# A Tree Machine for Searching Problems[1]

Jon Louis Bentley[2]
H. T. Kung
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

30 August 1979

## Abstract

In this paper we describe a new tree-structured machine (suitable for VLSI implementation) that solves a large class of searching problems. A set of N elements can be maintained on an N-processor version of this machine such that insertions, deletions, queries and updates can all be processed in 2 lg N time units. The queries can be very complex, including problems arising in ordered set manipulation, data bases, and statistics. The machine is pipelined so that M successive operations can be performed in M-1 + 2 lg N time units. In this paper we will study both the basic machine structure and the actual implementation of the machine.

---

[1]This research was supported in part by the Defense Advanced Research Projects Agency under Contract F33615-78-C-1551 (monitored by the Air Force Office of Scientific Research), in part by the National Science Foundation under Grant MCS 76-236-76, and in part by the Office of Naval Research under Contract N00014-76-C-0370.

[2]Also with the Department of Mathematics.

## 1. Introduction

Very Large Scale Integrated circuitry (VLSI) has been increasing in speed and decreasing in size at an amazing rate over the past decade, and it promises to continue at this rate far into the next decade (see Mead and Conway [1979]). In this paper we will describe a tree-structured machine for solving searching problems that is ideally suited for implementation in VLSI. The searching problems that the machine solves arise in a number of applications areas (including ordered set manipulation, data bases and statistics), and it is able to solve all of the problems very efficiently.

Before describing this machine in detail, it is helpful to characterize its contribution in general terms. The authors believe that there is a spectrum of impacts that advances in VLSI technology will have on computer architecture. At one extreme, this technology will allow conventional architectures to be implemented as smaller, faster and cheaper machines -- this will lead to more sophisticated interconnections of conventional machines (see, for example, Swan, Fuller and Siewiorek [1977], or Sequin, Despain and Patterson [1978]). Also at this end of the spectrum will be minor (register level) architectural changes that exploit certain features of VLSI; this area has been explored by Sites [1979]. At the other extreme, VLSI architectures have been proposed that are radical departures from the von Neumann tradition (see, for example, Backus [1978], Mago [1979] or Wilner [1978]). In this paper we will investigate an approach that lies between these two extremes: a high-performance, special-purpose, non-von Neumann computing device that is designed to be used in conjunction with a conventional computer. In general, such devices should be constructed only when they solve a problem satisfying two criteria: the problem should currently consume large quantities of computer time, and the proposed special-purpose device must be much more efficient than conventional ways of solving the particular problem. When such a problem is identified it is reasonable to augment a general-purpose computing system with a special-purpose device for solving the problem; the structure of such a system is depicted in Figure 1. Many such special-purpose devices have recently been

proposed; see, for example, Kung [1979] and Kung and Leiserson [1978].

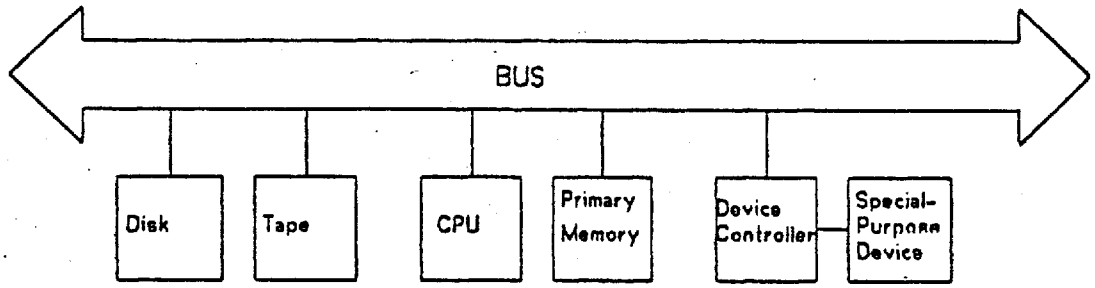



Figure 1. General system structure.

In this paper we will investigate a special-purpose machine for solving searching problems. This machine is described at an abstract level in Section 2, where we will also review some necessary background in searching problems. An architecture (that is, a user's view) of the machine is described in Section 3, and issues of implementing that architecture in VLSI are discussed in Section 4. Conclusions are then offered in Section 5.

## 2. The Abstract Machine

In this section we will investigate the tree-structured searching machine at an abstract level, apart from the details of architecture or implementation. The general searching problem it solves calls for maintaining a *file* of fixed-format *records*. We must be able to perform the operations of *inserting* a new record into the file, *deleting* an existing record from the file, *updating* records in the file, and *querying* the file to answer questions. Before we examine the general searching problem, we will investigate one searching problem in particular.

That particular problem is called *member searching*. In its abstract form, it involves maintaining a set of elements so we can determine if a new element is a member of the set. In concrete applications, other information is usually also requested. For example, after finding that a particular social security number is a member of a set of social security numbers, we often wish to retrieve other information (such as Year-to-Date taxes). We will now investigate how the tree

machine solves the abstract member searching problem, and then return in the next section to the complicating issues that arise in applications.

