AN EXECUTABLE SPECIFICATION FOR THE MESSAGE PROCESSOR IN A SIMPLE COMBINING NETWORK

David Middleton [1]
ICASE
NASA Langley Research Center
Hampton, VA 23681-0001

Abstract

While the primary function of the network in a parallel computer is to communicate data between processors, it is often useful if the network can also perform rudimentary calculations. That is, some simple processing ability in the network itself, particularly for performing parallel prefix computations, can reduce both the volume of data being communicated and the computational load on the processors proper. Unfortunately, typical implementations of such networks require a large fraction of the hardware budget, and so combining networks are viewed as being impractical.

The FFP Machine has such a combining network, and various characteristics of the machine allow a good deal of simplification in the network design. Despite being simple in construction however, the network relies on many subtle details to work correctly. This paper describes an executable model of the network which will serve several purposes. It provides a complete and detailed description of the network which can substantiate its ability to support necessary functions. It provides an environment in which algorithms to be run on the network can be designed and debugged more easily than they would on physical hardware. Finally, it provides the foundation for exploring the design of the message receiving facility which connects the network to the individual processors.

[1] This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA. 23681-0001.

1 Introduction

While the primary function of the network in a parallel computer is to communicate data between processing elements (PEs), it is often useful if the network can also perform rudimentary calculations. That is, some simple processing ability in the network itself, particularly the ability to do parallel prefix computations, can reduce both the volume of data being communicated as well as the workload of the processors proper. Unfortunately, typical implementations of such networks require a large fraction of the hardware budget, and so combining networks are often viewed as impractical. For example, the message processor in the IBM RP3 computer could implement Fetch_And_Add operations. The hardware involved included, as well as the ALU, specialized associative memory to hold messages until a response was received from the main memory modules [RP3]. That research concluded that, while being a very powerful facility, a combining network did not justify the hardware costs incurred in massively parallel computers.

The FFP Machine uses a combining network, and the characteristics of the machine are exploited to allow a much simpler design [Magó&Stanat]. However, despite or perhaps because the network is simple in construction, it involves many subtle interactions that make the design difficult to verify and to use correctly. This paper describes an executable model of the network which will serve several purposes. First, it provides a complete and precise description of the network, an important precursor to constructing the network. Second, it can substantiate the network's ability to support several functions whose efficient operation is needed by the FFP Machine. Third, it provides an environment in which algorithms to be run on the network can be designed and debugged more easily than would be possible on physical hardware. Fourth, it provides the foundation for exploring the design of the message receiving facility which connects the network to the individual PEs.

Section 2 describes the characteristics of the FFP Machine that are relevant to the design of the network. Section 3 describes the required behavior of, and then derives the consequent structure of, the nodes in the network. Section 4 describes the form in which algorithms are developed, and Section 5 illustrates this process with a series of operations that are useful for the machine.

2 The FFP Machine as the context for the network design

FFP is one of Backus's Functional Programming languages [Backus]. An FP program is viewed as a string of symbols which are repeatedly altered until a result is derived. The

FFP Machine follows this model directly by holding a running program in a linear array of PEs (called the *L array*) that are interconnected via the leaves of a tree-structured network of *T cells*.

Possibly the most novel and powerful feature in the FFP Machine is *partitioning*, an operation which divides the machine's hardware resources (the L cells and T cells) into disjoint *area machines* such that every available computation has its own dedicated virtual computer assigned to it.

By the semantics of Backus's FP languages, each available computation is a contiguous subset of the whole program and is entirely self-contained. This allows the various area machines to be isolated from each other. Two important consequences follow from this: first, if messages meet, they belong together in some sense, so their interaction is intensional and the possibilities can be deliberately limited. Second, since messages that are unrelated cannot meet, the network avoids the need for associative caches that are necessary if arbitrary messages can arrive in indeterminate orders.

As a massively parallel computer (in the fine-grained sense of the term), the FFP Machine is barred from using rich interconnection networks, such as hypercubes. The leading choice for network topology is the simple binary tree, because of the logarithmic diameter. Partitioning removes most of the contention that occurs at the root of the tree: messages in each particular area machine have no effect on communication in any other area machine.

Available computations may occur at any place in an FP program. Partitioning supports this by allowing area machines to be located at any position in the FFP Machine without effect. Hence, while each area machine is also a binary tree, there are no alignment or size constraints linking it to the physical tree structure. An area machine may contain any number of the machine's L cells, starting at any position in the L array. Area machines are almost always marginally unbalanced. By accounting for this, the network can be fit within other network topologies. For example, meshes are a popular choice for massively parallel networks, and this network can be easily embedded in a mesh. While the resulting embedded trees would be significantly unbalanced (their depth being $O(\sqrt{n})$ rather than $O(lg(n))$), designers of mesh networks expect this diameter anyway, and in return would gain powerful new processing facilities for their system.

At the level of individual messages, the network is synchronizing in the sense that each node will require a message from both inputs before generating an output message.

The result is that each PE that wants to communicate will block until all PEs in the area machine are ready to join in. This characteristic is both proper from the software view and simple from the hardware view: since an area machine is dedicated to a single function, it is proper that when the computation is doing communication, all the cells should be synchronized and participate. It is simpler for the hardware because the need for arbitration decisions is avoided. Each area machine network acts like a single (very large) pipelined ALU, that takes a sequence of messages from each PE and from these, generates a sequence of result messages for each PE.

At the level of individual bits, the network is almost asynchronous, at least in a *self-timed* sense of there being no clock. Each node in the area network awaits a signal from both inputs before generating a result. In this way, the network is better able to avoid problems that come with global clock signals, such as arbitration decisions and clock skew.

That area machines can be created anywhere leads to a style of computation that proceeds without using hardware (L cell) addresses for the data. Computations are developed using a *self-addressing* style for accessing operands (the algorithm for generating this *auxiliary representation*, described in Section 5, was first debugged using an early version of this system). Indeed, computations can be relocated invisibly, during execution, to a new area machine composed of different hardware cells. For the purpose of network design, this means that messages cannot be targeted to a particular destination PE, but rather the receiver on its own must recognize and retrieve the messages intended for it.

The final network characteristic to be presented derives from two factors. First, computations may grow in size, (for example, when the result of the computation occupies more L cells than the initial expression did), and second, the left to right order of elements in a computation must be kept (in order that enclosing computations will still be contiguous and to support the self-addressing character of operation). From these two facts, it follows that when a computation grows, it may need to push aside neighboring computations, before a partitioning operation allocates it a new contiguous area machine that is large enough. Such a neighboring computation could be in the middle of a slow message wave involving an arbitrary number of messages flowing sequentially through the root of its area machine. To prevent these slow communication operations affecting other computations, *message waves* are interruptible, to be resumed in a new area machine after partitioning. Thus, each PE must be able to determine whether the messages it has sent got through, so as to decide whether to resend them.

This section merely summarizes the characteristics of the FFP Machine as they apply to the network design. A fuller description of the FFP Machine, and in particular the motivations for these choices are described elsewhere [Magó&Stanat].

3 Network behavior

From the characteristics of the FFP Machine described above, the detailed behavior of the network can now be derived. This system simulates the tree-structured network within a single area machine as it performs a single message wave. Where necessary, this network design tries to satisfy the constraints that the FFP Machine has for machine wide behavior (for example, with partitioning and interrupts). However, the effects are not particularly visible at the level that this system operates, so they are mostly ignored.

3.1 Broadcasting messages

Basically, each PE feeds a sequence of messages into the network. These sequences are processed (merged and combined), then each PE receives back a sequence of messages for its own use. Within the network, all messages head towards the root and are then broadcast back to the entire area machine. The network makes no routing decisions at all, which avoids the time and hardware costs of transferring and processing address fields (which are not available in the ordinary sense, anyway, given the "self-addressing data" philosophy of the machine design). Since there is no a priori knowledge, outside the programmer's mind, of whether messages will be combining with others, all messages must reach the area machine root anyway, so broadcasting them after that causes no additional time delay (assuming that the PEs can deal with arriving messages at the same rate as the network transfers them). Broadcasting messages directly provides for multi-casting within the area machine, and also satisfies one last important requirement. With the possibility of interrupts during a message wave, PEs must decide whether to resend messages afterwards. By arranging that interrupts do not interfere with messages after they have left the root, the incoming stream to each PE provides the acknowledge signals that let it determine which of its messages got through before the interrupt occurred.

3.2 Sorting messages

While not inevitable, it follows naturally from the broadcast design that the messages could be sorted as they approach the root of the area machine. As mentioned earlier, the self-addressing style of computation imposes the responsibility for acquiring the appropriate messages entirely on the receiving PEs. The main work of implementing FP functions,

like transpose and reverse, can be simply achieved by sorting the data in the network and then having each PE pick its message according to its position in the incoming message sequence.

Implementing sorting in the network defines some additional aspects of network behavior. Each sequence of messages must end with a sentinel, which is called the *End-Of-Wave* (EOW) message, to support the decentralized self-timed processing of messages as they move towards the area root. Messages that undergo sorting will require a *key* field to determine their order. In fact, the network allows sorting under multiple keys; for example to transpose a matrix, the entries might be sorted first by column number and then by row number. In general, the number of other fields in a message may vary, since PEs may have different amounts of data they need to transfer.

A message processor is provided in each node of the network to combine two incoming message streams into a single result stream. It has an ALU which can calculate the minimum of two messages as its result: that one, having the smaller key, wins the comparison and proceeds up to the next node. A message processor also has a *Loser register* in which it stores the maximum of the two messages. It has enough state information such that for the next comparison, it re-uses the value stored in the Loser register instead of taking the following message from the same stream that supplied the loser. The EOW messages from the two streams, being identical, will merge, and the Loser register will be marked as being empty.

