A VLSI Combinator Reduction Engine

by

William. C. Athas, Jr.

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

8 June 1983

The research described in this document was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

Caltech Computer Science Department Document Number: **5086:TR:83**

Table of Contents

List of Figures

1 Introduction

The design of concurrent architectures for high performance computing engines has been speculated upon for more than twenty years [8]. Such systems can now readily be realized with VLSI technology if the two following guidelines are adhered to. First of all, locality in communcation must be preserved since it is intrinsic to the physics of integrated circuits. Secondly, since VLSI is a replication technology, this aspect of the technology can be exploited to reproduce homogeneous parts. Based on these two guidelines, a class of machines has been defined as "Ensemble Architectures" [16]. They are characterized by the regular interconnection of a single computing element replicated many times over.

Architecture experiments are currently underway at Caltech to investigate this class of machines. The machines are primarily differentiated by interconnect strategy and granularity of concurrency. Listed in order of increasing granularity they are:

1. Cosmic Cube - Boolean N-Cube , 128K bytes per processor. [11]
2. Mosaic Mesh - Torus, 1 to 4K bytes per processor.
3. Mosaic Tree - Binary tree, 1 to 4 K bytes per processor. [2]
4. Super Mesh - Torus, Single Instruction Multiple Data Path

As would be expected, if an algorithm can be partitioned so that it maps directly onto the topology of any of these machines, then the finer grain machine will perform more efficiently. However, as the granularity becomes finer, deriving an effective partition is more difficult, if not impossible. This trade-off results in a dichotomy between generality and performance.

An example of this dichotomy is illustrated with the problem of sorting a set of integers into ascending order. Choosing a fine grain concurrency implementation where computation and storage are embedded in the same structure yields the linear array of Figure 1. Each cell can compare its internal value to that of its neighbor and swap if necessary. This type of array is referred to as "systolic" [9] since data is "pumped" through the different cells. The worst case configuration is when the largest element is in a[1] and/or the smallest in a[n]. For this configuration n swaps are necessary where n is the number of

cells in the array. Therefore this implementation is very efficient at sorting but unfortunately, can do nothing more than sort integers, so it is very problem specific.

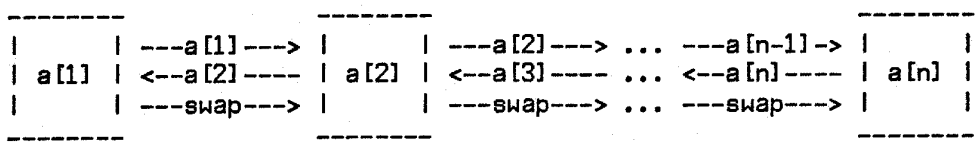


Figure 1: Sorting Element Using Linear Array of Computing Elements

The question arises now as to whether the above dichotomy is absolute, that is, as the granularity of concurrency becomes finer, efficiency increases while generality decreases. The combinator reduction engine described in this paper is an attempt to sidestep the dichotomy by defining an automaton based on simple, primitive cells. The cells can store only a small number of variables and their autonomous sequencing is governed by a small controller. The controllers are homogeneous throughout the machine, but each controller has a programmable action code which is not necessarily the same for all the cells.

Such an approach has been taken before. A very crude example is content addressable storage or the more general Content Addressable Parallel Processors [5]. What is unique about the combinator reduction engine is that cellular behavior is governed by a control structure based on combinators. This approach provides an interesting theoretical basis for the computing instead of an approach where control structures are based on *ad hoc* methods to achieve Turing computability. Before delving any deeper into the mechanics of the engine, it is necessary to discuss the underlying computation model of the engine.

2 Alternative Computation Models

One way to organize the different computation models is through Klaus Berkling's computational hierarchy [1] shown in Figure 2. The hierarchy starts with an abstract notion of computation and represents it in several ways. The Cosmic Cube , Mosaic Mesh, and Super Mesh all rely on the Turing Machine representation since they are ensembles of Von Neuman processors. In the SKIM machine [17] and the REDEX simulator in Section 4,

combinators are also interpreted by a Von Neuman machine. There has also been a combinator evaluator designed explicitly for stack machines [14]. Likewise, LISP, the de facto computer language for the λ calculus, is interpreted on a Von Neuman or stack machine.

The fundamental characteristics of the Von Neuman machine model are a single control sequencer and a global address space. These two features result in a single focus of attention for a computation, yet the attention can shift to or access any information in the data space. Thus programming notations based on this model, indirectly on the Turing machine, rigorously enforce a single instruction stream yet have a disrespect for localizing access to storage. Additional semantics can be embedded in these languages to create and synchronize concurrent activity, an example is Hoare's Communicating Sequential Processes [7] (CSP). Using this paradigm, as the granularity of concurrency becomes finer, the storage space decreases, and the number of processors increases. If this programming model is taken to the extreme, then the processors devolve to state machines, and at the same time, there is a very large number of them. Whether programmers are able to effectively program a large number of state machines interconnected in a regular fashion remains to be discovered. Object oriented programming [10] or Actor type [6] languages may be feasible for this type of ensemble architecture providing the granularity of concurrency is sufficiently coarse.

Choosing the λ calculus for a programming notation under the guise of a LISP-like syntax could yield a different result when the granularity is taken to the extreme. Since the programming notation is based on a computational representation rather than an implementation, changing the implementation could yield results independent of the language. For example, both combinators and the λ calculus have the property of reduction, that is, a given functional expression when supplied with an argument can be rewritten in another form, hopefully simpler. This process continues until the expression converges on some steady state result. An example of this process is the the evaluation of arithmetic

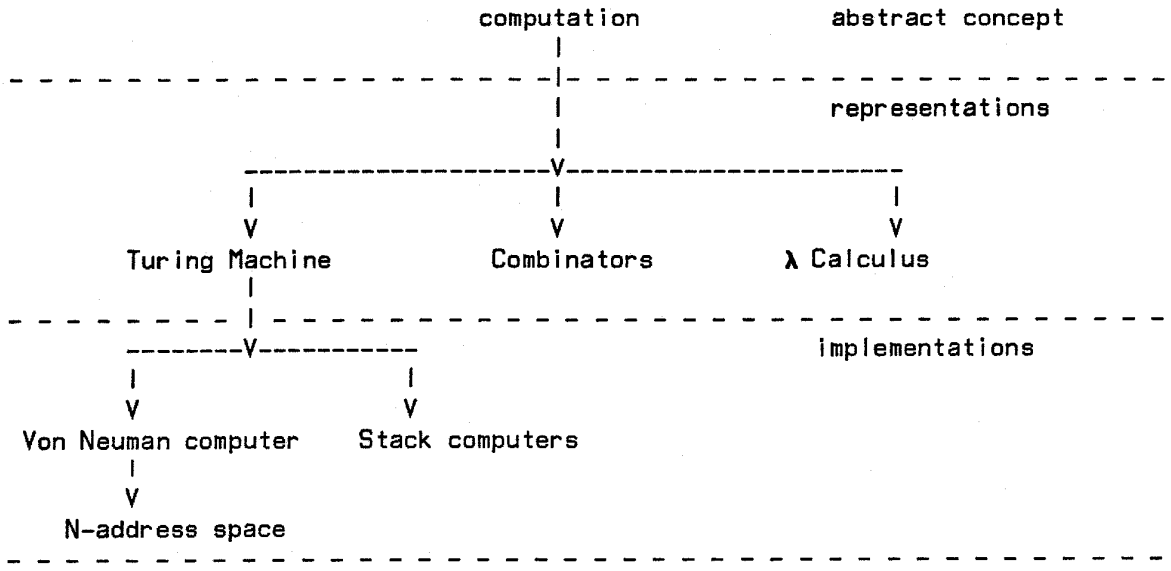


Figure 2: Berkling Hierarchy of Computation

expressions comprised entirely of constants. The rewrite rules for reduction are embedded in the precedence of the arithmetic operations. Figure 3 shows the reduction of the quadratic equation (2+3)*(2+5).

