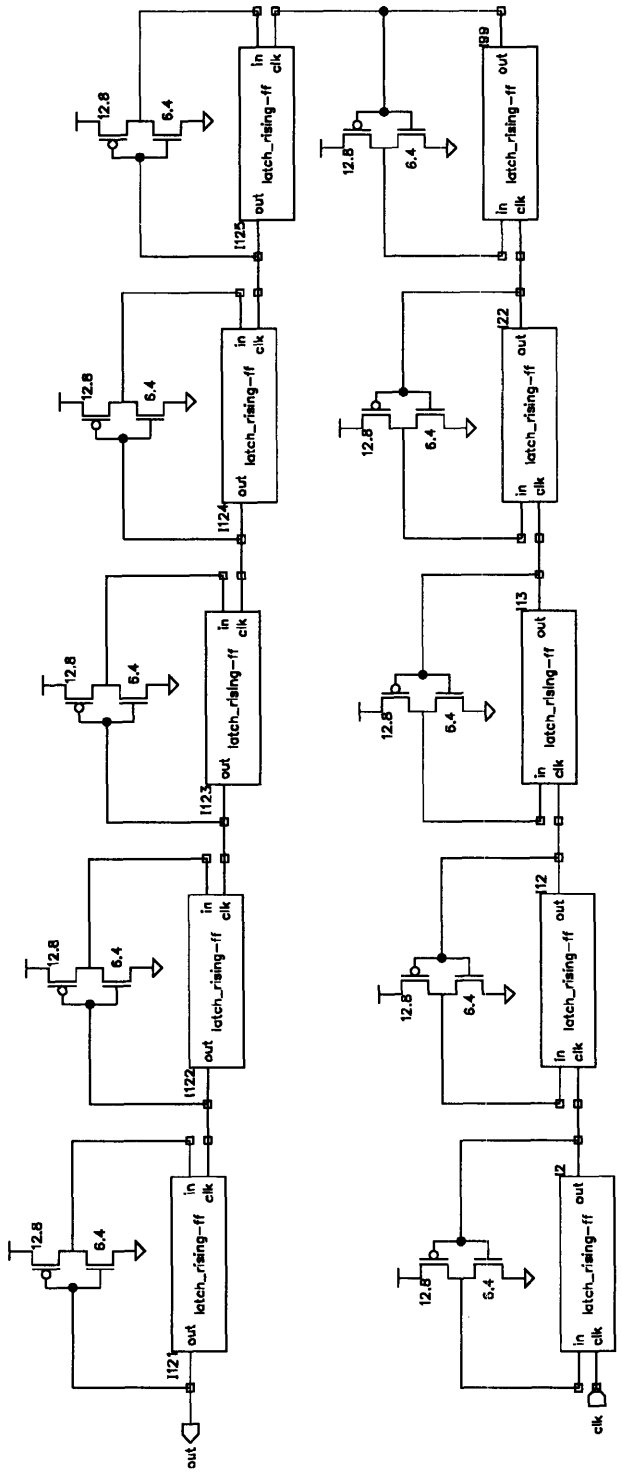


Figure E-4: Transistor Schematic of Clock Divider

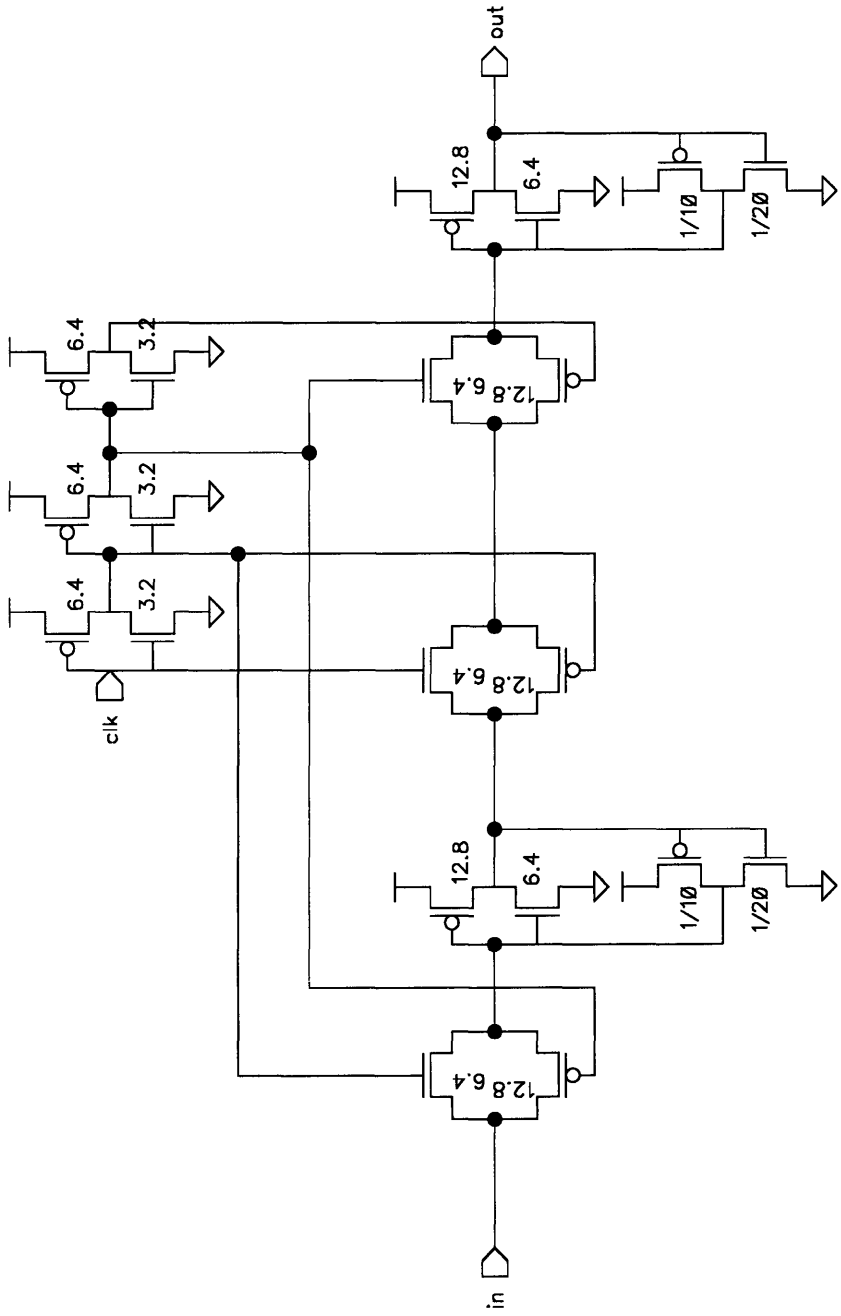


Figure E-5: Transistor Schematic of Latch Rising Ff

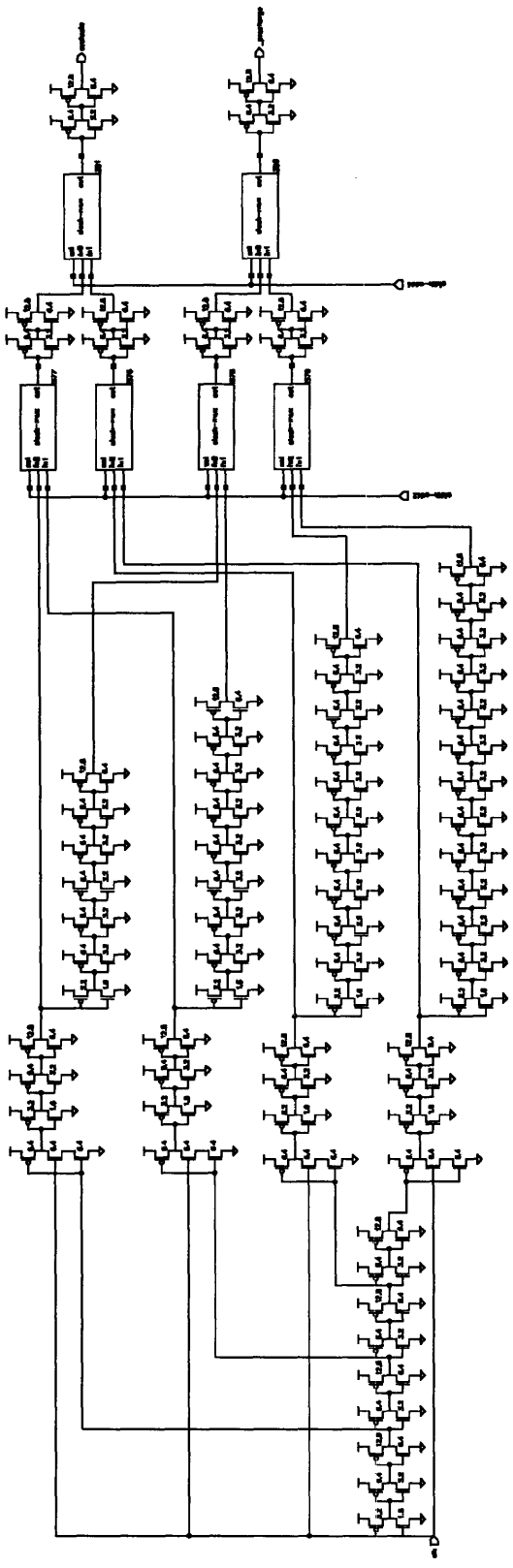


Figure E-6: Transistor Schematic of Clock Pulse Gen

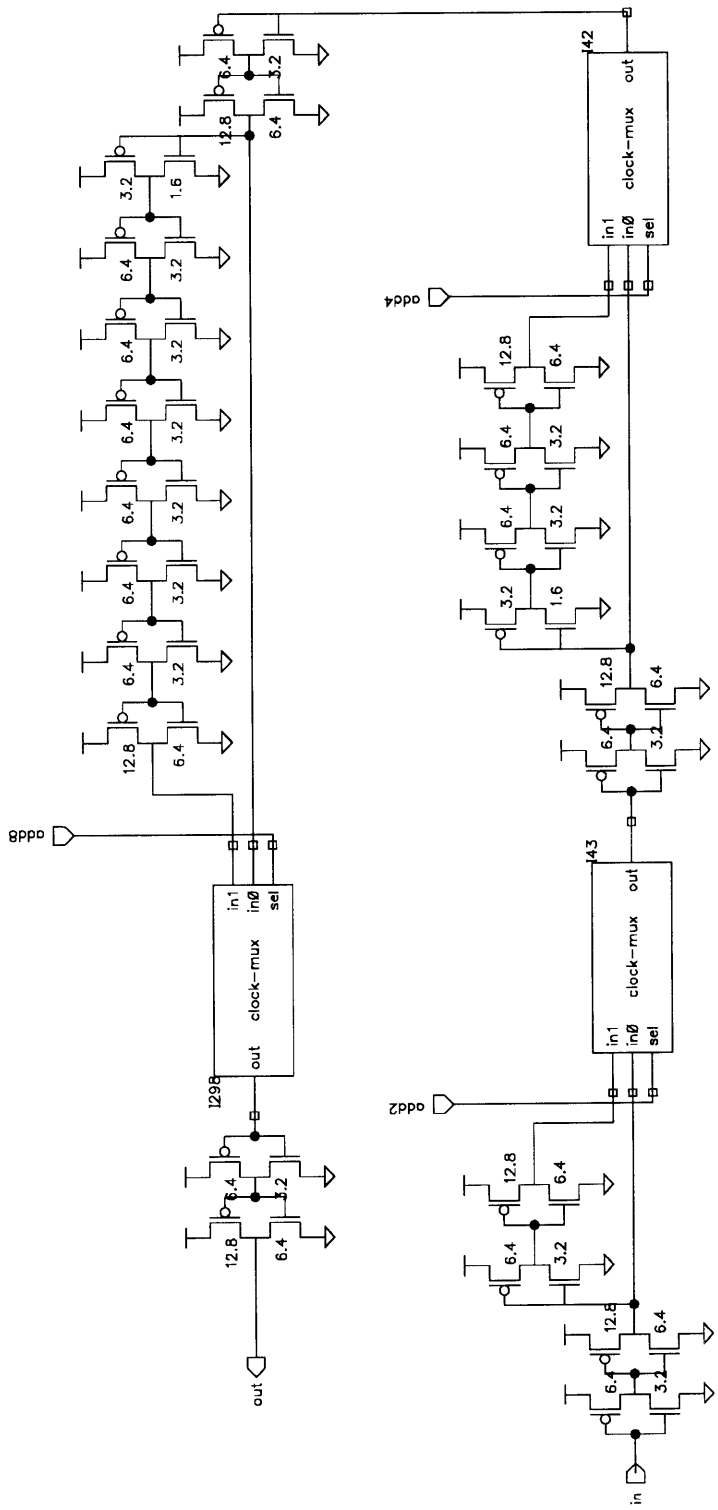


Figure E-7: Transistor Schematic of Clock Delay

Appendix F

Test Section

This chapter *briefly* describes the test logic. The section with a simplified block diagram is shown in Figure F-1. The purpose of the section is to load test vectors into an array of registers at some slow speed, run the test vectors at high speed, store the important outputs from the internal logic, and finally dump the data back to the user.

The array of registers function in two modes. The first is in snake mode, where the muxes are asserted and values snake from the input pin through the register array. The second mode is where each row operates in parallel, forcing inputs into the internal logic and loading outputs from the internal logic.

The fsm with a generalized state diagram is shown in Figure F-2. Each of the states shown is actually implemented as a collection of a few states to provide the appropriate timing. However, the basic idea is apparent.