3.3 Combining messages

One of the original goals was to implement a combining network, that is, a network which can perform simple arithmetic operations on messages. Messages are extended to contain an opcode, and the ALU is extended to perform other operations, such as integer *addition* for counting, bitwise operations, such as *and* and *xor*, and *minimum* since that is already present for sorting *keys*. (Now, however, it combines messages and the Loser register is left empty). Note that *maximum* and bitwise *or* can be accomplished by performing appropriate inversions in the PEs. Multi-precision *minimum* and *addition* are provided through additional opcodes. If a bit or byte serial operation is used in the network, it is the responsibility of the PEs to supply operands in the appropriate order: most significant bit(s) first for *minimum*, and least significant bit(s) first for *addition*.

Two more simple opcodes will be introduced later. While this is the full complement we anticipate needing at the moment, in general any operation might be included, although

it is worth noting that operations like floating point arithmetic would be too complex to justify their presence: such operations can be implemented fairly cheaply in the PEs together with communication operations described later for shifting data.

3.4 The structure of messages

In general, every message consists of a series of optional parts called *packets*. The network only sees packets; the overlying structure by which packets form messages is an illusion in the mind of the designers. Each *packet* consists of a *header* and a *value*. The *header* is further divided into two parts, the *type*, which includes *EOW*, *Key* and *Value* (an unfortunate overloading of the term), followed either by an *opcode* for *Value*-type packets (in which the *value* field contains the data to be combined), or a *key number* for *key*-type packets (in which the *value* field contains the key). Since multiprecision addition is provided, short *value* fields are not constraining; the size of the *value* field is generally that of an *address integer*, the word size necessary to label each PE uniquely in the machine.

3.5 Parallel Prefix calculations

Parallel prefix operations are a powerful capability that have appeared under several names in the literature, including cumulative sums, scan operations, and Fetch-and-Add, and are starting to appear in mainstream languages, such as High Performance Fortran (as "prefix" and "suffix" versions of library functions) [HPF]. For this network, a parallel prefix operation is defined in the following fashion. Given a sequence $(x_i,\ i = 0, 1, ...)$ distributed in left to right order through the PEs of an area, the parallel prefix sum under some operation, O, is a sequence of values y_i, aligned with x_i, where $(y_i = O_{j=0}^{j=i-1} x_j,$ $i > 0)$, and $(y_0 = Unit(O))$. This is the exclusive form of the parallel prefix operation; the inclusive form, where y_i incorporates the value x_i, can be achieved locally by the PE and so is ignored in the design of the network.

Parallel prefix operations can be implemented in the network by adding a second ALU to the message processor. (Timesharing the first ALU is inappropriate since both directions, up and down, will be busy at the same time.) As before, messages are combined according to an opcode they contain, as they pass up through the network, but now, in addition, the left message is kept in a (FIFO) buffer. Now, imagine that, during the downsweep of messages, each node in the network receives the combination, under O, of all values within the network to the left of the subtree rooted by that node. This value passes to the left subtree unchanged (just as it did under simple broadcasting) since it is also the sum of values to the left of that subtree. This value is combined with the message which was kept from the left subtree during the upsweep. This result, being the parallel

prefix sum of all x_i to the left of the right subtree, is sent to that subtree. In this way, the message arriving at each PE is the result y_i and the total message traffic is the same as if a single message had been sent through the network. The FIFO buffer feeding this second ALU is called the *T cell filter* since it discards all messages whose types do not mark them as parallel prefix messages.

This second ALU is identical to the first and so in particular, it expects the stream of packets coming through the filter to be terminated by an EOW packet. Since the EOW packet eventually arriving from the left subtree may be delayed by an arbitrary number of intervening packets, relying on that packet raises the possibility of deadlock. For this reason, the sub-sequence of parallel prefix packets within an overall stream of packets has its own sentinel packet (*End-Of-Cumulative-Left* or EOCL), which the filter transforms into an *EOW* packet, for the sake of the second ALU.

An interesting possibility arises with this network, which is usually absent in other designs. The rigid left to right ordering of the values x_i causes them to be combined in that order. It follows that non-commutative operations yield well-defined results, and these turn out to be extremely useful. The projection function *2nd*, used in a parallel prefix calculation, performs a right shift by a single position, that is ($y_i = x_{i-1}$, $i = 1, 2...$), moving an arbitrary number of data values in no more time than it takes one message to pass through the network. Implementing *2nd* is a trivial extension to the opcode, *minimum*. *Minimum* initializes its state prior to the comparison to show no winner* In the same way, opcodes *2nd* (and, later, *1st*) can be implemented by initializing the state to force the left or right packet to win.

By symmetry, the network can implement parallel suffix operations, which in particular will allow left shifts to be implemented by introducing a *1st* opcode. The message processor will require a second *T cell filter* to select and hold parallel suffix messages arriving from the right child. It is probably better to add a third ALU to generate the message stream being sent to the left subtree [Magó]. This ALU combines the stream arriving from above with the parallel suffix messages filtered from the upwards stream. While it is the case that the second and third ALUs never operate at the same time, using a single ALU requires merging the two filtered streams and then unmerging the ALU results to go to different subtrees; this appears to involve greater delays and hardware resources overall.

* For multi-word minimum, there is a *minimum-continued* opcode, which acts the same without initializing the state, so that any previous decisions regarding a winner will continue to apply.

Schwartz describes an extension to parallel prefix operations called a *group bit* which turns out to be essential in generating the auxiliary representation [Schwartz]. The sequence of values x_i is broken into sub-sequences over which independent parallel prefix calculations are performed, the boundaries of the sub-sequences being marked by *group bits* in the packets. How the message ALU is extended to handle *group bits* can be derived by considering a parallel prefix operation. First, *group bits* in packets must be *OR*ed together, since a boundary in the sequence in either left or right subtree is a boundary in this whole subtree.

Next, imagine there is a *group bit* indicating a boundary in the right subtree. In this case, the value coming from the right child should pass unchanged as the result from this tree: any values from the left child belong to a different subsequence and so should not contribute. This right hand *group bit* should not affect the value from the left subtree passing through the filter; it will be used later, during the downsweep, to contribute to the y_i's intended for the right subtree.

Now, instead imagine there is a *group bit* indicating a boundary in the left subtree. The related value will be in the same subsequence as the value from the right subtree (assuming the right hand packet has no *group bit*), and so they can be combined as in the original parallel prefix scheme. However, during the downsweep, this *group bit* will inhibit the packet arriving from above, whose value represents the combined values from left of this whole subtree, from contributing to the y_i's intended for the right subtree.

Finally, considering the situation where the subtrees are individual PEs, it can be seen that, for parallel prefix calculations, *group bits* mark the first element of each subsequence. From these considerations can be derived the treatment of *group bits*, which can be summarized as follows. They are logically summed into the result packet and when they occur in the *right-hand* operand to an ALU, they override the opcode and force just the second value to proceed. By similar consideration, the behavior of *group bits* can be derived for parallel suffix operations, noting that *group bits* now mark the last element of each sub-sequence.

Magó's approach to designing the message processors reduced all actions to variations specified by the opcodes [Magó], and this included the action of *group bits*. First it should be noted that nothing has been said about the behavior of the message processor when it combines packets containing different opcodes. Since no useful ability has been found for allowing programmers to mix these operators, the behavior is defined to suit other criteria and in particular, the implementation of *group bits*. It can be seen that a *group*

bit acts like the *2nd* operator for parallel prefix calculations, and like the *1st* operator for parallel suffix calculations. By simultaneously (1) specifying that the opcode in the result packet is the minimum of the two opcodes, and (2) encoding *group bits* as *2nd* or *1st* operators with minimal bit patterns, the correct implementation of *group bits* ensues. This approach, while incurring no increase in hardware complexity, does introduce one slight problem: Inside the message ALU, when combining is to occur despite the presence of a *group bit* (for example, when the *group bit* occurs on the left in a parallel prefix calculation), care must be taken finding the opcode since it has been lost from one packet (having been replaced by the opcode implementing the *group bit*). The solution used in this system is to use the left opcode for all parallel suffix operations, and the right opcode otherwise. In this way, the correct operation, overridden by *2nd* or *1st* when appropriate, will be performed.

The description of cumulative operations given earlier assumed that a unit value appropriate for the particular operation would be injected into the root of the area machine network. Partain and Magó invented an alternative which avoids any special behavior in the network to generate, or carry and then insert, such values [Partain]. Notice that the value generated at the root of the network is the cumulative combination of the entire sequence $x_{0..n}$, and in the general scheme, this is the value that is then broadcast down through the network. If the rightmost PE is programmed to send any value but marked with a *group bit*, then for parallel prefix operations, this value will reach the root unaltered. This method naturally allows any unit value to be provided. Similarly, the leftmost PE can provide unit values for parallel suffix operations.

In general, when *group bits* are also being used explicitly to operate on sub-sequences, the *last* element of each subsequence (for parallel prefix; the *first* element for parallel suffix) must also send a copy of the appropriate unit value protected by a *group bit*. In this situation, the *group bits* described earlier as abstractly starting (or ending, in the case of parallel suffix operations) sub-sequences actually become redundant.

If a *shift right* operation is performed by a parallel prefix calculation with the *2nd* operator and no identity value is provided, then the rightmost value, x_n reaches the network root, returns, and becomes y_0. The result is a *rotate right* operation. At the hardware level, this behavior is indistinguishable from deliberately using x_n as the intended unit value and protecting it with the prefix group bit, encoded as the the *2nd* operator.