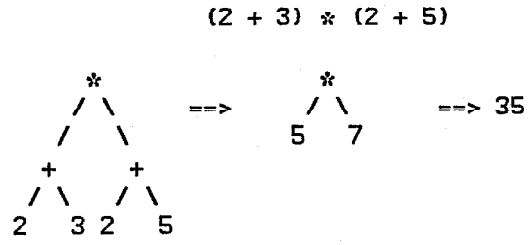


Figure 3: Algebraic Reduction of Quadratic Formula

One might attempt to interpret the λ calculus directly, however the problems to be resolved are formidable. For example, using a LISP-like notation where everything can be represented in binary tree suggests that a tree architecture could be developed for direct execution. If the granularity of concurrency is made coarse, then the system is actually an ensemble of Von Neuman machines that interpret LISP expressions. If concurrency is at the same granularity as the combinator reduction engine, then many formidable issues have to be resolved:

1. Representation of data structures

2. Modeling of recursion
3. Generation of environments for variables
4. Functional application
5. Functional composition

It is difficult if not impossible to find a direct correspondence between the λ calculus and solutions to these problems in a VLSI implementation. Without such a correspondence, ad hoc 'heuristic' techniques must be used to resolve them.

An alternate equivalent representation for the λ calculus is the combinator notation developed by Shonfinkel and brought to the attention of the Computer Science community by David Turner [18]. Direct interpretation of combinators looks promising for the following reasons:

1. Combinators are a variable free notation.
2. No overhead for function composition.
3. No overhead for lazy evaluation.
4. Combinator code can be self-optimizing.

The following is a combinator expression with optimization for a recursive Fibonacci function:

```
(S (S (B COND (LEQ 1)) I) (S (B PLUS (B FIB (C MINUS 1)))
(B FIB (C MINUS 2))))
```

Hopefully no one would want to program in such a notation, but the following LISP-like function is acceptable.

```
(FIB (X) (COND ((= X 0) 0)
               ((= X 1) 1)
               ((> X 1) (PLUS (FIB (MINUS X 2) (MINUS X 1)))
         )
)
```

Through a process of functional abstraction, the combinator expression can be derived from the LISP expression. How this is done will be explained in the following section on the execution model for combinator evaluation.

3 Combinator Computation Model

Turner's paper [18] is the standard source on the use of combinators as an implementation technique for applicative languages and the reader is encouraged to read it. Nevertheless, a brief review of how combinators are used for computing is provided here to present them from a slightly different aspect.

Translation of λ calculus expressions to combinator expressions relies on the inverse operation of functional application being functional abstraction. By abstracting variables from a symbolic expression one at a time, a variable free notation for the function can be derived. In the process of abstraction, information as to where the variable is located in the expression is lost. The information must therefore be kept outside of the abstracted expression in the form of the combinator symbol. The abstraction process can be stated completely using the three following simple recursive rules:

```
[x] (E1 E2) ==> S ([x] E1) ([x] E2)
[x]  y   ==> K y
[x]  x   ==> I
```

The meaning of **[x]** is to abstract variable **x** from the expression immediately to its right. After an expression has been translated into combinators, functional application is used to evaluate the expression. The rules for application are the following:

```
S f g x  ==> f(x  (g x))
K a x    ==> a
I x      ==> x
```

Once a λ expression has been translated into a combinator expression, the combinators give directions as to how the arguments applied to the combinator expressions are to be jockeyed into the correct position for evaluation. As an example, consider the function in Figure 3 which computes the square of an integer number. From the example it is clear that the evaluation of combinator expressions is a reduction process. Furthermore, it is worth noticing that in step 2, the size of the expression increased. From the notation of Curry [4], the S combinator is often referred to as the formalizing combinator, K the elementary cancellator and I the elementary identificator. It is interesting to note that I is equivalent to

```
function (sqr (x) (times x x))

[x] (times x x) ==> S ([x] times x) ([x] x)
[x] (times x)   ==> S ([x] times)   ([x] x)
[x] (x)         ==> I
[x] (times)     ==> (K times)

sqr (x) = S (S (K times) I) I

S (S (K times) I) I) 5 ==> (S (K times) I 5) (I 5))     step 1
                       ==> ((K times 5) (I 5)) (I 5)    step 2
                       ==>  (times 5 5)                 step 3
                       ==> 25                           step 4
```

Figure 4: Completely Worked Example of the SQR Function

SKK. From an informational standpoint, the S combinator duplicates information in the system, K destroys information, and I is the identity.

Even though only two combinators are needed for universal computing, it is clear from this example that combinator expressions can become arbitrarily large. As a matter of fact, in the REDEX and CRE-1 simulators, the compilers must do optimization while compiling else the intermediate expressions exceed the available storage space of the LISP systems. An example of this explosion is the following simple function to compute Fibonacci numbers:

```
(S (S (S (K COND) (S (S (K LEQ) (K 1)) I)) I) (S (S (K PLUS)
(S (K Y) (S (S (K MINUS) I) (K 1)))) (S (K Y)
(S (S (K MINUS) I) (K 2)))))
```

The optimizations that can be applied to a combinator expression are taken directly from Turner's paper:

```
S (K E1) (K E2) ==> K (E1 E2)
S (K E1) I      ==> E1
S (K E1) E2     ==> B E1 E2 if no earlier rule applies
S E1 (K E2)     ==> C E1 E2 if no earlier rule applies
```

Using Curry's notation, B is the elementary compositor and C the elementary permutator. Their application result in the following:

```
B f g x ==> f (g x)
C f g x ==> f (x g)
```

Applying these optimizations to the simple Sqr function:

S (S (K times) I) I ==> S times I

The Fib function optimized was shown previously.

It is interesting that in the abstraction process, expressions are recursively subdivided into smaller expressions until they converge to atomic expressions and then are either tagged with K or I. In this subdivision process, the expressions are subdivided by putting the entire expression except for the last subexpression in E1 and the last subexpression in E2. It would seem more appropriate to try to more evenly divide the expression so that a balanced tree would result. Unfortunately this is not the case, since application is always done by placing the argument next to the tail of the combinator expression. Argument evaluation propagates from right to left. Thus translation must be done in the same direction as application.

In order to have a programming language, the combinators have to be enhanced with a set of primitive operators to execute the standard repertoire of binary arithmetic operations, plus a conditional to provide some type of dynamically alterable control flow. The essential additional primitives are:

```
COND x y z  ==> if x then y else z
TIMES x y   ==> x * y
DIV  x y    ==> x / y
PLUS  x y   ==> x + y
MINUS x y   ==> x - y
PR    x y   ==> CONS x y
HD PR x y   ==> x       { like CAR in LISP }
TL PR x y   ==> y       { like CDR in LISP }
U f PR x y  ==> f x y
Y f         ==> f Y f
```

The U primitive is for converting N-ary functions into monadic ones. N-ary functions represent a problem since the translator is capable only of abstracting a single variable from an expression. Functions must either be monadic or the function has to be converted into a monadic form. This task can be done through a process called "Currying" where application

of a function to a variable produces a new function. Currying is implicit to the abstraction process and is implemented by abstracting a variable while treating the other variables in the expression as constants. Naturally this operation requires the resulting combinator expression to have a special tag, the U combinator, in front to indicate that "Uncurrying" must be done during application. The process continues until all the variables have been abstracted from the function.

