SUPERMESH

Wen-King Su

Computer Science Department
California Institute of Technology

5125:TR:84

SUPERMESH

by

Wen-King Su

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

May 1984

5125:TR:84
Computer Science
California Institute of Technology
Pasadena, CA 91125

The research described in this paper was sponsored by the
Defense Advanced Research Projects Agency, ARPA Order number 377
and monitored by the Office of Naval Research under contract num
N00014-79-C-0597.

TABLE OF CONTENT

1 INTRODUCTION 1
2 SYSTEM LEVEL 2
2.1 COMMAND DISTRIBUTION 6
2.2 DATA EXCHANGE 8
2.3 THE CLOCK 11
2.4 THE ALTERNATIVE 17
3 NODE LEVEL 19
3.1 THE REGISTER STORE 22
3.2 THE DATA CHANNEL 25
3.3 THE FLOATING POINT ARITHMETIC UNIT 28
3.3.1 NUMBER SYSTEM AND CONVENTIONS 29
3.3.2 MULTIPLY ALGORITHM 31
3.3.3 ADDITION ALGORITHM 35
3.3.4 DIVISION ALGORITHM 36
3.4 COMMANDS 39
4 CONCLUSION 41

PREFACE AND ACKNOWLEDGMENT

To relieve the pressure on the readers and author (mostly the author), and to try to capture a record of the design process, this paper is written using a deliberately "unsophisticated" mode of presentation. A paper for publication, and with more technical taste, will be written when appropriate.

To an unusual extent, this project is a joint effort with my advisor C.L. Seitz, to whom I express my deepest thanks.

During the first part of this project we held many meetings. In these enormously productive meetings we tried to forge into a single entity the many ideas we had, triggering even more thoughts and ideas. By the time a final form emerged, each of us found it impossible to distinguish the source of the ideas presented.

During the second part of this project, he would again patiently read over and correct each draft of this paper, making its writing a valuable experience for me.

1. INTRODUCTION

Many applications one encounters in computing are difficult not because of their complexity but because of their size. Applications such as computer simulations and computer graphics typically require massive amounts of rather simple calculations. Pressure for larger and faster machines increases; yet, present day computers are already hard pressed to their limits. We must find new machine architectures or face a dead end.

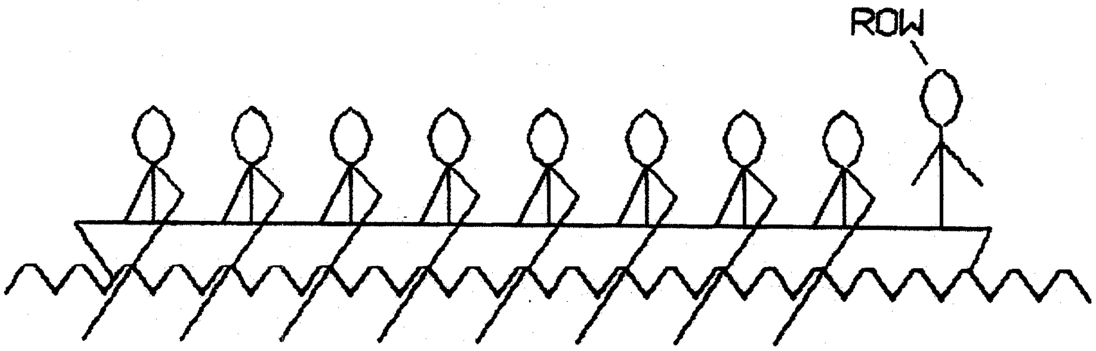
Of those applications that require computing power beyond our current ability, a substantial fraction of them can be formulated as a large collection of smaller tasks that run concurrently. The applications that divide into numerous small pieces are most economically run on an an ensemble of many small computers, but for those that divide into only a handful of large chunks, we can run them in a network of a few large machines. No particular machine of this type will prevail over the others because the economy of using any one of them is a delicate balance between what is needed and what is provided; a balance between flexibility and performance. Fine grain machines generally have higher performance because they contain many more processors than comparably sized coarse grain counterparts; however, they are less flexible for they have less memory to store programs, and in some extreme cases are totally dependent on hard wired logic for their sequence of operations.

In light of the mounting demand for machines of ever increasing size and power, we hope to learn how one constructs large machines of this type economically. In this experiment we present the design of an arbitrarily extensible machine called Supermesh, an SIMD machine whose cost / performance is pushed to the limit and whose architecture scales perfectly with advances in VLSI technology.

Although the SIMD machine is but one of the many types of machines that will ultimately face the problem of size, its design exposes fundamental problems of synchronization and broadcast communication, and its study provides insights to how other machines may solve their problems.

2. SYSTEM LEVEL

An SIMD machine is a collection of processors called nodes operating on data stored locally in each node according to commands emitted by a control computer. These nodes are interconnected in a regular network [Siegal 79] to communicate with each other. Thus these individual nodes are nothing more than simple calculators fitted with communication ports and some storage registers, and operated in parallel by the control computer.



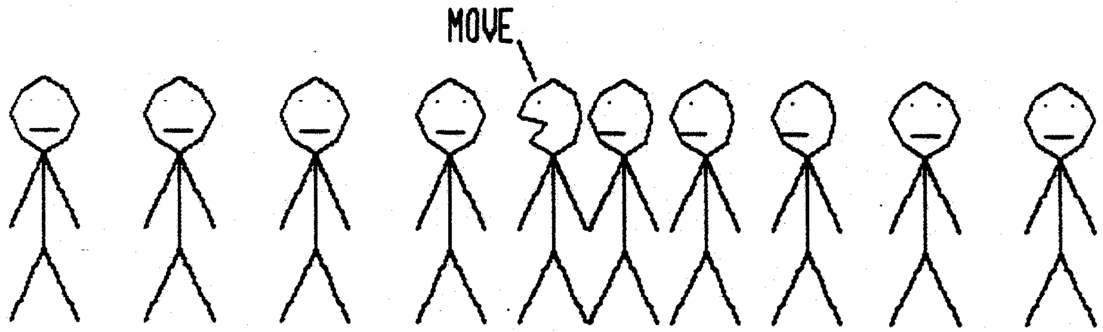
A simple example of SIMD machine

This type of machine belongs to the finer grain end of a taxonomy [Seitz 82] of machines ranging from the single processor general purpose computers to simple memories.

General purpose computer
Loose MIMD network
Tight MIMD network
SIMD network
Computational array
Logic-enhanced memories
Simple memories

Unlike the computational or systolic arrays as described in [Kung & Leigerson 80], the sequence of its operation is not built in into the hardware; yet, unlike the MIMD machines, the individual nodes are not full blown computers. By throwing away the program related functions found in MIMD machines, SIMD machines gain a speed and size advantage at the expense of flexibility. However, many applications fit nicely in an SIMD environment, and some may require power achievable only with such a machine.

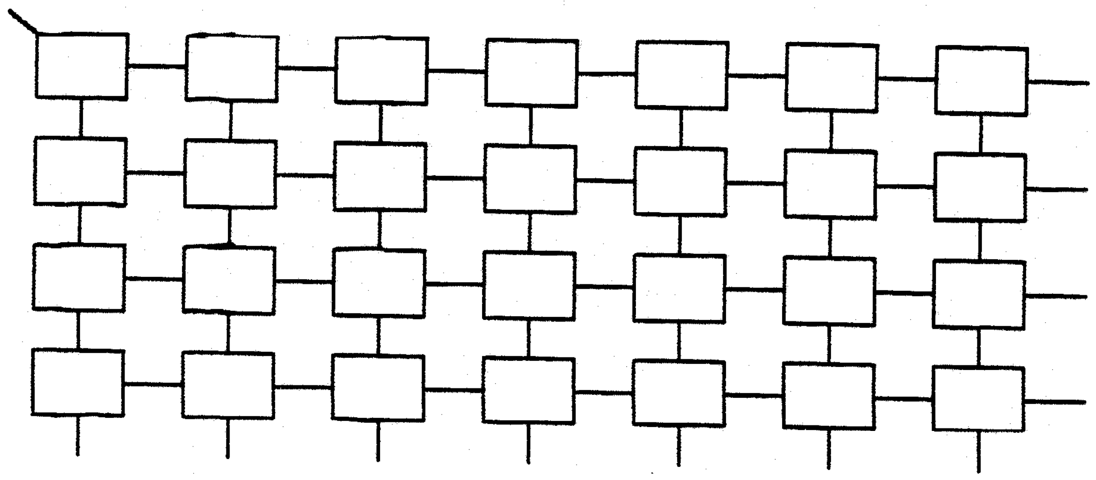
To get an overall picture of this machine, let us look first at the tie that holds it together: how the control computer send commands through the machine. In a simpler system with only a few nodes, the practical way is to directly connect all nodes to the central control computer. But, if we have thousands or even millions of nodes, the machine will look like porcupine many times over. The maximum amount of power that can be concentrated at the control computer to drive all these wires, and the maximum number of nodes we can practically fit around the control computer, will ultimately limit the machine size, while we insist that our design be arbitrarily extensible.



Sending command across an arbitrarily long array of people

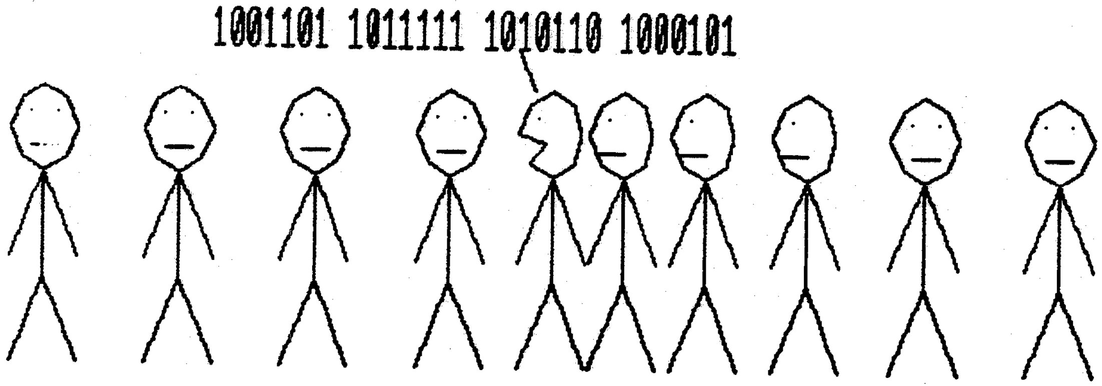
The only solution is first to have the control computer give the commands to a selected one or a few nodes, then have the commands amplified and pass from node to node until every node in the system has received the commands. Thus the machine should look like a large network of nodes with a single entry point where the command enters and disperses.

In the old days when a node was something as large as a file cabinet, and a machine is just an arbitrary pile of these cabinets cabled together, we really did not have many restrictions on how they are interconnected; the machine will not get too cluttered up if we apply just a few more spools of coaxial cable. For this VLSI machine, our options are limited by the shape of the chips on which many nodes would be built (flat).



Nodes connected in a two dimensional mesh
In all figures, command entry is at the upper left corner.

A regular two dimensional mesh is a good choice because most problems requiring this machine map naturally onto two dimensional manifolds. In such an network, each node can communicate directly only to its immediate neighbors. Thus one might call this architecture a "processing surface," a name coined by A J Martin [Martin 81] to describe an MIMD machine that could be made arbitrarily physically extensible by the same approach as is described here.