In some algorithms, not all PEs have a value to contribute. For algorithms doing a parallel prefix addition, for example, PEs with nothing to contribute could explicitly inject

zeroes without difficulty. The same is not true for shift operations, since the opcodes *1st* and *2nd* do not have general identities. It turns out in this network that by sending no message whatsoever in a shift operation, a PE allows a value on its one side to shift to a destination on its other. In fact, the shift (and rotate) operation sends the value x_i not only to the PE contributing x_{i+1} but also to all the PEs in between. This is important to the FFP Machine in a larger context because empty PEs may be interspersed throughout each *area machine*, but even beyond that, this ability of values to jump over several others turns out to be useful.

A message wave may contain several independent *cumulative* messages (parallel prefix and suffix calculations), just as it can *simple* messages. For example, the auxiliary representation algorithm described in the next section, has each PE inject a separate cumulative packet for each array element. *Cumulative messages* can use keys to compartmentalize different calculations in one message wave. An example of this is illustrated in the algorithm for the FP rotate left function, *rotl*, also described in Section 5. In that case, the keys are essential for enabling PEs to align their contributions with those of other appropriate PEs.

3.6 Implementation of the Message ALU

With the detailed behavior of the network derived above, the message ALU can now be summarized. Three identical copies of this ALU (together with two T cell filters) form a *message processor*, which is the principal part of the T cell. (Besides the message processor, a T cell contains configuration switches and their control hardware which operate during *partitioning* and possibly some further arithmetic hardware which operates during *storage management*)[Magó&Stanat]. Specific restrictions on the inputs of the two downward ALUs (for example, that one input [the one from a filter] has only cumulative messages on it) are not exploited.

Each ALU consists of a simple arithmetic unit, a switch that selects its inputs, the Loser register, and some internal state registers. The internal state consists of three parts, totaling 5 bits: A *carry* bit for multi-word addition, two *min-state* bits for multi-word comparisons, and two *loser* bits to indicate which side, if any, has a packet currently residing in the Loser register. That last element of the state controls the input switch to the arithmetic unit; the first two affect the addition and minimum operations when the arithmetic unit is actually combining packets, rather than just sorting them into a result and a loser.

As mentioned earlier, each packet contains a *header* and a *value*. The *header* is divided into a 4-bit type field followed by either a 4-bit *key number* or a 4-bit *opcode*. The first type

bit distinguishes *simple* packets from *cumulative* ones. The second type bit distinguishes *right-to-left* packets (parallel suffix) from *left-to-right* ones (parallel prefix). The third type bit distinguishes *end* packets from "useful" ones (called *body* packets for no good reason). The fourth and final type bit distinguishes *key* packets from *value* ones.

Packets with different types never combine: they are always sorted. *End* packets always follow (that is lose to) other 'similar' packets, thus, the *end* type bit is 1 for *end* packets and 0 for *body* packets. When one *key* packet loses to another, it moves to the Loser register to try again. From there it should meet and lose to the *value* packets belonging to the winning key; thus, the *key* type bit is 1 for *key* packets and 0 for *value* packets. (Since it might also need to lose to additional keys if they are being used, the primary key uses 15 for its *key number*, the secondary key uses 14 and so on. For unrelated and somewhat less compelling reasons, *cumulative* packets precede *simple* packets and left-to-right packets precede right-to-left ones.

End packets contain an opcode which can be used, for example, to vote whether some loop has completed. It is important to note that *cumulative end* packets become *simple end* packets within the T cell filters* and so are constrained to use opcodes and values consistent with the *simple end* packets being sent in that particular message wave.

The following packet types are possible:

0000	Cumulative Left value
0001	Cumulative Left key
0010	End Cumulative Left
0100	Cumulative Right value
0101	Cumulative Right key
0110	End Cumulative Right
1100	Simple value
1101	Simple key
1110	End Simple
—11, 10—	unused.

The arithmetic unit generates two outputs from each pair of packets it receives. It always feeds to the Loser Register the maximum of the packets, viewed just as sequences of bits. From the state information it generates, the subsequent operation will decide whether to recycle the Loser register's contents via the input switch.

* This translation imposes a constraint on the hardware that does not apply to this simulation system: the hardware is intended to allow *bit-pipelining*, that is, message processing can be done bit-serially if that is cost-effective. This translation requires that a filter receive three bits of the packet type in order to detect that the *simple* packet is also an *end* packet and so should be altered instead of being discarded.

The result fed into the network from the arithmetic unit is calculated as follows. If the packets have different types, or if they are keys then ordering will occur. (Given the former condition, the latter can be replaced by "or if one is a key" with either key being picked arbitrarily). In this case, the result fed into the network is the minimum of the two packets (viewed as bit strings) and the Loser register is marked as empty exactly when the two packets are identical, which can happen with *key* packets.

When the packet types match and are not keys, the Loser register will be marked as empty and the packets will combine to feed a result into the network. The opcode (in fact the entire header field) in the result packet is the minimum of the two headers being combined. As noted above, the opcode *being applied* may differ from this. If the packets are right-to-left packets, then the left opcode is applied (because the right opcode might have been overwritten by a parallel suffix *group bit* which proceeds but should not be performed in this ALU); otherwise, the right opcode is applied.

The following opcodes are implemented in this system:

0000	"??(0)"	Unused (to enable broader error detection in system).
0001	"(Order)"	Notes ordering inside ALU - should never appear in actual packets.
0010	"2ndC"	Group bit for cumulative (left-to-right) packets.*
0011	"1stC"	Group bit for cumulative (right-to-left) packets.*
0100	"Min"	Minimum (combining, not sorting).
0101	"MinC"	Minimum continued (min state not initialized).
0110	"2nd"	Take second value (for right shifts).
0111	"1st"	Take first value (for left shifts).
1000	"+"	Integer addition (carry cleared initially).
1001	"+C"	Integer addition continued (carry not cleared).
1010	"And"	Bitwise And.
1011	"Xor"	Bitwise Exclusive Or.
1100	"??(c)"	Unused.
1101	"??(d)"	Unused.
1110	"??(e)"	Unused.
1111	"??(f)"	Unused.

Figures 1 and 2 shows the C function, *malu_step()*, which defines the behavior of the arithmetic unit in processing a single pair of packets. While the software system being described matches the intended hardware *behavior*, it does not follow likely hardware *implementation*. Though this might pose a challenge should the system be connected to actual hardware (for testing, for example), it allows significant gains in speed and clarity.

* Since there are opcodes to spare, a separate group bit opcode has been created from the 2nd and 1st below. As far as current algorithms have gone, those are redundant.

```
malu_step (Ltype, Lhead, Lval, Rtype, Rhead, Rval, Out, Loser, State)
unsigned char           Ltype, Lhead, Rtype, Rhead ;
int                     Lval, Rval ;
struct  PacketType      *Out, *Loser ;
unsigned char           *State ;

{
unsigned char   opcode ;
unsigned char   loserstate, minstate, carrystate ; /* Local State subfields:  */

loserstate =  *State  & Sloser ;
minstate   =  *State  & Smin ;
carrystate =  *State  & Scarry ;

if (debug>20)
    fprintf(stderr,"\nState:  rec-%02X,(%02X:%02X:%02X) %d %d %d %d\n",
        *State, loserstate, minstate, carrystate,
        Ltype, Lhead, Rtype, Rhead) ;

/* this is the operation to be done in the ALU, vs that sent in result.  */
opcode = ((Ltype!=Rtype) || ((Ltype&HVK)==HK))      ?  Oo
         :    KeyMask & ( ((Ltype&HLR)==HR)  ?  Lhead :  Rhead ) ;

loserstate = ((opcode != Oo) || ((Lhead == Rhead) && (Lval == Rval)))
        ?  SNoLoser
        :              (  ((Lhead < Rhead) || ((Lhead == Rhead) && (Lval < Rval)) )
                ?  SRLost :  SLLost ) ;

if (debug>20) {
    fprintf(stderr, "logic parts %d (= %d || %d && %d)    %d (= %d || %d && %d)\n",
        ((opcode != Oo) || ((Lhead == Rhead) && (Lval == Rval))),
         (opcode != Oo),    (Lhead == Rhead),   (Lval == Rval),
        ((Lhead < Rhead) || ((Lhead == Rhead) && (Lval < Rval)) ),
         (Lhead < Rhead),    (Lhead == Rhead),   (Lval < Rval) ) ;
    fprintf(stderr,"Opcode %s, new loserstate %02X.\n", OpSyms[opcode], loserstate) ;
    }

/* Now set up contents of loser register */
/* Just like probable hardware, loser gets value even if empty */
/* Easiest to set up Out assuming Order now, and overwrite as needed later.  */
/* If combining, will send min opcode, ie effectively do OR of group bits */

if ( (Lhead<Rhead) || ((Lhead==Rhead) && (Lval<Rval)) )  {
    Loser -> header = Rhead ; Loser -> value  = Rval ;
    Out   -> header = Lhead ; Out   -> value  = Lval ;
    }
else    {
    Loser -> header = Lhead ; Loser -> value  = Lval ;
    Out   -> header = Rhead ; Out   -> value  = Rval ;
    }

if (debug>20) {
    PrintPacket(stderr, *Loser," -loser, ") ;
    PrintPacket(stderr, *Out," -min out,") ;
    }
```

Figure 1. The behavior of the message processor ALU (part I).

