VLSI Mesh Routing Systems

Charles M. Flaig

Computer Science Department
California Institute of Technology

5241:TR:87

VLSI
Mesh Routing
Systems

by

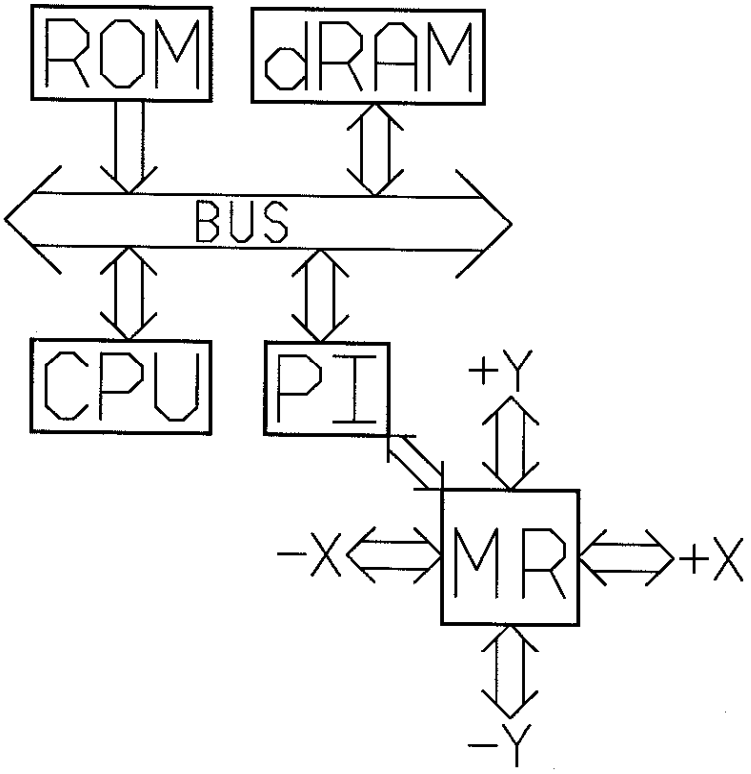
Charles M. Flaig

In partial fulfillment of the Requirements for the
Degree of Master of Science

May 1987

5241:TR:87

California Institute of Technology
Computer Science



The research described in this thesis was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597, and in part by grants from Ametek Computer Research Division and from Intel Scientific Computers.

VLSI Mesh Routing Systems

Contents:

1. Message Routing Networks

1.1 Background

For message-passing concurrent computers with very few nodes, it is practical to use a full interconnection scheme between nodes. A full interconnection of channels quickly becomes impractical as the number of nodes increases since each node of an N node machine must have $N-1$ connections. A configuration used for larger message-passing multicomputers such as the Caltech Cosmic Cube [Seitz 85] and its commercial descendants is that of a binary n-cube (or hypercube) to connect $N = 2^n$ nodes. Each node has $n = log_2 N$ connections, and a message never has to travel through more than n channels to reach its destination.

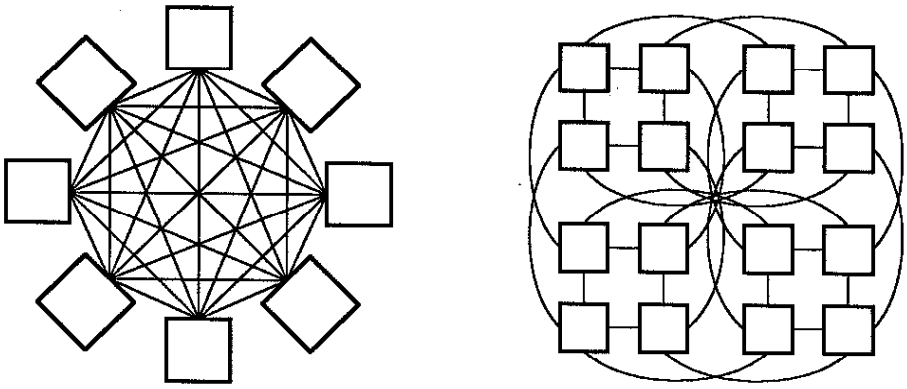


Figure 1: Full interconnection and hypercube channel configurations

Although the choice of the binary n-cube for the first generation multicomputers is easily justified, the analyses presented by a 1986 Caltech PhD thesis by William J. Dally [Dally 86] show that the use of lower dimension versions of a k-ary n-cube [Seitz 84a] connecting $N = k^n$ nodes – *eg*, an $n = 2$ (2-D) torus or mesh – is optimal for minimizing message latency under the assumptions of:

- constant *wire* bisection.
- "wormhole" routing [Seitz 84b].

These 2-D (or optionally 3-D) networks also have the advantage that each node has a fixed number of connections to its immediate neighbors, and if the nodes are also arrayed in two or three dimensions, the projection of the connection plan into the packaging medium has *all* short wires. Also, the number of nodes in a machine can be increased at any time with a minimum amount of rewiring. The low dimension

k-ary n-cube greatly decreases the number of channels, so that with a fixed amount of wire across the bisection, one may use wider channels of proportionately higher bandwidth. This higher bandwidth, particularly with wormhole routing, can more than compensate for the longer average path a message packet must travel to reach its destination.

The time required for a packet to reach its destination in a synchronous router is given by:

$$T_n = T_c(pD + \lceil L/W \rceil)$$

where T_c is the cycle time, p is the number of pipeline stages in each node, D is the number of channels that a packet must traverse to reach its destination, L is the length of the packet, and W is the width of a flow control unit (flit).

As an example, let us assume we have $N = 256$ nodes, 512 wires crossing the bisection for communication (neglecting overhead from synchronization wires), a message length of 20 bytes, and an internal 2-stage pipeline. The bisection of a binary hypercube has 128 channels in each direction, each with a width of 2 bits, and an average of $(log_2 N)/2 = 4$ nodes must be traversed, so $T_n = (2 \times 4 + 160/2)T_c = 88T_c$. The bisection of a 2-D $(k \times k)$ mesh, where $k = 16$, has 16 channels in each direction, each with a width of 16 bits, and an average of $(2k/3) \approx 11$ nodes must be traversed, so $T_n = (2 \times 11 + 160/16)T_c = 32T_c$. The binary hypercube network in this example thus has over twice the average latency of the bidirectional mesh network with the same wire bisection.

The Torus Routing Chip (TRC), designed at Caltech in 1985 [Dally & Seitz 86], used unidirectional channels between nodes connected in a torus:

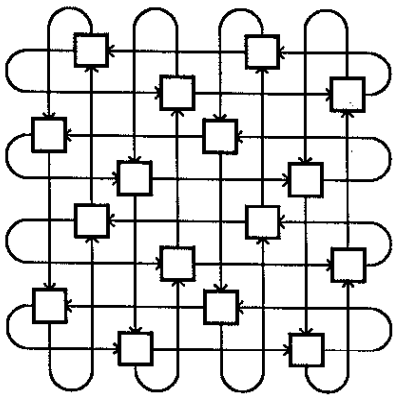


Figure 2: Torus network

The torus is shown here *folded* in its projection onto the plane in order to keep all channels the same length. Deadlock was avoided by using the concept of virtual

channels, by which a packet injected into a network travels along a spiral of virtual channels, thus avoiding cyclic dependencies and the possibility of deadlock. The TRC was self-timed to avoid the problem of delivering a global clock to a large network. There were a total of 5 channels to deal with, channels to and from the node, and 2 virtual channels each in x and y. Thus the heart of the TRC involved a 5×5 crossbar switch. Although the initial version had a slow critical path, the revised version was expected to operate at 20MHz, with a latency from input to output of 50ns. Since each channel had 8 data lines, the TRC achieves a data rate of 20MB/s. A *packet* is made up of a header, consisting of 2 bytes containing the relative x and y address of the destination, any number of non-zero data bytes, and a 0 data byte signifying a tail. Upon entering the router, each packet has the address in its header decremented and tested for 0, and is passed out through the proper output channel. The connection stays open for the rest of the message and closes after the passage of the tail (wormhole routing). If the desired output channel is unavailable, the message is blocked until the channel becomes available.