Communication between neighbors should be serial

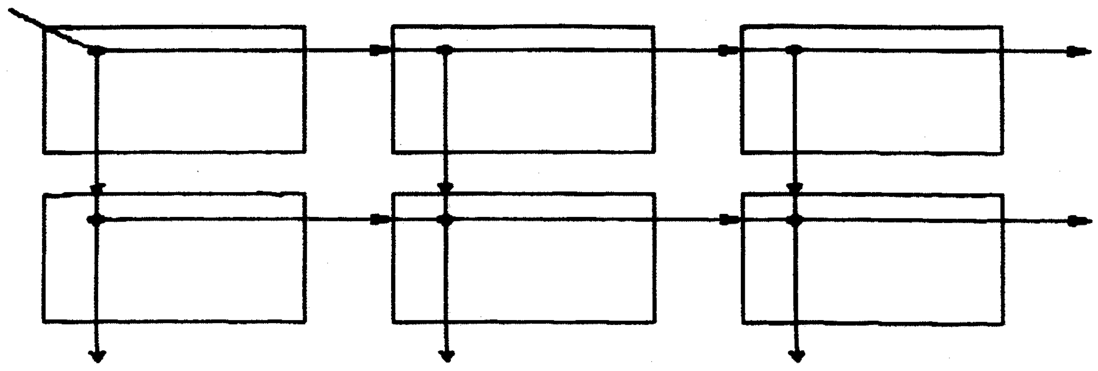
Furthermore, since the distance between neighbors is short, the links connecting them will have a very high bandwidth. The high bandwidth of the links allows the use of serial communication, saving extra carrier pins, chip area, and power used in driving the links.

From these logical deductions, an SIMD machine of small node size can exploit future technologies if it is configured as a flat sheet of mesh-connected nodes driven from one corner by a control computer, and have each node connected with serial links only to their neighbors. The overall picture for this machine is now clear; however, we also need to describe what the nodes do, and how they work. The node design will be described in chapter 3.

In this chapter the network is analyzed and defined in detail. The distribution of commands, the generation of clocks, and the exchange of data are the three major areas. Since they are tightly intertwined it is very hard to present this subject in a linear fashion; indeed the parts of this whole machine fit each other like the core of a Rubik's cube, making it difficult to describe. Here, the work on the network level is presented in a more or less chronological sequence. First, recognizing the difficulties in distribution clock signal through a very large machine, and knowing a number of possible solutions, we decided to postpone its definition and instead to place a weak assumption on the clock as the starting point. We assumed that the clock signal can be generated and distributed throughout the system in a way that bounds the skew between neighboring nodes but not necessarily across the array, the minimal condition assuring reliable synchronous data exchange in SIMD machines. We then devised a system of pipelined command distribution and data exchange based on this assumption alone. However, later when we designed the mechanism for producing the clock, we realized that it has some stronger properties that permits an alternative to the original plan. This discussion may seem to take a circle, but only in this way can we clearly and logically describe the evolution of these two equally workable plans.

2.1. COMMAND DISTRIBUTION

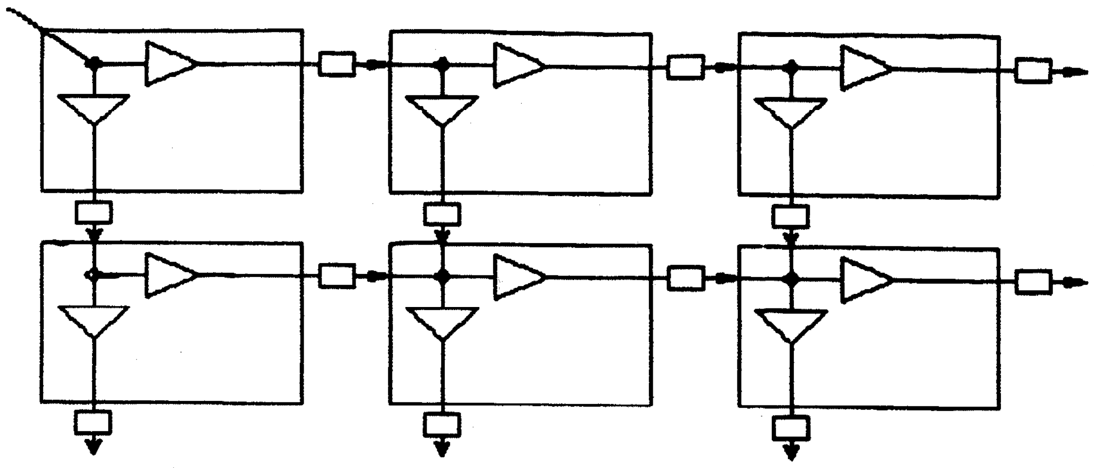
In a small SIMD machine, the control computer brings about concerted action in the nodes by asserting its command at every part of the network at the same time.



Example of command distribution. The command enters from top left corner and is distributed through the grid. Actually half of the path shown here are redundant. The paths are shown running in both directions for symmetry and clarity.

For an arbitrarily extensible machine, sending command from the control computer directly to every node of the network is impossible. As described earlier, in a large machine, the command must be amplified and passed from node to node, and all these actions take time. The control computer must allow enough time for the command to propagate all the way to the very last node; thus an SIMD machine that depends on concerted actions of its nodes must have a clock period proportional to its physical dimension. Such an approach violates our design principle that the machine be arbitrarily extensible.

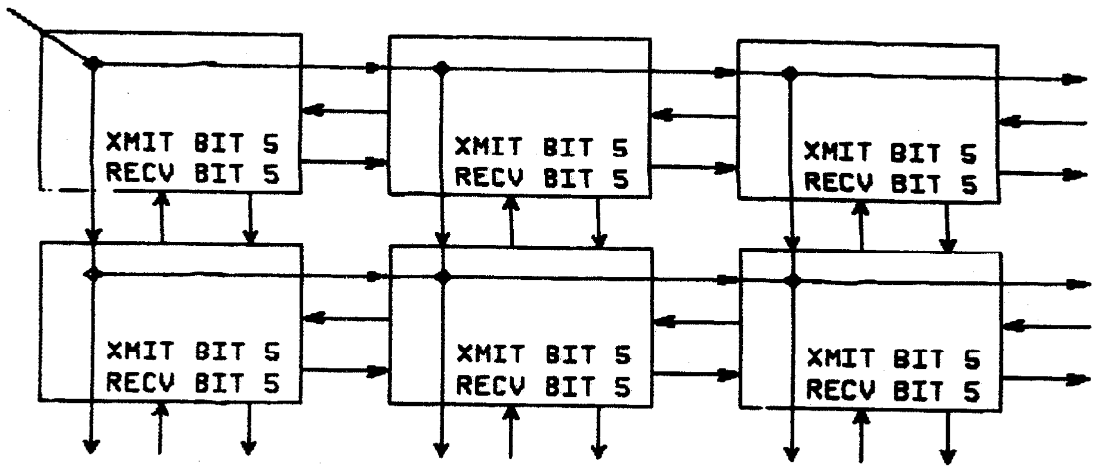
Pipelining is a solution to this kind of problem [Cohen 78]. In a pipelined system, the command from the control computer advances only one amplification stage for each clock cycle. Each amplification stage contains a set of one or more nodes residing in the same equipotential [Seitz 80] region.


Each large box represent an equipotential region.

The commands sent out by the control computer will travel in wavefronts through out the network, where only those stages along the same front are in step with each other. Since commands from control computer initiate and synchronize node functions, the state of nodes will skew with the commands; nodes in one amplification stage in the network lag the nodes in the stage behind them by one cycle, and lead the ones in front of them by one cycle. Concerted action is no more; however, the control computer no longer needs to wait for the command to be fully propagated. Thus, if a machine has N amplification stages, it can potentially run N times faster with pipelining. However, pipelining is not all good. If we need, at any moment, for the control computer to be able to observe valid results in every node, the control computer must issue no-ops to allow all previous commands to complete. However, this will not occur here because we look into the network only through its corner; in order to see the entire network, the control computer must bring data from each node to this entry point. As long as the node at the corner stays synchronized with the control computer, the control computer will not see the skew.

2.2. DATA EXCHANGE

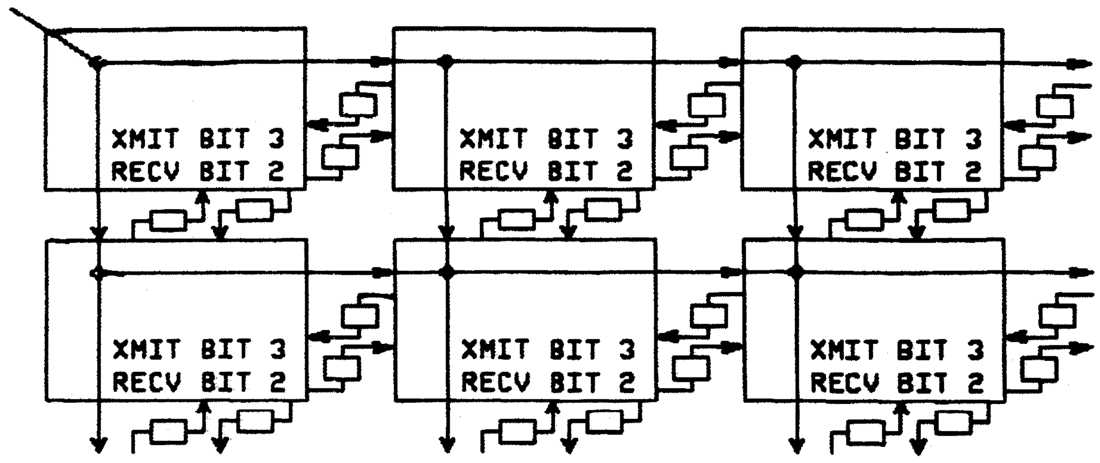
Now we consider how information is exchanged among the nodes. In SIMD machines the command to send data is synonymous with the command to receive data. When the control computer says "give data X to your east neighbor," it is also saying "take data X from your west neighbor."



Communication is simple in an unpipelined SIMD machine.

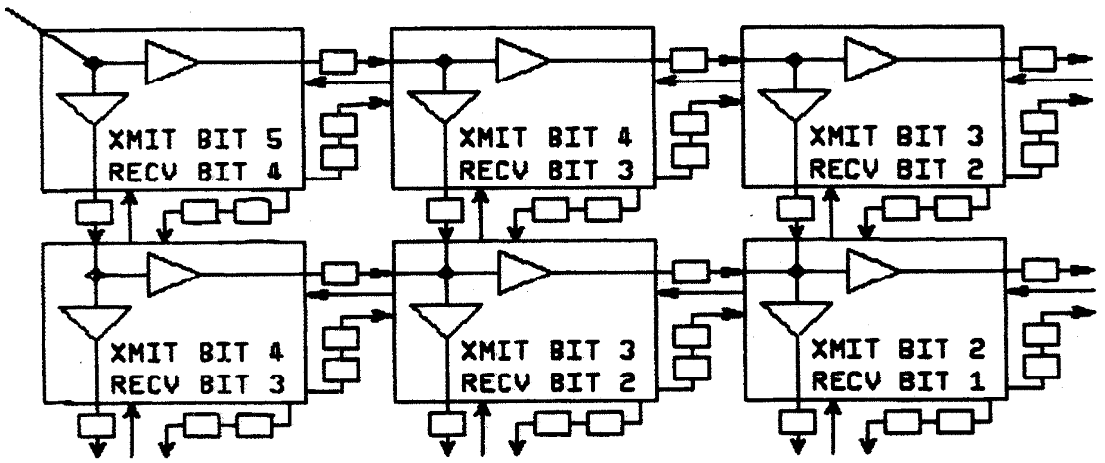
Doing transfer in a machine without pipelining is easy; in this example each node simply uses an east-west running shift register as a port; write X into the register; then shift its contents out in a west to east direction; and at the same time expect data from its west neighbor to shift in simultaneously from the other end. If a word has n bits, then each node does n shifts for each communication cycle.

However, things get more complicated with pipelining. Since commands must ripple through the array, not all nodes can be in the same step as their neighbors. For those that must communicate across the boundary, the data bits from nodes upstream would arrive one clock too early and the data bits from nodes downstream would arrive one clock too late. We can easily compensate for this skew by first adding one clocked delay element to all data links in the un-pipelined array.



An unit of delay is inserted between all links.

Although only links on the boundary of equipotential regions are shown, links between nodes inside the same region are similarly delayed. Now all data bits arrive one clock late in all directions. In this array each communication cycle takes n+1 shifts to complete.



SIMD machine with compensated data links.