In particular, where the hardware ALUs are intended to behave in a pipelined fashion down to the bit level, each simulated ALU performs all its processing at one time, without any overlap. This requires that message buffers be inserted at each interface. The arguments to *malu_step()* consist of the left and right input packets (broken into their fields for

```
/* Now initialise alu state for various opcodes */
switch (opcode) {
    case Om :      minstate = SEq ;
                   break ;
    case O1 :
    case O1c :     minstate = SLess ;
                   break ;
    case O2 :
    case O2c :     minstate = SGtr ;
                   break ;
    case Omc :     /* initial value of received state will be used */
                   break ;
    case Op :      carrystate = SNoCarry ;
                   break ;
    case Opc :     break ;
    case Oa :      break ;
    case Ox :      break ;
    case Oo :      break ;
    default :      fprintf(stderr, "(%d) used bad opcode [%02X]\n", II, opcode) ;
                   break ;
        }

/* Now do various opcodes */

switch (opcode) {
    case O1 :
    case O1c :
    case O2 :
    case O2c :
    case Om :
    case Omc :     if (minstate==SEq) { /* calculate new state */
                     if (Lval<Rval) minstate = SLess ;
                     if (Lval>Rval) minstate = SGtr ;
                     }
                   /* calculate new value */
                   Out -> value  =  (minstate==SLess)  ?    Lval  :    Rval ;
                   break ; /* end of processing min family */
    case Opc :
    case Op :      Out -> value  =  (Lval & ValueMask) + (Rval & ValueMask) +
                                 ((( carrystate & Scarry) == SIsCarry) ?  1 :  0) ;
                   carrystate = ((Out->value) & PacketCarry)
                                 ?  SIsCarry :  SNoCarry ;
                   Out -> value &= ValueMask ;
                   break ;
    case Oa :      /* no state change */
                   Out -> value  =  Lval & Rval ;
                   break ;
    case Ox :      Out -> value  =  Lval ^ Rval ;
                   break ;
    case Oo :      /* This is initialised when loser is set up */
                   break ;
    default :      break ;
    }

*State = minstate + loserstate + carrystate ;

if (debug>20) {
    fprintf(stderr, "\nStates:  final-%02X, loser-%02X, min-%02X, carry-%02X\n",
                   *State, loserstate, minstate, carrystate) ;
    PrintPacket(stderr, Out," -final out.\n") ;
    }
}
```

Figure 2. The behavior of the message processor ALU (part II).

convenience), the result packet, the new contents of the Loser register, and the state

reflecting conditions sensed by the ALU during the operation.

First, the code determines from the headers whether an ordering or combining action will occur and saves this information in an internal opcode variable. Second, it determines whether the Loser register will be empty. Third, the Loser and Out values are calculated under the assumption that ordering will occur. (Unlike hardware, the code has no difficulty altering these values when this assumption is incorrect, and this approach is more concise). Fourth, the initial state of the ALU may be set to prepare for applying an opcode. Finally, the opcode is applied to the pair of input values. The calculation creates state which is returned in the *State* argument for the subsequent call. These steps are interleaved with trace output statements that may be performed depending on the depth of debugging in effect.

```
Filter(In, Out, side)
struct PacketType *In, *Out ;
char    side ;
{
char No_EndPkt = 1 ;
while ( BadOrESPacket(*In, (char) 0) != EndStream ) {
    if (((In->header)&(HCS+HLR)) == (HC+side)) {
        Out -> header = In -> header ;
        Out -> value = In -> value ;
        if (((Out->header) & HBE) == HE) {
            /* Now change EC{L/R} to ES for relevant down malu.  */
            Out->header |= (HCS+HLR) ; /* ES is ESRV by convention */
            /* ?  Should this step clear K for safety ?  */
            /*  No, should already be clear in well formed packets */
            No_EndPkt = 0 ;
            }
        Out++ ;
        }
    In++ ;
    }
if (No_EndPkt) {
    fprintf(stderr, "\t\tHad to insert End packet in %s filter.\n",
                (side==HR)?"right":"left") ;
    Out   -> header = HS + HR ;
    Out++ -> value  = 666 ;
    }
}
```

Figure 3. The behavior of a filter in the message processor.

Figure 3 shows the C function that defines the behavior of a filter. The procedure takes a list of packets and a note indicating whether this filter transmits left-to-right or right-to-left packets, and returns the list of packets matching that type. In addition the appropriate *cumulative end* packet is transformed into a *simple end* packet.

```
malu (LeftList, RightList, OutList, Loser, State)
struct  PacketType      *LeftList, *RightList, *OutList, *Loser ;
unsigned char           *State ;
{
struct PacketType       *Out = OutList ;

do  {
    struct      PacketType      *Lp, *Rp ;
    if(debug>10) {
        fprintf(stderr,"%c%c ",
                ( (*State&Sloser) == SLLost) ?  'B' : 'L',
                ( (*State&Sloser) == SRLost) ?  'B' : 'R') ;
        }
    Lp = ( (*State&Sloser) == SLLost) ?  Loser :  LeftList++ ;
    Rp = ( (*State&Sloser) == SRLost) ?  Loser :  RightList++ ;
    if(debug>10) {
        PrintPacket(stderr, *Lp," (*) ") ;
        PrintPacket(stderr, *Rp," ==> ") ;
        }
    malu_step( (unsigned char)  (Lp->header & TypeMask),
                                Lp->header,
                                Lp->value,
               (unsigned char) (Rp->header & TypeMask),
                                Rp->header,
                                Rp->value,
                                Out,   Loser,   State) ;
    if(debug>10) {
        PrintPacket(stderr, *Out,".") ;
        fprintf(stderr,"%s[%02X].\n",
                ( BadOrESPacket(*Out, (char) 0)!= EndStream)
                 ?  "rpt" :  "end",
                *State ) ;
        }
    }
while (BadOrESPacket(*(Out++), (char) 0)  !=  EndStream) ; /* end do */

if (debug>10) {
    fprintf(stderr,
            "\nThe last two packets processed from the input lists are:\n") ;
    PrintPacket(stderr, *--LeftList, "") ;
    PrintPacket(stderr, *--RightList, "") ;
    }
}
```

Figure 4. The behavior of the ALU in the message processor.

Figure 4 shows the C function that defines the behavior of the message processor in performing a complete message wave. It consists of a loop that repeatedly calls *malu_step()* until a *simple end* packet is generated. It takes as arguments two lists of input packets, which it will merge and combine into an output list. In addition, it contains a Loser register and a State variable. Before combining two packets, the loop body must first use the state information to determine which of the three inputs (the Loser register and the two streams of packets entering the message ALU from elsewhere) will provide the two inputs to *malu_step()*, which returns a result packet to proceed into the network, new state information, and possibly a packet in the Loser register.

4 The structure of the simulator

The previous sections described the abstract network; this section describes the actual simulator which models the behavior of an area network on a single wave of messages. Algorithms usually involve multiple message waves; this is accomplished by constructing a pipeline of individual programs. The first two programs usually generate the auxiliary representation; these are the first examples given in the next section. (A couple of simple rules are provided later that help the output from a program be both convenient for subsequent programs to read, and clear for human consumption).

The simulation of any particular message wave consists of four parts, a file of constants (".constants.h"), a file of support routines (".subrs.c") which includes the code for the ALU shown in the previous section, the procedure "main" which animates the system (in ".skeleton.h"), and the primary file which contains the routines that specify how each node in the network behaves, and incorporates the other three files at appropriate points.

Algorithm designers can ignore all but the primary file, and in fact can further focus their attention on the three parts that deal with the leaf processors. Those three parts specify how the message system is going to be used in a particular algorithm, while the remaining parts simply define how message processing occurs in general, and so remain unchanged. Properly, these other parts may well belong, from a designer's point of view, in a different file.

The primary file begins by declaring a structure, *struct LcellType*, specifying the local variables used by a leaf processor during the algorithm. Following that, the primary file declares *struct TcellType* to contain the local variables for the internal nodes, and then it includes ".skeleton.h".

The file ".skeleton.h" starts by declaring arrays of these two structures, together with naming macros (like "Self", "LeftChild" and "RightChild") which construct a tree within these arrays. The naming convention defines the children of the tree node residing in the i'th array element as residing in array elements $(2i)$ and $(2i+1)$. The root resides in element (1); element (0) is unused. The tree is perfectly balanced with a power of 2 number of elements. This is not essential, but otherwise, connecting leaf processors into the network is more complicated* and no advantage is gained.

* Imagine a tree with two internal nodes, numbered 1 and 2. Internal node (2) is the left child of internal node (1). Leaf processors (0) and (1) are the children of internal node (2) while leaf processor (2) is the right child of internal node (1).

The naming macros use a global variable, II, as a reference point to identify a node; from there it is used to calculate subscripts for related nodes. The procedure "main" (in ".skeleton.h") iterates through this variable to focus activity on each node of the tree in turn, first in an upsweep then in a downsweep. Within these loops, the functions specifying cell behaviors are called appropriately. Thus, these routines should only access local cell data via the naming macros. Under this scheme, each function looks like an entirely self-contained and active part of a completely distributed algorithm.

These functions, which complete the primary file, are:

InitL ()	Initialize the leaf node, reading its local data from input.
InitT ()	Initialize the state of an internal node.
Lup ()	To process upward packets in a cell having leaf children.
Tup ()	To process upward packets in a cell having internal children.
Troot ()	To process messages in the root of an area.
Tdown ()	To process downward packets in a cell with internal children.
Ldown ()	To process downward packets in a cell having leaf children.
PrintT ()	Print any desired final information from an internal node.
PrintL ()	Print the results a processor acquired during the message wave.

Figure 5 shows the code for *Troot()*. It defines both the upward and downward processing of packets in the root of an area machine. It is the concatenation of the two procedures, *Tup()* and *Tdown()*, with the addition that it reports the number of packets sent out since that is the primary factor determining the speed of a communication wave.

Only the routines *InitL()* and *PrintL()* actually concern the algorithm designer. The others simply perform the message processing, so they are left out of the following examples. These two routines show some recurring patterns throughout various algorithms they implement.