In the winter and spring of 1986, concurrently with the developments described above, groups of students in the Caltech "VLSI Design Laboratory" project course were put to work designing different parts of the Mosaic C element. This single-chip node of a message-passing multicomputer was to contain a 16-bit processor, several KBytes of on-chip dRAM, and some sort of routing circuitry for communication with other chips. Each chip would form a complete node in a fine grain concurrent computer.

After looking at a few possible implementations, including the TRC, the group working on the routing section decided that a simple bidirectional 2-D mesh would be used. The mesh had the advantage of keeping the length of wires between chips down to less than 1 inch, which allowed the use of a synchronous protocol, since clock skew could be made very small between chips. The mesh also allowed the channels at the edge of the array to be reserved for communication with the outside world. The group also decided to use a bit-serial protocol for packets, both to minimize the number of pins on each chip and to minimize the number of connections needed between them, but to organize the packets into sufficiently large flits that all of the routing information could be contained in the first flit. As in the TRC this first Mosaic C router used virtual channels to avoid the possibility of deadlock. Each packet consisted of a 20 bit header with the relative x and y addresses of the destination, and an arbitrary number of 20-bit flits consisting of a 16-bit data word and 4 control bits. The router also used wormhole routing with one of the control bits signifying a tail. Internally, flits were switched between input and output channels using a time multiplexed bus. The control circuitry was kept as simple as possible, and as a result did not know how to forward a packet by itself. Each time the header of a packet came in, the processor would be interrupted (using

a dual-context processor for fast interrupt handling) to determine which output channel the packet should be connected to. This approach resulted in a latency of several μs per step in path formation, but allowed a lot of flexibility in routing under software control. Acknowledgement packets would automatically be sent and received between chips using the same channels to announce the availability of buffers. With a 20MHz system clock (expected for 2μm SCMOS technology) the bandwidth would be about 2MB/s on each channel.

A layout was not finished for this first try at a routing circuit for the Mosaic C, and I started working to complete it over the summer 1986. In retrospect, this router consumed a large amount of silicon area to achieve fairly dismal performance. However, two new ideas about packet routing appeared, both due principally to my advisor, Chuck Seitz, and my project turned into one of designing routing circuits of an entirely new type.

The first new idea was the use of prefix encoding to allow the packet header to encode the relative address of the destination on several small successive flits. This scheme was the key to getting around the problem of having to see a large amount of header information before one could decide where to send the head of the packet. This scheme is described in detail in section 1.2. This change simplified the routing circuitry enough to allowed it to handle forwarding automatically without having to disturb the processor. The deadlock-free routing method decided upon was to send packets on the fixed route on a mesh of first x, then y, instead using of virtual channels to avoid deadlock. Initial designs involved 5-bit flits (4 data, 1 control, with an acknowledge wire in the reverse direction) to be sent in parallel on each channel, and to be internally switched using a crossbar switch.

The second new idea goes under the general name of *routing automata*. Since we first started making routers, it was realized that a crossbar switch, while very general, had the disadvantage of taking up a space proportional to $(nW)^2$, where n is the number of inputs and outputs, and W is the number of bits being switched. Now that we were back to a fixed routing scheme, the generality of a full crossbar switch was no longer needed, and it was highly desirable to devise a scheme in which the area would only increase linearly with nW, or as close to that as possible. This scaling would make it much easier to modify the router for more dimensions or wider flits without involving major layout changes, would decrease the path length in the switch and hence increase its speed, and would hopefully decrease the overall area for designs with wide flits and a large number of dimensions.

Each of these automata would be responsible for switching the packet streams for 1 dimension of the router. An automaton's input would consist of streams from the + and − directions as well as from the previous dimension, and its output would consist of streams to the + and − directions as well as to the next dimension. For n

dimensions, n of these automata could be strung together in series, and if properly constructed, their size would increase roughly linearly with increased width of the flits, for a net increase in area proportional to nW.

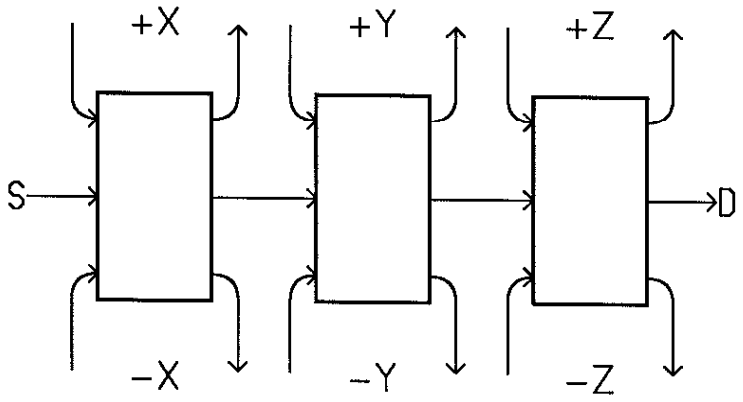


Figure 3: 3 Automata connected for x, y, and z routing

A more detailed description of these routing automata will be given in section 1.3. A synchronous automaton fitting this description was designed for the Mosaic C, and it will be described in detail in Chapter 2.

Work was also begun on a self-timed version of routing automata of this type. This version is intended to have each of its components be highly modular so that they could be used not only to implement mesh routing, as in the Mosaic C, but could also be fit together to implement unidirectional routers, routers for hypercubes, or many other structures limited only by the desires of the designer. The basic components included a FIFO for glue between stages and buffering, a switch used both to divide and merge data streams, a decrementer for adjusting the relative address of the destination as the packet passes through, and control structures for all of the above. The implementation of these routing automata will be discussed in Chapter 3.

1.2 Prefix Encoding

The prefix encoding scheme allows packets to travel through nodes at a constant rate, with the first flit of the header generally containing enough information to determine the output channel. The scheme involves the use of a *leading zero* flit that can be used to limit how much of the relative address needs to be looked at before a decision can be made or the address decremented. The following example uses 3-bit flits (2 data, 1 control), which is the minimum width that allows the encoding of the necessary "alphabet" of symbols, which are +,-,.,T,0,1,2,3. Each line represents the packet as it leaves the node listed to the left, with "source" indicating the node sending the packet and "destination" indicating the node receiving the packet. Time flow is towards the left. As header flits are no longer needed, they are stripped off.

The diagram below follows the packet's path through the 2-D network, as shown in figure 4.

```
+        control, turn to + direction
-        control, turn to - direction
.        control, leading zero
T        control, tail
0...3    radix-4 relative address
M        data flit, one of the symbols 0...3

source                              T M M M M . 3 + 1 2 +
node 1                            T M M M M . 3 + 1 1
node 2                          T M M M M . 3 + 1 0
node 3                        T M M M M . 3 + . 3
node 4                      T M M M M . 3 + . 2
node 5                    T M M M M . 3 + . 1                <--time
node 6                  T M M M M . 3 + . .
node 7                T M M M M . 2
node 8              T M M M M . 1
node 9            T M M M M . .
destination       M M M M
```

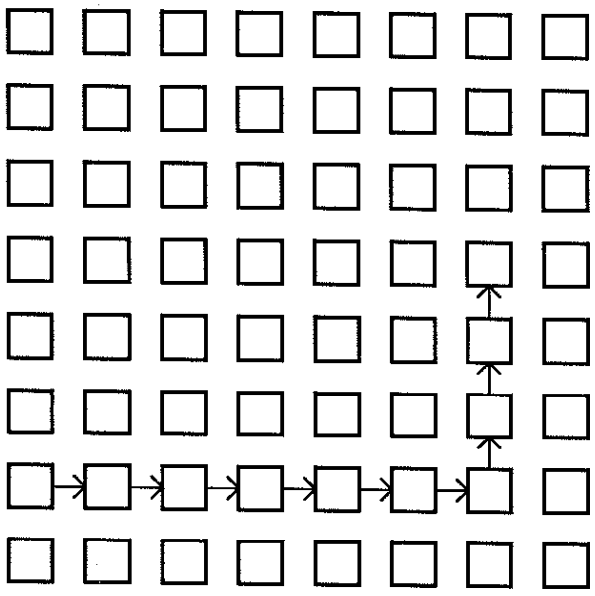


Figure 4: Packet's path through 2-D network

Originally the packet enters the network and takes the + direction in the first (x) dimension. The leading flit is decremented while passing through each node until

it reaches 0. During the decrement to 0 in node 2 the following flit is examined to see if it is a digit. In this case it is, so the leading flit becomes 0. In node 3, the second flit needs to be decremented to 0, but since it is not followed by a digit, it becomes a leading zero. This process continues until the relative address is completely decremented to 0, at which point the leading zeroes are stripped off. The next flit forces the packet to take the + direction in the second (y) dimension, and once again the relative address in the header is decremented to 0. At this point the packet has run out of dimensions to traverse, so it is passed into the receiving node. The tail which makes up the end of the packet closes all of the channel connections as it passes through the nodes, and is finally stripped off at the destination.