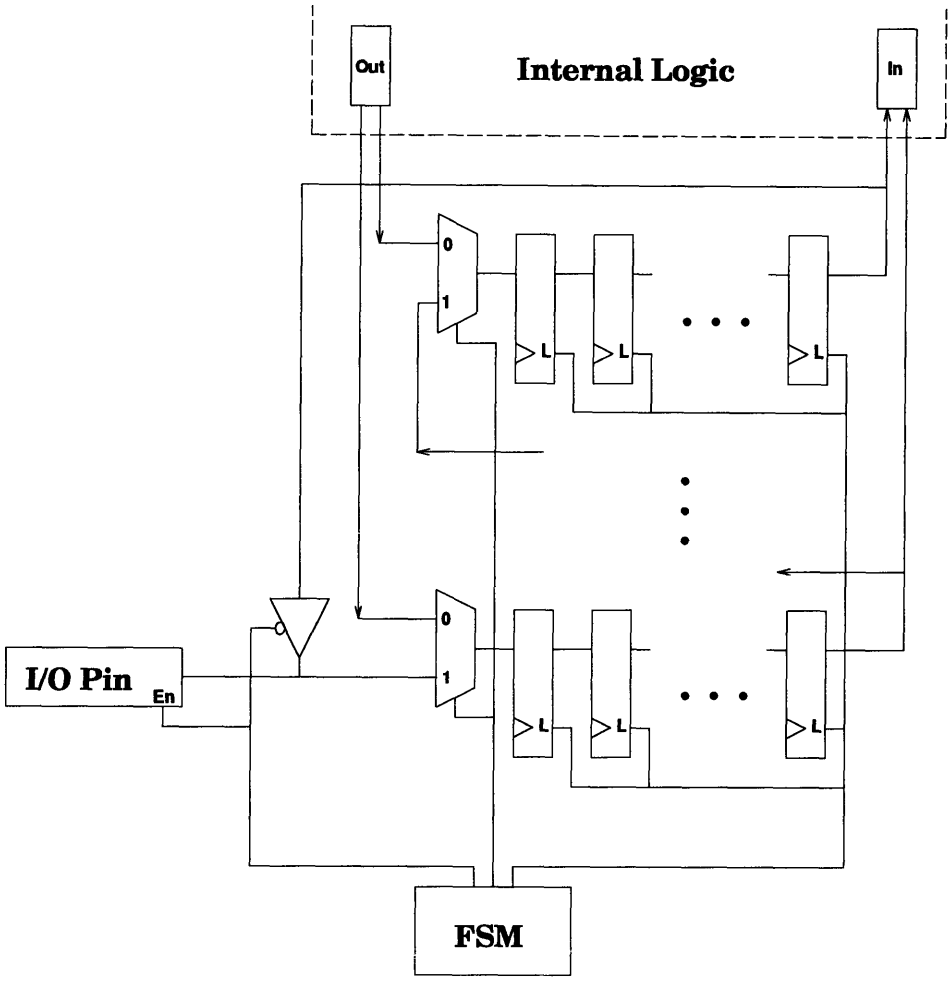


Figure F-1: Generalized Block Diagram for Test Section

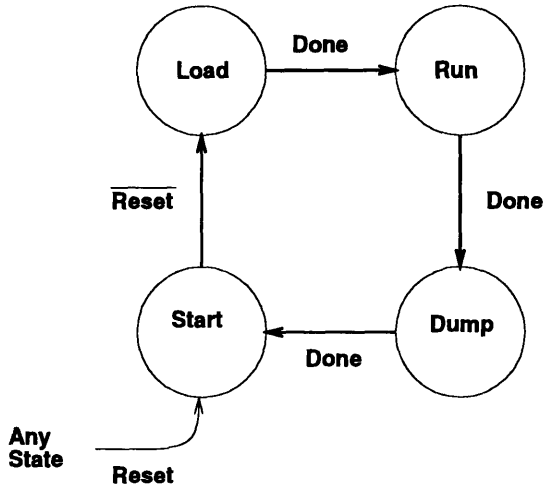


Figure F-2: Generalized State Diagram for Test FSM

Bibliography

[1] S. Arora, T. Leighton, and B. Maggs, "On-line Algorithms for Path Selection in Nonblocking Networks," Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, pp.149-158, May 1990.

[2] Thomas F. Knight Jr. and Patrick G. Sobalvarro, "Routing Statistics for Unqueued Banyan Networks," AI memo 1101, MIT Artificial Intelligence Laboratory, September 1990.

[3] Tom Leighton and Bruce Maggs, "Expanders might be Practical: Fast Algorithms for Routing around Faults on Multibutterflies," IEEE 30th Annual Symposium on Foundationsof Computer Science, 1989.

[4] Fred Chong, Eran Egozy, and André DeHon, "Fault Tolerance and Performance of Multipath Multistage Interconnection Networks," Advanced Research in VLSI and Parallel Systems 1992, pp. 227-242, March 1992.

[5] L. A. Bassalygo and M. S. Pinsker, "Complexity of Optimum Nonblocking Switching Networks without Reconnections," Problems of Information Transmission, 9:64-66, 1974.

[6] George Adams and Howard Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," Transactions on Computers, vol. c-31, no. 5, pp 443-454, May 1982, IEEE.

[7] Henry Minsky, André DeHon, and Thomas F. Knight Jr. "RN1: Low-Latency, Dilated, Crossbar Router," Hot Chips Symposium III, 1991.

[8] Y. Tamir and H. C. Chi, "Symmetric Crossbar Arbiter for VLSI Communication Switch," Transactions on Parallel and Distributed Systems, vol. 4, no. 1, pp. 13-27, January 1993, IEEE.

[9] Nelson F. Gonclaves and Hugo J. DeMan, "NORA: a Racefree Dynamic CMOS technique for pipelined logic structures," IEEE JSSC, vol SC-18, no.3, June 1986, pp.261-266