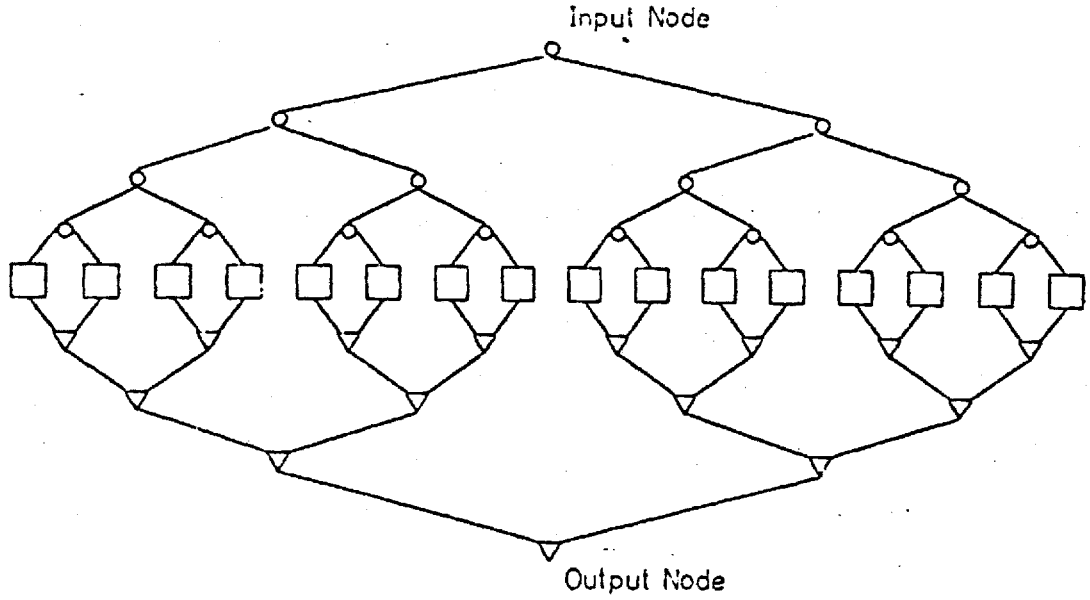



Figure 2. Structure of the tree machine.

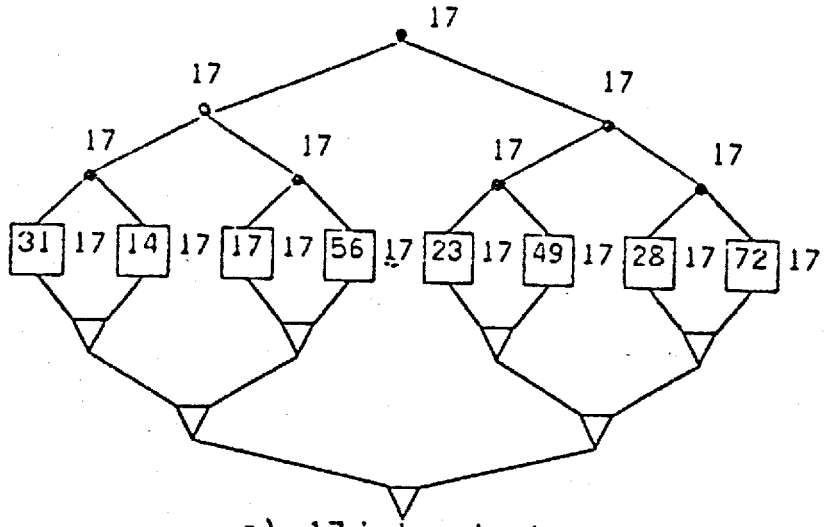
The basic organization of the tree-structured searching machine is depicted in Figure 2. There are three kinds of nodes in the machine: circles (which broadcast data), squares (which have limited storage and computation power), and triangles (which "combine" answers to queries). A set of N elements is stored in this machine by placing each element of the set into a distinct square node of the tree. Consider now the problem of performing the member search to answer the query "Is 17 an element of the set?". We accomplish this by inserting 17 into the input node and broadcasting it down the tree -- lg N steps later the value 17 will arrive at all of the squares. This situation is illustrated in Figure 3a. At that point we compare the values stored in each square to 17 and set a bit to one if the value is equal to 17 and zero otherwise; this is shown in Figure 3b. We can now combine the bits together through the bottom portion of the network by letting each triangle compute the logical *or* of its two inputs, as illustrated in Figure 3c. So after a total of 2 lg N time units have passed since the query was posed, a single bit emerges from the output node telling whether or not 17 is an element of the set. We have thus described a procedure for determining whether a given object is a member of the

set whose elements are stored in the square nodes.

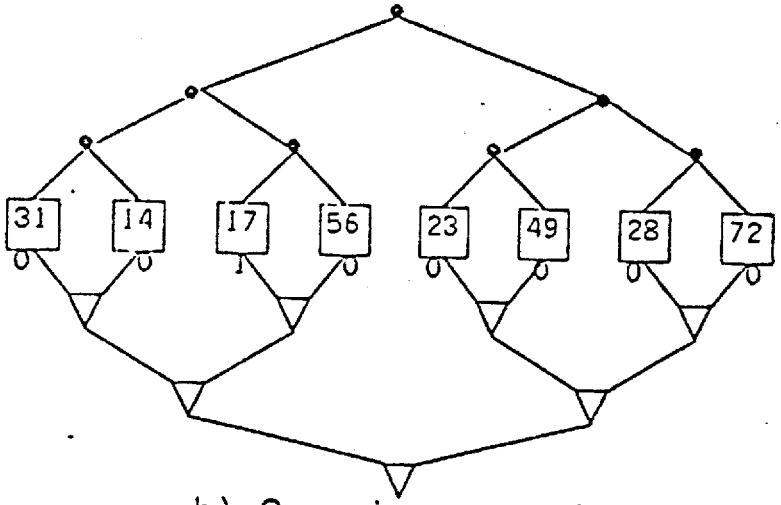
It is important to note that the tree machine has a very regular data flow: the data moves in discrete steps in only one direction (from the input node to the output node). Thus if many successive elements are going to be tested for membership in the set stored in the square nodes, then the process of answering those queries can be pipelined. As the value of the first element to be tested is going down the tree, the next value can follow one step behind, and so on. If M successive tests are performed in this manner, exactly M-1 + 2 lg N time units pass between the entry of the first query at the top of the tree and the exit of the last of the answers at the bottom of the tree.