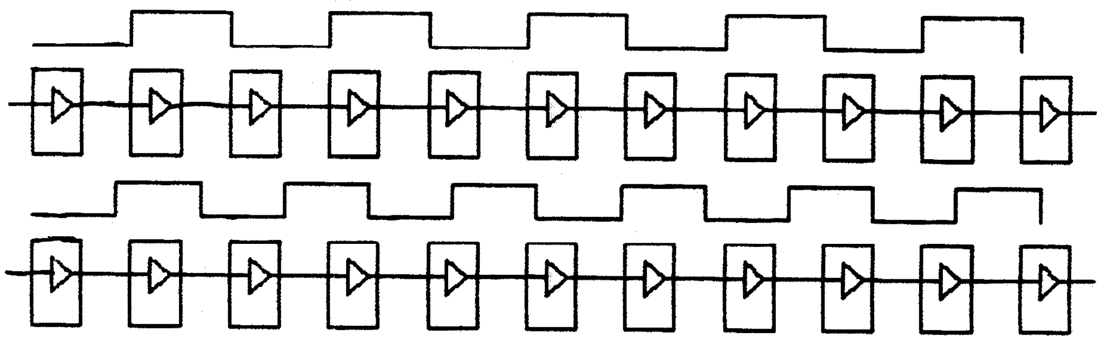
Now when we add the pipelining on the command, we introduce one additional clock delay across the equipotential boundaries. To counter that extra delay, we simply remove the delay from the backward links that crosses the boundaries and add it to the forward link. With this skewing in data, individual node sees the mesh, not as one with a built in command skew, but as one that is same in all directions and exists in exactly the same clock cycle as the node itself.

2.3. THE CLOCK

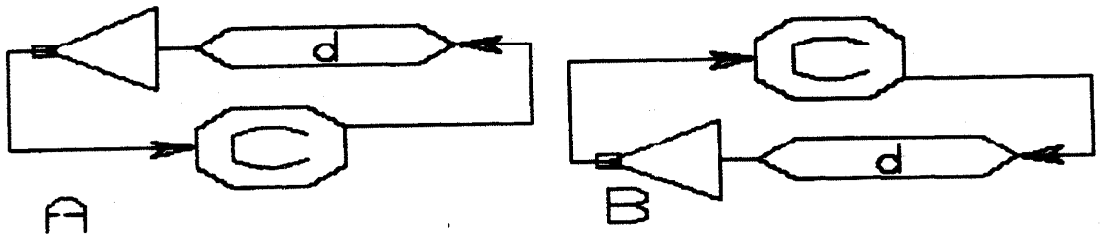
The data exchange and command distribution mechanism as described in the previous sections represents one of the simplest of all communication techniques available. However, it places some restrictions on the behavior of the clock. The machine must guarantee that every pair of neighbors remain in step at all time. One small drift will cause nodes to take in invalid data or command bits. Machines of larger node sizes keep everything in synchronization by making one node wait for another; waiting, however, is not an easy task for nodes in SIMD machines. These nodes do not contain any programs, therefore, they cannot enter any waiting loops of their own. Making sure the clock signals of neighboring nodes always stay in phase is the only alternative.

This objective is not too difficult unless we need a machine arbitrarily extensible in all dimensions. In a conventional computer, the system clock is just a signal; a bit stronger perhaps, but nothing more then a signal to be distributed evenly throughout the machine. In a larger machine, when driven from a single point, any signal will eventually die out at the far end of the machine, like water pressure at the periphery of a metropolitan water district. We have no practical way of massing enough power at the driving point to propel an arbitrarily large machine. Buffering the signal along the way is necessary but it introduces skew. The skew accumulated along two different routes can differ widely, and as a result, nodes controlled by the same signal will still go out of step [Fisher & Kung 83].

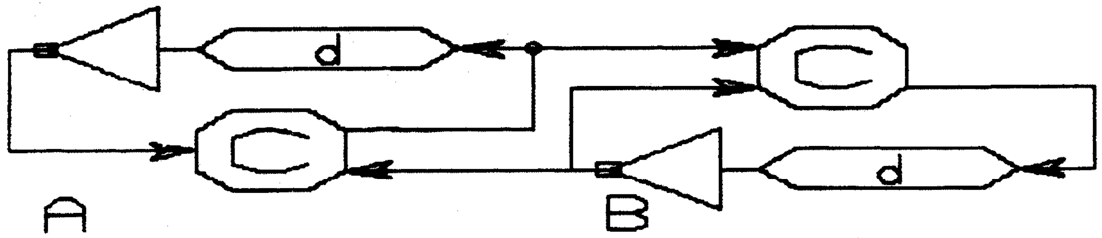
Our most satisfactory solution is one in which the clock generator is a gigantic array



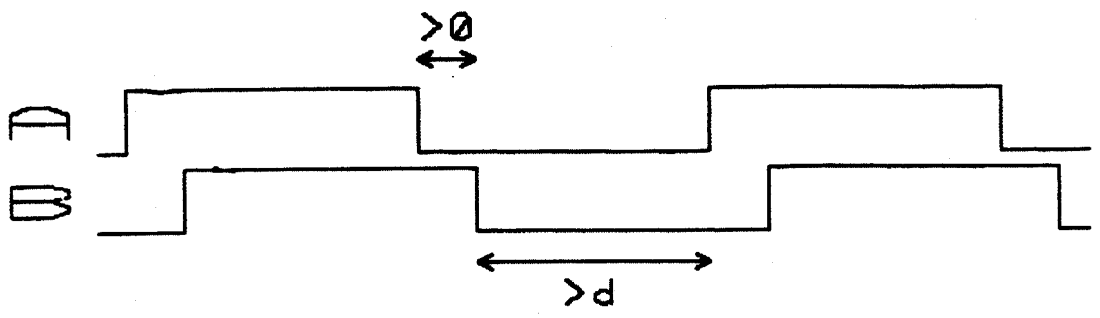
of coupled ring oscillators. In the remainder of this section I shall describe one derivation of this clock system. It is not the original derivation; however, it is easier for me to explain and understand. First, let us consider ring oscillators. A ring oscillator is an amplifying negative feedback circuit usually drawn as a loop containing an ideal inverter and the lumped delay d. Suitably initialized, this circuit will oscillate with period of 2d. Shown here are two ring oscillators with a C-element spliced into each loop. C-elements are simple asynchronous sequential machines whose outputs goes to zero when all inputs are zero, goes to one when all inputs are one, and stay unchanged under all other input conditions. In this first example, the C-elements have no effect on the oscillators. Later, these C-elements will serve synchronization functions.



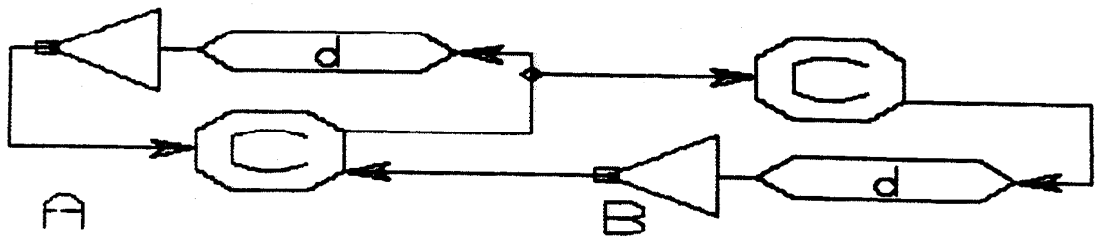
Let us treat the output of the C-elements as the output of the oscillators. Our objective is to lock the phase of these two oscillators. We can do this by enforcing an ordering of transitions between the two clocks. By feeding the clock output of A into the C-element of B, we force all clock transitions of B to follow that of A.



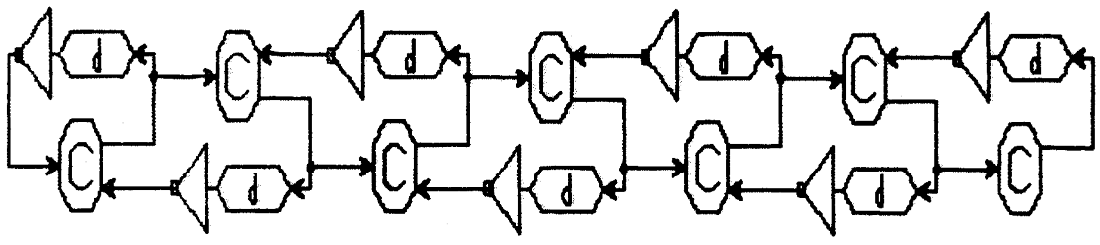
By feeding the inverter output of B into the C-element of A, we prevent A from making a transition until B has caught up with its previous transition. At the same time we ensured that both A and B will stay in the same state for time d. The sequence of operation now is: A rise, B rise, d delay, A fall, B fall, d delay, A rise ...



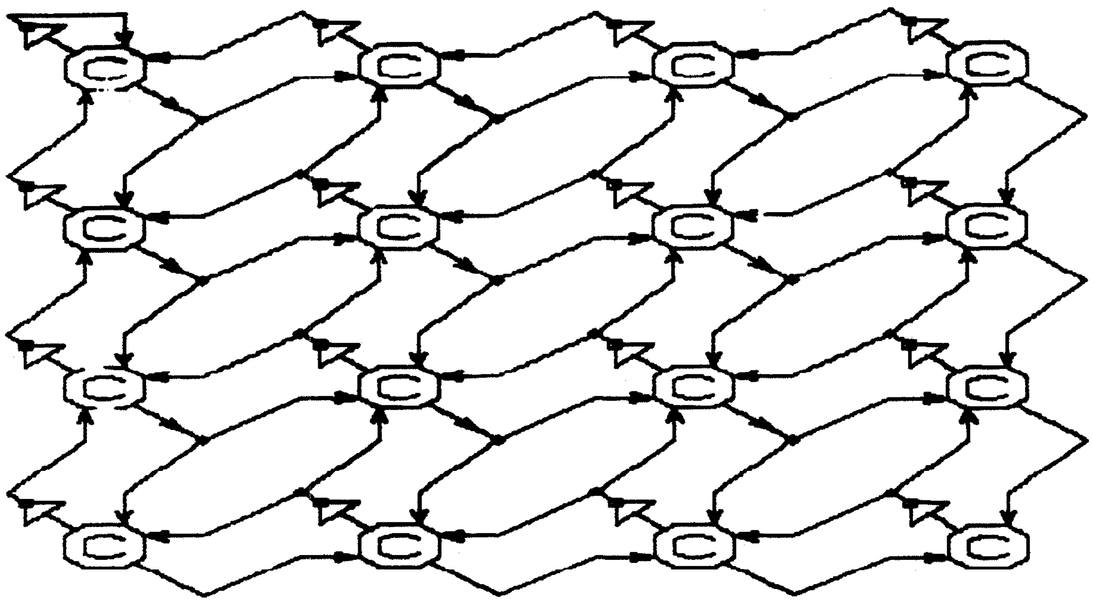
Finally, by removing a redundant link, we arrive at the final form of the locked oscillators.



This pair of oscillators can easily extend to higher dimensions. To produce one dimensional array, we append extra stages to the right.



To produce two dimensional array, we extend in two directions. For simplicity, as drawn, the delay elements are incorporated into the inverters.



This two dimensional mesh of oscillators can extend indefinitely, and will oscillate with frequency determined by the longest delay element. Each oscillator will stay in step with its neighbors, and their clocks will have a guaranteed minimum overlap determined by their own d. Furthermore, since the oscillator in front leads the oscillators behind it, the skew is consistently positive in the down and right direction. The clock will appear like waves propagating from the upper left corner of the mesh down toward the lower right. It will look like as if the clock is actually generated at one corner and then distributed over the whole array like a series of expanding ripples. Unlike the typical driven array, however, the diagonally running crests and troughs of the waves continue from one end to the other without breaks.