This encoding scheme allows the use of small flits to represent large offsets while allowing decisions to be made based only on two flits of the header at once, which helps minimize the latency of forwarding through a node. The simple decisions involved also allow a simple controller to be used.

1.3 Routing Automata

As previously mentioned, the reduction of the routing circuitry to simple automata that control the switching through only one dimension greatly simplifies the modification and expansion of a complete router. An individual automaton is also much easier to design and lay out, due to the reduced number of inputs and outputs, and independence from the routing occurring in other dimensions.

The 1-D automaton:

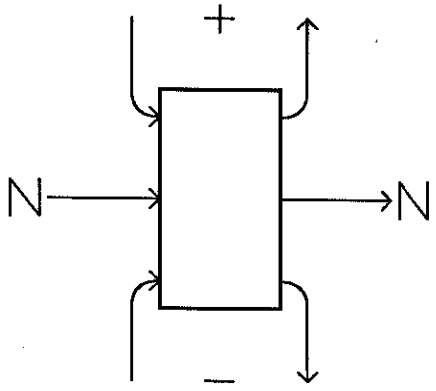


Figure 5: 1-D automaton

has three inputs consisting of packets:

(1) traveling in the + direction,

(2) traveling in the − direction, and

(3) from the previous dimension.

Simple finite state machines can then process the input streams, decide on a switch configuration that allows the largest number of packets to be forwarded, and then connect the streams:

(1) to the + direction,

(2) to the − direction, and

(3) to the next dimension.

Now, here is a useful trick. Each of these automata can further be broken down into a series of decision and merge operations performed on a subset of the data streams. The following illustrates one possible conceptualization:

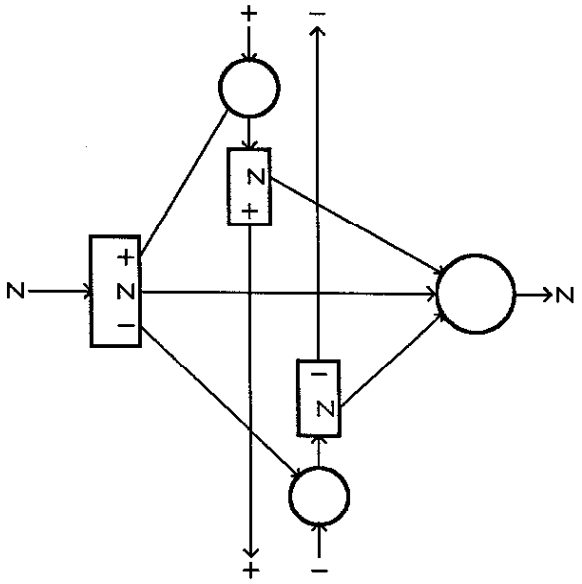


Figure 6: Sample internal structure of automata

The boxes in figure 6 represent decision elements that process their incoming data streams and switch them onto their proper output stream. The circles represent merge elements that take their input streams and arbitrate which of them to connect to their output stream. Thus a packet coming from the previous dimension that is to exit in the + direction would enter the leftmost decision element, be switched onto its upper output stream, and merge into the stream exiting in the + direction.

Breaking down the internal structure of the automata in this way can further simplify the design and layout, in the same manner as breaking up the router into the 1-D automata. Even when parts cannot be directly reused, time can often be saved by employing modifications. In the extreme case, each of the decision and merge operations can be converted into binary form, where 3-way elements are replaced by cascaded binary elements. In this case the elements become very

homogeneous and the automata can be formed out of a minimal subset of very simple elements. This approach is the one used in the self-timed routing automata discussed in Chapter 3.

These automata can also be constructed for different channel configurations using the same set of internal elements. For example they can be constructed for unidirectional channels as used in a torus or a hypercube. Examples of the internal structure of such automata are shown below:

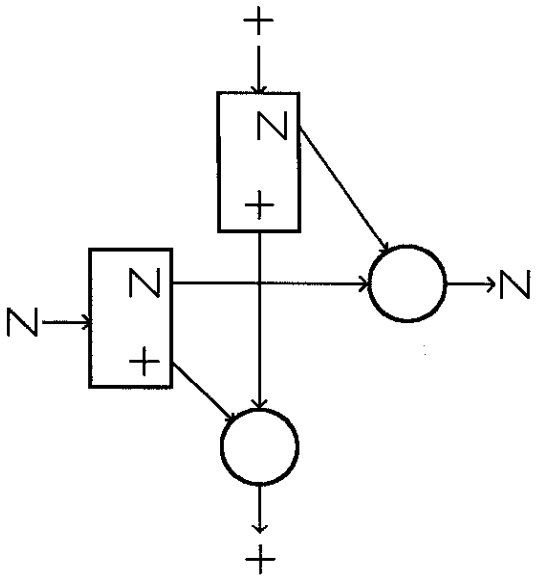


Figure 7: Unidirectional torus routing automata

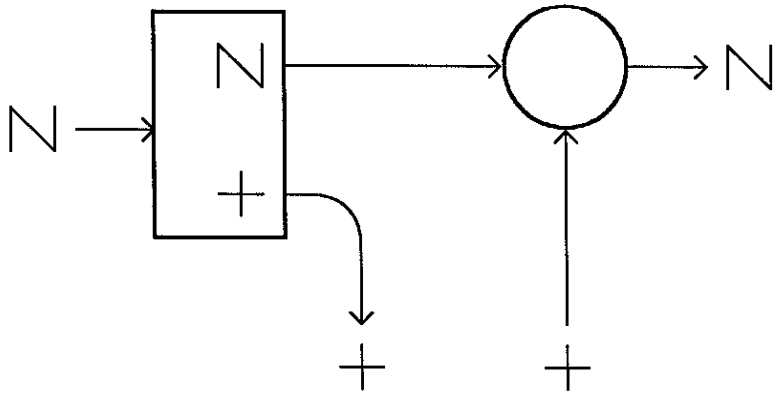


Figure 8: Hypercube routing automata

2. The Mosaic Router

2.1 Overview

The Mosaic chip is a complete node of a fine-grain concurrent computer. It contains all of the necessary elements including a 16-bit processor, several KBytes of ROM and dRAM, and routing circuitry for communicating with neighboring nodes in a mesh. All of these elements are tied to a common bus.

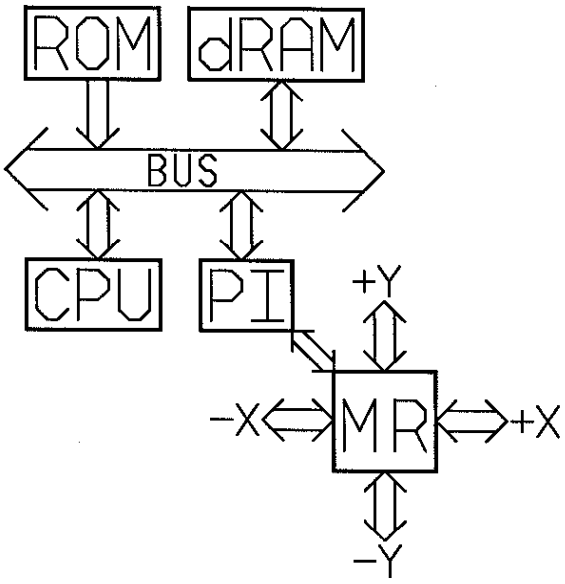


Figure 9: Mosaic chip

Since the router must fit on a chip along with a processor and memory, the design had to be simple and compact. Some inefficiencies that appeared in the design were retained, however, because a functional router was needed before the Mosaic could go into production, and this goal didn't allow time for endless redesign.