The tree machine is able to solve many problems besides member searching. For example, if a multiset of elements (that is, a set in which one element can appear many times) were stored in the square nodes of the tree, we might wish to count how many times a given object appears in the set. We proceed exactly as we did for member searching, first broadcasting the given element through the circles to the square nodes. We load a one into each square if its element is equal to the given object and zero otherwise, and then combine the answers by letting the triangles *sum* the values of their inputs. Another example is given by nearest neighbor searching. If we wish to find the distance to the element of the set that is closest to 17, then we do the following: broadcast 17 through the input node to all squares, subtract the value stored in the square from 17 and take the absolute value of the difference, and finally take the minimum of all those values by having the triangles return the *minimum* of their two inputs. As for member searching, for both member counting and nearest neighbor searching, we can answer a single query in 2 lg N time and a series of M queries in M-1 + 2 lg N time.

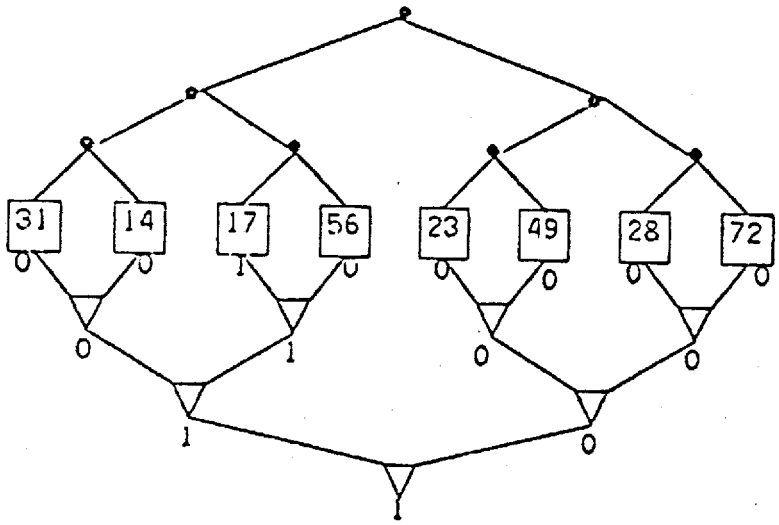
In general, the tree machine can solve any problem that can be phrased as computing some function over every element in the set (such as equality or absolute value of difference) and then combining the values of those functions by some associative, commutative binary operator. For example, the rank of an element X in




a.) 17 is broadcast.




b.) Comparisons are made.




c.) Answer is returned.

Figure 3. A member search.

a set (that is, the number of elements in the set less than X) can be calculated by

storing in each square a one if the element is less than X and zero otherwise; the final answer is then computed by having the triangles add their inputs. Other problems defined on totally ordered sets that can be solved by the tree machine include predecessor (what is the greatest element less than the given?), successor (what is the least element greater than the given?), and minimum (what is the least element in the set?). In general, the tree machine can solve all of the "Decomposable Searching Problems" defined by Bentley and Saxe [1979]. That reference contains both an algebraic definition of the class and a list of over twenty particular searching problems in the class.

The tree machine is also able to answer much more complicated kinds of queries (of the form that arise in data base applications, for instance). Suppose, for example, that every square node of the tree contains a record with ten keys. We might want to know how many records there are in the file with first key equal to a given value, the second key at least as great as the third key, the fourth key in a certain range, and so on. This type of query is easily answered: we merely broadcast each of the conditions down to the square nodes, keeping track in each node of whether it has satisfied all the conditions shipped so far. We load a one if all conditions have been satisfied and a zero otherwise, and combine by having the triangles sum their inputs. Many applications call for a list of the satisfying records instead of merely their count, and this can be accomplished by letting the triangles compute the *union* of their inputs. This can be viewed intuitively by observing each triangle independently, and imagining a person "tapping" the entire machine at each time step. As each triangle is tapped, there are three cases to consider: if it has no items in its inputs, it reports that; if it has one item, it returns it; and if it has two items, it returns only one (delaying the other until the next tap). This "tapping" process continues as long as there are elements that have yet to be reported. (Note that to compute unions in this manner, the pipelining must be carefully designed to ensure that no "overflow" occurs.)

Having discussed searching at some length, we will now turn to the issues of maintaining the set of elements stored in the square nodes. A tree machine with N

square nodes (where N is a power of two) can store up to N records. A new record can be inserted into the set by placing it in any unused square. We find such a square by having each circle keep track of the number of unused square descendants of each of his two sons. When a request comes to the root for a new (unused) position, he passes the request to one of his sons with unused square leaves, and so on. Mechanically, this is accomplished by turning off all of the squares except the one finally chosen as the holder of the new record; this square is then loaded with the desired data. Note that a single record can be inserted in lg N steps, and a set of M records can be inserted in M-1 + lg N steps.

Another maintenance operation is that of updating a set of records: this can be easily accomplished by broadcasting the conditions that the changed records must meet, turning off all processors that do not meet the conditions, and then making the desired changes. (Although the update set will often have just one element, an example of a "mass update" might be processed on the first of the month: for all salesmen with Month-Of-Starting-Employment equal to This-Month, add one to Years-Of-Service.) To delete a single record we set a flag in its square node saying that it is unused and then adjust the counts in all of the circles above it. This can be accomplished either by pushing information "backward" to the top of the tree (adding one to each counter as you go), or by doing a dummy reinsertion of that element, and modifying the counters on the way down. The time for either of these operations is proportional to lg N. Notice that after a set of elements in squares have been identified for deletion, they can be deleted in parallel (in a single step) and all counters can be reset (by pushing the information up the tree) in lg N steps. Although having information go up the tree is handy for deletion, it does complicate the basic design severely; this feature might therefore not be implemented.