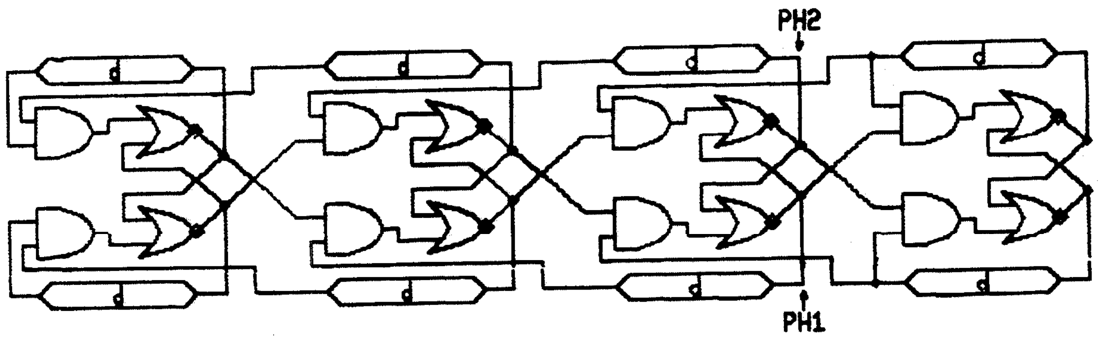
The following are the outputs of computer simulations of the two different methods of distributing clocks. Once again the corner is on the upper left. It is also the point where ripples of clock originate. On the left is a simulation generated with the oscillator method we have discussed, and on the right with the conventional amplifier method. Each character position represents a clock element in the clock array. Where there is an '@', the clock is high; where there is a space, the clock is low. A '*' means the clock has just made a transition from low to high, and a '+' means transition in the other direction. The delays of the amplifiers were picked with a random number generator, and the same distribution of delay is used for both simulations. Due to the limited space available, the spread of the delay values is greatly exaggerated to show

what can happen in a very large array.

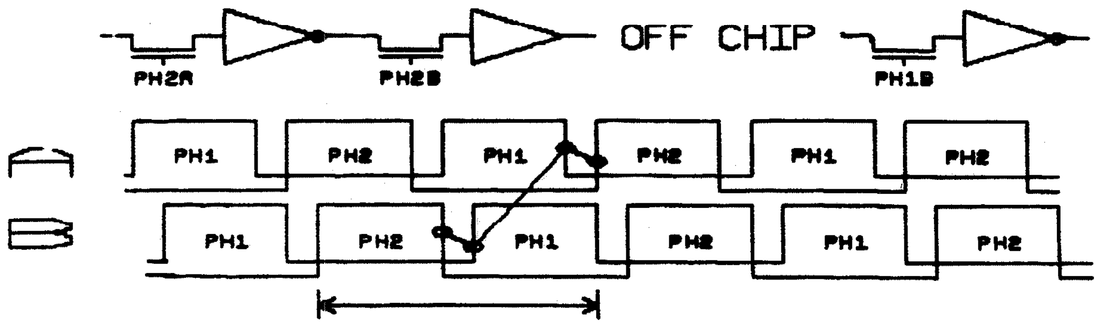
```
:--------------------:      :--------------------:
:++@@@      ++@@**   :      :* +@***  @@***++@@@@:
:@@@*    +++@@@**    :      : ++ *@+ + * **+@@@@*:
:@***   +++@@@@**    :      :+ @ *@++@ *++ @@@@**:
:***   +++@@@**     +:      :+ @++@@@@+*+ @@@****:
:**    +@@@***     ++:      :@  ++@@@@@++ @@@*** :
:*   ++@@@**      ++@:      :@  +@@@@@ +@@@@**   :
:  ++@@@@**     +++@@:      :@+++@**** @@@*@*    :
: ++@@@@@*     ++@@@@:      :@++*@**** @@@ *     :
: +@@@@@**    ++@@@@@:      :@+@*****  *@* *     :
: +@@@@**    ++@@@@@@:      :@@@*** *  *** *     :
:++@@@**    ++@@@@@**:      :*@@ *    +***+*   + :
:@@@@**    +++@@@@***:      :*@@      + * +   ++ :
:@@@**    ++@@@@***  :      :*@*      @   @   ++ :
:@@@*   ++@@@@@****  :      : **      @+  *  ++@+:
:@@**   +@@@@@***    :      : **+ +   @+  *  +@@+:
:@**    +@@@***     +:      : **+ +  +@@ +* ++@@+:
:**    +@@@@**     ++:      :+  @ ++ +*@ + ++@@@@:
:**   ++@@@**     ++@:      :+  @ @+++*@ + +@@@*@:
:*   ++@@@**     ++@@:      :+  @+@++@*@ @ +@@**@:
:   ++@@@@*     ++@@@:      :@  @+@@+@ *+* @@@**@:
:--------------------:      :--------------------:
 locked  oscillator  array   regular  amplifier  array
```

As we can see from the simulation, in the first case, the neighbors of any element are either in the same clock cycle, or about to enter the same clock cycle, or have just left the same clock cycle as the element itself. In the second case, however, the waves are completely broken up.

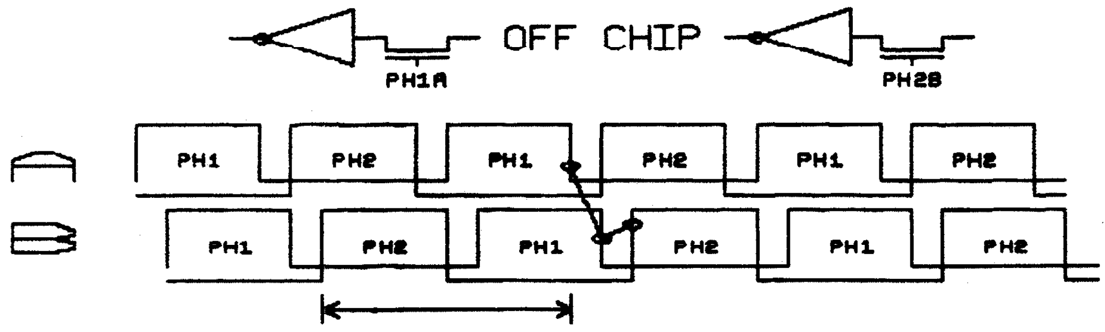
This nice wave-like behavior of the clock network, due to a consistent direction of skew, enables a different form of command distribution, as we shall see later. But first, let us see how this clock might be used in data transfer. In a typical MOS two-phase synchronous system, we only need to assure that data latched out on phase 2 clock on one node will be reliably latched in on the following phase 1 of its neighbor. But first of all, we need a two phase clock.



This figure depicts a gate implementation of 4 clock stages in a linear array. The cross coupled nor-gates and the two and-gates in each stage represent the C-element. The two and-gates each receive a complement of two signals, one is the output from the stage before and the other from the stage after. Since cross coupled nor-gates generate non-overlapping complementary outputs, its outputs can be tapped directly for the two clock phases needed. Further, since complementary outputs are available at the nor-gates, inverters are not needed; however the delay elements are duplicated, one to delay each complement of the output.



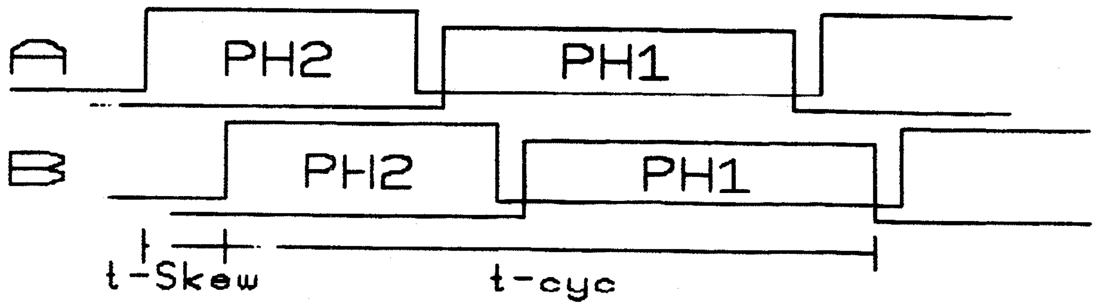
Now, with the clocks generated as above, when sending data in same direction as the clock waves, resynchronize the data with phase 2 of the receiving node. The phase 2 clock will not pick up wrong data by accident because of the guaranteed sequence of transitions indicated by circles joined with line segments.



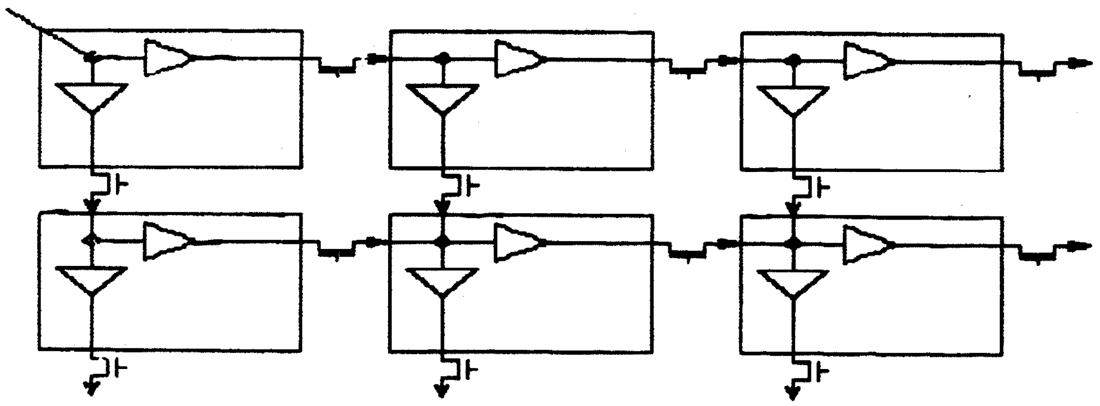
Data transfer in the other direction requires no resynchronization at all. Valid data transfer is again guaranteed by a fixed sequence of transitions. In either cases, data has 2d time to jump from node to node.

2.4. THE ALTERNATIVE

As we have seen, in the previous pipelined command distribution, one full clock cycle is allowed for carrying the command from one equipotential region to another, carried in a wavy fashion across the network by clock which themselves are wavy. We allowed one full clock cycle in data transfer.



However the command, like the clock, is an external signal, it should only have a skew comparable to that of t-skew. Nothing more can be said of the relationship if the clock circuit is built with a different circuit technology. Otherwise, the skew of the command may be much less then that of the clock, allowing it to hop from stage to stage not in one t-cyc, but in one t-skew. If this is the case, all we need is a simple pass gate to move the commands along the wavefronts of the clock, like ocean waves carry surfers gliding ashore in one swift stroke. The pipelining is not needed, and the elaborate mechanism used to cancel the effect of pipelining is not needed, either.

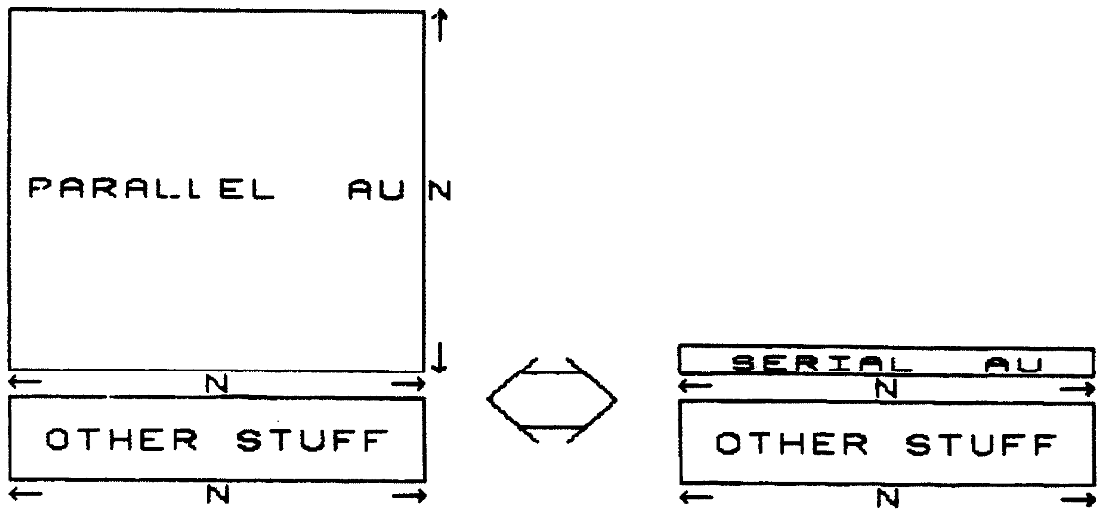


Surf board instruction distribution

However, depending on the choice of technology, the latter method may have a lower performance because the device on which we build logic may not be the ideal device for a clock driver. Yet, where our technology leads, we cannot tell, but whereever it leads, these are two equally workable plans.