The Packet Interface (PI) takes care of encoding the packet header, and transferring packet data to and from memory. A simple cycle-stealing form of Direct Memory Access (DMA) is used to keep up with the high data rates supported by the router. The PI also contains some memory mapped locations that are used to specify the relative x and y addresses of the destination node, and an interrupt control register.

The PI generates the appropriate direction control flits and multiplexes the relative x and y addresses in its registers into the flit width required by the router.

It does the same multiplexing for the packet's data words, which come from an output queue in dRAM. A tail flit is added when the output queue becomes empty. These flits are injected into the previous dimension input of the automaton for the first dimension. Any packets coming out of the last dimension simply have their tail stripped off and the flits are demultiplexed into a 16-bit word, which is then stored in an input queue in dRAM. The processor may be interrupted either when the output queue becomes empty, or when a tail is received, or when the input queue becomes full.

The Mosaic router, that is illustrated here, communicates with other nodes using a 3-bit wide flit (2 data, 1 control) with an acknowledge wire in the reverse direction. The flit can also be made wider to include more data bits. The first "production" Mosaic chips will use a 5-bit flit. Any time an acknowledge is present, flits are allowed to progress through the pipeline.

As a result of the bit-slice design used, it became more efficient in the data path to combine the decision and merge operations slightly differently than in the sample automata shown in Figure 6 of Chapter 1. The actual configuration is shown below:

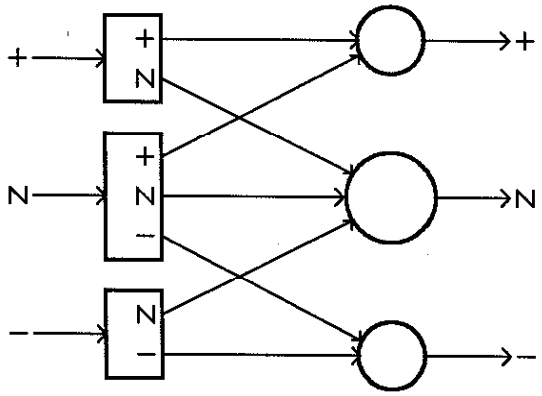


Figure 10: Mosaic routing automata

The following sections will discuss the design and layout of key pieces of the routing automaton in more detail. The last section in this chapter will discuss the processor interface.

The completed layout of the Mosaic routing automata measures $760\lambda \times 1130\lambda$ for a minimum width flit (2 data, 1 control) At least one-third of this area is control circuitry, so it is advantageous to use wider flits as far as layout efficiency is concerned. It is expected to run at the same rate as the processor, which should have at least a 20MHz clock with 2μm SCMOS (10 MB/s with a 5-bit wide flit). A complete automaton was packaged and sent out for fabrication in February, 1987. It came back at the beginning of April, and was tested in early June.

The test router correctly made connections, stripped or decremented portions of headers, and closed the connection after the arrival of a tail. Problems were encountered when a channel was required to block, however, so that garbage appeared on the output and synchronization was lost. The cause of this problem was eventually traced to a mistake in the layout of the acknowledge switch, resulting in short circuits or floating inputs to logic blocks. The test sequences used to simulate the router were also found to be defective, and the two errors happened to cancel, resulting in a "correct" simulation, but an incorrect chip. The layout and test vectors have since been corrected. It wasn't possible to perform speed tests on the test router since taps were placed on a large number of internal nodes to help track down errors, and the taps significantly loaded these nodes.

2.2 The Decision/Merge Switch

The decision and merge operations were lumped together into one switch matrix, which handles both the multiplexing and demultiplexing of the data streams, with 4 minimally encoded control wires and their complements selecting one of the possible switch configurations. There are several possible control combinations that result in floating or random outputs, but, since this occurs only if a possible connection is not used, these outputs cause no problems. A fair amount of work went into encoding the control wires so as to minimize the number of switches and simplify the control circuitry. The resulting layout is quite compact and, being a bidirectional switch, can be used for both the forward connection for data and the reverse connection for the acknowledge signal.

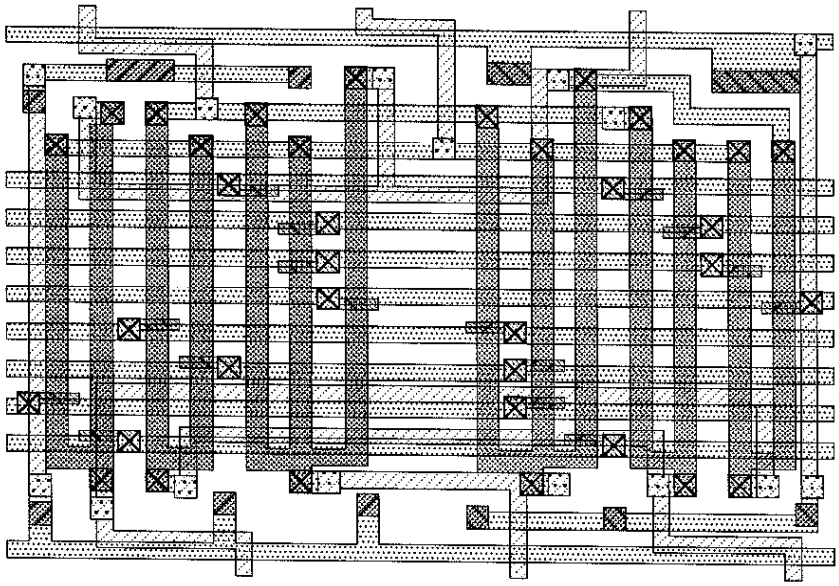


Figure 11: Stream switching element

Although the combined switch approach probably minimized the overall size of the router by homogenizing the data path layout for the different channels, it again complicated the control circuitry. It would certainly not be a good approach for wide data path slices, but for 1-bit slice paths it may have been the best approach. It is difficult to say without having an alternate layout for comparison.

2.3 The Data Path

In an attempt to minimize the overhead of extending the width of a flit, the Mosaic router uses an idea proposed by Wen-King Su of constructing the data path out of 1-bit wide slices, with the +, −, and N paths for each bit being placed immediately next to one another. This approach allows the same switching elements to be used no matter how wide the flit is. Unfortunately, it also means that the control signals for all three data paths had to be propagated through all of the elements, and this led to a somewhat larger overhead in wiring than is necessary. It also complicated the layout of the control circuitry since it had to fit in an effectively smaller pitch.

The data path is made up of a number of elements:

(1) the input latch

(2) the input shift register

(3) the zero/tail detect circuitry

(4) the decrementer, or leading zero generator

(5) the stream switching element

(6) the output latch

(7) the output buffer/disabler

When connected sequentially in the order listed, these elements form a complete data path for a Mosaic routing automaton. A bit-slice section of this path, for +, −, and N, including signal wires passing through the slice, is shown below:

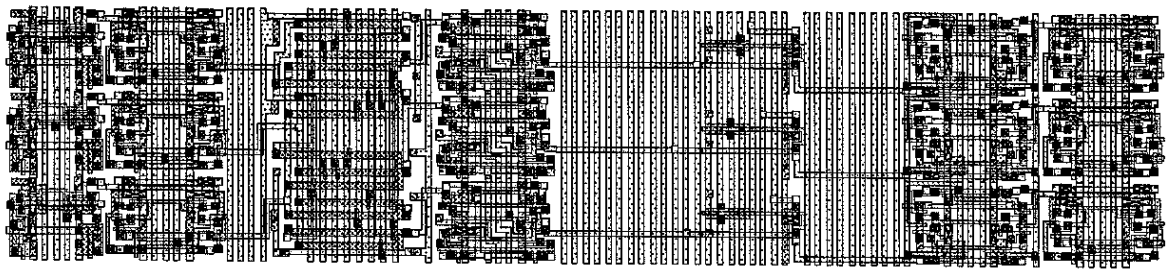


Figure 12: Data path slice for Mosaic router

The direction of data flow in this path is from right to left. One of the most obvious features are the large number of wires running through the center of the path. These wires connect the various data bits in the flit to the control circuitry. Each wire must appear 3 times, once each for the +, −, and N paths. If examined closely, this triplication of control wires can also be seen running down the center of each element in the data path. In addition to requiring more wiring space for this triplication of control lines, the bit-slice approach also causes the bits at one end of the data flit to be located a large distance from the control circuitry, so relatively large drivers must be used for the control lines.