InitL() starts by reading the data for its L cell from input. Usually (at least when the auxiliary representation has been generated), each L cell's data occupies its own line of input. To simplify parsing input, any potentially blank string field has "H<backspace>" appended. This disappears when viewing the output on a screen and enables the standard library routines to read the string. After some local processing, *InitL()* prepares the list of packets for the L cell to send into the network. It is important that empty L cells contribute exactly the three *end* packets or else the network will hang. By convention, the opcode in the *simple end* packet is *and* for those cases when the PEs need to vote on

```
struct  TcellType {
        struct PacketType       Up[ListLength],
                                Down[ListLength],
                                FilterLeft[ListLength],
                                FilterRight[ListLength],
                                LoserUp,
                                LoserLeft,
                                LoserRight;
        unsigned char           StateUp, StateLeft, StateRight ;
        } ;

void    Troot() {
malu (LeftChildT.Up, RightChildT.Up, Self.Up, &Self.LoserUp, &Self.StateUp) ;
fprintf(stderr,"Root malu passed %1d packets.\n", StreamLength(Self.Up)) ;
Filter (LeftChildT.Up, Self.FilterLeft, HL) ;
Filter (RightChildT.Up, Self.FilterRight, HR) ;

malu (Self.Up, /* skip copying to the Down list */
        Self.FilterLeft, RightChildT.Down, &Self.LoserRight, &Self.StateRight) ;
malu (Self.FilterRight, Self.Up, /* skip copying to the Down list */
        LeftChildT.Down, &Self.LoserLeft, &Self.StateLeft) ;
if (debug>5) {
    char label[40] ;
    sprintf(label,"\t\t{malu(%2d) (root) sent up, left, right:}\n",II) ;
    PrintList(stdout, label, Self.Up) ;
    PrintList(stdout, "", LeftChildT.Down) ;
    PrintList(stdout, "", RightChildT.Down) ;
    }
}
```

Figure 5. Processing in the root of an area network.

whether they have completed some iterative process. Consequently, the two *cumulative end* packets must also use that opcode.

PrintL() starts by processing the incoming packet list. For the corresponding hardware to operate fast enough, restrictions are placed on how much PrintL() can do at this point. In the current system, the PE is allowed to count incoming messages and save a short contiguous set from within the arriving stream. It turns out that although interrupts are supposedly transparent to the system, one of their consequences is that *key* packets may occur multiple times in the arriving stream. Consequently, this system counts non-*key* packets and on reaching some locally calculated value, notes the current position in the list of incoming packets. Following this, *PrintL()* is free to use this packet and a small number of subsequent ones to generate new values in the PE. This approach is best illustrated in the *transpose* example given in the next section. Finally, *PrintL()* prints its local data, following the appropriate rules described above.

Once compiled, the principal program is linked with the support routines residing in the file ".subrs.c". They include ones to read and write packets, both individually and in streams, and the filter and message processor functions that were presented in the previous section.

Finally, it should be noted that the example algorithms described in the next section are deliberately not robust against illegal input (for example, FFP expressions that are ill-formed), since their purpose is clarity as demonstrations and not reliability as production code.

5 Example communication algorithms

Several examples are now presented to show not only how this system simulates the message network, but also to show how the message network implements certain interesting operations for the FFP Machine.

5.1 Generating the auxiliary representation for FFP expressions

In Backus's Functional Programming languages, data are constructed hierarchically from atoms (such as integers), using sequence constructors. An FFP object can be written as an expression tree in which each atom corresponds to a leaf node in the tree and each sequence corresponds to an internal node whose children are the expression trees corresponding to the sequence elements. (There is no connection between the expression tree and the tree structure of either the area machine or the overall physical network; indeed it is not even binary).

In order to carry out its actions, the FFP Machine augments the FFP objects with an *auxiliary representation* which describes their structure in a way suited to the self-addressing style of computation. The *auxiliary representation* has five components. The *index* provides a unique identifier for each PE in an area machine, by numbering the symbols from left to right. The *relative level number* is the depth of a symbol within the expression tree. The *directory* is the historical term for the component of the auxiliary representation that describes the position of a symbol within the expression tree. Each element of the directory specifies a particular child of the sequence element reached by the preceding directory elements. The directory is restricted to length four, that being the most detail that has been needed for the FFP Machine to implement the functions considered to date. (This arises in matrix multiplication: the first element distinguishes the function from the operand, the pair of matrices to be multiplied, which are distinguished by the second element. The third and fourth entries then label rows and columns). The final two components of the auxiliary representation are a pair of boolean arrays, *firstL* and *lastL*, which are derived from the *relative level number* and the sequence bracket symbols in the FFP expression. They mark the first and last PEs holding an FFP object at a given depth. Together with directory elements, they provide a simple way to mark locations

where actions must be performed and are used often enough to be worth creating at the outset.

The algorithms for generating this auxiliary representation are subject to the rules for how FFP expressions may actually be represented in the L array. Empty PEs may be interleaved with others in the area machine: this complicates the auxiliary representation algorithm because the same values should be generated for the symbols irrespective of where empty PEs are located. There are also several alternatives regarding the amount of an FFP expression that each PE can hold. The various algorithms described here assume that such a leaf processor may hold an arbitrary number of left sequence brackets, $<^n$, an atom or function parenthesis, and an arbitrary number of right sequence brackets, $>^n$; however, the algorithm should work basically unchanged, even with significant variation of the chosen representation.

A leaf processor (that is, PE) may hold several symbols; the auxiliary representation is calculated with respect to the main symbol in the PE, not the sequence brackets. The *index* is calculated by counting non-empty PEs, using a parallel prefix sum. The *relative level number* is calculated by counting the number of (unmatched) opening sequence brackets to the left of the main symbol in the PE. That means, each PE contributes to the parallel prefix sum calculation, the difference between the opening and closing brackets that it holds. Since the *relative level number* relates to the main symbol in a PE, each PE must add the number of left sequence brackets that it itself holds, to the result it gets from the parallel prefix sum. The algorithm "addr1", shown in Figure 6, calculates these two components of the *auxiliary representation*.

The value *directory[i]* counts the number of complete objects having a *relative level number* of (i) within the *same* level $(i-1)$ object. Such complete objects are identified with atoms or closing sequence brackets having *relative level number* i; thus, the directory calculation is a separate message operation because it cannot begin until the PE already has its *relative level number*. An object at level i should also prevent any count of deeper objects $(level > i)$ from passing across it; this is achieved by injecting a group bit (together with zero as the addition unit value to begin the subsequent count) into calculations carried out at deeper levels. The calculation of the *directory*, together with those of *firstL* and *lastL*, since they also depend on *relative level number*, are shown in Figure 7.

```
/* Calculate RLN and Index using cumulative message waves */
struct  LcellType {
        int                     lb, rb, rln, index, dir[4] ;
        char                    symbol[80], empty, firstL[4], lastL[4] ;
        struct PacketType       Up[ListLength], Down[ListLength];
        } ;

void    InitL() {
int msg_count ;
char line[80] ;

/* read local L cell data from input */

/* Send packets */
if (II == Size-1) { /* last cell in area - create identities for CL */
    L.Up[msg_count].header  = HC + HL + O2c ; /* RLN */
    L.Up[msg_count].value = 0 ;
    }
else {
    L.Up[msg_count].header  = HC + HL + Op ; /* RLN */
    L.Up[msg_count].value = L.lb - L.rb ;
    }
msg_count++ ;
if (II == Size-1) { /* last cell in area - create identities for CL */
    L.Up[msg_count].header  = HC + HL + O2c ; /* Index */
    L.Up[msg_count].value = 0 ;
    }
else { /* Index:  unique numbering over non empty cells, and o.w.  repeatable */
    L.Up[msg_count].header  = HC + HL + Op ;
    L.Up[msg_count].value = (!L.empty) ?  1 :  0 ;
    }
msg_count++ ;

L.Up[msg_count].header  = HE + HC + HL + Oa ; /* ecla */
L.Up[msg_count].value = 1 ;
msg_count++ ;
L.Up[msg_count].header  = HE + HC + HR + Oa ; /* ecra */
L.Up[msg_count].value = 1 ;
msg_count++ ;
L.Up[msg_count].header  = HE + HS + HR + Oa ; /* ea - ESRV by convention */
L.Up[msg_count].value = 1 ;
msg_count++ ;
}

void    PrintL() {
int msg =0, pkt_count =0, save ;

for ( ; ; ) { /* RECEIVE PACKETS */
    if ( BadOrESPacket(L.Down[msg], (char) 0) == EndStream ) {
        fprintf(stderr,"Filter in %1d hit ES too soon.\n", II) ;
        break ;
        }
    if (pkt_count == 0 ) { save = msg ; break ; }
        /* Keys repeat after interrupts --> only count non keys.  */
    if ((L.Down[msg].header&HVK) != HK ) pkt_count++ ;
    msg++ ; /* deal with next packet.  */
    }
L.rln    = L.Down[save+0].value + L.lb ; /* USE RETAINED PACKETS */
L.index  = L.Down[save+1].value ;

/* Print out L cell contents here */
}
```