So far in our discussion each machine has represented but one set. In some applications, however, a given user might wish to represent many sets, or many users might want to use the machine independently for their respective sets. Either of these can be accomplished so long as the sum of the sizes of the sets is less than N, the number of square nodes. Although we could "slice" the machine into

sections to accomplish this, there is a much more elegant solution. Namely, a fixed portion of each record is dedicated to a "set identification field", or "SetID". To process an operation on Set 56 (or a set belonging to user 56), we have as a prelude to the operation the sequence "check SetID for equality with 56 and turn off the processor if not equal". (Notice that we are not requiring that all records in all sets be of the same format, but just that they have one field in common.) In an environment with much sharing, this prelude will occur so often that it might be advantageous to provide a single instruction that accomplishes its purpose.

Although so far we have used the tree machine to solve only searching problems, it can be applied to many other problems as well. For instance, it can be used to sort a set of M elements in time proportional to M (as long as M is between lg N and N, where N is the number of square nodes in the tree machine). This is accomplished by making two passes through the M elements: the first inserts the elements into the machine, and the second counts for each element the number of elements less than it (that is, it computes the element's rank, as we saw before). This tells precisely where each element occurs in sorted order (the output is a permutation vector), and it is then trivial to arrange the elements into sorted order. By use of pipelining, both steps run in time linear in M. Note that it was critical to phrase sorting as a counting problem, rather than as extracting the minimum, to make use of pipelining in the second step -- this algorithm essentially implements an $N^2$ algorithm in N time by using all N processors in parallel. There are many other examples of such speedups for problems that are not *prima facie* searching problems. Two such examples are computing all nearest-neighbor pairs in a k-dimensional point set (which arises in data analysis) and reporting all pairwise design rule violations in a VLSI mask (a design automation task). The application of this machine to the problem of constructing minimum spanning trees has been discussed by Bentley [1979]--he shows how an N/lg N-processor version of the tree machine can construct the minimum spanning tree of an N-node graph in O(N lg N) time, which is optimal for complete graphs. Other applications of tree-structured machines have been studied by Browning [1979].

This concludes our discussion of the machine at an abstract level, and we can now state the properties that a concrete embodiment of the machine must possess. There must be three kinds of nodes in such a machine: circles, squares and triangles. The circles must broadcast data and have a small amount of state (namely, to remember how many unused squares are descendants of each of their sons). The only processing required of a circle is incrementing or decrementing by one. The squares, however, must have substantial memory and computation power. Each square must have enough processing capability to handle the most difficult kinds of queries and updates desired and (usually) enough memory to store the largest record in most applications. The triangles must be able to combine answers. Most of the "combinators" we desire are very simple to implement; these are *and*, *or*, *min*, *max*, and *plus*. The only complicated combinator is *union*, and we are willing to "turn off" pipelining in the presence of that operator.

## 3. An Architecture

In Section 2 we described the tree-structured searching machine at an abstract level, ignoring many issues of implementation. In this section we will move one step closer to an implementation, and describe a particular architecture (that is, a user's view of the machine) realizing the abstract machine. It is essential that the reader understand that the architecture we will investigate is not proposed as the best possible architecture realizing the abstract machine of the last section. Rather, it is put forth only as evidence that there is *at least one* reasonably efficient architecture for the machine. In Section 4 we will discuss how this architecture can be implemented in VLSI.

The basic structure of the architecture we will investigate is that studied in Section 2 (illustrated in Figure 2). The flow of instructions and data in the machine is exclusively from the input node (at the top of the figure) to the output node (at the bottom) -- we will not have deletions that employ any "backwards flow". The machine is based on 16-bit instructions and 32-bit data words (which are interpreted either as integers in two's-complement or as 32-bit vectors). The top

data paths in the machine (the son links from circles in Figure 2) are 16 bits wide; the bottom data paths (links to triangles) are 80 bits wide. The entire machine operates synchronously; an operation is (perhaps) performed at each node and data is transmitted from the node to its sons on each *major cycle*. Having described the machine at this gross level, we will now examine the circles, squares and triangles individually.

The primary function of the circle on each major cycle is to broadcast what it just received to its sons. In only three contexts must it perform a more sophisticated operation. As a new element is being inserted, it must decide which way to direct the insertion (to one that has unused square leaves) and then decrement the appropriate counter by one; it then ships a "no-op" to the other son. The no-op is effected by having one bit in the instruction turned off as the 16-bit instruction is passed to the "other" son. To accomplish a deletion we insert an instruction packet of three 16-bit instructions at the root node. The first instruction is the deletion and the next two 16-bit words contain the binary address of the node to be deleted. The circles can tell by looking at appropriate bits of the address whether they should increment one of their counters as they see this instruction. The final capability the circles must have is that of passing data to the squares, without interpreting that as an instruction to them; we will return to this issue as we discuss the squares.

While the circles have the simplest architectures of the three units we will see, the squares have the most complex. The abstract machine requires that the squares be able to store data and to perform enough calculations to answer queries and perform updates. This architecture will accomplish both these tasks by shipping combinations of instructions and data to the machine. We now have to make a fundamental design decision: should the individual squares be special-purpose devices (honed for a particular view of the tree machine's task), or should they be (in some limited sense) general-purpose computing devices? We will choose the latter course, and make each square a "baby" von Neumann computer; it is important, however, to emphasize that this is merely a design decision and not an

inherent property of the abstract machine.

Each square will be a small von Neumann-like processor that receives its instructions and data from an external, 16-bit stream. An individual processor contains sixteen 32-bit words of memory, two 32-bit registers, and a vector of eight single-bit data flags (F[0], F[1], ..., F[7]). The processor also contains an eight-bit Set Identification number (SetID), and an Instruction Register. The first bit of the F vector (F[0]) is used as the "Active" bit of the processor; a special "Enable" command turns on all processors (by setting F[0] to one), and a processor can conditionally turn itself off by storing a zero in F[0]. The basic layout of the machine is shown in Figure 4 (notice that because the machine is rotated 90°, the data flows from right to left rather than from top to bottom).