An embarrassment in the design is the location of the decrementers. Rather than placing the decrementers on the input and output paths, I placed them on all three input paths and used some extra logic to decrement only the first flit if the stream was connected to an output path. This placement improved the symmetry of the data paths, but adds more complexity to the control circuitry than is worthwhile.

2.4 The Controllers

Since the design of the automata started with the data path, the control circuitry took up all of the slack for design inefficiencies. As a result, it is rather large and ungainly. The small pitch of the control lines going into and out of the data path necessitated a lot of extra wiring to route sets of signals to the corresponding control circuitry, which could not be placed on the same pitch. In an attempt to minimize this extra wiring, the control circuitry was split into a top half and a bottom half, trying to exploit locality as much as possible. The sections of the top control circuitry, from largest to smallest, are the channel contention and switch control arbiters, the decrementer controllers and carry propagators, the tail detect and propagation controllers, and a few gates to determine the current connectivity state. The bottom half deals mainly with acknowledge propagation and the resulting flow control, since the acknowledge bit happens to be at the bottom of the flit.

One of the requirements for the synchronous routing automata that has been glossed over before, as it depends on implementation, is the need for an acknowledge signal to travel along the same connections as the data, but in the reverse direction. The natural pipelining needed within routing automata complicates this propagation, since the acknowledge bit has to be used for flow control at each stage in the pipeline. Extra care has to be taken, especially when making or breaking a connection, to properly synchronize the interaction of these two streams or glitches will result which can affect the validity of the data stream, or cause the stream to block irreversibly due to losing the acknowledge signal.

2.5 The Packet Interface

Using the minimum sized flit data width of 2 bits and the Mosaic word size of 16 bits, we see that the router can deliver one word every 8 cycles. The data rate becomes even faster if wider flits are used. There is no way that the processor can keep up with this data rate under software control. Therefore it was decided by the Mosaic design group to use a simple form of cycle-stealing DMA to transfer packets between the router and memory.

Four extra registers were added in the processor, and are used as address pointers and limit registers for the input and output channels. Each time the storage bus is not being used by the processor (about once every 3 or 4 cycles for typical code) the microcode PLA emits a bus release signal. A simple finite state machine then arbitrates between bus requests from a refresh counter, the input channel, and the output channel, and grants the bus cycle to one of them. If a channel is given the cycle, it pulls on a line which causes the corresponding address pointer in the processor to be placed on the address bus, and the channel then reads or writes data from that location in the dRAM. The address pointer is then incremented and compared with its limit register. If the two are equal, the DMA is disabled and the processor is interrupted to process the I/O queues. If an interrupt occurs when an output packet word is requested, a tail is sent following that packet.

A packet is sent by setting the output address pointer to the starting location of the desired data in dRAM and setting the limit register to the location of the end of that data. The processor then writes the relative addresses of the destination node of the packet to memory mapped locations in the channel (using sign and magnitude form). When the last location is written, the header is encoded and sent, with the data following. Data is best received by setting the input channel pointer to the starting location of a queue and setting the limit register to the end of the queue. The processor can examine the value of the address pointer at any time to see how many words are in the queue. Currently there is no provision for marking a tail, so if explicit knowledge of the length of a data packet is required, one of two methods must be used. The length of the packet can be encoded in the

first word of the packet, which the processor can then examine, or the processor can be interrupted when the tail of a packet arrives, and then the interrupt routine examines the input address pointer register to determine the length of the packet.

3. The Asynchronous Routing Automata

3.1 Overview

Much of what was learned from the Mosaic synchronous router can be applied to an asynchronous routing automaton. An asynchronous router can be used in physically larger systems, such as second generation "cubes", in which the interconnections are not limited to being very short wires. The mesh routing chip (MRC) described here is designed to meet the specifications for these second generation "cubes".

These routing automata are intended to be a separate chip, a part of a large computing node. As in the Mosaic router, the 2-D MRC has 5 bidirectional channels, with channels in the $+x$, $-x$, $+y$, $-y$ directions, and a channel connecting it to the packet interface.

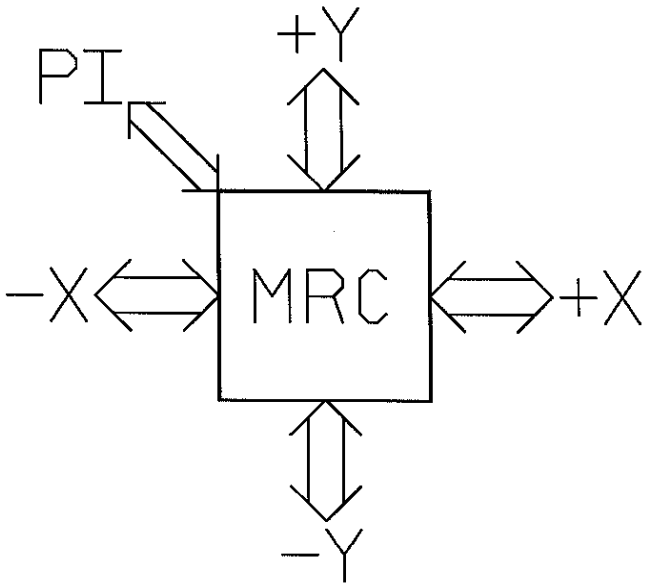


Figure 13: Mesh routing chip

Data is represented on each of the channels using 9-bit wide flits (1 tail, 8 data) as shown in figure 14, where the first bit is the tail bit. The 3rd bit in a header flit is reserved for the future addition of broadcast support. The timing of the wires carrying these flits, as well as the corresponding 2-cycle request and acknowledge signals, is shown in figure 15.