Figure 6. Calculating Index and Relative Level Number.

```
/* Create directory and firstL/LastL using RLN */

struct  LcellType {
        int                     lb, rb, rln, index, dir[4] ;
        char                    symbol[80], empty, firstL[4], lastL[4] ;
        struct PacketType       Up[ListLength],
                                Down[ListLength];
        } ;

void    InitL() {
char brackets[80], c ;
int msg_count, i ;

msg_count = 0 ;

/* read local L cell data from input */

/* Send packets */
    { int object ;
    for (msg_count=0 ; msg_count<4 ; msg_count++) {
        object = L.rln-L.rb ;
        L.Up[msg_count].header = ((II == Size-1)  ||  (msg_count > object))
                    ?    HC+HL+O2c  :    HC+HL+Op ;
        L.Up[msg_count].value = ((II==Size-1) || (msg_count!=object)
                               || (L.rb==0&&L.symbol[0]=='\0'))
                               /* Ignore '<' when rest is empty */
                    ?    0  :    1 ;
        }

    L.Up[msg_count].header  = HE + HC + HL + Oa ; /* ecla */
    L.Up[msg_count].value   = 1 ;
    msg_count++ ;
    L.Up[msg_count].header  = HE + HC + HR + Oa ; /* ecra */
    L.Up[msg_count].value   = 1 ;
    msg_count++ ;
    L.Up[msg_count].header  = HE + HS + HR + Oa ; /* ea - ESRV by convention */
    L.Up[msg_count].value   = 1 ;
    msg_count++ ;
    }
}

void    PrintL() {
int msg =0, pkt_count =0, save, i ;

for ( ; ; ) {
    if ( BadOrESPacket(L.Down[msg], (char) 0) == EndStream ) {
        fprintf(stderr,"Filter in %1d hit ES too soon.\n", II) ;
        break ;
        }
    if (pkt_count == 0 ) { save = msg ; break ; }
    if ((L.Down[msg].header&HVK) != HK ) pkt_count++ ;
    msg++ ; /* deal with next packet.  */
    }

for (i=0 ; i<4 ; i++) {
    L.dir[i]    = L.Down[save+i].value + (i<=L.rln);
    L.firstL[i] = (i >= L.rln-L.lb) && (i < L.rln+(L.symbol[0]!='\0')) ;
    L.lastL[i]  = (i >= L.rln-L.rb) && (i < L.rln+(L.symbol[0]!='\0')) ;
    }

/* Print out L cell contents here */
}
```

Figure 7. Calculating directories, firstL and lastL.

Each PE sends four parallel prefix sum packets into the tree network. For the *highest* level object that it finishes,* it sends a value of 1 and for *all* others it sends 0. Down to the level of that object, it sends the addition opcode with zero; below that level it sends a group bit with the addition unit of zero to separate groups and to provide the identity for the next group's cumulative sum (that is, counting within the sequence to the right of this one).

5.2 Compacting brackets

Given the way FFP symbols may be allocated to PEs, a series of left brackets followed by an atom might be either stored in a single PE or spread over many. *Compaction*, shown in Figure 8, is the process of causing FFP expressions to occupy a minimal number of PEs. The method described here moves brackets towards the appropriate nearby atom (or towards each other in the case of a deeply embedded empty sequence). Using prefix sums over left brackets, with group bits supplied where atoms or right brackets occur, the value received by a PE with an atom is the brackets immediately to its left, that can be moved to it. The PE increments its left bracket register by that amount, while the other registers in the group clear their brackets. In a similar fashion, right brackets can be replicated towards the left with a suffix sum. Care must be taken that left and right brackets moving towards each other do not pass.

5.3 Transposing arbitrary FFP matrices

In contrast to the previous algorithms which used cumulative operations, *transpose*, shown in Figure 9, uses simple messages with sorting. PEs send the contents of their three symbol registers, preceded by a pair of keys. Simply, the primary key is the column number and the secondary key is the row number, both of which are entries in the *directory*. This however fails to transpose matrices whose entries are more complicated than just a single atom. Instead, the *index* is used as the secondary key. This provides the same effect, but also transfers multi-symbol objects together.

To provide the correct structure around the reordered elements, the initial sequence brackets delimiting the rows are deleted and new pairs of sequence brackets are created delimiting the final rows (what were initially the columns). For this, the number of rows must be counted and broadcast beforehand to enable each object in the last row to create

* For example, a PE containing "1>" holds the end of both the atomic object, 1, and the sequence of which it is the last element. The sequence is the higher level object, so the *relative level number* of the ">" (which is $rln - 1$) is the value used.

```
/* Compact brackets together, assuming <* P >* symbol registers */
struct  LcellType {
        int                     lb, rb, rln, index, dir[4] ;
        char                    symbol[80], empty, firstL[4], lastL[4] ;
        struct PacketType       Up[ListLength], Down[ListLength];
        } ;

void    InitL() {
char c, brackets[80] ;
int msg_count =0, i ;

/* read local L cell data from input */

/* Send packets */
if (L.symbol[0]!='\0' || L.rb>0) { /* group bit separating '<' sequences */
    L.Up[msg_count].header  = HC + HL + O2c ;
    L.Up[msg_count].value   = 0 ;
    }
else {
    L.Up[msg_count].header  = HC + HL + Op ; /* combine brackets */
    L.Up[msg_count].value   = L.lb ;
    }
msg_count++ ;
L.Up[msg_count].header  = HE + HC + HL + Oa ; /* ecla */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
if (L.symbol[0]!='\0' || L.lb>0) { /* group bit separating '>' sequences */
    L.Up[msg_count].header  = HC + HR + O1c ;
    L.Up[msg_count].value   = 0 ;
    }
else {
    L.Up[msg_count].header  = HC + HR + Op ; /* combine brackets */
    L.Up[msg_count].value   = L.rb ;
    }
msg_count++ ;
L.Up[msg_count].header  = HE + HC + HR + Oa ; /* ecra */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
L.Up[msg_count].header  = HE + HS + HR + Oa ; /* ea - ESRV by convention */
L.Up[msg_count].value   = 1 ;
}

void    PrintL() {
int msg =0, pkt_count =0, save ;

for ( ; ; ) {
    if (pkt_count == 0 ) { save = msg ; break ; }
        /* Keys repeat after interrupts --> only count non keys.  */
    if ((L.Down[msg].header&HVK) != HK ) pkt_count++ ;
    msg++ ; /* deal with next packet.  */
    }

if (L.symbol[0]!='\0' || L.rb>0)
    L.lb += L.Down[save+0].value ;
else
    L.lb  = 0 ;
if (L.symbol[0]!='\0' || L.lb>0)
    L.rb += L.Down[save+2].value ;
else
    L.rb  = 0 ;

/* Print out L cell contents here */
}
```

Figure 8. Compacting brackets.

```
/* Transpose a general matrix by sorting.  Part 1 gave length to fix brackets.  */
struct  LcellType {
        int                     lb, rb, rln, index, dir[4] ;
        char                    symbol[80], empty, firstL[4], lastL[4] ;
        struct PacketType       Up[ListLength], Down[ListLength];
        int                     length ;
        } ;

void    InitL() {
char c, brackets[80] ;
int msg_count =0, i ;
int newlb, newrb ;

/* read local L cell data from input */

newlb = L.lb ;
newrb = L.rb ;
if (L.firstL[1])                        newlb-- ; /* Delete '<' of each row.    */
if (L.lastL[0])                         newrb-- ; /* Must compact final '>'.    */
if (L.lastL[1]  && L.dir[1]<L.length)   newrb-- ; /* Delete '>' of each row.    */
if (L.firstL[2] && L.dir[1]==1)         newlb++ ; /* Create '<' to start col.  */
if (L.lastL[2]  && L.dir[1]==L.length)  newrb++ ; /* Create '>' to end col.     */

/* Send packets */
L.Up[msg_count].header  = HE + HC + HL + 0a ; /* ecla */
L.Up[msg_count].value   = 1 ; msg_count++ ;
L.Up[msg_count].header  = HE + HC + HR + 0a ; /* ecra */
L.Up[msg_count].value   = 1 ; msg_count++ ;
if (!L.empty) {
    L.Up[msg_count].header  = HS + HR + HK + 15 ; /* key k0 */
    L.Up[msg_count].value   = L.dir[2] ; msg_count++ ;
    L.Up[msg_count].header  = HS + HR + HK + 14 ; /* key k1 */
    L.Up[msg_count].value   = L.index ; msg_count++ ;
    L.Up[msg_count].header  = HS + HR + 01 ;
    L.Up[msg_count].value   = (L.symbol[0]<<8) + L.symbol[1] ; msg_count++ ;
    L.Up[msg_count].header  = HS + HR + 01 ;
    L.Up[msg_count].value   = newlb ; msg_count++ ;
    L.Up[msg_count].header  = HS + HR + 01 ;
    L.Up[msg_count].value   = newrb ; msg_count++ ;
    }
L.Up[msg_count].header  = HE + HS + HR + 0a ; /* ea - ESRV by convention */
L.Up[msg_count].value   = 1 ;
}

void    PrintL() {
int msg =0, pkt_count =0, save ;

if (!L.empty) {
    for ( ; ; ) {
        if (pkt_count > 3*L.index+2 ) { save = msg ; break ; }
        if ((L.Down[msg].header&HVK) != HK ) pkt_count++ ;
        msg++ ;
        }

    /* non-empty cell catches next triple of symbol stuff */
    L.symbol[0] = L.Down[save+0].value >> 8 ;
    L.symbol[1] = L.Down[save+0].value & 0x00ff ;
    L.lb        = L.Down[save+1].value ;
    L.rb        = L.Down[save+2].value ;
    }

/* Print out L cell contents here */
}
```

Figure 9. Transpose an arbitrary matrix.

a closing bracket. This length is trivially calculated in a prior message wave which is not shown.

5.4 Rotate left (by one position) within groups

This algorithm performs rotations within individual subsequences of the entire FFP expression. Specifically, it rotates a single data value within the PEs of each subsequence, and can be viewed as individually rotating each row in a matrix (the algorithm of the next section could rotate matrix columns vertically, which can be viewed as rotating the rows upwards, with the first row becoming the last). Ironically, this algorithm doesn't really use *group bits*, although the opcode for rotation does in fact perform the same way.

During the first communication wave, shown in Figure 10, all values are shifted as a single rotation. This leaves the values that were at the start of each subsequence incorrectly placed at the end of the preceding subsequence rather than at the end of their own. A second communication wave, shown in Figure 11, rotates just those values to the right, over their entire subsequence, to the tail of their own subsequence. This is their proper position.