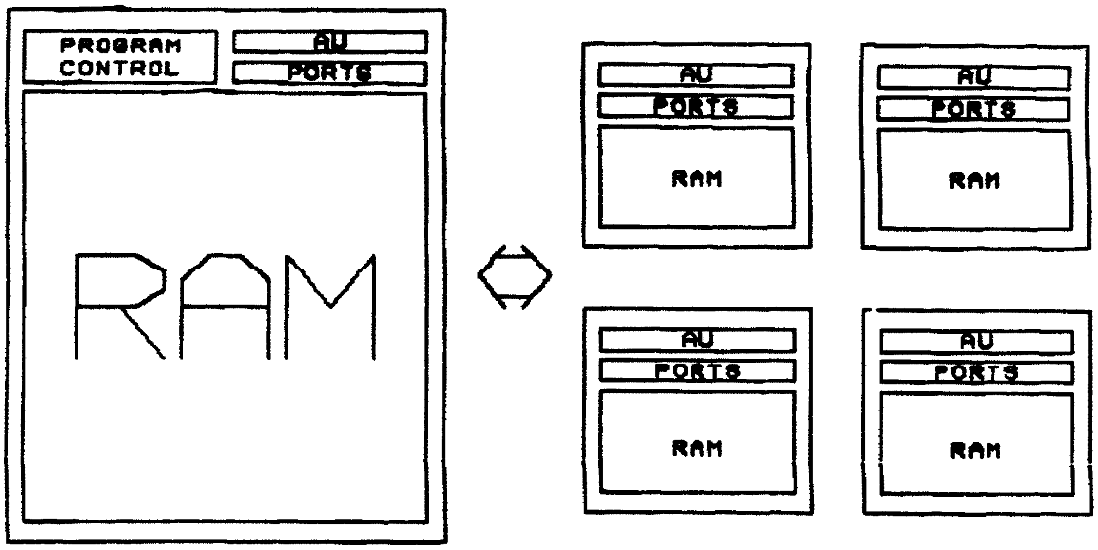
3. NODE LEVEL

As described in the previous section, the nodes in this SIMD machine communicate serially due to the small size of the nodes, the closeness of neighbors, and the promise of future technology to package many more nodes onto a single chip. Serial communication also helps make the nodes small by allowing the use of serial arithmetic; if commands from the control computer arrive slowly, there is no sense making the nodes lightning fast. Serial communication and serial arithmetic complement each other; one either has to make the nodes large and fast and parallel and suffer all the ills of Illiac IV, or make the nodes small and serial. However, as it turns out, the arithmetic cannot be entirely serial, just enough to match the speed of communication.



Use of serial arithmetic reduces node size

By using serial arithmetic, we achieve an order of magnitude complexity reduction in area of the AU (Arithmetic Unit). This reduction is secondary, however. The primary reduction in node size is accomplished by factoring controls out of the nodes. Conventional processors are largely occupied with program related circuits. A typical computer instruction may require the processor to fetch the instruction stored at location pointed by a program counter, update the program counter, compute operand addresses, read the operands, operate on the operands, and then store the result back to memory. All this involves a lot of circuitry and registers, yet if this type of processor is used for building SIMD machines, every step except the AU operations will be exactly duplicated in every node. This duplication in circuit and the associated duplication in program storage is a great waste of chip area and power in applications that can be formulated instead for SIMD machines.



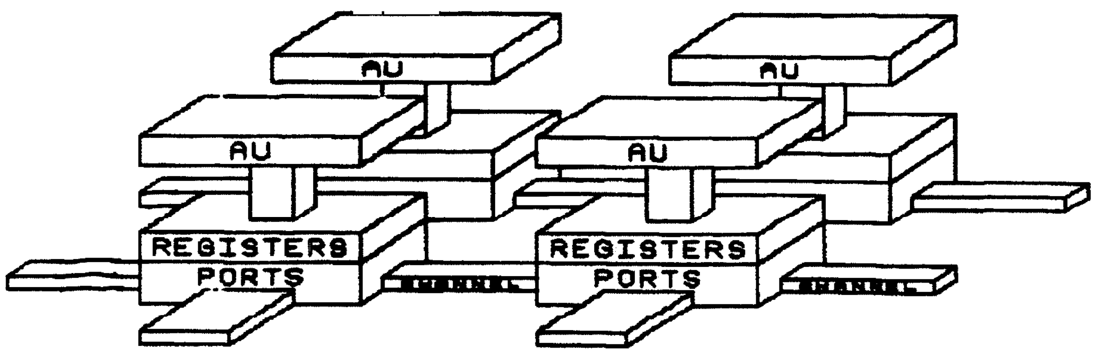
We can build more processors in the same chip area by removing program and program related functions.

A more reasonable thing to do, as has been done in all other SIMD machines, is to extract as much of the duplicated control out of the nodes as possible and put them into the control computer. Every complicated operation is digested once and only in the control computer. Further, since there is nothing in the nodes having to do with programs, there is no need for storing them either; therefore the amount of RAM in each node can be trimmed to that minimum required for data storage.

After much cutting and trimming, the node element is cleanly divided into three independent units:

1. The Register Store
2. The Communication Ports
3. The Arithmetic Unit

Each of these three units has their elementary command streams coming from the control computer. These command streams are the only way by which these three units are tied together - even the synchronization mechanism has been taken out of the nodes.



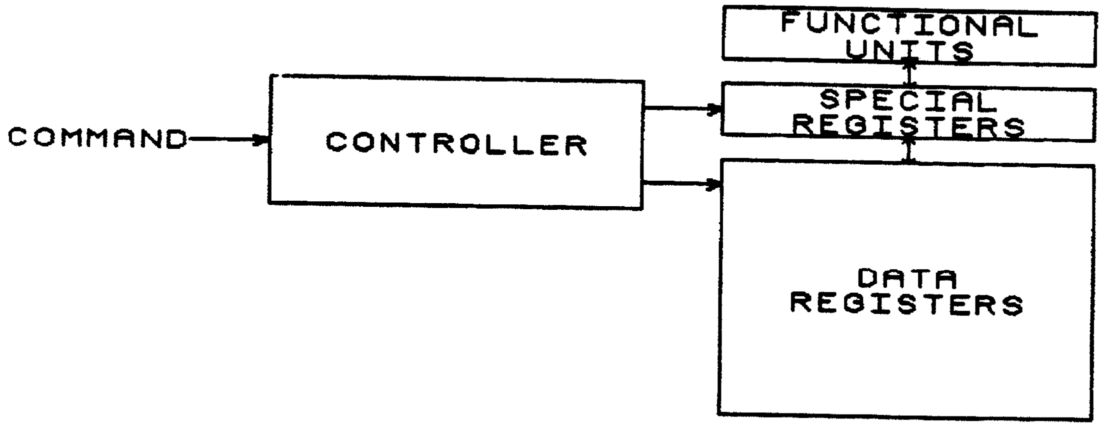
The interfacing of these three units are simple; the AU simply writes and reads results and operands to and from fixed registers inside the register store, and the communication ports simply appear as a set of fixed registers also in the register store. It is the responsibility of the register store to see that the right data appear in the right register when needed by the other two units, and it does so under the command of the control computer. The simplicity can best be realized by looking at the commands sent to each unit: the AU needs only the commands for add, subtract, and multiply; the register store needs only two addresses for moving data words around in the register bank; the communication ports need no command, it is configured as a continuously flowing stream. Controls for these three units are very simple, especially for the communication ports and the register store. The register store is the center of the node, yet it is the simplest and thus should be described first.

3.1. THE REGISTER STORE

The register store unit contains two sets of registers and a very simple controller. The larger set of registers contains the data registers, which is primarily composed of an array of dynamic memory cells used for data storage. The smaller set of the registers contains the special registers used to tie the rest of the node together. Among the special registers, 3 of them are associated with the AU and 4 of them with the communication ports. The 3 special registers associated with the AU are dual port registers connecting the register store and the AU; while the 4 special registers for the communication ports are the ports themselves. More special registers may be required if more parts are added to the node; some applications such as computer graphics may employ units that perform other functions.

The operation of the register store is simple; in serving the AU, the controller transfers the operands from designated data registers to AU's operand registers before every AU cycle, and the controller transfers the result from result register back to some designated data register after every AU cycles. Other function units are similarly treated. Thus, the register store needs only two addresses and a direction flag to operate.

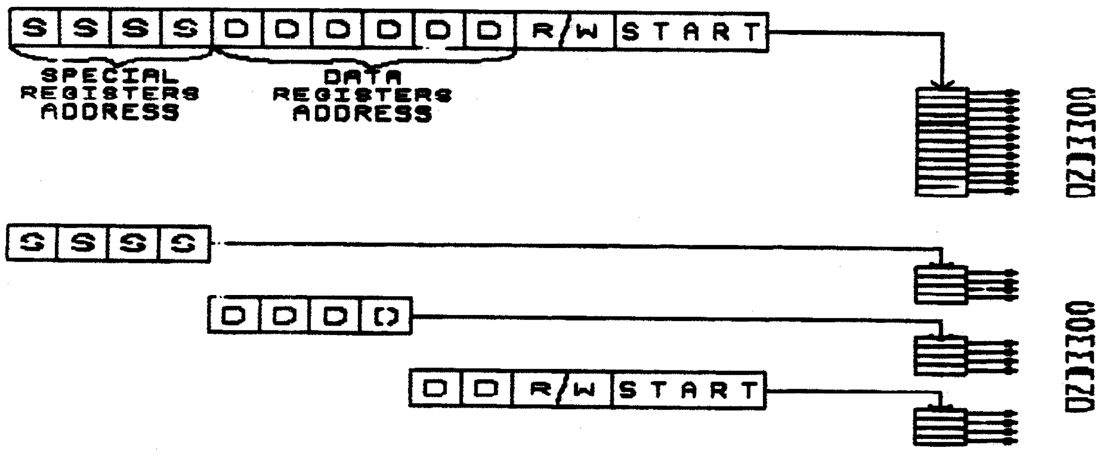
Let us now look closer at the register store controller itself. All it does in every cycle is to take in two addresses from the control computer and do a transfer between



Picture showing how register store relate to other elements.

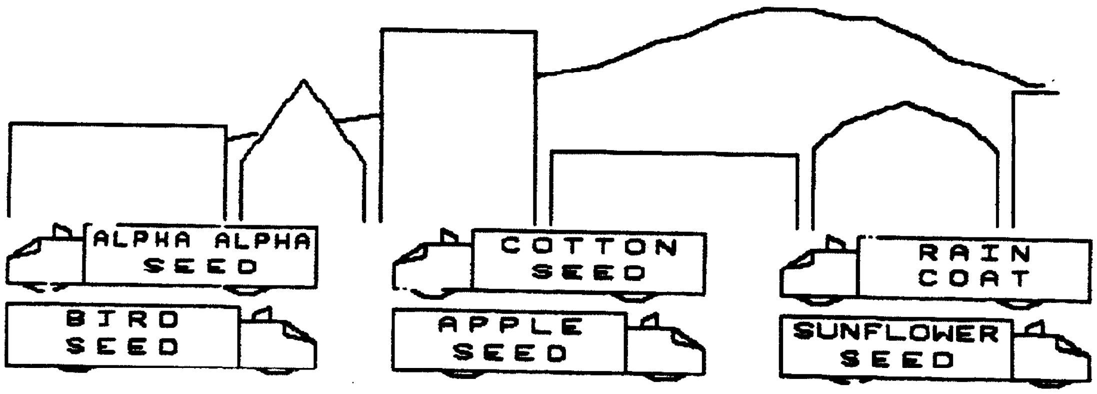
the selected data registers and the selected special register. When the control computer asks it to do a data transfer, it does a data transfer without regard for what such a move may be for. It may be used to refresh a particular data register; it may be used to take a result from the AU; it may be used to write a word into a particular communication port; it may even be used to mess up the node by creating conflicting writes into a dual port register; but what ever it does, it is the control computer's business. It is the control computer's task to generate each elementary move command from complicated instructions, and send them out at the right moments relative to the commands for other parts of the node so that right data appear at the right place at the right time - a complicated task that would otherwise have to be duplicated in every processing node.