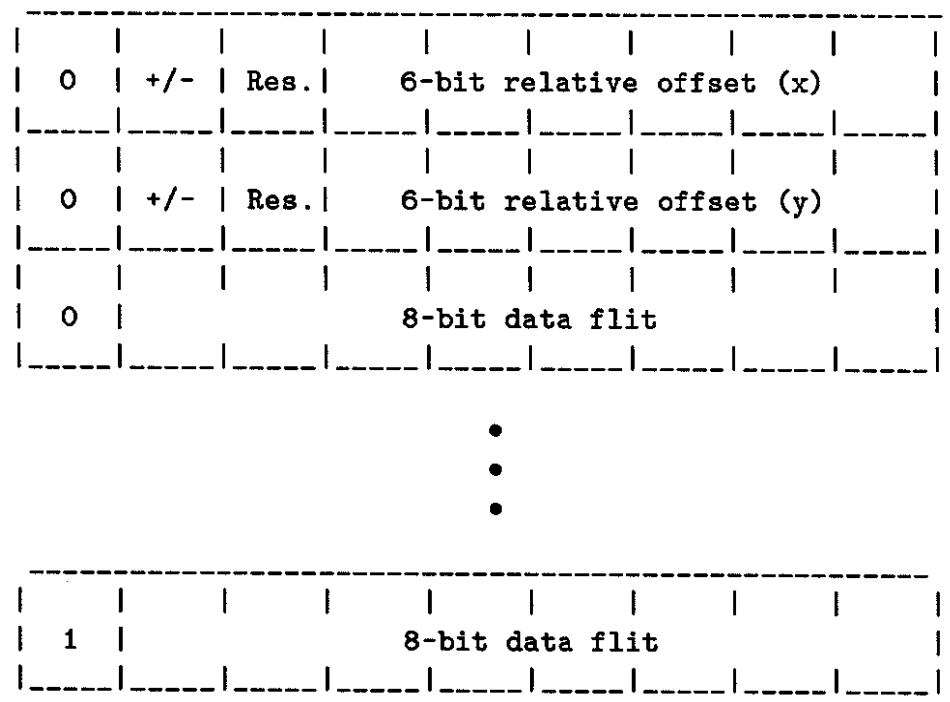


Figure 14: Packet format

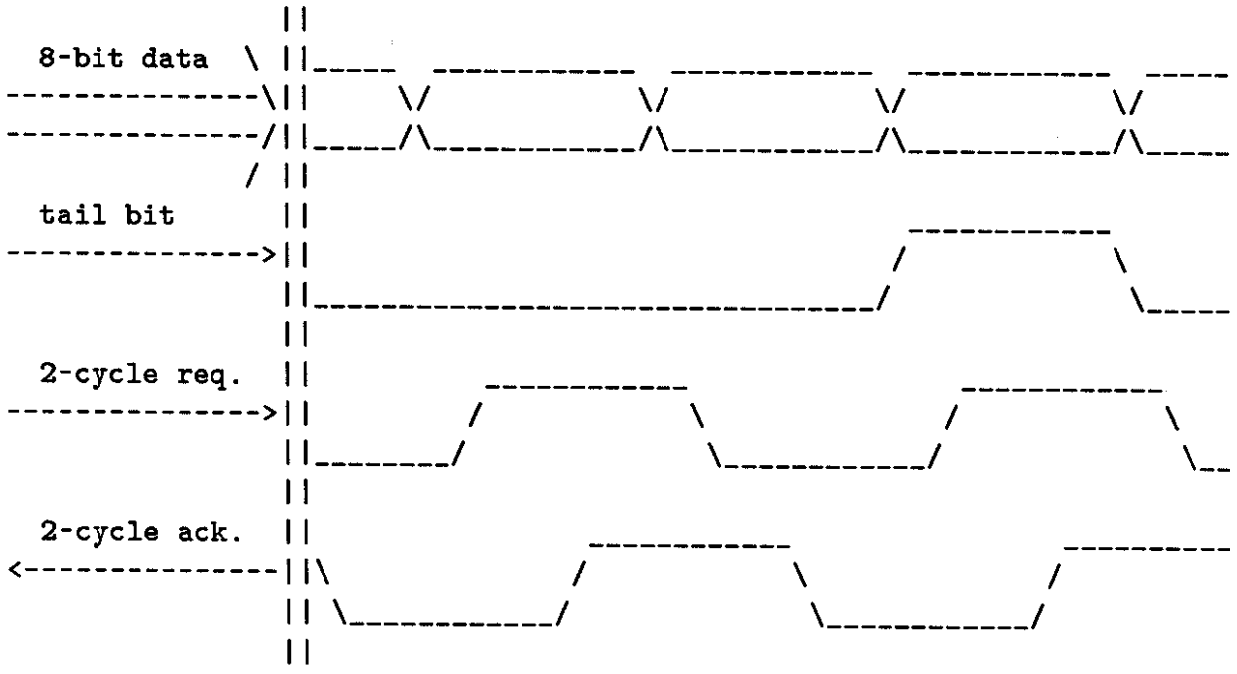


Figure 15: MRC signals and timing

A relative offset of 6 bits allows for up to 64 nodes along a single dimension, which should be sufficient for any 2nd generation machine with large nodes. A 9-bit flit together with the asynchronous request and acknowledge signals for each channel require a total of 11 pins. Five bidirectional channels ($+x$, $-x$, $+y$, $-y$, and the node) then require 110 pins, to be placed in a 132 pin PGA package. The remaining pins are used for a reset and for multiple Vdd and GND pins to minimize noise.

3.2 Pipelining

Pipelining is used in many synchronous systems to increase their throughput. Each cycle, each stage of the pipeline accepts data from the previous stage, performs some relatively simple operation on the data, and passes the resulting data on to the next stage. Data is passed between stages during each cycle by clocked registers. A typical synchronous pipeline section is shown in figure 16.

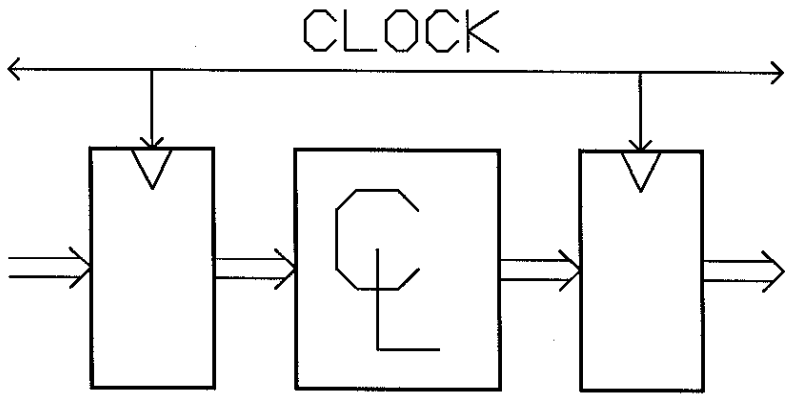


Figure 16: Synchronous pipeline

The combinational logic in each stage has one clock period in which to produce valid output data based on its input data. This time, T_c, is the same for all stages in the pipeline, and the time required for data to flow through the pipeline is $T_n = T_c p$, where p is the number of stages in the pipeline.

A similar arrangement can be used in an asynchronous system. Instead of a global clock, the 4-cycle request and acknowledge signals [Mead & Conway 80] are used to control data flow between stages, as shown in figure 17.

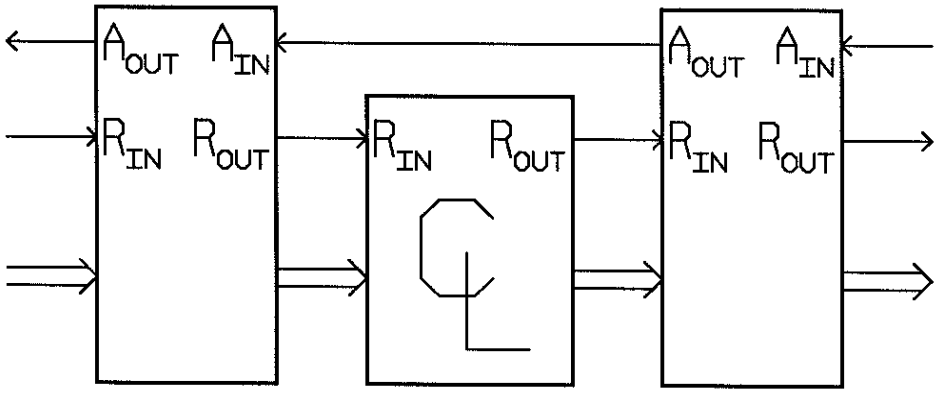


Figure 17: Asynchronous pipeline

In an asynchronous pipeline, each stage processes data at its own rate, and passes its output data to the next stage when it is finished. Each stage thus has its own cycle time, t_c, which is the time it requires to complete its request and acknowledge 4-cycle. Each stage also has a characteristic *fallthrough* time, t_f, which is the time required from when an input request is received until the data is processed and an output request is generated. The ratio of t_c/t_f determines across how many stages a cycle (and an item of data being worked on) extends. By necessity, $t_f < t_c$, and for most designs $t_c \approx 2t_f$.

Looking back at the automata shown in figure 3, it can be seen that data flows in only one direction within an automaton, and the different paths are independent (except for merge operations). It is thus an easy transition to think of routing automata as being implemented using a pipeline structure, as in the MRC.

The transit time, from source to destination, of an unblocked packet in the synchronous case is given by $T_n = T_c(pD+L/W)$ (Chapter 1). For the asynchronous case some of these terms are changed, because the head of a packet advances at the fallthrough rate, which is less than the cycle time. The formula for network latency can thus be expressed as follows:

$$T_n = T_f D + T_c\lceil L/W \rceil$$

where T_f is the fallthrough time for a node. For relatively short packets, D is comparable to L/W, so there is no strong motivation to reduce either T_f or T_c at the expense of the other. T_c and T_f can be expressed as:

$$T_c \approx 2t_p + t_c$$

$$T_f \approx t_p + t_f p$$

where t_p is the time required to drive the pads, p is the number of stages in the internal pipeline, and t_f and t_c are the average fallthrough and cycle times for a single stage of the pipeline, as described above. A pad, and the external components

connected to it, are relatively difficult for a VLSI chip to drive, so $t_p \gg t_c > t_f$. This means we can increase the number of stages in an asynchronous pipeline without significantly increasing the overall delay. In the case of the MRC, increased pipelining has a significant advantage in that having more pipeline stages provides the network with more internal storage for packets, and consequently helps prevent congestion of the network.