5.5 Rotate left by k positions

This rotation algorithm differs from the one above in that it allows rotations by an arbitrary distance. This could be used for the *rotate-left* function in the FP language, which must handle shifting an entire FFP object consisting of an arbitrary number of symbols to the other end of a sequence of such objects. It can also be used in algorithms that simulate a mesh or torus on the physically tree-structured FFP Machine, to avoid the full communication cost of the "north-south" communication phase [Middleton&Smith].

In contrast to the previous rotation algorithm in which only a single message passes through the root of the area machine network, this algorithm passes roughly as many messages as the distance of the shift, that is, the number of symbols in the object being shifted. This procedure works by incorporating sorting into the cumulative rotation scheme used above. The *keys* allow several overlapping rotations to occur in the same message wave, without interfering with each other.

To illustrate the operation, consider rotating a sequence of 10 letters left by 4 places, as shown in the top part of Figure 12, using upper case to denote the initial position, and lower case to denote the final position.

```
/* Left rotation in groups by one position - Part 1:  rotate entire sequence */

struct  LcellType {
        int                     lb, rb, rln, index, dir[4] ;
        char                    symbol[80], empty, firstL[4], lastL[4] ;
        struct PacketType       Up[ListLength],
                                Down[ListLength];
        } ;

void    InitL() {
char c, brackets[80] ;
int msg_count =0, i ;

/* read local L cell data from input */

/* Send packets */
L.Up[msg_count].header  = HE + HC + HL + 0a ; /* ecla */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
if (!L.empty) {
    L.Up[msg_count].header  = HC + HR + 01 ; /* par suffix rotate left opcode */
    L.Up[msg_count].value   = (L.symbol[0]<<8) + L.symbol[1] ;
    msg_count++ ;
    }
L.Up[msg_count].header  = HE + HC + HR + 0a ; /* ecra */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
L.Up[msg_count].header  = HE + HS + HR + 0a ; /* ea - ESRV by convention */
L.Up[msg_count].value   = 1 ;
}

void    PrintL() {
int msg =0, pkt_count =0, save ;

if (!L.empty) {
    for ( ; ; ) { /* RECEIVE MESSAGES */
        if ( BadOrESPacket(L.Down[msg], (char) 0) == EndStream ) {
            fprintf(stderr,"Filter in %1d hit ES too soon.\n", II) ;
            break ;
            }
        if (pkt_count > 1 ) { /* just grab first packet after ECL */
            break ;
            }
        save = msg ;
        if ((L.Down[msg].header&HVK) != HK ) /* Only count non keys.  */
            pkt_count++ ; /*  Keys repeat after storage mgmt so don't count.  */
        msg++ ; /* deal with next packet.  */
        }

    /* USE MESSAGES */
    L.symbol[0]  = L.Down[save+0].value >> 8 ;
    L.symbol[1]  = L.Down[save+0].value & 0x00ff ;
    }

/* Print out L cell contents here */

}
```

Figure 10. Left rotation of groups - left shift by one place.

The middle part shows the same sequence, numbered in a trivial way to separate the symbols into the four interleaved groups that will undergo individual rotations. Below the data, "send" is the *key* that will be sent with a symbol, and "recv" is the *key* that each

```
/*  Rotlag2 corrects rotlag1 by shifting just the overflow atoms right again.  */

struct  LcellType {
        int                     lb, rb, rln, index, dir[4] ;
        char                    symbol[80], empty, firstL[4], lastL[4] ;
        struct PacketType       Up[ListLength],
                                Down[ListLength];
        } ;

void    InitL() {
char c, brackets[80] ;
int msg_count, i ;

msg_count = 0 ;

/* read local L cell data from input */

/* Send packets */
if (!L.empty && L.lastL[1]) {
    L.Up[msg_count].header  = HC + HL + 02 ; /* right shift par prefix opcode */
    L.Up[msg_count].value   = (L.symbol[0]<<8) + L.symbol[1] ;
    msg_count++ ;
    }
L.Up[msg_count].header  = HE + HC + HL + 0a ; /* ecla */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
L.Up[msg_count].header  = HE + HC + HR + 0a ; /* ecra */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
L.Up[msg_count].header  = HE + HS + HR + 0a ; /* ea - ESRV by convention */
L.Up[msg_count].value   = 1 ;
}

void    PrintL() {
int msg =0, pkt_count =0, save =0 ;

if (!L.empty) {
    for ( ; ; ) {
        if ( BadOrESPacket(L.Down[msg], (char) 0) == EndStream ) {
            fprintf(stderr,"Filter in %1d hit ES too soon.\n", II) ;
            break ;
            }
        if (pkt_count > 0 ) { /* just grab first (CL) packet */
            /* fprintf(stderr,"%d filter bailed at msg %d (pkt %d); kept %d.\n",
                    II, msg, pkt_count, save) ; /* */
            break ;
            }
        save = msg ;
        if ((L.Down[msg].header&HVK) != HK ) /* Only count non keys.  */
            pkt_count++ ; /*  Keys repeat after storage mgmt so don't count.  */
        msg++ ; /* deal with next packet.  */

        }
    /* USE MESSAGES */
    if (L.lastL[1]) {
        L.symbol[0]  = L.Down[0].value >> 8 ;
        L.symbol[1]  = L.Down[0].value & 0x00ff ;
        }
    }

/* Print out L cell contents here */

}
```

Figure 11. Left rotation of groups - right correction of final elements.

```
          A     B     C     D     E     F     G     H     I     J
          e     f     g     h     i     j     a     b     c     d

cell      A     B     C     D     E     F     G     H     I     J
send      0     1     2     3     0     1     2     3     0     1
recv      0     1     2     3     0     1     0     1     2     3
          e     f     g     h     i     j     i     j     c     d

cell      A     B     C     D     E     F     G     H     I     J
send      2     3     0     1     0     1     2     3     0     1
recv      0     1     2     3     0     1     2     3     0     1
          c     d     g     h     i     j     a     b     c     d
```

Figure 12. Straightforward keys fail to rotate data correctly.

PE will be looking for as marking the packet it intends to retain. A problem exists if the rotation distance (here, 4) does not evenly divide the length of the sequence (here, 10). Notice that while G should receive a, the message containing i, having the same key as a, blocks it. As an alternative, cells A, B, C and D could be sent with different keys as shown in the bottom part of the figure. Now the problem is that c blocks e from reaching A and c ends up in both A and I.

The solution is to use additional key values and to alter the key that the leftmost PEs are looking for. This pattern is illustrated for this example in Figure 13. When rotating a sequence of l atoms (the length of the entire string) by k positions, this algorithm passes $k + remainder(l, k) + 3$ messages through the root of the network. This is still O(k) rather than O(l).

```
cell      A     B     C     D     E     F     G     H     I     J
send      2     3     4     5     0     1     2     3     0     1
recv      0     1     2     3     0     1     2     3     4     5
          e     f     g     h     i     j     a     b     c     d
```

Figure 13. Correct keys for rotation.

Figure 14 shows the actual rotation code, as FP would use it to rotate a sequence that consisted of arbitrary objects. Before this can occur, all PEs must determine k and l in order to calculate their *send* and *recv*; this is shown in Figure 15.

```
/* Use 'send', 'recv' to rotate symbols the desired `k` places.  */

struct  LcellType {
        int                     lb, rb, rln, index, dir[4] ;
        char                    symbol[80], empty, firstL[4], lastL[4] ;
        struct PacketType       Up[ListLength], Down[ListLength];
        int                     length, k, send, recv ;
        } ;

void    InitL() {
char c, brackets[80] ;
int msg_count =0, i ;

/* read local L cell data from input */

/* Send packets */
L.Up[msg_count].header  = HE + HC + HL + 0a ; /* ecla */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
if (!L.empty) {
    L.Up[msg_count].header  = HC + HR + HK + 15 ; /* key k0 */
    L.Up[msg_count].value   = L.send ;
    msg_count++ ;
    L.Up[msg_count].header  = HC + HR + 01 ;
    L.Up[msg_count].value   = (L.symbol[0]<<8) + L.symbol[1] ;
    msg_count++ ;
    L.Up[msg_count].header  = HC + HR + 01 ;
    L.Up[msg_count].value   = L.lb ;
    msg_count++ ;
    L.Up[msg_count].header  = HC + HR + 01 ;
    L.Up[msg_count].value   = L.rb ;
    msg_count++ ;
    }
L.Up[msg_count].header  = HE + HC + HR + 0a ; /* ecra */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
L.Up[msg_count].header  = HE + HS + HR + 0a ; /* ea - ESRV by convention */
L.Up[msg_count].value   = 1 ;
}

void    PrintL() {
int msg =0, pkt_count =0, save ;

if (!L.empty) {
    for ( ; ; ) { /* RECEIVE MESSAGES */
        if (pkt_count > 3*L.recv + 1 ) { /* 3 CR packets per symbol + EOCL */
            break ;
            }
        save = msg ;
        if ((L.Down[msg].header&HVK) != HK ) /* Only count non keys.  */
            pkt_count++ ; /*  Keys repeat after storage mgmt so don't count.  */
        msg++ ; /* deal with next packet.  */
        }

    L.symbol[0] = L.Down[save+0].value >> 8 ; /* USE MESSAGES */
    L.symbol[1] = L.Down[save+0].value & 0x00ff ;
    L.lb        = L.Down[save+1].value ;
    L.rb        = L.Down[save+2].value ;
    }

/* Print out L cell contents here */

}
```