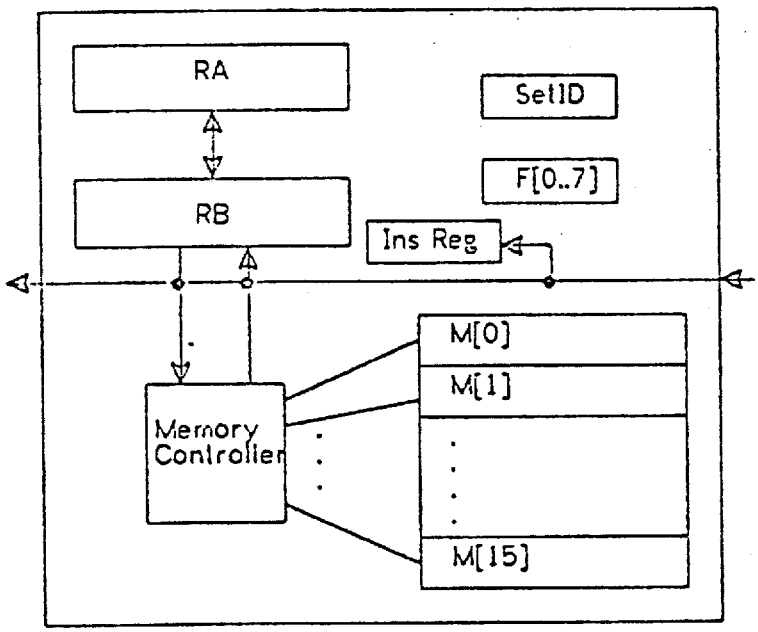



Figure 4. Components of the square.

The 16-bit instruction format for the square processor is shown in Figure 5. The first bit of an instruction processed by the squares is always zero; a one in that bit signifies an instruction that is ignored by the squares but passed on to the triangles. The two Fam bits specify one of the four families to which an instruction can belong (Arithmetic-Logical, Load-Store, Bit or Special), and the Code gives the opcode of the instruction. There is a one bit flag (Flag) in each instruction, and arguments to

the instruction are either two four-bit addresses (A1 and A2), an 8-bit string (Name) or a five-bit integer (Num). The actual instructions are described by group in an ISP-like language in Table 1. All of the arithmetic-logical instructions are zero-address instructions, combining registers RA and RB and storing the result in either RA or RB (usually RA). The load-store instructions specify one of 16 memory addresses as their operand; the data movement is then between that address and the register RB. The bit operations generally have two addresses: they combine the first and the second operands, storing the result in the first. The exceptions to this pattern are the unary not operator and the compare (comp) operation; the latter compares RA with RB and stores in the first bit (F[A1]) whether or not the values are equal and tells which inequality in the second bit (F[A2]) -- this is just a straightforward encoding of three states into two bits.

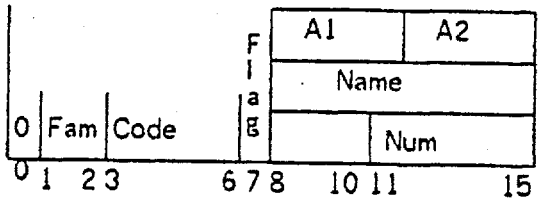



Figure 5. Instruction format.

The only instructions that are not entirely obvious are the special instructions. The *enable* instruction turns on all processors in the tree. The *ins* (insert) instruction turns on precisely one processor, turning off the rest (and decrementing the counters in the circles). The *del* (delete) instruction has no effect on the processors; it only increments the appropriate counters in the circles (the squares must ignore the two following instructions packets, though -- they are just the processor address). The *ship* instruction allows data to enter the RB register from the data/instruction stream. The Flag bit tells whether the next one or two 16-bit packets should be loaded into RB; the data can then be processed as desired. The *chksid* and *setsid* instructions are for manipulating the 8-bit SetID register; the former turns off the processor if SetID is not equal to Name, and the latter loads the SetID field from Name.

Arithmetic-Logical

| | | |
|---|---|---|
| add | → | RA ← RA + RB |
| sub | → | RA ← RA - RB |
| neg | → | RA ← - RA |
| rand | → | RA ← RA ∧ RB |
| ror | → | RA ← RA ∨ RB |
| rxor | → | RA ← RA ⊕ RB |
| rnot | → | RA ← ~ RA |
| shift Num | → | RA ← RA left shifted by Num |
| tab | → | RA ← RB |
| tba | → | RB ← RA |
| swap | → | RA ↔ RB |

Load-Store

| | | |
|---|---|---|
| ldb Num | → | RB ← M[Num] |
| stb Num | → | M[Num] ← RB |

Bit

| | | |
|---|---|---|
| band A1,A2 | → | F[A1] ← F[A1] ∧ F[A2] |
| bor A1,A2 | → | F[A1] ← F[A1] ∨ F[A2] |
| bxor A1,A2 | → | F[A1] ← F[A1] ⊕ F[A2] |
| bnot A1 | → | F[A1] ← ~ F[A1] |
| comp A1,A2 | → | F[A1] ← RA=RB; F[A2] ← RA < RB |

Special

| | | |
|---|---|---|
| enable | → | F[0] ← 1 |
| ins | → | F[0] ← this processor selected |
| del | → | (defined in text) |
| ship Flag | → | (defined in text) |
| chksid Name | → | F[0] ← SetID = Name |
| setsid Name | → | SetID ← Name |

Table 1. Instruction set for squares.