During application, when the U combinator is encountered, the argument is expected to be a PR construct. The U causes the PR to be broken down into its left and right parts and then applied to the expression. Unfortunately, the process of functionally abstracting one variable at a time causes the size of the combinator expression to grow as a polynomial of the number of variables in the source expression.

The Y primitive is the fixed point that maps the combinator expression back onto itself. It is also the graphical primitive which introduces cycles into the tree expressions, thus recursion.

4 Combinator Evaluation for a Single Path Reduction Engine - The REDEX Simulator

Before attempting a concurrent evaluator for a combinator system, a single path reduction engine was simulated to gain some initial experience with what could be expected from the direct execution of combinator expressions. The REDEX[1] simulator is a single path reduction engine since it simulates the evaluation of only one point in the combinator tree for any discrete time step. The system is composed of a compiler written in LISP and a simulator written in Pascal. The compiler accepts a pure LISP-like language whose syntax is given in Appendix I. Compilation consists of the input of a source symbolic function, which is then

[1]The name REDEX comes from the λ calculus where the redex of an expression is replaced with its contractum during reduction.

separated into the function name, the arguments, and the body of the function. The list of arguments are sequentially abstracted out of the body of the function from left to right and the resulting expression is appropriately tagged with U primitives. Abstraction is done on a single variable basis and consists of translation using the three recursive rules immediately followed by optimization using the four optimization rules.

The output of the compiler is input to a parser written in Pascal. The parser generates parse trees that are compatible with Turner's trees. As shown in Figure 5, the LISP binary trees and the trees used in the combinator evaluator are not the same. The reason for this difference is the way the trees are evaluated.

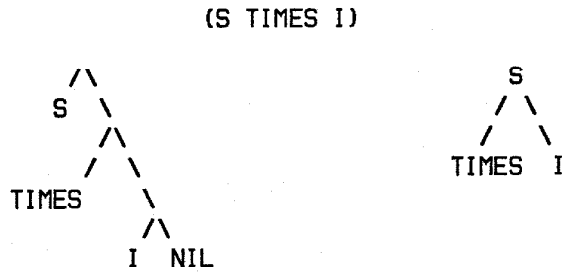


Figure 5: LISP versus Turner Data Structures for the SQR function

Once a function is stored in the simulator, it may be evaluated by applying an argument to it. Evaluation is done interactively by typing the function name followed by the argument it is to be applied to. If the function is n-ary, then the argument must be in a PRed form, for example f(x,y,z) would be entered f (pr x (pr y z)). Once a function and an argument have been entered, the simulator creates a new storage cell and binds the function to the left side, the argument to the right side, and then passes the new tree to the evaluator.

The evaluation sequence for traversing this new tree is very important. One evaluation scheme, called normal order graph reduction, achieves lazy evaluation; the other, called applicative order, is equivalent to conventional inner most to outer evaluation of functions.

The simulator uses normal order evaluation, which is done as follows: The current pointer to the subexpression of the tree traverses the left subtrees saving pointers to the right along the way. When it reaches the bottom, it reduces whatever combinator it found, then backs up the required number of links and continues. The reduction rules are shown graphically in Appendix III. Since combinator trees serve as templates for the reduction sequence, it is crucial never to destroy the original template. A dynamic storage management policy is then mandatory, and as a result, a garbage collector.

A major issue during the implementation of this simulator was proper detection of the end of reduction. There is no stop primitive and functions that generate infinite recursions will never yield a combinator graph that is constant from one iteration to the next. The solution used was to reduce on a basis of need to print. The third phase of the simulator is a pretty printer routine to format the output of the combinator tree. The pretty printer therefore controls the evaluator. For example, consider the following function INF that generates an infinite list of integers:

LISP: (INF (X) (PR X (PLUS X 1)))

REDEX: INF (S PR (B Y (PLUS 1)))

Evaluation of this function on the first invocation of INF 1 yields the result:

PR 1 (B INF 2)

If the user requests to print the HD of this function, the simulator will return 1. If the user asks for the HD (TL (INF 1)), the system will compute:

```
HD (TL (INF 1))
HD (TL (PR 1 (PR 2 (INF 3))))
HD (PR 2 (INF 3))
2
```

Thus evaluation is done solely on a need to print. The combinator graph has the subtle property that it embeds both data and control all in the same tree structure. It is also interesting that since evaluation is driven by the need to print, and evaluation is nothing more than application, it is conceivable to have a system where the abstraction process is

done on the need to evaluate, thus:

printing ==> evaluation ==> application ==> compilation ==> abstraction

If the compiler could compile itself, which could readily be done, then entire functional expressions need not be completely abstracted , but rather parts of the expression would be abstracted based on the need to compute.

5 Foundations for a Concurrent Reduction Engine

The REDEX simulator provided the framework for what can reasonably be expected from combinator tree expressions. The chief desirable qualities of the combinator expressions for direct execution are that they are a variable free notation and that they can be reduced using lazy evaluation. Lazy evaluation is important since it allows the embedding of both data and control in the same physical structure. The main obstacles to overcome are: (1) How to model the dynamic evolution of the reduction on a fixed machine; (2) How to represent large dynamic data structures and (3) How to implement recursion.

The representation of data structures is important for it will determine the physical implementation of the machine's structure. It is desirable to make the cells of the combinator reduction engine functionally compatible, if not interchangeable, with the cells of the data structures. This design decision permits the two different types of cells to interact on an atomic level of concurrency. Therefore, the interconnect structure of the combinator reduction engine is a fixed depth binary tree.

In the REDEX simulator, the reduction of the combinator expression was accomplished by following the reduction rules which generated new subexpressions whenever necessary. For a fixed tree machine this is not possible since tree growth is limited to two dimensions. Tree expansion in a third dimension could be possible if an arbitrarily large stack was incorporated into each node of the machine. Whereas reduction could be conducted in this fashion, it is not desirable for two reasons. First of all, the large stacks do not allow the machine to be constructed of small cellular parts since the stacks will require a considerable

amount of space. Secondly, the stacks will not grow uniformly across the entire machine, thus it would be likely to run out of stack space at one node while having unused stack space at another.

An alternate solution is to treat all of the nodes in the combinator tree as self-timed cells where arguments are allowed to enter the tree only through the root node. A self-timed environment is desirable since the tree can be arbitrarily large with many small reductions being performed concurrently in different parts of the tree. Furthermore, the interaction between neighboring cells conforms naturally to a request/acknowledge protocol.

The execution model is to associate a combinator or primitive tag with each node where the tag determines the behavior of the cell during reduction. Using this configuration, application is accomplished by sending the argument to a cell as opposed to application by juxtapositioning. Since it is not possible to transmit data structures between neighboring cells in the tree, arguments must be represented by packets that are typed as either pointer or atomic. An atomic packet can be operated on directly, whereas a pointer packet provides a level of indirection through to the main store.

An example of how application is actually implemented may be helpful. Consider the simple function which computes the quadratic function of (x+3)*(x+5) applied to 2 . The rules for application are from Appendix II. These rules plus the ones of Appendix V are expressed in a notation based on CSP primitives . The function in graphical form for the quadratic is shown below.