Since the functions of the controllers are few and simple, the commands for the register store are also simple; they are made of only two addresses, a direction flag and a start signal. All the controller has to do is to shift in the command serially, interpret the direction flag, and do a data transfer when the start signal is detected. Unfortunately, since the length of this command depends only on the length of the addresses needed to select the registers, and since the command is to be transmitted serially, the time it take to transmit a command will not scale with the word length. However, we can always increase the bandwidth by using more wires to carry the



Example of breaking long command into several streams.

command. The number of wires needed would equal the number of bits the command contains divided by the number of available clock cycles between each command. As the word length becomes larger, the number of clock cycles allocated for each register command also increase; therefore at sufficiently large word length, multiple wires may not be needed.

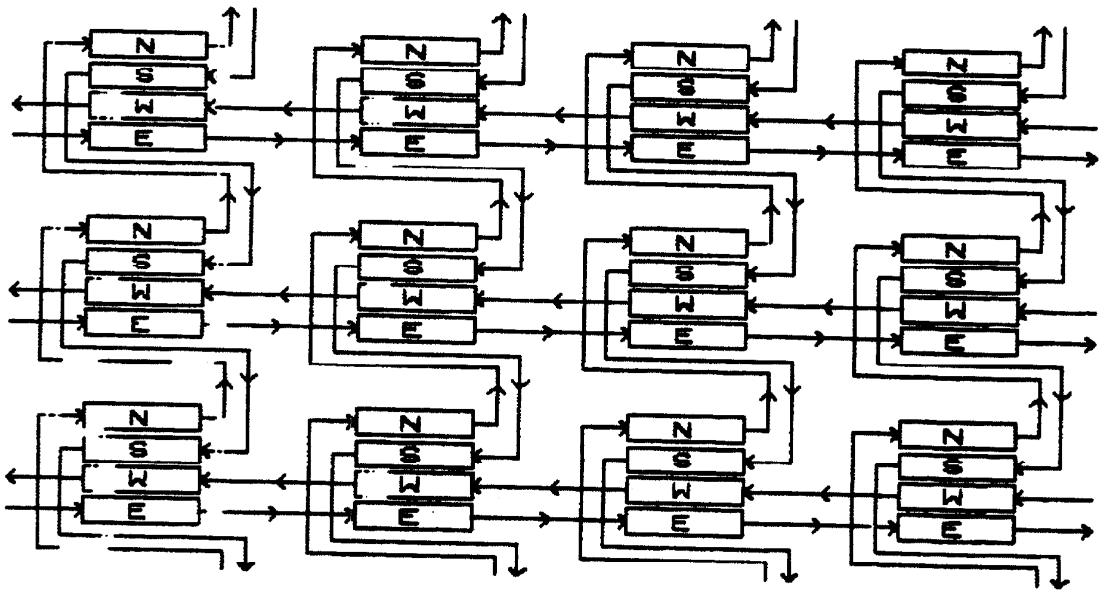


Shifting data within a bunch of RAM cells and registers is all this unit does; however, the real complexity lies hidden behind an addressing mechanism that makes everything seem simple and uniform. This uniform addressing mechanism not only makes the design of the controller simple, it also makes adding more functions to the node easy. Like our highway system, first we lay out the roads then a city grow around it, feeding on goods shipped through the highway. Furthermore, we do not care what is being shipped over them as long as they are carried on something that has wheels and rolls.

3.2. THE DATA CHANNEL

If nodes are cities, then their communication ports are the train stations and the links among them are the railroads. Actually, these ports are more like continuously rolling conveyor belts made of continuously shifting registers.

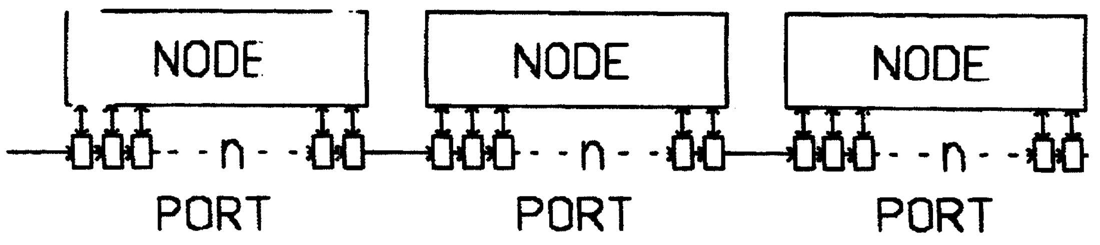
Nodes in this machine are connected in a 2 dimensional mesh; therefore each node has 4 neighbors and 4 communication ports, each for one neighboring node. These ports are named "north", "south", "east", "west" according to the orientation of their associated neighbors; the north port transmits data to to the north, the east port transmit data to the east, etc.



Picture showing how ports are connected in 2-D mesh

Besides sending data in a certain direction, these ports also receive data from the opposing directions; as the north port shifts its contents out northward, it also shifts data in from the south. Indeed, all the registers in the same row or column in the same direction are chained head to tail to form a giant shift register spanning the width or length of the array (and thus the conveyor belt).

All the node has to do in transmitting data is to put data onto the conveyor belt and pick data up some specific time later. This is conveniently done by the data move operations of the register store; writing a word into the special register corresponding to a port puts the word onto the giant shift register, and reading a word from a special register reads the current state of the shift register. For example, if there are 64 bits in each shift register, and a word is written into the east port, then if east port is read 64 clock cycles later.

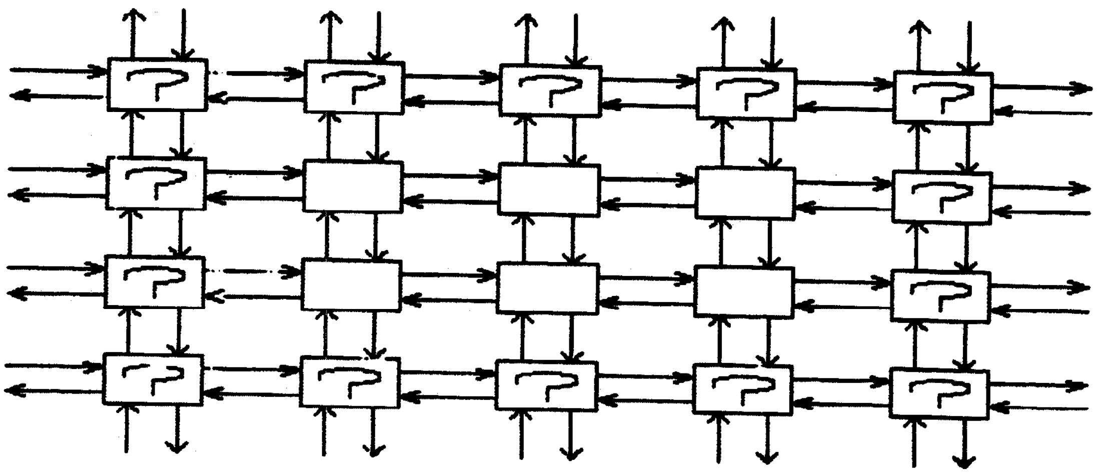


Picture showing how ports of same group are chained

The word read would be the word written by the node one location to the west; if the port is read 256 clock cycles later, the word read would be the word written by the node 4 locations to the west. Of course the communication port cycle does not have to be 64 clock periods; recall the one extra clock delay between each pair of neighbors as imposed by command pipelining. In general, whatever the length of communication cycle is, when a word is written into a communication port, it would be transported k nodes away in k communication cycles.

However, if the contents of the ports were not read at the right moment, the word obtained would be a snapshot of some data in transit, yet there is no circuit in the nodes to guard against such action. The responsibility for synchronization is placed on the control computer; it has to emit commands for data moves to the register store so that data will be read out of the ports at the right moment after it was written.

Another problem that appeared in designing this machine is the boundary problem. All square meshes of finite size have boundaries, and nodes on the boundaries have fewer than 4 neighbors; nodes on the edge have 3 and nodes on the corners have only 2. Where does a node on the north edge send north port data to, and where does it south port take data from?


Picture showing confused boundary nodes

This is a question to be answered at a later time. In this example the north port can simply be tied back to the south port forming closed loops. These dangling communication lines may also be terminated into some mass storage devices. The specific connection on the boundary will not be decided at this time because the specific configuration would probably be application dependent.

3.3. THE FLOATING POINT ARITHMETIC UNIT (AU)

The essential structure of this machine is determined by the register store and the communication ports. The definition of the structural architecture ends here. However, to be useful, something has to be done with the data being shoved around among the nodes. Since applications requiring this type of machine usually involve large amounts of number crunching using numbers of bewildering magnitude, floating point arithmetic is the most obviously useful form of operation. Other units may be added as required. The AU, like all other units, is self contained and independent; it can be grafted clean off a node and the rest of the node will still function properly because it only writes and reads three special registers in the register store unit.

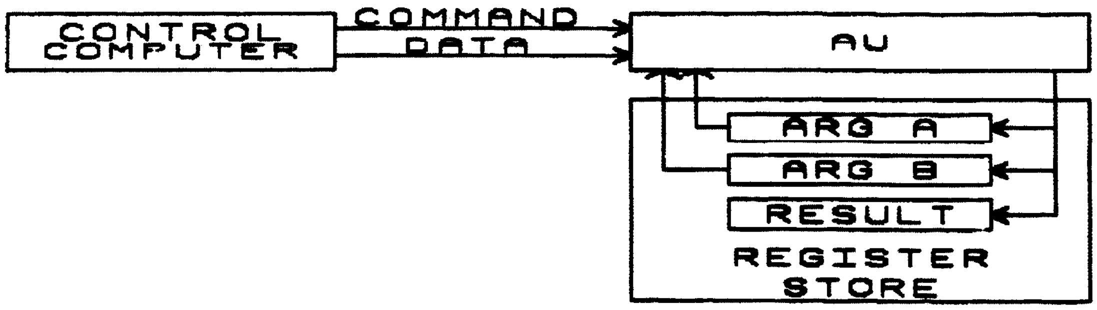
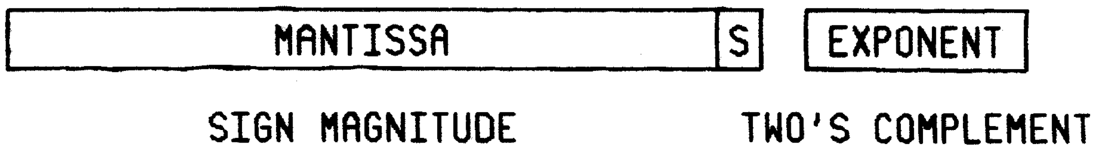


Figure showing how AU interact with other elements.

The functions of the AU are also simple; it responds to commands for floating point add, floating point subtract, floating point multiply, and the generation of an initial value for a subroutine that calculates the reciprocal of a number. Floating point subtract is identical to add with the sign of one of the numbers complemented, and will be described without distinction under the section "Addition Algorithm". Floating point multiply is described under "Multiply Algorithm". As for divide, it is implemented by an iterative algorithm described under "Division Algorithm". However, before getting into the algorithms that operate on numbers, let us look at the number being operated on.