To illustrate the operation of the processors we will study two program segments for performing searches. The first segment is for member searching.

```
chksid   ThisSet   // Turn off undesired processors
ship     Two       // The next two packets hold the comparand
data_L
data_R
tab                // Put comparand in RA
ldb      KeyAd     // Put key in RB
comp     1,2       // Answer is in F[1]
```

The search key enters the RB register from the data stream and is then transferred to the RA register. The program then loads the key field of the record into the RB register (KeyAd is an integer identifying which of the 16 memory words holds the key), and makes the comparison. F[1] is then one if and only if the record's key field is equal to the data shipped in the stream. At this point, the answer can be combined in the triangle network.

The next program that we will examine arises in "nearest neighbor" searching; it computes the distance between the data and the key field of the record. Since we desire the absolute value of the difference of the key and the data, we must have a conditional step in our program.

```
chksid   ThisSet
ship     Two       // RA ← Data
data_L
data_R
tab
ldb      KeyAd     // RB ← Key
comp     2,0       // If Data≤Key, leave processor on
swap
chksid   ThisSet   // Turn all processors back on
sub                // RA ← |Key-Data|
```

The crucial step of this program is the comp instruction: if Data is less than Key then a one is stored in F[0], which leaves the processor on; the swap then interchanges key and data. The next instruction (chksid) turns all appropriate processors back on, and the subtract correctly computes a positive value. The triangles can then be instructed to return the minimum of these values.

The two code segments that we have just seen illustrate many of the aspects of

coding the tree machine. Many other examples have been coded, and all of them appear to be fairly efficient. More quantitatively, the ratio of tree machine instructions to "critical" operations in the task clusters very closely around 2.5. This statistic is evidence for the vindication of our design decision to make the squares general-purpose machines, rather than special devices tailored to the searching task domain. (Pursuing that alternative remains an interesting open problem.)

Before ending our discussion of the squares, it is interesting to compare the design of the processor with a more typical von Neumann processor. In some ways, we faced exactly the same problems: the choices of data representation, instruction formatting, operation set, and addressing were all taken from the von Neumann design space as discussed by Blaauw and Brooks [1979]. On the other hand, we avoided many of the issues faced by designers of typical machines; these include instruction sequencing, interrupt handling, and input/output control.

Before we discuss the architecture of the triangle, we must settle one more point about what we want it to do. In most applications that compute the minimum of a set (for instance), we want to know not only what the value of the minimum is but also what element has that value. We therefore have three objects associated with computing the minimum: the operation (minimum), the value, and the name (which is a 32-bit word associated with the value; its address or "key" in many applications). When combining two such objects, we take the value as the minimum of the two values, and the name from the name of the smaller value. The name is thus *inherited* from the minimum. We will also associate names with other binary operators: the name of *maximum* is inherited from the node with greater value; for *plus*, from a nonzero element; for *or*, from a nonzero bit vector (arbitrary if both are zero); and for *and* from a zero bit vector.

Having defined the concepts of value, name and inheritance, it is straightforward to describe the architecture of the triangles. They will operate on 80-bit packets: 16 bits of instruction, and 32 bits each of value and name. Computing *min*, *max*,

*plus*, *and*, and *or* are all simple. Union is a bit more detailed, but also conceptually straightforward. One aspect that we have not mentioned is the interface between the squares and the triangles; we must include instructions for transferring the contents of the RB register to the name or value field of the triangle immediately beneath it (these could be included in the load-store family). This allows us to give complete programs for answering queries. After computing the answers (as illustrated in the two segments shown above), we load them into the desired fields of the triangles, and combine them as desired.

It is important to emphasize that the architecture we have just seen is not the architecture that the ultimate user of the machine will see. Rather, there will be a hierarchy of functions available to him. At the highest level, he will be able to perform operations on sets (load a set, erase a set, for each element in the set, and so forth); at an intermediate level there are record-handling operations (defining queries or inserting, deleting and updating records); and at the lowest level there are the machine instructions themselves. At the lowest level the user can make very efficient code by knowing the details of the machine; at the higher levels he sacrifices efficiency for clean and easy code.

An important part of the implementation of this architecture is that there be a fairly sophisticated device controller for the tree machine (such as an off-the-shelf microprocessor). This controller will implement the hierarchy of functions mentioned above. This will also reduce the bus activity substantially by having the controller fetch items from main memory and issue instructions to the tree machine; it appears that having the CPU itself perform these tasks would lead to a substantial degradation in overall system performance.

## 4. Discussion of Implementation

In this section we will discuss one implementation of the architecture of Section 3 in VLSI technology. The fundamental description of the implementation is that it is bit-serial. There are two motivations for this: one, to exploit the shift-register technology of VLSI, and two, to use very few pins on packages.

The implementation of both the circles and the triangles described in the last section is straightforward. The squares are also easy to implement bit-serially. The 16-word memory is in fact a parallel shift register, 16 bits wide and 32 bits long. The two registers RA and RB are also shift registers. To load or store a word, RB and the memory shift register are shifted in parallel, and the memory controller of Figure 4 is just a multiplexor (decoding a 4-bit address to one of 16 lines). All of the arithmetic-logical operations are accomplished by putting a single-bit function box between the RA and RB registers, and then shifting the pair through it (all operations require at most one bit of memory). Notice that we have assumed that the squares have 32 minor cycles during each major cycle of the machine. The bit operations are straightforward to implement if the Flag array is just a small RAM. Estimates by experienced VLSI designers indicate that the chip area for the functionality in the square is about equal to the chip area required for the 512-bit memory. Using current technology, it is easy to imagine putting 16 squares on a single chip.