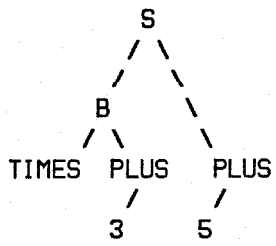


Figure 6: (X+3) * (X+5) Function

Starting with the S combinator the 2 argument is supplied to the B combinator and the PLUS

primitive in the right subtree. PLUS returns a 7 which is sent to the left subtree. Meanwhile the B primitive delivered the 2 to the PLUS 3, which then returns 5. Now both 5 and 7 are sent to TIMES which computes 35 and returns this result to the root.

Lazy evaluation now plays a very important role with primitive operators such as PR, which is now the "do what I am" primitive. To see why this is so, consider the INF function described previously. This function is shown in the graphical expression of Figure 7. The result of evaluating this cell is an acknowledge in the self-timed system sense. Once the acknowledge is received at the root, the pretty printer that interrogates the tree must then send a request for either the HD or the TL of this cell. Thus the PR node serves a dual purpose, it joins data structures and is a data structure.

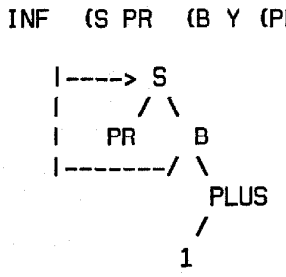


Figure 7: Combinator Tree for Infinite Integer List

The HD and TL operators must also be able to operate on the data structure that is in the main store. This data structure must be described in terms of the PR operator so it is in fact also a combinator tree. Furthermore, it could also be a implemented as a fixed binary tree! If it were another fixed tree, the composition of the expression tree with the argument tree would then form a single tree again.

Since all requests to the main store and from the pretty printer must be channelled through the root node, a bandwidth problem would develop. This is unavoidable however, since the devices that interact with combinator tree are fundamentally sequential access. The function of the root node and the rest of the tree is to diffuse and converge computation into and out of this sequential stream at an exponential rate.

The major problem now with implementing a fixed binary tree for combinator evaluation is the modeling of the Y combinator. This combinator is the fixed point and introduces a cycle into the tree. The cycle can be avoided by supplying an offset for the Y combinator that indicates the number of tree levels that have to be traversed backward before execution can continue downward again. Previously only result packets were allowed to travel up the tree. Therefore a new type must be introduced to allow recursion arguments to work their way back up the tree.

At this point the semantics of the REDEX simulator parse trees become very tedious. As an example, consider the binary COND operator in Appendix III. The COND cell must have influence on the two tree levels above. Clearly this is not an efficient use of locality and it makes the firing rules of the cells much more difficult. The problem with COND is that it is a ternary function in a binary tree. To alleviate this problem, COND can be replaced with two new primitives, J and G. J is the 'join' primitive and it distributes the packet it receives to its left and right subtrees. It expects only one subtree to return a result or recursion packet and from the other subtree it expects an acknowledge packet. The G primitives are always on the tree level directly below a J primitive. G is the 'guard' primitive and it sends the argument received from above to the left subtree first. If the left subtree returns true then it dispatches the packet to the right subtree, else it sends an acknowledge to the cell above.

The compiler for the CRE-1 simulator recognizes these primitives. Aside from simplifying the firing rules for the cells, they introduce concurrency by allowing many guards to be evaluated at once. Naturally, if the guards are not mutually exclusive, nondeterministic results will occur. An example of the use of these guards is shown in Figure 8. The factorial example has been avoided until now since it is so overused in the functional programming community, however here it demonstrates a subtlety about the combinator trees. By playing the packet game on the tree in Figure 8, an application sequence can be derived that

requires no additional storage of packets in the tree but produces the correct result.

```
(FAC (X) (COND ((EQ  X 0) 1)
          ((NEQ X 0) (TIMES (FAC (MINUS X 1))))
      )
)

FAC (J (G (EQ 0) (K 1)) (G (NEQ 0) (S TIMES (B (Y 4) (C MINUS 1)))))
```

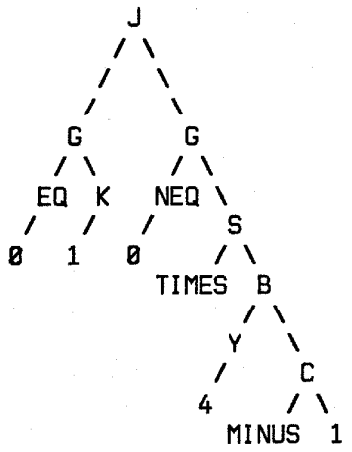


Figure 8: Factorial Function Using Modified Combinator Tree

This is an important observation since it illustrates the relationship between combinator trees, state machines and λ expressions. The combinator tree does have a unique state during a computation since each combinator and primitive has a set of intermediate applications through which it progresses. By taking a snapshot of the tree and examining the applicative state of each cell, the unique state of the machine can be determined. This state is equivalent to some reduced expression of the original combinator expression template.

This example presents a paradox since combinator trees are clearly program schemas but they are also equivalent to λ expressions. λ expressions however, cannot be represented by program schemas [15] and require a pushdown automata for implementation. The paradox is resolved by noticing that the recursion in the factorial function is "linear recursion" [3]. Algorithms exist to translate linear recursion directly into an iterative form.

This iterative form can be executed on the combinator tree by modifying the behavior of the cells to execute special rules.

Linear recursion is identified by having only one recursive call in a function. An alternate graphical method for determining linear recursion in a combinator tree is the following: the different combinators and primitives either duplicate, destroy or maintain the number of packets in the tree. Obviously if there are no cycles in the tree, then the number of producible packets is bounded. In the factorial example, there is a cycle but it is located so that the number of packets in the circuit remain constant. For example, the S combinator receives one packet but produces two, the TIMES primitive receives two packets and produces one. The reader can confirm that the other primitives and combinators in the circuit keep the packet count constant. If the number of packets in the circuit can be shown to be bounded, then the recursion present is linear.

An example of a tree recursive function is the Fibonacci function presented previously. The Fibonacci function is not a good example however since there is a functional dependence between the two recursions. This functional dependence can be used to directly produce an iterative form.

Since non-linear recursion can generate an unbounded number of packets in the tree, either each cell has to be a pushdown automata or non-linear recursive arguments must exit the combinator tree and be temporarily saved in an auxiliary store. The initial solution was to adopt the wasteful policy of providing each node in the tree with its own local stack. These large stacks are undesirable since they do not scale well in VLSI, plus they limit the machine's performance at the node level. If a fixed binary tree was not used for the implementation, the local stack problem could be alleviated in many ways.

For example, if the combinator tree was mapped onto a toroidal mesh [13], a practically infinite combinator tree could be interpreted. The local store on each of the processing elements in the torus would be used for combinator and primitive codes and for

the stack. Cycles in the graph caused by the Y primitive could either be resolved by backtracking up the tree or by allowing circuits in the mesh. This same approach would also work for the Cosmic Cube which has the additional feature that the concurrent combinator evaluator could be readily implemented. Clearly as the concurrency becomes coarser, the combinator trees become easier to implement.

6 The CRE-1 Simulator

To evaluate the trade-offs of the different recursion schemes for the binary fixed tree, the construction of a second simulator was undertaken. Denoted CRE-1 for Combinator Reduction Engine-1, the simulator was designed explicitly for a fine grain fixed binary combinator tree. The compiler was written in LISP and its BNF description is given in Appendix IV. This compiler is the same as the REDEX compiler but has been enhanced to compile the J and G primitives.