3.3.1. NUMBER SYSTEM AND CONVENTIONS USED IN ARITHMETIC

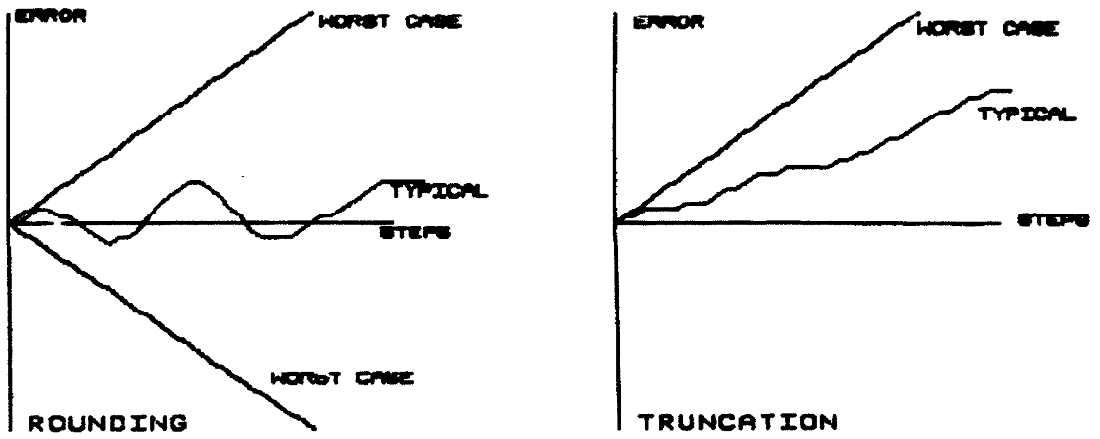
First of all, and as usual, there is a mantissa, a mantissa sign, and an exponent. The mantissa and the sign form a sign & magnitude number; the exponent however, is a two's complement number. An excess something representation of the exponent offers only a cosmetic effect of having zero represented as all zeros in all bit positions, and is identical to two's complement representation with the most significant bit of the exponent inverted.



The mantissa represents a fraction less then one, and a normalized mantissa has a 1 in the most significant bit. The exponent represents the number of times the mantissa is to be shifted to the left to obtain the fixed point representation of the same number.

All this sounds typical enough; however, in many representations, efforts are made to include numbers that cannot be normalized; here they are considered too small and are simply set to zero. If numbers of that small a magnitude are badly needed, then a simple one bit extension of the exponent will more than solve the problem. Also in some other forms of floating point representation, notably the IEEE floating point standard, the most significant bit of the mantissa is implied; but similarly, a one bit extension in the mantissa will achieve the same accuracy without the trouble of the implied bit. For machines with a very long word an extra bit is a small price for avoiding time consuming operations.

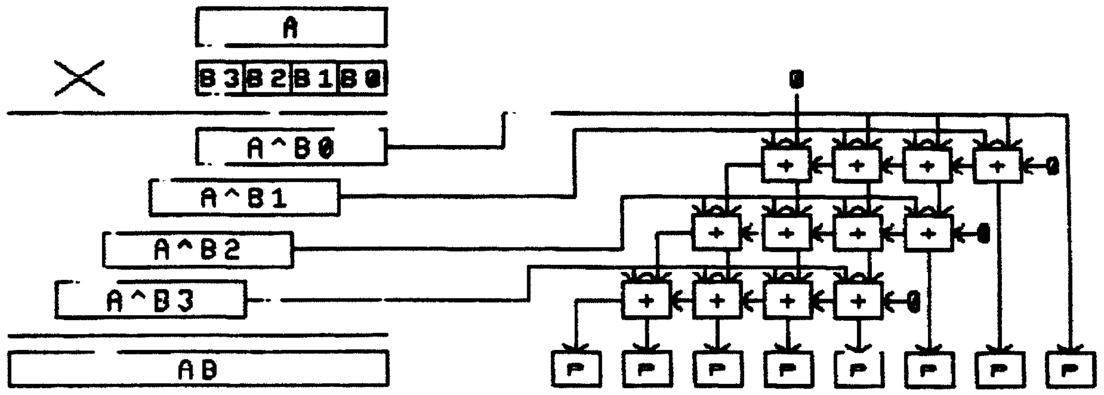
Similarly, truncation is chosen over rounding. Rounding has an advantage over truncation because rounding gives an unbiased error, thus the error accumulated is on average smaller because consecutive errors tend to cancel. However, rounding is slow; it causes a delay equal to that of a parallel add. Truncation turns out to be not so bad an alternative. Indeed, by extending the mantissa one more bit, we can have the same average and worst case error as with rounding. This slightly unorthodox yet very clean and simple system turned out to be the inevitable choice when the conditions central to the designing of this machine are considered.



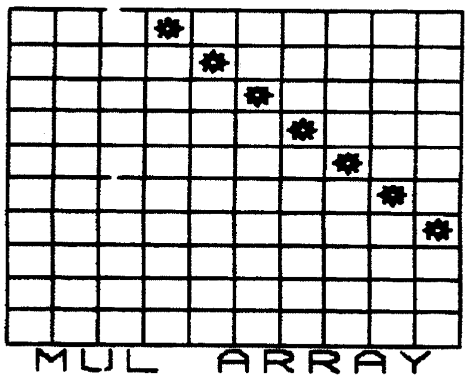
Accumulated error in rounding (left) and truncation (right).

3.3.2. MULTIPLY ALGORITHM

The heart of every floating point multiply mechanism is an integer multiplier. Made of an array of adders, the parallel multiplier is the simplest of all multipliers. However it is inefficient because it does a multiply in O(n) time while using up O(n*n) area; each adder in the multiplier is active 1/n of the time. Furthermore, with an area complexity of n*n, the size of the multiplier circuit may soon get out of hand as the demand on word size grows.



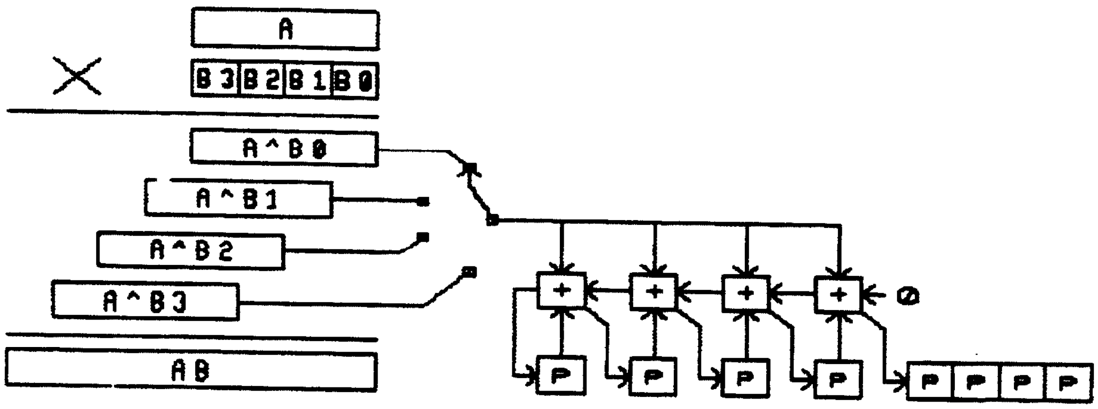
Idealized parallel multiplier



Idealized array representation of multiply process

One way to reduce multiplier area is to pipeline the multiply array. As seen in the array representation of multiply, only those elements on the advancing fronts are active, and the front advances only one position for every adder delay. Thus, instead of having many elements sitting idle, fewer elements may be used by moving and recycling them alone the active fronts.

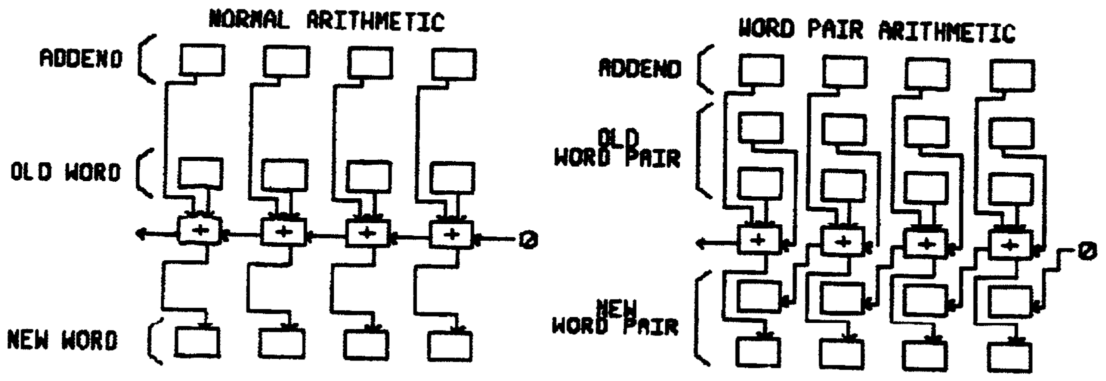
This is one form of carry save multiply; it is not the form we used however. Since not all adders are involved at all time, performance improvements are still possible. Indeed, this form of carry save multiplier multiplies in 2*n*tsum time at maximum clock speed (tsum is time need for an adder to generate the sum output) while a parallel multiplier can do it in n*(tsum + tcarry) time. A parallel multiplier would be faster because tcarry < tsum. The form of carry save multiplier used here can perform a multiply in the same n*(tsum + tcarry) time as in parallel multiplier. It can best be explained using a new number system. Let us look first at the conventional shift & add multiplier.



Idealized word serial shift & add multiplier.

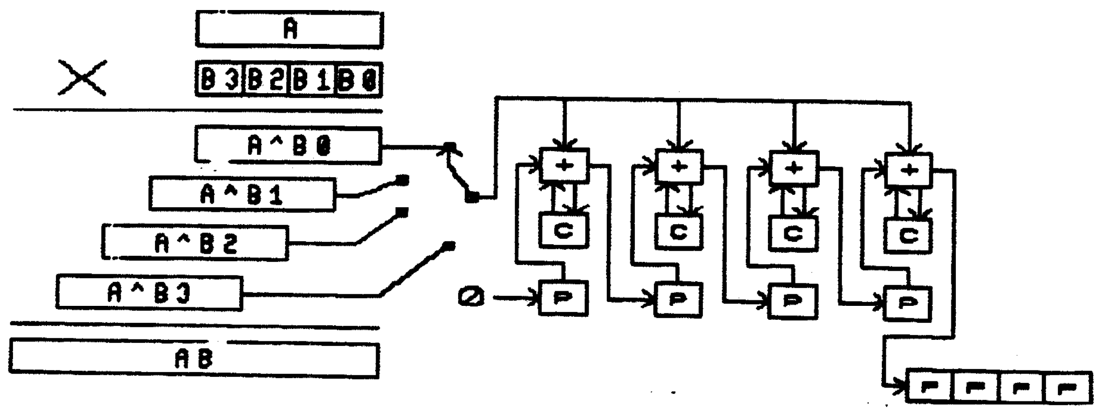
In this multiplier, a parallel adder is used to add up the multiply array one row at a time. However each add takes n*tcarry time, and each multiply takes n*(n*tcarry + tsum) time. Though it has an area complexity of n, it has has an unacceptable time complexity of n*n. It is slow because binary addition require full carry propagation. However, if we were to use a different number system, addition may not be that time consuming, for instance, the word pair number system.

In such a system, each number is made of two words. Its value is equivalent to that of a normal binary number if the sum of the two words making up the word pair is equal to that binary number. This is a non-unique number representation; we can subtract a number from one word and then add it to the other and then have a new representation of the same number. Interestingly, suppose we add the two words bit-wise and store the sum of the k-th bits at the same location in one word, and the carry at the next higher location of another word, we would get the same number back in a different representation. In it, the CARRY that is normally propagated by adding into the next bit position is SAVED, thus a full carry propagation is avoided.



Addition in binary (left) and word pair (right) number system.

In this system, adding a ordinary binary number to a word pair takes tsum time instead of n*tcarry. All we have to do is to add all three numbers bit position by bit position. At location k, the sum of all three bits goes to make the bit at location k of one of the result words, while the carry of the three bits goes to make the bit at location k+1 of the other word.



Idealized carry save multiplier.

Using this numbering system, and using a carry save adder instead of parallel adder to accumulate the numbers in the multiplier array, it takes only n*tsum time to do multiply if clocked at maximum frequency. However, at the end, in the accumulator the result is represented in the form of a word pair. Converting it to binary with a parallel add takes an additional n*tcarry time. The total time of n*(tsum+tcarry) is the same as parallel multiplier. Thus carry save multiplier has both an area complexity and a time complexity of n. Using this multiplier, the multiply mechanism can be described:

1 cycle:	load operands.
m cycle:	carry save addition.
k cycle:	parallel add to get binary product.
1 cycle:	adjust mantissa and exponent.
1 cycle:	save result.

Where m is the length of mantissa and k is the number of cycles needed to do a parallel add. The sign and exponent operation can occur concurrently with this sequence. Mantissa adjustment takes only one cycle because all numbers are assumed to be normalized and the worst that can happen is (0.1)(0.1) = 0.01.