Now that we know how we will implement the individual processing elements (circles, squares and triangles), we must describe how to place them on a chip. The first simplification we will make is to consider them as standard binary trees rather than the "mirrored" binary tree of Figure 2; the unmirroring process is illustrated in Figure 6. We now face the problem of laying out a binary tree on a chip. This problem has been studied by Mead and Rem [1979], who suggest the space-economical layout illustrated in Figure 7. The amount of space used in that layout is proportional to the number of processors on the chip. Note that each edge in that layout is realized by two "wires" on the chip -- one for data going to the squares, and one for data coming from the squares.

Since only some fixed number of the processors in a tree machine will fit on a single chip, it is important that we discuss the packaging of the chips. The packaging strategy we propose is illustrated in Figure 8. There are two kinds of chips in that figure: the *leaf* chips and the *internal* chips. The leaf chips contain

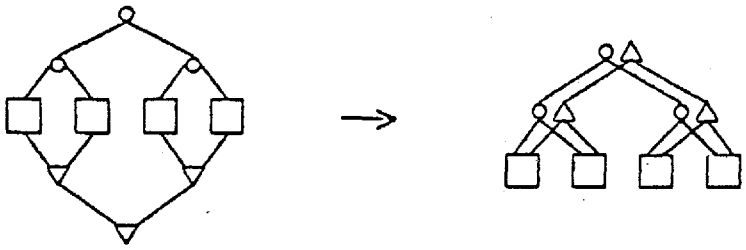


Figure 6. "Unmirroring" the tree machine.

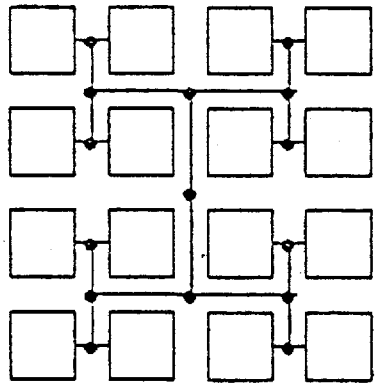


Figure 7. Tree layout on a chip.

(say) 16 square nodes and 15 circle and triangle nodes. All the communication to a leaf chip is through two wires, so the chip needs only two communications pins (besides power, ground and timing synchronization pins). Notice that this implies that with technological advances in VLSI, we will be able to place many more processors on a square chip; we are not bound by pin limitations. The internal chips would probably be constructed with seven circles and triangles on them; this implies that there is one input-output pair of wires at the top of the chip and eight pairs at the bottom. The total number of pins for this chip is therefore eighteen (plus miscellaneous pins). This chip is therefore pinbound even in today's fabrication technology; unless there are unexpected advances in packaging technology, the internal chips will probably continue to have seven or at most fifteen pairs of circles and triangles.

To get a better feeling for the size of the tree machine, we will briefly consider how one might be built today. Suppose that we put sixteen square nodes on each leaf chip, and seven circle-triangle pairs on each internal chip (both of these are

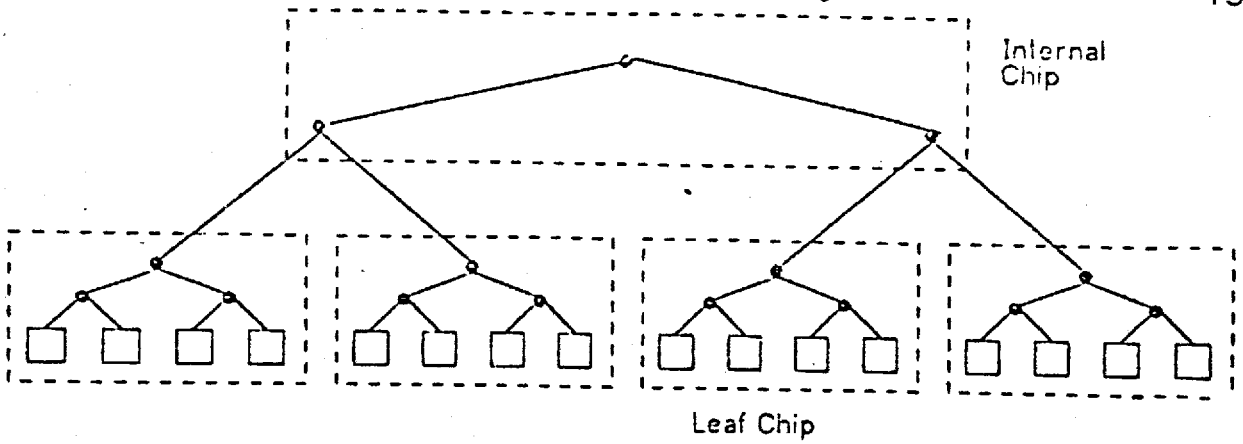



Figure 8. Two kinds of chips.

easily accomplished in today's technology). We will now put 64 leaf chips and nine internal chips on a board; this gives us 1024 square nodes. We can then put sixteen of these boards in a small cabinet, giving a tree machine of more than sixteen thousand square nodes, each holding a 512-bit record. If we assume that technology continues to double the number of components on a chip every two years, this implies that we can expect a tree machine of one million records to fit in about a cubic foot of space by the end of the 1980's.

These rough (but fairly conservative) estimates indicate that the tree machine might be one reasonable way to exploit the processing power that VLSI will give us. Before we can assert this with confidence, however, we must show that the tree machine is a wiser way to invest resources than other structures for searching. For example, might it be better to put the same resources into a large RAM memory rather than a tree machine? The authors' preliminary investigations strongly suggest that the excess cost of the tree machine compared to a RAM is very small compared to the functionality purchased, but the detailed comparison of this architecture to the RAM and its other competitors remains an open problem.

## 5. Conclusions

In this paper we have investigated the tree machine for searching problems on several levels. In Section 2 we studied it in an abstract setting and showed that it

can rapidly solve many searching problems, as well as some other problems that do not immediately appear to be searching problems. In Section 3 we saw an architecture (that is, a user's view) of the machine, and in Section 4 we saw that that architecture can be efficiently implemented in VLSI technology. Having studied the machine at these various levels, we will now spend a few moments summarizing the contributions of this work.