The simulator is written in Mainsail and uses a recursively defined data structure to model the behavior of the combinator cells. The reduction rules of Appendix III were modified so that application was done by transmitting arguments to the cells in the combinator tree. The interesting aspect of the simulator is how it handles recursion. The combinator tree is fixed after the functional expression is read in and stored in the parse tree. This fixed part will be referred to as the template. After a recursion is encountered, new cells are dynamically allocated to store the current values of the cell and provide space for the recursion values. The net effect is to stack values using a linked list structure. The depth of the stack is dynamic but limited by the number of reachable unused cells on the periphery of the template.

When the fixed point is encountered, the packet is tagged as type recursive, the offset is loaded , and the packet then travels upward through the tree. At each branch cell the packet offset is decremented as was described previously. When a cell receives a recursive packet from a sub-tree, it keeps track of the direction of the recursive call it has

received by pushing the direction onto a two bit wide stack. Two bits are necessary to encode the directions of left, right and up. It is also the pushdown automata that transforms the combinator tree from a program schema into an executable representation of the combinator expression.

This information is sufficient to unravel the recursion of all the combinators, but problems develop with the J and S cells. As was mentioned previously, the combinators and primitives either duplicate, maintain, or remove packets in the tree. J and S duplicate packets, and if their subtrees are concurrently evaluated, there is the possibility that both will return with recursion requests. This condition occurs in the Fibonacci function with the S combinator, and as a result, the subtrees must be evaluated sequentially. Sequential transversal of the subtrees is still not sufficient since if one subtree returns with a recursion request, the original argument destined for both subtrees has to be safely stored somewhere. It cannot remain in the branch cell since these cells cannot save packets, only direction bits. The solution is to send the argument to the non-evaluated subtree. The packet is typed as data and will travel down the subtree until it reaches a leaf of the template. Although the argument is typed as data, the packet has a non-zero offset so the packet is stored and retyped as recursion. It now travels back up the tree, saving the path so that the recursion will unravel correctly when a packet of type unravel is received.

The rules of Appendix V will compute both the factorial and Fibonacci functions but have not been used with symbolic or N-ary functions yet. Since the execution steps for the different cells become very tedious, a stack convention was adopted to simplify the actions. On each new invocation of a cell, a sequence of directions bits are looked up and then pushed onto the cell's bit stack. These control bits will determine the initial response of the cell. As recursion is encountered the bit stack will grow until the packet offset reaches zero. Only the J and S cells detect zero packet offsets, after which they change the packet type from recursion to data and send it back down the tree. J also monitors result packets and if

the stack is not empty, it will retype them as unravel and send them back down the tree.

The recursion model in this approach is feasible since dynamic allocation of new cells occurs only at the periphery of the template. This property of periphery nesting is provable since the abstraction process always appends the combinator tag to the head of the combinator expression under formation. In the parsing phase, this cell is always put at a branch in the tree. Packets are always stored directly below primitives which are on the leaf nodes of the template. When both leaf nodes contain packets, the cell is in a fireable state. The question now is, what happens when the tree is exhausted? This problem can be delayed by creating two linked lists from the template leaf, an elegant trick since it conforms to the loading stage of the machine that is described in the next section and it also fills out the unused portion of the fixed tree with recursion values. Nevertheless, there is the possibility that the tree will still be exhausted. The simplest solution is to design special leaves for the physical tree which are large sequential buffers for the linked lists. Thus only the template for the combinator function need be stored in the fixed tree, the intermediate results can be either dispatched to the main store or kept in the linear buffers at the leaf nodes.

7 Speculation on a VLSI Implementation

A primary concern of this thesis has been whether a realizable combinator reduction engine could be constructed at all with no immediate inhibitions to efficiency issues. Efficiency measurements must wait until a complete, robust CRE-1 simulator is available. The scalability and physical design of the cells of the machine however, can be discussed. Scalability has been a primary concern since this project is explicitly targeted for VLSI. In reviewing the necessary components per cell, the following is required for a hardware implementation:

1. Arithmetic Logic Unit
2. Controller (Probably microcoded)
3. 2 bit wide stack

4. 3 bidirectional serial interfaces
5. Register to hold packet
6. Register to hold cell code
7. Register to hold value of cell

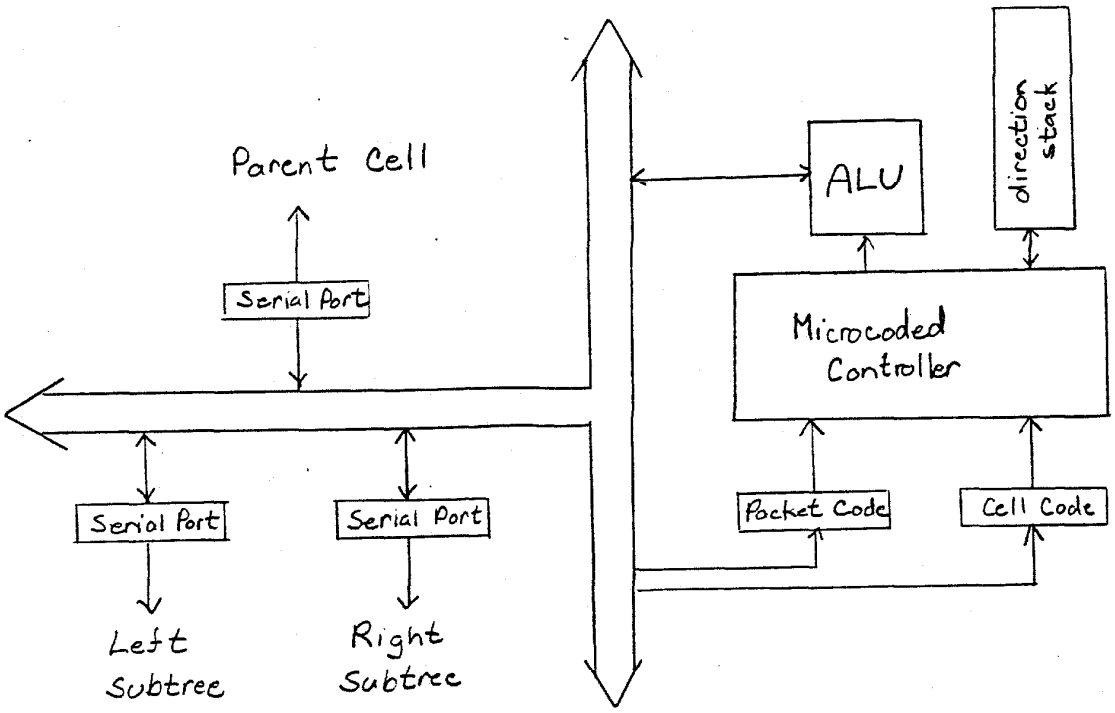
Serial interfaces are proposed since the bit length of the packets may be arbitrarily long. It is reasonable to presume that many branch cells could be packaged upon a single chip carrier. Using a 40 pin package and serial I/O, a four level tree could be fabricated per chip. A block diagram of the functional layout for the branch cells is shown in Figure 8. The microcoded controller would be based on the same execution rules of Appendix V.

The microcoded controller must also have the capability to load itself with a combinator tree upon initialization. This is crucial since this mechanism must also be used for storing values during recursion. The easiest loading method is to define a new packet type called load. Each time a cell receives a load packet, it alternates moving the old cell code between the left and right subtrees and saves the new code. Whether it sends to the left or right is not important providing the entire tree can be initialized to the same direction. This scheme is compatible with the recursion model for filling out the unused branches of the physical tree. However, an unfortunate aspect of this implementation model is the two bit wide stack. Unlike the method for stacking arguments, directions must be saved at the branch cells, thus limiting the performance of the machine at the cellular level instead of at the ensemble level.