3.3.3. ADDITION ALGORITHM

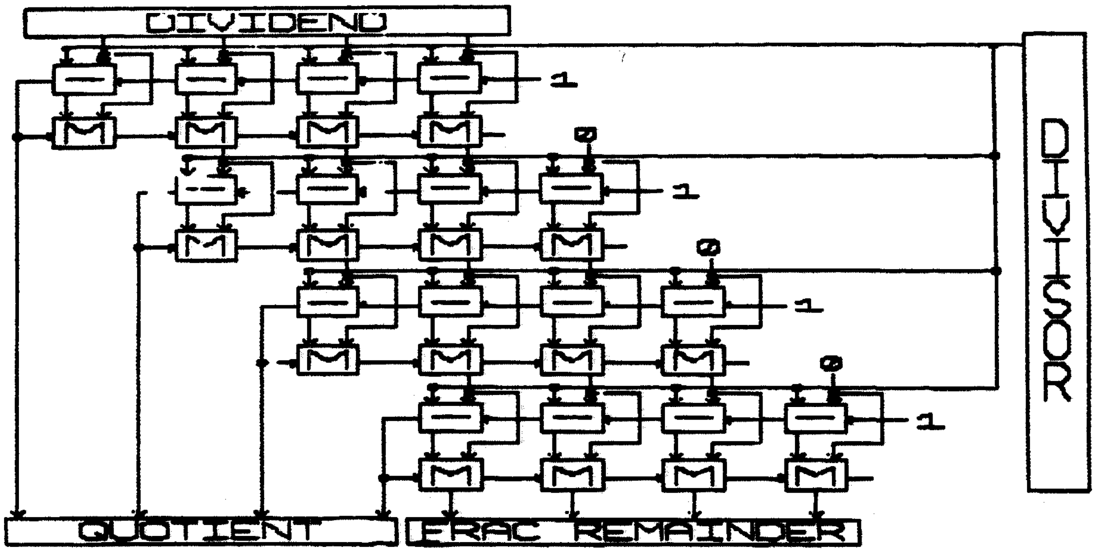
The word pair numbering system as presented in the previous section has its shortcoming, however. If a machine is to use such representation entirely, it would take twice as much memory to store the data. Furthermore, magnitude comparison is difficult; the numbers have to be converted to binary format first. Thus such representation is best used inside high speed AUs as unsigned accumulators while the rest of the machine use normal binary numbers.

The add operation is rather typical. Ordinary floating point adders require a pre-adjustment and post-adjustment of m cycles each, m being the length of the mantissa. Adding the time to do parallel add and a few small steps makes addition almost twice as long as multiply. Unlike the multiply, the bulk of its time is taken up in the less demanding mantissa adjustment, which can easily be trimmed to match the speed of multiply. A simple way is to use shifters that can shift by two locations in one cycle. With this the add timing becomes:

1 cycle	load operand.
1 cycle	compare exponents.
1 cycle	adjust by 1 for if difference is odd.
m/2 cycle	adjust by 2 for the rest of difference.
k cycle	parallel add or subtract.
m/2 cycle	adjust by 2 until one of the highest two bits is one.
1 cycle	adjust by 1 if msb is 0.
1 cycle	store result.

3.3.4. DIVISION ALGORITHM

Of the four elementary operations on numbers, integer or floating point, division is the most troublesome not only because of the possibility of dividing by zero, but also because of the complexity in the operation itself. Division is complicated because it must produce one quotient digit at a time, and each digit takes a full carry propagation.



Typical parallel divider

However, divisions are so rare that the use of specialized dividers in most machines cannot be justified, certainly not in a SIMD machine where every piece of circuit is duplicated thousands of times through out the system.

Replacing the divider with a successive approximation algorithm using only add, subtract, and multiply is an attractive approach. One suitable and popular algorithm is the Newton's method of successive approximation.

```
NEWTON'S METHOD:
    To find x such that f(x) = 0
    1) Xo := guess an initial value
    2) Xo := Xo - f(Xo)/f'(Xo)
    3) repeat 2
```

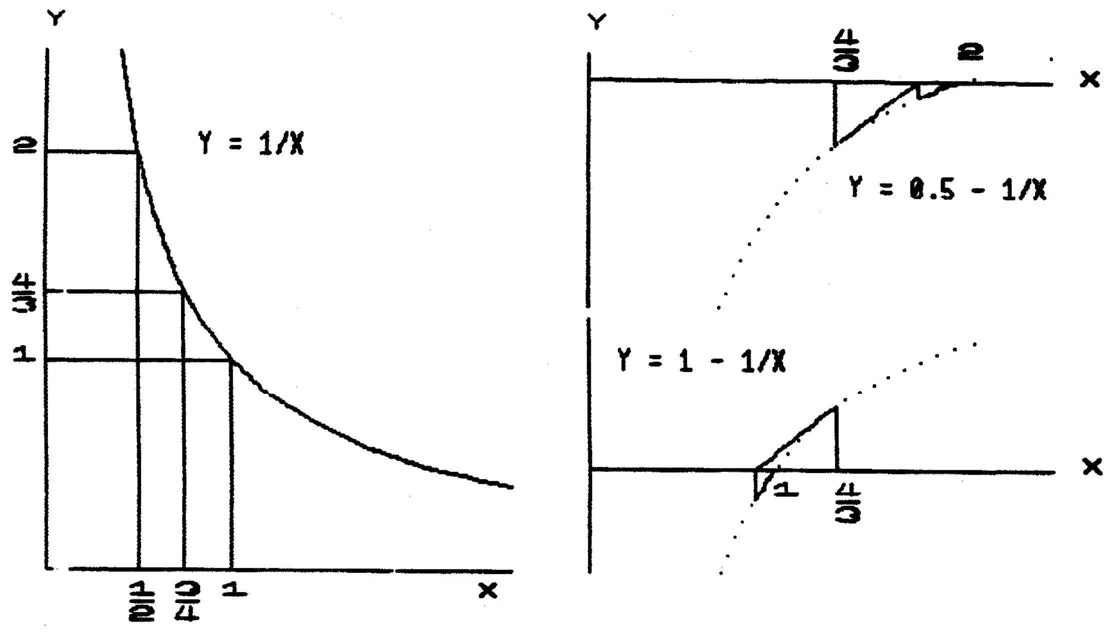
All we need is a nice function that crosses zero at the target value and an initial guess close enough to the target value for the iterations to converge. In addition, to be useful in an SIMD machine, we need a way to generate a good initial guess that guarantees convergence in a limited number of iterations. And, to be useful at all, the resulting iteration step should involve only functions that we know how to compute.

Division will be simpler if we break it down into an inverse operation and a multiply operation. Now suppose we are given A and want to find 1/A, we need a function f(x) such that the function depends on A and has 1/A as its root. Many such functions exist but not all of them are good because some yield iteration steps that themselves require divisions. Of all that qualify, the function $f(x) = A - 1/x$ is a good choice. It yields this simple iteration step: Xnew = Xold(2-A*Xold).

```
NEWTON'S METHOD FOR FINDING INVERSE OF A:
    f(X) = A - 1/X
    f'(X) = 1/(X*X)
    1) Xo := (1/1.5) * 2^(1 - exponent of A)
    2) Xo := Xo(2 - A*Xo)
    3) repeat 2
```

Newton's method roughly doubles the number of significant digits for each iteration. Taking advantage of that, many computers that use Newton's method for division use a short lookup table to obtain an initial guess that is accurate to four or five digits, trading off chip area with the first few less productive iterations. In this machine, however, a lookup table for every node is not feasible; instead, a good guess with approximately one digit significance can be generated with a special AU command. Assume we need to find the inverse of a non-zero number, since every non-zero number is normalized, the mantissa is somewhere between 0.5 and 1.

Therefore, given a number M * 2^E a good initial guess would be 1/(0.75 * 2^E) or (1/1.5)*2^(1-E). To generate this number, we can simply replace the mantissa with the bit pattern corresponding to 1/1.5 and the exponent with (1 - exponent). Tests show with this initial guess, 6 iterations are enough for full precision in a 64 bit floating point system. Indeed, it takes about log(n) iterations for a word length of n, and it works for all numbers except zero and those numbers that cause overflow or underflow. Zero can be detected during generation of initial guess, while overflow and underflow can occur during the subsequent iteration steps.



Left: inverse of 3/4 is 4/3
Right: two sample runs using Newton's Method

3.4. COMMANDS

At this point, we have gone through the bulk of what we know about this machine. Conspicuously missing, however, is the description of the control of the node, which is also a specification of the control computer. As stated before, the control computer must emit commands at the right moments relative to each other to bring about meaningful use of this SIMD machine.

For instance, if each node in the machine is to add a number, located two nodes away, to another number stored locally, each node must do the following things:

A data move to put the word into the proper port;

A data move to fill one of the AU registers;

Two port transfers to carry the word two nodes away;

A data move to read the received word;

A data move to put this word into the other AU register;

An AU operation to calculate the sum;

And finally a data move to store the result.

All this may take place in the sequence as in the figure on the following page. The upper half of the figure represents the operation within each of the three units.

The first row

is for the AU, occupied by the box labeled "AU o".

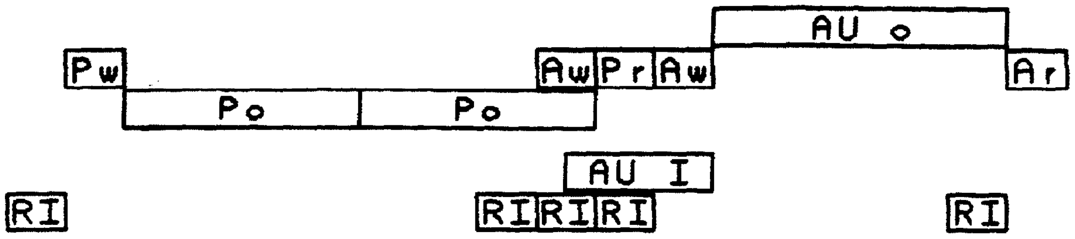
The second row

is for the register unit, marked by boxes labeled with "Pw", "Pr", "Aw", and "Ar" representing write and read, to and from the port and AU respectively.

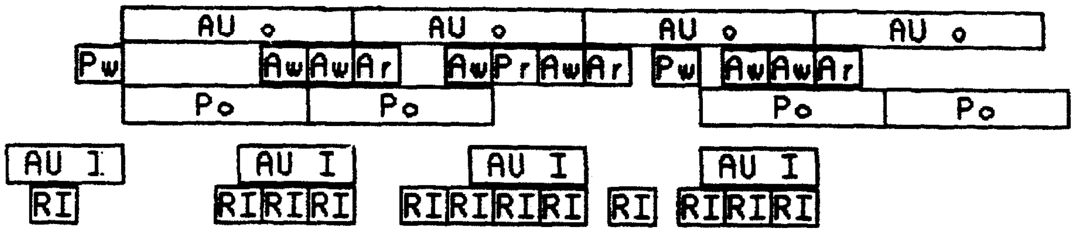
The third row

is for the communication ports, occupied by the boxes labeled "P o".

The lower half represent the instruction being transmitted. The first row is for the AU and the second is for the register.



Since the control computer has absolute control over the sequence of operation, very high degree of parallelism can be achieved. Following is a typical segment in the operation of the machine.



4. CONCLUSION

During the past several months, I have been working with a team from the VLSI Design Laboratory class to develop a complete design and layout for a supermesh node. Although not yet complete, it appears that such a node is readily designed in about 2500 by 1600 or 4M square lambda, and will operate at a 20MHz clock rate in 3um nMOS technology. Taking my advisor's very conservative metric of $5 per million square lambda, a node would cost $20, packaged and powered (in system price). With a 4usec 64 bit floating point operation time, one sees that such a system does indeed reach an extreme point in cost/performance, that is about $80 per megaflop. This number might be compared with about $140,000/megaflop ($7 million / 50 megaflop) for the Cray-1S. Although the comparison is admittedly not fair, the Supermesh delivers megaflops at a price about 2000 times less than that of Cray.

Though the Supermesh project is far from complete, it is my deepest wish that what we discovered during the past year would spark further research into this and other related fields for the benefits of all mankind.