3.3 The Asynchronous FIFO

The asynchronous FIFO structure being used is based on chained Muller C-elements [Mead & Conway 80]. Its basic structure is shown below:

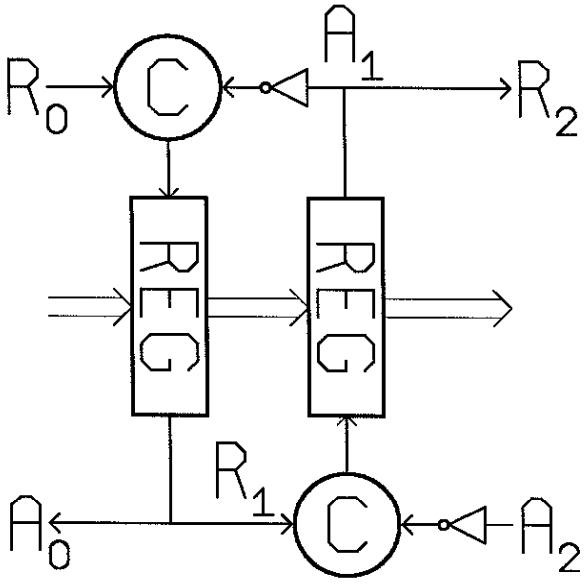


Figure 18: Asynchronous FIFO

Initially, all of the C-elements are reset to 0. Data is presented on the inputs, and the request line (R_0) is pulled high. This causes the output of the first C-element to be pulled high, causing the data to be latched. When this load control line becomes high, the data is assumed to be latched, a request (R_1) is passed to the next stage, and the acknowledge line (A_0) to the previous stage is pulled high. When the next stage latches the data and an acknowledge (A_1) is received from it and the request line (R_0) goes low, the FIFO stage is reset to its initial condition. In this manner, the data quickly falls through the chain of FIFOs, with the data always spread across at least 2 stages. If the request time is significantly less than the acknowledge time, then the flit will be spread across more than 2 stages while it is falling through the pipeline.

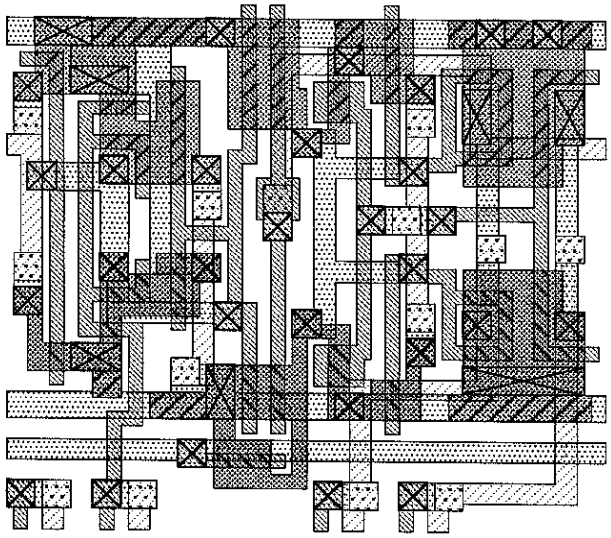


Figure 19: Asynchronous FIFO controller

Care must be taken to insure that the register cells controlled by the C-element are fully turned on or off before request and acknowledge signals are generated and that they are fast enough to latch the data before the load line changes state again. The first requirement can be taken care of by introducing sufficient delay in the request and acknowledge lines, or, more safely, by using a Schmitt trigger (a gate with hysteresis on each input) to detect the state of the load control line. A typical Schmitt trigger inverter of the kind used in these automata is shown below. It operates by using a feedback current to offset the switching threshold of the inverter.

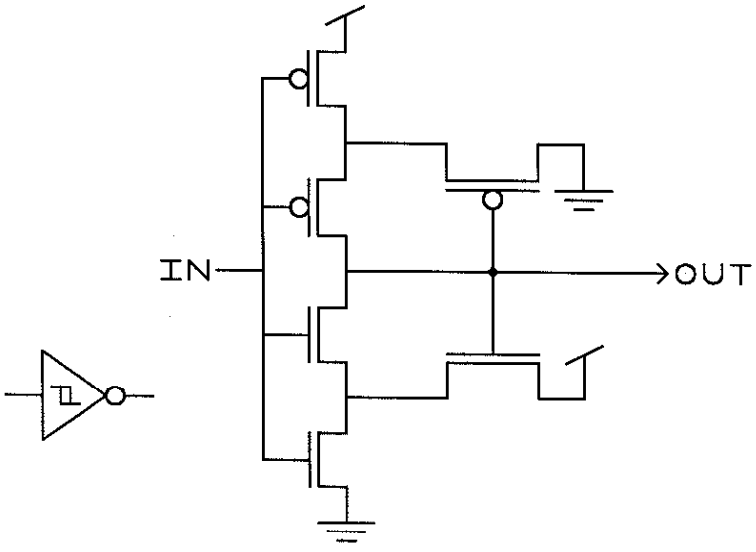


Figure 20: Schmitt trigger inverter

The second requirement of fast latches is usually easy to meet, since latches are generally much faster than C-elements (and Schmitt triggers). The registers used in these FIFOs consist of a closed loop of a strong and weak inverter, with the data gated to the input node of the strong inverter. Thus the data is inverted at each stage of the FIFO, and these stages should be used in multiples of two to preserve the sense of the data. In order to save space, these register cells are flip-composed vertically, for data path widths that are multiples of 2. In the MRC, the path width is 10 bits, so there is an extra bit available for propagating information between stages of the pipeline, if it becomes desirable to do so.

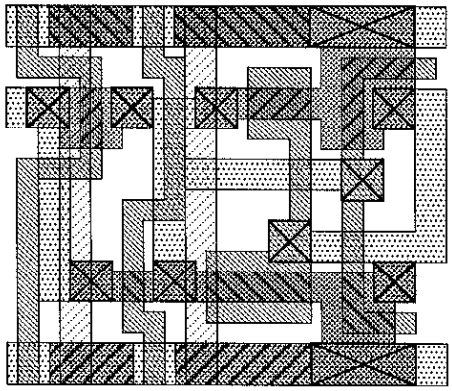


Figure 21: FIFO storage register

For 1.2μm SCMOS, it is expected (from τ-model calculations) that each FIFO should have a t_f (fallthrough time) of about 1ns. This follows the assumption of $t_f < t_p$ (pad driving time), which is about 5ns (more with a large load or long connection line), so that extra stages of pipelining do not add significantly to the latency of a packet passing through a node.