8 Conclusions and Future Work

The purpose of this paper has been to propose a cellular architecture for averting the problems associated with Von Neuman computing machines as the granularity of concurrency becomes finer. The path taken by the CRE-1 machine to circumnavigate these problems is shown in Figure 10. The implementation projected in the previous section shows promise for a successful machine. The most interesting results are the elegant scheme for handling recursion and the lazy evaluation property. Both of these characteristics are a result of the

Figure 9: Block Diagram for CRE-1 Machine



use of combinators. The major drawbacks are the need for local two bit wide stacks, and the suppression of concurrency for programs that have tree recursion.

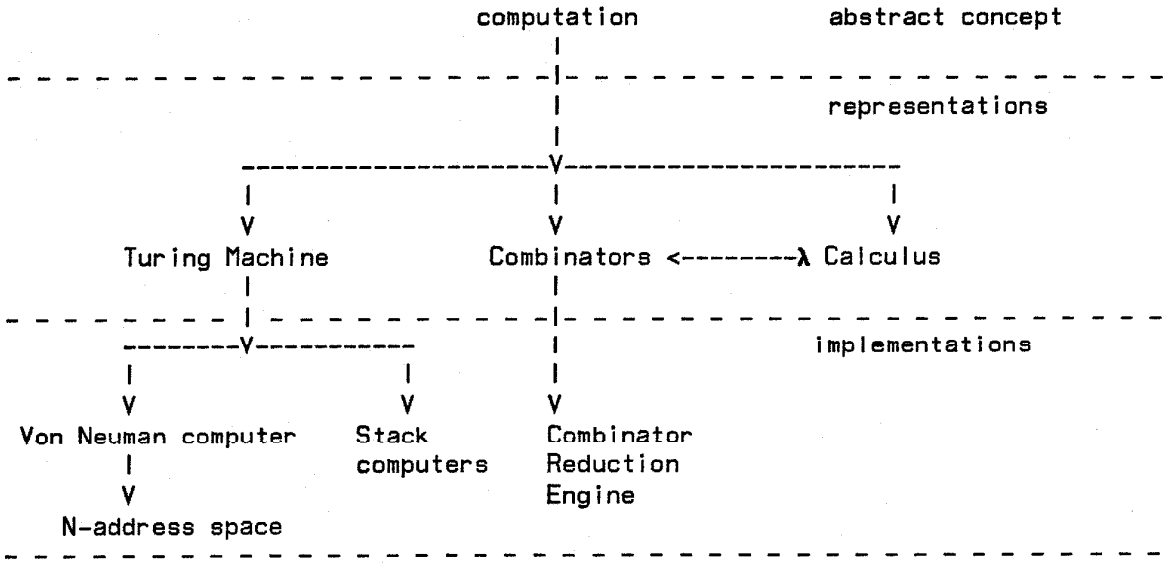


Figure 10: Computation Hierarchy Including CRE-1 Machine

A major question left to be answered is given that the combinators are an equivalent representation for the λ calculus, are there others which are even better suited for a hardware implementation? An ideal solution would be a set of primitives that would not need to store internal state at the branches. The necessary state would be encoded in how data was forced onto the leaves during recursion. This data driven scheme would greatly increase the computing power per silicon device area by using simpler firing rules and the eliminating of the direction stacks.

Work to be completed now is to bring the entire CRE-1 simulator system together. One version of the simulator does linear recursion correctly and another recursion by tree expansion, but the two have not been coalesced into one. The current storage manager needs to be enhanced to allow the combinator tree to fully manipulate the data structures in the main store. The firing rules of Appendix V also need to be completely transferred to the simulator so that simulation is driven by said rules and not by procedure invocation in

Mainsail. Once these technical problems are resolved and the simulator is stable enough so that a compiler written in the CRE-1 LISP notation can compile itself, performance evaluation will be under taken. This thesis has provided a framework for the implementation of a concurrent combinator reduction engine, the results of the performance evaluation will determine whether an experimental VLSI engine should be constructed.

Acknowledgements

I would like to extend my deepest appreciation to Chuck Seitz for his encouragement, endless patience, and support in my research of these computing engines. I am also indebted to Alain Martin for the use of his programming notation in describing the behavior of the CRE-1 cells.

I would also like to thank Dudley Irish and Bob Pendleton for their enthusiasm and continuing interest in the CRE-1 machine. Finally, I would like to thank Al Davis for his guidance and assistance during my undergraduate curriculum.

I. REDEX LISP BNF

BNF notation for REDEX LISP

```
<fn>      ::= (<fnname> (<varlist>) <object>)
<fnname>  ::= <symbol>
<varlist> ::= <atom> | <atom> <varlist> | {}
<object>  ::= <expr> | <atom>
<expr>    ::= (COND <test> <object> <object>) |
              (PR <object> <object>) |
              (TIMES <object> <object>) |
              (DIV  <object> <object>) |
              (PLUS  <object> <object>) |
              (MINUS <object> <object>) | <fnname>

<test>    ::= (EQ   <object> <object>) |
              (NEQ  <object> <object>) |
              (LT   <object> <object>) |
              (GT   <object> <object>) |
              (LE   <object> <object>) |
              (GE   <object> <object>) | <bool>

<atom>    ::= <symbol> | <number> | <bool>
<bool>    ::= NIL | T
<symbol>  ::= <char> <subsym>
<subsym>  ::= <char> <subsym> | <number> <subsym> | {}
<char>    ::= A | B | C | ... | X | Y | Z
<number>  ::= - <digit> <subnum> | <digit> <subnum>
<subnum>  ::= <digit> <subnum> | {}
<digit>   ::= 0 | 1 | 2 | ... | 9
```

{} denotes empty string

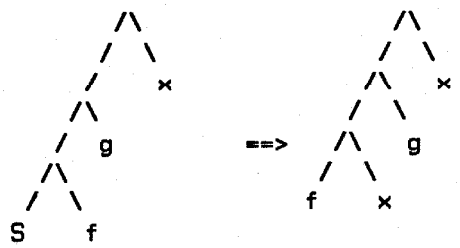
II. Simple CSP Rule Set

Simple rules for computing combinator function in Figure 6. Rules are expressed in CSP-like syntax which is explained in Appendix V

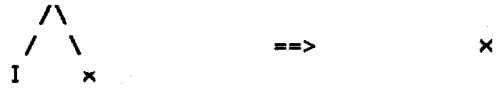
```
S:    *[ U?x;
         (L!x // R!x);
         R?x;
         L!x;
         L?x;
         U!x
       ];

B:    *[ U?x;
         L!x; L?x;
         R!x; R?x;
         U!x
       ]

PLUS , TIMES: *[ U?x;
            [  qR > 0 --> R?y;
                      x.val := x.val OP y.val;
                      x.type := RESULT

             | qL > 0 --> L?y;
                      x.val := x.val OP y.val;
                      x.type := RESULT

             | qR = qL --> L!x;
                       x.type := ACK
             ];
             U!x
            ]
```

III. REDEX Reduction Rules in Graphical Form

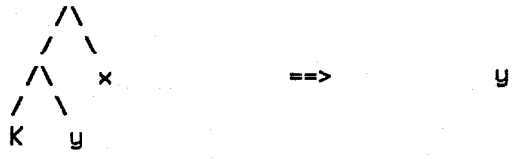
S:



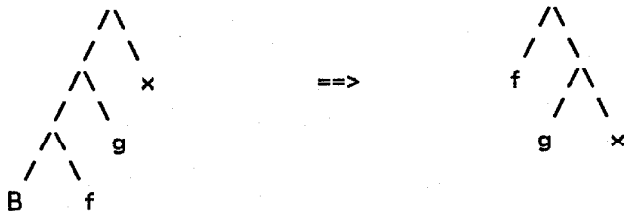
I:



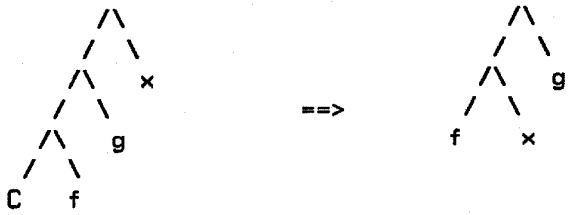
K:



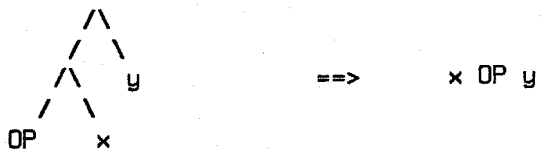
B:



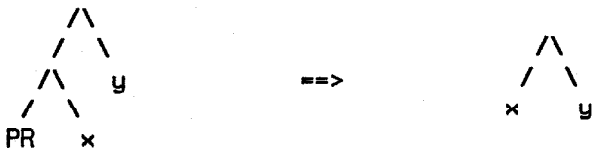
C:



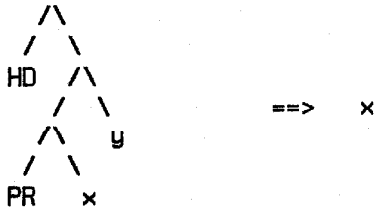
TIMES, DIV , PLUS , MINUS, EQ , NEQ:



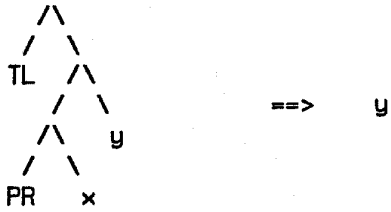
PR:



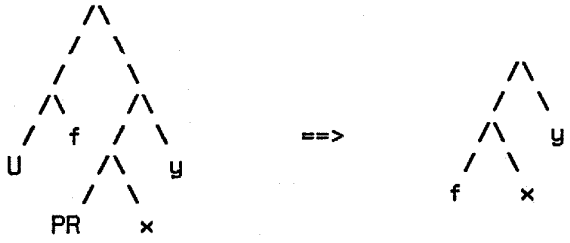
HD:



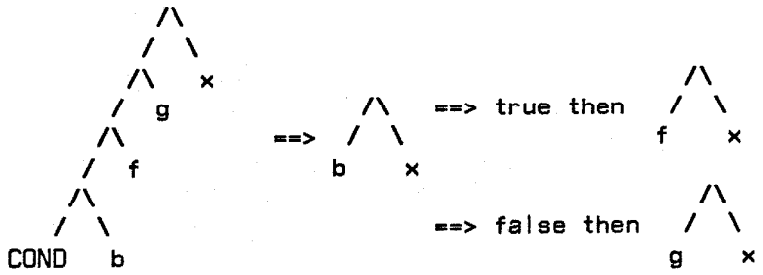
TL:



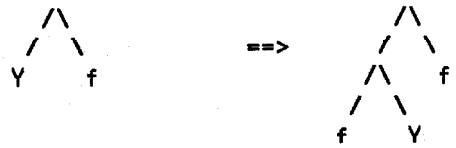
U:



COND:



Y:



IV. BNF Description of CRE-1 LISP Language

BNF notation for CRE-1 LISP

```
<fn>       ::= (<fnname> (<varlist>) <object>)
<fnname>   ::= <symbol>
<varlist>  ::= <atom> | <atom>  <varlist> | {}
<object>   ::= <expr> | <atom>
<expr>     ::= (COND <guardset>)
               (PR <object> <object>) |
               (TIMES <object> <object>) |
               (DIV  <object> <object>) |
               (PLUS  <object> <object>) |
               (MINUS <object> <object>)


<guardset> ::=  <guard> | <guard> <guardset>

<guard>    ::= (<test> <object>)

<test>     ::= (EQ   <object> <object>) |
               (NEQ  <object> <object>) |
               (LT   <object> <object>) |
               (GT   <object> <object>) |
               (LE   <object> <object>) |
               (GE   <object> <object>) | <bool>


<atom>     ::= <symbol> | <number> | <bool>
<bool>     ::= NIL | T
<symbol>   ::= <char> <subsym>
<subsym>   ::= <char> <subsym> | <number> <subsym> | {}
<char>     ::= A | B | C | ... | X | Y | Z
<number>   ::= - <digit> <subnum> | <digit> <subnum>
<subnum>   ::= <digit> <subnum> | {}
<digit>    ::= 0 | 1 | 2 | ... | 9
```

{} denotes empty string

V. CSP Firing Rules for CRE-1 Combinator Tree Cells

The following syntax was developed by Alain Martin [12] based on Hoare's CSP semantics except for the following differences:

1. The label preceding the input and output commands are channel names.
2. qX is the number of packets waiting on channel X.
3. // denotes concurrency.

The letters R,L, and U refer to the directions of right, left and up respectively, that are pushed onto the direction stack. Certain combinators and primitives like B are broken down into separate cases depending on whether they apply one argument or two.