Figure 14. Rotating k places.

```
/* Find 'send' and 'recv' keys to rotate sequence of 'l' symbols 'k' places.  */

#include        ".constants.h"

struct  LcellType {
        int                     lb, rb, rln, index, dir[4] ;
        char                    symbol[80], empty, firstL[4], lastL[4] ;
        struct PacketType       Up[ListLength], Down[ListLength];
        int                     length, k ;
        } ;

void    InitL() {
char c, brackets[80] ;
int msg_count =0, i ;

/* read input L cell data following address generation */

/* Send packets */
L.Up[msg_count].header  = HC + HL +  ((II==Size-1) ?  O2c :  Op) ;
L.Up[msg_count].value   = ((II==Size-1) ?  0 :  !L.empty) ;
msg_count++ ;
L.Up[msg_count].header  = HE + HC + HL + Oa ; /* ecla */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
L.Up[msg_count].header  = HE + HC + HR + Oa ; /* ecra */
L.Up[msg_count].value   = 1 ;
msg_count++ ;
L.Up[msg_count].header  = HS + HR + Op ;
L.Up[msg_count].value   = !L.empty  &&  (L.dir[0]==1) ; /* full length */
msg_count++ ;
L.Up[msg_count].header  = HS + HR + Op ;
L.Up[msg_count].value   = !L.empty  &&  (L.dir[1]==1) ; /* length of 1st part */
msg_count++ ;
L.Up[msg_count].header  = HE + HS + HR + Oa ; /* ea - ESRV by convention */
L.Up[msg_count].value   = 1 ;
}

void    PrintL() {
int msg =0, pkt_count =0, save =0, send =0, recv =0 ;

if (!L.empty) {
    for ( ; ; ) { /* RECEIVE MESSAGES */
        if (pkt_count > 0 ) {
            break ;
            }
        save = msg ;
        if ((L.Down[msg].header&HVK) != HK ) /* Only count non keys.  */
            pkt_count++ ; /*  Keys repeat after storage mgmt so don't count.  */
        msg++ ; /* deal with next packet.  */
        }

    L.index  = L.Down[save+0].value ; /* USE RETAINED MESSAGES */
    L.length = L.Down[save+3].value ;
    L.k      = L.Down[save+4].value ;
    send = recv = L.index % L.k ;
    if (L.index < L.k) send = L.index  +  L.length%L.k ;
    if (L.index+L.k >= L.length) recv = L.length%L.k + L.index + L.k - L.length ;
    }

    /* L cell contents printed here */
}
```

Figure 15. Finding k and l when rotating one FP object.

6 Conclusions and future work

This paper describes an executable specification for the combining network that was designed for the FFP Machine. As a complete and definitive description, it gathers all the details into a single document, which is a necessary precursor to any hardware implementation, and checks that they work. As an executable description it can substantiate both the anticipated ability of the network to support certain operations that are fundamental to the operation of the FFP Machine, and the expected simplicity of the hardware required to implement the network.

In addition to debugging the network design, the system has provided a reasonably convenient environment for designing and debugging communication algorithms themselves. A prior version provided the first working method for generating directories, and this system quickly showed several trivial errors in detail in the transpose and rotation algorithms.

Extensions to the system are possible at various levels. Cumulative operations involving opcodes too complex to fit in the message processor can still be effectively handled, by repeatedly shifting partial values to neighboring PEs which then perform the calculation. This would involve, first, using the *simple end* packets to vote on whether the reduction is complete (this is already available), and second, arranging that the program return the result of the vote to the operating system to enable the program to be invoked an appropriate number of times. A similar need exists in some matrix multiplication operations. This change is trivial.

A more important extension relates to the design of the receiving mechanism within the PEs. It is entirely the responsibility of the receiving PE to detect and keep the appropriate packets, but currently, the only facility for doing so is to count incoming packets and at a certain point retain a few consecutive ones for use after the message wave has completed. While arbitrary processing of incoming packets is not possible, given that PEs must accept packets at network speeds, some extensions that are only slightly more complicated (matching specific values in the packet stream, rather than specific positions) appear to be sufficient to implement some interesting graph matching algorithms quickly [Smith].

Acknowledgements

The design of the network that is simulated here was developed mainly during the period 1982 – 1984 under the direction of Professor Magó and with some other students.

A precursor to this system was developed at that time to study algorithms and issues in the network design.

References

[Backus] J. Backus, *"Can programming be liberated from the von Neumann style? A functional style and its algebra of programs"*, Communications of the ACM, Volume 21 No. 8, pp. 613-641, August 1978.

[HPF] High Performance Fortran Forum, *"High Performance Fortran Language Specification"*, Version 1.0, May 1993.

[Magó] G. Magó, private communications.

[Magó&Stanat] G. Magó and D.F. Stanat, *"The FFP Machine"*, in *Topics in High-Level Language Computer Architecture*, by V. Milutinović, Computer Science Press.

[Middleton&Smith] D.J. Middleton and B.T. Smith, *"FFP Machine Support for Language Extension"*, Nineteenth Hawaiian International Conference on System Sciences. pp. 59-66. January 1986.

[Partain] W. Partain and G. Magó, private communication.

[RP3] G.F. Pfister, *"The Architecture of the IBM Research Parallel Processor Prototype (RP3)"*, Nineteenth Hawaiian International Conference on System Sciences. pp. 214-221. January 1986.

[Schwartz] J.T. Schwartz, *"Ultracomputers"*, ACM Transactions on Programming Languages and Systems, Volume 2 No. 4, pp. 484-521, October 1980.

[Smith] B.T. Smith, private communication.

Appendix - Additional program listings

```
/* Algorithm designers may change first 2 values, to vary area size.  */

#define Size            32
#define LogSize         5
#define HalfS           (Size>>1)

#include        <stdio.h>
#include        <string.h>

#define ListLength      (Size*40)       /* 40 pkts per cell, 8 is realistic */

#define HC              0x00            /* Cumulative       -    2nd    */
#define HS              0x80            /* Simple            } field   */
#define HCS             0x80            /* CS field         -          */
#define HL              0x00            /* Left to Right    -    3rd    */
#define HR              0x40            /* Right to Left     } field   */
#define HLR             0x40            /* LR field         -          */
#define HB              0x00            /* Body             -    1st    */
#define HE              0x20            /* End               } field   */
#define HBE             0x20            /* BE field         -          */
#define HV              0x00            /* Value            -    4th    */
#define HK              0x10            /* Key               } field   */
#define HVK             0x10            /* VK field         -          */

#define Scarry          0x08            /* Bit positions for state vbls */
#define Sloser          0x03
#define Smin            0x30

#define SNoLoser        0x00            /* State values for Loser       */
#define SRLost          0x01
#define SLLost          0x02
#define SEq             0x00            /* State values for minimum     */
#define SLess           0x10
#define SGtr            0x20
#define SNoCarry        0x00            /* State values for addition    */
#define SIsCarry        0x08

#define BadPacket       2               /* Qualities of a packet.  */
#define EndStream       1
#define NotEndStream    0

#define Oo              1               /* Opcodes.  */
#define O2c             2
#define O1c             3
#define Om              4
#define Omc             5
#define O2              6
#define O1              7
#define Op              8
#define Opc             9
#define Oa              10
#define Ox              11

#define ValueMask       0x0ffff         /* Subfields in a packet.  */
#define KeyMask         0x0f
#define TypeMask        0xf0
#define PacketCarry     (ValueMask+1)

struct  PacketType {
        unsigned char header ;
        unsigned int value ;
        } ;
```

Figure 16. ".constants.h"

```
/*  This file animates a set of routines that describe
/*      an FFP Machine algorithm as distributed local actions.
/*  It
/*    declares arrays of the appropriate size,
/*    defines some naming macros for the local algorithms to use, and
/*    iterates up and down the tree once, repeating those routines.
/*
/*  It must be included  ** AFTER **  the Lcell and Tcell definitions,and
/*    ** BEFORE **  the routines that use the naming macros.
/*
/*  Naming convention:
/*    User routines should avoid variables starting with capitals.
 */

void    InitL () ;
void    InitT () ; /* probably no use:  T cells are initially empty */
void    Lup () ;
void    Tup () ;
void    Troot () ;
void    Tdown () ;
void    Ldown () ;
void    PrintL () ;
void    PrintT () ;

struct  LcellType       Lcells[Size] ;
struct  TcellType       Tcells[Size] ;

int     debug =0 ;
int     II ;                            /* Iterates through the tree.      */
                                        /* Used by naming macros.              */

#define Root            Tcells[1]       /* ref values in the tree root  */
#define Self            Tcells[II]      /* ref values in T cell of interest */
#define LeftChildT      Tcells[2*II]
#define RightChildT     Tcells[2*II+1]
#define LeftChildL      Lcells[2*II-Size]
#define RightChildL     Lcells[2*II+1-Size]
#define L               Lcells[II]

main(argc,argv) int argc ; char **argv ;
{int i,argvalue;
for(i=1;i<argc;i++){
    sscanf(argv[i]+2,"%d",&argvalue) ;
    switch (argv[i][1]){
        case 'd' :  fprintf(stderr,"Debug set to %d\n", debug=argvalue) ;
                    break ;
        default  :  fprintf(stderr,"Tree ignored bad argument %c\n",
                                argv[i][1]) ;
                    break ;
        }
    fflush(stderr) ;
    }
for ( II = 0  ;  II < Size  ;  II++ )           InitL() ;
for ( II = 1  ;  II < Size  ;  II++ )           InitT() ;
for ( II = Size-1  ;  II >= HalfS  ;  II-- )    Lup() ; /* L cell parents */

for (              ;  II > 1  ;  II-- )         Tup() ; /* T cell parents, not Root */
                                                Troot() ;
for ( II = 2  ;  II < HalfS  ;  II++ )          Tdown() ;
for (         ;  II < Size  ;  II++ )           Ldown() ;
for ( II = 0  ;  II < Size  ;  II++ )           PrintL() ;

}
```

Figure 17. ".skeleton.h"