This machine can be compared with many other architectures. It is similar to an associative memory in many aspects, but it can perform many more operations than even the most powerful associative memories considered to date (see, for example, Lamb and Vanderslice [1978]). One might consider the square processors as forming a Single-Instruction, Multiple-Data stream (SIMD) computer, but each square is considerably simpler than most SIMD machines proposed to date. The tree machine is also superficially similar to the CASSM computer of Su *et al* [1979], but there are fundamental differences in the two machines at both the architectural and implementation levels. Two other machines to which it might be compared are the tree-structured machines of Mago [1979] and Sequin, Despain and Patterson [1978]. Both of these machines, however, are put forward as general-purpose computing devices, while our machine is much more specialized to the particular problem of searching.

Although we explored only one design path in this paper, it is important to remember that there are many variants of the tree machine. For example, in the unmirrored tree machine of Figure 6, the circle-triangle nodes could be made more powerful so that they could interact with passing data in more sophisticated ways, thereby substantially enhancing the machine's capability. So far we have investigated only binary trees; in certain applications, other branching factors may prove superior. Other interesting variants of the machine come from changing the amount of memory in a square processor; might it be reasonable, for instance, to have thousands of memory words in each square? Many other design paths remain unexplored -- in this paper he have only attempted to describe the fundamental concepts of the machine.

An interesting aspect of the tree machine is what we might call its "computational structure", which is illustrated in Figure 9. That diagram has three interpretations. First, it illustrates the tree machine itself: very small input and output channels, with massive computation going on in between. Second, it describes the searching problem: a small question is asked about a large set, giving a small answer. And finally, the figure illustrates the constraints of working with pinbound VLSI: the number of pins on a chip is very small compared to the number of functional components. The fact that the abstract structure of both the searching problem and the tree machine's solution to it closely model the medium of VLSI indicates that this approach might be very successful.

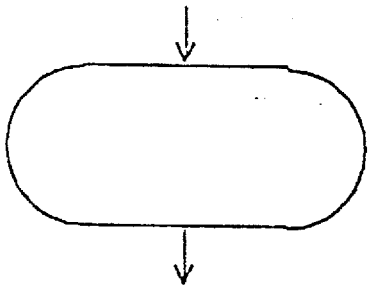


Figure 9. A computational structure.

To summarize the tree machine, the authors feel that this work has three contributions. The first is the abstract tree machine: it gives a number of nice "theoretical" solutions to a large set of problems. The second contribution is the architecture and implementation we have proposed; they indicate that this machine might be a reasonable device to build as further advances in VLSI technology occur. Finally, we feel that the "computational structure" we just investigated provides an example of the kind of argument that will justify special-purpose architectures proposed for implementation in VLSI.

## Acknowledgements

The authors would like to acknowledge the careful comments of Dorothea Haken, Dave McKeown, Jim Saxe, Dick Sites and Siang Song and the helpful criticisms

received when presenting these ideas in a seminar at Xerox PARC.

## References

Backus, J. [1978]. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM 21*, 8 (August 1978), pp. 613-641.

Bentley, J. L. [1979]. "A parallel algorithm for constructing minimum spanning trees," to appear in the Seventeenth Annual Allerton Conference on Communication, Control and Computing, October 1979.

Bentley, J. L. and J. B. Saxe [1979]. "Decomposable searching problems," in preparation. (Preliminary version by J. L. Bentley, *Information Processing Letters 8*, 5 (June 1979), pp. 244-251.)

Blaauw, G. A. and F. P. Brooks, Jr. [1979]. *Computer Architecture*, unpublished draft.

Browning, S. [1979]. "Computation on a tree of processors," Caltech Internal Memorandum.

Kung, H. T. [1979]. "Let's design algorithms for VLSI", Caltech Conference on Very Large Scale Integration: Architecture, Design, Fabrication, (January 1979).

Kung, H. T. and C. Leiserson [1979]. "Systolic arrays (for VLSI)," *Carnegie-Mellon University Computer Science Research Review, 1977-78*, pp. 37-57. To appear in Mead and Conway [1979].

Lamb, S. M. and R. Vanderslice [1978]. "Recognition memory: low cost content-addressable parallel processor for speech data manipulation," presented at the Acoustical Society of America and Acoustical Society of Japan Joint Meeting, Session BB, Honolulu (29 November 1978).

Mago, G. [1979]. "A network of microprocessors to execute reduction languages," to appear in *International Journal of Computer and Information Sciences*.

Mead, C. A. and L. A. Conway [1979]. *Introduction to VLSI Systems*, to appear.

Mead, C. A. and M. Rem [1979]. "Cost and performance of VLSI Computing

Structures," *IEEE Journal of Solid State Circuits SC-14*, 2, April 1979, pp. 455-462.

Sequin, C. H., A. M. Despain, and D. A. Patterson [1978]. "Communication in X-tree, a modular multiprocessor system," *ACM 78 Proceedings*.

Sites, R. L. [1979]. "How to use 1000 registers," Caltech Conference on Very Large Scale Integration: Architecture, Design, Fabrication, (January 1979).

Su, S. Y. W., L. H. Nguyen, A. Emam, and G. J. Lipovski [1979]. "The architectural features and implementation techniques of the multicell cassm," *IEEE Trans. Comp. C-28*, 6, (June 1979), pp. 430-445.

Swan, R. J., S. H. Fuller and D. P. Siewiorek [1977]. "Cm*: A Modular Multiprocessor," *AFIPS Conf. Proc. 46*, pp. 637-644.

Wilner, W. [1978]. "Recursive machines," Xerox PARC SSL Internal Memorandum (January 21, 1978).