3.4 The Decision and Merge Elements

For simplicity and easy modularity, I decided to use binary decision and merge elements in the asynchronous automata. With careful design, it was possible to use basically the same switch for both elements, simply by flipping it sideways. Each section of the switch consists of a simple 1-to-2 multiplexer (or 2-to-1 demultiplexer), and enough of these are connected along a diagonal to handle the width of a flit. Because of the use of binary switches, the internal construction of the asynchronous automata is as shown below, in which the boxes labeled "D" are decrementers:

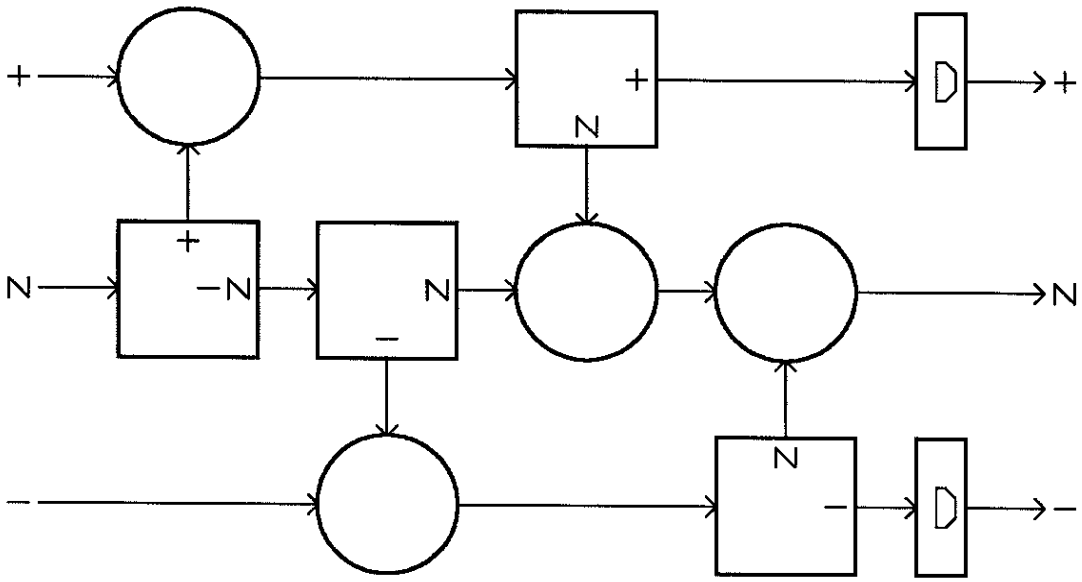


Figure 22: Asynchronous automaton

3.5 The Decrementers

The decrementers are a simple asynchronous ripple borrow type, with a line that is pulled low to indicate completion. Completion is defined by a stage that receives a borrow in, and produces no borrow out because of having a 1 on its data input. This completion signal is used to generate the request signal for the next stage of the pipeline. As with the registers in the FIFO, it is assumed that the forward propagation through the decrementer is less than the cycle time of a C-element and Schmitt trigger combination.

3.6 The Complete Datapath

Two of the biggest problems of working the elements into a datapath was matching the pitches of all the elements to be multiples of each other, and properly spacing the wires in the switches so that they would compose properly under all orientations. Both of these goals were necessary for easy modularity, so that for any automata design a lot of design time would not have to be spent pushing wires and cells around to make all the proper connections. A completed datapath is shown below, and can be compared with the automata shown in Figure 22, on which the topology is based. The size of this complete datapath is $1200\lambda \times 1500\lambda$.

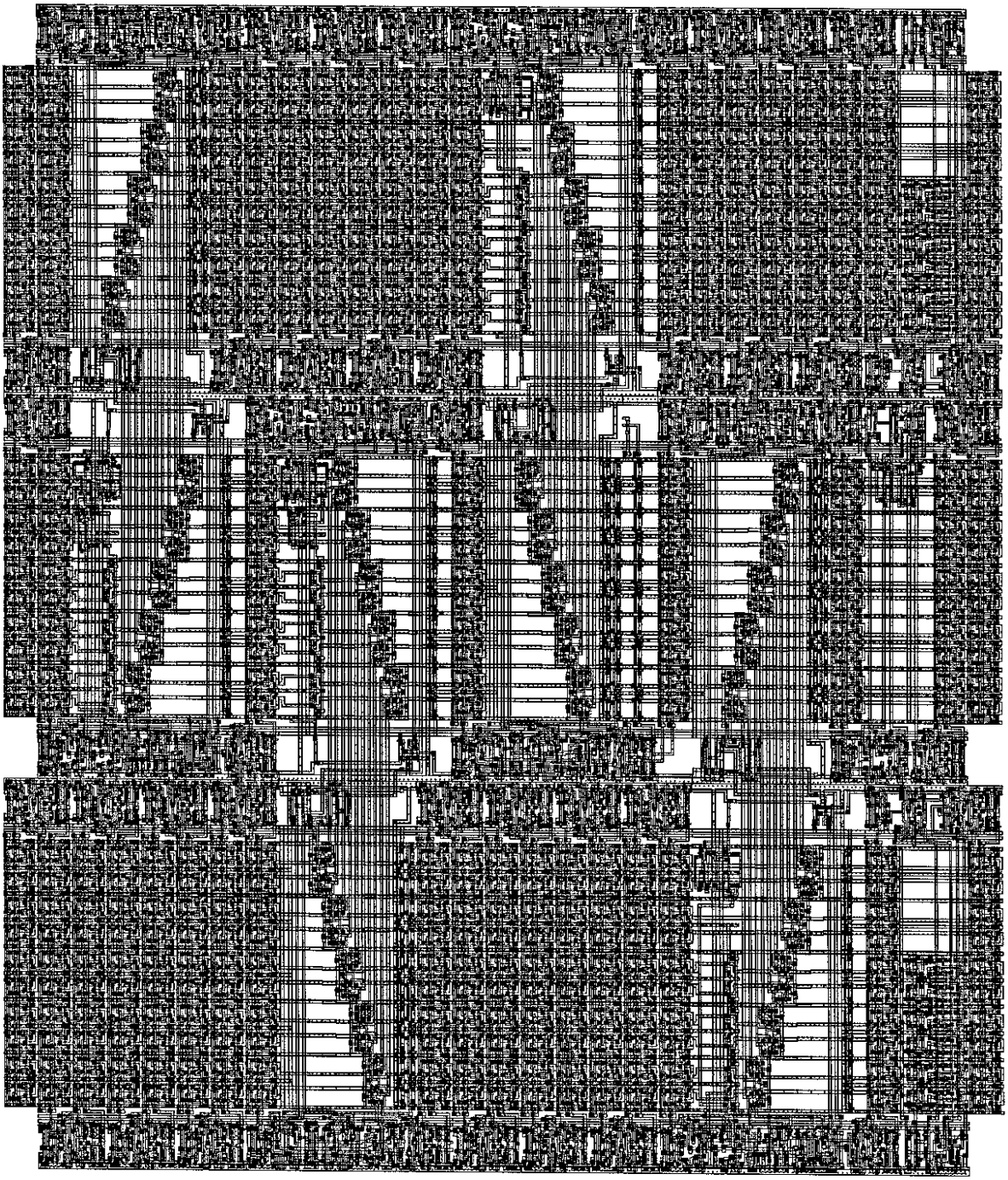


Figure 23: Asynchronous automaton

3.7 The Pad Interface

The internal automata use 4-cycle signaling for flow control, but, to increase speed and conserve power, signals sent off chip must use a 2-cycle convention. A small amount of conversion must therefore be done before driving the pads. This conversion also adds a small amount of delay in the request/acknowledge path, which helps ensure that the data is valid by the time a request is received, even if the delays in the lines are slightly skewed. If the delays are skewed by a large amount, a simple lumped RC delay can be added to the request line external to the chip.

Bibliography

[Dally 86]

Dally, William J., "A VLSI Architecture for Concurrent Data Structures," Caltech Computer Science Technical Report 5209:TR:86.

[Dally & Seitz 86]

Dally, William J. and Charles L. Seitz, "The Torus Routing Chip," *Distributed Computing*, vol 1, no 4, pp 187-196, Springer-Verlag, October 1986.

[Mead & Conway 80]

Mead, Carver A. and Lynn A. Conway, *Introduction to VLSI Systems*, Chapter 7, Addison-Wesley, 1980.

[Seitz 84]

Seitz, Charles L., "Concurrent VLSI Architectures," *IEEE TC*, vol C-33 no 12, pp 1247-1265, December 1984.

[Seitz 85]

Seitz, Charles L., "The Cosmic Cube," *Communications of the ACM*, vol 28, no 1, pp 22-33, January 1985.

Acknowledgements

A lot of thanks go to my advisor, Chuck Seitz, for the instruction he gave me and for the patience he showed for my ideosyncracies. I especially thank him for the time and effort he put into helping me correct my atrocious writing style for this thesis.

Thanks also go to the other members of our research group and the Mosaic project group, for their ideas, helpful suggestions, and support.

Last, but certainly not least, thanks go to my parents for their love and the care they showed in raising me and putting me through school, without which none of this would have been possible.