{ makestk(cell) pushes the following sequences onto the bit stack

```
    B1 : R L U              B2 : R L U L U
    C  : L R L U             S : L R L U
    J  : L R U              G  : L if true R U else U
    K  : L U                I  : L
  CONST : U                   Y : L R U
   OP1 : R U                 OP2 : L U R U
}
```

```
I: *[U?x; [   x.type = DATA -->
              [ x.offset > 0 --> pop(dir);
                                 [   dir = L --> L!x; push(R);
                                   | dir = R --> R!x; push(L)
                                 ];
                                 x.type := RECURS
              | x.offset = 0 --> x.type := RESULT
              ]
        | x.type = UNRAVEL --> pop(dir);
                               [   dir = L --> R?x;
                                 | dir = R --> L?x
                               ]
       ]

    U!x;
   ];

K: *[U?x; [   x.type = DATA -->
              [ x.offset > 0 --> R!x;
                                 x.type := RECURS

              | x.offset = 0 --> L!x;
                                 x.type := RESULT;
                                 L?x
              ]
        | x.type = UNRAVEL --> R?x;
       ]
  U!x;
   ];
```

```
J: *[U?x;
    [ x.type = DATA -->
            makestk(cell); { load directions onto stack }
            pop(dir);      { get first direction }

            y := x;        { save arg }

            dir!x; dir?x;
            [  x.type = RECURS -->
                    dir1 := tos;
                    push(dir)
                    dir1!y;  dir1?y;
                    x.offset := x.offset - 1;
                    [x.offset = 0 --> x.type := DATA]

             | x.type <> RECURS -->
                    pop(dir);
                    [ x.type = ACK -->  dir!y; dir?x ];
                    [ x.type = RECURS -->
                            x.offset := x.offset - 1;
                            push(dir);
                            [x.offset = 0 --> x.type := DATA]
                     | x.type <> RECURS --> pop(dir)

                    ]
            ]
    ];

    [ x.type = UNRAVEL --> pop(dir);
                    [dir <> U --> dir!x; dir?x;
                                  pop(dir);
                     ];
    ];
    [  x.offset = 0 ∧ x.type = RECURS --> x.type := DATA
     | x.offset > 0 ∧ x.type = RECURS --> U!x;
     | x.type = RESULT ∧ qS > 0      --> pop(dir);
                                  x.type := UNRAVEL;
                                  dir!x; dir?x
     | x.type = RESULT ∧ qS = 0      --> U!x
    ]
 ]
```

```
G: *[U?x;
    [  x.type = DATA -->
            maktree(cell);
            x.type = DATA;
            y := x;
            pop(dir);
            dir!x; dir?x;
            [  x.type = RECURS --> x.offset := x.offset - 1]
             | x.type <> RECURS -->
                      pop(dir);
                      [  x.val --> dir!y ; dir?x
                       | NOT x.val --> x.type := ACK;
                                  pop(dir)
                      ]
            [  x.type = RECURS --> x.offset := x.offset - 1
                             push(dir);
             | x.type <> RECURS --> pop(dir)
            ]


      | x.type = UNRAVEL --> pop(dir);
                      [dir <> U --> dir!x; dir?x;
                                 pop(dir);
                      ]

    ];
   U!x
  ]

Y: *[U?x;
    [  x.type = DATA -->
            [  x.offset = 0 -->
                      L?y;
                      x.offset := y.val;
                      x.type := R!x;

             | x.offset > 0 --> { we are caught up in another recursion }
                      R!x;
                      x.type := RECURS
            ]

      | x.type = UNRAVEL -->
             qR > 0 --> R?x;
                    x.type := RECURS;
                    L?y
                    x.offset := y.val
    ]
   ]
```

```
S: *[U?x;
   [ x.type = DATA -->
          makestk(cell); { load directions onto stack }
          pop(dir);      { get first direction }
          y := x;        { save arg }

          dir!x; dir?x;
          [ x.type = RECURS -->
                 dir1 := tos;
                 push(dir)
                 dir1!y;  dir1?y;
                 x.offset := x.offset - 1;

          | x.type <> RECURS -->
                 pop(dir);
                 dir!y; dir?x;
                 [ x.type = RECURS -->
                        x.offset := x.offset - 1;
                        push(dir);
                  | x.type <> RECURS -->
                        pop(dir)
                        dir!x; dir?x;
                        [x.type = RECURS -->
                               x.offset := x.offset - 1;
                               push(dir)
                         ]
                 ]
          ]
   ]

   | x.type = UNRAVEL --> pop(dir);
                 [dir <> U --> dir!x; dir?x;
                        pop(dir);
                 ];
                 U!x;
]
```

```
B: *[U?x;
    [  x.type = DATA V
       (x.type = RECURS Λ x.offset = 0) -->
              maktree(cell);
              x.type = DATA;
              pop(dir);
             *[dir <> U Λ x.type <> RECURS --> dir!x; dir?x;
                                        pop(dir);
              ]
              [x.type = RECURS --> x.offset := x.offset - 1];


      | x.type = UNRAVEL --> pop(dir);
                        [dir <> U --> dir!x; dir?x;
                                   pop(dir);
                        ]

    ];
   U!x
  ]

C: *[U?x;
    [  x.type = DATA V
       (x.type = RECURS Λ x.offset = 0) -->
              maktree(cell);
              x.type = DATA;
              pop(dir);
             *[dir <> U Λ x.type <> RECURS --> dir!x; dir?x;
                                        pop(dir);
              ]
              [x.type = RECURS --> x.offset := x.offset - 1];


      | x.type = UNRAVEL --> pop(dir);
                        [dir <> U --> dir!x; dir?x;
                                   pop(dir);
                        ]

    ];
   U!x
  ]
```

```
{ PLUS , MINUS , TIMES , DIV , EQ , NEQ , GT , LT , GE , LE }

OP: *[U?x;
    [ x.type = DATA -->
                [ x.offset = 0 -->
                     [ qL = 0 ∧ --> L!x;
                       qR = 0       push(L);
                                    x.type := ACK;

                     | qL > 0 ∧
                       qR = 0   --> L?y;
                                    x := x OP y;
                                    x.type := RESULT;
                                    [qS > 0 --> pop(dir)]


                     | qL > 0 ∧
                       qR > 0
                            -->  pop(dir);
                                 dir?y;

                     ]

                | x.offset > 0 --> x.type := RECURS;
                                   pop(dir);
                                   [ dir = L --> R!x;
                                                 push(dir);
                                                 push(R);

                                     | dir = L --> L!x;
                                                   push(dir);
                                                   push(L);
                                   ];
                ]
     | x.type = UNRAVEL -->
              pop(dir);
             *[dir <> U --> dir!x; dir?x;
                            pop(dir);
              ];
     ];
     U!x;
    ]

CONST: *[U?x; U!c] { where c is the constant }
```

References

[1]
Klaus Berkling.
Computer Science Seminar at Caltech.
Talk given by Berkling at Caltech.

[2]
Sally A. Browning.
The Tree Machine.
Technical Report 37590:TR:80, California Institute of Technology, 1980.

[3]
Ashok K. Chandra.
Efficient Compilationof Linear Recursive Programs.
In *Confererence Record of the IEEE 14th Switching and Automata Theory Conference.* Institute of Electrical Engineeers, 1973.

[4]
Haskell B. Curry, Robery Feys.
Combinatory Logic.
North-Holland Publishing Company, Amsterdam, 1968.

[5]
Caxton C. Foster.
Content Addressable Parallel Processors.
Van Nostram Reinhold, New York, 1976.

[6]
Carl Hewitt.
The Apiary Network Architecture for Knowledge Systems.
In *Conference Record of the 1980 LISP Conference.* 1980 LISP Conference, Cambridge, Mass. 02139, August, 1980.

[7]
C.A.R. Hoare.
Communicating Sequential Processes.
Communications of the A.C.M. 21(8), 1978.

[8]
J. Holland.
A Universal Computer Capable of Executing An Arbitrary Number of Sub-Programs Simultaneously.
In *Proceedings of the 1959 Joint Computer Conference.* Massachusetts Institute of Technology, 1959.

[9]
H.T. Kung.
Let's Design Algorithms for VLSI Systems.
In *Caltech Conference on VLSI.* California Institute of Technology, January, 1979.

[10]
C.R. Lang, Jr.
The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture.
Technical Report 4014:TR:82, California Insitute of Technology, 1982.

[11]
Bart N. Locanthi.
The Homogeneous Machine.
Technical Report 3760:TR:80, California Institute of Technology, 1980.

[12]
Martin, A.J.
A Distributed Implementation Method for Parallel Programming.
Internation Federation of Information Processing , 1980.

[13]
Martin, A.J.
The Torus: An Exercise in Constructing a Processing Surface.
In *Second Caltech Conference on VLSI.* California Institute of Technology, 1981.

[14]
Steven S. Muchnick, Neil D. Jones.
A Fixed-Program Machine for Combinator Expression Evaluation.
In *1982 ACM Symposium on LISP and Functional Programming.* Association for Computing Machinery, August, 1982.

[15]
Michael S. Paterson, Carl E. Hewitt.
Comparitive Schematology.
In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation.* Association for Computing Machinery, Cambridge, Mass., June, 1970.

[16]
Charles Lewis Seitz.
Ensemble Architectures for VLSI - A Survey and Taxonomy.
In *MIT Conference on Advance Research in VLSI.* Massachusetts Institute of Technology, January, 1982.

[17]
Philip C. Treleaven, David R. Brownbridge, Richard P. Hopkins.
Data-Driven and Demand-Driven Computer Architectures.
Computing Surveys 14(1), 1982.

[18]
David A. Turner.
A New Implementation Technique for Applicative Languages.
Software-Practices and Experience 9, 1979.